# Escribiendo código RTL con Rust
# Writing RTL code with Rust


Trabajo de Fin de Grado
Curso 2024–2025

Autor
**Óscar López Centenera**
**Christina Cabañés Sorensen**

Director
**Alberto Antonio del Barrio García**

Colaborador
**Raul Murillo Montero**

**Grado en Ingeniería de Computadores**
**Facultad de Informática**
**Universidad Complutense de Madrid**

# Escribiendo código RTL con Rust
# Writing RTL code with Rust

Trabajo de Fin de Grado en Ingeniería de Computadores


## Autor
**Óscar López Centenera**
**Christina Cabañés Sorensen**

## Director
**Alberto Antonio del Barrio García**

## Colaborador
**Raul Murillo Montero**


**Convocatoria:** *Junio 2025*


**Grado en Ingeniería de Computadores**
**Facultad de Informática**
**Universidad Complutense de Madrid**


**26 de Mayo de 2025**

# Dedicatoria

*A nuestros padres y profesores:*

*Gracias por acompañarnos en cada paso de este camino.*
*A nuestros padres, por su apoyo constante, su esfuerzo y por creer en nosotros incluso en los momentos difíciles.*
*A nuestros profesores, por compartir su conocimiento, su tiempo y por guiarnos con dedicación y paciencia.*

*A nuestros compañeros de clase y de voleibol:*
*Gracias por ser parte esencial de esta etapa.*
*A nuestros compañeros de clase, por su compañerismo, sus ideas compartidas y los momentos de estudio y risas que hicieron más llevadero el camino.*
*Al equipo de voleibol de Químicas, por enseñarnos el valor del trabajo en equipo, la disciplina y el espíritu deportivo dentro y fuera de la cancha.*

*Este logro no sería posible sin ustedes.*
*Con todo nuestro agradecimiento, les dedicamos este momento especial.*

# Resumen


## Escribiendo código RTL con Rust

El objetivo de este Trabajo de Fin de Grado es explorar las posibilidades prácticas del lenguaje de programación de alto nivel Rust en el diseño de circuitos digitales, utilizando la biblioteca Rust-HDL. A través del desarrollo de proyectos concretos (como un sumador combinacional, un reconocedor de patrones y un multiplicador escalar secuencial) se pretende evaluar la eficacia de Rust como herramienta alternativa a los lenguajes tradicionales de descripción hardware, especialmente Verilog.

Para ello, se han implementado versiones equivalentes de cada uno de los diseños tanto en Rust-HDL como en Verilog, y se han realizado comparaciones en términos de consumo de recursos, tiempos de ejecución y facilidad de desarrollo. Las herramientas utilizadas para el análisis incluyen simuladores como GTKWave y herramientas de síntesis como Yosys, además de herramientas propias del ecosistema Rust como 'cargo run'y 'cargo test'.

Este trabajo tiene como objetivo principal determinar si Rust puede considerarse una opción viable y eficiente para el diseño hardware a nivel profesional, especialmente en proyectos de mayor escala.



## Palabras clave

Rust, Rust-HDL, Verilog, exploración, diseño, eficiencia, rendimiento, sintesis.

# Abstract


## Writing RTL code with Rust

The objective of this Bachelor's Thesis is to explore the practical capabilities of the low-level programming language Rust in the design of digital circuits, using the Rust-HDL library. Through the development of specific projects (such as a combinational adder, a pattern recognizer, and a sequential scalar multiplier) we aim to evaluate Rust's effectiveness as an alternative to traditional hardware description languages, particularly Verilog.

To this end, equivalent designs have been implemented in both Rust-HDL and Verilog, and comparisons have been made in terms of resource usage, execution time, and development ease. The analysis relies on tools such as GTKWave for simulation, Yosys for synthesis, and Rust's own toolchain via 'cargo run' and 'cargo test'.

The main goal of this work is to assess whether Rust can be considered a viable and efficient option for professional-scale hardware design projects.



## Keywords

Rust, Rust-HDL, Verilog, exploration, design, efficiency, performance, synthesis.

# Índice

# Índice de figuras

# Índice de tablas

Capítulo 1

# Introducción

## 1.1. Motivación

En la actualidad, se presta mucha atención al desarrollo y evolución de los lenguajes de programación de alto nivel, lo que a menudo deja en segundo plano a lenguajes más tradicionales. Aunque lenguajes como C y C++ siguen siendo ampliamente utilizados por su eficiencia y cercanía al hardware, también arrastran ciertas limitaciones que han cambiado poco con el tiempo. Esto ha dificultado, en parte, la exploración de enfoques más modernos en el diseño de sistemas digitales.

En este contexto, al descubrir la existencia de Rust, un lenguaje de programación creado por Graydon Hoare en 2010, que actualmente sigue en desarrollo, nos interesó mucho explorar las posibilidades que nos podría ofrecer. Rust es un lenguaje de programación de sistemas relativamente reciente y mucho más nuevo que los demás que ya conocíamos hasta el momento. Prometía resolver varias deficiencias detectadas en otros lenguajes como C y C++, como la gestión insegura de memoria o la dificultad para implementar concurriencia sin errores. Decía ser más éficaz, rápido y seguro que sus predecesores, utilizando un enfoque moderno y herramientas integradas de alta calidad.

Como estudiantes en el Grado en Ingeniería de Computadores, tenemos un especial interés en el desarrollo hardware. Por ello, estas promesas nos llamaron la atención y consideramos relevante poner a prueba el lenguaje en el contexto de desarrollar sistemas digitales, evaluando si Rust puede ser una alternativa realista y sostenible en este ámbito.

## 1.2. Objetivos

El objetivo principal de este trabajo es explorar el lenguaje de programación Rust mediante el desarrollo de distintos programas cortos y analizar sus características a través de métricas cuantitativas y cualitativas. La comparación de estos resultados con los obtenidos en otros lenguajes tradicionales de bajo nivel, pretende evaluar hasta qué punto Rust facilita el desarrollo hardware, especialmente en términos de curva de aprendizaje frente a lenguajes RTL y si hay una mayor facilidad para verificar los diseños.

Para realizar estas observaciones, utilizaremos varios programas (descritos en el siguiente apartado), los cuales nos ayudarán a comparar el código escrito en Rust con código equivalente escrito en Verilog.

En el ámbito de la programación, uno de los objetivos principales es lograr una alta eficiencia, ya sea en consumo de memoria, velocidad de ejecución o escalabilidad. Esta eficiencia impacta directamente en el rendimiento del sistema, el consumo energético y el aprovechamiento del hardware disponible.

En este sentido, este trabajo también se propone analizar si Rust, gracias a su control detallado del sistema y su modelo de memoria seguro, puede ser una alternativa viable en el desarrollo de proyectos que requieren un acceso cercano al hardware, como sistemas empotrados, controladores, videojuegos o aplicaciones de alto rendimiento.

## 1.3. Plan de trabajo

El plan de trabajo seguido para lograr los objetivos previamente mencionados es:

1. **Familiarización con el lenguaje:** Al ser un lenguaje nuevo para nosotros, la primera etapa de nuestro proyecto fue comprender en profundidad la gramática y estructura que debe seguir un código escrito en Rust. Para eso, recurrimos a los manuales oficiales de Rust y Rust-HDL, este último siendo una librería en Rust para diseñar circuitos digitales, la cual será clave para nuestro objetivo.

2. **Programas a utilizar:** Para conseguir comparaciones sustanciales con nuestro código Rust, a recomendación de nuestros tutores, decidimos usar estos tres programas:

    - **Icarus Verilog:** Un simulador que compila y ejecuta código Verilog. Con él escribiremos los códigos equivalentes a los escritos en Rust para luego compararlo con las herramientas siguientes.

- **Yosys:** Una herramienta de síntesis lógica de código abierto. Convierte diseños hechos en Verilog a una escructura de puertas lógicas optimizada, lo cual suele ser el primer paso para pasar un diseño a un circuito real.
- **GTKWave:** Un visualizador de señales de simulación. Muestra gráficamente las ondas generadas durante una simulación a partir de archivos .vcd, los cuales podemos generar a partir del código en Verilog o Rust mediante una herramienta integrada en el lenguaje.

3. **Primeros intentos:** Una vez instalados los programas a utilizar, empezamos a intentar crear un programa simple, un sumador de dos entradas, para poner en práctica toda la teoría estudiada.

4. **Creación de códigos más complejos:** Una vez comprobado que sabíamos usar todas las herramientas y que éstas funcionaban de la manera que necesitábamos, pasamos a desarrollar ideas más complejas que pondrían a prueba los límites de Rust.

El flujo de todas nuestras pruebas será escribir el código tanto en Rust como en Verilog, luego simular ambos códigos y visualizar los archivos .vcd en GTKWave para comprobar el comportamiento. Si está todo correcto, sintetizaremos con Yosys para generar una netlist que más adelante se puede implementar en componentes hardware como una FPGA.

Capítulo 2

# Estado de la Cuestión

En las últimas décadas, el diseño de hardware ha experimentado una evolución significativa, pasando de lenguajes de descripción de hardware (HDL) tradicionales como Verilog y VHDL a enfoques más modernos que buscan mejorar la productividad, mantenibilidad y reutilización del código. Esta transición ha sido impulsada por la necesidad de abordar la creciente complejidad de los sistemas digitales y la demanda de ciclos de desarrollo más rápidos.

En este contexto, han surgido diversas herramientas, tanto de código abierto como comerciales, que automatizan y facilitan el proceso de diseño hardware. A continuación, se presenta una revisión de algunas de las herramientas más relevantes en este ámbito.

## 2.1. Herramientas de Código Abierto

### 2.1.1. PyMTL

PyMTL (Python-based Hardware Modeling, Translation, and Simulation Framework) es un marco de trabajo de código abierto desarrollado en la Universidad de Cornell. Está diseñado para la generación, simulación y verificación de hardware, permitiendo a los diseñadores modelar sistemas a múltiples niveles de abstracción utilizando el lenguaje Python. PyMTL facilita la transición entre modelos funcionales y de nivel de transferencia de registros (RTL), y proporciona herramientas integradas para la simulación y verificación de diseños. Su tercera versión, PyMTL3, introduce mejoras en la modularidad y extensibilidad del marco, así como una representación intermedia en memoria que permite transformaciones y análisis más sofisticados. Derek Lockhart y Batten (2014); Shunning Jiang y Batten (2018a,b).

### 2.1.2. FloPoCo

FloPoCo (Floating-Point Core Generator) es una herramienta de código abierto que automatiza la generación de rutas de datos aritméticas personalizadas, especialmente para operaciones de punto flotante. No obstante, no trata los números denormalizados ni las excepciones, por lo que no cumple totalmente con el éstandar IEEE 754. Permite a los diseñadores especificar operaciones aritméticas de alto nivel y genera implementaciones optimizadas en HDL, considerando aspectos como la latencia y el uso de recursos. FloPoCo es particularmente útil en aplicaciones que requieren operaciones aritméticas complejas y de alto rendimiento, sin embargo, una limitación que presenta es que no genera controladores, por lo que no es posible generar operadores secuenciales con esta herramienta. Dinechin y Pasca (2011); Murillo et al. (2023a, 2024).

### 2.1.3. Bambu

Bambu es un marco de trabajo de síntesis de alto nivel (HLS) desarrollado en el Politécnico de Milán. Permite la conversión de descripciones en lenguaje C a implementaciones en HDL, facilitando el diseño de aplicaciones intensivas en memoria. Bambu soporta una amplia gama de construcciones de C y proporciona una arquitectura de memoria eficiente, así como la integración de unidades de punto flotante por medio de FloPoCo. Además, permite la personalización del flujo de síntesis mediante archivos de configuración XML y la generación automática de bancos de pruebas para la validación de los diseños. Pilato y Ferrandi (2013); Murillo et al. (2023b).

## 2.2. Herramientas Comerciales

Además de las herramientas de código abierto, existen soluciones comerciales ampliamente utilizadas en la industria para la automatización del diseño hardware. Entre ellas se encuentran:

- **Vivado HLS**: Desarrollado por Xilinx, permite la síntesis de alto nivel a partir de descripciones en C, C++ o SystemC, generando implementaciones optimizadas para FPGAs de Xilinx. Utilizado principalmente por ingenieros de hardware para mapear RTL generado en una FPGA standalone. Xilinx (2024b) El núcleo de Vivado tiene su origen en el proyecto de investigación abierto xPilot, desarrollado en la Universidad de California en Los Ángeles (UCLA) Cong et al. (2006). A partir de este proyecto se fundó la startup AutoESL, que fue adquirida posteriormente por Xilinx para integrar su tecnología en la herramienta Vivado HLS.
- **Vitis HLS**: También desarrollada por Xilinx, esta herramienta permite desarrolar acelerados a partir de código C, C++ o OpenCL. Está más enfocada

a ingenieros de software que buscan acelerar sus aplicaciones usando tarjetas FPGA conectadas por PCIe, sin necesidad de conocimientos profundos en RTL. Xilinx (2024a)

- **Intel HLS Compiler**: Herramienta de Intel que convierte descripciones en C++ en HDL, optimizadas para FPGAs de Intel. Intel Corporation (2024)
- **Synplify HLS**: Herramienta de síntesis de alto nivel desarrollada por Synopsys, orientada específicamente al diseño de hardware para FPGAs a partir de descripciones en lenguajes de alto nivel como C/C++ o SystemC. Synopsys Inc. (2024)
- **Synopsys Design Compiler**: Herramienta de síntesis lógica desarrollada por Synopsys que traduce descripciones en HDL (como Verilog o VHDL) a implementaciones optimizadas para ASICs. Se encarga de convertir descripciones RTL en puertas lógicas. Synopsys (2024)
- **Cadence Stratus**: Herramienta de síntesis de alto nivel de Cadence que permite la generación de hardware a partir de lenguajes de alto nivel como C/C++ y SystemC, facilitando el diseño para aplicaciones complejas. Systems
- **Cadence Genus Synthesis Solution**: Ofrece capacidades de síntesis RTL para diseños digitales complejos, con enfoque en rendimiento y eficiencia energética. Cadence Design Systems (2024)

Estas herramientas comerciales ofrecen soporte técnico, integración con otros productos del mismo proveedor y optimizaciones específicas para determinadas tecnologías de fabricación.

## 2.3. Comparativa y Consideraciones

La elección entre herramientas de código abierto y comerciales depende de diversos factores, como el presupuesto, los requisitos del proyecto, la familiaridad del equipo con las herramientas y la necesidad de soporte técnico. Las herramientas de código abierto, como PyMTL, FloPoCo y Bambu, ofrecen flexibilidad y la posibilidad de personalización, siendo ideales para entornos académicos y de investigación. Por otro lado, las herramientas comerciales proporcionan soluciones integradas y optimizadas, adecuadas para proyectos industriales con requerimientos estrictos de rendimiento y soporte.

En este trabajo, se explorará el uso de RustHDL, una biblioteca que permite la descripción de hardware utilizando el lenguaje Rust, evaluando su viabilidad y comparándola con las herramientas mencionadas anteriormente.

Capítulo 3

# RustHDL

## 3.1. Descripción, uso y ejemplo

RustHDL es una librería escrita en Rust que permite describir y simular circuitos digitales. Su objetivo principal es ofrecer una alternativa moderna a los lenguajes tradicionales de descripción hardware como Verilog o VHDL, utilizando las ventajas del lenguaje Rust: seguridad en memoria, manejo estricto de tipos y un ecosistema moderno de herramientas. RustHDL Contributors (2024a,b)

### 3.1.1. ¿Qué permite hacer RustHDL?

RustHDL permite:

- Describir circuitos digitales utilizando estructuras y tipos de datos de Rust.
- Generar automáticamente código Verilog a partir del diseño en Rust, para poder sintetizar el circuito con herramientas tradicionales (como Yosys o Vivado).
- Simular el comportamiento de los diseños directamente desde Rust, utilizando el sistema de pruebas del lenguaje.
- Integrar testbenches de forma exhaustiva y automatizada, algo especialmente útil para la verificación masiva de diseños.
- Ejecutar y controlar herramientas externas de síntesis y simulación, como Yosys e Icarus Verilog, directamente desde Rust, gracias a librerías específicas que facilitan la automatización del flujo de trabajo.

### 3.1.2. ¿Cómo probar los diseños?

Existen dos formas principales de probar los diseños desarrollados con RustHDL:

- **Generación y simulación en Verilog:** Se puede generar el código Verilog del diseño usando el comando:

  ```
  cargo run
  ```

  Esto produce un archivo Verilog (por ejemplo, `sumadorRust.v`) que se puede simular y sintetizar con herramientas externas como **Icarus Verilog** o **Yosys**.

  Para simular el diseño con un testbench en Verilog:

  ```
  iverilog -o test nombrefichero.v tv_nombrefichero.v
  vvp test
  ```

  Esto compila y ejecuta la simulación, mostrando la salida por consola.

- **Simulación y pruebas integradas en Rust:** RustHDL permite definir testbenches directamente en Rust, lo que facilita automatizar pruebas y verificar el comportamiento esperado sin salir del entorno Rust.

  Para ejecutar estas pruebas:

  ```
  cargo test -- --nocapture
  ```

  Este comando corre los tests definidos, mostrando la salida y permitiendo capturar trazas de depuración para análisis detallado.

En ambos casos, para visualizar el comportamiento temporal de las señales, se pueden generar archivos de traza en formato VCD (Value Change Dump) y abrirlos con **GTKWave**:

```
gtkwave nombrefichero.vcd
```

**Recomendaciones para el entorno de trabajo:**

- Instalar las herramientas necesarias: **Rust, RustHDL, Yosys, Icarus Verilog, GTKWave**.
- Usar un editor de código como **Visual Studio Code** para facilitar la lectura y edición.

- Mantener organizados los archivos Verilog y Rust en carpetas separadas para evitar confusiones.

De esta manera, se puede aprovechar tanto la flexibilidad y automatización de RustHDL como la robustez y amplio soporte de las herramientas tradicionales del ecosistema Verilog para el desarrollo, simulación y verificación de diseños digitales.

## 3.2. Diseño de Módulos en RustHDL

### 3.2.1. Módulos Combinacionales

#### 3.2.1.1. Definición y estructura

RustHDL permite definir módulos combinacionales de forma clara y concisa utilizando estructuras típicas del lenguaje Rust. Un módulo combinacional se caracteriza por no depender del reloj (`clk`) ni de estados internos, y su salida depende únicamente de sus entradas actuales.

#### 3.2.1.2. Ejemplo: sumador de 8 bits

A continuación, se muestra un ejemplo de un sumador de 8 bits con signo. Este módulo toma dos señales de entrada y produce una salida de 9 bits, correspondiente a la suma de ambas entradas.

```
use rust_hdl::prelude::*;

// Este módulo implementa un sumador de dos operandos con signo de 8 bits.
#[derive(LogicBlock, Clone)]
struct Sumador {
    // Señal de entrada 'a' con representación en complemento a dos de 8 bits.
    pub a: Signal<In, Signed<8>>,

    // Señal de entrada 'b' con representación en complemento a dos de 8 bits.
    pub b: Signal<In, Signed<8>>,

    // Señal de salida 'sum' con representación en complemento a dos de 9 bits.

    pub sum: Signal<Out, Signed<9>>,
}

```

```
// Implementación del trait `Default` para la inicialización por defecto del módulo.
impl Default for Sumador {
    fn default() -> Self {
        Self {
            a: Default::default(),
            b: Default::default(),
            sum: Default::default(),
        }
    }
}

// Se emplea la macro `#[hdl_gen]` para generar automáticamente el código HDL correspondiente.
impl Logic for Sumador {
    #[hdl_gen]
    fn update(&mut self) {
        // Realiza la conversión explícita de las señales de 8 bits a 9 bits
        // utilizando `signed_bit_cast`, asegurando la preservación del signo.
        // Posteriormente, se efectúa la suma.
        self.sum.next = signed_bit_cast::<9, 8>(self.a.val())
                      + signed_bit_cast::<9, 8>(self.b.val());
    }
}
```

Code 3.1: Definición del módulo Sumador en RustHDL.

**Explicación:** El módulo declara dos entradas `a` y `b`, ambas de tipo `Signed<8>`, y una salida `sum` de tipo `Signed<9>`. La operación se realiza en el método `update()` decorado con `#[hdl_gen]`, el cual será transformado en lógica Verilog combinacional.

#### 3.2.1.3. Pruebas funcionales del módulo combinacional

Para validar el comportamiento del sumador, se implementó un testbench en Rust e Icarus. Gracias a la integración con el sistema de pruebas del lenguaje, se pueden recorrer todas las combinaciones posibles de entrada en un rango dado.

- Se recorrieron todos los valores posibles en el rango `[-127, 127]`.
- Se verificó que la salida del sumador fuese la esperada en todos los casos.
- Las pruebas se ejecutan como parte del sistema de pruebas automatizadas de Rust.

A continuación, se muestra el código completo de la prueba funcional del módulo combinacional `Sumador`, utilizando el sistema de pruebas de Rust y la simulación

externa con Icarus Verilog. El testbench escrito en Verilog permite verificar rápidamente algunos casos representativos mediante simulación directa, mientras que el bloque principal del test, desarrollado en Rust, realiza una verificación exhaustiva.

El código Rust genera automáticamente el Verilog correspondiente al diseño, lo fusiona con el testbench, y lo ejecuta usando Icarus para observar su salida. Posteriormente, se lanza una simulación interna en RustHDL que recorre todos los pares de entrada en el rango `[-127, 127]`, comprobando que la salida sea exactamente la esperada. Esta combinación de pruebas rápidas e intensivas ofrece tanto eficiencia como cobertura completa.

```rust
#[test]
fn test_sumador() -> anyhow::Result<()> {
    // 1. Instanciación y conexión del módulo bajo prueba.
    let mut uut = Sumador::default();
    uut.connect_all();

    // 2. Generación del banco de pruebas en Verilog.
    // Este banco de pruebas estimula el módulo con varios valores de entrada.
    let verilog_tb = r#"
        module test;
          reg signed [7:0] a, b;
          wire signed [8:0] sum;

        // Instancia del módulo bajo prueba.
          Sumador uut(.a(a), .b(b), .sum(sum));

          initial begin

            // Monitor para visualizar valores durante la simulación.
            $monitor("a=%d, b=%d, sum=%d", a, b, sum);
            // Casos de prueba representativos.
            a = -128; b = -128; #10;
            a = -50;  b = 75;   #10;
            a = -50;  b = 50;   #10;
            a = 127;  b = 127;  #10;

            // Finalización de la simulación.
            #10
            $finish;
          end
        endmodule
    "#;

    // 3. Combinación del banco de pruebas y el código Verilog generado desde RustHDL.
    let tb = format!("{verilog_tb} {}", generate_verilog(&uut));

```

```
    // 4. Sustitución del nombre por defecto del módulo (top) por el nombre real del módulo.
    let code = tb.replace("module top(", "module Sumador(");

    // 5. Ejecución de la simulación mediante Icarus Verilog.
    let sim_output = get_icarus_verilog_output(&code)?;
    println!("(iverilog) Salida de Verilog:\n{}", sim_output);

    // 6. Definición de la simulación funcional en RustHDL.
    let mut sim = Simulation::<Sumador>::new();
    sim.add_testbench(move |mut ep| {
        // Inicialización del entorno de prueba.
        let mut x = ep.init()?;

        // 7. Generación de todos los casos de prueba posibles en el rango de -127 a 127.
        // Se realiza un recorrido exhaustivo del espacio de entrada (a, b).
        let test_cases: Vec<(i32, i32)> = (-127..128)
            .flat_map(|a| (-127..128).map(move |b| (a, b)))
            .collect();

        // 8. Aplicación de todos los casos de prueba y verificación de la salida.
        for &(a, b) in test_cases.iter() {
            x.a.next = Signed::<8>::from(a as i64);
            x.b.next = Signed::<8>::from(b as i64);

            let x_clone = x.clone();
            let x = ep.wait(1, x_clone)?;

            let expected_sum = a + b;

            // Comprobación de la suma esperada con la obtenida del módulo.
            sim_assert_eq!(
                ep,
                x.sum.val(),
                Signed::<9>::from(expected_sum as i64),
                x
            );
        }

        // 9. Finalización correcta de la simulación si todas las pruebas son satisfactorias.
        ep.done(x)?;
        Ok(())
    });

```

```
80      // 10. Eliminación de ficheros temporales generados por la
        simulación.
81      let _e = fs::remove_file("test_tb.vvp");
82      let _e = fs::remove_file("test_sumador.v");
83
84      // 11. Generación de un fichero VCD para visualización en
        GTKWave.
85      sim.run_to_file(Box::new(uut), 100000, "sumadorWave.vcd")
86          .map_err(|err| anyhow!("{:?}", err))?;
87
88      Ok(())
89  }
```

Code 3.2: Test en Rust e Icarus del Sumador.

### 3.2.2. Módulos Secuenciales

#### 3.2.2.1. Definición y estructura

A diferencia de los módulos combinacionales, los módulos secuenciales incluyen lógica de estado y dependen de una señal de reloj. Esto permite distribuir las operaciones a lo largo de varios ciclos, lo que habilita el uso de técnicas como el pipelining y el procesamiento segmentado. En RustHDL, estos módulos se construyen con registros tipo `DFF` y control explícito de señales como `clk` y `rstn`.

#### 3.2.2.2. Ejemplo: Reducción en Árbol con Pipeline

A continuación se muestra un ejemplo de un módulo secuencial que realiza la suma de 8 números con signo de 8 bits, utilizando una estructura en árbol de sumadores con segmentación por etapas (pipeline). Esto permite procesar datos de forma eficiente a lo largo de tres ciclos de reloj, donde cada etapa acumula y propaga los resultados de forma sincronizada.

```
1   use rust_hdl::prelude::*;
2
3   // Módulo que implementa una reducción en árbol con segmentaci
    ón en pipeline.
4   // Este bloque toma 8 entradas de 8 bits con signo y produce
    una salida de 11 bits con signo, correspondiente a la suma
    total de las entradas.
5   #[derive(LogicBlock, Clone)]
6   struct ArbolSumadoresSegmentacion {
7       // Entradas: arreglo de 8 señales de 8 bits con signo.
8       pub inputs: [Signal<In, Signed<8>>; 8],
9
10      // Salida: señal de salida de 11 bits con signo.
```

```
    pub result: Signal<Out, Signed<11>>,

    // Señal de reloj.
    pub clk: Signal<In, Clock>,

    // Señal de reset activo en bajo.
    pub rstn: Signal<In, Bit>,

    // Etapa 1: suma de pares de entradas.
    sum1: DFF<Signed<9>>,
    sum2: DFF<Signed<9>>,
    sum3: DFF<Signed<9>>,
    sum4: DFF<Signed<9>>,

    // Etapa 2: suma intermedia de resultados de etapa 1.
    sum1_1: DFF<Signed<10>>,
    sum1_2: DFF<Signed<10>>,

    // Etapa 3: suma final.
    final_sum: DFF<Signed<11>>,
}

// Inicialización por defecto del módulo.
impl Default for ArbolSumadoresSegmentacion {
    fn default() -> Self {
        Self {
            inputs: Default::default(),
            result: Default::default(),
            clk: Default::default(),
            rstn: Default::default(),
            sum1: DFF::default(),
            sum2: DFF::default(),
            sum3: DFF::default(),
            sum4: DFF::default(),
            sum1_1: DFF::default(),
            sum1_2: DFF::default(),
            final_sum: DFF::default(),
        }
    }
}

// Implementación de la lógica secuencial del bloque.
impl Logic for ArbolSumadoresSegmentacion {
    #[hdl_gen]
    fn update(&mut self) {
        // Asignación del reloj a todos los registros.
        self.sum1.clock.next = self.clk.val();
        self.sum2.clock.next = self.clk.val();
        self.sum3.clock.next = self.clk.val();
```

```
60          self.sum4.clock.next = self.clk.val();
61          self.sum1_1.clock.next = self.clk.val();
62          self.sum1_2.clock.next = self.clk.val();
63          self.final_sum.clock.next = self.clk.val();
64
65          // Lógica de reset sincrónico.
66          if !self.rstn.val() {
67              self.sum1.d.next = 0.into();
68              self.sum2.d.next = 0.into();
69              self.sum3.d.next = 0.into();
70              self.sum4.d.next = 0.into();
71              self.sum1_1.d.next = 0.into();
72              self.sum1_2.d.next = 0.into();
73              self.final_sum.d.next = 0.into();
74              self.result.next = 0.into();
75          } else {
76              // Etapa 1: se suman pares de entradas de 8 bits,
    generando resultados de 9 bits.
77              self.sum1.d.next = signed_bit_cast::<9,8>(self.
    inputs[0].val()) + signed_bit_cast::<9,8>(self.inputs[1].
    val());
78              self.sum2.d.next = signed_bit_cast::<9,8>(self.
    inputs[2].val()) + signed_bit_cast::<9,8>(self.inputs[3].
    val());
79              self.sum3.d.next = signed_bit_cast::<9,8>(self.
    inputs[4].val()) + signed_bit_cast::<9,8>(self.inputs[5].
    val());
80              self.sum4.d.next = signed_bit_cast::<9,8>(self.
    inputs[6].val()) + signed_bit_cast::<9,8>(self.inputs[7].
    val());
81
82              // Etapa 2: se suman los resultados anteriores en
    dos grupos, generando 10 bits.
83              self.sum1_1.d.next = signed_bit_cast::<10,9>(self.
    sum1.q.val()) + signed_bit_cast::<10,9>(self.sum2.q.val());
84              self.sum1_2.d.next = signed_bit_cast::<10,9>(self.
    sum3.q.val()) + signed_bit_cast::<10,9>(self.sum4.q.val());
85
86              // Etapa 3: se suman los dos resultados
    intermedios para producir la salida final.
87              self.final_sum.d.next = signed_bit_cast::<11,10>(
    self.sum1_1.q.val()) + signed_bit_cast::<11,10>(self.sum1_2
    .q.val());
88              self.result.next = self.final_sum.q.val();
89          }
90      }
91  }
```

Code 3.3: Reducción en árbol con pipeline.

**Explicación:** Este módulo implementa una estructura de reducción en árbol para sumar ocho operandos de 8 bits con soporte para segmentación y pipeline. La segmentación permite distribuir la operación de suma en etapas secuenciales, mientras que el uso de una arquitectura tipo árbol (en lugar de una suma en cadena) mejora la latencia y permite un diseño más escalable. Este enfoque es especialmente útil cuando se desea alta frecuencia de operación y procesamiento continuo de datos.

#### 3.2.2.3. Pruebas funcionales del módulo secuencial

Dado que la operación está dividida en múltiples etapas, se requieren múltiples ciclos de reloj para obtener el resultado final. Además, al emplear pipeline, los datos fluyen entre etapas internas, lo que implica que la sincronización temporal es crítica para la correcta operación del diseño.

Para verificar el correcto funcionamiento del módulo, se diseñaron pruebas funcionales tanto en Verilog como en RustHDL. Estas pruebas se enfocaron en evaluar:

- La **correctitud del resultado** en distintos ciclos de operación.
- La **respuesta del módulo frente a entradas extremas** (valores negativos, máximos, mínimos y cero).
- La **validez temporal del resultado** (es decir, cuándo debe estar disponible el resultado en función del retardo interno del pipeline).
- La **coherencia entre la implementación en Verilog y RustHDL**, facilitando la validación cruzada.

En Verilog, se emplearon testbenches clásicos, donde se controla manualmente el reloj y se verifican los resultados en distintos ciclos. Este enfoque es útil para observar el comportamiento interno paso a paso.

En RustHDL, el test se automatizó: se definieron múltiples casos de prueba con sus valores esperados, y el framework RustHDL se encargó de aplicar los datos, avanzar el reloj, y comprobar la salida. Esta automatización permitió lograr una cobertura más amplia en menor tiempo, facilitando además la detección de errores sutiles de sincronización.

El siguiente fragmento de código muestra un test completo que:

1. **Define y conecta el módulo**: Se instancia el módulo **ArbolSumadores** y se conectan automáticamente todas sus señales.
2. **Declara un banco de pruebas en Verilog**: Se inserta una cadena Verilog (`verilog_tb`) para ser compilada junto al diseño, útil para validación cruzada con Icarus Verilog.

3. **Ejecuta la simulación Verilog**: Compila y ejecuta el test en Icarus Verilog y muestra su salida.

4. **Configura un testbench en RustHDL**: Se definen varios casos de prueba en RustHDL, se simulan paso a paso con control del reloj y se verifican los resultados esperados.

5. **Genera un archivo de trazado VCD**: El resultado puede analizarse gráficamente con herramientas como GTKWave, permitiendo validar visualmente la alineación temporal entre entradas, propagación de datos intermedios y resultado final.

```
#[test]
fn test_reduccion_arbol() -> anyhow::Result<()> {
    // 1. Crear e inicializar el módulo bajo prueba.
    let mut uut = ArbolSumadoresSegmentacion::default();
    uut.connect_all();

    // 2. Banco de pruebas en Verilog: se define el entorno de
    simulación, incluyendo señales de entrada/salida, reloj,
   reset y estímulos.
    let verilog_tb = r#"
    module test;

    reg signed [7:0] in0, in1, in2, in3, in4, in5, in6, in7;
    wire signed [10:0] result;
    reg clk;
    reg rstn;

    ArbolSumadoresSegmentacion uut(
        .inputs$0(in0), .inputs$1(in1), .inputs$2(in2), .
   inputs$3(in3),
        .inputs$4(in4), .inputs$5(in5), .inputs$6(in6), .
   inputs$7(in7),
        .result(result), .clk(clk), .rstn(rstn)
    );

    // Generación del reloj: alterna cada 5 unidades de tiempo.
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    integer cycle;

    initial begin
        // Reset inicial y puesta a cero de las entradas.
        rstn = 0;
        {in0,in1,in2,in3,in4,in5,in6,in7} = 0;
```

```
34          #12;
35          rstn = 1;
36
37          $display("Empieza el test");
38          // 3. Se cargan distintos vectores de prueba cada 10 unidades de
    tiempo.
39          for (cycle = 0; cycle < 6; cycle = cycle + 1) begin
40              case (cycle)
41                  0: begin
42                      // Caso 0: mezcla de valores positivos y negativos.
43                      $display("Empieza a sumar el caso 0,
    resultado en 3 ciclos de 12");
44                      in0 = 0; in1 = -1; in2 = 2; in3 = 3;
45                      in4 = 4; in5 = 5; in6 = 6; in7 = -7;
46                  end
47                  1: begin
48                      // Caso 1: todos los valores mínimos representables
    (-128).
49                      $display("Empieza a sumar el caso 1,
    resultado en 3 ciclos de -1024");
50                      in0 = -128; in1 = -128; in2 = -128; in3 =
    -128;
51                      in4 = -128; in5 = -128; in6 = -128; in7 =
    -128;
52                  end
53                  2: begin
54                      // Caso 2: combinación de valores medios y altos
    positivos.
55                      $display("Empieza a sumar el caso 2,
    resultado en 3 ciclos de 125");
56                      in0 = 10; in1 = 15; in2 = 20; in3 = 5;
57                      in4 = 30; in5 = 25; in6 = 12; in7 = 8;
58                  end
59                  3: begin
60                      // Caso 3: valores máximos representables (127).
61                      $display("Empieza a sumar el caso 3,
    resultado en 3 ciclos de 1016");
62                      in0 = 127; in1 = 127; in2 = 127; in3 =
    127;
63                      in4 = 127; in5 = 127; in6 = 127; in7 =
    127;
64                  end
65                  default: begin
66                      // Caso por defecto: todos los valores en cero.
67                      $display("Caso por defecto, suma todo 0");
68                      {in0,in1,in2,in3,in4,in5,in6,in7} = 0;
69                  end
70              endcase
71              #10;
72              $display("Cycle %0d: Result: %0d", cycle, result);
```

```
73          end
74
75
76          $finish;
77      end
78      endmodule
79      "#;
80
81      // 4. Unir el banco de pruebas con el diseño generado en
    Verilog.
82      let tb = format!("{verilog_tb} {}", generate_verilog(&uut)
    );
83      let code = tb.replace("module top(", "module
    ArbolSumadoresSegmentacion(");
84
85      // 5. Ejecutar la simulación con Icarus Verilog y mostrar
    la salida.
86      let sim_output = get_icarus_verilog_output(&code)?;
87      println!("Salida Verilog:\n{}", sim_output);
88
89      // 6. Configurar la simulación en RustHDL.
90      let mut sim = Simulation::<ArbolSumadoresSegmentacion>::
    new();
91      sim.add_testbench(move |mut ep| {
92          let mut x = ep.init()?;
93
94          // Definición de casos de prueba: entradas, resultado
    esperado y nombre.
95          let test_cases: Vec<([i8; 8], i16, &'static str)> =
    vec![
96              ([0, -1, 2, 3, 4, 5, 6, -7], 12, "caso 0"),
97              ([-127; 8], -1016, "caso 1"),
98              ([10, 15, 20, 5, 30, 25, 12, 8], 125, "caso 2"),
99              ([127; 8], 1016, "caso 3"),
100             ([0; 8], 0, "default"),
101         ];
102
103         println!("Empieza el test");
104
105         // Secuencia de reset sincrónico.
106         x.rstn.next = false;
107         x.clk.next = Clock { clk: false };
108         x = ep.wait(1, x.clone())?;
109         x.clk.next = Clock { clk: true };
110         x = ep.wait(1, x.clone())?;
111
112         x.rstn.next = true;
113         x.clk.next = Clock { clk: false };
114         x = ep.wait(1, x.clone())?;
```

```
115         x.clk.next = Clock { clk: true };
116         x = ep.wait(1, x.clone())?;
117
118         // 7. Ejecutar cada uno de los casos de prueba.
119         for (cycle, (inputs, expected, case_name)) in
    test_cases.iter().enumerate() {
120             println!(
121                 "Ciclo {}: ejecutando {}, resultado esperado =
     {}",
122                 cycle, case_name, expected
123             );
124
125             for i in 0..8 {
126                 x.inputs[i].next = Signed::<8>::from(inputs[i]
     as i64);
127             }
128
129             // Ejecutar tres ciclos de reloj para propagar la
    señal a través del pipeline.
130             for _ in 0..3 {
131                 x.clk.next = Clock { clk: false };
132                 x = ep.wait(1, x.clone())?;
133                 x.clk.next = Clock { clk: true };
134                 x = ep.wait(1, x.clone())?;
135             }
136
137             println!(
138                 "Resultado obtenido: {}",
139                 x.result.val().bigint()
140             );
141         }
142
143         ep.done(x)?;
144         Ok(())
145     });
146
147     // 8. Eliminar archivos temporales generados por la
    simulación.
148     let _ = fs::remove_file("test_tb.vvp");
149     let _ = fs::remove_file("test_ArbolSumadoresSegmentacion.v
    ");
150
151     // 9. Guardar señales para su posterior visualización con
    GTKWave.
152     sim.run_to_file(Box::new(uut), 100_000, "
    ArbolSumadoresSegmentacionWave.vcd")
153         .map_err(|e| anyhow!("{:?}", e))?;
154
155     Ok(())
```

```
156 }
```

Code 3.4: Test en Rust e Icarus del sumador en Árbol con pipeline.

#### 3.2.2.4. Ventajas del enfoque secuencial

El diseño secuencial, aunque introduce mayor latencia, permite distribuir la carga computacional, usar menos recursos combinacionales, y facilita la escalabilidad. Además, se adapta mejor a diseños complejos con múltiples etapas de control.

## 3.3. Datos de simulación

Los resultados obtenidos de las simulaciones y las síntesis pueden observarse en los siguientes directorios dentro de la carpeta compartida que se facilita con la entrega del TFG:

- **Capturas de GTKWave**: Permiten analizar la evolución temporal de las señales y verificar su correcta propagación.
- **Capturas de Síntesis con Yosys**: Comparan la cantidad de recursos utilizados en los diseños generados por RustHDL frente a los escritos manualmente en Verilog.

Los archivos `.vcd` generados durante las simulaciones están disponibles dentro de los proyectos correspondientes (tanto de RustHDL como de Verilog). Las capturas de pantalla de las simulaciones y de las síntesis se encuentran organizadas en una carpeta llamada **Sintesis**, dentro del repositorio compartido en el siguiente enlace: RustHDL

Capítulo 4

# Casos de Uso

A continuación, se presentan los casos de uso correspondientes a los diferentes módulos implementados, destacando las comparaciones entre el código Verilog escrito a mano y el código generado a partir de RustHDL. Se incluyen las pruebas realizadas, los resultados obtenidos y las ventajas y desventajas observadas en cada enfoque.

## 4.1. Caso 1: Sumador binario con signo

### 4.1.1. Descripción del módulo

El primer caso de estudio implementado fue un **Sumador con Signo**, cuya función es sumar dos números de 8 bits con signo y devolver un resultado de 9 bits también con signo, para abarcar el posible desbordamiento. Este ejemplo sirvió como punto de partida para familiarizarnos con la estructura de módulos en RustHDL, entender sus limitaciones en la generación de hardware, y establecer el flujo de trabajo de generación de código Verilog, simulación y síntesis.

### 4.1.2. Metodología de verificación

La verificación del módulo se realizó en dos etapas complementarias:

- **Testbench en Icarus Verilog**: Se utilizó un conjunto de pruebas con valores representativos (positivos, negativos, extremos) para validar rápidamente que el código Verilog generado funcionaba correctamente.
- **Simulación en RustHDL**: Se implementó una simulación exhaustiva, generando pruebas para todos los posibles pares de entrada en el rango de

$[-128, 127]$. Esta metodología aprovecha una de las principales ventajas de RustHDL: poder ejecutar pruebas de alto nivel de forma automatizada y detallada dentro del propio entorno de desarrollo de Rust.

### 4.1.3. Comparación de resultados

#### 4.1.3.1. Correcto Funcionamiento

Ambos diseños generaron los mismos resultados para todos los casos probados. La única excepción fue un pequeño fallo observado en la simulación de RustHDL con el valor -128, que no se llegó a ejecutar correctamente por una limitación interna. Este caso sí fue cubierto correctamente en Icarus Verilog, lo que permitió confirmar la funcionalidad completa del diseño. A pesar de este detalle, la simulación exhaustiva en RustHDL cubrió la totalidad del espacio restante, validando su utilidad como entorno de pruebas automatizado.

```
(iverilog) Salida de Verilog:
a=-128, b=-128, sum=-256
a= -50, b=  75, sum=  25
a= -50, b=  50, sum=   0
a= 127, b= 127, sum= 254
test_sumador.v:15: $finish called at 50 (1s)
```

Figura 4.1: Ejemplo de Icarus.

```
thread 'test_sumador' has overflowed its stack
fatal runtime error: stack overflow
error: test failed, to rerun pass `--bin sumador`
```

Figura 4.2: Error de Test al intentar ejecutar el -128 en Rust.

#### 4.1.3.2. Síntesis lógica

Los resultados de síntesis obtenidos con Yosys para ambos códigos Verilog, el generado automáticamente desde Rust y el escrito a mano, fueron idénticos. Al tratarse de un módulo simple, sin optimizaciones adicionales, ambos diseños ocuparon los mismos recursos lógicos y presentaron estructuras de puertas lógicas equivalentes.

```
=== Sumador ===

   Number of wires:                 44
   Number of wire bits:             66
   Number of public wires:           3
   Number of public wire bits:      25
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 50
     $_ANDNOT_                      14
     $_AND_                          1
     $_NAND_                         7
     $_NOR_                          6
     $_ORNOT_                        1
     $_OR_                           5
     $_XNOR_                         8
     $_XOR_                          8

2.26. Executing CHECK pass (checking for obvious problems).
Checking module Sumador...
```

Figura 4.3: Síntesis de Código generado por Rust.

```
=== sumador ===

   Number of wires:                 44
   Number of wire bits:             66
   Number of public wires:           3
   Number of public wire bits:      25
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 50
     $_ANDNOT_                      14
     $_AND_                          1
     $_NAND_                         7
     $_NOR_                          6
     $_ORNOT_                        1
     $_OR_                           5
     $_XNOR_                         8
     $_XOR_                          8

2.26. Executing CHECK pass (checking for obvious problems).
Checking module sumador...
```

Figura 4.4: Síntesis de Código Verilog generado a mano.

#### 4.1.3.3. Visualización de señales

El análisis de las ondas generadas (`.vcd`) en las simulaciones y visualizadas con GTKWave demostró un comportamiento correcto de los dos diseños, sin diferencias en latencia, errores de lógica ni inconsistencias en la propagación de señales. Esto refuerza la validez del código generado por RustHDL para casos simples como este.

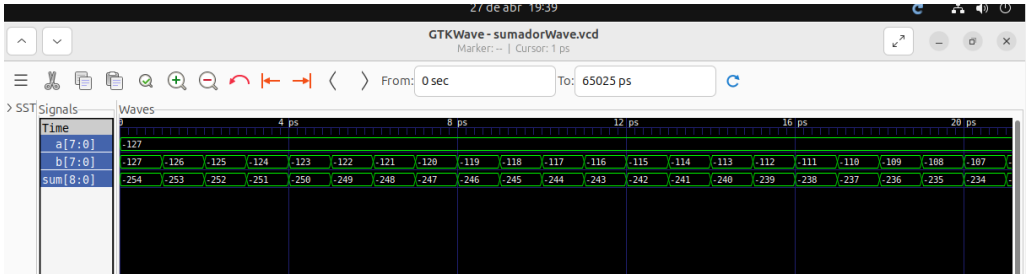


Figura 4.5: Forma de Onda del código generado en Rust.

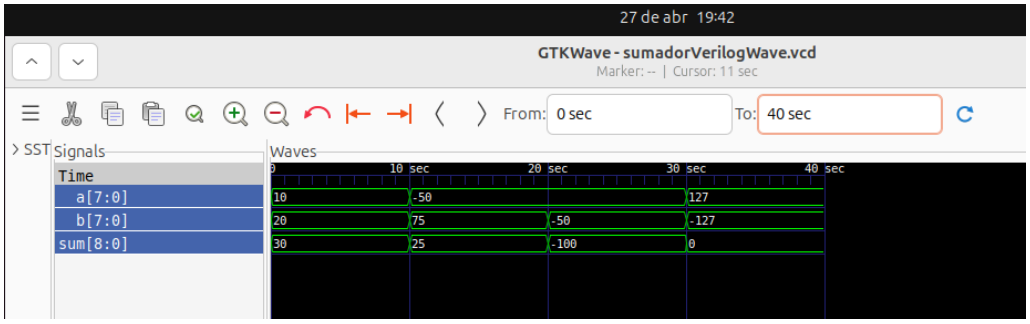


Figura 4.6: Forma de Onda del código Verilog generado a mano.

En el caso del sumador, destacamos la utilidad de realizar la simulación en Rust, ya que, si bien tanto en Verilog como en VHDL es posible implementar bucles dentro de los testbenches para probar exhaustivamente todas las combinaciones de entrada, Rust facilita esta tarea desde el punto de vista del desarrollo de pruebas. Además, el uso de Rust puede implicar menores requerimientos de herramientas, ya que la simulación de diseños RTL complejos o de larga duración en Verilog suele requerir herramientas especializadas, a menudo de pago, como Modelsim.

### 4.1.4. Observaciones particulares del caso

Este primer módulo permitió detectar algunos aspectos relevantes de RustHDL, como la necesidad de realizar conversiones explícitas entre anchos de bits y el uso de tipos firmados. También se observó que la herramienta de simulación propia de RustHDL presenta ciertas limitaciones al manejar valores extremos, como por ejemplo el -128 en representaciones de 8 bits. Aun así, su capacidad para realizar pruebas masivas resulta muy útil. En general, este caso sirvió para establecer una metodología de trabajo repetible y confiable, aplicable al análisis de módulos más complejos.

## 4.2. Caso 2: Reconocedor de Patrones Binarios

### 4.2.1. Descripción del módulo

El segundo caso de estudio implementado fue un **Reconocedor de Patrones Binarios** diseñado para detectar un patrón simple de cuatro bits: "1011". El módulo toma como entrada un bit a la vez, junto con señales de reloj y reset. Su salida es un valor binario que indica si el patrón ha sido detectado. Si el patrón "1011"se encuentra, la salida es 1; en caso contrario, es 0.

Este módulo sirvió como una prueba para evaluar la capacidad de RustHDL de manejar máquinas de estados finitas (FSM). Durante la implementación, nos enfrentamos a varios desafíos relacionados con la sincronización de la máquina de estados, lo que nos permitió conocer mejor las capacidades y limitaciones de RustHDL en comparación con los enfoques más tradicionales.

### 4.2.2. Metodología de verificación

La verificación del módulo se llevó a cabo mediante dos enfoques complementarios:

- **Testbench en Icarus Verilog**: Se realizaron tres pruebas clave que abordan distintas situaciones:
  - **Primera prueba**: Detección del patrón completo "1011".
    Esta prueba simula la entrada de los bits que componen el patrón **1011**. El objetivo de esta prueba es verificar que el módulo detecte correctamente el patrón completo en el orden adecuado. Al ingresar estos cuatro bits secuenciales, la salida debería ser 1, ya que se ha detectado el patrón esperado.
  - **Segunda prueba**: Patrón parcial "101".
    En esta prueba, se ingresan tres bits **101**. Esta secuencia no corresponde al patrón completo **1011**, pero sirve para comprobar que el módulo no produzca una salida errónea (es decir, no debería detectar el patrón). En este caso, la salida debería ser 0, ya que el patrón no está completo.
  - **Tercera prueba**: Superposición de patrones y secuencias no ideales.
    En esta prueba se introduce una secuencia más larga que incluye tanto el patrón **1011** como otros bits adicionales antes, después o intercalados, simulando así una entrada más compleja y menos estructurada. Aunque se siguen considerando condiciones ideales, esta prueba permite comprobar si el diseño es capaz de detectar correctamente el patrón objetivo incluso cuando aparece solapado o en medio de datos adicionales. Al final de la secuencia, el patrón debe haber sido detectado y la salida debe ser 1.

- **Simulación en RustHDL**: Al igual que con el testbench en Icarus, se implementaron las mismas tres pruebas en RustHDL. La ventaja de RustHDL en este caso fue la capacidad de realizar una verificación automática más amplia y detallada, abarcando todos los casos posibles dentro del rango de entradas.

### 4.2.3. Comparación de resultados

#### 4.2.3.1. Correcto Funcionamiento

En un principio, la simulación en RustHDL presentó dificultades debido a un problema con la implementación secuencial de la máquina de estados. Específicamente, el diseño requería un registro para almacenar el valor de la entrada durante más de un ciclo de reloj, lo que inicialmente no era posible con el flip-flop D (DFF) genérico utilizado. Para solucionar esto, se recurrió a la clase `EdgeDFF`, que permite la actualización del valor en el flanco del reloj, resolviendo el problema de sincronización y permitiendo que la máquina de estados funcionara.

Una vez corregido este problema, las pruebas en Icarus confirmaron que el módulo funcionaba correctamente. Sin embargo, al realizar la simulación en RustHDL la máquina de estados no actualizaba el estado adecuadamente. Después de investigar, se descubrió que el método `ep.wait()` utilizado en la simulación no realiza las transiciones de reloj necesarias. Este método, aunque adecuado para ciertas simulaciones, no simula la subida o bajada del reloj, lo que causaba el fallo en la actualización del estado.

Este inconveniente se solucionó realizando manualmente las transiciones de reloj dentro de la simulación, lo que permitió que el reconocedor de patrones funcionara correctamente en todos los casos, incluyendo aquellos con superposición de datos.

#### 4.2.3.2. Síntesis lógica

Los resultados obtenidos con Yosys para la síntesis lógica mostraron diferencias en el número de componentes utilizados entre los dos enfoques. El código Verilog generado automáticamente desde RustHDL produjo un diseño con un mayor número de celdas (45 frente a 31 en el diseño manual), lo que representa un incremento aproximado del 50 % en cuanto a área utilizada.

En resumen, aunque el diseño manual resultó en una implementación más compacta en términos de recursos lógicos, el enfoque automático de RustHDL sigue siendo una alternativa válida, especialmente por la automatización y rapidez que aporta al proceso de desarrollo. La diferencia observada en el área utilizada puede deberse tanto a las optimizaciones propias del generador de código como a la forma en que se expresa la lógica en Rust. Esto sugiere que quizás sea necesario explorar otros estilos o patrones de codificación más adecuados para RustHDL, algo

comprensible dado el estado aún incipiente de la herramienta.

```
=== reconocedorPatrones ===

   Number of wires:                 47
   Number of wire bits:             51
   Number of public wires:           7
   Number of public wire bits:      11
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 45
     $_ANDNOT_                      17
     $_AND_                          1
     $_MUX_                          3
     $_NAND_                         4
     $_NOR_                          2
     $_NOT_                          2
     $_ORNOT_                        9
     $_OR_                           6
     top$state                       1

=== top$state ===

   Number of wires:                  3
   Number of wire bits:              7
   Number of public wires:           3
   Number of public wire bits:       7
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                  3
     $_DFF_P_                        3
```

Figura 4.7: Síntesis de Código generado por Rust.

```
=== reconocedor_patrones ===

   Number of wires:                 30
   Number of wire bits:             34
   Number of public wires:           6
   Number of public wire bits:      10
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 31
     $_ANDNOT_                       9
     $_AND_                          2
     $_DFF_PP0_                      3
     $_MUX_                          1
     $_NAND_                         1
     $_NOT_                          2
     $_ORNOT_                        4
     $_OR_                           9
```

Figura 4.8: Síntesis de Código Verilog generado a mano.

#### 4.2.3.3. Visualización de señales

Se mostró un comportamiento correcto en ambos diseños. Las señales fueron consistentes en términos de latencia y propagación, y no se detectaron inconsistencias en la lógica del módulo. Dado que ambos enfoques (el generado por RustHDL y el escrito manualmente en Verilog) pasaron los mismos casos de prueba, las ondas generadas fueron idénticas en ambos casos. Esto refuerza la validez de que los dos diseños cumplen con los requisitos funcionales de manera similar.

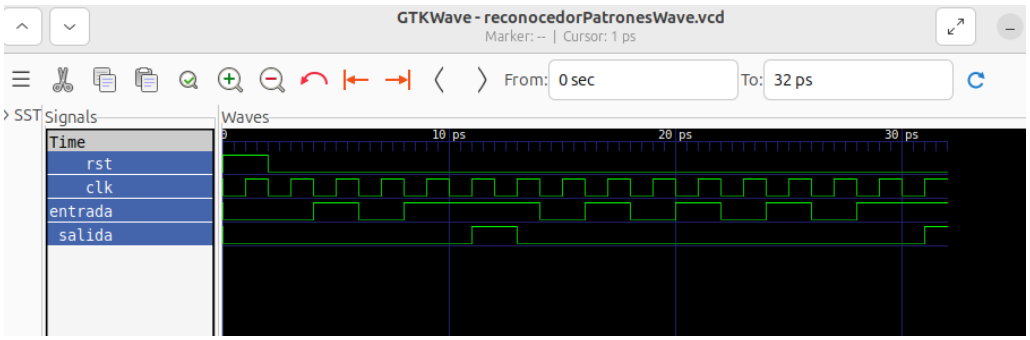


Figura 4.9: Forma de Onda del código generado en Rust.

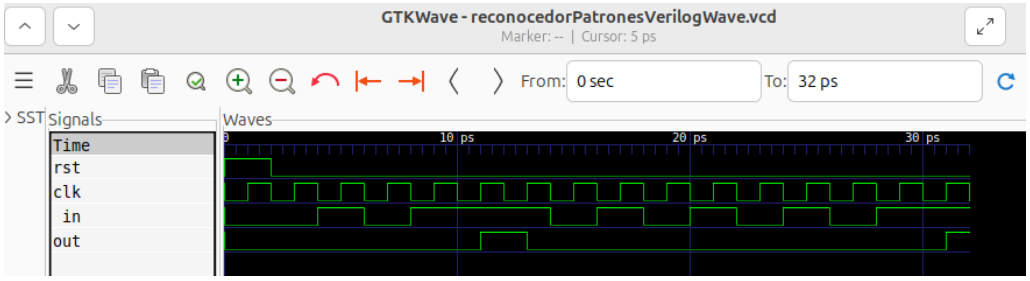


Figura 4.10: Forma de Onda del código Verilog generado a mano.

### 4.2.4. Observaciones particulares del caso

Este caso de uso evidenció algunos puntos clave en la implementación de máquinas de estados finitas (FSM) con RustHDL. En particular, la principal dificultad estuvo en la implementación secuencial de la máquina de estados, que inicialmente no funcionaba correctamente debido al uso de un flip-flop genérico. La solución fue utilizar la clase `EdgeDFF`, que maneja la sincronización con el flanco del reloj, lo que permitió que la máquina de estados funcionara de acuerdo a lo esperado.

Además, se identificó un detalle importante en la simulación: el método `ep.wait()` utilizado para la espera entre ciclos no realiza las transiciones de reloj necesarias, lo que provocó que el estado no se actualizara. Este problema se solucionó realizando manualmente las transiciones de reloj en la simulación.

En general, este caso sirvió para explorar la capacidad de RustHDL para manejar diseños secuenciales y máquinas de estados, y demostró cómo se pueden superar algunas limitaciones con un poco de esfuerzo y comprensión profunda del entorno de simulación.

## 4.3. Caso 3: Sumador segmentado en árbol

### 4.3.1. Descripción del módulo

El tercer caso de estudio se centró en la implementación de un **Sumador Segmentado con Estructura de Árbol**. Este módulo recibe como entrada un vector de ocho elementos de tipo `signed` de 8 bits, y genera como salida un valor `signed` de 11 bits que representa la suma total de esos elementos. Además, el módulo cuenta con una señal de reloj y un reset activo a baja (lógica inversa).

El objetivo principal de este diseño fue explorar el uso de técnicas de paralelismo mediante segmentación (pipeline) y una arquitectura de suma en árbol, con el fin de mejorar la eficiencia temporal. La estructura en árbol reduce la cantidad de operaciones secuenciales necesarias en la suma, lo que facilita dividir el proceso en varias etapas sincronizadas mediante registros (DFFs), implementando así un diseño

secuencial con pipeline.

A pesar de que podría suponerse que un diseño segmentado como este presentaría complicaciones debido al uso de pipeline, se comprobó que utilizando correctamente registros (DFFs) entre etapas y aplicando una segmentación cuidadosa, se obtiene un diseño funcional. La principal desventaja de este enfoque es el incremento en el uso de componentes lógicos durante la síntesis, como consecuencia del uso de biestables.

Desde el punto de vista de la implementación, este caso fue relativamente sencillo. No fue necesario comprobar extremos del rango de los valores (máximos y mínimos), ya que el principal objetivo era verificar que el paralelismo se aplicaba correctamente y que la salida se producía con la latencia esperada. En nuestro diseño, el resultado se obtiene tres ciclos de reloj después de aplicar las entradas.

### 4.3.2. Metodología de verificación

La verificación se realizó a través de una batería de pruebas centradas en comprobar la validez de los resultados. Las simulaciones se llevaron a cabo en Icarus Verilog y RustHDL, utilizando diferentes conjuntos de datos que permitieran verificar que la suma se completaba correctamente tras tres ciclos de reloj. No se realizó una verificación exhaustiva, ya que el comportamiento funcional del sumador sin segmentar ya había sido probado en una etapa anterior del desarrollo. Dado que el sumador segmentado reutiliza la misma lógica de suma, el foco se trasladó a comprobar casos límite relevantes para asegurar la correcta propagación de datos a través del pipeline.

Los casos de prueba empleados fueron los siguientes:

- **Caso 1: Suma mixta con resultado positivo (12)** Se introdujeron los valores {0, -1, 2, 3, 4, 5, 6, -7}. Este caso permite verificar que el módulo es capaz de manejar tanto valores positivos como negativos en una misma operación, y que el resultado es correcto tras tres ciclos de latencia.
- **Caso 2: Suma de valores mínimos (-1024)** Se probaron ocho valores de -128, que es el mínimo valor representable en 8 bits con signo. Este caso comprueba que el módulo maneja correctamente los límites inferiores del rango y que no se produce desbordamiento incorrecto.
- **Caso 3: Suma con valores positivos (125)** Se usaron entradas variadas como {10, 15, 20, 5, 30, 25, 12, 8}. Este caso simula una suma con datos de uso general para validar el comportamiento básico del módulo.
- **Caso 4: Suma de valores máximos (1016)** Se introdujeron ocho veces el valor 127, que es el valor máximo representable en 8 bits con signo. Este caso comprueba que el módulo soporta la suma acumulativa de los máximos valores sin error.

- **Caso por defecto: Todos los valores en 0 (resultado 0)** Se verificó el caso en que todos los valores de entrada son cero, lo que debe producir una salida de cero. Este test ayuda a descartar falsos positivos en la salida por errores de inicialización o sincronización.

Todos estos casos fueron evaluados tanto en el entorno de pruebas de Verilog como en RustHDL, lo que permitió verificar la consistencia funcional entre ambas implementaciones. Se detectó una pequeña variación en las formas de onda: en la simulación generada por RustHDL se prueba el valor -127 en lugar de -128. Esta diferencia se debe a una limitación comentada previamente, y puede explicar pequeñas discrepancias puntuales en los resultados de algunos test.

### 4.3.3. Comparación de resultados

#### 4.3.3.1. Correcto Funcionamiento

El módulo funcionó correctamente en todos los casos de prueba, entregando los resultados esperados con una latencia de tres ciclos de reloj. Esta latencia era coherente con el número de etapas de segmentación introducidas en el diseño debido a la estructura en árbol y el uso de registros entre etapas.

El uso del reset activo a baja funcionó adecuadamente. Al iniciar la simulación con el reset activo, se observaron salidas en cero, y al desactivarlo, el sistema comenzó a procesar correctamente las entradas desde el primer ciclo válido.

La implementación de la lógica de segmentación con registros (DFFs) permitió aislar correctamente cada etapa del cálculo, garantizando estabilidad en las señales y facilitando la verificación del paralelismo. En ningún momento se detectaron errores de sincronización ni inconsistencias en las salidas.

#### 4.3.3.2. Síntesis lógica

En cuanto a la síntesis lógica realizada con Yosys, se observaron diferencias claras entre esta implementación segmentada y otras más simples. Como era de esperarse, el uso de registros entre etapas aumentó el número total de componentes en el circuito, especialmente en términos de flip-flops y cables intermedios.

Este aumento de recursos es coherente con el objetivo del diseño: permitir un mayor rendimiento temporal mediante la segmentación del proceso, es decir, realizar mas operaciones en menos tiempo. La relación entre el coste en hardware y la mejora en el tiempo de operación justifica el uso adicional de componentes. La estructura final sintetizada evidenció una organización clara de las etapas del sumador en árbol y su segmentación interna.

No obstante, cabe señalar que una implementación manual más compacta puede requerir menos componentes lógicos totales, pues Rust tiende a crear subcomponentes como módulos intermedios o registros auxiliares, lo que incrementa el número final de celdas tras la síntesis.

```
=== ArbolSumadoresSegmentacion (Agrupado) ===

Number of wires:        449
Number of wire bits:    755
Number of public wires: 53
Number of public wire bits: 359
Number of memories:     0
Number of memory bits:  0
Number of processes:    0
Number of cells:        514
  $_ANDNOT_   215
  $_AND_      18
  $_DFF_P_    67
  $_NAND_     53
  $_NOR_      16
  $_NOT_      23
  $_ORNOT_    21
  $_OR_       8
  $_XNOR_     3
  $_XOR_      90
```

Figura 4.11: Síntesis de Código generado por Rust.

```
=== ReduccionEnArbol ===

   Number of wires:                341
   Number of wire bits:            647
   Number of public wires:          18
   Number of public wire bits:     144
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                447
     $_ANDNOT_                      90
     $_AND_                         14
     $_DFF_PN0_                     78
     $_NAND_                        46
     $_NOR_                         41
     $_ORNOT_                       10
     $_OR_                          55
     $_XNOR_                        60
     $_XOR_                         53
```

Figura 4.12: Síntesis de Código Verilog generado a mano.

#### 4.3.3.3. Visualización de señales

Las señales generadas durante la simulación en ambos entornos fueron visualizadas en GTKWave. Las ondas generadas para los mismos casos de prueba fueron idénticas, lo que indica que tanto el diseño en RustHDL como el escrito manualmente en Verilog producen un comportamiento funcionalmente equivalente.

En todas las pruebas, se observó que la salida del sumador aparecía exactamente tres ciclos de reloj después de aplicar las entradas, confirmando que la lógica segmentada se comporta correctamente.

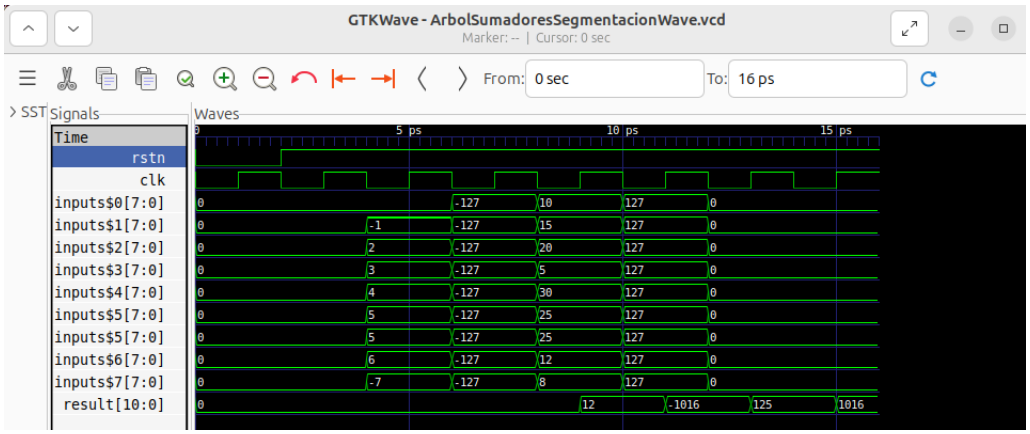


Figura 4.13: Forma de Onda del código generado en Rust.

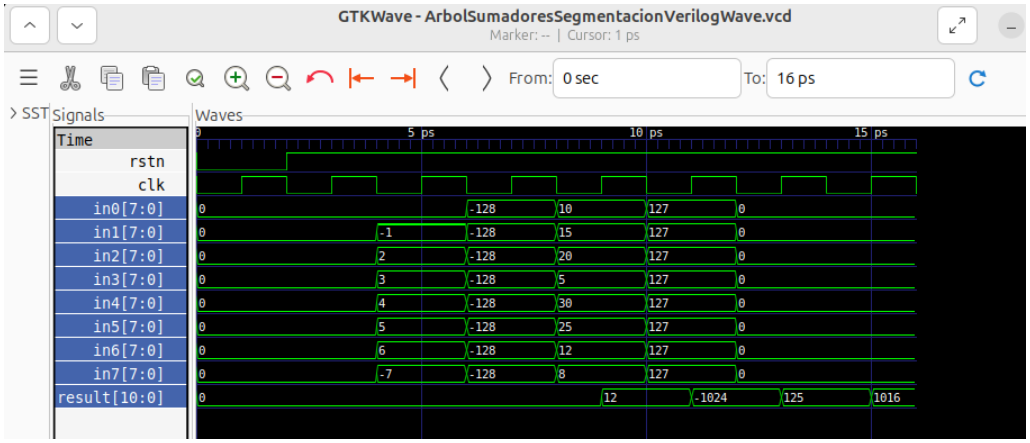


Figura 4.14: Forma de Onda del código Verilog generado a mano.

### 4.3.4. Observaciones particulares del caso

Este caso destacó por su claridad estructural y por el hecho de que, a pesar de aplicar técnicas como la segmentación y el uso de una arquitectura en árbol, el diseño fue relativamente sencillo de implementar. La clave del éxito fue el uso correcto de registros `DFF` entre etapas, lo que permitió mantener la sincronización adecuada entre ciclos y aprovechar el paralelismo ofrecido por la arquitectura.

El unico impacto negativo fue el aumento en los recursos utilizados durante la síntesis lógica, consecuencia directa del uso de biestables. Este incremento es especialmente notable al comparar con el diseño escrito manualmente en Verilog, que resulta más eficiente en términos de utilización de componentes. Aun así, este coste adicional es aceptable dada la mejora en el tiempo de procesamiento y la facilidad de segmentación lograda con el diseño en RustHDL.

Por otro lado, durante la simulación en RustHDL se detectó el mismo problema observado anteriormente: ciertas limitaciones en el manejo de valores extremos en números signed, en particular el valor -128, que el sistema no es capaz de representar o gestionar correctamente en los tests actuales. Esta limitación se aborda en las pruebas de Rust reemplazando dicho valor por -127.

En general, el caso del sumador segmentado demuestra que RustHDL permite la implementación de arquitecturas más avanzadas con segmentación temporal, siempre que se maneje cuidadosamente la lógica secuencial y la sincronización de las señales. Aun con pequeñas limitaciones en la simulación y un mayor uso de recursos lógicos, el diseño mantiene una funcionalidad correcta y predecible.

## 4.4. Caso 4: Búsqueda del máximo segmentada

### 4.4.1. Descripción del módulo

El cuarto caso de estudio se centró en la implementación de un módulo para la **Búsqueda del Valor Máximo en un Vector** de ocho elementos, utilizando una arquitectura segmentada con estructura de árbol. Este diseño fue desarrollado de forma paralela al **Sumador Segmentado**, ya que comparten principios fundamentales como la segmentación temporal y el paralelismo estructural.

El módulo recibe como entrada un vector de ocho elementos de tipo `signed` de 8 bits y entrega como salida un único valor `signed` de 8 bits, correspondiente al mayor de los elementos. La arquitectura implementada sigue un esquema jerárquico de comparadores dispuestos en árbol, permitiendo comparar pares de elementos en diferentes niveles. Cada etapa está separada mediante registros (`DFFs`), lo que garantiza la sincronización entre ciclos de reloj y permite implementar una segmentación efectiva del proceso.

Una particularidad relevante de este diseño fue la inclusión de señales de control `valid_in` y `valid_out`, que permiten verificar y propagar la validez de los datos a través de las etapas del pipeline. En este contexto, un dato válido es aquel que contiene información útil que debe ser procesada, mientras que un dato inválido representa una entrada no significativa o no disponible en ese momento. Estas señales son comunes en arquitecturas con pipelines o interfaces de comunicación como AXI o Avalon, donde es necesario coordinar el flujo de datos entre módulos que operan a diferentes ritmos.

El uso de `valid_in` y `valid_out` permite simular un entorno de procesamiento más realista, en el que los datos no se procesan continuamente, sino solo cuando están disponibles y correctamente alineados en el tiempo. Además, este mecanismo mejora la robustez del diseño, al evitar que se propaguen resultados incorrectos en caso de datos no preparados, y facilita la integración con sistemas más complejos que requieren control explícito del flujo de información.

Cabe destacar que, a diferencia del caso anterior, esta implementación no requiere operaciones de ampliación de bits ni transformaciones de tipo, lo cual simplificó su desarrollo.

### 4.4.2. Metodología de verificación

La verificación del módulo se llevó a cabo mediante simulaciones funcionales utilizando el entorno de Icarus Verilog. El enfoque consistió en evaluar el correcto funcionamiento del pipeline en términos de detección del valor máximo, sincronización de la salida y manejo adecuado de señales de control como el `reset`, `valid_in` y `valid_out`.

Se emplearon múltiples vectores de entrada con combinaciones diversas de valores negativos, positivos y extremos del rango de representación. Los objetivos fueron confirmar que el módulo identifica correctamente el mayor valor en cualquier configuración de entradas y verificar que la salida se produce con la latencia correspondiente al número de etapas en la arquitectura en árbol.

Durante las pruebas se confirmó que el módulo procesaba correctamente cada conjunto de datos tras la desactivación del reset, y que respetaba la latencia esperada sin perder sincronización entre las señales válidas y los datos asociados. Además, se comprobó que el sistema manejaba sin problemas los valores límite, como `-128` y `127`, sin errores de comparación.

### 4.4.3. Comparación de resultados

#### 4.4.3.1. Correcto funcionamiento

El módulo demostró un comportamiento correcto en todas las pruebas. La lógica de comparación funcionó de forma precisa en cada etapa del árbol, propagando correctamente el valor máximo hasta la salida. Las señales `valid_in` y `valid_out` funcionaron de manera correcta, permitiendo un control efectivo sobre el flujo de datos válidos.

La salida se presentó tras la latencia esperada de tres ciclos de reloj, esto concuerda con el número de niveles definidos en la arquitectura segmentada.

#### 4.4.3.2. Síntesis lógica

En cuanto a la síntesis lógica realizada mediante Yosys, se observó un patrón similar al del caso anterior: la segmentación con registros intermedios genera un aumento en el número de componentes lógicos utilizados. Este incremento se debe al uso de múltiples comparadores y biestables en cada etapa del árbol.

Este caso resulta especialmente ilustrativo, ya que, a pesar de tratarse de una funcionalidad relativamente simple —la búsqueda del valor máximo en un vector—, el uso intensivo de registros (`DFFs`) en cada etapa del diseño segmentado conlleva un aumento significativo en la cantidad de recursos empleados. Esto pone de manifiesto cómo incluso módulos conceptualmente sencillos pueden traducirse en estructuras complejas a nivel de síntesis cuando se emplean técnicas de segmentación temporal.

No obstante, este mayor uso de recursos se acompaña de una organización más estructurada del diseño, facilitando la segmentación. Aun así, el diseño en RustHDL se mantiene como una alternativa válida y robusta, especialmente por su claridad estructural y por permitir una implementación más rápida y menos propensa a errores de arquitecturas segmentadas.

```
=== maximoVectorSegmentacion (Agrupado) ===

Number of wires:        425
Number of wire bits:    684
Number of public wires: 73
Number of public wire bits: 332
Number of memories:     0
Number of memory bits:  0
Number of processes:    0
Number of cells:        479
  $_ANDNOT_   195
  $_AND_      2
  $_DFF_P_    59
  $_MUX_      56
  $_NAND_     34
  $_NOT_      1
  $_ORNOT_    65
  $_OR_       6
  $_XNOR_     27
  $_XOR_      34
```

Figura 4.15: Síntesis de Código generado por Rust.

```
=== max_pipeline_tree ===

   Number of wires:                 317
   Number of wire bits:             429
   Number of public wires:           23
   Number of public wire bits:      135
   Number of memories:                0
   Number of memory bits:             0
   Number of processes:               0
   Number of cells:                 362
     $_ANDNOT_                      126
     $_DFFE_PP0P_                    67
     $_DFF_PP0_                       1
     $_MUX_                          56
     $_ORNOT_                        56
     $_XNOR_                         28
     $_XOR_                          28
```

Figura 4.16: Síntesis de Código Verilog generado a mano.

#### 4.4.3.3. Visualización de señales

En todos los casos, se observó que la señal de salida `valid_out` se activaba de forma sincronizada con la aparición del valor máximo, exactamente tres ciclos después de la entrada válida, confirmando así el correcto funcionamiento y la fiabilidad de la lógica secuencial.

Los conjuntos de prueba utilizados fueron equivalentes, diseñados específicamente para validar los aspectos funcionales del diseño y asegurar que todos los escenarios fueran cubiertos. Como resultado, se pudo comprobar que las comparaciones se ejecutaban correctamente, lo que garantiza la robustez y estabilidad del sistema. La coherencia observada en las señales entre ambas implementaciones respalda la funcionalidad del diseño, independientemente del entorno de desarrollo utilizado, lo que demuestra la versatilidad y fiabilidad del enfoque adoptado en ambos lenguajes.

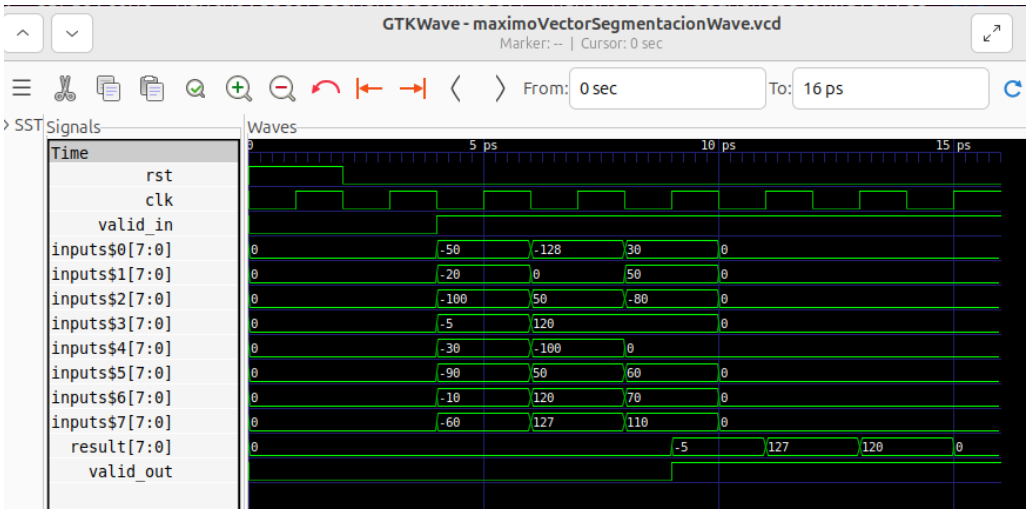


Figura 4.17: Forma de Onda del código generado en Rust.

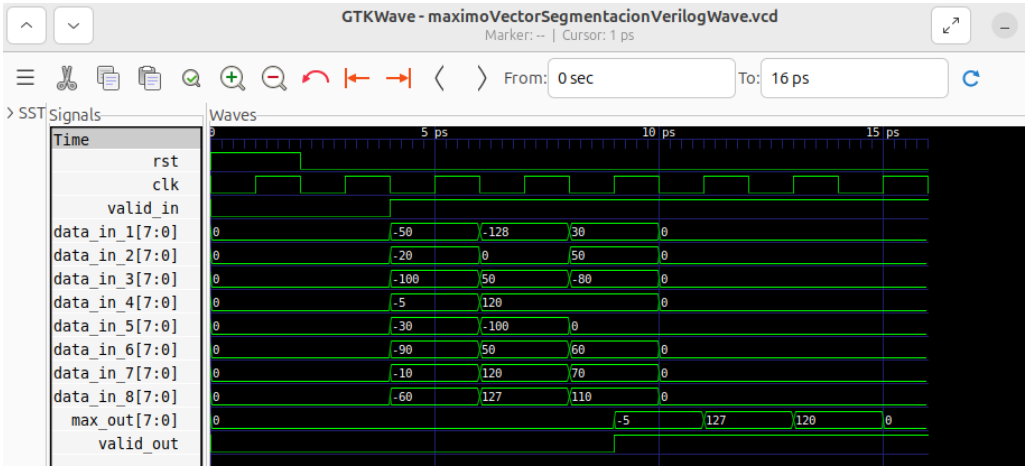


Figura 4.18: Forma de Onda del código Verilog generado a mano.

### 4.4.4. Observaciones particulares del caso

La ausencia de operaciones de casting o ampliación de tipo redujo la complejidad del diseño, permitiendo centrarse exclusivamente en la implementación del pipeline y la gestión de señales de control.

El principal valor de este caso reside en la correcta implementación del paralelismo estructural, que permitió obtener resultados con una latencia controlada y reproducible. El uso de señales `valid_in` y `valid_out` añadió realismo al componente y facilitó el control del flujo de datos válidos a través del sistema.

Una ventaja significativa del entorno RustHDL en este caso fue su capacidad nativa para trabajar con arrays, lo que agiliza notablemente la codificación de operaciones iterativas o estructuradas como las comparaciones entre elementos. Esta característica facilita el desarrollo en comparación con Verilog, donde la manipulación de vectores requiere mayor trabajo y control manual.

Sin embargo, al igual que en casos anteriores, el diseño en RustHDL mostró un aumento en el número de componentes utilizados, principalmente debido a la cantidad de registros (DFFs) necesarios para implementar la segmentación. Este enfoque facilita una estructuración más clara y acelera el desarrollo, pero también incrementa el uso de recursos lógicos, lo que podría representar una limitación en entornos con restricciones de hardware.

Como en casos anteriores, la implementación manual en Verilog resulta algo más eficiente en términos de uso de recursos lógicos, aunque el diseño en RustHDL ofrece una mayor claridad y rapidez en el desarrollo, especialmente al trabajar con estructuras segmentadas.

En resumen, este caso demostró que la segmentación en RustHDL es completamente viable para operaciones de comparación, y que el entorno permite desarrollar componentes robustos, predecibles y funcionalmente equivalentes a los escritos ma-

nualmente en Verilog, aunque hay un incremento en el uso de recursos durante la síntesis, especialmente debido al uso excesivo de registros.

## 4.5. Caso 5: Producto escalar secuencial

### 4.5.1. Descripción del módulo

El quinto caso de estudio se centró en la implementación de un módulo para realizar el cálculo del producto escalar secuencial de dos vectores de 8 bits sin signo. El producto escalar es una operación matemática fundamental que toma dos secuencias de números y genera un único valor como resultado de la suma de los productos de sus elementos correspondientes. En este caso, los dos vectores de entrada, $a$ y $b$, tienen una longitud de 8 bits sin signo, y la salida es un valor de 16 bits sin signo que representa el resultado de la multiplicación de estos elementos sumados.

El diseño sigue una estructura secuencial, donde los valores de $a$ y $b$ se multiplican y acumulan de manera iterativa, con un registro de la salida intermedia en cada ciclo de reloj. Además de los vectores de entrada, el módulo tiene señales de control como *reset*, *start*, *valid*, y una señal *busy* que indica si el módulo está ocupado realizando la operación. La señal de salida *valid* se activa cuando el cálculo está completo y el resultado está disponible.

Una particularidad de este desarrollo fue la necesidad de implementar un multiplicador propio, ya que la versión de RustHDL utilizada no proporciona un multiplicador genérico integrado para realizar multiplicaciones de 8 bits sin signo. En lugar de usar un multiplicador genérico, se implementó un multiplicador de 8x8 bits manualmente, simulando el funcionamiento de un multiplicador binario mediante la acumulación de productos parciales. Este enfoque permitió realizar las multiplicaciones necesarias para calcular el producto escalar de forma eficiente y controlada.

### 4.5.2. Metodología de verificación

La verificación del módulo se llevó a cabo mediante simulaciones funcionales en el entorno de Icarus Verilog. Los escenarios de prueba fueron diseñados para verificar que el producto escalar calculado fuera correcto. Se utilizaron diferentes combinaciones de valores de entrada para los vectores $a$ y $b$, incluyendo valores pequeños y grandes, con el objetivo de garantizar que el módulo pudiera manejar cualquier combinación de vectores sin signo de 8 bits. Además, se evaluó el comportamiento de las señales de control *start*, *valid* y *busy* durante el cálculo.

Durante las simulaciones, se confirmaron los siguientes aspectos clave:

- La correcta multiplicación de los elementos de los vectores $a$ y $b$.

- La correcta acumulación de los productos parciales en el resultado final.
- La correcta activación de las señales de control *valid* y *busy*.
- El cálculo correcto del producto escalar, de acuerdo con los valores esperados.

### 4.5.3. Comparación de resultados

#### 4.5.3.1. Correcto funcionamiento

El módulo demostró un comportamiento correcto durante todas las pruebas. En cada ciclo de reloj, se multiplicaban correctamente los elementos correspondientes de los vectores $a$ y $b$, y los productos parciales se acumulaban correctamente en la salida de 16 bits. Las señales de control *valid* y *busy* se gestionaron adecuadamente, garantizando que el módulo indicara cuándo estaba ocupado realizando el cálculo y cuándo el resultado estaba disponible.

En cuanto a la latencia, el módulo mostró un comportamiento predecible, con la salida válida disponible después de un número adecuado de ciclos de reloj, en función de la longitud de los vectores.

#### 4.5.3.2. Síntesis lógica

Durante el proceso de síntesis, se observó que el módulo utilizaba una cantidad considerable de recursos, ya que se optó por implementar manualmente el multiplicador utilizando sumas y desplazamientos bit a bit. Aunque RustHDL ofrece una implementación de multiplicación para tipos `signed` de 16 a 32 bits, se decidió desarrollar una versión personalizada con el objetivo de explorar la viabilidad de construir módulos propios para operaciones fundamentales.

Este enfoque permitió comprobar que es posible definir manualmente bloques básicos como la suma o la multiplicación, lo que proporciona un mayor control sobre la estructura del diseño y puede resultar útil en contextos donde se requiere una optimización específica o una arquitectura no estándar.

A pesar de las diferencias en el enfoque de implementación, la diferencia en el número total de componentes y puertas lógicas utilizados entre la versión generada con RustHDL y la escrita manualmente en Verilog resultó ser aceptable. Aunque la implementación en Verilog presentó una ligera reducción en la cantidad de lógica sintetizada, la versión en RustHDL se mantuvo dentro de márgenes razonables, lo que valida la viabilidad del diseño generado automáticamente.

```
=== productoEscalar (Agrupado) ===

Number of wires:        740
Number of wire bits:    922
Number of public wires: 33
Number of public wire bits: 215
Number of memories:     0
Number of memory bits:  0
Number of processes:    0
Number of cells:        806
  $_ANDNOT_   214
  $_AND_      16
  $_DFF_P_    25
  $_DLATCH_N_ 16
  $_MUX_      103
  $_NAND_     39
  $_NOR_      22
  $_NOT_      64
  $_ORNOT_    43
  $_OR_       121
  $_XNOR_     14
  $_XOR_      129
```

Figura 4.19: Síntesis de Código generado por Rust.

```
=== productoEscalar (Agrupado) ===

Number of wires:        569
Number of wire bits:    681
Number of public wires: 15
Number of public wire bits: 109
Number of memories:     0
Number of memory bits:  0
Number of processes:    0
Number of cells:        625
  $_ANDNOT_     191
  $_AND_        17
  $_DFFE_PP0P_  36
  $_DFF_PP0_    1
  $_MUX_        57
  $_NAND_       39
  $_NOR_        24
  $_NOT_        39
  $_ORNOT_      17
  $_OR_         66
  $_XNOR_       12
  $_XOR_        126
```

Figura 4.20: Síntesis de Código Verilog generado a mano.

#### 4.5.3.3. Visualización de señales

La simulación demostró que la señal *valid* se activaba correctamente al inicio del cálculo, y que la señal *busy* indicaba adecuadamente el estado de ocupación del módulo durante la operación. Las transiciones de ambas señales fueron coherentes con el ciclo de reloj, y la propagación de los productos parciales, junto con su acumulación, se observó de forma clara y estructurada. Cabe destacar que se identificó una ligera variación en el momento de activación de la señal *busy*, aunque esta diferencia no afecta al funcionamiento ni al resultado del módulo.

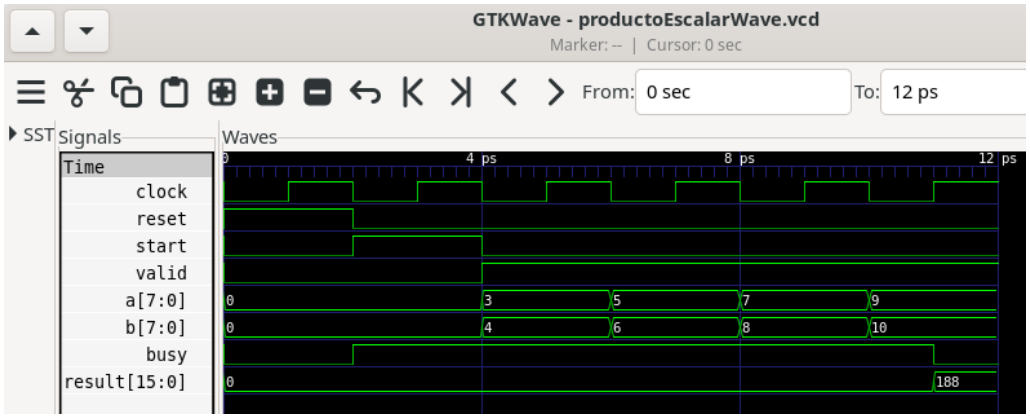


Figura 4.21: Forma de Onda del código generado en Rust.

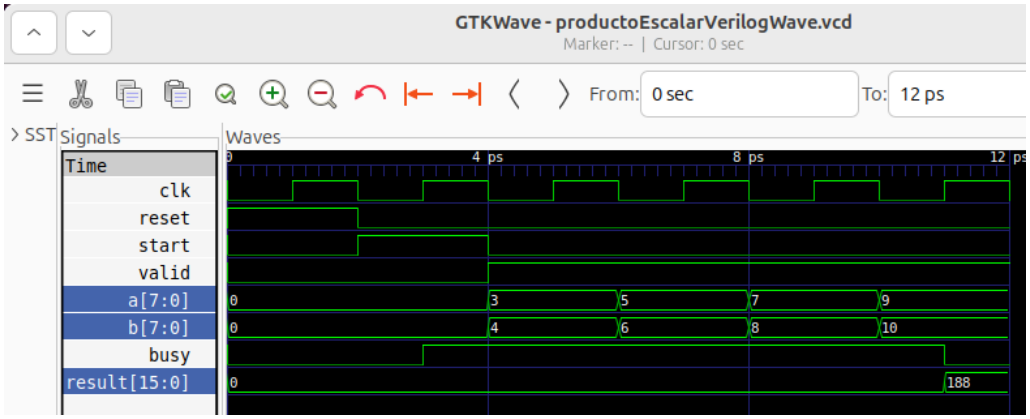


Figura 4.22: Forma de Onda del código Verilog generado a mano.

### 4.5.4. Observaciones particulares del caso

Este caso ofreció una oportunidad práctica para comprobar que, al igual que en otros lenguajes de descripción hardware como Verilog o VHDL, en RustHDL es posible construir manualmente bloques funcionales personalizados, incluso para operaciones básicas como la multiplicación. Aunque en RustHDL ya existe una implementación funcional del multiplicador para tipos **Signed(16)** a **Signed(32)**, se optó por desarrollar una versión propia con el fin de explorar su capacidad para definir módulos a bajo nivel. Este ejercicio permitió validar la viabilidad del diseño secuencial para el cálculo del producto escalar entre vectores de 8 bits.

Un aspecto importante de este caso fue la gestión de las señales de control *start*, *valid* y *busy*. La correcta activación de estas señales garantizó que el módulo operara de manera secuencial y controlada, y que el usuario pudiera saber en qué momento el módulo estaba listo para recibir nuevos datos.

A nivel de recursos, la implementación manual del multiplicador y la acumulación de los productos parciales en RustHDL implicaron un uso algo mayor de registros y lógica combinacional en comparación con un diseño equivalente implementado manualmente en Verilog. Esta comparación se realizó como prueba adicional, confirmando que la versión en Verilog presenta un consumo ligeramente inferior. No obstante, la diferencia en número de componentes es aceptable, y la claridad estructural y el mayor control que ofrece RustHDL en este tipo de diseños justifican el enfoque adoptado.

En resumen, este caso demostró cómo la implementación de operaciones matemáticas fundamentales como el producto escalar puede realizarse de manera efectiva en RustHDL, incluso cuando no se dispone de componentes predefinidos como multiplicadores. La flexibilidad del entorno y la capacidad para gestionar de forma explícita las señales de control y los registros intermedios proporcionan un enfoque robusto y funcional para el diseño de este tipo de operaciones.

## 4.6. Caso 6: Multiplicación de matrices 2x2 combinacional

### 4.6.1. Descripción del módulo

Este sexto caso de estudio se centró en la implementación combinacional de un módulo para realizar la multiplicación de matrices 2x2 con valores con signo. Las matrices de entrada están compuestas por elementos de tipo **Signed(4)**, mientras que la salida es de tipo **Signed(32)**. Dado que este diseño involucra múltiples operaciones de multiplicación y suma de forma simultánea, lo que lo hace más complejo que los casos anteriores, se optó por utilizar directamente el multiplicador nativo que proporciona RustHDL. Esta decisión permitió simplificar la implementación y centrarse en la estructura general del módulo, aprovechando la compatibilidad de RustHDL con multiplicaciones de tipo **Signed(16)** a **Signed(32)**.

El módulo toma como entrada dos matrices cuadradas de dimensión 2x2 y genera como salida una nueva matriz de la misma dimensión que resulta del producto matricial. El producto de matrices se realiza según la operación estándar: C=A×B donde cada elemento de la matriz resultante se obtiene como la suma de los productos de la fila i de la matriz A y la columna j de la matriz B. El módulo incluye una señal de entrada de `reset` para inicializar su estado, aunque el cálculo se realiza de forma puramente combinacional. Durante el desarrollo se encontró la limitación de que RustHDL no cuenta con un multiplicador genérico completamente parametrizable, lo que condicionó la implementación. Por esta razón, se utilizó el multiplicador de 16x16 bits con salida de 32 bits como base, y se adaptó la lógica para trabajar con entradas de 4 bits con signo.

### 4.6.2. Metodología de verificación

La verificación del módulo se llevó a cabo mediante simulaciones funcionales usando el entorno Icarus Verilog. Inicialmente, se diseñaron múltiples escenarios de prueba con valores específicos — incluyendo positivos, negativos, ceros y combinaciones extremas — para comprobar que la multiplicación de matrices funcionaba correctamente en casos representativos.

Algunas de las pruebas realizadas fueron:

- Multiplicación de matrices: $\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right) \cdot \left(\begin{smallmatrix}5&6\\7&-8\end{smallmatrix}\right) = \left(\begin{smallmatrix}19&-10\\43&-14\end{smallmatrix}\right)$
- Multiplicación con identidad negativa: $\left(\begin{smallmatrix}-1&0\\0&-1\end{smallmatrix}\right) \cdot \left(\begin{smallmatrix}2&3\\4&5\end{smallmatrix}\right) = \left(\begin{smallmatrix}-2&-3\\-4&-5\end{smallmatrix}\right)$
- Casos con ceros y positivos: $\left(\begin{smallmatrix}3&1\\0&2\end{smallmatrix}\right) \cdot \left(\begin{smallmatrix}1&0\\2&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}5&1\\4&2\end{smallmatrix}\right)$
- Combinaciones mixtas: $\left(\begin{smallmatrix}2&2\\1&3\end{smallmatrix}\right) \cdot \left(\begin{smallmatrix}1&4\\2&0\end{smallmatrix}\right) = \left(\begin{smallmatrix}6&8\\7&4\end{smallmatrix}\right)$

- Casos extremos: $\begin{pmatrix} -8 & -8 \\ -8 & -8 \end{pmatrix} \cdot \begin{pmatrix} -8 & -8 \\ -8 & -8 \end{pmatrix} = \begin{pmatrix} 128 & 128 \\ 128 & 128 \end{pmatrix}$
- Casos mínimos: $\begin{pmatrix} -8 & -8 \\ -8 & -8 \end{pmatrix} \cdot \begin{pmatrix} 7 & 7 \\ 7 & 7 \end{pmatrix} = \begin{pmatrix} -112 & -112 \\ -112 & -112 \end{pmatrix}$

Estas pruebas iniciales permitieron validar la funcionalidad básica del módulo de multiplicación de matrices. Para asegurar una verificación más exhaustiva, se realizó una prueba sistemática en RustHDL y Verilog que abarcó todas las combinaciones posibles de valores en el rango de **-2 a 2** para cada elemento. Este test exhaustivo garantiza un comportamiento correcto en una amplia variedad de escenarios y continuará siendo la base para futuras validaciones.

Ampliar el rango de valores es factible desde el punto de vista computacional. Sin embargo, dado el volumen de combinaciones resultante, este enfoque pierde utilidad práctica y visual, especialmente a la hora de interpretar resultados. Por ello, se optó por limitar el rango a un subconjunto reducido pero representativo.

### 4.6.3. Comparación de resultados

#### 4.6.3.1. Correcto funcionamiento

El módulo combinacional produjo resultados correctos en todos los casos evaluados. La multiplicación de cada par de elementos de fila y columna se realizó correctamente y la suma de los productos fue precisa, obteniendo siempre las salidas esperadas. Al tratarse de un módulo combinacional, la respuesta fue inmediata tras aplicar los valores de entrada, sin dependencia del reloj.

La señal de `reset` no influyó directamente en la lógica de cálculo, pero fue útil para mantener un comportamiento coherente en futuras ampliaciones o integraciones en sistemas secuenciales más complejos.

#### 4.6.3.2. Síntesis lógica

La síntesis del módulo mostró un consumo elevado de recursos para un diseño que, a priori, podría parecer sencillo. Esto se debe a dos factores clave:

- El uso de multiplicadores de 16x16 bits para realizar operaciones entre valores de 4 bits con signo, debido a las limitaciones de RustHDL.
- La naturaleza combinacional del diseño, que requiere que todas las operaciones estén disponibles simultáneamente, generando múltiples caminos lógicos en paralelo.

En este caso, ambas síntesis resultaron prácticamente idénticas en cuanto a utilización de recursos, sin diferencias significativas entre la implementación en RustHDL y la versión manual equivalente en Verilog.

```
=== multiplicacionMatrices ===

   Number of wires:               1070
   Number of wire bits:           1218
   Number of public wires:          13
   Number of public wire bits:     161
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               1093
     $_ANDNOT_                     400
     $_AND_                         40
     $_MUX_                          4
     $_NAND_                        82
     $_NOR_                         12
     $_NOT_                         29
     $_ORNOT_                       36
     $_OR_                         132
     $_XNOR_                        24
     $_XOR_                        334
```

Figura 4.23: Síntesis de Código generado por Rust.

```
=== multiplicacion_Matrices ===

   Number of wires:               1070
   Number of wire bits:           1218
   Number of public wires:          13
   Number of public wire bits:     161
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               1093
     $_ANDNOT_                     400
     $_AND_                         40
     $_MUX_                          4
     $_NAND_                        82
     $_NOR_                         12
     $_NOT_                         29
     $_ORNOT_                       36
     $_OR_                         132
     $_XNOR_                        24
     $_XOR_                        334
```

Figura 4.24: Síntesis de Código Verilog generado a mano.

#### 4.6.3.3. Visualización de señales

Dado que se trata de un módulo combinacional, las señales de salida reaccionaban de forma inmediata ante cualquier cambio en las entradas. Durante la simulación, se observó que las salidas correspondientes a la matriz resultante se actualizaban sin latencia, cambiando los valores de manera instantánea tras cualquier cambio en la entrada.

En cuanto a las pruebas realizadas, la simulación en Verilog se centró en una serie de casos predefinidos de matrices con valores específicos, diseñados para validar el correcto funcionamiento del módulo en diferentes configuraciones relevantes.

Por otro lado, en ambas simulaciones, tanto en RustHDL como en Verilog, se optó por recorrer sistemáticamente un rango de valores entre -2 y 2 para cada elemento de la matriz. Esta elección permitió validar un conjunto significativo de combinaciones, manteniendo un equilibrio adecuado entre cobertura funcional y tiempo de simulación.

Para la validación del módulo de multiplicación de matrices $2 \times 2$, se eligió un rango de valores entre $-2$ y 2 para cada uno de los elementos. Como cada matriz tiene 4 elementos y se multiplican dos matrices, el total de operandos es:

$$\text{Total de operandos} = 4 \text{ (elementos por matriz)} \times 2 \text{ (matrices)} = 8$$

Con 5 posibles valores por operando $(-2, -1, 0, 1, 2)$, el número total de combinaciones a simular es:

$$\text{Total de combinaciones} = 5^8 = 390{,}625$$

En cambio, si se ampliara el rango a valores entre $-8$ y 7, es decir, 16 posibles valores por operando, el número total de combinaciones sería:

$$16^8 = 4{,}294{,}967{,}296$$

Este crecimiento exponencial hace que una simulación completa sea inviable para rangos más amplios, por lo que se optó por trabajar con un subconjunto representativo.

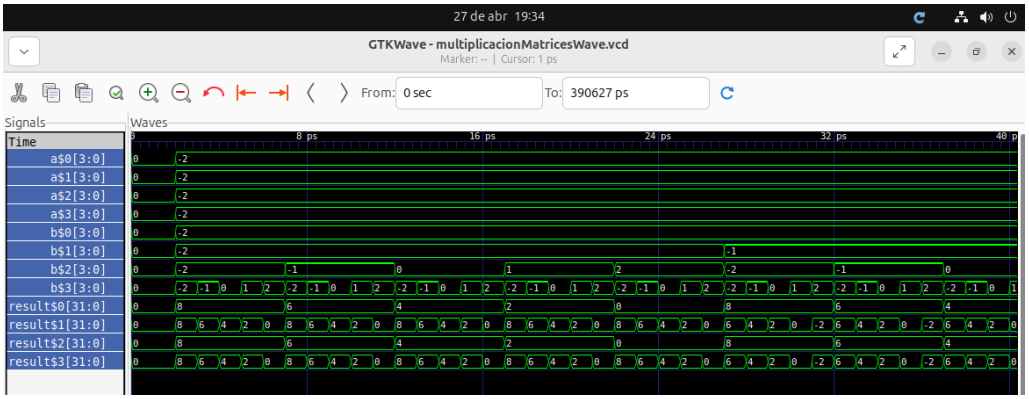


Figura 4.25: Forma de Onda del código generado en Rust.

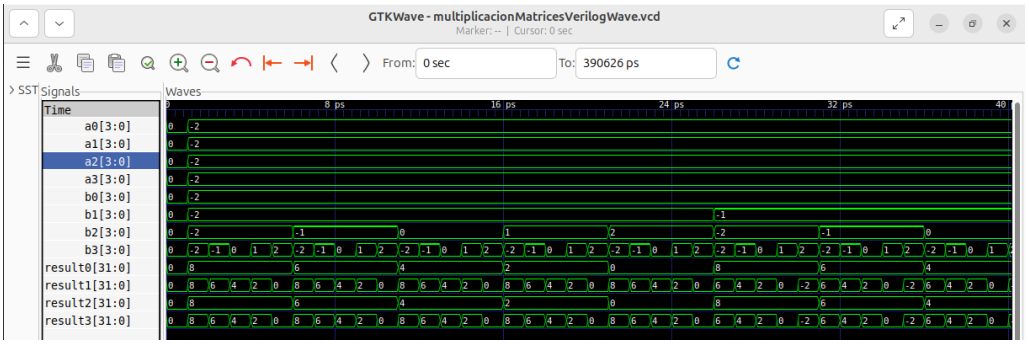


Figura 4.26: Forma de Onda del código Verilog generado a mano.

### 4.6.4. Observaciones particulares del caso

Este caso permitió explorar las implicaciones del diseño puramente combinacional aplicado a una operación matricial. La principal limitación detectada fue la falta de soporte para multiplicadores genéricos en RustHDL, lo que obligó a usar multiplicadores de 16 bits incluso para operandos de solo 4 bits. Esta decisión incrementó innecesariamente el uso de recursos en la síntesis.

Aunque la operación de multiplicación de matrices 2x2 es sencilla desde un punto de vista funcional, su implementación combinacional directa genera un diseño relativamente denso, en términos de lógica, debido al número de operaciones aritméticas involucradas.

Por último, si bien el módulo mostró un comportamiento correcto durante la simulación, este tipo de enfoque no resulta escalable para tamaños de matriz mayores ni adecuado para sistemas con limitaciones estrictas de área o consumo energético.

## 4.7. Caso 7: Multiplicación de Matrices 2x2 Secuencial

### 4.7.1. Descripción del módulo

Este séptimo caso de estudio aborda la implementación secuencial de un módulo encargado de realizar la **Multiplicación de Matrices 2x2** con valores con signo. Cada elemento de las matrices de entrada es de tipo `signed(4)`, mientras que las salidas se definen como `signed(32)`, debido a las limitaciones del entorno de desarrollo RustHDL, que impone multiplicadores de 16 bits como mínimo, generando una salida de 32 bits.

El módulo incluye una interfaz de control compuesta por las señales `clk`, `reset`, `start` y `done`, permitiendo así una coordinación clara del inicio y finalización del cálculo. Internamente, se utilizan registros tipo DFF para mantener el estado del sistema y avanzar a través de una máquina de estados que organiza la operación de multiplicación en fases secuenciales.

### 4.7.2. Metodología de verificación

La verificación del módulo secuencial se realizó siguiendo un procedimiento similar al empleado en la versión combinacional. Inicialmente, en ambos entornos — Icarus Verilog y RustHDL — se llevaron a cabo una serie de pruebas funcionales con matrices específicas, que incluían valores positivos, negativos, ceros y combinaciones extremas, con el objetivo de validar el correcto funcionamiento básico del módulo de multiplicación. Entre estos casos destacan:

- **Test de producto máximo:** Se utilizaron matrices llenas de valores mínimos (-8) para obtener el resultado máximo posible (128 en cada celda de la matriz resultante).
- **Test de producto mínimo:** Con matrices formadas por -8 y 7, se obtuvo el valor mínimo esperado de -112 por celda.

- **Test mixto:** Se emplearon combinaciones de valores positivos y negativos para evaluar el comportamiento en condiciones más heterogéneas. El resultado esperado fue una matriz con entradas tanto positivas como negativas.
- **Test con ceros:** Se verificó el manejo correcto de multiplicaciones en las que varios elementos eran cero, comprobando que el módulo no introducía errores en los cálculos parciales.

Estos tests iniciales permitieron asegurar la correcta implementación de la lógica secuencial y la activación adecuada de las señales de control `start` y `done`, así como la precisión en los resultados.

Posteriormente, para garantizar una cobertura más amplia y exhaustiva, se implementó una prueba sistemática en RustHDL y Verilog que exploró todas las combinaciones posibles de valores en el rango de **-2 a 2** para cada uno de los elementos de las matrices de entrada. Esta segunda fase de pruebas replicó el enfoque usado en el módulo combinacional y proporcionó una validación profunda del comportamiento del sistema dentro de un subconjunto representativo del espacio total de valores de 4 bits con signo.

De esta forma, la combinación de pruebas específicas iniciales y la exploración exhaustiva en un rango controlado asegura la robustez y fiabilidad del diseño secuencial frente a una amplia variedad de escenarios.

## 4.7.3. Comparación de resultados

### 4.7.3.1. Correcto funcionamiento

El módulo demostró un funcionamiento correcto durante todas las pruebas, mostrando una operación secuencial estable y predecible. La lógica de control mediante las señales `start` y `done` permitió identificar de manera precisa cuándo se iniciaba el cálculo y cuándo la salida estaba disponible. En cada ejecución, los productos parciales se calcularon correctamente siguiendo la secuencia esperada, y se almacenaron en registros intermedios hasta completar la matriz final de salida.

La señal `done` se activó tras el número esperado de ciclos de reloj, lo cual confirma que la máquina de estados funciona correctamente. En resumen, el módulo gestionó de manera adecuada tanto el cálculo como la coordinación temporal, entregando resultados consistentes y esperados para los casos de prueba definidos.

### 4.7.3.2. Síntesis lógica

La síntesis lógica del módulo mostró un incremento en el número de componentes respecto a versiones más optimizadas, debido al uso intensivo de DFF para implementar la lógica secuencial. Esta característica es propia del enfoque secuencial,

donde se distribuye el cómputo a lo largo de múltiples ciclos de reloj. No obstante, esta implementación utiliza menos recursos que su contraparte combinacional, al menos en Verilog.

Sin embargo, en el caso generado automáticamente por RustHDL, el número de componentes fue incluso mayor que en la versión combinacional. Esto se debe a que RustHDL tiende a generar una jerarquía de subcomponentes más compleja, con estructuras adicionales que incrementan el uso de recursos lógicos. Cabe destacar que tanto en la implementación combinacional como en la secuencial se utiliza el multiplicador proporcionado por defecto por la herramienta de síntesis, y no un diseño personalizado. Por tanto, la implementación secuencial manual en Verilog se mantiene como la opción más eficiente en cuanto al consumo de recursos, comparada tanto con su equivalente combinacional como con el diseño generado desde RustHDL.

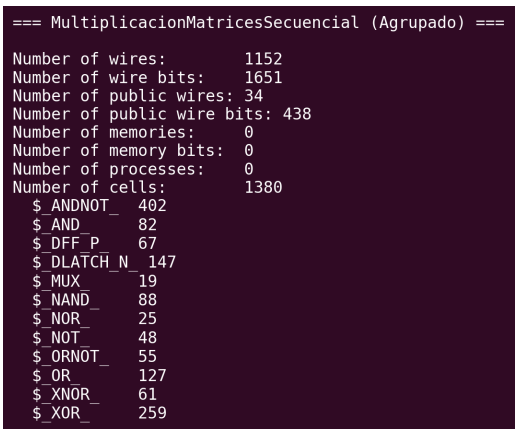


Figura 4.27: Síntesis de Código generado por Rust (secuencial).

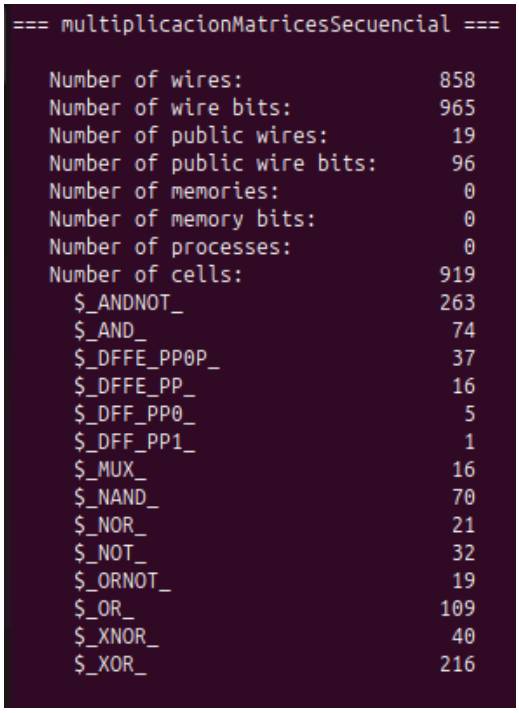


Figura 4.28: Síntesis de Código Verilog generado a mano (secuencial).

#### 4.7.3.3. Visualización de señales

Las simulaciones muestran que la evolución del sistema es correcta y coherente con una arquitectura secuencial. En ambos casos, las señales de control responden como se espera: `start` lanza la operación, `done` marca su finalización, y los productos intermedios se acumulan en ciclos consecutivos, lo que confirma un flujo de datos ordenado y predecible durante todo el proceso.

Tanto en RustHDL como en Verilog, se realizó una verificación exhaustiva recorriendo sistemáticamente todos los casos dentro del rango de valores **-2 a 2** para

cada elemento, proporcionando así una amplia cobertura que permite detectar posibles errores relacionados con el signo, overflow o comportamientos inesperados ante diferentes combinaciones de entrada.

A pesar de estas diferencias en los vectores de prueba iniciales, la forma de las señales en las waves es similar en ambos casos. Se observa una correcta sincronización con el reloj, activaciones precisas y consistentes de la señal `done`, y un comportamiento estable en el manejo de los estados internos del módulo, lo que refuerza la fiabilidad del diseño en entornos secuenciales.

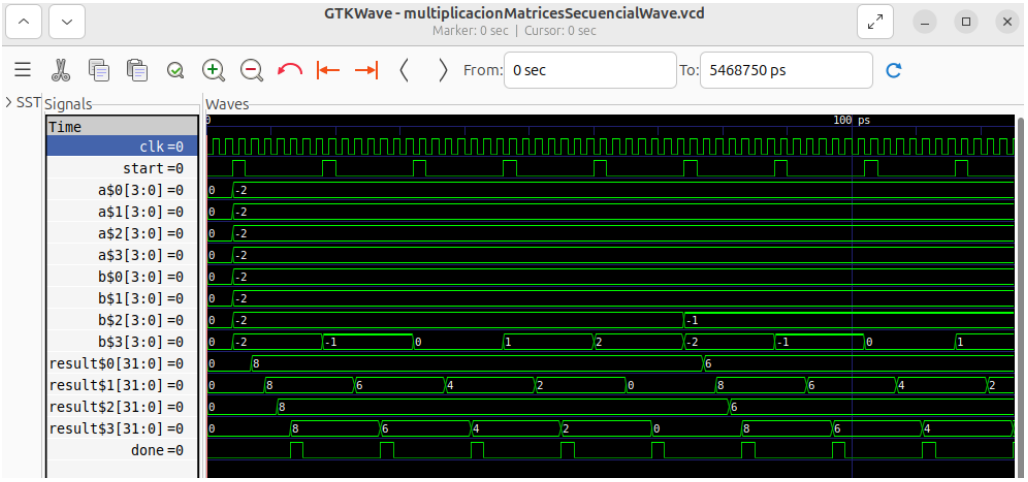


Figura 4.29: Forma de Onda del código generado en Rust (secuencial).

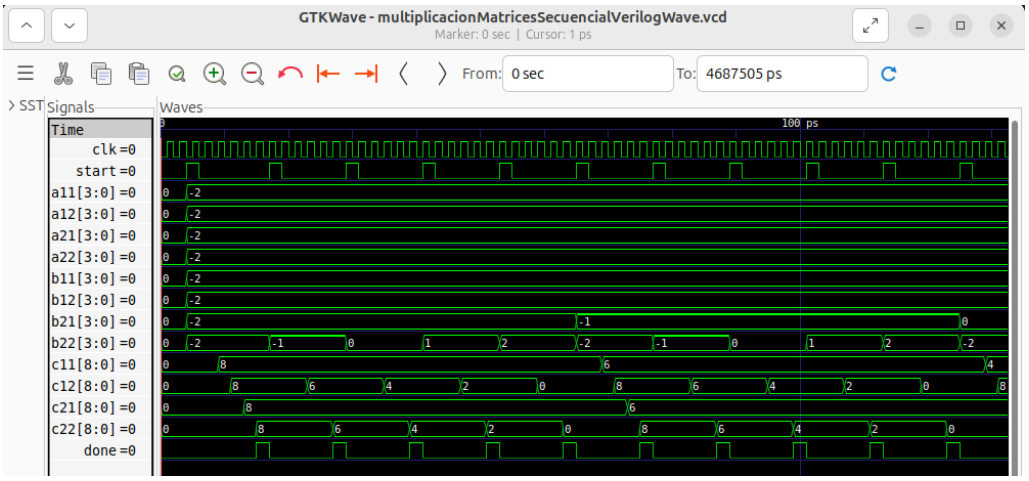


Figura 4.30: Forma de Onda del código Verilog generado a mano (secuencial).

### 4.7.4. Observaciones particulares del caso

Este caso permitió explorar el funcionamiento de una implementación secuencial para la multiplicación de matrices 2x2, complementando el enfoque combinacional

previamente desarrollado. La idea principal fue evaluar cómo se comporta esta arquitectura secuencial, que distribuye el cálculo a lo largo de varios ciclos de reloj, reduciendo la complejidad lógica en cada instante a cambio de una mayor latencia.

Aunque el diseño secuencial sigue utilizando multiplicadores de 16 bits debido a las limitaciones de RustHDL, el impacto en el uso de recursos se atenúa gracias a este enfoque escalonado. Sin embargo, la versión generada por RustHDL continúa empleando más recursos que la versión combinacional, mientras que la implementación en Verilog muestra un uso de recursos menor en su versión secuencial comparada con la combinacional.

Esta arquitectura secuencial resulta más escalable que la versión combinacional, especialmente en escenarios donde el ahorro de área es más importante que la velocidad inmediata. Además, el uso de señales de control como `start` y `done` facilita la integración en sistemas sincrónicos más complejos.

Capítulo 5

# Conclusiones y Trabajo Futuro

El desarrollo de este trabajo ha permitido explorar las capacidades del lenguaje `Rust` y su extensión para diseño hardware, `RustHDL`, como alternativa moderna para la generación de código de bajo nivel en sistemas digitales. A lo largo del proceso de implementación, simulación y análisis, se han podido extraer diversas conclusiones relevantes sobre su viabilidad, eficiencia y aplicabilidad práctica, tanto en comparación con enfoques tradicionales como con su potencial de evolución a futuro.

## 5.1. Conclusiones

En primer lugar, se ha comprobado que `RustHDL` es una herramienta potente y flexible para el diseño de sistemas digitales. Su integración con el ecosistema de `Rust` proporciona ventajas importantes en términos de organización, modularidad y verificación del código. La posibilidad de realizar pruebas de forma rápida, reutilizar estructuras de datos complejas como arrays y vectores, y trabajar en un entorno moderno de desarrollo con herramientas como `cargo test` y `cargo run`, representa un salto cualitativo respecto al flujo de trabajo tradicional en Verilog.

Durante las pruebas realizadas, se ha observado que la conversión automática de módulos combinacionales desde `RustHDL` a Verilog genera un código competitivo. En la mayoría de los casos, los resultados de síntesis (usando `Yosys`) muestran un uso de recursos similar al que se obtiene con código Verilog escrito manualmente. Esta equivalencia funcional es especialmente destacable en módulos puramente combinacionales, donde las transformaciones automáticas no introducen sobrecostes relevantes.

Las principales dificultades aparecieron al abordar la implementación de la lógica secuencial. En nuestra primera aproximación surgieron varios problemas, principalmente relacionados con el uso de registros para conservar el estado a lo largo de los

ciclos de reloj. Sin embargo, una vez comprendido el funcionamiento del componente `DFF` (flip-flop) y su integración en el diseño, el desarrollo se volvió considerablemente más sencillo.

Aun así, el uso de estos componentes, en particular su variante `EdgeDFF`, que permite reaccionar a flancos de reloj, puede hacer que el código generado resulte menos intuitivo y más difícil de mantener. Esta complejidad se ve acentuada por la falta de documentación oficial, ejemplos prácticos o guías detalladas sobre su uso en `RustHDL`. Como consecuencia, el proceso de desarrollo tiende a apoyarse en un enfoque de prueba y error, lo que puede dificultar el aprendizaje y ralentizar el avance del proyecto, especialmente para quienes no cuentan con una experiencia sólida en diseño digital con Verilog.

Además, se han detectado ciertas limitaciones técnicas en el propio `RustHDL`. Por ejemplo, las simulaciones presentan problemas al trabajar con números negativos o con componentes que deben manejar señales con signo, especialmente en el caso de multiplicadores genéricos. Estas restricciones están relacionadas con aspectos todavía inestables del `kernel` del lenguaje, y aunque es previsible que se solucionen en versiones futuras, actualmente suponen un freno a la adopción de la herramienta en entornos profesionales exigentes.

A pesar de estas limitaciones, hay aspectos muy positivos. La capacidad de generar automáticamente bancos de pruebas y simularlos de forma directa desde `Rust` representa una gran mejora en la eficiencia del ciclo de desarrollo. La integración con simuladores como `Icarus Verilog` desde el propio entorno de desarrollo es otro punto a favor, eliminando la necesidad de utilizar herramientas externas o flujos complejos. La posibilidad de trabajar directamente con estructuras como arrays para modelar cálculos vectoriales o matriciales es especialmente útil en aplicaciones de procesamiento de señales, comunicaciones o IA, donde este tipo de operaciones son frecuentes.

Cabe destacar que la salida generada por `RustHDL` corresponde a código Verilog estándar, lo que garantiza una alta compatibilidad con flujos clásicos de síntesis y simulación. No obstante, aunque el uso exclusivo de Verilog podría parecer restrictivo frente a las funcionalidades avanzadas de `SystemVerilog`, en la práctica ambos lenguajes pueden integrarse sin dificultad en un mismo proyecto, como demuestran numerosos diseños open-source basados en RISC-V.

En resumen, `RustHDL` presenta un enfoque prometedor para el diseño de hardware digital. Su principal defecto actual reside en la falta de madurez: documentación escasa, errores residuales en ciertas operaciones y una comunidad todavía pequeña. No obstante, su potencial es innegable. Con el paso del tiempo y el desarrollo de nuevas versiones, es razonable esperar que estas limitaciones se superen. En ese escenario, `RustHDL` podría consolidarse como una alternativa robusta y eficiente para el diseño de hardware, especialmente en proyectos que requieren alta modularidad, verificación automatizada y rapidez de desarrollo.

## 5.2. Trabajo Futuro

Entre las líneas de trabajo futuro más relevantes, destacan:

- **Profundización en la implementación de lógica secuencial compleja**: Investigar cómo optimizar el uso de componentes como `DFF` y `EdgeDFF`, y buscar patrones de diseño que minimicen el uso de recursos en síntesis.
- **Desarrollo de una librería de componentes estándar**: Crear una colección de módulos reutilizables y bien documentados para tareas comunes (registros, multiplexores, ALUs, etc.), que facilite el trabajo de futuros desarrolladores.
- **Contribución a la comunidad de `RustHDL`**: Participar activamente en foros, reportar errores y documentar ejemplos para mejorar la accesibilidad de la herramienta.
- **Comparativas más extensas con Verilog y SystemVerilog**: Ampliar los casos de estudio para evaluar el rendimiento en aplicaciones más exigentes, como controladores de memoria, interfaces de comunicación o procesadores simples.
- **Explorar la integración con FPGAs**: Integrar herramientas como `nextpnr` y `openFPGALoader` para generar el bitstream a partir del código en Rust y cargarlo directamente en una FPGA, completando así el flujo desde el diseño en alto nivel hasta la programación del dispositivo físico.

Con todo ello, se espera que `RustHDL` pueda consolidarse como una herramienta útil para el desarrollo de hardware, al facilitar la generación de código RTL desde un lenguaje de alto nivel como Rust. Su principal aportación radica en reducir la curva de aprendizaje y mejorar la seguridad y expresividad en la descripción de diseños, sin reemplazar el flujo tradicional de síntesis e implementación basado en RTL, que continuará siendo la base en entornos de producción tanto para ASICs como para FPGAs.

# Introduction

## 5.3. Motivation

Nowadays, significant attention is given to the development and evolution of high-level programming languages, often overshadowing more traditional ones. Although languages like C and C++ remain widely used due to their efficiency and closeness to hardware, they also carry certain limitations that have changed little over time. This has, to some extent, hindered the exploration of more modern approaches to digital system design.

In this context, upon discovering the existence of Rust, a programming language originally created by Graydon Hoare in 2010, which is still currently under development, we were very interested in exploring the possibilities it could offer. Rust is a relatively recent systems programming language, much newer than the others we had previously encountered. It claimed to address several shortcomings identified in other languages like C and C++, such as unsafe memory management and the difficulty of implementing error-free concurrency. It promised to be more efficient, faster, and safer than its predecessors, using a modern approach and high-quality integrated tools.

As Computer Engineering students, we have a special interest in hardware development, so these promises caught our attention, and we considered it relevant to put the language to the test in the context of developing digital systems, evaluating whether Rust can be a realistic and sustainable alternative in this field.

## 5.4. Objectives

The main objective of this work is to explore the Rust programming language by developing different short programs and analyzing its features through quantitative and qualitative metrics. Comparing these results with those obtained in other traditional low-level languages aims to evaluate the extent to which Rust delivers

on its promises of performance, security, and efficiency. To make these observations, we will use several programs (described in the next section) to help us compare code written in Rust with equivalent code written in Verilog.

In the world of programming, one of the fundamental goals is to achieve maximum efficiency, whether in terms of memory usage, execution speed, or scalability. This constant quest for efficiency has direct implications for system performance, energy consumption, and the utilization of available hardware resources.

In this regard, this work also aims to analyze whether Rust, thanks to its fine-grained system control and safe memory model, can be a viable alternative for developing projects that require close-to-the-hardware access, such as embedded systems, drivers, video games, or high-performance applications.

## 5.5. Work Plan

The work plan followed to achieve the previously mentioned objectives are:

1. **Language Familiarization:** Since it is a new language for us, the first stage of our project was to deeply understand the grammar and structure that a code written in Rust must follow. To do this, we turned to the official manuals for Rust and Rust-HDL, the latter being a Rust library for designing digital circuits, which will be key to our goal.
2. **Programs to use:** To achieve substantial comparisons with our Rust code, at the recommendation of our tutors, we decided to use these three programs:
   - **Icarus Verilog:** A simulator that compiles and executes Verilog code. We will use it to write code equivalent to that written in Rust and then compare it with the following tools.
   - **Yosys:** An open-source digital logic synthesizer. It converts Verilog designs to an optimized logic gate structure, which is often the first step in converting a design to a real circuit.
   - **GTKWave:** A simulation signal visualizer. It graphically displays the waves generated during a simulation from .vcd files, which can be generated from Verilog or Rust code using a tool already integrated into the language.
3. **First attempts:** Once the programs to be used were installed, we began to try to create a simple program, a two-signal adder, to put into practice all the theory previously studied.
4. **Creating more complex code:** Once we were sure we knew how to use all the tools and that they worked the way we needed, we moved on to developing more complex ideas that would test the limits of Rust.

The flow of all our tests will be to write the code in both Rust and Verilog, then simulate those codes and view the .vcd files in GTKWave to verify the behavior. If everything is correct, we will synthesize with Yosys to generate a netlist that can later be implemented on hardware components such as an FPGA.

# Conclusions and Future Work

The development of this project has allowed us to explore the de capacities of the `Rust` language and its extension to hardware design, `RustHDL`, as a modern alternative to the language for the development of low-level code generation in digital systems. Throughout the implementation process, simulation and analysis, various conclusions have been drawn about its viability, efficiency, and practical usage, both in comparison with traditional approaches and with its potential for future evolution.

## 5.6. Conclusions

First, `RustHDL` has proven to be a powerful and flexible tool for designing digital systems. Its integration with the `Rust` ecosystem provides significant advantages in terms of code organization, modularity, and verification. The ability to perform rapid testing, reuse complex data structures such as arrays and vectors, and work in a modern development environment with tools such as `cargo test` and `cargo run` represents a qualitative leap forward compared to the traditional Verilog workflow.

During testing, it has been observed that the automatic conversion of combinational modules from `RustHDL` to Verilog generates highly efficient code. In most cases, the synthesis results (using `Yosys`) show resource usage similar to that obtained with manually written Verilog code. This functional equivalence is especially notable in purely combinational modules, where automatic transformations do not introduce significant overhead.

However, the main limitations arise when attempting to implement sequential logic. In these cases, the need to use registers to maintain states across clock cycles introduces a high level of complexity into the design. The use of the `DFF` (flip-flop) component, and especially its `EdgeDFF` variant to react to clock edges, can make the generated code less intuitive and more difficult to maintain. This situation is exacerbated by the scarcity of official documentation, examples, or detailed guides on the use of these structures in `RustHDL`. In practice, this forces a trial-and-error approach, which hinders learning and slows down development, especially for users

without deep experience in digital design with Verilog.

In addition, certain technical limitations have been detected in `RustHDL` itself. For example, simulations present problems when working with negative numbers or components that must handle signed signals, especially in the case of generic multipliers. These restrictions are related to still unstable aspects of the `kernel` language, and although they are expected to be resolved in future versions, they currently hinder the tool's adoption in demanding professional environments.

Despite these limitations, there are some very positive aspects. The ability to automatically generate test benches and simulate them directly from `Rust` represents a great improvement in the efficiency of the development cycle. Integration with simulators such as `Icarus Verilog` from the development environment itself is another plus, eliminating the need to use external tools or complex workflows. The ability to work directly with structures such as arrays to model vector or matrix calculations is especially useful in signal processing, communications, or AI applications, where these types of operations are common.

It should also be noted that the output generated by `RustHDL` is standard Verilog code, not `SystemVerilog`, which ensures high compatibility with classic synthesis and simulation tool flows. However, this may also limit some modern functionality available only in `SystemVerilog`.

In summary, `RustHDL` presents a promising approach to the design of digital hardware. Its main current flaw lies in its lack of maturity: poor documentation, residual errors in certain operations, and a still small community. However, its potential is undeniable. With the passage of time and the development of new versions, it is reasonable to expect these limitations to be overcome. In that scenario, `RustHDL` could consolidate itself as a robust and efficient alternative to low-level hardware design, especially in projects that require high modularity, automated verification, and rapid development.

## 5.7. Future Work

Among the most relevant lines of future work, the following stand out:

- **Deep dive into the implementation of complex sequential logic**: Investigate how to optimize the use of components such as `DFF` and `EdgeDFF`, and look for design patterns that minimize resource usage in synthesis.
- **Development of a standard components library**: Create a collection of reusable and well-documented modules for common tasks (registers, multiplexers, ALUs, etc.) which will facilitate the work of future developers.
- **Contribution to the `RustHDL` community**: Actively participate in forums, report bugs, and document examples to improve the accessibility of the tool.

- **More extensive benchmarks with Verilog and SystemVerilog**: Expand the case studies to evaluate performance in more demanding applications, such as memory controllers, communication interfaces, or simple processors.
- **Explore integration with real FPGAs**: Automate the workflow from Rust code to deployment on physical hardware, using tools like `nextpnr` and `openFPGALoader`.

With all this, it is expected that in the near future, `RustHDL` will not only become a viable option for hardware designers, but can also be integrated into real production environments, offering a modern, secure and efficient alternative to traditional hardware description languages.

# Contribuciones Personales

## Oscar López Centenera

A lo largo del desarrollo del trabajo, he desempeñado un papel fundamental en diferentes fases del proyecto, asumiendo tareas técnicas, de desarrollo y de documentación. Mi participación comenzó desde la etapa inicial de preparación del entorno, y se extendió hasta la implementación de módulos complejos y la elaboración final de la memoria del proyecto.

En primer lugar, llevé a cabo la instalación y configuración de todas las herramientas necesarias. Esto incluyó el compilador de `Rust`, la extensión `RustHDL`, el entorno de desarrollo `Visual Studio Code`, y los programas de síntesis y simulación `Yosys` e `Icarus Verilog`. Cabe destacar que la instalación de `RustHDL` supuso un desafío particular. La documentación oficial es escasa y poco detallada, y fue mi primera experiencia instalando paquetes directamente desde el ecosistema de `Rust` mediante comandos y dependencias. Esto me obligó a recurrir a la ayuda de los tutores del proyecto en varias ocasiones, hasta conseguir que el entorno funcionara correctamente. Como resultado de este proceso, acabé elaborando algunos documentos complementarios con pasos básicos de instalación, que subimos al repositorio compartido, aunque reconozco que en el caso de `RustHDL` no llegamos a garantizar que funcionara en todos los entornos.

Una vez listo el entorno, comencé el desarrollo técnico con la implementación de un sumador con signo. Este fue diseñado inicialmente en `Verilog`, lo cual me permitió comprender a fondo la lógica combinacional requerida. A continuación, elaboré la versión correspondiente en `RustHDL`, adaptando la sintaxis y estructura del código al nuevo lenguaje. También preparé sus respectivos bancos de pruebas: utilicé el sistema de simulación `Icarus Verilog` para la versión original y el sistema de pruebas integrado de `Rust` (`cargo test - -nocapture`) para validar el módulo adaptado. En este proceso, partí de un test desarrollado por mi compañera para la versión en `Verilog`, que posteriormente adapté para que pudiera utilizarse también como base de prueba en `RustHDL`, asegurando así resultados coherentes.

En la siguiente fase, desarrollamos un reconocedor de patrones. Partiendo del diseño funcional en `Verilog`, el cual construimos conociendo ya la estructura y los tipos de entrada y salida esperados, procedí a implementar su equivalente en `RustHDL`, manteniendo la lógica de control y validando la salida en ambos entornos. Como en el caso anterior, primero realizamos las pruebas y simulaciones con `Icarus` y `Yosys`, para después validar su funcionamiento en `Rust` con su correspondiente test bench. Esto nos permitió observar directamente las diferencias en expresividad, legibilidad y capacidades de verificación de ambos lenguajes.

Uno de los módulos más interesantes fue el sumador en árbol con segmentación (*pipeline*), el cual representó un reto adicional al incorporar lógica secuencial y estructuras jerárquicas. Este módulo fue diseñado inicialmente en `Verilog`, utilizando `DFFs` para implementar la segmentación en varias etapas. Durante este proceso, compartí código y soluciones puntuales con mi compañera, ya que ambos nos encontramos con problemas similares al manejar los `DFFs` y definir la lógica de temporización. Una vez el diseño funcionaba correctamente en `Verilog`, procedí a replicarlo en `RustHDL`. Este paso fue particularmente complicado debido a la necesidad de usar componentes como `EdgeDFF` para sincronizar con el reloj, lo cual resultó en una sintaxis más densa y menos intuitiva que en `Verilog`. A pesar de ello, logré mantener la estructura y funcionalidad, verificando ambos diseños con sus respectivos tests.

Otro de los módulos que desarrollé fue un sistema para buscar el valor máximo en un vector de entradas. Comencé implementando una versión secuencial en `Verilog`, basada en una máquina de estados que recorría el vector (de tamaño fijo, por ejemplo 8 elementos), comparando cada valor con el máximo temporal almacenado y usando señales de habilitación para controlar las actualizaciones. Este enfoque permitió afianzar el manejo de lógica secuencial y el diseño de flujos de control compactos.

Una vez verificado el diseño, lo trasladé a `RustHDL`, replicando la FSM y adaptando la lógica a la sintaxis del nuevo entorno. Aunque resultó más exigente por el uso explícito de componentes como `EdgeDFF`, el control del flujo mediante estructuras del lenguaje `Rust` aportó claridad y modularidad. Preparé bancos de prueba para ambos entornos y verifiqué su funcionamiento con vectores de distintos tamaños y configuraciones. Las simulaciones confirmaron la correcta identificación del valor máximo y ofrecieron un buen punto de comparación entre ambos lenguajes en tareas de recorrido secuencial y toma de decisiones.

En la fase más avanzada del proyecto, planteé el desarrollo de un sistema de multiplicación de matrices, tanto en versión combinacional como secuencial. Este tipo de módulo suponía una carga computacional considerablemente mayor, por lo que resultaba ideal para evaluar las capacidades reales de síntesis y simulación de `RustHDL`. Diseñé primero la arquitectura en `Verilog`, implementando dos versiones diferenciadas: una que resolvía las operaciones en un solo ciclo combinacional, y otra que repartía el cálculo en varios ciclos mediante una máquina de estados. Una vez probadas y verificadas con `Icarus` y `Yosys`, trasladé ambos diseños a `RustHDL`. A lo largo de este proceso, desarrollé bancos de pruebas automatizados que permitían comprobar grandes rangos de datos de entrada, con el objetivo de acelerar

la verificación funcional y detectar errores de forma eficiente. Esta experiencia fue especialmente útil para explorar la expresividad de `RustHDL` en cálculos intensivos y operaciones sobre arrays, algo que resultó más sencillo de manejar que en `Verilog` gracias a la capacidad de abstracción del lenguaje.

Durante todas estas fases, también fui el encargado de capturar los resultados de simulación utilizando `GTKWave`, así como los informes de síntesis generados por `Yosys`. Clasifiqué y almacené estas imágenes para su posterior inclusión en la memoria, asegurando su trazabilidad y comprensión visual. Además, participé en la elaboración de documentos adicionales, como guías de uso, pequeños manuales y documentación complementaria alojada en el repositorio compartido.

Finalmente, me ocupé de estructurar y redactar buena parte de la memoria del proyecto. Esta tarea incluyó la organización de los contenidos, la explicación detallada de los módulos desarrollados, la integración de gráficos y capturas, y la comparación sistemática entre las versiones en `Verilog` y en `RustHDL`. Me aseguré de que el documento final fuera claro, completo y técnicamente riguroso, de modo que sirviera tanto como evidencia del trabajo realizado como guía futura para otros interesados en el uso de `RustHDL` en diseño digital.

En resumen, mi contribución ha sido transversal y constante a lo largo del proyecto. He intervenido tanto en la configuración técnica inicial como en la implementación de módulos funcionales complejos, colaborando puntualmente en la resolución de errores compartidos, y liderando el apartado documental. Todo ello me ha permitido adquirir experiencia valiosa en diseño digital, síntesis hardware y verificación, además de explorar las posibilidades reales de un lenguaje emergente como `RustHDL` frente a soluciones más consolidadas como `Verilog`.

## Christina Cabañés Sorensen

A lo largo del desarrollo del trabajo, mi participación fue igualmente clave en varias etapas del proyecto, si bien comenzó ligeramente más tarde debido a que me encontraba fuera del país durante el inicio. Una vez incorporada, me puse al día rápidamente descargando todas las herramientas necesarias y familiarizándome con el ecosistema de trabajo. Esto incluyó la instalación del compilador de `Rust`, la biblioteca `RustHDL`, el editor `Visual Studio Code`, así como los programas de síntesis y simulación `Yosys`, `GTKWave` e `Icarus Verilog`.

Durante este proceso inicial, también me encontré con varios problemas técnicos, en particular con la instalación de `Rust` y su entorno asociado. En mi caso, hubo ciertos conflictos que sospecho eran de compatibilidad, los cuales impedían ejecutar correctamente los comandos de compilación y testeo. Esto me obligó a realizar varias reinstalaciones y comparar directamente el comportamiento de mis programas con los de mi compañero hasta comprobar que el sistema funcionaba de forma correcta. Esta experiencia, aunque frustrante en algunos momentos, me ayudó a entender

mejor el entorno de desarrollo de `Rust`, y me preparó para afrontar los siguientes retos del proyecto.

Paralelamente a la configuración del entorno, dediqué tiempo a estudiar en profundidad los manuales de `Rust` y `RustHDL`. Dado que `Rust` era un lenguaje completamente nuevo para mí, necesitaba adquirir una base sólida antes de comenzar el desarrollo de código. A diferencia de `Rust`, en el lenguaje `Verilog` ya teníamos cierta soltura ya que cursamos una asignatura universitaria dedicada a ese lenguaje, por lo que no me resultaba necesario repasar sus fundamentos.

Con la base teórica asentada, me inicié en la parte técnica con un sumador combinacional con signo de dos entradas, tanto en `Verilog` como en `RustHDL`. Esta primera práctica me sirvió para poner en funcionamiento todas las herramientas y probar de forma concreta el flujo de trabajo entre codificación, simulación y verificación. Posteriormente, desarrollé los respectivos testbenches para simular el comportamiento del módulo en ambos lenguajes. Este paso representó un reto particularmente complejo para mí, ya que nunca había escrito un banco de pruebas desde cero y, además, la documentación existente sobre `RustHDL` era muy limitada. Me costó bastante tiempo conseguir una versión funcional que respetara la sintaxis del lenguaje y produjera resultados coherentes. No obstante, una vez superado este obstáculo, adquirí confianza en mi capacidad para trabajar con testbenches.

En la fase más avanzada del proyecto, mi contribución se centró en el desarrollo de un producto escalar secuencial, el cual supuso uno de los mayores desafíos a nivel técnico del proyecto. A diferencia de los módulos anteriores, que eran principalmente combinacionales, este módulo implicaba lógica secuencial, temporización, control de señales y sincronización mediante reloj. El principal reto vino derivado de la escasa documentación y recursos disponibles sobre cómo implementar circuitos secuenciales en `RustHDL`. Tuve que investigar extensamente para entender el comportamiento de los bloques secuenciales en este entorno, cómo definir correctamente los registros de estado y cómo manejar los flancos de subida del reloj.

El diseño del producto escalar se estructuraba en dos módulos: un módulo principal llamado `DotProduct`, encargado de la lógica de control general, y un submódulo multiplicador 8x8 que realizaba la multiplicación de dos señales de entrada. Este último componente fue especialmente problemático. Una de las limitaciones más importantes con las que me encontré fue la imposibilidad de realizar multiplicaciones de forma directa debido a las restricciones impuestas por el kernel de `RustHDL` (mediante la macro `hdlgen`). Inicialmente intenté implementar la multiplicación utilizando operaciones de desplazamiento binario, pero no logré obtener un comportamiento correcto ni control adecuado del desbordamiento.

Como alternativa, opté por una implementación basada en sumas parciales, que, aunque menos eficiente desde el punto de vista computacional, garantizaba un resultado correcto dentro de los límites del kernel y permitía una gestión explícita de los bits involucrados. Esta implementación implicó un control exhaustivo de los tipos de datos, ya que el sistema de tipos de `Rust` es especialmente estricto y presenta dificultades adicionales cuando se trata de señales binarias con signos o tamaños

variables. A pesar de la complejidad, con la ayuda de mi compañero, el módulo logró operar de forma correcta, y con ello obtuve un conocimiento mucho más profundo sobre cómo se construyen operaciones aritméticas en hardware utilizando un lenguaje como `Rust`.

Una vez finalizada la implementación, también me encargué de desarrollar el banco de pruebas correspondiente para verificar el comportamiento funcional del producto escalar. Esta tarea no fue sencilla, ya que el módulo requería múltiples ciclos de reloj para completar una operación y la sincronización debía respetarse de forma estricta. La creación de la simulación me permitió comprobar en tiempo real el funcionamiento interno del módulo, validar los resultados esperados y detectar posibles errores de sincronización o de propagación de señales.

Esta experiencia fue especialmente enriquecedora, ya que me permitió entender a fondo cómo se gestionan los diseños secuenciales en `RustHDL`, una habilidad que considero muy valiosa para futuros proyectos de diseño digital y para cualquier persona interesada en el desarrollo hardware con lenguajes modernos.

Finalmente, dentro de las tareas relacionadas con la elaboración de la memoria del proyecto, asumí la responsabilidad de redactar la introducción, el resumen, la bibliografía y las traducciones al inglés de los apartados que lo requerían. Además, me encargué de revisar y reorganizar el contenido global del documento, reescribiendo algunas secciones para mejorar su claridad y estructura. Durante este proceso, mantuve una comunicación constante con mi compañero para asegurar que ambos estuviéramos de acuerdo con los cambios realizados y que el resultado final fuera riguroso y profesional.

En resumen, mi contribución al proyecto ha sido técnica y documental. Aunque comencé mi participación algo más tarde, me involucré profundamente en el diseño, implementación y verificación de módulos clave, especialmente en lo relativo a circuitos secuenciales complejos. He aprendido a desenvolverme con herramientas nuevas como `RustHDL` en un entorno poco documentado, enfrentando retos reales de desarrollo y depuración. A su vez, he contribuido significativamente a la elaboración de una memoria clara y estructurada, que espero pueda servir como referencia útil para futuros estudiantes o investigadores interesados en las posibilidades de `Rust` en el ámbito del diseño digital.

# Bibliografía


Cadence Design Systems. Cadence electronic design automation tools. https://www.cadence.com/, 2024.

Cong, J., Fan, Y., Han, G., Jiang, W. y Zhang, Z. Platform-based behavior-level and system-level synthesis. En *2006 IEEE International SOC Conference*, páginas 199–202. Austin, TX, USA, 2006.

Derek Lockhart, G. Z. y Batten, C. 47th acm/ieee int'l symp. on microarchitecture (micro-47). En *PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research.*. 2014.

Dinechin, F. D. y Pasca, B. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, vol. 28(4), páginas 18–27, 2011.

Intel Corporation. *Intel High Level Synthesis (HLS) Compiler*, 2024. Disponible en: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

Murillo, R., Barrio, A. A. D. y Botella, G. A suite of division algorithms for posit arithmetic. En *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, páginas 41–44. 2023a.

Murillo, R., Barrio, A. A. D., Botella, G. y Pilato, C. Generating posit-based accelerators with high-level synthesis. *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70(10), páginas 4040–4052, 2023b.

Murillo, R., Hormigo, J., Barrio, A. A. D. y Botella, G. Hub meets posit: Arithmetic units implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71(1), páginas 440–444, 2024.

Pilato, C. y Ferrandi, F. Bambu: A modular framework for the high level synthesis of memory-intensive applications. En *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications (FPL)*, páginas 1–4. 2013.

RustHDL Contributors. Rusthdl - a framework for hardware design in rust. https://rust-hdl.org/, 2024a.

RustHDL Contributors. Rusthdl documentation - api reference. https://docs.rs/rust-hdl/latest/rust_hdl/, 2024b.

Shunning Jiang, B. I. y Batten, C. 55th acm/ieee design automation conf. (dac-55). En *Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks*. 2018a.

Shunning Jiang, C. T. y Batten, C. First workshop on open-source eda technology (woset'18) held in conjunction with iccad-37. En *An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework.*. 2018b.

Synopsys. Synopsys design tools. https://www.synopsys.com/, 2024.

Synopsys Inc. *Synplify HLS Tool*, 2024.

Systems, C. D. Cadence stratus high-level synthesis. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, ????

Xilinx. Vitis unified software platform. https://www.xilinx.com/products/design-tools/vitis.html, 2024a.

Xilinx. Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html, 2024b.