UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

VERIFICACIÓN DE UN SAR ADC CON UVM EN UN ENTORNO DMS

Autor: José Juan Cerdá

Tutor: Rafael Gadea Gironés

Tutor de Empresa: José Ibáñez Climent

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Curso 2021-22

Valencia, Marzo de 2022

Dedicado a mis abuelos y mis padres

Resumen

La complejidad de los circuitos integrados aumenta con los años a causa de la gran demanada de dispositivos electrónicos cada vez más potentes en la industria. Es por ello que observamos en componentes tales como los ADC, un incremento en su funcionalidad digital integrada.

Debido a este incremento de complejidad en los diseños digitales, se han desarrollado técnicas de verificación que permiten agilizar, estandarizar y mejorar las evaluaciones de los diseños, tales como UVM y las verificaciones con modelos DMS (diseños con circuitos analógicos y digitales).

El presente trabajo trata sobre la verificación de un conversor analógico-digital de aproximaciones sucesivas (SAR ADC) a nivel de bloque en un entorno DMS. Este bloque formará parte de un chip que va a ser desarrollado por Analog Devices.

La memoria se centrará en la verificación digital con UVM del bloque pero además se mostrará información respecto a los modelos y entorno DMS.

Resum

La complexitat dels circuits integrats augmenta amb els anys a causa de la gran demanada de dispositius electrònics cada vegada més potents en la indústria. És per això que observem en components com ara els ADC, un increment en la seua funcionalitat digital integrada.

A causa d'aquest increment de complexitat en els dissenys digitals, s'han desenrotllat tècniques de verificació que permeten agilitzar, estandarditzar i millorar les avaluacions dels dissenys, com ara UVM i les verificacions amb models DMS (disenys amb circuits analògics i digitals).

El present treball tracta sobre la verificació d'un convertidor analògic-digital d'aproximacions successives (SAR ADC) a nivell de bloc en un entorn DMS. Este bloc formarà part d'un xip que serà desenrotllat per Analog Devices.

La memòria se centrarà en la verificació digital amb UVM del bloc però a més es mostrarà informació respecte als models i entorn DMS.

Abstract

The complexity of integrated circuits increases over the years due to increasing demand for more powerful electronic devices in the industry. That is why we observe in components such as ADCs, an increase in their integrated digital functionality.

Due to this increase of complexity in digital designs, verification techniques have been developed to speed up, standardize and improve design evaluations, such as UVM and DMS (designs with analog and digital circuits) model verification.

The present work deals with the verification of a SAR ADC at block level in a DMS environment. This block will be part of a chip to be developed by Analog Devices.

The report will focus on the digital verification with UVM of the block but will also contain information about the models and DMS environment.

Índice general

Índice de figuras

Índice de tablas

Listado de siglas empleadas

AC Alternating Current / Corriente Alterna.

ADC Analog to Digital Converter / Conversor Analógico-Digital.

API Application Programming Interface / Interfaz de Programación de Aplicaciones.

CDAC Capactive DAC / DAC Capacitivo.

CS Chip Select / Selección de Esclavo.

DAC Digital to Analog Converter / Conversor Digital-Analógico.

DC Direct Current / Corriente Continua.

DMS Digital Mixed-Signal / Diseño Señal-Mixta.

DNL Differential Nonlinearity / No-Linealidad Diferencial.

DUT Device Under Test / Dispositivo Bajo Prueba.

EDA Electronic Design Automation / Automatización de Diseño Electrónico.

ENOB Effective Number Of Bits / Número Efectivo de Bits.

FFT Finite Fourier Transform / Transformada de Fourier.

GTD Getting Things Done.

INL Integral Nonlinearity / No-Linealidad Integral.

IP Intellectual Property / Propiedad Intelectual.

LSB Least Significant Bit / Bit de Menor Peso.

LUT Look-Up Table / Tabla de consulta.

MDV Metric Driven Verification / Verificación basada en Métricas.

MISO Master Input - Slave Output / Entrada Maestro - Salida Esclavo.

MOSI Master Output - Slave Input / Salida Maestro - Entrada Esclavo.

MSB Most Significant Bit / Bit de Mayor Peso.

OVM Open Verification Methodology / Metodología Abierta de Verificación.

PP Post-Processing / Post-Procesado.

RAL Register Abstraction Layer / Capa de abstracción de registro.

RNM Real Number Modeling / Modelado con Números Reales.

RTL Register Transfer Level / Lógica de Resistencia-Transistor.

SAR Successive Approximation / Aproximación Sucesiva.

SCLK SPI Clock / Reloj SPI.

SFDR Spurious-Free Dynamic Range / Rango Dinámico Libre de Espurios.

SINAD Signal-to-Noise And Distortion ratio / Relación Señal a Ruido y Distorsión.

SNR Signal-Noise Ratio / Relación Señal a Ruido.

SPI Serial Peripheral Interface / Interfaz Periférica en Serie.

THD Total Harmonic Distortion / Distorsión Armónica Total.

TLM Transaction-level modeling / Modelado a Nivel de Transacción.

UVM Universal Verification Methodology / Metodología Universal de Verificación.

VHDL Hardware Description Language / Lenguage de Descripción de Hardware.

VIP Verification IP / IP de Verificación.

VPLAN Verification Plan / Plan de Verificación.

Parte I

Memoria

Capítulo 1

Introducción

1.1. Objetivo

Se requiere verificar un conversor analógico digital para integrarlo en un chip que será desarrollado por Analog Devices.

Este bloque es una IP que se ha reutilizado y editado para satisfacer las necesidades del sistema completo. El ADC nunca ha sido verificado con las disciplinas de verificación de UVM, DMS y MDV. Por lo que para mayor agilidad y verificación óptima de los requerimientos del diseño, se ha optado por una estrategia de verificación a nivel de bloque y a nivel de sistema.

Los objetivos de esta verificación serán encontrar y reportar errores del ADC, y obtener un 100 % de *code coverage* (cobertura de código) y *functional coverage* (cobertura funcional) en el bloque.

Para lograrlo, se utilizarán las herramientas proporcionadas por Analog Devices para trabajar:

- DVT Eclipse: como entorno de desarrollo
- Cadence Xcelium: como simulador
- Cadence Simvision: para depurar las formas de onda
- Jenkins: servidor para ejecutar regresiones automáticamente
- Cadence vManager: para generar el plan de verificación y analizar métricas
- MATLAB: para autogenerar código C de los modelos empleados

1.2. Metodología de trabajo del TFM

Como el bloque a verificar es parte de un chip desarrollado por Analog Devices para un cliente, se deben seguir unas plazos de entrega ajustados.

La metodología de verificación llevada a cabo se puede subdividir en diferentes tareas diferenciadas. A continuación se muestra el diagrama temporal, seguido de una breve explicación de cada tarea, ya que se entrará en más detalle a lo largo de la memoria.

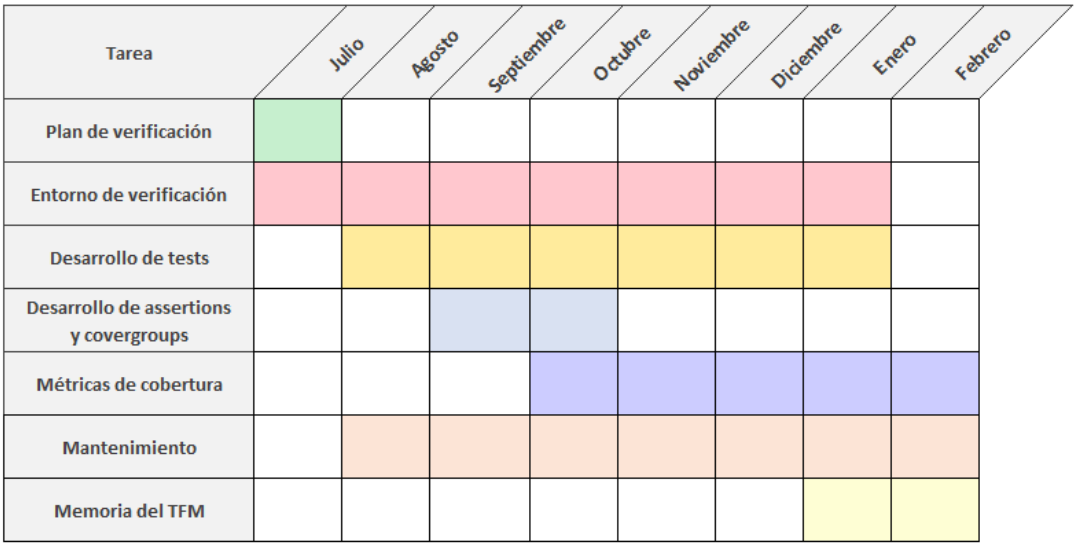


Figura 1.1: Diagrama Temporal del TFM

Plan de verificación: durante las primeras semanas del proyecto, el objetivo del verificador es familiarizarse con el bloque a verificar y preparar un plan de verificación. Para realizarlo se invierte un tiempo en leer toda la documentación sobre el bloque. Posteriormente con la herramienta vManager se diseña el VPLAN (plan de verificación) que contiene los tests, *assertions* y *covergroups* que se planean desarrollar.

Entorno de verificación: para implementar todos los elementos del VPLAN, se necesita un entorno de verificación en el que desarrollarlos. Esta tarea se extiende a lo largo del proyecto porque se realiza en paralelo a la implementación de tests, añadiendo los componentes necesarios a medida que se necesiten.

Desarrollo de tests: la finalidad de los tests es la de estimular el DUT para comprobar que funciona correctamente. A lo largo del proyecto se han tenido que implementar más que los planificados en la primera versión del plan de verificación. Esto es debido a que el equipo de diseño ha ido realizando cambios en el RTL (código del diseño a nivel funcional) del ADC para añadir nuevas funcionalidades y adaptar la IP para las necesidades del proyecto. Otra razón es la necesidad de desarrollar nuevos tests que ayuden a obtener un 100 % de métricas de cobertura.

Desarrollo de *assertions* y *covergroups*: se ejecutan en paralelo a los tests y complementan la verificación ya que forman parte de las métricas funcionales de cobertura.

Métricas de cobertura: se han ido analizando las métricas de cobertura des de Octubre con vManager. Esto nos ha permitido tener un seguimiento de como avanza la verificación, el impacto que tiene y como ir mejorándola para obtener un 100 % en la cobertura de código y funcional.

Mantenimiento: a medida que avanza el proyecto se requiere un continuo mantenimiento de los elementos ya implementados, tales como el entorno de verificación, los tests, los modelos... Teniendo así que estar constantemente editándolos, mejorándolos y revisándolos para cubrir todos los escenarios y arreglar posibles errores que surjan.

Memoria del TFM: esta memoria se ha escrito y revisado entre Enero y Febrero.

Además, para gestionar las subtareas del trabajo y redacción de la memoria he empleado la metodología GTD (Getting Things Done).

GTD se basa en una serie de hábitos productivos que permiten al usuario gestionar sus tareas y proyectos de una forma eficaz, productiva y sin estrés si se aplica correctamente.

Capítulo 2

Metodología de verificación

En este capítulo se presentan las disciplinas que forman parte de la verificación digital. Además de proporcionar información sobre los estándares, se muestran las diferentes técnicas y metodologías que se han utilizado en la verificación del bloque.

2.1. SystemVerilog

SystemVerilog es el lenguaje utilizado en este proyecto debido a su importancia en la industria para descripción de hardware y verificación de sistemas electrónicos. Además, forma parte del estándar IEEE 1800 des de 2005, siendo su última versión a esta fecha el IEEE 1800-2017 [1].

Este lenguaje está basado en el también estandarizado como IEEE 1364 Verilog. Verilog es un lenguaje de descripción de hardware utilizado para modelar circuitos electrónicos creado en 1983 y adquirido en 1990 por Cadence. Con los años se fue refinando y mejorando hasta que en 1995 pasó a ser un estándar debido a la gran popularidad que fue tomando VHDL [2].

A medida que el diseño de circuitos integrados se vuelve más complejo, el esfuerzo de verificación se vuelve aún más preponderante, requiriendo así mejores herramientas para optimizar su implementación y obtener un buen grado de cobertura de los diseños. Es de ahí donde surge SystemVerilog, un lenguaje que añade muchas nuevas funcionalidades y herramientas tanto de diseño como para crear entornos de verificación más complejos y robustos.

En la siguiente imagen, se muestra un resumen de las funcionalidades que se han ido añadiendo desde Verilog hasta la última versión de SystemVerilog:

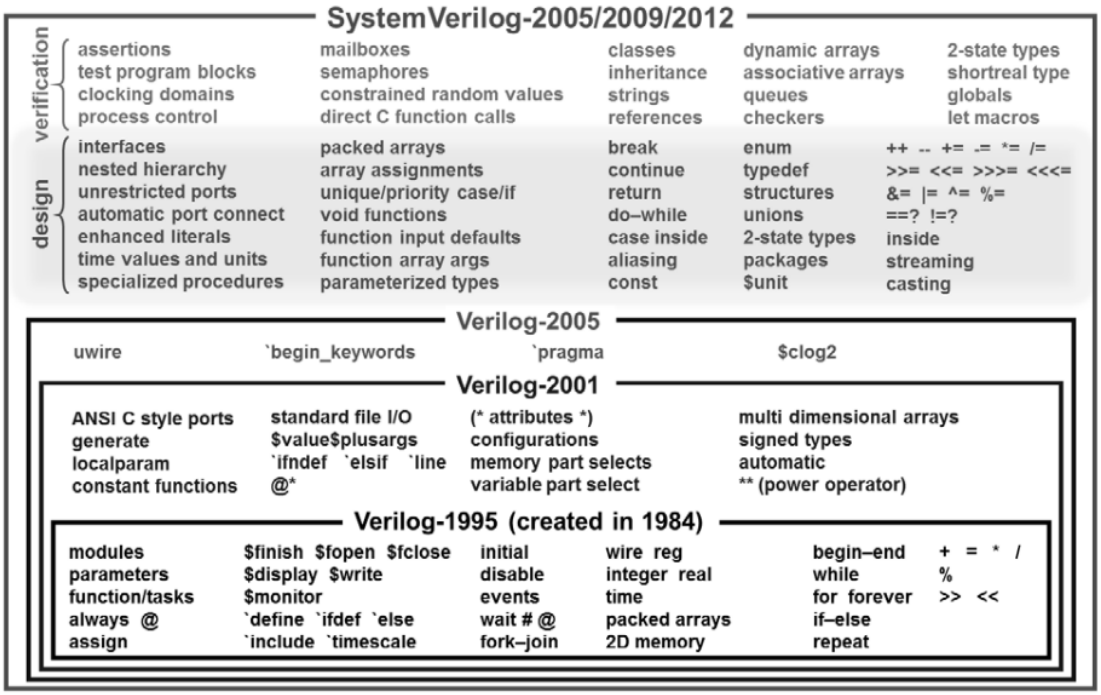


Figura 2.1: Evolución de funcionalidades temporal del estándar Verilog - SystemVerilog

Una funcionalidad esencial que también aporta SystemVerilog a la verificación es la pseudo-randomización. Esto es que las variables randomizables que sean usadas en un test tendrán siempre el mismo valor para una misma semilla. Esta semilla por defecto es randomizada pero puede ser especificada al ejecutar un test. Esto nos da una idea de porqué es tan beneficioso ya que por ejemplo si un test fallara y quisiéramos reproducirlo exactamente igual, solamente tendríamos que volver a ejecutarlo con la misma semilla con la que había fallado.

Dos piezas clave que añade SystemVerilog son las *assertions* y *covergroups* que se comentarán en más detalle en la próxima sección.

2.2. Cobertura Funcional y de Código

La cobertura a grandes rasgos es el porcentaje de objetivos de verificación que se han logrado. Básicamente se trata de una serie de métricas que nos ayudan a determinar cuanto el DUT ha sido ejercitado, logrando así tener un seguimiento continuo de cómo avanza la verificación y tener unas metas claras de cuando se puede considerar que el bloque está completamente verificado.

La cobertura está formada por la cobertura funcional y la de código:

- Cobertura de Código: se extrae automáticamente del código del diseño y proporciona información de él, como son:
  - Líneas de código que han sido alcanzadas
  - Lógica ejecutada

  - Transiciones de los bits de las señales del diseño (de 0 a 1, y de 1 a 0)
  - Que se hayan recorrido todos los caminos posibles de una máquina de estados

- Cobertura Funcional: es definida por el ingeniero y complementa la cobertura de código ya que se basa en las especificaciones del diseño. Las *assertions* y *covergroups* forman parte de ella.

Los *covergroups* están formados por una serie de puntos a cubrir que además se pueden cruzar para cubrir variables entrelazadas o correlacionadas.

La mejor forma de explicarlo sería con un ejemplo: Queremos asegurarnos de que se ejecuten todas las direcciones posibles en una transacción de SPI tanto de lectura como de escritura en los tests de SPI. Por lo tanto tendríamos que crear un *covergroup* que se muestree en los tests de SPI y que tenga la siguiente forma:

```
1  covergroup adc_spi_wr_addr_cg;
2
3    cp_addr : coverpoint addr {
4      bins REG[ADDR_MAX-ADDR_MIN+1] = { [ADDR_MIN:ADDR_MAX] };
5    }
6
7    cp_wr : coverpoint write_read {
8      bins write = { 1'b0 };
9      bins read  = { 1'b1 };
10   }
11
12   xc_wr_addr : cross cp_addr, cp_wr;
13
14 endgroup : adc_spi_wr_addr_cg
```

Figura 2.2: Ejemplo: Covergroup

- **cp_addr:** puntos de dirección, habrán un total de N = dirección_máxima - dirección_mínima
- **cp_wr:** puntos de tipo de transacción, 2 tipos (escritura o lectura)
- **xc_wr_addr:** puntos cruzados entre dirección y tipo de transacción, habrán 2*N puntos

Esta información nos serviría luego para comprobar si en los tests de SPI hemos ejercitado el DUT con todas las direcciones posibles tanto con transacciones de lectura como de escritura.

Por otro lado están las *assertions*, que nos ayudan a verificar los diferentes comportamientos del diseño. Esta a su vez está formada por dos tipos, las *assertions* inmediatas y las concurrentes.

Las *assertions* inmediatas son parecidas a un condicional if y están siempre activas. Sin embargo en la verificación del bloque no se ha usado ninguna *assertion* inmediata.

Las *assertions* concurrentes nos permiten desarrollar comprobadores más complejos ya que nos proporcionan una selección de funciones y operadores para implementarlas. Su estructura se muestra a continuación:

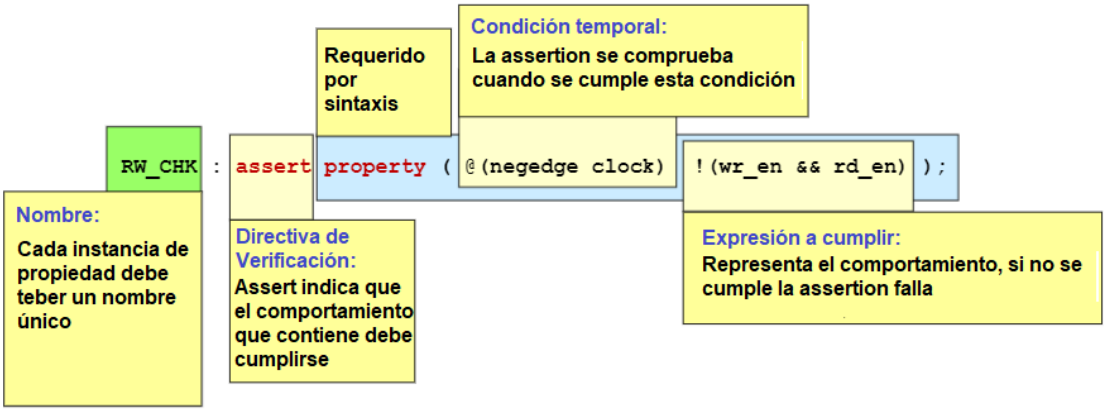


Figura 2.3: Estructura *assertion* concurrente [3]

Esta *assertion* comprobaría que nunca sean wr_en y rd_en igual a 1 cuando hay un flanco negativo de reloj.

Además, se les puede añadir condiciones para desactivarlas con disable_iff (por ejemplo mientras reset está activo) y mensajes de error que nos ayuden a entender qué comprueban.

Un ejemplo que utilice todas las características y funcionalidades comentadas sería el siguiente:

```
1   property var_num_cycles_chk;
2     @(posedge clk) disable iff(rstb !== 1 || disable_checks === 1)
3     var |-> ##[29:$] !var;
4   endproperty
5   assert property (var_num_cycles_chk)
6     else $error("FALLO: VAR DEBE ESTAR ACTIVO DURANTE AL MENOS 29 CICLOS")
7
```

Figura 2.4: Ejemplo: *assertion* concurrente

En esta assertion:

- Se desactiva si rstb es diferente de 1 y/o disable_checks es igual a 1
- Si está activa se ejecuta en cada flanco de subida de clk
- Si se observa que la variable “var” está a 1, debe mantenerse así por lo menos 29 ciclos
- Si no se cumple se mostrará el error con mensaje ”FALLO: VAR DEBE ESTAR ACTIVO DURANTE AL MENOS 29 CICLOS”

2.3. UVM

UVM es una metodología de verificación de circuitos estandarizada como IEEE 1800. Está basada en OVM y está respaldada por empresas EDA tales como Cadence, Synopsis, Mentor y Aldec.

UVM nace de la necesidad de desarrollar entornos de verificación con componentes que sean modulares, reusables y escalables. Por lo tanto es un estándar que ayuda a la industria electrónica a que cada empresa se pueda montar sus propias IP's de verificación (VIP) para reutilizarlas entre proyectos, reduciendo así costes de desarrollo en el proceso. Al ser un estándar también habilita el uso de VIP's de terceros, por lo que aparecen modelos de negocio donde las empresas las desarrollan para venderlas.

Para lograrlo UVM aporta un set de APIs que definen una librería de clases basada en el estándar de SystemVerilog [4] [5].

A lo largo de la memoria se hará referencia a los componentes que forman un banco de pruebas con UVM. Es por ello que a continuación, se muestra un ejemplo de la estructura que propone UVM y debajo la explicación de cada componente mostrado en la imagen:

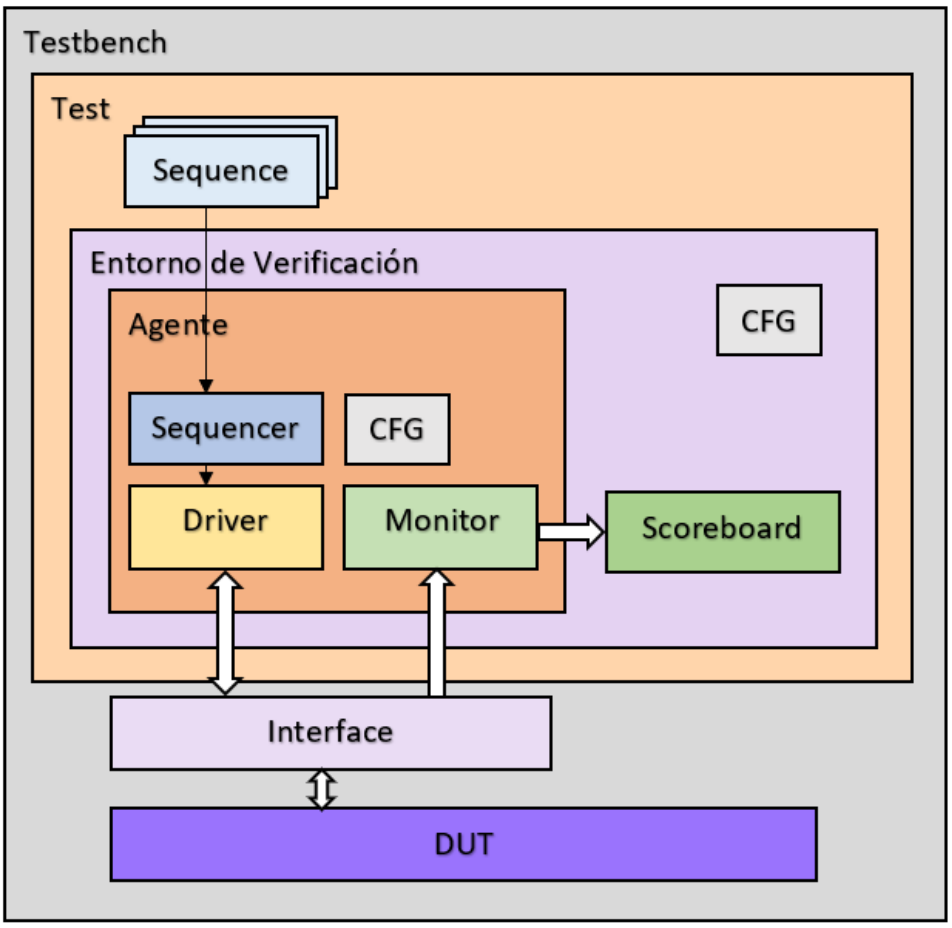


Figura 2.5: Ejemplo: Banco de pruebas con UVM

- **Test:** cada test implementado instancia el entorno de verificación y controla los estímulos que son enviados como secuencias.
- **Entorno de verificación:** instancia y configura los componentes UVM. Además se añaden variables a su propia configuración. Ejemplos de variables interesantes podrían ser: para deshabilitar las *assertions*, cambiar el tiempo de carrera de los tests, deshabilitar los predictores del scoreboard...
- **Agente:** este componente de UVM se comunica con las interfaces conectadas al DUT y encapsula al driver, el monitor y el sequencer. Las variables de configuración que tenga variarán dependiendo de la VIP que se use. Si un agente es activo significa que puede escuchar y enviar transacciones al DUT por lo que tendrá sequencer, driver y monitor. En cambio, si es pasivo solo puede escuchar, teniendo entonces solamente el monitor.
- **Sequencer:** se encarga de controlar la generación de estímulos, apuntando a las diferentes secuencias que hay y enviándoselas al driver conectándose vía interfaz TLM.
- **Driver:** recibe las secuencias del sequencer y forma transacciones a partir de ellas para transmitirlas a las interfaces del DUT conectándose vía interfaz virtual. Además, le indica al sequencer cuando ha terminado con una secuencia para que le envíe más.
- **Monitor:** es un componente pasivo ya que simplemente escucha las transacciones en la interfaz del DUT. Además, también se encarga de generar paquetes de secuencias a partir de las transacciones monitorizadas y transmitirlas al scoreboard.
- **Interface:** las interfaces se conectan directamente al DUT para enviarle señales o monitorizarlas. Es por ello que son un buen lugar para añadir elementos tales como *assertions* o *covergroups*.
- **Scoreboard:** recibe los paquetes de secuencias de los diferentes monitores del sistema y contiene los predictores y modelos ideales. Por lo tanto se encarga de asegurar que para unos estímulos de entrada determinados que tenga el DUT, que su resultado de salida sea igual al esperado.

2.4. DMS

En la industria de los semiconductores cada vez son más los chips que están formados por IP's analógicas y digitales que se comunican entre sí, formando así entornos Mixtos-Señal.

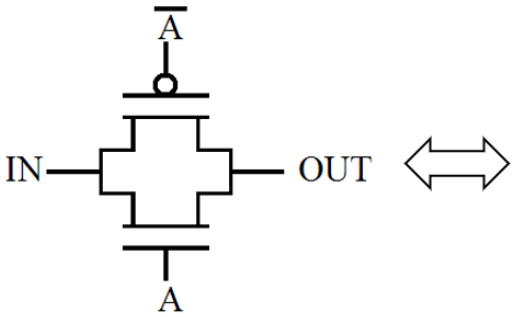
Las simulaciones en entornos analógicos son muy lentas a comparación con las digitales, pudiendo llegar a días de simulación para tests que puedan durar algunos milisegundos. Por lo tanto, estas simulaciones son inviables para cumplir plazos de entrega ajustados de verificación de circuitos digitales que interactúen con lo analógico.

La disciplina de DMS soluciona este problema con la implementación de modelos que imitan el comportamiento analógico, llamados RNM (Real Number Modeling, Modelado con Números Reales en español). Esto permite que podamos realizar simulaciones con el simulador digital, incrementando así la velocidad en que se ejecutan los tests y por lo tanto agilizar la verificación

digital de un sistema y/o bloque. Además, también nos permiten validar la conectividad analógica rápidamente en el proyecto y facilita la reutilización de tests para las simulaciones con circuitos analógicos.

El flujo de DMS consiste en añadir a las vistas analógicas del esquemático un código RNM para que posteriormente se cree una netlist de la jerarquía y se conecte con el resto de bloques digitales.

Para modelar un circuito analógico con RNM se pueden utilizar expresiones matemáticas, lógicas, aproximaciones o look-up tables (LUT, tabla de consulta en español). Un ejemplo sencillo podría ser la implementación de una puerta de transmisión con transistores P y N:




```
1 module TGATE(in, a, out);
2
3   input wrealsum in;
4   input a;
5   output wrealsum out;
6
7   assign out = a ? in : `wrealZState;
8
9 endmodule
```

Figura 2.6: Ejemplo: Puerta de transmisión analógica implementada con RNM

Sin embargo, estos modelos tienen que ser verificados y requieren de un mantenimiento a lo largo del proyecto. Dos formas de verificar un modelo podrían ser:

- Comparando el comportamiento del modelo con las especificaciones del diseño (Figura 2.7)
- Comparando para unos mismos estímulos de entrada el comportamiento del modelo con el del circuito analógico a nivel de transistor (Figura 2.8)

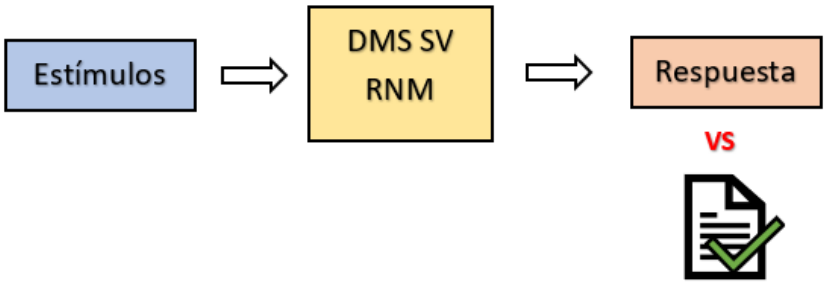


Figura 2.7: Verificación RNM vs Spec

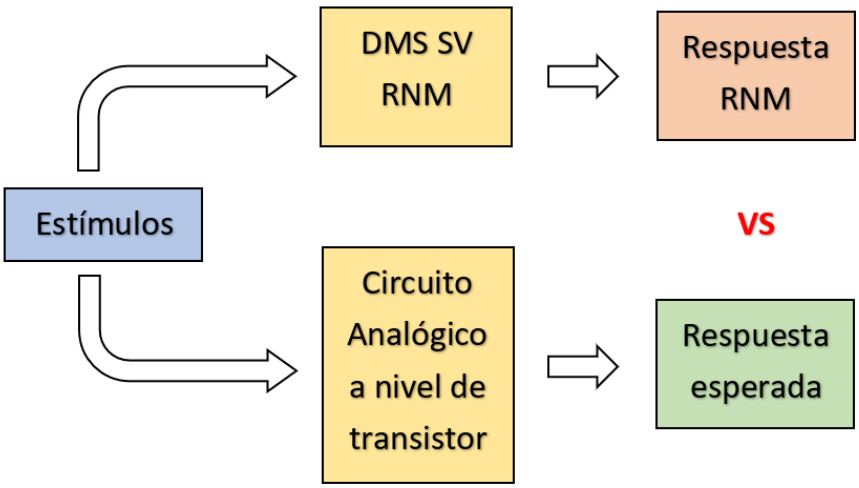


Figura 2.8: Verificación RNM vs Circuito analógico a nivel de transistor

Otro tema a tener en cuenta es que esta disciplina no sustituye enteramente a las simulaciones analógicas. En la verificación seguirán siendo necesarias tener simulaciones más precisas y tests específicos que ejerciten características como las alimentaciones o rendimiento por ejemplo.

Capítulo 3

Descripción del SAR ADC

Este capítulo se centrará en el ADC y contiene dos secciones diferenciadas.

La primera sección contiene la base teórica de un ADC para entender la terminología que aparecerá a lo largo de la memoria. En las subsecciones se expone de forma teórica qué es un ADC, los fundamentos que lo componen y las arquitecturas más utilizadas para implementarlos, centrándose en la de aproximaciones sucesivas (SAR). Cabe destacar que la información que contiene esta sección es la misma que realicé en mi TFG "Diseño de un SAR ADC para aplicaciones multicanal con tecnología CMOS de 180nm" [6] debido a la relación entre ambos trabajos.

La segunda sección muestra el SAR ADC a verificar. Al tratarse de una IP de Analog Devices se mostrará, sin entrar en mucho detalle, la información necesaria para entender su verificación, como son sus funcionalidades y el diagrama de bloques del circuito.

3.1. Base teórica

3.1.1. ¿Qué es un ADC?

Como indica su nombre, la función de un ADC es la de transformar señales analógicas en digitales. Esta conversión facilita el procesado de datos y la señal resultante se vuelve más inmune al ruido y a otras interferencias en las que las señales analógicas son más sensibles. Por lo tanto, como tienen un rol importante en el procesamiento de señales, son altamente usados en campos tales como audio, control, comunicación y medicina [6].

A grandes rasgos, el proceso de conversión que sigue un ADC tiene tres partes diferenciadas:

- Muestreo: medir la amplitud de una señal de entrada (normalmente voltaje) de forma periódica
- Cuantificación: redondear la muestra sampleada y procesada a un valor preestablecido de los niveles de cuantificación
- Codificado: se codifica el valor obtenido para registrarlo digitalmente

En la siguiente imagen se muestra el proceso comentado:

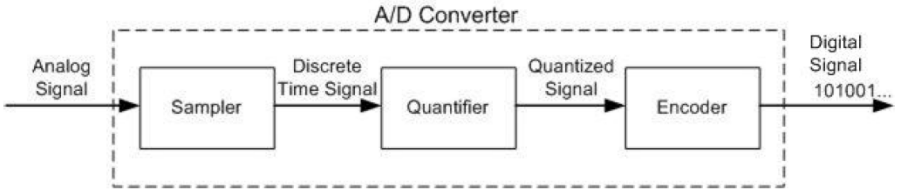


Figura 3.1: Proceso de Conversión de un ADC [7]

3.1.2. Fundamentos de un ADC

Resolución y LSB: La resolución es el número de bits o salidas de un ADC. Siendo el LSB el paso más pequeño que se puede hacer. La resolución será el que limite en mayor o menor medida la cantidad de niveles de cuantificación que pueda tener el ADC, siendo este equivalente a 2^{bits}. El LSB se calcula con el fondo de escala (normalmente será el voltaje de referencia) y los niveles de cuantificación $V_{LSB} = \frac{V_{REF}}{2^{bits}}$.

Error de cuantificación: es el error que aparece al convertir una señal analógica a unos valores preestablecidos (niveles de cuantificación) finitos. Se calcula sacando la diferencia entre la señal que obtenemos a la salida del ADC y la que se debería obtener en un conversor analógico digital ideal. La figura 3.2 muestra la curva de transferencia de un ADC ideal de 3 bits centrada en cero y su correspondiente error de cuantificación.

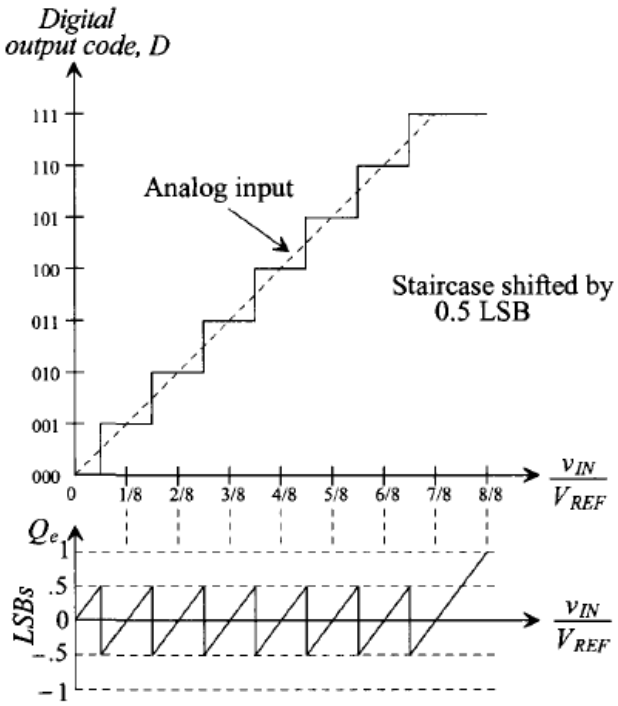


Figura 3.2: Error de cuantificación[8]

Características estáticas o DC: nos permiten analizar cuánto se desvía la curva de transferencia de la del ADC ideal. De esta forma se puede observar si hay errores de ganancia, offset o códigos perdidos (códigos que no aparecen en la curva). Dos métricas que utilizaremos en la verificación del ADC para ver si estamos cumpliendo las especificaciones son:

- **DNL:** la No-Linealidad Diferencial calcula cuanto se desvían los anchos de los códigos de nuestro ADC respecto al esperado (1LSB ya que es el mínimo paso del ADC). Siendo su ecuación: $DNL = Ancho_{ADCreal} - 1LSB$. Por lo que si el ancho es más pequeño que 1LSB tendremos un DNL negativo y positivo a la inversa.
- **INL:** la No-Linealidad Integral nos indica la desviación de la curva con la ideal y se puede obtener como el sumatorio de DNLs.

Por ejemplo en la imagen 3.3 en el código 001 tendríamos un DNL de -0.5LSB y en el 101 de +0.5LSB. Esto se traduciría en el INL en que a partir del 001 tengamos un valor de -0.5LSB y que vuelva a 0LSB en 101.

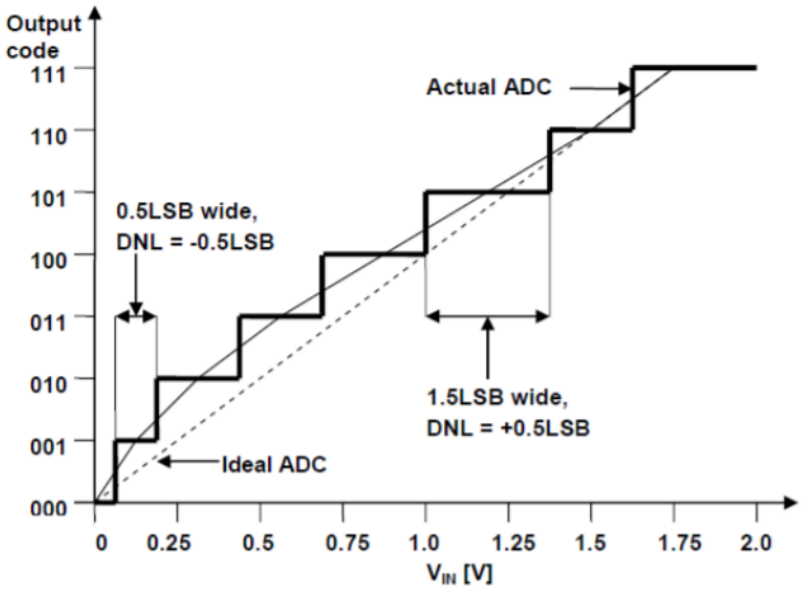


Figura 3.3: Ejemplo: desviación de una curva de un ADC real[8]

Características dinámicas o AC: analizan como se comporta el ADC en el dominio de la frecuencia, observando la relación que hay entre señal y rudio y/o distorsión armónica. Estas métricas se obtienen a través de la FFT de la señal de salida de un ADC después de introducirle a la entrada una senoidal.

Por ejemplo para una señal de salida de un ADC real:

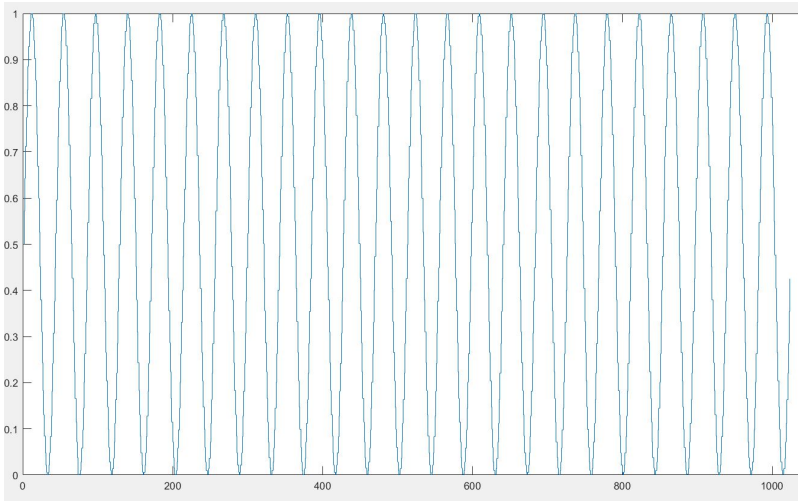


Figura 3.4: Señal de salida de un ADC real [6]

Si sacamos su FFT:

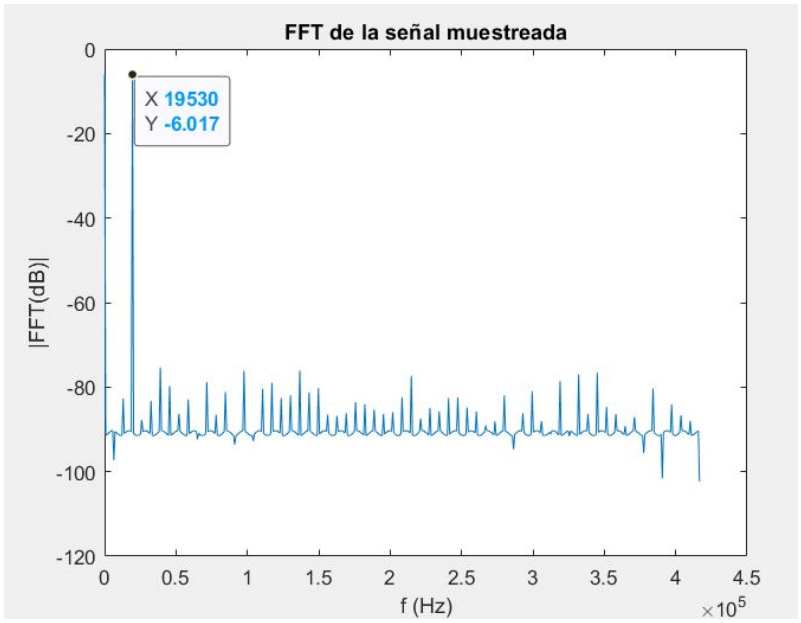


Figura 3.5: FFT de un ADC real [6]

Podríamos obtener las siguientes métricas dinámicas:

- **SNR:** la relación señal a ruido representa la proporción entre la señal que se transmite y el ruido que la corrompe:

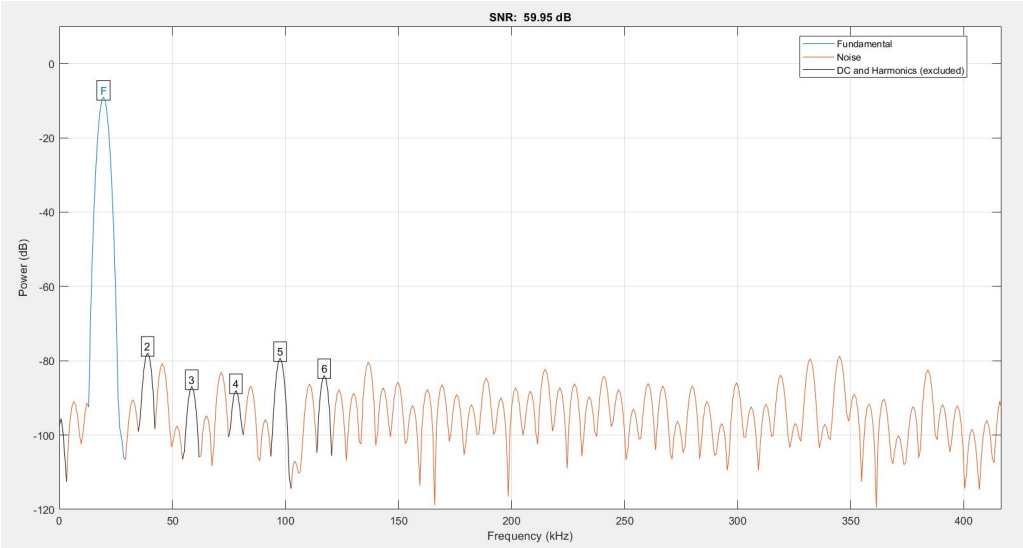


Figura 3.6: SNR de un ADC real [6]

- **THD:** la distorsión armónica mide la relación entre la suma de las potencias de los armónicos y la fundamental de la señal:

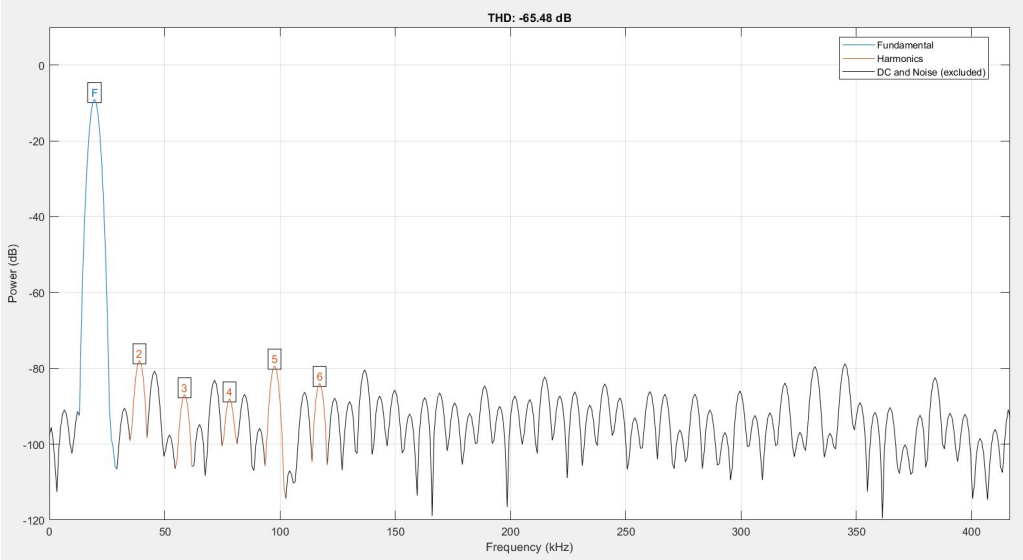


Figura 3.7: THD de un ADC real [6]

- **SFDR:** mide la relación entre la señal fundamental y el espurio de mayor amplitud:

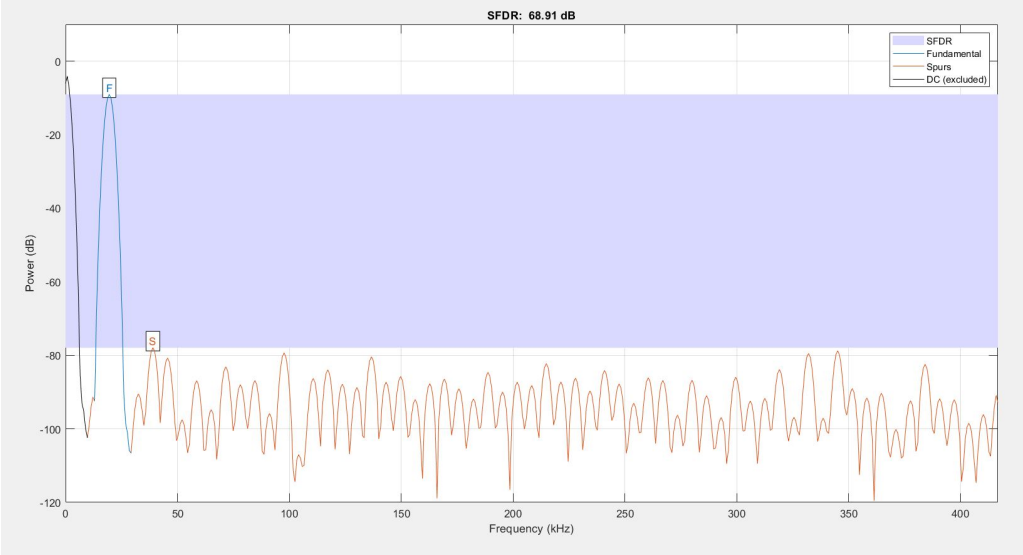


Figura 3.8: SFDR de un ADC real [6]

- **SINAD:** la relación señal a ruido y distorsión representa la proporción entre la señal fundamental frente al ruido y la distorsión armónica.

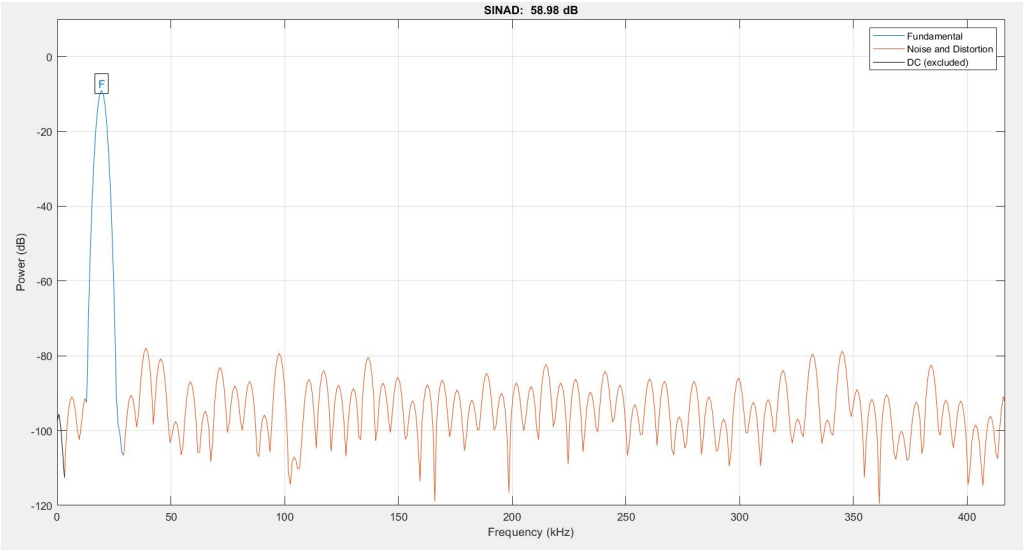


Figura 3.9: SINAD de un ADC real [6]

- **ENOB:** el número efectivo de bits nos indica cuan cercano nuestro ADC está de funcionar como uno ideal. Se obtiene con la fórmula $ENOB = \frac{SINAD - 1{,}76}{6{,}02} = 9{,}505$.

3.1.3. Arquitecturas ADC

Un ADC se puede diseñar con diferentes arquitecturas dependiendo de la aplicación para la que se vaya a utilizar. La decisión de cuál usar vendrá determinada de las características requeridas, siendo estas principalmente la velocidad de muestreo del ADC, su resolución y su consumo.

A continuación, se muestra una tabla comparativa entre las cinco arquitecturas más utilizadas en el mercado:

Arquitectura	Velocidad (muestras/s)	Resolución (bits)	Consumo	Aplicaciones
Flash	10M - 10G	4 - 8	Alto	Comunicaciones, Radar Osciloscopios digitales
Pipelined	5 - 200M	8 - 16	Medio	Captura de vídeo y audio Aplicaciones instrumentación
SAR	1 - 20M	8 - 18	Bajo - Medio	Adquisición de datos Aplicaciones de bajo coste
Rampa	2 - 100k	12 - 24	Bajo	Transductores de baja frecuencia (presión, temperatura...)
Sigma-Delta	10 - 100k	16 - 24	Medio	Audio digital Medición industrial

Tabla 3.1: Tabla Comparativa entre arquitecturas de ADC[9][10]

3.1.4. SAR ADC

Como se observa en la tabla 3.1, un ADC de tipo SAR o aproximaciones sucesivas proporciona una buena compensación entre velocidad y resolución, y se pueden obtener consumos bajos.

El SAR ADC utiliza un algoritmo de búsqueda binaria para realizar la conversión. El algoritmo realiza comparaciones y aproximaciones durante Nbits ciclos de reloj para aproximar el valor de entrada al más cercano de los preestablecidos que tiene.

La figura 3.10 muestra un ejemplo de cómo funciona.

Términos que aparecen en la imagen:

- V_{IN}: voltaje de entrada a convertir
- V_{REF}: voltaje de referencia que conocemos su valor (suele usarse para alimentar el ADC)
- V_{DAC}: voltaje que se va fijando a medida que el algoritmo toma decisiones

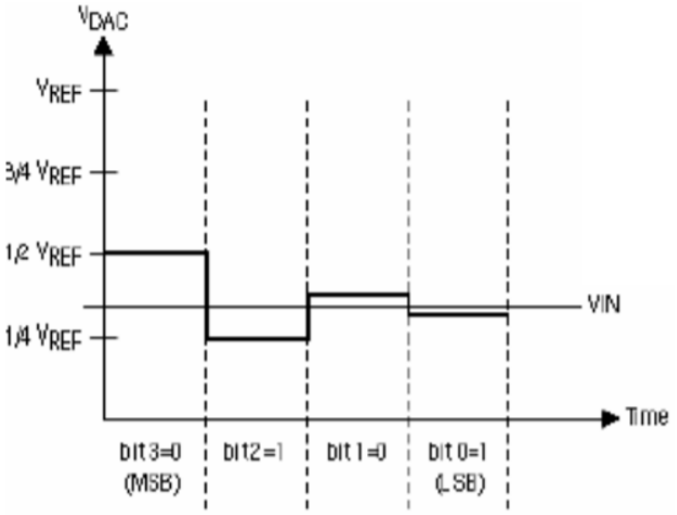


Figura 3.10: Ejemplo de conversión en un SAR de 4 bits [9]

Se observa como en 4 ciclos poco a poco el voltaje calculado (V_{DAC}) se acerca más al de entrada (V_{IN}) y se consigue el código digital que lo representa (en este caso V_{IN} equivale a 0101).

Para conseguirlo el algoritmo:

- Fija un primer voltaje de prueba que vale $\frac{V_{REF}}{2}$
- Comprueba si V_{DAC} es mayor o menor que V_{IN}
  - Si es mayor se guarda en ese bit un 0 y el siguiente V_{DAC} a comprobar será menor
  - Si es menor se guarda en ese bit un 1 y el siguiente V_{DAC} a comprobar será mayor
- A medida que avance el algoritmo los incrementos de voltaje entre ciclos serán la mitad de grandes respecto al anterior
- Y así hasta obtener el código final

Para conseguir ejecutar este algoritmo el SAR ADC debe estar formado por:

- Un **DAC** que transforme los voltajes de referencia a comparar con la entrada en base a las decisiones que se tomen en el algoritmo
- Un **comparador** rápido para comparar V_{DAC} y V_{IN}
- Un bloque digital que contenga el algoritmo **SAR**
- Un circuito de sample & hold que muestree la señal de entrada que se quiere digitalizar (aunque puede ir incluido en el circuito del DAC)

La imagen 3.11 muestra un posible diagrama de bloques de un SAR ADC:

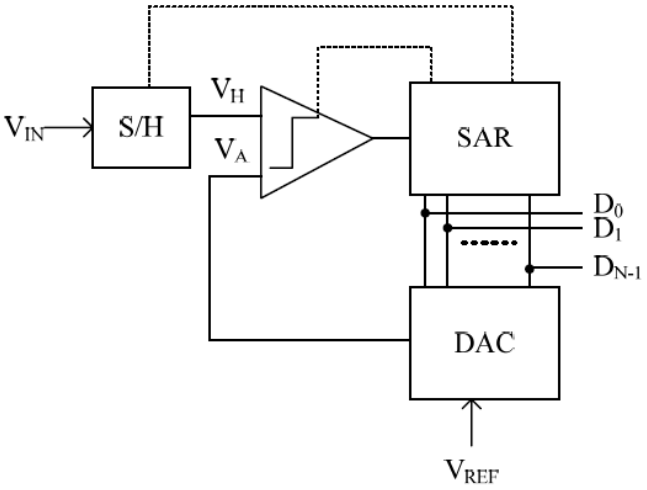


Figura 3.11: Diagrama de bloques de un ADC [11]

3.2. Introducción al bloque a verificar

A continuación se muestra el diagrama de bloques con las características y funcionalidades más importantes para la verificación digital del ADC:

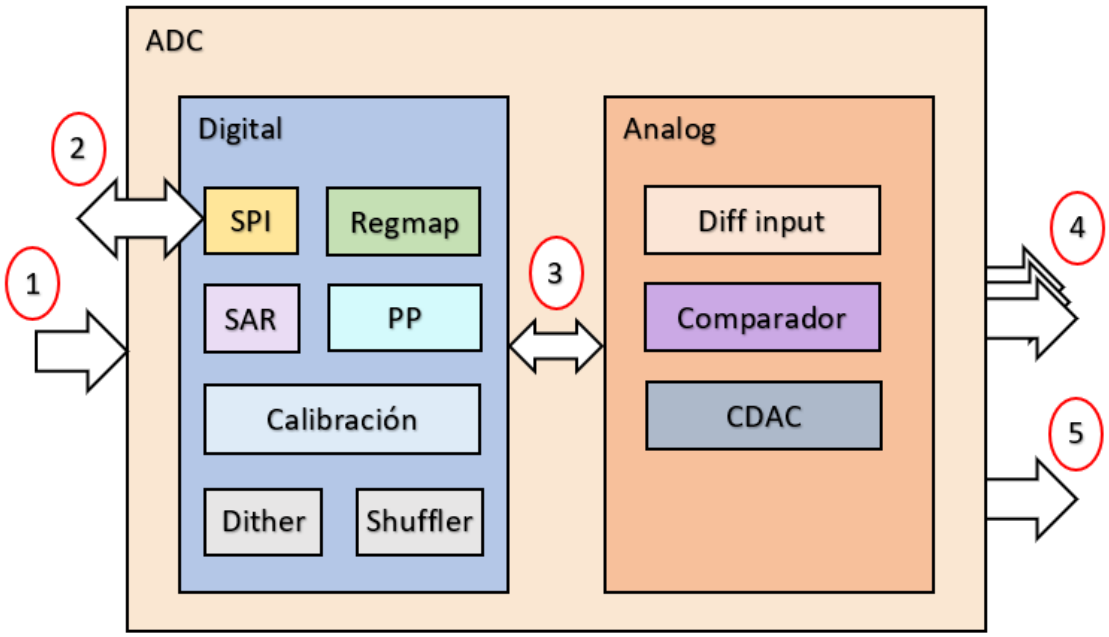


Figura 3.12: Diagrama de bloques del SAR ADC a verificar

De la parte **analógica** nos interesa saber que el DAC está implementado con condensadores (CDAC) y que la entrada es de tipo diferencial.

La parte **digital** se divide en 7 partes:

SAR: contiene toda la lógica del algoritmo SAR y se encarga de controlar la interacción entre señales digitales y analógicas durante los procesos de conversión.

Calibración: el ADC tiene integrado un algoritmo de calibración para compensar errores en los valores de los condensadores en el DAC, elementos parasíticos o inyección de carga. Este algoritmo calcula unos coeficientes que son posteriormente guardados en el mapa de registros.

Regmap: es el mapa de registros del ADC y puede ser accedido por el usuario vía SPI. A continuación se muestran los registros divididos por funcionalidad y algunos ejemplos de campos que contienen:

- Coeficientes de calibración
- Registros de configuración:
  - Conversión:
    - Habilitar/Deshabilitar el Dither y/o Shuffler
    - Añadir un offset al resultado
    - Multiplicar por una ganancia el resultado
  - Calibración:
    - Cuantos bits calibrar <=>cuantos coeficientes calcular
    - Elegir entre un promedio de 1024 o 4098 muestras para cada coeficientes
    - Habilitar/Deshabilitar calibración de ganancia
    - Habilitar/Deshabilitar calibración de offset
  - Para la parte analógica
- Registros para depurar (sólo de lectura):
  - Resultado del ADC
  - Valor aplicado de dithering
  - Valor aplicado de shuffling

SPI: es un estándar de comunicaciones usado para transmitir información entre circuitos electrónicos. Para que haya una comunicación SPI debe haber un maestro (el que envía las transacciones) y un esclavo (el que las recibe).

Las 4 señales que componen el bus SPI son las siguientes (figura 3.13):

- **CS:** es la única señal que no se comparte entre todos los esclavos que hayan. Se activa en el esclavo en el que se esté dando una comunicación con transacciones SPI.
- **MISO:** línea de datos que recibe el maestro del esclavo
- **MOSI:** línea de datos que envía el maestro al esclavo

- **SCLK:** es el reloj SPI

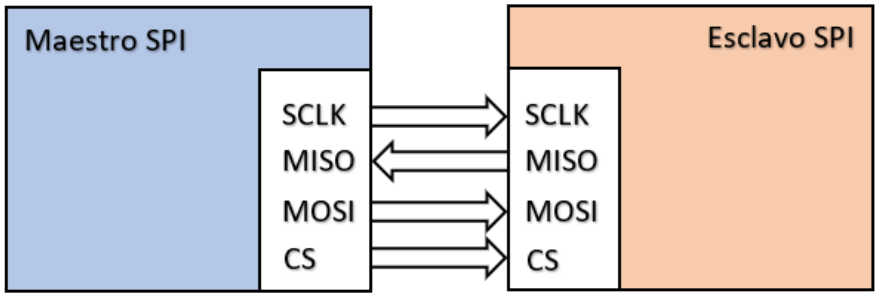


Figura 3.13: Bus SPI

Las transacciones de SPI se pueden configurar como:

- Lectura o Escritura: para leer o sobre-escribir un registro del regmap
- Single o Burst:
  - Single: se realiza una sola operación (lectura o escritura) sobre un registro
  - Burst: se realiza una operación sobre varios registros. En burst se indica la dirección sobre la que se parte y a medida que avanza se auto-incrementa. De este modo se pueden leer o escribir registros de forma continua sin tener que indicar la dirección ni enviar una transacción para cada uno.

PP: al finalizar la conversión se aplica un post-procesado al resultado para aplicar la corrección de coeficientes, el dither, el offset y ganancia.

Dither: el dithering es una técnica de inyección de ruido en las conversiones para combatir posibles correlaciones que aparezcan entre el ruido de cuantificación y la señal de entrada [12].

Shuffler: si un condensador tiene un error en su valor y se utiliza siempre en el mismo momento del proceso de conversión para generar V_{DAC}, obtendremos un ruido correlado entre el condensador y el mismo bit. La técnica de shuffling se encarga de decorrelarlos barajando qué condensadores se usan para los MSB.

Finalmente faltaría comentar las diferentes interfaces que hay en el ADC, representadas por números en la figura 3.12

- **1:** Señales de entrada del ADC
  - Conv_start: para iniciar una conversión
  - Cal_start: para iniciar la calibración
  - Diff_input: entrada diferencial
  - Reset
  - Señales de alimentación

- **2:** Señales del bus SPI (3.13)
- **3:** Interfaz Digital - Analógica
- **4:** Interfaz paralela donde se registra el resultado de conversión
- **5:** Señales de salida del ADC
    - Busy: activa durante el proceso de conversión
    - Eoc: indica el final de conversión y que el resultado está disponible
    - Eocal: indica el final de calibración y que los coeficientes están disponibles en el reg-map

Capítulo 4

Verificación del bloque

Este capítulo se centra en la verificación del ADC, empezando por mostrar la disciplina de verificación y las diferentes fases que la componen. Posteriormente se hace énfasis en cada fase y en las estrategias que se han utilizado para reducir al máximo el tiempo de implementación y verificación del bloque.

4.1. Disciplina de verificación

El objetivo de la verificación es la de asegurar que un diseño funciona correctamente para todos sus modos y funcionalidades. Todo esto con el propósito de encontrar y solucionar todos los errores posibles antes de enviarlos a fabricar a silicio, donde depurarlos y arreglarlos se vuelve más costoso.

Para realizar una verificación ordenada se ha seguido la disciplina de verificación llamada *Metric Driven Verification* (MDV) que sigue el flujo de estados mostrado en la figura 4.1:

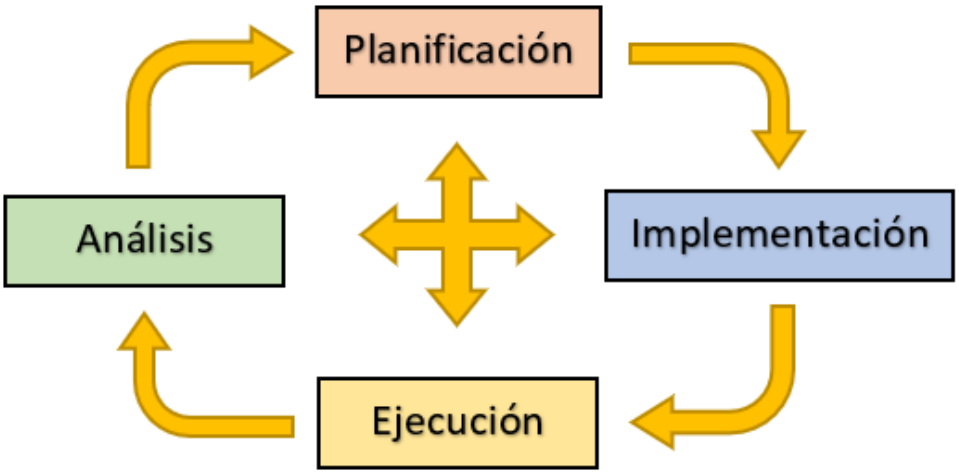


Figura 4.1: Flujo de estados de *Metric Driven Verification*

La disciplina de verificación empieza con el plan de verificación, seguido de su implementación, ejecución y análisis. Además, se muestra claramente que se puede ir en cualquier momento de una fase a otra. Esto es debido a que la verificación empieza casi al mismo momento que el diseño, por lo tanto a medida que avance el proyecto habrá que ir añadiendo más elementos para comprobar que los nuevos cambios y/o funcionalidades estén funcionando correctamente. Otros factores que pueden afectar también son temas de mantenimiento, mejora o adición de más elementos para ejercitar partes del diseño que no se estaban ejecutando con el actual plan de verificación.

Aplicando la disciplina a la verificación a nivel de bloque, cada estado del flujo contendría las siguientes acciones:

- **Planificación** de los elementos que se van a requerir para ejercitar todas las funcionalidades del diseño.
- **Implementación** del entorno y los elementos del plan de verificación.
- **Ejecución y análisis** o seguimiento de la verificación, donde se obtienen y analizan las métricas de cobertura ejecutando sets de tests.

4.2. Plan de verificación

En la fase inicial de un proyecto, el verificador dedica unas semanas a familiarizarse con el bloque a verificar. Para lograrlo invierte tiempo leyendo y procesando toda la documentación para identificar todas las funcionalidades y restricciones temporales que se deben comprobar.

En base a este estudio realizado se diseña un plan de verificación (VPLAN) con la herramienta vManager. Este plan contiene, divido por secciones organizadas por funcionalidad, todos los tests, *covergroups* y *assertions* que se van a implementar. Además, cada elemento dispone de una descripción y texto de implementación para facilitar su creación. La imagen 4.2 muestra como lucen estos elementos dentro de vManager:

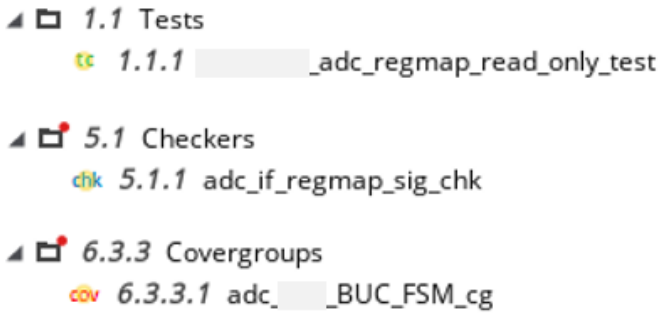


Figura 4.2: Elementos de verificación dentro de vManager

Nota: los "checkers" son las "assertions"

En base a la documentación y las características y funcionalidades del ADC mostradas en la sección "Introducción al bloque a verificar", se montó el siguiente VPLAN:

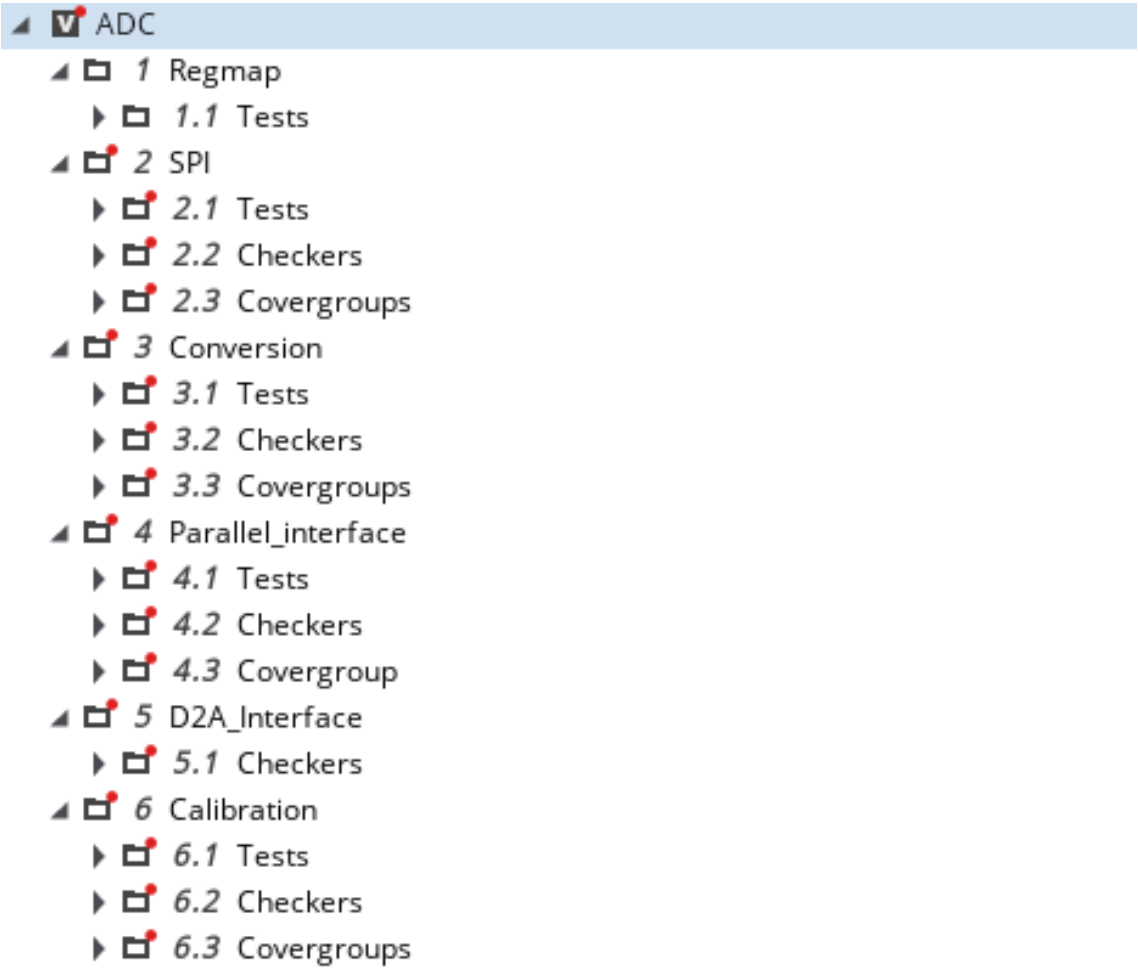


Figura 4.3: VPLAN del ADC

El VPLAN está formado por un total de 99 tests, 35 assertions y 10 covergroups.

4.2.1. Regmap

Tests Regmap:

1) adc_regmap_read_only_test:

- **Descripción:** comprueba que los registros de sólo lectura no puedan ser escritos de forma externa por el usuario vía SPI.
- **Implementación:**
  - Leer el valor de los registros de sólo lectura
  - Enviar transacciones SPI de escritura con las direcciones de estos registros

- Comprobar que el valor no se ha sobrescrito

2) adc_regmap_read_reset_value_test:

- **Descripción:** asegura que todos los valores después de reset (valor por defecto) son correctos
- **Implementación:**
  - Reset el ADC
  - Comprobar que el valor de reset de cada registro es el correcto

3) adc_regmap_write_cnv_read_test:

- **Descripción:** asegura que una conversión no afecte al valor de los registros
- **Implementación:**
  - Popular el regmap con datos randomizados
  - Realiza un conversión
  - Comprobar que no hayan cambiado los registros

4) adc_regmap_write_cal_read_test:

- **Descripción:** asegura que una calibración no afecte al valor de los registros
- **Implementación:**
  - Popular el regmap con datos randomizados
  - Realiza una calibración
  - Comprobar que no hayan cambiado los registros (excepto los de coeficientes)

4) adc_regmap_write_read_test:

- **Descripción:** asegura que todos los registros de lectura/escritura sean accesibles
- **Implementación:**
  - Para cada dirección (orden aleatorio) vía SPI:
    - Escribir un valor aleatorio
    - Leerlo
    - Comprobar que el valor leído es igual al escrito
  - Para cada dirección (orden de menor a mayor) vía SPI:
    - Escribir un valor aleatorio
  - Para cada dirección (orden de menor a mayor) vía SPI:
    - Leerlo
    - Comprobar que el valor leído es igual al escrito

- Para cada dirección (orden de mayor a menor) vía SPI:
  - Escribir un valor aleatorio
- Para cada dirección (orden de mayor a menor) vía SPI:
  - Leerlo
  - Comprobar que el valor leído es igual al escrito
- Para cada dirección (orden aleatorio) vía SPI:
  - Escribir un valor aleatorio
- Para cada dirección (orden aleatorio) vía SPI:
  - Leerlo
  - Comprobar que el valor leído es igual al escrito

4.2.2. SPI

Tests SPI:

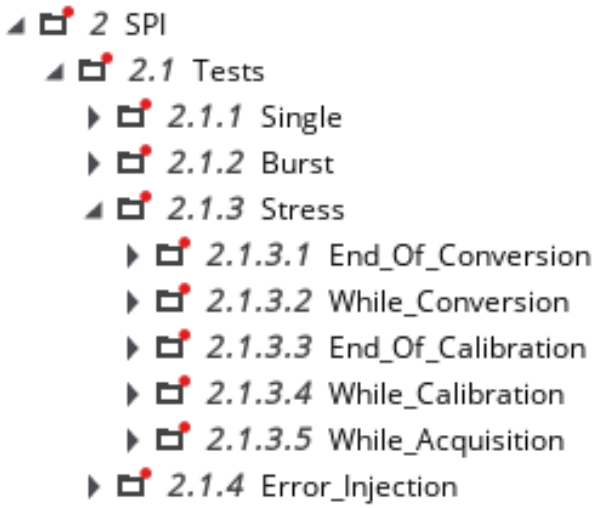


Figura 4.4: Secciones tests SPI

A partir de la sección SPI, esta y todas las próximas siguen la estrategia de usar un test base completamente randomizado para montar los demás. Los tests que sirven como base contienen variables randomizables que configuran cómo funciona el test.

Con esta estrategia se pueden obtener una gran variedad de tests en mayor o menor medida aleatorios en base a las restricciones que se agreguen. Esto dependerá de cuan específica queremos que sea la funcionalidad a ejercitar en el test.

Esta metodología de verificación ayuda a la paralelización en el análisis y la detección rápida de errores. Ya que se dispondrá de una serie de tests más directos que nos ayuden a identificar rápidamente qué funcionalidad falla y otros más aleatorios que estudien las diferentes combinaciones entre estas.

Es por ello que en cada sección, para no repetir la implementación, se explicará cómo funciona el test base y se mostrarán las variables de configuración que contiene. Posteriormente se describirá qué diferencia cada una de sus subsecciones y qué tipos de tests añaden.

Test base SPI:

Variables randomizables:

- Se randomizan al inicio:
  - Número de transacciones
  - Periodo SCLK
- Se randomizan para cada transacción:
  - Dirección
  - Valor de datos a escribir
  - Tipo de transacción (lectura o escritura)
  - Tipo de transacción (single o burst)
  - Retardo entre transacciones
  - Longitud de la transacción de tipo burst

Implementación:

- Popular el regmap con datos aleatorios vía SPI
- Randomizar variables
- Durante N_tran (variable que indica número de transacciones):
  - Transacción SPI con configuración randomizada
    - Cuando es de lectura: comprobar que el valor leído es igual al escrito
  - Introducir retardo entre transacciones
  - Randomizar variables para la próxima transacción

Single: todos los tests que contiene restringen las transacciones a que sean siempre de tipo single. Además de esta, algunos ejemplos de tests con restricciones específicas que añaden son:

- Test con sólo transacciones de lectura
- Test con sólo transacciones de escritura
- Test con máximo período de SCLK
- Test con mínimo período de SCLK
- Test con mínima separación entre transacciones

Burst: todos los tests que contiene restringen las transacciones a que sean siempre de tipo burst. Los tests que añade son los mismos que los de single pero con burst.

Stress: esta sección busca observar cómo se comporta el ADC dependiendo de cuándo se le envían las transacciones SPI. Por lo que contiene secciones con tests que envían transacciones al acabar una conversión, mientras se hace una conversión, al acabar una calibración o mientras se hace una calibración. El comportamiento esperado depende de cómo esté indicado en la documentación.

Error_Injection: es una sección con tests de inyección de errores que sirven para comprobar cómo de robusto es nuestro ADC frente a transacciones ilegales. Para ello se añaden más variables de randomización:

- Probabilidad de aparición de un error
- Qué error se introduce
- Qué valor tiene el error que se introduce (depende de cómo se pueda configurar el error)

Siendo los tipos de errores:

- Transacciones con direcciones ilegales (no contempladas en el regmap)
- Errores de la SPI VIP de Analog Devices

Por lo tanto esta sección tiene un test base que randomiza todo y otros que restringen qué tipo de error se introduce.

Checkers SPI: se reutilizan de una VIP de Analog Devices que asegura que el SPI está funcionando correctamente. Como por ejemplo:

- Que las transacciones tengan el correcto número de ciclos
- Que se envíen transacciones solamente cuando se baje el CS

Covergroups SPI: se aseguran de que se hayan realizado transacciones SPI con todas las direcciones para diferentes configuraciones, tales como: para cada test de stress, para cada tipo de inyección de error, para single y burst.

4.2.3. Conversión

Esta sección tiene el propósito de verificar el proceso de conversión y todas las funcionalidades que lo forman.

Tests Conversión:

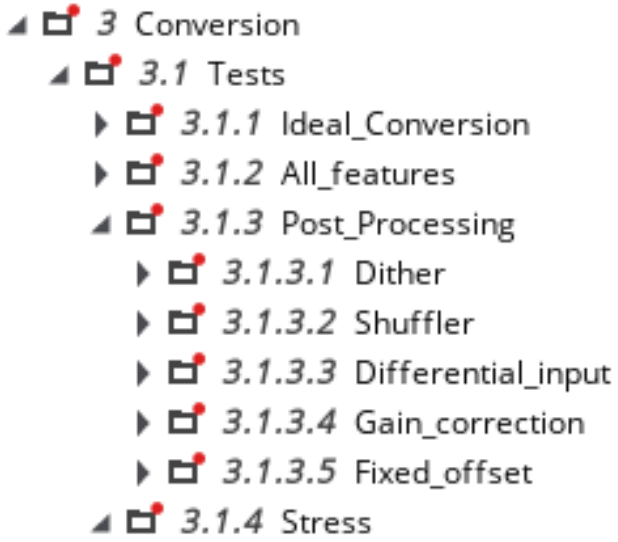


Figura 4.5: Secciones tests Conversión

Test base Conversión:

Variables randomizables:

- Número de conversiones
- Habilitar/deshabilitar dither
- Habilitar/deshabilitar shuffler
- Entrada diferencial, valores:
  - Saturación positiva
  - Saturación negativa
  - Valor mínimo negativo
  - Valor máximo positivo
  - Cero
  - Valor aleatorio entre el mínimo y máximo
  - Señal tipo rampa
  - Señal tipo cuadrada
- Popular el regmap (excepto coeficientes), valores :

    - MIN: a 0 todos los bits
    - RAND: aleatorio
    - MAX: a 1 todos los bits
- Popular los coeficientes del regmap
    - MIN: a 0 todos los bits
    - RAND: aleatorio
    - MAX: a 1 todos los bits
- Valor de offset
- Valor de ganancia

Implementación:

- Randomizar variables
- Popular regmap en base a las variables con MIN/RAND/MAX
- Durante N_conv (variable que indica número de conversiones):
    - Iniciar conversión
    - Al finalizar la conversión (EOC)
        - Obtener código de salida y datos del registro de depuración
        - Comprobar que tienen los valores correctos
        - Si tiene la entrada diferencial configurada como aleatorio, rampa o cuadrada se actualiza su valor para la siguiente conversión

Ideal_conversión: todas las variables están deshabilitadas (tienen valor 0 o MIN, dependiendo de cuál sea), por lo que sólo se randomiza la entrada diferencial. Además, con propósitos de cobertura incluye un test que configura la entrada diferencial para que sea de tipo rampa con un paso de 5 veces por código.

All_features: todas las funcionalidades están habilitadas y el regmap y coeficientes son RAND. También incluye el mismo test de rampa que el de la sección “Ideal_conversion”.

Post_Processing: contiene los tests relacionados con las funcionalidades que forman parte del post-procesado de la conversión. Cada sección contiene para cada funcionalidad que nombra, tests con los casos extremos para diferentes configuraciones de los coeficientes. Por ejemplo para dither sería:

- Test con Dither: habilitado + Coeff: MIN
- Test con Dither: habilitado + Coeff: RAND
- Test con Dither: habilitado + Coeff: MAX
- Test con Dither: deshabilitado + Coeff: MIN

- Test con Dither: deshabilitado + Coeff: RAND
- Test con Dither: deshabilitado + Coeff: MAX

Stress: contiene el test base de conversión y añade tests que inyectan pulsos erróneos de inicio de conversión o calibración a mitad de la conversión. El propósito no es que funcionen las conversiones con un pulso inyectado sino observar si el circuito se recupera y continúa funcionando correctamente después. Es por ello que estos últimos añaden variables de randomización como las comentadas en la sección de "Error_injection" del SPI.

Checkers Conversión:

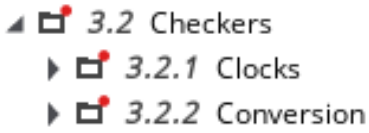


Figura 4.6: Secciones checkers Conversión

Las *assertions* se pueden dividir en dos secciones:

Clocks: se centran en analizar el correcto funcionamiento de los relojes y la relación entre ellos durante la conversión. Siendo estos el reloj interno que funciona durante el algoritmo SAR y el de fin de conversión (EOC). Estas *assertions* comprobarán que se cumplan comportamientos tales como:

- Los relojes tengan el ciclo y ancho de pulso correcto
- EOC aparezca sólo una vez durante la conversión
- Que no funcionen estos relojes cuando no hay una conversión (busy = 0)
- Que el reloj SAR pare cuando ya se hayan calculado todos los bits

Conversión: se centran en comprobar que las señales que puedan afectar a la conversión funcionen correctamente. Analizando así escenarios como:

- Que se cumplan anchos de pulsos indicados en la documentación
- Se produzcan el correcto número de ciclos de reloj durante la conversión
- Que el ADC aparezca como ocupado durante la conversión (busy = 1)
- Que la conversión pare si hay un reset

Covergroups Conversión: los puntos a cubrir implementados se enfocan en que se haya conseguido pasar por todos los códigos posibles de salida del ADC. Además, estos puntos se cruzan con casos como que todas las funcionalidades estén habilitadas, deshabilitadas y para diferentes valores de offset y ganancia.

4.2.4. Interfaz Paralela

Cuando hay varios relojes conviviendo (en este caso el interno del SAR y el SCLK del SPI) pueden aparecer problemas de sincronización dependiendo de cuando caigan sus flancos. Esta sección se centra en estudiar si los datos a monitorizar (código final de la interfaz paralela junto con los registros de depuración) se obtienen correctamente para diferentes retardos entre los relojes.

Tests Interfaz Paralela: para ello se implementan una serie de tests que se basan en el base de conversión pero añaden más variables a randomizar, siendo estas el periodo de SCLK y el retardo entre el reloj SAR y el SCLK.

Checkers Interfaz Paralela: añade una *assertion* que comprueba que el dato de salida es estable al acabar la conversión y que así lo es hasta la próxima conversión.

Covergroups Interfaz Paralela: contiene un covergroup para que indique si todos los valores posibles de retardo que se han definido se han utilizado.

4.2.5. Interfaz Digital-Analógica

Esta sección contiene solo *assertions* que aseguran el correcto funcionamiento de las señales que van de la parte digital a la analógica:

- *assertions* que comprueban que cuando no haya conversión (no se mueve el reloj SAR) las señales sean estables
- *assertions* que aseguran que los campos con funcionalidades analógicas del mapa de registros se transmitan correctamente a la parte analógica.

4.2.6. Calibración

Esta sección tiene el propósito de verificar el proceso de calibración y todas las funcionalidades que lo forman.

Tests Calibración:

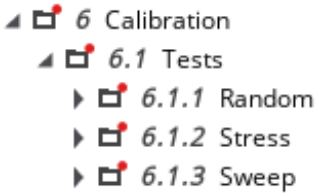


Figura 4.7: Secciones tests Calibración

Test base Calibración:

Variables randomizables:

- Bits a calibrar (MIN/RAND/MAX, siendo MIN el por defecto con todo habilitado)
- Habilitar/deshabilitar calibración de offset
- Habilitar/deshabilitar calibración de ganancia
- Promedio de muestras para cada coeficiente

Implementación:

- Inyectamos errores en las capacidades para degradar la linealidad
- Popular regmap en base a las variables randomizables
- Iniciar calibración
- Al finalizar la calibración (EOCal)
  - Obtener coeficientes via SPI
  - Comprobar que tienen los valores correctos
- Resetear el sistema
- Cargar los coeficientes calculados con la calibración
- Se realiza una comprobación de linealidad:
  - Se introduce una rampa que vaya del mínimo al valor máximo de la entrada diferencial y que incremente cada 5 conversiones. Al ser de 16 bits, el incremento será de $\frac{Vmax-Vmin}{2^{16}*5}$.
  - Mientras se realice la conversión se obtiene el DNL e INL
- Al finalizar las conversiones se analizan los valores máximos y mínimos de DNL e INL para observar si estamos dentro de especificaciones y el algoritmo de calibración ha mejorado la respuesta del ADC.

Random: contiene el test base de calibración y los que configuran las variables randomizables con los casos extremos (habilitado/deshabilitado o valor mínimo/máximo).

Stress: tienen el mismo propósito que los de conversión pero durante la calibración.

Sweep: tienen el propósito de obtener un 100 % de cobertura con los valores de coeficientes. Como existe una relación entre las capacidades y los coeficientes calculados, cada test realiza durante N_calibraciones un barrido del valor de una capacidad hasta obtener los valores mínimo y máximos saturados del coeficiente. Por lo tanto en este test la implementación respecto al base llega hasta el reset del sistema, no se realiza el análisis de linealidad y se comprueba únicamente los valores calculados en la calibración.

Checkers Calibración: se encargan de comprobar que la máquina de estados de la calibración funciona correctamente respecto a la configuración que se haya hecho en el test.

Covergroups Calibración: contiene covergroups que aseguran que todos los bits a calibrar hayan pasado por todos los estados posibles y cruzan las diferentes combinaciones entre funcionalidades de calibración (variables randomizables).

4.3. Entorno de verificación

En base al plan de verificación se monta un entorno de verificación que nos permita implementar todas sus elementos. El diagrama del entorno se muestra a continuación:

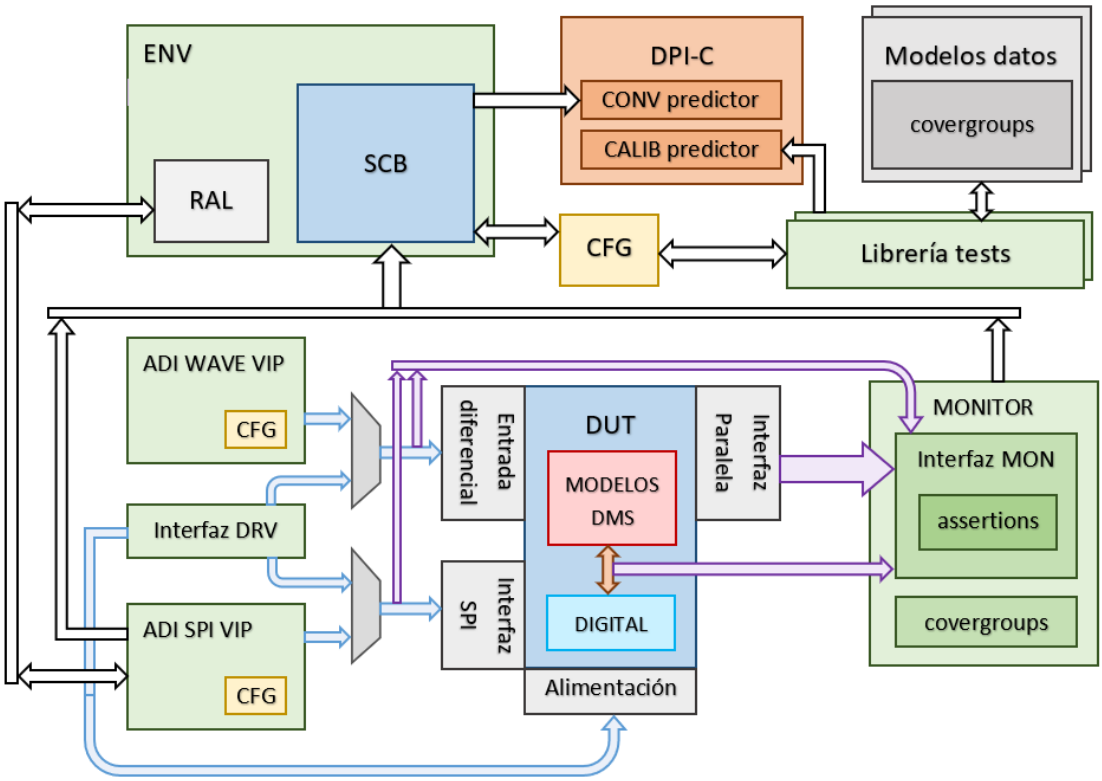


Figura 4.8: Diagrama de bloques del entorno de verificación

4.3.1. RAL

El RAL o capa de abstracción de registro es una entidad utilizada en UVM para modelar mapas de registros. Conteniendo así toda la información de los registros y campos que contengan, como es el ancho en bits de cada campo y registro, si son registros sólo de lectura o de lectura/escritura y los valores por defecto.

Además, se comunica con el agente de SPI (VIP de Analog Devices) para proporcionarnos una serie de funciones para enviar transacciones SPI o obtener información del regmap.

4.3.2. VIP

SPI VIP: se comunica con el RAL y se encarga de enviar transacciones SPI al ADC. Al instanciarlo en el entorno se ha configurado para que satisfaga las propiedades de nuestra aplicación. Además, monitoriza las transacciones para enviar paquetes de secuencias al scoreboard para que las procese.

WAVE VIP: se encarga de generar señales para inyectarlas en la entrada diferencial del ADC. Para lograrlo nos proporciona una colección de funciones que nos permiten configurarlas a placer. Al igual que la del SPI, se ha configurado al añadirla al entorno.

4.3.3. Modelos predictores

El equipo analógico proporcionó un modelo de MATLAB que emulaba el comportamiento del ADC. Este modelo fue reutilizado para generar funciones DPI-C usando la herramienta "MATLAB Coder". Esta herramienta se encarga de generar funciones en código C en base a funciones en código MATLAB.

Estas funciones DPI-C proporcionaron:

- Un predictor del resultado de conversión:
  - Entradas:
    - Valor muestreado de entrada
    - Habilitación/Deshabilitación y valor de dither
    - Habilitación/Deshabilitación y valor de shuffler
    - Coeficientes de calibración
    - Offset
    - Ganancia
  - Salida: resultado esperado del ADC
- Un predictor del resultado de la calibración:
  - Entradas: errores inyectados en las capacidades
  - Salidas: coeficientes de calibración
- Un predictor del resultado del dither y shuffler (la misma función sirve para los dos):
  - Entrada: valor actual del dither o shuffler
  - Salida: próximo valor del dither o shuffler

Los beneficios de utilizar funciones DPI-C frente a las de MATLAB en SystemVerilog es que se pueden ejecutar en tiempo de simulación y por lo tanto se pueden utilizar en un ratio de conversión-a-conversión.

4.3.4. Configuración

Contiene la configuración de los diferentes agentes que se instancian en el entorno y variables de configuración que son accedidas por los tests y el scoreboard.

Entre estas variables de configuración se encuentran:

- Tiempo de carrera de la simulación (tiempo máximo de simulación, si llega a él se para automáticamente con un error)
- Habilitar/Deshabilitar scoreboard
- Habilitar/Deshabilitar assertions (variable general y variables para cada sección del vplan)
- Variables de entrada de las funciones DPI-C

Además, tiene embebidas las funciones que actualizan automáticamente el valor de dither y shuffler para la siguiente conversión.

4.3.5. Modelos de datos

Como se ha observado en el plan de verificación muchas variables son requeridas para ser randomizadas y restringidas. Es por ello que decidimos crear una serie de objetos UVM ordenados por secciones/funcionalidades para ser reutilizados entre tests/secciones. Por lo tanto hay modelos de datos de:

- Entrada diferencial
- Conversión
- Calibración
- Interfaz paralela
- SPI

Para calibración sería:

```
3    // Variables
4    rand regmap_e coeff_config;
5    rand regmap_e reg_config;
6
7    rand regmap_e bit2cal;
8    rand bit2offset;
9
10   rand int num_cal;
11
12   rand bit en_gain_cal;
13   rand bit avebitcount;
14
15   rand int  err_prob;
16   rand time err_period;
17   rand time err_delay;
18
19   rand int cal_delay;
```

Figura 4.9: Variables del modelo de datos de calibración

```
42   constraint num_cal_value{
43      num_cal inside {[0:1000]};
44   }
45
46   constraint prob_value{
47      coeff_config dist{MIN:= 5, MAX:=5, RAND:= 90};
48      reg_config   dist{MIN:= 5, MAX:=5, RAND:= 90};
49      bit2cal      dist{MIN:= 90, MAX:=5, RAND:= 5};
50   }
51
52   constraint error_type{
53      err_prob   inside {[30:100]};
54      err_period inside {[62.5ns:100ns]};
55      err_delay  inside {[0ns:2000ns]};
56   }
57
58
59   constraint cnvst_value{
60      cal_delay    inside {[1:10]};
61   }
62
63   // Constructor
64    function new(string name = "calib_data_model");
65      super.new(name);
66   endfunction : new
```

Figura 4.10: Restricciones por defecto del modelo de datos de calibración

Si quisiéramos por ejemplo en un test utilizar las variables de calibración y de la entrada diferencial, simplemente tendríamos que añadir:

```
1 calib_data_model calib_data;
2 in_data_model    diff_in;
3
4 calib_data = calib_data_model::type_id::create("calib_data");
5 diff_in    = in_data_model::type_id::create("diff_in");
```

Figura 4.11: Uso de los modelos de datos en tests

Una vez ya tenemos el objeto en el test ya podemos acceder a todas las variables que contiene, pudiendo así, randomizarlas o restringirlas a placer.

Además, contienen todos los covergroups que sólo dependan de las variables a randomizar.

4.3.6. Librería de tests

La librería de tests contiene todos los test implementados del plan de verificación, siguiendo la misma estructura de carpetas de éste.

4.3.7. Interfaces driver

Son las interfaces que utilizan los tests para conectarse y transmitir señales al ADC.

Una de ellas contiene variables conectadas al bloque para alimentar el circuito y enviar las señales de control del ADC (reset, empezar calibración o conversión). Además, tiene la posibilidad de inyectar los valores de la entrada diferencial e interfaz SPI directamente sin usar las VIP. La selección de qué señal se utilice vendrá determinada por la señal de control del multiplexor asociado, que se encuentra en el driver.

La otra interfaz tiene la posibilidad de inyectar errores de capacidades cambiando su valor manualmente, por lo que también usa el método del multiplexor para que el usuario elija si las inyecta él o se añaden automáticamente desde el modelo analógico.

4.3.8. Monitor

Monitoriza las conversiones para enviar al scoreboard el valor de entrada muestreado en la entrada diferencial y el resultado de la conversión en la interfaz paralela.

Dentro de él está la interfaz de monitorización, que monitoriza señales de interés, como son las inyectadas al bloque, las de la interfaz digital-analógica y las de salida. Esta interfaz contiene a su vez las *assertions* debido a que monitoriza muchas de las señales que serán utilizadas por ellas.

4.3.9. Scoreboard

El scoreboard recibe los paquetes de secuencias del monitor y la SPI VIP.

Con los datos recibidos del monitor ejecuta al finalizar una conversión el predictor de conversión y actualiza los valores de dither y shuffler que serán utilizados en la próxima.

Con los datos recibidos de la SPI VIP actualiza los valores de las variables relacionadas con el mapa de registros, como son algunas variables de configuración y las variables utilizadas en las assertions que comprueban que se mapeen correctamente las señales que se transmiten a lo analógico.

4.3.10. Modelos DMS

Un aspecto clave de la verificación ha sido el uso de modelos RNM con diferentes niveles de abstracción, uno más funcional y otro de más bajo nivel.

El modelo de alto nivel captura la mayoría del comportamiento analógico del ADC y está basado en el modelo de MATLAB. Este modelo tiene el fin de emular de forma precisa las secuencias digitales de control durante las conversiones.

Por otro lado, el modelo de bajo nivel desciende, como se indica en el nombre, más abajo en la jerarquía analógica, llegando hasta la descripción y modelado de los interruptores analógicos. Permitiendo así inyectar errores en los valores de capacidad del CDAC para hacer funcionales los tests de calibración.

De esta forma, con el modelo de alto nivel se consiguen simulaciones muy rápidas y precisas relacionadas con la conversión, y con el de bajo nivel se hacen posibles los análisis de características dinámicas y estáticas del ADC.

4.4. Implementación del plan de verificación

4.4.1. Tests

Para una rápida implementación y creación de los tests se utilizó una herramienta interna que parsea el plan de verificación y genera todo el directorio de tests automáticamente. Creando así todas las carpetas y ficheros de la librería de tests siguiendo la estructura del VPLAN.

Una vez creado el directorio, lo que sigue es la implementación de los tests.

Cabe destacar que todos ellos extienden de uno base que les da acceso al RAL, variables de configuración, interfaces (driver y monitor) y a una librería de funciones para reutilizar código entre tests y facilitar su implementación.

La estrategia que se ha seguido en el código de los tests ha sido de hacerlos lo más descriptivos y cercanos posible al texto de implementación del plan de verificación. Por lo que se han implementado un conjunto de funciones parametrizables (con entradas) para reutilizarlas entre ellos.

Es por ello que la estructura general de un test será instanciar los modelos de datos que necesite y randomizarlos a placer, alimentar el circuito, enviar un reset y finalmente implementar la lógica del test con funciones de la librería.

Por ejemplo, para el test de conversión con dither, el cuerpo tendría el siguiente código:

```
7  data = conv_data_model::type_id::create("data");
8  in   = in_data_model::type_id::create("in");
9
10 rand_data_model();
11
12 lib.set_supply_and_bias(in.avdd, in.dvdd, in.vref);
13
14 lib.drive_reset();
15
16 set_diff_input(in);
17
18 lib.populate_regmap_data(data.coeff_config, data.reg_config);
19
20 data.en_dither = 1;
21 lib.set_conv_features_bitfields(data.en_dither, data.en_shuffler, data.offset, data.gain);
22
23 repeat(data.num_conv) begin
24   fork
25     begin
26       lib.do_conv(data.eoc_delay);
27     end
28
29     begin
30       @(negedge mon_if.busy);
31       if(in.is_wave)
32         update_diff_input(in);
33     end
34   join
35 end
36
37 lib.adc_power_down ();
```

Figura 4.12: Ejemplo: implementación de test de conversión con dither

Explicación por líneas de código:

- **7, 8 y 10:** instancia los modelos de datos de conversión y entrada diferencial, y los randomiza
- **12:** alimentamos el circuito en base a la alimentación analógica (avdd), digital (dvdd) y el voltaje de referencia (vref)
- **14:** se resetea el circuito con un ancho de pulso predeterminado
- **16:** se introduce el valor de la entrada diferencial, que dependerá de qué tipo se haya obtenido en la randomización
- **18:** se popula el regmap en base a los valores MIN/RAND/MAX de las variables para los registros de coeficientes y para el resto del regmap
- **20:** como este test siempre tiene dither, se restringe su habilitación a 1
- **21:** se sobrescribe el mapa de registros en función de las variables randomizadas de conversión
- **23:** durante N conversiones, de forma paralela (fork):
  - **26:** do_conv: empieza una coversión, se espera a que acabe o se interrumpa con un reset y al finalizar añade un retardo (eoc_delay), siendo este por lo tanto la separación entre conversiones

- **30, 31 y 32:** si la entrada diferencial está configurada como random, rampa o señal cuadrada, se actualiza el valor para la siguiente conversión

- **37:** se apaga el ADC

4.4.2. Assertions

Todas las *assertions* y la lógica que requieren están implementadas en una misma interfaz llamada "adc_assert" que se instancia en la interfaz monitor del bloque.

Cada *assertion* tiene un comentario con su nombre y explicación de qué hace. Además, todas ellas tienen un mensaje de error cuando fallan para ayudar con su depuración.

Un ejemplo sería esta que analiza que el pulso de que ha finalizado la conversión (eoc) ocurra una vez por conversión, es decir, mientras que busy es 1:

```
8  //================================================================
9  // Name: adc_eoc_clk_once_wconv_chk
10 // Details : eoc_clk only toggles once per busy = 1
11 //================================================================
12
13 property adc_eoc_clk_once_wconv_chk;
14   @(negedge busy) disable iff(rstb !== 1 || disable_conv_checks === 1 || disable_checks === 1)
15   eoc_count == 1;
16 endproperty
17 assert property (adc_eoc_clk_once_wconv_chk)
18   else `uvm_error("CONV_ASSERTION", $sformatf("wrong number of eoc_clk pulses per busy"))
```

Figura 4.13: Ejemplo: implementación de *assertion*

La lógica del contador (eoc_count) va embebida en el mismo fichero y se genera en base a las señales monitorizadas:

```
1 always @(posedge eoc or negedge busy) begin //Count how many eoc per busy
2   if(!busy)
3     eoc_count = 0;
4   else
5     eoc_count++;
6 end
```

Figura 4.14: Ejemplo: lógica de *assertion*

Y en tiempo de simulación:

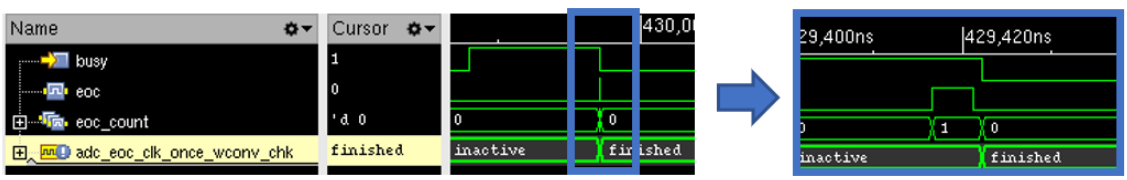


Figura 4.15: Ejemplo: *assertion* en tiempo de simulación

Como se puede observar en 4.13 hay dos variables de deshabilitación (las variables “disable”), el valor de estas señales dependen de unas variables de configuración con el mismo nombre. Existe una variable general que desactiva todas (disable_checks) y una para cada sección del VPLAN. De esta forma se puede controlar des de los tests cuáles no se quieren utilizar.

4.4.3. Covergroups

Se han seguido dos estrategias para implementarlos.

Aquellos que dependían enteramente de las variables randomizables se han creado dentro del modelo de datos al que pertenecen. Un ejemplo podría ser, el *covergroup* que analiza que se hayan ejecutado tests con todas las combinaciones de funcionalidades de conversión, por lo que dentro del modelo de datos de conversión:

```
44 covergroup adc_conv_features_cg;
45   option.per_instance = 1;
46   type_option.merge_instances = 1;
47
48   cp_dither : coverpoint en_dither;
49
50   cp_shuffler : coverpoint en_shuffler;
51
52   cp_offset : coverpoint has_offset;
53
54   cp_gain: coverpoint has_gain;
55
56   xc_features: cross cp_dither, cp_shuffler, cp_offset, cp_gain;
57 endgroup : adc_conv_features_cg
```

Figura 4.16: Ejemplo: *covergroup* dentro de un modelo de datos - implementación

```
60 // Constructor
61 function new(string name = "conv_data_model");
62   super.new(name);
63
64   adc_conv_features_cg = new();
65
66   adc_conv_features_cg.set_inst_name({get_full_name(), ".adc_conv_features_cg"});
67 endfunction : new
```

Figura 4.17: Ejemplo: *covergroup* dentro de un modelo de datos - creación

Posteriormente hay que indicar cuando se muestrea, en este caso un buen lugar sería justo después de la función “set_conv_features_bitfields” de la figura 4.12:

```
data = conv_data_model::type_id::create("data");
in   = in_data_model::type_id::create("in");

rand_data_model();

lib.set_supply_and_bias(in.avdd, in.dvdd, in.vref);

lib.drive_reset();

set_diff_input(in);

lib.populate_regmap_data(data.coeff_config, data.reg_config);

data.en_dither = 1;
lib.set_conv_features_bitfields(data.en_dither, data.en_shuffler, data.offset, data.gain);

sample_adc_conv_features_cg();

repeat(data.num_conv) begin
  fork
    begin
      lib.do_conv(data.eoc_delay);
    end

    begin
      @(negedge mon_if.busy);
      if(in.is_wave)
        update_diff_input(in);
    end
  join
end

lib.adc_power_down ();
```

Figura 4.18: Ejemplo: *covergroup* dentro de un modelo de datos - muestreo

El otro enfoque fue implementarlos dentro de suscriptores de UVM. Un suscriptor es un componente de UVM que como indica su nombre se suscribe a un agente transmisor del que recibe objetos cada vez que un paquete es enviado vía su puerto conectado.

Cuando se recibe un paquete en el suscriptor se ejecuta la función de tipo void write. Permitiéndonos así monitorizar los paquetes y extraer la información requerida en nuestros *covergroups*.

Otra ventaja que nos proporciona es que tienen la task run_phase, que es la fase en que UVM ejecuta la simulación. Es decir, que lo que se implemente dentro de ella se ejecutará en paralelo a la simulación. Por lo tanto se puede implementar toda la lógica de monitorización y muestreo necesaria de los *covergroups* más complejos dentro de ella.

En definitiva, con un suscriptor podemos contener en un mismo sitio todo lo relacionado a una serie de *covergroups*. Para SPI por ejemplo se crea un suscriptor conectado a los paquetes generados por la SPI VIP:

```
class adc_spi_cov extends uvm_subscriber #(advip_spi_mem_tr);
```

Figura 4.19: Ejemplo: suscriptor UVM - tipo de paquetes

Que nos recoja la información necesaria de los paquetes que envía en la función write, como es por ejemplo la dirección:

```
121   virtual function void write(advip_spi_mem_tr t);
122     mem_tr           = advip_spi_mem_tr::type_id::create("mem_tr", this);
123
124     ...
125
126     mem_tr.addr      = t.addr;
127
128     ...
129
130     adc_spi_dir_cg.sample();
131
132   endfunction : write
```

Figura 4.20: Ejemplo: función write suscriptor UVM

Y que tenga una run_phase que recoja información. Por ejemplo, muestrear el tipo de error si la transacción SPI que ha finalizado tenía alguna clase de error inyectado:

```
134    virtual task run_phase(uvm_phase phase);
135       forever begin
136         @(posedge mon_if.spi_access);
137         @(negedge mon_if.spi_access);
138
139         ...
140
141          error_type = mon_if.error_type;
142
143         if (error_type!=0)
144             adc_spi_err_inj_cg.sample();
145
146         ...
147
148       end
149    endtask
```

Figura 4.21: Ejemplo: task run_phase suscriptor UVM

4.5. Seguimiento de la verificación

Durante el proceso de verificación se deben configurar herramientas que nos ayuden a tener métricas de cómo avanza y otras que nos aseguren que los nuevos cambios (tanto de diseño como de verificación) no rompan el entorno o tests que ya funcionaban.

Para solucionar ambos problemas se utilizan regresiones, que son ficheros con un grupo de tests que se pueden ejecutar todos con un comando. Cada línea que contiene es el comando de un test y por lo tanto se puede configurar cada uno con los argumentos que requiera e indicar el número de veces que queremos que se ejecute (para que se ejecute con diferentes semillas).

Estos ficheros se configuran en la herramienta Jenkins para ser ejecutados automáticamente. Se han configurado dos:

- **Lista de paso:** contiene tests con ejercitan funcionalidades básicas del ADC con una semilla que debería funcionar siempre. Esta regresión se ejecuta siempre que haya un cambio en el entorno de bloque del ADC, tanto de diseño como de verificación. Por lo tanto nos sirve para observar si hemos roto algo al subir un cambio.
- **Regresión nocturna:** contiene todos los tests del ADC y se ejecuta con métricas de cobertura encendidas. Como indica el nombre se ejecutará todos los días de noche. Los resultados de esta regresión se envían automáticamente vía correo en formato de tabla, para que si algún test falla se indique el mensaje de error y la semilla para reproducirlo. Los resultados también aparecen automáticamente en vManager para tener un seguimiento de cómo avanza la verificación.

Cuando se ejecuta una regresión con métricas encendidas, se pueden analizar las métricas de cobertura de código y funcional.

Para analizar las funcionales se debe cargar el plan de verificación y mapear los elementos implementados en el entorno de verificación en los del VPLAN. Esto se configura una vez y ya quedan sincronizados para las próximas regresiones. Por lo tanto, estas métricas nos indicarán qué elementos faltan por implementar y depurar en caso de que fallaran.

Por otro lado, las métricas de código se pueden observar automáticamente en la pestaña de métricas de vManager, que nos muestra toda la jerarquía del ADC. Un aspecto importante en el análisis de estas métricas es que en un diseño habrán algunas que no se puedan cumplir, algunos ejemplos:

- Señales que son internas y constantes, por lo tanto no tendrán transiciones en sus bits
- Parte de una expresión lógica que no es accesible por las variables de las que depende, por ejemplo una señal constante
- Funcionalidades de la IP que no se utilizan en el proyecto y no se ejecutan

Para obviar estos casos se crea un fichero de exclusiones que como indica el nombre, excluyen estos casos. En cada exclusión se añade un comentario de porqué se ha añadido.

Capítulo 5

Resultados

Este capítulo muestra los resultados finales obtenidos en la verificación del ADC, mostrando las métricas obtenidas, resultados del rendimiento (características estáticas y dinámicas) y el número de errores (o bugs) encontrados.

5.1. Tests

La regresión nocturna final lanza todos los tests con varias semillas (el número de veces que se ejecuta cada uno depende de cuan aleatorio sea el test), siendo el número final de 893 simulaciones.

Como se puede observar en la 5.1 se ha conseguido que el 100 % de ellos funcionen:

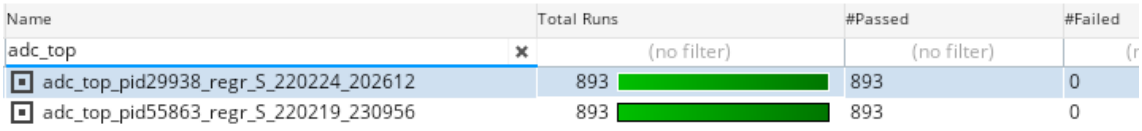


Figura 5.1: regresión con 100 % de tests funcionando

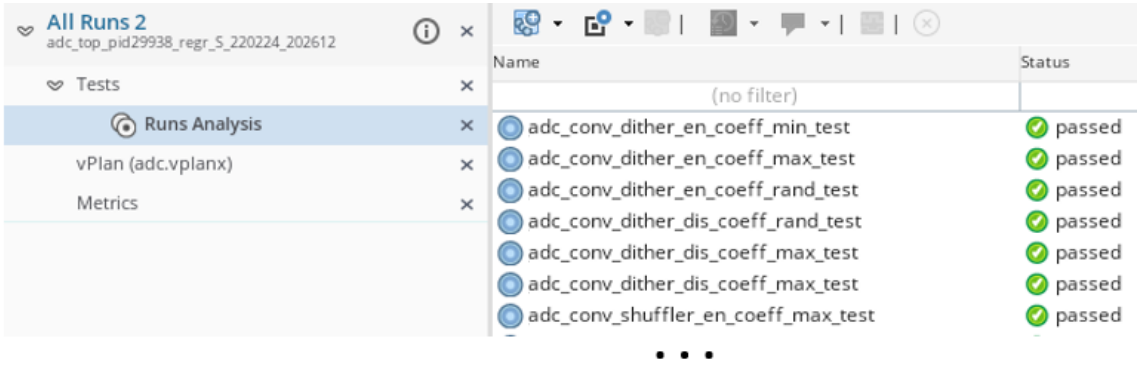


Figura 5.2: tests de la regresión

Y se ha mantenido limpia la regresión hasta el final del proyecto.

5.2. Cobertura funcional y de código

También se ha logrado el propósito de tener 100 % en todas las métricas de cobertura.

Cobertura funcional:

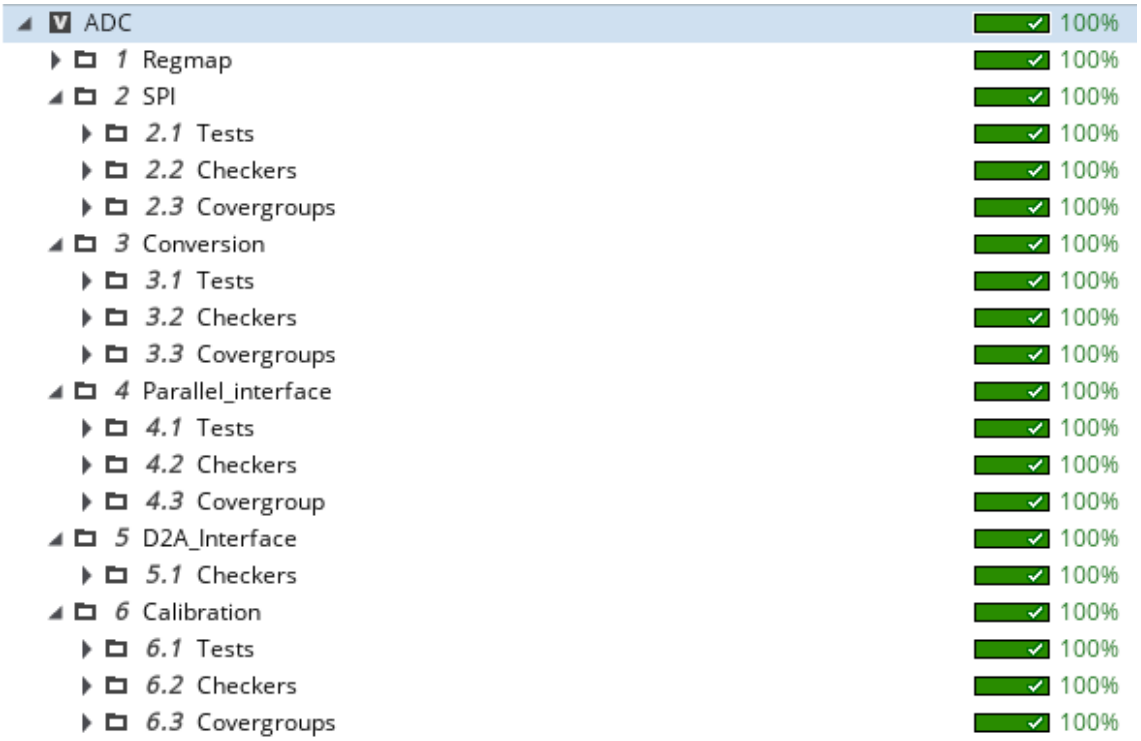


Figura 5.3: 100 % de cobertura funcional

Cobertura de código:

Figura 5.4: 100 % de cobertura de código

5.3. Rendimiento

En la disciplina de DMS, han asegurado que nuestras comprobaciones en los tests se han realizado correctamente con el modelo de bajo nivel. Para lograrlo han exportado los datos obtenidos en las simulaciones en ficheros de texto para ser procesados con MATLAB. De esta forma se comparan con nuestros análisis, y de paso comprueban que el comportamiento del modelo de bajo nivel coincide con el del esquemático (diseño analógico). Otra ventaja que nos proporciona MATLAB es que puede generar gráficos para que de forma visual se observe el efecto de las funcionalidades del ADC. Algunos ejemplos:

El efecto de shuffling, al decorrelar el ruido hace que el espectro de ruido se expanda por lo que métricas como el THD mejorarán y SNR empeorarán un poco. Esto equivale a ver como hay picos de armónicos con amplitud más pequeña y que el ruido base está un poco más elevado:

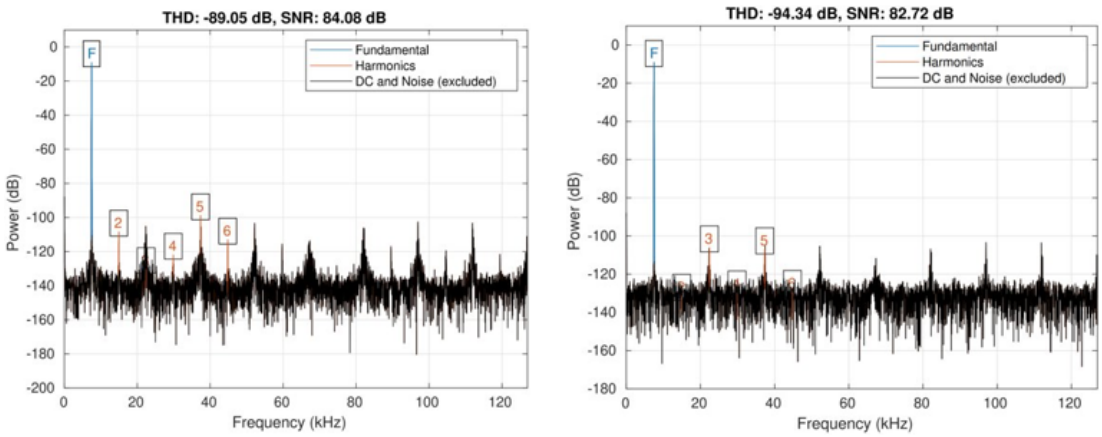


Figura 5.5: Efecto shuffling

En características estáticas como el INL, se observa una mejora drástica cuando activamos el dither + shuffler + corrección de coeficientes:

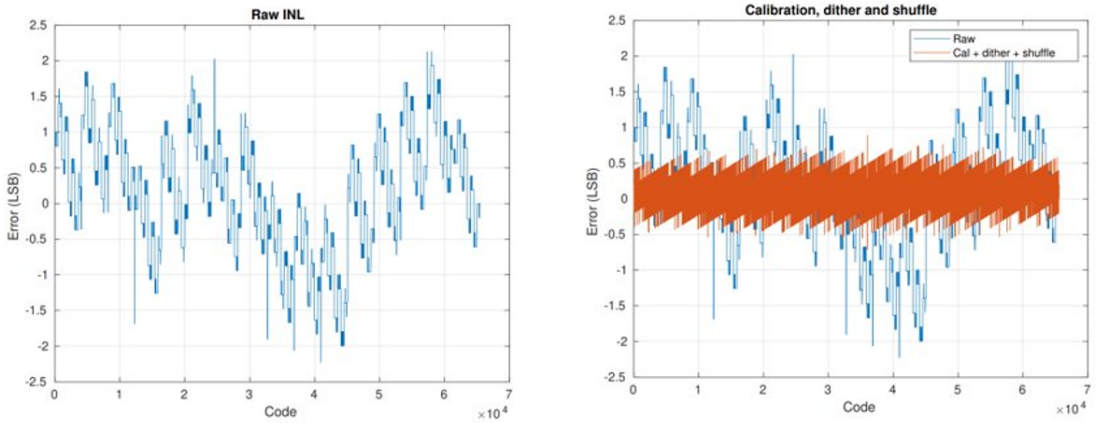


Figura 5.6: Efecto dither + shuffler + corrección de coeficientes

5.4. Errores

A lo largo de la verificación del bloque se han encontrado 10 errores en el diseño. La mayoría de ellos han sido causados por el proceso de integración de la IP y por fallos que están fuera del alcance del diseño y que solo se van a poder vislumbrar en la verificación comportamental. La figura 5.7 muestra con porcentajes la evolución en meses de como han ido apareciendo:

Figura 5.7: Evolución en porcentaje de detección de errores por meses

La mayoría de errores se encontraron en Septiembre y Octubre, que se corresponden con los meses en los que la mayoría de elementos del plan de verificación estaban implementados (diagrama temporal en figura 1.1). Posteriormente, a medida que avanza la verificación el número desciende.

5.5. Resumen

A modo de resumen, se ha montado la tabla 5.8 con los objetivos cumplidos con la verificación:

Lista de acciones	¿Conseguido?
Cobertura funcional al 100%	☑
99 tests pasando	☑
35 assertions pasando	☑
10 covergroups al 100%	☑
Cobertura de código al 100%	☑
Regresión nocturna con 100% de tests funcionando	☑

Figura 5.8: Objetivos cumplidos

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusión

La integración de un bloque que requiera ser verificado des de cero en un sistema, conlleva sus riesgos en el desarrollo de un producto debido a la incertidumbre que da y los plazos de entrega ajustados.

Con la verificación a nivel de bloque se ha conseguido focalizar la verificación al ADC para estresarlo lo máximo posible. Además, su verificación ha sido más rápida, debido a que las simulaciones son mucho más rápidas que a nivel de sistema porque se compilan menos elementos.

La disciplina de verificación y las diferentes estrategias aplicadas han permitido realizar un análisis e implementación rápido, eficaz y eficiente. El uso de funciones DPI-C para implementar predictores han mejorado el rendimiento de los tests ya que se ejecutan en tiempo de simulación. Además, la combinación de utilizar dos modelos para imitar el comportamiento analógico ha probado que es un buen enfoque, debido a la buena relación entre tiempo de simulación y precisión que han aportado.

Por un lado, el modelo de alto nivel, implementado en las primeras fases del proyecto, ha permitido la detección rápida de errores verificando el ADC para simulaciones en las que la calibración no es importante.

Por otro lado, el modelo de bajo nivel nos ha aportado un nivel muy alto de precisión, proporcionando así, un mayor grado de seguridad de que el bloque funciona. Otro punto importante ha sido la posibilidad de inyectar errores en el valor de las capacidades para hacer funcionales las simulaciones de calibración. Caracterizando así, las dependencias entre capacidades-coeficientes y las características dinámicas y estáticas, que han beneficiado a las diferentes disciplinas que convivían en la verificación del ADC, como son el equipo analógico, el digital y el de DMS.

Otro punto interesante que se ha conseguido con este trabajo es la reutilización. Gracias a la estrategia que se ha seguido en el entorno de verificación del bloque, se han reutilizado muchos de sus componentes para el del sistema. Siendo estos los modelos predictores, el fichero de configuración, las interfaces, el monitor, el scoreboard, los modelos de datos y los modelos DMS. Ya que la única acción que se ha tenido que hacer ha sido cambiar sus conexiones con las de nivel de sistema.

A la vista de los resultados, podemos concluir que se ha conseguido una verificación satisfactoria, en la que se han identificado durante el proyecto y de forma paralela a la implementación del bloque un total de 10 errores. Permitiendo así, que los diseñadores los solucionaran para prevenir que el chip funcione incorrectamente e introduzca costes añadidos durante su revisión y arreglo post-silicio.

6.2. Propuesta de trabajo futuro

Como propuesta de trabajo futuro y mejora, se podrían añadir más variables al fichero de configuración que unifiquen variables de funcionalidades que aparezcan repetidas en varios elementos del entorno. Un ejemplo sería parametrizar todas las variables relacionadas con el mapa de registros, como son la dirección máxima que tiene (o ancho del regmap) y en qué rango están los coeficientes de calibración. De esta forma los cambios que pudieran aparecer debido a adaptaciones de la IP para usarlas en nuevos diseños o nuevas versiones del mismo, se podrían adaptar en el entorno de verificación de forma más rápida.

Capítulo 7

Agradecimientos

Quería aprovechar este capítulo para agradecer a algunas personas que me han ayudado en este trabajo:

- Primero de todo a José Ibáñez y Javier Calpe por confiar en mí al acogerme en su equipo, ayudarme en todo lo que necesitara y permitirme realizar este trabajo en base a un proyecto de la empresa
- A Jaime Arroyo y Rubén Sánchez por ser mis maestros en la disciplina de verificación digital y DMS, y guiarme a lo largo del proyecto
- También a todo el equipo de verificación que siempre han estado dispuestos a ayudarme y enseñarme
- Y finalmente a Rafael Gadea por acceder a ser mi tutor de universidad

Bibliografía

[1] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". En: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), págs. 1-1315. DOI: 10.1109/IEEESTD.2018.8299595.

[2] Wikipedia contributors. *SystemVerilog — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=SystemVerilog&oldid=1070279266.

[3] Cadence. *SystemVerilog for Design and Verification*. URL: https://www.cadence.com/en_US/home/training/all-courses/82143.html.

[4] "IEEE Standard for Universal Verification Methodology Language Reference Manual". En: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), págs. 1-458. DOI: 10.1109/IEEESTD.2020.9195920.

[5] Wikipedia contributors. *Universal Verification Methodology — Wikipedia, The Free Encyclopedia*. 2021. URL: https://en.wikipedia.org/w/index.php?title=Universal_Verification_Methodology&oldid=1057204069.

[6] José Juan Cerdá. "Diseño de un SAR ADC para aplicaciones multicanal con tecnología CMOS de 180nm". En: 2020. URL: https://riunet.upv.es/handle/10251/156879.

[7] Rodrigo Lorente Sanjurjo. "ANALYSIS AND DESIGN OF CONVERTERS IN MATLAB". En: 2010. URL: https://e-archivo.uc3m.es/handle/10016/11150.

[8] R. Jacob Baker. *CMOS Circuit Design, Layout, and Simulation*. 3rd. Wiley-IEEE Press, 2010. ISBN: 0470881321. URL: https://www.researchgate.net/publication/295256070_CMOS_Circuit_Design_Layout_and_Simulation_Third_Edition.

[9] Rafael Ramos Lara. "Sistemas Digitales de Instrumentación y Control". En: 2007. URL: https://upcommons.upc.edu/bitstream/handle/2117/6122/TEMA2.pdf.

[10] Walt Kester. "Which ADC Architecture Is Right for Your Application?" En: 2005. URL: https://www.analog.com/en/analog-dialogue/articles/the-right-adc-architecture.html.

[11] Dai Zhang. "Design and Evaluation of an Ultra-Low Power Successive Approximation ADC Master thesis performed in Electronic Devices". En: 2009. URL: http://www.diva-portal.org/smash/get/diva2:216811/FULLTEXT01.pdf.

[12] Walt Kester. "ADC Input Noise: The Good, The Bad, and The Ugly. Is No Noise Good Noise?" En: 2006. URL: https://www.analog.com/en/analog-dialogue/articles/adc-input-noise.html.