НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Радіотехнічний факультет

Кафедра прикладної радіоелектроніки

До захисту допущено:

В.о. зав.кафедри

Андрій МОВЧАНЮК

«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньою програмою «Інтелектуальні технології радіоелектронної техніки»

за спеціальністю 172 «Телекомунікації та радіотехніка»

на тему: «Дизайн цифрових схем вимірювання напруги для систем резервного живлення»

Виконав :

студент 2 курсу групи РЕ-21мп

Дерікот Глєб Юрійович

Прізвище, ім'я та по батькові підпис

Керівник:

Старший викл., Новосад Андрій Анатолійович

Посада, науковий ступінь, вчене звання, Прізвище, ім'я та по батькові підпис

Рецезент:

Доцент, к.т.н кафедри РТС

Шпилька Олександр Олександрович

Посада, науковий ступінь, вчене звання, Прізвище, ім'я та по батькові підпис

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент ___ __ Дерікот Г.Ю.

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського »

Радіотехнічний факультет

Кафедра прикладної радіоелектроніки

Рівень вищої освіти – другий (магістерський)

Спеціальність 172 “ Телекомунікації та радіотехніка ”

Освітньо-професійна програма «Інтелектуальні технології радіоелектронної техніки»

ЗАТВЕРДЖУЮ

В.о.зав. кафедри

Андрій МОВЧАНЮК

«___»_____20__ р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Дерікота Глєба Юрійовича

Тема дисертації: «Дизайн цифрових схем вимірювання напруги для систем резервного живлення»

Науковий керівник дисертації Новосад Андрій Анатолійович, затверджені наказом по університету від «09» листопада 2023 р. №5206-с.

2. Термін подання студентом дисертації 11 січня 2024 року

3. Вихідні дані до проекту :

кількість вимірюваних сигналів – 13;

частота дискретизації – не менше 11 кГц для кожного каналу.

вибірка – не менше 220 значень за період сигналу

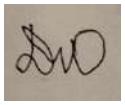
4. Зміст пояснювальної записки: Вступ, Аналіз існуючих підходів до вимірювання напруг у системах резервного живлення, Опис запропонованої

архітектури системи вимірювання, Розробка цифрового дизайну, Моделювання спроектованого цифрового дизайну, Висновок.

5. Дата видачі завдання 09 листопада 2023 року

6. Календарний план

№ з/п	Назва етапів виконання	Термін виконанняетапів проекту
1	Огляд існуючих підходів	9.11.23 – 13.11.23
2	Розробка та аналіз технічного завдання	14.11.23 – 18.11.23
3	Створення архітектури системи вимірювання	19.11.23 – 30.11.23
4	Розробка цифрового дизайну	31.11.23 – 18.12.23
7	Моделювання цифрового дизайну	18.12.23 – 25.12.23
8	Оформлення текстової документації	25.12.23 – 31.12.23

Студент: Глєб ДЕРІКОТ 

Керівник: Андрій НОВОСАД 

РЕФЕРАТ

Магістерська дисертація складається з пояснювальної записки обсягом 53 сторінки, що містять 39 рисунків, 5 додатків та 15 посилань.

В роботі за попередньо визначеною методикою проведена розробка інноваційного цифрового функціонального вузла в мікроелектронному проектуванні, призначеного для ефективного процесу вимірювання напруги в системах резервного живлення. Особливість цього вузла полягає в його здатності не тільки точно вимірювати напругу на підключених лініях, але й в реалізації механізму сигналізації для виявлення будь-яких аномалій або проблемних змін у цих сигналах.

Робота охоплює аналіз рішень, які застосовується у системах резервного живлення, на основі цього аналізу була розроблена власна архітектура цифрової системи вимірювання напруги, яка забезпечує високу точність та надійність вимірювань. Також було розроблено методику створення цифрового дизайну, що включає в себе розробку алгоритмів, логічних блоків та інтерфейсів, з подальшою перевіркою працездатності за допомогою моделювання. Етап перевірки дозволив не тільки підтвердити правильність функціонування розробленої системи, але й оцінити її ефективність та готовність до впровадження в реальні умови експлуатації систем резервного живлення.

Ключові слова: цифровий дизайн, схема вимірювання напруги, моделювання цифрового дизайну, системи резервного живлення, Verilog, RTL.

SUMMARY

The master's thesis comprises an explanatory note of 53 pages, containing 39 figures, 5 appendices, and 15 references.

The work involves the development of an innovative digital functional node in microelectronic design, intended for the efficient process of measuring voltage in backup power systems. The uniqueness of this node lies in its ability not only to precisely measure the voltage on connected lines but also in implementing a signaling mechanism to detect any anomalies or problematic changes in these signals.

The thesis covers an analysis of solutions used in backup power systems, and based on this analysis, a proprietary architecture of the digital voltage measurement system was developed, ensuring high accuracy and reliability of measurements. A methodology for creating digital designs was also developed, including the development of algorithms, logical blocks, and interfaces, with subsequent verification of functionality through modeling. The verification stage allowed not only to confirm the correctness of the system's operation but also to assess its efficiency and readiness for implementation in real conditions of backup power systems operation.

Keywords: Digital design, voltage measurement circuit, digital design simulation, backup power systems, Verilog, RTL.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до магістерської дисертації

на тему: «Дизайн цифрових схем вимірювання напруги для систем резервного живлення»

Київ – 2024 року

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

UPS - uninterruptible power supply

АЦП – аналого-цифровий перетворювач

ЛВ – логіка вимірювань

ЛП – логіка перевірки

MSQR – measurement sequencer

RTL – register transfer lever

HDL – hardware description language

Soc – start of conversion

EOA – end of acquisition

EOC – end of conversion

FSM – finite state machine

UVM – universal verification methodology

UVC – universal verification component

ВСТУП

Актуальність теми: Інтелектуальні системи часто містять мікроконтролер, який керує багатьма елементами системи, на основі її даних. Зокрема, важливим аспектом є вимірювання напруги різних елементів системи, кількість яких може бути значною. Розробка цифрового функціонального вузла в мікроелектронному виконанні, що дозволяє мікроконтролеру делегувати задачу вимірювання та перевірку виміряних даних на високошвидкісну цифрову схему, може значно знизити навантаження на мікроконтролер. Підхід оптимізації процесу вимірювання за рахунок використання спеціалізованих цифрових схем дозволяє підвищити ефективність, швидкість та надійність роботи інтелектуальних систем, особливо в умовах необхідності вимірювати велику кількість напруг. Враховуючи іноваційність таких рішень необхідно приділяти увагу методиці проектування подібних схем.

Мета дослідження: Розробка методів для створення цифрового дизайну, що дозволяють розробляти цифрові мікроелектронні функціональні вузли. Зокрема ставиться мета спроектувати цифрову схему вимірювання напруг для системи резервного живлення, що дозволяє мікроконтролеру делегувати задачу вимірювання напруги на дану цифрову схему. Це включає розробку архітектури системи вимірювання та її подальшого моделювання за допомогою цифрового симулятора від Cadence.

Об’єкт дослідження: методика та методи проектування і моделювання цифрових схем.

Предмет дослідження: Застосування інноваційних методик та методів проектування цифрових схем до розробки архітектура цифрових пристроїв вимірювання напруг для систем резервного живлення в мікроелектронному виконанні

Практичне значення одержаних результатів: Практичне застосування розроблених методів та схем дозволить значно знизити

навантаження на мікроконтролери, підвищити точність та швидкість вимірювання напруг, що є критично важливим для систем, які працюють у реальному часі.

1 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО ВИМІРЮВАННЯ НАПРУГ У СИСТЕМАХ РЕЗЕРВНОГО ЖИВЛЕННЯ.

Важливим аспектом реалізації алгоритмічної логіки в системах реального часу, побудованих на базі мікроконтролерів, є необхідність забезпечення швидкодії роботи. У багатьох випадках, алгоритми засновані на аналізі параметрів системи і, виходячи з отриманих даних, ініціюють визначені дії.

Однією з ключових величин, що підлягають аналізу, є напруга на входах/виходах та внутрішніх елементах системи. З огляду на необхідність оперативного оновлення цих значень, мікропроцесори виконують безперервне оновлення даних, що призводить до збільшення навантаження та зайнятості процесорного часу. Це, у свою чергу, може викликати затримки у роботі системи, обмежити можливості виконання алгоритму або навіть унеможливити його реалізацію.

Перше рішення полягає у безпосередньому підключенні напруг до аналогових входів мікроконтролера. Як приклад розглянемо принципову схему UPS "N-Power SVP-625", такими напругами є :

1. Напруга з сенсору вихідної потужності, аналоговий вхід AIC3.
2. Напруга з сенсору вхідної напруги, аналоговий вхід AIC0.
3. Напруга з сенсору вихідної напруги, аналоговий вхід AIC4.
4. Напруга з сенсору заряду акумулятора, аналоговий вхід AIC2.
5. Напруга з сенсору температури всередині корпусу, аналоговий вхід AIC1.

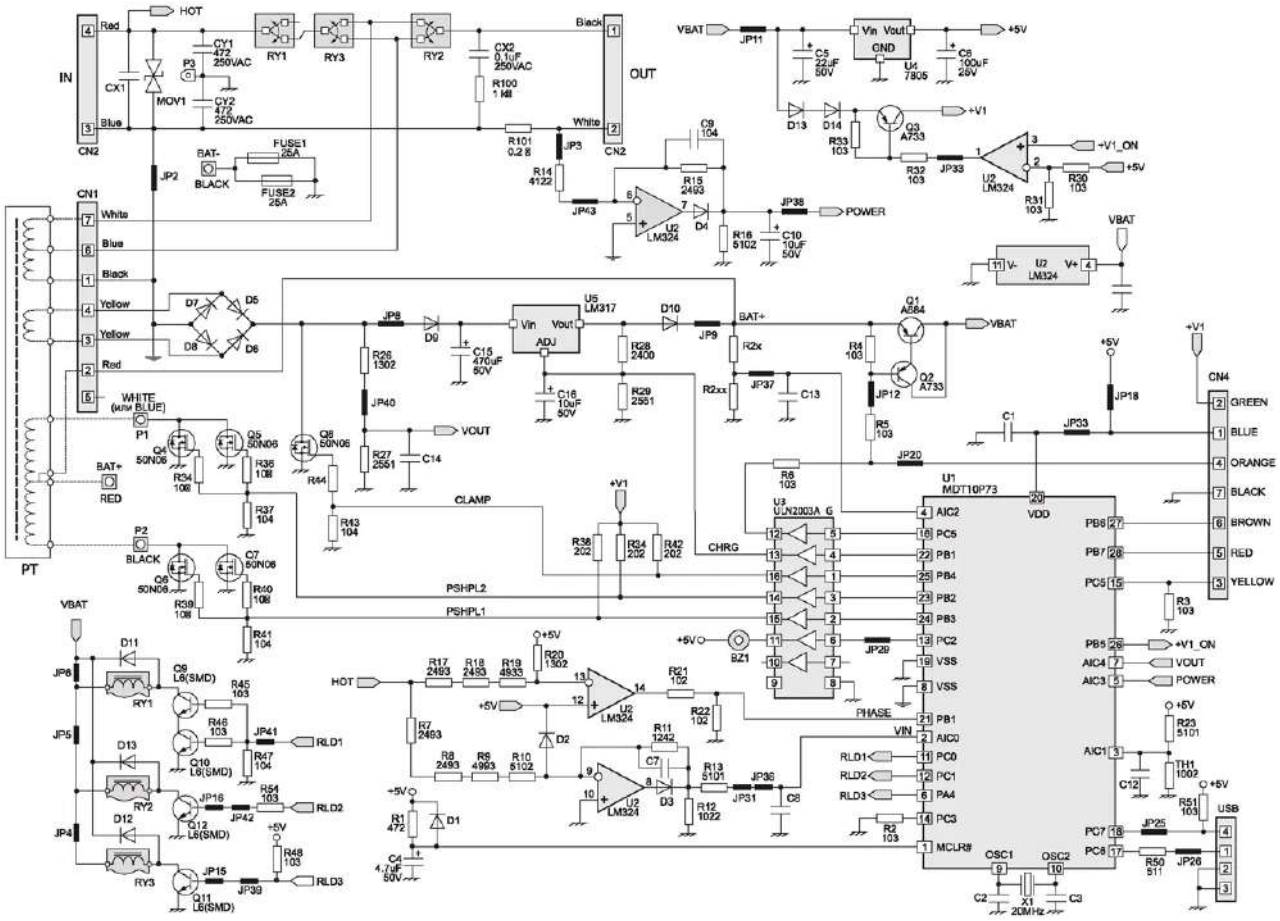


Рисунок 1.1 - принципова схема UPS N-Power SVP-625 [1]

Такий підхід вимагає, щоб мікропроцесор активно керував процесом вимірювання, що означає постійну взаємодію з внутрішнім аналого-цифровим перетворювачем та регулярне оновлення даних для кожного входу. Це забезпечує прямий контроль та можливість швидкого відгуку на зміни у параметрах напруги, але також ставить перед процесором великий обсяг задач, що може призвести до його перевантаження. Також при значній кількості аналогових сигналів - мікроконтролер має мати значну кількість аналогових входів.

Альтернативним методом є застосування зовнішніх АЦП, які мають власний інтерфейс для взаємодії з мікроконтролером. Такий підхід можна проілюструвати на прикладі UPS від компанії APC.

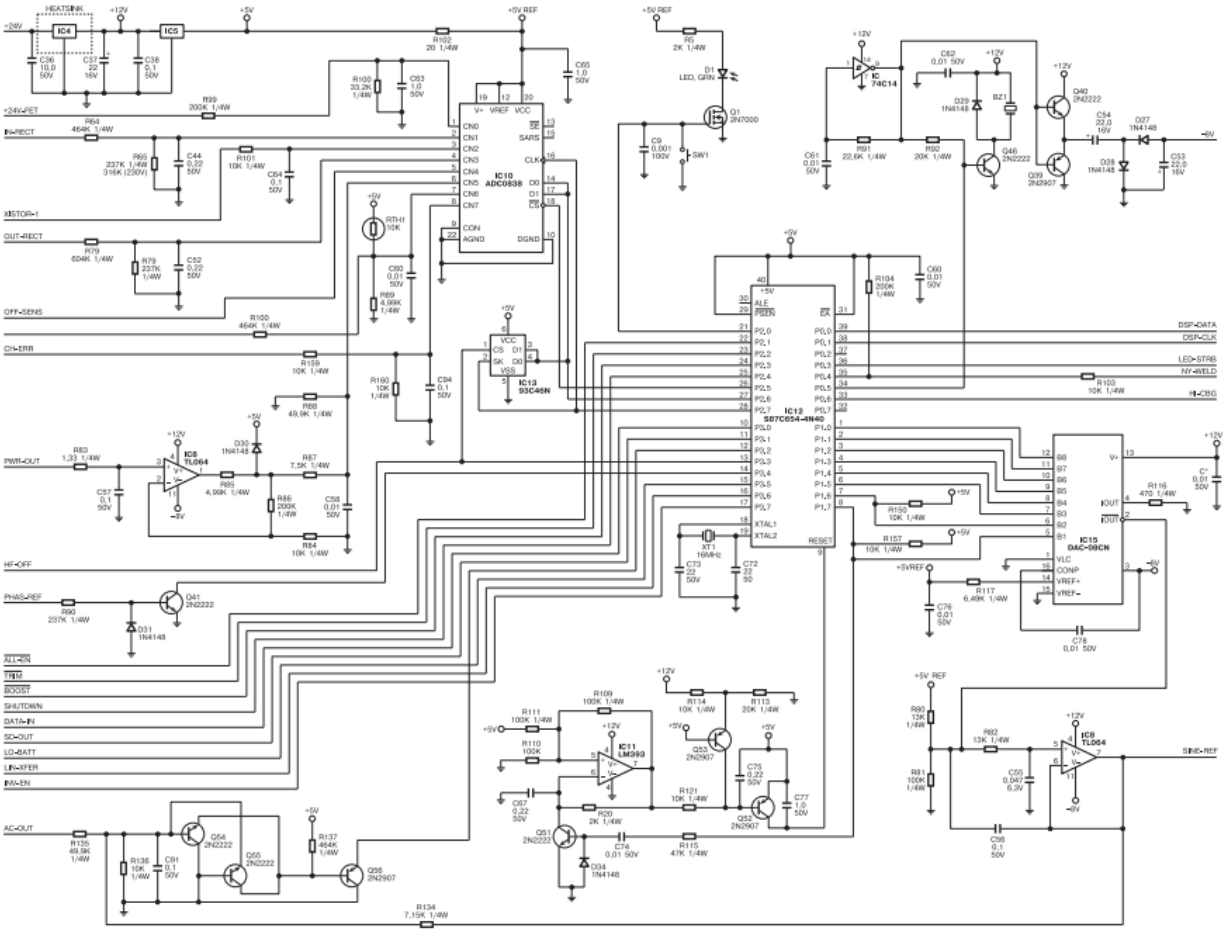


Рисунок 1.2 – принципова схема UPS від компанії APC 2 [2]

В цій системі до мікроконтролера підключений периферійний модуль - ADC0838, АЦП з послідовним інтерфейсом, що дозволяє підключити до нього декілька сигналів та обробляти їх завдяки вбудованому мультиплексору. У цьому сценарії, замість безпосереднього взаємодії з кожним аналоговим сигналом, мікроконтролеру потрібно тільки надсилати команди на зовнішній АЦП і потім зчитувати вимірянні данні.

Обговорені вище підходи виявляють дві різні стратегії організації процесу вимірювання напруги, проте в обох випадках мікроконтролеру потрібно постійно витрачати свій час та обчислювальні ресурси на виконання і аналіз вимірювань.

Розв'язанням цієї проблеми може стати розробка спеціалізованої інтегральної схеми, яка була б відповідальна за автономне вимірювання

підключених напруг та володіла б механізмом оповіщення у випадку виявлення проблемних змін у рівнях напруги. Таким чином, основна відповідальність за моніторинг і оцінку стану сигналів перекладається на інтегральну схему. Ця схема, у свою чергу, активує сигнал сповіщення для мікроконтролера, коли необхідне його безпосереднє втручання.

Таке рішення не тільки звільняє процесорні ресурси мікроконтролера для інших завдань, але й забезпечує можливість здійснення більш частого зчитування даних, що підвищує точність вимірювань. Крім того, завдяки автономності схеми, збільшується кількість сигналів, які можуть бути виміряні одночасно, без залучення додаткових ресурсів мікроконтролера.

Постановка задачі: В даній роботі потрібно запропонувати методику дизайну цифрових схем та спроектувати систему вимірювання напруг для систем резервного перевірити її на відповідність функціональним вимогам.

2 МЕТОДИКА РОЗРОБКИ ЦИФРОВОГО ДИЗАЙНУ

2.1 Етапи розробки цифрового дизайну

Розробка цифрового дизайну є процесом, що передбачає структурований та стандартизований підхід, який включає в себе ряд базових етапів (дивись рисунок 2.1).

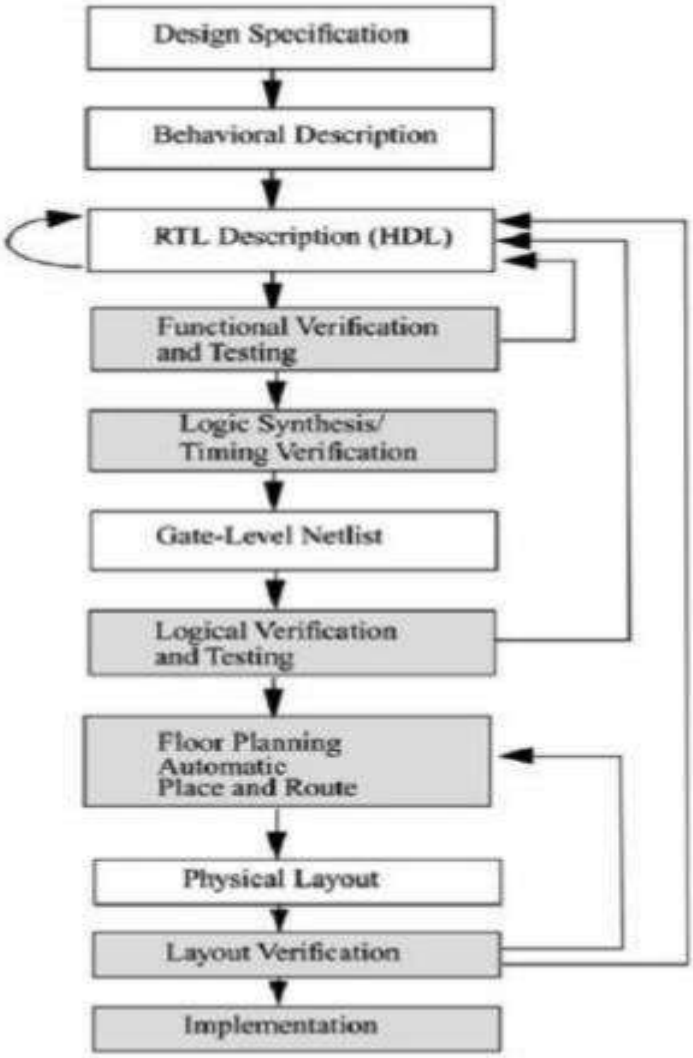


Рисунок 2.1 – Стандартизований підхід розробки цифрового дизайну[5]

Початковий етап проектування полягає у формуванні концепції, де визначаються головні цілі проекту, його функціональність та основні вимоги. Цей етап є критично важливим, оскільки він закладає фундамент для всієї подальшої роботи.

Завершивши етап концептуалізації, наступає фаза розробки архітектури проекту та його специфікації. Ця фаза включає детальне планування структури системи, визначення алгоритмів її роботи, а також розробку інтерфейсів між різними компонентами системи.

Після цього розпочинається безпосередньо розробка дизайну. Основним інструментом у цьому процесі є мова опису апаратури Verilog, яка вирізняється своїм зручним синтаксисом та можливістю опису на різних рівнях абстракції: від поведінкового (behavioural) до рівня вентилів (gate level) та рівня регістрів (RTL).

Gate Level - це рівень опису в HDL, де поведінка системи визначається через логічні вентилі, такі як І-НЕ, АБО-НЕ. Це дозволяє детально контролювати реалізацію на нижчому рівні, забезпечуючи точний контроль над тим, як функціонують окремі компоненти схеми. Приклади такого роду дизайну можна побачити на рисунках, де представлено специфічну конфігурацію вентилів для певної функції.

Behavioural Description - це вищий рівень абстракції в HDL, де поведінка системи описується без безпосереднього вказування на конкретну апаратну реалізацію. Це дозволяє описати складні алгоритми та функціональні блоки в більш зрозумілій та лаконічній формі. Під час симуляції, цей опис розгортається у детальніші моделі, які імітують реальну поведінку схеми.

Register Transfer Level (RTL) - це рівень, на якому описується перенесення даних між регістрами. На цьому рівні абстракції фокусується на потоках даних та на операціях, що здійснюються з цими даними. RTL дозволяє моделювати поведінку системи на більш високому рівні, не вдаючись до деталей реалізації на рівні вентилів.

Важливою перевагою використання Verilog є можливість симуляції та верифікації написаного коду до стадії фізичного втілення у вигляді інтегральної схеми. Це дозволяє виявляти та усувати потенційні помилки на

ранніх етапах, значно знижуючи ризики та витрати, пов'язані з виробництвом апаратури.

Після завершення процесу написання і верифікації коду на Verilog, RTL модель синтезується у Netlist. Він є ключовим елементом у процесі реалізації цифрового дизайну, представляючи детальний опис схеми. Він включає в себе всі необхідні компоненти, такі як логічні вентилі та регістри, а також точне визначення їх з'єднань. [6]

Рисунок 2.2 – Зв’язок коду Verilog з інтегральною схемою [7]

2.2 Синтаксис мови Verilog

Мова Verilog містить синтаксичні елементи, які є загальними для багатьох мов програмування, включаючи такі конструкції, як if/else, case, інкрементори, декрементори, логічні операції, а також операції порівняння. Ці елементи забезпечують зручність та зрозумілість при написанні коду, роблячи Verilog доступним для розробників з досвідом у традиційному програмуванні. Водночас, Verilog включає унікальні синтаксичні конструкції, які необхідно розуміти для коректного опису апаратного забезпечення.

До таких конструкцій належать:

1. Типи даних;
2. Блоки опису;
3. Модуль (module).

В мові Verilog єдва основні типи даних необхідних для опису дизайнуце *wire* та *reg*. Ці типи даних використовуються для представлення різних форм сигналів і мають свої унікальні характеристики та застосування.

Тип *wire* використовується для представлення з'єднань у схемі та передачі сигналів. Сигнал *wire* має комбінаторну природу, тобто його значення визначається безпосередньо з'єднаними з ним виразами (елементами). Цей тип даних використовується для представлення вихідних сигналів комбінаторних логічних елементів або для з'єднання модулів.

На відміну від *wire*, тип *reg* використовується для зберігання значень сигналів. Ці значення можуть змінюватися в часі відповідно до логіки, описаної в блоках always або initial.

Тип *reg* не обов'язково представляє фізичний регістр у схемі, але використовується для моделювання поведінки, яка залежить від попередніх станів або часу.

Для регулювання розміру шин даних у Verilog використовується конструкція опису розміру [size-1 : 0], де size вказує кількість бітів у шині (дивись рисунок 2.3).

```
reg         ShortRegister;  // Опис однобітного регістру
wire        ShortWire;      // Опис однобітного проводу
reg  [7:0]  WidthRegister;  // Опис 8-бітний регістр
wire [15:0] WidthWire;      // Опис 16-бітний провід
```

Рисунок 2.3 – Декларування сигналів

У мові Verilog, блоки *always* та *assign* є основними засобами для опису апаратної частини цифрових схем. Кожен з цих блоків має своє специфічне призначення і спосіб використання.

Always Block використовується для опису поведінки, яка залежить від змін у певних сигналах або станах. Він може бути використаний для синхронної та асинхронної логіки.

Для асинхронної логіки він використовує чутливий список для визначення, які сигнали викликають активізацію блоку.

Наприклад:

always @(sig1 or sig2) – означає, що блок реагує на зміни в sig1 або sig2;

lways @(*) – дозволяє блоку реагувати на зміну будь-якого сигналу, що використовується всередині блоку.

```
always @(sig1 or sig2) begin
    block_output = sin1 || sig2; // Логіка, що виконується при зміні sig1 або sig2
end

always @(*) begin
    block_output = sin1 || sig2 || sig3; // Логіка, що виконується при зміні sig1 або sig2 або sig3
end
```

Рисунок 2.4 – Опис асинхронних always блоків

У синхронній логіці *always block* часто використовує годинниковий сигнал (clock) і сигнал скидання (reset). Типовий синтаксис включає умову скидання та поведінку при кожному годинниковому імпульсі.

Також особливістю опису синхронної логіки є неблокуюче присвоювання (non-blocking assignments), позначається, як “<=” . Особливість полягає у здатності дозволити одночасне оновлення декількох регістрів без взаємного впливу одного присвоєння на інше протягом одного такту годинника.

Використання неблокуючих присвоєнь дозволяє уникнути таких проблем, як гонитва даних (race conditions) і проблеми з часуванням, оскільки воно забезпечує більш передбачувану та стабільну поведінку схеми.

```
reg [3:0] cnt;

always @(posedge clk or negedge rstb) begin
    if (!rstb) begin
        // Логіка скидання
        cnt <= 0;
    end else begin
        // Синхронна логіка, що виконується на кожному годинниковому імпульсі
        cnt <= cnt + 1;
    end
end
```

Рисунок 3.5 – Опис синхронного always блоку

Ключове слово *assign* використовується для опису комбінаційної логіки, де вихідний сигнал безпосередньо залежить від входів. Це використовується для простих логічних відношень або для опису постійних з'єднань.

```
assign oupt = inp_a && inp_b; // вихід oupt підключається до комбінаторної логіки, яка робить операцію логічне І між двома сигналами
```

Рисунок 2.6 – Опис комбінаторної логіки за допомогою assign

Блок *module* у мові Verilog є фундаментальним елементом, який служить як основний блок моделювання для будь-якої цифрової схеми. Він виконує роль контейнера, який об'єднує як опис інтерфейсу, так і опис апаратної частини.

Інтерфейс модуля визначає - як модуль взаємодіє з зовнішнім світом. Він включає в себе опис входів та виходів модуля.

Для опису інтерфейсу Verilog має такі ключові слова :

— ключове слово *input* використовується для опису сигналів, що приходять у модуль;

— ключове слово *output* використовується для сигналів, що виходять з модуля.

Окрім того, в інтерфейсі можна використовувати параметри (ключове слово parameter), які дозволяють змінювати певні аспекти поведінки модуля без зміни його коду. Параметри можуть бути використані для налаштування

таких характеристик, як розміри шин даних, таймінги або логічні умови, забезпечуючи гнучкість та масштабованість дизайну.

Апаратна частина описує логічну поведінку модуля та його внутрішню структуру. Це включає в себе визначення внутрішніх змінних, таких як *reg* та *wire*, і опис логічних операцій та поведінки, які використовують ці змінні. [8,9]

```
module TestModule #(parameter cnt_width = 10)(input clk,
                                               input rstb,
                                               output reg [cnt_width - 1:0] cnt_output);

    reg [cnt_width-1:0] counter;

    always @(posedge clk or negedge reset) begin
        if (!rstb)
            counter <= 0;
        else
            counter <= counter + 1;
    end

    assign cnt_output = internalReg;

endmodule
```

Рисунок 2.7 - Лічильник з асинхронним сигналом скидання

Рисунок 2.7 є прикладом реалізації лічильника з асинхронним сигналом скидання, де :

1. TestModule – назва модулю, яка буде необхідна для інстанціаціїї модулів даного типу.
2. Parameter cnt_width – параметр, який конфігурує розмір шини нашого лічильнику, має значення 10 при замовчуванні.
3. Інтерфейс з описом входів і виходів.
4. Опис внутрішнього регістру з назвою counter та розміром шини який задається параметром.
5. Always block - опис логіки синхронного лічильника з асинхронним скиданням.
6. Assign – опис підключення внутрішнього регістру до виходу.

Отже можна зробити висновок, що мова Verilog має всі необхідні засоби, зручний та зрозумілий синтаксис та дозволяє легко описувати необхідну апаратну частину цифрових схем (створювати дизайн).

3 ОПИС ЗАПРОПОНОВАНОЇ АРХІТЕКТУРИ СИСТЕМИ ВИМІРЮВАННЯ ТА СТВОРЕННЯ ДИЗАЙНУ

3.1 Обґрунтування параметрів системи

Обґрунтуємо працездатність запропонованого у минулому розділі рішення, на прикладі інтегральної схеми задача якої базується на вимірюванні 13 різних напруг.

У якості частоти тактування інтегральної схеми оберемо значення у 32 МГц, як найбільш поширене значення серед інтегральних схем. Ретельний аналіз доступних на ринку аналого-цифрових перетворювачів та мультиплексорів дозволяє зробити висновок, що типове значення часу встановлення вихідного сигналу мультиплексора, не перевищує відмітку 500нс, а типове значення часу одного перетворення сигналу за допомогою аналого-цифрового перетворювача (АЦП) знаходиться на рівні в 1мкс. Для розрахунку сценарію вимірювань, оберемо часові параметри, які в декілька разів перевищують їх типові значення[3,4]:

1. Час конвертації АЦП – 3мкс;
2. Час виставлення сигналу мультиплексору – 3мкс;

Період вимірювання одного сигналу розраховується за формулою :

$$T_{min} = N * (t_{adc} + t_{mux}) = 13 * (3 + 3) = 78 \text{ [мкс]} ,$$

де N – кількість значень, які необхідно виміряти,

t_{adc} – час конвертації АЦП,

t_{mux}– час виставлення сигналу мультиплексора.

Розрахуємо частоту дискретизації кожного сигналу за формулою:

$$f_s = \frac{1}{T_u} = \frac{1}{78 * 10^{-6}} = 12820{,}5 \text{ [кГц]} ,$$

Де T_u- період вимірювання одного сигналу.

Стандартизований сигнал напруги живлення має період в 20мс, отже розрахуємо кількість відліків одного сигналу отриману за цей період :

$$N_s = \frac{T_u}{t_{min}} = \frac{20 \cdot 10^{-3}}{78 \cdot 10^{-6}} = 256\,,$$

де T_u – період виміру,

T_{min}– період вимірювання одного сигналу.

Отримані значення підтверджують, що система відповідає поставленим вимогам до системи вимірювання для системи резервного живлення, а саме здійснювати достатню кількість вимірів для правильного функціонування системи.

3.2 Архітектура схеми вимірювання

Відповідно до поставленої задачі у Розділі 1 та обґрунтуванню можна запропонувати архітектуру схеми вимірювання (див рисунок 3.1).

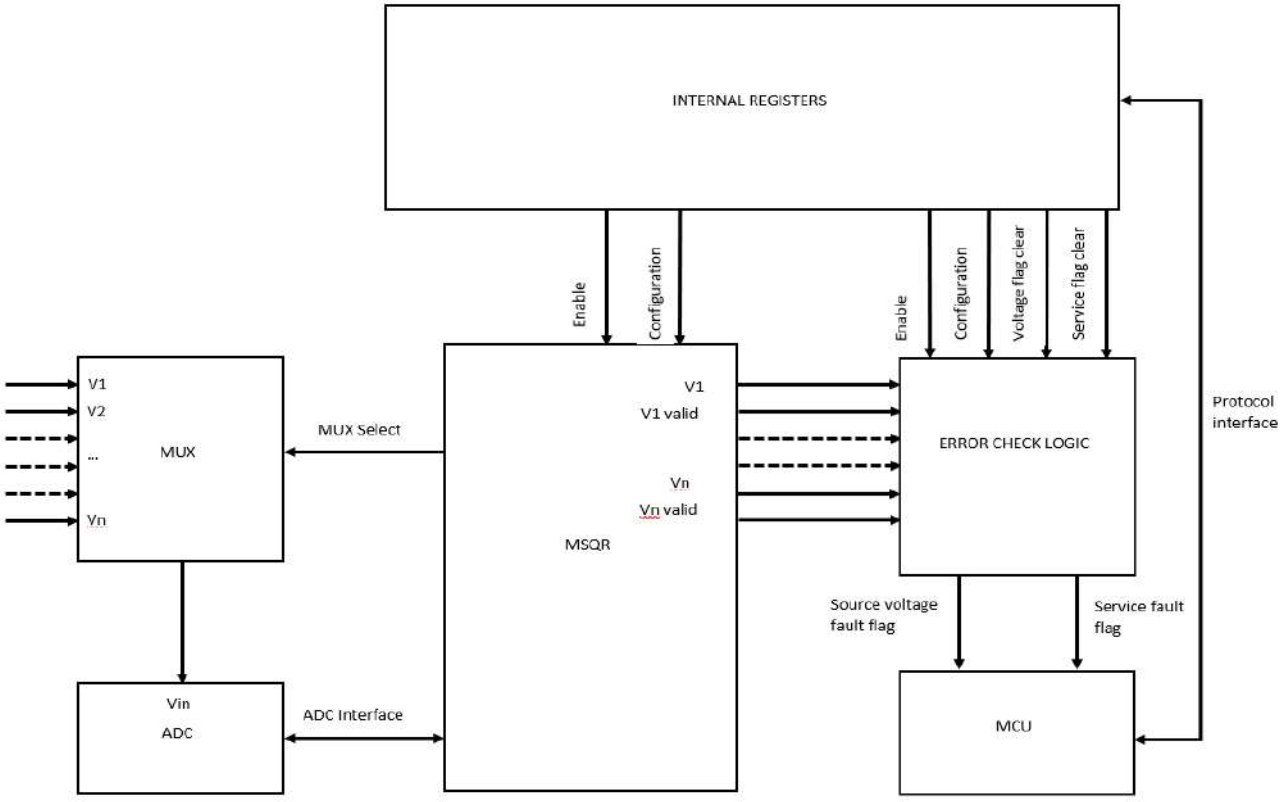


Рисунок 3.1 – архітектура системи вимірювання

Концепція функціонування інтегрованої системи вимірювання, що об'єднує аналого-цифровий перетворювач (АЦП), мультиплексор та логіку вимірювання (ЛВ), працює за наступною схемою:

Перша частина системи вимірювання — *логіка вимірювання (далі — ЛВ)*, що відповідає за управління мультиплексором, здійснюючи перемикання між каналами.

Оскільки мультиплексор є аналоговим компонентом, перед зчитуванням сигналу необхідно дочекатися часу його стабілізації. Цей проміжок часу, відомий як час встановлення сигналу (settling time), є критичним для забезпечення точності вимірювань.

Після стабілізації сигналу ЛВ ініціює процес вимірювання з АЦП, надсилаючи запит на перетворення аналогового сигналу у цифрове значення та очікує на завершення цього процесу.

Отримавши результат від АЦП, ЛВ передає це виміряне значення напруги на вихід, відзначаючи його як валідний, і готує систему до наступного циклу вимірювань.

Система повторює цикл, переходячи до вимірювання наступного входу, забезпечуючи таким чином неперервне відстеження стану всіх каналів.

Цей циклічний процес дозволяє системі вимірювання ефективно та послідовно здійснювати зчитування напруги з усіх вхідних каналів, гарантуючи актуальність даних та оперативне реагування на будь-які зміни в параметрах вхідних сигналів.

Друга частина системи вимірювання — *логіка перевірки (далі — ЛП)*, яка відіграє роль сторожового механізму, забезпечуючи контроль за відповідністю виміряних напруг до встановлених норм. ЛП з'єднана з вихідними сигналами від ЛВ і виконує подвійну функцію:

— детектування відсутності напруги живлення. З огляду на стандартизовані параметри напруги живлення, такі як частота зміни 50 Гц та амплітуду 220 В, ЛП моніторить наявність вимірювань вище

встановленого порога амплітуди протягом визначеного періоду (наприклад, 10 мс, що відповідає половині періоду стандартної синусоїди з частотою 50 Гц). Якщо протягом цього періоду не виявляється належної напруги, ЛП активує сигнал помилки, що вказує на відсутність сигналу живлення.

— детектування перевищення або недостатності напруги з використанням компараторів із фільтрацією. Компаратори фільтрують вимірювання, переконуючись, що вони залишаються у заданих межах, які були раніше конфігуровано. У випадку, коли виміряне значення виходить за ці межі і ця ситуація повторюється протягом декількох послідовних вимірювань, ЛП генерує сигнал помилки, який повідомляє про можливу проблему з напругою.

Ці механізми перевірки запрограмовані таким чином, щоб забезпечити раннє виявлення потенційних проблем з електропостачанням, що дозволяє підтримувати високий рівень надійності системи резервного живлення.

Опис елементів архітектури зображеної на рисунку 3.1 :

ADC (Аналого-цифровий перетворювач (АЦП)) - являє собою компонент зі змішаними сигналами, який інтегрує аналогову та цифрову логіку. Завдяки цьому АЦП отримує аналогові сигнали, обробляє їх і перетворює в цифрові дані, готові для подальшої обробки мікропроцесором.

MUX(Мультиплексор) - забезпечує можливість вибору одного з кількох вхідних сигналів для передачі на вихід. Це ключовий елемент, що дозволяє оптимізувати кількість використовуваних АЦП, надаючи можливість вимірювати багато сигналів через один пристрій.

MSQR (Measurement sequencerь, Логіка вимірювання) - цифровий компонент, відповідальний за послідовність вимірювань. MSQR керує процесом вимірювання, забезпечуючи правильну синхронізацію між АЦП та мультиплексором, а також визначає моменти для запису даних у регістри та систему моніторингу, гарантуючи їх валідність.

Error check logic (Логіка перевірки) - це система моніторингу, яка аналізує отримані вимірювання на предмет відповідності заданим параметрам. У разі виявлення відхилень від норми, таких як перевищення або зниження напруги, або повної відсутності живлення, вона генерує сигнал тривоги до мікроконтролера.

Internal registers - це блок пам'яті всередині інтегральної схеми, який зберігає конфігурацію системи та виміряні значення напруги. Вбудований контролер з інтерфейсом комунікації (це може бути I2C, SPI тощо) дозволяє мікроконтролеру зчитувати та записувати дані у ці регістри.

MCU (micro control unit) – мікроконтролер, який отримує від системи вимірювання сигнали про помилки та взаємодіє з нею за допомогою доступу до блоку пам’яті.

3.3 Створення модулів системи

Для підтвердження теоретичних розрахунків першого параграфу, поставимо собі задачу виміряти тринадцять напруг, використовуючи тактову частоту системи рівня 32 МГц.

Для чіткої відповідності спроектованого дизайну до системи резервного живлення визначимо ключові напруги, які властиві даним системам :

1. Напруга мережі 1 фази;
2. Напруга мережі 2 фази;
3. Напруга мережі 3 фази;
4. Випрямлена напруга 1 фази;
5. Випрямлена напруга 2 фази;
6. Випрямлена напруга 3 фази;
7. Напруга акумулятора;
8. Напруга зарядки акумулятора;
9. Вихідна випрямлена напруга 1 фази;
10. Вихідна випрямлена напруга 2 фази;
11. Вихідна випрямлена напруга 3 фази;
12. Напруга з сенсора температури інвертора;
13. Напруга з сенсора температури акумулятора.

В якості інтерфейсу для аналого-цифрового перетворювача візьмемо за основу протокол комунікації, який включає в себе цифрові сигнали SOC, EOA, EOC та Data (виміряні дані).

SOC (Start of Conversion): Цей сигнал використовується для ініціювання процесу перетворення в АЦП. Активація цього сигналу означає, що АЦП повинен почати процес перетворення аналогового сигналу.

EOA (End of Acquisition): Сигнал EOA вказує на завершення процесу зчитування аналогового сигналу. Це важливо для забезпечення достатнього часу на стабілізацію вхідного сигналу перед його перетворенням.

EOC (End of Conversion): Сигнал EOC сигналізує про завершення процесу перетворення та готовність цифрових даних до використання або подальшої обробки. Це ключовий момент для передачі даних до логіки вимірювання або логіки перевірки.

Протокол комунікації зображений на рисунку 3.2

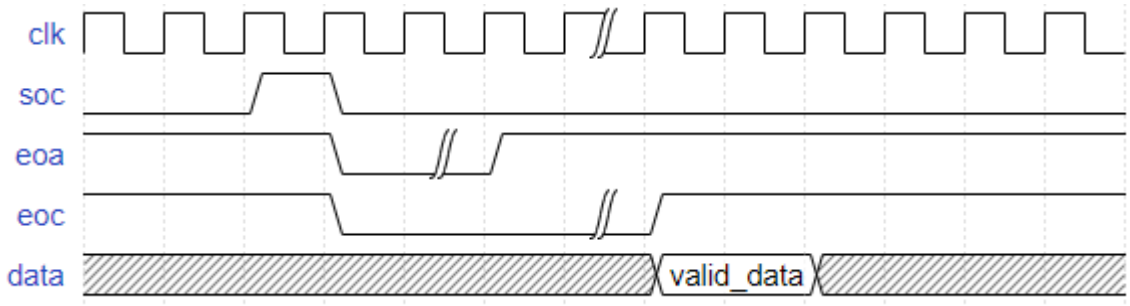


Рисунок 3.2 – Протокол комунікації АЦП

3.4 Модуль логіки вимірювання

Опишемо інтерфейс модулю відповідно до визначеного концепту роботи, обраного інтерфейсу АЦП та поставленої задачі - виміряти тринадцять напруг (див. Додаток А).

Можемо помітити додатковий вхід конфігурації за допомогою, якого ми додатково залишаємо можливість конфігурувати кількість тактів високого рівня для сигналу SOC, на випадок, якщо буде необхідність використовувати АЦП , яке працює з частотою нижчою ніж частота нашого сигналу тактування.

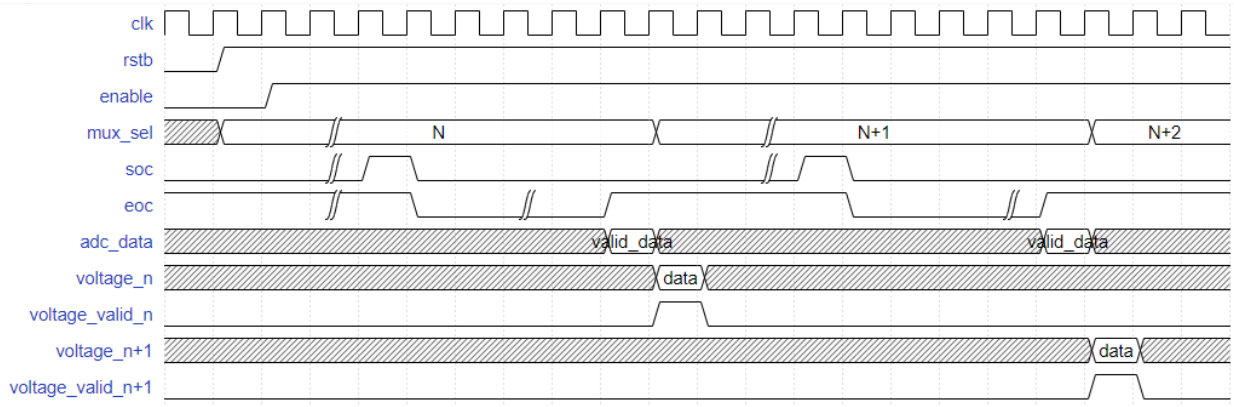


Рисунок 3.3 – Діаграма зміни сигналі логіки вимірювання

Тепер ми можемо визначити наші вимоги до виставлення сигналів у вигляді діаграми зміни сигналів та описати наш алгоритм роботи. Алгоритм включає в себе наступні кроки:

1. Ініціація Вимірювання:

   Спочатку, зі зміною значення сигналу mux_sel (або після активації модуля), система входить у стан очікування затримки виставлення сигналу мультиплексору (конфігурується за допомогою входу settling_time).

2. Запуск Вимірювання АЦП:

   Після закінчення затримки виставлення мультиплексору, система генерує сигнал SOC для АЦП, піднімається на один тік тактуючого сигналу, чим ініціює процес вимірювання.

3. Очікування Завершення Перетворення:

   Система очікує на зміну рівня сигналу EOC з низького на високий, що вказує на завершення процесу перетворення АЦП.

4. Оновлення Вихідних Даних:

   Після отримання сигналу EOC, система виставляє виміряне значення напруги на відповідний вихідний регістр і активує відповідний сигнал валідації, що підтверджує достовірність даних.

5. Повторення Циклу:

   Завершивши передачу виміряних даних, система оновлює значення вибору на мультиплексорі, переходячи до наступного каналу вимірювання, та повторює весь цикл знову.

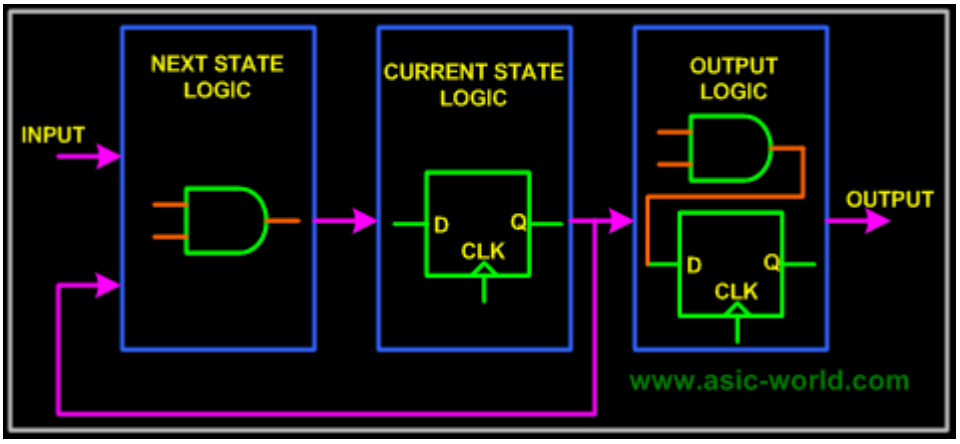


Рисунок 3.4 – Візуалізація скінченого автомату [10]

Алгоритм роботи логіки вимірювання можна ефективно представити за допомогою скінченного автомату (Finite State Machine, FSM). Використання FSM у проектуванні цифрових систем є ідеальним рішенням для організації та управління послідовністю операцій та станів.

Опис скінченого автомата складається з трьох основних частин [10]:

1. Комбінаційна логіка:

   Вона відіграє роль у визначенні наступного стану системи. На основі поточного стану та вхідних сигналів, комбінаційна логіка визначає, який стан буде активовано наступним. Це ключовий елемент для реагування системи на зміни в умовах експлуатації або даних.

2. Послідовна логіка:

   Ця частина FSM забезпечує зберігання інформації про поточний стан системи. Послідовна логіка важлива для збереження контексту між різними циклами роботи системи, дозволяючи відслідковувати переходи від одного стану до іншого.

3. Вихідна логіка:

   Вона відповідає за генерацію вихідних сигналів системи на основі поточного стану. Вихідна логіка є ключовою для взаємодії системи з зовнішнім середовищем та іншими компонентами системи, оскільки саме через неї реалізується відповідь на зовнішні події чи вимоги.

Створюємо скінчений автомат, який буде складатись з 5 станів :

1. IDLE – стан скидання або вимкнений модуль.
2. WAIT_SETTLING – сетлінг стан очікування затримки переключення мультиплексора.
3. SEND_SOC – стан відправки запиту виміру на АЦП.
4. WAIT_EOC - стан очікування результату перетворення АЦП.
5. MUX_CHANGE – стан зміни вибору мультиплексору.

Діаграма зміни станів зображена на рисунку 3.5

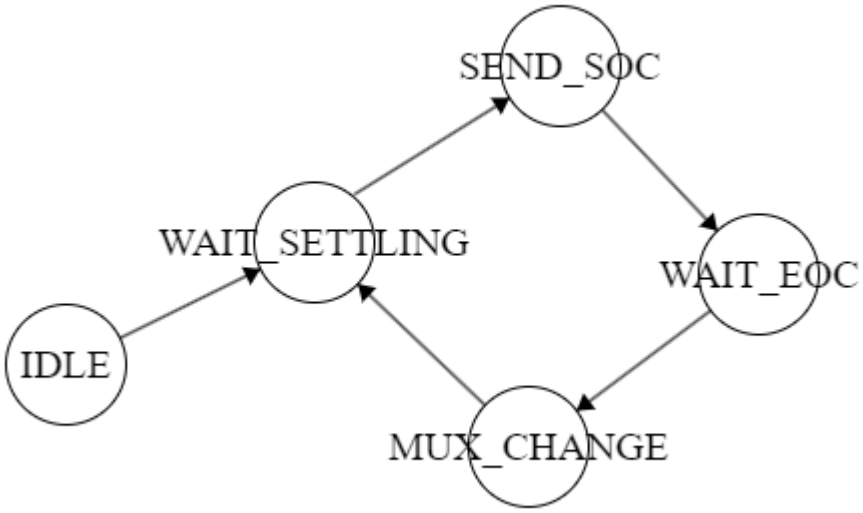


Рисунок 3.5 – Діаграма зміни станів FSM логіки вимірювання

Створений код модуля знаходиться у додатку А.

3.5 Модуль логіки перевірки

Модуль логіки перевірки виконує дві основні функції, що забезпечують цілісність та безпеку електричної системи.

1. Детекція відсутності напруги мережі:

Ця система постійно відстежує напругу протягом кожного півперіоду вхідного сигналу (синусоїда з частотою 50 Гц), який для стандартизованої

мережі становить 10 мс, аналізуючи чи було досягнуто референсних амплітудних значень, що були попередньо конфігуровано. Якщо протягом цього відрізку часу амплітудне значення не виявлено, система ідентифікує це як відсутність необхідної напруги, що може свідчити або про перебої в електропостачанні або його повне відключення. Завдання цієї складової модулю – забезпечити раннє виявлення таких станів з подальшим інформуванням мікроконтролеру о проблемі, яку той в свою чергу має обробити.

2. Перевірка напруг на пере/недонапругу:

Друга частина модулю зосереджена на вимірюваних напругах, що поступають від різних точок системи. Вона аналізує ці дані, порівнюючи їх із заданими порогами безпеки, щоб ідентифікувати стани перенапруги або недонапруги і сигналізує при знаходженні помилки.

Оптимальним рішенням в даній ситуації буде створення двох окремих модулів логіки перевірки, де кожен модуль призначений для роботи з однією конкретною напругою.

Перший модуль, який займається виявленням відсутності напруги, зосереджений на моніторингу окремої фази напруги мережі, використовуючи задані параметри для визначення достатності амплітудних значень.

Другий модуль, розрахований на виявлення пере/недонапруги одного сигналу. Кожна з цих напруг вимагає індивідуально налаштованого модуля. Таким чином такий підхід до структуризації системи не тільки додає зручність створення дизайну, але й зменшує кількість можливих помилок, що забезпечує високий рівень надійності та безпеки в роботі системи резервного живлення.

3.5.1 Підмодуль детекції відсутності напруги

Опишемо принцип роботи модулю:

Завдання модуля полягає у неперервній оцінці значення напруги, паралельно з якою запускається таймер, налаштований на визначений часовий

інтервал. Цей інтервал відповідає часу, протягом якого має бути виявлено задане значення напруги, яке у нашому випадку відповідає нормальним умовам роботи мережі.

Якщо протягом встановленого часу система реєструє вхідну напругу, яка відповідає заданим параметрам, вона вважає стан електромережі задовільним та продовжує циклічний моніторинг заново.

Однак, якщо після закінчення відліку система так і не зафіксувала необхідні значення напруги, активується сигнал оповіщення про помилку. Цей сигнал залишається активним до моменту його скидання, що здійснюється вручну за допомогою сигналу очистки, який приводить систему до початкового стану та готовності до повторного моніторингу.

Для покращення ефективності роботи модуля виявлення відсутності напруги та оптимізації використання ресурсів, було вирішено ввести додатковий сигнал *clk_en_100us*. Цей сигнал слугуватиме як таймер з базовим інтервалом у 100 мс, забезпечуючи можливість відліку розширених часових періодів без необхідності використання великої кількості регістрів для зберігання даних.

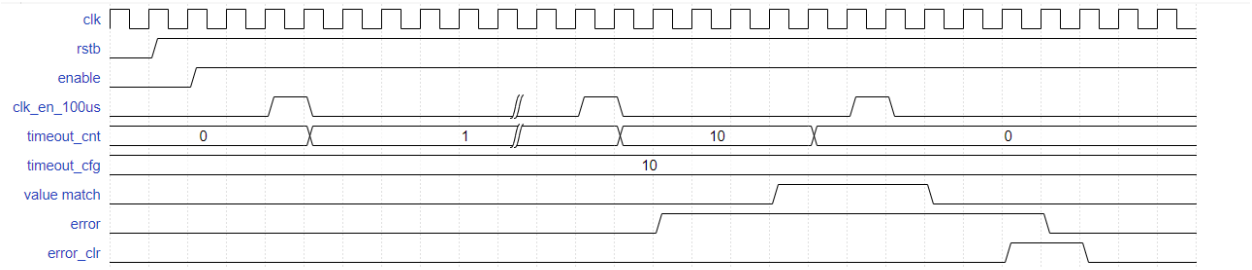


Рисунок 3.6 – Діаграма зміни сигналів підмодуля виявлення напруг

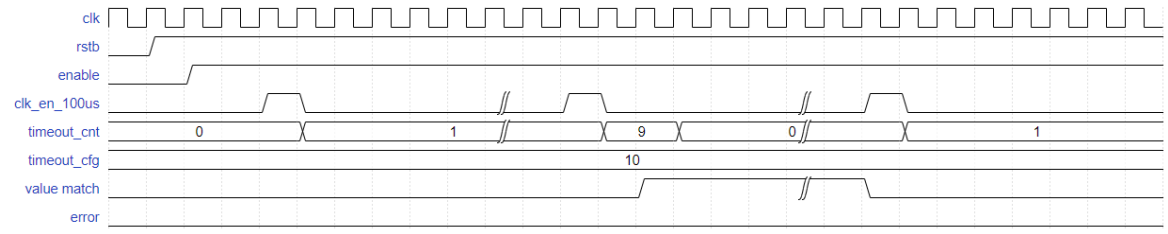


Рисунок 3.7 – Діаграма зміни сигналів підмодуля виявлення напруг

Опис модуля можна знайти у Додатку Б.

3.5.2 Підмодуль виявлення пере/недонапруг

Процес роботи модуля полягає в постійному моніторингу вимірюваного сигналу. Система аналізує, чи знаходиться сигнал в межах попередньо заданого діапазону вимірювань. Цей діапазон визначається в конфігурації модуля і є ключовим для ідентифікації пере- та недонапруг.

Якщо в ході послідовного вимірювання фіксується, що напруга не потрапляє в заданий діапазон протягом визначеної кількості вимірювань, модуль генерує сигнал помилки.

Цей сигнал помилки є індикатором потенційної проблеми в системі, що вимагає негайної уваги чи втручання. Такий підхід дозволяє забезпечити високу точність та надійність системи вимірювання, а також забезпечує можливість оперативного реагування на випадки аномальних змін напруги, підвищуючи загальну безпеку експлуатації системи резервного живлення.

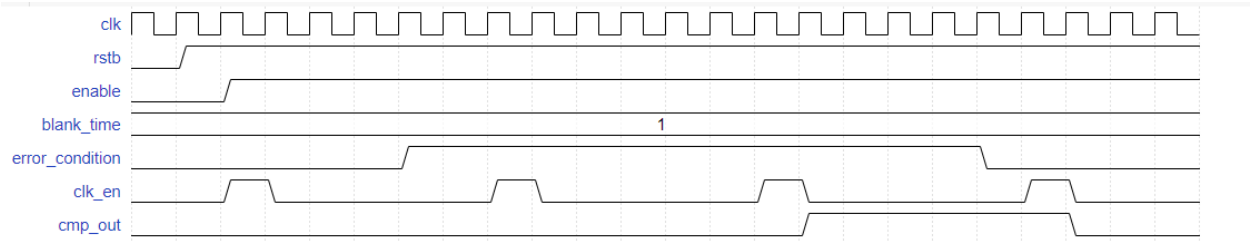


Рисунок 3.8 – Діаграма зміни сигналів підмодуля виявлення пере/недонапруг

Опис модуля можна знайти у Додатку В.

3.5.3 Топовий модуль логіки перевірки

Для реалізації топового модуля логіки перевірки ми ініціюємо створення екземплярів розроблених раніше підмодулів детекції відсутності напруги та перевірки на пере/недонапругу, інтегруючи їх в єдину систему.

Кожен екземпляр підмодуля підключається до інтерфейсу топового модуля та забезпечує взаємодію між окремими елементами системи. Це дозволяє ефективно управляти процесами моніторингу та діагностики різних параметрів напруги в мережі.

Окрім координації роботи підмодулів, важливою частиною топового модуля є логіка інформування про проблеми. Ця логіка активується у випадках виявлення станів недо- або перенапруги і відсутності напруги та генерує відповідні сигнали оповіщення. Ці сигнали можуть бути використані для зовнішнього інформування або активації захисних механізмів.

Також, в топовому модулі передбачена логіка очистки сигналів інформування. Ця логіка дозволяє скидати сигнали помилок після їх реєстрації та аналізу, готуючи систему до подальшої стабільної роботи.

Зазвичай, скидання сигналів помилок виконується вручну через активацію сигналу скиду або іншого контрольного механізму, визначеного в конфігурації системи. Це гарантує, що кожен інцидент буде належно оброблено та враховано перед поверненням системи до нормального стану.

Опис модуля знаходиться у додатку Г.

4 МОДЕЛЮВАННЯ ФУНКЦІОНУВАННЯ СПРОЕКТОВАНОГО ЦИФРОВОГО ДИЗАЙНУ

4.1 Інструменти моделювання

Для моделювання ми звертаємося до продуктів компанії Cadence, які є стандартом у галузі проектування та моделювання цифрових систем.

Для симуляції нашого цифрового дизайну ми використовуємо Cadence Xcelium, програмне забезпечення, яке дозволяє проводити точні та ефективні симуляції цифрових схем.

Xcelium надає широкий спектр можливостей для перевірки та аналізу поведінки системи в різних умовах, дозволяючи нам глибше зрозуміти функціональність та потенційні вузькі місця в нашому дизайні.

Для відображення результатів симуляцій ми обрали SimVision, інструмент, який ефективно візуалізує дані у вигляді вейвформ.

SimVision забезпечує гнучкий інтерфейс для аналізу хвильових форм, дозволяючи нам детально вивчати поведінку сигналів та інших параметрів нашої системи в часі. Цей інструмент стає незамінним у процесі діагностики та оптимізації цифрового дизайну, дозволяючи виявляти та усувати проблеми ще на етапі моделювання, перш ніж переходити до фізичного прототипування.

Використання цих інструментів дозволяє нам забезпечити високу якість та надійність розробленого цифрового дизайну, підвищуючи впевненість у його коректній роботі в реальних умовах. [11]

Подальша симуляція та аналіз забезпечують глибоке розуміння системи та її поведінки, що є критично важливим для створення ефективного та безпечного рішення в галузі електроніки.

4.2 Підходи до створення тестових оточень для моделювання

Для запуску симуляції та демонстрації роботи модуля відповідно до очікуваної функціональності необхідно створити тестове оточення. Існує кілька підходів для створення такого тестового середовища, а саме :

— Використання блоку initial для опису змін сигналів.

Цей метод ефективний, якщо модуль має простий інтерфейс та зрозумілий патерн взаємодії. Це дозволяє легко моделювати поведінку модуля за заданими умовами;

— Створення інтерфейсів з вбудованими механізмами

Цей варіант передбачає наявність інтерфейсів, які автоматизують процес встановлення певних патернів сигналів. Це забезпечує зручність у використанні, але може мати обмеження у гнучкості налаштування.

— Опис додаткової логіки для контролю сигналів;

Цей підхід включає розробку логіки, яка контролює сигнали, що подаються на вхід дизайну. Ця логіка може бути як синтезованою, так і несинтезованою;

— Створення тестового оточення за допомогою UVM .

Це популярний підхід, який використовується більшістю спеціалістів. UVM надає розширені можливості для тестування і верифікації, забезпечуючи високий рівень абстракції та гнучкості.

Кожен з цих підходів має свої переваги та обмеження, і вибір конкретного методу залежить від конкретних вимог та умов проекту. [12]

Для моделювання нашого дизайну нам необхідно створити тестове оточення за допомогою UVM, так як нам необхідна його висока гнучкість та змога використовувати його розширені можливості для створення вхідних стимулів.

4.3 Universal Verification Methodology

UVM (Universal Verification Methodology) є стандартизованою методологією, що використовується для верифікації цифрових дизайнів та систем на кристалі.

Цей фреймворк, створений на основі мови SystemVerilog, надає розробникам потужний інструментарій для створення стандартизованих, модульних тестових оточень, які можуть бути легко повторно використані в різних проектах. Основна його перевага полягає у здатності адаптуватися до широкого спектру дизайнів, від простих до вкрай складних систем.

UVM використовує об'єктно-орієнтовані принципи та шаблони проектування, що дозволяє створювати високо абстрактні та повторно використовувані компоненти. Це істотно знижує час та витрати на процес верифікації, роблячи UVM вибором багатьох інженерів та дизайнерів. Такий підхід до модульності і використання стандартизованих компонентів дозволяє легко інтегрувати тестове оточення у різні середовища розробки.

Одним з його ключових аспектів є його здатність до детальної перевірки взаємодії між різними блоками системи. Це дозволяє ідентифікувати та вирішувати потенційні проблеми на ранніх етапах розробки, що суттєво підвищує якість кінцевого продукту.

Крім того, він має велику базу інструкцій та найкращих практик, що забезпечує розробникам доступ до перевірених методик та технік. Це сприяє ефективному навчанню та обміну знаннями в галузі.

Враховуючи вищезазначене, UVM ідеально підходить для симуляцій та аналізу в контексті наших цілей. Він забезпечує високий рівень гнучкості у верифікації, дозволяючи розробникам точно налаштовувати тестове середовище для досягнення оптимальних результатів.

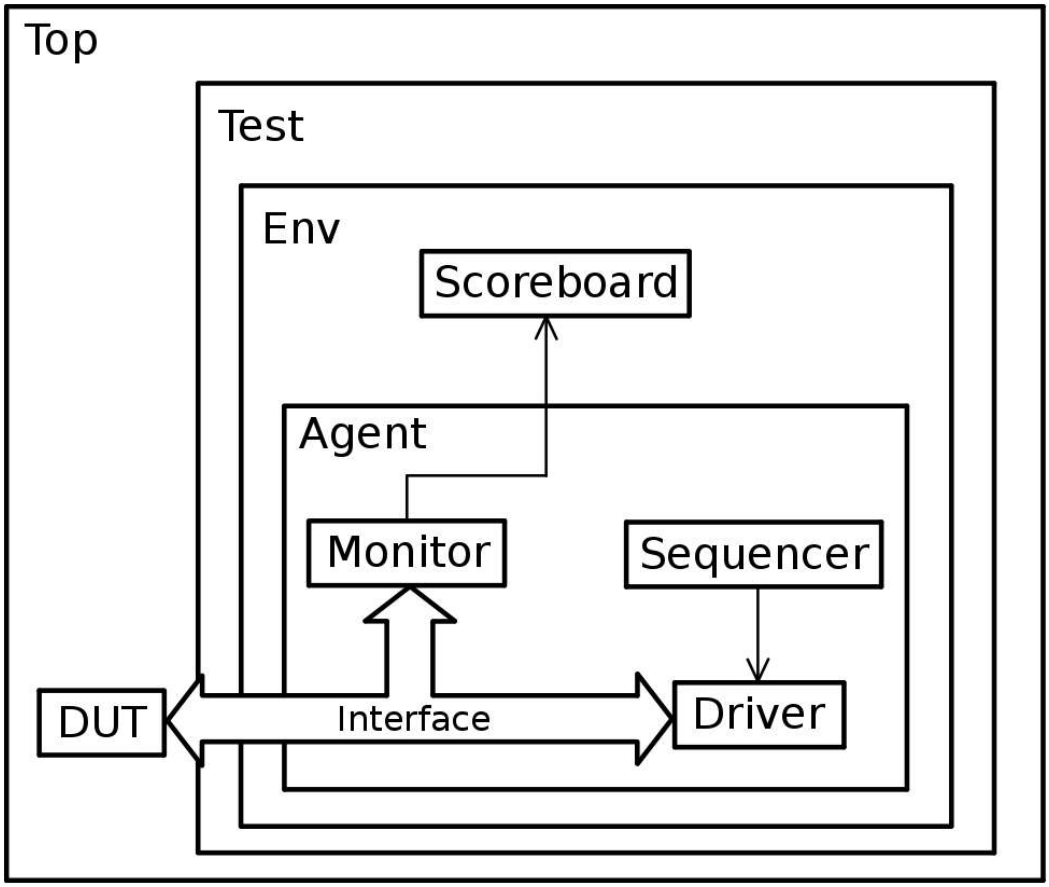


Рисунок 4.1 – Структура тестового оточення UVM [13]

Для ефективного використання UVM, важливо розуміти структуру та функції ключових компонентів тестового оточення.

Почнемо з базового елементу UVM – універсального компоненту (UVC).

UVC (Universal Verification Component) - це стандартизований компонент у методології UVM, який використовується для моделювання, верифікації та тестування певних аспектів цифрового дизайну або системи на кристалі.

Кожен UVC розробляється для виконання специфічних функцій у процесі верифікації та може включати різні елементи, такі як монітори, драйвери, секвенсери, транзакції тощо.

Використання UVC дозволяє стандартизувати та спростити процес верифікації, забезпечуючи високу гнучкість та можливість повторного використання компонентів у різних проектах.

Елементи з яких складається універсальний компонент :

1. UVM Transaction:

   У цьому елементі описуються дані, з якими будуть взаємодіяти інші елементи UVC. Тут визначаються конкретні дані, які необхідно передати/отримати різні конфігурації для їх виставлення, та інша сервісна інформація, що у подальшому дозволяє моделювати різноманітні сценарії взаємодії з верифікованим модулем.

2. UVM Monitor:

   Монітор відповідає за відстеження подій, що відбуваються на інтерфейсі чи шині. Цей компонент збирає інформацію про те, що відбувається в інтерфейсі, і перетворює ці дані у транзакції для подальшого аналізу.

3. UVM Driver:

   Драйвер взаємодіє з інтерфейсом, виконуючи дії, передбачені транзакцією. Цей елемент є критичним для моделювання взаємодії з верифікованим дизайном, відтворюючи реальні умови експлуатації.

4. UVM Sequencer:

   Секвенсер забезпечує зв'язок між транзакцією (sequence) та драйвером. Його головна задача - передача транзакції до драйвера, що забезпечує логічний потік даних та команд.

5. UVM Config:

   Конфігурація в UVM містить інформацію про інтерфейс та налаштування компонентів тестбенчу. Це може включати конфігурацію агенту, монітору, драйверу тощо.

6. UVM Agent:

   Агент об'єднує всі інші компоненти, створюючи повноцінне тестове середовище. Це дозволяє інтегрувати різні частини тестового оточення для забезпечення єдиної, координованої верифікації.

Першим кроком створення тестового оточення є розбиття інтерфейсу модуля на групи та визначення необхідних UVC для їх верифікації. Кожен компонент UVC повинен відповідати конкретній частині інтерфейсу модуля, щоб забезпечити всебічну перевірку його функціональності.

Після визначення необхідних UVC, ми переходимо до створення топ-рівневого файла SystemVerilog. У цьому файлі створюється екземпляр дизайну, що підлягає верифікації, разом з необхідними інтерфейсами для тестування. Ці інтерфейси також в подальшому будуть підключатися до UVC, які розміщені у середовищі верифікації.

Наступним етапом є створення UVM-середовища (UVM Environment), де агрегуються необхідні UVC-агенти. Важливо, щоб всі компоненти належно підключалися та взаємодіяли між собою для забезпечення ефективної верифікації.

Після створення UVM-середовища, ми переходимо до розробки базового тесту. Цей тест включає в себе все створене середовище, включаючи всі підключення та конфігурації. Базовий тест становить основу для подальшої розробки специфічних тест кейсів.

Кінцевий етап полягає у створенні конкретних тестів, які містять тест кейси, написані для конкретних сценаріїв взаємодії з інтерфейсом.

Для кожного агента створюються окремі послідовності (sequences), які описують певні сценарії або частини сценаріїв. Це дозволяє детально моделювати різні ситуації, які можуть виникнути під час роботи верифікованого модуля.

Кожна послідовність є унікальним набором дій, що імітують певні умови взаємодії з модулем. Це дозволяє не тільки перевірити основну функціональність, але й виявити потенційні проблеми в крайових випадках або при нетипових умовах роботи.

У процесі створення тестового оточення важливо також враховувати можливість повторного використання компонентів в майбутніх проектах. Це

не тільки знижує час на розробку нових тестів, але й забезпечує більшу гнучкість та надійність процесу верифікації. [14]

Завершальним кроком є аналіз отриманих результатів тестування, що дозволяє виявити слабкі місця дизайну та внести необхідні корективи.

Систематичний та детальний підхід до створення тестового оточення в UVM забезпечує високий рівень довіри до якості верифікованого продукту, знижуючи ризики та помилки у кінцевому дизайні.

На основі аналізу розроблюваних нами інтерфейсів модулів, ми визначили потребу у створенні двох ключових компонентів та двох інтерфейсів, які дозволять легко замінити нам необхідні нам додаткові компоненти, не втрачаючи зручності використання. Цей підхід дозволяє нам оптимізувати процес верифікації, забезпечуючи високу гнучкість, з цим набором ми можемо створити по тестовому оточенню на кожен наш модуль.

Перший компонент *adc_uvc* призначений для виконання транзакцій, пов'язаних з аналого-цифровим перетворенням. Використання *adc_uvc* дозволяє нам моделювати поведінку АЦП, включаючи різні сценарії обробки сигналів та передачі даних.

Другий компонент *wire_uvc* призначений для синхронного виставлення різноманітних сигналів. Цей універсальний компонент надає можливість моделювати різні варіанти взаємодії сигналів в нашій системі, що забезпечує всебічне тестування та перевірку функціональності різних частин дизайну.

Щодо інтерфейсів, які ми вирішили реалізувати, перший з них – це *clock_if*. Цей інтерфейс виконує функцію генерації тактових імпульсів і сигналів скиду, що є фундаментальним для синхронізації та контролю всієї системи. Завдяки йому ми можемо точно налаштувати та контролювати часові параметри нашої системи, що є критично важливим для належного функціонування та тестування цифрових компонентів.

Другий інтерфейс *clock_en_if* призначений для генерації однотактних пульсів через задані проміжки часу. Цей інтерфейс важливий для точного та

контрольованого виставлення часових сигналів, що дозволяє імітувати різноманітні робочі умови та сценарії в рамках тестування.

Для моделювання використаємо підхід симуляції з низу в гору, тобто промоделюємо кожен модуль починаючи з підмодулів до під'єднаних між собою модулів логіки вимірювання та логіки перевірки.

Опис топових файлів тестових оточень можна знайти у додатку Д.

4.4 Моделювання підмодулю виявлення відсутності напруги

Опишемо сценарій моделювання для підмодулю виявлення відсутності напруги:

1. Налаштовуємо конфігурацію модуля;
2. Вмикаємо модуль сигналом enable;
3. Подаємо цифрові значення напруги, які сигналізують про нормальну напругу мережі;
4. Подаємо цифрові значення напругі, які сигналізують про відсутність напруги мережі, періодично очищуючи сигнал помилки;
5. Подаємо цифрові значення напругі, які сигналізують про відсутність напруги мережі;
6. Подаємо цифрові значення напруги, які сигналізують про нормальну напругу мережі;
7. Очищуємо сигнал помилки;
8. Подаємо цифрові значення напруги, які сигналізують про нормальну напругу мережі.

За допомогою даного сценарію ми побачимо, що :

1. При нормальних значень напруги у нас не буде з'являтись помилка;
2. При значеннях відсутності напруги у нас буде з'являтись помилка;
3. Ми очікуємо конфігурований час виявлення перед виставленням помилки;
4. Ми можемо очистити помилку тільки за допомогою сигналу очистки.

Тепер ми можемо оцінити результат симуляції за допомогою продукту Simvision.

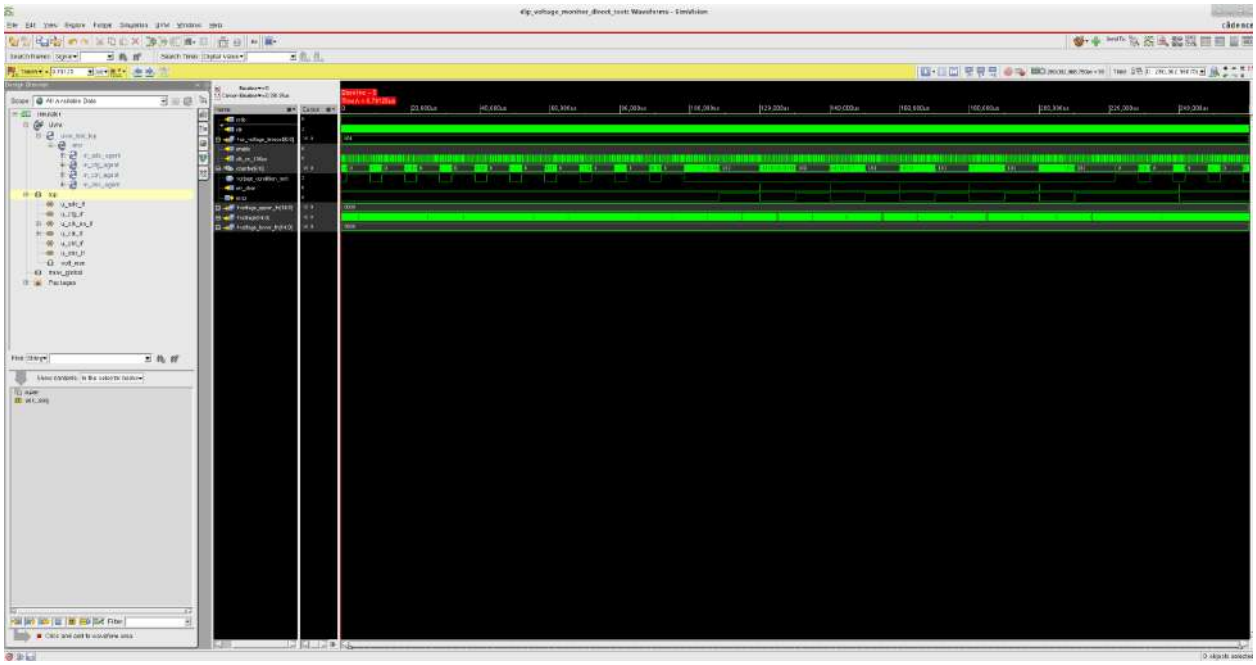


Рисунок 4.2 – Відображення вхідних та вихідних сигналів модулю виявлення відсутності напруги

На рисунку 4.2 ми можемо побачити відображення всіх сигналів нашого модулю, тепер для візуальної наглядності переведемо сигнал напруги з цифрового відображення (відображається цифровий код) до аналогового відображення та поставимо два курсори на амплітудні значення нашого сигналу. Тепер ми можемо оцінити, який сигнал стимулів ми отримали (рисунок 4.3). [15]

Рис 4.3 – Відображення вхідних та вихідних сигналів модулю виявлення відсутності напруги з масштабуванням

Також ми можемо порахувати період нашої синусоїди за допомогою виставлених курсорів та функції відображення часової різниці між курсорами (рисунок 4.4.) [15]

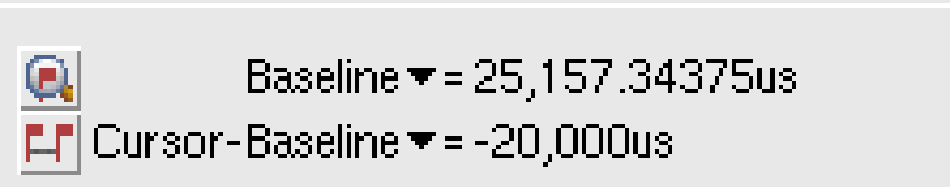


Рисунок 4.4 – Різниця часу між курсорами

Тепер оцінимо сценарії моделювання.

Сценарій стимулів №1.

На рисунку 4.5 ми можемо побачити частину сценарію, де наша напруга задовольняє нашим налаштуванням і сигнал помилки відсутній.

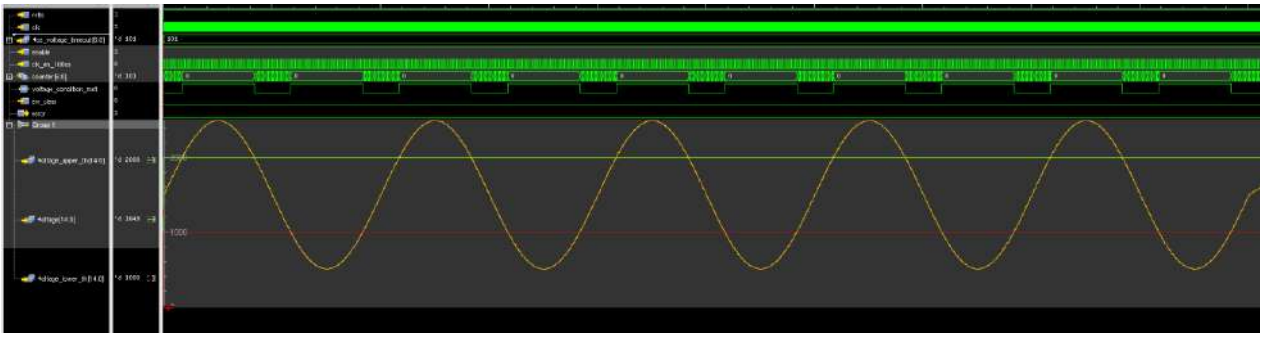


Рисунок 4.5 – Сценарій стимулів №1. Відображення вхідних та вихідних сигналів модулю виявлення відсутності напруги

Сценарій стимулів №2.

На рисунку 4.6 ми можемо побачити частину сценарію, де наша напруга не задовольняє нашим налаштуванням і сигнал помилки присутній до моменту приходу сигналу очищення.

Також за допомогою розташування курсорів ми можемо оцінити час появлення помилки і що цей час відповідає нашому налаштованому значенню

(невелика похибка значень отримана в наслідок налаштування точністю симулятора), а також за допомогою вбудованої функції обрахунку кількості переходів сигналу між рівнями.

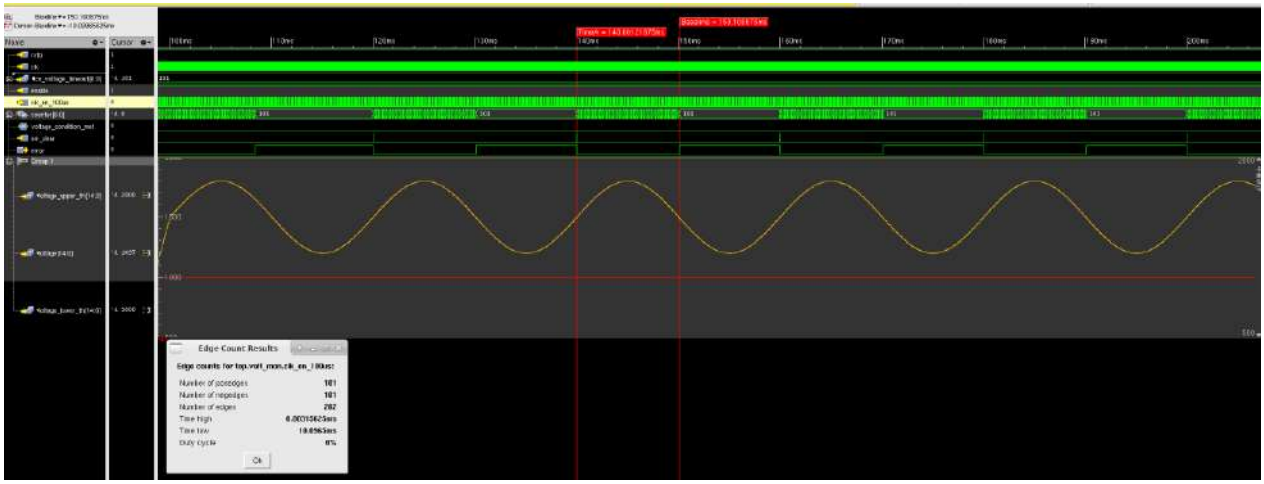


Рисунок 4.6 - Сценарій стимулів №2. Відображення вхідних та вихідних сигналів модулю виявлення відсутності напруги

Сценарій стимулів №3.

На рисунку 4.7 ми можемо побачити частину сценарію, в якій ми спочатку генеруємо сигнал помилки, а потім нормалізуємо напругу і перевіряємо, що сигнал не скидається до моменту отримання сигналу очистки.

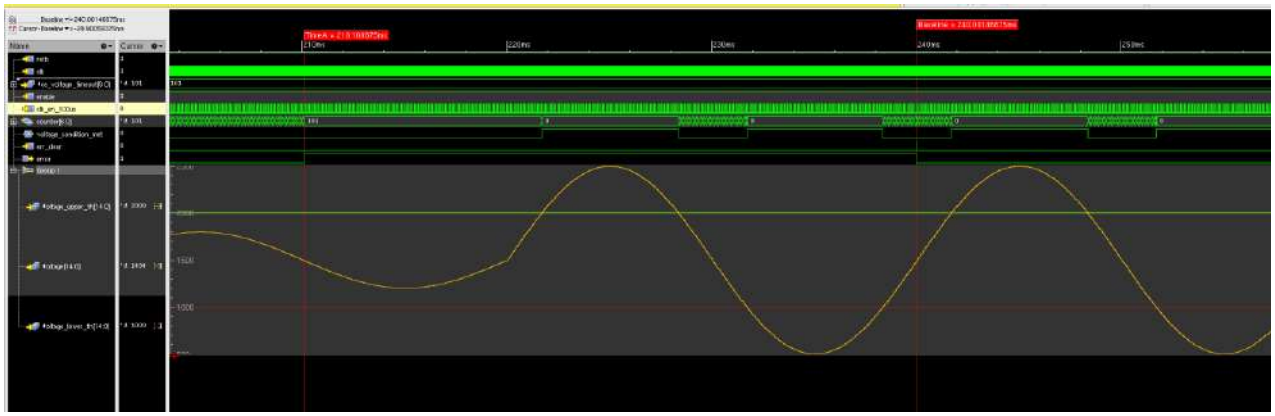


Рисунок 4.7 - Сценарій тестування №3. Відображення вхідних та вихідних сигналів модулю виявлення відсутності напруги

На рисунку 4.8 ми можемо побачити відображення структурної схеми описаного нами модулю виявлення відсутності напруг створеного за допомогою функції «Schematic tracer» застосунку Simvision.

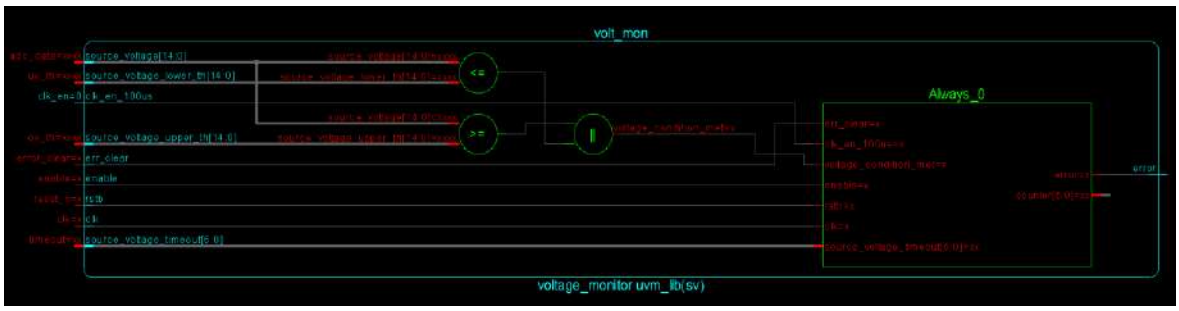


Рисунок 4.8 Структурна схема підмодулю виявлення відсутності напруги

4.5 Моделювання підмодулю виявлення недо/перенапруги

Опишемо сценарій моделювання для підмодулю виявлення недо/перенапруги :

1. Налаштовуємо конфігурацію модуля;
2. Вмикаємо модуль сигналом enable;
3. Подаємо різні значення напруги вичікуючи певні затримки перед зміною.

За допомогою даного сценарію ми побачимо, що :

1. При нормальних значень напруги у нас не буде з’являтись помилка;
2. При значеннях пере/недонапруги наш сигнал помилки буде з’являтись тільки якщо помилка буде зберігатись протягом конфігурованої кількості сигналу clk_en;
3. Значення помилки очищується, як тільки напруга стабілізувалась.

Тепер відкриваємо результати симуляції та налаштовуємо відображення.

Сценарій стимулів №1.

На рисунку 4.9 можемо побачити відображення всіх сигналів нашого модуля.

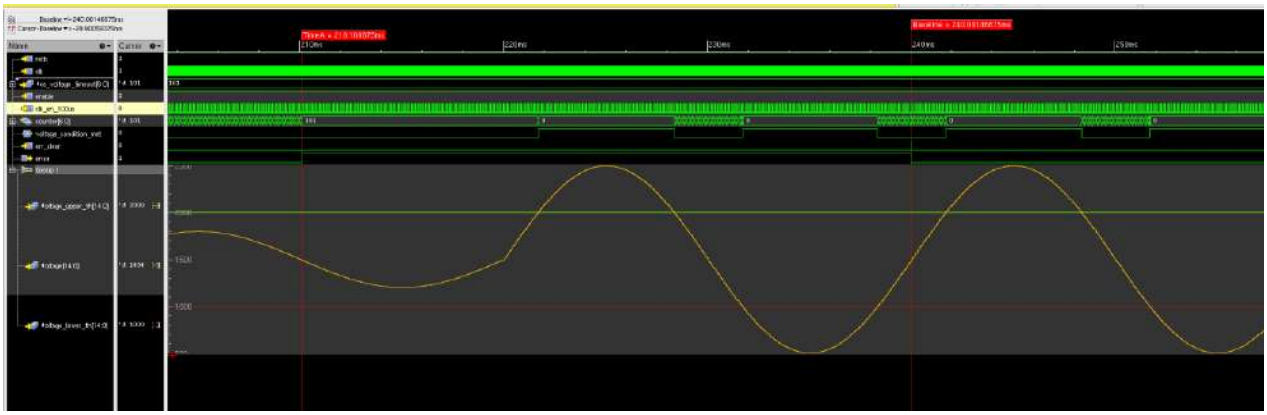


Рисунок 4.9 – Сценарій стимулів №1. Відображення вхідних та вихідних сигналів модулю виявлення недо/перенапруги

Сценарій стимулів №2.

На рисунку 4.10 можемо побачити, що якщо сигнал напруги не простояв необхідну кількість тактів з умовою недо/перенапруги, то сигнал проблеми не підніметься.

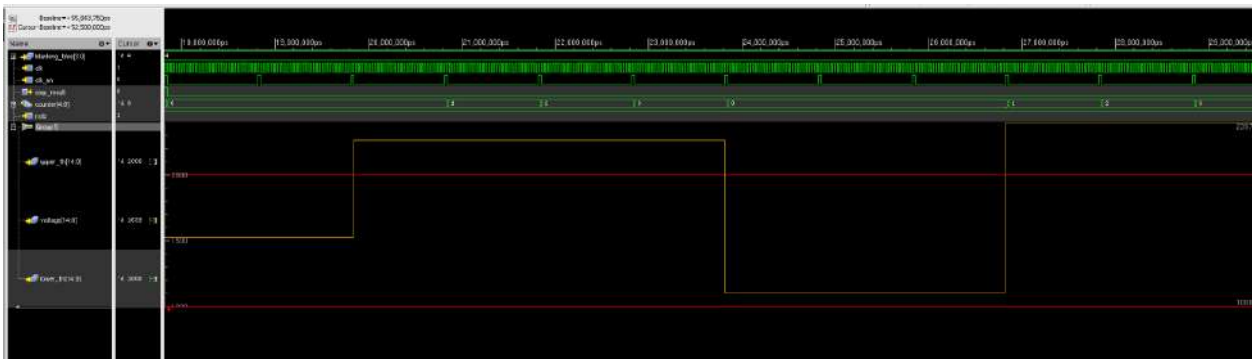


Рисунок 4.10 - Сценарій стимулів №2. Відображення вхідних та вихідних сигналів модулю виявлення недо/перенапруги

Сценарій стимулів №3.

На рисунку 4.11 можна побачити, що якщо сигнал напруги простояв необхідну кількість вимірів з умовою недо/перенапруги, то ми виставляємо сигнал проблеми, а також очищення сигналу проблеми після нормалізації напруги.

Рисунок 4.11 - Сценарій стимулів №3. Відображення вхідних та вихідних сигналів модулю виявлення недо/перенапруги

На рисунку 4.12 ми можемо побачити відображення структурної схеми описаного нами модулю виявлення недо/перенапруги створеного за допомогою функції «Schematic tracer» застосунку Simvision.

Рисунок 4.12 - Структурна схема підмодулю виявлення недо/перенапруги

4.6 Моделювання модулю логіки перевірки

Цей модуль складається з багатьох підмодулів і додатково додана функція збереження сигналу помилки від підмодулів пере/недонапруги з функціональністю його очистки, а також об’єднання сигналів помилки з підмодулів виявлення відсутності напруги.

Тому сценарій моделювання для модулю логіки перувірки буде заключатися в тому, щоб згенерувати сигнали помилки на виходах з

подальшим їх очищенням. В результаті цих дій ми зрозумієм, що функціональність працює вірно.

На рисунку 4.13, можна побачити виклик сигналу помилки відсутності напруги живлення та його подальшої очистки на прикладі сигналу *source_voltage 1*.

Рисунок 4.13 – Відображення вхідних та вихідних сигналів модулю логіки перевірки

На рисунках 4.14 та 4.15, можна побачити виклик сигналу помилки пере/недонапруги та її очистки на прикладі сигналу *battery_voltage*.

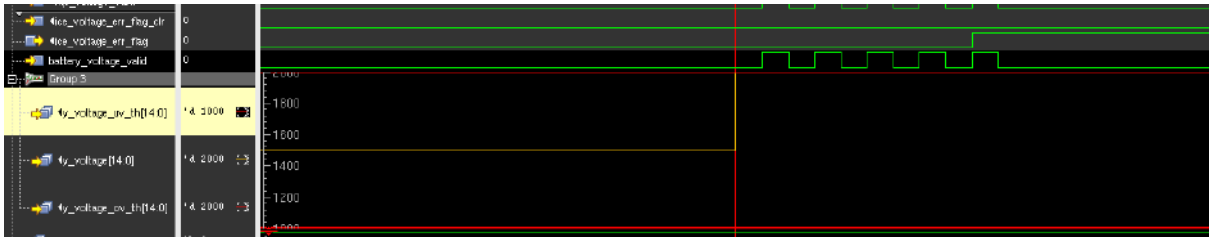


Рисунок 4.14 - Відображення вхідних та вихідних сигналів модулю логіки перевірки

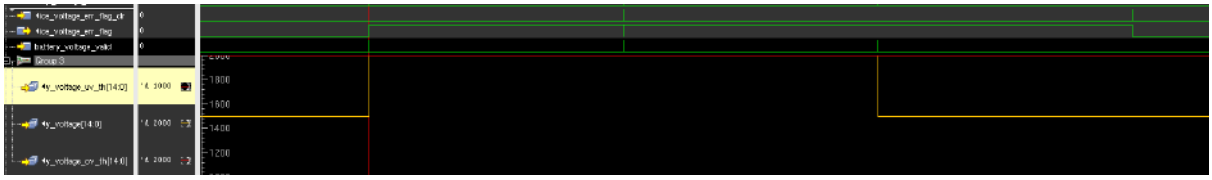


Рисунок 4.15 - Відображення вхідних та вихідних сигналів модулю логіки перевірки

На рисунку 4.16 ми можемо побачити відображення структурної схеми описаного нами модулю логіки перевірки створеного за допомогою функції «Schematic tracer» застосунку Simvision.

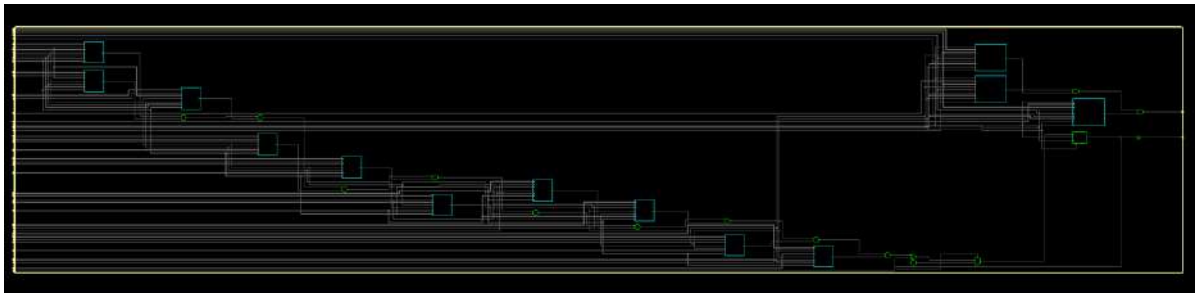


Рисунок 4.16 Структурна схема модулю перевірки

4.7 Моделювання модулю логіки вимірювання в комбінації з логікою перевірки

Так як модуль логіки вимірювання являється незалежним до модуля логіки перевірки, зручно не створювати окреме тестове оточення для моделювання ЛВ, а перевірити його в тестовому оточені яке комбінує ЛВ та ЛП.

Сценарій моделювання об’єднаних модулів логіки вимірювання та логіки перевірки :

1. Налаштувати модуль;
2. Ввімкнути модулі;
3. Запустити процес вимірювання, генеруючи сигнали живлення, які проходять по налаштованим параметрам;
4. Запустити процес вимірювання, генеруючи сигнали живлення, які не проходять по налаштованим параметрам;
5. Починаємо генерувати сигнали живлення, які проходять по налаштованим параметрам;
6. Очищуємо помилку відсутності напруг живлення;
7. Генеруємо інші напруги, які не проходять по налаштованим параметрам;
8. Генеруємо інші напруги, які проходять по налаштованим параметрам;
9. Очищуємо помилку пере/недонапруг.

Таким чином ми зможемо перевірити підключення модулів та виконання ними поставлених задач.

Для початку перевіримо, що логіка вимірювання працює коректно. На рисунку 4.17 ми можемо побачити її роботу, можна помітити, що наш модуль циклічно вимірює напруги, спочатку змінюючи сигнал вибору мультиплексора, далі вичікування затримки переключення мультиплексора, виконання вимірювання АЦП і в кінці виставленням виміряної напруги на відповідний вихід модуля.

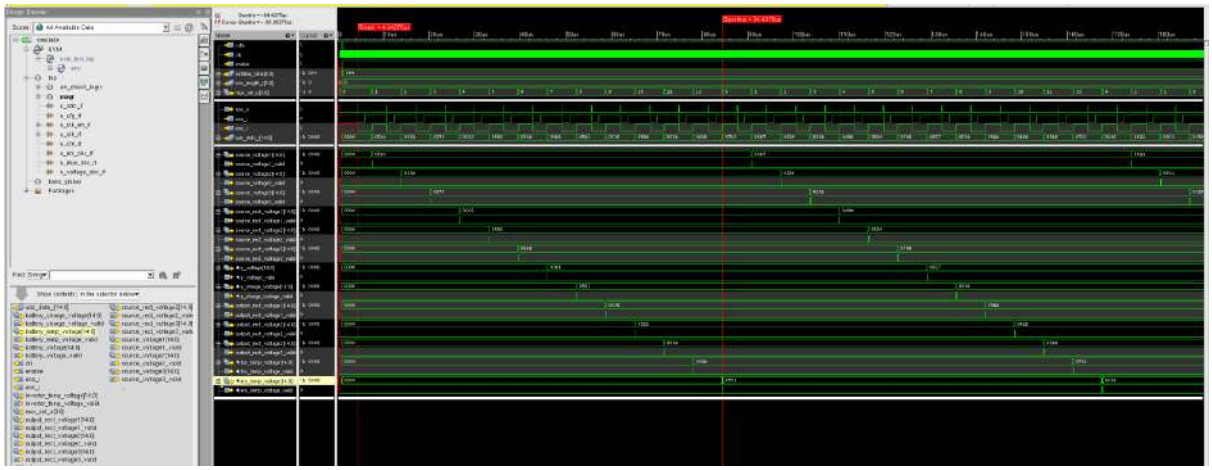


Рисунок 4.17 – Відображення вхідних та вихідних сигналів модулю логіки вимірювання

На рисунку 4.18 можна побачити, що після зміни сигналу вибору мультиплексору ми вичекали необхідну нам затримку переключення.

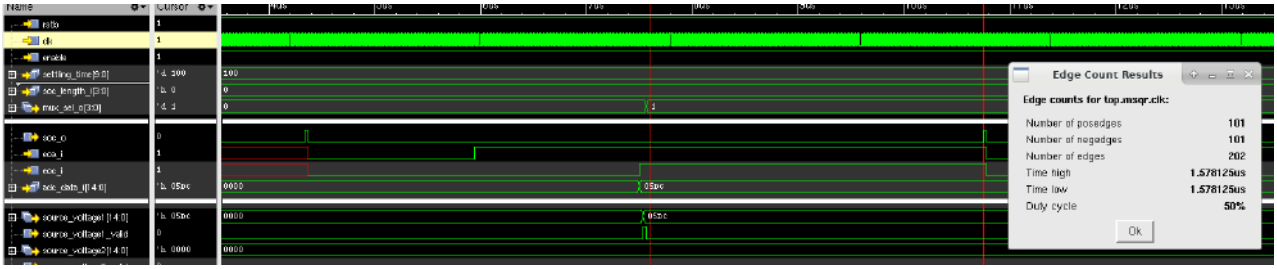


Рисунок 4.18 – Відображення вхідних та вихідних сигналів модулю логіки вимірювання

Тепер для підтвердження обґрунтування з розділу 3.1 змоделюємо ситуацію, коли затримка на переключення і затримка виміру АЦП в сумі

дадуть нам 6 мкс, та оцінимо кількість вимірів, які будуть виконані приблизно за 20 мс.

На рисунку 4.19 ми можемо побачити, що за відрізок часу який приближається до 20 мс ми отримали 252 відліки сигналу.

Відмініcть у 4 відліки викликана наявністю невеликої затримки викликаною логікою переключень скінченого автомату та точністю обробки симулятору, які в теоретичних розрахунках не було враховано, тому ми можемо зробити висновок, що концепт вимірювання працює належним чином.

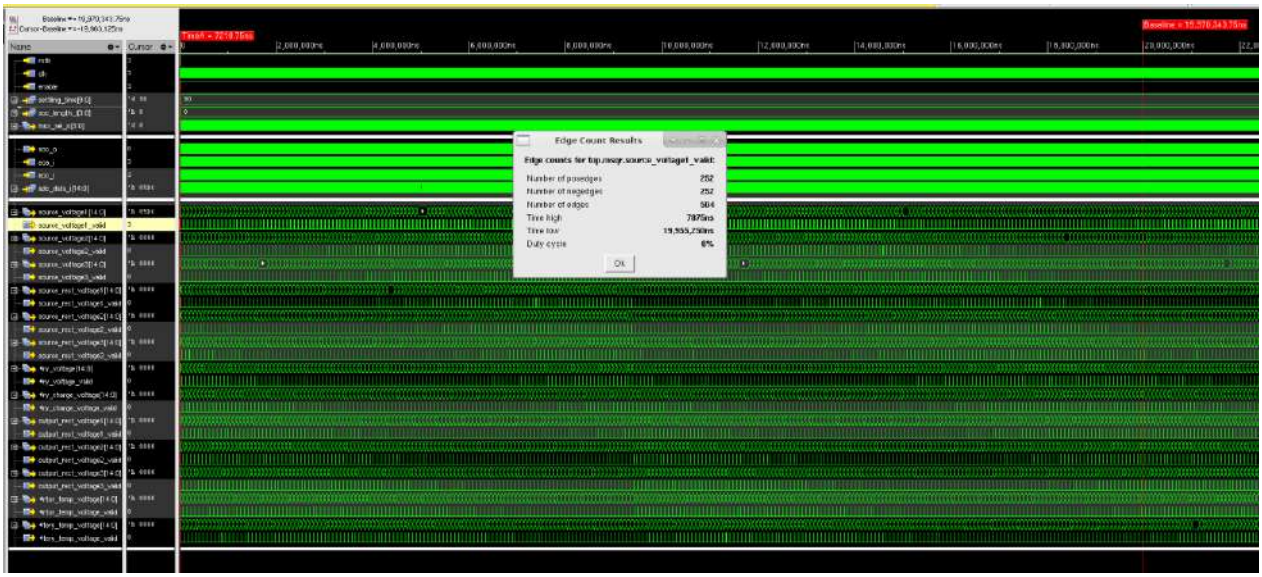


Рисунок 4.19 – Кількість вимірювань за час близький до 20 мс

На рисунку 4.20 ми можемо побачити відображення структурної схеми описаного нами модулю логіки вимірювання створеного за допомогою функції «Schematic tracer» застосунку Simvision.

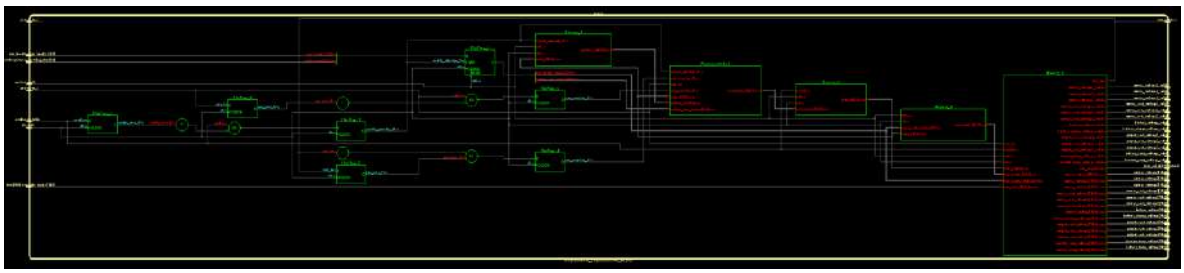


Рисунок 4.21 – Структурна схема модулю логіки вимірювання

Завершуємо моделювання перевіркою концептуальної функціональності – комбінації двох модулів.

На рисунку 4.22 відображена перевірка виміряної напруги на відсутність сконфігурованого рівня, сигналізація помилки та подальше її очищення.

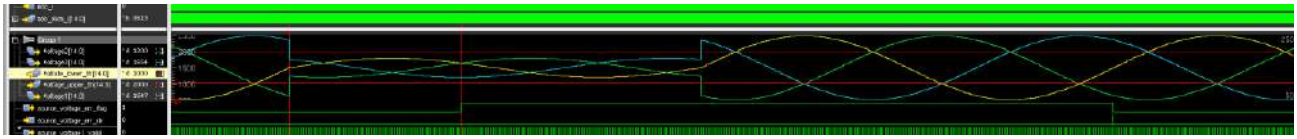


Рисунок 4.22 – Відображення сигналів взаємодії логіки вимірювання і логіки перевірки

На рисунку 4.23 відображена перевірка виміряних напруг на пере/недо живлення, сигналізація помилки та подальше її очищення.

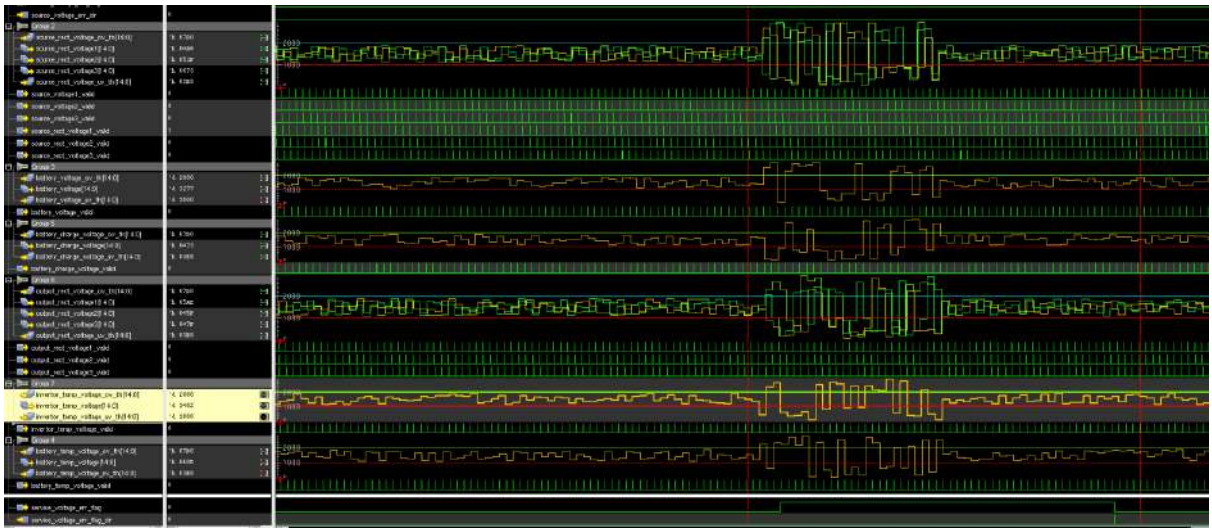


Рисунок 4.23 – Відображення сигналів взаємодії логіки вимірювання і логіки перевірки

В результаті моделювання, ми змогли підтвердити, що дизайн функціонує правильно і виконує всі поставлені до нього задачі.

ВИСНОВОК

У рамках цієї магістерської було запропоновано методику для створення цифрового дизайну, що дозволяють розробляти цифрові мікроелектронні функціональні вузли. Дана методика включає в себе етапи : обґрунтування параметрів системи, створення концепції роботи, розробка архітектури системи, створення модулі системи та моделювання спроектованої системи.

За визначеною методикою, використовуючи принципи сучасного цифрового проектування, було розроблено комплексне рішення для вимірювання напруги в системах резервного живлення, що дозволяє мікроконтролеру делегувати задачу вимірювання напруги даній цифровій схемі.

Була запропонована архітектура системи, що включає в себе кілька ключових компонентів, таких як аналого-цифровий перетворювач, мультиплексор, логіку вимірювання та перевірки, а також внутрішні регістри для зберігання даних та налаштувань.

Важливою частиною роботи стало створення RTL дизайну з використанням мови Verilog, що дозволило ефективно відобразити заплановану архітектуру у вигляді програмованої логіки.

Для перевірки функціональності та працездатності розробленої системи було використано методологію UVM (Universal Verification Methodology), яка дозволила створити ефективне тестове середовище. Завдяки UVM, було можливо перевірити аспекти системи, включаючи взаємодію між різними компонентами і відповідність специфікаціям.

Підтвердження працездатності розробленого технічного рішення за допомогою симуляції показує, що запропонований дизайн відповідає поставленим вимогам і може бути ефективно впроваджений у системи резервного живлення. Це не лише забезпечує надійність і точність вимірювань, але й підвищує загальну ефективність системи.

Також, продемонстровано важливе значення глибокого аналізу вимог до системи та важливість інтеграції цифрового дизайну. Впроваджені техніки і підходи до проектування і тестування підтвердили, що розроблена система є надійною, гнучкою та адаптивною до потреб користувачів.

Ця робота відкриває шлях для подальших досліджень у сфері цифрового дизайну систем вимірювання та контролю, пропонуючи підходи та рішення, які можуть бути адаптовані для інших застосувань у мікроелектроніці та суміжних галузях.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. *Схемотехніка джерела безперебійного живлення N-Power SVP-625* [Електронний ресурс]. – Режим доступу : https://www.radioradar.net/radiofan/power_supply/n_power_svp_625_uninterruptible_power_supply_circuitry.html – 10.01.2024. - Назва з екрану.
2. *Конструкція і ремонт джерел безреребійного живлення фірми APC* [Електронний ресурс]. – Режим доступу : https://electro-tech.narod.ru/schematics/power/ups/APC_Smart-UPS_450-1500_Back-UPS_250-600.pdf – 10.01.2024. - Назва з екрану.
3. *Analog to Digital Converters (ADC)* [Електронний ресурс]. – Режим доступу: https://www.digikey.ca/en/products/filter/data-acquisition/analog-to-digital-converters-adc/700 – 10.01.2024. - Назва з екрану.
4. *Analog Switches, Multiplexer, Demultiplexers* [Електронний ресурс]. – Режим доступу: https://www.digikey.ca/en/products/filter/analog-switches-multiplexers-demultiplexers/747– 10.01.2024. - Назва з екрану.
5. *Unit – I – Digital design using Verilog HDL-SECA3021* [Електронний ресурс]. – Режим доступу: https://www.studocu.com/in/document/sathyabama-institute-of-science-and-technology/electrical-and-electronics-engineering/seca3021/57200340– 10.01.2024. - Назва з екрану
6. Godse A. Digital Design using Verilog HDL. Conceptual approach / Godse A. & Godse D. // Pune, India : Technical Publications, 2020. - 250 p. ISBN: 978-93-332-2337-9
7. *RTL Verilog URL* [Електронний ресурс]. – Режим доступу: https://www.doulos.com/knowhow/verilog/rtl-verilog/ – 10.01.2024. - Назва з екрану

8. Navabi Z. Verilog Digital System Design : Analysis / Navabi Z. // Blacklick, OH, USA : McGraw-Hill Companies, The, 1999. - 477 p. ISBN: 0-07-047164-9
9. Floyd T. Digital Fundamentals / Floyd T. // London : Pearson Education, 2015. - 953 p. ISBN 10: 1-292-07598-822:24
10. *How to write FST?* [Електронний ресурс]. – Режим доступу: https://www.asic-world.com/tidbits/verilog_fsm.html – 10.01.2024. - Назва з екрану
11. *Xcelium logic Simulator* [Електронний ресурс]. – Режим доступу: :https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html – 10.01.2024. - Назва з екрану
12. Bergeron J. Writing Testbenches using SystemVerilog / Bergeron J. // NY, USA : Springer, 2006. - 438 p. ISBN-10 : 1846280230
13. *Defining the verification environment* [Електронний ресурс]. – Режим доступу: https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment/ – 10.01.2024. - Назва з екрану
14. *UVM Tutorial for Candy Lovers* [Електронний ресурс]. – Режим доступу: https://cluelogic.com/2011/07/uvm-tutorial-for-candy-lovers-overview/ – 10.01.2024. - Назва з екрану
15. *Simvision tutorial* [Електронний ресурс]. – Режим доступу: https://www.webpages.uidaho.edu/~jfrenzel/440/Handouts/Cadence/SimVisionIntro.pdf – 10.01.2024. - Назва з екрану

ДОДАТОК А

```
module measurement_sequencer #(parameter adc_data_width = 15, parameter settling_width = 10,
parameter soc_length_width = 4)(clk, rstb,
enable,
mux_sel_o,
settling_time,
source_voltage1,
source_voltage2,
source_voltage3,
source_voltage1_valid,
source_voltage2_valid,
source_voltage3_valid,
source_rect_voltage1,
source_rect_voltage2,
source_rect_voltage3,
source_rect_voltage1_valid,
source_rect_voltage2_valid,
source_rect_voltage3_valid,
battery_voltage,
battery_voltage_valid,
battery_charge_voltage,
battery_charge_voltage_valid,
output_rect_voltage1,
output_rect_voltage1_valid,
output_rect_voltage2,
output_rect_voltage2_valid,
output_rect_voltage3,
output_rect_voltage3_valid,
invertor_temp_voltage,
invertor_temp_voltage_valid,
battery_temp_voltage,
battery_temp_voltage_valid,
soc_length_i,
soc_o,
eoa_i,
eoc_i,
adc_data_i );

input    clk;
input     rstb;
input     enable;
output reg [3:0]   mux_sel_o;
input  reg [settling_width - 1 : 0]   settling_time;
output reg [adc_data_width - 1 : 0]   source_voltage1;
```

```
output reg [adc_data_width - 1 : 0]   source_voltage2;
output reg [adc_data_width - 1 : 0]   source_voltage3;
output reg                            source_voltage1_valid;
output reg                            source_voltage2_valid;
output reg                            source_voltage3_valid;
output reg [adc_data_width - 1 : 0]   source_rect_voltage1; //rectified
output reg [adc_data_width - 1 : 0]   source_rect_voltage2;
output reg [adc_data_width - 1 : 0]   source_rect_voltage3;
output reg                            source_rect_voltage1_valid;
output reg                            source_rect_voltage2_valid;
output reg                            source_rect_voltage3_valid;
output reg [adc_data_width - 1 : 0]   battery_voltage;
output reg                          battery_voltage_valid;
output reg [adc_data_width - 1 : 0]   battery_charge_voltage;
output reg                            battery_charge_voltage_valid;
output reg [adc_data_width - 1 : 0]   output_rect_voltage1;
output reg                            output_rect_voltage1_valid;
output reg [adc_data_width - 1 : 0]   output_rect_voltage2;
output reg                            output_rect_voltage2_valid;
output reg [adc_data_width - 1 : 0]   output_rect_voltage3;
output reg                            output_rect_voltage3_valid;
output reg [adc_data_width - 1 : 0]   invertor_temp_voltage;
output reg                            invertor_temp_voltage_valid;
output reg [adc_data_width - 1 : 0]   battery_temp_voltage;
output reg                          battery_temp_voltage_valid;
input  [soc_length_width - 1 : 0]     soc_length_i;
output reg  soc_o;
input          eoa_i;
input          eoc_i;
input [adc_data_width - 1 : 0]        adc_data_i;

typedef enum reg [2:0] { IDLE = 0, WAIT_SETTLING = 1, SEND_SOC = 2, WAIT_EOC = 3,
MUX_CHANGE = 4} state;
state state_ff;
state next_state_ff;
reg [soc_length_width  : 0 ]        soc_length_ff;
reg [soc_length_width - 1 : 0]     soc_length_latched;
reg [settling_width - 1 : 0]       settling_time_latched;
reg enable_posedge_ff;
reg enable_prev_ff;

always_ff @(posedge clk)
begin
   enable_prev_ff                          <= enable;
   enable_posedge_ff       <= (enable && !enable_prev_ff);
```

```
end

always @(posedge clk, negedge rstb)
begin
   if(!rstb) begin
      soc_length_latched <= 0;
      settling_time_latched <= 0;
   end else begin
      if(enable_posedge_ff) begin
         soc_length_latched <= soc_length_i;
         settling_time_latched <= settling_time;
      end
   end
end

reg soc_prev_ff;
reg soc_negedge_ff;

reg eoc_prev_ff;
reg eoc_posedge_ff;

always_ff @(posedge clk)
begin
soc_prev_ff                    <= soc_o;
soc_negedge_ff      <= (!soc_o && soc_prev_ff);
end

always_ff @(posedge clk)
begin
   eoc_prev_ff                 <= eoc_i;
   eoc_posedge_ff      <= (eoc_i && !eoc_prev_ff);
end

always @(posedge clk, negedge rstb)
begin
   if(!rstb) begin
      state_ff <= IDLE;
   end else begin
      if(!enable) begin
         state_ff <= IDLE;
      end
      else begin
         state_ff <= next_state_ff;
      end
   end
```

```
end

reg [settling_width - 1 : 0] settling_cnt;

always @(posedge clk, negedge rstb)
begin
   if(!rstb) begin
      settling_cnt <= 0;
   end
   else begin
      if(state_ff == WAIT_SETTLING && !enable_posedge_ff)
      begin
         settling_cnt <= settling_cnt + 1;
      end
      else begin
         settling_cnt <= 0;
      end
      end
end

always @(posedge clk, negedge rstb)
begin
   if(!rstb) begin
      soc_length_ff <= 0;
   end
   else begin
      if(state_ff == SEND_SOC)
      begin
         if(soc_length_ff <= soc_length_latched)
            soc_length_ff <= soc_length_ff + 1;
         end else begin
            soc_length_ff <= 0;
         end
      end
end

always_comb begin: FSM_COMBO
   next_state_ff = IDLE;

   case(state_ff)
   IDLE : begin
      if(enable_posedge_ff)
         next_state_ff = WAIT_SETTLING;
   end
   WAIT_SETTLING : begin
```

```
        if(settling_cnt == settling_time_latched && !enable_posedge_ff) //-1
            next_state_ff = SEND_SOC;
        else
            next_state_ff = WAIT_SETTLING;
    end

    SEND_SOC : begin
        if(soc_negedge_ff)
            next_state_ff = WAIT_EOC;
        else
            next_state_ff = SEND_SOC;
    end

    WAIT_EOC : begin
        if(eoc_i)
            next_state_ff = MUX_CHANGE;
        else
            next_state_ff = WAIT_EOC;
    end

    MUX_CHANGE : begin
        if(eoc_posedge_ff == 1)
            next_state_ff = WAIT_SETTLING;
        else
            next_state_ff = MUX_CHANGE;
    end

    default : next_state_ff = IDLE;
    endcase
end

always @(posedge clk, negedge rstb)
begin
    if(!rstb) begin
        mux_sel_o <= 0;
        soc_o <= 0;
        source_voltage1_valid <= 0;
        source_voltage2_valid <= 0;
        source_voltage3_valid <= 0;
        source_rect_voltage1_valid <= 0;
        source_rect_voltage2_valid <= 0;
        source_rect_voltage3_valid <= 0;
        battery_voltage_valid <= 0;
        battery_charge_voltage_valid <= 0;
        output_rect_voltage1_valid <= 0;
```

```
    output_rect_voltage2_valid <= 0;
    output_rect_voltage3_valid <= 0;
    battery_temp_voltage_valid<= 0;
    invertor_temp_voltage_valid <= 0;
    source_voltage1 <= 0;
    source_voltage2 <= 0;
    source_voltage3 <= 0;
    source_rect_voltage1    <= 0;
    source_rect_voltage2    <= 0;
    source_rect_voltage3    <= 0;
    battery_voltage <= 0;
    battery_charge_voltage  <= 0;
    output_rect_voltage1    <= 0;
    output_rect_voltage2    <= 0;
    output_rect_voltage3    <= 0;
    invertor_temp_voltage   <= 0;
    battery_temp_voltage    <= 0;
end else begin
    if(!enable) begin
        mux_sel_o <= 0;
        soc_o <= 0;
        source_voltage1_valid <= 0;
        source_voltage2_valid <= 0;
        source_voltage3_valid <= 0;
        source_rect_voltage1_valid <= 0;
        source_rect_voltage2_valid <= 0;
        source_rect_voltage3_valid <= 0;
        battery_voltage_valid <= 0;
        battery_charge_voltage_valid <= 0;
        output_rect_voltage1_valid <= 0;
        output_rect_voltage2_valid <= 0;
        output_rect_voltage3_valid <= 0;
        battery_temp_voltage_valid<= 0;
        invertor_temp_voltage_valid <= 0;
    end
    else begin
        case(state_ff)

        IDLE:begin
            mux_sel_o <= 0;
            soc_o <= 0;
            source_voltage1_valid <= 0;
            source_voltage2_valid <= 0;
            source_voltage3_valid <= 0;
            source_rect_voltage1_valid <= 0;
```

```
    source_rect_voltage2_valid <= 0;
    source_rect_voltage3_valid <= 0;
    battery_voltage_valid <= 0;
    battery_charge_voltage_valid <= 0;
    output_rect_voltage1_valid <= 0;
    output_rect_voltage2_valid <= 0;
    output_rect_voltage3_valid <= 0;
    battery_temp_voltage_valid<= 0;
    invertor_temp_voltage_valid <= 0;
end

WAIT_SETTLING : begin
    source_voltage1_valid <= 0;
    source_voltage2_valid <= 0;
    source_voltage3_valid <= 0;
    source_rect_voltage1_valid <= 0;
    source_rect_voltage2_valid <= 0;
    source_rect_voltage3_valid <= 0;
    battery_voltage_valid <= 0;
    battery_charge_voltage_valid <= 0;
    output_rect_voltage1_valid <= 0;
    output_rect_voltage2_valid <= 0;
    output_rect_voltage3_valid <= 0;
    battery_temp_voltage_valid<= 0;
    invertor_temp_voltage_valid <= 0;
end

SEND_SOC : begin
    if(soc_length_ff < soc_length_latched + 1)
    begin
        soc_o <= 1;
    end else begin
        soc_o <= 0;
    end
end

WAIT_EOC : begin

    if(eoc_i)
    begin
        if(mux_sel_o == 4'b0000)
    begin
        source_voltage1_valid <= 1;
        source_voltage1 <= adc_data_i;
    end
```

```
else if(mux_sel_o == 4'b0001)
begin
   source_voltage2_valid <= 1;
   source_voltage2 <= adc_data_i;
end
else if(mux_sel_o == 4'b0010)
begin
   source_voltage3_valid <= 1;
   source_voltage3 <= adc_data_i;
end
else if(mux_sel_o == 4'b0011) begin
   source_rect_voltage1_valid <= 1;
   source_rect_voltage1 <= adc_data_i;
end
else if(mux_sel_o == 4'b0100) begin
   source_rect_voltage2_valid <= 1;
   source_rect_voltage2 <= adc_data_i;
end
else if(mux_sel_o == 4'b0101) begin
   source_rect_voltage3_valid <= 1;
   source_rect_voltage3 <= adc_data_i;
end
else if(mux_sel_o == 4'b0110)
begin
   battery_voltage_valid <= 1;
   battery_voltage <= adc_data_i;
end
else if(mux_sel_o == 4'b0111)
begin
   battery_charge_voltage_valid <= 1;
   battery_charge_voltage <= adc_data_i;
end
else if(mux_sel_o == 4'b1000)
begin
   output_rect_voltage1_valid <= 1;
   output_rect_voltage1 <= adc_data_i;
end
else if(mux_sel_o == 4'b1001)
begin
   output_rect_voltage2_valid <= 1;
   output_rect_voltage2 <= adc_data_i;
end
else if(mux_sel_o == 4'b1010)
begin
   output_rect_voltage3_valid <= 1;
```

```
        output_rect_voltage3 <= adc_data_i;
    end
    else if(mux_sel_o == 4'b1011)
    begin
        invertor_temp_voltage_valid <= 1;
        invertor_temp_voltage <= adc_data_i;
    end
    else if(mux_sel_o == 4'b1100)
    begin
        battery_temp_voltage_valid <= 1;
        battery_temp_voltage <= adc_data_i;
    end
    end
end

MUX_CHANGE : begin
    if(mux_sel_o  < 4'b1100)
    begin
        mux_sel_o <= mux_sel_o + 1;
    end
    else begin
        mux_sel_o <= 0;
    end

    source_voltage1_valid <= 0;
    source_voltage2_valid <= 0;
    source_voltage3_valid <= 0;
    source_rect_voltage1_valid <= 0;
    source_rect_voltage2_valid <= 0;
    source_rect_voltage3_valid <= 0;
    battery_voltage_valid <= 0;
    battery_charge_voltage_valid <= 0;
    output_rect_voltage1_valid <= 0;
    output_rect_voltage2_valid <= 0;
    output_rect_voltage3_valid <= 0;
    battery_temp_voltage_valid<= 0;
    invertor_temp_voltage_valid <= 0;

end

default: begin
    source_voltage1_valid <= 0;
    source_voltage2_valid <= 0;
    source_voltage3_valid <= 0;
    source_rect_voltage1_valid <= 0;
```

```
            source_rect_voltage2_valid <= 0;
            source_rect_voltage3_valid <= 0;
            battery_voltage_valid <= 0;
            battery_charge_voltage_valid <= 0;
            output_rect_voltage1_valid <= 0;
            output_rect_voltage2_valid <= 0;
            output_rect_voltage3_valid <= 0;
            battery_temp_voltage_valid<= 0;
            invertor_temp_voltage_valid <= 0;
          end
        endcase
    end
    end
end
```

ДОДАТОК Б

```
module voltage_monitor #( parameter timeout_width = 7, parameter data_width = 15)
(
    input enable,
    input clk,
    input clk_en_100us,
    input rstb,
    input err_clear,
          input [data_width - 1 : 0] source_voltage,
          input [data_width - 1 : 0] source_voltage_lower_th,
          input [data_width - 1 : 0] source_voltage_upper_th,
          input [timeout_width - 1 : 0] source_voltage_timeout,
    output reg error
);

    reg [timeout_width - 1 : 0] counter;

    // Voltage condition check
    wire voltage_condition_met = (source_voltage >= source_voltage_upper_th) ||
                                 (source_voltage <= source_voltage_lower_th);

    always @(posedge clk or negedge rstb) begin
        if (!rstb) begin
            // Reset logic
            counter <= 0;
            error <= 1'b0;
        end
        else if (enable) begin
            if (voltage_condition_met || err_clear) begin
                // Reset the counter if voltage condition is met or error is cleared
                counter <= 0;
                // Clear error when err_clear is high
                if (err_clear) begin
                    error <= 1'b0;
                end
            end
            else begin
                // Increment the counter when the condition is not met
                if (counter < source_voltage_timeout && clk_en_100us)
                begin
                    counter <= counter + 1;
                end
                else if (counter == source_voltage_timeout)
                begin
```

```
                // Set error high if timeout is reached
                error <= 1'b1;
            end
        end
    end
end

endmodule
```

ДОДАТОК В

```
module comparator_with_blanking #( parameter data_width = 15)  // Default value for data width
(
    input clk_en,
    input clk,
    input rstb,
    input [data_width - 1 : 0] voltage,
    input [data_width - 1 : 0] lower_th,
    input [data_width - 1 : 0] upper_th,
    input [3:0] blanking_time,  // 4-bit input for blanking time configuration (0 to 14)
    output reg cmp_result
);

    reg [4:0] counter;  // 4-bit counter to count up to 15

    always @(posedge clk or negedge rstb) begin
        if (!rstb) begin
            // Reset logic
            counter <= 4'd0;
            cmp_result <= 1'b0;
        end
        else if (clk_en) begin
            if ((voltage >= upper_th) || (voltage <= lower_th)) begin
                if (blanking_time == 0) begin
                    // Set cmp_result high immediately if blanking_time is 0
                    cmp_result <= 1'b1;
                end
                else if (counter < blanking_time) begin
                    counter <= counter + 1;
                end
                else begin
                    // Set cmp_result high if condition met for blanking_time + 1 cycles
                    cmp_result <= 1'b1;
                end
            end
            else begin
                counter <= 5'd0;
                cmp_result <= 1'b0;
            end
        end
    end

endmodule
```

ДОДАТОК Г

```
module error_check_logic #(parameter adc_data_width = 15, parameter timeout_width = 7)
(
    clk,
    rstb,
    clk_en_100us,
    enable,
    source_voltage1,
    source_voltage2,
    source_voltage3,
    source_voltage_err_flag,
    source_voltage_err_clr,
    source_voltate_lower_th,
    source_voltage_upper_th,
    source_voltage_timeout,

    source_rect_voltage1,
    source_rect_voltage2,
    source_rect_voltage3,
    source_rect_voltage1_valid,
    source_rect_voltage2_valid,
    source_rect_voltage3_valid,
    source_rect_voltage_ov_th,
    source_rect_voltage_uv_th,

    battery_voltage,
    battery_voltage_valid,
    battery_voltage_ov_th,
    battery_voltage_uv_th,

    battery_charge_voltage,
    battery_charge_voltage_valid,
    battery_charge_voltage_ov_th,
    battery_charge_voltage_uv_th,

    output_rect_voltage1,
    output_rect_voltage2,
    output_rect_voltage3,
    output_rect_voltage1_valid,
    output_rect_voltage2_valid,
    output_rect_voltage3_valid,
    output_rect_voltage_ov_th,
    output_rect_voltage_uv_th,
```

```
    invertor_temp_voltage,
    invertor_temp_voltage_valid,
    invertor_temp_voltage_ov_th,
    invertor_temp_voltage_uv_th,

    battery_temp_voltage,
    battery_temp_voltage_valid,
    battery_temp_voltage_ov_th,
    battery_temp_voltage_uv_th,

    service_voltage_err_flag,
    service_voltage_err_flag_clr,

    blanking_time
);

input                       clk;
input                       rstb;
input                       enable;
input                       clk_en_100us;
input [adc_data_width - 1 : 0]  source_voltage1;
input [adc_data_width - 1 : 0]  source_voltage2;
input [adc_data_width - 1 : 0]  source_voltage3;
output                       source_voltage_err_flag;
input                       source_voltage_err_clr;
input [adc_data_width - 1 : 0]  source_voltate_lower_th;
input [adc_data_width - 1 : 0]  source_voltage_upper_th;
input [timeout_width - 1 : 0]   source_voltage_timeout;
input [adc_data_width - 1 : 0]  source_rect_voltage1;
input [adc_data_width - 1 : 0]  source_rect_voltage2;
input [adc_data_width - 1 : 0]  source_rect_voltage3;
input                       source_rect_voltage1_valid;
input                       source_rect_voltage2_valid;
input                       source_rect_voltage3_valid;
input [adc_data_width - 1 : 0]  source_rect_voltage_ov_th;
input [adc_data_width - 1 : 0]  source_rect_voltage_uv_th;
input [adc_data_width - 1 : 0]  battery_voltage;
input                       battery_voltage_valid;
input [adc_data_width - 1 : 0]  battery_voltage_ov_th;
input [adc_data_width - 1 : 0]  battery_voltage_uv_th;
input [adc_data_width - 1 : 0]  battery_charge_voltage;
input                       battery_charge_voltage_valid;
input [adc_data_width - 1 : 0]  battery_charge_voltage_ov_th;
input [adc_data_width - 1 : 0]  battery_charge_voltage_uv_th;
input [adc_data_width - 1 : 0]  output_rect_voltage1;
```

```
input [adc_data_width - 1 : 0]  output_rect_voltage2;
input [adc_data_width - 1 : 0]  output_rect_voltage3;
input                           output_rect_voltage1_valid;
input                           output_rect_voltage2_valid;
input                           output_rect_voltage3_valid;
input [adc_data_width - 1 : 0]  output_rect_voltage_ov_th;
input [adc_data_width - 1 : 0]  output_rect_voltage_uv_th;
input [adc_data_width - 1 : 0]  invertor_temp_voltage;
input                           invertor_temp_voltage_valid;
input [adc_data_width - 1 : 0]  invertor_temp_voltage_ov_th;
input [adc_data_width - 1 : 0]  invertor_temp_voltage_uv_th;
input [adc_data_width - 1 : 0]  battery_temp_voltage;
input                           battery_temp_voltage_valid;
input [adc_data_width - 1 : 0]  battery_temp_voltage_ov_th;
input [adc_data_width - 1 : 0]  battery_temp_voltage_uv_th;
output                           service_voltage_err_flag;
input                           service_voltage_err_flag_clr;
input [3:0]                       blanking_time;

wire source1_error_w;
wire source2_error_w;
wire source3_error_w;

voltage_monitor #(.timeout_width(timeout_width), .data_width(adc_data_width))
voltage_monitor_source1 (
.enable(enable),
.clk(clk),
.rstb(rstb),
.clk_en_100us(clk_en_100us),
.err_clear(source_voltage_err_clr),
.source_voltage(source_voltage1),
.source_voltage_lower_th(source_voltate_lower_th),
.source_voltage_upper_th(source_voltage_upper_th),
.source_voltage_timeout(source_voltage_timeout),
.error(source1_error_w)
);

voltage_monitor #(.timeout_width(timeout_width),
.data_width(adc_data_width))voltage_monitor_source2 (
.enable(enable),
.clk(clk),
.rstb(rstb),
.clk_en_100us(clk_en_100us),
.err_clear(source_voltage_err_clr),
.source_voltage(source_voltage2),
.source_voltage_lower_th(source_voltate_lower_th),
```

```
.source_voltage_upper_th(source_voltage_upper_th),
.source_voltage_timeout(source_voltage_timeout),
.error(source2_error_w)
);

voltage_monitor #(.timeout_width(timeout_width), .data_width(adc_data_width))
voltage_monitor_source3 (
.enable(enable),
.clk(clk),
.rstb(rstb),
.clk_en_100us(clk_en_100us),
.err_clear(source_voltage_err_clr),
.source_voltage(source_voltage3),
.source_voltage_lower_th(source_voltate_lower_th),
.source_voltage_upper_th(source_voltage_upper_th),
.source_voltage_timeout(source_voltage_timeout),
.error(source3_error_w)
);

assign source_voltage_err_flag = source1_error_w || source2_error_w || source3_error_w;

wire source_rect_voltage1_cmp_w;
wire source_rect_voltage2_cmp_w;
wire source_rect_voltage3_cmp_w;
wire battery_voltage_cmp_w;
wire battery_charge_voltage_cmp_w;
wire output_source_rect_voltage1_cmp_w;
wire output_source_rect_voltage2_cmp_w;
wire output_source_rect_voltage3_cmp_w;
wire invertor_temp_voltage_cmp_w;
wire battery_temp_voltage_cmp_w;

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_rect_voltage1 (
.clk_en(source_rect_voltage1_valid),
.clk(clk),
.rstb(rstb),
.voltage(source_rect_voltage1),
.lower_th(source_rect_voltage_uv_th),
.upper_th(source_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(source_rect_voltage1_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_rect_voltage2 (
.clk_en(source_rect_voltage2_valid),
```

```
.clk(clk),
.rstb(rstb),
.voltage(source_rect_voltage2),
.lower_th(source_rect_voltage_uv_th),
.upper_th(source_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(source_rect_voltage2_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_rect_voltage3 (
.clk_en(source_rect_voltage3_valid),
.clk(clk),
.rstb(rstb),
.voltage(source_rect_voltage3),
.lower_th(source_rect_voltage_uv_th),
.upper_th(source_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(source_rect_voltage3_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_battery_voltage (
.clk_en(battery_voltage_valid),
.clk(clk),
.rstb(rstb),
.voltage(battery_voltage),
.lower_th(battery_voltage_uv_th),
.upper_th(battery_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(battery_voltage_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_battery_charge_voltage (
.clk_en(battery_charge_voltage_valid),
.clk(clk),
.rstb(rstb),
.voltage(battery_charge_voltage),
.lower_th(battery_charge_voltage_uv_th),
.upper_th(battery_charge_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(battery_charge_voltage_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_output_rect_voltage1 (
.clk_en(output_rect_voltage1_valid),
.clk(clk),
```

```
.rstb(rstb),
.voltage(output_rect_voltage1),
.lower_th(output_rect_voltage_uv_th),
.upper_th(output_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(output_source_rect_voltage1_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_output_rect_voltage2 (
.clk_en(output_rect_voltage2_valid),
.clk(clk),
.rstb(rstb),
.voltage(output_rect_voltage2),
.lower_th(output_rect_voltage_uv_th),
.upper_th(output_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(output_source_rect_voltage2_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_source_output_rect_voltage3 (
.clk_en(output_rect_voltage3_valid),
.clk(clk),
.rstb(rstb),
.voltage(output_rect_voltage3),
.lower_th(output_rect_voltage_uv_th),
.upper_th(output_rect_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(output_source_rect_voltage3_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_invertor_temp_voltage (
.clk_en(invertor_temp_voltage_valid),
.clk(clk),
.rstb(rstb),
.voltage(invertor_temp_voltage),
.lower_th(invertor_temp_voltage_uv_th),
.upper_th(invertor_temp_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(invertor_temp_voltage_cmp_w)
);

comparator_with_blanking #(.data_width(adc_data_width)) comparator_battery_temp_voltage
(
.clk_en(battery_temp_voltage_valid),
.clk(clk),
```

```
.rstb(rstb),
.voltage(battery_temp_voltage),
.lower_th(battery_temp_voltage_uv_th),
.upper_th(battery_temp_voltage_ov_th),
.blanking_time(blanking_time),
.cmp_result(battery_temp_voltage_cmp_w)
);

   reg service_error_ff;

   always @(posedge clk or negedge rstb) begin
      if (!rstb) begin
         service_error_ff <= 0;
      end
      else if(enable) begin
         if(service_voltage_err_flag_clr) begin
            service_error_ff <= 0;
         end else begin
            service_error_ff <= service_error_ff | ( source_rect_voltage1_cmp_w ||
            source_rect_voltage2_cmp_w ||
            source_rect_voltage3_cmp_w ||
            battery_voltage_cmp_w          ||
            battery_charge_voltage_cmp_w ||
            output_source_rect_voltage1_cmp_w ||
            output_source_rect_voltage2_cmp_w ||
            output_source_rect_voltage3_cmp_w ||
            invertor_temp_voltage_cmp_w ||
            battery_temp_voltage_cmp_w);
            end
      end
   end

   assign service_voltage_err_flag = service_error_ff;

endmodule
```

ДОДАТОК Д

```
//Combination of msqr and error check logic tb
module top;

    import uvm_pkg::*;

    import dig_ms_top_pkg::*;
    import dip_dig_ms_top_tb_pkg::*;

    logic clk;
    logic reset_n;

    clock_if u_clk_if();

    assign clk = u_clk_if.clk;
    assign reset_n = u_clk_if.rstb;

    clock_en_if u_clk_en_if(.clk(clk));

    measurement_sequencer msqr (
        .clk(clk),
        .rstb(reset_n),
        .enable(u_cfg_if.data.enable),
        .mux_sel_o(u_mux_obs_if.data),
        .settling_time(u_cfg_if.data.settling_time),
        .source_voltage1(),
        .source_voltage2(),
        .source_voltage3(),
        .source_voltage1_valid(),
        .source_voltage2_valid(),
        .source_voltage3_valid(),
        .source_rect_voltage1(),
        .source_rect_voltage2(),
        .source_rect_voltage3(),
        .source_rect_voltage1_valid(),
        .source_rect_voltage2_valid(),
        .source_rect_voltage3_valid(),
        .battery_voltage(),
        .battery_voltage_valid(),
        .battery_charge_voltage(),
        .battery_charge_voltage_valid(),
        .output_rect_voltage1(),
        .output_rect_voltage1_valid(),
        .output_rect_voltage2(),
```

```
    .output_rect_voltage2_valid(),
    .output_rect_voltage3(),
    .output_rect_voltage3_valid(),
    .invertor_temp_voltage(),
    .invertor_temp_voltage_valid(),
    .battery_temp_voltage(),
    .battery_temp_voltage_valid(),
    .soc_length_i(u_cfg_if.data.soc_length),
    .soc_o(u_adc_if.soc),
    .eoa_i(u_adc_if.eoa),
    .eoc_i(u_adc_if.eoc),
    .adc_data_i(u_adc_if.data));

error_check_logic err_check_logic (
    .clk(clk),
    .rstb(reset_n),
    .clk_en_100us(u_clk_en_if.clk_en),
    .enable(u_cfg_if.data.enable),
    .source_voltage1(),
    .source_voltage2(),
    .source_voltage3(),
    .source_voltage_err_flag(),
    .source_voltage_err_clr(u_ctrl_if.data.source_voltage_err_clr),
    .source_voltate_lower_th(u_cfg_if.data.source_voltage_lower_th),
    .source_voltage_upper_th(u_cfg_if.data.source_voltage_upper_th),
    .source_voltage_timeout(u_cfg_if.data.source_voltage_timeout),
    .source_rect_voltage1(),
    .source_rect_voltage2(),
    .source_rect_voltage3(),
    .source_rect_voltage1_valid(),
    .source_rect_voltage2_valid(),
    .source_rect_voltage3_valid(),
    .source_rect_voltage_ov_th(u_cfg_if.data.source_rect_voltage_ov_th),
    .source_rect_voltage_uv_th(u_cfg_if.data.source_rect_voltage_uv_th),
    .battery_voltage(),
    .battery_voltage_valid(),
    .battery_voltage_ov_th(u_cfg_if.data.battery_voltage_ov_th),
    .battery_voltage_uv_th(u_cfg_if.data.battery_voltage_uv_th),
    .battery_charge_voltage(),
    .battery_charge_voltage_valid(),
    .battery_charge_voltage_ov_th(u_cfg_if.data.battery_charge_voltage_ov_th),
    .battery_charge_voltage_uv_th(u_cfg_if.data.battery_charge_voltage_uv_th),
    .output_rect_voltage1(),
    .output_rect_voltage2(),
    .output_rect_voltage3(),
```

```
    .output_rect_voltage1_valid(),
    .output_rect_voltage2_valid(),
    .output_rect_voltage3_valid(),
    .output_rect_voltage_ov_th(u_cfg_if.data.output_rect_voltage_ov_th),
    .output_rect_voltage_uv_th(u_cfg_if.data.output_rect_voltage_uv_th),
    .invertor_temp_voltage(),
    .invertor_temp_voltage_valid(),
    .invertor_temp_voltage_ov_th(u_cfg_if.data.invertor_temp_voltage_ov_th),
    .invertor_temp_voltage_uv_th(u_cfg_if.data.invertor_temp_voltage_uv_th),
    .battery_temp_voltage(),
    .battery_temp_voltage_valid(),
    .battery_temp_voltage_ov_th(u_cfg_if.data.battery_temp_voltage_ov_th),
    .battery_temp_voltage_uv_th(u_cfg_if.data.battery_temp_voltage_uv_th),
    .service_voltage_err_flag(),
    .service_voltage_err_flag_clr(u_ctrl_if.data.service_voltage_err_flag_clr),
    .blanking_time(u_cfg_if.data.blanking_time)
);

assign err_check_logic.source_voltage1 = msqr.source_voltage1;
assign err_check_logic.source_voltage2 = msqr.source_voltage2;
assign err_check_logic.source_voltage3 = msqr.source_voltage3;
assign err_check_logic.source_rect_voltage1 = msqr.source_rect_voltage1;
assign err_check_logic.source_rect_voltage2 = msqr.source_rect_voltage2;
assign err_check_logic.source_rect_voltage3 = msqr.source_rect_voltage3;
assign err_check_logic.source_rect_voltage1_valid = msqr.source_rect_voltage1_valid;
assign err_check_logic.source_rect_voltage2_valid = msqr.source_rect_voltage2_valid;
assign err_check_logic.source_rect_voltage3_valid = msqr.source_rect_voltage3_valid;
assign err_check_logic.battery_voltage = msqr.battery_voltage;
assign err_check_logic.battery_voltage_valid = msqr.battery_voltage_valid;
assign err_check_logic.battery_charge_voltage = msqr.battery_charge_voltage;
assign err_check_logic.battery_charge_voltage_valid = msqr.battery_charge_voltage_valid;
assign err_check_logic.output_rect_voltage1 = msqr.output_rect_voltage1;
assign err_check_logic.output_rect_voltage2 = msqr.output_rect_voltage2;
assign err_check_logic.output_rect_voltage3 = msqr.output_rect_voltage3;
assign err_check_logic.output_rect_voltage1_valid = msqr.output_rect_voltage1_valid;
assign err_check_logic.output_rect_voltage2_valid = msqr.output_rect_voltage2_valid;
assign err_check_logic.output_rect_voltage3_valid = msqr.output_rect_voltage3_valid;
assign err_check_logic.invertor_temp_voltage = msqr.invertor_temp_voltage;
assign err_check_logic.invertor_temp_voltage_valid = msqr.invertor_temp_voltage_valid;
assign err_check_logic.battery_temp_voltage = msqr.battery_temp_voltage;
assign err_check_logic.battery_temp_voltage_valid = msqr.battery_temp_voltage_valid;


bae_wire_if #(.data_t(cfg_data_t)) u_cfg_if(
    .clk(clk),
```

```
        .reset_n(reset_n)
    );

    wire_if #(.data_t(ctrl_data_t)) u_ctrl_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(voltage_obs_data_t)) u_voltage_obs_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(err_obs_data_t)) u_err_obs_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(mux_obs_data_t)) u_mux_obs_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    adc_if #(.data_t(adc_data_t)) u_adc_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    initial begin
uvm_config_db #(virtual wire_if #(cfg_data_t))::set(uvm_root::get(), "uvm_test_top", "cfg_vif",
u_cfg_if);

uvm_config_db #(virtual wire_if #(ctrl_data_t))::set(uvm_root::get(), "uvm_test_top", "ctrl_vif",
u_ctrl_if);

uvm_config_db #(virtual wire_if #(voltage_obs_data_t))::set(uvm_root::get(), "uvm_test_top",
"voltage_obs_vif", u_voltage_obs_if);

uvm_config_db #(virtual wire_if #(err_obs_data_t))::set(uvm_root::get(), "uvm_test_top", "err_obs_vif",
u_err_obs_if);

uvm_config_db #(virtual wire_if #(mux_obs_data_t))::set(uvm_root::get(), "uvm_test_top",
"mux_obs_vif", u_mux_obs_if);

uvm_config_db #(virtual clock_if)::set(uvm_root::get(), "uvm_test_top", "clk_vif", u_clk_if);
```

```
uvm_config_db #(virtual clock_en_if)::set(uvm_root::get(), "uvm_test_top", "clk_en_vif", u_clk_en_if);

uvm_config_db #(virtual  adc_if#(adc_data_t))::set(uvm_root::get(), "uvm_test_top", "adc_vif",
u_adc_if);

   end

   initial begin
      $timeformat(-9, 1, "ns", 10);
      run_test();
   end

endmodule

//comparator with blanking top tb
module top;

   import uvm_pkg::*;

   import comp_with_blank_pkg::*;
   import dip_comp_with_blank_tb_pkg::*;

   wire clk;
   wire reset_n;

   clock_if u_clk_if();

   assign clk = u_clk_if.clk;
   assign reset_n = u_clk_if.rstb;

   clock_en_if u_clk_en_if(.clk(clk));

   comparator_with_blanking comp_with_blank(.clk(clk),
                                    .rstb(reset_n),
                                    .clk_en(u_clk_en_if.clk_en),
                                    .voltage(u_adc_if.data.adc_data),
                                    .blanking_time(u_cfg_if.data.blanking_time),
                                    .lower_th(u_cfg_if.data.uv_th),
                                    .upper_th(u_cfg_if.data.ov_th),
                                    .cmp_result(u_obs_if.data));


   wire_if #(.data_t(adc_data_t)) u_adc_if(
      .clk(clk),
      .reset_n(reset_n)
```

```
    );
        wire_if #(.data_t(config_data_t)) u_cfg_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(logic)) u_obs_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    initial begin
 uvm_config_db #(virtual wire_if #(adc_data_t))::set(uvm_root::get(), "uvm_test_top", "adc_vif",
u_adc_if);

uvm_config_db #(virtual wire_if #(config_data_t))::set(uvm_root::get(), "uvm_test_top", "cfg_vif",
u_cfg_if);

uvm_config_db #(virtual wire_if #(logic))::set(uvm_root::get(), "uvm_test_top", "obs_vif", u_obs_if);

uvm_config_db #(virtual clock_if)::set(uvm_root::get(), "uvm_test_top", "clk_vif", u_clk_if);

uvm_config_db #(virtual clock_en_if)::set(uvm_root::get(), "uvm_test_top", "clk_en_vif", u_clk_en_if);

    end

    initial begin
        $timeformat(-9, 1, "ns", 10);
        run_test();
    end

endmodule

//error check logic top tb
module top;

    import uvm_pkg::*;
    import error_check_logic_pkg::*;
    import dip_error_check_logic_tb_pkg::*;
     logic clk;
    logic reset_n;

    clock_if u_clk_if();

    assign clk = u_clk_if.clk;
    assign reset_n = u_clk_if.rstb;
```

```
clock_en_if u_clk_en_if(.clk(clk));
.
error_check_logic err_check_logic (
    .clk(clk),
    .rstb(reset_n),
    .clk_en_100us(u_clk_en_if.clk_en),
    .enable(u_cfg_if.data.enable),
    .source_voltage1(u_adc_if.data.source_voltage1),
    .source_voltage2(u_adc_if.data.source_voltage2),
    .source_voltage3(u_adc_if.data.source_voltage3),
    .source_voltage_err_flag(),
    .source_voltage_err_clr(u_ctrl_if.data.source_voltage_err_clr),
    .source_voltate_lower_th(u_cfg_if.data.source_voltage_lower_th),
    .source_voltage_upper_th(u_cfg_if.data.source_voltage_upper_th),
    .source_voltage_timeout(u_cfg_if.data.source_voltage_timeout),
    .source_rect_voltage1(u_adc_if.data.source_rect_voltage1),
    .source_rect_voltage2(u_adc_if.data.source_rect_voltage2),
    .source_rect_voltage3(u_adc_if.data.source_rect_voltage3),
    .source_rect_voltage1_valid(u_adc_if.data.valid),
    .source_rect_voltage2_valid(u_adc_if.data.valid),
    .source_rect_voltage3_valid(u_adc_if.data.valid),
    .source_rect_voltage_ov_th(u_cfg_if.data.source_rect_voltage_ov_th),
    .source_rect_voltage_uv_th(u_cfg_if.data.source_rect_voltage_uv_th),
    .battery_voltage(u_adc_if.data.battery_voltage),
    .battery_voltage_valid(u_adc_if.data.valid),
    .battery_voltage_ov_th(u_cfg_if.data.battery_voltage_ov_th),
    .battery_voltage_uv_th(u_cfg_if.data.battery_voltage_uv_th),
    .battery_charge_voltage(u_adc_if.data.battery_charge_voltage),
    .battery_charge_voltage_valid(u_adc_if.data.valid),
    .battery_charge_voltage_ov_th(u_cfg_if.data.battery_charge_voltage_ov_th),
    .battery_charge_voltage_uv_th(u_cfg_if.data.battery_charge_voltage_uv_th),
    .output_rect_voltage1(u_adc_if.data.output_rect_voltage1),
    .output_rect_voltage2(u_adc_if.data.output_rect_voltage2),
    .output_rect_voltage3(u_adc_if.data.output_rect_voltage3),
    .output_rect_voltage1_valid(u_adc_if.data.valid),
    .output_rect_voltage2_valid(u_adc_if.data.valid),
    .output_rect_voltage3_valid(u_adc_if.data.valid),
    .output_rect_voltage_ov_th(u_cfg_if.data.output_rect_voltage_ov_th),
    .output_rect_voltage_uv_th(u_cfg_if.data.output_rect_voltage_uv_th),
    .invertor_temp_voltage(u_adc_if.data.invertor_temp_voltage),
    .invertor_temp_voltage_valid(u_adc_if.data.valid),
    .invertor_temp_voltage_ov_th(u_cfg_if.data.invertor_temp_voltage_ov_th),
    .invertor_temp_voltage_uv_th(u_cfg_if.data.invertor_temp_voltage_uv_th),
    .battery_temp_voltage(u_adc_if.data.battery_temp_voltage),
```

```
        .battery_temp_voltage_valid(u_adc_if.data.valid),
        .battery_temp_voltage_ov_th(u_cfg_if.data.battery_temp_voltage_ov_th),
        .battery_temp_voltage_uv_th(u_cfg_if.data.battery_temp_voltage_uv_th),
        .service_voltage_err_flag(),
        .service_voltage_err_flag_clr(u_ctrl_if.data.service_voltage_err_flag_clr),
        .blanking_time(u_cfg_if.data.blanking_time)
    );

    wire_if #(.data_t(adc_data_t)) u_adc_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(cfg_data_t)) u_cfg_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    wire_if #(.data_t(ctrl_datas_t)) u_ctrl_if(
        .clk(clk),
        .reset_n(reset_n)
    );

    initial begin

uvm_config_db #(virtual wire_if #(adc_data_t))::set(uvm_root::get(), "uvm_test_top", "adc_vif",
u_adc_if);

uvm_config_db #(virtual wire_if #(cfg_data_t))::set(uvm_root::get(), "uvm_test_top", "cfg_vif",
u_cfg_if);

uvm_config_db #(virtual wire_if #(ctrl_datas_t))::set(uvm_root::get(), "uvm_test_top", "ctrl_vif",
u_ctrl_if);

uvm_config_db #(virtual clock_if)::set(uvm_root::get(), "uvm_test_top", "clk_vif", u_clk_if);
uvm_config_db #(virtual clock_en_if)::set(uvm_root::get(), "uvm_test_top", "clk_en_vif", u_clk_en_if);

    end

    initial begin
        $timeformat(-9, 1, "ns", 10);
        run_test();
    end

endmodule
```

```
//voltage monitor top tb
module top;

   import uvm_pkg::*;
   import voltage_monitor_pkg::*;
   import dip_voltage_monitor_tb_pkg::*;

   logic clk;
   logic reset_n;

   clock_if u_clk_if();

   assign clk = u_clk_if.clk;
   assign reset_n = u_clk_if.rstb;

   clock_en_if u_clk_en_if(.clk(clk));

   voltage_monitor #(.timeout_width(7)) volt_mon (.clk(clk),
                        .rstb(reset_n),
                        .clk_en_100us(u_clk_en_if.clk_en),
                        .enable(u_cfg_if.data.enable),
                        .err_clear(u_ctrl_if.data.error_clear),
                        .source_voltage(u_adc_if.data.adc_data),
                        .source_voltage_lower_th(u_cfg_if.data.uv_th),
                        .source_voltage_upper_th(u_cfg_if.data.ov_th),
                        .source_voltage_timeout(u_cfg_if.data.timeout),
                        .error()
                        );

   assign u_obs_if.data =  volt_mon.error;

   wire_if #(.data_t(ctrl_data_t)) u_ctrl_if(
      .clk(clk),
      .reset_n(reset_n)
   );

   wire_if #(.data_t(adc_data_t)) u_adc_if(
      .clk(clk),
      .reset_n(reset_n)
   );

   wire_if #(.data_t(config_data_t)) u_cfg_if(
      .clk(clk),
      .reset_n(reset_n)
   );
```

```
    wire_if #(.data_t(logic)) u_obs_if(
    .clk(clk),
    .reset_n(reset_n)
  );

  initial begin
    uvm_config_db #(virtual wire_if #(ctrl_data_t))::set(uvm_root::get(), "uvm_test_top", "ctrl_vif",
u_ctrl_if);

    uvm_config_db #(virtual wire_if #(adc_data_t))::set(uvm_root::get(), "uvm_test_top", "adc_vif",
u_adc_if);

    uvm_config_db #(virtual wire_if #(config_data_t))::set(uvm_root::get(), "uvm_test_top", "cfg_vif",
u_cfg_if);

    uvm_config_db #(virtual wire_if #(logic))::set(uvm_root::get(), "uvm_test_top", "obs_vif",
u_obs_if);

    uvm_config_db #(virtual clock_if)::set(uvm_root::get(), "uvm_test_top", "clk_vif", u_clk_if);

    uvm_config_db #(virtual clock_en_if)::set(uvm_root::get(), "uvm_test_top", "clk_en_vif",
u_clk_en_if);

  end

  initial begin
    $timeformat(-9, 1, "ns", 10);
    run_test();
  end

endmodule
```