Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерної інженерії та управління_____
(повна назва)
Кафедра_____Автоматизації проектування обчислювальної техніки_____

(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

_____
(позначення документа)

_____Функціональна верифікація абстрактних цифрових автоматів_____
(тема)

Виконав: студент 2 курсу, групи СКСм-18-1
Пшеничний К.Ю._____
(прізвище, ініціали)
Спеціальність 6.050102 – Комп'ютерна_____
інженерія_____
(код і повна назва спеціальності)
Тип програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програмаСпеціалізовані комп'ютерні системи_____
(повна назва спеціалізації)
Керівник доцент кафедри АПОТ_____
канд.техн.наук. Хаханова Г.В._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2019 р.

РЕФЕРАТ

Пояснювальна записка містить 77 сторінок, 56 рисунків, 4 таблиці, 17 джерел за переліком посилань.

UVM, SYSTEMVERILOG, PROPERTYSPECIFICATIONLANGUAGE, ВЕРІФІКАЦІЯ, ФУНКЦІОНАЛЬНА ВЕРИФІКЦІЯ, ЦИФРОВИЙ АВТОМАТ, СПЕЦІФІКАЦІЯ, МОДЕЛІ ПРИСТРОЇВ РЕАЛЬНОГО ЧАСУ

У роботі розглянуті питання верифікації основних складових HDL-опису абстрактних цифрових автоматів за допомогоювластивостей та асерцій та інструментів для функціонального покриття.

Проаналізовано різні підходи до верифікації кодування станів, переходів, генерації виходів та якості тестових наборів.

ABSTRACT

Explanatory note contains 77 pages, 56 figures, 4 tables, 17 references.

UVM, SYSTEM VERILOG, PROPERTY SPECIFICATION LANGUAGE, VERIFICATION, FUNCTIONAL VERIFICATION, FINITE STATE MACHINE, SPECIFICATION, REAL TIME DEVICE MODELS

This paper covers main aspects of assertion and functional based finite-state machine verification.

Different verification approaches of main FSM HDL aspects such as state encoding, path coverage, output logic and testbench quality were covered.

ЗМІСТ

ВСТУП

Поява мов опису апаратури (Hardware Description Languages) значно спростила процес проектування та верифікацію складних цифрових систем. Це дозволило абстрагуватися від конкретних цифрових елементів за допомогою мовних конструкцій.

Іншим рівнем абстракції є представлення цифрової системи або підсистеми за допомогою абстрактного цифрового автомата (FSM).

Типовий цикл проектування містить декілька етапів. Найважливішою та найскладнішою фазою проектування є верифікація, яка має на меті виявлення всіх помилок та несправностей. Зростання складності та розмірів цифрових дизайнів зробили цей процес комплексною задачею. Одним з сучасних підходів вирішення цієї задачі є верифікація з використанням асерцій (Assertion Based Verification).

Метою даної роботи є дослідження, аналіз та порівняння методів верифікації абстрактних цифрових автоматів за допомогою асерцій.

Предмет дослідження – моделі цифрових автоматів, спроектованих з використанням мов опису апаратури.

Об’єкт дослідження – сучасні методи тестування, відладки та верифікації цифрових систем, описаних за допомогою мов опису апаратури (HDL) з використанням асерційного підходу.

Науково-практична задача– прискорення процесу верифікації цифрової системи за допомогою автоматизації ABV процесів.

Методи дослідження:

- апарат булевої алгебри;
- теоретичний апарат властивостей, обмежень та асерцій;
- апарат функціональної верифікації мови SystemVerilog;

– засоби автоматичного проектування цифрових пристроїв (моделювання HDL-моделі).

Задачі дослідження:

– знаходження потенційних помилок при використанні тих чи інших конструкцій HDL опису;
– розробка алгоритмів верифікації цифрових автоматів задопомогою використання асерцій;
– зменшення часу відладки дизайну за рахунок використання асерцій.

1 ПОСТАНОВКА ТА АНАЛІЗ ЗАВДАННЯ

Довгий час одним з найпоширеніших способів верифікації був підхід з використання випадкових обмежених тестових наборів (Constrained Random Testing) і інструментарію для покриття коду (Code Coverage Tools). Таке середовище представлено на рис. 1.1.

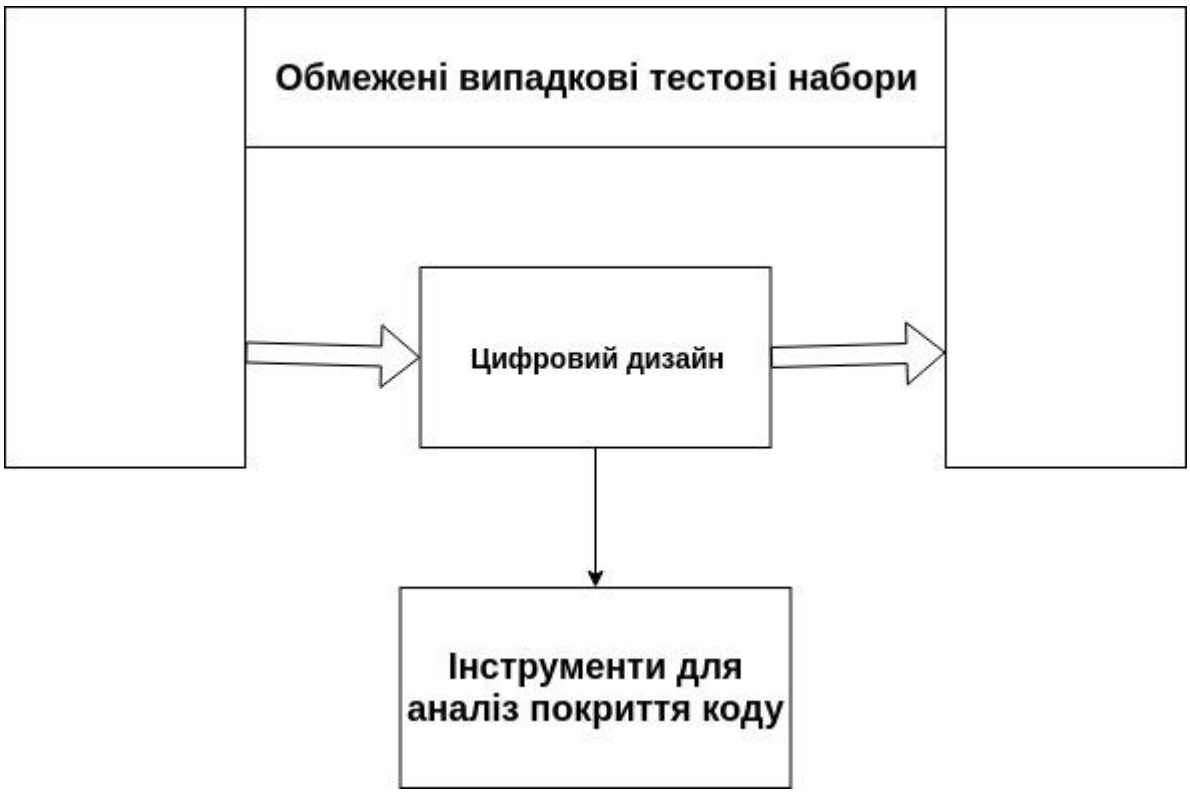


Рисунок 1.1 – Верифікаційне середовище за допомогою обмежених випадкових тестів

Основна задача процесу верифікації - ретельна перевірка дизайну на відповідність функціональній специфікації. Дані, що збираються під час аналізу покриття коду ніяк не пов’язані зі специфікацією. Такий процес надає інформацію про виконання кожного рядку коду. Звичайно, можно говорити про певний рівень якості тестів, гарантуючи, що кожний рядок дизайну було виконано хоча б раз.

Відомо, що основною одиницею тестування є тестбенч (Testbench). Намагаючись наблизитись до більш реальних сценаріїв, інженери використовують тестові дані, що генеруються випадковим чином. Як правило тестбенч виконує наступні задачі:

– генерація тестових наборів;
– подача тестових наборів на дизайн;
– автоматичний механізм для перевірки;
– вимірювання покриття функціональності.

Найпершою і найголовнішою задачею тестбенчу є генерація якісних тестових сигналів. Для цього існують спеціальні мови. OPEN VERA має вбудований механізм для опису складних тестових шаблонів. Такі мови, як правило, підтримують об’єктно-орієнтовану парадигму, що дозволяє повторно використовувати різні моделі тестбенчів.

Тестбенчі повинні також містити в собі певний механізм для самоперевірки, так як не завжди є можливість покрокової відладки після або під час моделювання. Такі підходи, як аналіз часових діаграм, також є непридатними для великих дизайнів через потенційну можливість людської помилки. Кожен тест має містити автоматичний та динамічний спосіб перевірки очікуваної поведінки. Такий підхід значно спрощує процес відладки, а регресійні тести робить більш ефективними.

Процес самоперевірки містить дві області: перевірка протоколів та перевірка даних. Перевірка протоколів полягає у валідації контрольних сигналів, які являються ядром системи. Перевірка даних пов’язана з цілісністю даних, якими оперує дизайн. Прикладом такої валідації є процес перевірки передачі пакетів у мережі без ушкоджень.

Функціональне покриття показує міру складності дизайну та містить наступну інформацію: покриття протоколів та покриття тестового плану. Покриття протоколів аналізує повноту тестів на перевірку всіх можливих валідних станів системи. Іншими словами, дане покриття показує яку частину

можливої функціональності було активізовано. Покриття тестового плану дає оцінку вичерпності тесту. Наприклад, чи створив тестбенч усі можливі розміри пакетів, чи виконав центральний процесор операції читання або запису до всіх можливих адрес ?

Асерціїні мови SystemVerilog Assertions(SVA) та Property Specification Language (PSL) дозволяють досягти максимальної ефективності та швидкодії верифікаційного середовища. Рис. 1.2 показує модифіковане середовище з використанням SVA.

ABV вирішує наступні проблеми:

– перевірка протоколів;

– покриття протоколів.

Ці задачі відносяться безпосередньо до сигналів дизайну, тому більш ефективно аналізувати їх за допомогою SVA.

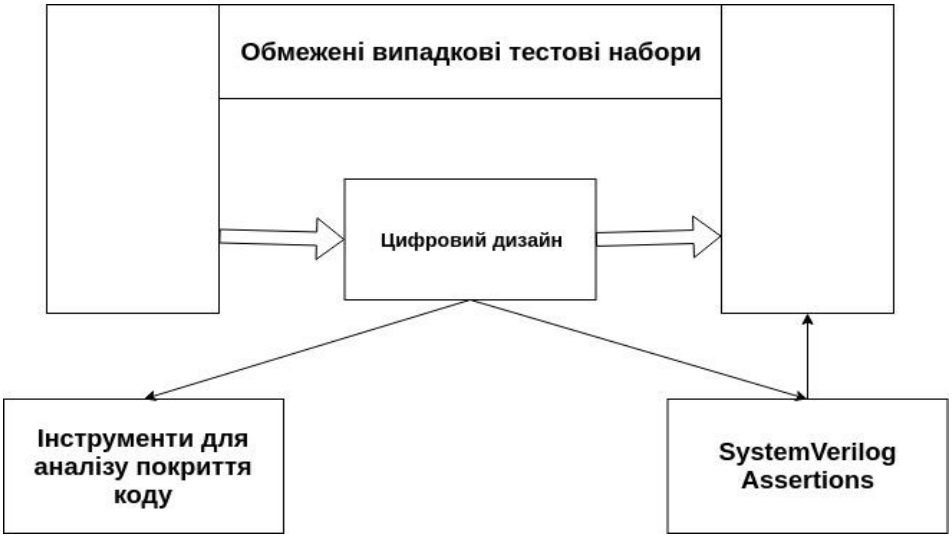


Рисунок 1.2– Модифіковане верифікаційне середовище з використанням SVA

Кінцевий автомат має дискретну множину станів, умов, які збуджують переходи між цими станами, та певний набір виходів. Як правило, кінцевий автомат моделює певну управляючу або операційну логіку.

Далі буде розглянуто та порівняно різні методи верифікації основних складових частин абстрактних цифрових автоматів з використанням асерцій за допомогою наступних критеріїв:

– кількість асерцій;
– швидкодія та продуктивність;
– ефективність;
– складність імплементації.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Перед аналізом, порівнянням та створенням методів верифікації кінцевих автоматів за допомого асерцій було виконано огляд джерел з відповідної предметної області та провести аналіз області. Виконано огляд і аналіз публікацій, що охоплюють:

- технологічний інструментарій для реалізації асерцій [3, 5, 7-8, 13, 16];
- основні принципи Assertion Based Verification (ABV) [1-3, 9, 10-12, 15];
- принципи Assertion Based Design [1-2, 5, 7-8, 10-12].

2.1 Переваги використання асерційної верифікації

Одним з традиційних методів верифікації цифрових дизайнів є верифікація за допомогою обмежених випадкових тестових наборів(Constrained Random Verification) [1]. За допомогою такого методу тестові вектори використовуються для пошуку помилок в проекті. Такий метод може показати, що в дизайні існує певна помилка проектування, проте не може вказати на її місце в коді. Для пошуку для подальшого усунення проектувальник має аналізувати поведінку системи на всіх рівнях та продивитися часову діаграми даного тесту цикл за циклом.

В [10] показно, що використання асерційного підходу при верифікації дає ряд суттєвих переваг.

Ефективність та гнучкість верифікації. Асерції можна верифікувати за допомогою формальних інструментів або середовищ моделювання. Це дає можливість підвищити якість тестування багатьох граничних випадків.

Огляд та контроль. В традиційних методах верифікації будь-який дизайн тестується за допомогою середовища моделювання та певних тестових наборів (testbench). Для ідентифікації помилки в дизайні необхідно подати відповідний тестовий набір та також вивести неправильне значення на вихід системи. Цей

підхід має назву Black-BoxVerification [5]. Асерційний підхід дозволяє знайти неправильне значення сигналу в момент його появи. Таким чином, асерція не тільки поліпшує огляд системи, а такоє покращує загальний контроль над нею завдяки зменшенню зони пошуку причини помилки. Такий підхід називається White-Box Verification та потребує певних знань про структуру системи.

Сумісність модулів. Сучасний процес розробки цифрової системи представляє собою складну систему, окремі частини якої проектують різні інженери, таким чином необхідно слідкувати за сумісністю інтерфейсів. Цього можна досягти за допомогою асерцій, які перевіряють властивості інтерфейсів [4].

Повторне використання. Як і окремі модулі, асерції можно використовувати повторно.

2.2 Стандартні мови для опису асерцій та чекери

Найбільш поширеними мовами для опису асерцій є System Verilog Assertions (SVA) та Property Specification Language (PSL) [5]. За допомогою цих мов можна описувати властивості, припущення та інваріанти дизайну. OpenVerificationLibrary представляє собою бібліотеку, що складається з певного набору модулів (чекери) для перевірки певної поведінки [6]. Такі чекери діляться на різні групи в залежності від перевіряємої поведінки, часових параметрів, взаємодії з середовищем моделюння, мови опису апаратури та вазємодії під час синтезу.

2.2.1 SystemVerilog Assertions

В джерелах [6, 7] описано способи описання асерцій цифрової системи за допомогою вбудованих властивостей мови SystemVerilog. Існує два типи асерцій: негайні(immediate) та паралельні(concurrent).

Негайні асерції представляють собою перевірку певного логічного нетемпорального виразу під час виконання інструкції [6]. На рис. 2.1 представлено синтаксис для такого типу асерцій.

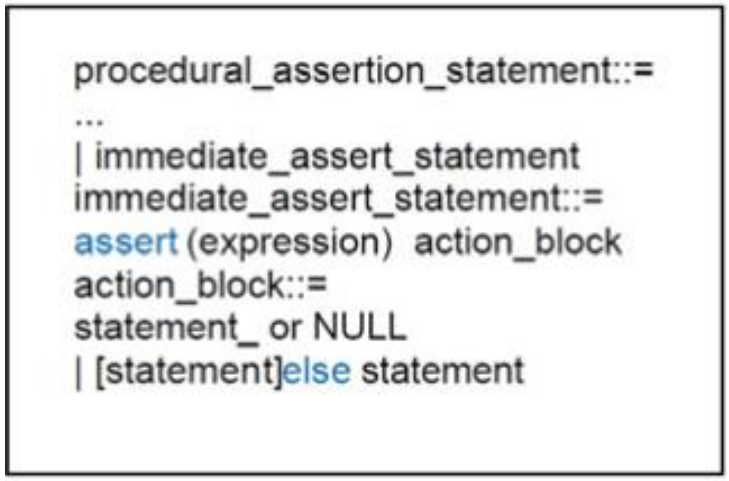


Рисунок 2.1 – Синтаксис негайної асерції на мові SVA[5]

Паралельні асерції описують поведінку з урахуванням часу. В такому виді асерційні виконання виразу завжди прив’язано до синхросигналу [6]. Вирахувані значення використовуються для перевірки певної послідовності події.

2.2.2 Property Specification Language (PSL)

PLS – окрема мова, розроблена для роботи з багатьма мовами опису апаратури (HDL) з урахуванням їхнього синтаксису для задання виразів [12]. PSLне можна використовувати напряму в HDL описі системи. Для інтеграції PSLвластивостей в цільовий дизайн використовуються bind-директиви [3]. Однією з цілей PSL властивостей є системна верифікація. Великі компанії (IBM, Intel) використовують методологію, при якій система моделюється у виді дискретного автомату та згодом верифікується згідно с архітектурними вимогами [6]. PSL надає спеціальну підтримку для такого підходу за допомогою GDL (Generic Definition Language).

PSLнадає можливість писати асерції для різних рівнів дизайну: від системного до RTL. Дана мова складається з багатьох рівнів абстракцій та має багату множину операторів, які можуть використовуватися для досягення різних цілей. Нижні рівні PSL можуть легко пристосовуватися до різних HDL та цифрових систем будь-якої складності.

У підсумку можна сказати, що PSL представляє собою багаторівневу, багатоцільову мову для написання асерцій, яка на відміну від SVA не має прив'язки до конкретної мови опису апаратури [10].

2.2.3 Open Verification Library (OVL)

OVL представляє собою набір асерційних модулів (чекерів), які перевіряють специфічні властивості дизайну. Такі чекери інстанціюються в певних місцях дизайну та надають методологію для формальної та динамічної верифікації.

Кожен OVL чекер представляє собою модуль, метою якого є гарантія, що в дизайні виконуються специфічні умови. Чекери складаються з однієї чи декількох властивостей, які необхідно перевірити, повідомлень, важливостей та покриттів.

– властивості - атрибути дизайну, які перевіряються асерціями;

– повідомлення - текстовий рядок, який буде виданий у разі провалу асерції;

– важливість показує, чим являється помилка, перехоплена чекером, звичайною проблемою чи фатальною помилкою.

Насамперед, асерційні чекери призначені для верифікаційних потреб та можуть використовуватися у наступних сценаріях:

– комбінація декількох асерційних чекерів для підвищення рівня тестового покриття дизайну;

– використання асерційних чекерів для перевірки корекності вхідних та вихідних сигналів;

– використання чекерів при взаємодії з модулями з інших бібліотек (third-party). В разі, якщо проектувальник не знайом з описом таких модулей або не до кінця розуміє всі тонкості його використання, ассерційні модулі можуть застережити від неправильного використання таких модулей.

Як правило, завжди існує спеціальний асерційний чекер для перевірки той чи іншої потенціальної проблеми. В іншому випадку можна комбінувати чекери для отримання бажаної перевірки. Також можна комбінувати асерційні модулі з додатковим HDL логікою для досягнення бажаної верифікаційної поведінки. Кількість асерцій може доходити до кількох тисяч в залежності від дизайну та складності перевіряємих властивостей.

Для більшої ефективності від використання всі асерції поділено на декілька класів:

– комбінаційні (Combinational assertions) - поведінка, яка перевіряє комбінаційну логіку дизайну;

– асерції одного циклу (1-cycle assertions) - поведінка перевіряється у поточному циклі синхросигналу;

– асерції двох циклів (2-cycle assertions) - поведінка перевіряється з поточного до наступного циклу синхросигналу;

– асерції на n-циклів (n-cycle assertions) – поведінка перевіряється впродовж декількох циклів синхросигналу;

– асерції, обмежені подіями (Event-bounded assertions) - поведінка перевіряється поміж двох подій.

Варто зазначити, що бібліотека OVL не є цілком незалежною від якоїсь мови опису апаратури.

Важливо зазначити, що частина чекерів мають HDL опис, який синтезується та може бути імплементований у цільовому пристрої на ряду зі звичайним HDL кодом. Вихід асерційного модуля можно під'єднати до контролера переривань системи для адекватної реакції на можливі помилки. Це чимось нагадує поведінку виключень у програмних системах.

Як було зазначено вище, асерційний модуль представляє собою певний модуль на мові опису апаратури зпевним набором вхідних та вихідних портів та параметрів. Нижче наведено приклад асерційного чекеру ovl_always з бібліотеки OVL, який перевіряє виконання певного логічного виразу.

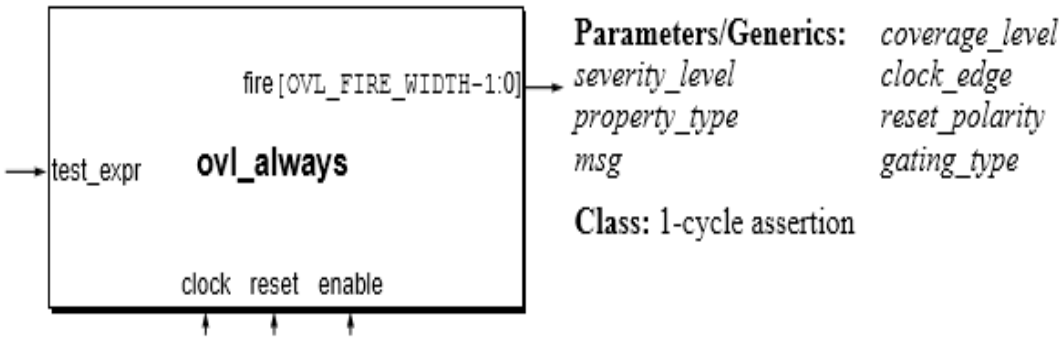


Рисунок 2.2- Схема модулю ovl_always з бібліотеки OVL[8].

Модуль ovl_always перевіряє що на кожному фронті синхросигналу clock однобітовий вираз test_expr дорівнює логічній одиниці.

2.2.4 Перевірка поведінки

В [13] стверджується, що всі технології для задання асерцій мають схожі можливості, проте відрізняються у певних деталях, а також у способах реалізації в різних програмних інструментах. У табл. 1.1 наведено порівняльні характеристики трьох технологій: SVA, PSL та OVL.

Як було сказано у розділі 2.3.3 OVL представляє собою набір попередньо визначених модулів, які можна підставляти в будь-якому місці дизайну для досягнення бажаного асерційного ефекту. Так як дана технологія не являє собою розширення будь-якої мови, виникають певні синтаксичні та семантичні обмеження. Таким чином, подальші вдосконалення будуть являть собою додавання нового порту або параметру, значення яких буде важко зрозуміти без відповідної документації.

PSL представляє собою зовсім окрему мову. Було неможливо реалізувати його за допомогою засобів мови Verilog. З цієї причини неможливо використовувати конструкції PSL явно в RTL описі. Натомість використовується механізм прагм (коментарії), який дозволяє зберігати незалежність інструментарію від підтримки асерцій.

SystemVerilog представляє собою розширення мови Verilog з можливостями написання асерцій та підтримкою функціонального покриття. Це дозволяє як проектувальникам, так і QA-інженерам створювати гнучкий та інструментально незалежний дизайн без необхідності додавання специфічних прагм. Це чистий HDL з можливостями специфікацій веріфікаційного рівня.

Таблиця 2.1 – Порівняння різних технік для написання асерцій [5]

Параметри	SVA	PSL	OVL
Декларація властивостей	Так	Так	Так
Перевірка властивостей	Так	Так	Так
Специфікація обмежень	Так	Так	Ні
Секвенційний механізм	Так	Так	Так
Функціональне покриття	Так	Так	Ні
Синхронізація	Так	Так	Так
Негайні асерції	Так	Ні	Ні
Декларативні асерції	Так	Так	Ні
Програмні асерції	Так	Ні	Так
Вбудовані функції	Так	Так	Ні
Складність імплементації	Легка	Легка	Важка

2.3 Сфери використання ABV

Джерело [15] вносить поняття критичних точок верифікації, які представляють собою певний набір проблем, які важко або неможливо перевірити за допомогою традиційних підходів. Важкість верифікації пояснюється великою кількістю послідовних станів. Така точка може знаходитися десь у глибині дизайну та бути абсолютно неконтрольованою за допомою входів системи. Таким чином, її неможливо перевірити за допомогою моделювання.

Цінність критичних точок полягає у явному визначенні знань, як ефективно використовувати формальну верифікцацію та моделювання для повного тестування певного логічного блоку [3]. Таким чином експертні знання одного інженера передаються іншому. Ці знання включають в себе: написання асерції для точки, правильне використання формального аналізу та моделювання, методи для обмеження вхідних даних, метрику для оцінювання точки [9].

2.3.1 Логіка контролю за ресурсами

Обчислювальні ресурси, шини систем на кристалі, буфери та пам'ять представляють собою логічні структури, які контролюються арбітрами або комплексною логікою для контроля. Створюючи конкретні тести (directedtests), QA-інженери фокусують увагу на специфікації. Дуже рідко вони покривають граничні випадки такої логіки, та, як результат, виникають проблемні місця. Далі наведено властивості, яким повинна задовольняти дана частина системи.

1. Забезпечення правильності генерації запитів в залежності від маски, ваги або схеми кредитування. Це робиться за допомогою створення додаткової логіки, яка асоціюється з асерцією.

2. Коректність схеми арбітражу. Для простих схем це можна забезпечити за допомогою arbiter-чекеру з бібліотеки асерцій.

3. Ресурси (пам’ять, шина) адресуються тільки одним користувачем. Це можна перевірити за допомогою асерційної мови або mutex-чекером з бібліотеки.

Якщо цільова логіка для перевірки знаходиться в окремому молулі та незалежна, то формальна перевірка може бути застосована на ранніх етапах верифікаії.

В більшості систем запит на ресурси можна робити в будь-який час та в будь-якій послідовності. Перевірка всіх сценаріїв неможлива. Натомість, описуючи інваріанти середовища у вигляді асерцій, формальний інструментарій може проаналізувати практично всі можливі випадки та виявити потенційний конфлікт за ресурси.

2.3.2 Інтерфейс дизайну

Міжмодульна комунікація та сумісність протоколів взаємодії є причиною багатьох помилок. Для спрощення інтеграції окремих частин дизайну можна використовувати асерційні протоколи, які будуть перевіряти сумісність інтерфейсів. Для цього система має задовольняти певним властивостям.

1. Правильність функціонування протоколів на всіх інтерфейсах, особливо під час різних операцій.

2. Прозорість інтерфейсів. Ця властивість гарантує цілісність транзакцій при передачі між різними модулями. Цю потребу можна виразити за допомогою асерційних мов з використанням темпоральних властивостей.

Публікація [8] стверджує, що монітори, написані з використанням асерцій можуть викоритовуватися як під час формальної верифікації, так і під час моделювання. Такий підхід дозволяє збирати статистичну інформацію, яка показує ретельність генерації та перевірки згенерованих транзакцій.

Для валідації прозорості інтерфейсів необхідно запевнитися, що транзакції інтерпретуються та передаються правильно. Моделюванням з асерціями можна перевірити транзакцію з відношенням один до одного. Однак для повного

покриття необхідно перевірити, що помилкові, несправні та повторні транзакції оброблюються коректно.

2.3.3 Дискретні автомати (FSM)

З точки зору верифікації в [5] виділяється два типи дискретних автоматів: інтерфейсні та комп'ютаційні. Інтерфейсні автомати використовують вхідні та вихідні сигнали з певними часовими вимогами. Як правило мірою часу в таких автоматах є такти сихнхросигналу. Прикладами таких автоматів є контролери шин, автомати для синхронізації процесів (handshake FSM), тощо. Комп'ютаційні автомати не використовують сигнали з чітко виділеними часовими потребами: алгоритмічні обчислювання, операційні та керуючі автомати.

Якісний опис автомату – відповідність HDL коду специфікації. Типові властивості якісного автомату:

1. Відповідність HDL автомату опису часовим обмеженням. Під час верифікації небхідно підтвердити, що FSM зчитує вхідні сигнали у відповідні періоди, а поява вихідних сигналів відповідає специфікації. Це можна виразити за допомогою асерційної мови, використовуючи темпоральні властивості.

2. Відповідність HDL опису автомату граф-схемі алгоритму (ГСА). Дуже часто для спрощення імплементації великі ГСА розбивають на частини, що відповідають різним автоматам. Така декомпозиція не повинна коректно відображати цільову ГСА. За допомогою асерцій можна перевіряти правильність переходів, вихідних сигналів, умов переходів.

Під час моделювання асерції моніторять дії всередині дизайну та представляють інформацію про ретельність тестового середовища. За допомогою вираження властивостей автомату асерціями можна більш детально дослідити поведінку FSM та зібрати інформацію про тестове покриття під час моделювання [1,5].

Під час взаємодії декількох автоматів необхідно впевнитися, що вони не потрапляють у граничний випадок. Для перевірки такої поведінки можна використовувати формальну верифікацію. Проте формальний аналіз не буде ефективним для складних властивостей, які необмежені у часі. Таким чином високорівневі часові потреби автомату необхідно декомпозувати та перевіряти декількома асерціями.

3 СЕМАНТИКА АСЕРЦІЙНОЇ ВЕРИФІКАЦІЇ

3.1 Поняття властивості

Властивість – певний атрибут, який використовується для характеристики дизайну. При вивчені властивостей виділяють 4 відокремлені рівні.

– Логічний рівень, який складається з логічних виразів (VHDL або Verilog виразів).

– Темпоральний рівень, котрий описує відносини логічних виразів у часі.

– Рівень моделі, який надає засоби моделювати складну поведінку вхідних ті вихідних сигналів, а також допоміжну логіку, яка не є частиною дизайну, але часто є необхідною для захоплення вимог вищих рівнів.

– Рівень верифікації, який описує, як використовувати властивість під час верифікації.

Таке ділення на частини дозволяє розбирати та обговорювати різні аспекти властивостей дизайну.

3.1.1 Логічний рівень

Логічний рівень властивості складається з логічних виразів, які в свою чергу складаються з сигналі дизайну. Наприклад, твердження “сигнали en1 та en2 взаємно виключні” (тобто активним може бути тільки один з двох) вираженоза допомогою мови VHDL на рис. 3.1.

```
1 not( en1 and en2 )
```

Рисунок 3.1 – Вираження логічного рівня властивості на мові VHDL

3.1.2 Темпоральний рівень

Темпоральний рівень дозволяє описувати відношення логічних виразів у часі. Таким чином, часові неоднозначності, які відносяться до властивості повність ліквідуються. Наприклад, якщо сигнали en1 та en2 взаємно виключні увесь час, то можна додати темпоральний оператор аби зазначити це явно. Темпоральні оператори дозволяють точно зазначити, коли саме логічний вираз має виконуватися.

3.1.3 Рівень верифікації

Темпоральний та логічні рівні описують загальну поведінку, але не описують як саме властивість має бути задіяно під час верифікації. Можливо три варіанти використання властивості.

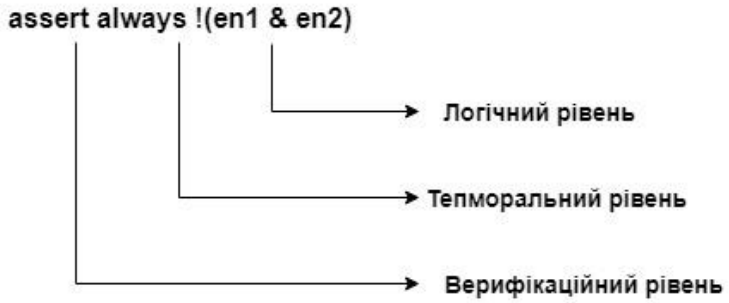


Рисунок 3.2– Три рівні властивості

1. Використання властивості у якості асерції, тобто виконання даної властивісті має виконуватися для даного дизайну.

2. Використання властивості у якості обмеження (constraint). Обмеження - умова (зазвичай на вхідні сигнали), яка обмежує множину поведінки, яку необхідно брати до уваги під час верифікації. Обмеження може представляти реальні вимоги (вимоги до синхросигналу) середовище, у якому працює дизайн, або штучні обмеження (конфігурації), які використовуються для спрощення верифікації.

3. Використання властивості у якості події для збору інформації про функціональне покриття.

3.2 Поняття асерції

Асерція – описання певної очікуваноїповедінки цифрової системи під час її функціонування. В більшості випадків асерція представляє собою HDL код, який не несе додаткової функціональності для цільової системи та не синтезується.

Основним джерелом властивостей є функціональна специфікація. Асерція – спосіб вираження властивості за допомогою HDL конструкцій. У подальшому асерції використовуються під час процесу моделювання або при верифікації за допомогою формальних методологій.

3.3 Види асерцій

Як правило модель дизайну складається зі статичних ієрархічних структур, в яких примітивні елементи взаємодіють між собою за допомогою мережі з'єднань. Примітиви можуть бути вбудованими простими функціями (наприклад, вентилі), або більш великим процедурним або алгоритмічними описами (VHDL процеси, конструкція Verilog always). В процедурному описанні інструкції виконуються послідовно, а у всьому дизайні примітиви та комунікації взаємодіють паралельно.

Так само як дизайн складається з паралельних (задекларованих) та послідовних (процедурних) блоків, властивості можуть бути представлені декларативно або процедурно. Таким чином, декларативна асерція конструкція, яка завжди активна та виконується (обчислюється) паралельно з іншими рівнями або примітивами у дизайні. Процедурна асерція конструкція в контексті послідовного блоку, наприклад конструкції VHDL process, яка виконується послідовно разом з іншими інструкціями блоку.

3.4 Переваги використання асерцій

3.4.1 Покращення пропагації несправностей

Концепція пропагації – одна з основних концепцій верифікації цифрових систем.

Як було сказано у розділі 2, основною задачею тестбенчу є наступне:

– генерація вхідних наборів відповідно до певних вимог;

– валідація вихідних сигналів шляхом самоперевірки.

Для ідентифікації несправної поведінки достатньо та необхідно виконання наступних умов:

– генерація вхідного вектору сигналів для збудження несправності;

– генерація вхідного вектору сигналів, які б забезпечили появу несправності на вихідних портах дизайну або її місце в HDL коді.

Виконання першої умови не завжди забезпечує виконання другої. У такому випадку інженер може просто не помітити несправності.

Використання асерцій забезпечує виконання другої умови, адже порушення властивості проявляється безпосередньо біля її джерела в HDL описі.

3.4.2 Зменшення часу відладки

Аналіз часових діаграм не є ефективним способом пошуку несправностей. Особливо це стосується великих дизайнів з сотнями тисяч сигналів. Необхідно мати динамічний та автоматизований спосіб моніторингу несправної поведінки.

Окрім того, традиційні методології верифікації не локалізують місце несправності. Як правило, несправна поведінка проявляється через певний час без будь-якої прив’язки до рядка або файла HDL опису. Для локалізації проблемного місця інженер повинен проаналізувати всі можливі місця, які потенційно можуть збуджувати таку поведінку.

Використання асерції дозволяє вирішити ці проблеми, тому що порушення асерційного виразу завжди повідомляє інженеру місцезнаходження даного порушення.

3.4.3 Покращення інтеграції систем

Однією з областей використання асерцій є розробка компонентів інтелектуальної власності (IP).

Команда проектувальників затверджує IP з певним набором функціональних обмежень та підтверджує його певними асерціями, які дозволяють моніторити правильну комунікацію інтерфейсів під час верифікації інтеграції. Такі асерції формують верифікаційні погодження між постачальником IP та її користувачем завдяки перевірці правильного користування. Такий підхід дозволяє суттєво полегшити відладку IP компонента у разі його неправильного використання, тобто порушення обмежень інтерфейсу. Це судження ілюструє рис. 3.3.

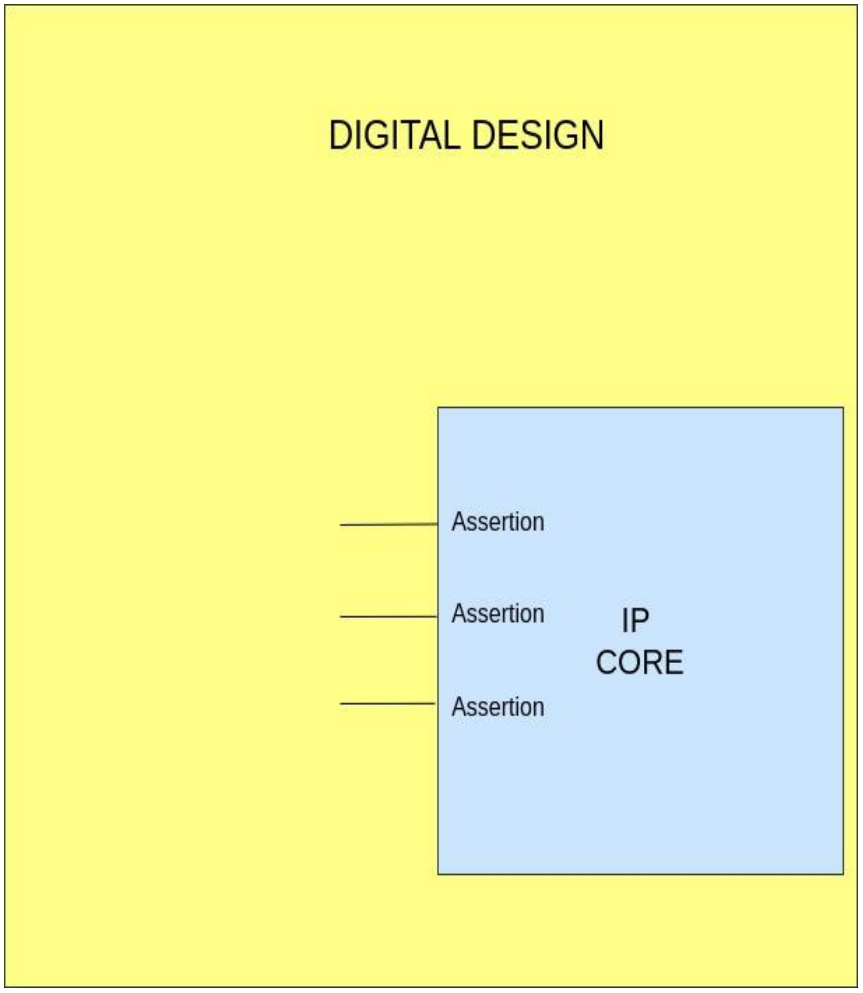


Рисунок 3.3– Використання асерцій для коректної інтеграції з IP компонентами

3.4.4 Покращення процесу верифікації

Найголовнішою перевагою використання асерцій порівняно з іншими методологіями є значне зменшення часу відладки, адже тепер інженер не має аналізувати великі часові діаграми для знаходження та локалізації несправності.

На відміну від традиційного процесу відладки асерції завжди моніторять коректну або некоректну поведінку. Наприклад, в традиційному процесі відладки проектувальник робить 4 основних кроки:

1. дослідження проваленого тесту;
2. виявлення проблемної поведінки;
3. виправлення помилки в HDL описі;
4. повторний запуск тесту для валідації поведінки.

Такий процес повторюється послідовно для всіх помилок в дизайні. Проте існують так звані особливі випадки (corner cases), які не завжди є очевидними. Такі випадки можуть повторно відтворювати проблему, яка вже була покрита тестом.

Асерції дозволяють уникнути такої проблеми, тому що асерції ніколи не припиняють моніторити дизайн на факт несправної поведінки та можуть допомогти локалізувати багато особливих випадків під час майбутніх сесій моделювання. Таким чином, новий набір тестових послідовностей може викрити додаткові проблеми, які, здавалось би, було вже виправлено.

4 АБСТРАКТНИЙ ЦИФРОВИЙ АВТОМАТ

4.1 Поняття цифрового автомата

Дискретний автомат – математична абстракція, що представляє собою певний процес обробки інформації або управління.
Математично автомат представляє собою шестикомпонентний вектор[1]:

$$S =< A, X, Y, \delta, \lambda, a1 > \quad (4.1)$$

де A{a_1,…., a_m, …. A_M} – множина станів (алфавітвнутрішніхстанів);

X = {x_1,….., x_f, ….. x_F} – множина вхідних станів (вхідний алфавіт);

Y = {y_1, …., y_g, …. Y_G} – множина вихідних станів (вихідний →алфавіт);

δ: A × X→A – функція переходів, що реалізує сюр'єктивне відображення:
A × X на A;

λ: A × X→Y –функція виходів, що реалізує сюр'єктивне відображення:
A × X: Y;

a є A – початковий стан автомата.

В момент дискретного часу tавтомат знаходиться в певному стані a(t) з множини внутрішніх станів A, причому в момент часу t=0 автомат завжди знаходиться у початковому стані a_1.

4.2 Типи дискретних автоматів

Існує два основних типи абстрактних автоматів: автомат Мілі та автомат Мура.

4.2.1 Автомат Мілі

Функціонування автомата Мілі задається наступними рівняннями:

$$A(t+1)= [a(t), x(t)] \quad (4.2)$$

$$y(t)= [a(t), x(t)] \quad (4.3)$$

З формул 4.2 и 4.3 видно, що наступний стан автомата Мілі залежить від поточного стану та вхідних сигналів, а вихідні сигнали залежать від поточного стану та вхідних сигналів. На рис. 4.1 зображено граф переходів такого типу автоматів.

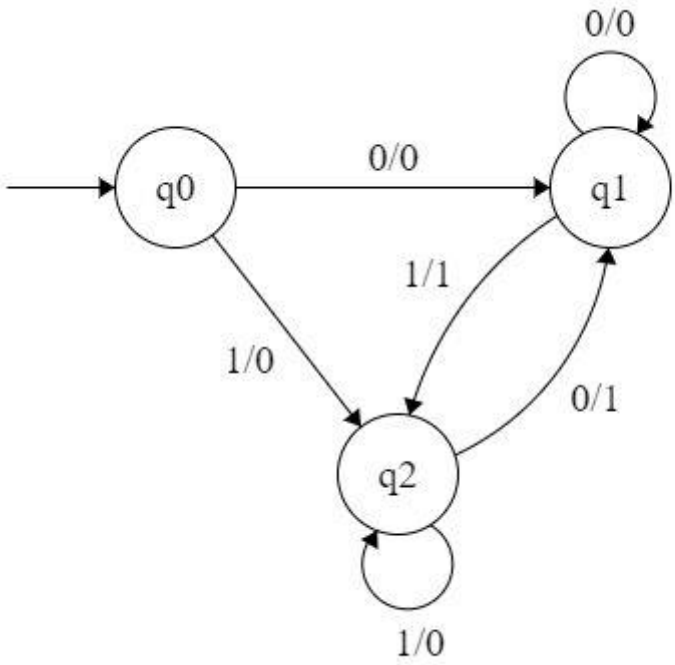


Рисунок 4.1– Граф переходів автомата Мілі з одним вхідним та одним вихідним сигналом

4.2.2 Автомат Мура

Функціонування автомата Муру описується рівняннями 4.4 та 4.5.

$$A(t+1)= [a(t), x(t)] \quad (4.4)$$

$$y(t)= [a(t)] \quad (4.5)$$

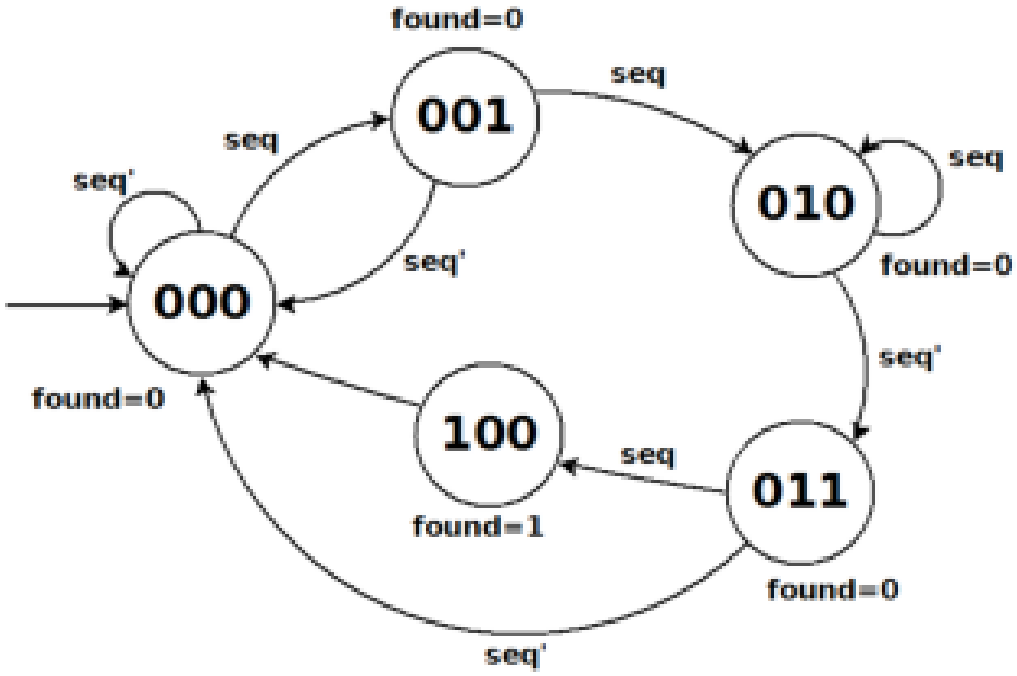


Рисунок 4.2 – Граф переходів автомата Мура

Як видно з рівняння 3.5 на відміну від автомата Мілі виходу автомата Мура залежать лише від поточного стану. На рис. 4.2 зображено граф схему автомата Мура.

4.3 Семантика HDL - опису автомата

Одним з найбільш поширеніших способів опису автоматів є апаратна реалізація алгоритму функціонування за допомогою мов опису апаратури (HDL) та його подальший синтез за допомогою системи автоматизованого проектування в ПЛІС. Перевагою такого підходу є апаратна гнучкість, швидкодія та швидкість імплементації.

Певний спосіб HDL - опису має назву автоматний шаблон. На рис. 4.3 зображено приклад трьохпроцесорного опису на мові SystemVerilog.

```
module statem(clk, in, reset, out);

  input clk, in, reset;
  output [3:0] out;

  reg [3:0] out;
  reg [1:0] current_state, next_state;

  parameter S1=0;
  parameter S2=1;
  parameter S3=2;
  parameter S4=3;

  // Логіка включення
  always @(posedge clk)
    begin
      if(reset)
        current_state <= S1;
      else
        current_state <= next_state;

    end
  // Логіка виходів
  always @(next_state)
    begin
      case (next_state)
        S1: out = 4'b0000;
        S2: out = 4'b0001;
        S2: out = 4'b0010;
        S3: out = 4'b0100;
        default: out = 4'b0000;
      endcase
    end
  // Логіка зміни станів
  always @(next_state, in)
    begin
      case (next_state)
        S1: next_state = S2;
        S2: next_state = in ? S1 : S3;
        S3: next_state = S4;
        S4: next_state = S1;
      endcase
    end

endmodule
```

Рисунок 4.3 –Приклад трьохпроцесного опису автомата на мові SystemVerilog

4.4 Семантика кодування автомата

Існує декілька способів кодування кінцевих автоматів. На кінцеву стратегію впливають наступні фактори:

– вимоги до швидкодії проектованої системи;

– вимоги до площі кристалу;

– вимоги до енергоспоживання;

– тип ПЛІС.

Вибір стратегії кодування є важливим етапом проектування, яке роблять інженери для задоволення нефункціональних вимог. Варто відзначити, що не існує універсального алгоритму для визначення способу кодування.

Так як кодування впливає на подальший результат синтезу, а також швидкодію та розміри кристалу – даний аспект важливо перевірити під час верифікації. Зокрема, це можна зробити за допомогою асерцій.

Далі буде розглянуто найпоширеніші алгоритми кодування та рекомендації щодо їх використання.

4.3.1 Унітарне кодування

Унітарне кодування (One Hot Encoding) – стратегія кодування, при якій бінарний код кожного стану містить тільки одну одиницю–прямий унітарний код або тільки один нуль– зворотний (інверсний) унітарний код. Таким чином, кожному стану буде відповідати окремий тригер. Для n станів знадобиться n тригерів.

При використанні такого типу кодування спрощується процес дешифрації станів, тому що тригери показують, у якому стані зараз знаходиться автомат.

Унітарне кодування добре підходить для архітектур з обмеженою кількістю входів та великою кількістю тригерів. Прикладом таких пристроїв є FPGA від фірм Xilinx, Actel та інших.

Завдяки відносно малій кількості комбінаційних елементів між тригерами підвищується швидкодія автомата, зменшується його площа та, як наслідок, споживання енергії. Крім того, швидкість не залежить від кількості станів, а залежить від кількості переходів в певний стан.Автомати з унітарним кодування легше імплементувати за допомогою HDL, так як переходи між станами та функцію виходів можна виразити безпосередньо з графу переходів.

В деяких випадках унітарне кодування може бути неефективним. Перш за все це стосується автоматів за невеликою кількістю станів. В такому випадку можлива ситуація, при якій очікувана швидкість від мінімального використання комбінаційних елементів може бути перекритою неефективним використанням програмованих логічних блоків.

4.3.2 Бінарне кодування

При використанні бінарного кодування відношення між кількістю тригерів та станів автомата можна виразити за допомогою формули 3.1, де q – кількість тригерів, n – кількість станів.

$$q = \log_2(n) \qquad (4.6)$$

При цьому підході стани кодуються двійковим кодом – від 0 до n - 1. В таблиці 4.1 приведено такого кодування для автомата з 4 станами.

Таблиця4.1 – Бінарне кодування станів для автомата з 4 станами

Стан	Y0	Y1
S1	0	0
S2	0	1
S3	1	0
S4	1	1

Бінарне кодування потребує менше тригерів, ніж унітарне. Наприклад, для синтезуавтомата зі 100 станами дана техніка потребує 7 тригерів, в той час як унітарне кодування – 100. Дана техніка максимально використовує наявні ресурси регістрів. Як наслідок, зростає кількість комбінаційної логіки для

дешифрації станів. Таким чином, бінарне кодування краще всього використовувати з пристроями типу PLA та CPLD, тому що в них велика кількість комбінаційних елементів. Як правило, такий тип кодування використовується при кількості станів, менших 8.

Головним недоліком цієї техніки є потенційна можливість випадкової зміни будь-якого біта регістру стану – як наслідок, неправильна поведінка. Якщо споживання енергії є критичною вимогою, то така техніка теж не підходить, тому що чим частіше змінюється регістр стану, тим більша використана енергія.

4.3.3 Кодування за допомогою коду Грея

Код Грея – спосіб кодування, при якому два послідовні коди відрізняються значенням лише одного біта. Іншими словами, при переході між двома сусідніми станами, змінюється лише один тригер. В таблиці 3.2 приведено кодування для автомата з 4 станами.

При кодуванні за допомогою коду Грея необхідно стільки ж тригерів, скільки при бінарному кодуванні. Як бінарне кодування, дане кодування ідеально підходить для PLA та CPLD з великою кількістю комбінаційних елементів.

Таблиця4.2 – Кодування за допомогою коду Грея для автомата з 4 станами

Стан	Y0	Y1
S1	0	0
S2	0	1
S3	1	1
S4	1	0

Мінімальна зміна окремих бітів регістру стану забезпечує мале споживання енергії. Ідеальною є ситуація, коли будь-який перехід збуджує зміну лише одного тригера.

В таблиці 4.3 наведено приклад кодувань для автомата з 5 станами.

Таблиця4.3 – Кодування за допомогою різних технік для автомата з 5станами

Стани	Унітарний код	Бінарний код	Код Грея
S0	00001	000	000
S1	00010	001	001
S2	00100	010	011
S3	01000	011	010
S4	10000	100	110

4.5 Темпоральні характеристики автоматних моделей

Відомо, що пристрої логічного управління, побудовані на основі автоматного шаблону, функціонують в автоматному часі. Під автоматним часом розуміються дискретні відрізки часу, за які автомат переходить з одного стану в інший. Тривалість автоматного такту, як правило, визначається частотою синхросигналу. Проте з іншої сторони, пристрої логічного керування представляють собою системи керування реального часу, і переходи між станами визначаються часовими параметрами алгоритму функціонування.

Для опису автоматних систем реального часу набули широкого використання автоматні моделі реального часу [17]. В реалізації такого автоматного часу кожному стану ставиться у відповідність певний інтервал метричного (безперервного) часу. Переходи в такому автоматі відбуваються

миттєво, проте автомат не переходить у новий стан доки не закінчиться інтервал заданого метричного часу. Дана модель відповідає дискретному автомату Мура.

Для внесення реального часу в опис структурного автомата використовується розширена функція переходів 4.7, в якій аргументом виступає реальний час.

$$Z(t + 1) = f(X(t), Z(t), T) \qquad (4.7)$$

Граф, який описує функціонування автомата реального часу, називається темпоральним графом переходів. В такому графі кожна вершина має затримку T, впродовж якої автомат знаходиться в даному стані. Затримку в кожній вершині темпорального графу реалізують за допомогою петлі, умовою для якої є підрахунок числа тактів синхросигналу. В ПЛІС це реалізується за допомогою лічильників, в мікроконтролерах –- за допомогою таймерів з перериванням. На рис 4.5 приведено відповідність умовного позначення затримок у вершинахтемпорального графу автомата Мура без петлями та без петель.

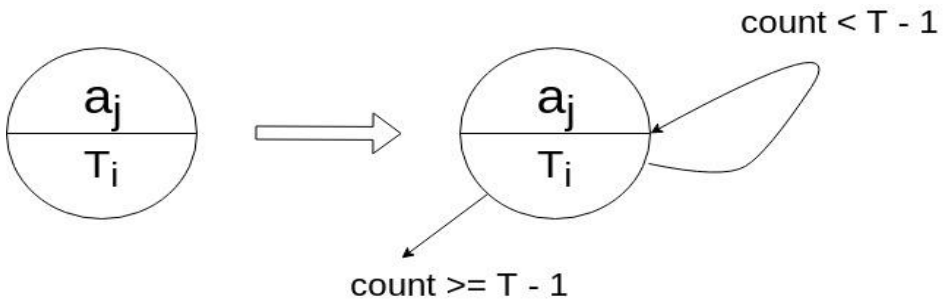


Рисунок 4.5– Реалізація затримок в темпоральному графі автомата Мура

5 АСЕРЦІЙНА ВЕРИФІКАЦІЯ АВТОМАТІВ РЕАЛЬНОГО ЧАСУ

5.1 Семантика асерційної верифікації пристроїв реального часу

У розділі 4 було сказано, що головною особливістю верифікації автоматів, що моделюють пристрої реального часу, є моделювання метричного часу. Цей процес відбувається за допомогою знаходження автомата у певному стані протягом певного числа циклів синхросигналу. Іншими словами, це є певною властивістю (property) дизайну, його семантичною особливістю.

Такі вимоги до функціонування необхідно перевіряти на ранніх стадіях верифікації. Існує два шляхи перевірки таких вимог:

– перевірка за допомогою миттєвих асерцій – перевірка стану внутрішніх лічильників в момент переходів;

– перевірка за допомогою паралельних асерцій – вираження темпоральних властивостей за допомогою спеціальних мов (SVA, PSL).

Варто зауважити, що перший спосіб вимагає попередньо написаної тестової моделі. Другий спосіб може використовуватися, як з класичними підходами тестування (тестбенчі), так і з формальними середовищами верифікації.

5.2 Приклад автоматної моделі пристрою реального часу

Для ілюстрації вищезазначених способів верифікації розглянемо модель пристрою для логічного керування дорожнім світлофором. На рис.5.1 представлено граф переходів даного пристрою керування.

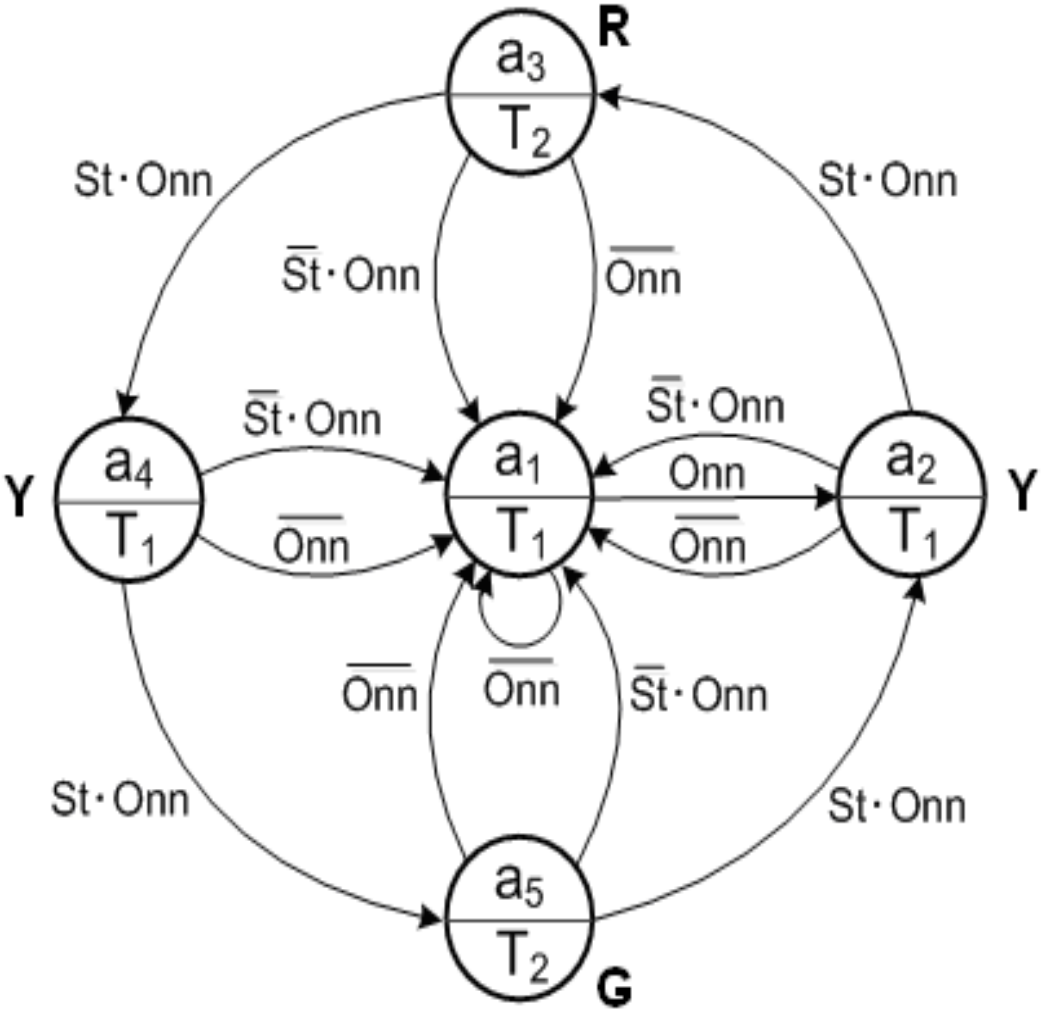


Рисунок 5.1 – Граф переходів автомата для керування дорожнім світлофором

Інтерфейс даного пристрою складається з наступних множин сигналів:

– множина вхідних сигналів {Onn, St}, де Onn {0, 1} – включення світлофора, St {0, 1} – сигнал запуску стандартного циклу роботи;

– множина вихідних сигналів {R, Y, G}, де R – сигнал включення червоного світла, Y – сигнал включення жовтого світла, G – сигнал включення зеленого світла.

Всі сигнали є однорозрядними.

Даний пристрій має два цикли роботи.

1. Нічний (аварійний) – блимання жовтим кольором з інтервалом T1.

2. Стандартний цикл роботи {Y - R - Y - G -Y}. В такому циклі роботи затримка в станах R та G буде T2 тактів, а в стані Y – T1.

5.3 SystemVerilog модель автомата

Головною особливістю HDL-реалізації автоматних моделей реального часу є наявність лічильників. Дані лічильники підраховують кількість тактів, впродовж яких автомат перебуває в певному стані. Для даної моделі T_1 дорівнює 1 такту, T_2– 5 тактам.

На рис. 5.2 наведено приклад декларації даних об’єктів на мові SystemVerilog.

```
reg [2:0] count, count1;
parameter T1 = 3'b010;
parameter T2 = 3'b101;
```

Рисунок 5.2 – Декларація лічильників та затримок

При переходах між станами необхідно враховувати значення лічильників. Дана перевірка виражає умову реального часу. На рис. 5.3 наведено приклад реалізації такого переходу між станами a2 та a3.

```
a2: begin
  if(count < T1 - 1) begin
    nextState = a2;
    count1 = count1 + 1'b1;
  end
  else if(!onn || !st) begin
    nextState = a1;
    count1 = 3'b000;
  end
  else
    begin
      nextState = a3;
      count1 = 3'b000;
    end
end
```

Рисунок 5.3 – Приклад реалізації переходів між двома станами з урахуванням значень лічильника

На рис.5.4 наведено часову діаграму моделювання данного автомата.

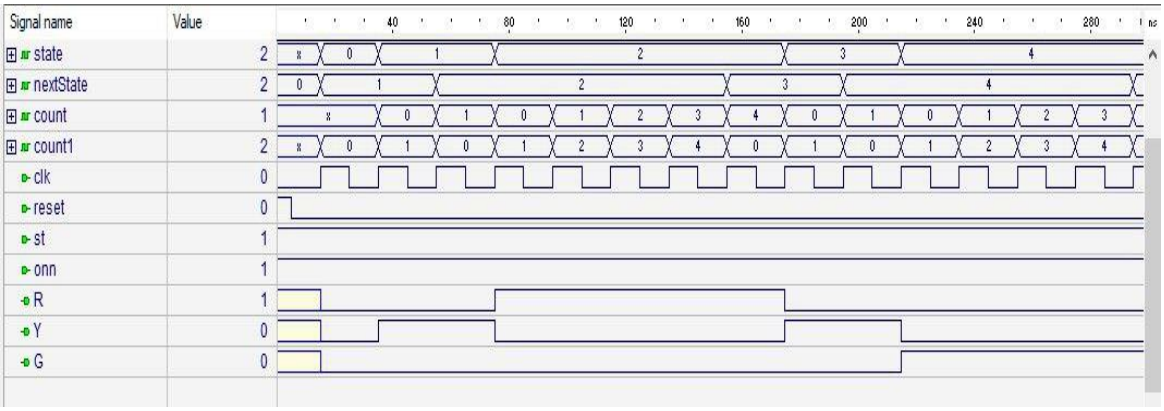


Рисунок 5.4 – Часова діаграма моделювання автомата

5.3 Верифікація темпоральних характеристик за допомогою миттєвих асерцій

Верифікації за допомогою миттєвих асерцій полягає у перевірці значення регістру стану впродовж певного часу затримки. Впродовж цього часу (певної кількості тактів) регістр має зберігати одне значення. На рис. 5.5 наведено приклад реалізації такої перевірки на мові SystemVerilog.

Під час кожного переднього фронту синхросигналу необхідно перевірити, що автомат знаходиться у бажаному стані та внутрішній лічильник знаходиться у початковому стані. Ця перевірка показує, що автомат щойно перейшов в цей стан. Далі необхідно перевірити, що стан не змінюється впродовж N-1 тактів. Для цього використовується конструкція assert.

```
always@(posedge clk) begin
  if(inst.count == 0 && inst.state == a2)
    repeat(inst.T1 - 1) @(posedge clk)
      assert(inst.state == a2);
end
```

Рисунок 5.5 – Верифікація темпоральних характеристик автомата за допомогою мови SystemVerilog

У разі порушення даної умови середовище моделювання видасть відповідне повідомлення. Приклад такого повідомлення наведено на рис. 5.6.

```
# KERNEL: Error: Immediate assert condition (inst.state==a2) FAILED at time: 75ns, testbench.sv(19),
# KERNEL: Error: Immediate assert condition (inst.state==a2) FAILED at time: 355ns, testbench.sv(19),
```

Рисунок 5.6 – Приклад повідомлення від середовища моделювання

Даний метод має ряд недоліків:

– неявність вираження тепмпоральної властивості;
– неможливість використання такого підходу з формальними середовищами верифікації;
– відсутність способу перевірки активізації верифікаційний коду.

5.4 Верифікація темпоральних властивостей за допомогою паралельних асерцій

Опис темпоральної властивості буде робитися за допомогою наступних конструкцій мови SystemVerilog Assertions (SVA):

– sequence (послідовність) – темпоральна послідовність логічних виразів;
– property (властивість) – верифікаційна конструкція, яка описує бажану поведінку дизайну та складається з конструкцій sequence, об’єднаних спеціальними операторами;
– assert property – оператор перевірки темпоральної властивості;
– cover property – конструкція перевірки активації задої властивості.

Бажану темпоральну властивість можна описати натуральною мовою наступним чином: якщо автомат перейшов зі стану a_1 в стан a_2, то він має перебувати у стані a_2 N-1 такт та перейти у стан a_3. Іншими словами властивість складається з трьох логічних частин: передумова, цикл та наслідок.

Рисунок 5.7 – Послідовність логічних подій для перевірки темпоральних властивостей автомата

```
sequence Red2YellowPrecondition;
  ($past(state) == a2) && (state == a3);
endsequence

sequence Red2YellowStay;
  (state == a3) [*4];
endsequence

sequence Red2YellowPostCondition;
  state == a4;
endsequence
```

Рисунок 5.8 – Складові темпоральної властивості на мові SVA

На рис. 5.8 представлено три складові перевірки темпоральної властивості між станами a_3 та a_4 на мові SVA.

Red2YellowPrecondition – описує передумову властивості. Вбудована функція $past повертає значення регістру state на попередньому такті. Для даної властивості попереднім станом має бути стан a_2.

```
property Red2Yellow;
  @(posedge clk) disable iff (!onn || !st || reset)
  Red2YellowPrecondition |=> Red2YellowStay ##1 Red2YellowPostCondition;
endproperty
```

Рисунок 5.9 –Конструкція property для перевірки темпоральних властивості

Red2YellowStay – описує затримку у стані a_3. Оператор [*4] вказує, що вираз (state == a_3) має повертати логічне значення “1” впродовж 4 тактів.

Red2YellowPostCondition – описує наступний стан автомата після затримки.

Далі необхідно об’єднати вищеописані послідовності у конструкцію property за допомогою спеціальних операторів. Приклад такої конструкції наведено на рис. 5.9.

Конструкція propery Red2Yellow встановлює відношення між трьома послідовностями. Під час кожного переднього фронту синхросигналу clk перевіряється послідовність-передумова Red2YellowPrecondition. Якщо ця умова виконалася – під час кожного наступного переднього фронту синхросигналу відбувається перевірка затримки у стані a_3 за допомогою послідовності Red2YellowStay. Останньою послідовністю у цій властивості є перехід у стан a_4– послідовність Red2YellowPostcondition.

На рис.5.10 червоним кольором позначено момент обчислення послідовності Red2YellowPrecondition, зеленим – Red2YellowStay, синім – Red2YellowPostcondition.

Варто зауважити, що ця властивість описує стандартний цикл роботи автомата, тобто коли сигнали onn та st мають логічне значення ‘1’. Окрім того, автомат у будь-який момент автомат може бути встановлений у початкове положення за допомогою сигналу сбросу reset. Усі ці умови виражаються за допомогою конструкції disable iff. Якщо виконається логічна умова у тілі disable iff – обчислення усіх послідовностей буде завершено.

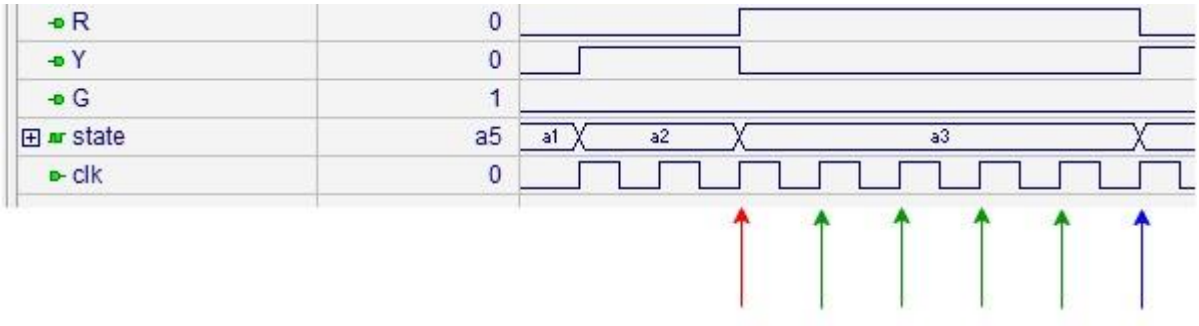


Рисунок 5.10 – Моменти обчислення трьох складових властивості

Далі необхідно задати перевірку властивості. Тому що властивість носить темпоральний характер – використовується механізм паралельних асерцій (concurrent assertion). Окрім ретельної верифікації асерції можуть надавати вичерпну інформацію щодо покриття функціональності. Для цих цілей використовується механізм cover property надасть наступну інформацію:

– кількість спроб вирахувати властивість;

– кількість успішних проходжень властивості;

– кількість невдалих проходжень властивості;

– кількість “вакуумних успіхів” властивості (не спрацювала послідовність Red2YellowPrecondition).

```
assert property (Red2Yellow);
cover property (Red2Yellow);
```

Рисунок 5.11 – Дефініція паралельної асерції

Використання механізму паралельних асерцій дає наступні переваги:

– явне вираження властивості дизайну завдяки прив’язці до подій синхросигналу як засобу вираження часу;

– можливість використання як з динамічними, так і з формальними середовищами верифікації;

– можливість проаналізувати статистику спрацювань верифікаційного коду за допомогою інформації про покриття.

6 ФУНКЦІОНАЛЬНА ВЕРИФІКАЦІЯ БАЗОВИХ ВЛАСТИВОСТЕЙ ЦИФРОВОГО АВТОМАТА

6.1 Асерційна верифікація переходів

Найважливішоючастиною будь-якого HDL-опису цифрового автомата є реалізація його переходів. Саме ця частина виражає послідовність станів, у яких перебуває автомат, і, як наслідок, безпосередньо впливає на вихідні сигнали. Дуже важливо впевнитися, що переходи відбуваються у правильній послідовності та не порушують часових обмежень.Далі розглядається матричний спосіб верифікації переходів, що полягає у створені матриці переходів та подальшій побудові властивостей та асерцій на мові SVA.

6.2 Модель автомата для арбітражу ресурсом

Для ілюстрації даного способу верифікації розглянемо модель керуючого автомата, який виконує функції арбітра. У будь-який момент часу автомат може обслуговувати запит від трьох пристроїв. Для уникнення ресурсного голоду використовується алгоритм циклічного планування. Як тільки пристрій отримує доступ до ресурсу, він використовує його впродовж певного часу. Потім пристрій повідомляє про кінець своєї роботи та доступність ресурсу до нових запитів. На рис. 6.1 представлено граф-схему переходів даного пристрою керування

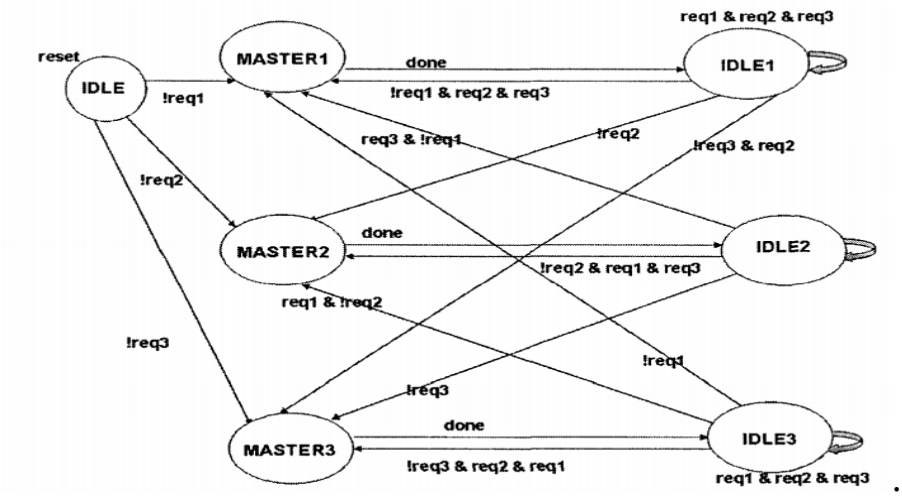


Рисунок 6.1 – Граф-схема автомата-арбітра

Як видно з рис. 6.1 автомат має наступні особливості:

– сигналом resetавтомат переводиться у початковий стан IDLE;

– знаходячись у стані IDLE, модель очікує на сигнал reqвід одного з клієнтів. Доступ до ресурсу надається за пріоритетом;

– знаходячись в одному зі станів MASTER*, автомат видає відповідний сигнал доступу до ресурсу gnt;

– по закінченню роботи з ресурсом автомат отримує від клієнта сигнал donета переходить у наступний IDLE* стан;

– знаходячись у стані IDLE1 автомат перевіряє сигнали req від клієнтів master2, master3, master1;

– знаходячись у стані IDLE2 автомат перевіряє сигнали req від клієнтів master3, master1, master2;

– знаходячись у стані IDLE3 автомат перевіряє сигнали req від клієнтів master1, master2, master3;

6.3 Матричний спосіб верифікації переходів

Автомат для арбітражу має 7 станів та 18 переходів. Для початку необхідно створити матрицю переходів. Дана матриця покаже можливі та неможливі

переходи. Якщо між двома станами є перехід на перехресті між цими станами у матриці буде “1”, в іншому випадку – “0”.

Таблиця 6.1 – Матриця переходів автомата для арбітражу

	IDLE	MASTERl	IDLE1	MASTER2	IDLE2	MASTER3	IDLE3
IDLE	1	1	0	1	0	1	0
MASTERl	1	1	1	0	0	0	0
IDLE1	1	1	1	1	0	1	0
MASTER2	1	0	0	1	1	0	0
IDLE2	1	1	0	1	1	1	0
MASTER3	1	0	0	0	0	1	1
IDLE3	1	1	0	1	0	1	1

Далі необхідно написати конструкції propertyна мові опису властивостей для кожного забороненого переходу.

Наприклад, для стану IDLE таку властивість натуральною мовою можна описати наступним чином: не можна переходити зі стану IDLEв стан IDLE1, IDLE2 або IDLE3. На рис. 6.2 представлено імплементацію цієї властивості на мові SVA.

```
property p_forbid_trans1;
@(posedge clk) disable iff (!reset)
    (state == IDLE1) || (state == IDLE2) ||
    (state == IDLE3) |->
        $past(state) != IDLE;
endproperty

assert property (p_forbid_trans1);
```

Рисунок 6.2 – Властивість та асерція для перевірки заборонених переходів на мові SVA

Для вираження даної властивості використовується оператор імплікації (|->). Якщо автомат знаходиться в одному зі станів IDLE1, IDLE2 або IDLE3, то під час наступного переднього фронту синхросигналу необхідно перевірити, що попереднім станом був стан, відмінний від IDLE (вираз праворуч від оператора |->).

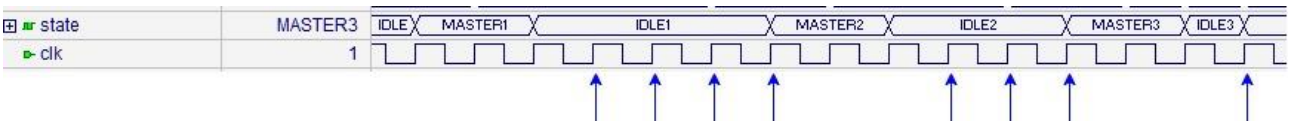


Рисунок 6.3 – Моменти обчислення властивості p_forbid_trans1

Аналогічним способом записуються інші властивості, які забезпечать перевірку правильності переходів під час моделювання. На рис. 6.4 наведено асерції та властивості для перевірки заборонених переходів між станами IDLE2, MASTER2, IDLE3, MASTER3 та MASTER1 (властивість p_forbid_trans2); IDLE1, IDLE3, MASTER1, MASTER3 та MASTER2 (властивість p_forbid_trans3); IDLE2, IDLE1, MASTER2, MASTER1 та MASTER3 (властивість p_forbid_trans4);

```
    property p_forbid_trans2;
        @(posedge clk) disable iff (!reset)
            (state == IDLE2) || (state == IDLE3) ||
            (state == MASTER2) || (state == MASTER3) |->
                $past(state) != MASTER1;
    endproperty

    property p_forbid_trans3;
    @(posedge clk) disable iff (!reset)
        (state == IDLE1) || (state == IDLE3) ||
        (state == MASTER1) || (state == MASTER3) |->
         $past(state) != MASTER2;
    endproperty

    property p_forbid_trans4;
    @(posedge clk) disable iff (!reset)
        (state == IDLE2) || (state == IDLE1) ||
        (state == MASTER2) || (state == MASTER1) |->
         $past(state) != MASTER3;
    endproperty

assert property (p_forbid_trans2);
assert property (p_forbid_trans3);
assert property (p_forbid_trans4);
```

Рисунок 6.4 – Властивості та асерції для перевірки з аборонених переходів

6.4 Регістрові виходи автомата

Одним з призначень абстрактних цифрових автоматів є подача управляючих сигналів на певний об’єкт керування. Така взаємодія визначає порядок (алгоритм) функціонування обчислювального або іншого пристрою.

Рисунок 6.5 – Взаємозв’язок між керуючим автоматом та об’єктом управління

Однією зі складових будь-якого автомата є вихідний алфавіт (Y). Правильна та своєчасна видача виходів є критичною для коректної роботи обчислювального об'єкта управління.

Як правило, типовий HDL-опис автомата складається з 2 або 3 процесів: процесс ініціалізації регістру поточного статну, процесс призначення наступного стану та процес ініціалізації виходів. Процесс, який відповідає за призначення виходів, має комбінаційний характер. Як наслідок, з цього випливають наступні проблеми:

– комбінаційні виходи можуть мати глітчі (короткочасні порушення) під час переходів;

– такі виходи споживають частини циклу синхросигналу, яка могла би бути доступною об’єкту керування;

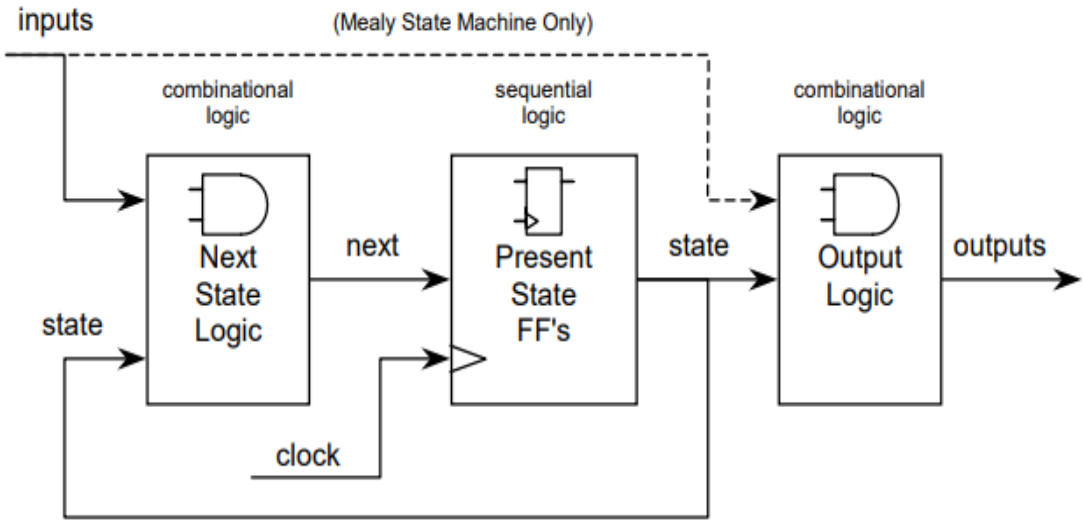


Рисунок 6.6 – Типова структура автомата

Об’єкт керування має менше часу для обробки вхідних сигналів та перш ніж вони будуть опрацьовані секвенційною логікою опису.

Для вирішення цієї проблеми існує техніка винесення усієї комбінаційної логіки, пов’язаної з виходами, на сторону входів об’єкта керування. Виходи автомата описуються як секвенційна логіка.

Головною ціллю такого підходу є значне полегшення задачі формування часових обмежень дизайну.

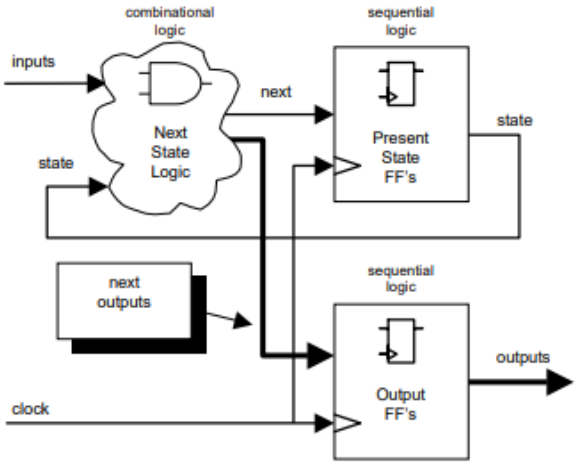


Рисунок 6.7 – Автомат з секвенційними виходами

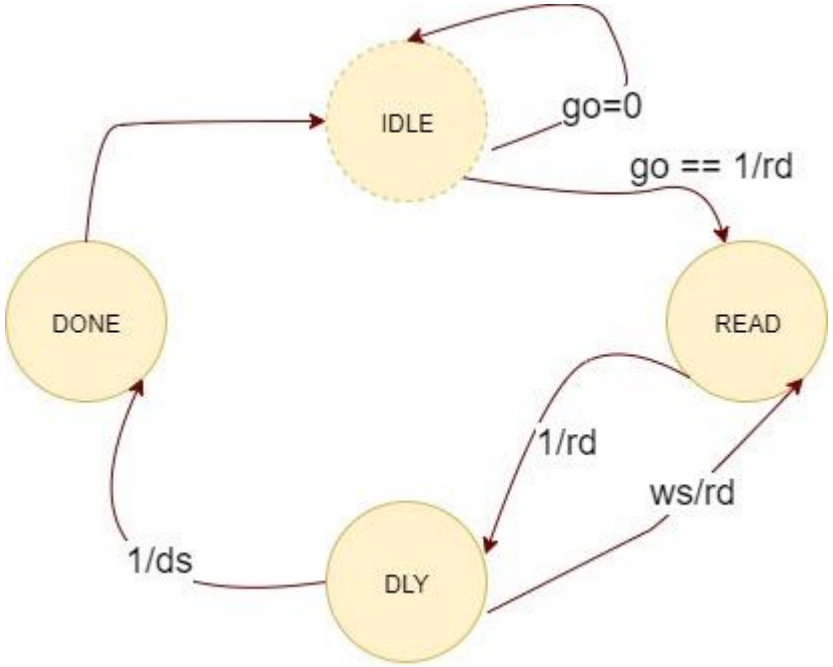


Рисунок 6.8– Граф переходів досліджуваного автомата

6.5 Асерційна верифікація регістрових виходів автомата

Явна прив’язка виходів то синхросигналу дозволяє додати додатковий рівень асерційної верифікації. Такі асерції мають певіряти наступне:

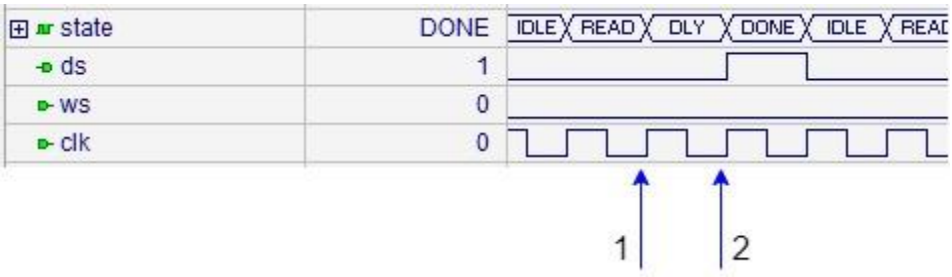
- вчасне спрацювання виході, якщо виконані відповідні умови;
- збереження значення вихідних сигналів впродовж певного часу;

Для наочності розглянемо автомат Мілі. На рис. 6.9 зображено його граф переходів, на рис. 6.9 – секвенційни процес на мові SystemVerilog, що описує регістрові виходи.

На етапі синтезу сигнали ds та rd будуть виходами тригерів з синхронізацією по передньому фронту та синхронним сбросом. Такі ж самі параметри має регістр стану автомата. Варто зауважити, що виходи будуть приймати активне значення рівно через такт після виконання відповідних умов стану та вхідних сигналів. Це ілюструє рис. 6.10.

```
always @(posedge clk)
    if (!rst_n) begin
            ds <= 1'b0;
            rd <= 1'b0;
        end
    else begin
            ds <= 1'b0;
            rd <= 1'b0;
            case (state)
                IDLE: if (go) rd <= 1'b1;
                READ: rd <= 1'b1;
                DLY: if (ws) rd <= 1'b1;
                    else ds <= 1'b1;
            endcase
        end
```

Рисунок 6.9– Секвенційний процес для регістрових виходів



6.10– Часова діаграма активізації виходу ds

На рис. 6.10 номером 1 позначено момент настання умови активізації виходу ds, номером 2– безпосередня активації даного виходу

Для уникнення затримки у такт в секвенційному процесі виходів замість поточного значення регістру стану (state) необхідно використовувати його значення на наступному такті (next_state).

```
always @(posedge clk)
    if (!rst_n) begin
            ds <= 1'b0;
            rd <= 1'b0;
        end
    else begin
            ds <= 1'b0;
            rd <= 1'b0;
            case (next_state)
                IDLE: if (go) rd <= 1'b1;
                READ: rd <= 1'b1;
                DLY: if (ws) rd <= 1'b1;
                    else ds <= 1'b1;
            endcase
        end
```

Рисунок 6.11 – Секвенційний процес виходів з використанням регістру next_state

Як видно з рис. 6.12 активізації виходу ds відбуєваться одочасно з переходом автомата у стан DLY.

Розуміння способу синхронізації виходів є важливим під час написання асерцій.

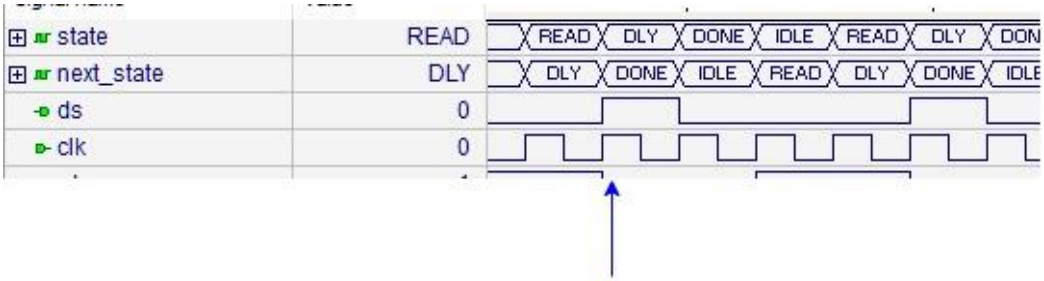


Рисунок 6.12– Часова діаграма активізації виходу ds

6.5.1 Семантика асерцій для регістрових виходів

Для описання властивості активізації виходів використовується шаблон наслідок – передумова. Натуральною мовою вона описується наступним чином: якщо під час переднього або заднього такту синхросигнала вихід *A* прийняв активне значення, то за такт до цієї події виконалася умова його активізації

(вхідні сигнали та регістр стану прийняли необхідні значення). Таке трактування подій дозволяє за допомогою однієї властивості перевіряти активізацію виходів тільки при виконанні описаних умов. На рис. 6.13 наведено властивість та асерцію для перевірки виходу rd.

```
property output_rd_prop;
@(posedge clk)
  rd |-> (($past(state) == READ) ||
          (($past(state) == IDLE) && ($past(go) == 1'b1)) ||
           (($past(state) == DLY) && ($past(ws) == 1'b1)));
endproperty

assert property (output_rd_prop);
```

Рисунок 6.13– Властивість та асерція на мові SVA для верифікації виходу rd

Дві частини властивості (передумова та наслідок) об’єднано за допомогою оператора імплікації (-|>). Якщо сигнал rdприйняв активне значення ‘1’ під час переднього фронту синхросигнала clk, то під час цього ж переднього фронту виконується перевірка правої частини виразу – значення регістру стану та вхідних сигналів. На рис. 6.14 зображено часу діаграму з позначками моментів обчислення властивості output_rd_prop.

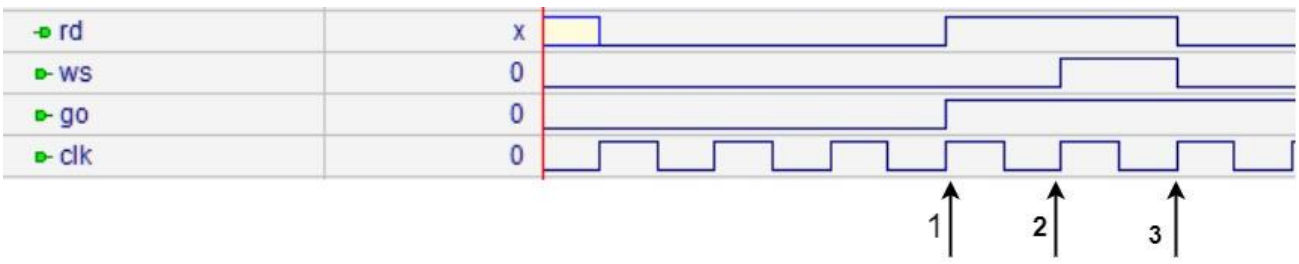


Рисунок 6.14 – Моменти обчислення складових властивості output_rd_prop

В момент переднього фронту під номером 2 буде виконається ліва частина властивості – сигнал rdприйняв активне значення. Далі властивість перевірить значення сигналів statета gов момент переднього фронту під номером 1.

Для секвенційного опису виходів, в якому використовується значення сигналу next_state, для опису властивості варто замінити сигнал state на next_state.

```
property output_rd_prop;
@(posedge clk)
  rd |-> (($past(next_state) == READ) ||
          (($past(next_state) == IDLE) && ($past(go) == 1'b1)) ||
           (($past(next_state) == DLY) && ($past(ws) == 1'b1)));
endproperty
```

Рисунок 6.15– Властивість для перевірки виходу rd для процесу без затримок

6.6 Асерційна верифікація кодування автомата

Існує декілька способів кодування автомата:

– ручне кодування станів відповідно до певних правил;

– кодування за допомогою спеціальних HDL конструкцій – прагм. Кожне середовище синтезу має свій набір прагм;

– автоматичне підбір оптимального кодування під час синтезу.

Дуже часто інженер використовує власне кодування, базуючись на переходах між станами. Наприклад, таке кодування може зменшувати кількість тригерів, що перемикаються під час переходу між станами. Насамперед, це робиться для подальшої мінімізації комбінаційної частини опису, зменшення споживання енергії або площі майбутнього кристалу.

Основна ідея написання асерцій для кодування полягає у явній верифікації намірів інженера при використанні певного способу кодування.

6.6.1 Верифікація унітарного кодування

У розділі 4.3 було сказано, що унітарне кодування дозволяє значно зменшити площу, енергоспоживання та кількість комбінаційних елементів. На рис. 6.16 показано два способи дефініції унітарного кодування станів на мові SystemVerilog.

```
typedef enum {IDLE = 1, READ = 2, DLY = 4, DONE = 8} t_state;

parameter IDLE = 4'b0001;
parameter READ = 4'b0010;
parameter DLY = 4'b0100;
parameter DONE = 4'b1000;
```

Рисунок 6.16 – Приклад унітарного кодування на мові SystemVerilog

Властивість унітарного кодування автомата можна виразити за допомогоюSVAабо іншої мови для опису властивостей. На рис. 6.18 зображено приклад такої властивості на мові SVA.

```
property one_hot_encoding;
    @posedge(clk) $onehot(state);
endproperty

property one_cold_encoding;
    @posedge(clk) $onehot(~state);
endproperty
```

Рисунок 6.17 – Приклад властивості для верифікації унітарного кодування SVA

Властивістьone_hot_encoding перевіряє, що під час кожного переднього фронту сигналу clkсигнал state містить рівно один біт зі значенням «1». Властивість one_cold_encoding робить таку ж саму перевірку для значення «0».

6.6.2 Верифікація кодування за допомогою коду Грея

Код Грея дозволяє значно зменшити енергоспоживання схеми за рахунок мінімального зміну стану тригерів під час переходів. Ідеальною є ситуація, коли кожен перехід змінює лише один біт у регістрі станів. Очевидно, що можлива кількість змінених бітів під час переходу визначається формулою:

$$G(a(t-1), a(t)) \in [0:\log_2(N)] \quad (6.1)$$

де G – функція, що визначає відстань Гемінга між двом станами;

a(t) – стан автомата у минулому такті;

a(t - 1) – стан автомата у поточному такті;

N – загальна кількість станів.

Очевидно, що кодування станів за допомогою коду Грея для виконання певного правила зміни бітів є нетривіальної задачею. Ця задача ускладняється при зростанні станів та переходів. Як правило, дана задача вирішується шляхом перебору декількох кодувань. Збір статистики зміни бітів динаміки значно спрощує аналіз та підбір оптимального коду.

6.6.3 Семантика конструкції covergroupмови SystemVerilog

Потужним інструментом мови SystemVerilog є набір конструкцій для сбору статистичної інформації про появу в ході моделювання значень різних об’єктів. Центральною конструкцією в такому підході є тип covergroup. Як правило декларація covergroup містить точки покриття (coverpoint), які збирають статистику значень конкретних об’єктів або виразів. На рис. 6.18 представлено приклад такої конструкції.

```
reg [3:0] count;
covergroup cg @(posedge clk);
  cnt : coverpoint count;
endgroup
cg grp = new();
```

Рисунок 6.18–Приклад конструкції covergroup

Під час кожного переднього фронту сигналу clk (@(posedgeclk)) covergroup збиратиме значення сигналу count. Далі середовище моделювання видасть звіт про покриття (coverage report). В даному звіті буде показано процент значень, які прийняв сигнал. На рис. 6.19 показано фрагмент такого звіту. З нього видно, що з усієї множини можливих значень (16), сигнал countприйняв тільки 25% (4).

```
COVERGROUP COVERAGE
============================================================
|       Covergroup        |  Hits   |  Goal /  |  Status   |
|                         |         | At Least |           |
============================================================
| TYPE /testbench/inst/cg | 25.000% | 100.000% | Uncovered |
============================================================
```

Рисунок 6.19–Приклад звіту про покриття

Для більшого гнучкості SystemVerilog надає можливість розбивати значення на так звані кошики (bins). Фактично кожний кошик є множиною значень. На рис. 6.20 зображено coverpointз кошиками для непарних значень сигналу count.

```
reg [3:0] count;
covergroup cg @(posedge clk);
  count : coverpoint cnt {
    bins odd[] = {1, 3, 5, 7, 9, 11, 13, 15};
  }
endgroup
```

Рисунок 6.20–coverpoint з кошиками

6.6.3.1 Конструкція covergroupдля аналізу зміни бітів регістру станів

Використання інструментів для функціонального покриття надасть динаміку та сукупну кількість змін за певний час роботи автомата. Це дозволить вже на етапі моделювання підібрати кодування, яке буде відповідати вимогам енергоспоживання. На рис. 6.21 показано декларацію конструкції covergroup для

збору статистики щодо змін бітів під час автоматних переходів. Вираз state ^ next_stateвиражає зміну станів. Системна функція $countonesповерне кількість одиниць переданого аргументу (відстань Геммінга).

```
covergroup encoding @(posedge clk);
    changes: coverpoint $countones(state ^ next_state);
endgroup
```

Рисунок 6.21–coverpoint для аналізу відстані Гемінга

```
covergroup encoding @(posedge clk);
    transition : coverpoint state {
        bins idle_read = (IDLE => READ);
        bins idle_idle = (IDLE => IDLE);
        bins read_dly = (READ => DLY);
        bins dly_read = (DLY => READ);
        bins dly_done = (DLY => DONE);
        bins done_idl = (DONE => IDLE);
    }
    changes: coverpoint $countones(state ^ next_state);
endgroup
```

Рисунок 6.22–coverpoint для аналізу частоти переходів

Конструкція covergroupдля аналізу частоти переходів складається з двох точок покриття. Точка transition містить кошики для усіх можливих переходів, changes–підрахунок сумарної відстані Геммінга за весь час моделювання. Такий підхід дозволить скорегувати ручне кодування. Наприклад, якщо перехід між станами DLY та READ відбувається частіше – ці стани необхідно закодувати сусідніми кодами Грея.

7 ФУНКЦИОНАЛЬНА ВЕРИФІКАЦІЯ ПЕРЕХОДІВ АВТОМАТА

У розділі 6 було представлено матричний спосіб верифікації автоматних переходів. Даний спосіб дозволяє на етапі моделювання або формальної верифікації підтвердити строгу відповідність переходів графу переходів. Проте у процесі функціональної верифікації існує задача перевірки активації усіх можливих умов переходів. Більш того, ретельність тестових наборів, які, як правило, генеруються випадковим чином, є важливою для активації відповідних умов асерцій та уникнення «вакуумних успіхів».

7.1 Поняття шляху автоматного переходу

Автоматний перехід складається з наступних частин: початковий стан, кінцевий стан та логічна умова активації переходу. Різні умови можуть збуджувати один й той самий перехід. Як видно з рис. 7.1 активне значення сигналу go або ws активують перехід зі страну IDLE в READ.

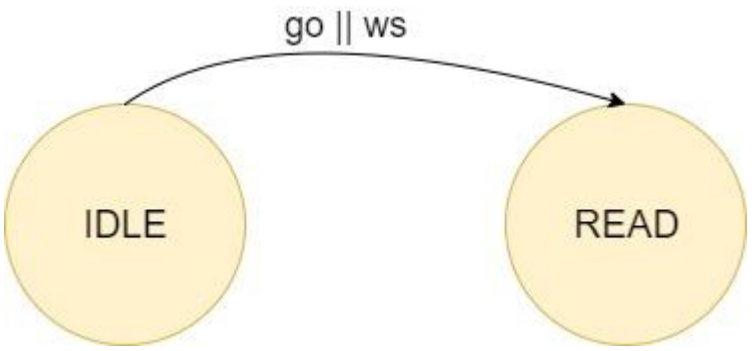


Рисунок 7.1 – Фрагмент автоматного переходу з комплексною умовою

Той же самий перехід можна представити у вигляді двох переходів з простими (атомарними) умовами.

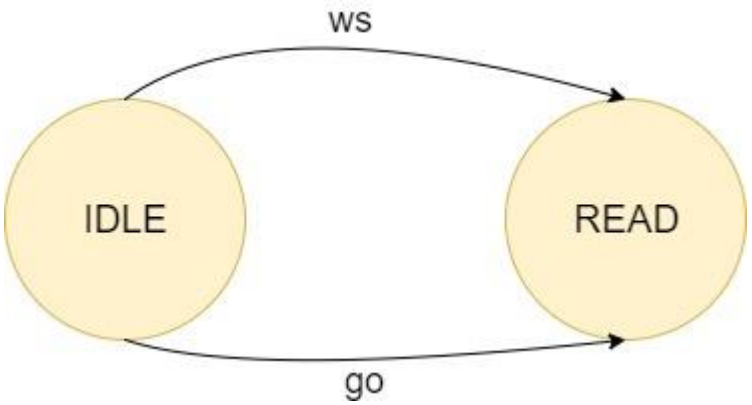


Рисунок 7.2 – Фрагмент автоматного переходу після декомпозиції умов

Таким чином з’являється концепція автоматного шляху. Автоматний шлях – перехід з одного стану в інший з атомарною умовою.

7.2 Покриття автоматних шляхів за допомогою coverдиректив

Три складові шляху формують властивість дизайну, яка натуральною мовою визначається наступним чином: якщо автомат знаходиться в стані A та виконується атомарна умова переходу в стан B, то на наступному такті автомат має перейти у стан B.

Розглянемо цей підхід на прикладі автомата, граф переходів якого зображено на рис. 7.3.

Даний автомат має 5 станів та 8 переходів. Переходи зі стану IDLE в READта з READ в PROC є комплексними. Для подальшого аналізу ці переходи необхідно розкласти на атомарні.

Далі необхідно виразити цей шляхи за допомогою засобів SVA. На рис. 7.3 виражено дві властивості, які виражають два шляхи переходу IDLE–READ.Ліва частина логічного виразу виражає поточний стан автомату(state == IDLE) та стан вхідних сигналів, які збуджують перехід (goабо ws). Права частина частина виражає стан, у якому автомат опиниться через такт.

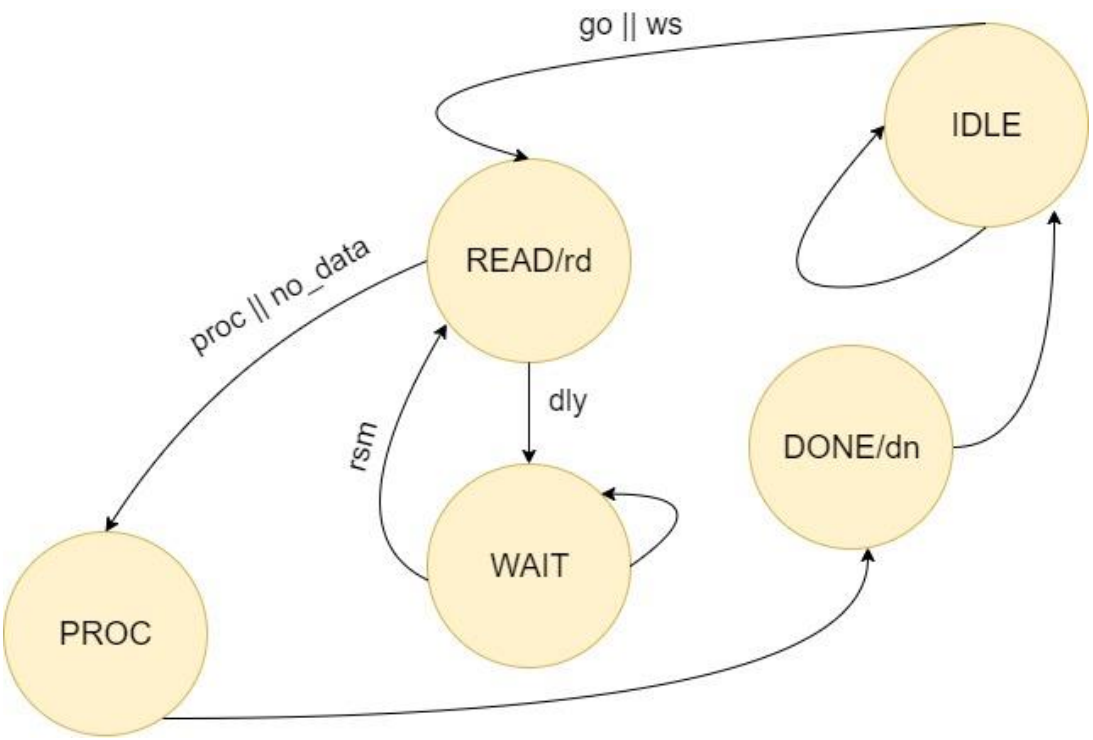


Рисунок 7.3 –Приклад автомата з комплексними умовами переходу

```
property trans_prop_IDLE_READ_go;
@(posedge clk)
  state == IDLE && go |=> state == READ;
endproperty

property trans_prop_IDLE_READ_ws;
@(posedge clk)
  state == IDLE && ws |=> state == READ;
endproperty
```

Рисунок 7.4–Вираження шляхів автоматного переходу за допомогою конструкцій мови SVA

Метою покриття шляхів є отримання статистичних даних про активацію умов переходів. Для цієї мети використовується конструкція coverproperty.

```
cover property (trans_prop_IDLE_READ_go);
cover property (trans_prop_IDLE_READ_ws);
```

Рисунок 7.5–Використання конструкції coverproperty для збору інформації про активацію властивості

Дана конструкція надасть наступну інформацію:

- кількість спроб вирахувати властивість;
- кількість успішних проходжень властивості;
- кількість невдалих проходжень властивості;
- кількість “вакуумних успіхів” властивості (не спрацювали вирази активації переходів).

Дана статистика допоможе поліпшити тестові набори для більш ретельного процесу верифікації.

Варто зауважити, що кількість властивостей, необхідних для верифікації шляхів, буде рости лінійно, якщо всі переходи між станами мають прості (атомарні) умови.

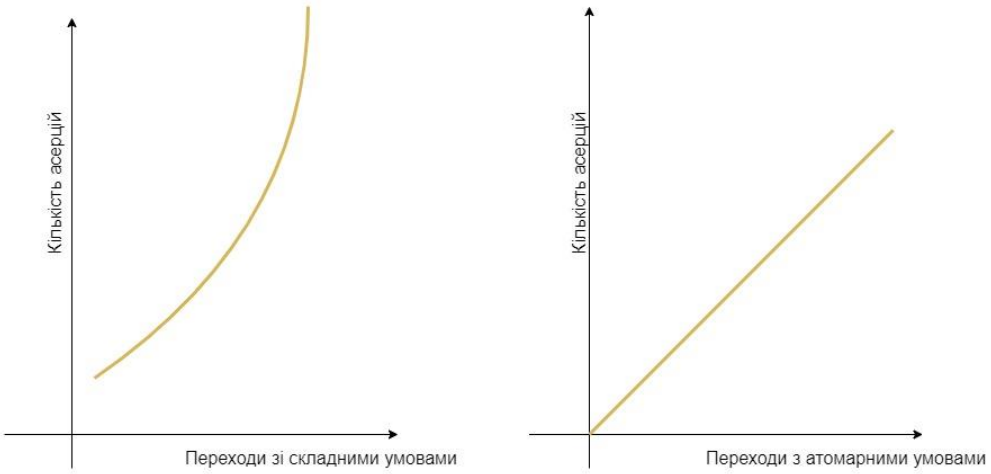


Рисунок 7.6–Залежність кількості асерцій від типу переходів

7.3 Покриття автоматних шляхів за допомогою конструкції covergroup

7.3.1 Семантика операції crossмови SystemVerilog

У розділі 6.3 було розглянуто конструкції covergroupта coverpointмови SystemVerilogдля функціонального покриття.

Під час аналізу статистики значень між кількома сигнали буває корисно знати кореляцію між множини їх значень. Наприклад, під час тестування арифметично-логічного пристрою буває корисно знати чи приймав один з операндів значення нуль, а інший – ненульове значення для операції відіймання. Фактично це є декартів добуток множин.В контексті мови SystemVerilog це називається кросом (cross).

На рис. 7.7 зображено операцію crossміж двома coverpoint для сигналів a и b. Кожен сигналможе приймати 16 значень. Відповідно результатом операції буде множина за 256 кортежів.

```
bit [3:0] a, b;
covergroup cg @(posedge clk);
  c1: coverpoint a;
  c2: coverpoint b;
  c1Xc2: cross c1,c2;
endgroup : cg
```

Рисунок 7.6–Операція cross між двома coverpoint

7.3.2 Конструкція covergroupдля автоматних шляхів

Конструкція covergroup для покриття автоматних шляхів складається мінімум з двох точок покриття та операції cross між ними. Перша точка покриває перехід між двома станами, інші– відповідні значення вхідних сигналів, які збуджують цей перехід.

На рис. 7.8 зображено приклад автомату з 3 станів та 3 переходів. Перехід

зі стану Transfery стан Sleep активізується, коли сигнал cmd приймає одне зі значень 11-12, зі стану Sleep до Standby–коли приймає 1-12. Таким чином, даний автомат має 24 шляхи.

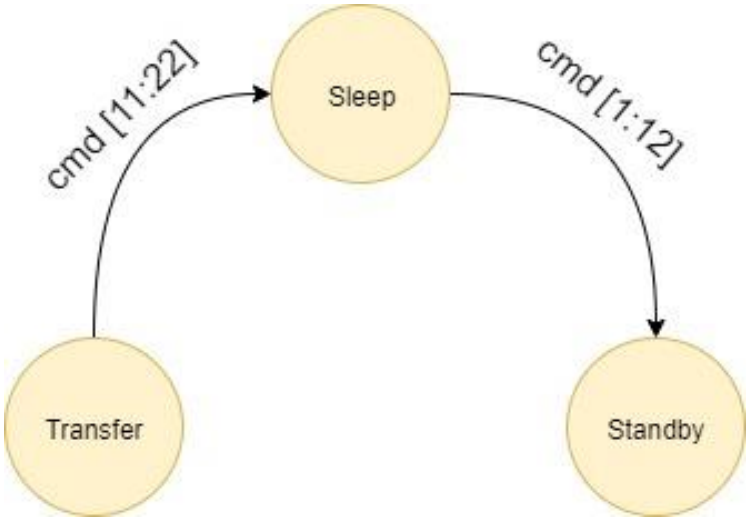


Рисунок 7.8– Приклад автомата з 24 шляхами

Модель для покриття шляхів буде складатися з однієї covergroup, яка містить дві конструкції coverpointта crossоперацію між ними.

На рис. 7.9 конструкцію coverpointдля покриття сигналу state. Ця конструкція містить в собі декларацію кошиків, кожен з яких покриває можливі переходи. Для цього початковий та цільовий стан об'єднуються оператором =>.

```
state: coverpoint state {
    bins stdby_to_sleep = ( STANBDY => SLEEP );
    bins tx_to_stdby = ( TRANSFER => STANBDY );
}
```

Рисунок 7.9– Декларація coverpointдля покриття statesигналу

Точка покриття для сигналу, що збуджує переходи має включати кошики (bins), які виражають множину значень, що активізують переходи. Рекомендується групувати значення, які збуджують різні переходи в різні кошики. На рис. 7.9 зображено coverpointдля покриття сигналу cmd. Так як цей

сигнал збуджує два різні переходи, ця точка покриття має двамасиви кошиків. Важливо декларувати саме масиви кошиків, так як важливо знати статистику кожного значення, а не діапазону значень. Масив амістить множину значень, які активізують перехід між станами Sleep та Standby, b–між станами Transfer та Sleep. Варто зауважити, що обидва масиви містять кошики для значеннь 11 та 12. Ці значення активізують різні переходи в залежності від початкового стану.

```
cmd: coverpoint cmd
{
    bins a[] = {[1:12]};
    bins b[] = {[11:22]};
}
```

Рисунок 7.10 – Декларація coverpointдля покриття сигналуcmd

Далі необхідно виразити відношення між двома кошиками за допомогою операції cross. За замовчуванням середовища проектування комбінують усі можливі комбінації між кошиками, тобто перехід зі Standby Sleep буде покриватися з урахуванням усіх можливих значень сигналу cmd (1-22). Це є надлишковою інформацією, адже даний перехід відбувається тільки при значеннях 1-12. Таким чином необхідні спеціальні кошики, які будуть обмежувати усю можливу множину комбінацій. Для цього існує спеціальна конструкція ignore_bins. Дана конструкція виражає кошики, що містять комбінації значень, які будуть відкинуті при виконанні операції кроссу. На рис. 7.10 представлено конструкцію ignore_bins для розглядаємого автомата.

```
cmdXstate: cross cmd, state
{
ignore_bins ignore = binsof(state.stdby_to_sleep) && (!binsof (cmd.a)) ||
                     binsof(state.tx_to_stdby) && (!binsof (cmd.b));
}
```

Рисунок 7.11 – Приклад конструкції ignore_bins

Вираз binsof(state.stdby_to_sleep) && (!binsof (cmd.a)) виражає, що перехід крос між кошиком stdby_to_sleep має ігнорувати значення з кошиків, які не належать кошику cmd.a (вираз !binsof (cmd.a)). На рис. 7.12 представлено кінцевий варіант конструкції covergroupдля покриття шляхів розглядаємого автомата.

```
covergroup transitions @(posedge clk);
    state: coverpoint state
    {
        bins stdby_to_sleep = ( STANBDY => SLEEP );
        bins tx_to_stdby = ( TRANSFER => STANBDY );
    }

    cmd: coverpoint cmd
    {
        bins a[] = {[1:12]};
        bins b[] = {[11:22]};
    }

    cmdXstate: cross cmd, state
    {
        ignore_bins ignore = binsof(state.stdby_to_sleep) && (!binsof (cmd.a)) ||
        binsof(state.tx_to_stdby) && (!binsof (cmd.b));
    }

endgroup
```

Рисунок 7.12 – Кінцевий варіант конструкції covergroup

Як правило комплексні переходи мають наявні умови, які містять декілька сигналів. Для таких переходів необхідно описувати декілька точок покриття, а відповідно зростає складність виразу для опису конструкції ignore_bins. На рис. 7.13 зображено фрагмент коду на мові SystemVerilog, який описує переходи типового автомата.

```
always_comb begin
  case(state)
    IDLE: if(go || ws)
      next_state = READ;
    else
      next_state = IDLE; // !go && !ws
    READ: begin
      if(dly)
        next_state = WAIT;
      else if(proc || no_data)
        next_state = PROC;
    end
    WAIT: next_state = rsm ? READ : WAIT;
    PROC: next_state = DONE;
    DONE: next_state = IDLE;
    default: next_state = IDLE;
  endcase
end
```

Рисунок 7.13 – Типовий код, що виражає логіку автоматних переходів

Після дефініції кошиків для переходів і сигналів рекомендується операцію застосовувати операцію кросу для покриття тільки комплексних переходів. Такими переходами в даному описі є переходи між станами IDLE та READ та READ-PROC. Атомарні переходи можна не включати в крос. Покриття таких переходів буде видно у відповідних кошиках точки покриття переходів. Такими переходами є READ-WAIT, WAIT-READ, WAIT-WAIT, PROC-DONE, DONE-IDLE. Варто зауважити, що циклічний перехід IDLE-IDLEє атомарним, хоча його умова містить два сигнали. На рис. 7.13 представлено дефеніцію cover_group для покриття шляхів даних переходів.

```
covergroup cg @(posedge clk);
  transitions: coverpoint state {
    bins idle_read = (IDLE => READ);
    bins idle_idle = (IDLE => IDLE);

    bins read_wait = (READ => WAIT);
    bins read_proc = (READ => PROC);

    bins wait_read = (WAIT => READ);
    bins wait_wait = (WAIT => WAIT);
  }

  c1: coverpoint ws { bins one = {1}; }
  c2: coverpoint go { bins one = {1}; }
  c3: coverpoint proc { bins one = {1};}
  c4: coverpoint no_data { bins one = {1}; }

  crs : cross transitions, c1, c2, c3, c4 {
    ignore_bins ignore =
    (binsof(transitions.idle_read) && (!binsof(c1.one) && !binsof(c2.one))) ||
    (binsof(transitions.read_proc) && (!binsof(c3.one) && !binsof(c4.one)));
  }

endgroup
```

Рисунок 7.14 – Дефініція cover_group для переходів,
що містять декілька сигналів

7.4 Порівняння методів покриття переходів

Головнимикритєріями порівняння двох розглянутих методів є обсяг верифікаційного коду та його незалежність від конкретної технології.

Метод cover директив підходить для автоматівз атомарними умовами переходів між станами. Зростання кількості шляхів між двома станами вимагає більшої кількості властивостей для їхнього покриття. До того ж, мова SystemVerilog забороняє написання SVA конструкцій в об’єктах. Як наслідок. цей підхід неможливо використовувати у верифікаційних методологіях, що базуються на класах. Прикладами таких методологій є Open Verification Methodology (OVM), Universal Verification Methodology (UVM), тощо.

Метод конструкцій cover_group має більш просту компактну структуру, яка потребує менше інструкцій для опису великої кількості шляхів між станами.

До того ж, даний спосіб підходить для UVM, OVM та інших методологіях з класами. Недоліком є те, що даний спосіб можливо реалізувати тільки на мові SystemVerilog. Крім того, даний спосіб не підходить для формальної верифікації.

Головною перевагою методу covergдиректив є йогонезалежність від конкретної мови опису апаратури. Ідею вираження шляхів за допомогою властивостей можна описати, наприклад, на мові Property Specification Language (PSL). PSL можна використовувати як з дизайнами, що описані на мові Verilog/SystemVerilog, так із дизайнами на мові VHDL.

```
-- psl default clock is (clk'event and clk= '1');
-- psl property trans_prop_idle_read_go is

-- always ({(state = idle and go = 1)} |=> {state = read});

-- psl cover trans_prop_idle_read_go;
```

Рисунок 7.15 – Приклад властивості на мові PSL

Крім того, вираження шляхів за допомогою мов опису властивостей дозволяє перевіряти їх за допомогою середовищ формальної верифікації. Такі інструментарії можуть дати набагато більше інформації про стан дизайну у разі порушення властивостей.

ВИСНОВКИ

В даній роботі було представлено огляд та систематизацію сучасних методів верифікації HDL описів абстрактних цифрових автоматів.

Було проаналізовано основні складнощі, що виникають під час процесу верифікації, а саме:

– обмеженність у часі;

– складність аналізу великого обсягу даних: часових діаграм, логів, тощо;

– проблема якості тестів.

У ході дослідження було доведено, що асерційна верифікація вирішує ці проблеми.

Розглянуто основні складові HDL опису автомата: переходи, кодування, логіка для виходів, темпоральні властивості. Для кожного аспекту було запропоновано відповідні способи верифікації за допомогою апарату асерцій та властивостей.

Було розглянуто та порівняно два підходи верифікації автоматних шляхів:

– за допомогою механізмів функціонального покриття мови SystemVerilog;

– за допомогою механізму властивостей.

Було практично доведено, що використання асерцій та властивостей має наступні переваги порівняно з класичним способом верифікації:

– зменшення часу відладки;

– явне вираження специфікації дизайну безпосередньо у HDL-описі;

– значне спрощення пошуку логічних помилок при автоматному описі;

– можливість використання асерцій з формальними середовищами верифікації.

Викладені способи можуть бути автоматизовані за допомогою мов програмування високого рівня (C++, Java, Python, тощо) та бути використані як складова частина комплексного середовища верифікації та моделювання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Turumella, B. Assertion-based verification of a 32 thread spareTM CMT microprocessor [Text] / B. Turumella.// in DAC 08: Proceedings of the 45th annual Design Automation Conference. – 2008. P. 256-261.
2. Jun, Yuan. Constraint-based verification [Text] /YuanJun, Carl Pixley, and Adnan Aziz–Springer Science & Business Media, 2006. – P. 260 - 332
3. Ping, Yeung. Practical assertion-based formal verification for SoC designs. [Text] / YeungPing, Kenneth Larsen. / System-on-Chip International Symposium on IEEE - 2005. – P 22 – 55.
4. Li, Yangyang, et al. A study on the assertion-based verification of digital IC [Text] // ICIC'09. Second International Conference on. Vol. 2. IEEE. - 2009. – P.40 - 55
5. Sonny, K. OVL, PSL, SVA: Assertion based verification using checkers and standard assertion languages [Text] / K Sonny, T. Ashley, and S. Lakshmiprabha, // Advanced Computing and Communication Systems (ICACCS), 2013 International Conference on. IEEE. - 2013. – P. 30 - 88
6. Eisner C. A methodology for formal design of hardware control with application to cache coherence protocols [Text] /Eisner C , Shitsevalov I, Hoover R // Proceedings of the 37th Annual Design Automation Conference. ACM, 2000. – P. 724-729.
7. Y. Wolfsthal. EU-Sponsored Deployment of PSL [Text] //Y. Wolfsthal / Sugar Consortium Meeting - 04, Feb 2004 . P. 50 – 100.
8. Foster, Harry, Kenneth Larsen, and Mike Turpin. Introduction to the new accellera open verification library [Text] // Proceedings of the Conference on Design and Automation and Test in Europe. - 2006. – P. 72 – 108.
9. Foster, H. Introduction to the new accellera open verification library [Text] // Foster Harry, Kenneth Larsen, and Mike Turpin // Proceedings of the

Conference on Design and Automation and Test in Europe. – 2006. – P.150 – 220.

10. Cohen, Ben. Using PSL/Sugar for formal and dynamic verification: Guide to Property Specification Language for Assertion-based Verification [Text] // Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari // VhdlCohen Publishing, 2004. – P. 13- 18.
11. Property Specification Language Reference Manual [Text] // Accellera, Version 1.1, 2004. – P. 25 – 32.
12. Ping Yeung. Applying Assertion-Based Formal Verification to Verification Hot Spots [Text] // Ping Yeung, Sundaram Subramanian // Mentor Graphics Technical Library – 2007. – P. 54 – 111.
13. Ping Yeung. Functional Verification of ARM-based Platform Design with Assertion-Based Verification Methodology [Text] // Ping Yeung // ARM Developers Conference - 2004. – P.120 – 200.
14. Harry Foster. Integrating Formal Verification into a Traditional Flow [Text] //Harry Foster// Mentor Graphics White Paper - 2006. – P. 45 – 120.
15. Пшеничний, К.Ю.Методи верифікації темпоральних властивостей цифрових автоматів [Текст]/К.Ю. Пшеничний, Хаханова Г.В. //Журн. Радиоэлектроника и информатика. –2019. –№3 –С. 34-39.
16. Радіоелектроніка та Молодь у XXI столітті[Текст] : матеріали 23-го міжнародного молодіжного форуму. (16-18 квітня 2019 р.) / відп. ред. О.С. Ляшенко. – Харків : ХНУРЕ, 2019 – С. 5-7.
17. Кулак, Є. М. Прикладна теорія цифрових автоматів» для студентів денної форми навчання спеціальностей[Текст]: конс. лекцій /Упоряд.: Є. М. Кулак;– Харків: ХТУРЕ, 2009. - 227 с.