*Лекция 4*

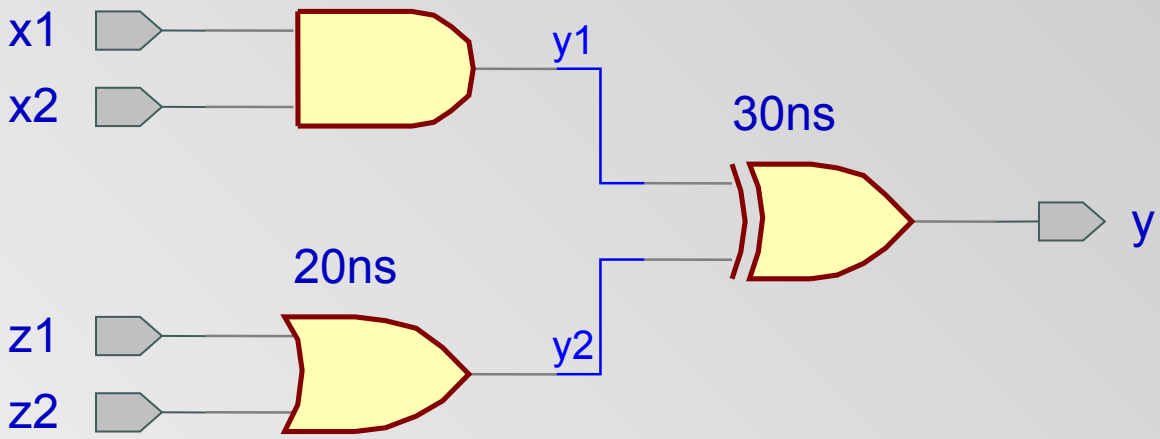
# МОДЕЛИРОВАНИЕ СИГНАЛОВ В VHDL

## Механизм процессов и концепция сигнала В VHDL

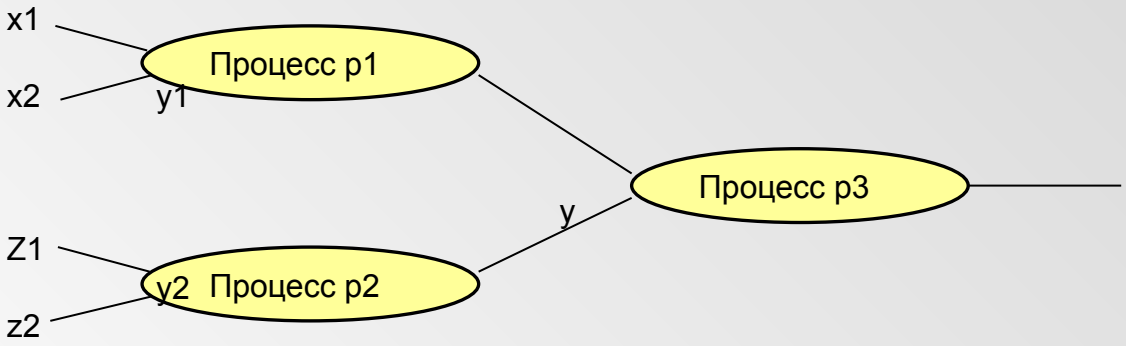
При программировании на VHDL нужно учитывать две особенности:

- моделирование параллельных процессов;
- моделирование задержек сигналах.

Пример. Схема состоящая из трех элементов: 2 И, 2 ИЛИ и 2 Исключающее ИЛИ.




Тогда данную схему можно представить как взаимодействие трех процессов



Процессы p1,p2,p3 описываются в VHDL следующим образом:

```
P1: process (x1,x2)
begin
. . . функционирование элемента 2И
end process p1;
P2: process (z1,z2)
begin
. . . — функционирование элемента 2ИЛИ
end process p2;
P3: process (y1,y2)
begin
. . . — функционирование элемента 2ИсключающееИЛИ
end process p3;
```

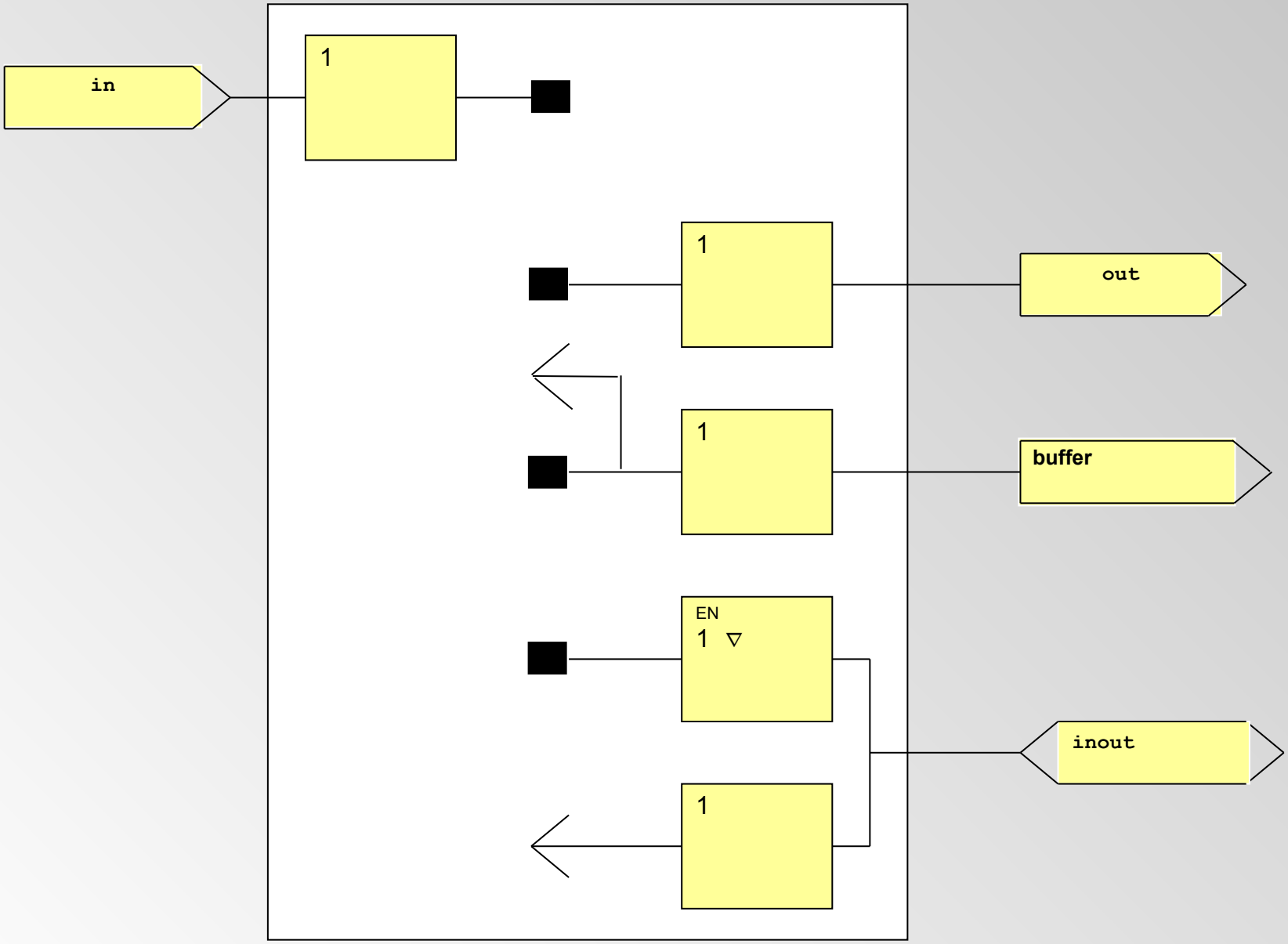
В VHDL есть два **типа сигналов**:

**ports -** порты - это внешние входы и выходы ОП:

**signals** **-** внутренние ( локальные) сигналы - выходы логических элементов.

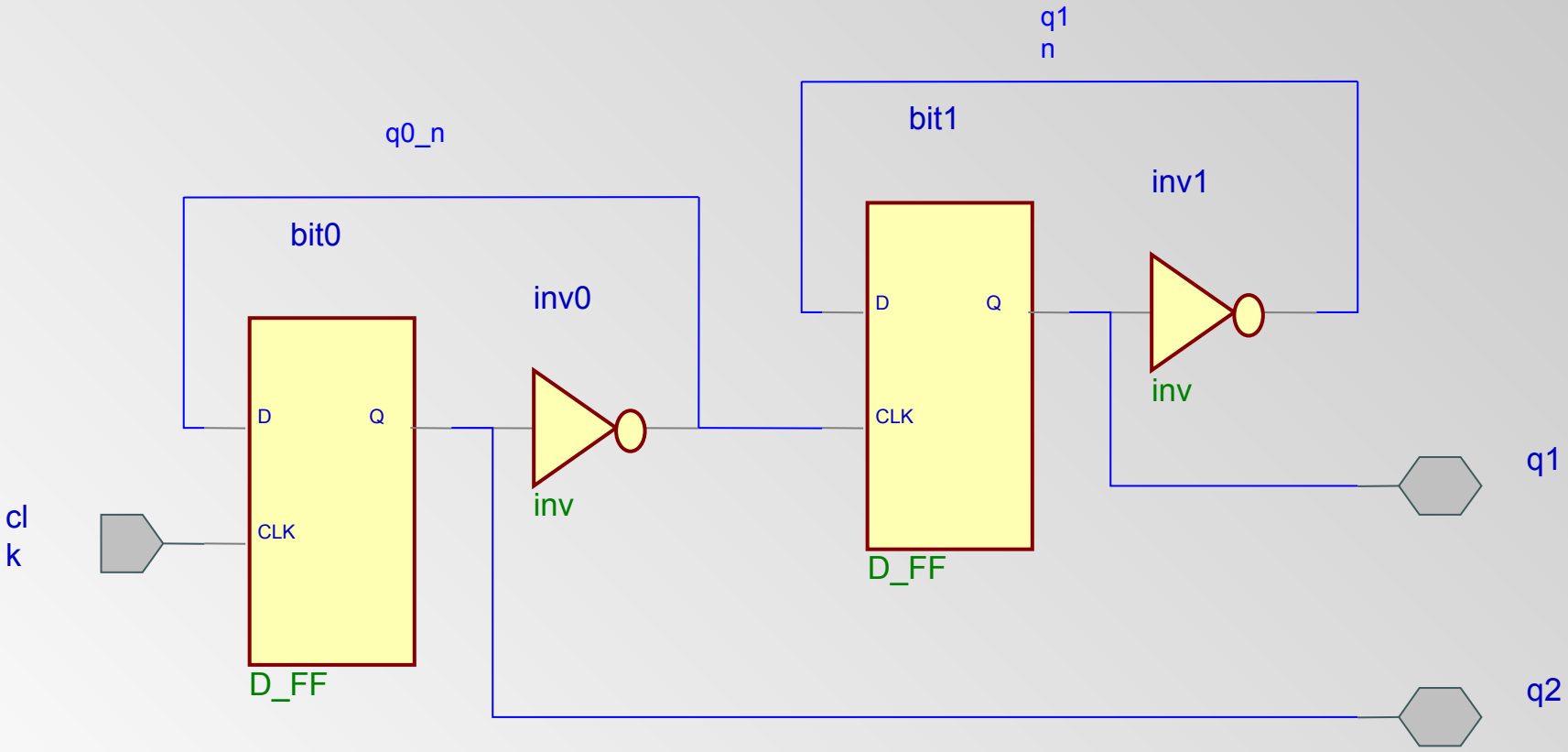
Порты описывают в интерфейсной части (entity) проекта.

port_declaration ::=
 **port** (port_list);
port_list ::= identifier{,...}: [mode] subtype_indication
 [:= expression]
mode::= **in | out | inout | buffer | linkage**


in
1
1
out
1
buffer
EN
1
1

**Декларация внутреннего сигнала**

signal_declaration ::=
***signal*** *identifier{,...}: subtype_indication [:= expression]*
**architecture** arch_name **of** entity_name **is**
{ signal_declaration}
{other_declarative_item}
**begin**
{concurrent_statement}
end [**architecture]** [arch_name];

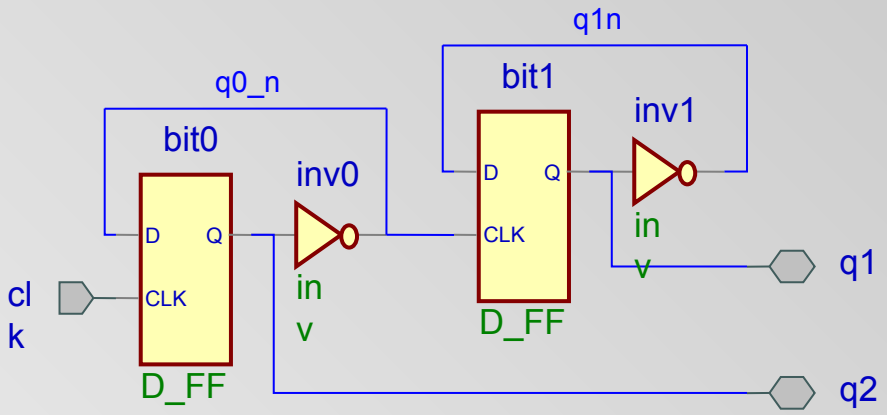

q1
n
bit1
q0_n
inv1
bit0
inv0
D
Q
D
Q
CLK
inv
inv
CLK
clk
D_FF
D_FF
q1
q2

```
entity D_flipflop is
  port ( clk, d : in bit;  q : buffer bit );
end entity D_flipflop;

architecture behavioral of D_flipflop is
begin
  q <= d when clk'event and clk = '1';
end architecture behavioral;

entity inverter is
  port ( a : in bit;  y : out bit );
end entity inverter;

architecture behavioral of inverter is
begin
  y <= not a;
end architecture behavioral;
```



```
entity count2 is
  port ( clk : in bit;  q0, q1 : buffer bit );
end entity count2;

architecture buffered_outputs of count2 is

  component D_flipflop is
    port ( clk, d : in bit;  q : buffer bit );
  end component D_flipflop;

  component inverter is
    port ( a : in bit;  y : out bit );
  end component inverter;

  signal q0_n, q1_n : bit;

begin

  bit0 : component D_flipflop
    port map ( clk => clk, d => q0_n, q => q0 );

  inv0 : component inverter
    port map ( a => q0, y => q0_n );

  bit1 : component D_flipflop
    port map ( clk => q0_n, d => q1_n, q => q1 );

  inv1 : component inverter
    port map ( a => q1, y => q1_n );

end architecture buffered_outputs;
```

**Простой оператор назначения сигналов *( Simple SA)***

*Simple SA - simple_signal_assignment( Simple SA)*-
простой оператор назначения сигнала;

```
simple_SA::=
  name<=[transport] waveform_element;
    waveform_element::=
    [label:] value_expression [after time_expression]
          |null [after time_expression];
```

**Разновидности** *(Simple SA):*

- *concurrent Simple SA* (CSA) – параллельный простой оператор назначения сигнала;
- *siquential Simple SA* (SSA) – последовательный простой оператор назначения сигнала.

CSA и SSA синтаксически неразличимы – их вид определяется областью действий

**Область действий CSA:**

- архитектура
- блок

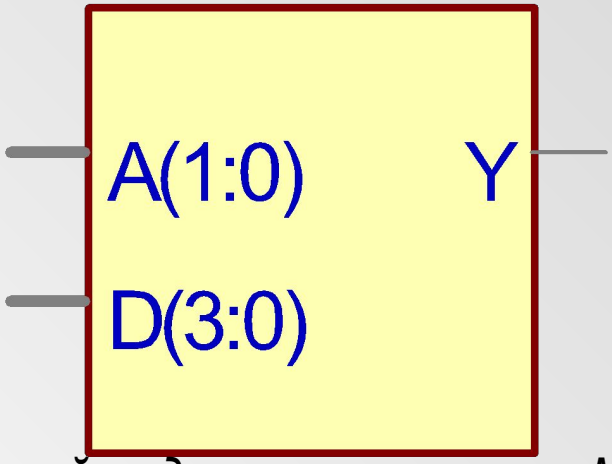
**architecture** arch_name **of** entity_name **is**
{ signal_declaration}
{other_declarative_item}
**begin**
{CSA}  -- *CSA в архитектуре*
{other_concurrent_statement}
    { **block** [(…)]
{ signal_declaration}
{other_declarative_item}
**begin**
{CSA} -- *CSA в блоке*
{other_concurrent_statement}
**end block}**
end [**architecture]** [arch_name];

**Область действий SSA:**

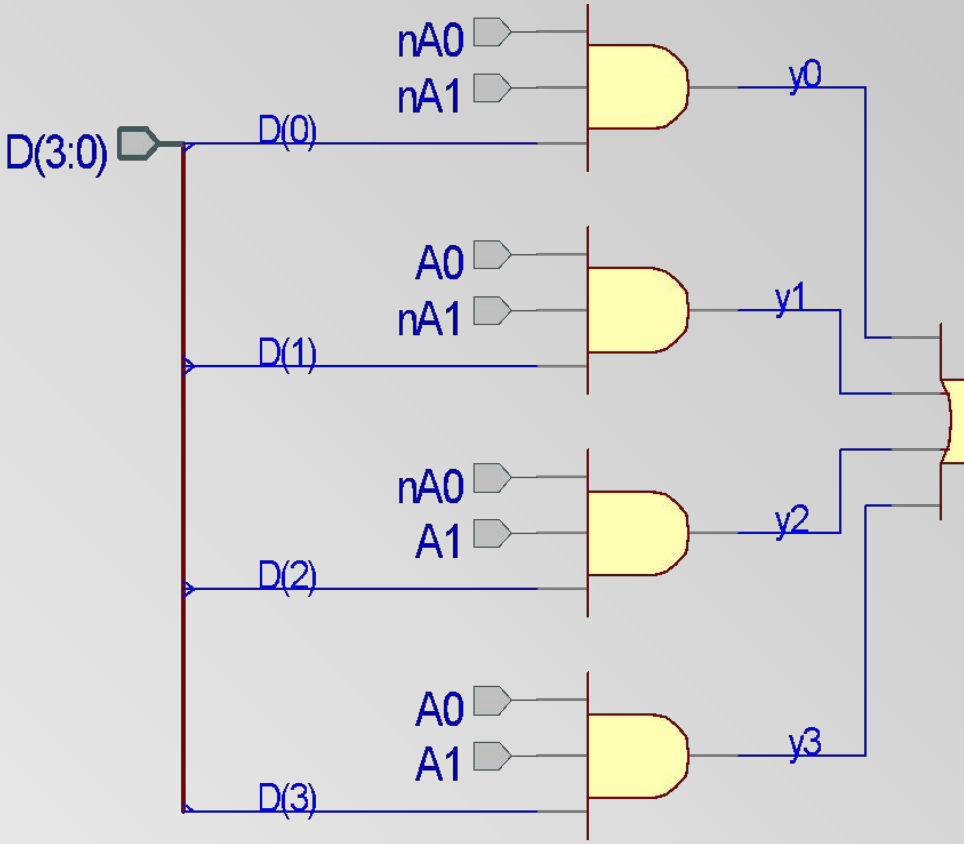
- процесс
- процедура
- функция

**architecture** arch_name **of** entity_name **is**
{ signal_declaration}
{other_declarative_item}
{ **function** function_name**(**parameters**) return** type_mark **is**
{declarative_item}
**begin**
{SSA} -- *SSA в функции*
{other_ siquential _statement}
**end function** function_name;
{ **procedure procedure_name(**parameters**) is**
{declarative_item}
**begin**
{SSA} -- *SSA в процедуре*
{other_ siquential _statement}
**end procedure** procedure_name;
**begin**
{CSA} -- *CSA в architecture*
{other_concurrent_statement}
{ **block** [(…)]
{ signal_declaration}
{other_declarative_item}
**begin**
{CSA}-- *CSA в block*
{other_concurrent_statement}
**end block}**
end [**architecture]** [arch_name];

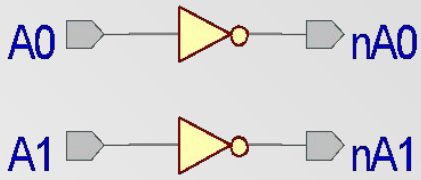
Пример описания работы мультиплексора потоком сигналов.




*Внешний вид мультиплексора MUX4*




*Логическая схема мультиплексора MUX4*



```
entity mux4 is
  generic(delay:time:=20ps);
  port(D:in std_logic_vector(3 downto 0);
       A:in std_logic_vector(1 downto 0);
       Y:out std_logic);
end mux4;
architecture data_flow_simple_SA of mux4 is
  signal      nA0, nA1, y0, y1, y2, y3, z :std_logic;
    begin       -- first level of signal flow
      nA0<=not A(0);
      nA1<=not A(1);
      -- second level of signal flow
      y0<=nA0  and nA1  and D(0);
      y1<=A(0) and nA1  and D(1);
      y2<=nA0  and A(1) and D(2);
      y3<=A(0) and A(1) and D(3);
      -- third level of signal flow
      z<=y0 or y1 or y2 or y3;
      Y<=z after delay
    end data_flow_simple_SA;
```

**Драйвер сигнала**

Каждый сигнал имеет один или несколько так называемых драйверов. Драйвер содержит текущее значение сигнала и набор планируемых значений. Его можно описать последовательностью пар {time/value - время/значение}, которые устанавливаются в момент назначения сигнала.

Например, на 0 ns драйвер сигнала *y* (*Dr_y*) типа *integer* после выполнения оператора

y<= 0 **after** 0 fs, 1 **after** 1 ps, 2 **after** 2 ps, 3 **after** 3 ns, 4 **after** 4 us,
5 **after** 5 ms, 6 **after** 6 sec, 7 **after** 7 min, 8 **after** 8 hour;

можно представить текущим состоянием

| *time* | *value* |
|---|---|
| 0 fs | 0 |

и списком последующих (планируемых, ожидаемых (scheduling, pending) следующим списком:

| *time* | *value* |
|---|---|
| 0 fs | 0 |

| *time* | *value* |
|---|---|
| 1 ps | 1 |

| *time* | *value* |
|---|---|
| 2 ps | 2 |

| *time* | *value* |
|---|---|
| 3 ns | 3 |

| *time* | *value* |
|---|---|
| 4 us | 4 |

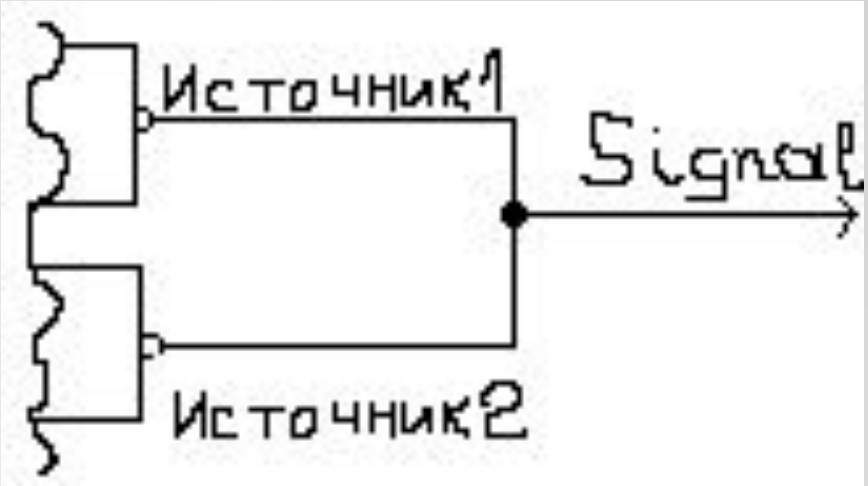
| *time* | *value* |
|---|---|
| 5 ms | 5 |

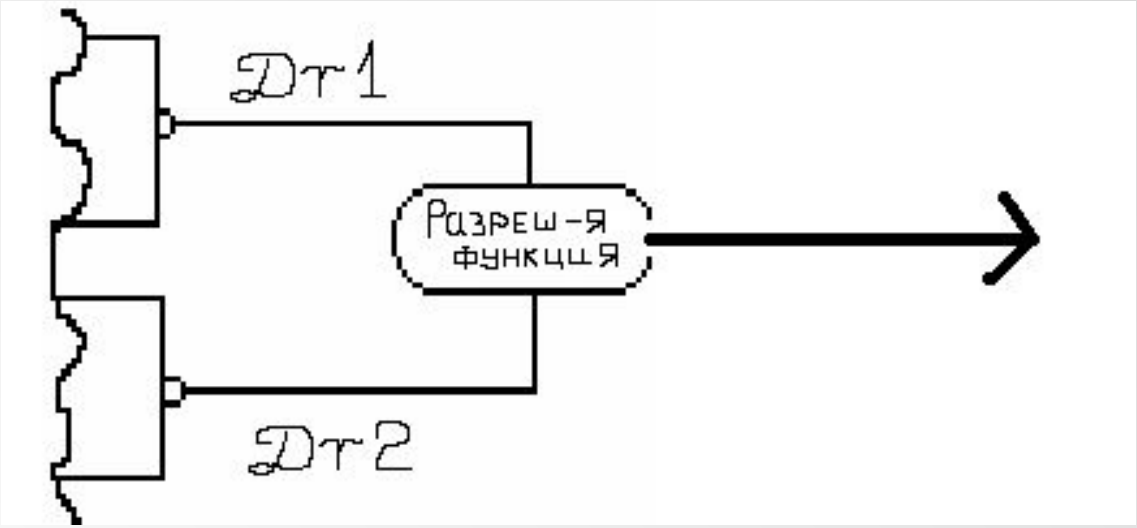
| *time* | *value* |
|---|---|
| 6 sec | 6 |

| *time* | *value* |
|---|---|
| 7 min | 7 |

| *time* | *value* |
|---|---|
| 8 hour | 8 |

Сигнал имеет несколько драйверов, если количество источников сигнала превышает 1, например при 2-направленных и трехстабильных цепей



**Металогический (metalogical) базис и многозначная логика**

Разновидности металогических базисов:

{01} – 2-х значный базис

{01X} – 3-х значный базис

{01XZ} – 4-х значный базис

{UX01ZWLH-} – 9-ти значный базис

# 9-ти значный базис – стандарт IEEE standart 1164:

```
PACKAGE std_logic_1164 IS

    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with the resolution function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;


    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
```

**Функция разрешения для** 9-ти значного базиса в пакете std_logic_1164:

```
-------------------------------------------------------------------
-- resolution function
-------------------------------------------------------------------
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' )  -- | - |
    );

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;
```

Другие базисы в пакете std_logic_1164:

```
SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
```

**Многозначная логика –** это логические операции над операндами, имеющими более 2-х значений.

В пакете std_logic_1164:

```
-------------------------------------------------------------------
-- overloaded logical operators
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

-------------------------------------------------------------------
-- vectorized overloaded logical operators
-------------------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;

FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
```

Реализация **логических функций в 9-ти значном базисе** в пакете std_logic_1164**:**

```
-------------------------------------------------------------------
-- tables for logical operations
-------------------------------------------------------------------

-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |    |
--      ---------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);

⊞   -- truth table for "or" function

⊞   -- truth table for "xor" function

⊞   -- truth table for "not" function
```

**Реализация** логических функций в других базисах **в пакете std_logic_1164**:

```
    -------------------------------------------------------------------
    -- overloaded logical operators ( with optimizing hints )
    -------------------------------------------------------------------

    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
    BEGIN
        RETURN (and_table(l, r));
    END "and";

⊞   FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS

⊞   FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS

⊞   FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS

⊞   FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS

⊞   function "xnor"  ( l : std_ulogic; r : std_ulogic ) return ux01 is

⊞   FUNCTION "not"  ( l : std_ulogic ) RETURN UX01 IS
```

# Моделирование задержек сигналов

**Простой оператор назначения сигналов**

*simple_SA::=*
*[Label:] signal_name <= [delay_mechanism] waveform;*

*waveform ::= {transaction} {,transaction}*

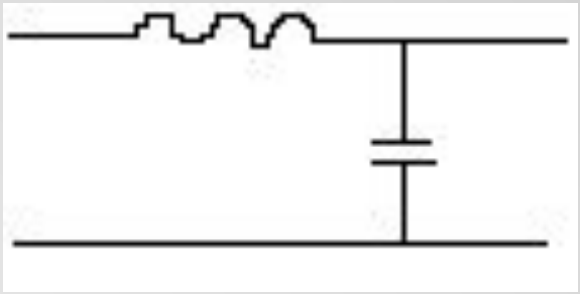
*transaction::= value_expression [**after** time_expression]*
*| **null** [**after** time_expression];*

*delay_mechanism::=* ***transport***
*| [**reject** reiect_time_expression ]* **inertial**

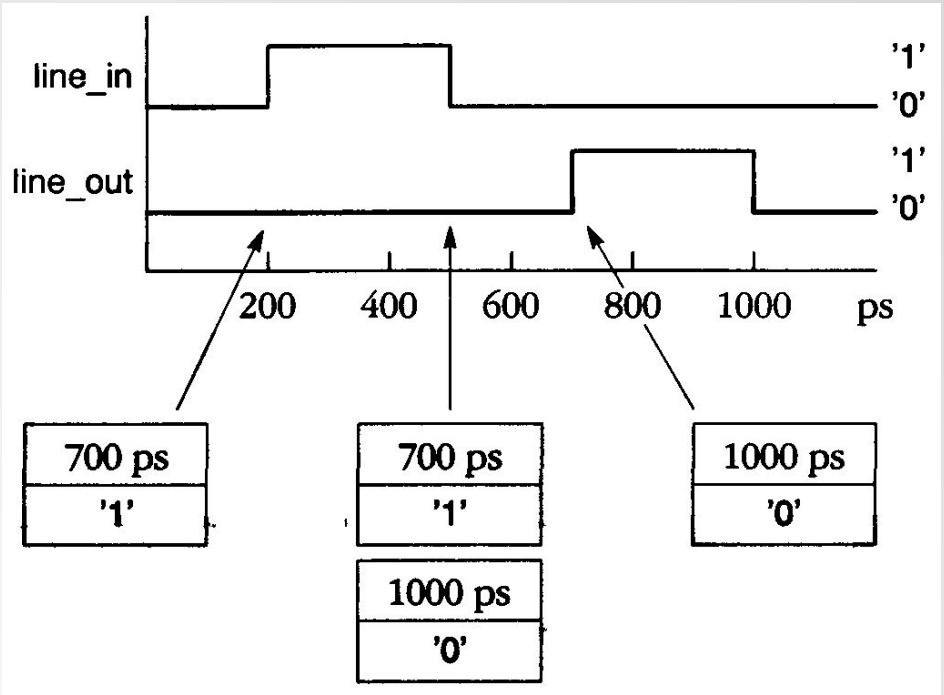
**Разновидности** задержек сигналов:

***- transport*** *- транспортная*
***- inertial*** *- инерционная*
***- reject inertial*** *– инерционная с фильтрацией*

***Транспортная задержка*** **–** описывает безинерционную задержку сигналов, вызванную наличием емкостей и индуктивностей и характерную для цепей передачи данных.

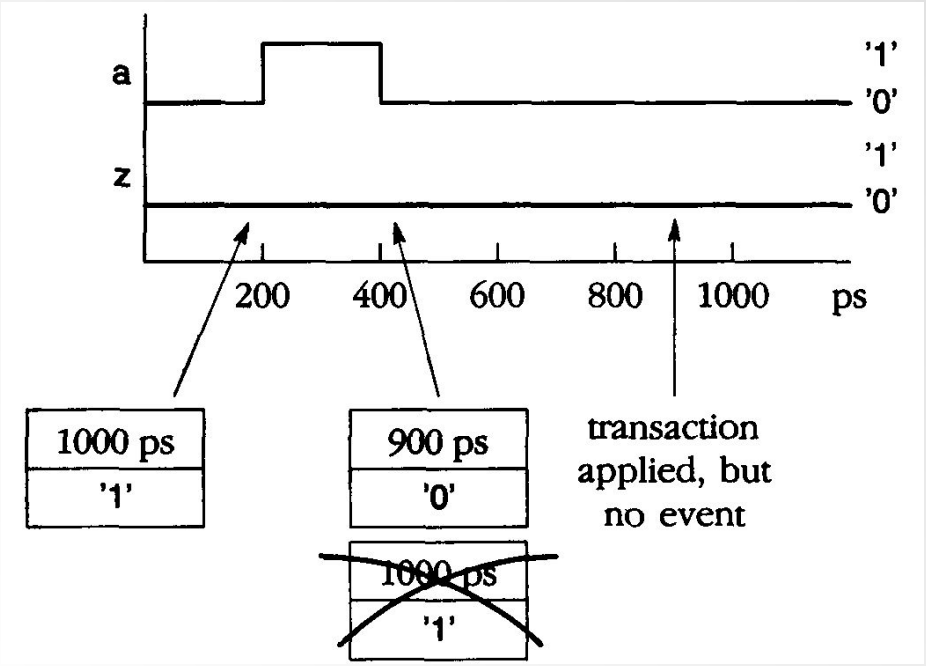


line_out <= **transport** line_in **after** 500 ps;



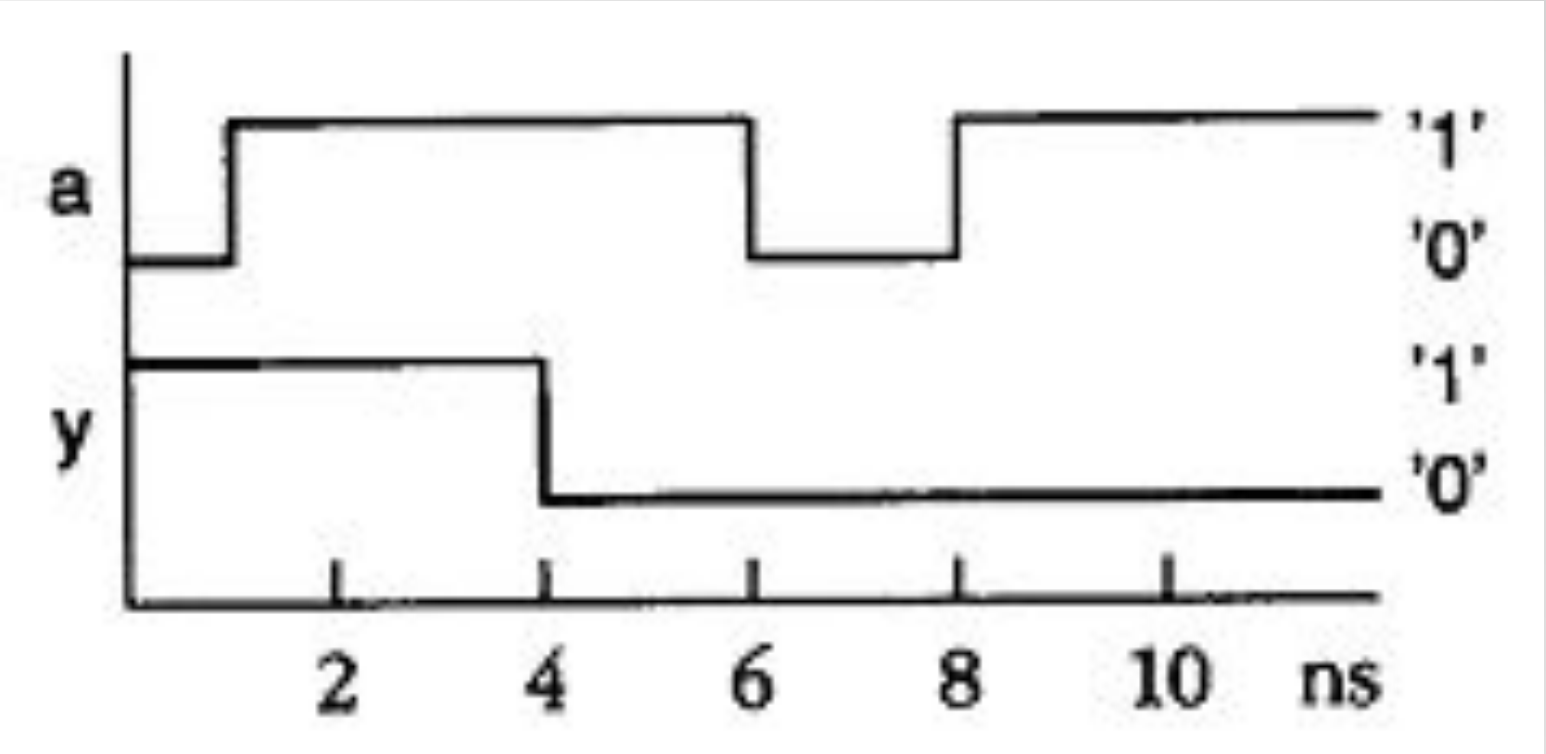
*Пример:*

```
asymmetric _delay : process (a) is
constant Tpd_01 : time := 800 ps;
constant Tpd_10 : time := 500 ps;
begin
if a =’1’  then
        z <= transport a after Tpd_01;
else
     z <= transport a after Tpd_10;
end if;
end process asymmetric_delay;
```



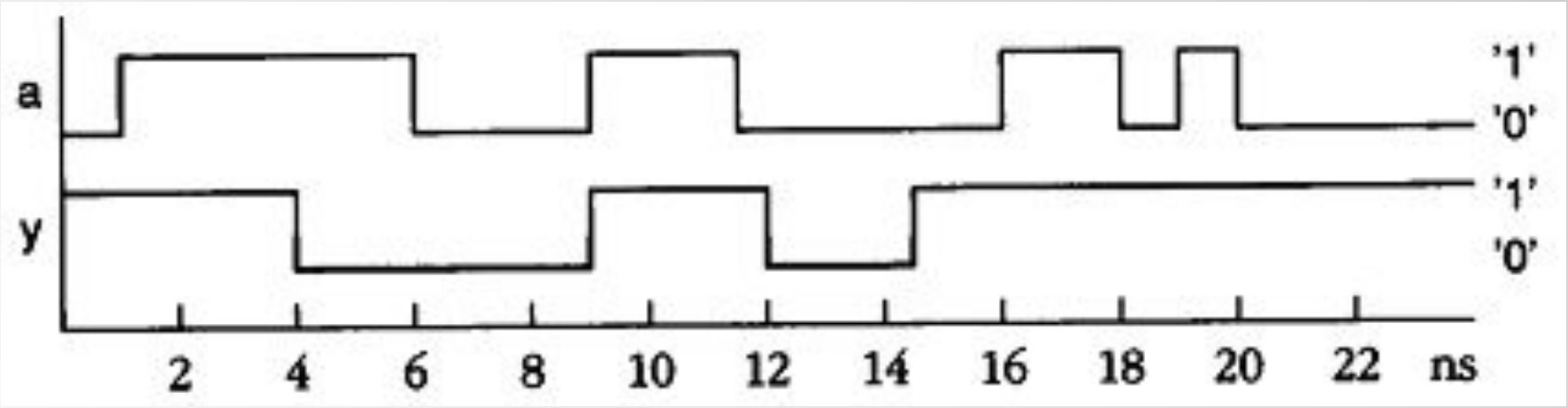
***Инерционная задержка –*** описывает задержки логических элементов

```
inv : process (a) is
begin
     y <= inertial not a after 3 ns;
end process inv;
```

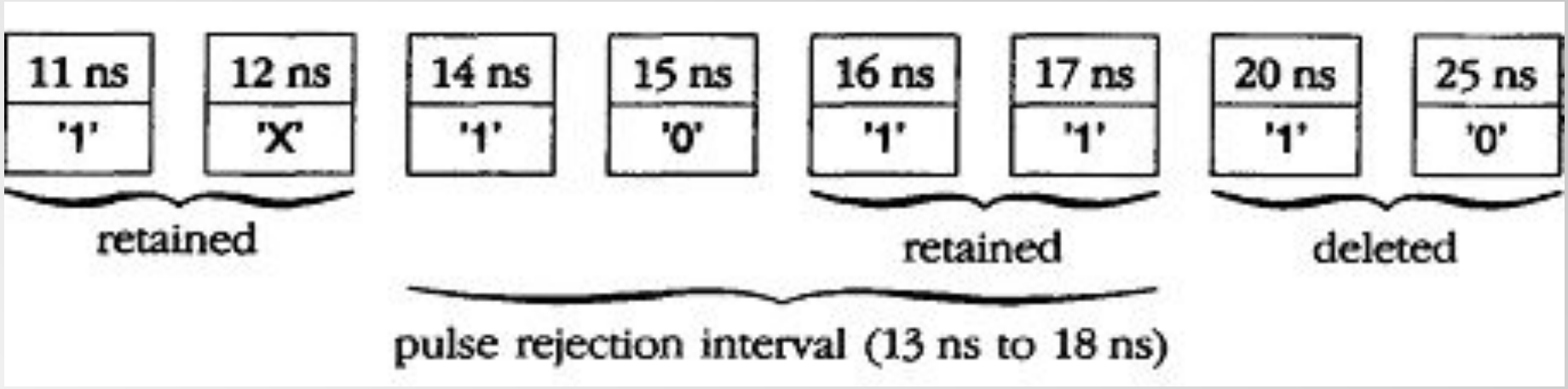


**Инерционная задержка с фильтрацией –** *описывает задержки логических элементов с возможностью отфильтровывать короткие импульсы*

```
inv : process (a) is
begin
    y <= reject 2 ns inertial not a after 3 ns;
end process inv;
```



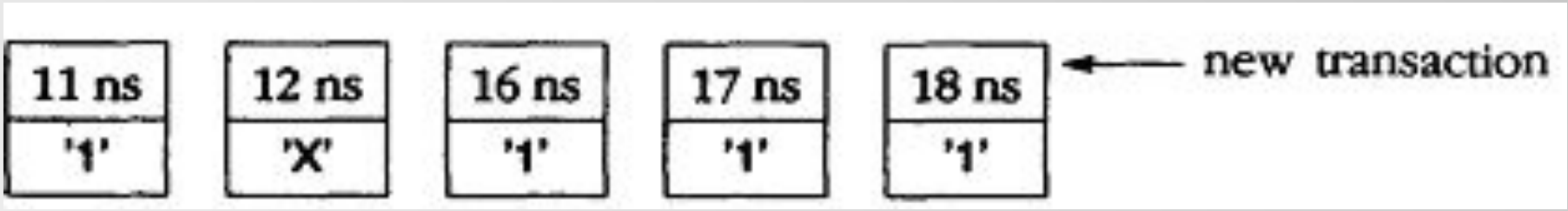
**Пример.** Предположим, что драйвер для сигнала s содержит следующие ожидаемые транзакции:




и процесс, содержащий драйвер выполняет следующее назначения сигнала во время 10 ns:

s <= **reject** 5 ns **inertial** '1' **after** 8 ns;

Тогда ожидаемые транзакции после этого назначения будут такими:



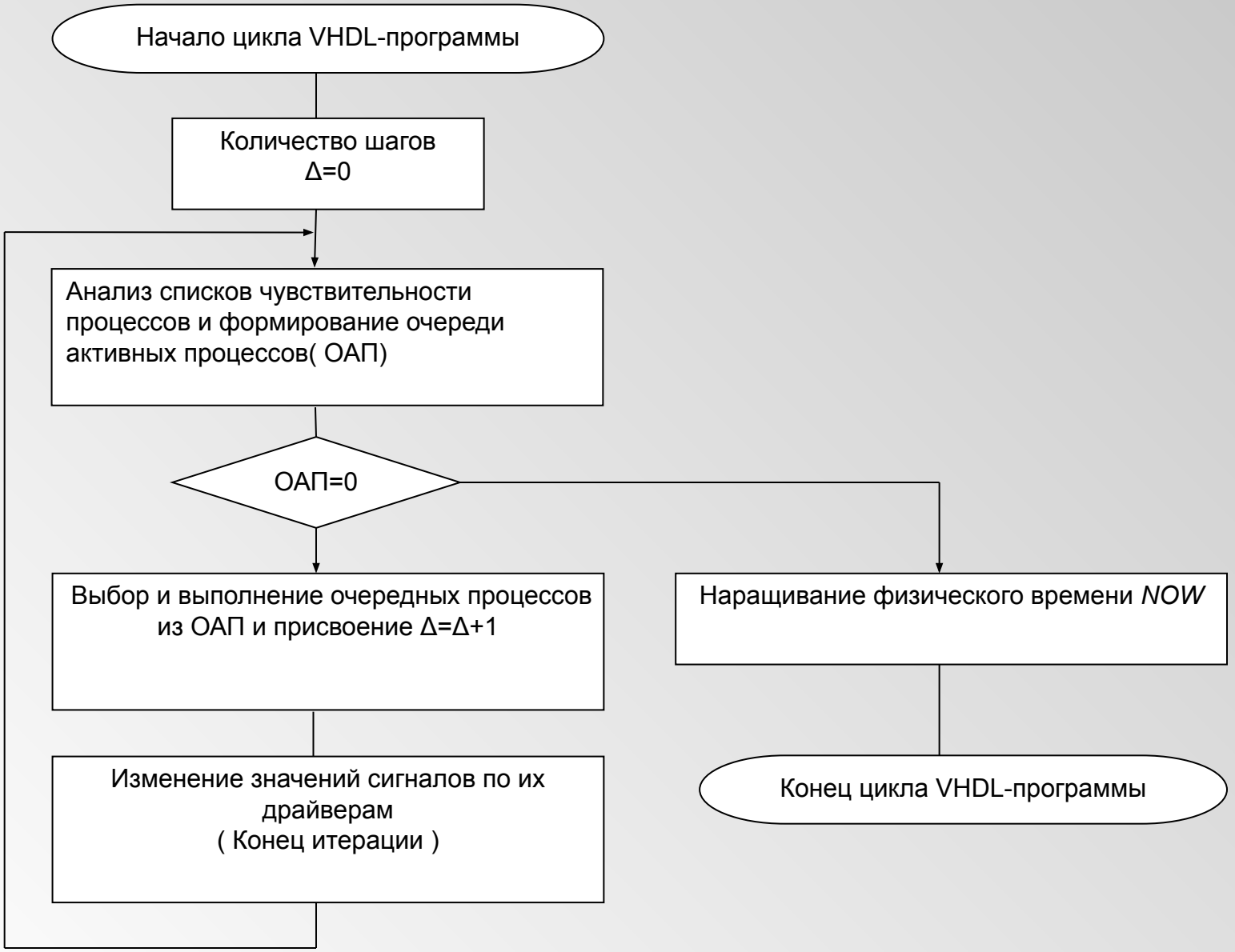
# Дельта задержка

**Delta delay - $\Delta$**

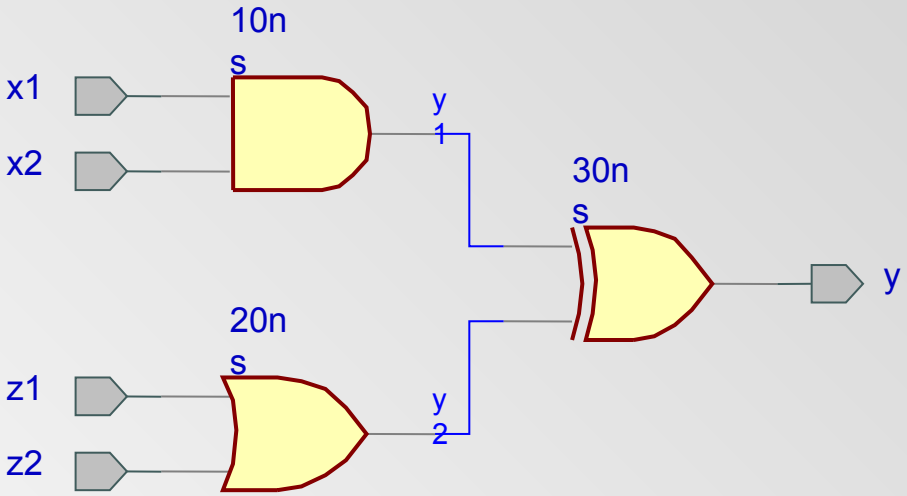
Все процессы в VHDL делятся на три вида:

- *active* - активный процесс;
- *executed* - выполняемый процесс;
- *postponed* - приостановленный процесс.

# Итерационный алгоритм работы VHDL-программы:

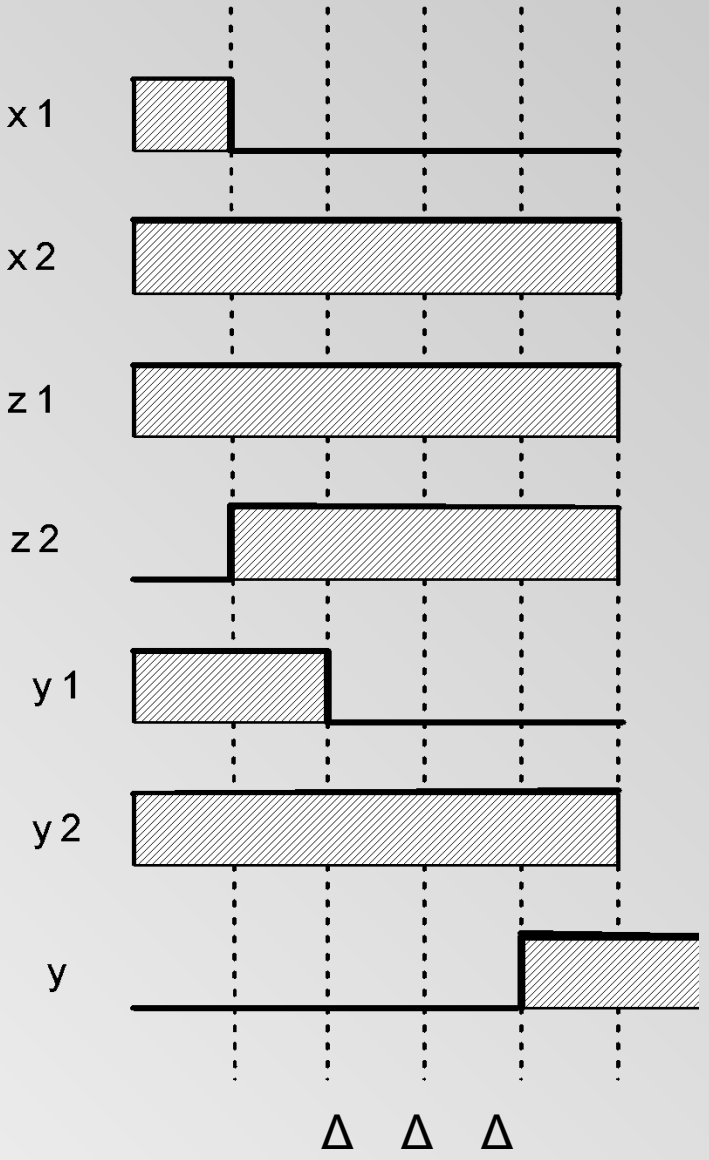


Например, для схемы

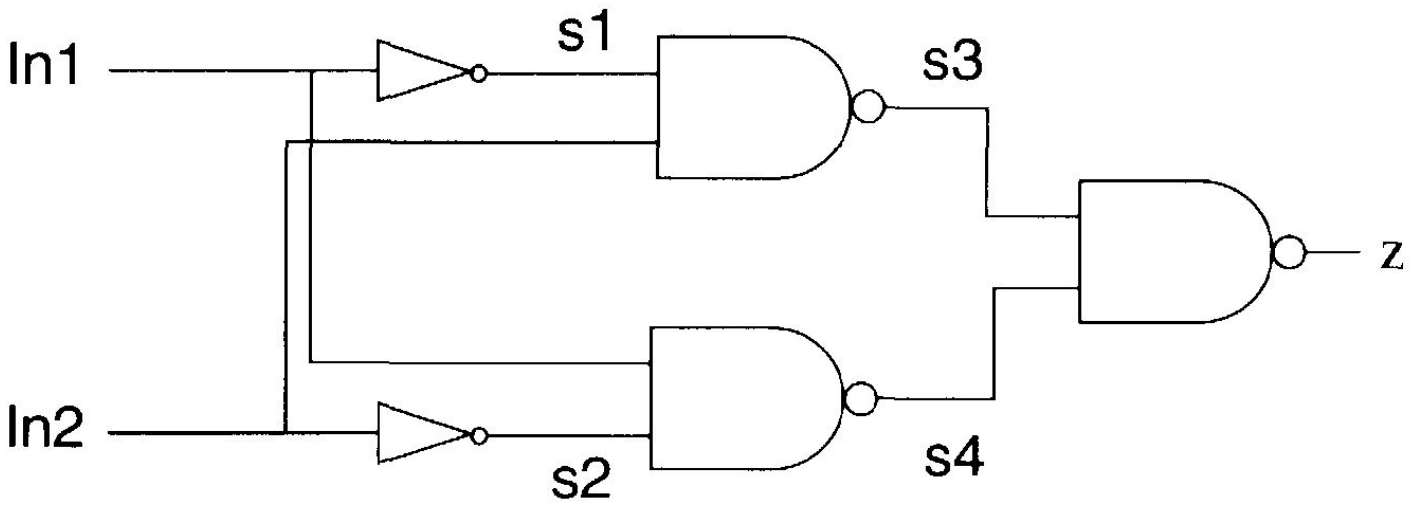



```
    y1 <= x1 and
x2;
    y2 <= z1 or
z2;
    y <= y1 xor
y2;
```

Итерационная временная диаграмма изменения сигналов:



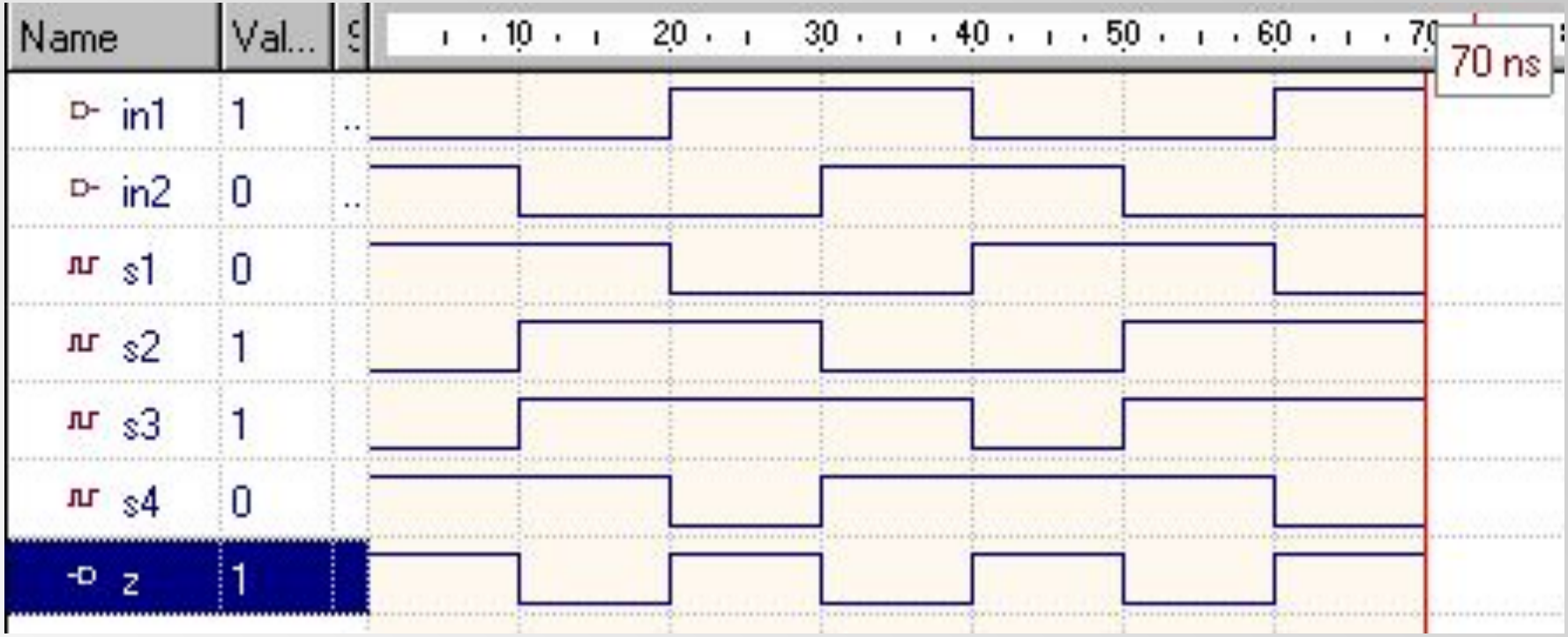
Рассмотрим комбинационную логическую схему и соответствующий VHDL код:



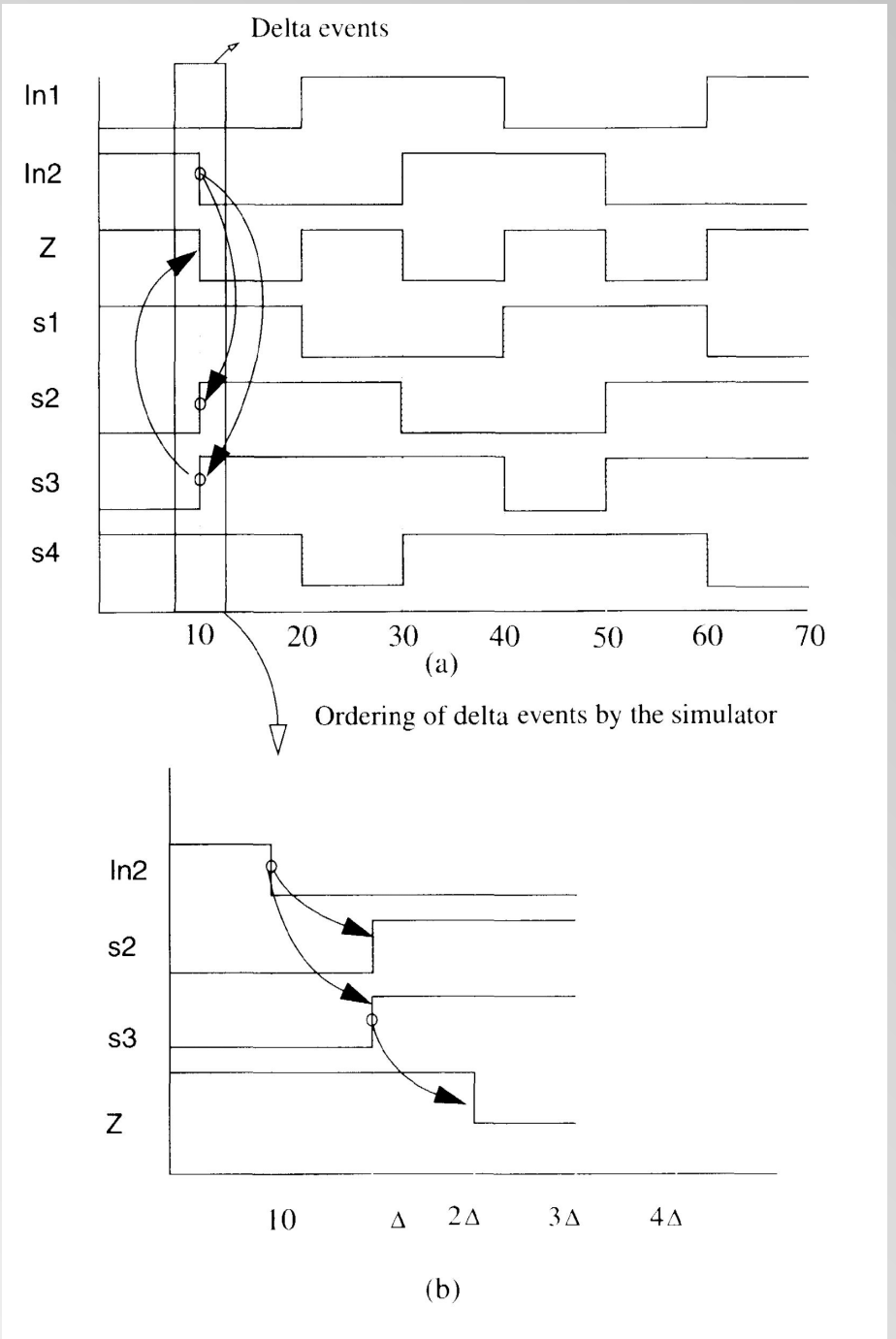
```
library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
   port (in1, in2:  in std_logic;
  z  : out std_logic);
end entity combinational;
architecture dataflow of combinational is
   signal s1, s2, s3, s4: std_logic:= '0';
begin
   s1 <= not in1;
   s2 <= not in2;
   s3 <= not (s1 and in2);
   s4 <= not (s2 and in1);
   z <= not (s3 and s4);
end architecture dataflow ;
```

Модель отображает потоковое описание схемы без спецификации *за*держек вентилей.

Результаты моделирования комбинационной схемы



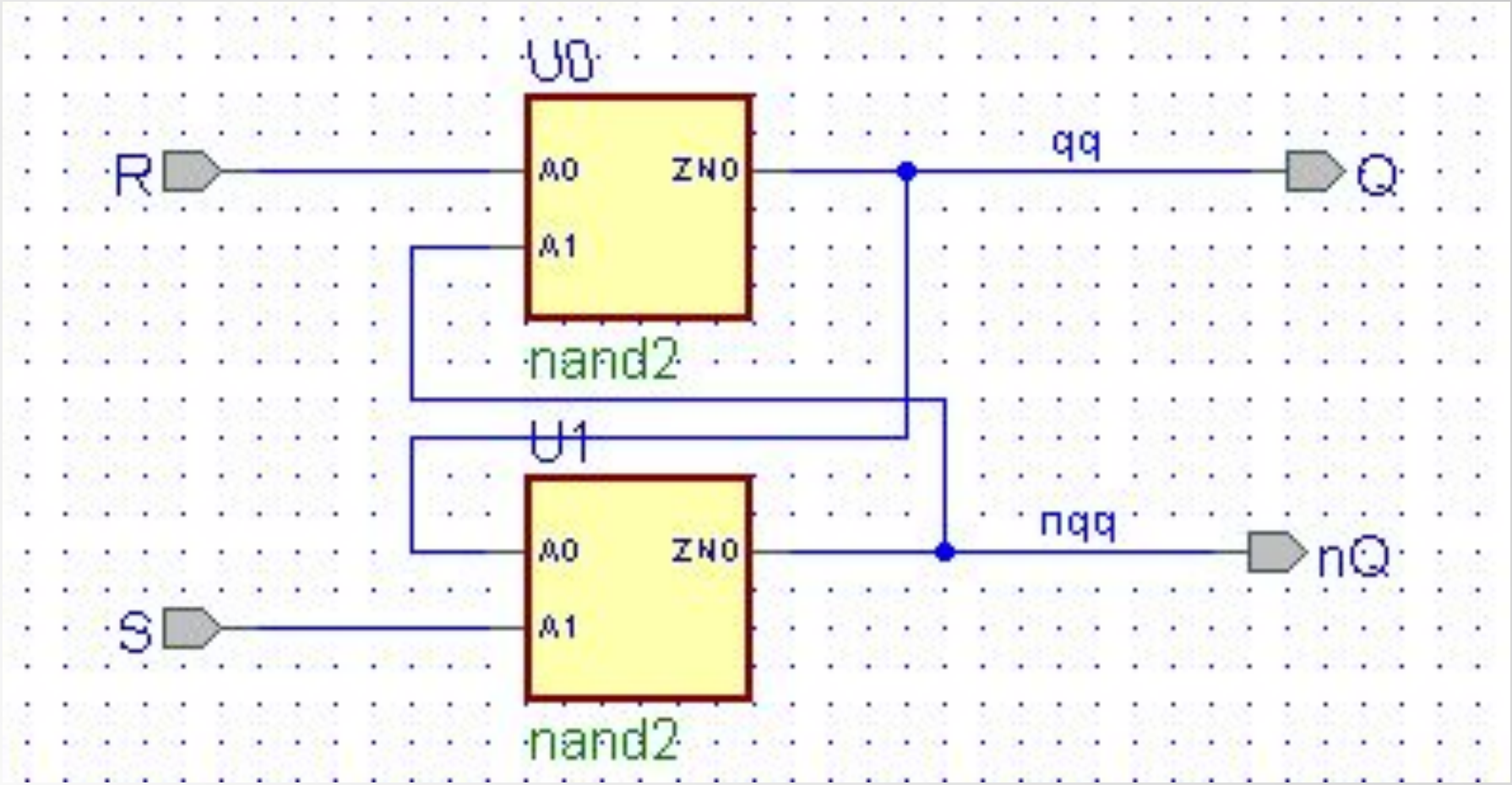
*очередность дельта-событий (а) и отображение дельта-задержек (б) сигналов при моделировании;*



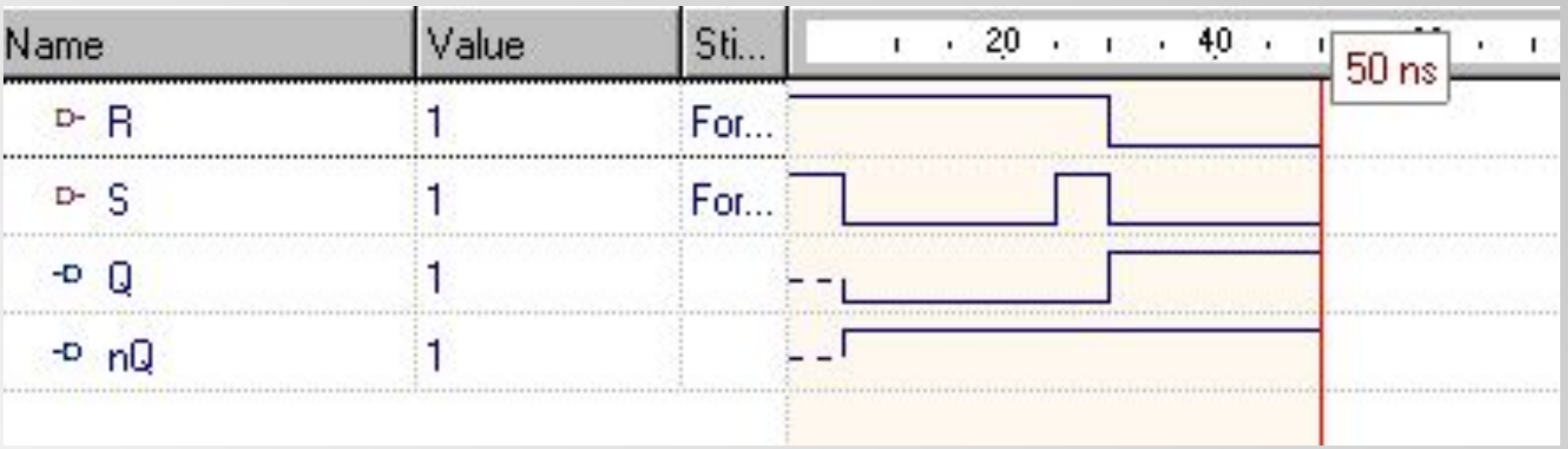
# Результаты итерационного моделирования комбинационной схемы

| Time | Delta | in1 | in2 | z | s1 | s2 | s3 | s4 |
|---|---|---|---|---|---|---|---|---|
| 0.000 ps | 0 | U | U | U | 0 | 0 | 0 | 0 |
| 0.000 ps | 1 | 0 | 1 | 1 | U | U | 1 | 1 |
| 0.000 ps | 2 | 0 | 1 | 0 | 1 | 0 | U | 1 |
| 0.000 ps | 3 | 0 | 1 | U | 1 | 0 | 0 | 1 |
| 0.000 ps | 4 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 10.000 ns | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 10.000 ns | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 10.000 ns | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 20.000 ns | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 20.000 ns | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 20.000 ns | 2 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 30.000 ns | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 30.000 ns | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 30.000 ns | 2 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 30.000 ns | 3 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 40.000 ns | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 40.000 ns | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 40.000 ns | 2 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 40.000 ns | 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 50.000 ns | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 50.000 ns | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

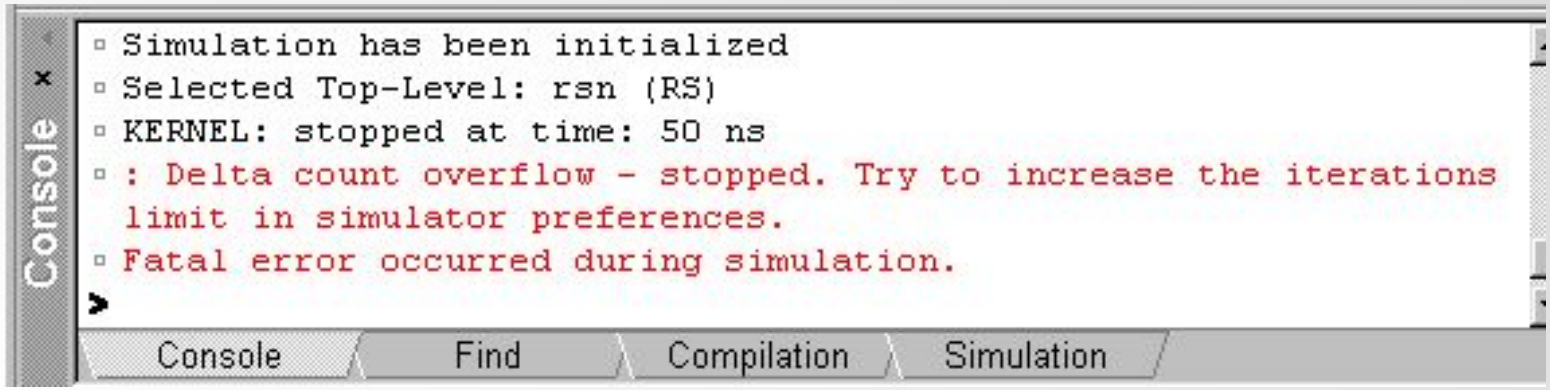
Пример моделирования RS-триггер без задержек в ЛЭ:



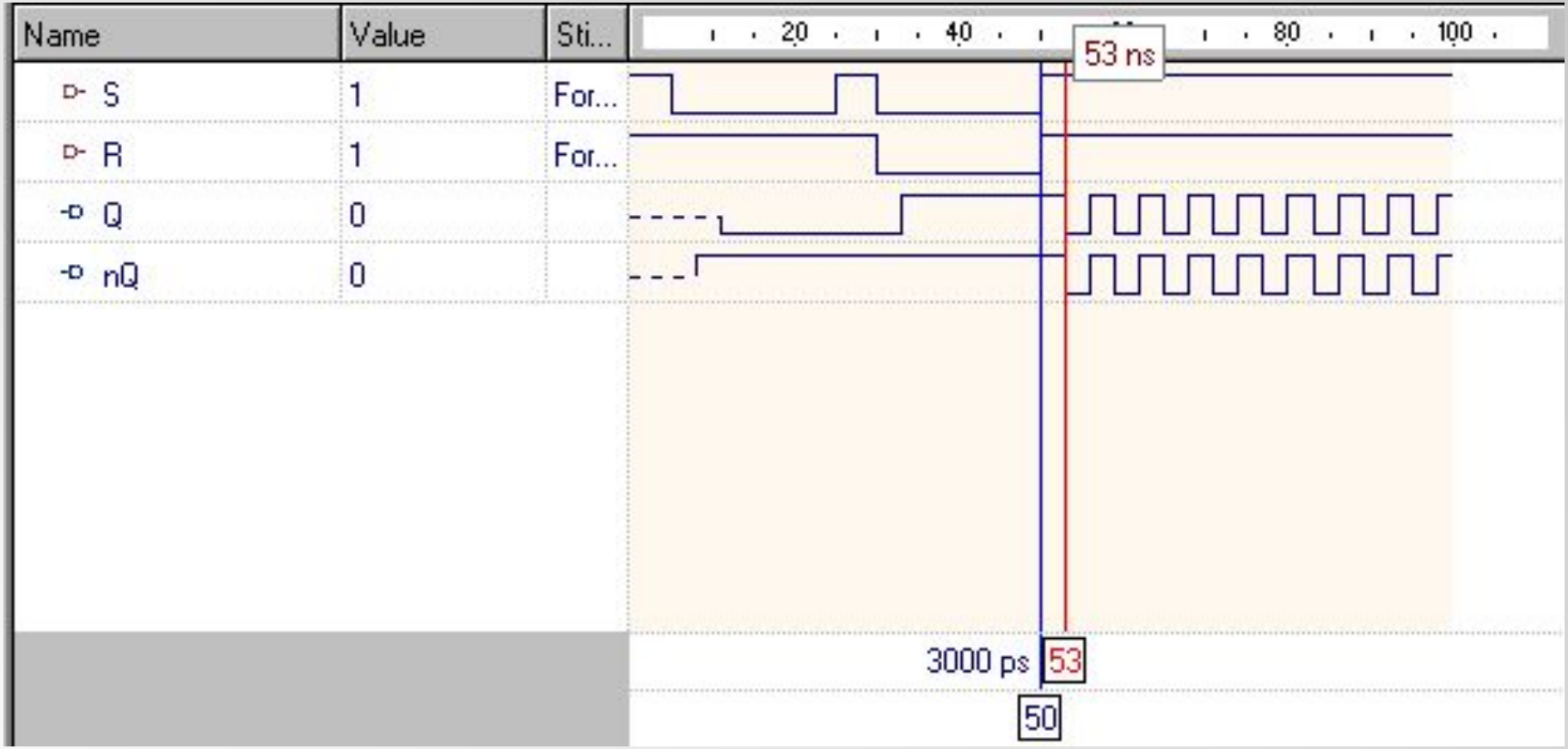
Временная диаграмма для триггера имеет вид:



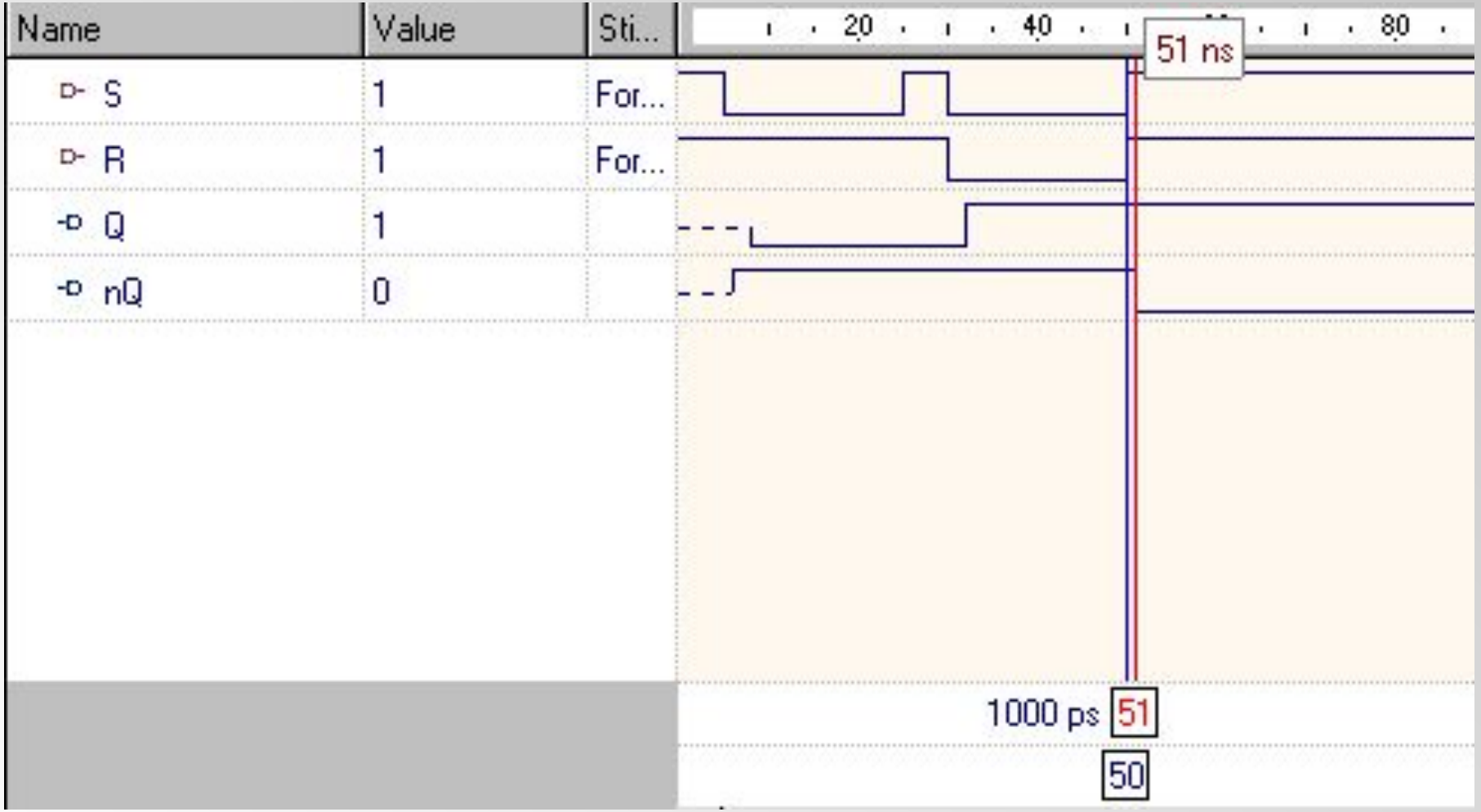

На 50 ns, когда сигналы ‘R’ и ’S’ после запрещенной комбинации ‘00’ перейдут в состояние хранения ‘11’, моделирование прекратится, так как при работе схемы RS-триггера без задержек возникают бесконечные итерации и цикл не завершается.
Редактор AHDL при этом выдаст следующее сообщение об ошибке:



Теперь промоделируем RS-триггер с равными задержками ЛЭ в 3 ns.



*Временная диаграмма RS-триггера с разными задержками*



# Предопределенные атрибуты сигналов

*Атрибут* - это значение, предопределенное системой или пользователем.. В первом случае атрибут называется предопределенным, во втором- пользовательским.

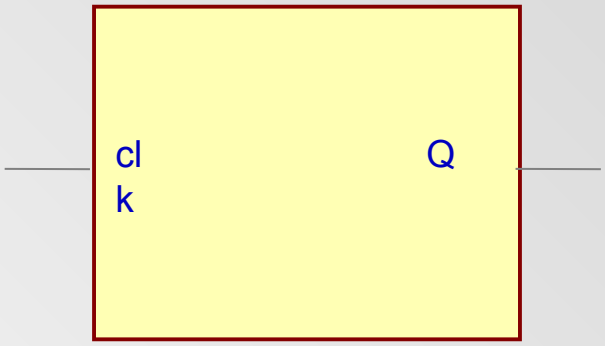
В VHDL есть ряд предопределенных атрибутов для таких объектов, как массивы, блоки, сигналы, типы.

Пользовательские атрибуты можно создать для:
- устройств;
- архитектурных тел;
- конфигураций;
- процедур;
- функций;
- пакетов;
- типов;
- подтипов;
- констант;
- сигналов;
- переменных;
- компонент;
- меток.

***attribute*** *::= prefix' attribute_name[( expression )]*

| Имя атрибута | Род атрибута | Возвращаемый тип данных | Примечание |
|---|---|---|---|
| ***'active*** | *Function* | *Boolean* | Перевычисление сигнала:<br>*Attr = true*, если сигнал перевычисляется в данный момент<br>*Attr = false*, иначе. |
| ***'delayed****(t)* | *Signal* | *Тип префикса* | Задержка сигнала:<br>*Attr = S(Now-t)* |
| ***'event*** | *Function* | *Boolean* | Изменение сигнала:<br>*Attr = true*, если *S(Now)* ≠ *S(Now-Δt)* |
| ***'quiet****(t)* | *Signal* | *Boolean* | Спокойствие сигнала:<br>*Attr = true*, если S не перевычислялся в течение последнего времени t. |
| ***'stable****(t)* | *Signal* | *Boolean* | Стабильность сигнала:<br>*Attr = true*, если сигнал не изменялся в течение последнего времени t. |
| ***'transaction*** | *Signal* | *Bit* | Активность транзакции:<br>*Attr = not Attr*, когда *S'active = true* |
| ***'last_active*** | *Function* | *Time* | Время активации сигнала:<br>*Attr* = время, прошедшее после последней активации (перевычисление) сигнала. |
| ***'last_event*** | *Function* | *Time* | Время изменения сигнала:<br>*Attr* = время, прошедшее с момента последнего изменения сигнала. |
| ***'last_value*** | *Function* | *Тип префикса* | *Attr* = предыдущее значение S |

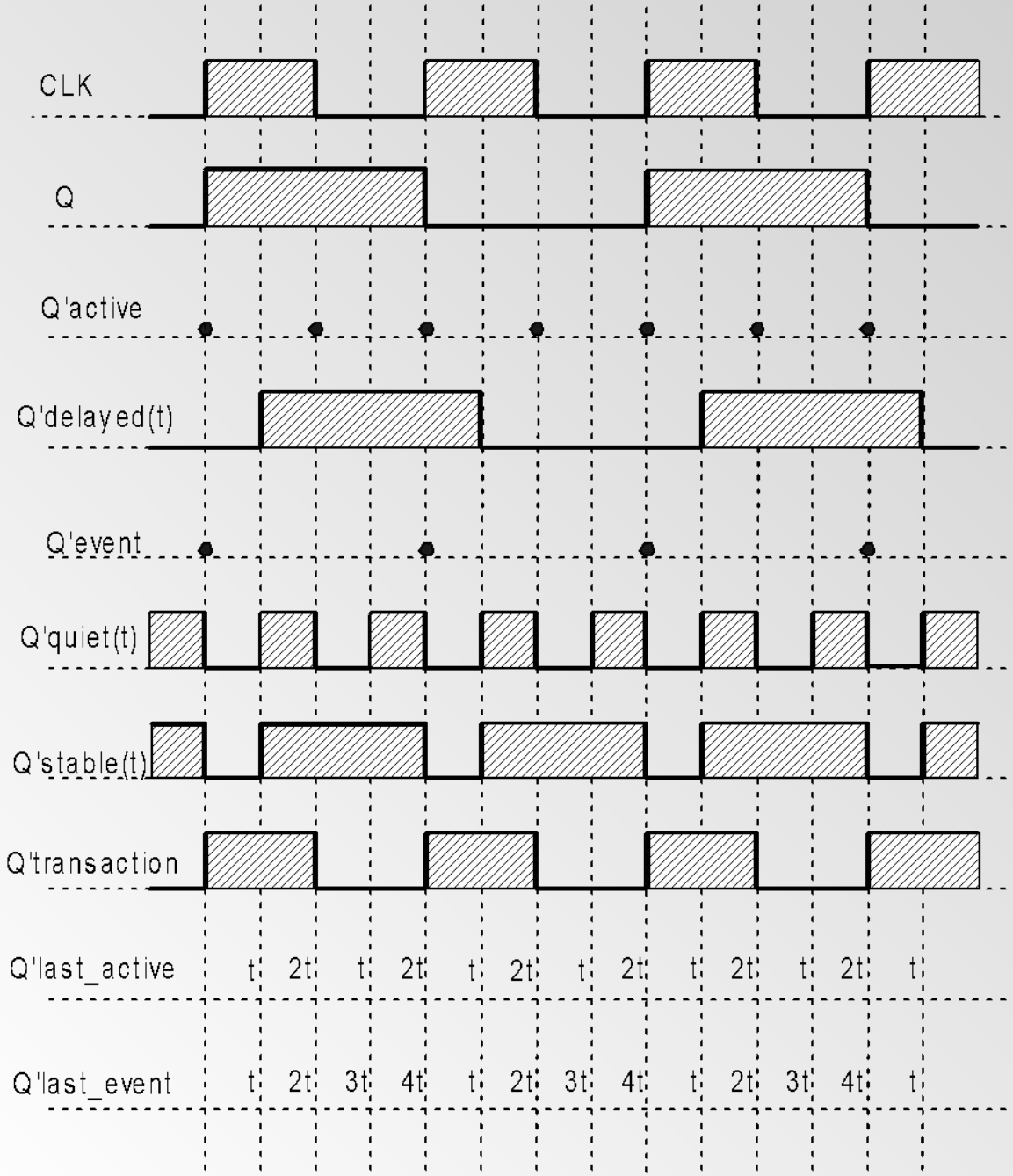
Использование атрибутов для сигналов приведем на примере Т-триггера




**T-trigger**

VHDL-код для данного примера будет следующим

```
entity T_FF is
  port( clk: in  std_logic;
          Q: out std_logic);
end T_FF;
architecture arch_T_FF of T_FF is
begin
  process( clk)
  variable x: bit :='0';
  begin
    if clk='1' then x:= not x; end if;
    Q<=x;
  end process;
end T_FF;
```


CLK
Q
Q'active
Q'delayed(t)
Q'event
Q'quiet(t)
Q'stable(t)
Q'transaction
Q'last_active
t 2t t 2t t 2t t 2t t 2t t 2t t
Q'last_event
t 2t 3t 4t t 2t 3t 4t t 2t 3t 4t t