Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño de un ambiente de verificación formal basado en SystemVerilog Assertions para una unidad aritmética lógica de punto flotante

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

Josué Quirós Barrantes

Cartago, 3 de diciembre de 2025

INSTITUTO TECNOLÓGICO DE COSTA RICA
ESCUELA DE INGENIERÍA ELECTRÓNICA
TRABAJO FINAL DE GRADUACIÓN
TRIBUNAL EVALUADOR
ACTA DE EVALUACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

Estudiante: **Josué Quirós Barrantes** Carné: 2020096808

Nombre del Trabajo Final de Graduación: *Diseño de un ambiente de verificación formal basado en SystemVerilog Assertions para una unidad aritmética lógica de punto flotante*

Los miembros de este Tribunal hacen constar que este Trabajo Final de Graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica

Nota de Trabajo Final de Graduación: 85

Miembros del Tribunal

Dr. Ing. Alfonso Chacón Rodríguez
Profesor lector

Dr. Ing. Ronny García Ramírez
Profesor lector

Dr. Ing. Roberto Molina Robles
Profesor asesor

Cartago, 3 de Diciembre de 2025

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

TRABAJO FINAL DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado *Diseño de un ambiente de verificación formal basado en SystemVerilog Assertions para una unidad aritmética lógica de punto flotante*, realizado por el señor Josué Quirós Barrantes y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Dr. Ing. Alfonso Chacón Rodríguez

Profesor lector

Dr. Ing. Ronny García Ramírez

Profesor lector

Dr. Ing. Roberto Molina Robles

Profesor asesor

Cartago, 3 de Diciembre de 2025

Resumen

Este proyecto desarrolla un entorno de verificación formal para la unidad aritmética lógica de punto flotante (FPU) del microcontrolador SIWA. Se requiere validar la ALU exhaustivamente antes de integrarla al estándar RISC-V con extensión F.

La verificación se realizó con SystemVerilog Assertions y VC Formal, siguiendo un enfoque grey-box para evaluar cada etapa interna del sumador y del multiplicador en precisión simple IEEE-754. Se definieron aserciones para comprobar el correcto desempaquetado, alineamiento de exponentes, suma y resta de mantisas, normalización, redondeo, empaquetado y manejo de excepciones.

El entorno permitió identificar fallos funcionales y validar comportamientos críticos como overflow, underflow y NaN. El resultado es un flujo estructurado y reproducible que fortalece la confiabilidad del diseño y sienta la base para futuras verificaciones de la FPU completa.

Palabras clave: Verificación formal, SystemVerilog Assertions, SVA, VC Formal, IEEE 754, Punto flotante, Formal Property Verification, FPV, RISC-V, RV32F, Aserciones, SIWA, Cobertura, Cono de Influencia, COI

Abstract

This project develops a formal verification environment for the floating-point arithmetic logic unit (FPU) of the SIWA microcontroller. The ALU must be thoroughly validated before being integrated into the RISC-V standard with the F extension.

The verification was performed using SystemVerilog Assertions and VC Formal, following a grey-box approach to evaluate each internal stage of the adder and multiplier in IEEE-754 single precision. Assertions were defined to check correct unpacking, exponent alignment, mantissa addition and subtraction, normalization, rounding, packing, and exception handling.

The environment enabled the identification of functional failures and the validation of critical behaviors such as overflow, underflow, and NaN. The result is a structured and reproducible flow that strengthens the design's reliability and establishes the foundation for future verification of the complete FPU.

Keywords: Formal verification, SystemVerilog Assertions, SVA, VC Formal, IEEE 754, Floating point, Formal Property Verification, FPV, RISC-V, RV32F, Assertions, SIWA, Coverage, Cone of Influence, COI

Declaratoria de Autenticidad

Yo, Josué Quirós Barrantes, estudiante de Ingeniería en Electrónica en el Instituto Tecnológico de Costa Rica, declaro que el presente proyecto es un trabajo original de mi autoría exclusiva. Todo el contenido técnico, teórico y metodológico aquí presentado ha sido producto de mi investigación personal o, en los casos donde se han utilizado fuentes externas, estas han sido debidamente citadas y referenciadas conforme a las normas académicas vigentes. Me comprometo formalmente a mantener los estándares de integridad académica, absteniéndome de cualquier forma de plagio o falsificación de información, en cumplimiento de los reglamentos institucionales

Agradecimientos

Agradezco a mi madre, Ana Grace, y a mi padre, José María, por todo el esfuerzo que han hecho por mí y mis hermanos, y que continúan haciendo.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Antecedentes

El Tecnológico de Costa Rica (TEC) es una institución nacional autónoma de educación superior. Dentro de las carreras que ofrece se encuentra la Licenciatura en Ingeniería Electrónica, ofrecida por la Escuela de Ingeniería Electrónica, fundada en 1976 [6], donde se tiene como propósito el rápido crecimiento de las tecnologías de la información y las comunicaciones, automatización, el mercado del entretenimiento y la tecnología médica [7].

En la escuela se ubica el Laboratorio de Diseño de Circuitos Integrados (**DCILab**), cuyo propósito es desarrollar tecnología electrónica avanzada en el campo de los circuitos integrados y microprocesadores para aplicaciones médicas y otros sectores. Según datos del TEC [8].Uno de sus avances más importantes es el desarrollo del microcontrolador SIWA (acrónimo de Sabiduría Ancestral en lengua cabécar), creado con el objetivo de ofrecer soluciones tecnológicas que respondan a necesidades locales mediante el fortalecimiento de capacidades en diseño de hardware [7]. SIWA ha sido diseñado como un sistema en chip (SoC) de bajo consumo energético, orientado a aplicaciones médicas implantables y portátiles. Está basado en un registro de cerrojo (*Latch-based register*) que es 30 % más pequeño y 25 % más eficiente energéticamente que una implementación flip-flop equivalente [3].

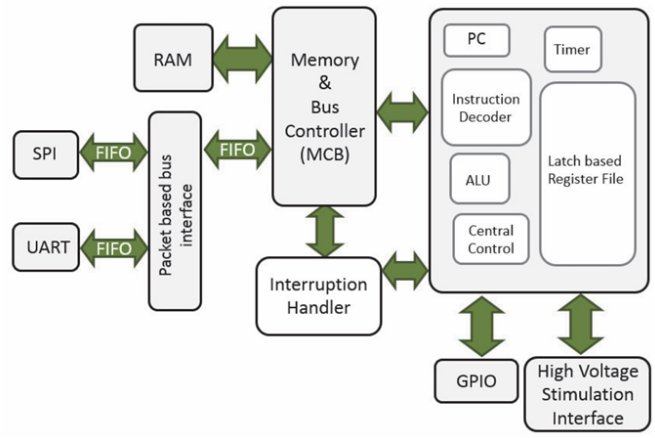

Figura 1.1. Diagrama de bloques SIWA. Tomado de [3]

1.2. Problema existente e importancia de la solución

Es importante mencionar que el SIWA es un sistema ya desarrollado, basado en una arquitectura RISC-V, pero actualmente se encuentra en una etapa de mejora. Una de estas actualizaciones es mediante la incorporación de una unidad de punto flotante, que se encuentra en fase de implementación. Esto tiene como objetivo mejorar su arquitectura, pensada en aplicaciones que requieran varios núcleos, acelerar los procesos y tener programas más cortos. Además, se pretende que esta unidad cumpla con el estándar RISC-V con extensión F, lo cual permitirá garantizar compatibilidad con operaciones en punto flotante dentro del ecosistema.

Una ALU de punto flotante (FPU, por sus siglas en inglés) es responsable de ejecutar operaciones como suma, resta, multiplicación, división, raíz cuadrada y comparaciones sobre números representados en formato de punto flotante. Este tipo de operaciones es fundamental en aplicaciones científicas, procesamiento de señales, gráficos por computadora y otros ámbitos donde se requiere un alto grado de precisión numérica.

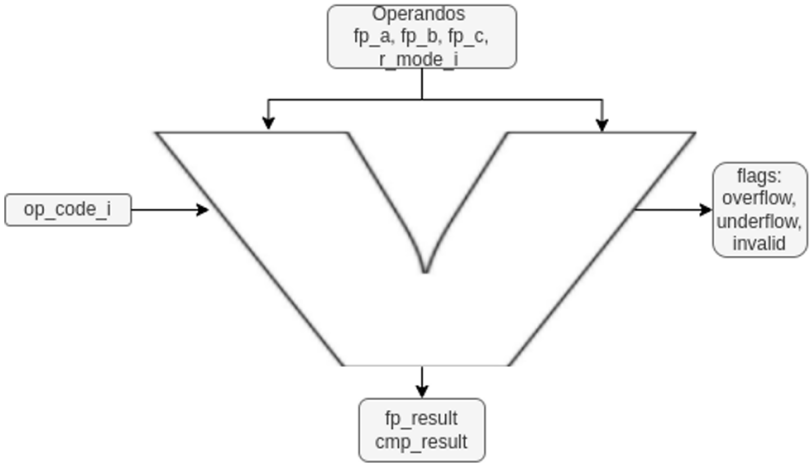

Figura 1.2. Diagrama simbólico de la ALU. Tomado de [4]

En la figura 1.2 se describen las entradas y salidas de la ALU.

- **fp_a, fp_b, fp_c**: Son los número en punto flotante a operar.
- **r_mode_i**: Criterio o modo de redondeo (ver tabla2.3).
- **op_code_i**: Operación matemática que debe realizar.
- **fp_result_i**: Resultado aritmético.
- **cmp_result**: Resultado de una comparación (mayor, menor, igual, etc).

Actualmente, la ALU desarrollada por el estudiante **Samuel Cabrera** [4], es capaz de realizar sumas, restas y comparaciones dentro del estándar IEEE754 de precisión simple; en esta se trabaja con 32 bits, utiliza 1 bit para signo, 8 bits para el exponente (ajusta la escala del número, permitiendo representar valores muy grandes o muy pequeños) y 23 bits de mantisa (los dígitos significativos con precisión); lo que significa que el valor máximo representable por la tecnología es $\pm 3,40 \times 10^{38}$ y el valor mínimo normal es $\pm 1,18 \times 10^{-38}$.

También presenta ciertas limitaciones en el diseño, no está implementada la división ni la raíz cuadrada. La unidad de multiplicación fue diseñada para tratar con valores subnormales (muy cercanos a cero), esta redondea esos valores a cero [2].

Sabiendo eso, el rediseño hacia una unidad de punto flotante, introduce una complejidad funcional considerable que debe ser abordada mediante un entorno de verificación riguroso. Esto es particularmente importante dado que el SIWA está destinado a aplicaciones médicas, donde los errores en los cálculos pueden comprometer la seguridad del usuario. Además, verificar exhaustivamente la FPU antes de su fabricación en hardware resulta esencial para garantizar su funcionamiento correcto y prevenir errores costosos en etapas posteriores de su ciclo de desarrollo. En este contexto, el diseño de un entorno de verificación específico y confiable se vuelve fundamental para asegurar la calidad y robustez del sistema.

Impulsados por la necesidad de ampliar las capacidades del microcontrolador SIWA, se ha decidido incorporar una unidad aritmética lógica de punto flotante. Esta adición busca fortalecer su arquitectura bajo el estándar RISC-V con extensión F, lo cual es clave para futuras aplicaciones médicas y científicas que requieren cálculos numéricos complejos y precisos. Sin embargo, esta ALU se encuentra aún en fase de desarrollo. La implementación actual ha sido construida de forma progresiva por diferentes estudiantes, lo que ha generado una estructura fragmentada, con funcionalidades incompletas y errores acumulados.

La dificultad radica en que ciertos errores pueden manifestarse solo bajo condiciones muy específicas, difíciles de anticipar. Por ello, es fundamental contar con un entorno estructurado que permita observar y comprobar rigurosamente el comportamiento de la unidad de punto flotante. Este entorno debe facilitar la descripción precisa de las condiciones que debe cumplir el sistema, permitiendo así detectar fallos sutiles y validar que el diseño responde correctamente ante una amplia variedad de situaciones. No es lo mismo encontrar un error en las etapas de producción que en etapas de diseño y pre-silicio.

De acuerdo con William K Lam, la verificación representa el mayor reto en el ciclo de diseño, al punto de consumir aproximadamente un 70 % del tiempo total de desarrollo de un circuito integrado moderno. Esto se debe a que, a diferencia del diseñador, quien trabaja con casos representativos, el verificador debe considerar todos los escenarios posibles, lo cual hace que el espacio de casos sea potencialmente infinito [9].

1.3. Solución seleccionada y requisitos de diseño

Para que el sistema SIWA evolucione hacia una arquitectura compatible con el estándar RISC-V con extensión F, es esencial una unidad de punto flotante (FPU), esta debe ser capaz de realizar correctamente operaciones en punto flotante e instrucciones fundamentales como suma, resta, multiplicación y división. Sin embargo, la conformidad con el estándar no se limita únicamente a ejecutar estas operaciones básicas. Es necesario que el diseño contemple adecuadamente una serie de condiciones especiales que pueden surgir en cálculos reales. Estas incluyen el manejo correcto de overflow, cuando el resultado excede la representación posible; underflow, cuando el resultado es demasiado pequeño para ser representado; y valores indefinidos o erróneos como NaN (Not a Number) [1]. Además, se deben considerar comportamientos como el redondeo según diferentes modos definidos por el estándar y la preservación de la precisión en cada etapa de cálculo.

El cumplimiento de estos requisitos no solo es indispensable para el funcionamiento interno del sistema, sino que también influye directamente en la elección del enfoque de verificación, pues algunos bloques de la unidad de punto flotante presentan un nivel de criticidad donde es necesario usar estrategias que permitan analizar su comportamiento de forma más completa y detallada. Además, la ALU no está completa según el estándar, falta implementar los módulos que permitan a la ALU dividir, realizar raíces cuadradas y el multiplicador asume que los valores subnormales son cero.

Es ahí donde entra la verificación formal. Esta parte de un enfoque matemático, que permite demostrar de forma exhaustiva que ciertas propiedades del diseño se cumplen en todos los casos

posibles, sin necesidad de aplicar miles de pruebas como una verificación basada en simulaciones. Sin embargo, la verificación formal también presenta limitaciones; su aplicabilidad está condicionada por la complejidad del circuito, ya que analizar diseños muy complejos requiere de una gran cantidad de recursos computacionales y tiempo. Es por ello que se vuelve un recurso viable cuando los módulos a validar están bien acotados, permitiendo detectar errores que podrían pasar desapercibidos con enfoques basados en testbenches.

El flujo general para implementar la Verificación Formal Propietaria (VFP), empieza por comprender los requisitos del diseño o cual implica analizar las especificaciones técnicas para identificar las propiedades críticas que deben ser validadas, dadas las limitaciones que presenta el método. Luego se configuraría el entorno donde el diseño a verificar está listo para ser analizado por la herramienta formal en combinación de aserciones. Una aserción es una descripción de una propiedad del diseño. Si una propiedad que se está verificando en una simulación no se comporta de la manera que esperamos, la aserción falla; si una propiedad que está prohibida en un diseño ocurre durante la simulación, la aserción falla [10].

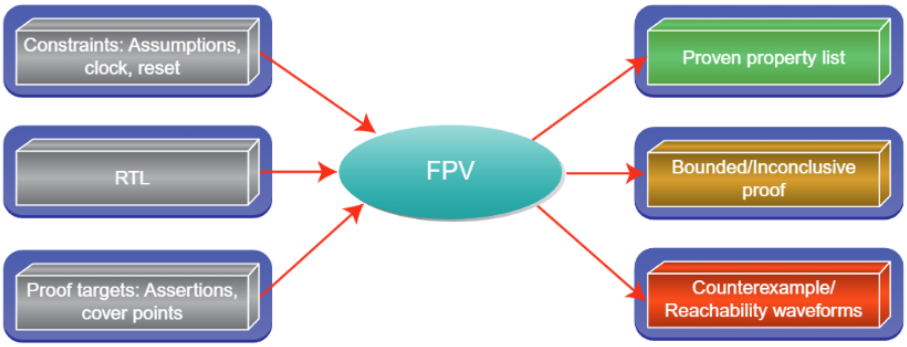

Figura 1.3. Ejecución de una herramienta de VFP. Tomado de [5]

Por estas razones, se ha optado por una estrategia centrada en la verificación formal, específicamente con el uso de aserciones. Se construirá un entorno de verificación donde se escribirán aserciones que permitan comprobar que la ALU de punto flotante se comporta según lo esperado y cumple con las especificaciones funcionales indicadas en su diseño. Esto resulta especialmente valioso para los bloques críticos de una unidad de punto flotante, donde se debe cumplir con el estándar RISC-V. Sin embargo, cabe destacar que las demás metodologías no son inadecuadas, sino que responden a otros contextos y necesidades.

1.4. Objetivos

1.4.1. Objetivo General

Diseñar un ambiente de verificación para una unidad aritmética lógica de punto flotante donde se evalúa si cumple con las especificaciones funcionales establecidas en la documentación original.

1.4.2. Objetivos Especificos

1. Identificar los bloques y componentes internos de la unidad aritmética lógica que son aptos para un verificación formal basada en SystemVerilog Assertions.

2. Diseñar e implementar un entorno de verificación formal que permita comprobar las propiedades funcionales de la unidad aritmética lógica de punto flotante.

3. Diseñar las aserciones del entorno de verificación formal que expresen el comportamiento funcional esperado de la unidad aritmética lógica de punto flotante.

Capítulo 2

Marco Teórico

El uso de la aritmética en punto flotante es fundamental en prácticamente todas las áreas donde se requiere procesar valores numéricos con un amplio rango dinámico y un control preciso del error. Desde simulaciones científicas y modelado físico, hasta gráficos computacionales, inteligencia artificial, procesamiento de señales y sistemas embebidos de alto rendimiento, el punto flotante se ha convertido en el formato estándar para representar y manipular números reales en hardware. Su principal ventaja es la capacidad de manejar números extremadamente grandes o pequeños manteniendo, al mismo tiempo, un alto grado de precisión.

La verificación de hardware para operaciones en punto flotante requiere un rigor especial. No solo es necesario validar la representación y el flujo de datos, sino también garantizar que cada operación cumpla exactamente con las especificaciones.

En esta sección se presenta el fundamento teórico que sustenta la verificación de operaciones en punto flotante bajo el estándar IEEE 754. Se describen los conceptos generales relacionados con la representación, normalización, redondeo y la verificación de hardware.

2.1. Representación en Punto Flotante

El estándar IEEE 754 define la forma en que los números en punto flotante deben representarse en hardware digital. En el caso de precisión simple (32 bits), un número se compone de tres campos [11]:

- **Signo (1 bit)**: indica si el número es positivo o negativo.
- **Exponente (8 bits)**: representado en formato con sesgo.
- **Mantisa (23 bits)**: corresponde a la fracción significativa del número, a la cual se le antepone un bit implícito en el caso de números normalizados.

La representación final de un número en precisión simple se define como:

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Sign	Exponente (8 bits)	Mantisa (23 bits)

Tabla 2.1. Formato IEEE 754 de precisión simple (32 bits).

Existen representaciones especiales: cero, infinito, números subnormales y NaN (Not a Number), las cuales se identifican por combinaciones particulares de exponente y mantisa.

Tabla 2.2. Representaciones IEEE 754 en precisión simple (32 bits)

Valor	**Signo (1 bit)**	**Exponente (8 bits)**	**Mantisa (23 bits)**
+0	0	00000000	00000000000000000000000
−0	1	00000000	00000000000000000000000
+∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	x	11111111	xxxxxxxxxxxxxxxxxxxxxxx
NaN estandar	x	11111111	10000000000000000000000

2.2. Números Normales y Subnormales

En punto flotante (32 bits), un número normal es aquel cuyo exponente está entre 1 y 254, lo que le permite representar magnitudes que van aproximadamente desde $3{,}4028235e^{38}$ hasta $1{,}1754944e^{-38}$. Cuando el exponente llega a 0, los valores pasan a ser subnormales: ya no tienen bit implícito y solo cubren magnitudes más pequeñas, desde alrededor de $1{,}1754942e^{-39}$ hasta $1e^{-45}$ valores aún más cercanos a cero, pero con menor precisión [12].

2.3. Modos de Redondeo

El estándar IEEE 754 contempla varios modos de redondeo:

Tabla 2.3. Codificación del modo de redondeo [1].

Modo de Redondeo	Código	Significado
000	RNE	Redondear al más cercano, con empate hacia el par
001	RTZ	Redondear hacia cero
010	RDN	Redondear hacia abajo ($-\infty$)
011	RUP	Redondear hacia arriba ($+\infty$)
100	RMM	Redondear al más cercano, con empate hacia la mayor magnitud
101	—	Reservado para uso futuro
110	—	Reservado para uso futuro
111	DYN	En el campo `rm` de la instrucción, selecciona modo dinámico; en el registro de modo de redondeo, reservado

Para llevar a cabo un redondeo es necesario acondicionar el número con el uso de bits adicionales *guard* (es el penúltimo bit del número original), *round* (es el último bit del número original) y *sticky* (es el resultado lógico de una compuerta OR del resto de bits que van a ser "recortados"). Esto permite implementar correctamente las reglas de redondeo y mantener la precisión establecida por el estándar.

2.4. Verificación de Hardware

La verificación de hardware es un proceso fundamental cuya finalidad es garantizar que una implementación cumple con las especificaciones funcionales establecidas. Este proceso es, en esencia, el inverso al diseño: mientras que el diseño transforma una especificación abstracta en una implementación concreta, la verificación confirma que dicha implementación mantiene la funcionalidad descrita en los niveles superiores de abstracción.

En este caso se va centrar en la verificación de implementación, también conocida como property checking o model checking, donde se comprueba si una implementación satisface un conjunto de propiedades que representan las especificaciones [9]. Este enfoque es unidireccional, mientras mayor sea la diferencia entre el enfoque de diseño y el enfoque de verificación, mayor confianza se obtiene en los resultados.

2.4.1. Verificación Basada en Simulación

La simulación consiste en aplicar estímulos de entrada a un diseño y comparar los resultados con un modelo de referencia. Un entorno de simulación típico incluye un testbench, generadores de estímulos, un modelo para la salida esperada y un mecanismo de comparación. Se utilizan tanto pruebas dirigidas para escenarios específicos como pruebas pseudoaleatorias, que permiten explorar regiones del espacio

de estados no consideradas explícitamente [9]. Entre las metodologías más utilizadas se encuentra UVM (*Universal Verification Methodology*)

2.4.2. Verificación Formal

La verificación formal no necesita vectores de entrada, porque trabaja directamente sobre modelos matemáticos del diseño. Esto le permite explorar el espacio de estados de manera completa para ver si las propiedades que se definieron realmente se cumplen o si fallan [9]. Dentro de este enfoque, las dos técnicas principales son:

- Equivalence checking, que normalmente se basa en SAT o en BDD.
- Property checking, donde el verificador formal intenta demostrar o refutar propiedades lógicas o temporales que describen cómo debería comportarse el diseño.

Una de las ventajas más fuertes de la verificación formal es que no depende del muestreo del espacio de entrada, como sí pasa en simulación. Por eso se considera "completa". Pero aun así tiene limitaciones importantes: consume mucha memoria y tiempo, y además requiere definir restricciones correctas para que el análisis no explore estados que no representan escenarios reales. Sin embargo, no significa que el diseño esté totalmente libre de errores. Todavía pueden aparecer problemas si las especificaciones están mal escritas, si la cobertura funcional no es suficiente, si el usuario comete errores o incluso si las herramientas fallan.

2.4.3. System Verilog Assertions

Las aserciones en SystemVerilog (SVA) son una herramienta fundamental para describir y verificar el comportamiento esperado de un diseño hardware. Una aserción es una descripción que reporta cómo una propiedad debe cumplirse siempre, o bajo ciertas condiciones específicas. Su propósito principal es detectar violaciones de comportamiento de forma automática y lo más cerca posible de la fuente del problema. Esto permite verificar no solo valores lógicos, sino también el orden en que deben ocurrir los eventos. Las aserciones no sustituyen las pruebas tradicionales, sino que las complementan al ofrecer un mecanismo formal y continuo de vigilancia sobre el diseño [10].

Estas propiedades están compuestas por tres componentes principales. El requisito, que puede ser una secuencia de eventos o simplemente el estado de una variable, representa el inicio de la condición a verificar. La segunda parte es la consecuencia (grant), que describe lo que debe ocurrir cuando se cumple el requisito; si esto no sucede, se considera una violación y la aserción falla.Por último, está la relación temporal entre el requisito y la consecuencia, que también funciona como una instrucción directa para la herramienta.

Un ejemplo básico consiste en verificar que, si una señal de petición req se activa, la señal de reconocimiento gnt debe llegar dentro de un número limitado de ciclos. Esta es una relación temporal típica:

$$\texttt{assert property } (req \mid\!-\!> \#\#[1:3]\ gnt).$$

Esta aserción establece que cada vez que req sea 1, el diseño está obligado a producir un gnt entre 1 y 3 ciclos más tarde. Si eso no ocurre, la herramienta reportará una violación inmediatamente, lo que permite identificar fallos en protocolos, tiempos o control de flujo.

Las aserciones también se pueden usar para validar lógica combinacional. En estos casos, lo que interesa no es el paso del tiempo, sino que las relaciones entre las señales sean correctas en el ciclo actual. Este tipo de verificación es útil cuando se quiere asegurar que una salida se derive correctamente de las entradas, que un decodificador nunca produzca estados inválidos, o que una condición no permitida nunca aparezca. Este tipo de propiedades se modelan como expresiones lógicas simples [10].

$$\texttt{assert property } (a \&\& b -\!> y).$$

2.4.4. Fundamentos matemáticos de la Verificación Formal de Propiedades (FPV)

La Verificación Formal de Propiedades (Formal Property Verification, FPV) utiliza métodos matemáticos para demostrar si un diseño de hardware satisface incondicionalmente un conjunto de propiedades escritas como *assertions*. A diferencia de la simulación, FPV explora de manera simbólica todas las ejecuciones posibles del diseño mediante algoritmos basados en SAT, BDD y expansión temporal (*bounded model checking*), tal como se explica en el libro [13].

El flujo matemático interno consta de una serie de transformaciones lógicas que permiten convertir el problema de verificar una propiedad en un problema de satisfacibilidad proposicional. A continuación se describe detalladamente este proceso usando una propiedad de ejemplo.

1. Traducción de la propiedad a lógica proposicional

Consideremos la propiedad escrita en SystemVerilog Assertions:

$$\texttt{assert property } (req \mid\!-\!> \#\#1\ gnt).$$

Esta propiedad especifica que si req es verdadero en el ciclo t, entonces gnt debe ser verdadero en el ciclo $t + 1$. Su traducción lógica es:

$$\forall t: \quad req_t \rightarrow gnt_{t+1}.$$

La implicación se convierte algebraicamente en una disyunción:

$$\neg req_t \vee gnt_{t+1}.$$

Esta transformación es necesaria porque los procedimientos formales requieren expresar el problema en Forma Normal Conjuntiva (CNF). La conversión de una propiedad a CNF es un paso fundamental en la verificación formal basada en SAT, ya que permite expresar cualquier fórmula booleana como un conjunto de condiciones más pequeñas unidas por AND, donde cada una de esas condiciones es un OR de variables o sus negaciones, simplificando así su procesamiento por los motores de decisión. Esta estructura no se emplea para demostrar directamente que una propiedad es verdadera, sino para permitir que los algoritmos formales busquen de manera eficiente una asignación que la vuelva falsa. Lo que refleja que el objetivo práctico no es probar la verdad absoluta, sino detectar contraejemplos [13].

2. Expansión temporal del diseño (Bounded Model Checking)

FPV desenrolla el diseño durante k ciclos creando copias del estado:

$$S_0,\ S_1,\ S_2,\ \ldots,\ S_k.$$

Cada copia corresponde al valor de todas las señales en un instante del tiempo. [**?**].Para cada ciclo se añade la ecuación de transición del diseño:

$$S_{t+1} = T(S_t, I_t),$$

donde T es la función de transición lógica del RTL y I_t representa las entradas en el ciclo t.

3. Conversión completa a Forma Normal Conjuntiva (CNF)

Una vez que cada parte del diseño y de la propiedad ha sido expresada en términos proposicionales, el verificador formal reúne todo en una sola fórmula. Esta unifica todos los componentes necesarios para que el motor SAT pueda trabajar [13].

Para ello, el sistema lógico completo se organiza en cuatro componentes principales:

$$\Phi = \Phi_{\text{trans}} \ \wedge\ \Phi_{\text{init}} \ \wedge\ \Phi_{\text{prop}} \ \wedge\ \neg\Phi_{\text{prop}},$$

donde:

- Φ_{trans} contiene las ecuaciones de transición del diseño en cada ciclo desenrollado, describiendo cómo cada estado sucesor depende del estado actual y de las entradas.
- Φ_{init} representa las condiciones iniciales necesarias para definir los estados alcanzables.

- Φ_{prop} codifica la propiedad especificada por el usuario.
- $\neg\Phi_{\text{prop}}$ expresa la violación que el solver debe intentar satisfacer; los motores SAT se enfocan en "encontrar una asignación que invalide la fórmula" en lugar de demostrar su verdad [13].

Cada una de estas partes suele contener expresiones complejas (implicaciones, ecuaciones secuenciales, combinaciones lógicas profundas). Por lo que para convertirlas a CNF sin que el tamaño de la fórmula crezca de forma exponencial, los verificadores utilizan la transformación de Tseitin, la cual introduce variables auxiliares para cada subfórmula y produce un conjunto de cláusulas lógicamente equivalentes [13].

El resultado final es una fórmula de la forma:

$$\Phi \equiv \bigwedge_{i=1}^{m} C_i,$$

4. Algoritmo DPLL SAT: propagación booleana y aprendizaje

Una vez que todo el sistema ha sido convertido a CNF, la verificación formal se reduce al problema de determinar si existe una asignación de valores que satisfaga la negación de la propiedad. Para esto, FPV utiliza algoritmos SAT modernos, basados en el procedimiento clásico DPLL y mejorados mediante propagación booleana y aprendizaje de conflictos. En esta etapa, el verificador ya no trabaja con el diseño en sí, sino únicamente con las cláusulas resultantes de la transformación a CNF.

El procedimiento clásico de Davis–Putnam–Logemann–Loveland (DPLL) puede optimizarse si se aprovecha el hecho de que algunas variables no requieren dividir el problema en dos ramas. En particular, cuando una variable aparece como único literal en una cláusula —ya sea de forma directa o porque todas las demás variables de esa cláusula han sido fijadas a 0— su valor queda determinado de manera inmediata. Toda cláusula debe evaluarse a 1, por lo que si una variable aparece sola de forma positiva, debe asignarse a 1; si aparece negada, debe asignarse a 0. Esta deducción evita realizar divisiones innecesarias en el árbol de búsqueda [13].

Por ejemplo, en el subproblema:

$$(b \vee \neg d) \ \wedge \ (\neg b \vee c) \ \wedge \ c,$$

la variable c aparece sola como una cláusula unitaria. Por lo tanto, si existe una solución, c debe valer 1. Sustituyendo ese valor en las demás cláusulas, el problema se reduce a:

$$(b \vee \neg d) \ \wedge \ (\neg b).$$

Este tipo de simplificaciones incrementan sustancialmente la eficiencia del algoritmo, ya que eliminan la necesidad de explorar ramas completas del espacio de decisiones y permiten que el solver avance

mediante deducciones directas antes de recurrir a divisiones. Este tipo de simplificación recibe el nombre de *Boolean Constraint Propagation* (BCP), este mecanismo se considera una de las mejoras esenciales sobre el algoritmo original de Davis–Putnam, y por ello la versión extendida con propagación y decisiones sistemáticas suele denominarse *DPLL* (Davis–Putnam–Logemann–Loveland).

Si durante la búsqueda se alcanza un conflicto, es decir, una cláusula queda falsificada, el solver no simplemente retrocede. En su lugar, realiza un análisis del conflicto para descubrir qué combinación de decisiones condujo al error. A partir de ello, genera una nueva cláusula:

$$C_{\text{aprendida}} = \text{análisis\_de\_conflicto}(C_1, C_2, \ldots).$$

Esta cláusula aprendida se añade a la CNF, impidiendo que el solver repita la misma secuencia de decisiones erróneas [13].

2.4.5. EDA Tools

Electronic Design Automation (EDA) es el conjunto de herramientas de software destinadas a muchas apliacciones: automatizar el diseño, implementación, verificación de circuitos integrados, “bill of materials”, etc. Estas herramientas permiten manejar la complejidad creciente del hardware moderno mediante flujos especializados para síntesis lógica, simulación, verificación, análisis estático y diseño físico. Su uso es indispensable en cualquier proyecto de verificación digital, ya que proporcionan mecanismos matemáticos, visuales y estructurales para analizar exhaustivamente el comportamiento del RTL [14].

Las herramientas EDA que se encuentran disponibles, nos interesan aquellas relacionadas a la verificación formal. Estas permiten demostrar, mediante técnicas basadas en SAT, SMT y BDD, que un diseño cumple con un conjunto de propiedades sin necesidad de vectores de simulación. Entre las herramientas más utilizadas en la industria se encuentran Synopsys VC Formal, Cadence JasperGold y Siemens Questa PropCheck. Cada una implementa flujos de análisis equivalenciales y verificación basada en propiedades,integrando motores de decisión optimizados para explorar el espacio de estados de forma exhaustiva.

En este proyecto, se decidió utilizar VC Formal, es una plataforma industrial de verificación formal que ofrece análisis de propiedades, comprobación de equivalencia y cobertura formal, además, aprovechando que el TEC lo provee y paga las licencias. Su motor incorpora técnicas avanzadas como Bounded Model Checking (BMC), abstracción automática y aprendizaje de conflictos (*conflict-driven clause learning*). De acuerdo con Synopsys [15], la herramienta permite verificar módulos complejos mediante un flujo modular y optimizaciones de cone-of-influence, lo que la hace adecuada para diseños como unidades aritméticas en punto flotante.

2.4.6. Cobertura

La herramienta VC Formal proporciona tres métricas principales de cobertura que permiten evaluar qué tan completa y profunda ha sido la verificación del diseño. Estas métricas no sólo cuantifican el progreso, sino que además revelan qué partes del RTL han sido efectivamente analizadas por el motor formal y cuáles podrían requerir refinamiento en las propiedades.

- **Cover properties**: indican cuántas de las aserciones y propiedades que definimos como eventos de interés (por ejemplo, secuencias específicas o estados alcanzables) fueron realmente satisfechas por el motor formal. Se demuestra que el escenario descrito es alcanzable bajo las restricciones declaradas o no.
- **Alcance de la herramienta (reachability)**: esta cobertura refleja la porción del diseño que VC Formal logró explorar durante el análisis. Cuando el porcentaje es alto, significa que la herramienta pudo recorrer la mayor parte del RTL, lo cual es un buen indicador de que no existen regiones muertas o inaccesibles del código bajo las condiciones dadas.
- **COI (Cone of Influence)**: el COI representa el conjunto mínimo de señales del diseño que influyen directa o indirectamente en cada propiedad. Un COI pequeño no es negativo por sí mismo, pero sí revela que sólo una fracción de las señales contribuye realmente a la evaluación de la aserción, es decir, únicamente una parte relativamente reducida del código resultó necesaria para demostrar o refutar la propiedad. Esta métrica permite identificar lógica que no está siendo ejercida por las propiedades, condiciones excesivamente locales y, además, resaltar aquellas regiones del diseño que sí son críticas para la verificación [13].

Capítulo 3

Procedimiento metodológico

3.1. Descripción del bloque sumador suma

El módulo `fp_adder` implementa operaciones de suma y resta en punto flotante IEEE-754 de 32 bits. El objetivo es garantizar que su comportamiento sea correcto según la documentación original por el estudiante Samuel Cabrera, el procedimiento general para realizar una suma o resta es el siguiente [4]:

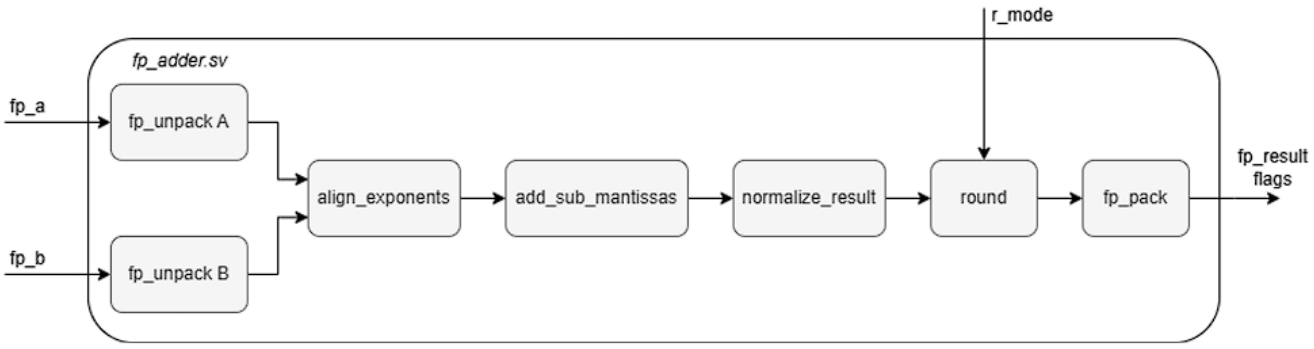

Figura 3.1. Diagrama de bloques del sumador

1. **fp_unpack**: Separación del signo, exponente y mantisa de cada operando.

   Reglas que debe cumplir este bloque:

   - El signo debe ser el MSB del número original (fp_x[31]), el exponente son los siguientes 8 bits (fp_x[30:23]) y la mantisa los últimos 23 bits (fp_x[22:0]).
   - Además el estudiante Cabrera [4] implementa en este bloque una forma para determinar si el número es “especial” (fp_x[30:23] != 8’hFF) o cero.

2. **align_exponents**: El número con menor exponente desplaza su mantisa hacia la derecha hasta igualar los exponentes del mayor.

Reglas que debe cumplir este bloque:

- Se desplaza la mantisa del número menor la misma cantidad que la diferencia entre los exponentes.
- El exponente resultante debe ser igual al exponente del número mayor.
- Si solo uno de los dos números es subnormal, el desplazamiento es igual al exponente mayor menos 1.
- Si ambos son subnormales, no hay desplazamiento.

3. **add_sub_mantissas**: Se suman si los signos son iguales, o se restan si los signos son diferentes.

   Reglas que debe cumplir este bloque:

   - Si los signos son iguales, las mantisas se suman.
   - Si los signos son diferentes, se resta la mantisa del número menor a la mantisa del mayor y se conserva el signo del mayor.

4. **normalize_result**: El resultado se ajusta para mantener la forma estándar de IEEE.

   Reglas que debe cumplir este bloque:

   - Si al menos uno de los operandos es normal, el MSB debe ser un 1, de esto surge dos casos:
     - Ceros a la izquierda: se debe hacer un corrimiento de la mantisa hasta que el MSB sea 1. el número total de corrimientos se debe restar del exponente común hasta que sea cero. en ese preciso momento, se detienen los corrimientos y el número se convierte en subnormal.
     - Uno a la derecha: podría que una suma de 24 bits(23 de mantisa y un implícito) tenga como resultado 2 (`10`) de implícito. en este caso, se desplaza a la izquierda la mantisa y se suma uno a el exponente, como si hubiera un carry.
   - Si ambos operandos son subnormales: la mantisa no se mueve, pero si el resultado genera un bit `1` en el implícito, el exponente aumenta en 1 y deja de ser un número subnormal.
   - Calcula y agrega a la mantisa resultante los bits round, guard y sticky de manera correcta.

5. **round**: Se aplica el modo de redondeo definido por IEEE 754.

   Reglas que debe cumplir este bloque:

   - Redondea correctamente la mantisa según el código de entrada.
   - La salidas del bloque son una mantisa de 23 bits y un bit de carry.
   - **Nota**: Cabrera suma el carry del round en unas lineas del código en el archivo top (fp_adder.sv).

6. **fp_pack**: El resultado se vuelve a codificar en los 32 bits de representación.

- El valor resultante debe ser la combinación de el signo del mayor, el exponente común y la mantisa redondeada sin el bit implícito.
- **Nota**: Cabrera le da formato a los números especial es lineas del cófdigo dentro del archivo top, los NaN tengan la forma `0X7fc00000`, infinito positivo `0X7f800000` e infinito negativo `0Xff800000`.

3.2. Descripción del bloque multiplicador

El módulo `fp_mul` implementa operación de multiplicación en punto flotante IEEE-754 de 32 bits. Su diseñador explica que los valores subnormales van a ser redondeados a un valor nulo. El objetivo es garantizar que su comportamiento sea correcto bajo todas las combinaciones de entrada y especificaiones de su diseñador.

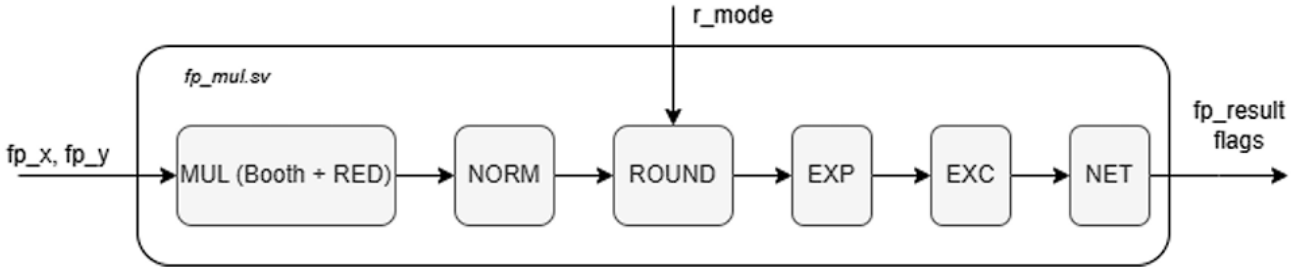

Figura 3.2. Diagrama de bloques del multiplicador

Según la documentación original por el estudiante Vianney Bernard, el procedimiento general para realizar una multiplicaión es el siguiente [2]:

1. **MUL (booth + RED)**: Implementa la operación de multiplicación optimizada mediante dos técnicas principales: el algoritmo de Booth y una etapa de reducción parcial (Radix-4). Su propósito es obtener el producto de dos operandos de manera eficiente, reduciendo el número de operaciones necesarias y mejorando el rendimiento del circuito.

   Reglas que debe cumplir este bloque:

   - Si al menos uno de los operandos es cero o subnormal, el resultado debe ser cero.
   - La mantisa resultante de ser de 48 bits.
   - La mantisa resultante debe coincidir concidir con la operación {1, mantisa_x}*{1, mantisa_y}

2. **NORM**: A diferencia del bloque normalizador del sumador, la mantisa que proviene del MUL nunca va contener ceros a la izquierda, no se considera los numeros subnormales. Entonces solo es

nesecario ajustar para mantener la forma estándar de IEEE en caso de un desbordamiento que se manifiesta o reporta como un carry.

Reglas que debe cumplir este bloque:

- Si el MSB de la mantisa es 1, debe desplazar la mantisa una vez a la derecha y encender el bit carry_n.
- Calcula y agrega a la mantisa resultante los bits round, guard y sticky de manera correcta.

3. **ROUND**: Se aplica el modo de redondeo definido por IEEE 754. Reglas que debe cumplir este bloque:
   - Redondea correctamente la mantisa según el código de entrada.
   - La salidas del bloque son una mantisa de 23 bits y un bit de carry.
4. **EXP**: Calcula correctamente el exponente tomando en cuenta los bits de carrys tanto del redondeo y el normalizador. Reglas que debe cumplir este bloque:
   - Detecta si ocurre un overflow (ovrf) o un underflow (udrf)
   - Ajusta el bias y calcula el exponente resultante.
5. **EXC**: Determina si se produjo un "not a number"(NaN), infinito o cero. Reglas de este bloque:
   - Si se multiplicó: cero por infinito, enciende la flag de NaN
   - Si sucedió un underflow, es un cero.
   - Si sucedió un overflow, es un infinito.
6. **NET**: Empaqueta devuelta el elresultado a un numero de 32 bits y le da la forma estándar de IEEE (casos de NaN, inf y zero)

3.3. Metodología de Verificación

La verificación del sumador se llevó a cabo utilizando un enfoque basado en SystemVerilog Assertions (SVA), complementado con las capacidades formales de *Synopsys VC Formal*. Este proceso requiere tanto la configuración adecuada del entorno de verificación como la integración de los módulos de aserciones sin modificar el código RTL original.

3.3.1. Configuración del Entorno en VC Formal

Para ejecutar las aserciones de manera formal, es necesario preparar un entorno de verificación mediante un conjunto de scripts `.tcl` específicos de *VC Formal*. Estos scripts permiten definir las rutas

de los archivos, establecer la topología del diseño, cargar las librerías correspondientes y compilar los módulos de verificación de forma estructurada.

De acuerdo con la documentación oficial de Synopsys [15], VC Formal proporciona mecanismos para analizar de manera exhaustiva el espacio de estados del diseño, lo que garantiza que las propiedades definidas con SVA se evalúen de forma completa. La herramienta soporta flujos basados en propiedades, cobertura formal y verificación modular, lo cual se aprovechó en este proyecto para validar cada bloque del sumador.

Entre las configuraciones realizadas en los scripts `.tcl` se incluyen los ajustes necesarios para habilitar el modo formal, activar la cobertura y preparar el diseño y las aserciones para su análisis. En particular, se habilita el motor formal mediante la variable `fml_mode_on`, junto con los parámetros de cobertura (`fml_cov_tgl_input_port` y `fml_enable_prop_density_cov_map`). Posteriormente, se define el módulo `top` del diseño y se realiza la lectura del RTL junto con los módulos de aserciones y el archivo `bind`. Finalmente, se ejecuta la corrida formal mediante `sim_run`, se almacena el estado estable del diseño y se genera un reporte detallado de las propiedades evaluadas.

Con el fin de mantener la integridad del diseño original, todas las aserciones fueron integradas utilizando un `bind`. Esta técnica permite asociar un módulo de verificación externo a un bloque RTL sin modificar sus archivos originales. De esta forma, el diseño permanece limpio y fiel a su implementación. En este trabajo, tanto el sumador como el multiplicador cuentan con un módulo de aserciones propio. Todos ellos fueron vinculados al diseño mediante `bind`. EL diagrama del entrono final es el siguiente:

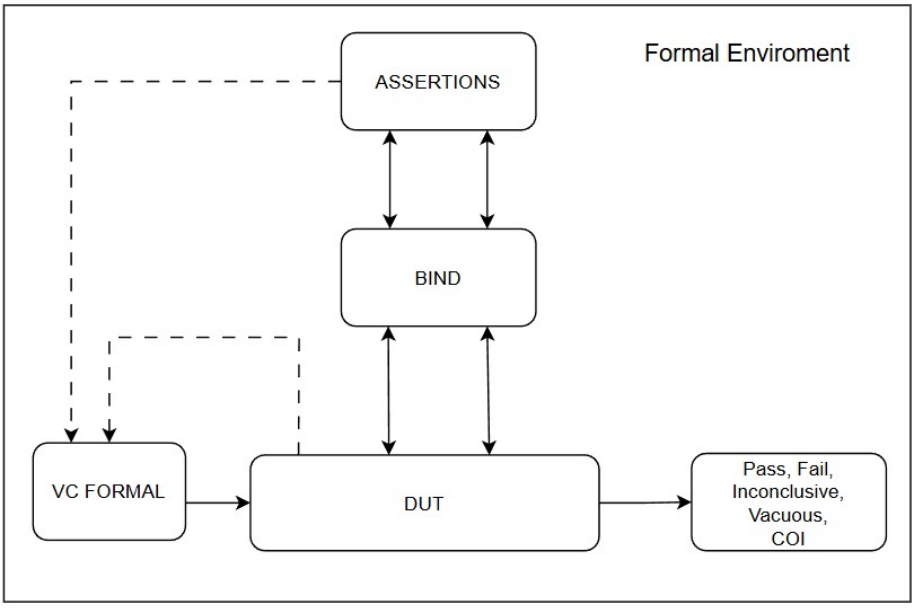

Figura 3.3. Diagrama del entrono diseñado

Aquí se muestra el flujo para probar la ALU dentro del entorno formal. Las líneas punteadas hacen referencia a la necesidad de que VC Formal primero obtenga y genere las condiciones y señales de entrada a partir del mismo diseño y de las aserciones escritas. Una vez que dichas condiciones están definidas, se

procede al flujo principal de verificación, donde VC Formal se encarga de estresar el DUT y compararlo contra la descripción de propiedades.

Al final, la herramienta entrega un desglose de los escenarios que pasaron, fallaron, no se ejecutaron, los escenarios incompletos y la cobertura alcanzada.

3.4. Diseño de aserciones

3.4.1. Integración de Aserciones

Se decide seguir una metodología estilo principalmente grey box. Pues se toma en cuenta cómo internamente la alu calcula y realiza sus operaciones. Esto principalmente, por dos razones principales: ambos diseños establecen como obtienen sus resultados, no están diseñados simplemente para seguir el estándar, pues su diseño va destinado a un aplicación específico y además, por que si evaluamos que cada etapa por separado, podemos analizar y obtener resultados más específicos, dándonos conocimiento en que bloque especifico del diseño se produjo los bugs, a diferencia de un enfoque totalmente black box, donde solo podríamos concluir que el diseño esta bien o mal.

3.4.2. Aserciones Implementadas para el sumador

Se diseñan aserciones tipo black box con el fin de verificar las operaciones aritméticas completas (suma y resta).Entre ellas, se escriben dos funciones que calculan el resultado de la suma y resta según el estándar y se compara con el valor obtenido. También, se desea verificar la conmutatividad de la operación, para ello, se crean dos instancias del dut, con los mismos valores de entrada, solo en orden invertido, fp_adder(a,b) == fp_adder(b,a).

Por último, para analizar y comprobar que las reglas definidas para cada bloque se cumplan, se establece un conjunto de aserciones basadas en SystemVerilog Assertions (SVA) que describen el comportamiento esperado en los distintos escenarios de operación.

3.4.2.1. Bloque fp_unpack

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

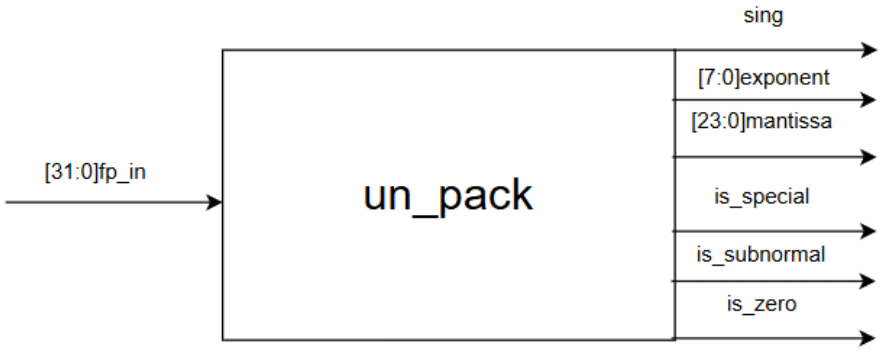

Figura 3.4. Diagrama de señales del un_pack

El diseño de las aserciones va destinado a que se haga un correcto desempaquetado. Cada salida sea correcta, según el número de entrada, a su vez identificando NaN o infinitos.

3.4.2.2. Bloque align_exponents

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

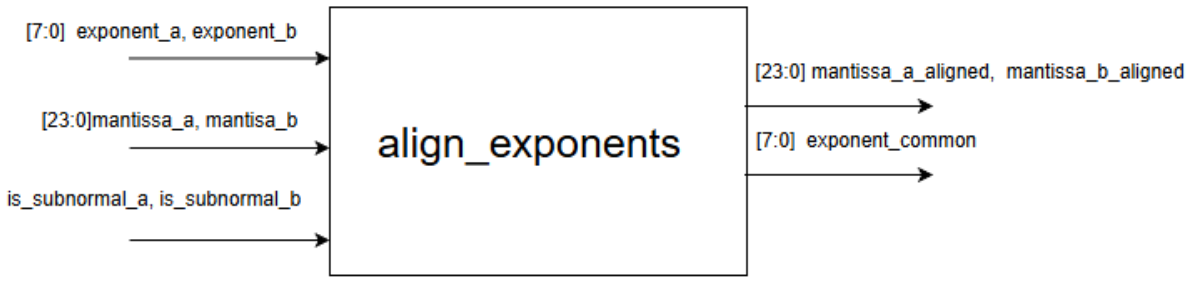

Figura 3.5. Diagrama de señales del align_exponents

Para verificar su correcto funcionamiento se define diferentes aserciones, para los diferentes escenarios dependiendo de sus entradas.

- Si ambos número son normales, el exponente común es igual al del mayor y la mantisa resultante es la mantisa del menor desplaza la diferencia de los exponentes.
- Si solo uno es subnormal, el exponente común es igual al del mayor y la mantisa resultante es la mantisa del menor desplaza la diferencia de los exponentes más 1 (osea se desplaza: exponente mayor - 1).
- Si ambos son subnormales, exponente común cero y no hay desplazamientos.

3.4.2.3. Bloque add_sub_mantissas

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

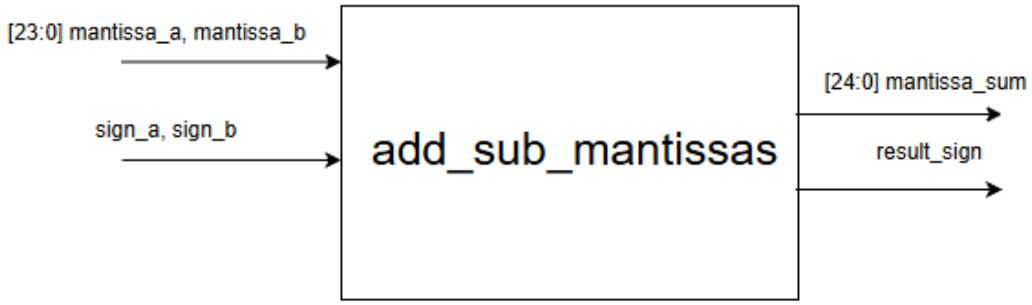

Figura 3.6. Diagrama de señales del add_sub_mantissas

A pesar de ser llamado sumador de mantisas, el bloque debe ser capaz de realizar sumas y restas, pues sumar números con signo diferente resultaría en una resta. Por ello, las aserciones de este bloque van enfocadas en: si el signo es igual, se suman las mantisas y se conserva el signo; y si son de distinto signo, se restan las mantisas y se conserva el signo del mayor.

3.4.2.4. Bloque normalize_result

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

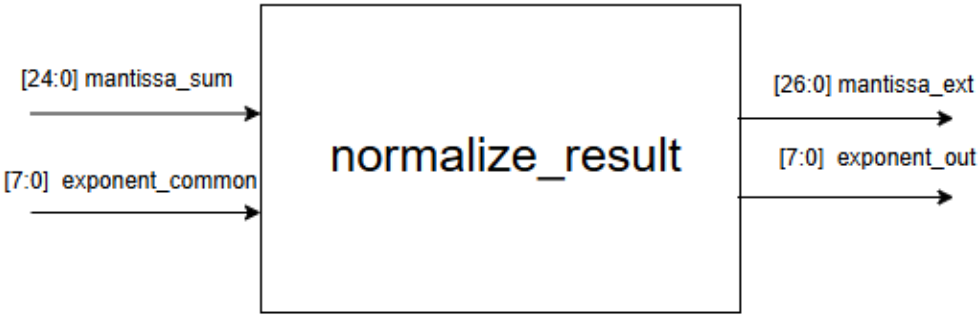

Figura 3.7. Diagrama de señales del normalize_result

Este bloque posee una complejidad mayor comparado al resto de bloques del sumador, pues, hay una gran variedad de condiciones que deben ser evaluadas. Las aserciones diseñadas cubren los siguientes casos:

- En una suma de números normales, si el resultado de la suma de mantisas generó un carry, se debe desplazar la mantisa a la derecha y aumentar el exponente una vez.

- En una suma de números subnormales, si el resultado de la suma de mantisas generó un carry, la mantisa se mantiene y se aumenta el exponente una vez.
- En una suma de números subnormales, si el resultado de la suma de mantisas generó un carry, la mantisa se mantiene y se aumenta el exponente una vez.
- En una resta de números normales, si el resultado de la suma de mantisas el MSB no es 1, se debe desplazar la mantisa a la izquierda y disminuir el exponente la misma cantidad que desplazamientos necesarios para que obtener MSB = 1 en la mantisa. Si en esta situación, se disminuyó tanto el exponente que se volvió cero, los desplazamientos se detiene y el número se vuelve subnormal.
- En una resta de números subnormales, el resultado no se modifica.

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

3.4.2.5. Bloque round

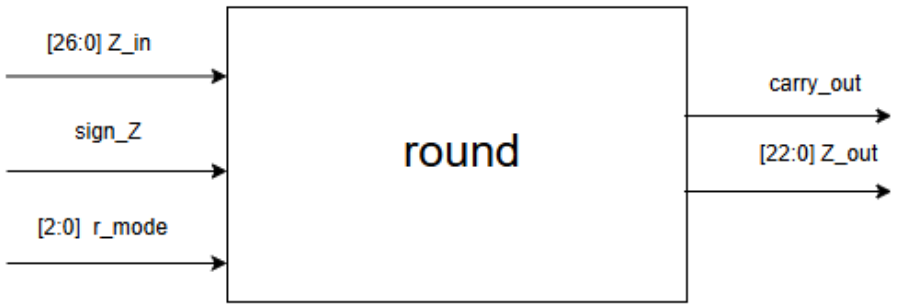

Figura 3.8. Diagrama de señales del round

Para verificar el bloque round se decide calcular el valor teórico verdadero según las señales de entrada y compararlas con la salida real de la etapa.

- Para el modo de redondeo RNZ, se decide hacer el cálculo de la mantisa teórica, dependiendo de los bits Guard, round y sticky.
- El redondeo hacia cero es simplemente la mantisa obtenida recortada
- Los modo de redondeo RDN y RUP, es necesario hacer el cálculo de la mantisa dependiendo del bit signo.
- Para el modo de redondeo RMM, es necesario hacer el cálculo de la mantisa dependiendo del bit round.
- Por último verificar cuando el carry se activa si corresponde a un desbordamiento por redondeo.

En resumen, debe cumplir con la siguiente tabla.

Tabla 3.1. Tabla de lógica de selección de redondeo [2]

r_mode	Modo	Signo	Round	Guard∨Sticky	Valor
000	Redondeo al más cercano, empates al par	–	0	–	Z
000	Redondeo al más cercano, empates al par	–	1	1	Z_{plus}
000	Redondeo al más cercano, empates al par	–	1	0	El que sea par
001	Redondeo hacia cero	–	–	–	Z
010	Redondeo hacia $-\infty$	0	–	–	Z
010	Redondeo hacia $-\infty$	1	–	–	Z_{plus}
011	Redondeo hacia $+\infty$	0	–	–	Z_{plus}
011	Redondeo hacia $+\infty$	1	–	–	Z
100	Redondeo al más cercano, empates lejos de cero	–	0	–	Z
100	Redondeo al más cercano, empates lejos de cero	–	1	–	Z_{plus}

3.4.2.6. Bloque fp_pack

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

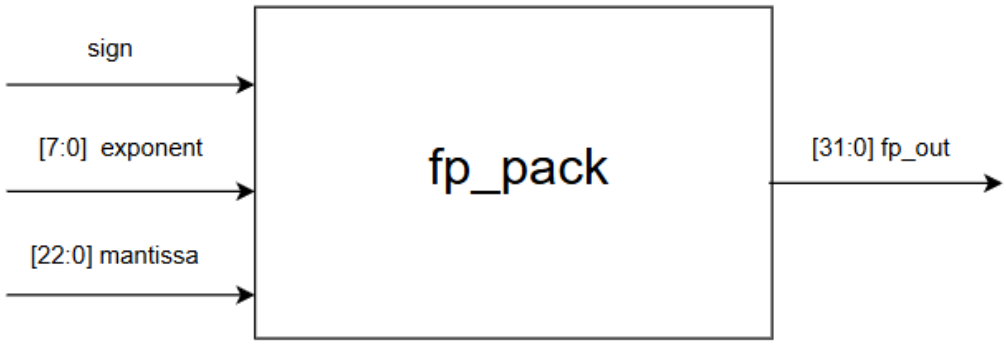

Figura 3.9. Diagrama de señales del fp_pack

Esta etapa solo recopila señales de las etapas anteriores y entrega el resultado final.

Sin embargo, hay ciertas señales y operaciones críticas que no son tomadas en cuenta por ninguna entrada o salida de ningún bloque, las cuales son: activación de underflow, overflow, NaN e infinitos y así como dar la correcta forma estándar en cada caso.

- Si exponent_common + carry por round + carry por norm >= 255, hay overflow.
- Si el valor de una resta es menor a 0X00000001, sin ser 0 , hay underflow.
- Si algún valor inicial era NaN, el resultado es NaN.

- Si se multiplica infinito positivo por infinito positivo, el resultado es NaN.
- El resultado de infinito positivo ± un número diferente de infinito negativo o NaN, es infinito positivo.
- El resultado de infinito negativo ± un número diferente de infinito positivo o NaN, es infinito negativo.

3.4.3. Aserciones Implementadas para el multiplicador

Para el multiplicador se desarrollaron aserciones de verificación en modalidad black-box, orientadas a comprobar el cumplimiento de propiedades aritméticas esenciales. Entre ellas se incluyen: la multiplicación por cero ($0 \times x = 0$), la identidad multiplicativa ($1 \times x = x$) y la propiedad de conmutatividad.

3.4.3.1. Bloque MUL (booth + RED)

La primera etapa de la multiplicación es el calculo de la multiplicación de mantisas, las señales de este bloque son las siguientes:

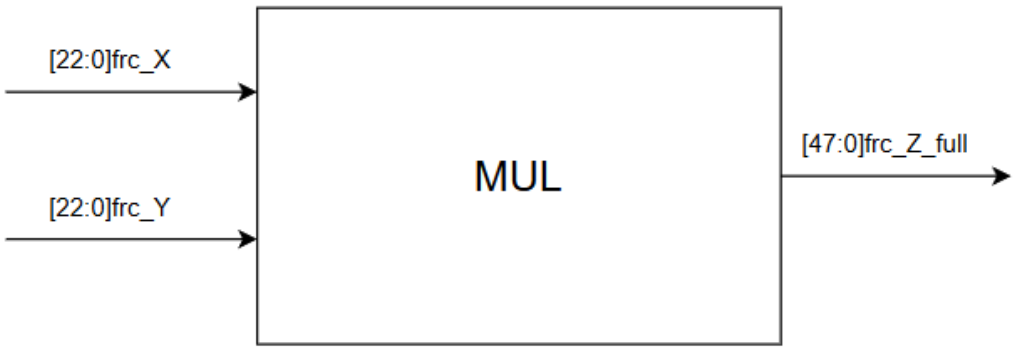

Figura 3.10. Diagrama de señales del MUL

Este bloque es, computacionalmente, más complejo, pues debe realizar cálculos con resultados de 48 bits, esto representa reto ante el motor de verificación utilizado (VC formal, FPV), pues tiene límite de cálculo de 12 horas.

Este bloque solo debe cumplir con la operación `frc_Z_full = {1'b1, frc_X} * {1'b1, frc_Y}`, sin embargo, no se logra hacer el cálculo de dicha asserción a tiempo. Por un tema de recursos, se toma la decisión de incorporar unos casos de verificación por simulación formal, donde se opta por utilizar un valor representativo del rango. Haciendo énfasis en que no es una prueba formal completa.

Un valor representativo del rango (Representative value) es aquel valor que está destinado a reflejar cómo se comporta la población a la cual pertenece, por lo que debería tener características similares a las de dicha población [16]. Entonces, se definen valores representativos para el rango de valores de números normales, subnormales y sus máximos y mínimos como casos de esquina.

3.4.3.2. NORM

En este caso, la normalización propuesta por Vianney se describe mediante un diagrama de bloques. Por lo que, mientras el diseño cumpla con el manejo de señales especificado en la siguiente figura y la normalización es correcta, se considera que opera exitosamente.

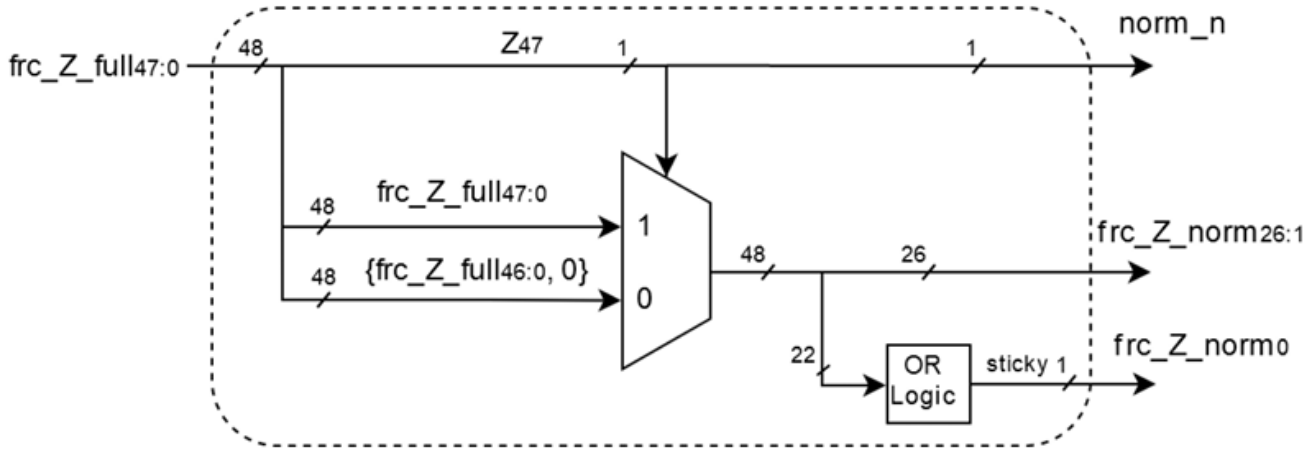

Figura 3.11. Modulo normalizador [2]

Sin embargo, también podemos verificar si el diseño cumple con la norma, estableciendo aserciones acerca del contenido y la forma del número resultante, por ejemplo: el bit implícito y la multiplicación por cero.

3.4.3.3. ROUND

Similar al round del sumador, este proceso es estándar, para ambos casos, la forma de la señal de entrada es un número de 26 bits, con la mantisa normalizada y el modo de redondeo. Por lo que solo se debe hacer el ajuste de la conexión con los nombres de las señales del multiplicador. Debe cumplir con la tabla3.1.

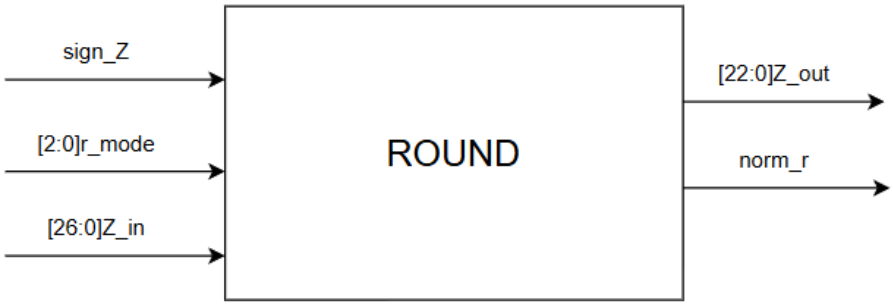

Figura 3.12. Diagrama de señales del ROUND

3.4.3.4. EXP

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

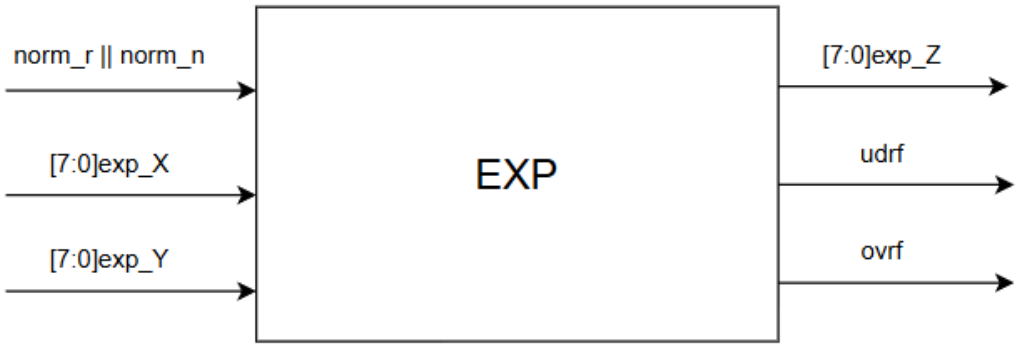

Figura 3.13. Diagrama de señales del EXP

Para concluir que el calculo del exponente es exitoso se decide verificar que el bias se ajusta correctamente si hubo algún tipo de carry, el exponente resultante es igual a la suma delo exponentes X, Y menos el bias, además que las banderas de underflow y overflow se enciendan de acuerdo al estándar, cuando no hay más bits para representar dichos escenarios respectivamente.

3.4.3.5. EXC

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

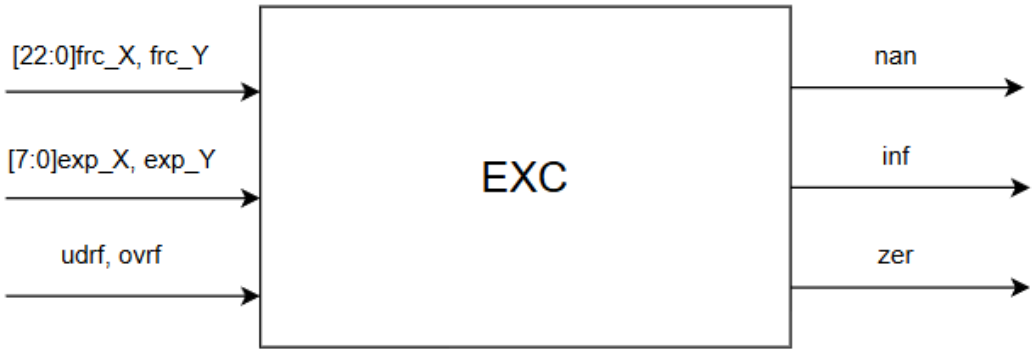

Figura 3.14. Diagrama de señales del EXC

Este bloque es relativamente sencillo, se encarga de solo identificar cuando una excepción se cumple, por lo que las aserciones diseñadas va enfocadas a los casos específicos:

- **Cero**: Si algún número era subnormal, cero o ocurrió un underflow.
- **Infinitos**: Si algún número era infinito o ocurrió un overflow.
- **Not a Number**: Si algún número era NaN o se esta intentando multiplicar un infinito por cero.

3.4.3.6. NET

Las señales de entrada y salida del bloque se especifican en el siguiente diagrama.

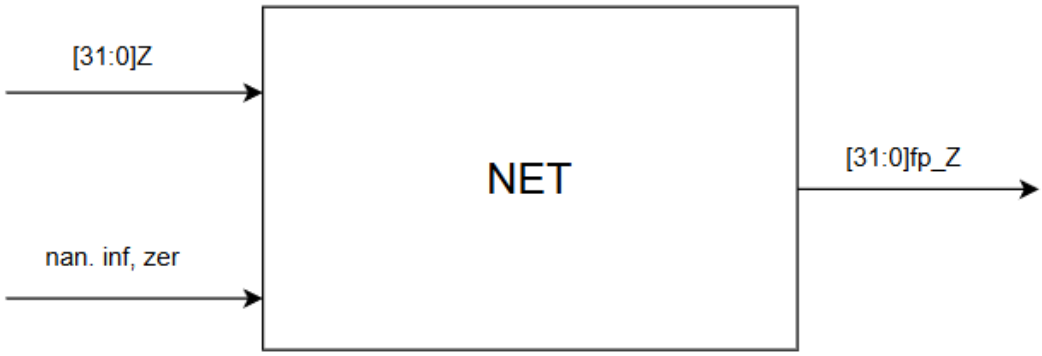

Figura 3.15. Diagrama de señales del NET

Este bloque es aún más sencillo, se encarga de aplicar las excepciones, dando forma estándar a las salidas del multiplicador. Si un número es NaN la salida debe ser `0x7fc00000`. Si un número es infinito positivo la salida debe ser `0x7f800000`, si es infinito negativo `0xff800000`. Si el numero es cero, la salida debe ser `0x7f800000` o `0xff800000` dependiendo también si es positivo o negativo

Capítulo 4

Análisis de resultados

Una vez redactadas las aserciones, se procede a evaluar su efectividad mediante los reportes de cobertura formal. Además, se incluye un análisis detallado de los resultados, explicando los posibles motivos por los cuales algunas aserciones pudieron fallar. Este paso es fundamental, limitarse a reportar que una propiedad "falló" sin analizar sus causas aporta poco valor al proceso de verificación. El verdadero aporte proviene de interpretar los resultados, identificar las razones del fallo y comprender su impacto en el diseño.

4.0.1. Análisis de resultados del sumador

```
man_full = fp_simple_add(fp_a, fp_b);
END_TO_END_SUMA: assert ((!(&fp_a[30:23] || &fp_b[30:23]) && (fp_a[31] ==
    fp_b[31]) && r_mode == 3'b001 && (man_full[30:23] != 8'hFF)) ->
   fp_result == man_full);

END_TO_END_RESTA: assert((!(&fp_a[30:23] || &fp_b[30:23]) && (fp_a[31] !=
    fp_b[31]) && r_mode == 3'b001) -> fp_result == fp_simple_sub(fp_a,
   fp_b));
```

Estas aserciones son particularmente importantes porque validan el comportamiento end-to-end del bloque bajo condiciones normales, sin operandos especiales ni escenarios extremos (inf ni NaN). Que estas propiedades no se cumplan indica que el diseño no está produciendo el resultado esperado en operaciones básicas. Se calculó el valor teórico ideal y se comparó contra el obtenido (las funciones se encuentran en anexos, figura1).

```
COMM_MANTISSA: assert (result_ab[22:0] == result_ba[22:0]);
```

Chequeando la propiedad de conmutatividad, falla la mantisa.

```
ALIGN_A_SUBNORM: assert ((is_subnormal_a && !is_subnormal_b &&
   !is_zero_b && !is_special_b)->
            (mantissa_b_aligned == mantissa_b && (mantissa_a_aligned ==
   mantissa_a >> (expo_diff - 1))));

ALIGN_B_SUBNORM: assert ((!is_subnormal_a && is_subnormal_b &&
   !is_zero_a && !is_special_a)->
            (mantissa_a_aligned == mantissa_a && (mantissa_b_aligned ==
   mantissa_b >> (expo_diff - 1))));
```

Que ambas aserciones hayan fallado no es casualidad. Sería inusual que una fallara y la otra no, ya que el RTL es exactamente el mismo: únicamente se instancia dos veces con diferentes entradas. El origen del problema parece estar relacionado con el manejo de números subnormales. El autor del módulo de suma no documenta en detalle cómo sus bloques internos realizan los cálculos, por lo que nos basamos estrictamente en el estándar IEEE-754. Todo indica que el fallo proviene de un ajuste incorrecto de la mantisa cuando uno de los operandos es subnormal y el otro no, lo cual genera un resultado inconsistente con lo esperado.

```
ALIGN_EXP_SUBNORMAL: assert ((is_subnormal_a && is_subnormal_b) ->
            (exponent_common) == 8'd0);
```

Nuevamente, el problema apunta a un manejo incorrecto de los números subnormales. Esto puede deberse a dos situaciones:

- El diseño modifica alguno de los operandos antes de calcular el exponente común.
- Asigna un exponente distinto de cero.

Si los valores fueron correctamente desempacados como subnormales, ambos deberían presentar un exponente igual a cero. Cualquier alteración previa a la etapa de alineamiento provocaría resultados inconsistentes, lo que explicaría el fallo observado.

```
ALIGN_SUBNORMAL: assert ((is_subnormal_a && is_subnormal_b) ->
            ((mantissa_b_aligned == mantissa_b) && (mantissa_a_aligned ==
    mantissa_a)));
```

Nuevamente, el problema apunta a un manejo incorrecto de los números subnormales. Si los ambos son subnormales, el exponente de ambos es 0, o en su defecto iguales, por lo que ninguna mantisa debería ser desplazada.

```
NORM_CARRY_EXPO_SUB: assert (( mantissa_sum[23] && (exponent_common ==
   8'b0) && mantissa_sum != 0) ->
            ((exponent_out == exponent_common + 1)));
```

Esta aserción está destinada para verificar un caso específico: cuando el operando tiene un exponente subnormal (igual a cero), pero la suma de las mantisas produce un carry adicional. Lo que implica incrementar el exponente y hacer que el número deje de ser subnormal para convertirse en un número normal.

```
NORM_CARRY_MANTISSA_SUBN: assert (( mantissa_sum[23] && (exponent_common
   == 8'b0) && mantissa_sum != 0) ->
            ((mantissa_ext[25:3] == mantissa_sum[23:0])));
```

Igual que el caso anterior. Si el número deja de ser subnormal, la mantisa ahora tiene MSB igual a 1, y se debe tratar como tal. Véase que se concede cierta flexibilidad, pues únicamente se comprueba que la mantisa original quede ubicada en la posición correcta después de la normalización, sin considerar explícitamente el bit implícito. Esto permite validar el comportamiento esencial del diseño sin introducir restricciones adicionales que podrían ocultar el problema real.

```
NORM_SHIFT_MANTISSA_NORM_A_SUBN: assert (((mantissa_sum != 0)
                            && (!mantissa_sum[24])
                            && (!mantissa_sum[23])
                            && (exponent_common > 0)
                            && (exponent_common <= shift_amount)) ->
            ((mantissa_ext[25:3] == mantissa_sum[23:0] «
   (exponent_common))));
```

Esta aserción evalúa si el diseño maneja correctamente el caso en que, debido a múltiples corrimientos hacia la izquierda para localizar el bit MSB igual a 1, el número termina convirtiéndose en subnormal.

```
CASO_INF_POSITIVO: assert (((fp_a == 32'h7f800000 && !(fp_b[30:23] ==
   8'hFF && |fp_b[22:0])) || (fp_b == 32'h7f800000 && !(fp_a[30:23] ==
   8'hFF && |fp_a[22:0]))) -> fp_result == 32'h7f800000);

CASO_INF_NEGATIVO: assert (((fp_a == 32'hff800000 && !(fp_b[30:23] ==
   8'hFF && |fp_b[22:0])) || (fp_b == 32'hff800000 && !(fp_a[30:23] ==
   8'hFF && |fp_a[22:0]))) -> fp_result == 32'hff800000);
```

En los dos casos relacionados con infinitos, el autor del sumador no especifica qué bloque ni qué lógica interna se encarga de generar las señales de inf. Por ello, se basó exclusivamente en el comportamiento esperado según el estándar, identificando cuándo debe producirse un infinito y observando la señal de salida. Sin embargo, los resultados muestran que el diseño no lo implementa adecuadamente.

```
UNDERFLOW: assert (!(fp_b == {!fp_a[31], fp_a[30:0]}) &&
   (fp_result_wire[30:0] == 31'b0) -> underflow);

OVERFLOW: assert (((exponent_common + carry_out + mantissa_sum[24]) >=
   255) -> overflow);
```

Igual que lo anterior, el autor no especifica qué bloque ni qué lógica interna se encarga de generar las señales de overflow o underflow. Por ello, se basó exclusivamente en identificar cuándo deben producirse y observar la señal de salida.

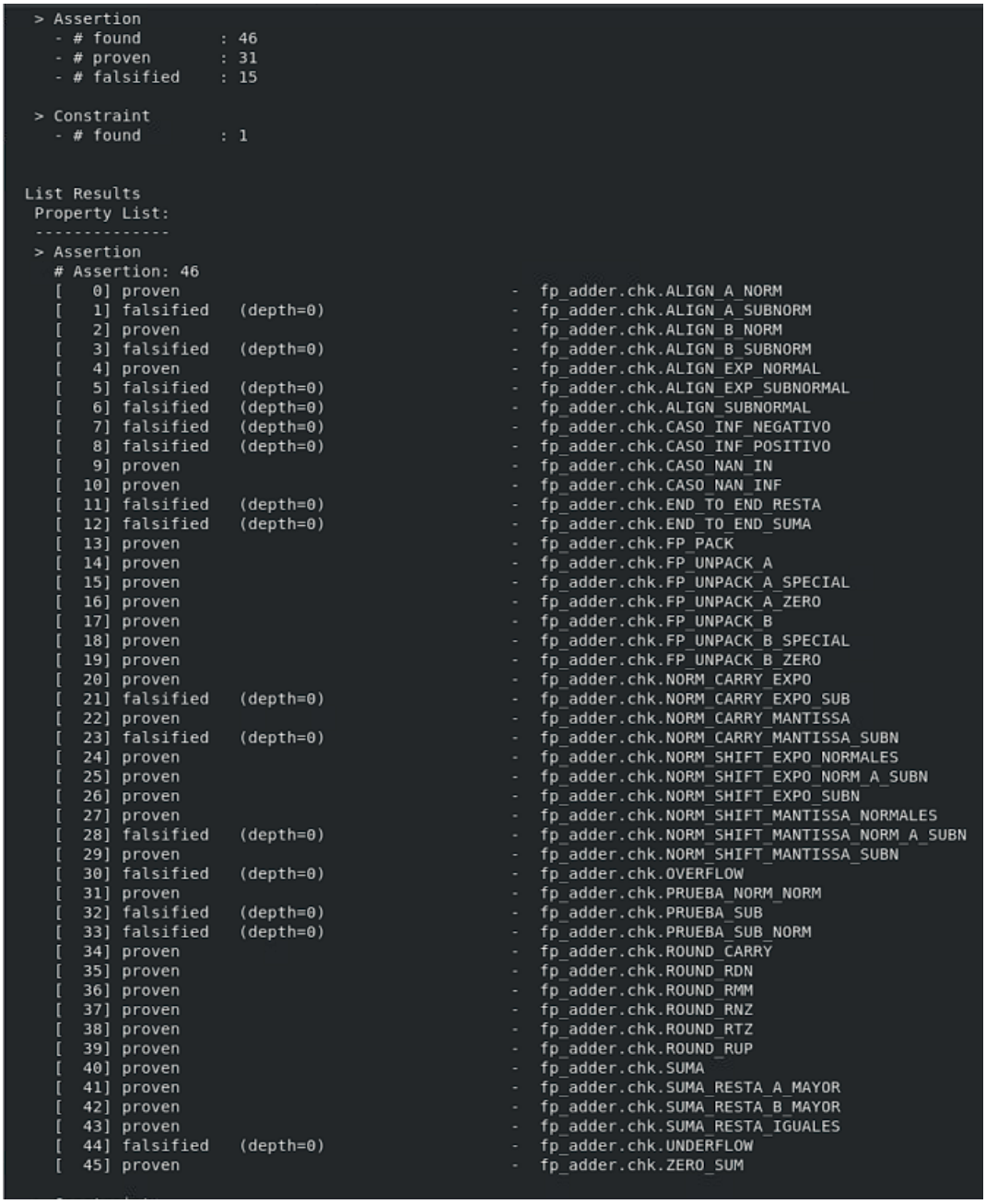

Figura 4.1. Resumen de resultados de las aserciones para el sumador

En conjunto, estos resultados muestran que el sumador presenta fallos importantes, especialmente en el manejo de números subnormales durante las etapas de alineamiento del exponente y normalización. También se observan errores menores en la generación de las banderas de underflow, overflow e infinito, mientras que el tratamiento de los valores Not a Number (NaN) sí parece estar correctamente implementado.

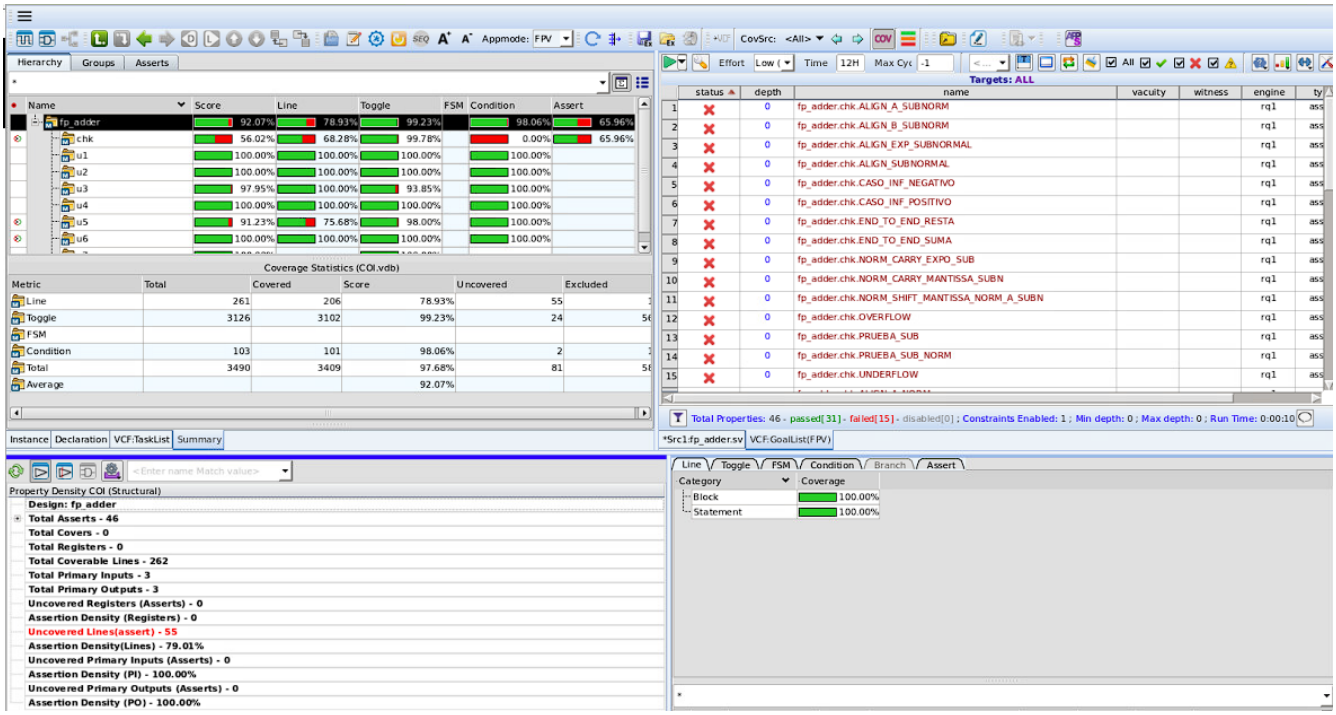

Figura 4.2. Cobertura de las aserciones para el sumador

Por otra parte, el 65.96 % corresponde a la cobertura de las aserciones (cover properties), es decir, al porcentaje de escenarios que fueron alcanzados exitosamente. Esta métrica, por sí sola, no resulta especialmente significativa, sin embargo, al combinarla con el resto de métricas de cobertura que proporciona la herramienta, es posible realizar un análisis más completo del módulo.

VCF alcanzó casi un 100 % (solo para el align_exponents 97.95 % y para el normalizador un 91.23 %) de cobertura estructural, lo que indica que prácticamente no existen segmentos de código muerto (dead code) ni partes del RTL que la herramienta no haya explorado. No obstante, el COI obtenido fue de 79.01 %. Esto indica que la herramienta exploró gran parte del diseño. Más sin embargo, el sumador presenta demasiados fallos en las propiedades (15 de 31) y tampoco realiza correctamente el cálculo de suma o resta para todos los valores.

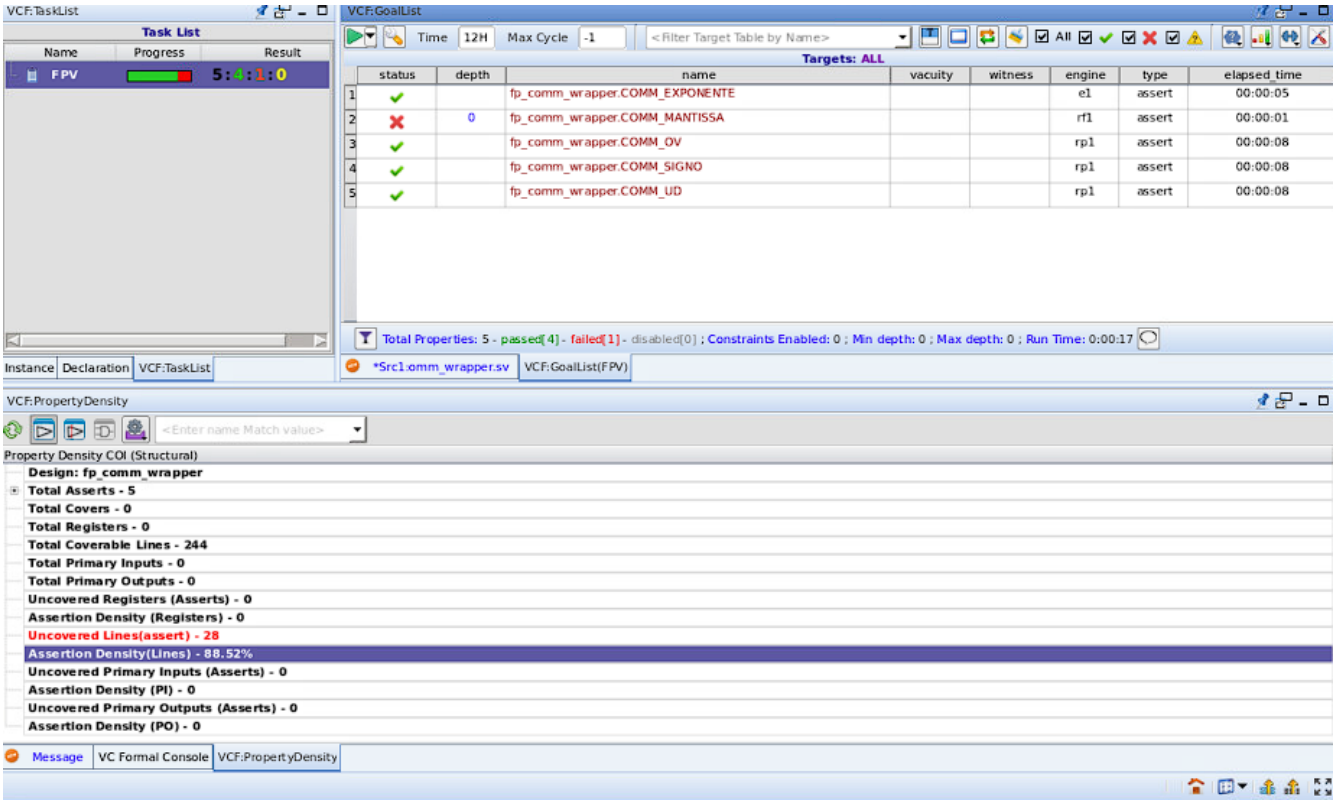

Figura 4.3. Cobertura suma conmutativa

Para la verificación de la propiedad de conmutatividad en la operación de suma fue necesario crear un archivo independiente, ya que el *bind* presentaba conflictos en el entorno original. En este nuevo archivo se comprobó la equivalencia fp_adder(a,b) = fp_adder(b,a). Sin embargo, la verificación mostró inconsistencias en la mantisa, lo que evidencia que el diseño no está preservando correctamente la conmutatividad.

4.0.2. Análisis de resultados del multiplicador de punto flotante

```
BOOTH_NORM_X_SUB: assert (((frc_X == equi_sub1[22:0]) && (frc_Y ==
    equi_norm2[22:0])) ->
                              (frc_Z_full == {1'b0, frc_X} * {1'b1,
    frc_Y}));

BOOTH_SUB_X_SUB: assert (((frc_X == equi_sub1[22:0]) && (frc_Y ==
    equi_sub2[22:0])) ->
                              (frc_Z_full == {1'b0, frc_X} * {1'b0,
    frc_Y}));
```

Estas aserciones se diseñaron con el objetivo de evaluar las capacidades reales del multiplicador. El autor original del diseño había indicado previamente que su módulo no manejaba correctamente los números subnormales. Todo apunta a que el multiplicador procesa de la misma forma los operandos normales y subnormales, probablemente para evitar incorporar lógica adicional y compleja.

Como consecuencia, cuando detecta que alguno de los operandos es subnormal, el módulo produce un resultado cero, independientemente del cálculo que debería realizarse según el estándar. No se considera un fallo importante.

```
EXP_UDRF_MANTISA: assert ((udrf && !Xsub && !Ysub) -> ((frc_Z) ==
    23'b0));

Z_PRUEBA_ZERO: assert ((fp_X == 32'h00000000 && fp_Y == 32'h00000000 &&
    r_mode == 3'b001) ->
                          (fp_Z == 32'h00000000) && !udrf);
```

El primer fallo mayor se encuentra en el manejo de las banderas, comenzando por la de underflow. El problema aparece cuando el resultado de la operación es exactamente cero o un valor subnormal, en estos casos, el autor del diseño activa la bandera de underflow. Esto constituye un error, ya que la activación indebida de esta señal puede afectar directamente el control de la ALU.

Por ejemplo, si la bandera de underflow se enciende de forma incorrecta, el controlador de la ALU podría interpretar que el resultado es inválido o no confiable, descartándolo o tomando decisiones equivocadas en etapas posteriores, cuando debió ser cero. Por el contrario, si el controlador ignora sistemáticamente esta bandera debido a su comportamiento erróneo, podría continuar realizando cálculos asumiendo que el resultado fue cero, generando incoherencias en la propagación de valores.

En ambos casos, el manejo incorrecto de la bandera compromete la integridad del flujo de operación

de la ALU.

```
NET_INF: assert (inf -> fp_Z == {z[31], 8'hff, 23'b0});

NET_ZERO: assert (zer -> fp_Z == {z[31], 31'b0});
```

El error relacionado con las banderas de Inf y Zero parece deberse a una condición de carrera. Por algún motivo, ambas señales llegan activas simultáneamente, lo cual es incorrecto. Dentro del bloque NET existe una lógica de prioridad (NAN $\rightarrow$ INF $\rightarrow$ ZERO) que selecciona la bandera correspondiente según el orden definido. Sin embargo, si dos o más banderas se activan al mismo tiempo, el bloque NET elige una de ellas basándose únicamente en la prioridad, ocultando el hecho de que ocurrió una activación múltiple indebida.

Este problema se confirma al evaluar casos en los que se excluye explícitamente el resto de las banderas, cuando las condiciones garantizan que solo una bandera puede estar activa, el error no se presenta. Esto indica que la falla proviene de la generación de las banderas antes de la etapa NET y que existe una condición de carrera o falta de exclusividad en la lógica que las produce.

```
BOOTH_FULL: assert (frc_Z_full == {1'b1, frc_X} * {1'b1, frc_Y});
```

Por último, hubo una única aserción cuyo resultado quedó inconcluso. La herramienta no encontró errores, pero tampoco logró explorar todos los casos posibles, por lo que no pudo marcarla como verificada.

```
Summary Results
 Property Summary: FPV
 ----------------
 > Assertion
   - # found        : 34
   - # proven       : 26
   - # falsified    : 6
   - # inconclusive : 2

 > Constraint
   - # found        : 1


List Results
 Property List:
 -------------
 > Assertion
   # Assertion: 34
   [   0] inconclusive                          -  fp_mul.chk.BOOTH_FULL
   [   1] proven                                -  fp_mul.chk.BOOTH_MAXFRAC_X_MAXFRAC
   [   2] proven                                -  fp_mul.chk.BOOTH_MAXFRAC_X_NORM
   [   3] proven                                -  fp_mul.chk.BOOTH_MINFRAC
   [   4] proven                                -  fp_mul.chk.BOOTH_NORM_X_NORM
   [   5] falsified   (depth=0)                 -  fp_mul.chk.BOOTH_NORM_X_SUB
   [   6] falsified   (depth=0)                 -  fp_mul.chk.BOOTH_SUB_X_SUB
   [   7] proven                                -  fp_mul.chk.EXC_INF
   [   8] proven                                -  fp_mul.chk.EXC_NAN
   [   9] proven                                -  fp_mul.chk.EXC_SUB_SON_ZERO
   [  10] proven                                -  fp_mul.chk.EXC_ZER
   [  11] proven                                -  fp_mul.chk.EXP_NORM
   [  12] proven                                -  fp_mul.chk.EXP_OVRF
   [  13] proven                                -  fp_mul.chk.EXP_UDRF
   [  14] falsified   (depth=5)                 -  fp_mul.chk.EXP_UDRF_MANTISA
   [  15] proven                                -  fp_mul.chk.MUL_SUB_POR_SUB
   [  16] proven                                -  fp_mul.chk.MUL_SUB_SON_ZERO
   [  17] proven                                -  fp_mul.chk.MUL_ZERO_POR_NUM
   [  18] proven                                -  fp_mul.chk.MUL_ZERO_POR_ZERO
   [  19] falsified   (depth=0)                 -  fp_mul.chk.NET_INF
   [  20] proven                                -  fp_mul.chk.NET_NAN
   [  21] proven                                -  fp_mul.chk.NET_Z
   [  22] falsified   (depth=17)                -  fp_mul.chk.NET_ZERO
   [  23] proven                                -  fp_mul.chk.NORM_MSB_UNO
   [  24] inconclusive(depth=64)                -  fp_mul.chk.NORM_SHIFT_MANTISSA
   [  25] proven                                -  fp_mul.chk.NORM_ZERO
   [  26] proven                                -  fp_mul.chk.ROUND_CARRY
   [  27] proven                                -  fp_mul.chk.ROUND_RDN
   [  28] proven                                -  fp_mul.chk.ROUND_RMM
   [  29] proven                                -  fp_mul.chk.ROUND_RNZ
   [  30] proven                                -  fp_mul.chk.ROUND_RTZ
   [  31] proven                                -  fp_mul.chk.ROUND_RUP
   [  32] proven                                -  fp_mul.chk.ROUND_SIGN
   [  33] falsified   (depth=0)                 -  fp_mul.chk.Z_PRUEBA_ZERO

 > Constraint
   # Constraint: 1
   [  34] constrained                           -  fp_mul.chk.unnamed$$_0
```

Figura 4.4. Resumen de resultados de las aserciones para el multiplicador

En conjunto, estos resultados muestran que el multiplicador presenta un único fallo importante, relacionado con el manejo de la condición de underflow, el cual podría comprometer el correcto control de la ALU.

Además, se identificaron fallos menores en el bloque EXC, donde deberían generarse exclusivamente una de las banderas NAN, INF o ZERO.

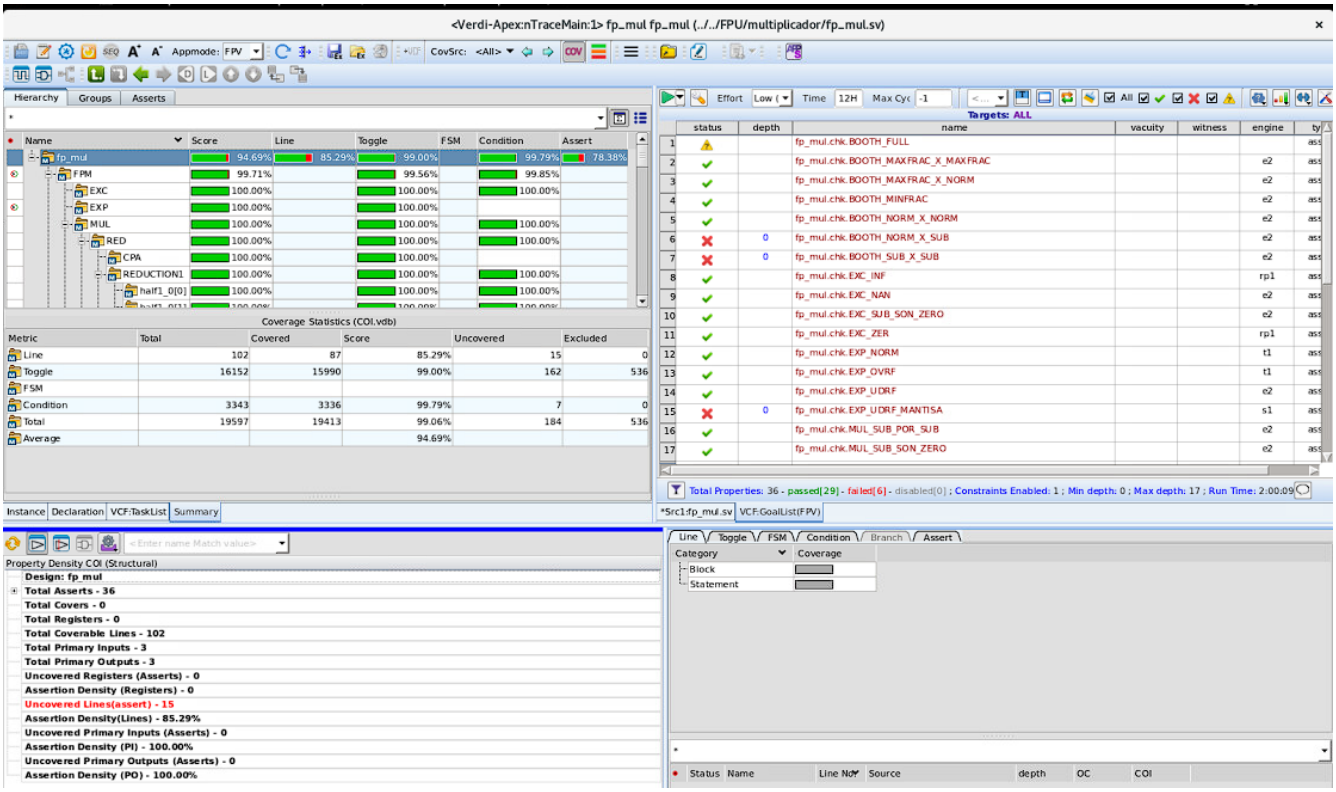

Figura 4.5. Cobertura de las aserciones para el multiplicador

El 79.41 % corresponde a la cobertura de las cover properties, es decir, al porcentaje de escenarios que fueron alcanzados exitosamente durante la verificación. En conjunto con un COI del 85 %, esto indica que la herramienta necesitó explorar gran parte del diseño. Esto sugiere que las aserciones fueron significativas y que el diseño es robusto, ya que la herramienta debió profundizar considerablemente para encontrar fallos. También es importante recalcar que todas las pruebas de propiedades black-box fueron exitosas.

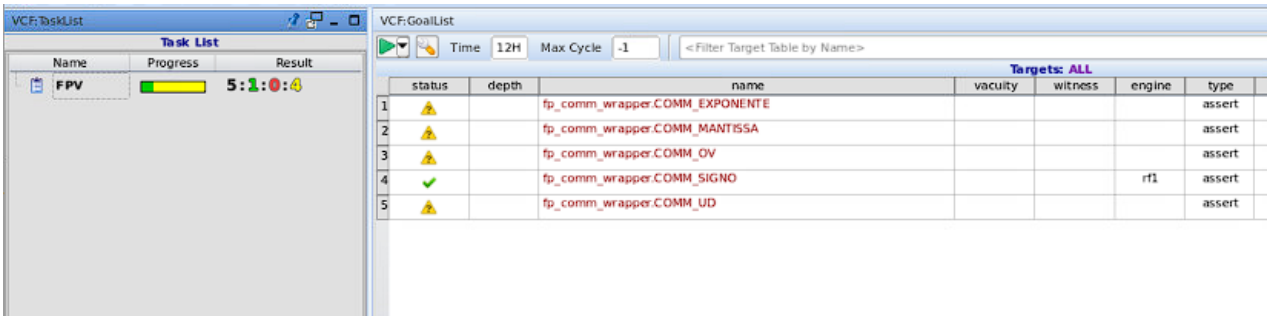

Figura 4.6. Cobertura de las aserciones para el multiplicador

De igual manera, la herramienta no es capaz de determinar si el multiplicador cumple con la propiedad de conmutatividad, esto relacionado por el gran cálculo computacional que representa realizar el booth encoding.

Capítulo 5

Conclusiones y Recomendaciones

El desarrollo de este proyecto permitió realizar un análisis exhaustivo y profundo del funcionamiento interno de la ALU de punto flotante destinado al microcontrolador SIWA. Mediante un enfoque de verificación formal basado en SystemVerilog Assertions, se logró evaluar cada etapa crítica del sumador y del multiplicador, identificando tanto fallos funcionales como comportamientos correctos que confirman la solidez de ciertas rutas internas del diseño. A excepción de la etapa de booth encoding, lo que requiere el uso de otro enfoque o técnicas avanzadas para reducir el espacio de búsqueda del FPV, manteniendo la misma robustez y confianza en las aserciones.

Se diseñó un entorno de verificación estructurado, estable y adecuado, capaz de albergar todas las aserciones necesarias sin modificar el RTL original. Dicho entorno permitió que el motor formal explorara casi la totalidad del diseño, demostrando que la metodología seleccionada fue apropiada para los objetivos planteados.

Actualmente ninguno de los diseños cumple de forma estricta con la especificación RISCV F, por lo que es necesario aplicar las modificaciones pertinentes. El sumador presenta fallos graves: pérdida de conmutatividad, errores en el cálculo de la suma y la resta, manejo incorrecto de números subnormales y anomalías en la mantisa. Estos problemas indican la necesidad de revisar las etapas clave del diseño. El multiplicador presenta errores en el manejo de las banderas e incorporar el manejo adecuado para los números subnormales.

Por otro lado, con respecto a la arquitectura interna de los bloques y en las rutas críticas especificadas por sus diseñadores, las aserciones demostraron que el sumador no cumple con las expectativas y el multiplicador, si hace el cálculo correcto para los parámetros de operación diseñados (números normales), pero presenta errores con el manejo de las banderas.

5.1. Recomendaciones para el sumador

El mayor número de fallos provino de un incorrecto procesamiento de números subnormales en las etapas de alineamiento y normalización. Se recomienda rediseñar completamente la lógica que

determina el desplazamiento de mantisas, el ajuste del exponente y la transición entre valores normales - subnormales.

En general, la etapa normalize_result mostró comportamientos inconsistentes, especialmente en casos de desplazamientos múltiples y normalizaciones que deberían convertir un número en subnormal. Se aconseja revisar su implementación o incluso considerar un rediseño más explícito basado en el comportamiento estándar IEEE-754.

Asegurar la consistencia en las banderas de overflow, underflow e infinito. Se recomienda definir claramente en qué bloque deben generarse estas señales y unificar su lógica, ya que actualmente existen discrepancias y condiciones no controladas.

5.2. Recomendaciones para el multiplicador

Actualmente la bandera de underflow se activa de manera incorrecta según el estándar, pero cuando vemos la aplicación original, donde se decide redondear los valores subnormales a cero, el manejo de underflow es correcto. Este comportamiento podría producir errores funcionales y posibles inconsistencias en el controlador de la ALU.

Se recomienda corregir las condiciones de carrera que permiten que varias banderas se activen simultáneamente. La lógica debe garantizar una única bandera activa por operación y luego respetar el orden de prioridad estándar. Sin embargo, el resultado final es el que prevalece, y lo calcula correctamente.

Bibliografía

[1] RISC-V International, *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*, 2025. [Online]. Available: https://riscv.org/technical/specifications/

[2] V. Bernard, "Implementation and contrast of area and power-e cient single-precision oating-point multipliers for the siwa microcontroller," 2020.

[3] R. García, A. Chacón, R. Castro, A. Arnaud, M. Miguez, J. Gak, R. Molina, G. Madrigal, M. Oviedo, E. Solera, D. Salazar, D. Sanchez, M. Fonseca, M. Arrieta, and R. Rimolo, "Siwa: a risc-v rv32i based micro-controller for implantable medical applications," *IEEE Xplore*, 2020, die.ucu.edu.uy. [Online]. Available: https://ieeexplore.ieee.org/document/9068952

[4] S. Cabrera, "Diseño de instrucciones de la extensión estándar para punto flotante de precisión simple rv32f y su respectiva unidad aritmética lógica," 2025.

[5] E. Seligman, T. Schubert, and M. V. Kumar, *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan-Kaufmann, 2015.

[6] J. Umaña. (2016) Historia, ciencia y tecnología, protagonistas en el 40 aniversario de la escuela de ingeniería electrónica. En línea, tec.ac.cr. [Online]. Available: https://www.tec.ac.cr/hoyeneltec/2016/08/09/historia-ciencia-tecnologia-protagonistas-40-aniversario-escuela-ingenieria-electronica

[7] Tecnológico de Costa Rica. (2024) Escuela de ingeniería electrónica. En línea, tec.ac.cr. [Online]. Available: https://www.tec.ac.cr/escuela-ingenieria-electronica

[8] J. Umaña. (2021) Microprocesador integralmente diseñado en costa rica demuestra altas capacidades tecnológicas. En línea, tec.ac.cr. [Online]. Available: https://www.tec.ac.cr/hoyeneltec/2021/04/27/microprocesador-integralmente-disenado-costa-rica-demuestra-altas-capacidades

[9] W. K. Lam., *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall PTR, 2005.

[10] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[11] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std. IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.

[12] H.-Schmidt, "Ieee-754 floating point converter," https://www.h-schmidt.net/FloatConverter/IEEE754.html, 2024.

[13] M. V. A. K. Kumar, T. Schubert, and E. Seligman, *Formal Verification: An Essential Toolkit for Modern VLSI*. Amsterdam: Elsevier / Morgan Kaufmann, 2015.

[14] S. Roy, "Best electronic design automation (eda) software and tools list 2025," *Techjockey Blog*, 2025. [Online]. Available: https://www.techjockey.com/blog/best-electronic-design-automation-eda-software/

[15] I. Synopsys, *VC Formal User Guide*, Synopsys, Mountain View, CA, 2023, product Documentation.

[16] J. Young. (2025) Representative sample: Definition, importance, and examples. Investopedia. [Online]. Available: https://www.investopedia.com/terms/r/representative-sample.asp

Capítulo 6

Anexos

Anexo A: Conjunto de Aserciones Implementadas para Aserciones del fp_adder

```
man_full = fp_simple_add(fp_a, fp_b);

END_TO_END_SUMA: assert ((!(&fp_a[30:23] || &fp_b[30:23])
                        && (fp_a[31] == fp_b[31])
                        && r_mode == 3'b001
                        && (man_full[30:23] != 8'hFF)) -> fp_result == man_full);

END_TO_END_RESTA: assert((!(&fp_a[30:23] || &fp_b[30:23])
                        && (fp_a[31] != fp_b[31])
                        && r_mode == 3'b001) -> fp_result == fp_simple_sub(fp_a, fp_b));
```

Figura 1. Código aserciones black-box, suma y resta

```
  function automatic [31:0] fp_simple_add
(
  input logic [31:0] a,
  input logic [31:0] b
);
  logic sign;
  logic [7:0]  exp;
  logic [23:0] mant_a, mant_b;
  logic [24:0] mant_sum;

  if (a[30:23] > b[30:23]) exp = a[30:23];
  else exp = b[30:23];

  mant_a = {|a[30:23], a[22:0]};
  mant_b = {|b[30:23], b[22:0]};

  mant_sum = mant_a + mant_b;
  sign = a[31];
  if (mant_sum[24]) begin
    mant_sum = mant_sum >> 1;
    exp = exp + 1;
  end

  return {sign, exp, mant_sum[22:0]};
endfunction
```

Figura 2. Código función suma ideal

```
  function automatic [31:0] fp_simple_sub
(
  input logic [31:0] a,
  input logic [31:0] b
);
  logic sign;
  logic [7:0]  exp;
  logic [23:0] mant_a, mant_b;
  logic [23:0] mant_sum;
  logic [7:0] dez;

  mant_a = {|a[30:23], a[22:0]};
  mant_b = {|b[30:23], b[22:0]};

  if (a[30:23] > b[30:23]) begin
    exp = a[30:23];
    sign = a[31];
    mant_sum = mant_a - mant_b;
  end
  else begin
    exp = b[30:23];
    sign = b[31];
    mant_sum = mant_b - mant_a;
  end
  if (exp > leading_zero_count(mant_sum)) dez = leading_zero_count(mant_sum);
  else dez = exp;
  if (!mant_sum[23]) begin
    mant_sum = mant_sum << (dez);
    exp = exp - dez;
  end

  return {sign, exp, mant_sum[22:0]};
endfunction
```

Figura 3. Código función resta ideal

ALU_vcf > adder > omm_wrapper.sv

```
1  module fp_comm_wrapper;
2
3    logic [31:0] a, b;
4    logic [31:0] result_ab, result_ba;
5    logic [2:0]  rmode;
6    logic ov1, ud1, ov2, ud2;
7
8    // DUT original
9    fp_adder dut_ab (
10     .fp_a(a), .fp_b(b),
11     .r_mode(rmode),
12     .fp_result(result_ab),
13     .overflow(ov1),
14     .underflow(ud1)
15   );
16
17   // DUT con inputs invertidos
18   fp_adder dut_ba (
19     .fp_a(b), .fp_b(a),
20     .r_mode(rmode),
21     .fp_result(result_ba),
22     .overflow(ov2),
23     .underflow(ud2)
24   );
25
26     always_comb begin
27         COMM_SIGNO: assert (result_ab[31] == result_ba[31]);
28         COMM_EXPONENTE: assert (result_ab[30:23] == result_ba[30:23]);
29         COMM_MANTISSA: assert (result_ab[22:0] == result_ba[22:0]);
30         COMM_OV: assert (ov1 == ov2);
31         COMM_UD: assert (ud1 == ud2);
32     end
33
34 endmodule
35
```

Figura 4. Código aserciones black-box, conmutatividad

```
//Caso de esquina 0 + 0 = 0
ZERO_SUM: assert ((fp_a == 32'h00000000 && fp_b == 32'h00000000) ->
                  (fp_result == 32'h00000000 && overflow == 0 && underflow == 0));

//fp_unpack de cada valor es el correcto
FP_UNPACK_A: assert (((fp_a[30:23] != 8'hFF) && (fp_a[30:0] != 31'd0)) ->
                  ((sign_a == fp_a[31]) && (exponent_a == fp_a[30:23]) && (mantissa_a == {|fp_a[30:23], fp_a[22:0]})));

FP_UNPACK_B: assert (((fp_b[30:23] != 8'hFF) && (fp_b[30:0] != 31'd0)) ->
                  ((sign_b == fp_b[31]) && (exponent_b == fp_b[30:23]) && (mantissa_b == {|fp_b[30:23], fp_b[22:0]})));

//Identifica que un valor especial: NaN o INF
FP_UNPACK_A_SPECIAL: assert ((fp_a[30:23] == 8'hFF) ->
                  ((sign_a == fp_a[31]) && (exponent_a == 8'hFF) && (mantissa_a == {1'b0, fp_a[22:0]}) && is_special_a));

FP_UNPACK_B_SPECIAL: assert ((fp_b[30:23] == 8'hFF) ->
                  ((sign_b == fp_b[31]) && (exponent_b == 8'hFF) && (mantissa_b == {1'b0, fp_b[22:0]}) && is_special_b));

//Identifica que un valor es cero
FP_UNPACK_A_ZERO: assert ((fp_a[30:0] == 31'b0) ->
                  (is_zero_a == 1));

FP_UNPACK_B_ZERO: assert ((fp_b[30:0] == 31'b0) ->
                  (is_zero_b == 1));
```

Figura 5. Código aserciones bloque fp_unpack

```
//Mantissa_a necesita ser alineada si exponent_b > exponent_a casos normales
expo_diff = (exponent_b - exponent_a);

ALIGN_A_NORM: assert (((exponent_b > exponent_a) && !is_subnormal_a && !is_subnormal_b)->
                (mantissa_b_aligned == mantissa_b && (mantissa_a_aligned == mantissa_a >> (expo_diff))));

//Mantissa_a necesita ser alineada si exponent_b > exponent_a casos subnormales
ALIGN_A_SUBNORM: assert ((is_subnormal_a && !is_subnormal_b && !is_zero_b && !is_special_b)->
                (mantissa_b_aligned == mantissa_b && (mantissa_a_aligned ==  mantissa_a >> (expo_diff - 1))));

//Mantissa_b necesita ser alineada si exponent_a > exponent_b casos normales
expo_diff = (exponent_a - exponent_b);
ALIGN_B_NORM: assert (((exponent_a > exponent_b)&& !is_subnormal_a && !is_subnormal_b)->
                (mantissa_a_aligned == mantissa_a && (mantissa_b_aligned == mantissa_b >> (expo_diff))));

//Mantissa_b necesita ser alineada si exponent_a > exponent_b casos subnormales
ALIGN_B_SUBNORM: assert ((!is_subnormal_a && is_subnormal_b && !is_zero_a && !is_special_a)->
                (mantissa_a_aligned == mantissa_a && (mantissa_b_aligned ==  mantissa_b >> (expo_diff - 1))));

//Alineamiento cuando ambos son subnormales
ALIGN_SUBNORMAL: assert ((is_subnormal_a && is_subnormal_b) ->
                ((mantissa_b_aligned == mantissa_b) && (mantissa_a_aligned == mantissa_a)));

//El exponente resultante es el mayor
ALIGN_EXP_NORMAL: assert ((!(is_subnormal_a || is_subnormal_b) && !is_special_a && !is_special_b) ->
                (exponent_common) == ((exponent_a > exponent_b) ? exponent_a : exponent_b));

//Exponente en ambos numeros subnormales
ALIGN_EXP_SUBNORMAL: assert ((is_subnormal_a && is_subnormal_b) ->
                (exponent_common) == 8'd0);
```

Figura 6. Código aserciones bloque align_exponents

```
//Suma de mantisas
SUMA: assert ((sign_a == sign_b) ->
                ((mantissa_sum == mantissa_a_aligned + mantissa_b_aligned) && (result_sign == sign_b)));

//Resta de mantisas cuando es A mayor
SUMA_RESTA_A_MAYOR: assert (((sign_a != sign_b) && (mantissa_a_aligned > mantissa_b_aligned))  ->
                ((mantissa_sum == (mantissa_a_aligned - mantissa_b_aligned) && (result_sign == sign_a))));

//Resta de mantisas cuando es A mayor
SUMA_RESTA_B_MAYOR: assert (((sign_a != sign_b) && (mantissa_b_aligned > mantissa_a_aligned))  ->
                ((mantissa_sum == (mantissa_b_aligned - mantissa_a_aligned) && (result_sign == sign_b))));

//Resta de mantisas cuando es B mayor
SUMA_RESTA_IGUALES: assert ((sign_a != sign_b && (mantissa_a_aligned == mantissa_b_aligned)) ->
                ((mantissa_sum == 0) && (result_sign == 0))); //Solo por que samuel lo define asi, no se norma
```

Figura 7. Código aserciones bloque add_sub_mantissas

```
shift_amount = leading_zero_count(mantissa_sum[23:0]); //Calculo util para saber cuanto dezplazamiento en caso de shift

//Si hay carry de la suma ajusta la mantizza
NORM_CARRY_EXPO: assert ((mantissa_sum[24] && !is_subnormal_a && !is_subnormal_b) ->
        ((exponent_out == exponent_common + 1)));

//Carry si el bit implicito tambien es 1 en subnormales
NORM_CARRY_EXPO_SUB: assert (( mantissa_sum[23] && (exponent_common == 8'b0) && mantissa_sum != 0) ->
        ((exponent_out == exponent_common + 1)));

//Si hay carry de la suma aumneta exponente en normalize
NORM_CARRY_MANTISSA: assert ((mantissa_sum[24] && !is_subnormal_a && !is_subnormal_b) ->
        ((mantissa_ext == {mantissa_sum,1'b0,1'b0})));  //{1'b0,mantissa_sum,1'b0}

//Carry si el bit implicito tambien es 1 en subnormales
NORM_CARRY_MANTISSA_SUBN: assert (( mantissa_sum[23] && (exponent_common == 8'b0) && mantissa_sum != 0) ->
        ((mantissa_ext[25:3] == mantissa_sum[23:0])));

//Ajuste normalize poner el primer 1 con shift a la derecha
NORM_SHIFT_MANTISSA_NORMALES: assert (((mantissa_sum != 0)
                        && (!mantissa_sum[24])
                        && (!mantissa_sum[23])
                        && (exponent_common > shift_amount)) ->
        (mantissa_ext[26:3] == (mantissa_sum[23:0]<<shift_amount)));

NORM_SHIFT_EXPO_NORMALES: assert (((mantissa_sum != 0)
                        && (!mantissa_sum[24])
                        && (!mantissa_sum[23])
                        && (exponent_common > shift_amount)) ->
        ((exponent_out == exponent_common - shift_amount)));
```

Figura 8. Código aserciones bloque normalize 1

```
//El resultado pasa de ser normal a subnormal
NORM_SHIFT_MANTISSA_NORM_A_SUBN: assert (((mantissa_sum != 0)
                                && (!mantissa_sum[24])
                                && (!mantissa_sum[23])
                                && (exponent_common > 0)
                                && (exponent_common <= shift_amount)) ->
            ((mantissa_ext[25:3] == mantissa_sum[23:0] << (exponent_common))));

NORM_SHIFT_EXPO_NORM_A_SUBN: assert (((mantissa_sum != 0)
                                && (!mantissa_sum[24])
                                && (!mantissa_sum[23])
                                && (exponent_common > 0)
                                && (exponent_common <= shift_amount)) ->
            ((exponent_out == 0)));

//Suma de sub normales no ocupa corrimiento
NORM_SHIFT_MANTISSA_SUBN: assert (((mantissa_sum != 0)
                                && (!mantissa_sum[24])
                                && (!mantissa_sum[23])
                                && (exponent_common == 8'b0)) ->
            mantissa_ext[25:3] == mantissa_sum[23:0]);

NORM_SHIFT_EXPO_SUBN: assert (((mantissa_sum != 0)
                                && (!mantissa_sum[24])
                                && (!mantissa_sum[23])
                                && (exponent_common == 8'b0)) ->
            ((exponent_out == exponent_common)));
```

Figura 9. Código aserciones bloque normalize 2

```
//Redondeo al mas cercano (pares en empate)
case ({mantissa_ext[2],(|mantissa_ext[1:0])})
  2'b00: mantissa_r = mantissa_ext[25:3];
  2'b01: mantissa_r = mantissa_ext[25:3];
  2'b10: mantissa_r = mantissa_ext[3] ? mantissa_ext[25:3] + 1'b1 : mantissa_ext[25:3]; //si es impar redondea para arriba
  2'b11: mantissa_r = mantissa_ext[25:3] + 1'b1;
//default: mantissa_r = mantissa_ext[25:3];
endcase

ROUND_RNZ: assert ((r_mode == 3'b000) -> mantissa_rounded == mantissa_r);

//Redondeo hacia cero
ROUND_RTZ: assert ((r_mode == 3'b001) -> mantissa_rounded == mantissa_ext[25:3]);

//Redondeo hacia abajo
case (result_sign)
  1'b0: mantissa_r = mantissa_ext[25:3];
  1'b1: mantissa_r = mantissa_ext[25:3] + 1'b1;
//default: mantissa_r = mantissa_ext[25:3];
endcase

ROUND_RDN: assert ((r_mode == 3'b010) -> mantissa_rounded == mantissa_r);

//Redondeo hacia arriba
case (result_sign)
  1'b0: mantissa_r = mantissa_ext[25:3] + 1'b1;
  1'b1: mantissa_r = mantissa_ext[25:3];
//default: mantissa_r = mantissa_ext[25:3];
endcase

ROUND_RUP: assert ((r_mode == 3'b011) -> mantissa_rounded == mantissa_r);
```

Figura 10. Código aserciones bloque round 1

```
//Redondeo al mas cercano (maxima magnitud en empate)
case (mantissa_ext[2])
  1'b0: mantissa_r = mantissa_ext[25:3];
  1'b1: mantissa_r = mantissa_ext[25:3] + 1'b1;
//default: mantissa_r = mantissa_ext[25:3];
endcase

ROUND_RMM: assert ((r_mode == 3'b100) -> mantissa_rounded == mantissa_r);

//Se genera carry por redondeo
carry = {1'b0, mantissa_ext[25:3]} + 1'b1;
ROUND_CARRY: assert ((carry_out) -> ((carry [23]) && (mantissa_rounded == mantissa_ext[25:3] + 1'b1)));
```

Figura 11. Código aserciones bloque round 2

```
//Se enpaqueta bien devuelta
FP_PACK: assert (fp_result_wire == {result_sign, exponent_final, mantissa_rounded});
```

Figura 12. Código aserciones bloque fp_pack

```
UNDERFLOW: assert (!(fp_b == {!fp_a[31], fp_a[30:0]}) && (fp_result_wire[30:0] == 31'b0) -> underflow);

OVERFLOW: assert (((exponent_common + carry_out + mantissa_sum[24]) >= 255) -> overflow);

CASO_NAN_IN: assert (((fp_a[30:23] == 8'hFF && |fp_a[21:0]) ||
        (fp_b[30:23] == 8'hFF && |fp_b[21:0])) -> fp_result == 32'h7fc00000);

CASO_NAN_INF: assert (((fp_a == 32'h7f800000 && fp_b == 32'hff800000) ||
        (fp_a == 32'hff800000 && fp_b == 32'h7f800000)) -> fp_result == 32'h7fc00000):

CASO_INF_POSITIVO: assert (((fp_a == 32'h7f800000 && !(fp_b[30:23] == 8'hFF && |fp_b[22:0])) ||
        (fp_b == 32'h7f800000 && !(fp_a[30:23] == 8'hFF && |fp_a[22:0]))) -> fp_result == 32'h7f800000)

CASO_INF_NEGATIVO: assert (((fp_a == 32'hff800000 && !(fp_b[30:23] == 8'hFF && |fp_b[22:0])) ||
        (fp_b == 32'hff800000 && !(fp_a[30:23] == 8'hFF && |fp_a[22:0]))) -> fp_result == 32'hff800000)

PRUEBA_SUB: assert ((fp_a == 32'h000a0000 && fp_b == 32'h000a0000 && r_mode == 3'b001) ->
            (fp_result == 32'h00140000));

PRUEBA_SUB_NORM: assert ((fp_a == 32'h01000000 && fp_b == 32'h00300000 && r_mode == 3'b001) ->
            (fp_result == 32'h01180000));

PRUEBA_NORM_NORM: assert ((fp_a == 32'h14300000 && fp_b == 32'h1FC00000&& r_mode == 3'b001) ->
            (fp_result == 32'h1FC00001));
```

Figura 13. Código aserciones extra

Anexo B: Conjunto de Aserciones Implementadas para Aserciones del fp_mul

```
Xsub  = !(|fp_X[30:23]); //Sub o cero
Xnif  = (fp_X[30:23] == 8'hFF); //Nan o inf
XZero = (fp_X[30:0] == 31'b0);

// Flags Y
Ysub  = !(|fp_Y[30:23]); //Sub o cero
Ynif  = (fp_Y[30:23] == 8'hFF); //Nan o inf
YZero = (fp_Y[30:0] == 31'b0);

equi_norm1 = 32'h402df854;
equi_norm2 = 32'h40490fdb;

equi_sub1 = 32'h002df854;
equi_sub2 = 32'h00490fdb;
```

Figura 14. Código variables utilices establecidas

```
//Numeros subnormales producen el mismo resultado que cero
MUL_SUB_SON_ZERO: assert (((Xsub && !Ynif) || (Ysub && !Xnif)) ->
                          (fp_Z == {(fp_X[31] ^ fp_Y[31]),31'b0}));

//Multiplicacion de dos numeros subnormales
MUL_SUB_POR_SUB: assert ((Xsub && Ysub) ->
                         (fp_Z == {(fp_X[31] ^ fp_Y[31]),31'b0}));

//Multiplicacion de 0 * 0
MUL_ZERO_POR_ZERO: assert ((XZero && YZero) ->
                           (fp_Z == {(fp_X[31] ^ fp_Y[31]),31'b0}));

//Multiplicacion de cero por cualquier numero que no se infinito o NaN
MUL_ZERO_POR_NUM: assert (((XZero && !Ynif) || (YZero && !Xnif)) ->
                          (fp_Z == {(fp_X[31] ^ fp_Y[31]),31'b0}));

//Multiplicacion por 1
MUL_UNO_POR_NUMX: assert ((!(Xsub || Xnif) && (r_mode == 3'b001) && ( fp_Y == 32'h3f800000)) ->
                          (fp_Z == fp_X));
MUL_UNO_POR_NUMY: assert ((!(Ysub || Ynif)&& (r_mode == 3'b001) && ( fp_X == 32'h3f800000)) ->
                          (fp_Z == fp_Y));
```

Figura 15. Código aserciones black-box de multiplicación

```
module fp_comm_wrapper;

  logic [2:0]rmode;
  logic [31:0]a, b;
  logic [31:0]result_ab, result_ba;
  logic ov1, ov2,ud1, ud2;
  // DUT original
  fp_mul dut_ab (
    .fp_X(a), .fp_Y(b),
    .r_mode(rmode),
    .fp_Z(result_ab),
    .ovrf(ov1),
    .udrf(ud1)
  );

  // DUT con inputs invertidos
  fp_mul dut_ba (
    .fp_X(b), .fp_Y(a),
    .r_mode(rmode),
    .fp_Z(result_ba),
    .ovrf(ov2),
    .udrf(ud2)
  );

    always_comb begin
        COMM_SIGNO: assert (result_ab[31] == result_ba[31]);
        COMM_EXPONENTE: assert (result_ab[30:23] == result_ba[30:23]);
        COMM_MANTISSA: assert (result_ab[22:0] == result_ba[22:0]);
        COMM_OV: assert (ov1 == ov2);
        COMM_UD: assert (ud1 == ud2);
    end

endmodule
```

Figura 16. Código aserciones de conmutatividad

```
//Booth encoding de las mantisas de dos numeros normales
BOOTH_NORM_X_NORM: assert (((frc_X == equi_norm1[22:0]) && (frc_Y == equi_norm2[22:0])) ->
                    (frc_Z_full == {1'b1, frc_X} * {1'b1, frc_Y}));//frc_Z_full = {1'b1, frc_X} * {1'b1, frc_Y};

//Booth encoding de las mantisas de valor maximo por un normal
BOOTH_MAXFRAC_X_NORM: assert (((fp_X == 32'h3fffffff) && (frc_Y == equi_norm2[22:0])) ->
                    (frc_Z_full == {1'b1, frc_X} * {1'b1, frc_Y}));

//Booth encoding de 2 mantisas de valor maximo
BOOTH_MAXFRAC_X_MAXFRAC: assert (((fp_X == 32'h3fffffff) && (fp_Y== 32'h3fffffff)) ->
                    (frc_Z_full == {1'b1, frc_X} * {1'b1, frc_Y}));

//Booth encoding de las mantisas subnormal por un normal
BOOTH_NORM_X_SUB: assert (((frc_X == equi_sub1[22:0]) && (frc_Y == equi_norm2[22:0])) ->
                    (frc_Z_full == {1'b0, frc_X} * {1'b1, frc_Y}));

//Booth encoding de las mantisas subnormal por un subnormal
BOOTH_SUB_X_SUB: assert (((frc_X == equi_sub1[22:0]) && (frc_Y == equi_sub2[22:0])) ->
                    (frc_Z_full == {1'b0, frc_X} * {1'b0, frc_Y}));

//Booth encoding de las mantisas valor minimo por cualquier valor
BOOTH_MINFRAC: assert ((!frc_X) ->
                    (frc_Z_full[45:23] == frc_Y));
```

Figura 17. Código aserciones bloque MUL

```
frc_Z_norm_check = (frc_Z_full[47])? frc_Z_full : {frc_Z_full[46:0],1'b0};//Dado el diagrama de Vianney

//Normalizacion de numeros
NORM_SHIFT_MANTISSA: assert ((frc_Z_norm[0] == |frc_Z_norm_check[21:0])
                    && (frc_Z_norm[26:1] == frc_Z_norm_check[47:22])
                    && (frc_Z_full[47] == norm_n));

//Despues de la normalizacion siempre el primer bit es 1 si el resultado no es subnormal
NORM_MSB_UNO: assert ((!Xsub && !Ynif && !Ysub && !Xnif) -> (frc_Z_norm[26] == 1'b1));

//Normalizacion de multiplicar por 0
NORM_ZERO: assert (((fp_X[31:0] == 31'b0) && !Ynif) -> (frc_Z_norm[25:3] == frc_Y));
```

Figura 18. Código aserciones bloque NORM

```
//Calculo del signo
ROUND_SIGN: assert (sign_Z == fp_X[31] ^ fp_Y[31]);

//Redondeo al mas cercano (pares en empate)
case ({frc_Z_norm[2],(|frc_Z_norm[1:0])})
    2'b00: mantissa_r = frc_Z_norm[25:3];
    2'b01: mantissa_r = frc_Z_norm[25:3];
    2'b10: mantissa_r = frc_Z_norm[3] ? frc_Z_norm[25:3] + 1'b1 : frc_Z_norm[25:3]; //si es impar redondea para arriba
    2'b11: mantissa_r = frc_Z_norm[25:3] + 1'b1;
    //default: mantissa_r = frc_Z_norm[25:3];
endcase

ROUND_RNZ: assert ((r_mode == 3'b000) -> frc_Z == mantissa_r);

//Redondeo hacia cero
ROUND_RTZ: assert ((r_mode == 3'b001) -> frc_Z == frc_Z_norm[25:3]);

//Redondeo hacia abajo
case (sign_Z)
    1'b0: mantissa_r = frc_Z_norm[25:3];
    1'b1: mantissa_r = frc_Z_norm[25:3] + 1'b1;
    //default: mantissa_r = frc_Z_norm[25:3];
endcase

ROUND_RDN: assert ((r_mode == 3'b010) -> frc_Z == mantissa_r);

//Redondeo hacia arriba
case (sign_Z)
    1'b0: mantissa_r = frc_Z_norm[25:3] + 1'b1;
    1'b1: mantissa_r = frc_Z_norm[25:3];
    //default: mantissa_r = frc_Z_norm[25:3];
endcase

ROUND_RUP: assert ((r_mode == 3'b011) -> frc_Z == mantissa_r);
```

Figura 19. Código aserciones bloque ROUND 1

```
//Redondeo al mas cercano (maxima magnitud en empate)
case (frc_Z_norm[2])
    1'b0: mantissa_r = frc_Z_norm[25:3];
    1'b1: mantissa_r = frc_Z_norm[25:3] + 1'b1;
    //default: mantissa_r = frc_Z_norm[25:3];
endcase

ROUND_RMM: assert ((r_mode == 3'b100) -> frc_Z == mantissa_r);

//Se genera carry por redondeo
carry = {1'b0, frc_Z_norm[25:3]} + 1'b1;
ROUND_CARRY: assert ((norm_r) -> (carry [23]));
```

Figura 20. Código aserciones bloque ROUND 2

```
bias = (norm_n||norm_r) ? 8'b01111110 : 8'b01111111; //si hubo un carry por round o normalizacion

//Calculo del exponenete
EXP_NORM: assert (exp_Z == fp_X[30:23]+fp_Y[30:23]-bias);

//Si ocurre un underflow por exponente
EXP_UDRF: assert (udrf -> ((fp_X[30:23]+fp_Y[30:23]) <= bias));
EXP_UDRF_MANTISA: assert ((udrf && !Xsub && !Ysub) -> ((frc_Z) == 23'b0));
//Si ocurre un overflow
EXP_OVRF: assert (ovrf -> ((fp_X[30:23]+fp_Y[30:23]) >= (bias + 255)));
```

Figura 21. Código aserciones bloque EXP

```
//Si algun input es subnormal, zero o se produce un underflow
EXC_ZER: assert ((udrf||Ysub||Xsub) -> zer);

//Chechk: numeros subnormales producen el mismo resultado que cero
EXC_SUB_SON_ZERO: assert ((Xsub) -> zer);

//Si algun input es inf o se produce un overflow
EXC_INF: assert (inf ->
                (( (fp_X[22:0] == 0) && &fp_X[30:23]) || ovrf
                || ((fp_Y[22:0] == 0) && &fp_Y[30:23])));

//Si algun input es NaN o se esta intentando multiplicar un inf por cero
EXC_NAN: assert (nan->
                (((|fp_X[22:0]) && &fp_X[30:23])
                || (Ysub && ((fp_X[22:0] == 0) && &fp_X[30:23]))
                || (Xsub && ((fp_Y[22:0] == 0) && &fp_Y[30:23]))
                || ((|fp_Y[22:0]) && &fp_Y[30:23])));
```

Figura 22. Código aserciones bloque EXC

```
z = {sign_Z, exp_Z, frc_Z};

NET_NAN: assert (nan -> fp_Z == 32'h7fc00000);

NET_INF: assert (inf -> fp_Z == {z[31], 8'hff, 23'b0});

NET_ZERO: assert (zer -> fp_Z == {z[31], 31'b0});

NET_Z: assert ((!nan && !inf && !zer && (z!=33'hffc00000)) -> fp_Z == z);
```

Figura 23. Código aserciones bloque NET

```
Z_PRUEBA_ZERO: assert ((fp_X == 32'h00000000 && fp_Y == 32'h00000000 && r_mode == 3'b001) ->
                       (fp_Z == 32'h00000000) && !udrf);
```

Figura 24. Prueba underflow erroneo

Anexo C: Assume

```
assume (r_mode inside {3'b000, 3'b001, 3'b010, 3'b011, 3'b100});
```

Figura 25. Restringir los Posibles valores de r_mode para ambos diseños

Anexo D: Repositorio de GitHub

Repositorio de GitHub utilizado para el desarrollo del proyecto:

`https://github.com/Josue096/Formal-FPU`

Anexo E: Uso de la herrmienta VCF en Verdi

Para ejecutar Verdi: con el comando `vcf -verdi -f <file.tcl>`

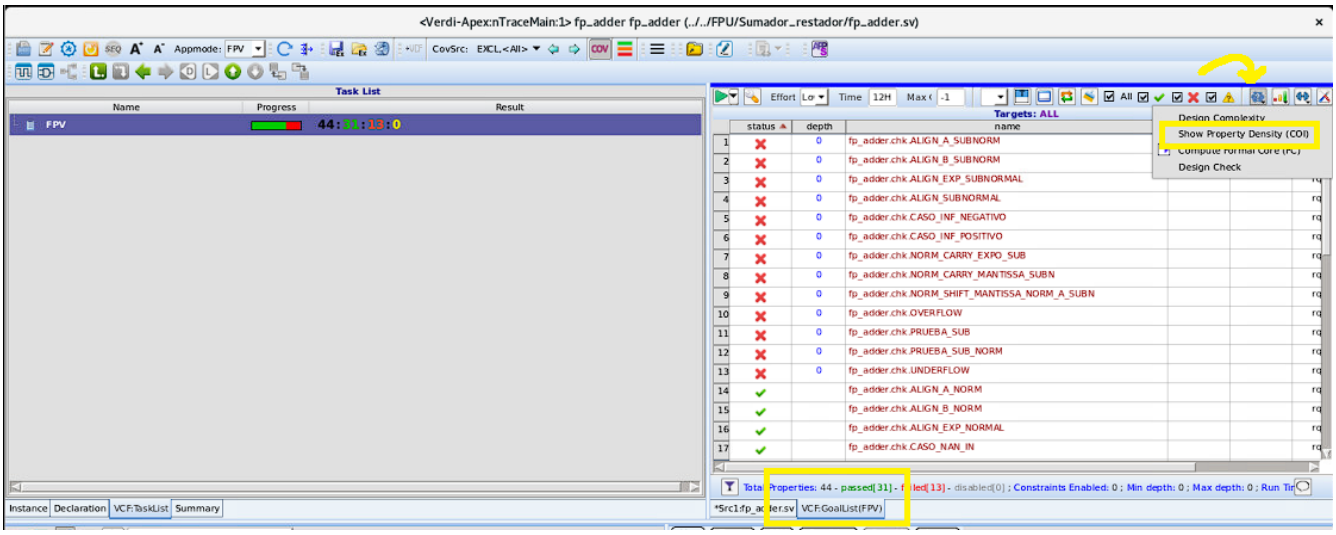

Figura 26. Como desplegar el COI dentro de verdi

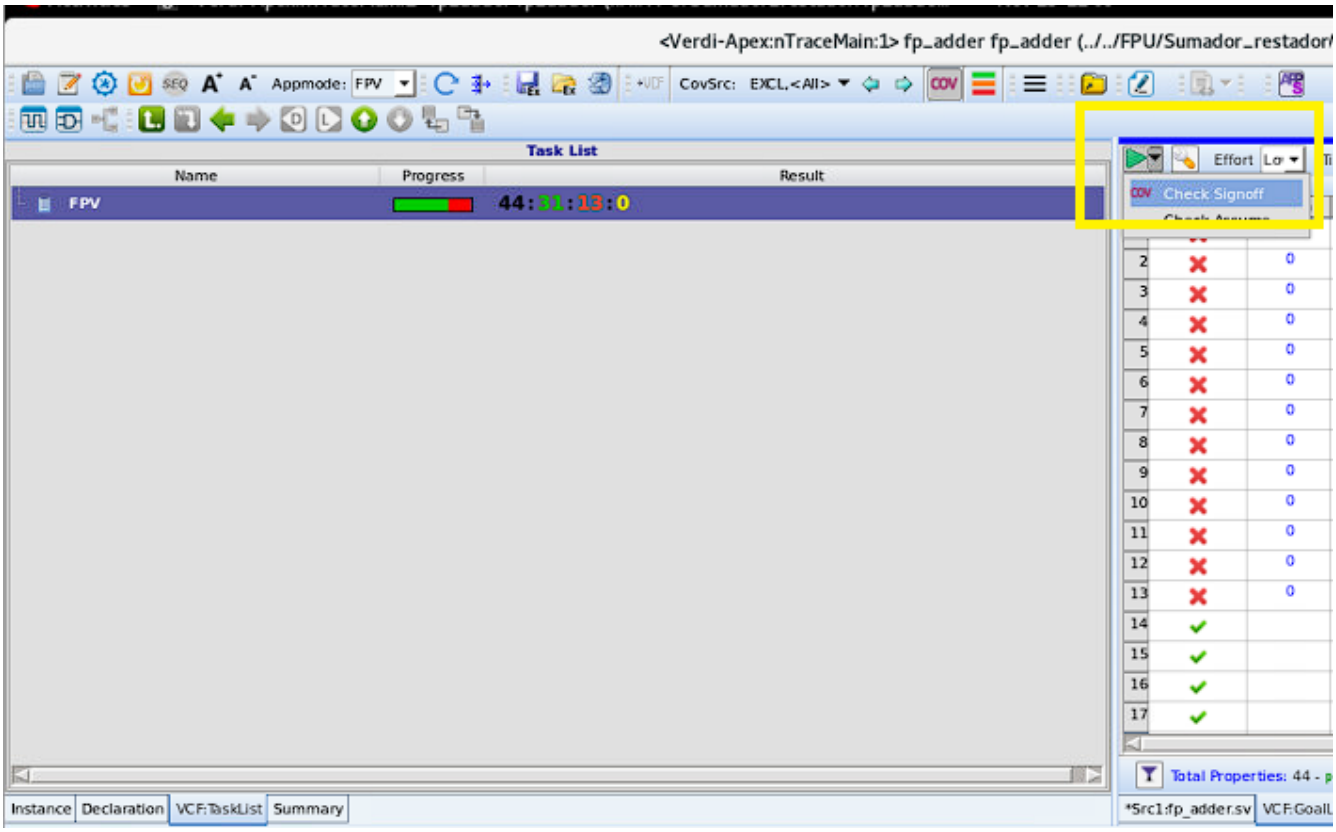

Figura 27. Como ejecutar el reporte de cobertura de la herramienta VCF (**No de las aserciones**)