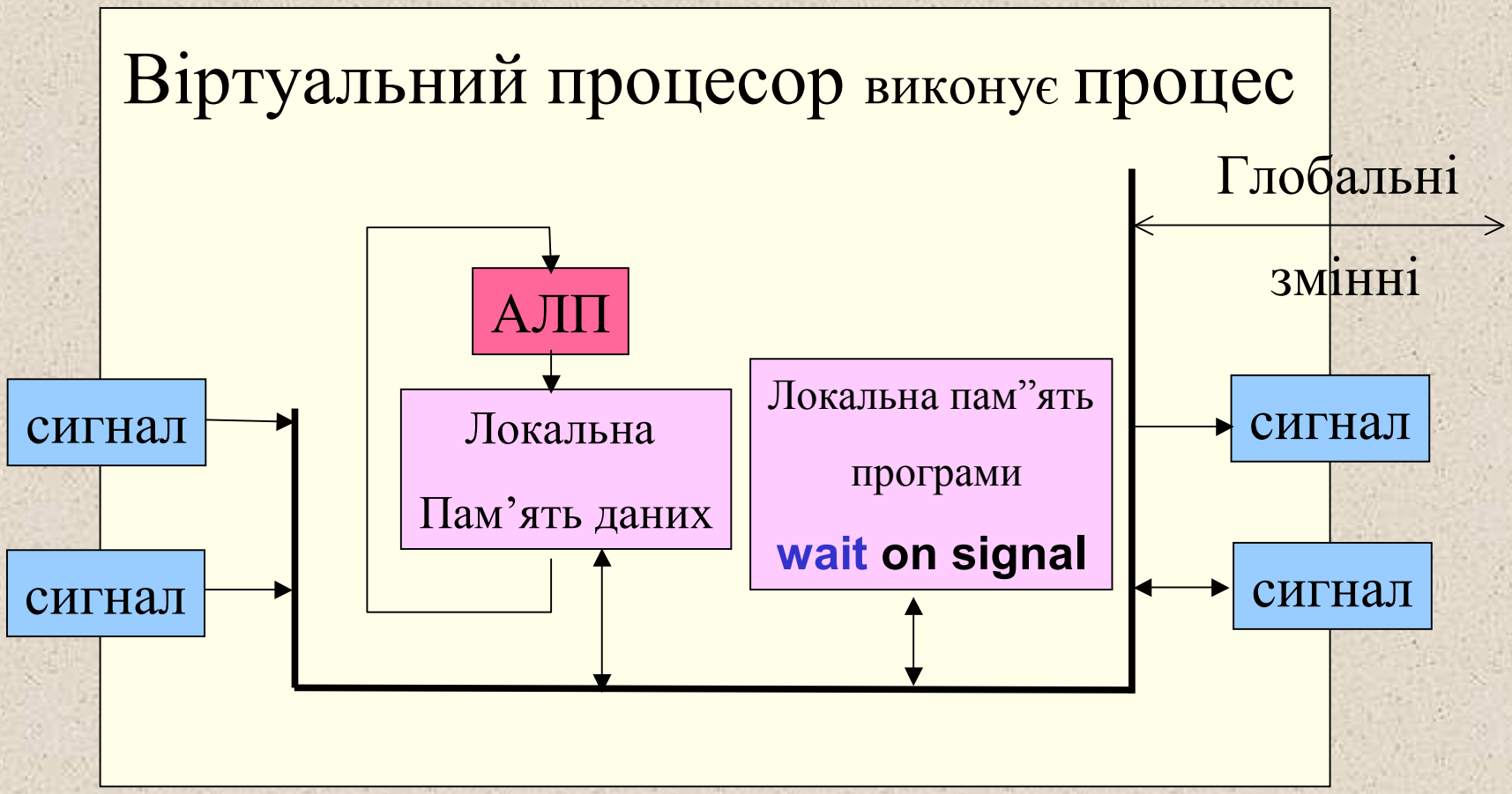
Паралельні оператори в VHDL

Обчислювальна модель для реалізації VHDL

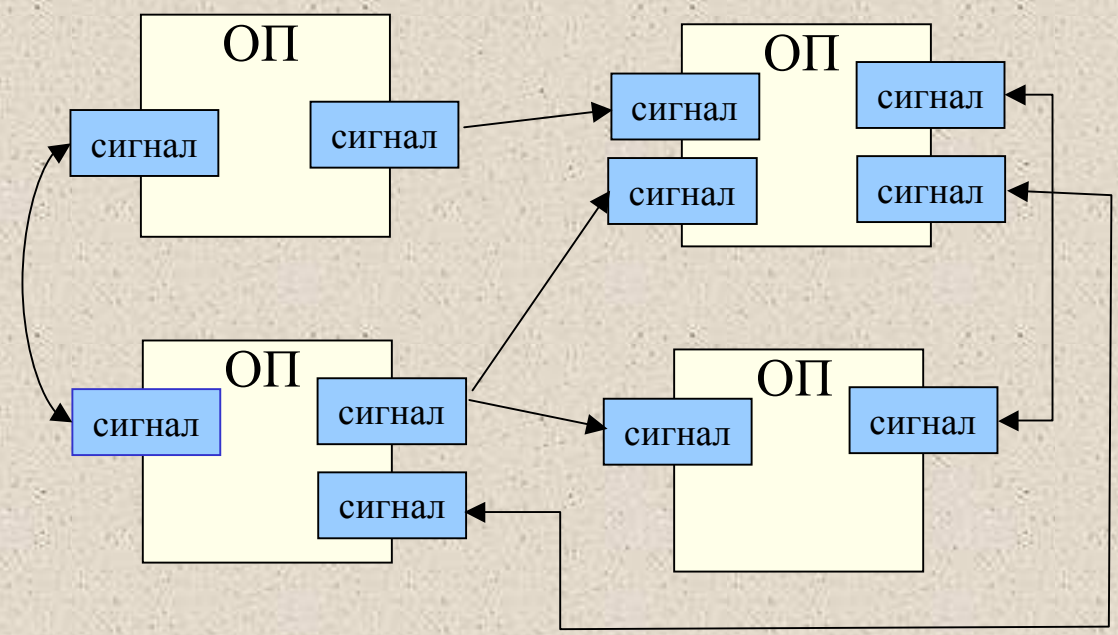


Обчислювальна модель для реалізації VHDL

Консоль

Дискова пам’ять

Пам’ять глобальних змінних



Паралельні оператори в VHDL

- Всі паралельні оператори виконуються *одночасно* під час одного і того самого циклу моделювання.
- Результати паралельних операторів доступні для інших операторів після *закінчення поточного циклу моделювання* .
- Всі параллельні оператори можуть бути перетворені в еквівалентні **процеси** (крім вставки компонента).
- Порядок паралельних операторів в описній частині архітектури - ДОВІЛЬНИЙ

Паралельні оператори в VHDL

- Процес **process**(a) **begin** a<=b;
  end process;
- Присвоювання сигналу A<=B;
- Умовне присвоювання сигналу **when-else**
- Вибіркове присвоювання сигналу **with-select**
- Вставка екземпляра компонента U1: AND(y,b,c);
- Паралельний виклик процедури SORT(A,B);
- Оператор **generate**
- Оператор **block**
- Оператори **assert, report**

Оператор процесу

- Оператор **process** визначає незалежний послідовний процес , який представляє собою поведінку деякої частини проекта.
- Кожен процесс може бути відмічений необов'язковою **міткою**.
- В декларативній частині процеса об'являються підпрограми, типи, підтипи, константи, змінні, атрибути, файли, оператори **use ,** які відносяться тільки до цього процесу.

```
[/мітка/:] process [(список_чутливості)] is
 --декларативна_частина
begin
  --послідовні_присвоювання
end process [/мітка/];
```

Список чутливості

- **Список чутливості** вказує сигнали, зміна яких викликає запуск процеса.
- **Список чутливості** - це компактна форма завдання операторів **wait on** , які є останніми операторами в розділі *послідовні_присвоювання* .
- **Список чутливості** вставляється зразу після слова **process**.
- Процес зі списком чутливості не повинен вміщувати ніяких операторів **wait**. Також якщо в процесі є виклики процедури, то в ній не повинно бути операторів **wait** .

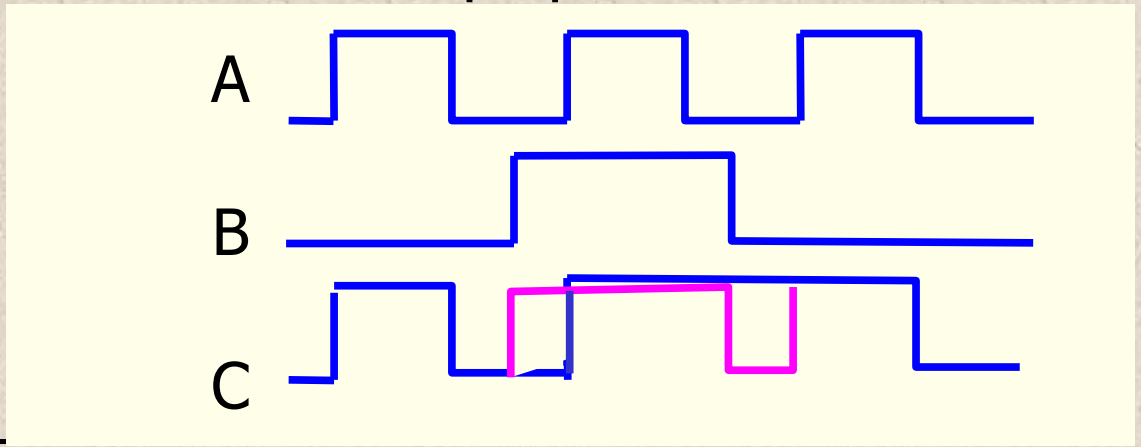
Список чутливості

При моделюванні логічних схем в список чутливості необхідно вносити **усі вхідні сигнали**, інакше моделювання може відрізнятись від бажаного.

Наприклад, процес

```
process(A, B )  begin
    c<= A or B;
  end process;
```

При моделюванні дає графіки:



Присвоювання сигналу

- Має такий самий синтаксис, що і оператор послідовного присвоювання сигналу.
- Еквівалентний оператору процеса, в якому останній оператор **wait** чекає на вхідні сигнали.

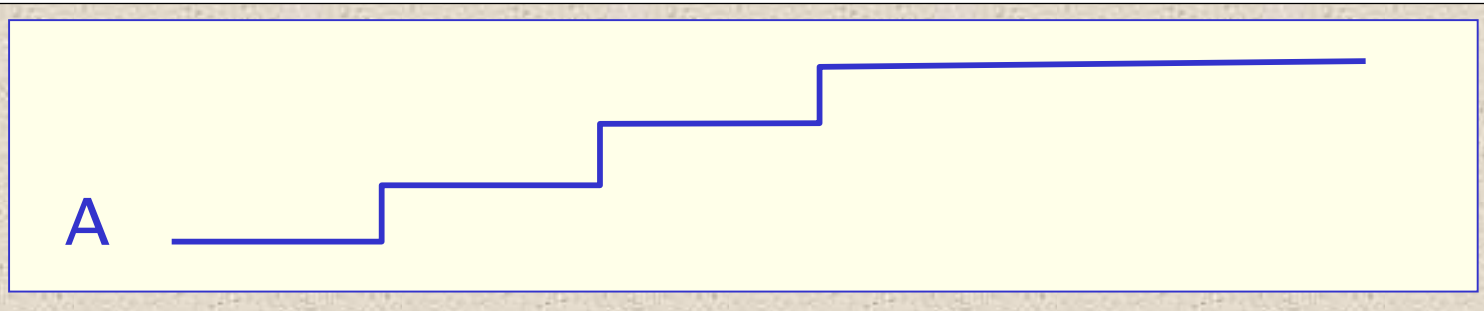
```
    A <= B + C;
-- еквівалентно
    process begin
        A <= B + C;
        wait on A,B;
    end process;
-- або
    process (A,B) begin
        A <= B + C;
    end process;
```

Присвоювання сигналу

при моделюванні

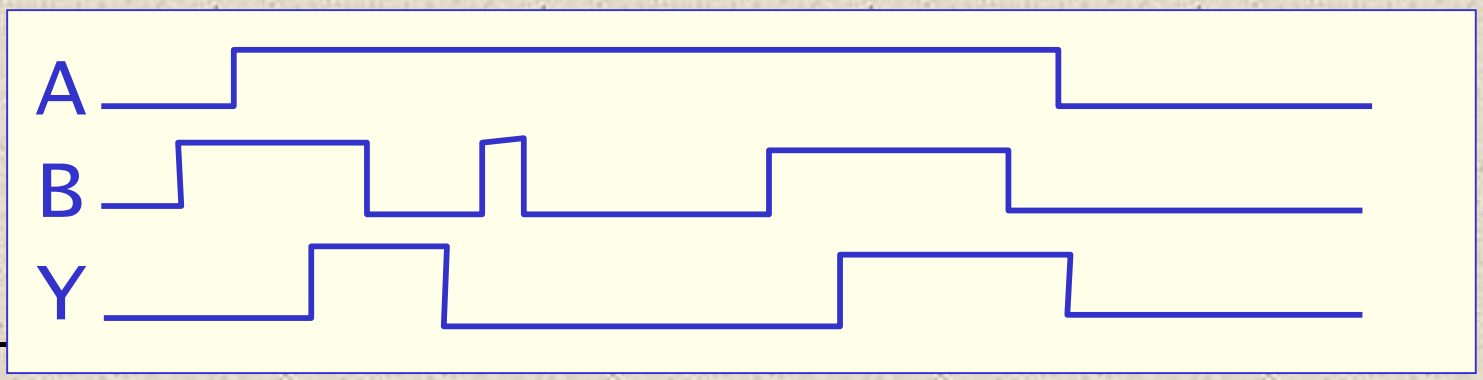
Генерація імпульсів з заданими параметрами

```
A <= 0, 1 after 5 ns, 2 after 10 ns, 3 after 15 ns;
```



Модель елемента AND з затримкою 5 нс, який не пропускає імпульсів коротше 2 нс

```
Y <= reject 2 ns inertial A and B after 5 ns;
```



Умовне присвоювання сигналу

- Нове значення присвоюється сигналу, якщо булевский вираз, що стоїть після слова **when** - істинний. Інакше перевіряється наступний вираз після слова **else** і т.д.
- Множини умов можуть пересікатись.
- Умовне присвоювання сигнала **повинне** закінчуватись безумовним виразом **else**.

```
Architecture \парал\ of \тріст_буф\ is
begin
      \вих_б\ <= \вх_1\ when \видати1\ = '1',
                 \вх_2\ when \видати2\ = '1'
                 else 'Z';
end architecture \парал\;
```

Вибіркове присвоювання сигналу

- **Вибіркове присвоювання сигналу** це паралельний еквівалент послідовної конструкції **case** .
- Всі варіанти вибору повинні бути перелічені, доки не зустрінеться слово **others** як останній варіант вибору.
- Як варіант вибору можуть використовуватись діапазони і переліки.
- Множини варіантів вибору не повинні пересікатися.

```
with \код_оп\ select
\вих_Mux\ <= A when  0 | 1, --перелік
             B when  2 to 5, --діапазон
             C when  6,
             D when  7,
            'Z' when  others;--решта
```

Паралельний виклик процедури

- має такий самий синтаксис, як і послідовний виклик.
- Виконується як оператор процесу, який виконує дії, що і дана процедура, і має в кінці оператор **wait**, що чекає на зміну сигналів – параметрів процедури.
- Після компіляції програма має стільки копій тіла процедури, скільки буде її викликів, тобто всі виклики заміняються відповідними операторами процеса.

```
procedure MY_AND2(signal a,b:in bit;
                  signal y:out bit) is begin
            Y<=a and b;
end procedure;
. . .
-- виклики процедури:
    MY_AND2(a,b,c);
    MY_AND2(c,d,e);
```

Вставка екземпляра компонента

Вставка екземпляра компонента визначає компонент елемента (entity) проекта, в якому той використовується, зв”язує сигнали з портами цього компонента і зв”язує настроювальні константи (generics) з настроювальними константами цього компонента.

/мітка/: [**component**] /ім’я_компонента/
generic map(список_зв’язування_настроювальних_констант);
port map (список_зв’язування_портів_і_сигналів);

/мітка/: **entity** /ім’я_елемента_проекта/[(/ідентифікатор_архітектури/)]
generic map(список_зв’язування_настроювальних_констант);
port map (список_зв’язування_портів_і_сигналів);

/мітка/: **configuration** /ім’я_конфігурації/
generic map(список_зв’язування_настроювальних_констант);
port map (список_зв’язування_портів_і_сигналів);

Вставка екземпляра компонента

Компонент представляє собою пару **об'єкт_проекта(архітектура).** Він задає підсистему, яка може бути використана як деякий екземпляр цієї підсистеми.
Така вставка екземпляра призводить до ієрархічної специфікації.
Вставка екземпляра компонента нагадує вставку мікросхеми в панельку на платі.

Приклад **ассоціативного зв"язування** настроювальних констант, портів і сигналів :

```
Architecture Structural of \АЛП\ is
    signal X,Y,S,C : bit;
    component \напівсуматор\ is
        port (In1, In2 : in bit;
              \сума\, \перенос\: out bit);
    end component \Напівсуматор\;
begin
    U_HS :\напівсуматор\port map (X,Y,S,C);
    . . .
End architecture Structural;
```

Вставка екземпляра компонента

Поіменоване зв'язування дає змогу проводити зв'язування настроювальних констант, портів і сигналів в порядку, що відрізняється від порядку, заданого в декларації компонента. В цьому випадку імена портів повинні вказуватися явно.

Ключове слово **OPEN** ставиться замість сигналу, якщо вихід не підключений.

```
architecture  Structural of \АЛП\  is
   signal X,Y,S,C : bit;
   component \напівсуматор\is
       port (In1, In2 : in bit;
             \сума\, \перенос\: out bit);
   end component  \напівсуматор\ ;
begin
  U_HS:\напівсуматор\ port map (Sum => S,
              Carry=> open, -- нікуди не підключено
             In1=>X ,  In2 =>Y);
      . . .
End architecture Structural;
```

Пряма вставка екземпляра об’єкта проекта

Необов’язково декларувати компонент для того, щоб його вставити - пара entity(architecture) може бути вставлена безпосередньо.
Для прямої вставки, фраза вставки компонента вміщує ім’я елемента проекта (entity) і необов’язкове ім’я архітектури, яка повинна бути використана для реалізації цього елемента.
Резервне слово **entity** повинне стояти за міткою вставленого екземпляра.

```
-- елемент-архітектура, яку вставляють
entity XOR2 is
port (IN1,IN2 : in Bit_vector(0 to 3);
          Out1 : out Bit_vector(0 to 3));
end entity XOR2 ;

architecture XOR_2 of XOR2 is
begin
      OUT <= IN1 xor IN2 after 5 ns;
end architecture XOR_2 ;
```

Пряма вставка екземпляра об’єкта проекта

Якщо не вказане ім’я архітектури при вставці екземпляра, то зв’язується архитектура, що скомпільована останньою, яка відноситься до даного елемента проекту

```
-- вставка екземпляра раніше визначених
-- елемента проекта і його архітектури
entity \приклад\ is
end entity \приклад\ ;

architecture \структура1\ of \прикладмер\ is
signal S1,S2 : Bit_vector(0 to 3);
signal S3 : Bit_vector(0 to 3);

begin
  X1 : entity WORK.XOR2(XOR_2)
              port map (S1,S2,S3);
end architecture \структура1\ ;
```

Вставка об’єкта з настроюванням його параметрів

Синтаксис зв’язування настроювальних констант **GENERIC** аналогічний синтаксису зв’язування сигналів і портів.

Наприклад, об’явлений n-бітний регістр:

```
component RGn is
  generic(n: integer);
  port(CLK:bit;  DI: bit_vector(n-1 downto 0));
       DO: out bit_vector(n-1 downto 0));
end component;
```

Тоді вставка 8-розрядного регістра виглядає як:

```
U_RG8:  RGn generic map(n=>8)  -- призначення розрядності 8
                port map (CLK,
                          DI=>DATA_IN,
                          DO=>DATA_OUT);
```

Оператор GENERATE

Призначений для того, щоб повторити групу паралельних операторів задану кількість разів.

Наприклад, модель регістра зсуву на бібліотечних триггерах:

```
signal t: std_logic_vector(1 to n +1);
…
t(1)<=DI;                    -- вхідний біт
-- ставиться обов'язкова мітка FIFO
FIFO: for i in 1 to n generate

-- в коді після компіляції наступний оператор
-- повториться n разів з різним параметром i
        U_ TT: FD(C=>CLK, D=>t(i), Q=>t(i+1));
   end generate;
DO<=t(n+1);          --вихідний біт
```

Оператор GENERATE

Умовний оператор **GENERATE** призначений для того, щоб згідно зі статичною умовою вставити або ні виділену групу паралельних операторів.

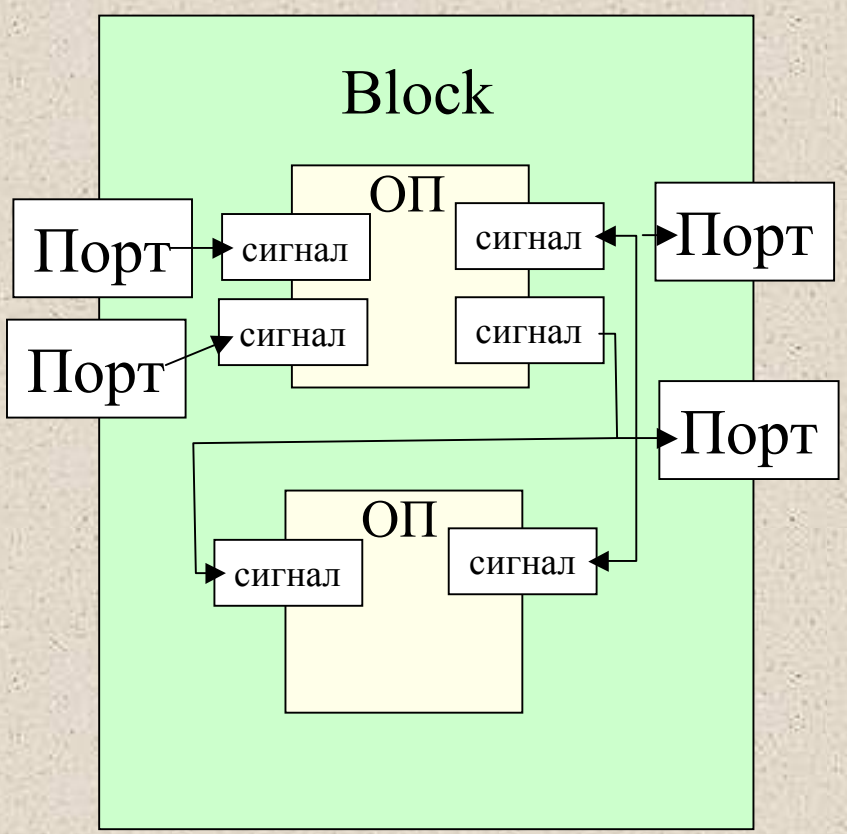
Наприклад, в проект можна за умовою \підключити_PULLUP\=1 підключити до шини DATA_BUS навантажувальні резистори PULLUP:

```
RESn: if \підключити_PULLUP\=1 generate
        RES1:  for i in DATA_BUS'range generate
                  U_ RES: PULLUP(DATA_BUS(i));
        end generate;
end generate;
```

Тут також показано, що оператори **generate** можна вставляти ієрархічно

Оператор **BLOCK**

Призначений для виділення множини паралельних операторів, які представляє як “програму” в програмі.



Оператор **BLOCK**

Блок призначений для:

- створення локальної пам’яті для сигналів
- обмеження області дії (видимості) локальних сигналів
- заміни одного блока іншим (за допомогою конфігурації)
- керованого включення дії сигналів (стримування).

Еквівалентним блоком можна замінити любий компонент

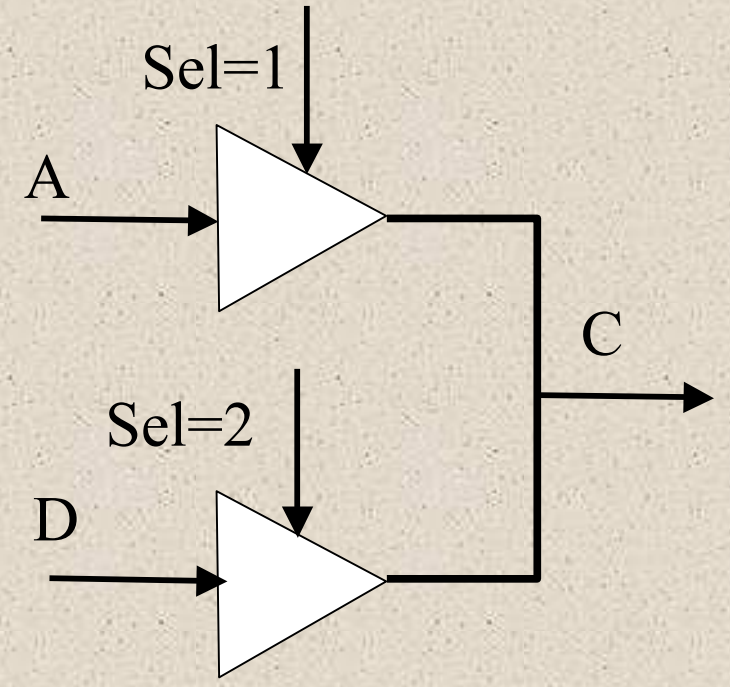
Синтаксис блока:

```
оператор_block::= [\мітка\]: block [\вираз стримування\]  [is]
                        [generic(\об’яви настроювальних констант\]
                  [generic map(\зв’язування настроювальних констант\]
                        [port (\об’ява порту\  {;\ об’ява порту\ });]
                        [port map (\ зв’язування портів\)];
                       {\ об’яви в блоці\}
                  begin
                        {\паралельні оператори\ }
                 end  block [\мітка\];
```

Оператор **BLOCK**

Приклад блоків, що імітують 2 тристабільні буфери:

```
 signal A,B,C:  integer bus:=0;
disconnect C:integer after 2 ns;
...
B1: block (sel = 1) is
          begin
             C <= guarded A;
   end block B1;
B2: block (sel = 2) is
          begin
             C <= guarded D;
   end block B2;
```



NOTE: конструкції з поведінкою, що задається словами **disconnect, bus, register, port, generic** для синтезу **не використовуються**

Паралельний оператор ASSERT і приклад стенду для іспитів

```
entity TB is
end entity TB ;
architecture \структура1\ of TB is
Begin
   U1 : Decoder_bcd port map (Enable,led,bcd);
   Enable<='1';
   bcd <= "00" after 5 ns,
       "01" after 15 ns,
       "10" after 25 ns,
       "11" after 35 ns,
   assert   bcd="00" and   led = "0001"
      or    bcd="01" and   led = "0010"
      or    bcd="10" and   led = "0100"
      or    bcd="11" and   led = "1000"
   report  "Неправильне значення виходa LED"
   severity error;
end architecture \структура1\ ;
```