

UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Memoria de titulación para optar al título de
Ingeniero Civil Electrónico

Integración de herramientas de MATLAB para el diseño, validación e implementación de diseños en FPGA

Memoria de titulación para optar al título de
Ingeniero Civil Electrónico

Alumno
Juan Guillermo Arzola Garrido

Profesor Guía
Dr. Gonzalo Carvajal

Profesor Correferente
Dr. Juan Agüero

28 de octubre de 2026, Valparaíso, Chile

CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Integración de herramientas de Matlab para el diseño y validación e implementación de diseños en FPGA

Nombre del candidato(a): Juan Guillermo Arzola Garrido

Carrera / Grado: Ingeniería Civil Electrónica

Campus: Casa central **Departamento:** Electrónica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Gonzalo Carvajal, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por **(marcar una opción)**:

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 2/11/2026 **Firma:** 

Estudiante o Candidato(a):

Fecha: 30/10/2026 **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

AGRADECIMIENTOS

Quiero agradecer a todas las personas que, de una u otra forma, estuvieron presentes durante este proceso y me acompañaron a lo largo de estos años.

En primer lugar, agradezco especialmente a mi madre, por su apoyo constante, por estar presente incluso en los momentos más difíciles y por ayudarme a seguir adelante cuando las cosas no resultaban como esperaba. También quiero reconocer todo el esfuerzo que hizo durante estos años para que yo pudiera continuar con mis estudios. Fueron muchos los sacrificios, responsabilidades y preocupaciones que asumió para apoyarme, muchas veces dejando de lado sus propias cosas para ayudarme. Su apoyo fue fundamental para que pudiera llegar hasta este punto, y por eso una parte importante de este logro también le pertenece a ella.

También quiero agradecer a mis amigos, tanto a quienes me acompañaron durante la universidad como a aquellos que han estado conmigo desde antes. Gracias por las conversaciones, las risas, los momentos de distracción y, sobre todo, por estar presentes durante los períodos de mayor estrés y cansancio. Su compañía hizo mucho más llevaderos estos años y fue una parte importante de todo este proceso.

Agradezco también a mis compañeros, profesores y a todas las personas que conocí durante mi etapa universitaria y que en algún momento me brindaron su ayuda, compartieron sus conocimientos conmigo o estuvieron dispuestos a darme un consejo cuando lo necesité.

Finalmente, agradezco sinceramente a todas las personas que estuvieron presentes durante estos años y que aportaron, de distintas maneras, para que pudiera completar esta etapa. Cada ayuda, consejo, conversación y muestra de apoyo tuvo su importancia, y estoy agradecido de haber podido contar con ustedes a lo largo de este camino.

El desarrollo de este trabajo contó con el apoyo de los siguientes proyectos ANID:

- FONDECYT 1261452
- AC3E CIA250006

ÍNDICE

AGRADECIMIENTOS **I**

ÍNDICE DE FIGURAS **V**

ÍNDICE DE TABLAS **VI**

RESUMEN **VII**

Resumen VII

Lista de abreviaturas VIII

1. INTRODUCCIÓN **1**

1.1. Motivación y contexto 1

1.2. Objetivos del trabajo de memoria 4

1.3. Alcances y contribuciones 4

1.4. Organización del informe 5

2. ANTECEDENTES **6**

2.1. Generación de descripciones de hardware a partir de representaciones de alto nivel . 6

2.1.1. Síntesis de alto nivel 7

2.1.2. Model-Based Design 8

2.1.3. Desafíos y barreras de adopción en FPGA 9

2.2. Metodologías de validación progresiva en el desarrollo basado en modelos 10

2.3. Herramientas de MATLAB/Simulink para generación y verificación de código . . 13

2.4. Discusión 15

3. METODOLOGÍA Y DESARROLLO **17**

3.1. Flujo de trabajo 17

3.1.1. Construcción del flujo 17

3.1.2. Etapas del flujo 18

3.2. Aplicaciones de ejemplo 22

3.2.1. Producto punto 22

3.2.2. PID con anti-windup 23

3.2.3. Control predictivo basado en modelo con ADMM 24

3.3. Discusión 25

4. RESULTADOS EXPERIMENTALES **26**
4.1. Producto punto 26
4.2. PID con anti-windup 29
4.2.1. Conclusión del caso 31
4.3. Control predictivo basado en modelo con ADMM 32

5. CONCLUSIONES Y TRABAJO FUTURO **36**
5.1. Conclusiones 36
5.2. Trabajo futuro 37

6. ANEXOS **38**
6.1. Material complementario 38
6.2. Guía breve de reproducción 38
6.3. Datos adicionales 39

BIBLIOGRAFíA **40**

Índice de figuras

2.1. Esquema de validación progresiva del subsistema objetivo mediante etapas in-the-loop 13
2.2. Relación entre herramientas para generación y verificación de código HDL. . . . 14

3.1. Flujo para la generación y verificación de código HDL. 20
3.2. Estructura referencial de un controlador PID con anti-windup. 23

4.1. Validación numérica de la conversión a punto fijo en el caso de producto punto. . . 27
4.2. Gráfico de la salida del producto punto obtenida mediante *FPGA-in-the-Loop*, la referencia utilizada en el banco de pruebas y el error entre ellos 29
4.3. Respuesta de la planta con controlador PID y anti-windup mediante *FPGA-in-the-Loop*. 31
4.4. Comparación funcional entre la simulación en MATLAB y la implementación del MPC mediante *FPGA-in-the-Loop*, considerando posición angular, velocidad angular y señal de control. 35

Índice de tablas

4.1. Reporte de recursos estimado por HDL Coder para las versiones en punto fijo y punto flotante. 27
4.2. Resumen post-route de implementación en Vivado para las versiones en punto fijo y punto flotante. 28
4.3. Reporte de recursos estimado por HDL Coder para el PID con anti-windup. 30
4.4. Reporte de recursos estimado por HDL Coder antes y después de la reescritura orientada a HDL. 32
4.5. Recursos utilizados por la implementación generada con HDL Coder y sintetizada en Vivado. 33
4.6. Comparación de recursos utilizados entre HDL Coder y Vitis HLS para el dispositivo XC7A100T-CSG324C. 34

RESUMEN

La utilización de descripciones de alto nivel para el prototipado rápido de algoritmos es una práctica recurrente tanto en la industria como en la academia. La abstracción y las herramientas disponibles facilitan el desarrollo, la depuración y la validación. Sin embargo, la mayoría de los prototipos debe ejecutarse finalmente en sistemas físicos embebidos que no cuentan con recursos para interpretar esas descripciones de manera directa. En consecuencia, una herramienta que reduzca la brecha entre el modelado de alto nivel y la implementación embebida mejora los tiempos de desarrollo, aumenta la productividad y disminuye el error humano al sistematizar el proceso.

Esta memoria se enfoca en aportar a proyectos del ámbito académico. Para ello, explora la utilidad práctica de *HDL Coder* y *HDL Verifier*, herramientas de MATLAB para la generación automática de HDL desde scripts y diagramas de bloques dentro de un flujo de *High-Level Synthesis* (HLS). En particular, se propone y valida una metodología sistemática para usar e implementar el código generado por *HDL Coder* en distintos casos de estudio ejecutados en FPGA y contrastados mediante co-simulación y *FPGA-in-the-Loop*. Los códigos fuente y la información necesaria para reproducir y extender los experimentos están disponibles en un repositorio.

A partir de los resultados experimentales, se concluye que *HDL Coder*, apoyado por *HDL Verifier*, es una herramienta útil para generar código HDL orientado a FPGA y mantener equivalencia funcional con las descripciones originales en MATLAB.

Lista de abreviaturas

ADMM *Alternating Direction Method of Multipliers.*

DSP *Digital Signal Processor* o bloque de procesamiento digital especializado.

DUT *Device Under Test.*

FIL *FPGA-in-the-Loop.*

FPGA *Field-Programmable Gate Array.*

HDL *Hardware Description Language.*

HIL *Hardware-in-the-Loop.*

HLS *High-Level Synthesis.*

LUT *Look-Up Table.*

MBD *Model-Based Design.*

MIL *Model-in-the-Loop.*

MPC *Model Predictive Control.*

MUTS *Minimum Unconstrained Time Step.*

PID Proporcional-Integral-Derivativo.

PIL *Processor-in-the-Loop.*

RTL *Register-Transfer Level.*

SIL *Software-in-the-Loop.*

SNN *Spiking Neural Network.*

Capítulo 1

INTRODUCCIÓN

Este capítulo introduce y formaliza el problema abordado en esta memoria de título. En términos generales, el trabajo apunta a explorar y validar un flujo de trabajo para el diseño, implementación y verificación de sistemas digitales en FPGA utilizando MATLAB para la generación de descripciones en HDL a partir de algoritmos y/o modelos de alto nivel. El interés central está en determinar bajo qué condiciones ese flujo puede utilizarse de manera útil y trazable cuando el punto de partida es un modelo algorítmico y el punto de llegada es una implementación validada sobre una plataforma de lógica reconfigurable.

La exposición de este capítulo se organiza desde lo general hacia lo específico. Primero se desarrolla la motivación y el contexto técnico que justifican el uso de herramientas de más alto nivel para el diseño de hardware; luego se formula el problema específico de la memoria y se explicita qué aspectos del flujo serán estudiados experimentalmente. A continuación se delimitan los alcances y contribuciones del trabajo, y finalmente se resume la organización del informe.

1.1. Motivación y contexto

El aumento en la complejidad de los sistemas digitales y la necesidad de ejecutar algoritmos cada vez más demandantes bajo restricciones de costo y consumo energético han impulsado el uso de sistemas de cómputo heterogéneos que incorporan microprocesadores de propósito general con aceleradores de hardware especializados. En este contexto, las *Field-Programmable Gate Arrays* (FPGAs) se han consolidado como una plataforma particularmente atractiva para la implementación de sistemas de alto rendimiento y con requerimientos de tiempo real en aplicaciones como procesamiento digital de señales, control, comunicaciones y visión computacional, entre otras. A diferencia de los procesadores de propósito general, que poseen arquitecturas fijas, las FPGAs permiten al diseñador explotar paralelismo espacial y temporal, adaptando la arquitectura de hardware a las propiedades y requerimientos específicos de un algoritmo. Esta capacidad de personalización del cómputo habilita implementaciones altamente eficientes y puede derivar en mejoras significativas tanto en desempeño como en eficiencia energética [1], [2], [3].

Tradicionalmente, el desarrollo de sistemas basados en FPGA ha estado asociado al uso de lenguajes de descripción de hardware como VHDL o Verilog, trabajando típicamente a nivel *Register Transfer*

Level (RTL). El nivel RTL corresponde a una representación del sistema digital donde se describen explícitamente las transferencias de datos entre registros y las operaciones lógicas que ocurren en cada ciclo de reloj, constituyendo una abstracción cercana a la implementación física del hardware. Este nivel de descripción permite un control preciso sobre la arquitectura del hardware y la temporización del sistema, pero su correcta utilización implica una curva de aprendizaje empinada y requiere un grado avanzado de especialización en diseño de hardware digital. Además, incluso con este nivel de experticia, la implementación y verificación de arquitecturas de hardware eficientes es un proceso que involucra varias subtareas a bajo nivel de abstracción, convirtiendo el desarrollo de hardware en un proceso inherentemente detallado, tedioso, propenso a errores y con largos tiempos de desarrollo. En consecuencia, la productividad del desarrollo puede verse limitada precisamente en aquellos casos donde más se necesita explorar variantes arquitectónicas y validar decisiones de diseño con rapidez.

En respuesta a las complicaciones prácticas del proceso de desarrollo de hardware, en los últimos años han surgido metodologías orientadas a elevar el nivel de abstracción del desarrollo de hardware. Entre ellas destaca el enfoque basado en *High-Level Synthesis* (HLS), que permite generar descripciones RTL a partir de modelos algorítmicos escritos en lenguajes de más alto nivel, como C/C++ [4], [5], [6]. Este paradigma busca acercar el desarrollo de hardware al flujo de diseño de software, permitiendo que el diseñador trabaje sobre descripciones funcionales del algoritmo mientras herramientas especializadas asisten o automatizan la transformación hacia descripciones de arquitecturas de hardware con cierto grado de optimización. La relevancia de HLS no radica solo en automatizar la escritura de VHDL o Verilog, sino también en facilitar la iteración temprana, la exploración del espacio de diseño y la comparación de alternativas sin reescribir manualmente toda la arquitectura en RTL.

En una línea similar a la planteada por el paradigma HLS, el enfoque de *Model-Based Design* (MBD) propone elevar aún más el nivel de abstracción [7], [8], [9]. Bajo este paradigma, el modelo corresponde a una representación ejecutable del comportamiento del sistema, descrita a partir de ecuaciones, diagramas de bloques, máquinas de estado, flujos de datos o algoritmos de alto nivel. Es decir, el diseño se expresa inicialmente en términos funcionales, capturando qué debe hacer el sistema y cómo se relacionan sus variables internas y externas, sin requerir todavía una descripción detallada para el entorno en el cual será implementado. Posteriormente, este modelo se transforma de manera gradual en una implementación mediante etapas sucesivas de refinamiento, en las que se incorporan detalles de la plataforma objetivo, restricciones temporales, decisiones arquitectónicas y mecanismos de generación de código. De esta forma, los modelos de funcionalidad iniciales no solo sirven como representaciones preliminares del sistema, sino como especificaciones ejecutables que permiten diseñar, simular, validar e implementar sistemas dentro de un flujo unificado.

En la práctica, MATLAB constituye un entorno particularmente adecuado para aplicar el enfoque MBD, ya que integra capacidades de programación numérica, modelado por bloques y herramientas de validación dentro de una misma plataforma [10]. Además del lenguaje de scripting propio de MATLAB, el entorno incorpora herramientas como Simulink y Stateflow, que permiten describir sistemas dinámicos, lógicos y de control mediante representaciones de alto nivel, centradas principalmente en la funcionalidad del sistema. La disponibilidad de toolboxes especializados facilita, además, el desarrollo y prototipado rápido de algoritmos complejos para aplicaciones en distintas disciplinas. Posteriormente, una vez validado el modelo en este nivel funcional, herramientas adicionales de generación de código permiten reducir progresivamente el nivel de abstracción, automatizando o asistiendo la generación de código especializado para la plataforma en la cual se implementará el sistema.

Dentro del marco de MBD en MATLAB, se han introducido recientemente herramientas como *HDL Coder* y *HDL Verifier* que permiten extender este flujo hacia la implementación y verificación en hardware. En particular, *HDL Coder* permite traducir modelos descritos en MATLAB y Simulink a descripciones HDL sintetizables [11], mientras que *HDL Verifier* incorpora mecanismos de co-simulación y *FPGA-in-the-Loop* (FIL) para verificar la funcionalidad de diseños implementados en hardware [12], [13]. En la práctica, estas herramientas permiten establecer una secuencia de validación coherente entre modelo, software y hardware, recorriendo etapas como *Model-in-the-Loop*, *Software-in-the-Loop*, co-simulación HDL y FIL, manteniendo una referencia funcional común para comparar los resultados obtenidos en cada nivel. Esta continuidad facilita la exploración temprana del espacio de diseño, ya que permite evaluar distintas alternativas arquitectónicas a partir de un mismo modelo, sin necesidad de modificar directamente una descripción RTL escrita manualmente. Asimismo, contribuye a reducir el costo de depuración, al permitir identificar con mayor claridad si una discrepancia entre la respuesta del modelo de referencia y la obtenida en las etapas posteriores de co-simulación o FPGA-in-the-Loop se origina en la cuantización numérica, en la generación del código HDL o en decisiones propias de la implementación en hardware.

Desde una perspectiva de productividad de ingeniería, el flujo de diseño basado en modelos integrado en MATLAB presenta un potencial significativo para reducir la brecha entre el diseño algorítmico y la implementación de sistemas físicos. Esto resulta especialmente relevante en entornos donde los diseñadores poseen una formación sólida en modelamiento, simulación y análisis del comportamiento dinámico del sistema, pero no necesariamente en diseño RTL o arquitecturas digitales. Por ejemplo, en el área de control automático, es común que el desarrollo de un controlador parta desde modelos matemáticos de la planta, simulaciones en lazo cerrado y análisis de desempeño mediante herramientas como MATLAB y Simulink. Sin embargo, cuando dicho controlador debe implementarse en una FPGA para cumplir restricciones estrictas de tiempo de muestreo, latencia o paralelismo, los especialistas en control no siempre cuentan con experiencia suficiente para traducir directamente el algoritmo a una descripción de hardware a nivel RTL. En este escenario, un flujo MBD puede actuar como un puente entre ambos dominios, permitiendo partir desde una especificación funcional familiar para el diseñador y avanzar gradualmente hacia una implementación hardware verificable. Del mismo modo, este tipo de metodología presenta un alto potencial en contextos docentes, ya que permitiría introducir conceptos de aceleración hardware de manera progresiva, facilitando que los estudiantes comprendan primero el comportamiento algorítmico antes de abordar los detalles propios de la implementación.

Por otro lado, más allá de las promesas asociadas a la automatización del proceso de implementación y optimización de hardware, en la práctica el diseño basado en modelos presenta limitaciones relevantes. En primer lugar, no todo modelo de alto nivel puede ser trasladado directamente a hardware, ya que es necesario considerar restricciones estructurales y estilos de modelado que no siempre son evidentes para quien diseña desde una perspectiva funcional. Además, la implementación en hardware involucra múltiples etapas subyacentes, tales como síntesis, implementación física y generación del archivo de configuración, cada una con su propia complejidad. Al integrarlas dentro de un flujo automatizado, la abstracción de estos detalles puede dificultar la depuración cuando se producen errores en pasos intermedios. Asimismo, el comportamiento de las herramientas de síntesis puede ser difícil de predecir, especialmente en algoritmos complejos o cuando no se cuenta con experiencia previa en diseño digital.

A lo anterior se suma que la documentación disponible para herramientas como *HDL Coder* y *HDL*

Verifier suele centrarse en ejemplos prediseñados y relativamente simples, lo que dificulta extrapolar buenas prácticas hacia casos de uso más complejos [14], [15]. En este contexto, aunque las metodologías basadas en diseños de alto nivel poseen un alto potencial para incrementar la productividad, todavía persisten desafíos significativos relacionados con la usabilidad de las herramientas, la disponibilidad de documentación práctica y la comprensión del impacto arquitectónico de las decisiones de modelado. Estos aspectos resultan especialmente relevantes tanto en aplicaciones de ingeniería como en su incorporación en actividades docentes.

Dado el contexto presentado, el presente trabajo se enfoca en el estudio de MATLAB, Simulink y las herramientas *HDL Coder* y *HDL Verifier* como base para el desarrollo de un flujo estructurado de diseño, implementación y verificación de sistemas en FPGA a partir de modelos algorítmicos de alto nivel. A través de experimentación sistemática y análisis de casos, se busca identificar las principales consideraciones de modelado, codificación y configuración de herramientas que inciden en la generación de hardware eficiente. Sobre esta base, se presenta la documentación de un flujo de trabajo estructurado, acompañado de recomendaciones prácticas y ejemplos concretos, orientado a facilitar el uso de estas herramientas por parte de diseñadores con conocimientos básicos de desarrollo RTL. De esta manera, el trabajo busca contribuir a mejorar la accesibilidad y productividad en el diseño de sistemas digitales basados en FPGA, particularmente en escenarios donde resulta necesario transitar desde una especificación algorítmica hacia una implementación de hardware verificable.

1.2. Objetivos del trabajo de memoria

El objetivo central de este trabajo es explorar, validar y documentar una metodología para llevar algoritmos descritos en MATLAB/Simulink a implementaciones verificadas en FPGA mediante HDL Coder y HDL Verifier, apoyándose en FIL para cerrar el ciclo de verificación sobre hardware. De forma complementaria, se explorará el uso del entorno FIL de MATLAB para la verificación de módulos HDL diseñados y optimizados en forma independiente.

Los objetivos específicos son:

- Definir un flujo que parta de la especificación de alto nivel y culmine en un bitstream funcional, destacando configuraciones clave de HDL Coder (tipos de dato fijos, gestión de recursos, *pipelining* y *resource sharing*).
- Validar la equivalencia funcional entre el modelo original y el código HDL generado mediante HDL Verifier, usando co-simulación y FIL para detectar divergencias numéricas o de temporización.
- Evaluar, de forma complementaria, la capacidad de HDL Verifier para integrar y validar mediante co-simulación o FIL módulos HDL desarrollados fuera de MATLAB, utilizando el mismo entorno de referencia empleado para verificar el HDL generado con HDL Coder.
- Documentar recomendaciones, limitaciones y advertencias que no aparecen de forma explícita en la documentación base, enfocadas en facilitar la extensión a nuevos algoritmos y plataformas.

1.3. Alcances y contribuciones

Sobre la base de evaluaciones preliminares a la definición del proyecto de memoria, para este trabajo se consideraron los siguientes alcances y restricciones.

- El estudio se concentra en las herramientas HDL Coder y HDL Verifier del ecosistema MATLAB/Simulink. Si bien existen otras herramientas de síntesis de alto nivel, como Vitis HLS o Siemens Catapult, no se realiza una comparación exhaustiva entre ellas, ya que el objetivo del trabajo es caracterizar el flujo de generación y verificación disponible dentro del entorno de MathWorks. Las comparaciones con herramientas externas que se presentan en casos específicos se utilizan únicamente como referencia complementaria.
- La evaluación experimental se realiza mediante un conjunto acotado de casos de estudio, seleccionados para aplicar el flujo de generación y verificación en distintos niveles de complejidad. En consecuencia, los resultados obtenidos permiten caracterizar el comportamiento de las herramientas en los escenarios evaluados, pero no constituyen una evaluación exhaustiva de todos los algoritmos susceptibles de ser implementados en FPGA.
- El trabajo se limita a verificar la equivalencia funcional entre la descripción original en alto nivel y la versión HDL generada o integrada en FPGA. También se reportan métricas de implementación, como latencia, frecuencia y utilización de recursos, pero no se realiza una optimización exhaustiva orientada a obtener la mejor implementación posible para cada plataforma.

Las principales contribuciones de esta memoria son la organización de un flujo de trabajo para generación y verificación de HDL utilizando herramientas de MATLAB/Simulink, la aplicación de este flujo sobre casos de estudio de distinta complejidad y la documentación de las configuraciones, dificultades y resultados observados durante su uso. Además, se entrega un repositorio con modelos, scripts, archivos de configuración y ejemplos reproducibles que permiten extender el trabajo a nuevos casos de estudio [16].

1.4. Organización del informe

Este trabajo tiene un enfoque experimental y se apoya en modelos, códigos, configuraciones y resultados obtenidos durante el uso de *HDL Coder* y *HDL Verifier*. El material práctico desarrollado, junto con los archivos necesarios para reproducir los casos estudiados, se encuentra disponible en el repositorio asociado a esta memoria [16].

El resto de este documento sirve como complemento al entregable principal, y se limita a entregar información técnica adicional, decisiones de implementación y resultados experimentales que permiten contextualizar y respaldar el material disponible en el repositorio.

- El Capítulo 2 presenta los antecedentes conceptuales y tecnológicos que justifican el flujo estudiado, con énfasis en *Model-Based Design*, validación progresiva y herramientas de MATLAB para generación y verificación de HDL.
- El Capítulo 3 presenta la metodología experimental del trabajo, incluyendo el flujo propuesto, las etapas que lo componen y la justificación de los casos de estudio utilizados para aplicarlo.
- El Capítulo 4 presenta los resultados experimentales: validación funcional de los casos, comparación de métricas de implementación e interpretación metodológica de las tendencias observadas.
- El Capítulo 5 expone conclusiones y pautas para trabajos futuros, incluyendo oportunidades de optimización y extensiones a otros algoritmos o dispositivos.

Capítulo 2

ANTECEDENTES

Este capítulo reúne los antecedentes conceptuales y técnicos sobre los cuales se apoya el desarrollo de esta memoria. En particular, se introducen fundamentos asociados al diseño digital sobre FPGA, a la generación automática de descripciones de hardware desde representaciones de alto nivel, a la síntesis de alto nivel y al diseño basado en modelos. El propósito es entregar el contexto necesario para comprender la importancia de estas metodologías en flujos de desarrollo de hardware.

Además, se revisan antecedentes relacionados con llevar algoritmos hacia FPGA y con la validación progresiva de modelos e implementaciones mediante esquemas *in-the-loop*. De esta forma, el capítulo entrega las definiciones y descripciones necesarias para comprender el flujo de trabajo y los experimentos desarrollados en los capítulos posteriores.

2.1. Generación de descripciones de hardware a partir de representaciones de alto nivel

El aumento sostenido en la complejidad de los sistemas digitales modernos ha impulsado el desarrollo de metodologías que permitan elevar el nivel de abstracción durante el proceso de diseño. En el contexto de plataformas FPGA, esta necesidad resulta particularmente relevante debido a que las ventajas de desempeño, paralelismo y flexibilidad que ofrece el hardware reconfigurable suelen venir acompañadas de un flujo de desarrollo considerablemente más complejo que el asociado a plataformas basadas en software. Tradicionalmente, el diseño sobre FPGA ha requerido conocimientos especializados en arquitecturas digitales, temporización, interfaces y lenguajes de descripción de hardware, lo que introduce una barrera importante para el desarrollo rápido de sistemas especializados complejos.

En respuesta a este problema, han surgido distintos enfoques orientados a reducir la brecha entre la especificación algorítmica y la implementación física. Entre ellos destacan las metodologías basadas en *High-Level Synthesis* (HLS) y *Model-Based Design* (MBD), las cuales permiten trabajar utilizando representaciones de mayor nivel de abstracción durante etapas tempranas del diseño [4], [5], [7]. Ambos enfoques buscan aumentar la productividad, facilitar validación temprana y reducir el esfuerzo asociado al desarrollo manual de descripciones RTL. Sin embargo, aunque HLS y MBD comparten objetivos similares, abordan el problema desde perspectivas diferentes.

HLS se centra principalmente en transformar una descripción algorítmica en una arquitectura RTL sintetizable, mientras que MBD propone utilizar modelos ejecutables como eje del flujo completo de diseño, simulación, validación y posterior implementación. En proyectos orientados a FPGA, estos enfoques suelen coexistir: el modelo de alto nivel permite explorar y validar el comportamiento funcional, mientras que las herramientas de generación automática de código permiten avanzar hacia una implementación hardware concreta.

2.1.1. Síntesis de alto nivel

La síntesis de alto nivel (HLS) surge como respuesta a una limitación clásica del diseño digital, asociada al requerimiento de expresar algoritmos complejos directamente en RTL. Este nivel de descripción permite un control detallado sobre la arquitectura hardware, pero exige un alto grado de especialización y hace que la exploración de alternativas de diseño sea costosa en tiempo y esfuerzo. Frente a esta dificultad, HLS propone elevar el nivel de abstracción del proceso de diseño, permitiendo generar descripciones RTL sintetizables a partir de modelos algorítmicos de más alto nivel, típicamente expresados en lenguajes como C/C++ [4], [5].

En términos generales, el objetivo de HLS es reducir la brecha entre la especificación algorítmica y la implementación hardware final. Para ello, el diseñador puede concentrarse inicialmente en describir la funcionalidad del sistema, mientras que la herramienta de HLS automatiza o asiste en tareas estructurales propias del diseño hardware, tales como la planificación temporal, la asignación de recursos, la inserción de *pipeline*, la paralelización y la inferencia de interfaces. De esta forma, parte del esfuerzo de diseño se desplaza desde la escritura manual de RTL hacia la formulación adecuada del algoritmo, la definición de restricciones y la interpretación de los resultados de síntesis.

Este enfoque adquiere especial relevancia en plataformas FPGA, debido a la capacidad de estas para implementar arquitecturas altamente paralelas y rutas de datos dedicadas. A diferencia de microprocesadores y DSPs, donde la ejecución ocurre principalmente de manera secuencial, una FPGA permite explotar paralelismo espacial, ajustar la precisión numérica y adaptar la arquitectura al patrón de cómputo de la aplicación. Esto hace posible implementar sistemas de baja latencia y alto *throughput*, particularmente útiles en áreas como procesamiento digital de señales, visión computacional, comunicaciones y control en tiempo real [4], [5], [6].

Sin embargo, HLS no elimina la necesidad de criterio de diseño ni de conocimientos de hardware digital. Aunque la herramienta se hace cargo de especificaciones estructurales del hardware, el diseñador sigue tomando decisiones que condicionan directamente la calidad de la implementación generada. Entre ellas se encuentran la selección de la precisión numérica, la definición de latencias objetivo, la estrategia de acceso a memoria, el grado de paralelismo, la organización de interfaces y los límites de iteración. Por esta razón, el uso efectivo de HLS requiere comprender cómo las decisiones tomadas en el nivel algorítmico se traducen en recursos hardware, frecuencia de operación, latencia y consumo de área [4], [5], [6].

En plataformas reconfigurables actuales, HLS se utiliza tanto para prototipado rápido como para acelerar etapas de diseño en las que la codificación RTL manual sería costosa. Herramientas industriales como AMD Vitis HLS [17], Siemens Catapult [18] y el ecosistema de generación de código de MathWorks [11] muestran que la adopción de estas metodologías no se limita al ámbito académico, sino que forma parte de flujos reales de diseño digital. En este contexto, el valor práctico de HLS no radica únicamente en la generación automática de HDL, sino también en habilitar ciclos iterativos de

exploración arquitectónica, en los que es posible modificar restricciones, regenerar implementaciones, comparar métricas de recursos y desempeño, y volver a validar la funcionalidad del sistema.

2.1.2. Model-Based Design

El enfoque de *Model-Based Design* (MBD) propone utilizar modelos ejecutables como elemento central del flujo de desarrollo. En lugar de iniciar el diseño directamente desde una implementación específica, este paradigma parte desde modelos funcionales de alto nivel que permiten representar el comportamiento esperado del sistema. Estos modelos pueden describirse mediante ecuaciones, diagramas de bloques, máquinas de estado, flujos de datos o algoritmos, y permiten capturar la funcionalidad del sistema antes de incorporar detalles propios de la plataforma de implementación [7], [8].

Dentro de este paradigma, el modelo deja de ser únicamente una representación conceptual o preliminar, y pasa a transformarse en el principal artefacto de diseño. A partir del modelo es posible realizar simulaciones tempranas, definir pruebas funcionales, validar el comportamiento del sistema y explorar alternativas de arquitectura antes de avanzar hacia niveles más concretos de implementación. De esta forma, el modelo actúa como una especificación ejecutable que acompaña las distintas etapas del desarrollo y permite mantener una referencia común de comparación.

Una de las principales ventajas del MBD es la integración de múltiples etapas del desarrollo dentro de un mismo entorno de trabajo. Especificación, simulación, validación e implementación pueden coexistir dentro de un flujo unificado, lo que facilita la detección temprana de errores y reduce el costo asociado a iteraciones y depuración. Asimismo, este enfoque favorece la reutilización de componentes, mejora la documentación del diseño y permite mantener trazabilidad entre los requerimientos funcionales, el modelo utilizado para validarlos y los artefactos generados en etapas posteriores. Estos últimos aspectos son especialmente relevantes en sistemas sujetos a mantenimiento y evolución continua.

En aplicaciones orientadas a FPGA, el enfoque MBD suele apoyarse en herramientas gráficas y entornos de simulación que permiten modelar algoritmos, sistemas dinámicos y lógica de control mediante representaciones de alto nivel, como diagramas de bloques o descripciones algorítmicas. Una vez validado el comportamiento funcional, estos modelos pueden refinarse progresivamente hacia implementaciones hardware mediante herramientas de generación automática de código HDL. En este proceso, el modelo se utiliza tanto como especificación funcional del sistema como referencia para comparar los resultados obtenidos durante las etapas posteriores de simulación, generación de código y verificación en hardware.

A diferencia de HLS, que se centra principalmente en la transformación de algoritmos hacia descripciones RTL sintetizables, MBD abarca un proceso más amplio que considera la especificación funcional, la simulación, la validación, el refinamiento progresivo y la implementación del sistema. Ambos enfoques pueden utilizarse de manera complementaria dentro de un mismo desarrollo. En esta memoria, el modelo desarrollado en MATLAB/Simulink se utiliza como referencia funcional, mientras que *HDL Coder* se emplea para generar la descripción hardware que posteriormente se verifica mediante simulación, co-simulación o pruebas sobre FPGA.

2.1.3. Desafíos y barreras de adopción en FPGA

A pesar de las ventajas que ofrecen HLS y MBD en términos de productividad, abstracción y reducción del esfuerzo de desarrollo, su adopción práctica en entornos FPGA presenta desafíos importantes. En particular, el hecho de trabajar a partir de modelos de alto nivel no elimina la complejidad propia del diseño hardware, sino que la desplaza hacia etapas de modelado, configuración de herramientas, interpretación de reportes y validación de la implementación generada.

Uno de los principales desafíos corresponde a la brecha entre la descripción funcional del sistema y su implementación física en hardware. Los modelos utilizados en etapas tempranas suelen estar orientados a capturar el comportamiento algorítmico esperado, pero no necesariamente capturan restricciones críticas de la plataforma objetivo, tales como temporización, latencia, consumo de recursos, organización de memoria, interfaces de comunicación o frecuencia de operación. Como consecuencia, un modelo funcionalmente correcto puede derivar en una implementación ineficiente, difícil de sintetizar o poco adecuada para los requerimientos del sistema final.

A esta dificultad se suman las restricciones impuestas por las herramientas de generación automática de código. En la práctica, no cualquier descripción de alto nivel puede transformarse directamente en hardware sintetizable. Normalmente es necesario reformular los algoritmos utilizando estructuras compatibles con síntesis, tales como tipos de datos estáticos, límites explícitos de iteración, flujos de control deterministas y patrones de acceso a memoria compatibles con una implementación hardware. Por lo tanto, la transición desde una descripción funcional hacia una representación sintetizable puede introducir complejidad adicional y requerir iteraciones sucesivas de ajuste del modelo.

Otro aspecto relevante es la pérdida parcial de control explícito sobre la arquitectura de hardware generada. En un flujo tradicional basado en HDL, el diseñador define directamente elementos como paralelismo, *pipelines*, sincronización, interfaces y organización de recursos. En contraste, en enfoques basados en HLS o MBD, parte de estas decisiones queda delegada a la herramienta, la cual utiliza reglas y heurísticas que no siempre son completamente transparentes para el usuario. Esto puede provocar que cambios aparentemente menores en el modelo de alto nivel generen variaciones significativas en el hardware resultante, afectando métricas como latencia, frecuencia máxima de operación, uso de recursos o estructura de memoria.

Desde el punto de vista de la validación, también existen riesgos asociados a las diferencias entre simulación e implementación real. Efectos como cuantización numérica, inserción automática de *pipelines*, latencias multiciclo, reordenamiento de operaciones o restricciones propias de la FPGA pueden introducir discrepancias que no son evidentes durante las primeras etapas de modelado. En sistemas realimentados, como los que se utilizan en sistemas de control automático, estas diferencias pueden ser especialmente relevantes, ya que no solo afectan la coincidencia numérica de las salidas, sino también la dinámica global del sistema y su estabilidad bajo ciertas condiciones de operación.

Por estas razones, aunque HLS y MBD buscan disminuir la complejidad del desarrollo sobre FPGA, su uso efectivo continúa requiriendo conocimientos de diseño digital y arquitectura de hardware. La abstracción proporcionada por estas herramientas no reemplaza la necesidad de comprender cómo las decisiones tomadas a nivel de modelo se traducen en estructuras físicas, recursos lógicos, memorias, registros, interfaces y restricciones temporales. Sin esta comprensión, el diseñador puede generar implementaciones funcionalmente correctas, pero ineficientes, difíciles de depurar o poco robustas frente a cambios en los requerimientos.

En este contexto, uno de los principales desafíos prácticos para la adopción de estos paradigmas

y herramientas radica en la falta de metodologías sistemáticas que permitan utilizarlas de manera eficiente, trazable y reproducible. La documentación disponible suele centrarse en herramientas específicas, utilizando ejemplos relativamente simplificados y adaptados a cada entorno, mientras que las aplicaciones reales requieren integrar múltiples herramientas, restricciones de diseño y criterios de validación. Por ello, resulta necesario avanzar hacia flujos de trabajo más estructurados, que permitan aprovechar las ventajas de estos enfoques sin perder visibilidad sobre la implementación hardware resultante.

2.2. Metodologías de validación progresiva en el desarrollo basado en modelos

En un flujo de diseño basado en modelos, el modelo inicial normalmente cumple un doble propósito. Por una parte, representa la funcionalidad del componente, algoritmo o subsistema que se desea implementar en una plataforma computacional específica. Por otra, puede describir el entorno de prueba necesario para validar el comportamiento de dicho componente o algoritmo, incluyendo estímulos de entrada, modelos de planta, señales de referencia, condiciones de operación y mecanismos de visualización y comparación de resultados.

A partir de este modelo inicial, el desarrollo suele avanzar mediante etapas de validación progresiva. En cada una de ellas se reemplaza o refina la representación del subsistema bajo prueba, incorporando mayor nivel de detalle de implementación y aumentando la fidelidad respecto de la plataforma objetivo. Así, primero se valida la funcionalidad a nivel de modelo; luego se verifica si dicha funcionalidad se conserva al generar código; posteriormente, dependiendo de la plataforma objetivo, se contrasta la ejecución sobre un procesador o la implementación HDL sobre una FPGA; y finalmente se evalúa el comportamiento de la implementación sobre hardware real o en condiciones cercanas a las de operación.

En este contexto, el concepto *in-the-loop* suele utilizarse para indicar que el componente bajo prueba no se evalúa de forma aislada, sino insertado dentro de un ciclo de simulación o validación que representa, total o parcialmente, el entorno en el cual operará. De manera intuitiva, el componente recibe entradas generadas por un banco de pruebas, un modelo de planta, señales simuladas o datos reales, y sus salidas son observadas, comparadas con resultados esperados o utilizadas nuevamente dentro del mismo entorno. Por lo tanto, la validación *in-the-loop* no busca únicamente comprobar si una entrada produce una salida correcta, sino también verificar si el componente interactúa adecuadamente con el resto del sistema.

En términos generales, estas etapas forman una progresión desde representaciones abstractas hacia implementaciones cada vez más cercanas al sistema final. El elemento ubicado “dentro del ciclo” puede ser, según la etapa del flujo, un modelo funcional, código generado automáticamente, código ejecutándose en un procesador, una descripción HDL implementada sobre una FPGA o incluso hardware físico. Esta progresión permite detectar errores de manera gradual y comprobar que cada transformación del diseño preserve el comportamiento esperado a medida que se agregan detalles de implementación. La motivación principal es identificar la mayor cantidad posible de errores en etapas tempranas, ya que el costo y la dificultad de depuración aumentan a medida que el diseño avanza hacia su implementación final.

Cabe señalar que la definición exacta de estas etapas no siempre está completamente estandarizada, y distintos entornos de desarrollo pueden utilizar términos similares con alcances ligeramente diferentes.

Bajo esta lógica, en el entorno de MATLAB/Simulink se utilizan las denominaciones *Model-in-the-Loop* (MIL), *Software-in-the-Loop* (SIL), *Processor-in-the-Loop* (PIL), *FPGA-in-the-Loop* (FIL)

y *Hardware-in-the-Loop* (HIL) para referirse a distintas etapas o rutas de validación progresiva, con niveles crecientes de fidelidad respecto de la implementación final [7], [19]. En particular, PIL y FIL responden a una lógica equivalente de validación, pero no deben interpretarse necesariamente como etapas consecutivas dentro de un mismo flujo: PIL se asocia a plataformas basadas en procesador, mientras que FIL se aplica a implementaciones HDL sobre FPGA. La importancia de esta progresión es que permite detectar discrepancias funcionales, numéricas o temporales antes de que el error aparezca en pruebas tardías, cuando el costo de depuración ya es mayor.

A continuación, se describen las principales etapas y rutas de validación consideradas en este tipo de flujo.

- **Model-in-the-Loop (MIL)**: corresponde a la primera etapa de esta cadena de validación, en la cual el modelo completo se evalúa íntegramente a nivel de simulación. En esta fase, tanto el entorno de prueba como el subsistema que posteriormente será llevado a las siguientes etapas se encuentran descritos en el mismo entorno, lenguaje de descripción y nivel de abstracción. Por ello, MIL permite verificar la funcionalidad global del modelo a alto nivel y, al mismo tiempo, establecer un comportamiento esperado antes de introducir transformaciones asociadas a la implementación.

  El objetivo de esta etapa es comprobar que el comportamiento descrito por el modelo sea correcto y utilizarlo como referencia para las etapas posteriores. De este modo, cuando el subsistema sea reemplazado por código generado, una descripción HDL, una simulación externa o una implementación física, será posible contrastar si dicha nueva representación conserva el comportamiento previamente verificado. Además, al trabajar todavía en el nivel de mayor abstracción, MIL facilita la comparación de variantes, el ajuste de parámetros y la detección temprana de errores de formulación, estructura o comportamiento esperado.

- **Software-in-the-Loop (SIL)**: corresponde a la etapa en la cual el subsistema validado previamente a nivel de modelo es reemplazado por una representación en código generado, el cual se ejecuta en una estación de trabajo, pero utiliza el lenguaje o formato destinado posteriormente a la plataforma objetivo. El resto del modelo se mantiene a alto nivel sirviendo como entorno de prueba y referencia de comparación. De este modo, la validación permite comprobar si esta nueva representación del subsistema conserva las respuestas esperadas del modelo completo o si introduce diferencias asociadas al proceso de generación, compilación o ejecución del código [20].

- **Processor/FPGA-in-the-Loop (PIL/FIL)**: corresponde a una etapa de validación en la cual la representación del subsistema bajo prueba deja de ejecutarse únicamente como modelo o código simulado, y pasa a ejecutarse sobre una plataforma objetivo o equivalente, mientras el modelo completo se mantiene como entorno de prueba, estímulo y comparación. En esta configuración, la herramienta de simulación coordina el intercambio de entradas y salidas con la plataforma de ejecución, permitiendo evaluar si el comportamiento del subsistema se conserva al incorporar efectos propios de la implementación.

  La diferencia entre PIL y FIL está dada principalmente por la plataforma objetivo utilizada. En *Processor-in-the-Loop* (PIL), el subsistema corresponde a código generado que se ejecuta sobre un procesador objetivo o una plataforma equivalente. Esta etapa permite incorporar efectos asociados al compilador, a la arquitectura del procesador, a la comunicación con el entorno de simulación y a las restricciones de tiempo de ejecución [21]. En cambio, en *FPGA-in-the-Loop*

(FIL), el subsistema corresponde a una descripción HDL sintetizada e implementada sobre una FPGA. En este caso, el diseño ya se encuentra físicamente implementado en hardware, pero su operación sigue siendo coordinada desde el entorno de simulación, que conserva los estímulos, la referencia funcional y los mecanismos de observación y comparación [13].

De este modo, PIL y FIL pueden entenderse como rutas equivalentes dentro de la validación progresiva, diferenciadas por el tipo de plataforma sobre la cual se ejecuta el subsistema. PIL resulta pertinente para implementaciones basadas en procesador, mientras que FIL cumple un rol análogo para implementaciones orientadas a FPGA. En ambos casos, el objetivo es verificar que la funcionalidad validada en etapas anteriores se mantenga al aproximarse a la plataforma final de implementación.

- **Hardware-in-the-Loop (HIL)**: corresponde a la etapa en la cual una parte física del sistema se incorpora dentro del lazo de validación, mientras el resto del sistema se mantiene representado mediante un entorno de simulación en tiempo real. A diferencia de PIL o FIL, donde la interacción con la plataforma objetivo es coordinada por la herramienta de simulación para fines de comparación, en HIL el subsistema físico interactúa con señales en tiempo real y bajo condiciones más cercanas a las de operación real, típicamente incluyendo partes de hardware e interfaces reales. De este modo, HIL permite evaluar si el comportamiento previamente verificado se conserva cuando el subsistema opera como parte física del sistema dentro de un entorno de prueba temporalmente representativo [22].

Dentro del alcance de esta memoria, *FPGA-in-the-Loop* adquiere una relevancia particular, ya que permite validar diseños destinados a lógica programable en una condición intermedia entre la simulación puramente software y la integración completa del sistema. Su valor radica en que el diseño ya no se evalúa únicamente como modelo o como código HDL simulado, sino como una implementación física ejecutándose sobre una FPGA y sincronizada con el entorno de simulación. Esto permite observar efectos asociados a la temporalidad real del hardware, tales como latencias de ciclo, sincronización de entradas y salidas, restricciones de interfaz y diferencias entre el tiempo de muestreo del modelo y la operación interna del circuito.

De este modo, FIL permite reducir el riesgo de integración antes de avanzar hacia pruebas con el sistema físico completo. Al conservar el entorno de prueba en MATLAB o Simulink, es posible reutilizar estímulos, señales de referencia y criterios de comparación desarrollados en etapas previas, pero aplicándolos ahora sobre una implementación hardware real. Esta capacidad resulta especialmente útil en flujos de diseño para FPGA, donde pequeñas diferencias entre el modelo de alto nivel y la implementación sintetizada pueden afectar el comportamiento funcional, la latencia o la interacción temporal con otros bloques del sistema.

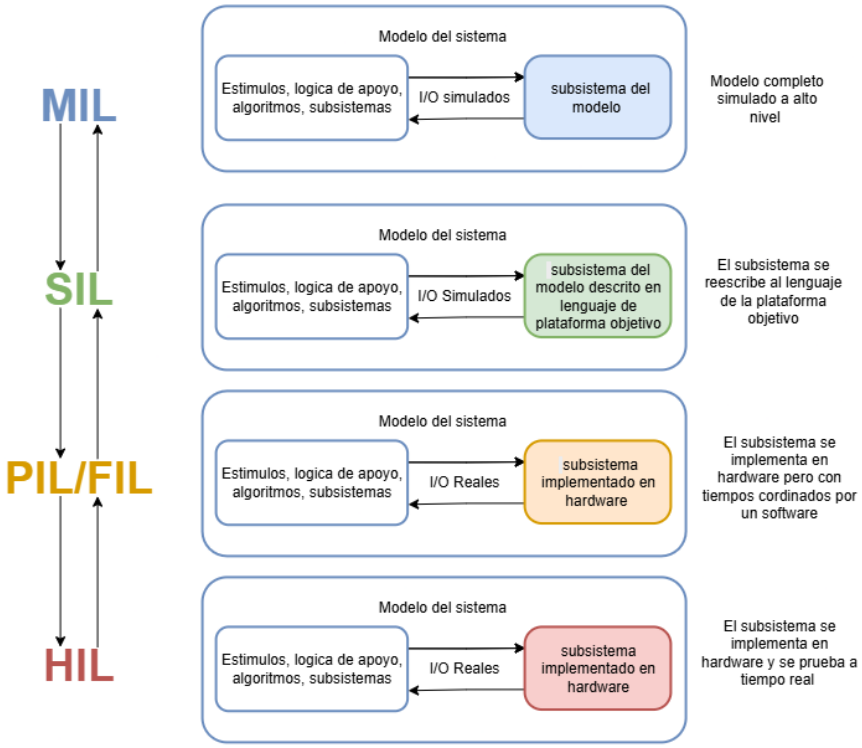


Figura 2.1: Esquema de validación progresiva del subsistema objetivo mediante etapas in-the-loop

La Figura 2.1 resume la progresión general de validación en un flujo basado en modelos. En todas las etapas se conserva un entorno de prueba encargado de generar estímulos, contener lógica de apoyo y permitir la comparación de entradas y salidas simuladas. Lo que cambia entre etapas es la representación del subsistema objetivo: inicialmente corresponde a un modelo de alto nivel, luego a código generado o a una descripción compatible con la plataforma de implementación, y posteriormente a una ejecución sobre la plataforma objetivo o sobre hardware físico.

2.3. Herramientas de MATLAB/Simulink para generación y verificación de código

Una vez descritas las etapas generales de validación progresiva, resulta necesario introducir las herramientas utilizadas para implementar este flujo. MATLAB y Simulink proporcionan el entorno base para modelar, simular y construir bancos de prueba, mientras que sus herramientas complementarias permiten realizar conversión numérica, generación automática de código y verificación de implementaciones HDL.

MATLAB y Simulink permiten formular algoritmos, construir modelos ejecutables y mantener una referencia funcional para comparar las etapas posteriores del flujo. En este contexto, el modelo de alto nivel no solo representa el comportamiento esperado del subsistema, sino que también sirve como punto de partida para la generación automática de código y para la validación progresiva de la implementación.

Fixed-Point Designer permite definir y analizar representaciones numéricas en punto fijo dentro de modelos y algoritmos. Esta herramienta resulta relevante en diseños orientados a FPGA, ya que la

selección del formato numérico influye directamente en la precisión, el rango dinámico, la utilización de recursos y la latencia de la arquitectura generada. Por ello, antes de avanzar hacia la generación de HDL, es necesario evaluar si el comportamiento del modelo se mantiene al reemplazar operaciones en punto flotante por representaciones compatibles con hardware [23].

HDL Coder permite generar descripciones RTL sintetizables, en VHDL o Verilog, a partir de modelos de Simulink o funciones de MATLAB compatibles con síntesis. Para ello, el diseño debe cumplir restricciones estructurales propias de una implementación hardware, tales como dimensiones estáticas, ciclos acotados, tipos de datos compatibles y una organización adecuada para síntesis. Además, la herramienta permite configurar aspectos de arquitectura, como inserción de *pipelines*, compartición de recursos, frecuencia objetivo, interfaces de entrada y salida, y generación de reportes de latencia y utilización de recursos [11].

Finalmente, *HDL Verifier* complementa el flujo mediante mecanismos de verificación orientados a diseños HDL. En particular, permite realizar co-simulación HDL y validación *FPGA-in-the-Loop*, donde el diseño generado se implementa en una FPGA mientras MATLAB o Simulink actúan como entorno de estímulo, observación y comparación. Esta capacidad permite contrastar el comportamiento del modelo de referencia con una implementación ejecutándose en hardware real, manteniendo trazabilidad entre el modelo inicial, el código generado y la respuesta obtenida en la plataforma objetivo [12], [13].

En conjunto, estas herramientas permiten construir un flujo de trabajo en el cual el diseño parte desde una descripción funcional de alto nivel y avanza progresivamente hacia una implementación verificable en FPGA. De esta manera, el entorno de MATLAB/Simulink sirve como base para relacionar decisiones de modelado, conversión numérica, generación de HDL y validación experimental, aspectos que serán utilizados en la metodología y en los casos de estudio presentados en los capítulos siguientes.

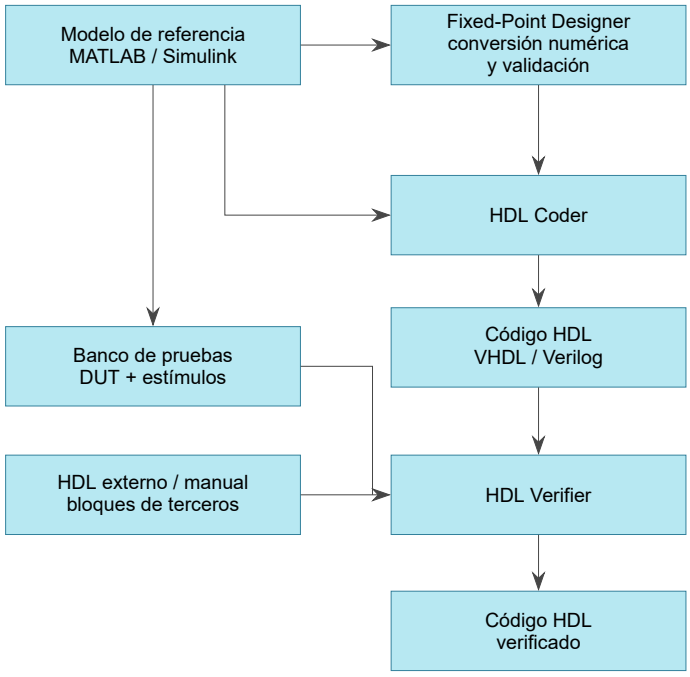


Figura 2.2: Relación entre herramientas para generación y verificación de código HDL.

La Figura 2.2 muestra cómo se relacionan las herramientas utilizadas para generar y verificar código HDL dentro del entorno de MATLAB/Simulink. El modelo de referencia constituye el punto de partida del flujo y también se utiliza para construir el banco de pruebas con el que se evaluará posteriormente la implementación. Cuando es necesario modificar la representación numérica del algoritmo, *Fixed-Point Designer* permite analizar y ajustar los tipos de datos antes de generar HDL mediante *HDL Coder*. También es posible utilizar directamente el modelo como entrada de *HDL Coder* cuando no se requiere esta conversión.

La etapa de verificación se concentra en *HDL Verifier*. El código HDL generado con *HDL Coder* puede ser evaluado utilizando el banco de pruebas definido a partir del modelo de referencia. El mismo entorno de verificación puede utilizarse además con módulos HDL desarrollados externamente, sin que estos deban haber sido generados previamente por *HDL Coder*. De esta forma, el diagrama muestra que la generación y la verificación corresponden a etapas relacionadas, pero independientes, y que *HDL Verifier* puede utilizarse tanto para diseños generados desde MATLAB/Simulink como para descripciones HDL obtenidas por otros medios.

2.4. Discusión

Los antecedentes revisados muestran que HLS y MBD pueden reducir el esfuerzo inicial de diseño al permitir que una parte importante del desarrollo comience desde modelos o algoritmos de alto nivel. Sin embargo, en una implementación sobre FPGA, esta abstracción no elimina la necesidad de tomar decisiones de diseño ni de comprender cómo el modelo será transformado en hardware. La calidad de la implementación final sigue dependiendo de la forma en que se representa el algoritmo, de las decisiones de modelado adoptadas y de la validación realizada durante el flujo de desarrollo.

En este contexto, herramientas como *HDL Coder* y *HDL Verifier* resultan relevantes porque cubren etapas complementarias del desarrollo basado en modelos. *HDL Coder* permite obtener una descripción RTL a partir de modelos en MATLAB o Simulink, pero la utilidad de esa descripción no queda garantizada solo por haber sido generada automáticamente. Es necesario comprobar que el comportamiento del diseño se conserve al pasar desde el modelo original hacia una representación HDL, considerando aspectos como la conversión numérica, la latencia introducida por la implementación y la comparación con una referencia funcional.

Por su parte, *HDL Verifier* permite realizar esa comparación mediante mecanismos de co-simulación o validación sobre FPGA. De esta manera, el modelo de referencia puede mantenerse como base de contraste frente a la implementación generada, lo que permite identificar diferencias funcionales o temporales antes de utilizar el diseño en una aplicación final.

El problema que se busca abordar en esta memoria no es la ausencia de herramientas para generar o verificar código HDL, sino la dificultad de utilizarlas de manera integrada y reproducible fuera de los ejemplos básicos. La documentación disponible describe funciones y procedimientos específicos, pero no siempre deja claro cómo relacionar las decisiones de modelado, conversión numérica, generación de HDL y verificación sobre FPGA dentro de un mismo proceso de desarrollo.

A partir de estos antecedentes, el capítulo siguiente presenta el trabajo experimental realizado con las herramientas de MATLAB orientadas a la generación y verificación de HDL. Aunque estas herramientas automatizan varias etapas del proceso, su utilización no es directa ni está exenta de dificultades. La adaptación de los modelos a estructuras sintetizables, la selección de representaciones numéricas, la configuración de las herramientas, la interpretación de los reportes y la resolución

de errores requieren tomar decisiones que suponen conocimientos tanto de diseño digital como del funcionamiento de las propias herramientas. Además, al trabajar fuera de los ejemplos proporcionados en la documentación, no siempre resulta evidente qué configuraciones utilizar, cómo modificar el modelo frente a una limitación o cómo interpretar el efecto de estas decisiones sobre el hardware generado. Por esta razón, en el capítulo siguiente se estudian estas dificultades a partir de pruebas prácticas y se organiza el conocimiento obtenido en un flujo de trabajo aplicado posteriormente a los casos de estudio.

Capítulo 3

METODOLOGÍA Y DESARROLLO

El capítulo anterior mostró que, aunque las herramientas de MathWorks permiten automatizar etapas de generación y verificación HDL, su aplicación práctica requiere adaptar los modelos, configurar las herramientas e interpretar los resultados obtenidos. A partir de este problema, en este capítulo se presenta el trabajo experimental realizado para organizar un flujo de desarrollo aplicable a diseños orientados a FPGA.

El capítulo se divide en dos partes. Primero se presenta el flujo de trabajo obtenido a partir de la exploración de las herramientas y, posteriormente, los casos de estudio utilizados para aplicarlo.

3.1. Flujo de trabajo

3.1.1. Construcción del flujo

El flujo utilizado en esta memoria se construyó a partir de la exploración práctica de las herramientas de MathWorks. El trabajo comenzó con pruebas orientadas a verificar descripciones HDL mediante MATLAB/Simulink y posteriormente se extendió hacia la generación automática de HDL y su validación.

Para ello, se revisó la documentación de las herramientas y se realizaron pruebas incrementales con distintos modelos. Estas pruebas permitieron identificar restricciones, configuraciones y dependencias que no siempre resultaban evidentes al trabajar únicamente con los ejemplos disponibles en la documentación.

Entre las dificultades identificadas se encontraron las restricciones de las funciones compatibles con generación HDL, la selección de tipos de datos adecuados y la necesidad de ajustar configuraciones de generación y validación según las características del algoritmo.

A partir de esta exploración, el proceso se organizó como una secuencia de etapas que relaciona el modelo de referencia, la configuración de las herramientas, la generación HDL y la validación de los resultados.

3.1.2. Etapas del flujo

El flujo considera como componentes principales el ajuste del modelo en MATLAB/Simulink para tipos y estructuras sintetizables, la configuración de *HDL Coder* para generación RTL, la verificación por co-simulación y *FIL* con *HDL Verifier*, y el registro de configuraciones y reportes en el repositorio. Todos los casos de estudio se someten a esta misma secuencia, no por uniformidad formal, sino porque esa repetición controlada permite atribuir diferencias observadas a decisiones de modelado e implementación, y no a cambios de procedimiento.

Las etapas que componen este flujo se describen a continuación, siguiendo el orden en que fueron aplicadas durante el desarrollo de los casos de estudio.

1. **Modelo de referencia y delimitación del subsistema objetivo**: se construye o selecciona un modelo inicial en MATLAB/Simulink que contiene tanto el algoritmo o subsistema que se desea implementar como los elementos necesarios para estimularlo, observar sus salidas y comparar su comportamiento. Dentro de este modelo se delimita el bloque o función que será llevado a hardware. El resto del modelo se mantiene como entorno de referencia durante las etapas posteriores de generación y validación.

2. **Conversión y validación numérica**: cuando el diseño parte con tipos de dato en punto flotante, se evalúa si conviene mantener esa representación o convertir el subsistema a punto fijo. En MATLAB, el tipo numérico por defecto es `double`, por lo que conservarlo puede simplificar la comparación con el modelo original, pero suele traducirse en un uso elevado de recursos al generar hardware para FPGA. Por ello, cuando corresponde, se proponen tipos de dato en punto fijo, se ajustan longitudes de palabra y fracciones, y se compara el comportamiento cuantizado respecto del modelo de referencia. Esta etapa permite observar el efecto de la cuantización antes de avanzar hacia la generación HDL y establecer un compromiso inicial entre precisión numérica y costo de implementación.

3. **Generación HDL**: se lleva a cabo la generación de código HDL a partir del subsistema objetivo, ajustando las opciones de generación según las decisiones de diseño. Estas opciones pueden incluir configuraciones asociadas a latencia, uso de recursos, frecuencia de operación, arquitectura interna y relación entre el tiempo de muestreo del modelo y el reloj interno del hardware. Esta etapa transforma la descripción de alto nivel en una representación RTL sintetizable y permite observar cómo las decisiones de modelado se reflejan en la arquitectura generada.

4. **Revisión de la implementación generada**: se analizan los reportes producidos durante la generación HDL, considerando recursos utilizados, ciclos internos, ciclos de latencia, frecuencia de operación y restricciones temporales. Esta etapa permite verificar si la arquitectura generada es coherente con las decisiones de diseño adoptadas previamente, como el uso de punto fijo o flotante, la inserción de *pipeline*, la compartición de recursos y la estructura del algoritmo. Si los resultados no son consistentes con lo esperado, el flujo vuelve a la etapa de generación HDL para ajustar sus opciones, o bien a la etapa de preparación del modelo cuando es necesario modificar u optimizar la descripción en MATLAB/Simulink.

5. **Verificación en co-simulación y FIL**: se contrasta inicialmente la salida del HDL generado con el modelo de referencia mediante co-simulación, verificando que la descripción HDL conserve el comportamiento esperado antes de avanzar hacia la ejecución en hardware. Posteriormente, se

realiza la validación mediante *FPGA-in-the-Loop* (FIL), donde el diseño se ejecuta sobre la FPGA y se comunica con MATLAB/Simulink para aplicar estímulos y observar respuestas. Si aparecen diferencias funcionales o temporales relevantes entre el modelo de referencia y la implementación HDL, el flujo vuelve a la etapa de generación HDL para ajustar opciones de arquitectura, latencia, *oversampling* u *overclocking*, o bien a la etapa de preparación del modelo cuando la discrepancia requiere modificar la descripción en MATLAB/Simulink.

Este esquema se aplica en todos los ejemplos para mantener una misma secuencia de trabajo al utilizar *HDL Coder* y *HDL Verifier*. La repetición del flujo permite documentar de forma ordenada cómo cambian los resultados cuando se modifican aspectos del modelo, los tipos de dato o las opciones de generación HDL. De esta manera, los casos de estudio no se presentan como ejercicios independientes, sino como aplicaciones de un mismo procedimiento de diseño, validación e implementación.

Durante el desarrollo, cada iteración del flujo se revisa a partir de los elementos necesarios para continuar hacia la etapa siguiente: el modelo de referencia y su entorno de prueba, la coherencia de los tipos de dato en las interfaces, la latencia introducida por la generación HDL, la relación entre el tiempo de muestreo y el reloj interno, y los reportes de recursos obtenidos. Cuando alguno de estos aspectos muestra una inconsistencia, el flujo vuelve a una etapa anterior para ajustar la descripción en MATLAB/Simulink, modificar la configuración de generación HDL o repetir la validación correspondiente.

En este sentido, la metodología propuesta debe entenderse como una guía práctica para avanzar desde un modelo de alto nivel hasta una implementación verificable en FPGA. Su valor está en ordenar las etapas del proceso, hacer explícitas las decisiones tomadas durante la generación HDL y conservar información suficiente para reproducir o extender los casos desarrollados. Sobre esta base se presentan a continuación los casos de estudio utilizados para aplicar el flujo propuesto.

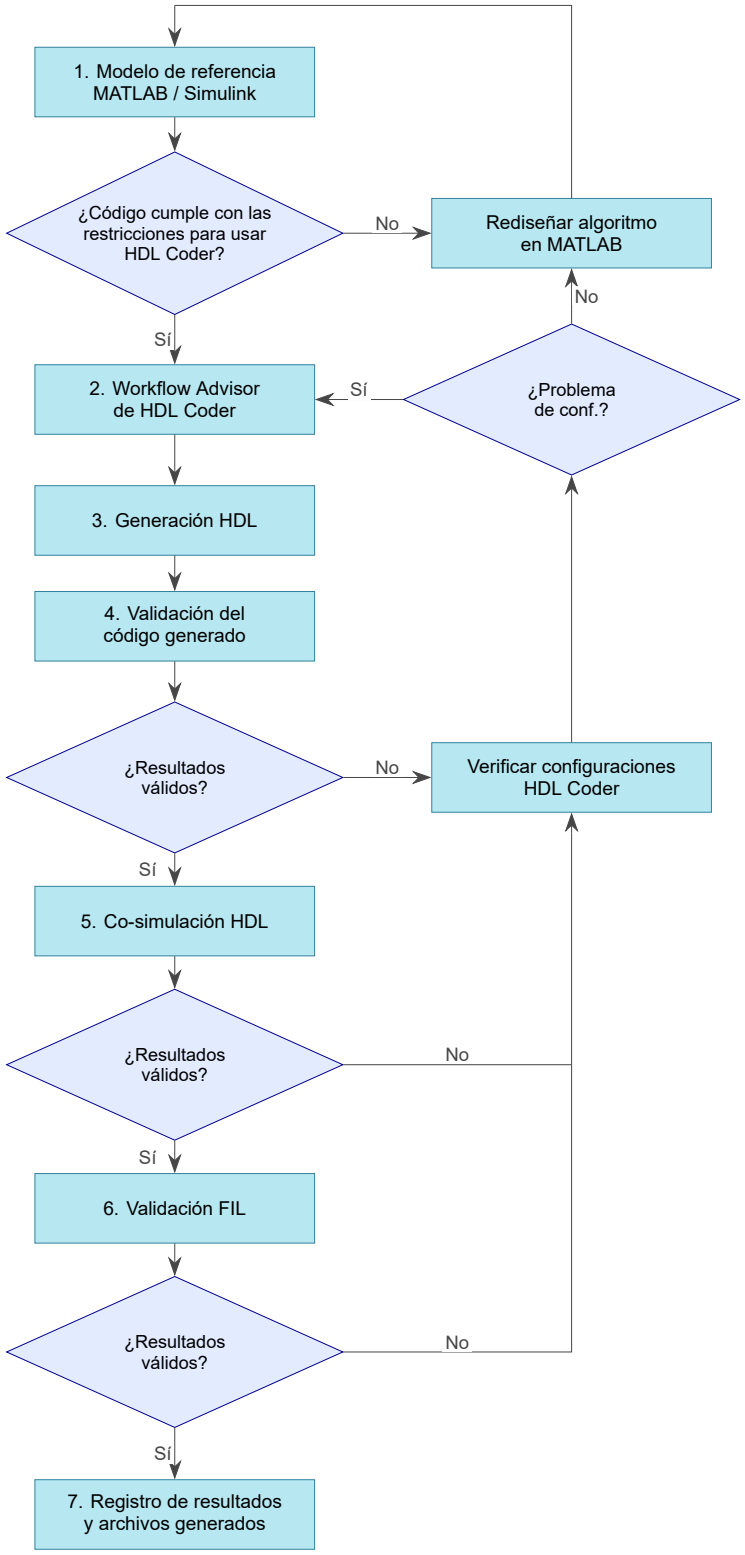


Figura 3.1: Flujo para la generación y verificación de código HDL.

La Figura 3.1 presenta una representación detallada del flujo descrito anteriormente. Mientras

que las cinco etapas generales resumen las principales actividades de generación y verificación HDL, la figura incorpora los bloques de configuración, los puntos de decisión y los retornos necesarios cuando los resultados no son satisfactorios. El proceso comienza con el modelo de referencia en MATLAB/Simulink y continúa con la adecuación del algoritmo, la configuración de *HDL Coder*, la generación y validación del código HDL, la co-simulación y la validación mediante *FPGA-in-the-Loop*. Cuando estas etapas entregan resultados satisfactorios, se registran los archivos y resultados obtenidos. A continuación, se explican los elementos representados en el diagrama y su función dentro del procedimiento.

1. **Modelo de referencia MATLAB / Simulink**: se define el modelo de referencia que se desea analizar. Este bloque entrega el punto de partida del flujo y permite establecer el comportamiento esperado para comparar las etapas posteriores. En este modelo se encuentra el subsistema o algoritmo que se desea implementar en HDL.

2. **Rediseñar algoritmo en MATLAB**: corresponde al ajuste del código o modelo cuando no cumple con las restricciones de *HDL Coder*. Esta etapa puede implicar simplificar estructuras, reemplazar funciones no compatibles, fijar tamaños de señales o reescribir partes del algoritmo para que puedan ser procesadas por la herramienta.

3. **Workflow Advisor de HDL Coder**: recibe el algoritmo o subsistema a implementar. Se decide el formato o cuantización de los tipos de datos y se configuran las opciones principales de generación HDL, tales como el objetivo de implementación, parámetros de síntesis, opciones de arquitectura y criterios de generación.

4. **Generación HDL**: corresponde a la generación de la descripción HDL del algoritmo. Este código constituye el resultado principal de la etapa de generación automática.

5. **Validación del código generado**: se revisa si el código HDL generado y los reportes entregados por la herramienta son coherentes con lo esperado. Si aparecen errores o resultados no válidos, se ajustan las configuraciones de *HDL Coder* y se repite la generación.

6. **Verificar configuraciones HDL Coder**: corresponde a la revisión de opciones de generación cuando la validación del código generado no es satisfactoria. Esta etapa puede incluir ajustes en parámetros de generación, configuración del objetivo, opciones de arquitectura, tipos de dato o condiciones usadas por la herramienta.

7. **Co-simulación HDL**: se compara la respuesta del HDL generado con la del modelo de referencia dentro del entorno de MATLAB/Simulink. Esta etapa permite detectar diferencias funcionales antes de pasar a la validación sobre hardware.

8. **Validación FIL**: se ejecuta el diseño en la FPGA mediante *FPGA-in-the-Loop*. Si los resultados son válidos, se continúa al registro final; si no, se revisan nuevamente las configuraciones o el modelo utilizado.

9. **Registro de resultados y archivos generados**: se almacenan los modelos, configuraciones, reportes, código HDL y resultados obtenidos. Este registro permite reproducir la prueba y comparar los casos de estudio bajo una misma estructura.

La validación se entiende dentro de este flujo como una etapa de comprobación progresiva entre el modelo de referencia, el HDL generado y la ejecución mediante co-simulación o *FPGA-in-the-Loop*. Su propósito es permitir que el proceso avance de forma ordenada desde la descripción de alto nivel hacia una implementación en FPGA, manteniendo una referencia común para observar diferencias funcionales, latencias introducidas y uso de recursos. Cuando aparecen discrepancias relevantes, el flujo vuelve a etapas anteriores para ajustar la descripción en MATLAB/Simulink o modificar las opciones de generación HDL.

3.2. Aplicaciones de ejemplo

En esta memoria se consideran tres casos de estudio que representan perfiles complementarios de diseño: producto punto como caso base, PID con anti-windup como caso de control en lazo cerrado y ADMM/MPC. El repositorio asociado contiene ejemplos adicionales, pero en este documento se reportan únicamente estos tres casos para mantener acotado el análisis y facilitar la comparación entre ellos.

En conjunto, los tres casos entregan una base suficiente para contrastar funcionalidad, temporalidad y costo de implementación bajo un mismo marco metodológico. En el documento, estos ejemplos se presentan desde su rol dentro de la metodología y desde los resultados necesarios para sostener la comparación, pero no se presentan en detalle el desarrollo, la configuración ni los archivos generados. Ese desarrollo práctico, junto con los modelos, bancos de prueba y materiales necesarios para reproducir o extender los experimentos, se encuentra en el repositorio asociado. Además, el repositorio incluye ejemplos adicionales y se plantea como un espacio extensible, donde pueden incorporarse nuevas implementaciones siguiendo la misma estructura de documentación. A continuación se presenta la justificación de los tres casos utilizados como referencia y el aprendizaje que aporta cada uno.

3.2.1. Producto punto

El primer caso de estudio corresponde al producto punto. Esta operación calcula un valor escalar a partir de dos vectores de igual dimensión, multiplicando sus componentes correspondientes y sumando posteriormente esos productos. Debido a su estructura aritmética directa, se utiliza como caso base para recorrer el flujo completo de generación y verificación HDL con un algoritmo simple y controlado.

El modelo considerado corresponde al producto punto entre dos vectores de dimensión fija,

$$p = \mathbf{a}^T\mathbf{b} = \sum_{i=1}^{N} a_i b_i.$$

En esta expresión, $\mathbf{a}$ y $\mathbf{b}$ representan los vectores de entrada, N corresponde al número de elementos de cada vector, a_i y b_i son sus componentes individuales, y p es el resultado escalar de la operación.

Este caso permite concentrar el análisis en las etapas básicas del flujo, como la descripción del algoritmo en MATLAB, la definición de tipos de dato, la generación de código HDL y la verificación de la implementación obtenida. Al tratarse de una operación de baja complejidad, el comportamiento esperado del algoritmo es fácil de interpretar, lo que facilita identificar diferencias asociadas al proceso de conversión numérica o a la implementación en hardware.

Además, el producto punto permite considerar una versión en punto fijo y una versión en punto flotante. La representación en punto fijo utiliza una cantidad definida de bits para representar la parte entera y fraccionaria de un número, por lo que suele ser más adecuada para implementaciones en

FPGA. En cambio, la representación en punto flotante ofrece mayor rango numérico, pero normalmente requiere más recursos de hardware.

En este trabajo, este caso se utiliza como ejemplo inicial para estudiar el comportamiento del flujo de generación y verificación HDL frente a una operación aritmética elemental. Se espera que permita analizar cómo influyen las decisiones de representación numérica en la implementación generada, antes de avanzar hacia casos con dinámica de control, realimentación o algoritmos de mayor extensión.

3.2.2. PID con anti-windup

El segundo caso de estudio corresponde a un controlador PID con anti-windup implementado dentro de un lazo de control en Simulink. El objetivo de un controlador PID es reducir el error entre una señal de referencia y la salida de un sistema, generando una acción de control que permita que la planta siga el comportamiento deseado. Para ello, combina tres acciones: la acción proporcional, que responde al error actual; la acción integral, que acumula el error en el tiempo para disminuir errores persistentes; y la acción derivativa, que considera la variación del error para anticipar cambios en la respuesta. El anti-windup se incorpora para limitar el efecto de acumulación de la acción integral cuando el actuador alcanza una condición de saturación.

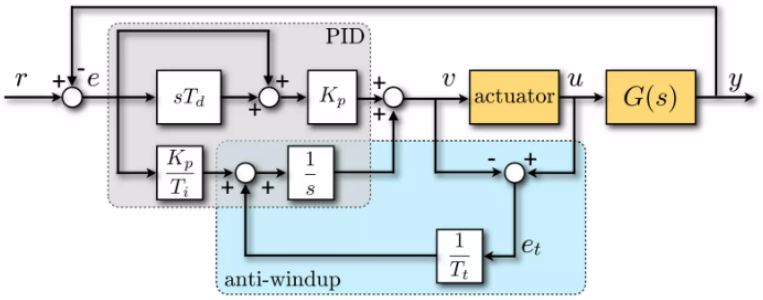


Figura 3.2: Estructura referencial de un controlador PID con anti-windup.

La Figura 3.2 muestra una estructura referencial del controlador PID con anti-windup. Aunque el diagrama utiliza notación asociada a sistemas continuos, como $G(s)$, $1/s$ y sT_d, en este trabajo se emplea principalmente para ilustrar la organización funcional del controlador. En particular, permite identificar la acción proporcional, la acción integral, la acción derivativa y la realimentación utilizada para corregir el integrador cuando existe saturación en el actuador.

En el diagrama, r corresponde a la señal de referencia, y a la salida de la planta y e al error de seguimiento, definido como la diferencia entre la referencia y la salida. El bloque PID genera la señal de control no saturada v, mientras que el actuador entrega la señal efectivamente aplicada u. La planta del sistema se representa mediante $G(s)$. Los parámetros K_p, T_i y T_d corresponden, respectivamente, a la ganancia proporcional, el tiempo integral y el tiempo derivativo. Por otra parte, e_t representa la diferencia entre la señal aplicada por el actuador y la señal calculada por el controlador, mientras que T_t define la rapidez con que dicha diferencia corrige la acción integral.

De acuerdo con la estructura referencial de la figura, la señal generada por la parte PID puede expresarse como

$$v(t) = K_p \left(e(t) + T_d \frac{de(t)}{dt} \right) + x_i(t),$$

donde $x_i(t)$ representa el estado asociado a la acción integral. La corrección anti-windup modifica la evolución de este estado mediante la realimentación de la diferencia entre la señal saturada y la señal no saturada:

$$\dot{x}_i(t) = \frac{K_p}{T_i} e(t) + \frac{1}{T_t} e_t(t),$$

con

$$e_t(t) = u(t) - v(t).$$

De esta manera, cuando el actuador no se encuentra saturado, la señal aplicada y la señal calculada por el controlador son equivalentes y el término de anti-windup no altera significativamente la acción integral. En cambio, cuando el actuador satura, la diferencia entre $u(t)$ y $v(t)$ introduce una corrección sobre el integrador, evitando que este continúe acumulando error de manera indefinida.

En este trabajo, este caso se utiliza como ejemplo de un sistema de control realimentado desarrollado en Simulink y orientado a una posterior implementación en hardware. Como requerimiento funcional, las ganancias K_p, K_i, K_d y K_{aw} se mantienen configurables desde el exterior del subsistema, de modo que puedan modificarse sin alterar la estructura interna del controlador. El modelo también incorpora saturación del actuador y la realimentación anti-windup correspondiente. A partir de este caso se estudia el comportamiento del flujo de generación y verificación HDL frente a un controlador con dinámica interna y señales realimentadas. El desarrollo práctico del ejemplo se encuentra en el repositorio [24].

Cabe señalar que la estructura presentada en la Figura 3.2 y las ecuaciones anteriores corresponden a una representación referencial en tiempo continuo del controlador PID con anti-windup. Sin embargo, para su implementación en FPGA se utiliza una formulación en tiempo discreto, debido a que estos dispositivos digitales realizan operaciones en instantes definidos por una señal de reloj y no procesan directamente operaciones continuas, como las integrales y derivadas representadas en el diagrama. Por esta razón, el controlador empleado en este caso de estudio corresponde a una versión discreta modificada del PID con anti-windup, desarrollada en Simulink. Su formulación, las modificaciones realizadas y los archivos necesarios para reproducir el ejemplo se encuentran disponibles en el repositorio [24]. El objetivo de esta memoria no es estudiar el diseño ni el proceso de discretización del controlador, sino analizar cómo llevar un modelo discreto desarrollado en Simulink hacia una implementación en FPGA mediante la generación y verificación de código HDL.

3.2.3. Control predictivo basado en modelo con ADMM

El tercer caso de estudio corresponde a un controlador predictivo basado en modelo, conocido en inglés como *Model Predictive Control* (MPC). Un MPC es una estrategia de control que utiliza un modelo del sistema para estimar cómo podría comportarse la planta en instantes futuros. A partir de esa predicción, el controlador calcula una acción de control que busca seguir una referencia y, al mismo tiempo, respetar restricciones sobre las variables del sistema o sobre la señal de control.

En este caso, el problema de control se resuelve mediante el método de dirección alternada de multiplicadores, conocido en inglés como *Alternating Direction Method of Multipliers* (ADMM). De forma general, ADMM es un método iterativo de optimización que permite resolver un problema dividiéndolo en pasos más simples que se repiten hasta obtener una solución aceptable. Para los fines de esta memoria, no se busca desarrollar en detalle la teoría del MPC ni del método ADMM, sino utilizarlos como un ejemplo de algoritmo ya existente en MATLAB que posee una estructura más elaborada que los casos anteriores.

El interés principal de este caso está en estudiar si un algoritmo previamente probado y validado en MATLAB puede transformarse en una implementación HDL y verificarse dentro del flujo propuesto, sin requerir que el usuario domine completamente la teoría interna del algoritmo. En este sentido, el foco no está puesto en diseñar un nuevo controlador predictivo, sino en analizar el proceso de transformación desde una descripción algorítmica funcional hacia una versión compatible con generación de HDL.

En comparación con los ejemplos anteriores, se espera que este caso sea más demandante desde el punto de vista de implementación, debido a que combina un mayor número de operaciones, iteraciones sucesivas y dependencias internas entre variables del algoritmo. Por esta razón, se utiliza para observar qué partes de la formulación deben adaptarse para facilitar la síntesis a hardware y reducir el uso de recursos. Estas adaptaciones no buscan modificar el comportamiento funcional del algoritmo, sino reescribir ciertas operaciones de manera más adecuada para una implementación en hardware.

En este trabajo, el caso MPC con ADMM se utiliza para estudiar los desafíos asociados a implementar en HDL un algoritmo de control más elaborado, descrito originalmente mediante un código MATLAB de mayor extensión y con más dependencias internas que los casos anteriores. Se espera que este ejemplo permita evaluar hasta qué punto el flujo de generación y verificación puede aplicarse a códigos existentes, y qué tipo de modificaciones de formulación pueden ser necesarias antes de obtener una implementación verificable. El desarrollo práctico del ejemplo, junto con los archivos utilizados para su reproducción, se encuentra disponible en el repositorio asociado a esta memoria [25].

3.3. Discusión

Este capítulo definió la metodología utilizada para aplicar un flujo de generación y verificación HDL a los casos de estudio seleccionados. La secuencia propuesta parte desde una referencia funcional en MATLAB o Simulink, continúa con la adecuación del modelo o algoritmo para generación HDL, incorpora la definición de tipos de dato y configuraciones de herramienta, y finaliza con etapas de validación mediante co-simulación o *FPGA-in-the-Loop* (FIL), según corresponda.

Bajo esta organización, los ejemplos no se plantean como demostraciones aisladas de una herramienta, sino como casos comparables dentro de un mismo marco metodológico. El producto punto permite revisar el flujo con una operación aritmética elemental; el PID con anti-windup permite estudiar un sistema de control realimentado desarrollado en Simulink; y el caso MPC con ADMM permite considerar un algoritmo MATLAB de mayor extensión y con más dependencias internas.

De esta forma, el capítulo deja establecidos los criterios que se utilizarán para interpretar los resultados posteriores: equivalencia funcional respecto de la referencia, coherencia temporal de la implementación, latencia introducida y uso de recursos en hardware. El capítulo siguiente aplica estos criterios a cada caso de estudio, con el objetivo de analizar qué aspectos del flujo se mantienen directos y qué adaptaciones pueden ser necesarias al llevar modelos o algoritmos de alto nivel hacia implementaciones compatibles con FPGA.

Capítulo 4

RESULTADOS EXPERIMENTALES

Este capítulo presenta los resultados experimentales obtenidos al aplicar la metodología definida en el Capítulo 3 a los tres casos de estudio seleccionados: producto punto, PID con anti-windup y control predictivo basado en modelo con ADMM. El análisis se centra en los resultados funcionales y cuantitativos obtenidos en cada caso, considerando la representación numérica, la utilización de recursos, el desempeño temporal y la validación de las implementaciones mediante co-simulación o *FPGA-in-the-Loop*.

El propósito del capítulo no es minimizar una métrica específica de implementación, como el uso de recursos, la latencia o el tiempo de ejecución, sino analizar los resultados obtenidos al aplicar el flujo sobre ejemplos de distinta naturaleza. En particular, interesa observar cómo las decisiones de representación numérica, estructura del modelo y forma de escritura del algoritmo influyen en la generación HDL, en la utilización de recursos y en la verificación final sobre FPGA.

4.1. Producto punto

El primer caso evaluado corresponde al producto punto. Este ejemplo se utiliza como punto de partida porque permite recorrer el flujo completo con una operación aritmética simple, de comportamiento conocido y fácil de verificar. La referencia funcional fue construida en MATLAB a partir del producto punto entre dos vectores de 10 elementos. En concordancia con lo planteado en el Capítulo 3, este caso cumple el rol de caso base, ya que permite observar las etapas principales del flujo sin que la complejidad del algoritmo o del banco de pruebas domine la interpretación de los resultados.

A partir de esta referencia, se realizó la conversión hacia una representación en punto fijo. Esta etapa es relevante porque, aunque el algoritmo es simple, la elección del formato numérico determina el rango representable, la precisión del resultado y la cantidad de recursos requeridos por la implementación en hardware. Para este caso se utilizó la notación de punto fijo (S, W, F), donde S indica si la representación posee signo, W corresponde al ancho total de palabra y F al número de bits fraccionarios. Mediante *Fixed-Point Designer* se analizaron los rangos alcanzados por las variables durante la ejecución del modelo y, a partir de estos rangos, se obtuvo una representación $(1, 30, 14)$, correspondiente a un dato con signo de 30 bits totales y 14 bits fraccionarios.

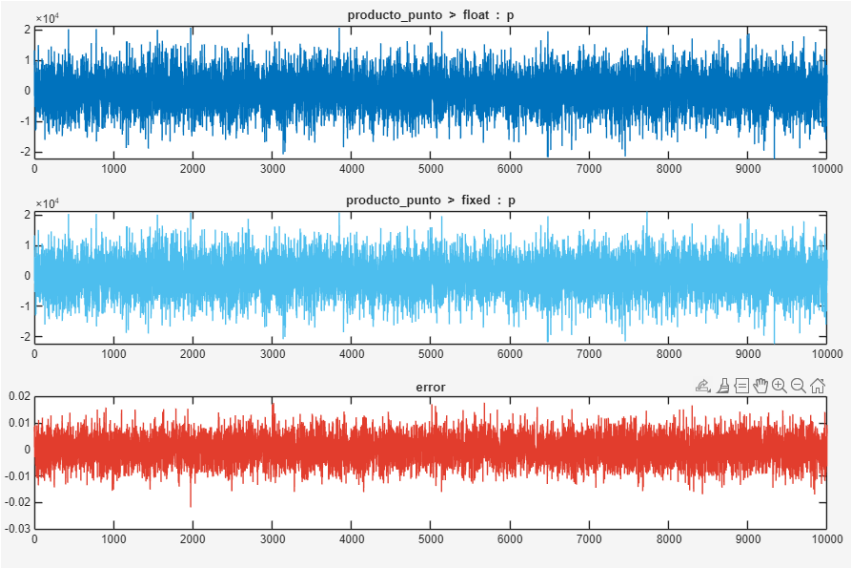


Figura 4.1: Validación numérica de la conversión a punto fijo en el caso de producto punto.

La Figura 4.1 muestra la comparación entre la referencia en punto flotante y la versión en punto fijo. La diferencia entre ambas representaciones es el calculó restando, el resultado en punto fijo al obtenido con la referencia en punto flotante. El error resultante se mantiene aproximadamente entre 0,02 y $-0{,}02$. Considerando que los resultados del producto punto alcanzan valores del orden de 10^4, la magnitud de este error es bastante menor en comparación con la señal evaluada.

Luego de validar la representación numérica, se avanzó a la generación de HDL mediante HDL Coder. En esta etapa se comparó la versión en punto fijo con una versión equivalente en punto flotante, con el objetivo de observar cómo cambia la arquitectura estimada al modificar el tipo de dato. En este punto del flujo, el reporte todavía no representa una implementación física final sobre una FPGA específica, pero sí entrega información útil para anticipar diferencias en operadores, registros, multiplexores, ancho de entradas y salidas, y latencia.

Tabla 4.1: Reporte de recursos estimado por HDL Coder para las versiones en punto fijo y punto flotante.

Recurso	**Punto fijo**	**Punto flotante**
Multipliers	1	1
Adders/Subtractors	6	23
Registers	8	269
Total Register Bits	207	5654
RAMs	0	0
Multiplexers	4	255
Shifters	0	6

La Tabla 4.1 muestra que ambas versiones mantienen un multiplicador, pero la versión en punto flotante incrementa de forma importante la cantidad de sumadores, registros, multiplexores y bits de entrada/salida. Esto se explica porque la aritmética en punto flotante requiere lógica adicional para

operar con mantisa, exponente, normalización y casos especiales. En cambio, la versión en punto fijo utiliza una representación más acotada, por lo que el hardware estimado por HDL Coder resulta más compacto.

Esta estimación inicial es útil dentro del flujo porque permite tomar decisiones antes de pasar a la implementación en Vivado. Sin embargo, el reporte de HDL Coder no reemplaza la síntesis ni la implementación sobre una FPGA concreta. Por esta razón, después de revisar la arquitectura estimada, ambas versiones fueron implementadas utilizando Vivado 2023.1, correspondiente al entorno de implementación definido para el desarrollo experimental de esta memoria. La Tabla 4.2 presenta el resumen post-route obtenido.

Tabla 4.2: Resumen post-route de implementación en Vivado para las versiones en punto fijo y punto flotante.

Métrica	**Punto fijo**	**Punto flotante**
Slices usadas	37	1096
SLICEL	23	745
SLICEM	14	351
Slice LUTs	110	3120
LUT as Logic	110	2867
LUT as Memory	0	253
LUT as Shift Register	0	253
Slice Registers	54	4010
Register as Flip Flop	54	4010
DSP48E1	1	9
CARRY4	12	255
Ciclos internos por muestra	3	3
Frecuencia máxima estimada [MHz]	122.190	135.336

La Tabla 4.2 muestra que la tendencia anticipada por HDL Coder se mantiene después de la implementación post-route: la versión en punto flotante utiliza más recursos internos que la versión en punto fijo. Estos recursos corresponden a elementos físicos limitados de la FPGA, como LUTs para lógica programable, registros para almacenamiento sincronizado, bloques DSP para operaciones aritméticas especializadas y cadenas de acarreo para sumas. HDL Coder entrega una estimación arquitectónica inicial, mientras que Vivado reporta la utilización resultante después de síntesis, ubicación y ruteo sobre una FPGA específica. Por ello, los valores finales pueden variar si cambia la familia de FPGA, el encapsulado, la herramienta o las opciones de implementación.

Además de la diferencia en recursos, la versión en punto flotante presenta una latencia inicial de 26 ciclos antes de entregar la primera salida válida. Después de esta latencia inicial, el circuito puede comenzar a procesar nuevas entradas mientras las anteriores todavía avanzan por distintas etapas internas de cálculo, lo que corresponde a un funcionamiento en *pipeline*. Por esta razón, es necesario distinguir entre latencia y *throughput*: la latencia indica cuántos ciclos transcurren hasta obtener el primer resultado, mientras que el *throughput* indica cada cuántos ciclos puede obtenerse un nuevo resultado una vez que el procesamiento se encuentra en régimen.

Finalmente, la versión en punto fijo fue validada mediante *FPGA-in-the-Loop*. La versión en punto flotante no se evaluó mediante FIL, dado su mayor uso de recursos. Esta etapa permite verificar que

el HDL generado conserve el comportamiento esperado al ejecutarse en la FPGA dentro del banco de pruebas.

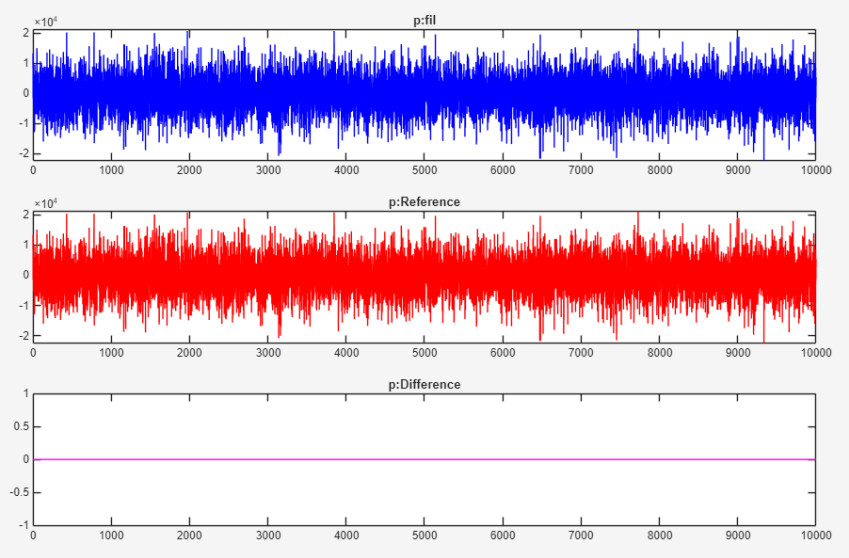


Figura 4.2: Gráfico de la salida del producto punto obtenida mediante *FPGA-in-the-Loop*, la referencia utilizada en el banco de pruebas y el error entre ellos

La Figura 4.2 muestra la comparación utilizada para validar la implementación mediante *FPGA-in-the-Loop*. El gráfico superior corresponde a la salida obtenida desde la FPGA, el gráfico central muestra la referencia utilizada en el banco de pruebas y el gráfico inferior presenta la diferencia entre ambas señales. Para las muestras evaluadas, la diferencia se mantiene en cero, lo que indica que la salida generada por la FPGA coincide con la referencia utilizada en la validación.

El caso de producto punto permitió verificar el funcionamiento básico del flujo con un algoritmo aritmético simple. A partir de este ejemplo se observa la importancia de definir una representación numérica adecuada antes de generar HDL, ya que esta decisión afecta tanto el error numérico como la cantidad de recursos utilizados. Además, la comparación entre punto fijo y punto flotante muestra que conservar una representación más general puede aumentar significativamente el costo de implementación, incluso en una operación de baja complejidad.

Un aspecto relevante de este caso es que el reporte entregado por HDL Coder permitió anticipar la tendencia observada posteriormente en Vivado. Aunque los valores reportados por MATLAB no corresponden exactamente a la utilización final de la FPGA, sí entregan una estimación temprana útil para comparar alternativas de implementación. En particular, el aumento de operadores, registros, multiplexores y bits de entrada/salida en la versión de punto flotante se reflejó después en un mayor uso de LUTs, registros, bloques DSP y otros recursos internos durante la implementación post-route.

4.2. PID con anti-windup

El segundo caso evaluado corresponde a un controlador PID con anti-windup implementado en Simulink dentro de un lazo cerrado. Como modelo de prueba se utilizó una planta genérica de segundo

orden descrita por

$$G(s) = \frac{10}{s^2 + 2s + 10}.$$

La planta no busca representar un proceso físico particular, sino proporcionar una dinámica sobre la cual evaluar el controlador, la saturación del actuador y la posterior implementación mediante HDL. A diferencia del producto punto, este ejemplo incorpora dinámica interna, saturación y realimentación, por lo que la validación incluye el análisis de la evolución temporal de la respuesta.

La referencia funcional fue construida en Simulink utilizando una planta, un actuador saturado y un controlador PID con anti-windup parametrizable. Para el ensayo se mantuvieron constantes la referencia de prueba, la planta y la configuración base del controlador, con $K_p = 2$, $K_i = 1{,}5$, $K_d = 0{,}5$, $K_{aw} = 2$ y $T_s = 10^{-4}$. Bajo estas condiciones, el modelo de Simulink permitió establecer la respuesta esperada antes de avanzar hacia la generación HDL y la validación en hardware.

A partir de esta referencia, se definió una representación en punto fijo para el controlador. Siguiendo la notación (S, W, F), `fixdt(1,32,26)` corresponde a una representación con signo de 32 bits totales y 26 bits fraccionarios. *Fixed-Point Designer* entregó este formato para las entradas y señales principales a partir de los rangos observados durante la simulación. Para las operaciones internas de suma y multiplicación, la herramienta entregó el formato `fixdt(1,36,26)`, con 36 bits totales y 26 bits fraccionarios. Además, se mantuvo saturación explícita frente a *overflow*.

Luego de validar la representación numérica, se avanzó a la generación de HDL mediante HDL Coder. En esta etapa se comparó una versión en punto fijo con una versión equivalente en punto flotante, con el objetivo de observar cómo cambia la arquitectura estimada al modificar el tipo de dato. Al igual que en el caso del producto punto, el reporte de HDL Coder entrega una primera aproximación del costo estructural antes de pasar a una implementación física sobre una FPGA específica.

Tabla 4.3: Reporte de recursos estimado por HDL Coder para el PID con anti-windup.

Recurso	**Punto fijo**	**Punto flotante**
Multipliers	6	6
Adders/Subtractors	7	79
Registers	7	949
Total 1-Bit Registers	144	18733
RAMs	0	0
Multiplexers	5	1007
I/O Bits	232	516
Static Shift operators	0	6
Dynamic Shift operators	0	12

La Tabla 4.3 muestra que ambas versiones mantienen la misma cantidad de multiplicadores, pero la versión en punto flotante incrementa de forma importante la cantidad de sumadores, registros, multiplexores y bits de entrada/salida. Este aumento se asocia a la lógica adicional necesaria para operar con números en punto flotante, especialmente en un controlador con varias operaciones aritméticas y señales internas realimentadas.

Finalmente, la versión generada fue validada mediante *FPGA-in-the-Loop*. Esta etapa corresponde al cierre del flujo aplicado al caso PID, ya que permite verificar que el controlador generado no solo pueda sintetizarse, sino que además conserve el comportamiento esperado al interactuar con la planta

dentro del lazo de simulación.

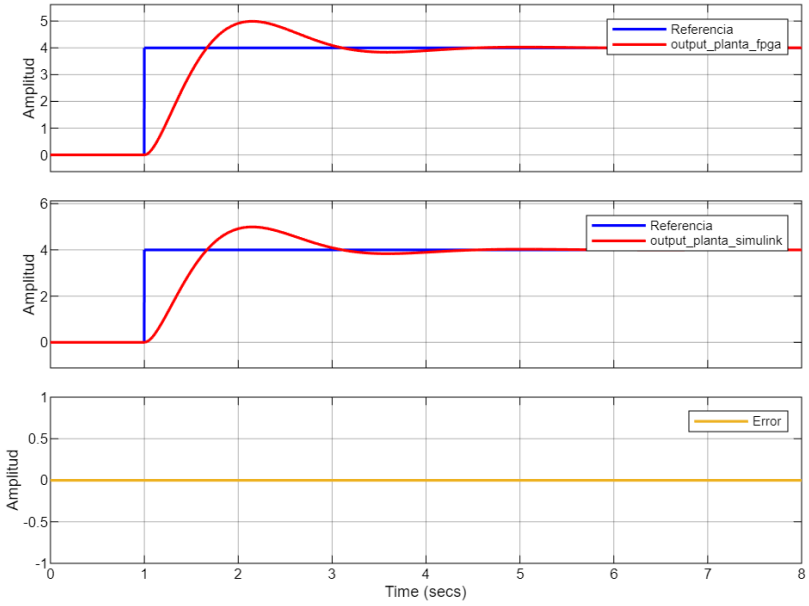


Figura 4.3: Respuesta de la planta con controlador PID y anti-windup mediante *FPGA-in-the-Loop*.

La Figura 4.3 muestra la respuesta de la planta obtenida al comparar la simulación de referencia con la ejecución mediante *FPGA-in-the-Loop*. Las respuestas se mantienen prácticamente superpuestas durante el ensayo, mostrando el error en 0. Esto indica que la implementación generada conserva el comportamiento dinámico del controlador para las condiciones evaluadas, incluyendo el efecto de la saturación y la realimentación anti-windup.

4.2.1. Conclusión del caso

El caso PID con anti-windup permitió verificar el flujo en un sistema de control realimentado. Para la configuración evaluada, la implementación en punto fijo reprodujo de forma cercana la respuesta obtenida en Simulink, con diferencias del orden de 10^{-4} durante la validación mediante *FPGA-in-the-Loop*. Este resultado corresponde específicamente a los parámetros, tipos de dato y condiciones de saturación utilizados en el ensayo.

La comparación de recursos del PID vuelve a mostrar la tendencia observada en el producto punto, con un mayor uso de recursos en la versión en punto flotante. En este caso, el aporte adicional corresponde a la validación del controlador dentro de un lazo cerrado, incorporando la dinámica de la planta, la saturación del actuador y la realimentación anti-windup.

Este caso complementa al producto punto porque introduce dinámica, realimentación y saturación del actuador. Su utilidad dentro de la memoria es mostrar que el flujo no solo puede aplicarse a operaciones aritméticas aisladas, sino también a controladores cuya respuesta debe interpretarse en el tiempo y dentro de un lazo cerrado.

4.3. Control predictivo basado en modelo con ADMM

El tercer caso evaluado corresponde a un controlador MPC resuelto mediante ADMM. En este caso, el interés no está puesto en diseñar un nuevo controlador, sino en estudiar cómo un algoritmo previamente desarrollado y validado en MATLAB puede adaptarse para generación HDL. En concordancia con lo planteado en el Capítulo 3, este ejemplo permite observar las dificultades que aparecen cuando el punto de partida es un código MATLAB más extenso, iterativo y con mayor dependencia entre operaciones internas.

La referencia funcional se mantuvo en MATLAB, utilizando el comportamiento original del solver como base de comparación. Esta referencia permitió verificar que las modificaciones posteriores no alteraran la función principal del algoritmo. A diferencia de los casos anteriores, el desafío no se concentró únicamente en la elección del tipo de dato, sino también en la forma en que el algoritmo estaba escrito y en qué tan adecuada era esa formulación para síntesis de hardware.

A partir de esta referencia, se realizó una reescritura orientada a generación HDL. Esta adaptación mantuvo la matemática del algoritmo, pero hizo explícitas decisiones necesarias para síntesis: limitar el número de iteraciones en hardware, reemplazar operaciones de alto nivel por bucles controlados y aplicar directivas de *streaming* en los tramos donde la compartición de recursos resultaba útil. El objetivo de esta modificación no fue cambiar el comportamiento funcional del controlador, sino expresar el cómputo de una forma más favorable para la herramienta de generación HDL.

Luego de realizar esta adecuación, se comparó el reporte entregado por HDL Coder antes y después de aplicar la reescritura orientada a hardware. Esta comparación permite observar cómo la forma de escribir el algoritmo en MATLAB puede modificar de manera importante la arquitectura estimada, incluso cuando la operación matemática que se busca implementar sigue siendo la misma.

Tabla 4.4: Reporte de recursos estimado por HDL Coder antes y después de la reescritura orientada a HDL.

Recurso	**Sin stream**	**Con stream**
Multipliers	4600	248
Adders/Subtractors	8954	652
Registers	52	119
Total Register Bits	728	2561
RAMs	0	0
Multiplexers	373	48
I/O Bits	51	74
Shifters	0	0

La Tabla 4.4 muestra que la reescritura orientada a HDL reduce de forma importante la cantidad estimada de multiplicadores, sumadores y multiplexores. Esto evidencia que, en algoritmos iterativos y con alta carga aritmética, la generación automática de HDL depende fuertemente de la formulación del código de entrada. En este caso, una descripción MATLAB funcional no necesariamente conduce a una arquitectura eficiente si las operaciones no se expresan de manera adecuada para hardware.

Después de obtener una versión más favorable para síntesis, el código HDL generado mediante HDL Coder fue implementado en Vivado utilizando como dispositivo objetivo la FPGA Xilinx Artix-7 XC7A100T-CSG324C de la placa Nexys A7. La Tabla 4.5 presenta el resumen de recursos obtenido

para esta implementación.

Tabla 4.5: Recursos utilizados por la implementación generada con HDL Coder y sintetizada en Vivado.

Recurso	**HDL Coder**
Slices usadas	2272
SLICEL	1551
SLICEM	721
Slice LUTs	7503
LUT as Logic	7503
LUT as Memory	0
LUT as Shift Register	0
Slice Registers	718
Register as Flip Flop	718
DSP48E1	0
CARRY4	1804
BRAM	0

La Tabla 4.5 permite contrastar la estimación previa con una implementación concreta sobre la FPGA objetivo. El diseño resultante no utiliza BRAM, pero sí ocupa una cantidad relevante de recursos, lo que es consistente con la naturaleza aritmética del algoritmo. Al igual que en el producto punto, esta etapa muestra que los reportes de HDL Coder sirven como orientación inicial, mientras que Vivado entrega una medición más cercana a la implementación física final.

Como evaluación complementaria, se comparó la implementación generada desde MATLAB mediante HDL Coder con una versión desarrollada en Vitis HLS. Esta comparación no busca establecer una superioridad general de una herramienta sobre otra, sino mostrar que distintas formas de describir y sintetizar el mismo algoritmo pueden producir compromisos diferentes entre uso de recursos, frecuencia y latencia.

Tabla 4.6: Comparación de recursos utilizados entre HDL Coder y Vitis HLS para el dispositivo XC7A100T-CSG324C.

Parámetro	**HDL Coder**	**Vitis HLS**
Slices usadas	2272	1214
SLICEL	1551	756
SLICEM	721	458
Slice LUTs	7503	2932
LUT as Logic	7503	2475
LUT as Memory	0	457
LUT as Shift Register	0	3
Slice Registers	718	2229
Register as Flip Flop	718	2229
DSP48E1	0	36
CARRY4	1804	441
Frecuencia máxima estimada	19.433 MHz	54.227 MHz
Ciclos	10	2483
Latencia	0.515 μs	45.787 μs

La Tabla 4.6 muestra que ambas herramientas producen arquitecturas con compromisos distintos. La implementación con Vitis HLS reduce el uso de LUTs y recursos CARRY4, y alcanza una frecuencia mayor, aunque utiliza 36 bloques DSP48E1 frente a los 0 utilizados por HDL Coder, pero requiere una cantidad de ciclos considerablemente superior. En cambio, la implementación generada mediante HDL Coder utiliza más recursos aritméticos, pero obtiene una latencia menor para el caso evaluado. Esta comparación refuerza que la viabilidad de una implementación no depende solo de que el HDL sea funcional, sino también del equilibrio entre recursos, frecuencia y ciclos de ejecución.

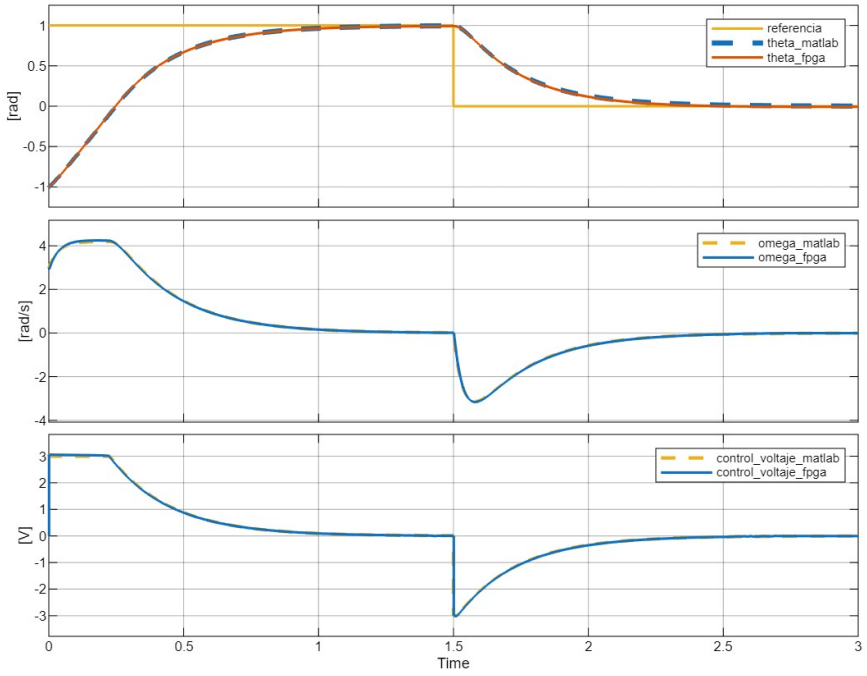


Figura 4.4: Comparación funcional entre la simulación en MATLAB y la implementación del MPC mediante *FPGA-in-the-Loop*, considerando posición angular, velocidad angular y señal de control.

La Figura 4.4 presenta la comparación funcional entre la simulación en MATLAB y la ejecución mediante *FPGA-in-the-Loop*. El gráfico superior muestra la posición angular θ junto con su referencia, mientras que los gráficos siguientes presentan la velocidad angular ω y la señal de control. En las tres señales, las respuestas obtenidas mediante MATLAB y FPGA se mantienen prácticamente superpuestas durante el ensayo, incluyendo el cambio de referencia aplicado en $t = 1{,}5$ s. Esto muestra que la implementación en FPGA reproduce el comportamiento funcional de la referencia utilizada para la comparación.

El caso MPC con ADMM permitió evaluar el flujo frente a un algoritmo MATLAB más elaborado que los ejemplos anteriores. A partir de este caso se observa que, en algoritmos iterativos y con mayor carga aritmética, la forma de escribir el código condiciona directamente la arquitectura generada. Por ello, partir desde un algoritmo funcional en MATLAB no garantiza por sí solo una implementación eficiente en HDL.

Un aspecto relevante de este caso es que la reescritura orientada a hardware redujo significativamente el costo estimado por HDL Coder sin modificar la intención funcional del algoritmo. Esto muestra que el flujo puede aplicarse a códigos existentes, pero también que en problemas de mayor complejidad puede ser necesario reformular operaciones, limitar estructuras iterativas y hacer más explícito el cómputo para que la herramienta genere una arquitectura utilizable.

Este caso complementa a los anteriores porque desplaza el foco desde la cuantización y la validación funcional hacia la adecuación estructural del algoritmo. Su utilidad dentro de la memoria es mostrar que el flujo MATLAB–HDL puede extenderse a algoritmos más complejos, siempre que el usuario revise los reportes intermedios y esté dispuesto a adaptar la formulación cuando la implementación inicial resulte demasiado costosa.

Capítulo 5

CONCLUSIONES Y TRABAJO FUTURO

5.1. Conclusiones

En esta memoria se aplicó un flujo de desarrollo basado en *HDL Coder* y *HDL Verifier* para llevar distintos algoritmos desde una referencia en MATLAB o Simulink hasta su implementación y validación en FPGA. En los casos evaluados, *FPGA-in-the-Loop* permitió comparar la ejecución del hardware generado con la referencia utilizada durante el desarrollo, verificando el comportamiento funcional de las implementaciones bajo las condiciones de prueba consideradas.

Los resultados obtenidos en los casos de estudio muestran que la calidad del diseño final depende menos de la complejidad nominal del algoritmo y más de la disciplina de modelado aplicada desde el inicio. Cuando se definen formatos numéricos coherentes, rutas de datos claramente secuenciadas y límites explícitos de latencia, el flujo de generación se vuelve estable y la síntesis converge con menor retrabajo. En cambio, cuando la especificación numérica queda implícita o se posterga la validación temporal, aparecen desajustes que luego demandan más iteraciones de ajuste.

También se observó que la relación entre recursos y rendimiento no es lineal y debe interpretarse en contexto. Diseños con menor número de operadores pueden presentar latencias elevadas si el patrón de cómputo es secuencial, mientras que arquitecturas más paralelas elevan consumo de recursos, pero reducen la espera por muestra y mejoran la capacidad de operación en tiempo real. La tensión entre paralelismo, área y latencia atraviesa todos los ejemplos y refuerza la necesidad de establecer criterios de diseño antes de optimizar casos particulares.

Un aspecto relevante es que la automatización no sustituye el criterio de ingeniería; lo potencia. Las herramientas aceleran tareas repetitivas y mejoran la trazabilidad, pero la decisión sobre qué *pipeline* usar, qué precisión numérica adoptar y qué compromisos aceptar sigue siendo del diseñador. En ese sentido, la principal contribución práctica de este trabajo no es solo el conjunto de implementaciones obtenidas, sino el marco metodológico para replicarlas y extenderlas de forma controlada.

Finalmente, la evidencia reunida respalda que el enfoque adoptado es útil para docencia, prototipado y desarrollo aplicado. El flujo permite pasar desde ejemplos básicos hasta aplicaciones más exigentes sin cambiar de paradigma de trabajo, conservando continuidad entre modelado, verificación

y despliegue. Esa continuidad facilita la formación de criterios técnicos sólidos y reduce la curva de adopción para nuevos proyectos de aceleración en FPGA.

5.2. Trabajo futuro

Como continuidad natural de este trabajo, se propone ampliar el conjunto de algoritmos evaluados con problemas de mayor dimensionalidad y con exigencias temporales estrictas. En particular, resulta pertinente incorporar variantes de control predictivo con horizontes extendidos, redes de inferencia con mayor profundidad y módulos de preprocesamiento orientados a señales ruidosas. Esa expansión permitiría caracterizar con más detalle el punto en que cada estrategia de paralelización deja de ser eficiente para una plataforma determinada.

Otra línea de mejora consiste en fortalecer la automatización del repositorio para transformar el flujo experimental en un entorno de evaluación continua. La integración de scripts de construcción, verificación y recolección de métricas permitiría regenerar tablas y figuras de manera sistemática cada vez que se modifica un modelo. Con ello, la comparación entre versiones pasaría a estar respaldada por evidencia homogénea y no por campañas manuales de prueba.

También se plantea profundizar en métodos de refinamiento numérico orientados a minimizar error sin penalizar área. El análisis de sensibilidad por bloque, la selección dinámica de escalas y la validación cruzada entre precisión y latencia pueden aportar reglas más precisas para decidir formatos *fixed-point*. Ese enfoque sería especialmente valioso en algoritmos iterativos, donde pequeños errores de cuantización se acumulan y afectan la convergencia global.

Desde la perspectiva de plataforma, es conveniente evaluar familias de FPGA con diferentes balances entre lógica programable, memoria interna y capacidad de DSP. Una campaña comparativa con criterios comunes de frecuencia, latencia y ocupación permitiría derivar guías de selección de hardware más concretas para proyectos futuros. Asimismo, la exploración de arquitecturas heterogéneas puede abrir nuevas posibilidades cuando conviene distribuir tareas entre procesamiento general y aceleración dedicada.

Por último, se propone extender la documentación metodológica con protocolos de validación por niveles, desde pruebas unitarias de bloques hasta evaluación integrada en lazo cerrado. Ese tipo de documentación no solo incrementa la reproducibilidad, sino que facilita la transferencia del flujo a otros equipos de trabajo. Con ello, la memoria deja de ser una fotografía puntual de resultados y se convierte en una base técnica para desarrollos posteriores.

Capítulo 6

ANEXOS

6.1. Material complementario

El repositorio asociado a esta memoria se encuentra disponible en [16] y concentra los modelos, scripts y reportes necesarios para reproducir los resultados presentados en los capítulos anteriores. Su organización fue pensada para que un lector pueda repetir el flujo completo sin depender de configuraciones ocultas ni de pasos manuales ambiguos. Por esta razón, cada ejemplo mantiene una estructura estable, con artefactos de entrada, configuraciones de generación HDL y evidencias de salida separadas por caso de estudio.

En particular, se incluyen:

- Modelos de cada caso de estudio, con variantes en *fixed-point* y configuraciones orientadas a implementación en FPGA.
- Scripts para ejecución del flujo, generación HDL, preparación de pruebas y configuración de *FIL*.
- Archivos de proyecto y restricciones de temporización para la plataforma objetivo empleada en esta memoria.
- Reportes de síntesis y utilización de recursos, junto con bitstreams generados.

El objetivo del material complementario es permitir que el lector replique el flujo completo y evalúe variaciones del diseño sin reestructurar los modelos base. Esta capacidad de reutilización es relevante porque permite iterar sobre un mismo ejemplo cambiando solo una variable de interés, por ejemplo la precisión numérica, el nivel de *pipeline* o la estrategia de recursos, y observar su impacto directo en las métricas finales.

6.2. Guía breve de reproducción

Para facilitar la repetición de resultados, se propone seguir una secuencia uniforme para todos los ejemplos:

1. Verificar que el modelo del caso de estudio ejecute correctamente en simulación funcional con los estímulos de prueba definidos.

2. Ejecutar la generación HDL y revisar mensajes de consistencia sobre tipos, latencias e inferencia de bloques.

3. Correr síntesis en la plataforma objetivo y registrar ocupación de multiplicadores, sumadores, registros, multiplexores y memorias.

4. Activar validación *FIL* para comparar salidas de hardware contra la referencia funcional bajo los mismos estímulos.

5. Consolidar resultados en tablas comparables entre ejemplos, manteniendo criterios unificados de reporte.

Esta secuencia evita que la validación quede reducida a una sola evidencia y promueve una lectura integral del diseño: corrección funcional, viabilidad de síntesis y comportamiento temporal en hardware. Además, la repetición disciplinada de los mismos pasos entre casos ayuda a detectar con rapidez cuándo una diferencia de resultados se debe al algoritmo y no a cambios en el procedimiento.

6.3. Datos adicionales

Para facilitar la comparación entre versiones, se mantuvieron registros de configuración del *HDL Workflow Advisor*, versiones de entorno y parámetros de hardware. Adicionalmente, se documentan las señales de prueba utilizadas en simulación y los criterios de aceptación para equivalencia funcional en la validación *FIL*.

Como apoyo a la interpretación de resultados, se incorporan también observaciones cualitativas de cada corrida: advertencias de síntesis, ajustes requeridos para cierre temporal y decisiones de arquitectura aplicadas durante la iteración. Estas anotaciones complementan las tablas numéricas, porque explican por qué dos versiones con métricas similares pueden tener costos de implementación muy distintos en la práctica.

Se recomienda mantener la trazabilidad de cambios en el repositorio y actualizar las tablas cuando se incorporen nuevos algoritmos o plataformas. Esta práctica permite construir una base de evidencia acumulativa y comparar de forma consistente el impacto de nuevas configuraciones, mejoras del modelo o refinamientos numéricos. En conjunto, los anexos no solo respaldan los resultados obtenidos, sino que definen una ruta concreta para extender el trabajo en futuras iteraciones.

Bibliografía

[1] L. Caseiro, D. Caires y A. Mendes, "Prototyping Power Electronics Systems with Zynq-Based Boards Using MATLAB/Simulink—A Complete Methodology," *Electronics*, vol. 11, n.º 7, pág. 1130, 2022. DOI: 10.3390/electronics11071130.

[2] M. Purraji, E. Zamiri y A. de Castro, "Easy and Straightforward FPGA Implementation of Model Predictive Control Using HDL Coder," *Electronics*, vol. 14, n.º 3, pág. 419, 2025. DOI: 10.3390/electronics14030419.

[3] M. Besbes, S. H. Saïd y F. M'Sahli, "HDL Coder-Based FPGA Implementation of a Continuous-Discrete Time Observer for Sensorless Induction Machine," *IETE Journal of Research*, vol. 68, n.º 3, págs. 1905-1913, 2020. DOI: 10.1080/03772063.2019.1682069.

[4] P. Coussy y A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2009. dirección: https://link.springer.com/book/10.1007/978-1-4020-8588-8.

[5] O. Gaide y P. Coussy, "High-Level Synthesis: A Decade of Progress," en *Design Automation Conference (DAC)*, 2011. DOI: 10.1145/2024724.2024734.

[6] A. Canis et al., "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," en *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011. DOI: 10.1145/1950413.1950430.

[7] MathWorks. "What Is Model-Based Design?" Visitado 10 de mar. de 2026. dirección: https://www.mathworks.com/discovery/model-based-design.html.

[8] A. Bergmann, "Benefits and Drawbacks of Model-Based Design," *KMUTNB International Journal of Applied Science and Technology*, vol. 7, n.º 3, págs. 15-19, 2014. DOI: 10.14416/j.ijast.2014.04.004.

[9] D. Akdur, V. Garousi y O. Demirörs, "A Survey on Modeling and Model-Driven Engineering Practices in the Embedded Software Industry," *Journal of Systems Architecture*, vol. 91, págs. 62-82, 2018. DOI: 10.1016/j.sysarc.2018.09.007.

[10] MathWorks. "What Is MATLAB?" Visitado 10 de mar. de 2026. dirección: https://www.mathworks.com/discovery/what-is-matlab.html.

[11] MathWorks. "HDL Coder. "dirección: https://www.mathworks.com/products/hdl-coder.html.

[12] MathWorks. "HDL Verifier. "dirección: https://www.mathworks.com/products/hdl-verifier.html.

[13] MathWorks. "FPGA-in-the-Loop Simulation," visitado 10 de mar. de 2026. dirección: https://www.mathworks.com/help/hdlverifier/fpga-in-the-loop-fil-simulation.html.

[14] R. Saralegui, A. Sanchez y A. de Castro, "Efficient Hardware-in-the-Loop Models Using Automatic Code Generation with MATLAB/Simulink," *Electronics*, vol. 12, n.º 13, pág. 2786, 2023. DOI: 10.3390/electronics12132786.

[15] R. Arjona e I. Baturone, "Using Simulink HDL Coder to Implement a Fingerprint Recognition Algorithm into an FPGA," en *2020 XIV Technologies Applied to Electronics Teaching Conference (TAEE)*, 2020, págs. 1-7. DOI: 10.1109/TAEE46915.2020.9163790.

[16] llillo2. "memoria_fpga_design_matlab," visitado 9 de feb. de 2026. dirección: https://github.com/llillo2/memoria_fpga_design_matlab.

[17] AMD Xilinx. "Vitis High-Level Synthesis. "dirección: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[18] Siemens EDA. "Catapult High-Level Synthesis. "dirección: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[19] J. Nibert, M. E. Herniter y Z. Chambers, "Model-Based System Design for MIL, SIL, and HIL," *World Electric Vehicle Journal*, vol. 5, n.º 4, págs. 1121-1130, 2012. DOI: 10.3390/wevj5041121.

[20] MathWorks. "Software-in-the-Loop Simulation," visitado 26 de mar. de 2026. dirección: https://www.mathworks.com/help/ecoder/software-in-the-loop-sil-simulation.html.

[21] MathWorks. "Processor-in-the-Loop Simulation," visitado 26 de mar. de 2026. dirección: https://www.mathworks.com/help/ecoder/processor-in-the-loop.html.

[22] MathWorks. "What Is Hardware-in-the-Loop (HIL)?" Visitado 26 de mar. de 2026. dirección: https://www.mathworks.com/discovery/hardware-in-the-loop-hil.html.

[23] MathWorks. "Fixed-Point Designer. "dirección: https://www.mathworks.com/products/fixed-point-designer.html.

[24] llillo2. "memoria_fpga_design_matlab: ejemplo PID con anti-windup," visitado 6 de jun. de 2026. dirección: https://github.com/llillo2/memoria_fpga_design_matlab/tree/main/examples/PIDaw.

[25] llillo2. "memoria_fpga_design_matlab: ejemplo MPC," visitado 6 de jun. de 2026. dirección: https://github.com/llillo2/memoria_fpga_design_matlab/tree/main/examples/mpc.