Trabajo Fin de Máster
"Máster Universitario en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos"

Validación del flujo de diseño de sistemas on-chip basados en RISC-V mediante herramientas de código abierto dentro del programa chipIgnite de eFabless

Alumno: Galán Benítez, Ismael

Tutores: Carmona Galán, Ricardo y De la Rosa Utrera, José Manuel

06 – 11 – 2023

Índice de contenidos

Lista de acrónimos

ABI	Application Binary Interface
ALU	Aritmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
CAD	Computer-Aided Design
CMOS	Complementary Metal-Oxide-Semiconductor
CTS	Clock Tree Synthesis
CVC	Circuit Validity Checker
DFT	Design For Testing
DRC	Design Rule Checking
DUT	Device Under Test
EDA	Electronic Design Automation
ERC	Electrical Rule Check
FOSS	Free and Open-Source Software
GDSII	Graphic Database System
HDL	Hardware Description Language
IDM	Integrated Device Manufacturer
IIC	Institute for Integrated Circuits
ISA	Instruction Set Architecture
LEC	Logic Equivalence Checking
LEF	Library Exchage Format
LVS	Layout Versus Schematic
OSIC	Open-Source Integrated Circuits

PDK	Process Design Kit
RAM	Random Access Memory
RISC	Reduced Set INstruction Computer
ROM	Read Only Memory
RTL	Register-Transfer Level
SCI	Scan Chain Insertion
SDF	Standard Delay Format
SPEF	Standard Parasitic Exchange Format
STA	Static Timing Analysis
TCL	Tool Command Language
TSMC	Taiwan Semiconductor Manufacturing Company
USD	United States Dollar
VHDL	Very high speed integrated circuit Hardware Description Language
VSLI	Very Large-Scale Integration

1. Resumen

Este trabajo se ha elaborado con el propósito de facilitar el acceso de nuevos usuarios al desarrollo de circuitos microelectrónicos. Siendo conscientes de que las herramientas de diseño automático que dominan el sector no están al alcance adquisitivo de cualquier persona, se pretende encontrar y validar alternativas dentro de los programas de código abierto.

Tras una breve exposición del paradigma actual de la industria, entraremos en detalle sobre las características del proceso de diseño y el desarrollo de circuitos integrados desde su descripción en lenguaje a nivel de hardware hasta la extracción del *layout* con el que implementarse físicamente. Revisaremos etapa por etapa el flujo de diseño de un circuito digital y presentaremos los rasgos generales de una de las arquitecturas de procesador de código abierto más popularizadas a día de hoy: el RISC-V. Procederemos a detallar todos y cada uno de los pasos seguidos para desarrollar exitosamente el flujo de diseño, así como se mostrará la instalación y manejo de las herramientas que serán empleadas para lograr tal labor. Todas ellas estarán contenidas en *IIC-OSIC-TOOLS*, un entorno creado por el Instituto de Circuitos Integrados de la Universidad Johannes Kepler, y cuyo programa insignia en el proceso será OpenLane.

Se espera que este documento pueda servir como manual práctico y que cualquier persona interesada pueda ejecutar de manera rápida y lo más sencillamente posible las aplicaciones citadas para obtener su propio circuito. Para demostrar la validez del proceso y los programas que se citarán, adicionalmente hemos desarrollado un procesador básico basado en la arquitectura RISC-V, cuyos archivos de descripción y resultados servirán como ejemplo ilustrativo de todo el contenido tratado.

2. Introducción

La microelectrónica, y en concreto los circuitos integrados, es un sector surgido hace sesenta años, y desde entonces en constante desarrollo y evolución. Actualmente, es complicado encontrar un ámbito en el que no tenga cabida, hasta haber llegado el punto en el que resulta imprescindible para mantener el modelo de sociedad en el que vivimos. Teléfonos móviles, ordenadores, televisores, electrodomésticos, automóviles, radares aéreos, equipos de diagnóstico médico, maquinaria pesada, etc., solo son algunos de los dispositivos en los que hay implementados circuitos integrados y que han revolucionado sectores como el de la comunicación, el procesado de datos, la industria, la medicina, y muchos otros.

Para reconocer el verdadero peso que tienen estas tecnologías en el mercado, nos remitimos a la facturación anual de la industria de semiconductores. En la Figura 1, además de retratarse el progreso del sector en la última década, se refleja en el año 2022 un máximo histórico de 639.220 millones USD. Ese mismo año, las tres empresas de circuitos integrados con mayores ingresos del mundo fueron Samsung Electronics (65.585 millones USD), Intel (58.373 millones USD) y SK Hynix (36.229 millones USD) [1]. Estos tres ejemplos de empresas se reconocen como fabricantes de dispositivos integrados (IDM), es decir, que diseñan y producen sus propios productos. A diferencia ellas, otros modelos de empresa son las "*fabless*", como Apple o Nvidia, que tan solo se ocupan del diseño de la tecnología; y las "*pure play foundries*", como TSMC, que se limitan únicamente a la fabricación por encargo de máscaras y a la litografía de los chips en semiconductores. En cualquier caso, la cadena completa del proceso exige un enorme coste, refiriéndose tanto a la adquisición de la licencia de un EDA como a la manufacturación del producto final[1].

El concepto de EDA se refiere a un segmento de software y servicios con el propósito de asistir en la definición, planificación, diseño, implementación, verificación y la subsecuente manufacturación de chips. Estas herramientas juegan un papel fundamental durante el flujo de desarrollo, pues son utilizadas para diseñar y validar el proceso de fabricación. Garantizan que se satisfacen especificaciones imprescindibles para la manufacturación, y se cumplen todas las condiciones necesarias de rendimiento y densidad de elementos. Actualmente, tres empresas son las que lideran el sector de los EDA: Synopsys, Cadence y Mentor [3].

[1] Para TSMC, se estima que el precio de venta de una oblea litografiada en la tecnología de 5 nm es de 16.988 USD [2].

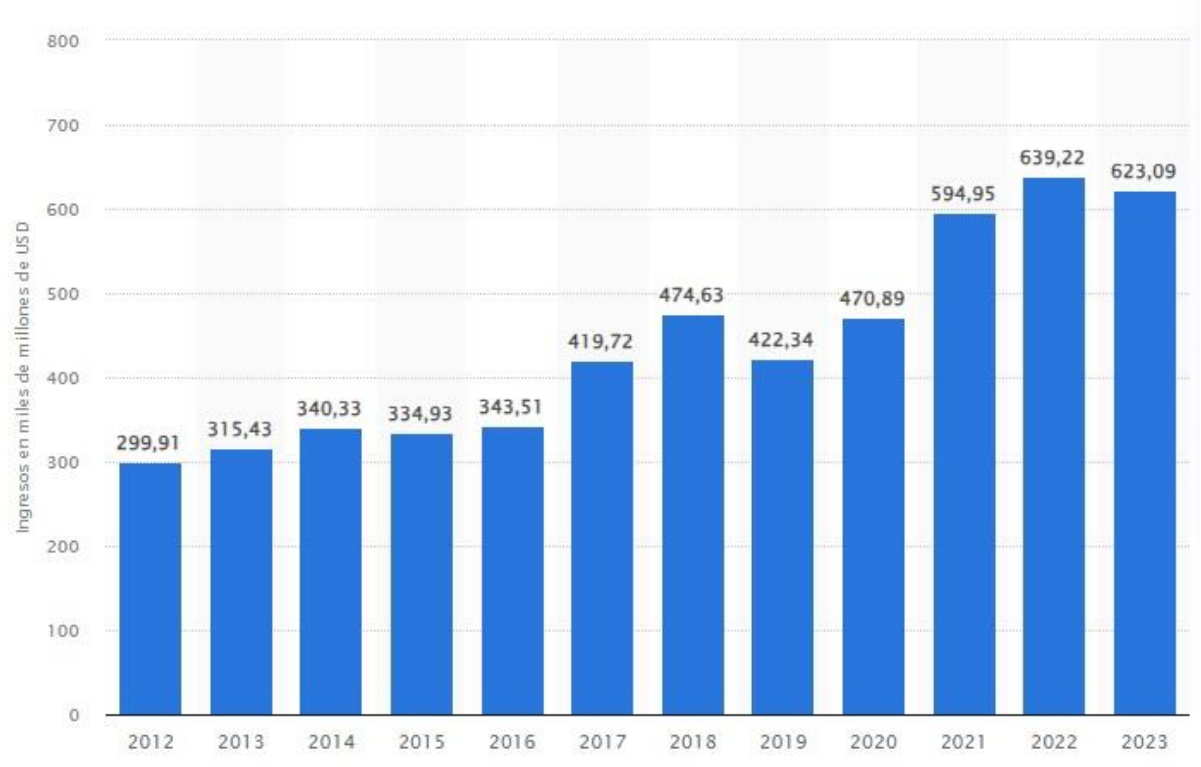

Figura 1. Ingresos de la industria de semiconductores a nivel mundial entre 2012 y 2023.
Fuente: https://es.statista.com/estadisticas/600812/facturacion-de-la-industria-de-semiconductores-a-nivel-mundial/

Ahora bien, considerando la relevancia para el proceso de fabricación y los beneficios de las empresas que los requieren, no debe de sorprender que la licencia de uso de uno de estos EDA pueda costar cientos de miles de dólares[2]. Es por ello, y junto a los costes de fabricación, que a pequeñas empresas que estén tratando de entrar en este mercado, o estudiantes y usuarios que pretenden formarse sobre el proceso, les resulte complicado poder iniciar un proyecto. No obstante, a día de hoy existen alternativas a los EDA de estas empresas que están orientadas por la filosofía del código abierto. Distintos proyectos e iniciativas, como *Open Circuit Design* o *eFabless*, se basan en principios de colaboración y libre distribución para tratar de facilitar e incentivar el desarrollo de circuitos integrados —y muy importante— de forma gratuita.

De entre todas las opciones valoradas para este trabajo, nuestro interés será puesto en estudiar *IIC-OSIC-TOOLS* [5]: un entorno desarrollado por el Instituto de Circuitos Integrados de la Universidad Johannes Kepler Linz (Austria), y que está basado en *FOSS-ASIC-TOOLS* [6] creado por efables.com. Se trata de una imagen para un contenedor Docker en el que se encuentran preinstaladas multitudes de herramientas CAD de código abierto con las que el usuario puede completar un flujo de diseño tanto digital como analógico. Algunas de estas

[2] Por ejemplo, una licencia anual para toda la tecnología de compilación de circuitos integrados de Synopsys ha estado valorada por 735.000 USD [4].

aplicaciones son más conocidas que otras, como Icarus Verilog (*Iverilog*), Magic, KLayout, o Yosys. No obstante, veremos que la que tiene mayor peso en nuestro desarrollo es OpenLane: una herramienta que combina la ejecución de otras tantas para crear un flujo automatizado desde el nivel RTL al GDSII. Además, el entorno cuenta también con los procesos SKY130, un PDK de la tecnología nodal de 130 nm de la *foundry* SkyWater Technology que han sido liberadas como código abierto en colaboración con Google [7]. Aunque esta tecnología pueda resultar desfasada con respecto a los nodos de 5 nm actuales, son unos procesos de extremada flexibilidad y utilidad para propósitos estándar como los nuestros.

IIC-OSIC-TOOLS cuenta con hasta 50 herramientas CAD, todas ellas creadas por distintos desarrolladores y equipos, cada una con propósitos específicos, diferentes ritmos de actualización, distintas formas de ejecución, algunas menos intuitivas en su uso, …; pero todas ellas con un manual completo propio. La cantidad total de información y contenidos necesaria para ejecutar el flujo de diseño completo puede resultar abrumante para un usuario inexperimentado o no familiarizado con el entorno de trabajo, y es a este perfil al que se tratará de asistir con este proyecto.

El propósito de esta investigación será el de indagar entre todo el material disponible y estructurar una guía con la que un usuario que nunca antes haya trabajado con *IIC-OSIC-TOOLS* pueda rápidamente saber cómo ejecutar, utilizar y encadenar las aplicaciones disponibles para completar el flujo de diseño digital de cualquier proyecto propio. Nuestra intención no será la de demostrar exhaustivamente las características del software, sino la de exponer las posibilidades que se ofrecen y agilizar la actividad del usuario en sus primeros diseños. Desarrollaremos los distintos niveles que componen el flujo de diseño digital, analizando brevemente y encada uno de ellos las herramientas empleadas con tal motivo. Finalmente, demostraremos la validez de este entorno para el flujo de diseño digital mediante la construcción de un modelo de procesador básico basado en la arquitectura RISC-V, y que al mismo tiempo servirá como manual rápido de ejecución.

3. Revisión de conceptos teóricos

3.1. Tipos de circuitos integrados

A la hora de construirse un circuito este puede ser catalogado como digital, analógico o de señal mixta. Aunque todos ellos siguen los mismos principios físicos y se constituyen por los mismos elementos básicos y materiales, se diferencian entre ellos por cómo es tratada la señal eléctrica. En el caso de los circuitos analógicos, el estímulo es analizado como una señal continua en el tiempo, de modo que el comportamiento del circuito es modelado en los dominios del tiempo y la frecuencia, atendiendo a la precisión, consistencia y rendimiento de las formas de onda resultantes. Por otro lado, los circuitos digitales consideran la señal de estímulo como una serie de valores lógicos: "unos" y "ceros", que representan la presencia y ausencia de corriente eléctrica, respectivamente [8]. Y aquellos que se consideran de señal mixta combinan los dos tipos de circuitos anteriores en un único semiconductor.

Así mismo, cada uno de los dos principales circuitos descritos en el párrafo anterior presentan distintos niveles de abstracción. Mientras que los diseños digitales tienen como entidad mínima las puertas lógicas, en los diseños analógicos se consideran hasta los transistores y se especifican en detalle sus propiedades y respuestas. Aunque existen excepciones, por lo general los diseños analógicos cuentan una menor cantidad de transistores que con respecto a los digitales, permitiendo ser viable que un diseñador se ocupe personalmente el circuito y pueda ajustar cualquier parámetro basándose en su experiencia. En cambio, los diseños digitales cuentan una mayor cantidad de celdas y hasta millones de redes de transistores, por lo que el proceso de diseño está completamente automatizado [9].

Como ya ha sido mencionado en la introducción, el entorno de *IIC-OSIC-TOOLS* dispone de herramientas tanto para desarrollarse un flujo de diseño digital como uno analógico. No obstante, este trabajo se centrará particularmente en el dominio digital, exponiéndose en las siguientes subsecciones las características del mismo flujo.

3.2. Flujo de diseño digital

El flujo de diseño de un circuito digital es un proceso ordenado que implica transformar una serie de especificaciones en bloques digitales y circuitos lógicos, dándose además ciertas restricciones establecidas por las limitaciones del proceso de fabricación de la *foundry* [10]. Estas restricciones se presentan a los diseñadores mediante la concesión de un PDK: una librería

de componentes básicos generados por la propia *foundry* que reúne información técnica y geométrica necesaria para la manufacturación del circuito integrado [11]. Así, un PDK puede interpretarse como el conjunto de celdas estándar, librerías de dispositivos, modelos, configuración de las capas de materiales, etc., que le permiten al diseñador construir una gran variedad de circuitos, simplificar el proceso de diseño al trabajar a nivel de puertas lógicas en lugar de a nivel de transistores, y reducir costes al utilizar componentes ya predefinidos y probados.

Ahora bien, para abordar el diseño de un ASIC —ya sea analógico, digital o mixto—, pueden plantearse dos estrategias: *top-down* y *bottom-up*. En un proceso de diseño *top-down* se parte de una visión del modelo completo sin detallar ninguno de los subsistemas que lo componen. De nuevo, estos subsistemas son subdivididos en otros más simples; y así, sucesivamente, se descompone el sistema completo hasta elementos básicos [12].

Por otro lado, el método *bottom*-up es más tradicional y parte de la elaboración de bloques individuales. Cada uno de ellos es construido a partir de unas características concretas y su correcto funcionamiento es evaluado como unidad independiente. Una vez que todos los bloques son aprobados, son combinados como una sola unidad y nuevamente se prueba el funcionamiento del sistema completo [13]. No obstante, este mecanismo presenta ciertos inconvenientes como que la verificación a nivel de sistema se realiza mediante lentas simulaciones a nivel de transistor, generalmente se pierden mejoras en el mayor nivel, y el tiempo de diseño puede extenderse debido a la multitud de verificaciones necesarias [14] —por lo que resultaría inviable para sistemas demasiado extensos—. Pese a ello, en nuestra práctica recurriremos al proceso *bottom-up*, ya que no plantearemos un diseño lo bastante grande ni complejo como para que estos inconvenientes sean relevantes. Además, nuestro proyecto presenta una estructura definida modularmente, acorde con la arquitectura de RISC-V, y que encaja adecuadamente con la metodología *bottom-up*.

En cualquier caso, el flujo de diseño digital de cualquier ASIC sigue las etapas presentadas en la Figura 2: partiendo de las especificaciones de diseño, se define el sistema en un lenguaje de hardware y se verifica su correcto funcionamiento, se procede a la síntesis en puertas lógicas del modelo anterior, se genera un *netlist* y se realizan nuevas comprobaciones funcionales y también temporales, se procede al *placement & routing* de elementos sobre el semiconductor para dar lugar al *layout* y sobre él se realizan las últimas comprobaciones temporales y de validación del diseño. Aunque el proceso de implementación física tenga su

propia complejidad, llegado a este punto consideraremos el final de nuestro desarrollo. A continuación, se detallarán cada una de las etapas del flujo.

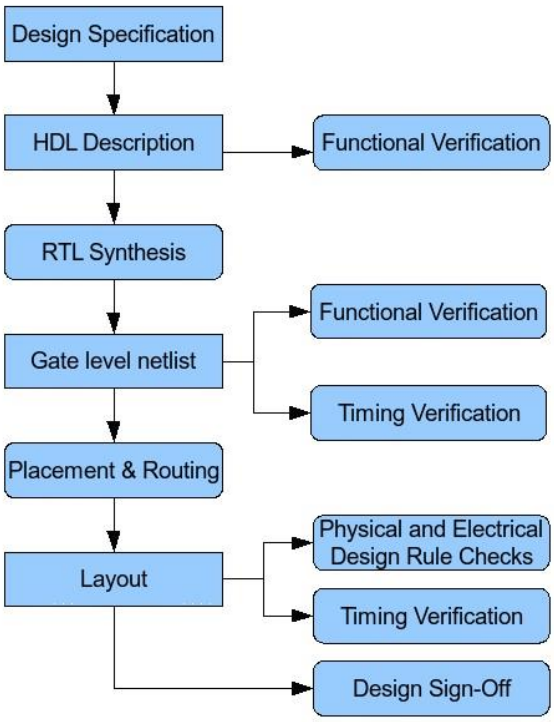

Figura 2. Diagrama del desarrollo del flujo de diseño digital.
Fuente: http://www2.imse-cnm.csic.es/elec_esi/asignat/MHCAD/tema1_digital/Digital_design_flow.html

3.2.1. Especificaciones del diseño

El primer paso a la hora de crear un ASIC es ser consciente de cuál será su aplicación y propósito. Es necesario especificar cuál será la labor que pueda desarrollar el chip, así como sus características estructurales y técnicas con las que definir, respectivamente, bloques con los que hacer más accesible su manejo y desarrollo; y las relaciones e interconexiones entre ellos. Por tanto, es tarea del diseñador recoger las funcionalidades y características del chip para facilitar el desarrollo y que todos los miembros del equipo involucrado tengan la misma visión del proyecto.

3.2.2. Descripción HDL

En este punto del desarrollo se procede a la descripción del diseño a *nivel de transferencia de registros* (RTL), por la cual el circuito se define como un grupo de registros, ecuaciones booleanas y estructuras lógicas de control (p. ej. declaraciones *if-else*); sirviendo de

puente entre descripciones de alto nivel, como algoritmos o especificaciones del sistema, e implementaciones de bajo nivel, como el nivel de puertas [15].

Con el propósito de desarrollar esta tarea, en la década de 1980 surgieron dos *lenguajes de descripción de hardware* (HDL) —siendo todavía a día de hoy los más utilizados—: Verilog y VHDL; especializados en la descripción de estructuras, funcionalidad y temporización de circuitos digitales. Estos lenguajes son independientes de la tecnología en la que el diseño será implementado y tratan de favorecer su documentación, la simulación del comportamiento y la síntesis a hardware. Entre las necesidades que satisfacen, se citan las dadas por [16]:

- Permiten la descripción de la estructura de un sistema de hardware, así como la deconstrucción del sistema completo en bloques y sus interconexiones.
- Permite especificar la funcionalidad del sistema mediante el uso de un lenguaje de programación familiar[3].
- El diseño de un sistema puede ser simulado antes de ser fabricado, lo que permite a los diseñadores comparar alternativas y realizar pruebas sin la demora ni el coste de construir un prototipo de hardware.
- Permite la síntesis de una estructura detallada de diseño partiéndose de unas especificaciones más abstractas, favoreciendo que los diseñadores puedan centrarse en decisiones más estratégicas. También, este proceso automático de síntesis reduce el tiempo de implementación del diseño.

Tal y como se mostraba en el diagrama de la Figura 2, es necesaria verificar la correcta funcionalidad del circuito en esta etapa. Un *testbench* sirve para ello, siendo este un módulo escrito en HDL cuyo propósito es testar otro módulo, llamado *dispositivo bajo test* (DUT) —el diseño en nuestro caso—. En el *testbench* se definen declaraciones para generar estímulos de entrada al DUT y, consecuentemente, poder evaluar la señal de salida como respuesta. El *testbench* es simulado al igual que cualquier otro modulo en HDL, sin embargo, no es sintetizable [17].

[3] Refiriéndose por familiar a que el formato del código es lo suficientemente legible como para poder ser interpretado sin excesiva dificultad por una persona.

3.2.3. Síntesis RTL

En esta etapa —también conocida como síntesis lógica— es un proceso altamente automatizado en el que la descripción desarrollada en HDL es transformada en un circuito lógico de tecnología específica. La herramienta de síntesis comienza desglosando las especificaciones de alto nivel del lenguaje HDL en funciones más simples y las vincula con elementos del PDK concedido por la *foundry*. El programa busca en la librería cuál es la celda adecuada para cada función, y cuando la encuentra la copia en el diseño —la instancia en el circuito— y le da un nombre único; repitiéndose esta acción hasta mapearlas todas al nivel de circuito lógico. Estas librerías incluyen la definición de funcionamiento, área, temporalidad, características de potencia y restricciones de entorno para cada puerta [18].

Generalmente, pueden existir cientos y miles de combinaciones diferentes de circuitos lógicos para implementar la misma función lógica. Es por ello que estas herramientas aplican reglas de optimización y algoritmos con los que hacer más eficiente el proceso de síntesis, siendo posible diferenciar tres niveles [19]:

- **Optimización a nivel de arquitectura**: Sucede en la descripción HDL e incluye tareas de síntesis de alto nivel como reordenar operadores, seleccionar implementaciones, compartir recursos e identificar expresiones aritméticas para la síntesis de la ruta de datos. Estas optimizaciones son llevadas a cabo sobre un diseño todavía no mapeado y se basan en restricciones impuestas y el código creado por el diseñador.
- **Optimización a nivel lógico**: Por un lado, consta de un proceso de estructuración en el que se añaden variables y estructuras lógicas intermedias con el propósito de reducir al área. Por otro lado, un proceso de "aplanamiento" convierte rutas de lógica combinacional en representaciones de suma de productos, favoreciendo la velocidad del circuito.
- **Optimización a nivel de puerta**: Trabaja sobre el *netlist* generado por la síntesis RTL. Afecta al mapeado, seleccionando las librerías adecuadas de la tecnología para que su implementación satisfaga condiciones de temporización y superficie; optimiza los retrasos para tratar de salvar las posibles violaciones de tiempo introducidas por el mapeado; y corrige fallos en las reglas el diseño incorporando *buffers* o reescalando celdas existentes.

El *netlist* que se acaba de mencionar se trata de un archivo que contiene la información sobre los componentes del circuito eléctrico y la interconexión entre ellos, dándose este como

resultado del proceso de síntesis lógica. Tanto Verilog como VHDL continúan siendo lenguajes aptos para esta descripción. Adicionalmente, también es generado un archivo SDF en el que quedan representados los retrasos del circuito.

Ambos ficheros son necesarios para realizar dos nuevos procesos de verificación. Por un lado, se comprueba que el *netlist* continúa satisfaciendo la funcionalidad para la que se ha diseñado el circuito; y por otro, se desarrolla un análisis temporal estático (STA), el cual consiste en la validación del rendimiento temporal del diseño mediante su descomposición en rutas sobre las que se evalúa el retraso en la propagación de una señal tratando de buscar violaciones en las restricciones de tiempo, tanto internamente como entre las interfaces de entrada y salida del diseño [20].

3.2.4. Place & Route[4]

El siguiente paso en el desarrollo de un ASIC consiste en implementar físicamente el contenido del *netlist* generado tras la síntesis lógica. El proceso se denomina como *placement & routing*, y puede ser comprendido en tres fases principales:

- ***Floorplaning***: Es la etapa en la que, primeramente, el diseño es dividido en bloques de circuitos, asignándosele a cada uno de ellos unas dimensiones y área específica. Estos módulos podrán ser clasificados como *hard blocks*, si sus dimensiones son inamovibles, o *soft bloks* si, aunque su superficie esté fijada, se puede modificar su relación de aspecto. Al conjunto completo de módulos junto con su disposición se designa como *floorplan*, que facilita el posicionamiento de las puertas lógicas y garantiza que todo pin con conexión externa esté localizado. Así, el *floorplaning* se considera la primera etapa del diseño físico de cualquier VLSI, y proporciona información temprana con la que considerar modificaciones arquitecturales, estimaciones del área del chip y estimaciones en la congestión y retraso de las señales debido a la interconectividad de los bloques. El *floorplan* puede ser además clasificado como *slicing*, si es posible obterse como resultado de subdividir horizontal o verticalmente formas rectangulares, o *non-slicing* en caso contrario —véase un ejemplo en la Figura 3—.

[4] Los contenidos tratados en este apartado han sido extraídos por igual de [21], [22] y [23].

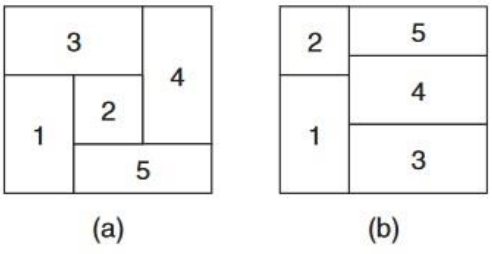

Figura 3. Ejemplificación de non-silicing floorplan (a) y slicing floorplan (b).
Fuente: Electronic Design Automation. Chapter 10 – Floorplaning. (2009, p. 577).

- ***Placement***: Una vez el diseño ha sido particionado, ahora de trata de dar una localización específica en el chip a cada uno de los elementos lógicos que componen el circuito, a la vez que se tratan de alcanzar ciertas metas de optimización. Algunas técnicas consideran un primer *placement global* con el propósito de enfatizar un posicionamiento general y determinar una densidad en la distribución de componentes, por lo que las restricciones de tamaño, forma, alineación, o superposición están más relajadas. Una fase de "legalización" sucede a la anterior para tratar de corregir estas imperfecciones, y un último *placement detallado* refina el proceso mediante operaciones locales, como intercambiar la posición de dos objetos, o el desplazamiento de varios elementos lo suficiente como para dar espacio a otros. Todo ello, además, consideran objetivos como evitar redes de conexión innecesariamente largas y/o excesiva densidad de cables, reducir el retraso en las señales, minimizar el número de redes de corte[5], o distribuir la generación de calor sobre todo el chip homogéneamente.
- ***Routing***: Este es el proceso en el que se generan las rutas de conductores por las que viajarán las señales eléctricas, e interconectan todos los pines eléctricamente equivalentes del circuito. De forma análoga al *placement*, una técnica general distingue dos fases: el *routing global* y el *routing detallado*. El primero divide el sistema por sectores y trata de determinar caminos óptimos para todas las redes. El segundo establece oficialmente los tramos de conductor basándose en los resultados del *global routing* y respetando las normas de diseño. Cada una de las redes tendrá un nivel de prioridad basado en el número de pines, su criticidad temporal, la señal específica de la que es responsable, etc., que será determinante a la hora de decidir que líneas pueden tomar ciertos desvíos y cuáles no, y que afectaran a la longitud y densidad de las redes, o efectos como el *crosstalk*.

[5] Se considera red de corte aquellos cables cuyos pines de conexión se encuentran en regiones diferentes del *floorplan.* Si ambos pines se localizan en la misma sección, la red se clasifica como "no cortada".

3.2.5. *Layout*: Verificación física y temporal

Como resultado del proceso de *place & route* se obtiene el *layout*, que se trata de la descripción geométrica del diseño —para nuestro, caso en formato GDSII—. En él se especifican la posición y las dimensiones de todos los elementos que serán litografiados sobre la oblea de semiconductor, y mediante un código de formas y colores —véase un ejemplo de *layout* de la tecnología SKY130 en la Figura 4— se le permite al fabricante interpretar otras características de la representación, como el tipo de difusión en regiones específicas, el espesor de ciertas capas de óxido, el nivel de ionización de los canales de los transistores, las distintas capas de material, etc. [24]

No obstante, entes de ser enviado al fabricante, el *layout* debe superar una nueva verificación para garantizar el correcto funcionamiento y características pese a los efectos físicos que se introducirán a la hora materializar el diseño. En primer lugar, un *design rule checking* (DRC) sirve para comprobar que se cumplen especificaciones de fabricación necesarias para evitar errores funcionales, como separación entre metales, o anchura de las vías. A continuación, un *layout versus schematic* (LVS) se asegura de que el comportamiento del *layout* equivale al del *netlist* a partir del cual se ha generado la geometría. Para ello, la herramienta de LVS crea un nuevo *netlist* a partir del *layout* y lo compara con el original para detectar, por ejemplo, posibles errores o faltas de conexión [25]. Siendo estas dos las pruebas principales realizadas, se pueden desarrollar a distintos niveles de abstracción para tratar de detectar mayor cantidad de parásitos, sin embargo, el tiempo consumido por las simulaciones será más prolongado.

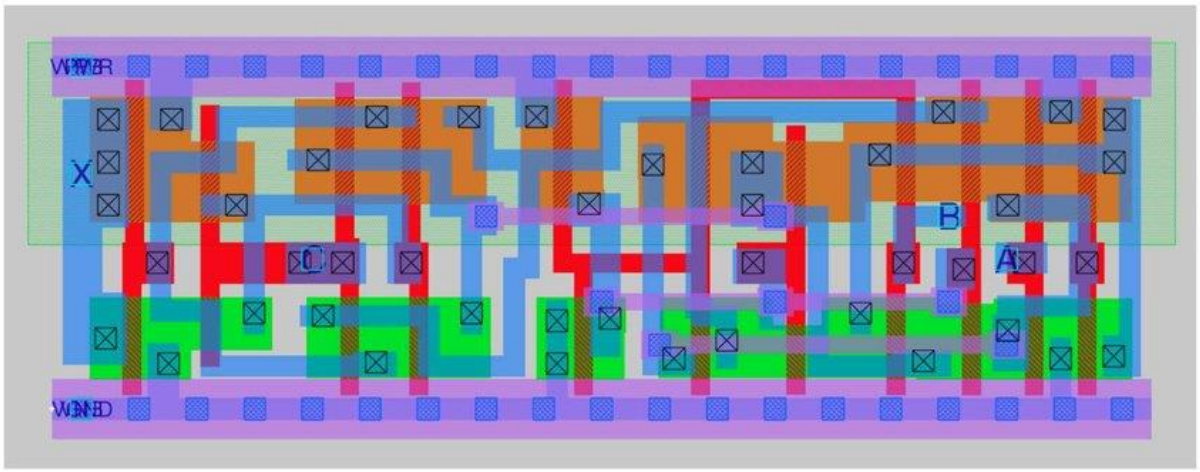

Figura 4. Layout de una celda estándar de la librería de SKY130 visualizada con Magic. Representación de las capas simbólicas. Fuente: https://www.researchgate.net/figure/Standard-Cell-Layout-from-SKY130-Libraries-visualized-in-Magic-Layout-Editor_fig3_347687534

Adicionalmente, se desarrolla un último análisis temporal con el que verificar que efectos como resistencias o capacitancias no afecten excesivamente al retraso de las señales. Si el resultado de todas estas simulaciones es favorable, el diseño puede ser enviado a la *foundry* para proceder a su fabricación. En caso contrario, debe de revisarse el modelo y subsanar los errores modificando la fase de RTL o de *netlist* y repetirse el proceso completo.

3.3. Flujo de diseño de OpenLane

Una vez analizado el flujo de diseño digital estandarizado, presentamos la principal herramienta con la que será desarrollado en esta práctica. OpenLane [26] se trata de un flujo automatizado desde el nivel RTL al GDSII construido a partir de varias aplicaciones EDA de código abierto como OpenROAD, Yosys, y Magic entre muchas otras.

En primer lugar, recurriremos a Icarus Verilog (Iverilog) [27]: un programa compilador de lenguaje Verilog descrito en el estándar IEEE-1364 con el que podremos llevar a cabo la descripción HDL del circuito. Este a su vez, será complementado por GTKWave [28] para poder visualizar la simulación de los *testbenches* necesarios en esta etapa. Se trata de una herramienta de análisis pensada específicamente para la depuración de modelos en Verilog o VHDL.

Ahora, tal y como ilustra la Figura 5, considerando la descripción HDL y el PDK de la tecnología en la que se pretende implementar, OpenLane ejecuta los programas Yosys [29] y ABC [30] para desarrollar las síntesis RTL y mapear cada uno de los elementos del modelo a la librería tecnológica adecuada. Seguidamente, recurre a OpenSTA [31] para ejecutar un STA y a Fault [32] para detectar fallos en el *netlist*. Este último se trata de un programa de *design for testing* (DFT) con las capacidades de *automatic test pattern generation* (ATPG) y *scan chain insertion* (SCI) con los que detectar errores de funcionamiento en el *netlist*.

A continuación, OpenROAD [33] toma el relevo y se convierte en la aplicación con mayor peso en el flujo de diseño. Compuesta por aplicaciones y *scripts* con propósitos específicos, OpenROAD desarrolla el *floorplaning* definiendo el *floorplan*, ubicando macro pines de entrada y salida, genera la red de distribución de energía e introduce celdas *welltap*[6] y *decap*[7]; desarrolla el *placement* completo introduciendo ciertas optimizaciones en el diseño,

[6] Estructura en diseños CMOS que conecta el *nwell* a la fuente de potencia y el sustrato p a tierra con el propósito de evitar el fenómeno de *latch-up* [34].
[7] Dispositivos capacitores de carga que provee a la red de corriente de forma inmediata si es necesario [35].

sintetiza la red de distribución de la señal de reloj (CTS), lleva a cabo el *routing*, y genera un archivo de extracción de parásitos SPEF (p. ej. resistencias y capacidades). Ha de tenerse en cuenta que pasos como el CTS o el DFT modifican el *netlist* original, por lo que es necesario realizar comprobaciones de equivalencia lógica (LEC) para garantizar que la funcionalidad no ha sido alterada. De nuevo, se recurre a Yosys para esta tarea [36].

En la etapa final, OpenLane utiliza Magic [37] para generar el *layout* definitivo en formato GDSII y un archivo LEF con información sobre la representación física. Con KLayout [38] se reproduce el mismo proceso para generar una segunda copia de seguridad del mismo. Ambas aplicaciones son herramientas de EDA orientadas a la manipulación del *layout* en VLSI, por lo que, entre otras utilidades, permiten visualizar la implementación final del diseño, realizar el DRC y la comprobación del posible efecto antena en la puerta de los transistores. Netgen, una herramienta incluida en Magic, ejecuta el análisis LVS, OpenSTA desarrolla un último STA y, finalmente, Circuit Validity Checker (CVC) [39] realiza ciertas validaciones finales, como un análisis ERC.

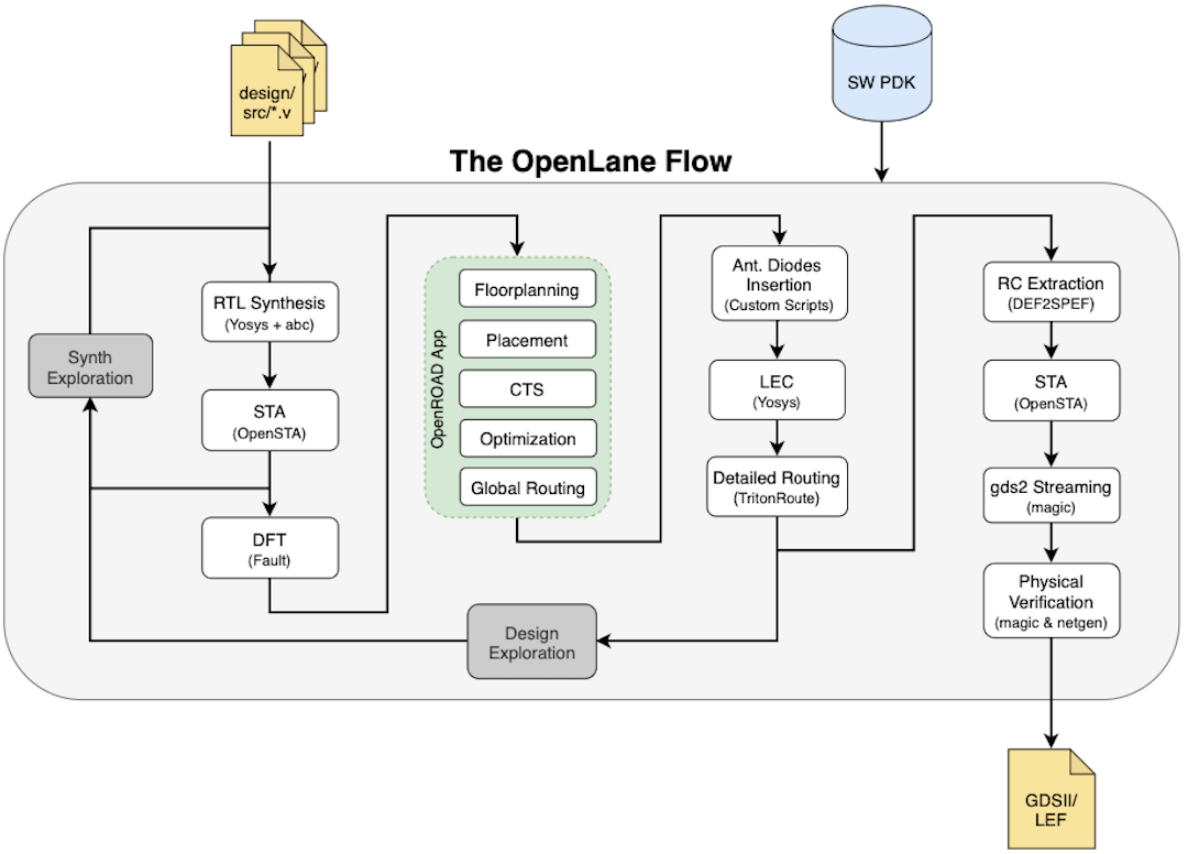

Figura 5. Esquema del desarrollo del flujo y arquitectura de OpenLane.
Fuente: https://openlane.readthedocs.io/en/latest/flow_overview.html

3.4. Arquitectura RISC-V

En el año 2010, como parte de un proyecto del laboratorio de computación en paralelo de la Universidad de Berkeley, el profesor Krste Asanović y los estudiantes graduados Yunsup Lee y Andrew Waterman crearon la ISA RISC-V. Aunque este no era un objetivo principal, su desarrollo y evolución siguió en adelante hasta que en 2015 se inauguró la fundación RISC-V [40]. Originalmente, el diseño de RISC-V pretendía servir como herramienta de apoyo para la investigación y la educación en arquitectura de computadores, pero ha crecido hasta ser un estándar de arquitectura de código abierto dentro de la industria [41].

Entre las características de RISC-V, podemos destacar su estructura modular y minimalista, ya que disponiendo tan solo de 47 instrucciones es posible implementar extensiones que satisfagan las necesidades cualquier desarrollador. Por ejemplo, además del conjunto base, podemos encontrar extensiones de introducen la multiplicación, la división, registros de punto flotante, entre otras. Adicionalmente, dado que su núcleo sigue siendo el mismo, se favorece a la compatibilidad entre proyectos; lo cual es realmente beneficioso dada la gran acogida que ha tenido RISC-V en el sector.

Aunque existen versiones de RISC-V capaces de trabajar con mayor número de bits, en este trabajo se presentará una versión que manipula datos de 32 bits (RV32I), siendo el conjunto de instrucciones ejecutables clasificado en seis formatos: operaciones entre registros (R-type), operaciones con inmediatos (I-type), escritura de datos en memoria (S-type), instrucciones de salto condicional (B-type), instrucciones salto incondicional (J-type), y carga de inmediatos superiores (U-type). En la Figura 6 se representa cómo cada una de estas instrucciones secciona la cadena de 32 bits en varios campos de longitudes variadas.

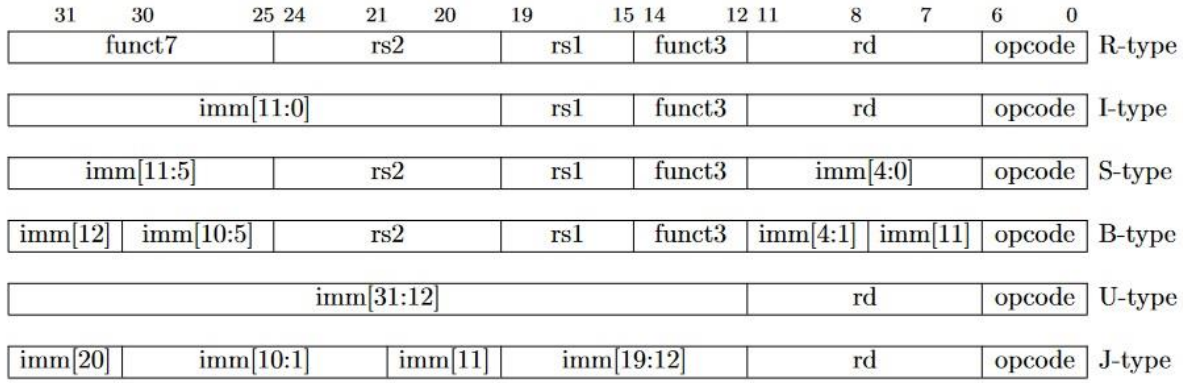

Figura 6. Conjunto básico de instrucciones de RISC-V mostrando la estructura de cada tipo de instrucción.
Fuente: The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA (2019, p. 16)

En primer lugar, *func7*, *func3* y *opcode* determinan la operación que se llevará a cabo. *opecode* tiene una extensión de 7 bits y define a cuál de los seis conjuntos pertenece la instrucción, es decir, como deben interpretarse los 25 bits restantes. *func3* (3 bits) y *funct7* (7 bits) especifican la operación realizar con los datos. En segundo lugar, *rd*, *rs1* y *rs2*, todas ellas de 5 bits, se refieren a direcciones de registro: *rd* al espacio en el que se almacenará el resultado de la operación, y tanto *rs1* como *rs2* los registros de los que extraer los datos con los que operar. Por último, *imm* (de 12 0 20 bits según el tipo de instrucción) refiere a un valor numérico inmediato con una aplicación distinta según lo demande la instrucción.

Adicionalmente, RISC-V propone un convenio para programadores a la hora de hacer uso de los 32 espacios de memoria del registro de datos. Como resume la Figura 7, son etiquetados ordenadamente como x0, x1, x2, …, x31, el registro x0 siempre preservará el valor decimal cero y no será modificable; el registro x1 se reserva para almacenar una dirección de retorno al programa; x2 un registro reservado para el puntero de la pila[8]; los registro x3 y x4 tienen restringido su uso para el puntero global y el puntero de hilo, respectivamente, necesarios para la gestión de excepciones. A continuación, desde x5 a x7 y desde x28 a x31 se consideran registros temporales y son de libre uso por el usuario; los registros x8, x9 y desde x18 a x27 se establecen como registros estáticos, con la misma aplicación que los temporales, pero garantizan que su contenido se conserva después de que un módulo del programa llame a otro para su ejecución. Por último, los registros x10 a x17 son registros de argumentos, también temporales y que sirven para intercambiar datos entre módulos.

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Figura 7. Convenio de registros de RISC-V. Nombre del registro, nombre ABI y descripción.
Fuente: The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA (2019, p. 137)

[8] Es una región de la memoria disponible para asignar subrutinas del programa principal. Su espacio es variable y es liberado una vez finaliza la subrutina [42].

4. Desarrollo práctico

4.1. Características del diseño

Para poner a prueba las prestaciones y validez de OpenLane como herramienta de diseño de circuitos integrados, se ha decido desarrollar como ejemplo ilustrativo un prototipo de procesador básico basado en la arquitectura RISC-V. Este modelo ha sido creado desde cero basándose en las estructura e instrucciones presentadas por la Universidad Rey Juan Carlos en una serie de tutoriales: *Arquitectura de Computadores: RISC-V* [43].

El diseño final se trata de una unidad de procesamiento monociclo capaz de ejecutar hasta 33 tipos de instrucciones diferentes clasificadas en: operaciones entre registros, operaciones con valores inmediatos, lectura y guardado de datos en memoria, y control de programa. En su conjunto, el sistema completo se constituye de siete módulos, cuyas especificaciones son:

- **Registro de datos**: Una memoria de 32 registros y 32 bits cada uno. Cuenta con 7 puertos de entrada y 2 de salida. Una señal de reloj controla la escritura de un solo dato de 32 bits, y la lectura simultánea de dos registros como señales de salida de 32 bits. La selección de los tres registros pertinentes la determinan 3 señales de 5 bits cada una. Adicionalmente, una señal de reinicio permite formatear el estado de la memoria, y otra señal habilita e inhabilita la escritura en la misma.
- **Unidad aritmético-lógica** (ALU): Cuenta con 2 puertos de entrada de 32 bits que establece los datos con los que operará el módulo y otro de 5 bits que determina la operación a ejecutarse. Entre las opciones disponibles están la suma, la resta y las operaciones lógicas AND, OR, XOR, determinar menor que, determinar menor que con la consideración de signo, desplazamiento lógico a izquierda, desplazamiento lógico a derecha, desplazamiento aritmético a la derecha, y determinar igual que. La respuesta se considera en la señal de 32 bits de salida del bloque.
- **Memoria de datos**: Una unidad de almacenamiento de datos de 1 KiB (256 espacios de memoria de 32 bits cada uno) con 4 puertos de entrada y uno solo de salida. El valor de escritura es de 32 bits y está sincronizado con la señal de reloj. En el mismo ciclo se produce de manera instantánea la lectura del valor de salida, también de 32 bits. Una señal de 8 bits determina el registro sobre el que simultáneamente se desarrollan ambas operaciones. Adicionalmente, otra señal permite habilitar e inhabilitar la escritura.

- **Contador de programa**: Se trata de un único registro de 12 bits sincronizado con la señal del reloj. En cada ciclo, su valor contenido se ve incrementado una cantidad introducida a través de una señal de 12 bits, y que es determinada por la serie de instrucciones del programa. Una señal de reinicio permite reestablecer el valor inicial de contador.
- **Codificador de instrucciones**: Recibe como única entrada una cadena de 32 bits, que contiene toda la información de una línea de instrucción del programa, y la descompone y estructura en hasta 9 señales de salida que serán distribuidas entre los demás módulos del sistema según son necesarias: 3 de ellas siempre se dirigen a la unidad de control y las demás pueden ser repartidas entre el registro de datos, la ALU y el contador de programa.
- **Unidad de control**: En función de 3 señales recibidas del codificador de instrucciones (dos de ellas de 7 bits y una de 3 bits), organiza 9 señales que se reparten entre todos los demás elementos del sistema. Por ejemplo, la habilitación de escritura en el registro de datos o en la memoria de datos, la operación a ejecutar en la ALU, y el control de ciertos multiplexores que organizan el flujo de datos dentro del sistema completo.
- **Red de interconexión**: Aunque realmente no constituye un bloque funcional en la estructuración de un procesador, el diseño requiere de la definición de un módulo superior que interconecte todos los puertos mencionados de cada uno de los módulos anteriores. Así, este bloque instancia a todos los demás e incorpora multiplexores que controlan el flujo de datos en el sistema. Su definición establece como entradas del sistema la señal de reloj, el reinicio del registro de datos y el contador de programa, y la instrucción de programa de 32 bits. La única señal de salida establecida es el contenido de 12 bits del contador de programa.

De forma externa al procesador, en el *testbench* del sistema completo, se han definido el ensamblador y la memoria de programa. El ensamblador desarrollado se trata de un programa capaz de leer lenguaje ensamblador desde un archivo de texto y traducirlo a código máquina dentro de la memoria de programa, la cual se trata de una memoria de ROM de 4 KiB. Lo habitual es que un espacio de la memoria de datos sea reservado para contener las instrucciones que ordenan el comportamiento del sistema, sin embargo, la manera que estamos presentado nos permite simplificar la construcción del ensamblador y la incorporación de las instrucciones en el sistema, aunque no hace posible sintetizar en el diseño estos componentes. La Figura 8 ilustra las interconexiones del sistema y ayudará a visualizar como se relacionan los módulos.

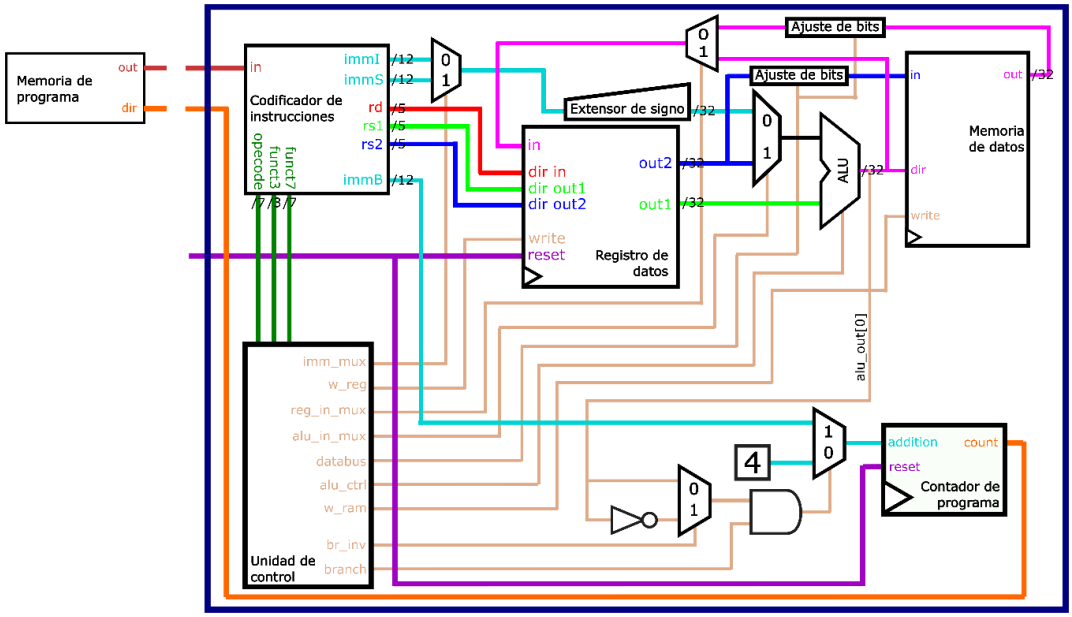

Figura 8. Esquematización de las interconexiones entre los módulos del prototipo de procesador desarrollado.

En la dirección (https://github.com/IsmGB/final-project-tfm) podrá ser encontrada la descripción completa del sistema en formato Verilog y cada uno de los *testbenches* asociados. Así mismo, también se incluyen los resultados obtenidos de su desarrollo en *IIC-OSIC-TOOLS*.

4.2. Manual de uso rápido del entorno y herramientas de *IIC-OSIC-TOOLS*

En esta sección se detallarán los pasos seguidos para la obtención del *layout* del procesador que se acaba de presentar. Partiendo desde la instalación de los programas necesarios para ejecutar *IIC-OSIC-TOOLS*, se enumeran las instrucciones que debería de seguir el usuario interesado, y se comentarán los aspectos e incidencias más relevantes a tener en consideración. En el Apéndice I puede encontrarse una guía inmediata que lista de forma extremadamente resumida los pasos a seguir.

4.2.1. Instalación y ejecución de Docker y *IIC-OSIC-TOOLS*

IIC-OSIC-TOOLS se trata de una imagen preparada para ser montada en un contenedor de Docker [44], por lo que previamente será necesario descargar e instalar esta aplicación en nuestro equipo. Docker se trata de una plataforma de código abierto diseñada para trabajar y gestionar dichos contenedores, los cuales puede ser interpretados como dependencias de software aislados. Manipular Docker a todo su potencial es algo complejo y que se escapas de las competencias de este proyecto, por lo que nos limitaremos a enumerar los pasos necesarios para ejecutar *IIC-OSIC-TOOLS*.

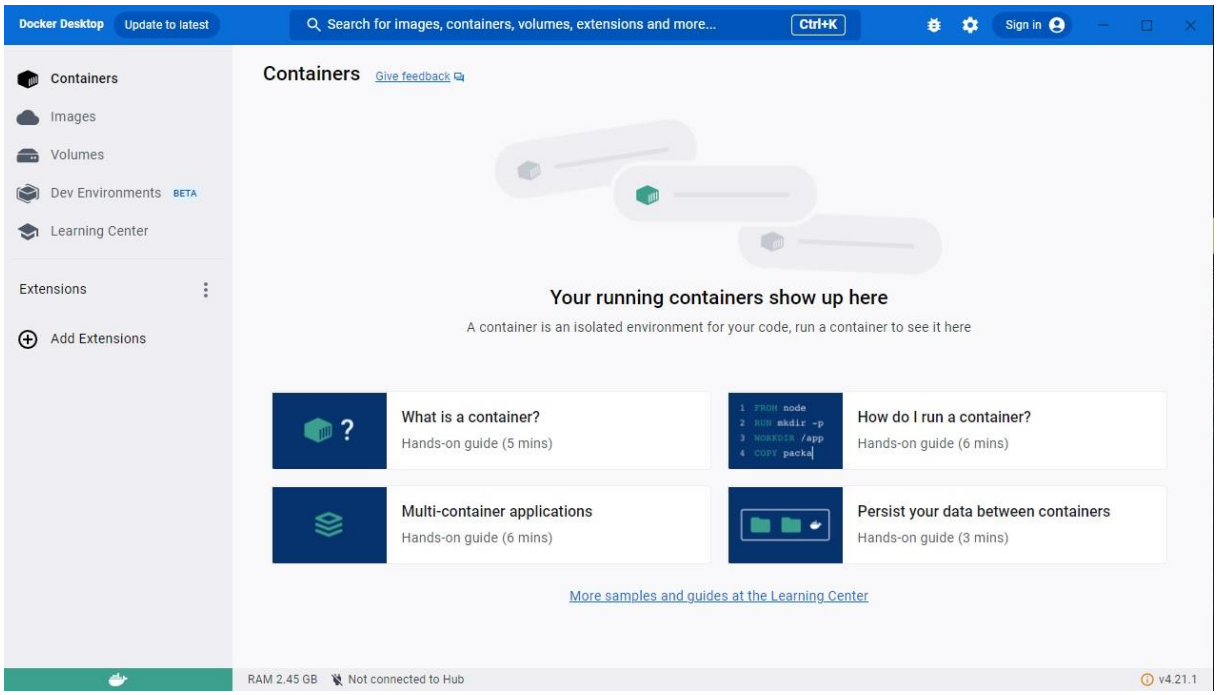

Figura 9. Interfaz gráfica de Docker. Primera inicialización del programa.

Desde el siguiente enlace (https://www.docker.com/products/docker-desktop/) podrá descargarse la versión de escritorio de Docker. Ejecutaremos el instalador y seguiremos todos los pasos recomendados. Una vez instalada, abriremos la aplicación y nos solicitará aceptar términos y condiciones de uso. A continuación, nos invitará a crear, opcionalmente, una cuenta de usuario, y en el caso de no hacerlo rellenaremos un breve cuestionario sobre nuestro propósito de uso. Siguiendo los pasos correctamente llegaremos a una interfaz como la de la Figura 9.

A partir de aquí, presentamos dos métodos para instalar *IIC-OSIC-TOOLS*. En el primero, haremos en la barra superior la búsqueda de "hpretl/iic-osic-tools", seleccionaremos la versión de nuestro interés y pulsaremos "*Pull*" para descargar la imagen del entorno —ver Figura 10—. La opción alternativa es acceder al GitHub de *IIC-OSIC-TOOLS* (https://github.com/iic-jku/IIC-OSIC-TOOLS) y descargar los contenidos en nuestro equipo. Desde la consola de comandos de nuestro sistema nos ubicaremos dentro de la carpeta que acabamos de descargar y ejecutaremos el archivo `start_x.bat`. La imagen comenzará a desplegarse y, en ambos casos, tras finalizar encontraremos en la pestaña "*Images*" de la interfaz de Docker que la imagen está disponible.

Ahora, pulsaremos a la derecha el botón "*Run*" para crear un contenedor a partir de esta imagen y se desplegará un menú como el de la Figura 11. Le daremos un nombre cualquiera al contendor y en "*Host port :80/tcp*" introduciremos el número 80. Cualquier otra configuración

ha resultado en la imposibilidad de entrar el contenedor. Aceptamos la configuración pulsando en "*Run*" y en la pestaña "*Containers*" encontraremos que el contenedor ha sido generado. Para para activar su uso presionaremos a la derecha "*Start*", y en puerto 80:80 para acceder a su interior. Se abrirá una ventana en nuestro navegador y nos será solicitada la siguiente contraseña: `abc123`. Acabamos de entrar exitosamente en el entorno de *IIC-OSIC-TOOLS*.

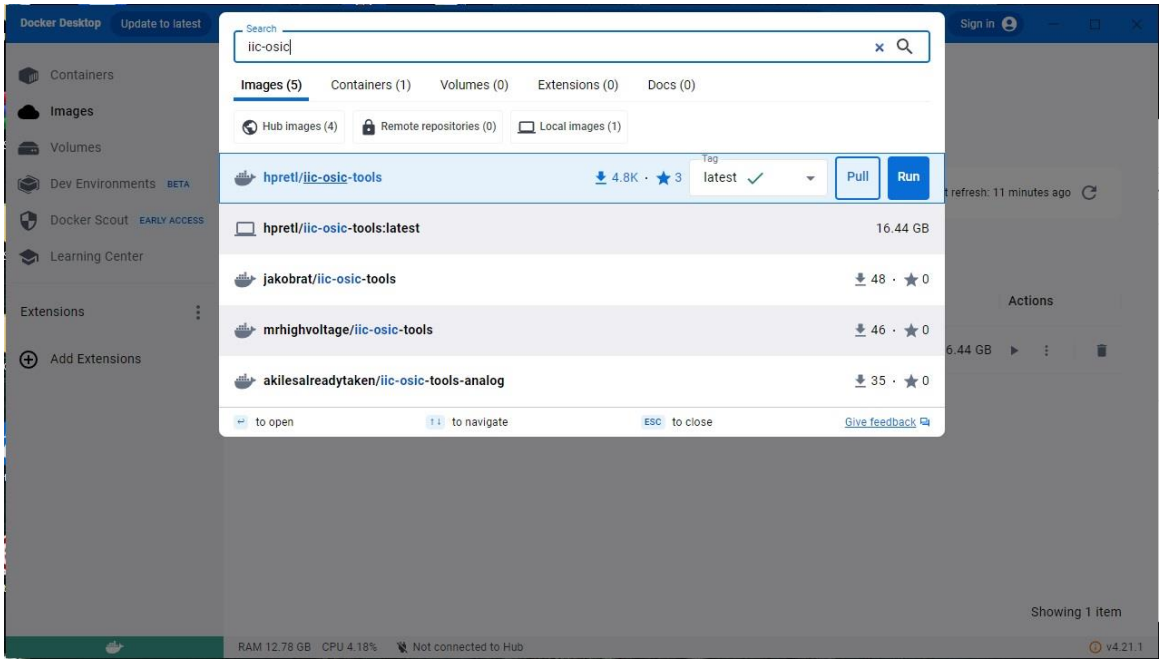

Figura 10. Descarga de la imagen de IIC-OSIC-TOOLS a través de la interfaz gráfica de Docker.

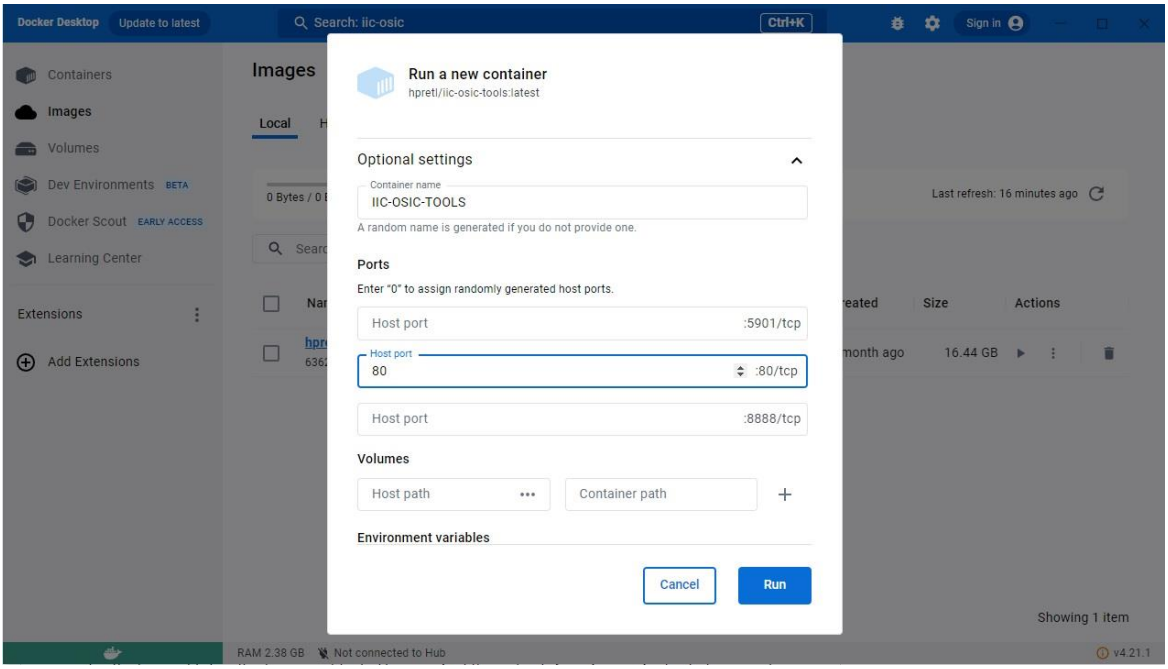

Figura 11. Menú de configuración para la creación de un nuevo contenedor a partir de una imagen.

A priori, el entorno ejecutado se nos presenta y maneja de igual forma que el sistema operativo Linux. Abajo a la izquierda tenemos el menú del sistema, un acceso directo al explorador de archivos y un acceso directo a la consola de comandos —ver Figura 12—. Podremos explorar todos los contenidos del sistema, pero la mayoría están protegidos para no ser eliminados ni modificados; de modo que el escritorio pude ser un buen directorio para crear y probar nuestros propios proyectos. Cuando queramos salir del entorno, pulsaremos "*Log Out*" en el menú y podremos decidir si guardar la sesión para la próxima inicialización o no.

Téngase en cuenta que este procedimiento de instalación ha sido considerado para Windows 10. Se recomienda que el disco en el que se llevará a cabo la instalación disponga al menos de 20 GB de memoria libres, y para hacer un uso eficiente de los programas instalados al menos de 16 GB de memoria RAM[9]. La versión de Docker utilizada para este proyecto ha sido la 4.21.1 y la de *IIC-OSIC-TOOLS* la 2023.06.

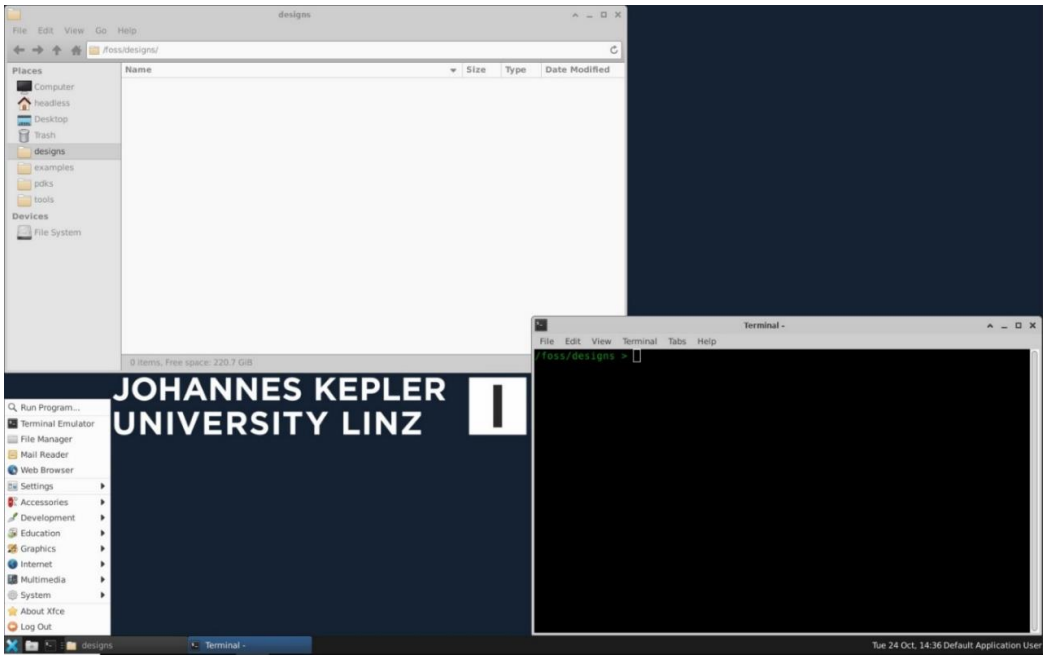

Figura 12. Apariencia del entorno de IIC-OSIC-TOOLS. Menú del sistema, explorador de archivos y terminal de comandos.

4.2.2. Descripción HDL: Iverilog y GTKWave

En adelante se muestra el procedimiento seguido para desarrollar un correcto flujo digital. Para facilitar la comprensión del lector y mostrar una expresión completa de los comandos y directorios referenciados, se advierte de que todo el texto contenido entre `< >` será genérico e identificativo. El usuario será libre de modificarlo a su conveniencia.

[9] En la práctica se ha detectado un uso habitual entre 2 GB y 8 GB de memoria RAM llegando a registrarse ocasionalmente picos de hasta 13 GB —dependiendo del proceso en ejecución—.

Para comenzar a diseñar nuestro sistema nos basta con hacer uso del editor de texto preinstalado. En el directorio que hayamos escogido para trabajar, podemos crear un documento de texto vacío haciendo clic derecho en el explorador de archivos, y añadirle a su nombre la extensión correspondiente al lenguaje que vayamos a requerir —en nuestro caso particular, la extensión para Verilog es ".v"—. También existe la posibilidad de importar diseños creados fuera del entorno. El directorio `/foss/designs` sirve como portal entre el entorno de *IIO-OSIC-TOOLS* y el resto de nuestro ordenador. Los archivos guardados en esta carpeta serán accesibles para nuestro sistema operativo desde `C:\Users\<carpeta de usuario>\eda\designs`, y viceversa.

Una vez dispongamos de un diseño y su correspondiente *testbench*, antes de proceder a su compilación y verificación de funcionalidad, debemos asegurarnos de que el *testbench* incluye dentro de un bloque `initial` el código necesario para volcar en un fichero los valores de las variables de entrada y salida del módulo. Estos comandos son `$dumpfile`, para generar el archivo con extensión ".vcd" en el que se depositarán los datos para la simulación; y `$dumpvars`, que especifica las variables que serán almacenadas —valores las entradas, salidas y registros del módulo superior (el *testbench*), y entradas y salidas de todos los módulos instanciados—. Para acceder a los valores de registros en niveles inferiores es necesario hacerlo explícitamente con `$dumpvars` para cada uno de ellos. También es necesario señalar en el *testbench* cuáles son los ficheros que corresponden a los módulos que se pretenden instanciar, lo cual se realiza con la instrucción `` `include ``. Por otro lado, es conveniente recurrir a la función `$finish` para llamar a la finalización del programa una vez transcurrido el tiempo que sea necesario[10].

Ahora, desplegaremos la consola de comandos y la haremos apuntar dentro de nuestro directorio de trabajo —allí donde estemos manipulando la descripción del módulo y su *testbench*—. Ejecutamos la siguiente línea de código para invocar a Iverilog y compilar el diseño:

```
iverilog -o <nombre_diseño>.vvp <nombre_diseño_testbench>.v
```

Si existe algún error en el módulo o en el *testbench*, en la propia consola se reportará el error. Si la compilación ha sido completada correctamente, se generará un archivo `<nombre_diseño>.vvp`, y escribiremos el siguiente comando para ejecutar la simulación:

```
vvp <nombre_diseño>.vvp
```

[10] Para más información sobre estas instrucciones de Verilog, y muchos otros fundamentos de este lenguaje, se sugiere consultar [45].

Una vez finalizada podemos visualizar los resultados con GTKWave ejecutando:

```
gtkwave <nombre_diseño>.vcd
```

Se abrirá una interfaz gráfica, como la de la Figura 13, en la que se podrán distinguir tres ventanas bajo una barra de herramientas: a la izquierda, un mapa desplegable de la jerarquía del diseño y las señales pertenecientes al módulo seleccionado; en el centro, las señales seleccionadas para su análisis; y a la derecha, la representación frente al tiempo de las señales añadidas en la anterior. Entre los botones y funciones disponibles destacaremos algunos de los más útiles para llevar a cabo el análisis funcional:

- El botón "*Append*" añade a la ventana "*signals*" las señales seleccionadas desde el mapa de la jerarquía para su análisis. El botón "*Insert*" las incluye bajo la selección hecha en "*signals*". Y "*Replace*" sustituye la selección en "*signals*" por las nuevas que se pretenden introducir.
- En la barra de herramientas, las entradas de texto "*From*" y "*To*" permiten especificar el intervalo de tiempo a representarse. El botón "*Zoom Fit*" ajusta este periodo para ocupar todo el espacio en la ventana "*waves*". Podemos ajustar el zoom con la tecla "Ctrl" del teclado y la rueda del ratón o con los botones "*Zoom in*" y "*Zoom out*". El botón "*Reload*" recarga la representación de las señales.
- Haciendo clic derecho sobre las señales añadidas a la ventana "*signals*" es posible seleccionar el formato de su valor numérico y el color de la onda. Haciendo doble clic sobre una señal de tipo bus se puede desplegar para visualizar cada uno de sus elementos.
- Haciendo clic izquierdo sobre la ventana "*waves*" se generará un cursor, y el valor marcado sobre las señales aparecerá en la ventana "*signals*". La posición exacta del cursor y el cursor del ratón se especifica bajo el título de la ventana principal.
- El botón "*Menu*" de la barra de herramientas despliega el menú completo de opciones del programa, donde se repiten algunas de las que acaban de ser enumeradas y otras muchas más avanzadas.
- La combinación de teclas "Ctrl+S" permite guardar en formato ".gtkw" el estado actual de la ejecución. Posteriormente, podrá ser cargado mediante la combinación "Ctrl+O", al iniciar el programa desde la consola de comandos tal y como:

  ```
  gtkwave <nombre_diseño>.gtkw
  ```

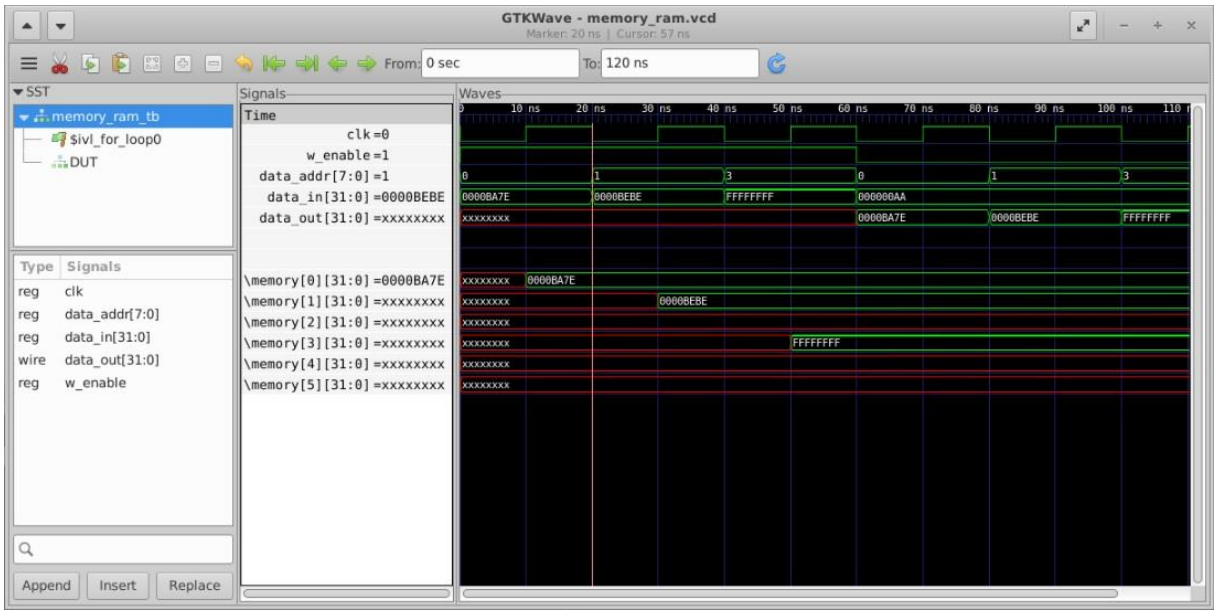

Figura 13. Interfaz gráfica de GTKWave. Ejemplo de ilustración basado en la verificación funcional de la memoria de datos.

4.2.3. Desarrollo del flujo de diseño con OpenLane

Una vez superada la primera etapa del flujo de diseño, se ejecutará OpenLane para desarrollar todas las demás restantes. Efectivamente, OpenLane orquestará todas las aplicaciones requeridas en adelante, y para ello lo primero de todo será definir nuestro directorio de trabajo como uno válido donde poder ejecutar OpenLane. Crearemos una carpeta —por ejemplo, `/<carpeta_diseño>`— y, señalando el mismo directorio en el que esté ubicada, ejecutaremos el siguiente comando en la consola:

```
flow.tcl -design <carpeta_diseño> -init_design_config -add_to_designs
```

Si se ha procedido bien, el terminal nos mostrará un mensaje de éxito indicándonos que en la carpeta que acaba de ser creada se ha generado un archivo con valores de configuración por defecto. Ahora, si entramos en `/<carpeta_diseño>`, encontraremos una nueva carpeta con el nombre `/src` y un archivo `config.json`. Adicionalmente, y de forma opcional, a la misma línea anterior se le puede añadir la instrucción:

```
-config_file <nombre_personalizado>.json
```

La cual permite generar el archivo de configuración con el nombre deseado y formato ".json" —o ".tcl" si se modifica la extensión—. Por defecto, este fichero contiene cinco instrucciones: `DESIGN_NAME`, `VERILOG_FILES`, `CLOCK_PORT`, `CLOCK_PERIOD` y `DESIGN_IS_CORE`. La explicación referida a estas variables y otras tantas pueden revisarse en el Apéndice II, pero por

ahora priorizaremos que ***DESIGN_NAME*** sea igual que el nombre de nuestro diseño y ***CLOCK_PORT*** se corresponda también con el nombre que le hemos dado a la señal de reloj en nuestra descripción. Por otro lado, en la carpeta ***/src*** incluiremos la descripción en Verilog del diseño que estemos tratando.

Ahora, para ejecutar OpenLane escribiremos en la consola el siguiente comando, que señala el diseño a desarrollarse y el archivo con las configuraciones pertinentes:

```
flow.tcl -design <carpeta_diseño> -config_file <carpeta_diseño>/config.json
```

Siendo posible incluir adicionalmente en la misma línea las opciones:

```
-tag <nombre_flujo> -overwrite
```

Que sirven, respectivamente, para darle un nombre a la ejecución del flujo y para sobrescribirse en el caso de que exista otro con el mismo nombre. Comprobaremos que se ha generado una carpeta ***/runs*** dentro de ***/<carpeta_diseño>***, y dentro de esta otra con un nombre por defecto en base a la hora de ejecución del comando —o bien ***/<nombre_flujo>*** si hemos hecho uso de la opción—.

Habiendo dado comienzo al flujo, en la consola podremos llevar un seguimiento del desarrollo de cada una de las fases. En primer lugar, antes de proceder a la síntesis RTL, OpenLane realiza un breve proceso de *linting* con Verilator [46] para advertir de errores en el diseño y sugerir optimizaciones en él. A continuación, se suceden las etapas de síntesis RTL, *floorplaning*, *placement* y *routing*, desarrollándose entre ellas diferentes procesos de análisis temporal, optimizaciones y reescalados del diseño. Cerca del final se generan los ficheros GDSII y LEF para el *layout* y se realizan las últimas comprobaciones sobre la implementación mediante los análisis DRC, LVS, CVC y ERC. En total el flujo consta de unas 42 etapas, de las cuales las que tomarán más tiempo —en función de la complejidad del diseño— serán las relacionadas con el *placement* y el *routing*. Si la ejecución se ha completado correctamente, en la consola se nos informará sobre el éxito del proceso. En el caso contrario, el flujo se verá interrumpido en la etapa que impide su continuación y se notificará la naturaleza del error.

En cualquier caso, dentro de ***/<nombre_flujo>*** encontraremos todos los archivos generados durante el proceso —véase un ejemplo en la Figura 14—. Los archivos contenidos en este directorio son:

- ***cmds.log***: Historial de códigos ejecutados en segundo plano por OpenLane en la consola durante el desarrollo del flujo.

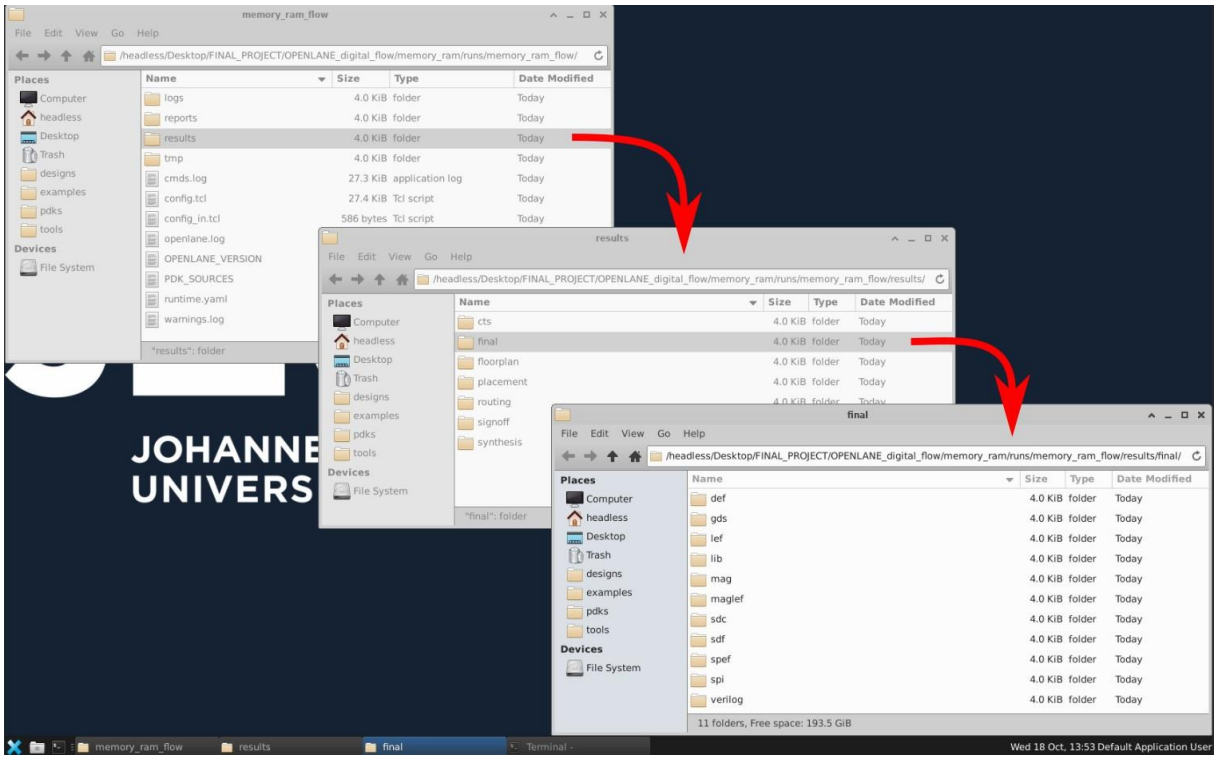

Figura 14. Estructuración del directorio del flujo de diseño. Ejemplo ilustrado con la memoria de datos.

- ***config.tcl***: Valores de configuración de todas las variables involucradas en el desarrollo del flujo. Las que no han sido especificadas previamente en ***config.json*** toman valores por defecto.
- ***config_in.tcl***: Valores de configuración tomados directamente de ***config.json***.
- ***openlane.log***: Historial de mensajes impresos en la consola durante la ejecución del flujo.
- ***OPENLANE_VERSION***: Versión de OpenLane utilizada para la ejecución.
- ***PDK_SOURCE***: Versión de los PDKs implementados en el diseño.
- ***runtime.yaml***: Tiempo requerido por cada una de las etapas del flujo en su desarrollo.
- ***warnings.log***: Advertencias generadas durante el desarrollo completo del flujo.
- ***/logs***: Contiene el historial de ejecución de cada uno de los programas requeridos durante el flujo. La información se refiere al estatus de cada aplicación, los avisos y los errores detectados.
- ***/reports***: Contiene archivos que reportan las características temporales y dimensiones del diseño, así como el resultado de los análisis DRC y LVS.
- ***/results***: Contiene los archivos generados por cada etapa del flujo y que son necesarios en las subsecuentes. Aquí podremos encontrar *netlists* del diseño en distintas fases, archivos SDF y los archivos finales GDSII y LEF del *layout*, entre muchos otros.

- ***/tem***: Contiene una gran variedad de archivos temporales. No resultan ser los definitivos para el desarrollo, pero son útiles para analizar el diseño en caso de no finalizarse correctamente la ejecución flujo.

Estos cuatro directorios a su vez organizan su propio contenido en seis carpetas para facilitar la localización de los ficheros: ***/cts***, ***/floorplan***, ***/placement***, ***/routing***, ***/signoff*** y ***/synthesis***. Adicionalmente, ***/results*** incluye un directorio ***/final*** donde están reunidos los archivos más importantes para la implementación física del diseño, entre los que podemos destacar el *layout* en formato ".gds" y ".lef".

En el caso de que nuestro diseño sea lo bastante sencillo, es posible haber incluido en ***/src*** los archivos con la descripción de los submódulos instanciados en el nivel superior del diseño que estemos tratando. Sin embargo, en cualquier caso, se recomendaría definir estos submódulos como macrobloques: cajas negras que serán incrustadas en el nivel superior de la jerarquía. Este enfoque de diseño no solo simplifica la estructuración del *layout* final, sino que además facilita los procesos de *placement* y *routing* y optimiza el tiempo de ejecución del programa. La técnica consiste en aplicar el proceso que se acaba de describir para generar los archivos GDSII y LEF de cada uno de los bloques que serán instanciados y tenerlos en consideración a la hora de ejecutar el flujo del nivel superior.

En el caso de esta práctica, el nivel superior sería la red de interconexión. Para mayor comodidad, en la carpeta de diseño de la red de interconexión —a la que hemos llamado ***/sistem***— hemos creado un directorio ***/<macro>*** en el que se han organizado la descripción Verilog de los seis módulos que instanciaremos en ***/<bb>***; el archivo GDSII generado para cada módulo en ***/<gds>***; y el archivo LEF de los mismos en ***/<lef>***. Para cada uno de los seis módulos referidos, los archivos GDSII y LEF están localizados, respectivamente, en:

```
../<carpeta_diseño>/runs/<nombre_flujo>/results/final/gds/<nombre_diseño>.gds
```

```
../<carpeta_diseño>/runs/<nombre_flujo>/results/final/lef/<nombre_diseño>.lef
```

Es muy importante añadir en una línea de texto ***/// sta-blackbox*** a cada uno de los archivos contenidos en ***/<bb>***. Este comentario es imprescindible para que las descripciones de los módulos sean identificadas como cajas negras y no entren en conflicto con el desarrollo del flujo del nivel superior. Por otra parte, el hecho de reunir todos los archivos de *layout* en el mismo directorio simplificará el proceso de ser referenciarlos en el archivo de configuración para el flujo. Las instrucciones pertinentes para ello son ***VERILOG_FILES_BLACKBOX***,

EXTRA_GDS_FILES y ***EXTRA_LEFS***. La ejecución y desarrollo del resto del programa será complemente análogo a lo descrito anteriormente.

4.2.4. Visualización y exploración de resultados con OpenROAD

Presentamos el uso de la interfaz gráfica de OpenROAD para visualizar el resultado de nuestro proyecto. Aunque también sería posible hacer uso de la interfaz de KLayout y Magic, la de OpenROAD resulta más completa e intuitiva para investigar la estructura y temporización del diseño. Para ser invocada basta con escribir en la consola de comandos la siguiente instrucción referenciando a la carpeta de nuestro diseño y la carpeta de flujo que queramos explorar:

```
flow.tcl -design <carpeta_diseño> -tag <nombre_flujo> -gui
```

La Figura 15 muestra el aspecto de esta interfaz, en la que la información está distribuida entre distintas ventanas de la siguiente manera:

- **Display Control:** Permite filtrar los elementos en la representación seleccionando en la columna identificada con un ojo qué capas serán mostradas u ocultas. Entre las opciones disponibles destacamos el uso de los mapas de calor para evaluar la densidad del *placement*, la densidad de potencia y la congestión del *routing*. Haciendo doble clic sobre cualquiera de las opciones subrayadas abriremos un menú en el que poder configurar las características de estos mapas —ver ejemplo en la Figura 16—.
- **Inspector**: Especifica las características de la celda, red o conjunto de ellos que hayamos seleccionado sobre la representación. Entre ellos podemos destacar el tipo de elemento, su nombre, la categoría a la que pertenece, su posición en el diseño, elementos contenidos y señales de entrada y salida. Si en "*Display Control*" seleccionamos una lámina de material aquí podremos ver sus características dimensionales, resistencia, capacitancia, y algunas otras.
- **Hierarchy Browser**: Nos presenta las celdas instanciadas en un mapa jerarquizado y enumera la cantidad de elementos de cada tipo presentes, a qué módulo o macrobloque pertenecen y el área que ocupan.

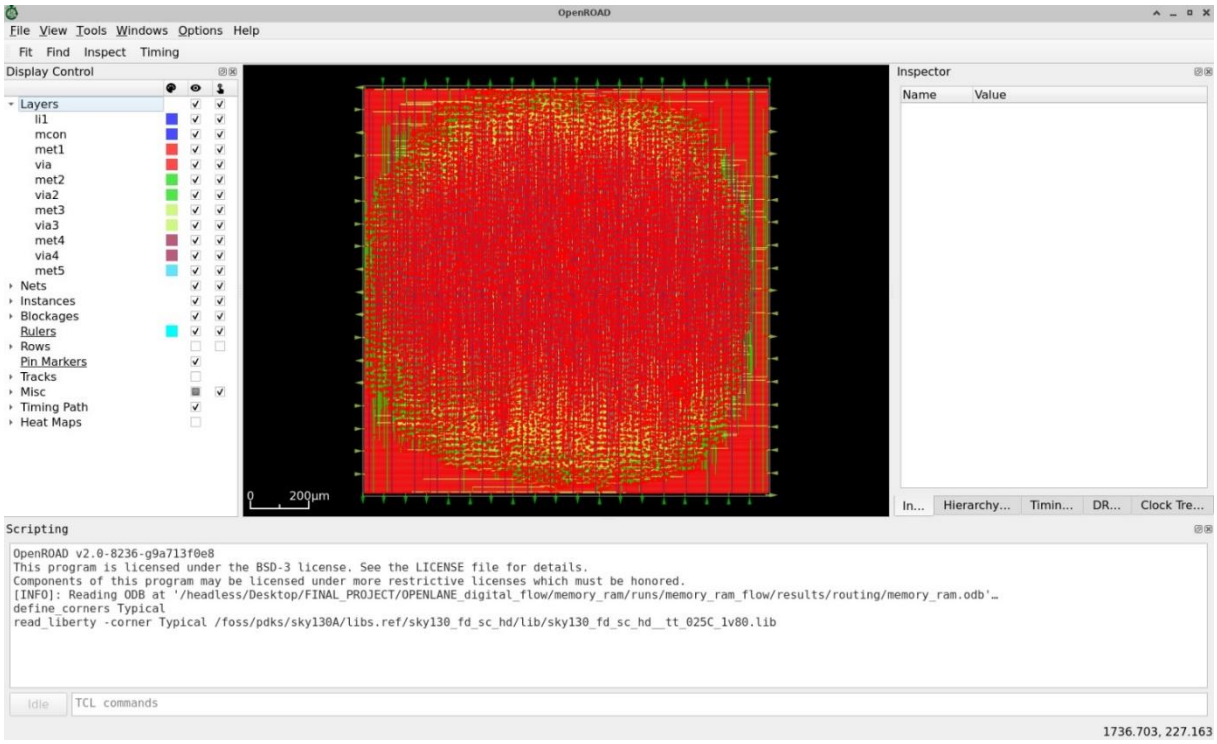

Figura 15. Interfaz gráfica de OpenROAD. Visualización del layput generado por la memoria de datos.

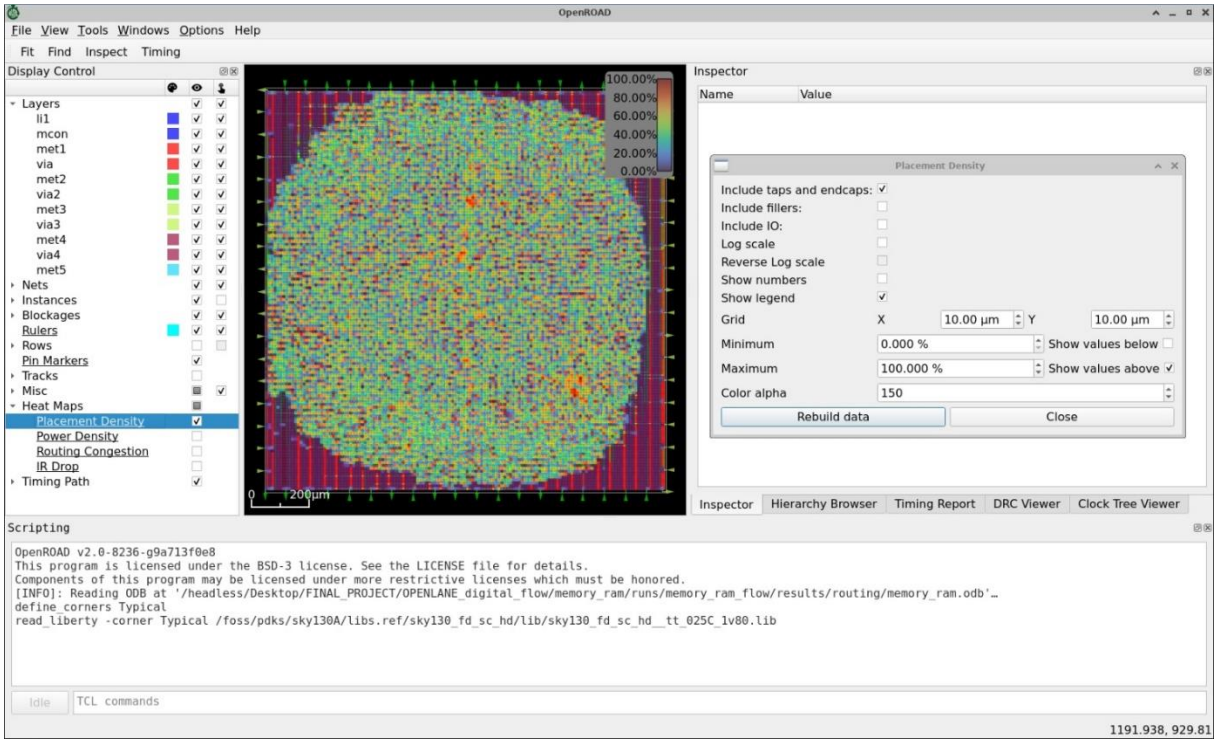

Figura 16. Interfaz gráfica de OpenROAD. Ilustración del menú de opciones para la densidad de placement aplicado sobre la memoria de datos.

- **Timing Report**: Haciendo clic en "*Update*" se cargarán múltiples rutas de la señal de reloj ordenadas por el *slack* que presente, y pudiendo diferenciarlas entre los periodos de de *setup* y *hold*. Haciendo clic sobre una de estas rutas podremos ver su recorrido resaltado sobre el gráfico —ver Figura 17—, y en la pestaña "*Data Path Details*" información específica para cada tramo de la señal. Pulsando en "*Settings*" podemos establecer en la opción *paths* el número de rutas que sean calculadas.
- **DRC Viewer:** Presionando "*Load*" se nos abrirá una ventana de búsqueda para cargar un archivo con información sobre el DRC del diseño. Estos archivos tienen la extensión ".rpt" y pueden encontrarse tanto en `/reports` como en `/tem` dentro de la carpeta de desarrollo del flujo, aunque no todos ellos pueden ser leídos por el programa. Si el diseño está libre de violaciones no se apreciará ningún cambio, pero haber alguna aparecerá listada en la ventana y marcada su posición sobre el gráfico.
- **Clock Tree Viewer:** Pulsar "*Update*" generará la representación de la red de distribución del reloj. En la propia ventana podemos ver la ramificación del árbol y el tiempo requerido por la señal entre los distintos niveles, mientras que en el gráfico se ilustra su extensión sobre el diseño —ver Figura 18—. Haciendo clic derecho sobre el árbol se desplegará un menú en el que poder seleccionar si mostrar u ocultar la representación del árbol, a qué nivel de la red queremos que se haga una distinción de color entre los caminos, y guardar la en formato ".png" la representación del árbol.
- **Scripting:** En esta ventana se mostrarán mensajes de información sobre la ejecución del programa. Además, incluye una línea de inserción de comandos con la que es posible manejar el programa mediante código TCL.

Si alguna de estas ventanas no aparece desbloqueada por defecto, puede ser activada desde la barra de herramientas en la pestaña "*Windows*". Es posible ajustar el zoom de la representación con la tecla "Ctrl" y la rueda del ratón. Seleccionar una región sobre el dibujo con el clic izquierdo resaltará todos los elementos y redes contenido, mientras que con el clic derecho se hará zoom sobre la zona. La tecla "F" nos permite regresar a un panorama global del diseño, y la cobinación "Ctrl+F" abre un menú de búsqueda en el que podremos localizar elementos por su nombre. Con la tecla "K" podremos dibujar reglas de medición haciendo clic izquierdo sobre dos puntos.

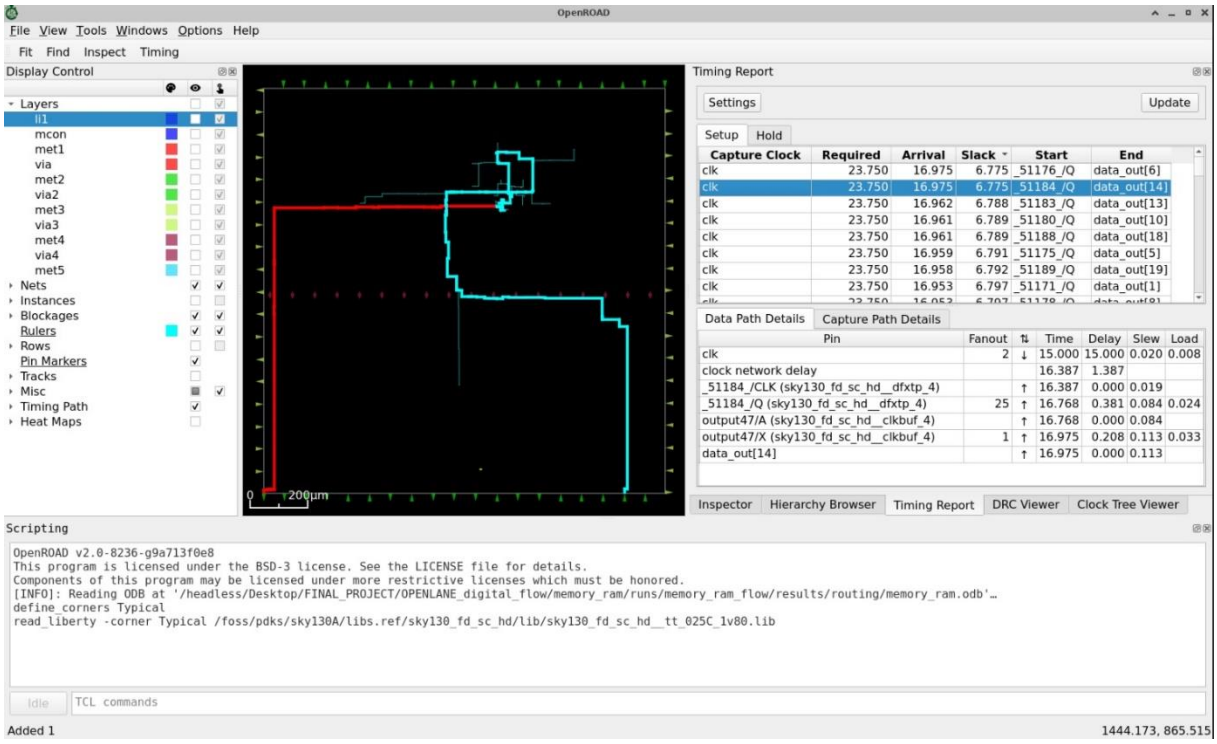

Figura 17. Interfaz gráfica de OpenROAD. Ilustración de la ventana Timing Report seleccionando una ruta de la señal de reloj sobre la memoria de datos.

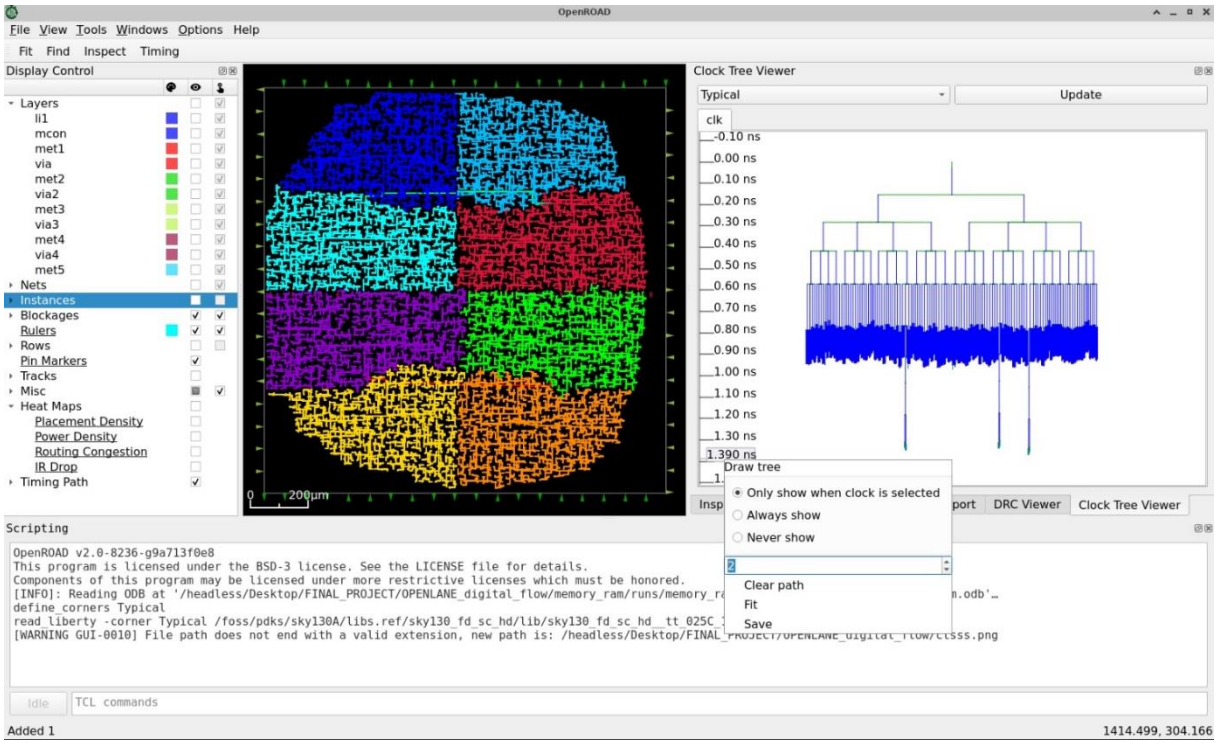

Figura 18. Interfaz gráfica de OpenROAD. Ilustración de la ventana Clock Tree Viewer mostrando la ramificación de la red en forma de árbol y su extensión sobre el circuito diferenciada al nivel dos.

4.2.5. Eventos ocurridos durante el desarrollo de nuestro prototipo y solución de problemas

En esta práctica, debemos de reconocer que desarrollar correctamente el flujo de diseño no ha sido algo inmediato, sino que se ha tratado de un proceso iterativo en el que múltiples errores se han sucedido en distintas etapas, y a los que hemos tratado de dar solución con éxito. A continuación, se listan todos los inconvenientes que se han podido encontrar y se detallarán soluciones y recomendaciones para optimizar el tiempo invertido en las ejecuciones del programa:

- **[ERROR] Traceback.** Se trata de un error referido a la sintaxis del archivo de configuración ***`config.json`***. Asegúrese de que las variables que se hayan añadido siguen el formato del fichero original —el último elemento de la lista no necesita ir sucedido por una coma—.
- **[ERROR] Errors found by Verilator.** Previamente a la síntesis RTL, Verilator revisa la descripción HDL introducida, reporta errores y advierte sobre modificaciones recomendables. El informe detallado de lo sucedido puede encontrarse en:
  `../<carpeta_diseño>/runs/<nombre_flujo>/logs/synthesis/verilator.log`
  Si es necesario modificar las descripciones de los módulos, es conveniente repetir la correspondiente simulación del *testbench* y comprobar que su funcionalidad sigue siendo la misma.
- **[ERROR] There are hold (setup) violations in the design at the typical corner.** La frecuencia de operación del reloj es demasiado elevada como para satisfacer las condiciones de *slack* en todos los puntos del diseño. Durante la ejecución se realizan análisis de tiempo en multitud de fases, por lo que podemos encontrar diversos informes dentro de ***`/reports`***. Como solución basta incrementar el periodo definido en ***`CLOCK_PERIOD`***.
- **[ERROR] There are cap violations in the design at the typical corner.** La capacitancia de ciertas redes está por encima de la que permiten los pines. Este error también está vinculado a la frecuencia de la señal de reloj, también se solventa aumentando el periodo de la misma. Los mismos archivos mencionados en el error anterior contienen la información referida a estas violaciones.
- **[ERROR] There are violations in the design after detailed routing.** Para obtener más detalles sobre lo sucedido, convienen ejecutar la interfaz gráfica de OpenROAD y cargar en "*DRC Viewer*" el archivo:

```
../<carpeta_diseño>/runs/<nombre_flujo>/reports/routing/drt.drc
```

En nuestro caso, los errores han saltado debido a la superposición de pistas al tratar de conectar los pines de entrada al macrobloque de la ALU. La solución a consistido en definir unos valores de `MACRO_HALO` y `MACRO_CHANNEL` de 10 um en la dirección vertical y horizontal para separar los macrobloques entre sí.

- **[ERROR PPL-0024] Number of IO pins exceeds maximum number of available positions.** Las dimensiones del chip no permiten que todos los pines de entrada y salida que se pretenden implementar respeten una distancia mínima entre sí. Esto nos ha sucedido con el codificador de instrucciones —el módulo con más puertos entre todos los diseñados— y se ha solventado fácilmente estableciendo `FP_SIZING: "absolute"` y `DIE_AREA: "0 0 200 200"`, lo que define un chip cuadrado de 200 um de lado.
- **[ERROR GRT-0076] Net not properly covered.** OpenLane no ofrece mucha información acerca de este error. Nos ha sucedido durante la ejecución del flujo para la red de interconexión. Inspeccionando el diseño se ha intuido que la red determinada no resulta accesible, por lo que se ha modificado el factor de forma del chip arbitrariamente con `FP_SIZING: "absolute"` y `FP_ASPECT_RATIO: "1.2"`.
- **[ERROR GRT-0119] Routing congestion too high.** Debido a la alta densidad de celdas instanciadas y vías de interconexión en el diseño, no resulta factible continuar con el desarrollo de la implementación. En nuestro caso ha sucedido para la generación de la memoria de datos, y ha sido salvado reduciendo la utilización del núcleo con `FP_CORE_UTIL: "20"` y la densidad de elementos con `PL_TARGET_DENSITY: "0.25"`. Otra alternativa sería aumentar las dimensiones del chip con `FP_SIZING: "absolute"` y `DIE_AREA`, pero a costa de prolongar el tiempo de ejecución. Si `FP_CORE_UTIL` y `PL_TARGET_DENSITY` se reducen por debajo de cierto límite tampoco será posible lograr la implementación si las dimensiones del chip no lo permiten, por lo que habría que establecer un equilibrio entre estas variables.
- **[ERROR] There are XOR differences in the designs.** En las últimas etapas Magic y KLayout generan cada uno su propio archivo GDSII y KLayout los compara haciendo uso de su herramienta preinstalada XOR Tool. De los siete flujos ejecutados para completar esta práctica, este paso únicamente ha dado problemas para la finalización de la red de interconexión, dando lugar a miles de discrepancias entre ambos modelos. La falta de información generada por OpenLane y no haber podido encontrar discusiones de los desarrolladores al respecto no nos permite justificar la causa de este error. No

obstante, la generación de los archivos GDSII sí es exitosa, por lo que saltando esta comparativa con `RUN_KLAYOUT_XOR: false` el flujo se completa sin más inconvenientes.

Téngase en cuenta que cualquiera de los errores surgidos durante la ejecución de OpenROAD darán lugar a la creación de `/issue_reproducible` dentro de la carpeta de desarrollo del flujo. El directorio alberga material referido al estado del flujo con el que poder repetirse el incidente y con el que los desarrolladores de OpenROAD podría ayudar en el caso de que se presente en su GitHub [47] el adecuado formulario reportando el error.

Adicionalmente, se darán ciertas recomendaciones y observaciones consideradas en nuestro diseño que, aunque su desuso no tenga por qué desembocar en errores de ejecución, sí son convenientes de ser aplicadas. Así, empezando por el tamaño de los diseños, los desarrolladores de OpenLane recomiendan que los macrobloques tengan unas dimensiones mínimas de 200 um x 200 um debido a que bloques muy pequeños pueden tener problemas para encajar en la red de potencia [48]. Por defecto, OpenLane puede establecer automáticamente longitudes inferiores a estas, por lo cual, si se nos da este caso, es adecuado configurar `FP_SIZING: "absolute"` y `DIE_AREA: "0 0 200 200"`.

Refiriéndonos a la temporización del sistema, si el módulo que estamos desarrollando no es dependiente de la señal del reloj, sería conveniente que el correspondiente archivo `config.json` prescinda de `CLOCK_PORT` y se establezca `CLOCK_PERIOD: null`. Del mismo modo, tampoco es necesaria la fase de CTS, por lo que se puede establecer `RUN_CTS: false`. Por otra parte, si disponemos de varios módulos que comparten la misma señal de reloj, sería adecuado comenzar por la síntesis de aquel que intuyamos que trabajará a la menor frecuencia. De esta forma ajustaremos el periodo del reloj para evitar violaciones de *slack* en este módulo y no será necesario resintetizar los otros en el caso de que hubiéramos escogido un periodo válido para los ellos, pero no para este. El archivo `metrics.csv` —que podemos encontrar en la carpeta `/reports` dentro del directorio de ejecución del flujo— contiene una gran variedad de información referida las características del diseño una vez completado, entre las cuales están el periodo del reloj empleado y el periodo sugerido para evitar las violaciones de *slack*. Tal archivo puede ser leído en la consola de comandos ejecutando el código:

```
file="../reports/metrics.csv";
IFS="," read -ra line < $file;
for ((i=1; i<=${#line[@]}; i++));
do cut -d "," -f$i $file | tr "\n" "\t";
echo; done
```

O bien puede ser exportado fuera del entorno a través de /foss/designs para ser leídos con Excel. Para ello, en Excel creamos un libro en blanco, desde la pestaña *Datos* hacemos clic en *Desde un archivo de texto*, seleccionamos `metrics.csv` en nuestro equipo y lo importamos, pulsamos *Siguiente*, en el paso 2 seleccionamos separación por comas y pulsamos *Siguiente*, *Finalizar* y *Aceptar*.

El desarrollar nuestro prototipo de procesador se estableció un periodo inicial de 10 ns. Aunque el registro de datos y el contador de programa podían trabajar a esa frecuencia, para la memoria de datos era demasiado rápido. En el archivo metrics.csv se recomendaba un periodo de 11 ns, sin embargo, esta modificación también desembocaba en error y sugería ahora 12 ns. En definitiva, recomendamos que se haga uso de intuición para ajustar la señal de reloj en base a la diferencia por la que se viola el *slack*. En nuestro caso, al final la señal de reloj ha pasado a ser de 30 ns.

A continuación, tratando la consolidación final del diseño, queremos reincidir en la implementación de submódulos como macrobloques. Primeros intentos por sintetizar todo nuestro prototipo de una sola vez han finalizado en errores de *routing* por la sobrecarga de vías de conexión, y tiempos de ejecución mucho más largos que sintetizando los bloques independientemente para luego combinarlos sobre la red de interconexión. Aceptando la estrategia de los macrobloques, es necesario definir en el `config.json` de cada submódulo `DESIGN_IS_CORE: false`. Esta opción limita el *routing* del módulo hasta la cuarta capa de metal —de las cinco que presenta la tecnología sky130—, dejando la última disponible para el *routing* y la red del reloj del módulo superior del diseño, el cuál sí toma `DESIGN_IS_CORE: true`. Otra alternativa sería definir en la variable `RT_MAX_LAYER` cuál es la lámina más alta disponible para el *routing*. Para cualquiera que sea el módulo superior del diseño, también deberíamos establecer `FP_PDN_CORE_RING: true` para crear un anillo alrededor del chip con las conexiones de alimentación y tierra.

Por último, mencionemos que la disposición de elementos sobre el chip es automática, pero puede ser configurada por el usuario. Es aconsejable que el diseño respete ciertas razones de simetría para equilibrar la difusión de calor sobre el silicio, y posicionar los elementos con pines de entrada y salida directos del sistema en el contorno para reducir la longitud y cantidad de las vías necesarias por el *routing*. En la Figura 19 podemos ver un ejemplo de posicionamiento automático en el que la distribución de los macrobloques está completamente descompensada sobre el lado izquierdo del chip. Haciendo uso de la opción

MACRO_PLACEMENT_CFG podemos señalar un archivo con formato ".cfg" que defina las posiciones —en micrómetros— de los macrobloques. El contenido de este archivo se estructura de la siguiente manera, considerando ***<nombre_módulo_instanciado>*** tal y como es definido su nombre en la descripción HDL del en el módulo superior en el que es introducido, y ***<orientación>*** puede ser ***N***, ***S***, ***E*** o ***W***:

```
<nombre_modulo_instanciado1> <posición_X> <posición_Y> <orientación>
<nombre_modulo_instanciado2> <posición_X> <posición_Y> <orientación>
                                    ...
```

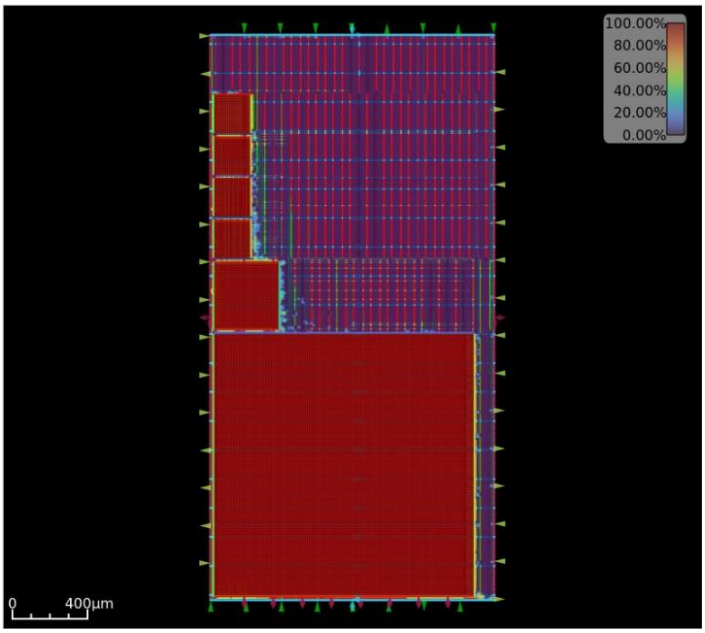

Figura 19. Mapa de la densidad de placement sobre el layout del prototipo de procesador desarrollado. Distribución automática de macrobloques con FP_SIZING: "absolute" y FP_ASPECT_RATIO: "2".

4.3. Resultado obtenido de la síntesis del prototipo

Como apartado final, revisaremos las características específicas del modelo de procesador que ha sido creado para evaluar las características de *IIC-OSIC-TOOLS* y OpenLane. En la siguiente Tabla 1 se recoge una serie de parámetros de interés que han sido extraídos de la interfaz gráfica de OpenROAD y los distintos archivos de reportes generados en el proceso. Se puede comprobar cómo tanto la memoria de datos como la memoria de programa desarrollan un periodo de *routing* considerablemente más largo que el resto de módulos, debido a la significativa mayor cantidad de elementos instanciados y redes de conexión. Salvo en el caso de la memoria de datos y la red de interconexión, el proceso de *routing* ocupa más del 50% del tiempo total invertido, dándose el máximo caso en el registro de datos con un 76.97%.

Tabla 1. Características destacadas del procesador.

	Tiempo total de ejecución	**Tiempo de ruteo**	**Tamaño del diseño (um x um)**	**Número de instancias**	**Número de redes**	**FP_CORE_UTIL**	**PL_TARGET_DENSITY**
ALU	2’ 33’’	1’ 56’’	200 x 200	1302	1371	50	0.6
Unidad de control	39’’	23’’	200 x 200	101	120	50	0.6
Codificador de instrucciones	35’’	19’’	200 x 200	168	202	50	0.6
Memoria de datos	47’ 58’’	22’ 13’’	1379.355 x 1390.075	48931	42679	20	0.25
Contador de programa	1’ 3’’	46’’	200 x 200	130	146	50	0.6
Registro de datos	8’ 54’’	6’ 51’’	347.92 x 358.64	5924	5854	50	0.6
Red de interconexión	29’ 25’’	2’ 46’’	1927.605 x 2321.635	1341	529	50	0.6

Aunque la red de interconexión sea el diseño más grande y contenga a los seis restantes, el tiempo invertido en el *routing* es proporcionalmente el mínimo —un 9.41% del tiempo total— debido a que en cada uno de los macrobloques ya ha realizado previamente esta fase, limitándose solo al *routing* entre estos módulos y las celdas estándar instanciadas en el nivel superior. Teniéndose en cuenta que en primeras ejecuciones se trató de sintetizar todo el sistema de una sola vez sin la definición de macrobloques, y que estos intentos resultaron en periodos de ejecución de hasta tres horas desembocadas en error por la alta congestión del *routing*, es claramente evidente los beneficios de instanciar los submódulos como cajas negras. Otros motivos por los que la memoria de datos y la red de interconexión tienen los periodos de desarrollo más largos son su considerable mayor tamaño con respecto a los demás; y particularmente para la red de interconexión el análisis DRC le ha tomado 18' 7'' —el 61.59% del tiempo total—.

Centrándonos un poco más en las características de la memoria de datos, debemos comentar que ha sido el módulo con mayor número de dificultades a la hora de implementarse. Originariamente, se pretendía que su capacidad de almacenamiento fuese de 4KiB —1024 espacios de memoria de 32 bits cada uno—, sin embargo, todas las configuraciones planteadas concluían en un error de congestión en el *routing*. Se había propuesto reducir la densidad de elementos instanciados y la utilización del núcleo, así como ampliar las dimensiones del diseño, pero aun así persistía el mismo error. Habíase llegado a considerar 3000 um x 3000 um únicamente para la memoria de datos con los parámetros `FP_CORE_UTIL: 20` y `PL_TARGET_DENSITY: 0.25`, y tras más de tres horas de ejecución no habían mejorado los resultados. Como el tamaño del almacenamiento no resulta imprescindible para los propósitos de nuestro procesador, finalmente se optó por reducir su capacidad a 1KiB, y con ello el número de celdas instanciadas en el módulo —permitiéndonos extraer su *layout* en un tiempo razonable—. Una conclusión que podríamos sacar de este escenario es el por qué las memorias de datos de acceso rápido son diseñadas como un elemento externo al procesador: por un lado, el gran tamaño del que requieren para sí mismas, y por otro, el calor que puedan generar todas sus instancias son alejadas del núcleo de procesamiento para aliviarlo.

Las Figuras 20 a 25 se corresponden con el *layout* visualizado en OpenROAD de cada uno de los macrobloques. La Figura 26 se trata de la composición del sistema completo, en la que podemos ver los seis módulos anteriores instanciados. La distribución ha sido detallada manualmente para que el sistema esté centrado en el chip y los macrobloques respeten una cierta simetría con el propósito de distribuir la densidad de componentes y la generación de

calor. Menciónese que al completar el flujo de este nivel se nos ha advertido de que existen ciertas violaciones de *fanout*, las cuales estaban presentes en la extracción del *layout* del registro de datos y la memoria de datos. No obstante, tal y como se concluye en esta discusión [49], es algo que no debe resultar preocupante siempre que no se den violaciones de capacitancia y *slew*. Por otro lado, pese a completarse sin inconvenientes el análisis DRC, analizando el archivo ***`manufacturability.rpt`*** referido al diseño de la red de interconexión, encontramos que una violación ***`"Diffussion contact to gate < 0.55um (licon.11)"`*** ha sido detectada 2297158 de veces. Puesto que este inconveniente no se ha dado para ninguno de los seis macrobloques, y basándome en las discusiones [50] y [51], intuimos que se tratan de falsos positivos debido a que el identificador del área de las celdas estándar genera un conflicto en el DRC. Puesto que Magic crea su propio modelo en GDSII y ejecuta el análisis DRC, y KLayout también su propio GDSII y los compara, es posible que sean estos falsos errores los que detenían la finalización de la XOR Tool. En cualquier caso, y pese a estas objeciones, el flujo es completado con gran éxito.

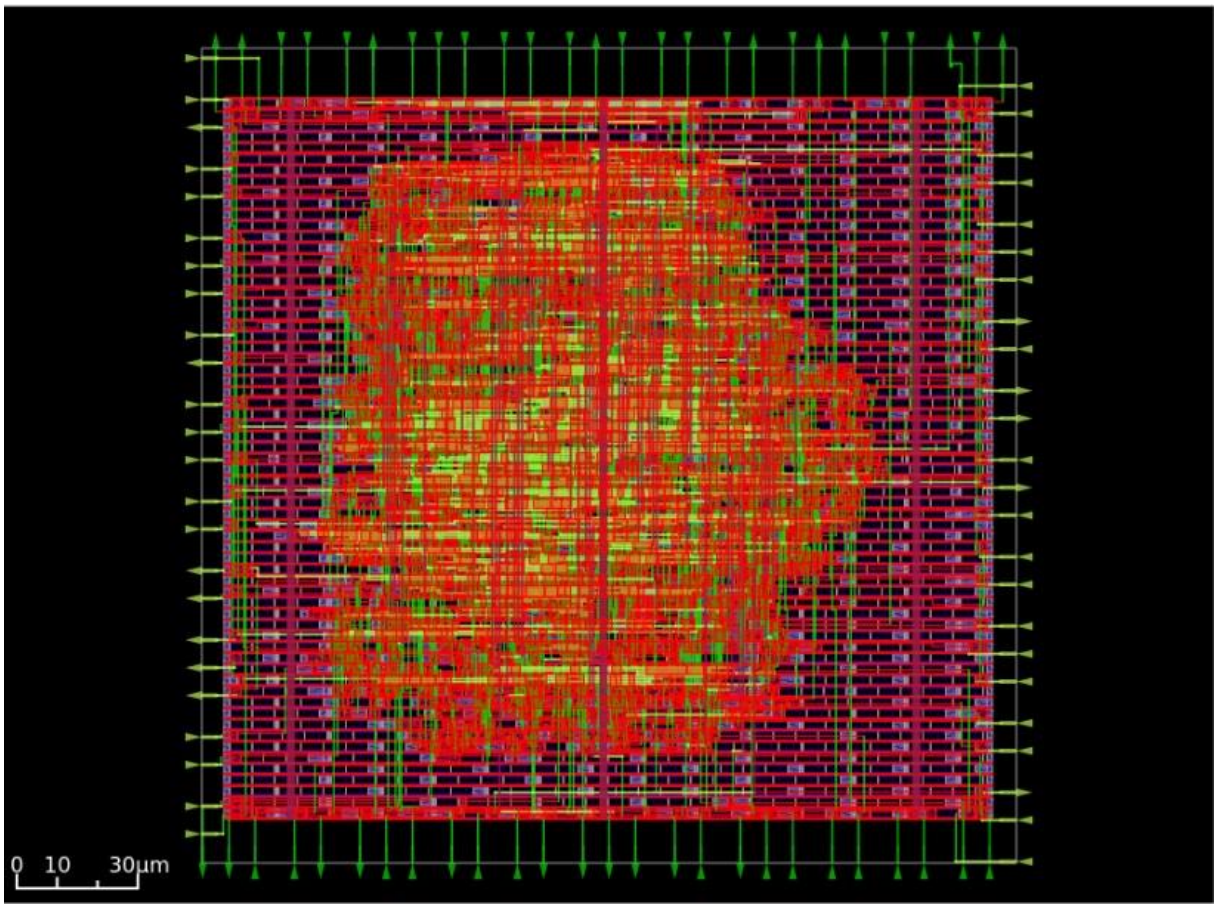

Figura 20. Layout de la unidad aritmética-lógica. Visualización con OpenROAD.

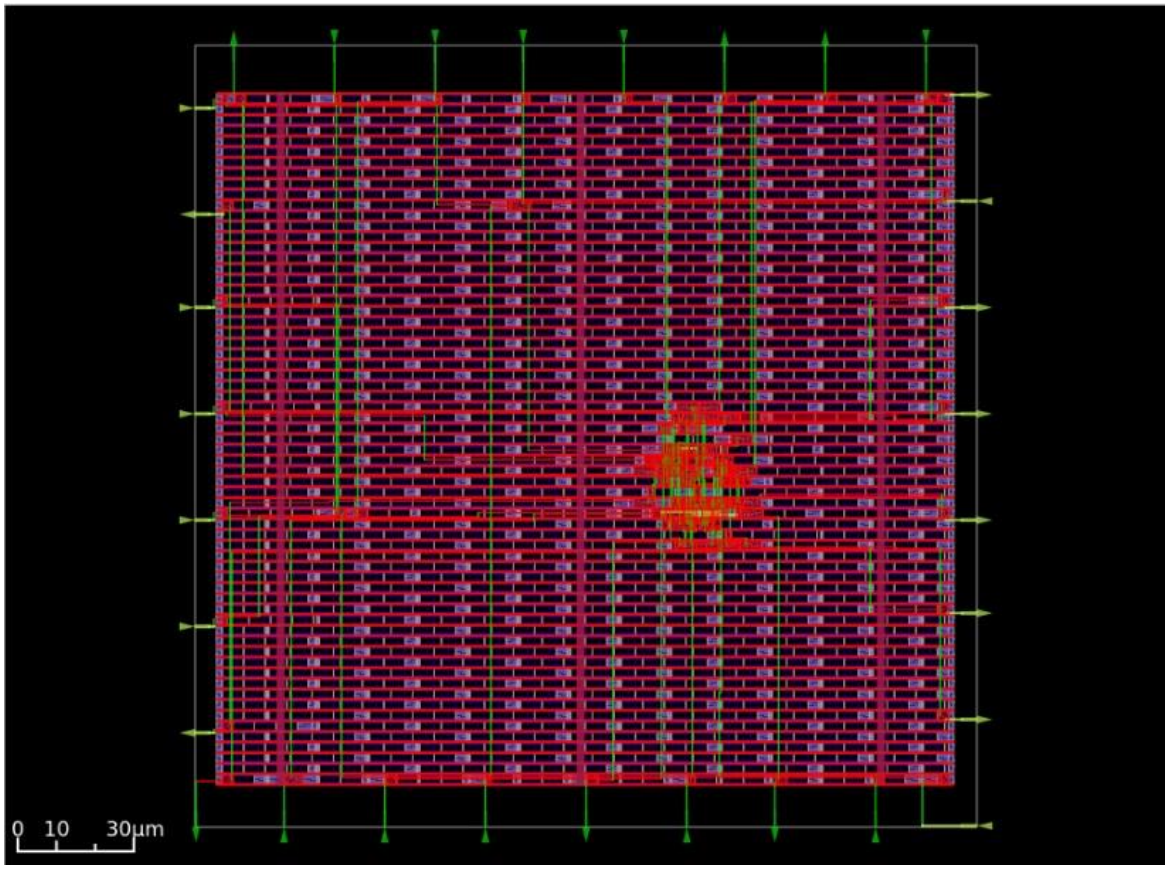

Figura 21. Layout de la unidad de control. Visualización con OpenROAD.

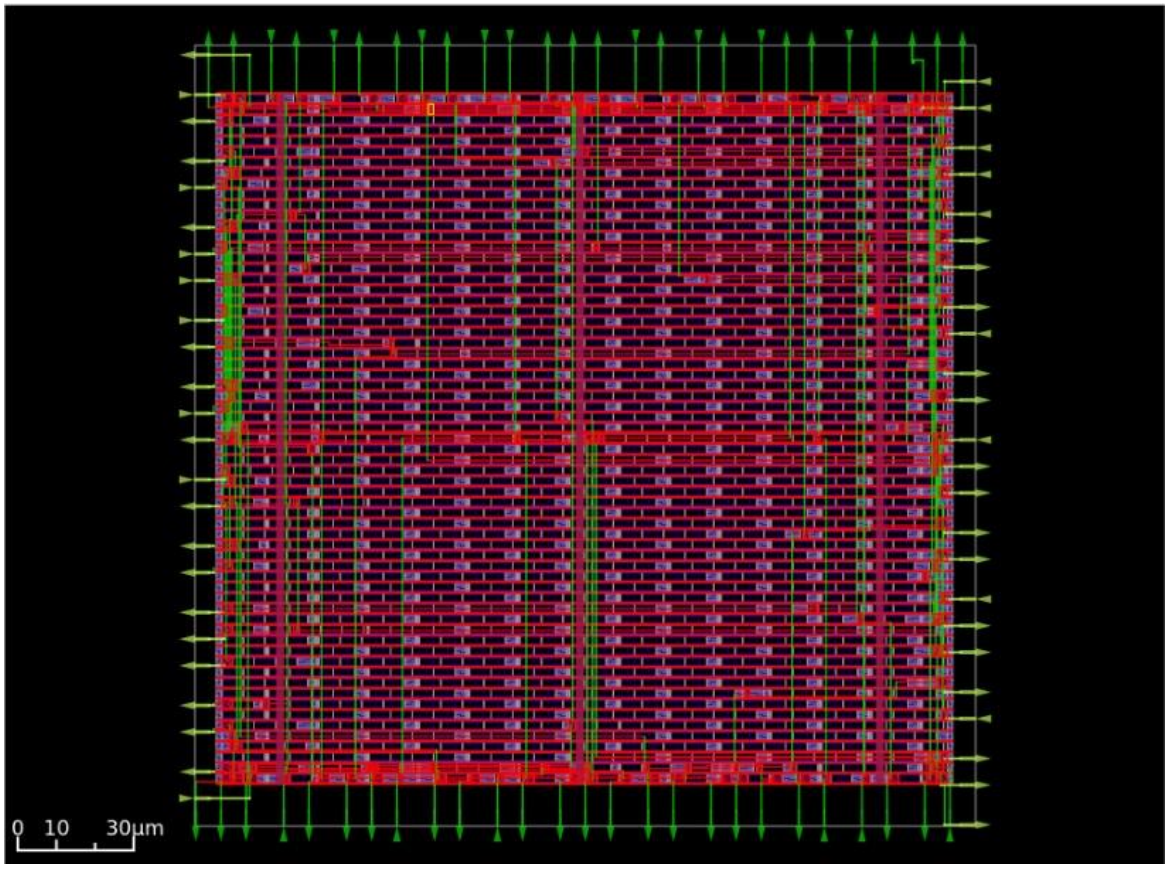

Figura 22. Layout del codificador de instrucciones. Visualización con OpenROAD.

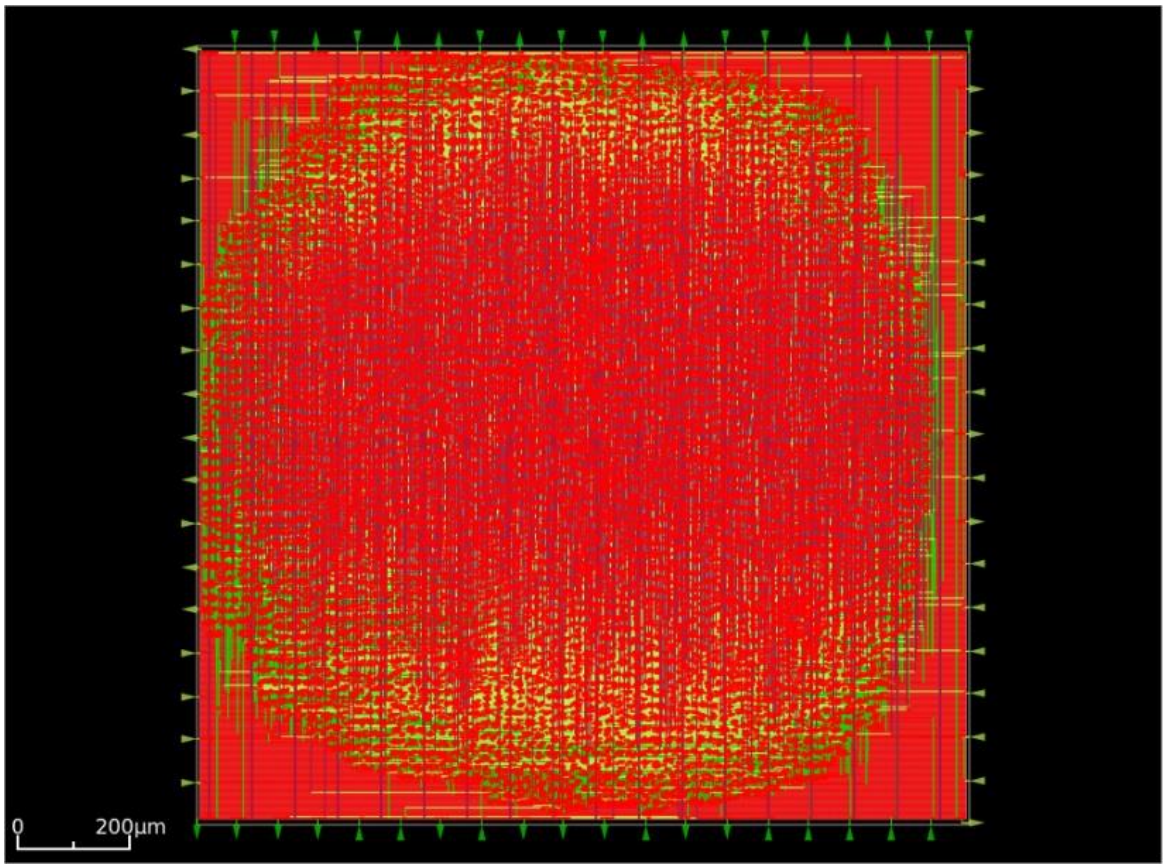

Figura 23. Layout de la memoria de datos. Visualización con OpenROAD.

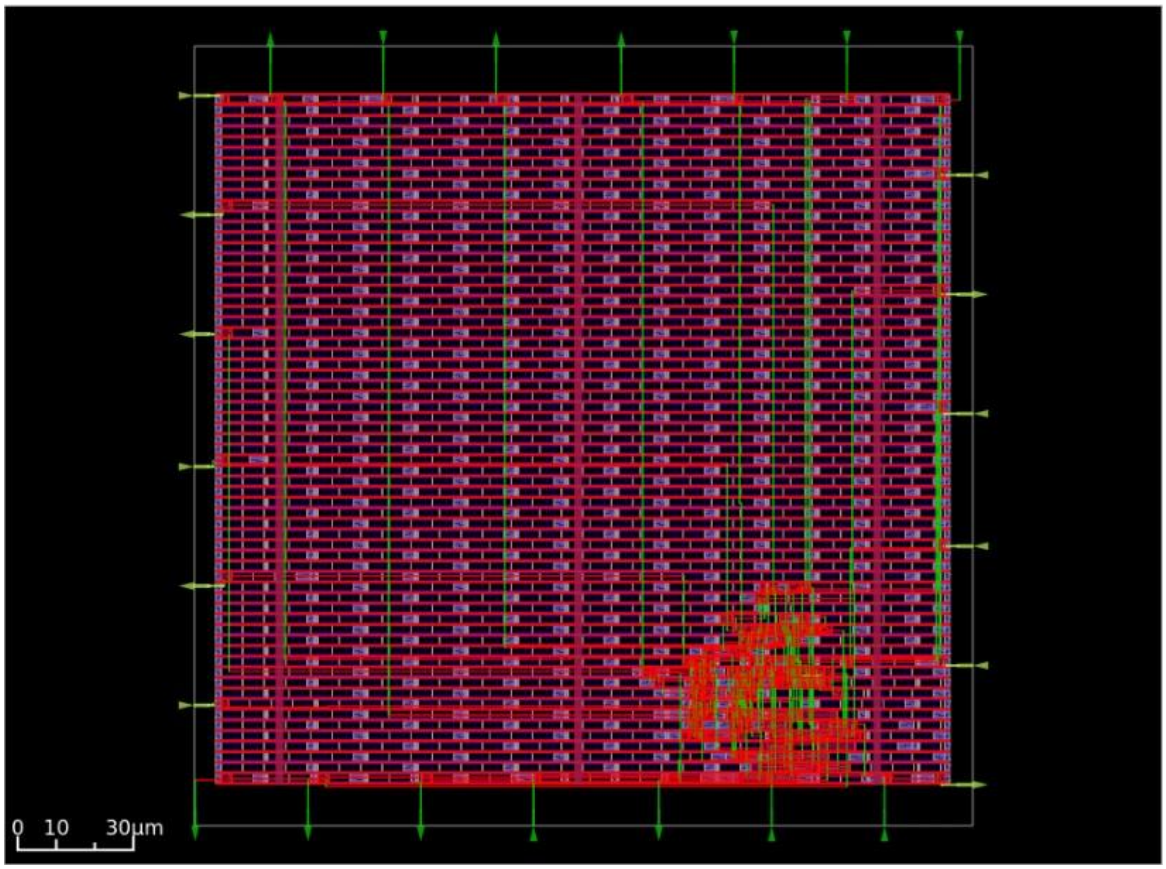

Figura 24. Layout del contador de programa. Visualización con OpenROAD.

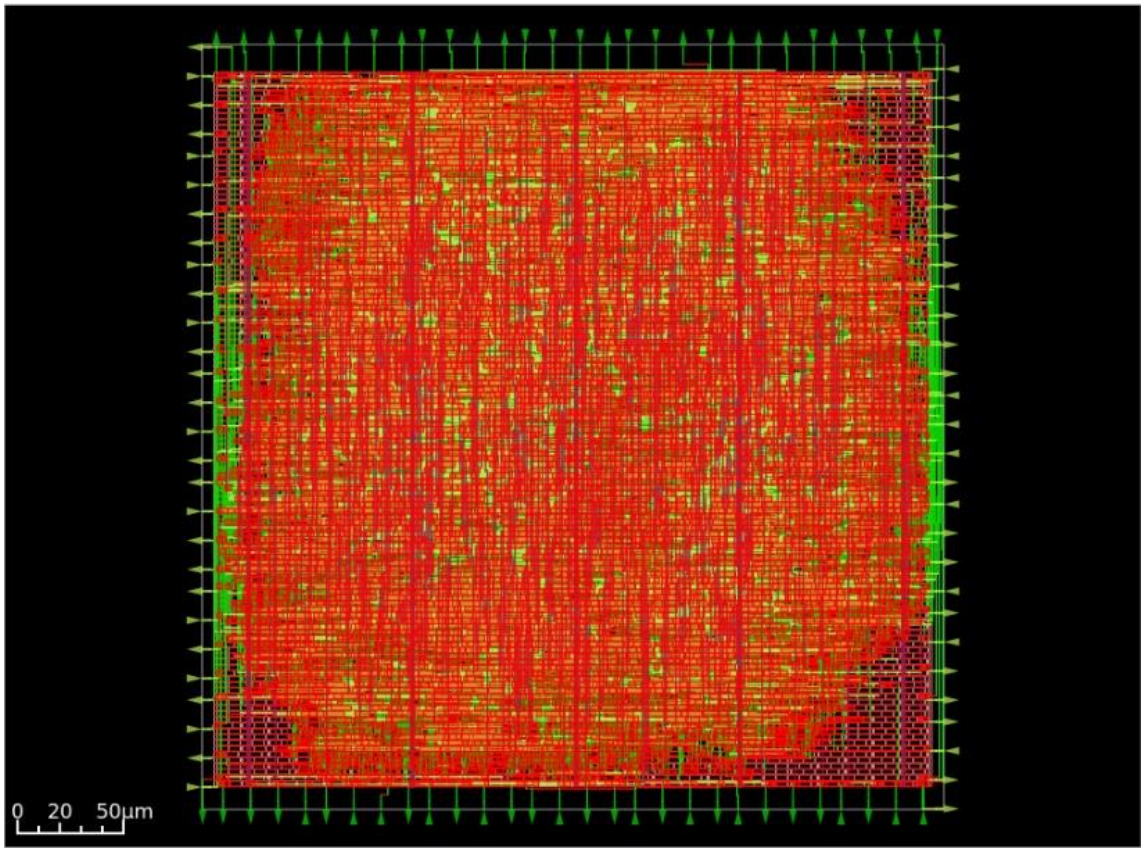

Figura 25. Layout del registro de datos. Visualización con OpenROAD.

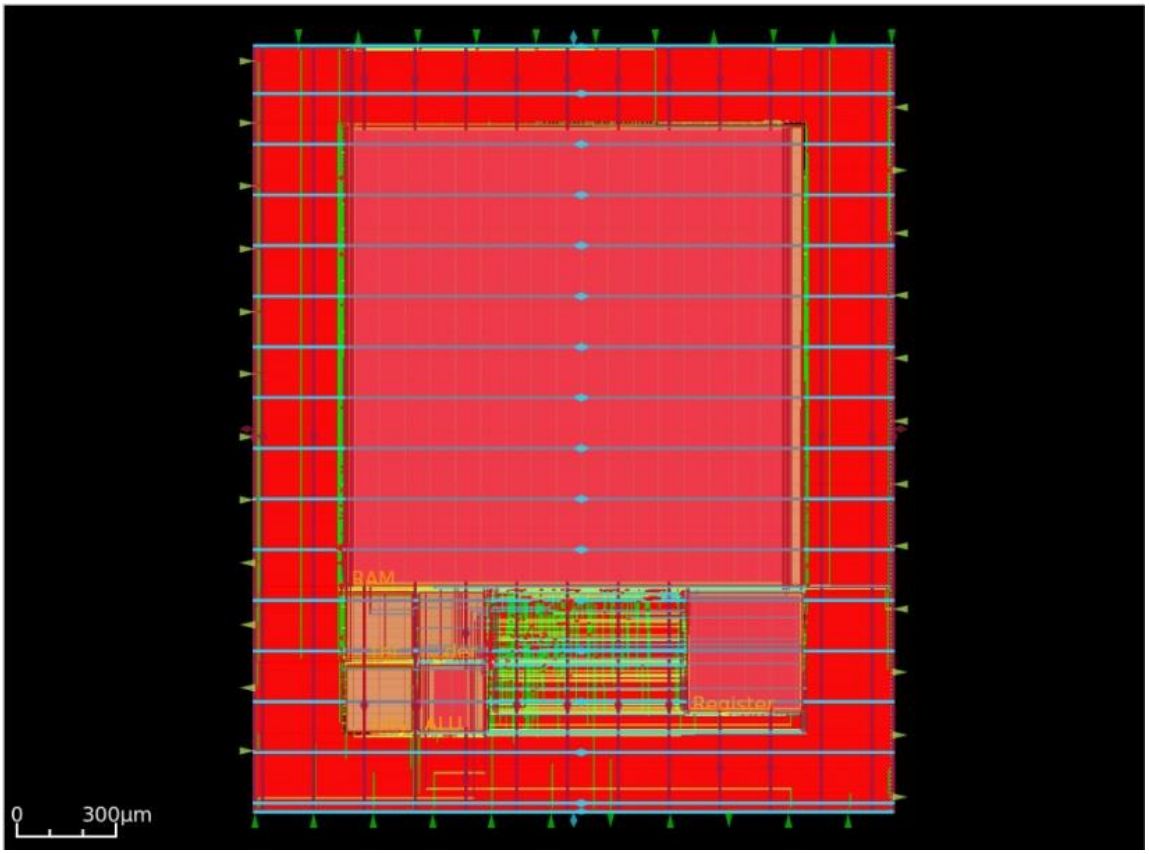

Figura 26. Layout del sistema completo. Visualización con OpenROAD.

5. Conclusión

Con la realización de este proyecto se han puesto de manifiesto la viabilidad y la capacidad de *IIC-OSIC-TOOLS*, como entorno de trabajo, y la de OpenLane, como herramienta de desarrollo, para completar el flujo de diseño digital y obtener el *layout* final de un modelo para su futura implementación como ASIC. Tras instruir en las etapas que componen el proceso desarrollo, se han expuesto las características de un diseño ilustrativo, un prototipo básico de procesador basado en la arquitectura RISC-V, sobre el que analizar resultados y que se espera que sirva como ejemplo de orientación en la labor de nuevos usuarios que quieran hacer uso de este software. Además, habiendo fundamentado la elaboración de este trabajo en programas de código abierto, tratamos de fomentar el foco de atención sobre este tipo de iniciativas que se desarrollan, esencialmente, gracias al interés de una comunidad activa. Cuantas más personas encuentren interés en los programas presentados y más peso ganen dentro del sector tecnológico, mayores beneficios y soporte podrá darse a los nuevos desarrolladores para seguir creciendo.

No obstante, y por encima de todo, el propósito de esta investigación es que el material mostrado pueda servir como manual de iniciación. Se pretende que cualquier persona que disponga de los conocimientos necesarios de Verilog pueda ejecutar con éxito, y en apenas unas horas, el desarrollo completo del flujo y extraer el *layout* de su propio diseño. Se han consultado múltiples manuales de referencia y foros de desarrolladores para tratar de presentar de manera organizada cuál es el procedimiento inmediato para hacer uso de este software, determinar cuáles son las variables y las configuraciones que pueden resultar más útiles de manejar para un primer proyecto, y como hacer frente a todos los problemas y errores que han surgido durante la elaboración de esta guía.

Nosotros mismos hemos seguido las instrucciones enumeradas en este documento y hemos demostrado que sí es posible lograr la extracción de los archivos necesarios para la implementación física de un circuito integrado. Aunque debamos dejar constancia que ha sido necesario saltar la comparativa de XOR Tool entre archivos GDSII, se ha logrado completar el flujo de diseño digital exitosamente. Llegar un paso más allá sería seguir ahondando en las funcionalidades de OpenLane para tratar de aproximar las optimizaciones sobre el diseño en desarrollo hasta los límites de la excelencia. O bien, alguien interesado por la implementación física definitiva de su diseño, podría decidirse a participar en un proyecto como chipIgnite [52] de eFabless.

Apéndice I. Resumen de instrucciones del proyecto

En este apartado se concentran de manera muy resumida y simplificada los contenidos de las secciones 4.2.1, 4.2.2 y 4.2.3 para completar la ejecución del flujo de diseño digital en el entorno de *IIC-OSIC-TOOLS*.

1. Descargar *Docker Descktop* (https://www.docker.com/products/docker-desktop/) y el contenido de *IIC-OSIC-TOOLS* (https://github.com/iic-jku/IIC-OSIC-TOOLS).
2. Instalar y ejecutar *Docker Descktop* en nuestro ordenador.
3. Entre los archivos de *IIC-OSIC-TOOLS*, ejecutar `./start_x.bat` desde la consola de comandos del sistema.
4. En la interfaz de *Docker Descktop*, acceder a la pestaña "*Images*" e inicializar la imagen recién instalada. En el menú de configuración establecer 80 en "*Host port :80/tcp*" y pulsar "*Run*".
5. Inicializar el contenedor desde la ventana "*Containers*" y pulsar en el puerto "*80:80*" para entrar al entorno. La contraseña de acceso es `abc123`.
6. Dentro del entorno, crear la descripción HDL del diseño y su *testbench* en archivos de texto con el formato específico del leguaje (Verilog en el caso de este tutorial).
7. Apuntando al directorio de trabajo, ejecutar los siguientes tres comandos para la compilación y simulación del diseño:

```
iverilog -o <nombre_diseño>.vvp <nombre_diseño_testbench>.v
vvp <nombre_diseño>.vvp
gtkwave <nombre_diseño>.vcd
```

8. Crear un nuevo directorio de trabajo `<carpeta_diseño>` e incluirlo a los diseños de OpenLane:

```
flow.tcl -design <carpeta_diseño> -init_design_config -add_to_designs
```

9. Copiar en `<carpeta_diseño>/src` la descripción HDL del diseño y establecer adecuadamente las variables de configuración de OpenLane en `config.json`.
10. Ejecutar el siguiente comando para iniciar el proceso automático del flujo de diseño:

```
flow.tcl -design <carpeta_diseño> -config_file <carpeta_diseño>/config.json
```

11. Para visualizar los resultados con OpenRoad:

```
flow.tcl -design <carpeta_módulo> -tag <RUN_...> -gui
```

Apéndice II. Variables de configuración para OpenLane

En esta sección se recogen todas las variables de configuración de OpenLane empleadas y consideradas en el desarrollo del flujo de nuestro prototipo de procesador. La lista completa de opciones puede ser consultada en [53].

CLOCK_PERIOD: Periodo de la señal de reloj —en nanosegundos— con el que se desarrollará el flujo de diseño.

CLOCK_PORT: Nombre del puerto del diseño en el que será introducida la señal de reloj.

CORE_AREA: Área específica del núcleo —área del chip menos los márgenes—. Establecer ***FP_SIZING: "absolute"*** y determinar las cuatro esquinas de un rectángulo como ***"x0 y0 x1 y1"*** —en micrómetros—.

DESIGN_IS_CORE: Controla las láminas usadas para la red de potencia. Establecer ***false*** si el diseño es un macrobloque dentro del chip completo, o ***true*** en otro caso.

DESIGN_NAME: Nombre del módulo superior del diseño considerado en la ejecución del flujo.

DIE_AREA: Área específica disponible para el *floorplaning*. Establecer ***FP_SIZING: "absolute"*** y determinar las cuatro esquinas de un rectángulo como ***"x0 y0 x1 y1"*** —en micrómetros—.

EXTRA_GDS_FILES: Especifica la ubicación de los archivos GDSII de los macrobloques implementados en el diseño.

EXTRA_LEFS: Especifica la ubicación de los archivos LEF de los macrobloques implementados en el diseño.

FP_ASPECT_RATIO: Factor de forma del diseño —cociente entre el alto y el ancho—. Por defecto igual a ***1***.

FP_CORE_UTIL: Porcentaje de utilización del núcleo —establecer un valor en 0 y 100—. Por defecto es igual a ***50***.

FP_PDN_CORE_RING: Activa la adicción de un anillo alrededor del diseño para la red de potencia. Por defecto es ***false***.

FP_SIZING: Configura el uso de medidas relativas acorde a la densidad de elementos, o permite la entrada de valores absolutos. Acepta como parámetros ***"relative"*** o ***"absolute"***.

MACRO_PLACEMENT_CFG: Especifica la ubicación de un archivo ".cfg" con la posición concreta de los macrobloques instanciados.

PL_MACRO_CHANNEL: Espacio de separación —en micrómetros— entre macrobloques. El formato es ***"<horizontal> <vertical>"***.

PL_MACRO_HALO: Espacio —en micrómetros—desde el macrobloque al halo que lo aísla. El formato es ***"<horizontal> <vertical>"***.

PL_TERGET_DENSITY: Densidad de celdas colocadas durante el *placement*. Representa distribución de las celdas sobre el núcleo —***1*** es muy agrupadas y ***0*** muy dispersas—. Por defecto es igual a ***(FP_CORE_UTIL+10)/100***.

RT_MAX_LAYER: Especifica la capa más elevada disponible para el *routing*.

RUN_CTS: Habilita la ejecución la síntesis de la red del reloj. Por defecto es ***true***.

RUN_KLAYOUT_XOR: Habilita la ejecución la XOR Tool de KLayout. Por defecto es ***true***.

VERILOG_FILES: Especifica la ubicación de los archivos Verilog considerados para la ejecución del flujo.

VERILOG_FILES_BLACKBOX: Especifica la ubicación de los archivos Verilog que serán implementados como cajas negras. Es necesario que estos archivos contengan una línea con el texto ***/// sta-blackbox*** para que no entren en conflicto con procesos del flujo de diseño

Referencias

[1] Flaherty, N. (20 de enero de 2023). *Top ten semiconductor vendors in 2022*. eeNews Europe. https://www.eenewseurope.com/en/top-ten-semiconductor-vendors-in-2022/

[2] Saif M. Khan and Alexander Mann, "AI Chips: What They Are and Why They Matter" (Center for Security and Emerging Technology, April 2020), cset.georgetown.edu/research/ai-chips-what-they-are-and-why-they-matter/. https://doi.org/10.51593/20190014

[3] Synopsys, Inc. (s.f.). *What is Electronic Design Automation (EDA)?*. https://www.synopsys.com/glossary/what-is-electronic-design-automation.html

[4] Synopsys, Inc. (s.f.). *Synopsys Unveils Galaxy IC Compiler – Next-Generation Physical Design Solution*. https://news.synopsys.com/index.php?s=20295&item=122377

[5] Pretl, H., & Zachl, G. (2023). GitHub repository of the IIC-OSIC-TOOLS [Computer software]. Recuperado el 13 de agosto de 2023 en https://github.com/iic-jku/IIC-OSIC-TOOLS

[6] Kassem, M., & Farid, K. (8 de agosto de 2022). GitHub repository of the foss-asic-tools. Recuperado el 14 de agosto de 2023 en https://github.com/efabless/foss-asic-tools

[7] Google & SkyWater Technology. (s.f.). *Welcome to SkyWater SKY130 PDK's documentation!*. Recuperado el 7 de agosto de 2023 de https://skywater-pdk.readthedocs.io/en/main/index.html

[8] Synopsys, Inc. (s.f.). *What is Analog Design?*. https://www.synopsys.com/glossary/what-is-analog-design.html

[9] Liening, J., & Scheible, J. (2020). *Fundamentals of Layout Design for Electronic Circuits*. (pp. 130, 151-153). Springer. https://doi.org/10.1007/978-3-030-39284-0

[10] DeLisle, J. (2 de diciembre de 2020). *What Is Digital IC Design?*. All About Circuits. https://www.allaboutcircuits.com/technical-articles/what-is-digital-ic-design/

[11] Synopsys, Inc. (s.f.). *What is a Process Design Kit and How Does it Work?*. https://www.synopsys.com/glossary/what-is-a-process-design-kit.html

[12] Rouse, M. (13 de octubre de 2014). *Top-Down Design*. Techopedia. https://www.techopedia.com/definition/9744/top-down-design

[13] Shariat-Yazdi, R. (2001). *Mixed Signal Design Flow: A mixed signal PLL case study*. [Tésis de maestría, University of Waterloo]. https://uwspace.uwaterloo.ca/bitstream/handle/10012/916/rshariatyazdi2001.pdf?sequence=1&isAllowed=y

[14] M. Anderson, J. Wernehag, A. Axholt and H. Sjoland, "Teaching top down design of analog/mixed signal ICs through design projects," *2007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, Milwaukee, WI, USA, 2007, pp. T1C-1-T1C-4, doi: 10.1109/FIE.2007.4417874.

[15] Malik, S. (14 de abril de 2023). *RTL Design: A Comprehensive Guide to Unlocking the Power of Register-Transfer Level Design*. Wevolver. https://www.wevolver.com/article/rtl-design-a-comprehensive-guide-to-unlocking-the-power-of-register-transfer-level-design

[16] Xiu, L. (2008). *VLSI Circuit Design Methodology Demystified: A Conceptual Taxonomy*. John Wiley & Sons, Inc. https://picture.iczhiku.com/resource/eetop/ShiyDTRFDksWGXMV.pdf

[17] Harris, S., & Harris, D. (2022). *Digital Design and Computer Architecture*. (RISC-V Edition, p. 218). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-820064-3.00004-0

[18] Avedillo, M., Castro, R., Fernández, F., y Jiménez, C. (s.f.). *RTL Synthesis*. Metodologías de Diseño y Herramientas CAD. Universidad de Sevilla. Recuperado el 7 de agosto de 2023 en http://www2.imse-cnm.csic.es/elec_esi/asignat/MHCAD/tema1_digital/RTL_synthesis.html

[19] S. Gayathri and T. C. Taranath, "RTL synthesis of case study using design compiler," *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, Mysuru, India, 2017, pp. 1-7, doi: 10.1109/ICEECCOT.2017.8284603.

[20] Synopsys, Inc. (s.f.). *What is Static Timing Analysis (STA)?*. https://www.synopsys.com/glossary/what-is-static-timing-analysis.html

[21] Kahng, A., Lienig, J., Markov, I., & Hu, J. (2011). *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer. https://doi.org/10.1007/978-90-481-9591-6

[22] Stroud, C., Wang, L.-T., & Chang, Y.-W. (2009). Chapter 1 – Introduction. In Wang, L.-T., Chang, Y.-W., & Cheng, K.-T. (Ed.), *Electronic Design Automation*. (pp. 1-38). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-374364-0.50008-4

[23] Chen, T.-C., & Chang, Y.-W. (2009). Chapter 10 – Floorplaning. In Wang, L.-T., Chang, Y.-W., & Cheng, K.-T. (Ed.), *Electronic Design Automation*. (pp. 575-634). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-374364-0.50017-5

[24] Avedillo, M., Castro, R., Fernández, F., y Jiménez, C. (s.f.). *Layout*. Metodologías de Diseño y Herramientas CAD. Universidad de Sevilla. Recuperado el 7 de agosto de 2023 en http://www2.imse-cnm.csic.es/elec_esi/asignat/MHCAD/tema1_digital/Layout.html

[25] Mukundan, S. (12 de octubre de 2013). *Physical Design Flow V: Physical Verification*. VLSI Professional Network. https://vlsi.pro/physical-design-flow-v-physical-verification/

[26] The OpenLane Documentation. https://openlane.readthedocs.io/en/latest/index.html

[27] Icarus Verilog. https://steveicarus.github.io/iverilog/

[28] GTKWave. (s.f.). *GTKWave 3.3 Wave Analyzer User's Guide* [Archivo PDF]. https://gtkwave.sourceforge.net/gtkwave.pdf

[29] Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/

[30] ABC: A System for Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc/

[31] OpenSTA: Open System Testing Architecture. http://opensta.org/

[32] Gaber, M., Manarabdelaty, M., & Shalan, M. (17 de mayo de 2017). GitHub repository of Fault. Recuperado el 4 de septiembre de 2023 en https://github.com/AUCOHL/Fault

[33] OpenROAD documentation. https://openroad.readthedocs.io/en/latest/index.html

[34] Team VLSI. (29 de agosto de 2020). *Well Tap Cells in Physical Design*. Team VLSI. Recuperado el 5 de septiembre de 2023 en https://teamvlsi.com/2020/08/well-tap-cell-in-asic-design.html

[35] Team VLSI. (30 de agosto de 2020). *DeCap Cells in Physical Design | Use of DeCap Cells in PD*. Team VLSI. Recuperado el 5 de septiembre de 2023 en https://teamvlsi.com/2020/08/decap-cell-in-physical-design.html

[36] M. Shalan and T. Edwards, "Building OpenLane: A 130nm OpenROAD-based Tapeout-Proven Flow: Invited Paper," *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, San Diego, CA, USA, 2020, pp. 1-6.

[37] Edwars, T. (29 de mayo de 2020). *Magic VLSI Layout Tool*. Open Circuit Designs. Recuperado el 6 de septiembre de 2023 en http://opencircuitdesign.com/magic/index.html

[38] KLayout Layout Viewer and Editor. https://www.klayout.de/

[39] Bailey, M., & Gaber, M. (8 de mayo de 2023). GitHub repository of cvc. Recuperado el 6 de septiembre de 2023 en https://github.com/d-m-bailey/cvc

[40] RISC-V International. (s.f.). *History of RISC-V*. https://riscv.org/about/history/

[41] Waterman, A., & Asanovic, K. (13 de diciembre de 2019). *The RISC-V Instruction Set Manual: Volume I: Unprivileged ISA* [Archivo PDF]. https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

[42] González, J. (22 de abril de 2022). *Sesión Laboratorio 10: Práctica 4-2*. GitHub. https://github.com/myTeachingURJC/2019-20-LAB-AO/wiki/L10:-Practica-4

[43] Felipe Machado. (2 de septiembre de 2021). *Arquitectura de computadores: RISC-V* [Archivo de Vídeo]. YouTube. https://www.youtube.com/playlist?list=PLSUmB2yTypWGcPlfTymmaQ3HWxvamxfoW

[44] Docker: Accelerated Container Application. https://www.docker.com/

[45] Nyasulu, P., & Knight, J. (5 de octubre de 2003). *Introduction to Verilog* [Archivo PDF]. Carleton University. https://www.doe.carleton.ca/~gallan/478/pdfs/PeterVrlR.pdf

[46] Verilator User's Guide — Verilator 5.016 documentation. https://verilator.org/guide/latest/index.html

[47] The OpenROAD Project. (s.f.). *The OpenROAD Project*. GitHub. https://github.com/The-OpenROAD-Project

[48] Efabless Corporation and contributors (s.f.). *Hierarchical chip design (with macros)*. The OpenLane Documentation. Recuperado el 21 de octubre de 2023 en https://openlane.readthedocs.io/en/latest/tutorials/digital_guide.html

[49] antonblanchard. (18 de julio de 2022). @vvvverre as @maliberty says, you want to pay more attention to the cap/slew violations. It looks like your fan [Comentario en la página web *Max fanout Limit ? · Issue #1206 · The-OpenROAD-Project/OpenLane*]. GitHub. https://github.com/The-OpenROAD-Project/OpenLane/issues/1206

[50] w32agobot. (3 de noviembre de 2022). Flow unexpectedly fails with DRC error Diffusion contact to gate < 0.055um (licon.11). The boundary boxes in results/final/gds are [Comentario en la página web *Standardcell boundary redefined, flow fails at DRC · Issue #1470 · The-OpenROAD-Project/OpenLane*]. GitHub. https://github.com/The-OpenROAD-Project/OpenLane/issues/1470

[51] RTimithyEdwards. (26 de agosto de 2022). @jainsoumil2 : I do not know what you have done, but this is not an error. The presence of the area_sc [Comentario en la página web *At least one cell (sky130_fd_sc_hd__clkdlybuf4s15_1) violates SkyWater's own DRC rules and needs to be fixed. · Issue #261 · google/skywater-pdk*]. GitHub. https://github.com/google/skywater-pdk/issues/261

[52] efabless.com, https://efabless.com/

[53] Efabless Corporation and contributors (s.f.). *Flow Configuration Variables*. The OpenLane Documentation. Recuperado el 24 de octubre de 2023 en https://openlane.readthedocs.io/en/latest/reference/configuration.html