

# **Diseño e implementación de un Microprocesador RISC-V RV32I con arquitectura Harvard**

v1.0.0 — Diciembre 2025

Erika Garcia, Sebastián Ramírez, Manuel Rivera, Israel Frausto, Ramón Estrada, Rodrigo Mata, Jaime Navarro, Sabina Bañuelos

Diseño Digital I

# Índice

<b>Index</b>	<b>3</b>
<b>1. Introducción</b>	<b>5</b>
<b>2. Marco Teórico</b>	<b>6</b>
2.1. RISC-V . . . . .	6
2.2. Características generales . . . . .	6
2.3. Formato de instrucciones . . . . .	6
2.4. Unidad de control . . . . .	8
2.4.1. Funciones de la unidad de control . . . . .	8
2.4.2. Requisitos de la arquitectura . . . . .	9
2.5. Banco de registros . . . . .	9
2.5.1. El Banco de Registros en RISC-V . . . . .	9
2.6. ALU . . . . .	10
2.6.1. Operaciones disponibles . . . . .	10
2.7. Program Counter . . . . .	10
2.8. Memorias . . . . .	11
2.8.1. Arquitectura de Memoria (Harvard Modificada) . . . . .	11
2.8.2. Organización y Direcccionamiento (Addressing) de memoria . . . . .	11
2.8.3. Técnicas de Implementación en FPGA . . . . .	11
2.8.4. Especificaciones de los Módulos Diseñados . . . . .	12
2.9. UART . . . . .	13
<b>3. Estado del Arte</b>	<b>14</b>
<b>4. Desarrollo</b>	<b>15</b>
4.1. Microarquitectura general . . . . .	15
4.2. Unidad de control . . . . .	15
4.2.1. Entradas y Salidas . . . . .	15
4.2.2. Fragmento de RTL de la CU . . . . .	16
4.3. Banco de registros . . . . .	18
4.3.1. Optimización de Recursos (Virtualización de x0) . . . . .	18
4.3.2. Lógica de Lectura Combinacional (Bypass de x0) . . . . .	18
4.3.3. Escritura Síncrona Protegida . . . . .	18
4.4. ALU . . . . .	19

4.4.1. Entradas y Salidas . . . . .	19
4.4.2. Fragmento de RTL de la ALU . . . . .	19
4.5. Extensión de signo . . . . .	20
4.5.1. Descripción General . . . . .	20
4.5.2. Entradas y Salidas . . . . .	20
4.5.3. RTL . . . . .	20
4.6. Program Counter (PC) . . . . .	22
4.6.1. Tamaño e incrementos del PC . . . . .	22
4.6.2. Actualizaciones del PC . . . . .	22
4.6.3. Secciones del PC . . . . .	22
4.7. Memoria . . . . .	23
4.7.1. Implementación de la Memoria de Instrucciones (Instruction Memory) . . . . .	23
4.7.2. Implementación de la Memoria de Datos (Data Memory) . . . . .	24
4.7.3. Parametrización del Diseño . . . . .	25
4.8. UART . . . . .	26
4.8.1. Descripción General . . . . .	26
4.8.2. Especificaciones . . . . .	26
4.8.3. Entradas y Salidas . . . . .	27
4.8.4. Módulo baud_rate_generator . . . . .	27
4.8.5. Módulo receiver . . . . .	27
4.8.6. Módulo shift_register_fsm . . . . .	28
4.8.7. Módulo transmitter . . . . .	29
4.8.8. Comunicación con Python . . . . .	30
<b>5. Simulación y verificación</b>	<b>32</b>
5.1. Verificación . . . . .	32
5.1.1. Testbench . . . . .	32
5.1.2. Generación de instrucciones. . . . .	32
5.1.3. Formatos de instrucción . . . . .	32
5.1.4. Ejecución aleatoria de instrucciones . . . . .	33
5.1.5. Monitoreo de entradas y salidas de los módulos internos. . . . .	34
5.2. Simulación . . . . .	36
<b>6. Resultados</b>	<b>37</b>
<b>7. Conclusiones</b>	<b>39</b>

<b>Apéndice A:</b> .....	<b>41</b>
<b>Apéndice B:</b> .....	<b>46</b>
<b>Apéndice C:</b> .....	<b>47</b>
<b>Apéndice D:</b> .....	<b>49</b>
<b>Apéndice E:</b> .....	<b>50</b>
<b>Apéndice F:</b> .....	<b>52</b>
<b>Apéndice G:</b> .....	<b>53</b>
<b>Apéndice H:</b> .....	<b>61</b>

## Índice de figuras

1. Formato de instrucción I-type. . . . .	7
2. Formato de instrucción S-type. . . . .	7
3. Formato de instrucción B-type. . . . .	7
4. Formato de instrucción U-type. . . . .	8
5. Formato de instrucción J-type. . . . .	8
6. Microarquitectura del microprocesador RISC-V para procesamiento de serie fibonacci . . . . .	15
7. Señales de entradas y salidas . . . . .	34
8. Resultados de verificación . . . . .	35
9. Enfoque de error en los resultados de la verificación . . . . .	35
10. Resultados de Simulación en 10 iteraciones . . . . .	36
11. Envio de bytes por protocolo UART a la FPGA . . . . .	38
12. Cargado del programa Fibonacci en la FPGA por protocolo UART. . . . .	38
13. Funcionamiento en el FPGA . . . . .	38

## Índice de cuadros

1. Tabla de especificación de entradas y salidas de la CU . . . . .	16
2. Tabla de señales de salida para la instrucción <b>addi</b> . . . . .	16
3. Tabla de especificación de entradas y salidas de la ALU . . . . .	19
4. Tabla de especificación de entradas y salidas del extensor de signos . . . . .	20

5.	Codificación de imm_sel . . . . .	20
6.	Especificaciones globales del módulo UART. . . . .	26
7.	Entradas y salidas del módulo uart_top. . . . .	27
8.	Entradas y salidas del módulo Baud Rate Generator . . . . .	27
9.	Entradas y salidas del módulo receiver. . . . .	28
10.	Entradas y salidas del módulo shift_register_fsm. . . . .	29
11.	Entradas y salidas del módulo transmitter . . . . .	30
12.	Formato ADD . . . . .	32
13.	Formato ADDI . . . . .	32
14.	Formato BEQ . . . . .	33
15.	Formato JAL . . . . .	33

## 1. Introducción

En el panorama actual de la ingeniería electrónica, la arquitectura RISC-V se ha establecido como un estándar fundamental debido a su naturaleza de código abierto y modularidad, lo que permite el diseño de hardware personalizado sin las restricciones de licencias propietarias. Este proyecto surge de la necesidad de trasladar los conceptos teóricos de la arquitectura de computadoras a una implementación física verificable, utilizando SystemVerilog para el diseño de un núcleo de procesamiento funcional en FPGA. El sistema desarrollado consiste en un microprocesador de ciclo único (Single-Cycle) basado en la ISA RV32I de RISC-V, diseñado bajo una arquitectura Harvard Modificada que separa físicamente las memorias de instrucciones y de datos para optimizar el acceso a la información. La función principal del sistema es la ejecución del algoritmo de la serie de Fibonacci , una tarea que permite validar integralmente la capacidad de la Unidad Aritmético-Lógica (ALU), la gestión de saltos de la Unidad de Control y el almacenamiento temporal en el Banco de Registros.

Con el objetivo específico de dotar al sistema de flexibilidad y autonomía, el diseño resuelve el problema de la carga estática de programas mediante la integración de un módulo de comunicación serial UART que actúa como cargador dinámico (bootloader). Esta implementación aborda retos técnicos críticos como la coherencia de datos (Endianness) y la discrepancia de ancho de banda entre la transmisión serial de 8 bits y el procesamiento de 32 bits, solucionado mediante una arquitectura de memoria de instrucciones con bancos intercalados (Interleaved Memory). Esta estrategia permite la reconstrucción eficiente de instrucciones en tiempo real sin requerir la re-síntesis del hardware para ejecutar diferentes programas.

El alcance del proyecto comprende el diseño a nivel de transferencia de registros (RTL), la simulación y la implementación física de las instrucciones base tipo R, I, S, B, U y J necesarias para aritmética entera y control de flujo. Sin embargo, el diseño presenta limitaciones inherentes a su arquitectura y propósito educativo: al operar como un procesador monociclo, la frecuencia máxima de operación está restringida a 50 MHz por la ruta crítica de propagación, a diferencia de arquitecturas segmentadas (pipeline) de mayor rendimiento. Asimismo, el sistema se limita al conjunto de instrucciones enteras de 32 bits, excluyendo extensiones de punto flotante, multiplicación por hardware o modos privilegiados requeridos para sistemas operativos complejos.

## 2. Marco Teórico

### 2.1. RISC-V

RISC-V es una Arquitectura de Conjunto de Instrucciones (Instruction Set Architecture, ISA) abierta y estandarizada: es una especificación formal del lenguaje máquina que utilizan software y hardware para comunicarse. La ISA define el archivo de registros, los formatos numéricos, el conjunto de instrucciones y sus codificaciones, las convenciones de llamada y los niveles de privilegio, así como una forma estándar de ampliar o añadir grupos de instrucciones opcionales.

Las especificaciones de RISC-V otorga un estandar con el que cualquiera puede diseñar un microprocesador, núcleo de CPU, SoC o acelerador que implemente algún subconjunto de la ISA. Las ISA de RISC-V básicas más relevantes para los diseñadores son RV32I (32 bits), RV64I (64 bits) y RV128I (128 bits). Las implementaciones de RISC-V se amplían comúnmente con extensiones opcionales y estandarizadas como M (multiplicación/división de enteros), A (atómicas), F/D (coma flotante), y C (codificaciones comprimidas de 16 bits). Los documentos formales ratificados que describen estas extensiones están disponibles públicamente en RISC-V International [1].

### 2.2. Características generales

La ISA RISC-V se define como una ISA entera básica, que debe estar presente en cualquier implementación, además de extensiones opcionales a la ISA básica. La ISA entera básica es muy similar a la de los primeros procesadores RISC-V. La base se limita cuidadosamente a un conjunto mínimo de instrucciones suficientes para proporcionar un objetivo razonable para compiladores, ensambladores, enlazadores y sistemas operativos, por lo que proporciona una ISA conveniente y sólida.

Cada conjunto de instrucciones enteras básicas se caracteriza por al anchura del registro y el tamaño correspondiente del espacio de direcciones del usuario. Hay dos variantes enteras básicas, previamente mencionadas, RV32I y RV64I, que proporcionan espacios de direcciones de nivel de usuario de 32 bits o 64 bits, respectivamente. Las implementaciones de hardware y los sistemas operativos pueden proporcionar solo una o ambas para los programas de usuario. Esta ISA de enteros base se denomina *Iz* contiene instrucciones de cálculo de enteros, cargas de enteros, almacenamientos de enteros e instrucciones de flujo de control, y es obligatoria para todas las implementaciones de RISC-V [1].

RISC-V ha sido diseñado para admitir una amplia personalización y especialización. La ISA entera básica se puede ampliar con una o más extensiones opcionales del conjunto de instrucciones, pero las instrucciones enteras básicas no se pueden redefinir. Las extensiones estándar deben ser útiles en general y no deben entrar en conflicto con otras extensiones estándar. Las extensiones no estándar pueden ser muy especializadas o pueden entrar en conflicto con otras extensiones estándar o no estándar. Para dar soporte a un desarrollo de software más general, se han definido un conjunto de extensiones estándar que proporcionan operaciones de multiplicación/división de enteros, operaciones atómicas, y aritmética de coma flotante de precisión simple y doble.

### 2.3. Formato de instrucciones

La arquitectura RISC-V de 32 bits utiliza instrucciones de longitud fija de 32 bits organizadas en diferentes formatos. Todos los formatos tienen una estructura base, pero cada formato tiene distribución específica de campo. Las instrucciones utilizadas en este proyecto son: R-type, I-type, S-type, B-type, U-type y J-type[1].

Todas las instrucciones RISC-V de 32 bits están conformadas por los siguientes campos básicos:

- **opcode (bits 6–0)**: Determina la categoría general de la instrucción.
- **rd (bits 11–7)**: Registro destino.
- **funct3 (bits 14–12)**: Selecciona la operación dentro del opcode.
- **rs1 (bits 19–15)**: Primer registro fuente.
- **rs2 (bits 24–20)**: Segundo registro fuente.
- **funct7 (bits 31–25)**: Extiende la codificación para operaciones aritméticas y lógicas.

Algunos formatos reemplazan ciertos campos con partes de un inmediato. A continuación se describen cada uno de estos formatos.

La instrucción I-type contiene un inmediato de 12 bits ubicado en `instr[31:20]` y es utilizado por instrucciones como ADDI.

imm[11:0]	rs1	funct3	rd	opcode
31:20	19:15	14:12	11:7	6:0

Figura 1: Formato de instrucción I-type.

El inmediato del formato S-type está dividido en dos partes dentro de la instrucción:

- `instr[31:25]` corresponde a `imm[11:5]`
- `instr[11:7]` corresponde a `imm[4:0]`

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
31:25	24:20	19:15	14:12	11:7	6:0

Figura 2: Formato de instrucción S-type.

El inmediato B-type está distribuido y reordenado de la siguiente manera:

- `imm[12] = instr[31]`
- `imm[11] = instr[7]`
- `imm[10:5] = instr[30:25]`
- `imm[4:1] = instr[11:8]`
- `imm[0] = 0` (alineación por palabra)

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
31	30:25	24:20	19:15	14:12	11:8	7	6:0

Figura 3: Formato de instrucción B-type.

El inmediato U-type contiene 20 bits:

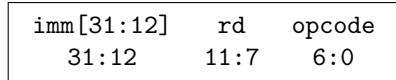


Figura 4: Formato de instrucción U-type.

El inmediato J-type está reordenado en varias secciones:

- imm[20] = instr[31]
- imm[19:12] = instr[19:12]
- imm[11] = instr[20]
- imm[10:1] = instr[30:21]
- imm[0] = 0

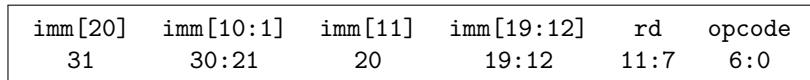


Figura 5: Formato de instrucción J-type.

## 2.4. Unidad de control

La unidad de control (CU) es el componente que interpreta la palabra de instrucción obtenida de la memoria y coordina todas las operaciones internas y rutea los datos necesarios para ejecutar dichas instrucciones. La CU transforma la codificación de la instrucción en una secuencia coordinada de señales de control que hacen que la ruta de datos se comporte correctamente para esa instrucción.

### 2.4.1. Funciones de la unidad de control

- **Decodificar** los códigos operacionales y los campos de función de la instrucción
- **Generar señales de control** que determinan:
  1. Operaciones de la unidad aritmética lógica.
  2. Selección de inmediatos y extensión de cero/sígno.
  3. Habilitar la lectura y escritura de memoria.
  4. Seleccionar la siguiente dirección del puntero de programa.
- **Sincronizar** todas las subunidades para que la ruta de datos ejecute en el orden correcto las instrucciones.

### 2.4.2. Requisitos de la arquitectura

La instrucción RV32I utiliza una codificación fija de 32-bits. La CU identifica la instrucción usando

- **opcode** [6:0]
- **funct3** [14:12]
- **funct7** [31:25]

Estos campos determinan el formato de la instrucción y las acciones que las señales de salida de la CU deben realizar. En un sólo ciclo de la ruta de datos, la CU produce las señales:

- **alu\_op**: selecciona la operación que la ALU realizará
- **prf\_wr\_en**: habilita la escritura del banco de registros
- **imm\_sel**: selector de valores inmediatos
- **branch\_taken**: control de un MUX conectado al PC para operaciones de salto.
- **data\_mem\_wr\_en**: habilita la escritura en la memoria de datos

Entre otras señales. Las señales que orquesta la CU están determinadas por las ISA soportadas por la microarquitectura.

## 2.5. Banco de registros

### 2.5.1. El Banco de Registros en RISC-V

El Banco de Registros (Register File) constituye el nivel más alto y veloz en la jerarquía de memoria de un procesador. En la arquitectura RISC-V (específicamente la variante RV32I), se define un conjunto de 32 registros de propósito general de 32 bits de ancho, denominados x0 a x31. Estos registros almacenan los operandos inmediatos para la Unidad Lógica Aritmética (ALU) y los resultados de las operaciones.

#### Requisitos de la Arquitectura

El estándar RISC-V impone dos restricciones críticas de diseño para este módulo:

- El Registro Cero (x0): Se define arquitecturalmente como una constante cableada a tierra. Cualquier lectura al registro x0 debe retornar siempre el valor 0, y cualquier escritura en él debe ser ignorada o descartada, sin alterar el estado del procesador.
- Acceso Multi-Puerto: La mayoría de las instrucciones tipo-R requieren leer dos operandos fuente (rs1, rs2) y escribir un resultado (rd) en el mismo ciclo de instrucción. Por lo tanto, el hardware debe soportar una topología 2R1W (2 Puertos de Lectura, 1 Puerto de Escritura).

#### Direccionamiento y Decodificación

El acceso a los registros se realiza mediante direcciones de 5 bits, lo que permite seleccionar uno de los  $2^5 = 32$  registros disponibles. Estas direcciones provienen directamente de campos específicos en la palabra de instrucción de 32 bits:

- rs1 (Source 1): Bits de la instrucción.
- rs2 (Source 2): Bits de la instrucción.
- rd (Destination): Bits [11:7] de la instrucción.

El banco de registros actúa como el puente entre la decodificación de la instrucción y la ejecución aritmética, suministrando los datos almacenados en rs1 y rs2 a la ALU y capturando el resultado en rd al finalizar el ciclo.

## 2.6. ALU

La unidad aritmética y lógica (ALU) es unidad computacional central del microprocesador. Es el subsistema responsable de realizar todas las operaciones aritméticas y lógicas requeridas por el conjunto de instrucciones. En un procesador RISC-V RV32I, la ALU es fundamental para la ejecución de instrucciones enteras, cálculos de direcciones, comparaciones y decisiones de flujo de control. En pocas palabras, la ALU incorpora las primitivas lógicas de la ISA.

### 2.6.1. Operaciones disponibles

La ALU para la ISA de enteros RV32I debe admitir las siguientes operaciones:

- Aritméticas: Las operaciones aritméticas básicas de la ISA de enteros comprenden suma y resta (ADD, SUB), además de la suma inmediata (ADDI) con una constante generada por el generador de inmediatos.
- Lógicas: Las operaciones lógicas comprenden la conjunción y disyunción (AND, OR), la disyunción exclusiva (XOR), y sus variantes inmediatas (ANDI, ORI, XORI).
- Desplazamiento: Las operaciones de desplazamiento comprenden el desplazamiento lógico hacia la izquierda (SLL), desplazamiento lógico hacia la derecha (SRL), desplazamiento aritmético hacia la derecha (SRA), y las variantes con inmediatos (SLLI,SRLI,SRAI).
- Comparaciones: Las ALU también se encarga de realizar comparaciones para la decisión de saltos condicionales e incondicionales. Estas operaciones incluyen igualdad (BEQ, BNE), menor que signado (BLT, BGE), y menor que sin signo (BLTU, BGEU).

## 2.7. Program Counter

El contador de programa (Program Counter, PC) determina cual instrucción el procesador debe recuperar y ejecutará a continuación, determinando la secuencia de ejecución del programa.

El PC es un registro dedicado que contiene la dirección de byte de la siguiente instrucción que se va a recuperar de la memoria. En RV32I, este valor es siempre de 32 bits, alineado con los límites de las instrucciones de 32 bits (normalmente incrementado en 4). Durante cada ciclo de la etapa de obtención de instrucciones del procesador, el PC proporciona una dirección a la memoria de instrucciones, y la palabra devuelta es decodificada y ejecutada por la canalización o la ruta de datos de ciclo único.

## 2.8. Memorias

### 2.8.1. Arquitectura de Memoria (Harvard Modificada)

El diseño del microprocesador implementado se basa en una arquitectura Harvard, la cual se caracteriza por tener espacios de almacenamiento físicamente separados para las Instrucciones (Código) y los Datos (Variables).

- Instruction Memory (IMEM): Almacena el programa a ejecutar. En este diseño, aunque funciona lógicamente como una ROM (Read-Only Memory) durante la ejecución del procesador, se implementa físicamente como una RAM para permitir la carga dinámica de programas mediante un cargador externo (Bootloader via UART).
- Data Memory (DMEM): Almacena los resultados temporales y variables del programa. Permite operaciones de lectura y escritura en tiempo de ejecución.

### 2.8.2. Organización y Direccionamiento (Addressing) de memoria

#### Byte Addressing vs. Word Addressing

La arquitectura RISC-V utiliza un esquema de direccionamiento por bytes (Byte-Addressable). Esto significa que cada dirección de memoria (0, 1, 2...) apunta a un bloque de 8 bits. Sin embargo, las instrucciones estándar de RISC-V (RV32I) tienen una longitud de 32 bits (1 Word). Esto implica que:

1. Cada instrucción ocupa 4 direcciones de memoria consecutivas.
2. El Contador de Programa (PC) debe incrementarse en pasos de 4 bytes ( $PC \leftarrow PC + 4$ ).
3. Para mapear una dirección de byte (proporcionada por el CPU) a un índice físico de un arreglo de palabras en Verilog, se deben descartar los dos bits menos significativos (*LSB*), lo que equivale a una división entera por 4:

$$\text{Índice físico} = \text{Dirección CPU} \gg 2$$

#### Endianness (Little-Endian)

RISC-V es una arquitectura Little-Endian. Esto dicta que el byte menos significativo (LSB) de una palabra multibyte se almacena en la dirección de memoria más baja. Dado que el protocolo de comunicación serial (UART) recibe los datos en orden secuencial (y el módulo de recepción los empaqueta en formato Big-Endian por defecto), es necesario implementar una lógica de "Byte Swapping."º reordenamiento de bytes en la etapa de escritura de la memoria para garantizar la coherencia de los datos al ser leídos por el procesador.

### 2.8.3. Técnicas de Implementación en FPGA

#### Memoria Bancarizada (Memory Banking)

Para la Memoria de Instrucciones, se implementó una arquitectura de bancos intercalados (Interleaved Memory) para resolver la discrepancia entre el ancho de banda de escritura y el de lectura.

- Problema: La memoria se carga byte a byte (o en paquetes desalineados) pero debe leerse en palabras completas de 32 bits simultáneamente.

- Solución: Se divide el espacio de direcciones en 4 bancos independientes de 8 bits de ancho cada uno (bank0 a bank3). Esto permite acceder a 4 direcciones de byte simultáneamente en un solo ciclo de reloj, aumentando el throughput de lectura y permitiendo la reconstrucción de la instrucción de 32 bits en lógica combinacional.

### **RAM Distribuida vs. Block RAM**

En la síntesis de FPGAs (como en Vivado), las memorias pueden inferirse de dos formas:

- Distributed RAM: Utiliza las LUTs (Look-Up Tables) de la FPGA. Es ideal para memorias pequeñas y rápidas con lectura asíncrona (combinacional).
- Block RAM (BRAM): Bloques de memoria dedicados. Son síncronos (requieren reloj para lectura).

Dado que se diseñó un procesador de Ciclo Único (Single Cycle), la lectura de las instrucciones y datos debe ocurrir dentro del mismo ciclo de reloj en que se calcula la dirección. Por lo tanto, se optó por una inferencia de memoria con Lectura Asíncrona, lo que generalmente resulta en el uso de RAM Distribuida (LUTRAM) para garantizar que los datos estén disponibles inmediatamente para la Unidad de Control y la ALU.

#### **2.8.4. Especificaciones de los Módulos Diseñados**

##### **Módulo Instruction Memory**

Este módulo actúa como una interfaz de doble puerto asimétrica:

- Capacidad: 1024 Bytes (1 KB), capaz de almacenar hasta 256 instrucciones.
- Puerto de Escritura (Carga): Síncrono. Recibe datos de 32 bits desde el módulo receptor UART/Shift Register. Implementa lógica de acomodo de bytes para cumplir con el formato Little-Endian.
- Puerto de Lectura (Ejecución): Combinacional. Recibe una dirección de 32 bits del PC, la trunca a 10 bits ([9:0]) para direccionamiento local, y concatena los datos de los 4 bancos internos para entregar una instrucción válida de 32 bits.

##### **Módulo Data Memory**

Diseñada como una RAM estándar de lectura/escritura:

- Capacidad: 4 KB (1024 Palabras de 32 bits).
- Organización: 32 bits de ancho por 1024 de profundidad.
- Funcionamiento: Utiliza la señal Write Enable (WE) generada por la Unidad de Control para almacenar resultados de la ALU. La lectura es combinacional directa, permitiendo el acceso inmediato a los operandos para instrucciones de carga (LW).

## 2.9. UART

La **Universal Asynchronous Receiver/Transmitter (UART)** es un protocolo de comunicación serial asíncrona que permite la transferencia bidireccional de datos entre dispositivos sin requerir una señal de reloj compartida entre el transmisor (TX) y el receptor (RX). La sincronización se establece mediante la tasa de baudios (baud rate) configurada previamente y el uso de bits de control incorporados en cada trama de datos. La UART transforma datos paralelos generados en un flujo serial para su transmisión y, de manera recíproca, reconstruye los datos paralelos a partir del flujo recibido [2].

La unidad básica de comunicación en UART es la trama de datos. Cada trama incluye cuatro componentes: un bit de inicio, bits de datos, un bit de paridad (opcional) y bits de parada. Esta estructura permite que el receptor identifique correctamente el comienzo y fin de cada unidad de información [3].

El bit de inicio tiene un nivel lógico bajo (0) y marca el inicio de la transmisión. Los bits de datos siguen al bit de inicio y pueden configurarse entre 5 y 9 bits, siendo 8 bits la configuración más común en sistemas modernos. Los bits de parada (1 o 2 bits en nivel alto) indican el término de la trama y aseguran un intervalo mínimo entre transmisiones consecutivas.

La velocidad de transmisión en UART se define mediante el baud rate , que indica el número de bits transmitidos por segundo [2]. El transmisor y receptor deben coincidir en este valor para asegurar la correcta comunicación. Valores comunes incluyen 9600, 57600, 115200 y 230400 bps [3].

La arquitectura del UART se estructura en tres bloques fundamentales:

- **Transmisor:** Convierte los datos paralelos en un flujo serial según el protocolo establecido.
- **Receptor:** Realiza la función inversa, convirtiendo el flujo serial en datos paralelos.
- **Generador de baud rate (BRG):** Produce pulsos de temporización precisos que determinan la duración de cada bit transmitido o recibido. Se emplea sobremuestreo (típicamente 16×) para robustez en la detección de bits. El muestreo ideal se realiza en el punto medio de cada bit, por ejemplo en el tick 7 u 8 de un ciclo de 16 muestras, garantizando una lectura precisa [2].

La relación entre la frecuencia del sistema y el baud rate se establece mediante el BRG, usando la fórmula:

$$\text{Divisor} = \frac{F_{\text{clk}}}{\text{Baud Rate} \times \text{Factor de Sobremuestreo}}$$

donde  $F_{\text{clk}}$  es la frecuencia del reloj del sistema y el factor de sobremuestreo depende de la configuración del BRG.

### **3. Estado del Arte**

Diversos estudios han explorado el diseño y la implementación de procesadores RISC-V adaptados a distintos objetivos. Por ejemplo, el trabajo de Toker (2023) propone un System on Chip basado en RISC-V RV32I usando High Level Synthesis (HLS) en C++, generando código Verilog para FPGA de bajo costo, como Basys3. El diseño se prueba con programas en ensamblador y C compilados con GNU RISC-V, y se evalúan recursos utilizados, consumo de potencia y tiempos de ejecución[4].

Rajyan y Saini (2024) presentan el diseño e implementación de un núcleo de procesador RISC-V RV32I utilizando SystemVerilog, con un enfoque modular y extensible. El trabajo integra componentes clave como Instruction Fetch, Instruction Decode, Register File, ALU, Data Memory, Control Unit y Branch Control Unit. La arquitectura propuesta permite la ejecución del conjunto de instrucciones RISC-V con alta confiabilidad y eficiencia, y proporciona una base para futuras mejoras, incluyendo la incorporación de instrucciones complejas y operaciones de múltiples ciclos[5].

Anjana et al. (2025) implementan un procesador RISC-V de 32 bits monociclo con Verilog HDL, enfocados a plataformas FPGA para aplicaciones educativas y de sistemas embebidos. La arquitectura incluye las etapas de ejecución de instrucciones y un enfoque de cinco etapas de pipeline para mejorar el rendimiento. Se implementaron módulos como el program counter, memoria de instrucciones, register file, ALU, memoria de datos y unidad de detección de hazards (peligros que pueden causar retrasos o conflictos en el pipeline debido a dependencias de datos o control). La síntesis y verificación se realizaron en Xilinx Spartan-6 y Spartan-3E, alcanzando una frecuencia máxima de 32 MHz y un consumo estimado de 7.9 mW[6].

Por otra parte, Shahriar y Harun-ur-Rashid (2025) presentan un procesador RISC-V RV32I de cinco etapas de pipeline implementado en SystemVerilog, basado en arquitectura Harvard. Esta elección permite separar físicamente la memoria de instrucciones de la memoria de datos, facilitando accesos simultáneos y aumentando el rendimiento del pipeline. El diseño está orientado a FPGAs pequeñas como la DE2-115 y se verifica tanto a nivel de hardware como en entornos EDA en línea[7] .

## 4. Desarrollo

### 4.1. Microarquitectura general

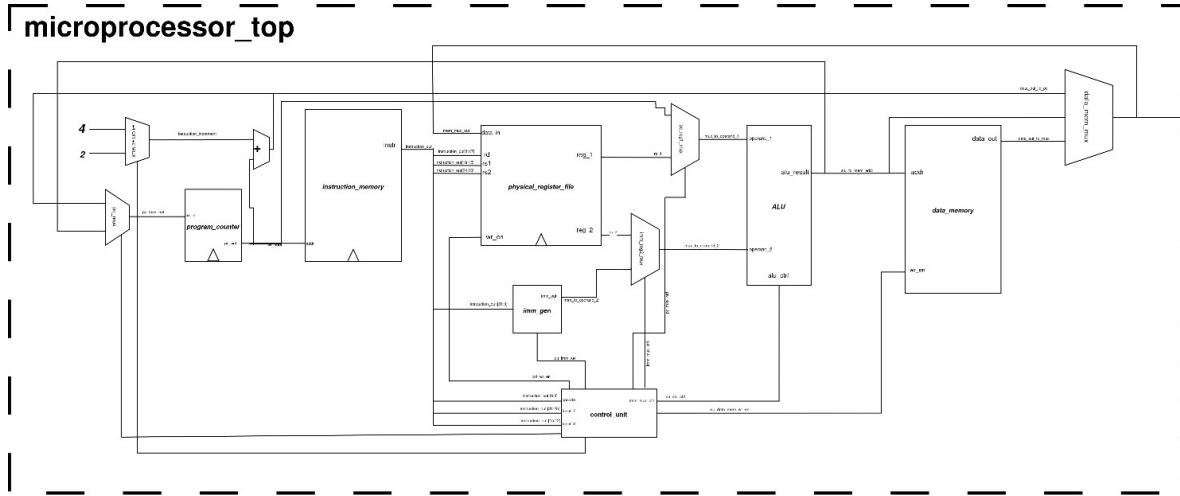


Figura 6: Microarquitectura del microprocesador RISC-V para procesamiento de serie fibonacci

### 4.2. Unidad de control

La Unidad de Control es la parte encargada de la toma de decisiones y la coordinación de la ruta de datos. Su función principal es dirigir el tráfico de datos y asegurar que todas las partes del procesador sepan qué hacer y cuándo hacerlo. Sus funciones incluyen:

- **Interpretar instrucciones:** Lee las instrucciones que provienen de la memoria (el código de un programa).
- **Decodificar:** Traduce esas instrucciones de código binario a señales específicas que el hardware pueda entender.
- **Secuenciar:** Asegura que las tareas se realicen en el orden correcto y sincronizadas con el reloj del sistema.
- **Generar señales de control:** Envía órdenes a otros componentes.

#### 4.2.1. Entradas y Salidas

La CU que cuenta con conexiones directas o indirectas (a través de multiplexores) con el resto de módulos del procesador.

Señal	Dirección	Ancho (bits)	Descripción
opcode	Entrada	7	Tipo de instrucción [6:0]
funct_3	Entrada	3	Instrucción [14:12]
funct_7	Entrada	7	Instrucción [31:25]
zero	Entrada	1	Resultado de una comparación cuando los operandos son iguales
prf_wr_en	Salida	1	Habilitar escritura en el PRF
cu_imm_sel	Salida	3	Selector de inmediatos
prf_pc_mux_ctrl	Salida	1	Selector para el MUX de PC
prf_imm_mux_ctrl	Salida	1	Selector para el MUX del PRF hacia el imm_gen
cu_alu_ctrl	Salida	4	Selector de operaciones para la ALU
cu_mem_out_mux_sel	Salida	2	Selector para la dirección de Memoria, ALU y PC
cu_data_mem_wr_en	Salida	1	Habilita la escritura en la memoria de datos
cu_pc_add_sel	Salida	1	Selecciona el aumento del PC cuando ocurre un salto
branch_taken	Salida	1	Evaluación cuando una instrucción de salto ocurrió

Cuadro 1: Tabla de especificación de entradas y salidas de la CU

Cuando la CU recibe un código de operación y las secciones funct\_3 y funct\_7 de una instrucción, esta decodifica la información recibida y coordina las señales de operación de todos los módulos del procesador. Por ejemplo, cuando la CU recibe una instrucción de suma del tipo I, la CU le indica a la ALU que debe tomar de operandos un registro y un valor inmediato, deshabilita la escritura en memoria y le indica al contador de programa que no hubo un salto de instrucción. Las señales se muestran en la siguiente tabla.

Señal	valor	Descripción
prf_wr_en	[1]	Habilita la lectura del banco de registros
cu_imm_sel	[000]	Le indica el IG que necesita un inmediato para instrucción I
prf_pc_mux_ctrl	[0]	Le indica a la ALU que debe tomar un valor de un registro
prf_imm_mux_ctrl	[1]	Selecciona un valor inmediato en lugar de leer un registro
cu_alu_ctrl	[000]	Operación de suma en la ALU
cu_mem_out_mux_sel	[00]	Envía el resultado de la ALU al banco de registros
cu_data_mem_wr_en	[0]	Deshabilita la escritura en la memoria de datos
cu_pc_add_sel	[0]	Le indica al PC que no debe aumentar la dirección actual de su puntero
branch_taken	[0]	Evaluación cuando una instrucción de salto ocurrió

Cuadro 2: Tabla de señales de salida para la instrucción **addi**

#### 4.2.2. Fragmento de RTL de la CU

```

1 module control_unit(
2   input logic [6:0] opcode,           //instructions [6:0]
3   input logic [2:0] funct_3,         //instructions [14:12]
4   input logic [6:0] funct_7,         //instructions [31:25]
5   ...
6 );
7 //Define the instructions
8 `include "defines.svh"
9 always_comb begin
10   prf_wr_en = 1'b1;                //enable prf write
11   cu_data_mem_wr_en = 1'b0;        //disable data_mem write
12   cu_imm_sel = IMM_I;             //use IMM_I_TYPE
13   prf_pc_mux_ctrl = 1'b0;         //use rs1 as opread 1 in ALU
14   prf_imm_mux_ctrl = 1'b1;         //don't use rs2 instead use imm_gen
15   cu_pc_add_sel = 1'b0;            //Add +4 to pc_in

```

```

16    cu_mem_out_mux_sel =      ALU_TO_PRF;      //send the result of ALU to prf data_in
17    cu_alu_ctrl =            ALU_ADD;        //ADD operation in ALU
18    branch_taken =          1'b0;           //Never take a branch in I_TYPE
19    case (opcode)
20      OPCODE_I_TYPE: begin
21        case (funct_3)
22          //((addi)
23          ADDI: begin
24            cu_alu_ctrl =          ALU_ADD;      //ADD operation in ALU
25          end
26      ...

```

Listing 1: Fragmento del RTI de la Unidad de Control

### 4.3. Banco de registros

Para la implementación del módulo `physical_register_file`, se optó por un diseño optimizado que reduce el consumo de recursos en la FPGA al no instanciar físicamente el registro cero. A continuación se detallan las estrategias de diseño.

#### 4.3.1. Optimización de Recursos (Virtualización de x0)

En una implementación ingenua, se declararía un arreglo de memoria de 32 posiciones (`reg [31:0] mem [0:31]`). Sin embargo, dado que el registro `x0` es inmutable, dedicarle 32 Flip-Flops o celdas de memoria es un desperdicio de área.

Solución Implementada: Se diseñó un arreglo físico de 31 registros (índices 1 al 31). El acceso al registro 0 se maneja mediante lógica combinacional en los puertos de salida.

- Declaración del Arreglo: `logic [DATA_WIDTH-1:0] prf [1:31];`
- Beneficio: Se ahorra la instanciación de un registro de 32 bits completo, delegando la función de `x0` a un multiplexor de lectura.

```
1 // Extracto de physical_register_file.sv
2 logic [DATA_WIDTH - 1:0] prf [1:(2**DIR_WIDTH) - 1];      //Physical Register
File 32*32bits
```

Listing 2: `physical_register_file`

#### 4.3.2. Lógica de Lectura Combinacional (Bypass de x0)

La lectura se configuró de manera asíncrona para satisfacer los tiempos de un procesador Single Cycle. Se implementaron dos multiplexores implícitos en los puertos de salida `read_data1` y `read_data2`.

Estos multiplexores evalúan la dirección de lectura entrante:

- Si la dirección es 0: El hardware fuerza la salida a 32'b0 (Tierra), ignorando el contenido de la memoria.
- Si la dirección es 1-31: Se accede al arreglo físico `prf` y se entrega el valor almacenado.

```
1 // Fragmento: Multiplexor de Lectura para x0
2 read_data1 = (read_dir1 == 5'b00000) ? '0 : prf[read_dir1];
```

Listing 3: `physical_register_file`

#### 4.3.3. Escritura Síncrona Protegida

La escritura se diseñó como un proceso secuencial activado por el flanco positivo del reloj (`posedge clk`). Se incluye una protección lógica para evitar corrupción de datos:

- Validación de Write Enable (`w_en`): Solo se escribe si la señal de control está activa.
- Protección de `x0`: Aunque el código lógico intenta escribir, al estar el arreglo físico definido desde el índice 1 (`prf[1:31]`), cualquier intento de escribir en la dirección 0 no tiene un destino físico válido mapeado, por lo que la escritura se descarta implícitamente, cumpliendo con el estándar RISC-V.

## 4.4. ALU

### Descripción General

La Unidad Aritmética Lógica (Arithmethic Processing Unit, ALU) es el modulo encargado de realizar todos los cálculos aritméticos y lógicos del microprocesador.

Para el conjunto de instrucciones mínimo de RV32I del estándar RISC-V, se necesitan las operaciones aritméticas de suma y resta, las operaciones lógicas AND, OR, XOR, las comparaciones de igualdad y menor que con signo y sin signo, desplazamiento lógico de registro izquierdo y derecho, y desplazamiento aritmético de registro derecho.

#### 4.4.1. Entradas y Salidas

Tabla de puertos.

Señal	Dirección	Ancho (bits)	Descripción
operand1	Entrada	32	Primer operando, un registro o la dirección actual del PC
operand2	Entrada	32	Segundo operando, un registro o un valor inmediato
alucontrol	Entrada	4	Selector proveniente de la unidad de control
alu_result	Salida	32	Valor de salida de la operación realizada en la ALU

Cuadro 3: Tabla de especificación de entradas y salidas de la ALU

#### 4.4.2. Fragmento de RTL de la ALU

```
1 'include "defines.sv"
2 module alu #(parameter N = 32)(
3     input logic [N-1:0] operand1,      // First operand, sourced from a register
4     input logic [N-1:0] operand2,      // Second operand sourced from a register
5     input logic [3:0] alucontrol,      // Use 4 bits for the opcode at the moment
6     output logic [N-1:0] alu_result    // ALU result, destiny is another register
7 );
8
9     always_comb begin
10         unique case(alucontrol)
11             ALU_ADD: begin // At the moment we don't care about the carry
12                 alu_result = operand1 + operand2;
13             end
14             ALU_SUB: begin
15                 alu_result = operand1 - operand2;
16             end
17         ...
18     end
19 
```

Listing 4: Modulo generador de inmediatos

## 4.5. Extensión de signo

### 4.5.1. Descripción General

La arquitectura RISC-V define varios formatos de valores inmediatos distribuidos en diferentes campos dentro de cada instrucción tipo I,S,B,U y J. El módulo imm\_gen, identifica estos campos, extrae los bits correspondientes y aplica la extensión de signo para generar el valor inmediato a un ancho de 32 bits, el cual es utilizado por la ALU y el PC del procesador.

### 4.5.2. Entradas y Salidas

Tabla de puertos.

Señal	Dirección	Ancho (bits)	Descripción
instr	Entrada	32	Instrucción leída de la memoria de programa
imm_sel	Entrada	3	Selector proveniente de la unidad de control
imm_out	Salida	32	Valor inmediato generado con extensión de signo

Cuadro 4: Tabla de especificación de entradas y salidas del extensor de signos

La unidad de control indica el tipo de instrucción a procesar mediante la señal imm\_sel.

Valor	Señal	Descripción
000	IMM_I	Formato I (instrucciones inmediatas)
001	IMM_S	Formato (almacenamientos)
010	IMM_B	Formato B (saltos condicionales)
011	IMM_U	Formato U (instrucciones de carga )
100	IMM_J	Formato J (saltos incondicionales)

Cuadro 5: Codificación de imm\_sel

### 4.5.3. RTL

El módulo implementa un bloque combinacional que procesa la instrucción de entrada según el código de selección.

Se identifica el tipo de inmediato mediante imm\_sel, se extrae y se reordena los bits de acuerdo al tipo de instrucción, se aplica la extensión de signo hasta 32 bits y se envía el resultado por imm\_out.

```

1 module imm_gen(
2     input logic [31:0] instr,           // Instructions from program memory
3     input logic [2:0] imm_sel,          // of control unit
4     output logic [31:0] imm_out        // output for ALU and PC
5 );
6
7 'include "defines.svh"             //define the imm instructions
8
9 always_comb begin
10     unique case (imm_sel)
11         /* I-type (ADDI) original immediate 12 bits instr[31:20]
12             12 bits      5bits   3bits   5bits   7bits
13             31:20       19:15    14:12   11:7    6:0
14             imm[31:20]  rs1      funct3   rd      opcode
15         */

```

```
16      IMM_I: begin //000 = I-type
17          imm_out = {{20{instr[31]}}, instr[31:20]}; //signed extension (repeat 20
18          times the MSB)
end
```

Listing 5: Modulo generador de inmediatos

## 4.6. Program Counter (PC)

Este modulo es un registro que contiene la dirección de la próxima instrucción. Se incrementa en 4 bytes por defecto o toma un nuevo valor en ramas o saltos. Solo accede a la memoria de instrucciones, permitiendo ejecución paralela con el acceso a datos.

### 4.6.1. Tamaño e incrementos del PC

En este microprocesador todas las instrucciones miden 32 bits = 4 bytes, por lo tanto, en ejecución secuencial se encuentra:

$$PC_{next} = PC_{current} + 4 \quad (1)$$

Este es el comportamiento estándar en cada ciclo de reloj mientras no haya saltos o ramas.

### 4.6.2. Actualizaciones del PC

**Secuencial.** Esta se mantienen funcionando mientras no se envíe alguna instrucción de la unidad de control.

**Rama.** También conocida como Branch o condicional, en el microprocesador implementado se utiliza la instrucción Branch If Equal(BEQ), esto quiere decir que si la unidad de control envía este tipo de instrucción también se recibe el LSB de la ALU que compara si los registros usados son iguales y si es el caso (True):

$$PC_{next} = PC_{current} + offset \quad (2)$$

Donde el offset esta dado por la extensión de signo.

Y para (false):

$$PC_{next} = PC_{current} + 4 \quad (3)$$

**Salto.** También conocida como JAL o incondicional, para esta instrucción no le es necesario que se cumpla algo solo ocurre:

$$PC_{next} = PC_{current} + immediate \quad (4)$$

### 4.6.3. Secciones del PC

El PC esta conformado por diferentes secciones que dentro del sistema se pueden visualizar como un registro, 2 multiplexores que están condicionados por la unidad de control y un bloque combinacional. El conjunto permite que el PC funcione de forma correcta a partir de las señales que le envíe la unidad de control.

Principalmente el PC es un registro de 32 bits que se actualiza si *prog\_ready* esta habilitado. Un multiplexor permite seleccionar si el incremento de PC sera de 2 bytes o 4 bytes, esto se implemento para que el procesador permita trabajar con instrucciones de 32 o 16 bits. El segundo multiplexor es el que se encarga de ejecutar si se suma el inmediato o se suma el valor numérico, ya sea de 2 o 4. Y por ultimo se tiene un bloque combinacional en el que se realiza la suma. Para obtener el valor actual de PC.

## 4.7. Memoria

Para la implementación del almacenamiento, se diseñaron dos módulos en SystemVerilog: instruction\_memory y data\_memory. A continuación se describe la lógica de diseño y las soluciones adoptadas para cada uno.

### 4.7.1. Implementación de la Memoria de Instrucciones (Instruction Memory)

El objetivo de este módulo es almacenar el programa proveniente del módulo de carga (Shift Register) y servir las instrucciones al procesador. Se identificaron dos retos principales durante el desarrollo: la síntesis de hardware y la coherencia de los datos (Endianness).

#### Arquitectura de Bancos Intercalados (Memory Banking)

Inicialmente, se planteó un arreglo lineal de bytes (logic [7:0] mem [0:1023]). Sin embargo, dado que la escritura ocurre en bloques de 32 bits (4 bytes simultáneos), esto implicaría acceder a 4 direcciones distintas (dir, dir+1...) en un mismo ciclo de reloj. Esto impide que las herramientas de síntesis infieran bloques de memoria RAM estándar (Block RAM), consumiendo excesivos recursos lógicos (Flip-Flops).

**Solución:** Se implementó una arquitectura de "Bancos Intercalados". La memoria se dividió en 4 arreglos paralelos de 256x8 bits cada uno.

- Ventaja: Al escribir una palabra de 32 bits, cada banco recibe un solo byte en la misma dirección relativa (índice), lo cual es perfectamente sintetizable como 4 memorias RAM en paralelo.

```
1 // Fragmento de código: Definición de Bancos
2 logic [BYTE_WIDTH-1:0] bank0 [0:BANK_DEPTH-1]; // Banco para Byte 0
3 logic [BYTE_WIDTH-1:0] bank1 [0:BANK_DEPTH-1]; // Banco para Byte 1
4 logic [BYTE_WIDTH-1:0] bank2 [0:BANK_DEPTH-1]; // Banco para Byte 2
5 logic [BYTE_WIDTH-1:0] bank3 [0:BANK_DEPTH-1]; // Banco para Byte 3
```

Listing 6: Memoria de instrucciones

#### Garantía de Formato Little-Endian

La arquitectura RISC-V sigue el estándar Little-Endian, lo que dicta que el byte menos significativo (LSB) de una instrucción debe residir en la dirección de memoria más baja.

Para cumplir con este estándar y asegurar la integridad de la instrucción al ser leída por el procesador, se diseñó la lógica de escritura respetando estrictamente esta jerarquía:

- Asignación de Bancos: El byte menos significativo proveniente del cargador (data\_in[7:0]) se asigna físicamente al Banco 0 (que representa la dirección base + 0).
- Jerarquía Ascendente: Los bytes subsecuentes ([15:8], [23:16] y [31:24]) se asignan ordenadamente a los bancos 1, 2 y 3 respectivamente.

Esta distribución física asegura que, cuando el procesador solicite una lectura, la concatenación de los bancos reconstruya la instrucción (0x00000513) con los campos de operación en la posición correcta, independientemente de cómo se recibieron los datos serialmente.

#### Lógica de Alineación y Reconstrucción de Instrucciones

A diferencia de la memoria de datos que fuerza una alineación estricta a palabra, la memoria de instrucciones implementa una lógica de lectura flexible basada en los dos bits menos significativos de la dirección (rd\_addr[1:0]). Esto permite discriminar entre accesos alineados y semi-alineados, una característica útil para la robustez del sistema o soporte futuro de instrucciones comprimidas.

**Implementación del Multiplexor de Lectura:** Se diseñó un bloque combinacional (`always_comb`) que evalúa el offset de la dirección solicitada y reconstruye el dato de salida (`rd_data`) acorde al caso:

- **Acceso Alineado a Palabra (2'b00):**Corresponde al flujo normal de ejecución ( $PC = 0, 4, 8\dots$ ). El sistema concatena los cuatro bancos completos para entregar la instrucción estándar de 32 bits.

$$rd\_data = \{bank3, bank2, bank1, bank0\}$$

- **Acceso Alineado a Media Palabra (2'b10):**Corresponde a una dirección que apunta a la mitad superior de una palabra (ej.  $PC = 2, 6, 10\dots$ ). En este caso, la lógica recupera los dos bytes superiores (`bank3` y `bank2`) y los posiciona en la parte baja del bus de datos, llenando con ceros la parte alta. Esto permite leer instrucciones de 16 bits (comprimidas) que residan en la parte alta de una palabra de memoria sin generar errores.

$$rd\_data = \{16'h0000, bank3, bank2\}$$

- **Manejo de Desalineación (default):**Para direcciones impares (2'b01, 2'b11) que no son válidas en la arquitectura RISC-V estándar, el módulo entrega un valor seguro de cero (32'b0), lo cual el procesador interpreta como una instrucción NOP (No Operation) o una instrucción ilegal, evitando comportamientos indeterminados.

```

1 // Fragmento de codigo: Logica de Alineacion
2 always_comb begin
3     case (rd_addr[1:0])
4         2'b00: rd_data = {bank3[rd_addr[ADDR_WIDTH-1:2]], bank2[rd_addr[
5             ADDR_WIDTH-1:2]], bank1[rd_addr[ADDR_WIDTH-1:2]], bank0[rd_addr[
6                 ADDR_WIDTH-1:2]]};
7         2'b10: rd_data = {16'b0, bank3[rd_addr[ADDR_WIDTH-1:2]], bank2[
8             rd_addr[ADDR_WIDTH-1:2]]};
9         default : rd_data = '0;
10    endcase
11 end

```

Listing 7: Memoria de instrucciones

#### 4.7.2. Implementación de la Memoria de Datos (Data Memory)

Este módulo funge como el espacio de trabajo principal para las variables y resultados del procesador. A diferencia de la memoria de instrucciones, donde se requirió una arquitectura de bancos para manejar la escritura desalineada, la memoria de datos se diseñó bajo un esquema de acceso alineado a palabra, simplificando su estructura interna.

##### Organización Física (Word-Oriented Architecture)

Dado que el procesador RISC-V opera nativamente con datos de 32 bits y el bus de escritura (`wd`) entrega palabras completas, no fue necesario dividir la memoria en bancos de bytes. Se implementó un arreglo lineal único con las siguientes características:

- Ancho de Palabra: 32 bits (coincidente con el bus del CPU).
- Profundidad: 1024 Direcciones.
- Capacidad Total: Al ser 1024 palabras de 4 bytes cada una, la capacidad total de almacenamiento es de 4 KB (4096 Bytes), proporcionando espacio suficiente para el stack y las variables globales del programa Fibonacci.

```

1 // Fragmento: Definicion de Memoria de Palabras
2 logic [DATA_WIDTH-1:0] ram [0:MEM_DEPTH-1]; // 1024 x 32 bits

```

Listing 8: Memoria de datos

### Interfaz de Direccionamiento (Word Alignment)

Uno de los retos en la interfaz con la ALU es que esta genera direcciones a nivel de byte (ej. 0x00, 0x04, 0x08), mientras que el arreglo de memoria en Verilog se indexa por número de renglón.

**Solución:** Se implementó un mecanismo de truncamiento de bits (Bit Slicing) en la entrada de dirección. Se descartan físicamente los dos bits menos significativos (addr[1:0]) y se utilizan los bits superiores (addr[ADDR\_WIDTH-1:2]) como índice. Matemáticamente, esto equivale a dividir la dirección entre 4 (Index = Address/4), garantizando que cualquier acceso dentro del rango de una palabra (ej. bytes 0, 1, 2 o 3) apunte al mismo renglón físico de 32 bits.

```

1 // Logica de Lectura Combinacional
2 // Se usan los bits [ADDR_WIDTH-1:2] para ignorar el offset de byte
3 assign rd_data = ram[addr[ADDR_WIDTH-1:2]];

```

Listing 9: Memoria de datos

### Sincronización para Ciclo Único

El control de tiempos se diseñó para cumplir con los requisitos de un procesador Single Cycle:

1. **Lectura Combinacional (Asíncrona):** Se utilizó lógica assign para la lectura. Esto permite que, en una instrucción de carga (LW), el dato esté disponible en el puerto rd\_data en el mismo ciclo de reloj en que la ALU calcula la dirección, eliminando latencias de lectura que requerirían detener el procesador (Stalling).
2. **Escritura Secuencial (Síncrona):** La escritura se protege mediante el reloj y la señal de control w\_en (Write Enable). El dato solo se almacena en el flanco positivo del reloj, asegurando que solo se capture el resultado final y estable de la ALU, evitando corrupción de datos por "glitches combinacionales previos.

#### 4.7.3. Parametrización del Diseño

Con el objetivo de maximizar la reusabilidad del código y facilitar la escalabilidad del sistema, ambos módulos de memoria se implementaron utilizando la sintaxis de parámetros (parameter) de SystemVerilog. Esta metodología permite desacoplar la lógica de control de las dimensiones físicas del hardware, permitiendo instanciar memorias de diferentes capacidades o anchos de banda desde el módulo superior (top\_module) sin necesidad de modificar el código fuente RTL.

La configuración del subsistema se rige por los siguientes parámetros globales:

- DATA\_WIDTH (Valor por defecto: 32) Define la arquitectura del bus de datos. Para este proyecto, se fija en 32 bits para cumplir con el estándar RV32I de RISC-V.
- ADDR\_WIDTH (Valor por defecto: 32) Establece el ancho del bus de direcciones proveniente del Contador de Programa (PC) y la ALU. Este parámetro dimensiona los puertos de entrada (rd\_addr, wr\_addr), asegurando la compatibilidad de interfaz con el núcleo del procesador.
- MEM\_DEPTH (Valor por defecto: 1024) Determina la profundidad del arreglo de memoria (cantidad de localidades direccionables).

## 4.8. UART

### 4.8.1. Descripción General

El módulo UART implementado en este diseño constituye una comunicación serial asíncrona encargado de recibir, ensamblar, direccionar y almacenar instrucciones al módulo de memoria de instrucción del procesador, específicamente para ejecutar algoritmos para el cálculo de la serie de Fibonacci.

El sistema está compuesto por módulos principales: baud\_rate\_generator, receiver, shift\_register\_fsm, instruction\_memory y uart\_top, y módulos auxiliares: transmitter, display\_7\_segments.

Se emplean scripts en Python capaces de generar las instrucciones que serán cargadas por UART. Estos scripts producen los archivos binarios o secuencias de bytes que el módulo UART recibirá y utilizará para inicializar el programa contenido en el procesador.

El proceso global se ejecuta de la siguiente manera:

- **Sincronización:** El baud\_rate\_generator produce el pulso tick a  $16 \times$  el baud rate y define la temporización para todos los módulos internos.
- **Recepción:** El receiver toma los bits seriales recibidos (rx) y los convierte en bytes de 8 bits y activa rx\_done al finalizar.
- **Ensamblaje y Control:** El shift\_register\_fsm reúne 4 bytes de 8 bits para formar la instrucción de 32 bits, gestionando la dirección de escritura (wr\_addr). Cuando la palabra está completa, activa inst\_rdy.
- **Almacenamiento:** Recibe inst\_rdy y escribe la instrucción de 32 bits usando wr\_addr, en instruction\_memory.
- **Finalización:** Supervisa el proceso; cuando todas las instrucciones están cargadas, activa prog\_rdy para habilitar el inicio del fetch del procesador.

### 4.8.2. Especificaciones

Especificación	Descripción
Formato de comunicación	UART estándar: 1 bit de inicio, 8 bits de datos, sin paridad, 1 bit de parada.
Sobremuestreo	$\times 16$ para mejorar precisión en muestreo del receptor.
Longitud de instrucción	32 bits (organizada en 4 bytes).
Profundidad de memoria	1024 bytes totales .
Direcciones de memoria	10 bits para direccionar 1024 posiciones.
Reloj del sistema	50 MHz.
Tasa de baudios	9600
Proceso de carga	Secuencia de bytes generada por los scripts de Python siguiendo el orden Little Endian requerido.

Cuadro 6: Especificaciones globales del módulo UART.

#### 4.8.3. Entradas y Salidas

Señal	Dirección	Ancho (bits)	Descripción
clk	Entrada	1	Reloj principal del sistema (50 MHz).
rst_n	Entrada	1	Reset asíncrono activo en bajo.
rx	Entrada	1	Línea serial para recepción UART.
baud_sel	Entrada	3	Selección de tasa de baudios.
next_program	Entrada	1	Reinicia el proceso de carga del programa.
tx_start	Entrada	1	Inicia transmisión UART (modo prueba).
rdaddr	Entrada	10	Dirección de lectura hacia la memoria de instrucciones.
prog_rdy	Salida	1	Indica que el programa se cargó completamente.
rx_done	Salida	1	Señal que indica que un byte se recibió correctamente.
data_out	Salida	32	Palabra ensamblada lista para escribir en memoria.
read_data	Salida	32	Dato leído desde la memoria de instrucciones.
tx	Salida	1	Línea de transmisión serial UART.
state	Salida	4	Estado interno del FSM de carga.
busy	Salida	1	Indica que la carga de programa está en proceso.
ready	Salida	1	Indica que el sistema está listo para recibir más instrucciones.
tx_done	Salida	1	Indica fin de transmisión UART.

Cuadro 7: Entradas y salidas del módulo `uart_top`.

#### 4.8.4. Módulo `baud_rate_generator`

El módulo `baud_rate_generator` implementa la lógica de sincronización. A partir del reloj principal del sistema y de la selección de baudios, genera un pulso tick cuya frecuencia equivale a diecisésis veces el baud rate.

El bloque de lógica secuencial utiliza un contador que se incrementa con cada pulso del reloj del sistema. Cuando el contador alcanza el valor del divisor correspondiente al baud rate seleccionado, se genera un pulso de un ciclo (tick), y el contador se reinicia.

Este módulo proporciona al módulo de `receiver` y `transmitter` sincronización.

Señal	Dirección	Ancho (bits)	Descripción
clk	Entrada	1	Reloj del sistema.
rst	Entrada	1	Reset síncrono.
tick	Salida	1	Pulso para los módulos TX y RX.

Cuadro 8: Entradas y salidas del módulo Baud Rate Generator

#### 4.8.5. Módulo `receiver`

Este módulo reconstruye un byte a partir del flujo serial recibido por la linea rx. Utiliza el pulso de muestreo generado por el baud rate generator para detectar el bit de inicio, muestrear cada bit de datos y validar el bit de parada. Cuando se completa la recepción de 8 bits, el módulo activa la señal `rx_done` y entrega el byte hacia el módulo `shift_register_fsm`.

Esta estructurado en dos bloques principales: Un circuito secuencial encargado de almacenar el estado y los contadores e implementa un circuito combinacional para la máquina de estados finitos con cuatro estados:

- **Estado IDLE:** Monitoriza continuamente la línea rx. Espera transición de 1 a 0. La transición a START se da cuando detecta rx = 0
- **Estado START:** En el tick 7, verifica que rx siga en 0 y si es válido, confirma que es un start bit válido y si en tick 7 rx=1, regresa al estado anterior. Al completar los 16 ticks, transiciona al estado DATA para comenzar a recibir los bits de datos.
- **Estado DATA:** Recibe 8 bits de datos, LSB primero. Después del bit 7, transición a STOP
- **Estado STOP:** Espera 16 ticks completos. Verifica que rx = 1 (bit de parada válido). Activa rx\_done = 1. Transición a IDLE para siguiente byte.

Señal	Dirección	Ancho (bits)	Descripción
clk	Entrada	1	Reloj principal del sistema.
arst_n	Entrada	1	Reset asíncrono activo en bajo.
tick	Entrada	1	Pulso proveniente del generador de baud rate.
rx	Entrada	1	Línea serial donde llegan los bits transmitidos.
rx_done	Salida	1	Indica que el byte completo se recibió correctamente y está disponible en data_out.
data_out	Salida	8	Byte reconstruido a partir de los bits recibidos.

Cuadro 9: Entradas y salidas del módulo receiver.

#### 4.8.6. Módulo shift\_register\_fsm

Su función principal es tomar los bytes de 8 bits que provienen del módulo receiver y concatenarlos secuencialmente para formar la palabra de instrucción completa de 32 bits, respetando la convención Little Endian. Se implementa un circuito secuencial que almacena y actualiza el estado de forma síncrona. Además, implementa una máquina de estados finitos combinacional de 8 estados que gestiona el ciclo completo de carga, desde la lectura de la longitud del programa hasta la habilitación de la escritura en la memoria y la señalización final al procesador. Los estados son:

- **WAIT\_PARAMS:** Espera recibir el primer byte que indica el número total de instrucciones a cargar. Cuando w\_en = 1, almacena data\_in en n\_of\_instructions, reinicia los contadores y transiciona a WAIT\_BYTETO.
- **WAIT\_BYTETO:** Espera el byte 0 (LSB) de la primera instrucción. Cuando w\_en = 1, almacena el byte en temp\_data\_out[7:0] y transiciona a WAIT\_BYTE1.
- **WAIT\_BYTE1:** Espera el byte 1 de la instrucción. Cuando w\_en = 1, almacena el byte en temp\_data\_out[15:8] y transiciona a WAIT\_BYTE2.
- **WAIT\_BYTE2:** Espera el byte 2 de la instrucción. Cuando w\_en = 1, almacena el byte en temp\_data\_out[23:16] y transiciona a WAIT\_BYTE3.
- **WAIT\_BYTE3:** Espera el byte 3 (MSB) de la instrucción. Cuando w\_en = 1, almacena el byte en temp\_data\_out[31:24] (instrucción completa de 32 bits) y transiciona a WRITE\_INSTRUCTION.

- **WRITE\_INSTRUCTION:** Estado donde la instrucción de 32 bits está completa. Activa `inst_rdy = 1` para indicar que la instrucción debe escribirse en memoria. Incrementa `instruction_counter` en 4 (byte addressing) y: Si es la última instrucción: transiciona a `DONE` y si no: transiciona a `WAIT_BYTE0` para siguiente instrucción.
- **DONE:** Activa `prog_rdy = 1` para indicar al procesador que el programa está listo. Si `next_program = 0`, transiciona a `CLEAN_MEMORY` para preparar nueva carga.
- **CLEAN\_MEMORY:** Limpia la memoria escribiendo ceros en todas las posiciones.

Señal	Dir.	Ancho (bits)	Descripción
<code>clk</code>	Entrada	1	Reloj principal del sistema.
<code>rst_n</code>	Entrada	1	Reset asíncrono activo en bajo.
<code>next_program</code>	Entrada	1	Indica el reinicio del proceso de carga una vez que el programa previo terminó.
<code>w_en</code>	Entrada	1	Pulso que indica que un nuevo byte fue recibido desde el módulo UART Receiver ( <code>rx_done</code> ).
<code>data_in</code>	Entrada	8	Byte recibido desde el UART que se irá ensamblando en la instrucción de 32 bits.
<code>data_out</code>	Salida	32	Instrucción ensamblada (32 bits) lista para escribirse en memoria.
<code>wr_addr</code>	Salida	10	Dirección de escritura para la memoria de instrucciones (avanza de 4 en 4).
<code>inst_rdy</code>	Salida	1	Habilitación de escritura para la memoria. Se activa cuando se completan 4 bytes
<code>n_instructions</code>	Salida	8	Número total de instrucciones que se cargarán, proveniente del primer byte recibido.
<code>busy</code>	Salida	1	Indica que el módulo está en proceso de carga.
<code>state</code>	Salida	4	Valor actual del estado de la FSM.
<code>ready</code>	Salida	1	Indica que el módulo está esperando parámetros y puede iniciar una nueva carga.
<code>prog_rdy</code>	Salida	1	Indica que todo el programa ha sido cargado exitosamente en memoria.

Cuadro 10: Entradas y salidas del módulo `shift_register_fsm`.

#### 4.8.7. Módulo transmitter

Este módulo convierte bytes paralelos de 8 bits en flujo serial. Esta sincronizado con la señal `tick` proveniente del `baud_rate_generator`, que determina el instante exacto de muestreo y avance entre los estados. El módulo implementa una maquina de estados finitos combinacional:

- **Estado IDLE:** Mantiene la línea `tx` en nivel alto y espera la señal `tx_start`. Cuando `tx_start = 1`, carga el byte a transmitir en el registro de desplazamiento, establece `tx = 0` y transiciona al estado `START`
- **Estado START:** Mantiene `tx = 0` durante 16 ticks completos. En cada tick del `baud generator` incrementa el contador. Al llegar a `BIT_SAMPLING = 15`, reinicia el contador y transiciona al estado `DATA`.
- **Estado DATA:** Se transmite el byte LSB-first, usando `shifted_data`. Tras enviar los 8 bits, avanza a `STOP`.

- **Estado STOP:** Transmite el bit de parada

Señal	Dirección	Ancho (bits)	Descripción
clk	Entrada	1	Reloj principal del sistema.
rst_n	Entrada	1	Reset asíncrono activo en bajo.
data_in	Entrada	8	Byte a transmitir serialmente por la línea tx.
tick	Entrada	1	Pulso del generador de baud rate.
tx_start	Entrada	1	Orden de inicio de transmisión.
tx	Salida	1	Línea serial de transmisión. Contiene el bit start, los bits de datos y el bit de stop según la FSM.
tx_done	Salida	1	Indica que el byte completo ha sido transmitido.

Cuadro 11: Entradas y salidas del módulo transmitter

#### 4.8.8. Comunicación con Python

Las instrucciones del programa son cargadas a través del modulo UART al microprocesador, para brindarle las instrucciones al modulo UART se realizó un script en python que apoya en la transmisión de toda la información necesaria para realizar cierta cantidad de iteraciones de la serie de Fibonacci.

```

1
2     try:
3         ser = serial.Serial(port, baudrate, timeout=timeout)
4         time.sleep(0.05)
5
6         # Separar el inmediato en dos bytes (little-endian)
7         imm_high = (imm & 0x0F) << 4          # LSB
8         imm_low  = (imm >> 4) & 0xFF # MSB
9
10        # Construye payload dinamico
11        payload = [
12            bytes([0x13, 0x05, 0x00, 0x00]),
13            bytes([0x93, 0x05, 0x10, 0x00]),
14            bytes([0x13, 0x06, imm_high, imm_low]), # <- SE ACTUALIZA AQUI
15            bytes([0x63, 0x0C, 0x06, 0x00]),
16            bytes([0xB3, 0x02, 0xB5, 0x00]),
17            bytes([0x13, 0x85, 0x05, 0x00]),
18            bytes([0x93, 0x85, 0x02, 0x00]),
19            bytes([0x13, 0x06, 0xF6, 0xFF]),
20            bytes([0x6F, 0xF0, 0xDF, 0xFE]),
21            bytes([0x6F, 0x00, 0x00, 0x00])
22        ]
23
24        print(f"\n==> Enviando inmediato {hex(imm)} -> HIGH={hex(imm_high)}, LOW={hex(
25             imm_low)} ==")
26
27        # Enviar numero de instrucciones (10 -> 0x0A)
28        n_instr = b'\x0A'
29        ser.write(n_instr)
30        time.sleep(0.005)
31
32        # Enviar instrucciones una por una
33        for frame in payload:
34            ser.write(frame)
35            print(f"Sent: {frame}")
36            time.sleep(0.005)
37
38        # Pausa opcional entre iteraciones
39        time.sleep(0.05)

```

```
39     ser.close()
40     print("Serial port closed.")
41
42 except Exception as e:
43     print("ERROR:", e)
```

Listing 10: Script en python

## 5. Simulación y verificación

### 5.1. Verificación

El proceso de verificación de un procesador consiste en asegurar que su comportamiento funcional coincide con la arquitectura y especificación definidas. Este proceso se realiza antes de la síntesis o implementación en hardware, y tiene como objetivo identificar errores de diseño, comprobar la correcta interacción entre módulos internos y garantizar que las instrucciones se ejecuten según lo establecido en el conjunto de instrucciones.

La verificación se lleva a cabo mediante la simulación del procesador en un entorno controlado, donde se generan estímulos de entrada, se observan las señales internas y se comparan los resultados obtenidos con los valores esperados. Esta comparación permite validar que el procesador responde correctamente bajo diferentes condiciones.

#### 5.1.1. Testbench

Se desarrolló un testbench encargado de instanciar el procesador completo y de proveer los estímulos necesarios para la verificación del correcto funcionamiento del procesador.

El objetivo del testbench es generar instrucciones aleatorias que entran al sistema y que por medio de aserciones se realiza un análisis de las entradas que se colocan con las salidas que se obtienen y estas se comparan las salidas esperadas, y por medio de estas pruebas se busca el escenario en el caso que se presentan errores en caso contrario se demuestra que efectivamente el sistema se comporta como se espera.

#### 5.1.2. Generación de instrucciones.

Para la generación de instrucciones se define un *interface* que permite generar y enviar instrucciones al procesador durante la verificación y simulación, este solo recibe como señales externas el reloj y un reset. Dentro de este se declara una señal que se encarga de contener la instrucción RISC-V generada, para luego ser enviada al procesador.

Para el objetivo de este procesador, que es ejecutar la Secuencia de Fibonacci se usaron 4 tipos de instrucciones ADD, ADDI, BEQ y JAL.

#### 5.1.3. Formatos de instrucción

El *interface* incluye funciones para construir cada instrucción de forma específica. Todas las instrucciones se arman colocando los campos correspondientes en sus posiciones dentro del formato binario según la arquitectura RISC-V. Y para las instrucciones usadas estos son los formatos a utilizar.

funct7	rs2	rs1	funct3	rd	opcode
--------	-----	-----	--------	----	--------

Cuadro 12: Formato ADD

imm[11:0]	rs1	funct3	rd	opcode
-----------	-----	--------	----	--------

Cuadro 13: Formato ADDI

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
---------	-----------	-----	-----	--------	----------	---------	--------

Cuadro 14: Formato BEQ

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
---------	-----------	---------	------------	----	--------

Cuadro 15: Formato JAL

#### 5.1.4. Ejecución aleatoria de instrucciones

Se implementa un módulo que es un entorno de verificación para un procesador RISC-V. Su objetivo es generar instrucciones aleatorias del conjunto ADD, ADDI, BEQ y JAL, enviarlas al procesador y comprobar que cada módulo interno funcione correctamente.

Se instancia la interfaz `microprocessor_if`, la cual construye instrucciones combinando registros e inmediatos aleatorios. Cada instrucción se asigna a `instruction` para ser enviada al procesador. Mediante directivas `define` se crean accesos abreviados a señales internas como la ALU, el PC y el banco de registros.

Después de liberar el reset, el testbench ejecuta un millón de iteraciones. En cada una, `randcase` selecciona aleatoriamente una instrucción con probabilidades iguales. Para cada instrucción se calculan los valores esperados:

- **ADD/ADDI:** suma de operandos.
- **BEQ:** actualización del PC según la comparación de registros.
- **JAL:** dirección de salto y valor PC+4 para rd.

### 5.1.5. Monitoreo de entradas y salidas de los módulos internos.

Los resultados de la ejecución aleatoria de instrucciones alimentan las propiedades SVA, que comparan automáticamente el comportamiento real del procesador con el esperado, validando así la ALU, el PC y el banco de registros y a partir del monitoreo de señales se permite observar en tiempo real el comportamiento interno del procesador durante la ejecución de cada instrucción. A través de la herramienta *SimVision* se visualizan señales clave como el valor del PC, las operaciones realizadas por la ALU, el contenido de los registros y los cambios en las líneas de control. Esto facilita identificar si el procesador está decodificando, ejecutando y escribiendo resultados correctamente en cada ciclo.

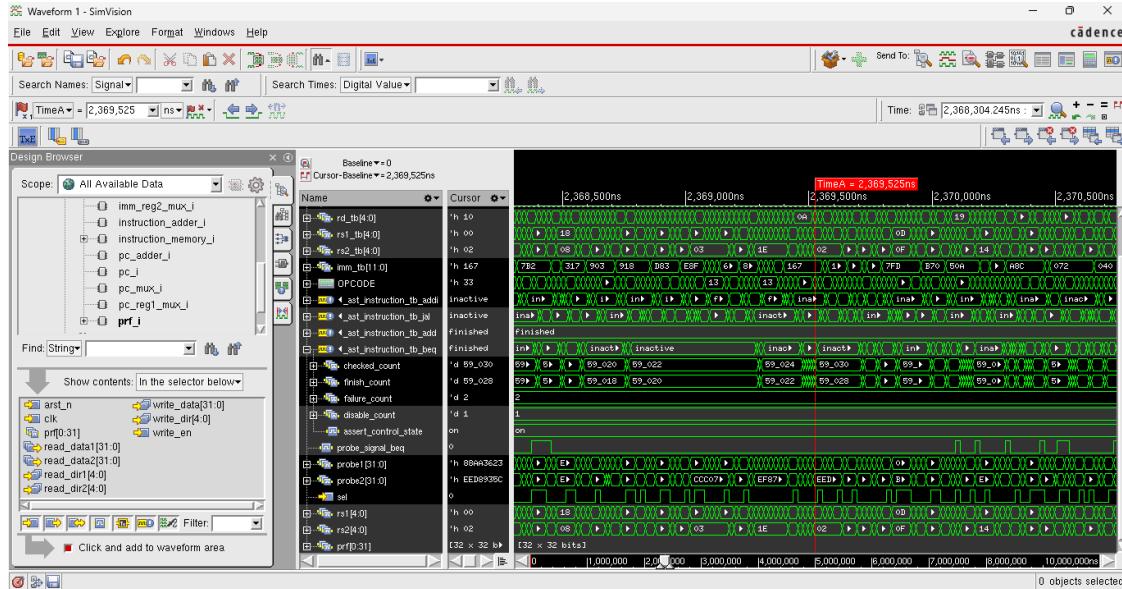


Figura 7: Señales de entradas y salidas

De esta manera, es posible analizar el comportamiento de cada módulo en el sistema.

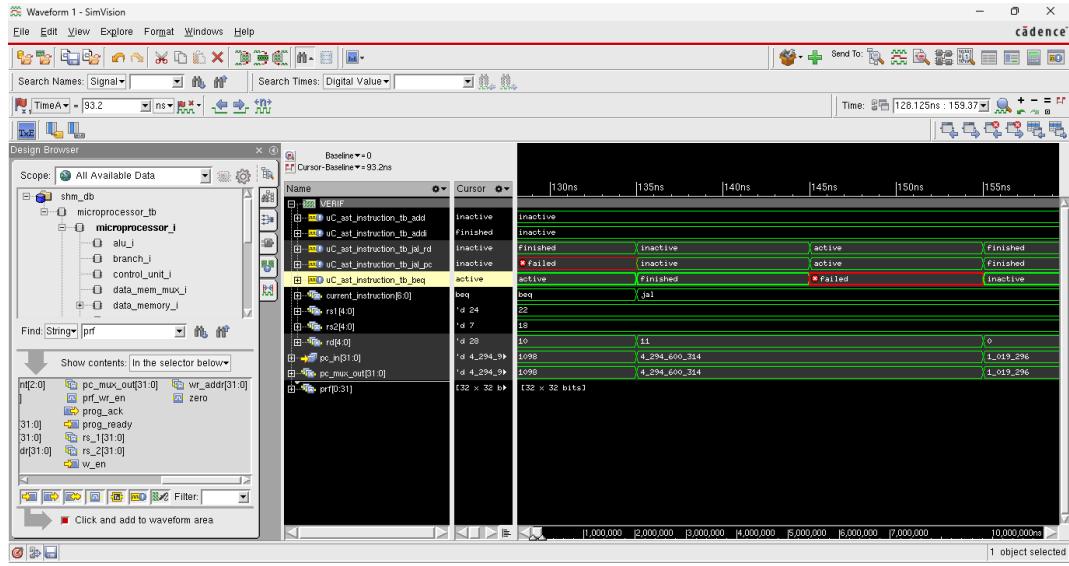


Figura 8: Resultados de verificación

Finalmente, la simulación se detiene automáticamente, garantizando que el análisis cubra un número suficiente de ciclos sin extender innecesariamente el tiempo de ejecución.

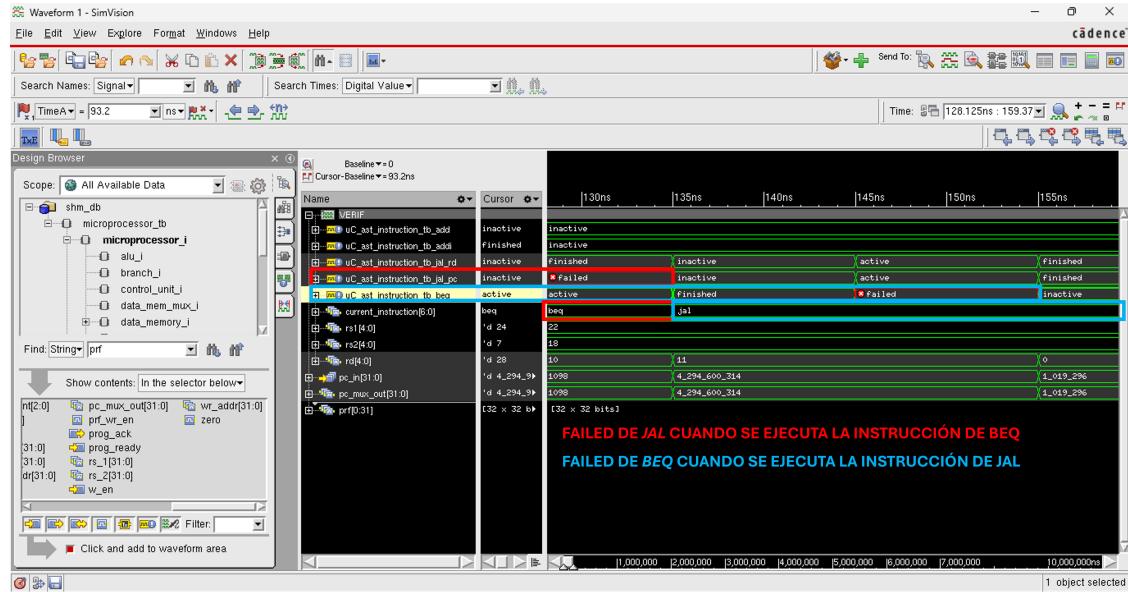


Figura 9: Enfoque de error en los resultados de la verificación

Como puede apreciarse en la Figura 9, la simulación de la verificación mediante aserciones sólo reporta un tipo de error: fallas en las instrucciones `jal` cuando en realidad se está ejecutando una `beq`, y viceversa. Tras un análisis exhaustivo de las señales involucradas y de su evolución temporal, se identificó que este comportamiento se debía a que la aserción evaluaba un flanco de señal posterior al ciclo en el que la instrucción correspondiente estaba activa, provocando así una desincronización entre la condición verificada y la instrucción efectivamente en ejecución.

## 5.2. Simulación

La simulación consiste en ejecutar el procesador con una secuencia de instrucciones definida, para comprobar que cumple una tarea específica. En este proyecto, el objetivo de la simulación es verificar que el procesador ejecuta correctamente la secuencia Fibonacci.

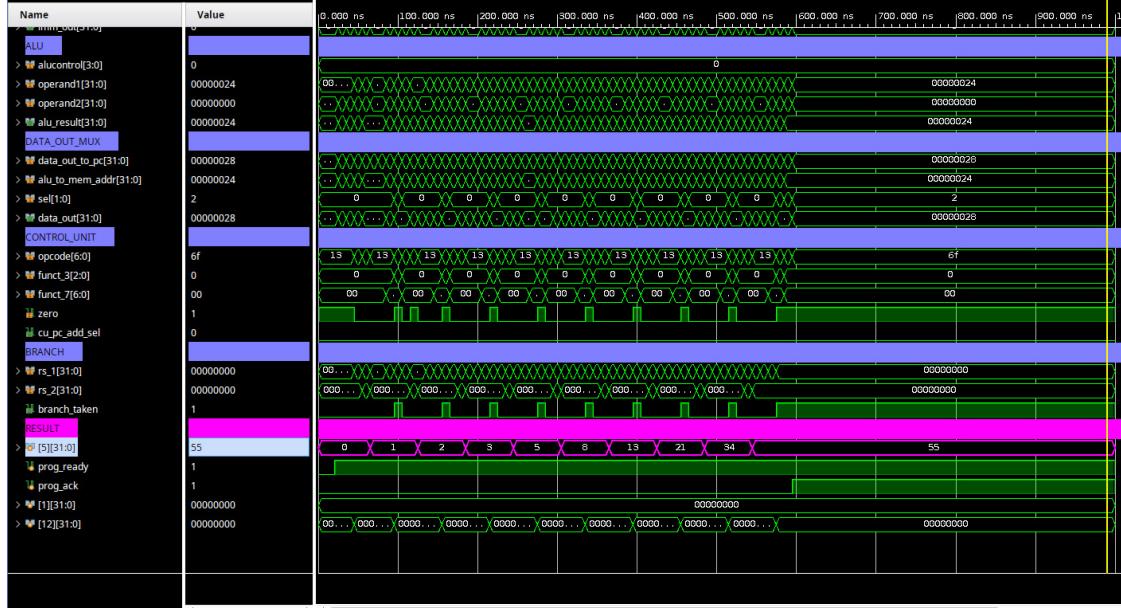


Figura 10: Resultados de Simulación en 10 iteraciones

## 6. Resultados

La comunicación y carga del programa se realizó mediante UART, tras lo cual el módulo Program Counter recibió la señal de inicio y comenzó la secuencia de iteraciones.

Durante la simulación, el procesador ejecutó correctamente el algoritmo iterativo de la serie Fibonacci. Se inicializaron los registros  $x10$  en 0,  $x11$  en 1,  $x12$  como contador encargado de determinar los términos a ejecutar de la serie Fibonacci. A modo de prueba, se asignó al contador 43 que corresponde 44 términos de la serie ( $F_0=0, F_1=1, \dots, F_{44}=43$ ).

El bucle principal realizó la suma de los dos términos previos, actualizó los registros y decrementó el contador hasta llegar a cero. Al cumplirse la condición, se ejecutó el salto a la etiqueta fin, momento en el cual la unidad de control detectó que la serie había dejado de incrementar (condición de parada). Una vez alcanzado este punto, se generó una señal hacia la máquina de estados del shifter, encargada de limpiar la memoria interna y regresar al estado de espera para la siguiente carga de programa.

```
1 .globl _start    # Hace visible la etiqueta de inicio
2
3 start:
4 #---Inicializacion
5 li a0, 0          #a0 = F(n) actual (empezamos en 0)
6 li a1, 1          #a1 = F(n+1) siguiente
7 li a2, 43         #a2 = Contador (calcularemos 10 veces)
8
9 loop
10 #---Condicion de parada
11 beqz a2,fin     # Si el contador (a2) llega a 0, ir a 'fin'
12
13 # --- Calculo de Fibonacci
14 add t0, a0, a1   # t0 = a0 + a1 (Calculamos el siguiente)
15 mv a0, a1         # a0 toma el valor viejo de a1
16 mv a1, t0         # a1 toma el nuevo valor calculado
17
18 #---Actualizar contador
19 addi a2, a2, -1  # Restamos 1 al contador
20 j loop            # Repetimos el ciclo
21
22 fin:
23 j fin             # Bucle infinito para detenernos aqui
```

Listing 11: Código ensamblador para secuencia de Fibonacci

```
00000000 <_start>:
0: 00000513    addi x10 x0 0
4: 00100593    addi x11 x0 1
8: 02b00613    addi x12 x0 43

0000000c <loop>:
c: 00060c63    beq x12 x0 24 <fin>
10: 00b502b3   add x5 x10 x11
14: 00058513   addi x10 x11 0
18: 00028593   addi x11 x5 0
1c: fff60613   addi x12 x12 -1
20: fedff06f   jal x0 -20 <loop>

00000024 <fin>:
24: 0000006f   jal x0 0 <fin>
```

Listing 12: Código en lenguaje máquina para secuencia de Fibonacci

La implementación del algoritmo de la serie Fibonacci se sintetizó y ejecutó en la FPGA. Con el protocolo UART se envia byte por byte el programa Fibonacci a la FPGA.



Figura 11: Envío de bytes por protocolo UART a la FPGA

Finalizada la carga del programa Fibonacci, la UART le envía una señal de inicio al *Program Counter*. Dada la señal de inicio, el microprocesador empieza a operar correctamente.



Figura 12: Cargado del programa Fibonacci en la FPGA por protocolo UART.

Durante la ejecución, el hardware generó los términos correspondientes y los colocó en los registros de salida, los cuales fueron visualizados en formato hexadecimal. Dicho valor se expresa únicamente con los 32 bits menos significativos, tal como está definido en el diseño para manejar casos de desbordamiento. El valor mostrado corresponde al término F(44) de la serie.

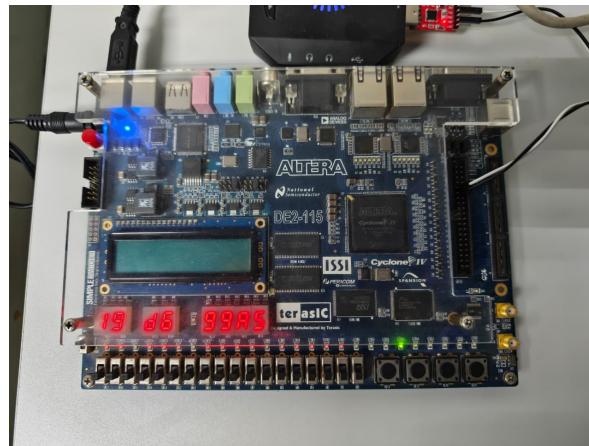


Figura 13: Funcionamiento en el FPGA

Los resultados mostrados confirman:

- La UART carga correctamente el programa en la memoria de instrucciones.
- El *Program Counter* inicia la ejecución únicamente cuando recibe la señal de habilitación.
- El procesador calcula correctamente la serie Fibonacci.
- El valor final reportado por la FPGA coincide con el resultado teórico esperado.

## 7. Conclusiones

### Logros y Cumplimiento de Objetivos

El objetivo principal del proyecto se cumplió satisfactoriamente con la implementación funcional de un microprocesador basado en la arquitectura RISC-V RV32I de ciclo único (Single-Cycle). Se logró integrar exitosamente una arquitectura Harvard Modificada, donde la separación de las memorias de instrucciones y datos permitió un flujo de ejecución eficiente<sup>1</sup>. Un hito crítico alcanzado fue el diseño del sistema de carga dinámica mediante UART, el cual sustituyó la necesidad de memorias ROM estáticas. El sistema es capaz de recibir, ensamblar y escribir instrucciones de 32 bits a partir de una transmisión serial de 8 bits, validando el diseño mediante la ejecución correcta del algoritmo de la serie de Fibonacci. No obstante, el diseño se limitó estrictamente al conjunto de instrucciones base de enteros, dejando fuera la implementación de excepciones de hardware o interrupciones, las cuales son necesarias para un sistema operativo completo.

**Aprendizajes Técnicos:** El desarrollo del proyecto proporcionó conocimientos profundos sobre la gestión de recursos en FPGA y la coherencia de datos:

- **Gestión de Memoria y Ancho de Banda:** Se comprendió la complejidad de interconectar periféricos de baja velocidad (UART de 8 bits) con un núcleo de alto ancho de banda (CPU de 32 bits). La solución de "Memoria Bancarizada"(Interleaved Memory) fue un aprendizaje clave, permitiendo escrituras parciales de bytes sin romper la estructura de lectura de palabras completas, esencial para que las herramientas de síntesis infirieran correctamente la RAM.
- **Endianness y Protocolos:** Se asimiló la importancia de la alineación de bytes (Byte Alignment). Dado que RISC-V es Little-Endian y la UART transmite secuencialmente, fue necesario implementar lógica de reordenamiento en la etapa de recepción para garantizar que el byte menos significativo se almacenara en la dirección más baja, asegurando la integridad de la instrucción decodificada.
- **Optimización de Hardware:** La implementación del Banco de Registros demostró que la "virtualización" de componentes (como el registro *x0*) mediante lógica combinacional ahorra recursos secuenciales (Flip-Flops) valiosos en la FPGA.

### Relevancia de SystemVerilog frente a Verilog

La elección de SystemVerilog fue determinante para la claridad y robustez del diseño, superando las limitaciones del Verilog tradicional (Verilog-95/2001) en varios aspectos observados en el código fuente:

- Abstracción de Tipos de Datos: El uso del tipo logic eliminó la confusión común entre wire y reg, simplificando la declaración de puertos y señales internas.
- Intención de Diseño Explícita: Bloques como always\_comb y always\_ff permitieron separar explícitamente la lógica combinacional de la secuencial. Esto no solo mejora la legibilidad, sino que ayuda a las herramientas de síntesis a detectar latches indeseados automáticamente.
- Máquinas de Estado y Enumeraciones: La capacidad de definir tipos enumerados (typedef enum) facilitó la implementación de la FSM del módulo UART y del receptor, haciendo el código auto-dокументado y fácil de depurar en comparación con la codificación manual de estados en binario (localparam) típica de Verilog.

- Verificación: Estructuras como unique case en la ALU proporcionaron directivas claras al sintetizador para evitar prioridades innecesarias y asegurar que todos los casos (operaciones) estuvieran cubiertos.

### Trabajo Futuro y Posibles Mejoras

Para evolucionar el diseño actual hacia un procesador de mayor rendimiento comercial, se identifican las siguientes líneas de trabajo:

- **Segmentación (Pipelining):** La arquitectura actual de ciclo único limita la frecuencia máxima de reloj (actualmente 50 MHz) debido a la propagación de la ruta crítica más larga. Implementar una segmentación de 5 etapas (Fetch, Decode, Execute, Memory, Writeback) incrementaría drásticamente el throughput y la frecuencia de operación.
- **Soporte de Interrupciones (CSRs):** Implementar los Registros de Estado y Control (CSR) para permitir el manejo de excepciones y temporizadores, un requisito para soportar sistemas operativos en tiempo real.
- **Extensiones de la ISA:** Incorporar la extensión 'M' (Multiplicación/División hardware) para acelerar cálculos matemáticos complejos que actualmente se realizan por software.

## Referencias

- [1] Andrew Waterman, Krste Asanović y et. al. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Specification. Document Version 20250508. RISC-V International, mayo de 2025. URL: <https://riscv.org/specifications/>.
- [2] Vikash Prajapati y Alpana Pandey. «FPGA based UART Design: Simulation, Synthesis & System Verilog functional verification». En: *2025 IEEE Guwahati Subsection Conference (GCON)*. Itanagar, India: IEEE, jun. de 2025, págs. 1-5. ISBN: 9798331513450. DOI: 10.1109/GCON65540.2025.11173388. URL: <https://ieeexplore.ieee.org/document/11173388/> (visitado 08-12-2025).
- [3] Weilun Huang y Guolun Sheng. «Analysis and Research on UART Communication Protocol». En: *2024 4th Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. Shenyang, China: IEEE, feb. de 2024, págs. 768-771. ISBN: 9798350359985. DOI: 10.1109/ACCTCS61748.2024.00140. URL: <https://ieeexplore.ieee.org/document/10612928/> (visitado 08-12-2025).
- [4] Onur Toker. «A High-Level Synthesis Approach for a RISC-V RV32I-Based System on Chip and Its FPGA Implementation». En: *The 10th International Electronic Conference on Sensors and Applications*. MDPI, nov. de 2023, pág. 72. DOI: 10.3390/ecsa-10-16212. URL: <https://www.mdpi.com/2673-4591/58/1/72> (visitado 08-12-2025).
- [5] Abhinav Rajyan y Gaurav Saini. «SystemVerilog Based Design of an RV32I Compliant RISC-V Processor Core». En: *2024 5th IEEE Global Conference for Advancement in Technology (GCAT)*. Bangalore, India: IEEE, oct. de 2024, págs. 1-5. ISBN: 9798350376685. DOI: 10.1109/GCAT62922.2024.10923874. URL: <https://ieeexplore.ieee.org/document/10923874/> (visitado 08-12-2025).
- [6] Anjana K M et al. «Implementation of RISC-V Single Cycle Core». En: *International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE)* 14.1 (2025). Peer-reviewed & Refereed, Impact Factor 8.102. ISSN: 2278-1021 (O), 2319-5940 (P). DOI: 10.17148/IJARCCE.2025.14143.
- [7] Shahriar Ahmed y A. B. M. Harun-Ur-Rashid. «Design, Implementation and Verification of Five Stage Pipeline RISC-V Core (RV32I ISA)». En: *2025 International Conference on Electrical, Computer and Communication Engineering (ECCE)*. Chittagong, Bangladesh: IEEE, feb. de 2025, págs. 1-6. ISBN: 9798350357509. DOI: 10.1109/ECCE64574.2025.11013913. URL: <https://ieeexplore.ieee.org/document/11013913/> (visitado 08-12-2025).

## Apéndice A: RTL de la unidad de Control

```

1  `timescale 1ns / 1ps
2  module control_unit(
3      input logic [6:0] opcode,           //instructions [6:0]
4      input logic [2:0] funct_3,        //instructions [14:12]
5      input logic [6:0] funct_7,        //instructions [31:25]
6      input logic zero,               //input from branch(module) if rs1 and rs2 are
7          equal
8      output logic prf_wr_en,         //write enable of prf
9      output logic [2:0] cu_imm_sel,   //control selector to immediate generator(
10         imm_gen)
11     output logic prf_pc_mux_ctrl,   //selector to control input in ALU opeorand
12         1
13     output logic prf_imm_mux_ctrl,  //selector to control input in ALU opereand
14         2
15     output logic [3:0] cu_alu_ctrl,  //selector for ALU operation
16     output logic [1:0] cu_mem_out_mux_sel, //selector for mux data_out,alu_result,
17         next_instruction directions to prf data in
18     output logic cu_data_mem_wr_en, //write enable for data mem
19     output logic cu_pc_add_sel,    //selector to add +4 or +2 to the next
20         instrucion (+2 is for future architecture implementation)
21     output logic branch_taken     //flag in order to evaluate only in BEQ and
22         JAL instructions
23 );
24 //Define the instructions
25 `include "defines.svh"
26 always_comb begin
27     prf_wr_en = 1'b1;                //enable prf write
28     cu_data_mem_wr_en = 1'b0;       //disable data_mem write
29     cu_imm_sel = IMM_I;            //use IMM_I_TYPE decodification
30     prf_pc_mux_ctrl = 1'b0;        //use rs1 as opreand 1 in ALU
31     prf_imm_mux_ctrl = 1'b1;        //don't use rs2 instead use imm_gen as
32         operand 2 in ALU
33     cu_pc_add_sel = 1'b0;          //Add +4 to pc_in
34     cu_mem_out_mux_sel = ALU_TO_PRF; //send the result of ALU to prf data_in
35     cu_alu_ctrl = ALU_ADD;        //ADD operation in ALU
36     branch_taken = 1'b0;           //Never take a branch in I_TYPE so that
37         mux_pc_in is PC+4
38     case (opcode)
39         OPCODE_I_TYPE: begin      //Type I instructions
40             case (funct_3)
41                 //addi
42                 ADDI: begin
43                     cu_alu_ctrl = ALU_ADD;      //ADD operation in ALU
44                 end
45                 //slti
46                 SLTI: begin
47                     cu_alu_ctrl = ALU_SLT;      //SLT operation in ALU
48                 end
49                 //sltiu
50                 SLTIU: begin
51                     cu_alu_ctrl = ALU_SLTU;     //SLTU operation in
52                         ALU
53                 end
54                 //(xori)
55                 XORI: begin
56                     cu_alu_ctrl = ALU_XOR;      //XOR operation in ALU
57                 end
58                 //slti
59                 ORI: begin
60                     cu_alu_ctrl = ALU_OR;      //OR operation in ALU
61                 end
62                 //(ANDI)
63             end
64         end
65     end
66 end

```

```

53          ANDI: begin
54              cu_alu_ctrl = ALU_AND;           //AND operation in ALU
55          end
56          default: begin
57              cu_alu_ctrl = ALU_ADD;           //ADD operation in ALU
58          end
59      endcase
60  end
61 OPCODE_B_TYPE: begin    //Type B instructions
62     prf_wr_en = 1'b0;             //disable prf write
63     cu_imm_sel = IMM_B;          //use IMM_B_TYPE
64     prf_pc_mux_ctrl = 1'b1;      //use rsi as operand 1 in ALU
65     //Type B(BEQ)
66     case (funct_3)
67         BEQ: begin
68             if (zero == 1'b1) begin //This flag is going to be '1' if
69                 rsi == rs2
70                 cu_alu_ctrl = ALU_ADD;           //ADD
71                 branch_taken = 1'b1;           //Select the
72                     direction of the branch (branch taken)
73             end else
74                 branch_taken = 1'b0;
75         end
76         //TODO: Add more types of branches in order to round any
77         program
78         default: begin
79             cu_alu_ctrl = ALU_ADD;           //ADD operation in
80             ALU
81             branch_taken = 1'b0;           //branch not taken
82         end
83     endcase
84 end
85 OPCODE_R_TYPE: begin
86     cu_imm_sel = IMM_NF;          //use IMM_NO_FUNCTION_TYPE
87     prf_imm_mux_ctrl = 1'b0;      //use rs2 as operand 2 in ALU
88     if (funct_7 == 7'b0000000) begin //The instructions below
89         have this code funct_7
90         case (funct_3) //The changes of this types of R operations
91             comes from funct_3
92             //ADDITION OPERATION
93             ADD: begin
94                 cu_alu_ctrl = ALU_ADD;           //ADD operation
95                 in ALU
96             end
97             //LOGICAL LEFT SHIFT
98             SLL: begin
99                 cu_alu_ctrl = ALU_SLL;           //SLL operation
100                in ALU
101            end
102            //SIGNED COMPARISON
103            SLT: begin
104                cu_alu_ctrl = ALU_SLT;           //SLT operation
105            end
106            //UNSIGNED COMPARISON
107            SLTU: begin
108                cu_alu_ctrl = ALU_SLTU;          //SLTU operation
109                in ALU
110            end
111            //XOR
112            XOR_: begin
113                cu_alu_ctrl = ALU_XOR;           //XOR operation
114            end
115        end

```

```

106          //RIGHT LOGICAL SHIFT
107          SRL: begin
108              cu_alu_ctrl =           ALU_SRL;           //SRL operation
109              in ALU
110          end
111          //OR
112          OR_: begin
113              cu_alu_ctrl =           ALU_OR;           //OR operation in
114              ALU
115          end
116          //AND
117          AND_: begin
118              cu_alu_ctrl =           ALU_AND;           //AND operation
119              in ALU
120          end
121          default: begin
122              cu_alu_ctrl =           ALU_ADD;           //ADD operation
123              in ALU
124          end
125      endcase
126  end else if (funct_7 == 7'b0100000) begin    //The instructions
127      below have this code funct_7
128      case(funct_3)
129          //SUBTRACTION OPERATION
130          SUB: begin
131              cu_alu_ctrl =           ALU_SUB;           //SUB operation
132              in ALU
133          end
134          //SIGN-EXTEND SHIFT OPERATION
135          SRA: begin
136              cu_alu_ctrl =           ALU_SRA;           //Arithmetic
137              right shift operation in ALU
138          end
139          default: begin
140              cu_alu_ctrl =           ALU_ADD;           //ADD operation
141              in ALU
142          end
143      endcase
144  end else begin
145      cu_alu_ctrl =           ALU_ADD;           //ADD operation
146      in ALU
147  end
148  OPCODE_J_TYPE: begin    //In this RISCV architecture it only exists jump
149      and link (JAL) operation
150      cu_imm_sel =           IMM_J;           //use
151      IMM_NO_FUNCTION_TYPE
152      prf_pc_mux_ctrl =     1'b1;           //use pc_out as
153      opearand 1 in ALU
154      cu_mem_out_mux_sel =  INSTRUCTION_TO_PRF; //send the result of
155      ALU to prf data_in
156      cu_alu_ctrl =           ALU_ADD;           //ALU operation in
157      ALU
158      branch_taken =        1'b1;           //always take the
159      branch
160  end
161  //TODO: Add more types of J instructions
162  default: begin
163      prf_wr_en =             1'b0;           //enable prf write
164      cu_alu_ctrl =           ALU_ADD;           //ADD operation in ALU
165  end
166 endcase
167 end
168 endmodule

```

Listing 13: Unidad de Control

```
1 module mux_3_to_1 (
2     input logic [31:0] data_out_to_pc ,
3     input logic [31:0] alu_to_mem_addr ,
4     input logic [31:0] data_out_to_mux ,
5     input logic [1:0] sel ,
6     output logic [31:0] data_out
7 );
8     'include "defines.sv"
9     always_comb begin
10         case (sel)
11             ALU_TO_PRF: begin
12                 data_out = alu_to_mem_addr;
13             end
14             DATA_OUT_TO_PRF: begin
15                 data_out = data_out_to_mux;
16             end
17             INSTRUCTION_TO_PRF: begin
18                 data_out = data_out_to_pc;
19             end
20             default: data_out = '0;
21         endcase
22     end
23 endmodule
```

Listing 14: Multiplexor ALU, Memoria, PC

```
1 module plus_4_or_2_mux(
2     input logic sel ,
3     output logic [2:0] instruction_add
4 );
5
6 always_comb begin
7     case (sel)
8         1'b0: begin
9             instruction_add = 3'b100;
10        end
11        1'b1: begin
12            instruction_add = 3'b010;
13        end
14        default: instruction_add = 3'b000;
15    endcase
16 end
17 endmodule
```

Listing 15: Multiplexor auxiliar 4-2 para PC

```
1 module mux #( parameter WIDTH = 32)(
2     input logic [WIDTH-1:0] in1 ,
3     input logic [WIDTH-1:0] in2 ,          // array of N inputs
4     input logic             sel ,          // select
5     output logic [WIDTH-1:0] out           // mux output
6 );
7     assign out = (sel) ? in1 : in2; //0 = in_2
8 endmodule
```

Listing 16: Multiplexor auxiliar

```
1 `timescale 1ns / 1ps
2 module adder #( parameter DATA_WIDTH = 32)(
```

```

3   input logic [2:0] constant_operand,
4   input logic [DATA_WIDTH - 1:0] instruction_addr,
5   output logic [DATA_WIDTH - 1:0] next_instruction
6 );
7   assign next_instruction = constant_operand + instruction_addr;
8 endmodule

```

Listing 17: Sumador interno para PC

```

1 module branch #(parameter DATA_WIDTH = 32)(
2   input logic [DATA_WIDTH - 1: 0]rs_1,
3   input logic [DATA_WIDTH - 1: 0]rs_2,
4   output logic branch_taken
5 );
6
7   assign branch_taken = (rs_1 == rs_2) ? 1'b1 : 1'b0; //Zero flag
8
9 endmodule

```

Listing 18: Multiplexor auxiliar

## Apéndice B: RTL de banco de registro

```
1 'timescale 1ns / 1ps
2 module physical_register_file #(parameter DIR_WIDTH = 10, parameter DATA_WIDTH = 256)
3 (
4     input logic clk,                                //Clock signal
5     input logic arst_n,                            //Reset signal
6     input logic write_en,                          //Write enable signal
7     input logic [DIR_WIDTH - 1:0] read_dir1,      //Read direction 1
8     input logic [DIR_WIDTH - 1:0] read_dir2,      //Read direction 2
9     input logic [DIR_WIDTH - 1:0] write_dir,       //Write direction
10    input logic [DATA_WIDTH - 1:0] write_data,     //Data to write
11    output logic [DATA_WIDTH - 1:0] read_data1,    //Readed Data 1
12    output logic [DATA_WIDTH - 1:0] read_data2,    //Readed Data 2
13    output logic [DATA_WIDTH - 1:0] fibb_out
14 );
15
16 logic [DATA_WIDTH - 1:0] prf [1:(2**DIR_WIDTH) - 1]; //Physical Register File
17           32*32bits
18
19 always_ff (posedge clk, negedge arst_n)begin
20     if (rst_n == 1'b0)begin                         //Condition of reset
21         for (int i = 1 ; i < DATA_WIDTH; i++) begin   //For cicle to reset
22             to 0 all values of memory array
23             prf [i] <= '0;
24         end
25     end else begin
26         if (write_en)begin //If write enable is set and the direction of write
27             is not 0, then write memory array
28             prf [write_dir] <= write_data;           //Memory array
29             assignation
30         end
31     end
32 end
33
34 always_comb begin
35     read_data1 = read_dir1 == '0 ? '0 : prf [read_dir1]; //Assignation of
36             readed data 1 depending of read direction 1
37     read_data2 = read_dir2 == '0 ? '0 : prf [read_dir2]; //Assignation of
38             readed data 2 depending of read direction 2
39 end
40 assign fibb_out = prf[5];
41
42 endmodule
```

Listing 19: Banco de registros

## Apéndice C: RTL de ALU

```
1  `timescale 1ns / 1ps
2  // ALU for basic arithmetic instructions in RV32I
3  module alu #(parameter N = 32)(
4      input logic [N-1:0] operand1,      // First operand, sourced from a register
5      input logic [N-1:0] operand2,      // Second operand sourced from a register
6      //input logic [N-1:0] imm_in,        // Second operand sourced from immediate
7      //generator
8      //input logic [N-1:0] pc_in,        // First operand sourced from program
9      //counter
10     //input logic alusrc_r1,          // ALUsrc selector for the first operand
11     //input logic alusrc_r2,          // ALUsrc selector for the second operand
12     input logic [3:0]   alucontrol,    // Use 4 bits for the opcode at the moment
13     //output logic zero,
14     output logic [N-1:0] alu_result    // ALU result, destiny is another register
15 );
16
17 'include "defines.sv"
18 always_comb begin
19     unique case(alucontrol)
20         ALU_ADD: begin // At the moment we don't care about the carry
21             alu_result = operand1 + operand2;
22         end
23         ALU_SUB: begin
24             alu_result = operand1 - operand2;
25         end
26         ALU_AND: begin
27             alu_result = operand1 & operand2;
28         end
29         ALU_OR: begin
30             alu_result = operand1 | operand2;
31         end
32         ALU_XOR: begin
33             alu_result = operand1 ^ operand2;
34         end
35         ALU_EQ: begin
36             alu_result = operand1 == operand2;
37         end
38         ALU_SLT: begin // Compare sign bit
39             if (operand1[N-1] != operand2[N-1]) begin
40                 // Find which operand has the negative sign
41                 alu_result = operand1[N-1] > operand2[N-1];
42             end else begin
43                 // IF MSB is equal, compare bits directly
44                 alu_result = operand1[N-2:0] < operand2[N-2:0];
45             end
46         end
47         ALU_SLTU: begin
48             alu_result = operand1 < operand2;
49         end
50         ALU_SLL: begin
51             alu_result = operand1 << operand2;
52         end
53         ALU_SRL: begin
54             alu_result = operand1 >> operand2;
55         end
56         ALU_SRA: begin
57             alu_result = $signed(operand1) >>> operand2; // Implementation could
58             // be hard-coded if needed
59         end
60         default: alu_result = '0;
61     endcase
62 end
```

```
60  endmodule
```

Listing 20: Módulo de ALU

## Apéndice D: RTL de Módulo Extensión de signo

```
1 'timescale 1ns / 1ps
2 /////////////////////////////////
3 // Company:
4 // Engineer:
5 //
6 // Create Date: 11/21/2025 10:44:41 PM
7 // Design Name:
8 // Module Name: imm_gen
9 // Project Name:
10 /////////////////////////////////
11
12
13 module imm_gen(
14     input logic [31:0] instr,           // Instructions from program memory
15     input logic [2:0] imm_sel,          // 000 = I-type, 001 = S-type ,010 = B-type, 011 =
16         U-type, 100 = J-type of control unit
17     output logic [31:0] imm_out        // output for ALU and PC
18 );
19
20
21     'include "defines.sv"           //define the imm instructions
22
23     always_comb begin
24         unique case (imm_sel)
25             IMM_I: begin //000 = I-type
26                 imm_out = {{20{instr[31]}}, instr[31:20]}; //signed extension (repeat
27                     20 times the MSB)
28             end
29
30             IMM_S: begin //001 = S-type
31                 imm_out = { {20{instr[31]}},instr[31:25], instr[11:7] }; //signed
32                     extension (repeat 20 times the MSB)
33             end
34
35             IMM_B: begin //010 = B-type
36                 imm_out = { {20{instr[31]}}, {instr[7]}, instr[30:25], instr[11:8],
37                     1'b0 } ;
38             end
39
40             IMM_U: begin // 011 = U-type
41                 imm_out = { instr[31:12], {12{1'b0}} };
42             end
43
44             IMM_J: begin // 100 = J-type
45                 imm_out = { {12{instr[31]}}, {instr[19:12]}, instr[20], instr[30:21],
46                     1'b0 } ;
47             end
48             default: imm_out = IMM_NF;
49         endcase
50     end
51 endmodule
```

Listing 21: Generador de inmediatos

## Apéndice E: RTL de memoria de instrucciones

```
1  `timescale 1ns / 1ps
2  module instruction_memory #(
3      parameter BYTE_WIDTH = 8,           // Ancho de caj n de la memoria (1 Byte)
4      parameter MEM_DEPTH = 1024,        // N mero de renglones de memoria
5      parameter ADDR_WIDTH = 10,         // con 10 bits podemos direccionar 1024 renglones
6      parameter DATA_WIDTH = 32          // Ancho de instrucci n (32 bits)
7  )
8  (
9      input  logic                  clk,
10
11     // Puerto de Lectura (PC)
12     input  logic [ADDR_WIDTH-1:0] rd_addr,    // Direcci n de lectura (PC)
13     output logic [DATA_WIDTH-1:0] rd_data,    // Instrucci n que sale
14
15     // Puerto de escritura (FIFO)
16     input  logic [DATA_WIDTH-1:0] data_in,    // Dato (instrucci n) que viene de la
17         FIFO
18     input  logic [ADDR_WIDTH-1:0] wr_addr,    // Direcci n (10 bits para 1024
19         espacios)
20     input  logic                  w_en       // Write Enable
21 );
22
23     localparam BANK_DEPTH = MEM_DEPTH / 4; // 256 renglones por banco
24
25     // bank3 = byte [31:24] (MSB), bank0 = byte [7:0] (LSB)
26     logic [BYTE_WIDTH-1:0] bank0 [0:BANK_DEPTH-1];
27     logic [BYTE_WIDTH-1:0] bank1 [0:BANK_DEPTH-1];
28     logic [BYTE_WIDTH-1:0] bank2 [0:BANK_DEPTH-1];
29     logic [BYTE_WIDTH-1:0] bank3 [0:BANK_DEPTH-1];
30
31     // 1. Inicializaci n (Limpieza + programa)
32     initial begin : init_memory
33         // Limpia toda la memoria
34         for (int i = 0; i < BANK_DEPTH; i++) begin
35             bank0[i] = {BYTE_WIDTH{1'b0}};
36             bank1[i] = {BYTE_WIDTH{1'b0}};
37             bank2[i] = {BYTE_WIDTH{1'b0}};
38             bank3[i] = {BYTE_WIDTH{1'b0}};
39         end
40     end
41
42     // 2. Escritura paralela en los 4 bancos
43     // [ADDR_WIDTH-1:2] para elegir el rengl n dentro del banco (PC alineado a 4
44     // bytes)
45     always_ff (posedge clk) begin
46         if (w_en) begin
47             bank3[wr_addr[ADDR_WIDTH-1:2]] <= data_in[31:24]; // MSB
48             bank2[wr_addr[ADDR_WIDTH-1:2]] <= data_in[23:16];
49             bank1[wr_addr[ADDR_WIDTH-1:2]] <= data_in[15:8];
50             bank0[wr_addr[ADDR_WIDTH-1:2]] <= data_in[7:0];   // LSB
51         end
52     end
53
54     // 3. Lectura (Combinacional)
55     assign rd_data = {
56         bank3[rd_addr[ADDR_WIDTH-1:2]],
57         bank2[rd_addr[ADDR_WIDTH-1:2]],
58         bank1[rd_addr[ADDR_WIDTH-1:2]],
59         bank0[rd_addr[ADDR_WIDTH-1:2]]
60     };
61 endmodule
```

---

Listing 22: Memoria de instrucciones

## Apéndice F: RTL de memoria del programa

```
1 'timescale 1ns / 1ps
2 module data_memory #(
3     parameter DATA_WIDTH = 32,           // Ancho de instrucción / dato
4     parameter ADDR_WIDTH = 32,          // Ancho de dirección del PC
5     parameter MEM_DEPTH = 1024          // Profundidad (4 renglones: 0,1,2,3)
6 )(
7     input logic clk,
8     input logic w_en,                  // Write Enable desde CU
9     input logic [ADDR_WIDTH-1:0] wr_addr, // Address
10    input logic [DATA_WIDTH-1:0] data_in, // Write data de memoria de datos
11    output logic [DATA_WIDTH-1:0] rd_data // Read Data (Dato de salida) de
12        memoria de datos
13 );
14
15 // Memoria de 32 bits y 1024 renglones
16 logic [DATA_WIDTH-1:0] ram [0:MEM_DEPTH-1];
17
18 // Lectura debe de ser combinacional
19 assign rd_data = ram[wr_addr[ADDR_WIDTH-1:2]];
20
21 // 1. Inicialización (Limpieza)
22 initial begin
23     for (int i=0; i<MEM_DEPTH; i++)
24         ram[i] = {DATA_WIDTH{1'b0}};
25 end
26
27 // Escritura debe de ser secuencial
28 always_ff (posedge clk) begin
29     if (w_en) begin
30         ram[wr_addr[ADDR_WIDTH-1:2]] <= data_in;           // si WE es 1, se escribe
31             en la ram el valor
32     end
33 end
34 endmodule
```

Listing 23: Memoria de datos

## Apéndice G: RTL de UART

```
1 module uart_top (
2     input logic clk,
3     input logic arst_n,
4     input logic rx,
5     input logic next_program,
6     input logic [BAUD_SEL_SIZE - 1 : 0] baud_sel,
7     input logic [ADDR_WIDTH - 1 : 0] rdaddr,
8     output logic prog_rdy,
9     output logic rx_done,
10    output logic [DATA_WIDTH - 1 : 0] data_out,
11    output logic [((DATA_WIDTH / 4) * 7) - 1 : 0] display,
12    output logic [DATA_WIDTH - 1 : 0] read_data,
13    output logic tx,
14    output logic ready,
15    output logic [3:0] state,
16    output logic busy,
17    output logic tx_done,
18    input logic tx_start
19 );
20
21 logic tick;
22 logic inst_rdy;
23 logic [BYTE_WIDTH - 1 : 0] uart_byte;
24 logic [ADDR_WIDTH - 1 : 0] wr_addr;
25 logic [BYTE_WIDTH - 1 : 0] n_instructions;
26
27 shift_register_fsm #(BYTE_WIDTH(BYTE_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
28     shift_register_fsm_i(
29         .clk (clk),
30         .arst_n (arst_n),
31         .next_program(next_program),
32         .w_en (rx_done), // input write enable coming from uart rx_done
33         .data_in (uart_byte), // data_in coming from uart data_out port
34         .data_out (data_out), // data out to be written in the program memory after
35         receiving 4 bytes from uart
36         .wr_addr (wr_addr), // write address for the memory
37         .inst_rdy (inst_rdy), // flag to indicate that a instruction is ready, this is
38         the write enable for the memory
39         .n_instructions(n_instructions),
40         .busy(busy),
41         .state(state),
42         .ready(ready),
43         .prog_rdy (prog_rdy) // flag to indicate that the program has been initialized
44         into the memory
45     );
46
47 baud_rate_generator #(CLK_FREQ(CLK_FREQ)) baud_rate_generator_i(
48     .clk(clk),
49     .arst_n(arst_n),
50     .tick(tick),
51     .baud_sel(baud_sel)
52 );
53
54 transmitter #(BYTE_WIDTH(BYTE_WIDTH)) transmitter_i(
55     .clk(clk),
56     .arst_n(arst_n),
57     .data_in(8'd10),
58     .tick(tick),
59     .tx_start(rx_done),
60     .tx(tx),
61     .tx_done(tx_done)
```

```

59 );
60
61 display_7_segments #(.DATA_WIDTH(DATA_WIDTH)) display_7_segments_i(
62   .data_in (read_data),
63   .display (display)
64 );
65
66 receiver #(.BYTE_WIDTH(BYTE_WIDTH)) receiver_i(
67   .clk(clk),
68   .arst_n(arst_n),
69   .tick(tick),
70   .rx(rx), // RX CONNECTED TO THE TX FROM THE TRANSMITTER MODULE
71   .rx_done(rx_done),
72   .data_out(uart_byte)
73 );
74
75 instruction_memory #(.BYTE_WIDTH(BYTE_WIDTH), .ADDR_WIDTH(ADDR_WIDTH), .MEM_DEPTH(
76   MEM_DEPTH), .DATA_WIDTH(DATA_WIDTH)) instruction_memory_i(
77   .clk(clk),
78   .rd_addr(rdaddr),
79   .rd_data(read_data),
80   .data_in(data_out),
81   .wr_addr(wr_addr),
82   .w_en(inst_rdy)
83 );
84
85 endmodule

```

Listing 24: Módulo uart\_top

```

1 module baud_rate_generator #(parameter CLK_FREQ =50_000_000)(
2   input logic clk,
3   input logic arst_n,
4   input logic [3:0] baud_sel,
5   output logic tick
6 );
7 /*
8  localparam int BAUD_TABLE [0:12] = {
9    1200,
10   2400,
11   4800,
12   9600,
13   19200,
14   28800,
15   38400,
16   57600,
17   76800,
18   115200,
19   230400,
20   460800,
21   921600
22 };
23 };
24 */
25
26
27 logic [31:0] counter_max;
28 logic [31:0] counter;
29
30 /*
31 */
32 always_comb begin
33   if (baud_sel <= 4'd12)
34     counter_max = (CLK_FREQ / (BAUD_TABLE[baud_sel] * 16)) - 1;

```

```

35      else
36          counter_max = (CLK_FREQ / (BAUD_TABLE[0] * 16)) - 1; // default
37      end
38  */
39
40 assign counter_max = (CLK_FREQ / (9600 * 16));
41
42 always_ff (posedge clk or negedge arst_n) begin
43     if(!rst_n) begin
44         tick    <= 1'b0;
45         counter <= 32'd0;
46     end else begin
47         if (counter == counter_max) begin
48             counter <= 32'd0;
49             tick    <= 1'b1;
50         end else begin
51             counter <= counter + 1;
52             tick    <= 1'b0;
53         end
54     end
55 end
56
57 endmodule

```

Listing 25: Módulo baud\_rate\_generator

```

1 module receiver #(parameter BYTE_WIDTH = 8)(
2     input logic clk,
3     input logic arst_n,
4     input logic tick, // TICK COMING FROM THE BAUD RATE GENERATOR
5     input logic rx, // INPUT SERIAL COMING FROM THE TRANSMITTER TX
6     output logic rx_done, // DONE SIGNAL TO INDICATE THAT THE BYTE HAS BEEN RECEIVED
7     output logic [BYTE_WIDTH - 1 : 0] data_out // OUTPUT SIGNAL FOR THE BYTE RECEIVED
8 );
9
10 localparam BIT_SAMPLING = 15;
11 localparam HALFBIT_SAMPLING = 7;
12
13 logic [4:0] nbits;
14 logic [4:0] nbits_next;
15 logic [4:0] oversampling_count;
16 logic [4:0] oversampling_count_next;
17 logic [BYTE_WIDTH - 1 : 0] data_out_reg;
18 logic [BYTE_WIDTH - 1 : 0] data_out_reg_next;
19 logic rx_done_next;
20
21 typedef enum logic [2:0] {IDLE = 3'b000, START = 3'b001, DATA = 3'b010, STOP = 3'b011
22     } state_type;
23
24 state_type state_reg, state_next;
25
26 always_ff (posedge clk or negedge arst_n) begin
27     if(!rst_n) begin
28         state_reg <= IDLE;
29     end else begin
30         oversampling_count <= oversampling_count_next;
31         state_reg <= state_next;
32         nbits <= nbits_next;
33         data_out_reg <= data_out_reg_next;
34         rx_done <= rx_done_next;
35     end
36
37 always_comb begin
38     oversampling_count_next = oversampling_count;

```

```

39      nbits_next          = nbits;
40      data_out_reg_next   = data_out_reg;
41      state_next          = state_reg;
42      rx_done_next        = 1'b0;
43
44      case(state_reg)
45          IDLE: begin
46              rx_done_next = 1'b0;
47              oversampling_count_next = '0;
48              data_out_reg_next   = data_out_reg;
49              nbits_next          = '0;
50              if (!rx) begin
51                  data_out_reg_next = '0;
52                  state_next       = START;
53              end
54          end
55
56          START: begin
57              if (tick) begin
58                  if (oversampling_count == BIT_SAMPLING) begin
59                      oversampling_count_next = '0;
60                      nbits_next          = '0;
61                      data_out_reg_next   = '0;
62                      state_next          = DATA;
63                  end else begin
64                      oversampling_count_next = oversampling_count + 1'b1;
65                      if (oversampling_count == HALFBIT_SAMPLING) begin
66                          state_next = (!rx) ? START : IDLE;
67                  end
68              end
69          end
70      end
71
72      DATA: begin
73          if (tick) begin
74              if (oversampling_count == BIT_SAMPLING) begin
75                  oversampling_count_next = '0;
76                  if (nbits == (BYTE_WIDTH - 1'b1)) begin
77                      state_next = STOP;
78                  end else begin
79                      nbits_next = nbits + 1'b1;
80                  end
81              end else begin
82                  oversampling_count_next = oversampling_count + 1'b1;
83                  if (oversampling_count == HALFBIT_SAMPLING) begin
84                      data_out_reg_next = {rx, data_out_reg[BYTE_WIDTH-1:1]};
85                  end
86              end
87          end
88      end
89
90      STOP: begin
91          if (tick) begin
92              if (rx) begin
93                  if (oversampling_count == BIT_SAMPLING) begin
94                      state_next = IDLE;
95                      oversampling_count_next = '0;
96                  end else begin
97                      oversampling_count_next = oversampling_count + 1'b1;
98                  end
99                  if (oversampling_count == HALFBIT_SAMPLING) begin
100                     rx_done_next = 1'b1;
101                 end
102             end else begin
103                 state_next = IDLE;

```

```

104           oversampling_count_next = '0;
105       end
106   end
107 endcase
110 end
112 assign data_out = data_out_reg;
113
114 endmodule

```

Listing 26: Módulo receiver

```

1 module shift_register_fsm#(parameter DATA_WIDTH = 32, parameter BYTE_WIDTH = 8,
2   parameter ADDR_WIDTH = 10)
3   input logic clk,
4   input logic arst_n,
5   input logic next_program,
6   input logic w_en, // input write enable coming from uart rx_done
7   input logic [BYTE_WIDTH - 1 : 0] data_in, // data_in coming from uart data_out
8   port
9     output logic [DATA_WIDTH - 1 : 0] data_out, // data out to be written in the
10    program memory after receiving 4 bytes from uart
11    output logic [ADDR_WIDTH - 1 : 0] wr_addr, // write address for the memory
12    output logic inst_rdy, // flag to indicate that a instruction is ready, this is
13    the write enable for the memory
14    output logic [BYTE_WIDTH - 1 : 0] n_instructions,
15    output logic busy,
16    output logic [3:0] state,
17    output logic ready,
18    output logic prog_rdy // flag to indicate that the program has been initialized
19    into the memory
20 );
21
22 // internal counters
23 logic [BYTE_WIDTH - 1 : 0] n_of_instructions;
24 logic [BYTE_WIDTH - 1 : 0] n_of_instructions_next;
25
26 logic [ADDR_WIDTH - 1 : 0] instruction_counter;
27 logic [ADDR_WIDTH - 1 : 0] instruction_counter_next;
28
29 logic [DATA_WIDTH - 1 : 0] temp_data_out;
30 logic [DATA_WIDTH - 1 : 0] temp_data_out_next;
31
32 logic [BYTE_WIDTH - 1 : 0] received_byte;
33
34 // states for the fsm
35 typedef enum logic [3:0]{
36   WAIT_PARAMS = 4'b0000,
37   WAIT_BYTETO = 4'b0001,
38   WAIT_BYTETO1 = 4'b0010,
39   WAIT_BYTETO2 = 4'b0011,
40   WAIT_BYTETO3 = 4'b0100,
41   WRITE_INSTRUCTION = 4'b0101,
42   DONE = 4'b0110,
43   CLEAN_MEMORY = 4'b0111
44 } state_type;
45
46 state_type state_reg, state_next;
47
48 // reset all the signals and assign the next state sequential logic
49 always_ff (posedge clk or negedge arst_n) begin
50   if (!rst_n) begin

```

```

47     state_reg <= WAIT_PARAMS;
48     instruction_counter <= '0;
49     n_of_instructions <= '0;
50     temp_data_out <= '0;
51   end else begin
52     state_reg <= state_next;
53     instruction_counter <= instruction_counter_next;
54     n_of_instructions <= n_of_instructions_next;
55     temp_data_out <= temp_data_out_next;
56   end
57 end
58
59 always_comb begin
60   //prog_rdy = 1'b0; // flag for program initialized set to 0
61   n_of_instructions_next = n_of_instructions; // default value
62   state_next = state_reg; // default state for state_next
63   instruction_counter_next = instruction_counter; // default value
64   temp_data_out_next = temp_data_out; // default value
65
66   case(state_reg)
67
68     WAIT_PARAMS: begin // receive a byte from the uart to define the number of
69       instructions to write into the memory
70       if(w_en) begin
71         n_of_instructions_next = data_in;
72         instruction_counter_next = '0;
73         temp_data_out_next = '0;
74         state_next = WAIT_BYTETO;
75       end else begin
76         state_next = state_reg;
77         n_of_instructions_next = n_of_instructions;
78       end
79     end
80
81     WAIT_BYTETO: begin // receive the 1st byte of the instruction
82       if(w_en) begin
83         temp_data_out_next = '0;
84         temp_data_out_next[7:0] = data_in;
85         state_next = WAIT_BYTE1;
86       end else begin
87         state_next = state_reg;
88       end
89     end
90
91     WAIT_BYTE1: begin // receive the 2nd byte of the instruction
92       if(w_en) begin
93         temp_data_out_next[15:8] = data_in;
94         state_next = WAIT_BYTE2;
95       end else begin
96         state_next = state_reg;
97       end
98     end
99
100    WAIT_BYTE2: begin // receive the 3rd byte of the instruction
101      if(w_en) begin
102        temp_data_out_next[23:16] = data_in;
103        state_next = WAIT_BYTE3;
104      end else begin
105        state_next = state_reg;
106      end
107    end
108
109    WAIT_BYTE3: begin // receive the 4th and last byte of the instruction
110      if(w_en) begin
111        temp_data_out_next[31:24] = data_in;

```

```

111         state_next = WRITE_INSTRUCTION;
112     end else begin
113         state_next = state_reg;
114     end
115 end
116
117 WRITE_INSTRUCTION: begin // state to write instruction once the shifter has
118     received 4 bytes = 32 bits instruction
119     if(instruction_counter == ((n_of_instructions - 1) * 4)) begin
120         state_next = DONE;
121     end else begin
122         instruction_counter_next = instruction_counter + 3'd4;
123         state_next = WAIT_BYTE0;
124     end
125 end
126
127 DONE: begin // done state to set program ready flag to 1
128     instruction_counter_next = '0;
129     temp_data_out_next = {DATA_WIDTH{1'b0}};
130     if(next_program) begin
131         state_next = CLEAN_MEMORY;
132     end else begin
133         state_next = DONE;
134     end
135 end
136
137 CLEAN_MEMORY: begin
138     temp_data_out_next = {DATA_WIDTH{1'b0}};
139     if(instruction_counter == ((n_of_instructions - 1 ) * 4)) begin
140         state_next = WAIT_PARAMS;
141         n_of_instructions_next = '0;
142     end else begin
143         instruction_counter_next = instruction_counter + 3'd4;
144         state_next = state_reg;
145     end
146 end
147
148 endcase
149 end
150 assign prog_rdy = (state_reg == DONE);
151 assign inst_rdy = (state_reg == WRITE_INSTRUCTION) || (state_reg == CLEAN_MEMORY);
152 assign data_out = temp_data_out;
153 assign wr_addr = instruction_counter;
154 assign state = state_reg;
155 assign n_instructions = n_of_instructions;
156 assign ready = (state_reg == WAIT_PARAMS);
157 assign busy = ~ready;
158
159 endmodule

```

Listing 27: Módulo shift\_register\_fsm

```

1 import serial
2 import time
3
4 imm = int(input("Enter the # iteration for the fibonacci series: "))
5
6 def fibonacci(n):
7     serie = [0, 1]
8     for _ in range(2, n):
9         serie.append(serie[-1] + serie[-2])
10
11     hex_result = format(serie[n-1], 'x')
12     print(f"El resultado de la serie Fibonacci para el numero ingresado es: Hex: {hex_result}")

```

```

        hex_result}, int: {serie[-1]}")
13     serie = [0, 1]
14     return None
15
16 fibonacci(imm)
17
18 port = 'COM4'
19 baudrate = 9600
20 timeout = 1
21
22 # Rango del inmediato que quieres enviar
23 IMM_START = 0x0000
24 IMM_END    = 0x0OFF    # cambialo al rango que necesites
25
26 try:
27     ser = serial.Serial(port, baudrate, timeout=timeout)
28     time.sleep(0.05)
29
30     # Separar el inmediato en dos bytes (little-endian)
31     imm_high = (imm & 0x0F) << 4          # LSB
32     imm_low  = (imm >> 4) & 0xFF # MSB
33
34     # Construye payload dinamico
35     payload = [
36         bytes([0x13, 0x05, 0x00, 0x00]),
37         bytes([0x93, 0x05, 0x10, 0x00]),
38         bytes([0x13, 0x06, imm_high, imm_low]), # <- SE ACTUALIZA AQUI
39         bytes([0x63, 0x0C, 0x06, 0x00]),
40         bytes([0xB3, 0x02, 0xB5, 0x00]),
41         bytes([0x13, 0x85, 0x05, 0x00]),
42         bytes([0x93, 0x85, 0x02, 0x00]),
43         bytes([0x13, 0x06, 0xF6, 0xFF]),
44         bytes([0x6F, 0xF0, 0xDF, 0xFE]),
45         bytes([0x6F, 0x00, 0x00, 0x00])
46     ]
47
48     print(f"\n==== Enviando inmediato {hex(imm)} -> HIGH={hex(imm_high)}, LOW={hex(imm_low)} ===")
49
50     # Enviar numero de instrucciones (10 -> 0x0A)
51     n_instr = b'\x0A'
52     ser.write(n_instr)
53     time.sleep(0.005)
54
55     # Enviar instrucciones una por una
56     for frame in payload:
57         ser.write(frame)
58         print(f"Sent: {frame}")
59         time.sleep(0.005)
60
61     # Pausa opcional entre iteraciones
62     time.sleep(0.05)
63
64     ser.close()
65     print("Serial port closed.")
66
67 except Exception as e:
68     print("ERROR:", e)

```

Listing 28: Script en python para UART

## Apéndice H: Testbench del microprocesador

```
1  module microprocessor_tb ();
2
3 // `include "defines.svh"
4
5     bit clk;
6     bit arst_n;
7
8
9 // para el testbench (randomizados)
10    logic [DIR_WIDTH-1:0] rd, rs1, rs2;
11    logic [12:0] imm_bq;
12    logic [20:0] imm_jal;
13
14    logic [DATA_WIDTH-1:0] instruction_tb, instruction_tb_addi, instruction_tb_add,
15        instruction_tb_beq, instruction_tb_jal;
16    logic [DIR_WIDTH-1:0] rd_tb, rs1_tb, rs2_tb;
17    logic [11:0] imm_tb;
18
19 // para el BIND
20    logic [DATA_WIDTH-1:0] alu_result;
21    logic [DATA_WIDTH-1:0] pc_out, pc_imm; // Se ales para conectar al bind
22    logic memread, memwrite, memtoreg;
23
24 //CHECKS
25    logic [31:0] pc_before;
26    logic [31:0] imm_check;
27    logic [31:0] rd_expected;
28    logic [31:0] beq_taken;
29    logic [31:0] beq_notaken;
30    logic equal;
31    logic [6:0] count_add, count_addi, count_beq, count_jal, count_add_g,
32        count_addi_g, count_beq_g, count_jal_g;
33
34    always #5ns clk = !clk;
35    assign #10ns arst_n = 1'b1;
36
37    microprocessor_if tbprocessor_if(clk, arst_n);
38    //microprocessor_top tbprocessor_top(clk, arst_n);
39
40    microprocessor_top microprocessor_i (
41        .clk(clk),
42        .arst_n(arst_n),
43        .prog_ready(1'b1),
44        .w_en(1'b0),
45        .instruction(tbprocessor_if.instruction)
46    );
47
48    `define MEM_PATH microprocessor_i.instruction_memory_i
49    `define ALU_PATH microprocessor_i.alu_i
50    `define BANK_REG_PATH microprocessor_i.prf_i
51    `define PC_PATH microprocessor_i.pc_i
52    `define IMM_GEN_PATH microprocessor_i.imm_gen_i
53    `define MUX_IMM_PATH microprocessor_i.mux_imm_gen_i
54    `define MUX_PC_PATH microprocessor_i.mux_pc_i
55    `define CU_PATH microprocessor_i.control_unit_i
56    `define BRANCH_PATH microprocessor_i.branch_i
57
58    typedef enum bit [6:0] {
59        addi = 7'b0010011,
60        add  = 7'b0110011,
61        beq  = 7'b1100011,
```

```

61     jal  = 7'b1101111
62 } operation;
63
64 operation current_instruction;
65
66 initial begin
67 count_add = 0; count_addi= 0; count_beq = 0; count_jal = 0;
68 count_add_g = 0; count_addi_g = 0; count_beq_g = 0; count_jal_g = 0;
69
70 wait (arst_n);
71 $display("--- Decodificador de Instrucciones (RISC-V Opcode) ---");
72
73 repeat (100) (posedge clk) begin
74 randcase
75   1 : begin // ADDI
76 instruction_tb_addi = '0; instruction_tb_addr= '0; instruction_tb_beq= '0;
77   instruction_tb_jal= '0;
78     std::randomize(rd,rs1);
79     tbprocessor_if.write_addi_instr(rd,rs1);
80     current_instruction = tbprocessor_if.instruction[6:0];
81     rd_tb = tbprocessor_if.instruction[11:7];
82     rs1_tb = tbprocessor_if.instruction[19:15];
83     imm_tb = tbprocessor_if.instruction[DATA_WIDTH-1:DATA_WIDTH-12];
84     instruction_tb_addi = tbprocessor_if.instruction;
85     count_addi = count_addi + 1;
86     (posedge clk); // Esperamos un ciclo
87       //if ('ALU_PATH.alu_result !== ('ALU_PATH.operand1 + 'ALU_PATH.
88         operand2)) begin
89         // $error("ERROR EN ADDI!! ALU: esperado %0d + %0d = %0d,
90           obtenido = %0d",
91           // 'ALU_PATH.operand1, 'ALU_PATH.operand2,
92           // 'ALU_PATH.operand1 + 'ALU_PATH.operand2, 'ALU_PATH.
93             alu_result);
94       //end
95       //if ('BANK_REG_PATH.write_dir !== rd) begin
96         // $error("ERROR EN ADDI!! Dirección esperada = %0d, dirección
97           escrita = %0d",
98             rd, 'BANK_REG_PATH.write_dir);
99       //end else begin
100         //count_addi_g = count_addi_g + 1;
101       //end
102   end
103
104   1 : begin // ADD
105 instruction_tb_addi = '0; instruction_tb_addr= '0; instruction_tb_beq= '0;
106   instruction_tb_jal= '0;
107     std::randomize(rs1,rs2,rd);
108     tbprocessor_if.write_add_instr(rs1,rs2,rd);
109     current_instruction = tbprocessor_if.instruction[6:0];
110     rd_tb = tbprocessor_if.instruction[11:7];
111     rs1_tb = tbprocessor_if.instruction[19:15];
112     rs2_tb = tbprocessor_if.instruction[24:20];
113     instruction_tb_addr = tbprocessor_if.instruction;
114     count_add =count_add +1;
115     (posedge clk);
116       //if ('ALU_PATH.alu_result !== ('ALU_PATH.operand1 + 'ALU_PATH.
117         operand2)) begin
118         // $error("ERROR EN ADD!! ALU: esperado %0d + %0d = %0d, obtenido %0
119           d",
120           // 'ALU_PATH.operand1, 'ALU_PATH.operand2,
121           // 'ALU_PATH.operand1 + 'ALU_PATH.operand2, 'ALU_PATH.
122             alu_result);
123       //end
124       //if ('BANK_REG_PATH.write_dir !== rd) begin

```

```

116      // $error("ERROR EN ADD!! Dirección esperada = %0d, dirección
117      // escrita = %0d",
118      //         rd, `BANK_REG_PATH.write_dir);
119      //end else begin
120      //count_add_g = count_add_g + 1;
121      //end
122    end
123
124    1 : begin //BEQ
125      instruction_tb_addi = '0; instruction_tb_addr= '0; instruction_tb_beq= '0;
126      instruction_tb_jal= '0;
127      std::randomize(rs1,rs2,imm_bq);
128      tbprocessor_if.write_beq_instr(rs1,rs2,imm_bq);
129      current_instruction = tbprocessor_if.instruction[6:0];
130      rs1_tb = tbprocessor_if.instruction[19:15];
131      rs2_tb = tbprocessor_if.instruction[24:20];
132      instruction_tb_beq = tbprocessor_if.instruction;
133      pc_before = $signed(`PC_PATH.pc_out);
134      equal = (rs1 == rs2);
135      imm_check = $signed({imm_bq[12], imm_bq[10:5], imm_bq[4:1], imm_bq
136                      [11], 1'b0});
137      beq_taken = $signed(pc_before) + imm_check;
138      beq_notaken = $signed(pc_before) + 4;
139      rd_expected = equal ? $signed(beq_taken) : $signed(beq_notaken);
140      count_beq = count_beq + 1;
141      (posedge clk);
142      //if ($signed(`PC_PATH.pc_out) != $signed(rd_expected)) begin
143      // $error("ERROR BEQ FALL !\n PC actual = %d \n PC anterior
144      // = %d \n %d, %d -> condición: %0s\n Inmediato B = %0d (%b)
145      // Esperado = %d (%0s)",
146      // $signed(`PC_PATH.pc_out), $signed(pc_before),
147      // rs1, rs2, equal ? "IGUALES -> TOMADO" : "DIFERENTES -> NO
148      // TOMADO",
149      // $signed(imm_check), $signed(imm_bq),
150      // $signed(rd_expected),
151      // equal ? "SALTO TOMADO" : "SALTO NO TOMADO");
152      //end else begin
153      // count_beq_g = count_beq_g + 1;
154    end
155
156    5 : begin //JAL
157      instruction_tb_addi = '0; instruction_tb_addr= '0; instruction_tb_beq= '0;
158      instruction_tb_jal= '0;
159      std::randomize(rd,imm_jal);
160      tbprocessor_if.write_jal_instr (rd,imm_jal);
161      current_instruction = tbprocessor_if.instruction[6:0];
162      rd_tb = tbprocessor_if.instruction[11:7];
163      instruction_tb_jal = tbprocessor_if.instruction;
164      pc_before = $signed(`PC_PATH.pc_out);
165      imm_check = $signed({{12{imm_jal[20]}}, imm_jal[20:0], 1'b0});
166      rd_expected = $signed(pc_before) + $signed(imm_check);
167      count_jal = count_jal + 1;
168      (posedge clk);
169      //if ($signed(`PC_PATH.pc_out) != $signed(rd_expected)) begin
170      // $error("ERROR JAL: Salto incorrecto!\n PC actual = %d\n
171      // Esperado = PC(%d) + imm(%0d) = %d",
172      // $signed(`PC_PATH.pc_out), $signed(pc_before), $signed(
173      // imm_check), $signed(rd_expected));
174    end
175    //if ($signed(`BANK_REG_PATH.write_dir) != $signed(rd_expected))
176    // begin
177    // $error("ERROR JAL: write_dir = %0d, esperado rd=%0d", $signed(`BANK_REG_PATH.write_dir), $signed(rd_expected));

```

```

170         //end
171         //if ($signed('BANK_REG_PATH.write_data) !== $signed(pc_before) + 4)
172             begin
173                 // $error("ERROR JAL: No guard PC+4 en rd!\n    Escrito = %d\n
174                 Esperado = PC+4 = %d",
175                 //      $signed('BANK_REG_PATH.write_data), $signed(pc_before) +
176                 4);
177             //end else begin
178             //    count_jal_g = count_jal_g + 1;
179         //end
180     endcase
181 end
182
183
184 initial begin // Initial block to open shared memory and probe signals
185     $shm_open("shm_db");
186     $shm_probe("ASMTR");
187 end
188
189 initial begin // Timeout thread
190     #1ms;
191     $finish;
192 end
193
194
195
196
197 logic probe_signal_beq;
198 logic [31:0] probe1, probe2;
199 assign probe1 = microprocessor_i.prf_i.prf[rs1];
200 assign probe2 = microprocessor_i.prf_i.prf[rs2];
201 assign probe_signal_beq = microprocessor_i.prf_i.prf[rs1] == microprocessor_i.
202     prf_i.prf[rs2];
203
204 //ADDI
205 // Comprueba que si rd = 0, se escribe 0. Si rd != 0, se escribe rs1_val_expected
206 // + Inmediato.
207 'AST(uC, instruction_tb_addi,
208     current_instruction == OPCODE_I_TYPE[6:0] |>,
209     $past(rd_tb) == 5'd0 ?
210         // Caso: rd es X0 (debe escribir 0)
211         $signed('BANK_REG_PATH.prf[$past(rd_tb)]) == 32'd0 :
212         // Caso: rd es un registro v lido (debe escribir la suma)
213         $signed('BANK_REG_PATH.prf[$past(rd_tb)]) ==
214             // rs1_val: Si rs1 == 0, se usa 0; si no, se usa el valor le do
215             ($past(rs1_tb) == 5'd0 ? 32'd0 : $past($signed('BANK_REG_PATH.
216                 read_data1)))
217             + $past($signed('IMM_GEN_PATH.imm_out))
218         )
219
220 // ADD
221 // Comprueba que si rd = 0, se escribe 0. Si rd != 0, se escribe rs1_val_expected
222 // + rs2_val_expected.
223 'AST(uC, instruction_tb_add,
224     current_instruction == OPCODE_R_TYPE[6:0] |>,
225     $past(rd_tb) == 5'd0 ?
226         // Caso: rd es X0 (debe escribir 0)

```

```

225     $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == 32'd0 :
226     // Caso: rd es un registro v lido (debe escribir la suma de los
227     // operandos corregidos)
228     $signed(`BANK_REG_PATH.prf[$past(rd_tb)]) == (
229         // rs1_val: Si rs1 == 0, se usa 0; si no, se usa read_data1
230         ($past(rs1_tb) == 5'd0 ? 32'd0 : $past($signed(`BANK_REG_PATH.
231             read_data1)))
232         +
233         // rs2_val: Si rs2 == 0, se usa 0; si no, se usa read_data2
234         ($past(rs2_tb) == 5'd0 ? 32'd0 : $past($signed(`BANK_REG_PATH.
235             read_data2)))
236     )
237
238 //BEQ
239     'AST(uC, instruction_tb_beq,
240         current_instruction == OPCODE_B_TYPE[6:0] |=>,
241         $past(`BRANCH_PATH.rs_1) == $past(`BRANCH_PATH.rs_2) ?
242             $signed(microprocessor_i.pc_i.pc_in) == $past(microprocessor_i.pc_i.pc_out) +
243                 $past(`IMM_GEN_PATH.imm_out) // branch taken
244             $signed(microprocessor_i.pc_i.pc_in) == $past(microprocessor_i.pc_i.pc_out) +
245                 32'd4 // branch not taken
246     )
247
248     // JAL
249     // Salto del PC
250     'AST(uC, instruction_tb_jal,
251         current_instruction[6:0] == OPCODE_J_TYPE[6:0] |=>,
252             microprocessor_i.pc_i.pc_in == $past(microprocessor_i.pc_i.pc_out +
253                 microprocessor_i.imm_gen_i.imm_out)
254     )

```

Listing 29: Testbench para Microprocesador