INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

TRABAJO TERMINAL I

Aplicación Multiplataforma para el Aprendizaje y Depuración de Código HDL sin Requerimiento de Hardware Físico

No. de Registro: TT 2026-B059

PRESENTA:

Ulrich Tamayo Daniel

DIRECTORES:

M. en C. Pastrana Fernández Carlos Jesús

Dr. Cifuentes Álvarez Alejandro Sigrido

Ciudad de México, Febrero de 2026

Advertencia

Este trabajo fue realizado en la Escuela Superior de Cómputo del Instituto Politécnico Nacional, como parte de los requisitos para obtener el título de Ingeniero en Sistemas Computacionales. La utilización e implementación de los resultados obtenidos se limita a los alcances definidos en este documento.

El autor autoriza al Instituto Politécnico Nacional para que este documento sea consultado con fines académicos y de investigación, siempre que se cite la fuente correspondiente.

Ciudad de México, a _____ de _____ de _____

_____

Ulrich Tamayo Daniel

Resumen

El presente trabajo terminal aborda la primera fase de desarrollo de Kmila, una aplicación multiplataforma de escritorio orientada al aprendizaje y depuración de código en VHDL (VHSIC Hardware Description Language), ejecutada en entorno local y sin requerimiento de hardware especializado (FPGA). La enseñanza del diseño de sistemas digitales enfrenta barreras significativas: el costo de tarjetas FPGA como la Altera Cyclone IV (~$1,866 MXN) o la Basys 3 (~$3,400 MXN), las limitaciones de las versiones gratuitas de herramientas comerciales como ModelSim o Quartus Prime, y la fragmentación del flujo de trabajo en herramientas de código abierto como GHDL y GTKWave. Ante este panorama, se propone una plataforma integrada, gratuita y de código abierto que unifique edición, simulación y visualización de señales digitales en una sola interfaz.

En esta primera entrega (TT1) se presentan el análisis de requerimientos, el diseño arquitectónico del sistema y la construcción de la interfaz gráfica unificada. La aplicación, desarrollada con .NET 9 mediante MAUI para plataformas nativas (Windows, macOS, Android, iOS) y Blazor para la versión web (Linux), integra un editor de código basado en Monaco Editor con resaltado de sintaxis para VHDL, un sistema de gestión de proyectos y archivos con persistencia en SQLite, un selector de dispositivos FPGA con especificaciones de 35 modelos de fabricantes como AMD/Xilinx, Intel/Altera y Lattice, un panel de configuración de parámetros de simulación, y un visualizador de formas de onda basado en Chart.js. Adicionalmente, la interfaz cuenta con soporte de internacionalización en siete idiomas. El soporte para Verilog se contempla como extensión futura.

El motor de simulación, que comprende el analizador sintáctico, el intérprete de ejecución y el módulo de control temporal, se desarrollará en la segunda fase (TT2).

Palabras clave: Academia de Ingeniería de Software, Departamento de Sistemas Electrónicos, Educación en Lenguaje Descriptivo de Hardware, Simulación Digital, Interfaz Multiplataforma

Abstract

This thesis addresses the first development phase of Kmila, a cross-platform desktop application aimed at learning and debugging VHDL (VHSIC Hardware Description Language) code, running locally without requiring specialized hardware (FPGA). The teaching of digital systems design faces significant barriers: the cost of FPGA boards such as the Altera Cyclone IV (~$1,866 MXN) or the Basys 3 (~$3,400 MXN), the limitations of free versions of commercial tools like ModelSim or Quartus Prime, and the fragmented workflow of open-source tools such as GHDL and GTKWave. In light of this, an integrated, free, and open-source platform is proposed that unifies code editing, simulation, and digital signal visualization within a single interface.

In this first deliverable (TT1), the requirements analysis, the system's architectural design, and the construction of the unified graphical interface are presented. The application, built with .NET 9 using MAUI for native platforms (Windows, macOS, Android, iOS) and Blazor for the web version (Linux), integrates a code editor based on Monaco Editor with VHDL syntax highlighting, a project and file management system with SQLite persistence, an FPGA device selector featuring specifications for 34 models from manufacturers such as AMD/Xilinx, Intel/Altera, and Lattice, a simulation parameter configuration panel, and a Chart.js-based waveform viewer. Additionally, the interface provides internationalization support for seven languages. Verilog support is planned as a future extension.

The simulation engine, comprising the syntactic analyzer, the execution interpreter, and the temporal control module, will be developed in the second phase (TT2).

Keywords: Software Engineering Academy, Electronic Systems Department, Hardware Description Language Education, Digital Simulation, Cross-platform Interface

Índice de Tablas

Índice de Figuras

Glosario y Acrónimos

Este glosario recopila los términos técnicos y acrónimos empleados a lo largo del documento. Cuando un mismo acrónimo admite más de una interpretación (por ejemplo, **RTL**, **DSP** o **ISP**), se incluyen entradas separadas indicando el sentido específico con el que aparece cada uno en el texto.

Término	Definición
Amaranth HDL	Lenguaje de descripción de hardware embebido en Python, sucesor conceptual de MyHDL. Genera Verilog como formato de salida para síntesis.
ASIC	*Application-Specific Integrated Circuit* — circuito integrado de aplicación específica, fabricado para una función concreta y no reconfigurable una vez manufacturado.
AST	*Abstract Syntax Tree* — árbol de sintaxis abstracta; representación intermedia del código fuente tras el análisis sintáctico.
Bitstream	Archivo binario resultado del proceso de síntesis, place-and-route y generación, que configura físicamente la lógica interna de un FPGA.
Blazor	Framework de Microsoft para construir interfaces web interactivas con C# sobre WebAssembly o servidor.
BRAM	*Block RAM* — bloque de memoria RAM dedicado embebido en la matriz lógica del FPGA, empleado para implementar memorias síncronas de hasta decenas de Kbits.
BSV	*Bluespec SystemVerilog* — lenguaje de descripción de hardware con paradigma de reglas guardadas (*guarded atomic actions*) desarrollado en el MIT.
C4	Modelo de arquitectura de software propuesto por Simon Brown con cuatro niveles jerárquicos: contexto, contenedores, componentes y código.
Chisel	*Constructing Hardware In a Scala Embedded Language* — DSL embebido en Scala para generar hardware parametrizable; base del ecosistema RISC-V y del procesador Rocket Chip.
CLB	*Configurable Logic Block* — bloque lógico configurable; unidad básica de la arquitectura de FPGAs de Xilinx, compuesta por LUTs, flip-flops y lógica de acarreo.
CPLD	*Complex Programmable Logic Device* — dispositivo lógico programable complejo; antecesor arquitectónico del FPGA, basado en matrices de términos producto y memoria no volátil.
DCM	*Digital Clock Manager* — bloque de gestión de reloj en FPGAs de Xilinx (familia Spartan-6), capaz de realizar multiplicación, división y desfase de frecuencia.
DSL	*Domain-Specific Language* — lenguaje de dominio específico; lenguaje diseñado para un problema o dominio particular (por ejemplo, Chisel o SpinalHDL para descripción de hardware).
DSP (disciplina)	*Digital Signal Processing* — procesamiento digital de señales; área del tratamiento matemático de señales muestreadas.

Término	Definición
DSP (bloque)	Bloque de hardware dedicado al procesamiento aritmético integrado en los FPGAs. Recibe distintos nombres según el fabricante: *DSP48A1*/*DSP48E1* en Xilinx, *multiplicador 18x18* en Intel/Altera, y *MAC 16x16* o *multiplicador 18x18* en Lattice. Es el sentido empleado en las tablas de dispositivos.
E^{2}PROM	*Electrically Erasable Programmable Read-Only Memory* — memoria no volátil borrable eléctricamente. Tecnología de base de los dispositivos GAL y de la configuración no volátil en familias como iCE40 e ispLSI.
EBR	*Embedded Block RAM* — denominación de Lattice para los bloques de memoria RAM dedicada embebidos en sus FPGAs (equivalente a BRAM en Xilinx y M9K en Intel/Altera Cyclone IV).
ESCOM	Escuela Superior de Cómputo del Instituto Politécnico Nacional.
ESL	*Electronic System Level* — nivel de abstracción de sistema electrónico, empleado por lenguajes como SystemC para modelado arquitectónico temprano.
FF	*Flip-Flop* — elemento de memoria básico biestable utilizado como unidad de almacenamiento síncrono en el hardware digital. Una columna dedicada en los catálogos de FPGAs reporta el número disponible.
FIRRTL	*Flexible Intermediate Representation for RTL* — representación intermedia utilizada por Chisel como paso previo a la generación de Verilog.
FPGA	*Field-Programmable Gate Array* — dispositivo de lógica programable en campo; circuito integrado reconfigurable por el usuario final después de su fabricación.
GAL	*Generic Array Logic* — familia de dispositivos lógicos programables no volátiles basados en E^{2}PROM, comercializada originalmente por Lattice a partir de 1985.
GHDL	Analizador, simulador y compilador de código abierto para VHDL.
GTKWave	Visor de formas de onda electrónicas de código abierto, ampliamente utilizado con archivos VCD y FST.
HDL	*Hardware Description Language* — lenguaje de descripción de hardware; categoría genérica que incluye VHDL, Verilog y SystemVerilog.
HLS	*High-Level Synthesis* — síntesis de alto nivel; traducción de código C/C++ o SystemC a RTL sintetizable en VHDL o Verilog.
IDE	*Integrated Development Environment* — entorno integrado de desarrollo que agrupa editor, compilador, depurador y otras herramientas en una sola aplicación.
IPN	Instituto Politécnico Nacional.
ISP (arquitectura)	*Instruction Set Processor* — formalismo propuesto por Bell y Newell en 1971 para describir la arquitectura de procesadores a nivel de transferencia de registros. Aparece en la Sección 2.1.1.
ISP (electrónica)	*In-System Programming* — técnica que permite reprogramar dispositivos lógicos no volátiles ya montados en la placa del circuito. Es el origen del nombre comercial de la familia *ispLSI* de Lattice.
Kmila	Nombre propio del software de simulación y depuración de código HDL desarrollado en el presente Trabajo Terminal.

Término	Definición
LE	*Logic Element* — elemento lógico; unidad básica de la arquitectura de FPGAs de Intel/Altera, compuesta por una LUT de 4 entradas, un registro y lógica auxiliar.
LUT	*Look-Up Table* — tabla de consulta; elemento lógico configurable fundamental de todo FPGA. El número de entradas varía según el fabricante y la familia: 4 entradas en Intel/Altera Cyclone IV y Lattice iCE40; 6 entradas en AMD/Xilinx series 6 y 7 y en Lattice ECP5.
MAC	*Multiply-Accumulate* — operación de multiplicación y acumulación fusionada; en los FPGAs Lattice iCE40 UltraPlus designa el tipo de bloque DSP disponible (MAC 16×16).
MAUI	*.NET Multi-platform App UI* — framework de Microsoft para construir aplicaciones nativas multiplataforma (Windows, macOS, Android, iOS) sobre .NET.
MMCM	*Mixed-Mode Clock Manager* — gestor mixto de reloj; evolución del DCM en FPGAs de Xilinx (serie 7 y posteriores) con capacidad de síntesis de frecuencia de precisión fraccionaria.
Monaco Editor	Editor de código embebible desarrollado por Microsoft, núcleo de Visual Studio Code, integrado en Kmila para la edición de archivos HDL.
MyHDL	Biblioteca de Python para descripción de hardware digital con capacidad de conversión a Verilog y VHDL.
Netlist	Lista de componentes lógicos primitivos y sus interconexiones, producto de la etapa de síntesis.
Place-and-Route	Etapa del flujo de diseño FPGA que asigna los elementos del netlist a recursos físicos (LUTs, flip-flops, BRAM) y determina el enrutamiento de las interconexiones.
PLL	*Phase-Locked Loop* — lazo de seguimiento de fase; circuito integrado de los FPGAs usado para generación y síntesis de frecuencias de reloj.
PSL	*Property Specification Language* — lenguaje de especificación de propiedades, estandarizado como IEEE 1850 e integrado en VHDL 2008.
RTL (abstracción)	*Register-Transfer Level* — nivel de transferencia de registros; nivel de abstracción en diseño digital que describe el circuito en términos de registros y la lógica combinacional que transfiere datos entre ellos en cada ciclo de reloj. Es el nivel empleado para síntesis.
RTL (escritura)	*Right-to-Left* — dirección de escritura de derecha a izquierda, empleada por idiomas como el árabe y el hebreo. En Kmila se aplica al idioma árabe mediante el atributo HTML `dir=rtl"`.
SerDes	*Serializer/Deserializer* — bloque transceptor de alta velocidad que convierte entre flujos de datos serie y paralelo, integrado en familias FPGA de gama media y alta (Artix-7, ECP5).
SoC	*System-on-Chip* — sistema en chip; circuito integrado que combina procesador, memoria, periféricos y eventualmente lógica programable en un único dispositivo (por ejemplo, la familia Xilinx Zynq).
SpinalHDL	Lenguaje de descripción de hardware embebido en Scala, alternativa a Chisel con énfasis en la legibilidad. Compila a Verilog y VHDL.

Término	Definición
SQLite	Motor de base de datos relacional embebido, sin servidor, utilizado en Kmila para la persistencia local de proyectos y configuración.
SVA	*SystemVerilog Assertions* — subconjunto de SystemVerilog orientado a la verificación formal y funcional mediante aserciones.
Síntesis lógica	Proceso de traducción de código RTL (en VHDL o Verilog) a un netlist de compuertas lógicas a partir de una biblioteca tecnológica específica.
SystemC	Biblioteca de C++ estandarizada como IEEE 1666 para el modelado de hardware a nivel de sistema (ESL).
SystemVerilog	Extensión de Verilog que añade programación orientada a objetos, aserciones (SVA), cobertura funcional y verificación con restricciones aleatorias; estandarizado como IEEE 1800.
Testbench	Banco de pruebas; código HDL (no sintetizable) que genera estímulos de entrada y verifica las salidas del circuito bajo prueba durante la simulación.
TT	Trabajo Terminal; proyecto de titulación en la ESCOM dividido en dos fases, TT1 y TT2.
UVM	*Universal Verification Methodology* — metodología estándar de verificación para ASIC y FPGA basada en SystemVerilog.
VCD	*Value Change Dump* — formato estándar ASCII para registrar cambios de valor en señales durante una simulación; soportado por GTKWave y por Kmila.
Verilog	Lenguaje de descripción de hardware con sintaxis inspirada en C, estandarizado como IEEE 1364 y fusionado con SystemVerilog en IEEE 1800 a partir de 2009.
VHDL	*VHSIC Hardware Description Language* — lenguaje de descripción de hardware desarrollado por el Departamento de Defensa de los Estados Unidos y estandarizado como IEEE 1076. Es el lenguaje primario soportado por Kmila.
VHSIC	*Very High Speed Integrated Circuit* — programa del Departamento de Defensa de los Estados Unidos (1980–1990) que financió el desarrollo de VHDL.
WASM	*WebAssembly* — formato de instrucciones binarias portable diseñado para la ejecución eficiente de código en navegadores web; empleado por Blazor.
Yosys	Suite de síntesis lógica de código abierto con soporte para Verilog y SystemVerilog, ampliamente utilizada con FPGAs Lattice iCE40 y ECP5.

Capítulo 1

Fundamentos del Proyecto

1.1. Introducción

La enseñanza del diseño de sistemas digitales y su implementación en dispositivos lógicos programables, como las Field Programmable Gate Arrays (FPGAs), enfrenta barreras significativas que limitan la formación práctica de los estudiantes. Aunque el dominio de lenguajes de descripción de hardware (HDLs) como VHDL es esencial, la comprensión profunda de la lógica subyacente exige una experiencia de depuración que va más allá de la simple escritura de código.

Las herramientas educativas actuales presentan limitaciones en dos dimensiones principales: económicas y pedagógicas. Por un lado, la dependencia de hardware especializado (tarjetas FPGA) constituye un obstáculo financiero relevante para estudiantes e instituciones. Una tarjeta FPGA Altera Cyclone IV puede costar alrededor de $1,866.00 MXN, mientras que la popular Basys 3 alcanza los $3,400.00 MXN, sin contar accesorios [1], [2]. A ello se suma el precio de cursos especializados, que en plataformas como Coursera puede superar los $1,000 MXN mensuales [3]. Por otro lado, existe un vacío pedagógico en la oferta de plataformas virtuales. Aunque fabricantes como Intel y AMD/Xilinx ofrecen entornos de diseño (Quartus Prime, Vivado), sus versiones gratuitas poseen limitaciones técnicas importantes [4], [5].

En cuanto a herramientas de código abierto, GHDL es un simulador potente, pero su uso está limitado a la línea de comandos y depende de la integración con GTKWave para visualizar señales, lo que genera un flujo de trabajo fragmentado y poco intuitivo para principiantes [6], [7]. En contraste, soluciones comerciales como ModelSim o QuestaSim ofrecen entornos robustos, pero sus licencias educativas restringidas impiden un uso completo en entornos académicos [8].

En la revisión de Trabajos Terminales (TTs) en la ESCOM, no se han encontrado propuestas previas dedicadas a la creación de un depurador interactivo de HDL. Sin embargo, existen antecedentes de proyectos relacionados con simulación, control lógico y sistemas educativos interactivos [9], lo que evidencia una línea institucional que puede ser ampliada con la presente propuesta.

Ante este panorama, el presente trabajo terminal propone el desarrollo de Kmila, una plataforma académica multiplataforma ejecutada como aplicación local, gratuita y de código abierto, enfocada en la depuración lógica y funcional de código VHDL. A diferencia de los simuladores tradicionales, Kmila propone una experiencia integrada con interfaz gráfica intuitiva, visualización de señales digitales y controles de simulación en una sola aplicación, lo que la convierte en una herramienta complementaria para fortalecer la enseñanza del diseño digital en la ESCOM, permitiendo a los estudiantes preparar y validar sus diseños antes de acceder al laboratorio físico, o como alternativa cuando el hardware no esté disponible.

En esta primera fase (TT1), se presenta el análisis de requerimientos del sistema, el diseño de su arquitectura y la construcción de la interfaz gráfica unificada, sentando las bases sobre las cuales se implementará el motor de simulación en la segunda fase (TT2).

1.2. Objetivo General

Desarrollar una aplicación multiplataforma, gratuita y de código abierto, que permita la simulación y depuración interactiva de código en lenguajes de descripción de hardware (HDL) en un entorno local, facilitando el aprendizaje de diseño digital sin requerir hardware especializado (FPGA).

1.3. Objetivos Específicos

1. Analizar los requerimientos funcionales y no funcionales del sistema, identificando las necesidades de los usuarios objetivo (estudiantes y docentes de ESCOM) en el contexto de la enseñanza del diseño digital.
2. Diseñar la arquitectura del sistema considerando una estructura modular que contemple los componentes de edición de código, simulación, visualización de señales y control temporal, utilizando .NET 9 con MAUI y Blazor como frameworks base.
3. Construir una interfaz gráfica unificada que integre un editor de código basado en Monaco Editor con resaltado de sintaxis para VHDL, un sistema de gestión de proyectos y archivos, y un panel de configuración de parámetros de simulación.
4. Implementar un selector de dispositivos FPGA que permita al usuario consultar las especificaciones de recursos de hardware (LUTs, flip-flops, pines I/O, bloques de memoria, DSPs) de múltiples modelos comerciales.
5. Desarrollar un visualizador de formas de onda que permita la representación gráfica de señales digitales, preparado para su integración futura con el motor de simulación.
6. Validar el funcionamiento de la interfaz en su versión web orientada a Linux, verificando la correcta renderización y funcionalidad de los componentes mediante Blazor.

1.4. Antecedentes y Estado del Arte

1.4.1. Contexto Histórico

El diseño digital ha evolucionado significativamente desde la introducción de los primeros lenguajes de descripción de hardware en la década de 1980. VHDL, desarrollado originalmente por el Departamento de Defensa de los Estados Unidos y estandarizado por IEEE en 1987 (IEEE Std 1076) [10], se consolidó como el lenguaje predominante en entornos académicos de Europa y Latinoamérica [11]. Por su parte, Verilog, estandarizado como IEEE Std 1364 [12], encontró mayor adopción en la industria norteamericana y asiática [13].

La enseñanza de estos lenguajes en instituciones como la ESCOM se ha apoyado históricamente en el uso de tarjetas FPGA físicas, principalmente modelos de la familia Cyclone de Intel/Altera y Spartan/Artix de AMD/Xilinx. Sin embargo, la disponibilidad limitada de este hardware en los laboratorios, combinada con su costo, ha restringido la práctica individual de los estudiantes fuera del aula.

En respuesta a estas limitaciones, ha surgido un ecosistema diverso de herramientas de simulación y desarrollo que puede clasificarse en cuatro categorías: herramientas de código abierto, herramientas comerciales, plataformas web y herramientas académicas especializadas. A continuación se analiza cada categoría en detalle, evaluando sus capacidades, limitaciones y pertinencia para el contexto educativo que motiva el presente proyecto.

1.4.2. Herramientas de Código Abierto

GHDL es un analizador, compilador y simulador de código abierto para VHDL, desarrollado principalmente por Tristan Gingold bajo licencia GPLv2 [6]. A diferencia de los simuladores interpretados, GHDL compila el código VHDL a código máquina nativo mediante backends seleccionables (GCC, LLVM o un generador mcode propio), lo que le confiere velocidades de simulación significativamente superiores. GHDL soporta los estándares IEEE 1076-1987, 1076-1993 y 1076-2002 de forma completa, con soporte parcial para IEEE 1076-2008 y 1076-2019. Adicionalmente, puede actuar como frontend de VHDL para el sintetizador Yosys mediante el plugin ghdl-yosys-plugin, habilitando un flujo de diseño completamente abierto desde RTL hasta bitstream. Sin embargo, GHDL opera exclusivamente por línea de comandos y no incluye visor de formas de onda, por lo que requiere la integración con una herramienta externa para la visualización de señales [14].

GTKWave es un visor de formas de onda de código abierto basado en GTK+, desarrollado por Anthony Bybell bajo licencia GPL [7]. Soporta múltiples formatos de archivo de volcado de simulación, incluyendo VCD, FST, LXT, LXT2, VZT y GHW (formato nativo de GHDL). GTKWave es la herramienta estándar de visualización en el ecosistema EDA de código abierto y es compatible con la salida de GHDL, Icarus Verilog y Verilator. Sus limitaciones principales son una interfaz gráfica que puede resultar poco intuitiva para principiantes y la ausencia de capacidad de simulación propia, lo que obliga al usuario a gestionar un flujo de trabajo fragmentado entre el simulador y el visor.

El flujo de trabajo combinado **GHDL + GTKWave** representa la alternativa de código abierto más completa para simulación de VHDL, pero exige que el estudiante instale, configure y opere dos herramientas independientes con interfaces y paradigmas distintos, generando una curva de aprendizaje que desvía la atención del objetivo pedagógico principal: comprender el comportamiento del hardware descrito en HDL.

Icarus Verilog (iverilog) es un compilador y simulador de código abierto para Verilog, creado por Stephen Williams bajo licencia GPL [15]. Compila código Verilog en un formato intermedio que es ejecutado por un motor de simulación (vvp), generando volcados de señales en formato VCD o FST. Icarus Verilog soporta los estándares IEEE 1364-1995, 1364-2001 y 1364-2005, con soporte parcial para SystemVerilog. Es ampliamente utilizado como backend de simulación en plataformas educativas como EDA Playground y HDLBits. Al igual que GHDL, opera exclusivamente por línea de comandos y depende de GTKWave para la visualización de señales.

Verilator es un simulador de Verilog/SystemVerilog de alto rendimiento que compila HDL sintetizable en modelos optimizados de C++ o SystemC, desarrollado por Wilson Snyder bajo licencia LGPLv3 [16]. Reporta velocidades hasta 100 veces superiores a las de simuladores interpretados como Icarus Verilog en un solo hilo, con aceleraciones adicionales de 2x a 10x mediante multithreading. Sin embargo, Verilator presenta limitaciones importantes para uso educativo: opera únicamente con simulación de dos estados (sin soporte para valores desconocidos “x” ni alta impedancia “z”), está restringido a diseños sintetizables (sin soporte para retardos `#delay` ni bloques `initial` con temporización), y requiere conocimientos de C++ para la escritura de testbenches.

Yosys (Yosys Open SYnthesis Suite) es un framework de síntesis RTL de código abierto desarrollado por Claire Xenia Wolf y mantenido por YosysHQ bajo licencia ISC [17]. Yosys procesa Verilog-2005 sintetizable y genera netlists para FPGAs de familias como Lattice iCE40, ECP5, Xilinx 7-Series y Gowin. Aunque no es un simulador en sí mismo, su backend CXXRTL permite simulación post-síntesis y, combinado con nextpnr, habilita un

flujo completo RTL-to-bitstream de código abierto. Yosys es principalmente una herramienta de síntesis y no aborda las necesidades de simulación comportamental ni visualización interactiva que requiere un entorno educativo integrado.

1.4.3. Herramientas Comerciales

ModelSim / QuestaSim (Siemens EDA) es el simulador HDL comercial de referencia en la industria, basado en tecnología Single Kernel Simulator (SKS) que permite simulación mixta transparente de VHDL, Verilog y SystemVerilog [8]. Ofrece depuración avanzada, cobertura de código (sentencias, ramas, condiciones, toggle, FSM) y un visor de formas de onda profesional con búsqueda de señales, marcadores y comparación. QuestaSim, su versión superior, añade soporte completo para UVM, aserciones SystemVerilog y mayor rendimiento. Sin embargo, la licencia completa de ModelSim/QuestaSim es comercial y su costo resulta prohibitivo para muchas instituciones educativas en Latinoamérica. Existen versiones gratuitas limitadas: la Questa-Intel FPGA Starter Edition impone un límite de 10,000 líneas de código y está restringida a bibliotecas de FPGAs Intel [8]. Cabe señalar que Siemens EDA ha anunciado la discontinuación de nuevas ventas de ModelSim a partir de 2024, migrando a la línea Questa One Sim.

Intel Quartus Prime es el entorno de diseño integral para FPGAs y CPLDs de Intel (anteriormente Altera), que proporciona un flujo completo desde la entrada RTL hasta la programación del dispositivo [4]. Su edición Lite es gratuita y no requiere archivo de licencia, pero está limitada a un subconjunto de familias de FPGAs Intel. Incluye la Questa-Intel FPGA Starter Edition para simulación, sujeta a las mismas restricciones de 10,000 líneas. Quartus Prime es ampliamente utilizado en laboratorios universitarios, incluyendo instituciones mexicanas, especialmente con tarjetas de la familia Cyclone. Sus principales limitaciones para uso educativo son su gran tamaño de instalación (decenas de GB), su restricción a Windows y Linux (sin soporte para macOS), y su vinculación exclusiva al ecosistema Intel.

AMD Vivado Design Suite (anteriormente Xilinx Vivado) es el entorno de diseño para FPGAs y SoCs adaptativos de AMD, que incluye XSIM como simulador HDL integrado [5]. La edición ML Standard (anteriormente WebPACK) es gratuita y soporta las familias FPGA más populares en educación, incluyendo Artix-7 (utilizada en la Basys 3), Spartan-7 y Zynq-7000. XSIM permite simulación comportamental, post-síntesis y post-implementación con soporte para VHDL, Verilog y SystemVerilog. Las limitaciones de Vivado incluyen un tamaño de instalación extremo (40–80+ GB según las familias seleccionadas), requisitos elevados de RAM y almacenamiento, soporte limitado a Windows y Linux, y una curva de aprendizaje pronunciada para principiantes. Además, no soporta familias FPGA legadas (Spartan-3, Virtex-5), para las cuales se requiere el discontinuado ISE Design Suite (v14.7, 2013).

1.4.4. Herramientas Web y Académicas

EDA Playground es un entorno de desarrollo integrado basado en web, mantenido por Doulos Ltd., que permite editar, simular y sintetizar código HDL directamente desde un navegador sin instalación alguna [18]. Ofrece múltiples backends de simulación, incluyendo Icarus Verilog y Aldec Riviera Pro (gratuitos) y Synopsys VCS y Cadence Incisive (con validación institucional). Integra EPWave, un visor de formas de onda en navegador, y permite compartir código mediante URLs. Sus limitaciones incluyen la dependencia de conexión a internet, un límite de 1,000,000 caracteres por proyecto, la imposibilidad de simular diseños grandes o multi-archivo de forma efectiva, y la ausencia de persistencia garantizada de los proyectos.

HDLBits es una plataforma web de ejercicios progresivos para la práctica de Verilog HDL, creada por Henry Wong [19]. Funciona como un "juez en línea" para diseño de hardware: el estudiante escribe código Verilog, el sistema lo simula con Icarus Verilog y compara automáticamente las formas de onda resultantes con las esperadas, proporcionando retroalimentación inmediata. HDLBits cubre lógica combinacional, secuencial, máquinas de estados finitos y diseños jerárquicos. Sus limitaciones principales son el soporte exclusivo para Verilog (sin VHDL), la restricción a módulos pequeños y aislados (sin proyectos multi-archivo), y la ausencia de panel de administración para docentes.

DigitalJS es un simulador de circuitos digitales orientado a la enseñanza que se ejecuta completamente en el navegador, desarrollado por Marek Materzok en la Universidad de Wroclaw, Polonia [20]. DigitalJS visualiza circuitos sintetizados por Yosys, permitiendo a los estudiantes interactuar con representaciones a nivel de compuertas de su código Verilog. Ofrece simulación interactiva con visualización de formas de onda en tiempo real y soporte para subcircuitos jerárquicos. Su enfoque es complementario al de un simulador comportamental, ya que requiere la síntesis previa del código y no soporta testbenches ni construcciones no sintetizables.

Logisim Evolution es una herramienta gráfica educativa para el diseño y simulación de circuitos lógicos digitales, mantenida como fork comunitario del Logisim original de Carl Burch [21], [22]. Permite diseñar circuitos mediante una interfaz de arrastrar y soltar con una biblioteca extensa de componentes (compuertas, multiplexores, registros, RAM, ROM), simulación interactiva en tiempo real, y generación de código VHDL y Verilog a partir de esquemáticos. Logisim Evolution es ampliamente utilizado en cursos introductorios de lógica digital a nivel mundial. Sin embargo, opera exclusivamente a nivel de esquemáticos y compuertas, no es un simulador de HDL, y no es adecuado para cursos avanzados de diseño digital o verificación.

1.4.5. Trabajos Académicos y Proyectos Previos

En la revisión de Trabajos Terminales de la ESCOM, se identificaron proyectos que involucran simulación y control de lógica digital de manera indirecta [9]:

- **TT0004 (ARACNOTRÓN):** Robot experimental en forma de araña con control electrónico y remoto que implicó uso de lógica digital simulada.
- **TT0007 (Control de brazo artificial):** Prototipo de mano artificial controlada por PC con lógica digital y control interactivo de hardware.
- **TT0012 (Control de brazo pintor):** Proyecto de control preciso de actuadores que requirió validación previa de lógica digital.
- **TT0006 (Sistema de electrocardiografía):** Adquisición y análisis de señales ECG con diseño de front-end analógico y validación SPICE.
- **TT0022 (Fonocardiógrafo digital):** Captura y análisis de sonidos cardíacos con uso implícito de simulación analógica y digital.
- **TT0030 HM1 (Electroencefalógrafo digital):** Adquisición y transmisión de señales EEG que requirió simulación mixta analógica (SPICE) y digital (HDL).
- **TT0024 (Especificación de protocolos con PROMELA):** Validación formal de protocolos de red con el simulador SPIN, conceptualmente paralelo a la simulación de hardware.

Ninguno de estos trabajos aborda directamente la creación de un entorno integrado de edición, simulación y depuración de código HDL. Esta ausencia evidencia un área de oportunidad institucional que el presente proyecto busca cubrir.

1.4.6. Análisis Comparativo

La Tabla 1.1 presenta un análisis comparativo de las herramientas existentes evaluadas en las secciones anteriores, contrastando sus características con las de la solución propuesta (Kmila). Los criterios de evaluación se seleccionaron con base en las necesidades identificadas en el planteamiento del problema: accesibilidad económica, integración del flujo de trabajo, compatibilidad multiplataforma y orientación educativa.

Tabla 1.1: *Análisis comparativo de herramientas existentes vs. Kmila.*

Herramienta	**Tipo**	**HDL**	**Plataformas**	**Costo**	**Editor integr.**	**Visor ondas integr.**	**GUI**	**Enfoque educativo**
GHDL + GTKWave	Open-source	VHDL	Linux, Win, macOS	Gratuito	No	Separado (GTKWave)	Parcial	Bajo
Icarus Verilog	Open-source	Verilog	Linux, Win, macOS	Gratuito	No	No (requiere GTKWave)	No (CLI)	Bajo
Verilator	Open-source	Verilog/SV (sint.)	Linux, Win, macOS	Gratuito	No	No	No (CLI)	Bajo
Yosys	Open-source (síntesis)	Verilog	Linux, Win, macOS	Gratuito	No	No	No (CLI)	Bajo
ModelSim / Questa	Comercial	VHDL/ Verilog/SV	Win, Linux	Comercial (Starter limitada)	No	Sí	Sí	Medio
Quartus Prime Lite	Comercial	VHDL/ Verilog/SV	Win, Linux	Gratuito (limitado)	Sí	Vía Questa Starter	Sí	Medio
Vivado ML Std.	Comercial	VHDL/ Verilog/SV	Win, Linux	Gratuito (limitado)	Sí	Sí (XSIM)	Sí	Medio
EDA Playground	Web	VHDL/ Verilog/SV	Navegador	Gratuito	Sí	Sí (EPWave)	Sí	Medio
HDLBits	Web	Verilog	Navegador	Gratuito	Sí	Sí	Sí	Alto
DigitalJS	Web (síntesis)	Verilog (vía Yosys)	Navegador	Gratuito	Sí	Sí	Sí	Alto
Logisim Evolution	Académica	Genera VHDL/ Verilog	Win, Linux, macOS	Gratuito	No (esquemático)	Cronograma	Sí	Alto
Kmila	**Open-source**	**VHDL**	**Win, macOS, Linux, Android, iOS, Web**	**Gratuito**	**Sí (Monaco)**	**Sí (Chart.js)**	**Sí**	**Alto**

Del análisis comparativo se identifican las siguientes brechas que la solución propuesta busca cubrir:

1. **Fragmentación del flujo de trabajo abierto.** Las herramientas de código abierto (GHDL, Icarus Verilog, Verilator) carecen de interfaz gráfica y requieren la combinación de múltiples programas independientes para completar el ciclo edición-simulación-visualización.
2. **Restricciones de las herramientas comerciales.** ModelSim, Quartus Prime y Vivado ofrecen entornos integrados pero con limitaciones significativas en sus versiones gratuitas (líneas de código, familias de dispositivos), grandes requisitos de almacenamiento, y soporte limitado a Windows y Linux.
3. **Limitaciones de las plataformas web.** EDA Playground y HDLBits resuelven el problema de instalación pero dependen de conexión a internet, presentan restricciones de tamaño de proyecto, y no permiten trabajo offline ni persistencia local garantizada.
4. **Ausencia de herramientas académicas para HDL.** Logisim Evolution es excelente para lógica combinacional y secuencial a nivel de esquemáticos, pero no aborda la simulación de código HDL. DigitalJS visualiza síntesis pero no soporta simulación comportamental.
5. **Cobertura multiplataforma insuficiente.** Ninguna de las herramientas analizadas ofrece una aplicación nativa para las seis plataformas principales (Windows, macOS, Linux, Android, iOS y navegadores web) desde un único código base.

Kmila se posiciona como una herramienta complementaria que integra en una sola aplicación local, gratuita y multiplataforma las funcionalidades que actualmente requieren múltiples herramientas o dependen de licencias comerciales restrictivas, con un enfoque explícito en el aprendizaje de diseño digital en el contexto académico de la ESCOM.

1.5. Planteamiento del Problema

Los estudiantes de la Escuela Superior de Cómputo que cursan asignaturas relacionadas con diseño digital, tales como Arquitectura de Computadoras, Fundamentos de Diseño Digital, Diseño de Sistemas Digitales y Sistemas en Chip, enfrentan una brecha entre el conocimiento teórico adquirido en el aula y la práctica efectiva con lenguajes de descripción de hardware.

Esta brecha se origina en tres factores principales:

Barrera económica. El acceso a hardware FPGA representa un costo significativo tanto para los estudiantes como para la institución. Una tarjeta Altera Cyclone IV tiene un costo aproximado de $1,866 MXN y una Basys 3 de $3,400 MXN [1], [2]. Para un grupo de 30 estudiantes, equipar un laboratorio con tarjetas individuales superaría los $55,000 MXN, sin considerar mantenimiento ni reposición. Esto obliga a que los estudiantes trabajen en equipos numerosos con acceso limitado al hardware, reduciendo la experiencia individual de aprendizaje.

Fragmentación de herramientas. Las alternativas de software gratuitas existentes no ofrecen una experiencia integrada. Un estudiante que utilice herramientas de código abierto debe instalar y configurar por separado un editor de texto, un compilador/simulador (GHDL) y un visualizador de señales (GTKWave), cada uno con su propia curva de aprendizaje y flujo de operación [6], [7]. Las herramientas comerciales como ModelSim o Quartus Prime, si bien integran más funcionalidades, restringen sus versiones gratuitas y están vinculadas a familias de hardware específicas [4], [5], [8].

Ausencia de una herramienta institucional. En la revisión de Trabajos Terminales de la ESCOM no se encontró ningún proyecto previo que aborde directamente la creación de un entorno integrado de depuración de HDL [9]. Esto implica que la institución carece de una herramienta propia, gratuita y adaptada a sus necesidades curriculares, que pueda ser distribuida libremente entre estudiantes y docentes.

En consecuencia, se identifica la necesidad de una plataforma unificada que reduzca la dependencia exclusiva de hardware especializado (FPGA), integre las funcionalidades de edición, simulación y visualización en una sola interfaz, y sea accesible sin costo desde múltiples dispositivos, complementando la formación práctica en el laboratorio.

1.6. Público Objetivo

El presente proyecto está dirigido a:

- **Estudiantes de ESCOM** que cursan asignaturas relacionadas con diseño digital, HDL y FPGA, tales como Arquitectura de Computadoras, Fundamentos de Diseño Digital, Diseño de Sistemas Digitales y Sistemas en Chip. Kmila les permite preparar y validar sus diseños en VHDL antes de acceder al laboratorio físico, funcionando como herramienta de pre-laboratorio que optimiza el tiempo limitado con el hardware real. En casos donde el acceso a tarjetas FPGA no sea posible por restricciones económicas o de disponibilidad, la plataforma puede servir como alternativa para completar la práctica de manera individual.
- **Docentes de ESCOM** que imparten las asignaturas mencionadas y requieren una herramienta gratuita y unificada para demostrar conceptos de diseño digital en el aula, asignar prácticas preparatorias que los estudiantes puedan realizar desde cualquier dispositivo previo a las sesiones de laboratorio, y reforzar la comprensión de la lógica HDL sin depender exclusivamente de la disponibilidad de hardware especializado (tarjetas FPGA).
- **Investigadores y desarrolladores** que necesiten un entorno de prototipado rápido para verificar diseños en VHDL de forma preliminar antes de su implementación en hardware, reduciendo tiempos de iteración en proyectos académicos y de investigación.

1.7. Propuesta de Solución

La solución planteada consiste en el desarrollo de Kmila, una plataforma académica multiplataforma ejecutada como aplicación local, gratuita y de código abierto, enfocada en la edición, simulación y depuración de código VHDL. El sistema se diseña como un entorno unificado que integra en una sola interfaz las funcionalidades que actualmente requieren múltiples herramientas independientes.

La arquitectura del sistema se organiza en dos capas principales:

Capa de presentación (alcance de TT1). Comprende la interfaz gráfica unificada, desarrollada con .NET 9 utilizando MAUI para plataformas nativas (Windows, macOS, Android, iOS) y Blazor para la versión web (Linux) [23], [24]. Esta capa integra los siguientes componentes:

- *Editor de código*: basado en Monaco Editor (el mismo motor de Visual Studio Code), proporciona edición de archivos VHDL con resaltado de sintaxis, numeración de líneas y atajos de teclado.
- *Sistema de gestión de proyectos*: permite crear, organizar y administrar proyectos y archivos VHDL con persistencia local mediante SQLite.
- *Selector de dispositivos FPGA*: ofrece un catálogo de 34 modelos de FPGAs de los fabricantes AMD/Xilinx (Spartan-6, Artix-7), Intel/Altera (Cyclone IV E) y Lattice (iCE40 HX, iCE40 UltraPlus, ECP5), con visualización detallada de recursos de hardware (LUTs, flip-flops, pines I/O, bloques de memoria, DSPs).
- *Panel de configuración de simulación*: permite definir parámetros temporales y valores de puertos de entrada para la ejecución futura de simulaciones.
- *Visualizador de formas de onda*: componente gráfico basado en Chart.js, preparado para representar señales digitales con escala temporal y codificación por colores según el tipo de señal.
- *Internacionalización*: soporte para siete idiomas (español, inglés, francés, alemán, chino simplificado, japonés y árabe), incluyendo dirección de escritura de derecha a izquierda (RTL, *Right-to-Left*) para este último.

Capa de simulación (alcance de TT2). Comprenderá el motor de ejecución del sistema, organizado en tres módulos: un analizador sintáctico (Parser) para procesar código VHDL y generar un Árbol de Sintaxis Abstracta (AST), un intérprete (Interpreter) para ejecutar la lógica descrita mediante ciclos delta y gestión de señales, y un módulo de control temporal (TimeMachine) para administrar el tiempo discreto de simulación y los ciclos de reloj.

A diferencia de las herramientas existentes, Kmila no requiere la instalación de múltiples programas ni depende de línea de comandos. Su enfoque integrado permite que el usuario edite, configure y —en la segunda fase— simule y visualice resultados dentro de la misma aplicación, reduciendo la curva de aprendizaje y complementando la práctica en el laboratorio físico.

1.8. Alcances del Proyecto

1.8.1. Alcances de TT1

- Análisis de requerimientos funcionales y no funcionales del sistema.
- Diseño de la arquitectura modular del sistema utilizando el modelo C4 (contexto, contenedores, componentes, despliegue).
- Construcción de la interfaz gráfica unificada con editor de código Monaco Editor, gestión de proyectos/archivos, y panel de configuración de simulación.
- Implementación del selector de dispositivos FPGA con especificaciones de 35 modelos comerciales.
- Desarrollo del visualizador de formas de onda preparado para integración con el motor de simulación.
- Validación del funcionamiento de la interfaz en su versión web orientada a Linux mediante Blazor.
- Soporte de internacionalización en siete idiomas.
- Documentación completa de análisis y diseño: diagramas UML (clases, secuencia, casos de uso, actividades), historias de usuario, casos de uso detallados y marco teórico.

1.8.2. Alcances de TT2

- Implementación del analizador sintáctico (Parser) para código VHDL con generación de AST.
- Implementación del intérprete de ejecución (Interpreter) con soporte de ciclos delta y gestión de señales.
- Implementación del módulo de control temporal (TimeMachine) para simulación en tiempo discreto.
- Integración del motor de simulación con la interfaz gráfica construida en TT1.
- Estimación de recursos de hardware (flip-flops, LUTs, pines I/O, dominios de reloj) contra especificaciones de dispositivos FPGA.
- Pruebas unitarias, de integración y de aceptación.
- Validación académica con usuarios reales (estudiantes y docentes).
- Documentación técnica final y manual de usuario.
- Publicación del código fuente en repositorio público.

1.8.3. Limitaciones

- El sistema no realiza síntesis de hardware ni genera archivos de configuración (bitstreams) para tarjetas FPGA reales.
- El soporte de HDL se limita a VHDL en su versión inicial; Verilog se contempla como extensión futura.
- La simulación se enfoca en el nivel comportamental (behavioral) y RTL, sin simulación a nivel de compuertas con retardos físicos.
- La aplicación opera exclusivamente en entorno local, sin componentes de servidor ni almacenamiento en la nube.
- La estimación de recursos FPGA es aproximada y no sustituye las herramientas de síntesis de los fabricantes.

1.9. Justificación

1.9.1. Justificación Técnica

Las herramientas actuales para la simulación de HDL presentan una fragmentación significativa en su flujo de trabajo. Un estudiante que utilice herramientas de código abierto debe configurar por separado un compilador (GHDL), un visualizador de señales (GTKWave) y un editor de texto genérico [6], [7]. Las soluciones comerciales como ModelSim integran estas funcionalidades pero dependen de licencias restrictivas y están vinculadas a familias de hardware específicas [8]. Kmila aborda esta fragmentación al ofrecer un entorno unificado donde edición, configuración y visualización coexisten en una sola aplicación multiplataforma sin requerir dependencias externas en la versión web [23], [24].

1.9.2. Justificación Educativa

La formación en diseño digital requiere que los estudiantes no solo escriban código HDL, sino que comprendan el comportamiento temporal de las señales que describen [25]. Sin una herramienta accesible para visualizar y experimentar con estos conceptos, la enseñanza queda condicionada al tiempo de acceso al laboratorio. Kmila funciona como herramienta de pre-laboratorio: permite a los estudiantes preparar y verificar sus diseños antes de la sesión con hardware real, optimizando el tiempo presencial y fortaleciendo la comprensión individual.

1.9.3. Justificación Académica

En la revisión de Trabajos Terminales de la ESCOM, no se identificó ningún proyecto previo dedicado a la creación de un entorno integrado de depuración de HDL [9]. El presente proyecto contribuye a la línea institucional de herramientas educativas para diseño digital, generando un recurso de código abierto que puede ser adoptado y extendido por la comunidad académica de ESCOM e IPN.

1.9.4. Justificación Económica

El costo de equipar un laboratorio con tarjetas FPGA individuales para un grupo de 30 estudiantes supera los $55,000 MXN considerando únicamente el hardware [1], [2]. Kmila se distribuye de forma gratuita y no requiere hardware adicional: cualquier computadora o dispositivo móvil con navegador web puede ejecutar la aplicación, reduciendo la barrera de entrada para la práctica individual y complementando la inversión institucional en infraestructura de laboratorio.

1.10. Factibilidad

1.10.1. Factibilidad Técnica

El desarrollo de Kmila se sustenta en tecnologías maduras, ampliamente documentadas y de uso libre. La Tabla 1.2 presenta el stack tecnológico seleccionado para el proyecto.

Tabla 1.2: *Stack tecnológico del proyecto Kmila.*

Componente	Tecnología	Versión	Justificación
Framework nativo	.NET MAUI	9.0	Aplicaciones nativas para Windows, macOS, Android e iOS desde un único código base [23]
Framework web	Blazor WebAssembly	9.0	Ejecución en navegador sin servidor, cobertura de Linux [24]
Lenguaje principal	C#	13.0	Ecosistema unificado con .NET, tipado fuerte, orientado a objetos
Editor de código	Monaco Editor (BlazorMonaco)	3.3.0	Motor del editor Visual Studio Code, resaltado de sintaxis y autocompletado
Visualización de ondas	Chart.js (BlazorBootstrap)	3.4.0	Gráficas interactivas para representación de formas de onda
Persistencia local	SQLite (sqlite-net-pcl)	1.9.172	Base de datos embebida, sin servidor, multiplataforma
Motor de simulación	Motor propio en C#	—	Control total del ciclo de simulación HDL (alcance de TT2)

Todas las tecnologías listadas son de código abierto o cuentan con licencias gratuitas para uso académico. El entorno de desarrollo requiere únicamente el SDK de .NET 9 (gratuito), un editor de código como Visual Studio Code o Visual Studio Community (ambos gratuitos), y un dispositivo para pruebas. No se requiere infraestructura de servidor ni servicios en la nube.

1.10.2. Factibilidad Económica

1.10.2.1. Metodología de estimación de costos

El análisis económico del proyecto se estructura mediante tres enfoques complementarios. La Cost Breakdown Structure (CBS) permite descomponer el costo total en categorías identificables (recursos humanos, equipo, licencias y distribución), facilitando la trazabilidad de cada partida. El Total Cost of Ownership (TCO) extiende el análisis al considerar no solo el desarrollo inicial sino también los costos recurrentes asociados a la distribución y mantenimiento del software. Finalmente, se realiza un análisis costo-beneficio en contexto educativo para contrastar la inversión requerida con el valor que la solución aporta frente a las alternativas existentes.

1.10.2.2. Estimación del esfuerzo de desarrollo

El costo principal del proyecto se deriva del esfuerzo de desarrollo, estimado en horas de trabajo. El proyecto abarca 10 meses (febrero 2026 a enero 2027) con una dedicación aproximada de 3.5 horas diarias, 6 días por semana. La Tabla 1.3 desglosa la estimación por módulo y fase.

Tabla 1.3: *Estimación de horas de desarrollo por módulo.*

Fase	Módulo	Horas estimadas
TT1	Investigación, documentación y protocolo	80
TT1	Arquitectura y diseño del sistema	60
TT1	Interfaz de usuario (Monaco Editor, pestañas, explorador de archivos)	100
TT1	Gestión de proyectos y persistencia (CRUD, SQLite)	60
TT1	Catálogo FPGA (carga JSON, 35 dispositivos)	40
TT1	Internacionalización (7 idiomas, soporte RTL)	40
TT1	Configuración (temas, eventos reactivos, .k-config)	30
TT1	Visor de formas de onda (Chart.js)	30
TT1	Pruebas de integración y ajustes	20
	Subtotal TT1	**460**
TT2	Analizador léxico y sintáctico (Parser VHDL)	120
TT2	Generación de AST y modelo intermedio	80
TT2	Motor de simulación (DeltaCycleEngine)	100
TT2	Depuración paso a paso (TimeMachine)	60
TT2	Construcción de estímulos (StimulusBuilder)	40
TT2	Integración motor-UI y visualización de resultados	50
TT2	Pruebas, depuración y optimización	60
TT2	Documentación de TT2	40
	Subtotal TT2	**550**
	Total del proyecto	**1,010**

1.10.2.3. Desglose de costos del proyecto

La Tabla 1.4 presenta el desglose conforme a la CBS del proyecto. Los costos se organizan en cinco categorías: recursos humanos (valuados a una tarifa de referencia de $200 MXN/hora para un desarrollador junior en el mercado mexicano), equipo de desarrollo, equipo de pruebas, licencias de software y costos de distribución. Los costos de equipo se prorratean a 10 meses sobre una vida útil estimada de 48 meses.

Tabla 1.4: *Desglose de costos del proyecto.*

Categoría	Concepto	Costo estimado
Recursos humanos	1,010 horas de desarrollo ($\times$ $200 MXN/hr)	$202,000 MXN
Equipo de desarrollo	Laptop Lenovo IdeaPad L340 — prorrateo 10/48 meses sobre valor de $15,000 MXN	$3,125 MXN
Equipo de pruebas	Smartphone POCO F5 Pro (Android) — prorrateo 10/48 meses sobre valor de $8,500 MXN	$1,771 MXN
Licencias de software	.NET 9 SDK, Visual Studio Code, Git, Android SDK	$0 MXN
Distribución	Google Play Store — registro de desarrollador (pago único)	$500 MXN
Distribución	Apple Developer Program (suscripción anual)	$2,000 MXN
Distribución	Dominio web (registro anual)	$250 MXN
Distribución	Hosting web — Netlify (nivel gratuito)	$0 MXN
Distribución	Repositorio de código — GitHub (público, gratuito)	$0 MXN
	Costo total estimado (TCO)	**$209,646 MXN**

1.10.2.4. Costos reales de implementación

El costo predominante es el recurso humano (96.3 %), lo cual es característico de proyectos de software cuyas herramientas de desarrollo son de código abierto. En el contexto académico del Trabajo Terminal, este costo constituye una estimación teórica que permite dimensionar el valor del esfuerzo invertido, pero no representa un desembolso monetario real dado que el desarrollo se realiza como parte de la formación del estudiante. Los costos efectivos del proyecto, es decir, aquellos que requieren un desembolso real para la publicación y distribución del software, ascienden a $2,750 MXN, distribuidos en el registro de las tiendas de aplicaciones ($2,500 MXN) y el registro de dominio web ($250 MXN).

1.10.2.5. Beneficio económico de la solución

Desde la perspectiva del análisis costo-beneficio, Kmila aborda directamente el problema económico identificado en la Sección 1.5: el costo de equipar un laboratorio con tarjetas FPGA individuales para un grupo de 30 estudiantes supera los $55,000 MXN considerando únicamente el hardware [1], [2], y las herramientas comerciales como ModelSim requieren licencias cuyo costo las hace inaccesibles para uso individual [8]. Kmila se distribuye de forma gratuita, se ejecuta en cualquier computadora o dispositivo móvil sin requerir hardware especializado (FPGA), y su código fuente permanece disponible en un repositorio público. Esto permite que cada estudiante acceda a un entorno de simulación completo sin costo alguno, reduciendo la dependencia del hardware especializado del laboratorio.

1.10.2.6. Conclusión de viabilidad económica

El proyecto es económicamente viable debido a que los costos operativos son prácticamente nulos (todas las tecnologías de desarrollo son de código abierto), el desembolso real para distribución es de $2,750 MXN, y la solución elimina las barreras económicas que actualmente limitan el acceso de los estudiantes a herramientas de simulación de circuitos digitales.

1.10.3. Factibilidad Operativa

El proyecto es desarrollado por un alumno (Ulrich Tamayo Daniel) bajo la supervisión de dos directores: M. en C. Pastrana Fernández Carlos Jesús y Dr. Cifuentes Álvarez Alejandro Sigrido. El cronograma se divide en dos fases: TT1 (febrero–junio 2026) y TT2 (julio 2026–enero 2027), conforme al calendario académico de la ESCOM y al protocolo registrado TT 2026-B059.

El desarrollo se realiza de forma local en el equipo de cómputo personal del alumno, sin requerir permisos institucionales especiales, acceso a servidores, ni infraestructura adicional. Las pruebas de la versión web se ejecutan en un navegador estándar sobre Linux, y las pruebas de la versión nativa se realizan en el mismo equipo de desarrollo.

No se identifican dependencias externas que pongan en riesgo la viabilidad operativa del proyecto. El código fuente se gestiona mediante Git con respaldo en repositorio remoto, lo que garantiza la trazabilidad y recuperación ante pérdida de datos.

Capítulo 2

Marco Teórico

2.1. Lenguajes de Descripción de Hardware (HDL)

2.1.1. Historia y Evolución

Los lenguajes de descripción de hardware (HDL, por sus siglas en inglés) surgen como respuesta a la creciente complejidad del diseño de circuitos integrados. Durante las décadas de 1960 y 1970, los primeros esfuerzos formales incluyeron lenguajes como ISP (Instruction Set Processor), propuesto por Bell y Newell en 1971, que introdujo el concepto de nivel de transferencia de registros (RTL) para describir el comportamiento de procesadores [26]. Sin embargo, el diseño de hardware continuó dominado por la captura esquemática manual hasta mediados de la década de 1980.

El punto de inflexión se produjo con dos desarrollos paralelos. Por un lado, el Departamento de Defensa de los Estados Unidos, a través de su programa VHSIC (Very High Speed Integrated Circuit), financió en 1983 la creación de VHDL como un lenguaje estándar para documentar y simular diseños de circuitos integrados [10], [11]. Por otro lado, en 1984, Phil Moorby y Prabhu Goel crearon Verilog en Gateway Design Automation como una herramienta propietaria de simulación con sintaxis inspirada en el lenguaje C [12], [13].

La introducción de la síntesis lógica a finales de la década de 1980, que permitió compilar código HDL en netlists a nivel de compuertas, transformó radicalmente la industria. Antes de 1990, la mayoría de los diseños se iniciaban mediante captura esquemática; para mediados de los años noventa, aproximadamente la mitad de los nuevos diseños utilizaban HDLs [26]. En la actualidad, prácticamente todos los diseños digitales de complejidad media o alta emplean métodos basados en HDL.

La evolución continuó con SystemVerilog (IEEE Std 1800), publicado inicialmente en 2005 y fusionado con el estándar Verilog en 2009, que incorporó capacidades de verificación orientada a objetos, restricciones aleatorias y aserciones [27]. En paralelo, VHDL mantuvo su evolución con revisiones en 2008 y 2019 que integraron mejoras como genéricos en paquetes y soporte completo de PSL (Property Specification Language) [10].

En cuanto a la adopción regional, estudios de la industria indican que VHDL predomina en Europa, sectores aeroespaciales y de defensa, y en la academia latinoamericana, mientras que Verilog y SystemVerilog dominan en la industria de semiconductores de Estados Unidos y Asia [28]. En el contexto del diseño con FPGAs, VHDL se mantiene como el lenguaje más utilizado a nivel global según encuestas recientes del Wilson Research Group [28].

2.1.2. VHDL

VHDL (VHSIC Hardware Description Language) fue desarrollado entre 1983 y 1985 bajo contrato del Departamento de Defensa de los Estados Unidos (contrato F33615-83-C-1003), adjudicado a un equipo conformado por Intermetrics, Inc. como contratista principal, Texas Instruments como experto en diseño de circuitos integrados, e IBM como especialista en diseño de sistemas computacionales [11]. El lenguaje fue fuertemente influenciado por Ada, del cual heredó su sintaxis verbosa, su sistema de tipos estricto y sus mecanismos de modularidad.

La primera versión pública (VHDL 7.2) se liberó en 1985, y en 1987 fue aprobado como estándar IEEE bajo la designación IEEE Std 1076-1987. Desde entonces, el estándar ha sido revisado en múltiples ocasiones, como se muestra en la Tabla 2.1.

Tabla 2.1: *Revisiones del estándar IEEE 1076 (VHDL).*

Estándar	Año	Cambios principales
IEEE 1076-1987	1987	Primera estandarización; definición base del lenguaje
IEEE 1076-1993	1993	Revisión mayor: caracteres ISO-8859-1, operador xnor, sintaxis más consistente
IEEE 1076-2000	2000	Tipos protegidos para control de concurrencia en variables compartidas
IEEE 1076-2008	2008	Revisión mayor: subconjunto PSL, genéricos en paquetes y subprogramas; absorbe IEEE 1164 y 1076.3
IEEE 1076-2019	2019	Revisión vigente: integración completa de PSL, mejoras diversas

Las características fundamentales de VHDL incluyen:

- **Tipado fuerte:** el compilador verifica estrictamente la compatibilidad de tipos, previniendo errores comunes en tiempo de compilación.
- **Modelo de ejecución concurrente:** las sentencias dentro de una arquitectura se ejecutan en paralelo, reflejando el comportamiento real del hardware.
- **Paquetes y bibliotecas:** permiten encapsular declaraciones reutilizables (tipos, funciones, constantes) para su uso en múltiples diseños.
- **Genéricos:** habilitan diseños parametrizados (por ejemplo, el ancho de un bus como parámetro configurable).
- **Configuraciones:** permiten vincular entidades con arquitecturas y componentes con implementaciones sin modificar el código fuente.

Los elementos estructurales básicos de VHDL son la *entity* (interfaz externa del módulo), la *architecture* (descripción interna del comportamiento o estructura), el *process* (bloque de ejecución secuencial activado por una lista de sensibilidad), la *signal* (representación de conexiones físicas entre componentes) y el *component* (mecanismo de instanciación jerárquica) [11], [29].

La preferencia por VHDL en la academia europea y latinoamericana se explica por factores pedagógicos e históricos: su tipado fuerte impone disciplina en el modelado de hardware, su sintaxis explícita favorece la comprensión en entornos de aprendizaje, y la tradición académica europea, donde VHDL fue adoptado tempranamente, se extendió a las instituciones latinoamericanas que siguen modelos curriculares similares [25].

2.1.3. Verilog

Verilog fue creado entre finales de 1983 e inicios de 1984 por Phil Moorby, Prabhu Goel y Chi-Lai Huang en Automated Integrated Design Systems, empresa posteriormente renombrada como Gateway Design Automation en 1985 [13]. Moorby es reconocido como el diseñador principal del lenguaje y también desarrolló Verilog-XL, el primer simulador de referencia industrial para Verilog.

Inicialmente un lenguaje propietario, Verilog dio un giro decisivo cuando Cadence Design Systems adquirió Gateway Design Automation en noviembre de 1989. En 1990, Cadence liberó la especificación del lenguaje al dominio público, lo que condujo a la formación de Open Verilog International (OVI) y, eventualmente, a su estandarización por IEEE como IEEE Std 1364-1995 [12]. La Tabla 2.2 resume las revisiones del estándar.

Tabla 2.2: *Revisiones del estándar IEEE 1364 (Verilog) e IEEE 1800 (SystemVerilog).*

Estándar	Año	Descripción
IEEE 1364-1995	1995	Primera estandarización (Verilog-95)
IEEE 1364-2001	2001	Revisión mayor: sentencias generate, declaración de puertos mejorada, aritmética con signo
IEEE 1364-2005	2005	Último estándar independiente de Verilog
IEEE 1800-2005	2005	Primer estándar de SystemVerilog
IEEE 1800-2009	2009	Fusión de Verilog (1364) con SystemVerilog; fin del estándar Verilog independiente
IEEE 1800-2017	2017	Correcciones y clarificaciones

Las características principales de Verilog incluyen:

- **Sintaxis similar a C:** resulta familiar para ingenieros de software, con una notación concisa y menos verbosa que VHDL.
- **Tipado débil:** un sistema de tipos más permisivo con conversiones implícitas, lo que acelera la escritura pero puede favorecer errores sutiles.
- **Lógica de cuatro valores:** los valores 0, 1, x (desconocido) y z (alta impedancia) están integrados en el lenguaje base.
- **Modelado comportamental y estructural:** soporta tanto bloques `always` (descripción de comportamiento) como instanciación de compuertas (descripción estructural).
- **Directivas de preprocesador:** similares a las de C (`` `define ``, `` `include ``, `` `ifdef ``).

SystemVerilog, su sucesor, se originó a partir del lenguaje Superlog desarrollado por Co-Design Automation en 1997 y de las capacidades de verificación de OpenVera de Synopsys [27]. Definido como un "lenguaje unificado de diseño, especificación y verificación de hardware," SystemVerilog añade programación orientada a objetos, verificación con restricciones aleatorias, aserciones (SVA) y cobertura funcional, convirtiéndose en el estándar dominante para verificación de ASICs modernos.

El dominio de Verilog y SystemVerilog en la industria estadounidense y asiática se atribuye a sus raíces en Silicon Valley (donde Gateway Design Automation operaba), a la posición de mercado de Cadence con Verilog-XL, y a la adopción por parte de las fundiciones y casas de diseño asiáticas que operan como proveedores de la industria

fabless norteamericana [13], [28].

2.1.4. Comparativa VHDL vs. Verilog

La Tabla 2.3 presenta una comparativa de las características principales de ambos lenguajes.

Tabla 2.3: *Comparativa entre VHDL y Verilog.*

Característica	**VHDL**	**Verilog**
Origen	DoD (EE.UU.), 1983	Gateway Design Automation, 1984
Estándar IEEE	1076 (1987–2019)	1364 (1995–2005), fusionado en 1800
Influencia sintáctica	Ada	C
Tipado	Fuertemente tipado	Débilmente tipado
Verbosidad	Alta	Baja
Modelo de ejecución	Concurrente	Concurrente
Lógica multivaluada	Vía paquete IEEE 1164 (`std_logic`: 9 valores)	Nativo (4 valores: 0, 1, x, z)
Verificación avanzada	Limitada (PSL externo)	SystemVerilog (UVM, SVA, cobertura)
Predominio académico	Europa, Latinoamérica	EE.UU., Asia
Predominio industrial	Aeroespacial, defensa, FPGA	Semiconductores, ASIC

Ambos lenguajes son funcionalmente equivalentes para la descripción de hardware digital y comparten el paradigma de ejecución concurrente. La elección entre uno u otro depende principalmente del contexto institucional, regional y del dominio de aplicación.

2.1.5. Otros Lenguajes de Descripción de Hardware

Aunque VHDL y Verilog (junto con su extensión SystemVerilog) concentran la práctica totalidad del diseño profesional de hardware digital, en las últimas dos décadas han surgido diversos lenguajes de más alto nivel orientados a mitigar la verbosidad y la rigidez sintáctica de los HDL clásicos. A continuación se describen los más relevantes en la literatura académica e industrial contemporánea.

SystemC no es estrictamente un HDL, sino una biblioteca de C++ con soporte para modelado concurrente de hardware a nivel de sistema (ESL, *Electronic System Level*). Fue introducido por la Open SystemC Initiative en 1999 y estandarizado como IEEE 1666 (última revisión en 2023) [30]. Su caso de uso principal es el modelado arquitectónico de SoCs y aceleradores en fases tempranas de diseño, donde la velocidad de simulación y la interoperabilidad con código C++ son prioritarias. SystemC no es sintetizable de forma directa: para obtener un *netlist* es necesario traducir los modelos a RTL en VHDL o Verilog mediante herramientas de síntesis de alto nivel (HLS).

MyHDL, creado por Jan Decaluwe en 2003, es una biblioteca de Python que permite describir hardware digital empleando la sintaxis del lenguaje. MyHDL incluye un simulador propio y la capacidad de convertir los modelos a Verilog y VHDL para su síntesis posterior [31]. Su adopción se ha mantenido circunscrita a proyectos académicos y a comunidades de hardware abierto.

Amaranth HDL (originalmente nMigen), surgido en 2019 en el ecosistema del proyecto SymbiFlow y mantenido actualmente como proyecto independiente, es una evolución conceptual de MyHDL construida sobre Python moderno. A diferencia de MyHDL, Amaranth integra de forma nativa el manejo de dominios de reloj, transiciones síncronas explícitas y un subsistema de verificación formal [32]. Amaranth genera Verilog como formato de salida para síntesis con herramientas como Yosys [17], consolidándose como el lenguaje preferido en la comunidad de hardware libre y en proyectos abiertos basados en FPGAs Lattice.

Chisel (*Constructing Hardware in a Scala Embedded Language*), desarrollado por el equipo de UC Berkeley liderado por Krste Asanović desde 2012, es un lenguaje de dominio específico (DSL) embebido en Scala diseñado para construir generadores de hardware parametrizables [33]. Chisel se ha consolidado como el lenguaje de referencia en el ecosistema RISC-V y es la base del procesador Rocket Chip. Genera Verilog como salida mediante su representación intermedia FIRRTL.

SpinalHDL, de Charles Papon, es otro DSL embebido en Scala aparecido en 2015, con enfoque en la legibilidad sintáctica y la facilidad de integración con proyectos Java [34]. Compila tanto a Verilog como a VHDL, y privilegia una descripción estructural explícita del circuito.

Bluespec SystemVerilog (BSV), desarrollado por Arvind en el MIT y comercializado desde 2000, adopta un paradigma de *reglas guardadas* (*guarded atomic actions*) que describe el comportamiento del hardware en términos de transiciones atómicas con condiciones de habilitación [35]. Esta semántica se traduce internamente a Verilog sintetizable. BSV ha tenido mayor penetración en el ámbito académico y en diseños de aceleradores de alto rendimiento que en la industria de propósito general.

Es importante destacar un patrón común en todas estas alternativas: ninguna reemplaza a VHDL ni a Verilog en el flujo de síntesis. Todas ellas generan, como formato final, código RTL en VHDL o Verilog que posteriormen-

te es procesado por las herramientas de síntesis convencionales de los fabricantes de FPGA. En consecuencia, el conocimiento de VHDL o Verilog permanece como requisito fundamental para el diseño digital profesional, independientemente de la capa de abstracción adicional que se emplee durante el desarrollo.

2.1.6. Justificación de la Elección de VHDL para Kmila

La selección de VHDL como lenguaje primario soportado por Kmila, con Verilog contemplado como extensión prevista para TT2, se fundamenta en cinco consideraciones alineadas con el propósito pedagógico del proyecto:

Cobertura curricular. Los planes de estudio de Ingeniería en Sistemas Computacionales de la ESCOM incorporan VHDL como lenguaje primario en las asignaturas de Fundamentos de Diseño Digital, Arquitectura de Computadoras, Diseño de Sistemas Digitales y Sistemas en Chip. El mismo patrón se observa en la mayor parte de las instituciones latinoamericanas y europeas, como se documentó en la Sección 2.1.1. Kmila está orientado explícitamente al estudiante en formación inicial dentro de este contexto institucional.

Dominancia industrial y vigencia. De acuerdo con las encuestas periódicas del Wilson Research Group [28], VHDL se mantiene como el lenguaje de mayor adopción en diseño con FPGA a nivel global, mientras que Verilog y SystemVerilog dominan el segmento ASIC. Ambos lenguajes cuentan con estándares IEEE vigentes (1076-2019 para VHDL; 1800-2017 para SystemVerilog) y respaldo institucional de los tres fabricantes de FPGA tratados en la Sección 2.3.

Dependencia de los lenguajes de alto nivel. Como se estableció en la Sección 2.1.5, los lenguajes alternativos (SystemC, MyHDL, Amaranth, Chisel, SpinalHDL, Bluespec) no reemplazan a VHDL ni a Verilog: su flujo de síntesis desemboca invariablemente en código RTL en uno de estos dos lenguajes. Enseñar únicamente un lenguaje de alto nivel dejaría al estudiante sin las bases necesarias para depurar la salida sintetizada, interpretar un *netlist* o integrar módulos en proyectos heterogéneos donde coexisten componentes escritos en VHDL o Verilog por terceros.

Madurez del ecosistema. VHDL y Verilog disponen de un ecosistema consolidado de simuladores (GHDL, Icarus Verilog, ModelSim, Verilator), visores de formas de onda (GTKWave) y herramientas de análisis estático. Este ecosistema facilita la implementación del simulador de Kmila, ya que permite adoptar formatos estándar (por ejemplo, VCD para el visor de formas de onda) y validar los resultados del simulador propio frente a herramientas de referencia ampliamente utilizadas [6], [7].

Nivel de abstracción apropiado al propósito pedagógico. El objetivo de Kmila, como se detalla en la Sección 2.4, es reducir la barrera de entrada al aprendizaje del diseño digital. El nivel de abstracción adecuado para un primer contacto con HDL es el nivel comportamental y RTL, donde el estudiante puede observar directamente la correspondencia entre las estructuras sintácticas del lenguaje y los elementos físicos del circuito (señales, registros, procesos). Los lenguajes de más alto nivel (Chisel, Amaranth, SystemC) introducen capas de abstracción que ocultan esta correspondencia y resultan más apropiados para etapas posteriores de formación.

En consecuencia, la versión actual de Kmila implementa soporte completo para VHDL como lenguaje primario, contempla Verilog como extensión prevista para TT2 y descarta explícitamente del alcance del proyecto el soporte nativo para lenguajes de alto nivel, sin perjuicio de que los archivos generados por dichos lenguajes (Verilog o

VHDL sintetizable) puedan analizarse en Kmila una vez integrado el soporte para Verilog.

2.2. Simulación Digital

Esta sección presenta los fundamentos teóricos de la simulación de circuitos digitales descritos en HDL, abarcando el modelo de eventos discretos, la semántica de señales y procesos, la temporización basada en ciclos de reloj y los niveles de abstracción empleados en el flujo de diseño.

2.2.1. Modelo de Eventos Discretos

La simulación de circuitos digitales descritos en HDL se fundamenta en el modelo de simulación dirigida por eventos discretos (DESM, Discrete Event Simulation Model), formalizado en el estándar IEEE 1076 para VHDL y en IEEE 1364/1800 para Verilog y SystemVerilog [10], [12]. A diferencia de la simulación analógica, donde las magnitudes varían de forma continua en el tiempo, la simulación digital opera sobre un conjunto finito de valores lógicos que cambian en instantes discretos.

El principio fundamental del modelo es que el simulador mantiene una cola de eventos ordenados cronológicamente. Un evento se define como un cambio de valor en una señal en un instante de tiempo determinado. En cada paso de simulación, el simulador extrae los eventos correspondientes al tiempo actual, actualiza las señales afectadas y evalúa los procesos sensibles a dichas señales. Si la evaluación produce nuevos cambios, estos se insertan en la cola como eventos futuros [29].

Un concepto central en la simulación VHDL es el **ciclo delta**. Cuando una asignación de señal se ejecuta sin un retardo explícito (es decir, con retardo cero), el nuevo valor no se aplica inmediatamente sino que queda programado como pendiente. Solo al finalizar el ciclo delta actual se aplican todos los valores pendientes de forma simultánea. Este paso infinitesimal de tiempo preserva el orden causal sin avanzar el tiempo de simulación. Dentro de un mismo instante de tiempo pueden ocurrir múltiples ciclos delta sucesivos hasta que el sistema alcanza un estado estable, es decir, hasta que ningún proceso genera nuevos eventos [11], [29].

Este mecanismo es la clave del determinismo en la simulación concurrente. Dado que los nuevos valores de las señales no se hacen visibles hasta el final del ciclo delta, todos los procesos que se ejecutan dentro de un mismo ciclo leen los mismos valores (los del ciclo anterior), independientemente del orden en que el simulador los evalúe internamente. Así, el resultado de la simulación es siempre el mismo sin importar el orden de ejecución de los procesos, análogamente a lo que ocurre en hardware real, donde todas las compuertas operan simultáneamente sobre las señales presentes en sus entradas.

Para ilustrar el mecanismo, considérese un diseño con tres procesos (A, B y C), donde A modifica una señal presente en la lista de sensibilidad de B, y B modifica una señal sensible de C:

1. **Ciclo delta 1:** los tres procesos se activan y ejecutan su lógica interna, leyendo los valores del estado inicial. A genera una asignación pendiente sobre una señal que B observa; B genera una asignación sobre una señal que C observa. Al finalizar el ciclo, se aplican todos los valores pendientes.
2. **Ciclo delta 2:** solo B y C se reactivan, porque las señales en sus listas de sensibilidad cambiaron. A permanece suspendido, ya que ninguna de sus señales sensibles fue modificada. B y C ejecutan y generan nuevas asignaciones. Se aplican los cambios.

3. **Ciclos delta sucesivos:** en cada ciclo, únicamente los procesos cuyas señales sensibles hayan cambiado se reactivan. El resto permanece suspendido.
4. **Estado estable:** cuando ningún proceso genera nuevos eventos al finalizar un ciclo delta, el sistema ha alcanzado un estado estable para el instante de tiempo actual. Solo entonces el simulador avanza al siguiente instante de tiempo con eventos programados (por ejemplo, el siguiente flanco de reloj).

El flujo de ejecución de cada ciclo delta sigue tres fases:

1. **Actualización de señales:** se aplican los valores programados durante el ciclo delta anterior (o los valores iniciales, en el primer ciclo).
2. **Evaluación de procesos:** se ejecutan todos los procesos cuyas señales de la lista de sensibilidad hayan cambiado de valor. Los procesos cuyas señales no fueron modificadas permanecen suspendidos.
3. **Programación de eventos:** las asignaciones realizadas por los procesos generan nuevos eventos, ya sea con retardo explícito (avanzan el tiempo de simulación) o con retardo delta (permanecen en el mismo instante).

Este ciclo de tres fases se repite hasta que no quedan eventos pendientes en el tiempo actual, momento en que el simulador avanza al siguiente instante con eventos programados [10].

2.2.2. Señales y Procesos

En el modelo de ejecución HDL existen dos paradigmas fundamentales de asignación: la asignación concurrente y la asignación secuencial. Comprender su diferencia es esencial para interpretar correctamente los resultados de simulación.

Las **señales** (*signals* en VHDL, *wires* y *regs* en Verilog) representan conexiones físicas entre componentes del circuito. En VHDL, las señales se actualizan al finalizar el ciclo delta actual, no en el momento de la asignación. Esto implica que múltiples asignaciones a la misma señal dentro de un mismo proceso resultan en que solo la última asignación tenga efecto, ya que las anteriores son sobrescritas antes de aplicarse [11].

Las **variables** (*variables* en VHDL) son elementos de almacenamiento local dentro de un proceso que se actualizan inmediatamente al ejecutar la asignación. A diferencia de las señales, las variables no tienen representación física directa en el hardware sintetizado y su uso principal es como almacenamiento temporal para cálculos intermedios dentro de un proceso [29].

Un **proceso** (*process* en VHDL, bloque *always* en Verilog) es la unidad fundamental de ejecución secuencial. Cada proceso se activa cuando alguna señal de su lista de sensibilidad cambia de valor. Una vez activado, el cuerpo del proceso se ejecuta secuencialmente de principio a fin, y al terminar, el proceso se suspende hasta la siguiente activación. Aunque la ejecución interna de un proceso es secuencial, todos los procesos de un diseño se ejecutan de forma concurrente entre sí, reflejando el paralelismo inherente del hardware [10].

Las **sentencias concurrentes** en VHDL (asignaciones de señal fuera de un proceso, instanciaciones de componentes, sentencias *generate*) son equivalentes a procesos implícitos con listas de sensibilidad derivadas automáticamente de las señales leídas en la expresión. Por ejemplo, la asignación concurrente `y <= a and b;` equivale a un proceso sensible a las señales `a` y `b` [11].

Esta distinción entre concurrencia y secuencialidad tiene implicaciones directas en la simulación: el simulador debe garantizar que el resultado sea independiente del orden en que se evalúan los procesos concurrentes, lo cual se logra mediante el mecanismo de ciclos delta descrito anteriormente.

2.2.3. Ciclos de Reloj y Temporización

En la simulación de circuitos digitales síncronos, la señal de reloj es el elemento temporal fundamental que gobierna la evolución del sistema. Un circuito síncrono organiza sus operaciones en torno a los flancos (transiciones) de una señal de reloj periódica, de modo que los elementos de memoria (flip-flops, registros) capturan nuevos valores únicamente en el instante del flanco activo [29].

La **resolución temporal** del simulador define la granularidad mínima con la que se representan los instantes de tiempo. En VHDL, la resolución base es el femtosegundo (fs, 10^{-15} s), aunque en la práctica las simulaciones emplean unidades mayores como nanosegundos (ns) o picosegundos (ps). En Verilog, la resolución se establece mediante la directiva `` `timescale ``, que especifica la unidad de tiempo y la precisión (por ejemplo, `` `timescale 1ns/1ps ``) [12].

El modelado de retardos en simulación se clasifica en tres categorías:

- **Retardo inercial** (*inertial delay*): modelo por defecto en VHDL, donde un pulso más corto que el retardo especificado es filtrado (absorbido) por el elemento. Este comportamiento modela la inercia eléctrica real de las compuertas, que requieren un tiempo mínimo para conmutar.
- **Retardo de transporte** (*transport delay*): modela una línea de transmisión ideal que propaga cualquier pulso, independientemente de su duración, con un retardo fijo. En VHDL se indica con la palabra reservada `transport`.
- **Retardo delta** (*delta delay*): retardo infinitesimal sin avance del tiempo de simulación, utilizado para resolver dependencias causales entre asignaciones concurrentes, como se describió en la Sección 2.2.1.

En un testbench típico, la señal de reloj se genera mediante un proceso que alterna el valor de la señal con retardos explícitos. La frecuencia del reloj define el periodo, y la relación entre el tiempo en alto y el periodo total determina el ciclo de trabajo (*duty cycle*). Los estímulos de entrada se sincronizan típicamente con los flancos de reloj mediante sentencias `wait until` (VHDL) o `@(posedge clk)` (Verilog), garantizando que el circuito bajo prueba reciba datos estables respecto a los tiempos de setup y hold de los elementos secuenciales [11].

2.2.4. Niveles de Abstracción

El diseño y la simulación de hardware digital operan en múltiples niveles de abstracción, cada uno con un grado diferente de detalle y un propósito distinto en el flujo de desarrollo. Estos niveles, de mayor a menor abstracción, son:

Nivel comportamental (*behavioral*). Describe la funcionalidad del circuito en términos algorítmicos, sin referencia a la estructura de hardware subyacente. En VHDL, este nivel se expresa mediante procesos con sentencias secuenciales (if-then-else, case, loops) y asignaciones de señal con retardos explícitos. La simulación comportamental verifica que el algoritmo produce los resultados correctos, sin considerar las restricciones de temporización del dispositivo físico. Es el nivel adecuado para la fase inicial de diseño y para la especificación de testbenches [29].

Nivel de transferencia de registros (*Register Transfer Level*, RTL). Describe el circuito en términos de registros (elementos de almacenamiento) y la lógica combinacional que transforma datos entre ellos en cada ciclo de reloj. Este es el nivel de abstracción principal para la síntesis: las herramientas de síntesis RTL traducen la descripción en HDL a una netlist de compuertas lógicas. En VHDL, el nivel RTL se caracteriza por el uso de procesos sensibles a flancos de reloj para lógica secuencial y asignaciones concurrentes o procesos combinacionales para la lógica entre registros. La simulación RTL verifica la funcionalidad ciclo a ciclo del diseño sintetizable [26].

Nivel de compuertas (*gate level*). Describe el circuito como una interconexión de compuertas lógicas primitivas (AND, OR, NOT, flip-flops) provenientes de una biblioteca tecnológica específica. Este nivel resulta de la síntesis del código RTL y se utiliza para simulación post-síntesis, donde se incorporan los retardos reales de las compuertas y las interconexiones. La simulación a nivel de compuertas es significativamente más lenta que la simulación RTL o comportamental, pero permite verificar que el circuito sintetizado cumple con las restricciones temporales del dispositivo objetivo [29].

Para el presente proyecto, Kmila implementa simulación a nivel comportamental, que es el nivel de abstracción más relevante para el aprendizaje de HDL en entornos académicos. Este nivel permite al estudiante concentrarse en la lógica y el comportamiento del diseño sin la complejidad adicional de las restricciones tecnológicas de un dispositivo específico. La extensión a simulación RTL y post-síntesis se contempla como trabajo futuro.

2.3. Fabricantes y Dispositivos FPGA Soportados

La industria de dispositivos lógicos programables (FPGA) está dominada globalmente por tres fabricantes principales: AMD (antes Xilinx), Intel (a través de Altera) y Lattice Semiconductor. En conjunto, estos tres proveedores concentran la mayor parte del mercado mundial de FPGAs, tanto en segmentos de alto desempeño como en nichos de bajo consumo y aplicaciones educativas [36]. Kmila toma como universo de dispositivos soportados un catálogo representativo de las familias más empleadas en el ámbito académico y en proyectos de hardware de gama media, cubriendo los tres fabricantes mencionados.

2.3.1. Historia de los Fabricantes

2.3.1.1. AMD (Xilinx)

Xilinx fue fundada en 1984 en Silicon Valley por Ross Freeman, Bernard Vonderschmitt y James V. Barnett II, con la premisa de que el costo de los transistores decrecería lo suficiente como para justificar arquitecturas lógicas configurables por el usuario final [36]. En 1985 introdujo el *XC2064*, reconocido como el primer FPGA comercial, con apenas 64 bloques lógicos configurables (CLBs) y 85 000 transistores. Durante las décadas siguientes, la compañía consolidó la arquitectura basada en bloques configurables, interconexión segmentada y memoria de configuración SRAM, que se mantiene vigente en la mayoría de FPGAs actuales.

La evolución tecnológica se materializó en líneas diferenciadas: la familia **Spartan** (introducida en 1998) orientada al segmento de bajo costo y uso educativo; la familia **Virtex** (1998) para aplicaciones de alto desempeño; y la familia **Artix-7** (2010), perteneciente a la serie 7 junto con *Kintex-7* y *Virtex-7*, que consolidó el uso del nodo de fabricación de 28 nm. Posteriormente se introdujeron la generación *UltraScale* (2013) y la plataforma *Zynq* (2011), que integra un procesador ARM Cortex-A9 con lógica programable en el mismo chip [37].

En febrero de 2022, Advanced Micro Devices (AMD) completó la adquisición de Xilinx por aproximadamente 49 000 millones de dólares estadounidenses, en lo que constituyó la mayor operación de la industria de semiconductores hasta esa fecha. A partir de ese momento, la línea de FPGAs pasó a operar bajo el *AMD Adaptive Computing Group*, manteniendo la nomenclatura histórica de sus familias [5], [38].

2.3.1.2. Intel (Altera)

Altera Corporation fue fundada en 1983 en San José, California, por Robert Hartmann, James Sansbury, Paul Newhagen y Michael Magranet, inicialmente como fabricante de dispositivos lógicos programables borrables (EPLDs). Su primer producto comercial, el *EP300*, apareció en 1984 y consolidó a la empresa como pionera en el segmento de lógica reconfigurable no volátil [36]. Durante los años noventa, Altera consolidó las familias **MAX** (dispositivos lógicos complejos, CPLDs) y **FLEX**, esta última siendo su primera arquitectura propiamente FPGA basada en LUTs.

A partir del año 2002, Altera reorganizó su oferta en tres líneas diferenciadas por costo y desempeño: **Cyclone** (bajo costo, orientada a aplicaciones industriales y académicas), **Arria** (gama media con transceptores integrados) y **Stratix** (alto desempeño). La familia *Cyclone IV*, introducida en 2009 en tecnología de 60 nm y utilizada ampliamente en tarjetas de desarrollo educativas como la DE0-Nano y la DE2-115, es particularmente relevante para el presente proyecto [39].

En diciembre de 2015, Intel Corporation completó la adquisición de Altera por 16 700 millones de dólares estadounidenses, integrándola como *Programmable Solutions Group* (PSG). Posteriormente, en 2024, Intel anunció la separación del grupo en una subsidiaria independiente que recuperó el nombre *Altera Corporation*, con un proceso de desinversión gradual formalizado a lo largo de 2025 [40], [41]. Para efectos de este documento, se emplea la denominación *Intel (Altera)* por ser la vigente en la mayor parte de la literatura y en las herramientas de desarrollo (Intel Quartus Prime).

2.3.1.3. Lattice Semiconductor

Lattice Semiconductor Corporation fue fundada en 1983 en Hillsboro, Oregón, con un enfoque inicial en dispositivos GAL (*Generic Array Logic*) basados en tecnología E^2PROM, alternativa no volátil a los productos de Altera y Xilinx en el segmento de lógica programable. A lo largo de los años noventa y dos mil consolidó las familias **ispLSI** (CPLDs) y **MachXO** (FPGAs no volátiles de bajo consumo), diferenciándose estratégicamente del segmento de alto desempeño dominado por Xilinx y Altera para concentrarse en aplicaciones de bajo consumo, automoción, comunicaciones e IoT [36].

El evento más relevante en su portafolio reciente fue la adquisición de SiliconBlue Technologies en diciembre de 2011, operación mediante la cual Lattice incorporó la familia **iCE40**, caracterizada por su consumo extremadamente bajo (del orden de microvatios en modo reposo) y por su baja densidad, lo que la hace idónea para dispositivos portátiles, sensores y hardware educativo [42], [43]. En 2014 la compañía introdujo la familia **ECP5**, orientada a aplicaciones de gama media con transceptores SerDes integrados, y en años posteriores las familias *MachXO3*, *CrossLink* y *CrossLink-NX*.

Un aspecto distintivo de Lattice, particularmente relevante para el contexto académico y de software libre, es que las familias iCE40 y ECP5 cuentan con flujos de síntesis y *place-and-route* completamente abiertos (*Project IceStorm*, *nextpnr* y *Yosys*), lo que las convierte en la opción preferida en entornos de investigación, *hardware hacking* y educación con herramientas de código abierto [17], [44].

2.3.2. Catálogo de Dispositivos Soportados por Kmila

Kmila incorpora un catálogo de 34 dispositivos FPGA distribuidos entre los tres fabricantes descritos, seleccionados por su presencia en el ámbito académico, su disponibilidad comercial en México y su cobertura del espectro de capacidades (desde dispositivos de entrada con menos de 2 000 LUTs hasta dispositivos de gama media-alta con más de 200 000 celdas lógicas). La distribución por fabricante es la siguiente:

- **AMD (Xilinx):** 16 dispositivos, distribuidos en 8 modelos de la familia Spartan-6 y 8 modelos de la familia Artix-7.
- **Intel (Altera):** 9 dispositivos de la familia Cyclone IV E.
- **Lattice:** 9 dispositivos, distribuidos en 3 modelos iCE40 HX, 2 modelos iCE40 UltraPlus y 4 modelos ECP5.

Las Tablas 2.4, 2.5 y 2.6 presentan una síntesis de los recursos principales de cada dispositivo, enfocada en las métricas pedagógicamente más relevantes: unidades lógicas (LUTs, celdas lógicas o elementos lógicos, según la terminología de cada fabricante), registros (flip-flops), memoria interna en bloques BRAM (expresada en Kbits totales), bloques DSP, PLLs y número máximo de pines de entrada/salida de usuario. La especificación completa de cada dispositivo, incluyendo el tipo de bloque BRAM, el tipo de multiplicador DSP, la frecuencia máxima de PLL, el voltaje del núcleo y el número de bancos de I/O, se presenta en el Anexo A.4.

Tabla 2.4: *Dispositivos FPGA de AMD (Xilinx) soportados por Kmila.*

Modelo	Familia	Celdas lóg.	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
XC6SLX4	Spartan-6	3 840	4 800	216	8	2	132
XC6SLX9	Spartan-6	9 152	11 440	576	16	2	200
XC6SLX16	Spartan-6	14 579	18 224	576	32	2	232
XC6SLX25	Spartan-6	24 051	30 064	936	38	4	266
XC6SLX45	Spartan-6	43 661	54 576	2 088	116	4	358
XC6SLX75	Spartan-6	74 637	93 296	3 096	180	6	498
XC6SLX100	Spartan-6	101 261	126 576	4 824	180	6	498
XC6SLX150	Spartan-6	147 443	184 304	4 824	180	6	576
XC7A12T	Artix-7	12 800	16 000	720	40	5	170
XC7A15T	Artix-7	16 640	20 800	900	45	5	250
XC7A25T	Artix-7	23 360	29 200	1 170	80	5	250
XC7A35T	Artix-7	33 280	41 600	1 800	90	5	250
XC7A50T	Artix-7	52 160	65 200	2 700	120	5	250
XC7A75T	Artix-7	75 520	94 400	3 780	180	6	300
XC7A100T	Artix-7	101 440	126 800	4 860	240	6	500
XC7A200T	Artix-7	215 360	269 200	13 140	740	10	500

Tabla 2.5: *Dispositivos FPGA de Intel (Altera) soportados por Kmila.*

Modelo	Familia	LEs	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
EP4CE6	Cyclone IV E	6 272	6 272	270	15	2	179
EP4CE10	Cyclone IV E	10 320	10 320	414	23	2	179
EP4CE15	Cyclone IV E	15 408	15 408	504	56	4	343
EP4CE22	Cyclone IV E	22 320	22 320	594	66	4	153
EP4CE30	Cyclone IV E	28 848	28 848	594	66	4	532
EP4CE40	Cyclone IV E	39 600	39 600	1 134	116	4	532
EP4CE55	Cyclone IV E	55 856	55 856	2 340	154	4	374
EP4CE75	Cyclone IV E	75 408	75 408	2 745	200	4	426
EP4CE115	Cyclone IV E	114 480	114 480	3 888	266	4	528

Tabla 2.6: *Dispositivos FPGA de Lattice Semiconductor soportados por Kmila.*

Modelo	Familia	LUTs	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
iCE40HX1K	iCE40 HX	1 280	1 280	64	0	1	96
iCE40HX4K[†]	iCE40 HX	—	—	—	0	—	178
iCE40HX8K	iCE40 HX	7 680	7 680	128	0	1	206
iCE40UP3K	iCE40 UltraPlus	2 800	2 800	80	4	1	39
iCE40UP5K	iCE40 UltraPlus	5 280	5 280	120	8	1	39
LFE5U-12	ECP5	12 000	12 000	576	12	2	197
LFE5U-25	ECP5	24 000	24 000	1 008	28	2	245
LFE5U-45	ECP5	44 000	44 000	1 944	72	4	259
LFE5U-85	ECP5	84 000	84 000	3 744	156	4	365

[†] El dispositivo iCE40HX4K comparte silicio con el iCE40HX8K, con recursos lógicos parcialmente deshabilitados; Lattice no publica cifras oficiales para LUTs, flip-flops ni capacidad de BRAM efectiva, por lo que el catálogo del software omite esos valores.

Las terminologías empleadas por cada fabricante para las unidades lógicas básicas no son equivalentes en sentido estricto: Xilinx denomina *celdas lógicas* (*logic cells*) a la suma ponderada de LUTs y slices, Altera utiliza *elementos lógicos* (*logic elements*, LEs) que contienen una LUT de 4 entradas y un registro, y Lattice emplea *LUTs de 4 entradas* como unidad base en iCE40 y ECP5. Esta heterogeneidad implica que la comparación directa del conteo de unidades lógicas entre fabricantes no refleja fielmente la capacidad real del dispositivo; para comparaciones precisas es necesario considerar también el tipo de LUT, la presencia de cadenas de acarreo dedicadas y la estructura del bloque DSP. Para efectos de Kmila, esta información se presenta al usuario como metadato informativo del dispositivo, sin que el simulador comportamental dependa de ella, ya que la simulación se realiza a nivel funcional e independiente de la tecnología objetivo.

2.4. Educación en Diseño Digital

2.4.1. Problemática Actual

La enseñanza del diseño digital en instituciones de educación superior enfrenta una tensión fundamental entre la naturaleza práctica de la disciplina y las restricciones materiales de los entornos académicos. El diseño de hardware, a diferencia de la programación de software, requiere en última instancia la validación sobre dispositivos físicos (FPGAs, CPLDs o ASICs) cuya adquisición, mantenimiento y disponibilidad están sujetos a limitaciones presupuestales y logísticas [25].

En el caso particular de la ESCOM, las asignaturas de Fundamentos de Diseño Digital, Arquitectura de Computadoras, Diseño de Sistemas Digitales y Sistemas en Chip conforman la columna vertebral del área de hardware dentro del plan de estudios de la carrera de Ingeniería en Sistemas Computacionales. Estas materias requieren que el estudiante adquiera competencia tanto en la descripción de hardware mediante HDL como en la verificación del comportamiento de los diseños mediante simulación.

Sin embargo, la práctica individual se ve restringida por varios factores concurrentes:

- **Disponibilidad limitada de laboratorios.** Los laboratorios de hardware son recursos compartidos entre múltiples grupos y asignaturas, lo que reduce el tiempo efectivo de práctica por estudiante a las horas asignadas en el horario curricular.
- **Dependencia de herramientas comerciales pesadas.** Las herramientas estándar de la industria (Vivado, Quartus Prime) requieren instalaciones de decenas de gigabytes, hardware con especificaciones elevadas y están restringidas a Windows y Linux, excluyendo a estudiantes con macOS o equipos de gama baja.
- **Curva de aprendizaje fragmentada.** Las alternativas de código abierto (GHDL, Icarus Verilog, GTKWave) requieren la instalación, configuración y coordinación de múltiples herramientas independientes operadas por línea de comandos, lo que desvía el esfuerzo del estudiante hacia problemas de infraestructura en lugar de concentrarse en el aprendizaje del diseño digital.
- **Ausencia de retroalimentación inmediata.** A diferencia de los entornos de desarrollo de software, donde el ciclo editar-compilar-ejecutar es inmediato, el flujo de trabajo típico en diseño digital involucra múltiples pasos manuales (edición, compilación, elaboración, simulación, visualización) que ralentizan la experimentación.

2.4.2. Limitaciones del Hardware en el Aula

El acceso a tarjetas FPGA representa la barrera económica más significativa para la práctica individual de diseño digital. Como se documentó en la Sección 1.5, una tarjeta Altera Cyclone IV tiene un costo aproximado de $1,866 MXN y una Digilent Basys 3 (Xilinx Artix-7) de $3,400 MXN [3], [4]. Estos costos resultan prohibitivos para la mayoría de los estudiantes de instituciones públicas como la ESCOM, donde la adquisición de hardware queda fuera del presupuesto personal típico.

A nivel institucional, equipar un laboratorio con tarjetas suficientes para trabajo individual implica una inversión considerable, a la que se suman los costos de mantenimiento, reposición de unidades dañadas y actualización tecnológica conforme los fabricantes introducen nuevas familias de dispositivos. En la práctica, esto resulta en que los estudiantes trabajan en equipos de tres o más personas por tarjeta, lo que reduce drásticamente la experiencia individual de aprendizaje.

Adicionalmente, el hardware FPGA presenta limitaciones inherentes para el aprendizaje:

- **Depuración opaca.** Las señales internas de un diseño implementado en FPGA no son directamente observables sin herramientas especializadas como analizadores lógicos integrados (ChipScope en Xilinx, SignalTap en Intel), cuyo uso requiere conocimientos avanzados que exceden el nivel de un curso introductorio.
- **Ciclo de retroalimentación lento.** El flujo completo desde la modificación del código hasta la verificación en hardware (síntesis, place-and-route, generación de bitstream, programación del dispositivo) puede tomar varios minutos incluso para diseños pequeños, lo que desincentiva la experimentación iterativa.
- **Portabilidad nula.** El hardware de laboratorio no puede trasladarse al domicilio del estudiante, limitando la práctica al horario y espacio físico del laboratorio.

Estas limitaciones no implican que el hardware FPGA sea prescindible en la formación del ingeniero. Por el contrario, la validación sobre hardware real es una competencia indispensable. Sin embargo, la simulación por software puede cubrir las etapas iniciales del aprendizaje, donde el objetivo es comprender la semántica del lenguaje HDL y el comportamiento lógico del diseño, reservando el hardware para las etapas posteriores de validación e implementación.

2.4.3. Simulación como Complemento Pedagógico

La simulación de código HDL por software constituye una herramienta pedagógica que complementa, sin reemplazar, la práctica con hardware real. Su valor educativo radica en varios aspectos:

Accesibilidad universal. Un simulador por software se ejecuta en cualquier dispositivo personal sin requerir hardware especializado. Esto permite que cada estudiante practique de forma individual, en cualquier momento y lugar, eliminando las restricciones de horario y espacio del laboratorio.

Ciclo de retroalimentación inmediato. La simulación comportamental omite las etapas de síntesis, place-and-route y programación del dispositivo, reduciendo el ciclo de iteración de minutos a segundos. Esta inmediatez favorece la experimentación y el aprendizaje por prueba y error, que es el mecanismo natural de adquisición de habilidades de programación [25].

Observabilidad completa. A diferencia del hardware, donde las señales internas son opacas, la simulación permite inspeccionar el valor de cualquier señal en cualquier instante de tiempo. Los visores de formas de onda muestran la evolución temporal de todas las señales del diseño, facilitando la comprensión del comportamiento concurrente y la identificación de errores lógicos.

Entorno seguro para el error. En simulación, un diseño incorrecto no daña ningún dispositivo ni consume recursos irrecuperables. El estudiante puede cometer errores, analizar sus consecuencias y corregirlos sin costo material, lo que fomenta una actitud exploratoria frente al aprendizaje.

Preparación para el laboratorio. La simulación funciona como etapa de pre-laboratorio: el estudiante verifica la corrección lógica de su diseño antes de implementarlo en hardware, optimizando el tiempo efectivo de laboratorio y reduciendo los ciclos de depuración sobre la tarjeta FPGA.

En el contexto del presente proyecto, Kmila se concibe como una herramienta que integra el ciclo completo de edición, análisis y simulación de código VHDL en una sola aplicación multiplataforma, con el objetivo de reducir las barreras de entrada al aprendizaje de diseño digital y maximizar el tiempo que el estudiante dedica a comprender el comportamiento del hardware descrito, en lugar de gestionar la infraestructura de herramientas necesaria para simularlo.

Capítulo 3

Marco Conceptual

Este capítulo presenta las tecnologías, metodologías y herramientas seleccionadas para el desarrollo del sistema, así como la descripción de la solución propuesta. A diferencia del Capítulo 2, que aborda los fundamentos teóricos del dominio de aplicación (diseño digital, HDL, simulación), este capítulo se centra en las decisiones de ingeniería de software que sustentan la implementación de Kmila.

3.1. Tecnologías de Desarrollo

3.1.1. .NET MAUI

.NET Multi-platform App UI (.NET MAUI) es un framework de código abierto desarrollado por Microsoft para la creación de aplicaciones nativas multiplataforma a partir de un único código base en C# y XAML [23]. MAUI es el sucesor de Xamarin.Forms y forma parte integral del ecosistema .NET desde la versión 6. El presente proyecto utiliza la versión incluida en .NET 9; si bien .NET 10 ya se encuentra disponible, se mantiene la versión 9 para garantizar la compatibilidad total del ecosistema, dado que los releases iniciales de una versión mayor pueden presentar incompatibilidades con bibliotecas de terceros. La migración a .NET 10 se realizará una vez que se confirme la estabilidad del nuevo release.

La arquitectura de .NET MAUI se fundamenta en una capa de abstracción que mapea controles de interfaz de usuario a sus equivalentes nativos en cada plataforma objetivo: Android (vistas Android nativas), iOS (UIKit), macOS (Mac Catalyst) y Windows (WinUI 3). Esto permite que una aplicación MAUI se compile en un binario nativo para cada plataforma, aprovechando las APIs y el rendimiento del sistema operativo subyacente, a diferencia de los enfoques basados en contenedores web que ejecutan la interfaz dentro de un navegador embebido.

Un componente clave de .NET MAUI para el presente proyecto es **BlazorWebView**, que permite alojar componentes de Blazor dentro de una aplicación MAUI nativa. Este modelo, denominado **Blazor Hybrid**, combina las ventajas de ambos frameworks: la distribución nativa de MAUI (acceso a APIs del dispositivo, instalación desde stores, ejecución offline) con el modelo de componentes de Blazor (interfaz definida en Razor/HTML/CSS, reutilización de componentes entre plataformas). En Kmila, la aplicación MAUI actúa como contenedor nativo que aloja la totalidad de la interfaz de usuario mediante BlazorWebView, delegando la lógica de presentación a componentes Razor compartidos [23], [24].

La selección de .NET MAUI se justifica por tres factores: (1) permite cubrir cuatro plataformas nativas (Windows, macOS, Android, iOS) desde un único proyecto, reduciendo drásticamente el esfuerzo de desarrollo y mantenimiento; (2) la integración nativa con Blazor mediante BlazorWebView habilita la reutilización completa de la interfaz de usuario con el proyecto web; y (3) el acceso a APIs nativas del dispositivo (sistema de archivos, almacenamiento

local, notificaciones) es directo, sin necesidad de puentes o plugins adicionales.

3.1.2. Blazor

Blazor es un framework de Microsoft para la construcción de interfaces de usuario web interactivas utilizando C# en lugar de JavaScript [24]. Forma parte de ASP.NET Core y permite definir componentes de interfaz mediante la sintaxis Razor, que combina marcado HTML con lógica en C# dentro de archivos `.razor`.

Blazor ofrece múltiples modelos de hospedaje, de los cuales dos son relevantes para Kmila:

- **Blazor Hybrid** (utilizado en la aplicación MAUI). Los componentes Razor se ejecutan directamente en el proceso nativo de la aplicación, con acceso completo a los recursos del dispositivo. La interfaz se renderiza en un control WebView embebido, pero la lógica de negocio y el acceso a datos se ejecutan de forma nativa, sin comunicación con un servidor remoto. Este es el modelo empleado por Kmila en sus versiones para Windows, macOS, Android e iOS.
- **Blazor Server** (utilizado en el proyecto web). Los componentes Razor se ejecutan en el servidor ASP.NET Core, y la interfaz se actualiza en el navegador del cliente mediante una conexión SignalR en tiempo real. Cada interacción del usuario se transmite al servidor, que procesa el evento, calcula el diferencial del DOM y lo envía de vuelta al navegador. Este modelo permite que la aplicación web funcione en cualquier navegador moderno sin requerir la descarga de un runtime adicional, lo que resulta adecuado para cubrir la plataforma Linux y para usuarios que prefieren no instalar una aplicación nativa.

La arquitectura de Kmila se estructura en torno a una **biblioteca de clases Razor compartida** (Kmila.Shared) que contiene la totalidad de los componentes de interfaz, páginas, servicios y modelos de datos. Tanto el proyecto MAUI como el proyecto web referencian esta biblioteca, de modo que la interfaz de usuario se define una sola vez y se ejecuta en ambos contextos sin duplicación de código. Esta estrategia de código compartido es posible gracias a que los componentes Razor son agnósticos respecto al modelo de hospedaje: el mismo componente funciona tanto en BlazorWebView (MAUI) como en Blazor Server (web), y la inyección de dependencias permite sustituir las implementaciones específicas de cada plataforma cuando es necesario.

3.1.3. C# y .NET 9

C# es un lenguaje de programación multiparadigma (orientado a objetos, funcional, genérico) desarrollado por Microsoft como parte de la plataforma .NET [45]. Desde su primera versión en 2002, C# ha evolucionado de forma continua, incorporando características modernas como expresiones lambda, LINQ, programación asíncrona (async/await), pattern matching, records y tipos de referencia anulables. La versión 13 del lenguaje, incluida en .NET 9, es la empleada en el presente proyecto.

.NET 9 es la plataforma unificada de desarrollo de Microsoft utilizada en el presente proyecto, publicada en noviembre de 2024 [46]. Consolida en un único SDK los frameworks previamente separados (.NET Framework, .NET Core, Mono, Xamarin) bajo una implementación común del runtime (CoreCLR para servidor y escritorio, Mono para móviles y WebAssembly). Cabe señalar que .NET 10 fue publicado posteriormente; sin embargo, se mantiene .NET 9 como plataforma objetivo para garantizar la estabilidad y compatibilidad plena con todas las dependencias del proyecto (BlazorMonaco, sqlite-net-pcl, Blazor.Bootstrap, entre otras). La migración a .NET 10 se contempla una vez que el ecosistema de bibliotecas confirme su compatibilidad con la nueva versión. Las características relevantes de .NET 9 para el proyecto incluyen:

- **Compilación AOT (Ahead-of-Time):** permite compilar la aplicación directamente a código nativo, reduciendo el tiempo de inicio y el consumo de memoria en dispositivos móviles.
- **Rendimiento mejorado del garbage collector:** optimizaciones en el recolector de basura que reducen las pausas durante la ejecución, relevante para la simulación en tiempo real.
- **Soporte multiplataforma nativo:** un mismo proyecto puede compilarse para Windows (x64/ARM64), macOS (x64/ARM64), Linux (x64/ARM64), Android e iOS sin modificaciones al código fuente.
- **Hot Reload:** permite aplicar cambios en el código C# y Razor durante la ejecución sin reiniciar la aplicación, acelerando el ciclo de desarrollo.

La selección de C# y .NET como plataforma de desarrollo se justifica por la unificación del ecosistema: el mismo lenguaje y las mismas bibliotecas se utilizan en la aplicación nativa (MAUI), la aplicación web (Blazor Server), el motor de simulación (Parser, Interpreter, Debugger) y la capa de persistencia (SQLite). Esta uniformidad elimina las fricciones de interoperabilidad entre lenguajes y reduce la complejidad del proyecto.

3.1.4. SQLite

SQLite es un motor de base de datos relacional embebido, implementado como una biblioteca en C sin dependencias externas, que almacena la base de datos completa en un único archivo [47]. A diferencia de los sistemas de gestión de bases de datos cliente-servidor (PostgreSQL, MySQL, SQL Server), SQLite no requiere un proceso separado ni configuración de red: la aplicación accede directamente al archivo de base de datos mediante llamadas a la biblioteca.

En Kmila, SQLite se emplea mediante la biblioteca **sqlite-net-pcl** (v1.9.172), un micro-ORM para .NET que proporciona acceso asíncrono a SQLite con un API basado en atributos y expresiones LINQ [48]. La clase `ApplicationDbContext` del proyecto extiende `SQLiteAsyncConnection` y define las tablas para las entidades del dominio (proyectos, archivos de proyecto, configuraciones).

La selección de SQLite se fundamenta en tres criterios: (1) es la base de datos embebida más desplegada del mundo, con soporte nativo en Android, iOS, macOS, Windows y Linux, lo que garantiza compatibilidad multiplataforma sin configuración; (2) su naturaleza embebida elimina la necesidad de infraestructura de servidor, alineándose con el requisito de que Kmila funcione completamente offline; y (3) su bajo consumo de recursos la hace adecuada para dispositivos móviles con memoria y almacenamiento limitados.

3.2. Plataformas Soportadas y Requisitos del Sistema

Kmila se distribuye en dos modalidades complementarias que cubren el conjunto completo de plataformas objetivo descritas en los requerimientos del proyecto: una **aplicación nativa multiplataforma** construida con .NET MAUI (Android, iOS, iPadOS, Windows y macOS) y una **aplicación web** construida con Blazor Server (acceso desde cualquier navegador moderno; despliegue en servidor Linux). Ambas modalidades comparten la misma biblioteca de lógica y vistas (`Kmila.Shared`), por lo que la experiencia funcional es equivalente entre ellas; las diferencias se concentran en el mecanismo de despliegue, el ciclo de vida de inyección de dependencias y los requisitos de conexión.

La Tabla 3.1 resume las versiones mínimas y recomendadas de cada plataforma soportada, junto con los requisitos de hardware. Los valores mínimos se extrajeron directamente de los descriptores de proyecto (`Kmila.csproj`, `Package.appxmanifest`, `Info.plist`); las recomendaciones reflejan las versiones bajo las cuales se ha verificado el funcionamiento durante el desarrollo.

3.2.1. Modalidades de Despliegue

Modalidad nativa (MAUI). La aplicación se empaqueta como ejecutable nativo de cada sistema operativo (APK en Android, IPA en iOS/iPadOS, MSIX o paquete autónomo en Windows, App Bundle en macOS). Toda la lógica, los recursos (catálogo FPGA, diccionarios de idioma, lecciones del módulo *Aprender*) y la base de datos SQLite se incluyen en el binario o se inicializan localmente al primer arranque. Esta modalidad funciona **completamente sin conexión a internet** tras la instalación.

Modalidad web (Blazor Server). La aplicación se ejecuta en un servidor con .NET 9 (típicamente un servidor Linux), y el cliente accede desde cualquier navegador moderno. La interfaz se renderiza en el servidor y se sincroniza con el navegador mediante una conexión persistente *WebSocket* sobre el protocolo SignalR. Esta modalidad requiere conexión continua entre cliente y servidor, pero elimina por completo los requisitos de instalación en el cliente: basta con abrir una URL.

Ambas modalidades comparten el mismo motor de simulación, el mismo catálogo de dispositivos FPGA, el mismo conjunto de idiomas y el mismo módulo de aprendizaje. La modalidad web es la vía de acceso recomendada para sistemas Linux, ya que .NET MAUI no compila para Linux como objetivo nativo.

3.2.2. Versiones Soportadas

Tabla 3.1: *Plataformas soportadas: versiones mínimas, recomendadas y arquitecturas de procesador.*

Plataforma	Versión mínima	Recomendada	Arquitecturas
Android	7.0 Nougat (API 24)	12 Snow Cone (API 31)+	ARMv7, ARM64, x86_64
iOS	iOS 15	iOS 17+	ARM64 (iPhone 6s+)
iPadOS	iPadOS 15	iPadOS 17+	ARM64 (iPad 5ª gen.+)
macOS	macOS 12 Monterey	macOS 14 Sonoma+	x64 (Intel), ARM64 (Apple Silicon)
Windows	Windows 10 v1809 (build 17763)	Windows 11 22H2+	x64
Linux (vía web)	Ubuntu 20.04 o equivalente†	Ubuntu 22.04 LTS+	x64, ARM64

† Distribuciones equivalentes oficialmente soportadas por el runtime de .NET 9: Debian 11, RHEL 8, Fedora 39, openSUSE 15, Alpine 3.18 (servidor).

Las versiones mínimas para Android y iOS quedan condicionadas por las restricciones de `SupportedOSPlatformVersion` declaradas en `Kmila.csproj`; la versión mínima de Windows está fijada por el rango `MinVersion=10.0.17763.0`, `MaxVersionTested=10.0.19041.0` declarado en `Package.appxmanifest`; macOS hereda la restricción de MacCatalyst 15.0, equivalente a macOS 12 Monterey. Linux no es un objetivo nativo de MAUI, por lo que el soporte en distribuciones Linux se obtiene exclusivamente a través de la modalidad web; las distribuciones listadas corresponden al conjunto soportado oficialmente por el runtime de .NET 9 [46].

3.2.2.1. Navegadores soportados (modalidad web)

Para la modalidad web, el cliente requiere un navegador con soporte completo de WebAssembly, WebSocket y módulos ES2020. La Tabla 3.2 resume los navegadores y versiones mínimas verificados.

Tabla 3.2: *Navegadores soportados para la modalidad web.*

Navegador	Versión mínima	Notas
Google Chrome	90 (abril 2021)	Plataforma de referencia
Microsoft Edge	90 (abril 2021)	Basado en Chromium
Mozilla Firefox	88 (abril 2021)	Soporte completo
Apple Safari	14 (sept. 2020)	macOS 11 / iOS 14
Chromium derivados (Brave, Opera, Vivaldi)	equivalente a Chrome 90+	Funcionalidad completa

3.2.3. Requisitos de Hardware

La Tabla 3.3 establece los requisitos mínimos de memoria RAM y almacenamiento por plataforma. Estos valores incluyen la huella del runtime `.NET` 9, los recursos empaquetados de Kmila (catálogo de 34 dispositivos FPGA, diccionarios de los 7 idiomas, manifiesto y bodies del módulo *Aprender*) y una holgura razonable para datos del usuario (proyectos, archivos VHDL, configuraciones guardadas).

Tabla 3.3: *Requisitos mínimos de hardware por plataforma.*

Plataforma	RAM	Almacenamiento	Procesador
Android	2 GB	150 MB libres	ARMv7-A o ARM64, doble núcleo $\geq$ 1.5 GHz
iOS / iPadOS	2 GB	150 MB libres	ARM64 (Apple A9 o posterior)
macOS	4 GB	400 MB libres	Intel x64 doble núcleo o Apple Silicon
Windows	4 GB	400 MB libres	x64 doble núcleo $\geq$ 1.8 GHz
Cliente web (cualquier OS)	2 GB	50 MB caché de navegador	Cualquiera con navegador moderno
Servidor web (Linux)	1 GB libre	200 MB libres	x64 o ARM64, 1 vCPU base

Los valores de RAM y almacenamiento son indicativos para el funcionamiento de la aplicación en estado de reposo y bajo cargas de trabajo típicas (un proyecto activo con un archivo VHDL de complejidad media, un dispositivo FPGA seleccionado y una simulación con 10^4 a 10^5 muestras de forma de onda). Para sesiones con múltiples proyectos abiertos simultáneamente, simulaciones con duraciones extensas o números elevados de señales observadas, se recomienda doblar los valores indicados.

3.2.4. Requisitos de Conexión y Permisos

Conectividad.

- **Modalidad nativa:** no requiere conexión a internet tras la instalación. Toda la funcionalidad (edición, análisis, simulación, visualización, persistencia) opera localmente.
- **Modalidad web:** requiere conexión persistente con el servidor durante toda la sesión. El protocolo subyacente es WebSocket (con fallback a long-polling sobre HTTP). Se recomienda un ancho de banda mínimo de 256 kbps y una latencia inferior a 200 ms para una experiencia fluida.

Permisos por plataforma.

- **Android:** `INTERNET` y `ACCESS_NETWORK_STATE` (utilizadas únicamente para descargar plantillas o componentes a futuro), `READ_EXTERNAL_STORAGE` y `WRITE_EXTERNAL_STORAGE` (para abrir y guardar archivos VHDL fuera del directorio de la aplicación).
- **iOS / iPadOS:** acceso al *sandbox* de la aplicación (estándar). El sistema operativo gestiona automáticamente los permisos de almacenamiento dentro del directorio asignado.
- **macOS / Windows / Linux (servidor):** acceso de lectura/escritura al directorio de instalación o al directorio del usuario donde se almacenan los proyectos (`./Projects/` por convención).

Pantalla mínima. El diseño responsivo se adapta desde 360 px de ancho efectivo (teléfonos en orientación vertical) hasta resoluciones de escritorio. Para la pantalla del editor con todas las herramientas visibles simultáneamente se recomienda un ancho mínimo de 1280 px.

3.3. Ingeniería de Software

3.3.1. Metodología de Desarrollo

El desarrollo de Kmila adopta la metodología **Kanban**, adaptada a las restricciones de un proyecto de Trabajo Terminal individual donde el equipo de desarrollo está conformado por una sola persona y los ciclos de retroalimentación están determinados por las revisiones con los directores del proyecto.

Kanban se fundamenta en la visualización del flujo de trabajo mediante un tablero con columnas que representan los estados de cada tarea (por ejemplo: *Backlog*, *En progreso*, *En revisión*, *Completado*). A diferencia de Scrum, que opera con iteraciones de duración fija (sprints), Kanban emplea un flujo continuo donde las tareas se incorporan y completan de manera individual conforme la capacidad lo permite. Este modelo resulta adecuado para un desarrollador individual, ya que elimina la sobrecarga ceremonial de los marcos ágiles diseñados para equipos multidisciplinarios (roles diferenciados, daily standups, retrospectivas, planificación de sprint) y permite adaptar las prioridades de forma inmediata ante cambios en los requerimientos o retroalimentación de los directores.

El flujo de trabajo se organiza de la siguiente manera:

- **Tablero Kanban con límite de trabajo en progreso (WIP).** Se mantiene un máximo de dos tareas simultáneas en estado “En progreso” para garantizar el enfoque y evitar la dispersión. Cada tarea corresponde a una funcionalidad concreta (por ejemplo, “implementar el editor de código con resaltado de sintaxis” o “implementar la persistencia de proyectos en SQLite”).
- **Priorización por valor educativo.** Las tareas se priorizan según su impacto directo en la experiencia del estudiante: primero la interfaz de edición y gestión de proyectos (TT1), después el motor de simulación y visualización de resultados (TT2).
- **Verificación en plataformas objetivo.** Cada tarea completada se verifica manualmente en al menos dos plataformas (Linux y Android, que constituyen el entorno de desarrollo principal) antes de considerarse terminada.
- **Refactorización progresiva.** La arquitectura modular del proyecto (descrita en la Sección 3.3.2) permite refactorizar componentes individuales sin afectar al resto del sistema, lo que facilita la evolución del diseño conforme se adquiere mayor comprensión del dominio.

El desarrollo se organiza en dos fases correspondientes a los periodos académicos TT1 y TT2, con entregas funcionales al final de cada periodo. Kanban no impone una cadencia fija de entrega, pero la estructura académica del Trabajo Terminal proporciona naturalmente los hitos de entrega y revisión que regulan el avance del proyecto.

3.3.2. Patrones de Diseño

La arquitectura de Kmila emplea varios patrones de diseño que promueven la separación de responsabilidades, la testabilidad y la reutilización de código:

Inyección de dependencias (DI). Todas las dependencias entre componentes se resuelven mediante el contenedor de inyección de dependencias integrado en .NET (`Microsoft.Extensions.DependencyInjection`). Los servicios se registran en el punto de entrada de cada aplicación (`MauiProgram.cs` para la versión nativa, `Program.cs` para la versión web) con el ciclo de vida apropiado (Singleton para servicios compartidos como `Traductor` y `ConfigurationService`, Scoped para contextos de base de datos). Este patrón permite que los componentes dependan de abstracciones (interfaces) en lugar de implementaciones concretas, facilitando la sustitución de servicios entre plataformas.

Patrón repositorio. El acceso a datos se encapsula detrás de la interfaz genérica `IRepository<T>`, que define operaciones CRUD asíncronas (`CreateAsync`, `UpdateAsync`, `RemoveAsync`, `GetListAsync`, `GetByAsync`). Las implementaciones concretas (`Projects`, `ProjectFiles`) encapsulan las consultas SQLite, de modo que los componentes de interfaz y los servicios de negocio no dependen directamente del mecanismo de persistencia. Este patrón facilita la migración futura a otros motores de almacenamiento sin modificar la lógica de negocio.

Arquitectura basada en eventos. El servicio central `ProgramBuilder`, que orquesta el flujo de compilación y simulación de código VHDL, emplea un modelo de comunicación basado en eventos (`OnSimulationEnded`, `OnProgressAdvanced`, `OnBuildProgressChanged`, `OnBuildError`). Los componentes de interfaz se suscriben a estos eventos para actualizar la visualización en tiempo real sin acoplamiento directo con la lógica del motor. Este patrón desacopla el motor de simulación de la capa de presentación y permite que múltiples componentes reaccionen a un mismo evento de forma independiente.

Biblioteca de clases compartida. Como se describió en la Sección 3.1.2, el proyecto Kmila.Shared centraliza los componentes Razor, servicios, repositorios y modelos de datos en una biblioteca de clases referenciada tanto por el proyecto MAUI como por el proyecto web. Este patrón de código compartido maximiza la reutilización y garantiza la consistencia funcional entre plataformas.

3.3.3. Herramientas de Gestión

El desarrollo del proyecto emplea las siguientes herramientas:

- **Git y GitHub.** Sistema de control de versiones distribuido utilizado para el seguimiento de cambios, la gestión de ramas y el respaldo del código fuente. El repositorio principal se aloja en GitHub, lo que habilita la distribución del código como proyecto de código abierto.
- **Visual Studio Code.** Editor de código ligero y extensible utilizado como entorno de desarrollo principal. Dado que el desarrollo se realiza íntegramente en Linux, Visual Studio Code es la herramienta más adecuada por su compatibilidad nativa con la plataforma, su integración con el SDK de .NET mediante la extensión C# Dev Kit, y su soporte para depuración, terminal integrada y control de versiones. No se utiliza Visual Studio 2022, ya que no está disponible de forma nativa en Linux.
- **Android Debug Bridge (ADB).** Herramienta de línea de comandos del SDK de Android utilizada para la depuración y despliegue de la aplicación en dispositivos Android físicos durante el desarrollo.
- **SDK de .NET 9.** Kit de desarrollo que incluye el compilador de C#, el runtime, las herramientas de compilación (MSBuild) y las plantillas de proyecto. Es el único requisito de infraestructura para compilar el proyecto completo.

3.4. Solución Propuesta

Kmila es una aplicación multiplataforma de código abierto diseñada para el aprendizaje y la depuración de código VHDL, orientada a estudiantes de ingeniería que cursan asignaturas de diseño digital. La aplicación integra en un solo entorno las funcionalidades que actualmente requieren la combinación de múltiples herramientas independientes: edición de código con resaltado de sintaxis, análisis léxico y sintáctico, simulación comportamental y visualización de resultados.

La solución se estructura en torno a tres pilares:

Entorno de edición integrado. Kmila incorpora el editor Monaco (el mismo motor que impulsa Visual Studio Code) mediante la biblioteca BlazorMonaco, proporcionando al estudiante un editor de código con resaltado de sintaxis para VHDL, numeración de líneas, plegado de bloques y búsqueda integrada. El editor se acompaña de un sistema de gestión de proyectos que permite crear, organizar y persistir múltiples archivos HDL en una base de datos SQLite local.

Motor de análisis y simulación. El núcleo computacional de Kmila se compone de cuatro módulos independientes desarrollados en C#: el **Parser** (análisis léxico y sintáctico de código VHDL, generación del árbol de sintaxis abstracta), el **Interpreter** (evaluación de expresiones mediante el algoritmo Shunting-yard, resolución de tipos y valores), el **Debugger** (orquestación del ciclo de simulación, gestión de breakpoints y ejecución paso a paso) y el **TimeMachine** (control del tiempo discreto de simulación con resolución configurable). Estos módulos implementan el modelo de eventos discretos y ciclos delta descrito en la Sección 2.2.1, permitiendo la simulación comportamental de diseños VHDL.

Cobertura multiplataforma desde un único código base. Mediante la combinación de .NET MAUI (Blazor Hybrid) y Blazor Server, Kmila se distribuye como aplicación nativa para Windows, macOS, Android e iOS, y como aplicación web accesible desde cualquier navegador moderno. La biblioteca compartida Kmila.Shared garantiza que la experiencia de usuario sea idéntica en todas las plataformas. Esta arquitectura resuelve directamente la brecha de cobertura multiplataforma identificada en el análisis comparativo de la Sección 1.4.6, donde ninguna herramienta existente ofrece presencia nativa en las seis plataformas objetivo.

Adicionalmente, Kmila incorpora un catálogo de dispositivos FPGA (familias Lattice iCE40, Intel/Altera Cyclone IV, Lattice ECP5) que permite al estudiante contextualizar su diseño respecto a los recursos disponibles en un dispositivo real (LUTs, flip-flops, bloques de RAM, pines de E/S), y un sistema de internacionalización con soporte para siete idiomas (español, inglés, francés, alemán, chino, árabe y japonés), alineado con el objetivo de accesibilidad universal.

El desarrollo se organiza en dos fases académicas: TT1 abarca la interfaz de usuario, la gestión de proyectos, el editor de código y la arquitectura del sistema; TT2 implementa el motor de análisis y simulación (análisis léxico/sintáctico, interpretación, depuración, control temporal), la visualización de formas de onda y la publicación en las tiendas de aplicaciones.

Capítulo 4

Análisis del Sistema

4.1. Modelo del Dominio

El modelo del dominio identifica los conceptos fundamentales del sistema y las relaciones entre ellos. Estos conceptos se derivan del análisis del problema descrito en la sección 1.5 y del alcance definido en la sección 1.8. Las Tablas 4.1 a 4.5 presentan las entidades del dominio agrupadas por categoría.

Tabla 4.1: *Entidades del dominio: gestión de proyecto.*

Entidad	Descripción
Proyecto	Contenedor lógico que agrupa archivos HDL, configuración de simulación y metadatos del usuario.
Archivo HDL	Archivo fuente en VHDL (`.vhd`) que contiene la descripción de un circuito digital. Un proyecto puede contener múltiples archivos.
Sesión	Estado persistente del espacio de trabajo del usuario, incluyendo archivos abiertos, posición del cursor y configuración activa.

Tabla 4.2: *Entidades del dominio: modelo HDL.*

Entidad	Descripción
Entidad (Entity)	Declaración VHDL que define la interfaz externa de un módulo: su nombre y sus puertos de entrada/salida.
Arquitectura (Architecture)	Cuerpo VHDL asociado a una entidad que describe su comportamiento o estructura interna.
Puerto (Port)	Punto de conexión de entrada (`in`), salida (`out`) o bidireccional (`inout`) declarado en una entidad.
Señal (Signal)	Elemento que transporta valores lógicos entre componentes dentro de una arquitectura.
Proceso (Process)	Bloque de ejecución secuencial dentro de una arquitectura, activado por una lista de sensibilidad.
Banco de pruebas (Testbench)	Archivo HDL especial que instancia la entidad bajo prueba y genera estímulos de entrada para la simulación.

Tabla 4.3: *Entidades del dominio: análisis de código.*

Entidad	Descripción
AST (Abstract Syntax Tree)	Representación jerárquica intermedia del código HDL generada tras el análisis sintáctico.
Diagnóstico	Mensaje de error, advertencia o información generado durante el análisis del código, asociado a una línea y columna específicas.
Punto de interrupción (Breakpoint)	Marca insertada por el usuario en una línea del código que detiene la simulación al ser alcanzada.

Tabla 4.4: *Entidades del dominio: simulación.*

Entidad	Descripción
Configuración de simulación	Conjunto de parámetros que definen la ejecución: tiempo máximo, resolución temporal, dispositivo FPGA objetivo y señales a observar.
Dispositivo FPGA	Modelo de referencia de un dispositivo FPGA comercial (familia, fabricante, cantidad de celdas lógicas, bloques de memoria) utilizado como contexto de la simulación.
Ciclo de simulación	Unidad discreta de avance temporal compuesta por una fase de actualización de señales y una fase de ejecución de procesos.
Forma de onda (Waveform)	Serie temporal de valores lógicos de una señal a lo largo de los ciclos de simulación.

Tabla 4.5: *Entidades del dominio: interfaz de usuario.*

Entidad	Descripción
Editor de código	Componente central de la interfaz basado en Monaco Editor que permite la escritura y edición de código HDL con resaltado de sintaxis.
Visualizador de ondas	Componente gráfico basado en Chart.js que presenta las formas de onda de las señales seleccionadas en un diagrama temporal.
Panel de diagnóstico	Componente que muestra la lista de errores, advertencias e información generados por el análisis del código.

Relaciones principales. Un **Proyecto** contiene uno o más **Archivos HDL**, cada uno de los cuales puede declarar una o más **Entidades**. Cada Entidad posee una **Arquitectura** asociada que contiene **Señales** y **Procesos**. Un **Banco de pruebas** instancia una Entidad y define los estímulos para la simulación. El análisis de un Archivo HDL produce un **AST** y, en caso de errores, uno o más **Diagnósticos**. La ejecución de la simulación, controlada por una **Configuración de simulación**, genera **Formas de onda** para cada señal observada, las cuales se presentan al usuario a través del **Visualizador de ondas**.

La Figura 4.1 presenta el diagrama del modelo del dominio con las entidades y relaciones descritas.

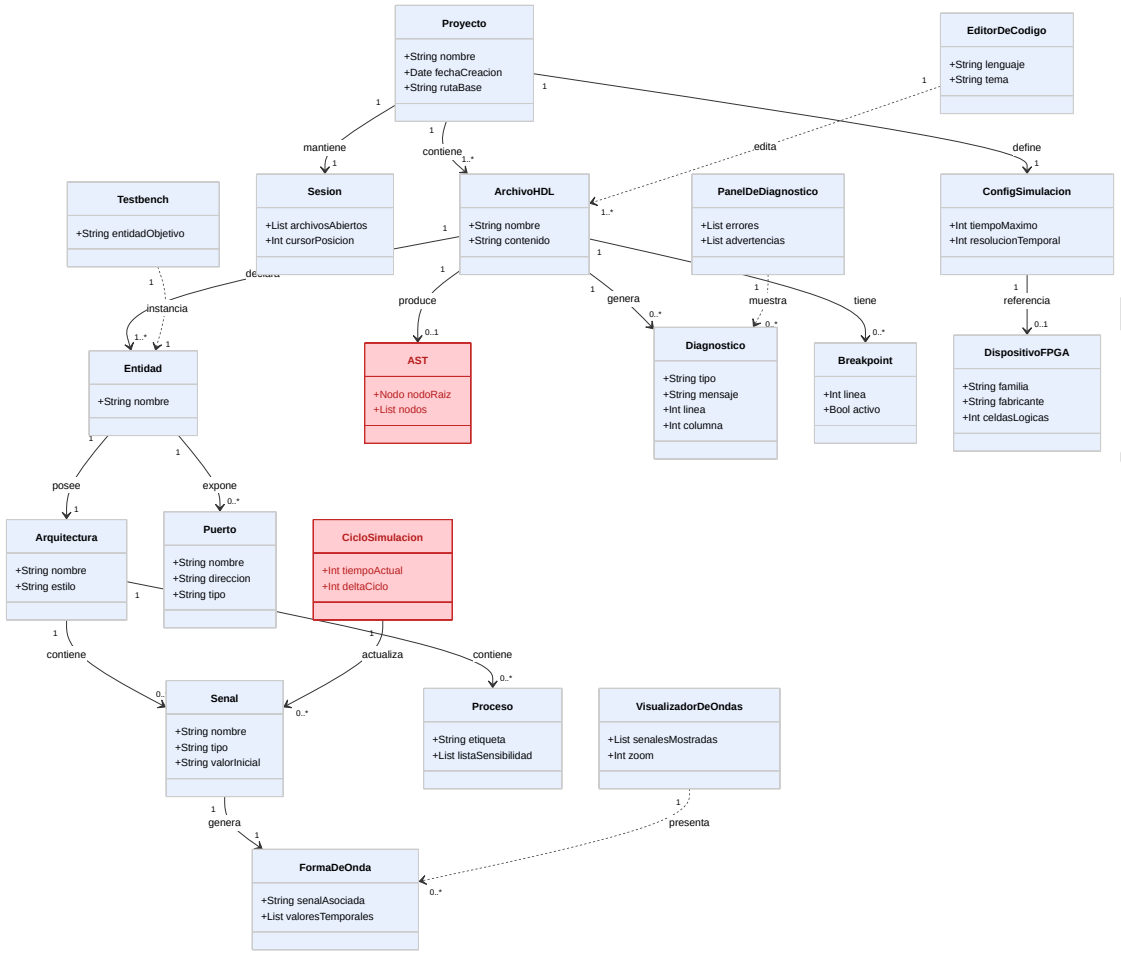

Figura 4.1: *Modelo del dominio completo del sistema. Las entidades en rojo (AST, CicloSimulacion) corresponden al motor de simulación, cuya implementación se contempla en TT2. Nota: por compatibilidad con el motor de renderizado de diagramas, los identificadores no incluyen caracteres especiales (acentos, ñ).*

Las Figuras 4.2 a 4.5 presentan vistas segmentadas del modelo del dominio por categoría, facilitando su lectura individual.

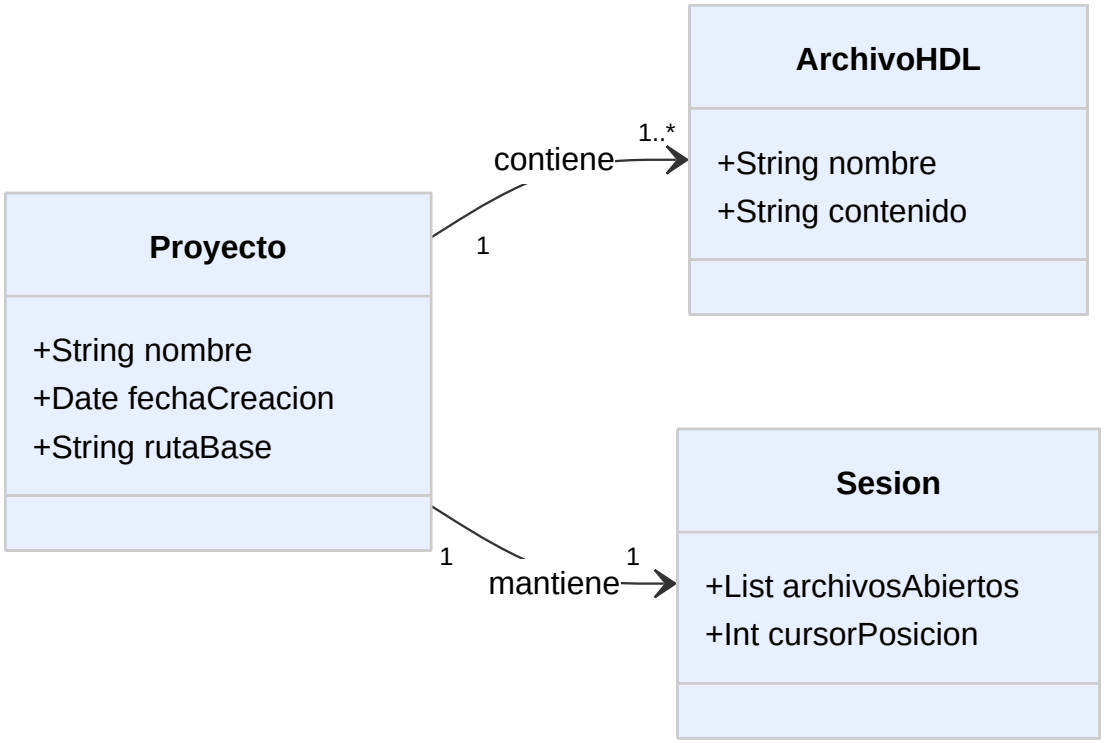

Figura 4.2: *Modelo del dominio — Gestión de proyecto. Nota: identificadores sin caracteres especiales por compatibilidad con el motor de renderizado.*

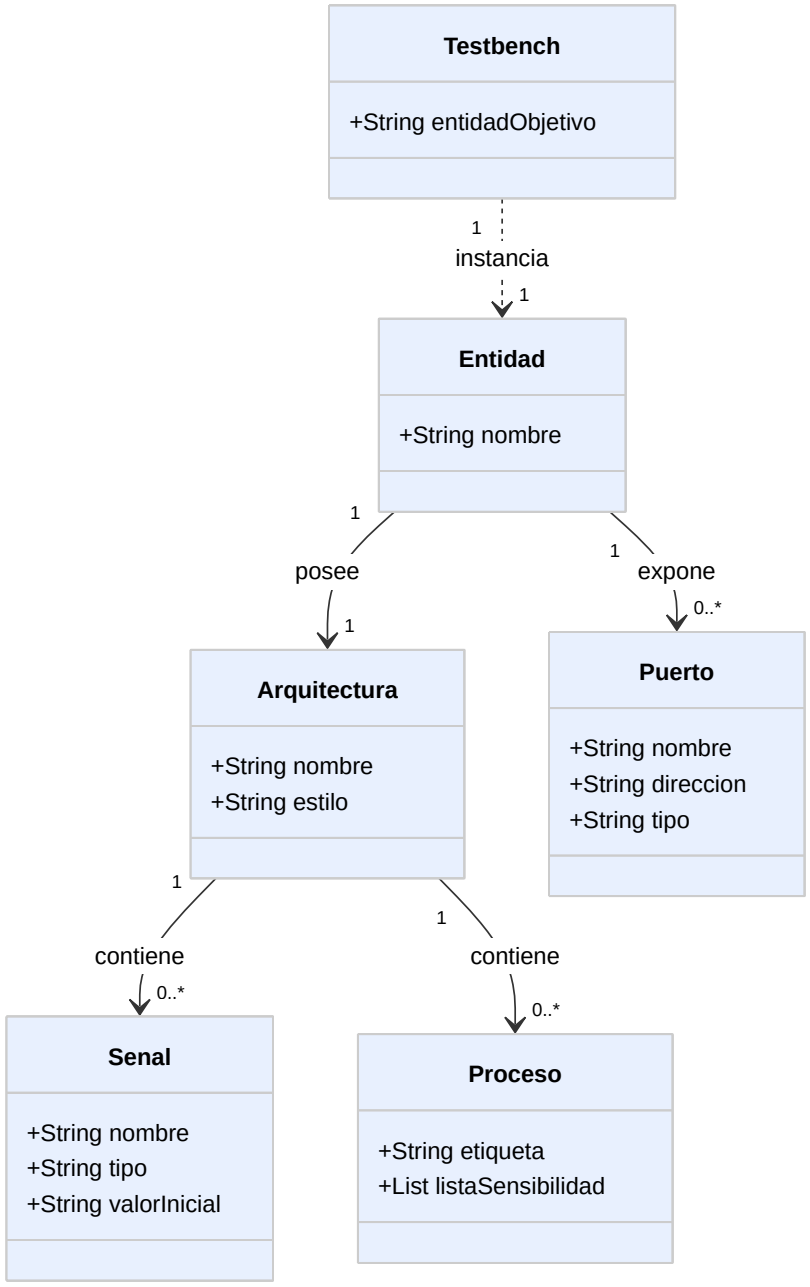

Figura 4.3: *Modelo del dominio — Modelo HDL. Nota: identificadores sin caracteres especiales por compatibilidad con el motor de renderizado.*

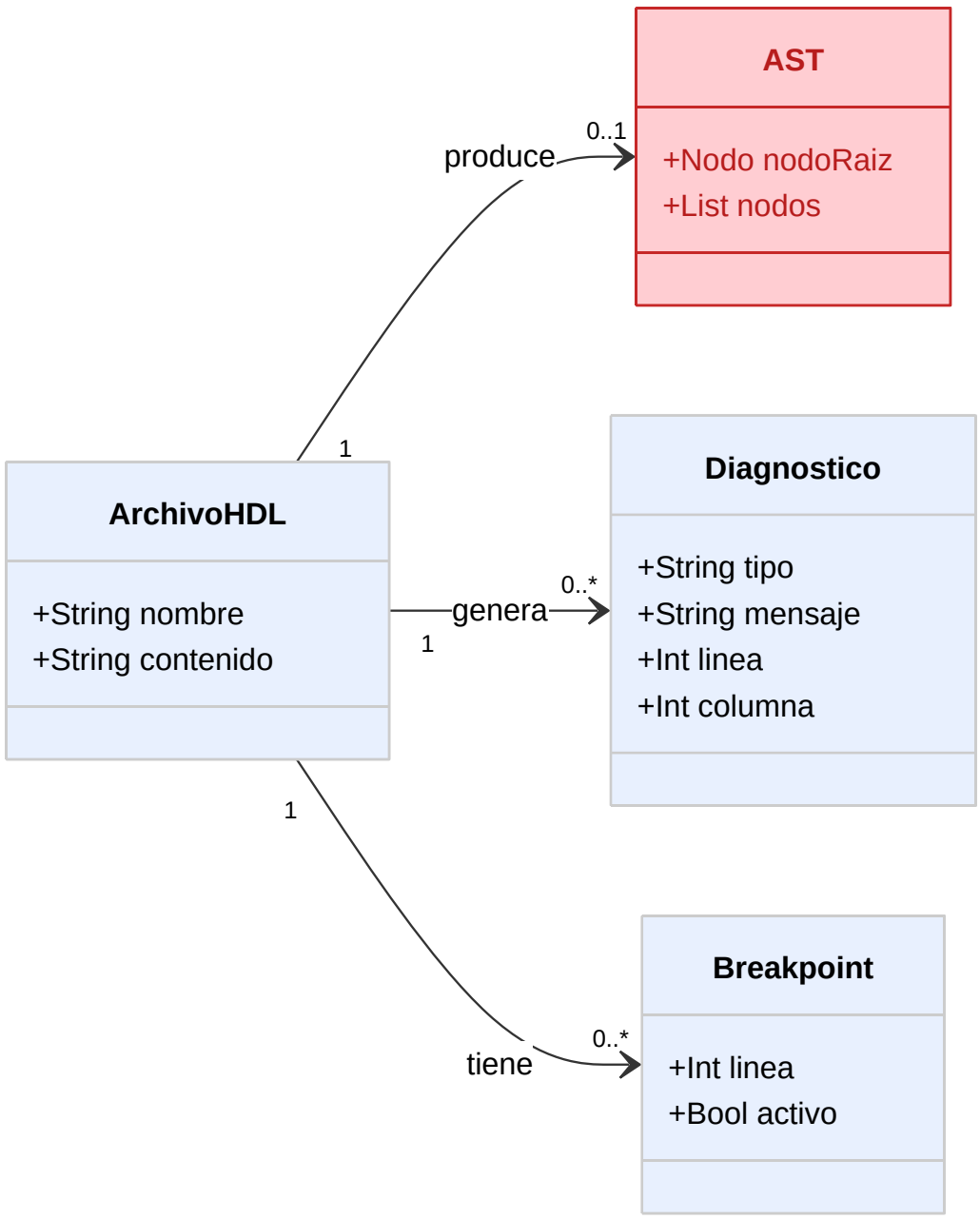

Figura 4.4: *Modelo del dominio — Análisis de código. La entidad AST (en rojo) se implementa en TT2. Nota: identificadores sin caracteres especiales por compatibilidad con el motor de renderizado.*

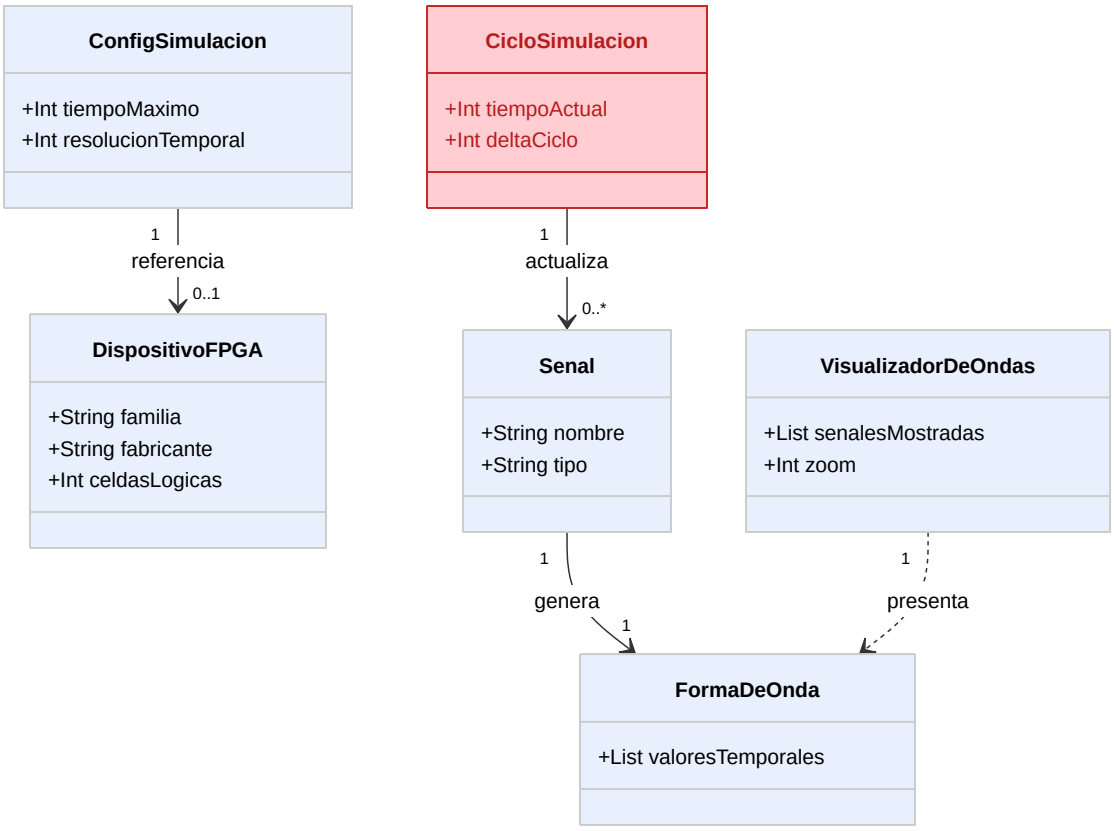

Figura 4.5: *Modelo del dominio — Simulación y visualización. La entidad CicloSimulacion (en rojo) se implementa en TT2. Nota: identificadores sin caracteres especiales por compatibilidad con el motor de renderizado.*

4.2. Requerimientos Funcionales

Los requerimientos funcionales se organizan en dos grupos correspondientes a las fases del proyecto. Los requerimientos de TT1 (interfaz de usuario y gestión) se detallan en su totalidad, dado que constituyen el alcance del presente trabajo. Los requerimientos de TT2 (motor de simulación) se enuncian para completitud del análisis.

Tabla 4.6: *Resumen de requerimientos funcionales — TT1 (interfaz de usuario y gestión).*

ID	Nombre	Descripción	Prioridad
RF-01	Gestionar proyectos	Crear, abrir, eliminar y listar proyectos que agrupen archivos HDL y configuración	Alta
RF-02	Gestionar archivos HDL	Crear, cargar y guardar archivos `.vhd` dentro de un proyecto con respaldo en base de datos	Alta
RF-03	Editar código HDL	Editar archivos VHDL con resaltado de sintaxis, seguimiento de posición del cursor e indicador de cambios sin guardar	Alta
RF-04	Navegar estructura del proyecto	Explorar archivos del proyecto mediante un panel lateral y una barra de navegación superior	Alta
RF-05	Seleccionar dispositivo FPGA	Elegir un modelo de FPGA de referencia de un catálogo de 34 dispositivos de 3 fabricantes	Media
RF-06	Configurar parámetros de simulación	Definir duración, escala temporal, frecuencia de reloj y estímulos por puerto	Media
RF-07	Visualizar diagnósticos y recursos	Mostrar entidades analizadas, puertos, señales y utilización de recursos del FPGA seleccionado	Alta
RF-08	Visualizar formas de onda	Presentar señales en un visor de ondas con codificación por color, zoom y exportación VCD	Alta
RF-09	Cambiar idioma de la interfaz	Seleccionar entre 7 idiomas (ES, EN, FR, DE, ZH, JA, AR) con dirección de escritura derecha-a-izquierda (*Right-to-Left*) para AR	Baja
RF-10	Cambiar tema visual	Alternar entre tema claro y tema oscuro	Baja
RF-11	Persistir datos localmente	Almacenar proyectos, archivos, configuraciones y preferencias del usuario en SQLite	Media

Tabla 4.7: *Resumen de requerimientos funcionales — TT2 (motor de simulación).*

ID	Nombre	Descripción	Prioridad
RF-12	Analizar sintaxis VHDL	Verificar la corrección sintáctica del código y generar diagnósticos de error	Alta
RF-13	Generar AST	Construir la representación jerárquica intermedia del código VHDL válido	Alta
RF-14	Ejecutar simulación	Simular la ejecución del circuito mediante el modelo de eventos discretos	Alta
RF-15	Controlar simulación	Iniciar, pausar, reanudar y detener la ejecución de la simulación	Alta
RF-16	Ejecutar paso a paso	Avanzar la simulación ciclo por ciclo para depuración	Media
RF-17	Insertar puntos de interrupción	Marcar líneas en el editor donde la simulación debe detenerse	Media
RF-18	Exportar resultados	Exportar formas de onda en formato VCD (Value Change Dump)	Baja

A continuación se detalla cada requerimiento funcional de TT1.

4.2.1. RF-01: Gestionar Proyectos

Campo	Descripción
ID	RF-01
Nombre	Gestionar proyectos
Descripción	El sistema debe permitir al usuario crear un nuevo proyecto (con nombre, descripción y ruta base), abrir un proyecto existente, eliminarlo y consultar la lista de proyectos registrados. La creación de un proyecto genera un directorio en el sistema de archivos y un registro en la base de datos local.
Prioridad	Alta
Entrada	Nombre, descripción y ruta de almacenamiento (creación); identificador del proyecto (apertura/eliminación)
Salida	Proyecto cargado en la interfaz con su estructura de archivos visible en el editor
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	El proyecto queda registrado en la tabla `Projects` de SQLite y su directorio existe en el sistema de archivos

4.2.2. RF-02: Gestionar Archivos HDL

Campo	Descripción
ID	RF-02
Nombre	Gestionar archivos HDL
Descripción	El sistema debe permitir crear nuevos archivos `.vhd` dentro de un proyecto, cargar archivos existentes desde el directorio del proyecto y guardar los cambios realizados. El contenido se almacena tanto en disco como en la base de datos SQLite como respaldo, con mecanismo de recuperación ante pérdida del archivo en disco.
Prioridad	Alta
Entrada	Nombre del archivo (creación); contenido editado (guardado)
Salida	Archivo registrado en la tabla `ProjectFiles` y escrito en el directorio del proyecto
Precondiciones	Existe un proyecto abierto
Postcondiciones	El archivo es accesible desde el panel de archivos del editor y su contenido persiste en disco y base de datos

4.2.3. RF-03: Editar Código HDL

Campo	Descripción
ID	RF-03
Nombre	Editar código HDL
Descripción	El sistema debe integrar el editor Monaco (BlazorMonaco 3.3.0) para la edición de archivos VHDL, proporcionando resaltado de sintaxis específico para VHDL, seguimiento de la posición del cursor (línea y columna), indicador visual de cambios sin guardar, detección del tipo de archivo por extensión y atajo de teclado Ctrl+S para guardar. Para archivos Markdown, el editor ofrece vista dividida con previsualización.
Prioridad	Alta
Entrada	Contenido del archivo HDL seleccionado
Salida	Contenido editado con resaltado de sintaxis y metadatos de posición
Precondiciones	Existe al menos un archivo abierto en el proyecto activo
Postcondiciones	Los cambios se reflejan en el editor; el indicador de cambios sin guardar se activa hasta que el usuario guarde

4.2.4. RF-04: Navegar Estructura del Proyecto

Campo	Descripción
ID	RF-04
Nombre	Navegar estructura del proyecto
Descripción	El sistema debe proporcionar un panel de navegación lateral (sidebar) con acceso a las secciones principales (Inicio, Proyectos, Configuración) y, dentro del editor, un listado de los archivos del proyecto activo con iconos diferenciados por tipo de archivo. La barra superior (TopBar) muestra el dispositivo FPGA seleccionado y el selector de idioma.
Prioridad	Alta
Entrada	Interacción del usuario con los elementos de navegación
Salida	Vista correspondiente a la sección o archivo seleccionado
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	La interfaz muestra el contenido solicitado sin pérdida del estado de las demás secciones

4.2.5. RF-05: Seleccionar Dispositivo FPGA

Campo	Descripción
ID	RF-05
Nombre	Seleccionar dispositivo FPGA
Descripción	El sistema debe permitir al usuario seleccionar un modelo de FPGA de referencia a partir de un catálogo de 35 dispositivos de tres fabricantes (Intel/Altera Cyclone IV E, Xilinx Spartan-6 y Artix-7, Lattice iCE40 y ECP5). La información de cada dispositivo se carga desde archivos JSON e incluye: modelo, familia, fabricante, recursos lógicos (LUTs, bloques de memoria, flip-flops, DSPs), especificaciones de reloj (PLLs, frecuencia máxima) e información de E/S.
Prioridad	Media
Entrada	Selección del dispositivo mediante menú desplegable en la barra superior
Salida	Dispositivo activo actualizado en el servicio SelectedFPGA; vista de detalle disponible en la pestaña FPGA
Precondiciones	Los archivos JSON de dispositivos se encuentran disponibles en el directorio Devices
Postcondiciones	El dispositivo seleccionado se utiliza como referencia para el cálculo de utilización de recursos

4.2.6. RF-06: Configurar Parámetros de Simulación

Campo	Descripción
ID	RF-06
Nombre	Configurar parámetros de simulación
Descripción	El sistema debe permitir al usuario definir los parámetros de simulación mediante un diálogo modal que incluye: duración de la simulación, escala temporal (NS, US, MS, S, M, H), frecuencia de reloj en MHz para señales de tipo CLK, y configuración de estímulos por puerto (pares tiempo-valor). Las configuraciones se almacenan en formato JSON en la tabla `AppSettings` de SQLite con extensión `.k-config`.
Prioridad	Media
Entrada	Valores de duración, escala, frecuencia y estímulos ingresados por el usuario
Salida	Objeto SimulationConfig construido por StimulusBuilder con los parámetros convertidos a ticks internos
Precondiciones	Existe un archivo HDL analizado con entidades y puertos identificados
Postcondiciones	La configuración persiste en SQLite y se puede recuperar en sesiones futuras

4.2.7. RF-07: Visualizar Diagnósticos y Recursos

Campo	Descripción
ID	RF-07
Nombre	Visualizar diagnósticos y recursos
Descripción	El sistema debe mostrar la información de las entidades VHDL analizadas (arquitectura, puertos, señales internas) y la utilización estimada de recursos respecto al FPGA seleccionado. La utilización se presenta mediante barras de progreso con codificación por color (verde: uso aceptable, amarillo: advertencia, rojo: capacidad excedida) para LUTs, bloques de memoria, flip-flops y DSPs. El componente de progreso de compilación muestra el avance en tiempo real con mensajes de estado y reporte de errores.
Prioridad	Alta
Entrada	Resultado del análisis de una entidad VHDL y dispositivo FPGA seleccionado
Salida	Panel con información de la entidad y porcentajes de utilización de recursos
Precondiciones	Se ha realizado el análisis de al menos un archivo HDL del proyecto
Postcondiciones	El usuario puede identificar si su diseño cabe en el dispositivo FPGA seleccionado

4.2.8. RF-08: Visualizar Formas de Onda

Campo	Descripción
ID	RF-08
Nombre	Visualizar formas de onda
Descripción	El sistema debe presentar las señales del circuito simulado en un visor de ondas basado en Chart.js (BlazorBootstrap 3.4.0) con gráficos de línea escalonada. Cada señal se muestra en un gráfico independiente con codificación por color según su tipo: verde para señales de reloj, azul para entradas, naranja para salidas y morado para vectores. El visor soporta tipos `std_logic`, `std_logic_vector` e `integer`, e incluye controles de zoom (acercar/alejar), desplazamiento horizontal, etiquetado inteligente del eje temporal (ns, us, ms, s) y botón de exportación a formato VCD.
Prioridad	Alta
Entrada	Historial de señales (SignalHistory) generado por el motor de simulación
Salida	Conjunto de gráficos escalonados con las formas de onda de las señales observadas
Precondiciones	Se ha ejecutado una simulación que produjo datos de señales
Postcondiciones	Las formas de onda se presentan al usuario y pueden ser exportadas en formato VCD

4.2.9. RF-09: Cambiar Idioma de la Interfaz

Campo	Descripción
ID	RF-09
Nombre	Cambiar idioma de la interfaz
Descripción	El sistema debe permitir al usuario seleccionar entre 7 idiomas: español (ES), inglés (EN), francés (FR), alemán (DE), chino simplificado (ZH), japonés (JA) y árabe (AR). El cambio de idioma se realiza de forma reactiva sin recarga de página, mediante el servicio Traductor que carga diccionarios JSON con más de 40 claves de traducción. Para el idioma árabe, la interfaz activa el atributo `dir=rtl"` para soporte de escritura de derecha a izquierda. La preferencia se almacena en SQLite.
Prioridad	Baja
Entrada	Código de idioma seleccionado por el usuario
Salida	Interfaz actualizada con las traducciones del idioma seleccionado
Precondiciones	Los archivos de diccionario JSON se encuentran disponibles en el directorio Dictionaries
Postcondiciones	La preferencia persiste en la tabla `AppSettings` y se aplica automáticamente en sesiones futuras

4.2.10. RF-10: Cambiar Tema Visual

Campo	Descripción
ID	RF-10
Nombre	Cambiar tema visual
Descripción	El sistema debe permitir al usuario alternar entre un tema claro y un tema oscuro. La infraestructura de temas se implementa mediante propiedades CSS personalizadas (`--kmila-primary`, `--kmila-primary-light`, `--kmila-primary-dark`, `--kmila-accent`) definidas en la hoja de estilos global. La preferencia se almacena en SQLite mediante la clave `KEY_THEME` de `AppSettings`.
Prioridad	Baja
Entrada	Selección del tema por el usuario
Salida	Interfaz actualizada con la paleta de colores correspondiente
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	La preferencia persiste en SQLite y se aplica automáticamente en sesiones futuras

4.2.11. RF-11: Persistir Datos Localmente

Campo	Descripción
ID	RF-11
Nombre	Persistir datos localmente
Descripción	El sistema debe almacenar de forma persistente toda la información del usuario mediante SQLite (sqlite-net-pcl 1.9.172). La base de datos contiene tres tablas: `Projects` (metadatos de proyectos), `ProjectFiles` (archivos con respaldo de contenido) y `AppSettings` (pares clave-valor para configuración, idioma, tema y parámetros de simulación). El sistema de eventos `StaticOnValueChanged` notifica a los componentes suscritos cuando un valor de configuración cambia, permitiendo actualizaciones reactivas sin recarga.
Prioridad	Media
Entrada	Datos generados por las operaciones del usuario (proyectos, archivos, configuraciones)
Salida	Registros almacenados en la base de datos SQLite local (`kmila.db`)
Precondiciones	La base de datos ha sido inicializada mediante `InitDBAsync` al inicio de la aplicación
Postcondiciones	Los datos persisten entre sesiones y son recuperables ante cierre inesperado de la aplicación

4.3. Requerimientos No Funcionales

Tabla 4.8: *Requerimientos no funcionales: plataforma y experiencia de usuario.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-01	Multiplataforma	El sistema debe ejecutarse en al menos dos modalidades de despliegue: aplicación nativa mediante .NET MAUI (Android API 24+) y aplicación web mediante ASP.NET Core accesible desde cualquier navegador moderno.	Ejecución verificada en Android (MAUI) y navegador web (ASP.NET Core)
RNF-02	Usabilidad	La interfaz debe ser comprensible para alumnos de ISC que cursan asignaturas de diseño digital sin experiencia previa en simuladores HDL profesionales. La disposición debe seguir convenciones de IDEs.	Un alumno de ISC puede crear un proyecto, abrir un archivo `.vhd` y navegar la interfaz en menos de 10 minutos sin asistencia
RNF-03	Rendimiento	Las operaciones de interfaz deben completarse sin retrasos perceptibles. La carga del catálogo de 34 dispositivos FPGA desde archivos JSON no debe bloquear la interfaz.	Tiempo de respuesta de operaciones de UI <1 segundo
RNF-04	Portabilidad web	La versión web no debe requerir plugins, extensiones ni dependencias externas. Toda la funcionalidad debe ejecutarse directamente en el navegador.	Ejecución completa en Chrome, Firefox y Edge sin software adicional

Tabla 4.9: *Requerimientos no funcionales: internacionalización y confiabilidad.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-05	Internacionalización	La interfaz debe soportar múltiples idiomas con cambio reactivo (sin recarga) y soporte RTL. Los textos deben obtenerse de diccionarios JSON externos.	7 idiomas funcionales (ES, EN, FR, DE, ZH, JA, AR) con RTL verificado en AR

ID	Categoría	Descripción	Métrica / Criterio
RNF-06	Confiabilidad	El sistema debe manejar errores de entrada sin interrumpir la ejecución. El contenido debe almacenarse con respaldo dual (disco + SQLite).	0 cierres inesperados por errores de entrada; recuperación verificada
RNF-07	Mantenibilidad	El código debe organizarse en una biblioteca compartida (Kmila.Shared) consumida por ambos hosts, evitando duplicación. Servicios registrados mediante inyección de dependencias.	Una modificación en Kmila.Shared se refleja simultáneamente en ambas plataformas

Tabla 4.10: *Requerimientos no funcionales: extensibilidad y compatibilidad.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-08	Extensibilidad	El catálogo FPGA debe ser extensible mediante archivos JSON en el directorio Devices, sin requerir recompilación.	Agregar un dispositivo requiere solo crear un archivo JSON
RNF-09	Persistencia	Toda la información del usuario debe persistir entre sesiones mediante SQLite local. Los cambios en configuración deben notificarse mediante eventos reactivos.	Datos persisten tras cierre y reapertura; cambios de idioma/tema sin recarga
RNF-10	Compatibilidad	El sistema debe construirse sobre .NET 9.0 con Blazor.Bootstrap 3.4.0, BlazorMonaco 3.3.0 y sqlite-net-pcl 1.9.172.	Compilación exitosa con las versiones especificadas en `.csproj`

4.4. Historias de Usuario

Las historias de usuario se redactan desde la perspectiva de un alumno de ISC que cursa asignaturas de diseño digital (Fundamentos de Diseño Digital, Diseño de Sistemas Digitales, Arquitectura de Computadoras, Sistemas en Chip). Se organizan en dos grupos: TT1 (interfaz y gestión) y TT2 (simulación y depuración).

4.4.1. Historias de Usuario — TT1

4.4.1.1. HU-01: Crear Proyecto de Práctica

Campo	Descripción
ID	HU-01
RF relacionado	RF-01
Rol	Alumno de ISC
Historia	Como alumno, quiero crear un proyecto nuevo para organizar los archivos VHDL de mi práctica de laboratorio, para tener mi trabajo separado por asignatura y ejercicio.
Criterios de aceptación	▪ Puedo asignar un nombre y descripción al proyecto ▪ Se crea un directorio en mi equipo con la estructura del proyecto ▪ El proyecto aparece en la lista de proyectos recientes

4.4.1.2. HU-02: Escribir Código VHDL

Campo	Descripción
ID	HU-02
RF relacionado	RF-02, RF-03
Rol	Alumno de ISC
Historia	Como alumno, quiero crear un archivo `.vhd` dentro de mi proyecto y escribir código VHDL con resaltado de sintaxis, para desarrollar mi diseño digital con asistencia visual.
Criterios de aceptación	▪ Puedo crear un nuevo archivo `.vhd` desde la interfaz ▪ El editor resalta las palabras reservadas de VHDL (`entity`, `architecture`, `signal`, `process`, etc.) ▪ Puedo ver la línea y columna del cursor en todo momento ▪ Se indica visualmente cuando hay cambios sin guardar

4.4.1.3. HU-03: Abrir Práctica Anterior

Campo	Descripción
ID	HU-03
RF relacionado	RF-01, RF-02
Rol	Alumno de ISC
Historia	Como alumno, quiero abrir un proyecto existente y recuperar los archivos que ya escribí, para continuar una práctica de una sesión anterior.
Criterios de aceptación	▪ La lista de proyectos recientes muestra mis proyectos anteriores con fecha de creación ▪ Al abrir un proyecto, los archivos se cargan desde disco ▪ Si un archivo se perdió del disco, se recupera desde el respaldo en base de datos

4.4.1.4. HU-04: Navegar Archivos del Proyecto

Campo	Descripción
ID	HU-04
RF relacionado	RF-04
Rol	Alumno de ISC
Historia	Como alumno, quiero ver todos los archivos de mi proyecto en un panel lateral y cambiar entre ellos, para trabajar con múltiples módulos VHDL (entidad principal, testbench, componentes).
Criterios de aceptación	▪ Los archivos se listan con iconos diferenciados por tipo ▪ Puedo cambiar de archivo con un clic ▪ La barra lateral muestra las secciones principales de la aplicación (Inicio, Proyectos, Configuración)

4.4.1.5. HU-05: Seleccionar FPGA del Laboratorio

Campo	Descripción
ID	HU-05
RF relacionado	RF-05
Rol	Alumno de ISC
Historia	Como alumno, quiero seleccionar el modelo de FPGA que usamos en el laboratorio (por ejemplo, Cyclone IV EP4CE6 o Artix-7 XC7A35T), para verificar que mi diseño es compatible con los recursos disponibles en la tarjeta.
Criterios de aceptación	▪ Puedo elegir entre dispositivos de Intel/Altera, Xilinx y Lattice desde un menú desplegable ▪ Se muestran las especificaciones del dispositivo (LUTs, memoria, flip-flops, DSPs) ▪ La selección se mantiene mientras trabajo en el proyecto

4.4.1.6. HU-06: Configurar Estímulos de Entrada

Campo	Descripción
ID	HU-06
RF relacionado	RF-06
Rol	Alumno de ISC
Historia	Como alumno, quiero configurar la frecuencia del reloj y los valores de las señales de entrada en distintos momentos de tiempo, para definir el escenario de prueba de mi circuito antes de simularlo.
Criterios de aceptación	▪ Puedo definir la frecuencia del reloj en MHz ▪ Puedo agregar pares tiempo-valor para cada puerto de entrada ▪ Puedo elegir la escala temporal (ns, us, ms, s) ▪ La configuración se guarda y puedo recuperarla después

4.4.1.7. HU-07: Revisar Diagnósticos y Utilización

Campo	Descripción
ID	HU-07
RF relacionado	RF-07
Rol	Alumno de ISC
Historia	Como alumno, quiero ver la información de las entidades de mi diseño (puertos, señales, arquitectura) y el porcentaje de recursos que consume respecto al FPGA seleccionado, para saber si mi diseño cabe en la tarjeta antes de probarlo en el laboratorio.
Criterios de aceptación	▪ Se listan las entidades, puertos y señales internas del diseño ▪ Las barras de utilización muestran el porcentaje de LUTs, memoria, flip-flops y DSPs consumidos ▪ Se indica con color si la utilización es aceptable (verde), cercana al límite (amarillo) o excedida (rojo)

4.4.1.8. HU-08: Ver Formas de Onda

Campo	Descripción
ID	HU-08
RF relacionado	RF-08
Rol	Alumno de ISC
Historia	Como alumno, quiero ver las formas de onda de las señales de mi circuito tras la simulación, para verificar visualmente que mi diseño se comporta como esperaba en la práctica.
Criterios de aceptación	▪ Cada señal se muestra en un gráfico de línea escalonada independiente ▪ Las señales se distinguen por color (verde=reloj, azul=entrada, naranja=salida, morado=vector) ▪ Puedo acercar y alejar el eje temporal ▪ Puedo exportar las formas de onda en formato VCD

4.4.1.9. HU-09: Usar la Aplicación en mi Idioma

Campo	Descripción
ID	HU-09
RF relacionado	RF-09
Rol	Alumno de ISC
Historia	Como alumno, quiero cambiar el idioma de la interfaz al español u otro idioma de mi preferencia, para entender todas las opciones y mensajes de la aplicación.
Criterios de aceptación	▪ Puedo seleccionar entre 7 idiomas disponibles ▪ La interfaz se actualiza inmediatamente sin recargar la página ▪ Mi preferencia se recuerda para la siguiente sesión

4.4.1.10. HU-10: Personalizar el Tema Visual

Campo	Descripción
ID	HU-10
RF relacionado	RF-10
Rol	Alumno de ISC
Historia	Como alumno, quiero alternar entre un tema claro y uno oscuro, para adaptar la interfaz a mi preferencia visual o a las condiciones de iluminación del laboratorio.
Criterios de aceptación	▪ Puedo cambiar entre tema claro y oscuro ▪ El cambio se aplica a toda la interfaz ▪ Mi preferencia se recuerda para la siguiente sesión

4.4.2. Historias de Usuario — TT2

Las siguientes historias de usuario corresponden a funcionalidades del motor de simulación, cuya implementación se contempla en la segunda fase del proyecto. Se enuncian para completitud del análisis.

Tabla 4.11: *Resumen de historias de usuario de TT2.*

ID	Historia resumida	RF relacionado
HU-11	Como alumno, quiero que la aplicación detecte errores de sintaxis en mi código VHDL antes de simular, para corregirlos a tiempo.	RF-12, RF-13
HU-12	Como alumno, quiero ejecutar la simulación de mi circuito y ver cómo se propagan las señales, para verificar mi diseño sin necesidad de la tarjeta FPGA.	RF-14, RF-15
HU-13	Como alumno, quiero avanzar la simulación ciclo por ciclo, para entender exactamente cómo cambian las señales en cada paso de tiempo.	RF-16
HU-14	Como alumno, quiero pausar y reanudar la simulación, para analizar el estado del circuito en un momento específico.	RF-15
HU-15	Como alumno, quiero insertar puntos de interrupción en mi código, para que la simulación se detenga automáticamente en las líneas que quiero depurar.	RF-17

4.5. Reglas de Negocio

Tabla 4.12: *Reglas de negocio: gestión y persistencia.*

ID	Regla	Descripción	Justificación
RN-01	Proyecto obligatorio para simulación	Todo archivo HDL debe pertenecer a un proyecto creado en Kmila antes de poder ser simulado o analizado.	La persistencia en SQLite (tabla `Projects`) requiere un contexto de proyecto para asociar archivos y configuraciones.
RN-02	Un dispositivo FPGA por proyecto	Cada proyecto debe tener asociado exactamente un modelo de dispositivo FPGA del catálogo de 34 modelos disponibles.	Los recursos del dispositivo (LUTs, flip-flops, bloques DSP, frecuencia máxima) determinan las restricciones de síntesis y simulación.
RN-03	Extensiones de archivo permitidas	Solo se permiten archivos con extensión `.vhd` o `.vhdl`. El soporte para otros lenguajes HDL (Verilog, SystemVerilog) queda fuera del alcance del presente proyecto y se considera trabajo futuro.	Evita que el usuario agregue archivos no compatibles con el motor de análisis VHDL.
RN-04	Persistencia dual de archivos	Todo archivo de proyecto se almacena en disco (sistema de archivos local) y simultáneamente en SQLite (tabla `ProjectFiles`, campo `Content`) como respaldo.	Garantiza recuperación ante pérdida del archivo en disco y permite portabilidad del proyecto.

Tabla 4.13: *Reglas de negocio: configuración y simulación.*

ID	Regla	Descripción	Justificación
RN-05	Idioma por defecto	El idioma inicial de la interfaz es español (ES). El usuario puede cambiarlo a cualquiera de los 7 idiomas disponibles desde el menú de configuración.	Orientado al contexto académico de la ESCOM donde el idioma principal es español.
RN-06	Configuración por proyecto	Los parámetros de simulación (frecuencia de reloj, tiempo de ejecución, formato de exportación) se almacenan por proyecto en formato `.k-config`.	Permite que cada proyecto tenga configuraciones independientes sin afectar otros proyectos.
RN-07	Validación previa a simulación (TT2)	Antes de ejecutar la simulación, el código debe pasar un análisis sintáctico completo sin errores.	Previene la ejecución de código malformado que podría generar resultados incorrectos o excepciones en el motor de simulación.
RN-08	Límite de tiempo de simulación (TT2)	La simulación se detiene automáticamente al alcanzar el tiempo máximo configurado en el proyecto.	Evita bucles infinitos y consumo excesivo de recursos en simulaciones mal definidas.

4.6. Análisis de Riesgos

Tabla 4.14: *Análisis de riesgos del proyecto.*

ID	Riesgo	Prob.	Impacto	Estrategia de Mitigación
R-01	Complejidad del análisis sintáctico de VHDL excede lo estimado	Alta	Alto	Limitar el subconjunto soportado de VHDL (entidades, arquitecturas, procesos, señales) y usar gramáticas de referencia del estándar IEEE 1076-2008 [10].
R-02	Rendimiento insuficiente de la simulación en WebAssembly	Media	Alto	Realizar benchmarks tempranos comparando ejecución nativa (MAUI) vs. WebAssembly (Blazor). Optimizar estructuras de datos del motor de simulación en TT2.
R-03	Incompatibilidad visual entre plataformas MAUI y Blazor	Media	Medio	Concentrar toda la UI en Kmila.Shared como componentes Razor reutilizables. Probar en Android y Web desde el inicio del desarrollo.
R-04	Retraso en el cronograma por alcance del motor de simulación	Media	Alto	Definir un subconjunto mínimo viable de instrucciones VHDL para TT2. Priorizar la simulación de entidades simples (compuertas, flip-flops, contadores).
R-05	Pérdida de datos del usuario por fallos en persistencia dual	Baja	Alto	Implementar escritura transaccional en SQLite y verificar integridad disco-BD al abrir un proyecto. Incluir pruebas unitarias de persistencia.
R-06	Obsolescencia de dependencias (.NET 9, BlazorMonaco 3.3.0)	Baja	Medio	Monitorear el ciclo de soporte de .NET 9 (soporte estándar hasta mayo 2026). Encapsular dependencias externas tras interfaces para facilitar actualización.
R-07	Curva de aprendizaje del usuario objetivo	Media	Medio	Diseñar la interfaz siguiendo convenciones de IDEs conocidos (VS Code). Incluir tooltips, mensajes de error descriptivos y documentación integrada.

Capítulo 5

Diseño del Sistema

5.1. Arquitectura del Sistema

La arquitectura de Kmila se describe mediante el modelo C4 (Context, Containers, Components, Code) propuesto por Simon Brown, que permite representar el sistema en cuatro niveles de abstracción progresiva. Cada nivel aumenta el detalle técnico, facilitando la comprensión tanto para directores del proyecto como para desarrolladores que deseen extender la plataforma.

5.1.1. Nivel 1 — Diagrama de Contexto

El nivel de contexto muestra a Kmila como una caja negra y sus interacciones con actores y sistemas externos. Kmila opera de forma completamente local: no requiere conexión a internet ni depende de servicios en la nube. El único actor es el estudiante de ISC que interactúa con la aplicación para editar código VHDL, configurar simulaciones y visualizar formas de onda.

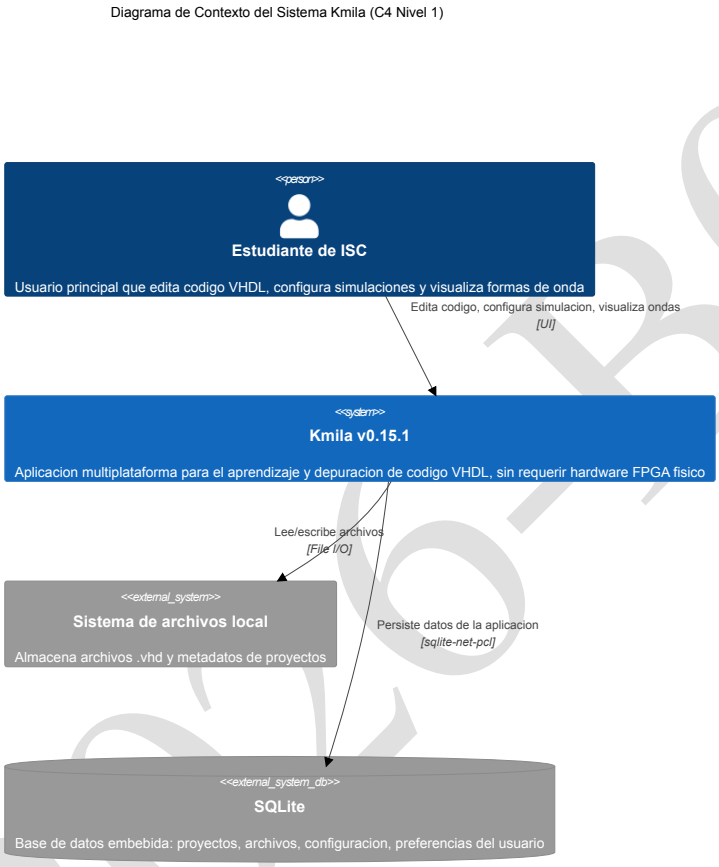

Figura 5.1: *Diagrama de contexto del sistema Kmila (C4 Nivel 1). Nota: se omiten acentos y caracteres especiales en los identificadores del diagrama por compatibilidad con el renderizador Mermaid.*

El estudiante interactúa con Kmila a través de la interfaz gráfica unificada. La aplicación persiste los datos del usuario en dos destinos complementarios: el sistema de archivos local (donde residen los archivos `.vhd` del proyecto) y una base de datos SQLite embebida (donde se almacenan metadatos de proyectos, respaldos de archivos, configuraciones de simulación e idioma seleccionado).

5.1.2. Nivel 2 — Diagrama de Contenedores

El nivel de contenedores descompone Kmila en sus proyectos ejecutables y bibliotecas. La solución (`Kmila.sln`) se compone de siete proyectos .NET organizados en dos capas: dos hosts de presentación que consumen una biblioteca compartida, y cuatro módulos de backend que implementan el motor de análisis y simulación.

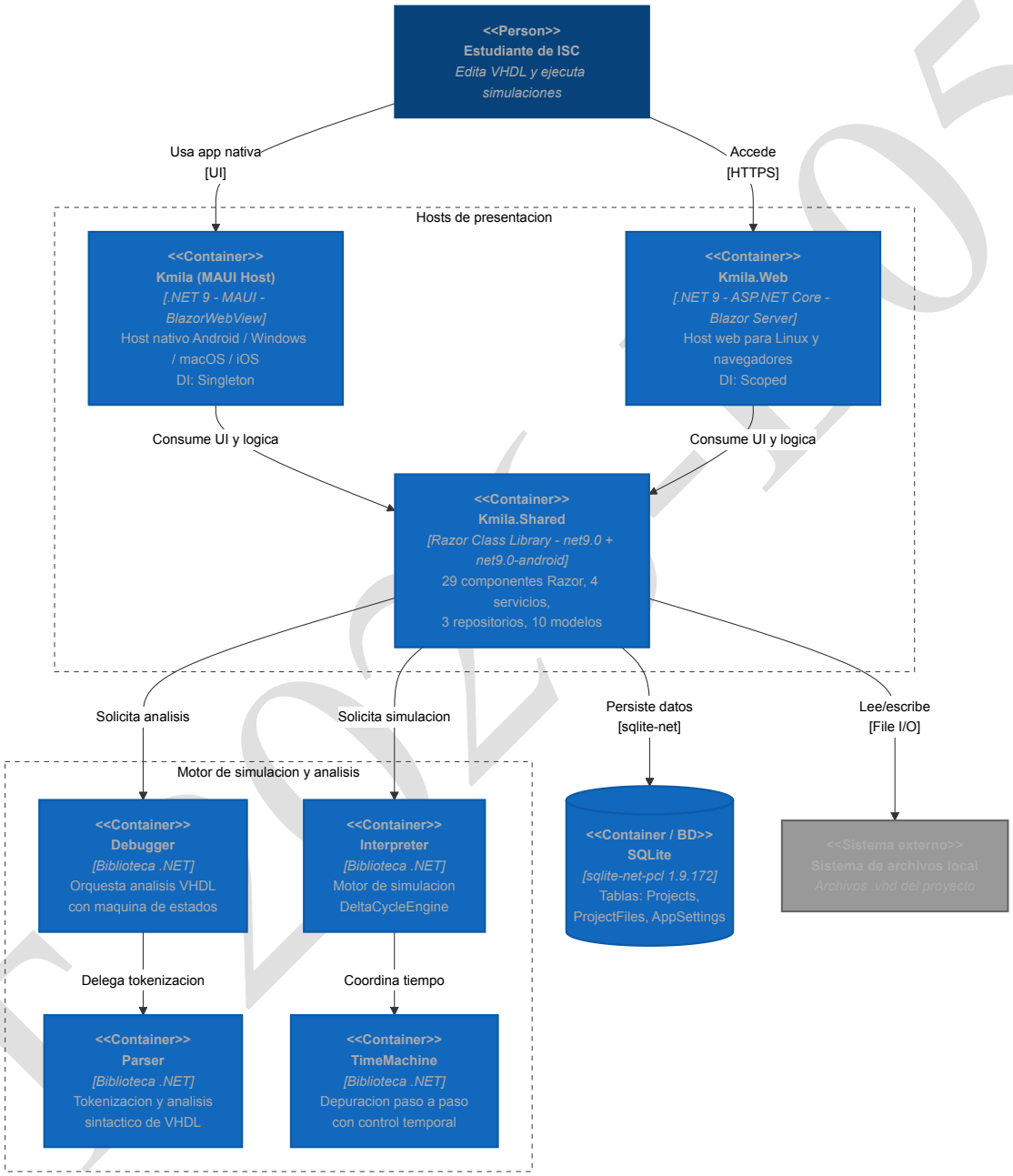

Figura 5.2: *Diagrama de contenedores del sistema Kmila (C4 Nivel 2). Nota: se omiten acentos y caracteres especiales en los identificadores del diagrama por compatibilidad con el renderizador Mermaid de Typora.*

Tabla 5.1: *Descripción de contenedores del sistema.*

Contenedor	Tecnología	Responsabilidad
Kmila (MAUI)	.NET 9, MAUI, BlazorWebView	Host nativo para Android, Windows, macOS e iOS. Registra servicios como Singleton. Carga recursos FPGA desde `FileSystem.OpenAppPackageFileAsync`.
Kmila.Web	.NET 9, ASP.NET Core, Blazor Server	Host web para Linux y navegadores. Registra servicios como Scoped. Sirve recursos FPGA desde `_content/Kmila.Shared/`.
Kmila.Shared	Razor Class Library (net9.0 + net9.0-android)	Contiene 29 componentes Razor, 4 servicios, 3 repositorios y 10 modelos. Concentra toda la lógica de UI y negocio.
Debugger	Biblioteca de clases .NET	Orquesta el análisis de archivos VHDL: extrae entidades y arquitecturas mediante una máquina de estados de 3 fases.
Parser	Biblioteca de clases .NET	Tokenización y análisis sintáctico de código VHDL. Genera tokens a partir del código fuente.
Interpreter	Biblioteca de clases .NET	Motor de simulación basado en `DeltaCycleEngine`. Ejecuta ciclos delta, gestiona señales y genera `SignalHistory` para visualización.
TimeMachine	Biblioteca de clases .NET	Framework de depuración paso a paso. Permite avanzar la simulación ciclo por ciclo con control temporal.
SQLite	sqlite-net-pcl 1.9.172	Base de datos embebida con 3 tablas: `Projects` (metadatos), `ProjectFiles` (respaldo de contenido), `AppSettings` (configuración clave-valor).

5.1.3. Nivel 3 — Diagrama de Componentes

El diagrama de componentes detalla la estructura interna de Kmila.Shared, que concentra la lógica de la aplicación. Los componentes se organizan en cuatro capas: páginas (interfaz de usuario), servicios (lógica de negocio), repositorios (acceso a datos) y datos (persistencia).

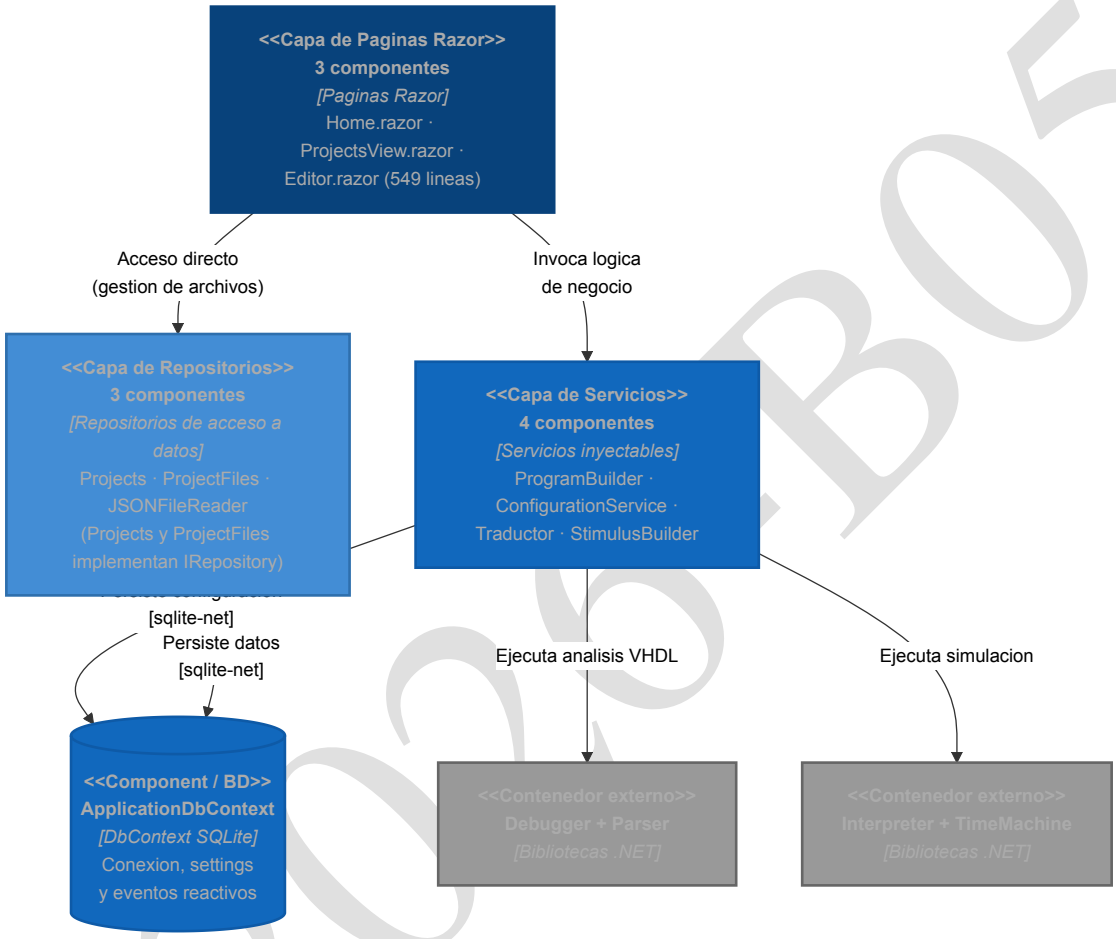

Figura 5.3: *Diagrama de componentes de Kmila.Shared (C4 Nivel 3). Nota: se omiten acentos y caracteres especiales en los identificadores del diagrama por compatibilidad con el renderizador Mermaid de Typora.*

La página `Editor.razor` (549 líneas) constituye el componente central de la aplicación: integra el editor Monaco, el explorador de archivos (`FilesCard`), el selector de dispositivos FPGA (`FPGATabContent`), el panel de configuración de simulación (`SimulationParametersModal`), el visualizador de formas de onda (`WaveformViewer`) y los controles de compilación (`ButtonCollapse`). El servicio `ProgramBuilder` orquesta el flujo de compilación y simulación: invoca al módulo `Debugger` para el análisis sintáctico y al `Interpreter` para la ejecución de la simulación, delegando al `StimulusBuilder` la conversión de los parámetros de la interfaz al formato `SimulationConfig` requerido por el motor.

A continuación se presentan las vistas segmentadas por capa para facilitar el análisis individual de cada grupo de componentes.

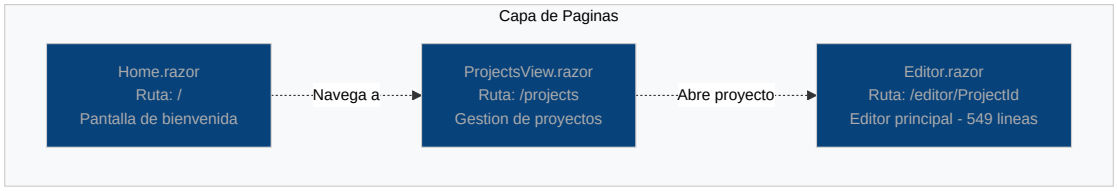

Figura 5.4: *Vista segmentada: capa de paginas. Las 3 paginas Razor definen las rutas principales de la aplicacion. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

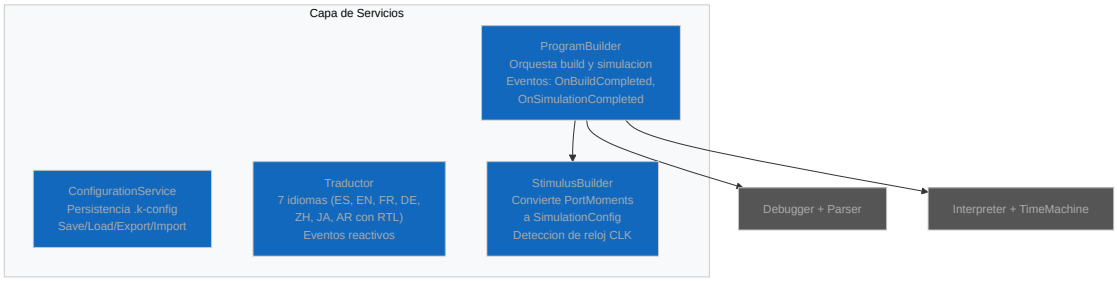

Figura 5.5: *Vista segmentada: capa de servicios. ProgramBuilder es el orquestador central que conecta la UI con los modulos de backend. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

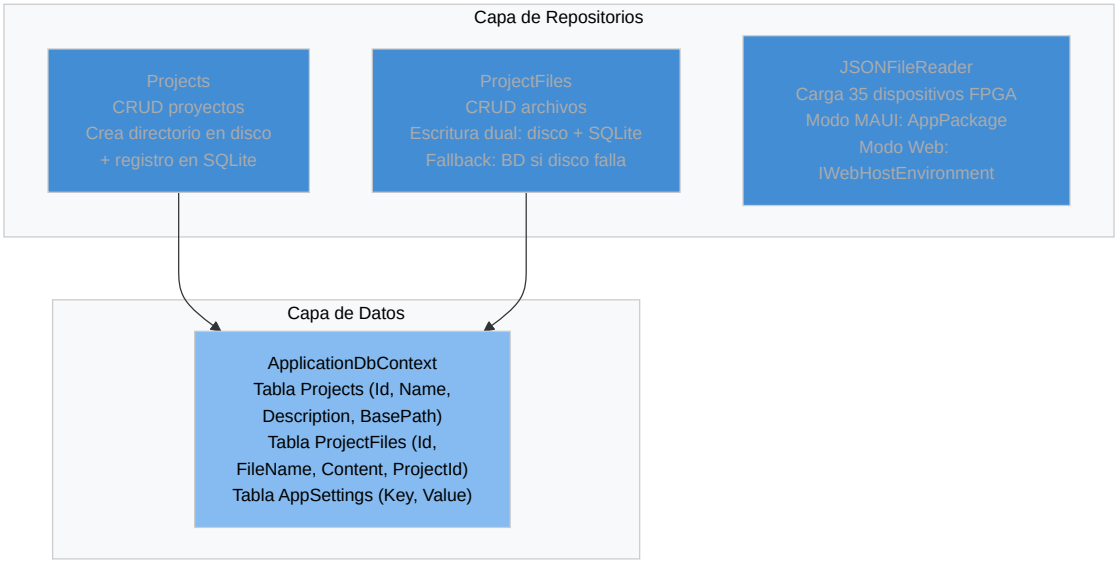

Figura 5.6: *Vista segmentada: capas de repositorios y datos. La persistencia dual (disco + SQLite) garantiza la recuperacion de archivos ante fallos. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.1.4. Diagrama de Despliegue

El diagrama de despliegue muestra cómo se distribuye Kmila en las distintas plataformas objetivo. Ambos hosts empaquetan Kmila.Shared dentro de su binario, pero difieren en el mecanismo de carga de recursos y el ciclo de vida de inyección de dependencias.

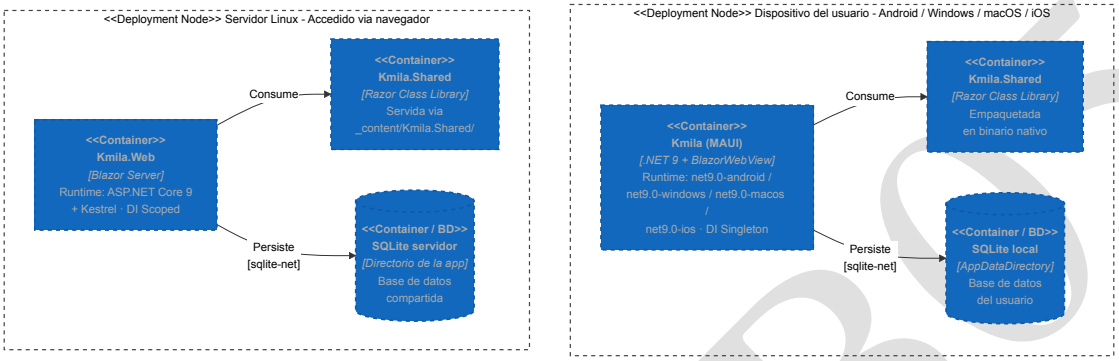

Figura 5.7: *Diagrama de despliegue del sistema Kmila (C4 Nivel 4). Nota: se omiten acentos y caracteres especiales en los identificadores del diagrama por compatibilidad con el renderizador Mermaid de Typora.*

Tabla 5.2: *Mecanismos de despliegue por plataforma.*

Plataforma	Mecanismo de despliegue	Ciclo de vida DI	Carga de recursos FPGA
Android	APK via .NET MAUI (min API 24)	Singleton	`FileSystem.OpenAppPackageFileAsync`
Windows	MSIX o publicación directa	Singleton	`FileSystem.OpenAppPackageFileAsync`
macOS	App Bundle	Singleton	`FileSystem.OpenAppPackageFileAsync`
iOS	App Bundle	Singleton	`FileSystem.OpenAppPackageFileAsync`
Linux / Web	ASP.NET Core + Blazor Server	Scoped	`IWebHostEnvironment` + `_content/Kmila.Shared/Devices/`

5.2. Diagrama de Clases

El diagrama de clases describe las clases principales del sistema, sus atributos, métodos y relaciones. Debido a la extensión del sistema (10 modelos, 4 servicios, 3 repositorios y 4 módulos de backend), se presenta primero un diagrama general con las relaciones entre clases y posteriormente vistas segmentadas por módulo.

5.2.1. Diagrama General

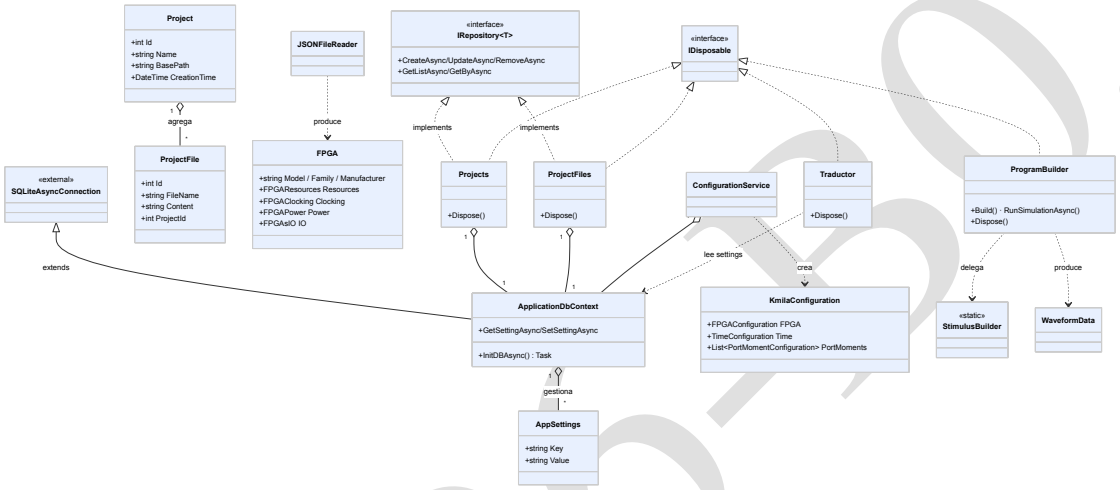

Figura 5.8: *Diagrama general de clases del sistema Kmila. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.2.2. Modelos de Datos

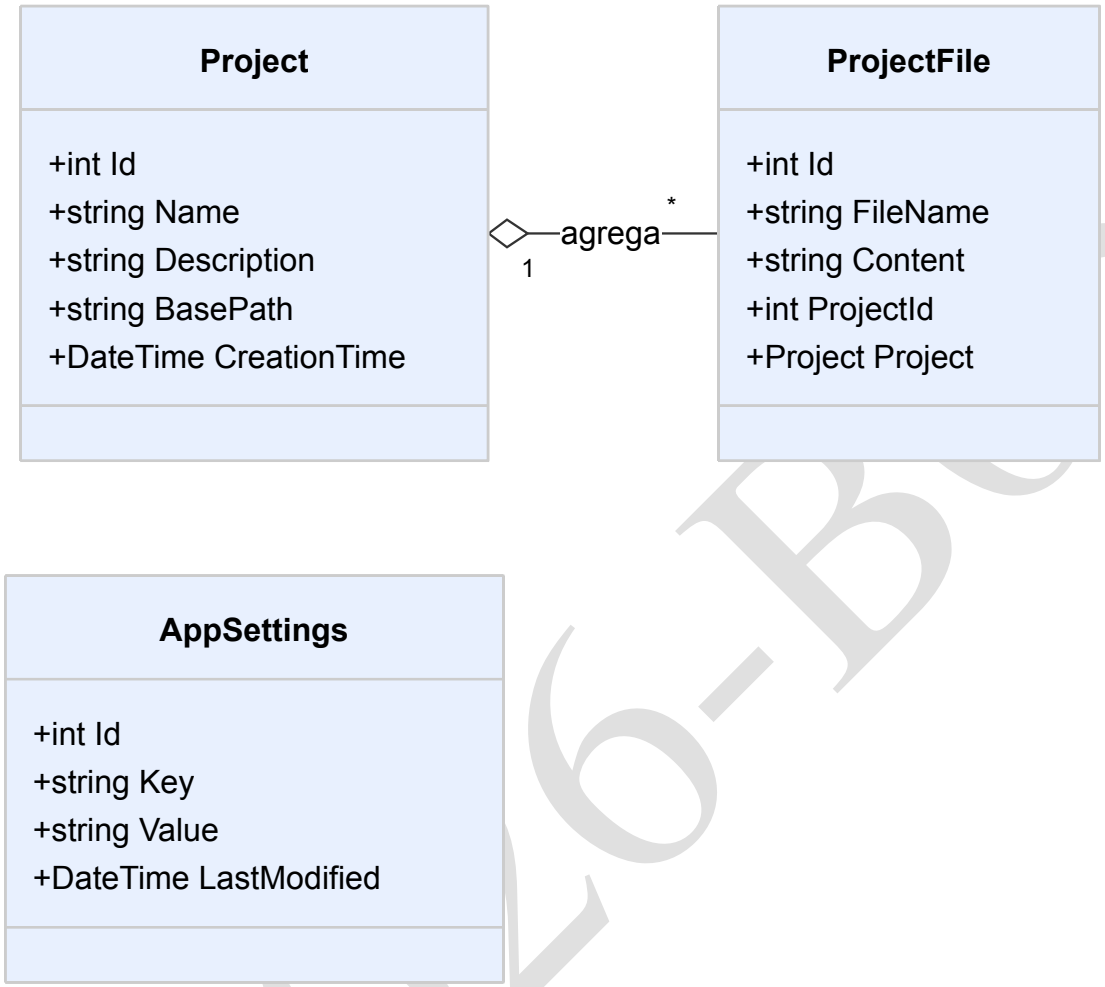

Figura 5.9: *Modelos de persistencia. Project y ProjectFile representan la persistencia en SQLite; AppSettings almacena la configuracion clave-valor del usuario. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

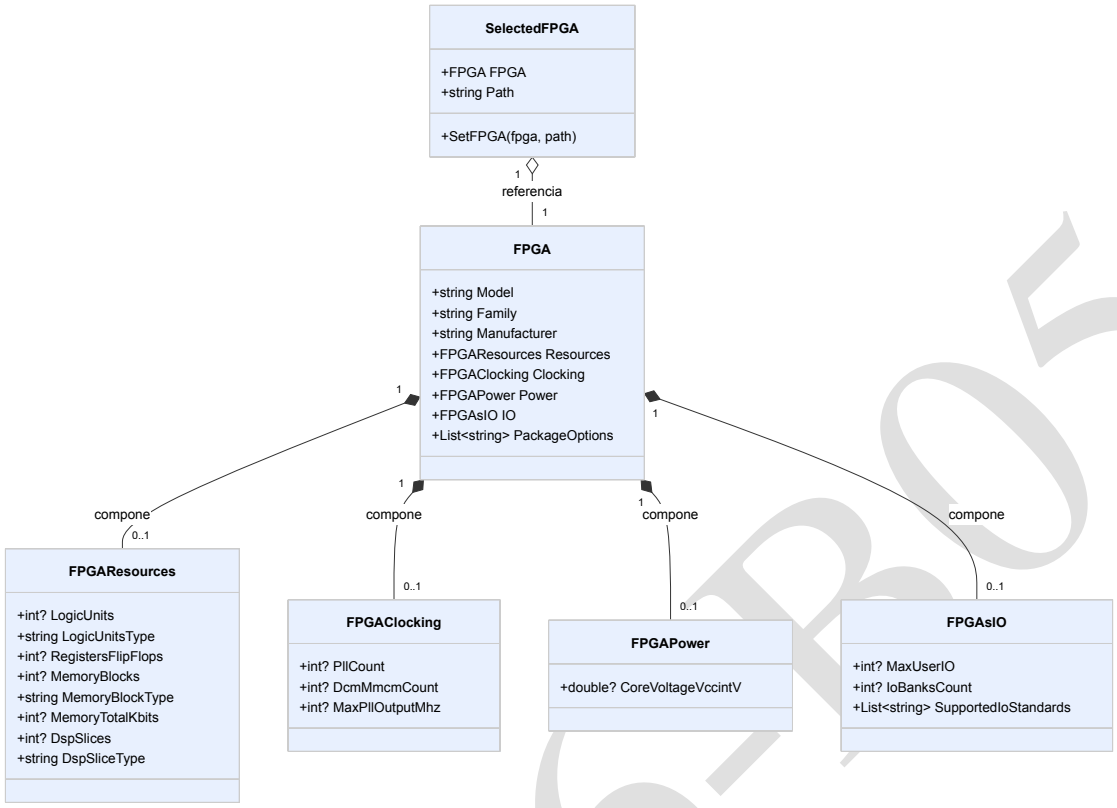

Figura 5.10: *Modelo del catalogo FPGA. FPGA modela las especificaciones de los 35 dispositivos; FPGAResources y FPGAClocking descomponen los recursos logicos y de reloj. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

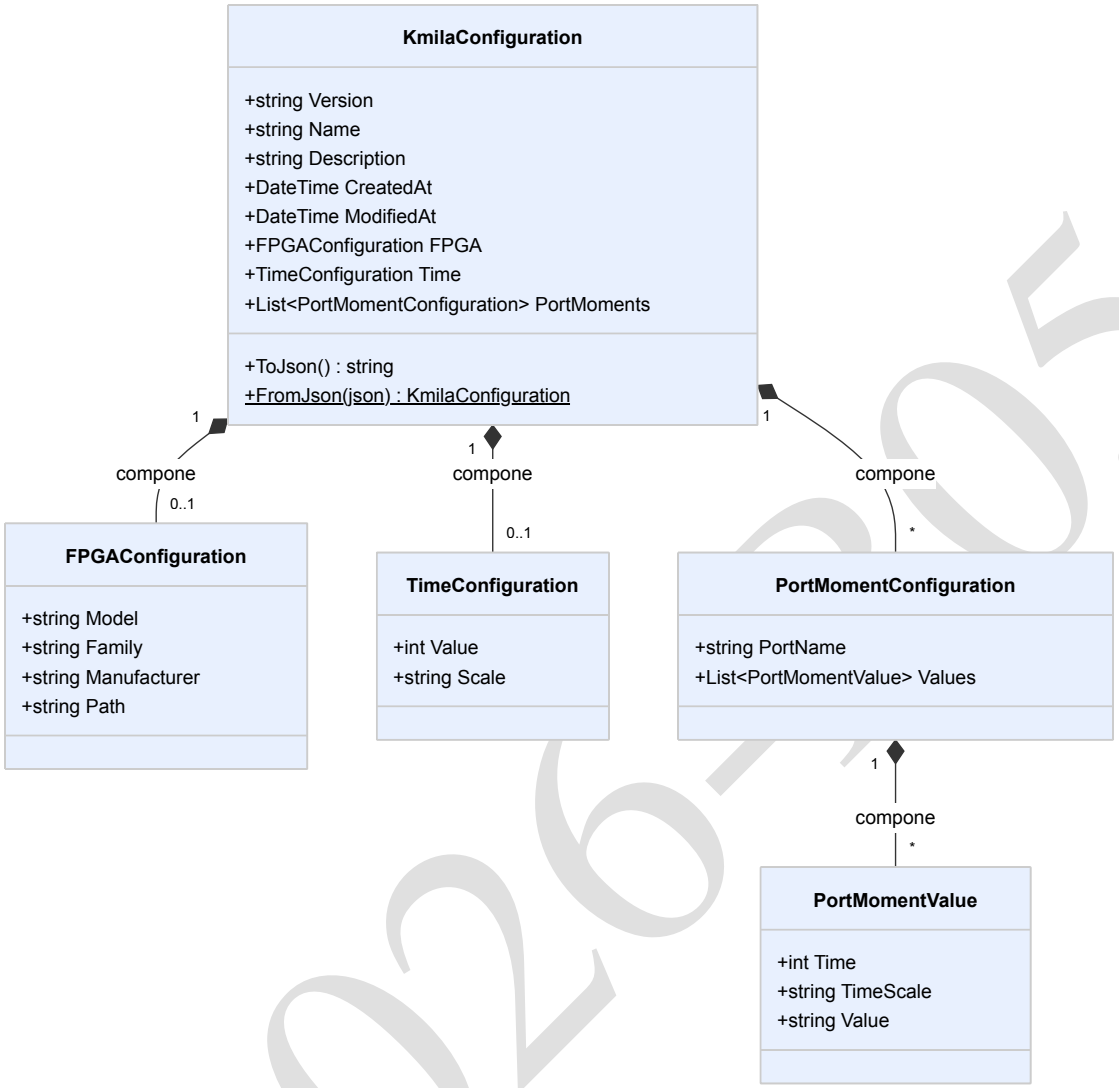

Figura 5.11: *Modelo de configuracion de proyecto. KmilaConfiguration define el formato .k-config; cada PortMoment especifica los estimulos de un puerto. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.2.3. Servicios y Repositorios

El siguiente diagrama muestra la vista general de las capas de servicios y repositorios, seguido de vistas particulares de cada clase.

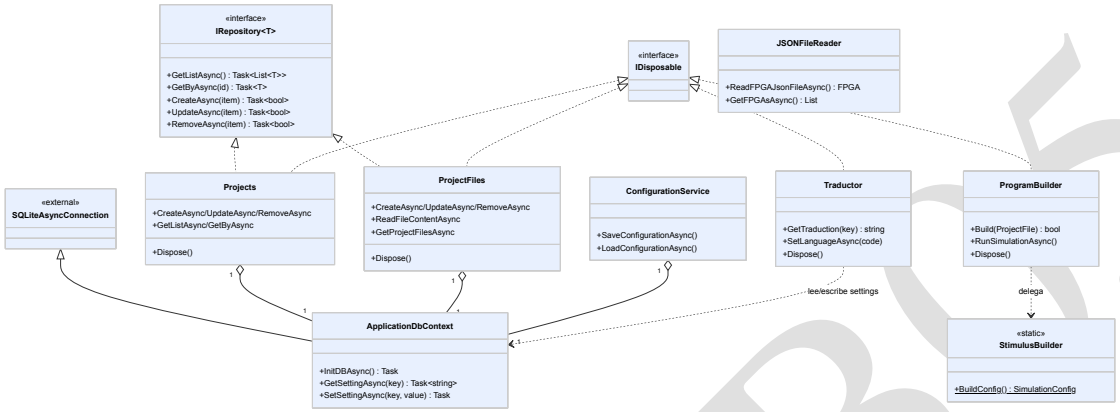

Figura 5.12: *Vista general: servicios y repositorios. Todos los repositorios y servicios de persistencia dependen de ApplicationDbContext. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

A continuación se presenta cada clase en detalle.

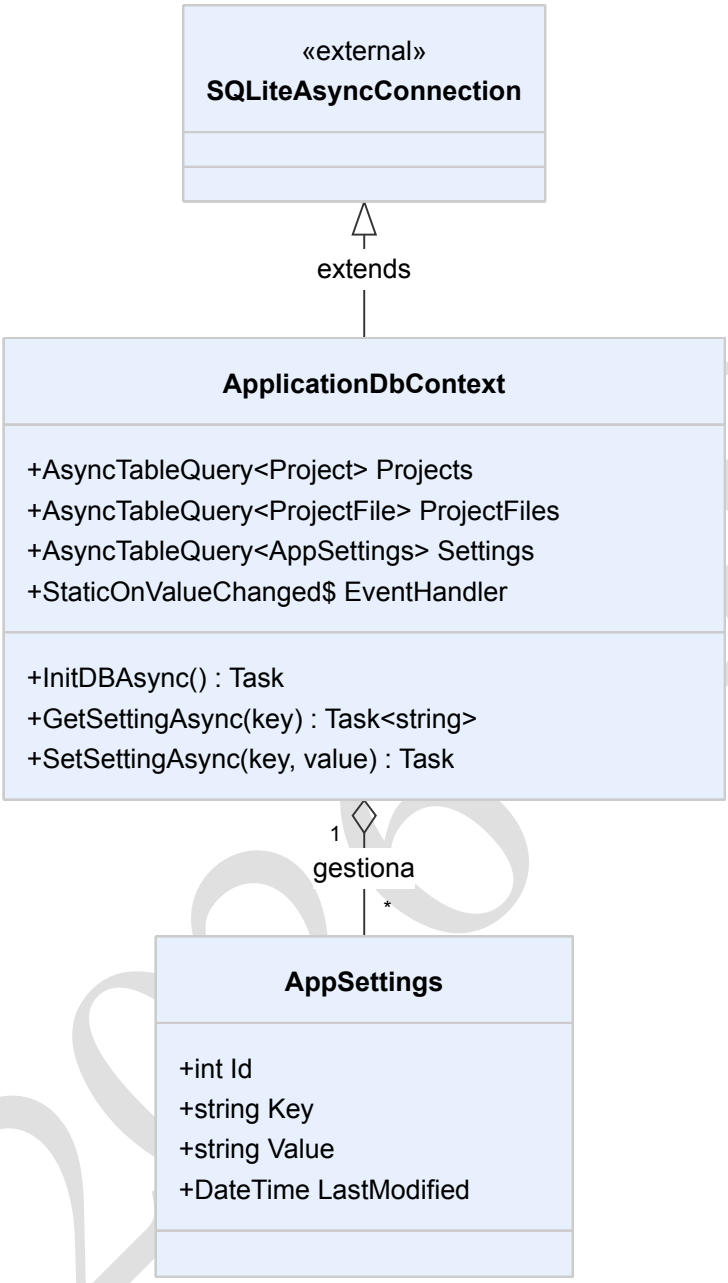

Figura 5.13: *Detalle: ApplicationDbContext. Gestiona la conexion SQLite y expone un evento estatico para notificar cambios en configuracion. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

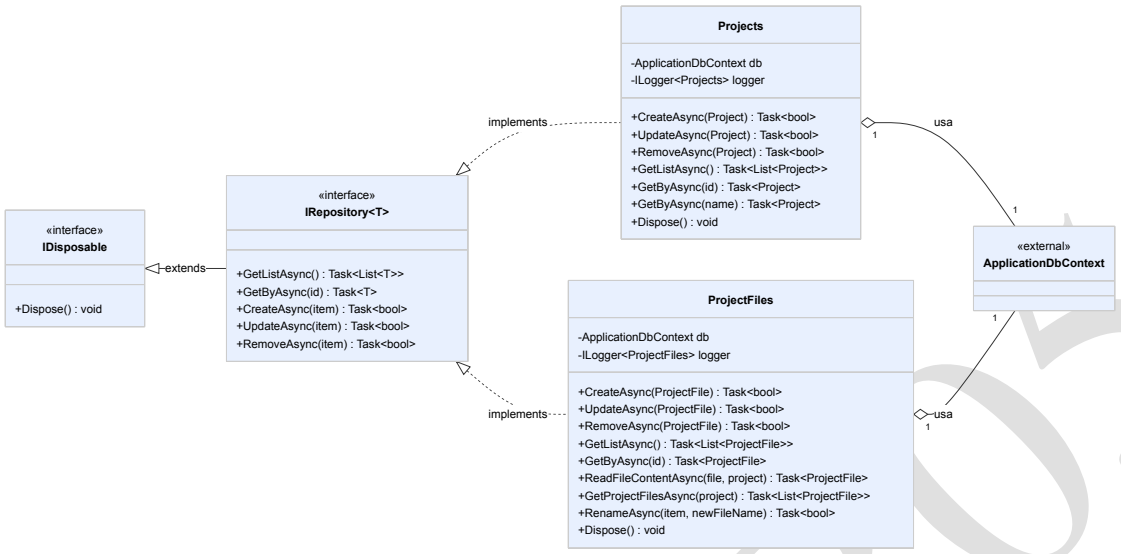

Figura 5.14: *Detalle: repositorios Projects y ProjectFiles. Ambos implementan IRepository y gestionan la sincronizacion entre disco y base de datos. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

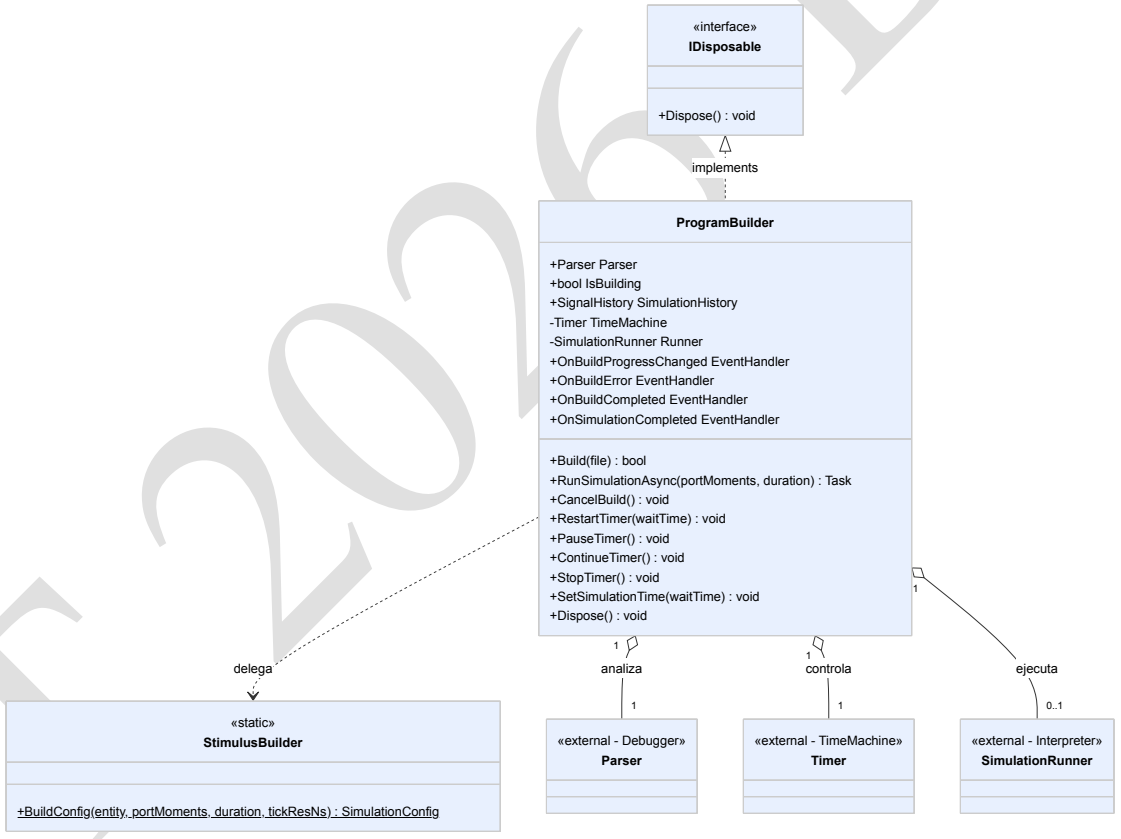

Figura 5.15: *Detalle: ProgramBuilder y StimulusBuilder. ProgramBuilder orquesta compilacion y simulacion; StimulusBuilder convierte la configuracion de la UI al formato SimulationConfig del motor. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

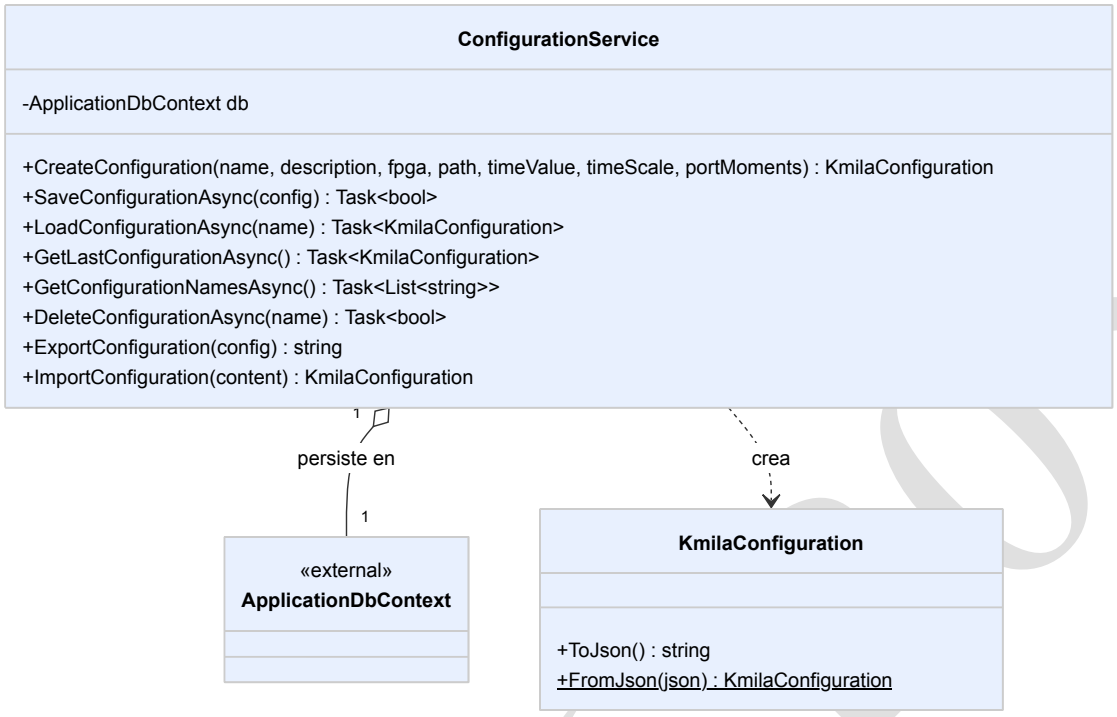

Figura 5.16: *Detalle: ConfigurationService. Gestiona la persistencia de configuraciones de simulacion en formato .k-config (JSON serializado en SQLite). Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

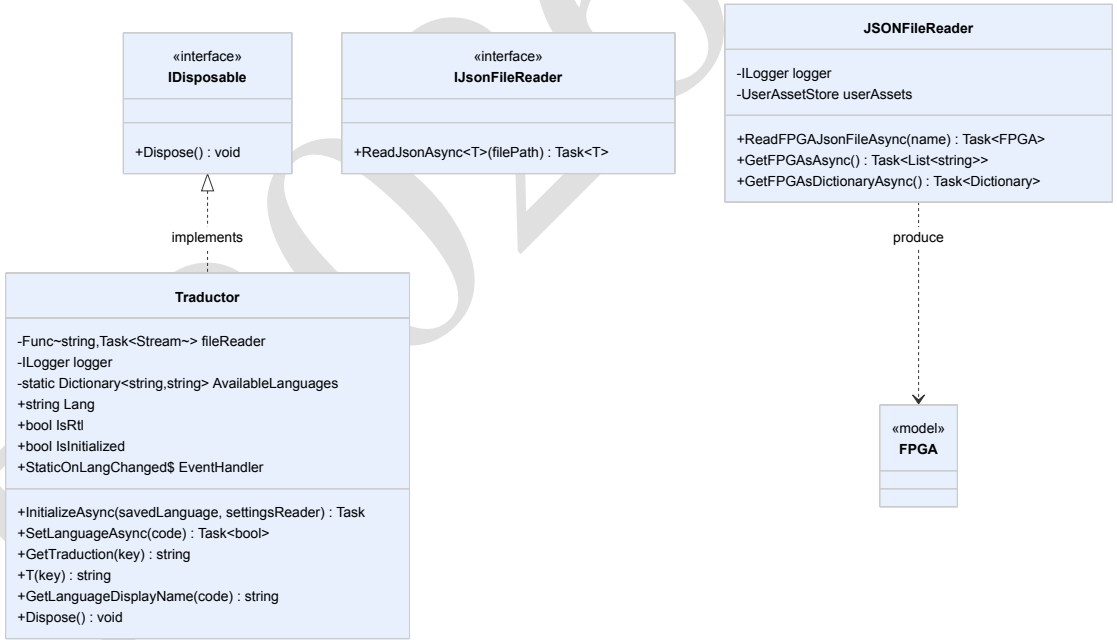

Figura 5.17: *Detalle: Traductor y JSONFileReader. Traductor gestiona la internacionalizacion con eventos reactivos; JSONFileReader carga el catalogo de 35 dispositivos FPGA con estrategias distintas segun la plataforma. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.2.4. Motor de Simulación

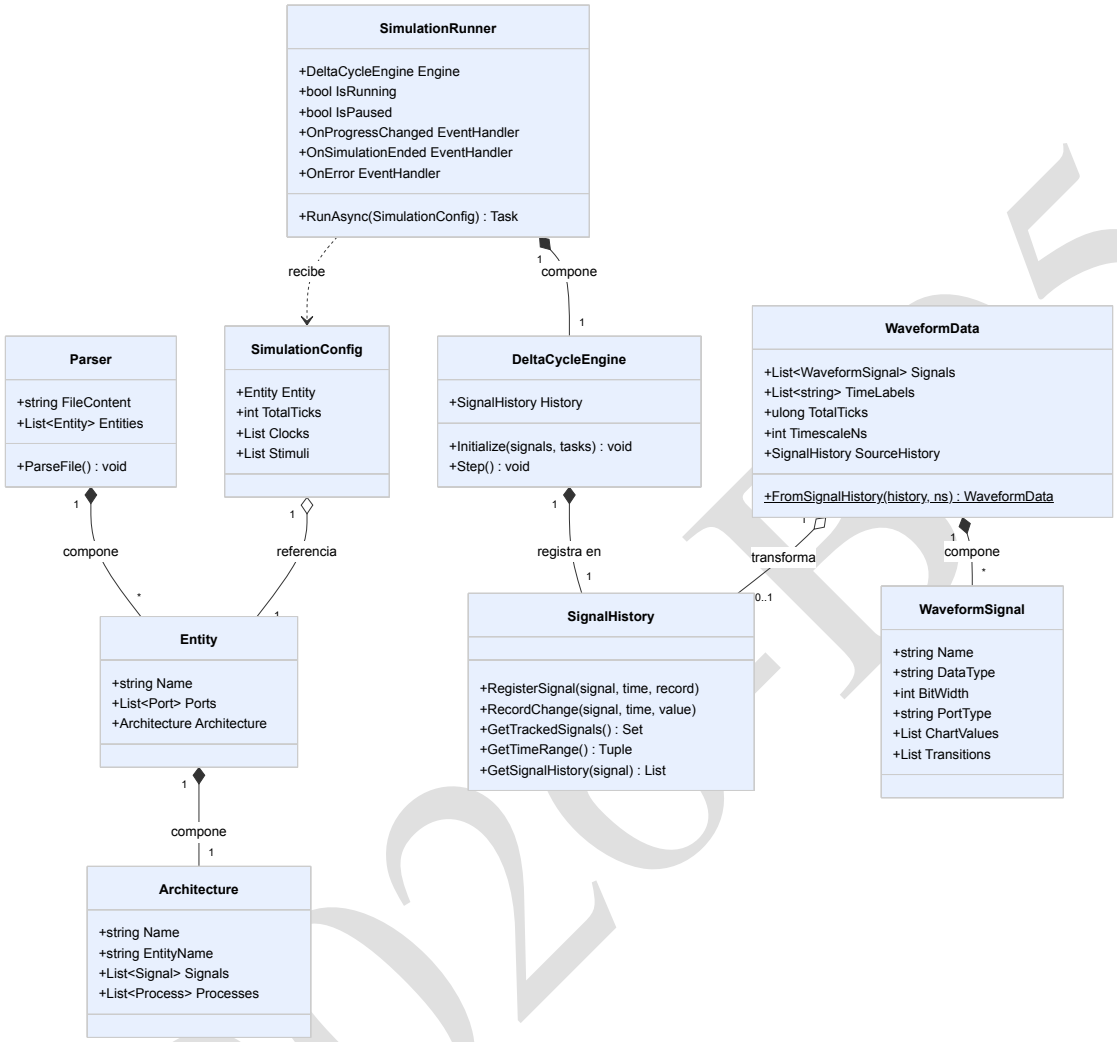

Figura 5.18: *Vista segmentada: motor de simulacion. El Parser extrae entidades y arquitecturas del codigo VHDL. SimulationRunner ejecuta la simulacion mediante DeltaCycleEngine, que registra los cambios de senales en SignalHistory. WaveformData transforma el historial en un formato compatible con Chart.js. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.3. Casos de Uso

Los casos de uso se organizan en dos grupos correspondientes a las fases del proyecto. El siguiente diagrama presenta los correspondientes a TT1 (interfaz y gestión).

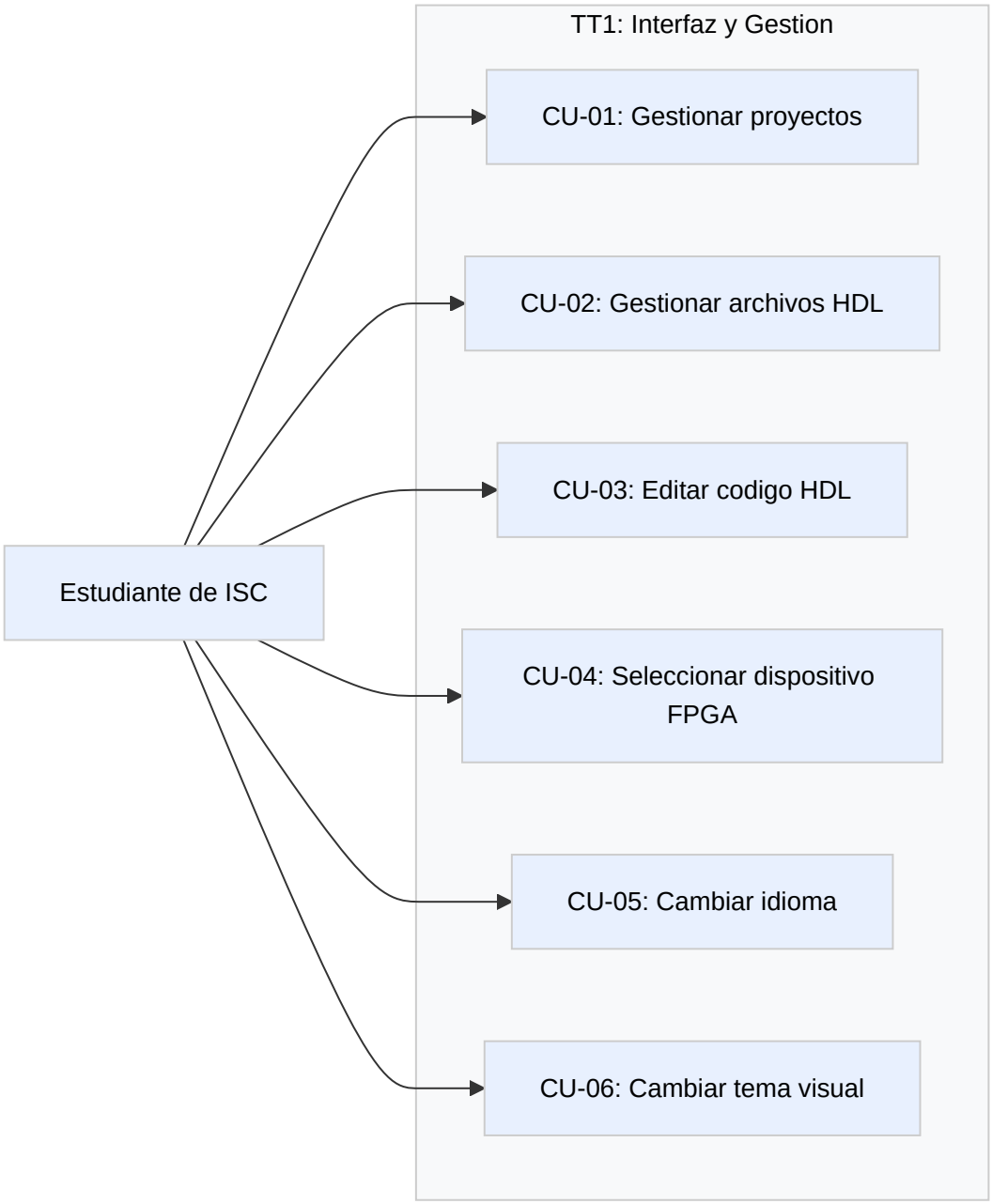

Figura 5.19: *Diagrama de casos de uso de TT1: interfaz y gestion. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

El siguiente diagrama presenta los casos de uso de TT2 (motor de simulación).

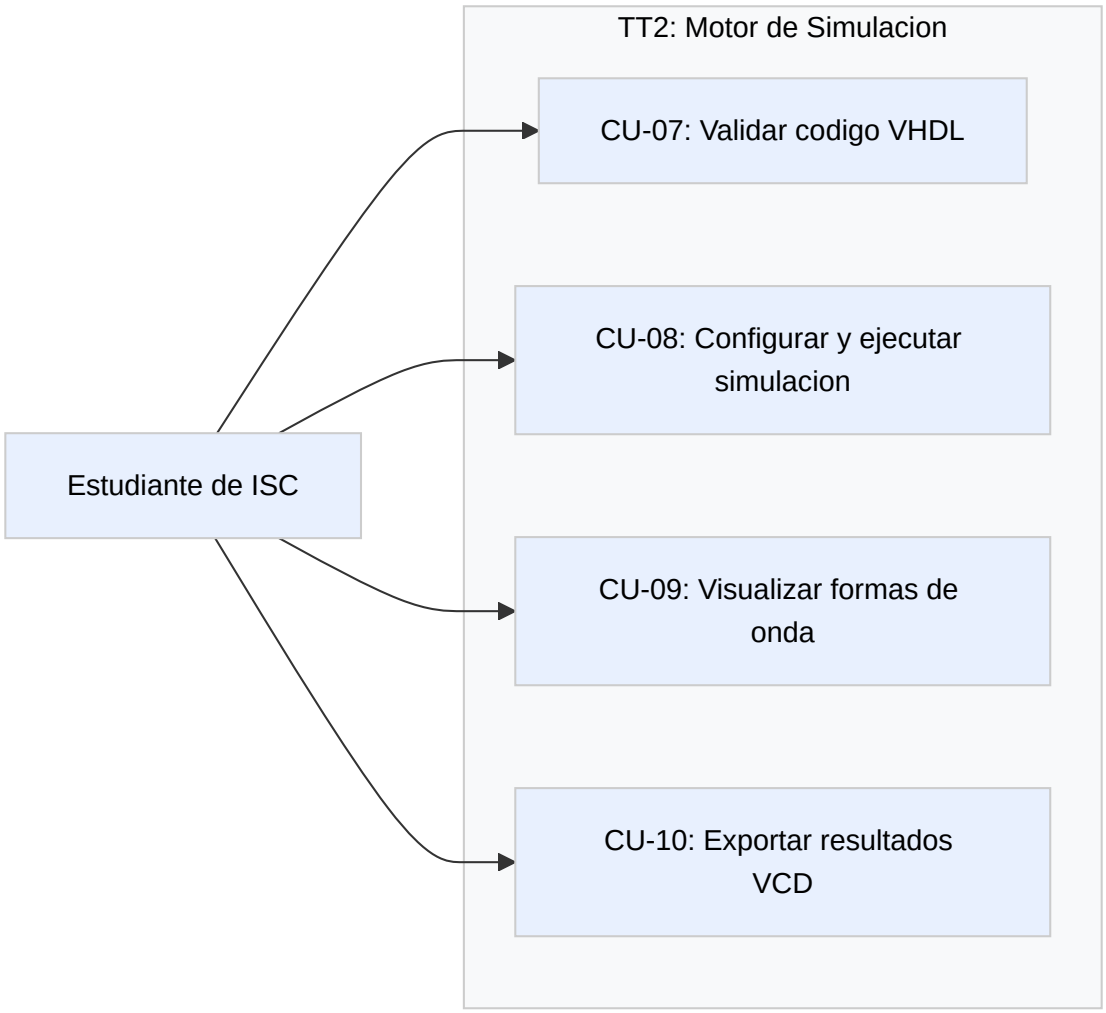

Figura 5.20: *Diagrama de casos de uso de TT2: motor de simulacion. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.3.1. CU-01 y CU-02: Gestión de Proyectos y Archivos HDL

CU-01: Gestionar Proyectos.

Campo	Descripción
ID	CU-01
Actor	Estudiante
RF relacionado	RF-01
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario navega a la vista de proyectos (`/projects`). 2. El usuario presiona "Crear proyecto". 3. El sistema muestra el modal `ProjectDetailsModal` solicitando nombre, descripción y ruta base. 4. El usuario ingresa los datos y confirma. 5. El repositorio `Projects` valida que no exista un proyecto con el mismo nombre y ruta. 6. El sistema crea el directorio en disco y registra el proyecto en SQLite. 7. El proyecto aparece en la lista de proyectos.
Flujo alternativo	5a. Ya existe un proyecto con el mismo nombre y ruta: el sistema notifica el error y no crea el proyecto.
Postcondición	El proyecto queda registrado en la base de datos con su directorio creado en el sistema de archivos.

CU-02: Gestionar Archivos HDL.

Campo	Descripción
ID	CU-02
Actor	Estudiante
RF relacionado	RF-02
Precondición	El usuario ha abierto un proyecto existente
Flujo principal	1. El usuario presiona "Nuevo archivo" en el panel lateral. 2. El sistema solicita el nombre del archivo. 3. El usuario ingresa un nombre con extensión `.vhd` o `.vhdl`. 4. El repositorio `ProjectFiles` crea el archivo en disco y registra una entrada en SQLite con el contenido inicial. 5. El archivo aparece en el listado de archivos del proyecto (`FilesCard`).
Flujo alternativo	3a. El usuario intenta crear un archivo con extensión no soportada: el sistema rechaza la operación. 6a. Al abrir un archivo, si el archivo en disco no existe pero sí en SQLite, el sistema recupera el contenido desde la base de datos.
Postcondición	El archivo persiste en disco y en SQLite (persistencia dual).

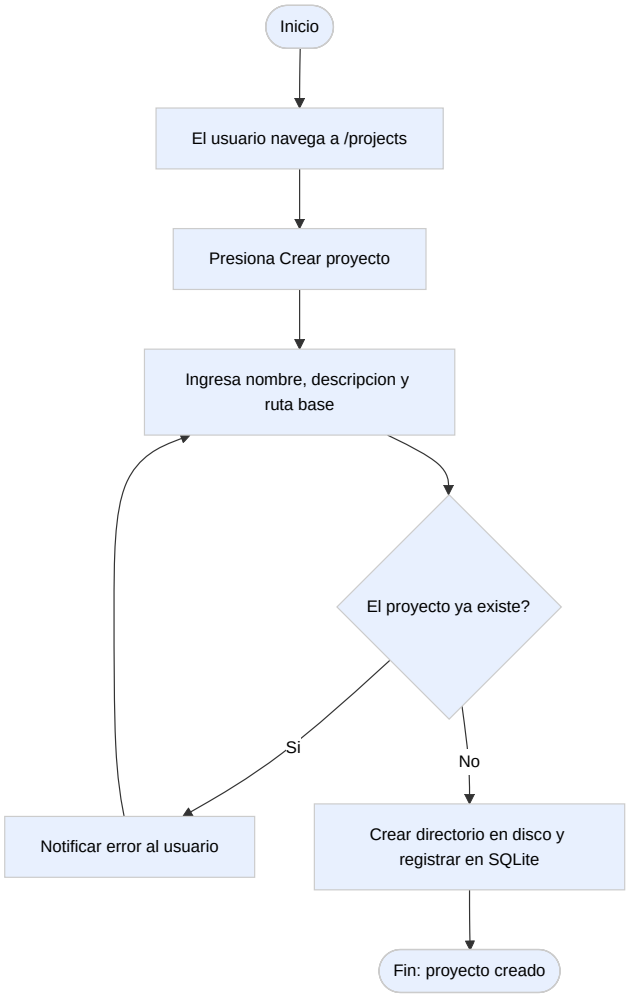

Figura 5.21: *Diagrama de flujo del CU-01: Gestionar Proyectos.*

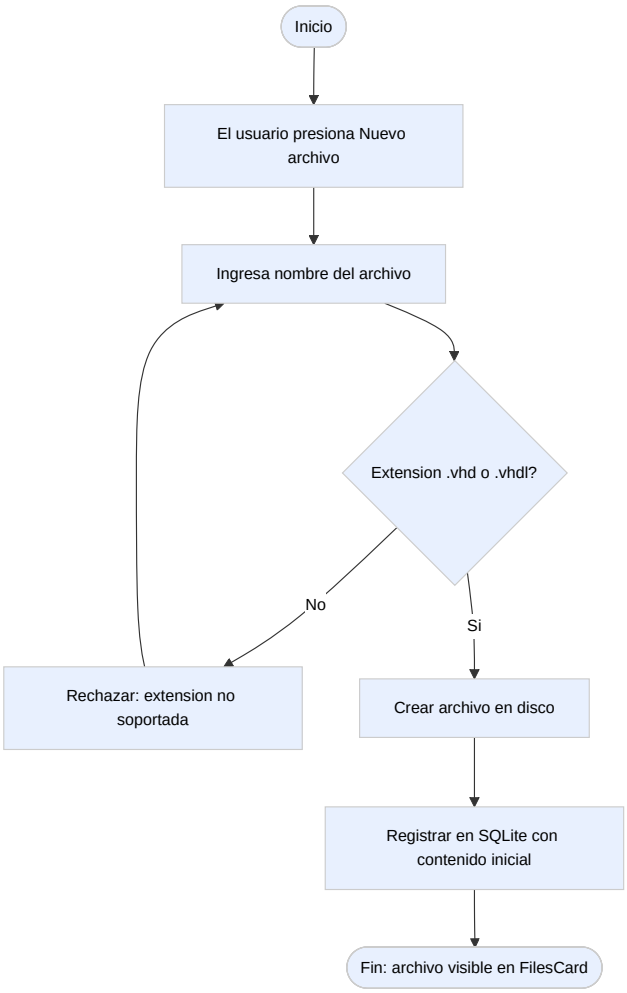

Figura 5.22: *Diagrama de flujo del CU-02: Gestionar Archivos HDL.*

5.3.2. CU-03 y CU-04: Edición de Código y Selección de Dispositivo FPGA

CU-03: Editar Código HDL.

Campo	Descripción
ID	CU-03
Actor	Estudiante
RF relacionado	RF-03
Precondición	Existe al menos un archivo abierto en el proyecto activo
Flujo principal	1. El usuario selecciona un archivo `.vhd` desde `FilesCard`. 2. El sistema carga el contenido en el editor Monaco (`StandaloneCodeEditor`) con resaltado de sintaxis VHDL. 3. El usuario edita el código. 4. El sistema actualiza el indicador de cambios sin guardar y la posición del cursor (línea y columna). 5. El usuario presiona Ctrl+S o el botón “Guardar”. 6. El repositorio `ProjectFiles` escribe el contenido en disco y actualiza el respaldo en SQLite.
Flujo alternativo	2a. Si el archivo tiene extensión `.md`, el editor muestra vista dividida con previsualización Markdown.
Postcondición	El contenido editado persiste en disco y base de datos; el indicador de cambios sin guardar se desactiva.

CU-04: Seleccionar Dispositivo FPGA.

Campo	Descripción
ID	CU-04
Actor	Estudiante
RF relacionado	RF-05
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario selecciona un dispositivo FPGA desde el menú desplegable en la barra superior. 2. El servicio `JSONFileReader` carga el archivo JSON correspondiente al dispositivo. 3. El servicio `SelectedFPGA` actualiza el dispositivo activo. 4. La pestaña FPGA (`FPGATabContent`) muestra las especificaciones del dispositivo: LUTs, bloques de memoria, flip-flops, DSPs, PLLs y frecuencia máxima.
Flujo alternativo	2a. El archivo JSON del dispositivo no se encuentra: el sistema notifica el error.
Postcondición	El dispositivo seleccionado se utiliza como referencia para el cálculo de utilización de recursos.

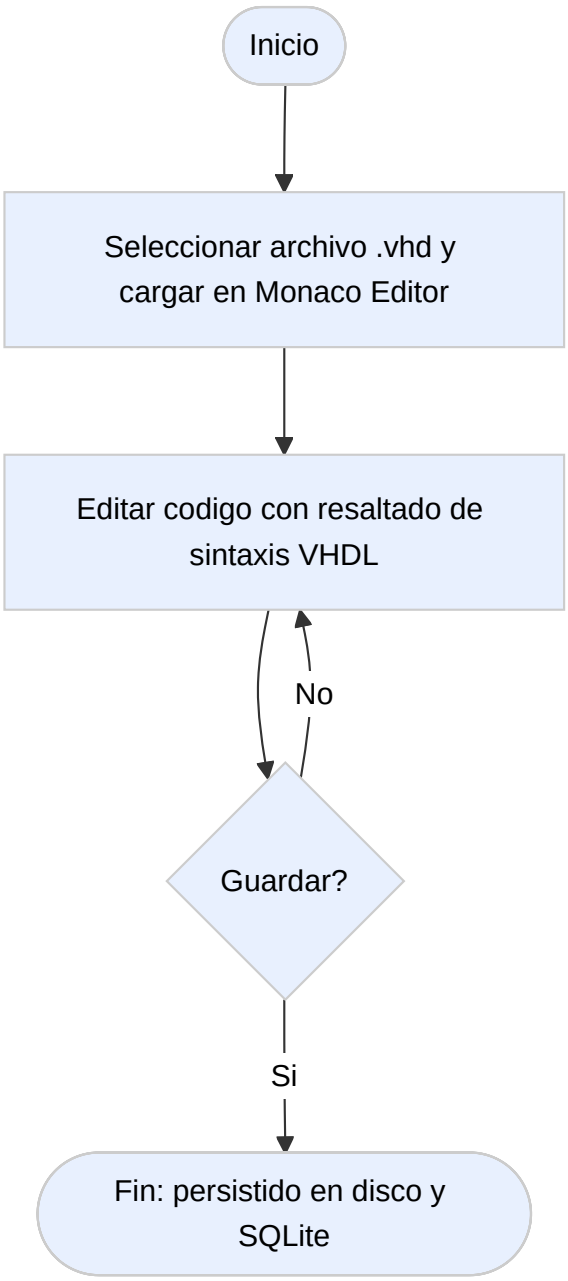

Figura 5.23: *Diagrama de flujo del CU-03: Editar Codigo HDL.*

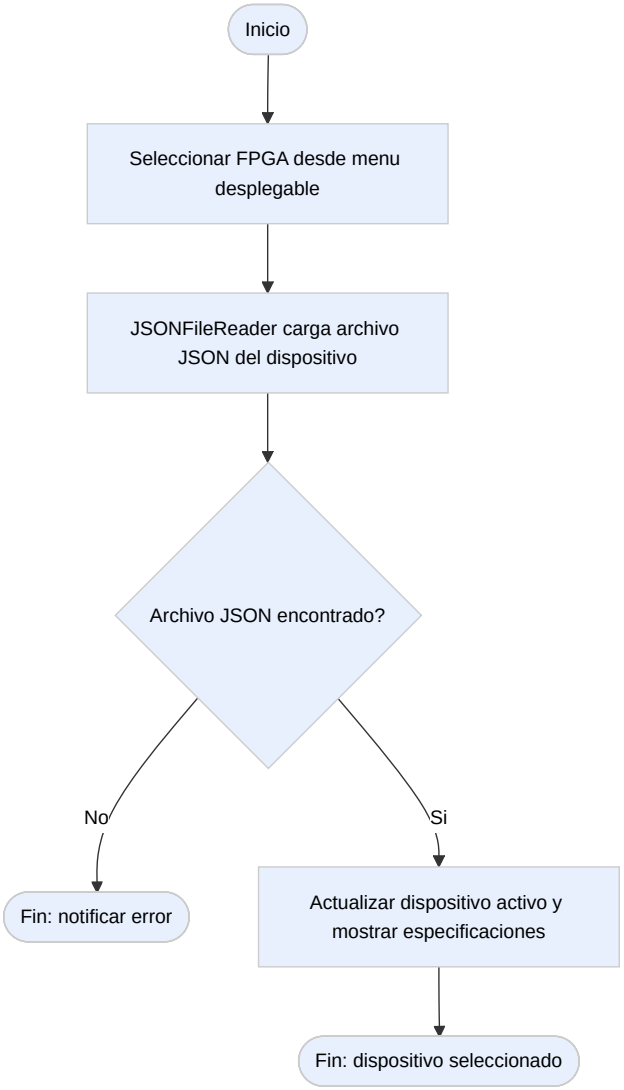

Figura 5.24: *Diagrama de flujo del CU-04: Seleccionar Dispositivo FPGA.*

5.3.3. CU-05 y CU-06: Preferencias de Idioma y Tema Visual

CU-05: Cambiar Idioma.

Campo	Descripción
ID	CU-05
Actor	Estudiante
RF relacionado	RF-09
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario selecciona un idioma desde el menú de configuración. 2. El servicio `Traductor` ejecuta `SetLanguageAsync(code)`. 3. El diccionario JSON del idioma seleccionado se carga desde `wwwroot/Dictionaries/`. 4. La preferencia se persiste en SQLite mediante `ApplicationDbContext.SetSettingAsync`. 5. El evento `StaticOnLangChanged` notifica a todos los componentes suscritos. 6. La interfaz se actualiza de forma reactiva sin recarga de página.
Flujo alternativo	1a. Si el idioma seleccionado es árabe (AR), el sistema activa el atributo `dir=rtl"` para escritura de derecha a izquierda.
Postcondición	La interfaz muestra todos los textos en el idioma seleccionado; la preferencia persiste entre sesiones.

CU-06: Cambiar Tema Visual.

Campo	Descripción
ID	CU-06
Actor	Estudiante
RF relacionado	RF-10
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario alterna entre tema claro y oscuro desde la configuración. 2. El sistema actualiza las propiedades CSS personalizadas (`-kmila-primary`, `-kmila-primary-light`, `-kmila-primary-dark`, `-kmila-accent`). 3. La preferencia se persiste en SQLite mediante la clave `KEY_THEME` de `AppSettings`. 4. El evento `StaticOnValueChanged` notifica a los componentes suscritos. 5. La interfaz refleja el nuevo tema sin recarga.
Flujo alternativo	—
Postcondición	La interfaz muestra la paleta de colores del tema seleccionado; la preferencia persiste entre sesiones.

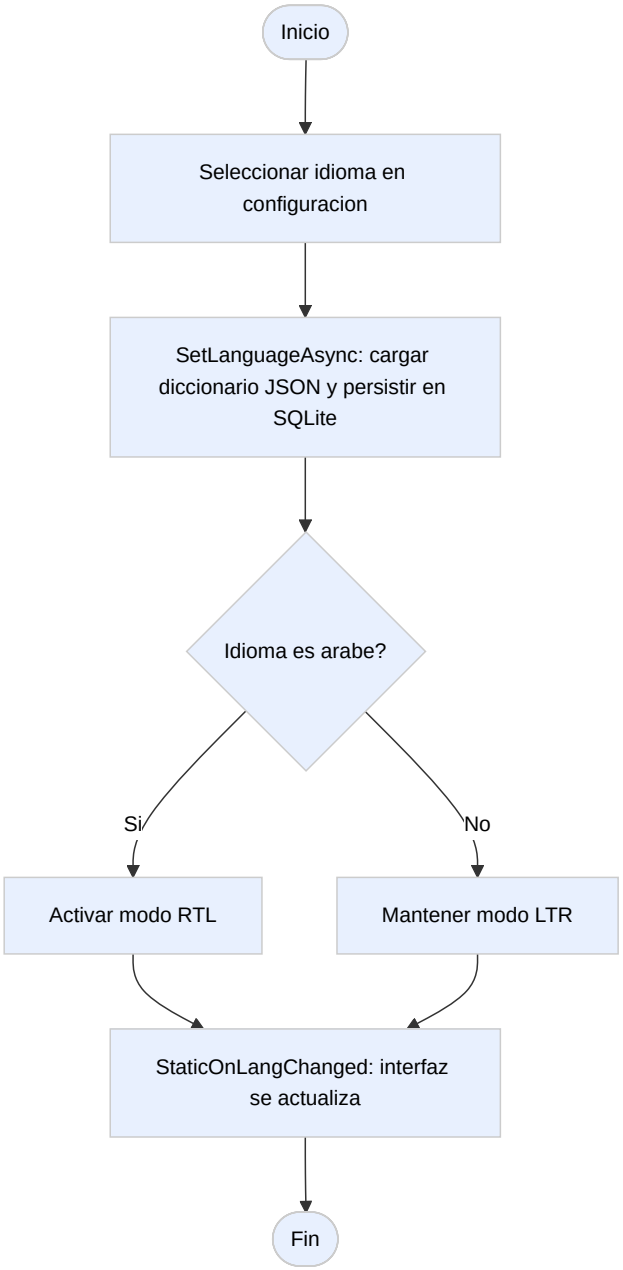

Figura 5.25: *Diagrama de flujo del CU-05: Cambiar Idioma.*

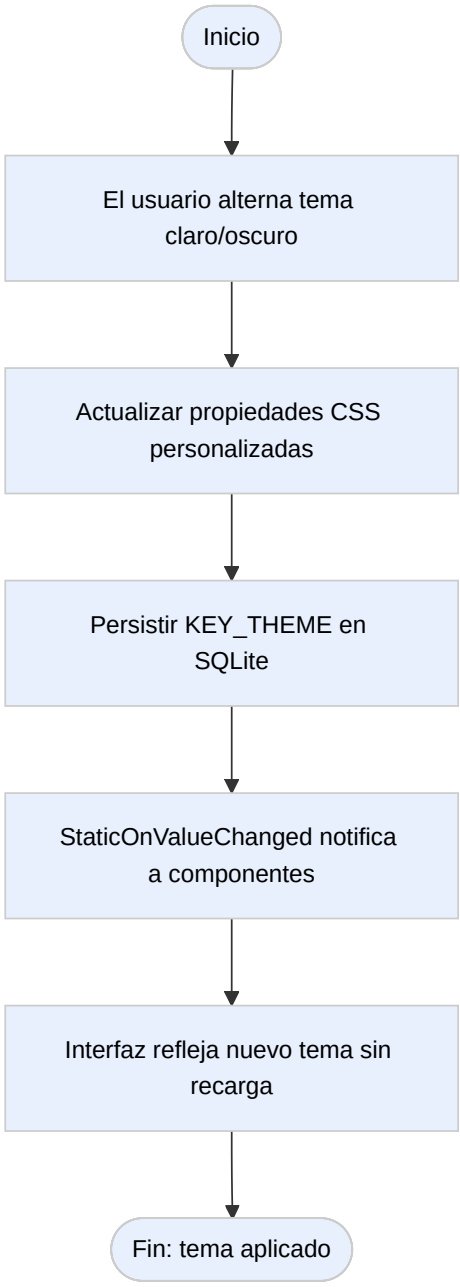

Figura 5.26: *Diagrama de flujo del CU-06: Cambiar Tema Visual.*

5.3.4. CU-07 y CU-08: Validación y Ejecución de Simulación (TT2)

CU-07: Validar Código VHDL.

Campo	Descripción
ID	CU-07
Actor	Estudiante
RF relacionado	RF-12, RF-13
Precondición	Existe un archivo `.vhd` o `.vhdl` abierto en el editor
Flujo principal	1. El usuario presiona "Validar programa" en `ButtonCollapse`. 2. `ProgramBuilder.Build(ProjectFile)` valida la extensión del archivo. 3. Se crea una instancia de `Parser` con la ruta del archivo. 4. `Parser.ParseFile()` ejecuta el análisis léxico y sintáctico mediante una máquina de estados (default, in-entity, in-architecture). 5. El sistema extrae entidades, puertos, señales y arquitecturas. 6. El componente `BuildProgressDisplay` muestra el progreso en tiempo real mediante el evento `OnBuildProgressChanged`. 7. Al completarse, el sistema muestra las entidades y recursos analizados en la pestaña de diagnósticos.
Flujo alternativo	4a. Error de sintaxis: `OnBuildError` notifica el error con el mensaje y la ubicación en el código.
Postcondición	`ProgramBuilder.Parser` contiene las entidades extraídas, habilitando la ejecución de simulación.

CU-08: Configurar y Ejecutar Simulación.

Campo	Descripción
ID	CU-08
Actor	Estudiante
RF relacionado	RF-06, RF-14, RF-15
Precondición	El código ha sido validado exitosamente (CU-07)
Flujo principal	1. El usuario presiona "Iniciar simulación". 2. El sistema muestra `SimulationParametersModal` con los puertos detectados. 3. El usuario configura la duración, escala temporal, frecuencia de reloj y estímulos por puerto (pares tiempo-valor). 4. `StimulusBuilder.BuildConfig()` convierte los parámetros a `SimulationConfig` (detecta relojes, calcula ticks, genera eventos de estímulo). 5. `SimulationRunner.RunAsync()` ejecuta la simulación tick por tick. 6. En cada tick, `DeltaCycleEngine` aplica estímulos, alterna relojes y ejecuta ciclos delta hasta alcanzar un estado estable. 7. `SignalHistory` registra los cambios de cada señal. 8. `OnSimulationCompleted` entrega el historial al componente `WaveformViewer`.
Flujo alternativo	6a. El usuario cancela la simulación: el sistema detiene la ejecución y entrega los resultados parciales.
Postcondición	El historial de señales está disponible para visualización y exportación.

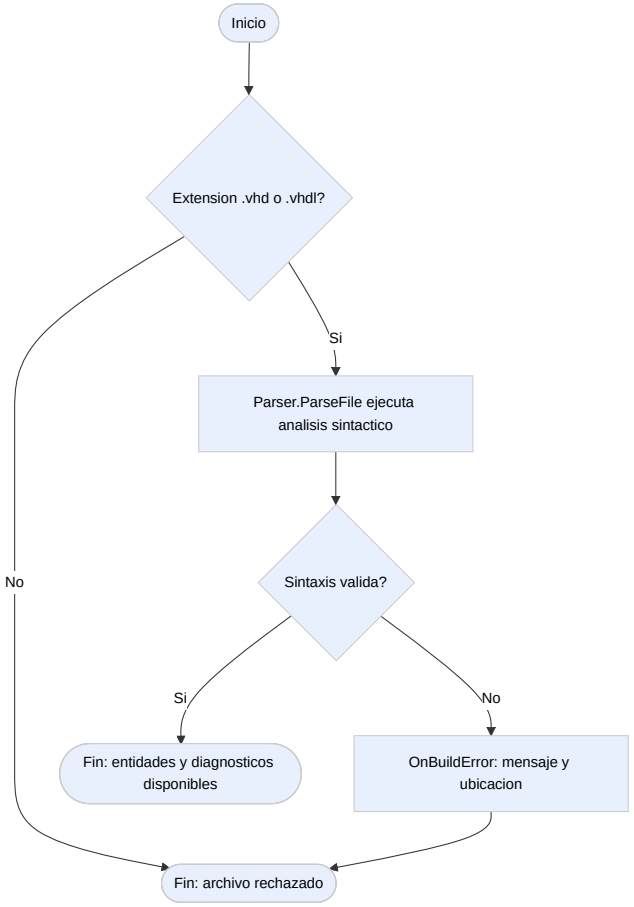

Figura 5.27: *Diagrama de flujo del CU-07: Validar Codigo VHDL.*

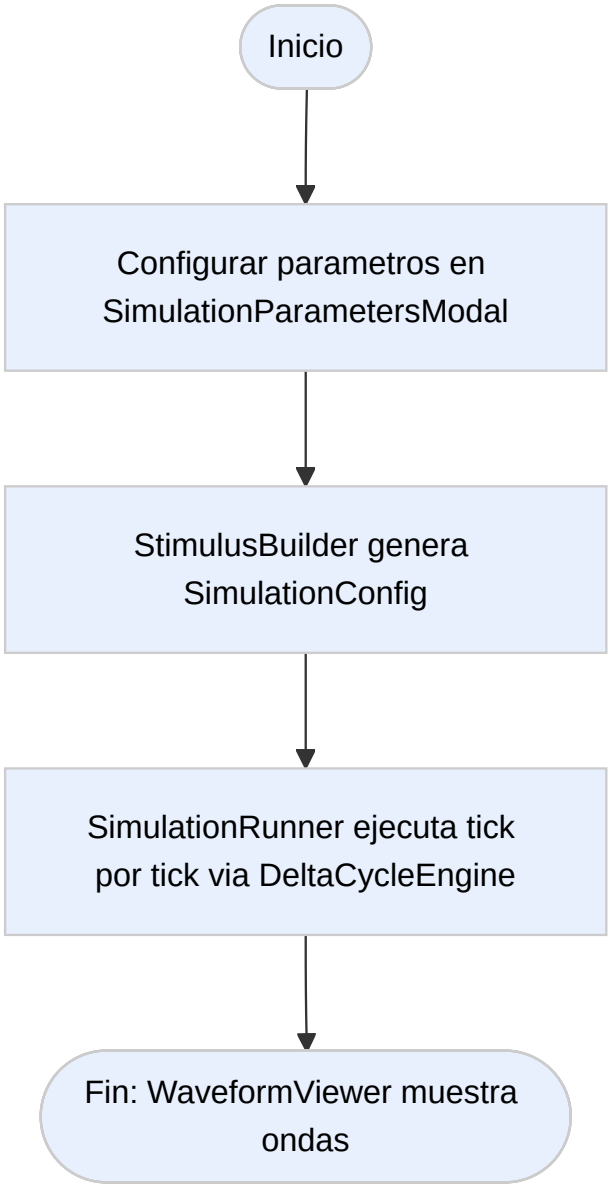

Figura 5.28: *Diagrama de flujo del CU-08: Configurar y Ejecutar Simulacion.*

5.3.5. CU-09 y CU-10: Visualización y Exportación de Resultados (TT2)

CU-09: Visualizar Formas de Onda.

Campo	Descripción
ID	CU-09
Actor	Estudiante
RF relacionado	RF-08
Precondición	Se ha ejecutado una simulación que produjo datos de señales
Flujo principal	1. `WaveformData.FromSignalHistory()` transforma el historial de señales en un formato compatible con Chart.js. 2. El componente `WaveformViewer` renderiza un gráfico de línea escalonada independiente por cada señal. 3. Las señales se codifican por color según su tipo: verde para señales de reloj, azul para entradas, naranja para salidas y morado para vectores. 4. El usuario puede acercar/alejar el eje temporal y desplazarse horizontalmente.
Flujo alternativo	—
Postcondición	Las formas de onda se presentan al usuario de forma interactiva.

CU-10: Exportar Resultados VCD.

Campo	Descripción
ID	CU-10
Actor	Estudiante
RF relacionado	RF-18
Precondición	Se ha ejecutado una simulación que produjo datos de señales
Flujo principal	1. El usuario presiona “Exportar VCD” en `WaveformViewer`. 2. `SignalHistory.ExportToVcd()` genera el contenido en formato Value Change Dump (VCD). 3. El sistema escribe el archivo `.vcd` en el directorio del proyecto.
Flujo alternativo	—
Postcondición	El archivo VCD es compatible con visores externos como GTKWave.

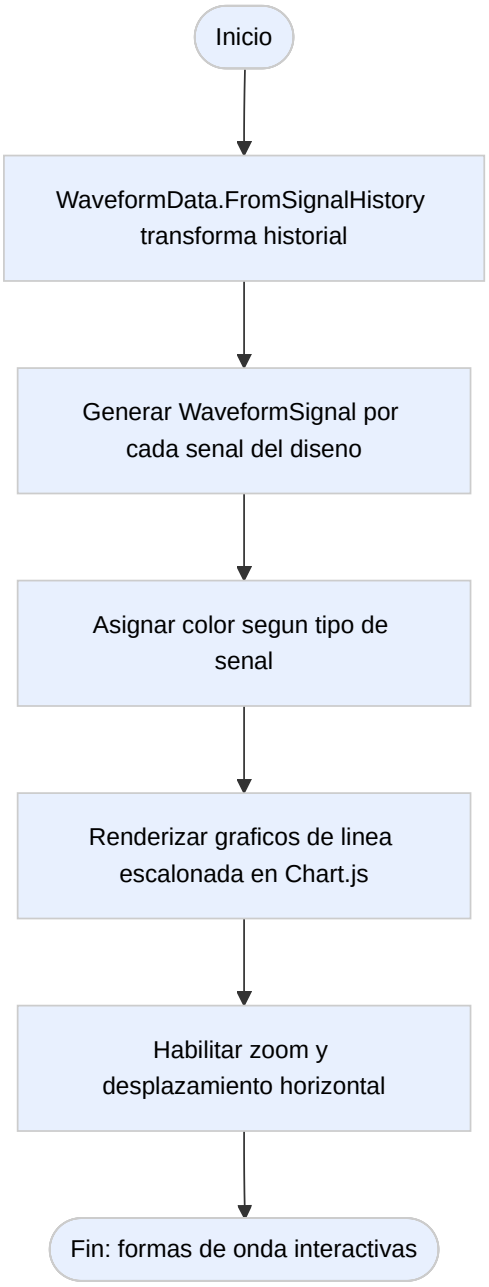

Figura 5.29: *Diagrama de flujo del CU-09: Visualizar Formas de Onda.*

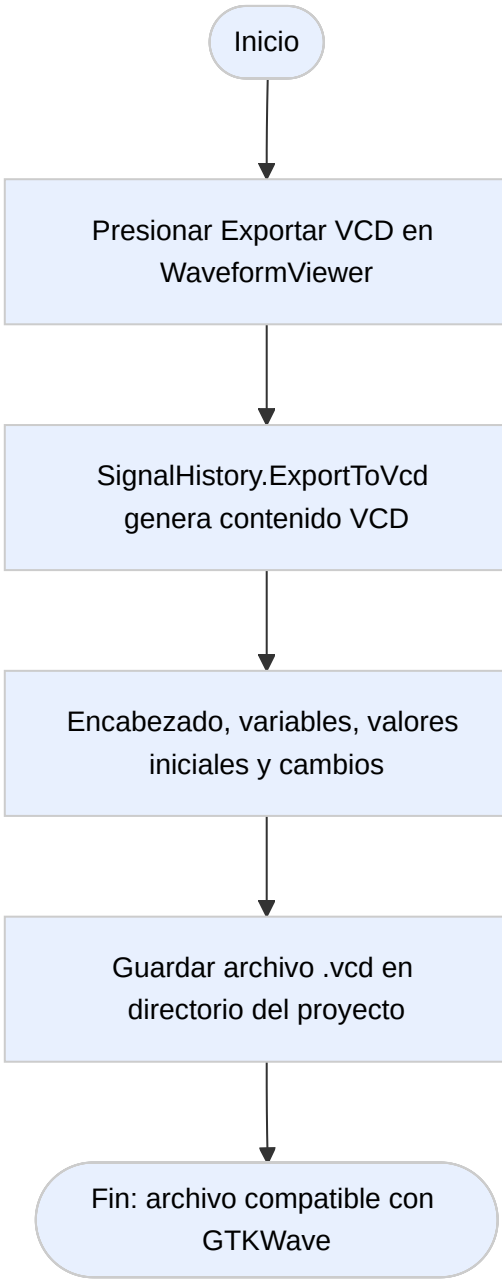

Figura 5.30: *Diagrama de flujo del CU-10: Exportar Resultados VCD.*

5.4. Diagramas de Secuencia

5.4.1. Secuencia: Carga de Archivo HDL

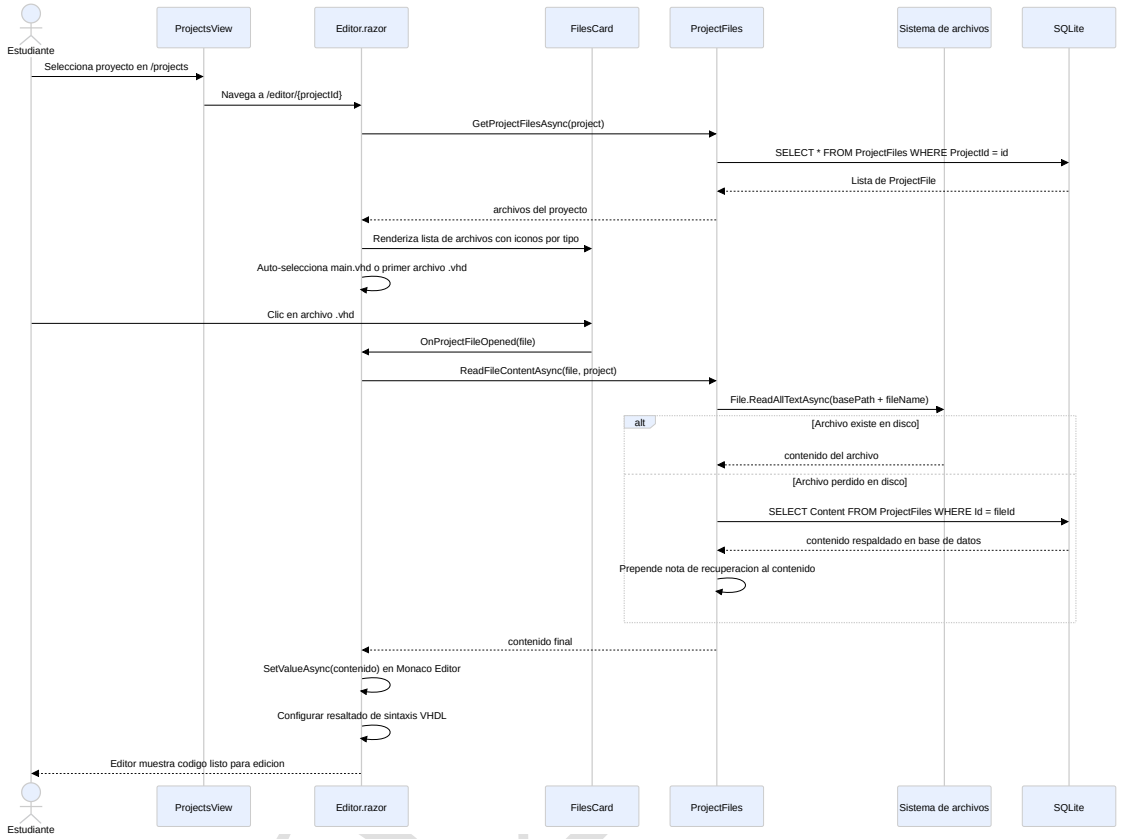

Figura 5.31: *Diagrama de secuencia: carga de archivo HDL. El repositorio ProjectFiles implementa persistencia dual: lee de disco y recurre a SQLite si el archivo fue eliminado. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.4.2. Secuencia: Validación de Código VHDL

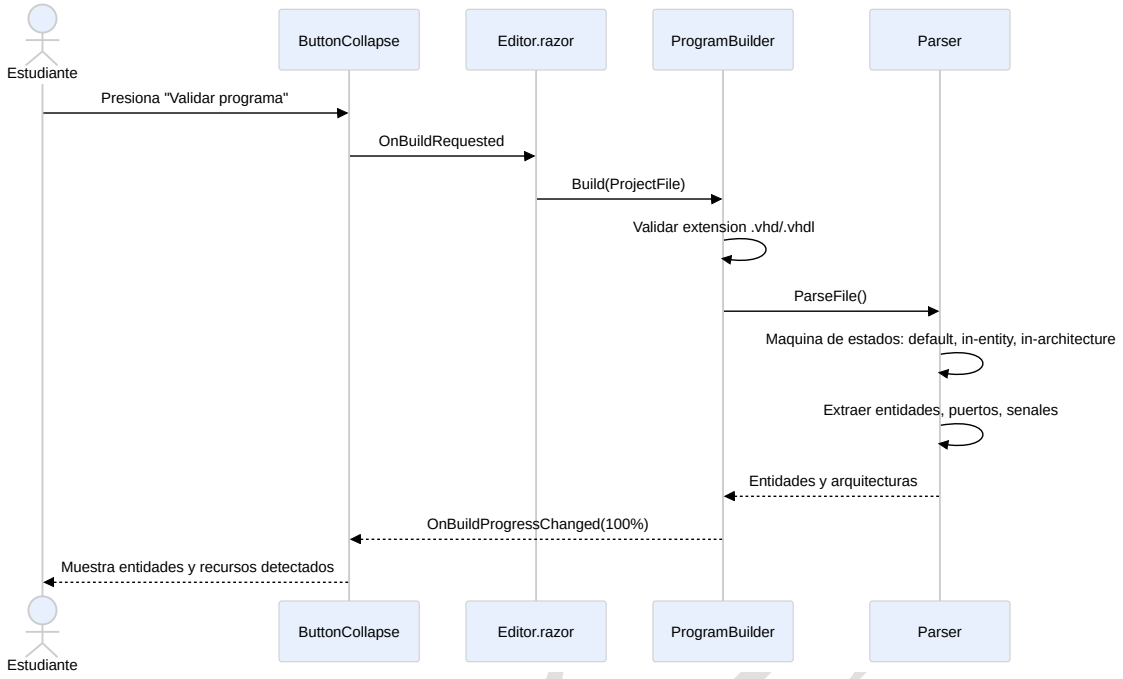

Figura 5.32: *Diagrama de secuencia: validacion de codigo VHDL. El Parser ejecuta el analisis lexico y sintactico mediante una maquina de estados. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.4.3. Secuencia: Configuración de Simulación

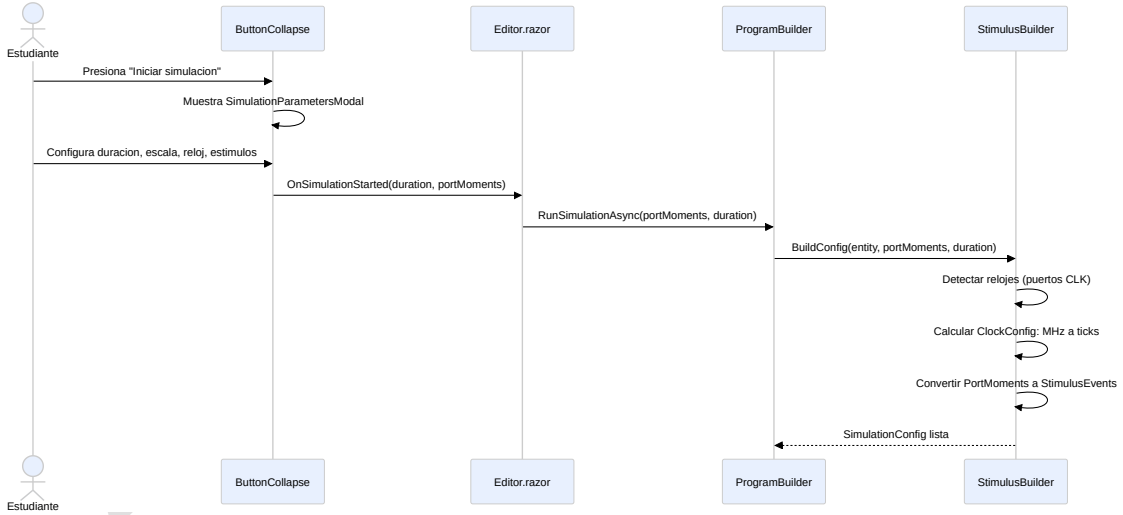

Figura 5.33: *Diagrama de secuencia: configuracion de simulacion. StimulusBuilder convierte los parametros de la interfaz al formato SimulationConfig del motor. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.4.4. Secuencia: Ejecución del Motor de Simulación

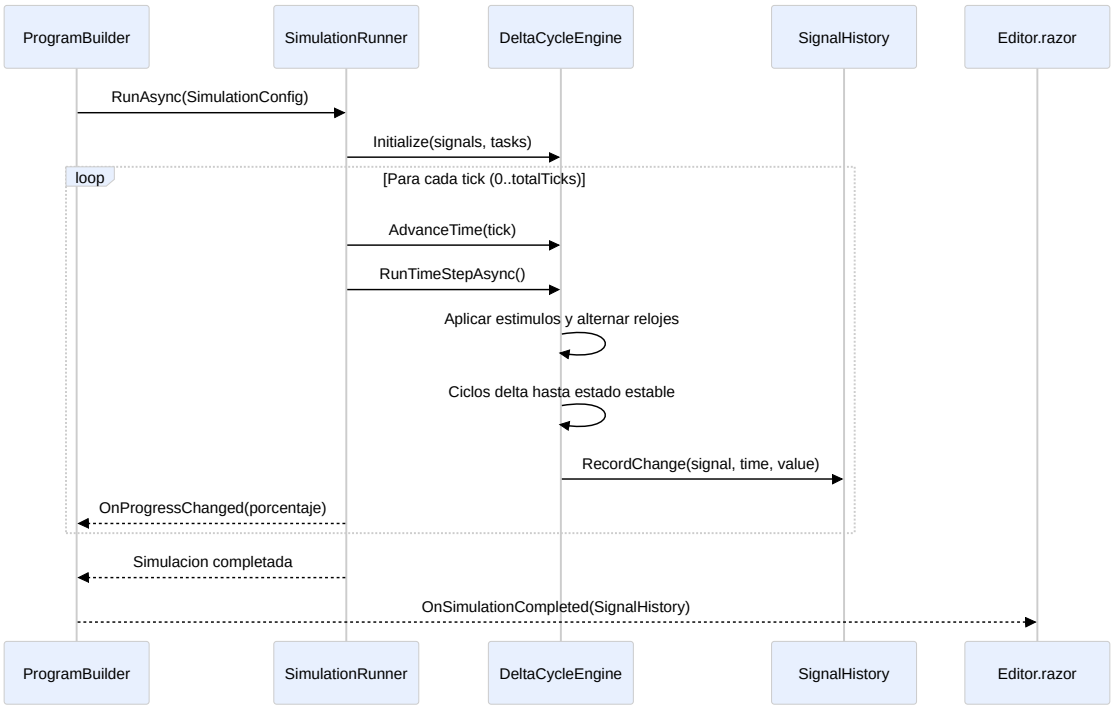

Figura 5.34: *Diagrama de secuencia: ejecucion del motor de simulacion. SimulationRunner itera tick por tick; DeltaCycleEngine resuelve ciclos delta y registra cambios en SignalHistory. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.4.5. Secuencia: Guardado de Archivo

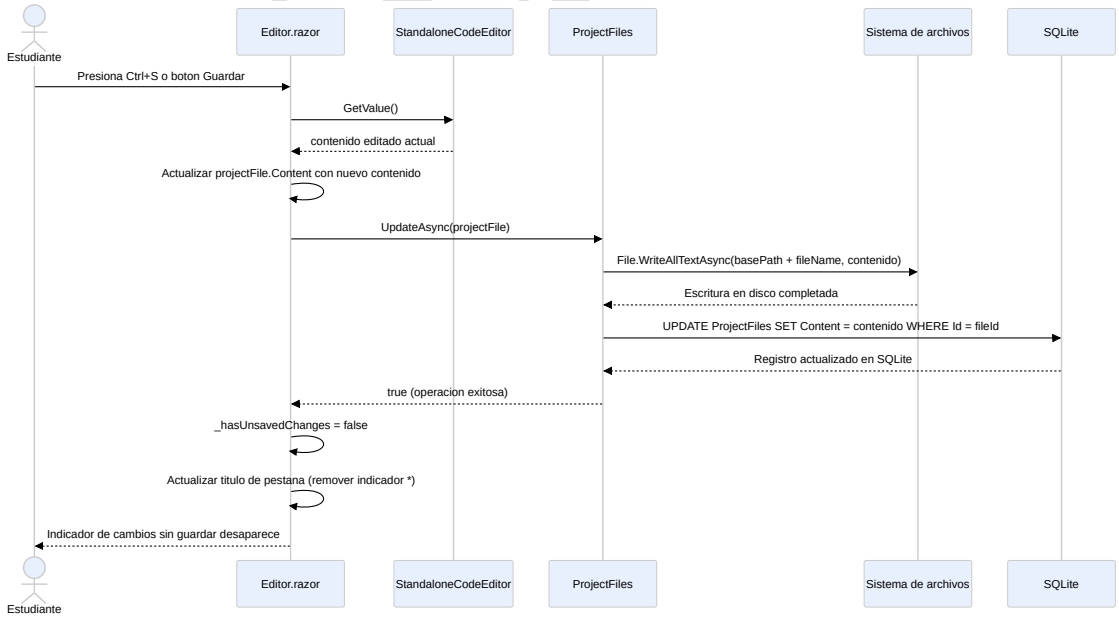

Figura 5.35: *Diagrama de secuencia: guardado de archivo. La persistencia dual escribe simultaneamente en disco y SQLite para garantizar la recuperacion ante perdida del archivo. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.4.6. Secuencia: Exportación de Resultados VCD

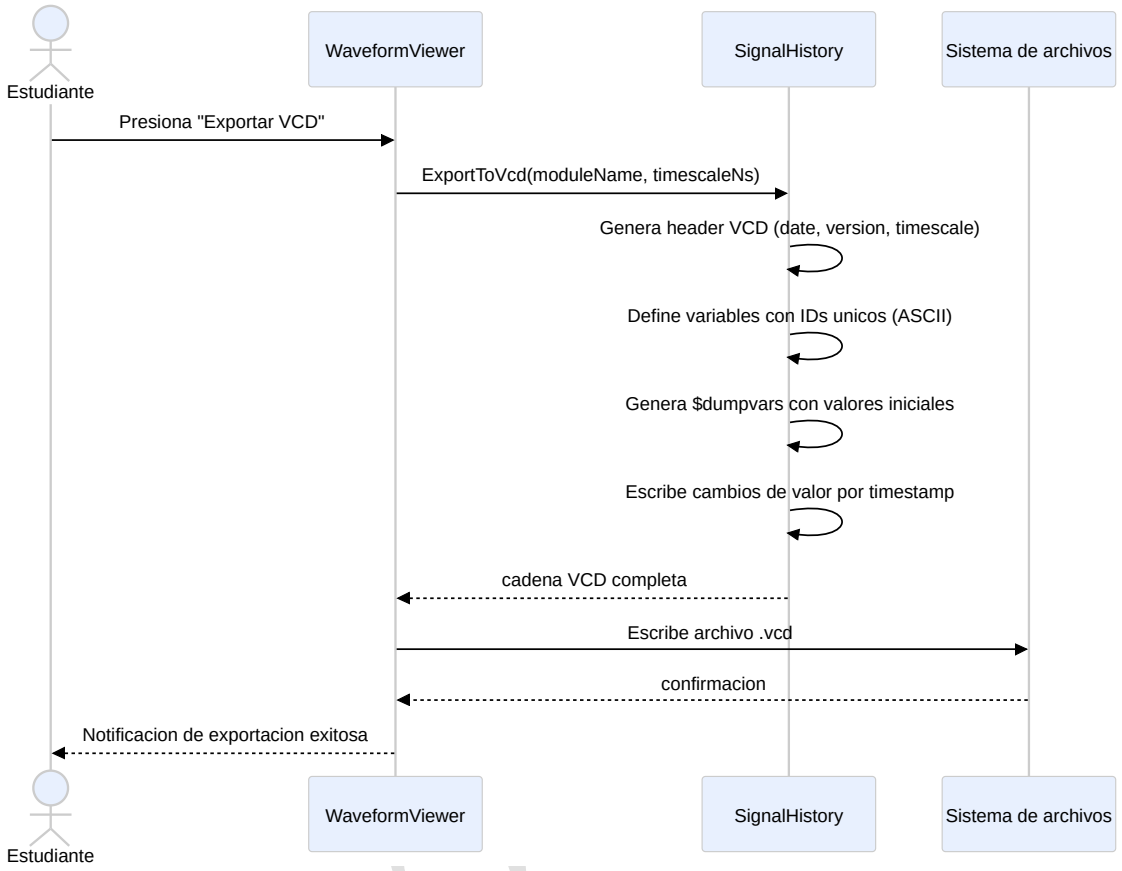

Figura 5.36: *Diagrama de secuencia: exportacion de resultados en formato VCD. El archivo generado es compatible con visores externos como GTKWave. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.5. Diagramas de Actividades

5.5.1. Actividad: Gestión de Proyecto y Edición

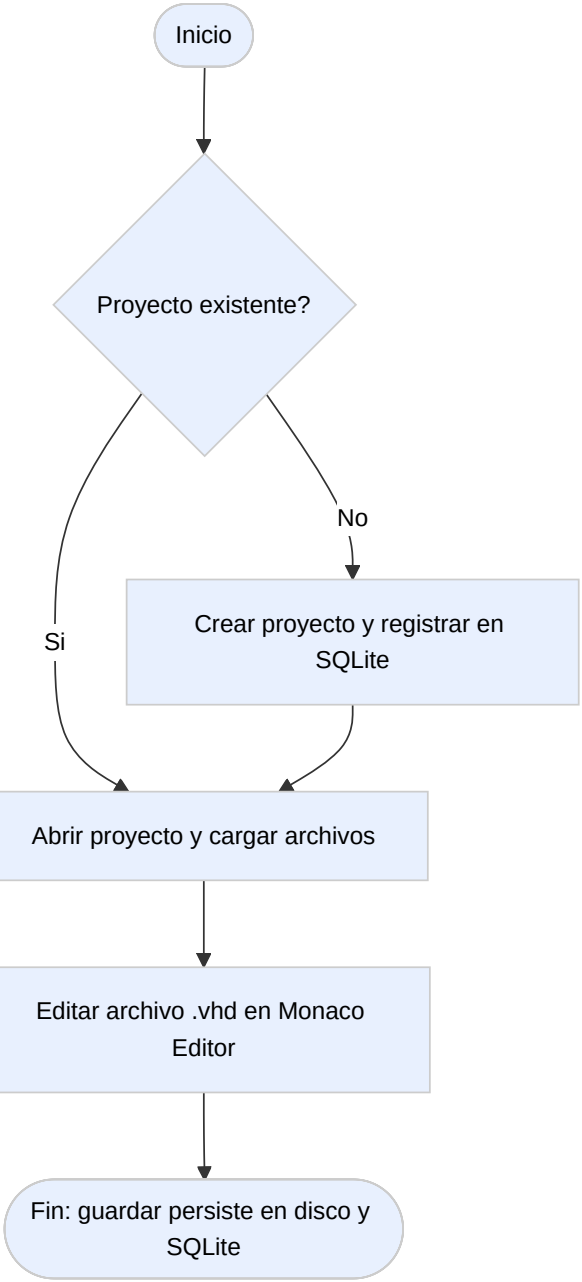

Figura 5.37: *Diagrama de actividades: gestion de proyecto y edicion. Cubre el flujo desde la creacion o apertura de un proyecto hasta el guardado del archivo editado. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.5.2. Actividad: Validación de Código VHDL

El proceso de validación se describe en dos partes: la preparación del Parser y el recorrido del archivo fuente.

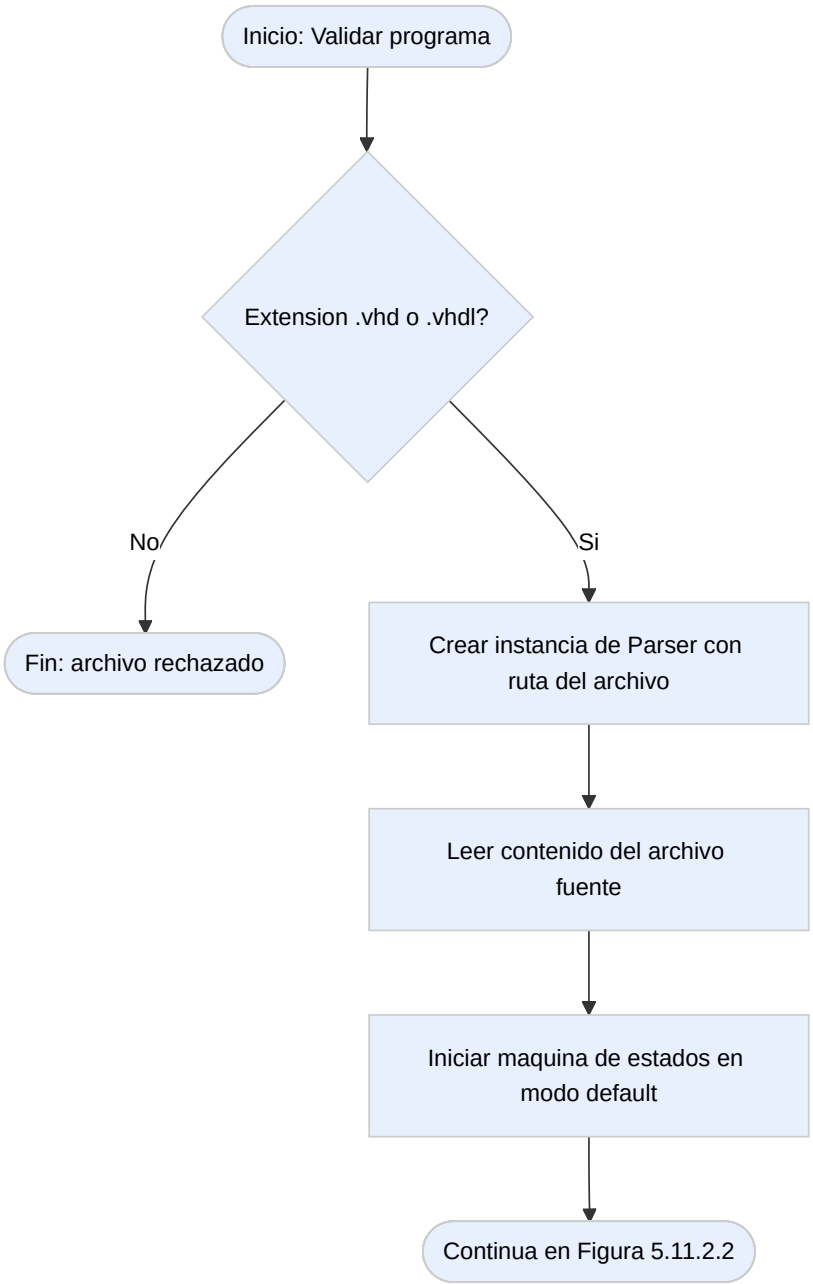

Figura 5.38: *Actividad de validacion: preparacion del Parser y lectura del archivo fuente.*

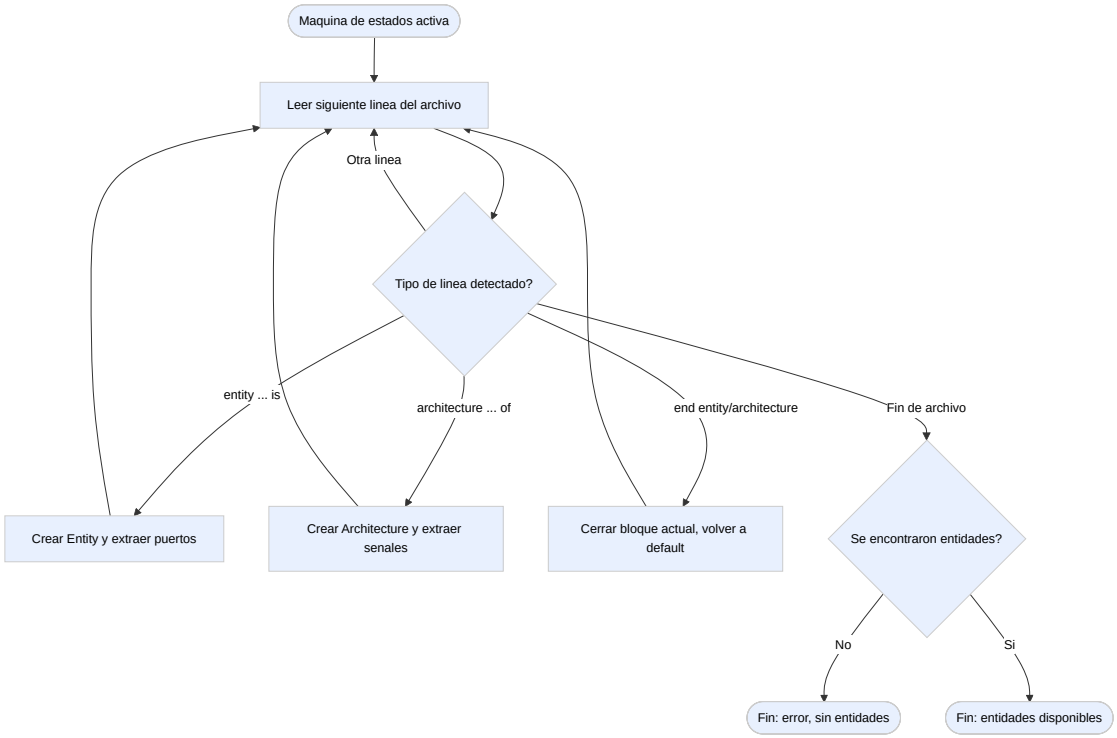

Figura 5.39: *Actividad de validacion: recorrido del archivo mediante maquina de estados (default, in-entity, in-architecture). Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.5.3. Actividad: Configuración de Simulación

El proceso de configuración se describe en dos partes: la captura de parámetros y la generación de la configuración del motor.

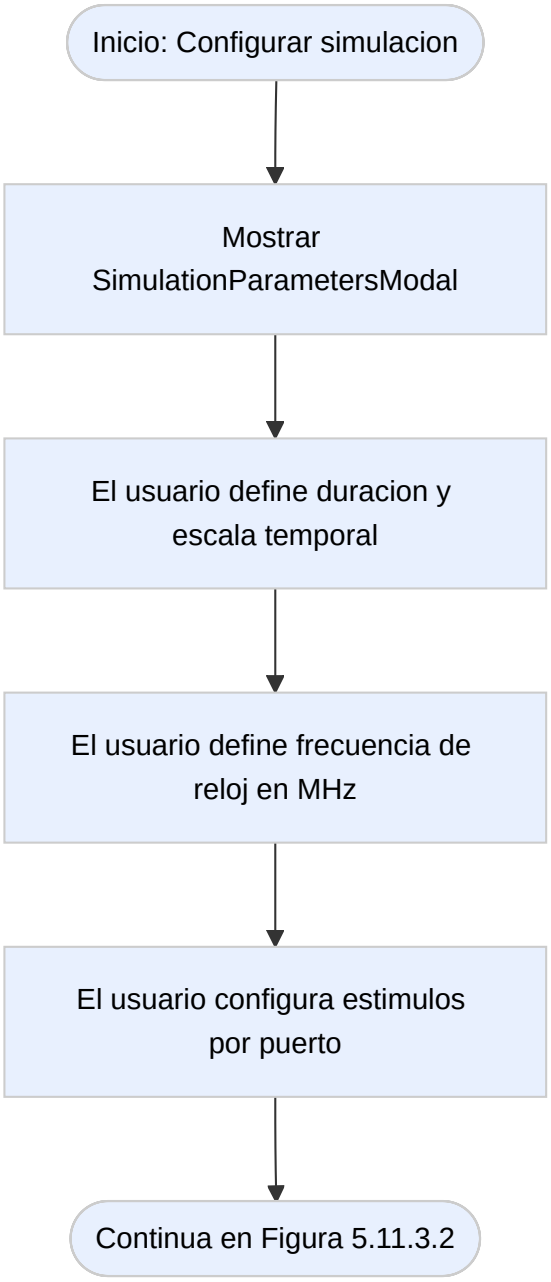

Figura 5.40: *Actividad de configuracion: captura de parametros del usuario.*

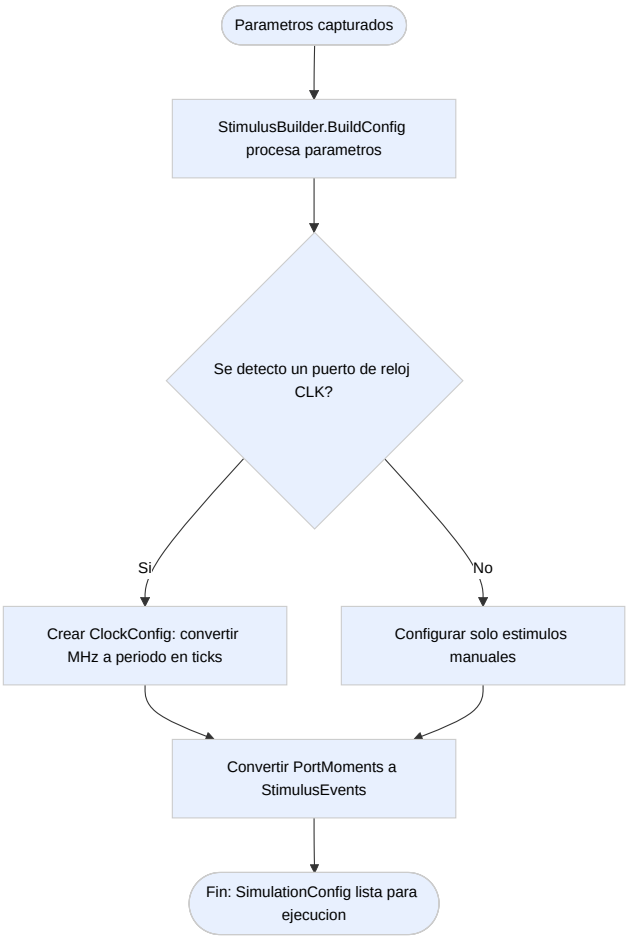

Figura 5.41: *Actividad de configuracion: StimulusBuilder genera SimulationConfig a partir de los parametros capturados.*

5.5.4. Actividad: Ejecución de Simulación

El proceso de ejecución se describe en dos partes: la inicialización del motor y el ciclo principal de simulación.

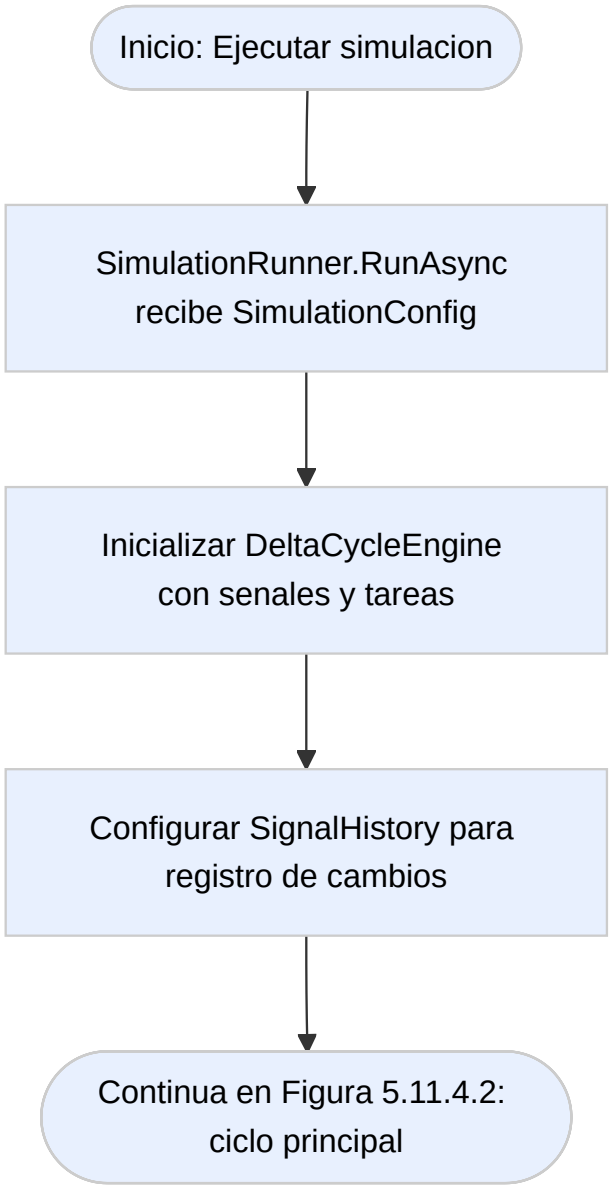

Figura 5.42: *Actividad de ejecucion: inicializacion del motor de simulacion.*

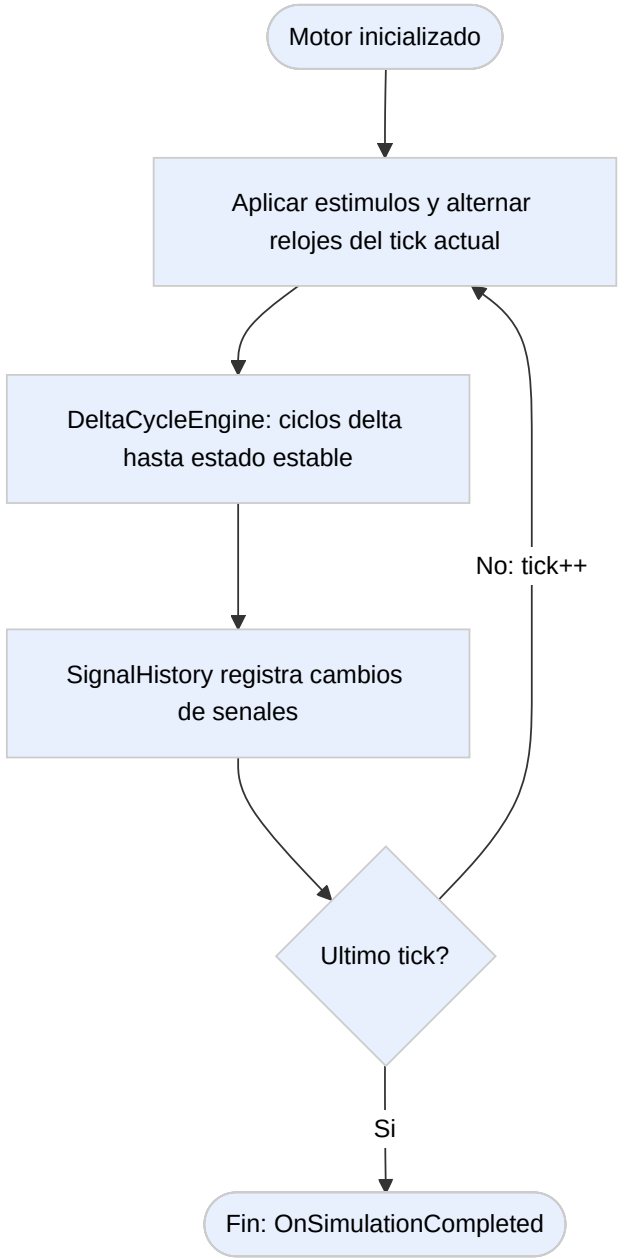

Figura 5.43: *Actividad de ejecucion: ciclo principal tick por tick. DeltaCycleEngine resuelve ciclos delta en cada instante de tiempo.*

5.5.5. Actividad: Visualización y Exportación

El proceso se describe en dos partes: la transformación y visualización de datos, y la exportación opcional.

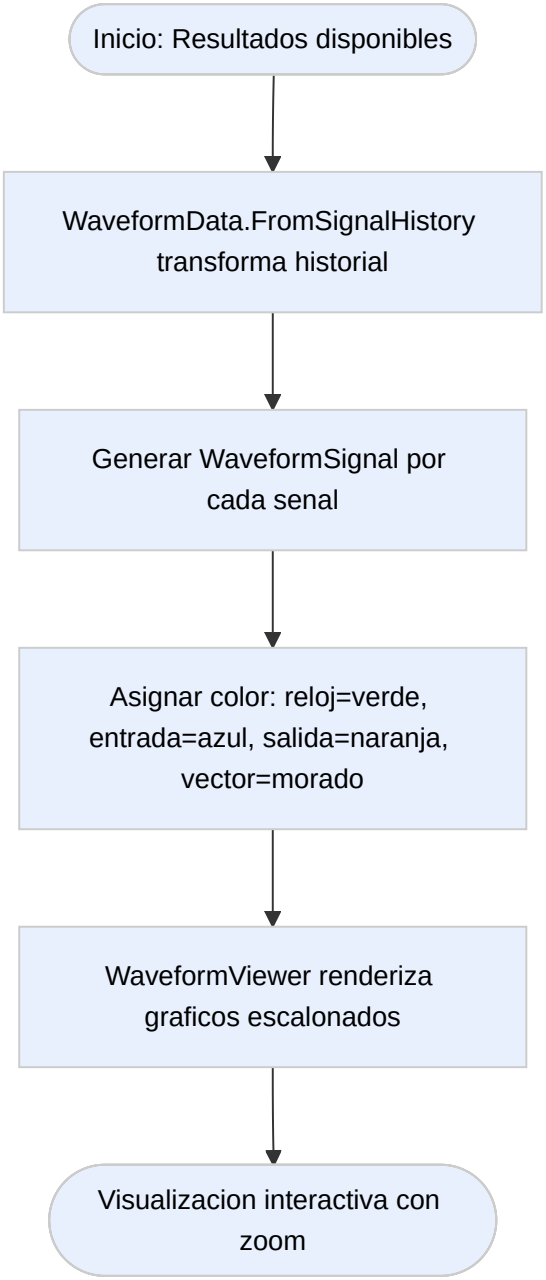

Figura 5.44: *Actividad de visualizacion: transformacion del historial de senales en graficos interactivos.*

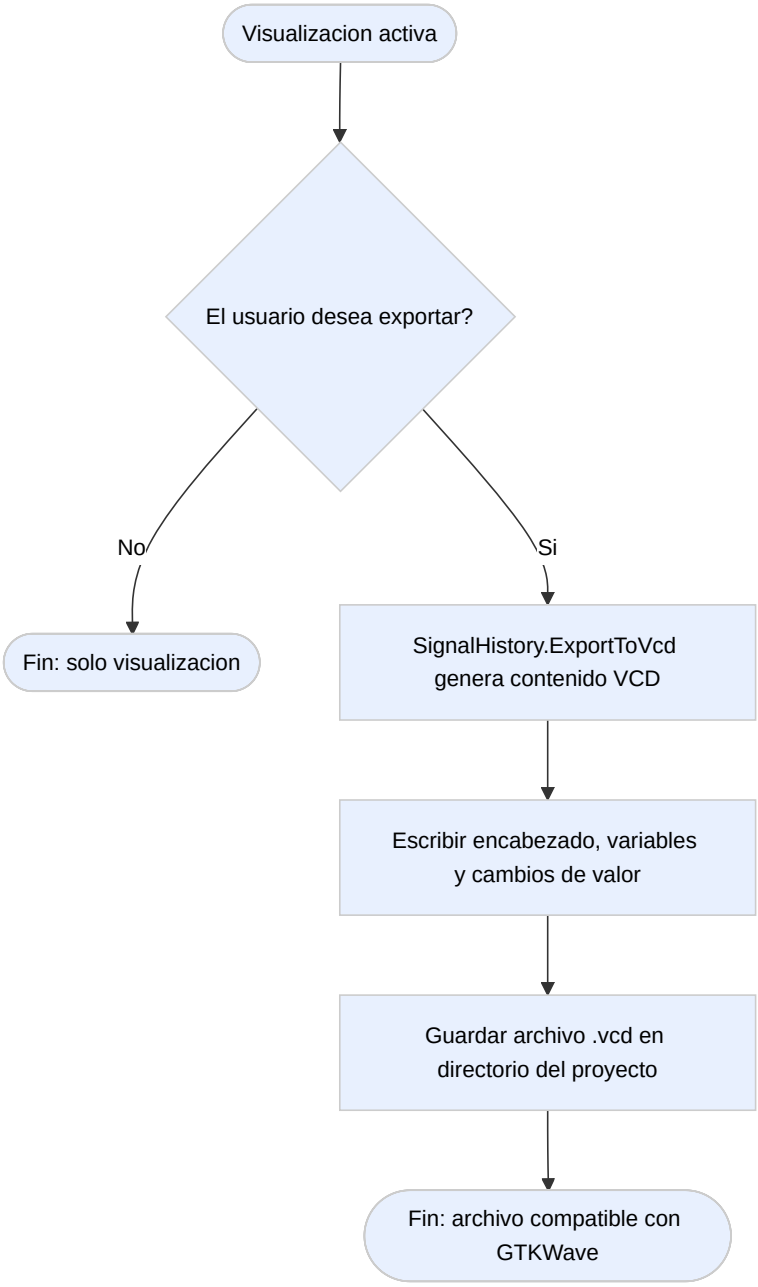

Figura 5.45: *Actividad de exportacion: generacion de archivo VCD a partir del historial de senales. Nota: se omiten acentos y caracteres especiales por compatibilidad con Mermaid.*

5.6. Mockups de la Interfaz

Los prototipos visuales de las pantallas principales de la aplicación se presentan a continuación. Las capturas corresponden al prototipo funcional de Kmila ejecutándose en modo web (ASP.NET Core + Blazor Server, accedido vía navegador en `localhost:5297`), con el idioma español y tema oscuro activos. Las capturas complementarias (pantalla de inicio, selector de idioma, tema claro, internacionalización con idioma árabe/japonés, módulo de aprendizaje y configuración avanzada) se encuentran en el Anexo A.3.

5.6.1. Pantalla Principal — Editor de Código

El editor de código es el componente central de la aplicación. Está construido sobre Monaco Editor (el mismo motor de Visual Studio Code), configurado con resaltado de sintaxis VHDL, numeración de líneas, minimapa de navegación, indicador de posición del cursor y atajo de guardado rápido (`Ctrl+S`). La Figura 5.46 muestra un archivo `gates.vhd` abierto en el editor, correspondiente a la lección de exploración de tablas de verdad del módulo de aprendizaje.

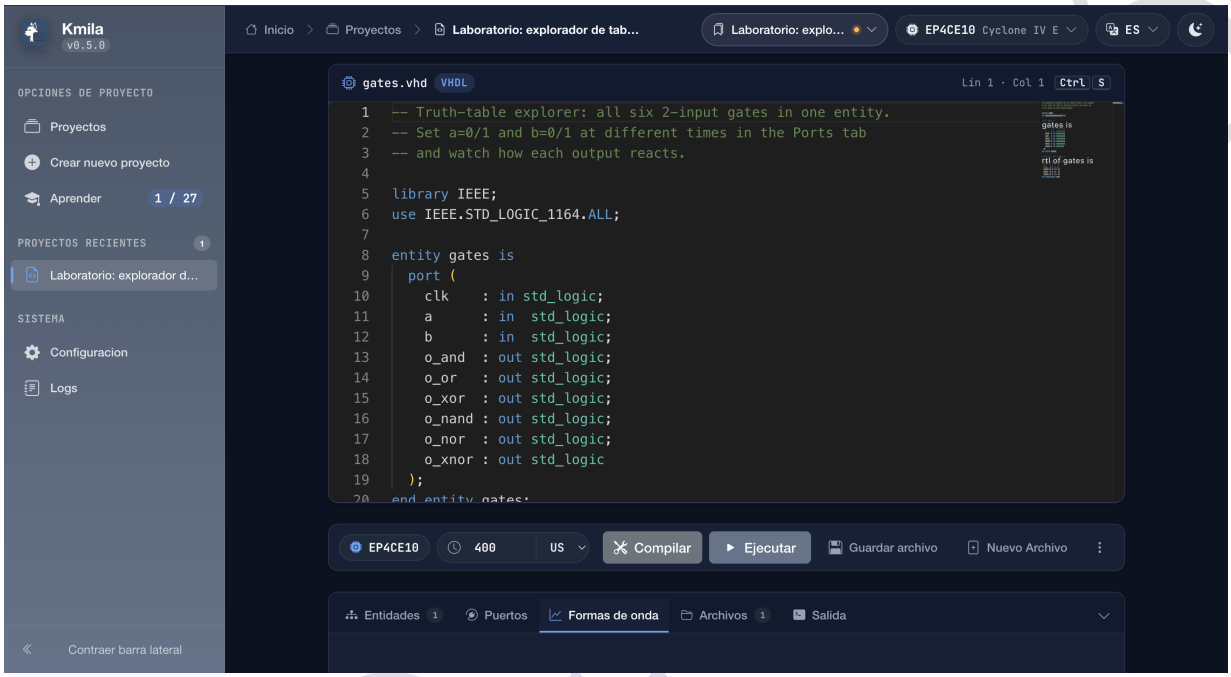

Figura 5.46: *Editor de código VHDL con Monaco Editor, mostrando el archivo `gates.vhd` del laboratorio de exploración de tablas de verdad. Se aprecia el resaltado de sintaxis, minimapa, numeración de líneas y la barra de acciones inferior (Compilar, Ejecutar, Guardar archivo, Nuevo Archivo) junto con el selector de FPGA (EP4CE10) y los parámetros temporales de simulación.*

El editor incorpora dos capacidades didácticas adicionales que distinguen a Kmila de un editor genérico: descripciones contextuales al posicionar el cursor sobre palabras clave VHDL (Figura 5.47) y plantillas de autocompletado inteligente con variantes pedagógicamente relevantes (Figura 5.48).

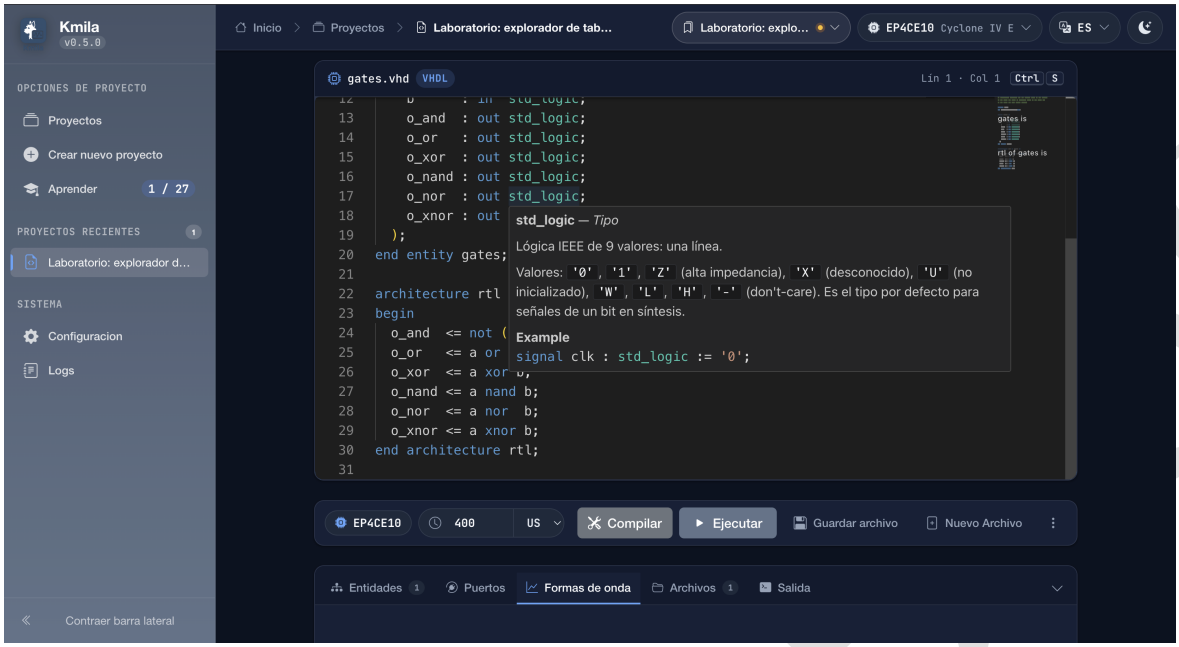

Figura 5.47: *Tooltip contextual sobre la palabra clave* `std_logic`*, mostrando la descripción del tipo junto con los nueve valores definidos por IEEE 1164 y un ejemplo de declaración. Este mecanismo refuerza el aprendizaje en el punto exacto donde el estudiante encuentra un concepto nuevo.*

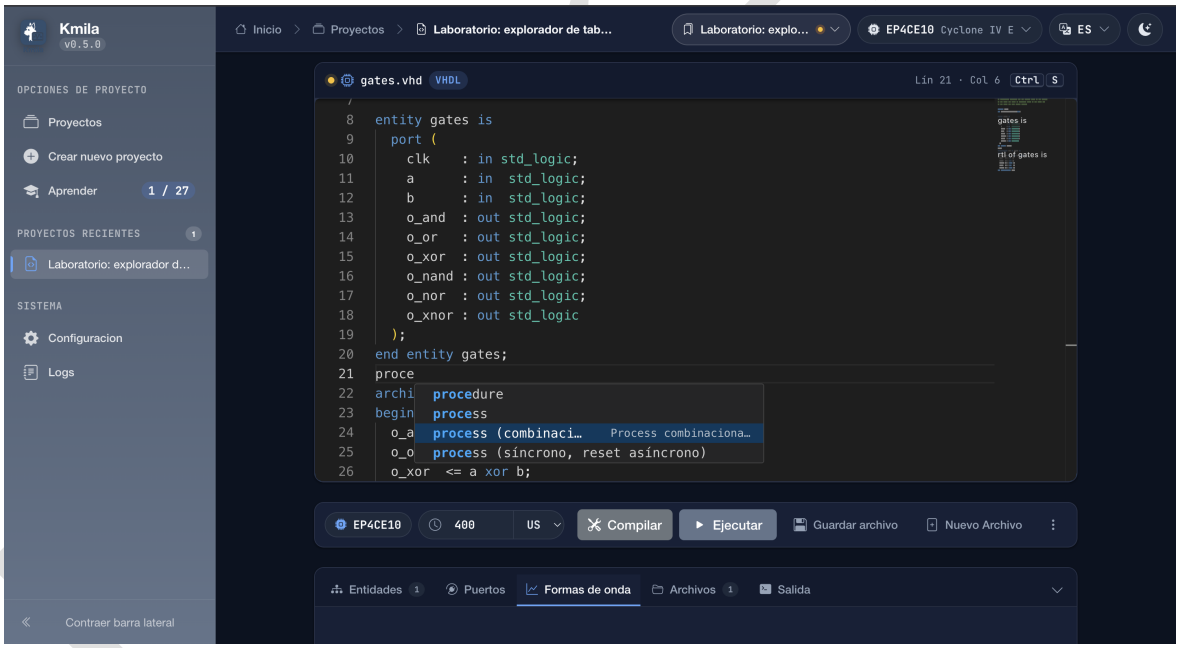

Figura 5.48: *Autocompletado con plantillas de snippets para* `process`*: se ofrecen tres variantes pedagógicamente diferenciadas (*procedure, process *genérico,* process combinacional, process síncrono con reset asíncrono*), guiando al estudiante hacia el estilo correcto según el caso de uso.*

5.6.2. Panel de Simulación

El panel inferior del editor agrupa las herramientas de análisis y configuración de simulación en pestañas. La pestaña *Puertos* (Figura 5.49) permite definir los estímulos de cada señal de entrada especificando momentos discretos en el eje temporal; cada fila representa un cambio de valor en un instante con su escala asociada (ns, us, ms). La pestaña *Entidades* (documentada en el Anexo A.3) permite inspeccionar la estructura de las entidades VHDL detectadas por el analizador. La pestaña *Uso de recursos* (Figura 5.50) presenta la estimación de utilización del dispositivo FPGA seleccionado.

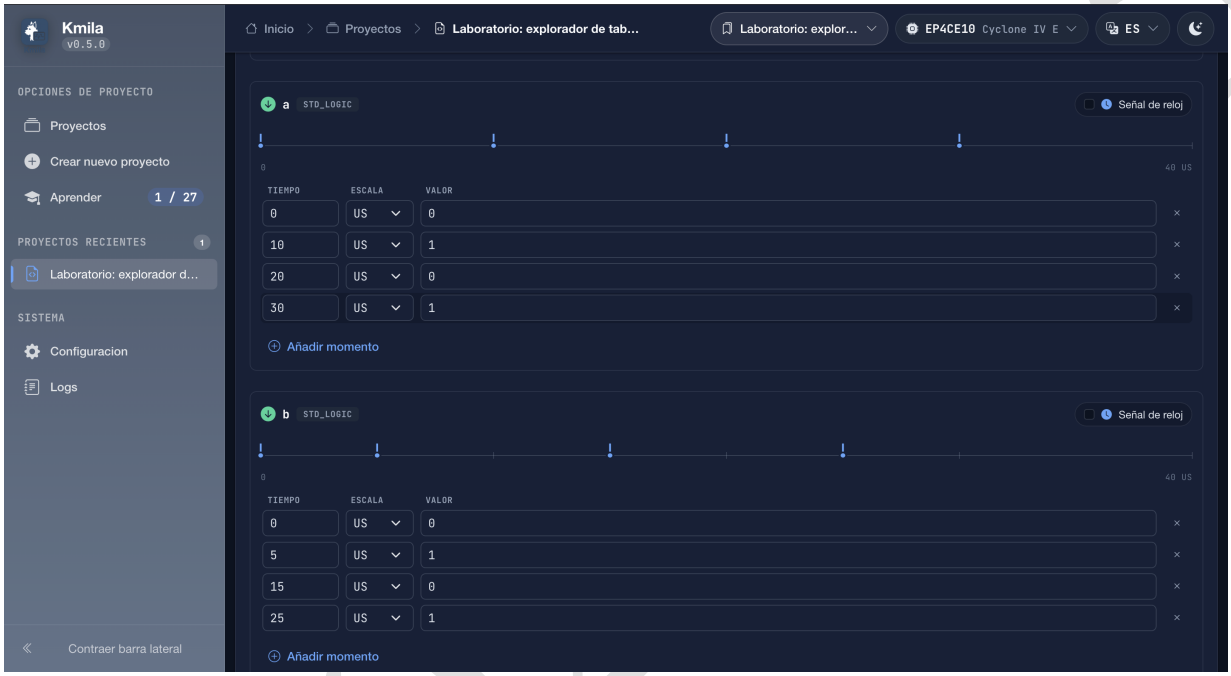

Figura 5.49: *Pestaña* Puertos*: configuración de estímulos para las señales de entrada* `a` *y* `b`*. Cada fila (Tiempo, Escala, Valor) define un instante en el que la señal cambia de valor. La línea temporal superior marca visualmente las transiciones programadas, y un interruptor permite designar cualquier señal como reloj (*clk*).*

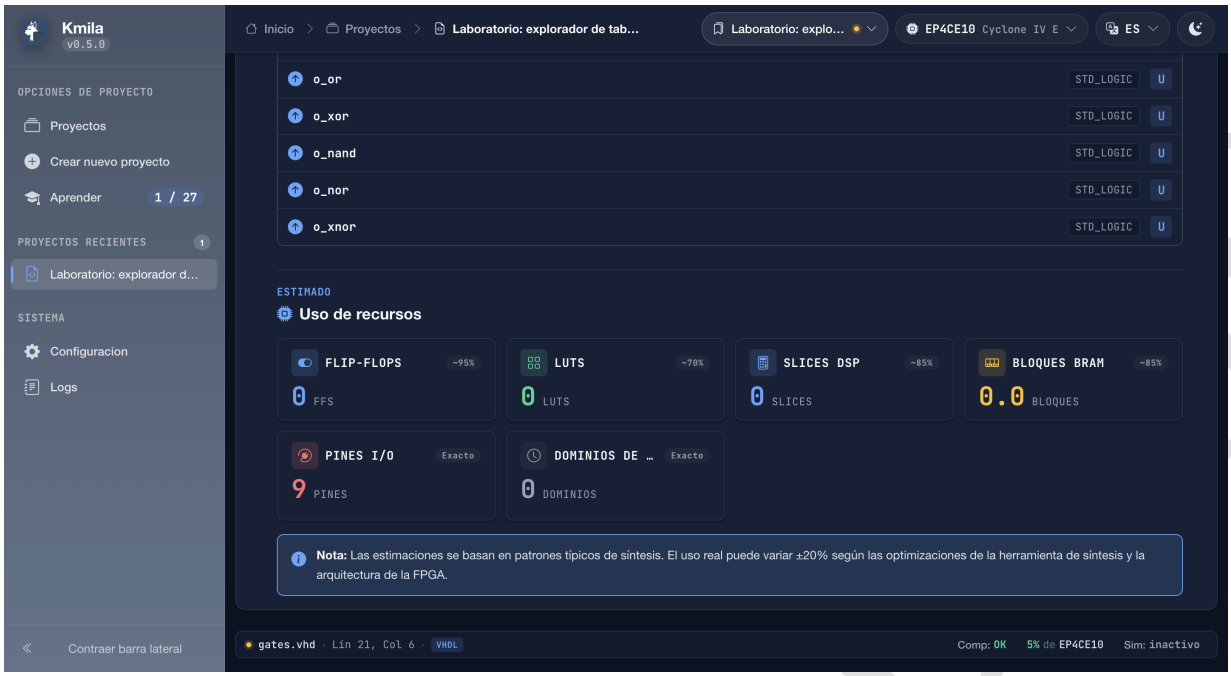

Figura 5.50: *Pestaña* Uso de recursos*: estimación de utilización del dispositivo FPGA seleccionado, desglosada en Flip-Flops, LUTs, Slices DSP, Bloques BRAM, Pines I/O y Dominios de reloj. La barra inferior indica la precisión de cada métrica (porcentaje de error esperado o resultado exacto) y una nota aclara la variabilidad inherente a las herramientas de síntesis reales.*

5.6.3. Visualizador de Señales (Ondas)

El visualizador de formas de onda, construido sobre Chart.js, presenta la evolución temporal de las señales del diseño tras ejecutar una simulación. Las señales de entrada se muestran en azul, el reloj en verde (con resaltado sólido) y las salidas en naranja, permitiendo al estudiante identificar visualmente el tipo de señal sin necesidad de leer la leyenda. La Figura 5.51 ilustra la salida del laboratorio de exploración de tablas de verdad con nueve señales simultáneas.

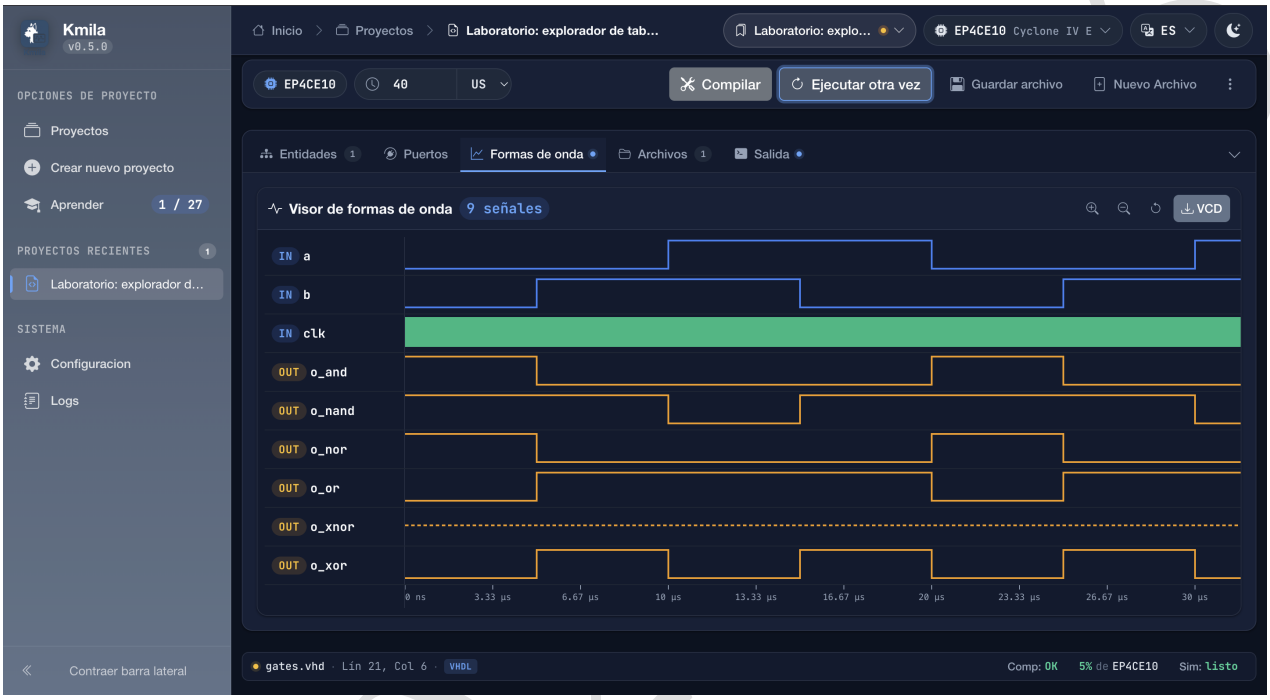

Figura 5.51: *Visualizador de formas de onda con nueve señales del laboratorio `gates.vhd`: tres entradas (`a`, `b`, `clk`), seis salidas combinacionales (`o_and`, `o_or`, `o_xor`, `o_nand`, `o_nor`, `o_xnor`). La línea temporal cubre el rango completo de simulación, con controles de acercamiento/alejamiento y botón de exportación al formato estándar VCD.*

> *Las formas de onda presentadas en este apartado son **ejemplos ilustrativos** del tipo de salida que producirá el visualizador una vez integrado completamente con el motor de simulación. Para el alcance de TT1, el foco se mantiene en la interfaz de visualización y su integración con los datos de entrada; el motor de simulación definitivo (DeltaCycleEngine y sus componentes) se consolida en TT2 según el cronograma del Capítulo 6.*

5.6.4. Panel de Navegación y Proyecto

La vista de *Proyectos* (Figura 5.52) presenta los proyectos existentes en tarjetas con resumen, fecha de última modificación y acceso directo al editor. El usuario puede crear un proyecto nuevo mediante un modal dedicado (Figura 5.53) o abrir rápidamente un ejemplo precargado. Una vez dentro del editor, el selector superior de proyecto ofrece el flujo de guardado: el proyecto activo se marca como **modificado** automáticamente al detectar cambios no guardados, y el menú desplegable permite guardar los cambios en la versión vigente o crear una nueva versión guardada (Figura 5.54).

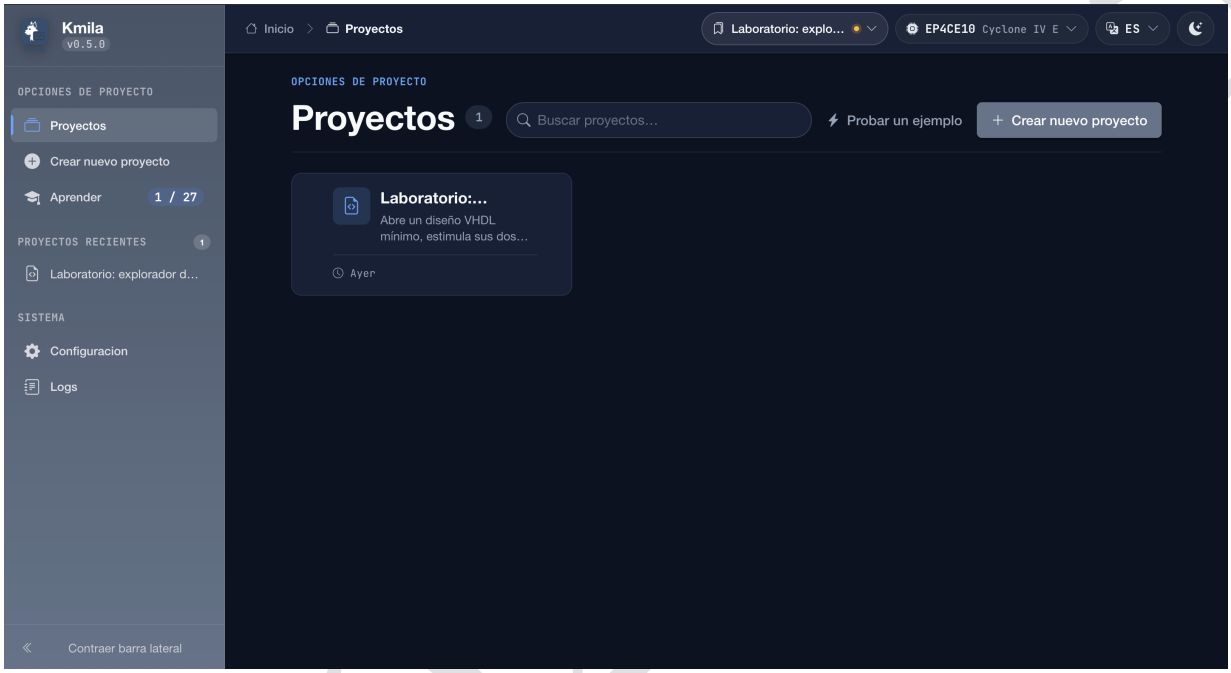

Figura 5.52: *Vista de* Proyectos*: lista de proyectos existentes con tarjetas de resumen. La barra superior ofrece búsqueda por nombre y los accesos directos* Probar un ejemplo *(que abre un proyecto de demostración) y* Crear nuevo proyecto *(que despliega el modal de la Figura 5.53).*

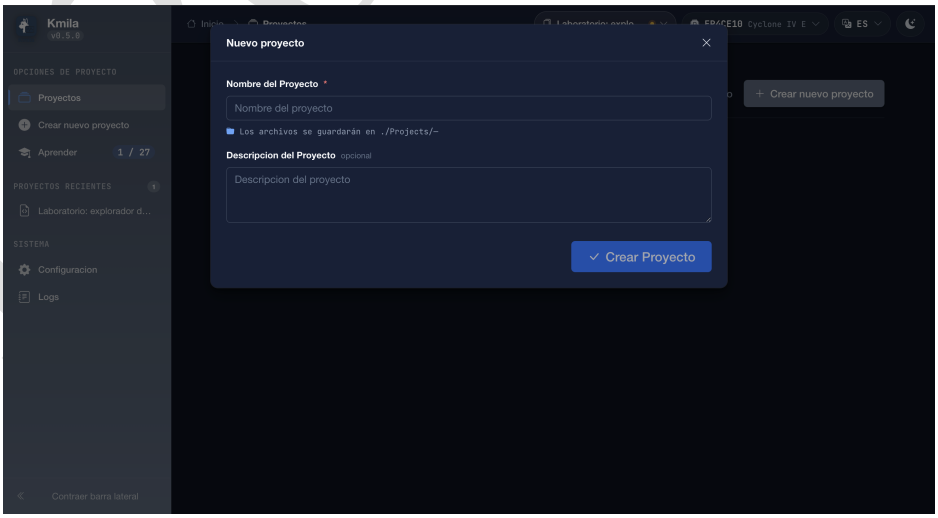

Figura 5.53: *Modal de creación de proyecto: el usuario indica nombre (obligatorio) y descripción (opcional). La ubicación de los archivos se preconfigura en* `./Projects/` *bajo el directorio de la aplicación, reforzando el principio de operación local sin servidores.*

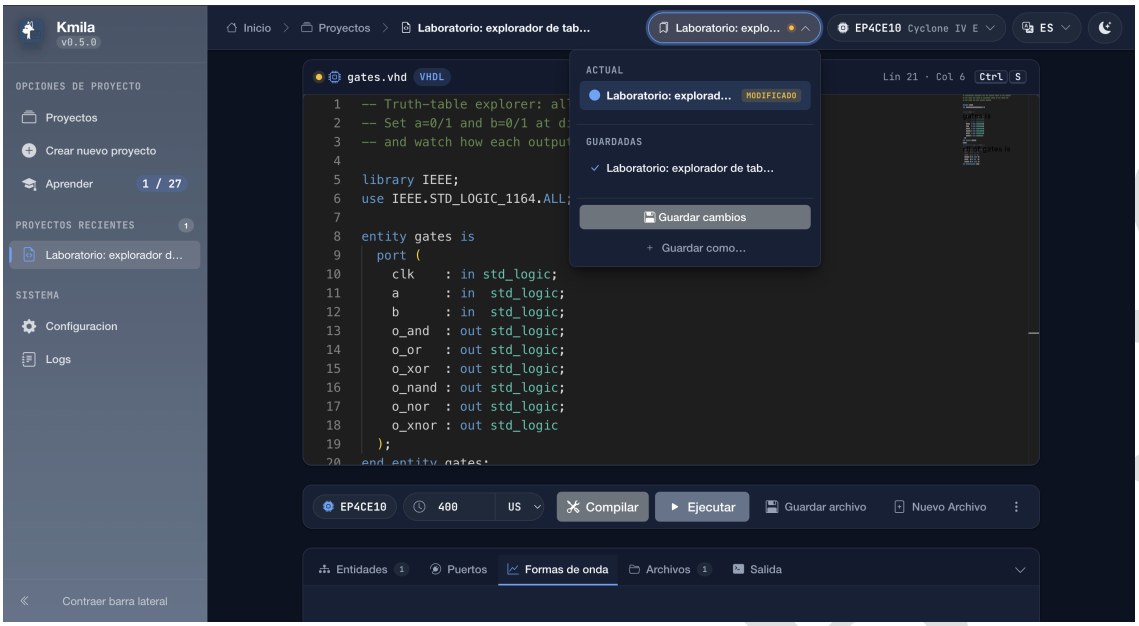

Figura 5.54: *Selector de proyecto activo con submenú de guardado. La sección* ACTUAL *marca el proyecto en edición con la etiqueta* MODIFICADO *cuando contiene cambios sin guardar; la sección* GUARDADAS *lista las versiones persistidas. Los botones* Guardar cambios *y* Guardar como... *cierran el flujo de versionado manual.*

5.6.5. Panel de Diagnóstico y Errores

El diagnóstico se presenta en dos niveles de detalle. La pestaña *Salida* del panel inferior del editor (Figura 5.55) muestra el registro cronológico del ciclo actual de compilación y simulación, con códigos de color por severidad. Para un análisis más profundo, la aplicación expone un visor de logs dedicado (Figura 5.56) accesible desde el menú *Sistema* de la barra lateral, que agrega todos los eventos del ciclo de vida de la aplicación.

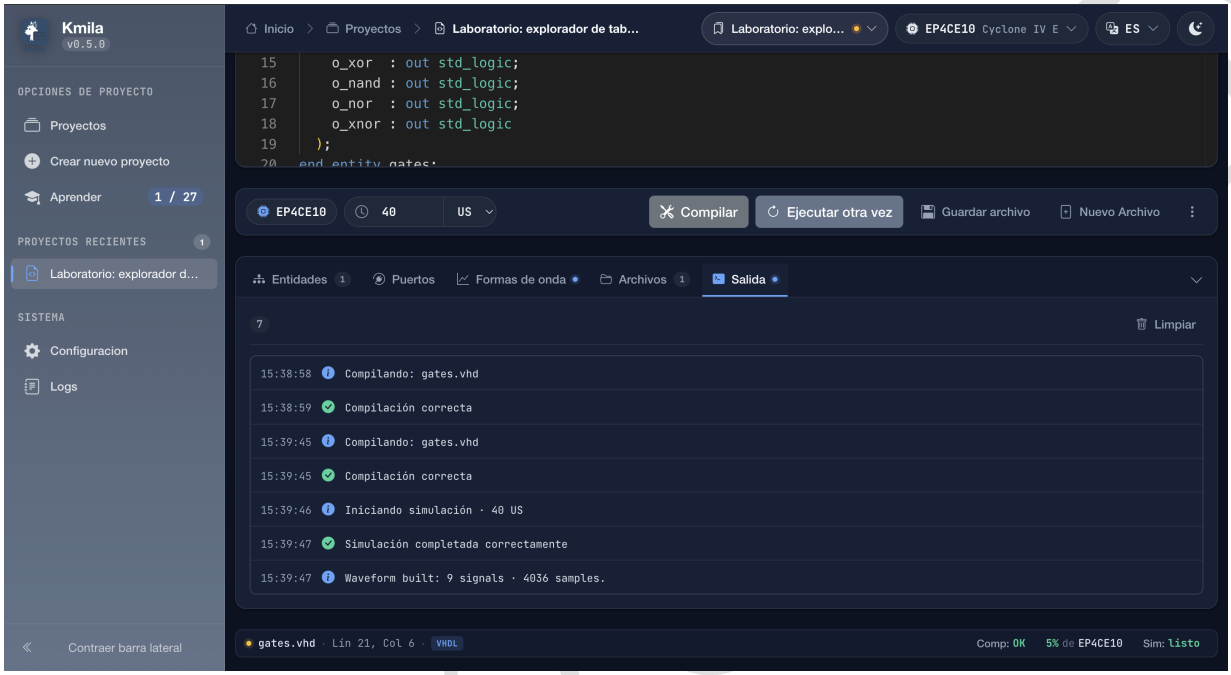

Figura 5.55: *Pestaña* Salida *del editor: log cronológico de la compilación y ejecución del proyecto activo, con indicadores de éxito (verde), información (azul) y advertencias. El botón* Limpiar *reinicia el panel.*

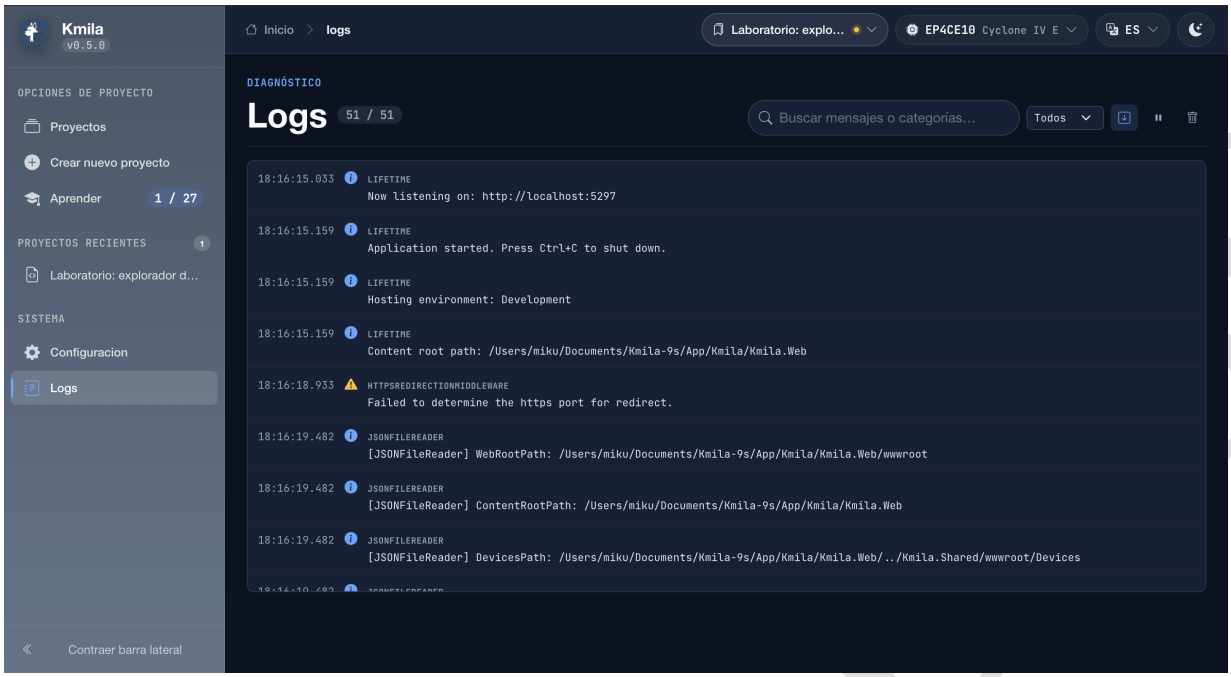

Figura 5.56: *Visor de logs del sistema: concentra 51 eventos del ciclo de vida de la aplicación. Cada entrada incluye hora, categoría (LIFETIME, JSONFILEREADER, HTTPSREDIRECTIONMIDDLEWARE, etc.), mensaje y nivel de severidad. Incluye búsqueda por texto, filtro por categoría y controles de exportación.*

5.6.6. Selector de Dispositivos FPGA

El selector de dispositivos FPGA, localizado en la barra superior del editor, despliega el catálogo completo agrupado por familia del fabricante seleccionado. La Figura 5.57 muestra el menú desplegado con los nueve modelos Intel (Altera) Cyclone IV E disponibles. El cambio de dispositivo recalcula automáticamente la estimación de utilización de recursos (Figura 5.50).

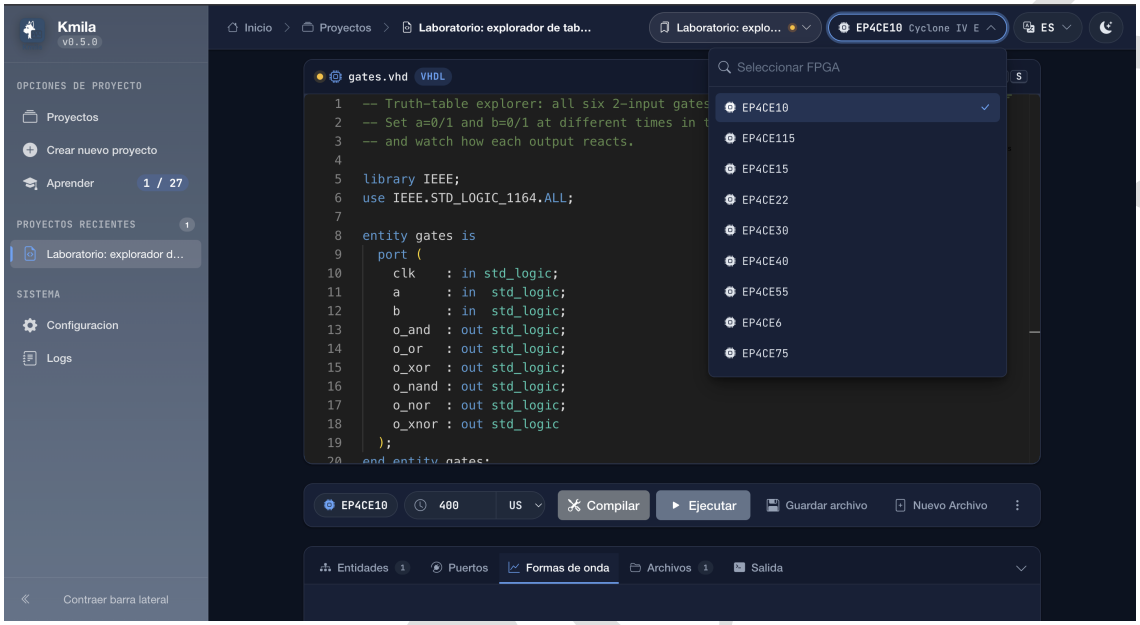

Figura 5.57: *Selector de dispositivos FPGA en la barra superior del editor, mostrando los nueve modelos Intel (Altera) Cyclone IV E del catálogo: EP4CE6, EP4CE10 (seleccionado), EP4CE15, EP4CE22, EP4CE30, EP4CE40, EP4CE55, EP4CE75 y EP4CE115. Los 34 dispositivos del catálogo completo se distribuyen entre tres fabricantes según la Sección 2.3.2.*

Capítulo 6

Cronograma de Actividades

El cronograma del proyecto abarca 12 meses, de febrero de 2026 a enero de 2027, divididos en dos fases correspondientes a Trabajo Terminal 1 (febrero–junio) y Trabajo Terminal 2 (julio–enero). Las actividades y su distribución temporal se derivan del protocolo registrado TT 2026-B059.

6.1. Cronograma General

Tabla 6.1: *Cronograma de actividades del proyecto (febrero 2026 – enero 2027).*

#	Actividad	FEB	MAR	ABR	MAY	JUN	JUL	AGO	SEP	OCT	NOV	DIC	ENE
1	Investigación del estado del arte	•	•										
2	Definición de requerimientos		•										
3	Análisis del alcance y planeación de módulos		•	•									
4	Definición de requerimientos técnicos y versiones compatibles		•	•									
5	Diseño de la arquitectura del sistema (MAUI/.NET, ASP.NET Blazor)	•	•	•									
6	Diseño de la interfaz y editor de código (Monaco)	•	•	•									
7	Desarrollo inicial del motor de simulación HDL			•	•								
8	Documentación preliminar y entrega para **Evaluación TT1**				•	•							
9	Desarrollo del motor de simulación avanzado			•	•		•						
10	Implementación de la visualización de señales y ciclos de reloj						•	•	•				
11	Generación de ejecutables multiplataforma compatibles							•	•	•			
12	Pruebas de validación (compatibilidad, rendimiento, usabilidad)										•	•	
13	Elaboración del manual de usuario y documentación técnica final											•	•
14	Publicación del código fuente en GitHub y entrega para **Evaluación TT2**												•

Las actividades 1 a 8 corresponden a la fase de Trabajo Terminal 1 y las actividades 9 a 14 a la fase de Trabajo Terminal 2. La actividad 9 (desarrollo del motor de simulación avanzado) inicia en la fase TT1 con un prototipo inicial y se extiende a TT2 con la implementación completa.

6.2. Avance Actual

Al cierre del mes de marzo de 2026, el avance del proyecto respecto al cronograma es el siguiente:

- **Actividad 1 — Investigación del estado del arte:** completada. Se realizó el análisis de herramientas existentes de simulación HDL en las categorías de código abierto, comercial, web/académica y académica de propósito general, documentado en la Sección 1.4 del presente trabajo.
- **Actividad 2 — Definición de requerimientos:** en curso. Se han identificado los requerimientos funcionales y no funcionales principales del módulo de interfaz de usuario.
- **Actividad 5 — Diseño de la arquitectura del sistema:** en curso. Se ha definido la arquitectura basada en el modelo C4, con la selección de .NET 9, MAUI y Blazor como tecnologías base.
- **Actividad 6 — Diseño de la interfaz y editor de código:** en curso. Se han elaborado los mockups de las pantallas principales y se ha integrado el editor Monaco como componente central.

El desarrollo se encuentra alineado con los tiempos establecidos en el protocolo registrado.

Referencias

[1] Amazon México. “Tarjeta de desarrollo FPGA Altera Cyclone IV EP4CE6,” visitado feb. de 2026. dirección: https://www.amazon.com.mx/dp/B09MYSM25M

[2] Digilent Inc. “Basys 3 Artix-7 FPGA Trainer Board,” visitado feb. de 2026. dirección: https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board/

[3] Coursera. “FPGA Design for Embedded Systems Specialization,” visitado feb. de 2026. dirección: https://www.coursera.org/specializations/fpga-design

[4] Intel Corporation. “Intel Quartus Prime Design Software,” visitado feb. de 2026. dirección: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[5] Advanced Micro Devices. “AMD Vivado Design Suite,” visitado feb. de 2026. dirección: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[6] T. Gingold. “GHDL — Open-source analyzer, compiler, simulator and synthesizer for VHDL,” visitado feb. de 2026. dirección: https://ghdl.github.io/ghdl/

[7] T. Bybell. “GTKWave Electronic Waveform Viewer,” visitado feb. de 2026. dirección: https://gtkwave.sourceforge.net/

[8] Siemens EDA. “ModelSim — FPGA Simulation,” visitado feb. de 2026. dirección: https://eda.sw.siemens.com/en-US/ic/modelsim/

[9] Escuela Superior de Cómputo. “Catálogo de Trabajos Terminales,” visitado feb. de 2026. dirección: https://www.escom.ipn.mx/

[10] IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE, 2009.

[11] P. J. Ashenden, *The Designer’s Guide to VHDL*, 3.ª ed. Burlington, MA, USA: Morgan Kaufmann, 2008.

[12] IEEE, *IEEE Standard for Verilog Hardware Description Language*, IEEE, 2006.

[13] D. E. Thomas y P. R. Moorby, *The Verilog Hardware Description Language*, 5.ª ed. New York, NY, USA: Springer, 2002.

[14] GHDL contributors. “GHDL Documentation: Backends and Features,” visitado feb. de 2026. dirección: https://ghdl.github.io/ghdl/about.html

[15] S. Williams. “Icarus Verilog,” visitado feb. de 2026. dirección: https://github.com/steveicarus/iverilog

[16] W. Snyder. “Verilator: Open-source SystemVerilog simulator and lint system,” visitado feb. de 2026. dirección: https://www.veripool.org/verilator/

[17] C. Wolf. “Yosys Open SYnthesis Suite,” visitado feb. de 2026. dirección: https://github.com/YosysHQ/yosys

[18] Doulos Ltd. “EDA Playground: Online IDE for HDL simulation,” visitado feb. de 2026. dirección: https://www.edaplayground.com/

[19] H. Wong. “HDLBits: Practice digital hardware design,” visitado feb. de 2026. dirección: https://hdlbits.01xz.net/

[20] M. Materzok, “DigitalJS: a Visual Verilog Simulator for Teaching,” en *Proc. 8th Computer Science Education Research Conference (CSERC ’19)*, ACM, 2020. DOI: 10.1145/3375258.3375272

[21] C. Burch, “Logisim: a graphical system for logic circuit design and simulation,” *Journal on Educational Resources in Computing (JERIC)*, vol. 2, n.º 1, págs. 5-16, mar. de 2002. DOI: 10.1145/545197.545199

[22] Logisim Evolution contributors. “Logisim-evolution: Digital logic design tool and simulator,” visitado feb. de 2026. dirección: https://github.com/logisim-evolution/logisim-evolution

[23] Microsoft. “.NET Multi-platform App UI (.NET MAUI) documentation,” visitado feb. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/maui/

[24] Microsoft. “ASP.NET Core Blazor documentation,” visitado feb. de 2026. dirección: https://learn.microsoft.com/en-us/aspnet/core/blazor/

[25] S. Brown y Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3.ª ed. New York, NY, USA: McGraw-Hill, 2009.

[26] S. A. Edwards, “Design and Verification Languages,” Columbia University, New York, NY, USA, inf. téc., 2004. visitado feb. de 2026. dirección: https://www.cs.columbia.edu/~sedwards/papers/edwards2004design.pdf

[27] IEEE, *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*, IEEE, 2018.

[28] H. Foster, “Part 6: The 2022 Wilson Research Group Functional Verification Study,” *Verification Horizons*, nov. de 2022. visitado feb. de 2026. dirección: https://blogs.sw.siemens.com/verificationhorizons/

[29] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. Hoboken, NJ, USA: John Wiley & Sons, 2008.

[30] IEEE, *IEEE Standard for Standard SystemC® Language Reference Manual*, IEEE, 2023.

[31] J. Decaluwe. “MyHDL: From Python to Silicon,” visitado abr. de 2026. dirección: https://www.myhdl.org/

[32] Amaranth HDL contributors. “Amaranth HDL: A modern hardware definition language,” visitado abr. de 2026. dirección: https://amaranth-lang.org/

[33] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek y K. Asanović, “Chisel: Constructing Hardware in a Scala Embedded Language,” en *Proc. 49th Annual Design Automation Conference (DAC ’12)*, ACM, 2012, págs. 1212-1221. DOI: 10.1145/2228360.2228584

[34] C. Papon. “SpinalHDL: A Scala based HDL,” visitado abr. de 2026. dirección: https://github.com/SpinalHDL/SpinalHDL

[35] Arvind, “Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification,” en *Proc. 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003)*, IEEE, 2003. DOI: 10.1109/MEMCOD.2003.1210092

[36] S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, n.º 3, págs. 318-331, 2015. DOI: 10.1109/JPROC.2015.2392104

[37] Xilinx, Inc. "7 Series FPGAs Data Sheet: Overview (DS180)," visitado abr. de 2026. dirección: https://docs.amd.com/v/u/en-US/ds180_7Series_Overview

[38] Advanced Micro Devices. "AMD Completes Acquisition of Xilinx," visitado abr. de 2026. dirección: https://www.amd.com/en/newsroom/press-releases/2022-2-14-amd-completes-acquisition-of-xilinx.html

[39] Intel Corporation. "Cyclone IV Device Handbook," visitado abr. de 2026. dirección: https://www.intel.com/content/www/us/en/docs/programmable/683375/current/device-handbook.html

[40] Intel Corporation. "Intel Completes Acquisition of Altera," visitado abr. de 2026. dirección: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[41] Altera Corporation. "Altera Launches as a Standalone FPGA Company," visitado abr. de 2026. dirección: https://www.altera.com/about/company-overview.html

[42] Lattice Semiconductor. "Lattice Semiconductor Completes Acquisition of SiliconBlue Technologies," visitado abr. de 2026. dirección: https://www.latticesemi.com/About/Newsroom/PressReleases/2011/12/LatticeCompletesAcquisitionofSiliconBlue

[43] Lattice Semiconductor. "iCE40 LP/HX Family Data Sheet," visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/iCE40

[44] Lattice Semiconductor. "ECP5 and ECP5-5G Family Data Sheet," visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/ECP5

[45] Microsoft. "C# documentation," visitado mar. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/csharp/

[46] Microsoft. "What's new in .NET 9," visitado mar. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-9/overview

[47] D. R. Hipp. "SQLite: A self-contained, serverless, zero-configuration, transactional SQL database engine," visitado mar. de 2026. dirección: https://www.sqlite.org/

[48] F. Krueger. "sqlite-net: Simple, powerful, cross-platform SQLite client and ORM for .NET," visitado mar. de 2026. dirección: https://github.com/praeclarum/sqlite-net

[49] Xilinx, Inc. "Spartan-6 Family Overview (DS160)," visitado abr. de 2026. dirección: https://docs.amd.com/v/u/en-US/ds160

[50] Lattice Semiconductor. "iCE40 UltraPlus Family Data Sheet," visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/iCE40UltraPlus

Apéndice A

Anexos

A.1. Ejemplo de Código VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_Gate is
    Port (
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Y : out STD_LOGIC
    );
end AND_Gate;

architecture Behavioral of AND_Gate is
begin
    Y <= A and B;
end Behavioral;
```

Listing A.1: *Ejemplo de una compuerta AND en VHDL.*

A.2. Ejemplo de Código Verilog

```
module and_gate (
    input  wire a,
    input  wire b,
    output wire y
);
    assign y = a & b;
endmodule
```

Listing A.2: *Ejemplo de una compuerta AND en Verilog.*

A.3. Capturas Complementarias del Prototipo

Este anexo agrupa las capturas del prototipo funcional de Kmila que complementan los mockups presentados en la Sección 5.6. Se incluyen vistas del resto de la aplicación: la pantalla de inicio, el selector de idioma y su internacionalización, los dos temas visuales, pestañas adicionales del panel de simulación, la vista de configuración avanzada y el módulo de aprendizaje. Todas las capturas provienen de la versión web del prototipo (`localhost:5297`) el 21 de abril de 2026.

A.3.1. Pantalla de Inicio

Figura A.1: *Pantalla de inicio de Kmila: presenta el nombre del sistema, el lema del proyecto, la mascota (*Kmila*) y los accesos directos a la gestión de proyectos y al módulo de aprendizaje.*

Figura A.2: *Sección* Características Principales *de la pantalla de inicio: visualización dinámica de variables, multiplataforma, autonomía sin conexión, función pre-laboratorio y accesibilidad.*

A.3.2. Internacionalización (7 idiomas)

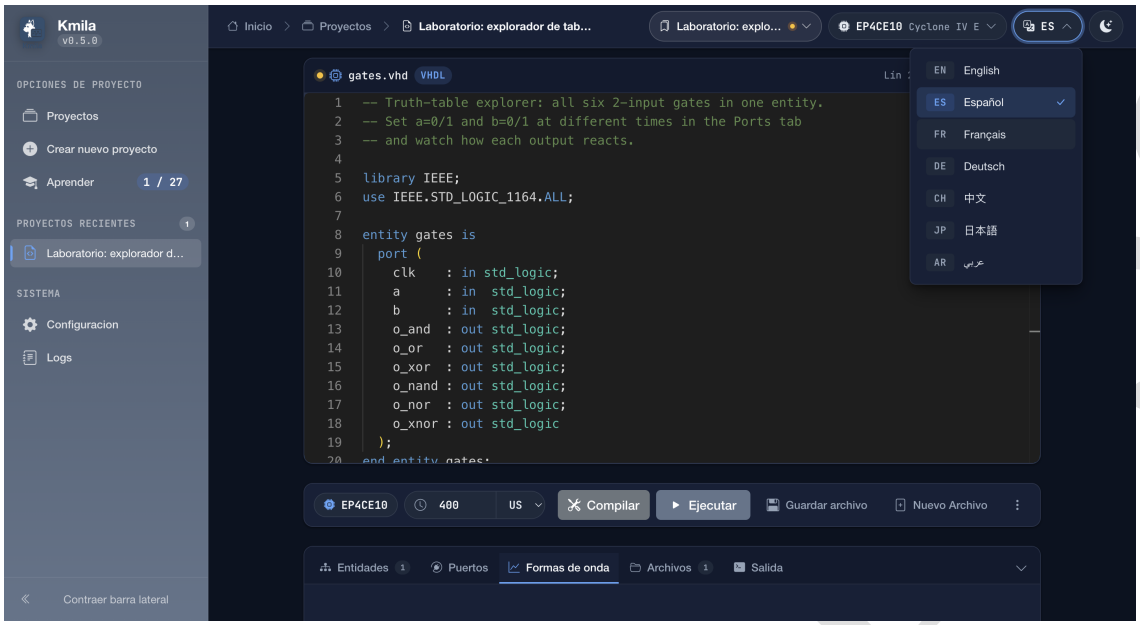

Figura A.3: *Selector de idioma en la barra superior: los siete idiomas soportados por Kmila (Inglés, Español, Francés, Alemán, Chino simplificado, Japonés y Árabe) con sus nombres nativos.*

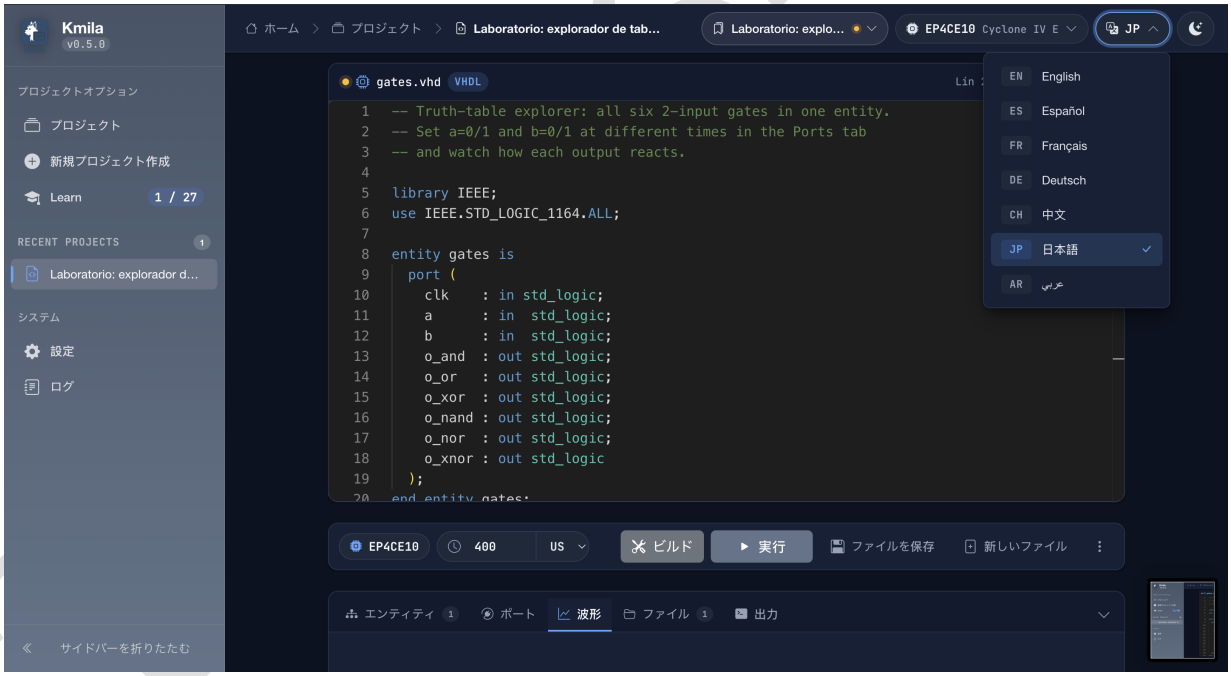

Figura A.4: *Interfaz completa en japonés (*Nihongo*): demuestra la integración reactiva del cambio de idioma sin recarga de página. Todos los rótulos de la barra lateral, barra superior, botones de acción y pestañas del panel inferior se traducen simultáneamente a partir del diccionario JSON correspondiente.*

A.3.3. Temas Visuales (claro y oscuro)

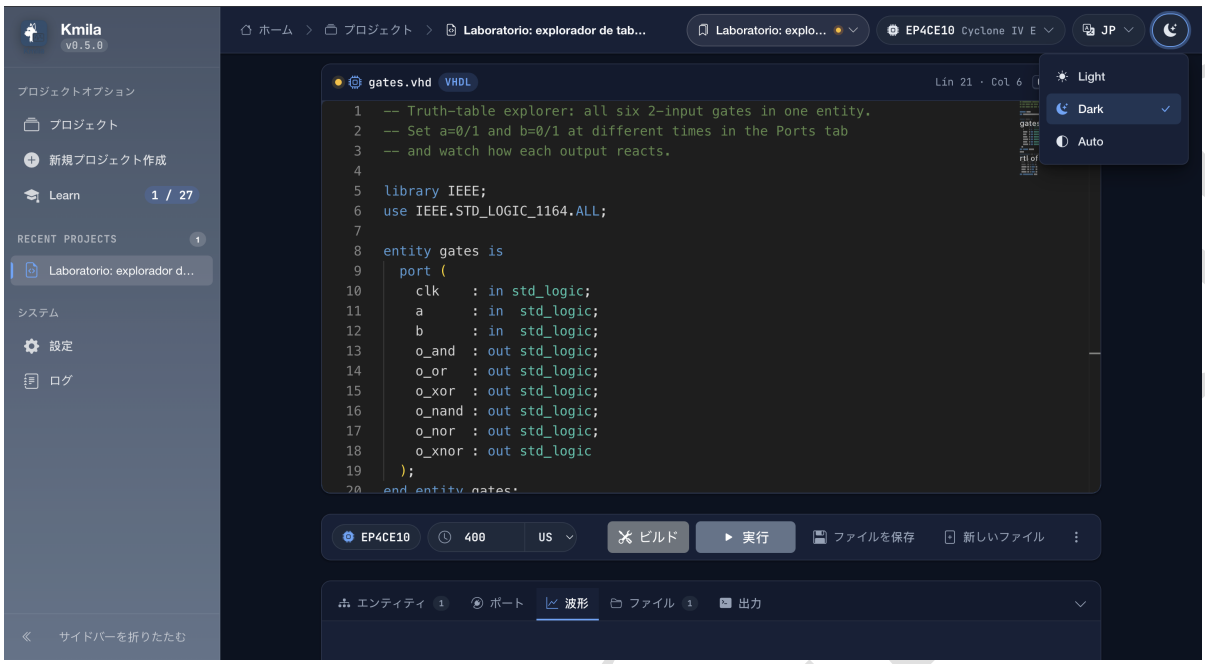

Figura A.5: *Selector de tema visual mostrando las tres opciones disponibles (*Light, Dark, Auto*) con el tema* Dark *activo, configuración por defecto.*

Figura A.6: *Aplicación con el tema* Light *aplicado: todos los componentes (Monaco, paneles, modales, tooltips) se adaptan a la paleta clara.*

A.3.4. Pestaña Entidades del Panel de Simulación

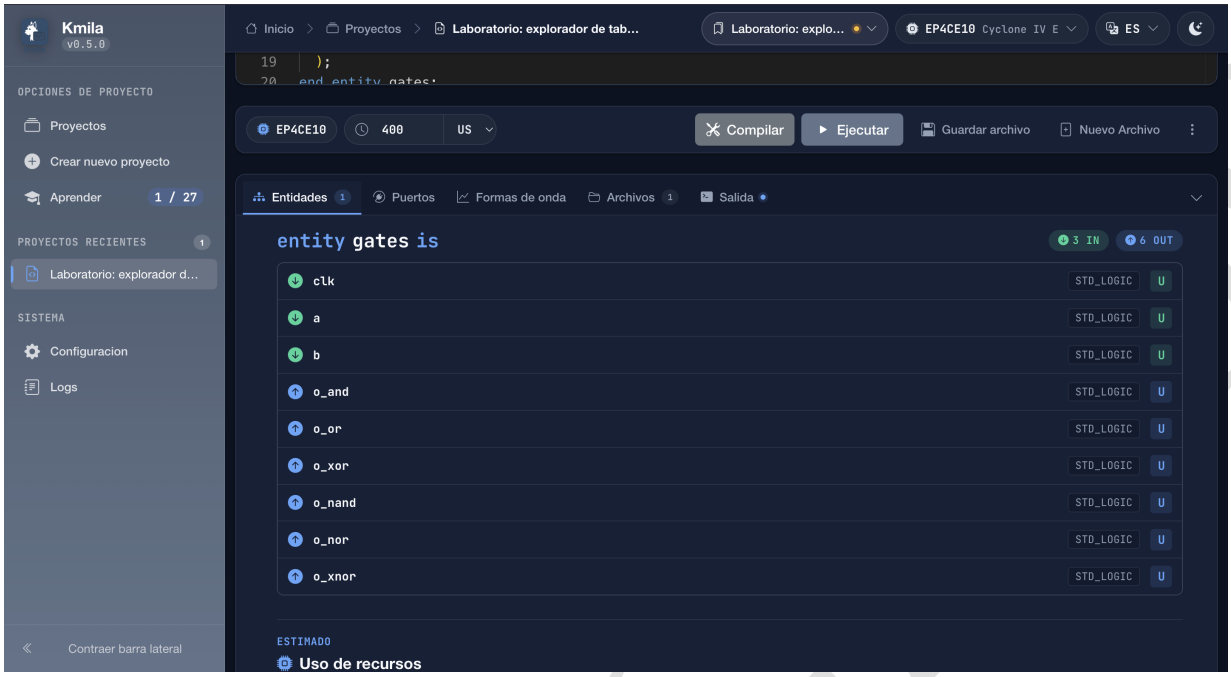

Figura A.7: *Pestaña* Entidades *del panel inferior del editor: muestra la firma de la entidad* `gates` *detectada por el analizador sintáctico, con tres señales de entrada (*`clk`*,* `a`*,* `b`*) y seis señales de salida, todas de tipo* `STD_LOGIC`*. El panel es consumido por el resto de pestañas (Puertos, Formas de onda) para conocer qué señales pueden estimularse o visualizarse.*

A.3.5. Configuración Avanzada

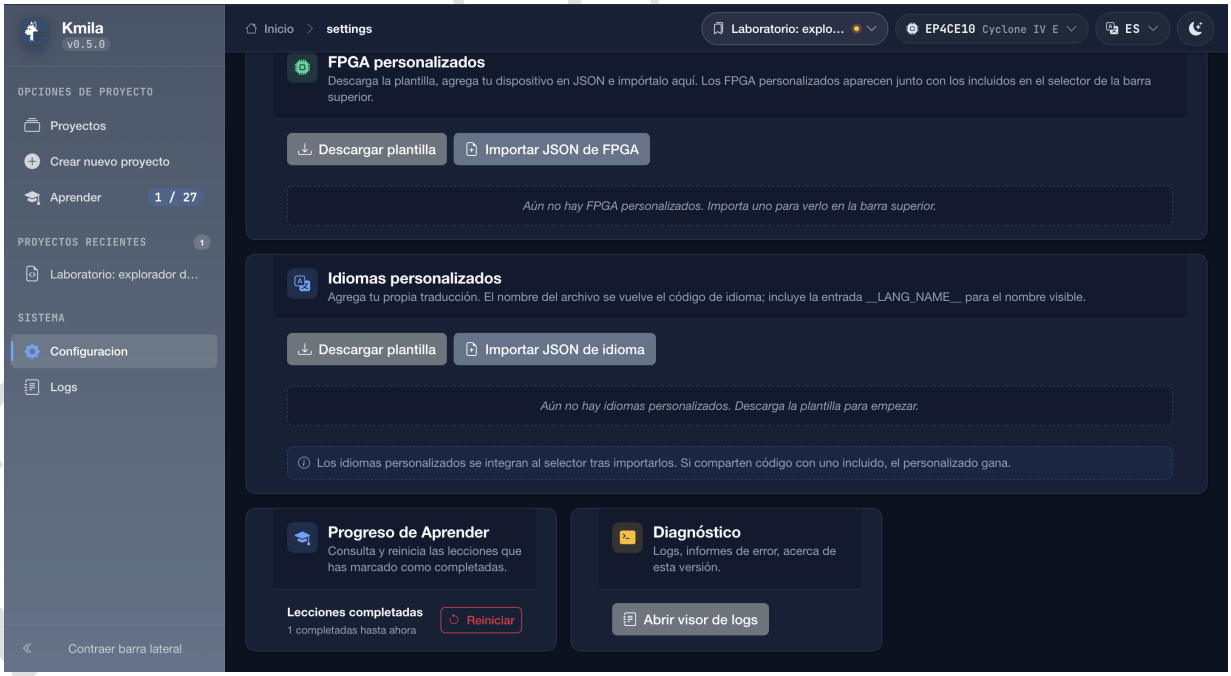

Figura A.8: *Vista de* Configuración*: permite al usuario extender el catálogo mediante la importación de archivos JSON de FPGAs personalizados (plantilla descargable incluida) y diccionarios JSON de idiomas personalizados (con la clave reservada* `__LANG_NAME__` *para el nombre visible). También incluye acceso rápido al progreso del módulo de aprendizaje y al visor de logs.*

A.3.6. Módulo de Aprendizaje

El módulo *Aprender* integra un plan de estudios progresivo con 27 lecciones distribuidas en cinco niveles de dificultad. Las lecciones incluyen archivos VHDL de ejemplo ejecutables directamente en el editor.

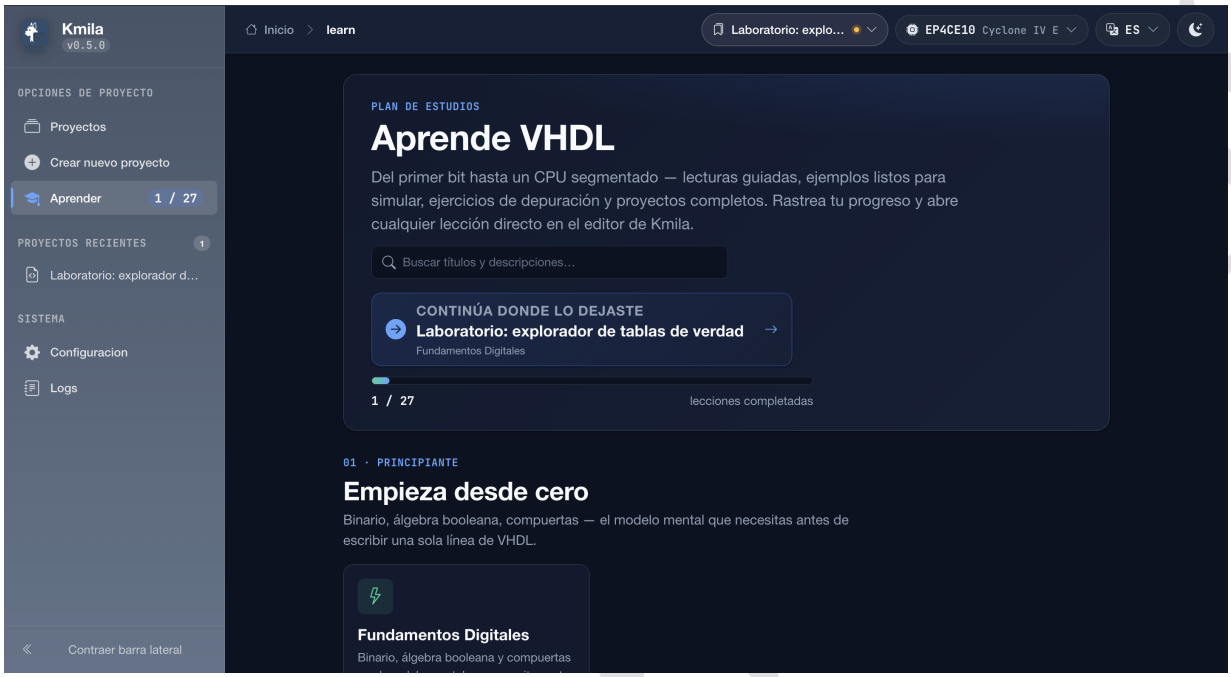

Figura A.9: *Pantalla principal del módulo* Aprender*: presenta el plan de estudios completo, una barra de búsqueda sobre títulos y descripciones, la tarjeta* Continúa donde lo dejaste *(recuperación del último progreso) y un indicador de progreso global (1 de 27 lecciones completadas en esta sesión).*

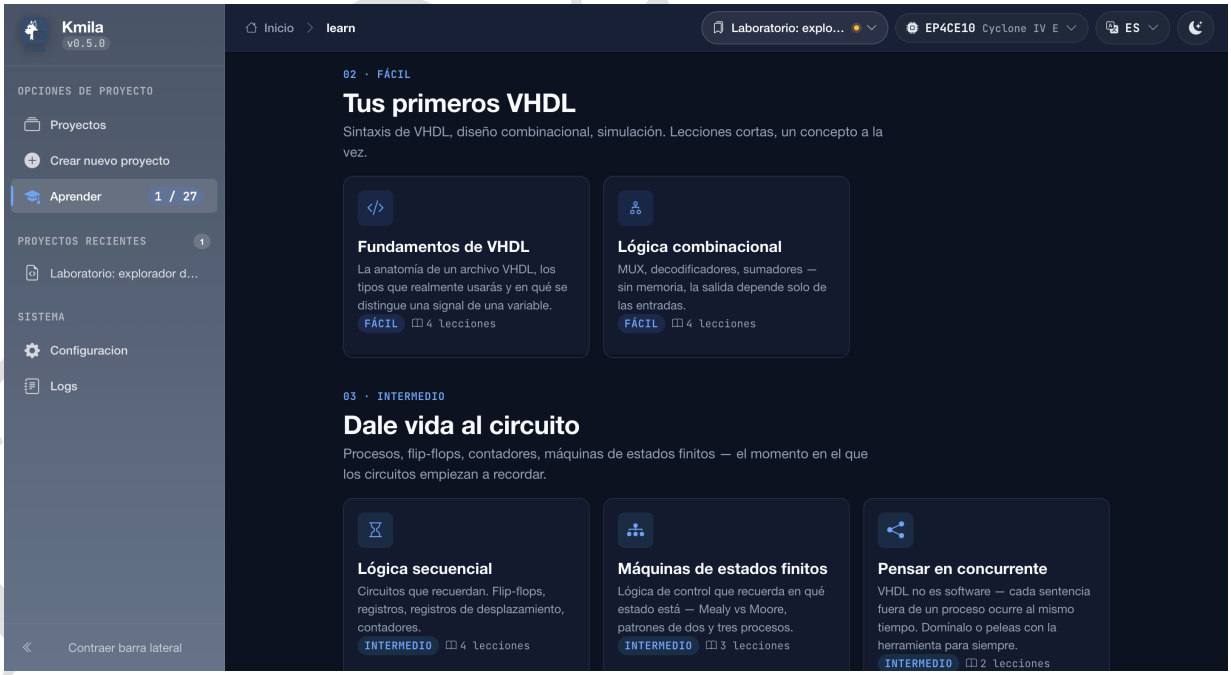

Figura A.10: *Niveles* Fácil *(Fundamentos de VHDL, Lógica combinacional) e* Intermedio *(Lógica secuencial, Máquinas de estados finitos, Pensar en concurrente): cada módulo presenta el número de lecciones y una descripción breve.*

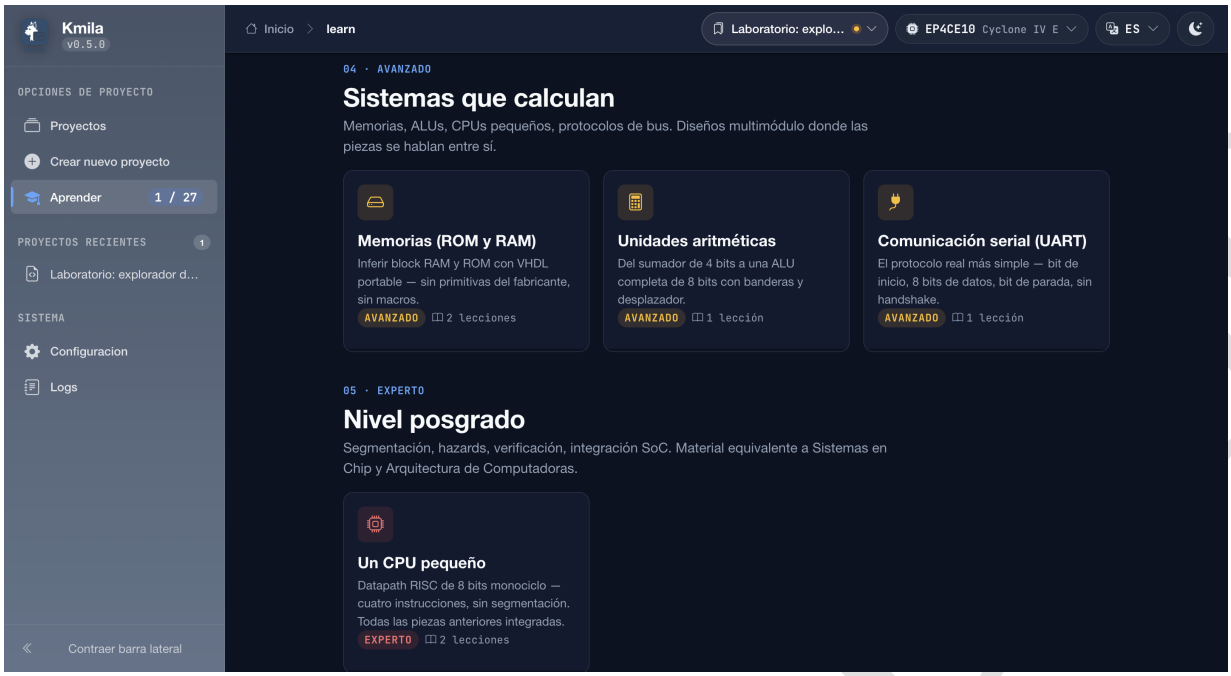

Figura A.11: *Niveles* Avanzado *(Memorias ROM/RAM, Unidades aritméticas, Comunicación serial UART) y* Experto *(Un CPU pequeño basado en datapath RISC de 8 bits). El módulo de aprendizaje complementa los capítulos de diseño digital del plan de estudios de ISC en la ESCOM.*

A.4. Especificación Detallada de Dispositivos FPGA Soportados

Este anexo presenta la especificación completa de los 34 dispositivos FPGA que conforman el catálogo de Kmila, complementando la síntesis presentada en la Sección 2.3.2. La información se distribuye en dos tablas por fabricante para preservar la orientación vertical del documento: la tabla *(A)* agrupa los recursos lógicos, de memoria y aritméticos (DSP); la tabla *(B)* agrupa los recursos de temporización (PLLs, DCM/MMCM), voltaje del núcleo y entrada/salida. Los valores fueron extraídos directamente de los archivos `*.json` del directorio `App/Kmila/Devices/` del repositorio del proyecto, que a su vez se fundamentan en las hojas de datos oficiales de cada fabricante [37], [39], [43], [44], [49], [50].

A.4.1. AMD (Xilinx)

Tabla A.1: *AMD (Xilinx)* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
XC6SLX4	Spartan-6	3 840	Logic Cells	4 800	12	BRAM 18 Kb	216	8	DSP48A1
XC6SLX9	Spartan-6	9 152	Logic Cells	11 440	32	BRAM 18 Kb	576	16	DSP48A1
XC6SLX16	Spartan-6	14 579	Logic Cells	18 224	32	BRAM 18 Kb	576	32	DSP48A1
XC6SLX25	Spartan-6	24 051	Logic Cells	30 064	52	BRAM 18 Kb	936	38	DSP48A1
XC6SLX45	Spartan-6	43 661	Logic Cells	54 576	116	BRAM 18 Kb	2 088	116	DSP48A1
XC6SLX75	Spartan-6	74 637	Logic Cells	93 296	172	BRAM 18 Kb	3 096	180	DSP48A1
XC6SLX100	Spartan-6	101 261	Logic Cells	126 576	268	BRAM 18 Kb	4 824	180	DSP48A1
XC6SLX150	Spartan-6	147 443	Logic Cells	184 304	268	BRAM 18 Kb	4 824	180	DSP48A1
XC7A12T	Artix-7	12 800	Logic Cells	16 000	20	BRAM 36 Kb	720	40	DSP48E1
XC7A15T	Artix-7	16 640	Logic Cells	20 800	25	BRAM 36 Kb	900	45	DSP48E1
XC7A25T	Artix-7	23 360	Logic Cells	29 200	32	BRAM 36 Kb	1 170	80	DSP48E1
XC7A35T	Artix-7	33 280	Logic Cells	41 600	50	BRAM 36 Kb	1 800	90	DSP48E1
XC7A50T	Artix-7	52 160	Logic Cells	65 200	75	BRAM 36 Kb	2 700	120	DSP48E1
XC7A75T	Artix-7	75 520	Logic Cells	94 400	105	BRAM 36 Kb	3 780	180	DSP48E1
XC7A100T	Artix-7	101 440	Logic Cells	126 800	135	BRAM 36 Kb	4 860	240	DSP48E1
XC7A200T	Artix-7	215 360	Logic Cells	269 200	365	BRAM 36 Kb	13 140	740	DSP48E1

Tabla A.2: *AMD (Xilinx)* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
XC6SLX4	2	4	1080	1.2	132	4
XC6SLX9	2	4	1080	1.2	200	4
XC6SLX16	2	4	1080	1.2	232	4
XC6SLX25	4	8	1080	1.2	266	4
XC6SLX45	4	8	1080	1.2	358	4
XC6SLX75	6	12	1080	1.2	498	6
XC6SLX100	6	12	1080	1.2	498	6
XC6SLX150	6	12	1080	1.2	576	6
XC7A12T	5	5	800	1.0	170	—
XC7A15T	5	5	800	1.0	250	—
XC7A25T	5	5	800	1.0	250	—
XC7A35T	5	5	800	1.0	250	5
XC7A50T	5	5	800	1.0	250	5
XC7A75T	6	6	800	1.0	300	—
XC7A100T	6	6	800	1.0	500	6
XC7A200T	10	10	800	1.0	500	10

El valor "—" indica que el descriptor JSON del catálogo no especifica el número de bancos de I/O para ese dispositivo.

A.4.2. Intel (Altera)

Tabla A.3: *Intel (Altera)* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
EP4CE6	Cyclone IV E	6 272	LEs	6 272	30	M9K 9 Kb	270	15	Mult. 18x18
EP4CE10	Cyclone IV E	10 320	LEs	10 320	46	M9K 9 Kb	414	23	Mult. 18x18
EP4CE15	Cyclone IV E	15 408	LEs	15 408	56	M9K 9 Kb	504	56	Mult. 18x18
EP4CE22	Cyclone IV E	22 320	LEs	22 320	66	M9K 9 Kb	594	66	Mult. 18x18
EP4CE30	Cyclone IV E	28 848	LEs	28 848	66	M9K 9 Kb	594	66	Mult. 18x18
EP4CE40	Cyclone IV E	39 600	LEs	39 600	126	M9K 9 Kb	1 134	116	Mult. 18x18
EP4CE55	Cyclone IV E	55 856	LEs	55 856	260	M9K 9 Kb	2 340	154	Mult. 18x18
EP4CE75	Cyclone IV E	75 408	LEs	75 408	305	M9K 9 Kb	2 745	200	Mult. 18x18
EP4CE115	Cyclone IV E	114 480	LEs	114 480	432	M9K 9 Kb	3 888	266	Mult. 18x18

Tabla A.4: *Intel (Altera)* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
EP4CE6	2	0	1300	1.2	179	8
EP4CE10	2	0	1300	1.2	179	8
EP4CE15	4	0	1300	1.2	343	8
EP4CE22	4	0	1300	1.2	153	8
EP4CE30	4	0	1300	1.2	532	8
EP4CE40	4	0	1300	1.2	532	8
EP4CE55	4	0	1300	1.2	374	8
EP4CE75	4	0	1300	1.2	426	8
EP4CE115	4	0	1300	1.2	528	8

Los dispositivos Cyclone IV E no incorporan DCMs ni MMCMs (gestión avanzada de reloj exclusiva de arquitecturas Xilinx); la generación de señales de reloj se realiza mediante los PLLs nativos. **Paquetes disponibles por modelo:** EP4CE6: E144, F256, U256, M164; EP4CE10: E144, F256, U256; EP4CE15: F256, U256, F484; EP4CE22: E144, F256, U256; EP4CE30: F484, F780; EP4CE40: F484, F780; EP4CE55: U484, F484, F780; EP4CE75: U484, F484, F780; EP4CE115: F484, F780.

A.4.3. Lattice Semiconductor

Tabla A.5: *Lattice* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
iCE40HX1K	iCE40 HX	1 280	LUTs	1 280	16	EBR 4 Kb	64	0	N/A
iCE40HX4K[†]	iCE40 HX	—	LUTs	—	—	EBR 4 Kb	—	0	N/A
iCE40HX8K	iCE40 HX	7 680	LUTs	7 680	32	EBR 4 Kb	128	0	N/A
iCE40UP3K	iCE40 UltraPlus	2 800	LUTs	2 800	20	EBR 4 Kb	80	4	MAC 16x16
iCE40UP5K	iCE40 UltraPlus	5 280	LUTs	5 280	30	EBR 4 Kb	120	8	MAC 16x16
LFE5U-12	ECP5	12 000	LUTs	12 000	32	EBR 18 Kb	576	12	Mult. 18x18
LFE5U-25	ECP5	24 000	LUTs	24 000	56	EBR 18 Kb	1 008	28	Mult. 18x18
LFE5U-45	ECP5	44 000	LUTs	44 000	108	EBR 18 Kb	1 944	72	Mult. 18x18
LFE5U-85	ECP5	84 000	LUTs	84 000	208	EBR 18 Kb	3 744	156	Mult. 18x18

Tabla A.6: *Lattice* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
iCE40HX1K	1	0	275	1.2	96	—
iCE40HX4K[†]	—	0	275	1.2	178	—
iCE40HX8K	1	0	275	1.2	206	—
iCE40UP3K	1	0	275	1.2	39	3
iCE40UP5K	1	0	275	1.2	39	3
LFE5U-12	2	2	—	1.1	197	7
LFE5U-25	2	2	—	1.1	245	7
LFE5U-45	4	4	—	1.1	259	7
LFE5U-85	4	4	—	1.1	365	8

[†] El iCE40HX4K comparte die con el iCE40HX8K; Lattice no publica cifras oficiales de LUTs, FFs ni BRAM efectiva para esta variante. Los dispositivos ECP5 no reportan frecuencia máxima de PLL en el catálogo del software. La sigla EBR corresponde a *Embedded Block RAM*.