МІНОСВІТИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Інститут Комп'ютерної Техніки та Автоматики

Кафедра Електронних Обчислювальних Машин

Курс лекцій

з дисципліни

"Мови опису апаратних засобів"

автор: Хомич С.В.

2005

Зміст

1. Вступ. Класифікація, призначення та сфери застосування мов опису апаратних засобів.

Проектування на основі логічних рівнянь

Без розуміння таких базових конструктивних блоків як логічні вентилі і тригери було б важко проектувати будь-яку цифрову систему. Більшість логічних схем на основі вентилів та тригерів традиційно розробляються за допомогою логічних рівнянь. Було розроблено багато методів для оптимізації цієї методики, в тому числі мінімізація рівнянь для більш ефективного використання логічних вентилів і тригерів.

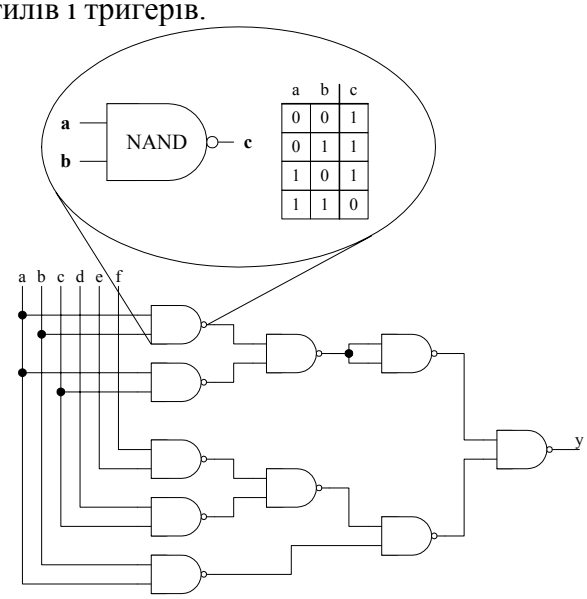


Рис.1.1.

Така техніка проектування вимагає створення логічних рівнянь для кожного тригеру та блоку вентилів. Це робить логічні рівняння непридатними для великих проектів, що містять сотні тригерів, оскільки логічні вирази стають надто складними.

Теоретично будь-яка система може бути представлена за допомогою логічних рівнянь. Однак практично неможливо оперувати тисячами логічних рівнянь, що мали би використовуватись в сучасних проектах.

Схемотехнічне проектування

Методи схемотехнічного проектування розширюють можливості проектування на основі логічних рівнянь завдяки використанню, крім вентилів та тригерів, інших додаткових схем. Оскільки ці схеми можуть складатись з вентилів та тригерів, а також з інших схем, їх використання дозволяє ввести ієрархічне проектування, що в свою чергу значно спрощує створення великих проектів у порівнянні із застосуванням логічних рівнянь.

Більшість людей надає перевагу графічному представленню проекту, оскільки при цьому зв'язки між блоками проекту виглядають наочніше. Оскільки схеми забезпечують графічне представлення проекту, таке проектування є досить популярним.

На протязі багатьох років схемотехнічне проектування вважалося найбільш оптимальним вибором для представлення проекту. Але через складність сучасних пристроїв створення схем стало дуже тривалим, а їх використання – дуже обмеженим в їх традиційній формі представлення проекту.

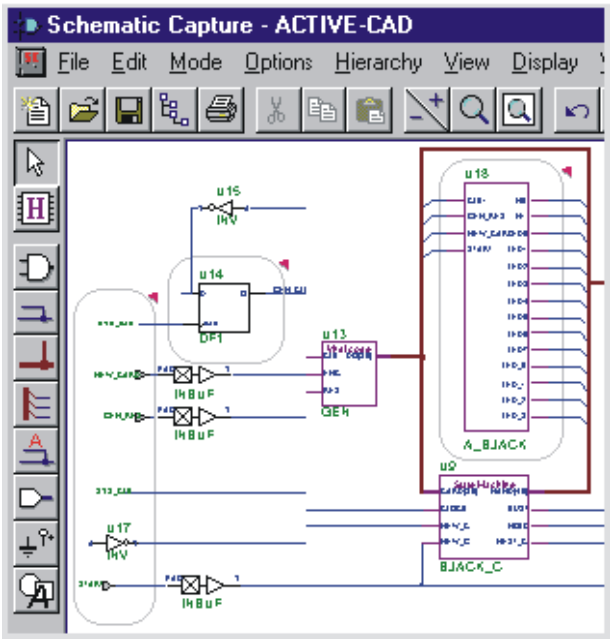


Рис.1.2.

Недоліки традиційних методів

Традиційна послідовність етапів створення проекту виглядає таким чином:

Рис.1.3.

Незважаючи на простоту використання, традиційні логічні рівняння та схемотехнічне проектування мають деякі недоліки. Найважливіший з них – система завжди описується як схема зв'язаних між собою елементів. При цьому відсутня можливість описати, як створена система і як вона працює.

З іншого боку, специфікація системи завжди подається в формі очікуваної поведінки системи (тобто "що система буде робити в тому чи іншому випадку").

Інший недолік традиційних методів проектування полягає в складності оперування великими проектами. Обробка сотень логічних рівнянь є складною, але можлива. Однак тисячі логічних рівнянь важко навіть уявити. А найновіші інтегральні схеми містять мільони вентилів і їх складність постійно зростає.

Використання мов опису апаратних засобів

Головний недолік традиційних методів проектування – це необхідність вручну транслювати опис проекту в набір логічних рівнянь. Цей крок може бути повністю виключений за допомогою мов опису апаратних засобів (HDL – *hardware description language*). Наприклад, більшість засобів HDL дозволяють використовувати кінцеві автомати (*finite state machines*) для послідовнісних систем і таблиці істиності для комбінаційних модулів. Такий опис проекту може бути автоматично конвертований в HDL-код, який реалізується засобами синтезу.

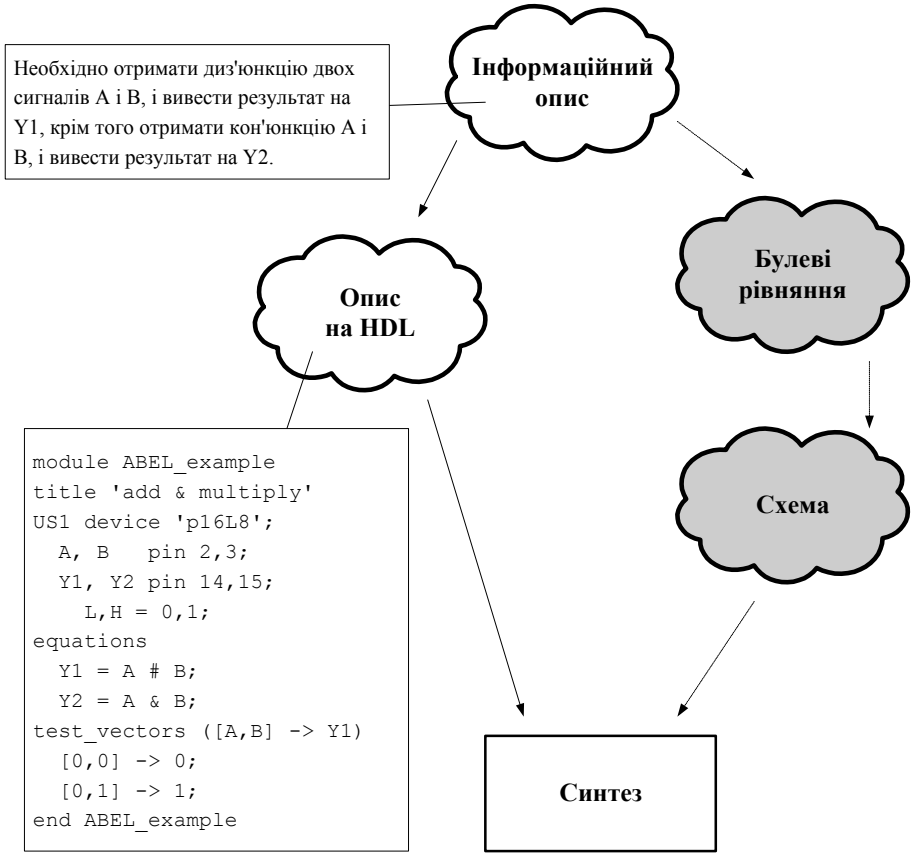


Рис.1.4.

Мови опису апаратних засобів знайшли приципове застосування для програмованих логічних матриць (PLD) різної складності, від простих PLD до CPLD (complex PLD) та FPGA (Field Programmable Gate Array). На сьогодні використовуються декілька мов апаратних засобів. Найбільш популярними є VHDL, Verilog та Abel.

Оперування різними рівнями опису

Оскільки схеми стають дедалі складнішими, мають місце дві паралельні тенденції:

- перехід до більш загальних форм вхідних даних, таких як опис поведінки системи,
- використання комп'ютерних систем автоматизації проектування.

Існує декілька рівнів опису системи: від рівня кремнієвого кристалу до складної специфікації системи. Ці рівні можна аналізувати як в термінах структури системи, так і в термінах її поведінки. Сьогоднішні технології настільки складні, що традиційні методи проектування не охоплюють усього спектру рівнів проектування. Засоби автоматизації проектування, які з'явились в останні роки, орієнтовані на найнижчий рівень (кремній) і на інтегрування процесу проектування.

Рис.1.5.

Класифікація мов опису апаратних засобів

Основними мовами опису апаратних засобів на сьогодні є VHDL і Verilog.

Мова **Verilog** досить проста для розуміння і використання, особливо для тих, хто знайомий з мовою С. Широко використовується для проектування НВІС, але тільки на рівні регістрів і нижче. Процес симуляції відносно швидкий. Головний недолік – відсутні конструкції для специфікацій системного рівня.

Мова **VHDL** більш складна, отже важча для розуміння і використання. Однак забезпечує набагато більшу гнучкість, що є одночасно її перевагою і недоліком – це пов'язано із її надлишковістю і можливостями стилів кодування. Але VHDL набагато більше підходить для створення великих проектів вищого і системного рівнів. Симуляція відносно повільна, але засоби симулювання постійно вдосконалюються.

Крім того існують ще і інші мови проектування, такі як Abel і Altera HDL.

Мова **Abel** підтримує досить багато поведінкових вхідних форм, в тому числі високорівневі вирази, діаграми станів та таблиці істиності. Дозволяє розробляти та проводити верифікацію, не піклуючись про архітектуру розроблюваного пристрою. Архітектурно-незалежні специфікації проектів (які не містять декларацій пристроїв та номерів виводів) вимають більш повних описів, ніж їх архітектурно-орієнтовані аналоги. Недоліком є складність опису великих проектів. На сьогодні ця мова використовується досить рідко.

Мова **Altera HDL** – це високо-рівнева модульна мова, що повністю інтегрована в систему MAX+PLUS II фірми Altera Corporation. Вона досить зручна для проектування складних комбінаційних схем, груп операцій, машин станів, таблиць істиності та параметризованої логіки. Оператори та елементи Altera HDL відносно потужні, універсальні та прості у використанні. Недоліком цієї мови є те, що вона не підтримується іншими фірмами-розробниками засобів автоматизованого проектування НВІС, такими як Aldec Inc. та Xilinx.

I. Мова VHDL

2. Представлення системи у VHDL: інтерфейс та архітектура, використання пакетів.

Що таке VHDL

VHDL – це скорочення від VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. Ця досить складна назва була введена Міністерством оборони США, яке було першим інститутом, де зрозуміли переваги документування, моделювання та симуляції електронних пристроїв з використанням мови проектування. При їх субсидуванні VHDL був розроблений та впроваджений в проектувальне прикладне програмне забезпечення.

VHDL-симулятори з'явились на початку 90-х. Після того, як з 1994 року вони стали доступні для PC, розробники PLD та FPGA почали використовувати їх для великих проектів.

Необхідно зазначити, що слово "синтез" не згадувалось як одна з причин створення мови VHDL. VHDL в першу чергу створювався для симуляції, моделювання та документування проектів. Синтез був добавлений деякими розробниками, які побачили в цьому шлях до автоматизації процесу проектування.

Роботи по стандартизації

Незважаючи на те, що VHDL був розроблений та впроваджений Міністерством оборони США, він дуже скоро почав використовуватись для невійськових проектів. Це стало можливим завдяки потужним зусиллям багатьох розробників, які зробили його широко доступним, розповсюджуючи стандарти через IEEE (the Institute of Electrical and Electronic Engineers), найбільш впливову професійну організацію в цій галузі. Перший стандарт VHDL з'явився у 1987 році як Std 1076, а пізніше у 1993 році за правилами IEEE він був виправлений та доповнений. Цей стандарт був орієнтований на проектування цифрових систем і підтримується в даний час всіма основними промисловими системами моделювання та синтезу. Остання версія стандарту була видана у 1999 році (усі стандарти IEEE переглядаються кожні 5 років) і місить розширення, що дозволяють описувати моделі аналогових та змішаних (цифро-аналогових) схем.

Відповідаючи потребам розробників, різноманітні групи експертів об'єднують зусилля для того, щоби зробити мову ще більш універсальною. Результат їх роботи знайшов відображення в декількох стандартах, що підтримуються основним 1076 IEEE VHDL стандартом.

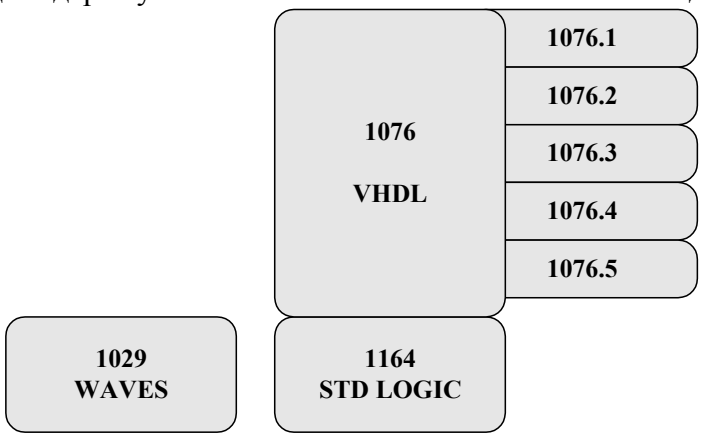


Рис.2.1.

IEEE Std 1076 – це основний стандарт VHDL, що містить опис мови. Існує дві його версії: 1076-87 та 1076-93, відповідно до років їх публікації. Незважаючи на поширеність версії 1076-93, можливості деяких засобів проектування обмежені версією 1076-87. Далі буде розглядаєтись версія 1076-93.

1076.1 – це аналогове розширення VHDL, запропоноване для опису аналогових та змішаних аналого-цифрових систем. Нереалізований.

1076.2 – математичні утіліти VHDL – забезпечує підтримку математичних операцій. Нереалізований.

1076.3 – синтез VHDL – це зусилля з встановлення синтезованої підмножини VHDL. До сих пір продавці засобів синтезу використовують власні підмножини VHDL, створюючи описи проектів залежними від продавця.

1076.4 – бібліотеки спеціалізованих BIC VITAL (VHDL Initiative Towards ASIC Libraries) – призначено для стандартизації моделей HBIC (ASIC).

1076.5 – бібліотека IEEE – присвячено розширенню та об'єднанню бібліотечних компонентів IEEE. Нереалізований.

1164 – розширює мову VHDL багатозначною логікою, необхідною для опису систем реального часу. Це де факто індустріальний стандарт і усі засоби VHDL його підтримують.

1029 – специфікація обміну даними між часовими діаграмами та векторами VHDL – визначає формати даних для тестування і часової верифікації.

Як система співпрацює із середовищем

Яку б функцію не виконувала система, вона має отримувати деякі вхідні дані і виводити деякі вихідні результати. Іншими словами, система має спілкуватись із середовищем.

Комунікаційна частина системи називається інтерфейсом. Системний інтерфейс описується у VHDL через *блок інтерфейсу* (***entity***) або просто *інтерфейс*, який є базовим елементом проектування будь-якої системи. Як неможливо створити систему без інтерфейсу, так і неможливо створити VHDL-систему без блоку інтерфейса.

Для досягнення певної функціональності дані повинні якимось чином перетворюватись всередині системи. Ця трансформація даних і вивід очікуваних функцій виконується внутрішньою частиною системи, або тілом системи, яке називається *архітектурою* (***architecture***).

Функціонування системи може бути як дуже простим (включення/виключення деякого перемикача), так і дуже складним (автопілот пасажирського літака). Однак в будь-якому випадку система може бути представлена набором *тіла* та *інтерфейса*.

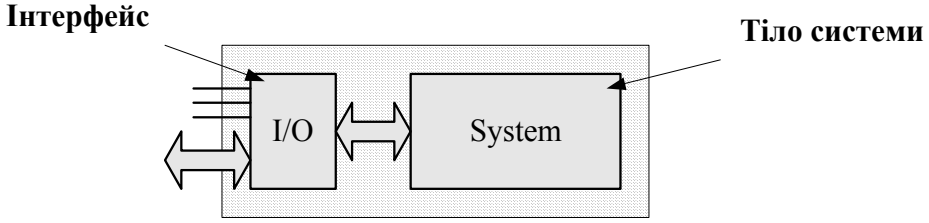


Рис..2.2.

Специфікація інтерфейсу

Оскільки починати будь-який проект необхідно з аналізу його середовища, не дивно, що *блок інтерфейсу* (***entity***), який описує інтерфейс між системою та її середовищем, є головною частиною кожного опису.

Не може існувати VHDL-специфікації якоїсь системи без декларації інтерфейсу. Крім того, все, що описано в інтерфейсі, автоматично стає видимим для усіх інших елементів проекту, зв'язаних з цим інтерфейсом. Більше того, ім'я системи завжди співпадає з іменем її інтерфейсу.

Отже, проектування довільної системи у VHDL завжди необхідно починати з декларації інтерфейсу.

Склад інтерфейсу

Блок інтерфейсу забезпечує специфікацію інтерфейсу системи і звичайно складається з трьох елементів:

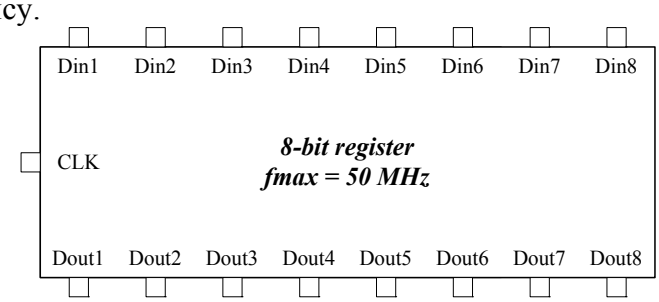
- *заголовку* інтерфейсу,
- *параметрів* системи (наприклад, ширина шини даних процесора або максимальна тактова частота),

- *з'єднання*, за допомогою яких інформація передається в систему та з неї (входи та виходи системи).

Елементи інтерфейсу

Два ключових елементи довільного інтерфейсу (*параметри* та *з'єднання*) у кожному інтерфейсі декларуються окремо:

- усі *параметри* декларуються як ***generics*** і передаються в тіло системи,
- з'єднання, через які передаються дані, називаються *портами* (***ports***) – вони формують другу частину інтерфейсу.




```
entity Eight_bit_register is

    LENGTH = 8
    fmax = 50 MHz
                              параметри

    D_IN  eight-bit input
    D_OUT eight-bit output
    CLK   one-bit input
                              з'єднання

end entity Eight_bit_register;
```

```
entity Eight_bit_register is

    generic ( LENGTH = 8
              fmax   = 50 MHz
            );
                              параметри

    port ( D_IN  eight-bit input
           D_OUT eight-bit output
           CLK   one-bit input
         };
                              з'єднання

end entity Eight_bit_register;
```

Рис.2.3.

Архітектура як опис тіла системи

Архітектура пристрою описується у VHDL як *архітектура інтерфейсу*:

```
entity Tvset is
...
end entity Tvset;

architecture TV2000 of Tvset is
...
end architecture TV2000;
```

Рис.2.4.

Типи архітектурних описів

Кожна система може бути описана як в термінах її функціональних можливостей (поведінки), так і в термінах її структури, відповідно тому, яка інформація про систему потрібна.

Звичайно засоби синтезу працюють з обидвома типами архітектурного опису: перший, який задає очікуване функціонування, має бути специфікований і формалізований, після чого він трансформується в структурний еквівалент, що більш придатний для засобів синтезу. Частина цієї

трансляції може бути проведена автоматично, однак повний функціональний (поведінковий) синтез на сьогодні ще не реалізований.

Поведінковий опис

Функціональний опис визначає те, що система, як очікується, буде робити. Наприклад, необхідно описати, як деяка система відреагує на вхідні сигнали від кнопок пульта дистанційного керування.

Поведінкова специфікація взагалі – це опис виходів як реакцій на вхідні дані.

Необхідно зазначити, що поведінковий опис не відповідає, як ці функціональні можливості системи повинні бути реалізовані.

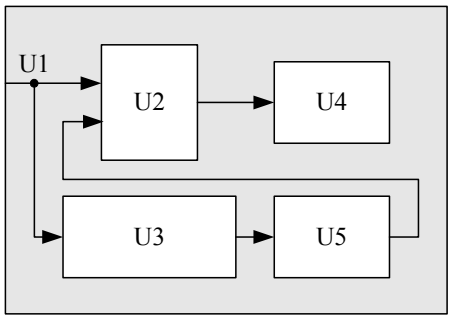
```
entity Tvset is
   port (ON,OFF : one-bit input
         ...
         )
end entity Tvset;

architecture TV2000 of Tvset is
   ...
   if ON then turn on the system
   if OFF then turn off the system
   ...
end architecture TV2000;
```

Рис.2.5.

Структурний опис

Структурний опис не торкається функціонування системи, а замість цього визначає компоненти, що повинні використовуватися, і як вони повинні бути з'єднані, щоби досягти очікуваних результатів. Іншими словами, він описує внутрішню структуру системи.

Структурний проект набагато легше синтезувати, ніж поведінковий, тому що він будується на конкретних фізичних компонентах. Однак, створення структурного опису системи дещо складніше, оскільки вимагає детального знання компонентів і принципів їх дії для підвищення його ефективності.




```
entity Tvset is
   port (
         ...
         )
end entity Tvset;

architecture TV2000 of Tvset is
   ...
   U1,U5 -> U2 -> U4
   U1 -> U3 -> U5
   U3 -> U5 -> U2
   U2 -> U4
   ...
end architecture TV2000;
```

Рис.2.6.

Один інтерфейс – багато архітектур

Оскільки різні типи архітектури можуть виконувати однакові функції, система (*інтерфейс*) може бути описана з використанням різних архитектур. Наприклад, велика кількість процесорів 80xx51, що випускаються різними виробниками, можуть використовуватись один замість одного, оскільки вони мають той самий інтерфейс. Це означає, що одному *інтерфейсу* може бути співставлено багато *архитектур*.

Це правило немає зворотньої сили – оскільки архітектура не може мати декілька інтерфейсів, вона не може відповідати декільком інтерфейсам.

```
entity SIXTIUM is
  ...
end entity SIXTIUM;
```

```
architecture AMC of SIXTIUM is
  ...
end architecture AMC;
```

```
architecture IMTEL of SIXTIUM is
  ...
end architecture IMTEL;
```

```
architecture CEDAR of SIXTIUM is
  ...
end architecture CEDAR;
```

```
architecture LYRIX of SIXTIUM is
  ...
end architecture LYRIX;
```

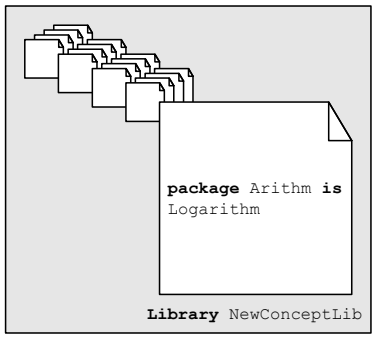
Рис.2.7.

Використання пакетів у VHDL

Деколи буває необхідно використовувати елементи, не визначені в стандартних VHDL-бібліотеках. В цьому випадку використовуються *пакети*.

Пакет – це набір типів, констант, підпрограм та ін., призначений для реалізації деякого сервіса або для відокремлення групи пов'язаних елементів. Крім того, пакети дозволяють приховувати реальні значення констант та тіла підпрограм, лишаючи видимими лише їх інтерфейси.

Єдиним обмеженням на використання пакетів є те, що вони повинні бути попередньо продекларовані, як правило, перед інтерфейсом. Є два оператора, за допомогою яких декларується пакет: ***library*** і ***use***. Їх використання пояснюється наступним прикладом.




```
library NewConseptLib;
use NewConseptLib.Arithm.Logarithm;

entity SomeSyst is
…
end entity SomeSyst;

architecture FirstArch of SomeSyst is
   …
   Logarithm
   …
end architecture FirstArch;
```

Рис.2.8.

Визначені пакети

VHDL – це потужне середовище проектування, що має декілька добре визначених пакетів. Найбільш популярні пакети, визначені IEEE:

STANDARD – містить всі базові декларації і визначення конструкцій мови, міститься в усіх VHDL-специфікаціях за замовчуванням;

TEXTIO – містить декларації базових операцій над текстом;

STD_LOGIC_1164 – не є частиною VHDL-стандарту, але є стандартом де факто, він містить найчастіше вживані розширення мови.

Пакет	Зміст	Використання
STANDARD	декларації всіх стандартних типів, операторів та інтерфейсів	використовується за замовчуванням в довільному VHDL-компіляторі і не вимагає використання операторів "***library***" і "***use***"

TEXTIO	декларації типів та інтерфейсів, що відносяться до читання і запису тексту у відповідності з VHDL-стандартом; ці оператори не синтезуються і можуть використовуватись тільки для симуляції і моделювання	перед описом інтерфейсу необхідно включити оператори
		
		`library Std;`
		`use Std.TextIO.all;`

STD_LOGIC_1164	розширення до стандартної мови VHDL: багатозначна логіка, функції аналізу і розширені оператори	перед описом інтерфейсу необхідно включити оператори
		
		`library IEEE;`
		`use IEEE.Std_Logic_1164.all;`

Крім цих трьох IEEE-пакетів, кожен продавець VHDL-засобів додає (і заохочує використовувати) його власні пакети. У таких випадках бібліотека, як правило, носить ім'я продавця.

3. Сигнали у VHDL: базові типи та декларування сигналів. Опис системного інтерфейсу: заголовок інтерфейсу, оператори *Port* і *Generic*.

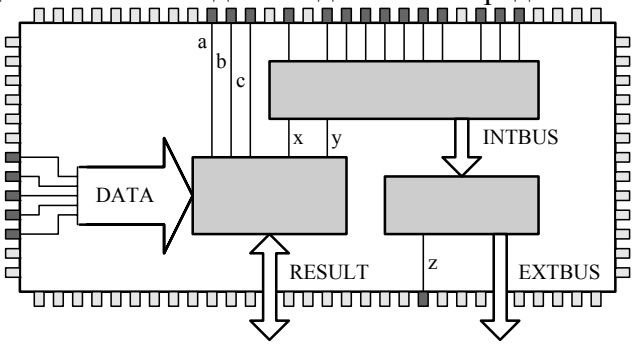
Базові типи сигналів

Одиничний сигнал та множинні сигнали

Електронні проекти будуються на компонентах і сигнальні лінії з'єднують ці компоненти. Сигнальні лінії можуть бути реалізовані або як однопровідні, або як багатопровідні лінії зв'язку.

Однопровідний зв'язок представляється сигнальною лінією, що має єдине двійкове значення в будь-який момент часу. Прикладом такого сигналу може бути тактова частота, що синхронізує усі блоки всередині системи.

Деякі проекти містять багатопровідні сигнали, які називаються шинами або векторами, що передають інформацію як комбінацію деяких двікових значень. Якщо кажуть про 32-розрядний процесор, то це означає, що по його шині даних одночасно передається 32 біти двійкових даних.



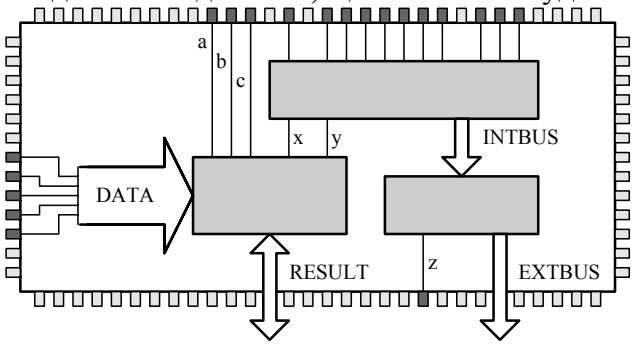
DATA, RESULT, INTBUS, EXTBUS: шина; a, b, c, x, y, z: провідник;

Рис.3.1.

Типи сигналів: bit і bit_vector

"Одиничний сигнал" і "вектор" – описові назви. Назви цих типів, що використовуються для формальної специфікації сигналів у VHDL – ***bit*** (для одиничних сигналів) і ***bit_vector*** (для шин). В обох випадках кожна сигнальна лінія може приймати значення '0', або '1'. Для типу ***bit_vector*** необхідно також визначати ширину шини.

Цих двох типів сигналів достатньо для того, щоби описати будь-який сигнал у VHDL.



DATA, RESULT, INTBUS, EXTBUS: **bit_vector**; a, b, c, x, y, z: **bit**;

Рис.3.2.

Ширина шини і порядок бітів

У випадку одиничного сигналу все просто – достатньо просто вказати: "цей сигнал має тип ***bit***" – і усе зрозуміло.

Однак, опис шин є більш складний. По-перше, необхідно вказати, що це – груповий сигнал, додаючи слово ***_vector*** до слова ***bit*** і утворюючи тип ***bit_vector***. По-друге, ширина шини і порядок нумерованих бітів повинні бути задані явно. Наприклад, дуже велике значення має те, чи є розряд

номер 7 найстаршим значащим бітом (MSB), а номер 0 – наймолодшим значащим битом (LSB) у шині, чи навпаки. Ця інформація вказується через діапазон вектора, що завжди задається в круглих дужках, наприклад (7 ***downto*** 0).

Діапазон вектора визначається у VHDL шляхом використання двох ключових слів: ***to*** або ***downto***. Перше служить для визначення діапазону, що зростає, (коли номер лівої границі менше ніж номер правої), а друге визначає діапазон, що спадає (коли номер лівої границі більше ніж номер правої).

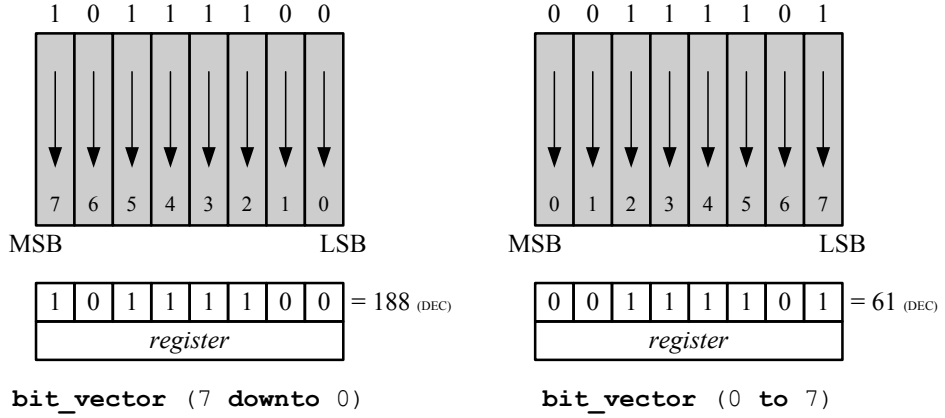


Рис.3.3.

Декларування сигналів

Зовнішні та внутрішні сигнали

Сигнали поділяються на зовнішні та внутрішні:

- *Зовнішні* сигнали – це сигнали, що з'єднують систему із зовнішнім світом. Іншими словами, вони формують інтерфейс системи.
- *Внутрішні* сигнали, невидимі ззовні, містяться повністю всередині системи і є частиною внутрішньої архітектури, забезпечуючи зв'язки між внутрішніми схемами.

Подібна відмінність існує між інтерфейсом та архітектурою проекту. Взагалі, зовнішні сигнали можуть бути задекларовані тільки в інтерфейсі, а внутрішні – тільки в архітектурі.

Декларування інтерфейсних сигналів

Сигнали, що з'єднують систему з її середовищем, називаються *портами* і описуються в секції (операторі) ***port*** інтерфейсу, який визначає системний інтерфейс. Кожний сигнал, який з'єднує систему з оточуючим світом, описується як *порт* всередині *інтерфейсу*.

Кожний сигнал повинен мати унікальне ім'я і тип. Крім того, *порт* повинен мати напрямок потоку інформації, який називається *режимом* (***mode***).

Напрямок потоку інформації: режим порта

Напрямок потоку інформації є ключовим питанням і обов'язково має бути вказаний для кожного порту, чи є він входом (*input*), виходом (*output*), або двонаправленим (*inout*). Напрямок сигналу повинен бути заданий явно (якщо він не заданий, за замовчуванням він приймається як вхід ***in***). У VHDL це робиться шляхом присвоєння відповідного режиму кожному інтерфейсному сигналу. Отже, декларація порта має наступний синтаксис:

```
port_name : mode port_type
```

Незважаючи на те, що VHDL підтримує п'ять режимів сигналів: ***in***, ***out***, ***inout***, ***buffer*** і ***linkage***, рекомендується використовувати тільки перші три, оскільки два інших підтримуються не усіма засобами синтезу:

'***in***' – всередині архітектури дані можуть бути прочитані, але не можуть бути записані;

'***out***' – всередині архітектури дані можуть бути згенеровані (записані), але не можуть бути прочитані;

'***inout***' – всередині архітектури дані можуть бути як записані так і прочитані.

В реальній специфікації усі порти декларуються в одному операторі ***port*** і розділяються символом ';':

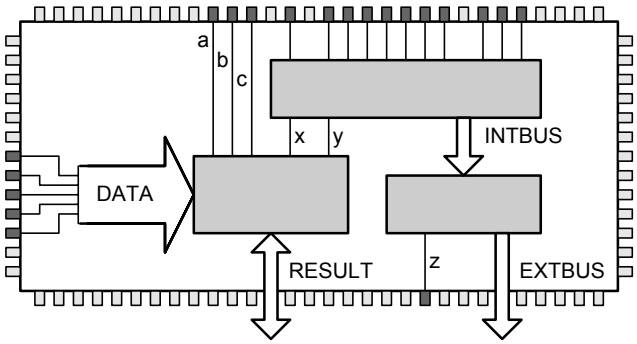
```
port ( A : inout bit_vector (0 to 7);
       B : in bit;
       C : out bit_vector (4 downto 0)
     );
```

Декларування внутрішніх сигналів

Блок інтерфейсу визначає інтерфейс системи із зовнішнім світом, а архітектура описує все, що знаходиться всередині системи. Внутрішні сигнали не є виключенням, і тому, щоби відрізнити їх від інших конструкцій VHDL-коду, в кожній декларації повинне використовуватися ключове слово ***signal***.

В операторі ***port*** ключове слово ***signal*** не потрібне, оскільки кожний порт за визначенням є сигналом.

Крім того, для внутрішніх сигналів не потрібно вказувати режим (***in***, ***out***, або ***inout***).




```
ENTITY
  port ( DATA: in bit_vector (7 downto 0);
         RESULT: inout bit_vector (0 to 7);
         EXTBUS: out bit_vector (4 downto 0);
         z: in bit;
         a,b,c: out bit
       );

ARCHITECTURE
  signal x,y: bit;
  signal INTBUS: bit_vector (7 downto 0);
  ...
```

Рис.3.4.

Області видимості сигналів

"Область видимості" кожного сигналу визначена місцем, де він декларується:

- сигнал, задекларований в пакеті, видимий в усіх елементах проекту, які використовують цей пакет;
- сигнал, задекларований як порт інтерфейсу, видимий в усіх архітектурах, що відносяться до цього інтерфейсу;
- сигнал, задекларований в декларативній частині архітектури, видимий тільки всередині цієї архітектури;
- сигнал, задекларований в блоці всередині архітектури, видимий тільки всередині цього блоку.

Ці правила випливають безпосередньо з ієрархічності проекту: якщо щось задекларовано на одному ієрархічному рівні, то воно буде видиме тільки в цій частині проекту і на всіх його нижчих рівнях.

```
package MyPack is
  signal A
                                                    1
```

```
use MyPack
entity En2 is
  port (C  )
                                                    3
```

```
use MyPack
architecture Arch2 of En1 is
  signal E
                                                    5
```

```
entity En1 is
  port (B  )
                                                    2
```

```
architecture Arch1 of En1 is
  signal D
                                                    4
```

```
architecture Arch3 of En2 is
  signal F
                                                    6
```

Області видимості сигналів: **A**: 1, 3, 5, 6 **B**: 2, 4, 5 **C**: 3, 6 **D**: 4 **E**: 5 **F**: 6

Рис.3.5.

Заголовок інтерфейсу

Ім'я інтерфейсу

Ім'я інтерфейсу, яке формально називається *ідентифікатором*, в першу чергу потрібне для документування проекту. Тому рекомендується вибирати в якості імен слова, або фрази, що найкраще описують функціонування системи. Оскільки мова VHDL є нечутлива до регістру символів, для більшої зручності ідентифікатор може записуватись символами змішаних регістрів (наприклад. *FastBinary_Cntr*).

Кожен ідентифікатор VHDL, включно з іменем інтерфейсу, повинен відповідати наступним правилам:

- вміщатися в один рядок;
- починатися з букви;
- може бути складений тільки з букв, цифр і знаків підкреслення;
- не може мати знаків підкреслення на початку, в кінці, а також подвійних знаків підкреслення;
- всередині ідентифікатора пробіли не допускаються;
- літери верхнього та нижнього регістрів не розрізняються;
- зарезервовані слова не можуть використовуватися в якості ідентифікаторів.

Так названі *розширені ідентифікатори* (які можуть містити зарезервовані слова і спеціальні символи) допускаються, але не рекомендуються.

Приклади ідентифікаторів:

```
Counter_4Bit    ALU    Receiver    Mux_4_To_1    UART_Transmit
```

Коментарі у VHDL

Кожний проект має бути документований. Документування проектів виконується за допомогою коментарів.

Коментарі у VHDL починаються із двох дефісів (**--**) і закінчуються кінцем рядка. Вони можуть бути розташовані будь-де, але не повинні починатися всередині ідентифікатора або зарезервованого слова.

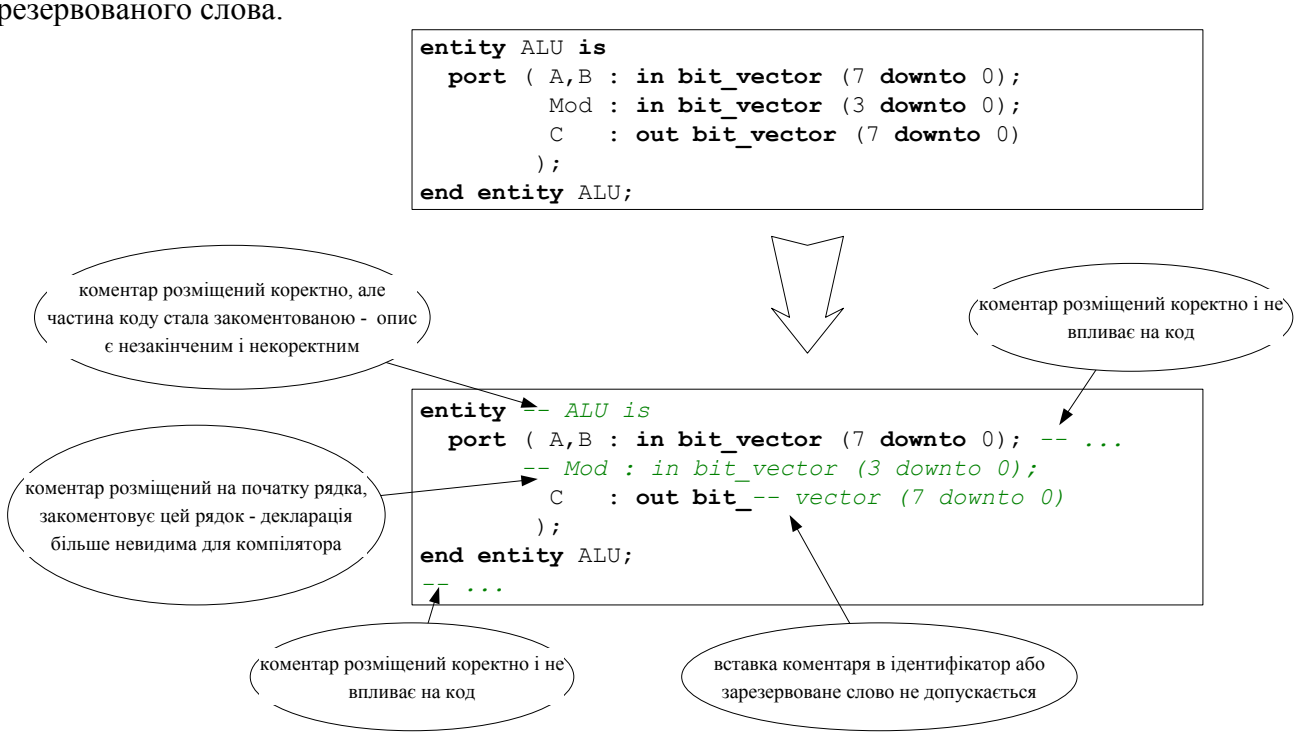


Рис.3.6.

Завершення інтерфейсу

Останній елемент шаблона інтерфейсу – це його завершення.

Інтерфейс закінчується оператором ***end***. Таким же чином закінчуються більшість інших VHDL-конструкцій. Для того, щоби підвищити читабельність, рекомендується використовувати обидва поля, які можуть слідувати за оператором ***end***: ключове слово ***entity*** і(або) ім'я інтерфейсу.

```
entity ExampleEnt is
  generic(...);
  port(...);
end;
```

дозволене та коректне завершення інтерфейсу, але не рекомендоване

```
entity ExampleEnt is
  generic(...);
  port(...);
end entity;
```

дозволене та коректне завершення інтерфейсу

```
entity ExampleEnt is
  generic(...);
  port(...);
end entity ExampleEnt;
```

дозволене та коректне завершення інтерфейсу - рекомендоване

Рис.3.7.

Оператор Port

Що таке порт

The VHDL Language Reference Manual визначає *порти* як "канали для динамічного зв'язку між блоком (наприклад, інтерфейсом), і навколишнім середовищем". На практиці цими каналами є сигнали, які формують інтерфейс системи. Оскільки усі зв'язки проходять через порти, кожний порт повинен бути точно визначений.

Кожний порт визначається оператором *port*, який складається з наступних елементів:

- необов'язкове ключове слово ***signal***;
- ім'я порту, за яким слідує двокрапка;
- режим порту;
- тип порту;
- необов'язкова ініціалізація – початкове значення, якому передує символ '**:=**';
- необов'язковий коментар, що описує порт (рекомендується).

```
entity CPU is
  port ( CS,RST : in bit;
         DATA   : inout bit_vector (15 downto 0);
         ADDR   : out bit_vector (19 downto 0)
       );
end entity CPU;
```

ім'я	опис	режим	тип	примітка
CS,RST	Chip Select, Reset	вхід	біт	декілька портів, що мають однакові тип та режим, можуть декларуватись в одному рядку
DATA	Data Bus	двонаправлений	16-розрядна шина	декларації портів розділяються крапкою з комою
ADDR	Address Bus	вихід	20-розрядна шина	остання декларація порта не закінчується крапкою з комою

Рис.3.8.

Режими порта

Порти виконують різні призначення: одні отримують вхідні дані, інші відправляють вихідні результати, а ще інші виконують і одне, і друге. Напрямок потоку даних через порт визначає *режим* порта. Є п'ять допустимих режимів:

- ***in***: порт тільки отримує дані; цей сигнал можна читати (записувати заборонено);

- ***out***: порт тільки відправляє дані; цей сигнал можна писати (читати заборонено);
- ***inout***: порт двонаправлений; допускається як читання так і запис сигналу;
- ***buffer***: подібний до *inout*, але обмежений тільки записом;
- ***linkage***: також двонаправлений порт, але з дуже обмеженими правилами читання і запису; в проектах практично ніколи не використовується.

При синтезі рекомендується використовувати тільки перші три режими.

Оператор Generic

Що таке Generic

The VHDL Language Reference Manual визначає *параметри* (***generics***) як канали статичної інформації, яка надходить в блок з навколишнього середовища. Іншими словами, параметри забезпечують значення констант для різних характеристик, які задаються поза системою.

Це означає, що декларація параметра повинна розміщуватись всередині інтерфейсу системи, який забезпечує зв'язок системи із навколишнім середовищем. Параметри використовуються для опису постійних значень. Наприклад, з їх допомогою можна керувати розміром моделі, зокрема шириною шин і розміром таких параметризованих блоків як *n*-розрядні суматори або компаратори. Вони можуть також використовуватися для задання часових параметрів, лічильників циклів, і т.ін.

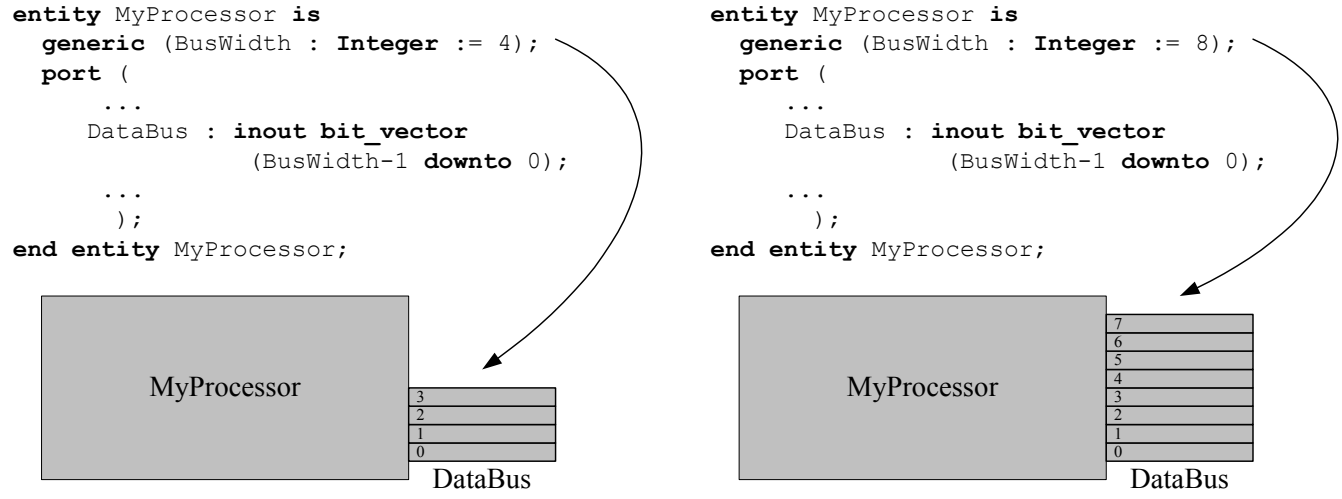


Рис.3.9.

Специфікація параметрів

Параметри описуються в інтерфейсах всередині оператора ***generic***, який розташовується перед описом портів. Оператор ***generic*** складається з ключового слова ***generic*** і списку параметрів в круглих дужках. Так само, як і порти, об'єкти параметрів в списку відокремлюються крапками з комою.

Кожна декларація параметра складається з наступних елементів:

- ім'я параметра, після якого ставиться двокрапка;
- тип параметра;
- необов'язкова ініціалізація – значення параметра, якому передує символ '**:=**'; якщо значення не задане, це означає, що система використовується як *компонент* на вищих рівнях специфікації, і це значення повинне бути задане при реалізації компонента;
- необов'язковий коментар, що описує параметр.

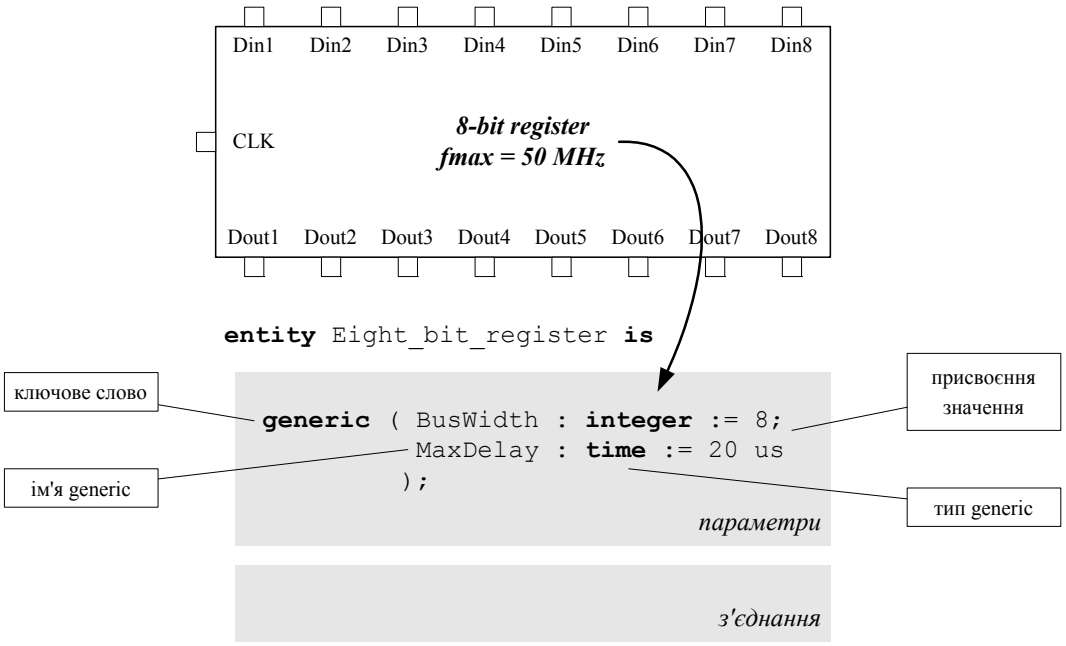


Рис.3.10.

Застосування параметрів

Параметри можуть використовуватися будь-де у коді, де необхідні статичні (тобто незмінні в часі) значення. Рекомендується замість жорсткого кодування використовувати параметри і константи, оскільки це спрощує внесення змін до проекту. Через механізм параметрів задаються:

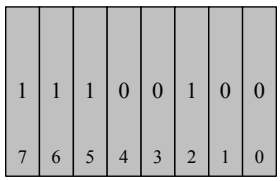
- розміри складних об'єктів, таких як масиви та шини (так якщо параметри використовуються для визначення ширини шини, змінити розмір шини можна, змінюючи оператор ***generic*** на початку специфікації);
- лічильники циклів;
- часові параметри: затримки, час встановлення, час призупинення, час переключення та інші аналогічні параметри, які використовуються при описі електронних пристроїв. Наприклад, для того, щоби замінити повільний пристрій на більш швидкий (без зміни його функціональності), потрібно тільки змінити значення параметрів, які описують часові параметри пристрою, якщо ж часові параметри задані жорстко, необхідно перевірити і змінити кожний рядок коду, який містить ці параметри.

опис
параметр в якості розміру об'єкту

декларація параметру
```
generic (BusWidth : integer := 8);
```

використання у VHDL-коді
```
port (DataBus : inout bit_vector (BusWidth-1 downto 0);
```

ілюстрація

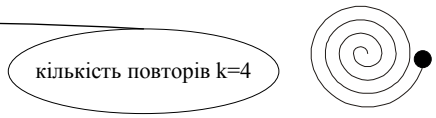


опис
параметр в якості лічильника циклу

декларація параметру
```
generic (LoopIterations : integer := 4);
```

використання у VHDL-коді
```
for k in 1 to LoopIterations loop ... end loop;
```

ілюстрація



опис
параметр в якості часового параметра

декларація параметру
```
generic (PropDel : time := 3 s);
```

використання у VHDL-коді
```
Y <= X after PropDel;
```

ілюстрація

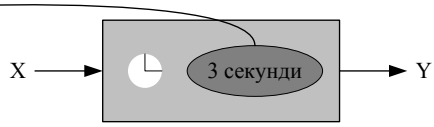


Рис.3.11.

4. VHDL-конструкції для опису поведінки системи: нелогічні типи даних, вирази та оператори, константи.

Нелогічні типи даних

Типи та підтипи

Більшість мов програмування передбачає різні типи даних, і мова VHDL не є виключенням. Однак, оскільки ця мова використовується для представлення апаратних проектів, засоби типізації даних мають особливо важливе значення.

В мові VHDL реалізована сувора типізація. Це означає, що змішування різних типів в одній операції (за деякими виключеннями) є помилкою.

Тип – це поіменована множина значень з деякими загальними характеристиками. *Підтип* – це підмножина значень даного типу.

Типи та підтипи визначаються в

- пакеті,
- архітектурному тілі,
- процесі.

Приклад визначення типу:

```
type byte_int is range 0 to 255;
```

Підтип визначається як тип із обмеженням (уточненням) наступним чином

```
subtype ім'я підтипу is базовий тип [уточнення];
```

Декларація підтипу не визначає новий тип. Оскільки значення підтипу відносяться до базового типу, між об'єктами підтипу та базового типу не потрібно виконувати ніяких конвертацій. Крім того, весь набір операцій, визначений для операндів базового типу, можна застосовувати до операндів підтипу.

Приклад визначення підтипу:

```
subtype nibble_int is byte_int range 0 to 15;
```

Скалярні типи

Скалярний тип – це узагальнене ім'я, яке об'єднує усі типи даних, об'єкти яких мають єдине значення в будь-який момент часу. Скалярні типи не мають елементів або внутрішньої структури. Усі значення скалярного типу упорядковані в певному діапазоні або явно перераховані.

У VHDL визначено декілька скалярних типів. Крім того можна визначати власні типи на підставі діапазону, або перечисленні значень. Звичайно такі визначені користувачем типи декларуються як *підтипи* типів, визначених раніше, як стандартних, так і визначених користувачем.

Стандартні скалярні типи даних:

- **BOOLEAN**

  декларація: `type Boolean is (false, true);`

  опис: на відміну від традиційного проектування на рівні вентилів, у VHDL значення типу ***Boolean True***/***False*** не еквівалентні логічним значенням 1/0; отже не можна вважати, що ***True*** – це те саме, що 1 і навпаки; типи ***Bit*** і ***Boolean*** – це абсолютно різні типи у VHDL;

  приклади: ***false***, ***true***

- **CHARACTER**

  декларація: `type Character is (nul, soh,…, 'a', …);`

  опис: набір значень, описаний як ***Character***, містить усі символи, визначені 8-розрядним символьним набором ISO 8859-1 (відомим також як Latin-1); зокрема він містить псевдографічні символи та національні символи західноєвропейських мов;

  приклади: '0', '+', '@', 'A', '{'

- **INTEGER**

  декларація: `type Integer is range –2147483647 to 2147483647;`

  опис: діапазон значень типу ***Integer*** залежить від реалізації, але має містити діапазон, наведений вище; при синтезі тип ***Integer***, як правило, обмежений деяким піднабором повного типу (наприклад від 0 до 15, або від 0 до 99) для зменшення витрат на пам'ять;

  приклади: -12; 0; 2147483647; -100; 16

- **REAL**

  декларація: `type Real is range –1.0E308 to 1.0E308;`

  опис: тип ***Real***, який також називається типом з плаваючою комою, визначається діапазоном, що залежить від реалізації; наведений діапазон є мінімально необхідним і має бути забезпеченим у будь-якій реалізації; крім того, усі реалізації повинні гарантувати точність з мінімум шістьма десятковими знаками після коми;

  приклади: 0.0; 1.000001; -1.0E5

- **BIT**

  декларація: `type Bit is ('0', '1');`

  опис: тип ***Bit*** – це перечислимий тип, який визначає два стандартних логічних значення: '0' і '1'; це єдиний тип, який може використовуватись у логічних операціях і, на відміну від проектування на вентильному рівні, не можна вважати, що логічні значення еквівалентні булевим значенням; типи ***Bit*** і ***Boolean*** – це абсолютно різні типи у VHDL;

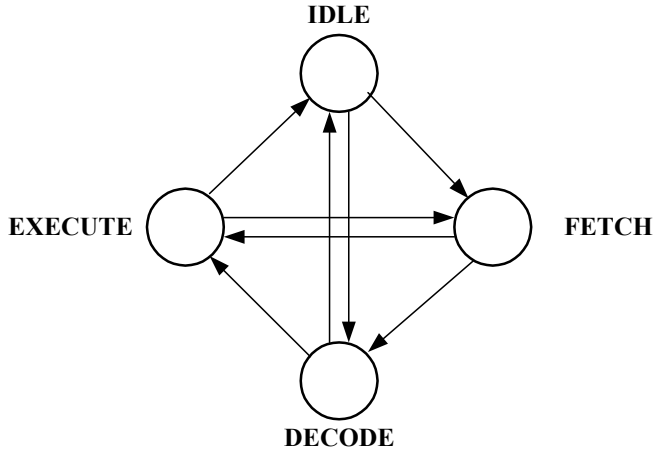
  приклади: '0', '1'.

Перечислимі типи, визначені користувачем

Проектувальникам часто доводиться кодувати деяку системну інформацію для підвищення ефективності або читабельності. Класичний приклад – скінчений автомат (*finite state machine – FSM*), який представляє послідовнісний проект.

З врахуванням особливостей синтезу кожний стан FSM кодується за допомогою змінних стану (тригерів), що зберігають інформацію про біжучий стан. Однак на рівні специфікацій зручніше присвоїти унікальні імена кожному стану і звертатись до цих станів за іменами. Усі імена станів задаються у вигляді списку при декларуванні нового перечислимого типу.

Значення в декларації типу (імена станів) задаються "як є", без апострофів або лапок.




```
type FSMStates is (Idle,Fetch,Decode,Execute);
```

Рис.4.1.

Фізичні типи

Фізичні типи унікальні у VHDL, тому що вони визначають не тільки значення об'єкта, а ще й одиниці, у яких ці значення виражені. Це дозволяє використовувати такі фізичні величини як час,

відстань, струм, температуру, і т.ін., які визначаються з необхідною точністю. Стандарт VHDL визначає тільки один фізичний тип – ***time*** (час), однак інші фізичні типи при бажанні також можуть бути визначені.

Є два види одиниць у деклараціях фізичних типів: *первинні* одиниці і *вторинні* одиниці, які визначаються в термінах первинних одиниць. Діапазон, зазначений у заголовку декларації типу відноситься лише до первинних одиниць.

Незважаючи на те що фізичні типи дуже наочні і привабливі для описів, вони не синтезуються.

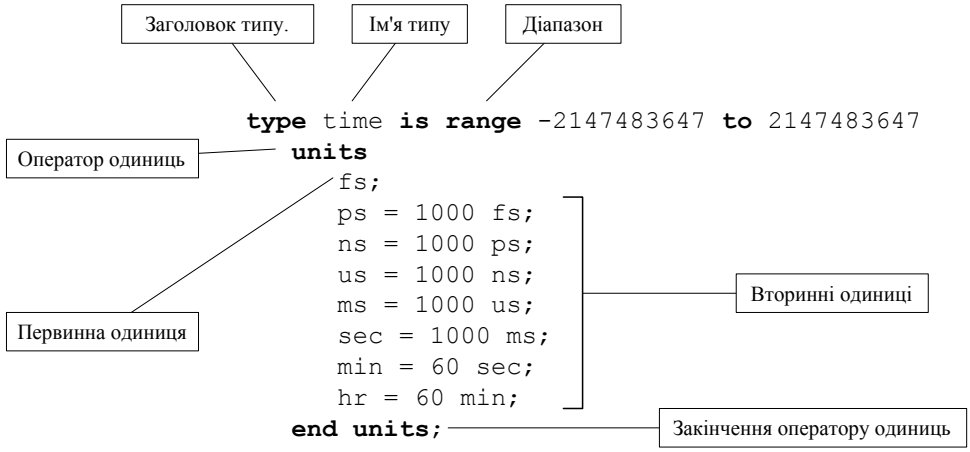


Рис.4.2.

Заголовок типу. Кожна декларація типу починається з ключового слова ***type***, після якого вказуються ідентифікатор типу і ключове слово ***is***. На відміну від інтерфейсу та більшості конструкцій VHDL по закінченні декларації не ставиться ключове слово ***end***.

Ім'я типу. Ім'я типу повинно відповідати тим самим правилам, що і будь-який ідентифікатор у VHDL, при цьому літери верхнього та нижнього регістрів не розрізняються.

Діапазон. Для кожного фізичного типу повинно бути задано діапазон його одиниць. Діапазон може бути як зростаючим, так і спадаючим, і задається як з цілими, так і з дійсними границями.

Оператор одиниць. Оператор одиниць (***units***) використовується тільки для декларування фізичних типів. Одиниці поділяються на *первинні* та *вторинні*. Обов'язковою є тільки первинна одиниця. Усі вторинні одиниці визначаються прямо або непрямо в термінах інтегральних множників первинної одиниці.

Первинна одиниця. Це єдина обов'язкова одиниця в декларації фізичного типу. Усі вторинні одиниці визначаються по відношенню до первинної із застосуванням інтегральних множників до неї. Отже, якщо сантиметр буде визначений як первинна одиниця, не можна визначити дюйм як вторинну одиницю, оскільки 1 дюйм дорівнює 2.54 сантиметри, а це не інтегральний множник.

Вторинні одиниці. Це додаткові одиниці, які визначаються як кратні первинній одиниці (або іншим вторинним одиницям, визначеним раніше). Можна змішувати різні одиниці. Так, метричні одиниці можуть змішуватись з дюймовими, якщо вони визначаються по відношенню до однієї первинної одиниці, наприклад міліметру.

Закінчення оператору одиниць. Цей рядок закінчує список одиниць, які визначаються для фізичного типу, і декларацію типу взагалі. Рядок може додатково містити ім'я типу, в цьому випадку він буде записаний як "**end units** time;".

Визначені масиви

Масиви – складний тип з регулярною структурою, який складається з елементів одного типу. Кількість елементів масиву визначається діапазоном масиву. Діапазон може бути необмеженим або обмеженим, з "підтримкою" лівої або правої границі і напрямку індексації. Необмежений діапазон визначається як:

```
range <>
```

У VHDL є два визначених масиви – ***bit_vector*** (з елементами типу ***bit***) і ***string*** (з елементами типу ***character***). Обидва вони декларуються з необмеженим діапазоном. Однак в цих масивах по різному визначається нижня границя: індексація елементів ***bit_vector*** починається з 0, а індексація ***string*** – з 1.

Одиничний елемент цих масивів вказується в апострофах, а два або більше елементів, які називаються *зрізом* (*slice*), вказуються в лапках.

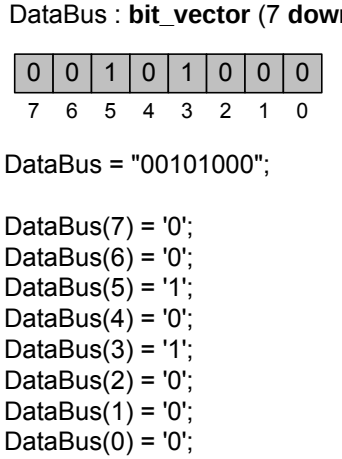


Рис.4.3.

Масиви, визначені користувачем

Оскільки визначені масиви є одновимірними, їх називають векторами. Користувач може декларувати масиви з довільним числом вимірів, але не рекомендується використовувати більше ніж трьохвимірні масиви, оскільки це погіршує читабельність проекту.

Типовий приклад двовимірного масиву – пристрій пам'яті, який може розглядатися як двомірний масив бітів. В реальній пам'яті біти, як правило, групуються в слова. Це практично перетворює пам'ять на "вектор векторів" – кожен елемент вектора у свою чергу є вектором. Приклади пам'яті двох типів наведені на рис.4.4.

Багатовимірний масив задається наступним чином:

```
type dim_array is array (dimrange1,dimrange2,…,dimrangeN) of elemtype;
signal name : dim_array;
```

де *dimrange1*, *dimrange2*, *dimrangeN* – діапазони індексів відповідних вимірів масиву.

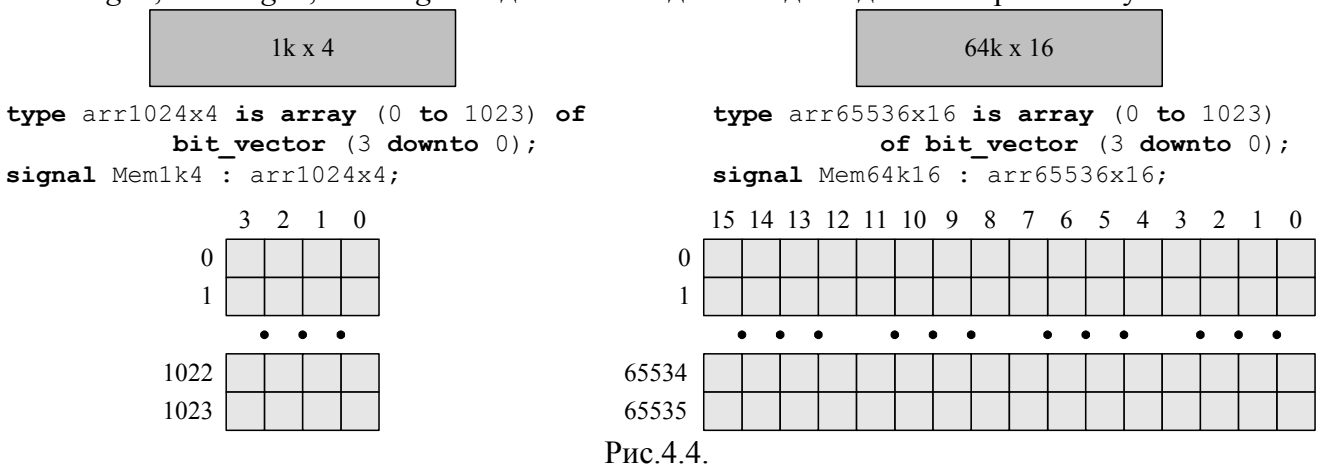


Рис.4.4.

Записи

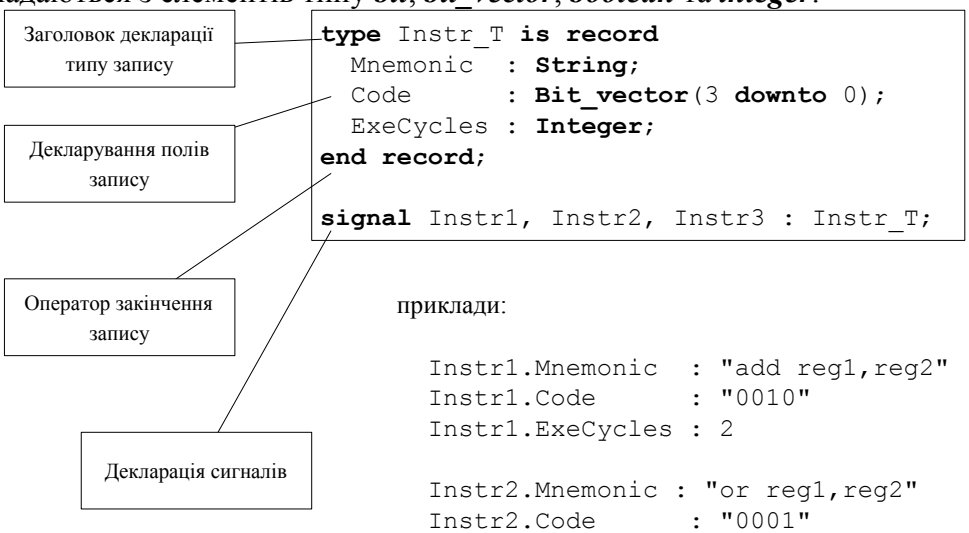
Записи, як і масиви, відносяться до складних типів. Однак між ними є дві суттєві відмінності: записи можуть містити елементи різних типів, а крім того до елементів записів звертаються за іменами, а не за індексами. Звернення до елементу запису має наступну форму:

```
record_name.element_name
```

Імена елементів завжди записуються без круглих дужок.

Головне призначення записів – згрупувати разом різнотипні значення одного об'єкту. До кожного такого об'єкту звертаються за одним ім'ям, що робить код більш компактним та читабельним.

Не кожний запис може бути синтезований, але більшість синтезаторів здатні обробити записи, які складаються з елементів типу ***bit***, ***bit_vector***, ***boolean*** та ***integer***.



приклади:

```
Instr1.Mnemonic  : "add reg1,reg2"
Instr1.Code      : "0010"
Instr1.ExeCycles : 2

Instr2.Mnemonic : "or reg1,reg2"
Instr2.Code      : "0001"
Instr2.ExeCycles : 1

Instr3.Mnemonic  : "mul reg1,reg2"
Instr3.Code      : "1110"
Instr3.ExeCycles : 8
```

Рис.4.5.

Заголовок декларації типу. Як і у випадку з іншими типами, декларація починається з ключового слова ***type***, після якого вказується ім'я типу. Ключове слово ***record*** ідентифікує клас типу і вказує на початок декларування полів.

Декларування полів запису. Кожне поле описується ім'ям та його типом. Прийнято кожне поле розташовувати в окремому рядку для наочності. Якщо два або більше полів мають той самий тип, вони можуть бути описані разом, при цьому імена полів розділяються комами, наприклад: `Name,Surname : string`.

Оператор закінчення запису. Цей оператор закінчує як список полів, так і декларацію типу.

Декларація сигналів. Сигнали типу запису не вимагають якоїсь особливої декларації. Необхідне лише ім'я типу. Від інших типів вони відрізняються способом звертання до полів запису.

Вирази та оператори

Логічні оператори

Оскільки сигнали в цифрових системах звичайно є логічними сигналами, найбільш часто використовується клас логічних операторів VHDL. Логічні оператори виконують такі двохоперандні операції як ***and***, ***or***, ***nand***, ***nor***, ***xor*** і ***xnor***. Операція ***not*** – це одноопернадна операція. Логічні операції визначаються для типів: ***Bit***, ***Boolean*** та ***Bit_Vector***, причому обидва операнди повинні мати один і той самий тип. Результат завжди має такий самий тип, як і операнди.

У випадку векторів бітів логічні операції виконуються побітово, тобто перший біт результуючого вектора обчислюється з перших бітів операндів, другий – з других бітів операндів, і т.д. Отже, операнди і результуючий вектор повинні мати однакову довжину.

A	B	A **and** B	A **or** B	A **nand** B	A **nor** B	A **xor** B	A **xnor** B	**not** A
0	0	0	0	1	1	0	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	1	0	0
1	1	1	1	0	0	0	1	0

Рис.4.6.

Чисельні оператори

Ця загальна назва відноситься до операторів, які виконують операції над чисельними операндами.

Операція	Символ	Опис	Приклади	
додавання	+	Додає два чисельних значення одного типу; результат має той самий тип, що і обидва операнди. Застосовується до цілих, дійсних, часу, універсальних цілих та універсальних дійсних. Універсальні значення можуть додаватись до аналогічних значень (універсальні цілі - до цілих, універсальні дійсні - до дійсних).	`IntX+17`	за умови, що IntX - ціле
			`RealX2+2.0`	якщо RealX2 - дійсне, 2 описує дійсне число
			`3ns + 1us`	1003 ns
віднімання	-	Віднімає чисельне значення від іншого, такого самого типу, результат має той самий тип, що і обидва операнди. Застосовується до цілих, дійсних, часу, універсальних цілих та універсальних дійсних. Універсальні значення можуть відніматись від аналогічних значень (універсальні цілі - від цілих, універсальні дійсні - від дійсних).	`BusWidth-1`	якщо BusWidth - ціле
			`-1.0-Dat`	-1.0 - дійсне, отже дійсним має бути Dat
			`8.33-5`	Заборонено (конфлікт типів - обидва мають бути дійсними або цілими)
множення	*	Перемножує два чисельних значення, тип результату залежить від типів операндів: якщо обидва операнди цілі або дійсні (загальні або універсальні значення) - результат має той самий тип, що і обидва операнди; якщо один з операндів має тип ***time***, результат теж буде типу ***time***.	`2*Pi*R`	площа кола
			`Mult*5ns`	добуток часу - Mult може бути цілим або дійсним
			`4*SomeVal`	дозволено, якщо SomeVal ціле або час; дійсне має множитись на 4.0
ділення	/	Ділить два чисельних значення, тип результату залежить від типів операндів: якщо обидва операнди цілі або дійсні (загальні або універсальні значення) - результат має той самий тип, що і обидва операнди; якщо перший операнд має тип ***time***, результат теж буде типу ***time***.	`CLK/2`	половина тактової частоти
			`5.0/2.0`	результат - дійсне значення 2.5
			`10ns/2ns`	результат 5 (ціле, а не час)
ділення за модулем	mod	Ділить два цілих чисельних значення для отримання результату цілочисельного ділення	`6 mod 4`	результат 2
			`6 mod (-4)`	результат -2
			`(-6) mod 4`	результат 2
залишок від ділення	rem	Операція залишку від ділення дає залишок цілочисельного ділення. Вона визначена тільки для цілих операндів і її результат - також ціле.	`6 rem 4`	результат 2
			`6 rem (-4)`	результат 2
			`(-6) rem 4`	результат -2
експонування	**	Експонування, де другий операнд цілий, еквівалентне циклічному множенню лівого операнда самого на себе стільки разів, скільки задано в другому операнді. Лівий операнд може бути цілим або дійсним.	`A**2`	рівне A*A
			`B**3`	рівне B*B*B
			`C**0.5`	заборонено в VHDL
абсолютне значення	abs	Абсолютне значення - це унарний оператор, тобто він має тільки один операнд. Визначено для довільного чисельного типу (ціле, дійсне або час). Результат того самого типу, але без знаку (завжди додатній).	`abs 1`	результат 1
			`abs (-1)`	результат 1
			`abs(5*(-2))`	результат 10

Рис.4.7.

До чисельних операндів відносяться *додавання*, *віднімання*, *множення*, *ділення*, *ділення за модулем*, *залишок від ділення*, *експонування* та *абсолютне значення*. Ці операції можуть виконуватись як над цілими, так і над дійсними числами, але вимагають, щоби обидва операнди були одного типу. Виключенням є експонування, де другий операнд завжди цілий.

Типи обох операндів повинні співпадати. Ця вимога настільки сувора, що навіть так звані *універсальні цілі* (ціле значення, вказане явно) не можуть використовуватись з дійсними числами.

Крім цілого та дійсного типів в чисельних операторах може використовуватись також *час*. В цьому випадку правила не такі суворі: *додавання* та *віднімання* вимагають обидвох операндів типу ***time***, але значення часу може бути помножене на ціле або дійсне, або поділене на ціле. В кожному випадку результат буде мати тип ***time***.

Оператори відношення

При необхідності порівняння двох об'єктів використовуються оператори *відношення*. До них відносяться: *рівне*, *не рівне*, *менше*, *менше або рівне*, *більше*, *більше або рівне*. Об'єкти, що порівнюються, повинні відноситись до одного типу і можуть бути ***Boolean***, ***bit***, ***character***, ***integer***, ***real***, ***time***, ***string*** або ***bit_vector***. Однак результат завжди має тип ***Boolean*** і є істинний або хибний.

Необхідно зазначити, що коли порівнюються два вектори бітів, вони не обов'язково повинні бути однакової довжини. Замість того, щоби дописувати додаткові нулі до коротшого операнда, обидва операнди вважаються вирівняні по лівому краю, наприклад, 1011 буде менше, ніж 110.

A	B	A = B	A /= B	A < B	A <= B	A > B	A >= B
0	0	***true***	***false***	***false***	***true***	***false***	***true***
0	1	***false***	***true***	***true***	***true***	***false***	***false***
1	0	***false***	***true***	***false***	***false***	***true***	***true***
1	1	***true***	***false***	***false***	***true***	***false***	***true***

Рис.4.8.

Оператори зсуву

Оператори *зсуву* зсувають елементи одновимірних масивів. Цей клас операцій обмежений масивами, елементи яких мають тип ***bit*** або ***Boolean***.

Оператори зсуву вимагають двох операндів. Лівий операнд представляє масив, а правий – ціле значення, яке вказує, на скільки позицій вміст масиву повинен бути зсунутий. Якщо значення другого операнда від'ємне, напрямок зсуву змінюється, наприклад *"**sll** –n"* еквівалентне *"**srl** n"* (зсув вліво на *-n* – це те саме, що зсув вправо на *n*), *"**ror** –n"* виконує таку саму операцію, що і *"**rol** n"* і т.д.

У випадку операторів логічногу зсуву в найлівішу позицію (для ***srl***) або в найправішу позицію (для ***sll***) вставляється значення, що формально визначається як перше значення типу елементу. Для типу ***bit*** таким значенням є '0', а для типу ***Boolean*** – '***false***'.

```
signal MyBus: bit_vector (7 downto 0) := '01101001'
```

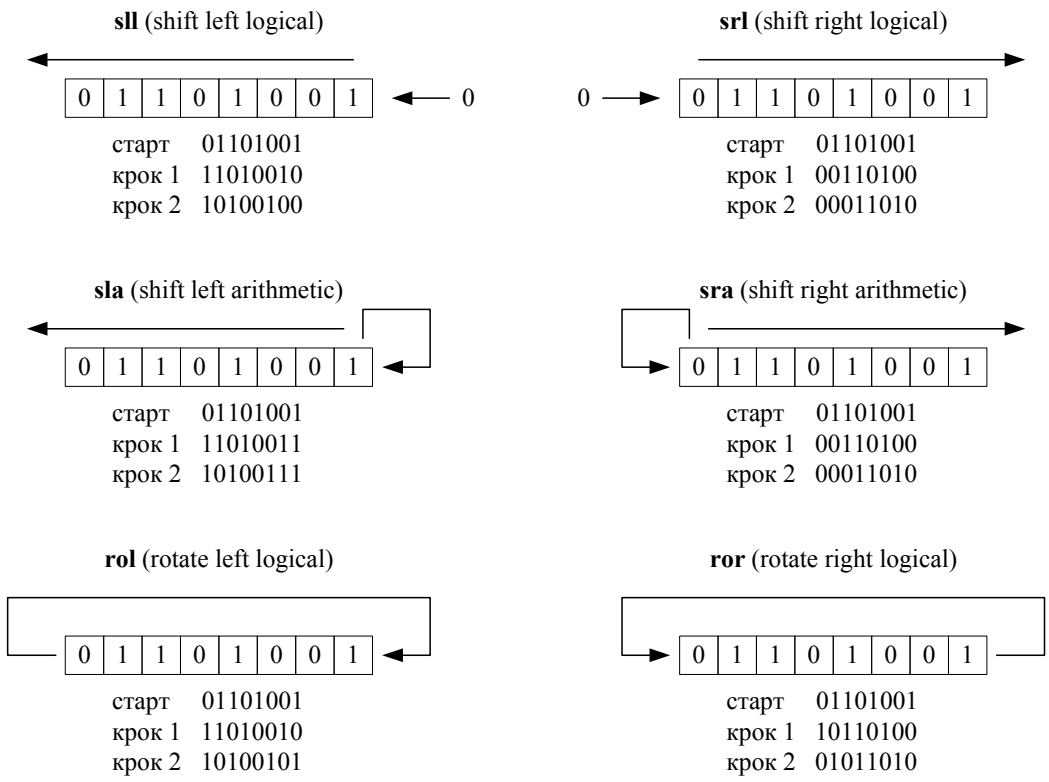


Рис.4.9.

Оператор конкатенації

Конкатенація – це зручний спосіб створення нових значень масивів будь-якого типу. Ця операція визначена для одновимірних масивів.

Операція *конкатенації* з'єднує два операнди за допомогою оператора *конкатенації*. Результатом є масив, довжина якого є сумою довжин обох операндів. Послідовність масивів зберігається, тобто в результуючому масиві значення елементів лівого операнду знаходяться зліва, а значення елементів правого операнда – справа.

Конкатенація не обмежується операндами масиву – дозволяється з'єднувати як масиви, так і окремі значення (окремі біти), якщо тип окремого значення і тип елементів масиву (другий операнд) співпадають.

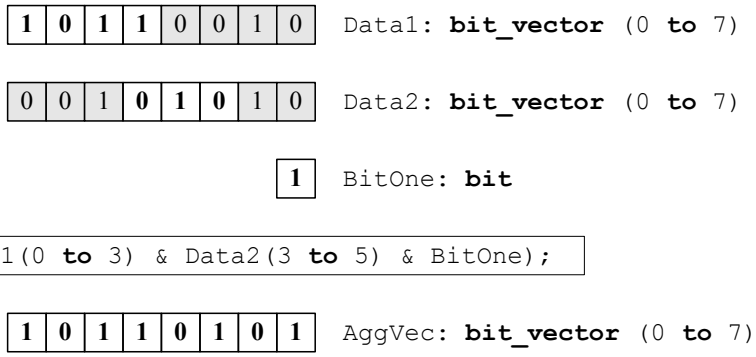


Рис.4.10.

Присвоєння сигналів

Результати операцій мають бути присвоєні вихідним сигналам. Механізм присвоєння простий: результуючий сигнал вказується зліва; за ним вказується символ присвоєння сигналу (<=), а після нього описується вираз присвоєння. Результат виразу буде переданий результуючому

сигналу, вказаному зліва. Для того, щоби краще запам'ятати цей символ, можна розглядати його як стрілку, що вказує напрямок потоку інформації.

Символ присвоєння сигналу завжди напрямлений справа наліво ('<=’). Подібний до нього символ ('=>’) також використовується у VHDL, але він має інше значення. Крім того, обидва символи '<=' і '=>' можуть використовуватись в операторах відношення. Їх інтерпретація залежить від контексту.

`x <= y <= z;`	Це не конвейєрне присвоєння (*z* в *y*, і потім в *x*), це присвоєння значення виразу відношення (*y* менше або рівне *z*) булевому сигналу *x*. Таке дещо дивне присвоєння є цілком коректним (хоча не зовсім зрозумілим для початківців).
`x <= b or c;`	Типове присвоєння сигналу значення логічної операції. Такі присвоєння є коректними для операндів типів ***boolean***, ***bit*** і ***bit_vector***.
`k <= '1';`	Присвоєння значення одиничного біта або символа (в залежності від контексту). Дуже важливо вказувати значення в апострофах, якщо це значення типу ***bit*** або ***character***. Апострофи не використовуються для значень типів ***boolean***, ***integer***, ***real*** та ***enumeration***.
`m <= "0101";`	Присвоєння значення ***bit_vector*** або рядка (в залежності від контексту і типу сигналу *m*). Значення вказується в подвійних лапках. Кількість елементів справа повинна відповідати кількості елементів сигналу зліва.
`n <= m & k;`	Вектори бітів можуть бути об'єднані з одиничними бітами для формування нового вектора. Необхідно, щоби результуючий сигнал був задекларований з відповідною кількістю елементів. Якщо ширина *m* - 4 біти, а *k* - одиничний біт, *n* повинно бути 5-розрядним вектором бітів.

Рис.4.11.

В реальності нічого не відбувається негайно. Події завжди відбуваються одна за одною. Така поведінка моделюється у VHDL за допомогою оператора ***after***, який може бути добавлений до будь-якого оператора присвоєння.

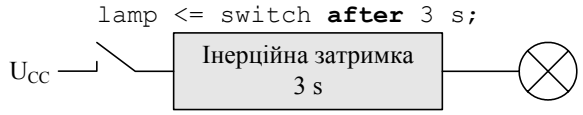
Оператор ***after*** описує інтервал часу, який повинен пройти, перш ніж відбудеться присвоєння. При цьому значення '0' в операторі ***after*** означає відсутність затримки взагалі.

Затримки

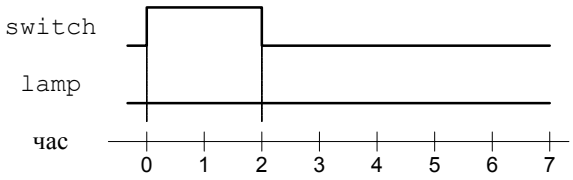
Інерційні затримки

Коли реакція системи затримана, постає важливе питання: наскільки швидко зміни на вході відіб'ються на виході? Якщо світлодіод переключається дуже швидко, збоку буде виглядати, ніби він світиться неперервно. Така властивість світлодіода називається *інерцією* і моделюється у VHDL за допомогою *інерційної затримки*.

Інерційна затримка є типовою для більшості реальних систем, в зв'язку з чим у VHDL ця модель є затримкою за замовчуванням. Оператор ***after*** автоматично вважає затримку інерційною. Характерною властивістю цієї моделі є те, що дві послідовних зміни вхідного сигналу будуть проігноровані, якщо час між ними коротший, ніж задана затримка.

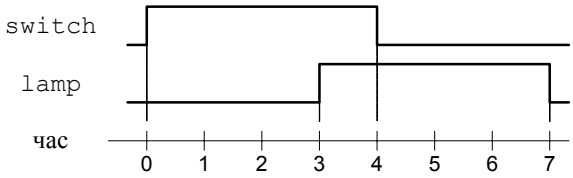


switch включити, потім виключити через 2 s



Інтервал між переключеннями switch коротша, ніж інерція системи, результатом чого є його "ковтання" - lamp не реагує.

switch включити, потім виключити через 4 s



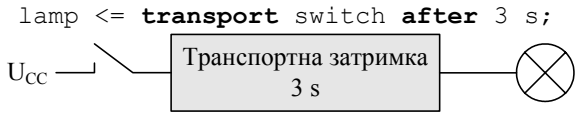
Інерційна затримка коротша, ніж інтервал між послідовними змінами значення на вході, приводить до очікуваної (затриманої) реакції на виході.

Рис.4.12.

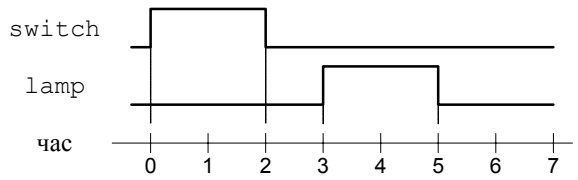
Транспортні затримки

Інерційна затримка є типовою для більшості електронних об'єктів, але не для всіх. Наприклад, лінія передачі, по якій передаються імпульси довільної ширини – оскільки ніякої інерції в цьому випадку не може бути, використання моделі інерційної затримки приводить до некоректного моделювання ліній передачі.

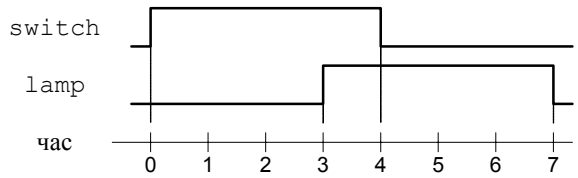
Виходячи з поведінки лінії передачі, необхідно мати іншу модель затримки. Оскільки ця модель застосовується для транспортування сигналів без будь-яких змін, вона називається моделлю *транспортної затримки*. Для того, щоби відрізнити її від інерційної моделі, прийнятої у VHDL за замовчуванням, використовується ключове слово ***transport***, яке вказується перед описом значення затримки.



switch включити, потім виключити через 2 s



switch включити, потім виключити через 4 s



Незважаючи на довжину інтервал між переключеннями switch, транспортна затримка розповсюджує його дію на вихід.

Рис.4.13.

Порівняння інерційних і транспортних затримок

Моделі *інерційної* і *транспортної* затримки є достатніми для опису у VHDL довільної фізичної системи. Вони мають наступні головні подібності та відмінності:

Інерційна затримка	Транспортна затримка
є затримкою за замовчуванням у VHDL і не вимагає ніяких додаткових декларацій	вимагає використання ключового слова ***transport***.
не поширює імпульси, коротші ніж задана затримка	поширює всі зміни вхідного сигналу, незалежно від того, як швидко і як часто вони відбуваються
описується за допомогою оператора ***after*** після якого вказується значення часу	
може застосовуватись до сигналів довільного типу	

```
Out1 <= Input after 3 s;
Out2 <= transport Input after 3 s;
```

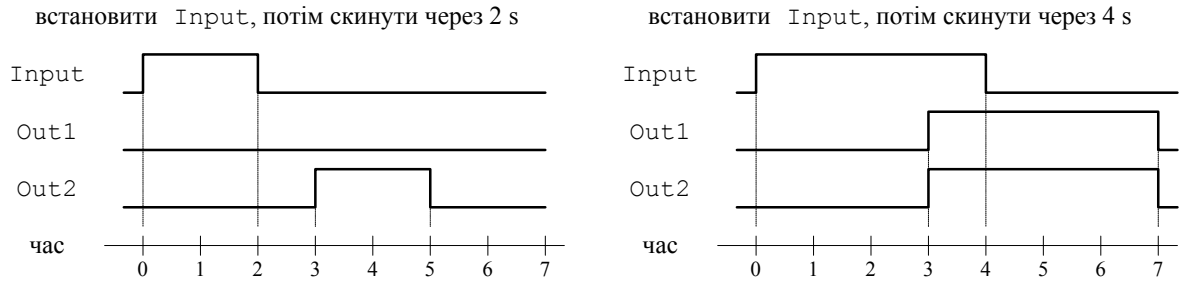


Рис.4.14.

Концепція дельта-затримок

Найпростіший оператор присвоєння сигналу (без затримки) складається з цільового сигналу та виразу, що визначає його значення. Таке присвоєння визначає, що значення виразу буде присвоєне сигналу через час *дельта*. Фізично цей час дорівнює нулю, але з точки зору обробки значень сигналів він ненульовий, тобто є логічним Наприклад, присвоєння, яке має відбутися через час, рівний двом *дельтам*, буде виконане після присвоєння, що має відбутися через одну *дельту*, при цьому результат більш пізнього присвоєння недоступний для більш ранніх присвоєнь. Однак обидва присвоєння відбудуться скоріше, ніж пройде найменший фізичний інтервал часу. Іншими словами, на протязі будь-якого фізичного інтервалу часу може бути довільна кількість *дельта*-інтервалів.

В наступному описі використовуються оператори присвоєння з нульовими затримками для внутрішніх вузлів схеми та затримане присвоєння для виходу *Z* із затримкою 36 наносекунд.

Припустимо, що зовнішні сигнали *A*, *B* і *C* проініціалізовані значенням ‘1’, отже їх значення перед моментом 0 та в момент 0 рівні ‘1’. Припустимо також, що зовнішній для цього опису сигнал *A* в момент 0 приймає значення ‘0’. Оскільки *A* – сигнал, це нове значення з’явиться пізніше, через *дельта*-інтервал в момент 1δ. Через один *дельта*-інтервал після зміни значення *A* нові значення отримають вузли *W* та *X*, отже *W* стає рівним ‘1’ (значення ***not*** *A*), а *X* стає рівним ‘0’ (значення *A* ***and*** *B*) в момент 2δ. Подія з *X* приводить до присвоєння нульового значення виходу *Z* через 36 наносекунд. Подія з *W* приводить до переобчислення виразу для *Y* і як результат – до зміни значення вузла *Y* через один *дельта*-інтервал після зміни *W*; це значення змінюється з ‘0’ в ‘1’ в момент 3δ. Після цього подія з *Y* приводить до переобчислення вихідного виразу, результатом якого знову буде присвоєння нового значення виходу через 36 наносекунд після цієї події.

Друге присвоєння виходу *Z* відмінює попереднє, і оскільки значення *Z* вже дорівнює ‘1’, воно не спричинює події на цій лінії. Оскільки встановлений стан значення *Z* є правильним, проміжні значеня *Z* не моделюються у відповідності до реальної схеми. Необхідно зауважити, що події, які виникають на *X* і *Y* в момент 0, не транслюються на вихід *Z* через 36 наносекунд. Це пов’язано з тим, що фізична затримка часу зглажує всі дельта-затримки.

```
architecture Delta of SomeEnt1 is
signal W,X,Y : bit;
begin
  W <= not A;
  X <= A and B;
  Y <= C and W;
  Z <= X or Y after 36 ns;
end architecture Delta;
```

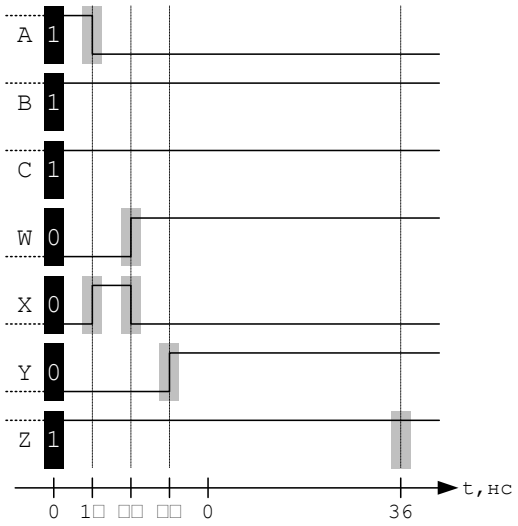


Рис.4.15.

Наступний опис наведено для кола з двох послідовно включених інверторів з нульовими затримками, де *A* – це вхід, *C* – вихід, а *B* – проміжна точка. Сигнали *A*, *B* і *C* ініціалізуються значенням '0'. Сигналу *A* присвоюється значення '1', і це спричинює зміну сигналу *B*, переключення якого генерує зміну для *C*.

Часова діаграма демонструє, що всі зміни сталися в момент 0 між $0+1\delta$ та $0+3\delta$. Кожна зміна приводить до події. В момент 0 сигнали *A*, *B* і *C* мають значення, визначені в декларації сигналів. В той час, коли '1' присвоюється *A*, інверсія *A*, значення якого все ще рівне '0' в момент 0, присвоюється *B*. Також в момент 0, інверсія сигналу *B*, значення якого все ще рівне '0', присвоюється сигналу *C*. Через один *дельта*-інтервал в 1δ сигнали *A*, *B* і *C* отримають нові значення, які всі рівні '1'. Нове значення *A* генерує присвоєння для *B* через один *дельта*-інтервал в 2δ, що спричинює подію зміни значення *B* на '0'. Аналогічно, значення *B* в 1δ спричинює подію для *C* в 2δ. Подія з *B* в момент 2δ приводить до переобчислення виразу оператора присвоєння для *C*, що в свою чергу приводить до ще однієї події з *C* через один *дельта*-інтервал в 3δ.

```
architecture Delta of SomeEnt2 is
signal A,B,C : bit := '0';
begin
  A <= '1';
  B <= not A;
  C <= not B;
end architecture Delta;
```

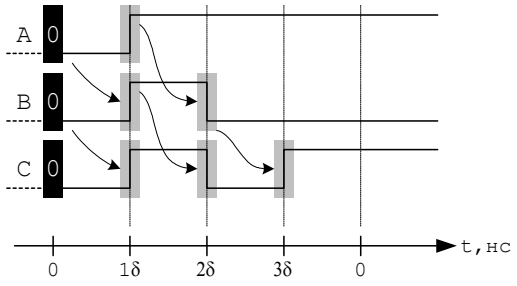


Рис.4.16.

Таким чином, дельта-затримки є логічними затримками і використовуються для визначення послідовності виконання одночасних подій.

Константи

Декларування констант

Константи відіграють ту саму роль, що і параметри: вони забезпечують статичну інформацію, яка може використовуватись всередині моделі. Однак, на відміну від *параметрів* (***generics***), що декларуються всередині інтерфейсів, *константи* декларуються всередині архітектур.

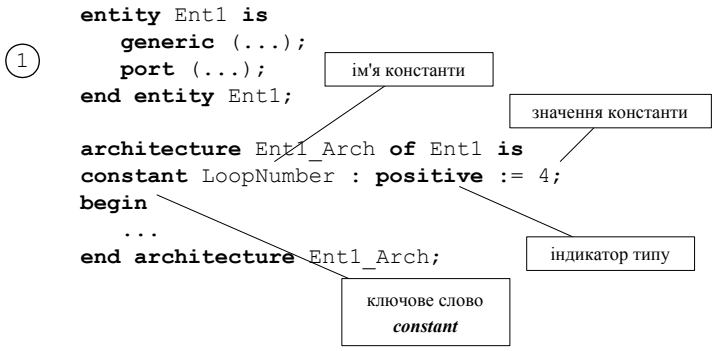
Декларація констант складається з наступних елементів:

- ключове слово ***constant***,
- ім'я константи (ідентифікатор константи),
- символ '**:**',

- індикатор типу константи,
- значення константи, задане після символа '**:=**',
- символ '**;**', яким закінчується рядок.

Якщо дві або більше констант мають однаковий тип і однакове значення, вони можуть бути описані в одній декларації.

В деяких особливих випадках присвоєння значення константі може бути відкладене.




```
2
architecture Ent1_Arch of Ent1 is
constant t_propagation : time := 10 ns;
begin
   ...
end architecture Ent1_Arch;

3
Block1 : block
constant N : integer := 123;
begin
   ...
end block Block_1;

4
p1 : process (A,B)
constant t_hold : time := 5 ns;
begin
   ...
end process p1;
```

Рис.4.17.

Використання констант

Головна мета використання констант полягає у використанні "зрозумілих імен" із зрозумілими визначеннями типів замість так званих "жорстко-кодованих" літеральних значень. При цьому код стає більш читабельним і простим у використанні, оскільки зміна константи в одному місці впливає на всі її входження в усій архітектурі. Якщо ж використовуються жорстко-кодовані значення, користувач повинен уважно перевірити весь опис рядок за рядком для того, щоби переконатись, що всі значення виправлені.

Взагалі константи використовуються так само, як параметри. Вони застосовуються зокрема для:

- опису розміру складних об'єктів (таких як масиви і шини);
- керування лічильниками циклів;
- визначення часових параметрів: затримок, часу встановлення, часу припинення, часу переключення, та ін.

Оскільки константи декларуються в архітектурах, вони не можуть використовуватись для визначення розміру векторів, задекларованих як порти в інтерфейсах. Однак, це обмеження можна обійти, якщо константа декларується в пакеті, що використовується інтерфейсом.

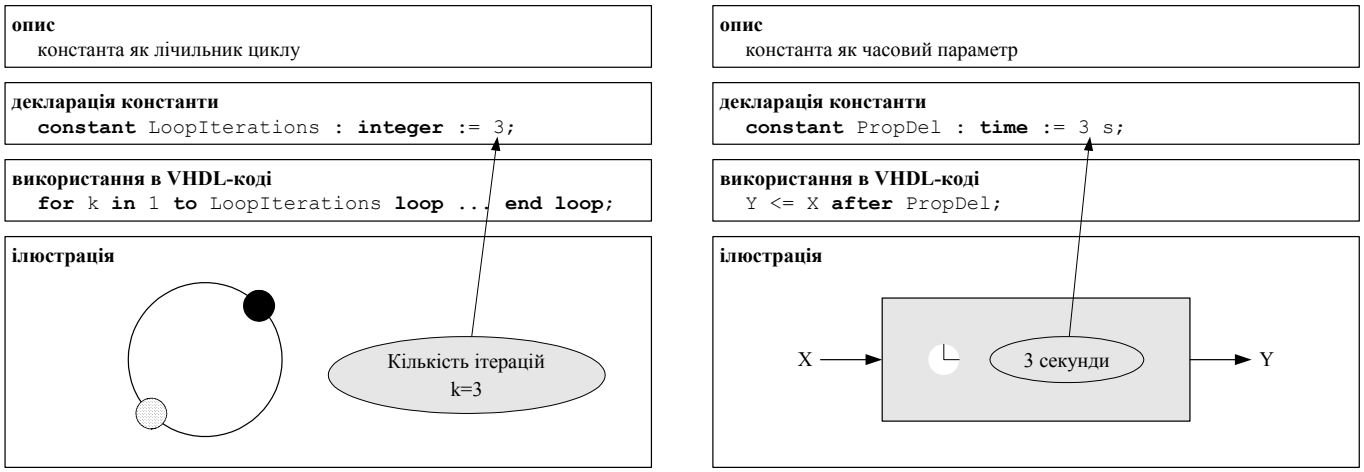


Рис.4.18.

Порівняння констант і параметрів

Може виникнути питання, навіщо VHDL містить дві такі подібні конструкції, як параметри і константи?

Головна відмінність між параметрами і константами полягає в тому, що параметри можуть використовуватись *динамічно*, а константи є виключно *статичними*. Це означає, що можна змінювати значення параметрів без будь-яких змін коду. З іншого боку, константи не можуть бути змінені без зміни коду. Це важливо особливо в тому випадку, коли специфікація використовується як *компонент* для опису на вищому рівні. Кожний раз при використанні такого компонента для нього можуть бути задані інші значення параметрів, якщо вони описані як параметри.

Інший аспект полягає в наступному. Один і той самий інтерфейс може використовуватись декількома архітектурами і усі параметри застосовуються до всіх архітектур інтерфейсу. Будь-яка зміна значення параметра впливає на всі його входження у всіх архітектурах. Якщо ж використовуються константи, їх зміни локалізуються лише у вибраній архітектурі.

	Параметр	Константи
Положення декларації	тільки в інтерфейсі (оператор *generic*)	в архітектурі або в пакеті

Декларація	список параметрів:
	`generic (generic_name: generic_type := optional_value;`
	`…`
	`generic_name: generic_type := optional_value);`
		одна декларація на константу:
		`constant constant_name: constant_type := value;`
		`constant constant_name: constant_type := value;`

Видимість	в інтерфейсі (в тому числі в операторі **port**) та в усіх архітектурах, що відносяться до цього інтерфейсу	в архітектурі (якщо задекларовано в архітектурі); в будь-якій частині проекту, де використовується пакет (якщо задекларовано в пакеті); це включає інтерфейси та архітектури, що відносяться до них

Рис.4.19.

5. Опис поведінки системи у VHDL: процеси, змінні, керування послідовністю виконання операторів.

Процеси

Опис поведінки

Головна мета будь-якої електронної ситеми або пристрою – трансформувати вхідні дані у вихідні результати. Такий тип діяльності називається "поведінкою" або "функціонуванням" системи, і опис системи задає порядок перетворення вхідних даних у вихідні результати.

Поведінковий опис є списком операцій, які повинні бути виконані для отримання очікуваних результатів.

машинний код

```
if event_occurred then buttons := "enabled"
if button1_pressed then output_result1
   else if button2_pressed then output_result2
            else output_result3
buttons := "disabled"
```

Рис.5.1.

Що таке процес?

Результат в попередньому прикладі залежить від послідовності операцій. Це дуже типово для більшості описів проектів. Коли хтось виконує процедуру для виконання чогось, він надає список дій, які повинні бути виконані послідовно крок за кроком.

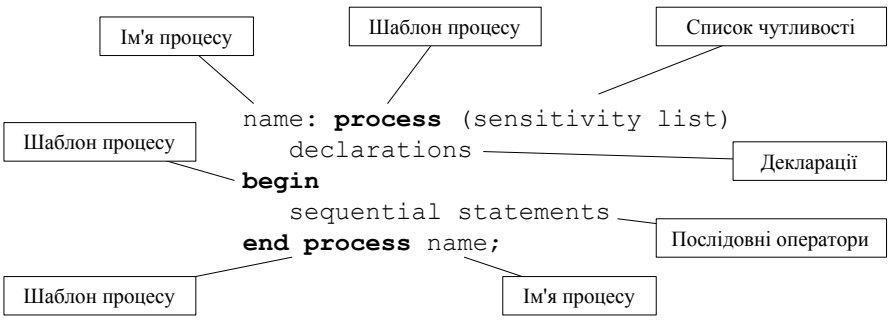
Процес – це формальний шлях для виконання таких послідовних операцій. Він має дуже структурований формат, навіть якщо ним представлено поведінку тільки незначної частини проекту.

Структура процесу

Процес декларується за допомогою ключового слова ***process***. Для покращення читабельності можна надати процесу ім'я. Таке ім'я повинно стояти перед ключовим словом ***process*** і закінчуватись двокрапкою. Це ж ім'я може бути повторене в кінці, відразу після оператора ***end process***, але без двокрапки.

Процес – це послідовність операцій (операторів). Список операторів процесу починається з ключового слова ***begin*** і закінчується ключовими словами ***end process***, після яких вказується ім'я процесу, якщо таке є.

Ключове слово ***begin*** вживається для того, щоби відокремити послідовні операції від декларацій змінних та констант. Воно вказується відразу після декларацій і перед списком послідовних операцій.




```
MUX2TO1: process (A,B,SEL)
   constant High : Bit := '1';
begin
   Y <= A;
   if (SEL='1') then Y <= B;
   end if;
end process MUX2TO1;
```

Рис.5.2.

Ім'я процесу є необов'язковим і, якщо задане, повинне передувати ключовому слову ***process*** і відокремлюватись від нього двокрапкою. Це ім'я може бути повторене в кінці процесу, відразу після оператора ***end process***, але без двокрапок.

Шаблон процесу містить три елементи, які містить кожний процес: ключове слово ***process*** на початку процесу, ключове слово ***begin*** на початку операторної частини і оператор ***end process*** в кінці процесу. Слід зауважити, що після перших двох операторів крапка з комою не ставляться.

Список чутливості. Виконання припиненого процесу може бути поновлено, коли один з сигналів його списку чутливості змінить своє значення. Поновлення та припинення процесів буде розглянуто далі.

Декларації. Всі декларації процесу розміщуються після заголовку процесу і перед ключовим словом ***begin***. Всі об'єкти, задекларовані всередині процесу, видимі тільки в цьому процесі. Сигнали не можуть декларуватись всередині процесу.

Послідовні оператори. Всі оператори в процесі, описані між ключовими словами ***begin*** і ***end process*** виконуються послідовно.

Виконання процесу

В процесі після виконання останнього оператора виконується негайний перехід на виконання першого оператора. Таким чином, процес ніколи не закінчується.

```
JustToShow: process
begin
   Some Statement 1;
   Some Statement 2;
   Some Statement 3;
   Some Statement 4;
   Some Statement 5;
end process JustToShow;
```

Всі оператори в процесі виконуються послідовно в нескінченому циклі. На відміну від звичайних мов програмування оператор *end* не закінчує виконання, а забезпечує тільки границю процесу.

Рис.5.3.

Оператор wait

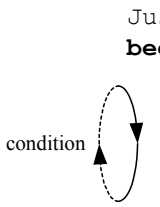
Реальні електронні пристрої працюють в нескінченому циклі, і після їх активації виконують внесені в список задачі, після чого повертаються в стан "очікування певної умови". Іншими словами, пристрій *припиняє* дії після того, як закінчує біжучу задачу, і *поновлює* дії після того, як знову виникнуть певні умови.

Така робота пристроїв описується у VHDL оператором ***wait***. Він використовується для

- безумовної зупинки (припинення) виконання процесу;

- задавання списку умов, що можуть поновити процес.

Якщо процес містить оператор ***wait***, він виконує послідовно всі оператори до оператора ***wait***. Після цього процес припиняється і очікує виконання умови, заданої в операторі ***wait***. Як тільки це відбувається, процес поновлюється і виконує всі оператори, поки знову не зустріне оператор ***wait***.




```
JustToShow: process
begin
    Some Statement 1;
    Some Statement 2;
    Some Statement 3;
    Some Statement 4;
    wait <condition>;
end process JustToShow;
```

Досягнувши оператора *wait*, процес припиняється і може бути поновлений тільки за умови виконання `condition`.

Рис.5.4.

Типи оператора wait

В зв'язку з тим, що в реальних аплікаціях процес може бути поновлений з різних умов, у VHDL є три типи оператора ***wait***: очікування на протязі певного часу, очікування виконання деякого логічного виразу і очікування зміни значення сигналу. Ці три типи конструкцій мають наступні форми:

- ***wait for*** *time_expression*,
- ***wait until*** *condition*,
- ***wait on*** *sensitivity_list*.

Якщо змішуються декілька таких умов, вони утворюють четверту конструкцію, яка називається *складною умовою*.

Тип оператора *wait*	**Опис**	**Приклади**
wait for time_expression	Припиняє процес на визначений час. Час може бути заданий безпосередньо, або як вираз, результатом якого є значення часу. Використовуються в основному для моделювання на тестових стендах.	`wait for 10 ns;` `wait for ClkPeriod / 2;`

wait until condition	Припиняє процес доти, доки задана умова не стане істинною завдяки зміні будь-якого сигналу, що містяться в умові. При цьому, якщо сигнал не змінюється, ***wait until*** не поновить процес, навіть якщо сигнал задовільняє умову.	`wait until CLK='1';`
		`wait until CE and (not RST);`
		`wait until IntData > 16;`

wait on sensitivity_list	Припиняє процес доти, доки не відбудеться подія з будь-яким з сигналів, вказаних в списку чутливості. Іншими словами, процес поновлюється тільки тоді, коли зміниться значення хоча б одного сигналу з списку чутливості.	`wait on CLK;` `wait on Enable, Data;`
складний ***wait***	Містить комбінацію двох або трьох різних форм оператора ***wait***.	`wait on Data until CLK='1';` `wait until Clk='1' for 10 ns;`

Рис.5.5.

Якщо симулятор знаходить оператор ***wait*** відразу на початку процесу, то процес буде негайно припинений і жодний оператор не буде виконаний. Якщо ж оператор ***wait*** розташований ближче до кінця процесу, деякі оператори будуть виконані, перш ніж процес буде припинений.

Оператор ***wait*** може розташовуватись в процесі будь-де. Але, як правило, він використовується або в кінці процесу, або на початку. Дозволяється також використання кількох операторів ***wait*** в процесі, що часто застосовується у тестових стендах.

```
process
begin
   wait on SigA;
   somestatement1;
   somestatement2;
   somestatement3;
end process;
```

Якщо оператор ***wait*** є першим оператором процесу, процес буде припинений відразу після старту.

```
process
begin
   somestatement1;
   somestatement2;
   somestatement3;
   wait on SigB;
end process;
```

Якщо оператор ***wait*** є останнім оператором процесу, всі оператори будуть виконані один раз, після чого процес буде припинений.

Рис.5.6.

Список чутливості процеса

Очікування за списком чутливості, тобто очікування зміни значення сигналу, найчастіше використовується в якості умови поновлення процесів. У VHDL застосовується конструкція, яка називається *списком чутливості процесу*. Цей список вказується відразу після ключового слова ***process*** і повністю відповідає оператору ***wait on*** *sensitivity_list* в кінці процесу.

Процес із списком чутливості не може містити явних операторів ***wait***.

```
process
begin
   somestatements1;
   somestatements2;
   somestatements3;
   wait on SomeSign;
end process;
```

```
process (SomeSign)
begin
   somestatements1;
   somestatements2;
   somestatements3;

end process;
```

Рис.5.6.

Коли починається симуляція, процес із списком чутливості виконується один раз, оскільки список чутливості еквівалентний оператору ***wait*** в кінці процесу, і після цього процес припиняється. Процес буде припиненим доти, доки будь-який сигнал з його списку чутливості не змінить свого значення. Така зміна поновлює процес і всі оператори процесу, з початку до кінця, будуть послідовно виконані. Під словом "всі" слід розуміти всі оператори процесу, а не тільки ті, що мають відношення до сигналу, який поновлює процес. Після того, як буде виконаний останній оператор, процес знову буде припинений.

```
process (Signal1,Signal2,Signal3)
begin
   SomeStatement1;
   SomeStatement2;
   SomeStatement3;
   SomeStatement4;
   SomeStatement5;
end process;
```

Зміна сигналу Signal2 поновлює процес, виконуються всі оператори процесу, після чого процес знову припиняється

Рис.5.8.

Змінні

Сигнали в процесах

Головна мета поведінкового VHDL-опису – задати реакцію виходів на зміну входів. Як входи, так і виходи є сигналами, отже реакція вихідних сигналів представляється як присвоєння

сигналів. Сигнали і присвоєння сигналів використовуються всередині процесів. Однак, використання сигналів в процесах регулюється трьома важливими обмеженнями:

- сигнали не можуть декларуватись всередині процесів;
- будь-які присвоєння сигналам виконуються тільки після припинення процесу; до того часу всі сигнали містять попередні значення;
- виконується тільки останнє присвоєння сигналу, вказане всередині процесу; отже немає сенсу присвоювати сигналу більше одного значення всередині даного процесу.

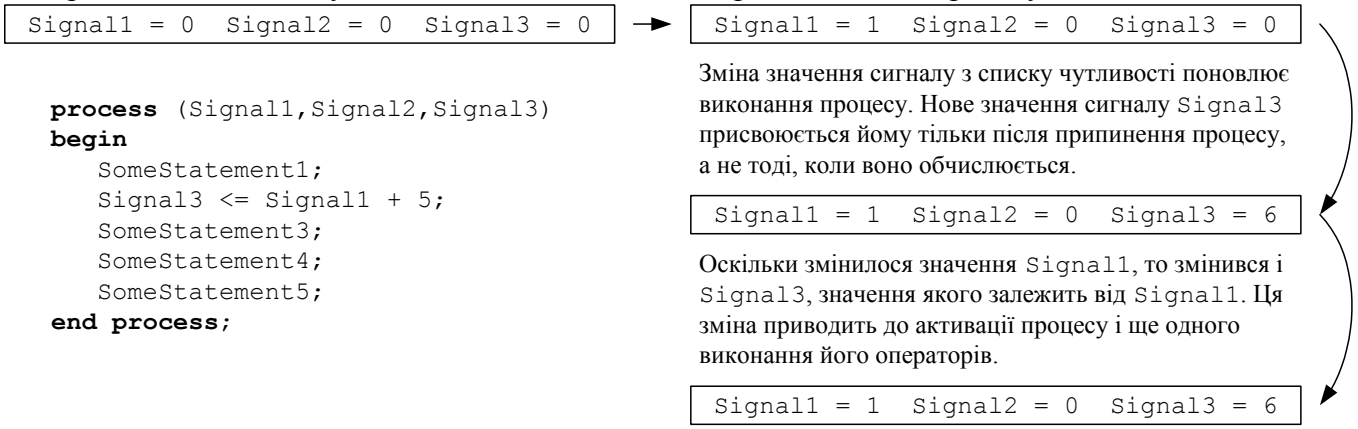


Рис.5.9.

Обмеження на використання сигналів має сильний вплив на їх практичне застосування. Неможливість декларування сигналів всередині процесів – не головна проблема, але правила присвоєння мають серьозні наслідки: оскільки сигнали можуть зберігати тільки останнє присвоєне значення, вони не можуть використовуватись для збереження проміжних та тимчасових данних всередині процесу. Інша важлива незручність полягає в тому, що нові значення присвоюються сигналам не під час виконання присвоєння, а після припинення процесу. Це значно ускладнює аналіз проекту.

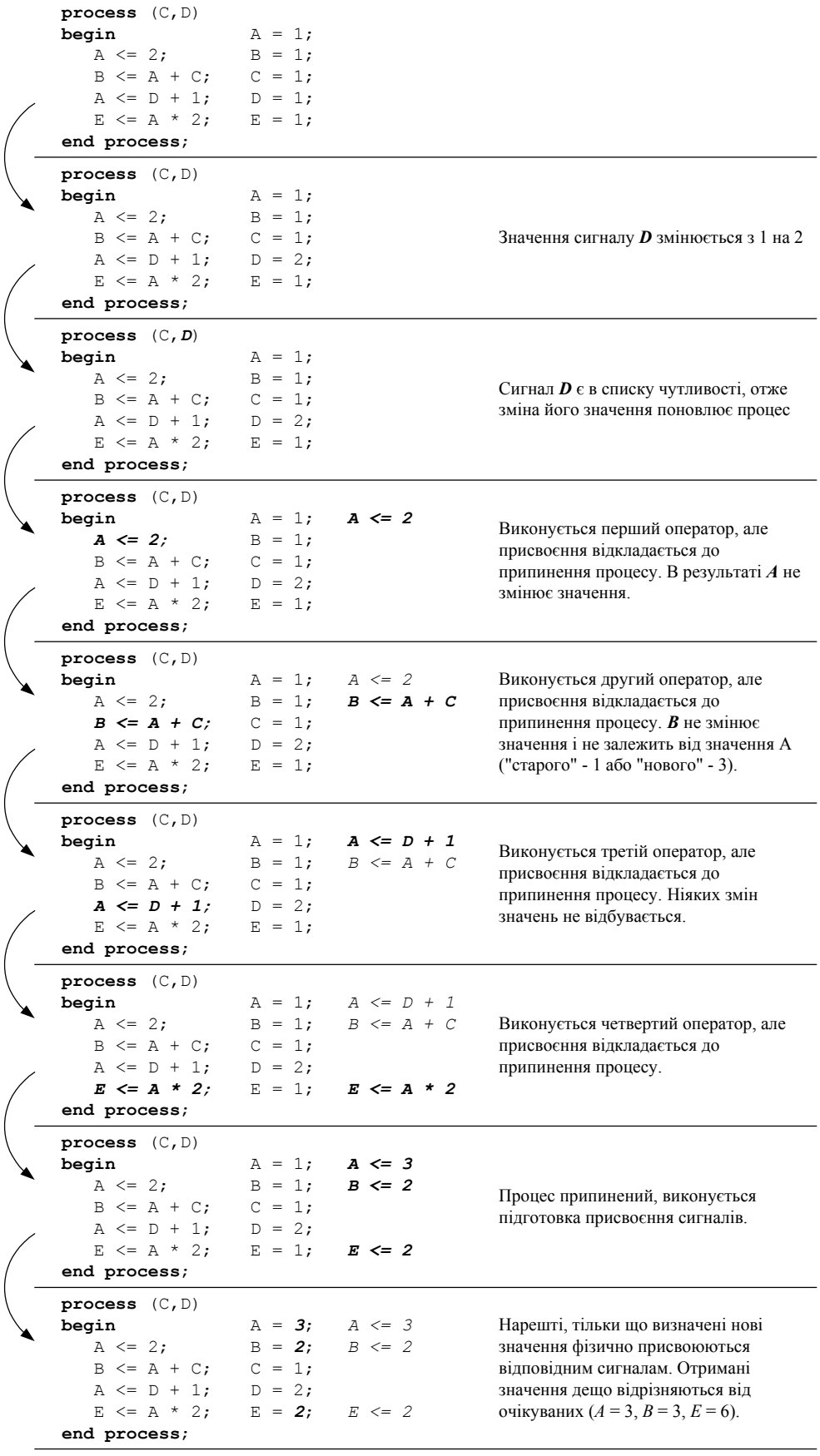


Рис.5.10.

Оскільки сигнали не можуть декларуватись всередині процесів і присвоєння ним значень виконується після припинення процесу, необхідний об'єкт, який можна було би декларувати всередині процесу, і який би забезпечував миттєве збереження тимчасових даних. Такий об'єкт у VHDL називається *змінною*.

Декларування та присвоєння змінних

Змінні подібні сигналам. Однак є і деякі відмінності.

Змінні обмежені процесами, вони декларуються в процесах і не можуть використовуватись поза ними. Декларація змінної виглядає аналогічно декларації сигналу. Єдина різниця полягає в тому, що змінні декларуються з ключовим словом ***variable*** замість ***signal***.

Присвоєння змінних виконується за допомогою символа '**:=**'. Це присвоєння виконується миттєво і кожній змінній може бути присвоєно нове значення стільки разів, скільки потрібно.

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 1;
E = 1;
```

Процес має типову структуру: вхідні дані передаються за допомогою сигналів, обчислення виконуються над змінними, а результати присвоюються вихідним сигналам, через які надходять в зовнішній світ. 1

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
```

Значення сигналу ***D*** змінюється з 1 на 2. 2

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
```

Сигнал ***D*** є в списку чутливості, отже зміна його значення поновлює процес 3

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    Av = 0;
                    Bv = 0;
                    Ev = 0;
```

Процес містить деякі декларації змінних, які ініціалізуються значенням '0'. Початкова ініціалізація є необов'язковою. 4

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    Av = 2;
                    Bv = 0;
                    Ev = 0;
```

Виконується перше присвоєння. Оскільки це присвоєння змінної, воно виконується негайно і значення змінної ***Av*** змінюється. 5

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    Av = 2;
                    Bv = 3;
                    Ev = 0;
```

Аналогічно попередньому присвоєнню змінної, ***Bv*** отримує нове значення без затримки. Це нове значення основане на новому значенні ***Av***. 6

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    Av = 3;
                    Bv = 3;
                    Ev = 0;
```

Змінній ***Av*** присвоюється нове значення. Це значення буде використовуватись у всіх наступних присвоєннях, в яких ***Av*** виступає як операнд. 7

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    Av = 3;
                    Bv = 3;
                    Ev = 6;
```

Змінній ***Ev*** присвоюється нове значення, обчислене на основі нового значення операнда ***Av***. 8

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    A <= Av  Av = 3;
                             Bv = 3;
                             Ev = 6;
```

Присвоєння сигналу відкладено до припинення процесу. Якщо сигнал має такий самий тип, що і змінна, їх значення можуть присвоюватись одне одному. 9

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    A <= Av  Av = 3;
                    B <= Bv  Bv = 3;
                             Ev = 6;
```

Присвоєння сигналу відкладено до припинення процесу. Якщо сигнал має такий самий тип, що і змінна, їх значення можуть присвоюватись одне одному. 10

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    A <= Av  Av = 3;
                    B <= Bv  Bv = 3;
                    E <= Ev  Ev = 6;
```

Присвоєння сигналу відкладено до припинення процесу. Якщо сигнал має такий самий тип, що і змінна, їх значення можуть присвоюватись одне одному. 11

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 1;
B = 1;
C = 1;
D = 2;
E = 1;
                    A <= Av  Av = 3;
                    B <= Bv  Bv = 3;
                    E <= 6   Ev = 6;
```

Результуючі значення сигналів будуть обчислені на основі значень змінних, оновлені на протязі виконання процесу. 12

```
process (C,D)
variable Av,Bv,Ev : integer := 0;
begin
   Av := 2;
   Bv := Av + C;
   Av := D + 1;
   Ev := Av * 2;
   A <= Av;
   B <= Bv;
   E <= Ev;
end process;
```

```
A = 3;
B = 3;
C = 1;
D = 2;
E = 6;
                    A <= 3   Av = 3;
                    B <= 3   Bv = 3;
                    E <= 6   Ev = 6;
```

Значення сигналів тепер відповідають очікуваним. 13

Рис.5.11.

Порівняння сигналів і змінних

Кожний сигнал має три властивості: *тип*, *значення* і *час*. Є досить тісний зв'язок між *значенням* і *часом*, оскільки для кожного сигналу відбувається його трасувальний запис в часі. Цей запис називається *історією сигналу* і дозволяє перевірити, які значення сигнал мав раніше, або в який момент відбулася його зміна.

З іншого боку, змінна має тільки дві властивості: *тип* і *значення*. Оскільки немає відповідності між її значенням і часом, змінна може мати тільки біжуче значення.

Як вже згадувалось раніше, якщо сигнали і змінні мають однаковий тип, їх значення можуть присвоюватись одне одному.

Властивості	Сигнали	Змінні
Декларація	Сигнали декларуються як порти в інтерфейсах і в декларативній частині архітектури. В кожному випадку вони можуть ініціалізуватись деяким початковим значенням. *Сигнали не можуть декларуватись всередині процесів.*	Оскільки змінні носять локальний характер, вони можуть декларуватись тільки в процесах і підпрограмах. *Змінні не можуть декларуватись поза процесами.*
Присвоєння	Сигнали отримають нові значення з присвоєнь тільки після припинення процесу. Тільки останнє присвоєння сигналу має силу. Автоприсвоєння (типу Sig1<=Sig1+1) не мають змісту і заборонені.	Змінні негайно отримують нові значення з присвоєнь. Кожне присвоєння змінної має силу, це означає, що змінна може мати декілька присвоєнь в одному процесі. Автоприсвоєння (типу Var1:=Var1+1) дозволені і часто використовуються.
Затримки	Затримки присвоєнь сигналів (оператор ***after***) дозволяються як інерційні, так і транспортні.	Затримки присвоєнь змінних (оператор ***after***) не дозволяються - змінним завжди присвоюється нове значення без будь-якої затримки.

Рис.5.12.

Керування послідовністю виконання операторів

Вступ

Процеси, в яких всі оператори виконуються послідовно зустрічаються дуже рідко. Оскільки процеси представляють реальні системи, в яких постійно змінюються умови середовища, і їх виконання звичайно відрізняються одне від одного.

Зміна умов середовища представляється у VHDL чотирма класами умовних операторів:

- умовне виконання операторів (оператори ***if*** ... ***then*** ...),
- умовні виконання з альтернативою (оператори ***if*** ... ***then*** ... ***else*** ... і ***if*** ... ***then*** ... ***elsif*** ...),
- оператори вибору (оператор ***case*** ...),
- цикли, що дозволяють повторювати виконання деяких операторів (оператори ***while*** ... ***do*** ... і ***for*** ... ***do*** ...).

```
ExProc: process (SensitivityList)
begin
  if Cond1
    then
      ...
      case Cond2 is
        when Val1 => ...
        when Val2 => ...
        when others => for l in 1 to 4 loop
                         ...
                       end loop;
      end case;
    else
      while Cond3 loop
        ...
      end loop;
      ...
    end if;
  ...
end process ExProc;
```

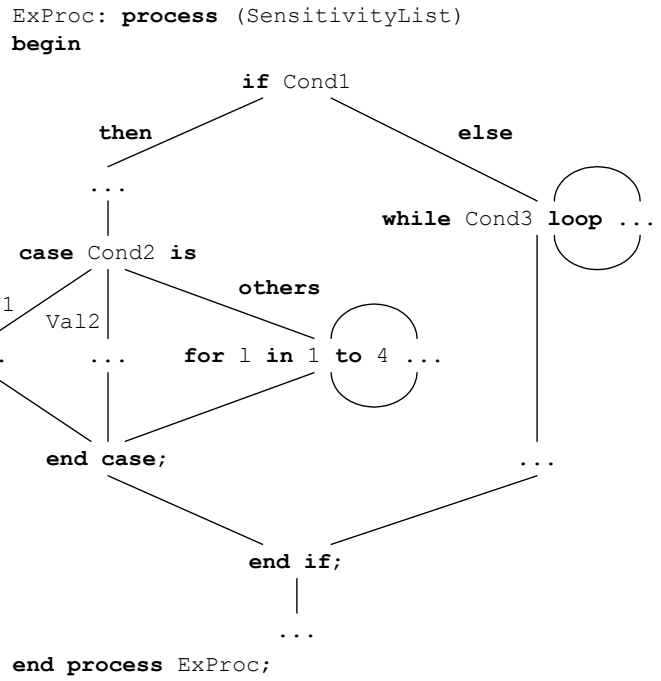


Рис.5.13.

Умовний оператор

Деякі оператори виконуються тільки коли виконуються вказані умови. Такі оператори називаються *умовними* і їх реалізація описується як

if *condition_met* ***then*** *execute_operations*

Під виконанням VHDL-операції розуміють наступну процедуру:

- умова "*condition_met*" обчислюється як булевий вираз (***true*** або ***false***); якщо він істинний, то активується підоператор "***then***";
- після підоператора "***then***" знаходиться список операцій (операторів), що мають виконуватись, кожний оператор закінчується крапкою з комою;
- для закінчення умовного оператора і для відокремлення його від наступних операторів відразу після останнього оператора вказується підоператор "***end if***".

```
D_FF: process (D,CLK)
begin
  if rising_edge(CLK)
    then
      Q <= D;
    end if;
end process D_FF;
```

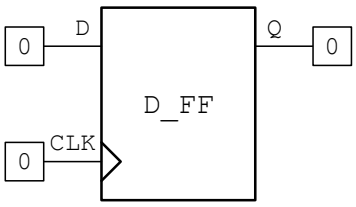


Рис.5.14.

Умовний оператор з альтернативою

Описаний вище умовний оператор досить простий і вимагає деякого розширення для підвищення гнучкості. Наприклад, необхідно описати такий випадок, як "якщо виконується умова, виконати деякі дії, якщо ні – виконати інші дії". Ця конструкція представляється оператором "***if*** ... ***then*** ... ***else*** ...".

Цей оператор розгалуження може бути розширений. Коли активний підоператор "***else***", він може містити інший умовний оператор, формуючи ієрархічну умову. Така ситуація підтримується у VHDL конструкцією "***if*** ... ***then*** ... ***elsif*** ...".

```
D_FF_RST: process (D,CLK,RST)
begin
  if RST = '1'
    then Q <= '0'
  elsif rising_edge(CLK)
    then
      Q <= D;
  end if;
end process D_FF_RST;
```

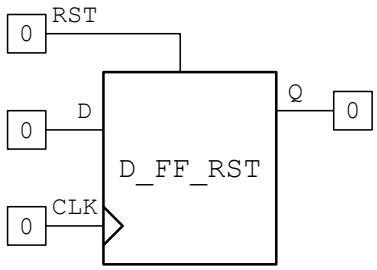


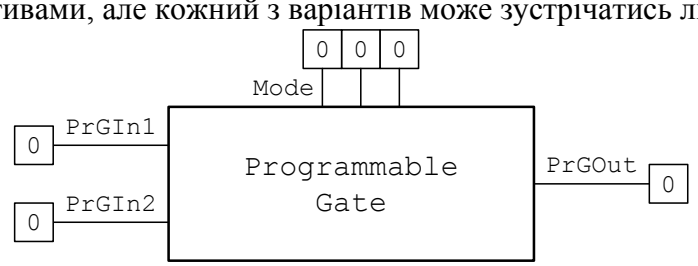
Рис.5.15.

Множинний вибір

Конструкція "***if*** ... ***then*** ... ***elsif*** ..." – це оператор *множинного вибору*, який дозволяє вибирати тільки одну гілку з декількох у відповідності з обчисленими умовами. Однак такі конструкції швидко стають нечитабельними, якщо кількість гілок (вкладених "***if***") перевищує три. В такій ситуації краще використовувати оператор множинного вибору "***case***".

Замість логічного виразу в операторі ***case*** описується вираз дискретного типу (довільний перечислимий або цілий тип), або одномірний масив символів. На практиці в якості цього виразу використовується ім'я об'єкту.

Кожна альтернативна група дій починається з підоператора "***when*** *choices* =>", де "*choices*" визначається значенням виразу. Ці значення можуть бути одиничними значеннями, діапазонами значень або альтернативами, але кожний з варіантів може зустрічатись лише один раз.




```
ProgrGate: process (Mode,PrGIn1,PrGIn2)
begin
  case Mode is
    when "000"   => PrGOut <= PrGIn1 and PrGIn2;
    when "001"   => PrGOut <= PrGIn1 or PrGIn2;
    when "010"   => PrGOut <= PrGIn1 nand PrGIn2;
    when "011"   => PrGOut <= PrGIn1 nor PrGIn2;
    when "100"   => PrGOut <= not PrGIn1;
    when "101"   => PrGOut <= not PrGIn2;
    when others  => PrGOut <= '0';
  end case;
end process ProgrGate;
```

Рис.5.16.

Умовний цикл

Оператор *умовного циклу* починається з логічної умови і працює аналогічно умовному оператору "***if*** ... ***then*** ...", але в ньому відбувається перехід на початок циклу після останнього оператора. Цикл повторюється до тих пір, поки виконується умова на початку циклу. Спочатку перевіряється умова, і, якщо вона істинна, виконуються оператори всередині циклу. Якщо ж умова хибна, цикл вважається *закінченим* і керування передається першому оператору після циклу.

Як правило, такі цикли використовуються для повторення виконання набору операторів до тих пір, поки сигнал або змінна відповідає вибраному критерію, наприклад певному значенню.

Наступний приклад – це лічильник, який підраховує додатні фронти сигналу *CLK* поки сигнал *Level* рівний '1'. При цьому стабільність сигналу *Level* не аналізується. На початку циклу

тільки перевіряється рівність '1' сигналу *Level* при визначенні додатнього фронту сигнала *CLK*. І немає значення, чи змінювалось значення сигналу *Level* після останньої перевірки.

```
process
variable Count : integer := 0;
begin
  wait until CLK = '1';
  while Level = '1' loop
    Count := Count +1;
    wait until CLK = '0';
  end loop;
end process;
```

Рис.5.17.

Цикл з лічильником

Деколи може бути потрібно повторювати операції визначену кількість разів. Для цього зручно використовувати цикл "***for*** ... ***loop*** ...".

Цикл ***for*** не містить явної логічної умови. Замість цього задається дискретний лічильник із діапазоном значень і цикл повторюється доти, доки цей лічильник не вийде за межі діапазона. Після кожної ітерації циклу, лічильнику присвоюється наступне значення із заданого діапазону. Лічильник, який не потрібно декларувати (його специфікація в заголовку циклу прирівнюється до декларування) всередині циклу вважається константою і може використовуватись в присвоєннях, індексах виразів, але не може бути змінений. Більше того, лічильник існує тільки всередині циклу, в якому він задекларований.

Діапазон лічильника не потрібно задавати в класичній формі вигляду ***from*** ... ***to*** .... Він може бути також заданий як підтип або перечислимий тип. В такому випадку тільки (під)тип задається в якості діапазону лічильника.

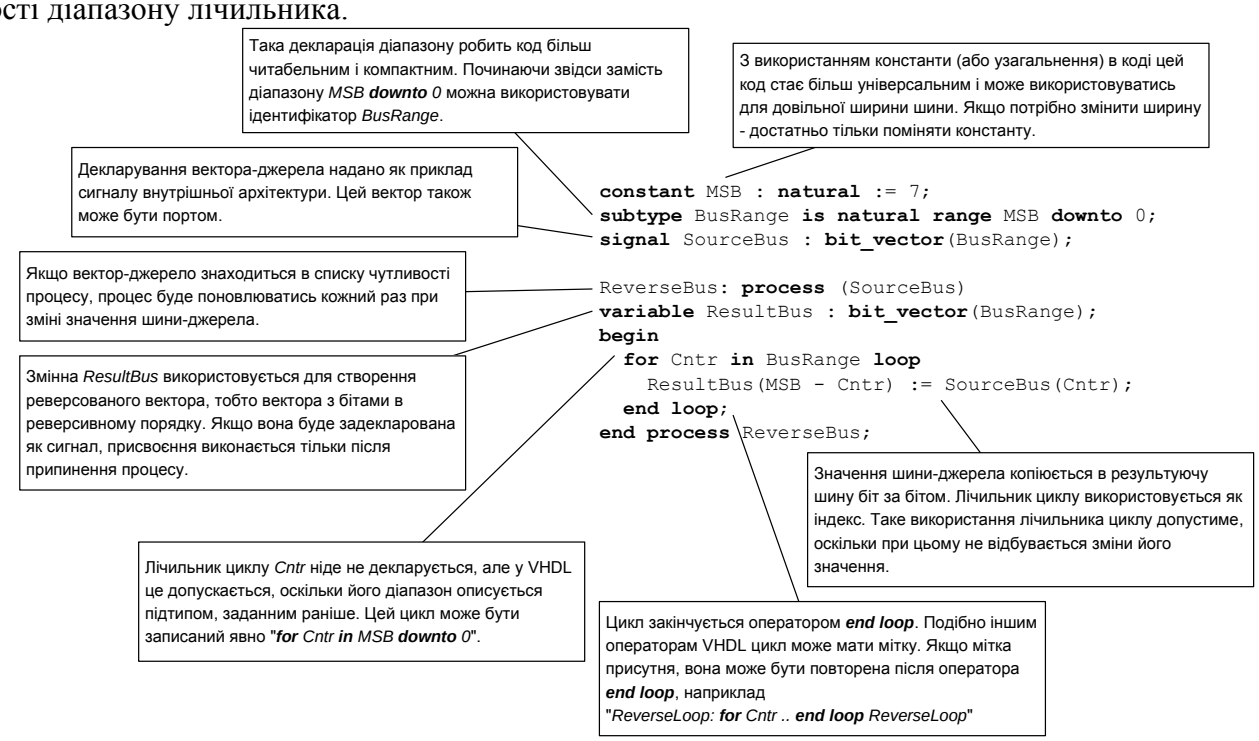


Рис.5.18.

Переривання циклів

Цикли ***while*** і ***for*** повторюють виконання всіх своїх операторів до тих пір, поки виконується умова циклу. Однак деколи буває необхідно пропустити деякі оператори циклу в біжучій ітерації і перейти безпосередньо до наступної ітерації цього циклу. Така ситуація може виникнути, наприклад, при підрахунку бітів, рівних '1' у векторі бітів (наступний приклад): якщо біт рівний '1' – лічильник збільшується на 1, якщо ж '0' – цикл повинен перейти до наступної ітерації. Такий тип операції забезпечується за допомогою оператора ***next***.

```
signal DataBus : bit_vector(3 downto 0);
signal Ones : integer;

CountOnes: process (DataBus)
variable NumOfOnes : integer := 0;
begin
 for Cntr in 3 downto 0 loop
   next when DataBus(Cntr) = '0';
   NumOfOnes := NumOfOnes + 1;
 end loop;
 Ones <= NumOfOnes;
end process CountOnes;
```

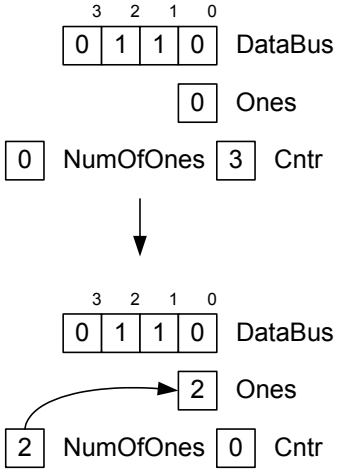


Рис.5.19.

Крім оператора ***next***, який виконує операцію переходу на наступну ітерацію, бувають ситуації, яки вимагають припинення циклу, навіть якщо його умова істинна. В таких випадках використовується оператор ***exit***.

Для переходу на наступну ітерацію і виходу з циклу може бути задана умова виконання. Оскільки такі операції використовуються дуже часто, у VHDL існує спрощений засіб для задання умов операторів ***next*** і ***exit***: ця умова задається в підоператорі ***when*** після вказаного оператора.

Оператор null

Оператор ***null*** не виконує ніяких дій і просто передає керування на наступний оператор. Він може використовуватись для того, щоби вказати, що при певних умовах ніяких дій виконувати не потрібно.

```
case OPCODE is
  when "001" => TmpData := RegA and RegB;
  when "010" => TmpData := RegA or RegB;
  when "100" => TmpData := not RegA;
  when others => null;
end case;
```

Рис.5.20.

6. Множинні процеси у VHDL-архітектурі. Паралельність. Оператори присвоєння сигналів як спрощені процеси. Драйвери та атрибути сигналів. Багатозначна логіка.

Паралельність

Паралельна природа систем

Хоча зручно описувати реальні процеси як послідовні, це дуже спрощенний підхід. Світ взагалі не є послідовним. Всі реальні події відбуваються одночасно, тобто *паралельно*.

Деякі системи можуть складатись з паралельних підсистем. На відповідному рівні деталізації поведінку окремої підсистеми можна описати як послідовність дій або *процесів*.

Оскільки VHDL – це ієрархічна мова, вона дозволяє описувати системи такими, якими вони є реально – як набір паралельно працюючих підсистем. Кожна з таких взаємнопов'язаних систем описується як окремий процес. Рівень деталізації залежить від потреби – один процес може описувати поведінку складного компонента (наприклад, процесора), а інший – логічного вентиля. Отже, поведінковий опис системи – це набір паралельних послідовних процесів.

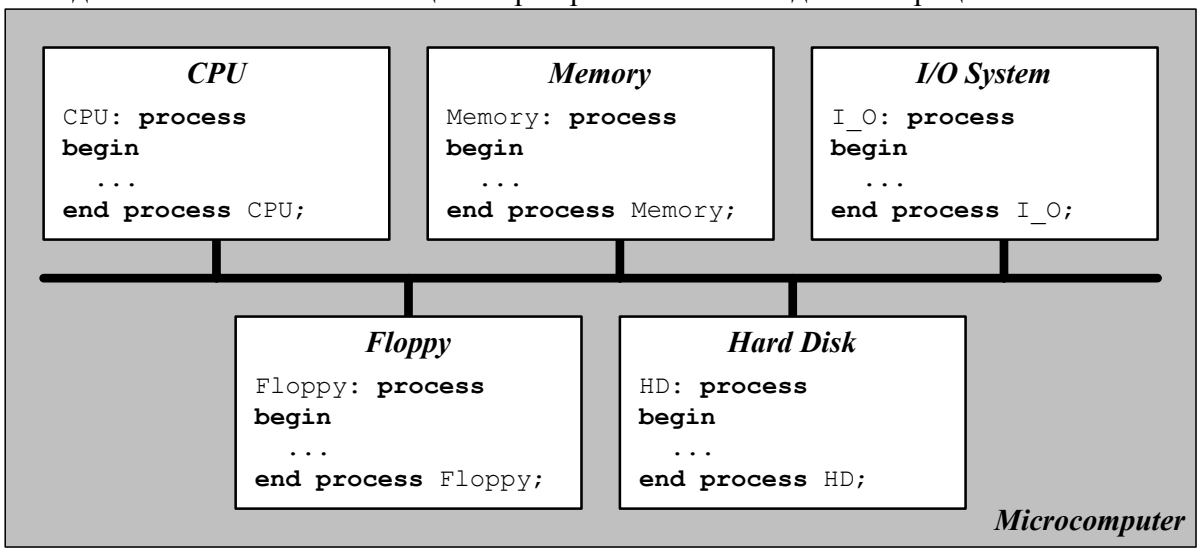


Рис.6.1.

Структура поведінкових архітектур

Подібно до інших складних конструкцій VHDL, *архітектура* складається з шаблона і тіла. Шаблон описує, що це є архітектура, задає її ім'я і границі. Він також містить декларації внутрішніх об'єктів архітектури. Тіло архітектури містить всі процеси архітектури.

Унікальною властивістю заголовку архітектури є асоціація її із інтерфейсом, оскільки кожна архітектура має відноситись до певного інтерфейсу.

Тіло архітектури має паралельну структуру, яка деколи є досить важкою для розуміння, оскільки всі оператори (*процеси* і їх вміст) записуються послідовно. Тим не менше, всі процеси в будь-якій архітектурі виконуються паралельно.

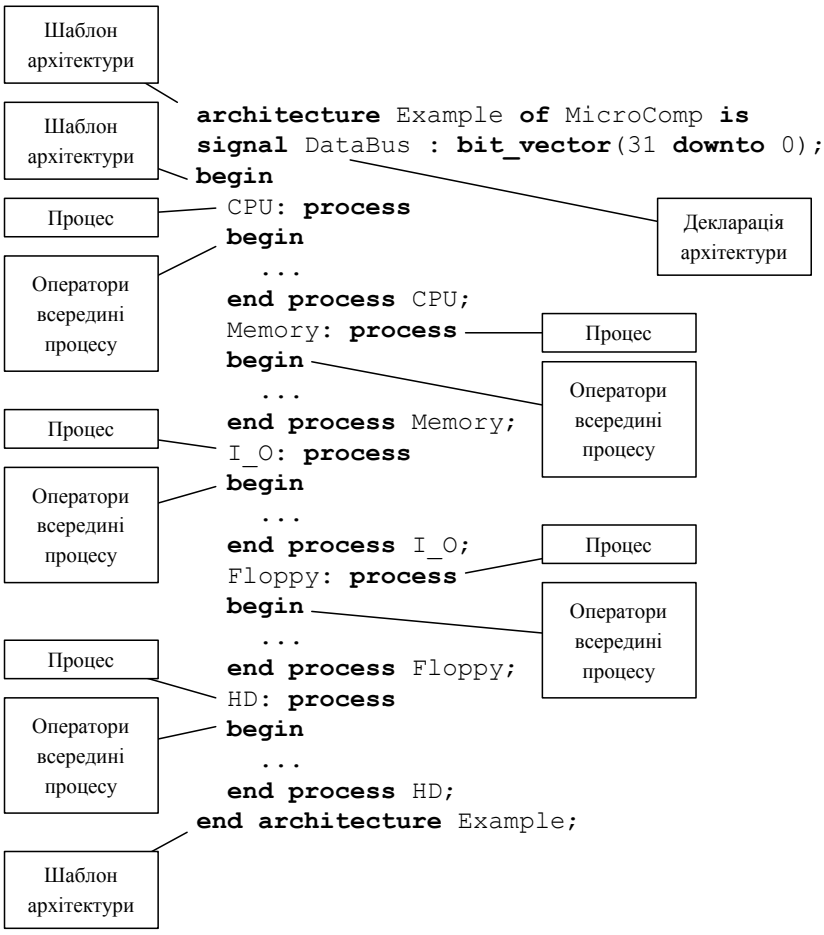


Рис.6.2.

Шаблон архітектури звичайно подібний на шаблон інтерфейса за виключенням того, що він має додаткову частину "ім'я інтерфейсу", яка відносить цю архітектуру до конкретного інтерфейсу.

Декларація архітектури. Будь-які локальні (внутрішні) об'єкти архітектури описуються між заголовком архітектури і ключовим словом ***begin***. Ці об'єкти невидимі поза архітектурою.

Процеси. Всі процеси в архітектурі є паралельними між собою, тобто поновлені процеси виконуються паралельно. В той же час всі оператори всередині процесів виконуються послідовно.

Оператори всередині процесів. Хоча в загальному процеси виконуються паралельно, для операторів кожного процесу забезпечується послідовний порядок виконання.

Виконання архітектури

Паралельне виконання процесів і послідовне виконання операторів всередині процесів є дещо незвичним. Змішування послідовних та паралельних операцій є однією з найскладніших концепцій VHDL.

Як вказувалось раніше, припинений процес поновлюється (активується), якщо змінюється значення якогось з сигналів з його списку чутливості. Це правило дійсне і для множинних процесів в архітектурі, і, коли змінюється значення сигналу, поновлюються *всі* процеси, в списки чутливості яких входить цей сигнал. Оператори всередині поновлених процесів виконуються послідовно. Але при цьому вони виконуються незалежно від інших процесів.

Ідею паралельності можна краще зрозуміти, якщо записати процеси не один під одним, а рядом.

```
architecture SomeArch of SomeEnt is
begin
  P1: process (A,B)
  begin
    somestatement;
    somestatement;
    somestatement;
    somestatement;
  end process P1;
  P2: process (A,C)
  begin
    somestatement;
    somestatement;
    somestatement;
  end process P2;
  P3: process (B)
  begin
    somestatement;
    somestatement;
  end process P3;
end architecture SomeArch;
```

→

```
architecture SomeArch of SomeEnt is
begin
P1: process (A,B)
begin
  somestatement;
  somestatement;
  somestatement;
  somestatement;
end process P1;

P2: process (A,C)
begin
  somestatement;
  somestatement;
  somestatement;
end process P2;

P3: process (B)
begin
  somestatement;
  somestatement;
end process P3;

end architecture SomeArch;
```

Всі наведені процеси, вміст яких показаний символично, припинені і очікують події, яка би їх активувала. Цією подією є зміна сигналів, вказаних в списках чутливості.
Зміна значення сигналу *A* активує процеси *P1* і *P2*. Оператори всередині цих процесів будуть виконані послідовно і незалежно один від одного, після чого ці процеси будуть припинені. Процеси *P1* і *P2* виконуються паралельно.
Зміна значення сигналу *B* активує процеси *P1* і *P3*. Оператори всередині цих процесів будуть виконані послідовно і незалежно один від одного, після чого ці процеси будуть припинені. Процеси *P1* і *P3* виконуються паралельно.
Зміна значення сигналу *C* активує тільки процес *P2*. Оператори всередині процесу будуть виконані послідовно, після чого процес буде припинений. В цьому випадку паралельність виконання процесів відсутня.

Рис.6.3.

Передача інформації між процесами

Оскільки процеси не розрізняють сигнали, згенеровані ззовні (що надходять від середовища) і згенеровані всередині (задекларовані всередині архітектури), сигнали, які активують процеси, можуть бути згенеровані іншими процесами в тій самій архітектурі. Як тільки сигнал із списку чутливості процесу змінює значення, процес активується. Це відбувається незалежно від того, був сигнал змінений системним середовищем чи іншим процесом.

Інформація між процесами може передаватись тільки за допомогою сигналів. Оскільки змінні є локальними об'єктами процесів, вони для цього непридатні.

```
architecture SomeArch of SomeEnt is
begin
  P1: process (A,B,E)
  begin
    somestatement;
    somestatement;
    D <= someexpression;
  end process P1;
  P2: process (A,C)
  begin
    somestatement;
    somestatement;
    somestatement;
    E <= someexpression;
  end process P2;
  P3: process (B,D)
  begin
    somestatement;
    somestatement;
    somestatement;
  end process P3;
end architecture SomeArch;
```

Зміна *A*

1. Зміна значення сигналу *A* активує процеси *P1* і *P2* . Послідовність операторів всередині цих процесів буде виконана, після чого процеси припиняються. Останніми операторами цих процесів є присвоєння сигналів *D* і *E* , які містяться в списках чутливості процесів *P3* і *P1* відповідно.
2. Оскільки сигнали *D* і *E* змінили значення, процеси *P3* і *P1* активуються. Однак якщо сигналу буде присвоєне теж саме значення, що він мав перед тим, він не активує процес, чутливий до цього сигналу.
3. Сигналу *D* присвоюється деяке значення. Якщо це значення відрізняється від попереднього, процес *P3* знову поновлюється. Якщо ж ні - процес *P3* не буде активований.

Зміна *B*

1. Зміна значення сигналу *B* активує процеси *P1* і *P3* . Послідовність операторів всередині цих процесів буде виконана, після чого процеси припиняються. Останнім оператором процесу *P1* сигналу *D* , який є в списку чутливості процесу *P3*, присвоюється деяке значення.
2. Сигналу *D* присвоюється деяке значення. Якщо це значення відрізняється від попереднього, процес *P3* знову поновлюється. Якщо ж ні - процес *P3* не буде активований.

Зміна *C*

1. Зміна значення сигналу *C* активує тільки процес *P2*. Послідовність операторів всередині цього процесу буде виконана, після чого процес припиняється. Останнім оператором цього процесу є присвоєння сигналу *E*, який міститься в списку чутливості процесу *P1*.
2. Сигналу *E* присвоюється деяке значення. Якщо це значення відрізняється від попереднього, процес *P1* буде поновлений. Якщо ж ні - процес *P1* не активується.
3. Якщо процес *P1* був поновлений, в останньому його операторі виконується присвоєння сигналу *D*, що міститься в списку чутливості процесу *P3*. Якщо це значення відрізняється від попереднього, процес *P3* поновлюється. Якщо ж ні - процес *P3* не буде активований.

Рис.6.4.

Присвоєння сигналів як спрощені процеси

Прості процеси – присвоєння сигналів

Деколи буває необхідно використовувати прості логічні вентилі в якості окремих модулів архітектури, наприклад:

```
OutAND <= InAND1 and InAND2;
```

Опис такої поведінки за допомогою процесу вимагає трьох додаткових операторів (заголовок процесу, оператор ***begin*** і оператор ***end process***), що є зайвим для наведеного вентиля AND. VHDL дозволяє спростити такий однооператорний процес до одиничного рядка оператора присвоєння, який називається *паралельним присвоєнням сигналу*.

Оператор паралельного присвоєння сигналу може з'являтися всередні архітектури паралельно з іншими процесами і виконуватись паралельно з іншими присвоєннями, так само, як і процеси. Однак, якщо вони представляються паралельними присвоєннями сигналів, то і виконуються вони паралельно.

```
architecture SomeArch of SomeEnt is
signal Int1,Int2 : bit;
begin
  Gate1: process (A,C)
  begin
    Int1 <= A or C;
  end process Gate1;
  Gate2: process (B,D)
  begin
    Int2 <= B or D;
  end process Gate2;
  Gate3: process (Int1,Int2)
  begin
    Out <= Int1 and Int2;
  end process Gate3;
end architecture SomeArch;
```

→

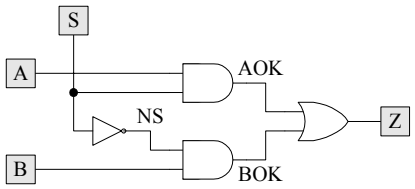
```
architecture SomeArch of SomeEnt is
signal Int1,Int2 : bit;
begin
  Int1 <= A or C;
  Int2 <= B or D;
  Out <= Int1 and Int2;
end architecture SomeArch;
```

Рис.6.5.

Активація паралельних присвоєнь

Процеси містять списки чутливості або оператор ***wait*** для визначення умови їх активації або поновлення. Але паралельні оператори присвоєння не містять ніяких операторів ***wait*** або списків чутливості.

Як показано в наступному прикладі, списки чутливості усіх процесів містять сигнали, які з'являються пізніше у виразах, значення яких присвоюються вихідним сигналам процесів. Отже, паралельні присвоєння сигналів є чутливими до зміни будь-якого сигналу, що знаходиться справа від символу присвоєння. Фактично, саме так VHDL-симулятори і працюють: зміна будь-якого з сигналів в правій частині паралельного присвоєння сигналу активує виконання цього присвоєння. Таке присвоєння може бути затримане за допомогою оператора ***after*** як інерційною, так і транспортною затримкою.




```
architecture Gates of Mux2to1 is
signal AOK,BOK,NS : bit;
begin
  AOK <= A and S after 1 ns;
  BOK <= B and NS after 1 ns;
  NS <= not S after 1 ns;
  Z <= AOK or BOK after 1 ns;
end architecture Gates;
```

Зміна сигналу *A* активує присвоєння *AOK*, зміна якого активує присвоєння *Z*.

Зміна сигналу *B* активує присвоєння *BOK*, зміна якого активує присвоєння *Z*.

Зміна сигналу *S* активує паралельно присвоєння *AOK* і *NS*. *AOK* активує *Z*. *NS* активує *BOK*, яке реактивує *Z*.

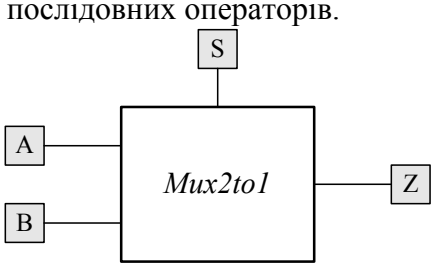
Рис.6.6.

Умовне присвоєння сигналу

Деколи необхідно виконати присвоєння сигналу, якщо умова істинна. В зв'язку з тим, що умовний оператор "***if*** ... ***then*** ..." є послідовним оператором і може використовуватись лише в процесах, необхідне *умовне присвоєння сигналу*, яке можна було би використовувати всередині архітектури. Це функціональний еквівалент умовного оператора, але він записується інакше для того, щоби їх можна було б розрізнити між собою.

Синтаксис умовного оператора присвоєння відповідає порядку, за яким він записується: "присвоїти визначене значення сигналу, якщо виконується умова, а якщо ні (***else***) – присвоїти інше значення". Підоператор ***else*** може бути відсутнім.

Важлива різниця між умовним присвоєнням сигналу і умовним оператором полягає в тому, що використання першого обмежено тільки присвоєннями сигналів, в той час як останній може використовуватись для довільних послідовних операторів.




```
architecture Conditional of Mux2to1 is
begin
  Z <= A when S = '1' else
       B;
end architecture Conditional;
```

→

```
architecture CondinProc of Mux2to1 is
begin
  Mux: process (A,B,S)
  begin
    if S = '1'
      then Z <= A;
      else Z <= B;
    end if;
  end process Mux;
end architecture Conditional;
```

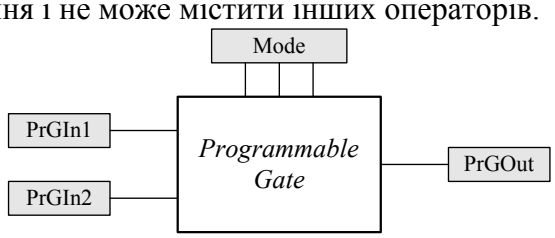
Рис.6.7.

Присвоєння сигналу із вибором

Для умовного присвоєння сигналу одного з декількох значень використовується конструкція *присвоєння сигналу із вибором*. Це паралельна конструкція, яка не може використовуватись всередині процесів, де замість неї використовується оператор ***case***, і навпаки, оператор ***case*** не

може використовуватись поза процесами, замість нього в такому випадку використовується присвоєння сигналу із вибором.

Слід зауважити, що присвоєння сигналу із вибором, на відміну від оператора ***case***, обмежено тільки операторами присвоєння і не може містити інших операторів.




```
architecture SelBased of ProgrammableGate is
begin
  with Mode select
    PrGOut <=
              PrGIn1 and PrGIn2 when "000",
              PrGIn1 or PrGIn2 when "001",
              PrGIn1 nand PrGIn2 when "010",
              PrGIn1 nor PrGIn2 when "011",
               not PrGIn1 when "100",
               not PrGIn2 when "101",
              '0' when others;
end architecture SelBased;
```

→

```
architecture CaseBased of ProgrammableGate is
begin
  ProgrGate: process (Mode,PrGIn1,PrGIn2)
  begin
    case Mode is
      when "000" => PrGOut <= PrGIn1 and PrGIn2;
      when "001" => PrGOut <= PrGIn1 or PrGIn2;
      when "010" => PrGOut <= PrGIn1 nand PrGIn2;
      when "011" => PrGOut <= PrGIn1 nor PrGIn2;
      when "100" => PrGOut <= not PrGIn1;
      when "101" => PrGOut <= not PrGIn2;
      when others => PrGOut <= '0';
    end case;
  end process ProgrGate;
end architecture SelBased;
```

Рис.6.8.

Драйвера і атрибути сигналів

Концепція драйвера

Присвоєння значення сигналу відбувається після припинення процесу. Більше того, якщо сигналу присвоюється декілька значень на протязі виконання процесу, тільки останнє присвоєння має силу. При цьому необхідно зберігати інформацію про події з сигналами.

Ця функція виконується *драйвером*. VHDL-компілятор створює драйвер для кожного сигналу, якому присвоюється значення всередині процесу. Правило дуже просте: неважливо, скільки значень присвоювалось сигналу – існує тільки один драйвер сигналу в процесі. Усі операції виконуються із драйвером, значення з якого копіюється в сигнал тільки після припинення процесу.

```
ExProc: process (SigA,SigB)
begin
  SigC <= SigA;
  ...
  SigC <= SigB + 1;
end process ExProc;
```

SigA = 0 SigB = 1

SigC = 0

ExProc_SigC = 0

```
ExProc: process (SigA,SigB)
begin
  SigC <= SigA;
  ...
  SigC <= SigB + 1;
end process ExProc;
```

SigA = 1 SigB = 1

SigC = 0

ExProc_SigC = 0

Будь-яка зміна сигналу з списку чутливості активує асоційований процес і драйвера всіх сигналів, яким присвоюється нове значення - по одному драйверу на сигнал.

```
ExProc: process (SigA,SigB)
begin
  SigC <= SigA;
  ...
  SigC <= SigB + 1;
end process ExProc;
```

SigA = 1 SigB = 1

SigC = 0

ExProc_SigC = 1

Присвоєння виконується не безпосередньо сигналу, а його драйверу. Значення драйвера буде потім скопійоване в сигнал після припинення процесу.

```
ExProc: process (SigA,SigB)
begin
  SigC <= SigA;
  ...
  SigC <= SigB + 1;
end process ExProc;
```

SigA = 1 SigB = 1

SigC = 0

ExProc_SigC = 2

Наступне присвоєння сигналу виконується над драйвером, відміняючи попередньо збережену інформацію. Ось чому тільки найостанніше присвоєння сигналу має силу.

```
ExProc: process (SigA,SigB)
begin
  SigC <= SigA;
  ...
  SigC <= SigB + 1;
end process ExProc;
```

SigA = 1 SigB = 1

SigC = 2

ExProc_SigC = 2

Коли процес припиняється, дані з драйверів копіюються у відповідні сигнали. Ця операція ще називається "фізичним присвоєнням сигналу".

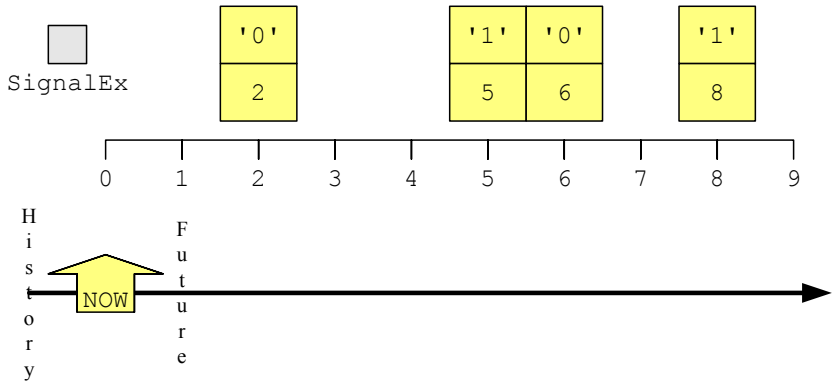
Рис.6.9.

Історія і майбутнє сигналів

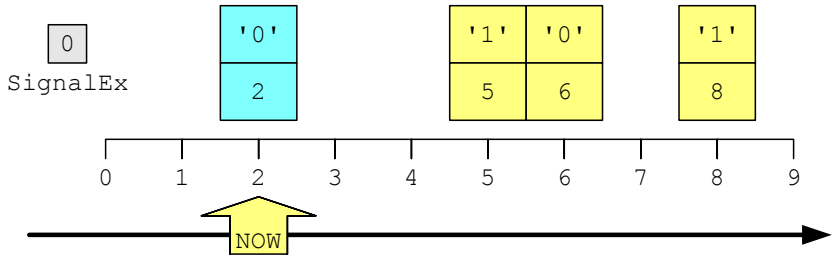
Завдяки драйверам сигнали можуть мати минулі, теперішні та майбутні значення. Драйвера містять біжучі значення сигналів. Крім того, драйвера можуть також описувати *проектовану вихідну часову діаграму* сигналу. Кожному драйверу може відповідати така часова діаграма, яка складається з послідовності однієї або більше *транзакцій*. Транзакція складається із значення сигналу і часу. Час описує момент, коли драйверу буде присвоєне нове значення, що описується транзакцією.

Часова діаграма може бути описана явно як послідовність значень з асоційованими затримками відносно однієї точки часу. Часові діаграми можна розглядати як проектоване *майбутнє сигналів*. Оскільки симулятори зберігають транзакції для кожного сигналу, вони створюють в результаті *історію сигналів*.

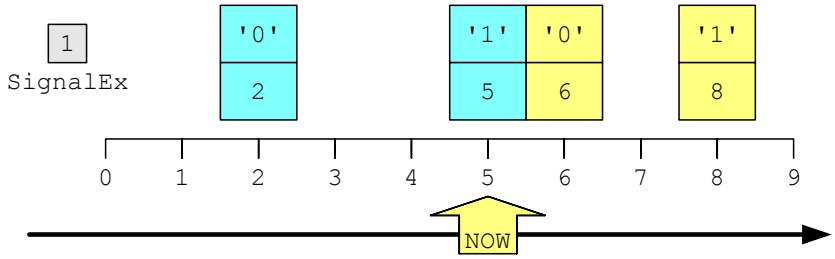
```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```



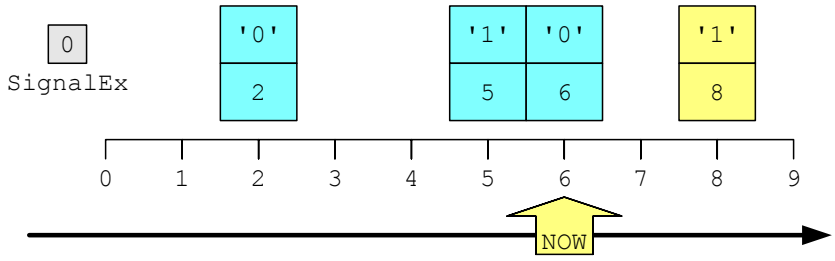

```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```



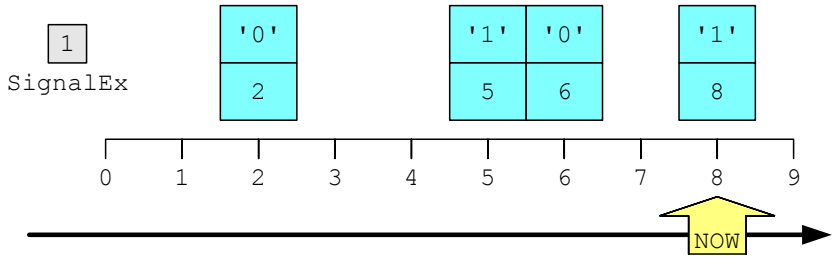

```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```




```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```




```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```




```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```

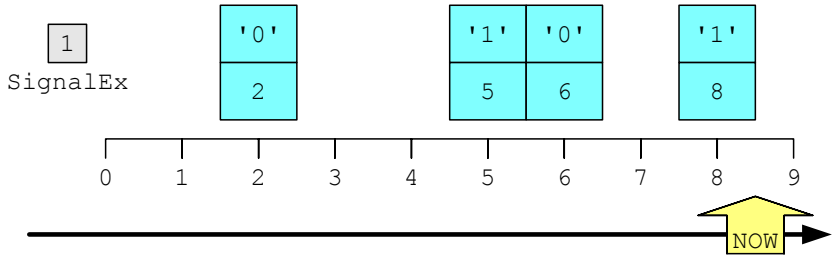


Рис.6.10.

Часозалежні атрибути сигналів

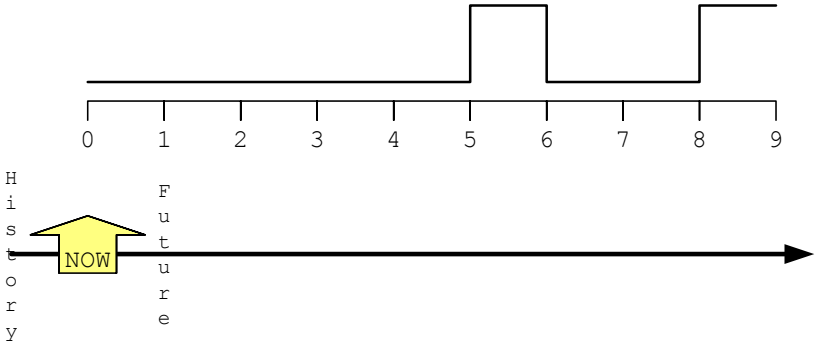
Опис майбутнього сигналів важливий для створення *тестових стендів*, на яких VHDL-описи тестуються у VHDL-середовищі, а стимулятори представляються часовими діаграмами. Тестові стенди будуть розглянуті пізніше.

Історія сигналу потрібна з декількох причин. Наприклад, маючи тільки біжуче значення сигналу, не можна перевірити, чи змінювався сигнал раніше, або перевірити його на зростаючий чи спадаючий фронт. Історія сигналу представляється *атрибутами сигналу*. Атрибути – це інформація про сигнал, яка автоматично оновлюється на основі історії сигналу. Ця інформація включає попереднє значення сигналу, інтервал часу, який пройшов з моменту останньої зміни значення сигналу, і т.п. Для отримання біжучого значення атрибуту потрібно задати ім'я сигналу, за ним слідує апостроф і ім'я атрибута.

До часозалежних атрибутів відносяться наступні атрибути сигналів:

- S'**transaction** – має тип **Boolean** із значенням **true**, коли відбувається переприсвоєння сигналу S;
- S'**event** – має тип **Boolean** із значенням **true**, коли відбулась зміна значення сигналу S;
- S'**last_event** – має тип **time** і визначає час, що пройшов від останньої події із сигналом S;
- S'**last_value** – попереднє значення, яке сигнал мав безпосередньо перед останньою зміною сигналу S;
- S'**stable(T)** – має тип **Boolean** із значенням **true**, якщо сигнал S стабільний на протязі останніх T одиниць часу.

```
SignalEx <= '0' after 2 s,
            '1' after 5 s,
            '0' after 6 s,
            '1' after 8 s;
```



	Час								
	0	1	2	3	4	5	6	7	8
SignalEx	0	0	0	0	0	1	1	0	1
SignalEx'Transaction	false	false	true	false	false	true	true	false	true
SignalEx'Event	false	false	false	false	false	true	true	false	true
SignalEx'Last_event	max time	max time	max time	max time	max time	0	0	1	0
SignalEx'Last_value	0	0	0	0	0	0	1	1	0

Рис.6.11.

Інші атрибути

Атрибути, що мають відношення до історії сигналів – не єдині доступні атрибути. Взагалі атрибути дуже часто використовуються у VHDL і мова визначає 36 різних атрибутів для різних класів об'єктів: скалярних типів, дискретних типів, масивів, сигналів та іменованих інтерфейсів. Повний список визначених атрибутів можна знайти в Reference Guide.

Крім вже визначених атрибутів можна визначати власні атрибути, розширюючи VHDL-набір майже необмежено.

Атрибути мають багато застосувань – від визначення фронту (на основі атрибуту **'event**) до незалежних від розміру описів. Деякі з найбільш типових прикладів застосування атрибутів представлені в наступній таблиці.

Визначення фронту	Для того, щоби перевірити сигнал (як правило, *CLK*) на зростаючий або спадаючий фронт, необхідно знати, що він тільки що змінився і його значення дорівнює '1' або '0' відповідно. Перша умова може бути перевірена через атрибут **'event**, а повна умова виглядає так: спочатку зростаючий фронт, потім спадаючий фронт.	`if CLK'event`
		`  and CLK = '1'`
		`  and CLK'last_value = '0' then ...`
		
		`if CLK'event`
		`  and CLK = '0'`
		`  and CLK'last_value = '1' then ...`

Універсальний цикл	Для усунення жорсткокодованих границь циклу можна використовувати атрибути діапазону. Немає значення, який діапазон описаний для одновимірного масиву DataArr (включаючи вектори бітів), цикл буде повторюватись рівно стільки разів, скільки елементів містить масив. Такі універсальні діапазони полегшують підтримку коду. При цьому цикл може повторюватись як в прямому, так і в зворотньому порядку.	`for i in DataArr'range loop ...` `for i in DataArr'reverse_range loop ...`
Індекси найстаршого або наймолодшого бітів	Деколи буває необхідно отримати індекс найстаршого або наймолодшого бітів для ***bit_vector***. Замість того, щоби писати їх жорстко-кодовані номери, які важко підтримувати у випадку змін, вони можуть бути легко отримані через атрибути **'left** і **'right** відповідно.	`ArrayType'left` `ArrayType'right`

Час встановлення	Час встановлення може бути порушений, якщо інтервал між останньою подією із сигналом і активним фронтом тактового імпульсу менше часу "встановлення". В прикладі булева змінна SetupViolated стає ***true***, якщо відбувається порушення часу встановлення сигналу SomeSignal. Це один з способів автоматичного визначення помилок у VHDL.	`if CLK'event`
		`  and CLK = '1'`
		`  and CLK'last_value = '0'`
		`then SetupViolated :=`
		`  SomeSignal'last_event < SetupTime;`
		`end if;`

Рис.6.12.

Багатозначна логіка

Багатодрайверні сигнали

Сигнали з декількома джерелами зустрічаються в багатьох реалізаціях. Наприклад, комп'ютерна шина даних може отримувати дані від процесора, пам'яті, дисків та пристроїв вводу-виводу. Кожний з цих пристроїв керує шиною і кожна сигнальна лінія шини може мати декілька драйверів. Оскільки VHDL є мовою опису цифрових систем, такі багатодрайверні сигнали природньо підтримуються у VHDL.

Важко завчасно визначити напевно, чи буде багатоджерельний сигнал керуватись завжди тільки одним джерелом в один момент часу. В деяких системах це завжди буде так, в інших – може бути навіть бажано змішувати сигнали від різних джерел, емулюючи, наприклад, операції "монтажне І" або "монтажне АБО". Взагалі, багатоджерельні сигнали вимагають встановлення методу визначення результуючого значення у випадку, якщо декілька джерел паралельно видають значення на одну і ту ж саму лінію.

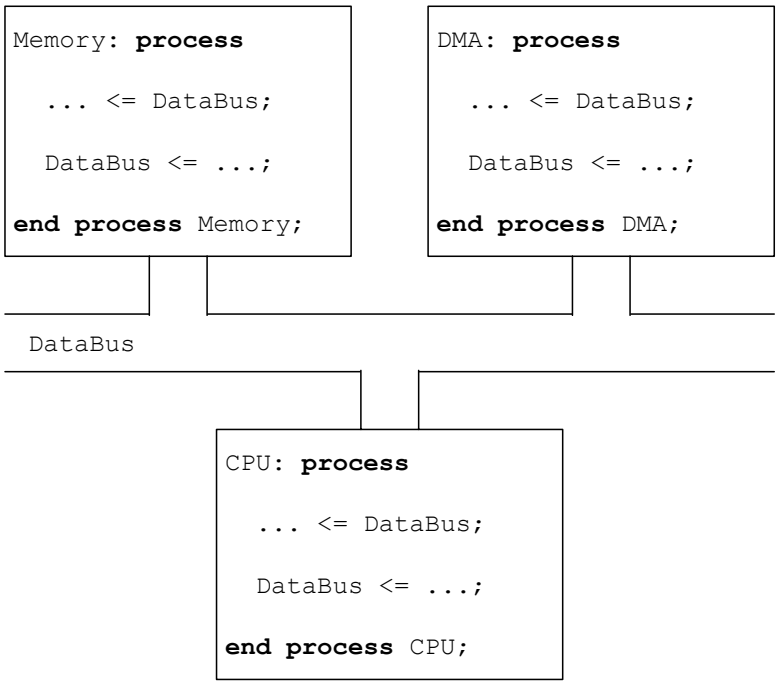


Рис.6.13.

VHDL-симулятор не може завчасно "знати" про те, чи буде багатодрайверний сигнал активований двома або більше джерелами одночасно. Через це симулятор має бути готовим до змішування значень сигналів. Таке "змішування" сигналів називається у VHDL *розділенням*. Правила для змішування значень сигналів визначаються як таблиця, яка називається *функцією розділення*. Таблиця будується з використанням усіх можливих значень сигналів по рядках і стовбцях, і кожна її комірка містить інформацію про значення, яке буде згенероване, якщо два відповідних значення будуть змішані.

Недоліки двозначної логіки

Розглянемо тип сигналу ***bit***. Що станеться, якщо змішати '0' і '1' (яке значення буде отримане з '0' і '1')? Проблема із типом ***bit*** полягає в тому, що неможливо визначити розділені сигнали, використовуючи тільки два значення, і немає відповіді на вказане питання.

Цей факт має важливі наслідки: використовуючи тільки типи ***bit*** і ***bit_vector***, неможливо описати у VHDL мікропроцесор. Такі нерозділимі типи не можуть використовуватись для багатодрайверних сигналів, які потрібні для опису шин даних (які обов'язково присутні у мікропроцесорі). Для вирішення цієї проблеми необхідно використовувати деякі інші логічні типи даних із більше ніж двома значеннями і функцією розділення, визначеною для всіх комбінацій значень сигналів.

```
signal A,B,Z : bit;
  ...
  Z <= A;
  Z <= B;
```

	0	**1**
0	0	1
1	0	1

Якщо при змішуванні 0 і 1 результатом є 0, це означає, що 0 подавляє всі інші значення як функція І. Однак неможна припускати, що для усіх цифрових систем буде домінувати роздільна функція І. Чому не АБО, наприклад?

Якщо при змішуванні 0 і 1 результатом є 1, це означає, що 1 подавляє всі інші значення як функція АБО. Однак неможна припускати, що для усіх цифрових систем буде домінувати роздільна функція АБО. Чому не І, наприклад?

Рис.6.14.

Багатозначна логіка

Одних тільки '0' і '1' недостатньо для розділення багатоджерельних сигналів. Навіть деякі одноджерельні сигнали часто вимагають більше двох значень для представлення реальних цифрових об'єктів:

- деколи неважливо, яке значення має сигнал – в цьому випадку використовується "довільне" значення,
- буфери з трьома станами відключають драйвери від ліній сигналів в стані високого імпеданса, це не є '0' або '1',
- деколи система може мати "неприсвоєні" або "невідомі" значення, які дещо відрізняються від "довільних".

Ці та декілька інших найбільш часто використаних значень описуються типом ***std_ulogic***, визначеним в пакеті *Std_Logic_1164*. Цей пакет містить також визначення векторного типу, що базується на типі ***std_ulogic*** – ***std_ulogic_vector***. Для обох типів визначений також набір базових логічних операцій.

Символ '***u***' в імені ***ulogic*** вказує на те, що це нерозділимі (*unresolved*) типи. Такі типи не можуть використовуватись із багатодрайверними сигналами.

Декларування	Тип ***std_ulogic*** задекларований в пакеті *std_logic_1164* і містить 9 перечислимих значень. Тип ***std_ulogic_vector*** - це необмежений вектор елементів ***std_ulogic***.	**TYPE** std_ulogic **is** { 'U', -- неініціалізовано 'X', -- сильні 0 або 1 '0', -- сильний 0 '1', -- сильна 1 'Z', -- високий імпеданс 'W', -- слабкі 0 або 1 'L', -- слабкий 0 (для відкритого еміттера ЕЗЛ) 'H', -- слабка 1 (для відкритого стоку або колектора) '-' -- довільне значення } **TYPE** std_ulogic_vector **is ARRAY** (**NATURAL RANGE** <>) **of** std_ulogic;
Правила використання	Типи ***std_ulogic*** і ***std_ulogic_vector*** не визначені в стандарті VHDL і тому мають використовуватись через зовнішнішній пакет. Це вимагає використання бібліотеки і оператора ***use***. Звичайно вони передують заголовку інтерфейса.	**library** IEEE; **use** IEEE.Std_logic_1164.all;
Логічні оператори	Всі логічні оператори, доступні для типів ***bit*** і ***bit_vector***, доступні і для ***std_ulogic*** і ***std_logic_vector*** відповідно. Це відноситься до операторів ***and***, ***nand***, ***or***, ***nor***, ***xor***, ***xnor*** і ***not***. В прикладі показано, які значення можна отримати при виконанні оператора ***and*** над значеннями ***std_ulogic***.	**CONSTANT** and_table : stdlogic_table := (-- -- -- ¦ U X 0 1 Z W L H - ¦ ¦ -- -- ('U','U','0','U','U','U','0','U','U'), -- ¦ U ¦ ('U','X','0','X','X','X','0','X','X'), -- ¦ X ¦ ('0','0','0','0','0','0','0','0','0'), -- ¦ 0 ¦ ('U','X','0','1','X','X','0','1','X'), -- ¦ 1 ¦ ('U','X','0','X','X','X','0','X','X'), -- ¦ Z ¦ ('U','X','0','X','X','X','0','X','X'), -- ¦ W ¦ ('0','0','0','0','0','0','0','0','0'), -- ¦ L ¦ ('U','X','0','1','X','X','0','1','X'), -- ¦ H ¦ ('U','X','0','X','X','X','0','X','X') -- ¦ - ¦);
Зростаючий/спадаючий фронт	Для більшої зручності визначені дві додаткові функції для сигналів ***std_ulogic***: *falling_edge* і *rising_edge*. Вони перевіряють наявність спадаючого або зростаючого фронта сигнала відповідно. Результат має булевий тип. В прикладі представлений D-тригер, описаний з використанням функції *rising_edge* і без використання.	**if** rising_edge(CLK) **then** Q <= D; **end if;** **if** (CLK'event **and** CLK = '1' **and** CLK'last_value = '0') **then** Q <= D; **end if;**

Рис.6.15.

Роздільна багатозначна логіка

Тип ***std_ulogic*** підтримує всі значення, потрібні для опису типової цифрової системи. Однак він є нерозділимим, що робить його непридатним у випадку багатодрайверних сигналів.

В зв'язку з цим в пакеті *Std_logic_1164* визначений ще один тип: ***std_logic***. Він об'єднує потужність дев'ятизначної логіки ***std_ulogic*** із розділенням, надаючи VHDL-розробнику дійсно універсальний логічний тип. Тип ***std_logic*** на сьогодні є промисловим стандартом де факто.

Єдина різниця між типами ***std_logic*** і ***std_ulogic*** полягає в тому, що перший є розділимою версією останнього. Завдяки цьому всі операції і функції (в тому числі *rising_edge* і *falling_edge*), визначені для ***std_ulogic***, можуть використовуватись для ***std_logic*** без будь-яких додаткових декларацій.

Існує також розділима версія для ***std_ulogic_vector***, яка називається аналогічно: ***std_logic_vector***.

	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Рис.6.16.

7. Підпрограми, пакети та бібліотеки VHDL.

Підпрограми

До підпрограм у VHDL відносяться процедури (***procedure***) і функції (***function***). Виклик процедури є оператором, а виклик функції повертає значення у виразі.

Підпрограми можуть бути задекларовані в тілі пакету, інтерфейсі об'єкту (***entity***), архітектурному тілі (***architecture***), процесі, процедурі або функції. Підпрограми можуть викликати інші підпрограми.

Функції

Загальний вигляд оператора декларування функції:

```
function ім'я функції (параметри) return тип функції;
function ім'я функції (параметри) return тип функції is
  декларативна частина
begin
  послідовні оператори
end function ім'я функції;
```

Загальний вигляд оператора виклику функції

```
ім'я функції (фактичний параметр {,фактичний параметр});
```

Фунція – це підпрограма, яка описує або алгоритм обчислення деяких значень, або поведінку деякої схеми. Важливою особливістю функції є те, що вони місять вирази, які повертають значення визначеного типу. Це їх головна відмінність від процедур.

Результат, що повертається функцією, може бути як скалярного, так і складного типу.

Функції можуть мати тип ***pure*** (без побічних ефектів) або ***impure*** (з побічними ефектами). ***Pure***-функції завжди повертають одне і теж значення для одного і того ж набору фактичних параметрів. ***Impure***-функції можуть повертати різні значення для одного і того ж набору параметрів. Крім того, ***impure***-функції можуть мати побічні ефекти, такі, як зміна зовнішніх по відношенню до неї об'єктів. За замовчуванням всі функції мають тип ***pure***.

Визначення функції складається з двох частин:

- *декларації функції*, яка складається з імені, списку параметрів та типу значення, що повертається функцією;
- *тіла функції*, що містить локальні декларації вкладених підпрограм, типів, констант, змінних, атрибутів та ін., а також послідовність операторів, що описують алгоритм, який виконується функцією.

Декларація функції є необов'язковою, оскільки тіло функції містить її копію. Однак, якщо декларація функції присутня, декларація тіла функції обов'язково повинно бути вказане в межах її видимості.

Ім'я функції, що вказується після ключового слова ***function***, може бути як ідентифікатором, так і символом оператора (якщо функція описує оператор). Задання нових функцій для існуючих операторів називається перевантаженням операторів.

Параметри функції за замовчуванням є входами, тому для них не потрібно задавати режим (напрямок). В якості параметрів можна використовувати тільки константи та сигнали. Перед іменем параметру вказується ключове слово, яким задається його клас (***constant*** або ***signal***). Якщо ключове слово не вказане, вважається, що параметр є константою.

Функції можуть викликатись рекурсивно. Функції можуть містити вкладені функції і процедури. Але тіло функції не може містити операторів ***wait*** або присвоєнь сигналів.

Приклад функції наведено на рис.7.1.

```
function Bool_2_Int (X : boolean) return integer is
begin
  if X then
    return 1;
  else
    return 0;
  end if;
end Bool_2_Int;

    .
    .
    .
    i <= Bool_2_Int (B);
    .
    .
    .
```

Рис.7.1.

Процедури

Загальний вигляд оператора декларування процедури:

```
procedure ім'я процедури (параметри);
procedure ім'я процедури (параметри) is
  декларативна частина
begin
  послідовні оператори
end procedure ім'я процедури;
```

Загальний вигляд оператора виклику процедури

```
ім'я процедури (фактичний параметр {,фактичний параметр});
```

Процедура є другим типом підпрограм. Вона містить локальні декларації та послідовність операторів. Може викликатись з будь-якого місця архітектури.

Визначення процедури складається з двох частин:

- декларація процедури, що складається з імені процедури та списку параметрів, необхідних при виклику процедури;
- тіло процедури, що містить локальні декларації та оператори, необхідні для виконання процедури.

Декларація процедури не є обов'язковою. Але якщо декларація процедури існує, обов'язково повинно бути тіло процедури, що відповідає цій декларації.

В якості параметрів процедура може приймати такі об'єкти, як константи, змінні та сигнали. Клас кожного параметра задається відповідним ключовим словом. Крім того, кожний параметр має також режим (напрямок), що може приймати значення: ***in***, ***out***, або ***inout***. Якщо ключове слово не задане, за замовчуванням вхідні параметри вважаються константами, а вихідні – змінними.

Під час виклику процедури формальні параметри заміняються фактичними. Якщо формальний параметр є константою, то фактичний параметр повинен бути виразом. Якщо ж формальний параметр є сигналом або змінною, фактичний параметр повинен мати той самий клас. Всі фактичні параметри повинні мати такі самі типи, як і відповідні формальні параметри. Процедура може і не мати параметрів.

В декларативній частині процедури можуть знаходитись декларації підпрограм, тіла підпрограм, декларації типів, підтипів, констант, змінних. Процедура може містити довільні послідовні оператори (включаючи оператори ***wait***). Однак оператор ***wait*** не можна використовувати в процедурах, що викликаються з процесів із списком чутливості або із функцій.

Виклик процедури може бути послідовним або паралельним оператором, в залежності від місця використання. Послідовний виклик процедури виконується при отриманні керування, а паралельна процедура активується, коли будь-який параметр з режимом ***in*** або ***inout*** змінює своє значення.

Процедури можуть бути вкладеними і можуть викликатись рекурсивно.

Приклад процедури наведено на рис.7.2.

```
procedure Parity
        (signal X : in std_logic_vector;
         signal Y : out std_logic) is
  variable TMP : std_logic;
begin
  TMP := '0';
  for I in X'range loop
    TMP := TMP xor X(I);
  end loop;
  Y <= TMP;
end Parity;

    .
    .
    .
    Parity(S,P);
    .
    .
    .
```

Рис.7.2.

Пакети

Пакет (***package***) у VHDL – це блок, який може містити багато різних декларацій для того, щоби використовувати їх в різних проектах. Для більшої зручності вони зберігаються в бібліотеках.

Синтаксис опису пакету:

```
-- декларація пакету
package ім'я пакету is
  декларації пакету
end package ім'я пакету;
-- тіло пакету
package body ім'я пакету is
  декларації тіла пакету
  декларації тіл підпрограм
  декларації відкладених констант
end package body ім'я пакету;
```

Пакет складається з обов'язкової декларації пакету і може містити одне необов'язкове тіло пакету.

Пакети використовуються для декларування типів, підтипів, констант, сигналів, компонентів, процедур та функцій, що можуть розділятись між різними незалежними проектами.

Об'єкти, задекларовані в пакеті, видимі в інших частинах проекту, якщо використовується оператор ***use***.

```
library Packages;
use Packages.AUXILIARY.all;
...
```

Двохрівнева специфікація пакету (декларація та тіло) дозволяє декларувати так звані *відкладені константи*, значення для яких не задається в декларації пакету. Значення відкладеної константи повинно бути вказане у відповідному тілі пакету.

Стандарт мови VHDL визначає два стандартних пакети, які доступні в довільному середовищі VHDL – пакет **STANDARD** і пакет **TEXTIO**. Перший містить базові декларації типів, констант та операторів, а другий визначає операції для маніпулювання текстовими файлами. Обидва розташовані в бібліотеці **STD**.

Тіло пакету містить закінчені визначення декларацій тіл підпрограм та значеня відкладених констант, задекларованих у відповідній декларації пакету. Інші декларації (подібні до декларації пакету) в тілі пакету також допускаються, але вони будуть видимі лише всередині тіла пакету.

Відкладені константи, задекларовані в декларації пакету, можуть використовуватись тільки у виразах за замовчуванням для локальних параметрів ***generic***, локальних портів та формальних параметрів підпрограм.

На рис.7.3 наведено приклад пакету.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package AUXILIARY is
  type MUX_input is array (integer range <>) of std_logic_vector (0 to 7);
  type operation_set is (SHIFT_LEFT, ADD);
  subtype MUX_Address is positive;
  function Compute_Address (IN1 : MUX_input) return MUX_address;
  constant Deferred_Con : integer;
end AUXILIARY;

package body AUXILIARY is
  function Compute_Address (IN1 : MUX_input) return MUX_address is
  begin
    ...
  end;
  constant Deferred_Con : integer := 177;
end package body AUXILIARY;
```

Рис.7.3.

Бібліотеки

Файл проекта може містити один або більше блоків проекту. Існують первинні та вторинні блоки проекту.

Первинні блоки проекту:

- декларація об'єкта в цілому (інтерфейсу ***entity***),
- декларація конфігурації,
- декларація пакету.

Первинний блок може бути пов'язаний із багатьма вторинними блоками проекту, що містять:

- архітектурне тіло;
- тіло пакету.

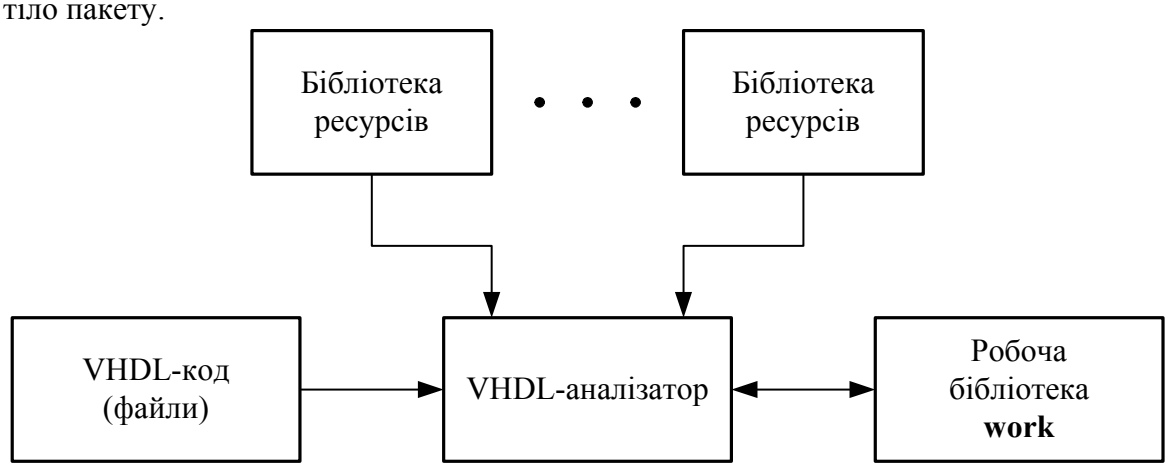


Рис.7.4.

Блоки проекту (первинні та вторинні) розміщуються в бібліотеках. *Робоча бібліотека* ***work*** може бути лише одна. В робочій бібліотеці розміщуються файли (блоки), що аналізуються VHDL-аналізатором.

В *бібліотеках ресурсів* розміщуються блоки, на які посилаються блоки, що аналізуються. Бібліотека ресурсів може бути відсутня під час аналізу.

Посилання на бібліотеку здійснюється за допомогою ключового слова ***library***. Наприклад, посилання на бібліотеку STD, що містить пакет STANDARD, і на робочу бібліотеку WORK, що містить пакет FXP, здійснюється наступним чином:

```
library STD, WORK;
use STD.STANDARD.all;
use WORK.FXP.all;
```

VHDL-аналізатор читає файли вихідного VHDL-коду, читає посилання на ресурсні бібліотеки і генерує базу даних моделювання в робочій бібліотеці.

8. Опис структури системи у VHDL. Структурні описи. Пряма реалізація інтерфейсів. Компоненти та конфігурації.

Структурні описи

Що таке структурний опис

Поведінковий VHDL-опис описує систему в термінах її операцій. *Структурний опис* описує, як система має бути зроблена, з яких підсистем або *компонентів* складатись, як вони з'єднуються між собою.

Структурний опис дозволяє організувати багаторівневу ієрархію, в якій кожний компонент може бути заданий як поведінковим, так і структурним описом. В останньому випадку він складається з підкомпонентів, які в свою чергу можуть задаватись як схеми з більш примітивних підкомпонентів і т.д. В решті решт, кожний примітивний компонент найнижчого рівня задається поведінковим описом.

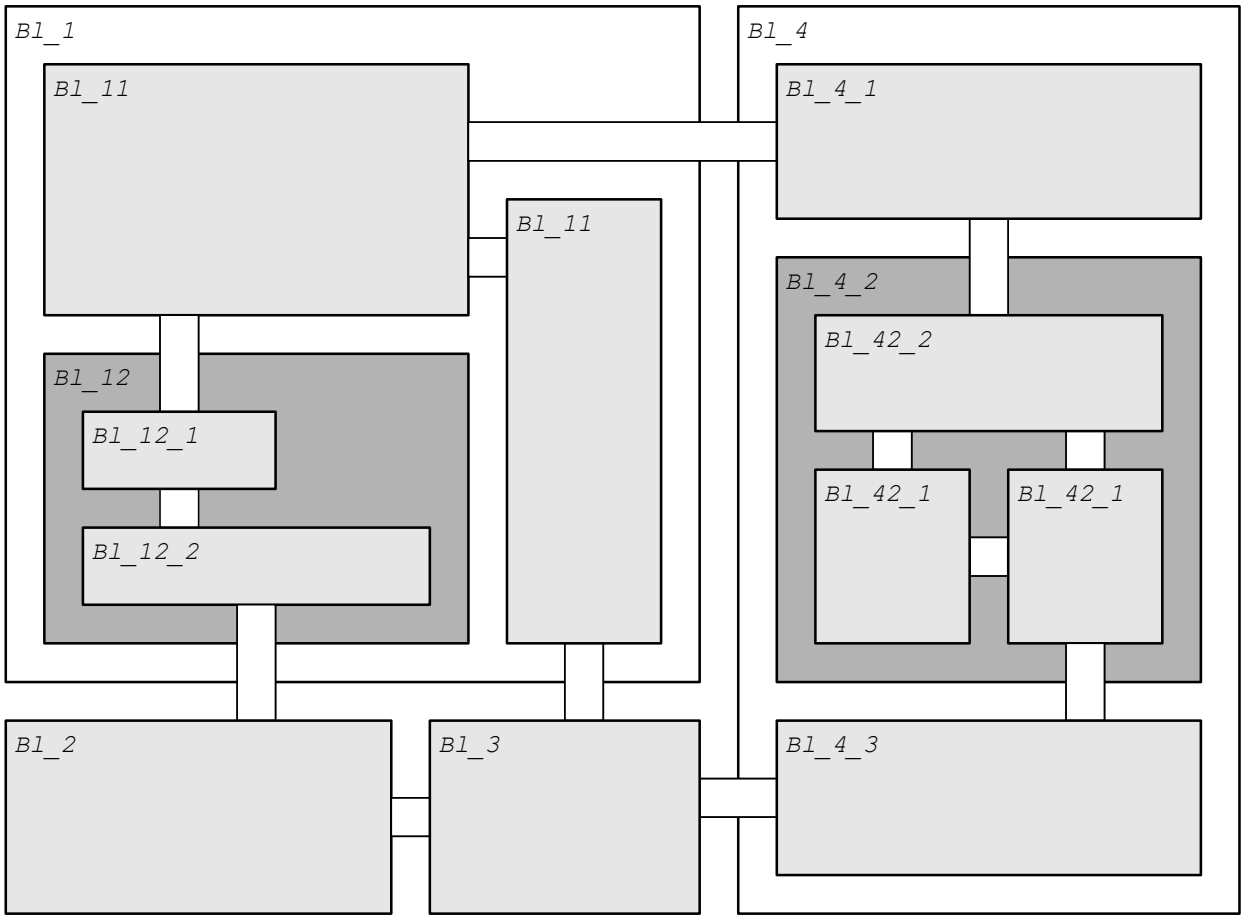


Рис.8.1.

Елементи структурного опису

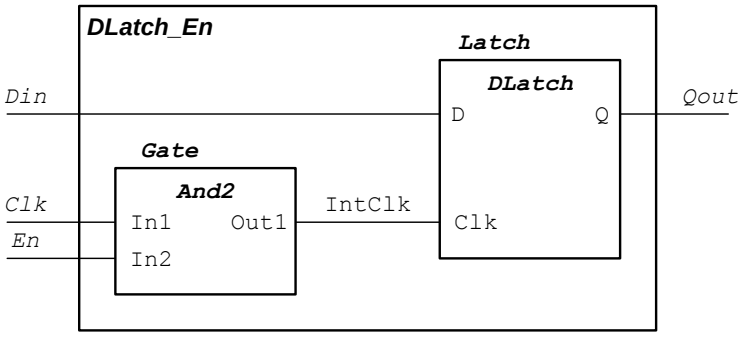
Структурний опис складається із з'єднаних компонентів. Ці компоненти з'єднуються між собою і з зовнішнім середовищем за допомогою сигналів. Структурний опис вимагає двох типів елементів:

- компонент може бути індивідуальною системою, описаною як окремий інтерфейс та архітектура;
- компонент може бути визначений всередині архітектури за допомогою декларації ***component***.

В обох випадках декларація компонента трактується як загальний опис – на зразок елементу каталогу. Для використання таких елементів вони мають бути *об'явлені* в структурному описі.

Реалізація компонента таким чином є базовим оператором структурної архітектури. Подібно іншим архітектурним операторам, реалізації компонентів паралельні між собою.

Реалізація компонента є незавершеною без оператора відображення портів ***port map*** – відповідності між реальними іменами портів структурного опису вищого рівня і формальними портами, описаними в операторі ***port*** інтерфейсу компонента. Сигнали, що використовуються у відображені портів можуть бути описані як порти, або як внутрішні сигнали системи. В останньому випадку вони мають бути задекларовані в декларативній частині архітектури.




```
entity DLatch_En is
port (Din,Clk,En : in Bit;
      Qout : out Bit);
end entity DLatch_En;

architecture Struct of DLatch_En is
signal IntClk : Bit;
begin
  Gate: entity work.And2(Beh)
    port map (In1 => Clk,
              In2 => En,
              Out => IntClk);
  Latch: entity work.Dlatch(Beh)
    port map (D => Din,
              Clk => IntClk,
              Q => Qout);
end architecture Struct;
```

Рис.8.2.

Компоненти та реалізації компонентів

Ефективне використання компонентів – центральний момент структурних VHDL-описів.

Взагалі, компонент – це довільний модуль, описаний поведінковим або структурним шляхом. В найпростішому випадку компонент – це інтерфейс із асоційованою архітектурою. Перш ніж компонент може бути використаний, він має бути реалізований. *Реалізація* – це вибір зкомпільованого опису в бібліотеці і з'єднання його з архітектурою, в якій він буде використовуватись.

Кожна реалізація компонента складається з наступних частин:

ім'я_реалізації: ***entity*** work.*ім'я_інтерфейса*(*ім'я_архітектури*)

де *work* – ім'я бібліотеки, в якій зберігаються всі елементи, визначені користувачем.

```
entity DLatch_En is
port (Din,Clk,En : in Bit;
      Qout : out Bit);
end entity DLatch_En;

architecture Struct of DLatch_En is
signal IntClk : Bit;
begin
  Gate: entity work.And2(Beh)
    port map (In1 => Clk,
              In2 => En,
              Out => IntClk);
  Latch: entity work.Dlatch(Beh)
    port map (D => Din,
              Clk => IntClk,
              Q => Qout);
end architecture Struct;
```

```
entity And2 is
port (In1,In2 : in Bit;
      Out1 : out Bit);
end entity And2;

architecture Beh of And2 is
begin
  Out1 <= In1 and In2 after 2 ns;
end architecture Beh;
```

```
entity DLatch is
port (D,Clk : in Bit;
      Q : out Bit);
end entity DLatch;

architecture Beh of DLatch is
begin
  process (Clk,D)
  begin
    if Clk = '1' then
      Q <= D after 3 ns;
    end if;
  end process;
end architecture Beh;
```

Рис.8.3.

Відображення портів

Реалізація компонента не може бути завершена без відповідного *відображення портів* – присвоєння *формальним* портам задекларованого компонента *реальних* сигналів системи.

Відображення, що використовується у наведеному прикладі, називається *іменованою асоціацією*, оскільки сигнали у відображенні портів записуються в тому ж порядку, що і порти в декларації інтерфейсу компонента.

```
entity DLatch_En is
port (Din,Clk,En : in Bit;
      Qout : out Bit);
end entity DLatch_En;

architecture Struct of DLatch_En is
signal IntClk : Bit;
begin
  Gate: entity work.And2(Beh)
    port map (In1 => Clk,
              In2 => En,
              Out => IntClk);
  Latch: entity work.Dlatch(Beh)
    port map (D => Din,
              Clk => IntClk,
              Q => Qout);
end architecture Struct;
```

```
entity And2 is
port (In1,
      In2 : in Bit;
      Out1 : out Bit
     );
end entity And2;

entity DLatch is
port (D,
      Clk : in Bit;
      Q : out Bit
     );
end entity DLatch;
```

Рис.8.4.

Пряма реалізація інтерфейсу

Пряма реалізація

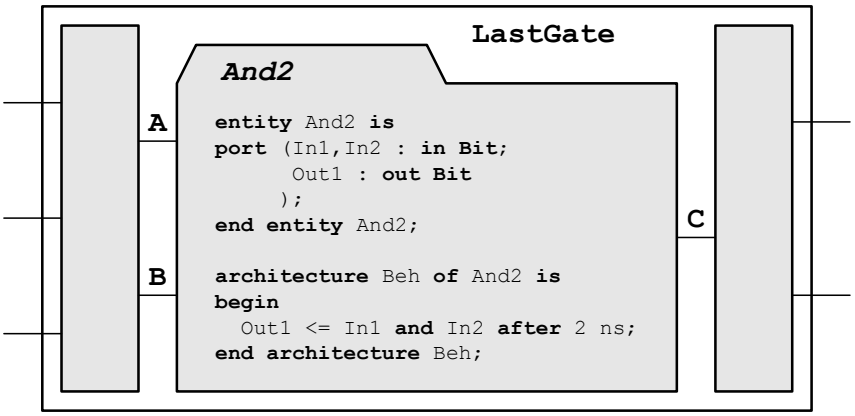
Пряма реалізація інтерфейсу – це найпростіший шлях опису структурної системи. Все, що потрібно в цьому випадку – зкомпільована специфікація компонента, яка буде реалізована, і її оператор реалізації.

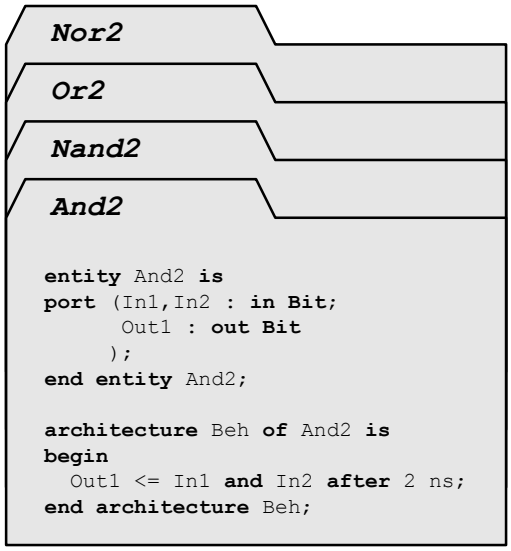
Оператор прямої реалізації складається з:

- обов'язкової мітки компонента, яка необхідна, оскільки один компонент може бути реалізований в архітектурі неодноразово,
- ключового слова ***entity***, після якого вказується ім'я бібліотеки і інтерфейсу, що містить специфікацію компонента,
- необов'язкового імені архітектури в круглих дужках,
- оператора відображення портів.

Якщо всі специфікації, визначені користувачем, компілюються в бібліотеку **work**, то і в операторі реалізації компонента бібліотека буде також називатись **work**.

Ім'я архітектури потрібне тільки в тому випадку, якщо існує декілька архітектур для одного інтерфейсу. Якщо ж специфікація системи, яка використовується як компонент, містить тільки одну архітектуру, ім'я цієї архітектури може бути не вказане.






```
entity SomeSystem is
  port (...);
end entity SomeSystem;

architecture Struct of SomeSystem is
begin
  LastGate: entity work.And2
    port map (A,B,C);
  ...
end architecture Beh;
```

Рис.8.5.

Позиційна асоціація портів

Кожний оператор відображення портів ***port map*** описує з'єднання між портами інтерфейсу (компонента) і сигналів архітектури, в якій цей компонент реалізується.

Є два шляхи відображення портів: *позиційна асоціація портів* та *іменована асоціація портів*.

При позиційній асоціації сигнали записуються в тому самому порядку, що і порти, задекларовані в інтерфейсі компонента. В цьому випадку з'єднання між сигналами та портами задається їх позицією. Наприклад, перший сигнал в списку підключається до першого порта, другий сигнал – до другого порта, і т.д. Звичайно, сигнали повинні мати той самий тип, що і відповідні їм порти.

And2

```
entity And2 is
port (In1,In2 : in Bit;
      Out1 : out Bit
     );
end entity And2;

architecture Beh of And2 is
begin
  Out1 <= In1 and In2 after 2 ns;
end architecture Beh;
```

```
entity SomeSystem is
  port (...);
end entity SomeSystem;

architecture Struct of SomeSystem is
begin
  LastGate: entity work.And2
    port map (     );
  ...
end architecture Beh;
```

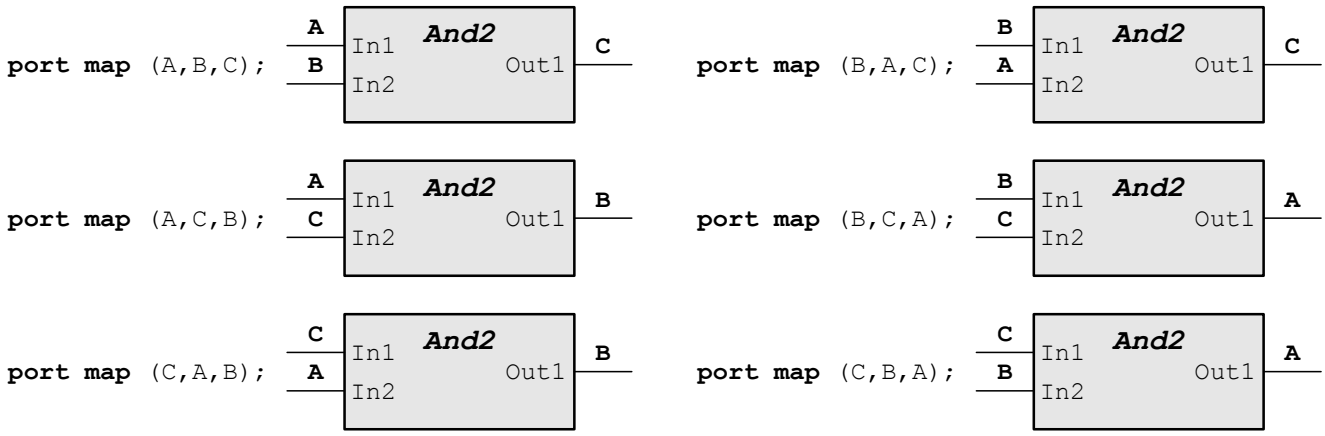


Рис.8.6.

Іменована асоціація портів

Позиційна асоціація сигналів і портів виглядає природньо. Однак деколи вона може приводити до проблем при визначенні, які сигнали підключені до яких портів. Це буває, коли в списку дуже багато сигналів.

Зручним рішенням цієї проблеми є *іменована асоціація*. В цьому випадку сигнали за іменами присвоюються асоційованим портам. При цьому порядок портів не має значення, оскільки задається пряма відповідність між формальними портами і реальними сигналами.

Асоціація між портами та сигналами задається за допомогою символа '=>'. Слід зауважити, що цей символ не задає напрямку передачі інформації через порт.

And2

```
entity And2 is
port (In1,In2 : in Bit;
      Out1 : out Bit
     );
end entity And2;

architecture Beh of And2 is
begin
  Out1 <= In1 and In2 after 2 ns;
end architecture Beh;
```

```
entity SomeSystem is
  port (...);
end entity SomeSystem;

architecture Struct of SomeSystem is
begin
  LastGate: entity work.And2
    port map (In1 =>  ,
              In2 =>  ,
              Out1 =>
             );
  ...
end architecture Beh;
```

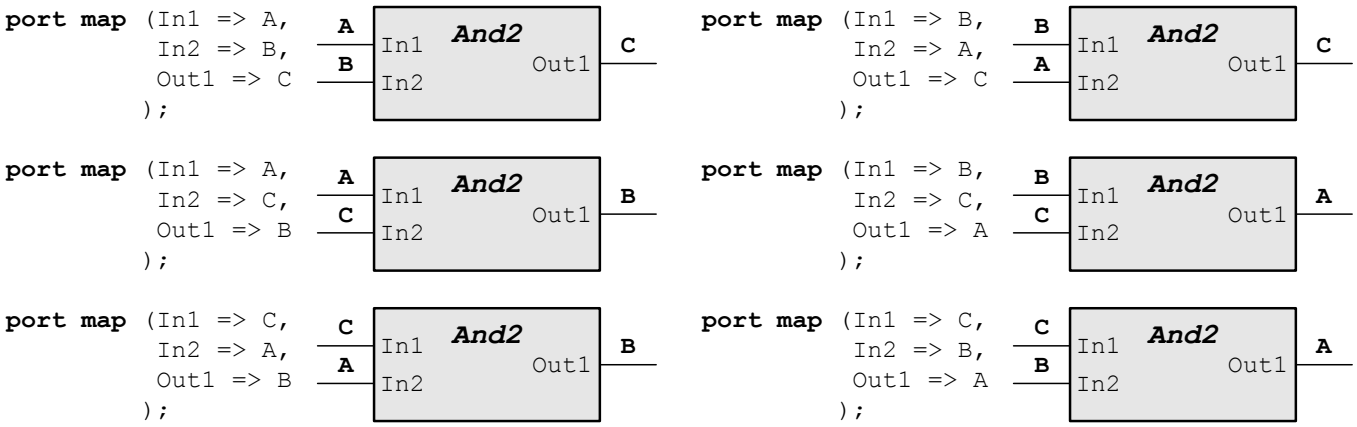


Рис.8.7.

Складні порти та сигнали

До сих пір розглядались тільки однорозрядні сигнали. Однак, складні сигнали та порти можуть також використовуватись в реалізації компонента.

VHDL забезпечує дуже велику гнучкість при присвоєнні реальних сигналів формальним портам для складних типів (масивів та записів). Такі сигнали-порти можуть присвоюватись поелементно або по шарах. Якщо типи асоційованих портів та сигналів сумісні, дозволяється будь-яка комбінація сигналів та портів.

Але при цьому всі елементи такого складного порта мають бути асоційовані із реальними сигналами.

Крім того, якщо елементи складного порта – індивідуально асоційовані сигнали, жодна з асоціацій цього порта не може бути розділена з іншими асоціаціями.

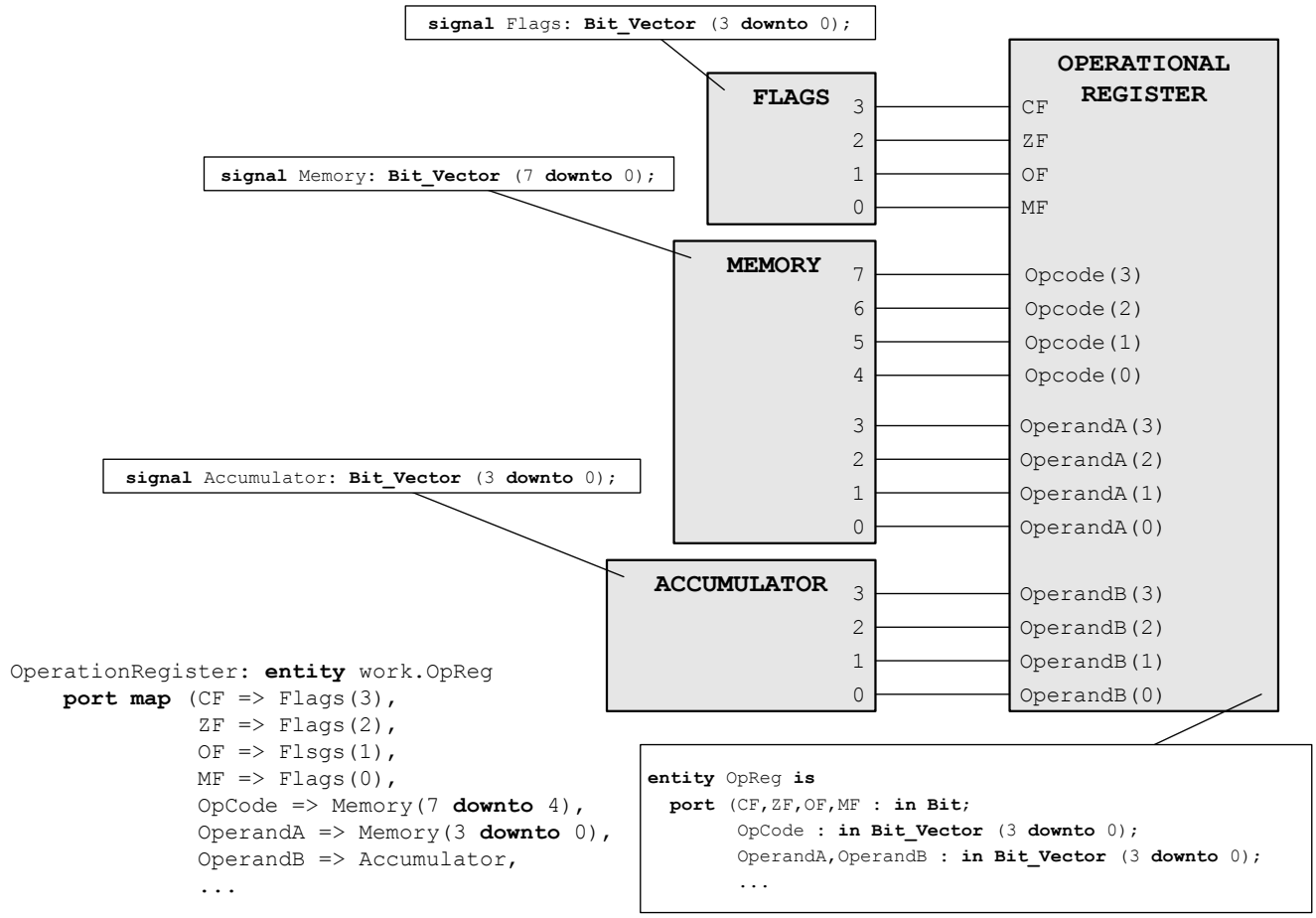
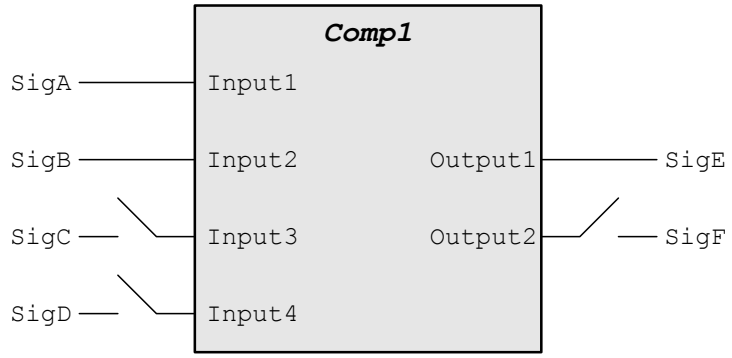


Рис.8.8

Неприєднані порти

Бувають ситуації, коли деякі порти нікуди не приєднані. Такі порти описуються як відкриті ***open*** у відображенні портів.

Крім того, порти, які не використовуються, можуть бути пропущені в операторі ***port map***. Це цілком дозволяється у VHDL, але не рекомендується, оскільки немає гарантії, що пропущені порти не вказані спеціально. Навіть із досвідченими VHDL-проектувальниками може статись ситуація, коли про один або кілька портів просто забули. Таку помилку в коді знайти дуже важко, тому рекомендується присвоювати статус ***open*** всім портам у відображенні портів реалізації компонента, які не використовуються в даній реалізації.




```
Comp1: entity work.SomeSystem
    port map (Input1 => SigA,
              Input2 => SigB,
              Input3 => open,
              Input4 => open,
              Output1 => SigE,
              Output2 => open );
```

Рис.8.9.

Компоненти та конфігурації

Вступ до компонентів

У випадку простих структурних описів, де всі компоненти повністю відомі, прямої реалізації інтерфейсів цілком достатньо.

Однак, коли розробляються нові проекти, компоненти створюються паралельно, і на деяких стадіях буває необхідно посилатися на компоненти, структурна або поведінкова реалізація яких ще не виконана.

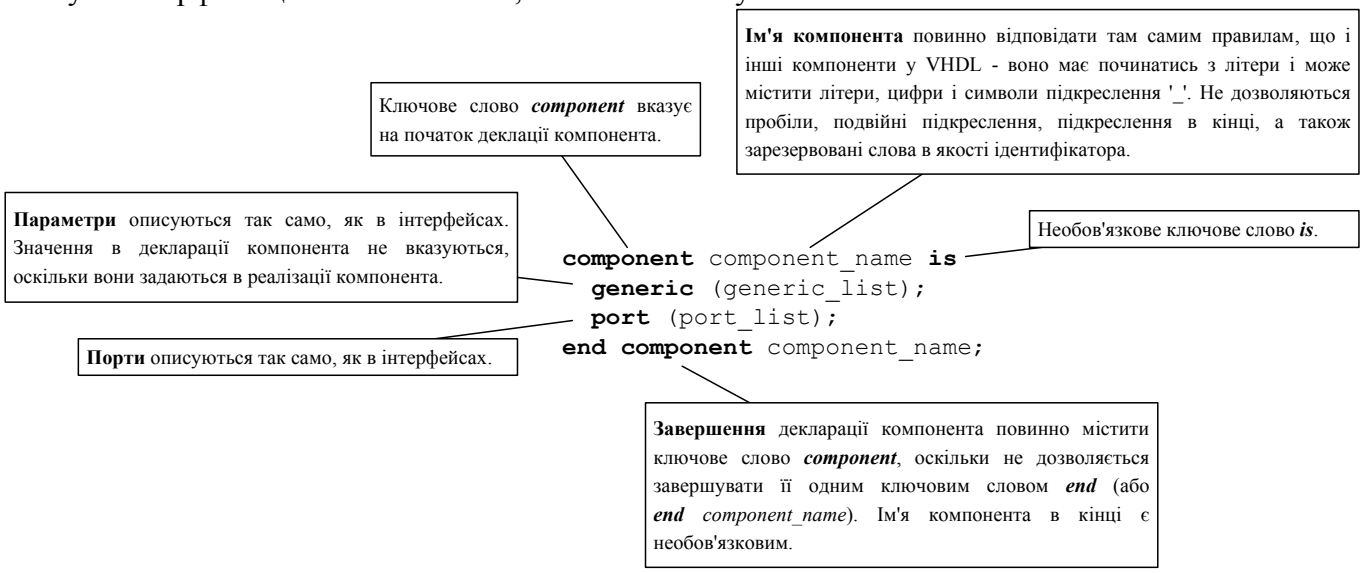
В таких ситуаціях для структурної специфікації було би достатньо мати декларацію інтерфейсу компонента (як вимагає система). Декларація інтерфейсу компонента називається *декларацією компонента* і розташовується в декларативній частині тіла архітектури (або інтерфейсу пакету).

Декларація компонента

Синтаксис *декларації компонента* і інтерфейсу подібні. Це не випадково, оскільки *компонент* та *інтерфейс* відіграють подібну роль у визначенні зовнішніх інтерфейсів модулів.

Але при цьому існує важлива різниця між цими двома конструкціями VHDL:

- декларація *інтерфейсу* визначає інтерфейс "реального" модуля, тобто системи або схеми, що фізично існує, система є окремою одиницею проекту і може бути індивідуально відсимульована, проаналізована і синтезована;
- декларація *компонента*, з іншого боку, визначає інтерфейс "неіснуючого" або "віртуального" модуля, він описується всередині архітектури і є скоріше припущенням про те, яким мав би бути інтерфейс цього компонента, ніж описом існуючої схеми.




```
architecture Struct of Reg4 is

component DFF is
  generic (t_prop  : time;  -- час розповсюдження
           t_setup : time); -- час встановлення
  port (D   : in bit;       -- інформаційний вхід
        Clk : in bit;       -- тактовий вхід
        Rst : in bit;       -- асинхронний скид
        Q   : out bit);     -- інформаційний вихід
end component DFF;
begin
  ...
```

Рис.8.10.

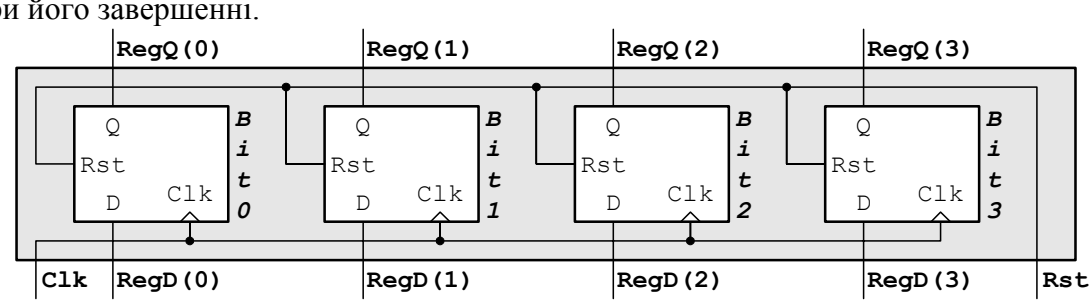
Реалізації компонентів

Реалізація компонента дуже подібна на реалізацію інтерфейсу. Вона містить наступні головні елементи:

- *мітка реалізації*, після якої стоїть двокрапка; ця мітка є обов'язковою і потрібна для ідентифікації реалізації;

- необов'язкове *ключове слово* ***component***, яке вказує на те, що це реалізація компонента, рекомендується використовувати для підвищення читабельності коду;
- *ім'я компонента*, яке описане в декларації компонента;
- *оператор* ***generic map***, який забезпечує реальні значення параметрів компонента; параметри можуть бути описані без значень, в цьому випадку реальні значення задаються оператором ***generic map***; якщо компонент описаний без параметрів, цей оператор може бути опущений;
- *оператор* ***port map***, який задається так само, як і при прямій реалізації.

Слід зауважити, що ім'я компонента та оператор ***generic map*** не завершуються крапкою з комою. Усі наведені елементи формують єдиний оператор реалізації і крапка з комою вказується тільки при його завершенні.




```
entity Reg4 is
  port (RegD : in bit_vector (3 downto 0);
        Clk,Rst : in bit;
        RegQ : out bit_vector (3 downto 0));
end entity Reg4;
```

```
architecture Struct of Reg4 is
component DFF is
  generic (t_prop,t_setup : time);
  port (D,Clk,Rst : in bit;
        Q   : out bit);
end component DFF;
begin
  bit0: component DFF
    generic map (t_prop => 2 ns,t_setup => 1 ns)
    port map (D => RegD(0),Clk => Clk,Rst => Rst,Q => RegQ(0));
  bit1: component DFF
    generic map (t_prop => 2 ns,t_setup => 1 ns)
    port map (D => RegD(1),Clk => Clk,Rst => Rst,Q => RegQ(1));
  bit2: component DFF
    generic map (t_prop => 2 ns,t_setup => 1 ns)
    port map (D => RegD(2),Clk => Clk,Rst => Rst,Q => RegQ(2));
  bit3: component DFF
    generic map (t_prop => 2 ns,t_setup => 1 ns)
    port map (D => RegD(3),Clk => Clk,Rst => Rst,Q => RegQ(3));
end architecture Struct;
```

Рис.8.11.

Оператор generate

Оператор ***generate*** забезпечує спрощення опису регулярних структур проекту. Як правило, він використовується для задання групи ідентичних компонентів.

Оператор ***generate*** складається з трьох основних частин:

- схеми генерації (*for*-схема або *if*-схема);
- декларативної частини (локальні декларування підпрограм, типів, сигналів, констант, компонентів, атрибутів, конфігурацій та ін.);
- паралельних операторів.

Схема генерації задає спосіб, яким генерується паралельна структура. Можуть використовуватись дві схеми генерації: *for*-схема або *if*-схема.

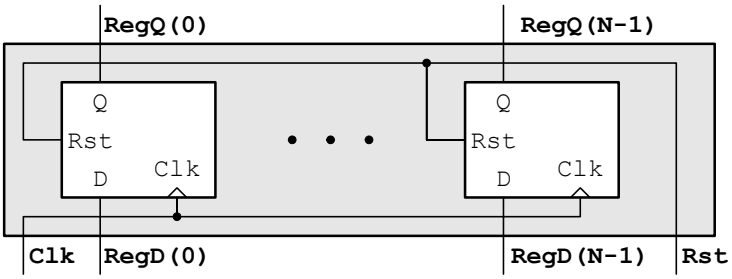
For-схема генерації використовується для опису регулярних структур проекту. В цьому випадку параметр генерації та набір значень генеруються аналогічно послідовному оператору циклу. Приклад на рис.8.12 демонструє використання *for*-схеми генерації.

Оператор ***generate*** може містити довільні паралельні оператори: оператори процесів, оператори паралельного виклику процедур, оператори реалізації компонентів, оператори паралельного присвоєння сигналів, інші оператори генерації. Останній механізм дозволяє створювати вкладені структури проекту і формувати багатовимірні масиви компонентів.

If-схема генерації використовується, як правило, у вкладених операторах генерації, коли регулярна структура містить деякі нерегулярні елементи. Приклад на рис.8.13 демонструє використання *if*-схеми генерації.

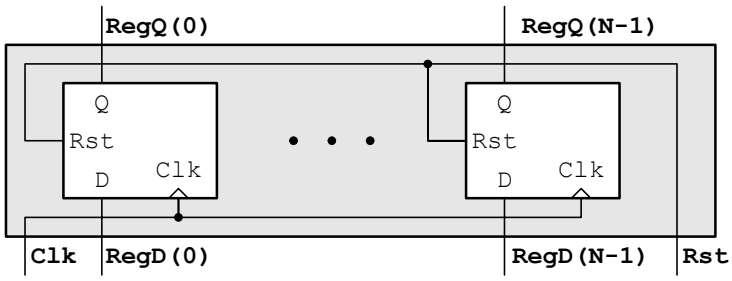
Вкладені оператори генерації використовуються для скорочення опису. Зовнішній оператор ***generate*** задає загальний лічильник, а компоненти всередині структури генеруються в залежності від його значення.

Кожний оператор генерації обов'язково повинен мати мітку. Якщо оператор генерації містить локальні декларації, то між ними та тілом оператора необхідно вставляти ключове слово ***begin***.




```
entity RegN is
  generic (N : integer := 4);
  port (RegD : in bit_vector (N-1 downto 0);
        Clk,Rst : in bit;
        RegQ : out bit_vector (N-1 downto 0));
end entity RegN;

architecture Struct of RegN is
component DFF is
  generic (t_prop,t_setup : time);
  port (D,Clk,Rst : in bit;
        Q   : out bit);
end component DFF;
begin
  regbits: for i in 0 to N-1 generate
     bitn: component DFF
       generic map (t_prop => 2 ns,t_setup => 1 ns)
       port map (D => RegD(i),Clk => Clk,Rst => Rst,Q => RegQ(i));
  end generate;
end architecture Struct;
```

Рис.8.12.




```
entity RegN is
  generic (N : integer := 4);
  port (RegD : in bit_vector (N-1 downto 0);
        Clk,Rst : in bit;
        RegQ : out bit_vector (N-1 downto 0));
end entity RegN;

architecture Struct of RegN is
component DFF is
  generic (t_prop,t_setup : time);
  port (D,Clk,Rst : in bit;
        Q   : out bit);
end component DFF;
begin
  regbits: for i in 0 to N-1 generate
     bit1: if i = 1 generate
       DFF1: component DFF
         generic map (t_prop => 4 ns,t_setup => 2 ns)
         port map (D => RegD(i),Clk => Clk,Rst => Rst,Q => RegQ(i));
     end generate;
     bits: if i /= 1 generate
       DFFS: component DFF
         generic map (t_prop => 2 ns,t_setup => 1 ns)
         port map (D => RegD(i),Clk => Clk,Rst => Rst,Q => RegQ(i));
     end generate;
  end generate;
end architecture Struct;
```

Рис.8.13.

Конфігурації

Декларації компонента та його реалізації недостатньо для того, щоби отримати завершену специфікацію структурної архітектури, оскільки відсутній опис реалізації компонентів. Інформація, яка *зв'язує* компонент з реальними інтерфейсами і архітектурами, задається в *конфігурації*, яка є окремим елементом проекту, і може бути відсимульована та проаналізована незалежно від інших елементів.

Якщо розглянути зв'язування компонентів ближче, можна зауважити, що воно дуже подібне прямій реалізації компонента. Однак, використання конфігурації є більш гнучке і зручне для підтримки, ніж інші реалізації цих компонентів, які могли б використовуватись. Якщо необхідно зробити якісь зміни, то вони вносяться тільки в файл конфігурації, який є відносно коротким. Структурна архітектура лишається незмінною. Використання ж прямих реалізацій вимагає внесення всіх змін безпосередньо в код.

Якщо використовується багато рівнів ієрархії, конфігурації можуть бути досить складними. Нижче наведено простий варіант конфігурації.

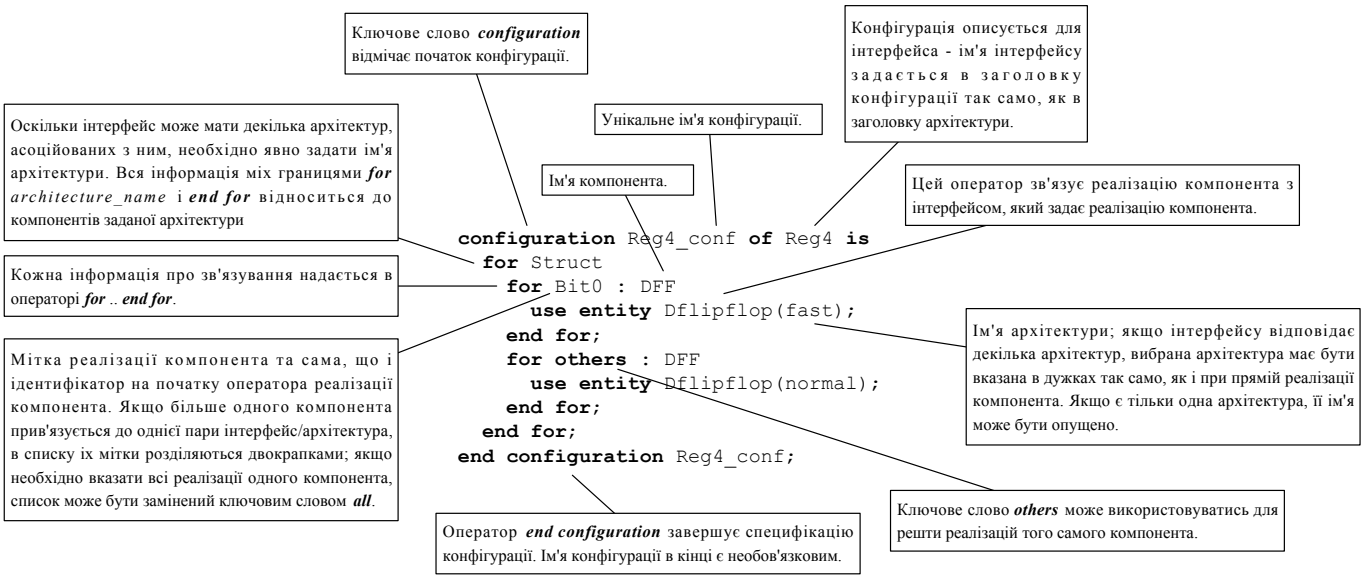

Ключове слово *configuration* відмічає початок конфігурації.
Конфігурація описується для інтерфейса - ім'я інтерфейсу задається в заголовку конфігурації так само, як в заголовку архітектури.
Унікальне ім'я конфігурації.
Оскільки інтерфейс може мати декілька архітектур, асоційованих з ним, необхідно явно задати ім'я архітектури. Вся інформація між границями *for architecture_name* і *end for* відноситься до компонентів заданої архітектури
Ім'я компонента.
Цей оператор зв'язує реалізацію компонента з інтерфейсом, який задає реалізацію компонента.
configuration Reg4_conf of Reg4 is
for Struct
for Bit0 : DFF
use entity Dflipflop(fast);
end for;
for others : DFF
use entity Dflipflop(normal);
end for;
end for;
end configuration Reg4_conf;
Кожна інформація про зв'язування надається в операторі *for* .. *end for*.
Ім'я архітектури; якщо інтерфейсу відповідає декілька архітектур, вибрана архітектура має бути вказана в дужках так само, як і при прямій реалізації компонента. Якщо є тільки одна архітектура, її ім'я може бути опущено.
Мітка реалізації компонента та сама, що і ідентифікатор на початку оператора реалізації компонента. Якщо більше одного компонента прив'язується до однієї пари інтерфейс/архітектура, в списку їх мітки розділяються двокрапками; якщо необхідно вказати всі реалізації одного компонента, список може бути замінений ключовим словом *all*.
Ключове слово *others* може використовуватись для решти реалізацій того самого компонента.
Оператор *end configuration* завершує специфікацію конфігурації. Ім'я конфігурації в кінці є необов'язковим.

Рис.8.14.

9. Принципи логічного моделювання та синтезу. Тестування VHDL-проектів за допомогою тестових стендів.

Моделювання VHDL-описів

VHDL-код описує поведінку проектованої цифрової системи і являє собою звичайний текстовий файл. Виконання VHDL-опису проводиться за допомогою спеціальної програми – системи моделювання.

Система моделювання включає засоби, призначені для:

- *організації проекту* – визначення директорії проекту, розташування в ній необхідних файлів з вихідними VHDL-кодами, необхідними пакетами та бібліотеками VHDL-описів;
- *компіляції* – перетворення VHDL-кодів у внутрішнє представлення, яке і виконується (моделюється);
- *зборки* (*лінкування*);
- *моделювання* – виконання VHDL-кодів, представлених у внутрішній формі;
- *візуалізації* вихідних описів та результатів моделювання в різних формах – текстовій або графічній (часові діаграми).

Для моделювання цифрової системи необхідно створення спеціального тестового опису (теста) – оболонки, основне призначення якої

- організувати подавання вхідних сигналів,
- отримати реакцію тестованої системи,
- порівняти, якщо потрібно, реакцію схеми з очікуваною.

Верифікація за допомогою тестових стендів

Що таке тестовий стенд?

Найбільш популярним способом верифікації є використання тестових стендів. *Тестовий стенд* – це середовище, в якому проект (називається *проект* або *тестований елемент*) перевіряється за допомогою сигналів (*стимуляторів*) з відображенням його реакцій. Іншими словами, тестовий стенд заміняє середовище проекту таким чином, що поведінка проекту може бути спостережена і проаналізована.

Тестовий стенд складається з наступних елементів:

- сокет для тестованого пристрою (UUT – unit under test),
- генератор стимуляторів (підсистема, що застосовує стимулятори до UUT, генеруючи їх автономно, або читаючи із зовнішнього джерела),
- засоби відображення реакцій UUT на стимулятори.

Тестові стенди у VHDL

Ідея тестових стендів адаптована до проектів у формі VHDL-специфікації. В цьому випадку тестовий стенд не є самостійною системою, а тільки VHDL-специфікацією, що симулюється VHDL-симулятором. Він складається з реалізації тестованого пристрою (UUT) і процесів, що підтримують стимулятори, які застосовуються до UUT. При цьому створюється гібридна специфікація, в якій використовуються як структурні, так і поведінкові оператори. Такий підхід дозволяється у VHDL, оскільки як реалізація компонента, так і процеси є паралельними операторами.

Стимулятори для UUT описуються всередині архітектури тестового стенду, або можуть бути прочитані із зовнішнього файлу. Реакції UUT, з іншого боку, можуть спостерігатись як засобами симулятора у вигляді повідомлень симуляції (наприклад, часові діаграми, що спостерігаються на екрані), так і у вигляді файлу, створеного операторами текстового вводу-виводу VHDL.

Для підвищення гнучкості при написанні тестових стендів використовується велика кількість опцій. Однак, їх використання може привести до того, що тестові стенди стануть складніші , ніж UUT, для яких вони були написані.

Елементи тестового стенду VHDL

Як зазначалось вище, тестовий стенд VHDL – це просто інша специфікація із власним інтерфейсом та архітектурою. Але вона має спеціальну структуру із декількома елементами, які є характерними для цього типу специфікації:

- **інтерфейс тестового стенду** не має портів,
- **реалізація компонента UUT** – відповідність між тестовим стендом і UUT задається за допомогою реалізації компонента і структурної специфікації,
- **стимулятори** – це набір сигналів, що декларуються всередині архітектури тестового стенду і присвоюються портам UUT в його реалізації; стимулятори визначаються як часові діаграми в одному або більше поведінковому процесі.

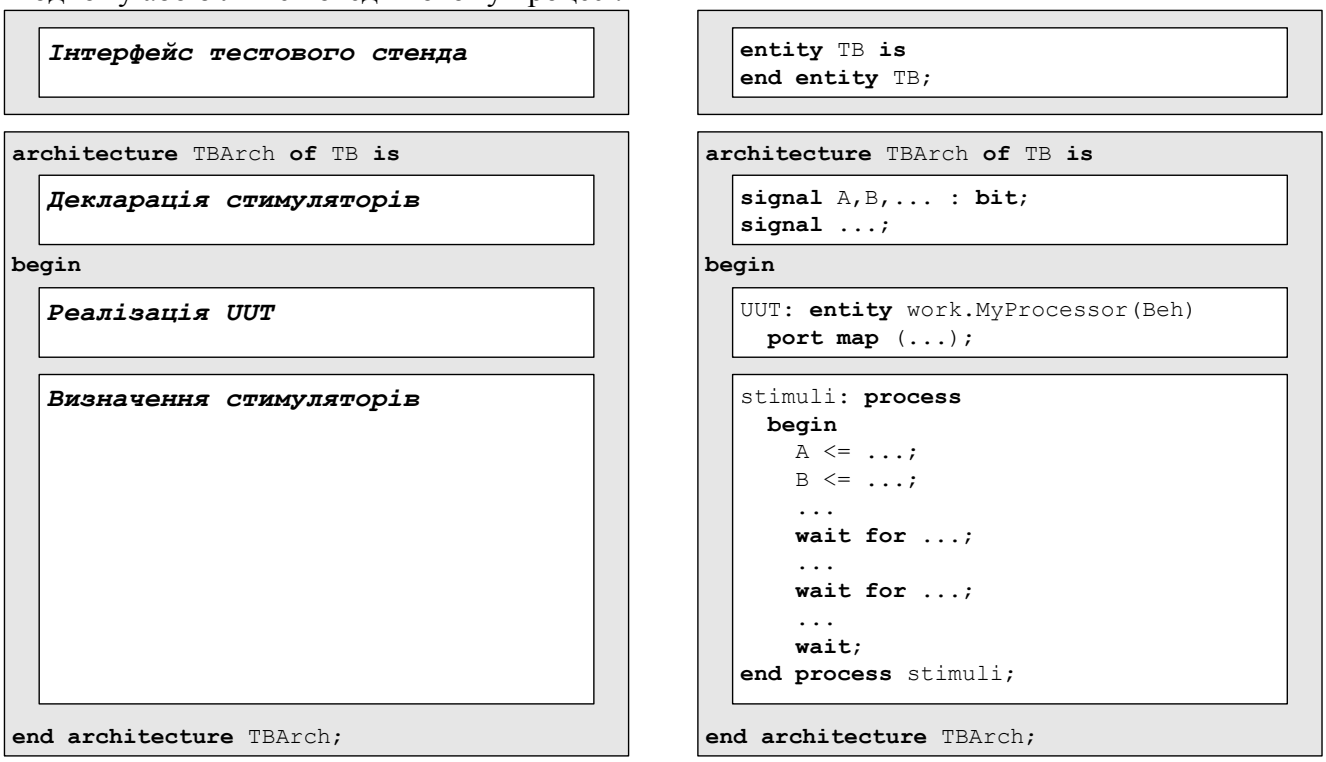
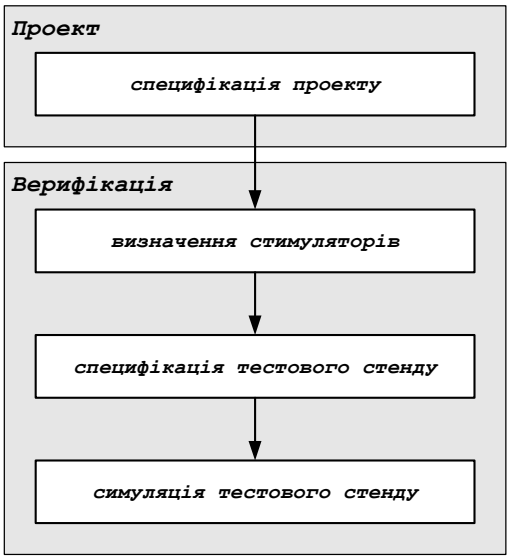


Рис.9.1.

Використання тестових стендів

Вже при створенні нової VHDL-специфікації, необхідно враховувати, що проект повинен бути верифікований. Очевидно, довільний проект може бути просто відсимульований, але для великих проектів це може зайняти досить багато часу. Набагато зручніше використовувати для верифікації проектів тестові стенди. Оскільки створення тестового стенду може бути досить складною задачею, рекомендується створювати відповідну документацію для майбутньої розробки стимуляторів вже на початку проектування. Деякі експерти навіть радять відразу створювати закінчені стимулятори.

Коли UUT і тестовий стенд для нього завершені, можна починати верифікацію. При цьому симулюватись буде не UUT, а тестовий стенд. UUT в цьому випадку – це просто один з компонентів, реалізованих в тестовому стенді. Обмежень на розмір тестових стендів не існує. Є тільки обмеження на ємність симулятора, що використовується.



Процес проектування складається з фаз **проекту** і **верифікації**. Метою першої фази є створення VHDL-специфікації, що відповідає вимогам до системи.

Створення **стимуляторів** може виконуваись паралельно із створенням специфікацій кожного блоку проекту. Набір стимуляторів має містити такий набір значень вхідних сигналів (і станів), що по можливості найбільше відповідає ральним ситуаціям.

Коли стимулятори створені, може бути написана специфікація **тестового стенду**. Вона буде містити стимулятори та реалізацію спроектованої системи (UUT). На цій фазі верифікації симулюється не проект, а тестовий стенд.

Симуляція тестового стенду - це остання фаза процесу проектування. На ній можна отримати відповідь на питання, "відповідає поведінка системи очікуванням?" Відповідь, отримана при симуляції тестового стенду, залежить від границь, що визначаються точністю і повнотою тестового стенду. Чим кращим є тестовий стенд, тим більш впевненим можна бути у тому, що система розроблена правильно.

Рис.9.2.

Структура тестового стенду

Інтерфейс тестового стенду

Тестовий стенд VHDL – це просто інша специфікація, яка складається з інтерфейсу та архітектури. Але є одна важлива особливість інтерфейсу тестового стенду – він не містить портів або параметрів.

Причина цього дуже проста – тестовий стенд не є реальним пристроєм або системою, що може з'єднуватись із середовищем, отже він не потребує входів або виходів. Усі значення для вхідних портів UUT описуються всередині архітектури тестового стенду як стимулятори. Виходи спостерігаються за допомогою симулятора, і відображаються у вигляді часових діаграм, або зберігаються в файлі.

Чому тестовий стенд представляється у вигляді інтерфейсу, якщо він практично є лише архітектурою? Як вказувалось вище, архітектура не може бути описана без інтерфейсу, і це правило розповсюджується також на тестові стенди.

Тестований пристрій

Система, верифікація якої буде проводитись за допомогою тестового стенду, не вимагає ніяких модифікацій, або додаткових декларацій. Завдяки цьому тестові стенди можуть бути застосовані до будь-яких VHDL-специфікацій, навіть отриманих із зовнішнього джерела. Така ситуація може мати місце, коли виконується симуляція декількох пристроїв на рівні макету. Однак ніяких модифікацій до специфікацій пристроїв не вноситься, оскільки вони проводяться на основі результатів верифікації.

UUT має бути реалізований в архітектурі тестового стенду. Це виконується так само, як і в будь-якій структурній специфікації – або шляхом прямої реалізації, або шляхом реалізації компонента з декларацією компонента і конфігурацією. Портам UUT присвоюються сигнали-стимулятори.

Як процеси, так і реалізації компонентів є паралельними операторами, отже немає значення, що буде виконано раніше – реалізація UUT чи визначення стимуляторів в архітектурі тестового стенду.

	Тестований пристрій (Unit Under Test)	*Тестовий стенд (Test Bench)*

Mux2To1	`entity Mux2To1 is`
	`  port (A,B,Sel : in bit;`
	`  Y : out bit);`
	`end entity Mux2to1;`
	
	`architecture Beh of Mux2to1 is`
	`...`
	`end architecture Beh;`
		`entity TestBench is`
		`end entity TestBench;`
		
		`architecture TBArch of TestBench is`
		`...`
		`begin`
		`  UUT: entity work.Mux2to1`
		`  port map (A => ...,`
		`  B => ...,`
		`  Sel => ...,`
		`  Y => ...);`
		`...`
		`end architecture TBArch;`

Reg8	`entity Reg8 is`
	`  port (D : in bit_vector (7 downto 0);`
	`  En,Clk : in bit;`
	`  Q : out bit_vector (7 downto 0));`
	`end entity Reg8;`
	
	`architecture Beh of Reg8 is`
	`...`
	`end architecture Beh;`
		`entity TestBench is`
		`end entity TestBench;`
		
		`architecture TBArch of TestBench is`
		`...`
		`begin`
		`  UUT: entity work.Reg8`
		`  port map (D => ...,`
		`  En => ...,`
		`  Clk => ...,`
		`  Q => ...);`
		`...`
		`end architecture TBArch;`

Dec2to4	`entity Dec2to4 is`
	`  port (Inputs : in bit_vector (1 downto 0);`
	`  Outputs : out bit_vector (3 downto 0));`
	`end entity Dec2to4;`
	
	`architecture Beh of Dec2to4 is`
	`...`
	`end architecture Beh;`
		`entity TestBench is`
		`end entity TestBench;`
		
		`architecture TBArch of TestBench is`
		`...`
		`begin`
		`  UUT: entity work.Dec2to4`
		`  port map (Inputs => ...,`
		`  Outputs => ...);`
		`...`
		`end architecture TBArch;`

JK_flipflop	`entity JK_flipflop is`
	`  port (J,K,CLK : in bit;`
	`  Q,NQ : out bit);`
	`end entity JK_flipflop;`
	
	`architecture Beh of JK_flipflop is`
	`...`
	`end architecture Beh;`
		`entity TestBench is`
		`end entity TestBench;`
		
		`architecture TBArch of TestBench is`
		`...`
		`begin`
		`  UUT: entity work.JK_flipflop`
		`  port map (J => ...,`
		`  K => ...,`
		`  CLK => ...,`
		`  Q => ...,`
		`  NQ => ...);`
		`...`
		`end architecture TBArch;`

Рис.9.3.

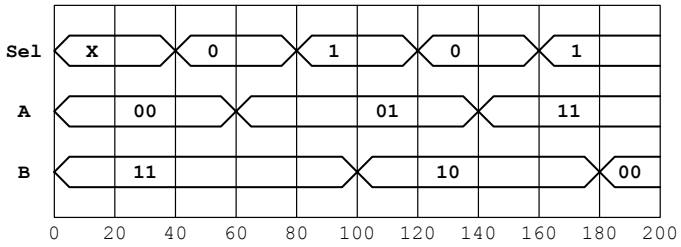
Стимулятори сигналів

Ядром кожного тестового стенду є набір стимуляторів – послідовність значень для кожного вхідного сигналу UUT, прив'язаних до часу. Оскільки тестовий стенд не з'єднується із середовищем за допомогою сигналів, всі стимулятори мають бути задекларовані всередині архітектури тестового стенду. Вони декларуються так само, як будь-які інші сигнали в декларативній частині архітектури.

Стимулятори можуть бути описані як паралельні присвоєння сигналів (із вхідними сигналами, зміни яких описуються як часові діаграми), або в процесах, що містять присвоєння сигналів, розділених операторами ***wait for***, за допомогою яких задаються затримки між

послідовними присвоєннями сигналів. В другому випадку в якості останнього оператора процесу вказується пустий (безумовний) оператор ***wait***. Це приводить до припинення симуляції (в протилежному випадку вона знову почнеться з початку процесу).

Відповідність між стимуляторами і UUT задається за допомогою асоціацій в операторі відображення портів реалізації UUT.



UUT

```
entity Mux2To1_2bit is
  generic (MuxDel : time := 5 ns);
  port (A,B : in std_logic_vector (1 downto 0);
        Sel : in std_logic;
        Y : out std_logic_vector (1 downto 0));
end entity Mux2to1_2bit;

architecture Beh of Mux2to1_2bit is
begin
  with Sel select
    Y <= A after MuxDel when '0',
         B after MuxDel when '1',
         "XX" when others;
end architecture Beh;
```

```
entity TestBench is
end entity TestBench;

architecture TestBenchArch of TestBench is
signal A,B,Y : std_logic_vector (1 downto 0);
signal Sel : std_logic;
begin
  UUT: entity work.Mux2to1_2bit
    port map (A,B,Sel,Y);
  stimuli: process
  begin
    Sel <= 'X'; A <= "00"; B <= "11"; wait for 40 ns;
                                -- зміна @ 0 ns
    Sel <= '0'; wait for 20 ns; -- зміна @ 40 ns
    A <= "01"; wait for 20 ns;  -- зміна @ 60 ns
    Sel <= '1'; wait for 20 ns; -- зміна @ 80 ns
    B <= "10"; wait for 20 ns;  -- зміна @ 100 ns
    Sel <= '0'; wait for 20 ns; -- зміна @ 120 ns
    A <= "11"; wait for 20 ns;  -- зміна @ 140 ns
    Sel <= '1'; wait for 20 ns; -- зміна @ 160 ns
    B <= "00"; wait for 20 ns;  -- зміна @ 180 ns
    wait;
  end process stimuli;
end architecture TestBenchArch;
```

⇔

```
entity TestBench is
end entity TestBench;

architecture TestBenchArch of TestBench is
signal A,B,Y : std_logic_vector (1 downto 0);
signal Sel : std_logic;
begin
  UUT: entity work.Mux2to1_2bit
    port map (A,B,Sel,Y);
  Sel <= 'X', '0' after 40 ns,
              '1' after 80 ns,
              '0' after 120 ns,
              '1' after 160 ns;
  A <= "00",
       "01" after 60 ns,
       "11" after 140 ns;
  B <= "11",
       "10" after 100 ns,
       "00" after 180 ns;
end architecture TestBenchArch;
```

Рис.9.4.

Оператор повідомлення

Останній елемент успішної верифікації пристрою – отримання результатів симуляції або результатів сигналізації. Їх можна отримати декількома шляхами, використовуючи вбудовані засоби симулятора, такі як відображення списку значень сигналів із зазначеними моментами змін або часових діаграм, запис в файл логів завершеної симуляції, або використання VHDL-оператора ***assert-report***.

Останній варіант легко реалізується і використовується для відображення повідомлення, коли щось відбувається неправильно. Якщо він використовується і не відображається ніяких повідомлень на протязі симуляції, можна вважати, що UUT працює, як і очікувалось.

Оператор ***assert-report*** складається з трьох елементів:

- оператора ***assert*** (перевіряє логічну умову),
- оператора ***report*** (визначає повідомлення, що буде відображено, якщо умова не виконується),
- оператора ***severity*** (інформує симулятор, наскільки серйозною є помилка – від загального зауваження до системної помилки і має перечислимий тип: **NOTE**, **WARNING**, **ERROR**, **FAILURE**).

```
assert (LeftNum > RightNum)
  report "RightNum greater than LeftNum"
  severity warning;
```

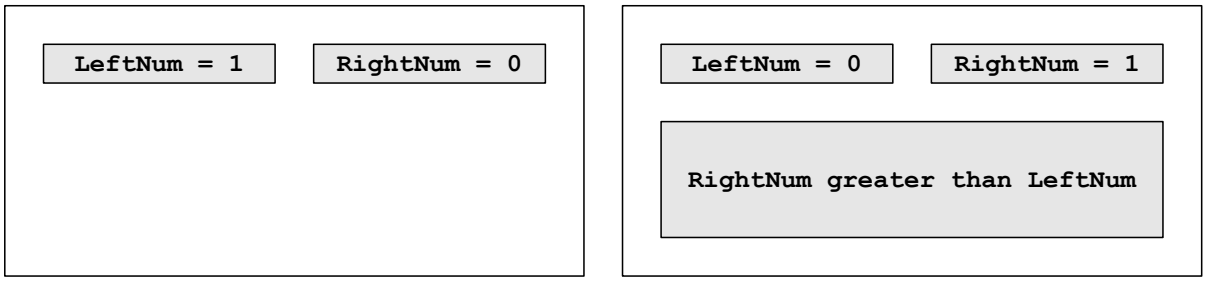


Рис.9.5.

Сигналізація за допомогою повідомлень

Оператор повідомлення є за природою послідовним, і тому використовується всередині процесів. Його застосування не обмежується тестовими стендами, але тут вони використовуються найчастіше.

Загальними правилами використання операторів повідомлень для сигналізації некоректних реакцій UUT на стимули є наступні:

- використовувати пару ***assert-report*** кожний раз, коли очікується нове значення на виході UUT;
- задавати очікуване значення в якості умови повідомлення;
- текстові рядки повідомлень повинні нести певний зміст – повідомлення "Error" нічого не дасть під час симуляції, краще вказувати, що саме неправильне і коли трапилось (наприклад, при якому стані входів).

Слід також пам'ятати, що нові значення присвоюються сигналам тільки при припиненні процесів – немає змісту очікувати зміни виходів відразу після присвоєння нових значень входам.

UUT

```
entity Mux2To1_2bit is
  generic (MuxDel : time := 5 ns);
  port (A,B : in std_logic_vector (1 downto 0);
        Sel : in std_logic;
        Y : out std_logic_vector (1 downto 0));
end entity Mux2to1_2bit;

architecture Beh of Mux2to1_2bit is
begin
  with Sel select
    Y <= A after MuxDel when '0',
         B after MuxDel when '1',
         "XX" when others;
end architecture Beh;
```

```
entity TestBench is
end entity TestBench;

architecture TestBenchArch of TestBench is
signal A,B,Y : std_logic_vector (1 downto 0);
signal Sel : std_logic;
begin
  UUT: entity work.Mux2to1_2bit
    port map (A,B,Sel,Y);
  stimuli: process
  begin
    Sel <= 'X'; A <= "00"; B <= "11"; wait for 0 ns;
    assert (Y="XX") report "Test failed on Sel=X";
    Sel <= '0'; wait for 40 ns;
    assert (Y="00") report "Test failed on Sel=0";
    A <= "01"; wait for 20 ns;
    assert (Y="01") report "Test failed on Sel=0 - A not changed";
    Sel <= '1'; wait for 20 ns;
    assert (Y="11") report "Test failed on Sel=1";
    B <= "10"; wait for 20 ns;
    assert (Y="10") report "Test failed on Sel=1 - B not changed";
    ...
    wait;
  end process stimuli;
end architecture TestBenchArch;
```

Час = 0 ns; виходу ***Y*** має бути присвоєне значення "*XX*", оскільки сигнал ***Sel*** встановлений в '*X*'. Будь-яке інше значення неправильне і має бути повідомлене як помилка тестування.

Час = 40 ns + *MuxDel*; виходу ***Y*** має бути присвоєне значення ***A***, тобто "*00*" . Будь-яке інше значення неправльне і має бути повідомлене як помилка тестування.

Час = 60 ns + *MuxDel*; вхід ***Sel*** не змінився, але змінилось значення ***A***, отже, відповідно, має змітись значення ***Y*** на "*00*". Будь-яке інше значення неправльне і має бути повідомлене як помилка тестування.

Час = 80 ns + *MuxDel*; вхід ***Sel*** змінився на '1', отже ***Mux*** має переключитись на ***B***. Значення ***B*** рівне "11", і це значення має з'явитись на виході ***Y***. Будь-яке інше значення неправльне і має бути повідомлене як помилка тестування.

Час = 100 ns + *MuxDel*; вхід ***Sel*** не змінився, але змінилось значення ***B***, отже, відповідно, має змітись значення ***Y*** на "*10*". Будь-яке інше значення неправльне і має бути повідомлене як помилка тестування.

Рис.9.6.

10. Синтезована підмножина VHDL.

Відмінності між системами моделювання і системами синтезу

Між системами моделювання та синтезу існують суттєві відмінності:

1. Система моделювання не перетворює проектну інформацію – специфікації майбутньої системи, записаної на мові VHDL. Система синтезу перетворює проектну інформацію – вихідний алгоритмічний опис системи у функціонально-структурний опис логічної схеми – це специфікації на етапі логічного проектування.
2. Система моделювання дозволяє працювати із всіма конструкціями мови VHDL, в той час як системи синтезу можуть обробляти "синтезовану" підмножину конструкцій мови VHDL.
3. Для моделювання найчастіше використовується VHDL-опис тестового стенду. Для синтезу це неприпустимо, на синтез повинен надходити опис, що має входи і виходи.
4. Розробляючи VHDL-код, призначений для синтезу, проектувальник повинен чітко уявляти які логічні підсхеми будуть заміняти ті або інші конструкції мови VHDL. Врахування цих особливостей приводить до більш ефективних результуючих логічних схем.

Синтезована підмножина VHDL

Для того, щоби скористатись системою синтезу (програмними засобами синтезу схем, що входять в САПР), яка дозволяє отримати опис апаратури (опис логічної схеми), що реалізує потрібну поведінку проектованої цифрової системи, проектувальник повинен написати VHDL-код на синтезованій підмножині цієї мови.

Засоби САПР розробляються різними організаціями (фірмами). Різні системи синтезу можуть працювати з різними синтезованими підмножинами. Тому поняття синтезованої підмножини не є суворим. Більш суворо можна говорити про синтезовану підмножину для деякої конкретної системи синтезу, тобто поняття синтезованої підмножини тісно пов'язане із системою синтезу.

Визначення синтезованої підмножини, як правило, проводиться через перерахування обмежень конструкцій мови, що не входять в синтезовану підмножину, тобто синтезована підмножина визначається через вказування заборонених конструкцій мови.

Нижче навено достатньо загальні обмеження синтезованої підмножини VHDL.

1. *Операції над типом **real** не підтримуються при синтезі* – якщо у VHDL-коді використовується тип ***real***, то система синтезу видасть помилку і схема не буде побудована.
2. *Операції над файлами не підтримуються при синтезі.*
3. *Атрибути **'behavior**, **'structure**, **'last_event**, **'last_active**, **'transaction** не підтримуються при синтезі*.
4. *Глобальні неконстантні сигнали (задекларовані в пакеті) не підтримуються при синтезі.*
5. *Ключове слово **after** ігнорується при синтезі* – синтез відбувається таким чином, ніби затримки ***after*** в присвоєнні немає.
6. *Ключове слово **transport** ігнорується при синтезі.*
7. *Обмеження на ініціалізацію значень*:
   - *не підтримуються ініціальні значення сигналів в розділі декларації сигналів,*
   - *не підтримуються ініціальні значення портів **in**, **out**, **inout** в інтерфейсі (**entity**),*
   - *в процесі не підтримуються ініціальні значення змінних в розділі декларації змінних.*
8. *Обмеження на використання оператора циклу* – цикли підтримуються, якщо циклові змінні обмежені константами.
9. *Обмеження на атрибути виділення фронтів сигналів:*
   - *атрибути **'event**, **'stable** можуть використовуватись тільки для того, щоби визначити додатній та від'ємний фронти сигналів,*
   - *вирази, що задають фронти сигналів, можуть використовуватись тільки як умови.*
10. *Обмеження на використання оператора **wait**:*
    - *вираз в умові **until** оператора **wait** повинен визначати додатній або від'ємний фронт,*

- *багатократні оператори **wait** допускаються в операторі процесу, якщо всі умови очікування однакові.*

11. *Для опису вхідних та вихідних портів синтезованих схем використовуються лише типи **std_logic** і **std_logic_vector**.*
12. *Обмеження на синтезовані оператори і конструкції:*
    - *оператори процесу підтримуються при синтезі,*
    - *всі логічні оператори схемно реалізуються,*
    - *всі арифметичні оператори схемно реалізуються,*
    - *оператор **if** схемно реалізується,*
    - *оператор **case** схемно реалізується,*
    - *оператор циклу **loop** схемно реалізується,*
    - *оператор повідомлення **assert** ігнорується при синтезі,*
    - *при схемній реалізації операторів зсуву потрібно використовувати пакет **EXEMPLAR**,*
    - *оператор генерації **generate** підтримується при синтезі,*
    - *оператор реалізації компонента (створення екзепляра компонента) підтримується при синтезі,*
    - *перевантаження операторів схемно реалізується при синтезі,*
    - *використання змінних допускається в синтезованому коді,*
    - *якщо в інтерфейсі схеми є вихідні порти типу **std_logic**, що не використовуються, то для таких портів в синтезовану схему встановлюються тристабільні елементи,*
    - *механізм передачі параметрів **generic** підтримується при синтезі.*

II. Мова Verilog

11. Вступ у Verilog.

Коротка історія створення

Мова Verilog була розроблена в 1984-85 рр. Філіпом Мурбі (Philip Moorby).

Verilog ввійшов в систему автоматизації вентильного проектування (Gateway Design Automation), яку пізніше придбала фірма Cadence Design Systems. З 1990 р. Cadence відкрила цю мову для публічного використання, після чого вона була стандартизована IEEE (IEEE Standard 1364).

Verilog від початку був дуже популярний в США. За статистикою в 1993 р. 85% проектів з створення ВІС розроблялось на Verilog. В 90х рр. в створення засобів Verilog було вкладено більше мільярда доларів.

Базові поняття мови

Модуль у Verilog

Базовим блоком у Verilog є модуль. Модулі можуть описувати системи різної складності. Структура модуля наведена на рис.11.1.

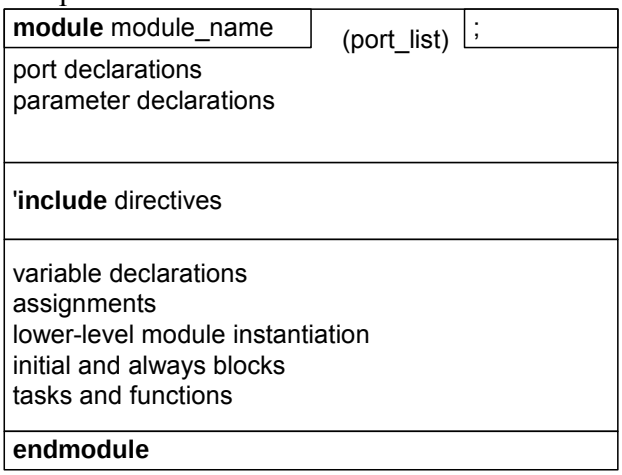


Рис.11.1

Модуль розміщується між ключовими словами ***module*** і ***endmodule***. Всередині модуля знаходяться три елементи: *інтерфейс* (містить порти та декларації параметрів), *тіло* (специфікація внутрішньої частини модуля), необов'язкові *додатки*, які включаються за допомогою директиви компілятора ***'include***.

Ідентифікатори та коментарі у Verilog

Ідентифікатор у Verilog має відповідати наступним правилам:

- має складатись з літер, цифр, символів долара '$' та підкреслення '_';
- має починатись з літери або символа підкреслення;
- не містити пробілів;
- літери верхнього та нижнього регістрів розрізняються;
- не має співпадати із зарезервованими словами.

Коментарі у Verilog аналогічні коментарям у мові С:

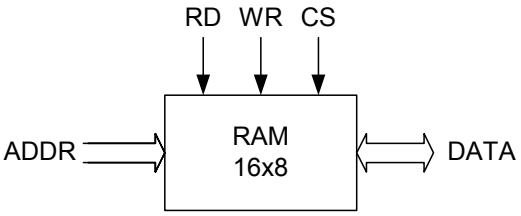
- блочні /*.. */
- рядкові //.

Інтерфейс модуля Verilog

Інтерфейс модуля складається з двох частин:

- списку портів, що містить тільки імена портів в дужках після імені модуля,
- декларацій портів – перша група декларацій модуля – задає напрямок потоку даних для кожного порта та його ширину.

На рис.11.2 наведено приклад інтерфейсу модуля.




```
module RAM16x8(DATA,ADDR,RD,WR,CS);
inout [7:0] DATA;
input [3:0] ADDR;
input RD;
input WR;
input CS;
...
endmodule
```

Рис.11.2

Стилі опису у Verilog

Для опису схем у Verilog (як і у VHDL) застосовуються три стиля:

1. *структурний* стиль – використовуються примітиви та низькорівневі реалізації модулів;
2. стиль опису *потоків даних* – задання вихідних сигналів в термінах перетворень вхідних сигналів;
3. *поведінковий* стиль – очікувана поведінка схеми описується у вигляді алгоритму.

Кожному з цих стилів опису відповідають певні конструкції мови, які будуть розглянуті далі.

12. Сигнали у Verilog.

Вступ до сигналів

4-значна логіка

У Verilog використовується 4 значна логіка:

`0`	**Логічний 0** або **хибна умова**. '0' - це одне з двох взаємно комплементарних булевих логічних значень.
`1`	**Логічна 1** або **істинна умова**. '1' - це одне з двох взаємно комплементарних булевих логічних значень.
`x`	**Невизначене логічне значення**. 'X' інтерпретується як '0', або '1', або 'Z', або зміна стану.
`z`	**Високий імпеданс**. Фізично 'Z' еквівалентно відключенню джерела сигналу, але Verilog-симулятори трактують його як 'X'.

Рис.12.1

При цьому, незважаючи на те, що Verilog є чутливим до регістру, для значень "невизначено" ('x' або 'X') та "високий імпеданс" ('z' або 'Z') можуть використовуватись як великі, так і маленькі літери.

Логічні операції над 4-значними сигналами

Таблиці істинності для усіх 8-ми основних логічних операторів з врахуванням 4-значної логіки Verilog мають вигляд:

buf	
вхід	**вихід**
0	0
1	1
x	x
z	x

Передає сигнал з входу на вихід без змін. Тільки "високий імпеданс" перетворюється у "невизначений".

not	
вхід	**вихід**
0	1
1	0
x	x
z	x

Інвертує значення '0' і '1', але 'z' і 'x' передаються на вихід як "невизначені".

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Якщо на одному з входів '0', результат рівний '0' незалежно від інших входів. Якщо на одному з входів 'x', результат "невизначений" (якщо на інших входах немає '0').

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

Якщо на одному з входів '0', результат рівний '1' незалежно від інших входів. В інших випадках (крім '1' **nand** '1') результат "невизначений".

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Якщо на одному з входів '1', результат рівний '1' незалежно від інших входів. В інших випадках (крім '0' **or** '0') результат "невизначений".

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

Якщо на одному з входів '1', результат рівний '0' незалежно від інших входів. В інших випадках (крім '0' **nor** '0') результат "невизначений".

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Якщо на одному з входів 'x' або 'z', результат рівний 'x'. В інших випадках поведінка повністю відповідає двозначному XOR.

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Якщо на одному з входів 'x' або 'z', результат рівний 'x'. В інших випадках поведінка повністю відповідає двозначному XNOR.

Рис.12.2.

Сигнали у Verilog

Класи сигналів

Кожний сигнал у Verilog належить до одного з двох класів: це або *провідник* (*net*), або *регістр* (*register*).

Провідники представляють фізичні з'єднання між апаратними елементами. Вони не мають ніякої запам'ятовуючої ємності і їх значення або визначаються значеннями їх драйверів (джерелами сигналів), або рівні високому імпедансу (коли провідник не підключений ні до якого драйвера).

Регістри здатні зберігати значення, навіть коли вони відключені. Попередньо присвоєне значення зберігається до тих пір, поки не буде присвоєне нове значення. В результаті *регістри* відіграють ту ж саму роль, що і змінні в мовах програмування.

Концепція регістрів у Verilog відрізняється від цифрових регістрів, які будуються на основі синхронних тригерів. Регістри у Verilog не передбачають ніякого синхронізуючого сигналу.

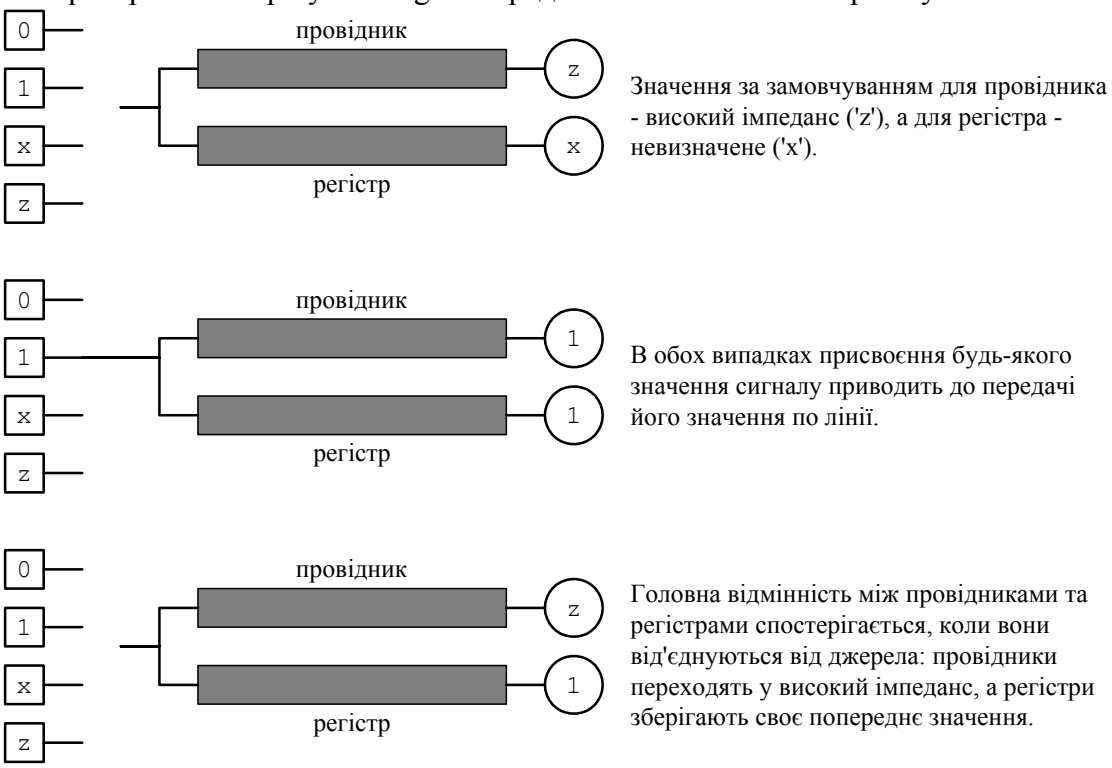
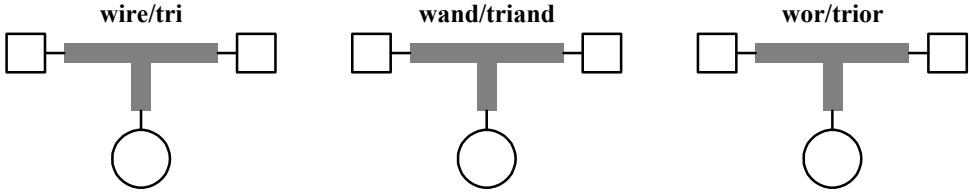


Рис.12.3.

Провідники

Більшість сигналів в системі можуть бути віднесені до *провідників*, оскільки вони з'єднують пристрої і керуються виходами пристроїв. '*Net*' не є ключовим словом у Verilog. Це ім'я класу сигналів, який складається з декількох типів:

- ***wire*** і ***tri*** – ці сигнали використовуються найчастіше; два різних імені використовуються тільки для наочності: ***wire*** прийнято використовувати для однодрайверних провідників, а ***tri*** – для провідників з багатьма джерелами;
- ***wand/triand*** і ***wor/trior*** – представляють провідники з монтажними логічними операціями кон'юнкції та диз'юнкції; аналогічно, різниця між версіями ***w-*** і ***tri-*** використовується лише для документування;
- ***supply0***, ***supply1***, ***tri0***, ***tri1***, ***trireg*** – ці провідники мають спеціальні властивості для спрощення специфікацій низького рівня: транзисторів або орієнтованих на технологію вентилів.



wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Рис.12.4.

Специфікація сигналу

Перш ніж використовувати сигнали у Verilog-специфікаціях, їх необхідно задекларувати. Існує дві групи сигналів: зовнішні (інтерфейсні) та внутрішні.

Синтаксис декларації сигналу починається з ключового слова, що визначає тип сигналу, за яким вказується ім'я сигналу:

- в якості типу сигналу можуть задаватись ключові слова: ***wire***, ***tri***, ***tri1***, ***supply0***, ***wand***, ***triand***, ***tri0***, ***supply1***, ***wor***, ***trior***, ***trireg***, ***reg***, ***time***, ***integer***, ***real***, ***realtime*** – всі літери нижнього регістру;
- правила задання імен сигналів:
  - ім'я сигналу не може починатись з символа '$',
  - ім'я сигналу не може містити пробілів,
  - в якості імені не можна використовувати зарезервовані слова, такі як: ***wire***, ***tri***, ***tri1***, ***supply0***, ***wand***, ***triand***, ***tri0***, ***supply1***, ***wor***, ***trior***, ***trireg***, ***reg***, ***time***, ***integer***, ***real***, ***realtime***.

Декілька сигналів одного типу можуть декларуватись разом. В цьому випадку їх ідентифікатори повинні розділятись комами. Декларація сигналу завершується крапкою з комою.

Синтаксис Verilog дозволяє декларувати внутрішні сигнали в довільному місці тіла модуля.

Специфікація вектора

Вектор у Verilog не є спеціальним типом сигналу. Навіть скалярний сигнал може бути представлений як спеціальний випадок вектора, що складається тільки з однієї лінії, і в якому найстарший розряд (MSB) рівний наймолодшому (LSB).

Якщо вектор складається з декількох бітів, його декларація повинна описувати їх індекси. Verilog забезпечує гнучке індексування розрядів вектора: можна використовувати від'ємні, 0-і та додатні значення. Крім того, індекс MSB може бути меншим, ніж індекс LSB. Але найлівіший розряд завжди є найстаршим.

Most Significant Bit (MSB). Значення, що задається тут, визначає індекс найлівішого біта (традиційно - найстаршого) у векторі. Воно може задаватись як константний вираз (тобто такий, що отримує значення на протязі компіляції), а також може мати додатнє, від'ємне або 0 значення. Воно може бути більшим, рівним, або меншим, ніж LSB.

Least Significant Bit (LSB). Значення, що задається тут, визначає індекс найправішого біта (традиційно - наймолодшого) у векторі. Воно може задаватись як константний вираз (тобто такий, що отримує значення на протязі компіляції), а також може мати додатнє, від'ємне або 0 значення. Воно може бути більшим, рівним, або меншим, ніж MSB.

```
wire [ ] : [ ] Bus;
```

```
wire [  0 : 0  ] Bus;
```
MSB = **LSB**. Коли індекси найстаршого та наймолодшого бітів однакові, вектор де факто стає скаляром і може декларуватись просто як **wire Bus**. В будь-якому випадку до нього можна звертатись просто як **Bus** (без індексів).

```
wire [  7 : 0  ] Bus;
```
MSB > **LSB**. В цій декларації індекс MSB більше, ніж LSB. Така ситуація відповідає традиційному підходу, прийнятому при проектуванні цифрових систем.

```
wire [  0 : 7  ] Bus;
```
MSB < **LSB**. У деклараціях векторів Verilog MSB завжди знаходиться зліва, отже, навіть якщо його індекс менше, ніж LSB, це цілком допускається. описані індекси відносяться тільки до індивідуальних бітів.

Рис.12.5.

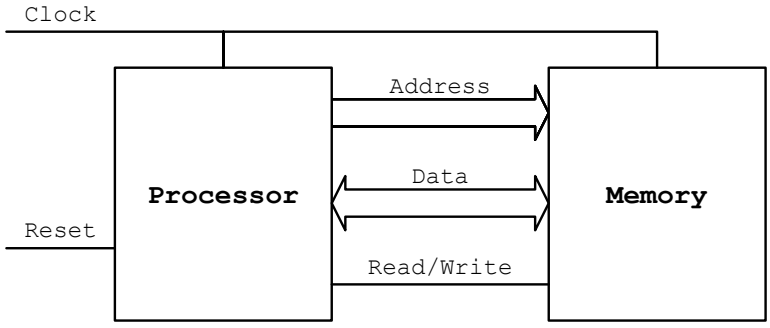
Зовнішні сигнали

Порти модуля

Всі інтерфейсні (зовнішні) сигнали модуля називаються портами модуля. Модуль взаємодіє із своїм середовищем через порти. Порт характеризується двома властивостями: шириною порта (скалярний одиничний біт або вектор) і напрямком передачі даних через порт (*in* або *out*).

Кожний порт може мати один з трьох напрямків:

- ***input*** – дані читаються модулем з середовища через *вхідні порти*; неможливо записувати такі порти із модуля;
- ***output*** – дані відправляються модулем у середовище через *вихідні порти*; читання модулем таких портів неможливе;
- ***inout*** – дані дозволяється як читати, так і писати; такі порти називаються двонапрямленими.




```
module Processor(Clock, Reset, Address, Data, Read_Write);
input Clock;
input Reset;
output Address;
inout Data;
output Read_Write;

module Memory(Clock, Address, Data, Read_Write);
input Clock;
input Address;
inout Data;
input Read_Write;
```

Clock. Сигнал *Clock* синхронізує процесор і пам'ять. Обидва модулі тільки читають цей сигнал, отже в обох випадках *Clock* описується як вхідний (**input**) порт.
Reset. Сигнал *Reset* надходить ззовні процесора і читається процесором. Він декларується як вхідний (***input***) порт в модулі *Processor*.
Address. Адреса звертання до пам'яті генерується процесором. Отже, цей сигнал декларується як вихідний (**output**) порт в модулі *Processor* і як вхідний (**input**) порт в модулі *Memory*.
Data. В залежності від значення сигналу *Read_Write*, дані формуються пам'яттю і читаються процесором, або навпаки. Отже, сигнал *Data* декларується в обох модулях як двонапрямлений (**inout**) порт.
Read_Write. Цей сигнал відправляється процесором до пам'яті і визначає читання або запис даних пам'яті процесором. Він декларується як вихідний (**output**) порт в модулі *Processor* і як вхідний (**input**) порт в модулі *Memory*.

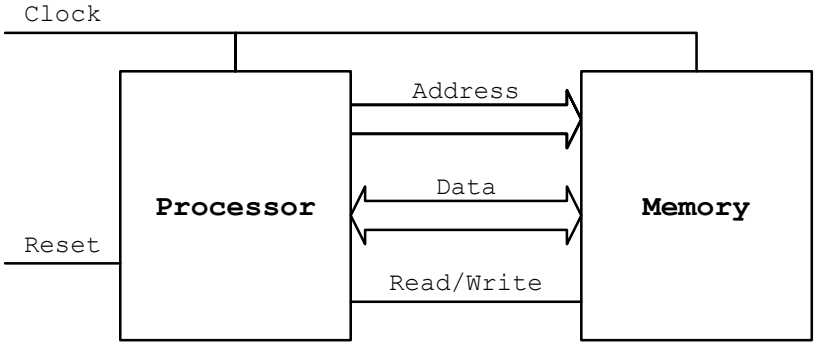
Рис.12.6.

Специфікація портів

Специфікація порта модуля складається з двох елементів:

- *Імені порта*, вказаного в *списку портів*, що слідує після імені модуля; список вказується в дужках і містить імена всіх портів, розділених комами. Він використовується при реалізації модуля в ієрархічних специфікаціях, тобто коли він використовується як компонент нижнього рівня.
- *Декларації порта*, що описує напрямок, розмір (ширину вектора) та ім'я порта. Декларації портів формують першу частину декларацій модуля і вказуються відразу після заголовку модуля. Декларації портів описуються аналогічно внутрішнім сигналам:
  keyword vector_range identifier;

Однак існує і суттєва відмінність – в специфікації сигналу використовується ключове слово, що описує тип сигналу, а в декларації портів – ключове слово, яке визначає напрямок порта (***input***, ***output*** або ***inout***).




```
module Processor(Clock, Reset, Address, Data, Read_Write);
input Clock;
input Reset;
output [19:0] Address;
inout [15:0] Data;
output Read_Write;
...
endmodule
```

```
module Memory(Clock, Address, Data, Read_Write);
input Clock;
input [19:0] Address;
inout [15:0] Data;
input Read_Write;
...
endmodule
```

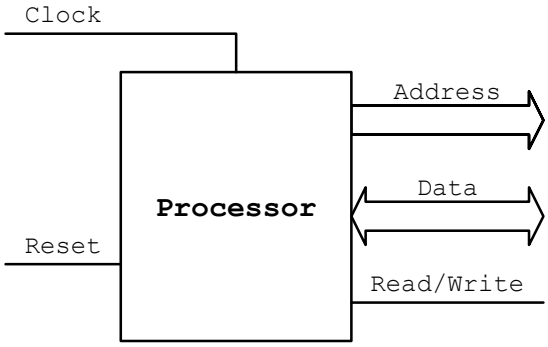
Рис.12.7.

Регістровані виходи

Оскільки порти можуть бути тільки *вхідними* або *вихідними*, стандарт Verilog вважає всі порти провідниками типу ***wire***.

Однак в деяких випадках типу ***wire*** недостатньо для відповідного опису функціонування системи, оскільки сигнали провідників потрібно підтримувати весь час. В протилежному випадку вони відразу переходять у високий імпеданс. Така поведінка небажана для більшості послідовнісних схем, де вихід має утримувати постійне значення до наступного фронту тактового імпульса, незалежно від станів входів. Це може бути легко реалізовано при використанні типу виходів ***reg*** замість ***wire***. Але, оскільки неможливо задавати тип порта, використовується інший шлях.

Тільки вихідні порти можуть бути регістрами. Ключове слово ***reg*** використовується на початку декларації. Тип *регістр* вихідного порта вимагає додаткового декларування сигналу.




```
module Processor(Clock, Reset, Address, Data, Read_Write);
input Clock;
input Reset;
output [19:0] Address;
inout [15:0] Data;
output Read_Write;
...
reg [19:0] Address;
reg Read_Write;
endmodule
```

Вхідні порти не можуть мати тип *регістр* і тому декларація цих портів не змінюється.
Вихідні порти можуть мати тип *регістр*, якщо додати нову додаткову декларацію для порта, яка вказує, що цей порт є регістром.
Двонапрямлені порти не можуть мати тип *регістр* і тому декларація цих портів не змінюється.

Рис.12.8.

13. Структурні описи у Verilog.

Примітиви Verilog

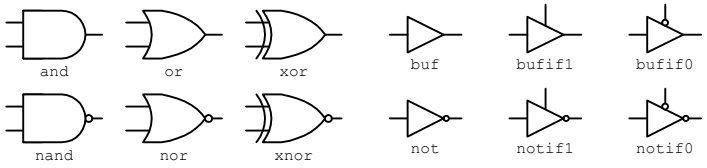
Визначені логічні примітиви

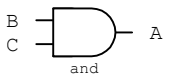
Verilog пропонує набір 12 логічних примітивів, що можна використовувати для структурних описів схем. Логічні примітиви розбиті на 3 групи: багатовходові, багатовихідні, та тристабільні.

Група багатовходових вентилів містить 6 логічних вентилів: **and**, **nand**, **nor**, **or**, **xnor** і **xor**. Кожний з них виконує логічні операції з довільною кількістю входів.

Група багатовихідних вентилів складається з двох вентилів: **buf** і **not**. Кожний з них має одних вхід і багато виходів.

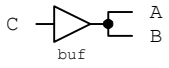
Група тристабільних вентилів містить буфери з трьома станами, які керуються '0' або '1' (**bufif0** і **bufif1**) та інвертори з трьома станами, які керуються '0' або '1' (**notif0** і **notif1**). При відсутності дозволу на виході тристабільного вентиля знаходиться 'z'. Ці вентилі мають тільки 1 вхід, 1 вихід і 1 керуючий вхід.





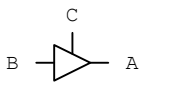

```
wire A, B, C;
and (A, B, C);
```

Перший аргумент багатовходового вентиля - завжди *вихід*. Другий і всі наступні аргументи - його *входи*. Мінімальна кількість аргументів - один вихід і один вхід.




```
wire A, B, C;
buf (A, B, C);
```

Останній аргумент багатовихідного вентиля - завжди *вхід*. Всі аргументи, крім останнього - його *виходи*. Мінімальна кількість аргументів - один вхід і один вихід.




```
wire A, B, C;
bufif1 (A, B, C);
```

Перший аргумент тристабільного вентиля - його *вихід*. Другий аргумент - *вхід*. Третій аргумент - *керуючий вхід*. Усі три аргументи є обов'язковими.

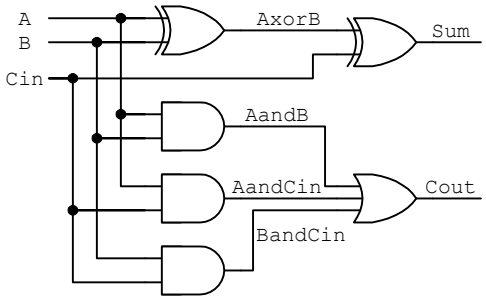
Рис.13.1.

Побудова описів на основі примітивів

Для побудови специфікації на основі примітивів необхідно описати декларації реалізації вентилів та декларації з'єднань сигналів.

Під реалізацією примітива розуміють використання вентиля із бібліотеки і підключення сигналів до його входів та виходів. Вона складається як мінімум з двох елементів: імені примітива та списку сигналів, що називаються виводами.

Крім того, реалізація примітиву може містити додаткову інформацію: часова затримка, унікальне ім'я окремого вентиля, діапазон масиву реалізацій та ін.



Визначення модуля
Декларації портів
Декларація внутрішнього сигналу
Перший вентиль **xor**
Перший вентиль **and**
Вентиль **xor** суми
Другий вентиль **and**
Третій вентиль **and**
Вихідний вентиль **or**

```
module FullAdd_1Bit (Sum,Cout,A,B,Cin);
output Sum,Cout;
input A,B,Cin;
wire AxorB,AandB,AandCin,BandCin;

xor #5 HalfSum (AxorB,A,B);
and #4 (AandB,A,B);
xor FullSum(Sum,AxorB,Cin);
and #4 And_2 (AandCin,A,Cin);
and #4 And_3(BandCin,B,Cin);
or (Cout,AandB,AandCin,BandCin);

endmodule
```

Рис.13.2.

Визначення модуля. Містить ім'я модуля та сигнали інтерфейсу. Слід пам'ятати, що у Verilog спочатку вказуються виходи, а потім входи.

Декларації портів. Усі порти, вказані в списку портів повинні бути задекларовані в заголовку модуля. Порядок декларацій не обов'язково має відповідати порядку портів в списку портів.

Декларація внутрішнього сигналу. При використанні структурної специфікації повинні бути задекларовані усі сигнали, що з'єднують компоненти. В комбінаційних схемах вони завжди мають бути провідниками.

Перший вентиль *xor*. Цей вентиль описаний із затримкою 5 одиниць часу та іменем *HalfSum*. Його вхідними сигналами є *A* і *B*, а вихідним сигналом – *AxorB*.

Перший вентиль *and*. Verilog дозволяє реалізовувати примітиви без задання імен. Це дозволяється тільки для примітивів вентилів. Вентиль описаний із затримкою 4 одиниць часу. Його вхідними сигналами є *A* і *B*, а вихідним сигналом – *AandB*.

Вентиль *xor* суми. Цьому вентилю присвоєно ім'я, але не задана затримка. В такому випадку симулятор вважає, що операція виконується без затримки.

Другий та третій вентилі *and*. Як і перший вентиль ***xor***, ця реалізація задає затримку та ім'я. Звичайно імена вентилів відповідають їх нумерації на схемі і усі вентилі називаються подібно, наприклад *U1*, *U2*, *U3* і т.д. Присвоєння імен реалізаціям вентилів не є обов'язковим, але рекомендується.

Вихідний вентиль *or*. Цей примітив заданий найкоротшою формою: тільки назва примітиву та список виводів.

Примітиви, визначені користувачем

Примітиви, визначені користувачем

Verilog дозволяє описувати і використовувати примітиви, визначені користувачем (User-Define Primitives – UDP). Такі примітиви можуть бути як комбінаційними, так і послідовнісними і в загальному просто представляють більш складні пристрої, ніж вентилі. Вони відрізняються від модулів як структурою, так і шляхом їх реалізації. UDP може мати тільки один вихід і як входи, так і виходи не можуть бути векторами. Їх можна представити як конструкції мови, розташовані між примітивами і модулями.

UDP має спеціальну структуру і завжди визначається в термінах таблиці істинності. Таблиці істинності для комбінаційних та послідовнісних схем мають різні форми.

```
primitive UDP_name    (port_list)    ;
port declarations
UDP initialization
truth- or state table
endprimitive
```

Визначення примітива. Декларація примітива починається з ключового слова **primitive**, після якого вказується ім'я примітива і список виводів (портів). Декларація завершується ключовим словом **endprimitive**.

Декларації виводів. Аналогічно портам модуля, виводи UDP мають бути задекларовані в списку заголовку примітива та нижче - із заданням напрямку. Список виводів завжди починається з одного і тільки одного виходу, за яким слідують входи. Порядок не обов'язково зберігати в деклараціях. Виходи послідовнісних UDP повинні бути регістровані. Не дозволяються двонаправлені або векторні сигнали в UDP.

Ініціалізація. Необов'язкове поле, яке може використовуватись тільки в деклараціях послідовнісних UDP. Вона починається з ключового слова **initial**, після якого ініційоване значення присвоюється виходу. В комбінаційних UDP використання ініціалізації не дозволяється.

Таблиця істиності або станів. Таблиця, що вказується між ключовими словами **table** та **endtable**, задає всі можливі комбінації входів та відповідні значення виходу. У випадку послідовнісних UDP кожний рядок таблиці містить біжучий та наступний стан. Якщо деякі з можливих комбінацій не вказані, на виході в такому випадку буде значення 'x'. Входи в кожному рядку таблиці задаються в тому самому порядку, що і виводи в списку заголовку примітива.

```
primitive Mux2to1 (Out,Sel,In0,In1) ;
output Out;
input Sel,In0,In1;
// для комбінаційних примітивів
// ініціалізації відсутні
table
  //  Sel   In0   In1    :  Out
       0     0     ?     :   0;
       0     1     ?     :   1;
       1     ?     0     :   0;
       1     ?     1     :   1;
       x     ?     ?     :   x;
endtable
endprimitive
```

Приклад комбінаційної схеми - 2-х входовий мультиплексор. Значення '?' в таблиці істиності означає "довільне значення". Таблиця не містить випадків, коли на вході Sel '0' або '1', а на активному вході (In0 або In1 відповідно) 'x'. За визначенням незазначені комбінації приводять до 'x' на виході, що є коректним значенням в даному випадку. Випадок Sel='x' не розглядається з тієї ж причини.

```
primitive TTF (Q,Clk,Clr) ;
output Q; reg Q;
input Clr,Clk;
initial
  Q = 0;
table
  // Clk Clr : Q : Q+
      ?   1  : ? : 0 ; // асинхронний скид
      r   0  : 0 : 1 ; // переключення по
      r   0  : 1 : 0 ; // додатньому фронту Clk
      f   0  : ? : - ; // ігнорування від'ємного фронту Clk
      ?   f  : ? : 0 ; // ігнорування від'ємного фронту Clk
endtable
endprimitive
```

Приклад послідовнісної схеми - Т-тригер. Цей примітив ініціалізується '0' конструкцією **initial**. Таблиця містить три групи колонок: входи, біжучий стан та наступний стан виходу. Додатній та від'ємний фронти описуються відповідними скороченнями (*rising* - r, *falling* - f). Символ '-' в якості наступного стану виходу означає "без змін").

Рис.13.3.

Комбінаційні UDP

Визначення комбінаційного UDP містить декларацію портів з одним виходом і одним або більше входами, та таблицю істинності, що описує функціонування примітива.

Порядок входів в списку портів дуже важливий для таблиці істинності. Перший сигнал в списку завжди є виходом примітиву, входи вказуються після нього.

В таблиці істинності використовується зворотня послідовність – кожний рядок таблиці починається із значень входів, після яких вказується двокрапка і, в кінці, значення виходу, що відповідає комбінації входів. Кожний рядок завершується крапкою з комою.

Слід пам’ятати, що порядок входів в таблиці **повинен** точно відповідати їх порядку в списку портів заголовку примітива. Змінювати цей порядок всередині таблиці забороняється.

Кожна комбінація входів, що приводить до визначеного значення на виході повинна бути описана. Для неописаних комбінацій вихід приймає значення ‘x’.

Крім явних значень (‘0’, ‘1’ або ‘x’ – ‘z’ в UDP не дозволяється), входи можуть також бути описані з використанням скорочених нотацій: символ ‘?’ заміняє довільне значення (‘0’, ‘1’ або ‘x’), а ‘b’ заміняє ‘0’ або ‘1’.

```
primitive Mux2to1 (Out,Sel,In0,In1);
output Out;
input Sel,In0,In1;
table
  //  Sel   In0   In1   :  Out
  //  ------------------------
       0     0     0    :   0;
       0     0     1    :   1;
       0     0     x    :   0;
       0     1     0    :   1;
       0     1     1    :   1;
       0     1     x    :   1;
       0     x     0    :   x;
       0     x     1    :   x;
       0     x     x    :   x;
       1     0     0    :   0;
       1     1     0    :   0;
       1     x     0    :   0;
       1     0     1    :   1;
       1     1     1    :   1;
       1     x     1    :   1;
endtable
endprimitive
```

Мультиплексор 2to1 передає значення з входу *In0* на вихід *Out* при *Sel* = '0' і *In1* при *Sel* = '1'. В інших випадках виходу присвоюється "невідоме" значення.

```
primitive Mux2to1 (Out,Sel,In0,In1);
output Out;
input Sel,In0,In1;
table
  //  Sel   In0   In1   :  Out
  //  ------------------------

       0     0     ?    :   0;


       0     1     ?    :   1;


       0     x     ?    :   x;


       1     ?     0    :   0;


       1     ?     1    :   1;

endtable
endprimitive
```

Той самий результат можна отримати, якщо використати замість "ловільного значення" символ '?'. Однак це значення не можна використовувати для виходу - в яких випадках слід використовувати 'x'.

```
primitive Mux2to1 (Out,Sel,In0,In1);
output Out;
input Sel,In0,In1;
table
  //  Sel   In0   In1   :  Out
  //  ------------------------
       0     0     ?    :   0;
       0     1     ?    :   1;
       1     ?     0    :   0;
       1     ?     1    :   1;
endtable
endprimitive
```

Фінальним кроком є створення компактного коду. Рядок з 'x' на виходи видалений як зайвий. Це необов'язково робити, хоча деколи для покращення читабельності такі рядки варто залишати в коді.

Рис.13.4.

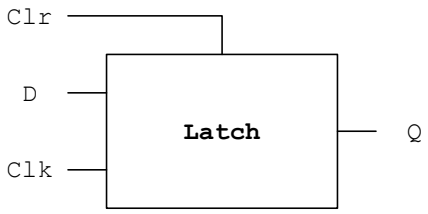
Послідовнісні UDP, чутливі до рівня

Якщо примітив є послідовнісною схемою, вихід, який представляє наступний стан, залежить не тільки від комбінації вхідних значень, але і від біжучого стану. В зв'язку з цим послідовнісні таблиці істинності містять додатковий стовпчик для біжучого значення.

Рядок містить всі входи, які закінчуються двокрапкою. Далі задається біжуче значення, після якого також стоїть двокрапка. В кінці рядка вказується очікуване значення наступного стану. Для деяких комбінацій результат нового стану не відрізняється від біжучого – в таких випадках використовується символ '-'. Він вказує , що зміни стану схеми не відбувається.

Є ще один важливий момент: послідовнісні примітиви можуть бути проініціалізовані. Для присвоєння виходу примітиву початкового значення необхідно вказати ключове слово ***initial***, після якого слідує присвоєння виходу.

```
primitive Dlatch (Q,Clr,D,C);
output Q; reg Q;
input Clr,D,C;
table
  //  Clr    D    Clk   :  Q  :  Q*
  //  ------------------------------
       1     ?     ?    :  ?  :  0 ;
       0     0     1    :  ?  :  0 ;
       0     1     1    :  ?  :  1 ;
       0     ?     0    :  ?  :  - ;
endtable
endprimitive
```



D-тригер із скидом. Коли *Clr* = '1', тригер скидається (*Q* стає рівним '0'). Коли *Clr* = '0' або неактивний і *Clk* = '1', тоді *Q* = *D*. Якщо *Clr* ='0' і *Clk* = '0', стан тригера зберігається незмінним.

Рис.13.5.

Послідовнісні UDP, чутливі до фронту

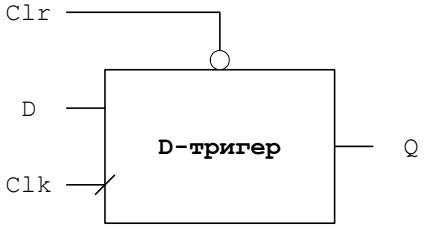
Багато послідовнісних схем реагує не на рівень вхідних сигналів, а на перепади (фронти) сигналів. Найкращим прикладом такої поведінки є синхронний тригер.

Фронти можуть бути описані двома шляхами: як пара значень в дужках, наприклад (01) для перепаду з 0 в 1, або символічно, з використанням символів:
'r' – для зростаючого фронту – те саме, що і (01),
'f' – для спадаючого фронту – те саме, що і (10),
'p' – для додатнього фронту – те саме, що і (01), (0x) або (x1),
'n' – для від'ємного фронту – те саме, що і (10), (1x) або (x0),
'*' – довільна зміна – те саме, що і (??).

Verilog дозволяє описувати тільки один перепад сигналу в одному рядку таблиці.

UDP, чутливі до фронту, можуть бути проініціалізовані так само, як і ті, що чутливі до рівня.

```
primitive Dlatch (Q,D,Clk,Clr);
output Q; reg Q;
input D,Clk,Clr;
initial
  Q = 0;
table
  //   D    Clk    Clr   :  Q  :  Q*
  //  ------------------------------
       ?     ?     0    :  ?  :  0 ;
       0     r     1    :  ?  :  0 ;
       1     r     1    :  ?  :  1 ;
       ?     n     1    :  ?  :  - ;
       *     ?     1    :  ?  :  - ;
       ?     ?     p    :  ?  :  - ;
endtable
endprimitive
```



D-тригер, що спрацьовує по фронту із асинхронним скидом. Коли *Clr* = '0', тригер скидається в '0'. В інших випадках в тригер записується значення з входу *D* при зростаючому фронті на вході *Clk*. Будь-які зміни на вході *D* при стабільному значенні на вході *Clk* ігноруються.

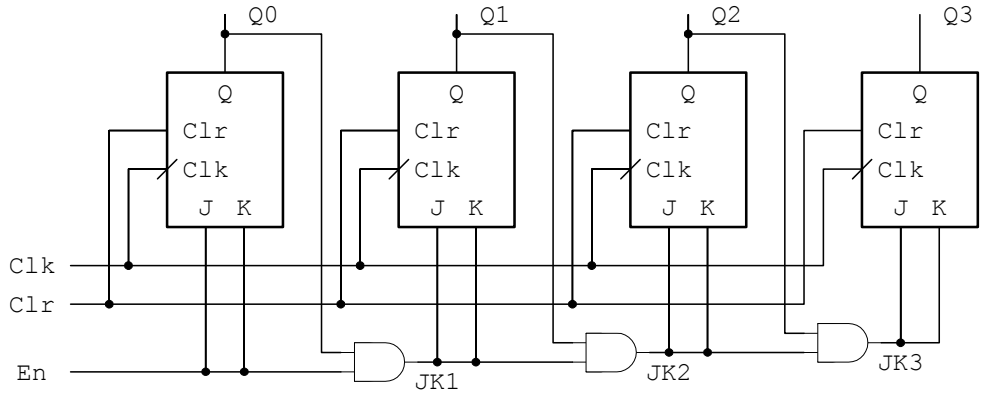
Рис.13.6.

Використання UDP

UDP використовуються так само, як і визначені примітиви вентилів. Єдина різниця полягає в тому, що UDP мають бути продекларовані.

Декларація UDP відноситься до того ж самого рівня ієрархії, що і модуль, отже всередині модулів (між ключовими словами ***module*** і ***endmodule***) описувати примітиви заборонено. Замість цього примітиви описуються або перед, або після модуля, в залежності від того, використовуються вони в даному модулі, чи ні. Вони можуть також описуватись в окремому файлі з використанням директиви компілятора ***include***. Ця опція дозволяє створювати бібліотеки UDP.

UDP суттєво розширюють невеликий набір визначених примітивів, особливо за рахунок послідовнісних примітивів. Однак такий примітив може мати тільки один вихід. Це обмеження заважає створювати такі базові схеми як повні суматори або регістри, які повинні бути описані як звичайні модулі.




```
module Count4En (Q,Clr,Clk,En);
output [3:0] Q;
wire [3:0] Q;
input Clr,Clk,En;
wire JK1,JK2,JK3;
JKMS Bit0 (Q[0],Clr,Clk,En,En);
JKMS Bit1 (Q[1],Clr,Clk,JK1,JK1);
JKMS Bit2 (Q[2],Clr,Clk,JK2,JK2);
JKMS Bit3 (Q[3],Clr,Clk,JK3,JK3);
and (JK1,Q[0],En);
and (JK2,Q[1],JK1);
and (JK3,Q[2],JK2);
endmodule
```

```
primitive JKMS (Qout,Clr,Clk,Jin,Kin);
output Qout; reg Qout;
input Clr,Clk,Jin,Kin;

table
  // ... відповідні декларації
endtable
endprimitive
```

4-розрядний лічильник із скидом та дозволом.
Сигнал асинхронного скиду *Clr* = '0' скидає лічильник. Якщо на вході дозволу *En* знаходиться 0 - рахування не відбувається. Коли на обох входах *Clr* і *En* присутня '1', кожний зростаючий фронт на вході *Clk* збільшує вміст лічильника, але новий стан з'являється на виході тільки після спадаючого фронта на вході *Clk*.

Рис.13.7.

Реалізації модулів

Реалізація модулів надає більшої гнучкості, ніж використання примітивів, оскільки модулі можуть мати довільну складність і довільну кількість портів. В примітивах не можна реалізовувати інші примітиви, але в модулях можна реалізовувати інші модулі, отже можна будувати проекти з багаторівневою ієрархією.

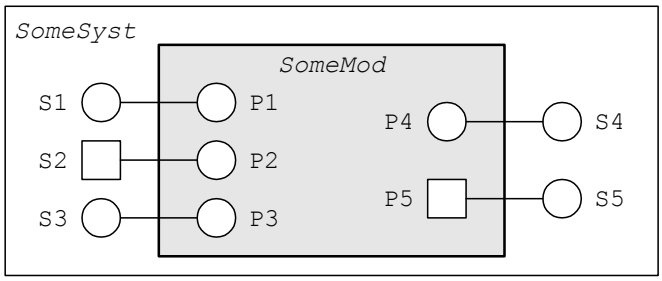
Є одне суттєве обмеження – модулі не можуть бути вкладені, тобто декларовані один всередині іншого.

Правила підключення портів

Кожний порт складається з двох з'єднаних частин: одна всередині модуля (внутрішня частина) та одна ззовні (зовнішня частина). Існують обмеження на їх типи:

- ***input*** порти повинні мати тип *net* всередині і ззовні можуть бути підключені до сигналів типу *net* або *reg*;
- ***inout*** порти повинні мати тип *net* як всередині, так і ззовні;
- ***output*** порти повинні мати тип *reg* або *net* всередині, а ззовні мають бути підключені до сигналів типу *net*.

У всіх трьох випадках порти можуть бути як скалярами, так і векторами.




```
module SomeSyst (...);
...
wire S1,S3,S4,S5;
reg S2;
...
PortExample SomeMod (S4,S5,S1,S2,S3);
endmodule
```

```
module PortExample (P4,P5,P1,P2,P3);
input P1,P2;
inout P3;
output P4,P5;
reg P5;
...
endmodule
```

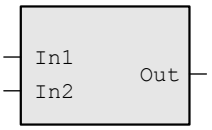
S1-P1. Сигнал типу ***net*** підключений до вхідного порта типу ***net***.
S2-P2. Сигнал типу ***reg*** підключений до вхідного порта типу ***net***.
S3-P3. Сигнал типу ***net*** підключений до двонапрямленого (***inout***) порта типу ***net***.
S4-P4. Сигнал типу ***net*** підключений до вихідного порта типу ***net***.
S5-P5. Сигнал типу ***net*** підключений до вихідного порта типу ***reg***.

Рис.13.8.

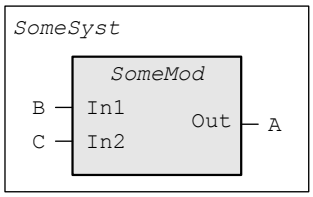
Впорядкований список портів

Асоціація між сигналами та портами *за впорядкованим списком* задається порядком, за яким порти описані в реалізованому модулі: перший сигнал в операторі реалізації призначається першому порту, другий сигнал – другому порту і т.д.

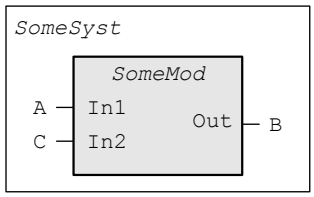
```
module PortEx (Out,In1,In2);
output Out;
input In1,In2;
...
endmodule
```



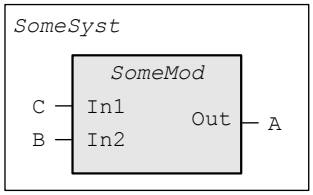

```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (A,B,C);
endmodule
```



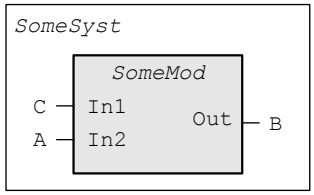

```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (B,A,C);
endmodule
```



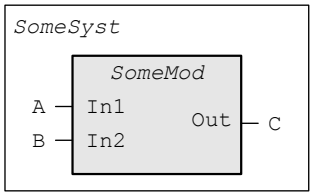

```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (A,C,B);
endmodule
```




```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (B,C,A);
endmodule
```




```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (C,A,B);
endmodule
```




```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (C,B,A);
endmodule
```

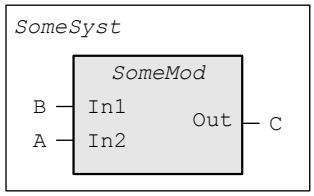


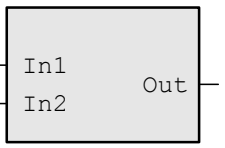
Рис.13.9.

Підключення портів за іменами

Для полегшення підключення до портів сигналів за іменами, Verilog підтримує *іменоване призначення портів*. В такому випадку кожний сигнал повинен мати ім'я, що асоціюється з відповідним портом. Це ім'я описується в дужках після імені порта. Перед іменем порта ставиться крапка.

Оскільки для кожного порта явно задається ім'я, немає необхідності зберігати той самий порядок, що і в декларації модуля. Іменоване призначення може задаватись в довільному порядку.

```
module PortEx (Out,In1,In2);
output Out;
input In1,In2;
...
endmodule
```




```
module SomeSyst (...);
...
wire A,B,C;
...
PortEx SomeMod (.In1(A),.In2(B),.Out(C));
endmodule
```

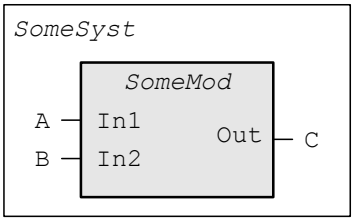


Рис.13.10.

Непідключені порти

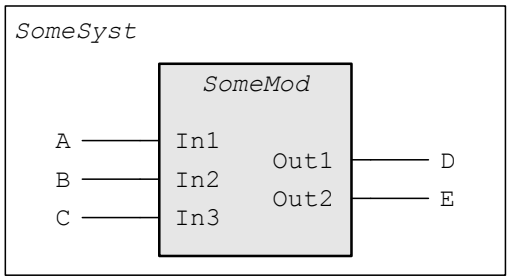
Крім можливості використання іменованих призначень портів, модулі мають ще одну властивість, якої не мають примітиви – будь-який порт може лишатись непідключеним.

Шлях визначення, що окремий порт є непідключеним, залежить від того, який тип підключення портів використовується:

- коли сигнали підключаються до портів за впорядкованим списком, кількість портів в списку дуже важлива, тому кожний непідключений порт представляється пробілом (оскільки кількість портів всередині списку портів визначається кількістю ком, пробіли можуть не ставитись);
- у випадку іменованих підключень непідключені порти просто пропускаються.

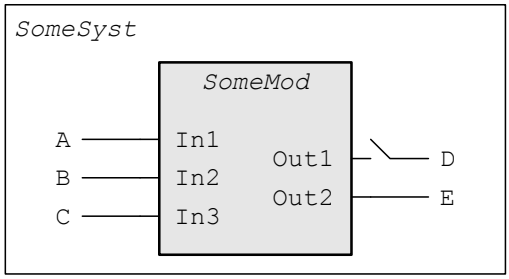
```
module PortEx (Out1,Out2,In1,In2,In3);
output Out1,Out2;
input In1,In2,In3;
...
endmodule
```

```
module SomeSyst1 (...);
...
wire A,B,C,D,E;
...
PortEx SomeMod (A,B,C,D,E);
...
endmodule
```




```
module SomeSyst1 (...);
...
wire A,B,C,D,E;
...
PortEx SomeMod (.In1(A),.In2(B),.In3(C),.Out1(D),.Out2(E));
...
endmodule
```

```
module SomeSyst1 (...);
...
wire A,B,C,D,E;
...
PortEx SomeMod (A,B,C,,E);
...
endmodule
```




```
module SomeSyst1 (...);
...
wire A,B,C,D,E;
...
PortEx SomeMod (.In1(A),.In2(B),.In3(C),.Out2(E));
...
endmodule
```

Рис.13.11.

14. Засоби мови Verilog.

Вирази

Операнди

Найпростішою і найочевиднішою категорією операндів є посилання на провідники або регістри за їх іменами.

Константи є іншою групою операндів. Verilog дозволяє використання констант різних типів, заданих в різних форматах. Константи не потрібно декларувати – вони можуть задаватись безпосередньо у виразі.

Для визначення векторів (як провідників, так і регістрів) використовуються біт-виборки (одиничний біт) та часткові виборки (декілька бітів). Частина багатовимірного масива, наприклад, пам'яті, також може використовуватись, але в такому випадку задаються повні слова.

Останньою категорією операндів є результат виклику функції (як системної, так і визначеної користувачем), якщо тип значення, що повертає функція, відповідає операнду.

`wire In1;`
`...`
`... In1 ...`
`...`
	Просте посилання. *In1* визначено як провідник, і на нього посилаються за тим самим ім'ям.

`real Radius;`
`...`
`... Radius ...`
`...`
	Просте посилання. Доступ до дійсного регістру *Radius* здійснюється безпосередньо за його іменем, формуючи простий операнд.

`reg [7:0] DataBus;`
`...`
`... DataBus[0] ...`
	Біт-виборка. *DataBus* задекларовано як 8-розрядний вектор, однак для деякіх операцій може знадобитись лише окремий біт. Такий біт можна вибрати, звернувшись за допомогою операнда біт-виборки до імені вектора та індексу необхідного біта.

`reg [7:0] DataBus;`
`...`
`... DataBus[7:4] ...`
`...`
	Часткова виборка. Використовується, коли необхідно звернутись до декількох (але не всіх) розрядів вектора. Необхідно задати лише ім'я вектора та діапазон бітів.

`...`
`... 1.3e7 ...`
`...`
	Значення константи. Дійсні числа можуть бути описані як значення констант, як в експоненціальній формі, так і з десятковою комою.

`...`
`... 127 ...`
`...`
	Значення константи. Це значення константи - ціле десяткове число - якщо база не задана, Verilog вважає його десятковим значенням.

`...`
`... 4'b1001 ...`
`...`
	Значення константи. Це двійкове значення 1001 із заданим розміром (4 біта).

`...`
`... CircleArea(Radius) ...`
`...`
	Виклик функції. Функція *CircleArea()* є визначеною користувачем і повертає дійсне значення. Отже, такий виклик функції може використовуватись як операнд лише там, де дозволяються дійсні операнди.

Рис.14.1.

Цілі константи

Цілі константи, тобто цілі числа, описані явно, є десятковими за замовчуванням і записуються в звичайному вигляді. Перед числом може знаходитись знак мінус, що вказує на його від'ємний знак. Це найпростіший, але не єдиний шлях задавання цілих чисел у Verilog.

Мова дозволяє задавати числа не тільки в десятковому форматі, але й в шістнадцятковому, вісімковому та двійковому форматах. Якщо число повинно бути записано не в десятковому форматі, на його початку повинен стояти апостроф і символ, що задає базу: *h* або *H* – шістнадцяткове, *o* або *O* – вісімкове, *b* або *B* – двійкове. Будь-яке значення, задане таким чином є цілим беззнаковим.

Розмір цілої константи визначається компілятором, але повинен бути не менше 32 біт. Дуже часто цього буває забагато і компілятор дозволяє використовувати *розмірні константи*, перед якими вказується додатнє ціле значення, яке задає кількість біт. Для вводу розмірних специфікацій десяткових чисел використовуються символи *d* або *D*, які задають базу.

Особливість	Опис	Приклади
Безрозмірні цілі	Якщо ціле не містить розміру, воно буде представлене кількістю біт, що визначається компілятором, але не менше 32. Цілі можуть бути задані в довільному форматі мови (двійковому, вісімковому, десятковому або шістнадцятковому) і, за виключенням десяткових, мають містити декларацію бази. Символ бази не чутливий до регістру. Цифри, що використовуються для чисел, мають відповідати базі: 0 та 1 для двійкових, 0..7 для вісімкових, 0..9 для десяткових і 0..9 та a..f для шістнадцяткових. Літери в шістнадцяткових значеннях нечутливі до регістру.	12 // база не задана -> десяткове число 'h12 // шітнадцяткове число, рівне десятковому 18 'ha0 // інше десяткове число 'b1001 // двійкове число a0 // помилка - 'a' не десяткова цифра
Розмірні цілі	Число, яке починається із задання бази, сприймається комілятором як розмірна декларація. Розмір задається беззнаковим десятковим числом і описує кількісьт біт, потрібних для представлення наступного числа. Якщо задана ціла константа вимагає менше розрядів, ніж задано, вона зліва розширюється 0-ми.	4'd4 // десяткове 4, що записується 4 бітами 8'b10011001 // 8-розрядне двійкове число 8'b1 // представляється як 00000001 8'h1 // представляється як 00000001
Від'ємні значення	Всі від'ємні значення представляються у Verilog 2-ма формами. Вставляти знак мінус між символом бази формату і значенням не дозволяється. Якщо число має бути від'ємним, знак мінус розташовується перед усією константою.	-10 // від'ємне 10 представляється 32 бітами -8'd10 // від'ємне 10 представляється 8 бітами, // еквівалентне -(8'd10)
Використання символу '?'	Позначення '?' може використовуватись в якості цифри замість 'z' для покращення читабельності у випадках, коли високий імпеданс не є строгою умовою. Символ '?' встановлює в 'z' 4 розряди шітнадцяткового числа, 3 розряди вісімкового числа та 1 розряд двійкового числа. Він не може використовуватись із десятковими числами.	8'h1? // еквівалентно 0001zzzz 2'b1? // еквівалентно 1z
Використання символів 'x' та 'z'	Цифри 'x' та 'z' задають значення "невідоме" та "високий імпеданс" відповідно. Вони можуть використовуватись в двійкових, вісімкових та шістнадцяткових числах (крім десяткових) і одиничні 'x' та 'z' розширюються до 4 розрядів шітнадцяткового числа, 3 розрядів вісімкового числа та 1 розряду двійкового числа	4'b01xx // останні 2 біти 4-розрядного числа невідомі 8'h1x // останні 4 біти 8-розрядного числа невідомі 'hx // 32-розрядне невідоме число 8'hxx // 8-розрядне невідоме число
Використання символу '_'	Символ підкреслення може використовуватись будь-де в числі, за виключенням першого символа. Він дозволяє покращити читабельність (оскільки пробіли всередині чисел не дозволяються).	16'b1001_1100_1110_0001 197_832_001
Розширення зліва	Розширення зліва відноситься до ситуації, коли розмір беззнакового числа менше, ніж заданий розмір константи. В такому випадку число записується справа, а розряди, яких не вистачає зліва, заповнюються 0-ми. Але, якщо найлівіший біт константи рівний 'x' або 'z', то біти зліва заповнюються цими значеннями.	8'b0 // еквівалентно 00000000 8'b1 // еквівалентно 00000001 8'bx // еквівалентно xxxxxxxx 16'hx10 // еквівалентно xxxxxxxx00010000

Рис.14.2.

Операнди біт-виборки та часткової виборки

Коли потрібно обробляти частину вектора провідника або регістра, використовуються операнди *виборки*. У випадку одиничних бітів використовують операнд *біт-виборки*. Якщо потрібно задати декілька сусідніх бітів, використовують операнд *часткової виборки*.

Синтаксис операнда біт-виборки дуже простий: ім'я вектора, після якого вказується індекс потрібного біта в квадратних дужках.

Часткова виборка використовується для задання діапазону бітів вектора. Діапазон задається подібно декларації векторів: [MSB:LSB]. Ліва границя повинна адресувати більш старший біт, ніж права.

Якщо хоча б один із заданих індексів виходить за межі діапазона вектора, або рівний 'x' або 'z', операнду автоматично буде присвоєне значення 'x'.

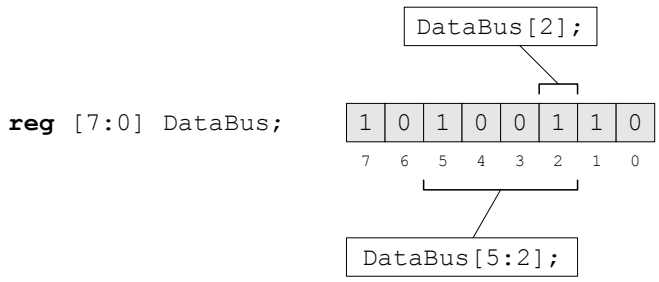


Рис.14.3.

Оператори

Арифметичні оператори

Verilog підтримує досить велику кількість операторів (32), до яких відносяться: *унарні* (з одним оператором), *бінарні* (з двома операторами) та *тернарні* (з трьома операторами). Ці оператори розбиті на декілька груп.

Арифметичні оператори зрозумілі навіть початківцю: '+' – додавання,'-' – віднімання, '*' – множення, '/' – ділення двох операндів. Перші два оператори ('+' і '-') можуть також використовуватись з одиничним операндом для визначення його знаку.

Крім чотирьох наведених операторів існує ще один, який називається "ділення за модулем" ('%'). Його результатом є залишок цілочисельного ділення двох операндів.

Важливим моментом є те, що арифметичні оператори по різному виконуються над операндами *регістрових* та *цілих* типів. В першому випадку операнди вважаються *беззнаковими*, а в другому – *знаковими*. В результаті при виконанні однакових операцій над однаковими операндами різних типів будуть отримані різні результати.

Тип операндів	Verilog - код	Опис
Ціле	**integer** intA; intA = -4'd12; intA / 3 = -4	*IntA* - ціле і розглядається як знакове число. Результат дорівнює -4, як і очікується. Однак, якщо цей результат присвоїти регістровій змінній, він буде забережений як 65532 - двійкове доповнення -4.
Регістр	**reg** [15:0] intA; intA = -4'd12; intA / 3 = 21841	Оскільки *IntA* - регістр, він зберігає від'ємні значення як двійкове доповнення. В цьому випадку -12 буде представлено як 65524. Це число, поділене на 3 дасть результат 21841, що і є результатом операції.
Беззнакова константа	-12 / 3 = -4	Коли -12 задається безпосередньо воно представляється цілим операндом. Результат дорівнює -4, як і очікується. Однак, якщо цей результат присвоїти регістровій змінній, він буде забережений як 65532 - двійкове доповнення -4.
Розмірна константа	-4'12 / 3 = 1431655761	Значення -12, задане як розмірна константа, приводить до такого неочікуваного результату, оскільки під час симуляції вона представляється 32-розрядним регістром.

Рис.14.4.

Оператори відношення

Оператори відношення порівнюють два операнди на нерівність. До них відносяться такі бінарні оператори: '<' (менше), '>' (більше), '<=' (менше або рівне), '>=' (більше або рівне). Перевірка на рівність відноситься до іншої групи операторів.

В усіх чотирьох випадках результат виразу є *0*, якщо результат відношення хибний і *1*, якщо результат відношення істинний. Якщо хоча б один з розрядів хоча б одного з операндів, що порівнюються, невідомий або рівний високому імпедансу, неможливо визначити результат і на виході отримаємо невідоме значення – *x*.

Якщо один з операндів містить менше розрядів, ніж інший, він буде розширений 0 зліва.

Оператори відношення мають нижчий пріоритет, ніж арифметичні оператори.

Лівий операнд	Правий операнд	<	>	<=	>=
0110	0110	0	0	1	1
0110	0101	0	1	0	1
0110	0xx0	x	x	x	x
0110	zzzz	x	x	x	x
0101	0110	1	0	1	0
0101	0101	0	0	1	1
0101	0xx0	x	x	x	x
0101	zzzz	x	x	x	x
0xx0	0110	x	x	x	x
0xx0	0101	x	x	x	x
0xx0	0xx0	x	x	x	x
0xx0	zzzz	x	x	x	x
zzzz	0110	x	x	x	x
zzzz	0101	x	x	x	x
zzzz	0xx0	x	x	x	x
zzzz	zzzz	x	x	x	x

Рис.14.5.

Оператори рівності

В більшості мов програмування та опису апаратних засобів *оператори рівності* не утворюють окремої групи, а відносяться до операторів відношення. Однак у Verilog вони розглядаються окремо. Причина полягає у 4-значній логіці. В залежності від потреби користувач може порівнювати два операнди точно, з використанням значень 'x' та 'z'. Для операторів відношення результат буде невизначений, якщо хоча б один операнд містить невизначене значення.

"Точні" оператори рівності називаються *фактичною рівністю* (*нерівністю*) і позначаються '===' та '!==' відповідно. Подібні операторам відношення *логічні рівності* (*нерівності*) позначаються '==' та '!='. Різницю в позначеннях легко запам'ятати: "більша точність вимагає додаткового символа '=' ".

Лівий операнд	Правий операнд	===	!==	==	!=
0110	0110	1	0	1	0
0110	0101	0	1	0	1
0110	0xx0	0	1	x	x
0110	zzzz	0	1	x	x
0101	0110	0	1	0	1
0101	0101	1	0	1	0
0101	0xx0	0	1	x	x
0101	zzzz	0	1	x	x
0xx0	0110	0	1	x	x
0xx0	0101	0	1	x	x
0xx0	0xx0	1	0	x	x
0xx0	zzzz	0	1	x	x
zzzz	0110	0	1	x	x
zzzz	0101	0	1	x	x
zzzz	0xx0	0	1	x	x
zzzz	zzzz	1	1	x	x

Рис.14.6.

Різні логічні оператори

Логічні оператори можуть бути джерелом помилок для початківців, оскільки вони служать для різних цілей, хоча позначаються однаковими символами. Стандарт Verilog поділяє ці оператори на три класи: *логічні оператори*, *побітові оператори* та *редукційні оператори*.

Логічні оператори отримують два операнди довільної довжини, виконують логічне порівняння і повертають однорозрядний результат, що не залежить від довжини операндів.

Побітові оператори отримують два операнди довільної довжини, виконують логічну операцію над кожною парою бітів (*i*-біт лівого операнда та *i*-біт правого операнда). Результат має довжину найдовшого операнда (коротший операнд розширюється зліва).

Редукційні оператори обробляють тільки один операнд (вектор) і формують однорозрядний результат, виконуючи логічну операцію над усіма бітами операнда.

Логічний and (&&)

для A = 0 і B = 2 A && B рівне 0

для A = 2'b00 і B = 2'b0x A && B рівне x

Логічний or (||)

для A = 0 і B = 2 A || B рівне 1

для A = 2'b00 і B = 2'b0x A || B рівне x

Логічний not (!)

для A = 0 !A рівне 1

для B = 2'b0x !B рівне x

Побітовий and (&)

для A = 4'b1010 і B = 4'b0011 A & B рівне 0010

для A = 4'b1010 і B = 4'b001x A & B рівне 001x

Побітовий or (|)

для A=4'b1010 і B=4'b0011 A | B рівне 1011

для A=4'b1010 і B=4'b001x A | B рівне 101x

Побітовий xor (^)

для A = 4'b1010 і B = 4'b0011 A ^ B рівне 1001

для A = 4'b1010 і B = 4'b001x A ^ B рівне 100x

Побітовий not (~)

для A = 4'b1010 ~A рівне 0101

для B = 4'b001x ~ B рівне 110x

Побітовий xnor (^~)

для A = 4'b1010 і B = 4'b0011 A ^~ B рівне 0110

для A = 4'b1010 і B = 4'b001x A ^~ B рівне 011x

Побітовий xnor (~^)

для A = 4'b1010 і B = 4'b0011 A ~^ B рівне 0110

для A = 4'b1010 і B = 4'b001x A ~^ B рівне 011x

Редукційний and (&)

для A = 4'b1010 & A рівне 0

для B = 4'b0001 & B рівне 0

для C = 4'b111x & C рівне x

Редукційний or (|)

для A = 4'b1010 | A рівне 1

для B = 4'b0001 | B рівне 1

для C = 4'b111x | C рівне x

Редукційний xor (^)

для A = 4'b1010 ^ A рівне 0

для B = 4'b0001 ^ B рівне 1

для C = 4'b111x ^ C рівне x

Редукційний xnor (^~)

для A = 4'b1010 ^~ A рівне 1

для B = 4'b0001 ^~ B рівне 0

для C = 4'b111x ^~ C рівне x

Редукційний nand (~&)

для A = 4'b1010 ~& A рівне 1

для B = 4'b0001 ~& B рівне 1

для C = 4'b111x ~& C рівне x

Редукційний nor (~|)

для A = 4'b1010 ~| A рівне 0

для B = 4'b0001 ~| B рівне 0

для C = 4'b111x ~| C рівне x

Редукційний xnor (~^)

для A = 4'b1010 ~^ A рівне 1

для B = 4'b0001 ~^ B рівне 0

для C = 4'b111x ~^ C рівне x

Рис.14.7.

Оператори зсуву

Оператори зсуву містять два операнди: перший – це вектор бітів, який необхідно зсунути, а другий – кількість розрядів, на яку необхідно виконати зсув. Напрямок зсуву визначається оператором: '<<' – для зсуву вліво, і '>>' – для зсуву вправо. Циклічні зсуви у Verilog відсутні.

Під час зсуву найлівіший (при зсуві вправо) або найправіший (при зсуві вліво) розряд стає пустим і заповнюється 0. Якщо правий операнд має невідоме значення або високий імпеданс, результат буде невизначеним.

Оскільки зсув на одну позицію вліво еквівалентний множенню вектора на 2 (або діленню на 2 у випадку зсуву на один розряд вправо), операції зсуву дуже корисні для ефективної апаратної реалізації деяких арифметичних операцій.

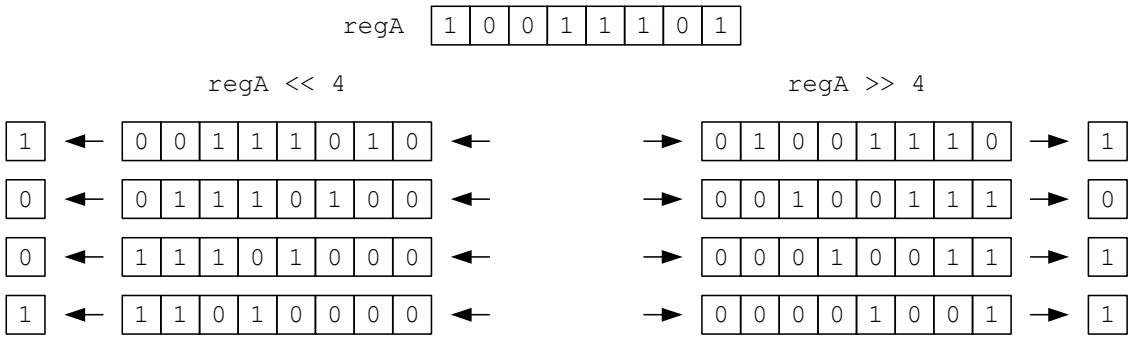


Рис.14.8.

Оператори конкатенації та реплікації

Конкатенація ({ , }) – це спеціальний оператор, який не вставляється між операндами, а оточує їх. Мета його використання – об'єднати декілька операндів в один вектор. Операнди розділяються комами. Операнди можуть бути скалярними провідниками або регістрами, векторами провідників або регістрів, біт-виборками, частковими виборками або розмірними константами.

Оператор реплікації використовується для декількаразової конкатенації того самого операнда. Це розширена версія оператора конкатенації, в якому репліковані операнди вказуються в конкатенаційних дужках, яким передує реплікаційне число, тобто для того, щоби виконати конкатенацію чотирьох копій Data[1:0] можна записати або {Data[1:0], Data[1:0], Data[1:0], Data[1:0]}, використовуючи оператор конкатенації, або {4{Data[1:0]}}, використовуючи оператор реплікації.

Реплікаційні вирази можна використовувати всередині оператора конкатенації як операнди.

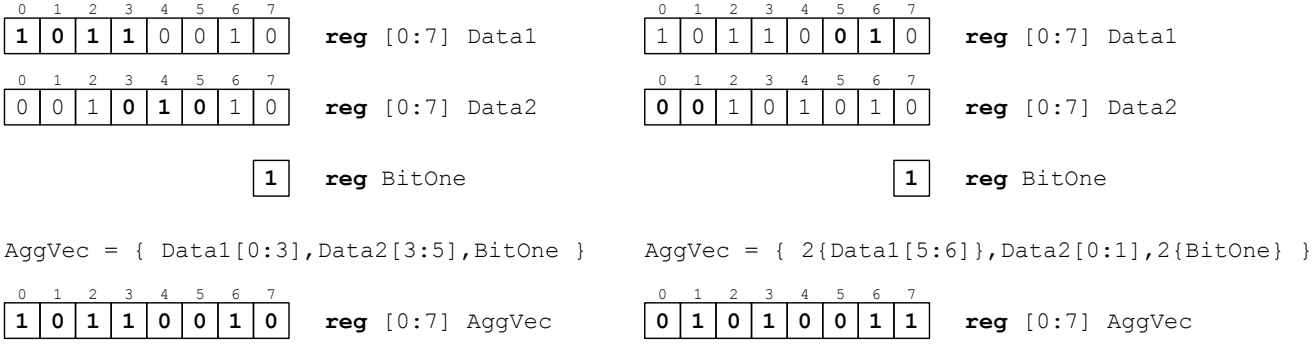


Рис.14.9.

Континуальні присвоєння

Континуальні присвоєння

Після того, як вирази або операції виконані, їх значення повинні бути присвоєні сигналу для того, щоби результат став доступним. Є два основних типи присвоєнь: *континуальні присвоєння* та *процедурні присвоєння*. Перші використовуються в основних операторах моделювання потоків даних. Другі будуть розглянуті пізніше.

Оператори континуального присвоєння складаються з наступних елементів:

- ключового слова ***assign***,
- необов'язкової декларації затримки,
- лівої частини присвоєння, де вказується цільовий сигнал, який може бути провідником (скаляром або вектором) або конкатенацією провідників; регістри не можуть використовуватись як цільові сигнали в континуальних присвоєннях,
- символа присвоєння '=',
- правої частини присвоєння, тобто виразу, операнди якого можуть бути довільного допустимого типу (константи, провідники, регістри, виклики функцій та ін.).

```
assign OutC = InA + InB;
assign OutC = InA & InB;
assign OutC = { InA,InB };
```

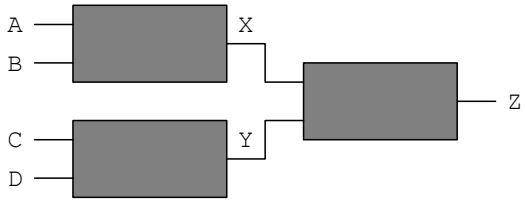
Рис.14.10.

Неявні континуальні присвоєння

Декларації провідників та присвоєння описуються у Verilog на одному рівні (всередині модуля) і не розділяються ключовим словом ***begin***. Вони можуть змішуватись і навіть об'єднуватись. Декларації провідників, об'єднані із присвоєннями, називаються *неявними континуальними присвоєннями* або *декларативними присвоєннями провідників*.

Коли використовується неявне континуальне присвоєння, ключове слово ***assign*** опускається. Замість цього декларація провідника включає ім'я провідника, за яким слідує символ присвоєння '=' і вираз.

Деколи може бути простіше використати неявне континуальне присвоєння, ніж "нормальне" (явне). Однак слід пам'ятати, що кожний провідник може бути задекларований тільки один раз, отже допускається тільке одне присвоєння значення виразу провіднику під час декларування. З іншого боку, такому провіднику можна присвоювати значення багатьох інших виразів за допомогою явних присвоєнь.



Явні присвоєння

```
module SomeMod (Z,A,B,C,D);
output Z;
input A,B,C,D;

wire X;
wire Y;

assign X = ...; // деякий вираз з операндами A і B
assign Y = ...; // деякий вираз з операндами C і D
assign Z = ...; // деякий вираз з операндами X і Y

endmodule
```

Неявні присвоєння

```
module SomeMod (Z,A,B,C,D);
output Z;
input A,B,C,D;

wire X  = ...; // деякий вираз з операндами A і B;
wire Y  = ...; // деякий вираз з операндами C і D;

assign Z = ...; // деякий вираз з операндами X і Y

endmodule
```

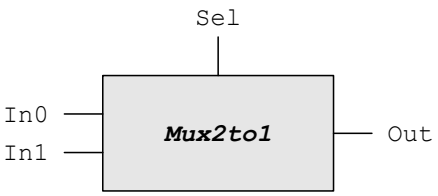
Рис.14.11.

Умовні присвоєння

Умовні оператори корисні не тільки в мовах програмування, а також при описі апаратних засобів. Однак з континуальними присвоєннями не можна використовувати конструкції ***if…then***. Замість цього всередині виразу присвоєння використовується *умовний оператор*.

Умовний оператор відрізняється від інших операторів, оскільки він має три операнди. Перший операнд – це умова виконання одного з двох інших операндів: якщо вона істинна (вираз дорівнює 1), результуючому сигналу присвоюється значення другого операнда, якщо хибна (0) – значення третього операнда, якщо невизначена (x) – обидва операнди порівнюються порозрядно, і, якщо вони однакові, їх значення присвоюється результату, якщо ж ні – у відповідний біт результату заноситься значення 'x'.

Кожний з операндів може бути виразом і може містити інші умовні вирази.



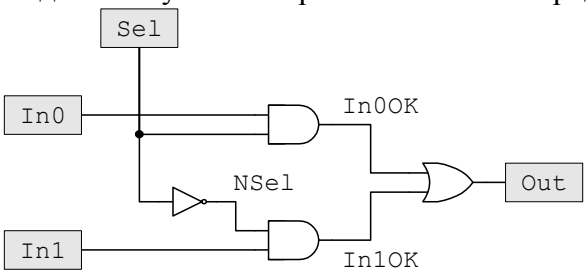

```
module Mux2to1 (Out,In0,In1,Sel);
output Out;
input In0,In1,Sel;

assign Out = Sel ? In1 : In0;

endmodule
```

Рис.14.12.

Коли все відбувається

Виконання континуальних присвоєнь відбувається згідно простого правила: континуальні присвоєння завжди активні. Як тільки хоча б один операнд з правого боку змінює значення, вираз переобчислюється і результат присвоюється провіднику в лівій частині присвоєння.

Якщо присутні декілька присвоєнь, не існує визначеної послідовності їх виконання, оскільки вони завжди виконуються паралельно. Отже порядок їх розташування не має значення.




```
module Mux2to1 (Out,In0,In1,Sel);
output Out;
input In0,In1,Sel;
wire Nsel,In0OK,In1OK;
  assign In0OK = In0 & Sel;
  assign In1OK = In1 & NSel;
  assign NSel = ~Sel;
  assign Out = In0OK | In1OK;
endmodule
```

- Зміна сигналу *Sel* активує паралельно присвоєння *In0OK* і *NSel*. *In0OK* активує *Out*. *NSel* активує *In1OK*, який реактивує *Out*.
- Зміна сигналу *In0* активує присвоєння *In0OK*, яке в свою чергу активує *Out*.
- Зміна сигналу *In1* активує присвоєння *In1OK*, яке в свою чергу активує *Out*.

Рис.14.13.

Затримки

Всі присвоєння, розглянуті вище, є ідеальними і не витрачають час на виконання. Однак реальним схемам притаманні часові затримки і Verilog надає механізм для специфікації схемних затримок. Для задання затримок є декілька опцій (мінімум, максимум, та ін.)

Затримки задаються в *одиницях часу*. Фізична реалізація одиниць часу залежить від користувача і одиницями часу необов'язково є наносекунди або мікросекунди. Визначення одиниць часу задається директивою компілятора ***timescale***.

Затримка в континуальному присвоєнні задає час між зміною операнда в правій частині присвоєння і власне присвоєнням результуючого значення або виразу цільовому сигналу. Затримка задається як *#t*, де *t* – кількість одиниць часу. Ця затримка вказується між ключовим словом ***assign*** і самим присвоєнням.

Затримки у Verilog інерційні.

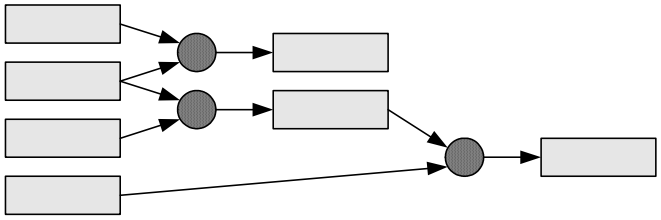
15. Поведінковий підхід.

Змінні та параметри

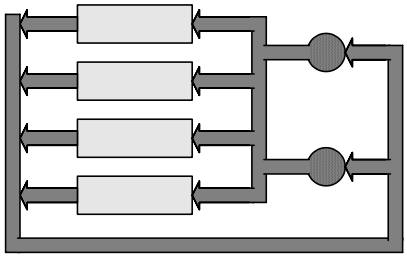
Чи достатньо провідників?

Використання провідників для опису потоків даних в цифрових схемах є природнім: кожний сигнал має певне значення, коли його драйвер включений. Коли ж його драйвер відключений, сигнал знаходиться у високому імпедансі. Подібна поведінка представляється провідниками і, фактично, вони були введені в мову для представлення таких фізичних процесів.

Коли необхідно описати поведінку складних схем, або систем за допомогою *алгоритмів*, необхідно мати щось подібне на *змінні* у мовах програмування. Змінній не призначено постійно драйвера, але вона постійно містить деяке значення. Змінна може зберігати значення настільки довго, скільки потрібно, і в будь-який момент може отримати інше значення. Якщо для цього використовувати провідник, він перейде в 'z'-стан, як тільки буде відключений від драйвера. Для запобігання втраті сигналом його значення при відключенні драйвера були введені інші об'єкти – *регістри*. Вони можуть зберігати значення між присвоєннями.



В описах *потоків данних* всі операції (кола) постійно присвоюються сигналам (прямокутники). Це означає, що апаратна реалізація вимагає стільки функціональних блоків, скільки операцій є у специфікації. Будь-яка зміна значення сигналу спричинює виконання асоційованих операцій і цей процес продовжується до тих пір, поки всі сигнали на стануть стабільними.



В описах *алгоритмів* операції виконуються не тоді, коли змінна змінює своє значення, а тоді, коли настає час диспетчеризованої операції, незалежно від значень її операндів. В результаті функціональні блоки можуть розділятись між операціями, виконання яких розділяється в часі.

Рис.15.1.

Регістри

Регістр формально визначається як "абстракція елементу збереження даних". Простіше кажучи, у Verilog – це аналог змінної в таких мовах програмування, як C або Pascal. Декларування регістрів у Verilog дуже подібне декларуванню змінних у C.

Регістр декларується так само, як і провідник – єдина різниця полягає в ключовому слові ***reg*** замість ***wire***. Аналогічно провідникам, існує декілька типів регістрів. Крім звичайних регістрів, що використовуються переважно для логічних значень, можна використовувати *цілі*, *дійсні*, *часові* змінні та змінні *дійсного часу*.

Тип регістра	Приклади	Опис
reg	**reg** enable; **reg** Mealy_out_1;	Цей регістр зберігає присвоєне значення до тих пір, поки йому не буде присвоєне нове значення. Він не є еквівалентом апаратного регістра, що будується на основі синхронних тригерів, і не він не вимагає тактового сигналу. Перед тим, як регістру буде присвоєне нове значення, він має значення за замовчуванням 'x' (на відміну від провідників, для яких значенням за замовчуванням є 'z'). Багаторозрядним регістрам можна присвоювати від'ємні значення (вони зберігаються у двійковому доповненні), але якщо цей регістр є операндом у виразі, його значення сприймається як беззнакове. Декларування змінної типу ***reg*** складається з ключового слова ***reg***, після якого вказується ім'я змінної. В одній декларації може бути задано кілька регістрів, тоді їх ідентифікатори розділяються комами.
time	**time** sim_time; **time** setup_time;	Спеціальний тип регістрів, який використовується для зберігання та маніпулювання часом симуляції, який присвоюється йому за допомогою системної функції ***$time***, і для відладки та звітування про симуляцію. Він веде себе так само, як і 64-розрядні регістри. Декларування змінної типу ***time*** складається з ключового слова ***time***, після якого вказується ім'я змінної. В одній декларації може бути задано кілька регістрів, тоді їх ідентифікатори розділяються комами.
integer	**integer** loop_cntr; **integer** cycles_passed;	Тип регістра, який служить для маніпулювання чисельними значеннями. Він поводиться як нормальний регістр з двома важливими виключеннями: його значення за замовчуванням має довжину не менше 32 розряди (залежить від реалізації), коли цілому присвоюється від'ємне значення і потім воно використовується як операнд, воно сприймається як від'ємне (всі цілі значення зберігаються як знакові). Цілому регістру не присвоюється за замовчуванням ніякого конкретного значення - це залежить від симулятора. Декларування змінної типу ***integer*** складається з ключового слова ***integer***, після якого вказується ім'я змінної. В одній декларації може бути задано кілька регістрів, тоді їх ідентифікатори розділяються комами.
real	**real** radius; **real** average;	Регістри дійсного типу служать для зберігання дійсних (нецілих) значень. Значення можуть задаватись як в десятковій, так і в експоненціальній нотації. На дійсні регістри накладаються деякі обмеження: набір доступних операндів дещо обмежений, вони не можуть декларуватись в діапазоні, вони мають значення за замовчуванням - 0. Дійсні числа можуть бути приведені до цілих шляхом округлення до найближчого цілого. У Verilog симулятори виконують неявне приведення дійсних до цілих і навпаки. Декларування змінної типу ***real*** складається з ключового слова ***real***, після якого вказується ім'я змінної. В одній декларації може бути задано кілька регістрів, тоді їх ідентифікатори розділяються комами.
realtime	**realtime** exact_simtime;	Тип ***realtime*** використовується для задавання часу, використовуючи дійсні числа, і є взаємозамінним їз дійсним типом. Декларування змінної типу ***realtime*** складається з ключового слова ***realtime***, після якого вказується ім'я змінної. В одній декларації може бути задано кілька регістрів, тоді їх ідентифікатори розділяються комами.

Рис.15.2.

Вектори та масиви

Регістрові *вектори* типу ***reg*** можуть бути задекларовані аналогічно векторам провідників, шляхом задавання діапазону індексів, що розміщується між ключовим словом, який задає тип регістру та його ім'ям (ідентифікатором).

Для інших типів регістрів Verilog пропонує масиви, недоступні для провідників. Головною візуальною відмінністю між векторами та масивами з однаковим діапазоном є місце розташування цього діапазону. Для векторів він розташований між типом та ідентифікатором, а для масивів він слідує після ідентифікатора. Є ще одна суттєва відмінність між векторами та масивами: вектор – це одиничний *n*-розрядний об'єкт, а масив – це впорядкована сукупність *n* одно- (або більше) розрядних об'єктів. Більше того, вектору, на відміну від масиву, може бути присвоєне значення одним оператором присвоєння.

Масиви можуть складатись з регістрів типів ***reg***, ***integer*** або ***time***, але не можуть містити елементи ***real*** або ***realtime***.

Характеристика	Вектори	Масиви
Декларування	Діапазон вектора задається між ключовим словом типу та його ідентифікатором, наприклад `reg [7:0] MyData;` описує регістровий 8-розрядний вектор.	Діапазон масиву задається після його ідентифікатора, наприклад `reg MyData [7:0];` описує масив 8ми однорозрядних регістрів.
Елементи	Вектор може складатись з елементів тільки типу **reg**.	Масив може складатись з елементів типу **reg**, **integer** або **time**.
Посилання	Для посилання на елемент вектора задається його ім'я та індекс, наприклад MyData[5] задає 5й розряд вектора MyData. Крім того, дозволені часткові виборки елементів вектору, наприклад MyData[2:0].	Посилання на елемент масива аналогічне векторам: ім'я та індекс, наприклад MyData[5] задає 5й елемент масива MyData.

Рис.15.3.

Слово пам'яті як операнд

Використання посилань на пам'ять у виразах Verilog дозволяється, але обмежено тільки частиною, що називається *словом пам'яті*. Це пов'язано з тип, як пам'ять описується: це вектор векторів і адресувати можна тільки найзовнішній вектор. Якщо використовується одиничний біт, або частина слова, повне слово має бути скопійоване в тимчасовий регістр і доступ до потрібних бітів може бути здійснений через цей регістр.

Синтаксис адресації пам'яті складається з імені пам'яті та виразу адреси:

```
mem_name [mem_expr]
```

Вираз може бути довільний, включно з словами з цієї ж пам'яті, що дозволяє здійснювати непряму адресацію. Наприклад, якщо адреса потрібного слова пам'яті зберігається в слові номер 12 тієї ж самої пам'яті, доступ до нього можна здійснити

```
MyMem[MyMem[12]]
```

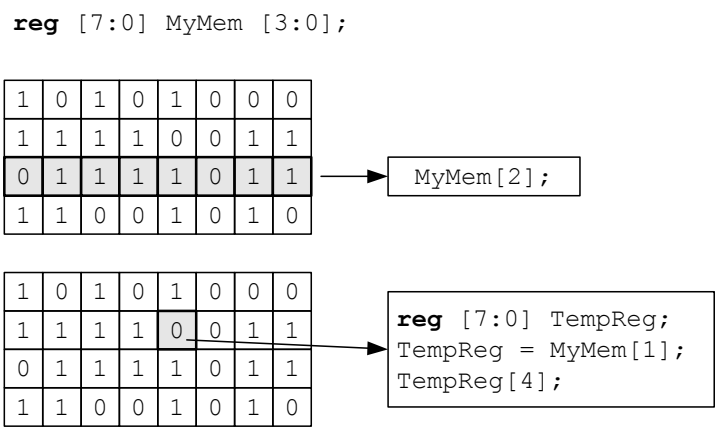


Рис.15.4.

Константи у Verilog

Головною метою використання констант є застосування "зрозумілих імен" замість літеральних значень, що також називаються "жорсткокодованими". При цьому код стає більш читабельним і простішим з точки зору підтримки, оскільки зміна константи в одному місці вплине на всі її входження у всьому модулі.

Якщо ж використовуються жорсткокодовані значення, користувач повинен уважно перевірити всю специфікацію рядок за рядком для того, щоби бути впевненим, що всі значення виправлені.

Константи не можуть мінятись в модулі і не можуть використовуватись як змінні (тобто їм не можна присвоювати нові значення). Тому вони не є провідниками або регістрами. У Verilog вони називаються *параметрами*.

```
reg [7:0] DataBus;
  ...
for (cntr = 0; cntr < 8; cntr = cntr + 1)
  ...
for (cntr = 7; cntr >= 0; cntr = cntr - 1)
  ...
```

Якщо чисельний параметр схеми (такий як розмір шини) жорсткокодований, будь-яка зміна його значення вимагає уважного сканування коду для гарантованої заміни всіх значень.

```
parameter BufSize = 8;
reg [BufSize-1:0] DataBus;
  ...
for (cntr = 0; cntr < BufSize; cntr = cntr + 1)
  ...
for (cntr = BufSize - 1; cntr >= 0; cntr = cntr - 1)
  ...
```

Якщо параметр схеми описаний як константа, зміна його значення відбувається тільки один раз - в декларації константи. Решта коду не змінюється. Більше того, код стає читабільнішим, оскільки користувачу не потрібно пам'ятати, що кожне число означає.

Рис.15.5.

Декларування та використання параметрів

Декларування параметрів виконується за наступним синтаксисом:

- ключове слово ***parameter***,
- ім'я параметра,
- значення параметра, що задається після символа '=',
- крапка з комою, що закінчує рядок.

Дві або більше констант можуть бути задані в одному рядку, в такому випадку вони розділяються комами.

Найчастіше параметри використовуються для представлення затримок та розмірів об'єктів (ширини шин або векторів). Отже, найчастіше вони зустрічаються у деклараціях діапазонів та лічильників циклів.

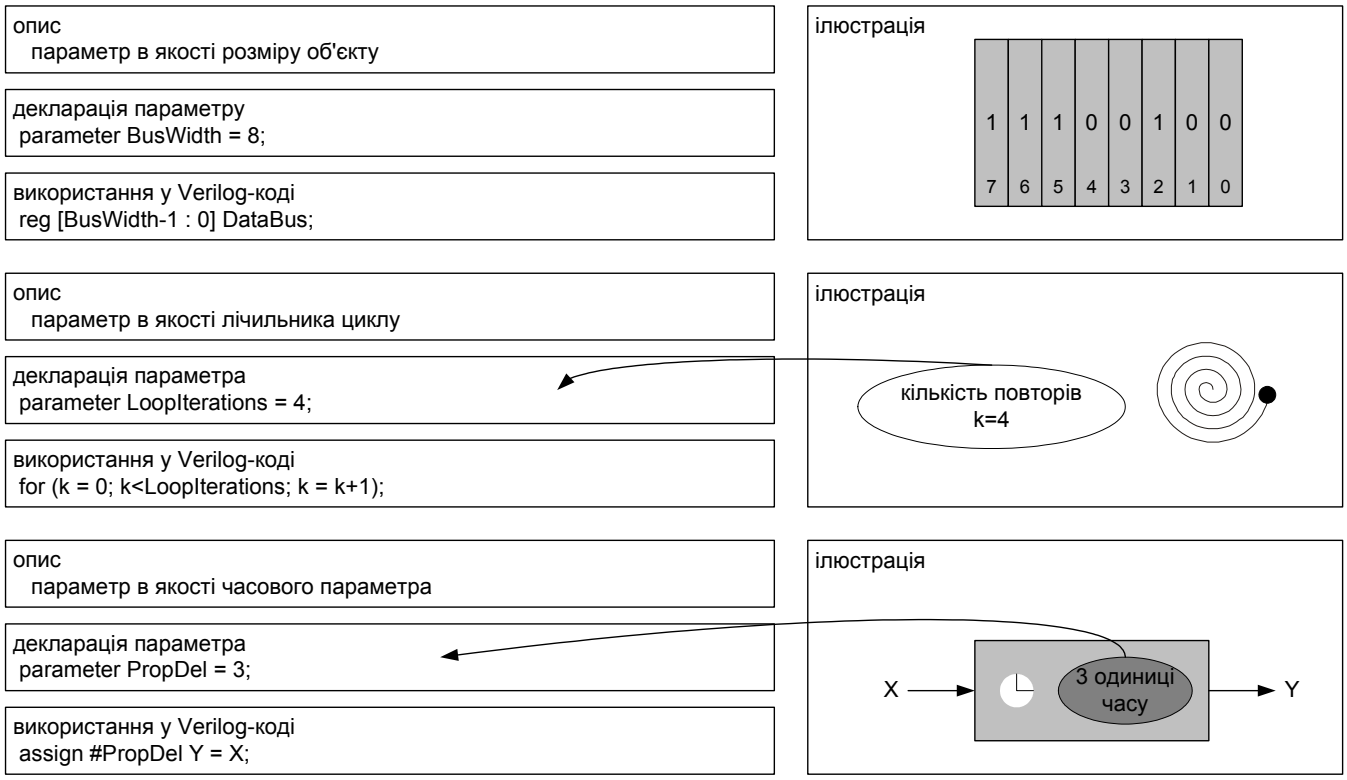


Рис.15.6.

Основи поведінкового опису

Від потоку даних до поведінки

Всі поведінкові специфікації повинні бути інкапсульовані в один або більше *блоків*, які визначаються як *процедурні оператори*, що розташовуються всередині модуля так само, як і континуальні присвоєння або реалізації, і так само виконуються паралельно. Коли виконується блок, його оператори виконуються відповідно специфікації (послідовно або паралельно). Існує два типи блоків: ***initial****-блоки* та ***always****-блоки*. Кожний модуль Verilog може містити довільну кількість блоків обох типів, але вони не можуть включати один одного.

Якщо блок містить декілька операторів, вони можуть бути згруповані за допомогою пар ключових слів ***begin-end*** та ***fork-join***. Перша пара використовується для групування операторів, що виконуються послідовно, а друга – для операторів, що виконуються паралельно.

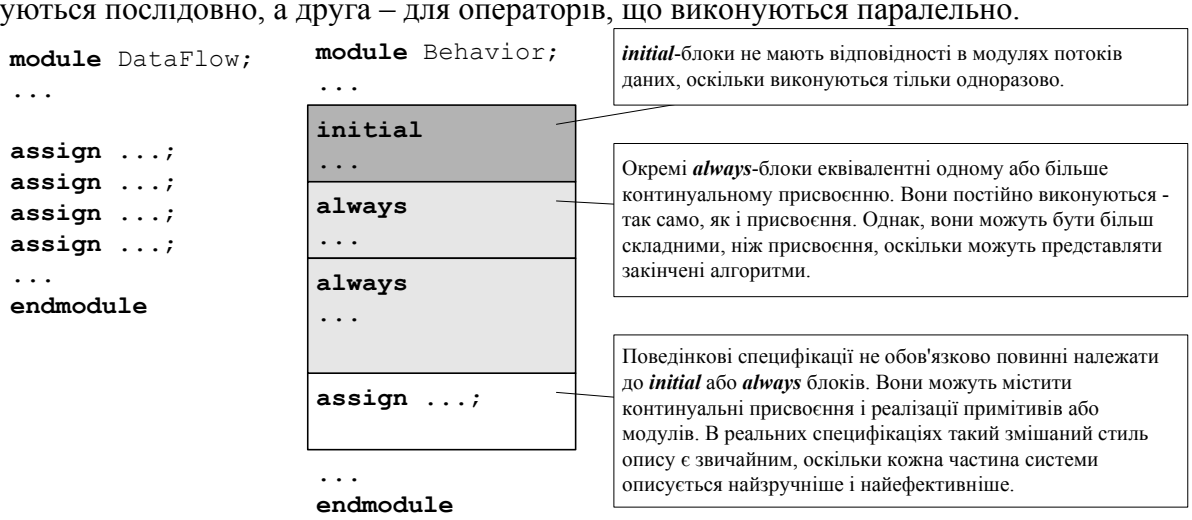


Рис.15.7.

Поведінкові блоки

Два *поведінкових блоки* – ***initial*** та ***always*** – є ідентичними за виключенням одного моменту: Вміст ***initial***-блоку виконується на самому початку симуляції (в момент часу 0) тільки один раз і *ніколи* не виконується знову, ***always***-блок також починає виконуватись в момент часу 0, але його вміст виконується у неперервному циклі і повторюється постійно на протязі всього часу симуляції.

Модуль може містити по декілька блоків обох типів. В такому випадку блоки виконуються паралельно з моменту часу 0.

Оскільки для регістрів початковим значенням за замовчуванням є 'x', їх необхідно встановлювати в '0' або інше початкове значення, перш ніж почнеться симуляція. Для цього використовуються ***initial***-блоки, які задають початкові значення змінних (регістрів).

always-блоки, з іншого боку, представляють поведінку цифрових схем, що працюють весь час, поки є живлення.

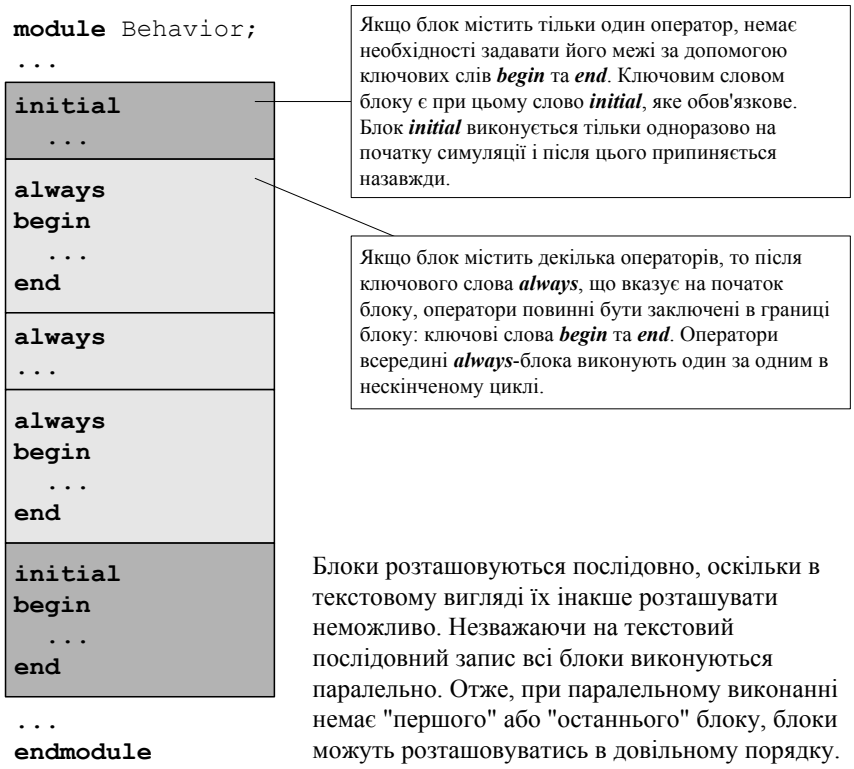


Рис.15.8.

Присвоєння змінних

Найбільш важливою операцією в поведінкових специфікаціях є присвоєння значень виразів змінним. Таке присвоєння формально називається *процедурним присвоєнням* і дещо відрізняється від континуального присвоєння:

- цільовим сигналом присвоєння (ліва частина присвоєння) має бути регістр (***reg***, ***integer***, ***real*** або ***time***) – біт, часткова виборка регістра, але не провідник,
- присвоєнню не передує ніякого ключового слова,
- має бути описано в поведінковому блоці (***initial*** або ***always***),
- континуальні присвоєння змінюють значення цільового провідника, як тільки хоча б один з операндів правої частини присвоєння змінює своє значення, а процедурне присвоєння змінює значення цільового регістру тільки тоді, коли присвоєння виконується у відповідності з послідовністю операцій у поведінковому блоці.

Наступний приклад демонструє найпростішу форму процедурного присвоєння, що називається *блочним присвоєнням*. Інший тип – *неблочне присвоєння* буде наведено пізніше.

```
reg A;
integer LoopCount;
reg [7:0] VecM;

initial
begin
  A = 1'b0;
  LoopCount = 0;
  VecM = 8'b0;
end
```

Всі поведінкові присвоєння (змінних) мають бути описані в процедурному блоці - або ***initial***, або ***always***. В наведеному прикладі ініціалізуються три змінні. Ця операція виконується тільки один раз на початку симуляції і описується в ***initial***-блоці.

Цільовий сигнал процедурного присвоєння має бути регістровою змінною.

Рис.15.9.

Затримки в присвоєннях

Одне з головних правил симуляції модулів є те, що Verilog-симулятор не працює без часового керування операторами, такого як *затримки* та *події*. Без них симуляційний час буде постійно рівний 0 і всі оператори будуть виконуватись миттєво.

Найпопулярнішим методом опису затримок є використання *регулярних затримок присвоєнь*. Ці затримки задаються так само, як і в континуальних присвоєннях (символ '#' і число одиниць часу, що передує оператору присвоєння), але їх значення в коді різне. Кожна затримка задає інтервал часу між обчисленням оператора (симулятором) та його виконанням. Іншими словами, це час, що має пройти між виконанням попереднього та біжучого операторів.

Це тактовий генератор. Тактовий сигнал переключається кожні півциклу, що описаний як параметр.

Сигнал описується як *регістр*, оскільки він повинен утримувати значення між послідовними присвоєннями. Його зміна описується в поведінковому стилі.

Тривалість тактового циклу з використанням параметру, а не жорсткокодованого значення. Його значення представляє тут півцикл, але він може задавати також і повний цикл. В такому випадку затримка буде записана як *#(FullCycle / 2)*. Затримка може бути задана як *константний вираз*, тобто вираз, значення якого визначається на етапі компіляції.

Тактовий сигнал ініціалізується з використанням блоку ***initial***. Якщо це присвоєння розмістити в ***always***-блоці, тактовий сигнал буде ініціалізуватись кожні півциклу, оскільки ***always***-блок повторюється у неперервному циклі.

Головна частина опису тактового сигналу записується в ***always***-блоці. Як і попередній ***initial***-блок, цей блок складається тільки з одного оператора, отже границі ***begin - end*** не обов'язкові. Присвоєння виконується кожні *HalfCycle* часових одиниць і переключає сигнал *Clk*.

```
module ClockGenerator;

reg Clk;
parameter HalfCycle = 20;

initial Clk = 1'b0;

always #HalfCycle Clk = ~Clk;

endmodule
```

Рис.15.10.

Виконання блоку

Розглянемо симуляцію поведінкового блоку із затримками. Для більшої наочності виконання послідовності операторів будемо використовувати приклад із іменами операторів замість реальних операторів.

Коли починається симуляція, всі поведінкові блоки активізуються. Перший оператор після ключового слова ***begin*** виконується без будь-якої затримки. Якщо наступний оператор не затриманий, він також виконується в той самий час (момент часу 0), і т.д., поки не буде знайдено затримку в блоці. Симуляційний час в розмірі кількості часових одиниць, заданих в затримці, спливає. Далі виконується наступний оператор, і т.д. Якщо в блоці немає більше операторів (тобто зустрілось ключове слово ***end***), блок припиняється назавжди, якщо це ***initial***-блок. Якщо ж це ***always***-блок, то керування передається на перший оператор блоку після слова ***begin***.

```
initial
begin
      S1;
  #5  S2;
end

always
begin
      S3;
  #10 S4;
  #10 S5;
end;

always
  #5 S6
```

- Після старту симуляції (момент часу 0) всі блоки активуються і виконується перший "незатриманий" оператор *S1*. Оскільки оператор *S3* в ***always***-блоці не має затримки, він також виконується в момент часу 0.
- В момент часу 5 виконуються два оператори: *S2* і *S6*. *S4* і *S5* очікують наступні 5 одиниць часу. ***initial***-блок більше виконуватись не буде, оскільки всі його оператори вже виконані.
- Після наступних 5 одиниць часу (10), паралельно виконуються *S4* і *S6*. Наступними операціями в обох ***always***-блоках - оператори *S5* і *S6* відповідно. Оскільки вони обидва затримані, більше операції не виконуються наступні 5 одиниць часу.
- ...

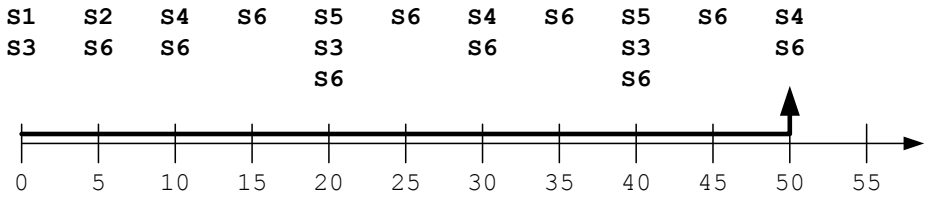


Рис.15.11.

Складні оператори

Умовні оператори

Деякі оператори виконуються тільки тоді, коли виконується задана умова. Такі оператори називаються *умовними* і їх реалізація у Verilog виглядає наступним чином:

```
if (expression_true) true_statement; else false_statement;
```

Цей оператор Verilog виконується за наступною процедурою:

- якщо результат обчислення умови *expression_true* є істинним (ненульове значення), виконується оператор *true_statement*; це має бути одиничний оператор або блок операторів (розміщений між ключовими словами ***begin*** і ***end***);
- якщо результат обчислення умови *expression_true* є хибним (‘0’) або невідомим (‘x’ або ‘z’), виконується оператор *false_statement*; аналогічно *true_statement* це має бути одиничний оператор або блок операторів; підоператор ***else*** може бути опущено, в такому випадку, якщо умова хибна, не виконується жодного оператора.

```
module ShiftReg (Outs,Ins,Clk,Clr,Set,Shl,Shr);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk,Clr,Set,Shl,Shr;

initial
  Outs = 0;

always @(posedge Clk)
  if (Clr == 1) Outs = 0;
    else if (Set ==1) Outs = {Size{1'b1}};
      else if (Shl == 1) Outs = Outs << 1;
        else if (Shr == 1) Outs = Outs >> 1;
          else Outs = Ins;

endmodule
```

Рис.15.12.

Множинний вибір

Замість вкладених умовних операторів, що є складними і нечитабельними, використовують оператори *множинного вибору*, що задаються за допомогою ключового слова ***case***.

Вираз, вказаний в дужках після ключового слова ***case***, перевіряється і порівнюється по черзі із значеннями, вказаними нижче. Перший же оператор, значення умови виконання якого відповідає обчисленому значенню умови, виконується. Якщо жодне із наведених значень не відповідає умові, виконується оператор ***default***. Зауважте, що оператор ***case*** порівнює всі розряди так як вони є, тобто всі чотири значення враховуються. Для того, щоби сприймати високий імпеданс ('z') як несуттєве значення, необхідно використовувати оператор ***casez*** замість ***case***. Відповідно, ***casex*** інтерпретує 'x' і 'z' як несуттєве. Тобто 'zx' буде відповідати довільній парі бітів в операторі ***casex***, довільній парі бітів, що закінчується на 'x' ('0x' '1x' 'xx' 'zx') в операторі ***casez*** і тільки значенню 'zx' в операторі ***case***.

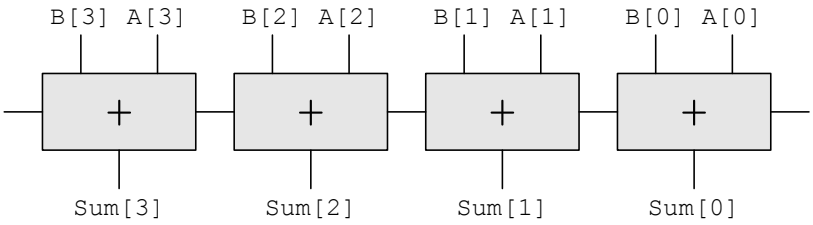
```
module ShiftReg (Outs,Ins,Clk,Clr,Set,Shl,Shr);
parameter Size = 8;
parameter MSB = Size - 1;
output [MSB:0] Outs; reg [MSB:0] Outs;
input [MSB:0] Ins;
input Clk,Clr,Set,Shl,Shr;

initial
  Outs = 0;

always @(posedge Clk)
  case ({ Clr,Set,Shl,Shr })
    4'b1xxx : Outs = 0;
    4'bx1xx : Outs = {Size{1'b1}};
    4'bxx1x : Outs = Outs << 1;
    4'bxxx1 : Outs = Outs >> 1;
    default   Outs = Ins;
  endcase

endmodule
```

Рис.15.13.

Цикли

Цикли використовуються з метою повторення заданих операторів. Оператори повторюються задану кількість разів або так довго, скільки буде виконуватись умова. Є чотири різних конструкції циклів, що називаються за їх ключовими словами: ***forever***, ***repeat***, ***while*** і ***for***. Вони подібні операторам циклів в мові С:

- цикл ***forever*** – найпростіший з чотирьох циклів і повторює свій вміст постійно,
- цикл ***repeat*** – виконує свій вміст фіксовану кількість разів, що задається константою або змінною, розміщеною справа після ключового слова,
- цикл ***while*** – замість кількості повторів задається умова виконання,
- цикл ***for*** – найгнучкіший цикл, задає початкову умову, умову закінчення та оператор, що оновлює керуючу змінну циклу після кожної ітерації.




```
module RCAdd (Sum,A,B);
output [3:0] Sum; reg [3:0] Sum;
input [3:0] A,B;
reg C;
integer I;

always
begin
  C = 0;
  for (I = 0; I <= 3; I = I + 1)
  begin
    Sum[I] = A[I] ^ B[I] ^ C;
    C = A[I] & B[I] | A[I] & C | B[I] & C;
  end
end

endmodule
```

Це приклад 4-розрядного некаскадованого суматора із наскрізним переносом (він не має вхідного та вихідного переносу).

Рис.15.14.

Розширене керування поведінкою

Керування часом в поведінкових блоках

Складні оператори керування, представлені вище, дозволяють організовувати умовне виконання коду. Однак, вони не залежать від часових умов. Отже, необхідно розглянути ще одну конструкцію, що забезпечує керування часом: *затримка присвоєння*. Такі затримки можуть бути задані не тільки для присвоєнь, але і для довільного оператора поведінкового блоку. Ця затримка описується перед оператором і затримує виконання оператора на визначену кількість одиниць часу.

Затримка може бути також описана всередині присвоєння, формуючи внутрішню затримку присвоєння. Така затримка задається відразу після знаку присвоєння ('=') і інтерпретується дещо інакше: права частина присвоєння виконується миттєво, як тільки цей оператор присвоєння отримує керування, але присвоєння результату цільовому сигналу в лівій частині присвоєння відкладається на зазначену затримку, що еквівалентно використанню тимчасової змінної.

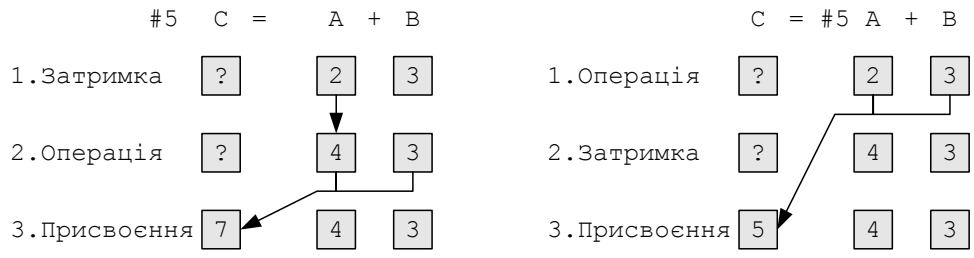


Рис.15.15.

Події

Затримки – це не єдиний спосіб явного часового керування виконанням процедурних (поведінкових) операторів. Інший відноситься до *подій*, тобто змін значень провідників та

регістрів. Типовим прикладом події є зростаючий або спадаючий фронт тактового сигналу, що використовується для синхронізації операцій схеми.

Оператор *керування подією* описується за допомогою символа '**@**', після якого вказується ім'я регістра або провідника, і впливає на виконання оператора. Ця конструкція керування подією може бути розташована там же, де і затримки, тобто перед або всередині оператора.

Крім загальних подій (коли довільна зміна приводить до активації оператора), існують дещо вдосконалені події, які записуються ***negedge*** і ***posedge***, і позначають спадаючий та зростаючий фронти відповідно.

```
@(CLK)Q = D;

@(posedge CLK)Q = D;

@(negedge CLK)Q = D;
```

- Будь-яка зміна в логічну '1' вважається *додатнім фронтом*. В такій ситуації визначаються перша ("будь-яка зміна *CLK*") і друга ("додатній фронт *CLK*") події і відповідні оператори виконуються.
- Будь-яка зміна в логічний '0' вважається *від'ємним фронтом*. В такій ситуації визначаються перша ("будь-яка зміна *CLK*") і третя ("від'ємний фронт *CLK*") події і відповідні оператори виконуються.

Рис.15.16.

Оператор wait

Керування подіями відноситься до зміни значення сигналом або змінною. Але деколи важливим є не зміна, а значення (наприклад, рівень сигналу дозволу тристабільного буфера). Таке керування *подіями, чутливими до рівня* можуть бути визначені за допомогою оператора ***wait***.

Оператор ***wait*** починається з ключового слова ***wait***, після якого вказується логічна умова в дужках, і оператор, виконання якого залежить від результату умови. В більшості випадків в якості умови задається значення сигналу дозволу:

```
wait (enable) statement;
```

Оператор ***wait*** затримує виконання оператора до того часу, коли умова стане істинною (у наведеному прикладі – до тих пір, поки сигнал *enable* не стане рівний '1').

Керування подіями, чутливими до рівня може також використовуватись для припинення виконання блоку за допомогою *нуль*-оператора (тобто вказується тільки ключове слово ***wait*** і умова).

```
wait(EN) #5 C = A + B;
```

Як тільки сигнал *EN* змінює своє значення на '1', оператор присвоєння починає виконуватись і через 5 одиниць часу сигналу *C* буде присвоєне значення *A+B*. Якщо значення сигналу *EN* буде рівне '0', 'z' або 'x', оператор присвоєння не виконується.

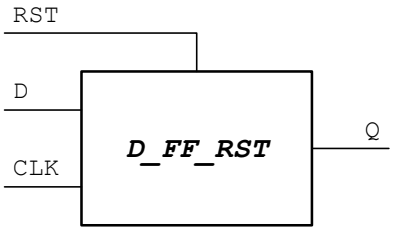
Рис.15.17.

Список чутливості

Якщо оператор активується подією із сигналом, він називається *чутливим* до цього сигналу. Verilog не обмежує чутливість тільки одним сигналом, а дозволяє задавати оператори, чутливі до довільної кількості подій. Такі події (або імена сигналів) розділяються ключовим словом ***or*** і називаються *списком чутливості*.

Керування подіями і списки чутливості найчастіше використовуються в ***always***-блоках. Оскільки керування подіями може бути задане для довільного оператора, то і складні оператори, заключені між границями ***begin-end*** також можуть керуватись подіями. Це приводить до конструкцій, в яких ***always***-блоки описуються із списком чутливості, розташованим відразу після ключового слова ***always***. Ця подія відноситься до блоку, розташованого нижче, і записується після ключового слова тільки для покращення читабельності.

Список чутливості дозволяє припиняти та поновлювати нескінчений цикл ***always***-блоків: блок виконується, після чого припиняється на початку наступної ітерації до тих пір, поки якийсь з сигналів у списку чутливості не змінить свого значення.




```
always @(posedge RST or posedge CLK)
begin
  if(RST)
    Q = 1'b0;
  else if(posedge CLK)
    Q = D;
end
```

- Якщо значення сигналу *RST* змінюється в '1', то, оскільки *RST* присутній в списку чутливості блоку, визначається його додатній фронт, і блок активується і послідовні оператори починають виконуватись. Вихідному сигналу *Q* присвоюється значення '0'.
- Будь-яка зміна сигналу *D* не активує ***always***-блок, оскільки *D* не присутній в списку чутливості блоку. Іншими словами, блок нечутливий до подій із сигналом *D*.
- Якщо значення сигналу *CLK* змінюється в '1', блок активується і послідовні оператори починають виконуватись. Вихідному сигналу *Q* присвоюється значення входу *D*.

Рис.15.18.

Неблочні присвоєння

Неблочні присвоєння вносять паралельність в послідовні оператори.

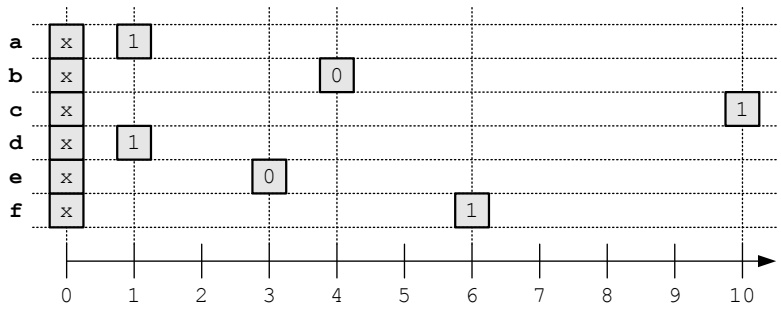
Більше того, вони є класичним прикладом використання внутрішніх затримок присвоєнь. Як зазначено в назві, коли активуються неблочні присвоєння, витримується затримка (задана всередині присвоєння) для того, щоби присвоїти коректне значення змінній з лівого боку присвоєння, а керування передається наступному оператору, незалежно від значення затримки.

В результаті неблочні присвоєння можуть виконуватись паралельно з іншими операторами блоку. Для того, щоби відрізнити неблочні присвоєння від блочних, використовується інший символ присвоєння '<='.

Неблочні присвоєння використовуються для моделювання множинних паралельних передач даних після загальної події.

Блочні присвоєння виконуються послідовно, що може привести до "змагань" умови. Крім того, читання (обчислення правої частини присвоєння) і запис (реальне присвоєння) розділені в цих присвоєннях, виключаючи "змагання" умови в деяких присвоєннях (як своппінг змінних).

```
initial
begin
  a = #1 1;  // 0 + 1 - присвоєння в 1
  b = #3 0;  // 1 + 3 - присвоєння в 4
  c = #6 1;  // 4 + 6 - присвоєння в 10
end

initial
begin
  d <= #1 1; // 0 + 1 - присвоєння в 1
  e <= #3 0; // 0 + 3 - присвоєння в 3
  f <= #6 1; // 0 + 6 - присвоєння в 6
end
```



В лівій частині кожного присвоєння мають бути регістри, які ініціалізуються значенням 'x'.
В момент 0 активуються праві частини перших операторів кожного блоку. Оскільки присвоєння в другому блоці неблочні, виконуються і праві частини другого та третього оператора.
В момент 1 результати відповідних виразів присвоюються *a* і *d*, і вони отримають значення '1' одночасно. Завершення першого присвоєння першого блока автоматично стартує другий оператор - значення '0' для *b* відкладається на 3 одиниці часу, тобто до момента 4.
В момент 3 виконується друге присвоєння другого блока, затримане на 3 одиниці часу від початку виконання блоку. Значення '0' записується в *e*.
В момент 4 *b* отримає нове значення '0', визначене 3 одиниці часу назад. Після присвоєння починає виконуватись наступний (третій) операнд першого блока - значення '1' для *c* відкладається на 6 одиниці часу, тобто до моменту 10.
В момент 6 *f* отримує нове значенні '1'.
В момент 10 *c* отримує нове значення '1', відкладене 6 одиниць часу тому (в момент 4).
Отже всі пари регістрів (*a*-*d*, *b*-*e* і *c*-*f*) отримали однакові значення, але в різні моменти часу, оскільки описані вони по різному.

Рис.15.19.

Паралельні блоки

Неблочні присвоєння вводять паралелізм в поведінкові блоки, але їх застосування обмежене. Існує більш загальний шлях визначення паралельних операторів всередині процедурних ***initial***- та ***always***-блоків. Для забезпечення паралельного виконання (і симулювання) операторів необхідно використовувати ключові слова ***fork*** та ***join*** замість ***begin*** та ***end***.

Всі оператори, описані всередині паралельного блоку, виконуються паралельно. Отже, всі задані для цих операторів затримки відносяться до моменту початку виконання цих операторів, а не до закінчення виконання попереднього оператора (як у випадку послідовних блоків). Це дуже подібне описам потоків даних, розглянутих раніше.

```
reg [7:0] RegX;
initial
begin
  #50 RegX = 'hFF;
  #50 RegX = 'h01;
  #50 RegX = 'h2F;
  #50 RegX = 'h00;
end
```

В *послідовному блоці з блочними присвоєннями* затримки задаються по відношенню до попередньої операції. Деколи це буває дуже незрозуміло, особливо при створенні часової діаграми. З іншого боку, це найприродніший шлях опису послідовності операторів.

```
reg [7:0] RegX;
initial
begin
  RegX <= #50  'hFF;
  RegX <= #100 'h01;
  RegX <= #150 'h2F;
  RegX <= #150 'h00;
end
```

В *послідовному блоці з неблочними присвоєннями* затримки задаються по відношенню до початку виконання блоку. Легко зрозуміти, що і коли трапиться. Але є деякі недоліки: затримки мають бути задані всередині присвоєнь (а це погіршує читабельність).

```
reg [7:0] RegX;
initial
fork
  #50  RegX = 'hFF;
  #100 RegX = 'h01;
  #150  RegX = 'h2F;
  #200  RegX = 'h00;
join
```

Паралельний блок поєднує переваги блочних та неблочних присвоєнь послідовних блоків: кожна затримка задається по відношенню до "часу входження" в блок (а не один відносно іншого) і вказується перед присвоєнням. Перевага врівноважується недоліком можливості появи "змагань" в паралельних блоках.

Рис.15.20.

Завдання та функції

Підпрограми у Verilog

Як і в інших мовах програмування, у Verilog код може бути повторно використаний в різних специфікаціях. Функціональні специфікації побудови блоків або інтерфейсних протоколів – найтиповіші приклади повторного використання коду. Якщо потрібно повторно використовувати блоки або інтерфейсні протоколи, їх оформляють як підпрограми. У Verilog є два типи підпрограм: *завдання* та *функції*.

Повторне використання є не єдиною причиною використання *завдань* та *функцій*. Дуже часто частина коду, яка більше ніде не використовується, виділяється у формі підпрограми з метою покращення читабельності та полегшення підтримки. Більшість інструкцій та операцій мікропроцесорів описуються таким чином.

Категорія	Завдання	Функції
Характер	Як завдання, так і функції є поведінковою частиною коду, тобто вони можуть містити тільки поведінкові оператори і використовуються тільки всередині поведінкових блоків (***initial*** або ***always***). Вони можуть також використовуватись всередині інших підпрограм, але з деякими обмеженнями. Локальні змінні, регістри, цілі та дійсні можуть визначатись і в завданнях, і в функціях, і в обох випадках провідники всередині них визначатись не можуть.	
Декларування	Як завдання, так і функції декларуються всередині модуля і є локальними по відношенню до модуля, в якому вони задекларовані. Для того, щоби можна було використовувати завдання та функції в різних модулях, вони повинні бути описані в окремому файлі, який підключається до модуля директивою компілятора ***include***.	
Симуляційний час	Немає обмежень на використання будь-якого керування часом та подіями всередині завдання. Отже, завдання виконуються в час, заданий розробником.	Використання керування часом або подіями будь-якого типа у функціях не дозволяється. Вважається, що функція виконується за одну одиницю часу.
Дозвіл інших підпрограм	Немає обмежень на використання інших підпрограм всередині завдання, тобто завдання може викликати інше завдання або функцію.	В зв'язку з обмеженнями на керування часом, функції не можуть викликати завдання, навіть якщо в цих завданнях керування часом не виконується. Але викликати з функцій інші функції дозволяється.
Аргументи	Кількість аргументів для завдань необмежена. Завдання може мати 0 або більше аргументів довільного типу: вхідні, вихідні або двонапрямлені. З іншого боку, завдання не повертають ніяких значень за визначенням, але можуть передавати значення (довільну кількість) через вихідні та двонапрямлені аргументи.	Кожна функція повинна мати хоча б один вхідний аргумент. Більше того, вони не можуть мати вихідних або двонапрямлених аргументів. Результат виконання функції - це завжди одиничне значення, що повертається через ім'я функції.
Мета	Завдання можуть використовуватись у Verilog-коді, що містить затримки, часові або подійні конструкції. Вони можуть бути багатоцільовими і обчислювати декілька результатів. Завдання викликаються як окремі оператори.	Головна мета використання функцій - реакція на деякі вхідні сигнали (часто на один) з єдиною відповіддю. В зв'язку з обмеженнями на керування часом, функції скоріше відносяться до комбінаційного коду, подібно перетворенням та обчисленням. Функції викликаються як операнди у виразах.

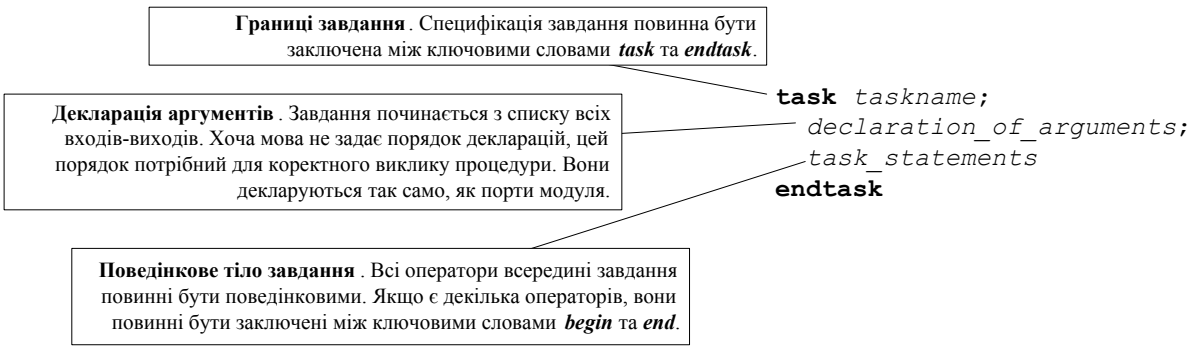
Рис.15.21.

Декларування завдання

Завдання може розглядатись як іменована частина коду з визначеними *входами* (***input***), *виходами* (***output***) та входами-виходами (***inout***). Воно забезпечує більш абстрактний вигляд системи, що описується, і дозволяє використовувати той самий код повторно в інших розділах проекту.

Визначення завдання виконується за наступною процедурою:

1. Описується код, що має міститись у завданні. Якщо він складається з більше ніж одного оператора, то заключається в границі ***begin-end***.
2. Визначається роль кожної змінної всередині коду: *входи*, *виходи*, *входи-виходи*, *локальні* та *тимчасові* змінні. Вони описуються перед кодом.
3. Визначається ім'я завдання і задається після ключового слова ***task***, але перед декларацією. Після коду вказується ключове слово ***endtask***.




```
task factorial;
  output [31:0] OutFact;
  input [3:0] n;

  integer Count;

  begin
    OutFact = 1;
    for (Count=n; Count>0; Count=Count-1)
      OutFact = OutFact * Count;
  end

endtask
```

Рис.15.22.

Виклик завдання

Завдання може бути викликане (або *дозволене*, як це формально називається в Language Reference Manual) в довільному поведінковому блоці модуля, в якому це завдання визначено. Виклик виглядає як окремий оператор і вказується згідно наступного синтаксису:

```
taskname (list_of_arguments);
```

Ключовим моментом є список аргументів. Для кожного формального аргументу *входу*, *виходу* та *входу-виходу* (тобто, описаного в декларації завдання) ставиться у відповідність фактичний аргумент.

```
task SomeTask;
  input a,b;
  output k,m;

  begin
    k = a + b;
    m = a - b;
  end

endtask
```

```
SomeTask (r,s,w,x);
```

```
task SomeTask;
  input r,s;
  output w,x;

  begin
    w = r + s;
    x = r - s;
  end

endtask
```

Рис.15.23.

Декларування функцій

Різниця між *завданнями* та *функціями* відбивається на тому, як вони визначаються та викликаються.

1. Всі *функції* повертають одиничний результат, який доступний через виклик функції. Це вимагає присвоєння його значення локальній змінній, що має таке саме ім'я, як і функція. Ніяких інших присвоєних локальних змінних функція повернути не може.
2. Результат функції за замовчуванням є 1-розрядним регістром. Якщо використовується функція будь-якого іншого регістрового типу (цілого, дійсного, дійсного часу або вектора), це має бути зазначено у заголовку функції.

3. Декларативна частина функції може містити тільки декларації входів і локальних змінних – виходи та входи-виходи не допускаються.
4. Функція починається з ключового слова ***function*** і закінчується ключовим словом ***endfunction***.

```
function [function_type] function_name
  declaration_of_inputs;
  [declaration_of_local_variables;]
  begin
    behavioral_statements;
    function_name = expression;
  end
endfunction
```

```
function [31:0] Factorial;
  input [3:0] Operand;
  reg [3:0] i;
  begin
    Factorial = 1;
    for (i=2; i<=Operand; i=i+1)
      Factorial = i * Factorial;
  end
endfunction
```

```
function ParityCheck;
  input [3:0] Data;
  begin
    ParityCheck = ^Data;
  end
endfunction
```

Рис.15.24.

Виклик функції

На відміну від *дозволу завдань*, що викликаються як окремі оператори, *виклик функції* – це операнд виразу. Практично це означає, що значення функції має бути присвоєно змінній, або бути частиною виразу, або бути аргументом іншої функції.

Результат виконання функції повертається через її ім'я. Внутрішньо це виконується як неявне декларування змінної з таким самим іменем, що і функція. Результат операцій всередині функції присвоюється цій змінній і передається далі як ця змінна.

`function ParityOdd;`
`  input [15:0] DataVectorLong;`
`  begin`
`  ParityOdd = ^DataVectorLong;`
`  end`
`endfunction`
	Функція визначає парність 16-розрядного вектора, що використовується як вхід. Результат операції, що виконується через редукційний *xor*-оператор, присвоюється імені функції, і використовується там, де ця функція викликається.

`Flags[3] = ParityOdd(RegAX);`	Найпростіший приклад виклику фукнції - присвоєння регістру або провіднику.
`Select = ParityOdd(RegAX) & RegAX[0];`	В цьому присвоєнні виклик функції є одним з операндів. Виклик функції може бути використано в будь-якому місці, де може бути використано операнд такого типу.
`if (ParityOdd(AddressBus)) ...`	Такий виклик функції також допускається - умова оператора ***if*** - це вираз і в окремому випадку вираз може мати єдиний операнд.

Рис.15.25.

Системні завдання та функції

Крім надання можливостей користувачу з створення *завдань* та *функцій*, у Verilog також є набір визначених підпрограм. Існує більше сотні системних завдань, згрупованих в декілька категорій, найчастіше з яких використовуються *дисплейні завдання*. *Завдання файлового вводу-виводу* та *завдання керування симуляцією та синхронізацією*.

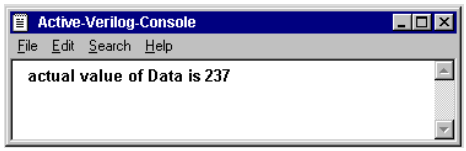
Системні завдання та функції можна легко відрізнити від тих, що визначені користувачем через символ долара '***$***', який вказується перед їх іменами.

Оскільки системні завдання та функції вважаються частиною мови, користувачу не потрібно їх визначати і вони можуть використовуватись в довільному модулі.

Системні завдання використовуються для полегшення синтезу. Але самі вони не синтезуються.

```
reg [15:0] Data;

initial
  Data = 231;

  Data = 237;
```

```
initial
  $display ("actual value of Data is %d",Data);
```




```
reg [15:0] Data;
integer SimLogFile;

initial
  Data = 237;
```

```
initial
  begin
  SimLogFile = $fopen("simlog.txt");
  $fdisplay (SimLogFile,"actual value of Data is %d",Data);
```

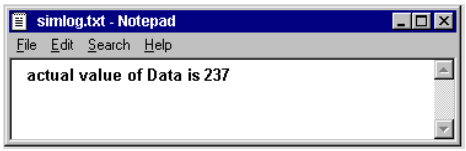


Рис.15.26.

16. Тестові стенди Verilog.

Структура тестового модуля

Елементи тестового стенда Verilog

Тестовий стенд Verilog – це просто інша специфікація із власним модулем. Але вона має декілька характерних властивостей:

- **модуль тестового стенду** не має портів,
- **реалізація компонента UUT** – відповідність між тестовим стендом (стимуляторами) і UUT задається через реалізацію компонента і структурну специфікацію,
- **стимулятори** – це набір сигналів, що декларуються всередині архітектури тестового стенду і присвоюються входам UUT в його реалізації; стимули визначаються або безпосередньо в модулі тестового стенду, або в окремому модулі, що реалізується в модулі тестового стенду,
- **спостереження виходів** – стимулятори визначають зміну входів UUT, а результати (зміни виходів UUT) відображаються за допомогою системних завдань, які, як правило, описуються в окремому блоці для підвищення читабельності.

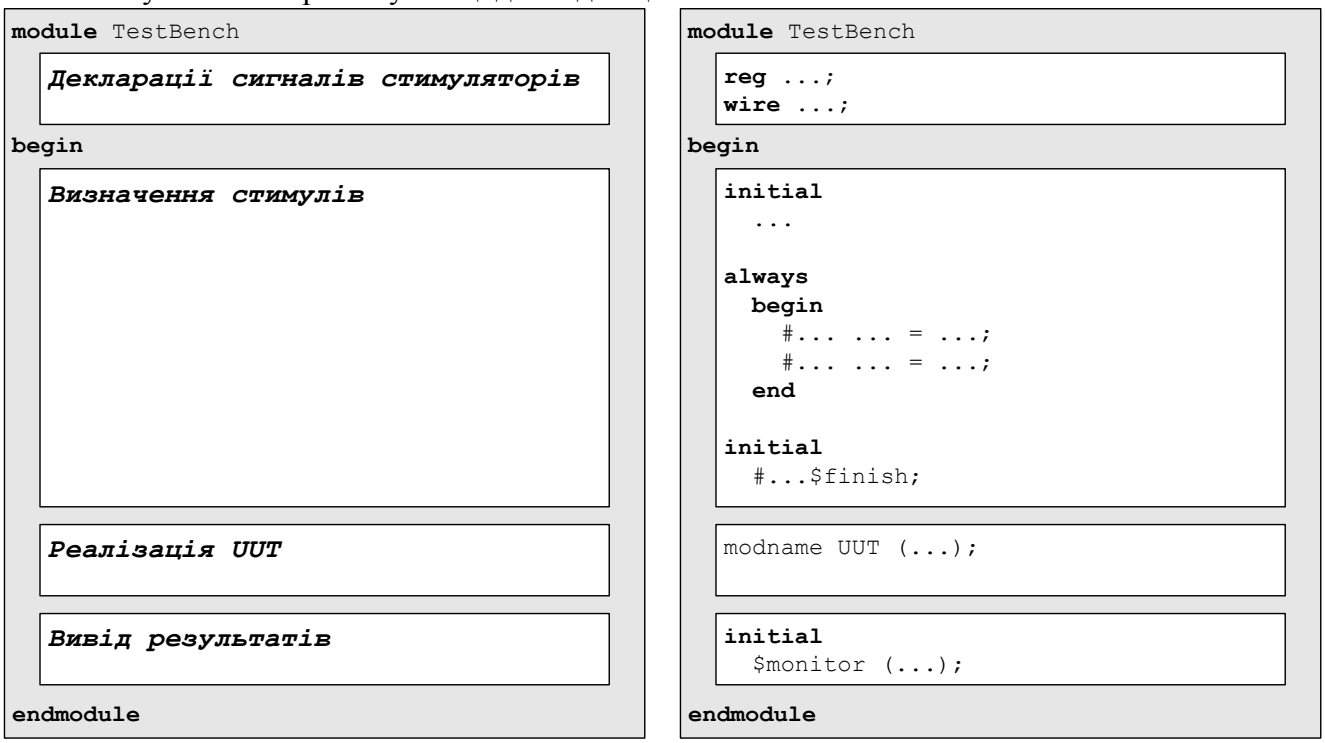


Рис.16.1.

Модуль тестового стенду

Тестовий стенд Verilog – це просто інша специфікація, написана у вигляді модуля, що не містить портів.

Причина цього дуже проста – тестовий стенд не є реальним пристроєм або системою, що має з'єднуватись із середовищем, отже він не потребує входів або виходів. Усі значення для вхідних портів UUT описуються всередині архітектури тестового стенду як стимулятори. Виходи спостерігаються за допомогою симулятора у вигляді часових діаграм і зберігаються в файлі.

Тестований пристрій

Система, верифікація якої буде проводитись за допомогою тестового стенду, не вимагає будь-яких модифікацій, або додаткових декларацій. Завдяки цьому тестові стенди можуть бути застосовані до довільних Verilog-специфікацій, навіть отриманих ззовні. Однак ніяких

модифікацій до специфікацій пристроїв не вноситься, оскільки вони проводяться після отримання результатів верифікації.

UUT має бути реалізований в архітектурі тестового стенду. Це виконується так само, як і в будь-якій структурній специфікації. Спочатку вказується ім'я модуля UUT, потім його унікальне ім'я і, нарешті, список асоціацій портів (впорядкований, або іменований). Портам UUT призначаються сигнали-стимулятори.

	Тестований пристрій (Unit Under Test)	***Тестовий стенд (Test Bench)***
Mux2To1	**module** Mux2To1 (Y,A,B,Sel); **output** Y; **input** A,B,Sel; ... **endmodule**	**module** TestBench; ... Mux2to1 UUT (.Y(...),.A(...),.B(...),.Sel(...)); ... **endmodule**
Reg8	**module** Reg8 (Q,D,En,Clk); **output** [7:0] Q; **reg** [7:0] Q; **input** [7:0] D; **input** En,Clk; ... **endmodule**	**module** TestBench; ... Reg8 UUT (.Q(...),.D(...),.En(...),.Clk(...)); ... **endmodule**
Dec2to4	**module** Dec2to4 (Outs,Ins); **output** [3:0] Outs; **input** [1:0] Ins; ... **endmodule**	**module** TestBench; ... Dec2to4 UUT (.Outs(...),.Ins(...)); ... **endmodule**
JK_flipflop	**module** JK_FF (Q,NQ,J,K,Clk); **output** Q,NQ; **reg** Q,NQ; **input** J,K,Clk; ... **endmodule**	**module** TestBench; ... JK_FF UUT (.Q(...),.NQ(...),.J(...),.K(...),.Clk(...)); ... **endmodule**

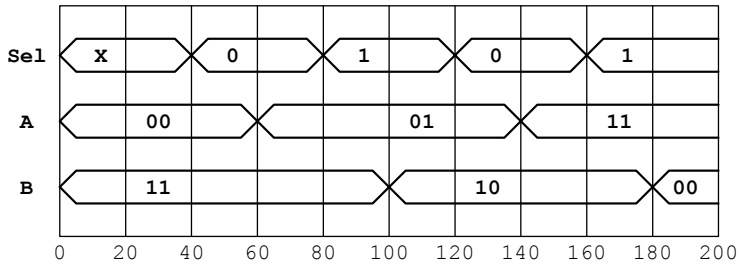
Рис.16.2.

Сигнали-стимулятори

Ядром кожного тестового стенду є набір стимуляторів – послідовність значень для кожного вхідного сигналу UUT, прив'язаних до часу. Оскільки тестовий стенд не з'єднується із середовищем за допомогою сигналів, всі стимулятори мають бути задекларовані всередині модуля тестового стенду і асоційовані з портами UUT.

Власне стимулятори (зміни сигналів, асоційованих з входами UUT) можуть бути описані безпосередньо в тестовому стенді, або в окремому модулі. В обох випадках найпростіше описувати їх в послідовних поведінкових ***initial***- або ***always***-блоках. Як правило, використовується додатковий ***initial***-блок із заданим системним завданням *$finish*. Він забезпечує завершення симуляції у визначений час. В протилежному випадку ***always***-блоки будуть весь час повторюватись.

Відповідність між стимулами і UUT задається за допомогою асоціацій в реалізації UUT.



UUT

```
module Mux2To1 (Y,A,B,Sel);
output Y;
input [1:0] A,B;
input Sel;

assign Y = Sel ? B : A;

endmodule
```

```
module TestBench
reg Sel;
reg [1:0] A,B;
wire Y;

Mux2to1 UUT (Y,A,B,Sel);
initial
  begin
    Sel = 1'bx; A = 2'b00; B = 2'b11;
  end
always
  begin
    #40 Sel = 1'b0; -- зміна в 40
    #20 A = 2'b10;  -- зміна в 60
    #20 Sel = 1'b1; -- зміна в 80
    #20 B = 2'b10;  -- зміна в 100
    #20 Sel = 1'b0; -- зміна в 120
    #20 A = 2'b11;  -- зміна в 140
    #20 Sel = 1'b1; -- зміна в 160
    #20 B = 2'b00;  -- зміна в 180
  end
initial
  #200 $finish;
endmodule
```

Рис.16.3