ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕКОНОМІКИ, УПРАВЛІННЯ, ПРАВА ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Пояснювальна записка

до кваліфікаційної роботи на здобуття ступеня вищої освіти магістр

на тему: **«Розроблення процедури верифікації електронних проектів FPGA із використанням методології OSSVM (OpenSource VHDL Verification Methodology)»**

Виконав: здобувач вищої освіти
за освітньо-професійною програмою
Інформаційні управляючі системи та
технології спеціальності
126 Інформаційні системи та технології
ступеня вищої освіти магістр
групи 126ІСТ_мд_22
Гришко А. С.
Керівник: Одарущенко О.М.
Рецензент:Сергій ЯХІН

Полтава – 2023 року

ВСТУП

Актуальність теми дослідження. На сьогодні, в промисловості по виготовленню електронних приладів, систем автоматизації, будівництві автомобілів, літаків, потягів та взагалі любої техніки, де використовуються інтегральні схеми, актуальною проблемою є ріст числа транзисторів на кристалі. За законом Мура, який сформулювали ще в 1965 році, досі актуальне твердження про ріст кількості транзисторів, а саме подвоєння їх кількості кожні 18 місяців. Збільшення складності НВІС на базі розвитку інтегральної технології дозволяє мати в апаратурі все більш різноманітні та складні функції. Для ефективного використання цих можливостей необхідний перехід на нові технології проектування і застосування апаратно реалізованих вузлів, блоків, систем . Типова логічна схема в графічному поданні містить на сторінці фрагмент, еквівалентний порядку 200 вентилів . Відповідно, схема НВІС на 10 тис. вентилів буде об'ємом в 50 сторінок. Легко уявити собі, у що виллється (за часом) складання і введення в графічну нотацію схем НВІС складністю 50 тис., 100 тис., 500 тис. вентилів і більше. Альтернативою малювання деталізованих схем з низькорівневих елементів є мови опису апаратури високого рівня. Мови цього класу називають мовами HDL (Hardware Description Language). Вони не тільки забезпечують компактний запис для проектуємої схеми, дають значне скорочення трудомісткості і термінів розробки великих схем, а й спрощують міграцію, перенесення проекту на різні варіанти інтегральних технологій, реалізацію їх в НВІС з урахуванням специфіки технологій різних виробників. Розробник отримує можливість оцінювати варіанти реалізації проектованого пристрою в НВІС при різних варіантах проектних обмежень, на різних технологіях, у різних виробників. Але використовуючи традиційні засоби опису апаратури, складно отримати цілісний, достатньо суворий, однозначний опис сучасних багатокомпонентних і функціонально складних цифрових систем . Зростаюча алгоритмічна складність апаратно реалізованих пристроїв призводить до того, що, як проблеми розробки, опису та застосування апаратури, так і підходи до їх вирішення, стають подібні до

проблем і методів рішення для сучасних програмних систем. Перспективний напрямок вирішення цих проблем - застосування алгоритмічного підходу, створення алгоритмічної мови для опису апаратури, програмування і структури, функціонування апаратних засобів обробки інформації. Найбільш поширеною мовою цього класу, специфікованою міжнародними стандартами, є мова VHDL, яка розроблена в рамках американського проекту створення нового покоління високошвидкісної елементної бази (Very High Speed Integrated Circuits - VHSIC). Абревіатура VHDL розшифровується як VHSIC Hardware Description Language. Мова VHDL призначена для вирішення комплексу завдань в ході проектування і застосування цифрових систем, та їх апаратних засобів. Так як мова йде про створення ПЗ для ПЛІС, актуальною проблемою будь – якого ПЗ є дефекти. Тому стає актуальною проблема знаходження дефектів, їх усунення та запобігання. Саме тому дуже важливим є тестування програмного продукту, створеного на мові VHDL. Одним із найважливіших та найскладніших видів тестування для ПЗ створеного на мові VHDL є функціональне тестування. ФТ розглядає заздалегідь вказану поведінку і ґрунтується на аналізі специфікацій функціональності компонента або системи в цілому. Виконання ФТ це цілий процес, який має багатокрокову структуру та низку звітних документів на виході. Тому для вдалого та ефективного проведення ФТ необхідно створення процедури проведення ФТ для ПЗ розробленого на мові VHDL, яка б надавала актуальну та продуктивну модель тестової системи та дозволяла виконувати ФТ з мінімумом зусиль зі сторони інженера, для створення надійного та якісного ПЗ. Саме тому тема розробки процедури функціонального тестування для коду на мові VHDL є актуальною.

Мета дослідження – розробка процедури проведення функціонального тестування коду на мові VHDL для систем на базі ПЛІС.

Завданнями кваліфікаційної роботи є:

1. Опис продуктивної та актуальної моделі тестової системи;
2. Максимальна оптимізація процесу ФТ;
3. Підвищення продуктивності інженерів, які проводять ФТ;

4. Опис створення якісних звітних документів.

Об'єкт дослідження – процес розробки та тестування коду на мові VHDL для систем на базі ПЛІС.

Предмет дослідження – процедура проведення функціонального тестування коду на мові VHDL для систем на базі ПЛІС.

Інформаційна база кваліфікаційної роботи сформована з наукових фахових статей, наукових монографій, аналітичних звітів державних та міжнародних експертних груп щодо експлуатації критичних технічних систем, виконаних науково-дослідних та дисертаційних робот заданою тематикою.

Елементи наукової новизни роботи полягають в тому, що в розробленій процедурі, описана нова модель тестової системи, яка відрізняється від відомих наступними функціями:

1. Ручне задавання тестових векторів;

2. Вдосконалена система моніторингу, яка за рахунок меншої кількості ітерацій, швидше проводить порівняння результатів;

3. Автоматична генерація звіту, що зменшує час на аналіз та оформлення результатів;

4. Відсутність підрахунку функціонального покриття, через вимоги стандарту IEC 61508-2010 для SIL 3 (Використання еквівалентних класів та перевірки граничних значень).

Практична значущість роботи полягає в тому, що розроблена процедура ФТ дозволяє покращити якість виробляємого ПЗ для ПЛІС, покращити продуктивність роботи відповідальних за ФТ інженерів, зменшити затрачуваний на ФТ час та підвищити точність результатів, через те, що аналіз результатів ФТ за розробленою процедурою проводиться майже повністю автоматично, отримати якісні звітні документи.

Апробація результатів дослідження відбувалася шляхом оприлюднення доповідей на міжнародній та студентській конференціях.

Публікації. За результатами проведеного дослідження опубліковані тези: «Моделювання оцінювання готовності та функційної безпечності електричних,

електронних, програмовних електронних систем», матеріали щорічної студентської наукової конференції Полтавського державного аграрного університету, 10 листопада 2023 р. Полтава: ПДАУ, 2023.

Зв'язок роботи з науковими програмами, планами, темами: дослідження, проведені в кваліфікаційній роботі виконувалися в рамках/інтересах науково-дослідної роботи «Розвиток підприємництва: управлінські, економічні, іноваційна та правові аспекти» відповідно до договору №9 від 15.05.2023 р. між ТОВ «ПАФ Гарант» та Полтавським державним аграрним університетом (розділ «Обгрунтування показників оцінювання гарантоздатності розподілених інформаційних систем»).

Структура та обсяг роботи. Робота складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 71 сторінок; робота містить 21 рисунок; 5 таблиць; список використаних джерел, що включає 56 найменувань.

РОЗДІЛ 1
VHDL КОД ДЛЯ FPGA, ЯК ОБ'ЄКТ ФУНКЦІОНАЛЬНОГО ТЕСТУВАННЯ

1.1 Основна інформація про FPGA

Згідно відомому емпіричному правилу, так званому закону Мура, кількість транзисторів на кристалі надвеликих інтегральних схем подвоюється кожні 18 місяців (для процесорних схем). Це правило, сформульоване в 1965 році, коли на кристалі інтегральної схеми поміщалось всього 30 транзисторів, працює до сих пір, коли на кристалі процесорної надвеликої інтегральної схеми розміщується більш ніж 50 мільйонів транзисторів. В деяких надвеликих інтегральних схемах, а саме програмуємих логічних інтегральних схемах (ПЛІС – FPGA), вже знаходиться більш ніж мільйон вентилів. Дамо визначення ПЛІС.

ПЛІС – представляють собою цифрові інтегральні схеми (ІС), які складаються з програмованих логічних блоків та програмованих з'єднань між цими блоками. Можливість конфігурувати ці пристрої дозволяє інженерам – розробникам вирішувати велику кількість задач.

Можливості ПЛІС:

1. ПЛІС є альтернативою для мікропроцесорних цифрових систем, таких як програмуємі логічні контролери (ПЛК);

2. Інтегральна мікросхема ПЛІС, після її програмування, є технічним засобом, не підверненим збоям та не маючим системи команд переривань (як у мікропроцесорів);

3. ПЛІС може реалізувати різну математичну логіку обробки, необхідну для виконання функцій безпеки та управління різноманітними автоматичними системами управління.

4. Функції даних технічних засобів конфігуруються за допомогою мови опису апаратури (HDL або Verilog);

5. Конфігурація логіки ПЛІС може оновлюватись після їх ретельної верифікації та контролю (в тому числі після установлення обладнання);

6. Якщо необхідно, то мікропроцесори можуть бути реалізовані на основі ПЛІС.

В залежності від способу виготовлення ПЛІС можуть програмуватися один раз, або багаторазово. Пристрої, які можуть програмуватися тільки один раз, називаються одноразово програмуємими.

Словопоєднання «field programmable», що міститься в розшифровці абревіатури FPGA, означає, що програмування FPGA – приладів виконується на місці, «в польових умовах» (на відміну від приладів, внутрішня функціональність яких, жорстко прописана виробником). Це може значити, що FPGA- прилади (ПЛІС), конфігуруються в лабораторних умовах, та свідчити про те, що справа йде про можливість модифікації функцій приладу, вбудованого в електронну систему, яка вже використовується. Якщо прилад може бути запрограмовано, залишаючись у складі системи більш високого рівня, воно називається внутрішньо-системно програмуємим.

Існує безліч різних типів цифрових мікросхем, в тому числі і такі як «розсипна логіка » (невеликі компоненти, що містять кілька простих фіксованих логічних функцій), пристрої пам'яті та мікропроцесори. У даному випадку інтерес представляють програмовані логічні пристрої (ПЛП), спеціалізовані замовні інтегральні мікросхеми (ASIC - application specific integrated circuit, спеціалізована інтегральна схема і ASSP - application specific standard parts, спеціалізована стандартна схема) і ПЛІС. Причому термін ПЛП об'єднує два типи пристроїв: прості програмовані логічні пристрої (прості ПЛП) і складні програмовані логічні пристрої (складні ПЛП або CPLD).

Внутрішня архітектура ПЛП визначена виробником, таким чином, що вони можуть бути налаштовані (перепрограмовані) «на місці» для виконання самих різних функцій. На відміну від ПЛІС ці пристрої містять меншу кількість вентилів і використовуються для вирішення невеликих і досить простих завдань.

Разом з тим, існують замовні інтегральні схеми ASIC і ASSP, які містять сотні мільйонів логічних вентилів і можуть виконувати неймовірно великі і складні функції. В основі ASIC і ASSP лежать одні й ті ж конструкторські

рішення, і у них одна і таж технологія виробництва. Обидва типи пристроїв розробляються для використання в складі спеціальних додатків, але при цьому ASIC розробляються і виготовляються за замовленням спеціалізованих компаній, а ASSP призначаються масовому користувачеві.

Незважаючи на те, що пропоновані користувачеві замовні мікросхеми відрізняються високим ступенем інтеграції, рівнем складності розв'язуваних завдань і продуктивністю, їх розробка і виробництво досить тривалий і дорогий процес. До того ж, остаточний варіант схеми «заморожується в кремнії», і для її модифікації потрібно створення нової версії.

Таким чином, ПЛІС займають проміжне положення між ПЛП і замовними інтегральними схемами. З одного боку, їх функціональність може бути задана безпосередньо на місці відповідно до вимог замовника-користувача. З іншого боку, вони можуть містити мільйони логічних вентилів і, отже, реалізовувати надзвичайно великі і складні функції, які спочатку могли бути реалізовані тільки за допомогою замовних інтегральних схем.

Використання ПЛІС:

Цифрова обробка сигналів.

Високошвидкісна цифрова обробка сигналів традиційно проводилася за допомогою спеціально розроблених мікропроцесорів, званих цифрові сигнальні процесори (ЦСП). Однак сучасні ПЛІС містять вбудовані помножувачі, схеми арифметичного переносу і великий об'єм оперативної пам'яті всередині кристалу. Все це в поєднанні з високим ступенем паралелізму ПЛІС забезпечує перевагу ПЛІС над найшвидшими сигнальними процесорами в 500 і більше разів.

Вбудовуванні мікроконтролери.

Нескладні завдання управління зазвичай виконуються вбудовуваними процесорами спеціального призначення, які називаються мікроконтроллерами. Ці недорогі пристрої містять вбудовану програму, пам'ять команд, таймери, інтерфейси введення / виводу, розташовані поряд з ядром на одному кристалі. Ціни на ПЛІС падають, до того ж, навіть найпростіші з них можна використовувати для реалізації програмного мікропроцесорного ядра з

необхідними функціями введення/виводу. В результаті ПЛІС стають все більш привабливими пристроями для реалізації функцій мікроконтролерів.

Фізичний рівень передачі даних.

ПЛІС вже давно використовуються в якості зв'язуючої логіки, що виконує функцію інтерфейсу між мікросхемами, що реалізовують фізичний рівень передачі даних, та вищими рівнями мережевих протоколів. Той факт, що сучасні ПЛІС можуть містити безліч високошвидкісних передавачів, означає, що мережеві та комунікаційні функції можуть бути реалізовані в одному пристрої.

Системи з перебудовуємою архітектурою.

Можна використовувати «апаратне прискорення» програмних алгоритмів, грунтуючись на таких властивостях програмованих логічних інтегральних схем (ПЛІС), як паралелізм і реконфігуруємість. В даний час різні компанії зайняті створенням величезних перенастроювати обчислювальних машин на основі ПЛІС. Такі системи можуть використовуватися для виконання широкого спектру завдань - від моделювання апаратури до криптографічного аналізу або створення нових ліків.

Автоматизовані системи управління.

Для автоматизованих систем управління, є актуальним використання ПЛІС, тому що:

1. Технологія, перевірена у використанні на АЕС та інших підприємствах з особливими вимогами до безпеки;

2. Реалізація функцій безпеки без використання будь-якого прикладного програмного забезпечення або операційної системи;

3. Паралельне виконання алгоритмів керування і функцій комунікації, що забезпечує швидку реакцію з заданими (фіксованими) величинами часу;

4. Зрозуміле і просте надання електронного дизайну, що дозволяє зменшити витрати необхідні для розробки, контролю та верифікації ;

5. Сумісність HDL коду з різними типами ПЛІС, виготовленими різними виробниками;

6. Підходить для реверс- інжинірингу застарілих центральних процесорів без модифікацій існуючих кодів програмного забезпечення;

7. Специфічні властивості щодо інформаційної безпеки, за рахунок відмінності від інших технологій на основі мікропроцесорів і мікроконтролерів (для ПЛІС немає вірусів).

Переваги ПЛІС:

1. Низьке споживання енергії, що ідеально підходить для портативних електронних пристроїв;

2. Реконфігуруємість, максимальна універсальність застосування;

3. Оновлення за допомогою програмного забезпечення, замість обширної заміни обладнання ;

4. Необмежена кількість циклів перепрограмування ;

5. Можливість вдосконалення, без необхідності покупки нового пристрою.

6. Зниження кількості виходів дозволяє створювати більш компактні модулі;

7. Паралельні обчислювальні можливості .

Недоліки ПЛІС:

1. Висока вартість виготовлення абсолютно нового чіпа ;

2. Проблеми з потоком вихідних / вхідних даних пристрою;

3. Багато додатків є, і можуть залишитися тільки в теорії.

Розглянувши, більш детально що таке ПЛІС, їх можливості, та області застосування та переваги, можна без сумніву сказати чому в багатьох сферах виробництва вони набувають популярність.

З ростом популярності ПЛІС, зростає і потрібність в інженерах, які вміють з ними працювати та програмувати їх. Потрібність в програмуванні на мовах опису апаратури з’явилась через ріст логічних схем. Типова логічна схема в графічному поданні містить на сторінці фрагмент, еквівалентний порядку 200 вентилів . відповідно, схема на 10 тис. вентилів буде об'ємом в 50 сторінок. Легко уявити собі, у що виллється (за часом) складання і введення в графічної нотації схем складністю 50 тис., 100 тис., 500 тис. вентилів і далі. Альтернативою малювання

деталізованих схем з низькорівневих елементів є мови опису апаратури високого рівня. Мови цього класу називають мовами HDL (Hardware Description Language). Вони не тільки забезпечують компактну запис для проектуємої схеми, і дають значне скорочення трудомісткості і термінів розробки великих схем, а й спрощують міграцію, перенесення проекту нарізні варіанти інтегральних технологій, реалізацію їх з урахуванням специфіки технологій різних виробників. Розробник отримує можливість оцінювати варіанти реалізації проектованого пристрою при різних варіантах проектних обмежень, на різних технологіях, у різних виробників . Тому необхідно розглянути електронні проекти, що програмуються в ПЛІС, з точки зору програмного забезпечення.

1.2 Моделі життєвого циклу програмного забезпечення

Так, як електронні проекти, які зашиваються в ПЛІС, є програмним забезпеченням, розглянемо основні моделі життєвого циклу програмного забезпечення.

Життєвий цикл програмного забезпечення - ряд подій, що відбуваються з системою в процесі її створення і подальшого використання. Кажучи іншими словами, це час від початкового моменту створення програмного продукту, до кінця його розробки та впровадження. Життєвий цикл програмного забезпечення можна представити у вигляді моделей.

Модель життєвого циклу програмного забезпечення - структура, яка містить процеси, дії і завдання, які здійснюються в ході розробки, використання та супроводу програмного продукту.

Розглянемо безпосередньо існуючі моделі і оцінимо їх переваги та недоліки.

Каскадний життєвий цикл

Каскадний життєвий цикл (іноді званий водоспадний) заснований на поступовому збільшенні ступеня деталізації опису всієї системи, що

розробляється. Кожне підвищення ступеня деталізації визначає перехід до наступного стану розробки (рис. 1.1).

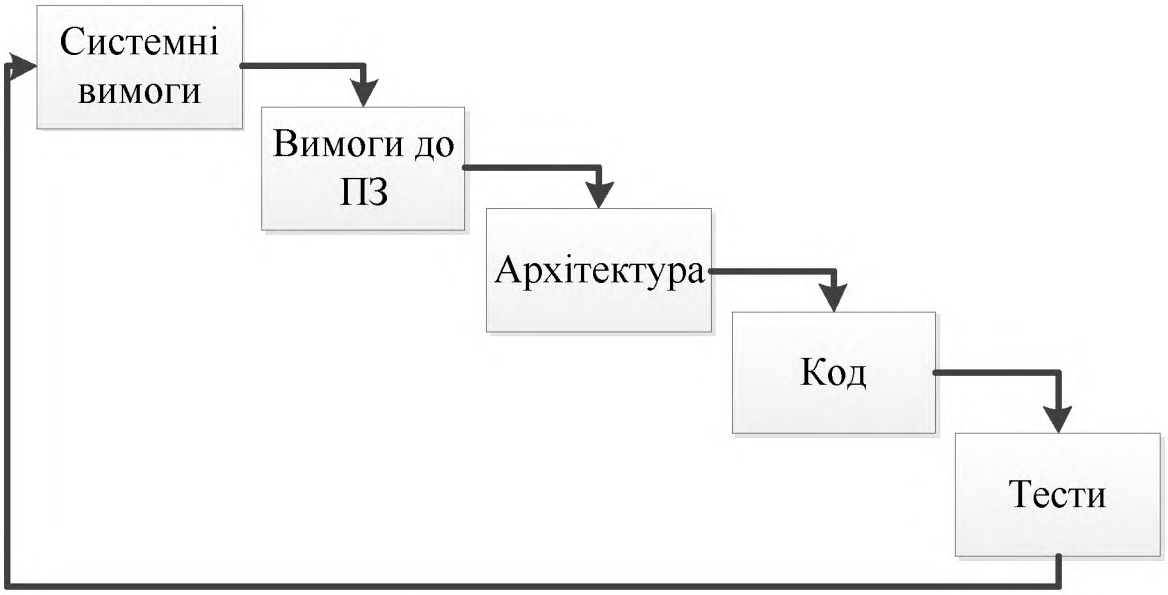

Рисунок 1.1 – Каскадна модель життєвого циклу

На першому етапі складається концептуальна структура системи, описуються загальні принципи її побудови, правила взаємодії з навколишнім світом - визначаються системні вимоги.

На другому етапі з системних вимогам складаються вимоги до програмного забезпечення - тут основна увага приділяється функціональності програмної компоненти, програмним інтерфейсам. Природньо, всі програмні комплекси виконуються на якій-небудь апаратній платформі . Якщо в ході проекту потрібна також розробка апаратної компоненти, паралельно з вимогами до програмного забезпечення йде підготовка вимог до апаратного забезпечення.

На третьому етапі на основі вимог до програмного забезпечення складається детальна специфікація архітектури системи - описуються розбиття системи по конкретних модулям, інтерфейси між ними, заголовки окремих функцій і т.п.

На четвертому етапі пишеться програмний код, відповідаючий детальній специфікації, на п'ятому етапі виконується тестування - перевірка відповідності програмного коду вимогам, визначеним на попередніх етапах.

Особливість каскадного життєвого циклу полягає в тому, що перехід до наступного етапу відбувається тільки тоді, коли повністю завершені всі роботи попереднього етапу. Тобто спочатку повністю готуються всі вимоги до системи, потім по ним повністю готуються всі вимоги до програмного забезпечення, повністю розробляється архітектура системи і так далі до тестування.

Звісно, що в разі досить великих систем такий підхід себе не виправдовує. Робота на кожному етапі займає значний час, а внесення змін до первинних документів або неможливо, або викликає лавиноподібні зміни на всіх інших етапах.

Як правило, використовується модифікація каскадної моделі, яка припускає повернення на будь-який з раніше виконаних етапів. При цьому фактично виникає додаткова процедура ухвалення рішення.

Дійсно якщо тести виявили невідповідність реалізації вимогам, то причина може критися:

1. В неправильному тесті;

2. В помилці кодування (реалізації);

3. В невірній архітектурі системи;

4. Некоректності вимог до програмного забезпечення і т.д.

Всі ці випадки потребують аналізу для прийняття рішення про те, на який етап життєвого циклу треба повернутися для усунення виявленої невідповідності.

V-подібний життєвий цикл

В якості своєрідної «роботи над помилками» класичної каскадної моделі стала застосовуватися модель життєвого циклу, що містить процеси двох видів - основні процеси розробки, аналогічні процесам каскадної моделі і процеси верифікації, що представляють собою ланцюг зворотного зв'язку стосовно до основних процесів (рис. 1.2).

Таким чином, в кінці кожного етапу життєвого циклу розробки, а часто і в процесі виконання етапу, здійснюється перевірка взаємної коректності вимог різних рівнів.

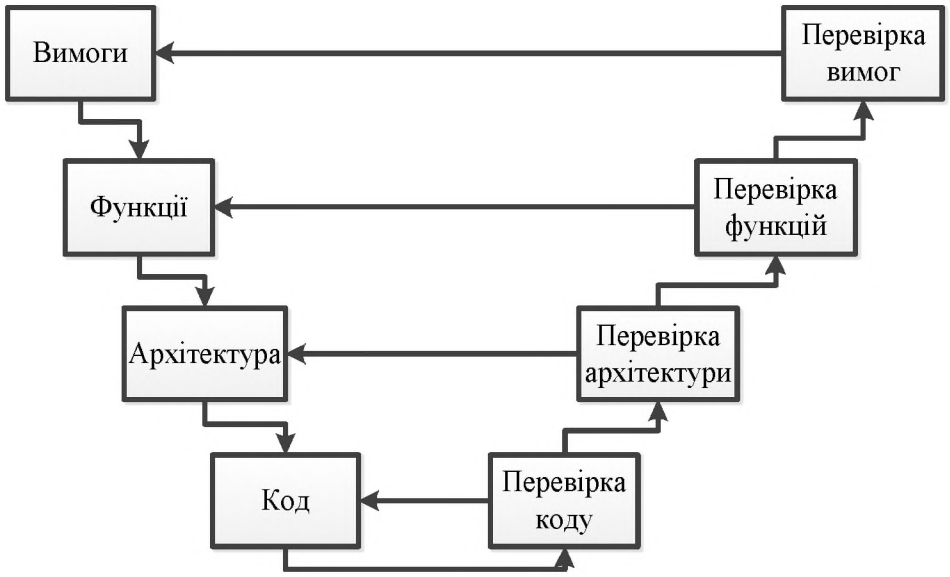

Рисунок 1.2 – V-подібний життєвий цикл

Дана модель дозволяє більш оперативно перевіряти коректність розробки, проте, як і в каскадної моделі передбачається, що на кожному етапі розробляються документи, що описують поведінку всієї системи в цілому.

Спіральний життєвий цикл

Обидва розглянутих типи життєвих циклів припускають, що заздалегідь відомі всі вимоги користувачів або, принаймні, передбачувані користувачі системи настільки кваліфіковані, що можуть висловлювати свої вимоги до майбутньої системи, не бачачи її перед очима.

Звісно, така картина досить утопічна, тому поступово з'явилося рішення, що виправляє основний недолік V-образного життєвого циклу – припущення про те, що на кожному етапі розробляється черговий повний опис системи. Цим рішенням стала спіральна модель життєвого циклу (рис. 1.3).

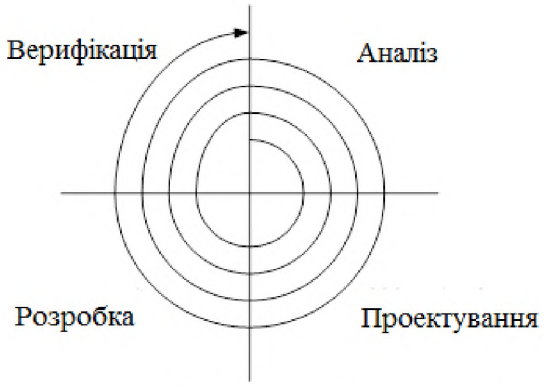

Рисунок 1.3 – Спіральний життєвий цикл

У спіральної моделі розробка системи відбувається повторюваними етапами - витками спіралі. Кожен виток спіралі – один каскадний або V -подібний життєвий цикл . Наприкінці кожного витка виходить закінчена версія системи, що реалізує деякий набір функцій. Потім вона пред'являється користувачеві, на наступний виток переноситься вся документація, розроблена на попередньому витку, і процес повторюється.

Таким чином, система розробляється поступово, проходячи постійні узгодження з замовником. На кожному витку спіралі функціональність системи розширюється, поступово доростаючи до повної.

Екстремальне програмування

Реалії останніх років показали, що для систем, вимоги до яких змінюються досить часто, необхідно ще більше зменшити тривалість витка спірального життєвого циклу. У зв'язку з цим зараз стали вельми популярними швидкі життєві цикли розробки, наприклад життєвий цикл в методології eXtreme Programming (XP).

Основна ідея життєвого циклу екстремального підходу - максимальне укорочення тривалості одного етапу життєвого циклу і тісна взаємодія із замовником. По суті, на кожному етапі відбувається реалізація і тестування однієї функції системи, після завершення яких, система відразу передається замовнику на перевірку або експлуатацію.

Основна проблема даного підходу – інтерфейси між модулями, що реалізують цю функцію. Якщо у всіх попередніх типах життєвого циклу інтерфейси досить чітко визначаються на самому початку розробки, оскільки заздалегідь відомі всі модулі, то при екстремальному підході інтерфейси проектуються «на льоту», разом з розроблювальними модулями.

Порівняння різних типів життєвого циклу і допоміжні процеси

Особливості розглянутих вище типів життєвого циклу зведені в таблицю 1.1. З неї можна бачити, що різні типи життєвих циклів застосовуються залежно від планованої частоти внесення змін в систему, термінів розробки і її складності. Життєві цикли з більш короткими фазами більше підходять для розробки систем,

вимоги до яких ще не устоялися і виробляються у взаємодії з замовником системи під час її розробки.

Таблиця 1.1 – Порівняння різних типів життєвого циклу

Тип життєвого циклу	Довжина циклу	Верифікація та внесення змін	Інтеграція окремих компонент системи
Каскадний	Всі етапи розробки системи. Довгий	Наприкінці розробки всієї системи. Рідко.	Чітко визначені до початку кодування інтерфейси.
V-подібний	Всі етапи розробки системи. Довгий	Наприкінці повної розробки кожного з етапів системи. Середньо.	Рідко змінювані інтерфейси.
Спіральний	Розробка однієї версії системи. Середній.	Наприкінці розробки кожного з етапів версії системи. Середньо.	Періодично змінювані інтерфейси, рідко міняється в межах версії.
XP	Розробка однієї історії. Короткий.	Наприкінці розробки кожної історії. Дуже часто.	Часто змінювані інтерфейси.

У наведеному вище описі різних моделей життєвого циклу по суті піднімався тільки один процес – процес розробки системи. Насправді в будь-якої моделі життєвого циклу можна побачити чотири види процесів:

1. Основний процес розробки;
2. Процес верифікації;
3. Процес управління розробкою;
4. Допоміжні (підтримуючі) процеси.

Процес верифікації – процес, спрямований на перевірку коректності розроблюваної системи та визначення ступеня її відповідності вимогам замовника.

Процес управління розробкою – окрема дисципліна, на управління дуже сильно впливає тип життєвого циклу основного процесу розробки. По суті, чим коротше один етап життєвого циклу, тим активніше управління, і тим більше завдань стоїть на менеджері проекту. При класичних схемах досить просто побудувати ієрархічну піраміду підпорядкованості, у якої кожен нижчий менеджер відповідає за розробку певної частини системи . У XP- підході немає жорсткого поділу системи на частини і менеджер повинен охоплювати всі історії.

При цьому процес управління активний протягом усього життєвого циклу основного процесу розробки.

Допоміжні (підтримуючі) процеси забезпечують своєчасне створення всього, що може знадобитися розробнику або кінцевому користувачеві. Сюди входить підготовка документації користувача, підготовка приймально-здавальних тестів, управління конфігураціями та змінами, взаємодія із замовником і т.д. Взагалі кажучи, допоміжні процеси можуть існувати протягом всього життєвого циклу розробки, а можуть бути своєрідними стиками ланок між процесом розробки та процесом експлуатації.

Існують значні подібності між FPGA і процесами розробки програмного забезпечення. IEC 61508-3 рекомендує V-модель, для розробок пов'язаних з відповідальним за безпеку програмним забезпеченням. V-модель вимагає чітко структурований процес розробки та модульну структуру програмного забезпечення, для запобігання та контролю систематичних помилок.

1.3 Функціональне тестування

Компанія Altera, згідно рекомендованої стандартом IEC 61508-3 V-моделі, надає V-модель для розробки систем на основі FPGA (рис. 1.4).

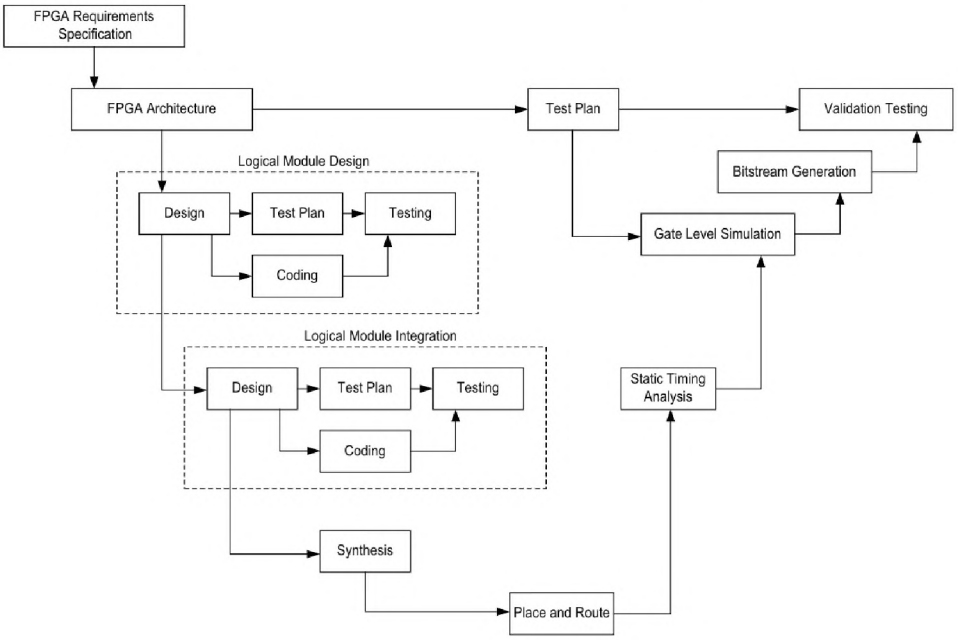

Рисунок 1.4 – V-модель для розробки систем на основі FPGA (Altera)

Тестування програмного забезпечення мовою VHDL відноситься до наступних етапів, які виділені на рис.1.4:

Розробка логічного модуля

Розробка. Цей етап V-моделі описує фазу розробки кожного модуля, який був визначений на етапі архітектури FPGA. На цьому етапі необхідно детально документувати метод отримання вимог на модуль.

Тест план (Опис тестів). На даному етапі необхідно з функціональної специфікації на модуль сформулювати специфікацію тестів для цього модуля.

Кодування. На цьому етапі детальна функціональна специфікація на модуль переводиться і синтезується в опис дизайну, у формі опису основних функцій мовою VHDL/Verilog.

Тестування. Цей етап включає в себе формування і виконання ТВ. Після виконання тестів формується звіт з тестування.

Інтеграція логічного модуля.

Розробка і Тест план (Опис тестів). Ці етапи мають такі ж кроки, як і на етапі розробки логічного модуля.

Кодування. На цьому етапі описується інтеграція розроблених на попередніх етапах модулів. Ці модулі комбінуються для створення високорівневих функцій і в кінцевому підсумку - дизайну верхнього рівня FPGA.

Тестування. Ці етапи мають такі ж кроки, як і на етапі розробки логічного модуля. Проте, верифікація концентрується на високорівневих блоках (функціях) або на тестуванні всього чіпу.

Для тестування програмного забезпечення мовою VHDL, слідуючи V-моделі використовуються такі види тестування, як статичний аналіз коду, огляд коду і функціональне тестування.

Функціональне тестування було обране для створення процедури через те, що воно є найбільш складнішим видом тестування для ПЗ ФБ розроблених на мові VHDL.

Функціональне тестування розглядає заздалегідь вказану поведінку і ґрунтується на аналізі специфікацій функціональності компонента або системи в цілому.

Функціональні тести ґрунтуються на функціях, виконуваних системою, і можуть проводитися на всіх рівнях тестування. Як правило, ці функції описуються у вимогах, функціональних специфікаціях або у вигляді випадків використання системи.

Тестування функціональності може проводитися у двох аспектах:

1. Вимоги;
2. Бізнес-процеси.

Тестування в перспективі «вимоги» використовує специфікацію функціональних вимог до системи як основу для дизайну тестових випадків (Test Cases). У цьому випадку необхідно зробити список того, що буде тестуватися, а що ні, пріоритезувати вимоги на основі ризиків (якщо це не зроблено в документі з вимогами), а на основі цього пріоритезувати тестові сценарії (test cases). Це дозволить сфокусуватися і не упустити при тестуванні найбільш важливий функціонал.

Тестування в перспективі «бізнес-процеси» використовує знання цих самих бізнес-процесів, які описують сценарії щоденного використання системи. У цій перспективі тестові сценарії (test scripts), як правило, ґрунтуються на випадках використання системи (use cases).

Створення якісної процедури функціонального тестування, є важливим моментом в розробці ПЗ на мові VHDL. Тому необхідно розглянути призначення цієї процедури та її основні елементи.

1.4 Призначення процедури функціонального тестування

Процедура функціонального тестування призначена для визначення основних етапів процесу проведення ФТ та використовується для перевірки коректності реалізації всіх функціональних вимог до ПЗ.

Основні складові частини процедури ФТ:

Ціль процедури ФТ – визначити яку ціль ставить перед собою дана процедура;

Об'єкт процедури ФТ – визначити, до якого об'єкту можливе застосування даної процедури;

Складання плану проведення ФТ – опис основних елементів плану проведення ФТ та вимоги до звітного документу (План ФТ);

Складання специфікації тестів ФТ – опис основних елементів специфікації тестів та вимоги до звітного документу (Специфікація ФТ);

Методика написання ТВ – опис процедури створення ТВ;

Методика складання ТС – опис процедури складання ТС;

Складання керівництва користувача ТВ для ФТ – опис основних елементів керівництва користувача ТВ для ФТ;

Опис проведення ФТ – детальний покроковий опис проведення ФТ;

Аналіз результатів ФТ – опис проведення аналізу результатів ФТ, основні критерії аналізу;

Складання звіту ФТ – опис основних елементів звіту та вимоги до звітного документу (Звіт ФТ).

Розглянувши основні відомості про ПЛІС, ФТ та процедуру ФТ та проаналізувавши їх, було вирішено створити найбільш оптимальну процедуру ФТ, за вимогами IEC 61508, яка б мала модель тестової системи, що відповідає критеріям SIL 3 (пункт 2.1) і мала переваги над існуючими.

Висновки до розділу 1

В першому розділі виконано аналіз поточного росту складності НВІС, було підтверджено актуальність використання ПЛІС. Функції ПЛІС дозволяють реалізовувати їх можливості у чи малій низці сфер, а саме у цифровій обробці сигналів, у ролі вбудованих мікроконтролерів, для фізичного рівня передачі даних, у системах з перебудовуємою архітектурою та автоматизованих системах

управління. Через збільшення використання ПЛІС, росте й актуальність використання таких мов програмування як VHDL. Так як для ПЛІС на мові VHDL, створюється ПЗ, то були проаналізовані моделі життєвого циклу ПЗ та обрано V-модель, так як V-модель вимагає чітко структурований процес розробки та модульну структуру програмного забезпечення, для запобігання та контролю систематичних помилок, що є дуже актуальним, особливо при використані ПЛІС в автоматизованих системах управління. Згідно цієї моделі розглянуто місце ФТ в життєвому циклі всього продукту, та визначено задачі ФТ, так як і призначення процедури ФТ та її зміст.

РОЗДІЛ 2
РОЗРОБКА ПРОЦЕДУРИ ФУНКЦІОНАЛЬНОГО ТЕСТУВАННЯ

2.1 Модель тестової системи

Більшість процедур функціонального тестування схожа за своєю будовою, так як вони створюються за міжнародним стандартом IEEE 829, в якому описаний основний підхід до побудови тестових процедур. Одною з основних відмінностей процедур функціонального тестування є модель тестової системи. Саме на моделі тестової системи, зроблено акцент в розробленій процедурі функціонального тестування. Важливим фактором при розробці моделі тестової системи, були поставлені вимоги до функціонального тестування згідно стандарту IEC 61508-3 2010 Table B.3, на який я орієнтувався при розробці процедури функціонального тестування, через те, що моя робота пов'язана з забезпеченням якості програмного забезпечення, яка відповідає третьому рівню повноти безпеки(Рівень повноти безпеки або Safety Integrity Level(SIL) – рівень забезпечення цілісності безпеки, який є мірою безпеки при роботі електричного, електронного обладнання, а також, програмованих електронних систем.). Для SIL 3 вимогами для функціонального тестування є використання розділення тестових векторів на класи еквівалентності та перевірки граничних значень при виборі тестових векторів для тестування функціонального блоку. Згідно [37], розділення тестових векторів на класи еквівалентності дозволяє знайти помилку у цілому еквівалентному класі, лише вибравши одне значення з цілого класу. Якщо дивитись більш об'єктно, то якщо при виконанні якоїсь операції у коді, перевірити її лише одним значення, і виявиться помилка, то і при інших значеннях буде помилка. Також згідно [37] перевірка граничних значень при виборі тестових векторів, являє собою перевірку граничних значень класів еквівалентності, тобто окрім будь якого значення з класу еквівалентності, необхідно перевірити мінімальне та максимальне значення класу. Якщо ж на одну операцію, порт, сигнал і т.д. обрано 2 та більше класи еквівалентності, необхідно

перевіряти перехідні значення між цими класами. Описані вище 2 методи вибору тестових векторів значно зменшують час, який затрачується на роботу.

Для порівняння з існуючими моделями тестової системи була використана модель представлена в [35] під назвою «SELF CHECKING TESTBENCH» (рис. 2.1). Основними функціями цієї моделі є:

1. Генерація тестових векторів;
2. Отримувач вихідних сигналів;
3. Монітор вихідних сигналів;
4. Зберігач очікуваних результатів;
5. Порівняння отриманих результатів з очікуваними результатами;
6. Підрахунок функціонального покриття.

Рисунок 2.1 – «SELF CHECKING TESTBENCH»

Основними недоліками даної моделі (саме для моєї роботи) є генерація тестових векторів та відсутність автоматичної генерації звіту. Також не малозначущим є те, що дана модель є розгалужена, порівняння результатів та монітор це 2 окремих процеси, хоча можливо оптимізувати ці 2 процеси та створити один. Тому мною була розроблена більш досконала на мій погляд модель. Для порівняння були приведені функції розробленої моделі (рис. 2.2):

1. Задання тестових векторів – після аналізу вимог для ФБ, інженер вибирає тестові вектори, саме ті які йому необхідні, та задає їх у ТВ;

2. Монітор, який розраховує на основі тестових векторів очікуваний результат для порівняння з отриманим результатом – ця функція дозволяє відстежувати кожну помилку по вимогам, відштовхуючись від порівняння отриманого результату з очікуваним;

3. Генерація звіту на основі даних отриманих при порівнянні результатів – виводить детальний звіт по кількості помилок, якщо вони були знайдені, показує які вимоги порушені, а які ні, та виводить сумарну інформацію по помилкам. В разі знаходження помилки, буде виведена інформація про те за якою вимогою знайдена помилка, час помилки, та вивід тестового вектору при якому була отримана помилка (опціонально).

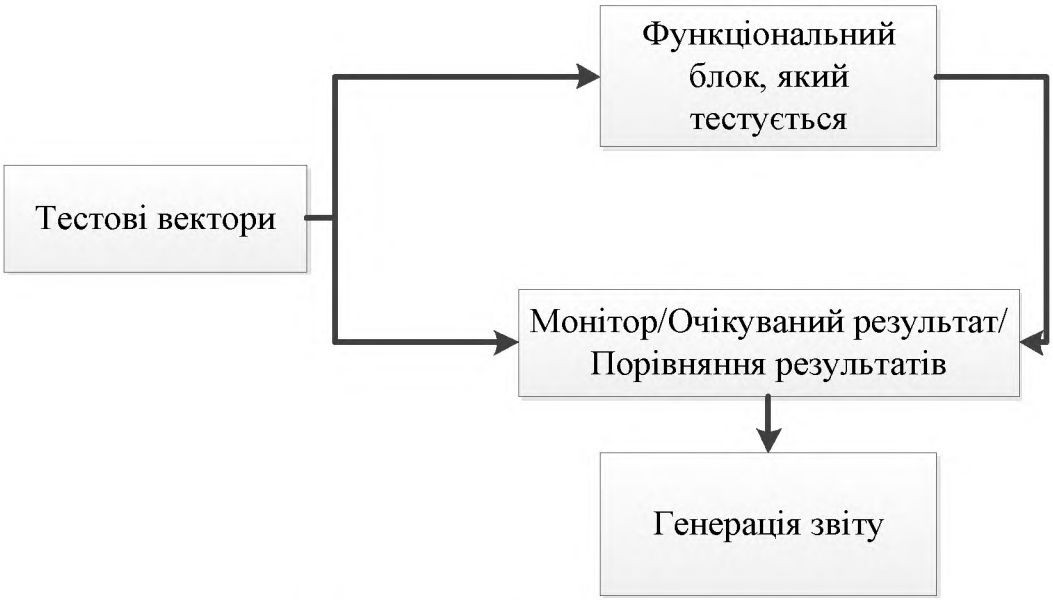

Рисунок 2.2 – Розроблена модель тестової системи

Таким чином відмінностями розробленої моделі є:

1. Ручне задавання тестових векторів;

2. Більш оптимізована система моніторингу, яка за рахунок меншої кількості ітерацій, швидше проводить порівняння результатів.

3. Генерація звіту, що зменшує час на аналіз та оформлення результатів.

4. Відсутність підрахунку функціонального покриття, через вимоги стандарту IEC 61508-2010 для SIL 3 (Використання еквівалентних класів та перевірки граничних значень).

Для виконання більшої частини функцій моделі, крім стандартних функцій VHDL, були використані функції розробленої на мові VHDL бібліотеки monitor_pkg.vhd, опис якої представлено в пункті 3.2. Проведення функціонального тестування для функціонального блоку на мові VHDL.

З результатів порівняння, можна сказати, що розроблена модель, є більш оптимізована та дозволяє швидко аналізувати та оформлювати результати, але є вузькоспеціалізованою під тестування функціональних блоків, розроблених за стандартом IEC 61508-2010 для SIL 3, на мові VHDL.

Після аналізу та опису моделі тестової системи, необхідно описати, процедуру проведення функціонального тестування. Розділимо функціональне тестування на два етапи, планування проведення тестування та опис проведення тестування.

2.2 Планування проведення функціонального тестування

Перед тим, як перейти до планування проведення функціонального тестування (далі по тексту ФТ), необхідно визначити, для чого необхідна процедура ФТ, тобто її ціль та об’єкт.

Даний документ визначає основні етапи процесу проведення ФТ для функціональних блоків (далі по тексту ФБ), розроблених на мові VHDL для програмування FPGA – інтегральних схем. Процедура ФТ використовується для перевірки коректності реалізації всіх функціональних вимог до ФБ на мові VHDL, описаних в специфікації вимог на ФБ.

Ціллю ФТ є підтвердження відповідності результатів виконання реалізованих функцій ФБ очікуваним результатам, встановлених у вихідних документах, таких як план та специфікація ФТ, складених на основі описових документів розробленого ФБ (наприклад, специфікації вимог).

ФТ може застосовуватись до ФБ розробленого на мові VHDL, для якого закінчений етап розробки коду, складений план функціонального тестування та специфікація тестів ФТ чи об’єднаний документ (План та специфікація ФТ).

Планування проведення ФТ полягає в складені плану ФТ. Ціллю складення плану проведення ФТ є складення порядку та розкладу виконання процесу ФТ.

Згідно з вимогами стандарту IEEE 829 план проведення ФТ повинен містити:

1. Вступ, в я кому формулюється ціль ФТ;

2. Короткий опис об'єкту, для якого буде виконано ФТ;

3. Ідентифікацію функцій об'єкту, для якого буде проведено тестування, та пов'язану з кожною функцією чи набором функцій специфікацію тестів ФТ;

4. Ідентифікацію функцій об'єкту, що не підлягають ФТ;

5. Підхід до ФТ, що визначає методику та інструментальні засоби (далі по тексту ІЗ) використовувані під час виконання ФТ, а також перелік необхідних апаратних засобів та програмного забезпечення;

6. Список людей, відповідальних за виконання ФТ;

7. Критерії призупинення тестування та вимоги по відновленню тестування (наприклад, тестування закінчується після виконання усіх запланованих тестів, при умові досягнення необхідного покриття вимог, а відновлюється після усунення дефектів та/чи невідповідностей, а також у разі недостатнього покриття вимог;

8. Перелік документів, які повинні бути зіставлені у відповідності з даним планом проведення ФТ (наприклад, специфікації тестів ФТ з описом тестів, звіт про виконання ФТ).

Специфікація тестів ФТ складається в відповідності з вимогами стандарту IEEE 829 на основі плану проведення ФТ та повинна описувати:

1. Деталізацію підходу до ФТ;

2. Визначення характеристик та функцій компонентів, для яких буде і не буде проведено ФТ;

3. Визначення складу тестових наборів (Test Cases, далі по тексту ТС);

4. Визначення критеріїв виконання (не виконання) тестових наборів;

5. Визначення переліку тестових симулючих (задаючих) модулів (Test Benches, далі по тексту ТВ) та змінюємих в них при реалізації конкретних ТС

параметрів(тестових векторів), які повинні виконуватись при проведенні ФТ, із зазначенням переліку перевіряємих з їх допомогою функцій та очікуваних результатів (наборів вихідних даних та часових діаграм).

В межах даної роботи план проведення та специфікація тестів ФТ об'єднані в один документ. Приклад оформлення плану та специфікації ФТ наведено в додатку А.

2.3 Опис проведення функціонального тестування

Для виконання ФТ необхідно створити деяку середу – тестовий симулюючий (задаючий) модуль (Test Bench), який забезпечить запуск та виконання тестового ФБ, передасть йому тестові вектора та зафіксує реальні вихідні данні, отримані в результаті моделювання функціонування ФБ з заданими тестовими векторами.

При створенні ТВ на мові VHDL, можна використовувати ІЗ текстового редактору, а також можливе використання шаблону ТВ, створеного за допомогою ІЗ розробки ФБ на мові VHDL. В даній процедурі, у якості прикладу, розглянуто створення ТВ (файл формату *.vht) за допомогою ІЗ Notepad++.

Для створення ТВ, необхідно в ІЗ Notepad++ зберегти файл в форматі *.vht та заповнити його необхідним VHDL кодом (пункт 3.2).

При заповнені файлу ТВ на мові VHDL він повинен мати наступну типову структуру:

1. Не маючу портів сутність ТВ (Test Bench Entity);

2. Компонент реалізації тестує мого модулю, який установлює відношення між ТВ та тестуємим модулем;

3. Стимулюючі впливи (тестові вектори) – набір сигналів, які об'являються в середині архітектури ТВ і присвоюються портам тестує мого модуля в його реалізації;

4. Функції бібліотеки monitor_pkg.vhd.

Результатом тестових векторів є часові діаграми поведінки процесів у тестуємому модулі. Реакція тестує мого модуля на тестовий вектор може бути зареєстрована за допомогою ІЗ моделювання та тестування, або результати можуть бути записані у окремий файл формату *.txt.

Створені ТВ повинні бути описані в керівництві користувача (UserManual). Цілю створення керівництва користувача ТВ є визначення поведінки ТВ при їх використанні під час проведення ФТ, опис обмежень на їх використання (якщо вони є), а також опис ТС, що провіряються даними ТВ.

Керівництво користувача ТВ повинно включати в себе:

1. Концептуальну структуру ТВ;

2. Перелік використовуваних ТВ для виконання ФТ;

3. Обмеження/допущення при використанні ТВ;

4. Порядок виконання ТВ.

5. Безпосередньо для ФТ, окрім ТВ, необхідно визначити тестові набори – ТС, які будуть тестовими векторами при моделювання та тестуванні ФБ.

Кожен ТС, повинен мати у собі:

1.Тестові вектори для ФТ;

2.Вказівки, яким саме ТВ, виконується даний ТС;

3. Опис начальних умов чи/або вимог (при необхідності надається опис будь-яких спеціальних обмежень на процес тестування, який виконується ТВ);

4. Опис процесу виконання ТС, який полягає в визначенні покрокових дій в рамках ТС, наприклад: опис послідовності дій, необхідних, для підготовки тесту, для початку його виконання, під час виконання, для зупинки тестування, коли трапляються незаплановані події, для перезапуску тесту;

5. Очікувані вихідні значення (їх опис чи опис);

6. Критерії оцінки результату виконання тесту.

7. Ціллю виконання любого ТС є або демонстрація наявності дефекту в ФБ, або доказ його відсутності.

Основним джерелом інформації, для створенні ТС, є описова документація по розробленому ФБ, де повинні бути визначені функціональні вимогу, що

описуються поведінку ФБ з позиції того, що повинен виконувати ФБ, у різних ситуаціях (реакція ФБ на різні тестові вектори).

Для того, щоб виявити різні дефекти в функціонуванні ФБ необхідно використовувати різні групи (типи) вхідних даних, в залежності від особливостей призначення ФБ, наприклад:

Граничні значення (даних).

Окремий вид допустимих даних, передача яких на вхід ФБ може виявити дефект. Зазвичай за допомогою тестування граничних умов виявляються проблеми з арифметичним порівнянням чисел чи з літераторами циклів.

В даному випадку використовуються вхідні значення, які знаходяться в середині допустимого діапазону. Один із способів перевірки стійкості функціонування ФБ на значеннях, близьких до граничних – створювати для кожного входу, як мінімум 3 тестових вектори:

1. Значення в середині діапазону;

2. Мінімальне значення;

3. Максимальне значення.

Для ще більшої впевненості в працездатності ФБ використовують 5 тестових вектори:

1. Значення в середині діапазону;

2. Мінімальне значення;

3. Мінімальне значення +1;

4.Максимальне значення;

5.Максимальне значення -1.

Такий спосіб перевірки називається перевіркою на граничних значеннях. Така перевірка дозволяє виявити проблеми, пов’язані з виходом за границі діапазону.

Відсутність даних.

Дефекти можуть проявлятися в випадку, якщо на вхід ФБ не передається ніяких даних чи передаються дані нульового розміру.

Повторне введення даних.

У випадку повторної передачі на вхід ФБ тих самих даних, можуть проявлятися відмінності в вихідних даних, не передбачені в вимогах. Як правило, дефекти такого типу виявляються в результаті того, що ФБ не встановлює внутрішні змінні в вихідний стан.

Невірні дані.

При перевірці поведінки ФБ необхідно перевіряти його поведінку при передачі даних, не передбачених вимогами. Невірні дані, як і допустимі, також можливо розділяти на різні класи еквівалентності.

Реініціалізація системи.

Механізми повторної ініціалізації ФБ під час його роботи також може мати дефекти. В першу чергу ці дефекти можуть проявлятися в тому, що не всі внутрішні дані системи, після ре ініціалізації будуть у первинному стані. В результаті може трапитись збій при роботі ФБ.

Класи еквівалентності.

При розробці тестових векторів може виникнути така ситуація, в якій різні вхідні значення приводять до одних і тих же реакцій ФБ. Якщо при цьому вхідні значення мають щось спільне, то можливо об'єднання цих значень в класи еквівалентності, тобто виконання еквівалентного розбиття множини допустимих вхідних значень. Розбиття на класи еквівалентності – в першу черну, спосіб зменшення необхідного числа тестових векторів. Розбиття на класи еквівалентності особливо корисне, кола на вхід ФБ може бути подано більша кількість різних значення, тестування кожного можливого значення призвело б, до збільшення об'єму тестування.

Розглянуті вище граничні умови можуть служити прикладом класів еквівалентності:

1. Значення з середини інтервалу;

2. Граничне значення.

Таким чином, тестування граничних умов є окремим випадком тестування з використанням класів еквівалентності – замість того, щоб тестувати всі недопустимі значення, вибираються тільки сусідні з граничними.

При визначені класів еквівалентності слід керуватися наступними правилами:

Завжди буде, як мінімум, 2 класи: коректний та некоректний;

Якщо вхідні умови визначають діапазон значень, то, як правило, буває три класи: менше за діапазон, в середині діапазону та більше за діапазон(значення на кінцях діапазону можуть трактуватися, як граничні значення);

Якщо елементи діапазону обробляються по різному, то кожному варіанту обробки будуть відповідати різні вимоги.

В стандарті IEC 61508-3 для ФТ по SIL 3, вказаний як рекомендований метод вибору тестових векторів «класи еквівалентності та розділ вхідних значень, включаючи аналіз граничних значень» (Equivalence classes and input partition testing, including boundary value analysis).

Методика виконання ФТ.

Після закінчення етапу огляду керівництва користувача ТВ у строгій відповідності з зіставленим планом та специфікацією тестів ФТ, безпосередньо здійснюється виконання ФТ для розробленого ФБ.

Проведення процедури по крокового автоматизованого виконання ФТ за допомогою ІЗ Modelsim, детально описано в пункті 4.1 Опис середи моделювання. В цьому пункті описана загальна методика виконання ФТ:

Крок 1. Після інсталяції та запуску на персональному комп'ютері середи моделювання ІЗ Modelsim, необхідно створити директорію з іменем проекту та помістити в неї файли формату *.vhd, також файли формату *.vht, які містять ТВ, та файли формату *.do, які містять сценарії тестування, для проведення моделювання.

Крок 2. Запустити середу моделювання ІЗ Modelsim та указати директорію з якою буде вестись робота, тобто директорія створена на першому кроці.

Крок 3.

Запустити на виконання сценарій тестування, описаний у файлі *.do, який проведе компіляцію необхідних *.vhd та *.vht файлів, обере необхідні параметри симуляції та запустить симуляцію.

Крок 4. Після завершення симуляції проаналізувати результати тестування, тобто звіт який згенерувався після симуляції у файлі report.txt та переглянути результати кодового покриття за допомогою ІЗ Modelsim.

Для повторного моделювання необхідно завершити симуляцію командою quit –sim, введену у консоль ІЗ Modelsim, та повторити крок 3, 4.

Аналіз результатів ФТ

Під час проведення аналізу результатів ФТ розглядаються питання, невідповідності та/або дефекти, виявлені під час виконання ФТ, а також приймаються рішення про степінь важливості виявлених невідповідностей та/або дефектів. Під час аналізу представлених результатів, можуть бути знайдені нові невідповідності та/або дефекти, не виявлені на попередніх етапах.

Для виконання аналізу результатів ФТ, перевіряється наявність описової документації на розроблений код ФБ на мові VHDL, плану проведення та специфікації тестів ФТ.

Задачі, вирішувані під час проведення аналізу результатів ФТ:

1. Провести огляд звіту по результатам ФТ, якщо в результаті виконання ФТ були знайдені дефекти, тоді необхідно оцінити степінь впливу знайдених дефектів на інші документи, та етапи розробки ФБ, розроблені раніше в рамках розглядає мого конкретного проекту та визначити необхідність внесення в них змін.

2. Проаналізувати чи досягнуто необхідний рівень тестового покриття. В якості метрики оцінки якості ФТ використовується тестове покриття, яке являє собою платність покриття тестами вимог або виконує мого коду. Оцінка якості результатів ФТ здійснюється з використанням наступних підходів до його виміру.

Визначити покриття функцій або покриття вимог (в більшості випадків функції і вимоги рівносильні), яке представляє собою оцінку покриття тестами вимог(функцій) приведених в специфікації тестів ФТ. Для цього інженерам необхідно проаналізувати вимоги (функції) ФБ, які приведені в специфікації тестів ФТ та розрахувати покриття вимог (функцій) по формулі:

$$\mathrm{T}_{cov} = \frac{L_{cov}}{L_{total}} \cdot 100\%, \qquad (2.1)$$

де T_{cov}–покриття вимог,

L_{cov}–кількість вимог(функцій), перевірених тестовими наборами (ТС) приведеними в специфікації тестів ФТ,

L_{total}–загальна кількість вимог (функцій) визначених в специфікації тестів ФТ.

Проаналізувати покриття коду, яке являє собою оцінку покриття виконує мого коду ФБ на мові VHDL тестовими наборами (ТС), шляхом відстежування неперевірених в процесі ФТ ділянок VHDL-коду ФБ.

Аналіз повинен підтвердити, що повнота покриття тестовими наборами (ТС) структури коду ФБ, відповідає вимагаємому виду покриття та заданому проценту покриття.

Для виконання даної задачі необхідно проаналізувати таблицю з результатами кодового покриття (пункт 4.2) отриманої під час проведення ФТ за допомогою ІЗ ModelSim.

Аналіз повноти покриття коду ФБ тестовими наборами (ТС) може виявити частину коду, яка не виконалась в ході моделювання (тестування). Ця не виконувана частина коду може бути результатом:

1. Недоліків в формуванні ТС – тоді необхідно розширити перелік ТС для забезпечення покриття пропущеної частини коду при тестуванні;

2. Неадекватності в вимогах до реалізації ФБ – тоді необхідно модифікувати (уточнити) вимоги, розробити та виконати додаткові ТС;

3. Деактивованого коду, який не виконується при кожній активації;

4. Надмірність вимог;

5. Логіка роботи відповідних умовних операторів повинна бути переглянута;

6. Реалізація захисного VHDL-коду – ця частина коду виконується для запобігання виключних ситуацій, які можуть трапитись у процесі функціонування ФБ.

Фіксування усіх невідповідностей повинно бути у звіті з ФТ, показуючи його місце в об’єкті ФТ.

Огляд результатів проведення процедури ФТ.

Огляд результатів проведення процедури ФТ це стадія, під час якої класифікують причини дефектів, найдених під час виконання всіх етапів, визначених структурою даної процедури. Ця діяльність є важливим етапом для попередження виникання аналогічних дефектів під час подальшої роботи.

Задачі, вирішувані під час обзору результатів проведення процедури ФТ:

1. Здійснити вибірку та аналіз причин різних (по типам) дефектів з звітів по результатам проведення ФТ з цілю обговорення най частіш зустрічаємих з них;

2. Визначити потенційні причини виявлених типових та рідкісних дефектів.

Порядок оформлення сумарного звіту по результатам ФТ.

Всі дії по проведенню ФТ повинні бути детально задокументовані для обліку невідповідностей та/чи дефектів, а також відстежування процесу їх усунення. По завершенню проведення ФТ затверджується документальний сумарний звіт. Приклад звіту знаходиться в додатку Б.

В звіті про проведення ФТ повинні бути вказані:

1. Дані про об'єкт ФТ;

2. Атрибути ІЗ, використовуваних для проведення ФТ;

3. Опис порядку проведення ФТ;

4. Первинні та проміжні результати проведення ФТ;

5. Результати повторного проведення ФТ після усунення невідповідностей та/чи дефектів (якщо такі були знайдені в процесі проведення ФТ);

6. Загальні висновки за результатами проведення ФТ з указанням оцінки рівня відповідності результатів ФТ очікуваним результатам, встановленим в плануючих та нормативних документах, а також кількості знайдених, усунених та не усунених невідповідностей та/або дефектів.

Якщо знайдені невідповідності та/чи дефекти не усунені, то у звіті повинні бути зафіксовані пояснення причин, з яких вони буди не усунуті – по кожній такій невідповідності чи дефекту окремо повинна бути виконана та задокументована оцінка впливу такої невідповідності та/чи дефекту на необхідність внесення яких-небудь змін на попередніх етапах проекту (життєвого циклу продукту).

В даному розділі була описана модель тестової системи, та процедура проведення функціонального тестування ФБ на мові VHDL. За допомогою цієї процедури можливе проведення ФТ при наявності необхідних ІЗ. Були описані основні дії під час ФТ, види документів які потрібно оформити, та приведені в додатках приклади цих документів. Згідно розділу 2 можна провести процедуру на реальному прикладі. Розробка ФБ для проведення ФТ та реалізація проведення ФТ для нього, описані в наступному, третьому, розділі.

Висновки до розділу 2

В другому розділі розроблено процедуру ФТ, в якій описано створення нової моделі тестової системи. Приведено її основні функції. Розроблена модель, є більш оптимізована та дозволяє швидко аналізувати та оформлювати результати, але є вузькоспеціалізованою під тестування функціональних блоків, розроблених за стандартом IEC 61508-2010 для SIL 3, на мові VHDL. Після аналізу та опису моделі тестової системи, описано процедуру проведення функціонального тестування. Вказано ціль та об'єкт процедури ФТ. Описано розробку плану та специфікації ФТ. Вказано основні елементи цих документів. В межах даної процедури план та специфікація об'єднані в єдиний документ, який подано в додатку А. Описано проведення функціонального тестування, а саме опис ІЗ необхідних для виконання ФТ. Описано створення ТВ для ФТ, структуру ТВ, склад керівництва користувача ТВ. Описано підхід до вибору ТС. Описані можливі типи вхідних даних для ТС, а саме:

1. Граничні значення даних;
2. Відсутність даних;
3. Повторне введення даних;
4. Невірні дані;
5. Реініціалізація системи;
6. Класи еквівалентності.

Далі описано покрокову методику проведення ФТ та ІЗ які необхідні для виконання ФТ. Описано, як необхідно проводити аналіз результатів ФТ та оформлювати сумарний звіт з ФТ, який надано в додатку Б.

РОЗДІЛ 3
ПРОВЕДЕННЯ ФУНКЦІОНАЛЬНОГО ТЕСТУВАННЯ

3.1 Розробка функціонального блоку на мові VHDL

Для проведення функціонального тестування на мові VHDL, був реалізований мультиплексор (мультиплексор - пристрій, що має кілька вхідних портів даних, один або більше контролюючих вхідних портів і один вихідний порт. Мультиплексор дозволяє передавати сигнал з одного з вхідних портів на вихідний порт; при цьому вибір бажаного вхідного порту здійснюється подачею відповідної комбінації сигналів на контролюючі вхідні порти) з 4 вхідними портами даних, 2 контролюючими вхідними портами та 1 вихідним портом даних. Функціональна діаграма блока представлена на рисунку 3.1.

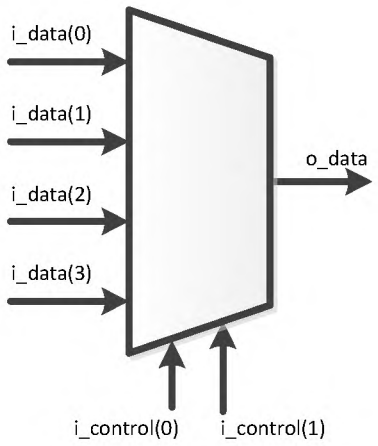

Рисунок 3.1 – Функціональна діаграма мультиплексора

Для ФБ, були складені вимоги, описані в таблиці 3.1. В промисловості основним джерелом вимог є замовник продукції. Вимогу трасуються з технічного завдання на продукт, у опис архітектури продукту і після опису архітектури продукту у детальний опис продукту, в якому зазначене функціональне призначення продукту, його функціональна діаграма, перелік інтерфейсів, взаємодія компонентів та інші особливості роботи продукту. Інтерпретацію вимог технічного завдання у вимоги детального опису в більшості випадків здійснюють розробники продукту.

Таблиця 3.1 – Вимоги до ФБ

Ідентифікатор вимоги	Опис вимоги
MUX.01	Повинен формувати вихідні дані відповідно до контролюючого сигналу
MUX.02	Контролюючий сигнал повинен вибирати номер порту вхідних даних, з якого будуть подаватися дані на вихідний порт

Для реалізації даного мультиплексора на мові VHDL, створено ФБ з 2 вхідними портами (i_data та i_control) і 1 вихідним портом (o_data) у файлі mux.vhd. Щоб отримати 4 вхідних порти даних, 2 контролюючих вхідних порти необов'язково створювати 6 портів, достатньо використати тип даних std_logic_vector, який дозволяє створювати n - бітну шину, тому 6 портів реалізовано 2 портами, типу std_logic_vector, а саме 4 бітний i_data та 2 бітний i_control. Для створення вихідного порту, був використаний тип даних std_logic, який створює бітову лінію. Для того, щоб використовувати ці типи даних, необхідно підключити бібліотеку std_logic_1164. Бібліотека std_logic_1164 містить визначення типів, підтипів, і функцій, які розширюють VHDL в багатозначну логіку. Цей пакет не є частиною стандарту VHDL, це окремий стандарт того ж самого розділу стандартизації Головною причиною для розвитку та стандартизації пакета std_logic_1164 була потреба в більш послідовних значеннях з функцією роздільної здатності. Типи std_logic і std_logic_vector фактично стали індустріальними стандартами. Лістинг створення цих портів, на мові VHDL, тобто сутність ФБ, та підключення бібліотеки std_logic_1164 представлено на рисунку 3.2.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
port(
i_data          : in std_logic_vector (3 downto 0);
i_control       : in std_logic_vector (1 downto 0);
o_data: out std_logic
);
end mux;
```

Рисунок 3.2. – Сутність функціонального блоку та підключення бібліотеки

Далі необхідно описати архітектуру ФБ, тобто частину VHDL коду, в якій описаний принцип роботи блоку. Для цього, с початку, необхідно визначити, при якій комбінації контролюючих сигналів буде обрано певний номер порту вхідних даних. Це можливо визначити, склавши таблицю істинності для даного ФБ:

Таблиця 3.2 – Таблиця істинності ФБ

i_control(1)	i_control(0)	i_data
0	0	i_data(0)
0	1	i_data(1)
1	0	i_data(2)
1	1	i_data(3)

Використавши данні з таблиці 3.2, опишемо роботу ФБ на мові VHDL (рис. 3.3).

```
architecture arch of mux is
begin
mux_p:
process(i_control,i_data)
begin
        case i_control is
                when "00" =>
                        o_data <= i_data(0);
                when "01" =>
                        o_data <= i_data(1);
                when "10" =>
                        o_data <= i_data(2);
                when others =>
                        o_data <= i_data(3);
        end case;
end process mux_p;
end arch;
```

Рисунок 3.3 – Архітектура функціонального блоку

На цьому розробка ФБ на мові VHDL завершена, тепер необхідно перевірити його працездатність, тобто чи відповідає створений ФБ, тим вимогам, що ставились до нього.

3.2 Проведення функціонального тестування для функціонального блоку на мові VHDL

Функціональне тестування починається з аналізу вимог до ФБ. Проаналізувавши вимоги, було визначено, що найбільш якісним та швидким методом функціонального тестування для розробленого функціонального блоку, є створення моделі мультиплексору, та порівняння вихідних даних моделі з вихідними даними створеного ФБ.

Після цього були визначені тестові вектори, тобто вхідні дані які будуть подаватися на входи ФБ. Так, як в даному ФБ всього два вхідних порти розрядністю 4 та 2 біти, вхідними даними будуть всі можливі комбінації. Тобто маємо 64 варіанти вхідних даних, які містяться в таблиці 3.3.

Таблиця 3.3 – Тестові вектори для функціонального тестування

№	i_control	i_data
1	2	3
1.	00	0000
2.	00	0001
3.	00	0010
4.	00	0011
5.	00	0100
6.	00	0101
7.	00	0110
8.	00	0111
9.	00	1000
10.	00	1001
11.	00	1010
12.	00	1011
13.	00	1100
14.	00	1101
15.	00	1110
16.	00	1111
17.	01	0000
18.	01	0001
19.	01	0010
20.	01	0011
21.	01	0100
22.	01	0101
23.	01	0110
24.	01	0111
25.	01	1000
26.	01	1001
27.	01	1010

Продовження таблиці 3.3

1	2	3
28.	01	1011
29.	01	1100
30.	01	1101
31.	01	1110
32.	01	1111
33.	10	0000
34.	10	0001
35.	10	0010
36.	10	0011
37.	10	0100
38.	10	0101
39.	10	0110
40.	10	0111
41.	10	1000
42.	10	1001
43.	10	1010
44.	10	1011
45.	10	1100
46.	10	1101
47.	10	1110
48.	10	1111
49.	11	0000
50.	11	0001
51.	11	0010
52.	11	0011
53.	11	0100
54.	11	0101
55.	11	0110
56.	11	0111
57.	11	1000
58.	11	1001
59.	11	1010
60.	11	1011
61.	11	1100
62.	11	1101
63.	11	1110
64.	11	1111

Після того, як тестові вектори отримані, необхідно створити згадану вище модель мультиплексора. Модель реалізована у файлі mux_tb.vht на мові VHDL. Цей файл є test bench'єм (Далі по тексту ТВ), тобто тестовою середою для функціонального блоку. Модель мультиплексора на мові VHDL представлена на рисунку 3.4.

MUX_model_p: process(i_control,i_data)

```
begin
if (i_control = "00") then
        o_data_exp <= i_data(0);
elsif (i_control = "01") then
        o_data_exp <= i_data(1);
elsif (i_control = "10") then
        o_data_exp <= i_data(2);
else
        o_data_exp <= i_data(3);
end if;
end process MUX_model_p;
```

Рисунок 3.4 – Модель мультиплексора на мові VHDL

Після створення моделі необхідно створити процеси, які будуть передавати дані в модель, та на вхідні порти, та процес який буде генерувати сигнал тактової частоти для синхронізації передачі даних (рис. 3.5).

```
clk_p: process
begin
clk <= '0';
wait for 2 ns;
clk <= '1';
wait for 2 ns;
end process clk_p;

control_p: process
begin
for i in 0 to 15 loop
wait until rising_edge(clk);
end loop;
i_control <= std_logic_vector(unsigned(i_control)+1);
end process control_p;

data_p: process
begin
wait until rising_edge(clk);
i_data <=  std_logic_vector(unsigned(i_data)+1);
end process data_p;
```

Рисунок 3.5 – Процеси передачі даних на мові VHDL

Це все були підготовчі кроки. Головним кроком являється перевірка ФБ на відповідність вимогам. Так, як є створена модель мультиплексору, то для

тестування ФБ, достатньо порівняти вихідні дані моделі з вихідними даними ФБ. Та у разі виявлення помилки, тобто якщо дані не будуть рівними, повідомити про неї, та відобразити вхідні дані при яких виникла помилка. Для цього порівняння було створено процес, в якому за повідомлення про помилки відповідає функція event_error, яка визивається з розробленої власноруч бібліотеки monitor_pkg.vhd (опис цієї бібліотеки приведено нижче). Процес перевірки даних є однією з двох частин, монітору тестування ФБ (рис. 3.6).

```
check_MUX_req_p: process
begin
wait until i_control'event or i_data'event;
if (o_data /= o_data_exp) then
        event_error(i_data,info_of_error(1));
        event_error(i_data,info_of_error(2));
end if;
end process check_MUX_req_p;
```

Рисунок 3.6 – Перевірка ФБ на відповідність вимогам

Останнім кроком при тестуванні ФБ є вивід результатів тестування у формі звіту. В залежності від результату порівняння даних, тест може бути позитивним (Pass), чи негативним (Fail). Для отримання звіту створено процес, що визиває функцію part_message з бібліотеки monitor_pkg.vhd, яка відповідає за формування сумарного звіту тестування, котрий по вибору можна сформувати в файл report.txt або вивести в консоль середи моделювання (рис. 3.7).

```
report_p:
process
begin
wait for 255 ns;
part_message(1,FILE_NAME,REQUIREMENT_LIST,info_of_error);
end process report_p;
```

Рисунок 3.7 – Формування сумарного звіту тестування

Також особливість при написанні ТВ є те, що на відміну від реалізації ФБ не треба описувати сутність ТВ, треба лише описати архітектуру ТВ, тому що ми оперуємо лише сигналами, а для того, щоб прив’язати ці сигнали до портів ФБ,

необхідно описати його, як компонент, та описати карту портів, в якій зазначено зв'язок сигналів ТВ і портів ФБ (рис. 3.8).

```
entity mux_vhd_tst is
end mux_vhd_tst;
architecture arch of mux_vhd_tst is
        signal i_data       : std_logic_vector (3 downto 0) := (others => '0');
        signal i_control    : std_logic_vector (1 downto 0) := (others => '0');
        signal o_data               : std_logic := '0';
        signal clk                  : std_logic := '0';
        signal info_of_error : error_list_on_requiment(2 downto 1) := (others =>
((others=> (others => '0')),(others => 0 ps),0,0));
        signal o_data_exp           : std_logic := '0';
        file out_file : text open WRITE_MODE is "report.txt";
        constant FILE_NAME : file_verified :=
    (resize("mux_tb.vht_(V01R00)",LENGTH_FILEN),resize("mux.vhd_(V01R00)"
,LENGTH_FILEN));
        constant REQUIREMENT_LIST : requirement_statement :=
                (
                        1 => (resize("MUX.01",LENGTH_ID), resize("Should form
output data according to control signal",LENGTH_RS)),
                        2 => (resize("MUX.02",LENGTH_ID), resize("Control signal
should choose number of input data bit, that will be transmitted to
output",LENGTH_RS))
                );
component mux
 port (
        i_data              : in std_logic_vector (3 downto 0);
        i_control    : in std_logic_vector (1 downto 0);
        o_data              : out std_logic
);
end component;
```

Рисунок 3.8 – Архітектура ТВ

Окрім сигналів, які зв’язані з портам ФБ, ТВ містить сигнали тактової частоти, інформування про помилку, очікуваних вихідних даних моделі мультиплексора, опис звітного файлу та константи, які відповідають за формування звіту.

При реалізації ТВ, окрім бібліотеки std_logic_1164, яка була використана при реалізації ФБ, також були використані бібліотеки numeric_std (в ній описані

функції роботи з різними типами, логічні, арифметичні операції), textio (необхідна для формування звітного текстового файлу) та власноруч створена бібліотека monitor_pkg (рис. 3.9).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;
library work;
use work.monitor_pkg.all;
```

Рисунок 3.9 – Використані бібліотеки при реалізації ТВ

Опис бібліотеки monitor pkg.

Бібліотека monitor_pkg використовується для оптимізації отримання результатів при проведенні функціонального тестування ФБ. Основними функціями цієї бібліотеки є підрахунок помилок при тестуванні, сортування помилок по кожній вимозі до ФБ, вивід вхідних даних, при яких виявлена помилка, та формування сумарного звіту, та його вивід в консоль середи моделювання та у звітній текстовий файл. Використання цієї бібліотеки скорочує затрати часу на аналіз помилок, затрати на оформлення звітних документівта дає кращу від стандартної візуалізацію результатів тестування. Також в ній описані додаткові функції, які супроводжують роботу основних функцій. При створенні бібліотеки були використані вже знайомі стандартні бібліотеки std_logic_1164, numeric_std, textio (рис. 3.10).

```
-- Підключення необхідних бібліотек
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library std;
use std.textio.all;
package monitor_pkg is
-- Опис необхідних типів та констант
   type counter_vector is array (50 downto 1) of integer;
constant ARRAY_SIZE : integer := 50;
constant LENGTH_ID : integer := 20;
```

```
        constant LENGTH_RS : integer := 200;
        constant LENGTH_FILEN : integer := 60;
        type err_time is array (ARRAY_SIZE downto 0) of time;
        type err_vector_64 is array (ARRAY_SIZE downto 0) of std_logic_vector (63
downto 0);
        type req is record
                iD : string(1 to LENGTH_ID);
                requirement_statement : string(1 to LENGTH_RS);
        end record;
        type file_verified is record
                file_vht : string(1 to LENGTH_FILEN);
                file_vhd : string(1 to LENGTH_FILEN);
        end record;
        type requirement_statement is array (natural range <>) of req;
        type err_vector_list is record
                vector_of_err : err_vector_64 ;
                time_of_err   : err_time ;
                count_of_err  : integer ;
                size          : integer ;
        end record;
        type error_list_on_requiment is array (natural range <>) of err_vector_list;
        --функція знаходження максимального значення
        function max_val(l,r : integer) return integer;
        --функція зміни рядку до необхідного розміру
        function resize(s : string; NewSize : natural) return string;

        procedure write_message(
                str     :       string
        );
        --функція виводу повідомлення в консоль та в файл
        procedure write_message(
                enable : integer;
                str     :       string
        );
        --функція додавання помилки в масив
        procedure event_error
        (
                data            : in std_logic_vector;
                signal error_items   : inout err_vector_list
        );
        --функція виводу версифікаційного повідомлення
        procedure FBTB_verification_message
        procedure part_message
        (
```

```
        enable                  : integer;
        FILE_NAME               : file_verified;
        REQUIREMENT_LIST   : requirement_statement;
        signal info_of_error : in error_list_on_requiment
    );
    --функція виводу інформації про помилку
    procedure info_error
    (
        enable         : integer;
        error_statement : err_vector_list
    );
    --функція виводу звітного повідомлення
    procedure end_message
    (
        enable : integer;
        requirement_id : string;
        count : integer
    );
    end package monitor_pkg;
    -- реалізація вищеперерахованих функцій
    package body monitor_pkg is

    function max_val(l, r : integer) return integer is
    begin
        if (l > r) then
            return(l);
        else
            return(r);
        end if;
    end max_val;
        function resize(s : string; NewSize : natural) return string is
        variable res : string (1 to NewSize);
        constant nulStr : string(1 to max_val(s'length-NewSize,1)) := (others =>
nul);
    begin
        if(s'length > NewSize)then
            if(s(s'right-nulStr'length+1 to s'right) = nulStr)then
                res := s(1 to NewSize);
                return(res);
            else
                assert false
                report "can not be performed RESIZE"
                severity error;
            end if;
```

```
		else
			res(1 to s'length) := s;
			if (NewSize /= s'length) then
				res(s'length+1 to NewSize) := (others => nul);
			end if;
			return(res);
		end if;
	end resize;
	--
	procedure write_message(
		str	:	string
	) is
		variable buf : line;
	begin
		write(buf, str);
		writeline(output, buf);
	end write_message;

	procedure write_message(
		enable : integer;
		str	:	string
	) is
		variable buf : line;
		file out_file : text open WRITE_MODE is "report.txt";
	begin
		if (enable = 1) then
			write(buf, str);
			tee(out_file, buf);
		end if;
	end write_message;

	procedure info_error(
		enable : integer;
		error_statement : err_vector_list
	) is
		constant NO_VECTOR : std_logic_vector(63 downto 0) := (others => 'U');
	begin
		for i in 0 to error_statement.count_of_err-1 loop
			write_message(1,"    ERROR #" & to_string(i + 1));
			write_message(1,"                    Time error @" &
to_string(error_statement.time_of_err(i)));
			if not(error_statement.vector_of_err(i) = NO_VECTOR) then
				write_message("                    Vector error " &
to_string(error_statement.vector_of_err(i)(error_statement.size-1 downto 0)));
```

```
                end if;
            end loop;
            write_message(1,"----------------------------------------------------------------
");
        end info_error;
            procedure event_error(
            data : in std_logic_vector;
            signal error_items : inout err_vector_list
        ) is
        begin
            error_items.count_of_err <= error_items.count_of_err + 1;
            error_items.time_of_err(error_items.count_of_err) <= now;
            error_items.vector_of_err(error_items.count_of_err)(data'length-1   downto
0) <= data;
            error_items.size <= data'length ;
        end event_error;

        procedure FBTB_verification_message(
            enable : integer;
            fbtb_name_vr : string
        ) is
        begin
            write_message(1,"----------------------MONITOR  MESSAGE---------------
-------------");
            write_message(1,"FBTB " & fbtb_name_vr & " verified");
            write_message(1,"----------------------------------------------------------------
");
        end procedure;
            procedure end_message(
            enable : integer;
            requirement_id : string;
            count : integer
        ) is
        begin
            write_message(1,"Requirement   "   &   requirement_id   &   ":   "   &
to_string(count) & " errors found");
        end end_message;

        procedure part_message(
            enable : integer;
            FILE_NAME : file_verified;
            REQUIREMENT_LIST : requirement_statement;
            signal info_of_error : in error_list_on_requiment
        ) is
```

```
        variable all_errors : integer := 0;
    begin
        FBTB_verification_message(1,FILE_NAME.file_vht);
        for i in 1 to REQUIREMENT_LIST'right loop
            if info_of_error(i).count_of_err = 0 then
                write_message(1,"-----------------------MONITOR
MESSAGE----------------------------");

    write_message(1,REQUIREMENT_LIST(i).requirement_statement);
                write_message(1,REQUIREMENT_LIST(i).iD & " Pass");
            else
                write_message(1,"-----------------------MONITOR
MESSAGE----------------------------");

    write_message(1,REQUIREMENT_LIST(i).requirement_statement);
                write_message(1,REQUIREMENT_LIST(i).iD & " Fail");
                write_message(1,"Error info:");
                info_error(1,info_of_error(i));
            end if;
        end loop;
        write_message(1,"-----------------------MONITOR  MESSAGE----------------
--------------");
        write_message(1,FILE_NAME.file_vhd & " verified");
        for i in 1 to REQUIREMENT_LIST'right loop

    end_message(1,REQUIREMENT_LIST(i).iD,info_of_error(i).count_of_err);
            all_errors := all_errors + info_of_error(i).count_of_err;
        end loop;
        write_message(1,"The total number of errors : " & to_string(all_errors));
        write_message(1,"----------------------------------------------------------------
");
    end part_message;

    end package body monitor_pkg;
```

Рисунок 3.10 – Реалізація бібліотеки monitor_pkg

Після створення ТВ, необхідно створити файл сценарію, в якому будуть задані параметри симуляції (mux_tb.do). В цьому файлі описані файли, які треба скомпілювати, тобто необхідні для симуляції. Назва архітектури ТВ, яку необхідно викликати для симуляції, сигнали які необхідно відтворити при

симуляції, час симуляції, та папка, в яку будуть скомпільовані необхідні для симуляції файли (рис. 3.11).

```
vlib work
vmap work work

vcom -2008 -work work {monitor_pkg.vhd}
vcom -2008 -work work {mux.vhd}
vcom -2008 -work work {mux_tb.vht}

vsim mux_vhd_tst

add wave *

run 255 ns
```

Рисунок 3.11 – Файл сценарію симуляції

В даному розділі було розроблено ФБ мультиплексор, складені вимоги до нього. Для виконання тестування були обрані тестові вектори та створено test bench, на мові VHDL. Для запуску тестування був написаний сценарій симуляції. Також була описана бібліотека, яка була розроблена для оптимізації отримання результатів при функціональному тестуванні ФБ. Тобто є все необхідне, щоб перейти до наступного етапу: запустити тест, та провести симуляції роботи ФБ, та тестів для нього в середовищі моделювання.

3.3 Опис середовища моделювання

Для моделювання роботи ФБ було обрано середовище Modelsim, версія PE 10.0c. Modelsim це середовище симуляції апаратних пристроїв та налагодження їх програмного забезпечення. Розроблена компанією Mentor Graphics,що працює в області автоматизації проектування електроніки (EDA) для електротехніки і електроніки.

Для того, щоб розпочати симуляцію, спочатку необхідно обрати папку, в якій розташовані файли, описані у третьому розділі.

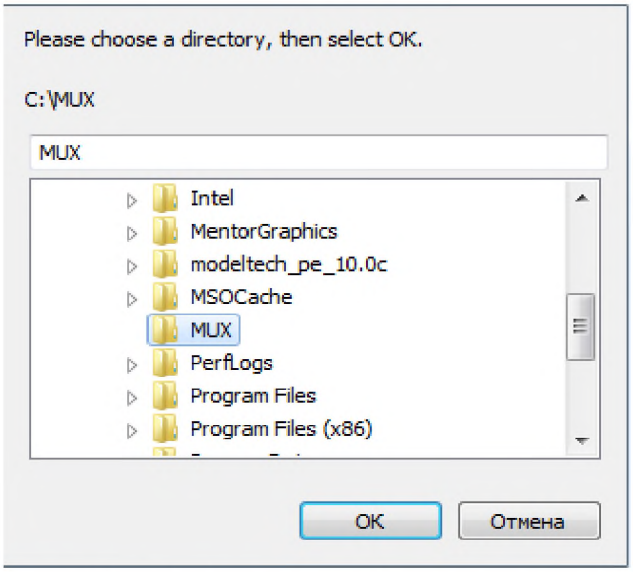

Рисунок 3.12 – Вибір папки зі створеними файлами

Для цього необхідно виконати наступну послідовність File -> Change Directory…->MUX->OK (рис. 3.12).

Після цього, для запуску симуляції необхідно лише виконати файл сценарію тестування, який був описаний у третьому розділі (mux_tb.do) і він виконає всі указані команди автоматично, тобто відкомпілює задані файли, додасть для відображення необхідні сигнали та запустить симуляцію, все це можливо виконати в ручному режимі, але це дуже збільшує час тестування, та підвищує складність, особливо якщо необхідно проводити повторне тестування. Для виконання файлу mux_tb.do необхідно виконати наступні дії: Tools->Tcl->Execute macro…->mux_tb.do-> Open (рис. 3.13).

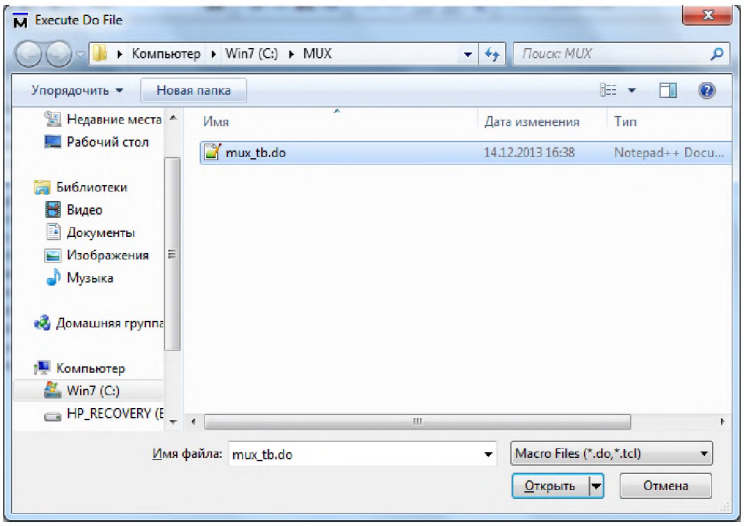

Рисунок 3.13 – Виконання файлу сценарію тестування

Після виконання цього файлу, тестування виконається автоматично, та завдяки створеній бібліотеці monitor_pkg.vhd, автоматично відобразить результати тестування у консолі середовища моделювання та запише їх у файл report.txt, також буде відображено часову діаграму, на якій відображена зміна сигналів, на протязі симуляції, це необхідно для ситуацій, коли ФБ функціональний блок функціонує не вірно, та при тестуванні виявлено помилку, тобто щоб знайти її та полагодити (рис. 3.14, рис. 3.15, рис. 3.16, рис. 3.17).

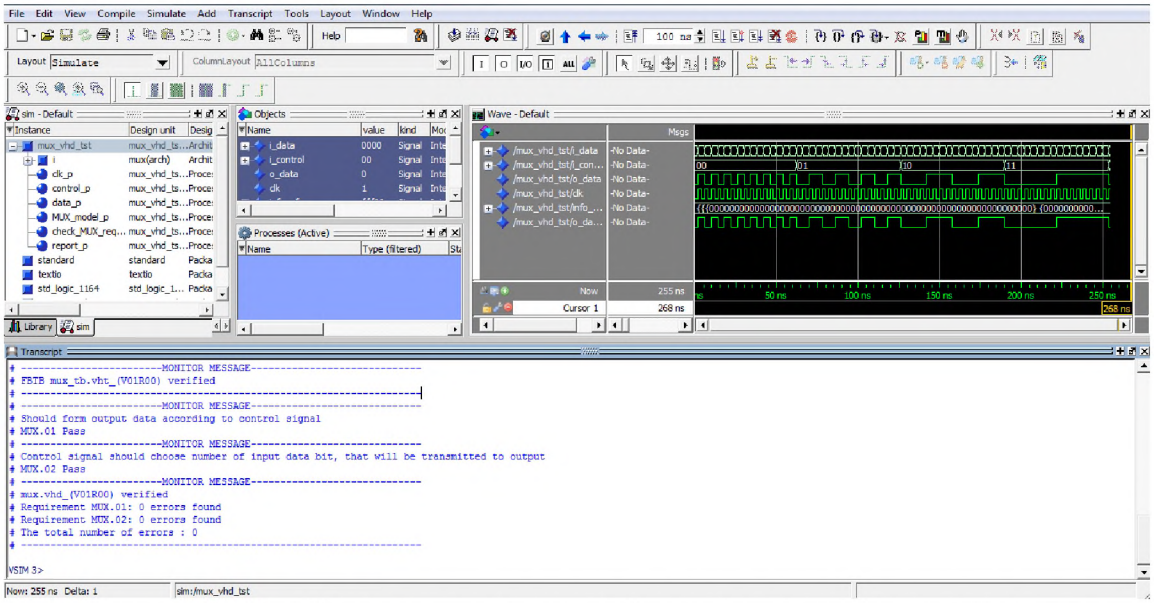

Рисунок 3.14 – Результати симуляції у середовищі моделювання

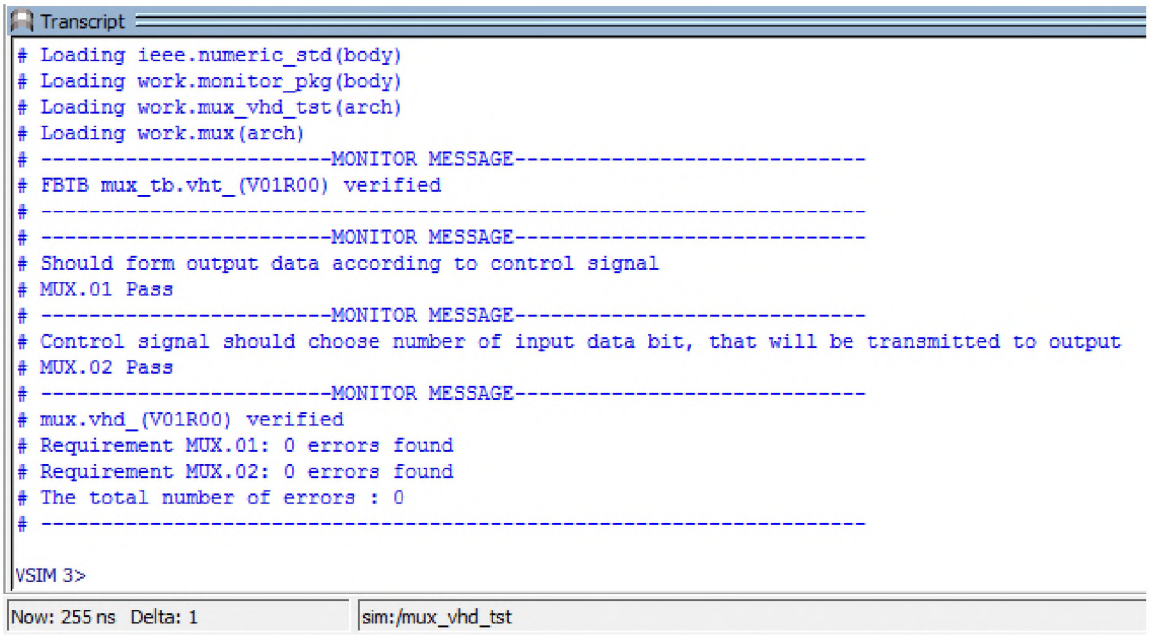

Рисунок 3.15 – Результати симуляції у консолі середовищі моделювання

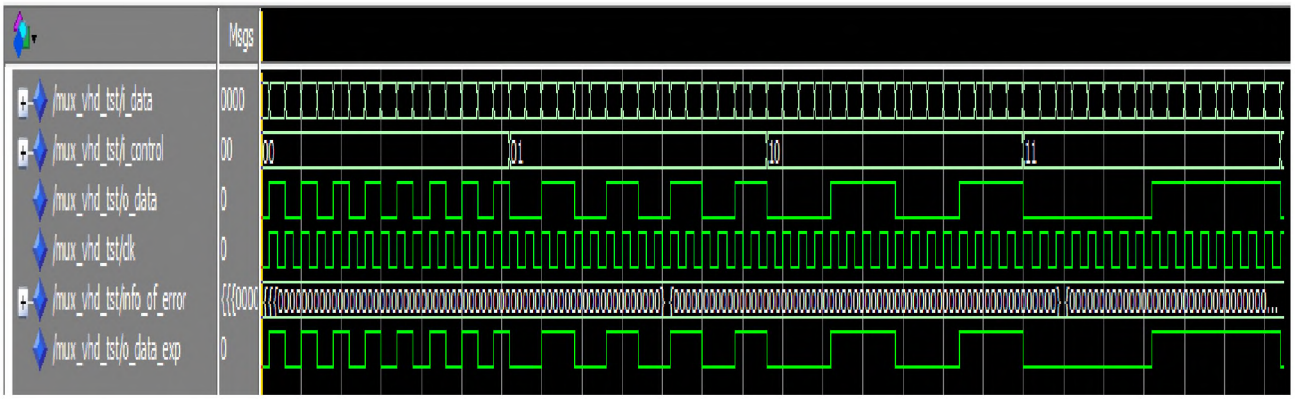

Рисунок 3.16 – Результати симуляції на часовій діаграмі середовища моделювання

----------MONITOR MESSAGE----------

FBTB mux_tb.vht_(V01R00) verified

----------MONITOR MESSAGE----------

Should form output data according to control signal

MUX.01 Pass

----------MONITOR MESSAGE----------

Control signal should choose number of input data bit, that will be transmitted to output

MUX.02 Pass

----------MONITOR MESSAGE----------

mux.vhd_(V01R00) verified

Requirement MUX.01 : 0 errors found

Requirement MUX.02 : 0 errors found

The total number of errors : 0

Рисунок 3.17 – Результати симуляції у текстовому файлі

Після того, як симуляція виконана, необхідно оцінити, покриття вимог та кодове покриття.

3.4 Оцінка результатів виконання процедури функціонального тестування

Щоб оцінити покриття вимог, необхідно кількість вимог, яким за результатами тестування ФБ відповідає, поділити на загальну кількість вимог. Кількість вимог, яким відповідає ФБ дорівнює двом, загальна кількість вимог також два, тобто ми маємо 100% покриття вимог.

Також важливим критерієм оцінки функціонального тестування є кодове покриття. Але для його підрахунку необхідно використовувати середовище моделювання. Modelsim підтримує оцінку кодового покриття за багатьма параметрами, але згідно стандарту IEC 61508-3 2010 Table B.2, для SIL 3 необхідними являються наступні параметри:

1. Покриття операторів;
2. Покриття шляхів.

Для того, щоб середовище моделювання підрахувала ці параметри підчас симуляції, необхідно у файлі сценарію тестування mux_tb.do задати компіляцію ФБ з параметром +cover=sb (sb – це statements (оператори), branches (шляхи)), та задати симуляцію з параметром –coverage. Після цього виконати файл сценарію тестування та зберегти результати кодового покриття. Для того, щоб зробити це, необхідно виконати наступні дії: Tools->Coverage Save…, потім встановити параметри, як вказано на рисунку 3.18. та натиснути OK.

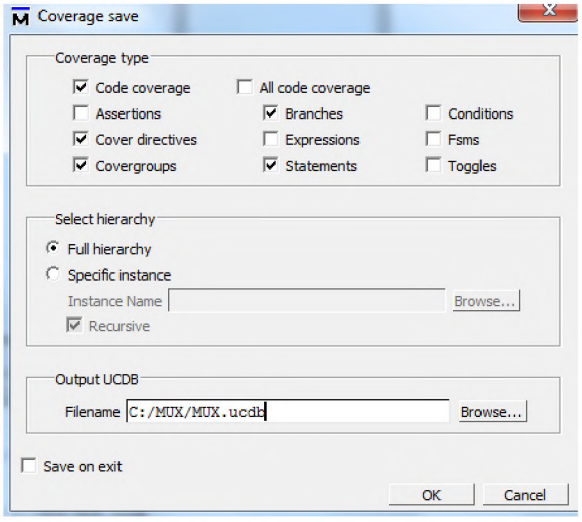

Рисунок 3.18 – Параметри зберігання результатів кодового покриття

Після цього, щоб побачити результати, необхідно згенерувати звіт, найбільш зручним для перегляду, є звіт в форматі HTML, для його генерації необхідно виконати наступні дії: View->Verification Manager->Browser->Add file->MUX.ucdb->HTML Report…->OK. Після цього отримаємо звіт, в якому є сумарна та детальна інформація про кодове покриття. (рис. 3.19).

Coverage Type	Bins	Hits	Misses	Coverage (%)
Statement	5	5	0	100.00%
Branch	4	4	0	100.00%

Рисунок 3.19 – Кодове покриття

100% кодового покриття по обом параметрам свідчить про те, що ТВ, цілком покриває тестами ФБ. Тобто розроблений ТВ є ефективним.

Підрахуємо затрачений в цілому час на проведені процеси, а саме розробку та функціональне тестування ФБ.

Таблиця 3.4 – Затрачений час

Вид робіт	Затрати часу (у відсотках)
Розробка	
Складання(інтерпретація) вимог	~5%
Розробка ФБ згідно вимог	~25%
Функціональне тестування	
Аналіз вимог	~2%
Вибір тестових векторів	~15%
Розробка ТВ	~50%
Симуляція в середовищі моделювання та аналіз результатів	~3%

Чи малозначущим є оцінка часу затраченого на створення ТВ. В цілому затрати на тестування продукту займають 70% часу, а розробка всього 30%, це не означає, що розробляти продукт це легко. Але зазвичай інженер, який розроблює продукт, не проводить його тестування. Так, як створений власноруч продукт важно якісно оцінити, тому цю роботу виконують сторонні, незалежні від розробника, інженери. Саме тому тестування займає більше часу, так як розібратись в чужому продукті, хай і з повним пакетом документації на нього важче, ніж у власноруч створеному. Тому оцінимо час на тестування відносно

часу на розробку, як 2 до 1, тобто тестування це 2/3, розробка – 1/3. Але 2/3 це не є час затрачений на функціональне тестування, так як воно не є єдиним видом тестування продукту. З цієї 2/3 на функціональне тестування, з урахуванням оформлення звітів, затрачається приблизно 20% часу, тому можна сказати, що функціональне тестування, займає 2/15 проектного часу, що є на справді дуже багато.

Дана модель тестової системи, дозволяє скоротити затрати часу в цілому приблизно на 15-20%, тим що менше часу затрачується на аналіз результатів та оформлення звіту.

Саме тому дуже важливо створення максимально оптимізованої моделі тестової системи, яка дозволить знизити затрати, на функціональне тестування, та усунення знайдених помилок якомога швидше.

3.3 Техніко-економічне обґрунтування розробки

Витрати на розроблення процедури функційного тестування здійснюються за формулою:

$$Кп = Кпр + Кпк + Кт , \qquad (3.1)$$

де Кпр – витрати на процедури;

Кпк – витрати на створення процедури, з яких вона складається;

Кт – витрати на тестування процедури.

Загальний розрахунок на розробку процедури можливо виконати за формулою

$$Кп = Фз/п\ [(1+βд)(1+βс)+βн+βпр]+t_{ПЕОМ}, \qquad (3.2)$$

де Фз/п – фонд основної заробітної платні розробників та інших виконавців;

βд – коефіцієнт додаткової заробітної платні, можливо прийняти 0,10...0,15;

βс – коефіцієнт відрахування, який дорівнює 0,26;

βн – коефіцієнт накладних витрат організації 0,6...0,8;

βпр – коефіцієнт інших витрат 0,1...0,2;

$t_{ПЕОМ}$ –машинний час, затрачений час на тестування та відлагодження.

Відповідно до цього підходу обчислено, що загальний час роботи одного програміста-тестувальника над процедурою становив 156 годин. За умови оплати праці 20 грн на годину, загальна суму розробки склала 3120 грн.

Висновки до розділу 3

В третьому розділі розроблено тест-бенчі на мові VHDL:

1. Розроблені вимоги до ФБ;
2. Розроблений програмний код на мові VHDL.

Після цього проведено ФТ для розробленого блоку:

1. Проаналізовані вимоги до ФБ;
2. Створені ТС (обрані тестові вектори);
3. Розроблені ТВ для проведення ФТ;
4. Проведено ФТ для ФБ згідно покроковій методиці, описаній раніше.

Проведений опис середи моделювання та показано результати проведення ФТ.

Результати були проаналізовані та були оформлені звітні документи, такі як план та специфікація ФТ та звіт з ФТ, представлені в додатку А та Б відповідно.

Проведена оцінка результатів виконання процедури ФТ за тестовим покриттям, а саме за покриттям вимог та покриттям коду.

За допомогою розроблених ТВ для ФБ отримано 100% покриття вимог, та 100% покриття коду, що свідчить про 100% тестове покриття та про ефективність розробленого ТВ.

Оцінено часові затрати на тестування та розробку ФБ. З результатів оцінки видно, що тестування займає 2/3 часу, що свідчить про актуальність розробки моделі тестової системи яка б зменшила затрачуваний час на тестування, та процедури ФТ в цілому.

В ході роботи отримано нову модель тестової системи, яка дозволила автоматизувати процес ФТ та підвищити швидкість проведення ФТ, завдяки

скороченню затрачуваного часу приблизно на 15-20%, та підвищити точність ФТ, завдяки автоматизації тестування.

Розроблено бібліотекуmonitor_pkg.vhd на мові VHDL, яка надає корисні функції по автоматизації процесу ФТ, а саме аналізу отриманих результатів та генерації звіту з ФТ.

Розроблено процедуру проведення ФТ для коду на мові VHDL, яка дозволяє проводити швидке та точне ФТ.

ВИСНОВКИ

На основі проведеної розробки процедури ФТ для ПЗ на мові VHDL, можна зробити наступні висновки.

Після проведення аналізу нинішнього росту складності НВІС, було підтверджено актуальність використання ПЛІС. Функції ПЛІС дозволяють реалізовувати їх можливості у чи малій низці сфер, а саме у цифровій обробці сигналів, у ролі вбудованих мікроконтролерів, для фізичного рівня передачі даних, у системах з перебудовуємою архітектурою та автоматизованих системах управління. Через збільшення використання ПЛІС, росте й актуальність використання таких мов програмування як VHDL. Так як для ПЛІС на мові VHDL, створюється ПЗ, то були проаналізовані моделі життєвого циклу ПЗ та обрано V-модель, так як V-модель вимагає чітко структурований процес розробки та модульну структуру програмного забезпечення, для запобігання та контролю систематичних помилок, що є дуже актуальним, особливо при використані ПЛІС в автоматизованих системах управління. Згідно цієї моделі розглянуто місце ФТ в життєвому циклі всього продукту, та визначено задачі ФТ, так як і призначення процедури ФТ та її зміст.

Розроблено процедуру ФТ, в якій описано створення нової моделі тестової системи. Приведено її основні функції. Розроблена модель, є більш оптимізована та дозволяє швидко аналізувати та оформлювати результати, але є вузькоспеціалізованою під тестування функціональних блоків, розроблених за стандартом IEC 61508-2010 для SIL 3, на мові VHDL. Після аналізу та опису моделі тестової системи, описано процедуру проведення функціонального тестування. Вказано ціль та об'єкт процедури ФТ. Описано розробку плану та специфікації ФТ. Вказано основні елементи цих документів. В межах даної процедури план та специфікація об'єднані в єдиний документ, який подано в додатку А. Описано проведення функціонального тестування, а саме опис ІЗ необхідних для виконання ФТ. Описано створення ТВ для ФТ, структуру ТВ,

склад керівництва користувача ТВ. Описано підхід до вибору ТС. Описані можливі типи вхідних даних для ТС, а саме:

1. Граничні значення даних;
2. Відсутність даних;
3. Повторне введення даних;
4. Невірні дані;
5. Реініціалізація системи;
6. Класи еквівалентності.

Далі описано покрокову методику проведення ФТ та ІЗ які необхідні для виконання ФТ. Описано, як необхідно проводити аналіз результатів ФТ та оформлювати сумарний звіт з ФТ, який надано в додатку Б.

Після цього було розроблено ФБ на мові VHDL:

1. Розроблені вимоги до ФБ;
2. Розроблений програмний код на мові VHDL.

Після цього проведено ФТ для розробленого блоку:

1. Проаналізовані вимоги до ФБ;
2. Створені ТС (обрані тестові вектори);
3. Розроблені ТВ для проведення ФТ;
4. Проведено ФТ для ФБ згідно покроковій методиці, описаній раніше.

Проведений опис середи моделювання та показано результати проведення ФТ.

Результати були проаналізовані та були оформлені звітні документи, такі як план та специфікація ФТ та звіт з ФТ, представлені в додатку А та Б відповідно.

Проведена оцінка результатів виконання процедури ФТ за тестовим покриттям, а саме за покриттям вимог та покриттям коду.

За допомогою розроблених ТВ для ФБ отримано 100% покриття вимог, та 100% покриття коду, що свідчить про 100% тестове покриття та про ефективність розробленого ТВ.

Оцінено часові затрати на тестування та розробку ФБ. З результатів оцінки видно, що тестування займає 2/3 часу, що свідчить про актуальність розробки

моделі тестової системи яка б зменшила затрачуваний час на тестування, та процедури ФТ в цілому.

В ході роботи отримано нову модель тестової системи, яка дозволила автоматизувати процес ФТ та підвищити швидкість проведення ФТ, завдяки скороченню затрачуваного часу приблизно на 15-20%, та підвищити точність ФТ, завдяки автоматизації тестування.

Розроблено бібліотеку monitor_pkg.vhd на мові VHDL, яка надає корисні функції по автоматизації процесу ФТ, а саме аналізу отриманих результатів та генерації звіту з ФТ.

Розроблено процедуру проведення ФТ для коду на мові VHDL, яка дозволяє проводити швидке та точне ФТ.