

All rights not reserved.

Кондратенко Ю.П., Сидоренко С.А.,

Підопригора Д.М.

**ПОВЕДІНКОВИЙ СИНТЕЗ ЦИФРОВИХ
ПРИСТРОЇВ
В СЕРЕДОВИЩІ ACTIVE-HDL**

Кабінет Міністрів України
Національний університет "Кієво-Могилянська Академія"
Миколаївська філія

Кондратенко Ю.П., Сидоренко С.А.,
Підпригора Д.М.

ПОВЕДІНКОВИЙ СИНТЕЗ ЦИФРОВИХ ПРИСТРОЇВ В СЕРЕДОВИЩІ ACTIVE-HDL

Під редакцією
доктора технічних наук, професора
Ю.П. Кондратенка

*Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих закладів освіти,
що навчаються за напрямками
“Комп’ютерні науки”,
“Комп’ютеризовані системи управління і автоматики”
та “Електронні системи”*

Видавництво МФ НаУКМА

Миколаїв, 2001

УДК 621.374

Кондратенко Ю.П., Сидоренко С.А., Підпригора Д.М.

**ПОВЕДІНКОВИЙ СИНТЕЗ ЦИФРОВИХ ПРИСТРОЇВ В СЕРЕДОВИЩІ
ACTIVE-HDL.** Навчальний посібник. Під ред. Ю.П.Кондратенка. –
Миколаїв, МФ НаУКМА, 2001. – с.136

Навчальний посібник містить опис мови VHDL, теоретичні відомості та методичні вказівки з циклу лабораторних робіт для розробки поведінкових моделей цифрових пристроїв різного призначення. Поведінковий синтез базується на мові опису апаратного забезпечення VHDL з використанням системи автоматизованого проектування Active-HDL корпорації Aldec Inc. Цикл лабораторних робіт дозволяє засвоїти методику формування моделей цифрових пристроїв із застосуванням VHDL та теорії скінчених автоматів.

Навчальний посібник призначається для студентів, дипломників та магістрантів вищих закладів освіти, що навчаються за напрямками “Комп’ютерні науки”, “Комп’ютеризовані системи управління і автоматики”, “Електронні системи”.

Рецензенти:

Власенко В.О., доктор технічних наук, професор, завідувач кафедри інформаційних систем Одеського національного політехнічного університету

Гераїмчук М.Д., доктор технічних наук, професор, завідувач кафедри приладобудування Національного технічного університету "Київський політехнічний інститут"

Рябенський В.М., доктор технічних наук, професор, завідувач кафедри теоретичної електротехніки та електронних систем Українського державного морського технічного університету імені адмірала Макарова

© Миколаївська філія Національного
університету “Києво-Могилянська
академія”, 2001

ПЕРЕДМОВА

Цей навчальний посібник написано у відповідності зі спеціальними розділами, що стосуються засобів автоматизованого моделювання та проектування цифрових пристроїв, згідно навчальної програми з курсу "Програмні засоби інтелектуальних систем" в Миколаївській філії Національного університету "Києво-Могилянська академія" (напрямок – Комп'ютерні науки, спеціальність – Інтелектуальні системи прийняття рішень) та згідно навчальної програми з курсу "Автоматизоване проектування цифрових пристроїв" в Українському державному морському технічному університеті (напрямок – Комп'ютеризовані системи, управління і автоматика, спеціальність – Системи управління і автоматики).

Посібник складається з шести розділів:

- перший розділ містить загальну інформацію про характеристики програмованої логіки, в ньому розглядаються різні види технологій побудови програмованих логічних інтегральних схем (ПЛІС), області застосування та основні виробники ПЛІС;
- другий розділ знайомить студентів з алфавітом алгоритмічної мови опису електронного обладнання та апаратного забезпечення VHDL, зокрема, з правилами формування ідентифікаторів, символами, числами, рядками та ін.,
- в третьому розділі розглядається структура мови VHDL, пояснюються особливості формування умовних операторів та операторів вибору, циклічних операторів, а також особливості перетворення типів в VHDL та формування масивів даних;
- засобам VHDL для моделювання реальних об'єктів присвячено четвертий розділ, де вводяться поняття (в VHDL) об'єкту, сигналу, процесу, архітектури, описуються атрибути сигналів, спеціальні

оператори, способи реалізації затримки в VHDL, особлива увага приділяється поведінковому синтезу цифрових пристроїв на основі моделювання в VHDL, декомпозиції складних об'єктів і структурним складовим проектів, паралельній реалізації процесів та перекриттю сигналів;

- п'ятий розділ містить відомості про додаткові можливості САПР Active-HDL, що пов'язані з використанням діаграм скінчених автоматів в VHDL, віртуальними випробувальними стендами, особливостями відображення інформації;
- в шостому розділі наведено опис 8-ми лабораторних робіт на основі використання Active-HDL, виконання яких дозволить студентам більш глибоко засвоїти як безпосередньо мову VHDL, так і методику поведінкового синтезу цифрових пристроїв в середовищі Active-HDL.

Крім того, навчальний посібник містить короткий англо-український словник спеціальних термінів, якими найбільш широко користуються фахівці в області застосування мови апаратного забезпечення VHDL.

Засвоєння матеріалу навчального посібника дозволить студентам правильно синтезувати цифрові пристрої на основі останніх світових досягнень в області автоматизованого проектування пристроїв електронної техніки з відповідними етапами програмування, моделювання та віртуального тестування.

Навчальний посібник також доцільно використовувати для організації самостійної підготовки студентів, оскільки в ньому наведено значну кількість прикладів для підвищення практичного засвоєння матеріалу, а також контрольних запитань - для самоперевірки рівня знань.

Основний матеріал навчального посібника пройшов широку апробацію, оскільки з початку 2000 року він в режимі вільного доступу розміщений на українському сайті корпорації ALDEC Inc. - розробника середовища Active-HDL : www.aldec.com.ua

Автори висловлюють щирю вдячність керівництву корпорації Aldec Inc.(США), особисто її президенту доктору Стенлі М. Хайдуку, а також доктору технічних наук, професору Ю.С. Канєвському (Національний технічний університет України "Київський політехнічний університет") та доктору технічних наук, професору В.В. Мохору (Інститут проблем моделювання в енергетиці Національної академії наук України) за надану можливість роботи з САПР Active-HDL (ліцензовані версії Active-HDL 3.21, Active-HDL 3.5, Active-HDL 3.6, Active-HDL 4.1, Active-HDL 4.2).

Автори дякують доктору технічних наук, професору В.О.Власенку (Одеський національний політехнічний університет), доктору технічних наук, професору М.Д.Гераїмчуку (Національний технічний університет України "Київський політехнічний університет") та доктору технічних наук, професору В.М.Рябенському (Український державний морський технічний університет) за уважне рецензування рукопису та рекомендації щодо поліпшення окремих його розділів, які було враховано при остаточному доопрацюванні рукопису навчального посібника.

ВСТУП

Стрімкий розвиток цифрової техніки та широке її впровадження майже у всі сфери життєдіяльності людини обумовили виникнення потреб в значній інтенсифікації процесів проектування електронних пристроїв, що, в свою чергу, спричинило до активного пошуку нових, більш ефективних підходів до розробки засобів проектування, зокрема до розробки систем автоматизованого проектування (САПР) і моделювання. Одним з перспективних шляхів цього напрямку є синтез цифрових пристроїв на основі функціональних моделей, суть якого полягає в автоматизованому формуванні логічної структури пристрою на основі його бажаної поведінки, що описується фахівцем-розробником за допомогою спеціальної мови програмування – **HDL** (Hardware Description Language – мова опису обладнання та апаратного забезпечення). Найбільш відомими серед мов класу **HDL** стали **ABEL**, **Verilog** та **VHDL**.

ABEL (Advanced Boolean Equation Language – розширена мова логічних рівнянь) є промисловим стандартом, що розроблений Data I/O Corp. для програмованих логічних пристроїв. **ABEL** може застосовуватися для опису поведінки систем (за допомогою **C**-подібних операторів) в різних формах на основі: логічних рівнянь, таблиць істинності, діаграм стану.

В порівнянні з **ABEL** мови **VHDL** та **Verilog** є більш складними і, відповідно, більш потужними та більш придатними для опису великих систем. Практично вони подібні за своїми технічними можливостями. В Європі та в Сполучених Штатах Америки ширше застосовується **VHDL**, а в країнах Азії – **Verilog**. При розробці **VHDL** за основу було взято відому мову високого рівня **Ada**, яку фахівці відносять до найбільш потужних та продумано спроектованих. З мови **Ada** розробниками **VHDL** було запозичено синтаксис та основні структури.

Справжнього поширення мови **HDL** набули в результаті розвитку відповідної апаратної бази – інтегральних мікросхем (ІМС) із змінною внутрішньою логічною структурою. Цей клас ІМС називається програмованими логічними інтегральними схемами (ПЛІС).

У відповідності до сучасних вимог системи автоматизованого проектування ПЛІС повинні забезпечувати:

- реалізацію однієї або більше **HDL**-мов з можливістю введення, редагування та відлагодження вихідного тексту програм;
- реалізацію засобів графічного введення проектної схеми, наприклад за допомогою редактора скінчених автоматів, та засобів компіляції графічного представлення в **HDL**-код;
- реалізацію засобів моделювання поведінки описаного об'єкта;
- реалізацію засобів синтезу бітового потоку з підтримкою широкого класу серій ІМС;
- реалізацію засобів моделювання об'єкта на рівні вентилів;
- реалізацію засобів програмування ІМС.

В наш час розроблено ряд програмних продуктів, що реалізують ту чи іншу частину загальних вимог до САПР ПЛІС. Застосування цих програмних продуктів вимагає в кожному конкретному випадку розв'язання проблем сумісності між відповідними пакетами. Тому при виборі засобів автоматизованого проектування перевагу слід надавати програмним пакетам, що реалізують процес проектування ПЛІС якомога повніше. Одним з таких пакетів є **Active-HDL** корпорації Aldec Inc. (США).

Корпорація Aldec Inc. є одним з лідерів у розробці програмного забезпечення САПР ПЛІС. Завдяки орієнтованій на співпрацю політиці керівництва компанії по наданню науково-технічної підтримки провідним українським університетам, що готують фахівців за комп'ютерними напрямками, викладачі, науковці та студенти комп'ютерних та

електротехнічних факультетів Миколаївської філії Національного університету "Києво-Могилянська академія", Українського державного морського технічного університету, Одеського національного політехнічного університету та ін. мають можливість розробляти проектні схеми та схемотехнічні рішення на основі ПЛІС за допомогою повної ліцензійної версії САПР **Active-HDL**.

1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА ПРОГРАМОВАНОЇ ЛОГІКИ

1.1 Області застосування та основні виробники ПЛІС

Ще на початку 90-х років ПЛІС мали дуже вузьке коло користувачів, до якого в основному входили підприємства, що займаються розробкою та випуском замовних мікросхем (звичайних ІМС з постійною логічною структурою). За останнє десятиліття область застосування програмованих логічних ІМС значно розширилась.

Діаграма розподілу обсягів споживання ПЛІС в окремих галузях народного господарства наведена на рис. 1.1.

Найбільшим споживачем ПЛІС є галузь телекомунікацій і зв'язку (39% від всього обсягу виробництва). На другому місці йде галузь комп'ютерних мереж, що використовує 26% обсягу виробництва ПЛІС. Крім того, програмовані логічні ІМС застосовуються в області цифрової обробки даних (19%) та в промисловому виробництві (16%). Серед споживачів цих мікросхем можна назвати такі відомі фірми і концерни, як Alcatel, IBM, Boeing, Lockheed, Hewlett Packard, Fujitsu, Hitachi, Silicon Graphics, Texas Instruments, Motorola, Rockwell, Kodak та багато інших.

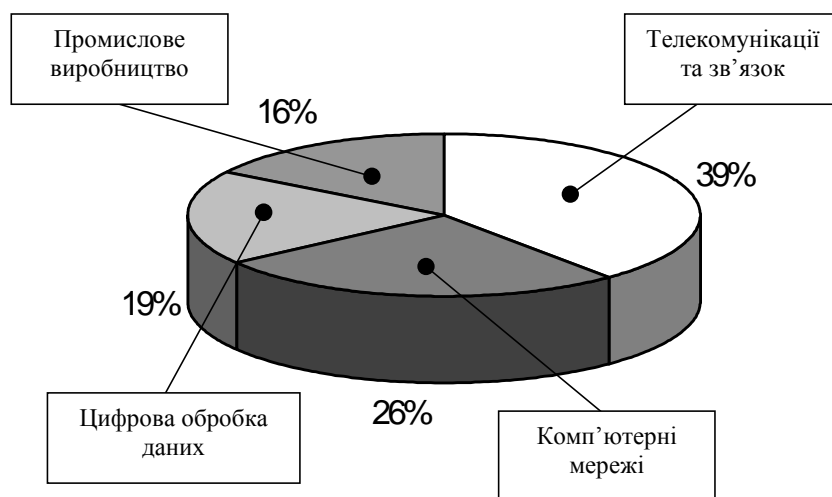


Рис. 1.1 – Обсяги застосування ПЛІС в окремих галузях
народного господарства

Проектуванням та виробництвом ПЛІС на даний момент часу займаються декілька десятків провідних фірм та компаній.

Лідером в цій області є фірма Xilinx (обсяг виробництва тільки у 1-му півріччі 1999 р. склав 211,4 млн. USD). Фірма Xilinx спеціалізується на виробництві ІМС високої якості, зокрема стійких до впливу радіації та інших зовнішніх збурень, що обумовлює широке застосування ПЛІС в аерокосмічній галузі, військовій техніці та в промисловому виробництві.

Програмовані логічні ІМС фірми Altera (197.8 млн. USD) займають нішу масових ПЛІС невисокої вартості.

Фірма Lattice (59.7 млн. USD) в основному розробляє перепрограмовані мікросхеми CPLD і утримує лідируючі позиції в цій галузі.

До основних виробників ПЛІС слід також віднести фірму Actel (41.6 млн. USD).

Серед номенклатури ПЛІС найбільшого поширення набули мікросхеми, що виготовлені за технологіями FPGA та CPLD. Розглянемо ці технології більш детально.

1.2 FPGA – технологія

Абревіатура FPGA означає Field Programmable Gate Array – програмована користувачем вентильна матриця.

Загальну структуру кристала FPGA-мікросхеми наведено на рис. 1.2. По периферії верхнього шару кристала розміщуються блоки вводу/виводу (БВВ), що можуть бути запрограмованими для виконання функцій

буферів: вхідного, вихідного, з запам'ятовуванням стану та ін. В деяких серіях FPGA-ІМС рівень напруги на двох БВВ може відрізнатись, що дає змогу поєднувати різні за рівнем живлення інтерфейси.

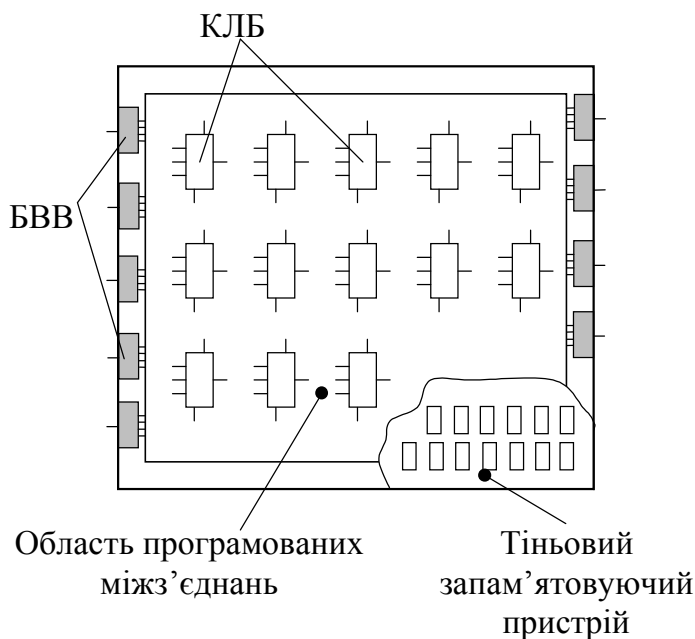


Рис. 1.2 – Узагальнена структура FPGA-мікросхеми

В центрі кристала у вигляді матриці розміщено конфігуровані логічні блоки (КЛБ). Швидкодія мікросхем визначається часовою затримкою “вхід-вихід” одного КЛБ. Структура КЛБ залежить від серії мікросхем, так, наприклад, в ІМС XC2000 кожен КЛБ має один елемент пам’яті (тригер), 2 виходи, 4 входи загального призначення та спеціальний вхід синхронізації (тактовий вхід). КЛБ ІМС цієї серії може генерувати будь-яку логічну функцію чотирьох змінних або дві логічні функції трьох змінних. Змінні для логічних функцій можуть надходити з чотирьох входів та виходу елементу пам’яті.

Область між конфігурованими логічними блоками називається областю програмованих між'єднань і являє собою розвинену ієрархію металічних ліній зв'язку, в місцях перетину яких розміщено спеціальні

швидкодіючі транзистори. Функція області міжз'єднань полягає в забезпеченні зв'язку між будь-якими виводами КЛБ та БВВ. Необхідний маршрут міжблокових з'єднань в FPGA-ПЛІС реалізується комутацією відповідних ліній за допомогою транзисторів.

Нижній шар кристалу займає тінювий запам'ятовуючий пристрій, інформація в елементах якого і визначає логічні функції КЛБ, конфігурацію БВВ та маршрути міжз'єднань.

В табл. 1.1 наведено основні характеристики мікросхем FPGA виробництва Xilinx Inc., зокрема, однієї з останніх розробок серії Virtex та масової серії Spartan.

Широкий діапазон ІМС FPGA-технології дозволяє проектувати на їх основі широкий спектр електронних пристроїв, серед яких: засоби поєднання різних за живленням інтерфейсів, перетворювачі кодів, периферійні контролери, мікро-програмні пристрої керування, скінчені автомати, універсальні та спеціалізовані процесори, пристрої цифрової обробки сигналів тощо.

Табл. 1.1

Характеристика	Серія Virtex			Серія Spartan	
	XCV50	XCV1000	XCS40/XL	XCS20/XL	XCS05/XL
Системна частота (MHz)	200	200	80	80	80
Технологія (мкм)	0.22	0.22	0.35/0.5	0.35/0.5	0.35/0.5
Напруга живлення ядра (В)	2.5	2.5	3.3/5.5	3.3/5.5	3.3/5.5
Кількість системних вентилів (шт)	57906	1124022	13К-40К	7К-20К	2К-5К
Кількість логічних комірок (шт)	1728	27648	1862	950	238
Макс. число портів вводу виводу (шт)	180	514	205	160	77

1.3 CPLD – технологія

Архітектура ІМС типу CPLD (Complex Programmable Logic Device – комплексний програмований логічний пристрій) представлена на рис. 1.3 (для прикладу взято архітектуру популярної серії мікросхем XC9500).

ІМС XC9500 має три групи виводів:

- 1) виводи JTAG-порту (стандарт IEEE Std. 1149.1) для програмування та периферійного сканування ІМС;
- 2) порти вводу/виводу (I/O);
- 3) управляючі виводи: сигнал тактування GCK, установки/скидування GSR, управління третім станом GTS.

Блоки вводу/виводу забезпечують буферизацію всіх входів та виходів ІМС. Кожен функціональний блок (ФБ) містить 18 макрокомірок (МК) із структурою “36 входів – 1 вихід” і дозволяє формувати 18 логічних функцій для будь-якої комбінації з 36 змінних. Матриця перемикачів (МП) забезпечує подачу будь-яких вхідних сигналів та вихідних сигналів ФБ на входи інших ФБ, а також подачу вихідних сигналів ФБ на блоки вводу/виводу.

Основні характеристики ряду CPLD-мікросхем фірми Xilinx наведено в табл. 1.2.

Програмовані логічні ІМС CPLD-технології широко застосовуються для проектування нестандартних арифметико-логічних пристроїв, дешифраторів, мультиплексорів та ін.

До недоліків CPLD (у порівнянні з FPGA) слід віднести: низьку кількість системних вентилів та високе енергоспоживання. Переваги їх полягають у більш високій швидкодії та забезпеченні можливості встановлення захисту від копіювання. Важливою перевагою також є те, що програмні засоби для розробки та синтезу систем на базі CPLD розповсюджуються вільно.

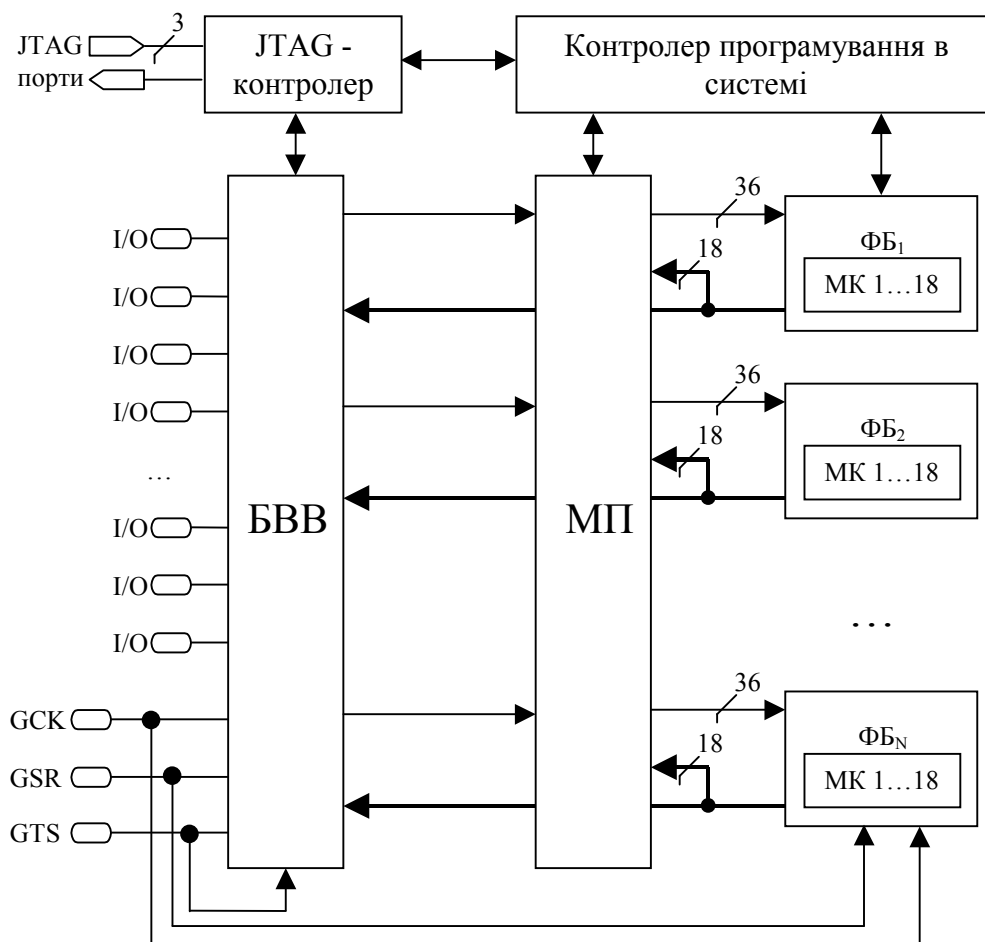


Рис. 1.3 – Узагальнена структура CPLD-мікросхеми

Табл. 1.2

Характеристика	Серія CoolRunner		Серія XC9500	
	XCR3320	XCR22V10	XC9536	XC952288
Кількість макрокомірок (шт)	320	10	36	288
Системна частота (MHz)	100	111	100	56.6
Напруга живлення (В)	5; 3	5; 3	5; 3	5; 3
Кількість циклів перепрограмування	1000	1000	10000	10000
Час "pin-to-pin" (нс)	7.5	7.5	5	15

1.4 Методи проектування логічної структури ПЛІС

Як вже зазначалося, стрімкий розвиток елементної бази спричинив виникнення необхідності у адекватному підвищенні продуктивності засобів та методів розробки ПЛІС. Історично засоби проектування ПЛІС розвивалися на основі трьох парадигм:

- логічного проектування;
- схемного проектування;
- проектування за допомогою мов опису апаратного забезпечення.

Остання концепція є найновішою і зараз знаходиться на стадії бурхливого розвитку. Ідея даного підходу полягає в тому, що проєктант описує на функціональному рівні необхідну (з точки зору алгоритму функціонування) поведінку цифрового пристрою за допомогою поведінкової моделі, а система автоматизованого проектування, відповідно, синтезує логічну структуру, що відтворює описану поведінку.

Поведінкові моделі можуть бути створені різними способами, зокрема, за допомогою спеціалізованих мов опису апаратного забезпечення, графів скінчених автоматів та ін. В даному навчальному посібнику основна увага приділяється методам синтезу цифрових пристроїв на основі мови опису апаратного забезпечення VHDL.

На рис. 1.5 зображено один з можливих варіантів побудови процесу проектування цифрового пристрою на базі ПЛІС від формулювання проектного (технічного) завдання до кінцевого результату – апаратно реалізованого зразка.

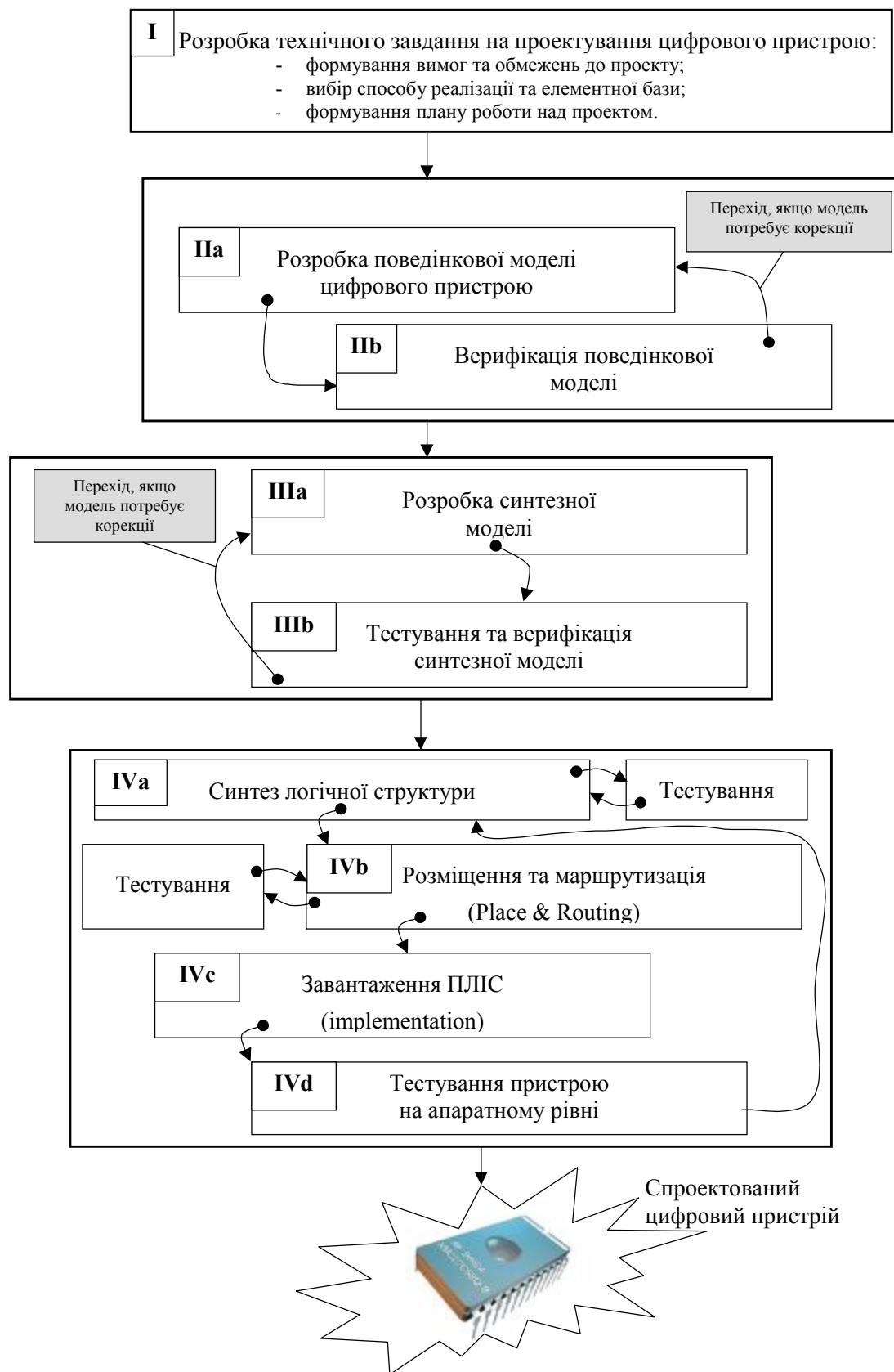


Рис. 1.5 – Поведінкове проектування цифрових пристроїв на базі ПЛІС

На всіх етапах процесу поведінкового проектування формується ряд моделей різного рівня абстракції, що відповідають різним аспектам функціонування цифрових пристроїв. При цьому суть процесу багатоетапного проектування полягає в переході від моделей з вищим рівнем абстракції до моделей з нижчим рівнем абстракції, тобто в поступовій деталізації опису цифрового пристрою на кожному з етапів. Наведемо класифікацію моделей, що застосовуються в процесі проектування:

ПОВЕДІНКОВА МОДЕЛЬ (Behavioral model, Interpreted model) – показує реакцію цифрового пристрою на зміну вхідних сигналів з урахуванням часу реакції. Ця модель не містить детального опису апаратної реалізації пристрою. Рівень абстракції залежить від рівня деталізації опису поведінкової моделі. Наприклад, на вищому рівні абстракції поведінкова модель може описувати процесор, що виконує абстрактний алгоритм, а на нижчому рівні – це може бути модель процесора з деталізацією системи команд (множини інструкцій) та алгоритмів їх виконання. Точність деталізації вхідних і вихідних даних залежить від рівня абстракції моделі.

ФУНКЦІОНАЛЬНА МОДЕЛЬ (Functional model) – функціональна модель описує функції системи без визначення способу реалізації цих функцій. Дана модель показує лише реакцію системи чи її компоненти, без урахування часового фактора (визначає значення виходу, але не час його встановлення). Рівень абстракції залежить від ступеня деталізації моделі. Рівень деталізації вхідних та вихідних даних залежить від рівня абстракції.

СТРУКТУРНА МОДЕЛЬ (Structural model) – структурна модель представляє компоненти системи з погляду їх ієрархії та взаємозв'язків між ними. Структурна модель має відповідати фізичній ієрархії в описуваному об'єкті. Ієрархія, в свою чергу, визначається фізичною організацією конкретної реалізації. Структурна модель описує фізичну

структуру конкретної реалізації шляхом визначення *компонент* та топології їх взаємозв'язків. Компоненти можуть бути описані на структурному, функціональному чи поведінковому рівні. Моделювання структурної моделі вимагає наявності поведінкових моделей всіх нижчих гілок ієрархії, отже ступінь деталізації аспектів модельного часу, значень об'єктів даних та функціональності структурної моделі залежить від ступеня деталізації моделей компонент.

МОДЕЛЬ ПРОДУКТИВНОСТІ (Performance Model, Uninterpreted Model) даний тип моделей дозволяє моделювати лише часові аспекти роботи цифрового пристрою (тобто швидкість реакції на зміну вхідного сигналу, без обчислення значення вихідного сигналу).

МОДЕЛЬ ІНТЕРФЕЙСУ (Interface Model, bus functional model) – можна порівняти з "чорним ящиком". Ця модель може містити деталізацію всіх аспектів обміну інформацією між об'єктом та зовнішнім середовищем, включаючи функціональність, часові характеристики, значення даних тощо. Така модель не містить інформації про внутрішню структуру об'єкта.

МОДЕЛЬ ЗМІШАНОГО РІВНЯ, Гібридна модель (Mixed-Level Model, Hybrid Model) – містить компоненти, що описані на різних рівнях абстракції або різними класами моделей.

ВІРТУАЛЬНИЙ ПРОТОТИП (Virtual Prototype) – Віртуальним прототипом називають комп'ютерну імітаційну модель кінцевого продукту (спроектованого пристрою), його компоненти чи системи з його включенням. При цьому від такої моделі не вимагається дотримання жодних спеціальних умов щодо її характеристик. Термін "віртуальний прототип" позначає клас моделей, що відіграють певну роль в процесі проектування, зокрема:

- ілюструє можливі варіанти реалізації проекту;
- демонструє концепцію проекту;

- дає можливість перевірки проекту на відповідність вимогам та адекватність поставленій задачі.

Повернемося до процесу проектування, наведеному на рис. 1.5. Весь процес розбито на 4 головних етапи.

На першому етапі формується технічне завдання на проектування цифрового пристрою, зокрема, формуються *інтерфейсна*, *продуктивна* та *функціональна* моделі, із застосуванням яких можна постійно здійснювати перевірку на відповідність технічному завданню моделей, сформованих на наступних етапах проектування. Так *продуктивна* модель визначає швидкодію проектованого пристрою, *інтерфейсна* – спосіб його інтеграції до вищого ієрархічного рівня, а *функціональна* модель – алгоритм перетворення інформації в проектованому пристрої.

На другому етапі формується *поведінкова* модель. У VHDL під власне *поведінковою* розуміється модель, що написана із застосуванням всіх наявних в цій мові конструкцій та типів даних, наприклад дійсних чисел, файлів, вказників динамічної пам'яті тощо. Поведінкова модель, яка розробляється на цьому етапі, повинна повністю відповідати вимогам та обмеженням, сформульованим на першому етапі. Саме це й перевіряється на підетапі IIб.

Після формування остаточного вигляду поведінкової моделі проєктант переходить до Етапу III – створення синтезної моделі. Синтезна модель також відноситься до класу поведінкових, однак може бути написана лише за допомогою певної підмножини конструкцій мови VHDL, які підтримуються засобами синтезу логічної структури. На сучасному етапі розвитку засоби синтезу (перетворення VHDL-програм на схеми логічних елементів) підтримують не всі наявні у VHDL мовні конструкції, зокрема, не підтримуються операції з дійсними числами та вказники. Такі конструкції як, наприклад, файли, зі зрозумілих причин відносяться до природньо-несинтезованих. Перехід від поведінкової моделі до

синтезованої характеризується зниженням рівня абстракції описання цифрового пристрою.

На четвертому етапі (етапі синтезу) здійснюється перехід від синтезної моделі до логічної структури та бітового потоку, який завантажується безпосередньо в ПЛІС. Підетап IVa представляє собою перехід від синтезної моделі до моделі рівня регістрових передач (Register Transfer Level Model – RTL-model). Така модель включає лише стандартні компоненти цифрових пристроїв, такі як регістри, лічильники, дешифратори, тригери тощо. На підетапі IVb (Place&Routing – розміщення і маршрутизація) RTL-модель розміщується на ПЛІС (наприклад, на КЛБ для FPGA) з формуванням зв'язків між окремими компонентами.

На підетапі IVc формується бітовий потік, який завантажується безпосередньо до ПЛІС. Після цього проводиться остаточне тестування вже апаратно реалізованого цифрового пристрою.

На всіх етапах тестування як генератор еталонних вихідних значень цифрового пристрою використовується поведінкова модель, що розроблена на другому етапі.

В наступних розділах перейдемо безпосередньо до конструкцій мови VHDL, за допомогою яких можна створювати перелічені моделі.

2. АЛФАВІТ МОВИ VHDL

2.1 Правила формування ідентифікаторів

Ідентифікатори в VHDL поділяються на прості й розширені. Прості ідентифікатори формуються за наведеними нижче правилами.

1. Ідентифікатори можуть включати латинські букви, цифри та знак підкреслення “_”.
2. Ідентифікатори повинні починатися з букви.
3. Ідентифікатори не повинні закінчуватися підкресленням.
4. Ідентифікатори не можуть включати в себе два знаки підкреслення підряд.
5. Ідентифікаторами не можуть бути зарезервовані слова стандартного VHDL, повний список яких наведено нижче:

abs	disconnect	is	out	severity
access	downto	label	package	signal
after	else	library	port	sra
alias	elseif	linkage	shared	srl
all	end	literal	sla	subtype
and	entity	loop	sll	then
architecture	exit	map	postponed	to
array	file	mod	procedure	transport
assert	for	nand	process	type
attribute	function	new	pure	unaffected
begin	generate	next	range	units
block	generic	nor	record	until
body	group	not	register	use
buffer	guarded	null	reject	variable
bus	if	of	rem	wait
case	impure	on	report	when
component	in	open	return	while
configuration	inertial	or	rol	with
constant	inout	others	ror	xnor
			select	xor

Крім того, мова VHDL дає можливість застосування розширених ідентифікаторів. Такі ідентифікатори можуть складатися з будь-яких символів, обмежених з двох сторін символами “\” (слеш), наприклад, **\Ідентифікатор 1&2**.

Приклади правильно сформованих ідентифікаторів:

Decoder_1

FFT

Sig_N

Not_Ack

\signal

\C:\\Cads

\Сигнал @#

Приклади неправильно сформованих ідентифікаторів:

_Decoder_1

2FFT

Sig_#N

Not-Ack

Сигнал_1

2.2. Спеціальні символи

Крім ідентифікаторів мова VHDL включає в себе спеціальні символи:

& ‘ () * + , - . / : ; < = > |

та пари спеціальних символів:

=> ** := /= >= <= <>

2.3. Числа в VHDL

В VHDL застосовуються цілі та дійсні числа.

Приклади запису цілих чисел:

23 0 146 -10

Приклади запису дійсних чисел:

23.1 12.0 3.11459 3.234E09 8.6E+21 34.0E-08

Необхідно звернути увагу на те, що прийнята в VHDL форма запису дійсних чисел обов'язково вимагає наявності хоча б одного знака після крапки, навіть якщо це нуль.

При необхідності числа можна задавати в системах зчислення відмінних від 10 (з основою від 2 до 16), *наприклад*:

2#11011# 2#0011# 16#10FA# 10#1024#E+00

В першому прикладі:

2	#	11011	#
основа	спец.символ	число	спец.символ

Цифри, більші ніж 9 задаються за допомогою латинських літер A, B, C, D, E, F.

Для більшої наочності запису довгих чисел всередині їх можна вставляти знак підкреслення “_”. На інтерпретацію числа комп'ютером цей знак не впливає, однак спрощує сприйняття цього числа програмістом-оператором, *наприклад*:

1_050_406	еквівалентно	1050406
2#1010_1001_1011_1110#	еквівалентно	2#1010100110111110#

2.4. Символи

Символи в VHDL – аналог типу *char* в мові Pascal. При запису вони поміщаються в одинарні лапки, наприклад:

‘a’ ‘b’ ‘;’ ‘>’

2.5. Рядки

На відміну від Pascal, в VHDL рядки поміщаються не в одинарні, а в парні лапки, наприклад:

“Рядок 1”

“Вивчаємо VHDL”

Якщо існує необхідність задавати довгі рядки, що не поміщаються в один екранний рядок, то можна записувати їх в декілька екранних рядків, ставлячи на початку знак конкатенації “&”, *наприклад*, запис

“Довгий, довгий, довгий рядок”

еквівалентний запису

“Довгий, довгий”

&”, довгий”

&” рядок”

3. СТРУКТУРА МОВИ VHDL

3.1. Умовний оператор

Умовний оператор дозволяє реалізувати розгалужені обчислювальні процеси. В мові VHDL існує можливість вкладання умовних операторів один в один на будь-яку глибину.

Синтаксис умовного оператора має вигляд:

```
if Boolean_Expression_1 then  
    Послідовність_Операторів_1  
elseif Boolean_Expression_2 then  
    Послідовність_Операторів_2  
else  
    Послідовність_Операторів_3  
end if;
```

Виконується умовний оператор таким чином: якщо результатом обчислення *Boolean_Expression_1* є “Істина”, то виконується *Послідовність_Операторів_1*, інакше обчислюється *Boolean_Expression_2*. Якщо результатом обчислення *Boolean_Expression_2* є “Істина”, то виконується *Послідовність_Операторів_2*, інакше виконується *Послідовність_Операторів_3*.

3.2. Оператор вибору

Оператор вибору **case** дозволяє вибрати одне з кількох можливих продовжень програми.

Формат запису оператора вибору в VHDL наведено нижче:

```
case    Селектор is  
    when choices_1 => послідовність_операторів_1;  
    when choices_2 => послідовність_операторів_2;
```

```

when others => послідовність_операторів_3
end case;

```

де

Селектор – змінна (вираз) порядкового типу;

choices_1, *choices_2* – локально-статично визначені вирази того ж типу, що й *expression*.

При цьому *локально-статично визначеним* називається вираз, значення якого може бути обчисленим на етапі компіляції.

Оператор вибору працює таким чином: якщо *choices_1* містить значення змінної *Селектор*, то виконується послідовність_операторів_1, якщо *choices_2* містить значення змінної *Селектор*, то виконується послідовність_операторів_2. У всіх інших випадках виконується послідовність_операторів_3.

Приклад:

```

variable k: integer := 3;
variable a: integer;
...
case    k    is
  when 1 => a := 5;
  when 3 => a := 6;
  when 2 => a := 7;
  when others a := 8;
end case;

```

В розглянутому прикладі в ролі *Селектора* виступає змінна *k*, так як *k* = 3, то змінна *a* отримає значення 6.

У випадку, коли альтернативи *choices* не покривають всього діапазону можливих значень *Селектора*, наявність альтернативи **when others** є обов'язковою.

Альтернативи *choices* можуть задаватися не тільки одиночними значеннями, а й діапазонами, наприклад:

```
when 'a' to 'c' => ...
when 0 to 15 => ...
```

Якщо одна альтернатива складається з декількох виразів, то вони записуються через знак '|' (об'єднуються за операцією “OR”), наприклад:

```
when 1|2|5|8 => ...
when 1 to 7|18 to 25 => ...
```

Альтернативу можна задавати підтипами, *наприклад*:

```
subtype New_Integer is integer range 0 to 10;
...
case k is
    when New_Integer => ...
    when New_Integer|11 =>...
end case;
```

3.3. Порожній оператор

Для запису порожнього оператора в мові VHDL використовується ключове слово **Null**. Цей оператор може застосовуватися для відлагоджувальних цілей на етапах розробки програм, *наприклад*:

```
if a < b then
    null;
end if;
```

3.4. Цикли

Цикли застосовуються для програмування фрагментів, що повторюються. Базовим для всіх циклічних конструкцій в VHDL є нескінчений цикл.

3.4.1. Нескінчений цикл

Формат запису нескінченного циклу має вигляд:

```
Loop_Label: loop  
    послідовність_операторів  
end loop Loop_Label;
```

Тут *Loop_Label* – мітка циклу (може бути пропущена). Розглянемо фрагмент програми:

```
Unlimited: loop  
    a:=a+1;  
end loop Unlimited;
```

Даний приклад буде нескінченно збільшувати змінну *a*.

3.4.2. Оператор Exit

Хоч нескінчені цикли і зустрічаються на практиці, однак набагато частіше виникає необхідність в циклах, які закінчуються при виконанні якоїсь умови. Для реалізації таких циклів в VHDL використовується оператор **Exit**.

Формат запису оператора **Exit**:

```
exit loop_label when boolean_expression;
```

Його найпростіша форма має вигляд:

```
exit;
```

За цією командою переривається виконання поточного циклу і управління передається на оператор, що є наступним після **end loop**.

В реальних програмах часто виникає необхідність в застосуванні фрагмента

```
if boolean_expression then
    exit;
end if;
```

який перериває виконання поточного циклу при досягненні *boolean_expression* значення **TRUE**.

Існує спрощена форма запису такого фрагмента:

```
exit when boolean_expression;
```

Застосовуючи мітки можна реалізувати вкладені цикли, наприклад, таким чином:

```
outer: loop
...
    inner: loop
        ...
        exit outer when Умова_1; -- Вихід з зовнішнього
        ...                      -- циклу по мітці
        exit when Умова_2;       -- Вихід з поточного
        ...                      -- (внутрішнього) циклу
    end loop inner;
end loop outer;
```

3.4.3. Оператор Next

За допомогою оператора **Next** можна перейти одразу до початку наступної ітерації циклу (тобто до оператора **loop**).

Синтаксис оператора **Next** такий же, як і оператора **Exit**:

```
next when boolean_expression;
```

3.4.5. Цикл “Поки” (**While**)

За допомогою оператора **exit** можна реалізувати будь-який цикл (“Поки”, “До”, з параметром), однак для спрощення запису та полегшення розуміння програм в мові VHDL введено спеціальні конструкції для основних типів циклічних операторів. Цикл “Поки” записується таким чином:

```
loop_label: while boolean_expression loop  
    послідовність_операторів  
end loop;
```

Працює цей цикл таким чином: поки *boolean_expression* має значення **TRUE** - виконується тіло циклу (послідовність_операторів).

Всередині циклу **while** можна використовувати будь-які оператори, включаючи **exit** та **next**.

3.4.6. Цикл з параметром (**for**)

Крім циклу “Поки” в мові VHDL існує також спеціальна конструкція для реалізації циклу з параметром. Його синтаксис:

```
loop_label: for parameter in diapason loop  
    послідовність_операторів  
end loop;
```

Тут *diapason* – діапазон порядкового типу, наприклад :

```
0 to 10  
10 downto 0.
```

Діапазон також може задаватися підтипом:

```
subtype New_Integer is integer range 0 to 10;
...
loop_with_subtype: for i in New_Integer loop
    ...
end loop loop_with_subtype;
```

Суттєва відмінність циклу з параметром у VHDL від аналогічних конструкцій в мовах Pascal та C++ полягає в тому, що параметр тут не являється змінною. Параметр не потрібно попередньо об'являти, а всередині циклу параметр може розглядатися як константа (тобто його можна використовувати в виразах, але не можна присвоювати параметру значення), після закінчення циклу параметр зникає, наприклад:

```
Error_Parametr_Demo: process is
variable i,j: integer;
begin
    i:=loop_par;      --Error (Невідомий ідентифікатор loop_par)
    for loop_par in 1 to 10 loop
        j := loop_par;
        loop_par:=5; -- Error (Параметр циклу не може
                       -- виступати як приймач оператора
                       -- присвоєння)
    end loop;
    j:= loop_par;    --Error (Невідомий ідентифікатор loop_par)
end Error_Parametr_Demo;
```

Рядки, в яких наведено неправильне застосування параметра в даному прикладі, помічено коментарем Error.

3.5. Змінні

В попередньому розділі розглядалося таке поняття, як «сигнал». Очевидно, що обмеження, які накладаються на використання сигналів (відсутність можливості декларування сигналу і зберігання проміжних даних всередині процесу; нове значення сигналу присвоюється тільки після закінчення роботи процесу) обумовлюють серйозні труднощі щодо їх практичного використання. Таким чином, виникає необхідність в об'єкті даних, який би компенсував всі відзначені «недоліки» сигналів. У VHDL таким об'єктом є «змінна».

Змінні можуть бути оголошені тільки всередині процесу, в якому вони використовуються, і є локальними об'єктами цього процесу. Оголошення змінних здійснюється наступним чином:

```
variable list_of_var_names: type_name [:= initial_value];
```

де

list_of_var_names – список змінних;

type_name – тип змінних;

initial_value – початкове значення;

[...] - вирази в квадратних дужках [...] є необов'язковими.

Основною функцією змінних є зберігання проміжної інформації всередині процесів і підпрограм (процедур і функцій).

3.6. Константи

Крім сигналів і змінних VHDL може оперувати джерелом статичної інформації в рамках окремо взятої архітектури – константою. Оголошення

константи робиться в розділі оголошень архітектури, діє тільки в рамках цієї архітектури і має вигляд:

```
constant constant_names: type_name: = constant_value;
```

де

constant_names – ідентифікатор константи;

type_name – тип константи;

constant_value – значення константи.

Зазвичай константи використовуються для:

- призначення розміру складних об'єктів (масивів, шин і т.п.);
- призначення числа ітерацій циклів;
- визначення параметрів часу (тимчасових установок, затримок, моментів перемикання і таке інше):

```
constant delay1: time: = 10 ns.
```

Примітка: Якщо константа оголошена не в архітектурі, а в *процесі*, то вона діє тільки в межах цього *процесу*.

3.7. Типи даних в VHDL, перетворення типів

VHDL містить ряд стандартних типів, які визначені в пакетах Standard і Std_Logic_1164. Наведемо деякі з них:

<i>BIT</i>	'0' або '1'
<i>BOOLEAN</i>	FALSE або TRUE
<i>INTEGER</i>	Цілі числа в діапазоні від $-(2^{31}-1)$ до $+(2^{31}-1)$

<i>REAL</i>	Числа з плаваючою комою в діапазоні від -1.0E38 до +1.0E38.
<i>CHARACTER</i>	Символьний тип
<i>TIME</i>	Цілочисельний тип з розмірністю fs, ps, ns, us, ms, sec, min, hr.

Крім стандартних типів VHDL дозволяє користувачеві створювати самостійно нові типи об'єктів даних. Найбільш загальними є так звані порядкові типи, в яких визначається повний список значень, що належать даному типу. Так, наприклад, запис:

```
type user_type is (V1, V2, V3, V4, V5, V6, V7);  
signal Sig1: user_type: = V1;
```

визначає деякий сигнал *Sig1*, що може приймати одне з значень, вказаних в дужках *V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *V7*, і набуває початкового значення *V1*. Якщо початкове значення не вказується явно, то змінна чи сигнал набуває крайнього лівого значення типу. Наприклад, запис

```
variable A: integer;
```

дасть початкове значення змінної *A* = -2 147 483 647.

Відзначимо, що VHDL строго типізована мова. Таким чином, ні сигнали, ні змінні різних типів звичайно не можуть використовуватися в одному виразі. При цьому, в даній мові опису обладнання не передбачені засоби автоматичного (неявного) перетворення типів. Так, наприклад, вираз *A <= B or C* правомірний тільки у випадку, коли *A*, *B* і *C* одного типу або аналогічних типів. Операції над сигналами (змінними) аналогічних

типів вимагають їх обов'язкового узгодження. Таким чином, узгодження може виконуватися тільки для сумісних типів і у відповідності з наступними правилами:

Цілі (*integer*) і дійсні (*real*) типи є аналогічними типами даних. При перетворенні дійсних чисел в цілі, результат округляється до найближчого цілого числа.

Два масиви розглядаються, як аналогічні, якщо вони:

- мають однакову розмірність;
- елементи обох масивів мають однаковий тип;
- індекси обох масивів мають однаковий тип або їх типи аналогічні.

Всі інші типи не являються аналогічними.

Приведемо приклад узгодження типів:

```
Variable A, B: integer;

...

A:=integer(25.17*real(B));
```

3.8. Масиви

В VHDL масив інтерпретується як тип, значення якого складається з ряду елементів єдиного підтипу. Кожен елемент масиву відрізняється своїм індексом або рядом індексів (в багатовимірних масивах). Індекс масиву повинен мати порядковий тип і знаходитись в області допустимих значень. Таким чином, перш ніж застосовувати масив в VHDL, необхідно оголосити тип масиву (а), а потім оголосити масив, як об'єкт (б):

```
(a) type array_type_name is array index_range of element_type;
```

```
(б) variable array_name: array_type_name [:= initial_values];
```

Приклад одновимірного масиву:

```
type A is array (5 downto 0) of bit;  
variable B : A := "0 1 1 0 0 1";
```

A(0)

Для багатовимірного масиву порядок і структура оголошення масиву залишаються така ж сама:

```
type A is array (1 to 3, 1 to 2) of integer;  
variable B : A := ((0 , 1), (5 , 2), (7 , 8));
```

В результаті отримаємо масив, зображений на рис. 3.1.

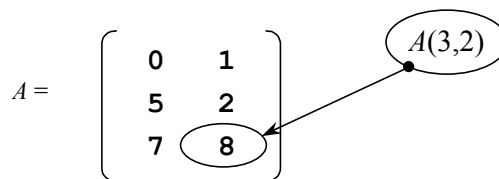


Рис. 3.1 – Двовимірний масив

Масивами можуть бути не тільки змінні, а й сигнали та константи. Тобто, в наведених прикладах замість службового слова **variable** може стояти **signal** або **constant**.

При оголошенні типу масиву дозволяється не визначати діапазон зміни його індексу (в). Такі типи масивів називаються необмеженими і застосовуються при написанні бібліотек та конфігурованих об'єктів. Навіть в цьому випадку при створенні екземплярів необмежених масивів слід вказувати локально-статично визначені діапазони зміни індексів (г):

```
в) type arr_type is array (index_type range <>) of
```

```
elements_type;
r) variable arr_name: arr_type (index_range)[:initial_val];,
```

де

index_type – тип індексів масиву (*integer*, *natural* etc.).

Багатовимірні необмежені масиви і їх типи оголошуються аналогічно до масивів з обмеженими індексами:

```
type multy is array (integer range <>, bit range <>, . . . ,
boolean range <>) of elements_type;
```

3.9 Підпрограми у VHDL

Як і у більшості мов програмування високого рівня у VHDL існують конструкції для описання та виклику підпрограм. Існує два класи підпрограм – процедури та функції. Спершу розглянемо процедури.

3.9.1 Процедури у VHDL

Синтаксис процедури в мові VHDL:

```
procedure Ідентифікатор (Список_Формальних_Параметрів) is
    Розділ_Локальних_Оголошень;
begin
    Послідовність_Операторів;
end procedure Ідентифікатор_Процедури;
```

Де *Ідентифікатор* визначає ім'я процедури і *Послідовність_Операторів* реалізує алгоритм, що виконується процедурою. В *Розділі_Локальних_Оголошень* можна створити локальні

(не доступні за межами процедури) змінні, константи, сигнали, типи, підтипи та підпрограми. *Список_Формальних_Параметрів* визначає механізм обміну даними між процедурою та програмою. Оператор оголошення процедури можна розміщувати в розділах оголошень архітектур, процесів та пакетів (дивіться наступний розділ).

Розглянемо спочатку приклад процедури без *Списку_Формальних_Параметрів*. Обмін інформацією між програмою, що викликає процедуру, та самою процедурою все одно буде можливим, так як всередині процедури дозволяється читати інформацію з глобальних об'єктів даних (змінних, сигналів, констант) та змінювати їх значення:

```
architecture Arch_Name of Entity_Name is
type TPoints is array (1 to 50) of integer;
begin
  process is
    variable XPoints, YPoints : TPoints;
    variable Min_Number: Integer;
    procedure Min_Element is
    begin
      Min_Number:=1;                                (**)
      for i in XPoints'Range loop
        if XPoints(i) < XPoints(Min_Number) then
          Min_Number := i;
        end if;
      end for;
    end procedure Min_Element;
  begin
    ...;
    Min_Element;                                    (*)
    X_Min <= XPoints (Min_Number);                  (***)
  end process;
end architecture Arch_Name;
```

В розглянутому прикладі процедура *Min_Element* реалізує алгоритм обчислення мінімуму. Результат її роботи – номер мінімального елементу масиву *XPoints* передається в процес через глобальну (відносно процедури) змінну *Min_Number*. Виклик процедури здійснюється в рядку (*), як видно з прикладу, оператор виклику процедури містить лише її ідентифікатор. При виконанні даного оператора управління передається на початок *Послідовності_Операторів* процедури, тобто на рядок (**). Після закінчення процедури (досягнення оператора **end procedure**) управління повертається до процесу на рядок (***), наступний за викликом процедури.

Існує можливість дострокового переривання виконання з поверненням до блоку, що викликав поточну процедуру. Для цього застосовується оператор **return**, що не має жодних додаткових параметрів.

Процедури без списку формальних параметрів звичайно пишуться для конкретного випадку і їх повторне застосування утруднене, через необхідність дотримуватися у всіх викликах одних і тих же ідентифікаторів для глобальних змінних, через які здійснюється обмін інформацією між процедурою та програмою.

Полегшити повторне застосування процедур можна введенням *Списку_Формальних_Параметрів*, які представляють собою псевдоніми для *фактичних параметрів*. Фактичними параметрами називають ті об'єкти даних, через які інформація надходить до процедур або здійснюється передача результатів роботи процедур. Переробимо попередній приклад із застосуванням механізму формальних та фактичних параметрів:

```
architecture Arch_Name of Entity_Name is
type TPoints is array (1 to 50) of integer;
```



```

begin
  process is
    variable XPoints, YPoints : TPoints;
    variable Min_Number: Integer;
    procedure Min_Element (variable Points: in TPoints;
                           variable Min_N: out integer) is
    begin
      Min_N:=1;
      for i in Points'Range loop
        if Points(i) < Points(Min_N) then
          Min_N := i;
        end if;
      end for;
    end procedure Min_Element;
  begin
    ...;
    Min_Element (XPoints, Min_Number);           (*)
    X_Min <= XPoints (Min_Number);
    Min_Element (YPoints, Min_Number);           (**)
    Y_Min <= YPoints (Min_Number);
    ...;
  end process;
end architecture Arch_Name;

```

В наведеному прикладі процедура *Min_Element* має 2 формальні параметри – вхідну змінну *Points* типу *TPoints* та вихідну змінну *Min_N* типу *integer*. В рядку (*) до виклику процедури додано список фактичних параметрів. Таким чином якщо при обробці даного виклику процедура звертається до формального параметру змінної *Points*, то звертання переадресовується до фактичного параметру *XPoints*.

Перевагу даного механізму добре ілюструє виклик цієї ж процедури в рядку (**) для інших фактичних параметрів. Щоб знайти мінімальний

елемент масиву *YPoints* за допомогою процедури без списку формальних параметрів, слід було б переслати інформацію з *YPoints* до масиву *XPoints*, що може викликати ряд незручностей з точки зору програміста. У розглянутому ж прикладі достатньо замінити один фактичний параметр іншим, отже така процедура є більш універсальною і придатною до повторного застосування.

В загальному випадку список формальних параметрів формується з правилами:

```
( Клас_Об'єкту_Даних Ідентифікатор : Режим Тип_Параметра :=
    Значення_По_Замовчуванні; ...)
```

де

Клас_Об'єкту_Даних – визначає який саме об'єкт даних передається процедурі як формальний параметр і може приймати значення **constant**, **signal** або **variable**, якщо **Клас** не задано явно, то він вибирається автоматично, в залежності від **Режиму**;

Ідентифікатор – ідентифікатор формального параметра;

Тип_параметра – тип інформації що передається через формальний параметр;

Режим – напрям передачі інформації в процедуру. Може приймати одне з трьох значень – **in**, **out** та **inout**. В залежності від **Режиму**, компілятор VHDL може самостійно визначити **Клас_Об'єкту_Даних** що передається, приклад ілюструє яким чином відбувається це визначення:

```
procedure NewProcedure (
    A1 : in integer;           -- вибирається клас constant
    A2 : inout integer;        -- вибирається клас variable
    A3 : out integer;          -- вибирається клас variable
    signal A4 : in integer;    -- явно задано клас signal)
```

```

constant A5 : out integer; -- помилка, так як константа не
    -- може змінювати значення і, відповідно, служити
    -- вихідним параметром
        ) is ...

```

Як видно з прикладу, у всіх випадках, коли є необхідність у передачі інформації за межі процедури, автоматично вибирається клас **variable**. Клас **signal** завжди слід задавати явно.

В мові VHDL існує можливість задавати значення формального параметру процедури по замовчанню. Розглянемо приклад:

```

procedure NewProcedure (
    A1 : in integer;
    A2 : in integer := 5;
    A3 : out integer ) is ...
end procedure NewProcedure;
...
NewProcedure (A, open, X);           (*)
...;

```

В наведеному прикладі другий формальний параметр має значення по замовчуванню 5. При виклиці процедури (рядок (*)) замість другого фактичного параметра поставлене ключове слово **open**, що і означає команду застосовувати значення по замовчуванню.

3.9.2 Функції у VHDL

Застосування функцій у VHDL подібне до решти мов високого рівня. Синтаксис функцій:

```
pure / impure function Ідентифікатор
    ( Список_Формальних_Параметрів) return Тип_Результату is
```

В мові VHDL функції не містять спеціального ідентифікатора для позначення об'єкта даних, що повертається в якості результату роботи функції. Натомість, всі функції повинні містити оператор **return**, параметром якого є значення результату роботи. Наприклад:

```
pure function Min (a, b: in integer) return integer is
begin
    if a<b then
        return a;
    else
        return b;
    end if;
end function Min;
```

Якщо всі процедури мають доступ до глобальних об'єктів даних, то для функцій такий доступ можна штучно заборонити. Це робиться для того, щоб для певного набору значень вхідних параметрів функції результат роботи функції був однаковим не залежно від умов роботи цієї функції (значень глобальних об'єктів даних). Якщо така штучна заборона потрібна та на початок заголовка функції ставиться ключове слово **pure**, в іншому випадку – слово **impure**.

Крім того, за допомогою функцій можливо перевизначати операції (такі як "+", "-", "<" тощо) для створених користувачем нових типів даних. Не зупиняючись докладно на даному питанні наведемо приклад такої функції та зазначимо, що *Список_Формальних_Параметрів* для всіх таких функцій має один і той же вигляд (дивіться приклад), а який саме алгоритм

обчислення операцій слід обирати у виразах компілятор визначає за типом операндів. Приклад:

```
function "+" (left, right: in bit_vector )           -- (*)
    return bit_vector is
begin
    ...
end function "+" ;
...
variable x,y,z : bit_vector;
begin
    ...
    z := x + y;    -- так як тип операндів bit_vector,
                   -- то викличеться функція (*)
    ...
```

3.10 Пакети та бібліотеки

Найбільш зручним шляхом групування елементів програми згідно їх функціонального призначення є застосування пакетів (**package**). Звичайно в пакети групують певні типи даних, а також процедури та функції для обробки цих типів.

Так як пакети представляють собою повністю виділені програмні блоки, то їх можна розробляти незалежно від інших елементів програми і, в подальшому, паралельно застосовувати в багатьох **entity** та архітектурах. Важливим для підвищення зрозумілості програмного коду є відділення у пакетах заголовків програмних елементів від їх реалізацій.

3.10.1. Оголошення пакету

Пакет складається з двох частин — *оголошення пакету* (**package declaration**) та *тіла пакету* (**package body**). Синтаксис оголошення пакету:

```
package Ідентифікатор is
    Розділ_Оголошень_Пакету;
end package Ідентифікатор;
```

В *Розділі_Оголошень_Пакету* можна розміщувати визначення типів, підтипів, констант та сигналів, а також заголовки процедур та функцій. Всі ці програмні елементи будуть доступні користувачам пакету. Нижче наведено простий *приклад* оголошення пакету:

```
package Pascal_Types is
    type Word is integer range 0 to 65535;
    type Byte is integer range 0 to 255;
    type Short is integer range -128 to 127;
end package Pascal_Types;
```

Пакет звичайно розміщується в окремому файлі проекту і є незалежним елементом бібліотеки. В будь-яких інших файлах проекту можливе застосування програмних елементів, описаних в пакеті, причому звертатись до цих елементів слід наступним чином:

```
Ім'я_бібліотеки.Ім'я_пакету.Ідентифікатор_елемента
```

Якщо пакет описано в поточному проекті, то *Ім'я_бібліотеки* замінюється ключовим словом **work**. Так для створення змінної типу Word з попереднього прикладу, за умови що пакет і його виклик знаходяться в одному проекті, слід скористатися оператором:

```
variable X: work.Pascal_Types.Word;
```

При розміщенні в пакетах процедур чи функцій в *оголошенні пакету* визначаються лише їх заголовки з ідентифікаторами підпрограм та їх інтерфейсом (переліком вхідних та вихідних параметрів із зазначенням типів), приклад такого заголовка:

```
procedure min (x1, x2: in integer; res: out integer);
```

Повна реалізація процедур та функцій з програмним кодом розміщується в тілі пакету.

При визначенні констант в *оголошеннях пакетів* також можна одразу не вказувати їх конкретні значення, *наприклад*:

```
constant Setting_Time: Time;
```

В цьому випадку, як і в попередньому, повне описання константи з її значенням розміщується в тілі пакету.

3.10.1. Тіло пакету

Повні реалізації програмних елементів, інтерфейс яких описано в *Розділі_Оголошення_Пакету*, поміщуються в тіло пакету (**package body**), синтаксис якого наведено нижче:

```
package body Ідентифікатор is  
    Розділи_оголошень;  
end package body Ідентифікатор;
```

Зрозуміло, що заголовки попередньо описаних в *оголошенні пакету* програмних елементів мають повністю відповідати заголовкам їх реалізацій в *тілі пакету*.

3.10.2. Оператор **use**

При інтенсивному використанні програмних елементів з пакета фрагмент тексту “*Ім’я_бібліотеки.Ім’я_пакету.*” буде постійно повторюватися в програмі. Цього можна уникнути шляхом введення оператора **use** з синтаксисом:

```
use Ім'я_бібліотеки.Ім'я_пакету.Ідентифікатор_елемента;
```

Після введення даного оператора, що може розміщуватися в будь-якому місці програмного файлу, *елемент* стає безпосередньо доступним в межах даного файлу. Так, наприклад, запис:

```
entity RS_Trigger is
    port( S, R:           in IEEE.std_logic_1164.Std_Logic;
          D, Inv_D: out IEEE.std_logic_1164.Std_Logic;
end entity RS_Trigger;
```

є адекватним запису:

```
library IEEE;
use IEEE.std_logic_1164.Std_Logic;
entity RS_Trigger is
    port( S, R:           in Std_Logic;
          D, Inv_D: out Std_Logic;
end entity RS_Trigger;
```


Якщо є необхідність у застосуванні всіх програмних елементів, що входять до складу пакету, то замість *ідентифікатора програмного елемента* слід поставити ключове слово **all**. Так при автоматизованому створенні файла VHDL-коду САПР Active-HDL додає до програми наступні рядки:

```
library IEEE;  
use IEEE.std_logic_1164.all;
```

що дозволяють застосовувати всі типи та підпрограми пакету стандартної логіки.

4. ЗАСОБИ VHDL ДЛЯ МОДЕЛЮВАННЯ РЕАЛЬНИХ ОБ'ЄКТІВ

4.1. Опис об'єктів в VHDL (Entity)

VHDL є мовою опису реальних об'єктів та процесів, що протікають в цих об'єктах. Для такого типу описів VHDL містить ряд спеціалізованих конструкцій.

Опис об'єкта в VHDL починається з визначення зв'язків між об'єктом та зовнішнім середовищем. Повний перелік таких зв'язків називається *інтерфейсом*.

Синтаксис опису інтерфейсу об'єктів:

```
entity identifier is
    port ( port_interface_list);
    entity_declarative_item1;
    entity_declarative_item2;
    ...
    entity_declarative_item;
end entity identifier;
```

де

identifier – ім'я об'єкта;

port_interface_list – інтерфейс об'єкта;

entity_declarative_items – необов'язковий список об'явлень, що може містити описання констант, типів, сигналів (більш детально буде розглянуто нижче);

port_interface_list містить повний перелік всіх портів (вхідних та вихідних сигналів) об'єкта з означенням їх типу та напрямку (вхідний, вихідний, двонаправлений).

Синтаксис опису портів:

```
(identifier1: mode  type := expression1;  
  identifier2: mode  type := expression2;  
  ...);
```

де

identifier1, *identifier2* – імена портів;

mode – параметр, що вказує на напрям порту і може приймати одне з трьох значень: **in** – вхідний, **out** – вихідний та **inout** – двонаправлений;

type – тип порту;

expression – початкове значення порту.

Для прикладу розглянемо опис інтерфейсу чотирьох-розрядного суматора з двома вхідними та одним вихідним портами:

```
entity adder is  
  port(  a: in Std_Logic_Vector (3 downto 0);  
         b: in Std_Logic_Vector (3 downto 0);  
         c: out Std_Logic_Vector (3 downto 0) := "0000");  
end entity adder;
```

В одному рядку можна об'явити кілька однакових за типом та напрямом портів. Так попередній приклад можна переписати у вигляді:

```
entity adder is  
  port(  a, b: in Std_Logic_Vector (3 downto 0);  
         c: out Std_Logic_Vector (3 downto 0) := "0000");  
end entity adder;
```

4.2. Архітектура

Після опису інтерфейсу об'єкта необхідно описати його архітектуру. *Архітектурою* називається внутрішня структура об'єкта, що визначає його

поведінку. Саме архітектура визначає яким чином вхідні порти об'єкта поєднуються з вихідними. Синтаксис архітектури:

```
architecture identifier of entity_name is
  Block_Declarative_Item;
begin
    Concurrent_Statements
end architecture identifier;
```

де

identifier – ім'я архітектури;

entity_name – ім'я об'єкта, поведінку якого описує дана архітектура;

Block_Declarative_Item – список елементів програми, що будуть доступні в межах архітектури; в цьому розділі можна оголошувати константи, типи, компоненти та сигнали.

Поведінка об'єкта описується всередині архітектури за допомогою конкурентних операторів (*Concurrent_Statements*). В послідовних мовах програмування, таких як Pascal та C++ поняття конкурентних операторів відсутнє. Ці оператори призначаються для реалізації паралельних процесів і виконуються одночасно. До конкурентних операторів відносяться процеси (**process**), компоненти та деякі інші.

4.3. Сигнали

Сигнали використовуються для зв'язків між різними об'єктами (**entity**) та процесами, що протікають в цих об'єктах. Фізично сигнал можна представити у вигляді лінії зв'язку, яка поєднує об'єкти або структурні складові всередині об'єктів. Крім того, така лінія зв'язку може нескінченно довго зберігати подане на неї значення сигналу. В реальних об'єктах будь-які процеси не можуть перебігати миттєво (стрибкоподібно), тому важливою особливістю *сигналу*, як моделі реального процесу

передачі інформації, є те, що сигнал не може змінювати свого значення миттєво. Тобто, між командою на зміну значення сигналу та реальною зміною значення завжди пролягає певний проміжок модельного часу.

Перед застосуванням сигнал необхідно попередньо об'явити. Синтаксис об'явлення сигналу має вигляд:

signal *Ідентифікатор*: *Тип_Сигналу* := *Початкове_Значення*;

де

Ідентифікатор – ім'я сигналу, яке сформоване за правилами формування ідентифікаторів;

Тип_Сигналу – може задаватися будь-яким типом чи підтипом;

Початкове_Значення – значення сигналу в момент часу 0 с. Якщо початкове значення не задається явно, то в такому випадку сигнал набуває крайнього лівого значення підтипу *Тип_Сигналу*.

Кілька однотипних сигналів з однаковим початковим значенням можуть об'являтися одним оператором **signal**. У цьому випадку їх ідентифікатори перераховуються через кому, наприклад:

signal *s1, s2, s3*: *Std_Logic* := '1';

Оператори об'явлення сигналів можуть розміщуватися в розділі оголошень архітектури: тоді кожний з об'явлених сигналів буде доступний всім процесам поточної архітектури. Крім того, сигнал можна об'явити в розділі оголошень **entity**: в цьому випадку сигнал буде доступний всім введеним архітектурам даного об'єкту.

4.4. Оператор направлення сигналу та способи реалізації затримки в VHDL

4.4.1. Оператор направлення сигналу

Для зміни значення сигналу використовується оператор направлення, його синтаксис має вигляд:

```
Target_Signal <= Source after delay;
```

де

Target_Signal – ідентифікатор сигналу, значення якого має змінитися;
Source – джерело нового значення (може застосовуватися змінна, константа, вираз або сигнал);
delay – часова затримка.

Мова VHDL є спеціалізованою мовою для описання та моделювання пристроїв, реальні сигнали в яких не можуть змінювати свого значення миттєво. Для врахування цієї особливості в VHDL застосовується механізм транзакцій.

Транзакцією називається відкладена дія призначення сигналу. Кожна транзакція містить ім'я сигналу, що буде змінюватись, нове значення сигналу та значення модельного часу, в якому буде проведено зміну значення сигналу.

Отже, якщо при моделюванні об'єкта зустрічається оператор направлення сигналу, то його значення не змінюється. Замість цього формується транзакція, в яку заноситься ім'я сигналу (*Target_Signal*), нове значення (*Source*) та значення модельного часу зміни сигналу, що формується як сума поточного значення модельного часу та часової затримки *delay* (у випадку, коли *delay* не задано явно, час затримки визначається кроком модельного часу). Сформована нова транзакція додається до списку транзакцій, який перевіряється системою на кожному кроці зростання модельного часу. Якщо ж настає момент зміни сигналу, то йому присвоюється нове значення. Зміна значення сигналу називається *подією* на сигналі. Такий алгоритм призводить до того, що сигнал не

змінює свого значення миттєво, і мінімальна часова затримка дорівнює кроку модельного часу.

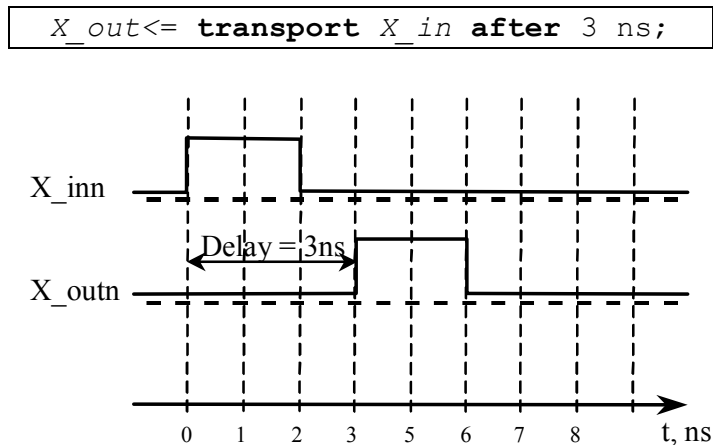
Для більш точного опису різноманітних умов передачі сигналів в реальних об'єктах в VHDL введено два типи часової затримки: транспортна та інерційна.

4.4.2. Транспортна затримка сигналу

Транспортна затримка сигналу (ТЗ) в загальному випадку моделює просту затримку в передачі вхідного сигналу на чітко визначений час. В загальному вигляді транспортна затримка може бути описана в VHDL як

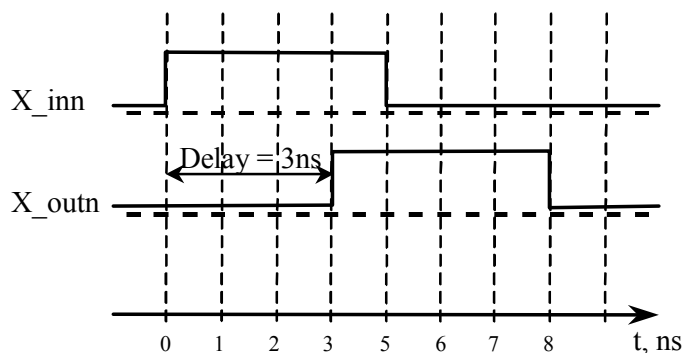
```
X_out <= transport X_in after time_value;
```

Приклад ілюструє вплив транспортної затримки на вихідний сигнал при різній тривалості вхідного сигналу.



a) Тривалість вхідного сигналу X_{in}
менше значення затримки 3 ns

```
X_out <= transport X_in after 3 ns;
```



б) Тривалість вхідного сигналу
 X_{in} більше або дорівнює значенню затримки 3 ns

Рис. 4.1 – Моделювання транспортної затримки сигналу

4.4.3. Інерційна затримка сигналу

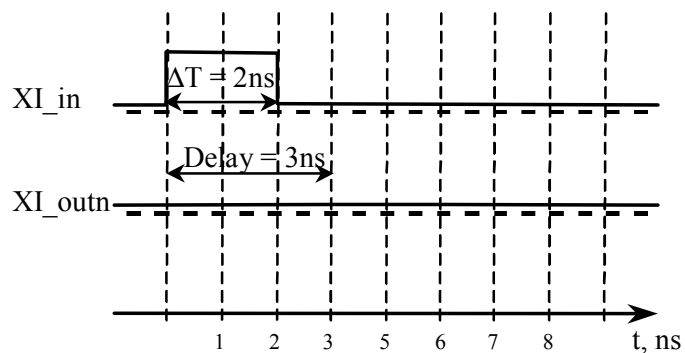
Інерційна затримка дозволяє моделювати роботу елементів схеми, які не пропускають короткі імпульси. Тобто, якщо будь-який з елементів схеми має інерційну затримку T (як, наприклад, логічні елементи **ТАК**, **АБО**, **НІ**), то вхідний сигнал буде затримуватися на час T , однак імпульси, тривалість яких менше T , взагалі передаватися не будуть.

Загальна форма запису сигналу з інерційною затримкою має вигляд:

```
XI_out <= inertial XI_in after time_value;
```

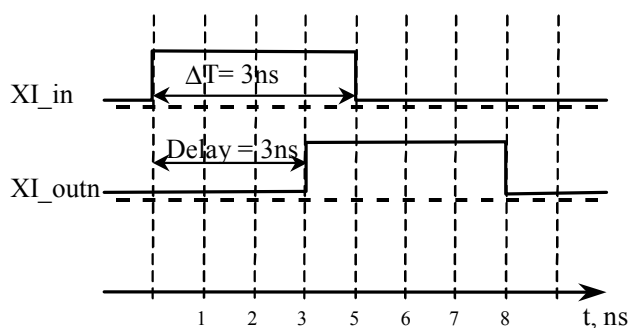
Наступний приклад демонструє роботу пристрою з інерційною затримкою при різній тривалості вхідного сигналу:

```
XI_out <= inertial XI_in after 3 ns;
```

а) тривалість вхідного сигналу менше значення затримки

```
XI_out <= inertial XI_in after 3 ns;
```



б) тривалість вхідного сигналу більше або дорівнює значенню затримки

Рис. 4.2 – Моделювання інерційної затримки сигналу

Відзначимо, що часто в реальних електронних пристроях мінімальна тривалість імпульсів, що повинні пропускатися, менша за значення інерційної затримки. У цих випадках форма запису даного типу затримки зміниться таким чином:

```
X out <= reject impuls length inertial X val after time value;
```

Приклад:

```
XI_out <= reject 1 ns inertial XI_in after 3 ns;
```

Якщо тип затримки не вказується явно, то автоматично "по замовчуванню" вибирається інерційний тип затримка сигналу.

4.4.4 Аналогії та відмінності між сигналами та змінними

Порівнюючи сигнали із змінними можна помітити, що головна відмінність між ними це час зміни значення. На відміну від сигналів, змінна набуває нового значення миттєво. З *прикладу* добре видно різницю, між результатами виконання одних і тих же операцій над змінними (а) і над сигналами (б):

а) операції над змінними

```
architecture var of EntA is
  signal InS, Sum: integer :=0;
begin
  process (InS)
    variable var1, var3: integer :=0;
    variable var2      : integer :=1;
    begin
      var1:=var2+1;
      var2:=var1-2;
      var3:=var1+var2;
      Sum<=var1+var2+var3;
    end process;
end var;
```

Результат виконання процесу:

Sum = 4

б) операції над сигналами

```

architecture var of EntA is
    signal InS, Sum: integer :=0;
    signal var1, var3: integer :=0;
    signal var2      : integer :=1;
begin
    process (InS)
        begin
            var1 <= var2+1;
            var2 <= var1-2;
            var3 <= var1+var2;
            Sum <= var1+var2+var3;           -- (*)
        end process;
    end var;

```

Результат виконання процесу:

Sum = 1

Відмінності в результатах пояснюються тим, що сигнали *var1*, *var2*, *var3* в прикладі (б) не змінюють свого значення на тому ж циклі моделювання, на якому виконується процес. Таким чином при виконанні рядка програми, позначеного знаком (*), перелічені сигнали ще мають старі значення, і саме на їх основі обчислюється результат виразу правої частини оператора призначення сигналу.

4.5. Атрибути сигналів

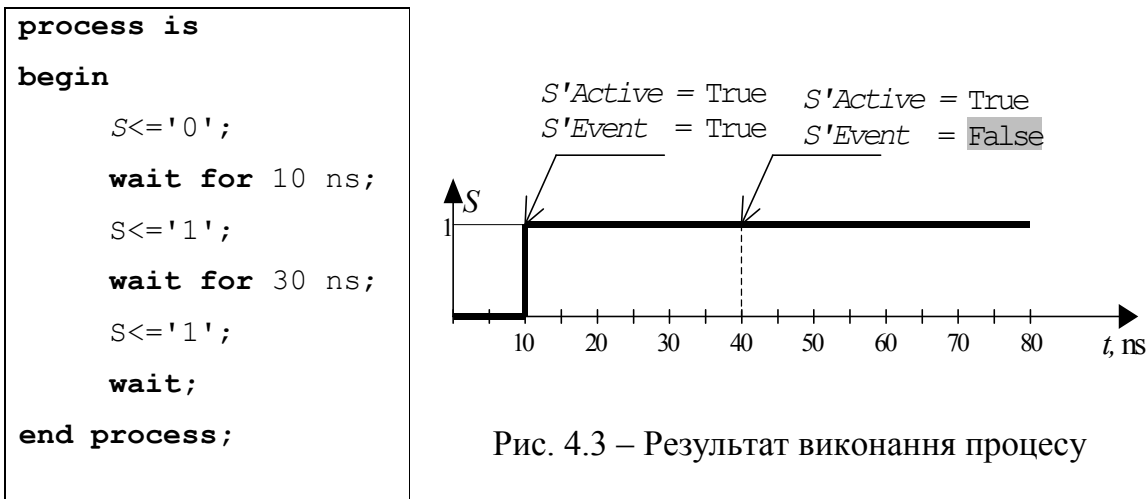
Сигнали в VHDL - це гнучкий і потужний засіб для моделювання реальних каналів обміну інформацією як всередині електронної системи, так і між електронною системою та зовнішнім середовищем. За допомогою *атрибутів* можна будувати програми, що реагують не лише на поточні значення сигналів, а й на “історію” їх подій і транзакцій.

Нехай визначено сигнал S і вираз T типу *time*. В таблиці наведено повний перелік атрибутів сигналів з описом їх функцій:

Ідентифікатор	Функція
$S'_{delayed}(T)$	Приймає таке ж значення як і S , але з затримкою на час T . Може застосовуватися для визначення попередніх значень сигналів
$S'_{stable}(T)$	Повертає булівське значення, що дорівнює <i>True</i> , якщо на сигналі S не відбувалося <i>подій</i> (змін значення) в продовж проміжку часу T до поточного значення модельного часу
$S'_{quiet}(T)$	Повертає булівське значення, що дорівнює <i>True</i> , якщо на сигналі S не виникало <i>транзакцій</i> за проміжок часу T перед поточним значенням модельного часу
$S'_{transaction}$	Сигнал типу <i>bit</i> , що змінює своє значення з '0' на '1' (і навпаки) кожного разу, коли на сигналі S з'являється транзакція
S'_{event}	Булівський сигнал, що дорівнює <i>True</i> , якщо на сигналі S в поточній ітерації моделювання відбувалася подія, і дорівнює <i>False</i> в іншому випадку
S'_{active}	Сигнал дорівнює <i>True</i> , якщо на поточній ітерації моделювання спрацювала транзакція для сигналу S
S'_{last_event}	Повертає значення типу <i>time</i> – проміжок часу до останньої події на сигналі S
S'_{last_active}	Повертає значення типу <i>time</i> – проміжок часу до останньої транзакції на сигналі S
S'_{last_value}	Повертає останнє перед поточним значення сигналу S

Слід звернути увагу на відмінності між застосуванням атрибутів *'Event* та *'Active*. Перший з них приймає значення *True* в тих циклах моделювання, де на відповідному сигналі відбулася подія (змінилося значення сигналу), а другий – в тих циклах коли відбулася транзакція.

На *прикладі* проілюстровано цю відмінність:



Тут в момент часу $t = 10$ ns значення сигналу S змінилось з '0' на '1'. Відповідно, на цьому циклі моделювання спрацювала транзакція ($S'Active = True$) і відбулася подія ($S'Event = True$). В момент часу $t = 40$ ns транзакція спрацювала ($S'Active = True$). Разом з тим, оскільки в момент часу $t = 40$ ns фактичної зміни значення сигналу S не відбулося (значення змінилось з '1' на '1'), то події на сигналі не було і, відповідно, $S'Event = False$.

Не дивлячись на те, що при апаратній реалізації VHDL-проектів застосування більшості атрибутів сигналів є ускладненим (так, наприклад, було б дуже важко визначити останнє попереднє значення напруги на провіднику чи час її зміни), використання інформації атрибутів є дуже корисним на етапі комп'ютерного моделювання VHDL-проекту, що розробляється або вже є розробленим. За допомогою механізму атрибутів можна більш детально аналізувати внутрішні процеси, що протікають в

електронних пристроях, і, відповідно, виявляти значну кількість проектних помилок та відхилень.

Область застосування атрибутів не обмежується лише задачами моделювання та тестування VHDL-проектів. Атрибути *'Event* та *'Active* підтримуються засобами синтезу ПЛІС, однак в цьому випадку їх можна використовувати лише в операторах **wait** та **if** для визначення подій на сигналах тактової частоти з метою синхронізації окремих частин проекту.

4.6. Процес

Одним з ключових елементів мови VHDL є процес (**process**). Процес є конкурентним оператором, тому поміщається в розділ *Concurrent_Statements* архітектури.

Синтаксис процесу має вигляд:

```
Мітка_процесу: process (Список_Чутливості) is
    Розділ_Оголошень;
    begin
        Послідовні_Оператори;
    end process Мітка_процесу;
```

В наведеному прикладі позначено:

Мітка_Процесу – сформований за правилами ідентифікатор процесу;
Розділ_Оголошень – розділ, в якому можна об'являти локальні (доступні лише в поточному процесі) константи, змінні, підтипи, а також підпрограми (процедури та функції);

Список_Чутливості – список сигналів, при виникненні подій на яких (зміна сигналу) здійсниться запуск процесу. Список чутливості може бути відсутнім, але в цьому випадку обов'язкова присутність у процесі хоча б одного оператора **wait** (розглядається нижче);

Послідовні_Оператори – оператори, що виконуються (на відміну від конкурентних) послідовно один за одним. До послідовних операторів відносяться *умовний оператор, оператор вибору, циклічні оператори, оператор присвоювання, оператор пересилання значення сигналу* та інші. Для аналогії зазначимо, що всі оператори таких мов як Pascal та C++ є послідовними.

Розглянемо роботу процесів на прикладі RS – тригера:

```
entity RSTrigger is
    port( S, R: in Std_Logic;
          D, Inv_D: out Std_Logic;
end entity RSTrigger;

architecture RSTrigger of RSTrigger is
begin
    Flip_Flop:    process (R,S) is
        begin
            if S = '1' and R = '0' then
                D<='1';
                Inv_D<='0';
            elseif S = '0' and R = '1' then
                D<='0';
                Inv_D<='1';
            else
                D<='Z';
                Inv_D<='Z';
            end if;
        end process Flip_Flop;
    end architecture RSTrigger;
```

Робота цього процесу протікає таким чином: процес очікує появи події (зміни сигналу) на сигналах *R* чи *S*. Після цього перевіряються

значення вхідних сигналів: якщо $R = '0'$, $S = '1'$, то на вихід тригера D подається сигнал логічної $'1'$, а на інверсний вихід - сигнал логічного $'0'$, якщо ж $R = '1'$, $S = '0'$, то на вихідний та на інверсний вихідний сигнали відповідно подаються логічні $'0'$ та $'1'$. У всіх інших випадках вихідний сигнал RS-тригера є невизначеним, тому на його виходи подається сигнал $'Z'$ (високий імпеданс).

4.7. Оператор Wait

Аналіз основних класів електронних пристроїв показує, що вони, як правило, мають безперервний цикл роботи. Тобто після ініціалізації вони виконують визначені для них операції, а потім переходять в режим очікування відповідної *події* нового запуску. Іншими словами пристрої припиняють свою роботу після завершення поточних задач і поновлюють її знову, як тільки виникає необхідна *подія*. Дана властивість систем описується в VHDL за допомогою службового слова **wait**, яке визначає не тільки момент розриву послідовності операцій, але і необхідні умови їх поновлення. Аналогічні функції виконує *список чутливості* процесу, тому один і той же процес не може містити одночасно *список чутливості* і оператор **wait**. При цьому процес, в який входить вираз, що містить **wait**, має вигляд:

```
process is
begin
    sequential_statement ;
    wait_statement;
    sequential_statement;
    wait_statement;
    ...
end process;
```


Даний процес виконує послідовність дій *sequential_statement* доти, доки не зустрине оператор **wait** (*wait_statement*), який припиняє виконання процесу до виникнення події, що зазначена в параметрах оператора **wait**.

Існує три основних і один додатковий тип оператора **wait**:

Основний оператор 1.

wait on *sensitivity_list* – очікування здійснюється доти, доки не зміниться який-небудь з сигналів в списку чутливості (*sensitivity_list*).

Приклад:

```
wait on CLK;
wait on Ain, Bout;
```

Основний оператор 2.

wait for *time_expression* – очікування здійснюється поки не мине час, що вказаний в *time_expression*.

Приклад:

```
wait for 5 ns;
wait for CLKperiod/2;
```

Основний оператор 3.

wait until *boolean_expression* – очікування виконання умови, що вказана в *boolean_expression*.

Приклад:

```
wait until CLK='1';
wait until IntData>15;
```

Додатковий тип можна визначити як *комбінований тип* запису припинення процесу, який може складатися із двох або трьох різних форм оператора **wait**.

Приклад:

```
wait on A until CLK='1';
```

```
wait until CLK='1' for 10 ns;
```

Слід відзначити, що окрім наведених вище типів запису оператора **wait**, оператор **wait** може використовуватись і без параметрів, наприклад:

```
process is  
begin  
    list_of_statements;  
    wait;  
end process;
```

В останньому випадку процес почне свою роботу відразу після початку моделювання і зупиниться, коли будуть виконані всі операції з *list_of_statements*. Такий оператор **wait** без параметрів корисний при створенні випробувальних стендів і використовується для формування задаючих або збурюючих впливів.

4.8. Моделювання в VHDL

Розглянемо детально процедуру моделювання об'єктів в VHDL. Моделювання поведінки об'єктів в ЕОМ передбачає розрахунок значень сигналів об'єкта (вхідних, вихідних та внутрішніх) через рівні (дискретні) проміжки часу. Такі проміжки часу називаються кроком або циклом моделювання. Зрозуміло, що адекватність результатів моделювання реальним процесам обернено-пропорційна величині циклу моделювання.

Нижче наведено основні етапи процедури моделювання об'єкта в VHDL (рис. 4.4):

1. Ініціалізація моделювання:
 - поточне значення *модельного часу* встановлюється в 0 (нуль);
 - всім сигналам об'єкта присвоюються початкові значення.
2. Запуск на виконання всіх процесів.

Процеси виконуються до їх повної зупинки, тобто для процесів із списком чутливості – до кінця (**end process**), або до оператора **wait** – в іншому випадку. Лише після зупинки всіх процесів система може перейти до наступного етапу.

3. *Модельний час* збільшується на *Цикл (крок) моделювання*.
4. Перевіряється список транзакцій: якщо є транзакції, що настроєні на поточний *Модельний час*, то відповідні сигнали набувають нових попередньо запрограмованих значень.
5. Перевіряються всі оператори **wait**, що задіяні в даний момент *Модельного часу*: якщо є оператори, для яких виконуються умови запуску, то запускаються відповідні процеси.
6. Якщо *Модельний час* не досягає значення *Кінець_моделювання*, то система повертається до п. 3.

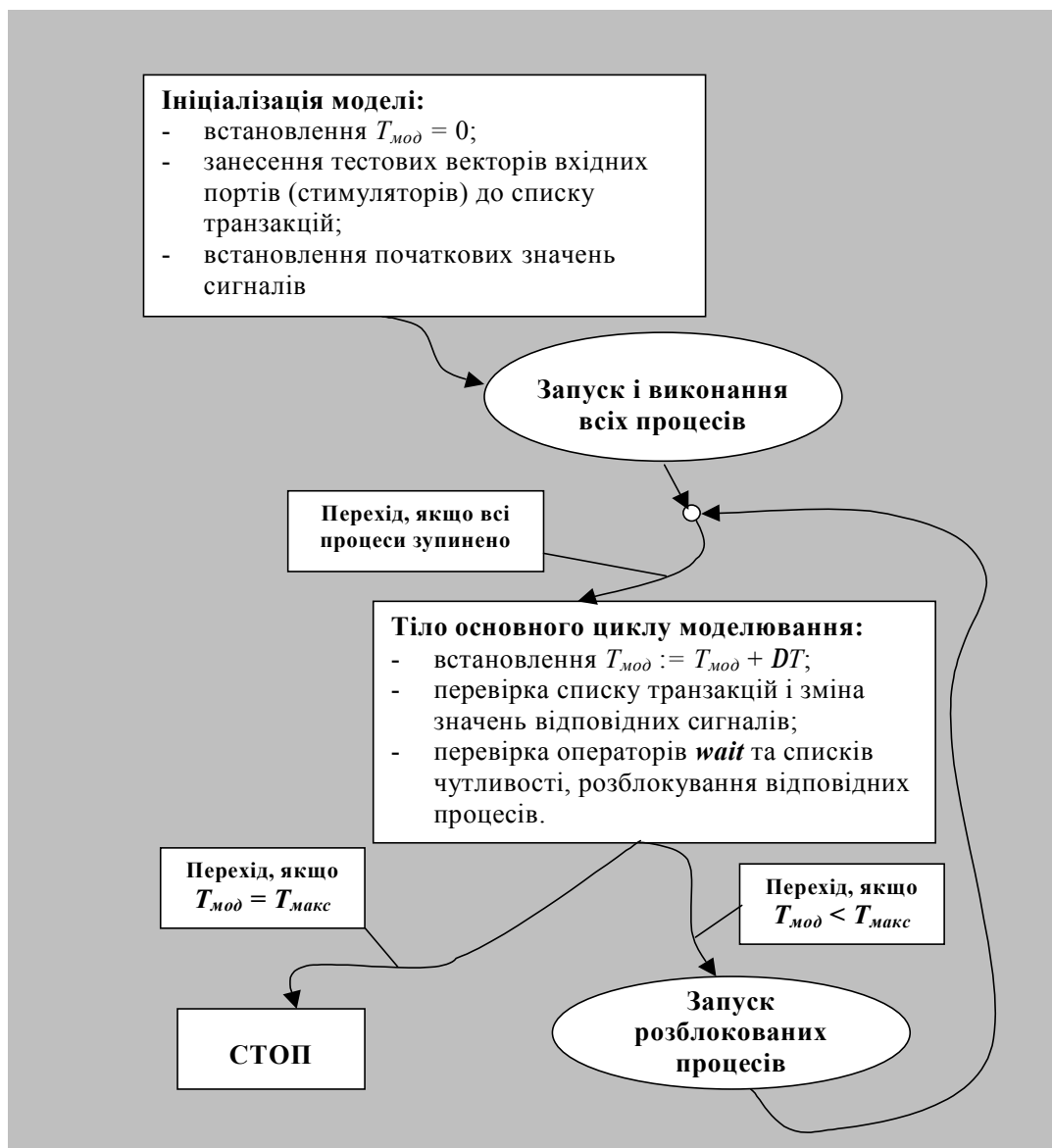


Рис. 4.4 – Процес імітаційного моделювання об'єктів в VHDL

4.9. Компонування складних об'єктів із структурних складових

При описанні складних об'єктів часто виникає необхідність в їх декомпозиції (розбитті на структурні складові). Крім того, часто буває корисною можливість повторного застосування типових елементів об'єкта (наприклад, таких як лічильники, суматори, мультиплексори тощо). Мова VHDL надає програмісту можливість ефективно розв'язувати вказані задачі за допомогою механізму компонентів.

Один з можливих шляхів компонування складного об'єкта з уже розроблених елементів – застосування *оператора включення*. Цей оператор являється конкурентним, отже розміщується в тілі архітектури (після слова **begin**).

Синтаксис *оператора включення* має вигляд:

```
Instance_Name:   entity work.Entity_Name
                  port map(S1=>A, S2=>B, S3=>C);
```

де

Instance_Name – ім'я включення;

work – ім'я бібліотеки (ім'я *work* вказує на поточний проект);

Entity_Name – ім'я об'єкта, що включається.

port map - після ключових слів **port map** в дужках необхідно вказати, які саме сигнали поточного об'єкта підключаються до входів та виходів об'єкта, що включається (в розглянутому прикладі *A,B,C* – сигнали складного об'єкта; *S1,S2,S3* – сигнали компонента).

Розглянемо приклад.

Необхідно спроектувати арифметико-логічний пристрій (АЛП), при цьому пристрої, що реалізують арифметичні операції, було розроблено попередньо. Опис інтерфейсів цих пристроїв наведено нижче, зокрема, для:

а) суматора

```
entity adder is                                     -- Суматор
    port(      a, b: in Std_Logic_Vector (7 downto 0);
            c: out Std_Logic_Vector (7 downto 0) := "00000000");
end entity adder;
```

б) помножувача

```
entity mul   is                                     -- Помножувач
```

```

    port(      a, b: in Std_Logic_Vector (7 downto 0);
           c: out Std_Logic_Vector (7 downto 0) := "00000000");
end entity mul;

```

в) поділювача

```

entity divider is                                -- Поділювач
    port(      a, b: in Std_Logic_Vector (7 downto 0);
           c: out Std_Logic_Vector (7 downto 0) := "00000000");
end entity divider;

```

VHDL-Програма, що реалізує арифметико-логічний пристрій на основі декомпозиційного підходу, наведена нижче:

```

entity ALU is
    port(X, Y: in Std_Logic_Vector (7 downto 0);
          operation: in Std_Logic_Vector (1 downto 0);
          en: in Std_Logic;
          Z: out Std_Logic_Vector (7 downto 0) := "00000000");
end entity ALU;

architecture ALU of ALU is
    signal S1, S2, S3: Std_Logic_Vector (7 downto 0);
begin
    Op1: entity work.Adder
        port map(a=>X, b=>Y, c=>S1);
    Op2: entity work.Mul
        port map(a=>X, b=>Y, c=>S2);
    Op3: entity work.Divider
        port map(a=>X, b=>Y, c=>S3);

    process (En) is
    begin
        case Op is
            when "00" => Z <= S1;

```

```

        when "01" => Z <= S2;
        when "10" => Z <= S3;
        when others => Z <= "ZZZZZZZZ";

    end case;

end process;

end architecture ALU;

```

На момент компіляції наведеної програми необхідно, щоб програми, які описують окремі елементи, були попередньо скомпільовані в окремих файлах і, якщо це необхідно, були підключені відповідні бібліотеки.

4.10. Константи **generic**

При компонуванні складних об'єктів часто виникає необхідність у зміні параметрів компонентів. Можливість такої зміни без перекомпілювання компоненти надає застосування констант **generic**. Ці константи розміщуються в розділі опису об'єкта **entity**. Синтаксис констант **generic** наведено нижче:

```

entity Entity_Name is
    generic (a,b,c: integer range 0 to 255; d: STD_LOGIC);
    port(      port_list      );
end Entity_Name;

```

Як видно з наведеного прикладу, константам описаним у розділі **generic** не присвоюється конкретне значення на етапі компіляції об'єкта, однак вирази, що містять ці константи, є локально-статично визначеними. Це дає можливість застосовувати константи **generic** для задавання діапазонів індексів масивів, підтипів тощо. Конкретне значення цим константам присвоюється при включенні об'єкта, як компонента, до складу складного об'єкта, *наприклад*:

```

entity Entity_Name is
    generic (a,b,c: integer range 0 to 255; d: STD_LOGIC);
    port      ( port_list_1      );
end Entity_Name;

entity Main_Entity is
    port(      port_list_2      );
end Main_Entity;

architecture MAIN_ENTITY of Main_Entity is
begin
    INST0: entity work.Entity_Name
        generic map(a=> 10, b=> 125, c=> 0, d => '1');
        port map( port_association_list );
    INST1: entity work.Entity_Name
        generic map(a=> 2, b=> 3, c=> 4, d => '0');
        port map( port_association_list );
    ...

end MAIN_ENTITY;

```

Розглянемо застосування **generic** на прикладі реалізації універсального паралельного регістра.

VHDL-Програма, що реалізує 8-бітний паралельний регістр, має вигляд:

```

entity Reg is
    port(
        X : in STD_LOGIC_VECTOR(7 downto 0);
        CLK: in STD_LOGIC;
        WE : in STD_LOGIC;
        Y: out STD_LOGIC_VECTOR(7 downto 0)
    );

```



```

end Reg;

architecture REG of Reg is
begin
process (CLK) is
variable U: STD_LOGIC_VECTOR (7 downto 0);
begin

    if (WE = '1' ) then
        U := X;
    end if;
    Y<=U;
end process;
end REG;

```

Щоб реалізувати універсальний регістр необхідно дати користувачу можливість змінювати розрядність регістра без створення нових **entity** та без перекомпіляції регістра. Здійснимо це за допомогою включення необхідної константи:

```

entity Reg is
    Generic (N: integer range 1 to 63);
    port(
        X : in STD_LOGIC_VECTOR(N downto 0);
        CLK: in STD_LOGIC;
        WE : in STD_LOGIC;
        Y: out STD_LOGIC_VECTOR(N downto 0)
    );
end Reg;

architecture REG of Reg is
begin
process (CLK) is

```

```

variable U: STD_LOGIC_VECTOR (N downto 0);
begin

    if (WE = '1' ) then
        U := X;
    end if;
    Y<=U;
end process;
end REG;

```

Тепер для включення 8-бітного або 16-бітного регістра треба записати відповідний оператор включення компоненти:

а) 8-бітний регістр

```

REG8:    entity work.Reg
        generic map (N=>7);
        port map (X=>S1, CLK=>S2, WE=>S3, Y=>S4);

```

б) 16-бітний регістр

```

REG16:   entity work.Reg
        generic map (N=>15);
        port map (X=>S1, CLK=>S2, WE=>S3, Y=>S4);

```

4.11. Компоненти з попередньо описаним інтерфейсом

Опис *компоненти* представляє собою визначення інтерфейсу елемента системи, включаючи його статичні (константи **generic**) і динамічні (порти) складові. Механізм компонент дозволяє зробити архітектуру повністю незалежною і закінченою в межах даного

ієрархічного рівня. Тобто, попередній опис інтерфейсу компонентів дає змогу запускати процес аналізу проекту, що не містить архітектур – реалізацій всіх включених об'єктів **entity**. В результаті відкриваються широкі можливості для розподілу великого проекту між окремими виконавцями.

Синтаксис *оператора оголошення компоненти* наведено нижче:

```
component Identifier is
    generic (список_констант_generic);
    port (список_портів);
end component Identifier;
```

Даний оператор розміщується в розділах оголошень архітектури або в пакетах (**package**). В першому випадку компонента доступна в межах даної архітектури, у другому випадку – в межах дії оператора підключення пакету.

Для застосування описаної компоненти застосовується конкурентний оператор включення компоненти:

```
мітка_включення:    component Identifier
                    generic map (список_асоціювання_generic)
                    port map (список_асоціювання_портів);
```

Для прикладу розглянемо реалізацію 4-х бітного регістра як комбінацію чотирьох простих тригерів.

Інтерфейс регістра можна записати наступним чином:

```
entity reg4 is
    port (    clk, clr: in STD_LOGIC;
            d: in STD_LOGIC_VECTOR (0 to 3);
            q: out STD_LOGIC_VECTOR (0 to 3) );
end entity reg4;
```

Програмна реалізація архітектури регістра має вигляд:

```

architecture struct of reg4 is
    component FlipFlop is
        generic ( Tprop, Tsetup, Thold: delay_length );
        port (      clk: in STD_LOGIC;
                   clr: in STD_LOGIC;
                   d: in STD_LOGIC;
                   q: out STD_LOGIC );
    end component FlipFlop;

begin
    bit0: component FlipFlop is
        generic map (Tprop=>2 ns, Tsetup=>2 ns, Thold=>1 ns )
        port map ( clk=>clk, clr=>clr, d=>d(0), q=>q(0) );

    bit1: component FlipFlop is
        generic map (Tprop=>2 ns, Tsetup=>2 ns, Thold=>1 ns )
        port map ( clk=>clk, clr=>clr, d=>d(1), q=>q(1) );

    bit2: component FlipFlop is
        generic map (Tprop=>2 ns, Tsetup=>2 ns, Thold=>1 ns )
        port map ( clk=>clk, clr=>clr, d=>d(2), q=>q(2) );

    bit1: component FlipFlop is
        generic map (Tprop=>2 ns, Tsetup=>2 ns, Thold=>1 ns )
        port map ( clk=>clk, clr=>clr, d=>d(3), q=>q(3) );

end architecture struct;

```

Таким чином, наведений приклад ілюструє механізм реалізації алгоритму функціонування регістра на основі тригерів без реалізації самого тригера.

На відповідному етапі розробки проекту в будь-якому випадку слід описати, включення якого з об'єктів *entity* здійснює компонента, а також вказати архітектуру, що реалізує даний *entity*. Задача асоціювання компоненти з конкретним об'єктом *entity* розв'язується оператором *configuration*.

Синтаксис оператора *configuration* наведено нижче:

```
configuration Identifier of Ім'я_entity is
  for Ім'я_Архітектури
    for Ім'я_компонента
      use entity Ім'я_асоційованого_entity1 (Архітектура1);
      use entity Ім'я_асоційованого_entity2 (Архітектура2);
    end for;
  end for;
end configuration Identifier;
```

Запишемо оператор конфігурації для регістра, розглянутого у попередньому прикладі. При цьому вважаємо, що тригер реалізовано в пакеті *edge_triggered_Dff* бібліотеки *star_lib*:

```
library star_lib;                -- Підключення бібліотеки
use star_lib.edge_trigg_Dff;    -- Включення відповідного пакету

configuration reg4_gate_level of reg4 is
  for struct      -- Вказано архітектуру об'єкта reg4
    for bit0: FlipFlop
      use entity edge_triggered_Dff (hi_fanout);
    end for;
  for others : FlipFlop
```

```

        use entity edge_triggered_Dff (basic);
    end for;
end for;
end configuration reg4_gate_level;

```

Як видно з даного прикладу, за допомогою операторів *component/configuration* можна призначати для різних включень однієї і тієї ж компоненти різні реалізації архітектур (для включення *bit0* призначено архітектуру *hi_fanout*, а для включень *bit1*, *bit2*, *bit3* – архітектуру *basic*). Отже застосування механізму компонент підвищує рівень абстракції окремих ієрархічних рівнів проекту шляхом “маскування” деталей їх технічної реалізації, що в результаті робить процедуру розуміння проекту більш простою.

4.12. Паралельна робота процесів

Головною особливістю *процесів* є їх здатність працювати паралельно. Тобто, якщо в середині архітектури міститься два або більше *процесів*, *списки чутливості* яких містять один і той же *сигнал*, то при зміні значення цього *сигналу* всі *процеси* запускаються одночасно. При цьому дії всередині кожного з них виконуються послідовно і незалежно від інших паралельних *процесів*.

Для наочності розглянемо приклад:

```

architecture SomeArch of SomeEnt is
    signal D, E : bit;
begin
    P1: process (A, B, E)
    begin
        some_statement;
        some_statement;
        D<=some_expression;
    end process;
end architecture;

```

```

end process P1;
P2: process (A, C)
begin
    some_statement;
    some_statement;
    E <= some_expression;
end process P2;
P3: process (D, E)
begin
    some_statement;
    some_statement;
end process P3;
end architecture SomeArch

```

Послідовність виконання процесів даного прикладу наведено на рис. 4.5. Якщо змінити значення сигналу *A* (змінені сигнали будемо позначати індексом (*)), два процеси *P1* і *P2* одночасно почнуть свою роботу. Внаслідок їх виконання зміняться значення сигналів *D* і *E*, які, в свою чергу, входять до списків чутливості процесів *P3* і *P1*. В результаті роботи процесу *P1* змінюється сигнал *D*, що призводить до нової ініціалізації процесу *P3*. Після виконання процесу *P3* жодному з сигналів в розглянутих списках чутливості не присвоюється нове значення, тобто всі процеси припиняються в очікуванні чергової зміни одного з сигналів із списків чутливості.

Слід відзначити важливий момент: якщо внаслідок виконання процесу *P1* сигналу *D* присвоюється таке ж значення, яке у нього було до ініціалізації процесу *P1*, то в цьому випадку процес *P3*, в списку чутливості якого знаходиться сигнал *D*, не поновлюється.

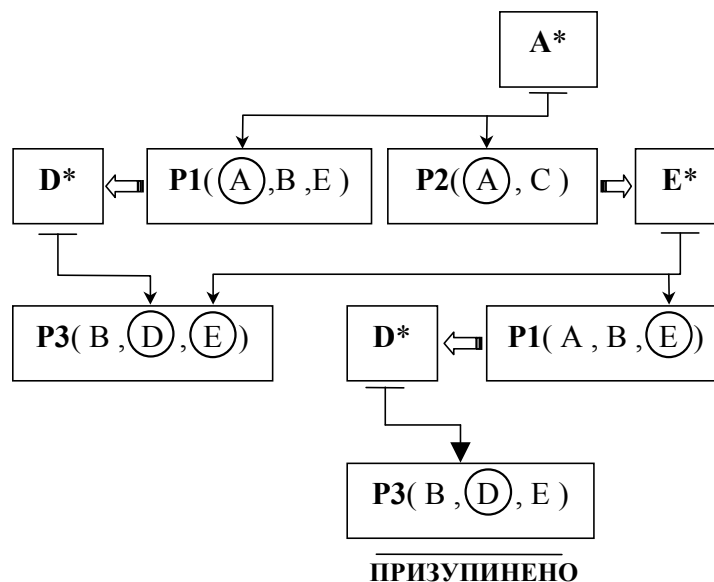


Рис. 4.5 – Паралельна робота процесів

4.13. Перекриття сигналів

Коли мова йде про паралельну роботу *процесів*, звичайно виникає питання: що станеться з сигналом, якщо внаслідок роботи двох або більше процесів, йому будуть одночасно присвоєні різні значення? Іншими словами, стоїть задача визначення сигналу, який має два або більше джерел. Для цього в VHDL існує так званий метод перекриття сигналів, принцип якого полягає в “змішенні” сигналів різного рівня відповідно до функції перекриття. *Функція перекриття* задається при описанні підтипу і визначає закон формування результуючого значення сигналу для всіх можливих комбінацій сигналів джерел.

Для моделювання реальних багаторівневих сигналів цифрових пристроїв в стандартному VHDL створено тип *std_logic*. Сигнал такого типу може набувати одного з дев’яти рівнів, а саме:

- ‘U’ – невизначений;
- ‘X’ – сильний невідомий (або ‘0’, або ‘1’);

- ‘0’ – сильний нуль;
- ‘1’ – сильна одиниця;
- ‘Z’ – високий імпеданс;
- ‘W’ – слабкий невідомий;
- ‘L’ – слабкий нуль;
- ‘H’ – слабка одиниця;
- ‘-’ – байдужий стан.

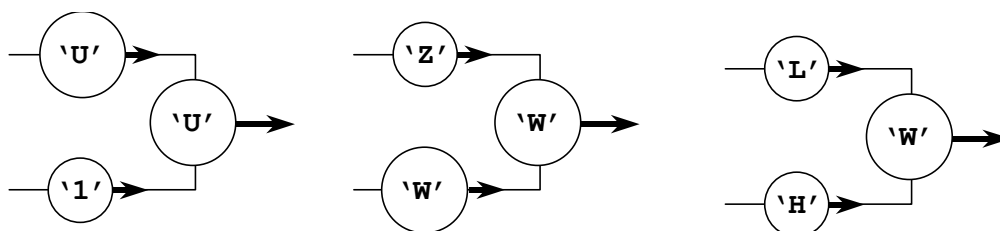
Відмінності між “сильними” та “слабкими” логічними значеннями полягають у способі їх накладання. Якщо перекриваються між собою значення ‘H’ і ‘0’, то результатом функції перекриття буде ‘0’. Відповідно, перекриття ‘1’ і ‘L’ дасть ‘1’. У більшості випадків для моделювання реальних цифрових пристроїв та їх реалізації на основі ПЛІС достатньо значень ‘0’, ‘1’, ‘Z’ та ‘X’. Значення ‘X’ та ‘W’ з’являться при моделюванні у тих випадках, коли значення сигналу неможливо обчислити.

Типи `Std_Logic` та `Std_Logic_Vector` (масив елементів типу `Std_Logic`) описано у пакеті `Std_Logic_1164` бібліотеки `IEEE`, що обумовлює необхідність звернення до нього перед початком опису об’єкта, який буде використовувати такі типи.

Синтаксис звернення має вигляд:

```
Library IEEE;
Use IEEE.Std_Logic_1164.all;
```

Принцип перекриття таких сигналів представлено на рис. 4.6:

Рис. 4.6 – Перекриття сигналів типу *Std_Logic*

Повний перелік усіх можливих комбінацій наведено у таблиці 4.1.

Таблиця 4.1 – Перекриття сигналів типу *Std_Logic*

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

Необхідно також зазначити, що серед стандартних типів сигналів операція перекриття у VHDL визначена тільки для типів *Std_Logic* та *Std_Logic_Vector*. Якщо це необхідно, то VHDL дозволяє користувачеві сформувати свою *функцію перекриття* для будь-яких створених ним типів сигналів. Для описання реальних логічних сигналів часто застосовуються також типи *Std_ULogic* та *Std_ULogic_Vector*, які є підтипами *Std_Logic* та *Std_Logic_Vector* без функції перекриття.

5. ДОДАТКОВІ МОЖЛИВОСТІ САПР ACTIVE-HDL

5.1. Діаграми скінчених автоматів в VHDL

Однією з форм представлення проекту в VHDL є діаграми скінчених автоматів, які дозволяють наочно представити роботу пристрою, що проектується, в формі ланцюга кіл (скінчених станів об'єкта) і стрілок (переходів між станами), змодельовати його роботу і згенерувати код VHDL.

Скінчена динамічна система називається *скінченим автоматом*, якщо стан системи на кожному такті однозначно визначається:

- а) станом системи на попередньому такті;
- б) умовами переходу на поточний такт або на попередній такт.

Таким чином, за допомогою діаграм скінчених автоматів можуть бути описані будь-які послідовні, синхронізовані у часі процеси.

В САПР Active-HDL формування і редагування діаграм проводиться у вікні *Редактора Скінчених Автоматів* (FSM Editor), яке може бути створене командою меню File\New\State_Diagram. На головній інструментальній панелі Редактора Скінчених Автоматів, крім загальноприйнятих управляючих кнопок для редагування (копія, вставка, збільшення і т.п.), додатково знаходяться наступні кнопки:



– генерує код VHDL для поточної діаграми;





– відображає згенерований для поточної діаграми код;




– переводить модель в наступний стан (застосовується при моделюванні).


На додатковій інструментальній панелі (ДІП) знаходяться кнопки, які дозволяють швидко розташовувати компоненти діаграми скінчених автоматів в робочій області:


 – Додає *стан* в діаграму. (*Стан* є основним компонентом діаграми скінчених автоматів і може набувати значень “TRUE” або “FALSE”, тобто бути або активним або пасивним).


 – Визначає *вхідні дії стану*. Дії, що визначаються, як вхідні виконуються тільки один раз – коли автомат переходить до цього стану.

 – Визначає *дію стану*. В цьому разі сигнал залишається активним протягом перебування автомату в цьому стані.

 – Визначає вихідну дію стану.

 – Малює *переходи* між станами. (*Переходи* визначають зміну одного стану автомата на інший відповідно до синхронізуючого імпульсу. Звичайно зміна станів відбувається при виконанні відповідних *умов переходу*, що записуються у вигляді Булевих виразів або **@lесе**. Запис **@lесе** набуває значення «TRUE» якщо всі інші умови початкового стану мають значення “FALSE”. Якщо дві або більше умов переходу від початкового стану набувають значення “TRUE”, то пріоритетний напрям переходу визначається по заздалегідь встановлених пріоритетах).

 – Визначає умову *переходу*.

 – Визначає *дію*, що відбувається при *переході*. (Під *дією* розуміють операцію (чи послідовність операцій) присвоєння нових значень портам, внутрішнім сигналам і змінним).


 – Дозволяє визначити дію самої діаграми.

 – Додає локальний сигнал або змінну до діаграми.

 – Додає вхідний порт.

 – Додає вихідний порт.

 – Додає джерело (задаючий вплив та синхронізуючий імпульс).

 – Додає довільний текст в будь-яку точку діаграми (коментарій).

 – Додає криву Бізье (коментарій).



– Малює коло (коментарій).

При формуванні діаграм скінчених автоматів потрібно враховувати, що наявність синхронізуючого імпульсу і задаючого впливу обов'язкова.

На Рис. 5.1 наведено приклад реалізації простого логічного блоку за допомогою редактора скінчених автоматів. Даний приклад дозволяє проілюструвати відповідність між станами та діями скінчених автоматів та VHDL-кодом (стрілки показують елементи діаграми, що реалізуються відповідними фрагментами коду). Наведена програма мовою VHDL автоматично згенерована системою Active-HDL на основі діаграми скінчених автоматів.

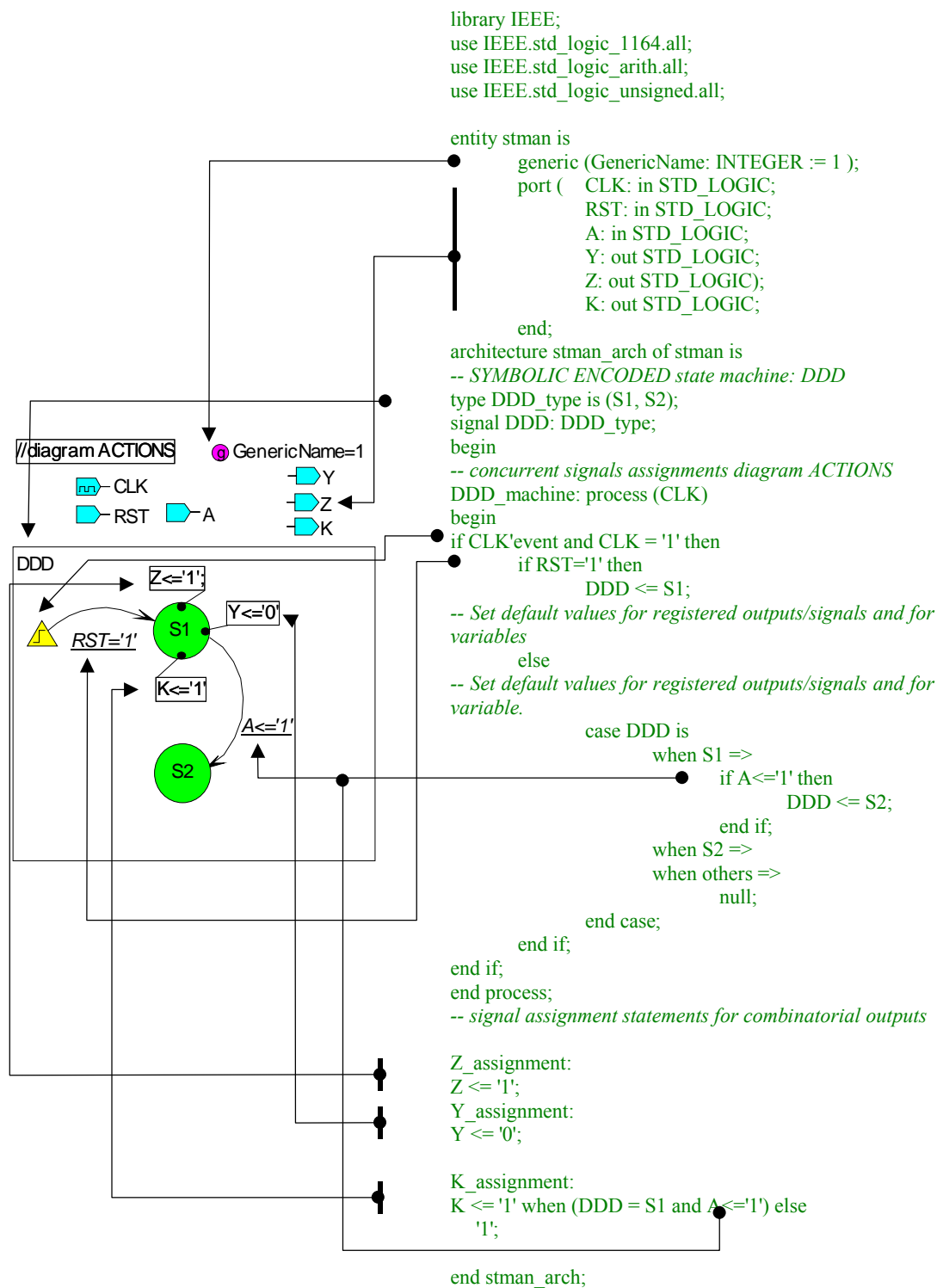


Рис. 5.1 – Відповідність між діаграмами скінчених автоматів
та VHDL-кодом

5.2. Випробувальні стенди

Очевидно, що процес розробки будь-якого проекту буде неповним без тестування. Існує багато різних способів виконання перевірки, однак найбільш популярним залишається застосування віртуального випробувального стенду (ВС). Невід'ємними складовими будь-якого ВС є:

- гніздо для пристрою, що тестується (UUT – Unit Under Testing);
- генератор задаючих сигналів;
- засоби для моніторингу режимів UUT;

Під *віртуальним випробувальним стендом* у VHDL розуміють об'єкт без інтерфейсу, в який UUT вводиться за допомогою оператора включення компоненти. Завдання ВС полягає в формуванні задаючих впливів для вхідних портів UUT. Результати тестування можна спостерігати за допомогою стандартних засобів моделювання Active-HDL.

Варіант структури коду VHDL для формування віртуального випробувального стенда показаний на прикладі.

```
entity TB is      -- Об'єкт без інтерфейсу - випробувальний стенд
end entity TB;

architecture TBArch of TB is
signal A,B . . .: bit;
signal          : bit;
begin
    UUT:entity work.MyProcessor (Beh) --Оператор включення UUT
        port map (. . .);

    Stimuli: process is      -- Процес - генератор тестової
        begin                -- послідовності
            A <= . . . ;
```

```

        B <= . . . ;           -- стимулятори
        . . .
        wait for . . . ;
        . . .
        wait for . . . ;
        . . . ;
        wait;
    end process stimuli;
end architecture TBArch;

```

Випробувальні стенди в Active-HDL можуть бути сформовані безпосередньо програмістом (вручну) і автоматично – за допомогою майстра Test Bench Wizard. Вказаний майстер є потужним засобом для створення випробувальних стендів і, крім формування структури ВС, дозволяє також під'єднувати джерела тестових векторів (часових діаграм для входних портів UUT). Зокрема, Active-HDL підтримує автоматичне генерування ВС, що відповідають індустріальному стандарту WAVES. Базовою для даної реалізації є draft-версія стандарту, розробка якої датована травнем 1997 р. (IEEE P1029.1/D1.0 May 1997). Стандарт WAVES (Waveform And Vector Exchange to Support Design and Test Verification – Обмін часовими діаграмами та векторами для підтримки тестової верифікації проектів) визначає формальний запис процесів верифікації і тестування проектів цифрових пристроїв.

Випробувальні стенди в Active-HDL можуть зчитувати тестові послідовності з:

- файлів типу (*.awf) , створених власним редактором часових діаграм (waveform-editor);
- файлів стандарту WAVES (*.vec);
- файлів – джерел коду, що генерує тестову послідовність, на мовах VHDL (*.vhs) або Verilog (*.ver).

В першому і третьому випадках формується процес, що генерує тестову послідовність, яка аналогічна послідовності в (*.awf) файлі, або безпосередньо містить фрагменти коду з файлів (*.vhs) чи (*.ver).

В другому випадку генерується паралельна процедура (як конкурентний оператор), що зчитує тестові послідовності з (*.vec)-файлів і направляє відповідні сигнали до портів об'єкта, що тестується. Прості стимулятори (типу *формула* чи *тактова частота*) одразу транслюються в процес – генератор тестових векторів. В цьому випадку зчитування (*.vec)-файлів під час моделювання не відбувається.

Для методу, що базується на стандарті WAVES, можливе порівняння виходів UUT з еталонними векторами, що зберігаються у відповідних файлах. Всі відмінності між результатами моделювання та еталонними результатами заносяться до log-файлу, а також виводяться в консоль.

Слід зауважити, що *.vec-файл може бути згенерований за допомогою waveform-editor середовища Active-HDL.

Приклад *.vec-файла:

```
% Begin Comment
%
% a : in std_logic
% b : in std_logic
% c : in std_logic
% d : in std_logic
% e : in std_logic_vector(3 downto 0)
%
% 50. ns : END_SIMULATION_TIME
% End Comment
%
% Begin Clocks
%
% a : '0'#10 ns%'1'#10 ns%
```

```
% c : '0'#13 ns%'1'#2 ns%
% End Clocks
%
% Begin of Vectors
%
%@ 1 1 1 1 1 1
0 0 XXXX : 17 ns ; % 0 ps
0 1 XXXX : 13 ns ; % 17 ns
0 1 1010 : 5 ns ; % 30 ns
1 1 1010 : 10 ns ; % 35 ns
%@ END_VECTORS
```

5.3. Виведення відлагоджувальної інформації. Оператор *assertion*

При моделюванні розроблених мовою VHDL пристроїв часто виникає необхідність в наданні оператору сигнальної інформації про виникнення помилок чи в передачі відлагоджувальної інформації. Найбільш зручним з можливих засобів для розв'язання даної проблеми є застосування оператора ***assert***, що виводить текстову інформацію до протоколу моделювання. В середовищі Active-HDL протокол ведеться в спеціальному файлі (в пакетному режимі моделювання). Крім того, при застосуванні інтерактивної графічної оболонки протокол відтворюється у вікні консолі, яке звичайно розміщується на екрані нижче від вікна редактора HDL-коду.

Синтаксис оператора ***assert*** має наступний вигляд:

```
мітка:      assert   булівський_вираз
            report текстове_повідомлення severity вираз;
```

Оператор **assert** являється послідовним, отже його можна розміщати в будь-якому місці процесів. При виконанні даного оператора перевіряється істинність *булівського виразу* і, якщо його значення дорівнює *False*, то в консоль або в файл протоколу виводиться повідомлення, текст якого поміщено в *рядковий вираз*. Тобто, оператор **assert** сліdkує за *виконанням* певної, попередньо запрограмованої, умови. В основному оператор **assert** застосовується для виявлення логічних помилок, а також для контролю даних, що надходять до **entity** і процесів. Підвищити ефективність застосування цього оператора можна шляхом визначення степені важливості помилки. У VHDL можна встановити один з чотирьох ранжованих рівнів важливості помилки – **note**, **warning**, **error** та **failure**. Звичайно цим рівням надають наступного змісту:

- note** – інформаційне повідомлення про негрубу помилку, яку можливо навіть проігнорувати;
- warning** – цей рівень звичайно застосовується для ідентифікації нештатних ситуацій, пов'язаних з надходженням неочікуваних динамічних даних (значень сигналів, змінних), після яких модель може продовжувати свою роботу, однак результати будуть непередбачуваними;
- error** – застосовується для ідентифікації помилок, що не дозволяють продовжувати моделювання роботи VHDL-програми, однак існує можливість поновити процес моделювання після певної коригуючої дії;
- failure** – такі повідомлення видаються при виникненні помилок, після яких продовження процесу моделювання є неможливим, тобто ці повідомлення є реакцією на такі комбінації даних, що ніколи не могли б виникнути при нормальній роботі програми.

Рівень помилки задається в операторі **assert** після ключового слова **severity**. Якщо рівень помилки не задається явно, то "по замовчуванню" приймається рівень **error**, у випадку, коли пропускається блок **report** – видається стандартне повідомлення *"Assertion violation"*.

Нижче наведено кілька прикладів застосування оператора **assert**.

В першому випадку представлено об'єкт, що з двох поданих на вхід цілих чисел вибирає більше число. Некоректною умовою, яку слід перевіряти в кожному порівнянні, є рівність двох чисел. В цьому випадку виникає "нестрогий максимум". В роботі такого порівняльного елементу це не являється помилкою і не повинно впливати на режими нормального функціонування інших елементів пристрою. Разом з тим, такі факти некоректності слід відмічати аналогічно наведеному нижче прикладу:

```
entity Max is
    port ( X1, X2: in integer range (0 to 65535);
          Y:   out integer range (0 to 65535));
end entity Max;

architecture Max_Behaviour of Max is
begin
    process (X1,X2) is
    begin
        assert X1 /= X2 -- умова обернена до умови виникнення похибки
        report "X1 is equal X2" severity note;
        if X1>X2 then
            Y<=X1;
        else
            Y<=X2;
        end if;
    end process;
end architecture Max_Behaviour;
```

Наступний приклад показує виникнення помилки типу **failure**.

Нехай пристрій управляє деякою буферною областю в оперативній пам'яті, причому поточний стан буфера визначається значеннями змінних *First_Position* (адреса початку буфера), *Last_Position* (адреса кінця буфера) та *Number_Of_Entries* (довжина буфера). Для цього випадку оператор

```
assert (Last_Position - First_Position + 1) = Number_Of_Entries
         report "inconsistency in buffer model"
         severity failure;
```

дозволяє відслідковувати виникнення неможливої ситуації при управлінні буфером.

Застосування оператора **assert** дає змогу зробити більш надійним процес тестування VHDL-моделей шляхом автоматизації контролю за станом сигналів і змінних об'єкта та за їх співвідношеннями. Крім того, оператор **assert** зручно застосовувати для порівняння результатів роботи поведінкових моделей та моделей інших рівнів (наприклад, рівня регістрових передач).

6. ЦИКЛ ЛАБОРАТОРНИХ РОБІТ З ФОРМУВАННЯ VHDL-МОДЕЛЕЙ ЦИФРОВИХ ПРИСТРОЇВ

ЛАБОРАТОРНА РОБОТА № 1

Вивчення інтегрованого середовища автоматизованого проектування Active-HDL

Мета роботи: ознайомитись з принципами автоматизованого проектування ПЛІС за допомогою Active-HDL. Вивчити структуру VHDL-проекту. Навчитися працювати з засобами управління проектом (*Майстер проекту, Вікно перегляду проекту*). Навчитися описувати об'єкти та їх інтерфейси за допомогою конструкції **entity**.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

Методологію формування VHDL-моделей будемо розглядати на основі середовища Active-HDL, що запропоновано проектантам цифрових пристроїв американською корпорацією Aldec Inc. - одним із світових лідерів в розробці систем автоматизованого проектування програмованих логічних інтегральних схем на базі мов описання апаратного забезпечення. Загальний вигляд екрана при роботі з САПР Active-HDL наведено на рис. Л1.1.

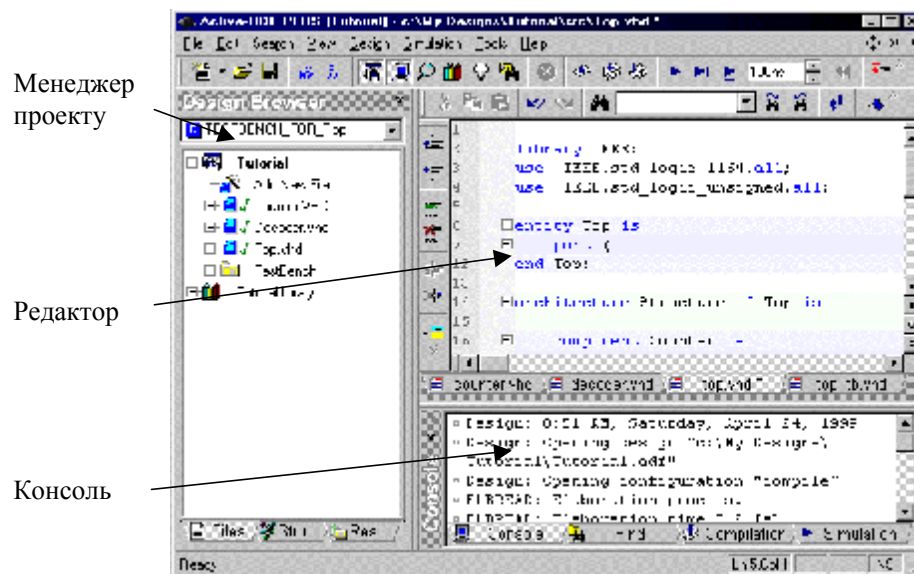


Рис. Л1.1 – Загальний вигляд екрана САПР Active-HDL

На рис Л1.1 показані також основні елементи інтерфейсу САПР Active-HDL: *Менеджер проекту, Редактор, Консоль*.

2. ПОРЯДОК ВИКОНАННЯ РОБОТИ:

1. Вивчити теоретичні відомості про програмну конструкцію **entity** (розділ 4.1).
2. Ознайомитись з складом, інтерфейсом та можливостями інтегрованого середовища Active-HDL за допомогою *Help*.
3. В наборі VHDL-прикладів відкрити проект *Modulator* (... \Active-HDL x.x\Projects\Modulator\modulator.adf) та вивчити його склад і структуру за допомогою *Design Browser*.
4. Скласти повний перелік об'єктів, що входять до складу проекту *Modulator*, описати їх інтерфейси.
5. Вивчити порядок застосування та функціональні можливості *Майстра Нового Проекту* (*New Design Wizard*).
6. Створити за допомогою *Майстра Нового Проекту* (*New Design Wizard*) порожній проект.

7. Описати на VHDL об'єкт, що являє собою RS-тригер.
Інтерфейс цього об'єкта: 2 вхідні порти R і S та 2 вихідні порти U та INV_U типу STD_LOGIC .
8. Згенерувати такий же об'єкт в окремому файлі за допомогою *Майстра*.
Порівняти об'єкт, сформований за пунктом 6, із згенерованим автоматично об'єктом (п.7).
9. Згенерувати за допомогою *Майстра* об'єкт, що реалізує 4-бітний лічильник. Його інтерфейс: 2 вхідних порти CLK і RST типу STD_LOGIC та один вихідний 4-розрядний порт X .
10. Підготувати до захисту звіт.

3. ЗМІСТ ЗВІТУ

В звіт необхідно включити:

- а) назву та мету виконання лабораторної роботи;
- б) опис засобів Active-HDL для управління проектами;
- в) опис структури VHDL-проекту;
- г) перелік об'єктів проекту *Modulator* та їх інтерфейси;
- д) склад та структуру проекту, сформованого в результаті роботи;
- е) перелік об'єктів проекту, сформованого в результаті роботи, та їх інтерфейси;
- є) висновки по роботі.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Склад та структура проекту в Active-HDL.
2. Аналіз можливостей Майстра Нового Проекту (*New Design Wisard*).
3. Які засоби надаються Active-HDL для управління проектом?
4. Які можливості має Менеджер проекту (*Design Browser*)?
5. Що таке інтерфейс?
6. Яким чином описуються об'єкти в VHDL?

ЛАБОРАТОРНА РОБОТА № 2

Розробка програми та моделювання декодера для рідкокристалічного індикатора

Мета роботи: Вивчити методи описання поведінки об'єктів за допомогою архітектур та процесів, методи застосування оператора вибору **case**. Засвоїти методику моделювання поведінки об'єктів в САПР Active-HDL.

1. ПОСТАНОВКА ЗАДАЧІ

Для візуалізації процесів, що протікають в складному цифровому пристрої необхідно виводити значення сигналів на рідкокристалічний індикатор. Розробіть за допомогою VHDL об'єкт (декодер), що перетворює двійкове число, яке подається на його вхід, у сигнал для одного розряду рідкокристалічного індикатора.

Кожен розряд індикатора (рис.Л2.1,а) представляє собою 7 сегментів. Подання на сегменти логічної одиниці (наявність напруги) викликає висвічення цих сегментів. Наприклад, подання на індикатор числа "1101101" викличе висвічення трійки (див. рис. Л2.1,б).

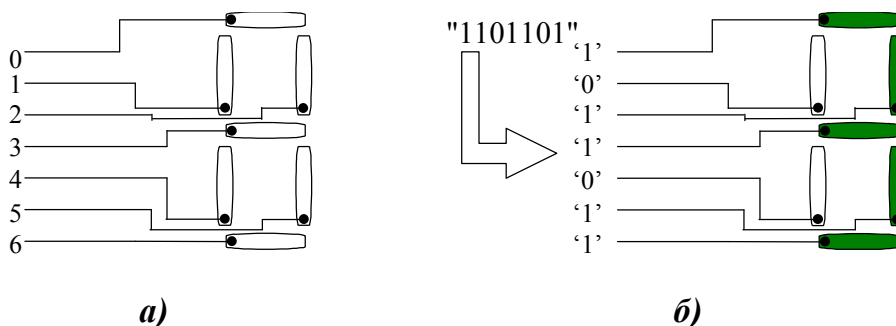


Рис. Л2.1 – Рідкокристалічний індикатор

Інтерфейс декодера: 4-розрядний вхідний сигнал X типу *STD_LOGIC_VECTOR* (3 **downto** 0); вихідний 7-розрядний порт *LCD_INDICATOR* типу *STD_LOGIC_VECTOR* (6 **downto** 0).

Робота декодера має здійснюватися таким чином: якщо на вхід X об'єкта подано число від 0 до 9, то декодер має подати на вихід *LCD_INDICATOR* сигнал, що сформує зображення цього числа. У іншому випадку на вихід подається сигнал "0000000" (жоден сегмент індикатора не висвічується). Декодер повинен оновлювати сигнал *LCD_INDICATOR* кожного разу, коли змінюється значення вхідного порту X .

Важливим етапом розробки проекту є перевірка правильності його роботи – тестування. Така перевірка звичайно здійснюється шляхом моделювання всіх ситуацій, що можуть виникнути на вході об'єкта.

Передбачається, що перед початком моделювання поведінка об'єкта вже описана VHDL-кодом, що компілюється без помилок. В Active-HDL процес моделювання складається з ряду наступних етапів:

Етап 1. Визначення верхнього рівня для моделювання (Top level selection) – більшість реальних проектів для синтезу цифрових пристроїв складаються з більше ніж одного об'єкта **entity**. Верхній рівень вказує на те, який саме з усіх наявних об'єктів **entity** буде моделюватися в наступній сесії.

Етап 2. Вибір способу відображення процесу моделювання – хід моделювання в Active-HDL можна візуалізувати різними способами – за допомогою часових діаграм (**waveform editor**) або таблиць істинності (**list**). Вибір способу здійснюється шляхом додавання відповідного файлу до проекту.

Етап 3. Формування тестових векторів (stimulators) – процес моделювання полягає в знаходженні залежностей значень сигналів вихідних портів об'єкта від часу при відомих залежностях значень сигналів вхідних портів від часу. Залежності сигналів вхідних портів від

часу у Active-HDL формуються за допомогою так званих **стимуляторів**, що приєднуються до цих портів.

Етап 4. Розрахунок залежностей значень вихідних портів об'єкта від часу – власне режим імітаційного моделювання.

2. РЕКОМЕНДАЦІЇ ДО НАПИСАННЯ ПРОГРАМИ

1. Програму слід писати за допомогою оператора вибору case (п. 3.2) або конкурентного оператора призначення сигналу (дивіться Help - допомогу Active-HDL). В останньому випадку програма не буде містити операторів process.
2. Зверніть увагу, що при застосуванні оператора вибору case мають бути задані у вигляді альтернатив всі можливі значення змінної чи сигналу, за якими здійснюється вибір.

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Вивчити розділи 4.2, 4.6, 3.2 теоретичної частини.
2. Створити новий проект в Active-HDL.
3. Створити об'єкт декодера та описати на VHDL його поведінку.
4. Скомпілювати створений об'єкт (меню Design\Compile або клавіша <F11>).
5. В вікні Design Browser для встановлення верхнього рівня моделювання (Top Level) вибрати з випадаючого списку об'єкт-декодер.
6. Ініціалізувати моделювання об'єкта (меню Simulation\Initialize Simulation).
7. Вставити в проект новий файл Waveform Viewer (меню File\New\Waveform).
8. Вставити в вікно Waveform Viewer сигнали входів та виходів декодера (меню Waveform\Add Signals).

9. Почергово помічаючи вхідні сигнали декодера в вікні Waveform Viewer, призначити їм стимулятори (меню Waveform\Stimulators).
10. Запустити процес моделювання об'єкта (меню Simulation\Run).
11. Вивчити отримані часові діаграми роботи декодера.
12. Змінюючи стимулятори на вхідних портах декодера перевірити правильність його роботи при всіх можливих значеннях входів.

4. ЗМІСТ ЗВІТУ

В звіт необхідно включити:

- а) назву та мету виконання лабораторної роботи;
- б) опис засобів Active-VHDL для моделювання об'єктів;
- в) текст VHDL-програми, що описує поведінку декодера;
- д) часові діаграми роботи декодера;
- е) висновки по роботі.

5. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Яким чином здійснюється моделювання поведінки об'єкта в Active-HDL?
2. Які типи стимуляторів надає Active-HDL для формування вхідних сигналів об'єкта?
3. Опишіть синтаксис та роботу оператора вибору **case**.
4. Що таке процес?
5. Що таке список чутливості?

ЛАБОРАТОРНА РОБОТА №3

Моделювання режиму очікування в цифрових пристроях за допомогою оператора `wait`

Мета роботи: Отримати практичні навички застосування оператора `wait` при моделюванні різноманітних режимів очікування в Active-HDL. Навчитися виконувати порівняння часових діаграм.

1. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Ознайомиться з матеріалом, наведеним у розділі 4.7 теоретичної частини.
2. Відкрити базовий проект *LabWait*, текст якого наведено на рис. Л3.1.
3. Промоделювати роботу *LabWait*. Отримати часові діаграми для сигналів *CLK* (задається як джерело з синхронізуючим імпульсом частотою 50 МГц), *Aout*, *Bout*.
4. Зберегти результати моделювання, як файл з ім'ям *Wform1*, у поточному каталозі.

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity LabWait is
    port (
        CLK: in std_logic;
        Aout: inout STD_LOGIC:='0';
        Bout: out STD_LOGIC:='0';
        Cout: out std_logic:='0');
end LabWait;

architecture LabWait of LabWait is
--    signal CLK: std_logic;
begin
```

```

--  Pr_CLK: process is
--
--      ...

--  end process Pr_CLK;

Pr_A: process (CLK) is
begin
    if CLK'event and CLK = '1' then
        Aout<='1'
after 5 ns;
    elsif CLK'event and CLK='0' then
        Aout<='0' after 5 ns;
    end if;
end process Pr_A;

Pr_B: process (Aout) is
begin
    if Aout'event then
        Bout<= not Aout;
    end if;
end process Pr_B;

end LabWait;

```

Рис. Л3.1 – Текст VHDL-програми для моделювання

5. За допомогою **wait for** сформувати синхронізуючий імпульс *CLK* самостійно (попередньо виключивши цей сигнал з інтерфейсу системи) в тілі процесу *Pr_CLK*. Рівень цього сигналу повинен змінюватись з високого на низький через кожні 10 ns.
6. Вивчити роботу процесу *Pr_A* та переписати його з використанням оператора **wait on**
7. Вивчити роботу процесу *Pr_B* та переписати його з використанням оператора **wait until**.

8. Промодельювати роботу створених за допомогою оператора **wait** процесів, отримати часову діаграму.
9. Порівняти поточну часову діаграму та базову часову діаграму Wform1.
 - Ввійти до меню Waveform та вибрати пункт Compare Waveforms. В результаті на екрані з'явиться вікно «Open».
 - Вибрати файл Wform1.awf та натиснути кнопку «Open». В результаті до поточної діаграми будуть додані часові залежності, які зберігаються у Wform1.awf. Темно-синім кольором позначені поточні часові залежності (поз.1, рис ЛЗ.2), червоним – додані часові залежності (поз.2, рис. ЛЗ.2). Розузгодження між сигналами, які порівнюються, позначається жирною блакитною лінією (поз.3, рис ЛЗ.2).
 - Якщо базова та поточна діаграми співпадають, то перейти до наступного пункту виконання роботи.

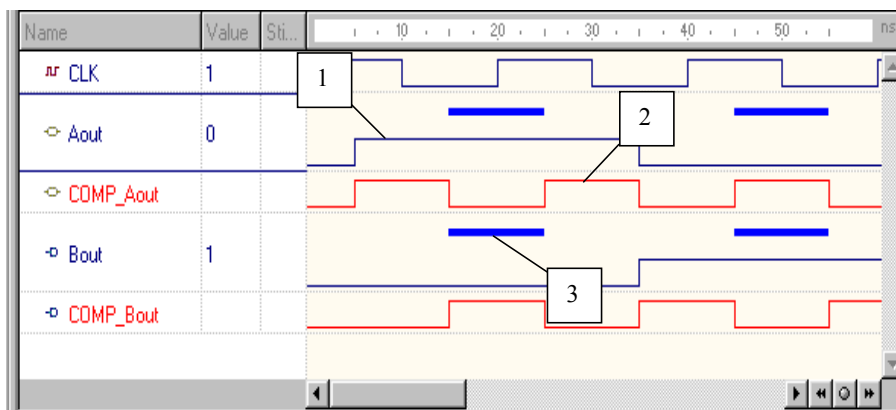


Рис. ЛЗ.2 – Приклад вікна порівняння результатів моделювання

10. Вийти з режиму порівняння. Для цього необхідно вибрати у вікні часових діаграм будь-яку лінію розузгодження сигналів (жирну блакитну лінію), та натиснути праву кнопку миші. У меню, що з'явилося, оберіть пункт «Remove Difference Marks».
11. Додати до послідовності операцій процесу *Pr_A* вираз **wait for 30 ns**.

12. Промодельовати роботу процесу, записати отриману часову діаграму як файл Wform2.awf.
13. У виразі **wait for**, який був доданий у п.10, змінити час затримки на 20 ns.
14. Промодельовати роботу процесу. Отриману часову діаграму порівняти з Wform2.awf.
15. Скласти звіт.

2. ЗМІСТ ЗВІТУ

У звіт необхідно включити:

- а) назву та мету виконаної лабораторної роботи;
- б) визначити основні види застосування оператора **wait**;
- в) навести варіант базового коду з файлу LabWait та відповідні результати моделювання (п.3);
- г) навести варіанти коду із застосуванням оператора **wait** (пп. 5, 6, 7);
- д) навести результати порівняння (згідно п.9);
- е) навести результати моделювання (згідно п.14);
- і) зробити висновки по роботі.

3. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Назвіть два способи призупинки процесів в VHDL, поясніть чим вони відрізняються.
2. Чи допускається одночасне використання оператора **wait** і списку чутливості?
3. Які три основних типи оператора **wait**, їх синтаксис?
4. Як формується комбінований тип оператора **wait**?
5. Опишіть порядок операцій, які потрібно виконати для порівняння двох часових діаграм.

6. Чим пояснюється різниця у зовнішньому вигляді базової та поточної часової діаграми, порівняння яких було зроблено при виконанні пп. 10, 12.

ЛАБОРАТОРНА РОБОТА №4

Моделювання транспортної та інерційної затримок часу

Мета роботи: Навчитися описувати транспортну та інерційну затримки часу за допомогою Active-HDL, з'ясувати принципові відмінності між ними.

1. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Ознайомитись з матеріалом, наведеним у розділі 2.4 теоретичної частини.
2. Для свого варіанту схеми, який визначається викладачем, описати об'єкт (entity) і відповідний інтерфейс в коді VHDL.

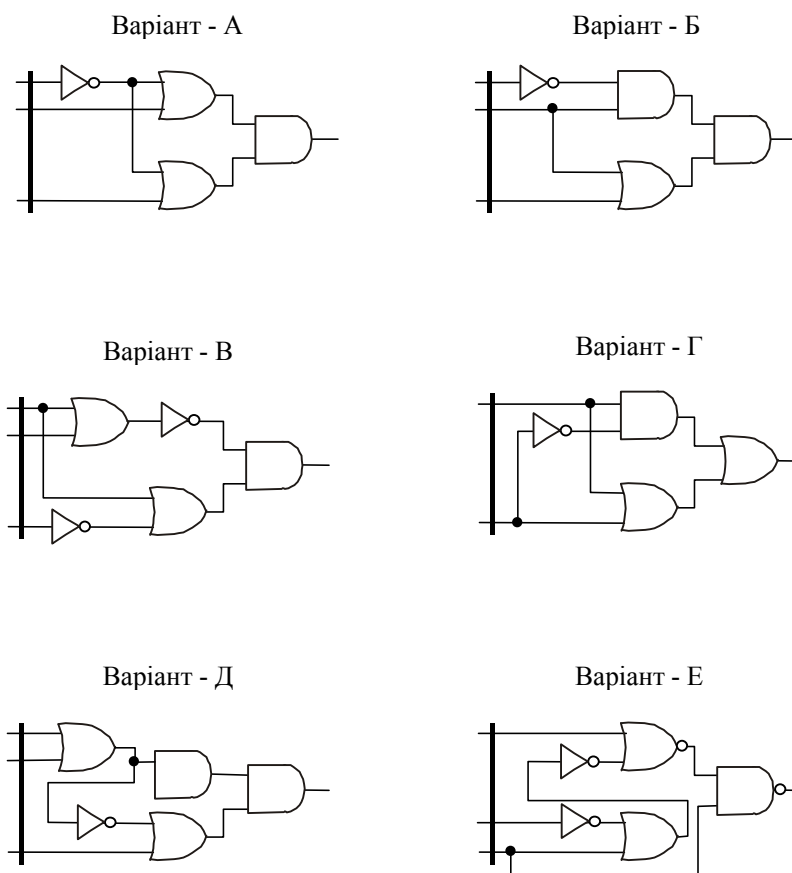


Рис. 4.1 – Варіанти завдань

3. Описати схему кодом VHDL з урахуванням часових затримок всіх її елементів (для логічних елементів ТАК та АБО – 6 ns, для інвертора – 2 ns).
4. Про моделювати роботу схеми для випадків: а) коли тривалість вхідного сигналу більша за інерційну затримку логічних елементів; б) коли тривалість вхідного сигналу менша за інерційну затримку логічних елементів.
5. Порівняти отримані часові діаграми за методикою, яка описана в лабораторній роботі № 3.
6. Змінити структуру програми, щоб пропускалися лише імпульси, тривалість яких складає 5 ns і більше.
7. Про моделювати роботу схеми.
8. Змінити структуру програми таким чином, щоб вхідні сигнали надходили з постійною затримкою 15 ns.
9. Про моделювати роботу схеми.

2. ЗМІСТ ЗВІТУ

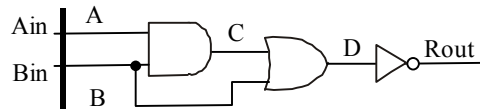
У звіт необхідно включити:

- а) назву та мету виконаної лабораторної роботи;
- б) визначення транспортної та інерційної затримок, метод їх моделювання за допомогою Active-HDL;
- с) принципову схему з'єднання логічних елементів для моделювання;
- д) опис схеми в VHDL- коді;
- е) результати моделювання для різної тривалості вхідних впливів (п.4), їх порівняння;
- є) навести текст коду для реалізації пп. 5, 7 завдання, та результати моделювання;

ж) зробити висновки по роботі.

3. ПРИКЛАД

Описати об'єкт з урахуванням затримок часу в його елементах та промоделювати його роботу. Схема логічних елементів об'єкта:



Опис схеми за допомогою VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;

entity LabDelay is
    port (    Ain: in STD_LOGIC;
             Bin: in STD_LOGIC;
             Rout: out STD_LOGIC
    );
end LabDelay;

architecture LabDelay of LabDelay is
    signal A, B, C, D: std_logic := 'U' ;
    constant TranD: time := 15 ns;
    constant GateD: time := 6 ns;
    constant InvD: time:=2 ns;
begin
    Pr_Start: process (Ain, Bin) is
        begin
            A<= transport Ain after TranD;
            B<= transport Bin after TranD;
        end process Pr_Start;

        C<= A and B after GateD;
  
```

```

    D<= C or B after GateD;

    Rout<= not D after InvD;
end LabDelay;

-- Фрагмент VHDL коду для п.5
C <= reject 5 ns inertial Ain and Bin after GateD;
D<= reject 5 ns inertial C or Bin after GateD;
Rout<= not D after InD;

```

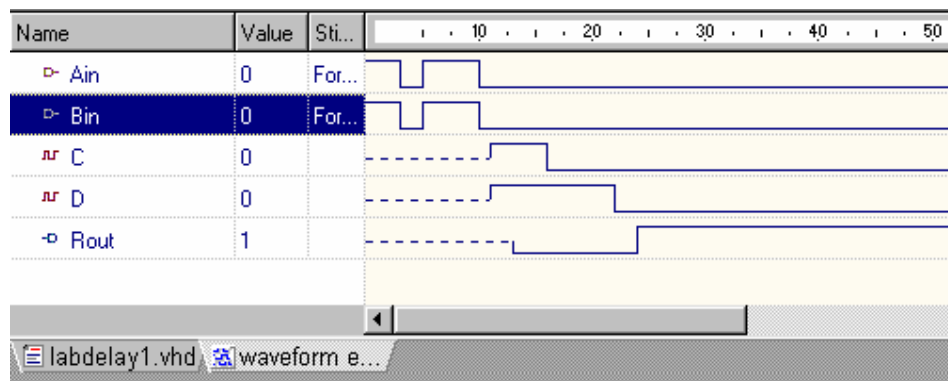
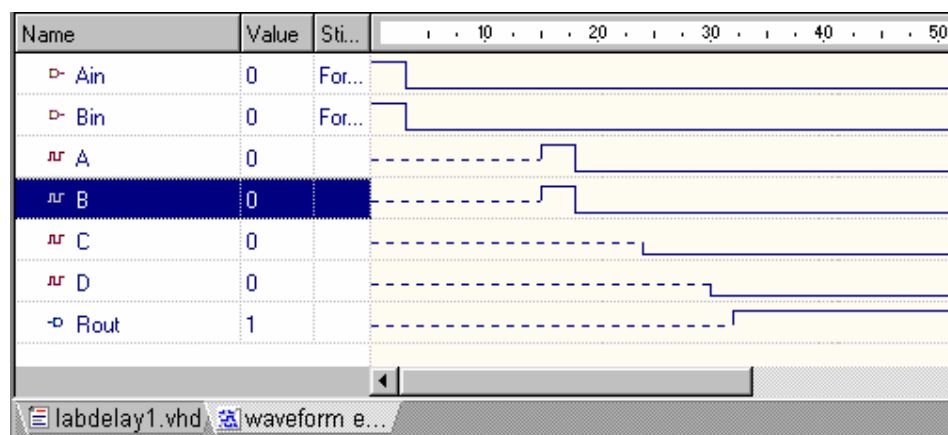
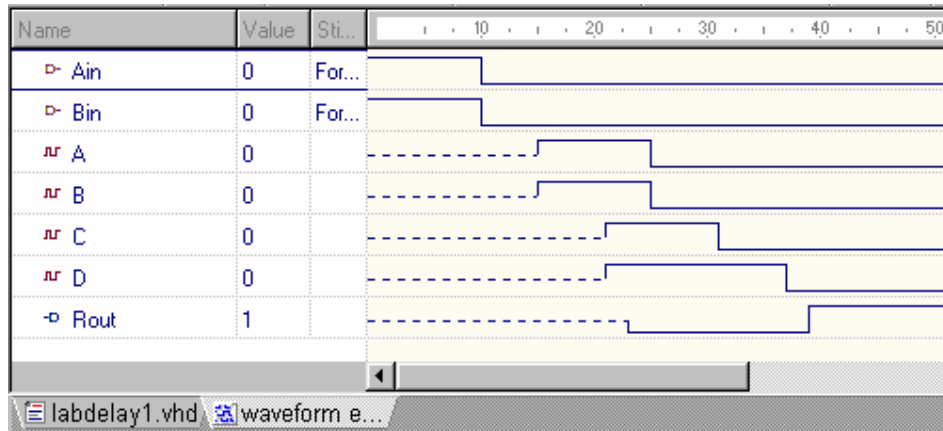


Рис. Л4.3 – Фільтрація сигналу з тривалістю менше ніж 5 ns



а) тривалість вхідного сигналу менша ніж
інерційна затримка логічного елемента



б) тривалість вхідного сигналу більша ніж
інерційна затримка логічного елемента

Рис. Л4.4 – Моделювання роботи схеми при наявності інерційної і транспортної затримок

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке інерційна затримка, її фізичний зміст?
2. Що таке транспортна затримка, її фізичний зміст?
3. Як описуються інерційна і транспортна затримки в VHDL?
4. Фільтрація імпульсів заданої тривалості засобами VHDL.

ЛАБОРАТОРНА РОБОТА № 5

Розробка програми і моделювання паралельного та зсувного регістрів

Мета роботи: Засвоїти принципи роботи паралельного та зсувного регістрів. Навчитися описувати синхронізовані процеси та використовувати циклічні оператори при описанні поведінки об'єктів за допомогою VHDL.

1. ПОСТАНОВКА ЗАДАЧІ

Необхідно описати за допомогою VHDL роботу паралельного та зсувного регістрів, промодельовати їх роботу та одержати часові діаграми.

Паралельний 8-розрядний регістр має:

8-розрядний вхід *DATA_IN* (7 **downto** 0) типу *STD_LOGIC_VECTOR* для передачі даних;

вхід синхронізації *CLK*;

вхід дозволу на запис *WE*;

вхід дозволу зчитування *RE* (типу *STD_LOGIC*).

Крім того, регістр має 8-розрядний вихід *DATA_OUT* (7 **downto** 0) для виводу даних (типу *STD_LOGIC_VECTOR*).

Робота регістра має здійснюватися таким чином:

1. У стані збереження байта на виході регістра постійно утримується високий імпеданс ("ZZZZZZZZ"). Це дає змогу організувати роботу кількох регістрів на одну шину, так як, згідно таблиці перекриття сигналів (див. розділ 2.11 теоретичної частини), сигнал високого імпедансу має найнижчий пріоритет.
2. Якщо *WE* = '1' і *RE* = '0', то здійснюється запис інформації до регістру.
3. Якщо *WE* = '0' і *RE* = '1', то на вихід регістру подається значення байта, що зберігається в цьому регістрі.

4. Всі інші комбінації *WE* та *RE* розглядаються як стан збереження байта.
5. Робота регістра має бути синхронізована за сигналом *CLK*.

Зсувний 8-розрядний регістр має:

один вхід *DATA_IN* типу *STD_LOGIC* для вводу інформації;

вхід синхронізації *CLK*;

вхід дозволу на запис *WE*;

вхід дозволу зчитування *RE* (типу *STD_LOGIC*).

Крім того, регістр має 8-розрядний вихід *DATA_OUT* (7 **downto** 0) для паралельного виводу даних (типу *STD_LOGIC_VECTOR*).

Робота регістра має здійснюватися таким чином:

1. У стані збереження байта на виході регістра постійно утримується високий імпеданс (“ZZZZZZZZ”).
2. Якщо *WE* = '1' і *RE* = '0', то здійснюється запис інформації до регістру, при цьому сигнал *DATA_IN* надходить в *DATA_OUT*(0), значення *DATA_OUT*(0) зміщується в *DATA_OUT*(1) і так далі.
3. Якщо *WE* = '0' і *RE* = '1', то на вихід регістру подається значення байта, що зберігається в цьому регістрі.
4. Всі інші комбінації *WE* та *RE* розглядаються як стан збереження байта.
5. Робота регістра має бути синхронізована за сигналом *CLK*.

2. РЕКОМЕНДАЦІЇ ДО НАПИСАННЯ ПРОГРАМИ

1. Для збереження проміжної інформації доцільно застосовувати змінні типу *STD_LOGIC_VECTOR* (7 **downto** 0).
2. Для забезпечення синхронізації в список чутливості процесів доцільно поміщати сигнал *CLK*.
3. Зсув у зсувному регістрі слід реалізовувати за допомогою циклу з параметром (**for**).
4. Більш ефективним є зсув від старшого (7-го) біта до молодшого (0-го).

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Вивчити розділи 4.13, 3.4 теоретичної частини.
2. Розробити VHDL-модель паралельного 8-розрядного регістру.
3. Промодельовати роботу паралельного регістру, розробленого в попередньому пункті, в режимах запису інформації, збереження байта та зчитування інформації.
4. Проаналізувати на основі одержаних часових діаграм відповідність роботи паралельного регістра заданому алгоритму.
5. Розробити VHDL-модель зсувного 8-розрядного регістру.
6. Промодельовати роботу зсувного регістра, розробленого в попередньому пункті, в режимах запису інформації, збереження байта та зчитування інформації.
7. Проаналізувати на основі одержаних часових діаграм відповідність роботи зсувного регістра заданому алгоритму.

4. ЗМІСТ ЗВІТУ

В звіт необхідно включити:

- а) назву та мету виконання лабораторної роботи;
- б) опис циклічних конструкцій, що використовуються в VHDL;
- в) тексти VHDL-програм, що описують поведінку паралельного та зсувного регістрів;
- д) часові діаграми роботи регістрів;
- е) висновки по роботі.

5. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Опишіть роботу паралельного регістру.
2. Опишіть роботу зсувного регістру.
3. Перерахуйте типи циклічних операторів у VHDL.
4. Яким чином в VHDL реалізується цикл з параметром?

5. Яким чином в VHDL реалізується цикл “Поки”?
6. Яким чином в VHDL реалізуються вкладені цикли?
7. Скільки модельного часу займає цикл, в тілі якого немає операторів **wait**? Чи залежить модельний час від числа ітерацій такого циклу?

ЛАБОРАТОРНА РОБОТА №6

Проектування постійного запам'ятовуючого пристрою

Мета роботи: Вивчити принципи роботи постійного запам'ятовуючого пристрою (ПЗП). Отримати навички застосування масивів при створенні проектів у Active-HDL.

1. ПОСТАНОВКА ЗАДАЧІ

Запам'ятовуючі пристрої (ЗП) цифрової техніки призначені для запису, зберігання та видачі інформації, що надається у вигляді цифрового коду.

Однак існує і така інформація, яка не повинна змінюватись, наприклад різноманітні константи, цифрові коди букв українського або латинського алфавіту та таке інше. Зазвичай така інформація записується у постійний запам'ятовуючий пристрій, для якого в процесі роботи дозволяється тільки зчитування інформації, яка в нього занесена. В ПЗП за кожною n -розрядною адресою записане одне заздалегідь визначене m -вимірне слово. Таким чином ПЗП є перетворювачем коду адреси в код слова, тобто комбінаційною системою з n входами та m виходами. Накопичувач ПЗП апаратно виконується у вигляді системи взаємно-перпендикулярних шин, на перетині яких або присутній (логічна 1), або відсутній (логічний 0) елемент, що пов'язує між собою відповідні (горизонтальну та вертикальну) шини (рис. Л6.1). Вибірка слів виконується за допомогою дешифратора.

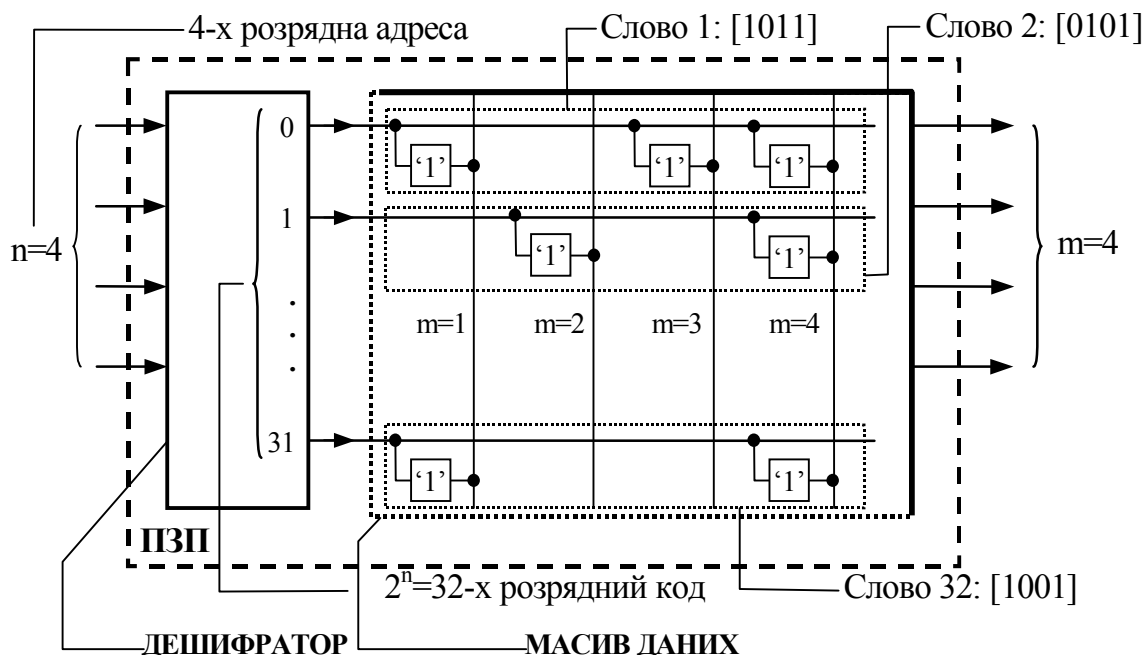


Рис. Л6.1 – Функціональна схема ПЗУ

2. ПОРЯДОК ВИКОНАННЯ РОБОТИ.

1. Ознайомитись з принципом роботи ПЗП та теоретичним матеріалом, наведеним у розділі 3.8.
2. За допомогою Active-HDL описати роботу ПЗП, який зберігає коди команд арифметичних операцій.
 - 2.1. Створити новий проект.
 - 2.2. Додати бібліотеку *ieee.std_logic_unsigned*.
 - 2.3. Описати інтерфейс ПЗП, маючи на увазі, що до його входу надсилається 4-бітний сигнал адреси (розмірність вхідного та вихідного сигналів даних вибрати самостійно);
 - 2.4. Визначити тип масиву даних, які будуть зберігатись в ПЗП.
 - 2.5. Сформувати масив даних, які зберігаються в ПЗП.
 - 2.6. Описати процес вибірки даних, які зберігаються в ПЗП.

3. Скомпілювати та промодельовати процес вибірки даних. Результати моделювання представити у табличній формі за допомогою “List Viewer”.
4. Виконати звіт по роботі.

3. ЗМІСТ ЗВІТУ

У звіт необхідно включити:

- а) назву та мету лабораторної роботи;
- б) функціональну схему ПЗП та його короткий опис;
- в) навести основні теоретичні відомості про використання масивів в Active-HDL;
- г) навести VHDL-код для опису ПЗП;
- д) навести результати моделювання у табличній формі;
- е) зробити висновки по роботі.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке ПЗП і які його основні функції?
2. Як описуються та використовуються масиви в Active-VHDL?
3. Яким чином можна задавати масиви із змінною розмірністю?
4. Яка роль бібліотеки *ieee.std_logic_unsigned* в програмі опису ПЗП?

ЛАБОРАТОРНА РОБОТА №7

Проектування сканеру клавіатури з застосуванням діаграм скінчених автоматів для опису об'єктів в САПР Active-HDL

Мета роботи: Вивчити принцип роботи сканера клавіатури. Отримати навички проектування цифрових пристроїв за допомогою скінчених автоматів засобами Active-HDL.

1. ПРИНЦИП ДІЇ СКАНЕРА КЛАВІАТУРИ (СК)

Клавіатура більшості обчислювальних пристроїв змонтована у вигляді прямокутної матриці (рис. Л7.1), у точках перетину рядків і стовпців якої розташовуються кнопкові контакти. Скануючий пристрій, що розробляється, повинен визначати номер елемента матриці, який відповідає натисненій клавіші, і видавати відповідний йому двійковий код. СК працює у двох головних режимах:

1. Режим очікування:

- 1.1 На всі стовпці матриці клавіатури (МК), які позначені на рис. Л7.1 ($C0, C1, C2$), подається логічна одиниця.
- 1.2 Всі рядки МК, які позначені на рис. Л7.1. ($R0, R1, R2$), мають низький рівень сигналу – логічний нуль.
- 1.3 Рівень сигналів R перевіряється з кожним синхронізуючим імпульсом CLK і, якщо один з них змінить своє значення з “0” на “1”, то це сигналізує про натиснення однієї з клавіш. При цьому СК переходить в наступний режим.

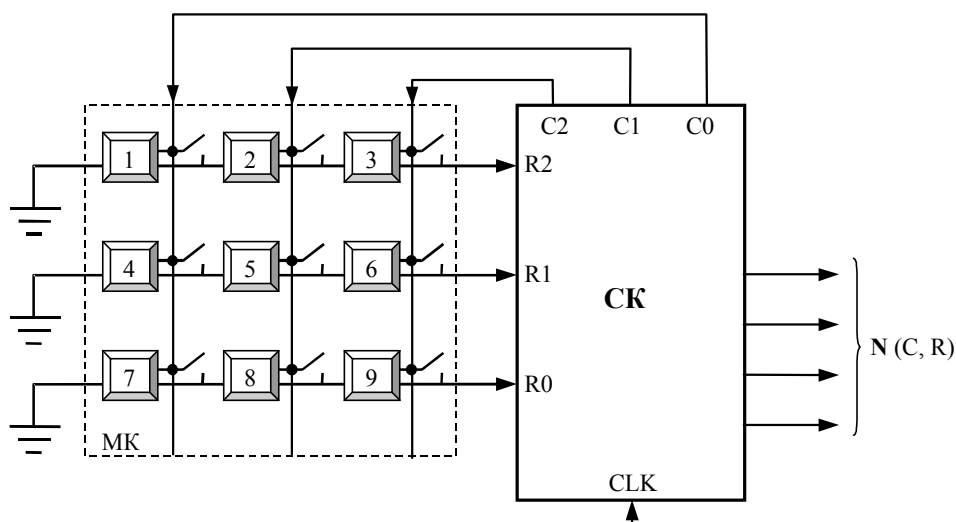


Рис. Л7.1 – Функціональна схема сканера клавіатури

2. Режим сканування:




- 2.1. На стовпець C0 подається сигнал високого рівня, а на всі інші стовпці – низького рівня.
- 2.2. Перевіряється рівень сигналів R.
- 2.3. Якщо рівень одного з R (наприклад R1) зміниться з “низького” на “високий”, то це означає, що було замкнено контакт, який розташований на перетині стовпця C0 та рядка R1. Тобто номер елемента МК, якому відповідає натиснена клавіша, знайдено.
- 2.4. Сканер передає цю інформацію у декодер для формування відповідного коду обраної цифри (або операції) та знову переходить до режиму очікування;
- 2.5. Якщо рівень всіх рядків R залишається низьким, то пп.2.1,2.2 повторюються послідовно для усіх стовпців C0 – C2 до тих пір, доки не буде виконана умова 2.3.



Приклад (Відповідно до рис.Л7.1)

№	Дії	Результати (<i>P</i>) та висновки (<i>B</i>)
I	$C = (1, 1, 1)$	$P: R=(0, 0, 0);$ $B:$ СК знаходиться у режимі очікування
II	$C = (1, 1, 1);$ <div style="display: inline-block; border: 1px solid black; padding: 2px; text-align: center;">5</div> – натиснена	$P: R=(0, 1, 0);$ $B:$ Одержано інформацію про натиснення клавіші. СК переходить в режим сканування
III	$C = (1, 0, 0)$	$P: R=(0, 0, 0);$ $B:$ Натиснена клавіша не знаходиться у першому стовпці МК
IV	$C = (0, 1, 0)$	$P: R=(0, 1, 0);$ $B:$ Натиснена клавіша знаходиться на перетині другого стовпця та другого рядка МК
VI	$N=(C, R)$	$P: N := "0101";$ СК переходить до режиму очікування

2. ПОРЯДОК ВИКОНАННЯ РОБОТИ.

1. Ознайомитися з можливостями редактора скінчених автоматів (FSM Editor) в Active-HDL (розділ 5.1 теоретичної частини).
2. Виходячи з основного принципу роботи сканера клавіатури, визначити основні стани пристрою і умови переходу від одного стану до іншого (див. Приклад). При цьому кількість клавіш на клавіатурі дорівнює 15 (0 ... 9, '+', '-', '/', '*', '=').
3. Визначити інтерфейс системи (див. Приклад).
4. Створити нову діаграму скінчених автоматів (меню File\NewDesign\FSM Diagrame).

5. За допомогою New Design Wizard і New Source File Wizard створити новий проект і макет діаграми скінчених автоматів.
6. Скласти діаграму скінчених автоматів.
 - 6.1. Використовуючи кнопку  робочої панелі FSM State Editor, створити в робочій області необхідну кількість кружків (поз.1, рис. Л7.2) відповідно до кількості можливих станів СК ($S_0, S_1 \dots$), які були визначені в п.2.
 - 6.2. Використовуючи кнопку  робочої панелі FSM State Editor, відобразити за допомогою стрілок зв'язки між станами (поз.2, рис.Л7.2);
 - 6.3. Оскільки як в режимі очікування, так і в режимі сканування потрібно перевіряти чи має хоча б один з сигналів R значення "1", то буде зручно ввести локальний сигнал $K = (R_0 \text{ or } R_1 \text{ or } R_2)$ (поз. 3, рис.Л7.2) за допомогою кнопки  робочої панелі FSM State Editor.
 - 6.4. Визначити властивості даного об'єкта (параметри, дії, умови) відповідно до п.2. Для відкриття відповідного діалогового вікна слід навести курсор на умовне позначення нульового стану (S_0), клацнути правою кнопкою миші і вибрати пункт "Properties" з локального меню.
 - 6.5. Аналогічним чином визначити властивості всіх інших елементів діаграми (станів $S_1 \dots S_n$ (поз.4, рис. Л7.2), переходів (поз 5, 6, рис. Л7.2) та локального сигналу K).
 - 6.6. Додати джерело діаграми (поз. 7, рис. Л7.2). Для цього треба клацнути лівою кнопкою миші за межами полів сторінки і вибрати з локального меню пункт "Properties", визначити синхронізуючий і установчий сигнали.

7. Послідовним натисненням кнопок   сформувати і відобразити код VHDL, відповідно створеній діаграмі скінчених автоматів.
8. Вивчити відповідність графічних об'єктів діаграми і VHDL коду.
9. Скомпілювати пристрій та промодельювати його роботу.
10. Підготувати звіт по роботі.

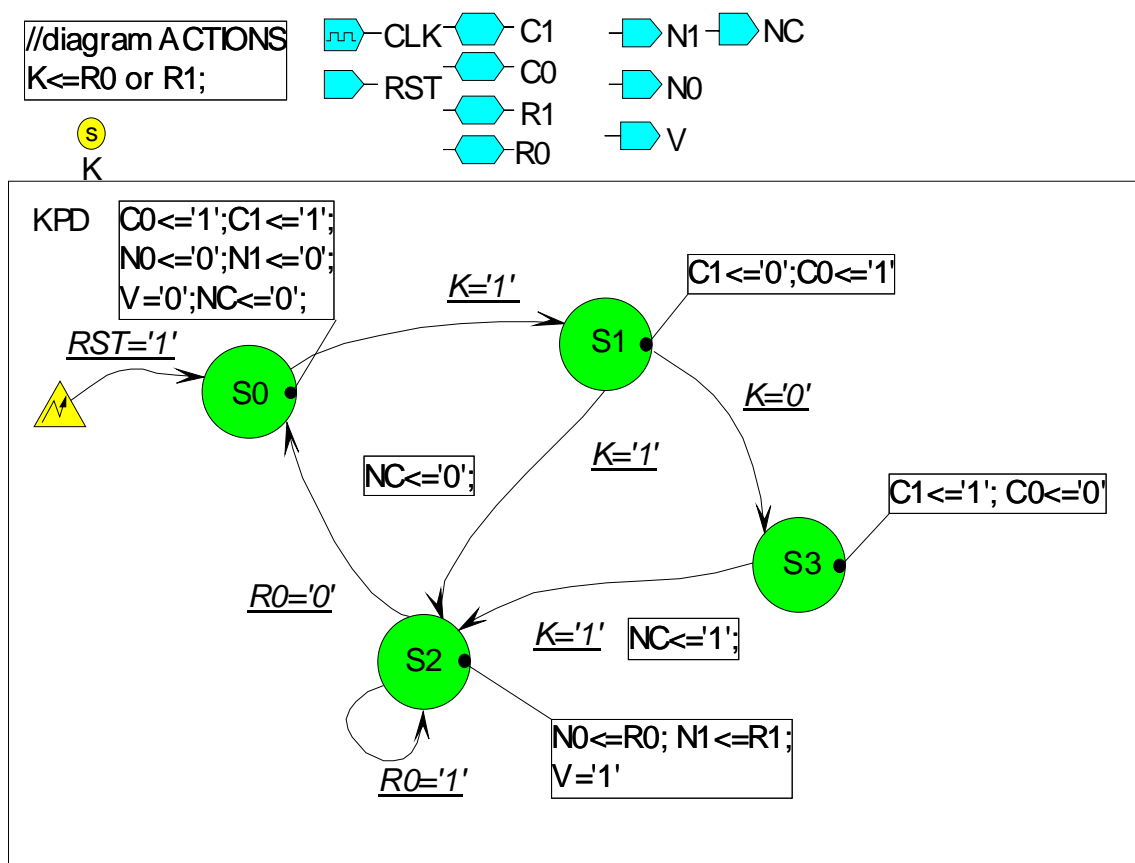


Рис. Л7.2 – Діаграма кінцевих автоматів, яка створена за допомогою
Active-HDL

3. ПРИКЛАД РЕАЛІЗАЦІЇ СКАНЕРА ДЛЯ 4-КЛАВІШНОЇ КЛАВІАТУРИ

Нижче наведено приклад реалізації сканера клавіатури для чотирьох клавіш. Визначимо основні стани та умови переходу від одного стану до іншого:

S0: $C0=C1=C2=1, R1=R0=0$

Якщо $K=R1$ or $R0=1$, переходимо в стан S1;

S1: $C0=1, C1=0$

Якщо $K=1$ – переходимо в S2; якщо $K=0$ – переходимо в S3;

S3: $C0=0, C1=1$

Якщо $K=1$ – переходимо в S2;

S2: Сформувати входні сигнали для дешифратора (номер рядка і стовпця матриці).

Якщо активований раніше сигнал R знов став рівним 0 (кнопка відпущена), то перейти в стан S0.

Результати моделювання діаграми скінчених автоматів наведені на рис. Л7.3.

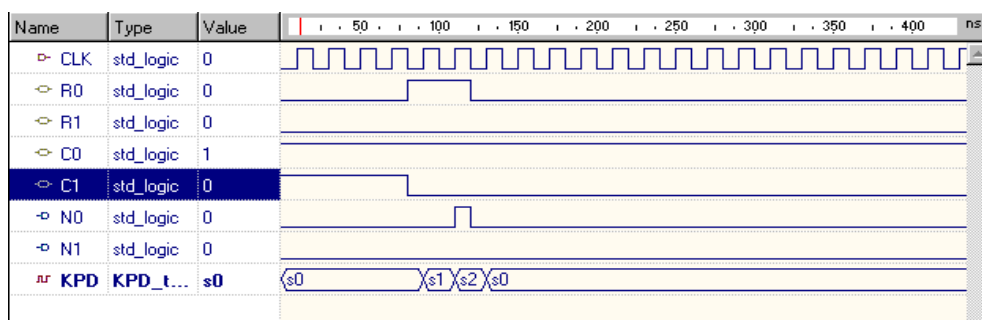


Рис. Л7.3 – Результати моделювання СК

4. ЗМІСТ ЗВІТУ

В звіт необхідно включити:

- назву та мету лабораторної роботи;
- функціональну схему сканера клавіатури для 15 кнопок та її короткий опис;

- в) основні стани СК і умови переходу від одного стану до іншого (відповідно до п.2);
- г) опис інтерфейсу системи;
- д) розроблену діаграму скінчених автоматів;
- е) відповідний VHDL-код для розробленої діаграми скінчених автоматів;
- є) результати моделювання в графічному та табличному вигляді та їх аналіз;
- ж) зробити висновки по роботі.

5. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Означення скінчених автоматів.
2. Який порядок формування нового макета діаграми скінчених автоматів при використанні New Design Wizard і New Source File Wizard?
3. Які основні етапи формування діаграми скінчених автоматів?
4. Яким чином формується джерело діаграми і що до нього входить?
5. Як описуються елементи створеної діаграми та їх властивості в VHDL-коді?
6. Яким чином визначаються дії і умови для станів і переходів між ними?
7. Чим відрізняються вхідна, стаціонарна і вихідна дії стану, відмінності в VHDL-коді?

ЛАБОРАТОРНА РОБОТА № 8

Використання блок-діаграм для декомпозиції складних пристроїв з виділенням типових структурних елементів в САПР Active-HDL

Мета роботи: Навчитися використовувати блок-діаграми (Block Diagrams) для декомпозиції складних об'єктів на структурні складові. Отримати навички застосування констант **generic** для проектування структурних елементів. Розробити модель 8-розрядного рідкокристалічного індикатора.

1. ПОСТАНОВКА ЗАДАЧІ

Створити блок – генератор сигналів для 8-розрядного рідкокристалічного індикатора. Декодер розряду індикатора, модель якого було розроблено в Л.Р.2, та паралельний регістр (Л.Р.3) мають входити до проекту як структурні складові.

Інтерфейс об'єкта – генератора сигналів містить:

вхідний 32-розрядний порт *X* типу *STD_LOGIC_VECTOR* (31 **downto** 0), на який у двійково-десятковій формі подається 8-розрядне десяткове число (на кожний десятковий розряд по 4 двійкові розряди);

вхідний порт *WE* типу *STD_LOGIC*, подання логічної '1' на який дозволяє запис у проміжний регістр блоку;

сигнал синхронізації *CLK*;

вісім 7-розрядних вихідних порти *LCD7...LCD0*, що підключаються безпосередньо до відповідних розрядів рідкокристалічного індикатора.

2. ПОРЯДОК ВИКОНАННЯ РОБОТИ:

1. Вивчити розділи 4.9, 4.10, 4.11 теоретичної частини.

2. Вивчити методи декомпозиції об'єктів за допомогою блок-діаграм.
3. Створити новий порожній проект.
4. Додати до проекту файл з описанням декодера розряду рідкокристалічного індикатора, розробленого в Л.Р.2. Для цього в менеджері проекту вибрати команду *Add_New_File*, потім у діалоговому вікні, що з'являється, натиснути кнопку *Add_Existing_File* (додати існуючий файл), в списку вибрати необхідний файл.
5. Додати до проекту файл з описанням паралельного регістра, розробленого в лабораторній роботі №5.
6. Введенням константи **generic** до 8-розрядного регістру перетворити його на універсальний регістр (див. розділ 4.10).
7. Додати до проекту новий файл – блок-діаграму (*Add_New_File, Block_Diagram*).
8. Додати в діаграму вхідні та вихідні порти (за допомогою відповідної кнопки на панелі інструментів).
9. В робочій області набрати схему блоку (рис. Л8.1)
10. Промодельовати роботу структурних складових та блоку в цілому (шляхом переключання *Top_Level* на панелі *Менеджера Проекту*).
11. Проаналізувати одержані часові діаграми роботи розробленого пристрою та зробити висновки про його адекватність поставленому завданню.
12. Конвертувати розроблену діаграму у VHDL-код, проаналізувати текст програми.

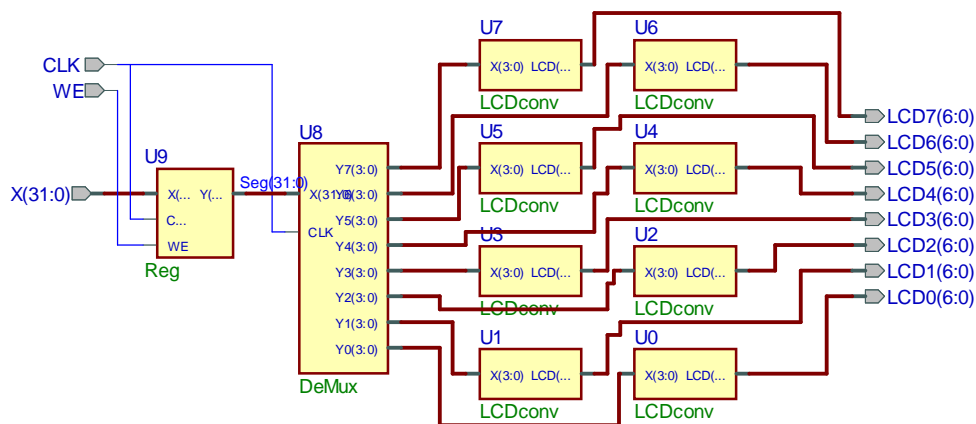


Рис. Л8.1 – Блок-діаграма генератора сигналів для 8-розрядного рідкокристалічного індикатора

3. ЗМІСТ ЗВІТУ

В звіт необхідно включити:

- а) назву та мету лабораторної роботи;
- б) опис методики застосування констант **generic** в VHDL;
- в) характеристики цілей та методів застосування блок-діаграм в САПР Active-HDL;
- г) тексти VHDL-програм, що були розроблені в роботі;
- д) часові діаграми роботи структурних складових та блоку в цілому;
- е) висновки по роботі.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Синтаксис та методи застосування констант **generic**.
2. Синтаксис та застосування оператора включення компоненти.
3. Методика декомпозиції складних об'єктів в САПР Active-HDL за допомогою блок-діаграм.
4. Який механізм включення компоненти застосовується в блок-діаграмах?

ВИСНОВКИ

В результаті вивчення матеріалу, викладеного в даному навчальному посібнику студент буде знати: методи та шляхи автоматизації проектування цифрових пристроїв; програмні засоби розробки цифрових пристроїв; методи моделювання цифрових пристроїв на ЕОМ; елементну базу для синтезу цифрових пристроїв (програмовані логічні інтегральні схеми); структуру і склад мови VHDL та її застосування у розв'язанні задач розробки та моделювання цифрових пристроїв на ЕОМ; методи реалізації типових елементів обчислювальної техніки.

Цикл лабораторних робіт (розділ 6 навчального посібник) забезпечує отримання практичних вмінь та навичок з описання поведінки цифрових пристроїв за допомогою VHDL та діаграм скінчених автоматів; здійснення декомпозиції складних цифрових пристроїв на типові елементи; моделювання роботи цифрових пристроїв на ЕОМ; використання САПР Active-HDL для проектування та моделювання цифрових пристроїв.

АНГЛО-УКРАЇНСЬКИЙ СЛОВНИК СПЕЦІАЛЬНИХ ТЕРМІНІВ

Abstract data type – Абстрактний тип даних (вказник динамічної пам'яті)

Alias – Псевдонім

Behaviour – Поведінка (с. 16)

Behaviour model – Поведінкова модель (с. 18)

Bit stream – Бітовий потік (с. 21)

Block diagram – Представлення пристрою у вигляді структурної схеми (с. 177).

Complex programmable logic device (CPLD) – Складний програмований логічний пристрій (с. 14)

Concurrent operator – Конкурентний оператор (оператор паралельного виконання) (с. 52)

Deferred constant – Попереднє визначення константи в розділі оголошення пакету (с. 47)

Delta-delay – Модельний цикл (крок моделювання) (с. 95)

Design browser – Менеджер проекту (с. 94)

Entity – Об'єкт (с. 50)

Enumeration type – Порядковий тип (с. 34)

Event – Подія (в мові VHDL зміна значення сигналу) (с. 61)

Finite state machine (FSM) – Діаграма скінчених автоматів, скінчений автомат (с. 83)

Flip-flop – Тригер

Field programmable gate array (FPGA) – Програмована матриця логічних елементів (с. 11)

Functional model – Функціональна модель (с. 18)

Gate – Вентиль (с. 11)

Globally static – Глобально статичний елемент

High impedance – Високий імпеданс (с. 81)

Impure function – функція з можливістю доступу до глобальних об'єктів даних (с. 44)

Interface Model – Інтерфейсна модель (с. 19)

JEDEC – Стандартний формат обміну даними з ПЛІС

Latch – Запам'ятовуючий елемент

Locally static – Локально-статичний (с. 27)

Macrocell – Макрокомірка (с. 14)

Netlist – Логічна структура (схема з логічних елементів) (с. 20)

Performance Model – Модель продуктивності (с. 19)

Pipeline – Конвеєр

Postponed process – Затриманий процес

Programmable logical device (PLD) – програмована логічна інтегральна схема (ПЛІС) (с. 4)

Pure function – Функція з заблокованим доступом до глобальних об'єктів даних (с. 44)

Resolution function – Функція перекриття (с. 80)

Resolved signal, port – Сигнал чи порт з більше ніж одним драйвером (с. 80)

Register Transfer Level Model (RTL-model) – моделі рівня регістрових передач (с. 21)

Sensitivity list – Список чутливості (с. 62)

Sequential statements – Послідовний оператор (с. 62)

Signal assignment – Оператор направлення сигналу (с. 53)

Signed – Ціле двійкове число, в якому старший розряд виділено під знак

Simulation time – Модельний час (с. 66)

State machine – Скінчений автомат (с. 83)

Stimulator – Стимулятор (задаючий вплив, значення входу порта в певний момент часу) (с. 98)

Structural model – Структурна модель (с. 18)

Systolic array – Однорідне обчислювальне середовище

Synthesis – Синтез (для мови VHDL – перетворення VHDL-програми на логічну структуру) (с. 21)

Test bench – Випробувальний стенд (віртуальний) (с. 87)

Test vector – Тестовий вектор, тестова послідовність, залежність значення вхідного порта від модельного часу (с. 98)

Transaction – Транзакція – відкладена зміна значення сигналу (с. 54)

Uninitialized value – Невизначена величина (об'єкт стандартного логічного типу, на який з стартового моменту часу не подавалося жодних сигналів) (с. 80)

Uninterpreted Model – Модель продуктивності (с. 19)

Unsigned – Беззнакове двійкове ціле число з діапазоном зміни від 0 й вище

Virtual Prototype – Віртуальний прототип (с. 19)

Waveform – Часова діаграма сигналу (с. 98)

WAVES – Waveform And Vector Exchange to Support Design and Test Verification – промисловий стандарт "Обмін часовими діаграмами та векторами для підтримки тестової верифікації проектів" (с. 88)

ЛІТЕРАТУРА

1. **Айзерман М.А.** и др. Логика, автоматы, алгоритмы. – М.: Физматгиз, 1963. – 556 с.
2. **Калниболотский Ю.М. и др.** Автоматизированное проектирование электронных схем. – К.: Техніка, 1987. – 301 с.
3. **Каневский Ю.С.** Компьютерная арифметика. Конспект лекций. – К.: “ДиаСофт”, 1994. – 240 с.
4. **Каневский Ю.С.** Систематические процессоры. – К.: Техніка, 1991. – 173 с.
5. **Кондратенко Ю.П.** Внедрение Active-HDL в ведущих университетах Николаевского региона Украины // Материалы международной научно-технической конференции "International Active-HDL Conference". – 15-16 октября, 2001, Харьков, Украина. – с. 25-27.
6. **Кондратенко Ю.П., Сидоренко С.А.** Методи проектування нечітких пристроїв прийняття рішень на основі програмованих логічних ІМС // Наукові записки НаУКМА, Київ, 2000. – Т. 18, ч. 2. – с. 401-412
7. **Кондратенко Ю.П., Сидоренко С.А.** Сучасні інформаційні технології для задач автоматизованого проектування цифрових пристроїв // Вісник ХДТУ. – 2000. – №1(7). – С. 229-235.
8. **Кондратенко Ю.П., Сидоренко С.А., Підопригора Д.М.** VHDL-моделі для проектування цифрових пристроїв. Методичні вказівки до циклу лабораторних робіт. – Миколаїв. УДМТУ, 2000. – 54 с.
9. **Корнейчук В.И.** и др. Вычислительные устройства на микросхемах: Справочник / В.И. Корнейчук, В.П. Тарасенко, Ю.Н. Мишинский. – К.: Техніка, 1986. – 264 с.
10. **Программируемые логические ИМС на КМОП-структурах и их применение** / П.П. Мальцев, Н.И. Гарбузов, А.П. Шарапов, Д.А. Кнышев. – М.: Энергоатомиздат, 1998. – 160 с.

11. **Електроніка і мікросхемотехніка** / В.І. Сенько, М.В. Панасенко, Є.В. Сенько та ін.; Під ред. В.І. Сенька. – К.: ТОВ "Видавництво "Обереги", 2000. – 300 с.
12. **Тутце У., Шенк К.** Полупроводниковая схемотехника. — М.: Мир, 1982. – 512 с.
13. **Цифрова техніка** / Б.Є. Рицар. – К.: УМК ВО, 1991. – 372 с.
14. **Active-HDL User's Guide**. Second edition. – Copyright ALDEC, Inc. 1999. – 213 p.
15. **Ashenden P.J.** The designer's guide to VHDL. San Francisco: Morgan Kaufmann Publishers, 1996. – 688 p.
16. **Kondratenko Y.P. , Kondratenko G.V., Pidoprigora D.M., Sidorenko S.A., Timchenko V.L.** Fuzzy approach for design of ship's decision-making systems // Proc. of 2-nd International Conference 'Management and Control of Production and Logistic' MCPL'2000, Grenoble, France, 5-8 July 2000, CD-ROM - paper 313
17. **Roth C.H.** Digital systems design using VHDL. – Boston: PWS Publishing Company, 1998. – 470 p.

ЗМІСТ

Передмова.....	4
Вступ.....	7
1. Загальна характеристика програмованої логіки.....	10
1.1 ОБЛАСТІ ЗАСТОСУВАННЯ ТА ОСНОВНІ ВИРОБНИКИ ПЛІС	10
1.2 FPGA – ТЕХНОЛОГІЯ.....	11
1.3 CPLD – ТЕХНОЛОГІЯ.....	14
1.4 МЕТОДИ ПРОЕКТУВАННЯ ЛОГІЧНОЇ СТРУКТУРИ ПЛІС.....	16
2. Алфавіт мови VHDL	22
2.1 ПРАВИЛА ФОРМУВАННЯ ІДЕНТИФІКАТОРІВ.....	22
2.2. СПЕЦІАЛЬНІ СИМВОЛИ.....	23
2.3. ЧИСЛА В VHDL.....	23
2.4. СИМВОЛИ.....	24
2.5. РЯДКИ.....	25
3. Структура мови VHDL	26
3.1. УМОВНИЙ ОПЕРАТОР.....	26
3.2. ОПЕРАТОР ВИБОРУ	26
3.3. ПОРОЖНІЙ ОПЕРАТОР.....	28
3.4. ЦИКЛИ.....	28
3.5. ЗМІННІ	33
3.6. КОНСТАНТИ.....	33
3.7. ТИПИ ДАНИХ В VHDL, ПЕРЕТВОРЕННЯ ТИПІВ.....	34
3.8. МАСИВИ	36
3.9 Підпрограми у VHDL.....	38
3.10 ПАКЕТИ ТА БІБЛІОТЕКИ	45
4. Засоби VHDL для моделювання реальних об’єктів	50
4.1. ОПИС ОБ’ЄКТІВ В VHDL (ENTITY).....	50
4.2. АРХІТЕКТУРА.....	51
4.3. СИГНАЛИ	52
4.4. ОПЕРАТОР НАПРАВЛЕННЯ СИГНАЛУ ТА СПОСОБИ РЕАЛІЗАЦІЇ ЗАТРИМКИ В VHDL	53
4.5. АТРИБУТИ СИГНАЛІВ.....	59
4.6. ПРОЦЕС.....	62
4.7. ОПЕРАТОР WAIT.....	64
4.8. МОДЕЛЮВАННЯ В VHDL.....	66
4.9. КОМПОНУВАННЯ СКЛАДНИХ ОБ’ЄКТІВ ІЗ СТРУКТУРНИХ СКЛАДОВИХ	68
4.10. КОНСТАНТИ GENERIC	71
4.11. КОМПОНЕНТИ З ПОПЕРЕДНЬО ОПИСАНИМ ІНТЕРФЕЙСОМ	74

4.12. ПАРАЛЕЛЬНА РОБОТА ПРОЦЕСІВ	78
4.13. ПЕРЕКРИТТЯ СИГНАЛІВ	80
5. Додаткові можливості САПР Active-HDL	83
5.1. ДІАГРАМИ СКІНЧЕНИХ АВТОМАТІВ В VHDL	83
5.2. ВИПРОБУВАЛЬНІ СТЕНДИ	87
5.3. ВИВЕДЕННЯ ВІДЛАГОДЖУВАЛЬНОЇ ІНФОРМАЦІЇ. ОПЕРАТОР ASSERTION.....	90
6. Цикл лабораторних робіт з формування VHDL-моделей цифрових пристроїв	94
Лабораторна робота № 1	94
Лабораторна робота № 2	97
Лабораторна робота №3	101
Лабораторна робота №4	106
Лабораторна робота № 5	111
Лабораторна робота №6	115
Лабораторна робота №7	118
Лабораторна робота № 8	125
Висновки.....	128
Література.....	129

Навчальний посібник

Кондратенко Юрій Пантелійович
Сидоренко Сергій Анатолійович
Підпригора Дмитро Миколайович

ПОВЕДІНКОВИЙ СИНТЕЗ ЦИФРОВИХ ПРИСТРОЇВ В СЕРЕДОВИЩІ ACTIVE-HDL

Під редакцією Ю.П. Кондратенка

Коректура.....

Комп'ютерний макет.....

Віддруковано в