



# Fundamentos Teóricos de Informática

Generador pseudoaleatorio XOR-SHIFT  
utilizando una Máquina de Turing

**Autor:** Agustín Palma

**Fecha:** 20 de noviembre de 2025

**Contacto:** [agpalma002@gmail.com](mailto:agpalma002@gmail.com)

**Cátedra:** Ing. Leonardo Moreno, Dr. Leonardo Morales

<https://github.com/agustxnpm/turing-xor-shift-simulator>



# 1. Introducción

El presente trabajo aborda la implementación y el estudio de un **Generador de Números Pseudoaleatorios (PRNG)** basado en el algoritmo **XOR-Shift**, utilizando como modelo de cómputo fundamental la **Máquina de Turing (MT)**. El objetivo principal es demostrar que un proceso inherentemente determinista, como es la ejecución paso a paso de una MT, es capaz de producir secuencias que simulan la aleatoriedad, un concepto central en la criptografía y la simulación por computadora.

## Marco Teórico: La Máquina de Turing como Modelo de Computación

La Máquina de Turing, introducida por Alan Turing en 1936, constituye el modelo teórico de cómputo más poderoso y aceptado dentro de los fundamentos teóricos de la informática. Una MT es un autómata con un conjunto finito de estados ( $Q$ ), una cinta infinita utilizada como memoria, y un cabezal de lectura/escritura.

A diferencia de las arquitecturas de von Neumann (donde la memoria es de acceso aleatorio), la MT opera de forma estrictamente secuencial, realizando solo tres acciones posibles en cada transición: **escribir un símbolo**, **cambiar de estado** y **mover el cabezal** (izquierda o derecha). Simular un algoritmo moderno de registros (como el XOR-Shift) en este modelo impone desafíos que fuerzan el diseño a utilizar la cinta de manera estructurada, segmentando la memoria en áreas lógicas para la semilla, el área de cálculo (**scratchpad**) y el **historial**.

## El Algoritmo XOR-Shift

El XOR-Shift es un algoritmo eficiente y popular en la generación de números pseudoaleatorios debido a su simplicidad y velocidad, que generalmente exhibe largos períodos de ciclo. El algoritmo se basa en aplicar operaciones lógicas XOR y desplazamientos de bits (Shift), operando sobre una semilla binaria ( $x$ ) según la fórmula:

$$x_{i+1} = (x_i \oplus (x_i \ll a)) \oplus (x_{\text{temp}} \gg b) \oplus (x_{\text{temp2}} \ll c)$$



donde  $a, b$  y  $c$  son constantes de desplazamiento.

La simulación de esta fórmula en la MT requiere la descomposición de la operación compleja en un conjunto de subrutinas básicas que se ejecutan secuencialmente:

1. **Operación lógica:** La función XOR se implementa mediante la lógica de estados de la MT, que viaja entre dos bits y aplica la suma módulo 2.
2. **Operación de desplazamiento ( $\ll$  o  $\gg$ ):** Se simula mediante el **movimiento controlado del cabezal** a una posición relativa ( $i +$ - desplazamiento) para buscar el segundo operando.
3. **Secuencialidad de operaciones:** El uso de las subrutinas **actualizar semilla** después de cada XOR parcial es crucial, ya que asegura que la siguiente operación de desplazamiento lea el *resultado intermedio* de la semilla, manteniendo la fidelidad al algoritmo XOR-Shift real.

## Implementación y Orquestación

La implementación se realiza mediante un diseño modular donde el **orquestador** actúa como el programa principal. Este Orquestador es, en sí mismo, un gran autómata que une los estados finales de una subrutina con los estados iniciales de la siguiente (método conocido como **Encadenamiento de Estados** o *State Chaining*).

Las principales subrutinas implementadas son:

- **COPY y ACTUALIZAR SEMILLA:** Rutinas de transcripción necesarias para mover los datos entre la semilla (fuente de referencia para el Shift) y el scratchpad (área de trabajo).
- **XOR con desplazamiento:** Rutinas que ejecutan el cálculo bit a bit con movimiento relativo del cabezal.
- **Detección de ciclo:** Rutina de  $O(n^2)$  que registra el historial de las semillas generadas y compara el nuevo valor con todos los previos, siendo la condición de parada del autómata.



Todo el desarrollo se ha realizado en **TypeScript** dentro del *framework Angular*, lo cual permite no solo la correcta codificación de las funciones de transición, sino también la creación de un simulador visual interactivo que permite observar la evolución de la cinta y el cabezal paso a paso, proporcionando una herramienta para la depuración y comprensión del comportamiento del autómata.

## 2. Metodología de implementación

La metodología de este trabajo se basó en el principio de **descomposición funcional** sobre la Máquina de Turing (MT). Dado que la MT es inherentemente un autómata secuencial de bajo nivel, la implementación del algoritmo XOR-Shift (un algoritmo complejo de alto nivel) se logró mediante la construcción de un lenguaje de **subrutinas** (o macros) que, al ser interconectadas por un **Orquestador**, simulan la funcionalidad de un procesador y su registro de memoria.

### Mapeo de la cinta

El primer desafío fue mapear los registros necesarios del algoritmo XOR-Shift a la única estructura de memoria disponible: la **cinta unidimensional** de la MT. La cinta se diseñó con tres secciones lógicas fundamentales, separadas por símbolos del alfabeto de la MT ( $\Gamma$ ):

1. **Scratchpad (área de trabajo):** Ubicada a la izquierda del separador  $@$ . Utilizada para realizar las operaciones XOR y como buffer de almacenamiento temporal del resultado ( $x_{temp}$ ).
2. **Registro semilla (estado  $x_t$ ):** Ubicada entre los separadores  $@$  y  $#$ . Representa el valor binario ( $w$ ) del que se leerán los bits para aplicar los desplazamientos (**Shift**).
3. **Historial:** Ubicada a la derecha del separador  $#$ . Almacena la traza completa de las semillas generadas ( $x_0, x_1, x_2, \dots$ ), cada una separada por un separador  $$$ , crucial para la detección de ciclos.

$< scratchpad > @ < semilla > # < historial > \$ <... > \$ <... > \$ \dots$

### La función de transición modular ( $\delta$ )



En lugar de construir una función de transición monolítica  $\delta$ , se empleó la **Composición de Autómatas**. Cada subrutina genera un subconjunto de estados y transiciones que se unen a un mapa de transiciones global. La conexión entre estos módulos se realiza mediante el **Encadenamiento de Estados**: el estado final ( $q_{fin}$ ) de la subrutina  $M_i$  se define como el estado inicial ( $q_{ini}$ ) de la subrutina  $M_{i+1}$ . Para evitar colisiones en el conjunto global de estados ( $Q$ ), se utilizó la técnica de **Namespacing** (prefijado de estados) en TypeScript.

## Implementación granular de las subrutinas

### 1. Subrutina COPY

Esta subrutina constituye el mecanismo fundamental de **persistencia y transferencia de datos** dentro de la máquina. Su función lógica es duplicar la cadena binaria almacenada en el segmento de la **semilla** hacia el segmento del **scratchpad**, preservando la integridad de la fuente original para futuros cálculos.

### Fundamento teórico: La máquina de copiado con marcadores

Desde la perspectiva de la teoría de autómatas, copiar una cadena  $w$  de longitud  $n$  en una cinta de una sola pista requiere un enfoque iterativo destructivo-restaurativo. Dado que la cabeza lectora no posee memoria interna para almacenar la cadena completa (solo puede "recordar" un símbolo finito a través de su estado actual), el algoritmo sigue estos pasos lógicos:

1. **Lectura y marcado:** El autómata lee el primer símbolo no procesado de la cadena origen. Para evitar procesarlo nuevamente en la siguiente iteración, lo reemplaza temporalmente por un **símbolo marcador auxiliar** (en la implementación:  $0 \rightarrow X, 1 \rightarrow Y$ ).
2. **Transporte de estado:** El autómata cambia a un estado específico que "carga" la información del bit leído y se desplaza hacia la zona destino.
3. **Escritura:** Al encontrar la primera posición libre en el destino (scratchpad), escribe el bit original.



4. **Retorno y repetición:** El cabezal regresa a la zona origen, busca el siguiente bit sin marcar y repite el proceso.
5. **Restauración (Limpieza):** Una vez copiados todos los bits, el autómata realiza un barrido final sobre la cadena origen para restaurar los marcadores (**X**, **Y**) a sus valores originales (**0**, **1**), dejando la cinta en un estado consistente.

## Implementación técnica y máquina de estados

En la implementación en TypeScript, esta lógica se descompone en una serie de fases secuenciales gestionadas por el mapa de transiciones. A diferencia de un diagrama teórico simplificado, la implementación real debe manejar la navegación bidireccional entre zonas separadas por delimitadores (@ y #).

- **Fases de navegación (0 y 1):** Se implementaron estados de tránsito (`q_copy_go_to_end`) que mueven el cabezal hasta el delimitador derecho **#**. Esto establece el punto de partida para un barrido de derecha a izquierda, optimizando la detección de bits pendientes.
- **Fase de selección (2):** El estado `q_copy_find_next_bit` escanea la semilla. Al encontrar un **0** o **1**, transiciona a los estados de transporte `q_copy_navigate_0` o `q_copy_navigate_1`, aplicando simultáneamente la marca (**X** o **Y**) en la cinta.
- **Fase de escritura (3 y 4):** Los estados de navegación cruzan el delimitador central (@) hacia la izquierda hasta encontrar el primer espacio en blanco (\_) en el scratchpad, donde depositan el bit transportado.
- **Fase de restauración (5):** Crítica para la reusabilidad del algoritmo. El estado `q_copy_cleanup_start` recorre la semilla transformando **X** → **0** y **Y** → **1**. Sin esta fase, la semilla quedaría "sucia" e inutilizable para las operaciones de desplazamiento (Shift) subsiguientes.

La subrutina finaliza dejando el cabezal posicionado estratégicamente en el **bit más significativo** del scratchpad, cumpliendo así con la precondition necesaria para que la siguiente subrutina (**XOR**) comience su ejecución sin ciclos de búsqueda adicionales.



## 2. Subrutina XOR con desplazamiento

Esta subrutina actúa como la **Unidad Lógica Aritmética (ALU)** del sistema. Es la encargada de ejecutar el núcleo matemático del algoritmo: la operación binaria compuesta

$x_{scratch} \leftarrow x_{scratch} \oplus (x_{semilla} \ll k)$ . A diferencia de la subrutina de copiado, que es una operación de transferencia, esta es una operación de **transformación in-situ**.

### Fundamento teórico: Aritmética posicional en la cinta

En una arquitectura clásica (Von Neumann), un desplazamiento de bits se realiza en paralelo dentro de un registro de hardware. En el modelo secuencial de Turing, el concepto de "desplazamiento" se traduce literalmente en **distancia física sobre la cinta**.

Para calcular el nuevo valor del bit en la posición  $i$ , el autómata debe resolver la ecuación

$$r_i = a_i \oplus b_{i+k}, \text{ donde:}$$

- $a_i$  es el bit local en el scratchpad
- $b_{i+k}$  es el bit remoto en la semilla, ubicado a una distancia relativa  $k$

La operación requiere un autómata que pueda "sostener" el valor de  $a_i$  en su memoria interna (estado), viajar una distancia calculada, leer  $b_{i+k}$ , aplicar la tabla de verdad XOR y regresar para sobrescribir  $a_i$ .

### Implementación técnica: cálculo bit a bit

La implementación en TypeScript construye dinámicamente un grafo de estados complejo que itera sobre cada bit de la palabra (de  $i = 0$  a 5). El ciclo de vida de una operación unitaria para un solo bit se desglosa en las siguientes fases:

1. **Lectura del primer operando (Fase 1):** El cabezal lee el bit  $i$  del scratchpad. Para no perder la posición durante el cálculo, marca este bit con un símbolo temporal (**A** si era **0**, **B** si era **1**). Esta marca cumple una doble función: guarda el valor del primer operando y sirve como punto de retorno.



2. **Navegación y desplazamiento (Fases 2 y 3):** El autómata viaja hacia la derecha buscando el delimitador `@`. Una vez en la semilla, aplica el **desplazamiento** simulando un bucle de movimiento.
  - Si el desplazamiento indica una posición válida dentro de la semilla, lee el segundo operando.
  - Si el desplazamiento cae fuera de los límites de la palabra (ej. índice negativo o  $> 5$ ), la lógica del autómata asume un valor de `0` (padding estándar para desplazamientos lógicos), garantizando la robustez ante bordes.
3. **Cálculo y retorno (Fase 4):** En este punto, el autómata "conoce" ambos valores (el que traía marcado desde el inicio y el que acaba de leer). El estado de retorno codifica el resultado de la operación XOR:
  - Estado `RETURN_A0_B1` implícitamente transporta el resultado `1` ( $0 \oplus 1$ ).
  - Estado `RETURN_A1_B1` implícitamente transporta el resultado `0` ( $1 \oplus 1$ ).

El cabezal viaja a la izquierda buscando su marca original (`A` o `B`).
4. **Escritura del resultado (Fase 5):** Al encontrar la marca en el scratchpad, la sobrescribe con el resultado calculado, completando la transformación para ese bit y avanzando a  $i + 1$ .

**Aislamiento de estados (namespacing):** Dado que esta subrutina se invoca tres veces en cada iteración del algoritmo principal (una para cada constante  $a$ ,  $b$ ,  $c$ ), es imperativo evitar colisiones en el espacio de estados  $Q$ . Se implementó un mecanismo de **Namespacing** que prefija dinámicamente todos los estados internos (ej. `op1_q_xor...`, `op2_q_xor...`), permitiendo que coexistan múltiples instancias de la misma lógica matemática en la función de transición global sin interferencias.

### 3. Subrutina actualizar semilla

La subrutina **actualizar semilla** simula la operación de **asignación de variable** o **escritura de registro** ( $x \leftarrow y$ ).

#### Fundamento teórico: Preservación del estado secuencial



En el modelo de la Máquina de Turing, el valor actual de la semilla  $x$  se encuentra en el scratchpad (el área de trabajo volátil) después de cada operación XOR. La subrutina tiene el propósito de propagar este resultado parcial o final a la zona de la **semilla** (el registro principal) para dos fines:

1. **Garantizar el encadenamiento lógico:** Como se discutió en la introducción, para que las operaciones  $x \oplus (x \gg b)$  y  $x \oplus (x \ll c)$  utilicen el valor transformado por los pasos previos, la semilla debe contener la versión más reciente del dato. Si este paso fuera omitido, las subsecuentes operaciones de desplazamiento (**Shift**) leerían el valor original  $x_0$  y no el valor temporal  $x_{temp}$ .
2. **Preparación para detección:** Asegura que el número que se va a registrar en el scratchpad y se va a comparar esté en su posición canónica (entre @ y #)

## Implementación técnica: Transcripción con sobreescritura

La lógica de esta subrutina es similar a la rutina de copiado (**COPY**), pero con dos diferencias clave: la dirección del flujo de datos es inversa (scratchpad  $\rightarrow$  semilla) y el destino de la escritura **no es un espacio en blanco**, sino una posición ya ocupada que debe ser sobreescrita.

1. **Bucle de transcripción:** La rutina itera a través de cada bit de la palabra (de  $i = 0$  a 5).
2. **Lectura de la fuente:** El cabezal lee y marca el bit  $i$  del scratchpad con una marca temporal (**A** o **B**).
3. **Viaje y posicionamiento:** Navega hacia la semilla y realiza un movimiento de **desplazamiento** de  $i$  pasos hacia la derecha desde el delimitador @. Este paso asegura que el cabezal se posicione exactamente sobre el bit  $i$  de la semilla.
4. **Sobreescritura:** Al llegar a la posición  $i$  de la semilla, el autómata **escribe el bit transportado sobre el valor existente** (ej.  $semilla[i] \leftarrow scratchpad[i]$ ). Esta es una operación de asignación destructiva.
5. **Retorno y limpieza:** El cabezal regresa a la marca temporal (**A** o **B**) en el scratchpad para desmarcarlo (restaurarlo a 0 o 1) y avanzar al siguiente bit  $i + 1$

**Uso de namespacing:** Al igual que en la subrutina XOR, la subrutina actualizar semilla es invocada múltiples veces dentro del ciclo principal (después de XOR A, XOR B, y XOR C).



El uso de **Namespacing** (ej. `upd1_`, `upd2_`, `upd3_`) garantiza que cada instancia de la subrutina genere un conjunto de estados único, evitando el conflicto que podría llevar a que los pasos de una actualización se mezclen con los pasos de otra.

## 4. Subrutina de detección de ciclo

Esta subrutina es la **condición de parada** de la Máquina de Turing, resolviendo la restricción fundamental de que un Generador de Números Pseudoaleatorios (PRNG) es, por definición, determinista y cíclico. La tarea de la subrutina es determinar si el valor de la semilla generado en la iteración actual ( $x_i$ ) ya existe en el historial ( $x_0, x_1, \dots, x_{i-1}$ ).

### Fundamento teórico: El problema de la memoria en tiempo polinomial

La detección de ciclos en una MT requiere que la máquina tenga memoria para la secuencia completa. Dado que la MT no tiene acceso indexado (RAM), la única forma de "recordar" todas las semillas previas es almacenarlas secuencialmente en la cinta (el historial).

- **Complejidad:** La subrutina implementa una estrategia de comparación  $O(N^2)$  (donde  $N$  es el número de iteraciones o números generados). Esto se debe a que, para verificar una nueva semilla ( $x_i$ ), el autómata debe compararla, una por una, con todas las  $i$  entradas previamente registradas en el historial. Esta alta complejidad es inherente a la naturaleza secuencial y no indexada de la memoria de la MT.

### Implementación técnica: Bucle anidado de búsqueda

La subrutina opera en dos fases principales, ambas utilizando la lógica de **copiado con marcadores y navegación dirigida**:

#### 1. Fase de registro (escritura en historial):

- El primer paso de la subrutina es **registrar** la semilla actual ( $x_0$ ) en el historial, copiándola al final del segmento delimitado por el símbolo `#`.



- Este paso es esencial para la lógica del Orquestador: la subrutina es invocada una vez al inicio del programa (para registrar  $x_0$ ) y una vez al final de cada ciclo de cálculo (para registrar  $x_{i+1}$ )

## 2. Fase de detección (bucle anidado de comparación):

- Bucle externo:** El cabezal se posiciona en la primera entrada del historial ( $x_0$ ). Los estados de transición (`q_detect_check_peek_entry`) navegan de separador en separador (\$) para recorrer cada semilla registrada.
- Bucle interno (comparación bit a bit):** Para cada semilla del historial ( $x_j$ ):
  - El cabezal lee un bit de  $x_j$ , lo marca con una *marca temporal* (A o B).
  - Navega a la zona de la semilla ( $x_i$ ) y se desplaza el número de pasos necesarios para leer el bit correspondiente.
  - Una serie de estados de transición resuelven la comparación booleana (ej. ¿0 es igual a 0?).
  - Si la comparación es exitosa, se continúa con el siguiente bit. Si se encuentra el primer bit distinto (Mismatch), la máquina aborta la comparación para  $x_j$ , limpia sus marcas y salta directamente a la siguiente entrada ( $x_{j+1}$ ).

## Condición de parada

La subrutina resuelve la condición final del autómata mediante dos posibles transiciones al finalizar la búsqueda:

- Éxito** (ciclo encontrado): Si el bucle interno de comparación finaliza con un **match total** (los 6 bits coinciden), la máquina transiciona al estado final de aceptación, `q_HALT_CICLO_ENCONTRADO`, cumpliendo con el objetivo de encontrar la repetición.
- Fallo** (continuar): Si el autómata recorre todo el historial sin encontrar ninguna coincidencia, transiciona al estado `q_{inicio\_ciclo}` del Orquestador, lo que reinicia el ciclo de cálculo para generar el siguiente valor ( $x_{i+2}$ )



## Arquitectura del orquestador: Integración del sistema

Si las subrutinas descritas anteriormente representan las instrucciones individuales de un procesador, el **Orquestador** actúa como el **Programa Principal** o la Unidad de Control. No es una subrutina que se ejecute en la cinta, sino una entidad de meta-programación encargada de ensamblar la **Función de Transición Global ( $\delta$ )** de la Máquina de Turing.

## Fundamento teórico: Composición secuencial de autómatas

La construcción de una MT compleja se vuelve inmanejable si se intenta definir  $\delta$  estado por estado. El Orquestador aplica el principio de **Composición Secuencial**, donde un algoritmo complejo  $A$  se define como la concatenación de máquinas más simples  $M_1, M_2, \dots, M_n$ .

Para lograr esto, el Orquestador implementa un mecanismo de **Encadenamiento de Estados** (*State Chaining*). Para cada par de subrutinas consecutivas  $(M_i, M_{i+1})$ , el Orquestador establece la siguiente equivalencia lógica:

$$q_{final}(M_i) \equiv q_{inicial}(M_{i+1})$$

De esta forma, cuando la subrutina  $M_i$  "termina", la máquina no se detiene, sino que fluye naturalmente hacia el inicio de la lógica de  $M_{i+1}$ .

## El flujo de ejecución del algoritmo XOR-SHIFT

El Orquestador define el ciclo de vida de una iteración completa del generador pseudoaleatorio. La secuencia programada en `construirPrograma()` refleja las dependencias de datos del algoritmo matemático:

1. **Inicialización ( $q_{inicio}$ ):** Configura la cinta y posiciona el cabezal sobre el delimitador `@`.
2. **Carga de datos (`COPY`):** Transfiere el valor actual de la semilla al scratchpad.
3. **Operación A (`XOR + ACTUALIZAR`):**
  - Ejecuta  $Scratch \leftarrow Scratch \oplus (semilla \ll a)$



- **Persistencia crítica:** Ejecuta **ACTUALIZAR SEMILLA** inmediatamente después. Esto es vital para que la siguiente operación lea el resultado intermedio y no el valor original.

#### 4. Operación B (**XOR** + **ACTUALIZAR**):

- Ejecuta  $Scratch \leftarrow Scratch \oplus (semilla \gg b)$
- Se ejecuta la una segunda actualización de la semilla

#### 5. Operación C (**XOR** + **ACTUALIZAR**):

- Ejecuta  $Scratch \leftarrow Scratch \oplus (semilla \ll c)$
- Actualización final: La semilla ahora contiene el nuevo número pseudoaleatorio  $x_{i+1}$

#### 6. Estado puente:

- *Problema técnico:* La subrutina **ACTUALIZAR** finaliza con el cabezal en el scratchpad (izquierda), pero la subrutina **DETECCIÓN** espera comenzar en el delimitador central (@).
- *Solución:* El Orquestador inyecta un **estado puente** (**q\_bridge\_to\_detect**) que mueve el cabezal a la derecha hasta encontrar @, garantizando el cumplimiento de la precondición posicional de la siguiente fase.

#### 7. Verificación (**DETECCIÓN**):

- Si se detecta un ciclo, la máquina se detiene en **q\_HALT\_CICLO\_ENCONTRADO**.
- Si no, la máquina realiza un bucle de retorno al estado inicial de la fase 2 (**COPY**), comenzando el cálculo de  $x_{i+2}$ .

## Gestión de memoria y namespaces

El Orquestador es también el responsable de garantizar la integridad del espacio de estados  $Q$ . Dado que reutiliza las mismas subrutinas lógicas múltiples veces (por ejemplo, **XOR** se llama tres veces con parámetros distintos), el Orquestador inyecta un **NameSpace** único (ej. '**op1**', '**op2**', '**op3**') en cada llamada. Esto asegura que la función de transición global no contenga ambigüedades ni colisiones, permitiendo que la máquina distinga entre "estar



haciendo el primer XOR" y "estar haciendo el tercer XOR", aunque la lógica subyacente sea idéntica.

### 3. Casos de prueba y resultados

La validación del sistema se llevó a cabo siguiendo una estrategia de pruebas ascendente, comenzando por la verificación aislada de cada subrutina mediante pruebas unitarias automatizadas, y culminando con pruebas de integración del orquestador completo para validar el comportamiento pseudoaleatorio.

#### Verificación unitaria (unit testing)

Se implementó una suite de pruebas automatizadas utilizando el framework **Jasmine** y el ejecutor **Karma**. El objetivo fue garantizar que cada autómata individual cumpliera con su post-condición antes de ser integrado.

- **Subrutina COPY:** Se verificó la capacidad de duplicar cadenas binarias arbitrarias (ej. **110010** y **000000**) y se validó la **limpieza de marcadores** (**X**, **Y**), asegurando que la semilla quede en estado original tras la copia.
- **Subrutina XOR con desplazamiento:** Se probaron desplazamientos positivos (simulando **>>**) y negativos (simulando **<<**). Se validó el manejo de bordes (*padding* con ceros) cuando el desplazamiento excede los límites de la palabra, confirmando la corrección de la lógica aritmética.
- **Subrutina actualizar semilla:** Se comprobó la correcta sobrescritura destructiva de la semilla con datos provenientes del scratchpad, garantizando la persistencia del estado entre fases de cálculo.
- **Subrutina detección de ciclo:** Se sometió a la máquina a escenarios de historial vacío, historial sin coincidencias y, finalmente, historial con coincidencia exacta, validando la transición al estado final de aceptación.

#### Prueba de integración: El caso del enunciado

Para la prueba principal del sistema, se configuró el Orquestador con los parámetros sugeridos en la documentación del trabajo práctico para generar un ciclo corto verificable.



### Configuración del escenario:

- **Semilla inicial  $x_0$ :** 110010
- **Desplazamientos:**  $a = 1, b = 2, c = 1$ .
- **Objetivo:** Verificar la generación de la secuencia y la parada automática al detectar la repetición de un valor.

**Resultados obtenidos:** La simulación se ejecutó exitosamente, deteniéndose automáticamente en el estado q\_HALTED\_CICLO\_ENCONTRADO, cerrando el ciclo en la séptima iteración al volver a generar la semilla original.

**Pasos ejecutados:** 24.871

**Estado final de la cinta:** La cinta resultante muestra la evidencia completa de la ejecución: el área de cálculo limpia (sin marcas temporales), la semilla actual restaurada al valor inicial (indicando el cierre del ciclo) y el historial completo de valores generados a la derecha.

Contenido final de la cinta:

\_1100100111010001111011110110101000110101@110010#110010\$1101  
01\$101000\$011010\$101111\$000111\$011101\$110010\$\_

**Análisis de la secuencia generada:** Al decodificar el segmento de historial (separado por \$), observamos la siguiente secuencia de estados ( $x_i$ ):

1. 110010 (semilla Inicial)
2. 110101
3. 101000
4. 011010
5. 101111
6. 000111
7. 011101
8. 110010 (Repetición Detectada → Parada)

### Validación de escenarios adicionales



Escenario B:

- **Entrada:** semilla **110010**,  $(a, b, c) = (1, 1, 1)$ .
- **Resultado:**
  - **Ciclos ejecutados:** 14
  - **Pasos totales:** 94.725
  - **Estado final:** q\_HALT\_CICLO\_ENCONTRADO
- **Análisis:** Este caso generó un ciclo el doble de largo que el escenario principal. El aumento no lineal en los pasos ejecutados (de  $\sim 24k$  a  $\sim 94k$ ) confirma empíricamente la complejidad cuadrática  $O(N^2)$  de la subrutina de detección, ya que al duplicar la longitud del ciclo, el trabajo de comparación se cuadriplica aproximadamente.
- **Secuencia Detectada:** La cinta final muestra 15 entradas en el historial, confirmando que el valor **110010** se repitió tras 14 transformaciones únicas.

Contenido final de la cinta:

```
11001010110010100011000011010100101101101001110001110111101110
111101011100011100111@110010#110010$100111$000111$010111$1011
11$111011$011101$011100$011010$001011$110101$110000$101000$101
100$110010$
```

Escenario C:

- **Entrada:** semilla **111111**,  $(a, b, c) = (1, 3, 1)$ .
- **Resultado**
  - **Ciclos ejecutados:** 8
  - **Pasos totales:** 31.694
  - **Estado final:** q\_HALT\_CICLO\_ENCONTRADO
- **Análisis:** La máquina demostró estabilidad procesando una semilla de alta densidad de bits, cerrando el ciclo correctamente al reaparecer **111111**.



Contenido final de la cinta:

11111001101010100111011100100110101001111000011@11111#11111  
\$000011\$001111\$110101\$100100\$111011\$010100\$001101\$11111\$

Escenario D:

Se utilizaron los mismos parámetros del escenario principal, pero alterando la semilla inicial.

- **Entrada:** semilla **111000**,  $(a, b, c) = (1, 2, 1)$ .
- **Resultado**
  - **Ciclos Ejecutados:** 8
  - **Pasos Totales:** 24.885
  - **Estado Final:** **q\_HALTED\_CICLO\_ENCONTRADO**
- **Análisis:** Este caso produjo la misma longitud de ciclo (7) que el Escenario Principal, pero con una leve diferencia en el costo computacional (**24,885** pasos vs **24,871** pasos en el caso principal).

Esta discrepancia de **14 pasos** es significativa desde el punto de vista teórico. Aunque la estructura macroscópica del ciclo es idéntica, los números binarios generados en el historial son distintos. En la subrutina de detección, el tiempo necesario para descartar un "falso positivo" (un número del historial que *casi* coincide con el actual) depende de la posición del primer bit diferente (Mismatch).

- Si el **mismatch** ocurre en el bit 0, el descarte es inmediato.
- Si el **mismatch** ocurre en el bit 5, la máquina debe recorrer toda la palabra antes de descartar. Esta variación confirma que el tiempo de ejecución de una Máquina de Turing depende no solo de la longitud de la entrada, sino de la **entropía específica de los datos procesados**.

Contenido final de la cinta:

\_111000100001001100110000011011110011110@11100#11100\$0111  
10\$111110\$000110\$011000\$100110\$100000\$111000\$\_

## Prueba de estrés y limitaciones computacionales



Con el objetivo de validar el comportamiento del sistema ante ciclos de periodo máximo, se configuró la máquina con los parámetros teóricos para un ciclo de 63 estados: **semilla 110010**,  $(a, b, c) = (1, 3, 2)$ .

**Observaciones experimentales:** Durante esta prueba de estrés, se evidenció una limitación significativa inherente al modelo de simulación elegido, relacionada con la **complejidad temporal y espacial**.

- **Explosión de pasos:** La simulación superó los 1.100.000 **pasos** de ejecución.
- **Consumo de recursos:** El proceso agotó la memoria disponible en el entorno de pruebas (equipo con 16GB de ram), forzando la interrupción de la simulación antes de alcanzar el estado de parada.

**Análisis de la limitación ( $O(N^2)$ ):** Este resultado no es un fallo de la lógica, sino una consecuencia directa de la teoría de la computación aplicada. La subrutina de **detección de ciclos** obliga a comparar el nuevo candidato contra *cada uno* de los elementos anteriores del historial. Dado un ciclo de longitud  $L$ , la cantidad de comparaciones crece cuadráticamente. Para un ciclo de periodo máximo ( $L = 63$ ) en una máquina de Turing física, el cabezal debe recorrer una cinta que crece linealmente, realizando viajes de ida y vuelta constantes.

$$Costo \approx \sum_{i=1}^{63} (Longitud_Historial_i \times Costo_comparacion)$$

Esto demuestra empíricamente por qué, aunque la Máquina de Turing es teóricamente capaz de computar cualquier algoritmo (Turing-Completa), la **trazabilidad** de dichos problemas decae rápidamente sin estructuras de acceso aleatorio (RAM), validando la necesidad de arquitecturas modernas para criptografía eficiente.

## 4. Conclusiones

El desarrollo e implementación de este generador pseudoaleatorio XOR-Shift sobre una Máquina de Turing ha permitido validar experimentalmente la hipótesis central del trabajo: **un modelo de computación estrictamente determinista y secuencial es capaz de simular comportamientos complejos asociados a la aleatoriedad**. A continuación, se detallan las



---

conclusiones fundamentales derivadas de los resultados obtenidos, respondiendo a las interrogantes teóricas planteadas.

## Análisis del periodo y sensibilidad paramétrica

Uno de los hallazgos más relevantes del estudio experimental fue la **extrema sensibilidad** del sistema a las condiciones iniciales, específicamente a los parámetros de desplazamiento ( $a, b, c$ ).

- **Observación experimental:** Al utilizar una palabra de 6 bits, el espacio total de estados posibles (excluyendo el cero) es  $2^6 - 1 = 63$ .
  - Con los parámetros (1, 2, 1), la máquina generó un ciclo de sólo **7 números** antes de repetirse.
  - Con una variación mínima a (1, 1, 1), el ciclo se duplicó a **14 números**
  - Con los parámetros teóricos óptimos (1, 3, 2), el ciclo se expandió hacia el máximo teórico.
- **Explicación teórica:** Esto demuestra que la "calidad" de la aleatoriedad (medida aquí como la longitud del período antes de repetir un patrón) no depende de la complejidad de la máquina, sino de las propiedades matemáticas de la **función de transición** elegida. Los desplazamientos actúan como funciones de permutación en el espacio de estados. Si la permutación elegida forma un "ciclo corto", la máquina quedará atrapada repitiendo pocos valores. Si la permutación es óptima (matemáticamente primitiva), recorrerá todo el espacio disponible antes de volver al inicio.
  - **Escalabilidad:** Si bien en este TP usamos 6 bits por convención, en un sistema real de 32 o 64 bits, el período crecería exponencialmente ( $2^{64} - 1$ ). Esto significa que, aunque el ciclo existe, su longitud es tan inmensa que, para propósitos humanos, la secuencia es indistinguible de una infinita y aleatoria.

## La paradoja del determinismo y la pseudoaleatoriedad

Una pregunta central del trabajo fue: *¿Por qué una máquina determinista puede aparentar aleatoriedad sin ser verdaderamente aleatoria?*

- **Determinismo absoluto:** La Máquina de Turing implementada es un sistema determinista puro. Dada una **semilla inicial** ( $S_0$ ) y una **tabla de reglas** ( $\delta$ ), el



siguiente estado ( $S_1$ ) está predeterminado únicamente. Si reiniciamos la simulación con la misma semilla, obtendremos exactamente la misma secuencia de números, paso por paso. No hay "azar" real.

- **La Ilusión de la aleatoriedad (caos determinista):** La apariencia de aleatoriedad surge de dos propiedades introducidas por el algoritmo XOR-Shift: **difusión** y **confusión**.
  - **Operación shift:** Toma información de un bit en la posición  $i$  y la mueve a una posición lejana  $i + k$ . Esto rompe la correlación local
  - **Operación XOR (suma módulo 2):** Es una operación no lineal que mezcla la información original con la desplazada. Al combinar estas operaciones repetidamente, el algoritmo destruye rápidamente cualquier patrón reconocible por un observador humano. El valor del bit  $x_i$  en el paso 10 depende de una combinación tan compleja de los bits anteriores que se vuelve **impredecible** sin conocer la semilla y el algoritmo, simulando así el "ruido" o la entropía de un proceso aleatorio natural, aunque sea matemáticamente preciso.

## Lecciones sobre complejidad computacional

Finalmente, la prueba de estrés realizada para validar el ciclo máximo de 63 estados reveló una lección práctica sobre las limitaciones físicas de la computación teórica.

- **El cuello de botella de la memoria secuencial:** Para que la máquina sepa si un número ya salió, debe mirar hacia atrás en la cinta ("recordar el pasado").
  - En una computadora moderna (RAM), mirar un dato pasado toma un tiempo constante ( $O(1)$ ).
  - En una Máquina de Turing (cinta), mirar un dato pasado obliga al cabezal a viajar físicamente por toda la cinta.
- **Explosión combinatoria:** En nuestra prueba, la simulación colapsó al intentar procesar más de 1.2 millones de pasos. Esto sucede porque el costo de verificar cada nuevo número crece cuadráticamente ( $O(N^2)$ ). Para el número 60, la máquina tiene que recorrer y comparar contra los 59 anteriores, y luego volver.
- **Conclusión:** Esto valida experimentalmente que, aunque la Máquina de Turing es **Turing-Completa** (puede calcular cualquier cosa que sea computable), no es **eficiente** para problemas que requieren acceso aleatorio a grandes volúmenes de datos



históricos. La estructura de la memoria define qué problemas son tratables en tiempo razonable.

## 5. Referencias y bibliografía

- <https://www.educative.io/answers/pseudo-random-number-using-xorshift-algorithm>
- Augusto, Juan Carlos. **Fundamentos de Ciencias de la Computación. Notas de Curso.** Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina.
- Teoría de la Computación, J. Glenn Brookshear

## 6. Anexo: Diseño y diagramas de la Máquina de Turing

A continuación se presentan los diagramas de bloques funcionales que componen el sistema **XorShift**. El diseño se ha modularizado en subrutinas especializadas para facilitar la comprensión del flujo de datos y las transiciones de estados.

### Leyenda de símbolos

Para la interpretación de los diagramas se utiliza la siguiente notación estándar:

- **R / L**: Movimiento del cabezal a la derecha (*Right*) o izquierda (*Left*).
- #: Separador de fin de semilla (**SEP\_FIN**).
- @: Separador de inicio de área de trabajo (**SEP\_INICIO**).
- \$: Separador de entradas en el historial (**SEP\_HISTORIAL**).
- Δ: Espacio en blanco (*Blank*).

### Máquina de copiado y restauración

Este diagrama representa una **MT de copia**. Su función es duplicar la **semilla** (derecha de @) en el **scratchpad** (izquierda de @) y restaurar el valor original de la semilla.

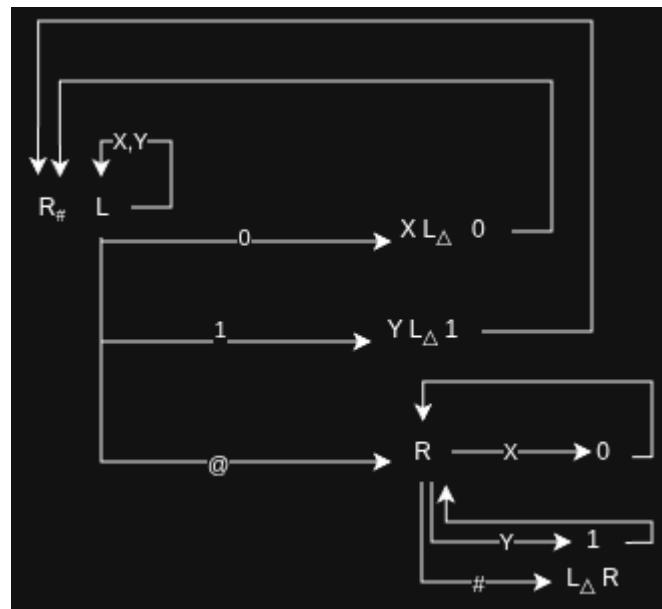


## 1. Ciclo de copia (bucle principal)

- a. **Inicialización:** El proceso comienza con  $R_{\#}$  para posicionar el cabezal al final de la semilla.
- b. **Marcado y viaje:** La máquina lee el bit más derecho no marcado (0 o 1), lo marca como **X o Y**, y se desplaza hacia la izquierda con  $L_{\Delta}$  para encontrar el primer espacio vacío en el *Scratchpad*.
- c. **Escritura y retorno:** Escribe el bit copiado (0 o 1) y regresa inmediatamente al nodo principal para buscar el siguiente bit no marcado.

## 2. Fase de limpieza y finalización

- a. **Activación:** Cuando la máquina encuentra el símbolo @ (indicando que toda la semilla ha sido marcada y copiada), sale del bucle principal.
- b. **Restauración:** Se mueve a la derecha (R) e inicia una limpieza, recorriendo la semilla y reemplazando las marcas **X→0** y **Y→1**.
- c. **Posicionamiento Final:** Al alcanzar el # (fin de la limpieza), ejecuta  $L_{\Delta} R$  para dejar el cabezal justo sobre el primer bit de la copia en el *scratchpad*, listo para la siguiente subrutina.



## Máquina XOR con desplazamiento



El siguiente diagrama representa la subrutina de **cálculo aritmético-lógico**. Su función es transformar cada bit del *scratchpad* aplicando una operación **XOR** contra un bit de la *semilla* ubicado a una distancia variable ( $k$ ).

### 1. Ciclo de lectura y memorización

- a. **Marcado:** El cabezal lee el bit actual en el *scratchpad*. Para "memorizar" este valor durante el viaje, lo sustituye por una marca temporal:
  - Si lee **0**, escribe **A** y toma la ruta superior.
  - Si lee **1**, escribe **B** y toma la ruta inferior.
- b. **Navegación:** Ambas rutas se dirigen hacia la derecha buscando el separador **@** para entrar en la zona de la semilla.

### 2. Bloque de desplazamiento abstracto ( $R_{shift}$ )

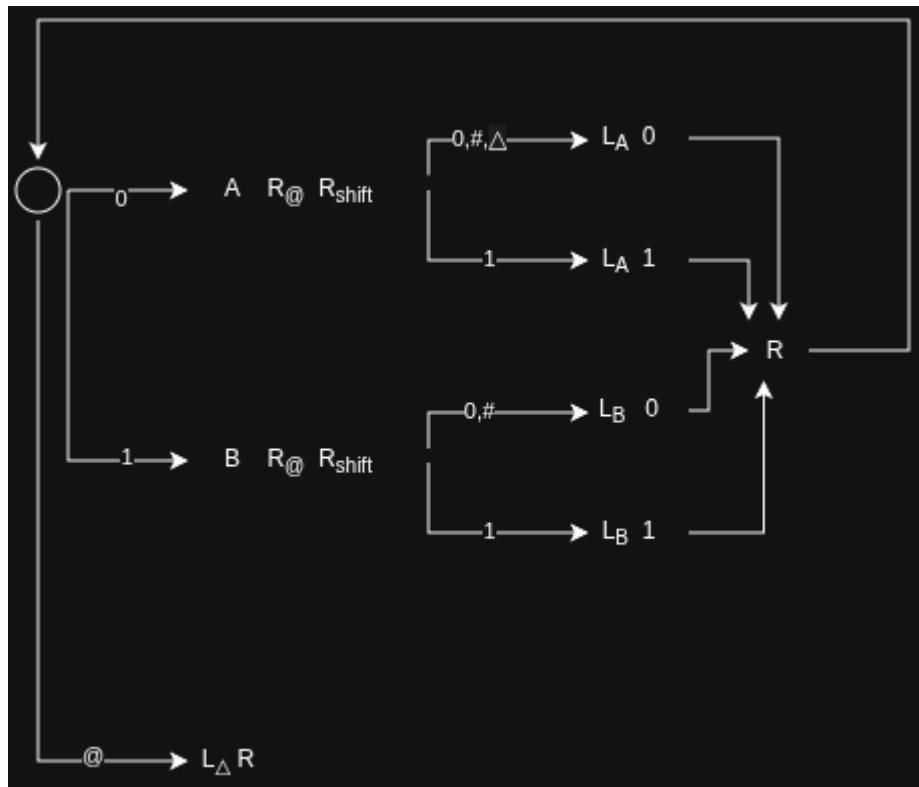
- a. **Función:** Una vez pasado el **@**, se ejecuta el bloque **shift**, que desplaza el cabezal  $k$  posiciones a la derecha (donde  $k = \text{índice} + \text{desplazamiento}$ )
- b. **Manejo de límites:** Este bloque tiene una "salida de emergencia". Si durante el desplazamiento encuentra el fin de cinta (#) o un blanco ( $\Delta$ ), interrumpe el movimiento y asume lógicamente que el bit leído es un **0**.

### 3. Retorno y cálculo (lógica XOR)

- a. **Lectura de Destino:** La máquina lee el bit en la posición destino de la semilla (o el 0 virtual si se salió de los límites).
- b. **Retorno a la Marca:** Ejecuta  $L_A$  (si partió de 0) o  $L_B$  (si partió de 1) para regresar a la posición original en el *scratchpad*.
- c. **Resolución:** Aplica la tabla de verdad XOR entre la marca (A/B) y el valor leído en la semilla:
  - $A(0) \oplus 0 \rightarrow \text{escribe } 0.$
  - $A(0) \oplus 1 \rightarrow \text{escribe } 1.$
  - $B(1) \oplus 0 \rightarrow \text{escribe } 1.$
  - $B(1) \oplus 1 \rightarrow \text{escribe } 0.$

### 4. Finalización

Cuando el cabezal lee el símbolo **@** en el estado inicial (indicando que se han procesado todos los bits del *scratchpad*), ejecuta  $L_\Delta R$  para rebobinar hasta el inicio y dejar la cinta lista para la siguiente etapa (actualizar semilla).



## Máquina de actualización de semilla

El diagrama ilustra la subrutina de **persistencia de datos**. Su función es transferir los valores calculados en el *scratchpad* hacia la *semilla*, sobrescribiendo los valores antiguos bit a bit.

### 1. Ciclo de transporte

- Lectura y marcado:** El cabezal lee el bit actual en el *scratchpad*. Para transportar el valor, sustituye el bit por una marca:
  - Si lee **0**, escribe **A**.
  - Si lee **1**, escribe **B**.
- Navegación:** Avanza hacia la derecha cruzando el separador **@** y utiliza el bloque abstracto  $R^i$  para situarse exactamente sobre el bit correspondiente en la semilla.

### 2. Escritura

A diferencia de la subrutina XOR, esta fase no lee el valor destino.

- Sobrescritura:** Dependiendo de la ruta tomada (origen 0 o 1), la máquina escribe directamente el nuevo valor en la cinta, eliminando el dato anterior.
- Retorno:** Inicia el movimiento a la izquierda (**L**) buscando la marca de origen.

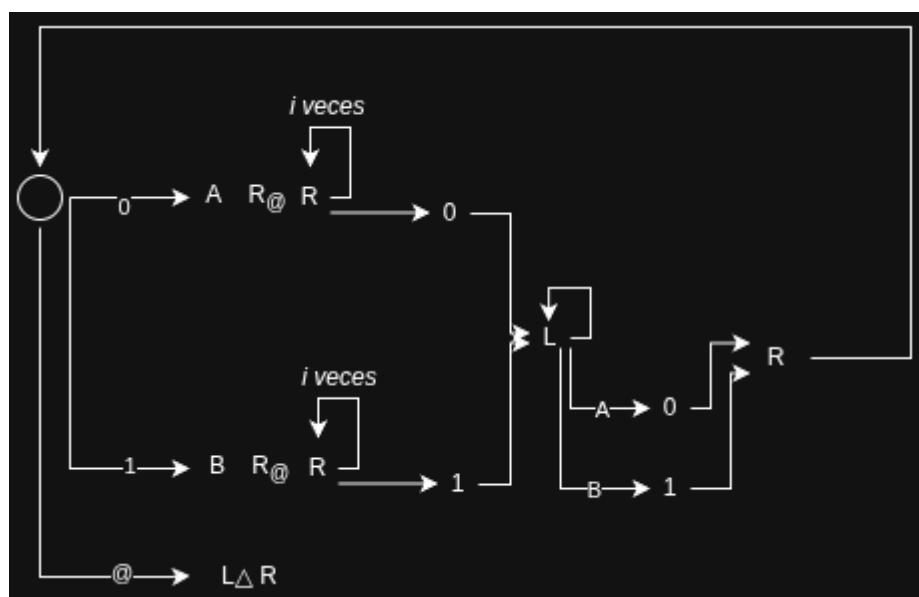


### 3. Restauración y avance

- Recuperación:** Al encontrar la marca en el *scratchpad*:
  - Si lee **A**, restaura el **0**.
  - Si lee **B**, restaura el **1**.
- Siguiente bit:** Ejecuta **R** para posicionarse sobre el siguiente bit a procesar y reiniciar el ciclo.

### 4. Finalización

Cuando el cabezal encuentra el separador **@** en el estado inicial (fin del *scratchpad*), ejecuta la secuencia de rebobinado  $L_{\Delta}R$  para devolver el cabezal al inicio de la cinta, completando la actualización.



## Máquina de detección de ciclos

El diagrama ilustra la subrutina de **detección**. Su función es verificar si el número recién generado ya existe en el historial para prevenir bucles infinitos.

### 1. Registro y preparación

- Anexar al historial:** La máquina se desplaza al final absoluto de la cinta ( $R_{\Delta}$ ) y ejecuta el bloque de **Copia**, duplicando la semilla actual al final del área de historial.



- b. **Delimitación:** Escribe un separador de entrada (\$) y una marca temporal de fin de búsqueda (&) para acotar la zona de investigación.

## 2. Bucle de verificación

La máquina rebobina y compara la semilla actual contra cada entrada del historial secuencialmente.

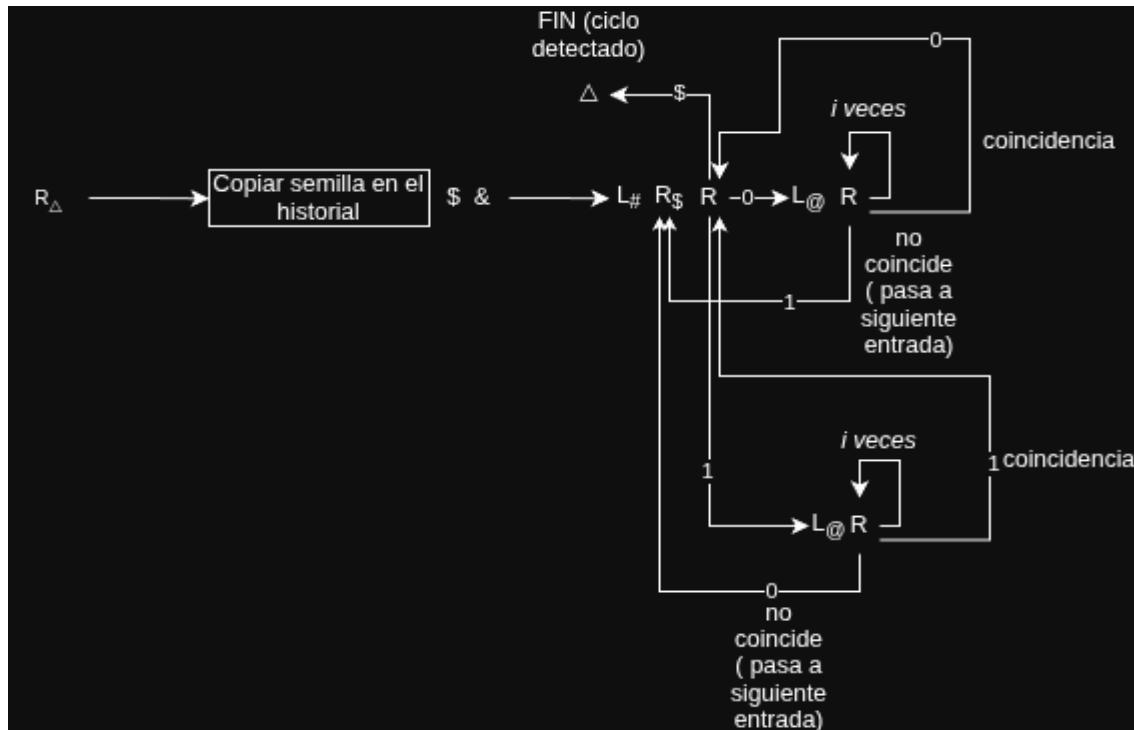
- a. **Comparación Bit a Bit:** Lee un bit del historial y viaja a la posición equivalente en la Semilla ( $L_{@} + desplazamiento$ ) para verificar la igualdad.

## 3. Manejo de discrepancias:

- a. **Coincidencia:** Si los bits son iguales, avanza al siguiente bit de la entrada histórica.
- b. **No coincidencia:** Si encuentra una diferencia, aborta la revisión de esa entrada y salta inmediatamente al siguiente separador  $R_{\$}$  para evaluar el siguiente número guardado.

## 4. Decisión final

- a. **Ciclo detectado (HALT):** Si la máquina logra comparar una entrada completa del historial sin encontrar ninguna discrepancia (llegando al separador \$), se confirma la repetición. La máquina entra en un **estado de parada**.
- b. **Número nuevo:** Si la máquina recorre todo el historial hasta encontrar la marca final & sin hallar coincidencias completas, transforma la marca en un separador permanente y retorna el control al inicio para generar el siguiente número.



## Orquestador del sistema XorShift

El siguiente diagrama representa la **arquitectura de control** del autómata. Este diagrama no procesa bits individualmente, sino que actúa como un autómata compuesto que coordina la ejecución secuencial de las subrutinas previamente definidas para implementar el algoritmo XorShift.

### 1. Inicialización y registro

- Detección inicial:** Antes de comenzar la generación, el sistema ejecuta una instancia especial de la subrutina de detección. Su único propósito es registrar la **semilla inicial** en el historial, estableciendo el primer punto de referencia para futuras comparaciones.

### 2. Núcleo de procesamiento (algoritmo XorShift)

El flujo entra en un bucle generativo compuesto por tres etapas secuenciales de transformación (correspondientes a los desplazamientos  $a$ ,  $b$  y  $c$ ). Para garantizar la integridad aritmética, se impone el siguiente patrón:

- Preparación:** Se ejecuta **COPY** para llevar los datos al área de trabajo.
- Cálculo y persistencia:** Se ejecutan pares de **XOR + ACTUALIZAR**.
  - El bloque **XOR** calcula el desplazamiento ( $a$ ,  $b$  o  $c$ ).



- El bloque **ACTUALIZAR** persiste inmediatamente ese resultado en la semilla.
- *Esto asegura que la operación B se realice sobre el resultado de A, y la C sobre el resultado de B.*

### 3. Alineación y verificación

- a. **Estado Puente:** Tras la última actualización, se ejecuta una transición técnica ( $R_{@}$ ) para alinear el cabezal desde el inicio del *scratchpad* hasta el separador central.
- b. **Detección final:**
  - **Coincidencia:** Si el número generado ya existe en el historial, el sistema transiciona al **estado de parada** (Fin de ciclo).
  - **Nuevo valor:** Si el número es único, el flujo retorna al bloque **COPY** (inicio del bucle) para generar el siguiente número de la secuencia pseudoaleatoria.

