Міністерство освіти та науки України
Чорноморський національний університет імені Петра Могили

Я. М. Крайник

РОЗРОБКА ЦИФРОВИХ СХЕМ МОВОЮ VHDL
Основи розробки цифрових схем

Навчальний посібник
з дисципліни «Вбудовані системи»

ЧНУ
ВИДАВНИЦТВО
Миколаїв – 2025

УДК 004.03
К 77

Рекомендовано до друку рішенням Вченої ради Чорноморського національного університету імені Петра Могили (протокол № 3 від 27 лютого 2025 року).

Рецензенти:
О. В. Старкова , д-р техн. наук, професор, зав. каф. кібербезпеки та інформаційних технологій Харківського нац. економ. ун-ту ім. Семена Кузнеця.
С. А. Іванець, канд. техн. наук, доцент, декан інженерної школи Київської школи економіки.

Крайник Я. М.

К 77 Розробка цифрових схем мовою VHDL. Основи розробки цифрових схем : навч. посіб. з дисципліни «Вбудовані системи» / Я. М. Крайник. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2025. – 80 с.

ISBN 966-336-462-9

Навчальний посібник містить теоретичні основи та практичні рекомендації щодо початку роботи з мовою схемотехнічного опису VHDL у середовищі ModelSim. Наведені відомості також можуть використовуватись для інших середовищ розробки та симуляції для програмованих логічних інтегральних схем (ПЛІС).

Посібник також містить відомості щодо основних конструкцій мови та їх використання і особливостей. Кожен розділ супроводжується контрольними запитаннями. Посібник призначений для використання при вивченні дисципліни «Вбудовані системи» здобувачами вищої освіти за технічними спеціальностями.

УДК 004.03

ISBN 966-336-462-9

ЗМІСТ

ВСТУП

ModelSim є програмним продуктом для симуляції роботи цифрових схем, описаних за допомогою схемотехнічних мов. Дане програмне забезпечення широко використовується для реалізації великих проєктів, а можливість симуляції у даному пакеті підтримують такі вендори ринку ПЛІС як Xilinx та Altera.

Даний матеріал призначений для тих, хто починає ознайомлення із засобами розробки та моделювання схемотехніки ModelSim та мовою схемотехнічного опису VHDL. ModelSim є програмним продуктом фірми MentorGraphics і надає надзвичайно широкі можливості для роботи з проєктування цифрових схем мовою VHDL. У свою чергу, мова VHDL є однією з мов, які мають найбільше поширення серед розробників апаратного забезпечення. Її основною перевагою є те, що вона дозволяє отримати більший контроль над процесом формування цифрової схеми порівняно з іншими мовами схемотехнічного опису (наприклад, Verilog).

Робота з наведеними матеріалами передбачає також використання читачем інших навчальних ресурсів. Для полегшення сприйняття матеріалу значну кількість інформації опущено, тому для більш ретельного вивчення окремих тем рекомендуємо користуватись офіційною документацією на стандарт VHDL та ресурсами за тематикою ModelSim. Приклади, що наводяться, також є базовими та мають на меті показати реалізацію найбільш важливих типових компонентів, представити найбільш поширені рекомендації щодо написання коду опису схеми.

Наприкінці кожного розділу наведені тестові питання для самоперевірки та контролю засвоєння матеріалу. Кожне питання передбачає відповідь «Так» або «Ні».

Наведений навчальний матеріал призначений для використання при вивченні нормативної дисципліни «Вбудовані системи» для здобувачів спеціальності 123 Комп’ютерна інженерія, а також інших технічних спеціальностей.

1.ПРИЗНАЧЕННЯ ТА СЕРЕДОВИЩА ВИКОРИСТАННЯ МОВИ VHDL

1.1 Ознайомлення з середовищем моделювання ModelSim

ModelSim – середовище для симуляції проєктів, реалізованих за допомогою мов схемотехнічного опису (VHDL, Verilog та ін.). Розробником даного програмного продукту є фірма Mentor Graphics. Деякі версії є безкоштовними для використання – ModelSim Student Edition, ModelSim Altera Starter Edition, але за їх використання доступ користувача до певних функцій та підтримуваних обсягів проєкту є обмеженим.

ModelSim може працювати з одним проєктом або із вказаною директорією, в якій знаходяться файли з описом. Створення нового проєкту виконується за допомогою команди **File/New…/Project**, для якої необхідно вказати розташування нового проєкту. У тому випадку, якщо деякі з файлів вже готові до використання, то їх можна додати до проєкту за допомогою команди контекстного меню **Add to Project/Existing file** в області перегляду файлів проєкту. Вікно ModelSim з проєктом, що містить 5 файлів на мові VHDL, показано на рис. 1.

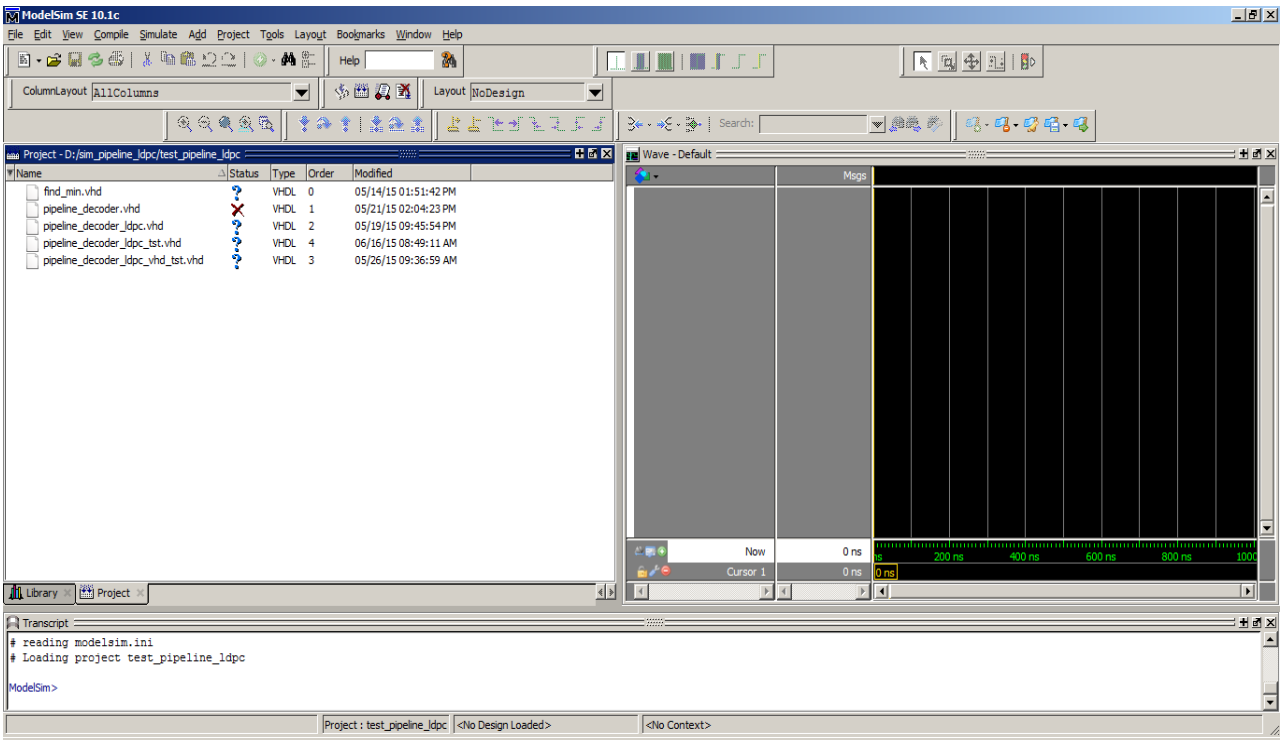


Рисунок 1 – Вікно ModelSim з робочим проєктом

Незважаючи на те, що ModelSim можна використовувати і як редактор файлів проєкту, проте краще використовувати інші редактори – Notepad++, Vim та ін.

Перед початком симуляції усі модулі, що використовуються при тестуванні, мають бути скомпільовані (у контекстному меню команда Compile та вибір необхідного пункту). Після того, як файл скомпільовано, його іконка у колонці **Status** змінюється так, як показано на рис. 2. Внесення змін до файлу потребує його повторної компіляції, інакше зміни не будуть застосовані.

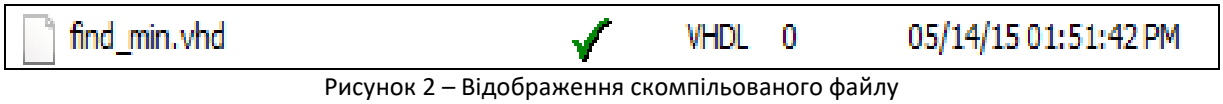


Рисунок 2 – Відображення скомпільованого файлу

У тому випадку, якщо всі файли скомпільовані, можна перейти до симуляції. Команда меню **Simulate/Start Simulation**. У результаті відкриється вікно вибору файлу для симуляції. У ньому необхідно відкрити пункт **work** (директорія, до якої за замовчуванням відносяться всі файли проєкту) на вкладці **Design** та обрати файл, що буде проходити тестування (рис. 3).

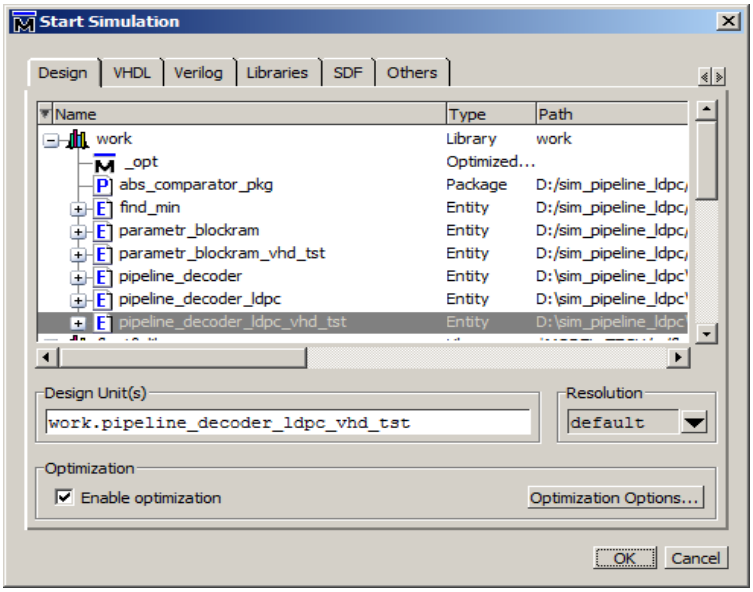


Рисунок 3 – Вибір файлу для тестування у вікні запуску симуляції

Запуск симуляції змінює компонування вікна програми та може відкривати нові вікна. Перегляд результатів симуляції відбувається за допомогою вікна **Wave** (рис. 4). У випадку, якщо вікно не відрилось одразу, його можна відкрити командою **View/Wave**.

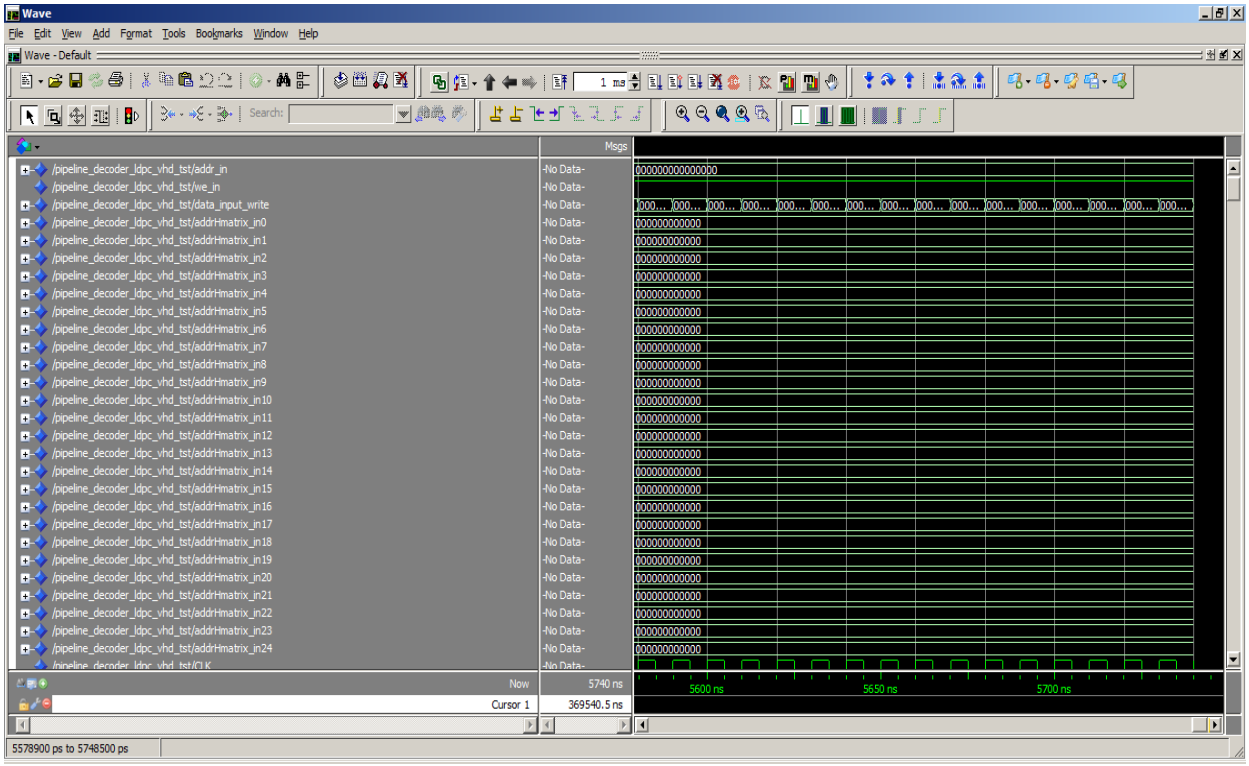


Рисунок 4 – Вікно **Wave**

Основними командами, що використовуються при симуляції, є запуск та зупинка. Вони розташовані на панелі інструментів (рис. 5).

Рисунок 5 – Кнопки команд управління симуляцією

Перша із вказаних на рис. 5 кнопок виконує перезапуск симуляції. Поле використовується для того, щоб вказати час симуляції при запуску за допомогою кнопки, що розташована праворуч від поля.

1.2 Середовище EDA Playground

EDA Playground – середовище для вивчення мов схемотехнічного опису, яке надає можливість перевірити роботу описаних цифрових схем без встановлення спеціалізованого програмного забезпечення, яке займає доволі багато місця.

Перш за все, для роботи у вебсередовищі необхідно перейти за наступним посиланням – *https://www.edaplayground.com/* – та зареєструватись. Реєстрація необхідна для того, щоб можна було зберігати проєкти у профілі. У профілі можна зберігати велику кількість проєктів. Рекомендований метод реєстрації – за допомогою акаунту Google.

Після проходження реєстрації будуть доступні 2 вкладки, а також панель налаштувань. Перша вкладка призначена для реалізації тесту – **testbench**. Друга – для реалізації модуля, який надає функціональність для тесту. У панелі налаштувань рекомендується встановити значення, які показані на рис. 6.

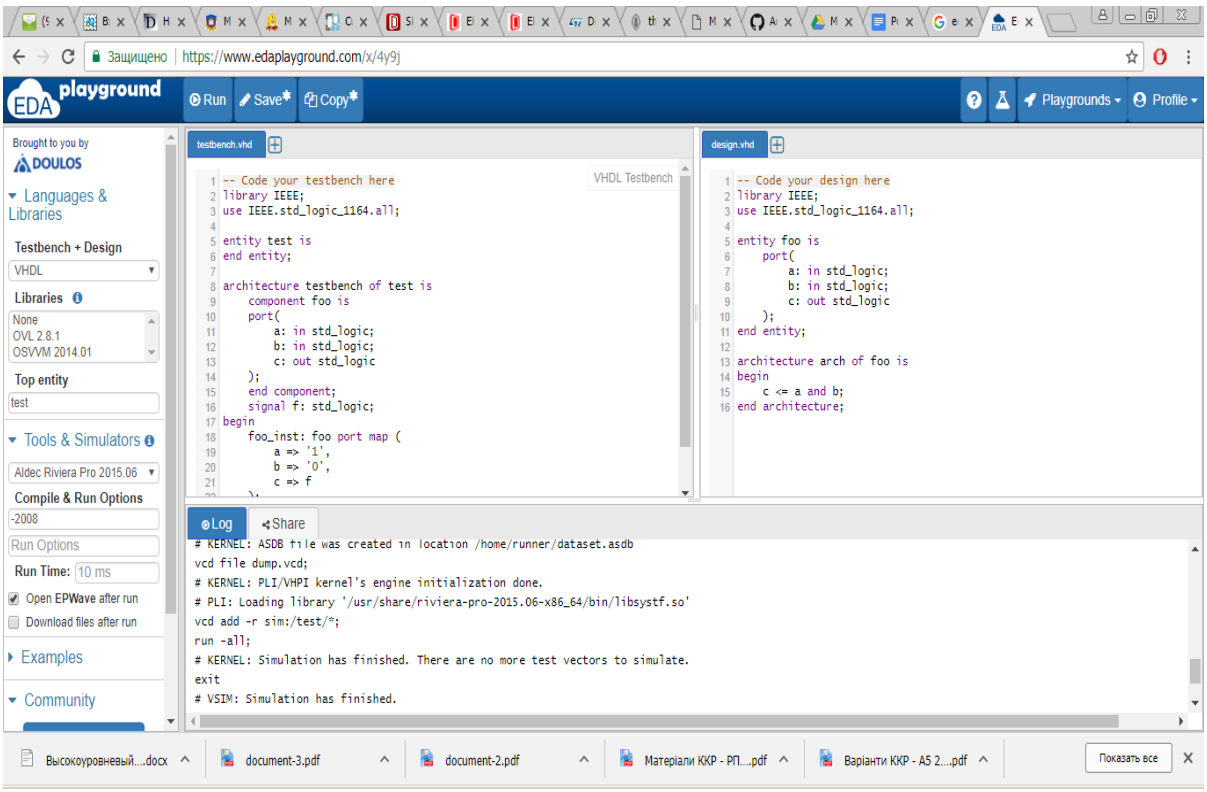


Рисунок 6 – Зовнішній вигляд середовища з рекомендованими налаштуваннями

Зверніть увагу, на те, що встановлена опція «**Open EPWave after run**». Дана опція відповідає за появу після запуску проєкту (кнопка **Run**) вікна, у якому буде показано візуальний результат виконання тесту (рис. 7).

Також важливим є те, що виконується саме файл тесту, тому необхідно, щоб на інстанційований модуль для тестування (**design-under-test**) у модулі тесту, який не має ні входів, ні виходів.

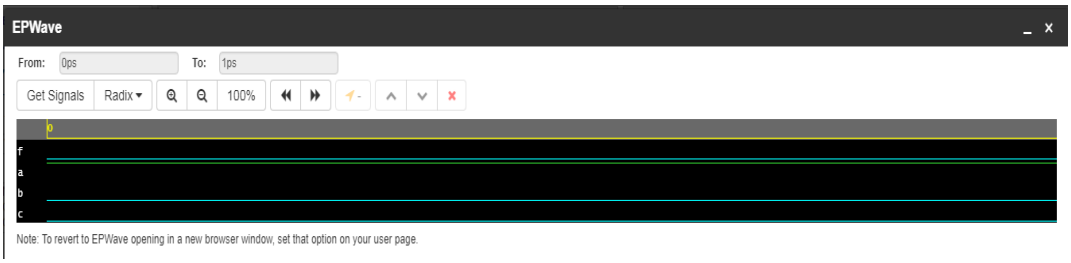


Рисунок 7 – Вікно EPWave

1.3 Мова схемотехнічного опису VHDL

VHDL (VHSIC Hardware Description Language, VHSIC – Very High Speed Integrated Circuit) – мова схемотехнічного опису, основними особливостями якої є типізація, наявність вбудованих типів, модульність. Мова не залежить від регістра символів.

Слід відзначити, що мова VHDL є достатньо простою: кількість операцій, типів, функцій є доволі малою порівняно з мовами програмування. Більше того, не всі типи VHDL є придатними для синтезу (створення опису на

рівні регістрів), тому вибір розробника скорочується до відносно невеликої множини засобів мови. Проте, така простота мови не відміняє те, що VHDL є потужним інструментом для створення опису роботи цифрових схем.

Основна мета мови схемотехнічного опису – отримання цифрової логічної схеми, що забезпечує виконання необхідних функцій. Оскільки метою є створення схеми, то мову слід розглядати як засіб схемотехнічного проєктування: використані інструкції опису відображаються як логічні елементи та з'єднання між елементами. Перевагою такого способу проєктування перед візуальним є те, що:

1) складні цифрові схеми можуть містити велику кількість компонентів та зв'язків, тому поелементне редагування за допомогою графічних засобів може займати багато часу. Натомість, схемотехнічний опис може, за умови правильного проєктування, спрощувати такі дії;

2) наявна можливість створення опису, який не залежить від набору інструментів, що використовуються, в той час, як графічні засоби часто не можуть бути використані поза спеціалізованими програмними засобами. Завдяки тому, що існує стандарт для мови VHDL, доступний набір інструкцій, що підтримуються всіма виробниками, його використання дозволяє зменшити залежність від програмних засобів. Це означає, що є можливість повторного використання компонентів;

3) присутня можливість більш тонкого налаштування та редагування компонентів системи під вирішувану задачу. Даний момент є, мабуть, вирішальним, оскільки надає можливість оптимізувати проєкт під задані вимоги.

Проєктування за допомогою графічних засобів передбачає використання додаткового програмного забезпечення (наприклад, Xilinx ISE WebPack), яке підтримує таку функціональність. Доцільним використання такого підходу є лише для структурного з'єднання компонентів. Для реалізації критичних частин проєкту доцільно використовувати безпосередньо створення схемотехнічного опису за допомогою редактора.

Тим не менш, часто два описані підходи поєднуються та використовуються в проєктах одночасно.

Призначення мови VHDL – створення опису для отримання певних апаратних функцій (прийом сигналів, передачі, паралельна обробка та ін.). Використовувати реалізовані апаратні функції можна з використанням програмованих логічних інтегральних схем (ПЛІС). Сучасні ПЛІС можуть бути запрограмовані велику кількість разів, відповідно, функції, які вона виконує, можуть змінюватись. ПЛІС містить великий набір базових компонентів (регістри, мультиплексори, LookUp Tables, з'єднувачі між ними та ін.), який на основі опису конфігурується таким чином, щоб виконувати описані дії. У результаті конфігурації з простих базових елементів та зв'язків між ними утворюється готовий пристрій.

Варто також зазначити, що кількість ресурсів для різних ПЛІС є різною, тому не кожний опис можливо використовувати для конкретної мікросхеми ПЛІС. Обмеження щодо ресурсів слід враховувати на стадії проєктування пристрою.

1.4 Рекомендації для переходу від підходу ООП до VHDL

Дана частина орієнтована на читачів, які мають досвід роботи з мовами програмування, що підтримують об'єктно-орієнтовану парадигму програмування, та призначення для того, щоб співставити відомі засоби мов програмування та засоби мови VHDL.

Не дивлячись на те, що за своєю суттю алгоритмічні мови програмування та мови опису схемотехніки є різними, вони мають також і спільні підходи, тому для спрощення переходу від ООП-підходу до рівня схемотехніки проведемо перехід між ними.

Першим, на що слід звернути увагу, є опис модулю. У VHDL він складається з двох частин: **інтерфейсу** та опису архітектури. У мовах програмування з підходом на основі ООП інтерфейс є базовим поняттям, що представляє собою набір сигнатур методів, що необхідно реалізувати у класі. Сам інтерфейс при цьому не містить реалізації функціональності, а лише декларує її. З цього погляду інтерфейси в ООП і в VHDL є надзвичайно близькими: інтерфейс з погляду схемотехніки задає вигляд модуля для інших модулів ззовні (опис **port**) на основі сигналів. (Зважаючи на це, можна сказати, що VHDL дозволяє повністю реалізувати принцип програмування через інтерфейси, оскільки інтерфейс є обов'язковою частиною опису модуля).

Оскільки самі по собі інтерфейси не містять реалізації функціональності, то має існувати можливість для реалізації. В ООП це відбувається через **класи, що реалізують інтерфейс**. Це означає, що клас повинен надати реалізацію всіх методів, які наявні в інтерфейсі. У VHDL аналогом цього виступає опис архітектури. Він описує логічну функцію, що буде виконувати модуль, тобто внутрішню реалізацію.

Ще одним базовим поняттям, за яким можна провести порівняння, є **поліморфізм**. В ООП-підході він означає різну поведінку коду залежно від контексту і надає широкі можливості для реалізації даної концепції: перевизначення функцій, віртуальні функції та ін. У VHDL це забезпечується тим, що інтерфейс може мати декілька архітектурних тіл, що реалізують даний інтерфейс. Практичне значення цього полягає у тому, що можна порізному виконати реалізацію модуля залежно від потреб схеми. Найпростішим прикладом є лічильник із прямим та зворотнім відліком: інтерфейс для лічильника може використовуватися один, а реалізацій – декілька.

Крім того, можна зауважити, що на основі понять інтерфейсу та архітектури реалізується своєрідна концепція **наслідування**, що також широко використовується в ООП.

Одним з основних понять в ООП є поняття **композиції**, що означає включення об'єктів одного класу в об'єкти іншого класу. У VHDL засобом реалізації цього є структурний підхід до опису схемотехніки, коли модулі створюються у явному вигляді у тілі архітектурного опису.

Цікаво також провести аналогії між підходом, який в ООП отримав назву Dependency Injection (DI), та використанням конфігурацій у VHDL. DI в перекладі означає внесення залежностей і дозволяє визначити, об'єкт якого саме типу слід використати на місці інтерфейсу, абстрактного класу. Частіше за все це робиться за допомогою спеціального об'єкта, DI-контейнера, що виконує дані дії. Часто це реалізується за допомогою спеціального фреймворка (наприклад, Spring, Java EE та велика кількість інших), що дозволяє зберігати **конфігураційні** налаштування проєкту у xml-файлі. Для подібних цілей у VHDL слугує блок конфігурації (**configuration**). Він дозволяє розробнику в явному вигляді вказати відношення між архітектурою та іншими модулями, що використовуються у ній.

Тим, хто працював з мовами C++ або Java, напевне, знайомі поняття шаблонів або **узагальнень** (**templates** та **generics**). Вони являють собою зручний механізм для параметризації коду відповідно до типу. Схожий механізм у VHDL отримав назву **узагальнених параметрів** (**generic**). Тим не менш, його можливості суттєво поступаються вищезгаданим інструментам з мов програмування. Зазвичай, можливості таких параметрів рекомендується обмежувати вказанням цілочислових параметрів, що використовуються для налаштування розміру шин модуля, відповідно до даного параметру.

Як бачимо, не дивлячись на те, що мови програмування та мова схемотехнічного опису VHDL мають різне функціональне призначення, про що також необхідно пам'ятати, між ними можна провести багато паралелей, знайти схожі риси. Більше того, у VHDL на рівні мови закладені можливості, що умовах програмування забезпечуються сторонніми бібліотеками.

Контрольні питання

1. Декларація entity визначає лише ім'я проєктованої цифрової системи та специфікацію її портів вводу/виводу?
2. Структурний опис може бути ієрархічним?
3. Опис усієї проєктованої цифрової системи може бути не ієрархічним?
4. Чи допускається не більше восьми рівнів ієрархії опису VHDL-проєкту?
5. Одна декларація entity може відповідати кільком архітектурним (architecture) описам?
6. Кожна цифрова система має інтерфейс (entity)?
7. Перш ніж використовувати пакет STANDARD, декларація entity має бути обов'язково зумовлена операторами library, use?
8. Кожна цифрова система, що описується в VHDL, описується парою entity та architecture?
9. Пакет (package) – це компактний спосіб зберігання логічно пов'язаних описів entity та architecture?
10. Одна декларація архітектури може відповідати декільком деклараціям entity?
11. Чи є конфігурація (configuration) модулем (VHDL-файлом) проєкту?
12. Чи конфігурація є об'єктом мови VHDL?
13. Пакет є модулем проєкту?
14. Пакет є об'єктом VHDL мови?
15. Чи сигнал є об'єктом мови VHDL?
16. Чи сигнал є модулем проєкту?
17. Чи є об'єктом мови VHDL?
18. Чи є Константа модулем проєкту?
19. Чи є об'єктом мови VHDL?
20. Чи є модулем проєкту?
21. Є лише три об'єкти мови VHDL: сигнал, змінна, константа?
22. Чи є тільки п'ять модулів проєкту (VHDL-файлів) цифрової системи: entity, architecture, configuration, package, package body (тіло пакета)?
23. Чи існує можливість дізнатися попереднє значення змінної?
24. Порядок нумерації бітів у векторі типу bit_vector завжди є однаковим?
25. Внутрішні сигнали системи визначаються у декларації entity?

2.ОСНОВНІ ПОНЯТТЯ ТА СИНТАКСИС МОВИ VHDL

2.1 Початок роботи з VHDL

Для початку ознайомлення з VHDL розглянемо простий компонент (рис. 8), на основі якого ознайомимось з основними структурними складовими опису.

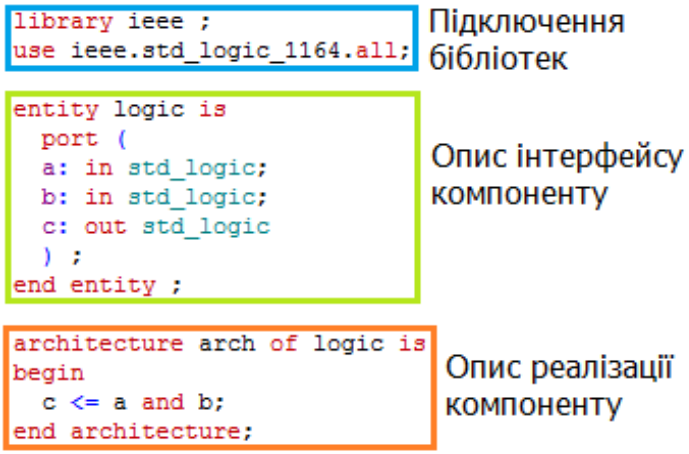


Рисунок 8 – Опис простого компонента на мові VHDL

Опис складається з трьох частин, які розміщуються в окремому файлі. Файл має розширення **.vhd**. На початку файлу розміщується секція підключення бібліотек та пакетів. Вони містять описи типів, які необхідні для реалізації компонента. Пакет **std_logic_1164** у складі бібліотеки **ieee** серед інших містить основний тип – **std_logic**.

Опис інтерфейсу (блок **entity**) включає в себе опис вхідних та вихідних сигналів, які розміщуються в секції **port**. У прикладі на рис. 8 тип входів та виходів вказаний як **std_logic**. Даний тип є ключовим для мови VHDL. Більш детально властивості даного типу пояснюються далі. Відповідно до прикладу, компонент має 2 вхідні порти (***a*** та ***b***) та один вихідний порт (***c***).

Опис реалізації виконується в блоці **architecture**. Саме в ньому визначається, які функції виконує компонент. У прикладі реалізація передбачає подачу на вихід ***c*** значення операції побітового І для сигналів ***b*** та ***c***. Незважаючи на те, що опис виглядає достатньо простим, у ньому використовується оператор присвоєння (<=), який дозволяє задати значення виходу. Даний оператор є важливим, оскільки він широко використовується за реалізації **асинхронної** логіки (такої, для якої не потрібен сигнал тактування).

Для програмістів! *У VHDL так само, як і в інших мовах, наявні коментарі. Усі символи в рядку після послідовності "—" (два дефіси), вважаються закоментованими. Багаторядкові коментарі відсутні.*

Як вже зазначалось, інструкції VHDL обробляються спеціальними програмами (синтезатор, трасувальник та ін.) та відображаються в мікросхему. Слід зазначити, що не всі типи та інструкції можуть будуть використані при генерації схеми – деякі можна застосовувати лише для симуляції та тестування. Обробка програмами коду на основі VHDL у спрощеному вигляді відповідає процесу компіляції для мов програмування. Також важливим, що для процесу симуляції не обов'язково виконувати усі стадії обробки опису – достатньо виконати компіляцію.

Результатом обробки коду всіма інструментами є бітовий файл, який завантажується у мікросхему ПЛІС.

2.2 Моделювання роботи компонента

За замовчуванням сигнали матимуть значення 'U' (від англ. Uninitialized), оскільки їм ще не присвоєне жодне значення. Присвоїти значення входу можна за допомогою контекстного меню:

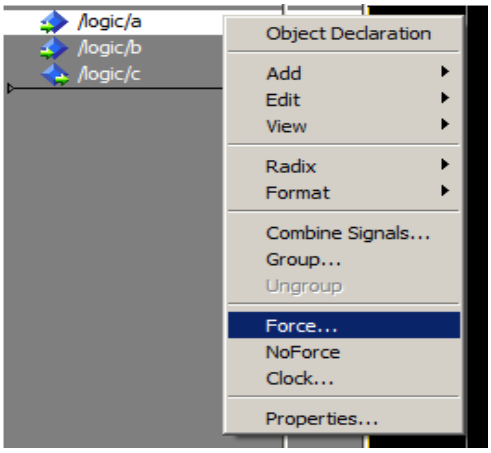


Рисунок 9 – Контекстне меню для керування станом сигналу

У діалоговому вікні необхідно вказати нове значення входу. Для прикладу почнемо процес моделювання із встановлення вхідних сигналів у ‘0’ (рис. 10). Присвоєння значення вихідному сигналу не має сенсу, оскільки він обчислюється на основі вхідних.

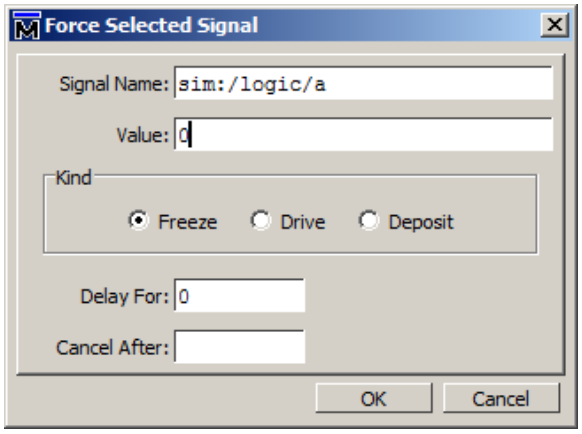


Рисунок 10 – Присвоєння значення сигналу

Для того, щоб промоделювати роботу компонента протягом певного інтервалу, встановимо значення інтервалу симуляції рівним 100 нс (у полі на рис. 5 ввести 100 ns) та натиснемо кнопку Run ( ). Таким чином далі необхідно вказати значення для всіх можливих комбінацій вхідних сигналів (наприклад, як на рис. 11) та перевірити роботу компонента.

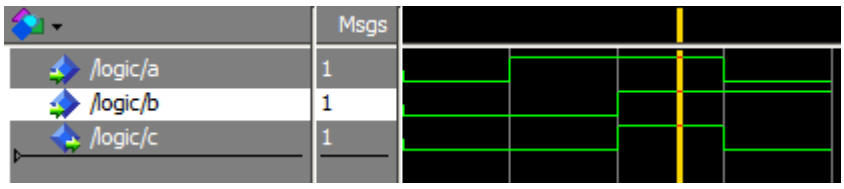


Рисунок 11 – Діаграма сигналів

Наведений приклад є дуже простим, тому моделювання його роботи можна проводити вручну. Тим не менш, у випадку великих проєктів ручне моделювання займе невиправдано багато часу. Для моделювання великих проєктів або їх складових реалізуються файли тестових оточень (англ. термін – **testbench**). Тестове оточення інстанціює компонент та подає на нього набір вхідних імпульсів, що дозволяє промоделювати його роботу в різних ситуаціях. Файли тестових оточень передбачають їх багатократне повторне застосування в ході роботи над проєктом.

2.3 Налаштування представлення сигналів у ModelSim

Налаштування сигналів для простих описів схем може проводитись вручну. Тим не меш, необхідно виконати певну мінімально необхідну кількість дій для того, щоб налаштувати виведення сигналів у симуляторі. Якщо роботу схеми планується перевіряти більше ніж один раз, то необхідно зберігати налаштування симуляції.

Налаштування формату представлення сигналів здійснюється командою контекстного меню Radix\<Signal type> у вікні Wave (рис. 12). Для вибору доступно декілька форматів, основними з яких є числові формати: зі знаком, без знаку, шістнадцятковий, вісімковий, двійковий. Інші формати є менш вживаними у використанні, проте при реалізації спеціальних задач можуть бути корисними (особливо рядкове представлення для випадків обробки текстових вхідних даних).

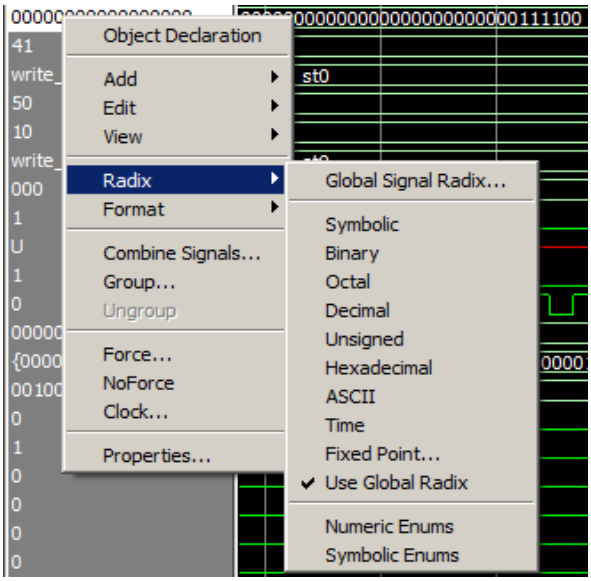


Рисунок 12 – Вибір базису представлення сигналу

Для збереження формату представлення симуляції у вікні Wave ModelSim використовує **do**-файли. **Do**-файли містять команди для додавання сигналів у вікно симуляції та управління їх представленням. Створити такий **do**-файл можна за допомогою команди меню File/Save Format у вікні Wave.

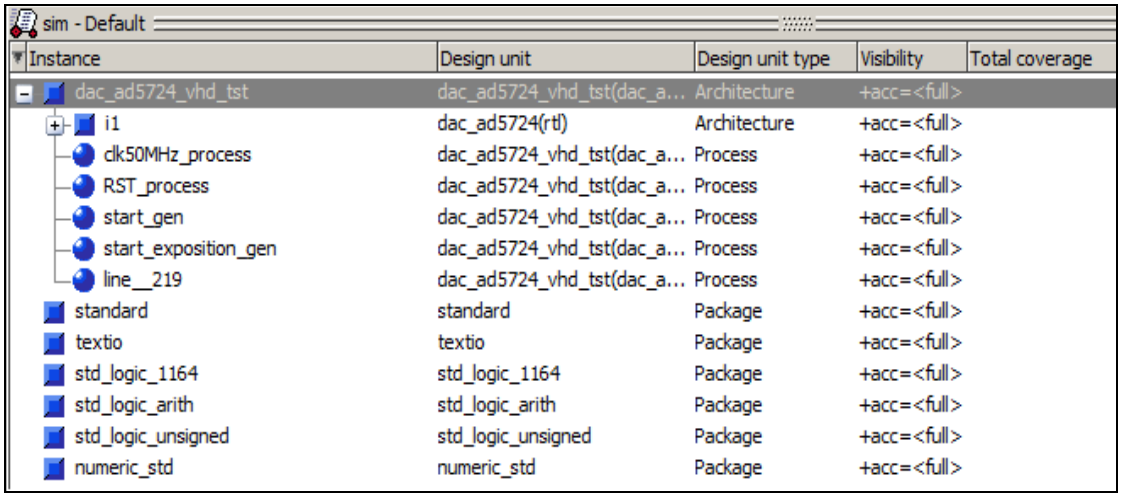
Варто зазначити, що в ModelSim **do**-файли мають набагато ширші можливості, аніж збереження налаштувань симуляції. Вони містять команди мови **Tcl**, які дозволяють виконувати дії з управління проєктом та налаштування усього середовища моделювання: компіляція файлів, запуск симуляції та ін.

Збереження налаштувань представлення сигналів для симуляції є обов'язковим при роботі з великими проєктами.

2.4 Об'єкти для моделювання

Для того, щоб розробник мав можливість проводити тестування і перевірку для об'єктів на будь-якому рівні, у ModelSim наявні вікна sim та Objects. У випадку, якщо дані вікна не є доступними, то увімкнути їх можна командами меню View\Structure та View\Objects відповідно.

У першому з них доступною для перегляду є ієрархічне представлення структури модулю, для якого проводиться тестування. Доступними для перегляду є усі елементарні складові опису модуля та його підмодулів.



Instance	Design unit	Design unit type	Visibility	Total coverage
dac_ad5724_vhd_tst	dac_ad5724_vhd_tst(dac_a...	Architecture	+acc=<full>	
i1	dac_ad5724(rtl)	Architecture	+acc=<full>	
clk50MHz_process	dac_ad5724_vhd_tst(dac_a...	Process	+acc=<full>	
RST_process	dac_ad5724_vhd_tst(dac_a...	Process	+acc=<full>	
start_gen	dac_ad5724_vhd_tst(dac_a...	Process	+acc=<full>	
start_exposition_gen	dac_ad5724_vhd_tst(dac_a...	Process	+acc=<full>	
line__219	dac_ad5724_vhd_tst(dac_a...	Process	+acc=<full>	
standard	standard	Package	+acc=<full>	
textio	textio	Package	+acc=<full>	
std_logic_1164	std_logic_1164	Package	+acc=<full>	
std_logic_arith	std_logic_arith	Package	+acc=<full>	
std_logic_unsigned	std_logic_unsigned	Package	+acc=<full>	
numeric_std	numeric_std	Package	+acc=<full>	

Рисунок 13 – Ієрархічна структура симуляції

Вікно **Objects** зв'язане з вікном **sim**. Так, вибір елементу у вікні sim призводить до відображення компонентів, які стосуються опису безпосередньо даного об'єкта. Найважливішими елементами, які відображаються у даному вікні і з якими найчастіше необхідно працювати, є сигнали, що входять до опису. Окрім того, що у вікні доступні назви сигналів, у ньому також відображається поточне значення конкретного сигналу.

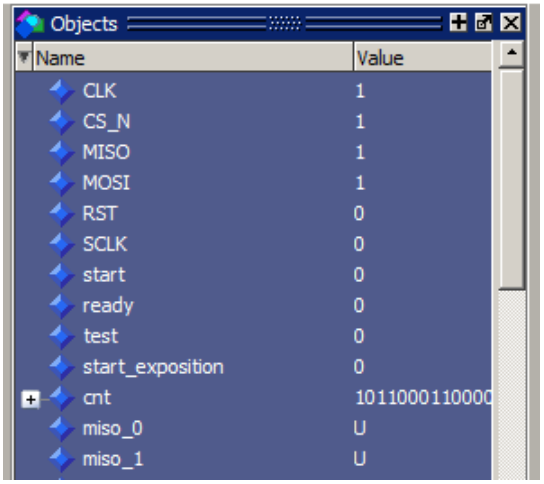


Рисунок 14 – Сигнали об'єкта симуляції

2.5 Встановлення міток часу (курсорів)

Для того, щоб визначити, у який саме момент моделювання відбулась конкретна подія, важливим є використання часових курсорів. Курсори доступні у вікні **Wave**. За замовчуванням у вікні доступний один курсор, проте їх кількість може змінюватись користувачем. Курсор у вікні **Wave** представлений жовтою вертикальною лінією, яка займає усю висоту області відображення сигналів.

Для того, щоб проводити вимірювання того, яку частину модельного часу займає окрема операція, зручним є використання двох курсорів. Додати курсор можна використовуючи панель інструментів у нижньому лівому куті вікна **Wave**.

Рисунок 15 – Меню додавання курсору

Після натискання на іконку «+» у вікні з'явиться ще один курсор. Для кожного курсору у робочій області передбачені виділені елементи керування. З їх допомогою можна задати точне розташування курсору, вказавши часову мітку, на яку він має переміститись. Також можливе видалення курсору з робочої області у тих ситуаціях, коли він стає непотрібним. Різниця між положеннями двох курсорів виводитиметься у нижній області відображення, як показано на рис. 16.

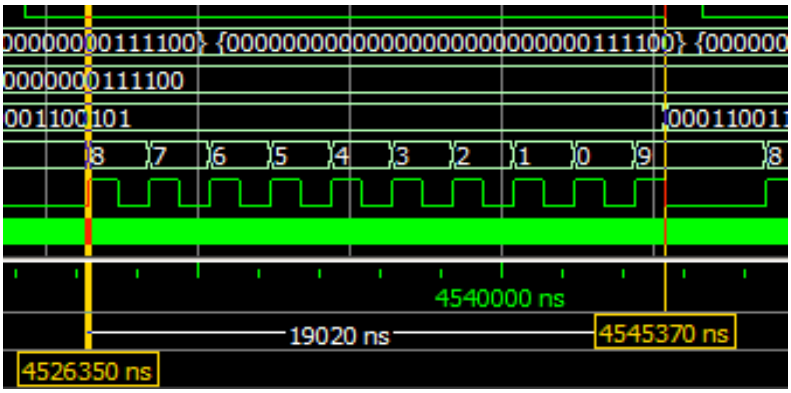


Рисунок 16 – Вимірювання модельного часу за допомогою курсорів

У тому випадку, коли у робочій області знаходяться одразу декілька курсорів, клік мишкою по робочій області означатиме переміщення одного з них. Переміщується той курсор, позиція якого була ближче до позиції кліку у робочій області. Це варто враховувати для того, щоб не виставляти позиції курсорів спочатку після нето-

чного вказання позиції для одного з них. Для того, щоб однозначно обирати курсор, для якого задається позиція, можна виділити необхідний курсор безпосередньо у нижній частині робочої області та переміщувати виділений курсор.

Також переміщенням курсорів можна керувати з використанням виділених елементів керування, розташованих на панелі інструментів. Дані елементи керування дозволяють переміщуватись до наступної або попередньої зміни стану сигналу, а також за фронтом або за спадом сигналу конкретно.

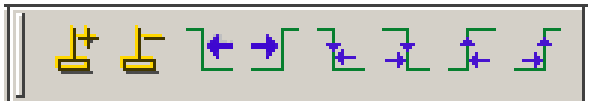


Рисунок 17 – Меню керування курсорами

Також на даній панелі інструментів доступні команди для того, щоб додати або видалити курсор.

Курсори надзвичайно зручні у використанні для тих ситуацій, коли необхідно оцінити, скільки часу займе виконання операції на реальній мікросхемі при заданій тактовій частоті. Також вони дозволяють більш детально дослідити послідовності сигналів, які генеруються у процесі моделювання.

2.6 Стилі опису

При описі схемотехніки з використанням VHDL прийнято розділяти стиль опису на наступні типи:

1) структурний (англ. structural) – передбачає використання інструкцій, які створюють компоненти, та забезпечення з'єднань між компонентами;

2) потоку даних (англ. dataflow) – описує мікросхему як функцію вихідного сигналу в залежності від вхідного. Даний стиль опису використовується для опису логіки, яка працює асинхронно;

3) поведінковий (англ. behavioral) – використовує інструкції всередині блоку **process** для опису мікросхеми; з розвитком засобів проведення синтезу мікросхеми саме поведінковий стиль є основним для написання більшості складових проєкту.

Крім того, можливий також змішаний стиль опису, який передбачає поєднання кількох вищезазначених стилів.

Перший розглянутий приклад відноситься до опису в стилі потоку даних. Інші стилі та їх комбінації будуть розглянуті далі.

2.7 Типи std_logic та std_logic_vector

Тип **std_logic** є типом, який призначений для моделювання схемотехнічної логіки. На відміну від типу **bit**, який може приймати лише 2 значення ('0' та '1'), змінна даного типу має одразу 9 можливих варіантів значень. Основними з них є:

- '1' та '0' (відповідають значенням типу bit);
- 'U' (позначає неініціалізоване значення);
- 'X' (вказує на наявність конфліктів при присвоєнні значень);
- 'Z' (стан нескінченого імпедансу; означає, що сигнал відсутній).

Тип **std_logic_vector** є масивом значень **std_logic**. Вхід або вихід типу std_logic_vector можна описати наступним чином:

bus: in std_logic_vector(7 downto 0).

Таким чином для компонента як вхід вказується 8-бітна шина. Розробник має можливість вказати варіант індексації. Індексація, яка наведена в прикладі? передбачає те, що найбільший значущий біт розташовуватиметься у позиції під індексом '7' і т. д. Саме такий опис рекомендується до застосування, оскільки він співпадає зі зручним та звичним для людини представленням. Тим не менш, можливий також інший варіант індексації:

bus: in std_logic_vector(0 to 7).

У цьому випадку найбільш значущий біт знаходитиметься під індексом '0'.

Дані типи є важливими тому, що саме з їх використанням рекомендується описувати з'єднання між компонентами схеми, порти компонентів. У випадку використання іншого типу його бажано привести до даних типів.

Контрольні питання

1. Чи можливо дізнатися попереднє значення сигналу?
2. Вихідні сигнали системи можуть визначатися у розділі декларацій архітектурного тіла?
3. Внутрішні сигнали системи визначаються розділ декларацій архітектурного тіла?
4. Декларація вхідного сигналу складається з імені сигналу, типу, напряму (in)?
5. Декларація вихідного сигналу складається з імені сигналу, типу, напряму (out)?
6. Декларація внутрішнього сигналу складається з імені сигналу та типу?

7. Всі сигнали, описані в entity, є видимими у всіх архітектурах, пов’язаних з entity?
8. Шина та вектор два позначення одного й того ж поняття (концепції) у VHDL?
9. Ліва межа порядку бітів у векторі завжди повинна бути меншою за його правий кордон?
10. Кожен порт повинен мати специфікацію свого напряму (наприклад, in, out, inout)?
11. Символом & позначається логічна операція І?
12. Усі сигнали системи описуються у її декларації entity?
13. Ім’я декларації entity може бути записано після оператора end цієї декларації entity?
14. Порт повинен бути або лише вхідним, або лише вихідним?
15. Чи значення настроюваного параметра (generic) може динамічно змінюватися в процесі моделювання?
16. Коментар починається і закінчується подвійним дефісом (--)?
17. Чи є у пакеті STANDARD декларація типу byte?
18. Параметр, що настроюється (generic) дозволяє задати введення постійної величини (константи) в проєктовану систему ззовні її?
19. Настроюваний параметр може бути використаний для специфікації портів, наприклад, для завдання ширини шин?
20. Маючи на увазі модельний час, чи можна сказати, що оператор a<=b?
21. Маючи на увазі фізичний час, можна сказати, що система моделювання (комп’ютерна програма) виконує оператор a<=b?
22. Кожен параметр, що настроюється, повинен бути визначений разом зі своїм напрямком?
23. Кожна декларація entity має закінчуватись ключовим словом end?
24. Порт є сигналом?
25. Чи може змінна типу integer бути гарним вхідним портом?

3.ОРГАНІЗАЦІЯ ІЄРАРХІЧНОЇ СТРУКТУРИ ПРОЄКТУ ТА РЕАЛІЗАЦІЯ ПРОЦЕСІВ

3.1 Структурний підхід до опису

Структурний стиль опису є необхідним при організації ієрархічної структури проєкту. У такому випадку проєкт може складатися з компонентів різних рівнів, які вирішують відповідні задачі. Компоненти більш високого рівня включають у себе компоненти нижчого рівня.

Розглянемо використання структурного опису на прикладі. Додамо до проєкту новий компонент, опис якого виглядатиме таким чином:

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

entity simple_or is
  port (
  a: in std_logic;
  b: in std_logic;
  c: out std_logic
  );
end entity;

architecture arch of simple_or is
begin
  c <= a or b;
end architecture;
```

Рисунок 18 – Код для реалізації логічного компонента АБО у VHDL

Даний компонент виконуватиме операцію побітового АБО над двома вхідними сигналами.

Створимо новий компонент на основі двох наявних. Він матиме 4 входи та 1 вихід. Новий компонент дозволятиме перевірити таблицю істинності для наступного логічного виразу:
(a and b) xor (c or d).

Принципова схема для даного прикладу виглядатиме таким чином:

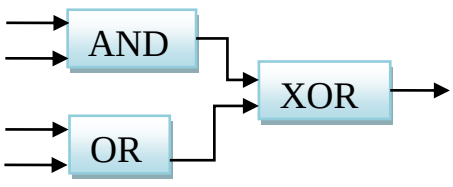


Рисунок 19 – Поєднання кількох логічних компонентів

При цьому реалізації побітових І та АБО вже наявні, тому їх можна використати у новому компоненті. Реалізація виключного АБО може проводитись безпосередньо в компонентах верхнього рівня. Варіант реалізації наведено на рис. 20.

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

entity structural is
  port (
   a : in std_logic;
   b : in std_logic;
   c : in std_logic;
   d : in std_logic;
   res : out std_logic
  ) ;
end entity;

architecture arch of structural is
  component logic is
    port (
    a: in std_logic;
    b: in std_logic;
    c: out std_logic
   ) ;
  end component;

  component simple_or is
    port (
    a: in std_logic;
    b: in std_logic;
    c: out std_logic
   ) ;
  end component;
  signal and_res : std_logic;
  signal or_res: std_logic;
begin
  and1 : logic port map (a => a, b => b, c => and_res);
  or1 : simple_or port map (a => c, b => d, c => or_res);
  res <= and_res xor or_res;
end architecture ;
```

Рисунок 20 – Приклад використання структурного підходу

Даний опис містить багато нових складових, порівняно з попередніми, тому зупинимось на них детальніше.

Для того, щоб використати компонент при реалізації нового, він має бути описаний у деклараційній секції архітектури (перед оператором **begin**). Опис компонента збігається з описом його інтерфейсу, за виключенням того, що ключове слово **entity** замінюється на **component**.

За створення компонента нижнього рівня відповідає інструкція

or1 : simple_or port map (a => c, b => d, c => or_res).

Дана інструкція передбачає створення компонента типу **simple_or** та співставлення його входів (***a*** і ***b***) із входами поточного компонента (***c*** та ***d***) та виходу із **сигналом**.

> **Для програмістів!** *Дана інструкція багато в чому аналогічна (наскільки це можливо для мови схемотехнічного опису та мови програмування) використанню конструктора в таких мовах програмування, як Java та ін.*

Сигнал використовується для передачі даних всередині архітектури компонента. Фактично, він є внутрішнім представленням стану компонента. Сигнали використовуються для реалізації як асинхронної логіки (наведено у прикладі), так і синхронної.

Опис сигналів виконується з використанням ключового слова **signal** та вказанням його типу. Також можлива ініціалізація сигналу початковим значенням. Сигнал доступний в межах усього тіла архітектури.

За співставлення входів компонентів відповідає інструкція **port map**. У дужках вказується вхід, який наявний в компоненті нижчого рівня (наприклад, ***a***) та вхід або сигнал поточного компонента (у даному випадку – ***c***). Очевидно, що імена при співставленні не обов'язково мають співпадати.

Для програмістів! *Дана частина опису дуже нагадує використання формальних та фактичних параметрів функції.*

Повертаючись до стилів опису, варто зазначити, що даний приклад використовує комбінований підхід, оскільки в ньому наявний як структурний підхід, так і підхід на основі потоку даних.

3.2 Побудова синхронних та асинхронних схем

До даного моменту декілька разів увага читача акцентувалась на тому, що за допомогою мови VHDL можна описувати синхронну та асинхронну логіку роботи схеми. Відмінність між ними полягає у наявності або відсутності тактового сигналу.

Елементною базою за реалізації асинхронних схем є тригери, що керуються за рівнем, – засувки (англ. термін «latch»). Базовими елементами для синхронних схем виступають тригери, що керуються тактовим сигналом, – регістри (англ. термін «flip-flop»). Фактично, регістри є синхронними D-тригерами.

З погляду реалізації синхронних тригерів у VHDL, регістри нерозривно пов'язані з процесами (блок **process**).

3.3 Опис синхронної логіки. Процеси та регістри

Розглянемо створення регістрів та використання процесів для їх реалізації на простому прикладі.

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

entity flip_flop is
  port(
    clk: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end entity;

architecture basic of flip_flop is
begin
  process(clk)
  begin
    if clk = '1' and clk'event then
      q <= d;
    end if;
  end process;
end architecture;
```

Оскільки схема є синхронною, то обов'язковою є наявність тактового вхідного сигналу (***clk***). Функціональність даного компонента проста: на вихід (***q***) подається значення, яке отримане на вході (***d***). Основною відмінністю від попередніх прикладів є розташування інструкції присвоєння значення сигналу всередині блоку **process**.

Опис процесу знаходиться в тілі архітектури. Тіло архітектури може містити опис кількох процесів. Процес має список чутливості (англ. «sensitive list»), який дозволяє задати сигнали зміна яких приводить до «виконання» тіла процесу. Обов'язково необхідно розміщувати усі вхідні сигнали, що перевіряються в умовних операторах всередині процесу. Неправильне формування списку чутливості процесу позначається на результаті симуляції. У версії мови VHDL-2008 вирішення даної проблеми спрощено за допомогою введення ключового слова **all**. У цьому випадку рядок у прикладі

process(clk)

слід замінити на

process(all).

Також необхідно змінити налаштування компілятора для даного файлу (рис. 21).

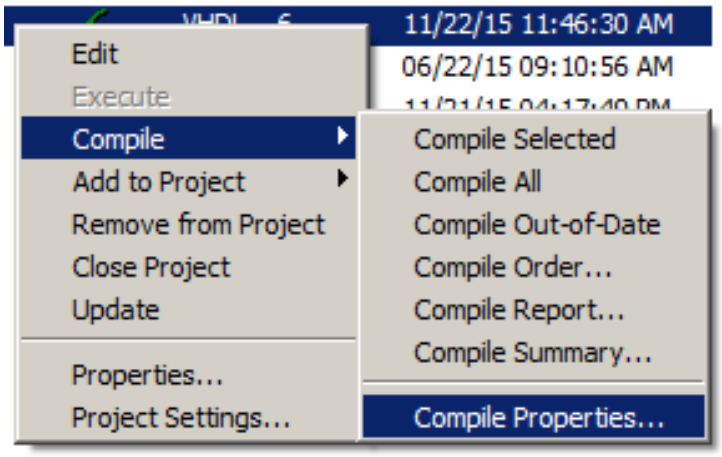


Рисунок 21 – Меню властивостей компіляції

Та обрати використання синтаксису VHDL-2008 для даного файлу (рис. 22).

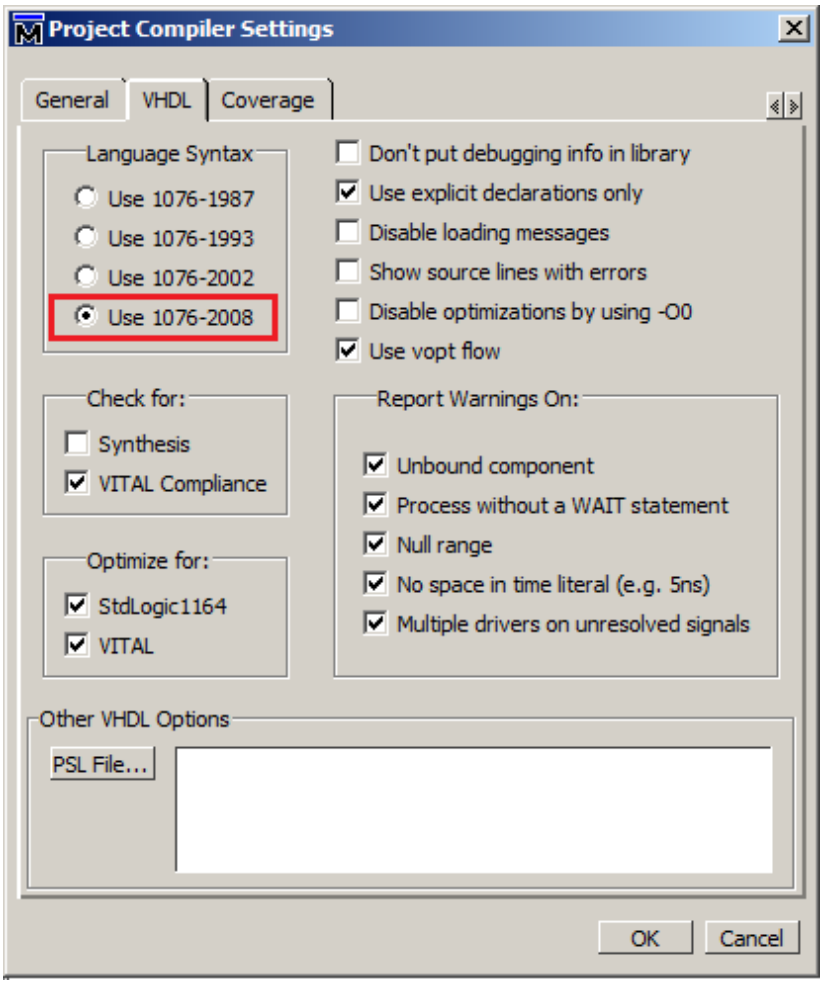


Рисунок 22 – Вибір опцій компіляції для VHDL

Після успішної компіляції файлу слід розпочати моделювання схеми. Оскільки в даній схемі наявний тактовий сигнал, то задання його значень вручну командою контекстного меню **Force** при моделюванні займе багато часу. Вирішення цієї проблеми полягає у використанні іншої команди контекстного меню – Clock. У діалоговому вікні (рис. 23) необхідно вказати налаштування тактового сигналу.

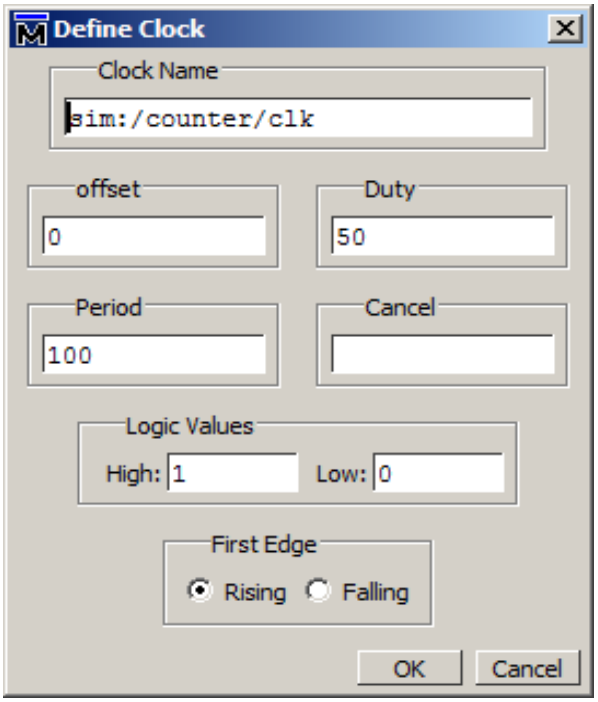


Рисунок 23 – Налаштування керування тактовим сигналом

Процес може не мати списку чутливості, проте тоді обов'язковою умовою є наявність у тілі процесу операції **wait**. Дана інструкція має велику кількість опцій, проте для опису компонентів, які призначені для синтезу, рекомендується використовувати форми

wait on <sensitivity list>

або

wait until <condition>.

Інструкція присвоєння значення сигналу знаходиться всередині умовного оператору **if**. Даний оператор відноситься до послідовних операторів. Також варто звернути увагу на умову, яка перевіряється:

if clk = '1' and clk'event then.

По-перше, умова складається з кількох частин, які з'єднані логічним оператором І. По-друге, даний підхід до перевірки настання фронту тактового імпульсу є найбільш поширеним і його можна вважати шаблоном для використання. Проводиться перевірка значення тактового сигналу (clk = '1'), а також перевірка настання події для даного сигналу за допомогою **атрибуту** (clk'event). З погляду опису тригерів, такий опис означатиме створення синхронного тригеру. Особливістю роботи даної схеми є затримка зміни сигналу на виході, яка становить один період тактового імпульсу (рис. 24). Це характерно для всіх регістрів, і це необхідно враховувати.

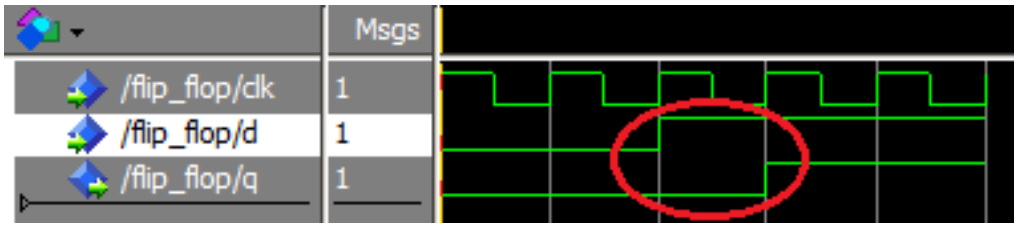


Рисунок 24 – Порівняння сигналів у регістрі

Саме таким чином у мові VHDL описуються регістри – елементи, що здатні зберігати значення.

Важливо! *Присвоєння значення сигналу всередині конструкції* ***if clk = '1' and clk'event then*** *… означає створення регістра.*

Незважаючи на те, що створення компонентів на основі синхронної логіки відбувається за допомогою процесів, вони можуть бути так само використані для опису асинхронної логіки. Наприклад, два наступні приклади згенерують однакові елементи:

```
c <= a and b;

process(a, b)
begin
  c <= a and b;
end process;
```

Для того, щоб робота схеми відбувалась коректно, до списку чутливості процесу додані сигнали ***a*** і ***b***. Це означає, що при будь-якій зміні цих сигналів, значення сигналу ***c*** буде обчислено з урахуванням цих змін.

Тим не менш, основне призначення процесів – створення синхронної логіки. Саме на базі синхронної логіки створюються великі проєкти зі значною кількістю складових компонентів. Усі компоненти в такому випадку об'єднуються в один частотний домен, у якому використовується одна робоча тактова частота. Дана частота подається на вхід усім компонентам схеми. Неможливість використання асинхронної логіки для великих схем пояснюється тим, що такий опис передбачатиме, що значна кількість сигналів має передаватись без затримки з входу на вихід, що призведе до значного використання ресурсів ПЛІС та неможливості реалізації такої схеми. Таким чином, затримка, що виникає на регістрах, є корисною для того, щоб схема могла бути реалізована.

Розглянутий приклад відноситься до поведінкового стилю опису. Основна його особливість в тому, що він дозволяє неявно задати бажані для генерації елементи, а програмні засоби, що обробляють вихідний код, визначають, як саме потрібно реалізовувати дані інструкції з використанням наявних у ПЛІС-ресурсів. Поведінковий стиль опису надає можливість розробнику працювати на більш високому рівні, ніж структурний та стиль потоку даних. Значна частина роботи з побудови схеми перенесена на програмні засоби, а, відповідно, завдання розробника спрощується. Саме тому поведінковий стиль опису знаходить найбільше застосування за реалізації великих проєктів.

3.4 Виконання інструкцій у процесі

Попередній приклад містив лише один умовний оператор, всередині якого розміщувався один оператор присвоювання сигналу. Оператори у VHDL можна розділити на:

1) паралельні;
2) послідовні.

Всередині тіла процесу можуть розміщуватись лише послідовні оператори (це **if…then**, **case…when**, **for…loop**, присвоєння значення сигналу або змінній). Оператори, які можуть розміщуватись поза тілом процесу, є паралельними. При цьому, сам опис процесу є паралельним оператором. Також варто зазначити, що оператор присвоювання значення сигналу може розташовуватись всередині процесу або поза ним. Залежно від цього розрізняють послідовний та паралельний оператор присвоєння значення сигналу.

Для програмістів! *Тіло блоку **process** виконується постійно, відповідно до його списку чутливості. Це ідентично тому, що забезпечує у мовах програмування нескінченний цикл **while(true)**.*

Тіло процесу виконується постійно відповідно до заданого списку чутливості процесу. Тобто, як тільки відбувається зміна одного із вказаних сигналів, виконуються всі інструкції всередині процесу. Саме так слід розуміти опис частини схеми за допомогою процесу.

Відносно роботи з сигналами всередині процесів, варто пам'ятати правила відносно зчитування та запису значень сигналів:

- будь-яка кількість процесів може зчитувати значення сигналу;
- лише один процес може встановлювати значення сигналу.

Це означає, що значення сигналу може присвоюватись лише одним процесом, причому кількість операцій присвоєння даному сигналу всередині процесу не обмежується.

3.5 Змінні процесів

VHDL надає можливість зберігати локальні значення у **змінних**. Змінна описується в деклараційній частині процесу за допомогою ключового слова **variable**. Як тип змінної можна вказати будь-який тип VHDL.

Головна особливість змінних і, водночас, відмінність між змінними та сигналами полягає в тому, що присвоєння значення змінній в процесі відбувається одразу. В той час, як було наведено вище, при присвоєнні значень сигналам, що представляють собою регістри, значення встановлюється з затримкою в один період тактової частоти.

Для програмістів! *Незважаючи на те, що змінні на перший погляд можуть здатися зручним інструментом для опису роботи процесу, загальноприйнятою практикою у проєктах, метою яких є створення продукту, а не лише симуляція, є мінімізація використання змінних. Це пов'язано з тим, що для створення регістрів – основних складових синхронних схем – призначені саме сигнали, тому краще використовувати їх. Програмні засоби, що обробляють опис VHDL для створення бітового файлу конфігурації, краще працюють саме із сигналами – синтез (один з етапів обробки) коду, що використовує змінні, може у результаті створити інший компонент логіки. Тим не менш, змінні корисні при тестуванні створених описів.*

3.6 Паралельні оператори

Паралельні оператори використовуються для того, щоб описати частину схеми, яка функціонує постійно. Опис процесу, як вже зазначалось, є паралельним оператором. Більшість із розглянутих вище прикладів використовували лише один оператор паралельного присвоєння значення сигналу.

Серед найбільш важливих паралельних операторів слід відзначити оператори умовного присвоювання. Такі оператори використовуються для реалізації мультиплексорів.

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

entity mux is
  port(
    sel: in std_logic_vector(1 downto 0);
    in1, in2, in3, in4: in std_logic_vector(3 downto 0);
    out_res: out std_logic_vector(3 downto 0)
  );
end entity;

architecture basic of mux is
begin
  out_res <= in1 when sel = "00" else
             in2 when sel = "01" else
             in3 when sel = "10" else
             in4 when sel = "11" else
             "0000";
end architecture;
```

Даний оператор дозволяє обрати значення для сигналу на виході на основі сигналу-селектору. У тому випадку, якщо жоден зі вказаних варіантів не задовольняє умов, то обирається значення за замовчуванням, яке вказано в кінці. Симуляція роботи для даного мультиплексора наведена на рис. 25.

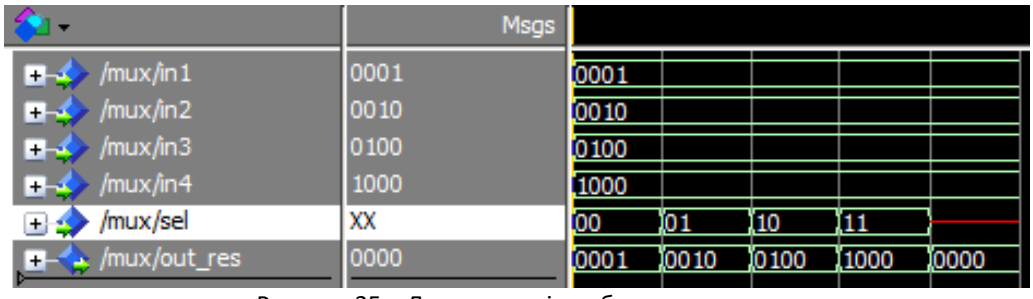


Рисунок 25 – Демонстрація роботи оператору

Варто звернути увагу на те, що при подачі сигналу-селектору, який відсутній серед очікуваних значень для перевірки, на виході отримуємо значення за замовчуванням.

Аналогічних результатів (реалізація мультиплексора) можна досягти з використанням оператору **with…select**. Він також є умовним оператором, на основі якого створюється комбінаційна логіка. Розглянемо, як зміниться вищенаведений опис при використанні **with…select**.

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

entity mux is
  port(
    sel: in std_logic_vector(1 downto 0);
    in1, in2, in3, in4: in std_logic_vector(3 downto 0);
    out_res: out std_logic_vector(3 downto 0)
  );
end entity;

architecture basic of mux is
begin
  with sel select out_res <=
                      in1 when "00",
                      in2 when "01",
                      in3 when "10",
                      in4 when "11",
                      "0000" when others;
end architecture;
```

Основною відмінністю даного оператора від попереднього є більша лаконічність у присвоювальній частині. Надання переваги одному з цих операторів залежить від правил, прийнятих для проєкту.

3.7 Послідовні оператори

Як вже зазначалось, в тілі процесу допускається знаходження лише деяких операторів. Одним з таких операторів є **case…when**. У деякій мірі його можна назвати відповідником паралельних умовних операторів присвоєння.

Розглянемо його використання на прикладі компонента, який інвертує подане на вхід 2-бітне значення (звичайно, запропонована реалізація не є оптимальною для вирішення даної задачі та наводиться лише в навчальних цілях).

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

entity case_unit is
  port(
    clk: in std_logic;
    sel: in std_logic_vector(1 downto 0);
    out_val: out std_logic_vector(1 downto 0)
  );
end entity;

architecture basic of case_unit is
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      case sel is
        when "00" =>
          out_val <= "11";
        when "01" =>
          out_val <= "10";
        when "10" =>
          out_val <= "01";
        when "11" =>
          out_val <= "00";
        when others =>
          out_val <= "00";
      end case;
    end if;
  end process;
end architecture;
```

Оператор **case** за перевірки можливих значень сигналу повинен вказати дію для кожного такого значення. При цьому кількість можливих значень може бути дуже великою. Саме для того, щоб зробити запис обробки різних умов більш лаконічним, додана можливість використовувати конструкцію **when others**, яка виконується у випадку, коли попередні можливі значення не збіглися. Для наведеного прикладу велика кількість можливих значень сигналу **sel** не вказана (наприклад, варіанти із ‘Z’, ‘X’ та ін.). Вказано лише чотири можливі значення, а інші будуть відповідати умові **when others**. Більш наочно ознайомитись з роботою можна на основі симуляції.

Проведена симуляція роботи компонента наведена на рис. 26.

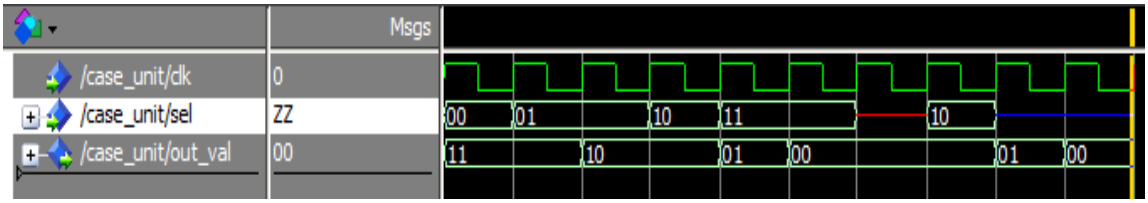


Рисунок 26 – Симуляція роботи оператору case

Оскільки для опису компонента використовувався процес, який працює з використанням тактового сигналу, то це означає створення синхронної логіки, а, отже, затримку на виході, яка становить 1 період тактового сигналу. Із симуляції видно, що компонент може обробляти як вказані значення вхідних сигналів, так і опущені (‘**XX**’ та ‘**ZZ**’) за допомогою секції **when others**.

3.8 Цикл for…loop

Останньою інструкцією, що може розміщуватись в процесах, є цикл **for…loop**. Її відмінність від аналогів у мовах програмування, мабуть, є найбільшою серед усіх інших. Як вже зазначалось, сам процес слід розуміти як нескінченний цикл, оскільки він працює постійно та представляє собою складову частину схеми.

Для програмістів! *Розташування циклу **for…loop** у тілі процесу означає, що за кожного входу в тіло процесу та досягнення виконання циклу відповідно до послідовності інструкцій у процесі, увесь цикл виконається одразу. Тобто, різниці в часі між умовно першою ітерацією та останньою немає – усі вони виконуються одночасно. Цикл **for…loop** дозволяє описати логіку роботи схеми, що має регулярну структуру всередині процесу. Часто такі цикли використовують у функціях та процесах VHDL.*

Розглянемо використання циклу на простому прикладі. Припустимо, що компонент приймає на вхід значення 8 сигналів (8-бітна шина), і необхідно реалізувати на виході операцію виключного АБО для кожної пари бітів. Це означає, що вихід є чотирибітним, причому, перший біт є результатом застосування виключного або до 1-го та 2-го бітів на вході, другий – 3-го та 4-го і т. д.

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

entity for_example is
  port(
    clk: in std_logic;
    in_bus: in std_logic_vector(7 downto 0);
    res: out std_logic_vector(3 downto 0)
  );
end entity;

architecture basic of for_example is
begin
  process(clk)
  begin
    if clk = '1' and clk'event then
      for i in 0 to 3 loop
        res(i) <= in_bus(i * 2) xor in_bus(i * 2 + 1);
      end loop;
    end if;
  end process;
end architecture;
```

Особливістю циклу є те, що змінна циклу (***i***) не потребує попередньої декларації. Вказується діапазон значень, на основі якого працює цикл.

Проведемо симуляцію роботи даного компонента (рис. 27).

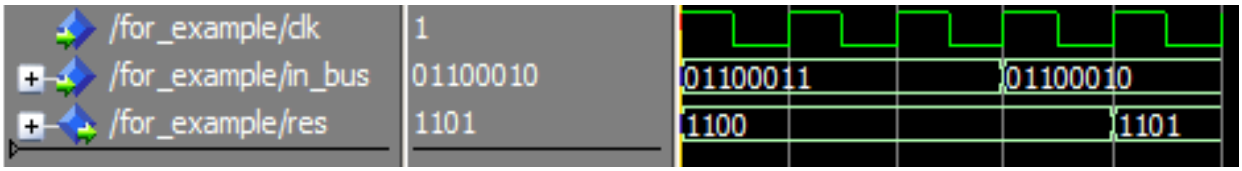


Рисунок 27 – Поведінка сигналів при використанні оператору **for**

Як видно із симуляції, сигнал на виході з'являється на один такт пізніше після зміни сигналу входу, що відповідає правилу про те, що присвоєння значення сигналу всередині перевірки на наявність фронту, призводить до створення регістра.

Контрольні питання

1. Чи може сигнал типу string бути вихідним портом?
2. Використання коментарів при описі порту є обов'язковою вимогою стандартів мови VHDL?
3. Початкове значення порту може бути задане?
4. Початкове значення змінної може бути задане?
5. Відповідно до побажання проєктувальника оператор призначення сигналу a <= b; може бути записаний у зворотному напрямку, тобто у вигляді b => a;?
6. У виразах (мови VHDL) можуть використовуватися константи?
7. Процедура може бути декларована у пакеті?
8. Чи може сигнал типу A integer передати своє значення сигналу типу B real за допомогою оператора B <= A after 5 ns?
9. Чи може значення true присвоєно сигналу типу bit?
10. Чи може значення '0' бути призначене сигналу типу boolean?
11. Чи може значення '0' призначатися сигналу типу integer?
12. Чи може значення 0 призначатися сигналу типу real?
13. Чи може значення 0 призначатися сигналу типу integer?
14. Чи всі елементи масиву повинні мати однаковий тип?
15. У VHDL можна множити дійсне число ціле?
16. У VHDL можна множити дійсне число тимчасово (тип time)?
17. Через 0.0 позначається значення нуля для типу real?
18. Чи можна декларувати двовимірний масив, елементами якого є натуральні числа, не більші за 1000?
19. Якщо тривалість сигналу більша за його затримку, то інерційна модель затримки дасть той самий результат, що й транспортна модель затримки?
20. Атрибут clk'event сигнал clk має тип boolean?
21. Атрибут S'left масиву S цілих чисел задає ліве значення номера числа (індексу) у масиві?
22. Чи може користувач визначити власний атрибут?
23. У VHDL є визначені атрибути?
24. Чи мають сигнали атрибути?
25. Константи мають атрибути?
26. Змінні мають атрибути?

4.ГЕНЕРАЦІЯ РЕГУЛЯРНИХ ЧАСТИН СХЕМИ. РЕАЛІЗАЦІЯ КОМПОНЕНТІВ

4.1 Оператор generate

У попередньому підрозділі було проведено ознайомлення з організацією циклів у VHDL та були описані їх особливості. Також вже було представлено, як використовувати вже готовий модуль всередині іншого, утворюючи ієрархію модулів. У деяких випадках один і той самий модуль необхідно включити всередину іншого не один чи два рази, а десятки, сотні разів. У такому випадку додавання модулів вручну займе, по-перше, багато часу, а, по-друге, багато місця у файлі опису, що не є найраціональнішим рішенням. Зазвичай, формування масиву з компонентів буде мати регулярну структуру, яка має закономірності щодо повторення. У таких випадках більш доцільним варіантом є використання оператору **generate** у комбінації з циклом loop.

Як простий приклад використання оператору **generate** розглянемо 8-бітний регістр, який буде побудований на основі раніше розглянутих регістрів. Звісно, більш простим способом було б використання сигналу типу **std_logic_vector** із виконанням необхідних присвоєнь сигналам, проте, результуюча структура в обох випадках буде однаковою. Також для ускладнення прикладу додамо умову, що результуючий вектор має бути дзеркально відображений відносно початкового біту. Лістинг коду з використанням інструкції **generate** представлено далі.

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

entity generate_example is
  port(
    clk: in std_logic;
    rst_n: in std_logic;
    ind: in std_logic_vector(7 downto 0);
    outd: out std_logic_vector(7 downto 0)
  );
end entity;

architecture behav_gen of generate_example is
begin
  registers: for i in 0 to 7 generate
    regx_inst: entity work.reg port map (
      clk => clk,
      rst_n => rst_n,
      d => ind(i),
      q => outd(7-i)
    );
  end generate;
end architecture;
```

Як видно з прикладу, безпосередньо перед циклом **for** встановлюється мітка (**registers**). Ім'я мітки може бути довільним, проте її присутність в описі є обов'язковою. У рядку закінчення оператору end generate можна також вказати ім'я мітки, проте у даному випадку це не є обов'язковим. Такий синтаксис необхідний для організації вкладених інструкцій generate для того, щоб розрізняти, з якого оператору має виконуватись вихід. Варто зазначити, що межі діапазону, який використовується для ітерації, мають бути відомі під час синтезу схеми – даний оператор не дозволяє динамічної зміни структури модулю.

Розглянемо подачу тестових векторів на розроблений модуль у тестовому оточенні.

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

entity generate_tb is
end entity;

architecture test of generate_tb is
  signal clock: std_logic := '0';
  signal rst: std_logic := '0';
  signal in_data: std_logic_vector(7 downto 0) := (others => '0');
  signal out_data: std_logic_vector(7 downto 0) := (others => '0');
begin
  clock <= not clock after 50 ns;

  registers: entity work.generate_example port map (
    clk => clock,
    rst_n => rst,
    ind => in_data,
    outd => out_data
  );

  stim_proc: process
  begin
    wait for 50 ns;
    rst <= '1';
    in_data <= "11110000";
    wait for 50 ns * 2;
    in_data <= "10010110";
    wait for 50 ns * 2;
    in_data <= "00001111";
    wait for 50 ns * 2;
    wait;
  end process;
end architecture;
```

Для виконання тесту подаються три різні варіанти вхідного слова. Моделювання модуля показує, що на наступному фронті тактового сигналу на виході модуля з'являється дзеркально відображений вхідний вектор (рис. 28).

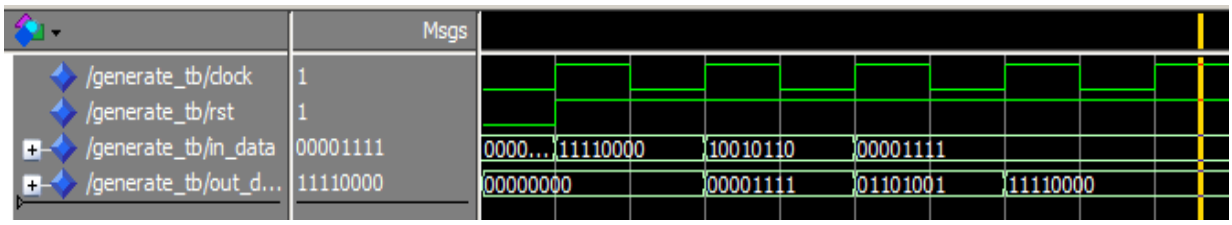


Рисунок 28 – Подача тестових сигналів за допомогою тестбенчу

Іншим варіантом використання оператору generate у мові VHDL є його використання в комбінації з оператором **if**. Таке сполучення використовується для того, щоб включити у опис частини/модулі, які необхідні лише за виконання певних умов. Умова, що вказується у операторі **if,** має бути обчислювальною на етапі побудови кінцевого модуля, як і для випадку зі змінною ітерації для **for…generate**. Таким чином, основне призначення даного оператору – реалізація опису, який буде змінюватись залежно від виконання певних умов. Ці умови можуть задаватись у модулі верхнього рівня.

Розглянемо також приклад такого оператору.

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

entity reg is
  port(
    clk: in std_logic;
    rst_n: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end entity;

architecture behav of reg is
  signal ds: std_logic := '0';
begin
  process(all)
  begin
    if rst_n = '0' then
      ds <= '0';
    elsif clk = '1' and clk'event then
      ds <= d;
    end if;
  end process;

  q <= ds;
end architecture;
```

4.2 Скидання схеми (RESET). Асинхронне скидання

Забезпечення повернення компонента до певного початкового стану прийнято реалізовувати за допомогою сигналу скидання – **RESET**. Це передбачає, що в описі інтерфейсу з'явиться ще один вхідний сигнал:

```
entity flip_flop is
  port(
    clk: in std_logic;
    rst: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end entity;
```

У свою чергу в реалізації компонента також необхідно внести зміни:

```
architecture basic of flip_flop is
begin
  process(all)
  begin
    if rst = '1' then
      q <= '0';
    elsif clk = '1' and clk'event then
      q <= d;
    end if;
  end process;
end architecture;
```

У випадку використання синтаксису VHDL більш ранньої, ніж 2008, необхідно додати **rst** до списку чутливості процесу.

Такий варіант реалізації передбачає, що незалежно від поданих вхідних даних на виході буде встановлено значення ‘0’ у тому випадку, якщо сигнал скидання **rst** є активним. Варто звернути увагу на те, що даний сигнал часто роблять інверсним, тобто, таким, для якого активним є значення ‘0’. Тому при використанні сторонніх компонентів варто ознайомитись із документацією для них, щоб правильно встановити сигнал скидання.

Розглянемо роботу схеми регістра за наявності сигналу RESET (рис. 29).

Рисунок 29 – Поведінка схеми при використанні сигналу RESET

По-перше, слід звернути увагу на появу сигналу червоного кольору на виході регістра. Це відбувається тому, що за замовчуванням усі сигнали не є ініціалізованими, в тому числі і сигнал **rst**, тому жодна інструкція всередині **if** не виконується – перевірки на ‘U’ для **rst** не проводиться. По-друге, зміна вихідного сигналу в ‘0’ при активному значенні сигналу **rst.**

Можливі два варіанти організації скидання:

1) синхронний;
2) асинхронний.

У прикладі наведено асинхронний варіант. Саме такий варіант рекомендується для використання. Це пов’язано з тим, що синхронне скидання потребує більшого використання ресурсів. Тому загальноприйнятою практикою є реалізація сигналу **RESET** у вигляді асинхронного сигналу. Реалізація синхронного скидання відрізнятиметься тим, що перевірка значення сигналу **rst** відбуватиметься всередині **if clk = '1' and clk'event then….**

Для програмістів! *Програмування у середовищі Arduino передбачає реалізацію двох функцій – **init** та **loop**. Багато в чому це аналогічно роботі процесів із використанням сигналу скидання (асинхронний варіант). Секція, що виконується при активному сигналі **rst**, відповідає за ініціалізацію (**init**), а основні функції реалізуються при перевірці фронту тактового сигналу.*

Таким чином, основне призначення скидання – ініціалізація сигналів компонента та всієї схеми певними початковими значеннями.

4.3 Реалізація лічильника на VHDL

Для подальшого дослідження можливостей VHDL розглянемо реалізацію лічильника. Лічильник приймає на вхід тактові імпульси, підраховує їх кількість та за умови досягнення певного значення починає лічбу спочатку. На виході компонента-лічильника можна отримати поточне його значення. Крім того, при досягненні кінцевого значення (для прикладу обрано значення 200) для лічби видається імпульс на виході.

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

entity counter is
  port(
    clk: in std_logic;
    rst: in std_logic;
    count_val: out std_logic_vector(7 downto 0);
    out_pulse: out std_logic
  );
end entity;

architecture basic of counter is
  signal count_reg: unsigned(7 downto 0) := (others => '0');
begin
  process(all)
  begin
    if rst = '1' then
      count_reg <= (others => '0');
      out_pulse <= '0';
    elsif clk'event and clk = '1' then
      count_reg <= count_reg + 1;
      out_pulse <= '0';
      if count_reg = "11001000" then
        count_reg <= (others => '0');
        out_pulse <= '1';
      end if;
    end if;
  end process;

  count_val <= std_logic_vector(count_reg);
end architecture;
```

Перша відмінність від попередніх прикладів – використання ще одного пакету з бібліотеки **ieee**. Пакет numeric_std містить визначення типу unsigned. Даний тип використовується для представлення чисел без знаку. Фактично, він представляє собою масив значень **std_logic**, як і **std_logic_vector**. Проте, на відміну від std_logic_vector, для unsigned у даному пакеті визначена велика кількість додаткових операцій, які спрощують роботи із сигналами даного типу. В їх числі і операції додавання (+) з числовим значенням, яка використовується в прикладі.

Ініціалізація сигналу лічильника початковим значенням здійснюється за допомогою оператору присвоювання (:=). Також варто звернути увагу на спосіб задання початкового значення. Для цього використана наступна інструкція:

(others => '0').

Вона означає, що всі біти, що належать сигналу, будуть встановлені в «0». Дану інструкцію зручно використовувати для ініціалізації, оскільки вона буде коректно працювати незалежно від розміру масиву. Наприклад, якщо виникне необхідність вести лічбу лише до 100 (для чого достатньо 7 бітів), то зміні підлягатиме лише розрядність лічильника.

Для порівняння значення лічильника з кінцевим значенням використовується літерал “11001000” (двійкове представлення числа 200). На відміну від значень для сигналу типу **std_logic** використовуються подвійні лапки (“ ”). Так само можна працювати з сигналами типу **std_logic_vector**, оскільки базовий їх тип є спільним.

Іншим важливим моментом у даному прикладі є організація з’єднання між сигналом-регістром всередині архітектури та виходом компонента. Присвоєння значення сигналу відбувається всередині процесу. Для того, щоб організувати з’єднання використовується інструкція

count_val <= std_logic_vector(count_reg).

Оскільки типи сигналу та виходу є різними, то використовується операція приведення типу.

При проведенні симуляції роботи лічильника для полегшення відслідковування значень бажано змінити представлення для регістра та виходу лічильника (рис. 30).

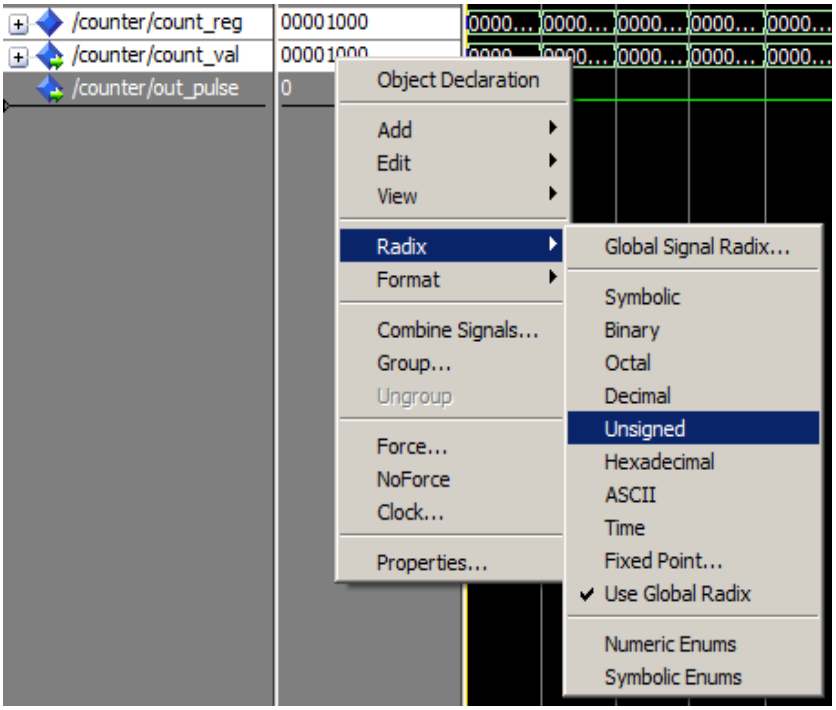


Рисунок 30 – Вибір числового представлення сигналів

Результат симуляції повинен відповідати зображеному на рис. 31.

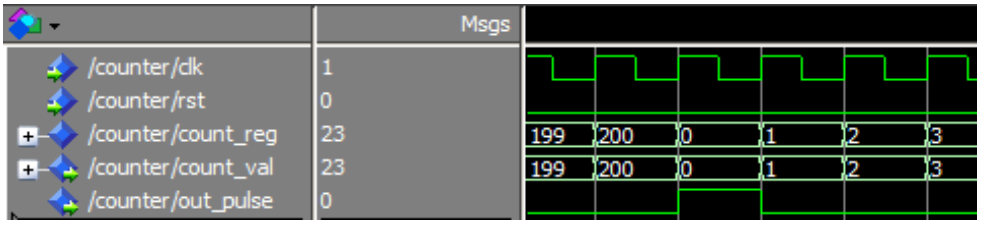


Рисунок 31 – Результат зміни представлення

4.4 Операції з числами у VHDL

У попередній частині було частково наведені основи роботи з беззнаковими цілими числами з пакету **numeric_std**. Розглянемо детальніше, які саме можливості щодо роботи з числами надає VHDL.

Цілі числа представлені двома типами:

1) signed (використовуються для представлення як додатних, так і від'ємних чисел);
2) unsigned (використовуються для представлення лише невід'ємних чисел).

Зручність використання саме числових типів для опису мікросхеми полягає у тому, що в пакетах реалізована також велика кількість функцій для роботи з ними (наприклад, операції +, –), які підтримуються програмними засобами обробки коду VHDL. Саме ці типи рекомендується використовувати для опису робочих мікросхем.

При цьому, слід зауважити що типи з такими назвами доступні одразу у двох пакетах: **numeric_std** та **std_logic_arith**. Через те, що другий пакет не є стандартизованим, рекомендується використовувати перший. Обидва пакети надають схожий набір функцій для роботи з цілими числами.

Для програмістів! *Роль цілих чисел при реалізації мікросхем на ПЛІС за сучасного рівня розвитку техніки є значною. Для підвищення швидкості роботи описаних схем рекомендується використовувати саме цілочисельні алгоритми та математичні операції замість алгоритмів, що потребують роботи з дробовими числами (з плаваючою або фіксованою десятковою комою). Це пов'язано з тим, що апаратна реалізація роботи з цілими числами є набагато простішою та потребує меншої кількості логічних ресурсів. Саме тому при розробці пристроїв на ПЛІС слід намагатись використовувати саме роботу з цілими числами.*

Окрім даних типів у VHDL також доступний тип **integer**, який є вбудованим типом і не входить до складу стандартних пакетів. Приклад використання даного типу для декларації сигналу є наступним:

```
signal int: integer range -8 to 7;
```

Як бачимо, підтримуються як додатні, так і від'ємні значення. Діапазон значень можна опустити, проте, якщо компонент планується використовувати для створення схеми, то встановлення діапазону значень є обов'язковим. Це пояснюється тим, що від вказаного за замовчуванням обирається увесь діапазон значень для integer, що призведе до значного використання ресурсів.

4.5 Записи у VHDL

Для того, щоб узагальнити опис певних типів, які мають об'єднувати декілька складових елементів, у VHDL доступний тип записів (**record**). За своїм використанням вони нагадують записи у мові Pascal або структури у мові C. Структури складаються з полів, які можуть мати різні типи. Основною перевагою їх використання є зменшення обсягу коду, який необхідний для опису портів, сигналів.

Розглянемо найпростіший приклад використання записів у VHDL. Для цього створимо інтерфейс без портів та відповідну йому архітектуру. У декларативній частині архітектури опишемо новий тип-запис, який буде складатись із двох елементів типу std_logic. Створимо сигнал описаного типу, та присвоїмо початкові значення сигналу.

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

entity record_work is

end entity ; -- record_work

architecture arch of record_work is
    type rec is record
        empty: std_logic;
        full: std_logic;
    end record;

    signal r: rec := (empty => '1', full => '0');
begin

    process
    begin
        wait for 10 ns;
        r.empty <= not r.empty;
        r.full <= not r.full;
    end process;

end architecture ; -- arch
```

Процес, що присвоює значення сигналу (а точніше його компонентам), виконує просту операцію заперечення для зміни значення сигналу через задані проміжки часу. Результат роботи такого опису представлено на рис. 32.

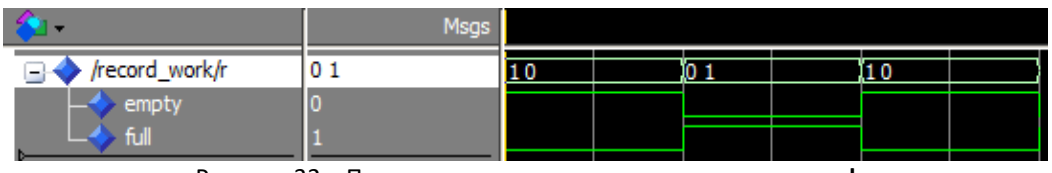


Рисунок 32 – Представлення простого сигналу типу **record**

Як видно з рисунку, представлення записів у вікні Wave відрізняється від представлення простих сигналів: через проміжок вказуються значення усіх полів. Крім того, є можливість відслідковувати зміни окремого компонента, розгорнувши представлення запису.

4.6 Організація пакетів у VHDL

Як вже згадувалось у попередніх підрозділах, важливою одиницею для організації коду є пакет. Загалом, пакет можна розглядати як аналог пакету Java. Проте, є деякі виключення, які відрізняють їх.

> **Для програмістів!** *Функціональна роль, яка призначена пакетам у VHDL, не є настільки великою, як у Java. У пакетах можна описувати обмежений набір компонентів – у них не можна розміщувати entity або architecture, відповідно, більшість коду має знаходитись поза межами пакету. Тим не менш, це не заперечує їх цінність з точки зору повторного використання коду.*

Основними структурними одиницями мови VHDL, які можна розміщувати у пакетах є користувацькі типи даних, функції, процедури. Розглянемо, як виконати перенос коду типу з попереднього прикладу у пакет та як підключити його для використання у необхідному модулі. У прикладі нижче наведено те, як виконується опис пакету, у якому розміщено згадуваний тип-запис.

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

package pack_record is
    type rec is record
        empty: std_logic;
        full: std_logic;
    end record;
end package ; -- pack_record

package body pack_record is

end package body;
```

Опис пакету виконується у окремому файлі (хоча, це не є обов'язковим). Розрізняють декларативну частину пакету та частину реалізації (тіло) пакету. У декларативній частині розміщуються описи типів, функцій та процедур. У частині реалізації мають бути реалізовані функції та процедури, які описані у декларативній частині. Як видно з прикладу, секція реалізації може бути порожньою.

У початковому файлі будуть проведені зміни, які показані нижче.

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

use work.pack_record.all;

entity record_work is
end entity ; -- record_work

architecture arch of record_work is
    signal r: rec := (empty => '1', full => '0');
begin
```

Тепер у файлі виконується підключення локального пакету, а з секції декларації прибрано опис типу-запису. Запуск моделювання модуля повинно показувати той самий результат, як і для початкового прикладу модуля.

4.7 Поняття контексту

Починаючи з версії VHDL-2008, у мову було введене поняття контексту. Контекст спрощує включення кількох згрупованих імен пакетів для повторного використання. Таким чином, підключення групи пакетів може відбуватись лише одним рядком коду. Розглянемо принцип використання контексту на прикладі.

По-перше, слід зауважити, що бібліотека **ieee** вже містить у своєму складі декларації двох контекстів. Розглянемо, як виглядає опис контексту.

```
context IEEE_STD_CONTEXT is
  library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.NUMERIC_STD.all;
end context IEEE_STD_CONTEXT;
```

Як бачимо, для оголошення контексту використовується виділене ключове слово context. У оголошенні контексту включені імена бібліотек та пакетів, що мають бути підключені.

Підключення контексту буде організоване таким чином.

```
library ieee;
context ieee.ieee_std_context;
```

Відповідно, як зрозуміло з прикладу, у більшості з наведених раніше і далі прикладів можна провести заміну стандартного підключення бібліотек на підключення контексту.

4.8 Функції та процедури

У попередньому підрозділі вказувалось, що, у більшості випадків, пакети використовуються для того, щоб описати типи функції та процедури, які мають використовуватись у різних одиницях компіляції, а в подальшому можуть бути перенесені у нові проєкти. І, якщо основні типи, в т. ч. користувацькі, були розглянуті до цього, то функції та процедури не отримали детального опису.

Роль процедур та функцій у VHDL є важливою, однак особливість полягає у тому, що неможливо перенести усю логіку роботи цифрової схеми у дані структурні одиниці. Процедури та функції мають виконувати роботу/повертати результат по обчисленням у той самий момент, коли відбувається їх виклик. Під виконання функцій або процедур не відводиться модельний час, тому до того моменту, коли не буде обчислено значення виклику, моделювання не може продовжуватись. Таким чином, функції та процедури інтерпретуються з точки зору середовища. У них не можна очікувати на зміну стану сигналу, як це робиться в процесах. Проте, з іншого боку, це означає, що для виконання заданих дій у мові VHDL виділена лише одна структурна одиниця, яка має використовуватись постійно. Так само і функціям та процедурам виділено чітке місце для застосування.

Загалом, процедури та функції відрізняються між собою тим, що останні повертають значення. У випадку, якщо необхідно повернути декілька значень, то у вхідних параметрах необхідно вказати, що параметр є вихідним за допомогою модифікатора out. Такий підхід працює як для функцій, так і для процедур. Таким чином, різниця між функціями та процедурами є доволі формальною і полягає у наявності оператора return у тілі функції. Розглянемо приклад використання функції. Додамо нову функцію у вже існуючий пакет для того, щоб продемонструвати, як виглядає повний опис функції.

```
    function rotate3(foo: std_logic_vector(3 downto 0)) return std_logic_vector;
end package ; -- pack_record

package body pack_record is
    function rotate3(foo: std_logic_vector(3 downto 0)) return std_logic_vector is
        variable res: std_logic_vector(3 downto 0);
    begin
        for i in 0 to 3 loop
            res(i) := foo(3-i);
        end loop;
        return res;
    end function;
end package body;
```

У деклараційній частині функцію необхідно спочатку описати, після чого у тілі пакету можна реалізувати функцію. Обов'язковим є вказання типу значення, яке повертає функція. Після типу значення, яке повертається функцією, вказується ключове слово **is**. У функцій та процедур наявна власна секція декларації, яка дозволяє оголосити змінні та сигнали. Варто також звернути увагу на наявність модифікатору **in** перед типом параметру, який вказує на те, що даний параметр є вхідним. Як вже зазначалось, повертати значення з функції або проце-

дури можна шляхом використання модифікатору out. Результат виконання функції обов'язково має бути присвоєний сигналу або змінній. Приклад використання описаної вище функції може бути таким:

```
process
begin
    result <= rotate3(test);
    wait;
end process;
```

Результат моделювання даного процесу представлено на рис. 33.

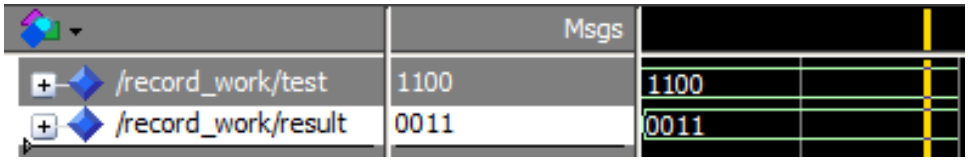


Рисунок 33 – Результат симуляції опису

Контрольні питання

1. Підтип є підмножиною значень типу?
2. Константі може бути призначене нове значення, якщо нове значення дорівнює попередньому?
3. У мові VHDL значення «істина» типу boolean дорівнює значенню '1' типу bit?
4. Під час визначення декларації константи достатньо описати лише її значення, оскільки тип константи визначається значенням?
5. Логічні оператори можуть виконуватися лише над операндами типу bit?
6. Логічні оператори можуть виконуватися лише над операндами типу boolean?
7. Логічні оператори можуть виконуватись лише над операндами типу std_logic?
8. Оператор очікування wait на початку процесу відіграє ту саму роль, ніби він був розташований у процесі перед словом end?
9. Умовний оператор if має бути завершено одним ключовим словом endif?
10. Булева умова в циклі типу while перевіряється на початку кожної ітерації?
11. Виконання процесу, який має списку сигналів запуску, закінчується, коли досягається ключове слово end?
12. Процес зі списком чутливості повинен не містити жодного оператора очікування?
13. Сигнали та змінні одного й того ж типу можуть бути присвоєні один одному?
14. У entity обов'язково мають бути описані порти (port)?
15. Параметр (generic) має бути константою?
16. Чи правильно те, що робоча бібліотека проєкту за замовчуванням має ім'я WORK?
17. Чи можна перед декларацією entity посилатися на кілька пакетів?
18. Чи можна використовувати іншу конфігурацію в тексті конфігурації?
19. Чи можна перед описом пакета посилатися на інший пакет?
20. Чи може Константа передати своє значення змінній того ж типу?
21. Вираз у заголовку оператора case може бути дискретного типу?
22. Символом & позначається операція конкатенації?
23. Чи містить процес послідовні оператори?
24. Чи є специфікація послідовних операторів?
25. Модельний час VHDL описується типом integer?
26. Модельний час у VHDL описується типом real?

5.ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ КОМПОНЕНТІВ ТА АВТОМАТИЗАЦІЇ ВИКОНАННЯ ЗАВДАНЬ

5.1 Параметризація опису

Важливим та потужним засобом VHDL для повторного використання опису компонентів є параметризація коду. Параметризація широко використовується при застосуванні готових компонентів у середовищах розробки, тому необхідно розуміти її базові принципи. Розглянемо параметризацію на прикладі компонента-лічильника.

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

entity counter is
  generic(
    N: integer := 8;
    constant end_val: unsigned := "11001000"
  );
  port(
    clk: in std_logic;
    rst: in std_logic;
    count_val: out std_logic_vector(N-1 downto 0);
    out_pulse: out std_logic
  );
end entity;

architecture basic of counter is
  signal count_reg: unsigned(N-1 downto 0) := (others => '0');
begin
  process(all)
  begin
    if rst = '1' then
      count_reg <= (others => '0');
      out_pulse <= '0';
    elsif clk'event and clk = '1' then
      count_reg <= count_reg + 1;
      out_pulse <= '0';
      if count_reg = end_val then
        count_reg <= (others => '0');
        out_pulse <= '1';
      end if;
    end if;
  end process;

  count_val <= std_logic_vector(count_reg);
end architecture;
```

Основна відмінність від початкового опису – наявність секції **generic** в описі інтерфейсу компонента. У ній описані параметризовані значення. В описі їм присвоюються значення за замовчуванням.

Розглянемо опис, який створює два лічильники з різними параметрами. Реалізація опису інтерфейсу для даного модуля є завданням для перевірки читача.

```
architecture counters of two_counters is
  component counter is
  generic(
    N: integer := 8;
    constant end_val: unsigned := "11001000"
  );
  port(
    clk: in std_logic;
    rst: in std_logic;
    count_val: out std_logic_vector(N-1 downto 0);
    out_pulse: out std_logic
  );
  end component;
begin
  counter1: counter port map (
    clk => clk, rst => rst, count_val => count1, out_pulse => count_pulse1
  );
  counter2: counter generic map (
    N => 7, end_val => "1100100"
  )
  port map (
    clk => clk, rst => rst, count_val => count2, out_pulse => count_pulse2
  );
end architecture;
```

Наведений опис організовано в структурному стилі. У ньому створюються два лічильники. Перший лічильник не змінює параметри за замовчуванням, в той час як другий – змінює ці параметри. Встановлюється зменшена ширина та інше кінцеве значення (для лічби до 100 достатньо 7 розрядів у регістрі). Налаштування параметрів відбувається з використанням інструкції **generic map**. Порти модуля верхнього рівня (англ. top-level unit) підключені безпосередньо до портів лічильників.

Симуляція роботи схеми за зменшеного масштабу сигналів має виглядати, як наведено на рис. 34.

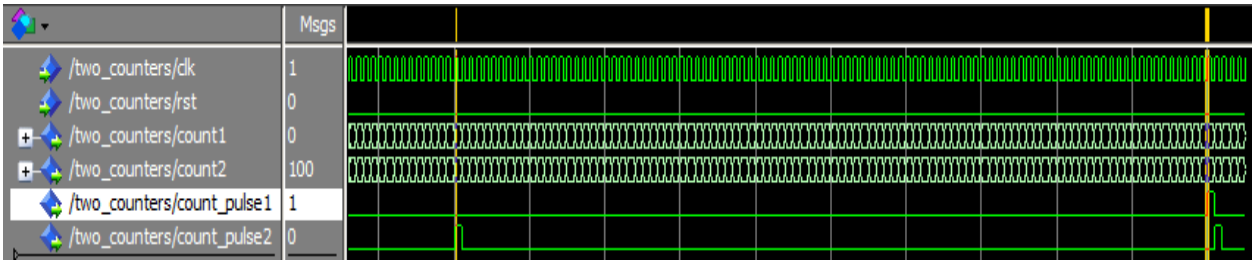


Рисунок 34 – Симуляція роботи схеми лічильника

5.2 Конфігураційні блоки у VHDL

Як зазначалось раніше, конфігурації є своєрідними аналогами контейнерів Dependency Injection у об'єктно-орієнтованих мовах програмування. Вони дозволяють обрати, який самий модуль необхідно включити у склад іншого модуля (яку саме реалізацію використовувати під час побудови модулю). Вони стають у нагоді, коли необхідно провести тестування для однотипних модулів із різними налаштуваннями, змінюючи інтерфейси взаємодії та у інших ситуаціях, де можна узагальнити інтерфейс та використовувати окремі блоки у конкретних випадках (наприклад, тестове оточення для простого і складного блоків, яке значно відрізняється по споживанню пам'яті під час симуляції).

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

entity or_unit is
  port(
    a: in std_logic;
    b: in std_logic;
    res: out std_logic
  );
end entity;

architecture struct of or_unit is
  component or_component is
  port(
    a: in std_logic;
    b: in std_logic;
    res: out std_logic
  );
  end component;
begin
  or_inst: or_component port map (
    a => a,
    b => b,
    res => res
  );
end architecture;

library ieee;
use ieee.std_logic_1164.all;

entity or_comp_entity is
  port(
    a: in std_logic;
    b: in std_logic;
    res: out std_logic
  );
end entity;

architecture or_arch of or_comp_entity is
begin
  res <= a or b;
end architecture;

use work.all;

configuration or_config of or_unit is
  for struct
    for or_inst: or_component
      use entity or_comp_entity(or_arch);
    end for;
  end for;
end configuration;
```

5.3 Реалізація пам’яті у VHDL

Блокова пам’ять (англ. block-RAM) у складі ПЛІС є важливим ресурсом. Вона призначена для збереження значних об’ємів даних. Кількість доступної блокової пам’яті для різних ПЛІС є різною. Блокова пам’ять активно використовується за реалізації систем передачі та обробки інформації.

Якщо програмісти звикли до того, що розмір найменшої числової змінної дорівнює 1 байту, то в блоковій пам’яті ПЛІС можна розмістити значення, розмірність яких не є кратною байту, наприклад, 4 біти. У зв’язку з цим, обсяг блокової пам’яті для ПЛІС вказується в одиницях не на основі байтів, а на основі бітів: кбіт, Мбіт.

Опис блокової пам’яті на мові VHDL є простим та навіть включений до шаблонів середовищ розробки. Розглянемо один із варіантів на наступному прикладі.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity mem is
  port(
    clk: in std_logic;
    addr: in std_logic_vector(9 downto 0);
    we: in std_logic;
    data: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0)
  );
end entity;

architecture memory of mem is
  type mem_arr is array(0 to 1023) of std_logic_vector(3 downto 0);
  signal mem_signal: mem_arr := (others => (others => '0'));
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
        mem_signal(conv_integer(addr)) <= data;
      end if;
      q <= mem_signal(conv_integer(addr));
    end if;
  end process;
end architecture;
```

У даному прикладі наведено опис однопортової синхронної блокової пам'яті з можливістю запису (RAM). Набір сигналів для такої пам'яті є наступним:

а) clk – тактовий сигнал;
б) addr – шина адреси;
в) data – шина вхідних даних;
г) q – шина вихідних даних;
д) we (від англ. write enable) – сигнал дозволу на виконання запису.

Особливість даного опису полягає в тому, що у ньому використовується користувацький тип. Користувацький тип можна визначити за допомогою ключового слова **type**. У даному випадку новий тип описаний як **масив** значень, які, у свою чергу, також є масивами. Таким чином, блокова пам'ять у VHDL представляє собою двовимірний масив. У деклараційній секції архітектури знаходиться сигнал даного типу, який ініціалізовано за допомогою вкладених операторів **others**. Усі значення пам'яті встановлені в '0'. Пам'ять містить 1024 значення, розрядність яких становить 4 біти. Для того, щоб мати доступ до всіх даних у пам'яті, ширина шини адреси має становити 10 біт ($1024 = 2^{10}$). Обране значення не є випадковим: рекомендується описувати пам'ять таким чином, щоб її глибина (кількість елементів) дорівнювала ступеню числа 2. Для організації доступу до конкретного значення за індексом використовується функція **conv_integer**, яка перетворює значення адреси в цілочислове значення.

Засобами VHDL можна описати також ROM-пам'ять. Пам'ять типу ROM відрізняється тим, що в неї відсутній сигнал дозволу на запис. Її вміст має бути визначений на етапі формування бітового файлу конфігурації, оскільки записати в неї дані під час роботи схеми неможливо.

Відносно наведеного прикладу, то однопортова пам'ять дозволяє або зчитувати, або записувати дані. У тому випадку, якщо необхідно виконувати операції читання та запису одночасно, необхідний інший тип пам'яті – двопортова (англ. dual-port). При цьому запис та зчитування мають виконуватись за різними адресами.

Проведемо симуляцію наведеного прикладу в ModelSim (рис. 35).

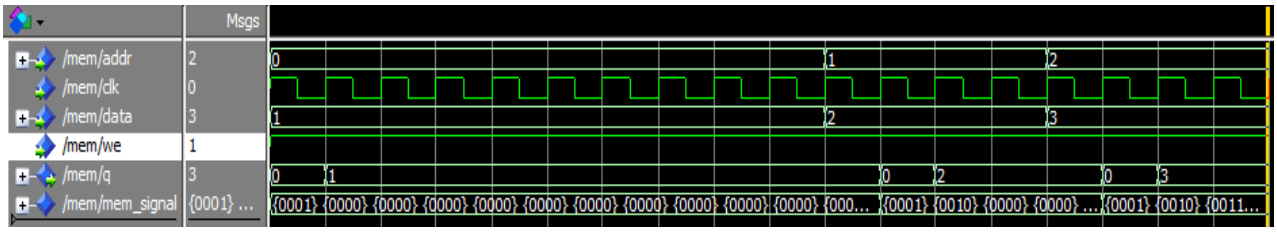


Рисунок 35 – Симуляція однопортової пам’яті

Переглянути вміст пам’яті можна у вікні Wave за допомогою спливаючої підказки (рис. 36).

```
sim:/mem/mem_signal @ 1666 ns
   0 : 0001 0010 0011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  16 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  32 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Рисунок 36 – Представлення вмісту пам’яті

Проте такий спосіб за великих обсягах пам’яті та необхідності доступу до всіх елементів не є достатньо зручним. Для перегляду вмісту пам’яті в ModelSim доступне спеціальне вікно Memory List. Увімкнути його можна командою меню **View/Memory List**. У даному вікні відображається список пам’ятей, які використовуються в схемі, що моделюється. Для прикладу доступна одна пам’ять. Переглянути вміст даної пам’яті можна за допомогою команди контекстного меню **View Contents** (рис. 37)

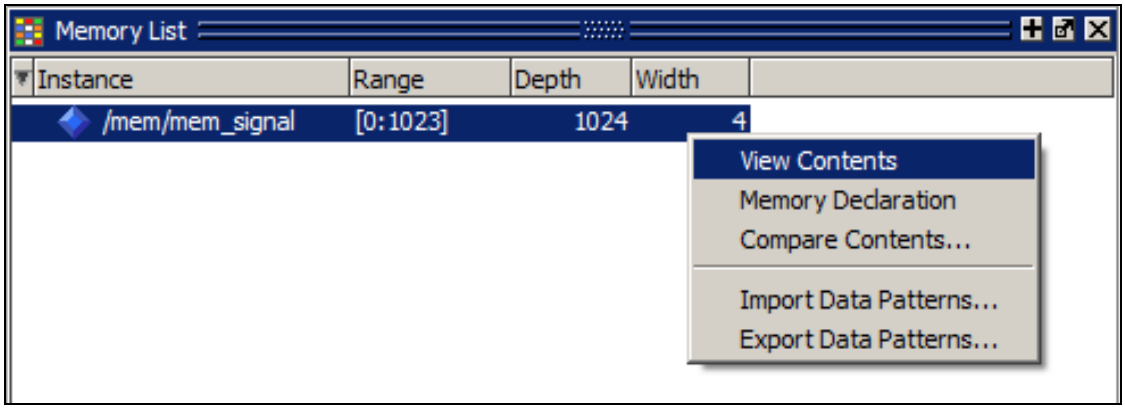


Рисунок 37 – Отримання доступу до пам’яті

У результаті має відкритися ще одне вікно для перегляду вмісту пам’яті – Memory Data (рис. 38). Контекстне меню даного вікна містить широкий набір налаштувань (кількість значень у рядку, система числення, тип даних для відображення) для представлення пам’яті відповідно до потреб користувача.

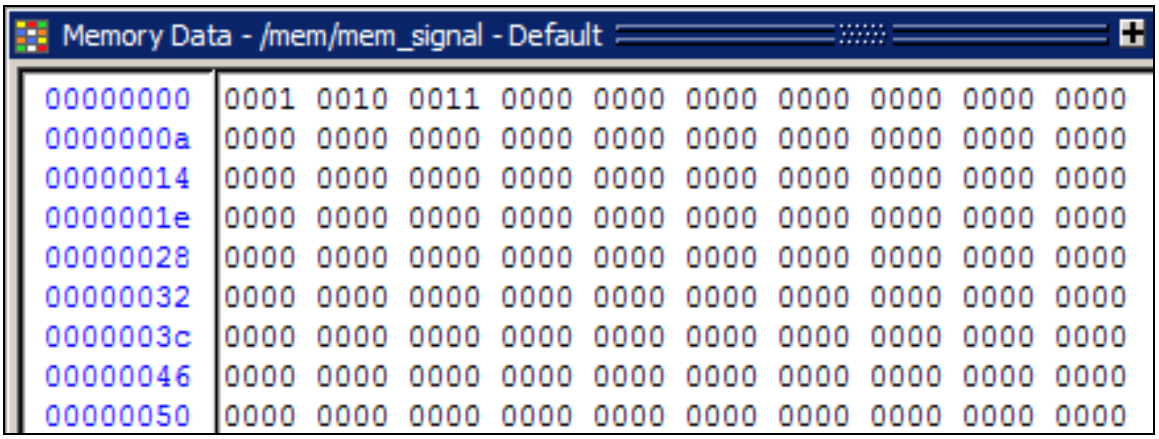


Рисунок 38 – Представлення вмісту пам’яті під час симуляції

При роботі з пам'яттю корисною може стати (особливо для перевірки) можливість експорту у файл за допомогою команди контекстного меню Export Data Patterns.

5.4 Створення тестових оточень (testbenches)

Незважаючи на те, що тестування попередніх прикладів проводилось вручну (додавання сигналів, їх налаштування, встановлення значень та ін.), такий підхід може застосовуватися лише в навчальних цілях. З метою автоматизації тестування, створення та випробування схеми в різних умовах (вхідні дані, внутрішні стани системи та ін.) загальноприйнятою стала практика використання тестових оточень (англ. testbenches).

Для програмістів! *Тестове оточення за своїм призначенням аналогічне **unit**-тестам у мовах програмування.*

Створення тестових оточень у більшості випадків виконується на основі наступної послідовності:

1) підключаються необхідні бібліотеки та пакети;
2) описується інтерфейс на частині (**entity**), що не має вхідних та вихідних сигналів: секція **port** відсутня;
3) у декларацій ній частині архітектури, що реалізує даний інтерфейс, додається опис компонента для тестування, а також сигналів, що будуть підключені до вхідних та вихідних портів компонента;
4) створюється екземпляр компонента, тестування якого проводиться (англ. unit-under-test, UUT, або design-under-test, DUT); підключаються сигнали (**port map**);
5) створюється процес, який імітує тактовий сигнал для схеми;
6) створюється процес або процеси для подачі вхідних сигналів (stimulus); подається загальний сигнал **RESET**, після чого подаються дані для симуляції.

Перевірка результатів роботи програми може проводитись або візуально у вікні моделювання, або на основі вихідних даних (текстові файли, імпортовані дампи пам'яті), отриманих у результаті симуляції, шляхом порівняння їх з очікуваними, або за допомогою інструкції мови VHDL **assert**.

Для програмістів! *Таким чином, хід проведення тестування у VHDL аналогічний відомому принципу написання **unit**-тестів у мовах програмування – AAA (Assign-Action-Assert). Відповідниками **testbench**'ів, які пропонуються засобами програмування основних мов, є модульні тести. Тим не менш, є велика відмінність між тестуванням. На відміну від шаблону, якого рекомендовано дотримуватись у мовах програмування, структура тестового модуля має набагато менше обмежень і на розробника покладено значно більше відповідальності за розробку структури тестового випадку.*

Написання тестових оточень для перевірки роботи розроблених компонентів є правилом хорошого тону для розробника схемотехніки.

Розглянемо основні особливості створення testbench'ів на прикладі з використанням компонента однопортової пам'яті.

Наведений приклад містить усі раніше вказані кроки формування коду тестового оточення. Тестування проводитиметься в такій послідовності: спочатку за всіма адресами записується певне значення, після чого дані зчитуються з пам'яті та перевіряється відповідність зчитуваних значень очікуваним.

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

entity mem_test is
end entity;

architecture test of mem_test is
  component mem is
  port(
    clk: in std_logic;
    addr: in std_logic_vector(9 downto 0);
    we: in std_logic;
    data: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0)
  );
  end component;
  constant period: time := 100 ns;
  signal clk: std_logic;
  signal addr: std_logic_vector(9 downto 0);
  signal we: std_logic;
  signal data: std_logic_vector(3 downto 0);
  signal q: std_logic_vector(3 downto 0);
begin
  memory: mem port map(
    clk => clk, addr => addr, we => we, data => data, q => q
  );
  clock: process
  begin
    clk <= '1';
    wait for period/2;
    clk <= '0';
    wait for period/2;
  end process;

  stimulus: process
    variable exp_val, val: std_logic_vector(3 downto 0);
  begin
    -- init signals
    addr <= (others => '0');
    we <= '1';
    data <= "0001";
    wait for period;
    -- write data to memory
    for i in 0 to 1023 loop
      addr <= std_logic_vector(unsigned(addr) + 1);
      data <= to_stdlogicvector(to_bitvector(data) rol 1);
      wait for period;
    end loop;
    -- read data
    exp_val := "0001";
    we <= '0';
    addr <= (others => '0');
    for i in 0 to 1023 loop
      wait for period;
      addr <= std_logic_vector(unsigned(addr) + 1);
      val := q;
      assert val = exp_val report "Values are not equal";
      exp_val := to_stdlogicvector(to_bitvector(exp_val) rol 1);
    end loop;
    wait;
  end process;
end architecture;
```

В описі наявні 2 процеси, в яких відсутні списки чутливості. Загалом, це характерно для написання тестових оточень, оскільки вони є модулями верхнього рівня і мають самі визначати, як керувати подачею даних на вхід модуля/-ів та зчитувати їх виходи. Це, як вже зазначалось, означає що в тілі процесу має міститись **wait**. Один із

процесів використовується для подачі тактового сигналу (позначений міткою clock). Всередині тіла даного процесу наявні інструкції

wait for period/2,

які дозволяють організувати затримку на заданий часовий період. Така інструкція **wait** вимагає вказання часу затримки. Для цього у VHDL доступний спеціальний тип – **time**. Його особливістю є те, що окрім числового значення для нього вказуються також одиниці виміру (в даному прикладі – наносекунди, нс). Зазвичай значення типу **time** описується як константа

Такі інструкції **wait** допускається використовувати при тестуванні схем, проте код, що реалізує компонент, не має містити таких інструкцій, оскільки, фактично, це означає, що необхідно згенерувати сигнал із нічого (без джерела), що, звичайно ж, неможливо. Процес clock працює постійно протягом симуляції.

Інший процес відповідає за подачу вхідних імпульсів на компонент та проведення перевірки. У ньому використовуються дві змінні для перевірки коректності запису. Даний процес можна умовно розділити на три логічні секції, що слідують принципу Assign-Action-Assert.

За реалізації даного процесу використовуються цикли **for…loop**. У даному випадку вони відповідають за організацію подачі значень на компонент. Це забезпечується завдяки наявності інструкції затримки **wait** всередині циклу. Така організація коду означає, що цикл виконається не одразу, вона може використовуватись лише для проведення тестування.

У частині процесу, що відповідає за зчитування та перевірку значень, при роботі з числами використовується приведення типів. У той же час, для обертання числового значення та подальшої зворотної операції застосовуються функції перетворення типів (**to_bitvector** та **to_stdlogicvector**).

5.5 Можливості автоматизації виконання завдань. Засоби мови Tool Command Line – Tcl

Тестові оточення відіграють велику роль у коректній організації роботи розробника та дозволяють значно зменшити час на проведення тестування за незначних змін, які вносяться до функціональних блоків, що тестуються. Уміння коректно описувати тестові оточення за значимістю не поступається за своєю важливістю реалізації самих модулів. Проте, якщо оцінювати час роботи розробника над проєктом, то важливим ресурсом є також час, який витрачається на взаємодію з графічним інтерфейсом середовища. Натискання клавіш, вибір необхідного елементу меню, вибір коректної послідовності дій, які необхідно виконати для того, щоб повторно запустити тестування блоків – усі ці операції слід повторно виконувати для відтворення необхідної тестової ситуації.

Саме з метою автоматизації більшості користувацьких дій у середовищі моделювання наявна підтримка скриптів. Для їх написання використовується мова **Tcl** (Tool command line). Ця мова є похідною від мови **Perl** і, фактично, повністю наслідує синтаксис. Доступні спеціальні команди (які є зверненнями до утиліт командного рядка – компіляторів, засобів по роботі з проєктом), які дозволяють значно спростити повторне виконання заданих дій.

Серед доступних команд першою, яку необхідно знати, є команда компіляції вихідних файлів – **vcom** (від **V**HDL **com**piler). Дана команда приймає на вхід як параметр шлях до файлу, що його необхідно скомпілювати.

Команди **Tcl** можна вводити безпосередньо у вікні **Transcript** (рис. 39).

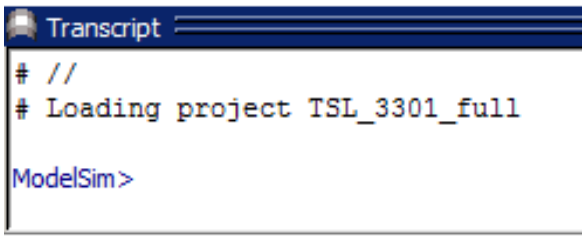


Рисунок 39 – Вікно команд **Tcl**

Варто зазначити, що компілювання шляхом виконання команди призводить до того, що оновлюється лише бібліотека, але візуальний компонент поряд з файлом не оновлюється. Тим не менш, оновлення бібліотеки означає, що внесені зміни будуть враховані під час моделювання модулю.

Таким чином можна виконувати одиночні команди. Проте, у випадку, коли необхідно повторювати послідовність команд, ручне введення команд займе забагато часу, тому команди організовуються у файли скриптів. Скрипти, зазвичай, мають розширення **.tcl** або **.do**. Для запуску скриптів використовується окрема команда – **do**. Їй як параметр передається шлях до файлу скрипту, який необхідно виконати.

Іншою важливою командою є **vsim** – команда запуску симуляції. Цій команді як параметр передається ім’я модулю, який потрібно обрати як модуль верхнього рівня. Також можлива передача деяких допоміжних пара-

метрів. Після успішного виконання команди **vsim** рядок запрошення змінює назву з **ModelSim** на **vsim**, вказуючи на те, що середовище знаходиться в стані моделювання.

Варто зазначити, що файли, які відповідають за конфігурацію відображення сигналів у вікні **Wave**, також, фактично, є скриптами **Tcl**. У них, у більшості випадків, використовується команда для додавання необхідних сигналів – **add wave**. Дана команда є двоскладовою, тобто, використання обох компонентів команди є обов'язковим.

Розглянемо також деякі інші команди (табл. 1), які виконують службові функції, проте без них неможливою є коректна автоматизація процесу моделювання.

Таблиця 1

Назва команди	**Призначення**
vdel	Видалення модуля з вказаної бібліотеки
vlib	Створення нової бібліотеки для розробки
vmap	Відображення між логічним ім'ям бібліотеки та фізичним розташуванням директорії

Скрипт, який забезпечуватиме автоматизацію процесу запуску необхідного тестового оточення для моделювання, зазвичай, виконуватиме наступні дії:

1) видалятиме усі попередньо скомпільовані модулі з бібліотеки;
2) створюватиме робочу бібліотеку з назвою **work**;
3) відображатиме робочу бібліотеку на директорію, яка створена на диску;
4) компілюватиме усі файли, які необхідні для моделювання конкретного модуля (команда **vcom**);
5) запускатиме процес моделювання (команда **vsim**);
6) додаватиме усі необхідні сигнали для досліджень у вікно **Wave**.

Така послідовність дій є універсальною і може повторюватись необхідну кількість разів із досягненням одного і того самого стану середовища. Єдиним зауваженням щодо використання цього сценарію є необхідність завершення симуляції або через меню **Simulation\End Simulation**, або введенням команди **quit –sim** у командному рядку симулятора.

Варто також зазначити, що послідовність файлів у секції компіляції є важливою. На момент компіляції модуля верхнього рівня мають бути скомпільовані усі модулі, на яких він базується. Невиконання даної умови призведе до помилки компіляції, оскільки відповідні модулі не будуть доступними.

У процесі встановлення середовища ModelSim робить запит до користувача, чи додавати системну директорію ModelSim з усіма інструментами у системну змінну PATH. Таким чином, вони стають доступними для запуску безпосередньо з командного рядка. Дана можливість надає можливість виконувати певні дії без запуску середовища.

5.6 Файл конфігурації modelsim.ini

Початкові налаштування, з якими виконується запуск середовища, знаходяться у файлі modelsim.ini. Даний файл розташований у директорії, у якій встановлено програму. Він містить велику кількість конфігурацій, які впливають на майже всі аспекти роботи програмного забезпечення. Так, наприклад, саме з цього файлу зчитуються налаштування версії мови VHDL, яка має бути застосована при компіляції файлів, або те, що робоча директорія проєкту називається саме work та ін.

Редагування даного файлу повинно виконуватись обачливо: обов'язково перед його редагуванням необхідно зробити резервну копію для того, щоб мати можливість відновити попередній стан середовища. Слід також зауважити, що налаштування за замовчуванням у більшості випадків є достатніми і змінювати їх немає потреби. Редагування даного файлу є доцільним лише у випадках тонкого налаштування середовища для специфічних проєктів, які потребують активації функцій середовища. Деякі налаштування середовища, які впливають на зміст файлу **modelsim.ini** можуть бути виконані безпосередньо з графічного інтерфейсу користувача.

Контрольні питання

1. Чи блок (block) містить послідовні оператори?
2. Оператор блоку (block) може мати охоронний вираз?
3. Пакет може мати охоронний вираз?
4. Ім'я процесу задекларовано після слова process?
5. Лічильник у циклі типу for є змінною, яку необхідно декларувати на початку процесу, у якому цикл використовується?

6. Чи сигналам призначаються значення тоді, коли всі процеси в даному такті моделювання виконані?
7. При декларації зростаючого спектра використовується ключове слово downto?
8. Ключове слово to вживається при декларації спадного спектра?
9. Для сигналів типу real ніколи не потрібна роздільна здатність?
10. Ідентифікатор @mail коректний?
11. Ідентифікатор 8mail є коректним?
12. Ідентифікатори Abc7, aBC7 є різними?
13. Рядкові літерали 'b', 'B' є різними?
14. Чи функція може бути декларована в пакеті?
15. Змінні можуть використовуватися для зберігання проміжних даних всередині процесів?
16. Чи рекомендується використовувати сигнали для зберігання проміжних даних усередині процесів?
17. Умовний оператор if має бути завершений ключовими словами end if?
18. Тільки сигнали можна використовувати як переносники інформації між процесами?
19. Вибіркове призначення сигналу представляє ту саму дію, що і умовне призначення сигналу, тільки вони записані в різний спосіб?
20. Перш ніж використовувати пакет STD_LOGIC_1164, декларація entity має бути обов'язково визначена операторами library, use?
21. Типи std_logic та std_logic_vector є промисловим стандартом для логічних сигналів і можуть застосовуватися для сигналів з багатьма джерелами, оскільки ці типи є «розв'язними»?
22. Чи є архітектура множиною паралельних операторів, які взаємодіють між собою і перебувають під впливом один одного?
23. Оператори процесу (process) не можуть використовуватися разом із операторами (<=) призначення сигналу у тієї же архітектурі?
24. Чи оператор null призначає нульове значення сигналу типу bit?
25. Чи оператор next дозволяє перейти до виконання наступного (по порядку запису) оператора архітектурного тіла?
26. Чи оператор next дозволяє перейти до виконання наступного (по порядку запису) оператора блоку?
27. Чи може оператор next перейти до виконання наступної ітерації циклу?
28. Оператор exit дозволяє завершити виконання операторів архітектурного тіла?
29. Чи можна за допомогою оператора assert видати повідомлення без перевірки умови?
30. Чи можна за допомогою оператора wait висловити нескінченне очікування?

6.ВІДЛАГОДЖЕННЯ РОБОТИ СХЕМ НА VHDL ТА ПЕРЕВІРКА КОРЕКТНОСТІ СТАНУ

6.1 Відлагодження засобами ModelSim

Процес відлагодження для програм, написаних такими мовами програмування, є важливим етапом перевірки коректності роботи програми та пошуку помилок. Як не дивно, але ModelSim також надає можливість відлагодити роботу схеми на VHDL.

Для відлагодження розробник має перейти в режим симуляції певного компонента, відкрити файл VHDL з описом та встановити точку переривання (англ. breakpoint) кліком вказівника поряд із номером рядка (рис. 40).

```
54      addr <= (others => '0');
55      for i in 0 to 1023 loop
56        wait for period;
57        addr <= std_logic_vector(unsigned(addr) + 1);
58        val := q;
59        assert val = exp_val report "Values are not equal
60        exp_val := to_stdlogicvector(to_bitvector(exp_val
61      end loop;
```

Рисунок 40 – Точка переривання

Після цього достатньо запустити симуляцію (команди **Run** або **Run All**), і симулятор призупинить роботу схеми на активній точці переривання (рис. 41).

```
54      addr <= (others => '0');
55      for i in 0 to 1023 loop
56        wait for period;
```

Рисунок 41 – Точка переривання активована

Це надасть можливість переглянути поточний стан сигналів, змінних для схеми. Крім того, після зупинці в точці переривання можна або продовжити роботу симулятора, або використати команди покрокової роботи для того, щоб більш детально проаналізувати роботу певної ділянки коду VHDL.

6.2 Створення цифрових схем з використанням процесів

У даній частині буде розглянуто два основні підходи до створення цифрових схем із використанням процесів:

1) на основі організації конвеєру;
2) на основі організації скінченних автоматів.

Принцип конвеєризації знайшов широке застосування в схемо технічному проєктуванні завдяки тому, що дозволяє досягти значних результатів з швидкодії. Мабуть, найбільш відомим прикладом конвеєру є мікропроцесор сучасних ПК: виконання інструкцій розділене на декілька стадій, усі стадії виконуються одночасно, результат попередньої стадії є входом наступної. Завдяки цьому вдається організувати виконання інструкції за один процесорний такт, а не за кількість тактів, що відповідає кількості стадій.

Розглянемо організацію конвеєрних обчислень з використанням процесів у VHDL на такому прикладі.

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

entity pipeline is
  port(
    clk: in std_logic;
    rst: in std_logic;
    in1: in std_logic_vector(3 downto 0);
    in2: in std_logic_vector(3 downto 0);
    in3: in std_logic_vector(3 downto 0);
    in4: in std_logic_vector(3 downto 0);
    result: out std_logic_vector(5 downto 0)
  );
end entity;

architecture basic of pipeline is
  signal sum1, sum2: signed(4 downto 0);
  signal sum_res: signed(5 downto 0);
begin
  process(all)
  begin
    if rst = '1' then
      sum1 <= (others => '0');
      sum2 <= (others => '0');
      sum_res <= (others => '0');
    elsif clk'event and clk = '1' then
      sum1 <= resize(signed(in1), 5) + resize(signed(in2), 5);
      sum2 <= resize(signed(in3), 5) + resize(signed(in4), 5);
      sum_res <= resize(signed(sum1), 6) + resize(signed(sum2), 6);
    end if;
  end process;
  result <= std_logic_vector(sum_res);
end architecture;
```

У прикладі представлено двостадійний суматор. На вхід подаються 4 значення шириною 4 біти. Організація додавання для 4 значень виконана попарно – одночасно визначається сума двох значень, яка записується для регістра. Зверніть увагу, що всередині компонента використовуються сигнали типу **signed**, проте для спілкування з іншими компонентами використовуються сигнали типу **std_logic_vector**, як того вимагають рекомендації. В описі активно застосовується приведення типів.

Для програмістів! *Незважаючи на те, що у мовах програмування додавання 4 значень можна реалізувати в одному рядку програмного коду, принцип реалізації даних дій для цифрової схеми є відмінним. 4 вхідні сигнали розбиваються на 2 пари і 2 операції додавання виконуються над 2 операндами. Результати додавання на першій стадії подаються на другу, де отримується кінцеве значення. Незважаючи на те, що на VHDL можна описати дані дії більш лаконічно (виконати 3 операції додавання одразу), проте такий підхід є більш складним з погляду використання логічних ресурсів ПЛІС. Це може зменшити загальну максимальну тактову частоту роботи схеми, а, отже, зменшити швидкість її роботи. Саме тому рекомендується виконувати розділення операцій та створення конвеєру для підвищення ефективності роботи схеми.*

Для того, щоб запобігти можливому переповненню при додаванні двох чисел (наприклад, 7 (0111)+7 (0111)=14 (1110) потребує для представлення 4 бітів, проте у випадку чисел зі знаком це число стане від'ємним) використовується функція **resize**. Вона дозволяє збільшити (можливе також зменшення) розрядність з урахуванням поточного знаку числа. У результаті переповнення при додаванні чисел не відбувається.

Проведемо симуляцію для наведеного вище опису (рис. 42).

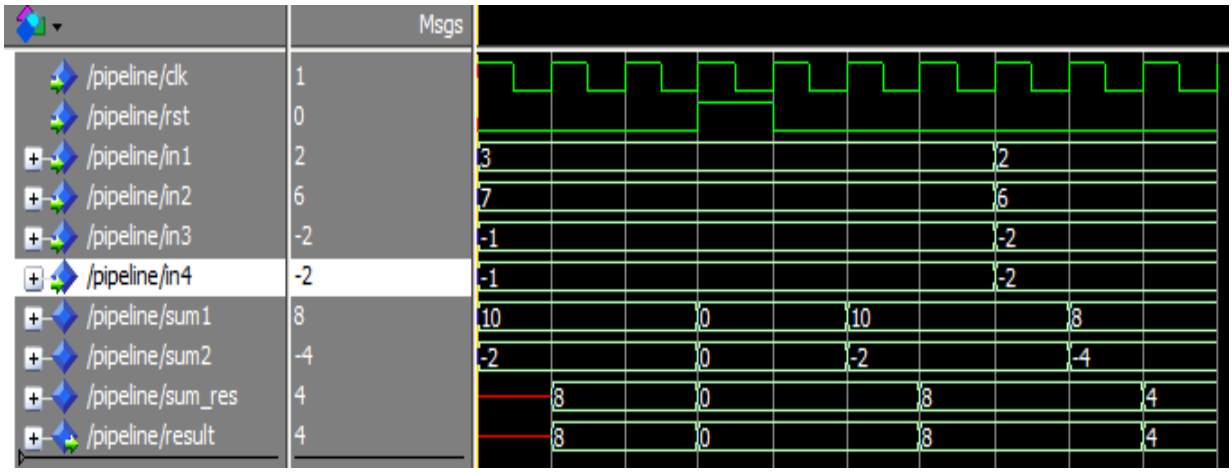


Рисунок 42 – Моделювання конвеєру обчислень

Читачеві варто звернути увагу на ту частину симуляції, яка відбувається після подачі активного рівня сигналу **rst**. Значення внутрішніх регістрів компонента ініціалізується як «0» у результаті скидання. Після того, як робота схеми розпочата, проходить 1 такт для того, щоб отримати результати на першій стадії конвеєру. Ще один такт необхідний для того, щоб отримати кінцевий результат. Продемонстровано також те, як працює компонент при зміні сигналів. Результуюче значення на виході при використанні обчислювального конвеєру отримується не одразу, а через певний сталий період, який залежить від **довжини (кількості стадій) конвеєру**. Це необхідно враховувати при зчитуванні даних.

Загалом можна сказати, що змінювати вхідні сигнали для конвеєру можна на кожному такті, узгодивши при цьому зчитування результатів відповідно до довжини конвеєру; при цьому результат обчислень можна отримувати на кожному такті, що дозволяє досягти високої продуктивності схеми.

6.3 Реалізація схем на основі скінченних автоматів

Іншим підходом до опису схем на основі процесів є підхід на основі скінченних автоматів (англ. finite state machine, FSM).

З теорії цифрових автоматів, читачу має бути відомо, що автомати прийнято розділяти на автомати Мура (вихід залежить лише від стану) та автомати Мілі (вихід залежить від стану та від входу). Обидва типи автоматів можливо реалізувати на VHDL.

Створення опису на основі скінченних автоматів слідує багатьом шаблонам, тому їх знання та розуміння є важливим для розробника. У даному підрозділі буде розглянуто деякі з них.

Скінченний автомат на VHDL можна описати з використанням трьох різних стилів:

1) однопроцесний – усі переходи та дії для станів визначені в одному процесі;
2) двопроцесний – переходи між станами та дії для станів рознесені у два різні процеси;
3) трипроцесний – логіка переходів між станами, перехід між станами та дії для станів визначені у трьох окремих процесах.

На практиці найчастіше використовуються два останні варіанти, тому навчальний приклад буде наведено для одного з них. У прикладі перетворимо конвеєрну реалізацію суматора на суматор на основі скінченного автомату. Якщо конвеєрному варіанті, усі дії виконуються одночасно, то для автомату характерне своєрідне рознесення дій у часі. Автомат можна представити як конвеєр, у якого активною є лише одна стадія.

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

entity fsm_example is
  port(
    clk: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    in1: in std_logic_vector(3 downto 0);
    in2: in std_logic_vector(3 downto 0);
    in3: in std_logic_vector(3 downto 0);
    in4: in std_logic_vector(3 downto 0);
    res: out std_logic_vector(5 downto 0);
    ready: out std_logic
  );
end entity;

architecture fsm of fsm_example is
  type states is (s_idle, s_first, s_finish);
  signal cur_state, next_state: states;
  signal sum1, sum2: signed(4 downto 0);
  signal sum_res: signed(5 downto 0);
begin
  process(all)
  begin
    if clk'event and clk = '1' then
      cur_state <= next_state;
    end if;
  end process;

  process(all)
  begin
    case cur_state is
      when s_idle =>
        if start = '1' then
          next_state <= s_first;
        else
          next_state <= s_idle;
        end if;
      when s_first =>
        next_state <= s_finish;
      when s_finish =>
        next_state <= s_idle;
    end case;
  end process;

  process(all)
  begin
    if rst = '1' then
      sum1 <= (others => '0');
      sum2 <= (others => '0');
      sum_res <= (others => '0');
      ready <= '0';
    elsif clk'event and clk = '1' then
      case cur_state is
        when s_idle =>
          sum1 <= (others => '0');
          sum2 <= (others => '0');
          ready <= '0';
        when s_first =>
          sum1 <= resize(signed(in1), 5) + resize(signed(in2), 5);
          sum2 <= resize(signed(in3), 5) + resize(signed(in4), 5);
        when s_finish =>
          sum_res <= resize(sum1, 6) + resize(sum2, 6);
          ready <= '1';
      end case;
    end if;
  end process;
  res <= std_logic_vector(sum_res);
end architecture;
```

У даному прикладі представлений трипроцесний стиль опису автомату. Головною особливістю опису, яка безпосередньо стосується організації роботи автомату, є наявність користувацького типу **states**. Даний тип є типом-переліченням. Цей тип використовується для представлення станів автомату. Крім того, до інтерфейсної частини компонента додані сигнали – запуску обчислення (**start**) та сигнал готовності даних (**ready**). Використання даних сигналів є характерним для компонентів на основі скінченних автоматів: вони працюють не постійно, а очікують надходження необхідних даних, а по завершенню обробки повідомляють про це **top-level**-компоненту.

Проведемо симуляцію роботи компонента (рис. 43) відповідно до того, як буде проходити його використання.

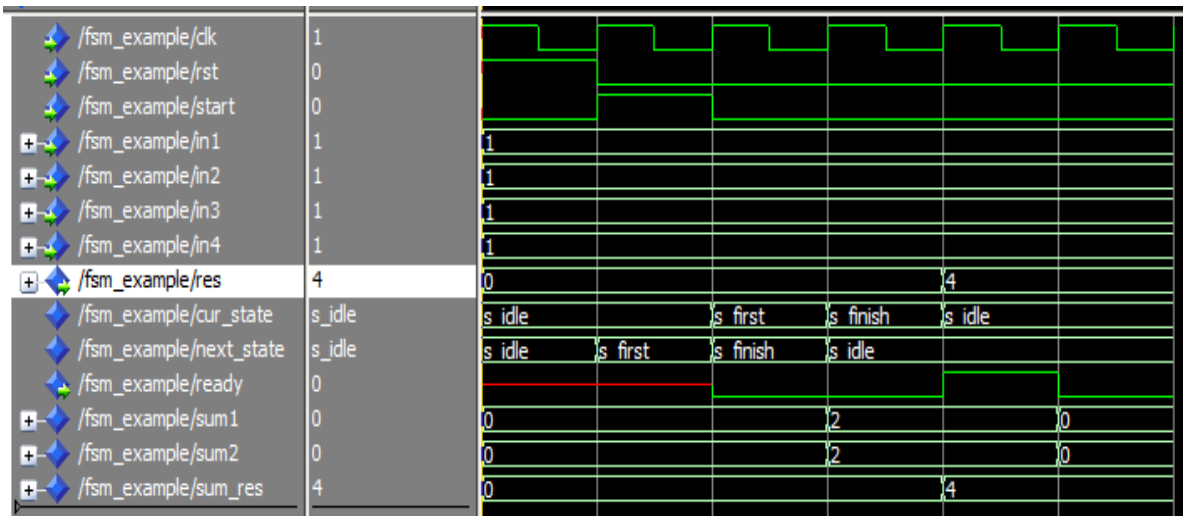


Рисунок 43 – Моделювання роботи FSM

Для коректної роботи на початку має бути поданий сигнал **rst**, який дозволить ініціалізувати внутрішні регістри компонента. Одразу після надходження сигналу **start**, починається виконання обчислень (стан **s_first**): зчитуються дані на вході та обчислюються дві суми на основі чотирьох вхідних значень. Це відповідає першій стадії розглянутого раніше конвеєру. Друга стадія виконується в результаті переходу до наступного стану (**s_finish**). На цій стадії обчислюється вихідне значення та подається сигнал готовності. Після цього компонент переходить до стану очікування.

Засобами ModelSim (у більш потужній версії) можна також переглянути структуру автомату (показано граф станів іншого автомату, який не наведено у прикладі) та те, як виконуються переходи між станами (рис. 44).

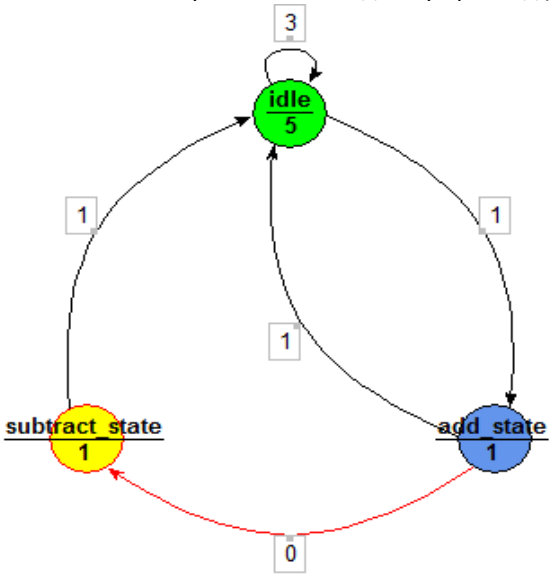


Рисунок 44 – Графічне представлення FSM

Така функціональність може бути корисною для проведення візуального відлагодження роботи скінченного автомату.

6.4 Читання/запис файлів засобами мови VHDL

Як вже зазначалось, тестування відіграє надзвичайно важливу роль у процесі розробки пристроїв на базі ПЛІС, оскільки дозволяє перевірити концептуально (без наявності фізичного пристрою) правильність розроблюваного рішення. Значення сигналів, які подаються на вхід, задаються у процесах, після чого перевіряється коректність роботи блоку. Тим не менш, якщо для простих випадків достатньо прописати усі можливі варіанти безпосередньо у тесті, то для складних модулів, які мають велику кількість входів та виходів, такий сценарій не є прийнятним, оскільки на написання такого тесту знадобиться занадто багато часу. Для того, щоб уникнути цього та зробити тестові оточення більш гнучкими, доцільно використовувати засоби для роботи з файлами, які наявні у мові VHDL.

Для програмістів! *На відміну від звичного широкого набору функцій для роботи з файлами у бібліотеках мов програмування, VHDL пропонує мінімальний необхідний набір для роботи з файлами та рядками. Про це варто пам'ятати при плануванні тестових сценаріїв, оскільки не всі вони можуть у кінцевому випадку бути реалізовані такими засобами. Дана проблема вирішується шляхом використання спеціальних бібліотек і засобів ModelSim.*

Концепція, яка пропонується для роботи з файлами у VHDL, є наступною:

1) наявний тип для представлення файлів – **file**. Доступні функції для запису і зчитування даних **файл**;
2) наявний тип для представлення рядків – **line**. Читання з файлу відбувається у змінну типу **line**;
3) зчитані дані, які були збережені у екземплярі **line**, конвертуються у необхідний тип для того, щоб можна було подати його на вхід модулю;
4) як видно з наведеного опису, функціонал є доволі простим.

З приводу написання самих тестових оточень з використанням можливостей роботи з файлами, то варто також пам'ятати, що отримувати дані з файлу не повністю, а поступово – по мірі того, як тестовий блок проводить обробку раніше поданих на нього даних. Відповідно, необхідно забезпечити синхронізацію у роботі процесів, що подають сигнали на тестовий модуль, і процесом, що читає/записує дані з файлу. Про дану особливість роботи з файлами необхідно пам'ятати. Зазвичай, це потребує відстеження стану одразу кількох сигналів для того, щоб коректно узгодити процеси між собою.

Також варто відзначити, що для функцій та типів, пов'язаних із роботою з файлами, відводиться окремі пакети – **std.textio** та **ieee.std_logic_textio**. Останній необхідний для того, щоб забезпечити роботу з типом даних **std_logic_vector** – одним з основних типів у VHDL.

Розглянемо приклад роботи з текстовим файлом для подачі значень у вигляді вектору. У даному прикладі розглянемо, які функції використовуються для роботи з файлами.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all ;
use std.textio.all;
use ieee.std_logic_textio.all;

entity text_test is
end entity ; -- text_test

architecture arch of text_test is
    file input_data: text;
    constant period: time := 100 ns;
    signal clk: std_logic := '0';
    signal input_signal : std_logic_vector(3 downto 0) := (others => '0');
begin

    clk <= not clk after period / 2;

    read_file: process
        variable in_line: line;
        variable input_variable: std_logic_vector(3 downto 0);
    begin
        file_open(input_data, "input.txt", READ_MODE);
        while not endfile(input_data) loop
            readline(input_data, in_line);
            read(in_line, input_variable);
            input_signal <= input_variable;
            wait until rising_edge(clk);
        end loop;
        file_close(input_data);
    end process;
end architecture ; -- arch
```

Перш за все необхідно звернути увагу на спеціальний тип **file**, за допомогою якого оголошується об'єкт для роботи з файлом. Відкриття і закриття файлу здійснюється за допомогою функцій **file_open** та **file_close**. Процес не містить списку чутливості, а також не містить простого оператору wait, який би вказував на завершення процесу. Це означає, що після першого виконання тіла процесу розпочнеться новий цикл виконання, відповідно, дані будуть зчитуватись із файлу постійно. Для того, щоб змінити цю поведінку, достатньо додати оператор **wait** без параметрів.

Переходячи до розгляду процесу зчитування даних з файлу, у процесі наявна змінна типу line для того, щоб зчитати рядок з файлу. Також наявна змінна типу **std_logic_vector**, у яку відбувається запис значення після зчитування рядка та подальший запис отриманого значення до сигналу. Для того, щоб цикл по роботі з файлом був розгорнутий у часовому вимірі, наявна інструкція **wait**, яка очікує на фронт тактового сигналу.

Файл, який має бути підготовлений із даним тестовим оточенням, повинен мати структуру, наведену на рис. 45.

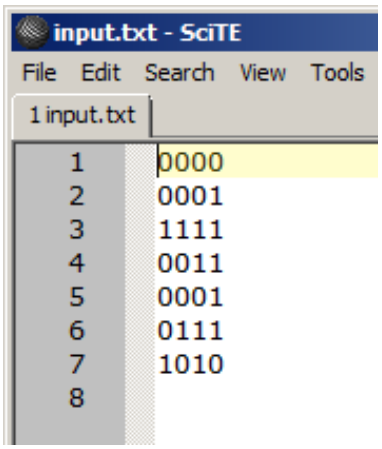


Рисунок 45 – Вміст текстового файлу

Кількість рядків у файлі не має значення, оскільки у прикладі перевіряється умова закінчення файлу **endfile**, а не конкретна кількість рядків. Під час моделювання має спостерігатись наступна ситуація (рис. 46), коли зчитаний шаблон постійно повторюється на діаграмі сигналів.

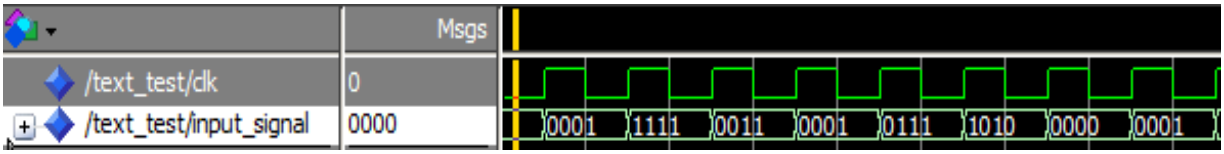


Рисунок 46 – Видача сигналів, зчитаних з файлу

Для того, щоб організувати більш складний сценарій зі зчитуванням файлів, є варіант, коли дані просто розміщені у різних файлах. Для роботи з різними типами даних, які розміщені в одному файлі, необхідно за допомогою функції read зчитувати відповідні дані у необхідні змінні. Проміжки (або інші символи, обрані в якості розділювачів) також необхідно зчитувати, оскільки таким чином реалізована обробка рядка у VHDL. Відповідно, за допомогою кількох послідовних викликів функції read з одним і тим самим рядком у якості параметру, можна отримати усі необхідні дані у змінні, після чого проводити запис у сигнали тестового модулю.

Запис у файл відбувається за допомогою функції **writeline**, яка як параметри приймає дескриптор файлу, а також рядок, який необхідно записати. У той же час функція **write** має деякі відмінності від свого аналогу для зчитування. Вона приймає як параметр рядок, дані, які необхідно записати, значення, яке вказує, як необхідно організувати вирівнювання, а також кількість символів, яку необхідно виділити під значення, яке буде записане у рядок.

При цьому, за виконання запису у файл, варто пам'ятати, що до реалізації запису у файл під час моделювання необхідно ставитись так само уважно, як і до зчитування. Справа у тому, що запис так само реалізується всередині тіла процесу і, якщо процес буде виконуватись постійно, а, відповідно, і запис, то файл може досягти дуже великих розмірів. Тому запис, зазвичай, проводиться при виконанні певних логічних умов у ході роботи тестового оточення.

Для того, щоб розглянути безпосередньо приклад запису даних у файл під час виконання коду тестового оточення, проаналізуємо наступний приклад, який був доданий як окремий процес у попередній файл. Необхідно додати декларацію об'єкта вихідного файлу у секції декларації архітектури.

```
write_file: process
    variable out_line: line;
begin
    file_open(output_data, "output.txt", WRITE_MODE);
    wait until rising_edge(clk);
    for i in 1 to 15 loop
        write(out_line, input_signal, RIGHT, 12);
        writeline(output_data, out_line);
        wait until rising_edge(clk);
    end loop;
    file_close(output_data);
    wait;
end process;
```

Як вже згадувалось, є деякі відмінності порівняно з читанням файлу. Перш за все, зміна порядку виклику функцій. По-друге, наявність виклику wait без параметрів для того, щоб зупинити виконання тіла процесу після першого проходження. Якщо цього не зробити, то, по аналогії з процесом читання, запис буде виконуватись постійно і файл може зайняти дуже багато дискового простору. Для того, щоб продемонструвати запис кількох значень у файл для тесту, використовується цикл. За тестування реальних модулів доцільно виконувати запис відповідно до стану сигналу готовності даних (зазвичай, мають частинку **ready** у назві сигналу). Результат моделювання зміненого тесту призведе до створення нового файлу з таким змістом (рис. 47):

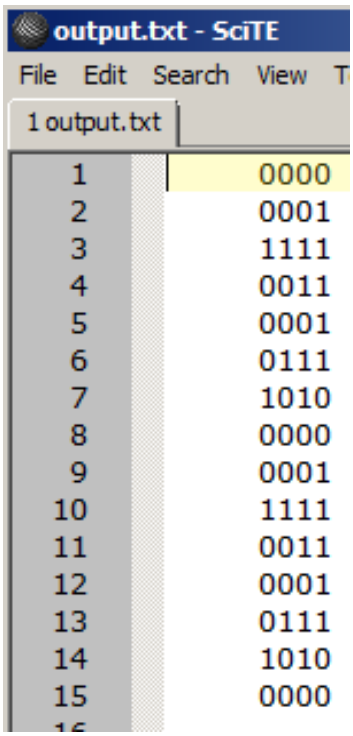


Рисунок 47 – Вихідний файл

Наявність відступів на початку рядка пояснюється тим, що було обрано вирівнювання по правому краю з шириною поля 12, в той час, як ширина записаних даних становить всього 4 символи.

6.5 Оператор assert

Оператор **assert** мови VHDL використовується для того, щоб організувати перевірку коректності очікуваної умови і видати повідомлення про статус перевірки у випадку її некоректності. Оператор assert використовується таким чином:

```
assert <condition> report <string> severity <severity_level>
```

Проводиться перевірка умови **condition** і у випадку, якщо дана умова не виконується (вираз дорівнює **false**), то виводиться рядок повідомлення string. Також вказується, наскільки серйозним є невиконання даної умови. Для даного поля можна вказувати одне з чотирьох значень – **NOTE, WARNING, ERROR, FAILURE**. Перший рівень використовується для того, щоб вказати, що це просто діагностичне повідомлення, другий – вказує на наявність певної неузгодженості, на яку слід звернути увагу, останні два – вказують на суттєві помилки, а останній варіант припиняє симуляцію. Зазвичай, повідомлення, що вказані для видачі, з'являються у вікні Transcript. Такою є інтерпретація за замовчуванням даних команд середовищем.

Розглянемо приклад використання операторів assert для проведення перевірки вихідного значення. Як модуль для тестування пропонується модуль синхронного регістра з можливістю його переводу у z-стан.

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

entity assertion is
  port(
    clk: in std_logic;
    d: in std_logic;
    rst_n: in std_logic;
    def: in std_logic;
    q: out std_logic
  );
end entity;

architecture behave of assertion is
  signal ds: std_logic;
begin
  process(all)
  begin
    if rst_n = '0' then
      ds <= '0';
    elsif clk = '1' and clk'event then
      if def /= '1' then
        ds <= d;
      else
        ds <= 'Z';
      end if;
    end if;
  end process;

  q <= ds;
end architecture;
```

Тестове оточення для даного модуля буде інстанціювати його, подавати необхідні сигнали на вхід із перевіркою вихідного стану регістра. Для того, щоб продемонструвати використання оператору assert, проводиться перевірка вихідного значення регістра. Зверніть увагу, що у даному випадку повідомлення виводиться, хоча результат на виході правильний. Більш коректною і поширеною реалізацією є ситуація, коли повідомлення виводиться, коли результат не є коректним (тобто, логічним було б використання оператора рівності, а не нерівності).

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

entity assertion_tb is
end entity;

architecture behave of assertion_tb is
  signal clock: std_logic := '0';
  signal ds, qs, defs, rsts: std_logic := '0';
begin
  clock <= not clock after 10 ns;

  inst: entity work.assertion port map (
    clk => clock,
    d => ds,
    rst_n => rsts,
    def => defs,
    q => qs
  );

  stimul_proc: process
  begin
    wait for 20 ns; -- reset all
    ds <= '1'; -- set value
    rsts <= '1';
    wait for 20 ns;
    assert qs /= '1' report "Wrong output value" severity note;
    defs <= '1';
    wait for 20 ns;
    assert qs /= 'Z' report "No z-value on output" severity note;
    wait for 20 ns;
    wait;
  end process;
end architecture;
```

Під час модулювання роботи тестового оточення у вікні **Transcript** можна побачити повідомлення (рис. 48), згенеровані саме командою **assert**.

```
VSIM 3> run
run
run
run
# ** Note: Wrong output value
#    Time: 40 ns  Iteration: 0  Instance: /assertion_tb
run
run
# ** Note: No z-value on output
#    Time: 60 ns  Iteration: 0  Instance: /assertion_tb
```

Рисунок 48 – Повідомлення у вікні **Transcript**

Окрім виводу безпосередньо самого повідомлення, середовище додає інформацію про модельний час, коли було видане дане повідомлення.

Діаграма сигналів для розробленого тестового оточення представлена на рис. 49.

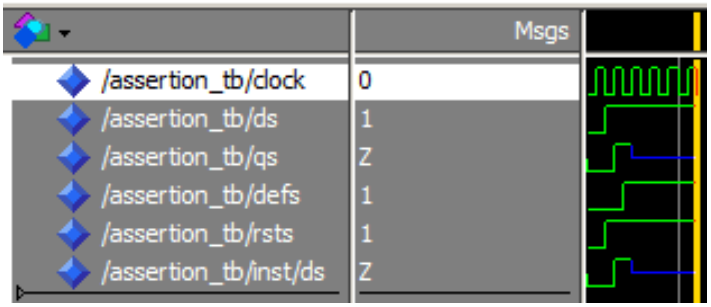


Рисунок 49 – Діаграма сигналів

У вікні **Wave** сигнал зі значенням **Z** відображається синім кольором, що відрізняє його від станів, які позначаються червоним кольором, оскільки, насправді, такий стан є коректним і можливим для реалізації. Стан **Z** на виході означає значення високого імпедансу. Такий стан виходу, на відміну від деяких значень **std_logic**, може використовуватись не лише під час симуляції, а і синтезований і реалізований у подальшому на кристалі ПЛІС, якщо наявні виходи відповідного типу. Дане значення використовується рідко, проте може знадобитись для реалізації взаємодії з деякими типами мікросхем.

Окрім відображення у вікні повідомлень, для досліджень операторів **assert** відведене окреме вікно, яке дозволяє більш детально у одному елементі управління (рис. 50, зображення з інтернету) переглянути статус кожного з таких операторів. Для того, щоб відкрити дане вікно, необхідно виконати команду **View\Coverage\Assertions**.

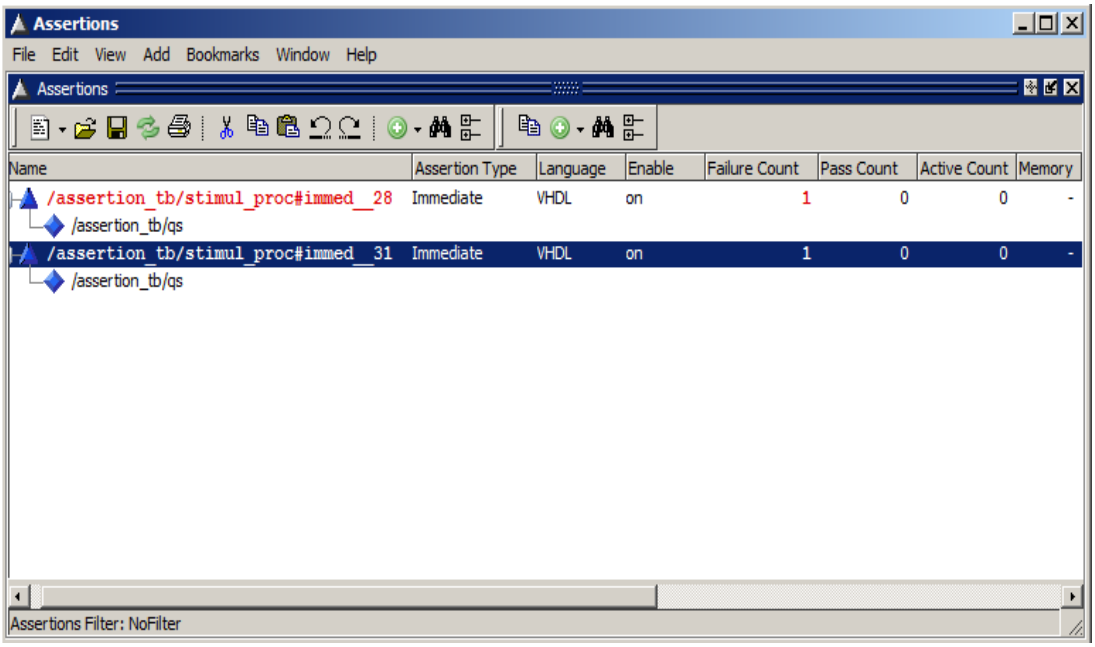


Рисунок 50 – Представлення **Assertions**

У даному прикладі продемонстровано, що у випадку, якщо умова не виконується, то відповідні пункти у списку позначаються червоним кольором. Також відстежується загальна кількість успішних проходів та проходів з помилкою, а також інша службова інформація.

Контрольні питання

1. Оператор конкретизації компонента (port map) може використовуватися як в архітектурному тілі (як паралельний оператор), так і у процесі (як послідовний оператор)?
2. Чи може локальна змінна процесу змінити своє значення кілька разів під час виконання процесу?
3. Тип bit як стандартний логічний тип не потребує вирішальної функції і може використовуватися для сигналів із багатьма джерелами?
4. Чи всі процеси в архітектурному тілі є активними весь час, коли архітектура активна?
5. Конструкція if … then може використовуватися всередині процесу?

6. Конструкція if … then може використовуватися всередині функції?
7. Цикловий параметр (лічильник числа ітерацій циклу) потрібно декларувати зовні циклу, який записується з ключовим словом for?
8. Процедура має ім’я?
9. Чи функція має ім’я?
10. Чи може функція містити послідовний оператор очікування (wait)?
11. Оператор wait зупиняє виконання процесу?
12. Оператор case може використовуватися всередині процедури?
13. Мітка компонента в операторі port map (конкретизації компонента) є необов’язковою?
14. Оператор port map може бути використаний у функції?
15. Оператор port map може бути використаний у процедурі?
16. Оператор port map може бути використаний у тілі оператора generate?
17. Оператор xor може виконуватись над операндами типу bit_vector?
18. Оператор and не може виконуватися над операндами типу boolean?
19. Оператор **or** може виконуватися над операндами типу std_logic?
20. Оператор **or** може виконуватися над операндами типу std_logic_vector?
21. Символом «:» (двокрапка) у VHDL позначається операція поділу?
22. Символом «=» VHDL позначається логічна операція еквівалентність?
23. Символом «**» у VHDL позначається операція вилучення квадратного кореня?
24. Чи може в одному операторі port map бути призначення на деякі порти одиночних біт, а на інші порти – векторів?
25. Структурний опис складається з компонентів та сигналів?
26. Чи завжди всі компоненти ієрархічного опису мають бути специфіковані, використовуючи лише поведінковий опис?

7.ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ТА ТЕСТУВАННЯ ЦИФРОВИХ СХЕМ НА БАЗІ ПЛІС

7.1 Поняття частотних доменів у мікросхемах ПЛІС

Частіше за все при проєктуванні цифрових схем на базі ПЛІС бажаною є розробка схеми, яка буде працювати на одній тактовій частоті (або, принаймні, більша частина схеми буде використовувати одну частоту і будуть необхідні мінімальні зміни для взаємодії цих частин). Відповідно, перевага з погляду простоти має віддаватись схемам з одним частотним доменом. Частотний домен (англ. clock domain) – частина мікросхеми ПЛІС, елементи якої використовують єдину тактову частоту. Відповідно, за реалізації проєкту можуть бути наявні декілька частотних доменів. У випадку необхідності їх взаємодії між собою постає задача коректної передачі даних з одного часового домену в інший.

Виділяють декілька технік, які дозволяють узгодити передачу даних між частотними доменами. Найпростішою і найпоширенішою технікою, яка використовується для вирішення цієї задачі є використання двох послідовно з’єднаних регістрів.

Розглянемо реалізацію такого синхронізатору більш детально на прикладі.

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

entity synchronizer is
  port(
    clkA: in std_logic;
    sA: in std_logic;
    clkB: in std_logic;
    sB: out std_logic
  );
end entity;

architecture behav of synchronizer is
  signal buf1, buf2: std_logic := '0';
begin
  process(all)
  begin
    if (clkB = '1' and clkB'event) then
      buf1 <= sA;
      buf2 <= buf1;
    end if;
  end process;

  sB <= buf2;
end architecture;
```

Як видно, в описі портів наявні дві тактові частоти, проте для опису функціональності модуля використовується лише одна з них – частота домену, у який необхідно передати сигнал. Відповідно, даний вхід можна просто прибрати при реальному використанні, оскільки він був доданий для наочності представлення під час моделювання. Сам же вхідний сигнал із домену A пропускається через 2 синхронні регістри, які працюють на основі тактової частоти домену B. Після цього видається результат на вихід модулю.

Розглянемо тестове оточення для проведення перевірки роботи синхронізатора.

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

entity synchronizer_tb is
end entity;

architecture tb of synchronizer_tb is
  signal periodA: time := 10 ns;
  signal periodB: time := 25 ns;

  signal clockA, clockB: std_logic := '0';
  signal valA: std_logic := '0';
  signal valB: std_logic;
begin
  clockA <= not clockA after periodA;
  clockB <= not clockB after periodB;

  synch_inst: entity work.synchronizer port map (
    clkA => clockA,
    sA => valA,
    clkB => clockB,
    sB => valB
  );

  stimul_proc: process
  begin
    wait for 100 ns;
    valA <= '1';
    wait for 100 ns;
    valA <= '0';
    wait for 200 ns;
    wait;
  end process;
end architecture;
```

У даному тестовому середовищі демонструється те, як передати імпульс із домену A у домен B. Результат моделювання, який отриманий у вікні **Wave**, представлено на рис. 51.

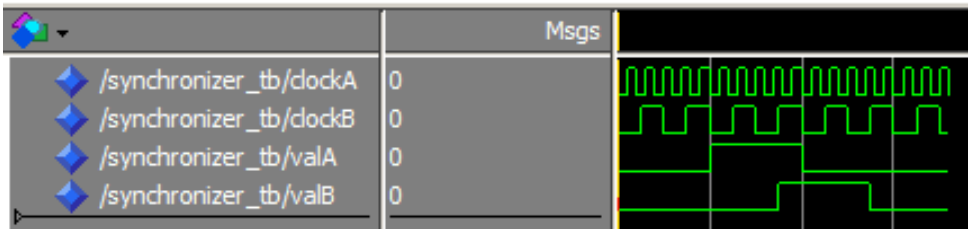


Рисунок 51 – Передача імпульсу з домену A у домен B

Як видно з рис. 51, імпульс (а точніше, просто активний сигнал) був переданий у цільовий частотний домен.

Тим не менш, недоліком такого підходу є те, що деякі сигнали можуть бути втрачені, наприклад, короткі імпульси, тривалість яких становить менше періоду імпульсу у другому тактовому домені. Тому необхідно враховувати також сигнал тактової частоти з першого домену. У даному випадку мова піде про сигнал-прапорець, який зводиться в активний стан на один період власного домену. Таким чином, це одиночний імпульс, довжина якого відповідає періоду тактової частоти домену. Для того, щоб передати такий сигнал у інший домен із нижчою тактовою частотою, необхідно використовувати іншу техніку.

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

entity flag_synch is
  port(
    clkA: in std_logic;
    sA: in std_logic;
    clkB: in std_logic;
    sB: out std_logic
  );
end entity;

architecture behav of flag_synch is
  signal toggleA: std_logic := '0';
  signal buf: std_logic_vector(2 downto 0) := (others => '0');
begin
  process(all)
  begin
    if (clkA = '1' and clkA'event) then
      toggleA <= toggleA xor sA;
    end if;
  end process;

  process(all)
  begin
    if (clkB = '1' and clkB'event) then
        buf <= buf(1 downto 0) & toggleA;
    end if;
  end process;

  sB <= buf(2) xor buf(1);
end architecture;
```

У даному підході використовується три проміжні регістри, а також враховується тактова частота домену, з якого передається імпульс. Для того, щоб зафіксувати встановлення прапорця, у модулі наявний внутрішній сигнал **toggleA**. Він змінює своє значення на протилежне при появі імпульсу. Це відбувається завдяки операції **xor**. Далі даний сигнал пропускається через ланцюг послідовно з'єднаних регістрів. Рішення про зміну стану вихідного сигналу приймається на основі застосування операції **xor** до значень двох останніх регістрів у ланцюгу. Дана операція у тому випадку, коли операнди мають однакове значення, повертає '0', а у тому випадку, якщо операнди мають різні значення, то повертається '1'.

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

entity flag_synch_tb is
end entity;

architecture tb of flag_synch_tb is
  signal periodA: time := 10 ns;
  signal periodB: time := 25 ns;

  signal clockA, clockB: std_logic := '0';
  signal valA: std_logic := '0';
  signal valB: std_logic;
begin
  clockA <= not clockA after periodA;
  clockB <= not clockB after periodB;

  synch_inst: entity work.flag_synch port map (
    clkA => clockA,
    sA => valA,
    clkB => clockB,
    sB => valB
  );

  stimul_proc: process
  begin
    wait for periodA * 2;
    valA <= '1';
    wait for periodA * 2;
    valA <= '0';
    wait for periodA * 10;
    valA <= '1';
    wait for periodA * 2;
    valA <= '0';
    wait for periodA * 10;
    wait;
  end process;
end architecture;
```

Тестове оточення у даному випадку двічі повторює одну і ту ж саму операцію: подача одиночного імпульсу у домені A. Результат моделювання роботи блоку представлено на рис. 52.

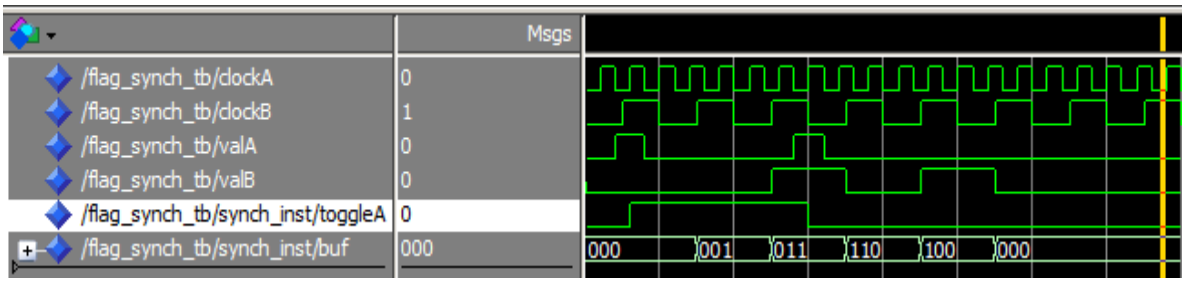


Рисунок 52 – Результат моделювання роботи блоку

Характерною особливістю у даному випадку є поведінка сигналу toggleA. Дві зміни значень цього сигналу, фактично, забезпечують передачу імпульсу в домен B. Також варто звернути увагу на затримку, з якою передаються сигнали до цільового домену.

Іншим інструментом, який дозволяє забезпечувати передачу даних між частотними доменами, є структури типу черги, які розглянуті в наступному розділі.

7.2 Реалізація модуля черги (First-in-First-Out – FIFO)

До цього моменту були розглянуті декілька типових модулів, які часто складають велику частину більш значних модулів: регістри, модулі пам'яті, лічильники. Іншим модулем, який відноситься до цього розряду, є модуль черги. Черга представляє собою модуль, який працює за принципом «перший зайшов – першим вийшов» (англ. First-in-First-Out – FIFO). Модуль черги, в основному, реалізується на основі вже розглянутого модуля пам'яті, до якого додається логіка по управлінню сигналами модулю. Мінімальний набір сигналів, які додаються у модуль черги порівняно з модулем пам'яті, складається з вихідних сигналів empty, full. Дані сигнали вказують на стан черги. Також прийнято додавати сигнали дозволу на запис та читання з черги. Ще однією відмінністю у сигналах між пам'яттю та модулем черги є відсутність сигналу адреси, оскільки вона має відслідковуватись самим модулем.

Для програмістів! *Модуль черги за своєю суттю дуже близький структурі черги, які, зазвичай, доступні як готові модулі у стандартних бібліотеках. Проте, якщо у випадку програмної черги доводиться мати справу з певною абстракцією, то у випадку апаратної реалізації черги це є окремим модулем, під який виділяються ресурси мікросхеми. Варто зазначити, що, зазвичай, у чергах зберігаються однотипні числові значення фіксованої розрядності, на відміну від черг у стандартних бібліотеках мов програмування, де можуть зберігатись повноцінні об'єкти.*

Особливістю блоків типу FIFO є те, що вони дозволяють забезпечити взаємодію між різними частотними доменами. У такому випадку блок працює на основі двох частот (наявні два відповідні входи) – для читання і для запису. Запис відбувається на одній частоті, а читання – на іншій. При цьому, на стороні домену запису перевіряється сигнал full: якщо він встановлений, то запис не може відбуватись до зміни стану даного сигналу в значення 0. Так само, на стороні зчитування перевіряється сигнал empty і на його основі вирішується, чи виконувати зчитування з модулю, чи очікувати надходження нових даних у чергу. Таким чином, достатньо лише інстанціювати такий модуль на межі двох частотних доменів, а узгодження буде забезпечуватись на рівні функціональності модуля. Особливістю такого модуля є те, що у ньому використовуються одразу два вхідні сигнали тактової частоти з різних частотних доменів.

Додатково, у багатьох реалізаціях FIFO передбачені механізми контролю рівня заповнення, які дозволяють оцінювати завантаженість черги та приймати рішення щодо регулювання потоку даних. Це може бути корисно для систем із змінними навантаженнями, де важливо не допускати переповнення або надмірного простою буфера. Для цього можуть використовуватися спеціальні порогові сигнали, такі як almost full і almost empty, що дозволяють заздалегідь реагувати на зміну стану черги та відповідним чином коригувати роботу пристрою.

Ще однією важливою характеристикою FIFO є метод організації пам'яті, яка може бути реалізована у вигляді регістрів або блокової пам'яті (BRAM) у програмованих логічних пристроях (FPGA). Використання регістрів забезпечує високу швидкість роботи, проте займає більше ресурсів, тоді як BRAM дозволяє створювати глибокі черги з мінімальними витратами на логіку. Деякі реалізації також підтримують параметризацію глибини FIFO, що дозволяє оптимізувати баланс між продуктивністю та використанням ресурсів у конкретному застосуванні.

Ще однією важливою особливістю FIFO є його можливість працювати в режимі динамічного керування глибиною черги. У деяких реалізаціях передбачені механізми автоматичного масштабування буфера залежно від поточного навантаження системи. Це дозволяє оптимально використовувати доступні ресурси, уникаючи як надмірного споживання пам'яті, так і втрати даних через недостатню ємність. Такі гнучкі FIFO-модулі можуть адаптивно змінювати свою глибину або працювати в режимах з різним рівнем пріоритетності, що особливо корисно для складних багатопотокових або реального часу систем.

Розглянемо реалізацію найпростішого модуля черги.

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

entity fifo is
  generic (
    dwidth: integer := 4;
    ddepth: integer := 5
  );
  port(
    clk: in std_logic;
    rst_n: in std_logic;
    din: in std_logic_vector(dwidth-1 downto 0);
    dout: out std_logic_vector(dwidth-1 downto 0);
    enr: in std_logic;
    enw: in std_logic;
    empty: out std_logic;
    full: out std_logic
  );
end entity;

architecture behave of fifo is
  type memory is array (0 to 2**ddepth-1) of std_logic_vector(dwidth-1 downto 0);
  signal mem: memory := (others => (others => '0'));
  signal rptr, wptr: integer := 0;
  signal emp, fl: std_logic := '0';
  signal numptr : integer := 0;
begin

  emp <= '1' when numptr = 0 else '0';
  fl <= '1' when numptr = 2**ddepth else '0';

  empty <= emp;
  full <= fl;

  dout <= mem(rptr);

  fifo_proc: process(all)
  begin
    if rst_n = '0' then
      rptr <= 0;
      wptr <= 0;
      numptr <= 0;
    elsif clk = '1' and clk'event then
      if (enw = '1' and enr = '0') then
        numptr <= numptr + 1;
      elsif (enw = '0' and enr = '1') then
        numptr <= numptr - 1;
      end if;
      if (enw = '1' and fl = '0') then
        if (wptr = 2**ddepth-1) then
          wptr <= 0;
        else
          wptr <= wptr + 1;
        end if;
      end if;
      if (enr = '1' and emp = '0') then
        if (rptr = 2**ddepth-1) then
          rptr <= 0;
        else
          rptr <= rptr + 1;
        end if;
      end if;
      if (enw = '1') then
        mem(wptr) <= din;
      end if;
    end if;
  end process;
```

Також розглянемо тестове оточення, розроблене для перевірки роботи даного модулю.

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

entity fifo_tb is
end entity;

architecture behav of fifo_tb is
  signal clock: std_logic := '0';
  signal reset: std_logic := '0';
  signal enwrite, enread: std_logic := '0';
  signal dinput: std_logic_vector(3 downto 0) := "1001";
  signal sfull, sempty: std_logic;
  signal sout: std_logic_vector(3 downto 0);
begin
  fifo_inst: entity work.fifo
  port map(
    clk => clock,
    rst_n => reset,
    enw => enwrite,
    enr => enread,
    din => dinput,
    dout => sout,
    empty => sempty,
    full => sfull
  );

  clock <= not clock after 5 ns;

  stim_proc: process
  begin
    wait until clock = '1';
    reset <= '1';
    wait until clock = '1';
    enwrite <= '1';
    for i in 0 to 3 loop
      wait until clock = '1';
    end loop;
    enwrite <= '0';
    enread <= '1';
    for i in 0 to 3 loop
      wait until clock = '1';
    end loop;
    enread <= '0';
    wait;
  end process;
end architecture;
```

У прикладі тестового оточення для початку виконується подача сигналу reset на вхід модулю, після чого подаються керуючі сигнали для перевірки роботи FIFO у наступній послідовності. Встановлюється сигнал дозволу на запис і відбувається запис чотирьох значень (у даному випадку подається однакове значення). Після запису значень сигнал дозволу на запис переводиться у пасивний стан, а сигнал дозволу на читання – в активний. На рис. 53 представлено процес моделювання роботи представленого модуля у тестовому середовищі.

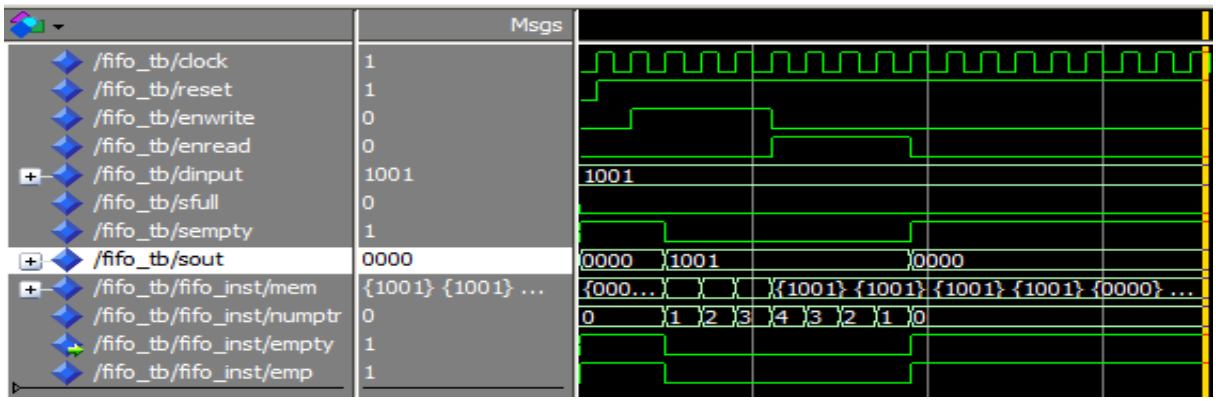


Рисунок 53 – Процес моделювання роботи представленого модуля у тестовому середовищі

Основним моментом, на який слід звернути увагу у даній демонстрації, є поведінка лічильника кількості записаних значень. У випадку виконання запису він збільшується, а при зчитуванні – зменшується. Саме в основі відстеження значення даного лічильника лежить формування вихідних сигналів статусу черги. Після того, як було зчитане останнє значення з черги, сигнал empty знову переходить в значення ‘1’.

Слід зауважити, що розробники засобів програмування для ПЛІС постачають у їх складі набір готових компонентів (IP-ядер), які можна використовувати у проєктах. Модуль FIFO є одним із таких модулів і тому, зазвичай, розробникам немає необхідності описувати функціональність черг самостійно, адже можна скористатись готовим рішенням. Такі компоненти, зазвичай, потребують налаштування основних параметрів, кількість яких може бути значною. Варто додати, що для синхронізації частотних доменів використовуються асинхронні модулі черг, які містять два входи для тактових частот із різних доменів.

7.3 Реалізація послідовних інтерфейсів. UART

UART (англ. Universal Asynchronous Receiver-Transmitter) – є одним із найпоширеніших інтерфейсів, які використовуються у вбудованих системах. Зазвичай, виробники залишають окрему опцію програмування мікросхеми за допомогою UART, оскільки підключення до даного інтерфейсу є доволі простим і дозволяє виконати програмування вже після збору пристрою. У процесорах для мінікомп’ютерів майже завжди є можливість підключення за допомогою UART і часто він стає першою опцією при роботі з ними. Навіть програмування і взаємодія з більшістю мережевих пристроїв можлива саме завдяки UART. Передача даних за допомогою UART здійснюється на основі двох провідників. Один із них працює на передачу, а інший – на прийом. Інтерфейс є асинхронним, тому лінія тактової частоти відсутня. Натомість, модулі, які взаємодіють між собою по даному інтерфейсу, використовують внутрішні модулі для забезпечення синхронізації.

Реалізація інтерфейсу UART на базі ПЛІС орієнтована, перш за все, на те, щоб забезпечити просту взаємодію цифрової схеми із зовнішніми пристроями. ПЛІС має реалізувати стандартний для UART протокол для взаємодії зі стороннім модулем:

1. Розпізнавання/видача початкового біту (лінія сигналу опускається у ‘0’ – рівень сигналу за замовчуванням – ‘1’).
2. Послідовна передача бітів даних.
3. Видача сигналу стоп-біт (лінія встановлюється у значення ‘1’).

Такий протокол є дещо спрощеним, оскільки не використовує перевірку на парність, але саме такий простий протокол використовується у більшості випадків. Також, зазвичай розділяють описи передавача та приймача на два окремі модулі для того, щоб забезпечити модульність системи (наприклад, коли необхідно лише приймати дані). Розглянемо описи модулів передавача та приймача на прикладі.

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

entity uart_rx is
  port(
    in_clk: in std_logic;
    rx: in std_logic;
    rx_data: out std_logic_vector(7 downto 0)
  );
end entity;

architecture behav of uart_rx is
  type UART_state is (idle, start_bit, data_bits, stop_bit, clean);
  signal cur_state: UART_state := idle;
  signal data_index: unsigned(3 downto 0) := "0000";
  signal rec_data: std_logic_vector(7 downto 0) := (others => '0');
  signal clk_count: unsigned(6 downto 0) := (others => '0');
  signal data_reg, buf_data_reg: std_logic := '0';
begin
  process(all)
  begin
    if in_clk = '1' and in_clk'event then
      buf_data_reg <= rx;
      data_reg <= buf_data_reg;
    end if;
  end process;

  process(all)
  begin
    if in_clk = '1' and in_clk'event then
      case cur_state is
        when idle =>
          data_index <= (others => '0');
          clk_count <= (others => '0');
          if data_reg = '0' then
            cur_state <= start_bit;
          else
            cur_state <= idle;
          end if;
        when start_bit =>
          if to_integer(clk_count) = 43 then
            if data_reg = '0' then
              clk_count <= (others => '0');
              cur_state <= data_bits;
            else
              cur_state <= idle;
            end if;
          else
            clk_count <= clk_count + 1;
            cur_state <= start_bit;
          end if;
        when data_bits =>
          if to_integer(clk_count) < 86 then -- 86
            clk_count <= clk_count + 1;
            cur_state <= data_bits;
          else
            clk_count <= (others => '0');
            rec_data(to_integer(data_index)) <= data_reg;
            if to_integer(data_index) < 7 then
              data_index <= data_index + 1;
              cur_state <= data_bits;
            else
              data_index <= (others => '0');
              cur_state <= stop_bit;
            end if;
          end if;
        when stop_bit =>
          if to_integer(clk_count) < 86 then
            clk_count <= clk_count + 1;
            cur_state <= stop_bit;
          else
            clk_count <= (others => '0');
            cur_state <= clean;
          end if;
        when clean =>
          cur_state <= idle;
        when others =>
          cur_state <= idle;
      end case;
    end if;
  end process;

  rx_data <= rec_data;
end architecture;
```

Приклад реалізує роботу UART у вигляді однопроцесного скінченного автомату: усі основні дії прописані у одному процесі. Описані 4 стани: стан очікування (idle), стан визначення стартового біту (start_bit), стан зчитування бітів даних із формуванням результуючого байту (data_bits), стан визначення стоп-біту (stop_bit). Для забезпечення внутрішньої синхронізації у даному випадку використовується лічильник. Даний лічильник проводить лічбу до визначеного значення (у випадку станів, які відповідають за прийом байтів). Тому швидкість передачі для даного UART має бути відомою заздалегідь. Налаштування швидкості передачі встановлюється значенням відношення кількості тактів внутрішньої тактової частоти до швидкості передачі UART:

$$clk\_to\_UART\_pulses = \frac{freq}{UART\_speed}.$$

У даному випадку цей показник буде розрахований наступним чином (робоча тактова частота – 10 МГц, швидкість передачі UART – 115200 біт/с):

$$clk\_to\_UART\_pulses = \frac{freq}{UART\_speed} = \left[\frac{10000000}{115200}\right] \approx 86$$

Дробова частина результату після коми була відкинута. Цього значення достатньо для того, щоб коректно проводити підрахунок для прийому даних з іншого пристрою. Проте, слід враховувати, що якщо відслідковувати лише значення по закінченні лічби, то виникають проблеми з надійністю, оскільки може виникнути ситуація, коли передавач вже змінив значення на лінії відповідно до наступного біту. Більш правильним рішенням є відслідковування значення всередині лічби. Саме тому при прийомі старт-біту лічба ведеться до половини цільового значення, після чого встановлюється у ‘0’. Після цього для наступних значень лічба ведеться до цільового значення, яке буде припадати на центральне значення. Деяка десинхронізація у роботі пристроїв не призведе до того, що будуть отримуватись некоректні дані.

Також розглянемо тест, який використовується для моделювання роботи представленого модулю. Він відрізняється від попередніх тестів, оскільки необхідно узгодити часові затримки для подачі сигналів на вхід. У даному випадку частота роботи схеми становить 10 МГц (що відповідає 50 нс для напівперіоду, який вказаний у тесті). Обчислений період, протягом якого сигнал на вході даних має бути стабільним – він становить 8680 нс. Дане значення, фактично, базується на кількості тактів, протягом якого необхідно забезпечити стабільність сигналу на вході. Подача даних на вхід оформлена у вигляді процедури, у яку необхідно передавати вектор даних. Така процедура спрощує передачу даних на модуль UART у тесті до одиночного рядка.

Розглянемо результати моделювання для розробленого модуля та тестового оточення. Для цього у вікно **Wave** були додані сигнали не лише з тестового оточення, а і окремі внутрішні сигнали модуля UART (рис. 54).

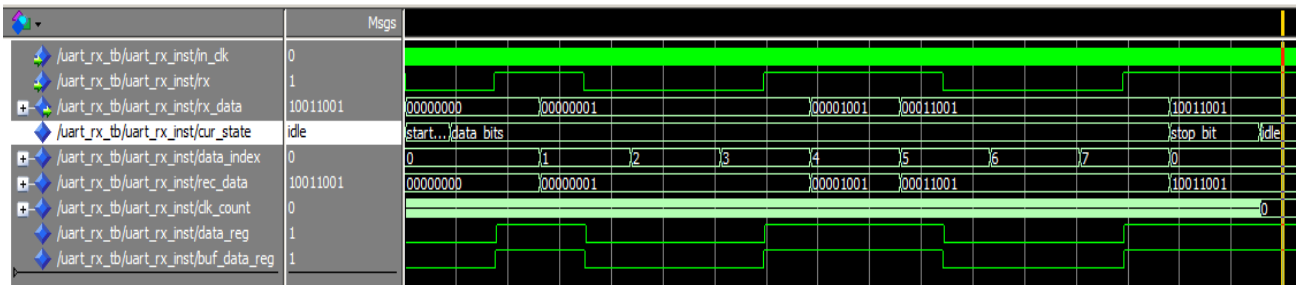


Рисунок 54 – Результати моделювання для модуля UART

На рис. 54 масштаб значно зменшено для того, щоб продемонструвати прохід через усі стани скінченного автомату. Також, порівнюючи частоту вхідних тактових імпульсів та те, як передаються дані по UART, можна бачити, наскільки великим є запас по обробці даних, що важливо за умови необхідності виконання складних операцій над отриманими даними.

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

entity uart_rx_tb is
end entity;

architecture tb of uart_rx_tb is
  signal clk: std_logic := '0';
  signal srx: std_logic := '1';
  signal received_data: std_logic_vector(7 downto 0) := (others => '0');
  constant HOLD_PERIOD: time := 8680 ns;

  procedure uart_write_rx(data: in std_logic_vector(7 downto 0);
    signal serdata: out std_logic) is
  begin
    serdata <= '0';
    wait for HOLD_PERIOD;

    for i in 0 to 7 loop
      serdata <= data(i);
      wait for HOLD_PERIOD;
    end loop;

    serdata <= '1';
    wait for HOLD_PERIOD;
  end procedure;

begin
  clk <= not clk after 50 ns;

  uart_rx_inst: entity work.uart_rx port map (
    in_clk => clk,
    rx => srx,
    rx_data => received_data
  );

  stim_proc: process
  begin
    wait for 10 ns;
    uart_write_rx("10011001", srx);
    wait;
  end process;
end architecture;
```

На рис. 55 більш детально представлено момент переходу з одного стану автомату в інший.

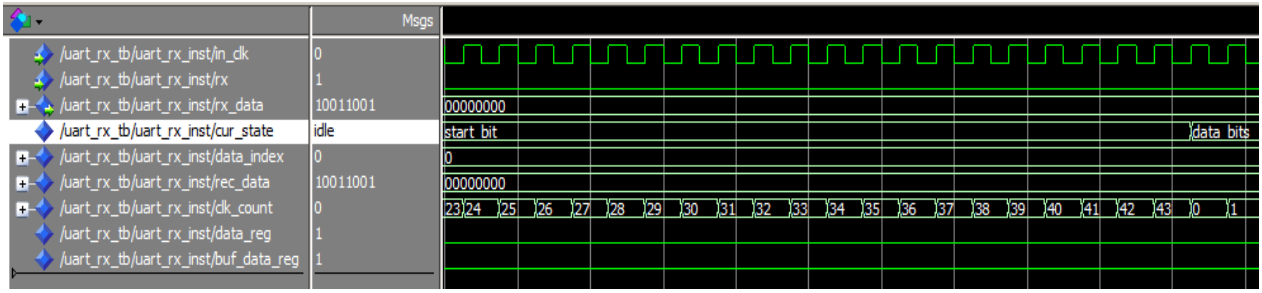


Рисунок 55 – Момент переходу з одного стану автомату в інший

На рис. 56 видно, як значення лічильника досягає половини обчисленого цільового значення, що означає центр старт-біту, і починає відраховувати значення з ‘0’ для прийому першого біту (автомат знаходиться у стані **data_bits**).

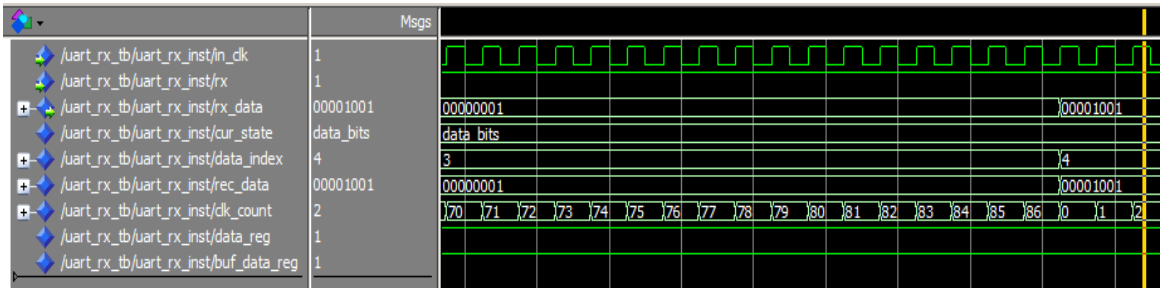


Рисунок 56 – Знаходження автомату у стані **data_bits**

7.4 Поняття покриття

Покриття (англ. coverage) використовується для позначення того, наскільки код, що використовується у проєкті, охоплений викликами від сторонніх модулів (зазвичай, це тестові оточення). Покриття дозволяє більш детально дослідити якість тестового оточення та підтвердити це відповідними метриками. Використання функціональності для організації покриття дозволяє вдосконалити набір інструментів розробника цифрових схем мовою VHDL. Слід зауважити, що дана функціональність середовища доступна лише у просунутих версіях, тому наводиться для більш повного ознайомлення з можливостями середовища та надання розуміння повного циклу розробки.

Функціональність для дослідження покриття включена безпосередньо у середовище ModelSim. За замовчуванням вона не є увімкненою, тому її необхідно активувати. Причиною того, що дана функціональність недоступна за замовчуванням є невелика (5–10 %) втрата у продуктивності середовища, яка пов’язана з виконанням додаткових перевірок. Проте, переваги від використання функціональності покриття перевершують незначне зменшення продуктивності.

> **Для програмістів!** *Аналогічні засоби, зазвичай, є вбудованими у більшість середовищ розробки (або доволі легко встановлюються у вигляді плагінів або сторонніх засобів). Дані засоби дозволяють, так само як і даному випадку, оцінити якість тестових сценаріїв та охоплення коду тестами.*

Для того, щоб активувати дослідження покриття у проєкті, необхідно виконати декілька налаштувань. По-перше, у властивостях файлів (команда контекстного меню **Properties**, вкладка **Coverage**), які необхідно перевіряти, слід вказати, що для них опція покриття є активною (рис. 57).

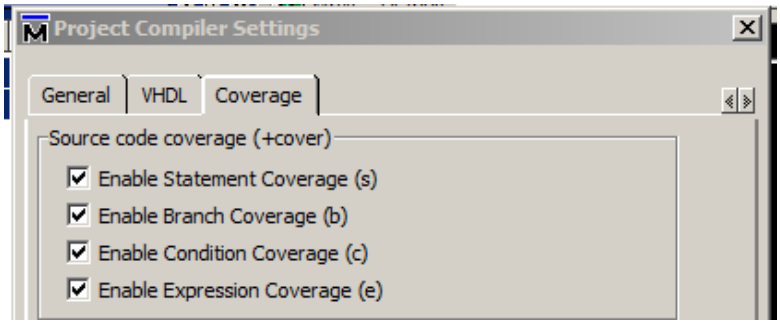


Рисунок 57 – Активація дослідження покриття у проєкті

Після цього знадобиться виконати повторну компіляцію обраних файлів. Наступним кроком є активація покриття під час моделювання. Для цього під час її запуску у вікні **Start Simulation** на вкладці **Others** необхідно увімкнути опцію **Enable code coverage** (рис. 58).

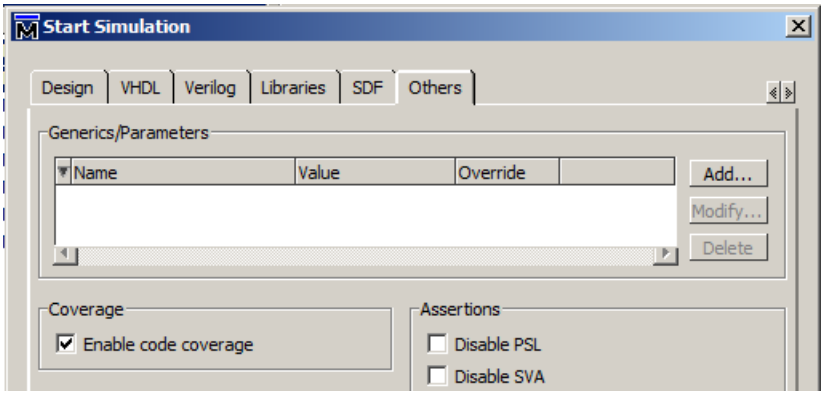


Рисунок 58 – Активація покриття під час моделювання

Після того, як буде запущено процес модулювання, зовнішній вигляд середовища під час цього процесу має змінитись: з'являються вікна, які відповідальні за відображення інформації про покриття. З'являються вікна Files, Instance та Code Coverage Analysis (із додатковими внутрішніми вікнами), а також має змінитись відображення вікна sim (додана інформація про покриття). У випадку, якщо цього не відбулось, то слід додати вказані вікна за допомогою команд меню View.

У вікні Files після виконаних налаштувань буде доступна статистика виконання виразу, гілки та ін. (рис. 59). Для того, щоб отримати статистку, необхідно запустити процес моделювання. Статистика ведеться лише для цільових файлів, для яких була задана опція аналізу покриття (у даному випадку використовуються файли з попереднього прикладу, який є доволі простим).

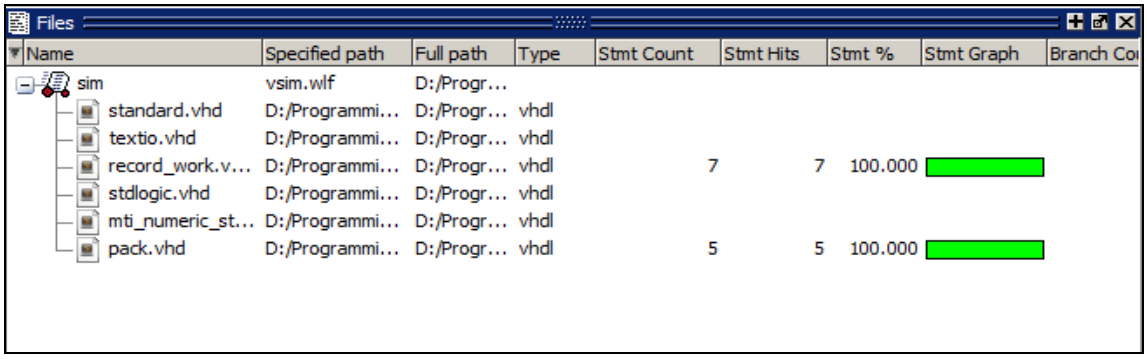


Рисунок 59 – Статистика для цільових файлів

У вікні **Code Coverage Analysis** буде доступно одразу декілька вкладок. На вкладці **Coverage** виводяться вирази, які стосуються окремих модулів, а також статус їх виконання (рис. 60). Даний інструмент зручний для того, щоб визначати, чи є ділянки коду, які ніколи не виконуються.

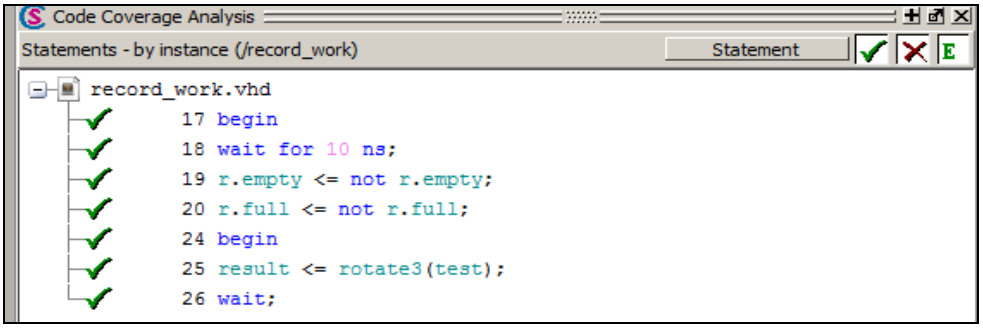


Рисунок 60 – Статус виконання модулів

Якщо деякі ділянки коду не мають проходити перевірку, то їх можна виключити зі списку виразів, за якими ведеться спостереження.

Інші вікна також є важливими, однак, для їх використання необхідно виконати більше налаштувань і у даному розділі вони не розглядаються.

7.5 Мова Property Specification Language (PSL)

У попередніх розділах представлено, як проводити перевірку операторами VHDL та засобами ModelSim. Тим не менш, згаданими засобами список не вичерпується. У версії 2008 мови VHDL було додано особливий механізм проведення перевірки, який дозволяє не додавати спеціальних операторів безпосередньо у функціональну частину, що значно спрощує використання коду як для тестування, так і для реалізації схеми, оскільки немає необхідності видаляти зайві оператори. Такі можливості надає Property Specification Language (PSL) – мова опису властивостей, яка дозволяє відслідковувати поведінку сигналів та визначати, наскільки результат відповідає очікуваній функціональності. PSL є частиною стандарту VHDL-2008, хоча використовує відмінний від VHDL синтаксис.

Розглянемо, як виглядатиме перевірка одного з найпростіших модулів – регістра – з використанням інструкцій PSL.

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

entity reg is
  port(
    clk: in std_logic;
    rst_n: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end entity;

architecture behav of reg is
  signal ds: std_logic := '0';
-- psl default clock is rising_edge(clk);
-- psl property check_val is always {d} |=> {q};
begin
  process(all)
  begin
    if rst_n = '0' then
      ds <= '0';
    elsif rising_edge(clk) then
      ds <= d;
    end if;
  end process;

  q <= ds;
  -- psl assert check_val;
end architecture;
```

Як видно з рисунку, інструкції PSL додаються у вигляді коментарів VHDL, проте вони розпізнаються середовищем і при збереженні документу отримують виділення кольором ключових слів та інших літералів. Слід зауважити, що відповідно до стандарту PSL, якщо інструкції PSL знаходяться лише в деклараційній частині архітектури, то вони не потребуються виділення у коментар. Для того, щоб проводити перевірку, у даному випадку описана властивість (**property**). Перевірку властивості забезпечує оператор **assert**. У модулі вказується джерело тактового сигналу. За зміни тактового сигналу відбувається перевірка умов. Власне, сама властивість описується з використанням ключового слова **always**, а також оператора, який вказує залежність між вхідним та вихідним сигналом.

Після запуску моделювання модуля у вікні **Assertions** має бути доступний пункт, який відповідає описаному випадку. Його можна додати у вікно **Wave** разом з іншими сигналами для більш зручного відслідковування.

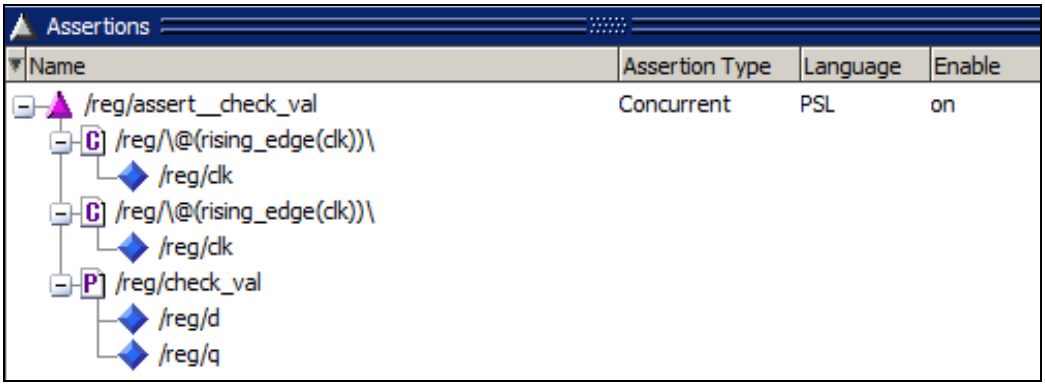


Рисунок 61– Інструкції PSL в деклараційній частині архітектури

Розглянемо, що відбувається у вікні **Wave** під час моделювання (рис. 62).

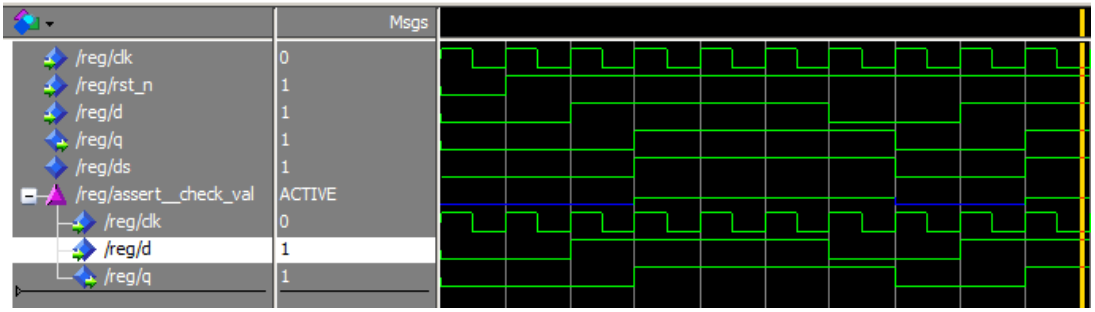


Рисунок 62 – Вікно **Wave** під час моделювання

У даному випадку видно, що властивість є активною лише у тому випадку, коли вихід знаходиться у логічному стані ‘1’.

7.6 Профілювання засобами ModelSim

Профілювання дозволяє визначити використання ресурсів під час роботи симулятора, а, у результаті, оптимізувати їх використання на основі проведеного аналізу. Слід зауважити, що дана опція доступна лише для професійних версій, а не для початкових, тому даний матеріал призначений для проведення теоретичного ознайомлення з можливостями старших версій середовища ModelSim.

Для того, щоб запустити профілювання, необхідно для початку виконати початок моделювання, після чого ввести команду **profile on**. Лише після цього можна запустити процес моделювання. Середовище збиратиме інформацію про те, виконання яких саме ділянок коду займає найбільше часу (рис. 63). Для того, щоб отримати доступ до вікон, у яких відображається статистика (рис. 64), слід увімкнути необхідні пункти у меню **View** (якщо вони не з’явились після введення команди запуску профілювання).

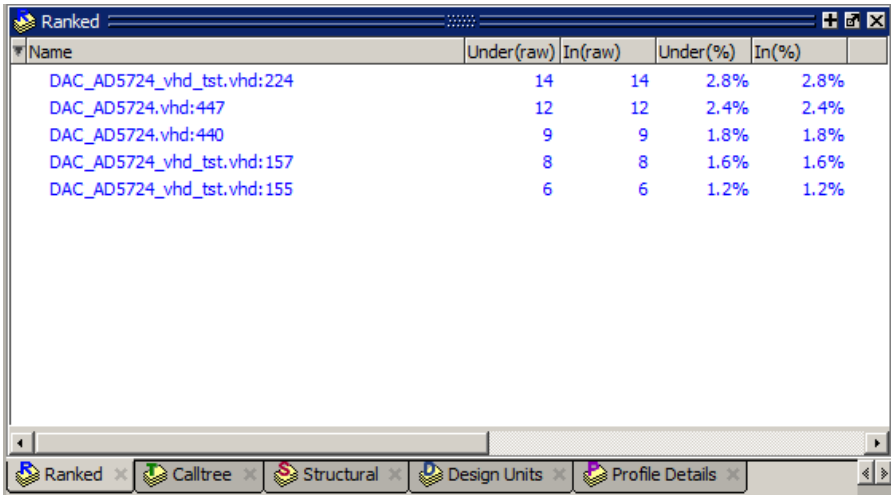
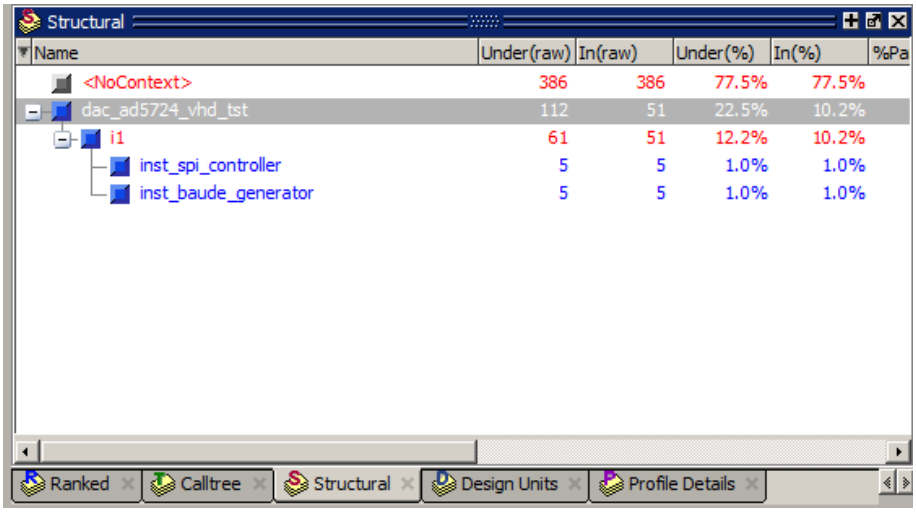
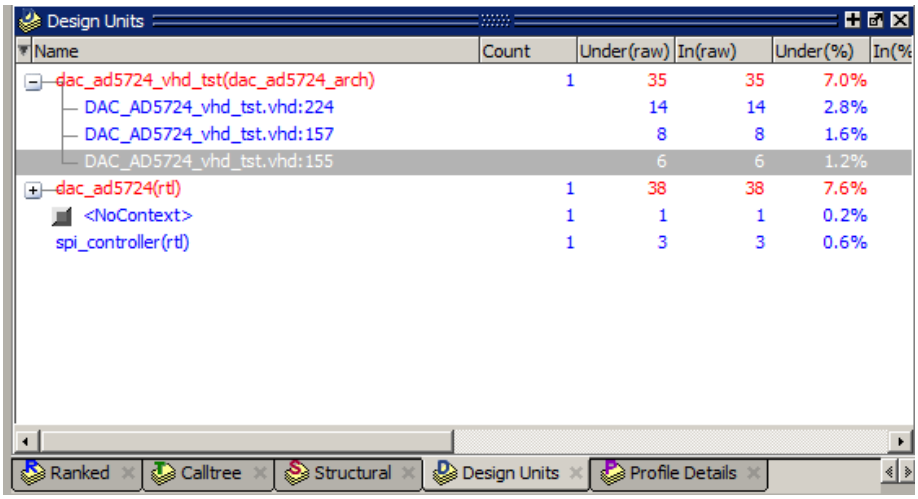


Рисунок 63 – Час виконання ділянок коду



а)



б)

Рисунок 64 – Статистика виконання ділянок коду

7.7 Поняття IP-ядер

Велика кількість компонентів є доступною для використання безпосередньо із середовищ розробки, які розповсюджуються виробниками. Це пов'язано з тим, що певні компоненти є типовими при розробці цифрових схем і їх поведінка є зрозумілою для розробників цифрових схем. Їх правильне використання дозволяє значно зменшити час, необхідний на отримання готового рішення. Такі компоненти достатньо налаштувати і додати інструкції по створенню екземплярів таких схем у проєкт. Параметри, які передаються для налаштування IP-ядер, зазвичай є узагальненими значеннями (**generics**). Іншою назвою таких блоків є IP-ядра (англ. Intellectual Property Core, IP core). Вихідний код IP-ядер може бути як відкритим, так і закритим (немає відповідного файлу мовою схемотехнічного опису, який можна переглянути за допомогою текстового редактору). Ситуація, коли розробник бажає приховати код компонента, є доволі типовою, і саме тому, зазвичай, складні компоненти є платними. Причому, оплачується як безпосередньо можливість використовувати компонент, так і окремо доступ до вихідних файлів компонента.

У середовищах розробки для налаштувань IP-ядер використовуються спеціальні майстри (**wizards**) для того, щоб повністю встановити необхідні параметри. Кількість доступних параметрів є значною навіть для такого компонента, як FIFO. Тому типовим є те, що такі майстри складаються з кількох сторінок налаштувань, де необхідно вводити параметри роботи компонента.

Наявність спеціального майстра характерна для компонентів, які постачаються разом із середовищем і які потребують оплати за їх використання, проте, доволі поширеною є також практика використання IP-ядер з відкритим вихідним кодом. Перевагою таких ядер є, по-перше, їх більша універсальність, оскільки вони, зазвичай, не призначені для використання в рамках одного середовища, а, по-друге, наявність коду, у який можна вносити зміни (слід враховувати особливості ліцензії, за якою поширюються компоненти при їх застосуванні у комерційних проєктах). Одним із найбільш відомих ресурсів, який надає доступ до відкритих IP-ядер, є *opencores.org*. Також велику кількість готових компонентів можна знайти у репозиторіях, розміщених на GitHub.

Контрольні питання

1. Оператор зсуву sll (зсув вліво) може бути оформлений як пакет?
2. Тип bit є перерахованим типом?
3. Тип boolean є перерахованим типом?
4. Дельта-затримка дорівнює одній пікосекунді модельного часу?
5. Дельта-затримка дорівнює одній фемтосекунді модельного часу?
6. Розмір дельта-затримки визначається користувачем?
7. Дельта-затримка дорівнює нулю модельного часу?
8. Маючи на увазі дельта-затримки, можна сказати, що затримка виконання оператора a <= b; становить дві дельта-затримки?
9. У розділі декларацій архітектурного тіла має бути лише одна функція?
10. У розділі декларацій архітектурного тіла має бути лише одна процедура?
11. У пакеті має бути лише дві функції?
12. У тілі пакета може бути понад 100 процедур?
13. У позиційній відповідності портів (оператор port map) сигнали розташовуються у тому ж порядку, як порти в entity?
14. Компоненти, декларовані усередині архітектури, специфікуються повністю, тобто з їх інтерфейсом та архітектурою?
15. У операторі port map символи => чи <= (відповідності) використовуються залежно від напрямку порту (для входу символи =>, для виходу символи <=)?
16. Чи можуть бути вкладені оператори block?
17. При виклику процедури в архітектурному тілі мітка є обов'язковою?
18. Мітка процесу в архітектурному тілі є обов’язковою?
19. Усі конструкції VHDL-описи цифрової системи реалізуються автоматично при синтезі логічними елементами чи логічними підсхемами?
20. Оператор складання цілих чисел реалізується під час синтезу логічної схемою?
21. Оператор складання дійсних чисел може бути реалізований під час синтезу логічною схемою?
22. У тестуючій (головній VHDL-програмі) є описи вхідних та вихідних портів?
23. У тестуючій програмі можуть тестуватися описи незв’язаних між собою цифрових систем?
24. Вираз, що перевіряється, clk='1' в операторі if (clk='1') має тип boolean?
25. Якщо сигнал clk має тип bit, то вираз (clk'event and clk='1') також має тип bit?
26. Вираз (clk'event and clk='1') відповідає задньому фронту сигналу clk?

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mealy B., Tappero F. Free range VHDL. 2023. 244 p. URL: https://github.com/fabriziotappero/Free-Range-VHDL-book (Last accessed: 19.09.2024).
2. Krainyk Y., Darnapuk Y. Configurable description of FPGA-based control system for sensor processing. *Proc. of the 2019 11th International Scientific and Practical Conference on Electronics and Information Technologies (ELIT 2019), Lviv, Ukraine,* 16–18 Sept. 2019. P. 210–213. DOI: 10.1109/ELIT.2019.8892313.
3. Bailey D. G. Design for embedded image processing on FPGAs. John Wiley & Sons (Asia) Pte Ltd, 2011. 482 p. DOI: 10.1002/9780470828519.
4. Cohen B. VHDL Coding Styles and Methodologies, 1999. 474 p.
5. Ashenden P. J. The designer's guide to VHDL. Morgan kaufmann, 2010. 909 p. DOI: 10.1016/B978-0-12-088785-9.X0001-9.
6. Pedroni V. A. Circuit design with VHDL. MIT press, 2020. 608 p.
7. ModelSim SE Tutorial. URL: http://pages.cs.wisc.edu/~markhill/cs552/Fall2006/handouts/se_tutor.pdf (Last accessed: 19.09.2024).
8. VHDL Register based FIFO. URL: https://www.nandland.com/vhdl/modules/module-fifo-regs-with-flags.html (Last accessed: 19.09.2024).
9. Crossing Clock Domains. URL: https://www.fpga4fun.com/CrossClockDomain1.html (Last accessed: 19.09.2024).
10. Home: OpenCores. URL: https://opencores.org/ (Last accessed: 19.09.2024).

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчальне видання

Ярослав КРАЙНИК

РОЗРОБКА ЦИФРОВИХ СХЕМ МОВОЮ VHDL
Основи розробки цифрових схем

Навчальний посібник
з дисципліни «Вбудовані системи»

Редактор *О. Михайлова*
Комп'ютерна верстка, дизайн обкладинки *К. Гросу-Грабарчук*
Друк *С. Волинець*. Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 04.03.2025.
Формат 60×84$^{1}/_{8}$. Папір офсет.
Гарнітура «Calibri». Друк ризограф.
Ум. друк. арк. 6,9. Обл.-вид. арк. 3,3.
Тираж 100 пр. Зам. № 6953.

Видавець і виготовлювач: ЧНУ ім. Петра Могили
вул. 68 Десантників, 10, м. Миколаїв, 54009
Тел.: +38 (0512) 50–03–32, +38 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2020.