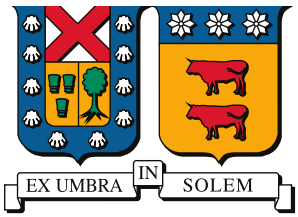


UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO DE ELECTRÓNICA

VALPARAÍSO - CHILE

"IMPLEMENTACIÓN DE MODELOS DE ELECTRÓNICA DE POTENCIA EN FPGAS CON TÉCNICAS DE GENERACIÓN DE CÓDIGO"

CRISTIAN ANDRÉS GONZÁLEZ BUSTOS

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO

Profesor guía
Dr. GONZALO ARIEL CARRASCO REYES

Profesores correferentes
Dr. GONZALO CARVAJAL BARRERA
Dr. PABLO LEZANA ILLESCA

Noviembre - 2024

Si no funciona como quieres, no es un fracaso, sólo significa que has aprendido algo. Y, mientras estés aprendiendo, no habrás fracasado.
-Bob Ross

A mi familia, por su apoyo, sin el cual no sé qué habría sido de mí

AGRADECIMIENTOS

EN PRIMERÍSIMO primer lugar, quisiera agradecer a toda mi familia cercana. A mis papás Roberto y Patricia, por ser fuentes infinitas de amor, apoyo, aliento, sabiduría y motivación. A mi hermana Daniela, por ser una constante en mi vida diaria y apoyarme y recordarme siempre que le fuera posible y en todo ámbito. A mi hermana Karen, quien activamente se preocupó por mi bienestar todos estos años e insistió siempre en recalcar mis capacidades. Los amo mucho y no los cambiaría por nada.

A Dani, Julio, Yangzi y Andrés, a Cata, Feña, Edu y Vale, y a todos mis amigos de la universidad, con quienes compartí los mejores años de mi vida. No tienen idea cuánto les agradezco haberlos conocido a todos ustedes.

A mi profesor guía, Gonzalo Carrasco, por su increíble apoyo y por la enorme cantidad de cosas que he aprendido de él. Me quedaría corto expresando en palabras lo agradecido que estoy de haber trabajado con alguien tan increíblemente interesante como usted.

A mis amigos de Reznor, a quienes conozco desde que soy niño y que nunca me han dejado de lado. El cariño que les tengo trasciende cualquier distancia, y espero sigamos siempre conectados.

A todas las personas con quienes he tenido el agrado de trabajar y conectar a lo largo de estos años de universidad.

Cristian Andrés González Bustos

RESUMEN

LA SIMULACIÓN *hardware-in-the-loop* basada en FPGA es una poderosa herramienta al momento de diseñar y testear prototipos de sistemas de electrónica de potencia. Sin embargo, su adopción se ve disminuida debido a la alta complejidad que conlleva modelar estos sistemas en un lenguaje de programación o de descripción de hardware (HDL).

En esta memoria se evalúan múltiples métodos de generación automática de código HDL, en búsqueda de un enfoque tal que el desarrollador solo deba concentrarse en el diseño del sistema mismo, y que genere código suficientemente optimizado. Se realizan múltiples simulaciones para una variedad de sistemas dinámicos distintos, con el fin de validar estos métodos.

Al encontrar un *toolchain* y flujo de trabajo adecuados, el método seleccionado se aplica de forma práctica, implementando un modelo de buck y boost converter en la plataforma BRAIn del AC3E. A nivel de simulación, los sistemas generados cumplen tanto con el comportamiento esperado como con los requisitos de optimización que se buscaban, y se logran integrar exitosamente los modelos en BRAIn a nivel de hardware.

El trabajo presentado a lo largo del documento se concreta en la forma de una *application note* que detalla el proceso de implementación de un circuito de electrónica de potencia en FPGAs para simulación *hardware-in-the-loop*.

Palabras Claves

Sistemas en tiempo real, FPGA, electrónica de potencia, hardware in the loop, sistemas dinámicos, high level synthesis, generación de código

ABSTRACT

FPGA-BASED hardware-in-the-loop simulation offers a powerful tool for designing and prototyping power electronic systems. However, the adoption of this technique is diminished by the high level of complexity that is involved in modelling these systems in a programming or hardware-description language (HDL).

In this thesis, multiple HDL code generation methods are evaluated, in search of an approach that will allow developers to focus primarily in the design of the system itself, and that will generate sufficiently optimized code. Several simulations for a variety of dynamic systems are performed, in order to assess these methods.

Once a suitable toolchain and workflow is found, the method is applied in a practical scenario, where a buck and a boost converter is to be implemented into the BRAIn platform, developed by the AC3E. At a simulation level, the implemented systems meet both the desired behaviour and optimization requirements, and the models are successfully integrated in BRAIn at the hardware level.

The work presented throughout this document takes the form of an application note that details the full process of implementing a power electronic system in FPGAs for hardware-in-the-loop simulation.

Keywords

Real-time systems, FPGA, power electronics, hardware in the loop, dynamic systems, high level synthesis, code generation

ÍNDICE

AGRADECIMIENTOS **I**

RESUMEN **II**

ABSTRACT **III**

ÍNDICE DE FIGURAS **IX**

ÍNDICE DE TABLAS **XII**

GLOSARIO **XIV**

1. INTRODUCCIÓN **1**

1.1. Contexto esencial 2

1.2. Trabajo por desarrollar 2

1.3. Estado del arte 3

1.3.1. Herramientas de software 3

1.3.2. Métodos de modelado de sistemas de electrónica 4

1.3.3. Aplicaciones en simulación HIL de sistemas de potencia . . 4

1.4. Objetivos 6

1.5. Alcances y contribuciones 6

1.6. Estructura del documento 7

2. MARCO CONTEXTUAL **9**

2.1. Electrónica de potencia 9

2.2. Electrónica digital y FPGAs 10
2.3. Tipos de plataformas digitales programables existentes 11
2.4. Plataforma de trabajo: "BRAIn" 13
2.4.1. Desafíos y consideraciones 14

3. EVALUACIÓN DE ALTERNATIVAS DE GENERACIÓN DE CÓDIGO 16

3.1. Metodología de testeo 16
3.2. "High Level Synthesis" y Vitis HLS 18
3.3. Implementación inicial de algoritmo en C 18
3.3.1. Oscilador dual de tipo biquad 18
3.3.1.1. Simulación previa 20
3.3.1.2. Reporte de síntesis 20
3.3.1.3. Simulación RTL 23
3.4. MATLAB Simulink y Embedded Coder 24
3.4.1. Oscilador biquad singular en Simulink 25
3.4.1.1. Simulación previa 25
3.4.1.2. Generación de código C para Vitis y segunda simulación 26
3.4.1.3. Reporte de síntesis 27
3.4.1.4. Simulación RTL 28
3.5. Exploración de "ORTiS" 29
3.5.1. Circuito RLC en ORTiS 29
3.5.1.1. Simulación previa 30
3.5.1.2. Generación de código C y segunda simulación . . 31
3.5.1.3. Reporte de síntesis 32
3.5.1.4. Simulación RTL 32
3.5.2. Inversor trifásico en ORTiS 33
3.5.2.1. Reporte de síntesis 34
3.6. Simscape y HDL Coder 35
3.6.1. Circuito RLC en Simscape 35
3.6.1.1. Simulación previa 36
3.6.1.2. Reporte de generación de código 37
3.6.1.3. Simulación RTL y reporte de Vivado 38

3.6.2. Rectificador de media onda en Simscape 38
3.6.2.1. Simulación previa 39
3.6.2.2. Reporte de generación de código 40
3.6.2.3. Simulación RTL y reporte de Vivado 41
3.6.3. Puente rectificador de onda completa en Simscape 42
3.6.3.1. Simulación previa 42
3.6.3.2. Reporte de generación de código 43
3.6.3.3. Simulación RTL y reporte de Vivado 43
3.7. Conclusiones 44

4. MODELO DE CONVERTIDORES DE ELECTRÓNICA DE POTENCIA EN BRAIN 46
4.1. Buck y Boost Converter 47
4.1.1. Buck Converter 47
4.1.1.1. Parámetros del buck converter 48
4.1.2. Boost Converter 49
4.1.2.1. Parámetros del boost converter 50
4.2. Hardware de BRAIn para simulaciones en tiempo real y HIL . . . 50
4.2.1. Especificaciones de la XC7A35T 51
4.2.2. Integración de módulos y comunicación DSP-FPGA . . . 52
4.3. Decisiones de diseño y restricciones 53
4.4. Flujo de Simscape a código HDL 55
4.4.1. Diseño del sistema de electrónica de potencia en Simscape . 56
4.4.2. Conversión del sistema con Simscape-HDL Workflow Advisor 57
4.4.3. Generación de código HDL 58
4.4.4. Modelo de verificación con HDL Verifier 59
4.4.5. Integración a BRAIn 61
4.5. Visión general de la Application Note 61
4.5.1. Enlace a la Application Note 62

5. RESULTADOS EXPERIMENTALES 63
5.1. Modelado y simulaciones 63
5.1.1. Buck Converter 63
5.1.2. Boost Converter 66
5.2. Reportes de implementación 69

5.2.1. Buck Converter 70
5.2.2. Boost Converter 71
5.2.3. Comparación de resultados de implementación 72
5.3. Ejecución en hardware 73
5.3.1. Prueba de conversión de 24V a 48V 75
5.3.2. Comportamiento dinámico al cambiar voltaje de entrada . . 76

6. CONCLUSIONES Y TRABAJO FUTURO 78
6.1. Conclusiones 78
6.1.1. Análisis de los resultados obtenidos 78
6.1.2. Trabajo realizado 79
6.2. Trabajo Futuro 79

A. USO DE HERRAMIENTAS DE SOFTWARE 81
A.1. Vitis HLS 81
A.1.1. Configuración de nuevo proyecto 82
A.1.2. Flow de Vitis 85
A.1.3. Comentarios adicionales 85
A.2. Embedded Coder 86
A.3. ORTiS 89
A.3.1. Uso y formato de netlist 90
A.3.2. Código generado por la herramienta 90
A.3.3. Función wrapper 91
A.3.4. Lista de componentes 92

B. CÓDIGOS, SCRIPTS, Y DIAGRAMAS 93
B.1. Oscilador biquad dual 93
B.1.1. Punto flotante 93
B.1.2. Punto fijo 97
B.2. Oscilador biquad singular con Embedded Coder 98
B.2.1. Diagrama de Simulink 98
B.2.2. Post-generación de código C 99
B.3. RLC en ORTiS 101
B.3.1. Definición de netlist 101
B.3.2. Post-generación de código C 102

B.4. Inversor trifásico en ORTiS 104
B.5. Simscape y HDL Coder 105

BIBLIOGRAFÍA **109**

Índice de figuras

2.1. Un sistema de electrónica de potencia 10
2.2. Tarjeta de control BRAIn 14

3.1. Flujo de trabajo para testear cada método de generación de código . 17
3.2. Diagrama de bloques del filtro IIR y del oscilador biquad. 19
3.3. Resultado de simulación de osciladores en C, compilado desde el IDE de Vitis 20
3.4. Simulación RTL del oscilador biquad con acercamiento 23
3.5. Simulación RTL con visualización análoga 24
3.6. Reacción del módulo ante nuevos impulsos en la entrada 24
3.7. Sistema del oscilador en Simulink. 25
3.8. Simulación del oscilador en Simulink (vista de Scope). 26
3.9. Resultado de simulación de oscilador diseñado en Simulink. 27
3.10. Simulación RTL con visualización análoga 29
3.11. Circuito RLC a implementar 29
3.12. Resultado de simulación de circuito RLC en LTSpice 30
3.13. Resultado de simulación de circuito RLC de ORTiS implementado en C++, compilado desde el IDE de Vitis 31
3.14. Simulación RTL con visualización análoga 33
3.15. Sistema de inversor trifásico en ORTiS 33
3.16. Circuito RLC de Simscape en el entorno Simulink 36
3.17. Simulación de circuito RLC de Simscape 37
3.18. Reporte de uso de recursos de RLC de Simscape 37
3.19. Reporte de reloj de RLC de Simscape 37
3.20. Uso de recursos de circuito RLC de Simscape en Vivado 38

3.21. Simulación RTL de circuito RLC de Simscape 39
3.22. Rectificador de media onda en Simscape 39
3.23. Simulación previa del rectificador de media onda de Simscape . . 40
3.24. Reporte de reloj de rectificador de Simscape 40
3.25. Reporte de uso de recursos de rectificador de Simscape 40
3.26. Uso de recursos de rectificador de Simscape en Vivado 41
3.27. Simulación RTL de rectificador de media onda 41
3.28. Puente rectificador de onda completa en Simscape 42
3.29. Simulación previa del puente rectificador de Simscape 42
3.30. Reporte de reloj de puente rectificador de Simscape 43
3.31. Reporte de uso de recursos de puente rectificador de Simscape . . 43
3.32. Uso de recursos de puente rectificador de Simscape en Vivado . . 44
3.33. Simulación RTL de puente rectificador de onda completa 44

4.1. Buck Converter 48
4.2. Boost Converter 49
4.3. Diagrama de bloques de la plataforma BRAIn 52
4.4. Flujo de Simscape 55
4.5. Pasos del diseño del circuito en Simscape 56
4.6. Circuito del Boost Converter en Simscape con entradas y salidas externas. 56
4.7. Pasos de la conversión del sistema Simscape a Simulink 57
4.8. Modelo de implementación generado por SSC-HDL Workflow Advisor 58
4.9. Pasos de la generación del código HDL 58
4.10. Pasos de la verificación del modelo RTL 59
4.11. Testbench para el módulo HDL, con HDL Verifier en Simulink. . . 60
4.12. Paso de integración a BRAIn 61

5.1. Circuito del buck converter en Simscape. Convierte 12V a 5V. . . 64
5.2. Simulación del buck converter en Simscape. 64
5.3. Simulación del modelo de implementación del buck converter . . . 64
5.4. Simulación RTL en Vivado del módulo del buck converter 65
5.5. Prueba de comportamiento dinámico del buck converter en Vivado. 65
5.6. Circuito del boost converter en Simscape. Convierte 24V a 48V. . . 66
5.7. Simulación. En orden de arriba a abajo: V_{diodo}, $I_{inductor}$, V_{carga} . . 66

5.8. Simulación del modelo de implementación del boost converter . . 67
5.9. Simulación RTL en Vivado del módulo del boost converter . . . 67
5.10. Prueba de comportamiento dinámico del boost converter en Vivado. 68
5.11. Uso de recursos por defecto del firmware de BRAIn. 69
5.12. Reporte de timing por defecto del firmware de BRAIn. 69
5.13. Uso de recursos del firmware de BRAIn con buck converter. . . . 70
5.14. Reporte de timing del firmware de BRAIn con buck converter. . . 70
5.15. Uso de recursos del firmware de BRAIn con boost converter. . . . 71
5.16. Reporte de timing del firmware de BRAIn con boost converter. . . 71
5.17. Pmod DA4 (Revision A) de Digilent 73
5.18. *Setup* de prueba del boost converter siendo ejecutado en BRAIn . . 74
5.19. Conversión de 24 a 48V del modelo del boost converter en BRAIn . 75
5.20. Comportamiento dinámico del boost converter en BRAIn 76

A.1. Ventana de creación de nuevo componente HLS: Hardware . . . 82
A.2. Ventana de creación de nuevo componente HLS: Settings 83
A.3. Jerarquía del componente recién creado. 83
A.4. Contenidos de `resonator3.cpp`. 84
A.5. Archivo `hls_config.cfg`. 84
A.6. Vitis Flow 85
A.7. Carga de Embedded Coder en Simulink 86
A.8. Selección de subsistema de Simulink 87
A.9. Código generado por Embedded Coder 87

B.1. Diagrama de bloques del oscilador biquad de 130.8 Hz 98
B.2. Circuito RLC 101
B.3. Inversor trifásico 104

Índice de tablas

3.1. Parámetros para los dos osciladores biquad 20
3.2. Estimación post-síntesis de timing del oscilador biquad para punto flotante 21
3.3. Estimación post-síntesis de rendimiento del oscilador biquad para punto flotante 21
3.4. Estimación post-síntesis de uso de recursos de FPGA usados por el oscilador biquad para punto flotante 21
3.5. Estimación post-síntesis de timing del oscilador biquad para punto fijo 22
3.6. Estimación post-síntesis de rendimiento del oscilador biquad para punto fijo 22
3.7. Estimación post-síntesis de uso de recursos de FPGA usados por el oscilador biquad para punto fijo 22
3.8. Comparación de uso de recursos entre p. flotante y p. fijo 23
3.9. Estimación post-síntesis de timing del oscilador de Simulink . . . 27
3.10. Estimación post-síntesis de rendimiento del oscilador de Simulink . 27
3.11. Estimación post-síntesis de uso de recursos del oscilador de Simulink 28
3.12. Estimación post-síntesis de timing del RLC de ORTiS 32
3.13. Estimación post-síntesis de rendimiento del RLC de ORTiS . . . 32
3.14. Estimación post-síntesis de uso de recursos del RLC de ORTiS . . 32
3.15. Estimación post-síntesis de timing del inversor trifásico 34
3.16. Estimación post-síntesis de rendimiento del inversor trifásico . . . 34
3.17. Estimación post-síntesis de uso de recursos del inversor trifásico . . 34
3.18. Comparación de uso de recursos entre RLC de ORTiS vs Simscape . 38

4.1. Resumen de recursos de la FPGA XC7A35T 51

5.1. Tablas de comparación de implementación para el buck converter . 72
5.2. Tablas de comparación de implementación para el boost converter . 72

GLOSARIO

Mayúsculas

FPGA : Field-Programmable Gate Array
DSP : Digital Signal Processor
BRAIn : Board for Research, Academic and Industrial application
HIL : Hardware-In-the-Loop
RTL : Register-Transfer Level
HDL : Hardware Description Language
SoC : System on Chip
SoM : System on Module
CPU : Central Processing Unit
RAM : Random Access Memory
HLS : High-Level Synthesis
LIM : Latency Insertion Method
LB-LMC : Latency-Based Linear Multistep Compound
SSC : Simulink Simscape
SBC : Single-Board Computer
FIR : Finite Impulse Response
IIR : Infinite Impulse Response
FF : Flip-Flop
LUT : Look-Up Table
I/O : Input/Output
DUT : Design Under Test
PWM : Pulse-Width Modulation

TCL : Tool Command Language
CLI : Command-Line Interface
EMIF : External Memory Interface

Minúsculas

dc : direct current
ac : alternate current
ap : arbitrary precision
clk : clock
ce : clock enable

Capítulo 1

INTRODUCCIÓN

EN AÑOS RECIENTES, la necesidad de agilizar el flujo de trabajo y proceso iterativo del modelado y simulación de sistemas de electrónica de potencia ha crecido considerablemente. A medida que la complejidad de los sistemas de electrónica de potencia aumenta, los ingenieros se encuentran con varios desafíos en el diseño y prueba de modelos de simulación tanto *offline* como en tiempo real, antes de implementar prototipos físicos.

La simulación *hardware-in-the-loop* (HIL) se presenta como una solución valiosa a estos desafíos, permitiendo que modelos virtuales de sistemas de electrónica de potencia interactúen con hardware de control real [1]. Sin embargo, la dificultad yace en encontrar métodos eficientes e –idealmente– automatizados para traducir estos modelos a un formato digital. Dicho formato puede ser en un lenguaje de descripción de hardware (*hardware-description language*, HDL) que se ejecute en un dispositivo como una FPGA (*field-programmable gate array*) en tiempo real. Más aún, lo que principalmente se desea es modelar sistemas de dinámicas rápidas, que para estos casos se consideran del orden de los microsegundos o nanosegundos [2,3], agregando una capa adicional de complejidad y dificultad.

Este trabajo está motivado por la necesidad de disponer de medios más sencillos y confiables para modelar y simular digitalmente un sistema de electrónica de

potencia en entornos de tiempo real. Métodos actuales suelen requerir mucho tiempo y esfuerzo para desarrollar, iterar y probar los modelos, lo cual ralentiza el desarrollo, o derechamente disuade a los ingenieros de realizar este tipo de simulación.

La meta del trabajo de esta memoria será abordar estos retos, identificando y aplicando flujos de trabajo que generen código HDL a partir de modelos de electrónica de potencia, de modo que este proceso de desarrollo se adopte con mayor frecuencia.

1.1. Contexto esencial

La electrónica de potencia es un subconjunto de la electrónica análoga, donde se usan los componentes electrónicos para lidiar con voltajes/corriente ac y dc, enfocándose en controlar el flujo de energía más que en procesar señales e información. El tipo de electrónica de potencia que se detalla en esta memoria se limita a topologías "simples", donde solo se utilizan componentes lineales (resistencias, condensadores, inductores), diodos, y transistores/switches.

Además, se hace un enfoque en integrar electrónica digital, específicamente dispositivos programables como FPGAs, para potenciar tanto el modelado como el control de estos sistemas de electrónica de potencia. Estos dispositivos son muy complicados de usar, pero a la vez muy poderosos.

"BRAIn" es una plataforma de control en tiempo real, desarrollada por el Centro Avanzado de Ingeniería Eléctrica y Electrónica (AC3E) de la Universidad Técnica Federico Santa María, lugar y cultura en donde se encuentra inmerso el trabajo investigativo de esta memoria. Esta plataforma contiene una FPGA de la serie AMD Artix-7.

1.2. Trabajo por desarrollar

El eje central de esta memoria gira en torno a la investigación y evaluación de métodos para generar automáticamente código HDL a partir de modelos de sistemas de electrónica de potencia, centrándose especialmente en aplicaciones de simulación en tiempo real y HIL. Esto incluye explorar múltiples enfoques y herramientas que permitan agilizar el proceso de modelado y simulación, haciéndolo más accesible y eficiente para los ingenieros.

La plataforma BRAIn combina un dispositivo DSP y una FPGA. Su arquitectura permite la ejecución de simulaciones HIL al ejecutar modelos en tiempo real en la FPGA al mismo tiempo que interactúa con el DSP para el control. Habiendo seleccionado un método de generación de código HDL, el módulo generado será simulado extensivamente, y luego implementado en esta plataforma.

A partir de lo anterior y a modo de corolario, se escribirá una *application note*, que proporcionará una guía detallada sobre cómo implementar un modelo de convertidor de potencia específico en BRAIn [4]. Dicho documento representa la culminación tanto del trabajo explorativo como del trabajo de aplicación práctica descrito a lo largo de esta memoria, y se presenta como su resultado de mayor valor.

1.3. Estado del arte

1.3.1. Herramientas de software

Actualmente existen varios programas de software que permiten diseñar y simular sistemas dinámicos, software que permite generar código, y software que permite programar FPGAs.

- **Diseño y simulación de circuitos análogos y digitales:**
  Hay una gran variedad de estos programas de simulación de circuitos análogos, en gran parte gratuitos, que se pueden utilizar para diseñar sistemas de electrónica de potencia. Entre los más populares se encuentran **LTSpice** [5], **Qucs-S** [6], **TINA-TI** [7], **PLECS** [8], y **NI MultiSim** [9].

- **Diseño de sistemas dinámicos:**
  Software que permite diseñar/modelar gráficamente y simular sistemas dinámicos, de control, automatización, etc. Los ejemplos más prevalentes son **Simulink de MATLAB** [10], **PSIM** [11], y **LabVIEW** [12].

- **Generación de código:**
  Las opciones en este ámbito son limitadas, al menos para la generación de código de sistemas dinámicos o de electrónica. De momento, programas como **MATLAB** [13, 14] y sus alternativas como **GNU Octave** [15] y **Scilab** [16] poseen capacidades de generación de código C/C++ o incluso HDL, aunque su aplicación a sistemas de potencia puede no ser muy clara o directa, o siquiera

posible.

- **Síntesis de diseños HDL para FPGAs:**
  Estas herramientas permiten diseñar, sintetizar o incluso generar código HDL para la programación de FPGAs y SoCs. Son ampliamente utilizadas en la industria, y varias de ellas incluyen capacidades de síntesis de alto nivel (*high-level synthesis*, HLS) para realizar diseños en lenguajes de programación convencionales en lugar de HDL. Los más notables son **Vitis Unified** [17] (que incluye Vitis HLS y Vivado) y **Altera Quartus Prime** [18], junto con toolboxes como Altera DSP Builder [19] para generar automáticamente código HDL para sistemas de control automático.

1.3.2. Métodos de modelado de sistemas de electrónica

Existen maneras de modelar sistemáticamente circuitos de electrónica con múltiples estados (conmutados), de modo que se pueda implementar y simular en un entorno digital y en tiempo real. La mayoría se basa en el análisis de modelos de espacio de estados que representan el circuito [20, 21], existiendo también otros métodos como el método de inserción de latencia (LIM) [22] y otros más innovadores como el método de compuesto lineal multipaso basado en la latencia (LB-LMC) [23] (que incorpora parte de LIM).

Estos métodos suelen requerir un análisis particular del circuito por un profesional que formule las ecuaciones, o son sumamente complejos de implementar [23], o se encuentran detrás de software privado [24]. Actualmente, no se han encontrado programas o repositorios públicamente disponibles que automaticen este proceso de manera transparente o replicable.

1.3.3. Aplicaciones en simulación HIL de sistemas de potencia

Múltiples trabajos de investigación y aplicación han requerido la simulación *hardware-in-the-loop* de sistemas de electrónica de potencia. La mayoría de éstos han recurrido al análisis matemático de los circuitos que se desean modelar, y posteriormente implementar estos sistemas escribiendo personalizadamente el código C/C++ o HDL, luego simulándolos en FPGAs o microprocesadores, y en algunas ocasiones recurriendo a generación de código para alguna etapa del proceso.

También se han visto trabajos de implementación de sistemas de *control* de electrónica de potencia, en lugar del sistema de potencia mismo.

- **Sin uso de HLS:**
  Se realiza la implementación de sistemas de potencia escribiendo manualmente el código HDL que modela el circuito o controlador presentado utilizando herramientas de síntesis como Vivado o Quartus [25–28].

- **Con uso de HLS:**
  Normalmente mencionando que el diseño RTL mediante código HDL es muy complejo, se apoya en el uso de herramientas de síntesis de alto nivel para generar código HDL a partir de código en lenguajes C/C++ [29, 30].

- **Uso de ambos enfoques:**
  Utiliza tanto HLS como codificación manual de HDL para realizar una comparación de rendimiento, o usa ambos enfoques en conjunto como parte del flujo completo [29, 31].

- **Generación de código:**
  Se apoyan en herramientas de generación de código HDL a partir de diagramas de alto nivel (i.e. no en un lenguaje de programación, scripting, o de descripción de hardware) [32–34]. Dichos diseños normalmente son representaciones del circuito o controlador que nacen también a partir de un análisis matemático del sistema.

Hasta el momento, no se han encontrado trabajos de implementación de modelos de convertidores de potencia o similar que no involucren un trabajo matemático previo, debiendo realizar labor extra posterior al diseño mismo del circuito para llegar a un modelo implementable en una FPGA, en lugar de solo enfocarse en dicho diseño y abstraerse de la matemática y/o del código.

Sin embargo, las herramientas de software existentes tienen el potencial de poder utilizarse en conjunto para lograr solucionar el problema planteado al inicio de este capítulo.

1.4. Objetivos

- Evaluar y analizar múltiples métodos y herramientas para la generación automática de código HDL a partir de modelos de sistemas de electrónica de potencia.
- Llegar a un método a partir del análisis anterior, que cumpla con el objetivo de permitir el diseño de sistemas de potencia y obtener un módulo descrito en código HDL.
- El modelo generado debe ser implementable en BRAIn sin consumir una excesiva cantidad de recursos ni exceder sus capacidades de cómputo.
- Simular, implementar y probar dos tipos de convertidores de potencia, específicamente el *buck converter* y el *boost converter*, para demostrar la efectividad del método propuesto.
- Producir una *application note* que guíe a ingenieros a aplicar los métodos investigados y evaluados en sus propios proyectos, donde se requiera modelar y simular sistemas de electrónica de potencia en FPGAs, utilizando un caso de estudio.

1.5. Alcances y contribuciones

En esta memoria se hará una exploración detallada de las herramientas y flujos de trabajo requeridos para la generación de código HDL para simulación en tiempo real y HIL. El enfoque será principalmente en sistemas de electrónica de potencia, en particular convertidores de potencia, aunque los métodos evaluados serían aplicables a diversas topologías más allá de las expuestas en este documento.

Se busca en particular que el usuario solo deba ocuparse del diseño a nivel de circuito del sistema, abstrayéndose del análisis matemático y de escribir código directamente.

Para la generación de los modelos de convertidores, solo se hará un enfoque en el comportamiento eléctrico del *sistema completo*, y no en el comportamiento *individual de cada componente*. Esto se menciona a propósito de que el estudio y simulación de temas como la disipación térmica son bastante prevalentes hoy en día,

pero esto no se tocará en esta memoria.

Las principales contribuciones de este trabajo incluyen:

1. Una evaluación profunda de herramientas y metodologías de generación de código HDL, presentando múltiples alternativas que sean potencialmente valiosas para contextos o situaciones incluso no abarcados en esta memoria.

2. Presentar un proceso confiable para simulaciones en tiempo real y HIL de sistemas de electrónica de potencia, usando hardware digital.

3. Una demostración práctica del proceso, utilizando la plataforma BRAIn.

4. La creación de una *application note* detallando la implementación del método en un caso de uso. Se espera que este documento fomente la adopción de FPGAs como herramienta para simulación *hardware-in-the-loop*.

1.6. Estructura del documento

Este documento está dividido en 5 capítulos y 2 anexos.

- **Capítulo 1: Introducción**
  Establece la motivación, objetivos y alcances de la memoria. Introduce el problema de querer agilizar el proceso de modelado de sistemas de electrónica de potencia y discute el contexto del trabajo a grandes rasgos.

- **Capítulo 2: Marco contextual**
  Provee información básica y fundamental sobre electrónica de potencia y la electrónica digital, destacando cómo ambos campos se intersectan en el área de simulación en tiempo real. Además, introduce la plataforma BRAIn y su rol a la hora de realizar simulaciones HIL.

- **Capítulo 3: Evaluación de alternativas de generación de código**
  Se realiza una evaluación de métodos, herramientas y procesos de generación de código HDL para simulación en tiempo real, para BRAIn. El capítulo explora varios enfoques y soluciones, y determina las ventajas y desventajas de cada método para distintos contextos, finalmente decidiéndose por uno en específico.

- **Capítulo 4: Modelos de convertidores de electrónica potencia en BRAIn**
  Indaga en la implementación práctica del modelo de un convertidor de potencia, enfocándose en el diseño y requisitos de esta implementación. Describe en más detalles BRAIn, explica el flujo de trabajo del método escogido en el capítulo anterior, y muestra un resumen de la *application note* que surge a partir de todo el trabajo realizado.

- **Capítulo 5: Resultados experimentales**
  Resume los resultados obtenidos en el capítulo 4, mostrando cada etapa de simulación, incluyendo la simulación RTL [35] de los módulos generados, y los modelos siendo ejecutados en tiempo real en hardware.

- **Capítulo 6: Conclusiones y trabajo futuro**
  Se presentan conclusiones a partir de los resultados anteriores y del trabajo de la memoria en su conjunto, y se discuten posibles trabajos a futuro.

- **Apéndice A: Uso de herramientas de software**
  Describe con mayor detalle el uso específico de las herramientas de software utilizadas a lo largo del capítulo 3.

- **Apéndice B: Códigos y scripts**
  Muestra los códigos y scripts (C, MATLAB) utilizados a lo largo del capítulo 3, para poder recrear los resultados presentados en éste.

Capítulo 2

MARCO CONTEXTUAL

En este capítulo se presenta el contexto en el cual se desarrolla la memoria, entregando un breve vistazo general a los conceptos teóricos o técnicos que son relevantes al trabajo por realizar, dando a conocer el hardware que se usará, y mencionando las herramientas de software y plataformas que existen actualmente en el área. Esta información sirve para lograr un entendimiento del trabajo expuesto en los capítulos siguientes.

2.1. Electrónica de potencia

Los sistemas de electrónica de potencia juegan un rol fundamental en la tecnología moderna, y tienen uso en prácticamente todo tipo de aplicaciones tecnológicas, como en computadores, telecomunicación, medicina, aeroespacial, entre otros.

En pocas palabras, el rol de la electrónica de potencia es procesar y controlar el flujo de energía eléctrica, suministrando voltaje y corriente de modo que se adapte de manera óptima y eficiente a las múltiples cargas del usuario. La fuente de energía puede ser de tipo ac (*alternate current*, corriente alterna) de una a tres fases, o dc (*direct current*, corriente continua), y puede provenir de elementos como baterías, transformadores, rectificadores, etc. [36, 37]

La Figura 2.1 muestra un diagrama simplificado de un sistema de electrónica de potencia con una fuente y solo una carga [38]. El sistema realiza la conversión de energía eléctrica desde el suministro de energía hasta la carga, y el circuito de electrónica de potencia es monitoreado por un controlador que compara la energía de salida (feedback) y de entrada (feedforward) con los valores de referencias necesarios para obtener los resultados deseados.

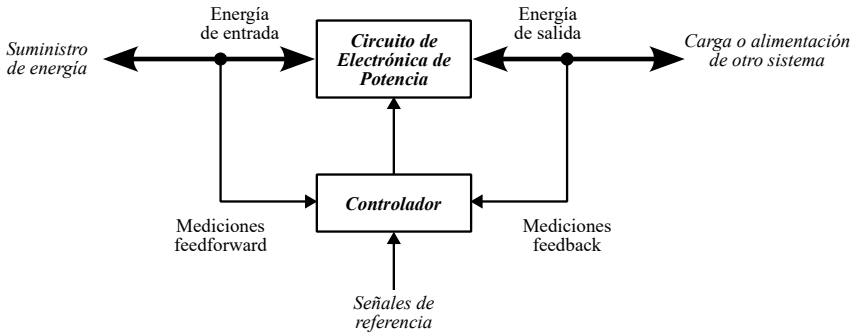


Figura 2.1: Un sistema de electrónica de potencia

El controlador puede ser diseñado de forma análoga o digital. Hoy en día, la forma digital es la más común, dejando la forma análoga para sistemas electrónicos de potencia más sencillos.

El trabajo a desarrollar a lo largo del documento implica la simulación de modelos de sistemas de electrónica de potencia, implementados digitalmente, para realizar HIL en tiempo real. Por ende, tanto el controlador como el sistema mismo harán uso de tecnologías de electrónica digital como herramienta y soporte computacional para el modelado del sistema..

Además, en particular se trabajará con *sistemas lineales conmutados*, los que corresponden a sistemas dinámicos compuestos por múltiples subsistemas lineales, donde una entidad externa, como un modulador o controlador, determina qué subsistemas se encuentran activos o interconectados en un determinado instante.

2.2. Electrónica digital y FPGAs

La electrónica digital constituyen el eje central de la computación y sistemas de control modernos, permitiendo manipulación precisa de datos y señales mediante representación discreta con lógica binaria. En áreas de telecomunicaciones,

automatización, y control en tiempo real, los sistemas digitales proveen precisión y flexibilidad necesarias para tareas de alta dificultad.

Una de las mayores ventajas de la electrónica digital moderna es la existencia de dispositivos programables, lo que le otorga la característica de flexibilidad y adaptabilidad, dado que también son reconfigurables. Entre los ejemplos más importantes para este tipo de sistemas se pueden encontrar dispositivos como microcontroladores, procesadores digitales de señales o DSPs, y *field-programmable gate arrays* o FPGAs.

Las FPGAs, en particular, se presentan como alternativas muy poderosas. Su fortaleza radica en su arquitectura reconfigurable que se presta para la implementación de diseños lógicos a medida, en lugar de algoritmos ejecutados secuencialmente como en un CPU. Esto les permite implementar diseños que se ajusten a las necesidades de las tareas que se desean cumplir, de modo que puedan lograr un nivel de optimización y rendimiento mucho mayor que si se utilizara un dispositivo programado por software (en un lenguaje de propósito general). Por ello, las FPGAs son una gran herramienta para simulación *hardware-in-the-loop*.

Una desventaja de las FPGAs es que su forma de programación es mediante lenguajes de descripción de hardware (*hardware description languages*, HDL). Escribir código desde cero en HDL para diseñar e implementar sistemas digitales complejos es considerablemente más difícil, ya que implica un cambio de paradigma desde lenguajes de programación imperativos, que lidian con procedimientos, módulos y funciones, hacia uno que describe la arquitectura de hardware, donde se define claramente cómo se interconectan los módulos físicos a nivel de registros.

2.3. Tipos de plataformas digitales programables existentes

Existe una gran variedad de dispositivos digitales programables, algunos de los cuales se ha mencionado en la sección anterior. Además, existen plataformas que combinan e integran dos o más de estos dispositivos para ampliar sus capacidades y funciones. A continuación se listan algunos ejemplos de estos dispositivos y plataformas.

- **Microcontroladores.**
  Son sistemas compactos de computación, diseñados para realizar tareas específicas en sistemas embebidos. Normalmente integran un CPU, memoria, puertos de entrada/salida, y otros periféricos, en un mismo chip optimizado para bajo consumo.
  Ejemplos: Arduino [39], Serie STM32 [40].

- **Microprocesadores.**
  CPUs de propósito general que ejecutan sistemas operativos y permiten manejar entornos multitarea.
  Ejemplos: ARM_Cortex-R [41], Intel Core i7 [42].

- **Procesadores digitales de señales.**
  Procesadores especializados para manejar procesamiento de señales en tiempo real de forma efectiva, como audio o video. A diferencia de los anteriores, buscan contar con latencias de respuestas deterministas, e idealmente reducidas.
  Ejemplo: Serie Texas Instruments TMS320 [43].

- **FPGAs.**
  Dispositivos de hardware reconfigurable que permiten implementar circuitos lógicos y sistemas digitales con arquitectura a medida, muy adecuadas para procesamiento de tareas en paralelo al permitir instanciar físicamente circuitos lógicos para cada tarea.
  Ejemplos: Xilinx Kintex-7 [44], Altera Agilex 7 [45].

- **System-on-chip (SoC).**
  Un SoC integra varios o todos los componentes de un computador en un mismo chip (típicamente un CPU, GPU, memoria, I/O e incluso una FPGA).
  Ejemplos: AMD Zynq 7000 [46], Broadcom BCM2711 [47]

Híbridos

Hay sistemas y plataformas que combinan varios de los elementos anteriores en una misma placa, para expandir y mejorar sus funcionalidades. Ejemplos de estas plataformas son:

- **Módulos de evaluación (EVM) y de desarrollo (DevKits).**
  Kits que combinan varios componentes y periféricos en conjunto con un dispositivo programable, para facilitar el prototipado y desarrollo.
  Ejemplos: Zybo Z7 [48], PolarFire SoC Discovery Kit [49].

- **Single Board Computer (SBC).**
  Computadores pequeños y típicamente monolíticos (en una sola placa o PCB).
  Ejemplos: Raspberry Pi [50], BeagleBone Black [51].

- **System-on-module (SoM).**
  PCBs con varios componentes: CPU, FPGA, MCU, PMIC, memorias, chips de radiofrecuencia, etc. A diferencia de los SBC, éstos se diseñan para incrustarse dentro de otros PCBs y sistemas.
  Ejemplo: Arduino Portenta H7 [52]

2.4. Plataforma de trabajo: "BRAIn"

Para el trabajo de memoria por realizar, se usará la plataforma BRAIn, del inglés *Board for Research, Academic and Industrial applications*, desarrollada por el AC3E de la Universidad Técnica Federico Santa María. Esta plataforma consiste en un sistema DSP + FPGA, y está específicamente diseñada para el control en tiempo real de sistemas de electrónica de potencia. Se puede clasificar como un SoM.

La Figura 2.2 muestra la tarjeta BRAIn. Contiene un gran número de puertos de entrada y salida, además de varios puertos de expansión, donde conectar periféricos para añadir funcionalidades y características extra. La FPGA se encarga de implementar módulos digitales adicionales al DSP, y sirve como complemento de lógica reconfigurable a éste. El DSP se suele usar como el principal elemento de procesamiento de datos, y se programa en C.

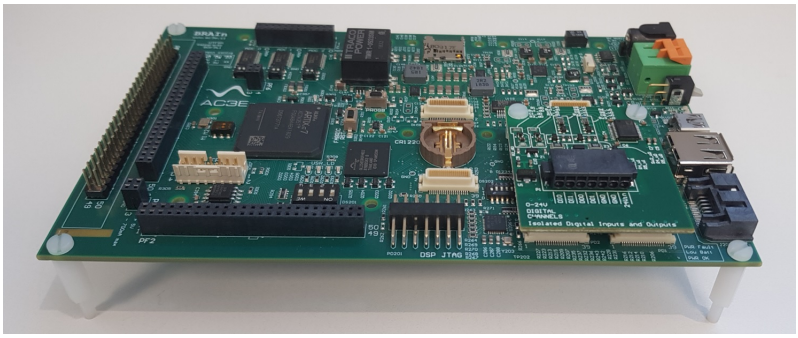


Figura 2.2: Tarjeta de control BRAIn

BRAIn se presenta como una muy buena opción para esta memoria, ya que la motivación principal es poder usar la FPGA para HIL, y el DSP para implementar el controlador del sistema.

Si bien dispositivos como microcontroladores y DSPs por sí solos también pueden usarse para control en tiempo real, el sistema de cómputo dual DSP + FPGA de BRAIn supera con creces la necesidad computacional de la mayoría de los sistemas de control de electrónica de potencia prácticos. Además, es una alternativa relativamente económica en comparación a otras plataformas similares.

Ya que se utilizará BRAIn como plataforma principal, el siguiente paso es evaluar las herramientas y métodos que faciliten la implementación de sistemas de electrónica de potencia en ella. Es importante contar con recursos que optimicen el uso de BRAIn y maximicen sus capacidades.

2.4.1. Desafíos y consideraciones

La plataforma BRAIn tiene integrada una FPGA de la línea Artix-7, considerada una línea económica y de gama baja. Por ende, se debe tener en cuenta las limitaciones que esto impone al momento de buscar alternativas de generación de código.

Entre las consideraciones más importantes a tomar en cuenta para la generación de código se resaltan las siguientes:

- **Bajo uso de recursos:**
  Por ser de gama baja, la FPGA de BRAIn cuenta con una cantidad de recursos

lógicos/digitales relativamente limitado. Si bien se pueden implementar diseños lógicos bastante complejos y grandes, se busca disminuir lo más posible los recursos para implementar modelos de sistemas dinámicos.

- **Optimización para rendimiento:**
  Usualmente, para algún diseño de electrónica digital, se puede priorizar integridad numérica por sobre rendimiento o viceversa. En el contexto de sistemas de electrónica de potencia y su simulación HIL, se considerará más importante el rendimiento que la integridad numérica. Esto quiere decir que la tolerancia será alta mientras se mantenga bien modelado el comportamiento dinámico general.

- **Velocidad de cómputo:**
  Al lidiar con sistemas de electrónica donde se manipulan variables como voltajes, corrientes y potencias con rangos dinámicos aceptables en torno a 60 dB - 80 dB, una alta precisión decimal no será de vital importancia. Por esto, se busca que el enfoque en rendimiento permita al usuario alcanzar el paso de simulación deseado.

Capítulo 3

EVALUACIÓN DE ALTERNATIVAS DE GENERACIÓN DE CÓDIGO

El trabajo de memoria busca encontrar métodos para generar automáticamente código HDL, de modo que sea implementable en una FPGA y ejecutable en tiempo real. El código HDL debe describir la arquitectura lógica del algoritmo numérico que implementa un modelo dinámico discreto de sistemas de electrónica de potencia. Además, estos métodos de generación de código deben ser lo suficientemente flexibles como para permitir al usuario cumplir con restricciones de hardware, en este caso de la plataforma BRAIn, y diseñar diversas topologías.

Este capítulo detalla varios de estos métodos y flujos de trabajo, evaluando cada uno para determinar su efectividad en cumplir con los requisitos de implementación en BRAIn, además de su viabilidad para algún otro tipo de aplicación.

3.1. Metodología de testeo

Se busca que el usuario tenga una manera de generar código HDL a partir de un esquemático de alto nivel de un sistema de electrónica de potencia. Es posible que el usuario deba escribir código C o C++ para hacer llamado al código generado o

para realizar ciertas optimizaciones, pero idealmente se busca evitar que:

- El usuario tenga que formular manualmente las ecuaciones de diferencia a partir del esquemático para poder implementar el sistema en algún lenguaje, desde cero.
- El usuario tenga que escribir código HDL de cualquier tipo (como Verilog o VHDL).

Cada método explorado en este capítulo será evaluado siguiendo un flujo de trabajo común, para determinar sistemáticamente su viabilidad. Este flujo se detalla en el diagrama de la Figura 3.1. A partir de él, se consideran los siguientes criterios:

- **Precisión:**
  Qué tan exacto es el modelo del código generado en comparación al modelo original.
- **Eficiencia:**
  Consumo de recursos (celdas lógicas, bloques DSP, etc) y rendimiento del código generado.
- **Flexibilidad:**
  Capacidad de armar distintas topologías de electrónica de potencia dependiendo de la cantidad de componentes soportados por la herramienta de diseño de circuitos.
- **Facilidad de uso:**
  Qué tan intuitivo y fácil de usar es el proceso de generación de código, desde el modelado del sistema hasta la simulación.

También se puede considerar como criterio de evaluación la cantidad de soporte/documentación que posea el método, pero éste será el de menor prioridad.

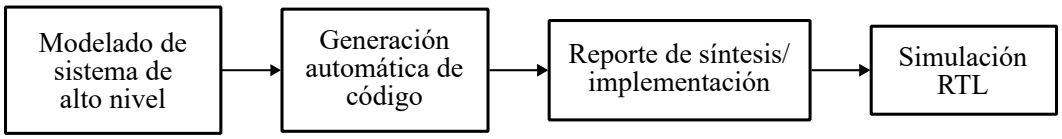


Figura 3.1: Flujo de trabajo para testear cada método de generación de código

3.2. "High Level Synthesis" y Vitis HLS

La síntesis de alto nivel (HLS - *high level synthesis*) es un proceso de diseñado automático que convierte un diseño digital descrito en un lenguaje funcional abstracto (por ejemplo, un lenguaje de programación como C o C++) en una descripción de lenguaje de transferencia de registros (RTL - *register transfer level* [53]) que se pueda sintetizar lógicamente a un HDL. [54, 55]

Vitis HLS es una herramienta de AMD que permite sintetizar código C y C++ en código RTL para implementación en FPGAs. En lugar de escribir hardware complejo en HDL, Vitis HLS permite a los desarrolladores utilizar abstracciones de alto nivel en un lenguaje más familiar, reduciendo así la dificultad y tiempo de desarrollo para crear hardware a medida. [56]

Además, Vitis permite utilizar aritmética de punto fijo en lugar de punto flotante, que consume una gran cantidad de recursos de hardware. Este será un punto importante para cumplir con las consideraciones mencionadas en el capítulo anterior sobre la utilización de recursos y complejidad. [57]

Vitis HLS será la principal herramienta empleada para sintetizar el código C/C++ que se desee simular o implementar. La sección a continuación detallará una implementación escrita en C manualmente, antes de seguir con herramientas de generación automática de código C.

3.3. Implementación inicial de algoritmo en C

Para comprobar la operatividad de Vitis HLS, se realiza la implementación de un algoritmo conocido en C++ con el que se prueba la metodología.

3.3.1. Oscilador dual de tipo biquad

El oscilador biquad es un caso particular de un filtro biquad IIR (filtro IIR de segundo orden con dos polos y dos ceros), y corresponde a un resonador ideal que genera ondas sinusoidales en respuesta a un impulso [58] [59]. Este filtro es sencillo de implementar en un sistema en tiempo real, por lo que es una excelente opción para evaluar la utilidad de Vitis para este propósito.

La función de transferencia de tiempo discreto del filtro biquad tiene la forma

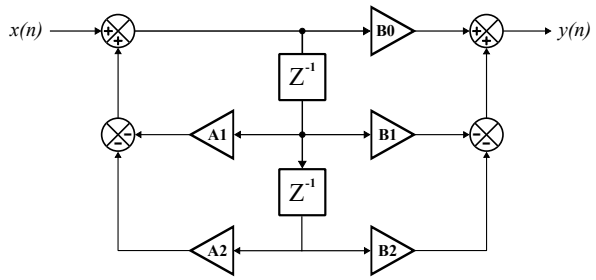
general como se muestra en la ecuación 3.1.

$$H_{\text{IIR}}(z) = \frac{b_0 + b_1\, z^{-1} + b_2\, z^{-1}}{a_0 + a_1\, z^{-1} + a_2\, z^{-1}} \tag{3.1}$$

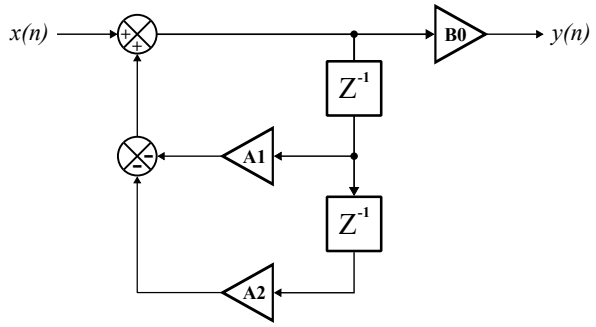
El oscilador biquad es tal que se cumple que $a_0 = 1$, $a_1 = -2 \cdot \cos(\omega_o)$, $a_2 = 1$, $b_0 = \sin(\omega_o)$, y $b_1 = b_2 = 0$, donde $\omega_o = 2\pi \cdot \frac{f}{f_s}$ es la frecuencia a la que se desea que oscile la onda sinusoidal generada, normalizada a la frecuencia de muestreo f_s. La ecuación 3.2 muestra la función de transferencia del oscilador.

$$H_{\text{biq}}(z) = \frac{\sin(\omega_o)}{1 - 2 \cdot \cos(\omega_o)\, z^{-1} + z^{-1}} \tag{3.2}$$

La Figura 3.2 muestra estas funciones de transferencia en su forma de diagrama de bloques. Utilizando como referencia el diagrama de bloques de la Figura 3.2b, se programan en C dos osciladores operando a una frecuencia de muestreo $f_s = 16\,\text{kHz}$,



(a) Forma directa II de un filtro IIR de segundo orden, normalizado para $a_0 = 1$



(b) Oscilador biquad (caso particular de filtro biquad)

Figura 3.2: Diagrama de bloques del filtro IIR y del oscilador biquad.

con los parámetros de cada uno detallados en la Tabla 3.1.

Tabla 3.1: Parámetros para los dos osciladores biquad

Oscilador 1	**Oscilador 2**
$f = 130{,}8$ Hz	$f = 164{,}8$ Hz
$\sin(\omega_o) = 0{,}051342456225139$	$\sin(\omega_o) = 0{,}064671642929027$
$-2 \cdot \cos(\omega_o) = -1{,}997362212708321$	$-2 \cdot \cos(\omega_o) = -1{,}995813196269491$

3.3.1.1. Simulación previa

Para validar el código que modela los dos osciladores, se construye un archivo `main.c` donde se llama la función recién programada, entregando un impulso de amplitud 1 a cada oscilador, y guardando los resultados de la salida de la función en un archivo de texto. El programa es compilado desde la IDE de Vitis, que incluye un compilador C. Los puntos generados son graficados con MATLAB.

La Figura 3.3 muestra el resultado de esta simulación, donde se observan las ondas generadas por el oscilador 1 (azul: 130,8 Hz) y el oscilador 2 (naranjo: 164,8 Hz). Esta simulación del oscilador será la referencia para verificar el comportamiento del código RTL generado por Vitis.

En el Apéndice B se encuentra el código completo, junto con el script de MATLAB que genera el gráfico del resultado del programa en C.

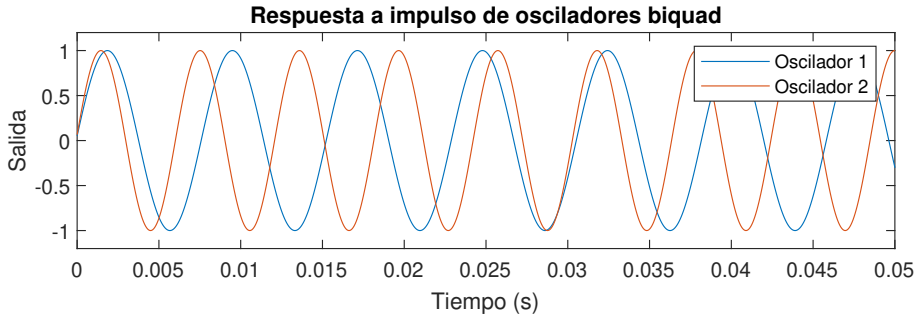


Figura 3.3: Resultado de simulación de osciladores en C, compilado desde el IDE de Vitis

3.3.1.2. Reporte de síntesis

Vitis genera un reporte de síntesis que entrega estimaciones sobre varios elementos, como uso de recursos de FPGA, frecuencia de operación, estimación de

timing, y la disposición de los puertos del módulo generado, basándose en el modelo de FPGA que se entregue al proyecto. Este reporte permite guiar la optimización del módulo previo a la generación del código RTL propiamente tal.

Se usa aritmética de punto flotante inicialmente y luego se cambia a punto fijo para comparar los reportes y evaluar el impacto. Aunque lo ideal sería usar punto fijo de 16 bits, se genera el reporte de síntesis para 32 bits para facilitar la comparación con el punto flotante, ya que Vitis solo soporta precisiones simple (32 bits) y doble (64 bits), no media (16 bits). [60]

Punto flotante de precisión simple

- **Timing:** (Periodo de reloj objetivo versus periodo de reloj mínimo requerido estimado.)

Tabla 3.2: Estimación post-síntesis de timing del oscilador biquad para punto flotante

Objetivo	Estimado	Incertidumbre
10.00 ns	6.968 ns	2.70 ns

- **Rendimiento:**

Tabla 3.3: Estimación post-síntesis de rendimiento del oscilador biquad para punto flotante

Latencia (ciclos)	Latencia (ns)	Intervalo
18	180.000	19

- **Uso de recursos:**

Tabla 3.4: Estimación post-síntesis de uso de recursos de FPGA usados por el oscilador biquad para punto flotante

Instancias	DSP	FF	LUT
Utilizados	16	1449	1091
Disponibles	90	41600	20800
Utilizados (%)	17	3	5

Punto fijo de 32 bits, con 24 bits de precisión fraccionaria

- **Timing:**

Tabla 3.5: Estimación post-síntesis de timing del oscilador biquad para punto fijo

Objetivo	Estimado	Incertidumbre
10.00 ns	6.880 ns	2.70 ns

- **Rendimiento:**

Tabla 3.6: Estimación post-síntesis de rendimiento del oscilador biquad para punto fijo

Latencia (ciclos)	Latencia (ns)	Intervalo
1	10.000	2

- **Uso de recursos:**

Tabla 3.7: Estimación post-síntesis de uso de recursos de FPGA usados por el oscilador biquad para punto fijo

Instancias	DSP	FF	LUT
Utilizados	8	306	368
Disponibles	90	41600	20800
Utilizados (%)	8	˜0	1

Al contrastar los resultados de síntesis de ambas implementaciones, se puede notar inmediatamente que utilizar punto fijo para los datos de precisión genera un gran impacto.

- **Timing y rendimiento:**
  El periodo de reloj estimado varía de manera muy minúscula (de 6.9868 ns a 6.8800 ns), pero el rendimiento del módulo mejora considerablemente: la latencia (tiempo entre que se recibe la entrada y se genera la salida) disminute desde 18 ciclos hasta tan solo 1 ciclo.

- **Uso de recursos:**
  La cantidad de recursos de la FPGA disminuye considerablemente, detallado en la Tabla 3.8.

Tabla 3.8: Comparación de uso de recursos entre p. flotante y p. fijo

Componente	P. flotante	P. fijo	Variación
DSP	16	8	↓ 50 %
FF	1449	306	↓ 79 %
LUT	1091	368	↓ 66 %

Esto demuestra la utilidad de trabajar con aritmética de punto fijo en cuanto a optimización del modelo generado. Desde ahora en adelante, cada implementación en Vitis usará punto fijo para los tipos de dato de precisión fraccionaria.

3.3.1.3. Simulación RTL

Tras el reporte de síntesis y la generación del modelo HDL del oscilador, se realiza una simulación RTL en Vivado. Se crea un testbench en SystemVerilog, que instancia el módulo generado por Vitis y proporciona las entradas adecuadas, junto con controlar la frecuencia de activación (enable). Esta simulación verifica que el comportamiento lógico y numérico del módulo coincida con el del programa en C.

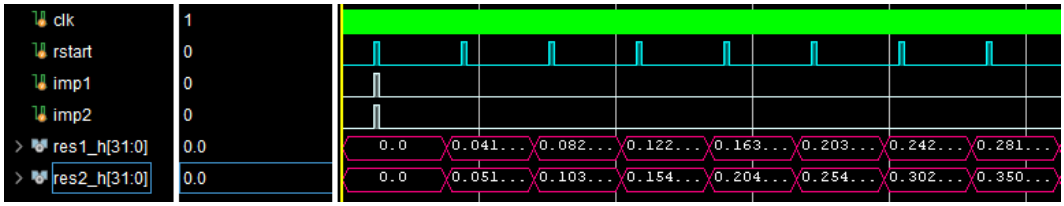


Figura 3.4: Simulación RTL del oscilador biquad con acercamiento

La Figura 3.4 muestra el módulo recibiendo un impulso en la entrada y luego siendo activado periódicamente para generar valores distintos en la salida cada vez, demostrando así que el módulo reacciona al impulso.

En la Figura 3.5 se cambia la forma de visualización de los datos de salida decimales a una onda análoga, para poder ver las ondas sinusoidales, y además se activa el módulo a una frecuencia igual a la de muestreo definida (16 kHz), así haciendo coincidir los tiempos con aquellos de la simulación de la Figura 3.3. (En la figura se han ocultado señales de control que no son importantes de mostrar.)

Para verificar si el módulo reacciona de la forma correcta, se excitan los osciladores con un par de impulsos más después de que ya hayan comenzado a oscilar.

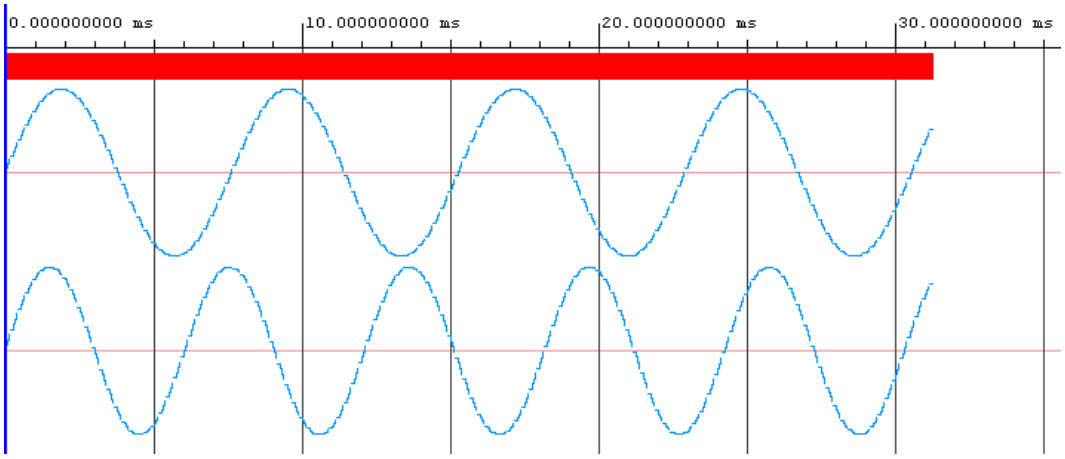


Figura 3.5: Simulación RTL con visualización análoga

El resultado de esto se observa en la Figura 3.6, donde los osciladores reaccionan al impulso de acuerdo al valor actual de la onda y su fase, demostrando que el sistema puede ser manipulado en tiempo real y no tiene un comportamiento fijo.

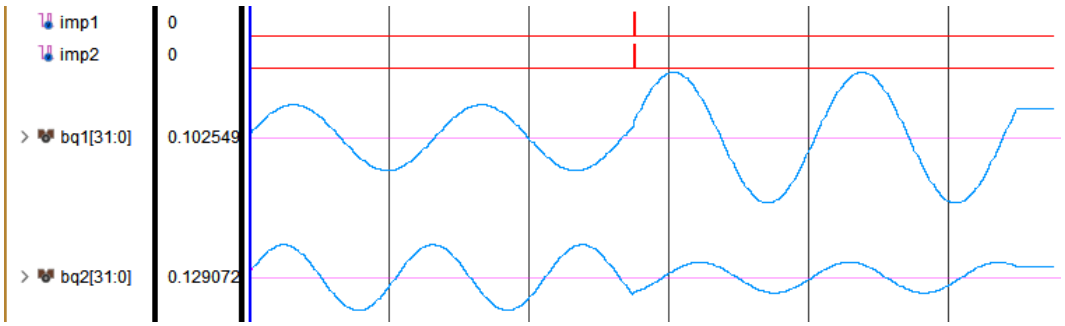


Figura 3.6: Reacción del módulo ante nuevos impulsos en la entrada

3.4. MATLAB Simulink y Embedded Coder

Ya que el proceso anterior requiere escribir código C manualmente, lo cual se quiere evitar si se desea modelar sistemas a alto nivel, se hace la prueba de utilizar Simulink para el diseño de estos modelos. Simulink ofrece un enfoque gráfico al modelado de sistemas, lo que simplifica el diseño y simulación de sistemas dinámicos. [10]

Una vez que se tiene el modelo completado, se utiliza el add-on de Simulink, “Embedded Coder”, para generar automáticamente el código C del modelo. Este

enfoque no solo elimina la necesidad de escribir código manualmente, sino que también garantiza que el código se adhiera a las especificaciones del modelo con un alto grado de precisión. [61]

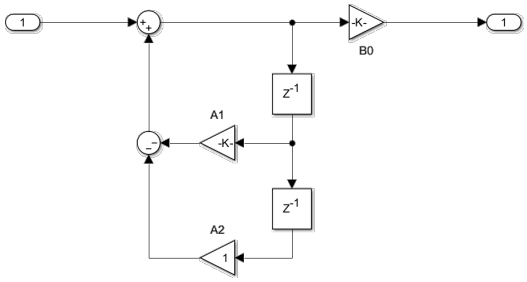
La desventaja de Simulink es que, para efectos de modelar sistemas de electrónica de potencia, hay que igualmente calcular las ecuaciones del circuito para poder generar su modelo matemático y poder realizar una implementación precisa utilizando bloques de Simulink, lo cual no es un proceso trivial.

3.4.1. Oscilador biquad singular en Simulink

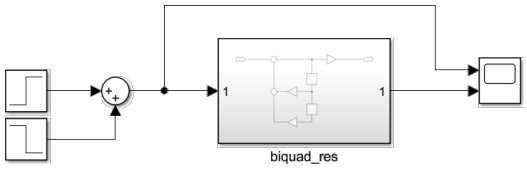
En esta ocasión solo se implementará el oscilador de 130.8 Hz en lugar del sistema dual de osciladores. Usando de referencia el diagrama de bloques de la Figura 3.2b, se arma el sistema del oscilador biquad en Simulink, mostrado en la Figura 3.7. La configuración de Solver de la simulación es utilizando “Type: Fixed-step” y “Solver: discrete (no continuous states)”, para emular un entorno digital discreto con un paso constante.

3.4.1.1. Simulación previa

En la Figura 3.7b se muestra el sistema encapsulado, con entradas y salidas asignadas para simular su comportamiento. Los dos bloques de escalón se utilizan para simular un impulso en la entrada. Esta simulación se muestra en la Figura 3.8, siendo ejecutada para distintos tiempos de simulación. Se observa que el periodo de la señal sinusoidal generada es de aproximadamente 122 muestras, esto a una

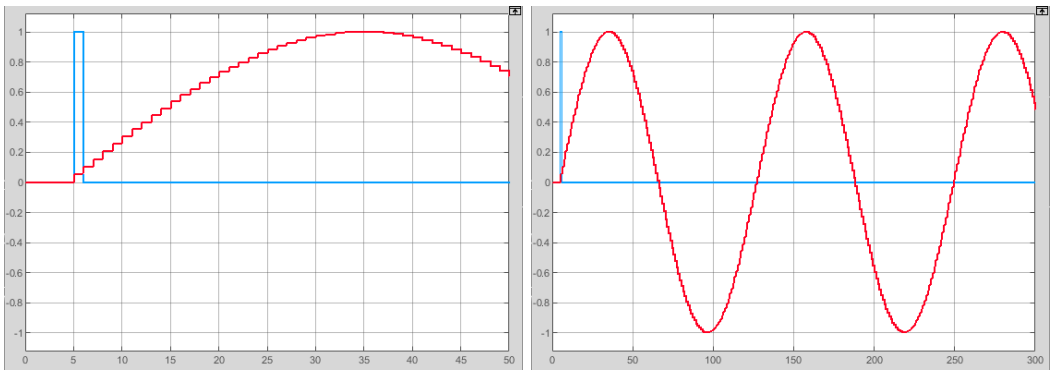


(a) Diagrama de bloques del oscilador a 130.8 Hz.



(b) Sistema encapsulado.

Figura 3.7: Sistema del oscilador en Simulink.



(a) Tiempo de simulación = 50 muestras (b) Tiempo de simulación = 300 muestras

Figura 3.8: Simulación del oscilador en Simulink (vista de Scope).

frecuencia de muestreo $f_s = 16$ kHz, resulta en:

$$f = \frac{16000 \text{ muestras/segundo}}{122 \text{ muestras}} = 131{,}1 \text{ Hz} \approx 130{,}8 \text{ Hz} \tag{3.3}$$

Por lo tanto, la respuesta a impulso del oscilador diseñado en Simulink funciona correctamente, y se procede a la generación de código.

3.4.1.2. Generación de código C para Vitis y segunda simulación

Embedded Coder permite generar código C del subsistema `biquad_res` de la Figura 3.7b (para no generar el código C de todo el sistema externo). Para corroborar la correctitud del código, se construye una prueba unitaria donde, al ejecutar el programa, se entrega un impulso en la entrada del sistema y se capturan los datos de la salida en un archivo de texto, similar a como se hizo en la sección 3.3.1.1.

Una vez que se tienen los datos, se grafican los puntos en MATLAB para visualizar y analizar el resultado. La Figura 3.9 muestra la simulación del oscilador de 130.8 Hz original a modo de referencia (arriba), y la simulación del oscilador de 130.8 Hz que nace a partir del código C generado por Embedded Coder (abajo).

Como se puede observar, el comportamiento se mantiene idéntico, por lo que el código C generado funciona correctamente.

El proceso de uso de Embedded Coder se encuentra en el Apéndice A, y los

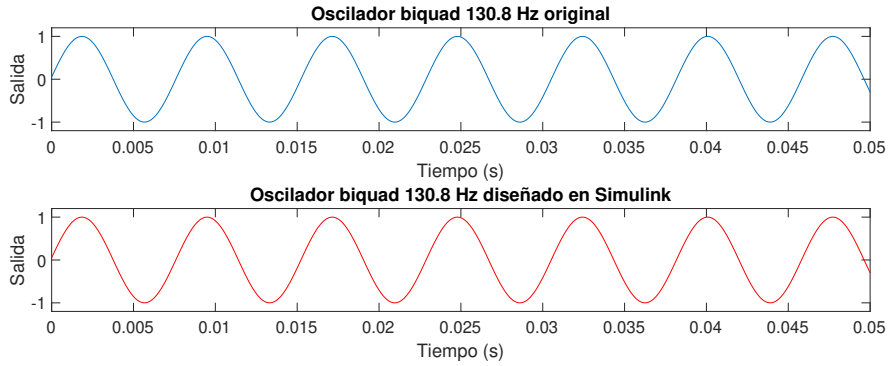


Figura 3.9: Resultado de simulación de oscilador diseñado en Simulink.

códigos y scripts de esta sección se encuentran en el Apéndice B.

3.4.1.3. Reporte de síntesis

El sistema generado por Embedded Coder se implementa en Vitis para comenzar el proceso de generación de código HDL, y posteriormente se genera el reporte de síntesis. Embedded Coder genera un archivo de cabecera (`.h`) donde fácilmente se puede cambiar el tipo de dato de precisión.

- **Timing:**

Tabla 3.9: Estimación post-síntesis de timing del oscilador de Simulink

Objetivo	Estimado	Incertidumbre
10.00 ns	6.880 ns	2.70 ns

- **Rendimiento:**

Tabla 3.10: Estimación post-síntesis de rendimiento del oscilador de Simulink

Latencia (ciclos)	Latencia (ns)	Intervalo
2	20.000	3

- **Uso de recursos:**

Tabla 3.11: Estimación post-síntesis de uso de recursos del oscilador de Simulink

Instancias	**DSP**	**FF**	**LUT**
Utilizados	6	187	171
Disponibles	90	41600	20800
Utilizados (%)	6	˜0	1

Como se puede observar, aún no se han encontrado problemas de reloj, ya que el reloj estimado sigue estando por debajo del reloj objetivo incluso considerando la incertidumbre.

También se aprecia que la latencia del módulo (20 ns) esta muy por debajo de la frecuencia de muestreo del módulo (62.5 μs), por lo que en cuanto a timing el módulo generado está dentro de los márgenes aceptables.

Finalmente, el uso de recursos de FPGA que utiliza el módulo se mantiene proporcionalmente similar al caso de los dos osciladores biquad de la sección anterior; para los dos osciladores, se utilizaron alrededor de 680 registros, mientras que para este único oscilador se utilizan alrededor de 360 registros.

3.4.1.4. Simulación RTL

Una vez generado el código HDL, se implementa el módulo en un proyecto de Vivado y se crea un testbench para validar el comportamiento del sistema. Similar a como se hizo en la sección 3.3.1.3, se envía un impulso inicial para echar a andar el oscilador, y posteriormente se envía un segundo impulso para verificar que el sistema es dinámico y responde en tiempo real.

El resultado de esta simulación se muestra en la Figura 3.10. Con esto se confirma que Simulink y Embedded Coder son una opción viable para generar código C a partir del modelo de un sistema, que además es compatible con el proceso HLS de Vitis.

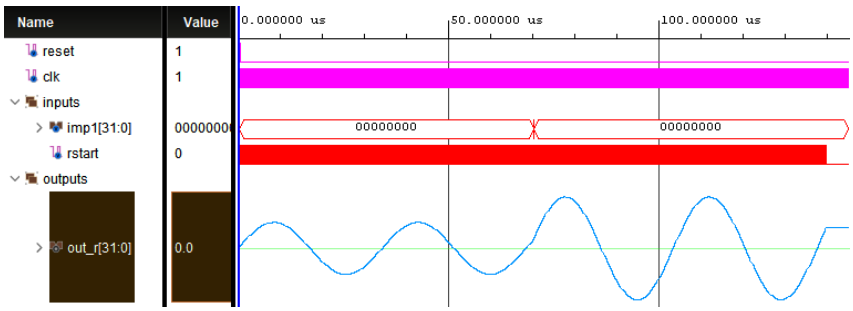


Figura 3.10: Simulación RTL con visualización análoga

3.5. Exploración de "ORTiS"

La herramienta ORTiS Solver Codegen es un *framework* para generar solvers de sistemas eléctricos en C++ para simulación tanto *offline* como en tiempo real. Se escribe un archivo netlist que describa el sistema eléctrico que se desea simular, siguiendo la sintaxis soportada por la herramienta, y posteriormente ésta genera código C++ sintetizable por herramientas HLS (como Vitis) [62]. Ya que se enfoca en simulación en tiempo real, esta netlist también permite definir un paso de simulación para los componentes.

Esta herramienta se presenta como una alternativa inicial para el diseño de circuitos electrónicos de potencia, gracias a su capacidad de generar código a partir de un sistema descrito a un nivel de abstracción más alto. Así, el usuario no debe enfocarse en calcular ecuaciones manualmente o describir el sistema en código C/C++.

3.5.1. Circuito RLC en ORTiS

ORTiS tiene una lista de componentes soportado por la herramienta, la cual se puede encontrar en el enlace al final del Apéndice A. Notablemente, **no existe**

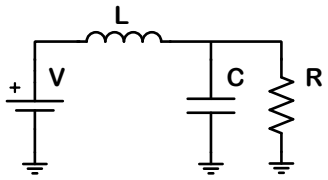


Figura 3.11: Circuito RLC a implementar

soporte para diodos ni transistores como componentes individuales, lo cual limita considerablemente la cantidad de topologías que se pueden diseñar manualmente.

De cualquier forma, se piensa validar su utilidad como herramienta. Para probar la generación de código a partir de una netlist, se empieza con un circuito RLC simple (Figura 3.11). La definición de la netlist (y por ende los parámetros de los componentes) queda de la siguiente manera:

```
#name RLC_Circuit
#const DT 1e-3
#const V  100.0
#const RV 0.001
#const R  10.0
#const L  1e-3
#const C  10.0e-3

% components
VoltageSource vs (V,RV) {1,0}
Resistor res (R) {2,0}
Inductor ind (DT,L) {1,2}
Capacitor cap (DT,C) {2,0}
```

3.5.1.1. Simulación previa

Utilizando LTSpice [5], una herramienta de simulación de electrónica análoga, se implementa el mismo circuito RLC para ser simulado y utilizar este resultado como referencia para el código por generar. La Figura 3.12 muestra esta simulación.

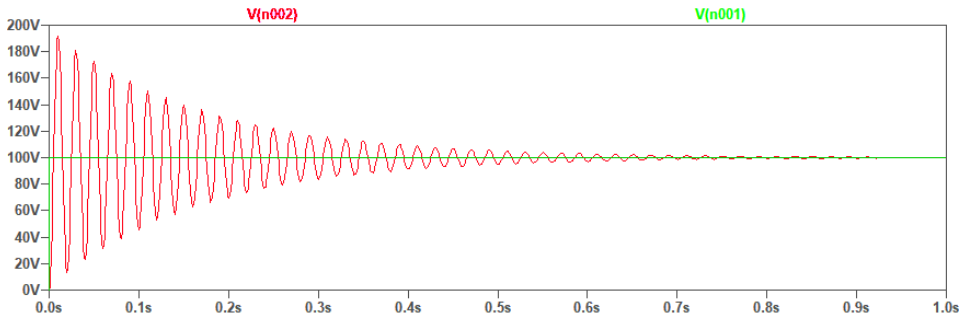


Figura 3.12: Resultado de simulación de circuito RLC en LTSpice

Con el comportamiento del circuito bien definido, se procede a la generación de código con ORTiS.

3.5.1.2. Generación de código C y segunda simulación

Al ejecutar ORTiS utilizando la netlist definida al inicio de esta subsección, se genera un archivo cabecera de C++. Este archivo contiene la definición de una función *template* que modela el sistema recientemente procesado, y se debe llamar desde una función "top-level" (esencialmente un *wrapper* que llama a la función template recién generada). Esta función top-level es la que se sintetiza finalmente por Vitis.

Al igual que en los casos anteriores, usando Vitis HLS se ejecuta una prueba unitaria compilando el código generado, guardando el resultado de la salida en un archivo de texto para ser graficado con MATLAB y posteriormente comparado con la simulación previa.

El resultado de esta segunda simulación se muestra en la Figura 3.13. El circuito se estimula con un escalón de voltaje de 100V (señal azul), generando el voltaje que se estabiliza en 100V en la carga (señal naranja).

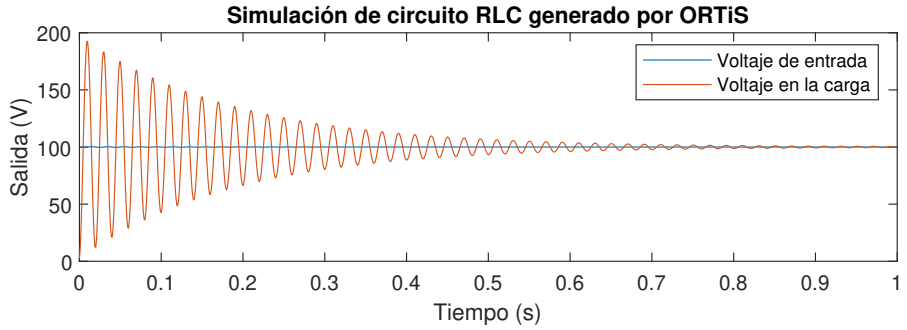


Figura 3.13: Resultado de simulación de circuito RLC de ORTiS implementado en C++, compilado desde el IDE de Vitis

El proceso de uso de ORTiS se encuentra en el Apéndice A, y los códigos y scripts de esta sección se encuentran en el Apéndice B.

3.5.1.3. Reporte de síntesis

- **Timing:**

Tabla 3.12: Estimación post-síntesis de timing del RLC de ORTiS

Objetivo	Estimado	Incertidumbre
10.00 ns	6.880 ns	2.70 ns

- **Rendimiento:**

Tabla 3.13: Estimación post-síntesis de rendimiento del RLC de ORTiS

Latencia (ciclos)	Latencia (ns)	Intervalo
7	70.000	8

- **Uso de recursos:**

Tabla 3.14: Estimación post-síntesis de uso de recursos del RLC de ORTiS

Instancias	DSP	FF	LUT
Utilizados	2	393	311
Disponibles	90	41600	20800
Utilizados (%)	2	1	1

Para timing y rendimiento, el sistema se sigue manteniendo muy por dentro de los límites. El uso de recursos, a pesar de que no es excesivo, se mantiene alrededor de lo que utilizaba el oscilador biquad *dual*.

Si bien se tienen recursos de sobra para manejar este incremento de recursos, el circuito RLC es extremadamente básico, y se esperaba que consumiera una cantidad menor.

3.5.1.4. Simulación RTL

Se crea un testbench para probar el módulo RTL recientemente generado. El sistema esta vez no recibe parámetros de entrada. La Figura 3.14 muestra el resultado de esta simulación.

Con esto, se puede asegurar que, en cuanto a funcionalidad, ORTiS funciona correctamente ya que se obtiene el comportamiento esperado del módulo generado.

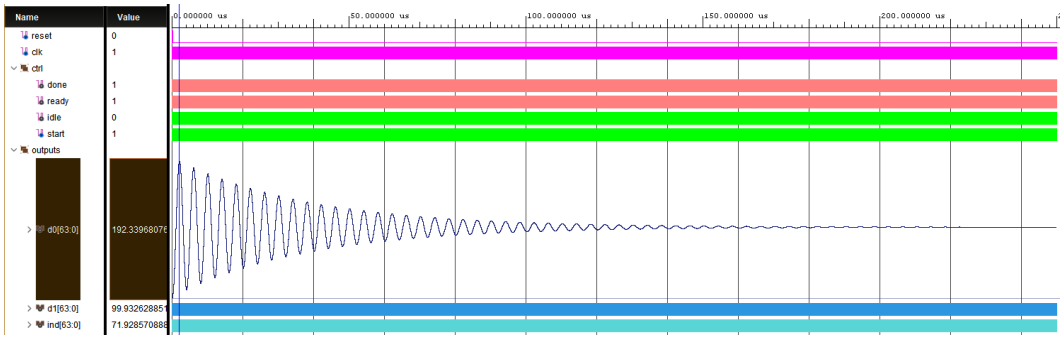
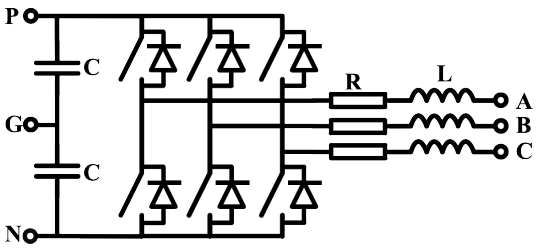


Figura 3.14: Simulación RTL con visualización análoga

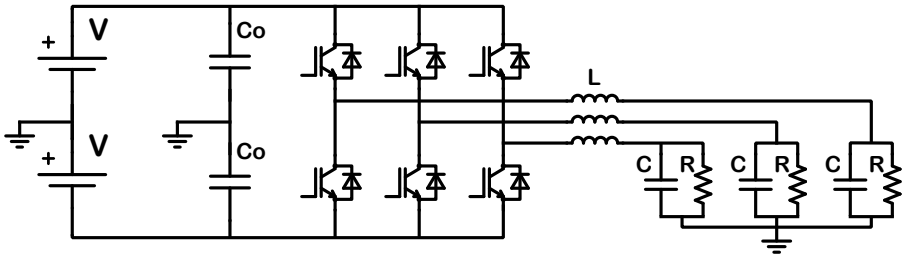
3.5.2. Inversor trifásico en ORTiS

Para esta sección solo se piensa estudiar la utilización de recursos de un circuito que utilice componentes más complejos de la lista que provee ORTiS, por lo que no se generarán simulaciones sino que solo reportes de síntesis.

El componente de la librería de ORTiS que se utilizará es el convertidor puente de tres piernas, mostrado en la Figura 3.15a. La topología del inversor trifásico que se sintetizará se muestra en la Figura 3.15b, y corresponde a un inversor trifásico.



(a) Componente de convertidor puente de 3 piernas de ORTiS



(b) Topología completa del inversor trifásico

Figura 3.15: Sistema de inversor trifásico en ORTiS

3.5.2.1. Reporte de síntesis

- **Timing:**

Tabla 3.15: Estimación post-síntesis de timing del inversor trifásico

Objetivo	Estimado	Incertidumbre
10.00 ns	7.020 ns	2.70 ns

- **Rendimiento:**

Tabla 3.16: Estimación post-síntesis de rendimiento del inversor trifásico

Latencia (ciclos)	Latencia (ns)	Intervalo
12	120.000	13

- **Uso de recursos:**

Tabla 3.17: Estimación post-síntesis de uso de recursos del inversor trifásico

Instancias	DSP	FF	LUT
Utilizados	66	2851	5385
Disponibles	90	41600	20800
Utilizados (%)	73	6	25

El uso de recursos para el circuito inversor trifásico utiliza alrededor del 20 % de lo disponible en la FPGA de la BRAIn. Si bien es un quinto de los recursos, esto de todos modos se puede considerar favorable, ya que un sistema de electrónica de potencia con aplicación real como este que pueda ser implementado con facilidad en una FPGA de baja gama es prometedor. Además, el timing y el rendimiento se siguen manteniendo fácilmente dentro de márgenes aceptables.

Desafortunadamente y a pesar de lo anterior, ORTiS es extremadamente limitante en cuanto a diseño de topologías, y al ser una herramienta de línea de comandos, el proceso de simulación es lento y propenso a errores. Junto a esto, el armado de circuitos es lento y tedioso al estar basado en una netlist de texto plano.

Por ser de código abierto, es posible que en un futuro se pueda volver mucho más viable si se implementan más componentes, pero aquel trabajo queda fuera del alcance de esta memoria.

3.6. Simscape y HDL Coder

Finalmente, se explora la posibilidad de generar el código HDL directamente, sin necesidad de pasar por la generación de código C para utilizar Vitis HLS.

Simscape es una herramienta de diseño que se ejecuta sobre el entorno de Simulink, y sirve para modelar y simular sistemas físicos de tipo eléctricos, mecánicos, hidráulicos, térmicos, y más. Permite al usuario crear modelos usando conexiones físicas en lugar de diagramas de bloques típicos. Más importante aún, **soporta una vasta variedad de componentes eléctricos**, en especial en comparación con ORTiS [63, 64].

HDL Coder es un add-on de MATLAB/Simulink para generar código HDL a partir de modelos de Simulink y funciones de MATLAB [65]. Desde la versión 2024 [1] de MATLAB, la herramienta "Simscape to HDL Workflow Advisor" (`sschdladvisor`) convierte modelos de Simscape a modelos de espacio de estados equivalentes en Simulink, para ser procesados por HDL Coder. También permite usar aritmética de punto fijo, reduciendo el uso de recursos en la FPGA.

En las subsecciones a continuación se implementan tres circuitos: el circuito RLC anterior (con ciertos parámetros distintos pero misma topología), un rectificador de media onda, y un puente rectificador de onda completa, todos construidos con componentes de la librería Simscape Electrical Foundation Library.

3.6.1. Circuito RLC en Simscape

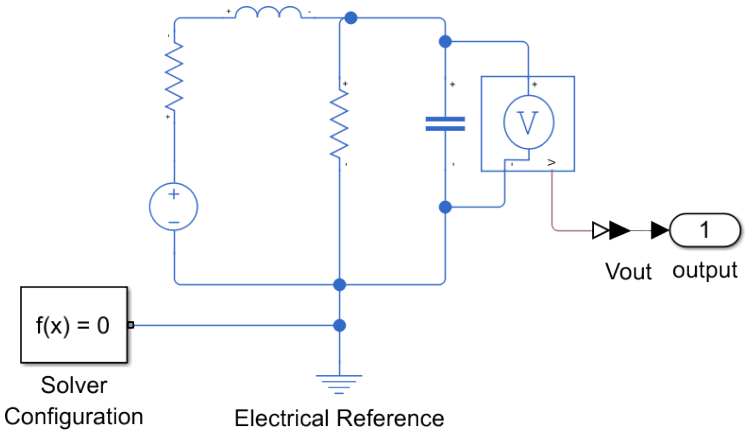
Utilizando la librería de Simscape se construye el circuito RLC de la Figura 3.16a. Para las señales que se desean observar en la salida, se agrega un sensor de voltaje/corriente y se envía la señal hacia fuera del sistema (por ejemplo, para ser observado por un bloque Scope de Simulink).

El bloque de Solver Configuration entrega información a las herramientas de simulación, como la frecuencia de muestreo que se usará para el circuito. (Nota: más información del uso de Simscape y HDL Coder se encuentra en la *application note* presentada en el capítulo siguiente).

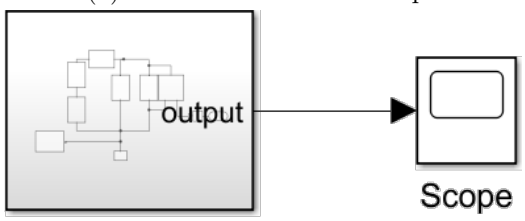
[1] Técnicamente desde la versión 2018, pero en junio de 2024 se arregló un bug donde, bajo varias condiciones, no se generaba el modelo de implementación

Los componentes tienen los siguientes valores:

- **Voltaje DC:** 100 V
- **Resistencia en serie:** 0,001 Ω
- **Inductor:** 25 mH
- **Condensador:** 47 mF
- **Resistencia en paralelo:** 10 Ω



(a) Circuito RLC en Simscape



(b) Subsistema RLC conectado a Scope

Figura 3.16: Circuito RLC de Simscape en el entorno Simulink

3.6.1.1. Simulación previa

Utilizando el entorno Simulink se puede simular el comportamiento de un circuito construído en Simscape. La Figura 3.16b muestra el subsistema RLC conectado a un bloque Scope, y la Figura 3.17 muestra el resultado de la simulación del circuito RLC.

Con el comportamiento del circuito bien definido, se procede a utilizar HDL Coder para la generación de código RTL.

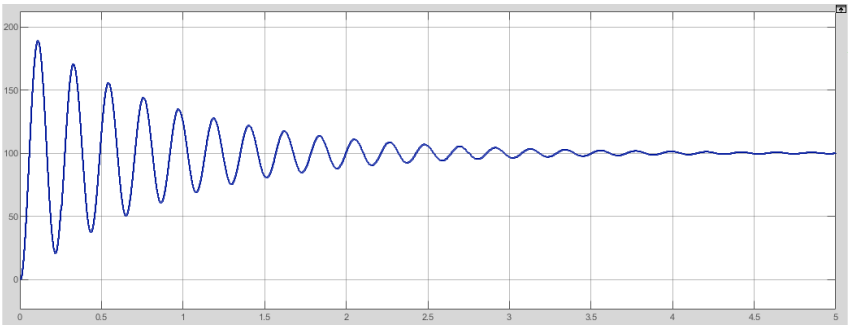


Figura 3.17: Simulación de circuito RLC de Simscape

3.6.1.2. Reporte de generación de código

HDL Coder también genera reportes a partir de la generación de código, incluyendo estimaciones de uso de recursos, reloj, puertos del módulo, etc. La Figura 3.18 muestra los reportes de uso estimado de recursos, y la Figura 3.19 los de reloj.

Multipliers	9
Adders/Subtractors	7
Registers	9
Total 1-Bit Registers	134
RAMs	0
Multiplexers	5
I/O Bits	36
Static Shift operators	0
Dynamic Shift operators	0

Figura 3.18: Reporte de uso de recursos de RLC de Simscape

Rate Information

Model Base Rate	1e-08
DUT Base Rate	0.001
Oversampling Factor	100000

Clock Enable Table

Clock Enable Name	Sample Time
ce_out	0.001

Output Signal Table

Output Signal	Clock Enable Name	Sample Time	Latency
PS_Simulink_Converter	ce_out	0.001	0

Figura 3.19: Reporte de reloj de RLC de Simscape

Tabla 3.18: Comparación de uso de recursos entre RLC de ORTiS vs Simscape

Herramienta	DSP	FFs y LUTs
ORTiS	2	704
Simscape	9	143

Como se puede ver, el sistema ocupa una cantidad menor a aquel generado por ORTiS, utilizando alrededor de 150 registros, en comparación con los más de 700 registros del circuito RLC de ORTiS, incluso tomando en cuenta que tienen el mismo paso de simulación. Sin embargo, la implementación de Simscape utiliza más bloques DSP (multiplicadores).

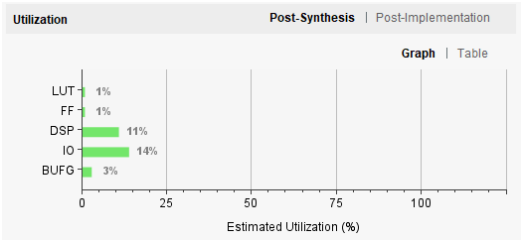
3.6.1.3. Simulación RTL y reporte de Vivado

Se crea un *top module* que encapsule el módulo generado, y luego se crea un testbench para hacer su simulación RTL. En la Figura 3.20 se puede ver el reporte de utilización de recursos de Vivado post-síntesis, y en la Figura 3.21 el resultado de la simulación RTL.

Se observa que el uso de recursos del modelo RTL generado por HDL Coder es mucho más compacto en recursos que aquel de ORTiS. Además, la simulación muestra que el comportamiento del sistema sigue siendo el esperado puesto que coincide con aquel de la simulación en Simulink.

3.6.2. Rectificador de media onda en Simscape

Para corroborar la viabilidad de Simscape y HDL Coder como alternativa de generación de código, se procede a modelar un sistema más complejo y que utilice



(a) Gráfico de uso de recursos

Resource	Estimation	Available	Utilization %
LUT	167	20800	0.80
FF	83	41600	0.20
DSP	10	90	11.11
IO	36	250	14.40
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 3.20: Uso de recursos de circuito RLC de Simscape en Vivado

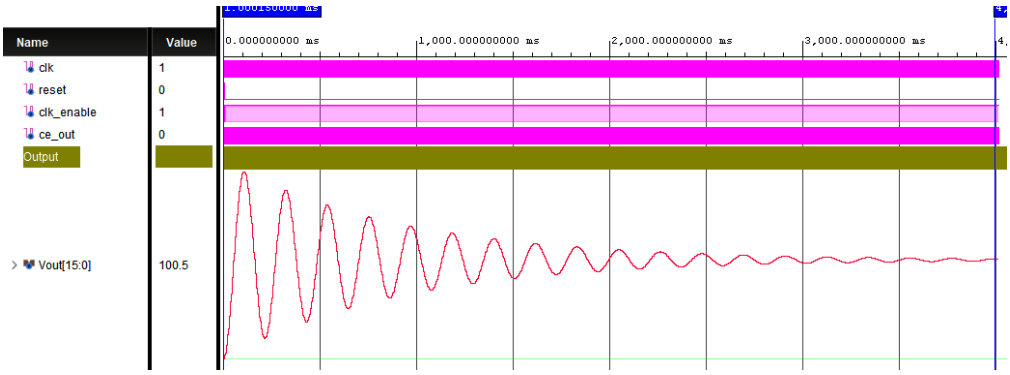


Figura 3.21: Simulación RTL de circuito RLC de Simscape

componentes no lineales como diodos.

La Figura 3.22 muestra el circuito de un simple rectificador de media onda implementado en Simscape, a partir del cual se va a generar un módulo RTL. El sistema recibe como entrada una onda sinusoidal de 10 kHz de frecuencia y 10 V de amplitud, y se monitorea el voltaje en el diodo (voltaje directo = 0.3 V), el voltaje en la resistencia (1 Ω), y la corriente de salida.

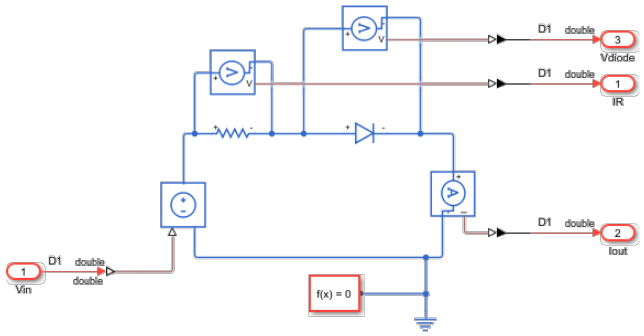


Figura 3.22: Rectificador de media onda en Simscape

3.6.2.1. Simulación previa

Al ejecutar la simulación del sistema entregándole la señal sinusoidal anteriormente mencionada, se obtiene el resultado mostrado en la Figura 3.23, donde se pueden apreciar las tres señales de salida.

Una vez el comportamiento del circuito se encuentra bien definido, se procede a la generación de código con HDL Coder.

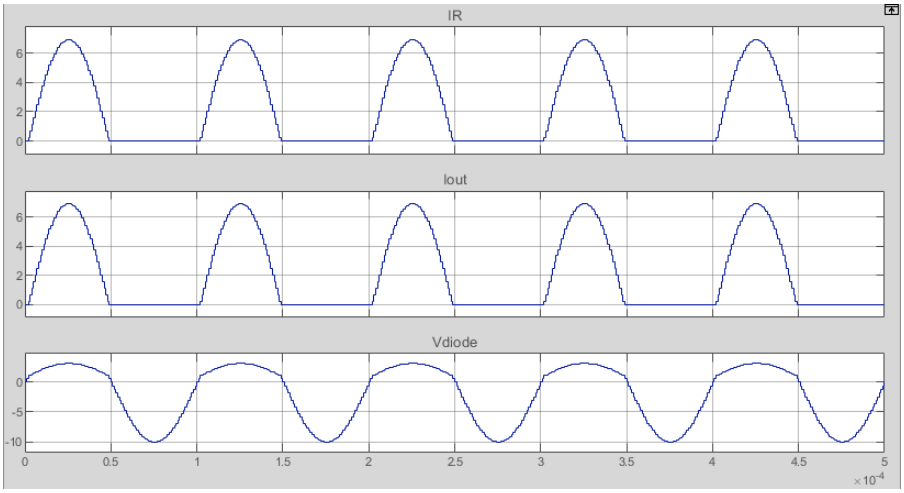


Figura 3.23: Simulación previa del rectificador de media onda de Simscape

3.6.2.2. Reporte de generación de código

En la Figura 3.24 se puede ver el reporte de reloj, y en la Figura 3.25 el reporte de utilización estimada de recursos.

Rate Information

Model Base Rate	1e-08
DUT Base Rate	1e-06
Oversampling Factor	100

Clock Enable Table

Clock Enable Name	Sample Time
ce_out	1e-06

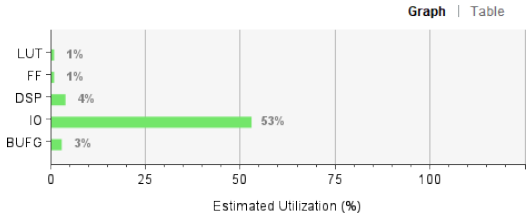
Figura 3.24: Reporte de reloj de rectificador de Simscape

Multipliers	6
Adders/Subtractors	9
Registers	8
Total 1-Bit Registers	82
RAMs	0
Multiplexers	4
I/O Bits	132
Static Shift operators	0
Dynamic Shift operators	0

Figura 3.25: Reporte de uso de recursos de rectificador de Simscape

3.6.2.3. Simulación RTL y reporte de Vivado

Al igual que para el circuito RLC, se crea un *top module* y un testbench para obtener el reporte de uso de recursos post síntesis (Figura 3.26) y hacer la simulación RTL (Figura 3.27) respectivamente. Para el testbench se genera un arreglo de valores que representen la señal sinusoidal de entrada al módulo generado.



(a) Gráfico de uso de recursos

Resource	Estimation	Available	Utilization %
LUT	304	20800	1.46
FF	75	41600	0.18
DSP	4	90	4.44
IO	132	250	52.80
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 3.26: Uso de recursos de rectificador de Simscape en Vivado

Observación: el uso de recursos IO se puede ignorar para efectos de esta memoria, ya que Vivado interpreta preliminarmente que las entradas y salidas del módulo vendrán de los pines físicos de la FPGA, cuando en verdad vendrán de registros de memoria controlables por software desde el DSP de la BRAIn.

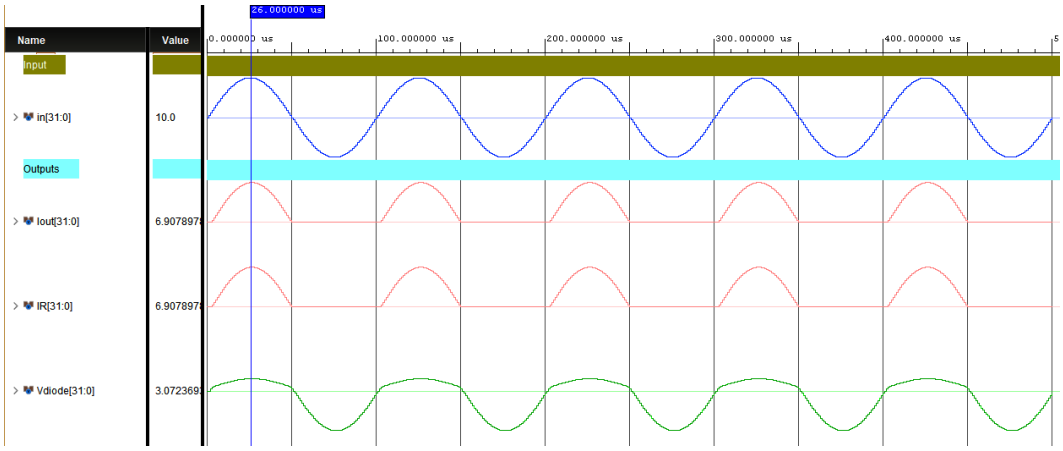


Figura 3.27: Simulación RTL de rectificador de media onda

Como se aprecia a partir de estos datos, la utilización de recursos se sigue manteniendo en niveles bastante bajos (menos del 2 % para FFs y LUTs, 10 % de DSPs), y la simulación RTL demuestra que el módulo HDL funciona correctamente, coincidiendo su comportamiento con aquel de la simulación de Simulink.

3.6.3. Puente rectificador de onda completa en Simscape

Por último, se decide hacer una última prueba con un circuito que implemente una topología más compleja junto con algún otro elemento no lineal, por lo que se decide modelar un puente rectificador de onda completa.

La Figura 3.28 muestra el circuito completo en Simscape. El sistema recibe una onda sinusoidal de 120 V_{RMS} y frecuencia de 60 Hz, y en la salida se ve un voltaje peak de aproximadamente 11 V.

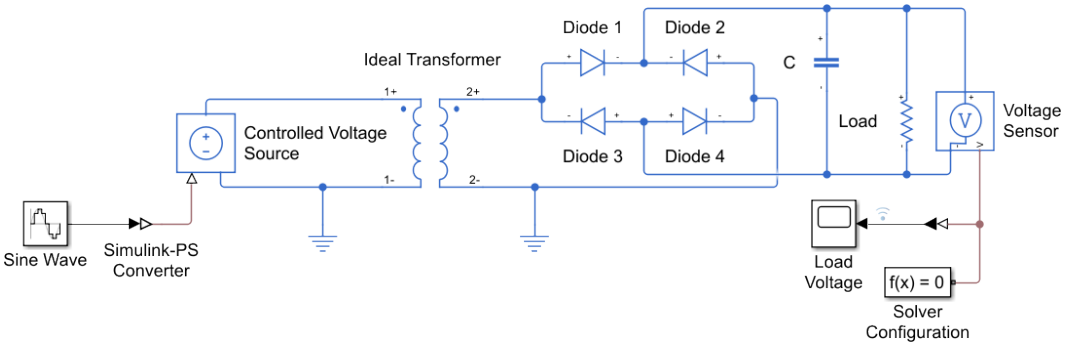


Figura 3.28: Puente rectificador de onda completa en Simscape

3.6.3.1. Simulación previa

Ejecutando la simulación de Simulink, se obtiene el resultado de la Figura 3.29. En esta ocasión solo se observa el voltaje de la carga en la salida.

Teniendo bien definido el comportamiento esperado del sistema, se procede a la generación de código HDL con HDL Coder.

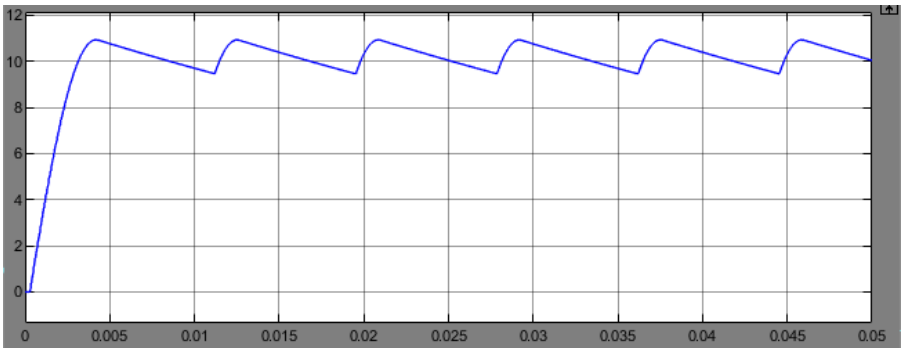


Figura 3.29: Simulación previa del puente rectificador de Simscape

3.6.3.2. Reporte de generación de código

En la Figura 3.30 se puede ver el reporte de reloj, y en la Figura 3.31 el reporte de utilización estimada de recursos.

Rate Information

Model Base Rate	1e-08
DUT Base Rate	5e-06
Oversampling Factor	500

Clock Enable Table

Clock Enable Name	Sample Time
ce_out	1e-05

Figura 3.30: Reporte de reloj de puente rectificador de Simscape

Multipliers	18
Adders/Subtractors	34
Registers	22
Total 1-Bit Registers	404
RAMs	0
Multiplexers	30
I/O Bits	68
Static Shift operators	0
Dynamic Shift operators	0

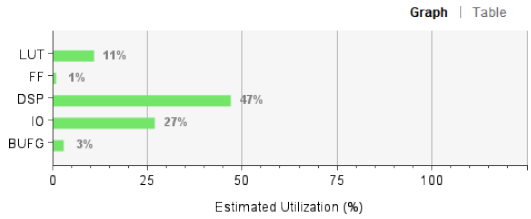
Figura 3.31: Reporte de uso de recursos de puente rectificador de Simscape

El sistema se ha complejizado en comparación con el rectificador de media onda, gracias al transformador añadido y la mayor cantidad de diodos, aunque sigue habiendo un bajo nivel estimado de recursos en comparación con el RLC de ORTiS: un total de aproximadamente 8 % de los registros, y 30 % de bloques DSP (multiplicadores).

3.6.3.3. Simulación RTL y reporte de Vivado

Nuevamente se crea un *top module* y un testbench para ver el reporte de utilización de recursos post-síntesis (Figura 3.32) y para realizar la simulación RTL (Figura 3.33), respectivamente. Para este testbench también se genera un arreglo que represente la onda sinusoidal de 60 Hz de la entrada.

Con un sistema considerablemente más complejo, el uso de recursos de FFs y



(a) Gráfico de uso de recursos

Resource	Estimation	Available	Utilization %
LUT	2358	20800	11.34
FF	282	41600	0.68
DSP	42	90	46.67
IO	68	250	27.20
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 3.32: Uso de recursos de puente rectificador de Simscape en Vivado

LUTs se mantiene alrededor del 10 %, aunque el uso de bloques DSP aumentó casi al doble.

Además, la simulación RTL muestra que el sistema se comporta de la manera esperable, demostrando así que Simscape y HDL Coder son una opción viable para el modelado de sistemas de electrónica de potencia.

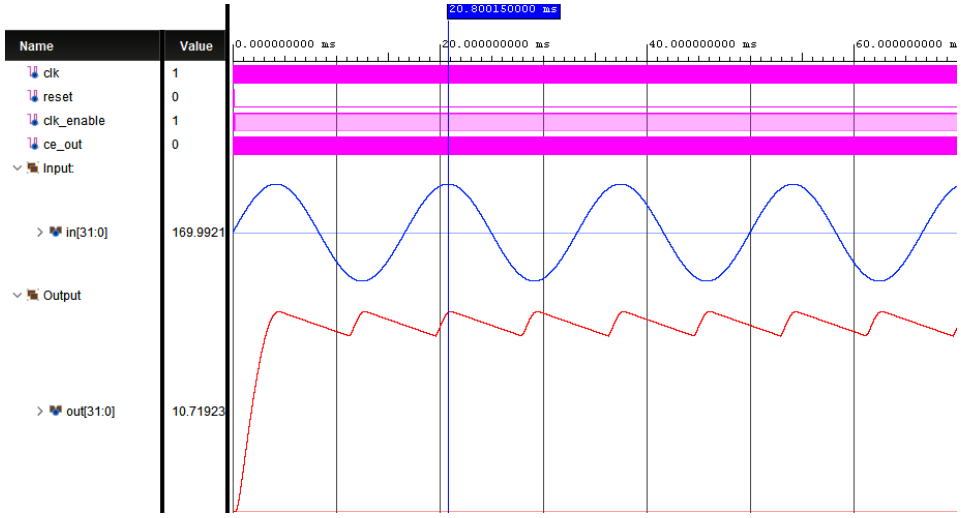


Figura 3.33: Simulación RTL de puente rectificador de onda completa

3.7. Conclusiones

La exploración de diversos métodos para generar código HDL para sistemas electrónicos de potencia deja en evidencia la complejidad de este campo.

Inicialmente, la combinación de Vitis HLS con Simulink se presentó como una buena opción para la implementación de modelos de sistemas de tiempo discreto en

HDL. La interfaz visual de Simulink y el código C/C++ generado por éste resulta en modelos funcionales y bien optimizados. La desventaja de este enfoque es que no es una opción efectiva cuando se desea implementar sistemas de electrónica de potencia más complejos y prácticos; en efecto, el usuario debe modelar manualmente el circuito a partir de un análisis matemático de éste.

El uso de ORTiS + Vitis mostró su potencial para generar código C++ a partir de una representación de alto nivel de circuitos electrónicos, permitiendo al usuario centrarse en el diseño del sistema mismo. Aunque ORTiS genera modelos funcionales, su rigidez en la creación de topologías personalizadas y su mayor consumo de recursos lo hicieron inadecuado para nuestra aplicación. Sin embargo, como herramienta open-source, podría ser útil en el futuro para quienes la librería de componentes de ORTiS le sea suficiente, o quienes deseen intentar implementar sus propios componentes.

Finalmente, la combinación de Simscape y HDL Coder emerge como la opción más viable para la generación de código HDL a partir de esquemáticos de alto nivel. Este *toolchain* provee un entorno flexible para el diseño de sistemas de electrónica de potencia y varias opciones para optimizar el código generado tanto en uso de recursos como en rendimiento, reduciendo significativamente el esfuerzo de codificación manual y manteniendo la eficiencia en el uso de los recursos de la FPGA. No obstante, a pesar de su efectividad, estas herramientas requieren licencias, lo cual puede llegar a limitar su accesibilidad para algún usuario o instituto.

En resumen, esta evaluación de alternativas demostró las fortalezas y debilidades de varios *toolchains* para la generación de código HDL, donde si bien Simulink y ORTiS tienen gran potencial en ciertos contextos, éstos resultan insuficientes para cumplir con las demandas de la aplicación objetivo de esta memoria. La combinación de Simscape con HDL Coder provee la forma más robusta y escalable para este propósito, y estos resultados preparan el terreno para el siguiente capítulo, el cual involucra la implementación del modelo de convertidores de potencia en la plataforma BRAIn.

Capítulo 4

MODELO DE CONVERTIDORES DE ELECTRÓNICA DE POTENCIA EN BRAIN

Con la exploración de varias herramientas para generar código HDL en el capítulo anterior, este capítulo se enfoca en la implementación práctica de modelos de convertidores en BRAIn. La meta es demostrar que el flujo de trabajo y las herramientas discutidas anteriormente pueden ser aplicadas a un ejemplo real, específicamente un boost o buck converter, en la FPGA de una plataforma específica.

El capítulo primero indaga en las especificaciones de los convertidores de potencia, detallando su estructura y propósito. También se hace una mirada a las especificaciones técnicas de BRAIn, en particular cómo su integración del sistema DSP/FPGA la vuelve una solución ideal para simulación *hardware-in-the-loop* (HIL).

Además, se detallará el flujo de trabajo de Simscape y HDL Coder, junto con las decisiones de diseño que permitan que el sistema funcione correctamente a nivel de FPGA. Este resultado finalmente se documentará y concretará en la forma de una *application note*, presentando el caso de aplicación anteriormente descrito de manera comprensiva.

4.1. Buck y Boost Converter

Para la implementación en BRAIn se han considerado dos convertidores de potencia dc-dc fundamentales, con topologías simples y similares entre sí. Estos convertidores corresponden al *buck converter* y el *boost converter*.

Ambos circuitos son un ejemplo de fuentes conmutadas controladas por PWM, y son esenciales para varios sistemas de potencia gracias a su capacidad de aumentar o disminuir el voltaje de la entrada con un alto grado de eficiencia. [36]

Por lo general, un convertidor PWM, como un boost o buck, contiene cuatro componentes además de la carga, modelada por una resistencia R:

- Un switch (usualmente en la forma de un MOSFET) que controla el flujo de energía entre la fuente y la carga.
- Un diodo rectificador.
- Un inductor.
- Un capacitor de filtro.

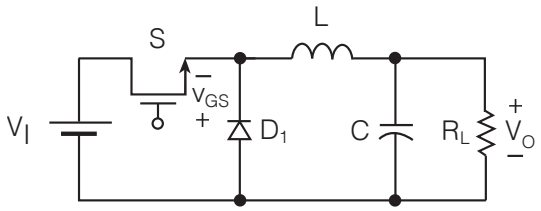
Para efectos de modelación y simulación HIL por FPGA, los switches serán considerados ideales. Esto a raíz de que el trabajo realizado se enfoca en el comportamiento del sistema como un todo, y no en el comportamiento de cada componente.

4.1.1. Buck Converter

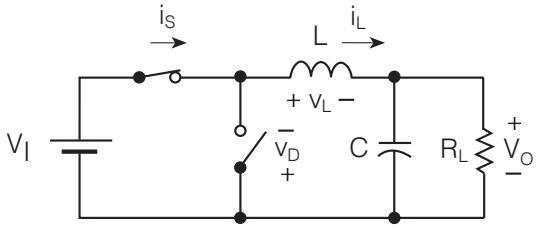
Un buck converter es un convertidor dc-dc cuyo voltaje en la salida es siempre menor que el voltaje en la entrada en estado estacionario. Este convertidor *reduce* el voltaje entregado. La Figura 4.1a muestra la topología completa del circuito. [66]

El switch y el diodo forman esencialmente un switch de un polo y doble vía, ya que al conducir uno el otro se encuentra apagado (Figuras 4.1b y 4.1c). El capacitor e inductor se encargan de almacenar y transferir energía, y la red de conmutación (switch + diodo) "corta" el voltaje de la entrada, causando un voltaje promedio menor que V_I.

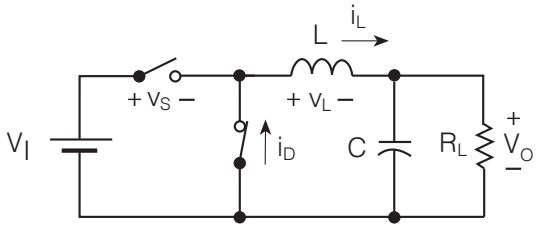
El voltaje de salida V_O queda definido por la ecuación:



(a) Circuito del buck converter



(b) Circuito equivalente cuando el switch está encendido y el diodo no conduce



(c) Circuito equivalente cuando el switch está apagado y el diodo conduce

Figura 4.1: Buck Converter

$$V_O = D \cdot V_I \tag{4.1}$$

donde D es el ciclo de trabajo de la señal PWM que controla la red de conmutación.

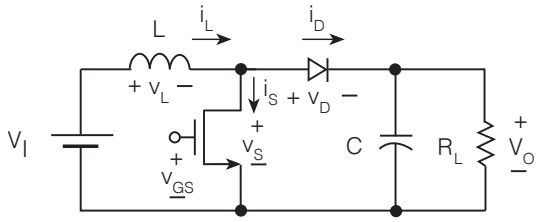
4.1.1.1. Parámetros del buck converter

Para la implementación de BRAIn, se creará un modelo de buck converter que convierta **12V a 5V**. Los parámetros de cada componente del circuito son los siguientes:

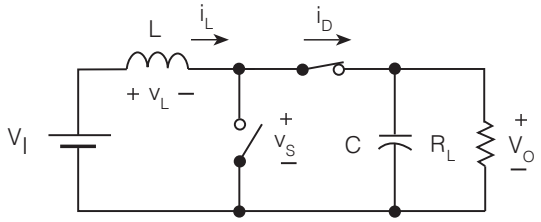
- $R = 10\ \Omega$
- $L = 100\ \mu$H
- $C = 1000\ \mu$F

- Voltaje directo del diodo: $V_D = 0{,}5$ V
- Periodo de conmutación del switch: $t = 40\ \mu$s
- Ciclo de trabajo del switch: $D = 40\,\%$

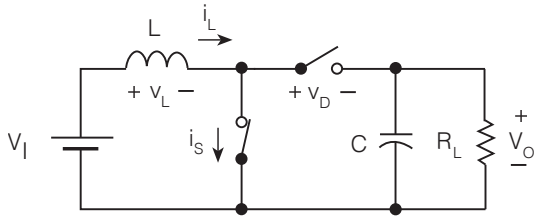
4.1.2. Boost Converter



(a) Circuito del boost converter



(b) Circuito equivalente cuando el switch está apagado y el diodo conduce



(c) Circuito equivalente cuando el switch está encendido y el diodo no conduce

Figura 4.2: Boost Converter

Un boost converter es un convertidor dc-dc cuyo voltaje en la salida es siempre mayor que el voltaje en la entrada en estado estacionario. Este convertidor *aumenta* el voltaje entregado. La Figura 4.2a muestra la topología completa del circuito. [67]

El switch y el diodo, al igual que con el buck converter, forman un switch de un polo y doble vía: cuando el diodo conduce, el switch se encuentra apagado (Figura 4.2b) y viceversa (Figura 4.2c). El inductor almacena energía en la forma de un campo magnético cuando el switch conduce, y libera esa energía a través del diodo

cuando el switch no conduce. Esto en conjunto con el voltaje de la entrada causa que el capacitor se cargue con un voltaje mayor a V_I.

El voltaje de salida V_O queda definido por la ecuación:

$$V_O = \frac{V_I}{1 - D} \tag{4.2}$$

donde D es el ciclo de trabajo de la señal PWM que controla la red de conmutación.

4.1.2.1. Parámetros del boost converter

Para la implementación de BRAIn, se creará un modelo de boost converter que convierta **24V a 48V**. Los parámetros de cada componente del circuito son los siguientes:

- $R = 50\ \Omega$
- $L = 2500\ \mu$H
- $C = 2\ \mu$F
- Voltaje directo del diodo: $V_D = 0{,}5$ V
- Periodo de conmutación del switch: $t = 20\ \mu$s
- Ciclo de trabajo del switch: $D = 50\,\%$

4.2. Hardware de BRAIn para simulaciones en tiempo real y HIL

La plataforma de control en tiempo real BRAIn, como se ha mencionado anteriormente, es una tarjeta de procesamiento de señales digitales que integra tanto una FPGA como un DSP, y está diseñada específicamente para el control en tiempo real. El DSP es el principal elemento de procesamiento de datos, y la FPGA se encarga de implementar en hardware una serie de módulos y funciones complementarias al DSP, sirviendo como su coprocesador/acelerador esclavo.

Esta arquitectura dual vuelve a BRAIn una herramienta adecuada para modelos

hardware-in-the-loop, donde la FPGA simula digitalmente el comportamiento de un sistema físico en tiempo real, y el DSP interactúa con ella como si se tratara del hardware verdadero. De este modo, los usuarios se pueden enfocar en la programación del DSP y se abstraen de la implementación de la planta (convertidor) en la FPGA.

4.2.1. Especificaciones de la XC7A35T

La FPGA incorporada en la tarjeta BRAIn es la XC7A35T de AMD (previamente Xilinx), específicamente el encapsulado `xc7a35t-fgg484-2`. Este dispositivo es parte de la serie Artix-7, una serie de FPGAs consideradas económicas [68], además de situarse en la mitad de más baja gama en esa serie[1]. Esto implica una posible limitación en la complejidad de los sistemas que se pueden modelar en BRAIn, resaltando la importancia de tener la información a continuación presente al momento de generar código HDL.

Para los reportes de generación de código del capítulo anterior ya se ha hecho alusión a la cantidad de recursos disponibles en el dispositivo; de todos modos, las especificaciones relevantes [69] se detallan en la Tabla 4.1.

Tabla 4.1: Resumen de recursos de la FPGA XC7A35T

Recurso	**Cantidad**
Celdas lógicas	33280
Flip-Flops	41600
Bloques DSP	90
Block RAM	1800 Kb

La frecuencia de reloj máxima soportada por la XC7A35T-2 es de 550 MHz [69], pero en BRAIn la frecuencia de reloj está configurada a **100 MHz**. Es importante mantener esto presente, ya que los sistemas diseñados en Simscape requieren definir un paso de simulación, y si éste es relativamente pequeño y el sistema tiene cierto grado de complejidad, pueden llegar a ocurrir problemas de timing.

El reloj de 100 MHz es una decisión deliberada. Si bien existe la libertad de implementar múltiples dominios de reloj en distintas áreas de la FPGA, la BRAIn

[1] Aunque el costo de solo ese chip FPGA es, a la fecha, de unos $80 USD precio lista.

no explota esta característica, por simplicidad.

En la *application note* (vinculado al final de este capítulo) se puede encontrar un anexo en el que se postulan maneras de arreglar estos problemas de *timing*, en caso de que el usuario se llegue a encontrar con alguno.

4.2.2. Integración de módulos y comunicación DSP-FPGA

Como se ha explicado antes, la tarjeta BRAIn implementa un sistema de cómputo dual DSP-FPGA, mostrado en la Figura 4.3 [70]. La comunicación entre ambos dispositivos se realiza mediante una interfaz de memoria externa (EMIF) presente en el TMS320C6748, que dispone de un bus de datos de 16 bits de ancho, y un módulo de firmware en la FPGA ("ace_emif") que se encarga de definir direcciones de memoria con las que el bus de datos puede intercambiar información.

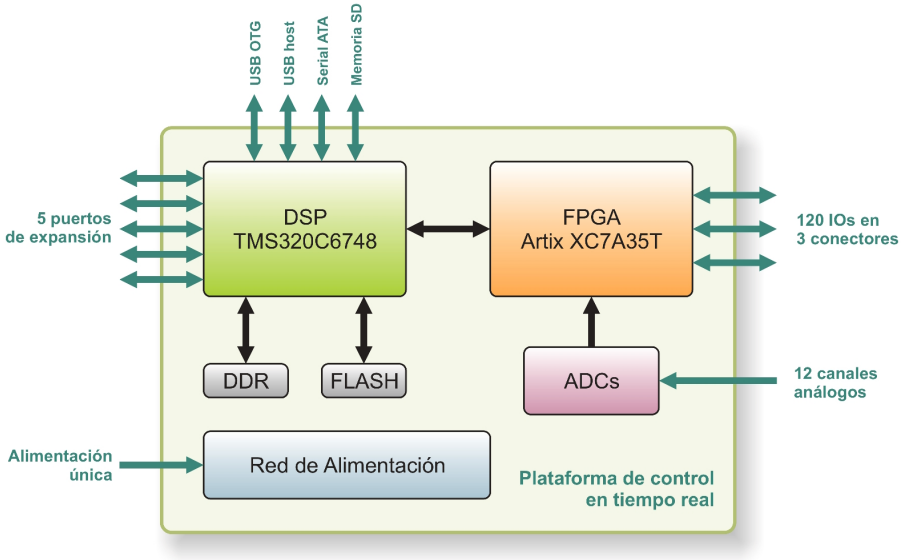


Figura 4.3: Diagrama de bloques de la plataforma BRAIn

Para poder implementar nuevos módulos en la FPGA de la BRAIn [71], es necesario actualizar "ace_emif" en el firmware para agregar los nuevos registros de memoria del módulo, que corresponden a sus puertos de entradas y salidas, de modo que puedan ser controlados desde el DSP mediante punteros. El repositorio del firmware FPGA de BRAIn contiene scripts TCL, que actualizan el módulo "ace_emif" en base a una lista de los registros que se usarán.

Posteriormente, el módulo se debe instanciar en el *top module* del proyecto FPGA BRAIn, conectando sus puertos con los registros correspondientes. Gracias a este método, las entradas y salidas del módulo son controladas digitalmente en lugar de físicamente, evitando utilizar los pines IO de la FPGA.

Los detalles de este proceso de implementación se pueden encontrar en la *application note*, a la cual se hace referencia al final de este capítulo.

4.3. Decisiones de diseño y restricciones

En esta sección se listan decisiones y restricciones de diseño importantes para los circuitos de boost converter y buck converter que se desean implementar. No pretende ser una lista exhaustiva, sino una lista a modo de referencia que contenga las consideraciones más relevantes.

- **Periodo de muestreo: 1 μs**
  En ambos circuitos se encuentran presentes señales de pulsos que controlan la frecuencia de conmutación de los switches; para el buck converter esta señal tiene un periodo de 40 μs, y para el boost converter un periodo de 20 μs. Para asegurar que esta señal PWM (una onda cuadrada) sea suficientemente bien muestreada, se escoge un paso de simulación de 1 μs, un orden de magnitud más rápido que el periodo de la señal; para el boost converter, se preservan 5 armónicos de la PWM.

- **Tipo de aritmética de precisión: punto fijo**
  Este tema fue tocado en el capítulo anterior, enfocado en cómo utilizar punto fijo ahorra una gran cantidad de recursos de la FPGA. Además, utilizar aritmética de punto fijo no implica una pérdida significativa de precisión; dado que estos sistemas operan con señales de voltajes que abarcan un amplio rango dinámico (tanto por sí mismas como entre señales), la precisión adicional ofrecida por la aritmética de punto flotante no es esencial para su funcionamiento correcto. Esto fue avalado por los resultados de las simulaciones realizadas en el capítulo 3.

- **Frecuencia de reloj máxima de la FPGA: 100 MHz**
  Se debe tener presente que la FPGA de BRAIn opera a 100MHz, equivalente

a un periodo de 10 ns, por lo que los diseños deben evitar realizarse con periodos de muestreo innecesariamente pequeños, en especial para sistemas más complejos que requieran más capacidad de cómputo.

- **Pipelining:**
  En caso de detectarse problemas de timing (ej. slack negativo, retardo de propagación), se aplica pipelining en las configuraciones de generación de código. Esto hará que las herramientas utilicen registros adicionales para combatir estos problemas de timing, y un correspondiente aumento de la latencia en ciclos de reloj.

- **Cantidad de señales observables/controlables:**
  Los puertos de entradas/salidas de los módulos de la FPGA se conectan a registros que son accedidos por la EMIF del DSP. La EMIF utiliza un espacio de memoria de 8Mb para comunicarse con la FPGA, por lo que en la práctica el límite de señales observables y controlables del módulo HDL se encuentra muy por sobre lo que se necesita en la realidad. Sin embargo, un parámetro observable adicional supone de todos modos una carga extra a la FPGA en cuanto a uso de recursos.

4.4. Flujo de Simscape a código HDL

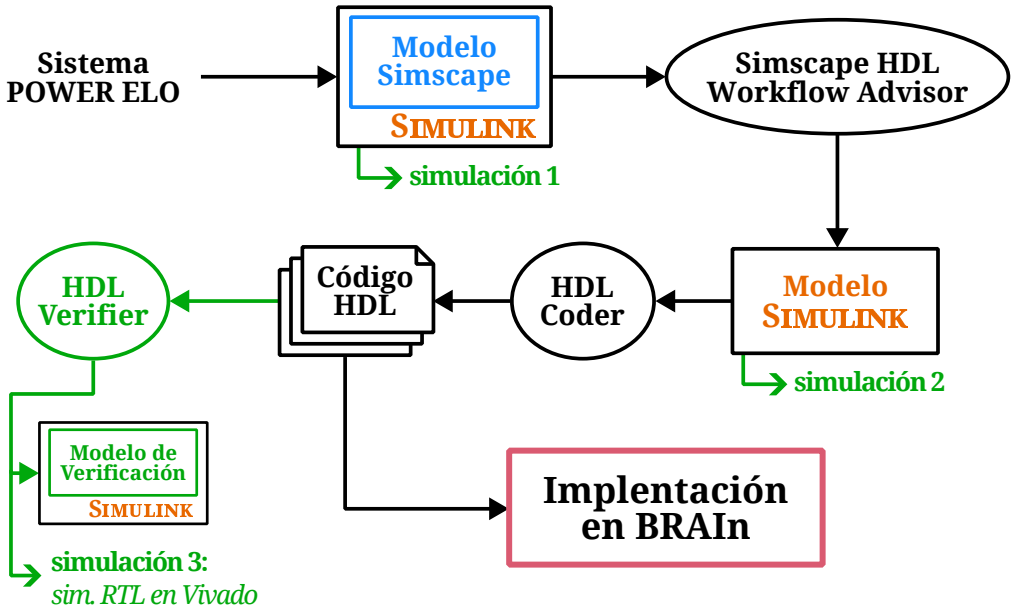


Figura 4.4: Flujo de Simscape

El proceso de implementación del modelo HDL de un sistema de electrónica de potencia a BRAIn, siguiendo el método decidido en el capítulo 3, requiere las siguientes herramientas de software:

- MATLAB y Simulink
- Simscape, de Simulink
- HDL Coder, add-on para Simulink
- HDL Verifier, add-on complementario a HDL Coder

El flujo de trabajo para la generación de código RTL se muestra en la Figura 4.4. El diagrama de esta figura utiliza verde para distinguir pasos o procedimientos que no son técnicamente necesarias para la generación de código como tal, pero esto no quiere decir que no sean importantes (muy por el contrario). A partir de este diagrama, se subdivide el proceso en etapas, las cuales se explican a continuación.

Los detalles completos de cada paso se pueden encontrar en la *application note*. Esta sección tiene como fin hacer una vista general del flujo de trabajo y justificar ciertas decisiones tomadas durante el proceso.

4.4.1. Diseño del sistema de electrónica de potencia en Simscape

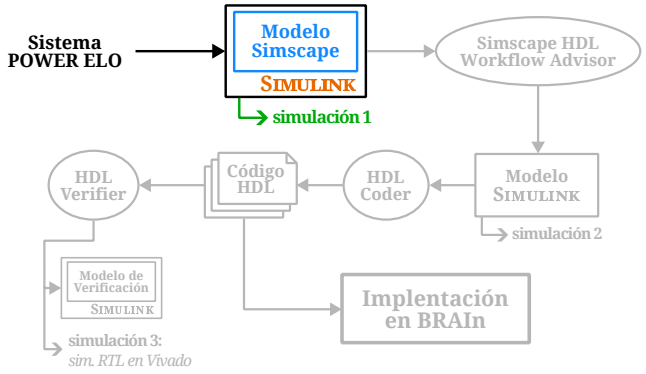


Figura 4.5: Pasos del diseño del circuito en Simscape

En esta etapa, se hace uso de la herramienta Simscape para diseñar el circuito que se desea implementar.

Dado que Simscape es un complemento de Simulink, es posible controlar las entradas y observar las salidas del circuito con bloques de Simulink; esto permite realizar una simulación para validar el comportamiento del circuito, tal como se haría en un programa de simulación de circuitos (como LTSpice).

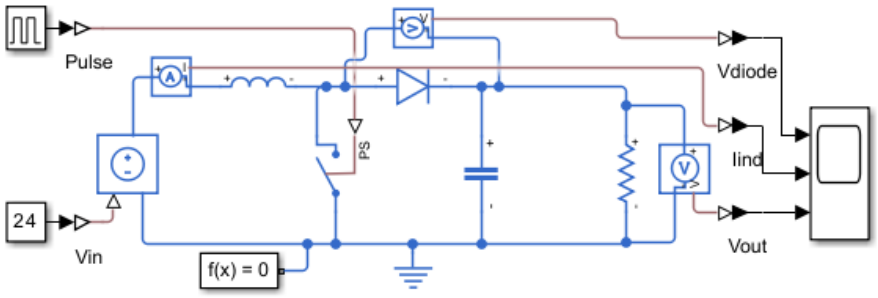


Figura 4.6: Circuito del Boost Converter en Simscape con entradas y salidas externas.

Aquí, el usuario debe definir qué parámetros serán controlables por el usuario, y qué parámetros del circuito se podrán leer, esto último haciendo uso de elementos como amperímetros y voltímetros (elementos que son parte de la librería de Simscape Electrical Foundation Library), como se muestra en la Figura 4.6.

Además, en esta etapa se debe definir un paso de simulación, o periodo de

muestreo, apropiado. Tal como se comentó en la sección 4.3, lo recomendable es fijar un paso de simulación al menos un orden de magnitud mayor que la frecuencia de Nyquist del sistema. Esto se puede configurar con el bloque de "Solver Configuration" (en la Figura 4.6, el bloque que dice `f(x) = 0`).

4.4.2. Conversión del sistema con Simscape-HDL Workflow Advisor

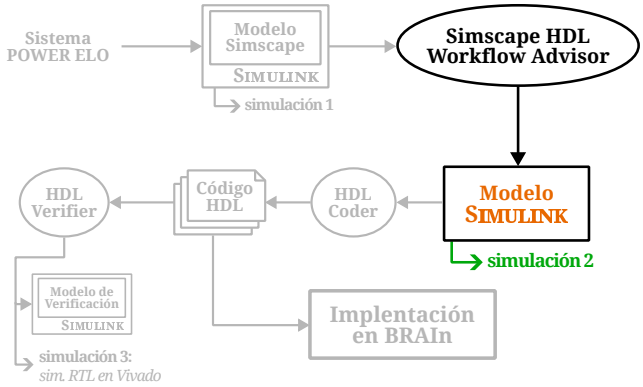


Figura 4.7: Pasos de la conversión del sistema Simscape a Simulink

La herramienta "Simscape to HDL Workflow Advisor" permite convertir el circuito de Simscape a su equivalente de espacio de estados de Simulink, es decir, solo utilizando bloques de Simulink en lugar de la librería de componentes de Simscape. Esto es debido a que HDL Coder no soporta directamente los componentes físicos de Simscape. El modelo resultante se denomina "modelo de implementación".

Aquí, el usuario debe definir la plataforma FPGA objetivo (en este caso, el paquete `xc7a35tfgg484-2`), junto con la resolución en bits de los tipos de datos de punto fijo. Al generar el modelo de implementación, se debe realizar una segunda simulación en Simulink, para corroborar que el sistema no ha sufrido un cambio de comportamiento.

Ya que el bus de datos del EMIF del DSP de BRAIn tiene una resolución de 16 bits, lo conveniente sería que la resolución de los datos de punto fijo del sistema también sea de 16 bits. Sin embargo, es posible que esta resolución sea insuficiente dependiendo de la complejidad del circuito, resultando en que la simulación no coincida con la de la etapa anterior; por esto, la solución es usar una resolución de

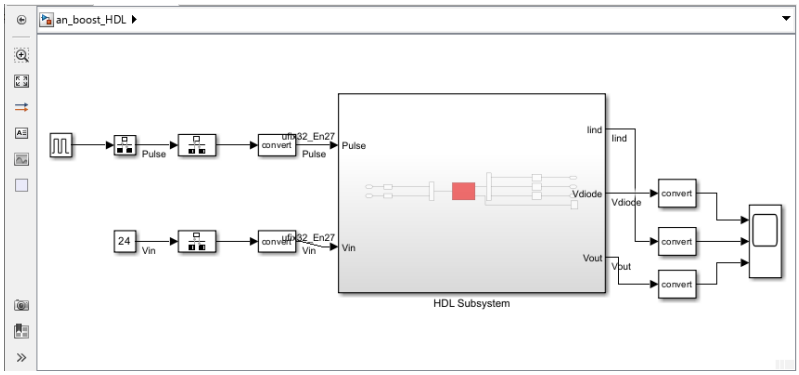


Figura 4.8: Modelo de implementación generado por SSC-HDL Workflow Advisor

24 bits o de 32 bits, pudiendo dividir los parámetros entre dos registros contiguos de 16 bits.

4.4.3. Generación de código HDL

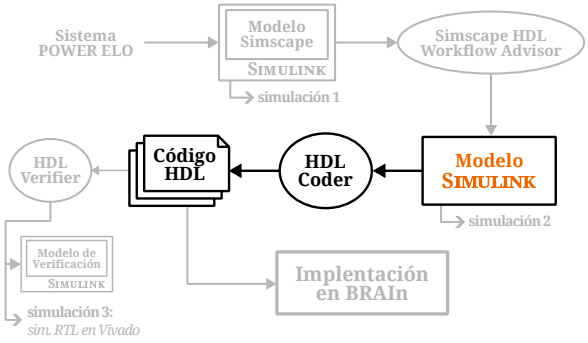


Figura 4.9: Pasos de la generación del código HDL

En esta etapa, se deben hacer las siguientes configuraciones y verificaciones al archivo del modelo de Simulink:

- Ajustar la frecuencia objetivo del sistema (**100 MHz**).
- Revisar que la plataforma FPGA objetivo sigue siendo `xc7a35tfgg484-2`.
- Definir el lenguaje HDL de salida, el cual puede ser *Verilog*, *SystemVerilog* o *VHDL*.

Junto a lo anterior, HDL Coder permite realizar una estimación del camino crítico del sistema por modelar, con el que puede determinar si existe la posibilidad de tener problemas de timing. Con ello ofrece alternativas para realizar pipelining,

logrando así mitigar este problema. **Es importante realizar esta subetapa de estimación de camino crítico**, ya que los problemas de *timing* pueden inducir errores muy impredecibles en la implementación física real.

Una vez esté todo lo anterior completado, se puede proceder a la generación de código como tal. HDL Coder creará una serie de archivos HDL en el lenguaje escogido, con los que se puede realizar una simulación de comportamiento a nivel RTL en lugar de dentro del entorno Simulink.

La generación de código genera un reporte, donde se pueden ver varios detalles acerca del módulo RTL generado. El usuario puede verificar que todo esté en orden, incluyendo la resolución de los datos de punto fijo, la cantidad de parámetros del sistema, la frecuencia de operación del módulo, entre otros.

4.4.4. Modelo de verificación con HDL Verifier

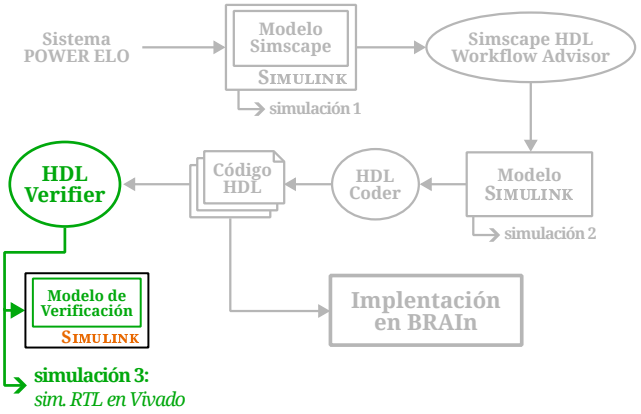


Figura 4.10: Pasos de la verificación del modelo RTL

Si bien este paso es técnicamente opcional para la generación de código como tal, **se considera la etapa más importante del proceso**.

Las razones de lo anterior son principalmente dos:

1. **Verificación de correcto funcionamiento:**
   La simulación permitirá confirmar que el modelo RTL funciona como debe, asegurando que el código HDL generado representa el sistema con la precisión necesaria. Esto es vital para detectar y evitar problemas de diseño previo a la

síntesis e implementación, ya que son tareas que consumen mucho tiempo y recursos computacionales como para tener que iterar en ellas.

2. **Facilidad de probar casos de borde:**
   También conocidos como *edge cases*. El entorno de HDL Verifier permite poner el código HDL generado bajo múltiples casos a los cuales se puede realizar análisis iterativamente, usualmente casos que serían difíciles de probar una vez implementado en hardware. Esto quiere decir que se puede realizar una prueba exhaustiva al comportamiento del módulo, y observar su comportamiento dinámico.

En esta etapa, se realiza una simulación del modelo, utilizando el código HDL generado y el entorno de Vivado. El add-on HDL Coder también incluye el add-on **HDL Verifier**, el cual esencialmente permite crear y armar *testbenches* en el entorno Simulink, usando bloques en lugar de HDL directamente. Un ejemplo de lo anterior se muestra en la Figura 4.11.

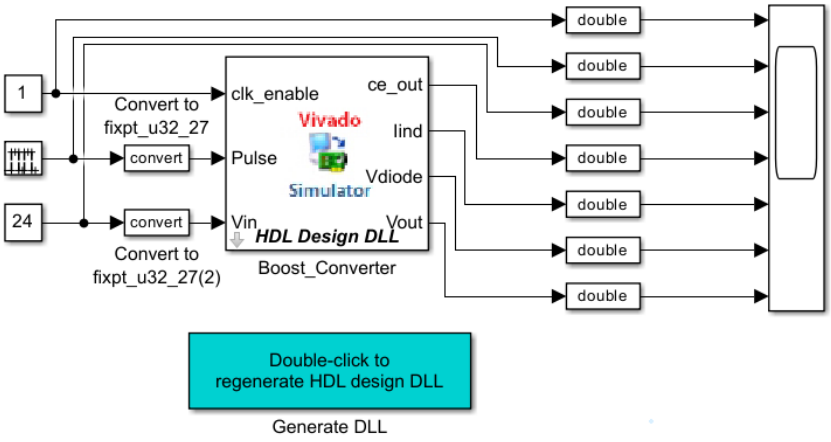


Figura 4.11: Testbench para el módulo HDL, con HDL Verifier en Simulink.

El resultado de esto es un archivo de simulación de Vivado, donde se comprueba que la implementación de la lógica del modelo representa correctamente el circuito.

Si todo se ha realizado correctamente, entonces se puede considerar con seguridad que **la generación de código ha sido exitosa, y el modelo está listo para ser implementado en BRAIn**.

4.4.5. Integración a BRAIn

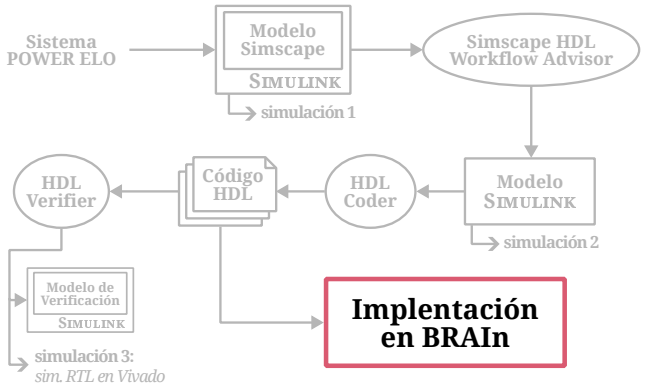


Figura 4.12: Paso de integración a BRAIn

Este paso involucra todo lo que tenga que ver con la identificación de los parámetros del módulo RTL generado con HDL Coder, para su correcta implementación en el *top module* del firmware FPGA de la plataforma BRAIn.

Dependiendo de la resolución de los datos de los parámetros, se puede llegar a requerir uno o dos registros por parámetro. Posterior a la instanciación del módulo y creación de los nuevos registros, se hace el proceso de síntesis, implementación, y generación de *bitstream* del proyecto, para luego *flashear* la memoria de BRAIn con el nuevo firmware.

Los detalles completos de implementación se pueden encontrar en la *application note*.

4.5. Visión general de la Application Note

El resultado de todo el proceso descrito a lo largo de este capítulo se concreta en la creación de una *application note*. Dicho documento pretende servir como guía para ingenieros que tengan la necesidad de realizar simulaciones HIL de modelos de sistemas de electrónica de potencia, ofreciendo un paso-a-paso para agilizar el flujo de trabajo y el proceso iterativo de diseño, particularmente para aquellos que posean menos conocimientos de FPGAs y electrónica digital.

La *application note* se subdivide en 6 secciones, cada una detallando un paso de implementación:

1. Diseño de sistema en Simscape
2. Conversión de sistema con Simscape Workflow Advisor
3. Consideraciones de modelo Simulink
   - Análisis de camino crítico y timing
4. Generación de código HDL con HDL Coder
5. Generación de modelo de verificación
6. Integración a BRAIn

Además, contiene secciones complementarias a los pasos anteriores, en la forma de 3 anexos:

- Optimizaciones de timing mediante pipelining
- Pasos para integrar manualmente el módulo generado
- Flasheo de memoria de FPGA de BRAIn

4.5.1. Enlace a la Application Note

La *application note* completa se encuentra en el enlace del siguiente repositorio: https://bitbucket.org/ac3edesarrollo/dc-hdlcoder_brain-hil/src/main/README.md

Capítulo 5

RESULTADOS EXPERIMENTALES

Este capítulo presenta los resultados experimentales obtenidos del proceso descrito en el Capítulo 4, en la implementación de los modelos de un buck converter y un boost converter en la FPGA de la plataforma BRAIn. Los resultados incluyen las simulaciones de Simulink de ambos sistemas, la simulación RTL de los códigos generados, y los reportes de síntesis sobre el uso de recursos y timing. Finalmente, se muestra el modelo del boost converter integrado y ejecutado en hardware en tiempo real.

5.1. Modelado y simulaciones

Esta subsección presenta las simulaciones realizadas en Simulink y Vivado. En total, se llevaron a cabo tres simulaciones: una del modelo en Simscape, otra del modelo de implementación, y una tercera del código HDL generado.

5.1.1. Buck Converter

La Figura 5.1 muestra el circuito del buck converter en Simscape, y la Figura 5.2 el gráfico de su simulación. Para este circuito, solo se observa el voltaje de la carga, denotado por el componente de sensor de voltaje conectado a la resistencia.

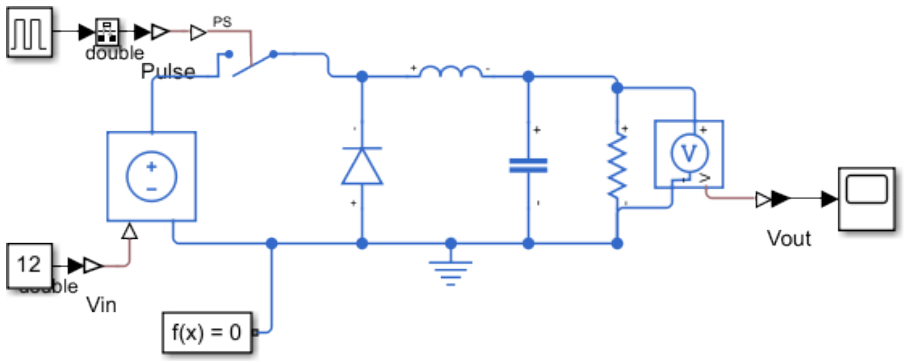


Figura 5.1: Circuito del buck converter en Simscape. Convierte 12V a 5V.

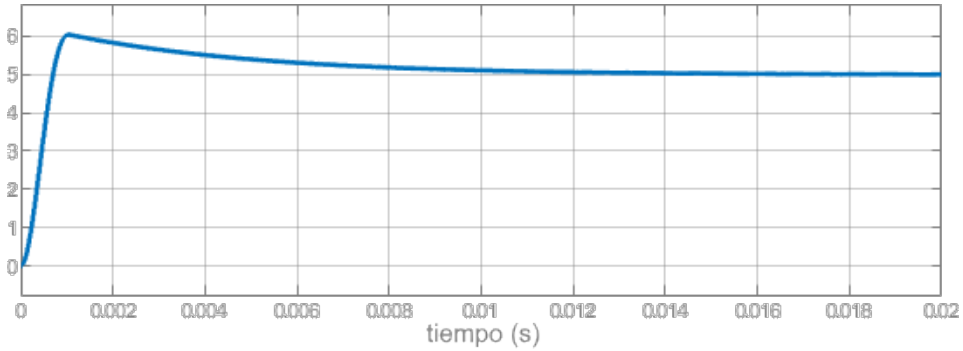


Figura 5.2: Simulación del buck converter en Simscape.

Posteriormente, se genera el modelo de implementación al ejecutar el modelo de Simscape a través de Simscape-HDL Workflow Advisor, para el cual se realiza la segunda simulación. Este modelo fue generado con una resolución de palabra de 32 bits de punto fijo. El resultado de esta simulación se muestra en la Figura 5.3.

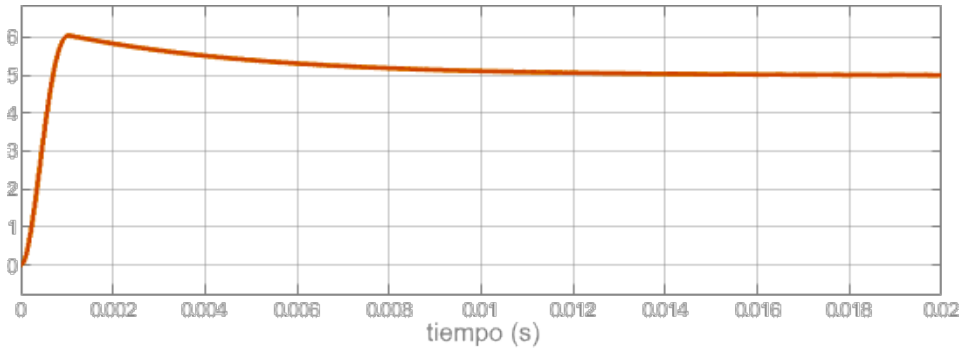


Figura 5.3: Simulación del modelo de implementación del buck converter

Se puede apreciar, al comparar ambos resultados, que el comportamiento del modelo del circuito no cambia, y el convertidor realiza correctamente la conversión de 12V a 5V.

Por último, se genera el código HDL del sistema. Este modelo es instanciado en un testbench y simulado en el entorno de Vivado, lo cual se muestra en la Figura 5.4.

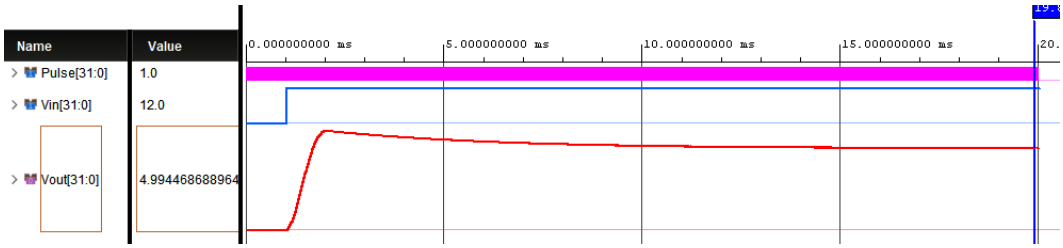


Figura 5.4: Simulación RTL en Vivado del módulo del buck converter

La simulación en Vivado muestra que el comportamiento del modelo es idéntico al del modelo de Simscape original, demostrando que el código HDL generado por HDL Coder funciona correctamente.

Como prueba adicional, se observa el comportamiento del modelo al cambiar su voltaje de entrada de 12V a 24V, cosa que debería resultar en un voltaje de salida aumentando de 5V a 10V. Si bien el buck converter técnicamente no fue diseñado con esto en mente, esto de todos modos permitirá demostrar que el sistema es efectivamente dinámico y responde a estímulos en tiempo real.

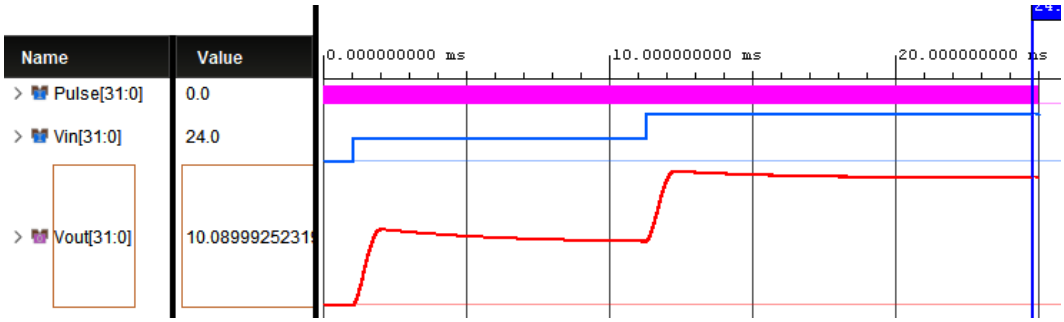


Figura 5.5: Prueba de comportamiento dinámico del buck converter en Vivado.

La Figura 5.5 muestra el resultado de este experimento. Al situar el cursor cerca

del final de la simulación, se puede observar bajo la columna de “Value” 24V de entrada al módulo, y aproximadamente 10V en la salida.

5.1.2. Boost Converter

La Figura 5.6 muestra el circuito del boost converter en Simscape, y la Figura 5.7 los gráficos de su simulación. En esta ocasión, se observan tres señales en lugar de una sola: la corriente que fluye por el inductor, el voltaje en el diodo, y el voltaje en la salida.

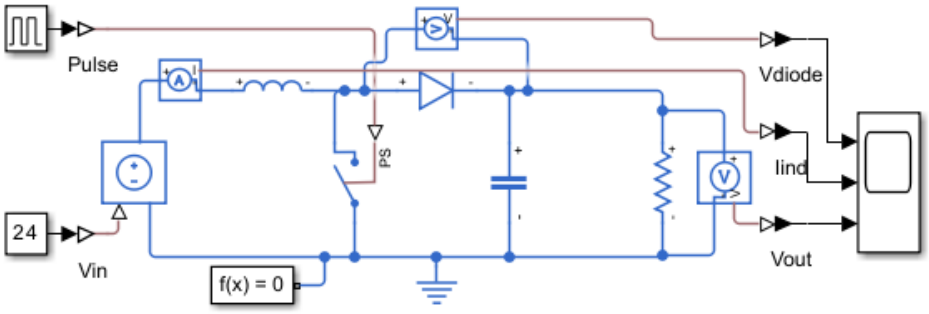


Figura 5.6: Circuito del boost converter en Simscape. Convierte 24V a 48V.

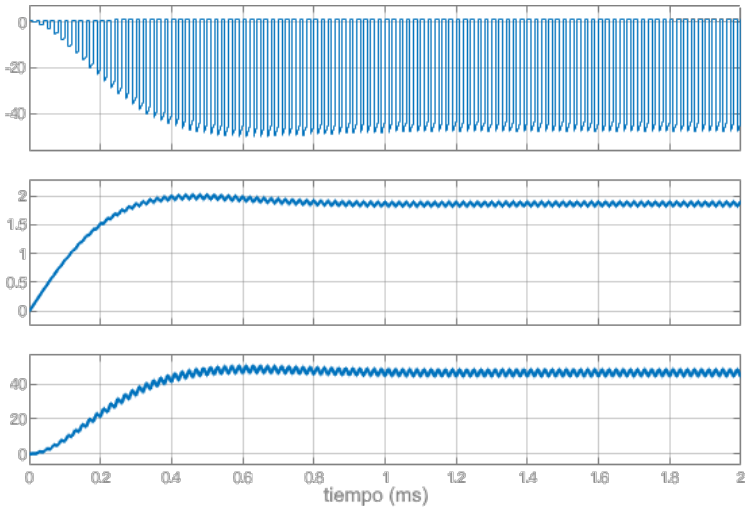


Figura 5.7: Simulación. En orden de arriba a abajo: V_{diodo}, $I_{inductor}$, V_{carga}

Luego, se genera el modelo de implementación utilizando Simscape-HDL Workflow Advisor, y se ejecuta la segunda simulación. Este modelo también fue

generado con una resolución de palabra de 32 bits de punto fijo. El resultado de esta simulación se muestra en la Figura 5.8.

Se puede apreciar, al comparar ambos resultados, que el comportamiento del modelo del circuito permanece inalterado, y el convertidor realiza correctamente la conversión de 24V a 48V como fue diseñado.

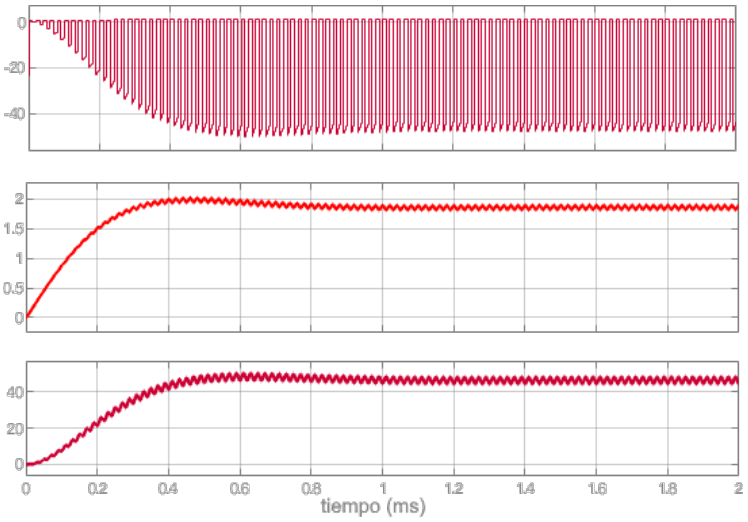


Figura 5.8: Simulación del modelo de implementación del boost converter

Finalmente, después de generar el código HDL se arma un testbench para realizar su simulación RTL en el entorno de Vivado. Esto se presenta en la Figura 5.9.

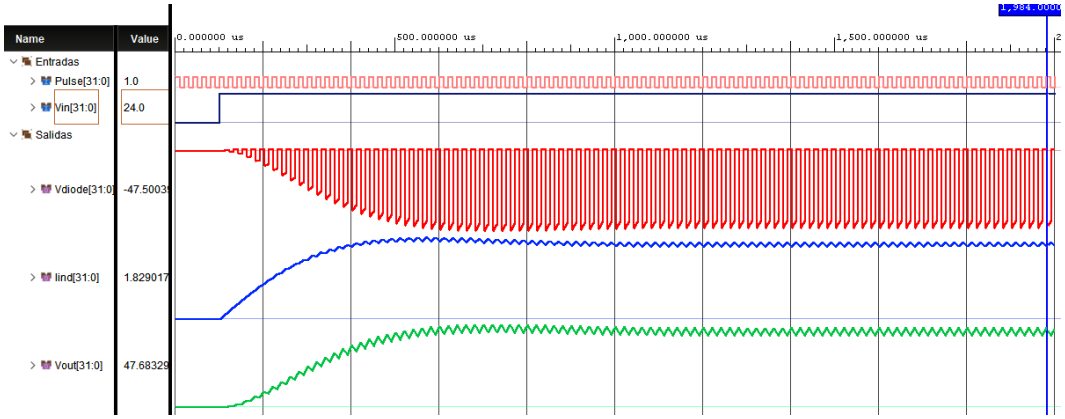


Figura 5.9: Simulación RTL en Vivado del módulo del boost converter

Como se puede observar, el comportamiento del modelo generado coincide con

el de los modelos de Simscape y Simulink.

Al igual que con el buck converter, se realiza una simulación extra para observar el comportamiento dinámico del modelo, cambiando el voltaje de entrada de 24V a 16V, lo que debería generar un voltaje de salida de 32V. Nuevamente, el propósito de esta simulación es confirmar que el modelo responde a cambios en la operación.

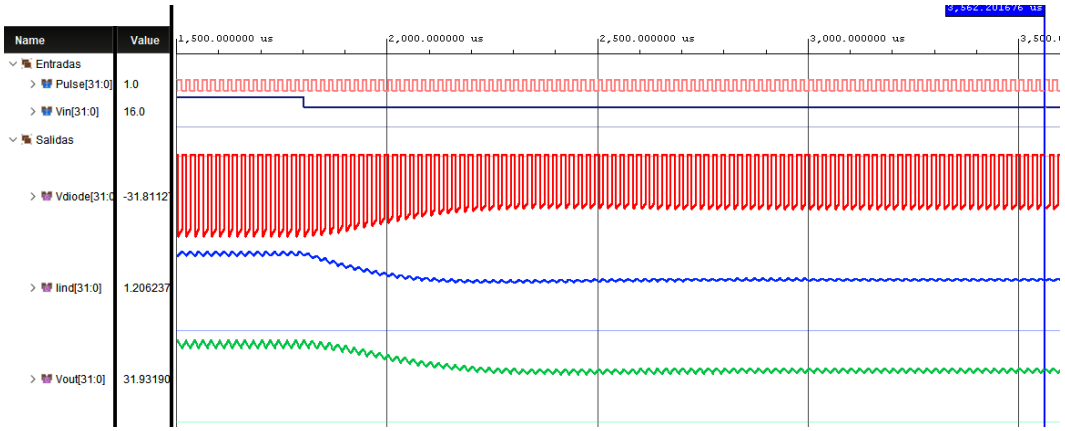


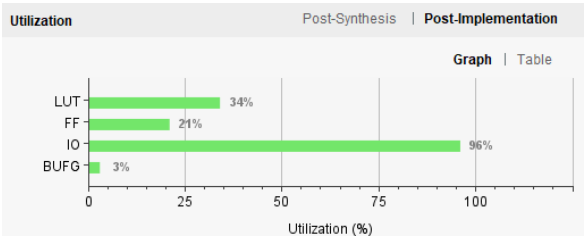
Figura 5.10: Prueba de comportamiento dinámico del boost converter en Vivado.

La Figura 5.10 permite confirmar, viendo la columna de "Value", que cuando el modelo recibe ahora un voltaje de 16V, el voltaje de la salida se estabiliza en alrededor de 32V.

5.2. Reportes de implementación

Después de haber obtenido las simulaciones del código generado y confirmado que funcionan correctamente, se procede a implementar el modelo en el firmware de BRAIn. El firmware ya contiene varios módulos como parte de su firmware "*essential*", y una de las metas del proceso de generación automática de código fue generar modelos suficientemente optimizados como para poder ser implementados en conjunto con este firmware.

A modo de referencia, en las Figuras 5.11 y 5.12 se muestran los datos de utilización de recursos y timing del firmware de BRAIn, **previo a la implementación** de los modelos de convertidores de potencia.



(a) Gráfico de uso de recursos

Utilization — Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	7058	20800	33.93
FF	8728	41600	20.98
IO	185	193	95.85
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 5.11: Uso de recursos por defecto del firmware de BRAIn.

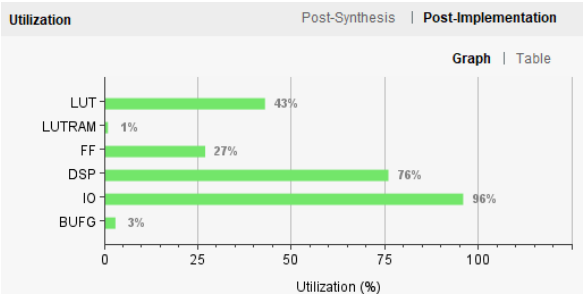
Timing

Worst Negative Slack (WNS):	0.769 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	12341

Figura 5.12: Reporte de timing por defecto del firmware de BRAIn.

5.2.1. Buck Converter

Al implementar el modelo del buck converter en el firmware de BRAIn, se obtienen los resultados de implementación mostrados en las Figuras 5.13 y 5.14.



(a) Gráfico de uso de recursos

Utilization — Post-Synthesis | Post-Implementation — Graph | Table

Resource	Utilization	Available	Utilization %
LUT	8966	20800	43.11
LUTRAM	32	9600	0.33
FF	11082	41600	26.64
DSP	68	90	75.56
IO	185	193	95.85
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 5.13: Uso de recursos del firmware de BRAIn con buck converter.

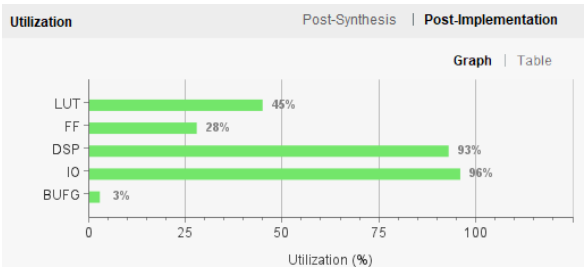
Timing

Worst Negative Slack (WNS):	0.424 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	19469

Figura 5.14: Reporte de timing del firmware de BRAIn con buck converter.

5.2.2. Boost Converter

Al implementar el modelo del boost converter en el firmware de BRAIn, se obtienen los resultados de implementación mostrados en las Figuras 5.15 y 5.16.



(a) Gráfico de uso de recursos

Utilization — Post-Synthesis | Post-Implementation — Graph | Table

Resource	Utilization	Available	Utilization %
LUT	9289	20800	44.66
FF	11610	41600	27.91
DSP	84	90	93.33
IO	185	193	95.85
BUFG	1	32	3.13

(b) Tabla de uso de recursos

Figura 5.15: Uso de recursos del firmware de BRAIn con boost converter.

Timing

Worst Negative Slack (WNS):	0.484 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	20997

Figura 5.16: Reporte de timing del firmware de BRAIn con boost converter.

5.2.3. Comparación de resultados de implementación

A continuación se presentan las Tablas 5.1 y 5.2, que resumen y comparan los resultados del proceso de implementación de su respectivo convertidor. Como se ha explicado en el Capítulo 4, no se considerarán los valores de IO, ya que las entradas y salidas de los módulos de firmware se manejan a través de registros en lugar de los pines de input/output de la FPGA.

Tabla 5.1: Tablas de comparación de implementación para el buck converter

(a) Utilización de recursos

Recurso	**Disponible**	**Sin buck**		**Con buck**		**Diferencia**
		Uso	Uso (%)	Uso	Uso (%)	
LUT	30400	7058	33.93	8998	43.11	**+1940**
FF	41600	8728	20.98	11082	26.64	**+2354**
DSP	90	0	0	68	75.56	**+68**
BUFG	32	1	3.13	1	3.13	0

(b) Timing

WNS sin buck	**WNS con buck**	**Variación**	**Cruce por 0**
0.769 ns	0.424 ns	-0.345 ns	No

Tabla 5.2: Tablas de comparación de implementación para el boost converter

(a) Utilización de recursos

Recurso	**Disponible**	**Sin boost**		**Con boost**		**Diferencia**
		Uso	Uso (%)	Uso	Uso (%)	
LUT	30400	7058	33.93	9289	44.66	**+2231**
FF	41600	8728	20.98	11610	27.91	**+2882**
DSP	90	0	0	84	93.33	**+84**
BUFG	32	1	3.13	1	3.13	0

(b) Timing

WNS sin boost	**WNS con boost**	**Variación**	**Cruce por 0**
0.769 ns	0.484 ns	-0.285 ns	No

De estas comparaciones se puede apreciar que los modelos de convertidores son cómodamente implementables en BRAIn: el aumento más grande de recursos es de parte del boost converter, utilizando un 10.73 % de los LUTs de la FPGA (obtenido: 44.66 % - 33.93 %) y un 6.93 % de sus FFs (obtenido: 27.91 % - 20.98 %).

El uso de DSPs es bastante alto, pero afortunadamente solo son utilizados por cada convertidor y no por otros módulos del firmware de BRAIn, por lo que no supone un problema en la implementación por ahora. Sin embargo, habría un problema si se llegase a implementar ambos modelos de convertidores al mismo tiempo.

5.3. Ejecución en hardware

Para visualizar las señales de salida del modelo del convertidor, se debe conectar un módulo DAC a la BRAIn, ya que ésta solo permite emitir señales digitales de forma nativa. Para esto, se utiliza el **Pmod DA4** de Digilent [72] (Figura 5.17), el cual recibe los valores desde la DSP de BRAIn mediante comunicación serial SPI.

El rango del voltaje de salida del Pmod DA4 es de 0-2.5V con una resolución de 12 bits, y se debe tener en consideración que este módulo "no está pensado para altas frecuencias" [73]. Por esto, las señales visualizadas en el osciloscopio no reflejarán con precisión total el comportamiento real del sistema, aunque siguen siendo útiles para su análisis cualitativo y como referencia.

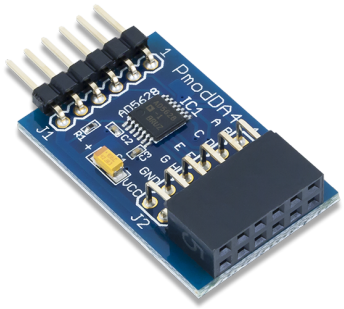


Figura 5.17: Pmod DA4 (Revision A) de Digilent

Por limitaciones de tiempo, solo se logra implementar el **boost converter**, y solo observar el **voltaje de la carga** (recordar que inicialmente se definen como

parámetros de salida el voltaje de la carga, el voltaje del diodo y la corriente por el inductor).

El *setup* de prueba, con BRAIn funcionando y con el Pmod DA4 conectado, se muestra en la Figura 5.18.

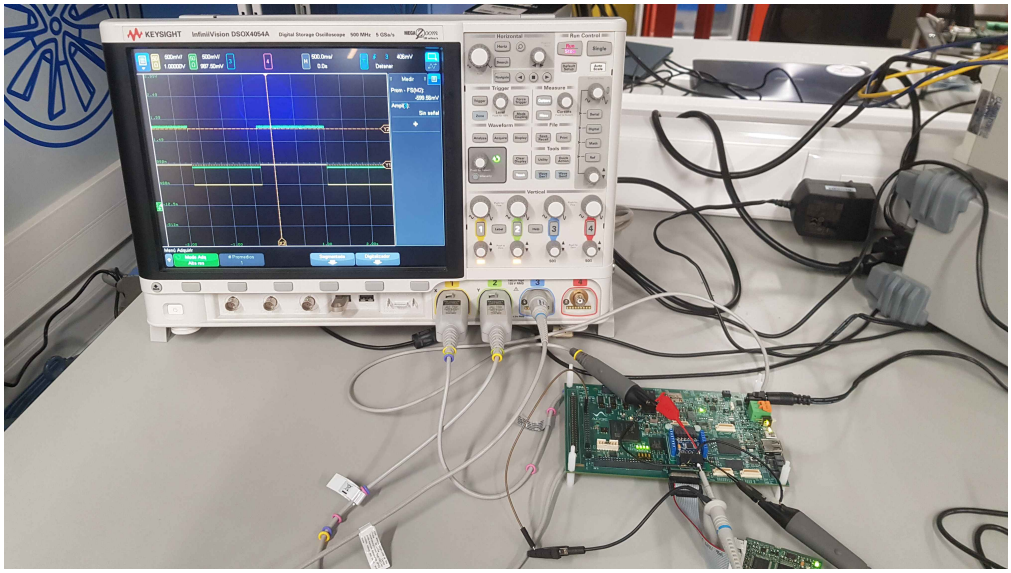


Figura 5.18: *Setup* de prueba del boost converter siendo ejecutado en BRAIn

En las simulaciones se obtuvieron voltajes de salida de hasta 48V. Dado que el Pmod DA4 tiene una salida máxima de 2.5V, se establece un valor de referencia de 64V como voltaje máximo, para posteriormente ajustar los valores generados por el modelo mapeándolos del rango [0-64] al rango [0-2.5].

Las imágenes de osciloscopio son sacadas en formato CSV, y luego se grafican los puntos en MATLAB.

5.3.1. Prueba de conversión de 24V a 48V

En esta prueba, se entrega un valor de 24V al modelo del boost converter, y se observa el voltaje de la carga en la salida. El resultado de esto se muestra en la Figura 5.19.

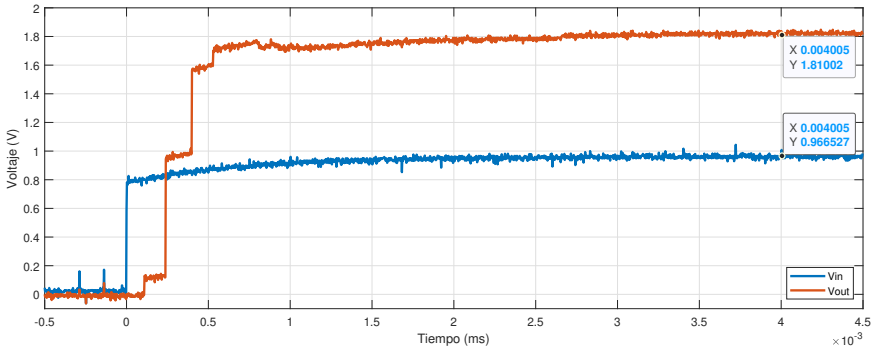


Figura 5.19: Conversión de 24 a 48V del modelo del boost converter en BRAIn

En el instante $t = 4{,}005$ ms, el voltaje de entrada al boost converter, entregado por el Pmod DA4, es de 0.96653 V, lo cual al ser mapeado al rango 0-64 equivale a:

$$\frac{0{,}96653}{2{,}5} \cdot 64 = \mathbf{24.74\ V}$$

El voltaje de salida del boost converter en ese mismo instante, entregado por el Pmod DA4, es de 1.81002 V, lo cual al ser mapeado al rango 0-64 equivale a:

$$\frac{1{,}81002}{5} \cdot 64 = \mathbf{46.34\ V}$$

Comparando con los valores deseados, se obtienen los siguientes errores porcentuales:

- **Voltaje de entrada:**

$$\frac{|24 - 24{,}74|}{24} \cdot 100 = 3{,}0833\,\%$$

- **Voltaje de salida:**

$$\frac{|48 - 46{,}34|}{48} \cdot 100 = 3{,}4583\,\%$$

5.3.2. Comportamiento dinámico al cambiar voltaje de entrada

En esta prueba, se hace oscilar el valor del voltaje de entrada entre 24V y 12V, y se observa el voltaje en la salida. El resultado de esto se muestra en la Figura 5.20.

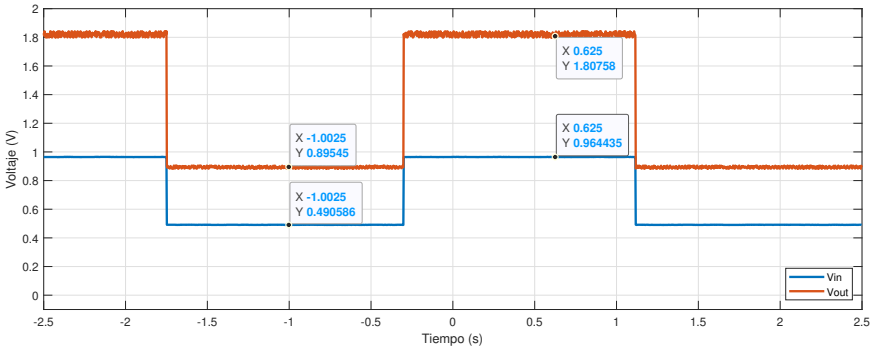


Figura 5.20: Comportamiento dinámico del boost converter en BRAIn

Instante $t = -1$ s:

El voltaje de entrada al boost converter, entregado por el DA4, es de 0.490586 V, y al mapearse al rango 0-64 esto equivale a:

$$\frac{0{,}490586}{2{,}5} \cdot 64 = \mathbf{12.56\ V}$$

El voltaje de salida del boost converter, entregado por el DA4, es de 0.89545 V, lo que al ser mapeado al rango 0-64 equivale a:

$$\frac{0{,}89545}{2{,}5} \cdot 64 = \mathbf{22.92\ V}$$

El error porcentual de cada voltaje, comparando con los valores deseados, es el siguiente:

- **Voltaje de entrada:**

$$\frac{|12 - 12{,}56|}{12} \cdot 100 = 4{,}6667\,\%$$

- **Voltaje de salida:**

$$\frac{|24 - 22{,}92|}{24} \cdot 100 = 4{,}5\,\%$$

Instante $t = 0{,}625$ s:

El voltaje de entrada y salida entregados por el DA4 son de 0.964435 V y 1.80758 V respectivamente. Dado que estos valores son muy parecidos a los de la prueba anterior, no se repetirán los cálculos de voltajes reales y errores porcentuales.

A partir de estos resultados, se observa que los valores obtenidos del módulo se mantienen dentro de un margen de error del 5 %. Además, según la Figura 5.19, el modelo responde y estabiliza la señal en un tiempo de aproximadamente 2 ms, similar a los 1.5-2 ms vistos en las simulaciones RTL (Figura 5.9).

Con esto, se valida el funcionamiento del boost converter siendo ejecutado y controlado en la plataforma BRAIn en tiempo real.

Capítulo 6

CONCLUSIONES Y TRABAJO FUTURO

6.1. Conclusiones

6.1.1. Análisis de los resultados obtenidos

Los resultados experimentales demostraron que tanto la generación como la validación del código HDL para ambos convertidores fueron exitosas. Tanto las simulaciones en el entorno de Simscape y Simulink como las simulaciones RTL en Vivado lograron confirmar que los modelos generados emulan adecuadamente el diseño de los convertidores en funcionalidad y comportamiento dinámico.

Además, la implementación en BRAIn de estos modelos (cada uno por su cuenta) reveló que éstos utilizan una baja cantidad de recursos lógicos de la FPGA integrada, sin contar los bloques DSP, y también logran cumplir con limitaciones de *timing*, sin sacrificar el comportamiento del modelo. El uso de bloques DSP, sin embargo, resulta ser bastante grande para ambos modelos; no se ha experimentado con formas de reducir su consumo.

Por último, las pruebas hechas al modelo del boost converter, implementado y siendo ejecutado en tiempo real en la BRAIn, confirman que el código generado funciona y responde de manera satisfactoria, mostrando un comportamiento similar

al visto en simulaciones y manteniendo los valores del voltaje de salida dentro de un margen de error del 5 % en comparación con los valores ideales.

6.1.2. Trabajo realizado

Se exploraron múltiples facetas de generación de código HDL para la simulación en tiempo real. El trabajo investigativo de los varios métodos y alternativas de flujos de trabajo ayudaron a acotar las técnicas más eficientes para modelar sistemas de electrónica de potencia para aplicaciones HIL.

Vitis HLS demostró ser una herramienta potente en combinación con opciones de generación de código en C/C++, como Simulink y ORTiS. Aunque estos entornos no facilitan el diseño de circuitos de electrónica de potencia (por lo que fueron descartados en esta memoria), mostraron buen rendimiento en aplicaciones de sistemas de tiempo discreto, siendo útiles para simulación HIL en otros contextos. El flujo de trabajo basado en Simscape y HDL Coder resultó el más adecuado para los objetivos planteados, permitiendo la generación de código HDL de modelos de sistemas de electrónica de potencia que resultan precisos y eficientes tanto en rendimiento como en uso de recursos.

HDL Verifier, un add-on de HDL Coder, facilita las simulaciones RTL mediante la creación automática de testbenches, evitando la escritura manual de código HDL. Aunque no es esencial en el flujo de generación de código, esta herramienta igualmente ayudó con los objetivos que se plantearon al comienzo: agilizar en gran medida la parte de simulación.

El resultado de mayor valor de este trabajo fue la creación de la *application note*, que documenta todo el flujo de trabajo para generar modelos HDL de sistemas de electrónica de potencia e implementarlos para realizar simulaciones HIL. Se espera que este documento simplifique el proceso para futuros usuarios.

6.2. Trabajo Futuro

La generación automática de código HDL ha demostrado ser eficiente en el uso de recursos, pero se puede optimizar aún más a costa del rendimiento. Dado que el paso de simulación de la mayoría de estos sistemas es considerablemente más rápido que el periodo de reloj de la BRAIn y muchos modelos emplean numerosos bloques

DSP, se podrían reasignar más registros a las funciones que utilizan estos DSP, con el fin de aumentar la latencia del sistema sin superar los márgenes de rendimiento. Además, se podría explorar la implementación simultánea de múltiples sistemas de electrónica de potencia, ya que actualmente solo se ha integrado un convertidor a la vez en el firmware de BRAIn debido al alto uso de bloques DSP, lo que impide la coexistencia de un buck y un boost converter.

Dado que la plataforma BRAIn sigue siendo un trabajo en continuo desarrollo, es posible que su arquitectura evolucione y sea escalada para mejorar aún más sus capacidades de simulación *hardware-in-the-loop*. De ser así, el aumento en recursos y capacidad de cómputo abrirían las puertas para la aplicación de las dos ideas anteriores.

Para el control de convertidores estáticos en electrónica de potencia, se suele manipular el ciclo de trabajo de las señales PWM que conmutan los *switches* del circuito, para así controlar el voltaje de la salida. Esto es algo que no fue explorado en esta memoria, y sería otra manera de poner a prueba el flujo de trabajo propuesto.

Finalmente, se espera que los avances tecnológicos y el estudio más profundo en esta área permitan ampliar la *application note* desarrollada en esta memoria. Como culminación del trabajo tanto investigativo como práctico realizado, dicha *application note* podría expandir su alcance para incorporar nuevos descubrimientos y experimentos, e incluso recibir contribuciones de otros ingenieros.

Apéndice A

USO DE HERRAMIENTAS DE SOFTWARE

En este apéndice se explica el uso de los programas utilizados a lo largo de los capítulos 3, para aquellos lectores que deseen recrear los resultados mostrados. Estos programas son:

1. Vitis HLS
2. Embedded Coder
3. ORTiS

El uso de Simscape y HDL Coder se encuentra extensivamente detallado en la *application note*, por lo que no se explica en este apéndice. (URL: https://bitbucket.org/ac3edesarrollo/dc-hdlcoder_brain-hil/src/main/README.md)

IMPORTANTE: Se asume que el usuario está familiarizado con MATLAB y Simulink, y con aspectos básicos de programación en C.

A.1. Vitis HLS

Esta sección detalla la creación de nuevos proyectos y los puntos específicos del flujo de trabajo que impone la herramienta.

Más detalles acerca de la creación de nuevos proyectos se encuentran en el siguiente enlace:
https://docs.amd.com/r/en-US/ug1399-vitis-hls/Creating-an-HLS-Component

A.1.1. Configuración de nuevo proyecto

Posterior a la creación de un **workspace**, se debe crear un nuevo **Componente HLS**. Los componentes actuarán como el equivalente a un "proyecto" HLS de Vitis. Para ejemplificar el proceso, se mostrarán los archivos del oscilador biquad dual del capítulo 3.

- Cuando se abra la ventana de creación de nuevo componente, se puede asignar un nombre y ubicación al nuevo componente. Está permitido crear un componente sin tener archivos fuente aún.
- En la pestaña de **Hardware**, se debe buscar el *part number* de la FPGA de BRAIn, que corresponde al `xc7a35tfgg484-2`.

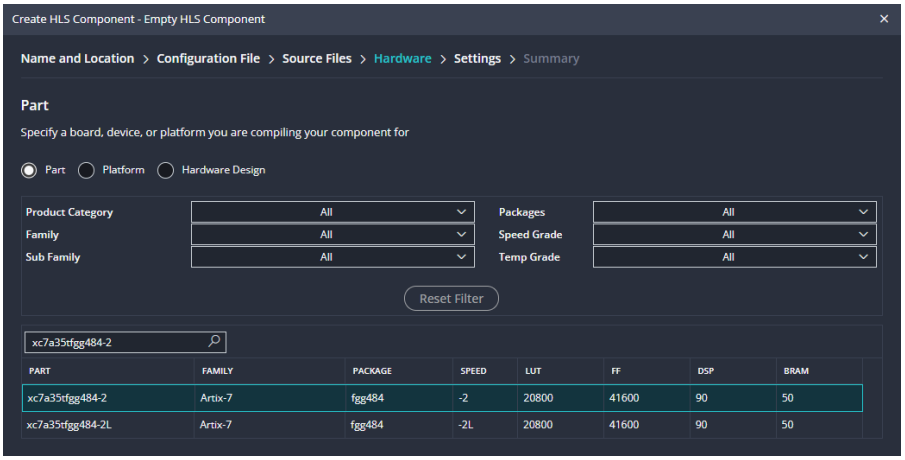


Figura A.1: Ventana de creación de nuevo componente HLS: Hardware

- Para la pestaña de **Settings**:
  - El reloj debe configurarse a **100 MHz**, por la FPGA de BRAIn.
  - "Flow Target" debe configurarse en **Vivado IP Flow Target**. Esto generará código RTL para implementarse como IP en Vivado.

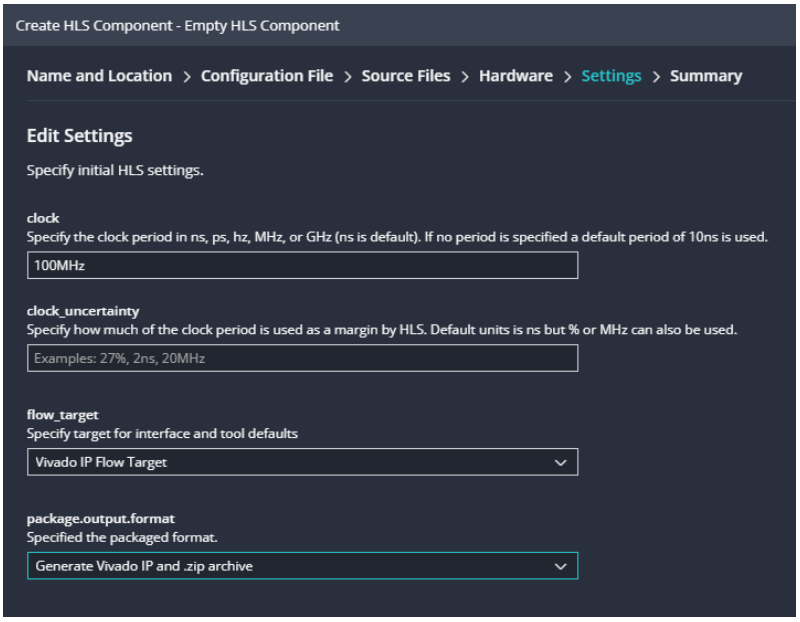


Figura A.2: Ventana de creación de nuevo componente HLS: Settings

- Una vez creado el componente, se pueden crear o cargar los archivos fuente para la función que se quiere implementar (Figura A.3: carpeta "Sources"), y en la carpeta "Test Bench" se puede crear un archivo .c/.cpp que contenga un `main()` con el que probar la función (Figura A.3: `resonator_v3_test.cpp`).

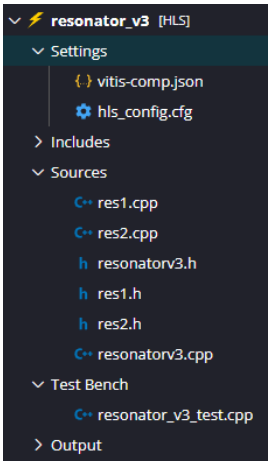


Figura A.3: Jerarquía del componente recién creado.

Entre los archivos fuente debe existir una función "top level", que encapsule

todo el módulo y que será la que finalmente se sintetice. Si se desea realizar una prueba del algoritmo previo a la generación del código RTL, debe existir un archivo .c/.cpp en la carpeta "Test Bench".

- En `hls_config.cfg` (Figura A.5), se debe asegurar de incluir todos los archivos fuente y archivos testbench correctamente, además de ingresar el *nombre de la función* que se sintetizará. Para el ejemplo del oscilador, el archivo `resonatorv3.cpp` llama a los otros archivos `res1.cpp` y `res2.cpp`, como se muestra en la Figura A.4, por ende la función *top level* en este caso es `resonator_v3()`.

```
1  #include "resonatorv3.h"
2  #include "res1.h"
3  #include "res2.h"
4
5  void resonator_v3(ap_uint<1> impulse1, ap_uint<1> impulse2, rFloat *bq1, rFloat *bq2)
6  {
7      /* resultados resonadores */
8      *bq1 = res1(impulse1);
9      *bq2 = res2(impulse2);
10 }
```

Figura A.4: Contenidos de `resonator3.cpp`.

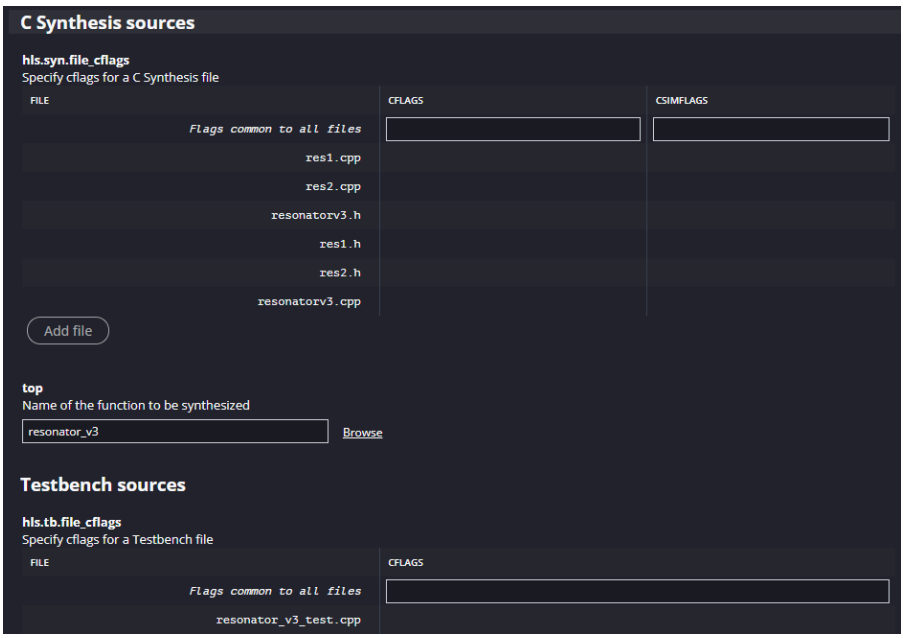


Figura A.5: Archivo `hls_config.cfg`.

- Las demás configuraciones se pueden dejar en su estado default.

A.1.2. Flow de Vitis

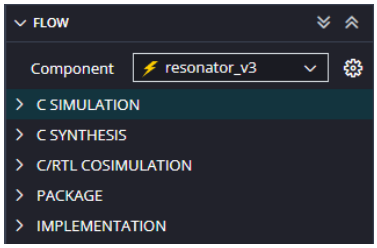


Figura A.6: Vitis Flow

Vitis tiene un flujo de trabajo:

- **C Simulation:**
  Simula el código C puramente como código C, usando el compilador GCC. Para esto, ejecuta el archivo de testbench.

- **C Synthesis:**
  Se genera el código RTL a partir del código C. Se genera un reporte de timing, pipelining/paralelismo, y throughput.

- **C/RTL Cosimulation:** Verifica que el comportamiento del código C y el código RTL coincidan, generando un archivo de *waveform data* que esencialmente genera un simulación RTL en el entorno de Vivado del módulo RTL, recreando las condiciones del archivo testbench en C/C++ del usuario.

- **Package:** Genera la IP como un archivo .zip que puede ser importado directamente en Vivado.

- **Implementation (opcional):** Genera síntesis, place-and-route e implementación de la IP para estimar uso de recursos y rendimiento una segunda vez.

A.1.3. Comentarios adicionales

- Para la implementación de punto fijo, simplemente se debe incluir la librería de Vitis HLS "`<ap_fixed.h>`". Ésta permite definir un nuevo tipo de dato `ap_fixed<x,y>` en lugar de un tipo de dato `float` o `double`, donde `x` corresponde al largo en bits del dato de punto fijo, e `y` corresponde a la cantidad de bits que representa la parte *entera*. **Nota:** esta librería solo se puede utilizar

en C++, obligando al usuario a trabajar completamente en C++ (aunque el código puede ser escrito como en C de todos modos).

- La generación del bloque IP incluye señales de control, de las cuales se puede leer más en el siguiente enlace:
  https://docs.amd.com/r/en-US/ug1399-vitis-hls/Block-Level-Control-Protocols
- Los códigos de las implementaciones de los sistemas del capítulo 3 se encuentran en el Apéndice B.

A.2. Embedded Coder

Corresponde a un add-on de MATLAB/Simulink, como extensión a MATLAB Coder y Simulink Coder. La ventaja de este add-on por sobre los otros es que, al estar pensado para sistemas embebidos y microcontroladores, el código que genera es en C/C++ (ideal para Vitis), y genera muy pocos archivos, haciendo su incorporación muy sencilla.

Una vez cargado Embedded Coder en el entorno de Simulink, se debe ir a la opción "Quick Start" de la barra de herramientas.

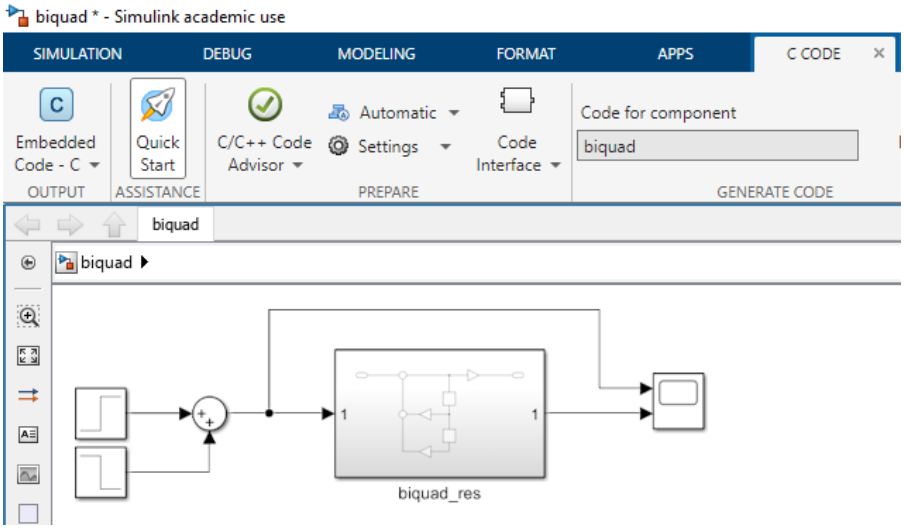


Figura A.7: Carga de Embedded Coder en Simulink

Se abrirá una ventana donde se sigue una serie de pasos para generar el código C del sistema:

- Para evitar generar código de *todo* el sistema de Simulink (ya que eso esencialmente fijaría las entradas y salidas), se debe seleccionar el *subsistema* que contenga el algoritmo que se desea implementar.

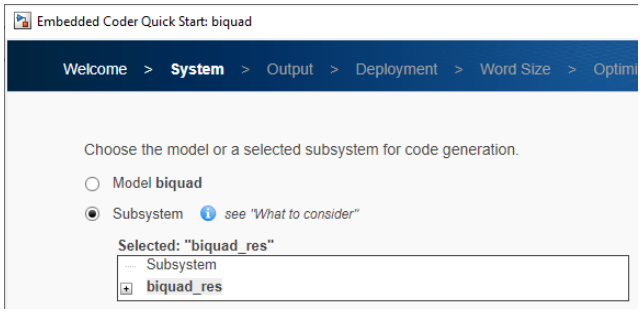


Figura A.8: Selección de subsistema de Simulink

- Luego, se debe seleccionar el lenguaje del código generado. No es estrictamente necesario seleccionar C o C++, pero se recomienda **C Code** con **Single instance**, ya que este código será compatible tanto para C como para C++.

- Como este add-on está pensado para sistemas embebidos, solicita seleccionar el procesador objetivo. Esto solo tiene como fin saber los tipos de datos soportados por el dispositivo embebido, por lo que se puede dejar en su estado default.

- Igualmente para la optimización, se puede dejar en **Execution efficiency**.

Al terminar de generar el código, se creará un reporte de éste, junto con cuatro archivos (Figura A.9).

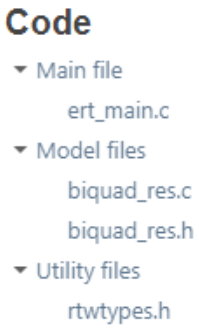


Figura A.9: Código generado por Embedded Coder

- **ert_main.c:** Contiene el `main()` que ejecuta la simulación y sería el equivalente al testbench escrito por el usuario en la sección de Vitis. También define una función `rt_OneStep()` que ejecuta el sistema un paso a la vez, simulando una llamada periódica del módulo. Es principalmente ilustrativo, y no es parte esencial del algoritmo del código generado.

- **<nombre_subsistema>.c/.h:** Son los archivos que efectivamente modelan el sistema. Incluye las estructuras `ExtU` y `ExtV`, instanciadas mediante las variables `rtU` y `rtV`, que almacenan las variables de entrada y salida.

- **rtwtypes.h:** Define los tipos de datos, teniendo en mente el hardware objetivo. Desde aquí se puede configurar rápidamente cualquiera de ellos, incluyendo los tipos de dato de precisión para poder implementar aritmética de punto fijo.

Si se planea utilizar punto fijo, entonces solo basta cambiar la extensión del archivo `<nombre_subsistema>.c` a `.cpp`.

El código generado es casi completamente funcional para Vitis, excepto que no cuenta con una función "top". En pocas palabras, el archivo del modelo genera dos funciones: `<nombre_subsistema>_initialize()` y `<nombre_subsistema>_step()`. La primera función inicializa el modelo y se debe ejecutar solo una vez, mientras que la segunda se ejecuta en cada paso. Se debe crear una tercera función que encapsule ambas ("*wrapper*"), considerando que `initialize()` solo debe ejecutarse la primera vez.

Esta función será nombrada `<nombre_subsistema>()` por simplicidad. Luego, se deben hacer las siguientes adiciones (reemplazando `<nombre_subsistema>` con `biquad_res`):

- **Declaración:**

```
/* Model entry point functions */
extern void biquad_res_initialize(void);
extern void biquad_res_step(void);
extern void biquad_res(ExtU in, ExtY *out);    // funcion wrapper
```

- **Definición:**

```
void biquad_res(ExtU in, ExtY *out)
{
    /* receive inputs */
    rtU = in;
    /* init */
    static bool init = true;
    if (init) {
        biquad_res_initialize();
        init = false;
    }
    /* step */
    biquad_res_step();
    /* generate outputs */
    *out = rtY;
}
```

Con esto, se puede definir `<nombre_subsistema>()` como la *top function* en Vitis. Adicionalmente, se puede cambiar el tipo de dato `real_T` en el archivo `rtwtypes.h` para que sea de tipo punto fijo, como se describió en los comentarios de la sección de Vitis.

A.3. ORTiS

ORTiS es un programa de línea de comandos (CLI), por lo que su uso es bastante directo pero también incómodo. En su repositorio de GitHub se puede encontrar el programa `codegen.exe`; la herramienta acepta como entrada un circuito de electrónica de potencia definido en un archivo de texto plano, y genera un archivo de cabecera de C++ (`.hpp`), donde se define una función *template*.

Por este motivo, se debe hacer manualmente el archivo fuente principal en `.cpp`, la cual contendrá la función *wrapper* que servirá como *top function*, similar a como se ha hecho anteriormente.

A.3.1. Uso y formato de netlist

Para convertir una netlist, se debe ejecutar desde el terminal:

```
$ codegen.exe archivo_netlist
```

donde `archivo_netlist` puede tener cualquier extensión (incluso una que no exista). El archivo que describe la netlist solo ejecuta una función por línea:

- **Comandos:**
  Solo existen dos comandos:

  ```
  #name nombre_sistema
  #const nombre_const valor_const
  ```

- **Comentarios:**
  Utilizando el símbolo %

- **Definición de componentes:**
  El formato básico para definir componentes es el siguiente:

  ```
  TipoComponente nombre (param1, ..., paramP) {nodo1, ..., nodoN}
  ```

En el Apéndice B se presentan las netlists usadas en el capítulo 3.

A.3.2. Código generado por la herramienta

La herramienta genera un único gran archivo header de C++ (`.hpp`), el que contiene un *template*. El archivo tiene una estructura similar a la siguiente:

```
template <int instance, typename real> void nombre_sistema
(
  real  x_out[No. salidas],
  real& y_solucion_interna1,
  real* y_solucion_interna2,
  ...,
  int   u_input_componente1,
  bool  u_input_componente2[4],
  ...
)
{
  // codigo generado por el solver modelando el sistema
}
```

El arreglo `x_out[N]` contiene tantos elementos como nodos distintos de 0 existan en el sistema, por lo que cada elemento de `x_out` corresponde al voltaje de un nodo. Adicionalmente, puede que los últimos elementos del arreglo correspondan a corrientes de las fuentes de voltaje ideales que hayan en el circuito, si es que existen.

El *template* recibe dos parámetros, `instance` y `real`:

- `instance` corresponde a la instancia del módulo, ya que un sistema puede estar ejecutando varios subsistemas idénticos en paralelo. Si solo se va a ejecutar uno, entonces por lo general simplemente se usa `0`.
- `real` corresponde al tipo de dato que usarán los parámetros de los componentes del sistema. Generalmente se usa float/double al simular en el CPU, y tipos de datos *fixed point* para FPGAs (ej. `ap_fixed` en Vitis).

Se pueden tener argumentos tanto de salida como de entrada al sistema; los argumentos de salida usualmente son pasados por referencia, y los de entrada por valor. Nuevamente, códigos de ejemplo se muestran en el Apéndice B si se necesita un ejemplo aplicado.

A.3.3. Función wrapper

Asumiendo que el archivo generado por ORTiS es `nombre_sistema.hpp` y que contiene la función `solver_sistema()`, se presenta un esqueleto de una función *wrapper* que se puede sintetizar por Vitis:

Archivo `wrapper_sistema.hpp`

```
#ifndef SISTEMA_HPP
#define SISTEMA_HPP

#include "nombre_sistema.hpp"

void wrapper_sistema(
  real      x_out[N],
  tipo_dato *output1,
  ...
  tipo_dato *outputA,
  tipo_dato input1,
  ...
```

```
  tipo_dato inputB
);

#endif
```

Aquí, "tipo_dato" corresponde al tipo de dato que le corresponda a cada entrada/salida según esté definido en nombre_sistema.hpp, y "real" el tipo de dato punto fijo. Este se puede definir en un archivo adicional, de manera análoga a como Embedded Coder utiliza rtwtypes.h para definir el tipo de dato para las variables reales.

Archivo wrapper_sistema.cpp

```
#include "wrapper_sistema.hpp"

void wrapper_sistema(
  real      x_out[N],
  tipo_dato *output1,
  ...
  tipo_dato *outputA,
  tipo_dato input1,
  ...
  tipo_dato inputB
)
{
  // pueden ir aqui las directivas HLS
  tipo_dato input1_i, ..., inputB_i;
  tipo_dato output1_i, ..., outputA_i;

  real x_out_i[N];

  solver_sistema<0,real>(x_out_i, &output1_i, ..., &outputA_i,
    input1_i, ..., inputB_i);
}
```

A.3.4. Lista de componentes

La lista de componentes completa se puede encontrar en la guía de usuario oficial de ORTiS:

https://github.com/OpenRealTimeSimulation/SolverCodegen/files/8312269/ortis_codegen_cli_guide-0.9.pdf

Apéndice B

CÓDIGOS, SCRIPTS, Y DIAGRAMAS

Este apéndice muestra los códigos utilizados a lo largo del capítulo 3, para aquellos lectores que deseen recrear los resultados mostrados utilizando las herramientas explicadas en el Apéndice A.

B.1. Oscilador biquad dual

B.1.1. Punto flotante

Carpeta "Sources"

Archivo resonatorv3.cpp (*top module*)

```
#include "resonatorv3.h"
#include "res1.h"
#include "res2.h"

void resonator_v3_f(ap_uint<1> impulse1, ap_uint<1> impulse2, rFloat *
    bq1, rFloat *bq2)
{
    /* resultados resonadores */
    *bq1 = res1(impulse1);
```

```
    *bq2 = res2(impulse2);
}
```

Archivo resonatorv3.h (*top module*)

```
#ifndef _RESONATOR_V3_H_
#define _RESONATOR_V3_H_

#include "res_types.h"
#include "ap_int.h"

typedef struct Biquad {
    rFloat bq1O;
    rFloat bq2O;
} Biquad;

void resonator_v3_f(ap_uint<1> impulse1, ap_uint<1> impulse2, rFloat *
    bq1, rFloat *bq2);

#endif
```

Archivo res1.cpp

```
#include "res1.h"

/* coeficientes para biquad */
rFloat g1B0 = 0.051342456225139;
rFloat g1A1 = -1.997362212708321;
rFloat g1A2 = 1.0;

rFloat res1(ap_uint<1> imp)
{
    /* vector de salida */
    static rFloat bq1Output[3] = {0.0, 0.0, 0.0};

    bq1Output[2] = bq1Output[1];
    bq1Output[1] = bq1Output[0];
    bq1Output[0] = (g1B0 * imp)
                 - (g1A1 * bq1Output[1])
                 - (g1A2 * bq1Output[2]);

    return bq1Output[0];
```

```
}
```

Archivo res1.h

```
#ifndef _RES1_H_
#define _RES1_H_

#include "res_types.h"
#include "ap_int.h"

rFloat res1(ap_uint<1> imp);

#endif
```

Archivo res2.cpp

```
#include "res2.h"

/* coeficientes para biquad */
rFloat g2B0 = 0.064671642929027;
rFloat g2A1 = -1.995813196269491;
rFloat g2A2 = 1.0;

rFloat res2(ap_uint<1> imp)
{
    /* vector de salida */
    static rFloat bq2Output[3] = {0.0, 0.0, 0.0};

    bq2Output[2] = bq2Output[1];
    bq2Output[1] = bq2Output[0];
    bq2Output[0] = (g2B0 * imp)
                 - (g2A1 * bq2Output[1])
                 - (g2A2 * bq2Output[2]);

    return bq2Output[0];
}
```

Archivo res2.h

```
#ifndef _RES2_H_
#define _RES2_H_

#include "res_types.h"
```

```
#include "ap_int.h"

rFloat res2(ap_uint<1> imp);

#endif
```

Archivo res_types.h

```
#ifndef _RES_TYPES_H_
#define _RES_TYPES_H_

typedef float rFloat;

#endif
```

Carpeta “Test Bench”

Archivo resonator_v3_test.cpp

```
#include "resonatorv3.h"
#include <stdio.h>
#include "ap_int.h"

/* constantes */
#define FS          16000       // frecuencia de muestreo a 16 KHz
#define SIM_TIME_S  1           // tiempo de simulacion en segundos

const int simTime    = SIM_TIME_S * FS;

int main() {
    //Biquad vec;
    rFloat vec1 = 0.0;
    rFloat vec2 = 0.0;
    ap_uint<1> imp1 = 1;
    ap_uint<1> imp2 = 1;

    FILE *file = fopen("output_1.txt", "w");
    if (file == NULL) return 1;

    for (int i = 0; i < simTime; i++)
    {
        resonator_v3_f(imp1, imp2, &vec1, &vec2);

```

```
25            fprintf(file, "%f %f\n", vec1, vec2);
26            imp1 = 0;
27            imp2 = 0;
28        }
29
30        fclose(file);
31
32        return 0;
33    }
```

Este testbench genera un archivo de texto “output.txt” que almacena los valores de la salida de los resonadores en cada línea. Se usa como entrada para el script de MATLAB que grafica estos puntos.

Script de gráfico de MATLAB

```
Fs = 16000;
SimTime = 1 * Fs;

data = load('output_1.txt');
time = (0:length(data)-1) / SimTime;

plot(time, data(:,1)); % Plot first oscillator
hold on;
plot(time, data(:,2)); % Plot second oscillator
xlabel('Tiempo (s)');
ylabel('Salida');
xlim([0 0.05])
ylim([-1.2 1.2])
title('Respuesta a impulso de osciladores biquad');
legend('Oscilador 1', 'Oscilador 2');
```

B.1.2. Punto fijo

La única diferencia con los códigos anteriores es `res_types.h`, y una línea de código en `resonator_v3_test.cpp`, donde se requiere convertir los números de punto fijo a punto flotante con `to_float()` para que `fprintf` pueda escribir los valores correctamente.

Archivo res_types.h

```
#ifndef _RES_TYPES_H_
#define _RES_TYPES_H_

#include <ap_fixed.h>
typedef ap_fixed<32,8> rFloat;

#endif
```

Archivo resonator_v3_test.cpp

```
        fprintf(file, "%f %f\n", vec1.to_float(), vec2.to_float());
```

B.2. Oscilador biquad singular con Embedded Coder

B.2.1. Diagrama de Simulink

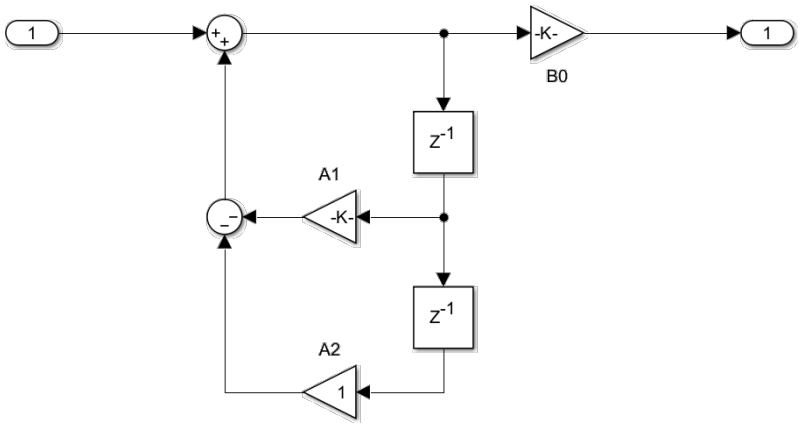


Figura B.1: Diagrama de bloques del oscilador biquad de 130.8 Hz

- $B0 = 0{,}051342456225139$
- $A1 = -1{,}997362212708321$
- $A2 = 1{,}0$

B.2.2. Post-generación de código C

(Notar números de líneas)

Carpeta "Sources"

Ajustes a biquad_res.cpp

```
78 void biquad_res(ExtU in, ExtY *out)
79 {
80     /* receive inputs */
81     rtU = in;
82
83     /* init */
84     static bool init = true;
85     if (init) {
86         biquad_res_initialize();
87         init = false;
88     }
89
90     /* step */
91     biquad_res_step();
92
93     /* generate outputs */
94     *out = rtY;
95 }
```

Ajustes a biquad_res.h

```
58 extern void biquad_res(ExtU in, ExtY *out);         // funcion wrapper
```

Ajustes a rtwtypes.h

```
25 #include <ap_fixed.h>
```

...

```
70 typedef ap_fixed<32, 8> real_T;
```

Nota: El oscilador también funciona perfectamente a resolución de 16 bits.

Carpeta "Test Bench"

Archivo ert_main.cpp

```
#include <stddef.h>
#include <stdio.h>
#include "biquad_res.h"

#define FS          16000       // frecuencia de muestreo a 16 KHz
#define SIM_TIME_S  1           // tiempo de simulacion en segundos

const int simTime    = SIM_TIME_S * FS;

int_T main()
{
    ExtU in;
    ExtY out;
    in.In1 = 5;

    FILE *file = fopen("output_2.txt", "w");
    if (file == NULL) return 1;

    int i;
    for (i = 0; i < simTime; i++)
    {
        biquad_res(in, &out);
        fprintf(file, "%f\n", out.Out1.to_float());
        in.In1 = 0;
    }

    fclose(file);

    return 0;
}
```

Script de gráfico de MATLAB

```
Fs = 16000;
SimTime = 1 * Fs;

data = load('output_1.txt');
data2 = load('output_2.txt');
```

```
time = (0:length(data)-1) / SimTime;

subplot(2,1,1)
plot(time, data(:,1)); % Plot first oscillator
xlabel('Tiempo (s)');
ylabel('Salida');
xlim([0 0.05])
ylim([-1.2 1.2])
title('Oscilador biquad 130.8 Hz original');

subplot(2,1,2)
plot(time, data2(:,1),'r'); % Plot first oscillator
xlabel('Tiempo (s)');
ylabel('Salida');
xlim([0 0.05])
ylim([-1.2 1.2])
title('Oscilador biquad 130.8 Hz disenado en Simulink');
```

B.3. RLC en ORTiS

B.3.1. Definición de netlist

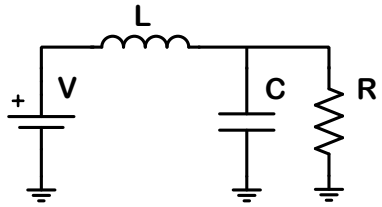


Figura B.2: Circuito RLC

```
#name RLC_Circuit
#const DT 1e-3
#const V  100.0
#const RV 0.001
#const R  10.0
#const L  1e-3
#const C  10.0e-3

% components
VoltageSource vs (V,RV) {1,0}
Resistor res (R) {2,0}
```

```
Inductor ind (DT,L) {1,2}
Capacitor cap (DT,C) {2,0}
```

B.3.2. Post-generación de código C

El código generado por ORTiS se guarda en el archivo `RLC_Circuit.hpp`.

Carpeta "Sources"

Archivo RLC_top.cpp

```
#include "RLC_top.hpp"

void RLC_top(
    real_t x_out[2],
    real_t* l_current_ind
)
{
    // directivas adicionales
    #pragma HLS ALLOCATION operation instances=mul limit=1
    #pragma HLS ALLOCATION operation instances=add limit=1
    #pragma HLS ALLOCATION operation instances=sub limit=1

    // llamada a funcion
    RLC_Circuit_v2_solver<0,real_t>(x_out, l_current_ind);
}
```

Archivo RLC_top.h

```
#ifndef RLC_TOP_HPP_
#define RLC_TOP_HPP_

#include "RLC_types.hpp"
#include "RLC_Circuit.hpp"

void RLC_top(
    real_t x_out[2],
    real_t* l_current_ind
);

#endif
```

Archivo RLC_types.cpp

```cpp
#ifndef RLC_TYPES_HPP_
#define RLC_TYPES_HPP_

#include <ap_fixed.h>

typedef ap_fixed<32,16> real_t;

#endif
```

Carpeta "Test Bench"

Archivo RLC_testbench.cpp

```cpp
#include <stdio.h>
#include "ap_int.h"
#include "RLC_top.hpp"

const double DT = 1.0e-3;
const double TFINAL = 1;
const long NUM_PASOS = long(TFINAL/DT);

int main() {
    real_t x_out[2];
    real_t I_current_ind;

    FILE *file = fopen("output_rlc.txt", "w");
    if (file == NULL) return 1;

    for(unsigned long step = 0; step < NUM_PASOS; step++)
    {
        RLC_top(x_out, &I_current_ind);
    for(const auto& x : x_out)
    {
      fprintf(file, "%f ", x.to_float());
    }
        fprintf(file, "%f\n", I_current_ind.to_float());
    }
    fclose(file);
    return 0;
}
```

Script de gráfico de MATLAB

```
Fs = 1000;
SimTime = 1 * Fs;

data = load('output_rlc.txt');
time = (0:length(data)-1) / SimTime;

plot(time, data(:,1));
hold on;
plot(time, data(:,2));
xlabel('Tiempo (s)');
ylabel('Salida (V)');
xlim([0 1])
ylim([0 200])
title('Simulacion de circuito RLC generado por ORTiS');
legend('Voltaje de entrada', 'Voltaje en la carga');
```

B.4. Inversor trifásico en ORTiS

Definición de netlist

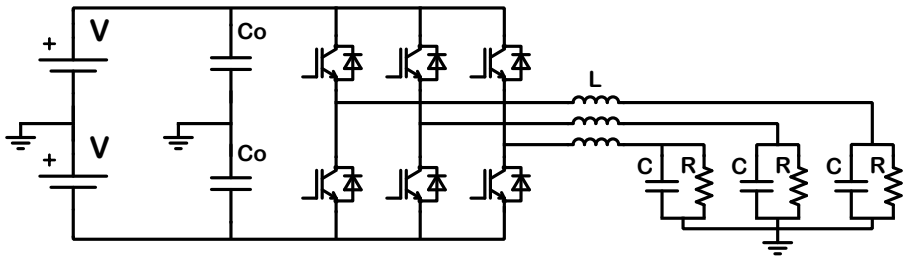


Figura B.3: Inversor trifásico

```
#name ExampleInverterSystem

% model parameters
#const DT 50.0e-6
#const VG 1000.0
#const RG 0.001
#const CI 0.1
#const VTH 1.0
#const LO 0.001
```

```
#const RO 0.0
#const CO 1.0e-6
#const RLOAD 10.0

% system netlist

VoltageSource vgp(VG,RG) {1,0}
VoltageSource vgn(VG,RG) {0,2}

BridgeConverter_3LegIdealSwitchesAntiParallelDiodes converter(DT, CI,
    LO, RO, VTH) {1,0,2,3,4,5}

Capacitor co1(DT,CO) {3,0}
Capacitor co2(DT,CO) {4,0}
Capacitor co3(DT,CO) {5,0}

Resistor rload1(RLOAD) {3,0}
Resistor rload2(RLOAD) {4,0}
Resistor rload3(RLOAD) {5,0}
```

B.5. Simscape y HDL Coder

Los sistemas de rectificadores simulados en el capítulo 3 requieren como entrada señales sinusoidales, lo que no se puede generar en Verilog/SystemVerilog si se desea hacer un testbench del código generado. Por ende, se emplean dos scripts de Python para generar arreglos que contengan los puntos de estas señales.

Rectificador de media onda

```
import numpy as np

type = input("Verilog (v) or SystemVerilog (sv)? ")
bit_width = int(input("Bit width: "))
dec_res = int(input("Decimal places: "))

num_samples = 100
amplitude = 10  # max amplitude for sine wave is 15, scaled to fit signed
    fixed-point representation
scaling_factor = (1 << dec_res)  # fixed-point fractional part
```

```
10
11  # sine wave values
12  print("\nGenerating sine wave table...")
13  sine_wave = [int(amplitude * np.sin(2 * np.pi * x / num_samples) *
        scaling_factor) for x in range(num_samples)]
14
15  format = ""
16  if (type == "sv"):
17      format = "System"
18  print(f"Writing sine wave table in {format}Verilog format...\n")
19  # print the values to fit verilog
20  with open('sine_hwbr.txt', 'w') as f:
21      space = ''
22      if (type == "v"):
23          f.write(f"reg signed [{bit_width-1}:0] sine [{num_samples-1}:0];\
        n\n")
24          for i,value in enumerate(sine_wave):
25              if i < 10:
26                  space = ' '
27              else:
28                  space = ''
29              if value < 0:
30                  value = (1 << bit_width) + value  # convert to unsigned
        if negative
31              str_aux = f"sine[{i}]{space} = {bit_width}'b{value:0{
        bit_width}b};\n"
32              str = str_aux[0:-(dec_res+2)] + "_" + str_aux[-(dec_res+2):]
33              f.write(str)
34      elif (type == "sv"):
35          f.write(f"localparam signed [{bit_width-1}:0] sine [{num_samples
        }] = '{{\n")
36          space = ","
37          for i,value in enumerate(sine_wave):
38              if (i == num_samples-1):
39                  space = ""
40              if value < 0:
41                  value = (1 << bit_width) + value  # convert to unsigned
        if negative
42              str_aux = f"    {bit_width}'b{value:0{bit_width}b}{space}\n"
43              str = str_aux[0:-(dec_res+1+len(space))] + "_" + str_aux[-(
        dec_res+1+len(space)):]
44              f.write(str)
```

```
        f.write("};")

print("Done! File generated at sine_hwbr.txt")
```

Puente rectificador de onda completa

```
import numpy as np
import matplotlib.pyplot as plt

type = input("Verilog (v) or SystemVerilog (sv)? ")
bit_width = int(input("Bit width: "))
dec_res = int(input("Decimal places: "))

sample_time = 1e-5
frequency = 60
num_samples = int((1/sample_time)/frequency)

amplitude = 170
scaling_factor = (1 << dec_res)  # fixed-point fractional part

# sine wave values
print("\nGenerating sine wave table...")
sine_wave = [int(amplitude * np.sin(2 * np.pi * x / num_samples) *
    scaling_factor) for x in range(num_samples)]

format = ""
if (type == "sv"):
    format = "System"
print(f"Writing sine wave table in {format}Verilog format...\n")

# print the values to fit verilog
with open('sine_fwbr.txt', 'w') as f:
    space = ''
    if (type == "v"):
        f.write(f"reg signed [{bit_width-1}:0] sine [{num_samples-1}:0];\
    n\n")
        for i,value in enumerate(sine_wave):
            if i < 10:
                space = ' '
            else:
                space = ''
```

```
            if value < 0:
                value = (1 << bit_width) + value  # convert to unsigned if negative
            str_aux = f"sine[{i}]{space} = {bit_width}'b{value:0{bit_width}b};\n"
            str = str_aux[0:-(dec_res+2)] + "_" + str_aux[-(dec_res+2):]
            f.write(str)
    elif (type == "sv"):
        f.write(f"localparam signed [{bit_width-1}:0] sine [{num_samples}] = '{{\n")
        space = ","
        for i,value in enumerate(sine_wave):
            if (i == num_samples-1):
                space = ""
            if value < 0:
                value = (1 << bit_width) + value  # convert to unsigned if negative
            str_aux = f"    {bit_width}'b{value:0{bit_width}b}{space}\n"
            str = str_aux[0:-(dec_res+1+len(space))] + "_" + str_aux[-(dec_res+1+len(space)):]
            f.write(str)
        f.write("};")

print("Done! File generated at sine_fwbr.txt")
```

BIBLIOGRAFÍA

[1] M. D. Omar Faruque, T. Strasser, G. Lauss, V. Jalili-Marandi, P. Forsyth, C. Dufour, V. Dinavahi, A. Monti, P. Kotsampopoulos, J. A. Martinez, K. Strunz, M. Saeedifard, X. Wang, D. Shearer, and M. Paolone, "Real-Time Simulation Technologies for Power Systems Design, Testing, and Analysis," *IEEE Power and Energy Technology Systems Journal*, vol. 2, no. 2, pp. 63–73, 2015.

[2] J. J. Rodríguez-Andina, M. D. Valdés-Peña, and M. J. Moure, "Advanced Features and Industrial Applications of FPGAs—A Review," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, 2015.

[3] S. Sinha and T. Srikanthan, "High-Level Synthesis: Boosting Designer Productivity and Reducing Time to Market," *IEEE Potentials*, vol. 34, no. 4, pp. 31–35, 2015.

[4] Michigan State University, "Application Note-Purpose and Expectations," [consulta: 2024-06-16]. [Online]. Available: https://www.egr.msu.edu/classes/ece480/capstone/docs/app_note.html

[5] Analog Devices, "LTspice," [consulta: 2024-05-05]. [Online]. Available: https://www.analog.com/en/resources/design-tools-and-calculators/ltspice-simulator.html

[6] Varios, "Qucs-S: Qucs with SPICE," [consulta: 2024-02-20]. [Online]. Available: https://ra3xdh.github.io/

[7] Texas Instruments, "TINA-TI Simulation tool," [consulta: 2024-03-02]. [Online]. Available: https://www.ti.com/tool/TINA-TI

[8] Plexim, "PLECS - The Simulation Platform for Power Electronic Systems," [consulta: 2024-02-24]. [Online]. Available: https://www.plexim.com/products/plecs

[9] National Instruments, "NI Multisim™," [consulta: 2024-03-02]. [Online]. Available: https://www.ni.com/es/shop/electronic-test-instrumentation/application-software-for-electronic-test-and-instrumentation-category/what-is-multisim.html

[10] MathWorks, "Simulación y diseño basado en modelos con Simulink," [consulta: 2024-04-24]. [Online]. Available: https://la.mathworks.com/products/simulink.html

[11] Powersim, "PSIM: The ultimate simulation environment for power conversion and motor control." [consulta: 2024-10-06]. [Online]. Available: https://powersimtech.com/products/psim/capabilities-applications/

[12] National Instruments, "LabVIEW," [consulta: 2024-03-02]. [Online]. Available: https://www.ni.com/es/shop/labview.html

[13] MathWorks, "MATLAB," [consulta: 2024-02-22]. [Online]. Available: https://la.mathworks.com/products/matlab.html

[14] MathWorks, "MATLAB Coder," [consulta: 2024-02-22]. [Online]. Available: https://la.mathworks.com/products/matlab-coder.html

[15] John W. Eaton, "GNU Octave," [consulta: 2024-02-27]. [Online]. Available: https://octave.org/

[16] Dassault Systèmes, "Scilab," [consulta: 2024-02-27]. [Online]. Available: https://www.scilab.org/

[17] AMD, "Vitis™ Unified Software Platform," [consulta: 2024-03-04]. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html

[18] Intel, "Quartus® Prime Design Software," [consulta: 2024-02-28]. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[19] Intel, "DSP Builder for Intel® FPGAs," [consulta: 2024-02-28]. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/dsp-builder.html

[20] C. Dufour, J. Mahseredjian, and J. Bélanger, "A Combined State-Space Nodal Method for the Simulation of Power System Transients," *IEEE Transactions on Power Delivery*, vol. 26, no. 2, pp. 928–935, 2011.

[21] R. Voiculescu and M. Iordache, "Analog circuit simulation by state variable method," *ResearchGate GmbH*, vol. 77, pp. 213–224, 01 2015.

[22] J. Schutt-Aine, "Latency insertion method (lim) for the fast transient simulation of large networks," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 1, pp. 81–89, 2001.

[23] A. Benigni and A. Monti, "A parallel approach to real-time simulation of power electronics systems," *IEEE Transactions on Power Electronics*, vol. 30, no. 9, pp. 5192–5206, 2015.

[24] M. Lordache, S. Deleanu, C. Curteanu, N. Galan, and A.-A. Moscu, "SYSEG — Symbolic state equation generation," in *2017 International Conference on Electromechanical and Power Systems (SIELMEN)*, 2017, pp. 184–190.

[25] Y. F. Oliveira, F. A. La-Gatta, R. A. F. Ferreira, and M. C. B. P. Rodrigues, "Development of a FPGA-Based Real-Time Simulation System," in *2019 IEEE 15th Brazilian Power Electronics Conference and 5th IEEE Southern Power Electronics Conference (COBEP/SPEC)*, 2019, pp. 1–6.

[26] W. Li and J. Bélanger, "An Equivalent Circuit Method for Modelling and Simulation of Modular Multilevel Converters in Real-Time HIL Test Bench," *IEEE Transactions on Power Delivery*, vol. 31, no. 5, pp. 2401–2409, 2016.

[27] A. Kiffe and T. Schulte, "FPGA-based hardware-in-the-loop simulation of a rectifier with power factor correction," in *2015 17th European Conference on Power Electronics and Applications (EPE'15 ECCE-Europe)*, 2015, pp. 1–8.

[28] M. Milton and A. Benigni, "Latency Insertion Method Based Real-Time

Simulation of Power Electronic Systems," *IEEE Transactions on Power Electronics*, vol. 33, no. 8, pp. 7166–7177, 2018.

[29] F. Montano, T. Ould-Bachir, and J. P. David, "An Evaluation of a High-Level Synthesis Approach to the FPGA-Based Submicrosecond Real-Time Simulation of Power Converters," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 1, pp. 636–644, 2018.

[30] Jiménez, Lucía, I. Urriza, L. A. Barragan, D. Navarro, and V. Dinavahi, "Implementation of an FPGA-Based Online Hardware-in-the-Loop Emulator Using High-Level Synthesis Tools for Resonant Power Converters Applied to Induction Heating Appliances," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 4, pp. 2206–2214, 2015.

[31] L. Koleff, M. Conde, P. Hayashi, F. Sacco, K. Enomoto, E. Pellini, W. Komatsu, and L. Matakas, "Development of a FPGA-Based Control System for Modular Multilevel Converter Applications," in *2019 IEEE 15th Brazilian Power Electronics Conference and 5th IEEE Southern Power Electronics Conference (COBEP/SPEC)*, 2019, pp. 1–6.

[32] Y. P. Siwakoti and G. E. Town, "Design of FPGA-controlled power electronics and drives using MATLAB Simulink," in *2013 IEEE ECCE Asia Downunder*, 2013, pp. 571–577.

[33] T. Sasaki, H. Hosoyama, Y. Yonezawa, A. Manabe, K. Huang, X. Liu, J. Chen, J. Kaneko, and Y. Nakashima, "Production code generation for server power supply controller," in *2015 IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2015, pp. 2656–2663.

[34] E. T. Mekonnen, J. Katcha, and M. Parker, "An fpga-based digital control development method for power electronics," in *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, 2012, pp. 222–226.

[35] AMD, "RTL-Level Design Simulation," [consulta: 2024-04-24]. [Online]. Available: https://docs.amd.com/r/en-US/ug895-vivado-system-level-design-entry/RTL-Level-Design-Simulation

[36] Marian K. Kazimierczuk, *Pulse-Width Modulated DC-DC Power Converters*, 2nd ed. John Wiley & Sons Inc, 2015, pp. 1–3.

[37] Ned Mohan, Tore M. Undeland, William P. Robbins, *Power Electronics: Converters, Applications, and Design*, 3rd ed. John Wiley & Sons Inc, 2002, p. 3.

[38] Krzysztof Sozański, *Digital Signal Processing in Power Electronics Control Circuits*, 2nd ed. Springer, 2017, p. 2.

[39] Arduino, "Arduino.cl - Plataforma de Código Abierto para Electrónica," [consulta: 2024-09-03]. [Online]. Available: https://arduino.cl/

[40] STMicroelectronics NV, "STM32 Microcontrollers (MCUs) - STMicroelectronics," [consulta: 2024-09-03]. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[41] Arm, "Cortex-M7," [consulta: 2024-09-03]. [Online]. Available: https://developer.arm.com/Processors/Cortex-M7

[42] Intel, "Procesadores Intel® Core™ i7," [consulta: 2024-09-03]. [Online]. Available: https://www.intel.la/content/www/xl/es/products/details/processors/core/i7.html

[43] Kun-Shan Lin, Frantz, G.A., Simar, R., "The TMS320 family of digital signal processors," vol. 75, no. 9, pp. 1143–1159, 1987.

[44] AMD, "Kintex™ 7 FPGAs," [consulta: 2024-09-03]. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-7.html

[45] Intel, "FPGA Agilex™ 7," [consulta: 2024-09-03]. [Online]. Available: https://www.intel.la/content/www/xl/es/products/details/fpga/agilex/7.html

[46] AMD, "Zynq™ 7000 SoCs," [consulta: 2024-09-03]. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

[47] Broadcom, "BCM2711 ARM Peripherals," [consulta: 2024-09-03]. [Online]. Available: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

[48] Digilent, "Zybo Z7," [consulta: 2024-09-03]. [Online]. Available: https://digilent.com/reference/programmable-logic/zybo-z7/start

[49] Microchip Technology, "PolarFire® SoC Discovery Kit," [consulta: 2024-09-03]. [Online]. Available: https://www.microchip.com/en-us/development-tool/mpfs-disco-kit

[50] Raspberry Pi Foundation, "Raspberry Pi," [consulta: 2024-09-03]. [Online]. Available: https://www.raspberrypi.com/

[51] BeagleBoard, "BeagleBone® Black," [consulta: 2024-09-13]. [Online]. Available: https://www.beagleboard.org/boards/beaglebone-black

[52] Arduino, "Portenta H7," [consulta: 2024-09-03]. [Online]. Available: https://docs.arduino.cc/hardware/portenta-h7/

[53] Frank Vahid, *Digital Design with RTL Design, Verilog and VHDL*, 2nd ed. John Wiley & Sons Inc, 2010, p. 247.

[54] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[55] AMD, "Benefits of High-Level Synthesis • Vitis High-Level Synthesis User Guide (UG1399)," [consulta: 2024-03-25]. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Benefits-of-High-Level-Synthesis

[56] AMD, "Vitis HLS," [consulta: 2024-09-08]. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

[57] AMD, "C++ Arbitrary Precision Fixed-Point Types • Vitis High-Level Synthesis User Guide (UG1399)," [consulta: 2024-05-12]. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/C-Arbitrary-Precision-Fixed-Point-Types

[58] Julius Orion Smith III, "Direct Form II," [consulta: 2024-04-23]. [Online]. Available: https://ccrma.stanford.edu/~jos/filters/Direct_Form_II.html

[59] Julius Orion Smith III, "The BiQuad Section," [consulta: 2024-04-23]. [Online]. Available: https://ccrma.stanford.edu/~jos/fp2/BiQuad_Section.html

[60] AMD, "Floats and Doubles • Vitis High-Level Synthesis User Guide (UG1399)," [consulta: 2024-04-26]. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Floats-and-Doubles

[61] MathWorks, "Embedded Coder - Generación de código C y C++ optimizado para sistemas integrados," [consulta: 2024-04-24]. [Online]. Available: https://la.mathworks.com/products/embedded-coder.html

[62] M. Milton and A. Benigni, "ORTiS solver codegen: C++ code generation tools for high performance, FPGA-based, real-time simulation of power electronic systems," *SoftwareX*, vol. 13, p. 100660, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711021000054

[63] MathWorks, "Simscape," [consulta: 2024-04-26]. [Online]. Available: https://la.mathworks.com/products/simscape.html

[64] MathWorks, "Bibliotecas de bloques de Simscape," [consulta: 2024-04-26]. [Online]. Available: https://la.mathworks.com/help/simscape/ug/introducing-the-simscape-block-libraries.html

[65] MathWorks, "HDL Coder," [consulta: 2024-05-03]. [Online]. Available: https://la.mathworks.com/help/hdlcoder/

[66] Marian K. Kazimierczuk, *Pulse-Width Modulated DC-DC Power Converters*, 2nd ed. John Wiley & Sons Inc, 2015, p. 23.

[67] Marian K. Kazimierczuk, *Pulse-Width Modulated DC-DC Power Converters*, 2nd ed. John Wiley & Sons Inc, 2015, p. 91.

[68] AMD, "Artix 7 FPGAs," [consulta: 2024-04-28]. [Online]. Available: https://www.amd.com/es/products/adaptive-socs-and-fpgas/fpga/artix-7.html

[69] AMD, "7 Series Product Tables and Product Selection Guide (XMP101)," [consulta: 2024-04-28]. [Online]. Available: https://docs.amd.com/v/u/en-US/7-series-product-selection-guide

[70] AC3E, "SE-002PL-DC-E-GN-006E - BRAIn - Características técnicas," Agosto 2018.

[71] AC3E, "SE-002PL-DC-E-MA-002E - Manual de programación de tarjeta de control," pp. 13–16, Noviembre 2016.

[72] Digilent, "Pmod DA4 - Digilent Reference," [consulta: 2024-09-12]. [Online]. Available: https://digilent.com/reference/pmod/pmodda4/start

[73] Digilent, "Digilent Pmod™ Interface Specification 1.2.0," [consulta: 2024-09-12]. [Online]. Available: https://digilent.com/reference/_media/reference/pmod/pmod-interface-specification-1_2_0.pdf