Zmienne i sygnały. Symulacja

Program wykładu

Sygnały i zmienne w VHDL

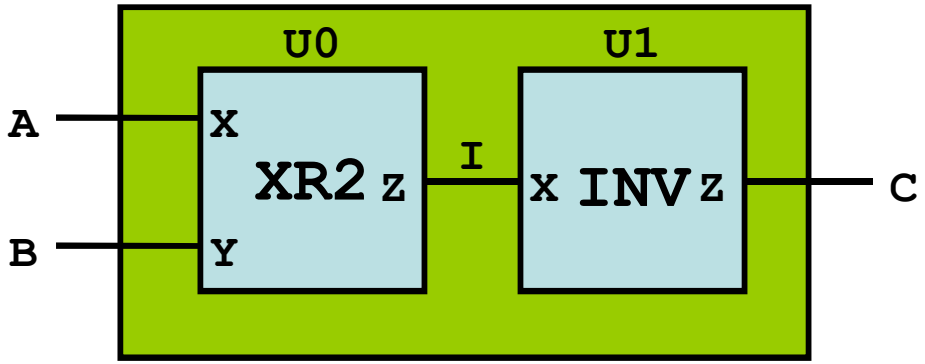
procesy, deklaracje, opóźnienia, hazardy

Symulacja

sygnały a zmienne, cykl Delta-Time,
***sensitivity list*, arbitraż, atrybuty**

Opis strukturalny

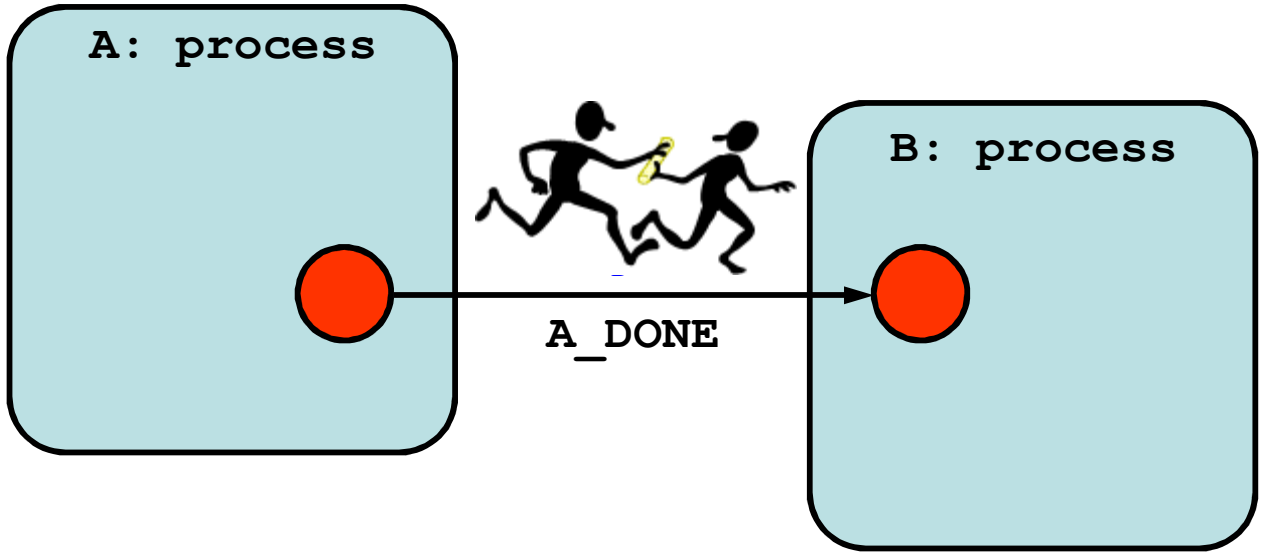
```
entity COMPARE is
  port (A, B: in bit;
        C: out bit);
end COMPARE;
```




```
architecture STRUCTURAL of COMPARE is
signal I: bit; -- sygnał wewnętrzny - brak kierunku!
  component XR2 port (X, Y: in bit; Z: out bit);
  end component;
  component INV port (X: in bit; Z: out bit);
  end component;
begin
  U0: XR2 port map (A, B, I);
  U1: INV port map (I, C);
end STRUCTURAL;
```

Komunikacja między procesami

```
architecture FIRST of ST_UNIT is
signal A_DONE: bit := '0';
begin
.....
```




```
A: process
begin
.....
if S1 then A_DONE <= '1';
```

```
B: process
begin
  wait until A_DONE = '1';
.....
```

Komunikacja między procesami

- Procesy mogą się ze sobą komunikować poprzez nadawanie wartości sygnałom.
- Proces może zawiesić swoje działanie w oczekiwaniu na zmianę w sygnale wejściowym.
- Zmienne zadeklarowane w procesie nie mogą przekazywać swych wartości do innych procesów.
- VHDL'93 definiuje zmienne globalne, które mogą zostać użyte do celów synchronizacji procesów.

Deklaracja sygnałów w VHDL

- Sygnały mogą być zadeklarowane w:
  - ➢ pakietach - sygnały globalne
  - ➢ sekcji deklaracji **`entity`** - sygnały globalne dla **`entity`**
    - porty: **`in`**, **`out`**, **`inout`**, **`buffer`**
    - inne deklaracje
  - ➢ sekcji deklaracji architektury - sygnały lokalne dla architektury
- Sygnały są inicjalizowane przy użyciu operatora :=
- Wartości przypisywane są sygnałom przy użyciu operatora <=

Deklaracja sygnałów w VHDL

Sygnały globalne:

```
package SIGDEC is
  signal VCC: bit := '1';
  signal GROUND: bit := '0';
end SIGDEC;
```

Sygnały globalne dla *entity:*

```
entity BOARD_DESIGN is
  port (DATA_IN: in bit;
        DATA_OUT: out bit);
  signal SYS_CLK: bit := '1';
end BOARD_DESIGN;
```

Sygnały lokalne dla architektury:

```
architecture DATA_FLOW of BOARD_DESIGN is
  signal INT_BUS: bit;
begin
..........
```

Deklaracja sygnałów w VHDL

W deklaracji sygnału jako portu dla **`entity`** należy wyszczególnić: nazwę sygnału, jego kierunek, typ i opcjonalnie wartość początkową.

Składnia:

`port` (*name* [, *more_names*]: *direction type* [:= *expression*] [; *more_ports*]);

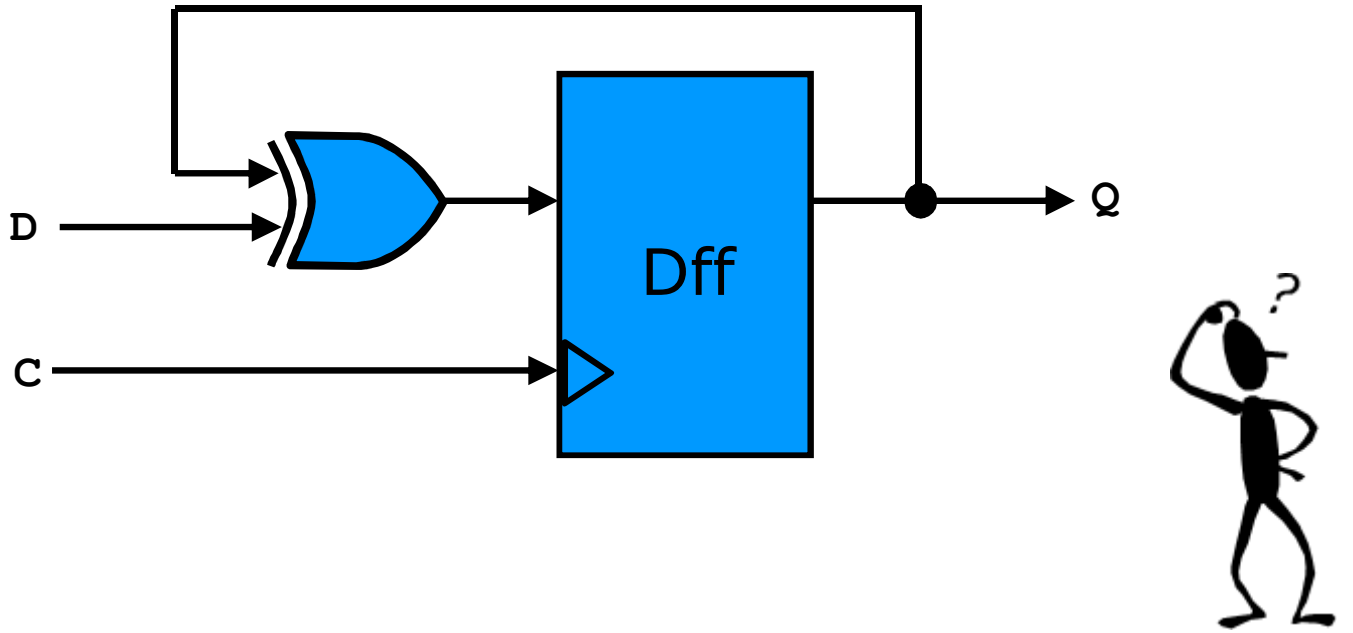
Kierunek	Użycie
`in`	Prawa strona przypisania wartości zmiennej lub sygnałowi
`out`	Lewa strona przypisania wartości sygnałowi
`inout`	Obydwa powyższe
`buffer`	Jak wyżej, ale tylko jedno źródło (w praktyce nieużywany)

Przykład:

```
port (DATA_IN: in bit; DATA_OUT: out bit);
port (B, A: in MyLib.MyPkg.MyType);
```

Sprzężenia zwrotne

Pytanie: Jaki jest najprawdopodobniej kierunek sygnału Q ?



Sprzężenia zwrotne

Reguły połączeń między modułami wymagają odpowiadających sobie trybów portów, np.: `buffer` ⇔ `buffer`. Ale to jest niepraktyczne...

```
entity ...
port(D: in ...
     C: in ...
     Q: buffer ... );
end entity;

architecture wrong
begin
 process(C)
 begin
  if C and C'event then
   Q <= Q xor D;
  end if;
 end process;
end wrong;
```

```
entity ...
port(D: in ...
     C: in ...
     Q: out ... ); -- OUT direction
end entity;

architecture good
 signal: T ...      -- T aux signal
begin
 process(C)
 begin
  if C and C'event then
   T <= T xor D;    -- T usage
  end if;
 end process;
 Q <= T;        -- concurrent assignment
end good;
```

Opóźnienie transportowe

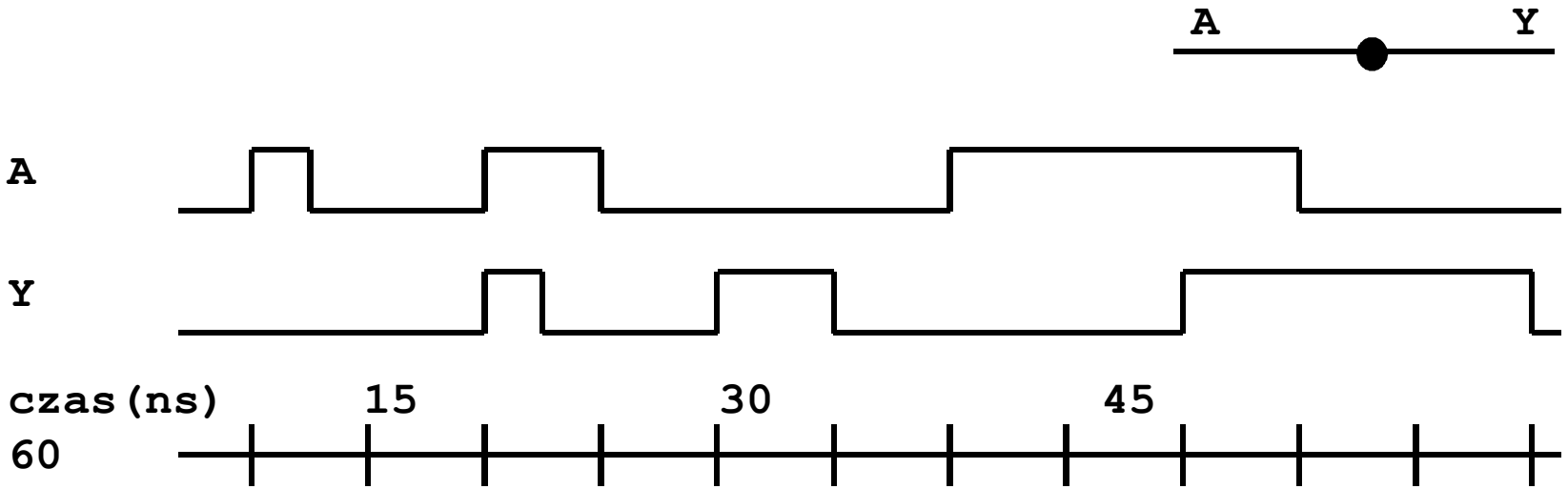
Modelowanie połączeń dokonywane jest przy pomocy opóźnienia transportowego. Propagowane są wówczas wszystkie zmiany wartości sygnałów, niezależnie od czasu ich trwania.

Składnia:

signal <= `transport` *expression* `after` *transport-delay*;

Przykład:

```
Y <= transport A after 10 ns;
```



Opóźnienie inercyjne

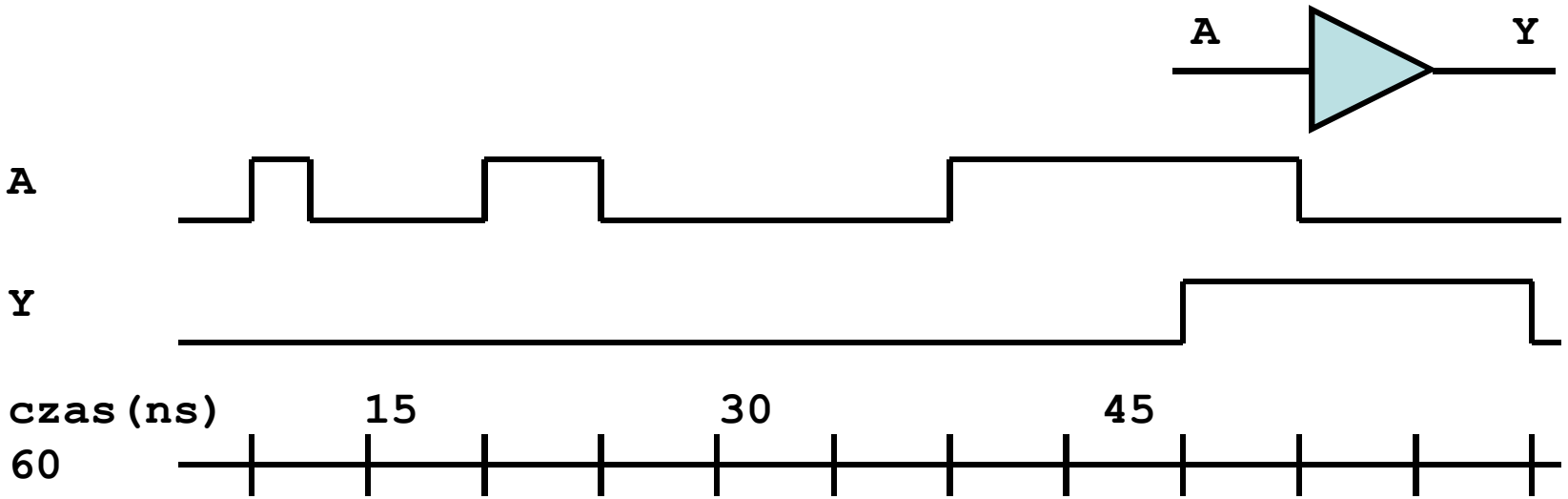
Modelowanie elementów dokonywane jest przy pomocy opóźnienia inercyjnego (domyślne). Propagowane są wówczas tylko te zmiany wartości sygnałów, które trwają nie krócej niż wartość opóźnienia.

Składnia:

signal <= [`inertial`] *expression* `after` *inertial-delay* ;

Przykład:

```
Y <= A after 10 ns;
```



Opóźnienie inercyjne

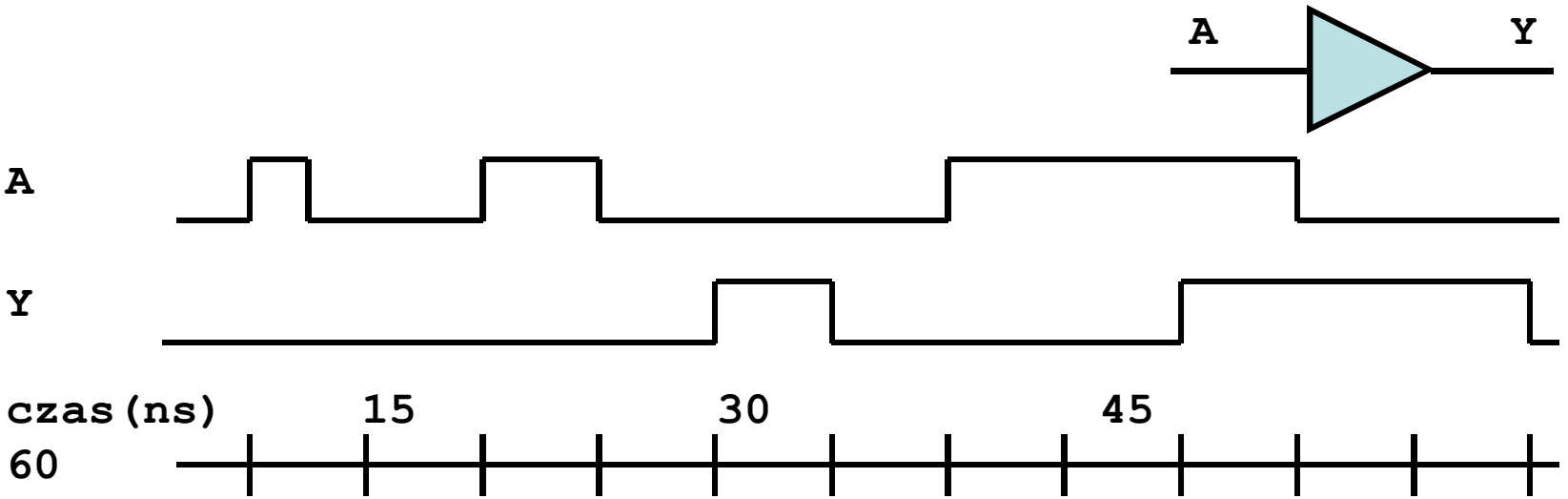
VHDL-93 pozwala na modelowanie elementów, które reagują na impulsy krótsze niż opóźnienie tych elementów.

Składnia:

signal <= `reject` *reject-delay* `inertial` *expression* `after` *inertial-delay* ;

Przykład:

```
Y <= reject 4 ns inertial A after 10 ns;
```



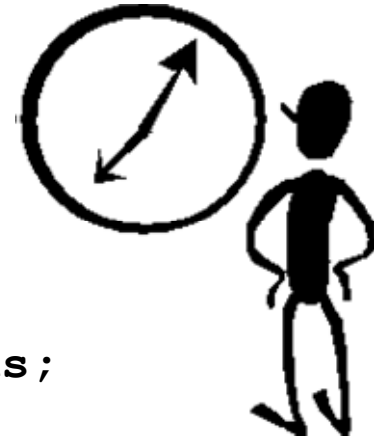
Opóźnienia w procesie

Nadawanie wartości sygnałom jest **sekwencyjne** w obrębie procesów, a **współbieżne** na zewnątrz nich.

W obrębie procesu nadawanie wartości sygnałom jest wstrzymywane do momentu uruchomienia cyklu symulacji, wyzwalanego przez wykonanie instrukcji `wait`.

Przykład:

```
process
begin
    sys_clk <= not (sys_clk) after 50 ns;
    int_bus <= data_in after 10 ns;
    data_out <= my_function (int_bus) after 10 ns;
    wait .....
end process;
```



Opóźnienia sygnałów

Instrukcje przypisania wartości sygnałom mogą zawierać kilka wartości dla różnych momentów czasowych. Własność ta jest użyteczna przy opisywaniu sygnałów zegarowych i innych przebiegów powtarzalnych.

Przykład:

```
S <= '1' after 4 ns, '0' after 7 ns;
T <= 1 after 1 ns, 3 after 2 ns, 6 after 8 ns;
```

Wewnątrz procesu sygnał powinien mieć tylko jedno źródło w danym czasie. W przeciwnym wypadku pod uwagę brane jest jedynie ostatnie przypisanie.

Przykład:

```
process
begin
  xyz <= 1 after 5 ns;    -- niezależnie od czasu...
  xyz <= 2 after 4 ns;    -- ...zajdzie tylko to !...
  pqr <= 10 after 5 ns;   -- ...no i oczywiście to.
wait .....
```

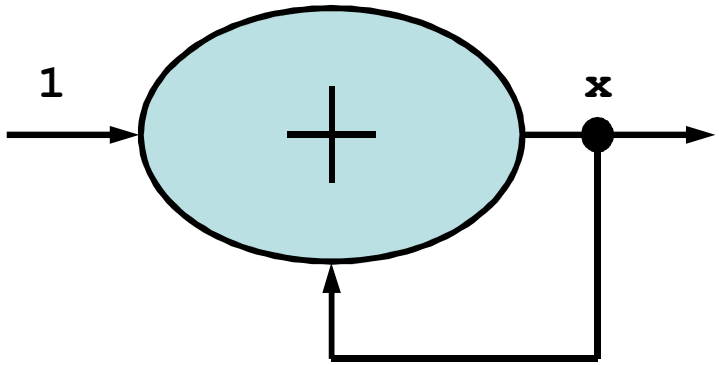
Opóźnienia sygnałów

```
signal X, Y: integer;
process
begin
  wait on Y;
  X <= Y + 1 after 10 ns;
```

W powyższym przykładzie X przyjmuje nową wartość dokładnie po 10 ns (nie 9.9999 lub 10.0001 ns).

Deklaracja opóźnień jest ignorowana przez narzędzia do syntezy.

```
signal X: integer;
process .....
begin
.....
X <= X + 1 after 10 ns;
.....
```



Opóźnienie zerowe

Przykład:

```
entity VAR is
port (A: in bit_vector (0 to 7);
      INDEX: in integer range 0 to 7;
      OUTPUT: out bit);
end VAR;

architecture VHDL_1 of VAR is
begin
  process
  begin
    OUTPUT <= A(INDEX); -- opóźnienie 0 ns
    wait .....;         -- wait inicjuje przypisanie
.....
end VHDL_1;
```

Zmienne a sygnały

Główne różnice między przypisywaniem wartości zmiennym i sygnałom:

Przypisywanie wartości sygnałom	Przypisywanie wartości zmiennym
• według reżimów czasowych • z uwzględnieniem opóźnień • po spełnieniu warunku w `wait`	• bez reżimów czasowych • bez opóźnień • natychmiastowe

Przykłady:

```
X <= 1;          --opóźnienie 0
wait .....;      --przypisanie zachodzi po wykonaniu wait

X <= Y;          --opóźnienie 0 - zamiana
Y <= X;
wait .....;      --oba przypisania zachodzą po wykonaniu wait

V := 1;         --przypisanie do zmiennej następuje natychmiast
S <= V;
A := S;          --A otrzymuje poprzednią wartość S
wait .....;      --S otrzymuje wartość V (=1) po wykonaniu wait

X <= 1;
X <= 2;
wait for 0 ns; -- po wykonaniu wait X otrzymuje wartość 2
```

Hazardy

```
entity MUX is
 port (Ain, Bin: in bit;
       SEL: in boolean;
       Y: out bit);
end MUX;

architecture WRONG of MUX is
 signal MUXVAL: integer range 0 to 1;
begin
 process
 begin
  MUXVAL <= 0;
  if (SEL) then
   MUXVAL <= MUXVAL + 1;
  end if;
  case MUXVAL is
   when 0 => Y <= Ain after 10 ns;
   when 1 => Y <= Bin after 10 ns;
  end case;
  wait on Ain, Bin, SEL;
 end process;
end WRONG;
```

Hazardy

```
entity MUX is
 port (Ain, Bin: in bit;
       SEL: in boolean;
       Y: out bit);
end MUX;

architecture BETTER of MUX is
begin
 process
  variable MUXVAL: integer range 0 to 1;   -- zmienna !
 begin
  MUXVAL := 0;                             -- zmienna !
  if (SEL) then
   MUXVAL := MUXVAL + 1;                   -- zmienna !
  end if;
  case MUXVAL is                           -- zmienna !
   when 0 => Y <= Ain after 10 ns;
   when 1 => Y <= Bin after 10 ns;
  end case;
  wait on Ain, Bin, SEL;
 end process;
end BETTER;
```

Sygnały a zmienne

```
architecture sig of counter is
signal SIG: ...
begin
process(CLK)
begin
  if CLK and CLK'event then
    SIG <= SIG + 1;
    if SIG = 9 then
      SIG <= 0;
    end if;
  end if;
end process;
```

```
architecture var of counter is
begin
process(CLK)
  variable VAR: ...
  begin
  if CLK and CLK'event then
    VAR := VAR + 1;
    if VAR = 9 then
      VAR := 0;
    end if;
  end if;
end process;
```

Pytanie:
Jak będą liczyły powyższe liczniki ?

Reguła: Czy nowo przypisana wartość musi być używana w tym samym przebiegu pętli symulatora? Jeśli tak – należy użyć zmiennej.
W innych przypadkach należy używać sygnałów (wolniej się symulują ☹).

Cykl symulacji

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

Przykład (w procesie):

```
A <= B;
B <= A;
```

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.

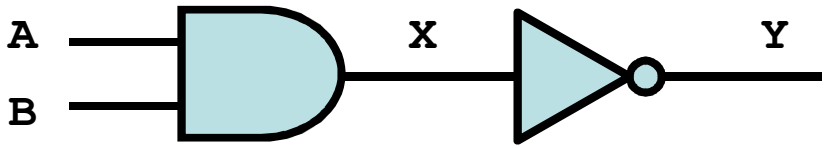
Cykl symulacji

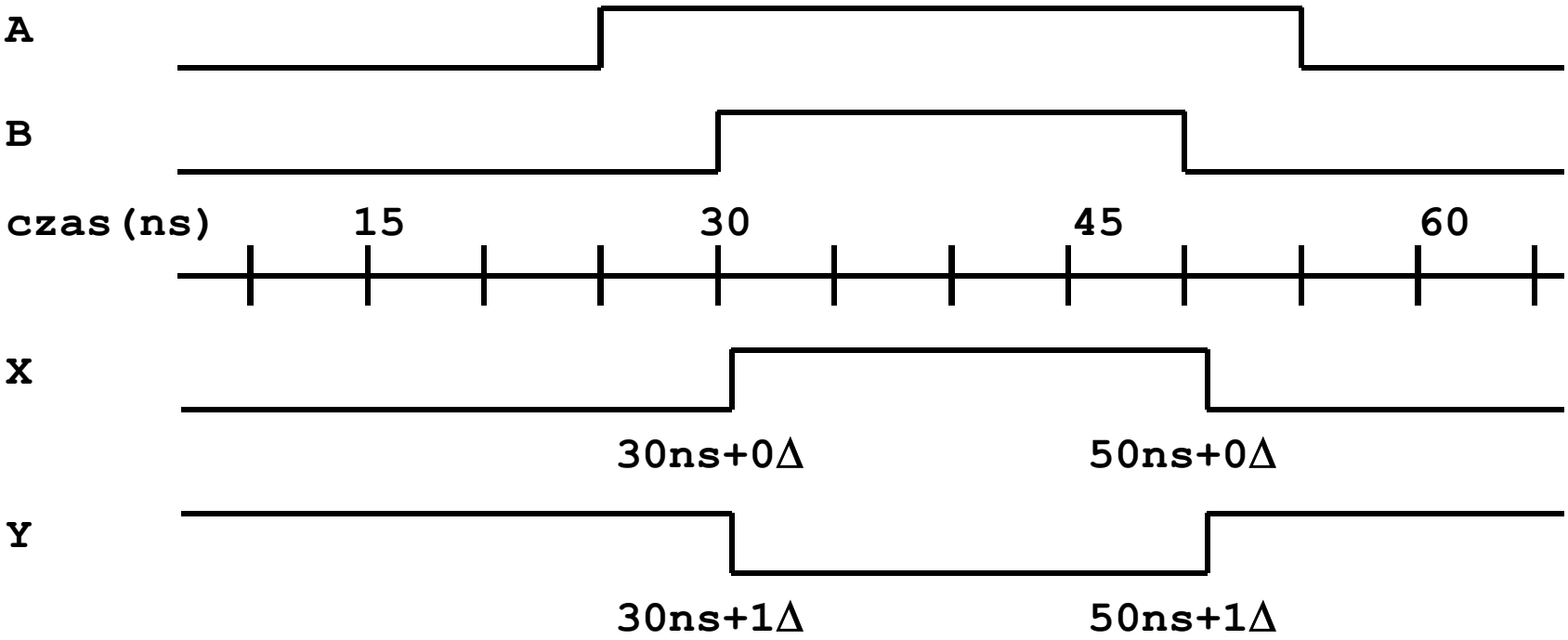
Cykl symulacji

Symulacja przypisań współbieżnych z użyciem cyklu Δ.

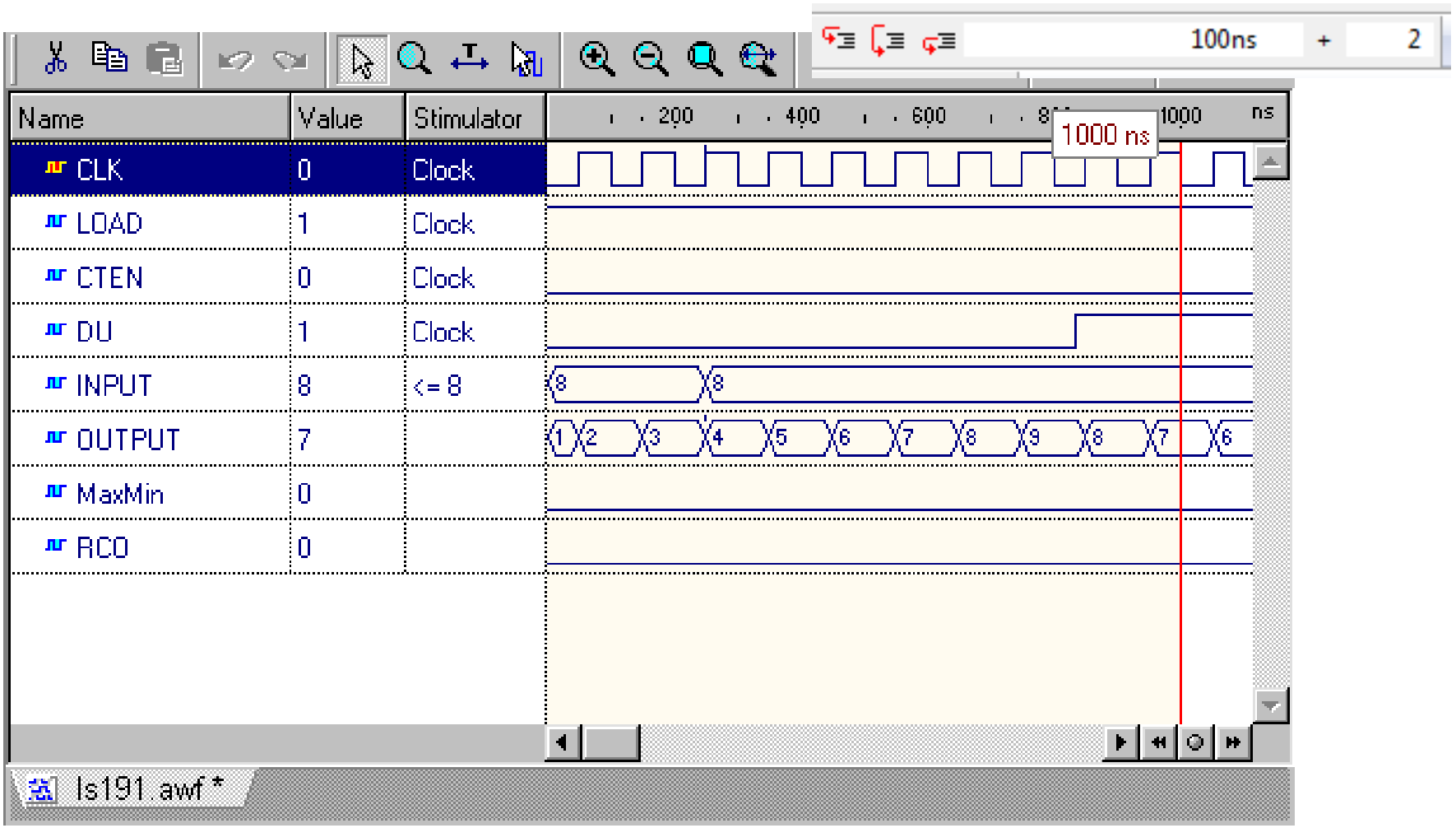
Przykład:

```
X <= A and B;
Y <= not X;
```

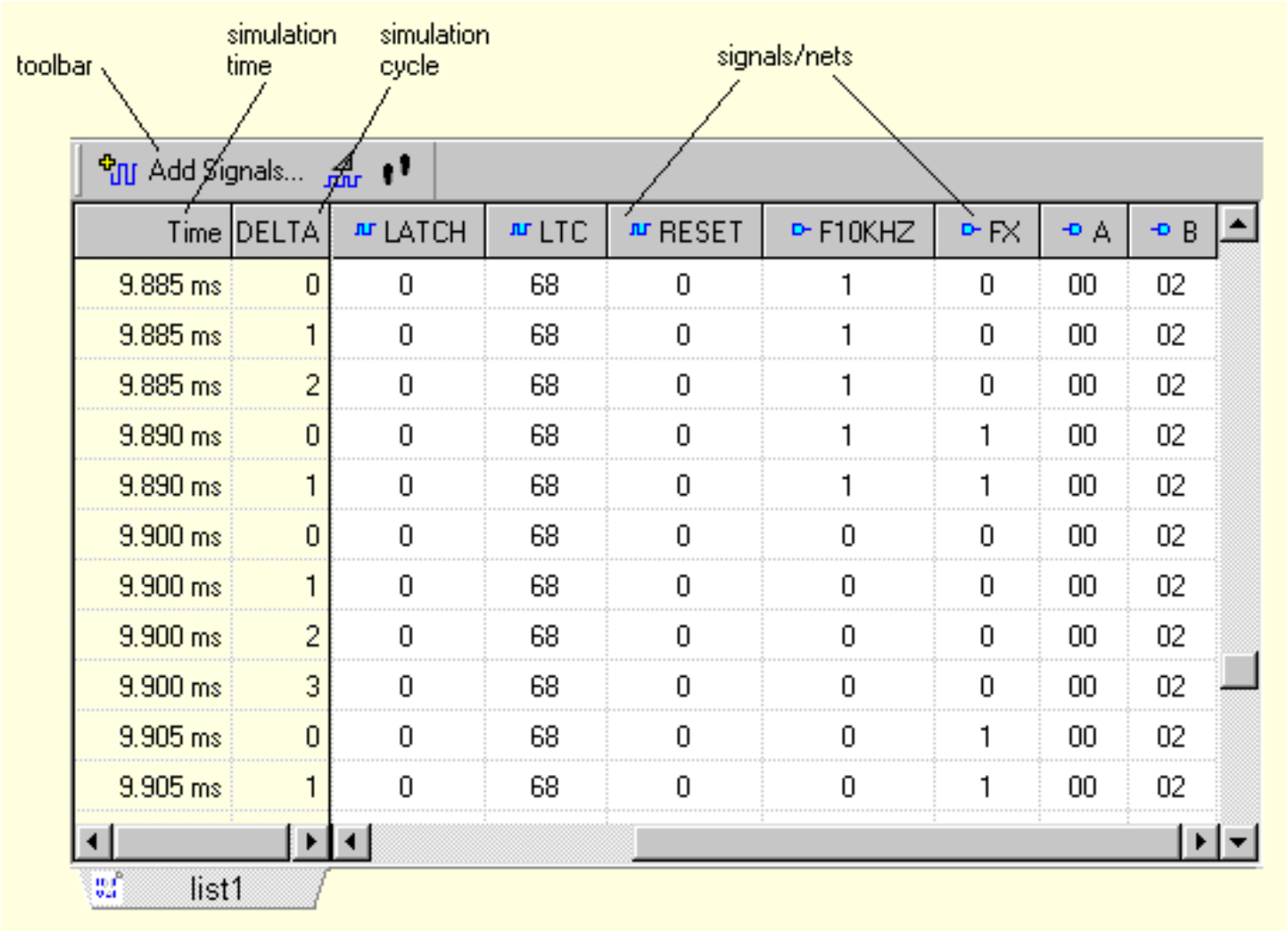




ActiveHDL – *Waveform Viewer*



ActiveHDL – *List Viewer*



Cykl symulacji

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

```
process
begin
   X <= 1;
   X <= 2;
   A <= X;
   X <= 3;
wait for 0 ns;
```

Instrukcja `wait`

Rodzaje instrukcji `wait`:

- `wait on A, B;`
  Zawiesza wykonywanie do momentu pojawienia się zdarzenia na A lub B.

- `wait until A > 10;`
  Zawiesza wykonywanie do momentu pojawienia się zdarzenia na A i spełnienia warunku A > 10.

- `wait for 10 ns;`
  Zawiesza wykonywanie procesu na 10 ns.

- `wait;`
  Zawiesza wykonywanie procesu na zawsze. Używana jako ‘kill’.

Symulacja i instrukcja `wait`

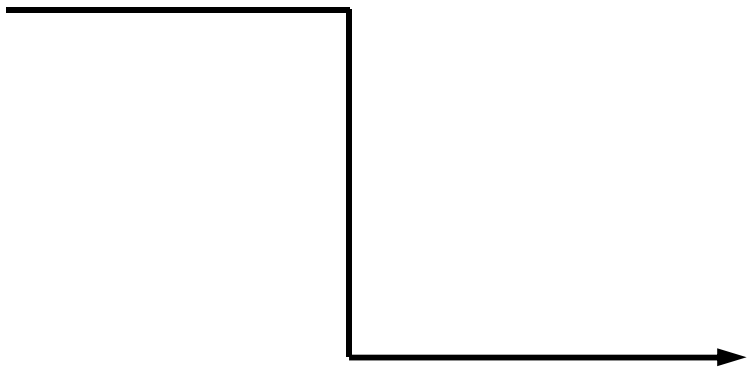
Zamiast instrukcji **`wait`** wewnątrz procesu można wyspecyfikować listę sygnałów, które aktywują dany proces (ang. *sensitivity list*). Sygnały te podaje się w nawiasie po słowie kluczowym **`process`**.

Jest to równoważne instrukcji **`wait`** występującej **na końcu** procesu.

Proces może posiadać **albo** listę **albo** taką instrukcję **`wait`**.

Przykład:

```
process (CLK)
begin
.....
.....
<statements>
.....
.....
end process;
```



```
process
begin
..........
..........
<statements>
..........
wait on CLK;
end process;
```

Pytanie:

Jaka jest różnica w zachowaniu się dwóch poniższych procesów ?

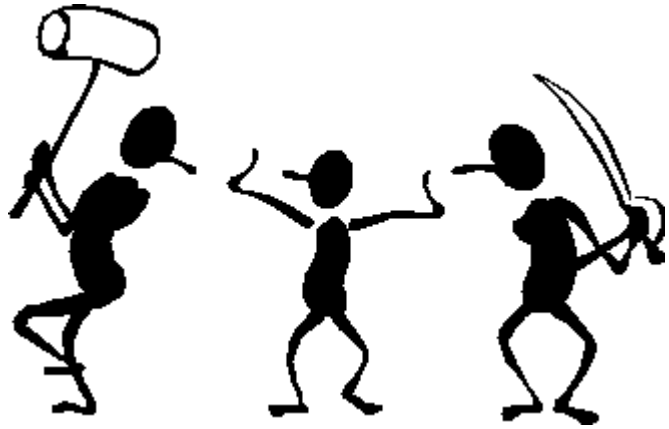
```
process (A, B)
begin
S <= A;
T <= B;
V <= S or T;
end process;
```

```
process (A, B, S, T)
begin
S <= A;
T <= B;
V <= S or T;
end process;
```

Funkcja arbitrażu

VHDL zezwala na sterowanie sygnałów z wielu źródeł, ale:

- każde ze źródeł musi być w osobnym procesie albo w różnych przypisaniach współbieżnych,
- dla takich sygnałów musi być zadeklarowana funkcja arbitrażu (*resolution function*),
- funkcja taka jest predeklarowana dla typu STD_LOGIC. Jest to preferowany typ obiektów (łatwiej używać jednego typu w całym projekcie), ale nie pozwoli na wykrycie podczas kompilacji omyłkowego połączenia dwóch *driverów* (możliwe dopiero podczas symulacji).

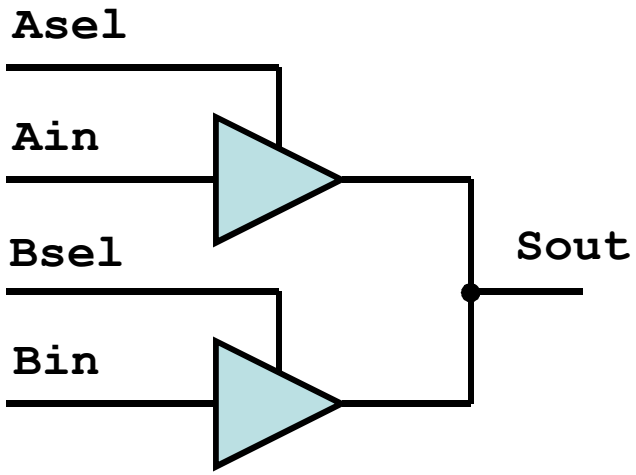


Funkcja arbitrażu

```
architecture SEQUENTIAL of TRISTATE is
 signal Ain,Bin,Asel,Bsel,Sout: STD_LOGIC;
begin

A:process (Ain,Asel)
  begin
   Sout <= 'Z';
   if (Asel='1') then
    Sout <= Ain;
   end if;
  end process;

B:process (Bin,Bsel)
  begin
   Sout <= 'Z';
   if (Bsel='1') then
    Sout <= Bin;
   end if;
  end process;
end SEQUENTIAL;
```




```
architecture CONCURRENT of TRISTATE is
signal Ain,Bin,Asel,Bsel,Sout: STD_LOGIC;
begin
 Sout <= Ain when Asel = '1' else 'Z';
 Sout <= Bin when Bsel = '1' else 'Z';
end CONCURRENT;
```

Funkcja arbitrażu – 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 – 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;
```

Funkcja arbitrażu

Funkcja:

- zwraca jedną wartość,
- ma wszystkie argumenty w trybie *input*,
- przekazuje argumenty przez wartość.

Funkcja arbitrażu:

- jest wymagana, gdy sygnał (węzeł) jest sterowany przez więcej niż jedno źródło,
- przeprowadza arbitraż sygnałów,
- jest wywoływana w przypadku zmian w każdym ze źródeł sygnału,
- otrzymuje tablicę sygnałów do arbitrażu,
- jest definiowana przez użytkownika,
- jest skojarzona z podtypem.

Atrybuty czasowe sygnałów

- *signal_name*`event - zwraca wartość TRUE jeżeli w danym kroku symulacji wystąpiło zdarzenie (event)
- *signal_name*`last_event - zwraca czas od ostatniego zdarzenia
- *signal_name*`last_value - zwraca poprzednią wartość sygnału (przed ostatnim zdarzeniem)

Przykłady:

```
if CLK'event and CLK='1' then ...

if SD_DAT'event and (SD_DAT='H' or SD_DAT='Z') and
   SD_DAT'last_value='1' then ...
```

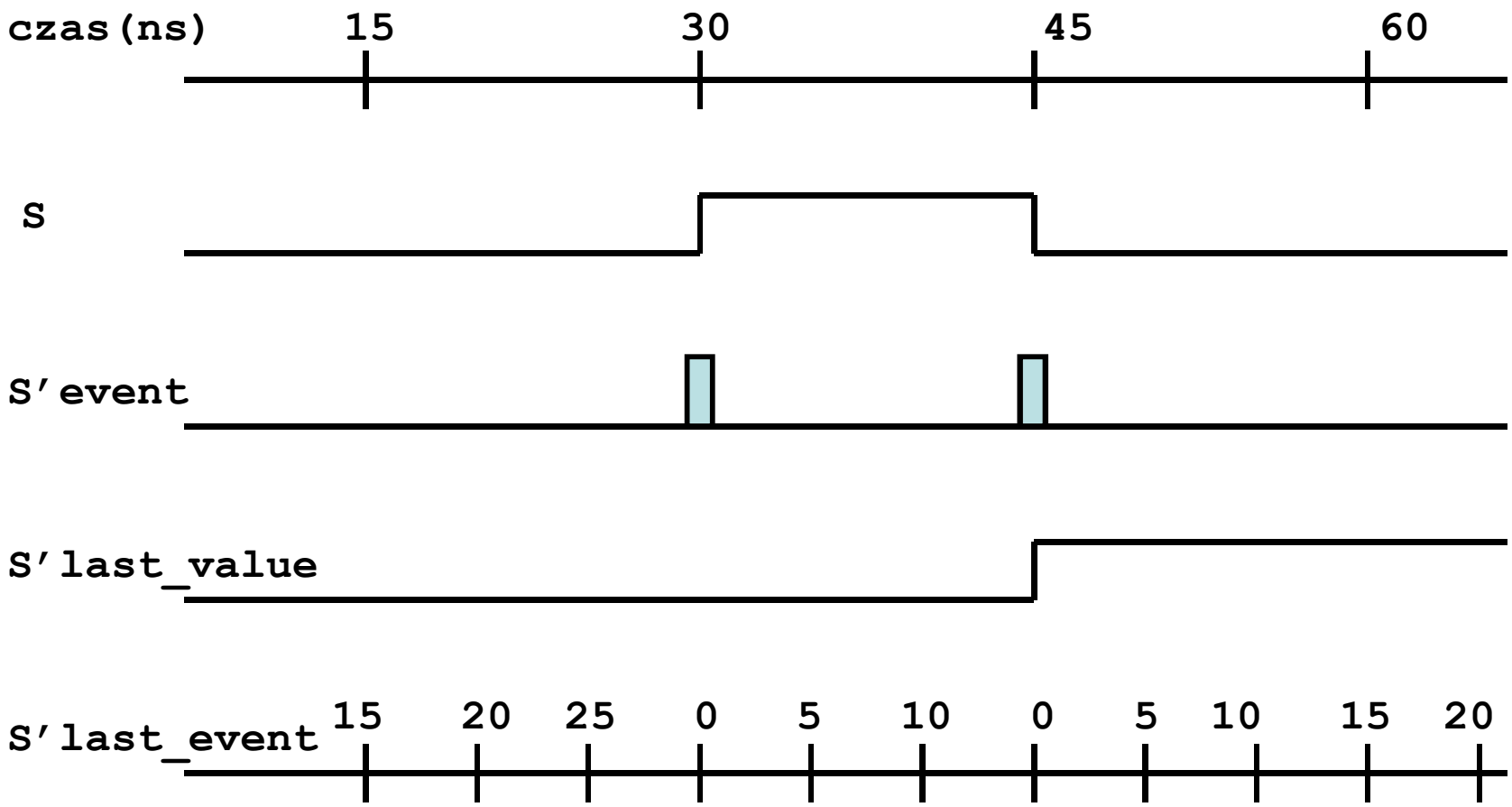
Zdefiniowane funkcje `rising_edge` i `falling_edge`:

```
if rising_edge(CLK) then ...
```

równoważne zapisowi:

```
if CLK'event and CLK='1' and CLK'last_value='0' then ...
```

Atrybuty czasowe sygnałów



Atrybuty predefiniowane

Atrybuty predefiniowane pozwalają uzyskać informacje o obiektach, typach, podprogramach itp.

Atrybuty sygnałów:

Attribute	Result
S'Delayed(t)	implicit signal, equivalent to signal S, but delayed t units of time
S'Stable(t)	implicit signal that has the value True when no event has occurred on S for t time units, False otherwise
S'Quiet(t)	implicit signal that has the value True when no transaction has occurred on S for t time units, False otherwise
S'Transaction	implicit signal of type Bit whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active)
S'Event	True if an event has occurred on S in the current simulation cycle, False otherwise
S'Active	True if a transaction has occurred on S in the current simulation cycle, False otherwise
S'Last_event	the amount of time since last event occurred on S, if no event has yet occurred it returns Time'High
S'Last_active	the amount of time since last transaction occurred on S, if no event has yet occurred it returns Time'High
S'Last_value	the previous value of S before last event occurred on it
S'Driving	True if the process is driving S or every element of a composite S, or False if the current value of the driver for S or any element of S in the process is determined by the null transaction
S'Driving_value	the current value of the driver for S in the process containing the assignment statement to S

Atrybuty predefiniowane

Atrybuty typów skalarnych:

Attribute	Result type	Result
T'Left	same as T	leftmost value of T
T'Right	same as T	rightmost value of T
T'Low	same as T	least value in T
T'High	same as T	greatest value in T
T'Ascending	boolean	true if T is an ascending range, false otherwise
T'Image(x)	string	a textual representation of the value x of type T
T'Value(s)	base type of T	value in T represented by the string s

Atrybuty typów i podtypów dyskretnych i fizycznych:

Attribute	Result type	Result
T'Pos(s)	universal integer	position number of s in T
T'Val(x)	base type of T	value at position x in T (x is integer)
T'Succ(s)	base type of T	value at position one greater than s in T
T'Pred(s)	base type of T	value at position one less than s in T
T'Leftof(s)	base type of T	value at position one to the left of s in T
T'Rightof(s)	base type of T	value at position one to the right of s in T

Atrybuty typów tablicowych i obiektów typów tablicowych:

Attribute	Result
A'Left(n)	leftmost value in index range of dimension n
A'Right(n)	rightmost value in index range of dimension n
A'Low(n)	lower bound of index range of dimension n
A'High(n)	upper bound of index range of dimension n
A'Range(n)	index range of dimension n
A'Reverse_range(n)	reversed index range of dimension n
A'Length (n)	number of values in the n-th index range
A'Ascending(n)	True if index range of dimension n is ascending, False otherwise

Ciąg dalszy nastąpi…