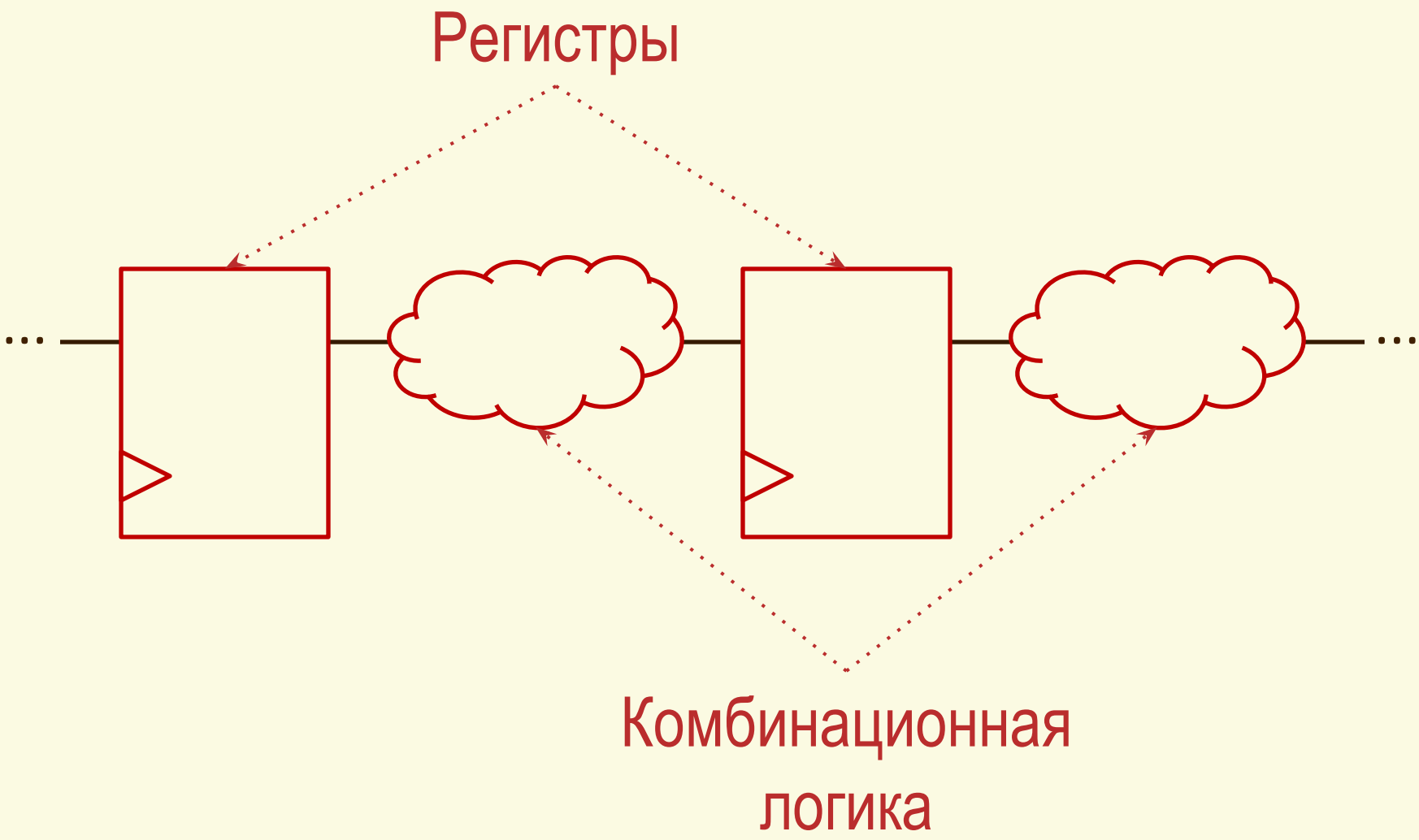
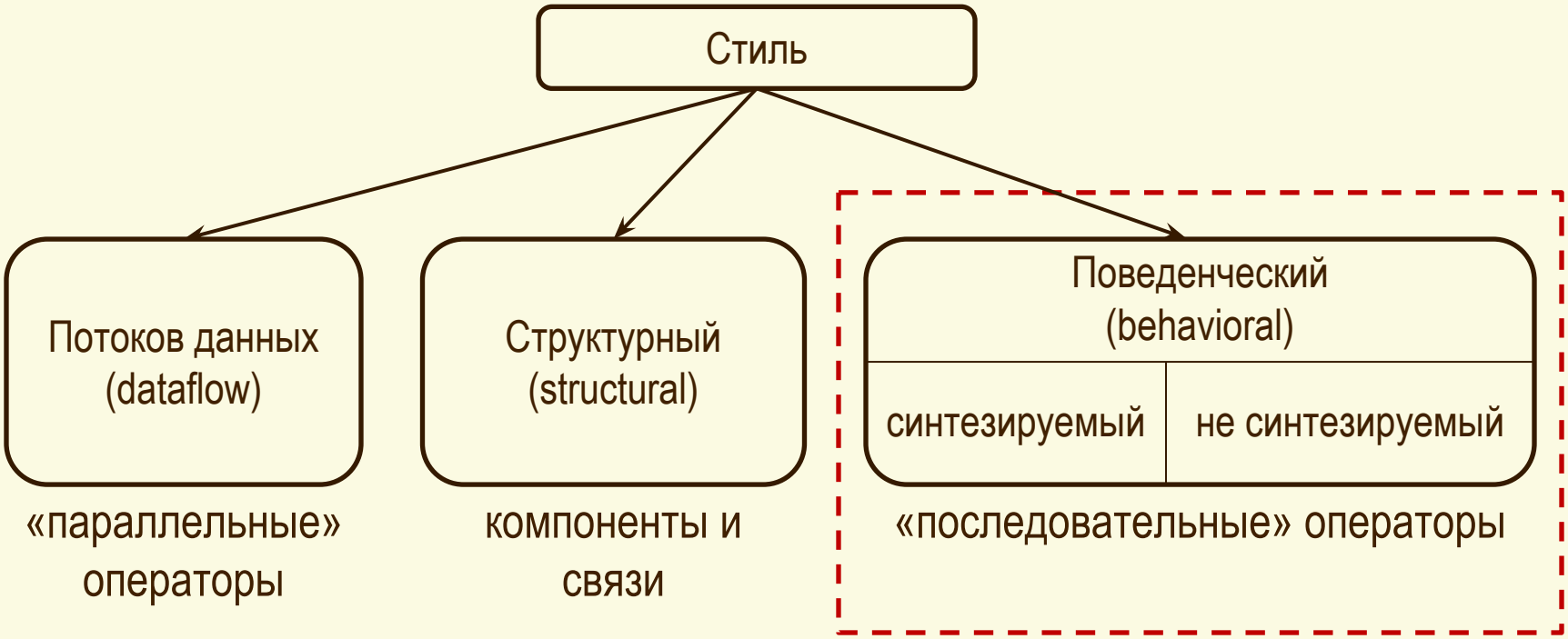
Проектирование цифровых устройств на языке VHDL

Описание схем с памятью

Структура цифрового устройства

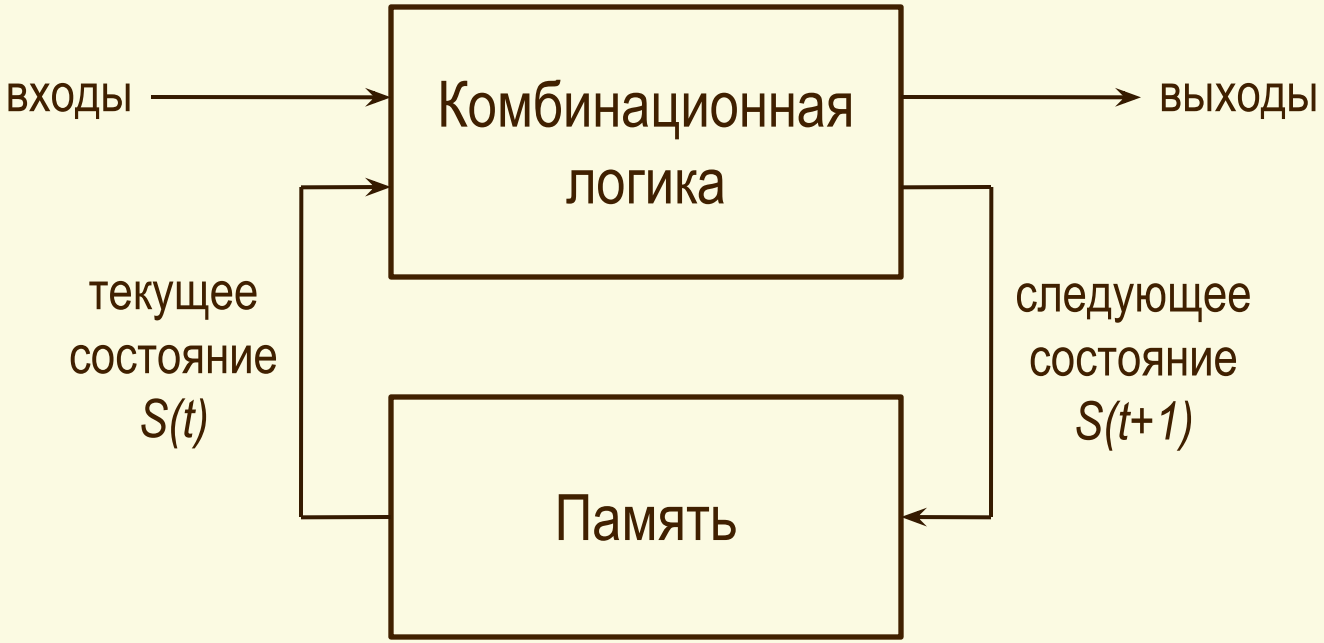


Стили описания на VHDL



Особенности схем с памятью

Структура схемы с памятью



Особенности схем с памятью

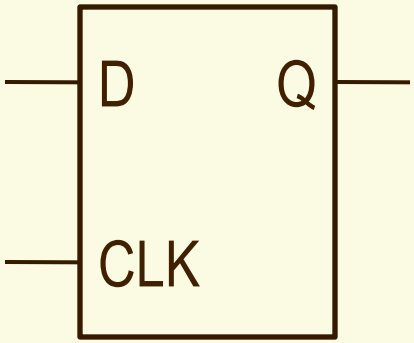
Комбинационная схема

- Модель описания – система булевых уравнений
- Выход зависит только от текущих входов
- Данные распространяются по схеме от входов к выходам
- Значения изменяются при изменении входов
- Для хранения значений могут использоваться статические триггеры (latches)

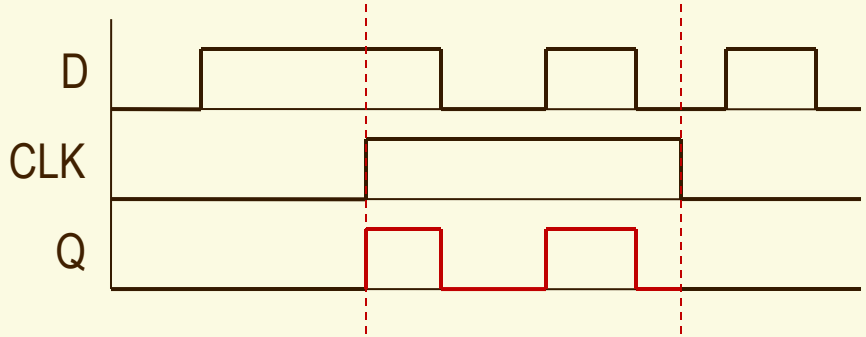
Схема с памятью

- Модель описания – конечный автомат
- Выход зависит от текущих входов и их истории
- В схеме присутствуют обратные связи
- Значения изменяются по фронту синхросигнала
- Для хранения значений используются динамические триггеры (flip-flops) и память (RAM)

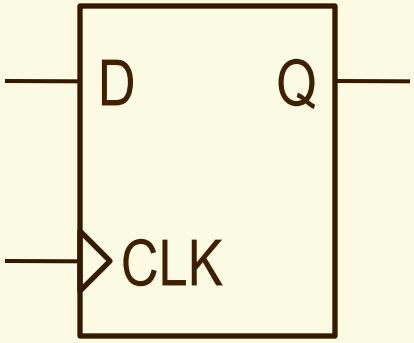
Статические и динамические триггеры



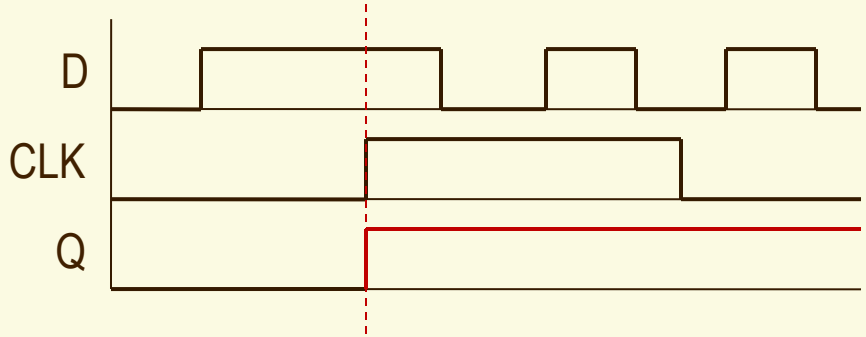
latch



Статический триггер «прозрачен» при высоком уровне синхросигнала



flip-flop



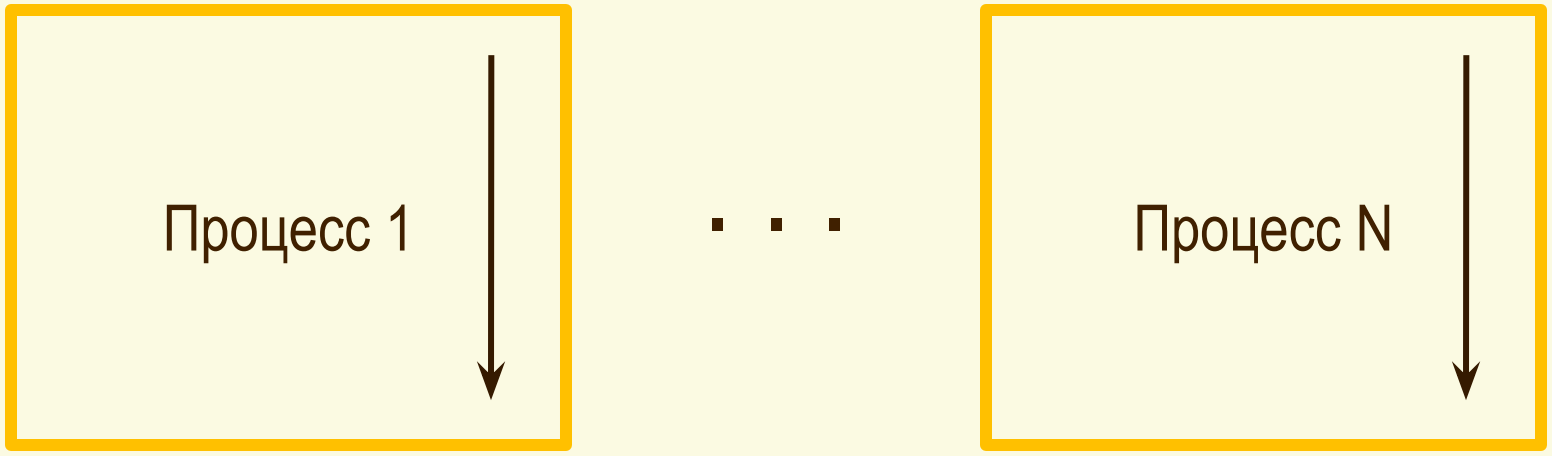
Динамический триггер запоминает значение в момент прихода фронта синхросигнала


PHY
TPO+
TPO-
TPI+
TPI-
LNKB
LBK
CNTRLB
RBIAS
LEDA#
RJ45INTLED
VCC3_3

Процессы

Поведенческий стиль

- Поведенческое описание ↔ алгоритм
- Для описания используются блоки **PROCESS**
  - Внутри процесса весь код выполняется последовательно
  - Между собой процессы выполняются параллельно



Процессы – синтаксис

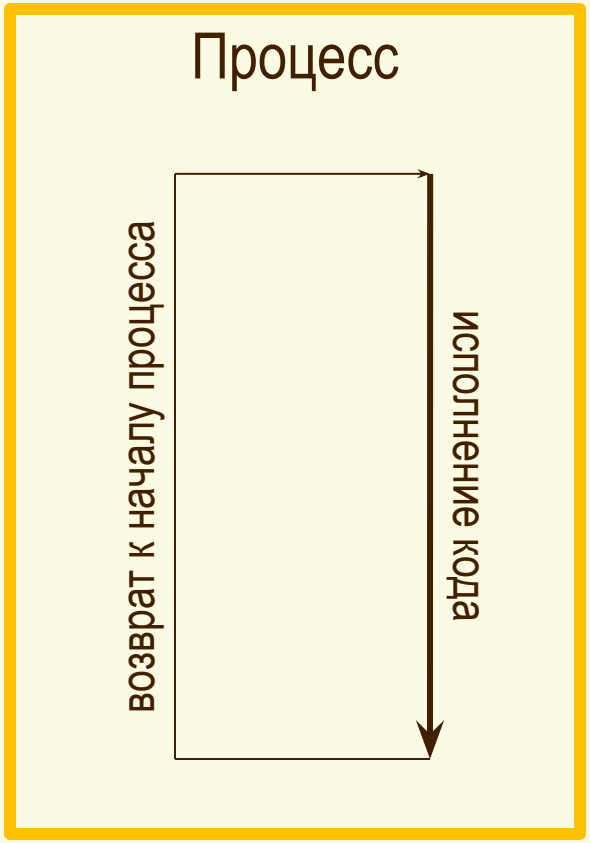
Тело процесса

- Содержит последовательные операторы
  - Выполняются по очереди (один за другим) ⟶ возможна организация ветвлений и циклов
  - Синтаксис операторов отличается от их параллельных аналогов
- Семантика близка к традиционным языкам программирования

```
process (sensitivity_list)
        declarations
begin
        statements
end process;
```

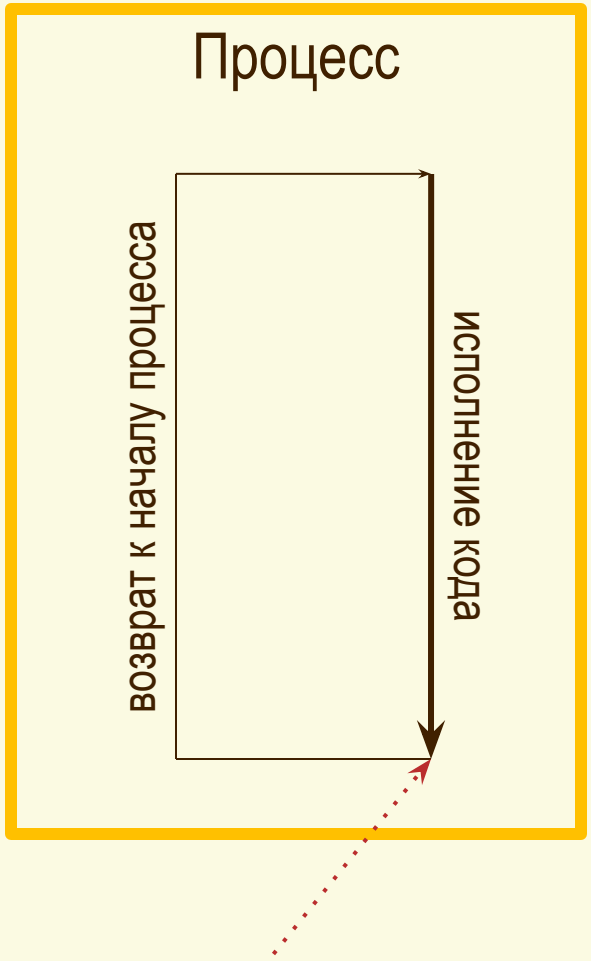
Тело процесса

- Внутри процесса все операторы выполняются последовательно
  - Порядок имеет значение
- Процесс выполняется в бесконечном цикле
  - Исполнение начинается при изменении сигналов, к которым чувствителен процесс
  - Исполнение приостанавливается в конце процесса



Список чувствительности (sensitivity list)

- Описывает сигналы, к которым чувствителен процесс
  - Сигналы перечисляются через запятую
  - Процесс начинает исполняться при изменении сигнала из списка чувствительности и приостанавливается по достижении **end process**
  - Эквивалентен оператору **wait on**
- Обязателен в синтезируемом коде
  - Может не использоваться в testbench



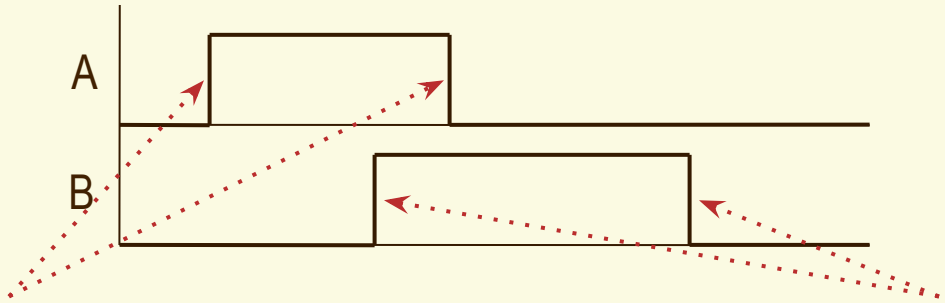

```
process (sensitivity_list)
        declarations
begin
        statements
end process;
```

Список чувствительности (sensitivity list)

```
process
begin
    wait on a, c;
    . . .
end process;
```

```
process (a, c)
begin

    . . .
end process;
```



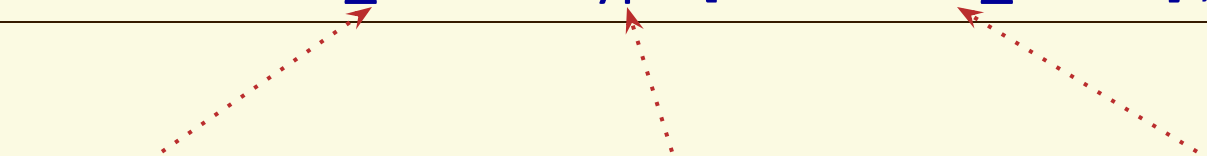
сигнал ***a*** в списке чувствительности – код процесса будет выполнен

сигнал ***b*** не в списке чувствительности – процесс не выполнится

Объявления переменных (declarations)

- Содержит описание локальных переменных процесса
  - Не может содержать объявления сигналов
- Синтаксис объявления переменной

```
variable var_name : type [ := default_value ] ;
```




```
process (sensitivity_list)
        declarations
begin
        code
end process;
```

Сигналы и переменные

Сигналы

- Могут быть объявлены в интерфейсе объекта или в архитектуре
- Оператор присваивания **<=**
- Значение присваивается при приостановке процесса
- Имеют «историю» (можно узнать, например, время последнего изменения)

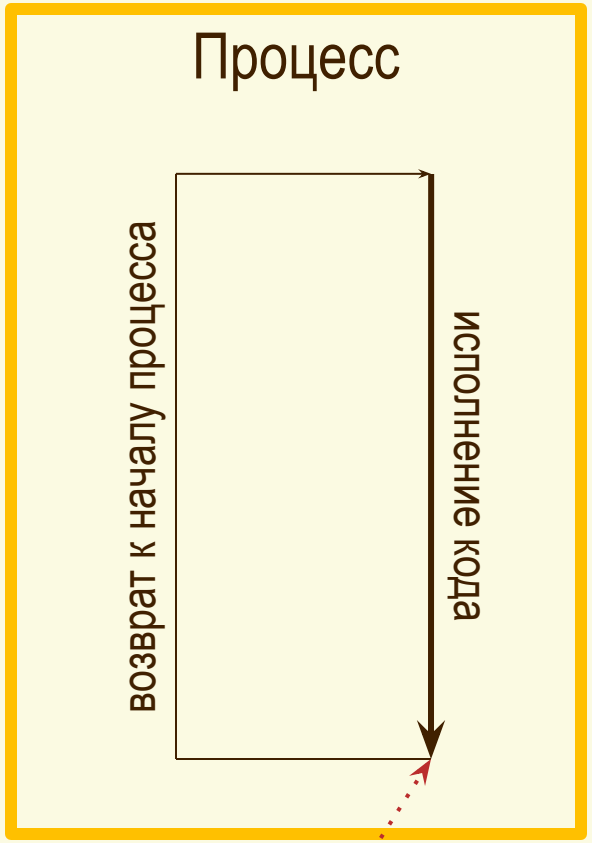
Переменные

- Могут быть объявлены и использованы только внутри процесса
- Оператор присваивания **:=**
- Значение присваивается немедленно
- Характеризуются только текущим значением

И сигналы, и переменные сохраняют свое значение между итерациями процесса

Присваивание значений сигналам

- Значения присваиваются сигналам при приостановке процесса:
  - По инструкции **wait**
  - По достижении конца процесса




```
process (clk)
begin
        . . .
        sig <= “0010”;
        if (sig = “0010”) then
                . . .
        end if;
        . . .
end process;
```

сигнал

проверяется предыдущее значение

значения сигналам будут присвоены здесь

Условный оператор if-elsif-else

Условный оператор

- Оператор **if-elsif-else**
  - Ветви **elsif** и **else** являются опциональными
- Выполняет первый блок кода $statements_i$, для которого условие истинно
  - Если все условия ложны, выполняется безусловная ветвь **else**

```
if (condition1) then
    statements1
elsif (condition2) then
    statements2
…
elsif (conditionN) then
    statementsN
else
    statementsN+1
end if;
```

Описание фронта сигнала – функция rising_edge

rising_edge(sig_name)

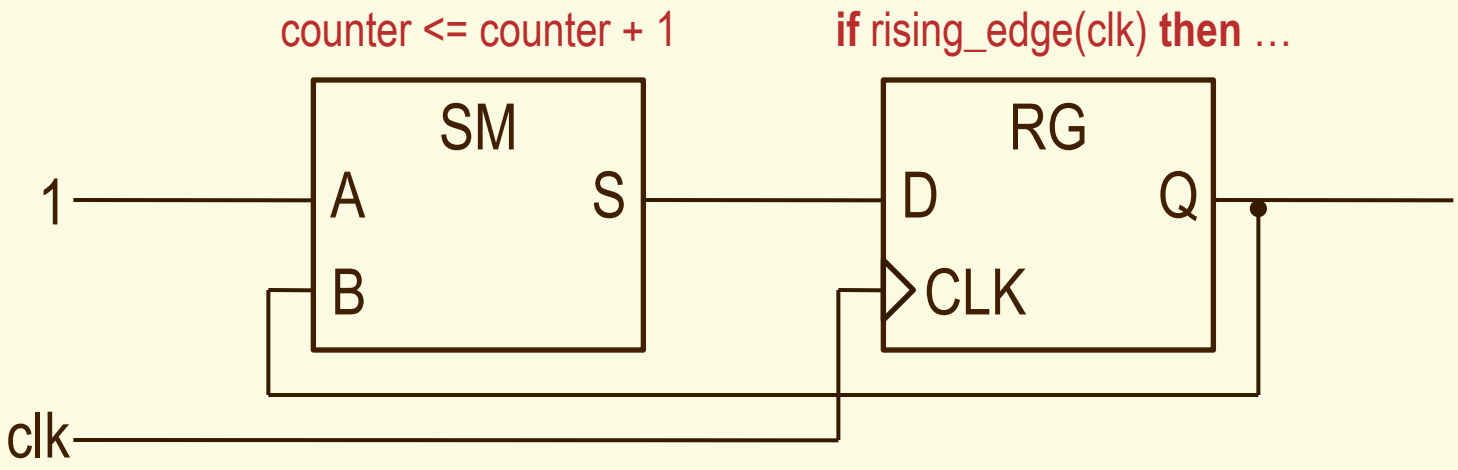
- Возвращает истинное значение в момент восходящего фронта сигнала
  - Иначе возвращает значение «ложь»
  - Для описания нисходящего фронта может использоваться функция **falling_edge**(sig_name)
- Используется для описания синхронных схем
  - Аппаратная модель – динамический триггер

шаблон описания синхронной схемы

```
process(clk)
begin
    if rising_edge(clk) then
        . . .          -- Значения сигналов и переменных будут присвоены
                       -- по фронту сигнала CLK
    end if;
end process;
```

Описание фронта сигнала – функция rising_edge

```
process(clk)
begin
      if rising_edge(clk) then
            counter <= counter + 1;
      end if;
end process;
```



rising_edge и sig_name’event

- Альтернативное условие на фронт синхросигнала

sig_name**’event and** sig_name = ‘1’

- Различия между rising_edge и sig_name’event
  - rising_edge определяет переходы 0⟶1, L ⟶1
  - sig_name’event and sig_name=‘1’ определяет любые переходы sig_name вида * ⟶1, кроме 1 ⟶1

Синхронные и асинхронные операции

- Синхронные операции – «привязаны» к фронту синхросигнала
  - Пример: запись в регистр
- Асинхронные операции – выполняются в произвольный момент времени (обычно по условию)
  - Пример: асинхронный сброс (reset)
- В условном операторе **if-else-elsif** как правило:
  - асинхронные операции предшествуют условию rising_edge(sig_name)
  - синхронные операции соответствуют условию rising_edge(sig_name)

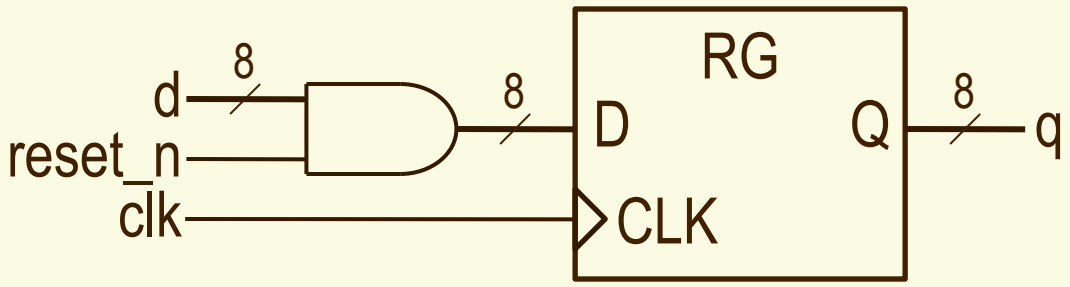
Пример 1

8-разрядный регистр с синхронным сбросом

```
library ieee;
use ieee.std_logic_1164.all;

entity reg8 is
        port (
                clk, reset_n  : in std_logic;
                d                               : in std_logic_vector(7 downto 0);
                q                               : out std_logic_vector(7 downto 0));
end entity;

architecture behav of reg8 is
begin
        process(clk)
        begin
                if rising_edge(clk) then
                        if (reset_n = '0') then
                                q <= x"00";
                        else
                                q <= d;
                        end if;
                end if;
        end process;
end architecture;
```



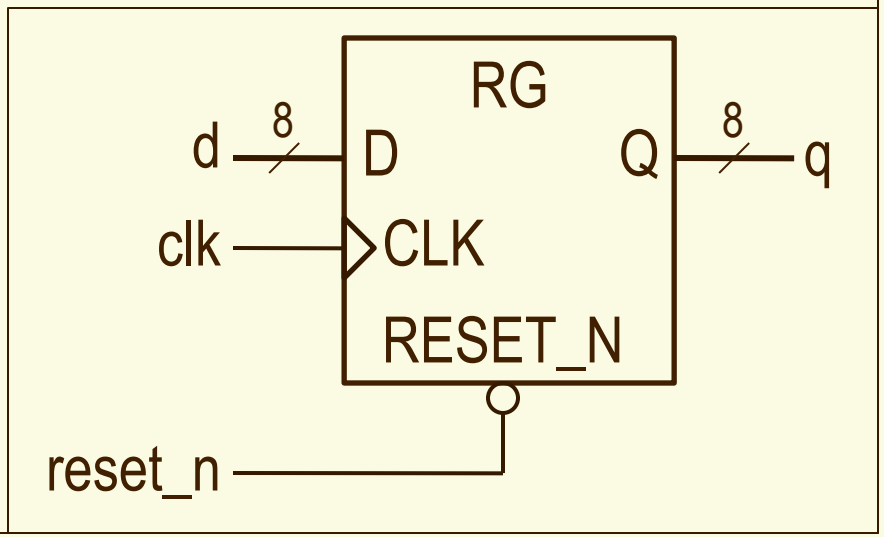
Пример 2

8-разрядный регистр с асинхронным сбросом

```
library ieee;
use ieee.std_logic_1164.all;

entity reg8 is
        port (
                clk, reset_n  : in std_logic;
                d                       : in std_logic_vector(7 downto 0);
                q                       : out std_logic_vector(7 downto 0));
end entity;

architecture behav of reg8 is
begin
        process(clk, reset_n)
        begin
                if (reset_n = '0') then
                        q <= x"00";
                elsif rising_edge(clk) then
                        q <= d;
                end if;
        end process;
end architecture;
```



Оператор множественного ветвления case

Оператор множественного ветвления

- Оператор **case**
  - Аналог в традиционных языках программирования – оператор **switch**
  - Аналог параллельного оператора **<=** с конструкцией **with-select-when**
- Выполняется блок $statements_i$, для которого совпало значение $choise_i$ анализируемого выражения
  - Значения должны быть известны на момент компиляции
  - Значения должны покрывать все возможные варианты expression
  - Несколько значений можно разделять вертикальной чертой |
  - Вариант **others** включает все явно не описанные значения
- В блоке $statements_i$ может содержаться один или более последовательных операторов

```
case expression is
    when choise1 =>
        statements1
    . . .
    when choiseN =>
        statementsN
    when others =>
        statementsN+1
end case;
```

Пустой оператор

```
null;
```

- Не выполняет никаких действий
  - Используется там, где оператор требуется формально

```
process(clk)
begin
    if rising_edge(clk) then
        case mode is
            when “00”    =>  result  <= op_a + op_b;
            when “01”    =>  result  <= op_a – op_b;
            when “10”    =>  result  <= 0;
            when others  =>  null;        -- Описывает все прочие
варианты mode
                                          -- и реализует
режим хранения
        end case;
    end if;
end process;
```

Оператор цикла for

Оператор цикла for

- Оператор **for-loop**
  - Аналог в традиционных языках программирования – оператор **for**
- Выполняет блок statements заданное количество раз
  - На каждой итерации изменяется значение счетчика counter типа integer
  - Переменную counter заранее объявлять не нужно; она является локальной для цикла
- Диапазон range:
  - low **to** high
  - high **downto** low

```
for counter in range loop
    statements
end loop;
```

Оператор цикла for

- Пример – генерация тестовых воздействий в testbench

```
process
begin
    test_value <= “00”;
    test_mode <= “00”;
    for mode in 0 to 3 loop
        for value in 0 to 3 loop
            wait for 10 ns;
            test_value <= test_value + 1;
        end loop;
        test_mode <= test_mode + 1;
    end loop;
end process;
```

Оператор цикла while

Оператор цикла while

- Оператор **while-loop**
  - Аналог в традиционных языках программирования – оператор **while**
  - Условие проверяется перед выполнением блока statements
- Выполняет блок statements, пока условие истинно

```
while condition loop
    statements
end loop;
```

Оператор приостановки процесса wait

Оператор приостановки процесса

- Оператор **wait**
- Используется для приостановки выполнения процесса:
  - **wait for** time – на определенное время
  - **wait until** condition – до выполнения условия
  - **wait on** signals – до определенного события (изменения сигнала)
  - **wait** без параметров – завершение выполнения процесса
- Полная форма оператора:

```
wait on signals until condition for time;
```

- Синтезируемое ожидание фронта синхронизации:

```
wait until rising_edge(clk);
```