



instituto
nacional de
astrofísica,
óptica y
electrónica

TRNGs para generación de secuencias muy largas

por

Ciro Fabian Bermudez Marquez

Tesis presentada en cumplimiento parcial de los requisitos para
el grado de:

Maestría en Ciencias en la Especialidad de Electrónica

en

Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE)

Mayo, 2023

Santa María de Tonantzintla, Puebla

Asesores:

Dr. Esteban Tlelo Cuautle
Departamento de Electrónica INAOE

Dr. Cuauhtemoc Mancillas López
Departamento de Computación CINVESTAV IPN

©INAOE 2023

Todos los derechos reservados

El autor otorga al INAOE el permiso para reproducir y
distribuir copia de esta tesis en su totalidad o en partes
mencionando la fuente.



Agradecimientos

- Para mi familia, mis hermanos Efraín, Alejandro y mis padres Juana y Efrain, les agradezco con todo mi corazón su apoyo.
- A mis compañeros de laboratorio, Victor, Juan y Julio, gracias por ayudarme resolviendo mis dudas y por hacer el laboratorio un lugar de trabajo agradable.
- A mi novia Julisa, gracias por regañarme y hacerme ver mis virtudes y mis defectos y sobre empujarme a seguir adelante.
- A mi asesor Esteban, gracias por su infinita paciencia y apoyo, el trabajo que realizamos juntos representa el inicio de mi carrera profesional.

Índice general

Agradecimientos	I
Índice general	v
Índice de figuras	vii
Índice de tablas	ix
Lista de códigos	xI
Lista de abreviaciones	xIII
Resumen	xv
1. Introducción	1
1.1. Clasificación de los RNGs	2
1.2. Estructura de los TRNG	5
1.3. Condiciones de funcionamiento y calidad de la salida del TRNG	6
1.4. Trabajos actuales	7
1.5. Objetivos	9
1.5.1. Objetivo general	9
1.5.2. Objetivos específicos	10
2. Generadores de números aleatorios (RNGs)	11
2.1. Parámetros de evaluación de los TRNG	11
2.1.1. Parámetros relacionados con la calidad	11
2.1.1.1. Fuentes de aleatoriedad en dispositivos lógicos	11
2.1.1.2. Métodos de extracción de aleatoriedad	13
2.1.1.3. Técnicas de postprocesamiento	13
2.1.1.4. Tasa de bits de salida	16
2.1.2. Parámetros relacionados con la seguridad	17
2.1.2.1. Modelado matemático de TRNG	17

2.1.2.2. Comprobabilidad	17
2.1.2.3. Evaluación de seguridad	17
2.1.3. Parámetros relacionados con el diseño	18
2.1.3.1. Uso de recursos	18
2.1.3.2. Consumo de energía	18
2.1.3.3. Viabilidad en FPGAs	19
2.1.3.4. Automatización del diseño	19
2.2. Fuentes de aleatoriedad en los dispositivos lógicos	20
2.2.1. Jitter del reloj	20
2.2.1.1. Jitter de fase	21
2.2.1.2. Jitter de periodo	22
2.2.1.3. Jitter de ciclo a ciclo	23
2.2.1.4. Composición del jitter	23
2.2.1.5. Extracción de la aleatoriedad del jitter del reloj	25
2.2.2. Metaestabilidad	26
2.2.2.1. Metaestabilidad en dispositivos lógicos	26
2.2.2.2. Metaestabilidad oscilatoria	27
2.2.3. Modelos estocásticos y pruebas específicas	28
2.3. Estándares de diseño y certificación de TRNG	29
2.3.1. Resumen de los requisitos del AIS-20/31	31
2.3.1.1. Clase PTG.1 TRNG	31
2.3.1.2. Clase PTG.2 TRNG	33
2.3.1.3. Clase PTG.3 TRNG	34
2.3.1.4. Clase DRG.1 DRNG	35
2.3.2. Resumen de los requisitos del NIST 800-90B	35
2.3.3. Conclusiones de la certificación de seguridad TRNG	37
2.4. Arquitecturas de núcleos TRNGs en FPGA	38
2.4.1. Elementary ring oscillator based TRNG (ERO-TRNG)	39
2.4.2. Coherent sampling ring oscillator based TRNG (COSO-TRNG)	40
2.4.3. Multi-ring oscillator based TRNG (MURO-TRNG)	42
2.4.4. Transient effect ring oscillator based TRNG (TERO-TRNG)	43
2.4.5. Self-timed ring based TRNG (STR-TRNG)	44
2.4.6. PLL based TRNG (PLL-TRNG)	45
2.4.7. Comparación entre nucleos TRNG	45
3. Mapas caóticos	47
3.1. Definición de caos	47
3.2. Puntos fijos y estabilidad lineal	48
3.3. Mapa logístico	48

3.3.1. Diagrama de cobwebs	51
3.3.2. Análisis cualitativo del mapa logístico	52
3.3.3. Análisis teórico del mapa logístico	58
4. Implementación de TRNG híbrido	61
4.1. Aritmética de punto fijo	61
4.2. Mapa caótico	63
4.3. Análisis de punto fijo	68
4.4. Simulador de arquitecturas digitales en C	69
4.5. Diseño de mapa caótico en VHDL	71
4.6. Multiplicador de una sola constante (SCM)	73
4.7. Diseño de generador de semillas	74
4.8. Teoría de FPGAs en Xilinx	75
4.8.1. Primitivas	76
4.8.1.1. LUT1: 1-Bit Look-Up Table with General Output . . .	76
4.8.1.2. OBUFDS: Differential Signaling Output Buffer	77
5. Resultados experimentales	79
5.1. Comunicación RS232	79
5.2. TRNG híbrido	79
5.3. Pruebas estadísticas	81
5.4. Uso de recursos y velocidad	83
6. Conclusiones	85
A. Códigos	87
A.1. Códigos en C	87
A.2. Códigos en VHDL de mapa caótico	94
Bibliografía	99

Índice de figuras

1.1.	Estructura general de un TRNG [3]	6
2.1.	Jitter del reloj.	21
2.2.	Fluctuaciones del nivel de referencia originadas por ruidos analógicos que provocan jitter del reloj en los circuitos digitales. [33]	21
2.3.	Ilustración de la jitter de fase del segundo flanco ascendente de la señal de reloj. [33]	22
2.4.	Ilustración de jitter de periodo de una señal de reloj real comparada con el reloj ideal.	22
2.5.	Ilustración de jitter de ciclo a ciclo.	23
2.6.	Composición del jitter del reloj.	24
2.7.	Extracción de aleatoriedad de la señal de reloj con jitter mediante su muestreo en el flanco ascendente de la señal de reloj de referencia.	25
2.8.	Elementary ring oscillator (ERO-TRNG)	26
2.9.	Metaestabilidad de un lanzamiento de moneda.	26
2.10.	Estructura interna de un TERO	28
2.11.	Clase PTG.1 TRNG.	32
2.12.	Clase PTG.2 TRNG.	33
2.13.	Clase PTG.3 TRNG.	34
2.14.	Clase DRNG.1.	36
2.15.	Arquitectura del núcleo ERO-TRNG.	39
2.16.	Arquitectura del núcleo COSO-TRNG.	41
2.17.	Arquitectura del núcleo MURO-TRNG.	42
2.18.	Arquitectura del núcleo TERO-TRNG.	43
2.19.	Arquitectura del núcleo STR-TRNG.	44
2.20.	Arquitectura del núcleo PLL-TRNG.	45
3.1.	Ejemplo de mapeo en una dimensión.	49
3.2.	Gráfica de mapa logístico con $A = 4$	50
3.3.	Diagrama de cobwebs de mapa logístico con $A = 2.8$ y $x_0 = 0.2$	51

3.4. Serie de tiempo de mapa logístico con $A = 2.8$	52
3.5. Diagrama de cobwebs de mapa logístico con $A = 3.2$ y $x_0 = 0.1$	54
3.6. Diagrama de bifurcación de mapa logístico [50]	56
3.7. Diagrama de cobwebs de mapa logístico con $A = 4.0$ y $x_0 = 0.1$	57
3.8. Serie de tiempo de mapa logístico con $A = 4.0$	58
3.9. Gráfica de mapa logístico para distintos valores de A	59
 4.1. Diferentes atractores caóticos del mapa bidimensional generados en aritmética de punto flotante.	65
4.2. Dominio de atracción de diferentes atractores caóticos del mapa bidimensional generados en aritmética de punto flotante.	67
4.3. Simulación del mapa caótico en punto fijo, Atractor 1.	70
4.4. Distribución de 100 millones de palabras de 16 bits.	71
4.5. Diagrama de bloques del mapa caótico.	72
4.6. Máquina de estados de mapa caótico.	73
4.7. Ejemplo de multiplicación de una sola constante.	73
4.8. Generador de semilla.	75
4.9. Esquemático de LUT1.	76
4.10. Esquemático de OBUFDS.	77
 5.1. Diagrama de bloques de transmisión RS232.	80
5.2. Máquina de estados para la transmisión RS232.	80
5.3. Diagrama de bloques de TRNG híbrido.	80
5.4. Máquina de estados de TRNG híbrido.	81
5.5. Distribución de 100 millones de palabras de 16 bits experimentales.	83
5.6. Histograma de resultados experimentales primeros 100 mil secuencias binarias.	83

Índice de tablas

2.1.	Parámetros recomendados para el conjunto de pruebas del NIST.	38
2.2.	Resumen de los resultados núcleos TRNGs [2].	46
3.1.	Diferentes número de ciclos para diferentes valores de A	55
4.1.	Ejemplos de formato, rango y conversión de números de 5 bits en formato de punto fijo.	62
4.2.	Conversiones para codificación de los atractores.	64
4.3.	Diversos identificadores, dimensión fractal y exponente de Lyapunov para atractores del mapa caótico bidimensional.	66
4.4.	Rangos usados para la condición inicial para cada uno de los atractores.	66
4.5.	Tamaños de tipos de datos en C, compilador de GCC versión 11.2.0 en una plataforma x64 en Linux.	69
4.6.	Número de bits usados en la implementación de cada uno de los atractores con aritmética de punto fijo.	70
4.7.	Tabla lógica de LUT1.	76
4.8.	Tabla lógica de OBUFDS.	77
5.1.	Resultados de la aplicación de las pruebas NIST al TRNG híbrido implementado en aritmética de punto fijo con 100 secuencias de un millón de datos.	82
5.2.	Resultados de la aplicación de las pruebas NIST al TRNG híbrido implementado en aritmética de punto fijo con 1000 secuencias de un millón de datos.	82
5.3.	Uso de recursos del TRNG híbrido con multiplicadores completos. . . .	84
5.4.	Uso de recursos del TRNG híbrido con multiplicadores de una sola constante en parámetros a_n	84

Lista de códigos

4.1.	Librería para primitivas de Xilinx.	75
4.2.	Primitiva de LUT1.	76
4.3.	Primitiva de OBUFDS.	77
A.1.	Comprobar el número de bytes de los tipos de dato del sistema.	87
A.2.	Simulación de mapa caótico en punto flotante.	88
A.3.	Simulación de mapa caótico en punto fijo.	89
A.4.	Generador de memoria ROM de condiciones iniciales.	90
A.5.	Convertidor de punto flotante a punto fijo.	91
A.6.	Simulación de mapa caótico en punto fijo y operación mod 256.	92
A.7.	Simulación de mapa caótico en punto fijo y operación mod 256 salida binaria.	93
A.8.	Multiplexor para control de condición inicial y retroalimentación.	94
A.9.	Sumador genérico compatible con punto fijo.	94
A.10.	ROM para almacenar parámetros en punto fijo del mapa caótico.	95
A.11.	Multiplicador en punto fijo con truncamiento.	95
A.12.	Flip-Flop con habilitación.	95
A.13.	Máquina de estados para control de las iteraciones del mapa caótico.	96
A.14.	Descripción completa del mapa caótico.	97

Lista de abreviaciones

COSO-TRNG	Coherent sampling ring oscillator based TRNG.
ERO-TRNG	Elementary ring oscillator based TRNG.
FPGA	Field Programmable Logic Array.
MURO-TRNG	Multi-ring oscillator based TRNG.
PLL-TRNG	Phase-locked loop based TRNG.
RNG	Random Number Generator.
STR-TRNG	Self-timed ring based TRNG.
TERO	Transient Effect Ring Oscillator.
TERO-TRNG	Transient effect ring oscillator based TRNG.
TRNG	True Random Number Generator.

Resumen

En esta tesis se diseñó e implementó un TRNG híbrido en la FPGA Xilinx Artix 7 xc7a35tcpg236-1 sobre la tarjeta de desarrollo Digilent Basys 3. Se utilizó un núcleo ERO-TRNG para generar una semilla de 64 bits que funciona como condición inicial para un mapa caótico bidimensional. Haciendo uso de la operación mod 256 se extraen 16 bits aleatorios por cada iteración del mapa.

En este trabajo se presenta toda la teoría necesaria para comprender los generadores de números aleatorios, así como su clasificación, fuentes de aleatoriedad, parámetros de evaluación pruebas estadísticas y arquitecturas de núcleos TRNG específicas para FPGA. Se estudian brevemente las características principales de los mapas caóticos como puntos fijos, estabilidad lineal, diagramas de bifurcación y diagramas de cobwebs. Después se expone la metodología de diseño para implementar en FPGA el mapa caótico bidimensional y el núcleo ERO-TRNG utilizando el lenguaje de descripción de hardware VHDL. El mapa caótico se implementó utilizando aritmética de punto fijo de 64 bits, 3 bits para la parte entera, 60 bits para la fraccionaria y un bit de signo y para comprobar su funcionamiento se empleó un simulador desarrollado lenguaje C. Posteriormente se analizó el dominio de atracción del mapa para diferentes parámetros con el fin de poder seleccionar un rango en que las condiciones iniciales produzcan caos, por último se utilizaron multiplicadores de una sola constante para reducir el uso de recursos. Utilizando diversos elementos digitales básicos y el núcleo ERO-TRNG se diseñó un generador de semillas de 64 bits que alimenta a las condiciones iniciales del mapa caótico. Finalmente, las secuencias binarias obtenidas por el TRNG híbrido se mandaron a una computadora utilizando el protocolo de comunicación RS232 y se analizaron con las pruebas estadísticas NIST SP 800-22.

Capítulo 1

Introducción

Los números aleatorios se utilizan en muchos ámbitos de la vida cotidiana. Se utilizan para elegir quién gana la lotería, para determinar quién ataca primero en un partido de fútbol, para garantizar una partida justa en juegos de mesa y desempeñan un papel fundamental en la criptografía y la seguridad de la información. Para seleccionar al equipo atacante en un partido de fútbol, basta con lanzar una moneda. Sin embargo, para jugar a un juego de mesa se requieren más de dos valores aleatorios, por lo que se utiliza un dado. En cambio, la criptografía requiere algo más que tirar un dado para asegurar la protección de los datos en comunicaciones digitales o en transacciones bancarias. La seguridad de las comunicaciones es una parte fundamental de la vida moderna, en la que las personas envían correos electrónicos, realizan llamadas o envían mensajes a sus amigos y realiza transacciones en línea millones de veces al día. La sociedad confía que cada uno de estos procesos cotidianos sean seguros y confidenciales. La seguridad de las comunicaciones dependen de la capacidad de estos procesos para verificar la identidad de las personas que se comunican. La única forma de garantizar la seguridad es mediante la distribución de identidades privadas conocidas solo por el usuario, denominadas claves. Las claves privadas son números aleatorios únicos generados para cada usuario, que aseguran que personas malintencionadas no puedan suplantar a nadie y causar daño. La aleatoriedad de los números de las claves privadas es crucial para garantizar la seguridad de las conexiones. La capacidad de generar números aleatorios es, por tanto, una parte muy importante de la seguridad de los sistemas de comunicación.

Los números aleatorios tienen una amplia variedad de aplicaciones. Se utilizan en simulaciones a computadora de fenómenos naturales, como en el modelado de la colisión de partículas en física nuclear y en el análisis logístico de pasajeros en aeropuertos en investigación operativa. También son útiles en el muestreo estadístico cuando no es práctico analizar todos los casos posibles. Los números aleatorios son una buena fuente de datos para probar la efectividad de los algoritmos de computadora y son cruciales para el funcionamiento de los algoritmos aleatorios. Además, se utilizan en análisis

numérico y en la estética, donde agregar un poco de aleatoriedad hace que los gráficos generados por computadora y la música parezcan más vivos. En algunos casos, es importante tomar decisiones completamente imparciales y la aleatoriedad es esencial en las estrategias óptimas de la teoría de juegos. Ejecutivos y profesores universitarios recurren a estas estrategias con frecuencia, tirando una moneda o lanzando dardos [1].

En un sistema criptográfico, se utilizan generadores de números aleatorios o Random Number Generators (RNG), no sólo para generar claves criptográficas, sino también para generar números de un solo uso (nonces), valores de relleno, vectores de inicialización, desafíos y máscaras aleatorias para la protección contra ataques de canal lateral [2].

A pesar de que hay muchas aplicaciones diferentes de los números aleatorios, todos comparten dos requisitos básicos: buenas propiedades estadísticas, concretamente una distribución de probabilidad uniforme donde cada valor de cualquier número aleatorio tenga la misma probabilidad de aparecer, e imprevisibilidad de los números aleatorios. Los números aleatorios, especialmente los utilizados para parámetros secretos como las claves, deben ser impredecibles para evitar que un atacante pueda calcular valores futuros o anteriores a partir de los datos ya generados y capturados. En los diseños modernos, se requieren algunas características adicionales: el generador debe ser intrínsecamente seguro, robusto y resistente a los ataques y probado en línea mediante pruebas específicas del generador [3]. Los números aleatorios son una herramienta valiosa y versátil que se utiliza en prácticamente todos los ámbitos de la ciencia y la tecnología y su importancia en la seguridad es fundamental.

En 1999, Intel introdujo el generador de números aleatorios basado en silicio [4], ese RNG fue el primero de la familia de primitivos de Intel, lanzado para la protección de datos y comunicaciones dentro del hardware del PC. Desde ese entonces muchos científicos han buscado la manera de crear mejores sistemas integrados que generen números aleatorios para proteger los datos de los usuarios. En este esfuerzo se ha desarrollado toda una metodología para analizar, categorizar y evaluar los RNGs. Agencias de estandarización como la NIST (National Institute of Standards and Technology) [5] en Estados Unidos o la BSI (Federal Office for Information Security) [6] en Alemania han creado una serie documentaciones, recomendaciones y pruebas estadísticas para evaluar los generadores de números aleatorios.

1.1. Clasificación de los RNGs

Dado el amplio rango de aplicaciones de los RNG, existen diferentes clases de RNG que satisfacen diversas necesidades. Basándonos en el método utilizado para generar números aleatorios, podemos distinguir dos tipos fundamentales de RNG:

1. Generadores de números aleatorios deterministas (DRNG/PRNG)

También conocidos como generadores de números pseudo-aleatorios o Pseudo-Random Number Generators (PRNG), son sistemas que producen una secuencia de aspecto aleatorio de forma matemática, es decir, hay un algoritmo subyacente y debido a esto los Deterministic Random Number Generators (DRNG) son fáciles de implementar en dispositivos lógicos. Si se conoce el algoritmo, la salida del generador es predecible. Incluso cuando no se conoce el algoritmo pero se han guardado algunas de las secuencias de salida del generador, su comportamiento durante la secuencia guardada puede utilizarse en futuros ataques. Los números producidos parecen aleatorios a corto plazo, pero la secuencia es periódica, normalmente con un periodo largo. Para producir una salida menos predecible, estos generadores utilizan valores de inicialización llamados semillas para empezar a generar números [7]. Para cada semilla se genera una secuencia diferente. Por esta razón, los DRNG deben ser computacionalmente seguros, el algoritmo no debe poder adivinarse computacionalmente y su valor inicial nunca debe reutilizarse. La reutilización del valor inicial puede evitarse guardando el último valor haciendo uso de un contador y utilizando el siguiente valor del contador la próxima vez. La secuencia de salida de un buen DRNG está perfectamente distribuida de manera uniforme, en otras palabras, tienen buenas propiedades estadísticas. Además, los DRNGs consiguen altas tasas de bits de salida y suelen utilizarse como generadores de claves en los cifrados de flujo [3].

2. Generadores de números aleatorios verdaderos (TRNG)

Los True Random Number Generators (TRNG) son sistemas que extraen la aleatoriedad de fenómenos aleatorios no algorítmicos impredecibles como fluctuaciones de temperatura, decaimiento radiactivo, ruido de radio ambiental, tiempos de acceso al disco duro o interacciones del usuario con el PC, jitter, entre otros. A diferencia de los DRNG, que utilizan fórmulas matemáticas para generar números aleatorios, los TRNG producen datos aleatorios reales que no se pueden predecir. Sin embargo, debido a que los procesos físicos utilizados por los TRNG están sujetos a fluctuaciones, la calidad de la secuencia aleatoria generada puede presentar algunos defectos estadísticos, como el sesgo.

Dado que los procesos físicos están sujetos a fluctuaciones, las características estadísticas de los TRNG suelen ser peores que las de los DRNG y están estrechamente relacionadas con la calidad de la fuente de entropía y con el método de extracción de la aleatoriedad. La velocidad final de los TRNG está limitada por el espectro de la señal aleatoria y por el principio utilizado para extraer la entropía de la misma, por ejemplo, la frecuencia de muestreo vinculada al espectro del ruido. En general, los TRNG son, más lentos que los DRNG. Dependiendo de la fuente utilizada los TRNG se pueden dividir en:

- Físicos (PTRNG), utilizan ruido físico a nivel de electrones presente en todos los semiconductores. Es un dispositivo físico y utiliza ruido físico.
- No físicos (NPTRNG), pueden no ser dispositivos físicos, sino más bien una pieza de software que utilizan fuentes de aleatoriedad no físicas, como las interacciones del usuario con un sistema operativo, por mencionar algunas el movimiento del cursor del mouse.

La imprevisibilidad de los generadores de números aleatorios deterministas está garantizada computacionalmente, mientras que la imprevisibilidad de los generadores verdaderamente aleatorios está garantizada por fenómenos físicos aleatorios y se caracteriza por la tasa de entropía a la salida del generador. Tanto los TRNG como los DRNG tienen sus ventajas y desventajas, por lo que muchos sistemas criptográficos utilizan RNG híbridos. Los generadores de números aleatorios híbridos, conocidos como Hybrid Random Number Generators (HRNG) combinan las fortalezas de los DRNG (rápidos y de buena calidad) sembrados repetidamente por un TRNG (lento pero impredecible). Es importante encontrar un equilibrio entre la velocidad del generador y su imprevisibilidad ajustando el intervalo de tiempo entre semillas y el tamaño de la semilla. En función de su implementación, existen dos tipos de RNG híbridos:

1. Generadores de números aleatorios verdaderos híbridos

Combinan un TRNG con un postprocesamiento criptográfico. El postprocesamiento criptográfico asegura el secreto hacia adelante y hacia atrás de los números aleatorios producidos, no pueden calcularse los valores pasados o los futuros a partir del valor actual. Si la fuente física falla, también garantiza unas propiedades estadísticas perfectas de los datos de salida, ya que el núcleo de un postprocesamiento criptográfico suele ser un cifrador. La tasa de bits de salida de un TRNG híbrido está limitada por la del núcleo del TRNG.

2. Generadores de números aleatorios deterministas híbridos.

Utilizan un TRNG para generar periódicamente semillas para un DRNG. Dado que la salida de un DRNG es predecible si conocemos su semilla, ir renovando la semilla mediante un TRNG puede reducir la predictibilidad de un DRNG híbrido. Además la secuencia de salida de un generador de este tipo es perfectamente uniforme, lo que podría no ser el caso de un TRNG puro. Su tasa de bits de salida viene determinada por la tasa de bits del DRNG subyacente, ya que se pueden producir números aleatorios mientras no se alcance el periodo de repetición del DRNG.

Para garantizar la seguridad de las claves confidenciales, es fundamental que se generen dentro del sistema criptográfico. Dado que la mayoría de los sistemas criptográficos actuales se implementan en dispositivos lógicos y sistemas digitales utilizando

algoritmos y protocolos criptográficos, es natural que la investigación se centre en la implementación de generadores de números aleatorios en dispositivos lógicos, como los Field-Programmable Gate Arrays (FPGA). Esto ayuda a prevenir el acceso no autorizado a las claves y garantiza que sean generadas de manera segura y confiable dentro del sistema criptográfico.

A pesar de la menor velocidad de los TRNG, estos se utilizan con más frecuencia en aplicaciones criptográficas que los DRNG. Los TRNG son las únicas primitivas criptográficas que no han sido objeto de normalización hasta ahora. Sin embargo, antes de utilizar un generador en la práctica, el principio de funcionamiento y su implementación dentro de un módulo criptográfico deben ser validados por una institución acreditada como parte de un proceso de evaluación de seguridad. Los generadores que no tienen un certificado de seguridad se consideran inseguros en cuanto a su uso en aplicaciones criptográficas. Por este motivo, es de gran interés el estudio de los principales TRNG existentes y sus características.

1.2. Estructura de los TRNG

La estructura general de un TRNG se muestra en la Figura 1.1. El generador debe utilizar un proceso físico incontrolable como fuente de aleatoriedad. Dado que los fenómenos físicos utilizados en los TRNG son en su mayoría procesos analógicos, suele ser necesario algún método que permita la conversión de datos del dominio analógico al digital. Se puede incluir una conversión de analógico a digital en el procedimiento de extracción de aleatoriedad. La señal binaria sin procesar obtenida, el llamado ruido digital, puede tener baja entropía, malas propiedades estadísticas o ambas. En este caso, se pueden usar algunos algoritmos de postprocesamiento para mejorar los parámetros estadísticos del flujo de bits de salida. Sin embargo, el postprocesamiento de la salida del TRNG a veces puede enmascarar una falla grave en el generador. Las pruebas estadísticas estándar pueden entonces fallar en detectar la debilidad enmascarada. Por lo tanto, se recomienda tener la posibilidad de probar el ruido digital sin procesar o Raw Bit Signal (RBS). La seguridad del generador se puede aumentar si las pruebas estadísticas se aplican sobre la marcha y si están diseñadas para adaptarse al principio de funcionamiento del generador teniendo presente sus posibles debilidades [3].

Los TRNG emplean diferentes fuentes de aleatoriedad y una gran variedad de principios para extraerla. En este sentido, resulta relevante evaluar los principales principios utilizados por los TRNG: parámetros relacionados con la calidad, parámetros relacionados con la seguridad y parámetros relacionados con el diseño.

- Parámetros relacionados con la calidad
 - Fuente de aleatoriedad.

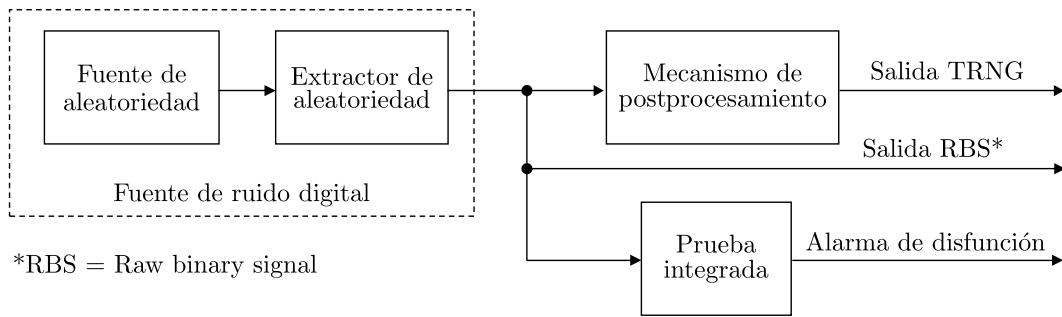


Figura 1.1: Estructura general de un TRNG [3].

- Método de extracción de aleatoriedad y entropía del ruido digital.
- Método de procesamiento posterior aplicado (opcional).
- Tasa de bits de salida y su estabilidad.
- Parámetros relacionados con la seguridad
 - Existencia de un modelo matemático.
 - Comprobabilidad interna.
 - Seguridad (robustez, resistencia contra ataques).
- Parámetros relacionados con el diseño
 - El uso de recursos.
 - El consumo de energía.
 - Viabilidad en dispositivos lógicos y FPGAs.
 - Automatización del diseño.

Es fundamental tener en cuenta que no todas las características de los TRNG tienen la misma relevancia en su evaluación. En particular, los parámetros relacionados con la seguridad, tales como la robustez, la disponibilidad de un modelo estocástico y la capacidad de prueba, adquieren una importancia crucial en un sistema de seguridad de datos. Su peso en la evaluación de un TRNG es significativamente mayor que el de otros parámetros, como el consumo de energía o la tasa de bits [3].

1.3. Condiciones de funcionamiento y calidad de la salida del TRNG

Para que un TRNG sea considerado de buena calidad, su salida debe ser prácticamente indistinguible de la salida de un TRNG ideal en todas las condiciones de funcionamiento y a lo largo del tiempo. En el diseño de un TRNG, es fundamental prestar atención tanto a la calidad del flujo de bits generado como a sus parámetros de seguridad, incluyendo la robustez ante el envejecimiento, cambios ambientales, posibles ataques y la existencia de pruebas de autodiagnóstico y en línea [8]. Un buen TRNG

debe proporcionar un flujo de 0s y 1s distribuido uniformemente en su salida. La calidad del flujo de bits generado está estrechamente relacionada con la calidad de la fuente de aleatoriedad y el método utilizado para extraer la aleatoriedad. Las características espectrales de la fuente y el método de extracción determinan los principales parámetros del flujo de bits generado, como el sesgo de los bits de salida, la correlación entre bits sucesivos y patrones visibles. Aunque algunos de estos problemas pueden corregirse mediante un postprocesamiento efectivo, es preferible que el generador produzca intrínsecamente un flujo de bits en bruto de alta calidad.

Si el extractor muestrea la fuente de aleatoriedad demasiado rápido, los bits adyacentes pueden estar correlacionados. Por esta razón, es una buena práctica comprobar si el flujo de bits generado presenta autocorrelación a corto plazo. Además, pueden existir otras dependencias a corto plazo en el ruido digital, que deben detectarse mediante pruebas específicas del generador. El comportamiento del generador puede verse afectado por interferencias eléctricas externas e internas. Un efecto evidente de estas interferencias son las frecuencias discretas que se originan en la fuente de alimentación y en diversas señales internas que aparecen en el espectro de ruido.

La señal de ruido generada puede verse significativamente afectada por el ruido de baja frecuencia causado por los semiconductores. Además, las capacidades internas pueden involuntariamente filtrar las frecuencias altas del espectro de ruido. Algunos generadores pueden presentar lo que se conoce como puntos malos, que son breves períodos en los que el generador deja de funcionar debido a interferencias eléctricas o sobrecargas en la parte sobrecargada del circuito.

Una característica peligrosa del generador es la existencia de una puerta trasera, que se refiere a las desviaciones de la aleatoriedad uniforme introducidas deliberadamente por el fabricante. Por ejemplo, supongamos que en lugar de utilizar un proceso físico, el generador genera una secuencia pseudoaleatoria de alta calidad con una semilla de 40 bits. Aunque sería imposible detectar este comportamiento aplicando pruebas estadísticas estándar al flujo de bits de salida, alguien que conozca la puerta trasera podría adivinar las claves sucesivas mediante un proceso computacionalmente factible.

En este trabajo nos enfocaremos en los generadores que pueden implementarse en sistemas digitales y en específico en FPGAs.

1.4. Trabajos actuales

Desde hace una década los TRNGs implementados en FPGA han ido evolucionando y fue en 2016 cuando [2] hizo una recopilación de todos los núcleos TRNGs que cumplen con las características descritas por la AIS20/31. Los TRNG destinados a ser utilizados en aplicaciones criptográficas deben cumplir con los siguientes requisitos: 1) su diseño debe ser sencillo y comprensible, la fuente de aleatoriedad debe estar claramente defi-

nida; 2) el proceso aleatorio subyacente debe ser estacionario y el modelo estocástico debe ser factible; 3) la señal binaria sin procesar debe estar disponible para posteriores pruebas off-line y on-line. Además, es útil que la fuente de aleatoriedad (por ejemplo, el jitter del reloj procedente del ruido térmico) sea cuantificable: su medición dentro del dispositivo puede servir de base para realizar pruebas estadísticas integradas rápidas y eficaces.

Además seleccionó núcleos TRNG que utilizan circuitos oscilantes: osciladores en anillo de evento único (es decir, osciladores en anillo estándar) [9], [10], osciladores en anillo multievento con colisiones de señal (es decir, osciladores en anillo de efecto de transición) [11], osciladores en anillo multievento sin colisiones de señal (es decir, anillos autotemporizados) [12] y bucles de fase bloqueada (PLL) [13]. Todos ellos deberían ser viables en las familias recientes y futuras de FPGAs. Todos ellos utilizan fuentes de aleatoriedad sencillas y comprensibles, su señal aleatoria sin procesar está disponible fuera del generador, y el modelo estocástico del generador existe o es factible. Por lo tanto, todos ellos cumplen los requisitos de AIS-20/31.

Con base en los criterios del AIS-20/31, los núcleos TRNG adecuados para utilizarse en dispositivos lógicos programables (FPGA) que usan estructuras oscilantes son:

- Elementary ring oscillator based TRNG (ERO-TRNG).
- Coherent sampling ring oscillator based TRNG (COSO-TRNG).
- Multi-ring oscillator based TRNG (MURO-TRNG).
- Transient effect ring oscillator based TRNG (TERO-TRNG).
- Self-timed ring based TRNG (STR-TRNG).
- Phase-locked loop based TRNG (PLL-TRNG).

En años recientes se ha buscado mejorar estos núcleos haciendo variaciones en las conexiones de los componentes internos sin modificar el método de extracción de aleatoriedad. En [14] se diseño un TRNG basado en PLLs y su contribución se enfocó en analizar su comportamiento a variaciones de voltaje y temperatura, utilizó 56 LUTs, 19 Registros y obtuvo una velocidad de salida de 100 Mbit/s. En [15] se modificó un TRNG basado en el núcleo COSO utilizando una Xilinx Spartan 7, pasó las pruebas estadísticas AIS-31, las NIST y tuvo una velocidad de salida de 1.47 Mbit/s. En [16] se diseño un TRNG basado en retardos de osciladores de anillo y posteriormente se realizó un postprocesamiento corrector Von Neumann, pasó todas las pruebas NIST, utiliza 528 slices de una Xilinx Spartan 3-A y obtuvo una velocidad de salida de 6 Mbit/s.

Los artículos anteriores se enfocan en mejorar los núcleos para obtener mejores tasas de bits de salida, disminuir el uso de recursos o mejorar la robustez, sin embargo, existe

otro enfoque que consiste en utilizar los TRNG únicamente para generar la semilla de un DRNG. Es decir, crear TRNGs híbridos que combinen las mejores característica de ambos generadores. Además el uso de caos como postprocesamiento o DRNG ha crecido en los años. En [17] se discretiza un sistema dinámico caótico y se genera una estructura combinada para sembrar el sistema y obtener números aleatorios, se utilizaron 12383 LUTs, 134883 registros, 145 DSPs de una FPGA de la familia Xilinx. En [18] se utiliza un sistema hipercaótico de 5 dimensiones discretizado utilizando los métodos numéricos de forward Euler, trapezoidal, y Runge-Kutta de cuarto orden para diseñar un TRNG híbrido, obtuvo una velocidad de salida de 416 Mbit/s y utiliza 1648 LUTs y 1112 registros. En [19] se estudiaron diversos mapas caóticos para utilizarse como DRNG implementados en FPGA, se analizó cual de todos los mapas estudiados genera los mejores resultados y se concluyó que el mapa de corrimientos de Bernoulli es el mejor, utilizando 575 LUTs, 108 registros y obteniendo una velocidad de salida de 7.389 Mbit/s. En los artículos anteriores se busca aprovechar el caos para aumentar la tasa de bits de salida de los TRNG, sin embargo esto es a costa de aumentar el uso de recursos.

Se ha probado extensamente que el caos es una herramienta muy útil para generar números aleatorios y en particular se utiliza mucho para crear esquemas de cifrado [20–22], cifrado de imágenes [23–29] y en particular los mapas caóticos han cobrado popularidad debido a que su estructura matemática por lo general es sencilla y puede describirse en hardware utilizando pocos recursos, a diferencia de los sistemas dinámicos caóticos en tiempo continuo los cuales requieren el uso de métodos numéricos y su implementación no es trivial.

Nuevos mapas caóticos [30] con mejores características se han estudiado en recientes años lo que abre la posibilidad de poder seguir mejorando los DRNGs. Además sistemas integrados con co-diseño, FPGA-Soc, [31] hacen cada vez más fácil poder realizar las pruebas estadísticas en linea y hacer sistemas cada vez más robustos.

En este trabajo utilizaremos un TRNG para sembrar un DRNG basado en un mapa caótico bidimensional, probaremos sus características estadísticas utilizando las pruebas NIST y analizaremos la versatilidad del diseño analizando las variaciones del mapa caótico.

1.5. Objetivos

1.5.1. Objetivo general

- Diseñar e implementar en FPGA un TRNG híbrido para la generación de secuencias muy largas.

1.5.2. Objetivos específicos

- Investigar el estado del arte de diferentes generadores de números aleatorios.
- Estudiar los diferentes tipos de generadores de números aleatorios y analizar sus características principales.
- Estudiar la teoría de los mapas caóticos y su utilidad en generadores de números aleatorios.
- Diseñar un generador de números aleatorios híbrido utilizando un TRNG como generador de semillas y un mapa caótico para realizar un postprocesamiento que mejore sus características estadísticas y comprobar estas utilizando las pruebas NIST.
- Implementar el TRNG híbrido en una FPGA.

Capítulo 2

Generadores de números aleatorios (RNGs)

En este capítulo se presentan los conceptos necesarios para comprender cómo funcionan las diferentes clases de generadores de números aleatorios, sus fuentes de aleatoriedad, sus principales características y cuales son factibles para implementación en FPGA.

2.1. Parámetros de evaluación de los TRNG

2.1.1. Parámetros relacionados con la calidad

2.1.1.1. Fuentes de aleatoriedad en dispositivos lógicos

Los TRNG pueden utilizar fuentes de ruido físicas o no físicas. Sin embargo, en el caso de los dispositivos lógicos, las fuentes de ruido físico son bastante limitadas, ya que estos dispositivos se diseñan para ser deterministas y estar en un estado bien definido en todo momento. Por lo tanto, para generar números verdaderamente aleatorios, es necesario contar con un fenómeno aleatorio incontrolable que pueda utilizarse como fuente de aleatoriedad. Los fenómenos físicos más utilizados para generar números aleatorios en los dispositivos lógicos son:

- Señales analógicas: como el ruido de disparo de un diodo, el ruido térmico, etc.
- Metaestabilidad: es la capacidad de un circuito de persistir en un estado indefinido durante un periodo de tiempo indefinido.
- Jitter del reloj: es una variación del flanco del reloj desde su posición ideal.
- Caos: es un comportamiento imprevisible de un sistema determinista, muy sensible a sus condiciones iniciales, lo que significa que incluso el más mínimo cambio en el estado inicial produce resultados muy diferentes.

La presencia de comportamientos impredecibles en un dispositivo lógico puede tener consecuencias graves en el comportamiento del sistema global. A pesar de que estos eventos son inevitables debido a la naturaleza física de la tecnología, los vendedores de dispositivos lógicos suelen minimizarlos. Por ello, es fundamental examinar de forma crítica los métodos de extracción de aleatoriedad utilizados en el diseño de los TRNG para mantenernos al día con la evolución de la tecnología.

La mayoría de los dispositivos lógicos, y concretamente los FPGA, no contienen bloques analógicos y de tenerlos sería necesario un convertidor analógico digital (ADC) para poder utilizar una señal analógica, lo cual agrega otra capa de complejidad. Por esta razón las fuentes de aleatoriedad más utilizadas en dispositivos lógicos están relacionadas con el funcionamiento de las puertas lógicas. Se pueden utilizar varios fenómenos y sus combinaciones como la variación del retardo de las puertas lógicas, el comportamiento analógico de las puertas lógicas entre dos niveles lógicos (metaestabilidad), la violación del tiempo de preparación y retención y el ruido térmico generado dentro del dispositivo.

La inestabilidad de los retardos de las puertas lógicas provoca variaciones en la propagación de la señal a lo largo del tiempo. Estas variaciones pueden verse como una inestabilidad en el periodo del reloj (jitter) en los generadores de reloj que contienen elementos de retardo ensamblados en un circuito cerrado (osciladores en anillo). Asimismo, la variación del tiempo de propagación se utiliza también en generadores con elementos de retardo ensamblados en una cadena abierta. La cadena se utiliza para aumentar o ajustar el retardo total.

La tecnología digital permite implementar fácilmente resistencias y condensadores, cuyo ruido térmico generado en las resistencias puede utilizarse para modular la frecuencia de un oscilador de funcionamiento libre, también conocido como oscilador RC. De esta manera, el ruido térmico se convierte al dominio temporal, lo que facilita su extracción. No obstante, este principio no es aplicable en FPGAs, ya que carecen de estructuras adecuadas para ello.

Algunos generadores utilizan el jitter de seguimiento introducido por los bucles de fase bloqueada, phase locked loops (PLL), para generar números aleatorios. Estos generadores, llamados PLL analógicos, pueden implementarse fácilmente en dispositivos digitales, incluidos los FPGA, debido a que el filtro RC presente en dichos PLL es el único bloque “analógico” y puede implementarse utilizando la misma tecnología. Por lo tanto, es posible considerar un TRNG basado en PLL como un generador que se puede implementar en dispositivos lógicos en general

2.1.1.2. Métodos de extracción de aleatoriedad

En los dispositivos lógicos que carecen de bloques analógicos, a menudo se extrae la aleatoriedad mediante el muestreo de una señal (reloj) en los flancos ascendentes o descendentes de una señal (reloj) de referencia utilizando flip-flops síncronos o asíncronos (latches). El flujo de bits aleatorio se puede obtener de dos maneras: muestreando señales aleatorias a intervalos regulares o muestreando señales regulares a intervalos de tiempo aleatorios. En los sistemas síncronos, se prefiere el primer método para garantizar una tasa constante de bits a la salida.

La elección entre flip-flops síncronos y asíncronos resulta crítica en las FPGA. Esto se debe a que los flip-flops síncronos se encuentran cableados en celdas lógicas optimizadas, reduciendo así su comportamiento metaestable. En cambio, los latches suelen implementarse en tablas de consulta (LUT) y, por tanto, están más expuestos a sufrir comportamiento metaestable. Sin embargo, aún no se ha evaluado adecuadamente la influencia del uso de flip-flops síncronos y asíncronos en la extracción de aleatoriedad en las FPGA.

El método de extracción de la aleatoriedad está estrechamente vinculado al principio básico del generador y a la fuente de aleatoriedad utilizada. En algunos casos, el proceso de extracción de la aleatoriedad y el postprocesamiento se fusionan en el mismo bloque y no pueden ser separados. En tales casos, la entropía de la fuente de aleatoriedad se modifica por el postprocesamiento, lo que dificulta su medición y evaluación precisa.

En la evaluación de un generador de números aleatorios, la fuente de aleatoriedad y el método de extracción están estrechamente relacionados y no pueden evaluarse por separado. Por tanto, lo más adecuado es evaluar ambos parámetros conjuntamente, tomando en cuenta la entropía contenida en el ruido digital.

2.1.1.3. Técnicas de postprocesamiento

La evaluación de los TRNG se basa principalmente en pruebas estadísticas que se aplican a las secuencias aleatorias producidas por el generador. Sin embargo, en ocasiones la fuente de entropía puede presentar ciertas debilidades que se traducen en la generación de números no aleatorios, como largas secuencias de ceros o unos. Por esta razón, puede ser necesario aplicar postprocesamiento para mejorar las propiedades estadísticas de los números aleatorios, como aumentar la entropía por bit, reducir el sesgo y la correlación, entre otros.

La calidad de la señal de ruido digital obtenida en el bloque de extracción de aleatoriedad puede verse afectada por diversas razones:

- (a) La entropía de la fuente no es lo suficientemente alta, lo cual suele ocurrir cuando se utiliza la metaestabilidad como fuente de aleatoriedad.
- (b) La entropía, que es alta en la señal original, no se extrae adecuadamente.

- (c) Las muestras extraídas están correlacionadas entre sí.

El objetivo del bloque de postprocesamiento es mejorar la calidad de la secuencia de salida, de manera que no se distinga de una secuencia aleatoria ideal que esté libre de correlaciones y se distribuya uniformemente. En general, existen dos tipos principales de postprocesamiento:

1. Postprocesamiento algorítmico

Consiste en la aplicación de un algoritmo de procesamiento de datos con el objetivo de mejorar los parámetros estadísticos de los números generados. Entre las técnicas más utilizadas se encuentra la operación XOR de varios bits de salida, la corrección de Von Neumann y varios tipos de algoritmos de compresión, entre otros.

2. Postprocesamiento criptográfico

Emplea algoritmos criptográficamente seguros para asegurar la imprevisibilidad de los números generados en dirección de avance y/o retroceso en caso de fallo en la fuente física de aleatoriedad. La implementación de un postprocesamiento criptográfico refuerza la robustez del TRNG frente a posibles ataques.

Las técnicas más utilizadas para el postprocesamiento de la señal obtenida en el bloque de extracción de aleatoriedad son:

- Corrector XOR

Es una función lineal simple que aplica una operación XOR en bloques no superpuestos de n bits para generar un bit de salida. Puede reducir drásticamente el sesgo de la salida del generador a costa de reducir su tasa de bits n veces. Sin embargo, el sesgo del flujo de bits de salida sólo se reduce si los bits originales son independientes. Las principales ventajas del corrector XOR son su simplicidad y la posibilidad de mantener una tasa de bits de salida constante.

- Corrector de Von Neumann

Es una función no lineal sencilla que toma pares sucesivos de bits y, si los bits no son iguales, utiliza el primer bit del par y descarta los pares idénticos. Por tanto, la tasa de bits de salida depende de los datos.

Aunque la secuencia de entrada sea estacionaria y pueda estar sesgada, la salida no tendrá sesgo. Sin embargo, si la secuencia original está autocorrelacionada, la salida también puede estar autocorrelacionada. También es importante resaltar que el corrector de Von Neumann producirá una salida sesgada si la secuencia de entrada presenta un ciclo con periodo 2. Si se implementa el corrector en hardware, puede interferir con el generador y resultar en este tipo de ocurrencia.

- Linear Feedback Shift Registers (LFSRs)

Los LFSR son ampliamente utilizados en generadores de secuencias de bits aleatorias por varias razones. En primer lugar, son fáciles de implementar en hardware. En segundo lugar, pueden generar secuencias con periodos largos. En tercer lugar, pueden producir secuencias con buenas propiedades estadísticas. Y en cuarto lugar, debido a su estructura, son fáciles de analizar utilizando técnicas algebraicas.

Un LFSR de longitud L consta de L elementos de retardo que pueden almacenar un bit cada uno y tiene una entrada, una salida y un reloj que controla el movimiento de los datos. En cada unidad de tiempo, se llevan a cabo las siguientes operaciones:

- (I) El contenido del primer elemento de retardo sale y forma parte de la secuencia de salida.
- (II) El contenido del elemento i se mueve a la etapa $i - 1$ para cada i , $1 \leq i \leq L - 1$.
- (III) El nuevo contenido del último elemento de retardo es el bit de retroalimentación que se calcula sumando módulo 2 los contenidos anteriores de un subconjunto fijo de elementos, dependiendo del polinomio subyacente.

- Funciones resilientes

Son funciones booleanas especiales que se utilizan en criptografía y teoría de la codificación. Su estudio es fundamental en el diseño de algoritmos de clave simétrica. Para analizar estas funciones, es importante considerar su grado, su forma algebraica normal, la transformada Möbius, la transformada de Walsh-Hadamard entre otros conceptos relacionados.

En palabras simples, las funciones resilientes resultan apropiadas para el postprocesamiento debido a que el conocimiento de cualquier conjunto de m valores de entrada no permite adivinar la salida con una precisión mayor a la del azar. Estas funciones se fundamentan en el mismo principio que los códigos de corrección de errores. Mientras que dichos códigos se emplean para eliminar errores aleatorios, las funciones resilientes se usan para extraer los bits aleatorios presentes.

La señal binaria sin procesar (RBS), también conocida como ruido digital, puede ser interpretada como un flujo de bits que contiene redundancia, y la función resiliente disminuye la tasa de bits al mismo tiempo que extrae los bits aleatorios. De esta manera, la función resiliente incrementa la entropía por bit en la salida del generador.

- Cifrado de la señal de ruido digital

Este tipo de postprocesamiento del ruido digital emplea las propiedades de difusión y confusión inherentes a las funciones criptográficas. Las características estadísticas óptimas de la mayoría de los algoritmos de cifrado pueden ser aprovechadas para enmascarar las imperfecciones del generador. Una ventaja de este enfoque radica en que la clave de cifrado puede ser utilizada como variable criptográfica para alterar dinámicamente el comportamiento del generador. A pesar de que este tipo de bloque de postprocesamiento (el cifrador) resulta bastante complejo y costoso, el TRNG puede reutilizar (compartir) el cifrador que se emplea para cifrar los datos.

- Hashing de la señal de ruido digital

Uno de los métodos más seguros, aunque también de los más tiempo consume, es el postprocesamiento criptográfico basado en funciones hash, como MD5, SHA-1, entre otras. Este enfoque aprovecha las propiedades de difusión y unidireccionalidad (a diferencia del cifrado de la señal binaria sin procesar) de las funciones hash para garantizar la imprevisibilidad de los bits generados por el TRNG en caso de un fallo completo en la fuente de ruido. En esta situación, debido a la propiedad de no linealidad de las funciones hash, el TRNG se comportará como un PRNG.

El postprocesamiento puede incrementar la tasa de entropía a la salida del generador, aunque ello implique reducir la tasa de bits de salida, no obstante, nunca puede generar entropía por si mismo. Idealmente, se busca producir números aleatorios sin procesar de alta calidad, de manera que el postprocesamiento no sea necesario. Este enfoque es especialmente beneficioso en aplicaciones de alta seguridad.

2.1.1.4. Tasa de bits de salida

En muchas aplicaciones criptográficas, la velocidad de generación de números aleatorios es un parámetro crítico a considerar, después de la seguridad. Por lo general, velocidades de salida de entre 100 kilobits por segundo hasta 1 megabit por segundo son suficientes. Sin embargo, hay ciertas aplicaciones que requieren una generación de números aleatorios a alta velocidad, como los servidores de telecomunicaciones de alta velocidad. Estos servidores necesitan generar claves de sesión a velocidades que pueden alcanzar decena de megabits por segundo. Por ejemplo, un servidor de 10 GBit necesitaría unos 20 Mbits/s de datos aleatorios para generar una clave de sesión de 128 bits por cada bloque de datos de 64 kB, con el fin de resistir los ataques de canal lateral.

Otro aspecto crucial a tener en cuenta en los generadores de números aleatorios es la variabilidad de su tasa de bits de salida. Algunos generadores producen números aleatorios periódicamente, mientras que otros generan la salida en intervalos de tiempo

irregulares, lo que requiere una memoria intermedia FIFO para almacenar los números generados. Una solución alternativa consiste en estimar la tasa de bits más baja disponible en la salida y muestrear la salida a esa tasa. No obstante, la desventaja de la primera solución es que las memorias intermedias FIFO pueden ser bastante grandes, dependiendo de la velocidad de bits promedio de salida y la demanda de números aleatorios. Por otro lado, la desventaja de la segunda solución radica en que si la tasa de bits estimada resulta ser incorrecta, puede haber momentos en los que los números aleatorios no estén accesibles en la salida.

2.1.2. Parámetros relacionados con la seguridad

2.1.2.1. Modelado matemático de TRNG

En un proceso de certificación de seguridad, la evaluación del diseño y la implementación del generador es fundamental. El objetivo principal es cuantificar la entropía por bit aleatorio, la cual es una propiedad de las variables aleatorias y no de las observaciones realizadas (números aleatorios). Para realizar una cuantificación precisa de la entropía, es necesario analizar la distribución de las variables aleatorias a través de un modelo estocástico. Dicho modelo especifica una familia de distribuciones de probabilidad de variables aleatorias, permitiendo verificar un límite inferior de entropía para la señal binaria en bruto. En diversos estudios, se han utilizado modelos que proporcionan límites inferiores para la entropía condicional media por número aleatorio interno. Estos modelos permiten comprobar la entropía de los números generados en tiempo real y son utilizados por varios autores en la evaluación de la calidad de los generadores de números aleatorios.

2.1.2.2. Comprobabilidad

La comprobabilidad o verificación interna se refiere a la capacidad del generador de evaluar la entropía de la señal binaria sin procesar la cual debe estar disponible para su evaluación. Esta funcionalidad es necesaria en los procedimientos de evaluación más recientes de TRNG. Sin embargo, en los casos en que la extracción de aleatoriedad y el postprocesamiento se fusionan en un solo proceso, la señal aleatoria sin procesar no siempre está disponible. Incluso cuando esta señal está disponible, a veces se compone de un patrón pseudoaleatorio combinado con un flujo de bits verdaderamente aleatorio, lo que dificulta la evaluación estadística de la señal.

2.1.2.3. Evaluación de seguridad

En ocasiones resulta muy complicado, e incluso imposible, construir un modelo estocástico para un generador específico. En tales situaciones, se puede utilizar un

enfoque alternativo para validar el uso del generador en aplicaciones criptográficas. Este enfoque se basa en el análisis del impacto de un entorno cambiante o de un ataque al generador, lo que se conoce como evaluación de seguridad. Hay tres posibilidades:

- (I) Se puede probar que el generador no puede fallar como resultado de un ataque o un entorno cambiante.
- (II) No existen pruebas de seguridad ni ataques.
- (III) Se ha informado de algún ataque a un generador en particular.

2.1.3. Parámetros relacionados con el diseño

2.1.3.1. Uso de recursos

Para evaluar la viabilidad de diferentes principios de TRNG, es crucial examinar los recursos necesarios para la implementación del generador en hardware. En general, todos los tipos de recursos disponibles en las FPGA pueden utilizarse para generar números aleatorios, incluyendo células lógicas basadas en LUTs o multiplexores, bloques de memoria integrados, bloques de reloj con PLLs y DLLs, osciladores RC integrados, multiplicadores cableados, interconexiones programables entre otros.

Los FPGA tienen una gran cantidad de celdas lógicas, lo que hace que su uso no suponga normalmente un problema en cuanto al área lógica. Sin embargo, la topología y los parámetros eléctricos de las interconexiones programables están fuertemente condicionados por la tecnología utilizada. En el caso de los diseños de TRNG, a menudo se requiere la intervención manual del diseñador durante la colocación y el enruteamiento (Placement and Routing). En cuanto a este último, algunos diseños pueden implementarse sin dificultad en una familia de FPGA, pero resultan complicados o incluso imposibles de implementar en otras. Asimismo, la elección y el número de bloques cableados integrados suelen ser mucho más limitados (PLL, osciladores RC, multiplicadores, bloques de memoria), y varían según el proveedor y la tecnología. La utilización de estos bloques puede ser un factor limitante para la reutilización del principio TRNG en diferentes plataformas FPGA.

2.1.3.2. Consumo de energía

El consumo de energía del generador está estrechamente relacionado con su fuente de aleatoriedad, la frecuencia de reloj utilizada y la complejidad del algoritmo. En algunas aplicaciones donde la eficiencia energética es crítica, el generador puede detenerse cuando no se está utilizando. No obstante, es importante tener en cuenta que la capacidad de controlar la generación del flujo de bits puede ser utilizada como una vía de ataque al generador.

2.1.3.3. Viabilidad en FPGAs

La implementación de TRNGs en FPGAs es mucho más limitada en comparación con su implementación en ASICs. La mayoría de los TRNG implementados en ASIC utilizan componentes analógicos para generar aleatoriedad, como TRNG basados en caos que utilizan convertidores analógico-digitales, generadores basados en osciladores de funcionamiento libre que utilizan ruido térmico de diodos y resistencias, y para procesar la aleatoriedad se utilizan amplificadores operacionales y comparadores. Sin embargo, la mayoría de estos bloques funcionales no están disponibles en FPGAs, aunque algunos de ellos pueden estar disponibles en familias seleccionadas, por ejemplo, osciladores RC en FPGAs de Actel Fusion y PLLs analógicos en la mayoría de las familias de Altera y Actel, estos no se encuentran en las antiguas familias de Xilinx. Algunos generadores son difíciles de implementar o no son viables en FPGAs, mientras que otros son viables solo en FPGAs seleccionadas. Sin embargo, los principios más generales son viables en todas las FPGAs.

2.1.3.4. Automatización del diseño

La disponibilidad de recursos en las FPGA no garantiza la implementación exitosa de un generador, ya que (el rango de tolerancias de) ciertos parámetros tecnológicos pueden limitar su viabilidad. El enrutamiento y sus características son los principales factores que restringen la implementación de los generadores en FPGAs. Algunos generadores requieren un enrutamiento equilibrado, lo que implica una colocación precisa de los módulos y un enrutamiento óptimo, por ejemplo la colocación simétrica de dos módulos en relación con otro módulo. Mientras que la mayoría de las herramientas de diseño FPGA permiten un control preciso de la colocación, el proceso de enrutamiento puede ser difícil o imposible de controlar en algunas familias, como Actel.

Incluso cuando el enrutamiento puede ser controlado como en las familias Altera y Xilinx, los retardos en la red de enrutamiento configurable varían entre dispositivos, lo que dificulta la equilibración de las interconexiones de los módulos de forma general. Por lo tanto, el diseño dependerá del dispositivo y requerirá equilibrio manual para cada uno. Esta intervención manual no es aceptable desde el punto de vista de una implementación práctica del generador. Aunque los mejores generadores (muy raros) pueden mapearse automáticamente en todas las familias de FPGA, la intervención manual es aceptable para la implementación del generador en cada familia y tipo de dispositivo. Sin embargo, los generadores que requieren una optimización manual por dispositivo no son viables para aplicaciones industriales.

2.2. Fuentes de aleatoriedad en los dispositivos lógicos

2.2.1. Jitter del reloj

Como se mencionó anteriormente, el jitter del reloj es una fuente confiable de aleatoriedad en dispositivos lógicos. El término “jitter” se refiere a la incertidumbre en la oscilación del reloj en el dominio del tiempo. En otras palabras, se trata de la variación a corto plazo de un evento con respecto a su posición ideal. Por lo general, se mide como la variación en el tiempo del cruce por cero (flanco ascendente o descendente) de la señal de reloj. Aunque esta definición proporciona una comprensión básica del concepto, cuantificar el jitter puede ser confuso debido a la variedad de técnicas de medición utilizadas en diferentes aplicaciones y a la falta de consenso entre los autores en la definición precisa del término [32].

Se supone que una señal de reloj ideal en dispositivos lógicos digitales es una señal rectangular con un ciclo de trabajo del 50 % y un periodo estable. Pero debido a diversos ruidos que afectan a los dispositivos electrónicos, la señal de reloj nunca es absolutamente estable y sus bordes se mueven de su posición estable. En otras palabras, la fase de la señal de reloj fluctúa. Esta fluctuación puede verse como un jitter del reloj en el dominio del tiempo y como un ruido de fase en el dominio de la frecuencia. En los dispositivos lógicos, la fluctuación de reloj suele ser no deseada, pero inevitable. Dado que el jitter afecta negativamente a las comunicaciones de alta frecuencia y a los sistemas de alta velocidad, se ha estudiado y caracterizado en profundidad.

En los sistemas analógicos, el jitter se caracteriza mejor en el dominio de la frecuencia. De este modo, los componentes de fase y amplitud pueden estudiarse y caracterizarse por separado. En cambio, en los sistemas digitales, las propiedades temporales del jitter son más importantes y, por tanto, el jitter se caracteriza en el dominio temporal [13].

El jitter del reloj en un sistema digital es una desviación del flanco de reloj real con respecto a un flanco de reloj ideal. Una señal de reloj ideal se define mediante la ecuación (2.1), donde $t(n)$ representa el tiempo del periodo n -ésimo de una señal de reloj y T es el periodo de una señal de reloj.

$$t(n) = n \cdot T \quad (2.1)$$

En la práctica, una señal de reloj real no alcanza siempre múltiplos enteros de su periodo ideal, sino que sus flancos fluctúan alrededor de este valor debido al jitter. Esta variación es causada por diversos fenómenos físicos, como el ruido térmico, el ruido de la fuente de alimentación y el ruido electromagnético ambiental, entre otros. La Figura 2.1 muestra cómo se ve una señal de reloj afectada por el jitter.

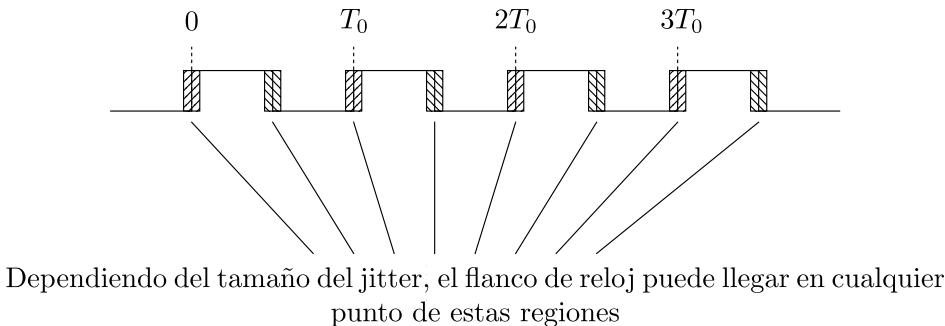


Figura 2.1: Jitter del reloj.

La Figura 2.2 muestra la causa principal del jitter en los circuitos digitales. Los circuitos digitales utilizan un nivel de referencia, generalmente ubicado en el centro del rango de voltaje de funcionamiento, para detectar los flancos de reloj. Es importante que este nivel de referencia sea lo más estable posible, pero en la práctica sufre fluctuaciones debido a diversos tipos de ruido. Cuando el nivel de referencia se desplaza, los flancos del reloj se detectan antes o después de lo previsto, lo que resulta en una variación temporal conocida como jitter de reloj.

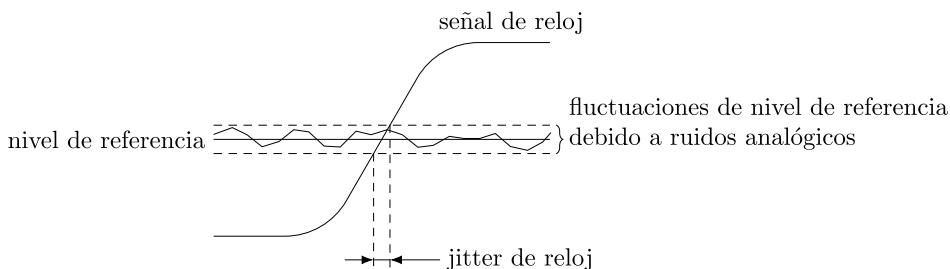


Figura 2.2: Fluctuaciones del nivel de referencia originadas por ruidos analógicos que provocan jitter del reloj en los circuitos digitales. [33]

A continuación se explican las diferentes medidas de jitter que observamos en circuitos digitales y las relaciones entre ellas.

2.2.1.1. Jitter de fase

El jitter de fase es una diferencia entre el tiempo del n -ésimo flanco de reloj real $t_r(n)$ y el tiempo (fase) del n -ésimo flanco de reloj ideal. La ecuación (2.2) define esta relación.

$$\delta_\varphi(n) = t_r(n) - n \cdot T_{\text{ref}} \quad (2.2)$$

La Figura 2.3 ilustra este jitter para $n = 3$. Para mayor claridad, sólo se muestra el jitter de fase de los flancos ascendentes, pero hay que tener en cuenta que el jitter de fase afecta a todos los flancos del reloj.

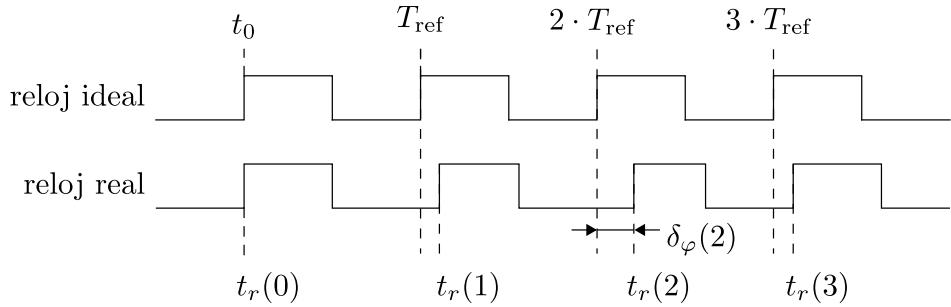


Figura 2.3: Ilustración de la jitter de fase del segundo flanco ascendente de la señal de reloj. [33]

En la Figura 2.3 se muestra que el jitter de fase $\delta_\varphi(2)$ no solo se ve afectada por la desviación de fase de $t_r(2)$, sino que también contiene contribuciones a la desviación de $t_r(1)$. Esto se conoce como acumulación de jitter de fase y provoca que el jitter de fase observado aumente cuanto mayor sea n .

2.2.1.2. Jitter de periodo

El jitter de periodo es la diferencia entre un periodo de reloj real y el de uno ideal. Se define como se muestra en la ecuación (2.4), una diferencia de primer orden del jitter de fase.

$$\delta_T(n) = [t_r(n) - t_r(n-1)] - T_{\text{ref}} \quad (2.3)$$

$$= \delta_\varphi(n) - \delta_\varphi(n-1) \quad (2.4)$$

La Figura 2.4 muestra el jitter de periodo. Podemos ver que los períodos reales cambian con el tiempo, a diferencia de los períodos ideales, que permanecen constantes.

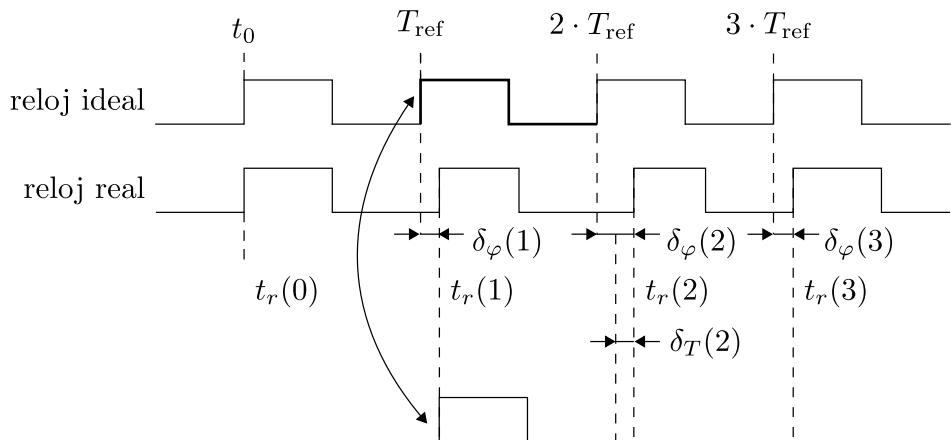


Figura 2.4: Ilustración de jitter de periodo de una señal de reloj real comparada con el reloj ideal.

2.2.1.3. Jitter de ciclo a ciclo

El jitter ciclo a ciclo es una diferencia de dos períodos de reloj reales consecutivos, tal y como se define en la ecuación (2.7).

$$\delta_c(n) = T_r(n) - T_r(n-1) \quad (2.5)$$

$$= [t_r(n) - t_r(n-1)] - [t_r(n-1) - t_r(n-2)] \quad (2.6)$$

$$= \delta_T(n) - \delta_T(n-1) \quad (2.7)$$

Todas estas medidas de jitter están relacionadas entre sí, el jitter de periodo es la diferencia de primer orden del jitter de fase y el jitter ciclo a ciclo es la diferencia de primer orden del jitter de periodo. Por lo tanto, en general basta con medir sólo uno y calcular cualquier otro que pueda ser necesario. En la Figura 2.5 se muestra el jitter ciclo a ciclo.

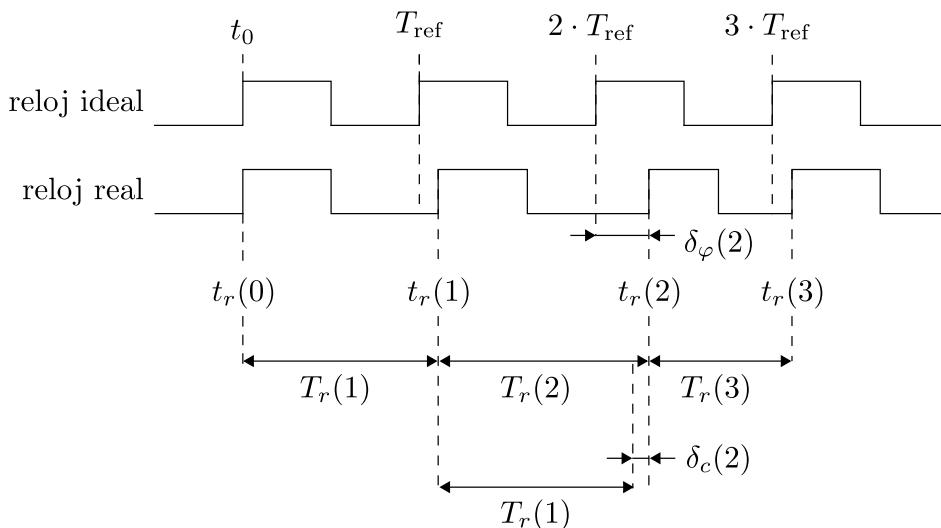


Figura 2.5: Ilustración de jitter de ciclo a ciclo.

2.2.1.4. Composición del jitter

El jitter del reloj suele tener dos componentes, ver la Figura 2.6, jitter aleatorio, causada por un fenómeno no determinista como el ruido térmico y jitter determinista, causada por algunos procesos deterministas.

El jitter aleatorio sigue una distribución de probabilidad gaussiana y se rige por el teorema del límite central. Debido a su comportamiento aleatorio, se utilizan herramientas estadísticas, como la media y la desviación estandar, para medirlo. El jitter aleatorio surge de la suma de múltiples factores independientes inherentes a cualquier circuito eléctrico, como las vibraciones térmicas de las estructuras cristalinas de los

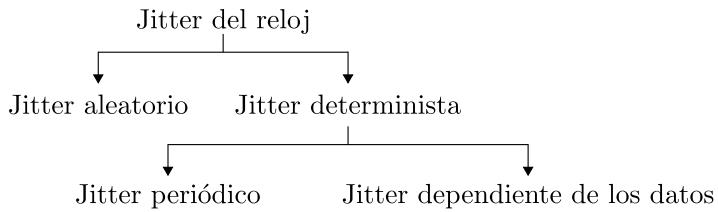


Figura 2.6: Composición del jitter del reloj.

semiconductores, las vibraciones térmicas de los átomos conductores y otros factores secundarios.

Los componentes deterministas del jitter no siguen ninguna ley probabilística debido a su naturaleza determinista, lo que los hace generalmente difíciles de caracterizar. El jitter determinista presenta otros dos componentes, el jitter periódico y el jitter dependiente de los datos. Estos componentes suelen ser causados por variaciones en la fuente de alimentación, diafonía (cross-talk), interferencias electromagnéticas (EMI), comunicaciones simultáneas de salidas y otras señales interferentes que ocurren de manera regular.

Cuando se trata de medir y cuantificar el jitter, una de las principales confusiones que surgen es que algunas métricas del jitter solo se aplican al jitter aleatorio y no al determinista, aunque este último está presente la mayor parte del tiempo.

En el contexto de los TRNG, los componentes deterministas del jitter no son deseables ya que no proporcionan ninguna fuente real de aleatoriedad. La aleatoriedad solo se puede extraer del jitter causado por ruidos aleatorios. Por lo tanto, es importante conocer las propiedades estadísticas del jitter antes de generar números aleatorios a partir de él.

Tanto el jitter aleatorio como el determinista pueden tener fuentes locales o globales. Las fuentes de jitter local afectan sólo a un área limitada del sistema electrónico, mientras que las fuentes de jitter global afectan a todo el sistema. Las fuentes locales de jitter suelen estar presentes en las proximidades de componentes de alta frecuencia y potencia, como osciladores y amplificadores. Las fuentes de jitter global son ruidos ambientales, ruidos procedentes de fuentes de alimentación, etc.

Desde un punto de vista estadístico, podemos distinguir entre ruidos independientes y dependientes. Los ruidos independientes no son fácilmente manipulables y suelen ser fáciles de caracterizar. Por esta razón, la mayoría de los diseños de TRNG utilizan la suma de ruidos independientes, también conocida como ruido gaussiano, como fuente de aleatoriedad. El desafío al diseñar un TRNG con ruido gaussiano es la necesidad de estimar solo la contribución de los ruidos no correlacionados (gaussianos) a los números aleatorios generados, descartando así las contribuciones de los ruidos dependientes a la aleatoriedad resultante.

2.2.1.5. Extracción de la aleatoriedad del jitter del reloj

El jitter del reloj se considera una buena fuente de aleatoriedad en los dispositivos digitales debido a que siempre está presente y contiene elementos aleatorios intrínsecos. Para generar números aleatorios a partir del jitter, es necesario digitalizarlo de alguna manera para producir bits aleatorios. El método más utilizado para extraer la aleatoriedad del jitter del reloj se basa en el muestreo del flanco de reloj con jitter. En la Figura 2.7 se muestra este método de extracción de aleatoriedad.

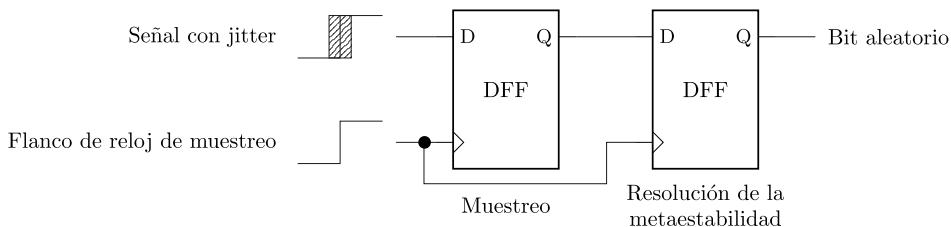


Figura 2.7: Extracción de aleatoriedad de la señal de reloj con jitter mediante su muestreo en el flanco ascendente de la señal de reloj de referencia.

Para producir bits aleatorios, la señal de reloj debe llegar durante el flanco afectado por el jitter de la señal con jitter. Esto requiere de una gran precisión en la sincronización del reloj, ya que el jitter es muy pequeño, normalmente del orden de los picosegundos o $1/1000$ del periodo del reloj. Además, incluso la señal de reloj de muestreo está sujeta a jitter, lo que aumenta la dificultad de una sincronización precisa. Por último, los bits aleatorios generados mediante el método de muestreo son propensos a estar sesgados, dependiendo en gran medida del ciclo de trabajo (duty cycle) del reloj muestreado. Si el ciclo de trabajo es del 50 %, la probabilidad de que el bit de salida sea 1 es del 50 %. Sin embargo, cuando el ciclo de trabajo está desequilibrado, la probabilidad de obtener 1 no es igual a la probabilidad de obtener 0, sino que es proporcional al ciclo de trabajo. A pesar de los inconvenientes de este método, sigue siendo el más utilizado para extraer aleatoriedad del reloj [9, 12, 34–36].

Una de las formas de convertir el jitter aleatorio en bits aleatorios es acumular el jitter hasta que su tamaño sea mayor que el periodo de la señal muestreada [9]. En ese caso, cada muestreo de dicha señal produciría un resultado completamente impredecible.

Uno de los TRNG que utiliza la acumulación de jitter es el TRNG basado en oscilador de anillo elemental (ERO-TRNG), propuesto y modelado en [9]. Su estructura interna se representa en la Figura 2.8. Consta de dos osciladores en anillo, un divisor de frecuencia y un flip-flop D.

Un divisor de frecuencia permite establecer un periodo más largo entre dos muestras del Oscilador de anillo 1, lo que permite que se acumule su jitter de fase. Cuando el valor de K es suficientemente grande, cada bit de salida es completamente impredecible

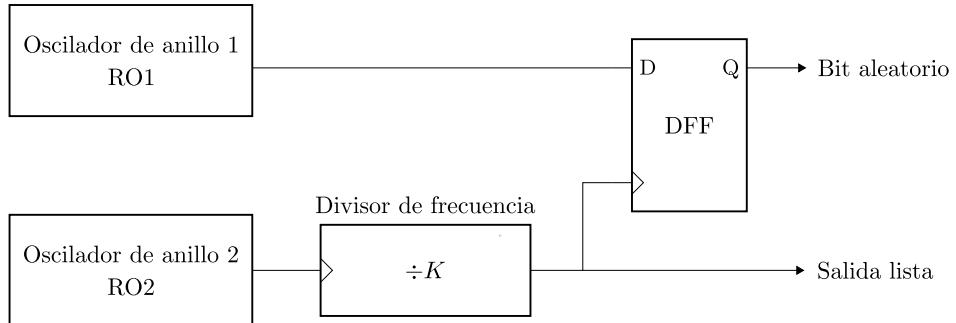


Figura 2.8: Elementary ring oscillator (ERO-TRNG).

debido a la acumulación del jitter.

En lugar de esperar más tiempo para acumular suficiente jitter, es posible reducir el tiempo de acumulación de jitter mediante el uso de múltiples relojes desfasados. Existen dos principios comunes para producir múltiples relojes desfasados: el primero consiste en utilizar múltiples osciladores independientes tal y como propone [34], mientras que el segundo se basa en utilizar múltiples salidas de un oscilador multifásico como propone [12].

2.2.2. Metaestabilidad

La metaestabilidad es la capacidad de un sistema para persistir en un estado ilegal durante un periodo de tiempo indefinido. Un ejemplo de metaestabilidad es el lanzamiento de una moneda, como se muestra en la Figura 2.9. Cuando se lanza una moneda, se espera que caiga sobre una de sus dos caras, las cuales representan estados legales. Sin embargo, cuando una moneda cae sobre su borde, el resultado del lanzamiento se vuelve incierto e ilegal, lo que se conoce como su estado metaestable. En contraste, cuando una moneda cae sobre alguna de sus caras, se encuentra en un estado estable. Para que una moneda permanezca en un estado metaestable, debe estar en un estado de equilibrio perfecto. La más mínima fuerza que se aplique hará que caiga sobre cualquiera de las dos caras.

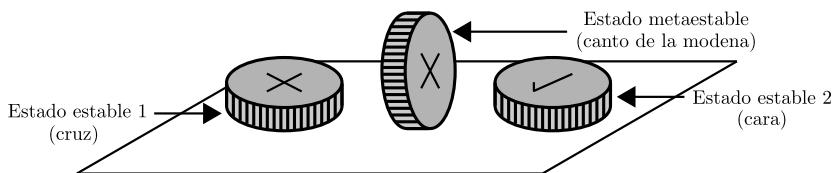


Figura 2.9: Metaestabilidad de un lanzamiento de moneda.

2.2.2.1. Metaestabilidad en dispositivos lógicos

En los dispositivos lógicos, puede producirse metaestabilidad en los registros durante el funcionamiento normal del dispositivo, cuando se violan los tiempos de establecimiento-

to y retención del registro. Los registros (flip-flops) requieren que una señal de entrada sea estable durante cierto tiempo antes del flanco de reloj (establecimiento) y también durante cierto tiempo después del flanco de reloj (retención). Sólo así se garantiza que el registro tenga un valor bien definido dentro de un retardo especificado a la salida. Si no se respetan los tiempos de establecimiento y retención, el registro puede caer en un estado metaestable, en el que permanece durante un periodo de tiempo indeterminado antes de volver a su valor anterior o al nuevo.

El hecho de que el registro acabe en el nuevo estado o en el anterior se determina por factores aleatorios, lo que hace que el resultado de la transición sea también aleatorio. Sin embargo, generar números aleatorios de esta forma es un gran desafío debido a la dificultad de lograr una sincronización suficientemente precisa entre las dos señales para que lleguen al registro al mismo tiempo. Esto se debe principalmente a los esfuerzos realizados por los fabricantes de dispositivos para reducir los tiempos de preparación y retención, evitando así los estados metaestables de los registros. En estos casos, los fabricantes de dispositivos realizan estudios exhaustivos de la vida útil de los dispositivos para determinar una tasa de fallos en el tiempo, failure in time (FIT) del dispositivo. La tasa de fallos en el tiempo está en el rango de una cada 10^9 horas. El FIT puede utilizarse para calcular el tiempo medio entre fallos, mean time between failures (MTBF), de un dispositivo y un diseño específicos. El MTBF sirve como estimación del tiempo entre dos fallos del sistema debidos a la metaestabilidad. Se calcula en función de las especificaciones del dispositivo y el diseño y su orden de magnitud típico es de decenas de años.

Teniendo en cuenta el MTBF típico, se tardaría mucho tiempo e incluso años en generar un bit aleatorio utilizando sólo la metaestabilidad de un circuito como fuente de aleatoriedad. Porque se considera que la metaestabilidad en sí no puede utilizarse para generar grandes cantidades de datos aleatorios.

2.2.2.2. Metaestabilidad oscilatoria

En los dispositivos electrónicos, también puede presentarse otro tipo de metaestabilidad conocida como metaestabilidad oscilatoria. A diferencia de la metaestabilidad en los registros, este comportamiento no conduce a un estado indefinido, sino que el sistema oscila entre estados bajos y altos durante un tiempo indeterminado. En un estudio realizado en [37], se demostró que es posible inducir metaestabilidad oscilatoria al introducir un retardo adicional en un circuito RS-latch. De esta forma, el circuito se inicializa en un estado ilegal para obtener su comportamiento metaestable.

En [11], se introdujo el oscilador en anillo de efecto transitorio, transient effect ring oscillator (TERO), que utiliza la metaestabilidad oscilatoria para generar aleatoriedad. Consiste en un RS-latch modificado, que se establece periódicamente en un estado ilegal

al establecerlo y reiniciarlo al mismo tiempo, violando de forma efectiva los tiempos de establecimiento y retención de dicho latch. La estructura interna de un TERO se muestra en la Figura 2.10.

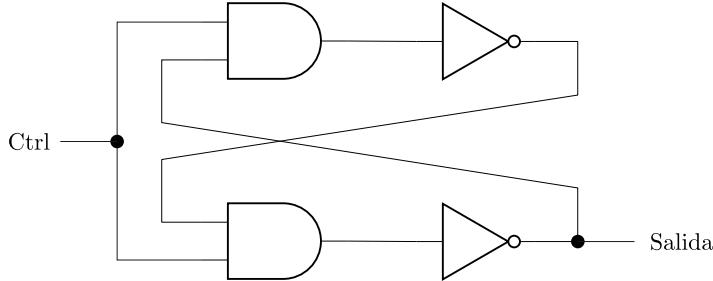


Figura 2.10: Estructura interna de un TERO

Cuando se activa una señal Ctrl, ver la Figura 2.10, el TERO pasa a un estado oscilatorio, en el que permanece durante un periodo de tiempo aleatorio. Después de la transición osculatoria, se establece en uno de los niveles lógicos (alto o bajo). Este estado final también es aleatorio. Sin embargo el número de oscilaciones en la salida del TERO y su estado final no son constantes. A diferencia de la metaestabilidad analógica, aquí la salida oscila entre dos estados definidos.

2.2.3. Modelos estocásticos y pruebas específicas

El modelo estocástico de un TRNG especifica una familia de distribuciones de probabilidad que contiene todas las posibles distribuciones de los números aleatorios sin procesar. Sus principales objetivos son caracterizar la probabilidad de que un bit de salida sea igual a uno ($P(X = 1)$) y la probabilidad de un vector de bits de cierto valor a la salida ($P(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n)$), y a partir de ellas estimar la tasa de entropía a la salida del generador.

El uso de modelos estocásticos resulta práctico únicamente cuando las probabilidades que definen se basan en parámetros medibles. Para ilustrar esto, examinaremos más de cerca los modelos estocásticos mediante un ejemplo de un TRNG elemental basado en osciladores de funcionamiento libre, tal y como se describe en [9]. Los autores de este trabajo proponen un modelo basado en la fase de la señal de reloj generada por un oscilador de funcionamiento libre y definen la probabilidad de que un bit de salida sea uno mediante la ecuación (2.8).

$$P(X = 1) = \frac{1}{2} - \frac{2}{\pi} \sin(2\pi(\mu t + \varphi(0))) e^{-2\pi^2 \sigma^2 t} + O(e^{-4\pi^2 \sigma^2 t}) \quad (2.8)$$

A partir de esta probabilidad, el límite inferior de entropía por bit a la salida del generador puede denotarse mediante la ecuación (2.9).

$$H_{\min} \approx 1 - \frac{4}{\pi^2 \ln(2)} e^{-4\pi^2 Q} = 1 - \frac{4}{\pi^2 \ln(2)} e^{\frac{-4\pi^2 \sigma_{\text{jit}}^2 T_2}{T_1^3}} \quad (2.9)$$

Podemos ver que la entropía depende de tres parámetros medibles: los periodos del oscilador (T_1, T_2) y el jitter combinado (σ_{jit}). Así que midiendo estos parámetros podemos estimar la entropía a la salida de los generadores. Esto es importante por dos razones: el TRNG puede caracterizarse en la fase de diseño y la tasa de entropía en su salida puede controlarse durante el funcionamiento normal del generador. Los periodos de los dos osciladores son fáciles de medir incluso dentro del dispositivo. La principal dificultad en la estimación de la entropía en línea está relacionada con la medición de σ_{jit}^2 sabiendo que sólo la contribución del ruido térmico debe tenerse en cuenta en el cálculo de la tasa de entropía. En este sentido, en [38] se propone un método se propone un método de evaluación embebida de la aleatoriedad (tasa de entropía por bit) basado en la medida de la varianza del jitter.

Se puede concluir que si el tamaño de los tres parámetros del modelo antes mencionados puede obtenerse a partir del límite inferior de entropía requerido, puede utilizarse una medición en línea de estos parámetros para verificar que el generador no desciende por debajo de este valor. Esta medición y la comparación con los umbrales obtenidos a partir del modelo constituyen una base para las pruebas en línea específicas.

2.3. Estándares de diseño y certificación de TRNG

Diferentes aplicaciones criptográficas requieren distintos niveles de seguridad, lo que ha llevado a la necesidad de estandarizar el uso de la criptografía para aplicaciones específicas. Ya existen muchos estándares que proporcionan algoritmos estándar para el cifrado como el AES [39, 40] o funciones hash como el SHA-3 [41]. Sin embargo, debido a que los TRNG son específicos de cada tecnología y plataforma, no es posible proporcionar un diseño estandarizado. Por esta razón, varias autoridades de certificación han desarrollado enfoques estándar para la certificación de TRNG.

El primer intento de certificación del diseño de TRNG se centró únicamente en comprobar las propiedades estadísticas de los números generados. Para ello se propusieron muchos conjuntos de pruebas, como FIPS 140-1, DIEHARD, NIST 800-22 [7], TestU01 entre otros. La idea detrás de las pruebas estadísticas de los TRNG es que deben producir una secuencia de salida que no se distinga de la secuencia ideal, la cual es estacionaria, uniformemente distribuida y con muestras independientes. Aunque las propiedades estadísticas pueden comprobarse fácilmente mediante pruebas estadísticas adecuadas, su independencia no puede ser verificada. El problema con este tipo de pruebas para TRNG es que consideran al generador como una caja negra y solo se tiene en cuenta su salida. Si se utiliza un buen generador de números pseudoaleatorios

en lugar de un TRNG, su resultado superará todas las pruebas estadísticas que se le realicen debido a que están hechos de algoritmos que producen secuencias con propiedades estadísticas perfectas, aunque su resultado no es realmente aleatorio, solo parece serlo, ya que su comportamiento está definido y es predecible como resultado de los algoritmos utilizados.

La evaluación de las propiedades estadísticas de un TRNG es necesaria, pero claramente no suficiente. Es por esto que existen normas más recientes que exigen la caracterización de la fuente de ruido, además de las pruebas estadísticas. Estas normas son la AIS-20/31 [6] de la Oficina Federal Alemana de Seguridad de la Información (BSI) y la NIST 800-90B [5] del Instituto Nacional de Estándares y Tecnología de Estados Unidos. Aunque las dos normas utilizan una metodología de evaluación ligeramente diferente, comparten la misma idea básica de la arquitectura de un TRNG. La Figura 1.1 muestra un diagrama de bloques de un TRNG de acuerdo con las especificaciones de AIS-20/31 y NIST 800-90B.

La fuente de ruido digital está compuesta por una fuente de ruido analógica y un digitalizador (extractor de aleatoriedad). Este bloque es el único que extrae la verdadera aleatoriedad del proceso subyacente, y proporciona la salida de señal binaria en bruto. Su salida debe estar disponible para su evaluación con el fin de comprobar la calidad de la señal en bruto y estimar la tasa de entropía a la salida. Es importante destacar que el postprocesamiento es un proceso algorítmico y, por lo tanto, es posible calcular fácilmente la tasa de entropía a la salida del generador a partir de la tasa de entropía a su entrada, que se conoce a partir del modelo y que puede ser verificada probando los datos de la señal en bruto. Es importante señalar que según la norma AIS-20/31, el postprocesamiento no debe reducir la entropía. Ambas normas, AIS-20/31 y NIST 800-90B, consideran el postprocesamiento como una opción. No obstante lo ideal sería que un buen TRNG no necesitara ningún postprocesamiento algorítmico.

Las pruebas integradas son una parte obligatoria del diseño de un TRNG. Según AIS-20/31, estas pruebas incluyen al menos dos tipos: la prueba de fallo total y las pruebas en línea. La prueba de fallo total debe ser capaz de detectar rápidamente la pérdida completa de entropía en la fuente, con una baja probabilidad de falsas alarmas. Esta pérdida total de entropía podría deberse a un cambio rápido o a la pérdida completa de la conexión física con la fuente. Por otro lado, las pruebas en línea deben ser capaces de detectar defectos irreparables en la secuencia de salida. Estos defectos irreparables son fallos estadísticos que no pueden corregirse con el postprocesamiento, y pueden ocurrir cuando el dispositivo funciona fuera de su rango de parámetros de funcionamiento, como en situaciones de sobretensión, subtensión o temperaturas extremas. Las pruebas en línea pueden ser ejecutadas de manera continua, bajo demanda o activadas por un evento interno específico. Es importante destacar que estas pruebas deben ser superadas con éxito cada vez que se enciende o reinicia el TRNG, por lo que

también funcionan como pruebas de encendido. [33]

2.3.1. Resumen de los requisitos del AIS-20/31

La norma AIS-20/31 reconoce varias clases diferentes de TRNGs en función de su principio de funcionamiento:

- Tres clases PTG para los generadores de números aleatorios verdaderos físicos (PTRNG), clases PTG.1 a PTG.3.
- Cuatro clases de DRG para los generadores aleatorios deterministas (DRNG), clases DRG.1 a DRG.4.
- Una clase NTG para los generadores de números aleatorios verdaderos no físicos, (NPTRNG), clase NTG.1.

Todas las clases de TRNG deben superar las pruebas estadísticas de caja negra. Estas pruebas se dividen en dos procedimientos de prueba descritos en la norma AIS-20/31:

- El procedimiento de prueba A contiene las pruebas estadísticas T0 a T5. Estas pruebas verifican las propiedades estadísticas generales, como el sesgo, y están destinadas a comprobar los datos postprocesados.
- El procedimiento de prueba B contiene las pruebas T6 a T8. Las pruebas T6 y T7 detectan la dependencia entre los números generados. La prueba T8 compara la entropía de Shannon estimada por bit con un umbral de 0.997. El procedimiento de prueba B está destinado a comprobar los datos sin procesar.

Las pruebas estadísticas estándar descritas en AIS-20/31 son:

- T0: Prueba de disyunción
- T1: Prueba de monobit
- T2: Prueba de póquer
- T3: Prueba de corridas
- T4: Prueba de corridas larga
- T5: Prueba de autocorrelación
- T6: Prueba de distribución uniforme
- T7: Prueba de homogeneidad
- T8: Estimación de la entropía

2.3.1.1. Clase PTG.1 TRNG

PTG.1 es una clase de TRNG físico de baja seguridad destinado a aplicaciones que no son críticas para la seguridad. La Figura 2.11 muestra el diagrama de bloques de dicho generador y los puntos de prueba requeridos por la norma.

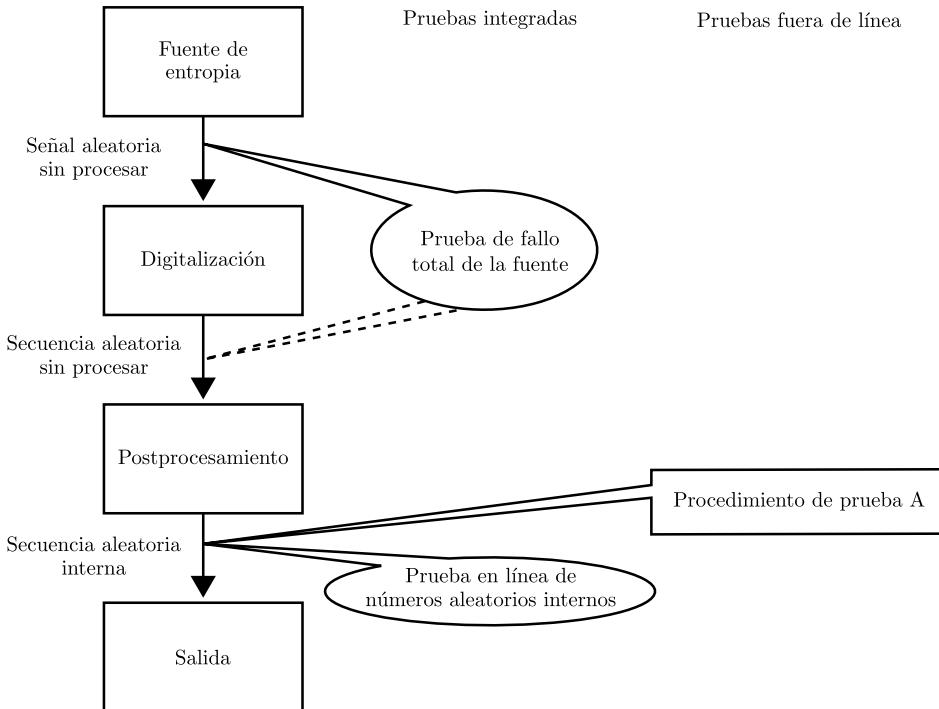


Figura 2.11: Clase PTG.1 TRNG.

Requisitos de la fuente de aleatoriedad

No se requiere ningún modelo estocástico para un PTG.1 TRNG. Sin embargo, la fuente de ruido debe ser física, estar claramente definida y descrita para que quede claro el origen de la aleatoriedad.

Pruebas integradas

La clase PTG.1 requiere la implementación de pruebas de fallo total y en línea. La prueba de fallo total debe detectar un fallo completo de la fuente de aleatoriedad. Las pruebas en línea deben supervisar continuamente y garantizar la calidad estadística de los números aleatorios producidos. La clase PTG.1 requiere que las pruebas en línea supervisen la calidad de los números aleatorios internos (es decir, a la salida del generador).

Postprocesamiento

La clase PTG.1 no requiere que el TRNG utilice ningún tipo de postprocesamiento. Tampoco desaconseja el uso del postprocesamiento. Sin embargo, un TRNG PTG.1 debe superar las pruebas estadísticas del procedimiento de prueba A, por lo que el postprocesamiento puede ser necesario si la señal binaria sin procesar no puede superar dichas pruebas.

2.3.1.2. Clase PTG.2 TRNG

PTG.2 es una clase de TRNG físico, que puede utilizarse para generar claves criptográficas, nonces, semillas para DRNGs, etc. En comparación con la clase PTG.1 de baja seguridad, el TRNG PTG.2 debe garantizar el secreto de los números aleatorios producidos (su imprevisibilidad). La Figura 2.12 muestra la estructura interna y las pruebas necesarias para el TRNG PTG.2.

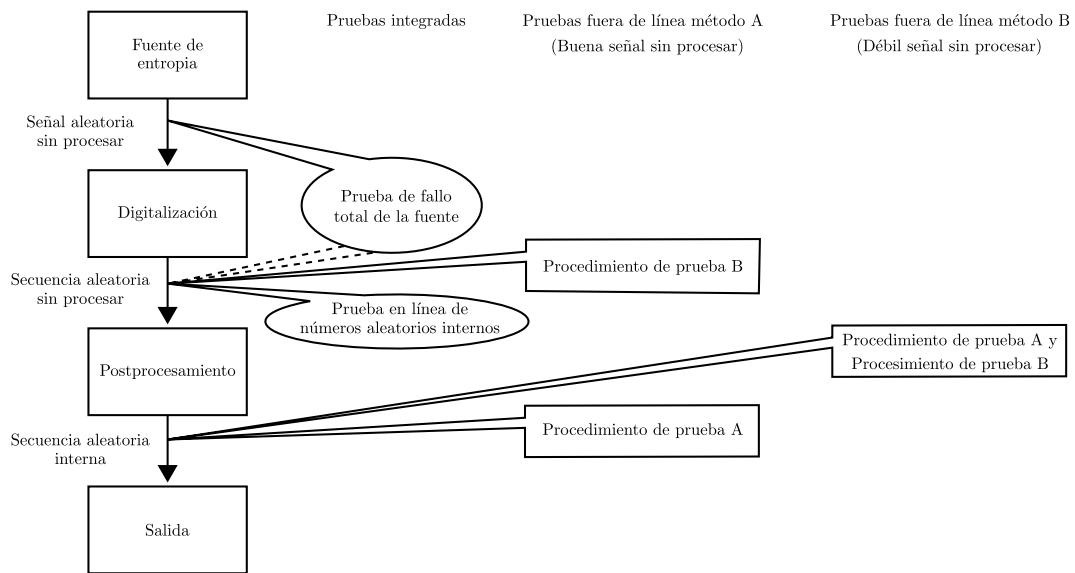


Figura 2.12: Clase PTG.2 TRNG.

Requisitos de la fuente de aleatoriedad

Todos los requisitos de la clase PTG.1 se aplican también a la clase PTG.2. Además, se requiere un modelo estocástico para la fuente de aleatoriedad. El modelo estocástico debe tener en cuenta el comportamiento de la fuente de aleatoriedad. Basándose en los parámetros de la fuente, el modelo estima la entropía de la señal binaria sin procesar. La entropía de Shannon de la señal binaria sin procesar debe ser superior a 0.997 por bit según la norma AIS-20/31.

Pruebas integradas

Para un TRNG PTG.2 es necesario implementar tanto las pruebas de fallo total como las pruebas en línea. La prueba de fallo total debe detectar un fallo de la fuente de aleatoriedad total.

Las pruebas en línea deben detectar las debilidades estadísticas intolerables de la señal binaria sin procesar. Deben funcionar con una señal binaria sin procesar porque el uso del postprocesamiento podría enmascarar algunos defectos potencialmente peligrosos. Las pruebas en línea deben adaptarse al modelo estocástico. De este modo,

pueden detectar los defectos específicos de la fuente de aleatoriedad utilizada de forma muy eficaz.

Postprocesamiento

De forma similar a la clase PTG.1, la clase PTG.2 no requiere el postprocesamiento cuando la señal binaria sin procesar proporciona números aleatorios de suficiente calidad. Si el postprocesamiento es necesario, la clase PTG.2 no pone ninguna restricción en el algoritmo utilizado, sin embargo, no debe reducir la entropía.

2.3.1.3. Clase PTG.3 TRNG

PTG.3 es una clase TRNG híbrida para TRNGs de alta seguridad. Los TRNG de esta clase no se basan únicamente en la seguridad proporcionada por la fuente de aleatoriedad, sino que añaden una segunda ancla de seguridad en forma de postprocesamiento criptográficamente seguro. Un TRNG híbrido es un TRNG compuesto por un TRNG físico, que vuelve a alimentar continuamente al RNG determinista. En este caso, el RNG determinista sirve como postprocesamiento para el TRNG físico que genera entropía. La Figura 2.13 muestra el diagrama de bloques de un TRNG híbrido de este tipo.

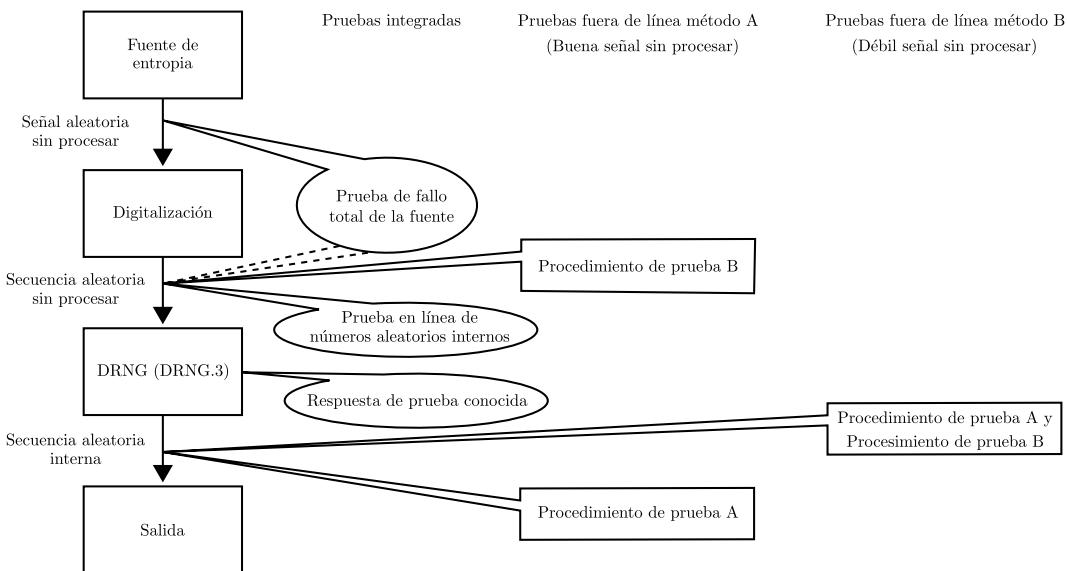


Figura 2.13: Clase PTG.3 TRNG.

Requisitos de la fuente de aleatoriedad

La fuente de aleatoriedad del TRNG PTG.3 debe cumplir todos los requisitos de la clase PTG.2. La entropía de Shannon de la señal binaria sin procesar debe ser superior a 0.997 por bit, lo que debe estar garantizado por el modelo estocástico de la fuente.

Pruebas integradas

Además de las pruebas de fallo total y en línea que se exigen también para el TRNG PTG.2, la clase PTG.3 requiere también la prueba de respuesta conocida (KAT) para el postprocesamiento. Esta prueba debe pasar con éxito cada vez que se inicie o reinicie el TRNG para verificar el correcto funcionamiento del algoritmo de postprocesamiento.

Postprocesamiento

A diferencia de los requisitos de PTG.1 y PTG.2, la clase PTG.3 requiere el uso de un postprocesamiento criptográfico. Además, requiere el uso de un DRNG de la clase DRG.3, que proporciona secreto hacia delante y hacia atrás mejorado. Esto significa que el postprocesamiento para un TRNG de clase PTG.3 debe ser una función criptográfica.

2.3.1.4. Clase DRG.1 DRNG

La clase DRG.1 establece los requisitos para los RNG deterministas. Los RNG que cumplen con esta norma deben generar una secuencia de números aleatorios que sea indistinguible de la secuencia generada por un RNG ideal a través de simples pruebas estadísticas de caja negra. Además, los DRNG conformes con DRG.1 proporcionan secreto hacia adelante, lo que significa que la seguridad de la secuencia generada no se ve comprometida aunque un atacante tenga acceso a una porción previa de la secuencia. La Figura 2.14 muestra el diagrama de bloques de este generador.

Los RNG que cumplen con la norma DRG.1 pueden ser útiles en aplicaciones que requieren datos frescos que difieran de las secuencias previamente generadas con una alta probabilidad. Por ejemplo, pueden ser utilizados para generar retos en protocolos criptográficos o vectores de inicialización en cifradores de bloques en modos especiales de funcionamiento, siempre que no sea necesario proteger la secuencia previa de números aleatorios. Los DRNG que cumplen con DRG.1 son también adecuados para pruebas de conocimiento cero. [6]

Para generar la semilla del DRNG se puede utilizar el PTRNG PTG.2 o PRG.3. Si el estado interno es al menos un 25 % mayor en bits que los límites de entropía mínima, no es necesario una evaluación explícita de la entropía mínima. Esto se justifica por el hecho de que los PTRNG PTG.2 y PTG.3 generan números aleatorios de alta entropía.

φ es la función de transición de estado y ψ es la función de salida

2.3.2. Resumen de los requisitos del NIST 800-90B

El NIST 800-90B exige, al igual que el AIS-20/31, que la secuencia de salida del TRNG pase las pruebas de caja negra. Estas pruebas de caja negra se dividen en dos caminos:

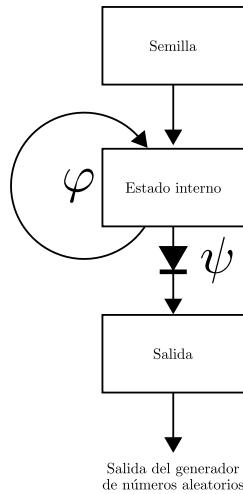


Figura 2.14: Clase DRNG.1.

- IID track, se utiliza para datos independientes e idénticamente distribuidos o Independent and Identically Distributed en inglés (IID).
- Non-IID track, se utiliza para los datos que no superan la prueba de detección de IID.

Además de superar las pruebas de caja negra, el NIST 800-90B también impone requisitos a los bloques individuales del TRNG.

Fuente de ruido

El NIST 800-90B tiene los siguientes requisitos sobre la fuente de ruido:

- Su comportamiento debe estar descrito..
- Su salida debe ser estacionaria.
- Debe indicarse la entropía de salida esperada.
- Debe protegerse de la observación e influencia de los adversarios.
- La fuente de ruido debe tener un comportamiento aleatorio.

Aunque la norma exige la declaración de estacionariedad de la salida y de entropía esperada, no requiere ninguna prueba matemática de estas afirmaciones. Sólo es necesaria la descripción técnica de por qué se cree que la fuente de ruido tiene el comportamiento afirmado.

Pruebas de salud

El NIST 800-90B exige tres tipos de pruebas de salud.

- Las pruebas de puesta en marcha deben verificar si todos los componentes necesarios de la fuente de ruido funcionan correctamente. La fuente de ruido no debe emitir datos antes de que las pruebas de puesta en marcha se completen con éxito.

- Las pruebas continuas comprueban los defectos y fallos en el comportamiento de la fuente de ruido. Estas pruebas se realizan de forma continua en todas las muestras emitidas por la fuente de ruido. El NIST 800-90B exige que se utilicen dos pruebas continuas aprobadas. Además de las pruebas aprobadas, también se pueden utilizar pruebas definidas por el desarrollador. La norma permite a los desarrolladores no utilizar las pruebas aprobadas, pero en su lugar deben utilizarse otras pruebas continuas que pueden detectar el mismo tipo de defectos que las pruebas aprobadas.
- Las pruebas bajo demanda no se realizan hasta que se solicitan. Debe haber una forma de realizar pruebas bajo demanda en la fuente de ruido. Las muestras utilizadas para las pruebas bajo demanda no deben salir hasta que las pruebas se completen con éxito.

Acondicionamiento

El NIST 800-90B entiende por componente acondicionador una función determinista responsable de reducir el sesgo y/o aumentar la tasa de entropía de los bits de salida resultantes [5]. El componente de acondicionamiento es completamente opcional, por lo que puede omitirse por completo.

La entropía de salida del TRNG se estima después del componente de condicionamiento. El estándar proporciona una lista de algoritmos de condicionamiento comprobados, para los que los desarrolladores pueden reclamar una entropía completa, aunque esta reclamación tiene que ser validada. Estos algoritmos son funciones hash combinadas con cifrado de bloques, para ver la lista de componentes de acondicionamiento verificados ver [5].

Se permite el uso de componentes de acondicionamiento no validados, pero estos componentes son penalizados en términos de estimación de entropía. Cuando se utiliza un componente condicionante no validado, la entropía en la salida se multiplica por una constante de 0.999, lo que le impide alcanzar la entropía total.

Los parámetros recomendados para configurar las pruebas NIST se muestran en la Tabla 2.1:

2.3.3. Conclusiones de la certificación de seguridad TRNG

Hay dos normas principales relativas a los TRNG que están en vigor en la actualidad:

- AIS-20/31 utilizado en muchos países europeos. [6]
- NIST 800-90B utilizado en los Estados Unidos. [5]

Las dos normas existentes en la actualidad exigen que el diseño de un TRNG esté bien documentado y su funcionamiento interno bien descrito, además de las pruebas

Tabla 2.1: Parámetros recomendados para el conjunto de pruebas del NIST.

Test	Configuration item	Setting
All tests	Bits per sequence	1000000
All test	Number of sequences (sample size)	1073
Frequency test within a block	Block length	20000
Non-overlapping template test	Template length	10
Overlapping template	Block length	10
Maurer's Universal Statistical test	Test block length L	7
Maurer's Universal Statistical test	Initialization steps	1280
Approximate entropy test	Block length	8
Linear complexity test	Block length	1000
Serial test	Block length	16

estadísticas. También exigen que se realicen pruebas integradas, de modo que un TRNG se supervise continuamente durante su funcionamiento.

Sin embargo, la norma AIS-20/31 profundiza en el problema de la descripción del TRNG y exige que se desarrolle también un modelo estocástico. Las pruebas integradas, según la AIS-20/31, también deben implementarse de acuerdo con el modelo estocástico. Este requisito no se aplica en la norma NIST 800-90B, que da más libertad en el diseño del TRNG, pero no en sus pruebas.

2.4. Arquitecturas de núcleos TRNGs en FPGA

El cumplimiento de la norma AIS-20/31 significa que los TRNG seleccionados deben tener una fuente de aleatoriedad claramente definida y bien descrita. El modelo estocástico debe ser factible y la salida de datos sin procesar debe estar disponible para las pruebas. La fuente de aleatoriedad también debe ser cuantificable, lo que permitiría su medición dentro del dispositivo. Una medición de este tipo puede constituir una base sólida para realizar pruebas integradas rápidas y eficaces.

Además del cumplimiento con la norma AIS-20/31, queremos diseños que sean factibles en cualquier dispositivo lógico. Dado que buscamos un diseño general, evitaremos características (por ejemplo, componentes analógicos) que sean específicas sólo para determinadas tecnologías. Dado que los diseños generales serían independientes de la tecnología, también deberían ser factibles en FPGAs.

Con base en los criterios del AIS-20/31, los núcleos TRNG adecuados para utilizarse en dispositivos lógicos programables (FPGA) que usan estructuras oscilantes son [2]:

- Single-event ring oscillators
 - Elementary ring oscillator based TRNG
 - Coherent sampling ring oscillator based TRNG

- Multi-ring oscillator based TRNG
- Multi-event ring oscillators with signal collisions
 - Transient effect ring oscillator based TRNG
- Multi-event ring oscillators without signal collisions
 - Self-timed ring based TRNG
- Phase-locked loops
 - PLL based TRNG

Todos los TRNG preseleccionados deberían ser viables en todas las familias de FPGA recientes y futuras, ya que no utilizan ninguna característica específica de la familia.

2.4.1. Elementary ring oscillator based TRNG (ERO-TRNG)

El ERO-TRNG se propuso y modeló en [9]. Dos osciladores en anillo idénticos forman la base del generador. Uno de ellos se utiliza para generar una señal de muestreo, que luego se utiliza para muestrear la salida del otro oscilador en anillo utilizando un flip-flop D (DFF). La frecuencia del oscilador de anillo de muestreo se divide por K para obtener una frecuencia más baja de la señal de muestreo, lo que permitiría acumular el jitter del oscilador de anillo muestreado. La Figura 2.15 muestra una arquitectura del ERO-TRNG tal y como se implementa en FPGAs.

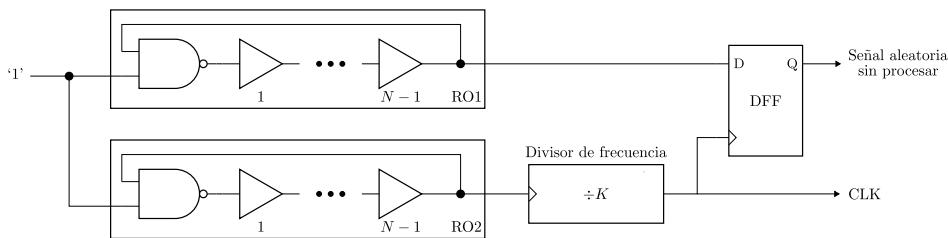


Figura 2.15: Arquitectura del núcleo ERO-TRNG.

Se utilizan osciladores en anillo compuestos por una puerta NAND y $N - 1$ búferes no inversores para construir un anillo de N elementos. La compuerta NAND puede utilizarse para encender o apagar el oscilador. En FPGA de Xilinx Spartan-6, con 3 elementos, (una compuerta NAND y 2 buffers), se obtienen frecuencias oscilantes en torno a 300 MHz, además el jitter de periodo se mantiene en torno a 4 ps para periodos entre 3 y 8 ns.

Los dos parámetros de diseño fundamentales del ERO-TRNG son la frecuencia de los osciladores de anillo y el factor de división K del reloj de referencia. El segundo

parámetro de diseño, el divisor K , determina el periodo de acumulación de jitter. El periodo de acumulación necesario depende del tamaño del jitter de fase y del límite inferior de entropía. El límite inferior de entropía viene definido por el modelo estocástico y, para el ERO-TRNG, puede calcularse mediante la ecuación (2.10).

$$H_{\min} = 1 - \frac{4}{\pi^4 \ln(2)} e^{-\frac{\pi^2 \sigma_{th}^2 K T_2}{T_1^3}} \quad (2.10)$$

donde σ_{th}^2 es la varianza del jitter debida al ruido térmico, K es el factor de división de la frecuencia de referencia y T_1, T_2 son los periodos de oscilación de los dos osciladores en anillo. Además se tiene la relación:

$$\sigma_{\text{jit}} = \sqrt{\frac{K \sigma_{th}^2}{4}} \quad (2.11)$$

Para un periodo de oscilación de 3 ns, sólo es necesario encontrar el tamaño de jitter de periodo de los osciladores en anillo. Mediciones del jitter de periodo para FPGAs de Xilinx Spartan-6 correspondientes a un periodo de 3 ns arrojan un $\sigma_T \approx 4ps$. Realizando los calculamos, el factor de división K según la ecuación (2.10) es $K = 80000$.

La implementación del ERO-TRNG es sencilla y los resultados son repetibles sin ninguna intervención manual. La colocación manual de los osciladores de anillo proporciona un mejor control de la frecuencia oscillatoria resultante, pero no es necesaria para el diseño adecuado del TRNG. La colocación de los osciladores de anillo ayuda a conservar las mismas propiedades de los osciladores a lo largo de diferentes proyectos o diferentes iteraciones de un proyecto [42].

El ERO-TRNG proporciona una tasa de bits de salida relativamente baja, porque K tiene que ser relativamente alto para garantizar una entropía suficiente. Sin embargo, una vez configurado correctamente, este TRNG ofrece una gran seguridad gracias a su sólido modelo estocástico. Las pruebas integradas sólo tienen que comprobar que los osciladores de anillo oscilan y que no están bloqueados [43, 44].

El consumo de energía del oscilador en anillo no depende de su tamaño (número de elementos), porque sólo se propaga un evento por el anillo. Por lo tanto, sólo un elemento del oscilador en anillo está activo (cambia de estado) a la vez, independientemente de la frecuencia del oscilador. El consumo de energía de todo el TRNG depende de las frecuencias de los osciladores en anillo utilizados.

2.4.2. Coherent sampling ring oscillator based TRNG (COSO-TRNG)

El COSO-TRNG se propuso por primera vez en [10]. Utiliza dos osciladores de anillo implementados idénticamente como fuente de aleatoriedad. La Figura 2.16 muestra la

estructura interna del COSO-TRNG.

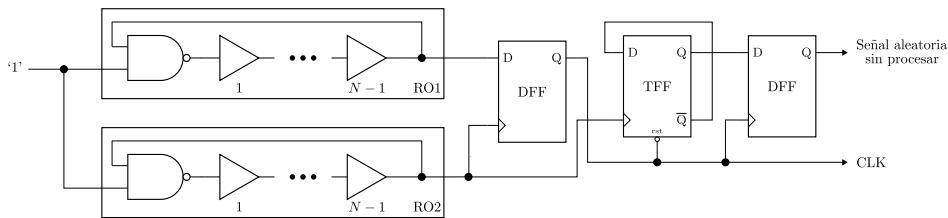


Figura 2.16: Arquitectura del núcleo COSO-TRNG.

Aunque los dos osciladores en anillo tengan exactamente la misma estructura interna, sus frecuencias varían un poco. Esta pequeña variación provoca una diferencia de fase en las salidas de los osciladores de anillo. Al muestrear la salida de uno de los osciladores mediante un flip-flop D (DFF) sincronizado con la salida del otro oscilador, obtenemos una señal con un periodo variable, que corresponde al desfase relativo de los dos osciladores.

El segundo flip flop es un flip-flop T (TFF), que corresponde a un contador de un bit que cuenta el número de flancos ascendentes de la señal del DFF durante un semiperíodo de la señal. Este último bit del contador se registra en el último DFF y se envía a la salida del TRNG como un bit aleatorio.

COSO-TRNG extrae aleatoriedad del jitter sólo si se cumple la condición de la ecuación (2.12).

$$\Delta_T < \Delta_{T_{\max}} = \sqrt[3]{\sigma_T^2 \cdot T} \quad (2.12)$$

Desgraciadamente, esta condición es muy difícil de cumplir, ya que los dos osciladores de anillo deben oscilar a frecuencias muy cercanas evitando el bloqueo entre ellos. En una FPGA, satisfacer esta condición es muy difícil ya que no tenemos un control preciso sobre la colocación y el enrutamiento de los anillos, es muy difícil, si no imposible, implementar dos anillos oscilando a frecuencias suficientemente cercanas pero no idénticas.

Debido a la extrema sensibilidad a Δ_T , el diseño que funciona en una FPGA no es directamente transferible a otra FPGA incluso de la misma familia porque incluso el más mínimo cambio causado por la variación del proceso de fabricación puede hacer que la diferencia de periodo oscile por encima de $\Delta_{T_{\max}}$. Así que el diseño debe colocarse y encaminarse manualmente para cada FPGA individual, lo que lo hace muy poco práctico.

Sin embargo, COSO-TRNG puede proporcionar una tasa de bits de salida relativamente alta con una huella de área baja.

El consumo de energía del COSO-TRNG es muy reducido, ya que la energía consumida por los osciladores en anillo es independiente de su tamaño y el COSO-TRNG

sólo presenta tres flip flops que aumentan su consumo de energía.

2.4.3. Multi-ring oscillator based TRNG (MURO-TRNG)

El MURO-TRNG utiliza múltiples osciladores en anillo, que se supone que son independientes, tienen la misma frecuencia media y fases uniformemente distribuidas. Para extraer de forma fiable la aleatoriedad de un grupo de osciladores en anillo, el número de osciladores en anillo debe satisfacer la condición de la ecuación (2.13), que especifica la relación entre el periodo medio de oscilación de todos los osciladores, su jitter y el número de osciladores utilizados. La Figura 2.17 muestra la estructura interna del MURO-TRNG.

$$m > \frac{T}{\sigma} \quad (2.13)$$

El principio TRNG sólo funcionará si las fases de los osciladores de anillo están uniformemente distribuidas. Sin embargo, los osciladores en anillo pueden bloquearse entre sí, en cuyo caso la distribución de las fases no será uniforme. Es más, la probabilidad de bloqueo es alta dado el elevado número de anillos necesarios para obtener una entropía alta en el MURO-TRNG.

El MURO-TRNG no requiere ninguna colocación o encaminamiento manual y su tasa de bits de salida, así como la tasa de entropía, son muy altas cuando se utiliza un número suficiente de osciladores de anillo. El consumo de energía de este TRNG es considerable debido al elevado número de osciladores.

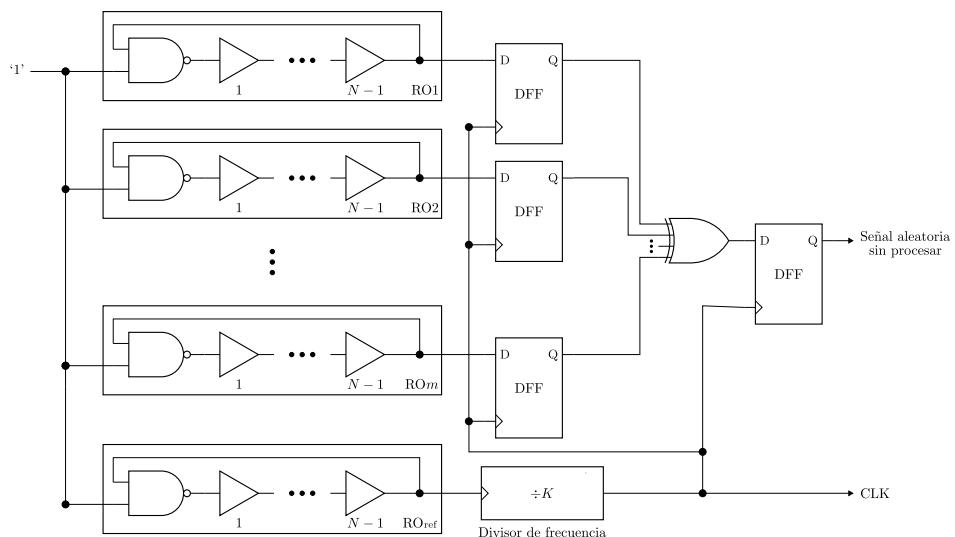


Figura 2.17: Arquitectura del núcleo MURO-TRNG.

2.4.4. Transient effect ring oscillator based TRNG (TERO-TRNG)

El TERO-TRNG genera bits aleatorios utilizando metaestabilidad osculatoria, la Figura 2.18 muestra la implementación del TERO-TRNG.

El flip-flop TFF representa el último bit del contador, que cuenta el número de oscilaciones de la celda TERO. Debido a la metaestabilidad osculatoria del TERO, el número de oscilaciones es aleatorio. Para producir un flujo de bits aleatorios, el TERO debe reiniciarse periódicamente [45].

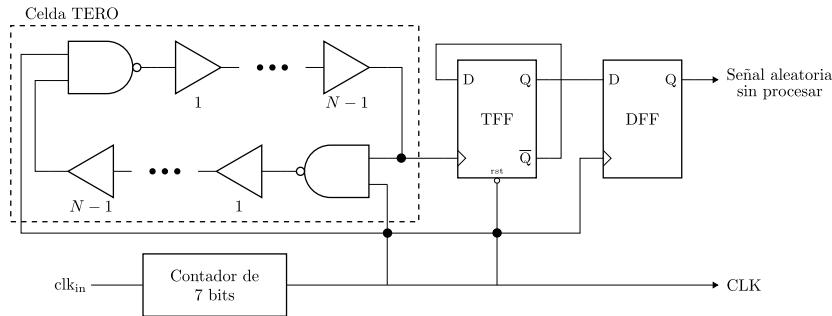


Figura 2.18: Arquitectura del núcleo TERO-TRNG.

Para conseguir suficiente entropía en la salida del TERO-TRNG, el número de oscilaciones del TERO debe estar dentro de los límites especificados por la ecuación (2.14).

$$100 < M < \frac{T_{\text{meas}}}{T_{\text{osc}}} \quad (2.14)$$

donde M es el número de oscilaciones, T_{meas} es el tiempo de medida y T_{osc} es el periodo de la señal de salida del TERO.

El límite inferior para el número de oscilaciones garantiza que habrá suficientes oscilaciones de las que extraer la aleatoriedad. El límite superior, en cambio, evita los casos en los que las oscilaciones no se detienen antes de reiniciar la medición. Cumplir esta condición de forma fiable para varios dispositivos es difícil porque la célula TERO se comporta de forma diferente en cada dispositivo (incluso dentro de una misma familia) aunque se utilice la misma configuración.

La celda TERO en sí no requiere una gran superficie y sólo se necesitan unos pocos flip flops para los contadores y el núcleo TRNG [46], por lo que la superficie total ocupada es relativamente pequeña. La energía consumida por el TERO no depende del número de elementos del TERO, pero el consumo de energía de los flip flops puede aumentar con el incremento de la frecuencia de reloj. El TERO-TRNG requiere una colocación y enrutamiento manuales y el diseño no es repetible ni siquiera en dispositivos de la misma familia.

2.4.5. Self-timed ring based TRNG (STR-TRNG)

Un anillo autotemporizado es un oscilador multievento sin colisiones de señales. La Figura 2.19 muestra la implementación del SRT-TRNG.

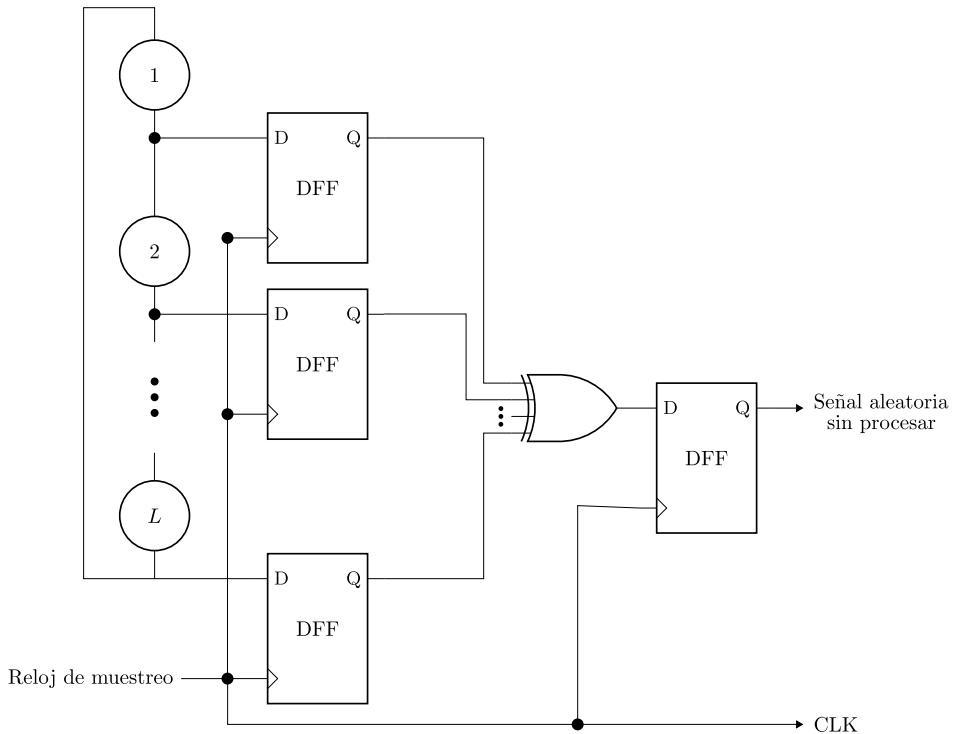


Figura 2.19: Arquitectura del núcleo STR-TRNG.

Un STR se compone de celdas L Muller (elementos C). Múltiples eventos pueden propagarse a través de un STR sin colisiones. Debido a las propiedades temporales de las STR, los eventos pueden propagarse en los dos modos: modo ráfaga y modo uniforme.

El principio de extracción de aleatoriedad de un STR-TRNG es el mismo que el de MURO-TRNG: el uso de un conjunto de señales de reloj con fases distribuidas uniformemente. Pero mientras que en muchos osciladores de anillo las fases se distribuyen estadísticamente, en STR la uniformidad de la distribución está garantizada por el principio.

El STR-TRNG consume mucha energía debido a que hay muchos eventos propagándose por el anillo. En el modo uniformemente espaciado, un STR oscila a su frecuencia máxima. La tasa de bits de salida del STR-TRNG es muy alta, lo que también contribuye a su elevado consumo de energía. El STR-TRNG requiere una gran superficie y, para garantizar que funciona en modo uniformemente espaciado, es necesario colocar manualmente los elementos STR.

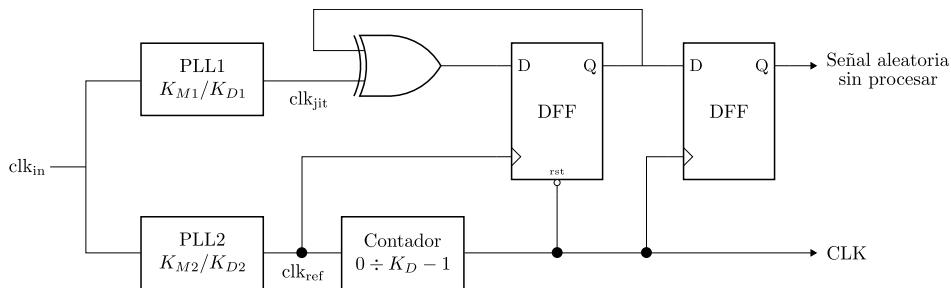


Figura 2.20: Arquitectura del núcleo PLL-TRNG.

2.4.6. PLL based TRNG (PLL-TRNG)

El PLL-TRNG no requiere ninguna colocación o enrutamiento manual. Proporciona alta seguridad y no se ve afectado por el ruido determinista global (es decir, dependiente de los datos), ya que los PLL están físicamente aislados del resto de la FPGA. La tasa de bits de salida del PLL-TRNG también es considerable [47].

El diseño del PLL-TRNG es sencillo, repetible y puede automatizarse. Sin embargo, la elección de los parámetros del PLL no es trivial. Hay que respetar muchas restricciones, incluidas las físicas del fabricante del PLL y las de seguridad del TRNG.

La huella de área del PLL-TRNG es relativamente pequeña si excluimos los PLL, que no ocupan lógica FPGA. Los PLL en sí no son baratos de implementar, ya que requieren un área de silicio considerable. En las FPGA, por el contrario, los PLL ya están incluidos, por lo que su uso no supone ningún coste. Es más, la mayoría de las familias de FPGAs proporcionan varios PLLs, lo que reduce aún más el coste de la implementación del PLL-TRNG.

El consumo de energía del PLL-TRNG depende de los PLLs proporcionados en la FPGA concreta. Algunas familias tienen todos los PLL activados por defecto, aunque no se utilicen (por ejemplo, las FPGA de Intel). En una familia así, PLL-TRNG no consume mucha más energía que una FPGA vacía, porque en ambos casos los PLL están funcionando. El consumo de energía es considerable en otras familias, que tienen los PLL apagados por defecto. Un PLL-TRNG implementado en una familia de este tipo consumirá mucha más energía que una FPGA vacía.

2.4.7. Comparación entre nucleos TRNG

En la Tabla 2.2 extraída de [2] se muestra un resumen de las cualidades de cada uno de los núcleos TRNG que implementó el autor en diferentes familias de FPGA.

El mensaje más importante que transmite la Tabla 2.2 es que no hay ningún TRNG que destaque en todos los parámetros evaluados ni tampoco hay ningún generador que sea el peor en todos los aspectos. Estos resultados confirman que ningún TRNG puede satisfacer las necesidades de todas las aplicaciones. Muchos parámetros de diseño

Tabla 2.2: Resumen de los resultados núcleos TRNGs [2].

TRNG Type	FPGA device	Area (LUT/Reg)	Power cons. [mW]	Bit Rate [Mbits/s]	Efficiency [bits/ μ Ws]	Entropy per bit	Entropy * Bit rate	Feasib. & Repeat.
ERO	Spartan 6	46/19	2.16	0.0042	1.94	0.999	0.004	5
	Cyclone V	34/20	3.24	0.0027	0.83	0.990	0.003	
	SmartFusion 2	45/19	4	0.014	3.5	0.980	0.013	
COSO	Spartan 6	18/3	1.22	0.54	442.6	0.999	0.539	1
	Cyclone V	13/3	0.9	1.44	1600	0.999	1.438	
	SmartFusion 2	23/3	1.94	0.328	169	0.999	0.327	
MURO	Spartan 6	521/131	54.72	2.57	46.9	0.999	2.567	4
	Cyclone V	525/130	34.93	2.2	62.9	0.999	2.197	
	SmartFusion 2	545/130	66.41	3.62	54.5	0.999	3.616	
PLL	Spartan 6	34/14	10.6	0.44	41.5	0.981	0.431	3
	Cyclone V	24/14	23	0.6	43.4	0.986	0.592	
	SmartFusion 2	30/15	19.7	0.37	18.7	0.921	0.340	
TERO	Spartan 6	39/12	3.312	0.625	188.7	0.999	0.624	1
	Cyclone V	46/12	9.36	1	106.8	0.987	0.985	
	SmartFusion 2	46/12	1.23	1	813	0.999	0.999	
STR	Spartan 6	346/256	65.9	154	2343.2	0.998	154.121	2
	Cyclone V	352/256	49.4	245	4959.1	0.999	244.755	
	SmartFusion 2	350/256	82.52	188	2286.7	0.999	188.522	

dependen unos de otros, lo que crea compensaciones en un diseño y hay que llegar a un compromiso en favor de los parámetros preferidos.

Capítulo 3

Mapas caóticos

En este capítulo se estudiarán los mapas caóticos utilizando como ejemplo el mapa logístico para entender cómo produce el caos y las diferentes técnicas para ver de manera cualitativa este fenómeno.

3.1. Definición de caos

Una manera sencilla de definir el caos sin la necesidad de introducir conceptos muy complicados es como un comportamiento aperiódico, aparentemente impredecible, en sistemas deterministas los cuales presentan extrema sensibilidad a las condiciones iniciales, el más mínimo cambio en la condición inicial produce un resultado muy diferente [48].

Según [49] los sistemas caóticos tienen las siguientes características:

1. Son aperiódicos, es decir nunca se repiten.
2. Presentan una dependencia sensible de las condiciones iniciales (y, por tanto, son imprevisibles a largo plazo).
3. Se rigen por uno o varios parámetros de control, una pequeña modificación de los cuales puede hacer aparecer o desaparecer el caos.
4. Sus ecuaciones son no lineales.

Los mapas caóticos, mapas iterados, ecuaciones de diferencias o simplemente mapas, son sistemas dinámicos en tiempo discreto que tienen la forma general $x_{n+1} = f(x_n)$, los cuales requieren una condición inicial x_0 y se iteran continuamente para conocer su comportamiento. A la secuencia x_0, x_1, x_2, \dots se le conoce como la órbita del mapa comenzando desde x_0 .

3.2. Puntos fijos y estabilidad lineal

Antes de comenzar a estudiar los mapas es necesario desarrollar algunas herramientas que nos sirvan para su análisis. Partimos de la ecuación general de los $x_{n+1} = f(x_n)$, supongamos que x^* satisface que $f(x^*) = x^*$, entonces x^* es un *punto fijo*, si $x_n = x^*$ entonces $x_{n+1} = f(x_n) = x^*$, por lo tanto la órbita permanecerá en x^* para todas las futuras iteraciones.

Para determinar la estabilidad de x^* , consideramos una órbita cercana $x_n = x^* + \eta_n$ y nos preguntamos si la órbita es atraída o repelida desde x^* . Es decir, ¿crece o decrece la desviación η_n a medida que aumenta n ? Utilizando la expansión de Taylor y sustituyendo da como resultado:

$$x^* + \eta_{n+1} = x_{n+1} = f(x^* + \eta_n) = f(x^*) + f'(x^*)\eta_n + O(\eta_n^2) \quad (3.1)$$

Pero como $f(x^*) = x^*$, entonces la ecuación se reduce a:

$$\eta_{n+1} = f'(x^*)\eta_n + O(\eta_n^2) \quad (3.2)$$

Supongamos que podemos despreciar con seguridad los términos $O(\eta_n^2)$. Entonces obtenemos el mapa linealizado $\eta_{n+1} = f'(x^*)\eta_n$ con multiplicador $\lambda = f'(x^*)$. La solución de este mapa linearizado puede hallarse explícitamente escribiendo unos pocos términos: $\eta_1 = \lambda\eta_0$, $\eta_2 = \lambda\eta_1 = \lambda^2\eta_0$ y así en general $\eta_n = \lambda^n\eta_0$. Si $|\lambda| = |f'(x^*)| < 1$, entonces $\eta_n \rightarrow 0$ a medida que $n \rightarrow \infty$ y el punto fijo x^* es linealmente estable. Por el contrario, si $|f'(x^*)| > 1$ el punto fijo es inestable. Aunque estas conclusiones sobre la estabilidad local se basan en la linealización, puede demostrarse que se mantienen para el mapa no lineal original. Pero la linealización no dice nada sobre el caso marginal $|f'(x^*)| = 1$, entonces los términos $O(\eta_n^2)$ despreciados determinan la estabilidad local.

3.3. Mapa logístico

Cuando nos encontramos con un fenómeno complejo como el caos, que se manifiesta en diversas situaciones, es útil adoptar un enfoque que permita identificar y estudiar el sistema más simple que lo ejemplifica. En este sentido, el *mapa logístico* representa el sistema matemático caótico más sencillo, ya que utiliza únicamente una variable y un parámetro de control. Las soluciones exactas de este sistema pueden obtenerse mediante álgebra y su representación gráfica facilita su visualización. Este modelo presenta muchas similitudes con sistemas caóticos más complejos, lo que lo convierte en un candidato perfecto para su estudio. De hecho, ha sido utilizado para modelar fenómenos en diversos campos como la ecología, oncología y finanzas [49].

La ecuación que define al mapa logístico se puede formular partiendo del modelo de

crecimiento exponencial en tiempo discreto que se muestra en la ecuación (3.3), donde A es la tasa de crecimiento.

$$x_{n+1} = Ax_n \quad (3.3)$$

Esta ecuación es un ejemplo de un sistema dinámico determinista, en el que el valor siguiente de x depende únicamente del valor actual. Se trata de un sistema lineal, ya que su representación gráfica muestra una línea recta al graficar x_{n+1} contra x_n . Se trata de una relación recursiva, ya que se aplica de forma repetida a valores sucesivos de n . Además es un ejemplo de mapa iterado, iterar significa retroalimentar la salida de la ecuación a la entrada en el siguiente paso temporal mientras que la iteración es el valor resultante. El término mapa deriva del proceso de transferir cada punto de la Tierra a su correspondiente punto en un mapa impreso. En este caso, estamos mapeando un punto a lo largo del eje x en otro punto a lo largo del mismo eje como se muestra en la Figura 3.1. Sin embargo para mapas más complicados el mapeo se puede extender a más de una dimensión.

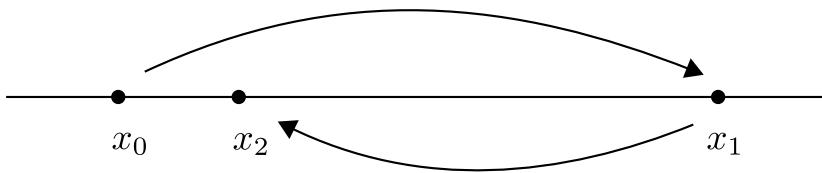


Figura 3.1: Ejemplo de mapeo en una dimensión.

Si el parámetro $A > 1$ el sistema presenta un crecimiento exponencial mientras que si $0 < A < 1$ presenta un decaimiento exponencial. Para toda condición inicial x_0 el sistema es atraído al punto $x = 0$ si $0 < A < 1$ y es atraído hacia el infinito si $A > 1$. Los sistemas cuya solución es atraída hacia el infinito son conocidos como no acotado. El término A es un parámetro de control que rige la naturaleza de la dinámica. El valor $A = 1$ separa dos regiones en las que el comportamiento difiere y se denomina punto de bifurcación. Si $|A| < 1$ el sistema oscila alrededor de $x = 0$ y su amplitud decrece con el tiempo. Por lo tanto $x = 0$ es un atractor para toda x_0 cuando $|A| < 1$. Sin embargo cuando $A < -1$ crece exponencialmente hacia el menos infinito.

Como el crecimiento exponencial no puede continuar para siempre, la ecuación (3.3) es poco realista para cualquier proceso natural. Típicamente, alguna no linealidad tiene e incluso revierte el crecimiento. La no linealidad es despreciable en pequeños valores de x pero empieza a dominar mientras más crece x . Una estrategia comúnmente utilizada es analizar primero el comportamiento lineal antes de introducir no linealidades. A pesar de que el sistema lineal pueda presentar deficiencias notorias, es importante comprender sus propiedades.

Imaginemos un cultivo de bacterias que crece cada hora, con las condiciones ideales

de crecimiento la población aumenta sin restricciones y podemos modelar este comportamiento con la ecuación (3.3). No obstante, si el espacio o el alimento escasea la población de bacterias no puede continuar creciendo a este ritmo. A medida que aumenta la población, el alimento eventualmente se vuelve insuficiente y algunas bacterias mueren antes de poder dividirse.

Para incluir un término que reduzca el crecimiento a medida que x aumenta es necesario modificar la ecuación (3.3). La manera más sencilla es considerar que la tasa de crecimiento decrece linealmente, de manera que a la ecuación (3.4) se le conoce como ecuación logística.

$$x_{n+1} = Ax_n(1 - x_n) \quad (3.4)$$

La no linealidad es cuadrática, debido a que el modelo se puede reescribir como $x_{n+1} = Ax_n - Ax_n^2$. El término cuadrático da una retroalimentación negativa no lineal y limita el crecimiento. La Figura 3.2 muestra la gráfica de la ecuación (3.4) con $A = 4$ llamada función logística o curva logística, la cual es una parábola.

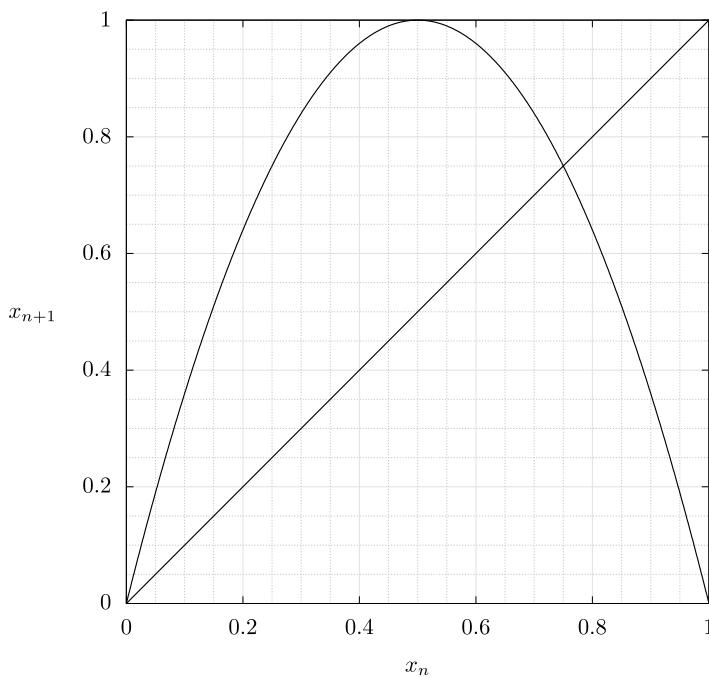


Figura 3.2: Gráfica de mapa logístico con $A = 4$.

En la Figura 3.1 también se muestra una recta descrita por $x_{n+1} = x_n$, a 45° cuyas intersecciones con la parábola dan los valores de x que no cambian con el tiempo. Para un mapa cuadrático, hay dos intersecciones de este tipo, que corresponden a las soluciones de la ecuación cuadrática que resultan de establecer $x_{n+1} = x_n = x^*$. Las soluciones $x^* = 0$ y $x^* = 1 - 1/A$ son llamados puntos fijos del mapa.

3.3.1. Diagrama de cobwebs

Es interesante examinar cómo x se aproxima a un punto fijo partiendo de una condición inicial $x_0 \neq x^*$, esto lo hacemos con un diagrama de cobwebs. La Figura 3.3 muestra el diagrama de cobwebs del mapa logístico para $A = 2.8$. Los diagramas de cobwebs son una herramienta valiosa que nos permiten observar el comportamiento global de un sistema de manera intuitiva, proporcionando información complementaria a la obtenida mediante el análisis lineal. Son especialmente útiles resultan cuando el análisis lineal no es suficiente.

Para construir el diagrama de cobwebs de un mapa iterado realizamos los siguientes pasos:

Dado $x_{n+1} = f(x_n)$ y una condición inicial x_0 , trazar una línea vertical hasta que intersecte la gráfica f , esa altura es la salida x_1 , en otras palabras dibujar una línea vertical desde $(x_0, 0)$ hasta (x_0, x_1) . Después trazar una horizontal hasta intersectar con la linea diagonal $x_{n+1} = x_n$, es decir, dibujar una línea desde (x_0, x_1) hasta (x_1, x_1) . Después trazar otra línea vertical hasta que intersecte la gráfica f otra vez, una línea desde (x_1, x_1) hasta (x_1, x_2) . Repetir este proceso n veces para generar los primeros n puntos en la órbita.

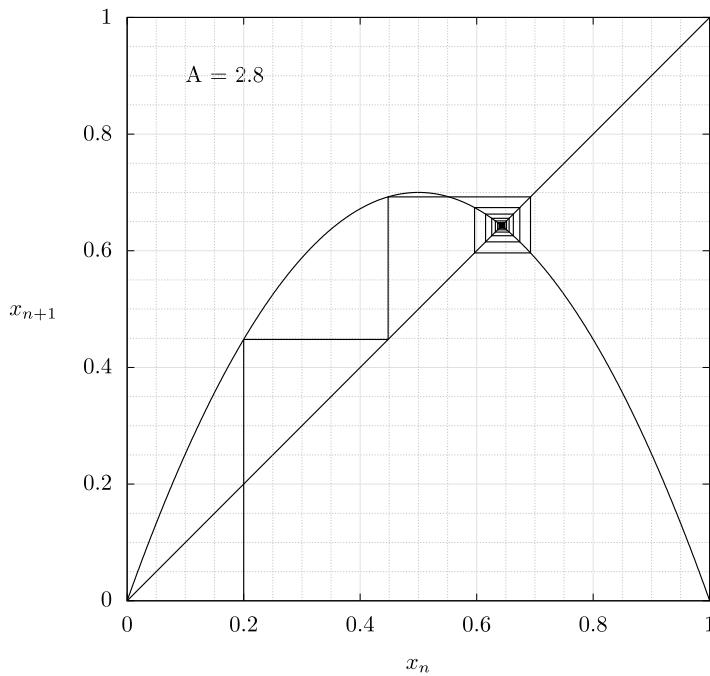


Figura 3.3: Diagrama de cobwebs de mapa logístico con $A = 2.8$ y $x_0 = 0.2$.

Para el mapa logístico con $1 < A < 3$, todos los puntos iniciales en el intervalo $0 < x_0 < 1$ se aproximan al punto fijo $x^* = 1 - 1/A$, aunque la solución puede oscilar a su alrededor antes de llegar al valor final. El comportamiento recuerda al de un péndulo simple que oscila alrededor de la vertical antes de que la fricción lo lleve a reposar en su posición final. Otra manera de visualizar este comportamiento es graficar la serie de

tiempo, x_n contra n , como se ve en la Figura 3.4. Es importante hacer la aclaración que se dibujaron líneas entre cada iteración para una mejor visualización, pero los datos son discretos.

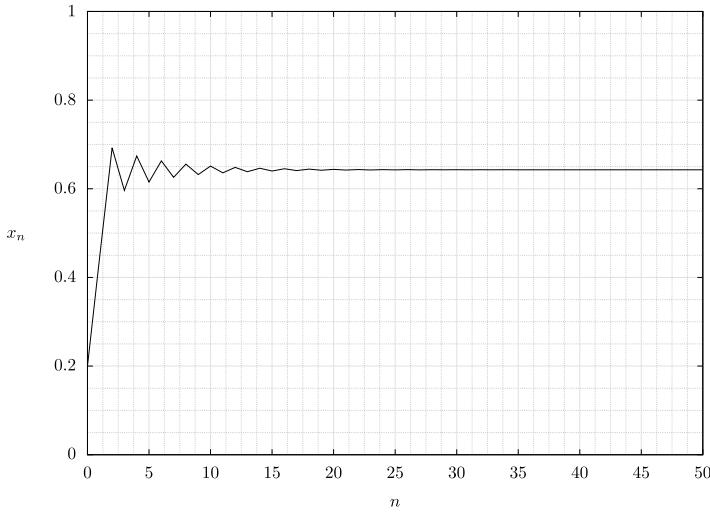


Figura 3.4: Serie de tiempo de mapa logístico con $A = 2.8$.

3.3.2. Análisis cualitativo del mapa logístico

En la ecuación de crecimiento exponencial en tiempo discreto (3.3) cuando $A = \pm 1$ el comportamiento cambia abruptamente de acotado a no acotado. La ecuación logística se comporta de manera similar, excepto que hay más bifurcaciones y el comportamiento en varias regiones es más diverso e interesante. Es útil estudiar las bifurcaciones del mapa logístico debido a que las características generales son comunes a muchos sistemas caóticos. Consideraremos sólo valores positivos de A y x .

- Caso $0 < A < 1$

En este rango la parábola solo puede intersectar a la linea de 45° una vez en valores no negativos, y por lo tanto solo hay un punto fijo en $x^* = 0$. Todas las condiciones iniciales en el rango $0 < x_0 < 1$ son atraídos hacia el punto $x^* = 0$. Decimos que estos puntos se encuentran dentro de una cuenca de atracción de x_0 y que el punto fijo x_0 es estable. Todos los puntos dentro de la cuenca de atracción se acercan al punto fijo con cada iteración. La no linealidad tiene poco efecto después de las primeras iteraciones. Los valores de x_0 fuera de la cuenca de atracción están no acotados, escapan al infinito.

- Caso $1 < A < 3$

Al igual que con la ecuación (3.3), para $A = 1$ se produce una bifurcación y el punto fijo en $x^* = 0$ se vuelve inestable. El atractor se convierte en un repulsor.

Si x resulta ser exactamente cero, permanecerá así, pero si es incluso ligeramente positivo, crecerá inicialmente a un ritmo exponencial. La situación es como la de un lápiz apoyado sobre su extremo puntiagudo. El más mínimo empujón hará que se caiga. Sin embargo, a diferencia de la ecuación (3.3) en la que las soluciones son ilimitadas, el mapa logístico en $A = 1$ desarrolla un nuevo punto fijo en $x^* = 1 - 1/A = 0$ que se aleja de cero para $A > 1$. Si A no es demasiado grande, entonces ese punto es un atractor porque todos los valores iniciales en el rango $0 < x_0 < 1$ son atraídos hacia él y finalmente se asientan en él. Decimos que el estado final es un ciclo de periodo 1, o simplemente un ciclo de 1, porque cada iteración es la misma que la anterior. Si la ecuación logística estuviera modelando la población de bacterias, entonces predeciría un crecimiento exponencial inicial para este rango de A , pero un estado estacionario final en el que el número de bacterias no cambia.

- Caso $3 < A < 3.44948 \dots$

Para $A = 3$, el punto fijo en $x^* = 1 - 1/A$ sigue existiendo, pero cambia de estable a inestable, convirtiéndose en un repulsor. Esta bifurcación se produce cuando la pendiente de la parábola en el punto fijo es igual a -1 . Para $A > 3$ tenemos un crecimiento exponencial alejándonos del punto, en lugar de una caída exponencial hacia él. Como la pendiente es negativa, la solución oscila a ambos lados del punto fijo mientras se aleja, igual que en la ecuación (3.3) con $A < -1$. De ahí que la bifurcación en $A = 3$ se denomine flip. Sin embargo, el crecimiento no continúa para siempre. En su lugar, se acerca a una condición en la que cada iteración es la misma $x_n = x_{n+2} = x_{n+4}$ como se muestra en el diagrama de cobwebs de la Figura 3.5.

Con un poco de álgebra, esta condición puede reducirse a una ecuación de cuarto grado

$$A^3x^4 - 2A^3x^3 + A^2(A+1)x^2 - (A^2 - 1)x = 0 \quad (3.5)$$

Como cualquier ecuación de cuarto orden, tiene cuatro raíces. Una es $x^* = 0$, otra es $x^* = 1 - 1/A$. La factorización de estos términos reduce la ecuación (3.5) a una ecuación cuadrática

$$A^2x^2 - A(A+1)x + A+1 = 0 \quad (3.6)$$

cuyas raíces son

$$x_{\pm}^* = \frac{A+1 \pm \sqrt{(A-3)(A+1)}}{2A} \quad (3.7)$$

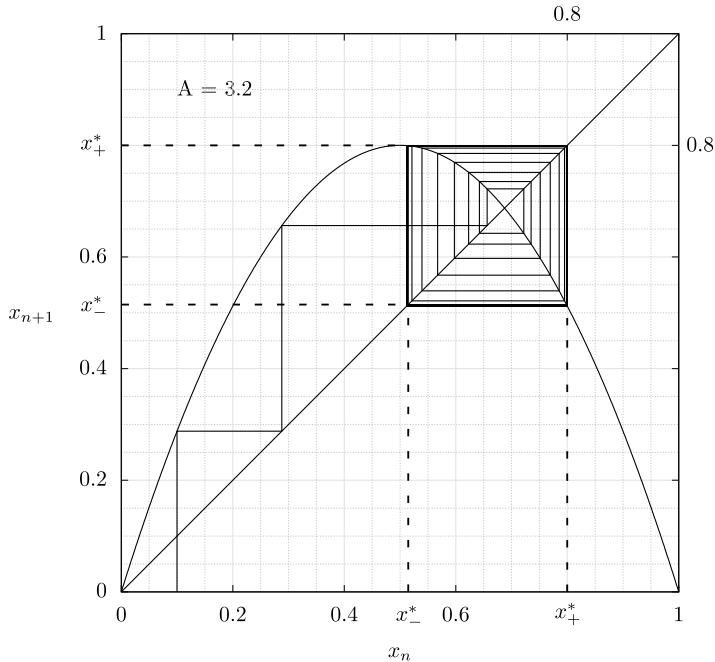


Figura 3.5: Diagrama de cobwebs de mapa logístico con $A = 3.2$ y $x_0 = 0.1$.

Para $-1 < A < 3$, la cantidad dentro de la raíz cuadrada es negativa, y no existe una solución real. Para $A > 3$ hay dos raíces reales entre las cuales x oscila en iteraciones sucesivas en el estado estacionario. Este es un ejemplo de un ciclo de 2. Se trata de un atractor cíclico o periódico ya que casi cualquier condición inicial en el intervalo unidad se aproxima a él. En tal caso, las bacterias abundarían una hora, escasearían a la siguiente y volverían a abundar. Obsérvese que para $A = 3$, la ecuación (3.7) tiene una única raíz en $x^* = 2/3$, que es lo mismo que $x^* = 1 - 1/A$, lo que significa que el ciclo de 2 bifurca continuamente.

- Caso $3.44948\dots < A < 3.56994\dots$

El ciclo de 2 existe para todo $A > 3$, pero se vuelve inestable cuando A alcanza un valor en el que la pendiente del mapa de segunda iteración evaluado en $x = x^*$ según la ecuación (3.5) es igual a -1 . El cálculo conduce a una ecuación cuadrática

$$A^2 - 2A - 5 = 0 \quad (3.8)$$

cuya raíz positiva es $A = 1 + \sqrt{6} = 3.449490\dots$. En esta bifurcación, el ciclo de 2 se vuelve inestable, y nace un ciclo de 4 estable. El mapa logístico puede tener como máximo una órbita periódica estable para cada valor de A . Sin embargo, esta propiedad no la comparten todos los mapas unimodales unidimensionales.

El proceso continúa con sucesivas duplicaciones de periodos (en cascada), apareciendo un nuevo periodo justo cuando el anterior se vuelve inestable. El inicio de estos desdoblamientos es cada vez más difícil de calcular, tanto analíticamente

como numéricamente. Los próximos desdoblamientos se muestran en la Tabla 3.1.

Tabla 3.1: Diferentes número de ciclos para diferentes valores de A .

Valores de A	Número de ciclos
$A_1 = 3.000000\dots$	2
$A_2 = 3.449490\dots$	4
$A_3 = 3.544090\dots$	8
$A_4 = 3.564407\dots$	16
$A_5 = 3.568759\dots$	32
$A_6 = 3.569692\dots$	64
$A_7 = 3.569891\dots$	128
$A_8 = 3.569934\dots$	256
$A_9 = 3.569943\dots$	521
$A_{10} = 3.569945\dots$	1024
.....
$A_\infty = 3.5699456718\dots$	punto de acumulación

Las duplicaciones del periodo se acercan sucesivamente y acaban acumulándose en el punto $A_\infty = 3.5699456718\dots$ conocido como punto de acumulación. En este punto, el periodo se hace infinito (nunca se repite), y la órbita visita infinitos valores de x pero llena una porción insignificante del intervalo interior unitario ($0 < x < 1$). El espacio entre bifurcaciones sucesivas se aproxima a una constante

$$\delta = \lim_{n \rightarrow \infty} \frac{A_n - A_{n-1}}{A_{n+1} - A_n} = 4.669201\dots \quad (3.9)$$

conocida como el número de Feigenbaum. Las bifurcaciones vienen dadas aproximadamente por $A_k \approx A_\infty - 1.542\delta^{-k}$. Una propiedad importante de la constante δ es su universalidad, ya que tiene el mismo valor para todos los mapas unimodales con un máximo cuadrático. Se trata de una nueva constante matemática, tan básica para la duplicación de periodos π para los círculos. La duplicación del periodo con aparentemente la misma constante se ha observado en muchos experimentos, incluyendo fluidos turbulentos, circuitos electrónicos, láseres, reacciones químicas, grifos que gotean, instrumentos musicales, e incluso sistemas biológicos que no se describen obviamente mediante mapas unidimensionales, como los latidos del corazón.

- Caso $3.56994 < A < 4$

Cuando A aumenta más allá del punto de acumulación, se desencadena el caos. El periodo es infinitamente largo y regiones finitas del intervalo unitario son visitadas por la órbita. Sin embargo, existen infinitas ventanas (rangos de A) de periodicidad.

dad. Todos los períodos están representados, pero la anchura de la ventana disminuye a medida que aumenta el período. Cada ventana de periodicidad aparece abruptamente a medida que A aumenta y contiene su propia ruta de duplicación de períodos de vuelta al caos con el mismo número de Feigenbaum. Por ejemplo, la ventana prominente de período 3 comienza en $A = 1+ = 3.82842712\dots$, aunque la demostración es difícil. La órbita de período 3 es especial, porque Sarkovskii y más tarde Li y Yorke (1975) demostraron independientemente que un mapa continuo unidimensional con una órbita de período 3 para un valor de parámetro particular tiene órbitas de todos los períodos (incluido el infinito) para ese parámetro y, por lo tanto, será caótico si todas esas órbitas son inestables. Lo contrario no es cierto, los sistemas caóticos no necesitan tener una órbita de período 3. Cada período superior a 3 ocurre más de una vez. Por ejemplo, hay dos ventanas de período 4, tres ventanas de período 5, cinco ventanas de período 6, etc. Todo el comportamiento puede resumirse en un diagrama de bifurcación como el de la Figura 3.6. Este diagrama representa todos los valores posibles de x en el estado final después de que los transitorios iniciales hayan desaparecido en función del parámetro de control A . Los puntos de bifurcación, son evidentes.

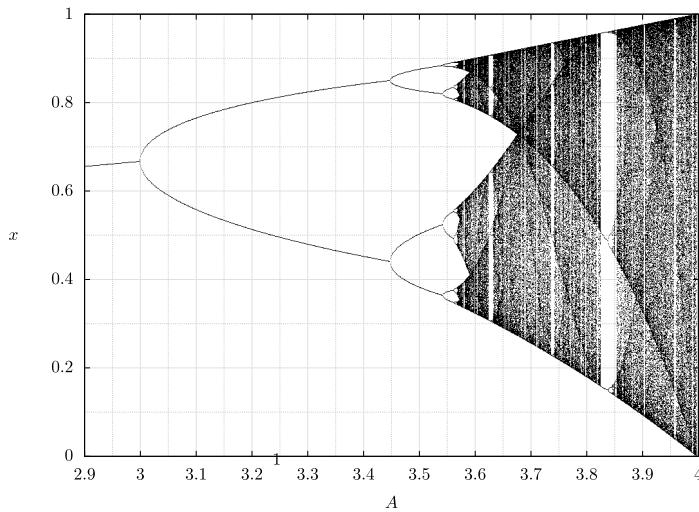


Figura 3.6: Diagrama de bifurcación de mapa logístico [50].

- Caso $A = 4$

El caso $A = 4$ es especial porque vuelve a mapear el intervalo unitario sobre sí mismo. Un mapa con esta propiedad se llama endomorfismo. Se puede imaginar el eje x entre cero y uno como una banda elástica, que con cada iteración se estira de modo que su punto medio, $x_n = 0.5$, alcanza $x_{n+1} = 1$ y luego el extremo lejano $x_n = 1$ se pliega de vuelta hacia $x_{n+1} = 0$. El estiramiento y plegamiento son responsables del caos. Dos condiciones iniciales cercanas se separan debido al estiramiento, mientras que el plegamiento las mantiene acotadas. El estiramiento

no es uniforme, sin embargo, debido a que es largo (un factor de cuatro por iteración) en $x = 0$ y $x = 1$ pero infinitamente negativo en $x = 0.5$. En promedio el estiramiento es un factor de dos, como sugiere el hecho que se mapea de sobre sí mismo dos veces.

La iteración x_n tiene dos preimágenes x_{n-1} dadas por

$$x_{n-1} = 0.5 \pm \sqrt{0.25 - x_n/A} \quad (3.10)$$

que no suelen coincidir. En consecuencia, en cada iteración se pierde un bit de información (un factor de 2) ya que no hay forma de saber de qué preimagen procede cada valor (decimos que el mapa es no invertible, siendo en este caso de dos a uno), en contraste con un mapa invertible (de uno a uno) en el que existe una preimagen única. Esta pérdida exponencial de información equivale al crecimiento exponencial de los errores en la condición inicial que caracteriza al caos. La no invertibilidad es necesaria para el caos en los mapas unidimensionales, pero no para los mapas en dimensiones superiores. El caos se puede mostrar en un diagrama de cobwebs, como en la Figura 3.7, o directamente en un gráfico de iteraciones sucesivas o serie de tiempo, como en la Figura 3.8.

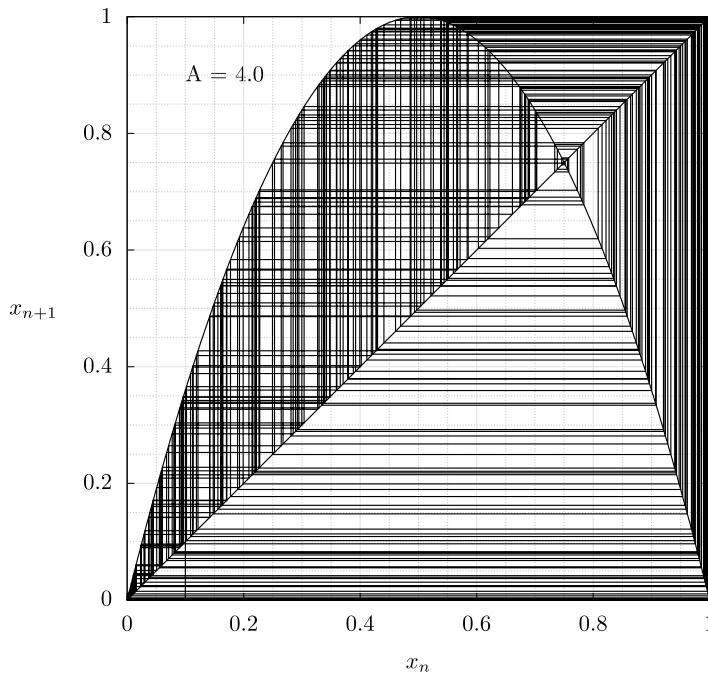


Figura 3.7: Diagrama de cobwebs de mapa logístico con $A = 4.0$ y $x_0 = 0.1$.

Aunque el comportamiento parece aleatorio, un examen más detallado revela el crecimiento exponencial en valores pequeños de x_n , valores pequeños de x_n que siguen a uno grande y una oscilación creciente alrededor del punto fijo inestable en $x_n = 1 - 1/A = 0.75$. Es sorprendente que una simple ecuación cuadrática pueda exhibir un comportamiento tan complejo. Si la ecuación logística con $A = 4$

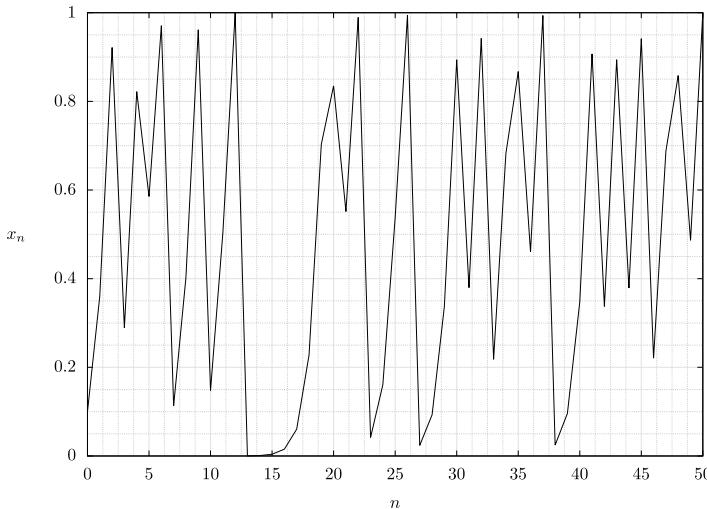


Figura 3.8: Serie de tiempo de mapa logístico con $A = 4.0$.

modelara el crecimiento de bacterias, entonces su población exhibiría fluctuaciones erráticas. El mapa logístico con $A = 4$, a veces llamado mapa de Ulam y escrito en la forma equivalente $x_{n+1} = 1 - 2x_n^2$, fue estudiado por Ulam mucho antes de la era del caos moderno. Ulam y von Neumann lo propusieron como un generador de números aleatorios por computadora en 1947.

El mapa logístico con $A = 4$ es especial en muchos sentidos. Es totalmente caótico en el sentido de que casi todos los puntos del intervalo unitario son eventualmente visitados por cualquier condición inicial, una propiedad conocida como transitividad topológica. Este hecho y la simplicidad de la ecuación nos permite calcular varias propiedades especiales del mapa logístico con $A = 4$.

- Caso $A > 4$

Para $A > 4$, el pico de la parábola es superior a uno. Por lo tanto, la mayoría de las condiciones iniciales tienen iteraciones que eventualmente alcanzan $x_n > 1$. Cuando esto ocurre, la siguiente iteración es negativa, y el repulsor en $x = 0$ empuja la órbita rápidamente a menos infinito. Así, la mayoría de las órbitas son ilimitadas para $A > 4$. Sin embargo, el punto fijo inestable en $x^* = 1 - 1/A$ persiste para $A > 4$, al igual que todas las órbitas periódicas inestables, como el ciclo 2 dado por la ecuación (3.7). La órbita de periodo 2 oscila entre un valor ligeramente superior a cero y un valor incluso ligeramente inferior a uno.

3.3.3. Análisis teórico del mapa logístico

Todo el análisis anterior se realizó considerando que el lector es capaz de realizar con facilidad todos los cálculos y desarrollos teóricos, no obstante en esta sección se muestra con más detalle como realizar el desarrollo teórico.

Consideremos la ecuación $x_{n+1} = Ax_n(1 - x_n)$ para $0 \leq x_n \leq 1$ y $0 \leq A \leq 4$. Los puntos fijos satisfacen $x^* = f(x^*) = Ax^*(1 - x^*)$. Por lo tanto $x^* = 0$ o $x^* = 1 - 1/A$. El origen es un punto fijo para todas las A , mientras que $x^* = 1 - 1/A$ solo es válido para las $x \geq 1$. La estabilidad depende de $f'(x^*) = A - 2Ax^*$. Como $f'(0) = A$ el origen es estable para $A < 1$ e inestable para $A > 1$. En el otro punto fijo, $f'(x^*) = A - 2A(1 - 1/A) = 2 - A$. Entonces $x^* = 1 - 1/A$ es estable para $1 < A < 3$ e inestable para $A > 3$.

En la Figura 3.9 se muestra un análisis gráfico de cómo se comporta el mapa logístico para distintos valores de A . Para $A < 1$ la parábola está por debajo de la diagonal, y el origen es el único punto fijo. A medida que A aumenta, la parábola se hace más alta, haciéndose tangente a la diagonal en $A = 1$. Para $A > 1$ la parábola intersecta la diagonal en un segundo punto $x^* = 1 - 1/A$, mientras que el origen pierde estabilidad. Así vemos como x^* se bifurca desde el origen en una bifurcación transcíptica en $A = 1$.

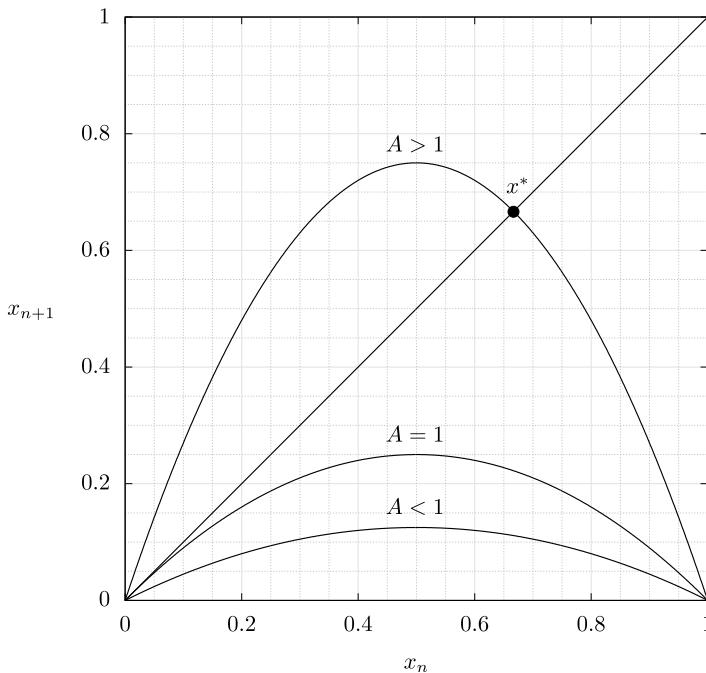


Figura 3.9: Gráfica de mapa logístico para distintos valores de A .

Cuanto más crece A , más allá de 1, la pendiente en x^* se vuelve cada vez más empinada, y x^* pierde estabilidad. La pendiente crítica ocurre cuando $f'(x^*) = -1$, la cual se obtiene cuando $A = 3$. La bifurcación resultante se denomina bifurcación flip.

Las bifurcaciones flip se asocian a menudo con la duplicación del periodo. En el mapa logístico, la bifurcación en $A = 3$ da lugar a un ciclo 2. Un ciclo 2 existe si y sólo si hay dos puntos p y q tales que $f(p) = q$ y $f(q) = p$. Equivalentemente, p debe satisfacer que $f(f(p)) = p$, donde $f(x) = Ax(1 - x)$. Por tanto, p es un punto fijo del mapa de segunda iteración. Como $f(x)$ es un polinomio de segundo grado, $f(f(x))$ es un polinomio de cuarto grado.

Para encontrar p y q , necesitamos resolver los puntos en los que la gráfica intersecta la diagonal, es decir, necesitamos resolver la ecuación de cuarto grado $f(f(x)) = x$. Los puntos $x^* = 0$ y $x^* = 1 - 1/A$ son soluciones triviales de la ecuación. Si factorizamos estos puntos fijos, el problema se reduce a resolver una ecuación cuadrática.

La expansión de la ecuación $f(f(x)) - x = 0$ da $A^2x(1-x)[1-Ax(1-x)] - x = 0$. Después de factorizar x y $x - (1 - 1/A)$ por división larga, y resolver la ecuación cuadrática resultante, obtenemos un par de raíces:

$$p, q = \frac{A + 1 \pm \sqrt{(A - 3)(A + 1)}}{2A} \quad (3.11)$$

que son reales para $A > 3$. Por lo tanto, existe un ciclo doble para todo $A > 3$. En $A = 3$, las raíces coinciden y son iguales a $x^* = 1 - 1/A = 2/3$, lo que demuestra que el ciclo 2 bifurca continuamente a partir de x^* . Para $A < 3$ las raíces son complejas, lo que significa que no existe un ciclo 2.

Para realizar el análisis de estabilidad es necesario seguir la siguiente estrategia. Para analizar la estabilidad de un ciclo, se reduce el problema a una pregunta sobre la estabilidad de un punto fijo de la siguiente manera. Tanto p y q son soluciones de $f(f(x)) = x$, por lo tanto p y q son puntos fijos del mapa de segunda iteración $f(f(x))$. El ciclo 2 original es estable precisamente si p y q son puntos fijos estables para $f(f(x))$.

Ahora que el problema se reduce a analizar la estabilidad de los puntos fijos de otro mapa. Para determinar si p es un punto fijo estable de $f(f(x))$, calculamos el multiplicador:

$$\lambda = \frac{d}{dx}(f(f(x)))_{x=p} = f'(f(p))f'(p) = f'(q)f'(p) \quad (3.12)$$

Se obtiene la misma λ para $x = q$ por simetría. Por lo tanto, cuando las ramas p y q se bifurcan, deben hacerlo simultáneamente. Después de realizar las derivadas y realizar las evaluaciones p y q , obtenemos:

$$\lambda = A(1 - 2q)A(1 - 2p) \quad (3.13)$$

$$= A^2[1 - 2(p + q) + 4pq] \quad (3.14)$$

$$= A^2[1 - 2(A + 1)/A + 4(A + 1)/A^2] \quad (3.15)$$

$$= 4 + 2A - A^2 \quad (3.16)$$

Por lo tanto el ciclo 2 es linealmente estable para $|4 + 2A - A^2| < 1$ entonces $3 < A < 1 + \sqrt{6}$. Los métodos analíticos para las próximas bifurcaciones se vuelven poco manejables. Se pueden obtener otros resultados exactos pero son difíciles de conseguir.

Capítulo 4

Implementación de TRNG híbrido

En este capítulo se describen los pasos para diseñar un mapa caótico y un ERO-TRNG en FPGA para generar bits aleatorios que servirán como semilla a las condiciones del mapa. Se analiza cómo seleccionar el tamaño de palabra de punto fijo utilizando un simulador en C, cómo mejorar el uso de recursos con multiplicadores de una constante y cómo seleccionar el rango valido de la semilla utilizando el dominio de atracción del mapa caótico.

4.1. Aritmética de punto fijo

Cuando se necesita trabajar con números de punto decimal en dispositivos digitales se puede optar por utilizar aritmética de punto fijo o aritmética de punto flotante. La aritmética de punto fijo en comparación de la de punto flotante tiene la ventaja de ser más rápida y utilizar menos recursos, sin embargo, tiene la desventaja de tener un rango de uso específico el cual se tienen que establecer al principio del diseño y el cual no puede modificarse posteriormente. Por el contrario la aritmética de punto flotante tienen un mayor rango dinámico y resulta útil cuando los algoritmos tienen una complejidad alta.

En el proceso de diseño de arquitecturas digitales de aplicación específica se busca utilizar la menor cantidad de recursos y reducir al máximo el tiempo de ejecución del algoritmo subyacente. Cuando se trabaja con los FPGA se cuenta con la libertad de poder analizar y estudiar el algoritmo antes de comenzar con el proceso de diseño. Por todo lo anterior, es preferible utilizar operaciones en punto fijo en lugar de punto flotante en los FPGA [51].

La representación de punto fijo de un número X es $X(a, b)$ donde a es la parte entera y b es la parte fraccionaria. De manera que el número de bits de la representación es $a + b + 1$, es decir las suma de la parte entera, la parte fraccionaria y el bit de signo. El rango de valores que se puede representar es $[-2^a, 2^a - 2^{-b}]$ y por lo tanto el número

más pequeño que puede representar o visto de otro modo la precisión es de 2^{-b} . En la Tabla 4.1 se muestran ejemplos de diferentes formas de interpretar números binarios utilizando 5 bits y diferentes formatos de punto fijo.

Cuando se hace la suma de dos números de punto fijo $X(a, b)$, ambos números tienen un rango de $[-2^a, 2^a - 2^{-b}]$. El mayor número que se puede obtener sumando dos extremos, los dos más positivos o los dos más negativos. Sumando los dos más negativos se obtiene:

$$-2^a + (-2^a) = 2(-2^a) = -2^{a+1} \quad (4.1)$$

sumando los dos más positivos:

$$(2^a - 2^{-b}) + (2^a - 2^{-b}) = 2(2^a - 2^{-b}) = 2^{a+1} - 2^{1-b} \quad (4.2)$$

y como $|-2^{a+1}| > |2^{a+1} - 2^{1-b}|$. Entonces hay que representar el mayor número negativo número -2^{a+1} .

Entonces el resultado de la suma de dos números de punto fijo $X(a, b)$ es un número $X(a + 1, b)$. Por ejemplo, si sumamos dos números con el formato de punto fijo $X(3, 0)$, el mayor número se genera sumando $-8 - 8 = -16$. Este resultado necesita una representación $X(4, 0)$, que tiene un rango de $[-16, 15]$.

Tabla 4.1: Ejemplos de formato, rango y conversión de números de 5 bits en formato de punto fijo.

Número	Conversión	Formato $X(a, b)$	Rango $[-2^a, 2^a - 2^{-b}]$
01110	3.5	$X(2, 2)$	$[-4, 3.75]$
10010	-3.5
00011	0.75
01110	1.75	$X(1, 3)$	$[-2, 1.875]$
10010	-1.75
00011	0.375
01110	7.0	$X(3, 1)$	$[-8, 7.5]$
10010	-7.0
00011	1.5

Para una multiplicación de dos números $X(a, b)$, analizamos cuantos bits se requieren para almacenar el resultado. Los números más grandes que se generan son el resultado más positivo generado al multiplicar ambos números negativos, y el más negativo, resultado de multiplicar el más positivo y el más negativo. Para los dos más negativos tenemos:

$$(-2^a)(-2^a) = 2^{2a} \quad (4.3)$$

para el más positivo y el más negativo tenemos:

$$(-2^a)(2^a - 2^{-b}) = -2^{2a} + 2^{a-b} \quad (4.4)$$

y a multiplicación de los dos más positivos es:

$$(2^a - 2^{-b}) = 2^{2a} - 2^{-2b} \quad (4.5)$$

El número positivo más grande que es necesario representar es 2^{2a} . También es necesario representar -2^{-2b} , entonces se necesita el número $X(2a + 1, 2b)$ para representar la multiplicación de dos números $X(a, b)$. $X(2a + 1, 2b)$ tiene un rango de $[-2^{a+1}, 2^{2a+1} - 2^{-2b}]$.

4.2. Mapa caótico

En el artículo [52] se analiza y estudia el mapa caótico cuadrático bidimensional más sencillo que esta dado por la ecuación (4.6).

$$\begin{aligned} x_{n+1} &= a_1 + a_2 x_n + a_3 x_n^2 + a_4 x_n y_n + a_5 y_n + a_6 y_n^2 \\ y_{n+1} &= a_7 + a_8 x_n + a_9 x_n^2 + a_{10} x_n y_n + a_{11} y_n + a_{12} y_n^2 \end{aligned} \quad (4.6)$$

donde los parámetros $\{a_1, a_2, \dots, a_{12}\}$ y las condiciones iniciales x_0 y y_0 determinan las características de la solución. La iteraciones se representan como puntos en una superficie bidimensional. Después de un cierto número de iteraciones, la solución hará una de estas cuatro cosas: (a) convergerá a un único punto fijo; (b) tomará una sucesión de valores que acabarán repitiéndose, produciendo un ciclo límite; (c) será inestable y divergirá hasta el infinito; (d) mostrará caos y llenará gradualmente alguna región a menudo complicada pero acotada del plano $x - y$.

Para saber qué valores de $\{a_1, a_2, \dots, a_{12}\}$ llevan al caos, [52] utilizó el siguiente procedimiento.

1. Elegir aleatoriamente los 12 coeficientes de a_1 hasta a_{12} aleatoriamente sobre algún intervalo.
2. Elegir las condiciones iniciales x_0 y y_0 .
3. Iterar las ecuaciones del mapa mientras se calcula el exponente de Lyapunov y se comprueba si no hay divergencia.
4. Mantener las soluciones que están acotadas y tienen un exponente de Lyapunov positivo.

Los coeficientes los eligió en incrementos de 0.1 en el intervalo de -1.2 a 1.2, es decir, 25 valores posibles. Coeficientes más pequeños hacen que se pierdan muchas

soluciones caóticas y más grandes producen sobre todo soluciones inestables. El incremento se eligió para que cada atractor sea visiblemente diferente y los coeficientes puedan codificarse en letras del alfabeto desde la A hasta la Y . De manera que $A = -1.2, B = -1.1, C = -1.0, \dots, Y = 1.2$. La Tabla 4.2 puede utilizarse para realizar la conversión de esta representación de manera rápida. Esta representación hace que sea fácil su replicación. Así, cada atractor se identifica unívocamente con un nombre de 12 letras. El número de posibles casos es 25^{12} o aproximadamente 6×10^{12} . De estos, aproximadamente 1.6% son caóticos. Verlos todos a un ritmo de uno por segundo requeriría más de 30 millones de años. Por tanto, es muy poco probable que los patrones producidos se hayan visto antes y, al igual que los copos de nieve, casi todos son diferentes.

Tabla 4.2: Conversiones para codificación de los atractores.

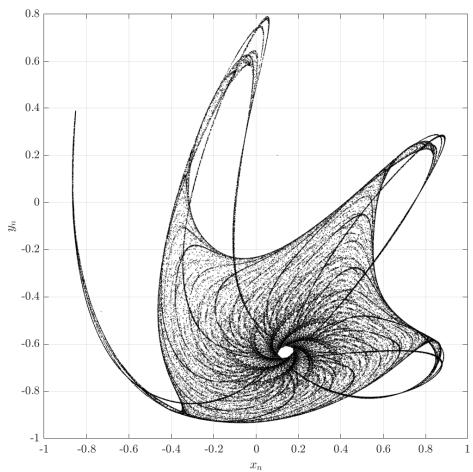
Letra	Codificación	Letra	Codificación
A	-1.2	N	0.1
B	-1.1	O	0.2
C	-1.0	P	0.3
D	-0.9	Q	0.4
E	-0.8	R	0.5
F	-0.7	S	0.6
G	-0.6	T	0.7
H	-0.5	U	0.8
I	-0.4	V	0.9
J	-0.3	W	1.0
K	-0.2	X	1.1
L	-0.1	Y	1.2
M	-0.0		

Algunos de los atractores más llamativos se muestran en la Figura 4.1 y se pueden obtener decodificando la palabra de 12 letras de la Tabla 4.3 del atractor deseado. En el artículo [52] no se tienen en cuenta las posibles variaciones que pueden ocurrir al modificar las condiciones iniciales. Estas se eligieron arbitrariamente a $x_0 = y_0 = 0.05$, además, no se especifica el rango que pueden tener las condiciones iniciales en el que se asegure que exista el caos.

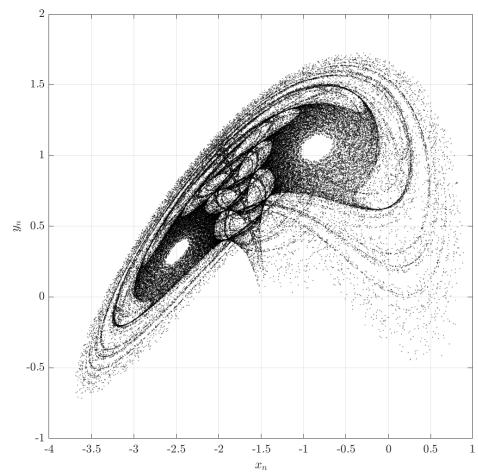
Para resolver este problema se puede analizar el dominio de atracción [53] que tienen cada uno de los atractores. Podemos calcularlo numéricamente siguiendo los siguientes pasos:

1. Elegir un rango para $x_0 \in [x_{izq}, x_{der}]$ y $y_0 \in [y_{izq}, y_{der}]$ y un tamaño de paso h donde se va a realizar el análisis.
2. Iterar el mapa unos cientos de veces para cada uno de los posibles valores de x_0 y

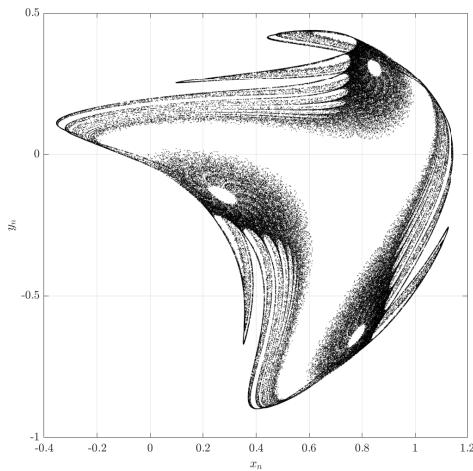
Figura 4.1: Diferentes atractores caóticos del mapa bidimensional generados en aritmética de punto flotante.



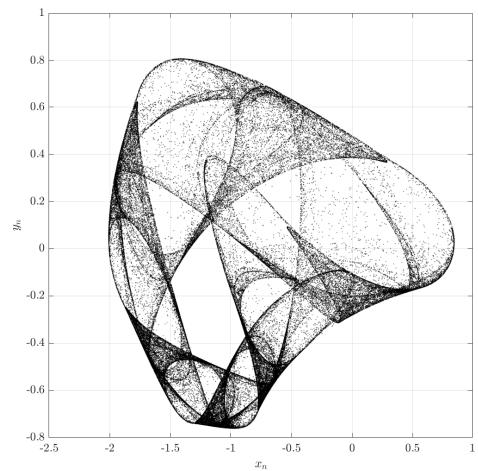
(a) Atractor 1



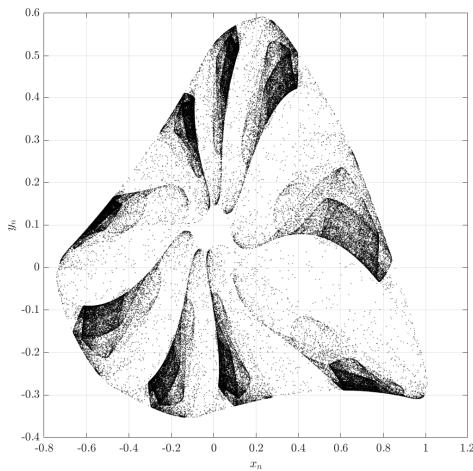
(b) Atractor 2



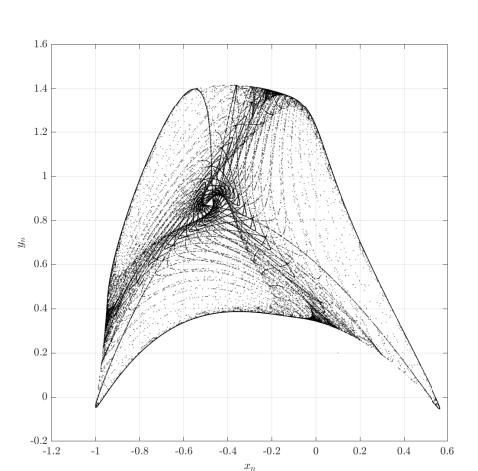
(c) Atractor 3



(d) Atractor 4



(e) Atractor 5



(f) Atractor 6

Tabla 4.3: Diversos identificadores, dimensión fractal y exponente de Lyapunov para atractores del mapa caótico bidimensional.

Atractor	Identificador	Dimensión fractal	Exponente de Lyapunov
1	GLXOESFTPSV	1.77	0.12
2	CVQKGHQTPHTE	1.79	0.14
3	UWACXDQIGKHF	1.42	0.10
4	GIIETPIQRRL	1.50	0.13
5	MCRBIPOPHTBN	1.39	0.05
6	ODGQCNXODNYA	1.31	0.07

y_0 dentro del rango y el tamaño de paso h seleccionado. Para cada combinación almacenar todas las iteraciones en un vector.

3. Comprobar cada vector en busca de puntos fijos, divergencia y de ser posible ciclos límite.
4. En una matriz de tamaño, $m \times n$, donde m es el número de elementos en el rango de y_0 y n el número de elementos del rango de x_0 , escribir 1 si esta dentro del rango que produce caos o 0 si esta fuera.

En la Figura 4.2 se muestran los dominios de atracción para cada uno de los mapas de la Tabla 4.3 siguiendo el algoritmo anterior.

Si analizamos los dominios de atracción de la Figura 4.2 podemos seleccionar un rango en el que sin importar cual sea la condición inicial existirá caos. Por comodidad seleccionaremos una ventana de 1 unidad tanto para x_0 como para y_0 como se muestran en la Tabla 4.4.

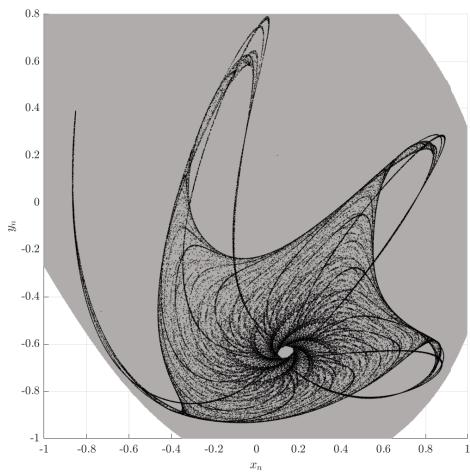
Tabla 4.4: Rangos usados para la condición inicial para cada uno de los atractores.

Atractor	Rango de valores para x_0 y y_0
1	$x_0 \in [-0.5, 0.5]$, $y_0 \in [-0.5, 0.5]$
2	$x_0 \in [-1.0, 0.0]$, $y_0 \in [0.0, 1.0]$
3	$x_0 \in [0.0, 1.0]$, $y_0 \in [-0.6, 0.4]$
4	$x_0 \in [-1.5, -0.5]$, $y_0 \in [-0.5, 0.5]$
5	$x_0 \in [-0.4, 0.6]$, $y_0 \in [-0.4, 0.6]$
6	$x_0 \in [-1.0, 0.0]$, $y_0 \in [0.1, 1.1]$

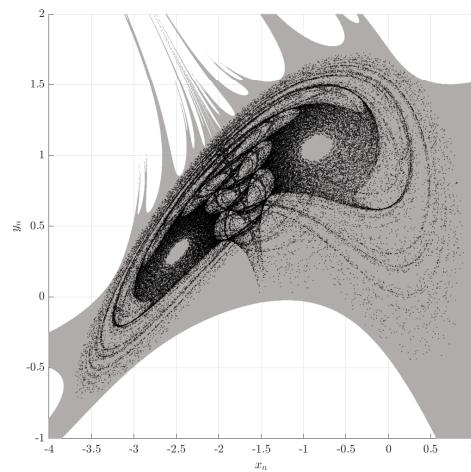
En [54] se utiliza el mapa caótico bidimensional propuesto en [52] y descrito en la ecuación (4.6) para diseñar un generador de números psedoaleatorios utilizando una arquitectura de punto fijo y extracción de bits haciendo uso de la operación mod 256 como se muestra en la ecuación (4.7).

$$s_{n+1} = \{x_{n+1} \bmod 256, y_{n+1} \bmod 256\} \quad (4.7)$$

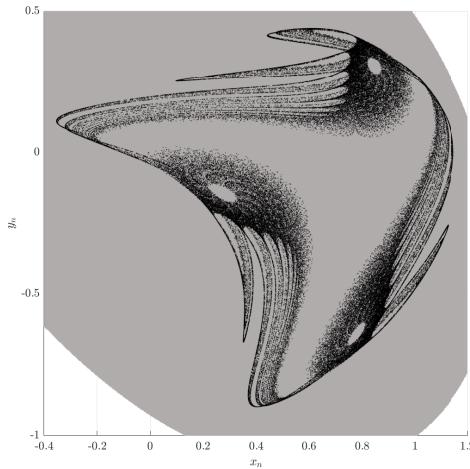
Figura 4.2: Dominio de atracción de diferentes atractores caóticos del mapa bidimensional generados en aritmética de punto flotante.



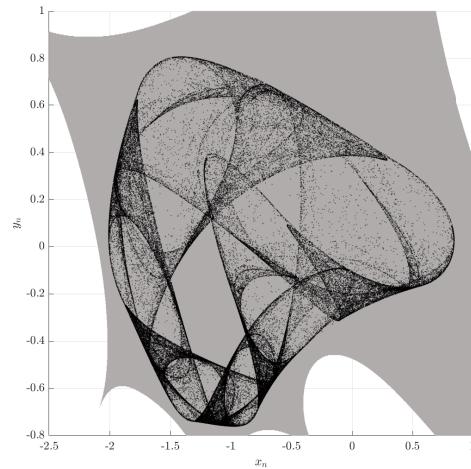
(a) Atractor 1



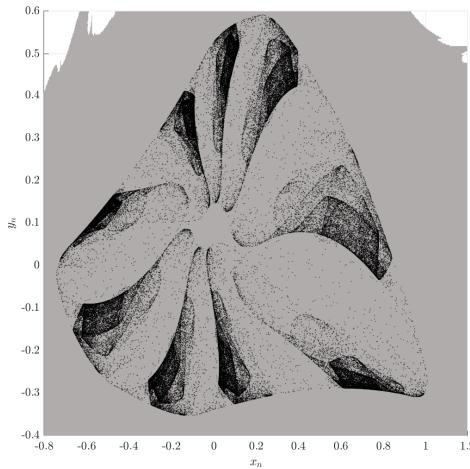
(b) Atractor 2



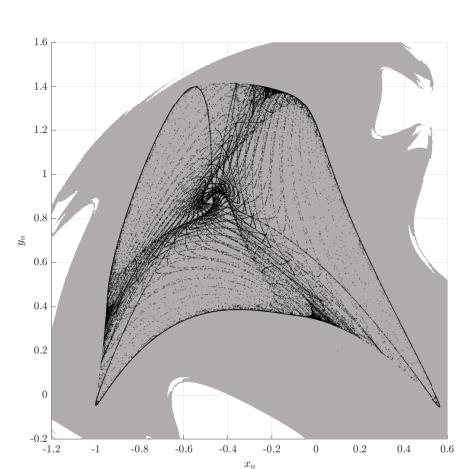
(c) Atractor 3



(d) Atractor 4



(e) Atractor 5



(f) Atractor 6

Asimismo se enfoca en utilizar solo los primeros 4 atractores que tienen los siguientes valores decodificados de la Tabla 4.3.

$$\begin{aligned} A_1 &= \{-0.6, -0.1, 1.1, 0.2, -0.8, 0.6, -0.7, 0.7, 0.7, 0.3, 0.6, 0.9\} \\ A_2 &= \{-1.0, 0.9, 0.4, -0.2, -0.6, -0.5, 0.4, 0.7, 0.3, -0.5, 0.7, -0.8\} \\ A_3 &= \{0.8, 1.0, -1.2, -1.0, 1.1, -0.9, 0.4, -0.4, -0.6, -0.2, -0.5, -0.7\} \\ A_4 &= \{-0.6, -0.4, -0.4, -0.8, 0.7, 0.3, -0.4, 0.4, 0.5, 0.5, 0.8, -0.1\} \end{aligned}$$

En este trabajo utilizaremos un enfoque similar para crear un generador de números aleatorios híbrido.

4.3. Análisis de punto fijo

Una vez definido el sistema y los atractores a utilizar hay que seleccionar el formato de punto fijo óptimo para cada atractor. Como ejemplo utilizaremos el Atractor 1. Podemos notar que el Atractor 1 esta acotado en un rango de $x_n \in [-0.866, 0.8915]$ y $y_n \in [-0.933, 0.7896]$. Si echamos un vistazo a la ecuación (4.6), que por comodidad se vuelve a mostrar abajo, podemos notar que las operaciones que pueden generar los números con magnitud más grandes son la combinación de alguna de las sumas. En cuanto a las multiplicaciones, debido a que la mayoría de los coeficientes $|a_n| < 1$, con la excepción de a_3 , después de realizar la multiplicación por cualquiera de ellos el número se vuelve más pequeño.

$$\begin{aligned} x_{n+1} &= a_1 + a_2x_n + a_3x_n^2 + a_4x_ny_n + a_5y_n + a_6y_n^2 \\ y_{n+1} &= a_7 + a_8x_n + a_9x_n^2 + a_{10}x_ny_n + a_{11}y_n + a_{12}y_n^2 \end{aligned}$$

Si tomamos los valores extremos de los rangos de x_n y y_n y analizamos las combinaciones de sumas que puedan generar el número más grande, o más pequeño en x_{n+1} y y_{n+1} , al cual llamaremos β , podemos encontrar la cantidad de bits necesaria para la parte entera a de la siguiente manera:

$$a = \log_2(\text{abs}(\beta)) \tag{4.8}$$

No obstante dependiendo el orden de las operaciones este análisis puede ser impreciso y solo nos da una aproximación del valor real de la parte entera. Consideremos lo siguiente, si se realizan primero todas las multiplicaciones y posteriormente se realizan las sumas, la acumulación de la suma puede ser mayor si se suman consecutivamente 3 números positivos a diferencia de intercalar números positivos con negativos.

Otra cosa a considerar es la cantidad de bits fraccionarios b que se requieren para que exista el caos. Si se tiene poca precisión en las operaciones la acumulación del error

puede provocar que el caos solo exista por un corto periodo de tiempo.

Debido a lo anterior muchos diseñadores optan por elegir el formato de punto fijo a prueba y error.

4.4. Simulador de arquitecturas digitales en C

Para no tener que recurrir al método de prueba y error para elegir el formato de punto fijo, en este trabajo se diseño un simulador de punto fijo en lenguaje C para poder comprobar la factibilidad de la arquitectura antes de pasar al diseño en hardware en FPGA.

La idea fundamental detrás de simular los diseños digitales en C antes de realizar su descripción en VHDL o Verilog es poder comprobar que las arquitecturas, ya sean de 16, 32 o 64 bits utilizando aritmética de punto fijo, funcionen correctamente desde el punto de vista de diseño, esto deja únicamente la posibilidad de cometer errores de sintaxis en el código en HDL los cuales se pueden analizar y solucionar por separado.

Se utilizó el compilador de GCC versión 11.2.0 en una plataforma x64 en Linux. El tipo de dato es el primer punto fundamental para el simulador. Existen diversos tipos de datos en lenguaje C sin embargo para arquitecturas con tamaños de bits definidos podemos reducirlo a cuatro: `_int128`, `long`, `int`, `short`. Cada uno de estos tipos datos tiene una cantidad definida de bytes asociados dependiendo de la computadora y el compilador, para revisar la cantidad de bytes podemos utilizar el Código A.1 del Apéndice A. La salida del código se muestra en la Tabla 4.5.

Tabla 4.5: Tamaños de tipos de datos en C, compilador de GCC versión 11.2.0 en una plataforma x64 en Linux.

Tipo	Bytes	Bits	Caracteres en hexadecimal
<code>char</code>	1	8	2
<code>short</code>	2	16	4
<code>int</code>	4	32	8
<code>long</code>	8	64	16
<code>_int128</code>	16	128	32

Primero se simuló el Atractor 1 de la ecuación (4.6) en punto flotante para tener una referencia de los posibles valores de salida, en el Código A.2 se muestra esta simulación. Posteriormente se simuló el mapa tomando como punto de partida una arquitectura de 64 bits para tener suficiente precisión. El análisis de punto fijo arrojó una parte entera aproximada $a = 1$, con esta parte entera el sistema funciona sin problemas, sin embargo para los otros atractores es necesario utilizar una mayor cantidad de bits para la parte entera. En la Tabla 4.6 se muestran los formatos seleccionados para cada atractor.

En el Código A.3 se muestra el diseño del simulador. Este cuenta con funciones para

Tabla 4.6: Número de bits usados en la implementación de cada uno de los atractores con aritmética de punto fijo.

Atractor	Bits parte entera	Bits parte fraccionaria	Rango	Precisión
1	3	60	$[-8.0, 8.0]$	8.6739×10^{-19}
2	4	59	$[-16.0, 16.0]$	1.7347×10^{-18}
3	4	59	$[-16.0, 16.0]$	1.7347×10^{-18}
4	3	60	$[-8.0, 8.0]$	8.6739×10^{-19}

realizar la conversión de punto flotante a punto fijo, multiplicación en punto fijo con truncamiento y finalmente conversión de punto fijo a punto flotante. En la Figura 4.3 se muestra el resultado de la simulación en punto fijo del atractor. Ya sea con 1, 2 o 3 bits para la parte entera, el Atractor 1 se simuló para 1000 millones de iteraciones y este no se salió de la órbita caótica.

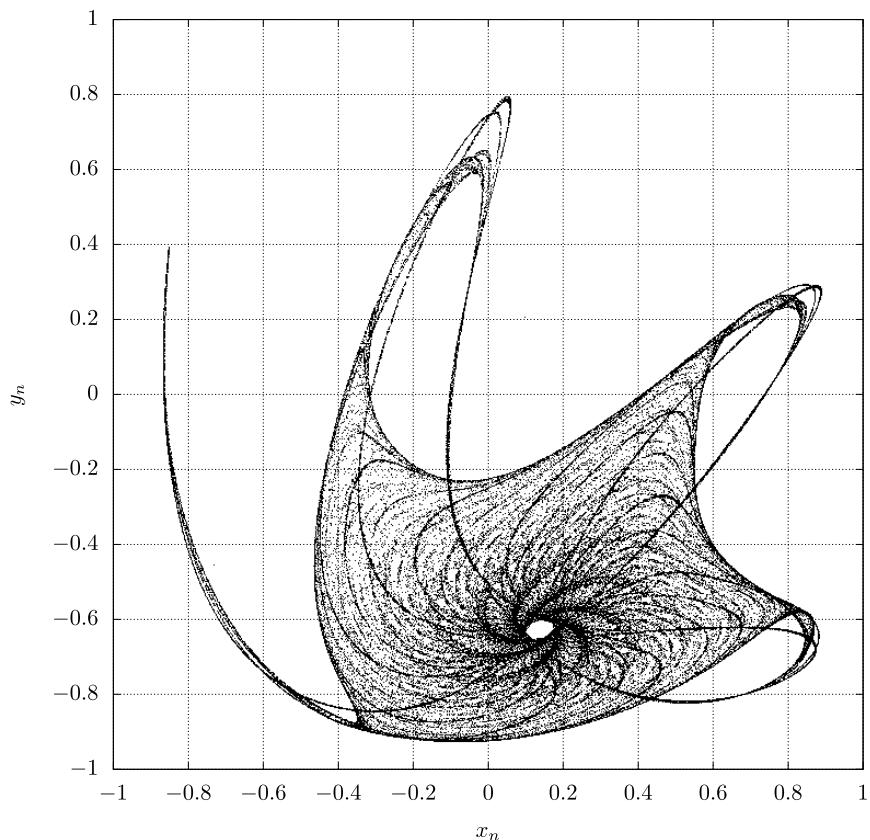


Figura 4.3: Simulación del mapa caótico en punto fijo, Atractor 1.

Para simular la extracción de bits de la ecuación (4.7) se pueden truncar los 8 bits menos significativos tanto para x_n como para y_n utilizando un casting del tipo `unsigned char` como se muestra en el Código A.6 y para poder convertir la salida a binario se utilizó el Código A.7.

Simulando 100 millones de iteraciones y generando palabras de 16 bits podemos ver la distribución de probabilidades de unos y ceros de cada palabra, en la Figura 4.4 se

muestra que la distribución es muy parecida a la ideal. Esto es lo esperado ya que los PRNG tienen por lo general buenas propiedades estadísticas.

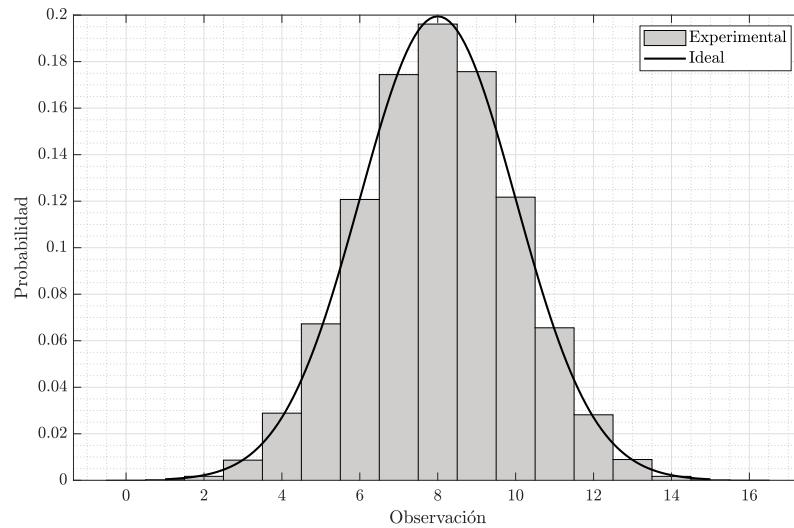


Figura 4.4: Distribución de 100 millones de palabras de 16 bits.

Una vez comprobado el diseño con el simulador en C, visualizando tanto su atractor como su distribución de probabilidades normal podemos pasar con seguridad a la fase de descripción del sistema en hardware. En este trabajo se utilizó VDHL.

4.5. Diseño de mapa caótico en VHDL

Si analizamos la ecuación (4.6) podemos notar que su representación no es la más óptima para programarse en hardware, ya que se requieren 10 sumas y 16 multiplicaciones. No obstante si se reutilizan las multiplicaciones de x_n^2 , $x_n y_n$ y y_n^2 solo se requieren 13 multiplicadores.

Si reescribimos la ecuación (4.6) como se muestra en la ecuación (4.9) se pueden eliminar dos multiplicadores más. Es decir la mínima cantidad de operaciones que se requieren para implementar el mapa son 10 sumadores y 11 multiplicadores.

$$\begin{aligned} x_{n+1} &= a_1 + (a_2 + a_3 x_n)x_n + a_4 x_n y_n + (a_5 + a_6 y_n)y_n \\ y_{n+1} &= a_7 + (a_8 + a_9 x_n)x_n + a_{10} x_n y_n + (a_{11} + a_{12} y_n)y_n \end{aligned} \quad (4.9)$$

Los bloques básicos para realizar la implementación serían:

1. Un sumador de 64 bits con signo, Código A.9.
2. Un multiplicador de 64 bits con signo y truncamiento al formato de punto fijo especificado, Código A.11.

3. Una memoria ROM que almacene los valores en el formato de punto fijo de las constantes a_n , Código A.10.
4. Un multiplexor para poder seleccionar entre la condición inicial y la retroalimentación del sistema, Código A.8.
5. Un registro que almacene la iteración actual, Código A.12.
6. Una máquina de estado para controlar todo el sistema, Código A.13.

Visto como diagrama de bloques la implementación el sistema completo se muestra en la Figura 4.5 y el Código de la implementación en A.14.

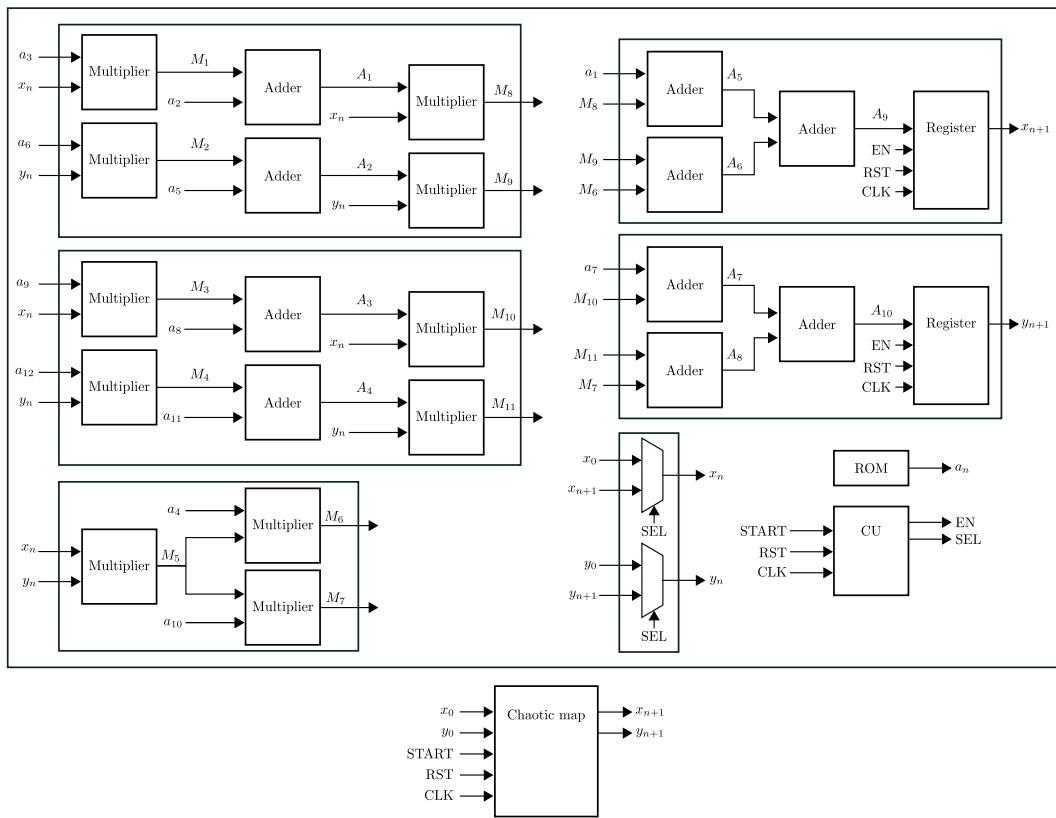
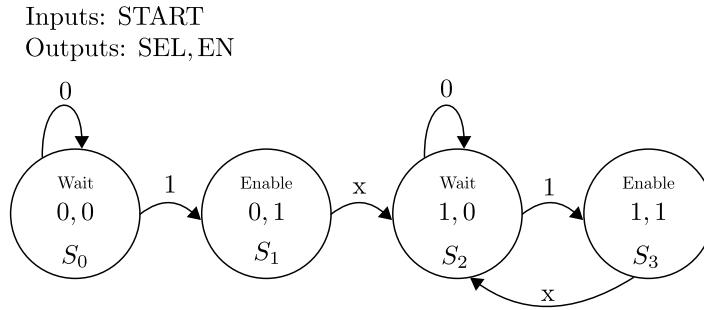


Figura 4.5: Diagrama de bloques del mapa caótico.

En la Figura 4.6 se muestra la máquina de estados que controla al sistema. El estado S_0 tiene la función de seleccionar la condición inicial direccionando el multiplexor a partir de la señal EN y esperar a que la señal de inicio START ocurra. El estado S_1 mantiene el direccionamiento del multiplexor y habilita el registro de salida que almacena la iteración actual a través de la señal EN. El estado S_2 es similar en funcionamiento al estado S_0 con la única diferencia que dirige el multiplexor al registro de salida para formar un lazo de retroalimentación para las futuras iteraciones. Finalmente el estado S_3 mantiene el multiplexor direccinando la retroalimentación y habilita el registro de salida para almacenar la iteración actual.

**Figura 4.6:** Máquina de estados de mapa caótico.

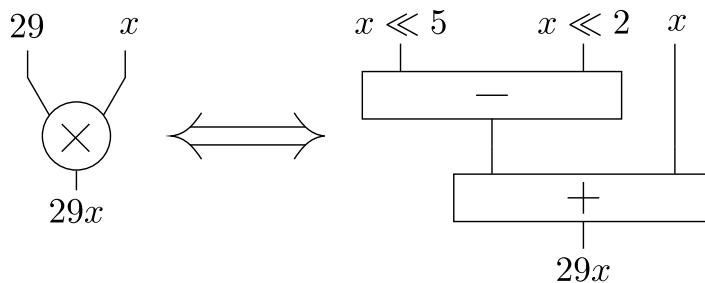
4.6. Multiplicador de una sola constante (SCM)

Es bien sabido que los multiplicadores son estructuras que consumen una gran cantidad de recursos en dispositivos digitales, por esta razón, es muy común que los diseñadores realicen optimizaciones creando multiplicadores de una sola constante para reducirlos [55, 56].

Al multiplicar por una constante conocida, podemos explotar las propiedades de la multiplicación binaria para obtener un circuito de hardware funcionalmente equivalente con menos recursos lógicos en comparación con conectar la constante en una entrada de un multiplicador genérico. La multiplicación constante puede implementarse como un conjunto de sumas, restas y desplazamientos binarios.

En los términos sencillos, el problema de la multiplicación constante puede describirse del siguiente modo: Dado un conjunto de constantes T , encontrar una realización suma-resta-desplazamiento de $t \cdot x$ para cada $t \in T$ donde x es una variable. El objetivo es minimizar el número de sumadores.

Por ejemplo supongamos que $T = \{29\}$, lo que significa que queremos implementar $29 \cdot x$. Una solución sería $29x = 32x - 4x + x = 2^5 - 2^2x + x = (x \ll 5) - (x \ll 2) + x$ donde $\ll n$ denota un desplazamiento a la izquierda de n bits. Ver Figura 4.7.

**Figura 4.7:** Ejemplo de multiplicación de una sola constante.

En [57] se propone un algoritmo óptimo y bastante práctico para poder crear multiplicadores de una constante de manera sencilla. Se requieren solo la constante y el número de bits fraccionarios, no obstante esta restringido a arquitecturas de máximo 32 bits y máximo 25 bits para la parte fraccionaria.

Se pueden crear a mano estos multiplicadores de una constante, no obstante encontrar una solución óptima para estos problemas es NP completo. NP es el conjunto de problemas en los que podemos comprobar en un tiempo razonable si una respuesta al problema es correcta o no y los problemas NP completos son los problemas más difíciles de todo NP. A grandes rasgos se buscan algoritmos que puedan resolver el problema de encontrar la solución de descomponer una multiplicación por una constante en sumas, restas y desplazamientos con la menor cantidad de recursos.

Si se tienen más de un multiplicador otro tipo de optimización que se puede realizar es analizar cuales corrimientos se comparten entre multiplicadores de una constante y reutilizar los corrimientos.

En este trabajo se diseñaron a mano multiplicadores de una constante para los parámetros a_n .

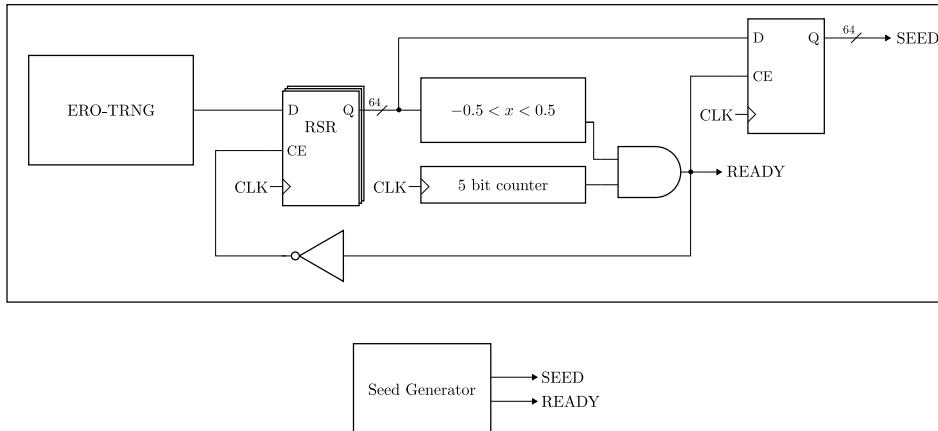
4.7. Diseño de generador de semillas

Hasta este punto, tenemos un mapa caótico implementado en FPGA capaz de generar números pseudoaleatorios en palabras de 16 bits cada dos ciclos de reloj. El problema es que las condiciones iniciales x_0 y y_0 son fijas, es decir, si algún atacante tuviera el conocimiento de las condiciones iniciales y conociera el mapa, este sería capaz de reproducir el sistema ya que el mapa por si mismo es determinista. Por esta razón se utiliza un TRNG para sembrar el generador de números pseudoaleatorios. Esto convierte al sistema en un generador de números aleatorios híbrido, aumenta la seguridad agregando una capa de complejidad y debido a que los TRNG extraen la aleatoriedad de elementos físicos, es muy difícil que alguien pueda adivinar las condiciones iniciales.

De todos los TRNG presentados en este trabajo el ERO-TRNG fue el candidato seleccionado para servir como generador de semillas. Este TRNG tiene una metodología bien estructurada, un modelo estocástico bien estudiado y sus propiedades tanto de entropía, uso de área y consumo de potencia son buenas. Además tiene la ventaja de ser el TRNG con mayor compatibilidad y su repetibilidad entre familias de FPGA. El diseño no requiere ninguna intervención manual para obtener resultados satisfactorios.

En la Figura 4.8 se muestra nuestra propuesta para extraer 64 bits aleatorios para generar la semilla del PRNG. Consiste en utilizar un registro de corrimiento a la derecha (RSR) que muestrea el ERO-TNG y compara si los 64 bits extraídos se encuentran dentro del rango del dominio de atracción del mapa caótico utilizando los parámetros del Atractor 1. Un contador de 5 bits de una sola vuelta asegura que se hayan generado 64 bits y no pasé una semilla incompleta.

Cuando ambas condiciones se cumplen el registro de corrimiento se deshabilita, la semilla pasa a un registro de salida y la señal READY se habilita para avisar que hay una semilla disponible.

**Figura 4.8:** Generador de semilla.

La semilla se utiliza como condición inicial tanto x_0 como para y_0 . Se utilizó la librería proporcionada por el https://labh-curien.univ-st-etienne.fr/cryptarchi/HECTOR_TRNG_designs que fue desarrollada durante el desarrollo del proyecto HECTOR (Hardware Enabled Crypto and Randomness [58]).

4.8. Teoría de FPGAs en Xilinx

Para poder realizar la implementación del núcleo ERO es necesario utilizar las primitivas y macros propias del fabricante de FPGA que para este trabajo es Xilinx. Las primitivas son componentes de Xilinx que son nativos de la arquitectura a la que se dirige y los macros son elementos que se encuentran en las bibliotecas UniMacro y Xilinx Parameterized Macros, las cuales se utilizan para instanciar elementos que son complejos de instanciar simplemente usando las primitivas, después las herramientas de síntesis expanden automáticamente estas macros a sus primitivas subyacentes. Los métodos de diseño disponibles son la instanciación, la inferencia, el catalogo IP y el soporte de macros, no obstante para este diseño solo se utilizan la instanciación, la cual permite instanciar un componente directamente en el diseño y es útil si se desea controlar el uso, la implementación y la ubicación exactos de los bloques individuales y el soporte de macros, el cual, utilizando las librerías antes mencionadas permiten abstraer la complejidad de utilizar únicamente primitivas simples.

Toda la información referente a los macros y primitivas se encuentran en la documentación oficial en el archivo llamado “Vivado Design Suite 7 Series FPGA Libraries Guide”. Para poder utilizar las primitivas y las macros es necesario agregar la librería UniMacro en la cabecera del archivo VHDL de la siguiente manera:

Código 4.1: Librería para primitivas de Xilinx.

```

LIBRARY UNISIM;
USE UNISIM.vcomponents.ALL;

```

4.8.1. Primitivas

4.8.1.1. LUT1: 1-Bit Look-Up Table with General Output

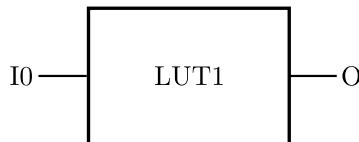


Figura 4.9: Esquemático de LUT1.

Este elemento proporciona una versión de look-up table de un búfer o inversor. Estos elementos son los bloques de construcción básicos. El parámetro INIT le da a la LUT su valor lógico. De forma predeterminada, este valor es cero, lo que lleva la salida a cero independientemente de los valores de entrada (actuando como tierra). Sin embargo, en la mayoría de los casos hay que determinar un nuevo valor INIT para especificar la función lógica de la primitiva LUT. Existen al menos dos métodos mediante los cuales se puede determinar el valor LUT. El método de la tabla lógica y el método de ecuación. En la Tabla 4.7 se muestran las entradas y salidas y la forma de configurar INIT y en el Código 4.2 se muestra su implementación en VHDL.

Tabla 4.7: Tabla lógica de LUT1.

Inputs I0	Outputs O
0	INIT[0]
1	INIT[1]

INIT = Binary number assigned to the INIT attribute

Código 4.2: Primitiva de LUT1.

```

library UNISIM;
use UNISIM.vcomponents.all;
-- LUT1: 1-input Look-Up Table with general output
-- 7 Series
-- Xilinx HDL Language Template, version 2021.2
LUT1_inst : LUT1
generic map (
    INIT => "00")
port map (
    O => O, -- LUT general output
    I0 => I0 -- LUT input
);
-- End of LUT1_inst instantiation
  
```

4.8.1.2. OBUFDS: Differential Signaling Output Buffer

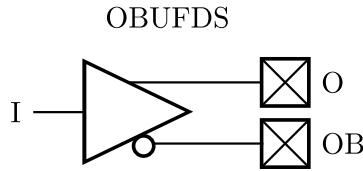


Figura 4.10: Esquemático de OBUFDS.

Este elemento de diseño es un búfer de salida única que admite señalización diferencial de bajo voltaje. OBUFDS aísla el circuito interno y proporciona corriente de accionamiento para las señales que salen del chip. Su salida se representa como dos puertos distintos (O y OB), uno considerado el “maestro” y el otro el “esclavo”. El maestro y el esclavo son fases opuestas de la misma señal lógica. En la Tabla 4.8 se muestran las entradas y salidas y en el Código 4.3 se muestra su implementación en VHDL.

Tabla 4.8: Tabla lógica de OBUFDS.

Inputs I	Outputs	
	O	OB
0	0	1
1	1	0

Código 4.3: Primitiva de OBUFDS.

```

Library UNISIM;
use UNISIM.vcomponents.all;
-- OBUFDS: Differential Output Buffer
-- 7 Series
-- Xilinx HDL Language Template, version 2021.2
OBUFDS_inst : OBUFDS
generic map (
    IO_STANDARD => "DEFAULT", -- Specify the output I/O standard
    SLEW => "SLOW") -- Specify the output slew rate
port map (
    O => O, -- Diff_p output (connect directly to top-level port)
    OB => OB, -- Diff_n output (connect directly to top-level port)
    I => I -- Buffer input
);
-- End of OBUFDS_inst instantiation

```

La librería del proyecto HECTOR [58] hace uso de estas librerías de Xilinx para describir el ERO-TRNG y parametriza el número de buffers para fácil modificación de los retardos.

Capítulo 5

Resultados experimentales

En este capítulo se presentan los resultados obtenidos de la implementación del TRNG híbrido, uso de recursos, velocidad de salida y pruebas estadísticas.

5.1. Comunicación RS232

Una vez implementado el generador de semillas y el mapa caótico es necesario poder transmitir la información para su posterior análisis. Para esto se realizó la implementación de una comunicación RS232 con un baudrate a 15200 sin bit de paridad. Como solo es necesario transmitir datos del sistema a una computadora solo se implementó la transmisión. En la Figura 5.1 se muestra el diagrama a bloques de la transmisión y en la Figura 5.2 el diagrama de la maquina de estados que controla la transmisión.

Se eligió no transmitir el bit de paridad para que los paquetes enviados sean fáciles de procesar, además esto aumenta un poco la velocidad de transmisión ya que se manda un bit menos, y para nuestro caso, en el que se requieren transmitir 100 millones de bits aleatorios es un ahorro significativo.

5.2. TRNG híbrido

De manera que el sistema completo se muestra en la Figura 5.3. Este consiste en el generador de semillas aleatorios utilizando el ERO-TRNG como núcleo, el mapa caótico el cual calcula cada dos ciclos de reloj una iteración nueva, la operación mod 256 para extraer 16 bits aleatorios del mapa, la comunicación RS232 en conjunto un multiplexor para mandar paquetes de 8 bits y una maquina de estados cuyo diagrama se muestra en la Figura 5.4, que controla y sincroniza todo el sistema.

El generador funciona de la siguiente manera. Cuando el sistema se enciende el generador de semillas comienza a generar bits aleatorios utilizando el ERO-TRNG interno, cuando este proceso termina la señal READY se activa avisando que hay una

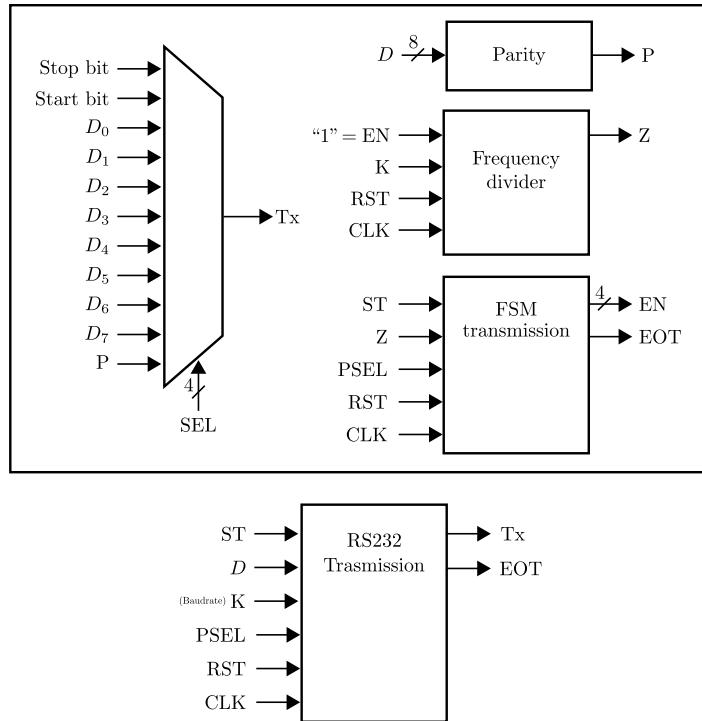


Figura 5.1: Diagrama de bloques de transmisión RS232.

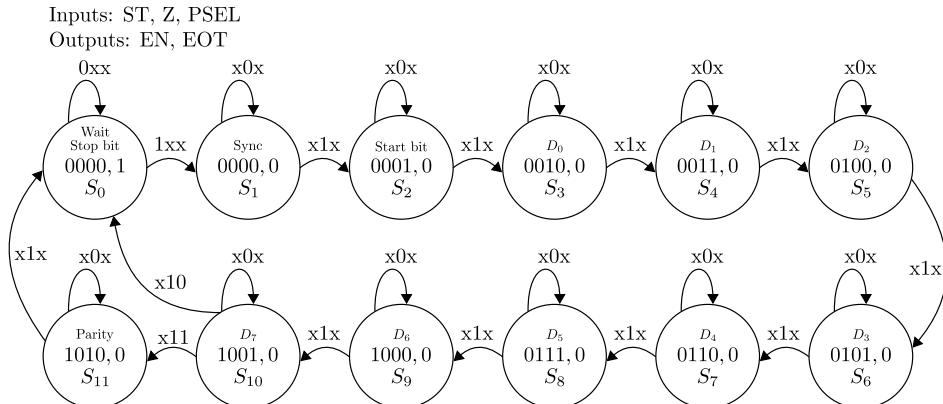


Figura 5.2: Máquina de estados para la transmisión RS232.

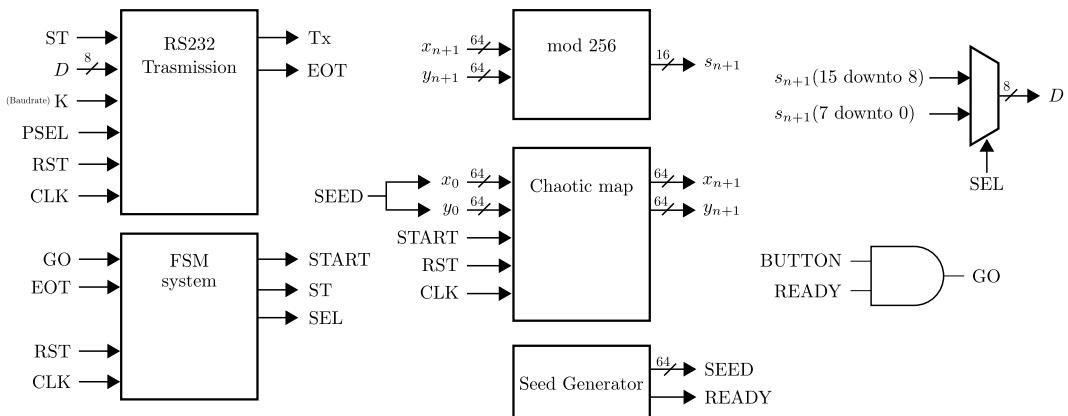


Figura 5.3: Diagrama de bloques de TRNG híbrido.

semilla válida en SEED y este proceso no vuelve a ocurrir a menos que la FPGA se reinicie.

El sistema espera que el usuario active la señal BUTTON para comenzar. Cuando el sistema se activa la maquina de estados calcula una iteración del mapa caótico, posteriormente se activa la transmisión que envía los primeros 8 bits aleatorios por medio del protocolo de comunicación RS232, una vez concluida, se envían los siguientes 8 bits aleatorios y el sistema regresa a esperar por la señal BUTTON.

Si la señal BUTTON se queda activa este proceso se vuelve cíclico y se generan tantos bits aleatorios cómo el usuario desee.

Inputs: GO, EOT
Outputs: START, ST, SEL

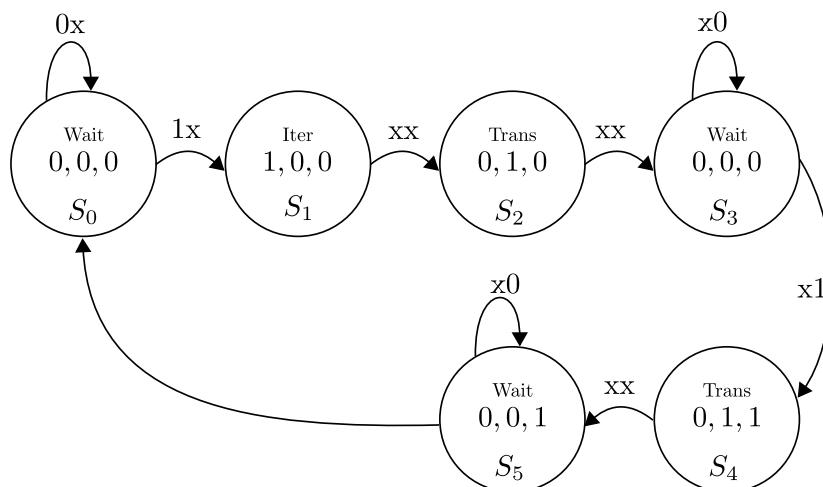


Figura 5.4: Máquina de estados de TRNG híbrido.

5.3. Pruebas estadísticas

Utilizando una computadora con Windows 10 y el programa PuTTY como interfaz, el sistema corriendo se ejecutó por aproximadamente 20 minutos y se almacenaron alrededor 100 millones de bits aleatorios, lo que representan un archivo de aproximadamente 700,000 KB. Los datos se almacenaron como caracteres ASCII donde cada carácter contiene 8 bits aleatorios. Utilizando un programa de C se convirtieron los caracteres a números binarios para poder ingresarse con facilidad a la suite de pruebas estadísticas NIST [5, 7]. En la Tabla 5.1 se muestran los resultados de las pruebas estadísticas NIST del sistema a 100 secuencias de un millón datos y en la Tabla 5.2 se muestran los resultados de las pruebas estadísticas NIST a 1000 secuencias de un millón de bits aleatorio.

La proporción mínima de aprobados para cada prueba estadística es aproximadamente de 96 para 100 millones de secuencias binarias y de 980 para 1000 millones de

Tabla 5.1: Resultados de la aplicación de las pruebas NIST al TRNG híbrido implementado en aritmética de punto fijo con 100 secuencias de un millón de datos.

Test name	p-value	%
Frequency	0.383827	0.99
Block frequency	0.108791	1.00
Cumulative sums	0.401199	1.00
Runs	0.971699	0.99
Longest Run	0.759756	1.00
Rank	0.383827	0.99
FFT	0.383827	0.99
NonOverlapping template	0.480298	0.99
Overlapping template	0.883171	0.99
Universal	0.574903	0.99
Approximate entropy	0.759756	0.98
Random excursions	0.265539	0.99
Random excursions variant	0.312463	0.99
Serial	0.595604	1.00
Linear complexity	0.574903	1.00

Tabla 5.2: Resultados de la aplicación de las pruebas NIST al TRNG híbrido implementado en aritmética de punto fijo con 1000 secuencias de un millón de datos.

Test name	p-value	%
Frequency	0.587274	0.989
Block frequency	0.796268	0.998
Cumulative sums	0.848047	0.990
Runs	0.614226	0.988
Longest Run	0.660012	0.992
Rank	0.255705	0.990
FFT	0.072514	0.994
NonOverlapping template	0.481082	0.989
Overlapping template	0.143686	0.988
Universal	0.228367	0.988
Approximate entropy	0.786830	0.992
Random excursions	0.447308	0.990
Random excursions variant	0.405096	0.992
Serial	0.124135	0.993
Linear complexity	0.008816	0.989

secuencias binarias. Para ambos casos todas las pruebas pasaron la proporción mínima.

Además analizando la distribución de probabilidad de unos y ceros que se muestra en la Figura 5.5 es consistente con el análisis en simulación y el histograma, que se muestra en la Figura 5.6 esta perfectamente distribuido de manera uniforme, por lo que de manera cualitativa y cuantitativa este TRNG híbrido cumple con las requisitos estadísticos.

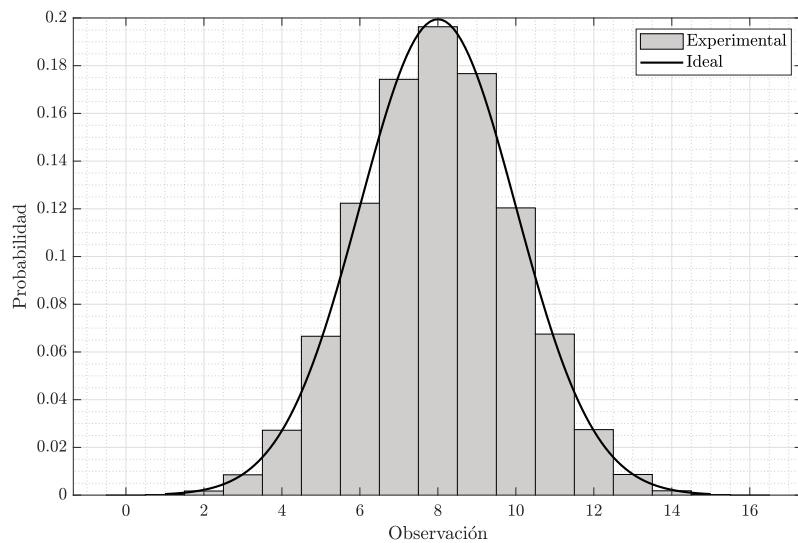


Figura 5.5: Distribución de 100 millones de palabras de 16 bits experimentales.

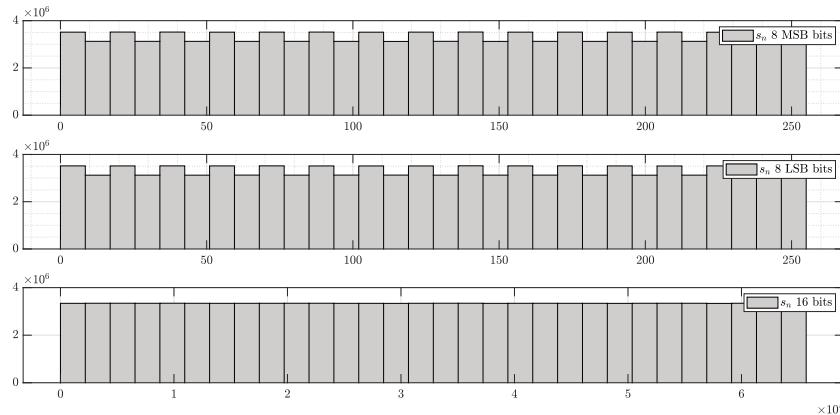


Figura 5.6: Histograma de resultados experimentales primeros 100 mil secuencias binarias.

5.4. Uso de recursos y velocidad

En cuanto recursos utilizados, en este trabajo se utilizó la tarjeta de desarrollo Basys 3 del fabricante Digilent, la cual cuenta con un FPGA Artix 7 xc7a35tcpg236-1

y un oscilador de 100 MHz. Esta tarjeta se encuentra dentro de la gama media-baja en lo concerniente a recursos, ya que dispone de una cuarta parte o incluso menos de los recursos de tarjetas hermanas como la Nexys A7 o la Zybo Z7. En la Tabla 5.3 se muestran los recursos utilizados todos los multiplicadores son multiplicadores completos. Debido que el sistema necesita 11 multiplicadores de 64 bits, y un solo multiplicador de este tamaño utiliza aproximadamente un 18 % de los DSPs disponibles, el diseño casi utiliza la totalidad de la FPGA.

Tabla 5.3: Uso de recursos del TRNG híbrido con multiplicadores completos.

Recursos	Utilización	Disponibles	Utilización %
LUTs	17767	20800	85.42
FF	148	41600	0.36
DSP	90	90	100.0

No obstante si se utilizan multiplicadores de una sola constante para todos los parámetros a_n el uso de recursos se reduce drásticamente como se muestra en la Figura 5.4.

Tabla 5.4: Uso de recursos del TRNG híbrido con multiplicadores de una sola constante en parámetros a_n .

Recursos	Utilización	Disponibles	Utilización %
LUTs	1469	20800	7.06
FF	146	41600	0.35
DSP	80	90	88.89

Para este diseño en particular es necesario utilizar los multiplicadores de una sola constante para tener disponibles suficientes recursos para realizar otras aplicaciones en conjunto con el TRNG híbrido.

Debido a que el sistema trabaja a una velocidad de 100 MHz y cada iteración del mapa caótico necesita 2 ciclos de reloj para calcular la iteración y un ciclo para mandar el resultado a un registro de salida, la velocidad del generador es de 30 ns y como se generan 16 bits aleatorios en ese tiempo, la velocidad de salida o throughput en inglés es de 533.33 Mbits/s, que en comparación con [54] que realizó una implementación similar en un microcontrolador de alto rendimiento de la familia STM32, en el que se alcanzó una velocidad de salida de 173.35 Kbits/s, nuestro sistema puede utilizarse para aplicaciones que requieran mayor velocidad.

Capítulo 6

Conclusiones

- El núcleo ERO-TRNG a pesar de no ser el más rápido de los generadores que se pueden implementar dentro de las FPGA, tiene muy buena entropía, un modelo estocástico muy bien estudiado, una metodología de diseño bien estructurada y por lo mismo es fácil de implementarse no requiriendo intervención manual en la colocación de los componentes dentro del FPGA. Debido a lo anterior fue perfecto para generar semillas, una vez obtenida la semilla la velocidad del sistema ya no depende del ERO-TRNG lo que contrarresta su desventaja principal. Su repetibilidad en diferentes familias de FPGA, la seguridad que agrega al generador y la pequeña cantidad de recursos que requiere son las ventajas que presentó este núcleo.
- El mapa caótico bidimensional que se seleccionó para este trabajo es muy flexible, ya que se tiene 12 parámetros diferentes para configurarse, lo que se ve reflejado en diferentes atractores que pueden utilizarse para generar secuencias de bits aleatorios. Mientras las condiciones iniciales del atractor se encuentren dentro del dominio de atracción de este, podemos sembrar el mapa sin problemas.
- La implementación del TRNG híbrido utiliza un 88 % de los DPS y tan solo un 7 % de los LUTs de la FPGA, no obstante considerando que la FPGA utilizada es de bajos recursos, para FPGAs más grandes las cuales tienen de 4 a 5 recursos, este sistema no representa un gran consumo de área.
- El TRNG híbrido que se diseño en este trabajo pasó todas las pruebas NIST y el análisis estadístico demostró distribuciones uniformes, además debido a que en su núcleo se encuentra un TRNG que pasa todas las pruebas de la AIS20/31 la seguridad del sistema se ve garantizada mientras nadie tenga acceso a la semilla.
- La velocidad obtenida por el TRNG híbrido fue de 533.33 Mbit/s, es ligeramente superior a sistemas similares las cuales rondan los 400 Mbit/s.

Comprobación de glosarios

FPGA

RNG

TRNG

TERO

ERO-TRNG

COSO-TRNG

MURO-TRNG

TERO-TRNG

PLL-TRNG

STR-TRNG

Apéndice A

Códigos

A.1. Códigos en C

Código A.1: Comprobar el número de bytes de los tipos de dato del sistema.

```
/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A1_check_sys_bytes.c
 * Description: Check the number of byte of each data type.
 * Compile: gcc -o A1_check.exe A1_check_sys_bytes.c
 * Run: ./A1_check.exe
*/
#include <stdio.h>
#include <stdlib.h>

int main(void){
    // Data type <float> -> 4 bytes, in other words 32 bits.
    printf("float is %lu bytes.\n", sizeof(float));

    // Data type <double> -> 8 bytes, in other words 64 bits.
    printf("double is %lu bytes.\n", sizeof(double));

    // Data type <__int128> -> 16 bytes, in other words 128 bits.
    printf("__int128 is %lu bytes.\n", sizeof(__int128));

    // Data type <long> -> 8 bytes, in other words 64 bits.
    printf("long is %lu bytes.\n", sizeof(long));

    // Data type <int> -> 4 bytes, in other words 32 bits.
    printf("int is %lu bytes.\n", sizeof(int));

    // Data type <short> -> 2 bytes, in other words 16 bits.
    printf("short is %lu bytes.\n", sizeof(short));

    // Data type <char> -> 1 bytes, in other words 8 bits.
    printf("char is %lu bytes.\n", sizeof(char));

    return 0;
}
```

Código A.2: Simulación de mapa caótico en punto flotante.

```
/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A2_chaotic_map_float.c
 * Description: Sproot chaotic map simulation using floating point, for unix and windows
 * Compile: gcc -o A2_chaotic_float.exe A2_chaotic_map_float.c
 * Run: ./A2_chaotic_float.exe
*/
#include <stdio.h>
#include <stdlib.h>

int main(void){

    // Open file
    FILE *fpointer = fopen("output_chaotic.txt","w");
    double ai[12] = {-0.6, -0.1, 1.1, 0.2, -0.8, 0.6, -0.7, 0.7, 0.7, 0.3, 0.6, 0.9};
    double x0 = 0.1;
    double y0 = 0.2;
    double xn = x0;
    double yn = y0;
    double xni = 0.0;
    double yni = 0.0;
    int iter = 100000;

    // Initial conditions
    printf(" # x0: %lf\n", x0 );
    printf(" # y0: %lf\n", y0 );
    printf(" # iter: %d\n", iter );
    printf(" # chaotic map generated, see output_chaotic.txt\n");

    fprintf(fpointer,"%32.29lf\t%32.29lf\n",xn, yn);
    for(int i = 0; i < iter; i++){
        xni = ai[0] + ai[1]*xn + ai[2]*xn*xn + ai[3]*xn*yn + ai[4]*yn + ai[5]*yn*yn;
        yni = ai[6] + ai[7]*xn + ai[8]*xn*xn + ai[9]*xn*yn + ai[10]*yn + ai[11]*yn*yn;

        xn = xni;
        yn = yni;
        fprintf(fpointer,"%32.29lf\t%32.29lf\n",xn, yn);
    }

    fclose(fpointer); // Close file
    return 0;
}
```

Código A.3: Simulación de mapa caótico en punto fijo.

```

/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A3_chaotic_map_fixed.c
 * Description: Sprott chaotic map simulation using fixed point
 * Compile: gcc -o A3_chaotic_fixed.exe A3_chaotic_map_fixed.c
 * Run: ./A3_chaotic_fixed.exe
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Global variables
int _a;                      // integer part
int _b;                      // fractional part
long _power;

// A(a,b) fixed point representation
void initialize( int a, int b ){
    _a = a;
    _b = b;
    _power = (long)1 << _b;
}

// double to fixed point conversion with truncation
long setNumber( double v ){
    return ( (long)(v*_power) );
}

// fixed point to double conversion
double getNumber( long r ){
    return ( (double)r/_power);
}

// fixed point multiplication with truncation
long multTrunc( long x, long y ){
    __int128 r;
    __int128 a=0;
    __int128 b=0;
    a = x;
    b = y;
    r = a*b;
    r = r >> _b;
    return( r );
}

int main(void){
    FILE *fpointer = fopen("output_chaotic_fixed.txt","w");

    long ai[12], xi, yi, xni, yni;
    double x0,y0;
    int integer, frac;

    integer = 3; frac = 64 - integer - 1;
    initialize( integer, frac );
    printf(" Representation A(a,b) = A(%d, %d)\n a: integer\tb: fractional \n",integer,frac);

    x0 = 0.1; y0 = 0.2;
    printf(" # x0: %f\n # y0: %f\n", x0, y0);

    xi = setNumber( x0 );
    yi = setNumber( y0 );
    printf(" # x0 real: %2.10f\n # y0 real: %2.10f\n",getNumber(xi), getNumber(yi) );
    printf(" # chaotic map generated, see output_chaotic_fixed.txt\n");

    ai[0] = setNumber( -0.6 );
    ai[1] = setNumber( -0.1 );
    ai[2] = setNumber( 1.1 );
    ai[3] = setNumber( 0.2 );
    ai[4] = setNumber( -0.8 );
    ai[5] = setNumber( 0.6 );
    ai[6] = setNumber( -0.7 );
    ai[7] = setNumber( 0.7 );
    ai[8] = setNumber( 0.7 );
    ai[9] = setNumber( 0.3 );
    ai[10] = setNumber( 0.6 );
    ai[11] = setNumber( 0.9 );

    fprintf(fpointer,"%32.29lf\t%32.29lf\t%16.16lx\t%16.16lx\n",getNumber( xi ), getNumber( yi ),xi,yi);
    for(int i = 0; i<100000; i++){
        xni = ai[0] + multTrunc( multTrunc( ai[2] , xi ) + ai[1] , xi ) + multTrunc(multTrunc(xi,yi), ai[3]) +
            multTrunc( multTrunc( ai[5] , yi ) + ai[4] , yi );

        yni = ai[6] + multTrunc( multTrunc( ai[8] , xi ) + ai[7] , xi ) + multTrunc(multTrunc(xi,yi), ai[9]) +
            multTrunc( multTrunc( ai[11] , yi ) + ai[10] , yi );

        xi = xni;
        yi = yni;
        fprintf(fpointer,"%32.29lf\t%32.29lf\t%16.16lx\t%16.16lx\n",getNumber( xi ), getNumber( yi ),xi,yi);
    }

    fclose(fpointer);
    return 0;
}

```

Código A.4: Generador de memoria ROM de condiciones iniciales.

```

/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A4_rom_gen_chaotic_map.c
 * Description: VHDL ROM generator for chaotic map
 * Compile: gcc -o A4_rom_cm.exe A4_rom_gen_chaotic_map.c
 * Run: ./A4_rom_cm.exe
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Global variables
int _a; // integer part
int _b; // fractional part
long _power;

// A(a,b) fixed point representation
void initialize( int a, int b ){
    _a = a;
    _b = b;
    _power = (long)1 << _b;
}

// double to fixed point convertion with truncation
long setNumber( double v ){
    return ( (long)(v*_power) );
}

// fixed point to double convertion
double getNumber( long r ){
    return ( (double)r/_power);
}

// fixed point to binary_string for 64 bits
char *to_binary(long n){
    char *binary = (char *)malloc(sizeof(char) * (64 + 1)); // extra byte for null terminator
    int k = 0;
    unsigned long mask, i;
    mask = ( (long)1 << (64 - 1) );
    for ( i = mask; i > 0; i >>= 1) {
        binary[k++] = (n & i) ? '1' : '0';
    }
    binary[k] = '\0';
    return binary;
}

int main(void){
    double an[12] = {-0.6, -0.1, 1.1, 0.2, -0.8, 0.6, -0.7, 0.7, 0.7, 0.3, 0.6, 0.9};
    long fixed_value = 0;
    int i = 0;
    int length = sizeof(an)/sizeof(an[0]);

    int integer, frac;
    integer = 3; frac = 64 - integer - 1;
    initialize( integer, frac );
    printf(" Representation A(a,b) = A(%d,%d)\n a: integer\tb: fractional \n",integer,frac);
    printf(" # Number of elements: %d\n", length);
    printf(" # ROM generated, see rom_cm.vhd\n");

    FILE *fpointer = fopen("rom_cm.vhd","w"); // Open file
    fprintf(fpointer,"library ieee;\n");
    fprintf(fpointer,"use ieee.std_logic_1164.all;\n\n");
    fprintf(fpointer,"entity rom_cm is\n");
    fprintf(fpointer," generic(n : integer := 64);\n");
    fprintf(fpointer," port(\n");
    fprintf(fpointer," ");
    for(i = 1; i <= length; i++){
        if(i == length)
            fprintf(fpointer,"a%d",i);
        else
            fprintf(fpointer,"a%d,",i);
    }
    fprintf(fpointer," : out std_logic_vector(n-1 downto 0)\n");
    fprintf(fpointer," );\n");
    fprintf(fpointer,"end;\n\n");
    fprintf(fpointer,"architecture arch of rom_cm is\n");
    fprintf(fpointer,"begin\n");

    char *binary_string;
    for(i = 0; i<length; i++){
        fixed_value = setNumber(an[i]);
        binary_string = to_binary(fixed_value);
        fprintf(fpointer," a%d <= \"%s\"; --%5.2lf\n",i+1, binary_string, getNumber(fixed_value) );
        free(binary_string);
        binary_string = NULL;
    }

    fprintf(fpointer,"end;\n");
    fclose(fpointer); // Close file
    return 0;
}

```

Código A.5: Convertidor de punto flotante a punto fijo.

```

/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A5_fixed_point_converter.c
 * Description: Fixed point converter for windows
 * Compile: gcc -o A5_converter.exe A5_fixed_point_converter.c
 * Run: ./A5_converter.exe
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Global variables
int _a;                      // integer part
int _b;                      // fractional part
long _power;

// A(a,b) fixed point representation
void initialize( int a, int b ){
    _a = a;
    _b = b;
    _power = (long)1 << _b;
}

// double to fixed point conversion with truncation
long setNumber( double v ){
    return ( (long)(v*_power) );
}

// fixed point to double conversion
double getNumber( long r ){
    return ( (double)r/_power);
}

// fixed point to binary_string for 64 bits
char *to_binary(long n){
    char *binary = (char *)malloc(sizeof(char) * (64 + 1)); // extra byte for null terminator
    int k = 0;
    unsigned long mask, i;
    mask = ( (long)1 << (64 - 1));
    for (i = mask; i > 0; i >>= 1) {
        binary[k++] = (n & i) ? '1' : '0';
    }
    binary[k] = '\0';
    return binary;
}

int main(void){
    FILE *fpointer = fopen("conversion.txt","w");

    double values[] = {0.1, 0.2};
    long fixed_value = 0;
    int length = sizeof(values)/sizeof(values[0]);
    int integer, frac;

    integer = 3; frac = 64 - integer - 1;
    initialize( integer, frac );
    printf(" Representation A(a,b) = A(%d, %d)\n a: integer\tb: fractional \n",integer,frac);
    printf(" # Number of elements: %d\n", length);
    printf(" # See conversion.txt\n\n");

    char *binary_string;
    for (int i = 0; i < length; i++){
        fixed_value = setNumber(values[i]);
        binary_string = to_binary(fixed_value);
        printf(" v %d <= \"%s\"; -- %.2lf\n",i+1, binary_string, getNumber(fixed_value) );
        fprintf(fpointer," v %d <= \"%s\"; -- %.2lf\n",i+1, binary_string, getNumber(fixed_value) );
        free(binary_string);
        binary_string = NULL;
    }

    fclose(fpointer);
    return 0;
}

```

Código A.6: Simulación de mapa caótico en punto fijo y operación mod 256.

```

/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A6_chaotic_map_mod.c
 * Description: Sprotochic map simulation using fixed point and mod 256
 * Compile: gcc -o A6_chaotic_mod.exe A6_chaotic_map_mod.c
 * Run: ./A6_chaotic_mod.exe
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Global variables
int _a; // integer part
int _b; // fractional part
long _power;

// A(a,b) fixed point representation
void initialize( int a, int b ){
    _a = a;
    _b = b;
    _power = (long)1 << _b;
}

// double to fixed point conversion with truncation
long setNumber( double v ){
    return ( (long)(v*_power) );
}

// fixed point to double conversion
double getNumber( long r ){
    return ( (double)r/_power);
}

// fixed point multiplication with truncation
long multTrunc( long x, long y ){
    __int128 r;
    __int128 a=0;
    __int128 b=0;
    a = x;
    b = y;
    r = a*b;
    r = r >> _b;
    return( r );
}

int main(void){
    FILE *fpointer = fopen("output_chaotic_mod.txt","w");

    long ai[12], xi, yi, xni, yni;
    unsigned short si;
    double x0,y0;
    int integer, frac;

    integer = 3; frac = 64 - integer - 1;
    initialize( integer, frac );
    printf(" Representation A(a,b) = A(%d,%d)\n a: integer\tb: fractional \n",integer,frac);

    x0 = 0.1; y0 = 0.2;
    printf(" # x0: %f\n # y0: %f\n", x0, y0);

    xi = setNumber( x0 ); yi = setNumber( y0 );
    printf(" # x0 real: %2.10f\n # y0 real: %2.10f\n",getNumber(xi), getNumber(yi) );
    printf(" # see output_chaotic_mod.txt\n");

    ai[0] = setNumber( -0.6 );
    ai[1] = setNumber( -0.1 );
    ai[2] = setNumber( 1.1 );
    ai[3] = setNumber( 0.2 );
    ai[4] = setNumber( -0.8 );
    ai[5] = setNumber( 0.6 );
    ai[6] = setNumber( -0.7 );
    ai[7] = setNumber( 0.7 );
    ai[8] = setNumber( 0.7 );
    ai[9] = setNumber( 0.3 );
    ai[10] = setNumber( 0.6 );
    ai[11] = setNumber( 0.9 );

    for(int i = 0; i<20; i++){
        xni = ai[0] + multTrunc( multTrunc( ai[2] , xi ) + ai[1] , xi ) + multTrunc(multTrunc(xi,yi), ai[3]) +
            multTrunc( multTrunc( ai[5] , yi ) + ai[4] , yi );

        yni = ai[6] + multTrunc( multTrunc( ai[8] , xi ) + ai[7] , xi ) + multTrunc(multTrunc(xi,yi), ai[9]) +
            multTrunc( multTrunc( ai[11] , yi ) + ai[10] , yi );

        xi = xni;
        yi = yni;
        /* fprintf(fpointer,"%c%c", (unsigned char)xi, (unsigned char)yi); */
        fprintf(fpointer,"%2.2x %2.2x\n", (unsigned char)xi, (unsigned char)yi);
    }

    fclose(fpointer);
    return 0;
}

```

Código A.7: Simulación de mapa caótico en punto fijo y operación mod 256 salida binaria.

```

/*
 * Author: Ciro Fabian Bermudez Marquez
 * Date: 16/06/2022
 * Desing name: A7_chaotic_map_mod_bin.c
 * Description: Sproto chaotic map simulation using fixed point and mod 256 output in binary
 * Compile: gcc -o A7_chaotic_mod_bin.exe A7_chaotic_map_mod_bin.c
 * Run: ./A7_chaotic_mod_bin.exe
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

// Global variables
int _a;           // integer part
int _b;           // fractional part
long _power;

// A(a,b) fixed point representation
void initialize( int a, int b ){
    _a = a;
    _b = b;
    _power = (long)1 << _b;
}

// double to fixed point conversion with truncation
long setNumber( double v ){
    return ( (long)(v*_power) );
}

// fixed point to double conversion
double getNumber( long r ){
    return ( (double)r/_power );
}

// fixed point multiplication with truncation
long multTrunc( long x, long y ){
    __int128 r;
    __int128 a=0;
    __int128 b=0;
    a = x;
    b = y;
    r = a*b;
    r = r >> _b;
    return( r );
}

// fixed point to binary_string for 64 bits
char *to_binary(unsigned char n){
    char *binary = (char *)malloc(sizeof(char) * (8 + 1)); // extra byte for null terminator
    int k = 0;
    unsigned char mask, i;
    mask = (char)1 << (8 - 1);
    for (i = mask; i > 0; i >= 1) {
        binary[k++] = (n & i) ? '1' : '0';
    }
    binary[k] = '\0';
    return binary;
}

int main(void){
    FILE *fp pointer = fopen("output_chaotic_mod_binary.txt","w");

    long ai[12], xi, yi, xni, yni;
    unsigned char x,y;
    double x0,y0;
    int integer, frac;

    integer = 3; frac = 64 - integer - 1;
    initialize( integer, frac );
    printf(" Representation A(a,b) = A(%d, %d)\n a: integer\tb: fractional \n",integer,frac);

    x0 = 0.1; y0 = 0.2;
    printf(" # x0: %f\n # y0: %f\n", x0, y0);

    xi = setNumber( x0 );
    yi = setNumber( y0 );
    printf(" # x0 real: %2.10f\n # y0 real: %2.10f\n",getNumber(xi), getNumber(yi) );
    printf(" # see output_chaotic_mod_binary.txt\n");

    ai[0] = setNumber( -0.6 );
    ai[1] = setNumber( -0.1 );
    ai[2] = setNumber( 1.1 );
    ai[3] = setNumber( 0.2 );
    ai[4] = setNumber( -0.8 );
    ai[5] = setNumber( 0.6 );
    ai[6] = setNumber( -0.7 );
    ai[7] = setNumber( 0.7 );
    ai[8] = setNumber( 0.7 );
    ai[9] = setNumber( 0.3 );
    ai[10] = setNumber( 0.6 );
    ai[11] = setNumber( 0.9 );

    char *xp, *yp;
}

```

```

for(int i = 0; i<20; i++){
    xni = ai[0] + multTrunc( multTrunc( ai[2] , xi ) + ai[1] , xi ) + multTrunc(multTrunc(xi,yi), ai[3]) +
        multTrunc( multTrunc( ai[5] , yi ) + ai[4] , yi );

    yni = ai[6] + multTrunc( multTrunc( ai[8] , xi ) + ai[7] , xi ) + multTrunc(multTrunc(xi,yi), ai[9]) +
        multTrunc( multTrunc( ai[11] , yi ) + ai[10] , yi);

    xi = xni; yi = yni;
    x = (unsigned char)xi; y = (unsigned char)yi;
    xp = to_binary(x); yp = to_binary(y);
    fprintf(fpointer, "%s %s\n",xp, yp);
    free(xp); xp = NULL;
    free(yp); yp = NULL;
}

fclose(fpointer);
return 0;
}

```

A.2. Códigos en VHDL de mapa caótico

Código A.8: Multiplexor para control de condición inicial y retroalimentación.

```

-- Engineer: Ciro Fabian Bermudez Marquez
-- Date: 14/06/2022
-- Design Name: mux_ic.vdl
-- Description: Multilexer to select between initial condition or feedback
-- Inputs:
--     X0 : Initial condition
--     Xn_1 : feedback
--     SEL : Mux selector
-- Outputs:
--     Xn : Mux output
-----

library ieee;
use ieee.std_logic_1164.all;

entity mux_ic is
    generic( n : integer := 64);
    port(
        X0 : in std_logic_vector(n-1 downto 0);
        Xn_1: in std_logic_vector(n-1 downto 0);
        SEL : in std_logic;
        Xn : out std_logic_vector(n-1 downto 0)
    );
end;

architecture arch of mux_ic is
begin
    Xn <= X0 when SEL = '0' else Xn_1;
end;

```

Código A.9: Sumador genérico compatible con punto fijo.

```

-- Engineer: Ciro Fabian Bermudez Marquez
-- Date: 14/06/2022
-- Design Name: adder.vdl
-- Description: Fixed point adder
-- Inputs:
--     X : First operand
--     Y : Second operand
-- Outputs:
--     A : Output of adder
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
    generic( n : integer := 64 );
    port(
        X,Y : in std_logic_vector(n-1 downto 0);
        A : out std_logic_vector(n-1 downto 0)
    );
end;

architecture arch of adder is
begin
    A <= std_logic_vector( signed(X) + signed(Y) );
end;

```

Código A.10: ROM para almacenar parámetros en punto fijo del mapa caótico.

```

-----  

-- Engineer: Ciro Fabian Bermudez Marquez  

-- Date: 14/06/2022  

-- Design Name: rom_cm.vdl  

-- Description: ROM for chaotic map  

--  

--      Outputs:  

--          an : All chaotic map parameters  

-----  

library ieee;  

use ieee.std_logic_1164.all;  

entity rom_cm is  

    generic(n : integer := 64);  

    port(  

        a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12 : out std_logic_vector(n-1 downto 0)  

    );  

end;  

architecture arch of rom_cm is  

begin  

    a1 <= "1111011001100110011001100110011001100110011001100110011001100110011000000"; -- -0.60  

    a2 <= "11111100110011001100110011001100110011001100110011001100110011000000"; -- -0.10  

    a3 <= "0000100110011001100110011001100110011001100110011001100110011000000"; -- 1.10  

    a4 <= "0000001100110011001100110011001100110011001100110011001100110011000000"; -- 0.20  

    a5 <= "1111001100110011001100110011001100110011001100110011001100110011000000"; -- -0.80  

    a6 <= "0000100110011001100110011001100110011001100110011001100110011000000"; -- 0.60  

    a7 <= "111101001100110011001100110011001100110011001100110011001100110011000000"; -- -0.70  

    a8 <= "00001011001100110011001100110011001100110011001100110011001100110011000000"; -- 0.70  

    a9 <= "0000101100110011001100110011001100110011001100110011001100110011000000"; -- 0.70  

    a10 <= "00000100110011001100110011001100110011001100110011001100110011000000"; -- 0.30  

    a11 <= "00000100110011001100110011001100110011001100110011001100110011000000"; -- 0.60  

    a12 <= "000011001100110011001100110011001100110011001100110011001100110011000000"; -- 0.90  

end;

```

Código A.11: Multiplicador en punto fijo con truncamiento.

```

-----  

-- Engineer: Ciro Fabian Bermudez Marquez  

-- Date: 14/06/2022  

-- Design Name: mult.vdl  

-- Description: Fixed point multiplier using truncation  

--  

--      Use the following equations to calculate correct values  

--      The format is A(a ,b ) = (3 ,60)  

--  

--      Ap(ap,bp) = (6 ,120)  

--      left_lim = bp + a = 120 + 3 = 123  

--      right_lim = bp - b = 120 - 60 = 60  

--  

--  

--      Inputs:  

--          X : First operand  

--          Y : Second operand  

--  

--      Outputs:  

--          M : Output of multiplication  

--  

--      Signals:  

--          temp : Partial multiplication  

-----  

library ieee;  

use ieee.std_logic_1164.all;  

use ieee.numeric_std.all;  

entity mult is  

    generic( n : integer := 64);  

    port(  

        X,Y : in std_logic_vector(n-1 downto 0);  

        M : out std_logic_vector(n-1 downto 0)
    );
end;  

architecture arch of mult is  

    signal temp : std_logic_vector(2*n-1 downto 0);
begin  

    temp <= std_logic_vector(signed(X)*signed(Y));  

    M <= temp(123 downto 60);
end;

```

Código A.12: Flip-Flop con habilitación.

```

-----  

-- Engineer: Ciro Fabian Bermudez Marquez  

-- Date: 14/06/2022  

-- Design Name: ff_hab.vdl  

-- Description: Flip-Flop with enable  

--  

--      Inputs:  

--          RST : Reset  

--          CLK : Clock  

--          EN : Enable  

--          D : Data  

--  

--      Outputs:  

--          Q : Flip-Flop output  

--  

--      Signals:  

--          Qp : FF_output - Present State
-----
```

```
-- Qn : FF_input - Next State
-----
library ieee;
use ieee.std_logic_1164.all;

entity ff_hab is
    generic(n : integer := 64);
    port(
        RST : in std_logic;
        CLK : in std_logic;
        EN : in std_logic;
        D : in std_logic_vector(n-1 downto 0);
        Q : out std_logic_vector(n-1 downto 0)
    );
end;

architecture arch of ff_hab is
    signal Qn, Qp : std_logic_vector(n-1 downto 0);
begin
    Qn <= Qp when EN = '0' else D;
    process(RST, CLK)
    begin
        if RST = '1' then
            Qp <= (others => '0');
        elsif rising_edge(CLK) then
            Qp <= Qn;
        end if;
    end process;
    Q <= Qp;
end;
```

Código A.13: Máquina de estados para control de las iteraciones del mapa caótico.

```
-- Engineer: Ciro Fabian Bermudez Marquez
-- Date: 14/06/2022
-- Design Name: fsm_cm.vdl
-- Description: Finite state machine to chaotic map operation.
--              There are 4 states then 2 bits are necessary.
--              Default/Wait, Enable IC, Wait/Change SEL, Enable
-- Inputs:
--     RST : Reset
--     CLK : CLK
--     START : Start FSM
-- Outputs:
--     EN : Enable output FF
--     SEL : Mux selector
-- Signals:
--     Qp : FF_output - Present State
--     Qn : FF_input - Next State
-----
library ieee;
use ieee.std_logic_1164.all;

entity fsm_cm is
    port(
        RST : in std_logic;
        CLK : in std_logic;
        START : in std_logic;
        EN : out std_logic;
        SEL : out std_logic
    );
end;

architecture arch of fsm_cm is
    signal Qp, Qn : std_logic_vector(1 downto 0);
begin
    process(Qp, START)
    begin
        case Qp is
            when "00" => SEL <= '0'; EN <= '0'; -- Default/Wait
            if START = '1' then
                Qn <= "01";
            else
                Qn <= Qp;
            end if;
        when "01" => SEL <= '0'; EN <= '1'; -- Enable
        when "10" => SEL <= '1'; EN <= '0'; -- Wait/Change SEL
            if START = '1' then
                Qn <= "11";
            else
                Qn <= Qp;
            end if;
        when "11" => SEL <= '1'; EN <= '1'; -- Enable
        when others => SEL <= '0'; EN <= '0';
        Qn <= "00";
        end case;
    end process;
    process(RST, CLK)
    begin
```

```

if RST = '1' then
  Qp <= (others => '0');
elsif rising_edge(CLK) then
  Qp <= Qn;
end if;
end process;

end;

```

Código A.14: Descripción completa del mapa caótico.

```

-----
-- Engineer: Ciro Fabian Bermudez Marquez
-- Date: 14/06/2022
-- Design Name: chaotic_map.vhd
-- Description: Implementation of Sprott chaotic map
-- Inputs:
--   RST : Reset
--   CLK : CLK
--   START : Start iterations
-- Outputs:
--   X : X variable
--   Y : X variable
-----

library ieee;
use ieee.std_logic_1164.all;

entity chaotic_map is
  generic( n : integer := 64);
  port(
    RST : in std_logic;
    CLK : in std_logic;
    START : in std_logic;
    X0,Y0 : in std_logic_vector(n-1 downto 0);
    X,Y : out std_logic_vector(n-1 downto 0)
  );
end;

architecture arch of chaotic_map is
  signal xn,yn,xn_retro,yn_retro : std_logic_vector(n-1 downto 0);
  signal sel,en : std_logic;
  signal an1,an2,an3,an4,an5,an6,an7,an8,an9,an10,an11,an12 : std_logic_vector(n-1 downto 0);
  signal m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,m11 : std_logic_vector(n-1 downto 0);
  signal a1,a2,a3,a4,a5,a6,a7,a8,a9,a10 : std_logic_vector(n-1 downto 0);
begin
  -- Mux
  mux_x : entity work.mux_ic generic map(n => 64) port map(X0,xn_retro,sel,xn);
  mux_y : entity work.mux_ic generic map(n => 64) port map(Y0,yn_retro,sel,yn);

  -- ROM
  mod_rom : entity work.rom_cm generic map(n => 64) port map(an1,an2,an3,an4,an5,an6,an7,an8,an9,an10,an11,an12);

  -- Multipliers
  mult_m1 : entity work.mult generic map(n => 64) port map(an3,xn,m1);
  mult_m2 : entity work.mult generic map(n => 64) port map(an6,yn,m2);
  mult_m3 : entity work.mult generic map(n => 64) port map(an9,xn,m3);
  mult_m4 : entity work.mult generic map(n => 64) port map(an12,yn,m4);
  mult_m5 : entity work.mult generic map(n => 64) port map(xn,yn,m5);
  mult_m6 : entity work.mult generic map(n => 64) port map(an4,m5,m6);
  mult_m7 : entity work.mult generic map(n => 64) port map(m5,an10,m7);
  mult_m8 : entity work.mult generic map(n => 64) port map(a1,xn,m8);
  mult_m9 : entity work.mult generic map(n => 64) port map(a2,yn,m9);
  mult_m10 : entity work.mult generic map(n => 64) port map(a3,xn,m10);
  mult_m11 : entity work.mult generic map(n => 64) port map(a4,yn,m11);

  -- Adders
  adder_a1 : entity work.adder generic map(n => 64) port map(m1,an2,a1);
  adder_a2 : entity work.adder generic map(n => 64) port map(m2,an5,a2);
  adder_a3 : entity work.adder generic map(n => 64) port map(m3,an8,a3);
  adder_a4 : entity work.adder generic map(n => 64) port map(m4,an11,a4);
  adder_a5 : entity work.adder generic map(n => 64) port map(an1,m8,a5);
  adder_a6 : entity work.adder generic map(n => 64) port map(m9,m6,a6);
  adder_a7 : entity work.adder generic map(n => 64) port map(an7,m10,a7);
  adder_a8 : entity work.adder generic map(n => 64) port map(m11,m7,a8);
  adder_a9 : entity work.adder generic map(n => 64) port map(a5,a6,a9);
  adder_a10 : entity work.adder generic map(n => 64) port map(a7,a8,a10);

  -- Registers
  ff_xn : entity work.ff_hab generic map(n => 64) port map(RST,CLK,en,a9,xn_retro);
  ff_yn : entity work.ff_hab generic map(n => 64) port map(RST,CLK,en,a10,yn_retro);

  -- Control Unit
  cu_cm : entity work.fsm_cm
    port map(RST,CLK,START,en,sel);

  X <= xn_retro;
  Y <= yn_retro;
end;

```

Bibliografía

- [1] D. E. Knuth, *The art of computer programming*. Addison-Wesley, 2014.
- [2] O. Petura, U. Mureddu, N. Bochard, V. Fischer, and L. Bossuet, “A survey of AIS-20/31 compliant TRNG cores suitable for FPGA devices,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, aug 2016.
- [3] B. Badrignans, J. L. Danger, V. Fischer, G. Gogniat, and L. Torres, eds., *Security Trends for FPGAS*. Springer Netherlands, 2011.
- [4] B. Jun and P. Kocher, “The intel random number generator,” *Cryptography Res. Inc., San Francisco, CA, USA*, 1999.
- [5] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, “Recommendation for the entropy sources used for random bit generation,” tech. rep., jan 2018.
- [6] W. Killmann and W. Schindler, “A proposal for: Functionality classesfor random number generators, version 2.0,” 2011.
- [7] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, N. Heckert, J. Dray, S. Vo, and L. Bassham, “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” *Software and SP800-22rev1a.pdf file, last date checked: Feb 28th, 2020*, 2010.
- [8] W. Schindler and W. Killmann, “Evaluation criteria for true (physical) random number generators used in cryptographic applications,” in *Cryptographic Hardware and Embedded Systems - CHES 2002*, pp. 431–449, Springer Berlin Heidelberg, 2003.
- [9] M. Baudet, D. Lubicz, J. Micolod, and A. Tassiaux, “On the security of oscillator-based random number generators,” vol. 24, pp. 398–425, oct 2010.

- [10] P. Kohlbrenner and K. Gaj, “An embedded true random number generator for FPGAs,” in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, ACM, feb 2004.
- [11] M. Varchola and M. Drutarovsky, “New high entropy element for FPGA based true random number generators,” pp. 351–365, Springer Berlin Heidelberg, 2010.
- [12] A. Cherkaoui, V. Fischer, A. Aubert, and L. Fesquet, “A self-timed ring based true random number generator,” in *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*, IEEE, may 2013.
- [13] V. Fischer and M. Drutarovský, “True random number generator embedded in reconfigurable hardware,” in *Cryptographic Hardware and Embedded Systems - CHES 2002*, pp. 415–430, Springer Berlin Heidelberg, 2003.
- [14] X. Wang, H. Liang, Y. Wang, L. Yao, Y. Guo, M. Yi, Z. Huang, H. Qi, and Y. Lu, “High-throughput portable true random number generator based on jitter-latch structure,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, pp. 741–750, feb 2021.
- [15] A. Peetermans, V. Rožić, and I. Verbauwhede, “Design and analysis of configurable ring oscillators for true random number generation based on coherent sampling,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 14, pp. 1–20, jun 2021.
- [16] N. N. Anandakumar, S. K. Sanadhya, and M. S. Hashmi, “FPGA-based true random number generation using programmable delays in oscillator-rings,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, pp. 570–574, mar 2020.
- [17] T. L. Liao, P. Y. Wan, and J.-J. Yan, “Design and synchronization of chaos-based true random number generators and its FPGA implementation,” *IEEE Access*, vol. 10, pp. 8279–8286, 2022.
- [18] S. Vaidyanathan, A. Sambas, B. Abd-El-Atty, A. A. A. El-Latif, E. Tlelo-Cuautle, O. Guillen-Fernandez, M. Mamat, M. A. Mohamed, M. Alcin, M. Tuna, I. Pehlivan, I. Koyuncu, and M. A. H. Ibrahim, “A 5-d multi-stable hyperchaotic two-disk dynamo system with no equilibrium point: Circuit design, FPGA realization and applications to TRNGs and image encryption,” *IEEE Access*, vol. 9, pp. 81352–81369, 2021.
- [19] L. G. de la Fraga, E. Torres-Pérez, E. Tlelo-Cuautle, and C. Mancillas-López, “Hardware implementation of pseudo-random number generators based on chaotic maps,” *Nonlinear Dynamics*, vol. 90, pp. 1661–1670, aug 2017.

- [20] K.-W. Wong, B. S.-H. Kwok, and W.-S. Law, “A fast image encryption scheme based on chaotic standard map,” *Physics Letters A*, vol. 372, pp. 2645–2652, apr 2008.
- [21] O. M. Al-Hazaimeh, M. F. Al-Jamal, N. Alhindawi, and A. Omari, “Image encryption algorithm based on lorenz chaotic map with dynamic secret keys,” *Neural Computing and Applications*, vol. 31, pp. 2395–2405, aug 2017.
- [22] Z. Liu, Y. Wang, Y. Zhao, and L. Y. Zhang, “A stream cipher algorithm based on 2d coupled map lattice and partitioned cellular automata,” *Nonlinear Dynamics*, vol. 101, pp. 1383–1396, jul 2020.
- [23] J. Li and H. Liu, “Colour image encryption based on advanced encryption standard algorithm with two-dimensional chaotic map,” *IET Information Security*, vol. 7, pp. 265–270, dec 2013.
- [24] R. Sivaraman, S. Rajagopalan, J. B. B. Rayappan, and R. Amirtharajan, “Ring oscillator as confusion – diffusion agent: a complete TRNG drove image security,” *IET Image Processing*, vol. 14, pp. 2987–2997, oct 2020.
- [25] N. Pareek, V. Patidar, and K. Sud, “Image encryption using chaotic logistic map,” *Image and Vision Computing*, vol. 24, pp. 926–934, sep 2006.
- [26] R. Kadir, R. Shahril, and M. A. Maarof, “A modified image encryption scheme based on 2d chaotic map,” in *International Conference on Computer and Communication Engineering (ICCCE'10)*, IEEE, may 2010.
- [27] W. Liu, K. Sun, and C. Zhu, “A fast image encryption algorithm based on chaotic map,” *Optics and Lasers in Engineering*, vol. 84, pp. 26–36, sep 2016.
- [28] S. Vaidyanathan, A. Akgul, S. Kaçar, and U. Çavuşoğlu, “A new 4-d chaotic hyperjerk system, its synchronization, circuit design and applications in RNG, image encryption and chaos-based steganography,” *The European Physical Journal Plus*, vol. 133, feb 2018.
- [29] E. García-Guerrero, E. Inzunza-González, O. López-Bonilla, J. Cárdenas-Valdez, and E. Tlelo-Cuautle, “Randomness improvement of chaotic maps for image encryption in a wireless communication scheme using PIC-microcontroller via zigbee channels,” *Chaos, Solitons and Fractals*, vol. 133, p. 109646, apr 2020.
- [30] C. García-Grimaldo and E. Campos, “Chaotic features of a class of discrete maps without fixed points,” *International Journal of Bifurcation and Chaos*, vol. 31, oct 2021.

- [31] B. M. Hernandez-Morales, S. Diaz-Santiago, and C. Mancillas-Lopez, “Co-design for generation of large random sequences on zynq FPGA,” *IEEE Embedded Systems Letters*, pp. 1–1, 2022.
- [32] B. Valtchanov, A. Aubert, F. Bernard, and V. Fischer, “Modeling and observing the jitter in ring oscillators implemented in FPGAs,” in *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, IEEE, apr 2008.
- [33] O. Petura, *True random number generators for cryptography : Design, securing and evaluation*. Theses, Université de Lyon, Oct. 2019.
- [34] B. Sunar, W. Martin, and D. Stinson, “A provably secure true random number generator with built-in tolerance to active attacks,” *IEEE Transactions on Computers*, vol. 56, pp. 109–119, jan 2007.
- [35] V. Rozic, B. Yang, W. Dehaene, and I. Verbauwhede, “Highly efficient entropy extraction for true random number generators on FPGAs,” in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, jun 2015.
- [36] Y. Zhang, J. Jiang, Q. Wang, and N. Guan, “A self-timed ring based true random number generator on FPGA,” in *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, IEEE, oct 2018.
- [37] L. Reyneri, D. D. Corso, and B. Sacco, “Oscillatory metastability in homogeneous and inhomogeneous flip-flops,” *IEEE Journal of Solid-State Circuits*, vol. 25, no. 1, pp. 254–264, 1990.
- [38] V. Fischer and D. Lubicz, “Embedded Evaluation of Randomness in Oscillator Based Elementary TRNG,” in *Workshop on Cryptographic Hardware and Embedded Systems 2014 (CHES 2014)*, (Busan, South Korea), p. 16 p., Sept. 2014.
- [39] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, “Advanced encryption standard (AES),” tech. rep., nov 2001.
- [40] V. R. Joan Daemen, *The Design of Rijndael*. Springer-Verlag GmbH, May 2020.
- [41] Q. H. Dang, “Secure hash standard,” tech. rep., July 2015.
- [42] S. Choi, Y. Shin, and H. Yoo, “Analysis of ring-oscillator-based true random number generator on FPGAs,” in *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, IEEE, jan 2021.
- [43] N. Bochard, F. Bernard, V. Fischer, and B. Valtchanov, “True-randomness and pseudo-randomness in ring oscillator-based true random number generators,” *International Journal of Reconfigurable Computing*, vol. 2010, pp. 1–13, 2010.

- [44] K. Wold and C. H. Tan, “Analysis and enhancement of random number generator in FPGA based on oscillator rings,” in *2008 International Conference on Reconfigurable Computing and FPGAs*, IEEE, dec 2008.
- [45] P. Haddad, V. Fischer, F. Bernard, and J. Nicolai, “A physical approach for stochastic modeling of TERO-based TRNG,” in *Lecture Notes in Computer Science*, pp. 357–372, Springer Berlin Heidelberg, 2015.
- [46] F. Bernard, P. Haddad, V. Fischer, and J. Nicolai, “From physical to stochastic modeling of a TERO-based TRNG,” *Journal of Cryptology*, vol. 32, pp. 435–458, mar 2018.
- [47] V. Fischer, F. Bernard, and N. Bochard, “Modern random number generator design – case study on a secured PLL-based TRNG,” *it - Information Technology*, vol. 61, pp. 3–13, jan 2019.
- [48] S. H. Strogatz, *Nonlinear dynamics and Chaos*. Addison-Wesley Pub., 1994.
- [49] J. C. Sprott, *Chaos and time-series analysis*. Oxford University Press, 2003.
- [50] T. S. Parker and L. Chua, *Practical Numerical Algorithms for Chaotic Systems*. Springer London, Limited, 2012.
- [51] E. Tlelo-Cuautle, J. de Jesús Rangel-Magdaleno, and L. G. de la Fraga, *Engineering Applications of FPGAs Chaotic Systems, Artificial Neural Networks, Random Number Generators, and Secure Communication Systems*. Springer London, Limited, 2016.
- [52] J. Sprott, “Automatic generation of strange attractors,” *Computers & Graphics*, vol. 17, pp. 325–332, may 1993.
- [53] L. D. Micco, M. Antonelli, and H. Larrondo, “Stochastic degradation of the fixed-point version of 2d-chaotic maps,” *Chaos, Solitons and Fractals*, vol. 104, pp. 477–484, nov 2017.
- [54] L. G. D. la Fraga, C. Mancillas-López, and E. Tlelo-Cuautle, “Designing an authenticated hash function with a 2d chaotic map,” *Nonlinear Dynamics*, vol. 104, pp. 4569–4580, may 2021.
- [55] J. Thong and N. Nicolici, “A novel optimal single constant multiplication algorithm,” in *Proceedings of the 47th Design Automation Conference*, ACM, jun 2010.
- [56] O. Guillén-Fernández, A. D. Pano-Azucena, E. Tlelo-Cuautle, and A. Silva-Juárez, *Analog/Digital Implementation of Fractional Order Chaotic Circuits and Applications*. Springer-Verlag GmbH, Nov. 2019.

- [57] J. Thong and N. Nicolici, “An optimal and practical approach to single constant multiplication,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 1373–1386, sep 2011.
- [58] M. Laban, Milos Drutarovsky, V. Fischer, and M. Varchola, “Platform for testing and evaluation of puf and trng implementations in fpgas,” 2016.