МІНІСТЕРСТВО ОСВІТИ УКРАЇНИ

**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ**

Методичні вказівки до використання мови VHDL для опису і моделювання цифрових електронних схем

при виконанні лабораторних і курсових робіт з дисциплін "Схемотехніка ЕОМ та елементи і схеми комп'ютерних систем", "ЕОМ та мікропроцесорні системи"

Для студентів спеціальностей 6.0804.00 "Інформаційні управляючі системи та технології", "Інформаційні технології проектування"

Затверджено на засіданні кафедри Систем автоматизації проектування і управління. Протокол № 1 від 30 серпня 1999 р.

КИЇВ КНУБА 1999

Методичні вказівки до використання мови VHDL для опису і моделювання цифрових електронних схем при виконанні лабораторних і курсових робіт з дисциплін "Схемотехніка ЕОМ та елементи і схеми комп‘ютерних систем", “ЕОМ і мікропроцесорні системи”. Для студентів спеціальностей 6.0804.00 “Інформаційні управляючі системи та технології”, “Інформаційні технології проектування” /Укл. О.А.Щербина, Київ: КНУБА, 1999. – 35 с.

Відповідальний за випуск – к.т.н., доцент В.Б.Задоров.
Укладач - к.т.н., доцент О.А.Щербина.
Рецензент – с.н.с. Ю.М.Виноградов.

Передмова. VHDL – мова опису цифрових схем

Розробка сучасних інтегральних мікросхем – є складною інженерно-технічною проблемою, вирішення якої неможливе без широкого застосування систем автоматизованого проектування. Створення подібних систем, як відомо, потребує засобів формального опису структур і функцій об‘єктів проектування. Мова **VHDL** (Very high speed integrated circuits Hardware Description Language) була розроблена у 1980 році в результаті реалізації в США проекту по створенню надшвидкісних інтегральних схем. У 1987 році Інститутом Інженерів з Електрики та Електроніки (IEEE) ця мова була визнана в якості стандарту США. Зараз **VHDL** є найбільш поширеною у світі мовою такого призначення. Вона застосовується у багатьох системах автоматизованого проектування, кількість користувачів яких стрімко зростає.

Пояснюється це тим, що раніше проектуванням інтегральних мікросхем займалася лише обмежена кількість фахівців електронної промисловості, тоді як розробники електронних пристроїв мали можливість застосовувати в своїх проектах лише стандартні мікросхеми з наперед визначеними функціями. Зараз, з появою на ринку мікросхем з програмованою користувачем архітектурою, у розробників електронних схем з‘явилась можливість під кожний конкретний проект самостійно створювати потрібні їм мікросхеми. Це суттєво підвищує якість, знижує вартість і змінює характер самого процесу проектування. Наприклад, задачі, що вирішувалися раніше на рівні розробки структури та вибору елементної бази пристрою, що проектується, зараз можуть бути вирішені на рівні формального опису його структури чи поведінки, комп‘ютерного моделювання та програмування мікросхем.

Дані методичні вказівки присвячені вивченню основ мови **VHDL**. Вони не претендують на вичерпне висвітлення стандарту та прийомів програмування цією мовою. Їх мета – у найбільш спрощеній формі ознайомити студентів з основами операторами та поняттями мови **VHDL** у об‘ємі, достатньому для виконання передбачених учбовим планом лабораторних та курсових робіт з дисциплін *"Схемотехніка ЕОМ та елементи і схеми комп‘ютерних систем"* та *"ЕОМ і мікропроцесорні системи"*. Студенти, що зацікавились даною тематикою, зможуть самостійно вдосконалювати свої знання по **VHDL** під час дипломного проектування та у подальшій професійній діяльності.

Перший розділ роботи містить прості приклади, розгляд яких дозволяє швидко ознайомитись з основними принципами опису схем на **VHDL**. У другом і третьому розділах елементи мови **VHDL** викладені більш детально.

Укладач висловлює вдячність співробітникам фірми **Aldec** (США) та асоціації **Aldec-Україна** з дозволу і за сприянням яких підготовлена ця робота. Зокрема ряд прикладів запозичено з довідкової системи та документації до програмного продукту **Active-VHDL** фірми **Aldec**.

1. Основні поняття мови VHDL

Опис будь-якої схеми, або її фрагменту складається з двох частин. Перша, що називається *сутністю (entity)*, містить опис зовнішнього інтерфейсу схеми (перелік входів, виходів тощо). Друга називається *архітектурою (architecture)* і містить опис, що визначає внутрішню будову та функціонування схеми.

1.1 Опис сутності

Опис сутності починається ключовими словами ***ENTITY ... IS.*** Він містить *ім'я сутності* і опис її зовнішніх виводів, що називаються *портами*. Крім того, опис може включати інші зовнішні параметри, такі як часові і температурні залежності тощо. Завершується опис ключовим словом ***END,*** за яким (для зручності сприйняття тексту програми і додаткового контролю коректності блочної структури) бажано вказати реквізити блоку, що завершується даним ***END***. В даному випадку це *entity* та *ім'я сутності*:

ENTITY *<ім'я сутності>* ***IS***
*<вхідні і вихідні порти>****;***
*[< фізичні та інші параметри>****;]***
END *[[entity] ім'я_сутності]*

1.2 Опис архітектури

Опис архітектури може виконуватись двома способами:

- як опис структури схеми *(structural)*, тобто схеми з'єднань її складових елементів – схем нижчого ієрархічного рівня, аж до рівня з'єднання вентилів *(dataflow)*;
- як опис поведінки схеми *(behavioral)*.

Опис архітектури починається ключовими словами ***ARCHITECTURE ... OF ... IS***, між якими вказується ім'я архітектури та ім'я сутності, якій вона відповідає. Далі вказуються декларації та тіло архітектури.

ARCHITECTURE *<ім'я архітектури>* ***OF*** *<ім'я сутності>* ***IS***
<декларації типів, сигналів, констант та ін.>
BEGIN
<тіло архітектури>
END *[[architecture] <ім'я_архітектури>]*

1.3 Приклад опису схеми D-тригера

Більш детально ці та інші конструкції мови **VHDL** будуть розглянуті нами пізніше, а поки що наведемо приклад опису схеми D-тригера, показаної на рис.1

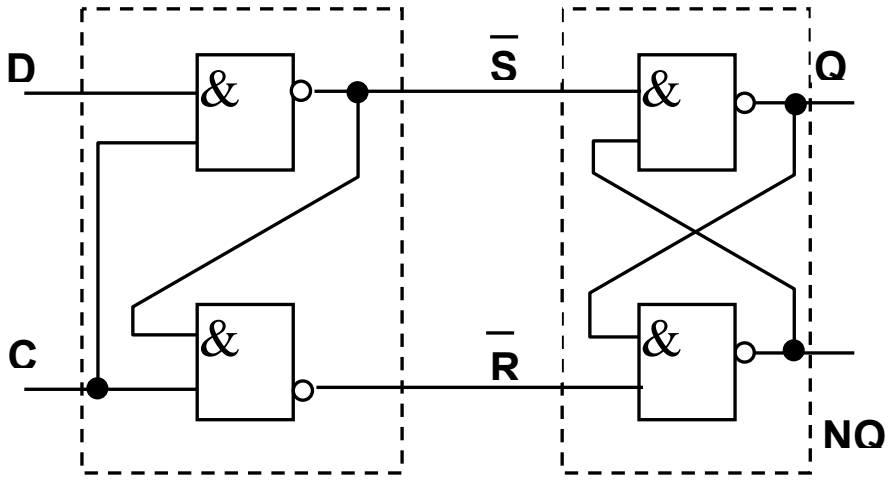

Рис. 1

1.3.1 Приклад опису сутності

Сутність схеми, показаної на рис. 1, може бути описана так:

```
ENTITY d_flipflop IS
   PORT ( D, C : IN bit;
            Q, NQ: OUT bit );
END ENTITY d_flipflop;
```

Цей опис свідчить, що схема на ім‘я *d_flipflop* має чотири зовнішні виводи (порти), з них два входи *(IN)*: *D* та *C* ; і два виходи *(OUT): Q* та *NQ*. Усі порти належать до типу *bit,* тобто сигнал на їх виводах може набувати значень *‘0’* і *‘1’*.

1.3.2 Приклад опису архітектури через опис поведінки

Як відомо, цей тригер працює так, що при *C = 1*, сигнал на виході *Q* повторює значення сигналу на вході *D*, а при *C = 0*, тригер зберігає попередній стан і сигнал на виході *Q* не змінюється. Сигнал на виході *NQ* є інверсією сигналу на виході *Q*.

Цю поведінку тригера можна описати так:

```
ARCHITECTURE behaviour OF d_flipflop IS
BEGIN
   PROCESS (C,D)
   BEGIN
      IF  C = '1'  THEN
          Q <= D  AFTER 5 ns;
          NQ <= not D AFTER 5 ns;
      END IF;
   END PROCESS;
END ARCHITECTURE behaviour;
```

Оператор *PROCESS (C,D)* вказує на те, що робота схеми описується процесом, де зміна сигналів на виходах можлива лише внаслідок зміни вхідних сигналів *C* або *D*. Оператор *IF* визначає, що при *C = '1',* сигнал *Q* набуває значення *D,* а *NQ* – інверсії *D*. Ключове слово *AFTER* вказує на те, що вказані зміни значень на виходах *Q, NQ* відбувається не миттєво, а з затримкою на 5 наносекунд.

1.3.3 Приклад опису архітектури на вентильному рівні

Схему на рис. 1 можна розділити на дві складові, що виділені пунктирними лініями:

- схему управління на ім‘я *contr_unit* з входами *DC* та *CC* і виходами *SC* і *RC,*
- асинхронний *RS*-тригер на ім‘я *rs_ff* з інверсними входами *SFF, RFF* і виходами *QFF, NQFF*.

Тут ми дещо змінили імена портів складових елементів, щоб відрізняти їх від імен портів схеми в цілому. Сутності складових частин та їх архітектури на вентильному рівні описані нижче.

```
ENTITY contr_unit IS
PORT ( CC, DC: IN bit;
       SC, RC: OUT bit);
END contr_unit ;

ARCHITECTURE dataflow OF contr_unit IS
BEGIN
    SC <= DC nand CC  AFTER 2 ns;
    RC <= (DC nand CC) nand CC  AFTER 2 ns;
END dataflow;

ENTITY rs_ff IS
  PORT ( RFF, SFF: IN bit;
    QFF, NQFF: BUFFER bit );
END rs_ff;

ARCHITECTURE dataflow OF rs_ff IS
BEGIN
   QFF <= SFF nand  NQFF  AFTER 2 ns;
   NQFF <= RFF nand  QFF  AFTER 3 ns;
END dataflow;
```

Зазначимо, що *nand* – є позначення операції *І-НЕ*; ***BUFFER*** – є позначення режиму порту виходу, значення якого можна зчитувати, тобто використовувати нарівні з портами входів під час опису архітектури[1]. Для стабільної роботи тригера тут зумисне вказано різний час затримки у різних його плечах.

1.3.4 Приклад опису архітектури через опис структури

Тепер опишемо сутність та архітектуру схеми на рис. 1 у вигляді структури, що складається з описаних вище схеми управління ***contr_unit*** та асинхронного *RS*-тригера *rs_ff*

.

```
ENTITY d_ff IS
   PORT (  D, C : IN bit;
   Q, NQ: BUFFER bit   );
END entity d_ff;

ARCHITECTURE structural OF d_ff   IS
      SIGNAL   P1, P2 : bit;
 BEGIN
  cu: ENTITY work.contr_unit(dataflow)
       PORT MAP (DC => D, CC => C, SC => P1, RC => P2 );
  c2: ENTITY work.rs_ff(dataflow)
       PORT MAP (P1,P2,NQ,Q);
END architecture structural;
```

[1] Якби *QFF, NQFF* були звичайними виходами (*OUT*, а не *BUFFER*), то компілятор виявив би помилку у формулах *SFF nand NQFF* та *RFF nand QFF*, бо сигнали портів *OUT* не можуть служити аргументами для обчислення інших значень. Більш детальна інформація про режими портів наведена в розділі 3.1.2.

Тут *SIGNAL P1, P2: bit;* – декларація внутрішніх сигналів, що застосовуються для з'єднання виходів схеми управління зі входами асинхронного *RS*-тригера. Для включення до складу даної схеми раніше описаного фрагменту використано конструкцію

*мітка: **ENTITY** ім'я_бібліотеки.ім'я_сутності(ім'я_архітектури).*

Відповідність між портами раніше описаного фрагменту та сигналами даної схеми задається конструкцією *PORT MAP*. Її можна застосовувати у двох варіантах. Так у нашому прикладі:

- для сутності *contr_unit* ця відповідність задана через імена: порт *DC* *contr_unit* відповідає у даній схемі портові *D* і т.д.;
- для сутності *rs_ff* відповідність задана через послідовність, у якій сигнали перелічені у списку *PORT MAP* і списку портів опису сутності *rs_ff* : перший елемент – сигнал *P1* відповідає першому елементові списку портів, тобто *RFF* і т.д.

2 VHDL як мова програмування

2.1 Лексеми VHDL

2.1.1 Коментарі

Коментар в **VHDL** розпочинається двома знаками мінус "--" і закінчується в кінці того ж рядка, наприклад:

```
-- Тут увесь рядок – коментар
a := 0 -- тут залишок рядка – коментар
```

2.1.2 Ідентифікатори

Ідентифікаторами називаються визначені програмістом імена. Як і у інших мовах програмування, ідентифікатор **VHDL** – це послідовність букв латинського алфавіту, цифр та знаків підкреслення, що розпочинається з букви. Ідентифікатори не повинні співпадати з зарезервованими словами мови **VHDL**. Ця мова нечутлива до регістру літер, наприклад *slovo, SLOVO і SloVo* – це один і той самий ідентифікатор. Кількість символів у ідентифікаторі не обмежується, але ідентифікатор повинен поміщатися у одному рядку.

Починаючи з версії **VHDL'93** у мові дозволено використовувати і так звані розширені ідентифікатори. Це послідовність будь-яких символів, серед яких можуть бути зарезервовані слова, пробіли, літери кирилиці тощо, що обмежується з обох боків символом антислеш - \, наприклад: *\for\, \IF\, \сигнал №1*. Якщо до складу ідентифікатора треба включити сам антислеш, то при написанні його слід подвоїти, наприклад *\файл C:\\main\\m1.doc*. Символи верхнього і нижнього регістрів у розширених ідентифікаторах розрізняються, тобто *\slovo\, \SLOVO\ і \SloVo* – це три різні ідентифікатори.

2.1.3 Числові константи

Десяткові числа з крапкою належать до типу *float,* без крапки – до типу *integer*. І ті, і інші можуть містити експоненту *E* та символи підкреслення, якими можна відділяти групи розрядів для зручності читання.

Приклади десяткових констант типу *float:*

```
1.25,   0.000_1,  5.25 E7
```

Приклади десяткових констант типу *integer*:

18, 524_758_000, 1E6

Недесяткові числа записуються у вигляді:

<основа системи числення>#<число у системі числення з даною основою>

При цьому значення основи системи числення (від 2 до 16), а також значення експоненти записуються тільки у десятковому вигляді, наприклад:

2#0000_0101 = 5, 16#1.0A E2 = 16#10A = 266

2.1.4 Символьні літерали

Символьний літерал – це будь-який символ ASCII, обмежений з обох боків апострофами, наприклад:

'A', '5', '+', '', ''.

2.1.5 Рядкові літерали

Рядковий літерал – це набір символів ASCII, взятий у лапки. Щоб включити у рядковий літерал сам символ лапок, його там треба повторити двічі, наприклад:

"Лев Толстой", "Роман ""Війна і світ"".".

2.1.6 Бітові рядки

Для запису бітового рядка спочатку вказують букву, що визначає основу системи числення: *B* – 2 , *O* – 8, *X* – 16, а за нею в лапках відповідно двійкове, вісімкове чи шістнадцяткове число. Як приклад нижче наводиться декілька варіантів запису рядка із восьми бітів:

B"10011100" = B"1_001_110_100" = O"1164" = B"1001_1100" = X"9C"

2.2 Типи даних

VHDL має багато скалярних типів даних та засоби для утворення на їх основі складених типів.

Скалярні типи включають числові, фізичні величини та перелічувальні типи. Є також велика кількість наперед визначених стандартних типів.

Складені типи включають масиви та записи. **VHDL** також має тип доступ (*access*) та файловий тип (*files*), які в даній роботі ми розглядати не будемо.

Визначення типу проводиться директивою:

***TYPE** <ідентифікатор> **IS** <опис типу>;*

2.2.1 Цілочисельні типи

Це типи, для яких задається діапазон значень:

***TYPE** <ідентифікатор> **IS RANGE** <значення від> **TO** |**DOWNTO** <значення до>;*

Максимальний діапазон від -2147483647 до +2147483647 визначено як тип *integer*.

Приклади:

***TYPE** byte_int **IS RANGE** 0 **TO** 255;*

***TYPE** signed_word_int **IS RANGE** -32768 **TO** 32767;*

***TYPE** bit_index **IS RANGE** 31 **DOWNTO** 0;*

2.2.2 Типи фізичних величин

Цими типами описують різноманітні фізичні величини: час, напругу, відстань тощо. Запис даних цього типу складається з числа за яким вказується одиниця виміру. Наприклад для запису часового проміжку тривалістю одна хвилина можна скористатись однією з наступних констант, що належать до стандартного типу *TIME* (час):
1 min, 60 sec, 60_000 ms, 60_000_000 us , 60_000_000_000 ns і т.д.

Типи фізичних величин, що не віднесені до числа стандартних, мають бути явно описані в тексті програми наступним чином:

TYPE *<ідентифікатор>* ***IS RANGE*** *<значення від>* ***TO*** **|*DOWNTO*** *<значення до>*
UNITS
<базова одиниця виміру>;
{<похідна одиниця>;}
END UNITS *[<ідентифікатор>];*

Наприклад для опису ємності конденсаторів можна так визначити тип *CAPACITY:*

```
TYPE CAPACITY IS RANGE 0 TO 1E5
  UNITS
    pF;                -- пікофаради
    nF = 1000 pF;      -- нанофаради
    uF = 1000 nF;      -- мікрофаради
    mF = 1000 uF;      -- міліфаради
    F  = 1000 mF;      -- фаради
  END UNITS CAPACITY;
```

2.2.3 Тип з рухомою крапкою

VHDL має стандартний, наперед визначений тип ***REAL****,* що включає дійсні числа в діапазоні від -1E-38 до +1E38. Крім нього можна задавати типи з рухомою крапкою для чисел з обмеженим діапазоном значень так само, як це робиться для цілочислельних типів, наприклад:

```
TYPE signal_level IS RANGE -10.00 TO +10.00;
TYPE probability IS RANGE 0.0 TO 1.0;
```

2.2.4 Перелічувальні типи

Перелічувальний тип – це впорядкований набір відмінних один від одного ідентифікаторів або символів, наприклад:*:*

```
TYPE logic_level IS (low, high, unknown);
TYPE octal_digit IS ('0','1','2','3','4','5','6','7');
```

До стандартних, наперед визначених перелічувальних типів належить зокрема

```
TYPE boolean IS (false, true);
TYPE bit is ('0','1');
TYPE character IS (... koi-7...)
```

та інші.

Символи *'0'* та *'1'* є одночасно членами типів *bit* та *character*. **VHDL** сам визначає тип *'0'* та *'1'* в залежності від контексту, де вони використовуються.

2.2.5 Масиви

Масив у мові **VHDL** – це набір індексованих елементів одного типу. Масиви можуть бути одновимірні (з одним індексом) та багатовимірні (з багатьма індексами). Масиви також можуть бути обмеженими та необмеженими. Кожний масив має бути задекларований :

***TYPE** <ідентифікатор масиву> **IS ARRAY** <межі для індексів> | **RANGE** <> **OF** <тип>;*

Наприклад:

***TYPE** word **IS ARRAY** (0 **TO** 15) **OF** bit;*
***TYPE** word **IS ARRAY** (15 **DOWNTO** 0) **OF** bit;*
***TYPE** memory **IS ARRAY** (address) **OF** word;*
***TYPE** Byte_Vector **IS ARRAY** (POSITIVE range 1 to 8, POSITIVE range 1 to 4) **OF** Byte;*
***TYPE** vector **IS ARRAY** (integer **RANGE** <>) **OF** real;*

Тут *word* є тип масивів де граничні значення індексу *0* та *15* вказані безпосередньо. Якщо першим вказане менше значення, то розділяються вони словом *TO*, якщо більше – то *DOWNTO*. У типі масивів *memory* індекс може набувати будь-яких значень, що належать до типу *address*. *Byte_Vector* є тип двомірних масивів 8 х 4, складених з елементів типу *Byte*. У типі масивів *vector* граничні значення індексів будуть визначені пізніше, а поки що їх місце займає символ '<>'.

VHDL має два наперед визначені необмежені типи масивів:

***TYPE** string **IS ARRAY**(positive **RANGE** <>) **OF** character;*
***TYPE** bit_vector **IS ARRAY**(natural **RANGE** <>) **OF** bit;*

2.2.6 Записи

Запис є набором іменованих елементів однакового або різних типів. Тип запису описується так:

TYPE** <ідентифікатор_запису> **IS RECORD
<ім'я_елементу>:<тип_елементу> {,<ім'я_елементу>:<тип_елементу>}
***END RECORD** [<ідентифікатор_запису>];*

Приклад:

TYPE** list **IS RECORD** name:string, quantity:positive; **END RECORD;

При посиланні на поля ім'я запису та ім'я поля відокремлюються крапкою, наприклад: *list.name, list.quantity*.

2.2.7 Підтипи

Підтип виступає як опис підмножини елементів відповідного базового типу:

***SUBTYPE** < ім'я підтипу> **IS** < ім'я базового типу> **RANGE** <діапазон>;*

Наприклад:

***SUBTYPE** digits **IS** character **RANGE** '0' **TO** '9'*

VHDL має два наперед визначених підтипи:

***SUBTYPE** natural **IS** integer **RANGE** 0 **TO** 2147483647;*
***SUBTYPE** positive **IS** integer **RANGE** 1 **TO** 2147483647;*

2.3 Об'єкти

В **VHDL** є три класи об'єктів: константи, змінні та сигнали.

2.3.1 Константи

Константа – це об'єкт, значення якому надається в момент його створення декларацією

***CONSTANT** <ідентифікатор>:<тип> [:=<значення>];*

і в подальшому не змінюється.

Допускається створення константи без присвоєння їй значення, наприклад при декларуванні пакетів. В такому випадку значення буде присвоюватись у відповідному тілі пакета.

Приклади констант:

***CONSTANT** pi: real:=3.1415;*
***CONSTANT** delay: Time:=5 ns;*

2.3.2 Змінні

Змінна – це об'єкт, значення якому може надаватися в момент її створення декларацією

***VARIABLE** <ім'я змінної >:<тип>[:=<значення>];*

і яке у подальшому може змінюватись.

Якщо у декларації змінної її значення відсутнє, то змінна набуває значення за замовчуванням. Для змінних скалярного типу ним є найменше з допустимих значень, або перший елемент списку перелічуваного типу, або найменше значення із зростаючого діапазону *TO*, або найбільше значення із спадаючого діапазону *DOWNTO*. Зокрема змінній типу *bit* надається значення *'0'*, а змінній типу *boolean* – значення *false*.

Якщо тип змінної складений, то значення за замовчуванням кожному його елементу присвоюється відповідного базового типу.

Після створення об'єкту, йому чи його частині можна надати інше, альтернативне ім'я, що називається аліасом чи синонімом і використовуються нарівні з основним іменем. Це досягається за допомогою декларації

***ALIAS** <альтернативне ім'я>: <тип> **IS** <ім'я змінної>.*

У наведеному нижче прикладі декларується 16-бітова змінна *AX*. Крім того її 8 старших бітів отримують альтернативне ім'я *AH*, в 8 молодших – *AL*.

***VARIABLE** AX: bit_vector (15 **DOWNTO** 0);*
***ALIAS** AH: bit_vector (7 **DOWNTO** 0) **IS** AX (15 **DOWNTO** 8);*
***ALIAS** AL: bit_vector (7 **DOWNTO** 0) **IS** AX (7 **DOWNTO** 0);*

2.3.3 Сигнали

Сигнали в мові **VHDL** відповідають в реальній схемі провідникам. Сигнали мають багато спільного зі змінними. Так декларація сигналу, що має передувати його використанню, має синтаксис аналогічний декларації змінної:

***SIGNAL** < ім'я сигналу>:<тип>[:=<початкове значення>];*

Більш того, сигнали мають ті самі типи, що і змінні, і так само, як і для змінної, початкове значення сигналу може бути надано в декларації, а якщо воно там відсутнє, то значення за замовчуванням вибирається за тими ж правилами, що і для змінних. Проте принципова відмінність сигналу від змінної полягає в тому, що змінна характеризується лише своїм поточним значенням і "не пам'ятає" які значення вона мала у

минулому, тоді як для сигналу усі минулі значення і значення, що плануються на майбутнє, зберігаються у пам‘яті і є доступними через атрибути.

Отже сигнал відрізняється від змінної наявністю ще одного виміру – часового. Для ілюстрації наведемо оператор, що надає значення сигналові *clk*.

*clk <= '1' **AFTER** 5 ns, '0' **AFTER** 10 ns, '1' **AFTER** 15 ns, '0' **AFTER** 20 ns;*

Цей сигнал набуває значення *'1'* після 5 наносекунд, *'0'*- після 10 нс., знову *'1'* - після 15 нс. і знову *'0'*- після 20 нс.

2.3.4 Атрибути

Багато додаткової інформації про об’єкти **VHDL** можна одержати з їх атрибутів. Значення атрибуту об‘єкту можна одержати, вказавши після імені об‘єкту апостроф та ім‘я атрибуту:

< ім’я об‘єкту>’< ім’я атрибуту>.

2.3.4.1 Атрибут визначений для усіх типів

Атрибут *base* дає базовий тип об‘єкту на основі якого побудовано даний тип. Наприклад, якщо

***TYPE** word **IS ARRAY** (0 **TO** 15) **OF** bit;*

то *word’base* повертає значення *bit*.

2.3.4.2 Атрибути скалярного типу та його підтипів

Якщо *S* – скалярний тип, або його підтип то:

S’left - ліва межа S;
S’right - права межа S;
S’low - нижня межа S;
S’high - верхня межа S;

Наприклад, якщо

variable w1 : word := "10001111_00001110";

то *w1'left* та *w1'low* повертають значення *'1'*. Зауважимо, що для зростаючого діапазону вважається, що *T’left = S’low, S’rigth = S’high*, а для спадаючого діапазону – *T’left = S’high, S‘right = S’low.*

S’Ascending – значення типу boolean, що дорівнює true для зростаючого діапазону і false – для спадаючого;

Наприклад, якщо

***TYPE** New_Range **IS RANGE** 1 to 10*;

то *New_Range’Ascending* повертає *true.*

S’Image(x) – значення типу string, що є текстовим представленням значення x типу S;
S’Value(x) – значення типу S’base, що відповідає текстовому (string) представленню x.

Останні два атрибута використовуються для перетворення типів. Наприклад для перетворення числа *x* типу *integer* у тип *real* можна скористатися формулою *Real'Value(Integer'Image(x))*, що використовує тип *string* як проміжний.

2.3.4.3 Атрибути дискретних типів фізичної величини та їх підтипів

Якщо *T* – дискретний тип або тип фізичної величини, *X* – є членом *T*, а *N* – *integer*, то:

T'pos(X) - *позиція X в T;*
T'val(N) - *значення позиції N в T;*
T'leftof(X) - *значення позиції зліва від X в T;*
T'rightof(X) - *значення позиції справа від X в T;*
T'pred(X) - *значення позиції нижче від X в T;*
T'succ(X) - *значення позиції вище від X в T;*

Для зростаючого діапазону *T'leftof(X)=T'pred(X), T'rightof(X)=T'succ(X).*

Для спадаючого діапазону *T'leftof(X)=T'succ(X), T'rightof(X)=T'pred(X).*

Наприклад, якщо

***TYPE** New_Values **IS** (Low, High, Middle);*

то *New_Values'Pred(High)* повертає *Low,* а *New_Values'Pos(High)* повертає *2.*

2.3.4.4 Атрибути типів масивів та об'єктів масивів

Якщо *A* – тип масивів чи об'єкт масивів, а *N* – *integer* в діапазоні від *1* до кількості вимірів масиву *A*, то:

A'left(N) - *ліва межа індексу N-ої вимірності масиву A;*
A'right(N) - *права межа індексу N-ої вимірності масиву A;*
A'low(N) - *нижня межа індексу N-ої вимірності масиву A;*
A'high(N) - *верхня межа індексу N-ої вимірності масиву A;*
A'range(N) - *діапазон індексів N-ої вимірності масиву A;*
A'reverse_range(N) - *реверсія індексів N-ої вимірності масиву A;*
A'length(N) - *довжина діапазону індексів N-ої вимірності масиву A.*
A'Ascending(N) -*true, якщо індекс N-ої вимірності масиву зростаючий, інакше – false .*

Наприклад, якщо

***TYPE** word **IS ARRAY** (0 **TO** 15) **OF** bit;*

то *word'left(1)* повертає *1*, *word'right(1)* повертає *15*, а *word'reverse_range(1)* повертає *15 **DOWNTO** 0*. Зауважимо, що оскільки *word* є тип одновимірних масивів, то номер вимірності може бути опущений, тобто ті ж самі результати можна одержати, записавши *word'left, word'right, word'reverse_range.*

2.3.4.5 Атрибути сигналів

S' Delayed(t) – *Неявний сигнал, еквівалентний сигналу S, але затриманий на час t.*
S'Stable(t) – *Неявний сигнал, що має значення True, якщо ніяка подія не відбулася на S за час t , False – у протилежному випадку.*
S'Quiet(t) – *Неявний сигнал, що має значення True, якщо ніякої транзакції не відбулося на S за час t ,*
S'Transaction – *Неявний сигнал типу Boolean, значення якого змінюється у тих циклах моделювання, де відбуваються транзакції на S (сигнал S стає активним).*
S 'Event – *True, якщо відбулася зміна сигналу S у поточному циклі моделювання, False – у протилежному випадку.*
S 'Active – *True, якщо в поточному циклі моделювання був задіяний сигнал S, False – у протилежному випадку.*

S'Last_event – *Кількість часу, що минув з останньої зміни сигналу S, якщо ніяких змін не відбулося, то повертається Time'High.*

S'Last_active – *Кількість часу, що минув з останньої операції, де був задіяний сигнал S. Якщо ніяка операція не виконувалась, то повертається Time'High.*

S 'Last_value – *Попереднє значення сигналу S, тобто те значення, яке він мав перед останньою операцією.*

S'Driving – *True, якщо процес управляється сигналом S або одним з його елементів. False – у протилежному випадку.*

S'Driving_value – *Поточне значення драйвера для S у процесі, що містить присвоєння S.*

2.3.4.6 Атрибути імен сутностей

E'Simple_name – *Рядок (string), що представляє просте ім'я, символьний літерал або символ оператора, визначений в оголошенні E.*

E'Path_name – *Рядок, що описує шлях через ієрархію проектів, від корінної сутності або пакета до E.*

E'Instance_name – *Рядок, що описує шлях через ієрархію проектів, від кореневої сутності або пакета до E, але включаючий імена об'єкта й архітектури, кожної складової в цьому шляху.*

Зауважимо що не всі наведені атрибути підтримуються засобами синтезу.

2.3.5 Атрибути, визначені користувачем

Крім розглянутих вище заздалегідь визначених атрибутів **VHDL** дозволяє використовувати атрибути, визначені користувачем. Для цього потрібно задекларувати атрибут та дати його визначення за допомогою операторів

***ATTRIBUTE** <ім'я атрибуту>:<тип>;*

***ATTRIBUTE** <ім'я атрибуту> **OF** <ім'я об'єкту>:<клас об'єкту> **IS** <вираз>;*

Перший з них декларує ім'я атрибуту та тип значення, яке йому буде надаватися. Другий оператор вказує до якого об'єкту застосовується цей атрибут і за допомогою якого виразу визначатиметься його значення.

Нижче наведено приклад, де декларується атрибут *Pin_code*, як такий що набуває значень типу *Positive*, і зокрема для сигналу *Gnd* цей атрибут дорівнюватиме 7.

***ATTRIBUTE** Pin_code: Positive;*
***ATTRIBUTE** Pin_code **OF** Gnd: **SIGNAL IS** 7;*

2.4 Вирази та оператори

Вирази в **VHDL** включають імена об'єктів, літералів, знаки операцій, виклики функцій тощо. Нижче наведені знаки операцій **VHDL** в порядку зменшення їх пріоритету:

***** ABS NOT***
**** / MOD REM***
+ (унарний) – (унарний)
+ – &
SLL SRL SLA SRA ROL ROR
= /= < <= > >=
AND OR XOR NAND NOR NXOR NOT

Операції ***ABS*** (абсолютне значення), ******* (піднесення до степеню), ****, /, +, –*** виконуються з числовими типами даних. Другий операнд операції піднесення до степеню повинен мати цілочисельний тип, причому якщо він від'ємний, то перший операнд повинен бути числом з рухомою крапкою. Операції ділення без остачі (***REM***) та остача від ділення (***MOD***) працюють лише з цілими.

Логічні операції ***AND, OR, XOR, NAND, NOR, NXOR*** і ***NOT*** виконуються над значеннями типу *bit* або *boolean*, а також над одновимірними масивами, що складаються з елементів вказаних типів. Операнди-масиви повинні мати однакову розмірність. Операція виконується над одноіменними елементами масивів і її результатом операції є масив того ж типу і розмірності.

Зліва від знаку операції зсуву ***SLL, SRL, SLA, SRA, ROL, ROR*** вказується масив типу *bit* або *boolean*, справа – ціле число, що визначає на скільки елементів виконуватиметься зсув елементів у масиві. Якщо це число від'ємне, то зсув відбуватиметься у протилежному напрямку, тобто зсув на –3 елементи вліво – це те ж саме, що на 3 елементи вправо.

SLL, SRL – це операції логічного зсуву вліво (***L***eft) і вправо (***R***ight) відповідно. Під час логічного зсуву крайні елементи заповнюються значеннями *0* чи *false*.

SLA, SRA – це операції арифметичного зсуву. Арифметичний зсув відрізняється від логічного тим, що крайні елементи під час зсуву заповнюються значеннями, яке перед початком операції мав крайній зліва елемент масиву, що містить інформацію про знак числа.

ROL, ROR – це операції циклічного зсуву (ротації) елементів масиву. Ротація означає зсув елементів по кільцю. Під час ротації вправо значення крайнього правого елемента переписується у крайній зліва елемент, а при ротації вліво – навпаки.

Операції порівняння **=** і ***/=*** можуть виконуватись над операндами різних типів. Результат порівняння завжди має тип *boolean* і дорівнює *true*, коли операнди однакові як за типом, так і за значенням. Операції порівняння **<**, **<=**, **>** та **>=** вимагають, щоб обидва операнди були однакового типу. Результат порівняння дорівнює *true*, коли операнди мають однакові значення.

Операція конкатенації ***&*** з'єднує два одновимірні масиви в один так, що його результатом є масив, що складається з елементів першого, за яким слідують елементи другого масиву. Операція конкатенації може також додати один новий елемент до масиву або два окремих елементи об'єднати в масив.

2.4.1 Оператор присвоєння значень змінним

Як і у багатьох інших мовах програмування, змінна набуває нового значення за допомогою конструкції присвоєння:

<ім'я змінної> := <вираз>;

Оскільки **VHDL** відноситься до мов зі строгим контролем типів, вираз повинен бути того ж типу, що і змінна.

2.4.2 Оператор присвоєння значень сигналам

Сигналу набуває нового значення за допомогою конструкції присвоєння:

<ім'я сигналу> ***<=*** *<вираз>****;***

Те, що знак присвоєння значення сигналу відрізняється від знаку присвоєння значення змінній, пояснюється не тільки тим, що йдеться про різні об'єкти. Принципова різниця між ними полягає у тому, що змінна набуває нового значення відразу, як тільки буде виконано оператор присвоєння. На відміну від цього, виконання оператору присвоєння значення сигналові не змінює значення сигналу, а лише готує його зміну. Сама ж зміна значень виконується одночасно для всіх сигналів, задіяних у процесі, в момент, коли даний процес буде зупинено. Більш детальна інформація про процеси та сигнали у них викладена у розділах 3.3- 3.5.

2.4.3 Особливості присвоєння значень агрегатам даних

Агрегат є спільною назвою масивів і записів, отже і змінна і сигнал може бути агрегатом. Наведемо приклади синтаксису операторів присвоєння значень агрегатам. Хай змінна *d1* є масивом з чотирьох елементів:

VARIABLE *d1 : BIT_VECTOR (1* ***TO*** *4)*

у якому першому і третьому елементам треба надати значення *'0'*, а другому і четвертому – *'1'*.

Це можна зробити, вказавши відповідні значення у списку, взятому в дужки:

d1:= ('0','1','0','1');

тут відповідність між елементами масиву і їх значеннями задана *позиційно*: першому елементу відповідає перше значення, другому – друге і т. д.

Відповідність можна встановлювати *за іменами*, безпосередньо вказуючи ім'я елементу у записі чи номер елементу в масиві, а потім, після знаку => – його значення, наприклад:

d1:= (2=>'0', 3 =>'1', 1=>'0', 4=>'1');

Послідовність, у якій тут вказуються значення, може бути довільною.

Можна одночасно застосовувати обидва способи, тоді у першій частині списку відповідність встановлюється позиційно, а в другій – за іменами, наприклад:

d1:= ('0', '1', 4=>'1', 3=>'0');

Можлива і третя частина такого списку – це службове слово ***OTHERS***, за яким вказується значення, що буде надано решті елементів, наприклад:

d1:= ('0', 3=>'0', ***OTHERS*** *=>'1');*

при цьому перші дві частини списку можуть бути відсутніми. Якби, наприклад, треба було б усім елементам масиву надати значення *'0'*, то найпростіше це можна зробити так:

*d1:= (****OTHERS*** *=>'0');*

Якщо значення декількох елементів агрегату співпадають, то це значення можна вказати лише один раз, а імена елементів перерахувати, відділяючи один від одного вертикальною рискою | , наприклад:

d1:= (1 | 3 =>'0', 2 | 4 =>'1');

Якщо співпадають значення декількох елементів підряд, то корисним є застосування службових слів ***TO*** чи ***DOWNTO.*** Наприклад, щоб сигналові *Data_Bus*, задекларованому як

SIGNAL *Data_Bus : Std_Logic_Vector (15* ***DOWNTO*** *0);*

надати значення ("1000000011111111"), можна скористатися оператором:

Data_Bus <= (14 ***DOWNTO*** *8 => '0',* ***OTHERS*** *=> '1');*

або

```
Data_Bus <= (15 | 7 DOWNTO 0 => '1', OTHERS => '0');
```

2.4.4 Затримки сигналів

У реальних схемах усі сигнали поширюються з затримками. Навіть якщо вихід і вхід зв‘язані між собою лише провідником, то і тут сигнал на виході з‘явиться з затримкою відносно вхідного сигналу на час, що дорівнює довжині провідника, поділеній на швидкість поширення сигналу в провіднику. Затримки такого типу в **VHDL** називаються *транспортними*. Приклад транспортної затримки показано на рис.2. Її особливість полягає у тому, що, якою б не була тривалість вхідних імпульсів, усі вони будуть відтворені на виході без жодних спотворень форми сигналу.

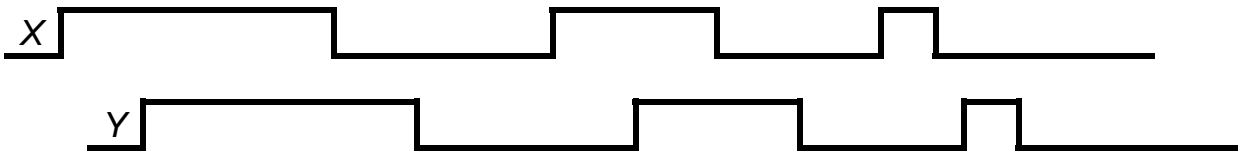

Рис. 2

Якщо сигнал *Y* формується як результат транспортної затримки сигналу *X* на час *T*, то мовою **VHDL** це записується так:

```
Y <= TRANSPORT  X  AFTER  T ;
```

Крім транспортних існують також *інерційні* затримки, обумовлені інерційністю напівпровідникових та інших електронних елементів і схем. Наприклад, якщо між входом і виходом включено підсилювач, то сигнал на виході буде повторювати вхідний сигнал з затримкою, що дорівнює часу, необхідному схемі підсилювача для переключення з одного стану в інший. Якщо, скажімо, підсилювач виконано на схемі з біполярним транзистором, то для нього це буде час виходу неосновних носіїв заряду з бази транзистора, перезаряду ємностей у схемі тощо.

Якщо, наприклад, підсилювач потребує 5 нс. для переключення з 0 в 1 чи з 1 в 0, то зрозуміло, що сформувати на виході імпульси, коротші за 5 нс. він не спроможний. Тому інерційна затримка може спотворювати форму вхідного сигналу, непропускаючи на вихід імпульси коротші ніж час цієї затримки (рис. 3).

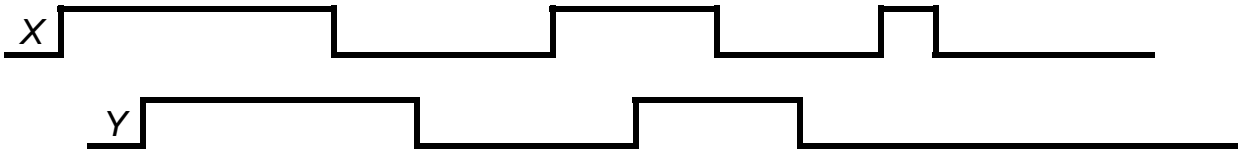

Рис. 3

Якщо сигнал *Y* формується як результат інерційної затримки сигналу *X* на час *T*, то мовою **VHDL** це записується:

```
Y <= INERTIAL  X  AFTER  T ;
```

або просто

```
Y <= X  AFTER  T ;
```

Якщо до інерційної затримки додається ще й транспортна, тоді значення часу затримки сигналу і мінімальної тривалості імпульсу, що нею пропускається, не співпадають. Для опису таких затримок використовується конструкція:

```
Y <= REJECT M INERTIAL  X  AFTER   T ;
```

де *M* – мінімальна тривалість імпульсу, яку спроможна пропустити дана затримка, *T* – час запізнення сигналу *Y* відносно сигналу *X*.

Затримки сигналів є дуже важливою характеристикою роботи схеми, вони точно відтворюються при моделюванні, проте **VHDL** не має таких засобів синтезу, що забезпечували б додержання вказаних значень затримок у реальних схемах.

2.4.5 Умовний оператор IF

Дозволяє виконувати чи не виконувати оператори в залежності від виконання умов. Умовами є вирази типу *boolean*.

```
IF <умова> THEN
   <послідовність операторів>
   { ELSEIF <умова> THEN <послідовність  операторів > }
   [ ELSE <послідовність  операторів >]
END IF;
```

2.4.6 Умовний оператор CASE

Дозволяє виконати послідовність операторів в одній із гілок алгоритму, що обирається в залежності від значення виразу.

```
CASE <вираз> IS
{ WHEN <вибір> { |<вибір> } | OTHERS => <послідовність операторів>}
END CASE;
```

У наведеному нижче прикладі змінній *Operation* типу *integer* надається значення:

- 1, якщо символьна змінна *Some_Characters* має значення *'a'* чи *'c'*;
- 2, якщо значення Some_Characters лежить в межах від *'d'* до *'g'*;
- 0 – у решті випадків.

```
CASE Some_Characters IS
   WHEN 'a'  | 'c' => Operation := 1;
   WHEN 'd'  TO  'g' => Operation := 2;
   WHEN OTHERS  => Operation := 0;
END CASE
```

2.4.7 Оператори циклу

VHDL має дві різновиди циклу:

У *циклі з передумовою*

```
<мітка циклу>:[ WHILE <умова виконання циклу>] LOOP
     <послідовність операторів>
END LOOP <мітка циклу>;
```

спочатку перевіряється *умова виконання циклу,* після чого *послідовність операторів* виконується нуль чи більше разів, доки ця умова має значення *true*. При відсу-

тності умови ***WHILE***, цикл буде нескінченим, якщо тільки його завершення не буде забезпечене оператором ***EXIT***.

У *циклі з параметром*

```
<мітка циклу>: FOR <параметр циклу> IN <діапазон> LOOP
    <послідовність операторів>
END LOOP <мітка циклу>;
```

послідовність операторів виконується для кожного значення *параметру циклу* з вказаного *діапазону*.

В обох різновидах циклу серед послідовності операторів, розташованих між ***LOOP*** та ***END LOOP*** можуть міститися оператори

```
NEXT [<мітка циклу>] [WHEN <умова >];
```

та

```
EXIT [<мітка циклу>][WHEN <умова >];
```

Перший з них викликає нове виконання *послідовності операторів* поточного чи вказаного міткою циклу, другий – вихід з нього. Для обох операторів може бути задана умова їх виконання.

Приклад оператора циклу:

```
Loop_1: FOR count  IN 1 TO 10 LOOP
    EXIT Loop_1  WHEN reset = '1';
    A_1: A(count) := '0';
END LOOP Loop_1;
```

2.4.8 Порожня конструкція

Оператор ***NULL*** не виконує жодних дій і використовується там, де треба підкреслити, що ніяка дія не виконується.

2.4.9 Оператор підтвердження

Оператор підтвердження

```
ASSERT <умова> [REPORT <повідомлення>] [SEVERITY <вираз>];
```

використовується для перевірки певної *умови* і для видачі *повідомлення* в разі, коли вона на виконується. Параметр *SEVERITY* дозволяє кожному оператору присвоїти певний рівень, що дозволяє зупиняти процес моделювання тільки при досягненню відповідного рівня помилки.

2.5 Підпрограми та пакети

Як і інші мови програмування, **VHDL** дає можливість використовувати підпрограми у формі процедур та функцій. **VHDL** також підтримує пакети для групування декларувань та об’єктів в окремі модулі. Пакети також забезпечують деяку абстракцію даних та приховування певних блоків інформації.

2.5.1 Підпрограми-процедури

Декларація підпрограми-процедури виконується так:

***PROCEDURE** <ім'я процедури> [(<список формальних параметрів>)] [**IS** <декаларації процедури>*
BEGIN
<послідовність операторів>
END PROCEDURE** <ім'я процедури>**];

В списку формальних параметрів можуть бути константи, змінні, сигнали та файли. Бувають процедури і без параметрів, а при використанні пакетів і опис самої процедури може бути перенесений в тіло пакету (див. розділ 2.5.4).

Для виклику процедури треба у програмі вказати її ім'я з відповідними списком фактичних параметрів.

Процедури в **VHDL** можуть бути вкладеними. Допускається і рекурсивний виклик процедур.

2.5.1.1 Приклад декларації функції без параметрів і тіла процедури:

***PROCEDURE** reset;*

Приклад виклику такої процедури:

reset;

2.5.1.2 Приклад процедури з параметрами:

***PROCEDURE** inc_reg (**VARIABLE** reg **: INOUT** word_32**; CONSTANT** incr **: IN** integer:=1);*

Ця процедура має два параметри: змінну *reg*, що має режим ***INOUT*** і тип *word_32*, та константу *incr*, що має режим ***IN,*** тип *integer* і значення *1*.

Така характеристика параметрів як режим вказує на напрямок передачі інформації між підпрограмою і програмою, яка її викликає. Значення параметрів з режимом ***IN*** можуть лише зчитуватись процедурою, параметрам з режимом ***OUT*** можна лише присвоювати значення, а параметрам з режимом ***INOUT*** доступне і те і інше. Тому, якщо слова ***CONSTANT*** чи ***VARIABLE*** відсутні в декларації параметрів, то параметри з режимом ***IN*** вважаються константами, а параметри з режимом ***INOUT*** чи ***OUT*** – змінними.

Значення, присвоєне формальному параметрові в декларації, залишиться в силі, якщо у виклику процедури відповідне йому значення фактичного параметру вказано не буде.

Виклик підпрограми включає список фактичних параметрів, що підставляються на місце формальних. Відповідність може встановлюватися

- за їх позицією у списку, наприклад *inc_reg (AX, 2);*
- за іменами, наприклад *inc_reg(incr=>1, reg=>AX);* або *inc_reg(reg=>AX);*
- комбінацією цих способів, наприклад *reg (AX,incr=>1);*

Зазначимо, що символ "**=>**" служить саме для позначення *відповідності* між формальними і фактичними параметрами і не має жодного відношення до *напрямку* передачі інформації.

2.5.2 Підпрограми-функції

Як і у інших мовах програмування, підпрограму-функцію **VHDL** можна розглядати як різновид підпрограми, де результат, що повертається у головну програму, асоціюється не з елементом у списку параметрів, а з самим іменем функції. Тому тут ми зупинимось лише на відмінностях підпрограми-функції від підпрограми-процедури.

Декларація функції виконується так:

```
FUNCTION <ім'я функції> [(<формальні параметри> ) RETURN <тип результату> ]  [ IS
   <декаларації процедури>
BEGIN
      <послідовність операторів>
END FUNCTION <ім'я функції >];
```

Тип значення, яке функція передає у головну програму як результату, описується в декларації після слова ***RETURN***. Крім того серед послідовності операторів має знаходитись хоча б один оператор ***RETURN***, за яким вказується значення результату, що повертається у програму, і виконанням якого завершується підпрограма, наприклад:

```
FUNCTION byte_to_int (byte:word) RETURN integer IS
   VARIABLE result: integer:=0;
   BEGIN
      FOR index IN 0 TO 7 LOOP
      result:= result*2+bit'pos(byte(index));
      END LOOP;
   RETURN result;
END byte_to_int;
```

2.5.3 Перевантаження підпрограм

VHDL дозволяє декільком підпрограмам, що мають різну кількість та типи формальних параметрів, мати однакові імена. Їх називають перевантаженими. Під час виклику такої програми береться до уваги не тільки її ім'я, а й кількість фактичних параметрів, їх порядок та типи, і навіть імена формальних параметрів (якщо відповідність між фактичними та формальними параметрами встановлюється через імена). Для функції враховується також і тип результату, який вона повертає. В результаті виклику виконується та підпрограма, яка відповідає параметрам виклику за усіма названими ознаками.

2.5.4 Пакети

Пакет – це набір типів, констант, сигналів, файлів, псевдонімів, підпрограм тощо, що можуть бути об'єднані користувачем за будь-якими логічними ознаками. Пакет складається з двох частин:

- інтерфейсної, що містить лише декларації складових пакету, та
- тіла пакета, з описами складових, задекларованих у інтерфейсній частині.

Тіло пакета може бути прихованими від користувачів пакета, які матимуть доступ лише до його інтерфейсної частини.

Інтерфейсна частина пакету описується так:

```
PACKAGE <ім'я пакету> IS
   {<декларації>};
END < ім'я пакету>;
```

Декларації, що тут розміщуються, можуть бути неповними. Наприклад, декларуючи константу, можна не вказувати її значення, (такі константанти називають затриманими) наприклад:

```
CONSTANT vector_ loc: address;
```

декларування підпрограм тут завершується перед словом ***IS*** і ніколи не містить тіла підпрограми, наприклад:

```
FUNCTION f1 (value: real) RETURN integer;
```

Тіло пакету записується так:

```
PACKAGE BODY <ім'я пакету> IS
   {<декларації тіла пакету>};
   {<декларації і тіла підпрограм>};
   {<декларації затриманих констант тощо>};
END < ім'я пакету>;
```

У наведеному нижче прикладі проілюстровано спосіб опису тіла підпрограми та надання значення затриманій константі.

```
PACKAGE BODY my_data  IS
    . . .
    FUNCTION f1(value: real) RETURN integer IS
    BEGIN
       . . .
    END f1;
    . . .
    CONSTANT vector_loc: address := 18;
END my_data;
```

2.5.5 Використання пакетів та видимість імен

Для того, щоб скористатися константами, типами, підпрограмами та іншими складовими пакету, можна перед їх іменами вказувати префікс з іменем пакету, що відділяється від решти імені крапкою, наприклад:

```
VARIABLE a1: my_data.address;
c:=5+ my_data.f1(x);
```

Щоб уникнути вживання префіксів можна скористатися, директивою ***USE***, наприклад**:**

```
USE my_data.adress, my_data.f1 ;
```

Тоді перераховані у ній імена стануть видимими для даної програми і надалі зможуть вживатися без префіксів. Якщо ж замість імені об'єкту вказати зарезервоване слово *all*:

```
USE my_data.all;
```

то це зробить доступними для даної програми усі об'єкти даного пакету.

Якщо пакет знаходиться в іншій бібліотеці, то перед префіксом пакета слід вказати ще й префікс бібліотеки, наприклад:

***USE** library_name.package_name.all;*

Але перед ним треба поставити ще й оператор ***LIBRARY***, що зробить видимою саму бібліотеку, наприклад:

***LIBRARY** library _name*

Існують дві наперед визначених бібліотеки, що неявно використовуються в кожному проекті: *STD* і *WORK*. Перша з них містить стандартні пакети *STANDARD* і *TEXTIO*, друга - це робоча бібліотека, де зберігаються усі розроблені і проаналізовані користувачем модулі проекту.

Визначені користувачем пакети зберігаються в робочій бібліотеці ***WORK.***

2.6 Блоки

Як і у інших мовах програмування, певні фрагменти коду програми на **VHDL** можуть бути об‘єднані у блок, що має такий синтаксис:

<мітка блоку> ***: BLOCK*** *[(умова захисту)]*
<декларації>
BEGIN
  <оператори>
END BLOCK *[<мітка блоку>]* ***;***

Причини, що спонукають користувача до застосування блоків полягають у тому, що, по-перше, програма, вміло розділена на блоки, краще структурована, а отже і краще сприймається читачем. По-друге, те, що задекларовані у блоці елементи програми (константи, змінні тощо) не видимі за його межами, дозволяє уникнути помилок у їх використанні.

Оператор може містити *умову захисту*, якою виступає вираз типу *Boolean*. В такому разі, значення цього виразу впливатимуть на виконання *захищених операторів* надання значень сигналам, що розміщені у тілі блоку і мають такий синтаксис:

<ім‘я сигналу> ***<= GUARDED*** *<вираз>****;***

Ці оператори виконуються лише тоді, коли значення *умови захисту* дорівнює *true*. Інакше вони ігноруються.

Замість використання *умови захисту*, з тією ж метою можна використати сигнал з зарезервованим іменем *GUARD*, що явно декларується і набуває значень у блоці. Перевага використання сигналу *GUARD* полягає у тому, що у такий спосіб рішення про виконання чи ігнорування захищених операторів може прийматись у результаті реалізації більш складного процесу, ніж просте обчислення булевого виразу, зазначеного в *умові захисту*.

3 Структурний опис мовою VHDL

Як уже відзначалось у розділі 1, опис будь-якої схеми, або її фрагменту складається з двох частин. Перша, що називається *сутністю (entity)*, містить опис зовнішнього інтерфейсу схеми (перелік входів, виходів тощо). Друга називається *архітектурою (architecture)* і містить опис, що визначає внутрішню будову та функціонування схеми.

Зазначимо, що одній сутності може відповідати багато архітектур, тоді як кожна архітектура належить лише одній сутності.

3.1 Декларування сутності

Звичайно складні схеми проектують за ієрархічним принципом. Схема складається з елементів, де кожний елемент, в свою чергу, може складатись з більш дрібних елементів і т.д.

Будь який елемент характеризується набором портів, що утворюють зовнішніх інтерфейс, через який даний елемент взаємодіє з оточуючим середовищем. Опис такого інтерфейсу мовою **VHDL** називається *сутністю (entity)* і має такий синтаксис:

```
ENTITY <ім'я сутності> IS
   [GENERIC (<список узагальнень>);]
   PORT (<список портів>);
END [[entity] <ім'я сутності>};
```

3.1.1 Список узагальнень

У *списку узагальнень (generic)* вказують перелік констант. Це дозволяє уникнути вживання конкретних числових значень в подальших описах, а отже дійсно надати їм більш загального вигляду. Наприклад розрядність шини *BusWidth* та частота синхронізації *clock_freq* можуть бути так описані константами у *списку узагальнень:*

```
GENERIC (BusWidth : Integer := 32,  clock_freq: frequency:=300 MHz);
```

що дозволить на етапі використання опису підставити інші числові значення, не вносячи виправлень у сам опис. Приклад використання списку узагальнень наведено у розділах 3.2.2 і 3.2.3.

3.1.2 Список портів

Список портів має такий синтаксис:

```
PORT ({<ім'я порту>:<режим порту>  <тип>});
```

Тут для всіх без винятку портів потрібно вказати їх імена, режими та типи.

В **VHDL** існує п'ять режимів портів:

- ***IN*** (вхід) – це режим, що дозволяє тільки зчитувати інформацію з порту. Записувати інформацію у порт, тобто надавати сигналові порта нових значень, заборонено;
- ***OUT*** (вихід) – це режим, що дозволяє тільки записувати інформацію у порт, зчитувати інформацію з порту заборонено;
- ***INOUT*** (двонаправлений вивід) – дозволяє і зчитувати інформацію з порту і записувати нову інформацію у порт. Кількість джерел, що формують сигнал на виході, може бути 0, 1, 2, ...;
- ***BUFFER*** (вихід-буфер) – це режим вихідного порту, де зчитування інформації дозволено. Але, на відміну від режиму *INOUT,* кількість джерел, що формують сигнал на виході тут може бути не більше одиниці.
- ***LINKAGE*** (редагування) – застосовуються тільки для відповідності інтерфейсу режиму редагування. Значення сигналу може записуватись і зчитуватись.

Приклади опису сутностей наведено в розділі 1.3.

3.2 Декларування архітектури

Архітектура – це опис, що визначає внутрішню будову елемента і його функціонування. В кінцевому результаті він повинен показувати як інформація на вихідних

портах елемента залежить від інформації на його вхідних портах. Досягти цієї мети можна двома способами:

- або безпосереднім описом поведінки елемента,
- або описом його структури, тобто схеми з‘єднань його компонентів – елементів нижчого рівня ієрархії, які в свою чергу можуть мати ієрархічну структуру, і т.д. При цьому компоненти самого нижчого ієрархічного рівня повинні бути описаними через поведінку.

Опис архітектури має такий синтаксис:

```
ARCHITECTURE <ім’я архітектури> OF <ім’я сутності> IS
    <декларації типів, сигналів, констант та ін.>
BEGIN
   <тіло архітектури>
END [ [architecture] <ім’я архітектури>]
```

3.2.1 Декларування сигналів

Сигнали в архітектурі є аналогом провідників в реальній схемі. Якщо елемент описаний у вигляді структури, то сигнали застосовуються для з’єднання компонентів, з яких складається елемент. Перед використанням усі сигнали мають бути задекларовані таким чином:

```
SIGNAL < ім‘я сигналу>:<тип> [:=<значення >];
```

Значення виразу в декларуванні сигналу використовується для надання сигналові початкового значення при моделюванні. Якщо вираз не вказано, то буде використовуватись значення за замовчуванням (див. розділ 2.3.2). Приклад декларації сигналів для схеми на рис. 1 наведено в розділі 1.3.4.

3.2.2 Декларування компонентів

Якщо сутності та архітектури компонентів, з яких складається елемент, відкомпільовані і записані до бібліотеки, то декларувати ці компоненти в архітектурі нема необхідності.

Якщо ж цього не зроблено, то є можливість дати опис компонентів у архітектурі елементу, де вони використовуються. Декларація компоненту дуже подібна до декларації сутності (див. розділ 3.1):

```
COMPONENT <ім‘я компоненту> [ IS ]
   [GENERIC (<список узагальнень>);]
   PORT (<список портів>);
 END COMPONENT [<ім‘я компоненту>];
```

Нижче наведено приклад декларування двох компонентів: двохвходового диз‘юнктора *Logic_OR* та елементу постійної пам‘яті *ROM:*

```
COMPONENT Logic_OR  IS
   GENERIC (delai: Time:=1ns);
   PORT (x1,x2: IN bitl;
   y: OUT bitl);
END COMPONENT Logic_OR;
```

```
COMPONENT ROM
   GENERIC (data_bits, addr_bits: positive);
   PORT (en: IN bit; addr: IN bit_vector(addr_bits-1 DOWNTO 0);
   data: OUT bit_vector (data_bits -1 DOWNTO 0));
END COMPONENT ROM;
```

Зазначимо, що часова затримка у диз‘юнкторі та розрядність шини адреси і даних визначається тут константами у списку узагальнень.

3.2.3 Використання компонентів

Щоб використати задекларовані компоненти у архітектурі користуються такою конструкцією:

```
<мітка> : <ім’я_компоненту>
[GENERIC MAP <список констант>;]
[PORT MAP <список сигналів>;]
```

Наприклад, фрагмент архітектури, що використовує описані вище компоненти, може виглядати так:

```
SIGNAL en1, en2, rom_sel : bit, A: bit_vector(31 DOWNTO 0),
 D: bit_vector(15 DOWNTO 0);
enable_data: Logic_OR
PORT MAP (en1, en2, rom_sel);
parametr_rom: ROM
GENERIC MAP (data_bits =>16, addr_bits =>32);
PORT MAP(en =>rom_sel, data =>D(15 DOWNTO 0), addr =>A(31 DOWNTO 0));
```

З цього опису випливає, що внутрішні сигнали *en1, en2* типу *bit* підключені до входів диз‘юнктора *Logic_OR*, вихід якого *rom_sel* з‘єднаний з входом *en* елементу пам‘яті *ROM*.

При застосуванні диз‘юнктора *Logic_OR* оператор ***GENERIC MAP*** опущено. Отже у цьому елементі затримка сигналу *delai* залишається такою, якою вона була вказана у декларації компонента, тобто 1 нс. Оператор ***GENERIC MAP*** задає розрядність шини адреси *addr_bits* і даних.

3.3 Процеси та оператор WAIT

Як відомо, робота комп‘ютера, що виконує команди програми, носить послідовний дискретний характер, тоді як реальні процеси у реальних схемах протікають неперервно у часі і паралельно у просторі. Для того, щоб, незважаючи на цю обставину, комп‘ютер міг адекватно відображати паралельні процеси, мова **VHDL** має спеціальну конструкцію з таким синтаксисом:

```
<мітка>: PROCESS [(<список чутливості>)]
<декларації>;
BEGIN
   <тіло>;
END PROCESS [<мітка>];
```

Основну частину *тіла* процесу складають оператори присвоєння значень сигналам **<=**, якими задається залежність вихідних сигналів від вхідних.

Як уже відзначалось, реальні процеси у реальних схемах протікають неперервно у часі, а от їх моделювання може зупинятись і відновлюватись, тобто виконуватись дискретно у часі.

Зупиняється процес на період, під час якого зміна значень вихідних сигналів неможлива, а відновлюється він тоді, коли виникають умови, що можуть спричинити таку зміну. Зокрема процес призупиняться після виконання останнього оператора у *тілі* процесу, а відновлюється тоді, коли змінюється значення хоча б одного сигналу, що входить до *списку чутливості.*

На відміну від оператора **:=**, що надає нового значення змінній, виконання оператора **<=** не змінює зразу значення сигналу, а тільки підготовлює таку зміну. Робиться це так. Вказане у операторі **<=** значення фіксується у драйвері сигналу (кожний сигнал процесу має свій драйвер), а у момент зупинки процесу зафіксовані у драйверах значення надаються усім сигналам одночасно.

Нижче наведено приклад процесу, що описує роботу D-тригера зі статичним синхровходом.

```
PROCESS (C,D)
BEGIN
    IF  C = '1'  THEN
        Q <= D ;
        NQ <= not D ;
    END IF;
END PROCESS;
```

Список чутливості вказує на те, що зміна значень вихідних сигналів може бути спричиненою зміною значень сигналів *C* і *D*. Характер цієї зміни описаний у тілі циклу: якщо *C = '1'*, то нове значення сигналу *Q* співпадатиме зі значенням сигналу *D*, а *NQ* – з його інверсією.

Зауважимо, що, виключивши сигнал *D* зі списку чутливості, і замінивши перший рядок на

```
PROCESS (C)   ,
```

ми отримаємо опис D-тригера з динамічним синхровходом, що змінюватиме своє значення тільки при зміні сигналу *C* з *‘0’* в *‘1’*.

Замість списку чутливості може застосовуватись оператор *WAIT,* що зупиняє процес на період, поки не буде виконана вказана у ньому умова, після чого процес відновлюється. Існують такі різновиди цього оператора:

```
WAIT;
WAIT ON <список сигналів>;
WAIT UNTIL <умова>;
WAIT FOR <час>;
```

Перший визначає зупинку процесу назавжди; другий визначає зупинку процесу, що буде відновлений у випадку будь-якої зміни значень сигналів, зазначених у списку; третій – доки вказаний у ньому вираз не набуде значення *true* і останній зупиняє процес на зазначений у операторі час. Можлива також комбінація декількох умов у одному операторі, наприклад

```
WAIT ON A, B UNTIL CLK = '1';
```

забезпечує зупинку процесу, який відновиться тільки після зміни значень сигналів *A, B,* якщо сигнал *CLK* матиме значення *'1'*.

Використовувати в одному процесі і список чутливості і оператор ***WAIT*** не дозволяється. Якщо процес має список чутливості, то в його тілі оператор ***WAIT*** чи підпрограми, що містять цей оператор використовувати не можна.

3.4 Альтернативні присвоєння значень сигналам

Оператори присвоєння значень сигналам використовуються як для опису процесу, так і для опису архітектури. Якщо опис процесу містить умовні оператори ***IF*** і ***CASE***, то у описі архітектури їм відповідають оператори умовного і вибіркового присвоєння значень.

Синтаксис оператора **умовного присвоєння** нагадує синтаксис оператора ***IF***:

```
<ім'я сигналу> <= [<механізм затримки>] <значення1> WHEN <умова1>
        ELSE [<механізм затримки>] <значення2> WHEN <умова2>
        . . .
        ELSE [<механізм затримки>] <значення>;
```

Згідно з цим оператором сигналові буде надано *значення1*, якщо виконується *умова1*, інакше перевіряється наступна *умова2* і т.д. Якщо жодна із умов не виконується, то сигналові присвоюється *значення*, яким завершується оператор, наприклад:

```
BufOut <= BufIn AFTER Tau WHEN = '1'  ELSE 'Z' AFTER Tau;
```

надає сигналу *BufOut* значення, що співпадає зі значенням *BufIn*, якщо *Enable = '1'*, інакше *BufOut* дорівнюватиме *'Z'*. Значення сигналу *BufOut* встановлюються після відповідної зміни сигналу *Enable* з затримкою на час *Tau*.

Синтаксис оператора **вибіркового присвоєння** нагадує синтаксис оператора ***CASE:***

```
WITH <вираз> SELECT
<ім'я сигналу> <= [<механізм затримки>]  <значення1> WHEN <вибір1>,
        [<механізм затримки>]  <значення2> WHEN < вибір 2>, ...
        [<механізм затримки>]  <значення> WHEN OTHERS ;
```

Тут сигналові буде надано *значення1*, якщо *вираз* відповідає *вибору1*, *значення2 –* якщо *вираз* відповідає *вибору2,* і т.д. Якщо ж *вираз* не відповідає жодному із вказаних виборів, то сигналові буде надано *значення*, вказаного перед словами ***WHEN OTHERS***.

Так само, як і у операторі ***CASE,*** вибір може містити перелік значень, що відділяються одне від одного вертикальною рискою |, та діапазон значень, межі якого розділяються службовим словом ***TO*** наприклад:

```
WITH instr_cod SELECT
instr_result <= op1+op2 WHEN 1 | 2,
      op1- op2 WHEN 3 TO 5,
      op1 AND op2 WHEN 6;
      0   WHEN OTHERS ;
```

Тут сигналові *instr_result* надається значення:

- *op1+op2*, якщо *instr_cod* дорівнює *1* чи *2*;
- *op1-op2,* якщо *instr_cod* лежить у межах від *3* до *5*;
- *op1* ***AND*** *op2,* якщо *instr_cod* дорівнює *6*, і
- *0* – у решті випадків.

3.5 Вирішення конфліктів між сигналами

На практиці зустрічаються схеми, у яких виходи декількох елементів з‘єднані між собою у один спільний вихід. Це можуть бути елементи з відкритим колектором, що реалізують функції монтажної логіки, елементи з трьома станами виходу[2] тощо.

Якби ці виходи не були з‘єднані, то значення сигналу на виході кожного з них визначалось би власним драйвером цього вихідного сигналу. Як же формувати значення сигналу на спільному виході після об‘єднання декількох виходів елементів в один? Адже різні драйвери можуть формувати різні значення сигналу.

Нагадаємо, що у системі **PCAD** ці проблеми розв‘язувались через поняття “логічної сили” сигналу, і у таких випадках один з сигналів “пересилював” інші.

У **VHDL** такі конфлікти вирішуються через застосування механізму розв’язувальної функції, (resolution function), яка бере всі значення, що виробляються драйверами, та формує якесь результуюче значення для цього сигналу.

Застосування розв‘язувальної функції потребує і використання відповідних типів даних для значень сигналів, бо тільки бітових значень *‘0‘* і *‘1‘* уже недостатньо для адекватного опису сигналів, у реальних схемах і вирішення конфліктів між ними. Типи даних, для яких визначена розв‘язувальна функція, називають розв‘язаними.

Нижче, як приклад, показано як вирішується ця проблема у пакеті *Std_Logic_1164,* що став загальновизнаним промисловим стандартом. Він містить визначення типів, підтипів, і функцій, що розширюють **VHDL** на багатозначну логіку.

```
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;
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' )   -- | - |
       );
```

[2] Крім станів логічної одиниці і логічного нуля, вихід таких елементів може перебувати у стані високого імпедансу, що еквівалентно відключенню елемента від даного виходу.

```
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
VARIABLE result : std_ulogic := 'Z';
BEGIN
 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;
```

Тут спочатку описано перелічуваний тип *std_ulogic* та тип *std_ulogic_vector.* Останній являє собою тип масиву, що складається з наперед невизначеної кількості елементів типу *std_ulogic.* Позначення елементів означають такі стани виходів:

- *'U'* – непроініціалізований (Uninitialized),
- *'X'* – посилене невідоме (Forcing Unknown),
- *'0'* – посилений нуль (Forcing 0),
- *'1'* – посилена одиниця (Forcing 1),
- *'Z'* – високий імпеданс (High Impedance),
- *'W'* – слабке невідоме (Weak Unknown),
- *'L'* – слабкий нуль (Weak 0),
- *'H'* – слабка одиниця (Weak 1),
- *'-'* – байдуже (Don't Care).

Далі наводиться декларація розв‘язувальної функції *resolved*, що визначена для аргументу типу *std_ulogic_vector* і повертає результат типу *std_ulogic*. Підтип *std_logic* визначено як такий, що утворюється функцією *resolved* з типу *std_ulogic*, а тип *std_logic_vector* - як тип масиву, елементи якого мають тип *std_logic*.

Константа *resolution_table* являє собою таблицю, що визначає правила формування значення результуючого сигналу в залежності від значень двох драйверів (їм відповідають рядки і колонки таблиці). Розв‘язувальа функція *resolved* написана так, що спочатку проміжному результатові надається значення *'Z'*. Потім виконується перевірка кількості драйверів, що описують даний сигнал. Якщо драйвер єдиний, то його значення і повертає функція. У випадку декількох драйверів значення функції формується шляхом багаторазового зчитування із таблиці, колонка і рядок якої відповідають значенням проміжному результату і наступного драйвера.

Пакет *Std_Logic_1164* містить також підтипи *X01, X01Z, UX01, UX01Z* типу *std_ulogic*. Назви цих підтипів складені зі значень сигналів, що в них присутні. Наприклад підтип *X01* містить тільки значення *'X','0','1'*.

3.6 Оператор generate

Оператор *generate* є засобом для спрощення опису архітектури регулярних структур (тобто однакових елементів однаковим чином з‘єднаних між собою). Існує два різновиди синтаксису цього оператора:

```
<мітка>: FOR <параметр> IN <діапазон> GENERATE
    [ {<декларації>}
    BEGIN ]
    <оператори>
    END GENERATE [<мітка>] ;

<мітка>: IF <умова> GENERATE
    [ {<декларації>}
    BEGIN ]
    <оператори>
    END GENERATE [<мітка>] ;
```

Перший є еквівалентом повторного вживання розташованих в його тілі ***операторів*** для кожного значення ***параметру*** з вказаного ***діапазону***, другий є еквівалентом повторного вживання ***операторів*** доти, доки виконується вказана ***умова***.

Наприклад архітектура показаного на рис. 4 N-розрядного двійкового лічильника, побудованого на D-тригерах може бути описана так:

```
ARCHITECTURE beh OF counter_bin_n IS
COMPONENT D_FF
  PORT (D, CLK_S : IN BIT; Q, NQ : OUT BIT);
END COMPONENT D_FF;
SIGNAL S : BIT_VECTOR(0 TO N);
BEGIN
 S(0) <= IN_1;
 G_1 : FOR i IN 0 TO N-1 GENERATE
       D_Flip_Flop :  D_FF PORT MAP  (S(i+1), S(i), Q(i), S(i+1));
 END GENERATE;
END ARCHITECTURE beh;
```

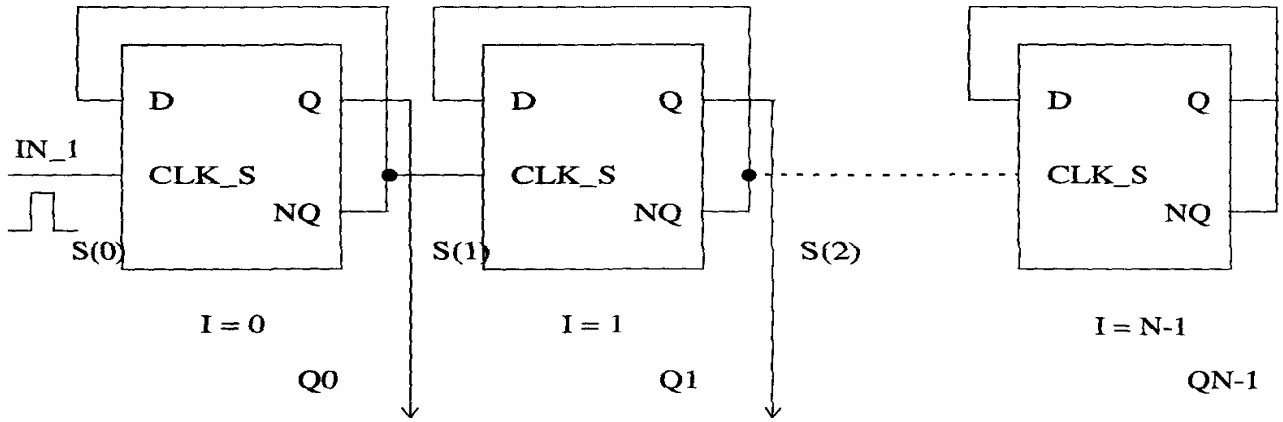

Рис. 4

Другий приклад ілюструє опис паралельного комбінаційного суматора з послідовним переносом, схема якого показана на рис. 5. Особливістю цього суматора є те, що у його молодшому розряді застосовується напівсуматор - спрощена схема без входу переносу.

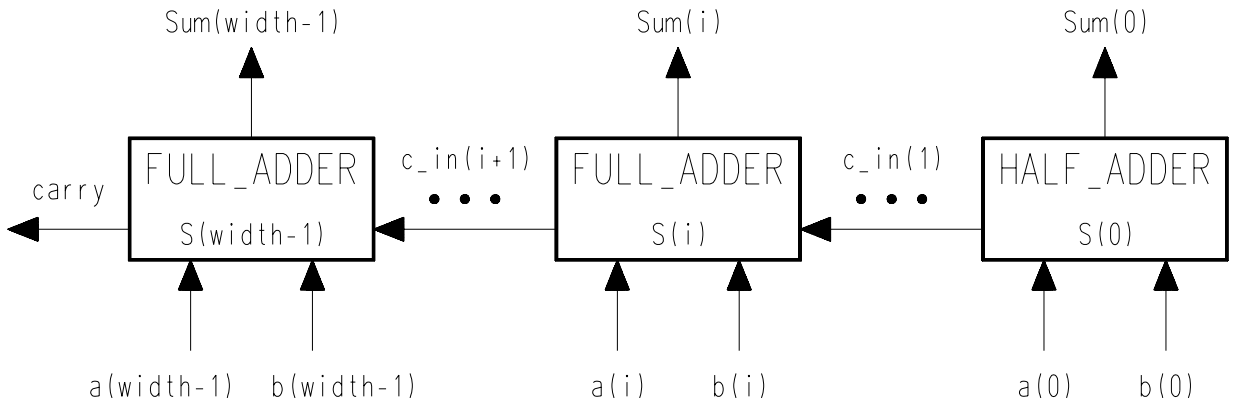

Рис. 5

```
ENTITY Half_adder IS
  PORT (a,b :  IN Bit; Sum,c : OUT Bit );
END ENTITY Half_adder ;

ARCHITECTURE Half_adder OF  Half_adder IS
BEGIN
  Sum <= a XOR b;
  c <= a AND b;
END ARCHITECTURE Half_adder ;

ENTITY Full_adder IS
  PORT (a,b,c0 :  IN Bit ;  Sum,c : OUT Bit );
END ENTITY Full_adder ;

ARCHITECTURE Full_adder OF  Full_adder IS
BEGIN
  Sum <= (a XOR b) XOR c0 ;
  c <= (a AND b) OR (a AND c0) OR (b AND c0);
END ARCHITECTURE Full_adder ;

ENTITY parallel_adder_N  IS
GENERIC (width : Integer := 32);
PORT (Sum : OUT Bit_Vector (0 TO  width -1);
     a,b :  IN Bit_Vector (0 TO  width -1); carry : OUT Bit );
END ENTITY parallel_adder_N;

ARCHITECTURE parallel_adder OF  parallel_adder_N  IS
 SIGNAL   c_in : Bit_Vector (1 TO  width );
 adder: FOR i IN 0 TO width-1 GENERATE
   ls_bit: IF i=0 GENERATE
   ls_cell: half_adder PORT MAP (a(0), b(0), sum(0), c_in(1));
   END GENERATE ls_bit;
```

```
    middle_bit: IF i>0 AND i<width-1 GENERATE
    middle_cell: full_adder PORT MAP (a(i), b(i), c_in(i), sum(i), c_in(i+1));
    END GENERATE middle_bit;

    ms_bit: IF i=width-1 GENERATE
    ms_cell: full_adder PORT MAP (a(i), b(i), c_in(i), sum(i), carry);
    END GENERATE ms_bit;
  END GENERATE adder;
END ARCHITECTURE parallel_adder ;
```

Література

1. Мова опису апаратних засобів комп'ютера – VHDL. Методичні вказівки до курсової роботи з дисципліни "Теорія і проектування комп'ютерів, комплексів, систем та мереж" для студентів спеціальності 7.091501 "Комп'ютерні та інтелектуальні системи та мережі" /Укл. А.О. Мельник, О.М.Почаєвець. Львів, ДУ "ЛП", 1997.

Зміст