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 та дослідження верифікаційної моделі конфігураційної пам'яті електронного проєкту ПЛІС»**

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
Чорний Б. В.
Керівник: Одарущенко О. М.
Рецензент: Яхін С.В

Полтава – 2023 року

ВСТУП

Актуальність теми дослідження. Сьогоднішнє життя неможливо уявити без обчислювальних пристроїв (ОП), прихованих в побутовій техніці, вимірювальних і медичних приладах, управлінні агрегатами транспортних засобів, телекомунікаціях, чіп-картах і т.п. У персональних комп'ютерах, яких зараз у світі сотні мільйонів штук, функції периферійних пристроїв виконують різні спеціалізовані ОП. Якщо подумки охопити всю електроніку, використовувану людиною, то в цілому, на один універсальний мікропроцесор припадають багато десятків спеціалізованих ОП. Число нових застосувань ОП стрімко зростає, а терміни їх розробки неухильно скорочуються.

Основна частина спеціалізованих ОП реалізована на мікроконтролерах. Але залишається велика кількість завдань, що вимагають екстремальних характеристик швидкодії, пропускної здатності, енергоспоживання, для реалізації яких необхідна розробка замовних мікросхем. Останнім часом ядро мікроконтролера і необхідні складні периферійні пристрої виконують у вигляді системи на кристалі (СНК). Зростаючий обсяг розробок спеціалізованих ОП і СНК вимагає освоєння нових технологій їх проєктування.

Проєктування ОП являє собою поетапне перетворення технічного опису ОП з уточненням і деталізацією опису на кожному новому етапі. Початковий опис спеціалізованого ОП включає в себе алгоритми його функціонування, порядок взаємодії з зовнішнім світом, а також технічні обмеження, такі як швидкодія, енергоспоживання, габарити і т.п.

Традиційне проєктування ОП засноване на складанні на кожному етапі різних графічних схем. Відповідно до назви етапу, розрізняють схеми алгоритмів, схеми електричні структурні, функціональні та принципові. Креслення схем пов'язано з високою трудомісткістю, витратами часу, а головне – з важкістю виявлення та виправлення помилок проєктування. Широке впровадження САПР електронних схем забезпечує зменшення трудовитрат, підвищення якості

проєктів, але не змінює докорінно характер проєктування – ручного складання схем.

Використання при розробці ОП таких мов проєктування, як VHDL і Verilog дозволяє відмовитися від складання схем на більшості етапів розробки ОП. Технологія проєктування ОП за допомогою VHDL передбачає опис вихідних алгоритмів функціонування апаратури та її інтерфейсу на мові VHDL і автоматичну трансляцію цього опису до рівня логічних схем і далі – до рівня масок мікросхем або прошивки програмованих логічних інтегральних схем (ПЛІС). При цьому схеми ОП в графічному вигляді практично не використовуються, а відсутність помилок в проєкті автоматично контролюється засобами верифікації. Як основний засіб верифікації, використовується моделювання функціонування ОП на VHDL-симуляторі на всіх етапах проєктування. Розробка сучасних ОП без використання мов VHDL і Verilog стала практично неможливою.

Мета дослідження – розробка моделі конфігураційної пам’яті для використання в модульному тестуванні.

Завданнями кваліфікаційної роботи є:

- аналіз процедури розроблення електронних проєктів ПЛІС;
- аналіз моделей конфігураційної пам’яті ПЛІС;
- розроблення моделі конфігураційної пам’яті ПЛІС;
- застосування розробленої моделі конфігураційної пам’яті ПЛІС.

Об’єкт дослідження – процес моделювання конфігураційної пам’яті на мові опису апаратних засобів.

Предмет дослідження – процедура тестування електронних проєктів на FPGA.

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 розроблена конфігураційної пам'яті відрізняється від відомих підтримуваними операціями, величиною затримок та логікою роботи функцій конфігурування.

Практична значущість роботи полягає в тому, що вдосконалена процедура тестування електронних проєктів на FPGA дозволяє покращити продуктивність роботи відповідальних за функційне тестування інженерів, зменшити затрачуваний на тестування час та підвищити точність їх результатів.

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, інноваційні 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, де: додаток А – програмний код розробленого додатку. Загальний обсяг роботи становить 62 сторінок; робота містить 32 рисунки; 8 таблиць; список використаних джерел, що включає 50 найменувань.

РОЗДІЛ 1
АНАЛІЗ ТЕХНОЛОГІЇ FPGA

1.1 Вибір FPGA як об'єкта дослідження

Основними перевагами FPGA є позбавлення від складного виробництва інтегральних схем та спрощення проєктування кристалу через відсутність проблем з топологією мікросхеми і взаємним впливом вузлів [20]. Недоліками можна назвати втрату продуктивності, понижену надійність, підвищену чутливість до перешкод, збільшене енергоспоживання кристалу та меншу віддачу від використання площі кристалу [20].

До основних напрямків використання FPGA можна віднести симуляцію та виготовлення мало то середньо серійних пристроїв.

Схеми будуються на базі логічних елементів. Кожен логічний елемент FPGA складається з двох основних частин: програмованого логічного елементу (так званого LUT – Lookup Table) та тригера на виході [3].

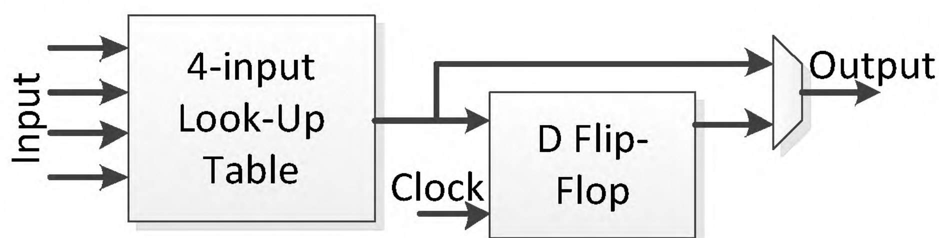

Рисунок. 1.1 – Логічний елемент FPGA

Один логічний елемент складається з декількох десятків логічних вентилів зроблених в базисі 2 І-НЕ або 2 АБО-НЕ. Програмований логічний елемент містить в собі таблицю істинності, що дозволяє реалізувати будь-яку комбінаційну схему. Як правило, такого типу елементи виконують у виді набору регістрів пам'яті. Конфігураційна пам'ять FPGA, містить в собі інформацію необхідну для з'єднання логічних елементів і таблиці істинності для LUT.

Логічні елементи, які складають FPGA, належать до рівня RTL. Їх з'єднання гарно описується структурними мовами VHDL та Verilog. Модель що використовується – модель дискретних подій [1-3].

З розвитком технологій виготовлення інтегральних схем проявляється тенденція до збільшення базових елементів FPGA. В більшості сучасних моделей додають пам'ять, елементи арифметико-логічних пристроїв, множники та цілі процесорні ядра.

FPGA на відміну від конкурента CPLD, містить більше вентилів та потребує менше логічних елементів для реалізації аналогічної схеми [9-11]. Більший логічний об'єм пояснюється будовою базового шару. Базовий шар CPLD містить збільшені логічні блоки елементарних вентилів (2ТА-НІ, 2АБО-НІ), а FPGA містить більш компактні логічні комірки на основі таблиць істинності LUT. У деяких FPGA (виробництва Xilinx та Altera) внутрішня пам'ять енергозалежна тому вони при увімкненні потребують конфігурації початковим завантажником – конфігураційною пам'яттю.

1.2 Моделі життєвого циклу

1.2.1 Каскадний життєвий цикл

Каскадний життєвий цикл (іноді званий водоспадним) заснований на поступовому збільшенні ступеня деталізації опису всієї системи, що розробляється . Кожне підвищення ступеня деталізації визначає перехід до наступного стану розробки (див. рис. 1.).

На першому етапі складається концептуальна структура системи, описуються загальні принципи її побудови, правила взаємодії з навколишнім світом - визначаються системні вимоги. На другому етапі складаються вимоги до програмного забезпечення - тут основна увага приділяється функціональності програмної компоненти, програмним інтерфейсам. Звичайно, всі програмні комплекси виконуються на який-небудь апаратній платформі. Якщо в ході проекту потрібна також розробка апаратної компоненти, паралельно з вимогами

до програмного забезпечення йде підготовка вимог до програмного забезпечення. На третьому етапі на основі вимог до програмного забезпечення складається детальна специфікація архітектури системи - описуються розбиття системи по конкретним модулям, інтерфейси між ними, заголовки окремих функцій і т.п. На четвертому етапі пишеться програмний код, відповідний детальної специфікації,

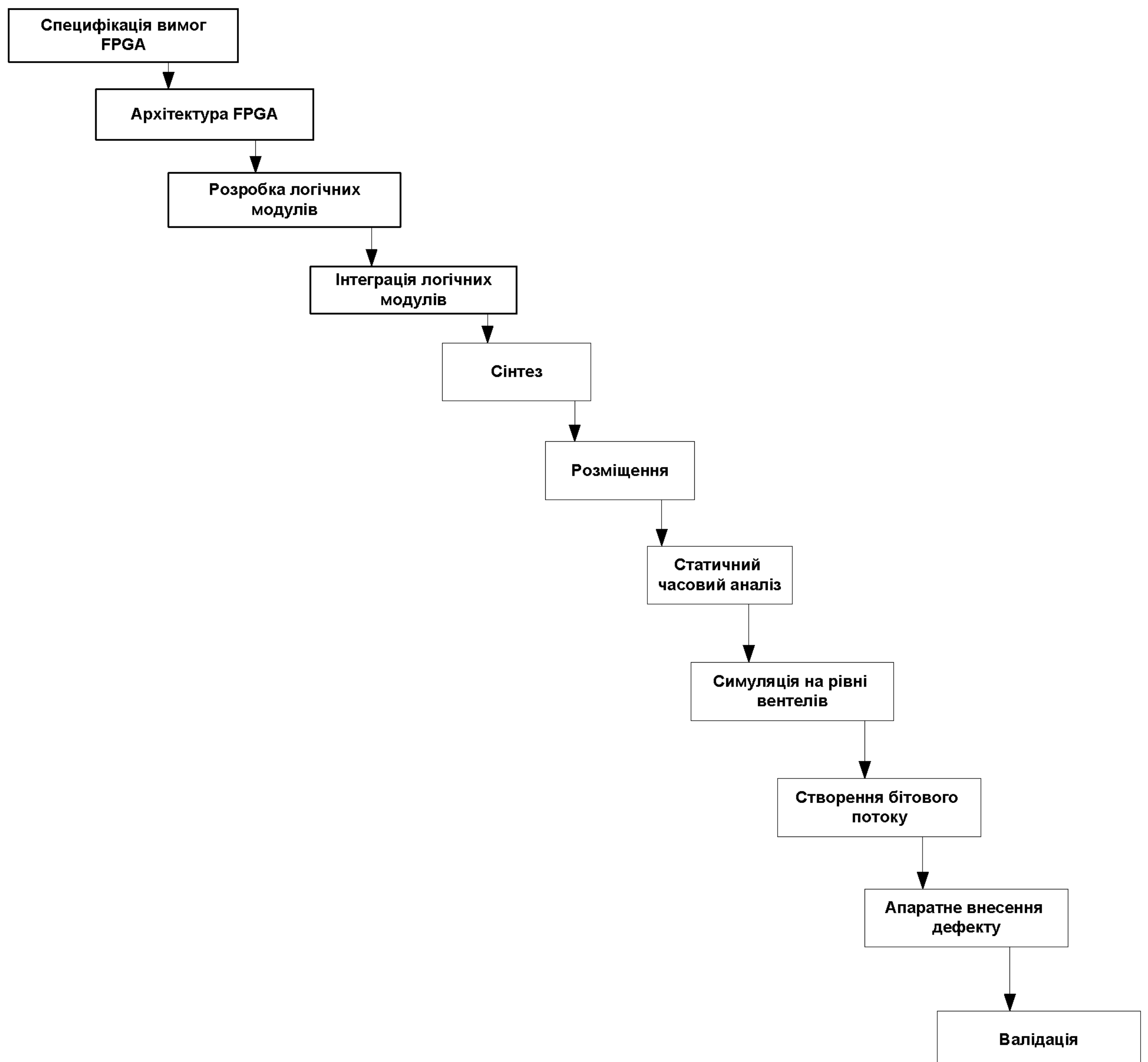

Рисунок 1.2.1. Каскадна модель життєвого циклу

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. Природно, що в разі досить великих систем такий підхід себе не виправдовує. Робота на кожному етапі займає значний час, а внесення змін до первинні документи або неможливо, або викликає лавиноподібне 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:

а) у неправильному тесті;

б) в помилки кодування (реалізації);

в) в невірній архітектурі системи;

г) 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 [16].

1.2.2 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 (див. рис. 1.2.2.1).

Таким чином, в кінці кожного етапу життєвого циклу розробки, а часто і в процесі виконання етапу, здійснюється перевірка взаємної коректності вимог різних рівнів. 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 [17].

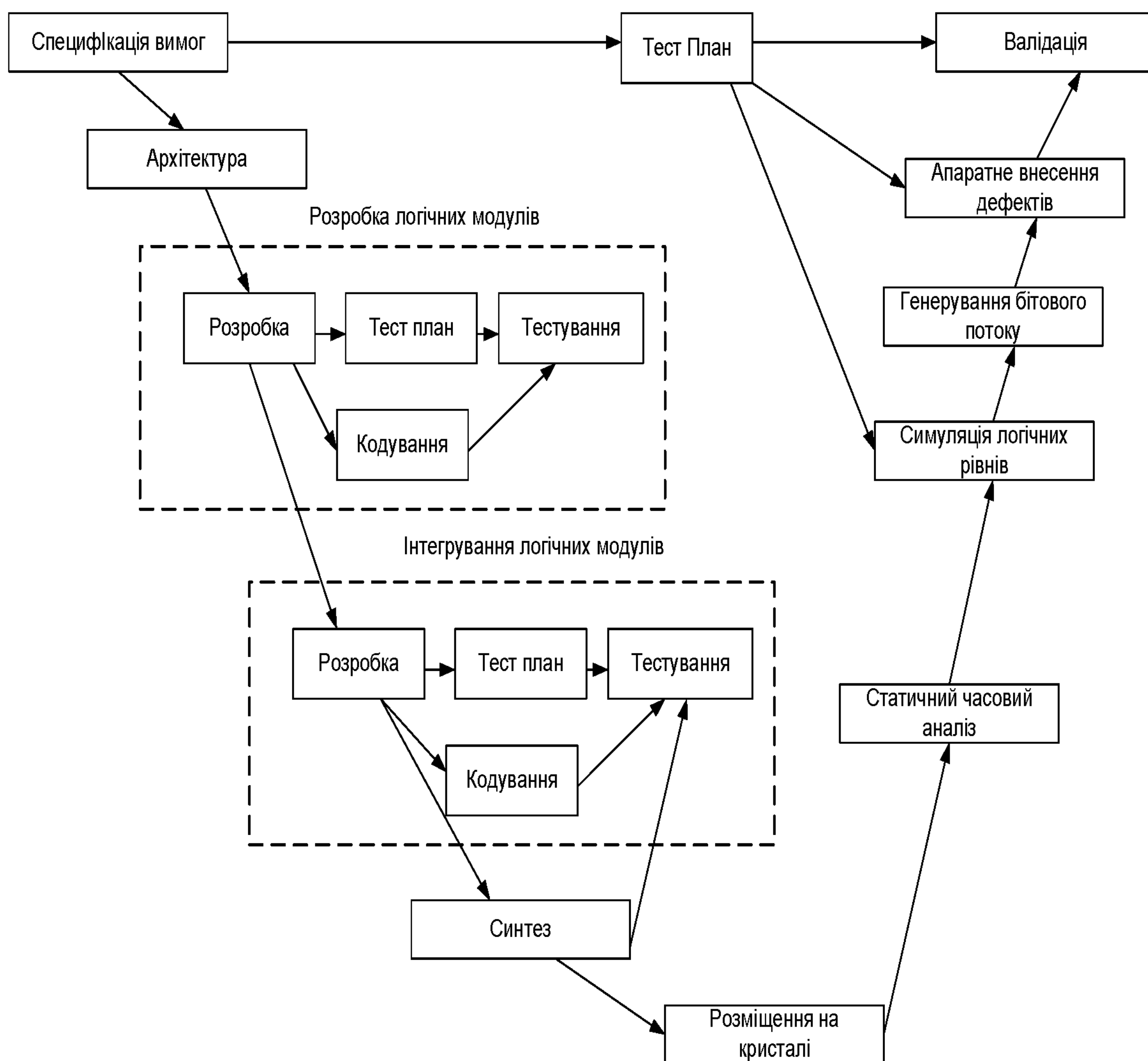

Рисунок. 1.2.2.1 V-подібний життєвий цикл

1.2.3 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. Природно, 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 (див. рис. 1.2.3.1).

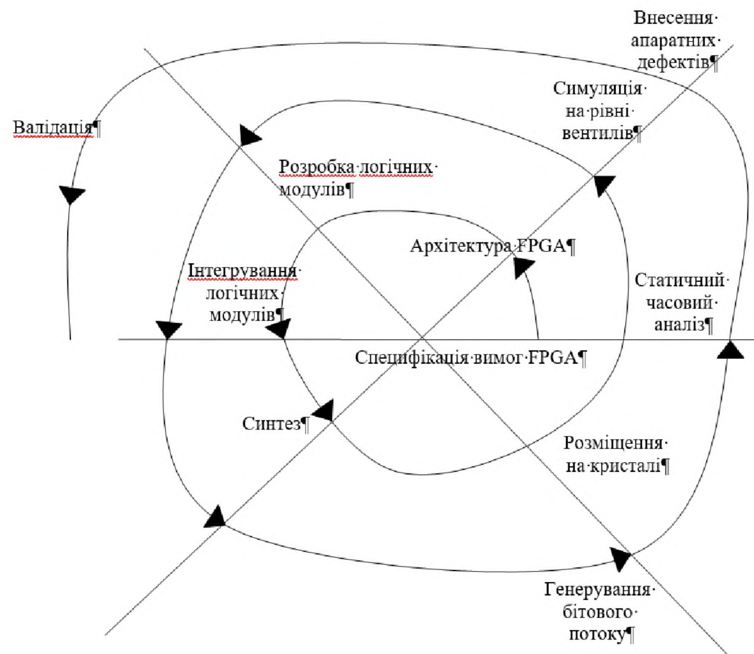

Рисунок. 1.2.3. Спіральний життєвий цикл

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

1.2.4 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 [25].

1.2.5 Порівняння різних типів життєвого циклу і допоміжні процеси

Особливості розглянутих вище типів життєвого циклу зведені в таблицю (див. табл. 1.2.5.1). 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.

У наведеному вище описі різних моделей життєвого циклу по суті порушувалося 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

Таблиця 1.2.5 –Порівняння життєвих циклів

Тип життєвого циклу	Довжина циклу	Верифікація та внесення змін	Інтеграція окремих компонентів 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 міняються в межах версії.
XP	Розробка однієї історії. Короткий.	Наприкінці розробки кожної історії. Дуже часто	Часто змінювані інтерфейси.

Процес верифікації – процес, спрямований на перевірку коректності розроблюваної системи та визначення ступеня її відповідності вимогам замовника. Докладного розгляду цього процесу і присвячений даний розділ.

Процес управління розробкою – окрема тема, 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 для менеджерів 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. Загалом, допоміжні процеси можуть існувати протягом всього життєвого циклу розробки, а можуть бути своєрідними стиками ланок між процесом розробки та процесом експлуатації [21].

Особлива увага приділяється найбільш значущим підтримуючим процесам - процесу управління конфігураціями і процесу забезпечення гарантії якості. Основна мета процесу управління конфігураціями – забезпечення цілісності всіх даних, що виникають у процесі колективної розробки. Під цілісністю розуміється, перш за все, ідентифікованість, доступність цих даних в будь-який момент часу і недопущення несанкціонованих змін. Важливим аспектом при цьому стає процес управління змінами даних, тобто планування та затвердження будь-яких змін у проектну документацію або програмний код, а також визначення області впливу цих змін. Процес гарантії якості забезпечує проведення перевірок, що гарантують, що процес розробки задовольняє набору певних вимог (стандартів), необхідних для випуску якісної продукції [28]. Фактично він перевіряє, що всі передбачені

стандартами розробки процедури виконуються і при виконанні дотримуються декларовані для них правила. Потрібно особливо відзначити, що процес гарантії якості не гарантує розробку якісної програмної системи. Він гарантує тільки лише, що процеси розробки побудовані і виконуються таким чином, щоб не знижувати якість продукції. Всі процеси сертифікації критичних систем супроводжується великим потоком документів. Найоптимальніша модель з точки зору роботи з документом оборотом є V подібний життєвий цикл.

1.3 Процедура розробки FPGA пристроїв

Процес розробки FPGA повністю стандартизований, починаючи від ідеї і закінчуючи готовим продуктом [1]. Це й зрозуміло, враховуючи конкуренцію між постачальниками і відносну зрілість галузі автоматичного проєктування (EDA - Electronic Design Automation).

Проте так було не завжди. Спочатку пристрої, виготовлені на основі FPGA, були порівняно простими. Складність проєктів швидко збільшується. Згідно з дослідженням, проведеним компанією Mentor Graphics в 2010 р., більше половини проєктів на FPGA містять, принаймні, один процесор. У схемах такого рівня заміна, вилучення або додавання одного елемента може спричинити за собою великі зміни, а перехід на нову матрицю супроводжується переробкою всього проєкту [12].

У даний час загальноприйнята методологія високорівневого проєктування спеціалізованих інтегральних схем (ASIC) включає в себе етап введення схеми проєкту і опис його мовою HDL на рівні регістрових передач або поведінковому рівні. Розроблений пристрій потім верифікується спеціальною програмою HDL-моделювання та синтезується для реалізації на вентильній матриці. Проте, такий підхід застосовувався далеко не завжди.

Колись загальноприйнятим було проєктування на рівні вентилів. Але коли середнє число вентилів в спеціалізованій мікросхемі перевищило 10000, цей

метод почав давати збої. Крім того, стали більш жорсткими вимоги до тривалості циклу проєктування. У результаті всього цього, високорівневе проєктування стало обов'язковою частиною підтримки конкурентоспроможності нових пристроїв. Аналогічна ситуація спостерігається зараз в області розробки пристроїв на базі програмованих вентильних матриць (FPGA) [21].

З ростом складності розроблюваних систем і появою доступних інструментів високорівневого проєктування, ринок FPGA почав переорієнтовуватися на нову методику. Фактично, більшість експертів і аналітиків, зайнятих у цій галузі промисловості, висловлюють думку, що в найближчому майбутньому до методологій високорівневого проєктування звернуться десятки тисячі розробників пристроїв на FPGA [20]. Це відбувається здебільшого завдяки досягненням технології в області автоматизованого проєктування електронних пристроїв (EDA) і вдосконаленню обладнання та програмного забезпечення, що стався за останні 5-10 років.

Щоб задовольнити унікальні вимоги розробників систем на базі FPGA, ефективна методологія високорівневого проєктування повинна стати важелем, що підштовхує ці технології до подальшого розвитку. Всупереч очікуваної адаптації методів високорівневого проєктування до потреб розробників, почався зворотний процес, і вже розробники повинні були перебудовуватися і освоювати нові методи.

Незважаючи на те, що високорівневе проєктування не є оптимальним абсолютно для всіх завдань, привабливість цього методу не можна не помітити. При кількості вентилів у FPGA до 10000 візуалізувати виконувані функції системи, аналізуючи його схемо технічне рішення [11], відносно легко. Але коли кількість вентилів перевищило 20000, 50000 та 100000, гарантувати, що весь проєкт можна усвідомити, лише поглянувши на його схему, практично неможливо.

Використання мови опису апаратних засобів (HDL) спільно з відповідними інструментами моделювання дозволило розробникам організовувати і програмно

аналізувати набагато більш складні системи. Іншими словами, аналіз результатів комп'ютерного моделювання для перевірки функціонування системи, що розробляється замінив візуальну перевірку схеми.

Розробники систем на базі FPGA, що звернулися до методології високорівневого проєктування, зараз насолоджуються деякими очевидними перевагами. По-перше, будь-який окремо взятий розробник може забезпечити вирішення задачі підвищеної (і постійно збільшуваної) складності, звертаючись до більш високих рівнів абстракції і передаючи реалізацію дрібних деталей проєкту автоматизованому процесу. По-друге, розробники можуть значно скоротити цикл виробництва і покращити якість виробів, завдяки перевірці функціональних можливостей, ще на етапі проєктування, коли внесення змін до системи легко і відносно дешево. Ці переваги і змусили розробників спеціалізованих мікросхем (ASIC) звернутися до високорівневого проєктування.

Перехід до нової методології проєктування ніколи не відбувається легко. Завжди є певні перешкоди на шляху до її впровадження. Шлях переходу розробників систем на базі FPGA до методології високорівневого проєктування не є винятком з правил.

Однією з перешкод на цьому шляху є загальне неправильне уявлення, що у програмному забезпеченні для пристроїв на базі FPGA розробникам потрібно значно меншу кількість функціональних можливостей, ніж для спеціалізованих мікросхем (ASIC). Насправді ж, всі вимоги, пов'язані з скороченням шляху виробу на ринок, справедливі і для них. Розробники пристроїв на базі FPGA будуть прагнути використовувати всі доступні їм функціональні можливості максимальним чином. Вони можуть включати вдосконалені методи налагодження, такі як аналіз потоків даних, використання мов VHDL і Verilog або передові методи моделювання та оцінки продуктивності.

Як правило, доступні розробникам пристроїв на FPGA кошти були або зовсім новими, невідпрацьованими, або дуже посередніми, що не забезпечують досягнутої якості та функціональних можливостей конкуруючих засобів

проєктування ASIC. Ця тенденція вимагала корінного перелому. Сьогодні кожен компонент пакету HDL проєктування пристроїв на FPGA повинен бути лідером у своєму класі і забезпечувати можливість розширення функціональних можливостей, порівнянну з аналогічними інструментами проєктування спеціалізованих інтегральних схем.

Іншою перешкодою, що вимагає обов'язкового подолання, є економічна відмінність між тим, які кошти платить розробник ASIC і тим, скільки може дозволити собі розробник пристроїв на FPGA. У той час, як вартість програмного забезпечення робочого місця проєктувальника ASIC традиційно складає більше $ 100000, на одне робоче місце проєктувальника з використанням FPGA компанії витрачають, як правило, не більше $ 10000. Значить, постачальники програмного забезпечення для автоматизованого проєктування, що сподіваються обслуговувати ринок пристроїв на FPGA, повинні враховувати, що їхні продукти повинні мати відношення вартість/функціональність набагато меншу, ніж те , що пропонується для ASIC. Здавалося б, це економічна вимога знаходиться в протиріччі з вимогами здійснення необхідної гнучкості і функціональності. Однак, ряд спеціалізованих професійних фірм пропонує такі кошти вже сьогодні.

Забезпечення високого рівня програмних засобів проєктування накладає певні вимоги на ефективність роботи їх користувача. Збереження цих організацій невеликими і сконцентрованими на певній задачі дозволяє значно скоротити накладні витрати. Крім того, компанії, розташовані до нововведень в галузі електронної торгівлі, електронного поширення програмного забезпечення та використанню для цього дешевих каналів, можуть значно знизити витрати, що спричинить за собою економію коштів у їхніх клієнтів, а значить, позитивно відіб'ється на вартості кінцевих виробів.

Зрозуміло, ключовим моментом у забезпеченні переходу до методології високорівневого проєктування є пошук шляхів переходу розробників до HDL-проєктування без втрат в обсягах виробництва [14]. Багато постачальників засобів проєктування відповіли на це випуском спрощених продуктів, але цього явно

недостатньо. Необхідно проводити навчання, яке дозволить розробникам пристроїв на FPGA значно збільшити рівень власних знань. У кінцевому рахунку, вдала комбінація зручних у роботі засобів проєктування і відповідного навчання дозволить згладити болісний перехід.

Для розробки FPGA пристроїв можна використовувати моделі розробки інформаційних систем. Одною з популярних моделей є V-подібна модель. Вона підходить більше для розробки ніж для обслуговування, ремонту та утилізації системи. Також дозволяє на будь-яких етапах розробки звертатися до тестування.

1.3.1 V-подібна модель розробки

V-подібна модель була створена з метою допомогти команді, що працює над проєктом в плануванні із забезпеченням подальшої можливості тестування системи. У цій моделі особливе значення надається діям, спрямованим на верифікацію та атестацію продукту. Вона демонструє, що тестування продукту обговорюється, проєктується і планується на ранніх етапах життєвого циклу розробки. План випробування приймання замовником розробляється на етапі планування, а компонувального випробування системи – на фазах аналізу, розробки проєкту і т.д. Цей процес розробки планів випробування позначений пунктирною лінією між прямокутниками V-подібної моделі(рис. 1.2.).

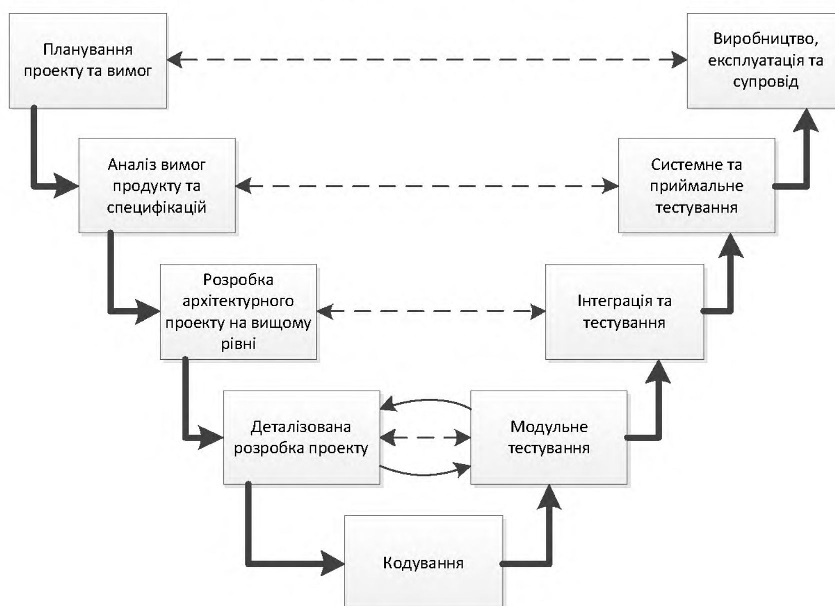

Рисунок. 1.2 – V-подібна модель життєвого циклу розробки ПЗ

1.3.2 Фази розробки V-подібної моделі

Нижче подано короткий опис кожної фази V-подібної моделі, починаючи від планування проєкту та вимог аж до приймальних випробувань:

- планування проєкту та вимог – визначаються системні вимоги, а також те, яким чином будуть розподілені ресурси організації з метою їх відповідності поставленим вимогам. (у разі необхідності на цій фазі виконується визначення функцій для апаратного та програмного забезпечення);

- аналіз вимог до продукту і його специфікації – аналіз існуючої на даний момент проблеми з ПЗ, завершується повною специфікацією очікуваної зовнішньої лінії поведінки створюваної програмної системи;

- архітектура або проєктування на вищому рівні – визначає, яким чином функції ПЗ повинні застосовуватися при реалізації проєкту;

- деталізована розробка проєкту - визначає і документально обґрунтовує алгоритми для кожного компонента, який був визначений на фазі побудови архітектури. Ці алгоритми в наслідку будуть перетворені в код;

- розробка програмного коду – виконується перетворенням алгоритмів, визначених на етапі деталізованого проєктування, в готове ПЗ;

- модульне тестування – виконується перевірка кожного закодованого модуля на наявність помилок;

- інтеграція і тестування – установка взаємозв'язків між групами раніше поелементно випробуваних модулів з метою підтвердження того, що ці групи працюють також добре, як і модулі, випробувані незалежно один від одного на етапі поелементного тестування;

- системне і приймальне тестування – виконується перевірка функціонування програмної системи в цілому (повністю інтегрована система), після переміщення в її апаратне середовище відповідно до специфікації вимог до ПЗ;

- виробництво, експлуатація та супровід - ПЗ запускається у виробництво. На цій фазі передбачені також модернізація і внесення поправок;

- приймальні випробування (на рис. не показані) - дозволяє користувачеві протестувати функціональні можливості системи на відповідність вихідним вимогам. Після остаточного тестування ПЗ й довколишнє апаратне забезпечення стають робочими. Після цього забезпечується супровід системи.

Переваги V-подібної моделі

При використанні V-подібної моделі при розробці проєкту, для якого вона в достатній мірі підходить, забезпечується кілька переваг:

- в моделі особливе значення надається плануванню, направленому на верифікацію та атестацію розроблюваного продукту на ранніх стадіях його розробки. Фаза модульного тестування підтверджує правильність деталізованого проєктування. Фази інтеграції та тестування реалізують архітектурне проєктування або проєктування на вищому рівні. Фаза тестування системи підтверджує правильність виконання етапу вимог до продукту і його специфікації;

- в моделі передбачені атестація та верифікація всіх зовнішніх і внутрішніх отриманих даних, а не тільки самого програмного продукту;

- в V-подібної моделі визначення вимог виконується перед розробкою проєкту системи, а проєктування ПЗ - перед розробкою компонентів;

- модель визначає продукти, які повинні бути отримані в результаті процесу розробки, причому всі отримані дані повинні піддаватися тестуванню;

- завдяки моделі менеджер проєкту може відслідковувати хід процесу розробки, так як в даному випадку цілком можливо скористатися тимчасовою шкалою, а завершення кожної фази є контрольною точкою;

- модель проста у використанні (щодо проєкту, для якого вона є прийнятному).

Недоліки V-подібної моделі

При використанні V-подібної моделі в роботі над проєктом, для якого вона не є в достатній мірі прийнятною, стають очевидними її недоліки:

- з її допомогою непросто впоратися з паралельними подіями;

- в ній не враховані ітерації між фазами;

- в моделі не передбачено внесення вимоги динамічних змін на різних етапах життєвого циклу;

- тестування вимог в життєвому циклі відбувається занадто пізно, внаслідок чого неможливо внести зміни, що не вплинувши при цьому на графік виконання проєкту;

- в модель не входять дії, спрямовані на аналіз ризиків.

Область застосування V-подібної моделі

Подібно своїй попередниці, каскадної моделі, V-подібна модель найкраще спрацьовує тоді, коли вся інформація про вимоги доступна заздалегідь. Загальнопоширених модифікація V-подібної моделі, спрямована на подолання її недоліків, включає в себе внесення ітераційних циклів для дозволу зміни у вимогах за рамками фази аналізу.

Використання моделі ефективно в тому випадку, коли доступними є інформація про метод реалізації рішення і технологія, а персонал володіє необхідними вміннями і досвідом у роботі з даною технологією.

V-подібна модель – це відмінний вибір для систем, в яких потрібна висока надійність. Вона використовується при створенні інформаційно-керуючих систем, в критичних галузях, таких як авіація, енергетика, космічні апарати.

1.3.3 Призначення моделі конфігураційної пам’яті

Модель конфігураційної пам’яті, що нам потрібно розробити, потрібна в процесі тестування, а саме в модульному тестуванні.

Метою модульного тестування є взяти невеликий шматок коду, який відповідає за включення деяких дуже специфічних можливостей в програмному забезпеченні, що розробляється, і тестуванні його. Щоб переконатися, що він веде себе саме так, як ми того хочемо, в різних умовах. Такий підхід дозволяє тестувати внутрішні частини програмного забезпечення, які зазвичай захищені безпосередньо від кінцевого користувача, наприклад, за допомогою графічного інтерфейсу користувача або екрану, а отже, не можуть бути легко перевірені через

звичайне тестування. Ця форма тестування також забезпечує ранній зворотній зв'язок для розробників, можливість визначити чи вони йдуть у правильному напрямку, і дозволяє їм зробити невеликі зміни якщо необхідно.

У звичайних випробувань, які можуть бути ручними або автоматизованими, перевірка функціональності зазвичай відбувається після розробки програмного забезпечення і до цього часу стає майже неможливо швидко вирішити які-небудь критичні дефекти або недоліки дизайну, і майже завжди затримує доставку програмного забезпечення. Тим не менш, з модульним тестуванням робота програміста перевіряється набагато швидше, шляхом тестування невеликих модулів програмного забезпечення по мірі їх розробки, що дозволяє швидко вносити зміни, якщо виявлені дефекти або відхилення від оригінального дизайну.

Є багато проблем на шляху засвоєння модульного тестування. Вони варіюються від страху робити щось нове, відсутності бажання збільшити якість коду і продуктивність програмістами, а також, не освідомлення про належну практику у в модульному тестуванні. Неправильно реалізовані модульні тести іноді можуть принести більше шкоди, ніж користі. Ці проблеми можна вирішити ефективно читаючи і розуміючи, що зробили інші, а потім трохи експериментів, перш ніж приступити до повного, або принаймні обмеженого прийняття.

1.4 Узагальнена архітектура і функціональні вимоги до об’єкта моделювання

Послідовні конфігураційні пристрої є флеш-пам'яттю з послідовним інтерфейсом, який може зберігати дані конфігурації для FPGA пристроїв [20-21], що підтримують активну послідовну конфігурацію і перезавантажують дані на пристрій при включенні живлення або реконфігурації.

В SRAM-пристроях, які підтримують активну послідовну конфігурації, дані конфігурації повинні бути перезавантаженні щоразу, коли пристрій включається, система змінює конфігурацію, або коли потрібні нові дані конфігурації.

Конфігураційні пристрої можуть мати процесор для доступу до невикористаної пам’яті та оснащуватися захистом запису для секторів пам’яті, використовуючи статус регістр.

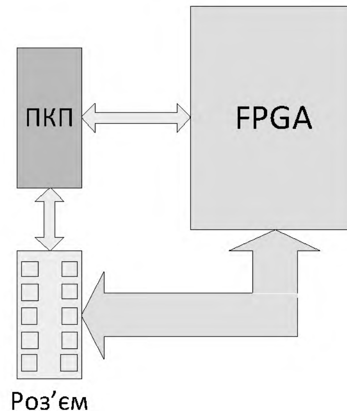

Рисунок. 1.3 – Узагальнена архітектура об’єкта моделювання

На рис. 1.3 зображена узагальнена архітектура об’єкта моделювання [15,23].

Основні операції конфігураційної пам’яті:

- Запис дозволений – дозволяє операції що змінюють вміст пам’яті;
- Запис заборонений – забороняє операції що змінюють вміст пам’яті;
- Прочитати статус – читає вміст регістра статусу;
- Прочитати байт – читає вміст пам’яті;
- Прочитати ідентифікатор кристалу – видає код пристрою;
- Швидке читання – прискорений режим читання пам’яті;
- Записати статус – записує дані в регістр статусу;
- Записати байт – записує дані в пам’ять;
- Очистити масив – видаляє дані з пам’яті;
- Очистити сектор – видаляє дані з цілого сектору пам’яті.

Після завершення процесу розробки та налагодження автономного пристрою на базі FPGA необхідно забезпечити його енергонезалежність. Для конфігурації FPGA в автономних пристроях використовують два способи: завантаження із зовнішнього паралельного ПЗП або мікропроцесора (режим Passive Parallel) та завантаження з послідовного конфігураційного ПЗП (режим

Passive Serial та Active Serial). В нашому випадку використовується конфігураційна пам'ять, тому слід розглянути режим Active Serial.

1.4.1 Режими підключення

Конфігураційна пам'ять використовує 4 контакти для взаємодії з керуючими сигналами FPGA [15,23]. Сигнали конфігураційного пристрою DATA, DCLK, ASDI та nCS з'єднуються з керуючими сигналами DATA0, DCLK, ASDO, та nCSO на ПЛІС відповідно.

Підключення конфігураційної пам'яті до FPGA можливе кількома способами. На рис.1.2. зображено спосіб з використанням кабелю завантаження який налаштовує FPGA в режимі Active Serial (AS).

FPGA діє як майстер конфігурації і забезпечує конфігураційну пам'ять імпульсом синхронізації. FPGA активує конфігураційну пам'ять спаданням сигналу nCS через nCSO сигнал (відповідно до рис. 1.2 та 1.3). Потім FPGA посилає інструкції та адреси до конфігураційної пам'яті через сигнал ASDO. Конфігураційна пам'ять реагує на інструкції надсилаючи конфігураційні дані в DATA0 контакт на FPGA при спаді сигналу з DCLK. Дані фіксуються в FPGA при наступному спаданні сигналу DCLK [15, 23].

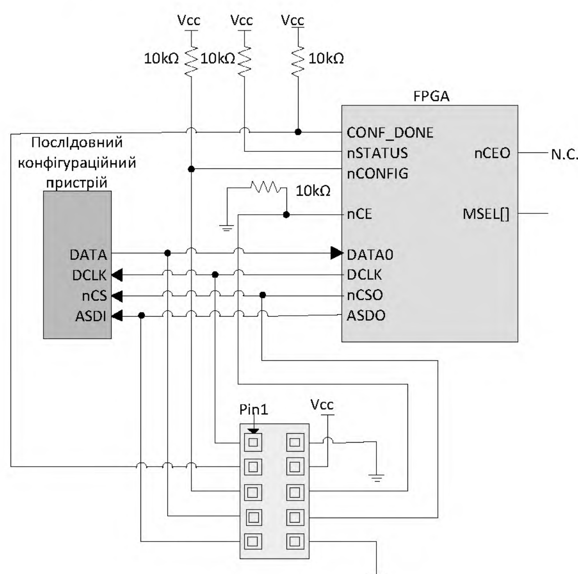

Рисунок. 1.4 – Підключення конфігураційного пристрою до FPGA в режимі AS(конфігураційна пам'ять запрограмована з використанням кабелю завантаження)

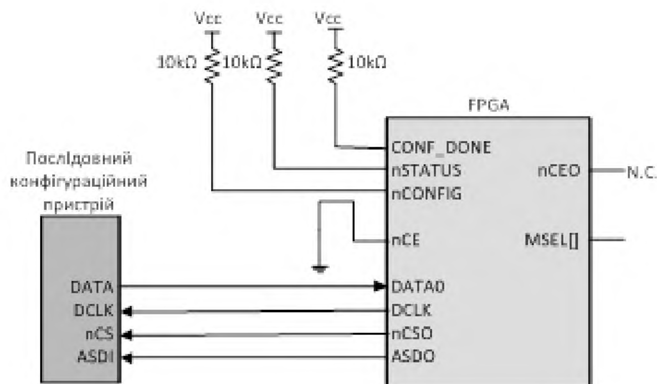

Рисунок 1.5 – Підключення конфігураційного пристрою до FPGA в режимі AS (конфігураційна пам’ять запрограмована з використанням стороннього програматора)

FPGA контролює nSTATUS і CONF_DONE контакти при конфігурації в AS режимі. Якщо сигнал CONF_DONE не стає високим при закінченні конфігурації або якщо стає високим занадто рано, то FPGA змінить сигнал nSTATUS на низький щоб почати реконфігурацію. Після успішної конфігурації FPGA звільняє контакт CONF_DONE, дозволяючи резистору 10кОм змінити цей сигнал на високий. Ініціалізація починається після того як сигнал CONF_DONE стає високим. Після ініціалізації FPGA переходить в режим користувача.

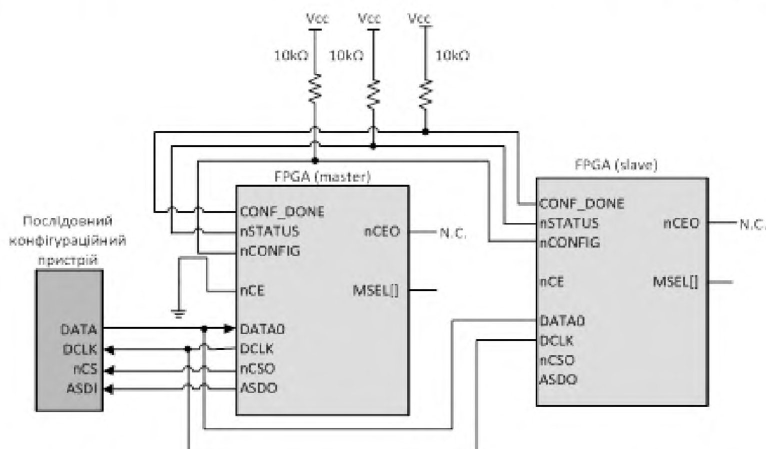

Рисунок 1.6 – Підключення конфігураційного пристрою до декількох FPGA в режимі AS

Кілька ПЛІС можуть бути налаштовані однією конфігураційною пам’яттю. Однак, конфігураційна пам’ять не може приєднуватися каскадно, тому потрібно

дивитися щоб програмований розмір файлу каскадних ПЛІС не перевищував розмір пам'яті конфігураційного пристрою. На рис. 1.4 показана схема з декількома ПЛІС в ланцюзі.

З переглянутого вище матеріалу видно що конфігураційна пам'ять працює з чотирма сигналами (3ма вхідними і одним вихідним). Вона здатна програмувати декілька FPGA, може бути запрограмована як за допомогою кабелю завантаження так і самою FPGA [15,23].

Висновки з розділу 1

В розділі детально розглянуто технологію FPGA, чому ця ПЛІС така популярна в наш час, і навіщо їй потрібна конфігураційна пам'ять. Також розглянуто режими підключення конфігураційної пам'яті до FPGA.

Поставлено завдання щодо подальших досліджень, а саме розроблення моделі конфігураційної пам'яті FPGA для її використання в ході тестування компонентів і електронних проєктів ПЛІС.

РОЗДІЛ 2
РОЗРОБЛЕННЯ МОДЕЛІ КОНФІГУРАЦІЙНОЇ ПАМ’ЯТІ

У цьому пункті ми розглянемо структурну схему конфігураційного пристрою, організацію масиву пам’яті, операції, які він підтримує та коди операцій.

На рис. 2.1 зображена структурна схема конфігураційного пристрою.

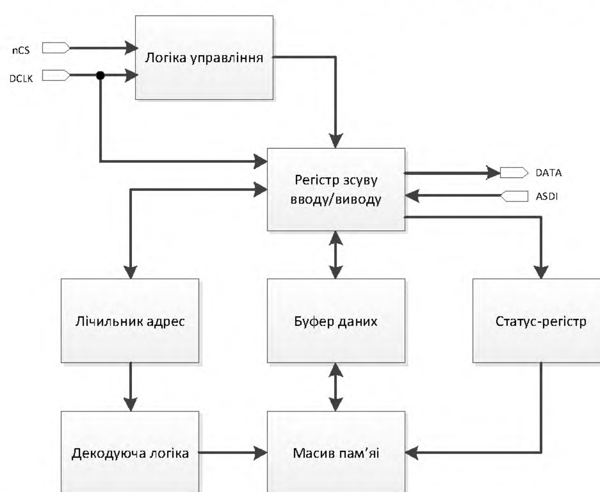

Рисунок. 2.1 – Структурна схема конфігураційної пам’яті

Блок логіки управління – блок керування на основі сигналу nCS.

Регістр зсуву вводу/виводу – регістр, який послідовно отримує і передає дані керуючись сигналами блоку логіки управління.

Лічильник адрес – лічильник, який збільшує початкову адресу пам’яті, як тільки блок даних буде переданий в комірки пам'яті, зазначені лічильником.

Буфер даних – область пам’яті, яка використовується для тимчасового зберігання даних.

Статус-регістр – регістр, що зберігає важливу інформацію про стан конфігураційної пам’яті.

Декодуюча логіка – блок управління адресами.

Масив пам’яті – пам’ять для збереження даних.

2.1.1 Організація масиву пам’яті

В таблиці 2.1 міститься докладна інформація про організацію масиву пам'яті в EPCS1[15,23].

Таблиця 2.1 – Організація масиву пам’яті в послідовному конфігураційному пристрої EPCS1

Деалі	EPCS1
Байти (біти)	131,072 байт (1 Мбіт)
Кількість секторів	4
Байтів (бітів) в секторі	32,768 байт (256 Кбіт)
Сторінок в секторі	128
Загальна кількість сторінок	512
Байт на одну сторінку	256 байт

В таблиці 2.2 міститься список діапазонів адрес для кожного сектора в EPCS1.

Таблиця 2.2 – Діапазон адрес для секторів в EPCS1

Сектор	Діапазон адрес	
	Початок	Кінець
3	H’18000	H’1FFFF
2	H’10000	H’17FFF
1	H’08000	H’0FFFF
0	H’00000	H’07FFF

2.1.2 Коди операцій

Опишемо операції, які можна використовувати для доступу до пам'яті в послідовному конфігураційному пристрої [15,23]. Для доступу до пам’яті потрібно використовувати сигнали DATA, DCLK, ASDI і NCS. Всі коди операцій, адреси і дані послідовного конфігураційного пристрою зміщуються з середини пристрою назовні – послідовно, причому найбільш старший біт (MSB) в першу чергу.

Пристрій підбирає(пробує) активний послідовний вхід при передньому фронті DCLK після спадання nCS. Зсуває код операції (MSB перший) послідовно в конфігураційний пристрій через контакт Active Serial Data input (ASDI). Кожен

біт коду операції фіксується в конфігураційному пристрої при передньому фронті DCLK.

Різні операції вимагають різної послідовності входів. Коли виконується операція, ми повинні зсувати бажаний код операції, керуючись адресними байтами, байтами даних, обома, або жодним. Пристрій повинен встановити nCS високим після зсуву останнього біту. В таблиці 2.3 поданий список операційних послідовностей для кожної операції підтримуваної послідовним конфігураційним пристроєм.

Таблиця 2.3 – Коди операцій послідовного конфігураційного пристрою

Операція	Код операції *(1)*	Байти адреси	Фіктивні байти	Байти даних	DCLK f_{max} (МГц)
Write enable	0000 0110	0	0	0	25
Write disable	0000 0100	0	0	0	25
Read status	0000 0101	0	0	Від 1 *(2)*	25
Read bytes	0000 0011	3	0	Від 1 *(2)*	20
Read silicon ID	1010 1011	0	3	Від 1 *(2)*	25
Fast read	0000 1011	3	1	Від 1 *(2)*	40
Write status	0000 0001	0	0	1	25
Write bytes	0000 0010	3	0	Від 1 до 256 *(3)*	25
Erase bulk	1100 0111	0	0	0	25
Erase sector	1101 1000	3	0	0	25

Примітки до таблиці 2.3:

(1) MSB вказаний першим і молодший значущий біт (LSB) перерахований останнім.

(2) Регістр стану, даних або ідентифікатор кремнію зчитуються принаймні раз на контакті DATA і буде постійно читатися доки nCS знаходитиметься на високому рівні.

(3) Операція запису байта вимагає принаймні одного байту даних на контакті DATA. Якщо більш ніж 256 байтів відправляються на пристрій, тільки останні 256 байт записуються в пам'ять.

Для операцій «read bytes», «read status», та «read silicon ID», зсунута операційна послідовність слідує за послідовністю вихідних даних на контакті

DATA. Ми можемо встановити контакт nCS високим після будь-якого біту послідовності даних, що зсувається із пристрою.

Для операцій «write byte», «erase bulk», «erase sector», «write enable», «write disable», та «write status», потрібно встановити контакт nCS високим точно на кордоні байту (встановити контакт nCS високим кратно восьми тактовим імпульсам після того як контакт nCS встановлений на низький); інакше, операція відкинеться і не виконається.

Всі спроби отримати доступ до вмісту пам’яті коли триває цикл запису або видалення будуть відкинуті, та цикл запису/ видалення продовжуватиметься без змін.

2.1.3 Операція «write enable»

Кодом операції «write enable» є b’0000 0110, і MSB йде першим. Операція «write enable» встановлює однойменний біт-засувку, який є першим в статус регістрі. Потрібно завжди встановлювати біт-засувку «write enable» перед записом байтів, На рис. 2.2 показана часова діаграма для операції «write enable». На рис. 2.4 та 2.5 зображено призначення бітів статус-регістра.

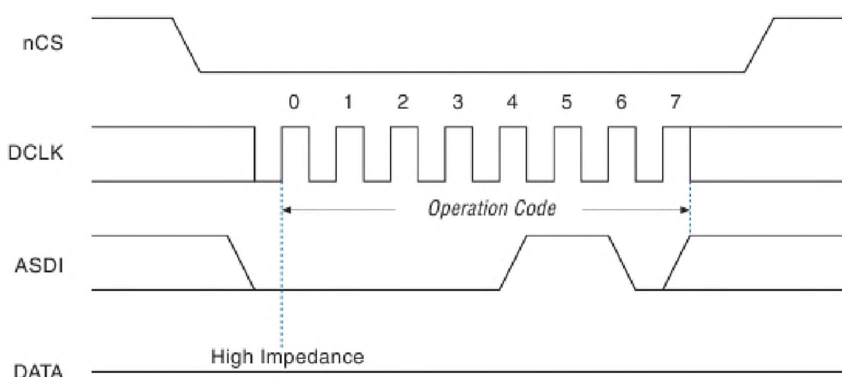

Рисунок 2.2 – Часова діаграма операції «write enable»

2.1.4 Операція «write disable»

Операція «write disable» має код b’0000 0100, MSB перший. Дана операція скидає біт-засувку, що відповідає за операцію «write enable» і є першим в статус регістрі. Для запобігання ненавмисного запису в пам’ять, біт-засувка автоматично скидається при виконанні операції «write disable», а також за наступних умов:

- вимкнуто живлення;

- завершенні операції write bytes;
- завершення операції write status;
- завершення операції erase bulk;
- завершення операції erase sector.

На рис. 2.3 показана часова діаграма для операції «write disable».

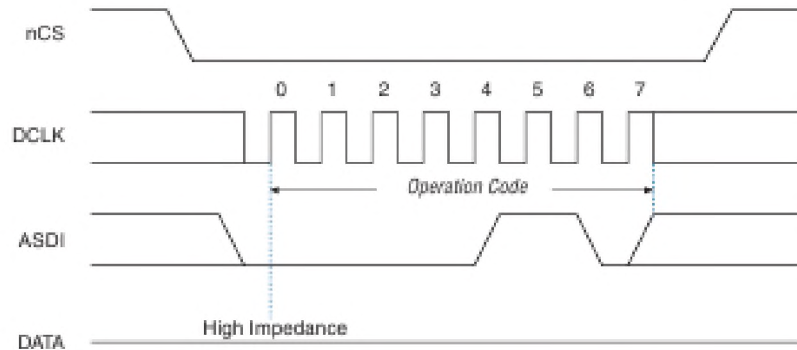

Рисунок 2.3 – Часова діаграма операції «write disable»

2.1.5 Операція «read status»

Код операції «read status»: b’0000 0101, MSB іде першим. Ми можемо використовувати цю операцію для читання вмісту статус-регістра. На рис. 2.4 показані біти статус-регістру послідовного конфігураційного пристрою.

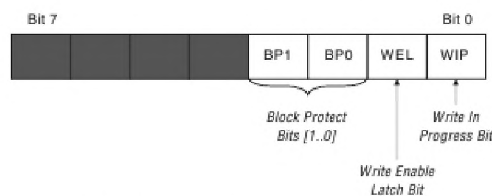

Рисунок 2.4 – Біти статус-регістра

Встановлення біту «write in progress» в 1 вказує на те, що послідовний конфігураційний пристрій зайнятий циклом запису або видалення. Скидаючи біт «write in progress» в 0 означає відсутність активних циклів запису або видалення. Скидаючи біт-засувку «write enable» в 0 вказує на те, що цикли запису або видалення не прийматимуться. Потрібно встановлювати біт-засувку «write enable» в 1 перед кожною операцією «write bytes», «write status», «erase bulk», та «erase sector».

Енергонезалежний блок мимоволі захищає біти визначення площі пам’яті, захищеної від запису або видалення. В таблиці 2.4 наведений список захищених областей послідовного конфігураційного пристрою у відповідності до бітів «block protect». Операція «erase bulk» доступна тільки тоді, коли всі біти захисту рівні 0. Коли один з бітів встановлений в 1, то відповідна частина захищена від запису операцією «write byte» або від видалення операцією «erase sector».

Таблиця 2.4 – Біти захисту

Вміст статус-регістру		Пам’ять	
Біт BP1	Біт BP0	Захищена область	Незахищена область
0	0	Відсутня	Всі чотири сектори: 0 до 3
0	1	Сектор 3	Три сектори: 0 до 2
1	0	Два сектори: 2 та 3	Два сектори: 0 та 1
1	1	Всі сектори	Відсутня

Ми можемо читати статус-регістр в будь-який час, навіть якщо активні цикли запису або видалення. Коли один з цих циклів активний, ми можемо перевірити запис в біті прогресу (біт 0 в статус-регістрі) перед посиланням нової операції в пристрій. Прилад також може читати статус-регістр безперервно, згідно рис. 2.5.

2.1.6 Операція «write status»

Код операції «write status»: b’0000 0001, зі старшим бітом – першим. Операція використовується для встановлення біта захисту блоків в статус-регістрі. Операція не впливає на інші біти. Отже, ми можемо виконувати цю операцію щоб захистити сектори пам’яті, вказані в таблиці 2.4. Після встановлення біта захисту блоків, сектори в захищеній пам’яті надаються тільки для читання. Ми повинні виконати операцію «write enable» перед операцією «write status» щоб прилад у статус-регістрі встановив біт-засувку «write enable» в 1.

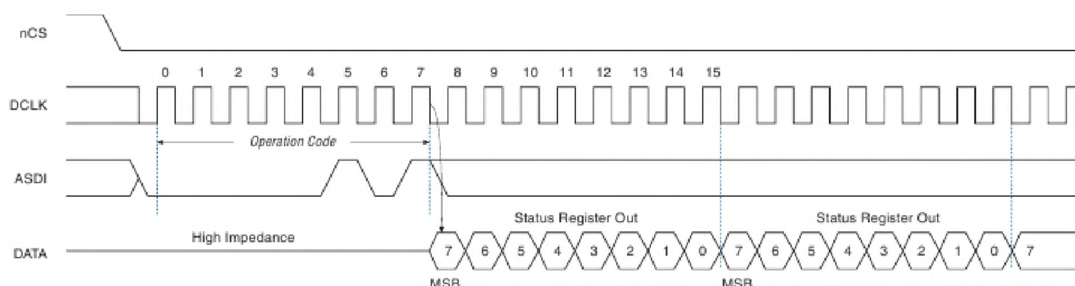

Рисунок 2.5 – Часова діаграма операції «read status»

Операція «write status» реалізовується при спаданні сигналу nCS, що слідує за зсувом коду операції «write status» та одним байтом даних для статус-регістра на контакті ASDI. На рис. 2.6 зображена часова діаграма операції «write status». nCS повинен перейти на високий рівень після восьми біт даних, що були зафіксовані, інакше операція не буде виконана.

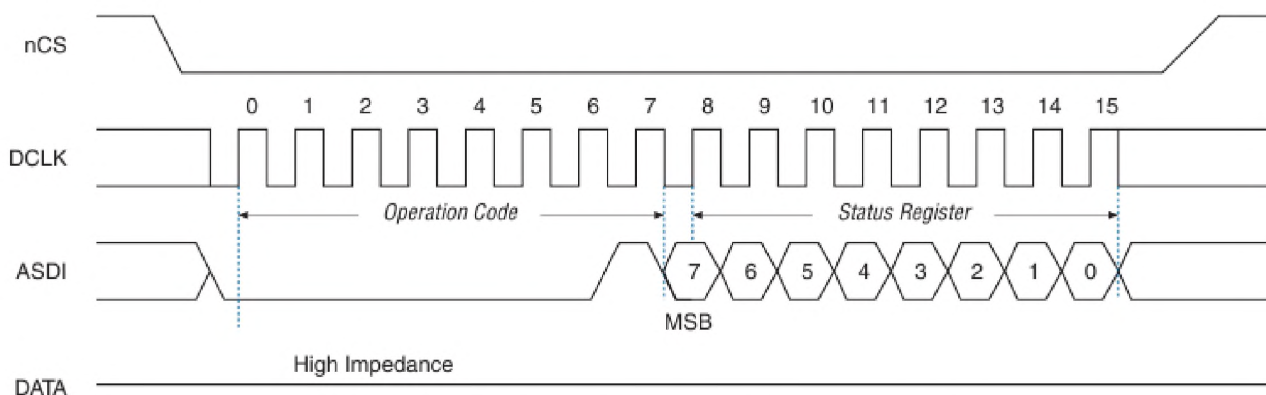

Рисунок 2.6 – Часова діаграма операції «write status»

Відразу після встановлення сигналу nCS на 1, пристрій ініціює само синхронний цикл «write status». Самосинхронний цикл «write status» зазвичай займає 5 мс для всіх послідовних конфігураційних пристроїв і гарантовано менше 15 мс (згідно до t_{WS} таблиці 2.6). Ми повинні враховувати цю затримку, щоб гарантувати, що в регістр стану написані бажані біти захисту блоків. Крім того, ми можемо перевірити біт «write in progress» в статус-регістрі виконавши операцію «read status» у процесі само-синхронного циклу «write status». Біт «write in progress» рівний 1 у процесі само-синхронного циклу «write status», і рівний 0 коли цикл буде завершений.

2.1.7 Операція «read bytes»

Код операції «read bytes» являє собою двійкову послідовність 0000 0011, старший біт заноситься першим. Щоб прочитати вміст пам’яті послідовного конфігураційного пристрою, прилад спочатку вибирається падінням сигналу на nCS. Потім, код операції «read bytes» зсувається перед 3-ох байтовою адресою. Кожен біт адреси повинен бути зафіксований на передньому фронті DCLK. Після фіксації адреси, вміст пам’яті за вказаною адресою зсувається послідовно на

контакт DATA, зі старшим бітом - першим. Для читання файлів Raw Programming Data (rpd), вміст зсувається послідовно з молодшим бітом – першим. Кожен біт даних зсувається при задньому фронті DCLK. Максимальна частота DCLK під час операції прочитати байт – 20 МГц. На рис. 2.7 показана часова діаграма операції «read bytes».

Перший байт адреси може бути в будь-якому місці. Пристрій автоматично збільшує адресу на одиницю до наступної адреси, після зсуву кожного байту даних. Отже, прилад може прочитати всю пам’ять за одну операцію «read bytes». Як тільки прилад досягне максимальної адреси, лічильник адрес перезапускається на 0x000000, дозволяючи читати вміст пам’яті невизначений час доки операція «read bytes» не буде перервана високим сигналом nCS. Прилад може встановити nCS високим в будь-який час після зсуву даних. Якщо операція «read bytes» зсувається під час виконання циклу запису або видалення, операція не виконується і не має ефекту на цикли запису або видалення.

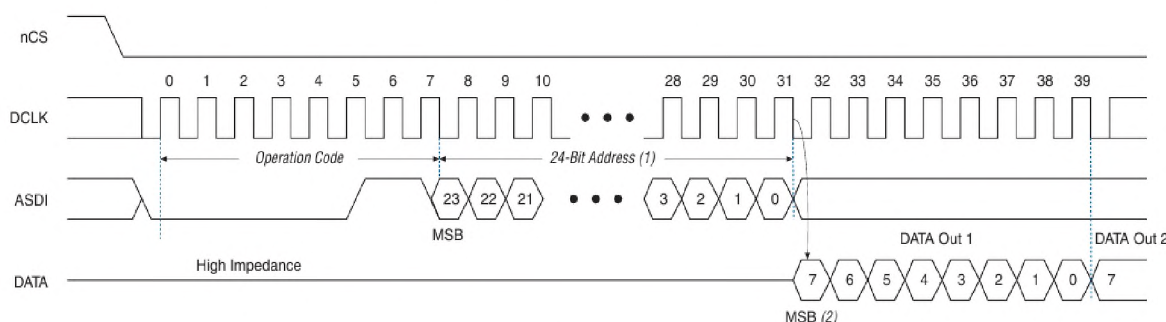

Рисунок 2.7 – Часова діаграма операції «read bytes»

Примітки до рис. 2.7:

(1) Біти адрес A[23] не враховується в EPCS64. Біти адрес A[23..21] не враховується в EPCS16. Біти адрес A[23..19] не враховується в EPCS4. Біти адрес A[23..17] не враховується в EPCS1.

(2) Для rpd файлів, послідовність, що читається, зсувається найменш значущим бітом першим(LSB).

2.1.8 Операція «fast read»

Прилад вибирається падінням сигналу на nCS. Код операції «fast read» передує 3-х байтовій адресі (A23-A0) та холостому біту, кожен біт фіксується

переднім фронтом DCLK. Потім вміст пам’яті за заданою адресою зсувається на контакт DATA, кожен біт зсувається на максимальній частоті 40 МГц, при задньому фронті DCLK.

Послідовність команд зображена на рис. 2.8.

Перший байт адреси може бути в будь-якому місці. Адреса автоматично збільшується на наступну більш високу адресу після кожного байта даних, що зсувається. Вся пам'ять може бути прочитана однією інструкцією «fast read». Коли досягнена найбільша адреса, лічильник адрес зашкалює на 000000h, дозволяючи читати послідовність невизначений термін. Інструкція «fast read» завершується підвищенням сигналу nCS в будь-який час під час виходу даних. Будь-які інструкції «fast read» відміняються під час операцій видалення, програмування, або запису без будь-якого ефекту на операцію, що триває.

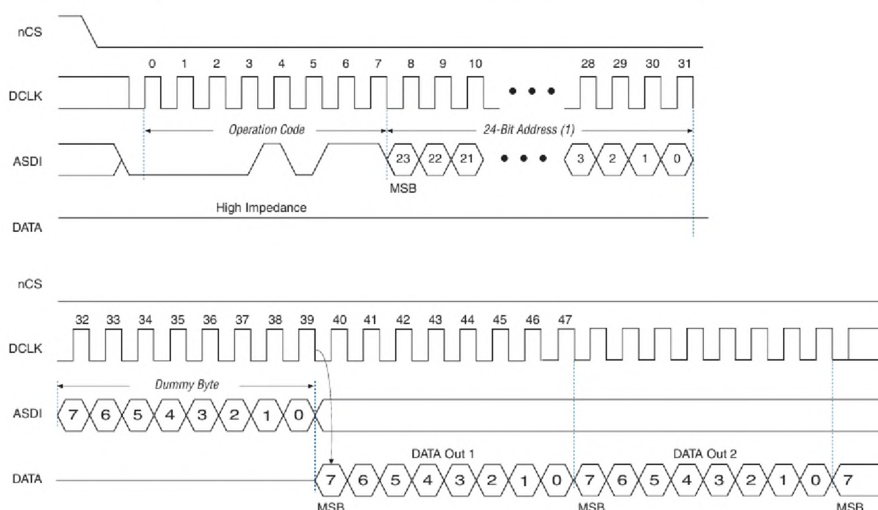

Рисунок 2.8 – Часова діаграма операції «fast read»

Примітки до рис. 2.8:

(1) Біти адреси A[23] не враховуються в EPCS64. Біти адреси A[23..21] не враховуються в EPCS16. Біти адреси A[23..19] не враховуються в EPCS4. Біти адреси A[23..17] не враховуються в EPCS1.

2.1.9 Операція «read silicon ID»

Кодом операції «read silicon ID» є: b’10101011, зі старшим бітом – першим. Цю операцію підтримують моделі EPCS1, EPCS4, EPCS16, та EPCS64. Вона читає 8-ми бітний ідентифікатор кристалу послідовного конфігураційного пристрою з

виходу DATA. Якщо ця операція надійшла під час циклу видалення або запису, вона ігнорується і не має ефекту на цикли, що тривають.

Таблиця 2.5 містить список ідентифікаторів послідовних конфігураційних пристроїв.

Таблиця 2.5 – Ідентифікатори кристалів послідовних конфігураційних пристроїв

Послідовний конфігураційний пристрій	Ідентифікатор кристалу (в двійковий формі)
EPCS1	b’0001 0000
EPCS4	b’0001 0010
EPCS16	b’0001 0100
EPCS64	b’0001 0110

Пристрій реалізує операцію «read silicon ID» падінням сигналу nCS та зсувом коду операції «read silicon ID», що передують трьом холостим бітам на ASDI. Як показано на рис. 2.9, 8-бітний ідентифікатор кристалу послідовного конфігураційного пристрою потім зсувається на контакт DATA при задньому фронті DCLK. Пристрій може припинити операцію «read silicon ID» встановленням високого сигналу nCS, після читання ідентифікатора приладу принаймні один раз. Відправка додаткових тактів на DCLK при низькому nCS може викликати повторне зміщення ідентифікатора кремнію.

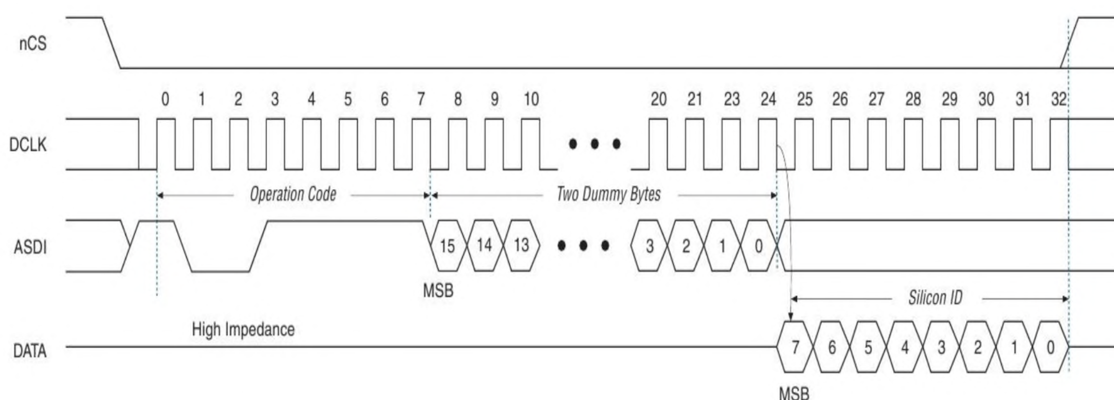

Рисунок 2.9 – Часова діаграма операції «read silicon ID»

Примітка до рис. 2.9:

(1) Тільки EPCS1, EPCS4, EPCS16, та EPCS64 підтримують операцію «read silicon ID»

2.1.10 Операція «write byte»

Код операції «write byte» – b’0000 0010, зі старшим бітом – першим. Операція «write enable» повинна бути виконана до операції «write byte» щоб встановити біт-засувку «write enable» в статус-регістрі в 1.

Операція «write byte» реалізується спаданням nCS та надходженням коду операції, трьох адресних байтів та мінімум одного байту даних на ASDI. Якщо вісім молодших біта адреси ([7 .. 0]) не всі рівні 0, то всі відправлені дані, які виходять за рамки поточної сторінки не записуються в наступну сторінку. Замість цього дані записуються на початкову адресу тієї ж сторінки (з адреси в якої вісім молодших розрядів є всі 0). Падіння сигналу nCS протягом усього робочого циклу «write byte», як показано на рис. 2.10.

Якщо більше 256 байтів даних переміщаються в серійний конфігураційний пристрій операцією «write byte», то заздалегідь замкнуті дані відкидаються і останні 256 байтів записуються на сторінку. Проте, якщо менш ніж 256 байтів даних переміщаються в послідовний конфігураційний пристрій, вони гарантовано будуть записаними у вказаних адресах і інші байти тієї ж сторінки залишаться незміненими.

Якщо потрібно написати більше ніж 256 байт даних в пам'ять, для цього потрібно більше однієї сторінки пам'яті. Коди операцій write enable і write byte надсилаються після трьох байт нових цільових адрес і 256 байт даних перед записом нової сторінки.

nCS повинен стати високим після восьми біт останнього байта даних, що був зафіксований. В іншому разі пристрій не виконуватиме операцію write byte. Біт засувка «Write enable» в статус-регістрі скидається в 0 до завершення кожної операції «write byte». Таким чином, операція «write enable» повинна проводитися до наступної операції «write byte».

Пристрій ініціює самосинхронний цикл запису відразу після встановлення високого сигналу nCS. В таблиці 2.6 на сторінкі 43 t_{WB} вказує час самосинхронного циклу запису для відповідних пристроїв EPCS. Таким чином,

необхідно враховувати цю затримку перед записом іншої сторінки пам'яті. Крім того, ми можемо перевірити біт «write in progress» в регістрі статусу, виконавши операцію «read status» в процесі самосинхронного циклу запису. Біт «write in progress» встановлюється в 1 протягом самосинхронного циклу запису, і 0, якщо він завершився.

Байти в пам'яті послідовного конфігураційного пристрою необхідно стерти до 1 або 0xFF до реалізації операції «write byte». Це може бути досягнуто або за допомогою команди «erase sector» в секторі, або команди «erase bulk» по всій пам'яті.

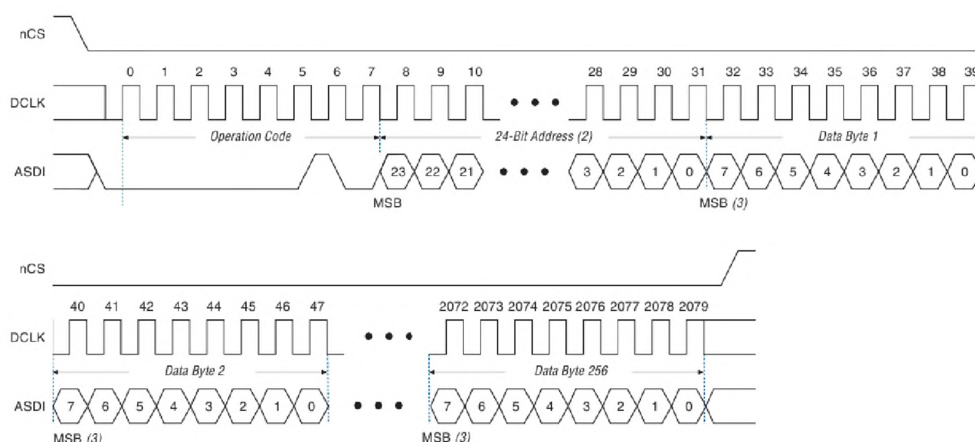

Рисунок 2.10 – Часова діаграма операції «write byte»

Примітка до рис. 2.10:

(1) Використовуйте інструкції «erase sector» або «erase bulk» щоб ініціалізувати байти пам'яті послідовного конфігураційного пристрою для всіх 1 або 0xFF перед реалізацією операції «write byte».

(2) Адресний біт [23] не враховується в EPCS64. Адресні біти [23 .. 21] не враховуються, в EPCS16. Біти адреси [23 .. 19] не враховується в EPCS4. Адресні біти [23 .. 17] не враховується в EPCS1.

(3) Для rpd файлів, LSB байту даних записується першим.

2.1.11 Операція «erase bulk»

Кодом операції «erase bulk» є b'1100 0111, з MSB першим. Операція «erase bulk» встановлює всі біти пам'яті на 1 або 0xFF. Як і в операції «write byte»,

операція «write enable» повинна бути виконана до операції «erase bulk», так що біт-засувка «write enable» в регістрі стану встановлюється в 1.

Ми можемо реалізувати операцію «erase bulk» при встановлені сигналу nCS на низький рівень, а потім змістивши код операції «erase bulk» на ASDI контакт. Сигнал nCS повинен стати високим після фіксації восьмого біту з коду операції «erase bulk». На рис. 2.11 показана часова діаграма.

Пристрій ініціює самосинхронний цикл «erase bulk» відразу ж після встановлення сигналу nCS у високий. В таблиці 2.6 t_{EB} вказує час самосинхронного циклу «erase bulk» для відповідних пристроїв EPCS.

Ми повинні враховувати цю затримку до доступу до вмісту пам'яті. Крім того, ми можемо перевірити «write in progress» біт в регістрі стану, виконавши операцію «read status» в той час як триває самосинхронний цикл стирання. Біт «write in progress» рівний 1 під час циклу самосинхронного стирання і 0, коли він завершений. Засувка «write in progress» в регістрі стану скидається в 0 перед завершенням циклу стирання.

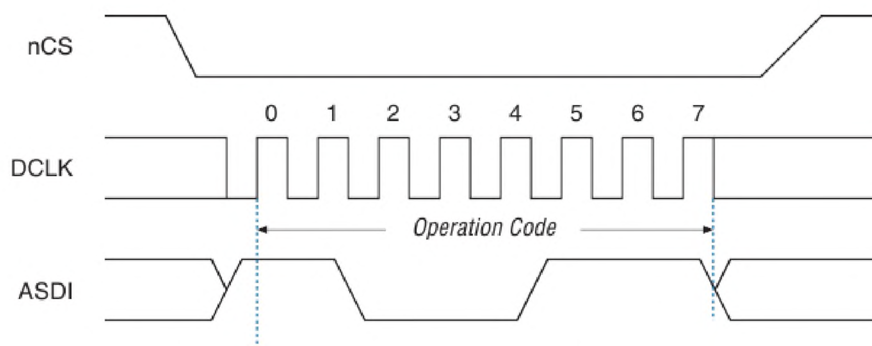

Рисунок 2.11 – Часова діаграма операції «erase bulk»

2.1.12 Операція «erase sector»

Кодом операції «erase sector» є послідовність b'1101 1000, з MSB першим. Операція «erase sector» дозволяє користувачеві видалити певний сектор в послідовному конфігураційному пристрої шляхом установки всіх бітів всередині сектора в 1 або 0xFF. Ця операція корисна для користувачів, які мають доступ до секторів, що не використовуються, як пам'яті загального призначення в своїх додатках.

Операція «write enable» повинна бути виконана до операції «erase sector» таким чином, щоб засувка-біт «write enable» в регістрі стану був встановлений в 1.

Операція «erase sector» реалізується першим падінням сигналу nCS, потім надходить код команди «erase sector» і три байти адреси вибраного сектору на контакт ASDI. Три байта адреси для операції «erase sector» може бути будь-якою адресою всередині зазначеного сектору. (див. таблицю 2.2 для інформації про діапазон адрес сектора.) Сигнал nCS встановлюється на високий рівень після фіксації восьмого біту коду операції «erase sector». На рис. 2.12 показана часова діаграма.

Відразу після встановлення сигналу nCS на високий, ініціюється самосинхронний цикл «erase sector». В таблиці 2.6 t_{ES} вказує час самосинхронного циклу «erase sector» для відповідних пристроїв EPC. Ми повинні враховувати цю затримку до доступу до вмісту пам'яті. Крім того, ми можемо перевірити біт «write in progress» в регістрі стану, виконавши операцію «read status» в процесі циклу стирання. Біт «write in progress» рівний 1 під час самосинхронного циклу стирання і 0, коли він завершений. Біт-засувка «write enable» в регістрі стану скидається на 0 до завершення циклу стирання.

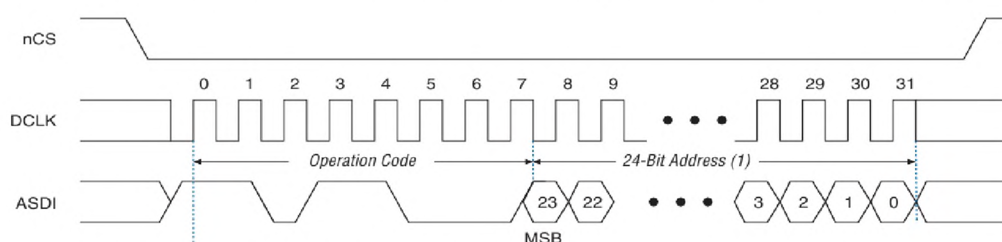

Рисунок 2.12 – Часова діаграма операції «erase sector»

Примітка до рис. 2.12:

Адресний біт [23] не враховується в EPCS64. Адресні біти [23 .. 21] не враховуються в EPCS16. Адресні біти [23 .. 19] не враховуються в EPCS4. Адресні біти [23 .. 17] не враховуються в EPCS1.

2.1.13 Інформація про часові діаграми

На рис. 2.13 показана форма сигналу для операції запису в послідовний конфігураційний пристрій.

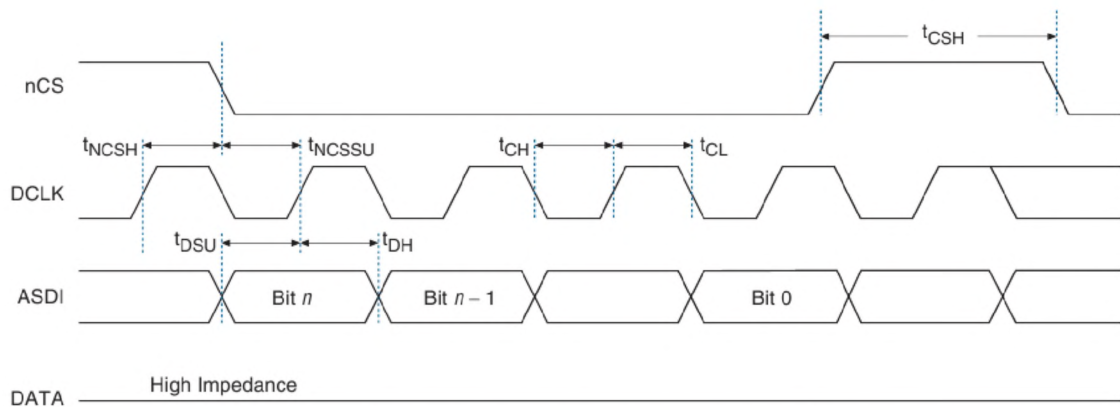

Рисунок 2.13 – Часова діаграма операції запису

У таблиці 2.6 визначені параметри синхронізації послідовного конфігураційного пристрою для операції запису.

Таблиця 2.6 –Параметри операцій запису

Символ	Параметр	Мін	Номінально	Макс	Одиниця виміру
t_{WCLK}	Тактова частота запису (від FPGA, кабелю завантаження або вбудованого процесора) для операцій write enable, write disable, read status, read silicon ID, write bytes, erase bulk, та erase sector	-	-	25	МГц
t_{CH}	Час високого рівня DCLK	20	-	-	нс
t_{CL}	Час низького рівня DCLK	20	-	-	нс
t_{NCSSU}	Час налаштування вибору чіпа (nCS)	10	-	-	нс
t_{NCSH}	Час затримки вибору чіпа (nCS)	10	-	-	нс
t_{DSU}	Час налаштування даних (ASDI) до переднього фронту DCLK	5	-	-	нс
t_{DH}	Час затримки даних (ASDI) після переднього фронту DCLK	5	-	-	нс
t_{CSH}	Час вибору чіпа	100	-	-	нс
t_{WB}	Час циклу Write bytes для EPCS1, EPCS4, EPCS16, та EPCS64	-	1,5	5	мс
	Час циклу Write bytes для EPCS128	-	2,5	7	мс
t_{WS}	Час циклу write status	-	5	15	мс
t_{EB}	Час циклу erase bulk для EPCS1	-	3	6	с
	Час циклу erase bulk для EPCS4	-	5	10	с
	Час циклу erase bulk для EPCS16	-	17	40	с
	Час циклу erase bulk для EPCS64	-	68	160	с
	Час циклу erase bulk для EPCS128	-	105	250	с
t_{ES}	Час циклу erase sector для EPCS1, EPCS4, EPCS16, та EPCS64	-	2	3	с
	Час циклу erase sector для EPCS128	-	2	6	с

На рис. 2.14 показано форму сигналу для операції читання з послідовного конфігураційного пристрою.

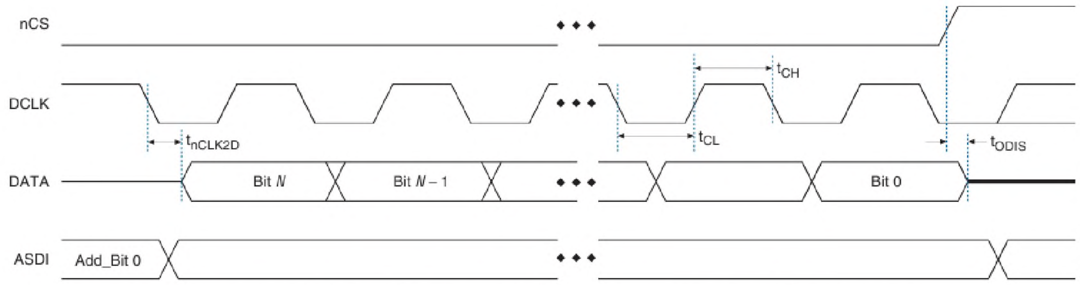

Рисунок 2.14 – Часова діаграма операції читання

У таблиці 2.7 визначено тимчасові параметри послідовного конфігураційного пристрою для операції читання.

Таблиця 2.7 – Параметри операції читання

Символ	Параметр	Мін	Макс	Одиниця виміру
t_{RCLK}	Тактова частота читання (з FPGA або вбудованого процесора) для операції read bytes	-	20	МГц
t_{CH}	Час високого рівня DCLK	25	-	нс
t_{CL}	Час низького рівня DCLK	25	-	нс
t_{ODIS}	Час відключення сигналу після читання	-	15	нс
t_{nCLK2D}	Час від задньоного фронту синхроімпульсу до надходження даних.	-	8	нс

На рис. 2.15 показано форму сигналу для схеми конфігурації FPGA AS за допомогою послідовного конфігураційного пристрою.

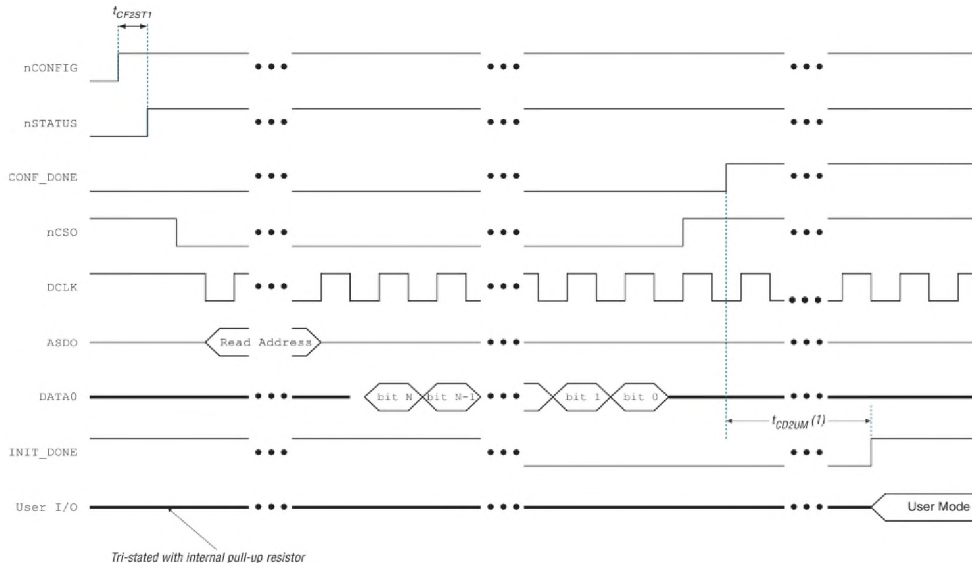

Рисунок 2.15 – Часова діаграма конфігурації AS

Примітка до рис. 2.15:

(1) t_{CD2UM} параметр що залежить від FPGA.

2.2 Функціональна схема (діаграма станів)

Конфігураційний пристрій EPCS1 має 8 контактів, з них чотири інформаційні контакти, тому для моделі будемо використовувати тільки чотири сигнали, дивитися таблицю 2.8.

Таблиця 2.8 – Опис інформаційних контактів послідовного конфігураційного пристрою

Ім’я контакту	Тип контакту	Опис
DATA	вихід	Вихідний сигнал DATA передає дані послідовно з конфігураційного пристрою в FPGA під час операції читання/конфігурації. Під час операції читання/конфігурації, конфігураційний пристрій активується встановленням сигналу nCS на низький. Сигнал DATA передається при спаді сигналу DCLK.
ASDI	вхід	Сигнал ASDI використовується для передачі даних послідовно в конфігураційний пристрій. Він отримує дані, які потрібно запрограмувати в конфігураційний пристрій. Дані фіксуються при встановленні сигналу DCLK на 1.
nCS	вхід	Цей сигнал вмикається на початку і зникає при отриманні правильної інструкції. Коли сигнал має високий рівень, він активує пристрій і переводить його в активний режим. Після включення живлення, конфігураційний пристрій потребує спадання сигналу nCS перед початком будь-якої операції.
DCLK	вхід	DCLK забезпечується ПЛІС. Цей сигнал забезпечує синхронізацію послідовного інтерфейсу. Дані представлені на ASDI фіксуються в послідовному конфігураційному пристрої при передньому фронті DCLK. Дані на контакті DATA змінюються при задньому фронті DCLK і фіксується в ПЛІС при наступному задньому фронті.

Використовуючи опис контактів можна скласти діаграму основних станів.

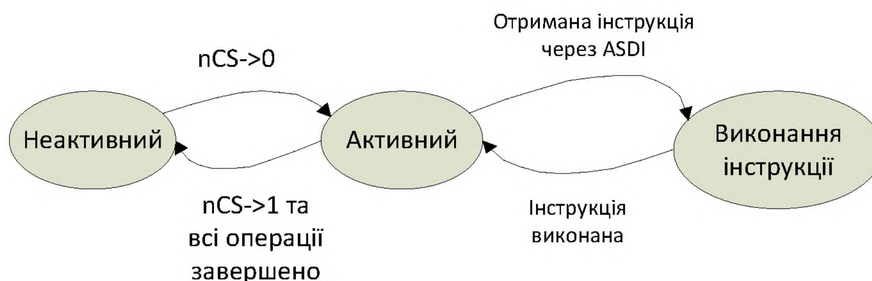

Рисунок. 2.16 – Діаграма основних станів конфігураційного пристрою

Висновки з розділу 2

У цьому розділі розглянуто структурну схему конфігураційної пам'яті, її контакти та їх призначення. Також описані основні операції, які вона підтримує, їх коди та умови виконання. Цього досить для опису пристрою на мові VHDL.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ КОНФІГУРАЦІЙНОЇ ПАМ’ЯТІ

3.1 Опис контактів та основних змінних

Для початку підключимо стандартну бібліотеку library ieee.

Використаємо наступні пакети:

- ieee.std_logic_1164 - пакет, що містить декларації, які підтримують єдине уявлення логічної величини в описі VHDL;
- ieee.numeric_std. – пакет, що місить арифметичні функції для векторів;
- std.env – містить функції для зупинки процесу симуляції.

Надалі вкажемо ім’я сутності:

```
entity EPCS1_vhd is
end EPCS1_vhd;
```

та опишемо її архітектуру, як опис поведінки схеми.

Опис архітектури найбільш об’ємна частина моделі, тому будемо робити його по частинам. Для початку визначимо які сигнали та змінні будуть нам потрібні. Сигнали DCLK, ASDI, DATA та nCS – як основні сигнали з якими працює пам’ять. Сигнал SR – восьми бітний статус-регістр, а ram – масив пам’яті. Конфігураційний пристрій що ми моделюємо має 131 072 біт пам’яті. По замовчуванню в неактивному режимі сигнал DATA має високий імпеданс, ASDI рівний логічному нулю, а nCS – логічній одиниці. Вміст статус-регістру та пам’яті обнуляється.

```
    architecture EPCS1_behavior of EPCS1_vhd is
    signal DCLK : std_logic;
    signal ASDI : std_logic := '0';
    signal DATA : std_logic := 'Z';
    signal nCS : std_logic :='1';
    signal SR : std_logic_vector(7 downto 0) := (others => '0');
    type ram is array (0 to 131072) of std_logic_vector(7 downto
0);
    signal ram_t : ram := (others => (others => '0'));
```

Основні функції пам’яті опишемо в процедурі під назвою EPCS1( ):

```
procedure EPCS1(
```

```
  signal i_dclk : in std_logic;
  signal i_ASDI : in std_logic;
  signal i_nCS : in std_logic;
  signal SR : inout std_logic_vector;
  signal ram : inout ram;
  signal o_DATA : out std_logic
) is
```

Наш пристрій має можливість блокувати сектори пам'яті для запису. Тому створимо тип даних ProtectRange і змінну Write_Protection типу ProtectRange. Цей тип матиме чотири стани: None, Quarter, Half, Full. Цим станам відповідають 3-ій та 2-ий біти (BP1 та BP0) в статус-регістрі SR, дивитися таблицю 2.4.

```
type ProtectRange is ( None, Quarter, Half, Full);
variable Write_Protection : ProtectRange;
```

Для роботи з кодами операцій, даними та адресами створимо змінні opcode, data, та addr відповідно, та типом даних std_logic_vector. Для коду операції та буферу даних достатньо 8 біт, а для адреси – 24 біти:

```
    variable opcode : std_logic_vector(7 downto 0);
    variable data : std_logic_vector(7 downto 0) := (others =>
'0');
    variable addr : std_logic_vector(23 downto 0) := (others =>
'0');
```

Щоб не заплутатися з байтами статус-регістру – надамо зрозумілі імена найбільш важливим з них. Згідно рис. 2.4 WEL – перший біт, BP0 – другий біт, BP1 – третій біт. Нумерація починається з нуля, а код виглядає так:

```
alias WEL : std_logic is SR(1);
alias BP0 : std_logic is SR(2);
alias BP1 : std_logic is SR(3);
```

Так як для моделі пам'яті EPCS1 максимальна адреса становить 1FFFF (табл. 2.2) то ми використовуватимемо тільки 17 біт:

```
    alias Current_Addr : std_logic_vector(17 downto 0) is addr(17
downto 0);
```

3.2 Основні операції

Перехід в активний стан, процес отримання коду операції та самі операції конфігураційного пристрою опишемо в тілі архітектури EPCS1_behavior.

Пристрій з неактивного режиму переходить в активний при встановленні сигналу nCS в логічний нуль:

```
 wait until i_nCS'event and i_nCS='0';
```

Далі запускаємо цикл зчитування коду операції з контакту i_ASDI при кожному передньому фронті сигналу синхроімпульсу i_dclk та записуємо в змінну opcode.

```
for l in 0 to 8 loop
  wait until i_dclk'event and i_dclk='1';
  opcode := opcode(6 downto 0) & i_ASDI;
end loop;
```

Перевіряємо отриманий код операції і в разі спів падання з відомим виконуємо відповідну операцію.

3.2.1 Операція «write enable»

Записуємо одиницю в 1-й біт статус-регістру (WEL):

```
if(opcode="00000110") then
  WEL <= '1';
  report "Status Register En";
end if;
```

Оператор report видає повідомлення для зручності перевірки роботи.

3.2.2 Операція «write disable»

Записуємо нуль в біт WEL статус-регістру:

```
if(opcode="00000100") then
  WEL <= '0';
  report "Status Register Dis";
end if;
```

3.2.3 Операція «write status»

Повністю переписуємо вміст статус-регістру даними, що послідовно подаються на контакт ASDI сигналом i_ASDI. Спочатку дані записуємо в буфер, потім при WEL = '1' записуємо в статус-регістр. В кожному колі циклу чекаємо наступного переднього фронту синхроімпульсу.

```
if(opcode="00000001") then
report "Write in register";
  for p in 0 to 8 loop
       wait until i_dclk'event and i_dclk='1';
       data := data(6 downto 0) & i_ASDI;
  end loop;
  if(WEL = '1') then
```

```
        SR <= data;
    end if;
  end if;
```

3.2.4 Операція «read status»

Послідовно видаємо вміст статус-регістру через сигнал o_DATA використовуючи цикл. В кожному колі циклу чекаємо наступного заднього фронту синхроімпульсу.

```
if(opcode="00000101") then
report "Read of register";
  for k in 7 downto 0 loop
       wait until i_dclk'event and i_dclk='0';
       o_DATA <= SR(k);
  end loop;
end if;
```

3.2.5 Операція «write bytes»

Після отримання коду операції «write bytes» запускаємо цикл отримання адреси та записуємо її в змінну addr при передньому фронті синхроімпульсу:

```
if(opcode="00000010") then
  for m in 0 to 23 loop
    wait until i_dclk'event and i_dclk='1';
    addr := addr(22 downto 0) & i_ASDI;
  end loop;
```

Також перевіряємо можливість запису в пам’ять. Зчитуємо біт WEL статус-регістру та порівнюємо його з логічною одиницею:

```
 if(WEL = '1') then
```

Надалі реалізуємо роботу функції захисту секторів пам’яті. Зчитуємо біти BP1 та BP2 статус-регістру та, в залежності від їх значень, присвоюємо змінній Write_Protection відповідне значення згідно табл. 2.4.

```
if(BP1='0' and BP0='0') then
  Write_Protection := None;
end if;
if(BP1='0' and BP0='1') then
  Write_Protection := Quarter;
end if;
if(BP1='1' and BP0='0') then
  Write_Protection := Half;
end if;
if(BP1='1' and BP0='1') then
  Write_Protection := Full;
end if;
```

Перевіряємо значення змінної Write_Protection. Відповідно до цього значення використовуємо для запису всю пам'ять, четверту частину, половину або відміняємо запис. Якщо всі умови для запису задоволені – створюємо цикл, в якому зчитуємо дані з сигналу i_ASDI і записуємо в буфер data. При оголошенні циклу, початкове значення і беремо зі змінної Current_Addr, що рівна першим 18 бітам адреси (змінна addr). А кінцеве значення i = 131072 для відсутнього захисту. Також реалізуємо можливість відміни процесу запису при встановленні високого сигналу nCS.

```
if( Write_Protection = None) then
  for i in to_integer(unsigned(Current_Addr)) to 131072 loop
    data := (others => '0');
    for j in 0 to 7 loop
      wait until i_dclk'event and i_dclk='1';
      data := data(6 downto 0) & i_ASDI;
    end loop;
    ram(i) <= data;
    exit when nCS='1';
  end loop;
end if;
```

При захищеній четвертій частині пам'яті кінцеве значення i знаходиться за формулою.

$$i = M - M/4,$$

де M – вся доступна пам'ять пристрою.

$$i = 131072 - \frac{131072}{4} = 98304$$

Тому умова має вигляд:

```
if( Write_Protection = Quarter) then
 for i in to_integer(unsigned(Current_Addr)) to 98304 loop
     -- код
  end loop;
end if;
```

При захищеній половині пам'яті:

$$i = 131072 - \frac{131072}{2} = 65536$$

А умова має вигляд:

```
if( Write_Protection = Half) then
 for i in to_integer(unsigned(Current_Addr)) to 65536 loop
     -- код
 end loop;
end if;
```

Опис операції запису закінчений. Не забуваємо закрити відкриті умови if.

3.2.6 Операція «read bytes»

Після отримання коду операції отримуємо з сигналу i_ASDI послідовно кожен з 24-ох бітів адреси при передньому фронті синхроімпульсу в змінну addr:

```
if(opcode="00000011") then
 for g in 0 to 23 loop
      wait until i_dclk'event and i_dclk='1';
      addr := addr(22 downto 0) & i_ASDI;
 end loop;
```

Потім, використовуючи отриману адресу, циклом зчитуємо дані з пам'яті (сигнал ram) і передаємо їх через сигнал o_DATA при задньому фронті синхроімпульсу:

```
 for i in to_integer(unsigned(Current_Addr)) to 131072 loop
      for k in 7 downto 0 loop
           wait until i_dclk'event and i_dclk='0';
           wait for 8 ns;
           o_DATA <= ram(i)(k);
      end loop;
```

Також реалізуємо можливість відміни операції високим рівнем сигналу nCS, закриваємо цикл та умову:

```
 exit when nCS='1';
 end loop;
end if;
```

3.2.7 Операція «erase bulk»

Перевіряємо код операції та дозвіл на запис:

```
if(opcode="11000111") then
          if(WEL = '1') then
```

Аналогічно з операцією «write bytes» зчитуємо біти BP1 та BP2 статус-регістру та, в залежності від їх значень, присвоюємо змінній Write_Protection відповідне значення:

```
if(BP1='0' and BP0='0') then
 Write_Protection := None;
end if;
 (BP1='0' and BP0='1') then
 Write_Protection := Quarter;
end if;
if(BP1='1' and BP0='0') then
 Write_Protection := Half;
end if;
if(BP1='1' and BP0='1') then
 Write_Protection := Full;
end if;
```

Перед початком видалення даних записуємо в нульовий біт статус-регістру (WIP) одиницю. Це дозволяє слідкувати за статусом виконання операції:

```
WIP := '1';
```

Відповідно до значення змінної Write_Protection встановлюємо всі біти доступної пам'яті в 1. Реалізуємо можливість виходу при nCS='1'. Встановлюємо затримку на 6 нс для завершення операції видалення.

```
if( Write_Protection = None) then
 for i in 0 to 131072 loop
     data := (others => '1');
     ram(i) <= data;
     exit when nCS='1';
 end loop;
 wait for 6 ns;
end if;
```

Аналогічний код пишемо і для інших значень Write_Protection.

```
if( Write_Protection = Quarter) then
 for i in 0 to 98304 loop
     …
if( Write_Protection = Half) then
 for i in 0 to 65536 loop
     …
```

В кінці операції видалення записуємо в WIP біт нуль:

```
WIP := '0';
```

3.2.8 Операція «erase sector»

Перевіряємо код операції, отримуємо значення адреси з i_ASDI та записуємо її в зміну addr:

```
if(opcode="11011000") then
     for m in 0 to 23 loop
          wait until i_dclk'event and i_dclk='1';
          addr := addr(22 downto 0) & i_ASDI;
     end loop;
```

Перевіряємо біт дозволу на запис:

```
if(WEL = '1') then
```

Якщо він рівний одиниці, то продовжуємо видалення. Аналогічно до операцій «erase bulk» та «write bytes» визначаємо значення Write_Protection та для кожної пишемо свій цикл видалення:

```
if( Write_Protection = None) then
 for i in to_integer(unsigned(Current_Addr)) to 131072 loop
     data := (others => '1');
```

```
          ram(i) <= data;
          exit when nCS='1';
     end loop;
     wait for 6 ns;
    end if;
```

Різниця між ними лише в максимальному і:

```
if( Write_Protection = Quarter) then
  for i in to_integer(unsigned(Current_Addr)) to 98304 loop
       …
if( Write_Protection = Half) then
  for i in to_integer(unsigned(Current_Addr)) to 65536 loop
       …
```

Не забуваємо для кожного процесу видалення поставити вихід при nCS='1', та затримку на 6 нс.

3.3 Опис середовища і процедур моделювання, верифікація

Як середовище моделювання використовуватимемо програму ModelSim версії 6.6d від Altera.

Потрібно навантажити вхідні контакти нашої моделі вхідними сигналами. Для генерації сигналу синхроімпульсу створимо процес clk_pr і встановимо потрібні нам затримки.

```
clk_pr : process
begin
 dclk <= '0';
 wait for 25 ns;
 dclk <= '1';
 wait for 25 ns;
end process clk_pr;
```

На контакт i_ASDI будемо подавати дані за допомогою процесу data_pr.

В змінній opcode ми вказуємо послідовність кодів операцій та даних що потрібні для цих операцій. У вказаному нижче прикладі ми надсилаємо коди операцій «write enable» (00000110) та «write status» (00000001), потім байт, даних (10011010) які потрібно записати команді «write status» в статус-регістр, далі пусті байти.

```
     data_pr : process
     variable opcode : data_for_send := ("00000110", "00000001",
"10011010", "00000000", "00000000");
```

```
     variable addr : std_logic_vector(23 downto 0) :=
"000000000000000000000000";
     begin
      for i in 0 to 4 loop
      wait until nCS'event and nCS='0';
      all_data_send <= '0';
      send_package(opcode(i), dclk, ASDI);
      if(i=3) then
           send_package(addr, dclk, ASDI);
           for j in 0 to 20 loop
                send_package(std_logic_vector(to_unsigned(j,8)),
dclk, ASDI);
           report "Finish Send data";
           end loop;
      end if;
      if(i=4) then
                     send_package(addr, dclk, ASDI);
                     for m in 0 to 160 loop
                     wait until dclk'event and dclk='0';
                     end loop;
      end if;
      wait until dclk'event and dclk='0';
      all_data_send <= '1';
      end loop;
      wait until dclk'event and dclk='1';
      wait until dclk'event and dclk='1';
      stop(0);
     end process;
```

Процедури send_data та send_package використовується для забезпечення послідовного зчитування даних з контакту i_ASDI при передньому фронті синхроімпульсу.

```
     procedure send_data(
      data : std_logic;
      signal i_dclk : in std_logic;
      signal i_ASDI : out std_logic
     ) is
     begin
      wait until i_dclk'event and i_dclk='1';
      i_ASDI <= data;
     end procedure;

     procedure send_package(
      data : std_logic_vector;
      signal i_dclk : in std_logic;
      signal i_ASDI : out std_logic
     ) is
     begin
      for i in data'length-1 downto 0 loop
           send_data(data(i),i_dclk,i_ASDI);
      end loop;
     end procedure;
```

Використовуючи процес S_pr ми робимо затримку в 50 нс на початку моделювання та керуємо вхідним сигналом nCS, що відповідає за перехід пристрою з пасивного режиму в активний.

```
S_pr : process
begin
  wait for 50 ns;
  nCS <= '0';
  wait until all_data_send'event and all_data_send='1';
  nCS <= '1';
 end process;
```

Змоделювавши наш пристрій ми отримали часову діаграму, дивитися рис.4.2. На ньому видно що на контакт ASDI при спаданні сигналу nCS надходять дані 00000110 за 450 нс. Сигнал nCS знову стає високим, дана послідовність розпізнається як операція «write enable» і WEL біт в статус-регістрі набуває значеня 1. При наступому спаданні сигналу nCS пристрій отримує код операції «write status» і починає зчитувати дані, які необхідно записати в статус-регістр. Звіряємо значення сигналу отримуваного на контакт ASDI та записаний в статус-регістр. Він співпадає. Отже наш пристрій може виконувати операції «write enable» та «write status». Щоб впевнитися чи відповідає отримана часова діаграма з часовою діаграмою реального пристрою порівнюємо її з рис. 2.2. Вони співпадають.

Перевіримо роботу операції читання з регістру (read status). Для цього запишемо в статус-регістр послідовність (10101010) та прочитаємо вміст регістру. Подамо на вхід ASDI послідовність бітів: "00000110", "00000001", "10101010", "00000101", "00000000". Наш конфігураційний, пристрій після отримання інструкцій, на виході DATA повинен видати послідовність "10101010". Проаналізувавши рис. 4.3 ми переконуємося що пристрій функціонує правильно.

Перевіримо операції запису в пам’ять та читання з пам’яті пристрою. Для цього на вхід ASDI подамо послідовність бітів: "00000110", "00000001", "10011010", "00000010", "00000011". Результат моделювання зображений на рис. 4.4, 4.5 та 4.6.

3.4. Результати моделювання

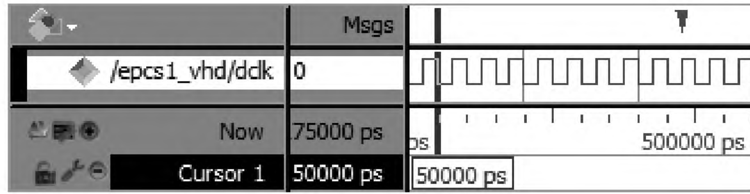

Рисунок 3.1 – Часова діаграма змодельованого сигналу синхроімпульсу

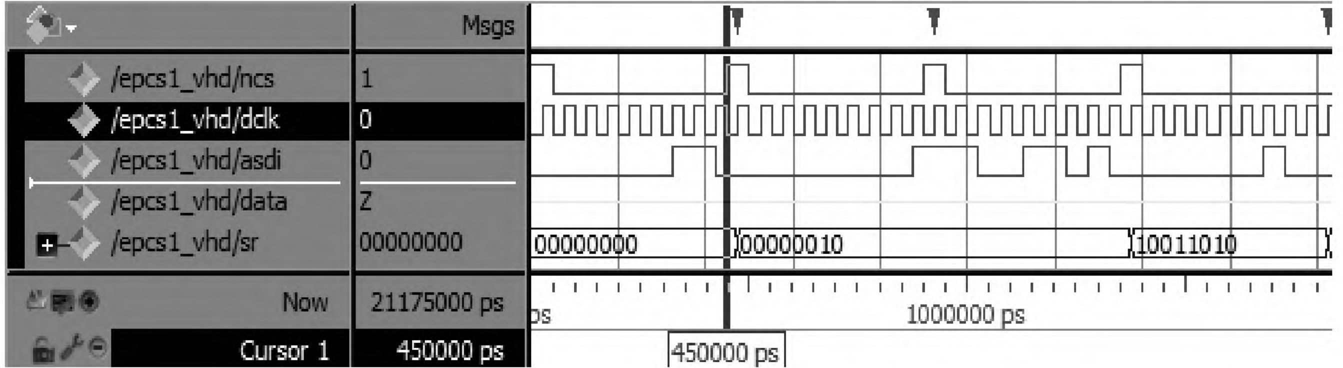

Рисунок 3.2 – Часова діаграма при операціях «write enable» та «write status»

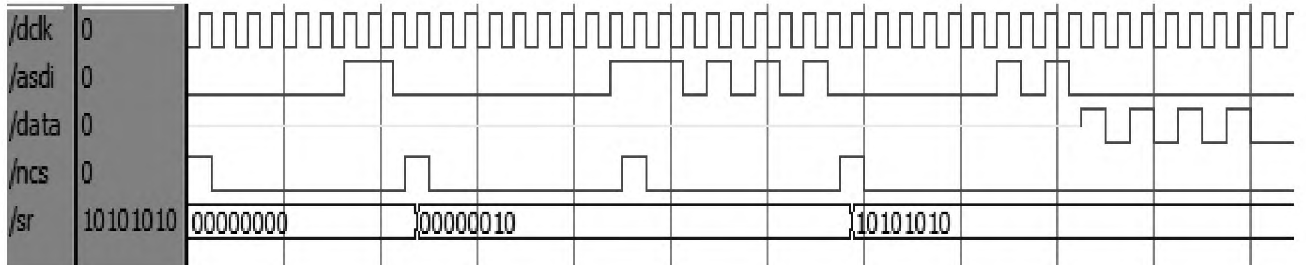

Рисунок 3.3 – Часова діаграма при запису і читанні статус-регістру

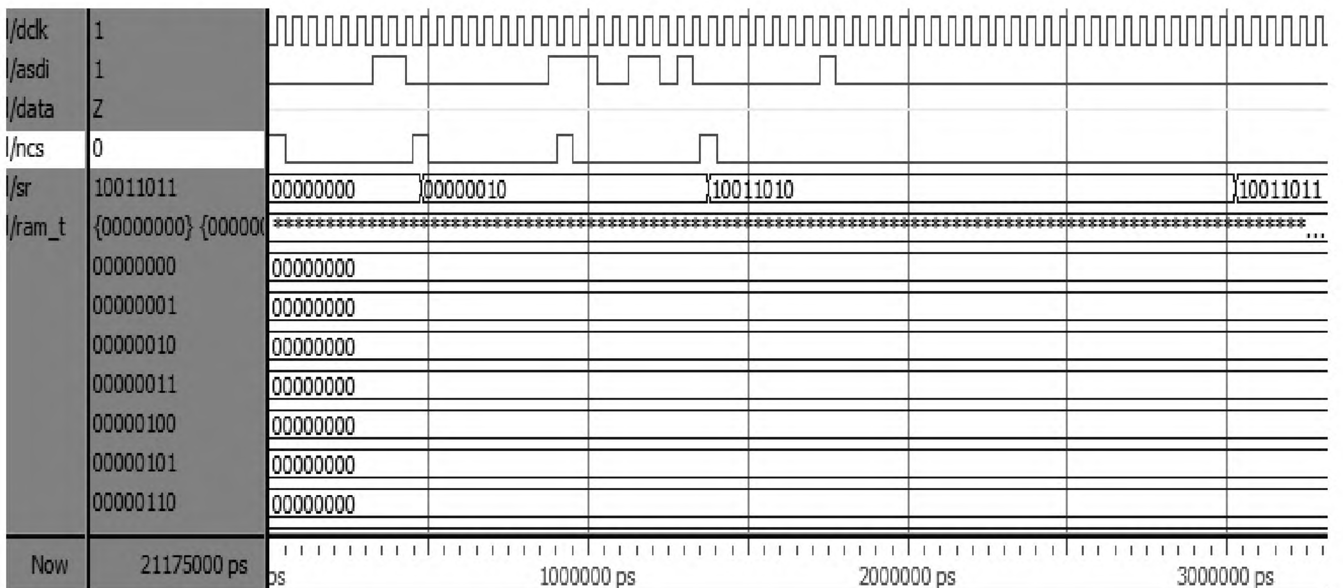

Рисунок 3.4 – Часова діаграма при операції «write enable» та «write bites»

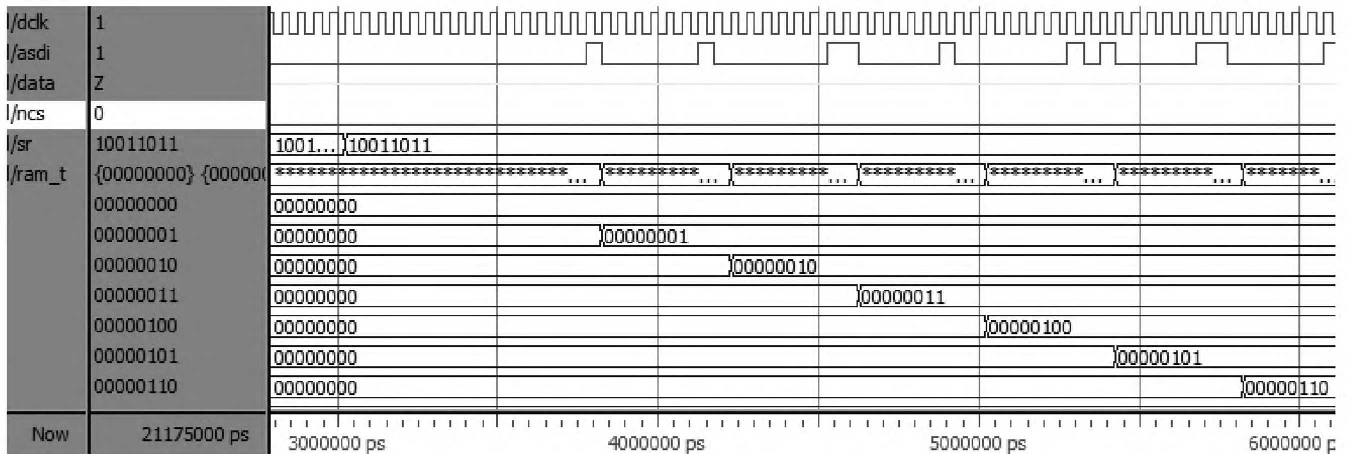

Рисунок 3.5 – Часова діаграма при операції «write bites»

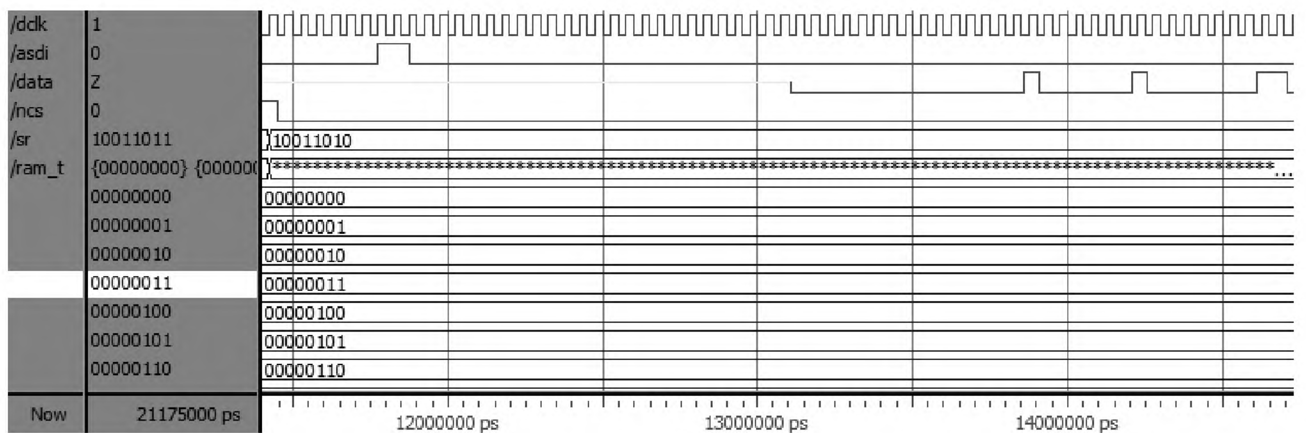

Рисунок 3.6 – Часова діаграма при операції «read bites»

Висновки до розділу 3

В цьому розділі описані середовище моделювання, основні процедури, операції над даними та процеси що дозволяють подавати сигнал на змодельовану пам’ять. Також розглянуті результати моделювання. Проаналізувавши отримані часові діаграми, можна сказати що ми створили модель пам’яті з параметрами що відповідають вимогам.

ВИСНОВКИ

У кваліфікаційній роботі виконані наступні роботи:

- зроблений аналіз літератури по архітектурі FPGA, проєктуванню та верифікації пристроїв на ПЛІС, засобам VHDL моделювання реальних об’єктів, дизайну цифрових систем використовуючи мову VHDL;
- досліджена структурна схема конфігураційної пам’яті EPCS1 фірми Altera, операції що підтримуються конфігураційною пам’яттю, кодами операцій та часовими діаграмами, технічний опис конфігураційних пристроїв EPCS1, EPCS4, EPCS16, EPCS64 та EPCS128;
- описані основні параметри та функції конфігураційної пам’яті EPCS1, результати моделювання в програмі ModelSim;

зроблено детальний опис моделей життєвого циклу, а саме: каскадний життєвий цикл, V-подібний життєвий цикл, спіральний життєвий цикл. Зроблено висновки та обрано V-подібний життєвий цикл. Зроблено висновок про те, 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;

- та розроблена верифікаційна модель конфігураційної пам’яті яка відрізняється від відомих підтримуваними операціями, величиною затримок та логікою роботи деяких функцій.

Згідно аналізу результатів моделювання, можна зробити висновок що нам вдалося зробити модель пам’яті що має такі ж параметри як і оригінал. Хоча вона і потребує подальшого доопрацювання. Зокрема винесення коду що повторюється в окрему функцію, доопрацювання різних ситуацій що виникають при поданні на вхідні контакти моделі невірних сигналів.