UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"ESTUDIO DE MECANISMOS PARA COMBATIR EVENTOS TRANSITORIOS EN MEMORIAS PARA APLICACIONES ESPACIALES"

TRABAJO FIN DE GRADO

Febrero – 2024

AUTOR: Alex Ananías Yáñez Sigüenza
TUTOR/ES: Roberto Gutiérrez Mazón
Daniel Gutiérrez Castro
DEPARTAMENTO: Ingeniería de Comunicaciones

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ILUSTRACIONES CÓDIGO

ÍNDICE DE TABLAS

1. PROBLEMÁTICA DE LOS DATOS DIGITALES EN EL ESPACIO

Cada periodo de avance tecnológico, la constante evolución de los circuitos integrados se ha convertido en materia de vital importancia. Los circuitos integrados continúan optimizándose cada vez más, reduciendo su tamaño, mejorando sus prestaciones y eficiencia con el fin de mantener la correspondencia con la Ley de Moore (Ilustración 1).

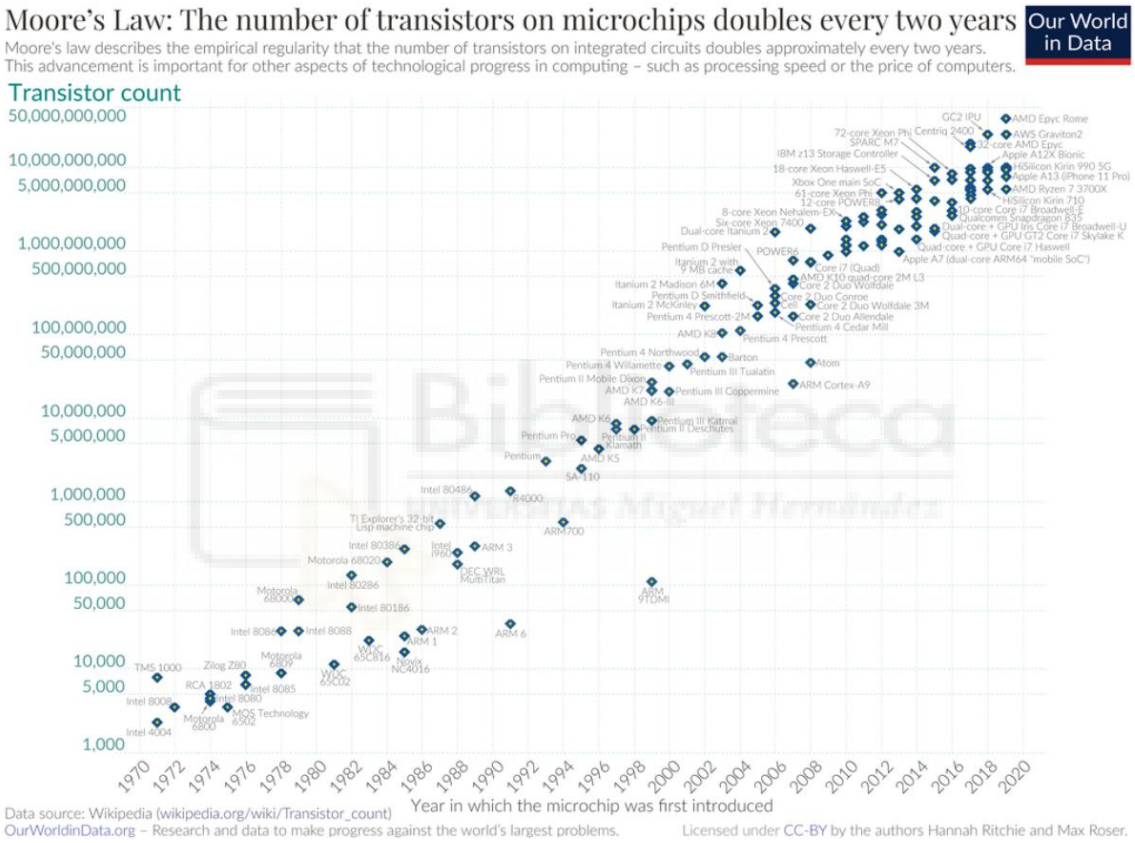

Ilustración 1. Gráfica de la Ley de Moore

En 1965, Gordon Moore, cofundador de Intel hizo una predicción en la cual el número de transistores en un chip se duplicaría aproximadamente cada dos años, con poco aumento en el coste de fabricación [1].

Las memorias RAM (Random Access Memory) también se han visto afectadas por esta evolución. Físicos, ingenieros e investigadores han centrado sus esfuerzos en reducir el tamaño de los transistores en las celdas de memoria a niveles de escala nanométrica. Esto permitirá reducir el tamaño por unidad de área de las mismas y por consiguiente el coste por bit de fabricación (al implementar más celdas en una misma oblea de silicio). Sin embargo, esta búsqueda de la miniaturización plantea desafíos significativos en el entorno espacial, entre ellos la estabilidad de los datos en la memoria RAM ante la presencia de radiación cósmica en el entorno [2] [3].

La radiación cósmica es una forma de radiación compuesta por partículas subatómicas altamente energéticas, como protones, partículas alfa y núcleos pesados, que viajan a velocidades cercanas a la velocidad de la luz (aproximadamente 300 km/s) a través del espacio. Su intensidad aumenta con la altitud y su abundancia varía con la latitud debido a la influencia de la magnetosfera terrestre. Este fenómeno fue descubierto por el físico austriaco Víctor Franz Hess en 1912, cuando demostró que una radiación ionizante provenía del espacio exterior [4].

En la actualidad, se conoce que parte de esta radiación se genera a partir del nacimiento de supernovas en estrellas masivas fuera de nuestro Sistema Solar, así como por los vientos solares emitidos por nuestro propio sol, entre otros fenómenos. Sin embargo, las teorías que especulan sobre los orígenes exactos de su procedencia son diversas y se sigue investigando en detalle [5] [6].

Este acontecimiento puede tener efectos significativos en los satélites espaciales de comunicación, provocando comportamientos inesperados en componentes digitales pertenecientes a sus circuitos eléctricos internos. Esto representa un desafío para la ley de Moore, puesto que a medida que los transistores en las celdas de memoria se fabrican cada vez de menor tamaño, esto también los hacen más susceptibles a interferencias causadas por la radiación cósmica.

La presencia de radiación cósmica en el espacio es un problema crítico que debe ser abordado. Esto se debe no solo a que compromete la integridad de los datos almacenados en satélites y sondas espaciales, sino a que además cualquier daño causado por dicha radiación puede amenazar la misión actual.

2. OBJETIVOS DEL PROYECTO

El objetivo de este proyecto se basa en el estudio del arte en lo que respecta a mecanismos para combatir las SEUs en las memorias SRAM utilizadas en aplicaciones espaciales. Este estudio se centrará en la aplicación de técnicas de corrección de errores y redundancia para abordar estos problemas.

Además, se propone el diseño y verificación sobre una Spartan 7 FPGA (Field Programable Gate Array) de dos tipos de memorias Static RAM utilizando el lenguaje VHDL (Very High Descriptive Language), la primera mediante la técnica de detección y corrección de errores (EDAC), y la segunda mediante la triple redundancia modular (TMR). Estas dos implementaciones serán sometidas a un análisis comparativo.

Es importante destacar que este proyecto está diseñado específicamente para su aplicación en orbitas cercanas a la Tierra, donde la radiación cósmica es relativamente baja. Esta consideración es esencial ya que influye en las decisiones tomadas en cuanto a estrategias de mitigación de errores y la evaluación de su rendimiento.

3. TIPOS DE MEMORIAS RAM MÁS COMUNES

3.1 MEMORIA SRAM

Las memorias SRAM (Static Random Access Memory) son dispositivos de almacenamiento compuestos por semiconductores capaces de retener los datos almacenados en su circuito de manera estática. Se tratan de memorias volátiles, es decir, pierden la información si se interrumpe la alimentación eléctrica. Además, son del tipo acceso aleatorio, lo que implica que los datos pueden ser escritos o leídos en cualquier posición de memoria de forma arbitraria.

El interior de una SRAM está compuesto por una matriz de celdas de memoria, las cuales se encargarán de almacenar los bits de datos introducidos (Ilustración 2). Cada celda tiene una posición concreta descrita denominada dirección de memoria (address), lo que permite seleccionar en qué ubicación se almacenan los bits de datos. Cada celda de memoria está compuesta por seis transistores de tipo FET (Field Effect Transistor) o MOSFET (Metal Oxide Semiconductor Field Effect Transistor). Cuatro de estos transistores, dispuestos en pares (M1-M2 y M3-M4), forman dos puertas lógicas inversoras en paralelo, dando lugar a un biestable. Estos transistores mantendrán el dato introducido de forma estática siempre y cuando estén alimentados eléctricamente. Ambos transistores restantes (M5-M6) se encargan de permitir la entrada y salida de los bits de datos. También se pueden encontrar otros tipos de SRAM con ocho, diez o más transistores por bit, dependiendo de sus características y aplicaciones específicas (en la Ilustración 3 se muestra un ejemplo de celda SRAM compuesta por 6 Transistores de tipo MOSFET).

Finalmente, cada celda de memoria está conectada a una línea de palabra (word line) que habilita la operación de escritura o lectura, así como a dos entradas para la línea de bits (bit lines) por las cuales circulan los bits que se desean escribir o leer. La matriz de celdas o células está conformada por estas líneas anteriores [7].

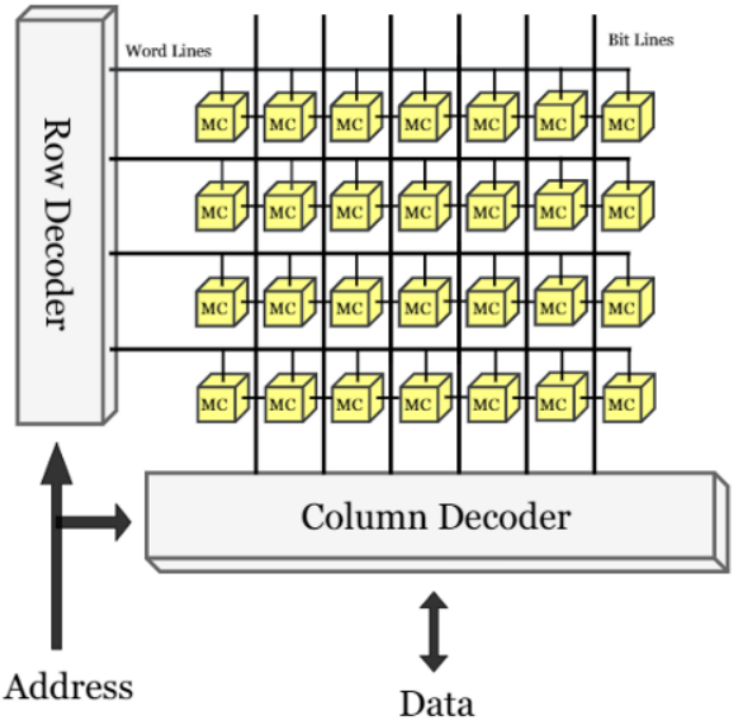

Ilustración 2. Matriz de celdas de memoria SRAM

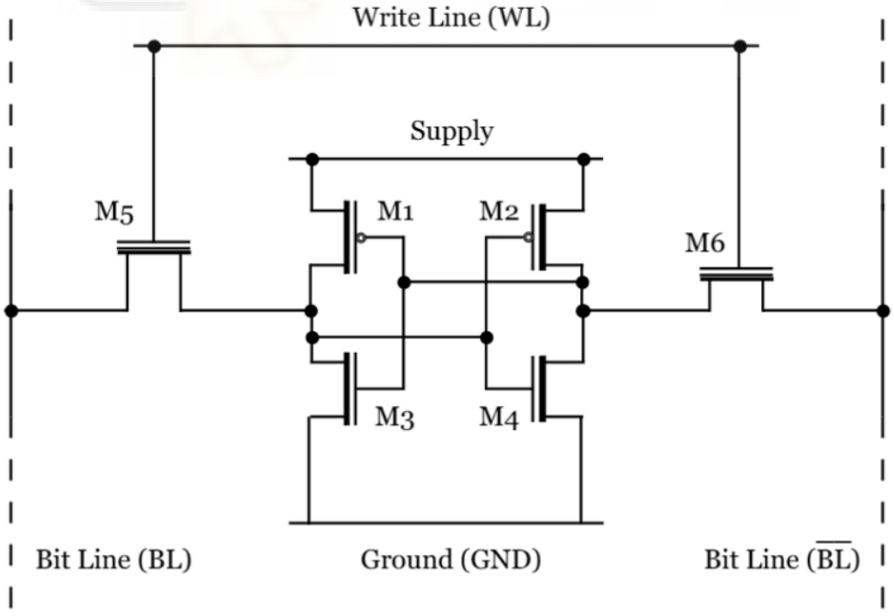

Ilustración 3. Circuito interior celda de memoria SRAM

La memoria SRAM tiene 3 modos de funcionamiento:

3.1.1 ESCRITURA

Supóngase que se quiere introducir un ‘1’ lógico:

1. Se habilita la “word line” (WL), la cual da corriente a las puertas de los MOSFET M5-M6, conectando la celda a las dos “bit lines” (BL).

2. Se introduce el ‘1’ lógico por la “bit line” (BL) conectada al transistor MOSFET M5, y su negada,‘0’ lógico, a la “bit line” (BL negada) conectada al transistor MOSFET M6. De esta manera se acelera el proceso de cambio de voltaje del biestable.

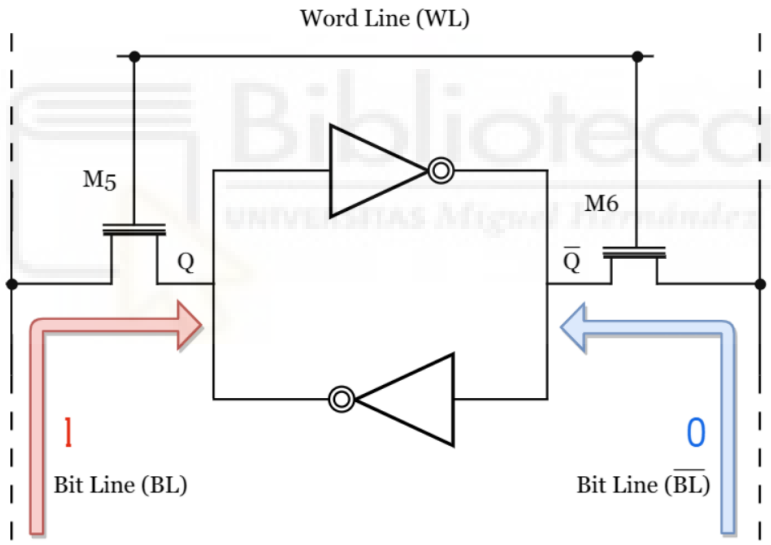

Ilustración 4. Escritura sobre celda de memoria SRAM (1)

3. El ‘1’ lógico pasa a través del inversor formado por los MOSFET M1-M2, invirtiendo su valor, y posteriormente a través del inversor formado por los MOSFET M3-M4, volviéndose a invertir y así sucesivamente de forma cíclica hasta volver a habilitar la línea de escritura o lectura, es decir, permanece almacenado dentro de la célula.

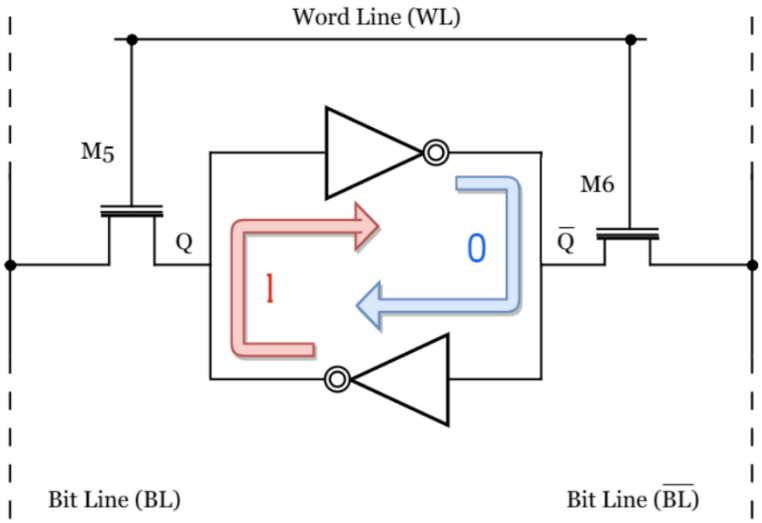

Ilustración 5. Escritura sobre celda de memoria SRAM (2)

Al introducir un '0' lógico ocurrirá prácticamente el mismo proceso:

1. Se habilita la "word line" (WL), dando corriente a las puertas de los MOSFET M5-M6.
2. Se introduce el '0' lógico por la "bit line" (BL) conectada al transistor MOSFET M5, y su negada,'1' lógico, a la "bit line" (BL negada) conectada al transistor MOSFET M6. Al tratarse de un '0' lógico no hay paso de corriente de la fuente al drenador del transistor MOSFET M5. Este caso es interpretado por la célula de memoria como un '0' lógico.

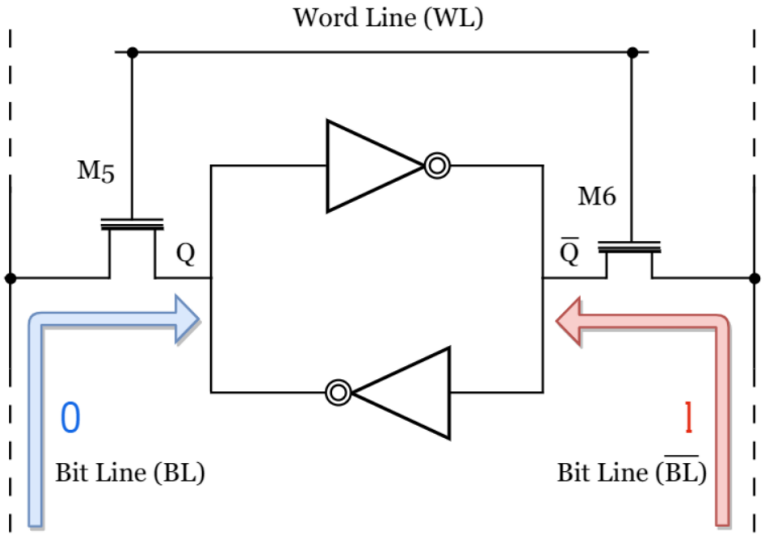

Ilustración 6. Escritura sobre celda de memoria SRAM (3)

3. El ‘0’ lógico pasa a través del inversor formado por los MOSFET M1-M2, invirtiendo su valor, y viceversa al atravesar el inversor formado por los MOSFET M3-M4, repitiéndose el proceso de manera cíclica, quedando almacenado como en el caso anterior hasta que la línea de escritura o lectura sea habilitada.

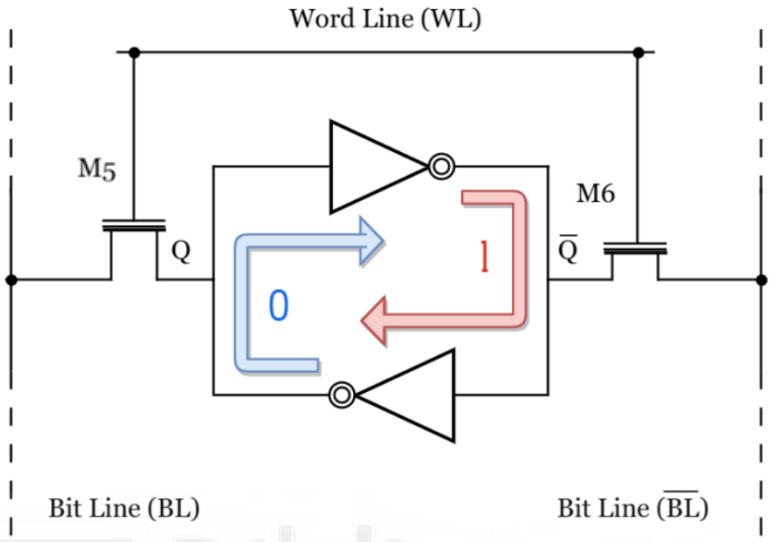

Ilustración 7. Escritura sobre celda de memoria SRAM (4)

Generalmente, el transistor habilitador de escritura (M5) es de mayor tamaño que los que forman los inversores en la celda de memoria. Esta práctica tiene el objetivo de garantizar que el MOSFET que controla la escritura pueda superar la resistencia de los MOSFET que almacenan el dato de la memoria, con el fin de poder sobrescribir el bit almacenado en su interior por el nuevo bit que se desee almacenar llegado a través de la línea de bits o “bit line” (BL).

3.1.2 LECTURA

1. Se habilita la “word line” (WL), dando corriente a las puertas de los MOSFET M5-M6.
2. Se introduce por ambas “bit lines” (BL) un valor de voltaje intermedio entre alto y bajo, es decir, entre ‘0’ y ‘1’ ´lógicos.
3. Si hay almacenado un ‘1’ lógico en la célula, el voltaje del transistor MOSFET M5 aumentará hasta alcanzar un valor cercano al de voltaje alto y el transistor MOSFET M6 disminuirá su voltaje hasta alcanzar el valor cercano al del voltaje bajo. Lo contrario ocurrirá si hay almacenado un ‘0’ lógico. Para acelerar este proceso se observa la diferencia entre los valores de voltaje de ambas “bit lines” (BL), observando si la “bit line” (BL) no negada aumenta de voltaje (leerá un ‘1’

lógico) o disminuye (leerá un ‘0’ lógico). Al desactivarse la lectura, el bit permanecerá almacenado y no se sobrescribirá.

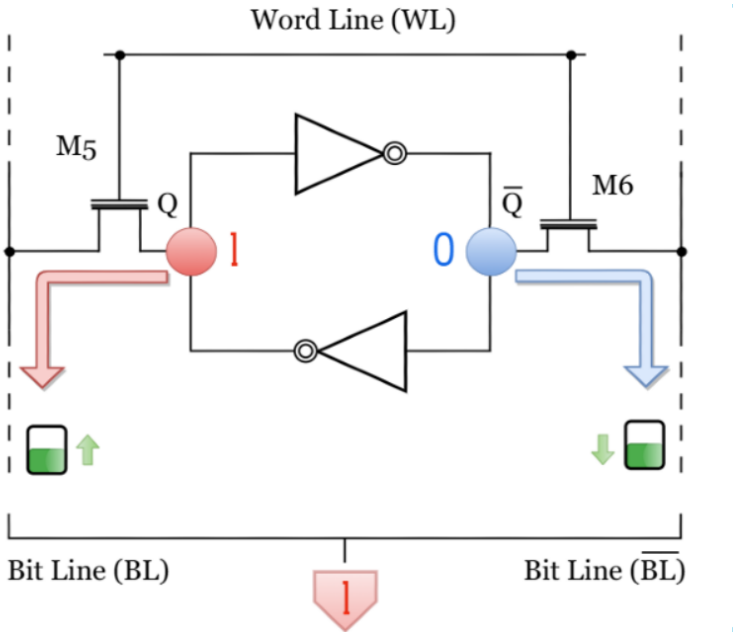

Ilustración 8. Lectura de celda de memoria SRAM

3.1.3 REPOSO

Este es el estado en el cual no se está habilitando la lectura ni la escritura. El bit permanece almacenado siempre y cuando los pares de transistores que forman el biestable, M1-M2 y M3-M4, estén alimentados.

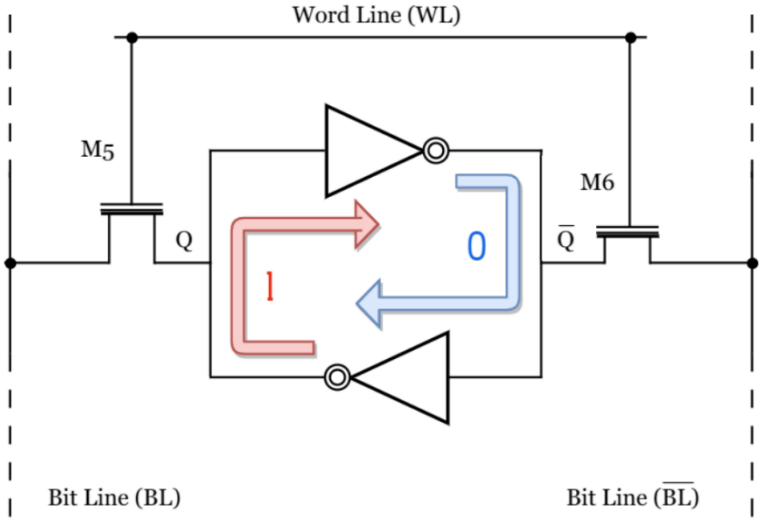

Ilustración 9. Celda de memoria SRAM en reposo

3.2 MEMORIA DRAM

Las memorias DRAM (Dynamic Random Access Memory) se basan en el uso de condensadores en sus celdas de memoria o células. Sin embargo, los condensadores tienden a perder su carga con el tiempo, por lo que se necesita reescribir los datos en las posiciones de memoria continuamente mediante un circuito de refresco que opere periódicamente cada cierto intervalo de tiempo.

Al igual que las SRAM, las DRAM también constan de una matriz de celdas de memoria y ofrecen acceso aleatorio. Sin embargo, la estructura de las células de memoria es diferente. En lugar de usar 6 transistores, las celdas de memoria constan de un solo transistor (FET o MOSFET) para el bit de datos, que conecta la "bit line" con la celda, y un condensador que almacena el valor del bit. Además, a diferencia de las SRAM, en las DRAM se utiliza un solo "bit line" para la operación. Este diseño permite una mayor densidad de almacenamiento [8].

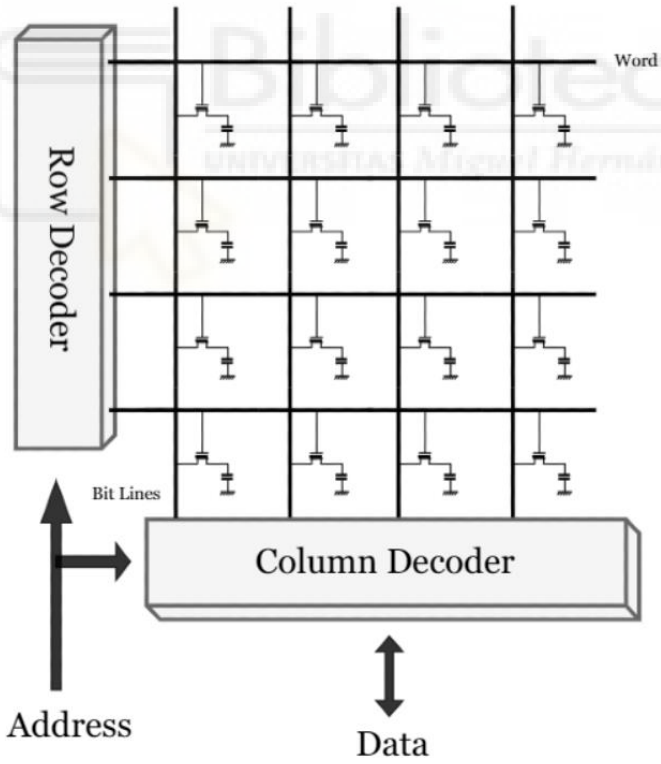

Ilustración 10. Matriz de celdas de memoria DRAM

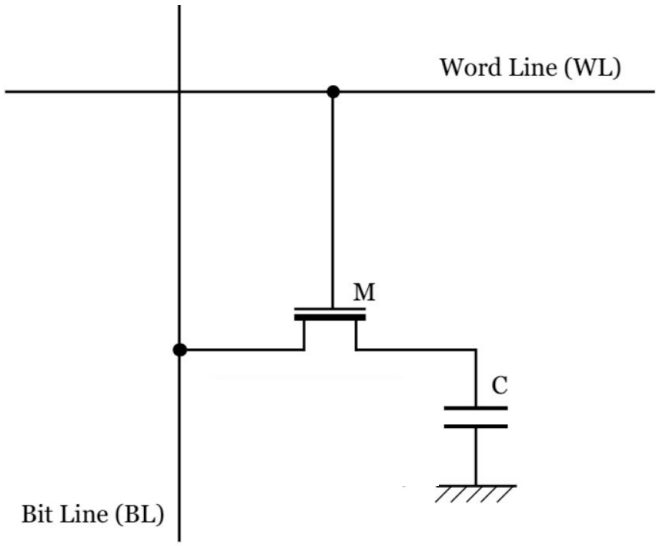

Ilustración 11. Circuito interior de celda de memoria DRAM

La memoria DRAM tiene 3 modos de funcionamiento:

3.2.1 ESCRITURA

Supóngase que se desea escribir un ‘1’ lógico:

1. Se habilita la “word line” (WL), dando corriente al transistor MOSFET.
2. Se introduce el ‘1’ lógico que a través de la “bit line” (BL). Este bit pasa a través del MOSFET, de fuente a drenador, hasta llegar al condensador, el cual se cargará con el voltaje equivalente al ‘1’ lógico.
3. Una vez desactivada la “word line” (WL), el condensador retendrá el valor ‘1’ lógico.

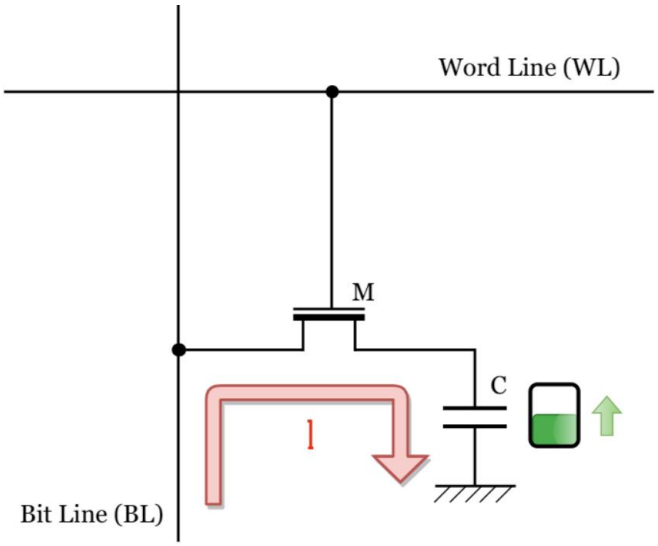

Ilustración 12. Escritura sobre celda de memoria DRAM (1)

Al escribir el ‘0’ lógico el proceso cambia ligeramente:

1. Se habilita la “word line” (WL), dando corriente al transistor MOSFET.
2. Introducimos el ‘0’ lógico a través de la “bit line” (BL). Al no existir corriente, el condensador se descargará a través de la misma si anteriormente estaba cargado con un voltaje equivalente a un ‘1’ lógico, o se mantendrá descargado en caso contrario.
3. Una vez desactivada la “word line” (WL) el condensador permanecerá descargado, representando el ‘0’ lógico que se pretendía guardar.

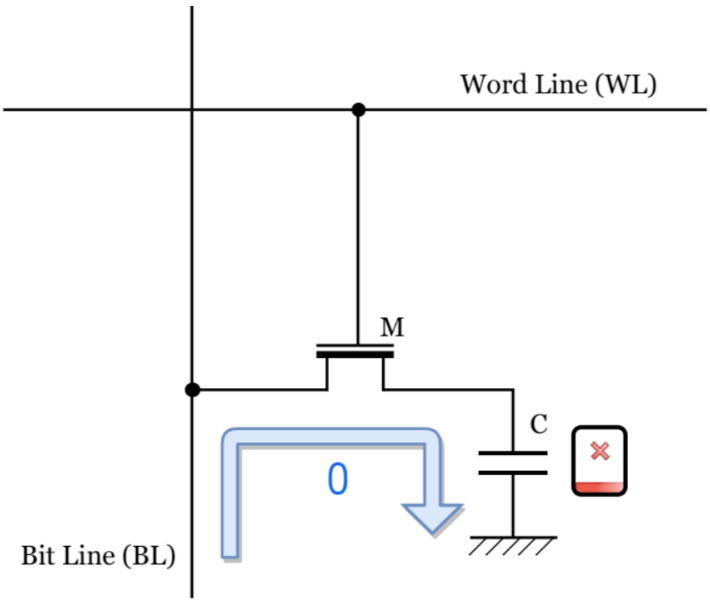

Ilustración 13.Escritura sobre celda de memoria DRAM (2)

3.2.2 LECTURA

1. Se habilita la “word line” (WL), dando corriente al transistor MOSFET.
2. El condensador descarga el valor del bit que tenía guardado a través de la “bit line” (BL). Si el condensador ya se encontraba descargado, se interpretará como la lectura de un ‘0’ lógico.
3. Una vez se desactive la “word line” (WL), el condensador quedará descargado.

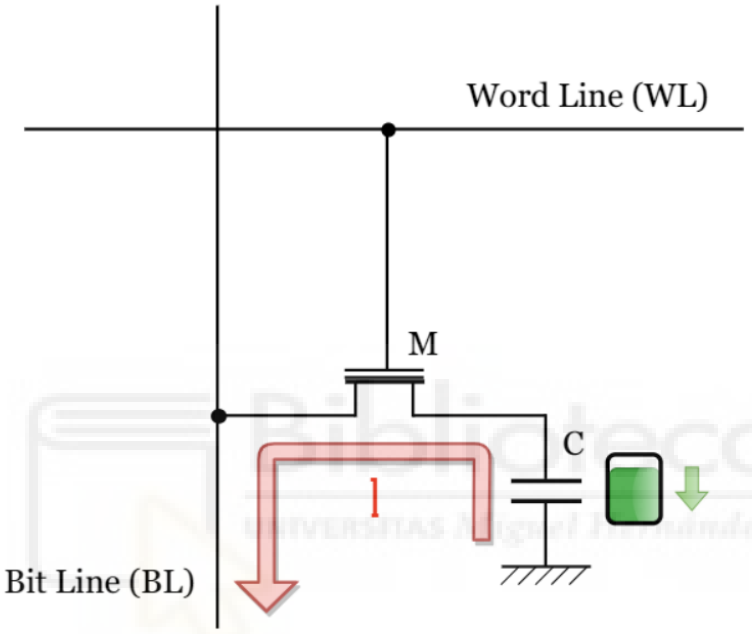

Ilustración 14. Lectura sobre celda de memoria DRAM

3.2.3 REPOSO

Este es el estado en el cual no está habilitada ni la escritura ni la lectura. El condensador permanecerá cargado si se ha guardado un ‘1’ lógico en él o descargado en el caso de haber guardado un ‘0’ lógico. En el caso de estar cargado con un ‘1’ lógico, el condensador irá disminuyendo su carga debido a las pérdidas de corriente que caracterizan a los condensadores, por lo que será necesario reescribir el voltaje equivalente al ‘1’ lógico de nuevo mediante un circuito de refresco, cada cierto periodo de tiempo, para evitar un futuro error debido a este evento. Dicho circuito de refresco, llegado el periodo asignado,

realizará una lectura de la celda y reescribirá de nuevo el valor presente originalmente antes de su lectura.

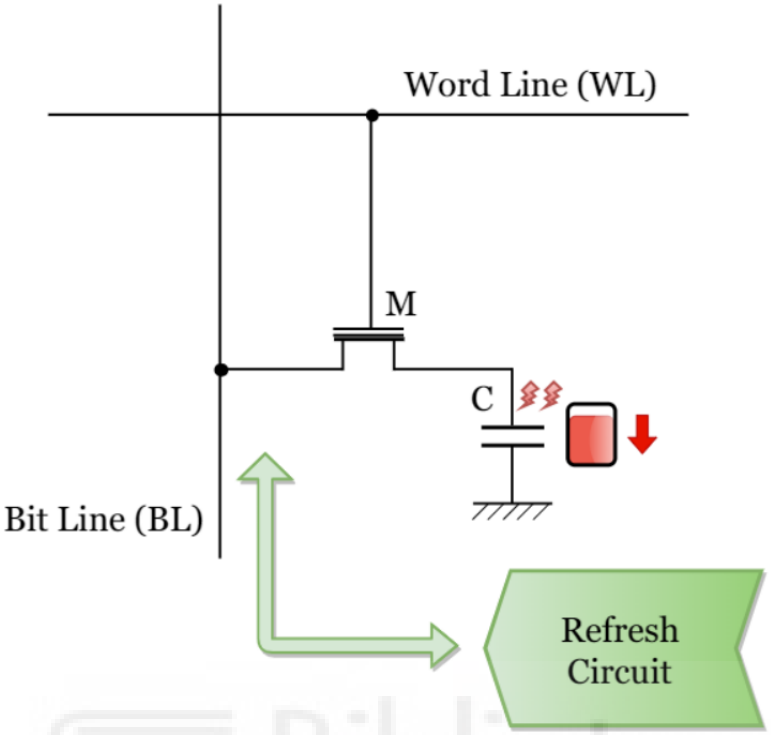

Ilustración 15. Celda de memoria DRAM en reposo

La lectura de una celda de memoria DRAM supone un inconveniente, puesto que tras este proceso el condensador pierde la carga. Como resultado, si la celda contenía '1' lógico, esa información se pierde durante la lectura. A esto se le conoce como lectura destructiva.

Para resolver este problema, tras leer el valor de la celda, se procede a reescribir el bit extraído, mediante el mismo circuito de refresco utilizado para solventar las pérdidas de carga del condensador. Este paso de reescritura garantiza que la información se restaure en la celda, evitando así la pérdida de datos y posibles errores de lectura en el futuro.

Para aplicaciones espaciales, se utiliza la memoria de tipo SRAM debido a sus ventajas específicas [9]:

Tamaño compacto: Ocupa un área reducida en comparación con otros tipos de memoria, lo que es fundamental en aplicaciones espaciales donde el espacio es limitado y valioso.

Mayor tolerancia a errores: Es menos propensa a sufrir SEUs (Single Error Upsets) en comparación con la DRAM, lo que es crucial en un entorno espacial donde la radiación puede causar errores en los datos almacenados.

Velocidad de acceso rápida: Ofrece una alta velocidad de acceso a la memoria, lo que la hace ideal para ejecutar tareas intensivas en tiempo real. En aplicaciones espaciales, la baja latencia y velocidad son críticas, ya que los satélites deben responder a eventos rápidos y llevar a cabo operaciones en tiempo real.

Durabilidad y Confiabilidad: Son menos propensas a sufrir desgaste que las DRAM, ya que no requiere un circuito de refresco constante para mantener los datos en la memoria. Esta característica las hace más duraderas y confiables en aplicaciones a largo plazo.

Eficiencia energética: Por lo general tienen un menor consumo de energía debido a su almacenamiento estático. Esto es esencial, ya que la eficiencia energética juega un papel importante para garantizar el funcionamiento de los sistemas a largo plazo.

Interfaz sencilla: Posee una interfaz más sencilla al no incluir los circuitos de refresco, lo que simplifica el diseño y la operación de sistemas espaciales.

Resistencia ante cambios de temperatura: Demuestra una buena resistencia a las fluctuaciones de temperatura, lo cual las hace idóneas en el espacio, donde las condiciones térmicas varían significativamente.

4. DESCRIPCIÓN DE LOS ERRORES SINGLE EVENT EFFECTS

Los “Single Event Effects” (SEEs) son fenómenos inducidos por la radiación cósmica que afectan a los componentes electrónicos en circuitos presentes en satélites y otros dispositivos en el espacio. Estos eventos se producen cuando una partícula cargada (ión pesado o neutrón) atraviesa el dispositivo, generando un par electrón-hueco, provocando así un evento transitorio o un error permanente en su funcionamiento. Estos errores pueden tener un impacto significativo en las operaciones de los dispositivos electrónicos y circuitos en entornos espaciales. En términos generales, existen dos categorías principales [10] [11]:

4.1 DESTRUCTIVOS

Los eventos destructivos se refieren a aquellos eventos que provocan daños en el hardware, es decir, aquellos que pueden causar daños significativos o catastróficos en los componentes electrónicos. Estos eventos pueden manifestarse de varias maneras:

- Single Event Burnout (SEB): Los SEB son eventos en los cuales un solo impacto de una partícula energética induce un estado de alta corriente en el dispositivo. Esta corriente excesiva puede quemar el circuito interno, lo que resulta en un fallo catastrófico e irreversible. Por lo general, suele afectar a los transistores MOSFET de potencia que manejan altas tensiones y corrientes.

- Single Event Gate Rupture (SEGR): Los eventos SEGR están relacionados con los SEB. El impacto de la partícula también induce una alta tensión o corriente en el transistor, provocando un cortocircuito y ,por tanto, daños permanentes en la puerta lógica. También suele afectar a los MOSFET de potencia que manejan altas tensiones y corrientes.

- Single Event Dielectric Rupture: Estos eventos son similares a los dos anteriores, pero en dispositivos que no están destinados a aplicaciones de potencia. La interferencia de las partículas energéticas puede dañar componentes no diseñados para soportar altas tensiones o corrientes.

- Single Event Latch-up: Los eventos de latch-up suceden cuando una partícula cargada induce una corriente excesiva en un dispositivo, provocando que entre en un estado de cortocircuito temporal o bloqueo (latch-up). Esto puede tener un impacto en la vida útil del dispositivo o, en casos extremos, dañar el dispositivo de forma permanente.

4.2 NO DESTRUCTIVOS

Los eventos no destructivos, comúnmente denominados “soft errors”, son eventos temporales o transitorios que no causan daño permanente al dispositivo o componentes, ni provocan daños permanentes en los datos almacenados en las memorias. Entre estos eventos, se incluyen:

- Single Event Upset (SEU): En un evento SEU, una partícula cargada (un neutrón o ión pesado) interactúa con una celda de memoria, induciendo una carga en ella que podría cambiar el estado uno o varios de sus transistores, pasando de un estado de saturación (‘1’) a corte (‘0’) o viceversa. Esto puede provocar cambios temporales en la información almacenada en la memoria, pero son cambios reversibles y no causan daños permanentes.

- Single Event Transient (SET): En un evento SET, una partícula cargada induce una perturbación temporal en la operación de un dispositivo. Puede generar uno o más impulsos de voltaje que se propaguen a través del circuito. Sin embargo, una vez la partícula ya lo ha atravesado, esta perturbación desaparece, por lo que no causa daños permanentes. Si la perturbación resulta en un valor incorrecto en una unidad secuencial, se considera un evento SEU.

- Single Event Functional Interruption (SEFI): Un evento SEFI ocurre cuando una partícula cargada induce una interrupción temporal en el funcionamiento del dispositivo, lo que puede causar un reinicio, bloqueo (lock-up) u otro comportamiento defectuoso. Esto conduce a una pérdida temporal de funcionalidad. Por lo general, el dispositivo vuelve a funcionar normalmente una vez el evento ha cesado.

Como se puede apreciar, la radiación cósmica puede dar lugar a diversos eventos que afectan a los circuitos electrónicos. En este trabajo, nos enfocaremos únicamente en el desarrollo y aplicación de algoritmos diseñados para abordar los Single Error Upsets (SEUs) en memorias SRAM para uso espacial.

5. MÉTODOS PARA LA CORRECCIÓN DE EVENTOS TRANSITORIOS

5.1 ALGORITMOS PARA LA CORRECCIÓN DE ERRORES (ECC)

Los algoritmos para la corrección de errores (ECC) son técnicas utilizadas para detectar y corregir errores que pueden ocurrir durante la transmisión o almacenamiento de datos digitales.

Existen varios tipos diferentes de corrección de errores, algunos de los más comunes incluyen [12] [13]:

Comprobación de la paridad (parity checking): Esta técnica implica la adición de un bit extra, conocido como bit de paridad, al dato original. Su función es garantizar que el número de '1' lógicos en el dato sea siempre par o impar. El bit de paridad se utiliza para detectar errores de un solo bit (single-bit errors), pero no tiene la capacidad de corregirlos.

Código Hamming (Hamming code): En esta técnica, se añaden bits de paridad extra al dato, basándose en la longitud del mismo. Estos bits de paridad permiten tanto la detección como la corrección de errores de un solo bit (single-bit errors).

Código Hamming extendido (Extended Hamming code): Similar al código Hamming, esta versión incluye un bit de paridad global adicional al principio del dato para verificar si el número de '1' lógicos es par o impar. A diferencia del código Hamming, el código Hamming extendido no solo puede detectar y corregir errores de un solo bit (single-bit errors), sino también tiene la capacidad de detectar dos errores de un solo bit (double-bit errors).

Verificación de redundancia cíclica (Cyclic redundancy check-CRC): Esta técnica consiste en calcular una suma de verificación o "checksum" basada en los bits del dato y los agrega al final del mismo. El "checksum" se utiliza para detectar errores y, si es posible, el dato puede ser reconstruido utilizando códigos de corrección de errores, como los códigos de Reed-Solomon.

Corrección de errores hacia delante (Forward error correction- FEC): En esta técnica, se añaden bits extra al dato, denominados bits de redundancia, que permiten la reconstrucción del dato original si se detecta un error. Existen varios tipos de códigos FEC, como los códigos convolucionales y los códigos de bloques, que utilizan diferentes formas de redundancia.

Estos métodos desempeñan un papel crucial en garantizar la integridad de los datos en sistemas de almacenamiento o comunicaciones inalámbricas. Cada uno de ellos tiene sus propias ventajas y desventajas dependiendo de los requisitos a cumplir del diseño y su propensión a sufrir errores, dependiendo del entorno donde vaya a realizar su función.

En una de las memorias SRAM a diseñar en el presente trabajo, nos centraremos en la corrección de errores mediante el algoritmo "codificación Hamming" y su mejora "codificación Hamming extendida".

6. DESCRIPCIÓN DE AMBOS MÉTODOS DE CORRECCIÓN DE ERRORES

6.1 CÓDIGO HAMMING

Richard Wesley Hamming (1915-1998) fue un destacado matemático estadounidense pionero de la cibernética y la teoría de la comunicación. Su carrera comenzó graduándose en Matemáticas en la Universidad de Chicago en 1937 y posteriormente doctorándose, también en Matemáticas, en la Universidad de Illinois en 1942.

Durante la Segunda Guerra Mundial, trabajó en Los Álamos en el marco del proyecto Manhattan, proyecto de investigación del gobierno americano con el objetivo de desarrollar la primera bomba atómica.

Después de finalizar la guerra, se incorporó al Departamento de Matemáticas de Bell Telephone en Murray Hill, New Jersey. Fue en ese entorno donde desarrolló la teoría de codificación que más tarde se conocería como "Código Hamming". Su motivación se impulsaba por mejorar la calidad de las comunicaciones telefónicas y la detección de errores en computadoras, lo que le llevó a realizar uno de los descubrimientos más importantes en la teoría de la comunicación y de la informática.

El "Código Hamming" permitió no solo identificar, sino también corregir un bit erróneo presente en un código binario compuesto por unos y ceros. Esta innovación revolucionó la forma de manejar y proteger los datos en sistemas de comunicación y computadoras [14].

6.1.1 DESARROLLO DEL ALGORITMO

El código Hamming funciona para cualquier longitud de bits de datos. Sin embargo, con el fin de simplificar la explicación, se detallará su versión más básica, el código Hamming (7,4).

Supóngase que se quiere almacenar una palabra de 4 bits, por ejemplo, la palabra "1011" en cualquier dirección de una memoria SRAM.

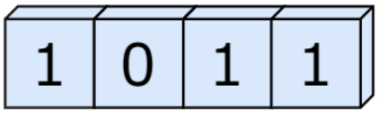

Ilustración 16. Ejemplo palabra de 4 bits

En primer lugar, el algoritmo propone el cálculo del número de bits de paridad que serán necesarios para la posterior detección y corrección del bit en caso de error, siguiendo la expresión:

$$2^p = (m + p) + 1 \qquad donde \qquad p \equiv numero\ total\ de\ bits\ de\ paridad$$
$$m \equiv numero\ total\ de\ bits\ de\ la\ palabra$$

Esta fórmula establece la relación entre el número de bits de paridad ($\boldsymbol{p}$) y el número total de bits en la palabra ($\boldsymbol{m}$) que se utilizará en el código. La elección de "$\boldsymbol{p}$" es fundamental para garantizar el funcionamiento del algoritmo.

Siguiendo la ecuación anterior, se obtienen tres bits de paridad, que se introducen en la palabra aumentando su tamaño a siete bits (por ello se denomina Hamming (7,4)). Con estos tres bits de paridad calculados, se pueden representar hasta 8 posiciones posibles, lo que es ideal debido a que se necesita un valor para representar la ausencia de error, cuatro valores para representar un error en cualquiera de los cuatro bits de datos y tres más para representar errores en cualquiera de los tres bits de paridad. Los tres bits de paridad nos proporcionan suficiente información para potencialmente corregir un "Single Event Upset" (SEU).

Cada bit de paridad cuenta la cantidad de '1' lógicos en la palabra original. Si se colocasen al final de esta, todos los bits de paridad tendrían el mismo valor, lo que no nos proporcionaría información útil. En cambio, cada bit de la palabra es utilizado para el cálculo de un único conjunto específico de bits de paridad. De esta manera, dependiendo de cuáles bits de paridad indiquen un error, se puede determinar la

posición donde ha ocurrido. Por ello, cabe definir en qué posiciones, de las 7 posibles en la presente palabra codificada, estarán ubicados los bits de paridad. Estas posiciones no pueden ser aleatorias, puesto que podría darse el caso de cualquier bit de paridad presente dependa del valor de otros bits de paridad restantes, lo que daría lugar a un enfoque incorrecto en la detección de errores. En otras palabras, se pretende que el cálculo de los bits de paridad se base sólo en los bits de la palabra y no intervenga el valor de otros bits de paridad. Para lograr esto, al desplegarse los 7 bits resultantes y describirse en binario cada una de sus posiciones formando una matriz, se determinó que los bits de paridad sean los bits que solo estén involucrados en una sola fila, de manera que no influyan entre sí, como se muestra a continuación:

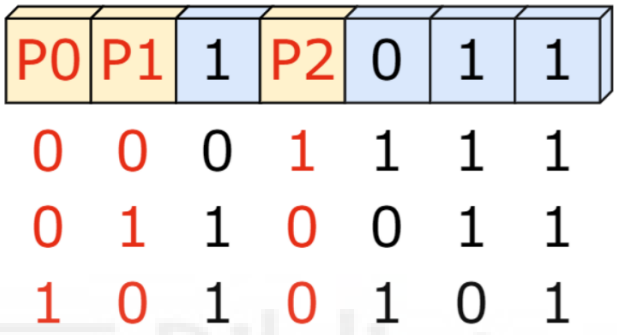

Ilustración 17. Colocación de bits de paridad

Se puede observar que cada número tiene una representación única, lo que servirá como guía para el cálculo de cada bit de paridad. Para el primer bit de paridad (P0), se calculará basándose únicamente en los bits que tienen un '1' en la primera fila de la matriz. Para el siguiente bit de paridad (P1), se calculará basándose solamente en los bits que tienen un '1' en la segunda fila de la matriz, y así sucesivamente para los siguientes bits de paridad (P2, P3, P4… Pn).

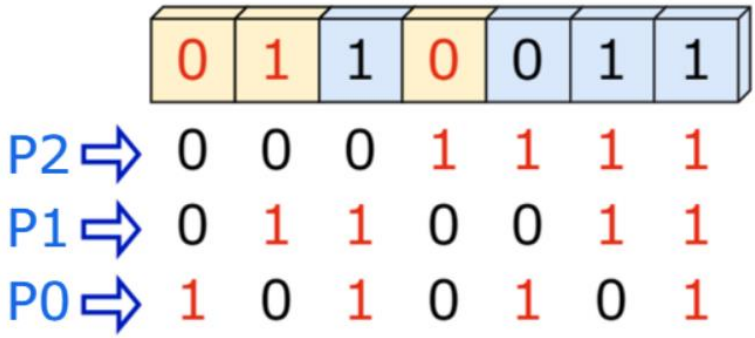

Ilustración 18. Cálculo de bits de paridad

Estas posiciones de los bits de paridad se pueden generalizar mediante la expresión:

$$2^{P_n} \qquad P_n \equiv bit\ de\ paridad\ (n = 0\ to\ (p-1))$$
$$p \equiv numero\ total\ de\ bits\ de\ paridad$$

En el ejemplo, la colocación de los bits de paridad es la siguiente: P0 en la posición uno, P1 en la posición dos y P3 en la posición cuatro, como muestra la **Figura 1**, y sus valores son 0, 1 y 0 respectivamente, como muestra la **Figura 2**.

¿Qué ocurre cuando uno de los bits ha cambiado?

Lo primero que debe observarse es cuáles de los bits de paridad indican un error. Si el valor de cada bit de paridad no coincide con los correspondientes datos de la palabra original utilizados para calcular dicho bit en la codificación, significa que se ha producido un "Single Event Upset" (SEU). Supóngase que el bit en la posición cinco del ejemplo ha cambiado:

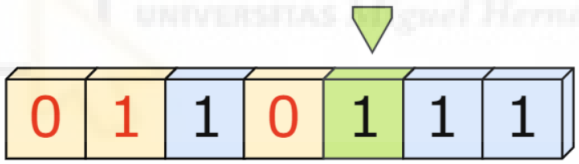

Ilustración 19. "Bit flip" provocado por SEU

Si se vuelve a calcular cada bit de paridad y se compara con los de la palabra codificada, se puede determinar que los bits de paridad uno y tres ya no coinciden, por lo que si se observa qué bits engloban ambas paridades erróneas (considerando que dicho bit erróneo no se encuentra incluido en el cálculo del bit de paridad dos, pues no ha variado) se concluye que es en la posición cinco donde ha ocurrido el error, cuya representación en binario es "101".

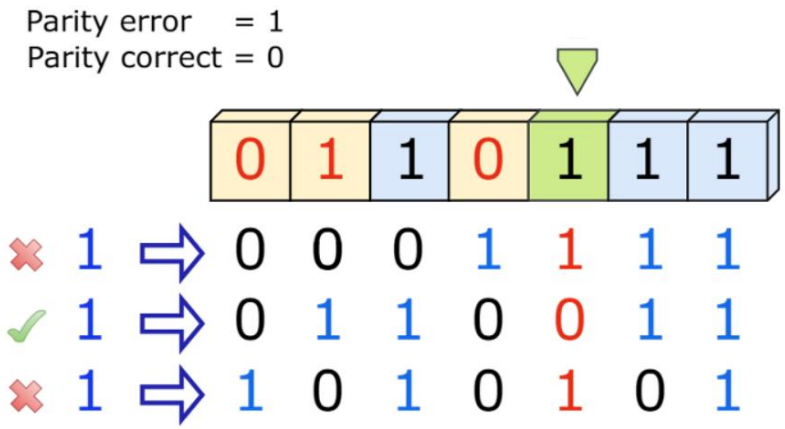

Ilustración 20. Detección del bit erróneo

Una vez detectado, simplemente invirtiendo ese bit, se habrá corregido el error de la palabra.

El código Hamming se vuelve más eficiente a medida que el número de bits en la palabra aumenta. Esto se debe a que, a medida que el tamaño de la palabra es mayor, el número de bits de paridad adicionales necesarios para realizar la detección y corrección de errores disminuye proporcionalmente en comparación con el tamaño total de la palabra. El resultado es un ahorro de memoria significativo, ya que el costo en términos de eficiencia en la utilización de la memoria es relativamente bajo en comparación con otras técnicas de redundancia o corrección de errores que requieren duplicar o triplicar la cantidad de datos almacenados para lograr una corrección efectiva. Por ejemplo, si se tiene una palabra de tamaño 1024 bits tan solo se necesitarían 10 bits de paridad para ser capaz de corregir cualquier "Single Error Upset" (SEU) posible [15].

6.2 VERSIÓN EXTENDIDA (EXTENDED HAMMING CODE)

Hasta ahora, al aplicar el código Hamming se obtenía una palabra con número impar de bits (como en el caso de Hamming (7,4), Hamming (15,11), etc…). Sin embargo, es ideal y más conveniente trabajar con palabras cuyo tamaño sea una potencia de dos. Para lograr esto, existe una forma de añadir un bit adicional, utilizándose como bit de paridad global de la palabra. De ese modo se tiene la capacidad de detectar, aunque sin posibilidad de corregir, dos bits afectados por un "Single Error Upset" (SEU).

6.2.1 DESARROLLO DEL ALGORITMO

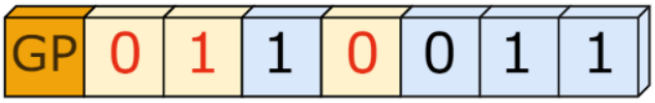

Ilustración 21. Colocación de bit de paridad global

Siguiendo el ejemplo anterior, donde se tenía la palabra compuesta por “1011”, cuya codificación Hamming resultante equivale a “0110011”, el nuevo bit de paridad global se calcula al contar la cantidad de ‘1’ lógicos presentes en la palabra codificada, determinando si es par o impar. Finalmente se coloca este bit en la posición cero, como se muestra en la figura:

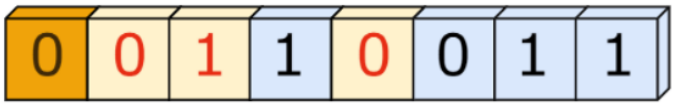

Ilustración 22. Cálculo de bit de paridad global

De esta manera, al produce un error en uno de sus bits la paridad global cambiará su valor, pero el resto de bits de paridad detectarán el error y lo corregirán. Por el contrario, al producirse un error en dos bits diferentes, el valor de paridad global permanecerá intacto, mientras el resto de los bits de paridad continuarán detectando que se ha producido un error en alguna posición de la palabra. Dado este último caso, el algoritmo indicará que efectivamente se han producido errores en dos bits de la palabra [16].

7. TRIPLE REDUNDANCIA MODULAR (TMR)

La Triple Redundancia Modular surge como sistema para combatir errores inducidos por radiación, dotando de mayor fiabilidad y tolerancia a fallos en sistemas críticos.

Este modelo de corrección de errores consiste en la instanciación de tres réplicas de la misma acción, es decir, tres componentes idénticos que realicen la misma operación en paralelo, de esta forma se obtiene la triple redundancia, y posterior implementación de un sistema de votación por mayoría el cual, en caso de que uno de los módulos falle, permita al resto seguir funcionando y enmascarar el evento ocurrido hasta ser resuelto posteriormente.

En una memoria SRAM con corrección de errores TMR el sistema de votación recibe, al momento de obtener una instrucción de lectura, los datos provenientes de cada una de las instancias. En caso de una de ellas haber sufrido uno o varios eventos singulares, permite que el dato de salida sea el más veces repetido, por lo que el error en la posición de memoria de dicho módulo fallido quedará enmascarado (sin posibilidad de corrección). No obstante, si dos de ellas han sufrido uno o varios eventos singulares, el sistema de votación no será capaz de identificar el dato original, debido a la no existencia de redundancia. Este caso se denomina "obsoleto" (deprecated).

Para solventar el problema, se tomarán las medidas que se consideren oportunas en cuanto al diseño para informar al usuario de este suceso o para la corrección del error mediante la implementación de códigos de corrección de errores, sobreescritura de la memoria, etc [17] [18].

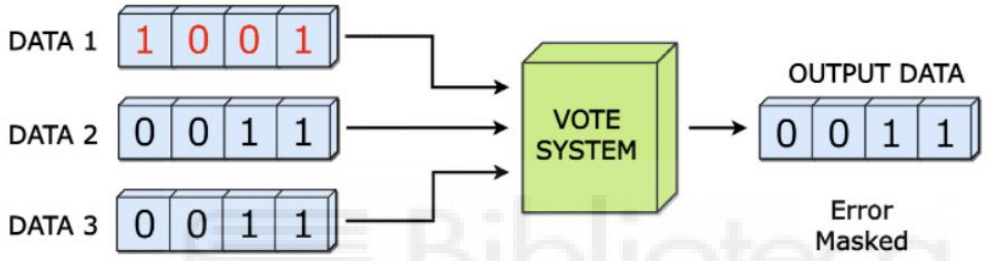

Ilustración 23. Diagrama TMR con error enmascarado

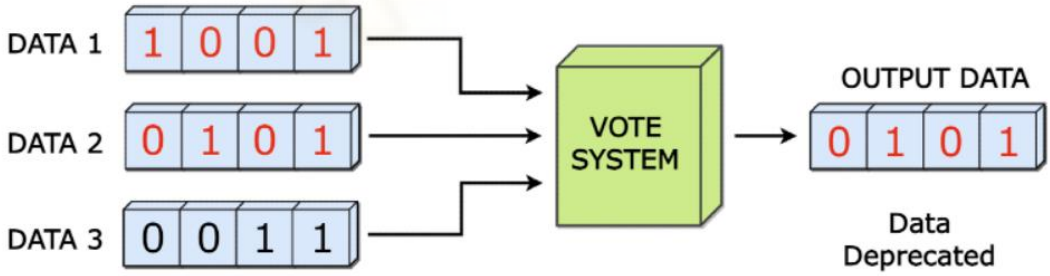

Ilustración 24. Diagrama TMR con dato obsoleto

8. SOFTWARE Y ENTORNO DE TRABAJO

El diseño y verificación de ambas memorias SRAM con diferentes sistemas de corrección de errores se realizará en el entorno de programación hardware VHDL (VHSIC Hardware Descriptive Language), a través del software Visual Studio Code

8.1 DISEÑO

En primer lugar, se procede a la creación de un entorno virtual mediante el software "Anaconda Navigator" (https://www.anaconda.com/download) . Una vez dentro, en el menú de opciones izquierdo, se selecciona la opción "Environments" y posteriormente en la parte inferior de la ventana, la opción "Create".

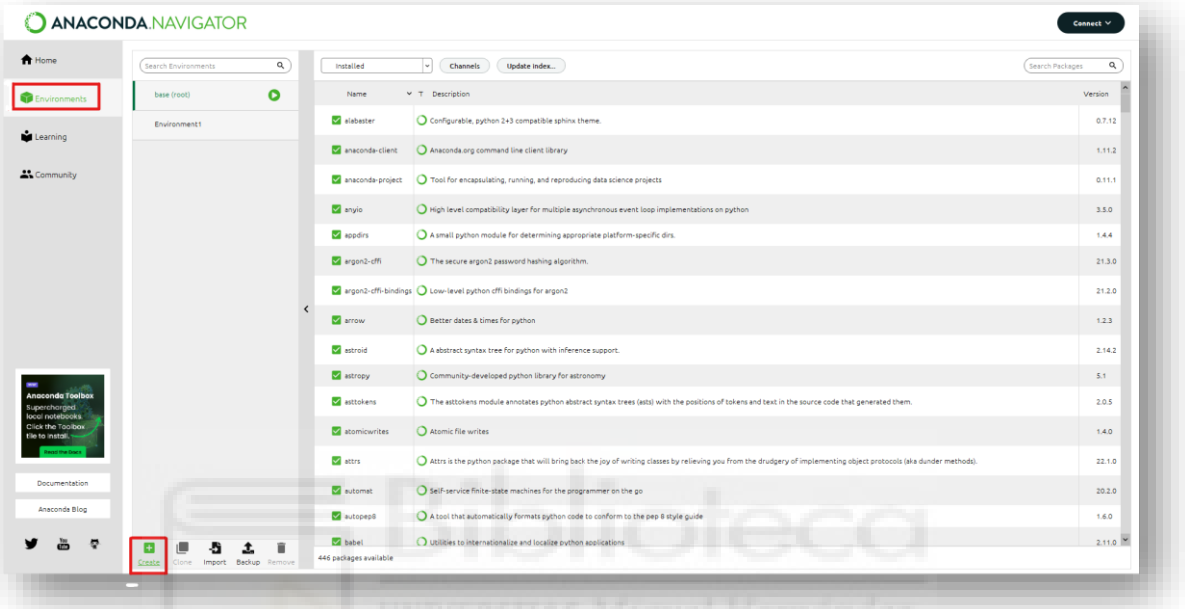

Ilustración 25. Creación de entorno virtual (1)

A continuación, aparecerá la ventana donde introducir el nombre y la versión de Python a utilizar. Una vez introducidos, se pulsa sobre "Create" y el entorno virtual se creará.

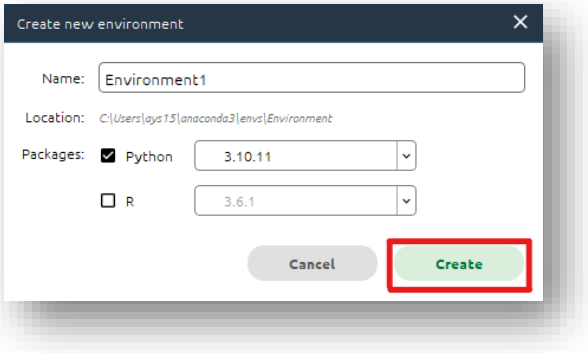

Ilustración 26. Creación de entorno virtual (2)

El siguiente paso será descargar el software editor de código fuente “Visual Studio Code” (https://code.visualstudio.com/download) . Una vez instalado, en la barra de ajustes izquierda, se selecciona el icono “Extensiones”. Sobre la barra de búsqueda superior se introduce “VHDL” y aparecerá una serie de extensiones de soporte para dicho lenguaje de programación, de las cuales bastará con instalar la primera “VHDL” .

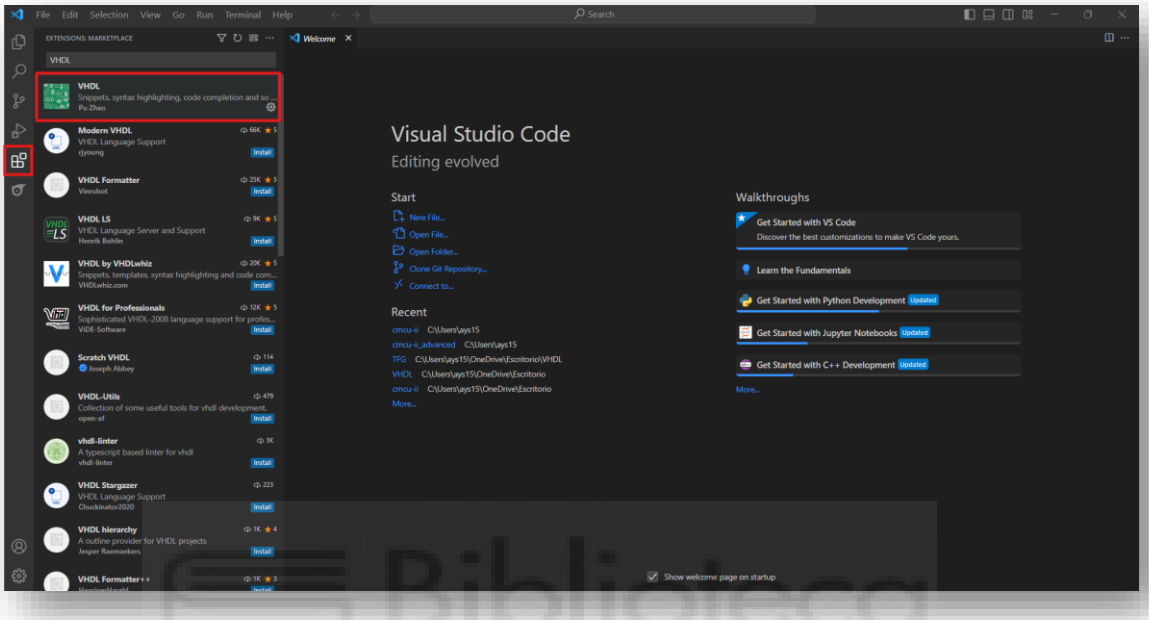

Ilustración 27. Configuración de entorno en VS Code (1)

Una vez instalada la extensión, mediante el atajo “cntrl+shift+p” se accede a la opción “Python: Select Interpreter” y se selecciona el entorno de programación creado previamente.

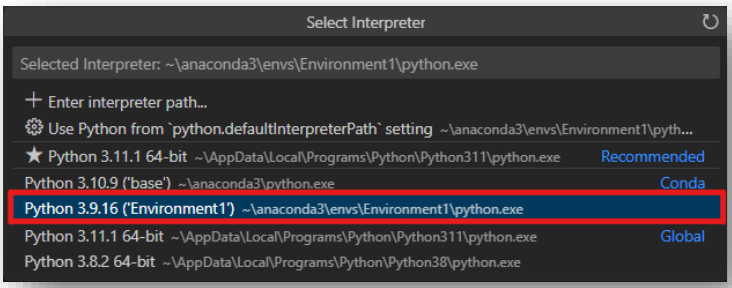

Ilustración 28. Configuración de entorno en VS Code (2)

Se abre una nueva terminal de comandos, mediante la opción “Terminal” en la parte superior de la ventana, donde se activará el entorno virtual a través del atajo “cntrl+shift+ñ”.

Por último, se selecciona el icono “Explorer > Open Folder” de la barra de herramientas en la parte izquierda de la ventana, donde se introducirá la carpeta en la que se pretenda elaborar y guardar los archivos de programación.

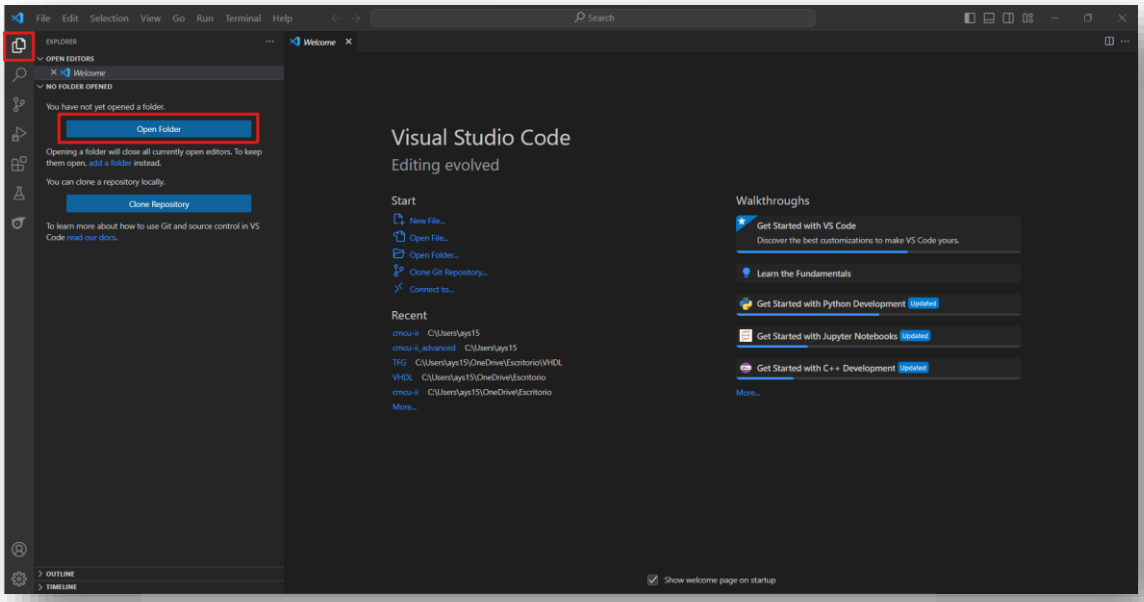

Ilustración 29. Configuración de entorno en VS Code (3)

8.2 VERIFICACIÓN

La verificación de los diseños se realizará utilizando la librería UVVM (Universal VHDL Verification Metodology), una metodología abierta al público de verificación para VHDL utilizada en la industria del diseño de hardware. Éste, proporciona un framework que dispone de una serie de bibliotecas y utilidades, con el fin de facilitar la comprobación del comportamiento de los diseños digitales en VHDL. Junto a ella, también se utilizará “vunit-hdl”, otro framework destinado a facilitar la realización de pruebas en el desarrollo de hardware digital, el cual permite la utilización de scripts en lenguaje de programación Python para la compilación de librerías y diseños, así como la utilización de softwares externos que permitan la visualización de las formas de onda.

En primer lugar, se instalará “vunit-hdl” en el entorno virtual previamente creado. Para ello, se abre una terminal de comandos mediante la opción “Terminal” en la barra superior de la ventana de Visual Studio Code, mediante el atajo “cntrl+shift+ñ” se activa el entorno virtual y con la instrucción “pip install vunit-hdl” se procede a la instalación.

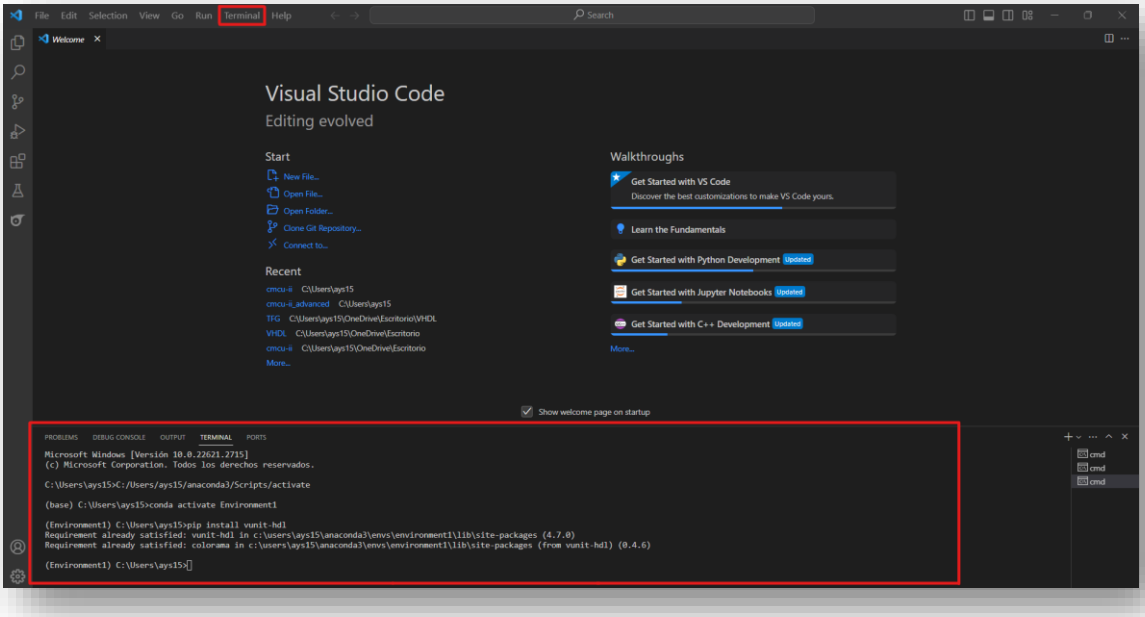

Ilustración 30. Configuración de entorno en VS Code (4)

Una vez instalado, se realizará la misma operación con la librería UVVM (https://github.com/UVVM/UVVM), la cual se descarga desde la web a través de un repositorio de GitHub, una plataforma de desarrollo colaborativo basado en Git (sistema de registro de cambios y control de versiones), seleccionando la pestaña "Code > Download ZIP".

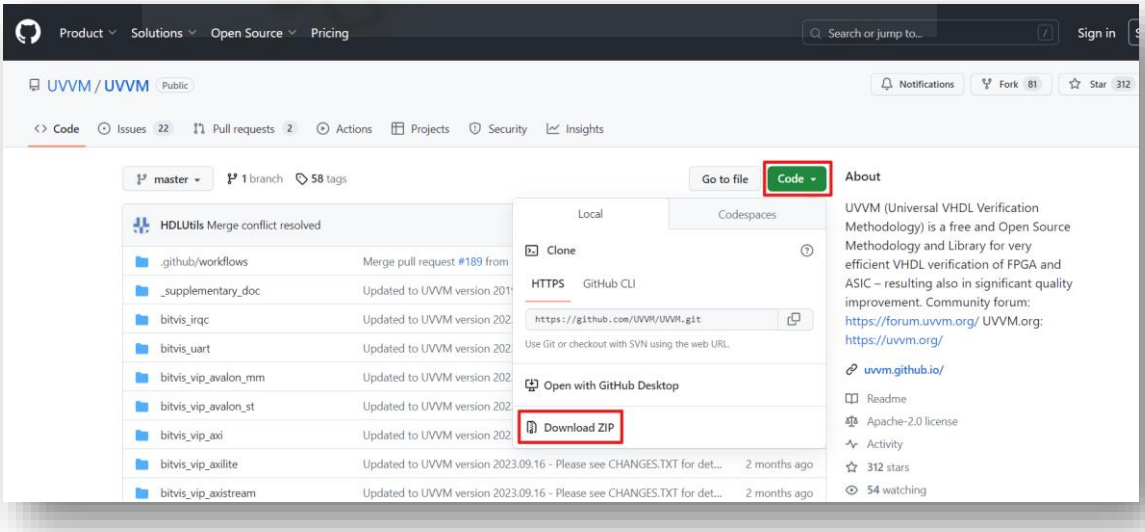

Ilustración 31. Instalación de UVVM (1)

Por último, se utilizará el software de visualización abierto al público ModelSim (https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html), el cual se usará para mostrar por pantalla

las formas de onda resultantes. Una vez descargado, se compila la librería UVVM a través de ModelSim, utilizando la ventana de comandos de Visual Studio Code, accediendo a la carpeta interna "UVVM-master/script" y ejecutando el archivo "compile_all.do" con el comando "modelsim -do compile_all.do".

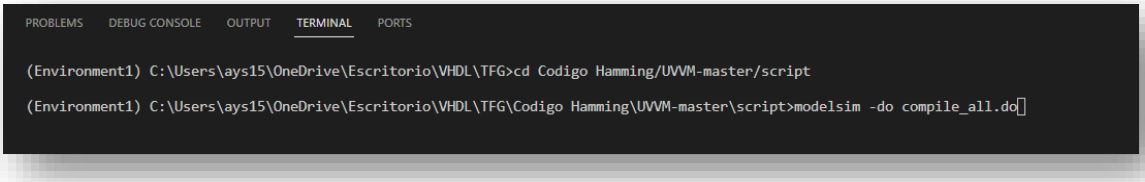

Ilustración 32. Instalación de UVVM (2)

8.2 SÍNTESIS

Para la síntesis de los diseños, se ha utilizado el software de Xilinx Vivado 2022.2 (https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2022-2.html), dedicado para la síntesis de cualquier FPGA (Field Programable Gate Array) fabricada por Xilinx. En él, se crea un nuevo proyecto mediante la opción "Create Proyect".

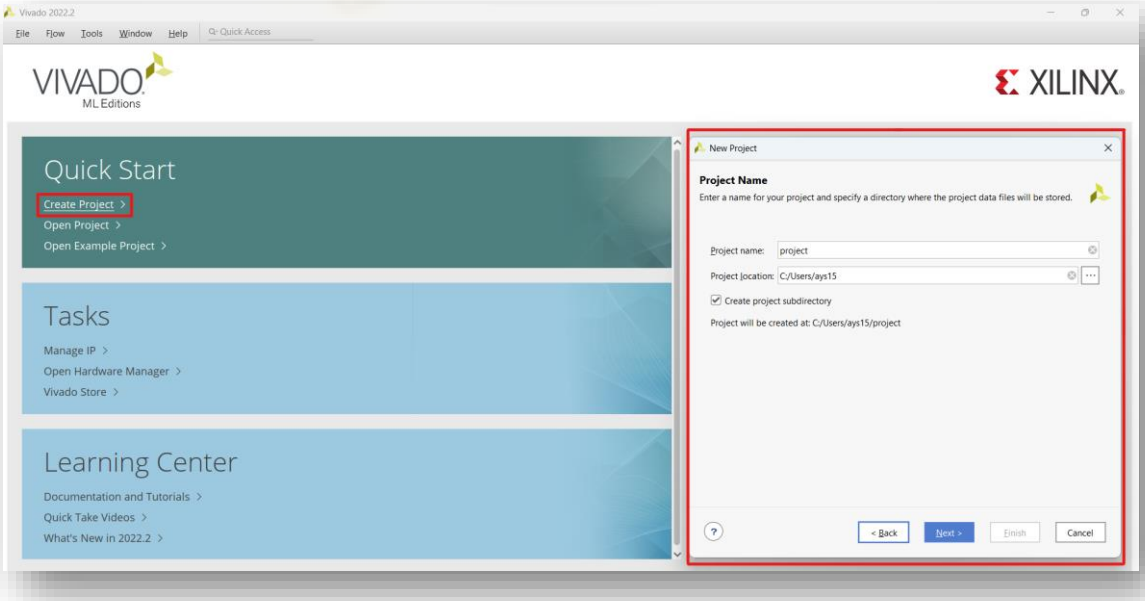

Ilustración 33. Creación de proyecto en Xilinx Vivado (1)

Se marca la casilla “RTL Project”, puesto que la opción adecuada para la implementación y síntesis de los diseños en hardware real.

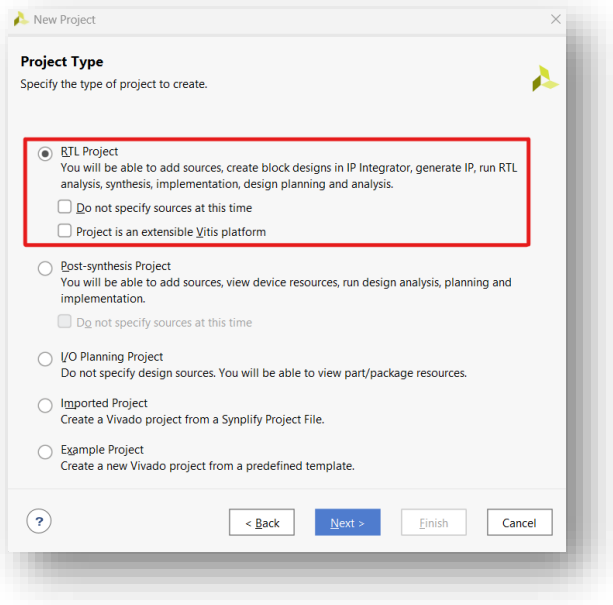

Ilustración 34. Creación de proyecto en Xilinx Vivado (2)

El siguiente paso trata de añadir los archivos de los diseños seleccionando “Add Files” , o directorios que contengan los archivos de los diseños seleccionando “Add Directories”. Es importante que en la casilla “Target Language” aparezca la instrucción “VHDL”, ya que es el lenguaje en el que se pretende sintetizar los diseños. Puesto que no se simulará mediante este software, el cual también permite la creación de archivos de verificación y posee un visualizador de formas de onda, la instrucción en la casilla “Simulator Language” no será relevante.

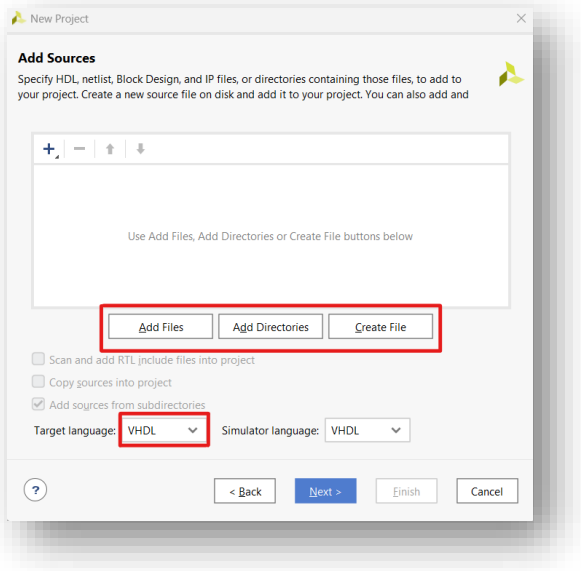

Ilustración 35. Creación de proyecto en Xilinx Vivado (3)

Finalmente, se elige la FPGA en la que se desea implementar el diseño, en este caso la Spartan 7 FPGA de Xilinx, se asigna uno de los paquetes disponibles y para concluir la creación del proyecto, se selecciona la opción “Finish”.

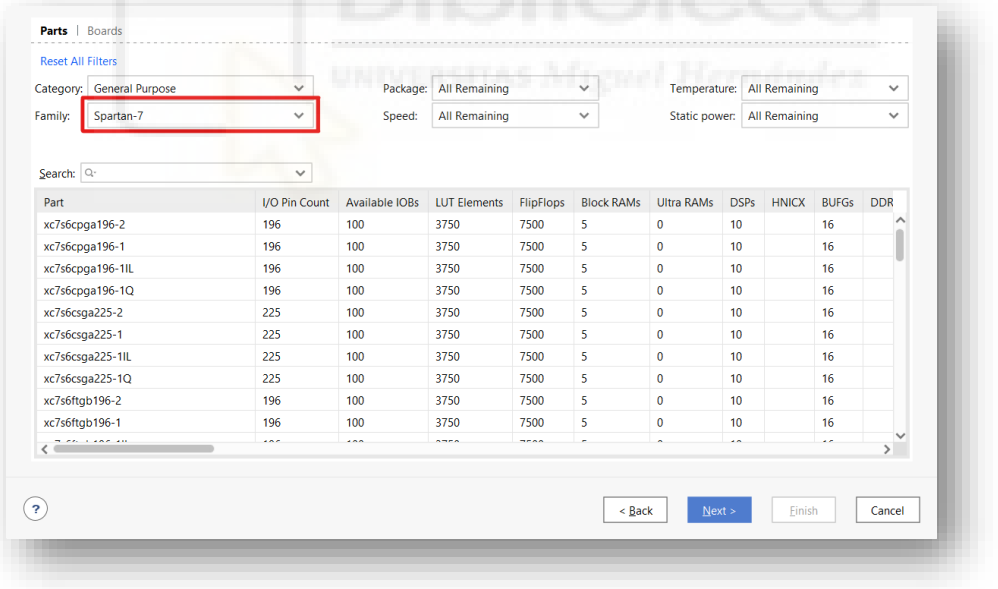

Ilustración 36. Creación de proyecto en Xilinx Vivado (4)

Una vez el diseño esté completo y listo para la síntesis, con el proyecto abierto, se selecciona “Run Synthesis” en la barra de herramientas situada en la parte izquierda de la ventana.

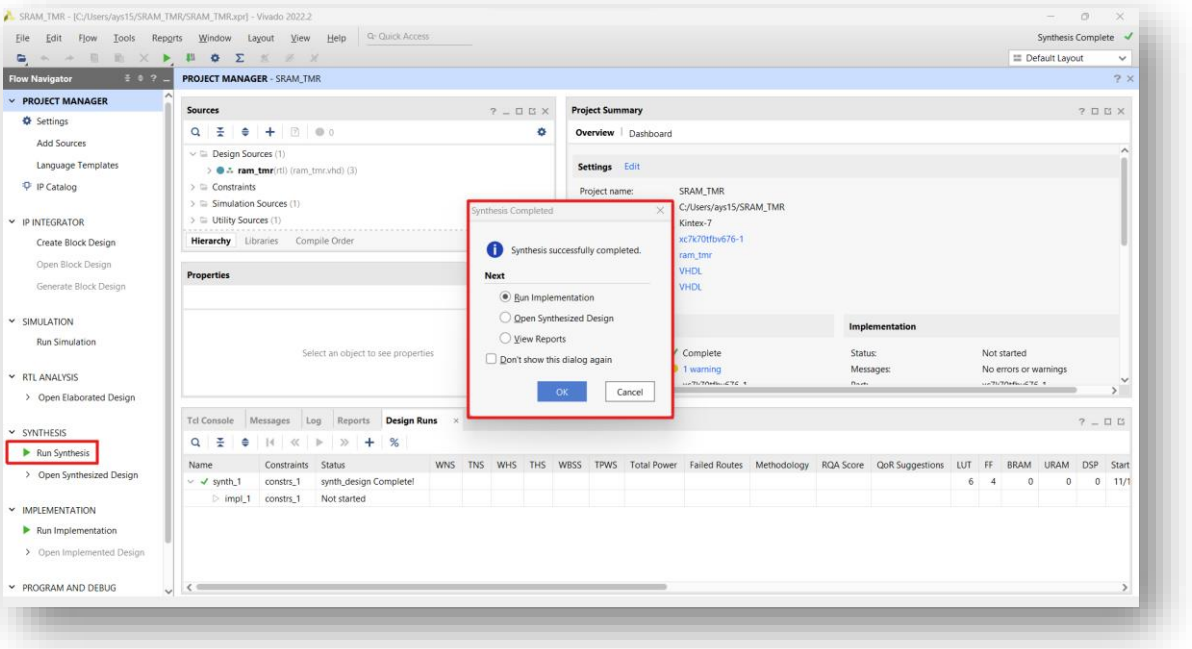

Ilustración 37. Sintetización de diseño en Xilinx Vivado

9. IMPLEMENTACIÓN CÓDIGO HAMMING EN VHDL

Para la implementación del código Hamming en VHDL de una forma más entendible y ordenada, se ha decidido dividir el proceso de codificación en varias funciones, cada una englobando un conjunto de operaciones específicas diferentes. Dichas funciones son las siguientes:

Nota : Los siguientes fragmentos de código se mostrarán mediante pseudocódigo, debido a la confidencialidad del archivo original, cuyos derechos pertenecen exclusivamente a Intigia S.L.

- **dec_parity_bits:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra sin codificar, calcula y devuelve la cantidad de bits de paridad utilizando la ecuación:

$$2^p = (m + p) + 1$$

Código:

```
Funcion dec_parity ← dec_parity_bits(tamaño_palabra)
    Para bit ← 1 hasta tamaño_palabra
        Si 2↑(bits_de_paridad) < (tamaño_palabra + bits_de_paridad) + 1 Entonces
            bits_de_paridad ← bits_de_paridad + 1
        Fin Si
    Fin Para
    dec_parity ← bits_de_paridad
Fin Funcion
```

Ilustración Código 1. Pseudocódigo función dec_parity_bits

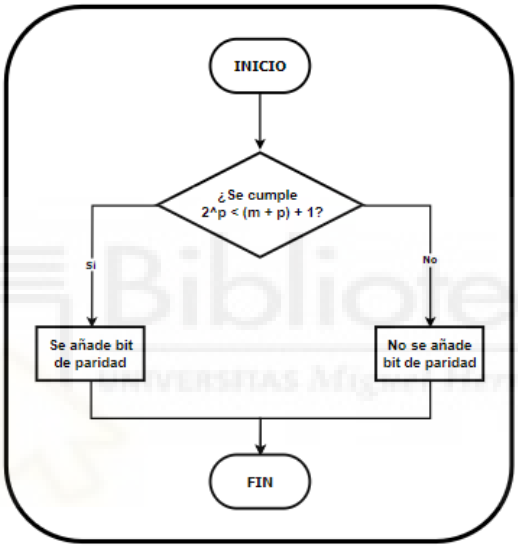

Ilustración Código 2. Diagrama de flujo función dec_parity_bits

La función consiste en recorrer las posiciones de la palabra, identificando aquellas que son potencias de 2 ($2^{0,1,2...n}$), ya que estos son los índices utilizados para establecer los bits de paridad. Finalmente, se suma la cantidad de veces que ocurre este evento, obteniendo el número total de bits de paridad de una palabra sin codificar.

- **enc_parity_bits:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra ya codificada, calcula y devuelve la cantidad de bits de paridad presentes.

Código:

```
Funcion enc_parity ← enc_parity_bits(tamaño_palabra_codificada)
    Para bit ← 1 hasta tamaño_palabra_codificada
        Si bit = posicion_bit_paridad Entonces
            bits_de_paridad ← bits_de_paridad + 1
        Fin Si
    Fin Para
    enc_parity ← bits_de_paridad
Fin Funcion
```

Ilustración Código 3. Pseudocódigo función enc_parity_bits

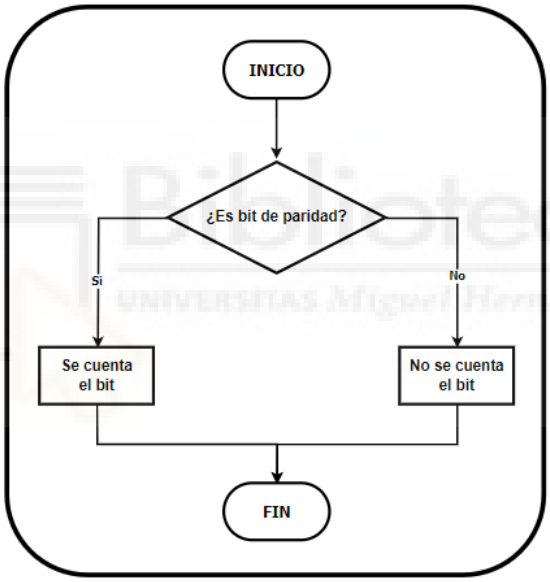

Ilustración Código 4. Diagrama de flujo función enc_parity_bits

La función consiste en recorrer las posiciones de la palabra codificada, identificando aquellas que son potencias de 2 $(2^{0,1,2\ldots n})$, ya que establecen los bits de paridad presentes en la codificación. Finalmente, se suma la cantidad de veces que ocurre este evento y se obtiene el total de bits de paridad de una palabra codificada.

- **dec_data_size:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra ya codificada y devuelve el tamaño de la correspondiente palabra decodificada.

Código:

```
Funcion dec_size ← dec_data_size(tamaño_palabra_codificada)
    numero_bits_paridad ← enc_parity_bits(tamaño_palabra_codificada)
    dec_size ← tamaño_palabra_codificada - numero_bits_paridad
Fin Funcion
```

Ilustración Código 5. Pseudocódigo función dec_data_size

La función consiste en restar la cantidad de bits de paridad al tamaño de la codificación original, obteniendo así el tamaño resultante de la palabra decodificada.

- **ext_dec_data_size:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra ya codificada y devuelve el tamaño de la correspondiente palabra decodificada (Código Hamming Extendido).

```
Funcion ext_dec_size ← ext_dec_data_size(tamaño_palabra_codificada)
    numero_bits_paridad ← enc_parity_bits(tamaño_palabra_codificada)
    ext_dec_size ← tamaño_palabra_codificada - numero_bits_paridad - 1
Fin Funcion
```

Ilustración Código 6. Pseudocódigo función ext_dec_data_size

La función consiste en restar la cantidad de bits de paridad al tamaño de la codificación original, obteniendo así el tamaño resultante de la palabra decodificada (Código Hamming Extendido).

- **enc_data_size:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra sin codificar y devuelve el tamaño de la correspondiente palabra codificada.

Código:

```
Funcion enc_size ← enc_data_size(tamaño_palabra)
    numero_bits_paridad ← dec_parity_bits(tamaño_palabra)
    enc_size ← tamaño_palabra + numero_bits_paridad
Fin Funcion
```

Ilustración Código 7. Pseudocódigo función enc_size

La función consiste en recibir la cantidad de bits de paridad, dada por la función dec_parity_bits, y sumarlos al tamaño de bits de la palabra original, obteniendo así el tamaño total de la palabra codificada.

- **ext_enc_data_size:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe el tamaño de la palabra sin codificar y devuelve el tamaño de la palabra codificada (Código Hamming Extendido).

Código:

```
Funcion ext_enc_size ← ext_enc_data_size(tamaño_palabra)
    numero_bits_paridad ← dec_parity_bits(tamaño_palabra)
    ext_enc_size ← tamaño_palabra + numero_bits_paridad + 1
Fin Funcion
```

Ilustración Código 8. Pseudocódigo función ext_enc_size

La función consiste en recibir la cantidad de bits de paridad, dada por la función dec_parity_bits, y sumarlos al tamaño de bits de la palabra original, obteniendo así el tamaño total de la palabra codificada (Código Hamming Extendido).

- **enc_data_ini:**

Archivo: hamming_pkg.vhd

Descripción: Esta función recibe la palabra sin codificar y devuelve un vector, de tamaño correspondiente a la palabra codificada, con los bits de datos en las posiciones correspondientes, dejando los bits de paridad inicializados a '0'.

Código:

```
Funcion enc_data_inicial[] ← enc_data_ini(palabra_decodificada)

    // Declaracion de vectores
    Dimension palabra_codificada[tamaño_palabra_codificada]

    // Inicializacion de vectores
    palabra_codificada ← 0

    // Desarrollo del algoritmo
    Para bit ← 0 hasta tamaño_palabra_codificada
        Si bit ≠ posicion_bit_paridad Entonces
            Si bit ≠ 0 Entonces
                palabra_codificada[bit] ← palabra_decodificada[tamaño_palabra_decodificada - contador]
                contador ← contador + 1
            Fin Si
        Sino
            posicion_bit_paridad ← posicion_bit_paridad + 1
        Fin Si
    Fin Para
    enc_data_inicial ← palabra_codificada
Fin Funcion
```

Ilustración Código 9. Pseudocódigo función enc_data_inicial

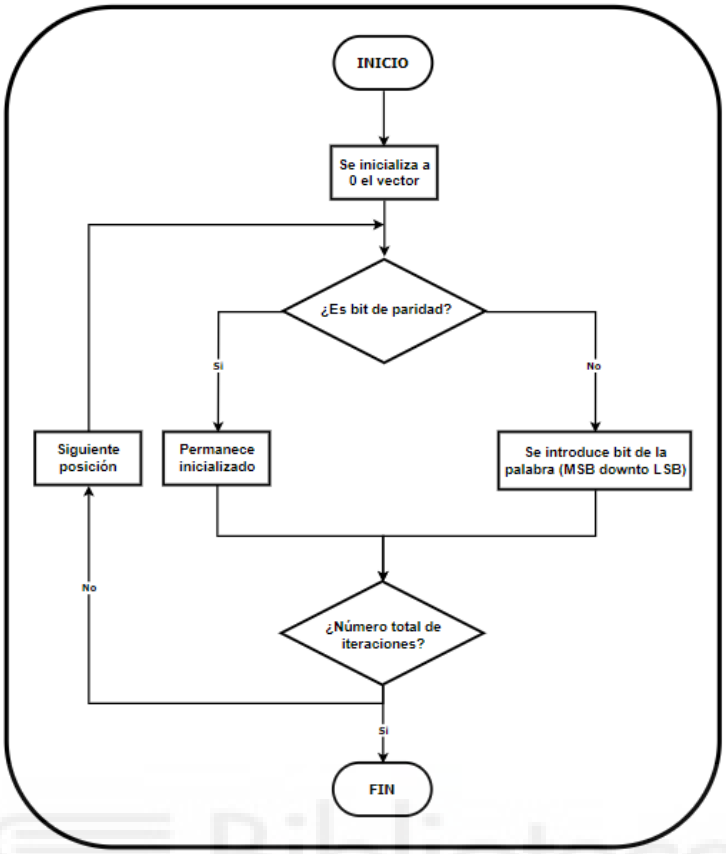

Ilustración Código 10. Diagrama de flujo función enc_data_ini

La función consiste en inicializar la palabra codificada, posteriormente recorrerla y rellenar las posiciones no correspondientes a bits de paridad con los bits de datos de la palabra original en orden descendente, dejando los bits de paridad inicializados a ‘0’.

- **dec_parity_result:**

Archivo: hamming_pkg.vhd

Descripción: Esta función calcula el valor de todos los bits de paridad (par/impar) de la palabra codificada y guarda los resultados en un vector.

Código:

```
Funcion parity_calc[] ← dec_parity_result (palabra_decodificada)

    // Declaración de vectores
    Dimension vector_filas[tamaño_palabra_codificada]
    Dimension vector_paridad[tamaño_numero_bits_paridad]
    Dimension vector_columnas[tamaño_numero_bits_paridad]
    Dimension palabra_codificada[tamaño_palabra_codificada]

    // Inicialización de vectores
    vector_paridad     ← 0
    palabra_codificada ← enc_data_ini(palabra_decodificada)

    // Desarrollo del algoritmo
    Para posicion ← 0 hasta tamaño_numero_bits_paridad
        Para i ← 0 hasta tamaño_palabra_codificada
            vector_columnas ← i_en_binario
            vector_filas[i] ← vector_columnas[posicion]
        Fin Para

        Para j ← 0 hasta tamaño_palabra_codificada
            Si vector_filas[j] = '1' Entonces
                vector_paridad[posicion] ← operacion_XOR(vector_paridad[posicion],palabra_codificada[j])
            Fin Si
        Fin Para
    Fin Para
    parity_calc ← vector_paridad
Fin Funcion
```

Ilustración Código 11. Pseudocódigo función dec_parity_calc

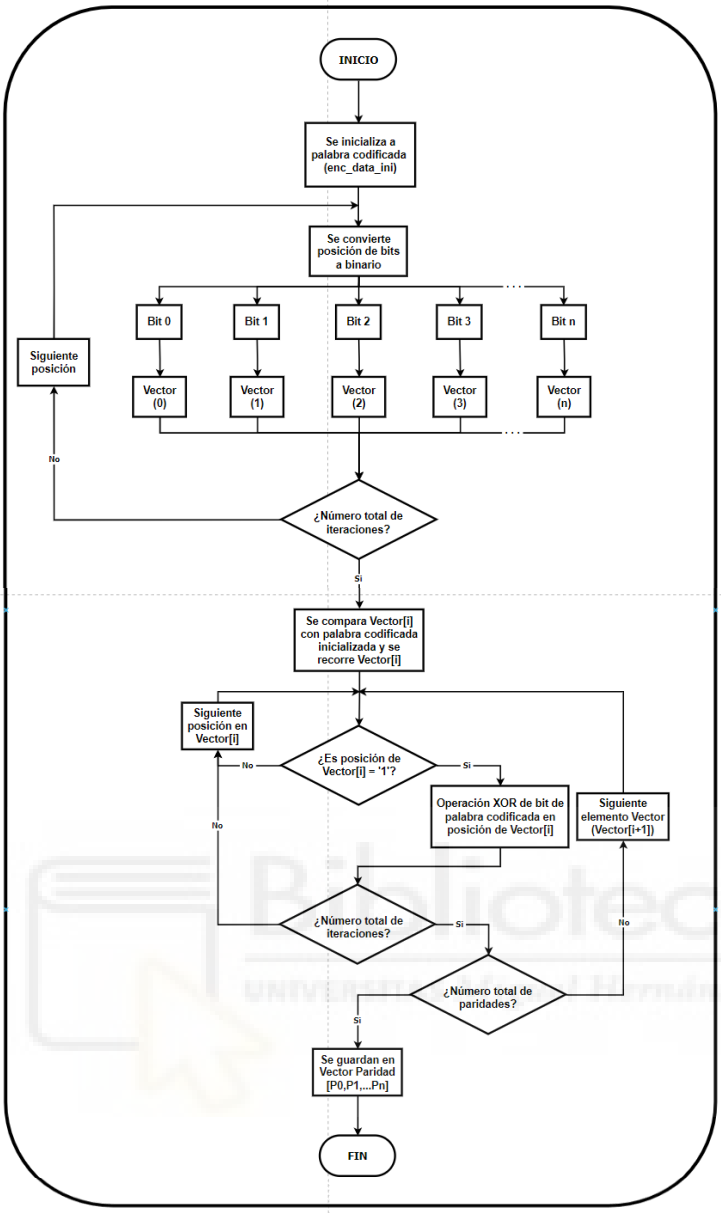

Ilustración Código 12. Diagrama de flujo función dec_parity_calc

Esta función realiza el cálculo de los bits de paridad de la palabra codificada, almacenándolos en un vector cuyo tamaño se ajusta según la cantidad de bits de paridad de la codificación.

En primer lugar, se genera la palabra codificada con únicamente los bits de paridad inicializados mediante la función “enc_data_ini”, la cual recibe la palabra decodificada como parámetro. A continuación, se busca generar las diferentes filas de bits que conforman el cálculo de cada bit de paridad, es decir, el patrón de bits de la palabra original que utiliza para el cálculo de su valor. Para lograr esto, se han creado dos vectores “vector_filas” y “vector_columnas”.

A través del uso de un bucle, se recorre desde cero hasta el máximo de posiciones de la palabra codificada. En cada iteración, se transforma el número correspondiente a la misma a binario y se almacena en “vector_columnas”. Además, se llena “vector_filas” con las posiciones de “vector_columnas” correspondientes a un contador implementado también como bucle, desde cero hasta el número de bits de paridad, el cual se incrementa una vez el bucle que recorre la palabra codificada ha finalizado. De esta forma, “vector_columnas” se irá sobrescribiendo cada nueva iteración, y su posición correspondiente con dicho contador se irá guardando en “vector_filas”, el cual también se sobrescribirá en cada nuevo acceso al bucle con sus respectivos nuevos valores.

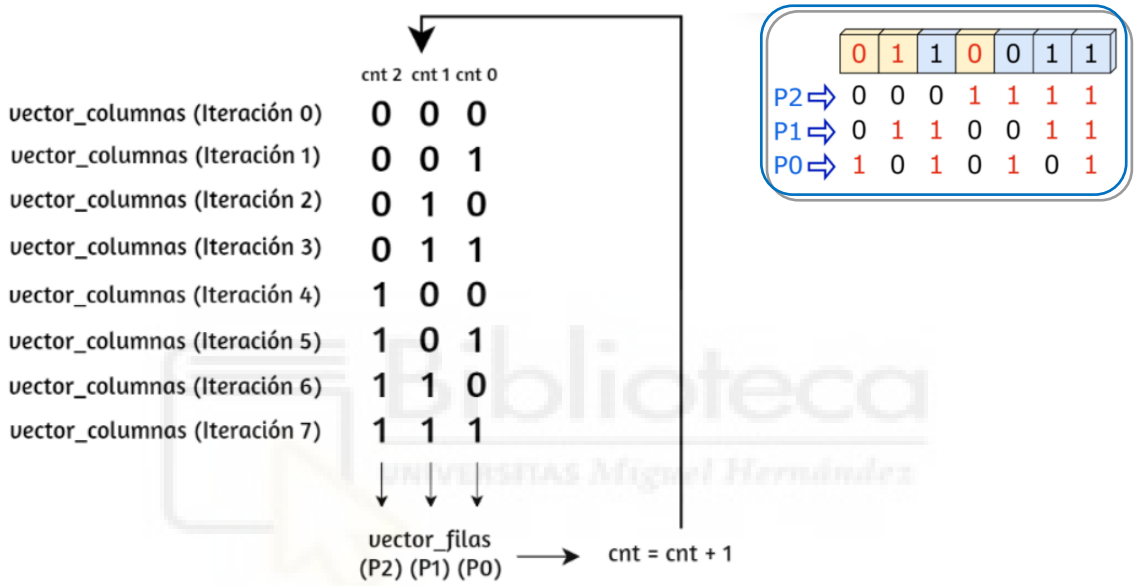

Ilustración 38. Ejemplo funcionamiento algoritmo para Código Hamming (7.4)

Finalmente, se itera sobre “vector_filas” y, en las posiciones donde se encuentra un ‘1’ lógico, se realiza el cálculo del correspondiente bit de paridad utilizando la operación lógica “XOR” en los bits de la palabra codificada en esa posición y los bits de un nuevo vector denominado “vector_paridad”. Este último, comienza inicializado a cero y va almacenando el resultado de las operaciones en las posiciones correspondientes al contador anterior.

- **global_parity:**

Archivo: hamming_pkg.vhd

Descripción: Esta función calcula y devuelve la paridad total de la palabra, incluyendo los bits de paridad (Código Hamming Extendido).

Código:

```
Funcion gb_parity ← global_parity(palabra_codificada)
    Para i ← 1 hasta tamaño_palabra_codificada
        paridad ← operacion_XOR(paridad,palabra_codificada[i])
    FinPara
    gb_parity ← paridad
FinFuncion
```

Ilustración Código 13. Pseudocódigo función global_parity

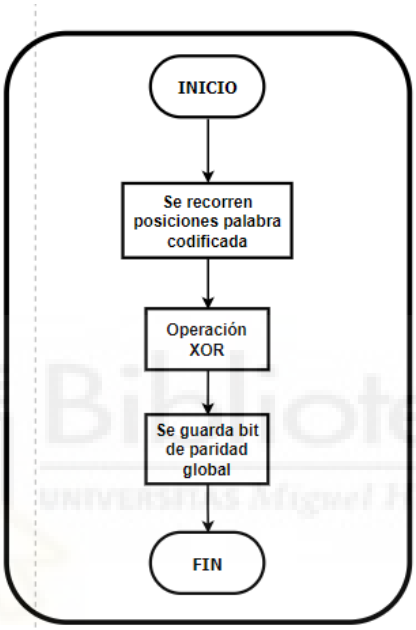

Ilustración Código 14. Diagrama de flujo función global_parity

Esta función se encarga de determinar el bit de paridad global, utilizado en el código Hamming extendido. Para ello, se recorre la palabra codificada y se realiza la operación "XOR" entre el bit actual y el resultado acumulado hasta el momento en la variable "paridad".

- **enc_parity_result:**

Archivo: hamming_pkg.vhd

Descripción: Esta función toma los bits de paridad de la palabra codificada y los guarda en un vector.

Código:

```
Funcion encoded_parity_calc[] ← enc_parity_result(palabra_codificada)

    // Declaración de vectores
    Dimension vector_paridad[tamaño_numero_bits_paridad]

    // Inicialización de vectores
    vector_paridad ← 0

    // Desarrollo del algoritmo
    Para i ← 0 hasta tamaño_palabra_codificada
        Si i = posicion_bit_paridad Entonces
            vector_paridad[contador] ← palabra_codificada[i]
            contador ← contador + 1
        FinSi
    FinPara
    encoded_parity_calc ← vector_paridad
FinFuncion
```

Ilustración Código 15. Pseudocódigo función enc_parity_result

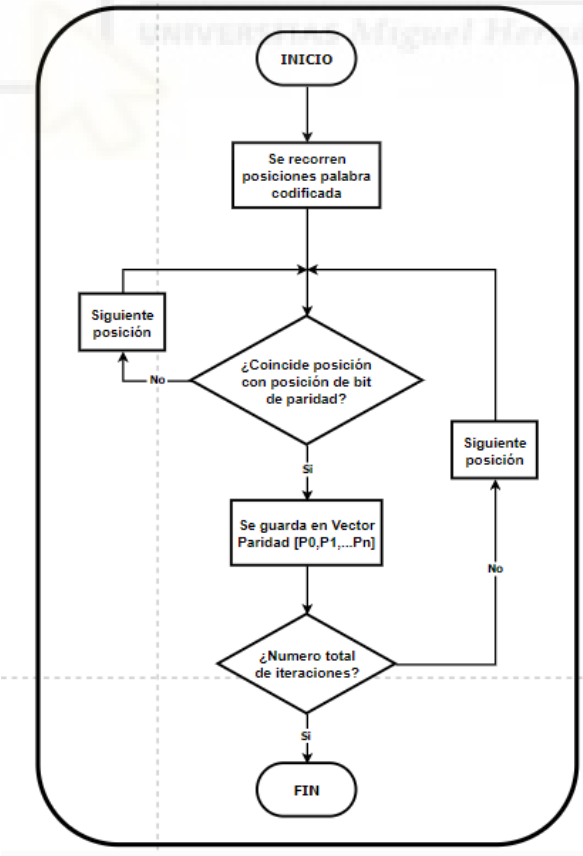

Ilustración Código 16. Diagrama de flujo función enc_parity_result

Esta función copia los bits de paridad de la palabra codificada en un vector denominado "vector_paridad". Se recorre la palabra con un bucle, si la iteración coincide con la posición de un bit de paridad, éste se copia en "vector_paridad" y el contador se incrementa. El tamaño de "vector_paridad" dependerá de cuantos bits de paridad tenga la palabra codificada.

- **ext_enc_parity_result:**

Archivo: hamming_pkg.vhd

Descripción: Esta función toma los bits de paridad de la palabra codificada y los guarda en un vector (Código Hamming Extendido).

Código:

```
Funcion ext_encoded_parity_calc[] ← ext_enc_parity_result(palabra_codificada)

    // Declaración de vectores
    Dimension vector_paridad[tamaño_numero_bits_paridad]

    // Inicialización de vectores
    vector_paridad ← 0

    // Desarrollo del algoritmo
    Para i ← 1 hasta tamaño_palabra_codificada
        Si i = posicion_bit_paridad Entonces
            vector_paridad[contador] ← palabra_codificada[i]
            contador ← contador + 1
        FinSi
    FinPara
    ext_encoded_parity_calc ← vector_paridad
FinFuncion
```

Ilustración Código 17. Pseudocódigo función ext_enc_parity_result

Esta función realiza la misma operación que la anterior para el código Hamming extendido.

- **hamming_enc:**

Archivo: hamming_pkg.vhd

Descripción: Esta función codifica la palabra original mediante el Código Hamming.

Código:

```
Funcion encoded_word[] ← hamming_enc(palabra_decodificada)

    // Declaración de vectores
    Dimension vector_paridad[tamaño_numero_bits_paridad]
    Dimension palabra_codificada[tamaño_palabra_codificada]


    // Inicialización de vectores
    palabra_codificada ← enc_data_ini(palabra_decodificada)
    vector_paridad     ← dec_parity_result(palabra_decodificada)

    // Desarrollo del algoritmo
    Para i ← 0 hasta tamaño_palabra_codificada
        Si i = posicion_bit_paridad Entonces
            palabra_codificada[i] ← vector_paridad[contador]
            contador ← contador + 1
        FinSi
    FinPara
    encoded_word ← palabra_codificada
FinFuncion
```

Ilustración Código 18. Pseudocódigo función hamming_enc

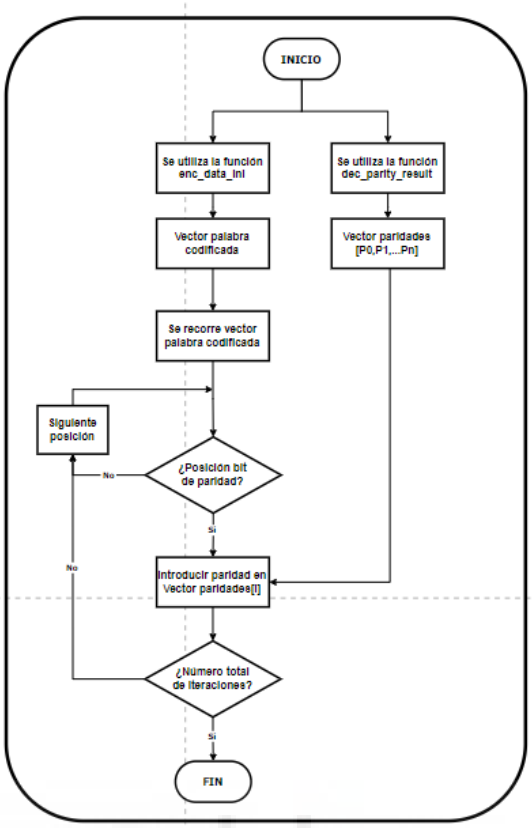

Ilustración Código 19. Diagrama de flujo función hamming_enc

Esta función toma la palabra original y la codifica, según el código Hamming, haciendo uso de dos vectores: “vector_paridad” y “palabra_codificada”.

En primer lugar, se inicializan dichos vectores, a través de la llamada a las funciones previamente descritas “enc_data_ini” y “dec_parity_result” respectivamente.

Posteriormente se tiene que “palabra_codificada” contiene la codificación con los bits de paridad inicializados a 0, mientras que “vector_paridad” contiene en orden los bits de paridad con sus valores ya calculados.

Finalmente, haciendo uso de un bucle, se ubican los bits de paridad contenidos en “vector_paridad” en las posiciones correspondientes de “palabra_codificada”. El resultado obtenido es la codificación completa de la palabra original.

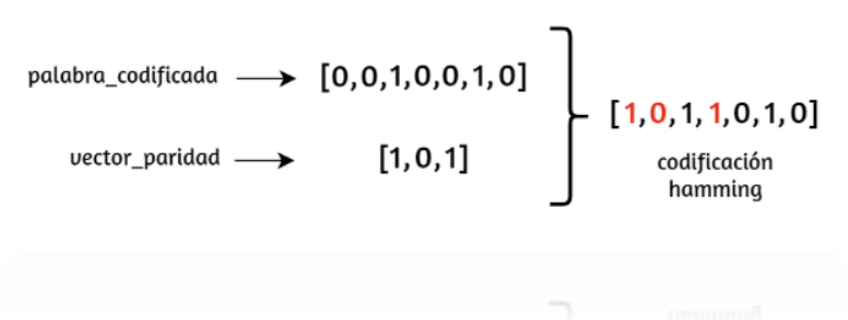

Ilustración 39. Ilustración algoritmo para Código Hamming (7,4)

- **ext_hamming_enc:**

Archivo: hamming_pkg.vhd

Descripción: Esta función codifica la palabra original (código Hamming extendido).

Código:

```
Funcion ext_encoded_word[] ← ext_hamming_enc(palabra_decodificada)

    // Declaración de vectores
    Dimension palabra_codificada[tamaño_palabra_codificada]

    // Desarrollo del algoritmo
    palabra_codificada[0] ← global_parity(hamming_enc(palabra_decodificada))
    palabra_codificada[resto_de_posiciones] ← hamming_enc(palabra_decodificada)
    ext_encoded_word ← palabra_codificada
FinFuncion
```

Ilustración Código 20. Pseudocódigo función ext_hamming_enc

Esta función es un añadido a la función "hamming_enc" para conseguir la codificación Hamming extendida. Consiste en guardar en la posición inicial del vector "palabra_codificada" el bit de paridad global, calculado utilizando la función "global_parity", y en el resto de posiciones se incluye la codificación hamming corriente de la palabra original, calculada utilizando la función "hamming_enc".

- **hamming_dec:**

Archivo: hamming_pkg.vhd

Descripción: Esta función decodifica la palabra codificada mediante el código Hamming.

Código:

```
Funcion decoded_word[] ← hamming_dec(palabra_codificada)

    // Declaración de vectores
    Dimension palabra_decodificada[tamaño_palabra_decodificada]

    // Desarrollo del algoritmo
    Para i ← 0 hasta tamaño_palabra_codificada
        Si i ≠ posicion_bit_paridad Entonces
            palabra_decodificada[tamaño_palabra_decodificada - contador] ← palabra_codificada[i]
            contador ← contador + 1
        Sino
            posicion_bit_paridad ← posicion_bit_paridad + 1
        FinSi
    FinPara
    decoded_word ← palabra_decodificada
FinFuncion
```

Ilustración Código 21. Pseudocódigo función hamming_dec

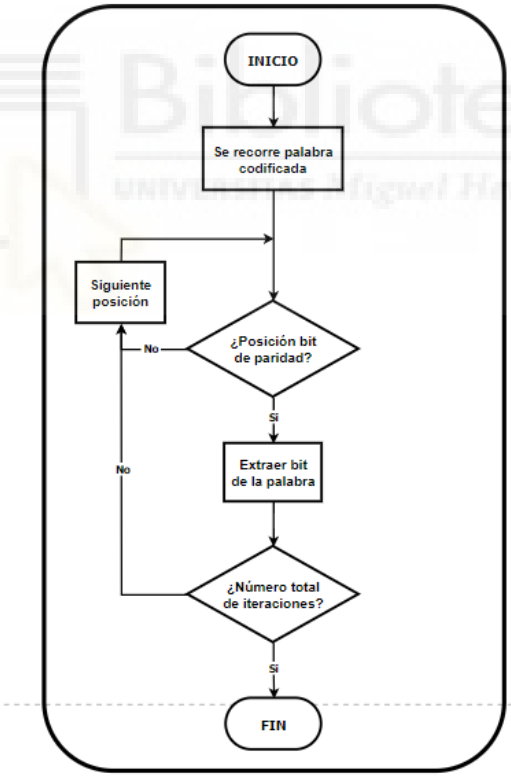

Ilustración Código 22. Diagrama de flujo función hamming_dec

Esta función decodifica la palabra copiando el contenido correspondiente a la palabra original, excluyendo los bits de paridad. Para conseguirlo, se recorre la palabra codificada, almacenando los bits que no se encuentren en una posición de bit de paridad en el vector "palabra_decodificada".

- **ext_hamming_dec:**

Archivo: hamming_pkg.vhd

Descripción: Esta función decodifica la palabra codificada mediante el código Hamming extendido.

Código:

```
Funcion ext_decoded_word[] ← ext_hamming_dec(palabra_codificada)

    // Declaración de vectores
    Dimension palabra_decodificada[tamaño_palabra_decodificada]

    // Desarrollo del algoritmo
    Para i ← 1 hasta tamaño_palabra_codificada
        Si i ≠ posicion_bit_paridad Entonces
            palabra_decodificada[tamaño_palabra_decodificada - contador] ← palabra_codificada[i]
            contador ← contador + 1
        Sino
            posicion_bit_paridad ← posicion_bit_paridad + 1
        FinSi
    FinPara
    ext_decoded_word ← palabra_decodificada
FinFuncion
```

Ilustración Código 23. Pseudocódigo función ext_hamming_dec

Esta función realiza la misma operación que la anterior para el código Hamming extendido.

- **err_correction:**

Archivo: hamming_pkg.vhd

Descripción: Esta función detecta el bit afectado por un SEU y lo corrige.

Código:

```
Funcion correction[] ← err_correction(palabra_codificada)

    // Declaración de vectores
    Dimension codificacion[tamaño_palabra_codificada]
    Dimension decodificacion[tamaño_palabra_decodificada]
    Dimension paridad_palabra_codificada[tamaño_numero_bits_paridad]
    Dimension paridad_palabra_decodificada[tamaño_numero_bits_paridad]

    // Inicialización de vectores
    codificacion                ← palabra_codificada
    decodificacion              ← hamming_dec(palabra_codificada)
    paridad_palabra_codificada  ← dec_parity_result(decodificacion)
    paridad_palabra_decodificada ← enc_parity_result(codificacion)

    // Desarrollo del algoritmo
    bit_erroneo ← operacion_XOR(paridad_palabra_codificada,paridad_palabra_decodificada)
    codificacion[bit_erroneo - 1] ← operacion_NOT(codificacion[bit_erroneo - 1])
    correction ← codificacion
FinFuncion
```

Ilustración Código 24. Pseudocódigo función err_correction

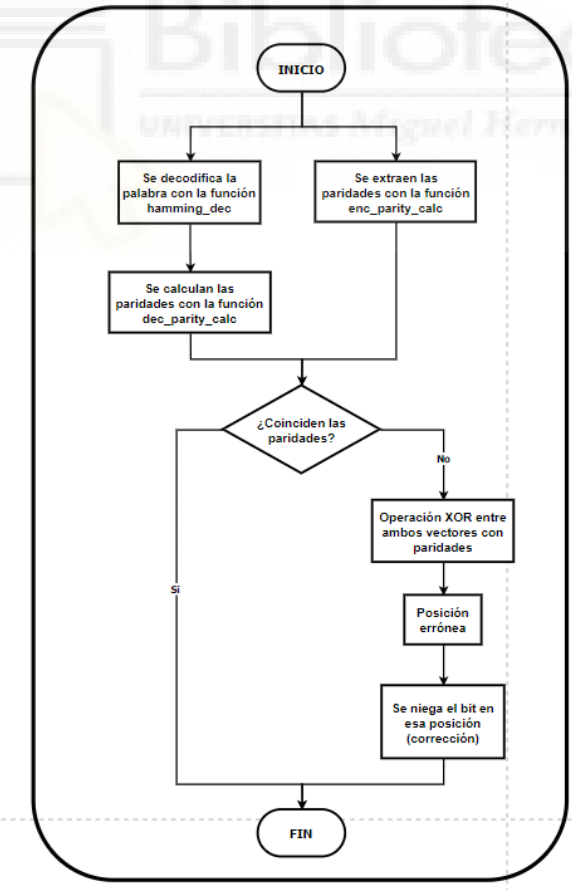

Ilustración Código 25. Diagrama de flujo función err_correction

Esta función se encarga de invertir el bit erróneo de la palabra codificada con el objetivo de su corrección. Para lograrlo, primero se ha de detectar en qué posición de la palabra ha ocurrido. Por ello, se calculan las paridades de la palabra codificada y decodificada (haciendo uso previamente de la función "hamming_dec") en los vectores "paridad_palabra_codificada" y "paridad_palabra_decodificada" respectivamente.

Una vez obtenidos los valores de paridad, se prosigue a la detección haciendo uso de la operación XOR entre "paridad_palabra_codificada" y "paridad_palabra_decodificada". De esta forma, si algún bit de paridad no coincide se devuelve un '1' notificando que el bit de paridad contiene el error. De forma contraria, si no existe cambio, se devuelve un '0' notificando que el bit erróneo no ha formado parte del cálculo de dicha paridad.

Gracias a la peculiaridad del código Hamming, el resultado de la posición del bit erróneo coincide exactamente con el resultado de la operación XOR, por lo que para corregirse se niega dicha posición de la palabra codificada mediante la operación NOT.

Finalmente se devuelve el resultado y la corrección se habrá realizado con éxito.

- **ext_err_correction:**

Archivo: hamming_pkg.vhd

Descripción: Esta función detecta el bit afectado por un SEU y lo corrige (Código Hamming Extendido).

Código:

```
Funcion ext_correction[] ← ext_err_correction(palabra_codificada)

    // Declaración de vectores
    Dimension codificacion[tamaño_palabra_codificada]
    Dimension decodificacion[tamaño_palabra_decodificada]
    Dimension paridad_palabra_codificada[tamaño_numero_bits_paridad]
    Dimension paridad_palabra_decodificada[tamaño_numero_bits_paridad]

    // Inicialización de vectores
    codificacion                ← palabra_codificada
    paridad_palabra_codificada  ← dec_parity_result(decodificacion)
    decodificacion              ← ext_hamming_dec(palabra_codificada)
    paridad_palabra_decodificada ← ext_enc_parity_result(codificacion)

    // Desarrollo del algoritmo
    bit_erroneo ← operacion_XOR(paridad_palabra_codificada,paridad_palabra_decodificada)
    codificacion[bit_erroneo] ← operacion_NOT(codificacion[bit_erroneo])
    ext_correction ← codificacion
FinFuncion
```

Ilustración Código 26. Pseudocódigo función ext_err_correction

Esta función realiza la misma operación que la anterior, pero utilizando las funciones propias del código Hamming extendido, con el fin de corregir un error codificado con el algoritmo mencionado.

10. MEMORIA SRAM CON SECDED EN VHDL

Archivo: ram_secded.vhd

Entidad:

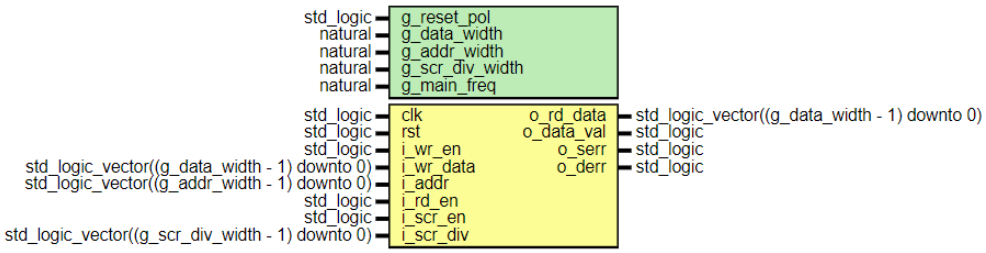

Ilustración 40. Entidad SRAM SECDED

Diagrama:

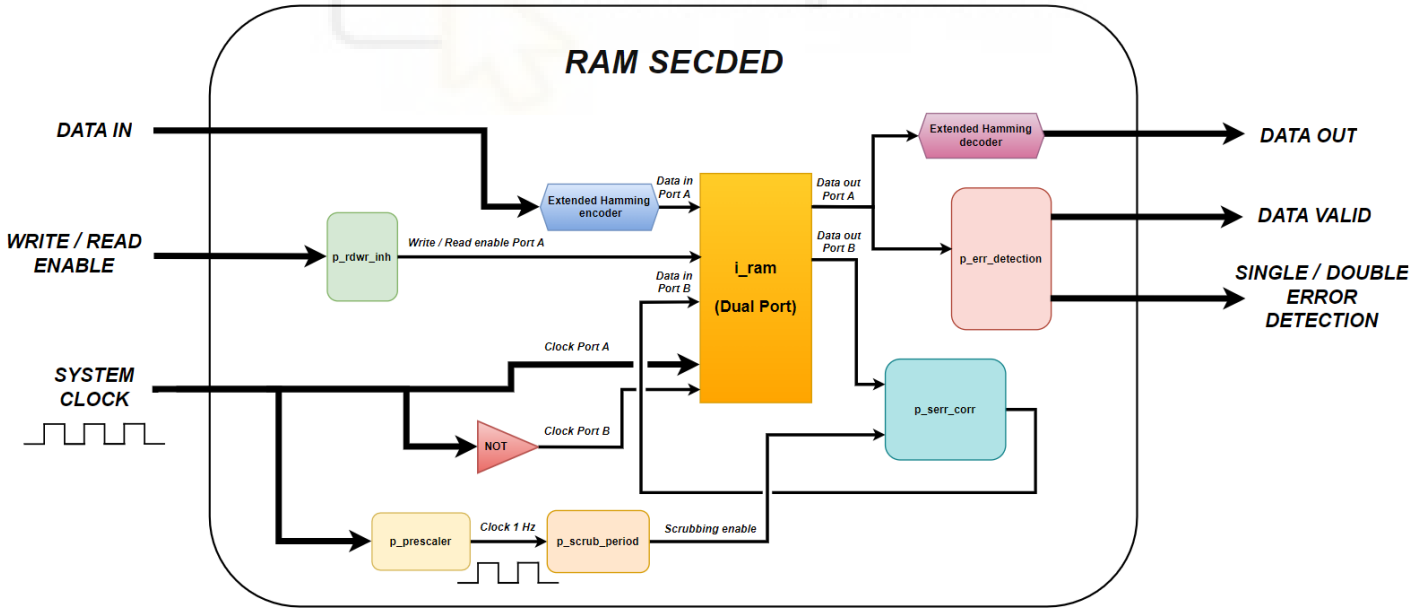

Ilustración 41. Diagrama RAM SECDED

Este módulo implementa una memoria SRAM con ECC (Error Correction Code) basado en el código Hamming extendido a través de un proceso de "scrubbing" diseñado mediante una máquina de estados.

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
g_reset_pol	std_logic	configurable	polaridad del reset
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones
g_scr_div_width	natural	configurable	ancho del divisor para "scrubbing"
g_main_freq	natural	configurable	frecuencia principal de reloj

Tabla 1. Genéricos RAM SECDED

Puertos:

Nombre del puerto	Dirección	Tipo	Descripción
clk	in	std_logic	reloj principal
rst	in	std_logic	reset principal
i_wr_en	in	std_logic	habilitador de escritura
i_wr_data[(g_data_width – 1): 0]	in	std_logic_vector	entrada de escritura
i_addr[(g_addr_width – 1):0]	in	std_logic_vector	dirección de memoria
i_rd_en	in	std_logic	habilitador de lectura
i_scr_en	in	std_logic	habilitador de "scrubbing"
i_scr_div[(g_scr_div_width – 1):0]	in	std_logic_vector	divisor de scrubbing
o_rd_data[(g_data_width – 1):0]	out	std_logic_vector	salida de lectura
o_data_val	out	std_logic	dato válido
o_serr	out	std_logic	detector de Single Error
o_derr	out	std_logic	detector de Double Error

Tabla 2. Puertos RAM SECDED

Instancias:

- **i_ram:**

Archivo: ram.vhd

Entidad:

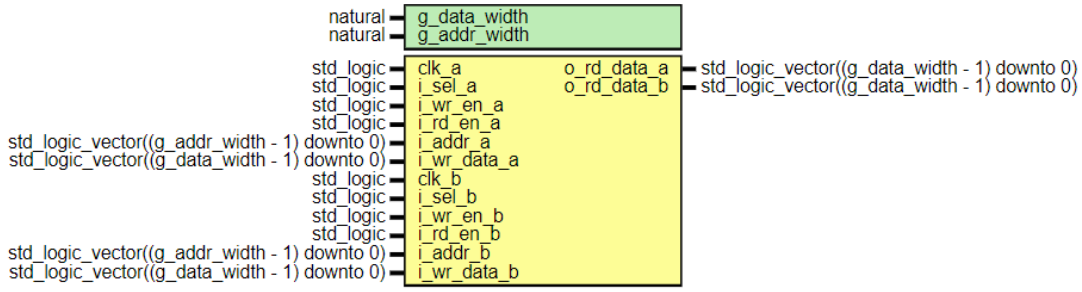

Ilustración 42. Entidad Dual Port RAM

Diagrama:

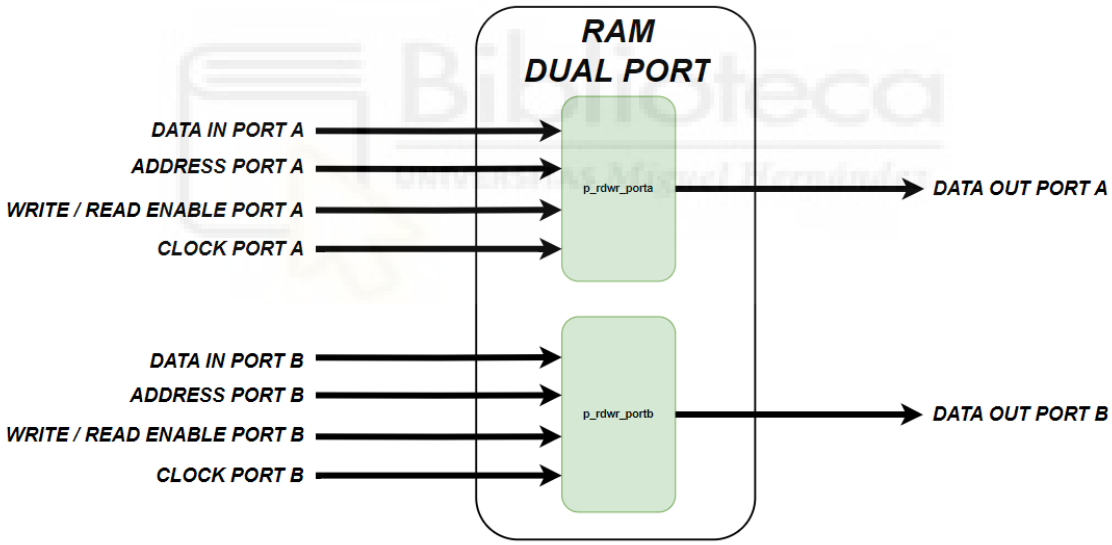

Ilustración 43. Diagrama Dual Port RAM

Descripción:

Este módulo implementa una Dual-Port Block RAM de Xilinx con dos puertos de escritura. El puerto A se utiliza para la lectura/escritura de los datos en la memoria, mientras que el puerto B está ligado al proceso de "scrubbing".

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones

Tabla 3. Genéricos Dual Port RAM

Puertos:

Nombre del puerto	Dirección	Tipo	Descripción
clk_a	in	std_logic	reloj puerto A
i_sel_a	in	std_logic	chip selector puerto A
i_wr_en_a	in	std_logic	habilitador de escritura puerto A
i_rd_en_a	in	std_logic	habilitador de lectura puerto A
i_addr_a[(g_addr_width – 1):0]	in	std_logic_vector	dirección de memoria puerto A
i_wr_data_a[(g_data_width – 1):0]	in	std_logic	entrada de escritura puerto A
clk_b	in	std_logic	reloj puerto B
i_sel_b	in	std_logic	chip selector puerto B
i_wr_en_b	in	std_logic	habilitador de escritura puerto B
i_rd_en_b	in	std_logic	habilitador de lectura puerto B
i_addr_b[(g_addr_width – 1):0]	in	std_logic_vector	dirección de memoria puerto B
i_wr_data_b[(g_data_width – 1):0]	in	std_logic_vector	entrada de escritura puerto B
o_rd_data_a[(g_data_width – 1):0]	out	std_logic_vector	salida de lectura puerto A
o_rd_data_b[(g_data_width – 1):0]	out	std_logic_vector	salida de lectura puerto B

Tabla 4. Puertos Dual Port RAM

Procesos:

- **p_rdwr_porta:**

Proceso para leer/escribir palabras de bits en el puerto A de la memoria RAM.

```
Proceso p_rdwr_port_a:
    Si flanco_ascendente_reloj_a:
        Si (selector_puerto_a y habilitador_escritura_puerto_a) es activado y
           (habilitador_lectura_puerto_a) es desactivado Entonces

            // Operación de escritura
            Memoria[direccion_puerto_a] ← entrada_escritura_puerto_a

        Sino, si (selector_puerto_a y habilitador_lectura_puerto_a) es activado y
            (habilitador_escritura_puerto_a) es desactivado Entonces

            // Operación de lectura
            salida_lectura_puerto_a ← Memoria[direccion_puerto_a]
        FinSi
    FinSi
Fin Proceso p_rdwr_port_a
```

Ilustración Código 27. Pseudocódigo proceso p_rdwr_port_a

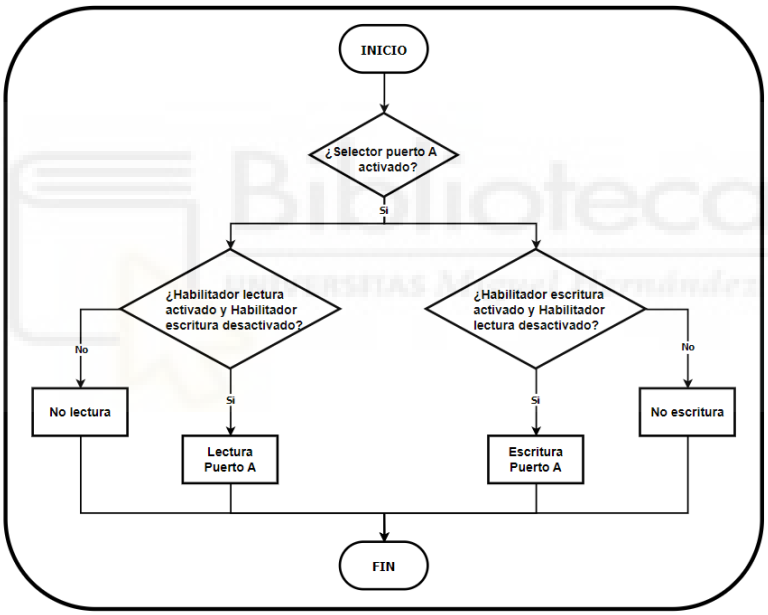

Ilustración Código 28. Diagrama de flujo proceso p_rdwr_porta

- **p_rdwr_portb:**

Proceso para leer/escribir palabras de bits en el puerto B de la memoria RAM.

```
Proceso p_rdwr_port_b:
    Si flanco_ascendente_reloj_b:
        Si (selector_puerto_b y habilitador_escritura_puerto_b) es activado y
           (habilitador_lectura_puerto_b) es desactivado Entonces

            // Operación de escritura
            Memoria[direccion_puerto_b] ← entrada_escritura_puerto_b

        Sino, si (selector_puerto_b y habilitador_lectura_puerto_b) es activado y
            (habilitador_escritura_puerto_b) es desactivado Entonces

            // Operación de lectura
            salida_lectura_puerto_b ← Memoria[direccion_puerto_b]
        FinSi
    FinSi
Fin Proceso p_rdwr_port_b
```

Ilustración Código 29. Pseudocódigo proceso p_rdwr_portb

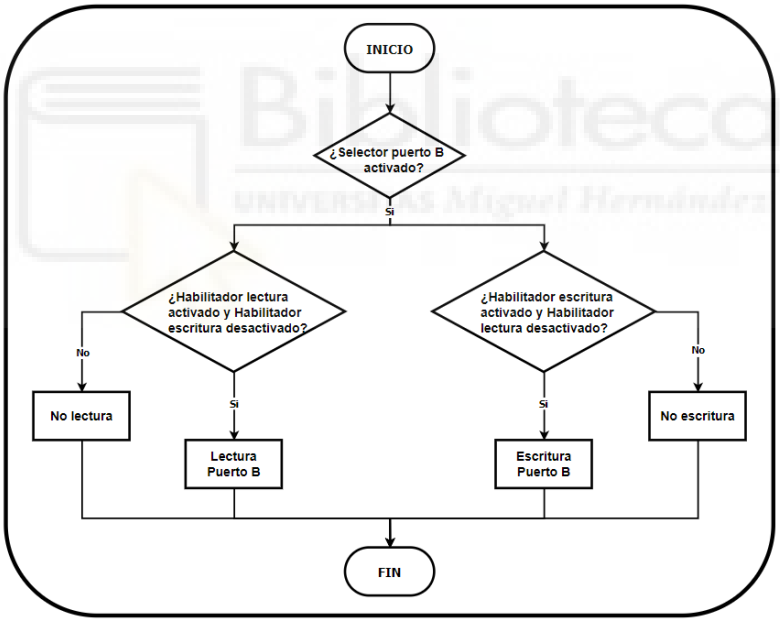

Ilustración Código 30. Diagrama de flujo proceso p_rdwr_portb

Procesos:

- **p_rdwr_inh:**

Proceso para inhibir la escritura/lectura de la instancia i_ram en caso de colisión.

Cuando tanto la lectura como escritura están activadas simultáneamente, se detienen ambas ejecuciones. Esto se lleva a cabo para evitar conflictos al intentar realizar las

operaciones en un mismo intervalo de tiempo, lo que podría resultar en una escritura/lectura de información incorrecta.

Finalmente, se configuran los habilitadores de escritura/lectura del puerto A de la instancia con respecto al valor de la señal inhibidora.

Nota: El diseño de este proceso se apoya en la versión VHDL 2008.

```
Proceso p_rdwr_inh:
    Si (habilitador_escritura y habilitador_lectura) es activado Entonces
        inhibidor ← activado
    SiNo
        inhibidor ← desactivado
    FinSi
Fin Proceso p_rdwr_inh

habilitador_lectura_a es habilitador_lectura si inhibidor es desactivado, sino desactivado
habilitador_escritura_a es habilitador_escritura si inhibidor es desactivado, sino desactivado
```

Ilustración Código 31. Pseudocódigo proceso p_rdwr_inh

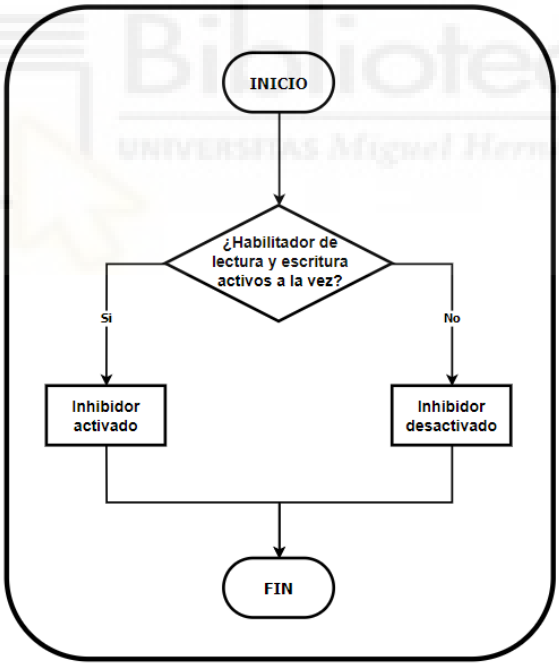

Ilustración Código 32. Diagrama de flujo proceso p_rdwr_inh

- **p_err_detection:**

Proceso para implementar un detector de “Single/Double Error” en la lectura del dato y comprobar su validación.

Se trata de un proceso de validación de datos al leer una posición de memoria. Al detectarse un “Double Error” o “Single Error” todavía no corregido por el proceso de “scrubbing”, se indica como salida la ocurrencia de estos eventos. Además, se notifica que el dato no es válido, asegurando así que se identifiquen y señalen las palabras de bits con errores no corregidos.

Nota: El diseño de este proceso es combinacional, apoyado en la versión VHDL 2008.

```
Proceso p_err_detection:
    Si (reset es activado)
        // Reset de los valores
        paridad_codificada          ← (todos los bits son '0')
        paridad_decodificada        ← (todos los bits son '0')
        paridad_global_codificada   ← '0'
        paridad_global_decodificada ← '0'
    SiNo
        paridad_codificada          ← ext_enc_parity_result(lectura_puerto_a)
        paridad_decodificada        ← dec_parity_result(ext_hamming_dec(lectura_puerto_a))
        paridad_global_codificada   ← lectura_puerto_a[0]
        paridad_global_decodificada ← global_parity(lectura_puerto_a[tamaño_lectura_puerto_a:1])
    FinSi

    detector_single_error es activado si (paridad_global_codificada es distinto de paridad_global_decodificada)
        y (paridad_codificada es distinto de paridad_decodificada) sino desactivado

    detector_double_error es activado si (paridad_global_codificada es igual que paridad_global_decodificada)
        y (paridad_codificada es distinto de paridad_decodificada) sino desactivado

    dato_valido es activado si (detector_single_error es igual que detector_double_error) sino desactivado
Fin Proceso p_err_detection
```

Ilustración Código 33. Pseudocódigo proceso p_err_detection

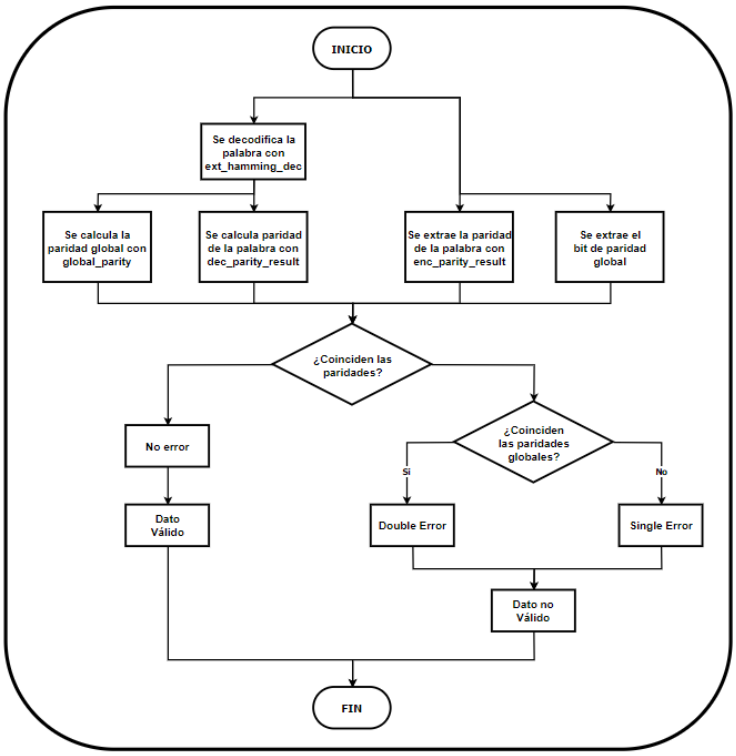

Ilustración Código 34. Diagrama de flujo proceso p_err_detection

- **p_prescaler:**

Proceso para implementar un "prescaler", con el fin de escalar el periodo de "scrubbing" para la operación ECC (Error Correction Code).

Se trata de un paso intermedio, donde se generan pulsos cada periodo de un segundo con una duración de un ciclo de reloj. Este segundo de periodo corresponderá con el periodo mínimo configurable para realizar la operación de "scrubbing".

```
Proceso p_prescaler:
    Si reset es activado Entonces
        prescaler ← desactivado
        contador  ←  0
    SiNo
        Si contador es igual que frecuencia_principal Entonces
            prescaler ← activado
            contador  ←  0
        SiNo
            prescaler ← desactivado
            contador  ← contador + 1
        FinSi
    FinSi
Fin Proceso p_prescaler
```

Ilustración Código 35. Pseudocódigo proceso p_prescaler

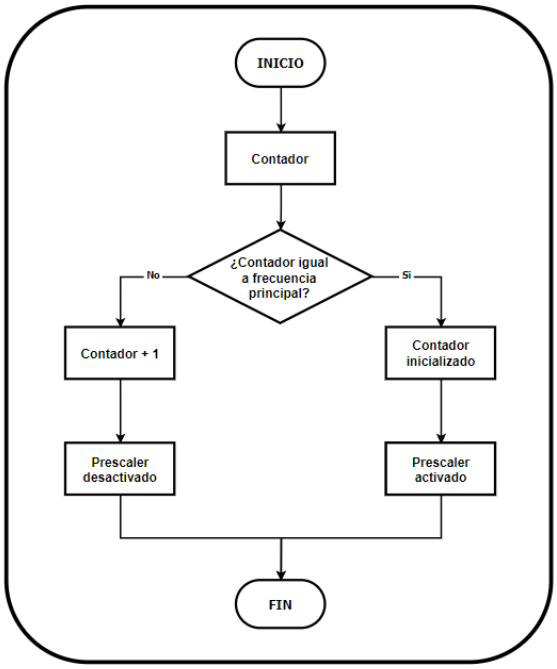

Ilustración Código 36. Diagrama de flujo proceso p_prescaler

- **p_scrub_period:**

Este proceso implementa un periodo de "scrubbing" configurable para ECC (Error Correction Code).

El tamaño de la entrada divisor_scrubbing es configurable por el usuario. Si el divisor no tiene valor, la señal de "scrubbing" no se generará.

```
Proceso p_scrub_period:
    Si reset es activado Entonces
        scrubbing_tick ← desactivado
        contador       ← 0
    SiNo
        Si (divisor_scrubbing es mayor que UNO) y (prescaler es activado) Entonces
            Si contador es menor que divisor_scrubbing Entonces
                contador ← contador + 1
            SiNo
                scrubbing_tick ← activado
                contador       ← 0
            FinSi
        SiNo, si divisor_scrubbing es igual que UNO Entonces
            scrubbing_tick ← prescaler
        SiNo
            scrubbing_tick ← desactivado
        FinSi
    FinSi
Fin Proceso p_scrub_period
```

Ilustración Código 37. Pseudocódigo proceso p_scrub_period

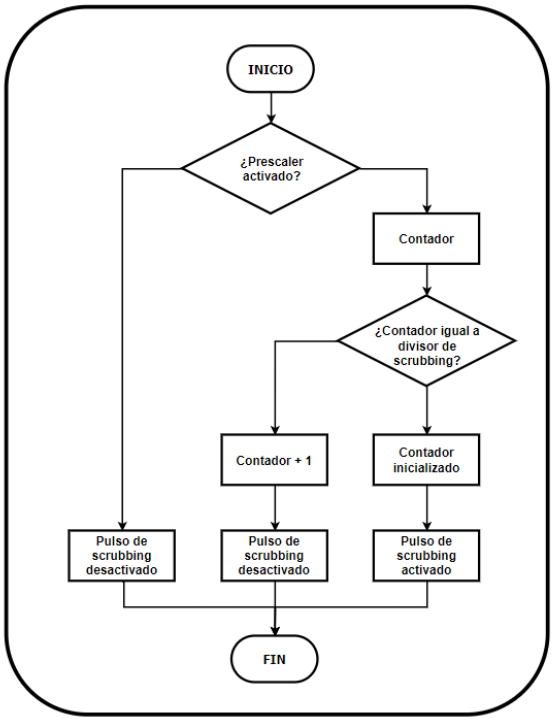

Ilustración Código 38. Diagrama de flujo proceso p_scrub_period

- **p_serr_corr:**

Este proceso implementa una técnica de lectura y escritura periódica de datos almacenados en memoria, con el objetivo de detectar y corregir errores de forma automatizada. Esta técnica se denomina “scrubbing”.

El “scrubbing” ayuda a identificar y corregir errores de bit que puedan ocurrir de manera aleatoria en la memoria a lo largo del tiempo debido a diversos factores, entre los más comunes, la radiación cósmica.

Con el fin de evitar colisión entre los accesos de escritura/lectura del usuario a través del puerto A y del “scrubbing” a través del puerto B de la memoria RAM, este proceso está sincronizado a una señal de reloj de frecuencia igual al reloj principal del módulo, con un desfase de noventa grados.

```
Proceso p_serr_corr:
    Si reset es activado Entonces
        inicializar_señales_y_maquina_estados()
    SiNo
        resetear_acceso_RAM_puerto_B()

        caso maquina_estados es
            Para IDLE ->
                Si habilitador_scrub es activado Entonces
                    Si scrubbing_tick es activado Entonces
                        acceso_lectura_RAM_puerto_B()
                        maquina_estados ← PARIDAD
                    FinSi
                FinSi

            Para PARIDAD ->
                paridad_codificada          ← ext_enc_parity_result(lectura_puerto_b)
                paridad_decodificada        ← dec_parity_result(ext_hamming_dec(lectura_puerto_b))
                paridad_global_codificada   ← lectura_puerto_b[0]
                paridad_global_decodificada ← global_parity(lectura_puerto_b[tamaño_lectura_puerto_b:1])
                maquina_estados             ← ESPERA

            Para ERROR ->
                acceso_escritura_RAM_puerto_B(ext_err_correction(lectura_puerto_b))
                maquina_estados ← ESPERA

            Para ESPERA ->

                // Single Error
                Si (paridad_global_codificada es distinto de paridad_global_decodificada) y
                   (paridad_codificada es distinto de paridad_decodificada) Entonces
                    maquina_estados ← ERROR
                Sino
                    maquina_estados ← SIGUIENTE_DIRECCION_MEMORIA
                FinSi

            Para SIGUIENTE_DIRECCION_MEMORIA ->
                Si direccion_memoria es MAXIMO_ESPACIO_RAM Entonces
                    direccion_memoria ← direccion_inicial
                SiNo
                    direccion_memoria ← direccion_memoria + 1
                FinSi
                maquina_estados ← IDLE

            Para OTROS_CASOS ->
                maquina_estados ← IDLE
        Fin Caso
Fin Proceso p_serr_corr
```

Ilustración Código 39. Pseudocódigo proceso p_serr_corr

Máquina de Estados Finitos (Finite State Machine):

IDLE: Este estado representa el punto inicial de la máquina de estados. Si la señal de habilitación de “scrubbing” está activa, se espera la llegada del pulso “scrubbing_tick” generado por el proceso anterior. Una vez recibido, se lleva a cabo la lectura de la memoria y la máquina avanza al siguiente estado “PARIDAD”.

PARIDAD: En este estado, se llevan a cabo dos operaciones. Por un lado, se extraen los bits de paridad, los cuales se almacenan en una señal, haciendo una distinción entre paridad local y paridad global. Por otro lado, se decodifica la palabra y se recalculan sus bits de paridad, manteniendo la distinción entre paridad local y paridad global. Finalmente, la máquina avanza al siguiente estado "ESPERA".

ESPERA: En este estado se comparan las paridades de la palabra codificada y decodificada calculadas en el estado anterior y se comparan sus valores. Si ambas paridades son diferentes se establece que ha ocurrido un "single error event" y se procede a transicionar al estado "ERROR". En cualquier otro caso (sin errores o "double error event"), se procede al estado "SIGUIENTE DIRECCIÓN DE MEMORIA". No se realizan distinciones entre "double error event" y la no existencia de errores, debido a que el código Hamming no es capaz de corregir un "double error event".

ERROR: En este estado se corrige el bit erróneo perteneciente a la palabra que ha sufrido un "single error upset" y la máquina regresa al estado "ESPERA" para una validación adicional.

SIGUIENTE DIRECCIÓN DE MEMORIA: Este estado se encarga de ir accediendo a las siguientes posiciones de memoria, mediante el empleo de un contador. Si ya ha alcanzado la posición máxima, reinicia el contador y vuelve a empezar la cuenta. Tras finalizar esta operación, la máquina regresa al estado "IDLE".

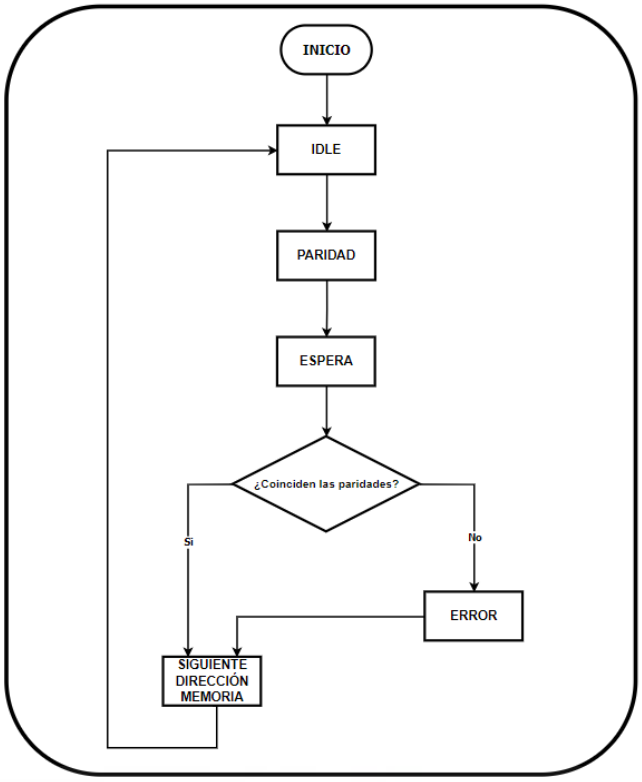

Ilustración Código 40. Diagrama de flujo proceso p_serr_corr

11. MEMORIA SRAM CON TMR EN VHDL

Archivo: ram_tmr.vhd

Entidad:

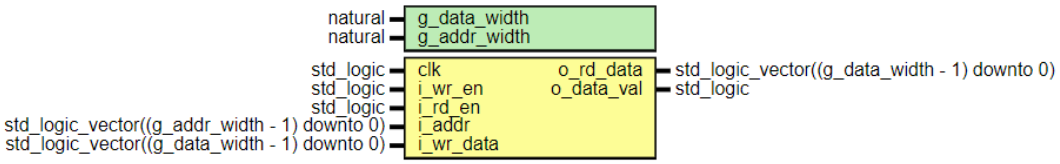

Diagrama:

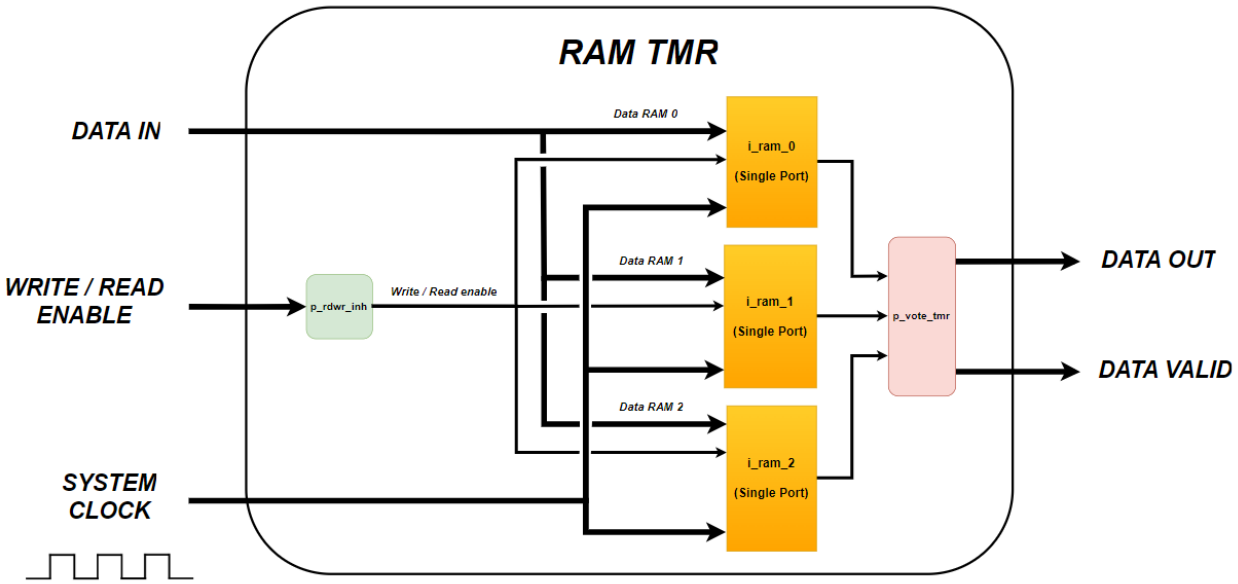

Ilustración 44. Diagrama RAM TMR

Descripción:

Este módulo implementa una memoria SRAM con TMR (Triple Modular Redundancy), basado en sistema de votación por mayoría.

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones

Tabla 5. Genéricos RAM TMR

Puertos:

Nombre del puerto	Dirección	Tipo	Descripción
clk	in	std_logic	reloj principal
i_wr_en	in	std_logic	habilitador de escritura
i_rd_en	in	std_logic	habilitador de lectura
i_wr_data[(g_data_width – 1): 0]	in	std_logic_vector	entrada de escritura
i_addr[(g_addr_width – 1):0]	in	std_logic_vector	dirección de memoria
o_rd_data[(g_data_width – 1):0]	out	std_logic_vector	salida de lectura
o_data_val	out	std_logic	dato válido

Tabla 6. Puertos RAM TMR

Instancias:

- **i_ram:**

Archivo: ram_single_port.vhd

Entidad:

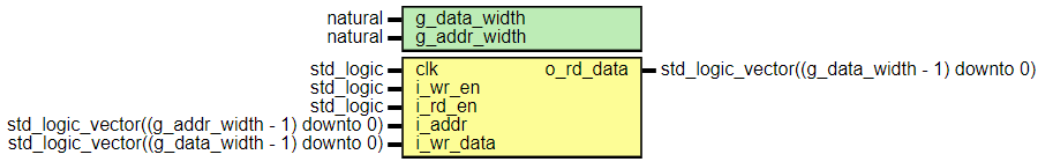

Ilustración 45. Entidad Single Port RAM

Diagrama:

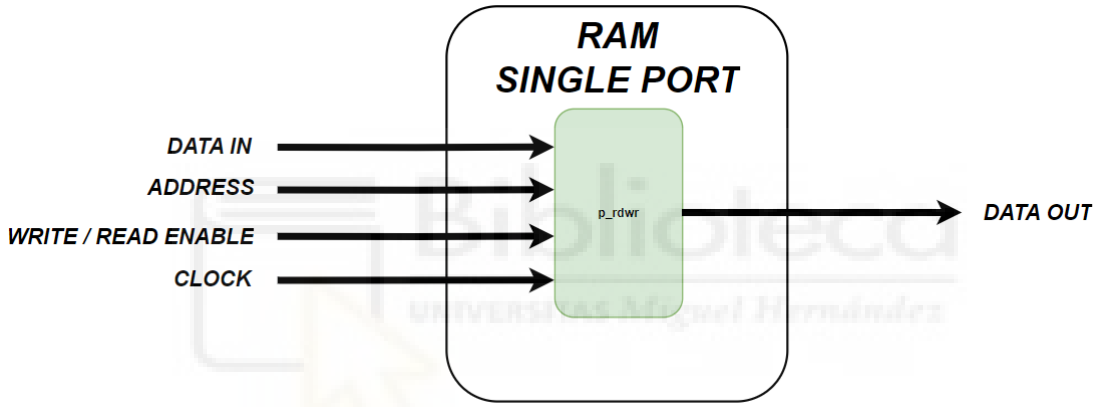

Ilustración 46. Diagrama Single Port RAM

Descripción:

Este módulo implementa una Single-Port Block RAM de Xilinx.

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones

Tabla 7. Genéricos Single Port RAM

Puertos:

Nombre del puerto	Dirección	Tipo	Descripción
clk	in	std_logic	reloj principal
i_wr_en	in	std_logic	habilitador de escritura
i_rd_en	in	std_logic	habilitador de lectura
i_wr_data[(g_data_width – 1): 0]	in	std_logic_vector	entrada de escritura
i_addr[(g_addr_width – 1):0]	in	std_logic_vector	dirección de memoria
o_rd_data[(g_data_width – 1):0]	out	std_logic_vector	salida de lectura

Tabla 8. Puertos Single Port RAM

Procesos:

Proceso para leer/escribir palabras de bits de/en la memoria RAM.

```
Proceso p_rdwr:
    Si flanco_ascendente_reloj:
        Si habilitador_escritura es activado y habilitador_lectura es desactivado Entonces

            // Operación de escritura
            Memoria[direccion_memoria] ← entrada_escritura

        Sino, si habilitador_escritura es desactivado y habilitador_lectura es activado Entonces

            // Operación de lectura
            salida_lectura ← Memoria[direccion_memoria]
        FinSi
    FinSi
Fin Proceso p_rdwr
```

Ilustración Código 41. Pseudocódigo proceso p_rdwr

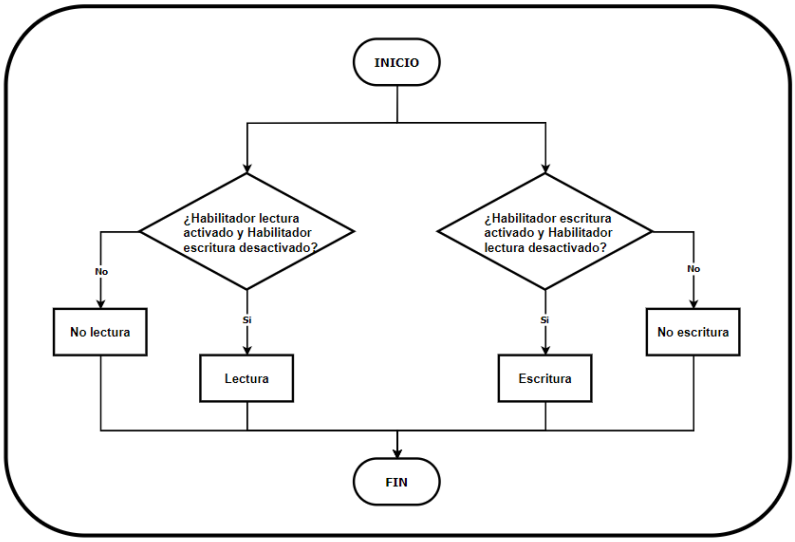

Ilustración Código 42. Diagrama de flujo proceso p_rdwr

Procesos:

Proceso para inhibir la escritura/lectura de la instancia i_ram en caso de colisión.

Su comportamiento es idéntico al proceso equivalente en la memoria SRAM con SECDED.

Nota: El diseño de este proceso se apoya en la versión VHDL 2008.

```
Proceso p_rdwr_inh:
    Si (habilitador_escritura y habilitador_lectura) es activado Entonces
        inhibidor ← activado
    SiNo
        inhibidor ← desactivado
    FinSi
Fin Proceso p_rdwr_inh
```

Ilustración Código 43. Pseudocódigo proceso p_rdwr_inh

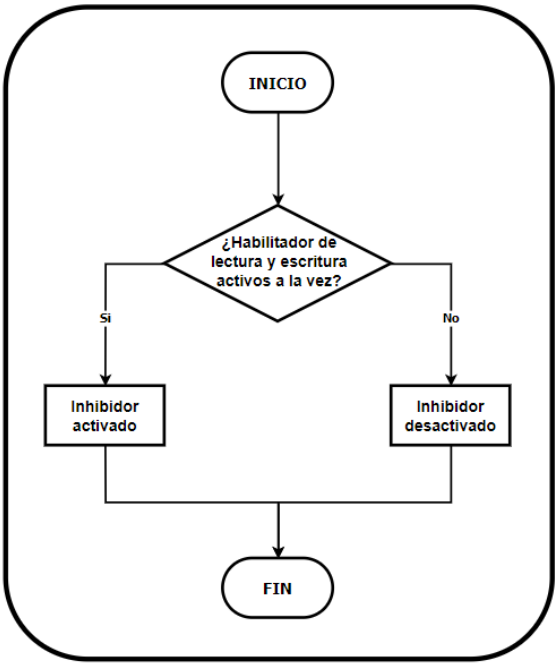

Ilustración Código 44. Diagrama de flujo proceso p_rdwr_inh

- **p_vote_tmr:**

Este proceso ejecuta un sistema de votación por mayoría, el cual recibe las lecturas de cada memoria RAM implementada, a través de un registro, compara los valores y devuelve el más veces repetido. En caso de no haber valor repetido, se notifica al usuario que el dato no es válido a través de una salida habilitada para ello.

Nota: El diseño de este proceso se apoya en la versión VHDL 2008.

```
Proceso p_vote_tmr:
    Si (registro_lectura_RAM[0] es distinto de registro_lectura_RAM[1]) y
       (registro_lectura_RAM[0] es distinto de registro_lectura_RAM[2]) y
       (registro_lectura_RAM[1] es igual que registro_lectura_RAM[2]) Entonces
        salida_lectura ← registro_lectura[1]
        dato_valido    ← activado

    SiNo, si (registro_lectura_RAM[0] es distinto de registro_lectura_RAM[1]) y
       (registro_lectura_RAM[0] es distinto de registro_lectura_RAM[2]) y
       (registro_lectura_RAM[1] es distinto de registro_lectura_RAM[2]) Entonces
        salida_lectura ← registro_lectura[0]
        dato_valido    ← desactivado
    SiNo
        salida_lectura ← registro_lectura[0]
        dato_valido    ← activado
    FinSi
Fin Proceso p_vote_tmr
```

Ilustración Código 45. Pseudocódigo proceso p_vote_tmr

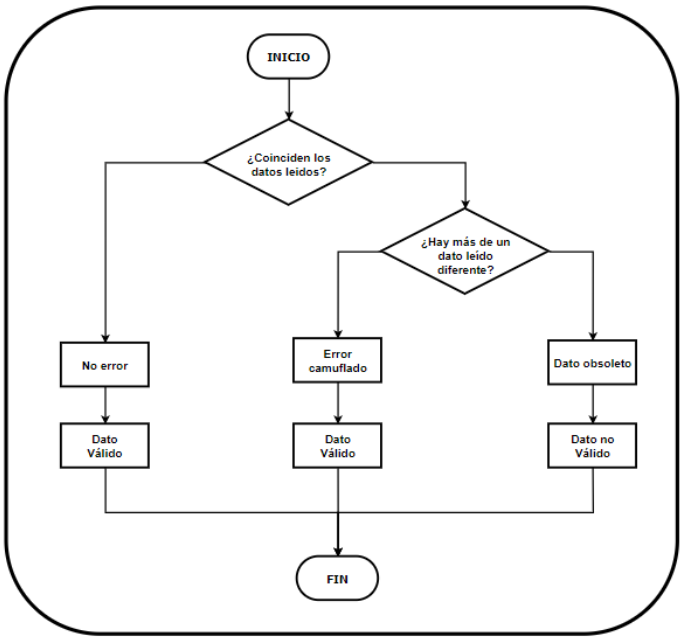

Ilustración Código 46. Diagrama de flujo proceso p_vote_tmr

Bloque de generación:

- **g_ram_tmr:**

Este bloque implementa un bucle que incluye las tres memorias RAM Single-Port de Xilinx para la triple redundancia.

```
g_ram_tmr: Para j ← 0 hasta 2
    instancia_memoria_RAM[j]
FinPara
```

Ilustración Código 47. Pseudocódigo proceso g_ram_tmr

12. VERIFICACIÓN / VALIDACIÓN DE AMBOS ALGORITMOS

A continuación, se describen los archivos de verificación utilizados para validar el diseño, por medio de la librería UVVM (Universal VHDL Verification Metodology), y se muestran los resultados en forma de onda. Cada archivo se compone de diversos pasos, cada cual con el fin de comprobar una característica concreta del diseño.

Se ha tenido en cuenta que cada memoria posee parámetros configurables, por lo que esto también se verá reflejado en la verificación. Se ha decidido que dichos parámetros sean enviados por el usuario a través de argumentos. De no ser enviado alguno de ellos, se les asignará un valor por defecto automáticamente.

El diseño se tomará como válido si, dados diferentes argumentos para cada parámetro, los correspondientes pasos de la prueba se cumplen sin error.

12.1 TESTBENCH MEMORIA SRAM CON SECDED VHDL

Archivo: ram_secded_tb.vhd

Entidad:

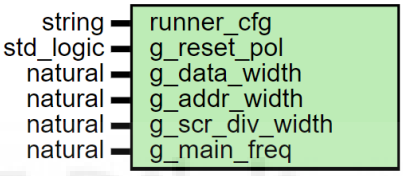

Ilustración 47. Entidad RAM SECDED Testbench

Diagrama:

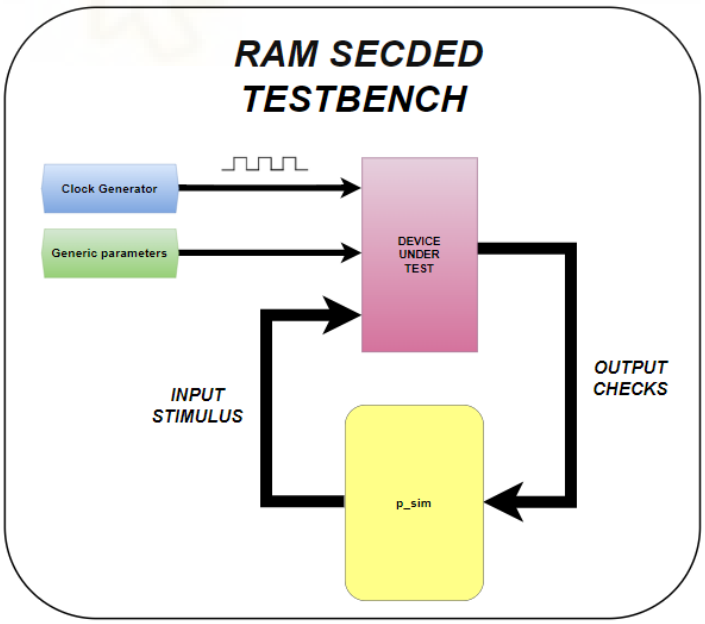

Ilustración 48. Diagrama RAM SECDED Testbench

Descripción:

Este testbench testea la memoria SRAM con ECC (Error Correction Code) basado en el código Hamming extendido.

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
runner_cfg	string	nulo	archivo de configuración
g_reset_pol	std_logic	configurable	polaridad del reset
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones
g_scr_div_width	natural	configurable	ancho del divisor para "scrubbing"
g_main_freq	natural	configurable	frecuencia principal de reloj

Tabla 9. Genéricos RAM SECDED Testbench

Instancias:

- **device_under_test:**

Esta instancia corresponde a la entidad de la memoria SRAM con SECDED a verificar.

Generación de reloj:

Para la generación de reloj se ha hecho uso de la siguiente función de la librería UVVM:

```
clock_generation(reloj, periodo)
```

Ilustración Código 48. Generación de reloj

Donde se le asigna como primer argumento la señal de reloj, y como segundo argumento el periodo de reloj deseado.

- Formas de onda:

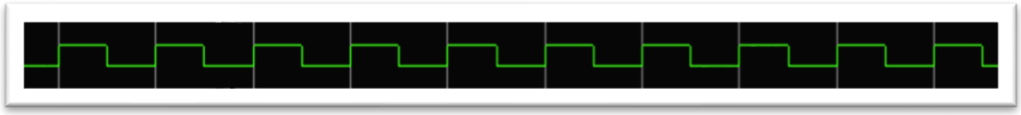

Ilustración Código 49. Forma de onda reloj RAM SECDED Testbench

Procesos:

- **p_sim:**

Proceso para simular los estímulos que recibe el "device under test". En su interior se describen los pasos a testear y se verifica su comportamiento ante dichos estímulos.

Este es el cuerpo principal del test, donde se pone a prueba las características objetivo del diseño y su respuesta ante situaciones adversas.

Procedures:

En esta sección se diseñan los "procedures" que se utilizarán más adelante para la verificación. El objetivo de los "procedures" es simplificar el proceso de testeo, hacerlo más entendible y reutilizable, encapsulando una serie de secuencias que describen una acción concreta.

inject_serr:

Este "procedure" inyecta un "Single Error" en una palabra de bits almacenada en la memoria. La dirección de memoria y el índice a inyectar el error son recibidos como argumento.

```
SubProceso inject_serr(direccion_memoria, indice):
    palabra_erronea            ← memoria[direccion_memoria]
    palabra_erronea[indice]    ← operacion_NOT(palabra_erronea[indice])
    memoria[direccion_memoria] ← palabra_erronea
FinSubProceso
```

Ilustración Código 50. Pseudocódigo procedure inject_serr

inject_derr:

Este "procedure" injecta un "Double Error" en una palabra de bits almacenada en la memoria. La dirección de memoria y ambos índices a inyectar los errores son recibidos como argumento.

```
SubProceso inject_derr(direccion_memoria, primer_indice, segundo_indice):
    palabra_erronea                ← memoria[direccion_memoria]
    palabra_erronea[primer_indice]  ← operacion_NOT(palabra_erronea[primer_indice])
    palabra_erronea[segundo_indice] ← operacion_NOT(palabra_erronea[segundo_indice])
    memoria[direccion_memoria]      ← palabra_erronea
FinSubProceso
```

Ilustración Código 51. Pseudocódigo procedure inject_derr

Secuencia de test:

PASO 0: Activar/Desactivar reset y generar Tablas

En este paso se testea el comportamiento del reset y se generan dos tablas con datos aleatorios. La primera tabla contendrá valores que se usarán como datos para la comprobación de la correcta escritura/lectura en la memoria. La segunda tabla contendrá los datos de la primera codificados mediante el código Hamming extendido, que se usarán como comprobación para validar que el sistema automático de corrección de errores funciona correctamente.

La ventaja de la generación de tablas recae en la flexibilidad que aportan al archivo de prueba al tratarse de una memoria configurable por el usuario. Su implementación permite la generación dinámica de datos basados en los argumentos proporcionados por consola. Esto evita la necesidad de reescribirlos cada vez que se desee introducir un nuevo valor para los argumentos.

Para la comprobación del reset se utiliza la función “check_value” [19] de la librería UVVM, cuyo formato es el siguiente:

check_value(valor_actual, valor_deseado, nivel_alerta[opcional], mensaje, [...])

```
// ------------------------------------------------------------------------------------------
//  PASO 0: Activar/Desactivar reset después de 100 us y generar Tablas
// ------------------------------------------------------------------------------------------

Escribir "Activar/Desactivar reset después de 100 us y generar Tablas"

Escribir "Inicializando entradas"
reset ← activado
inicializacion_entradas()

Escribir "Generando Tabla Aleatoria: "
Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    tabla_aleatoria[i] ← valor_aleatorio
    Escribir "Escribiendo valor "tabla_aleatoria[i]" en Tabla Aleatoria (posición "i")
FinPara
Esperar 0.1 milisegundos

Escribir "Generando Tabla Hamming Aleatoria: "
Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    tabla_hamming_aleatoria[i] ← valor_aleatorio
    Escribir "Escribiendo valor "tabla_hamming_aleatoria[i]" en Tabla Hamming Aleatoria (posición "i")
FinPara
Esperar 0.1 milisegundos

reset ← desactivado
Esperar 0.001 milisegundos

check_value(reset, desactivado, "Reset desactivado")
Esperar 0.1 milisegundos
```

Ilustración Código 52. Pseudocódigo estímulos “PASO 0” RAM SECDED Testbench

- Formas de onda:

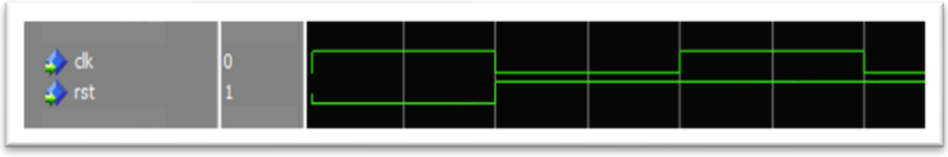

Ilustración Código 53. Formas de onda "PASO 0" RAM SECDED

PASO 1: Escribir valores en todas las direcciones de la SRAM

En este paso se testea la escritura de los datos en cada posición de memoria de la SRAM, dando los correspondientes estímulos a las entradas para realizar la operación.

```
// ------------------------------------------------------------------------------------------
//  PASO 1: Escribir valores en todas las direcciones de la SRAM
// ------------------------------------------------------------------------------------------

Escribir "Escribir valores en todas las direcciones de la SRAM"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_escritura ← activado
    direccion_memoria     ← i_en_binario
    dato_entrada          ← tabla_aleatoria[i]
    Esperar CICLO_RELOJ

    Escribir "Escribiendo valor "tabla_aleatoria[i]" en la dirección "direccion_memoria"
    Esperar CICLO_RELOJ

    habilitador_escritura ← desactivado
    Esperar 20 * CICLO_RELOJ
FinPara
```

Ilustración Código 54. Pseudocódigo estímulos "PASO 1" SRAM SECDED Testbench

- Formas de onda:

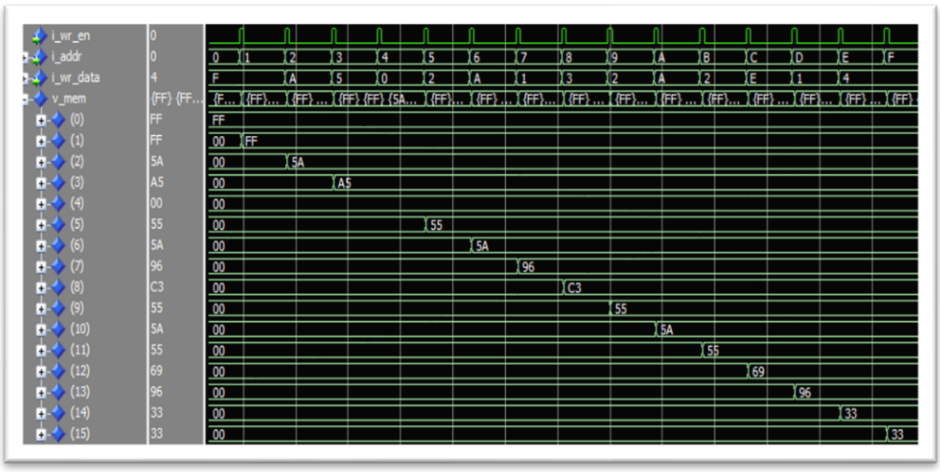

Ilustración Código 55. Formas de onda "PASO 1" RAM SECDED

PASO 2: Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los escritos en el PASO 1

Este paso se comprueba la lectura de la memoria SRAM, dando los correspondientes estímulos a las entradas para realizar la operación. Una vez leído el valor, se comprueba que coincide con el escrito en el paso anterior mediante la función “check_value”.

```
// ------------------------------------------------------------------------------------------
//  PASO 2: Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los
//  escritos en el PASO 1
// ------------------------------------------------------------------------------------------

Escribir "Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los escritos en el PASO 1"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección: "dirección_memoria") Lectura:  "dato_salida"     Esperado: "tabla_aleatoria[i])
    Esperar 20 * CICLO_RELOJ
FinPara
```

Ilustración Código 56. Pseudocódigo estímulos "PASO 2" SRAM SECDED Testbench

- Formas de onda:

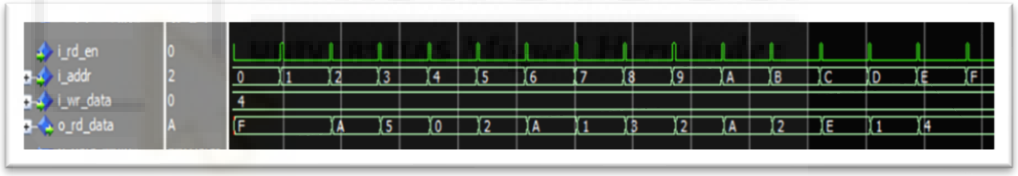

Ilustración Código 57. Formas de onda "PASO 2" RAM SECDED

PASO 3: Comprobar la detección de colisión de escritura/lectura

Este paso pone a prueba la detección de colisión entre escritura/lectura. Para ello, se genera la situación descrita habilitando la escritura y lectura en un mismo instante. Se espera que tanto la memoria como la salida de lectura no se actualicen durante ese intervalo, con el fin de evitar errores.

```
// ---------------------------------------------------------------------------------------------
// PASO 3: Comprobar la detección de colisión de escritura/lectura
// ---------------------------------------------------------------------------------------------

Escribir "Comprobar la detección de colisión de escritura/lectura"

Para i ← 0 hasta 2
    Escribir "Forzar colisión habilitando la escritura y lectura, escribiendo "(todos_los_bits '0')" en la dirección "i_en_binario"

    Escribir "Comprobar que el inhibidor de lectura/escritura está activo y no se ha realizado ninguna operación"
    habilitador_escritura ← activado
    habilitador_lectura   ← activado
    direccion_memoria     ← i_en_binario
    dato_entrada          ← (todos_los_bits '0')
    Esperar CICLO_RELOJ

    habilitador_escritura ← desactivado
    habilitador_lectura   ← desactivado
    check_value(inhibidor, activado, "Inhibidor de lectura/escritura activado")
    check_value(hablitador_escritura_puerto_a, desactivado, "Operación de escritura desactivada")
    check_value(habilitador_lectura_puerto_a, desactivado, "Operación de lectura desactivada")
    Esperar CICLO_RELOJ
FinPara
Esperar 0.01 milisegundos

Para i ← 0 hasta 2
    Escribir "Comprobar dato en la dirección "i_en_binario" no ha sido modificado"
    habilitador_lectura ← activado
    dirección_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección "direccion_memoria") Lectura: "dato_salida"       Esperado: "tabla_aleatoria[i])
    Esperar CICLO_RELOJ
FinPara
```

Ilustración Código 58. Pseudocódigo estímulos "PASO 3" SRAM SECDED Testbench

- Formas de onda:

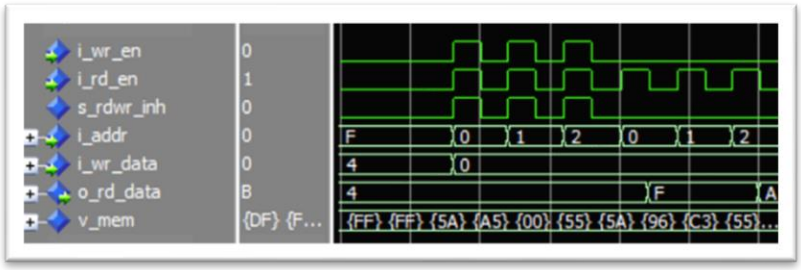

Ilustración Código 59. Formas de onda "PASO 3" RAM SECDED

PASO 4: Forzar “Single Errors” en algunas direcciones de memoria y comprobar que se corrigen automáticamente

En este paso se inyectan “Single Errors” en los datos de la memoria para evaluar la eficacia del proceso de “scrubbing” automático, tomando como caso esquina la inserción de un error en el bit 0 de cualquier dato. Para llevar a cabo esta acción, se emplea el “procedure” diseñado previamente: “inject_serr”.

```
// ------------------------------------------------------------------------------------------------
// PASO 4: Forzar "Single Errors" en algunas direcciones de memoria y comprobar que se corrigen
// automáticamente
// ------------------------------------------------------------------------------------------------

Escribir "Forzar "Single Errors" en algunas direcciones de memoria y comprobar que se corrigen automáticamente"

Escribir "Inyectando Single Errors: "
habilitador_scrubbing <- activado

Repetir para i <- 0 hasta 7
    Escribir "Single Error inyectado en la dirección "i_en_binario"
    inject_serr(i, valor_aleatorio)
    Si i > 5 Entonces
        Escribir "Single Error inyectado (bit 0) en la dirección "i_en_binario"
        inject_serr(i, 0)
    FinSi
FinPara
Esperar 0.1 milisegundos

Escribir "Leer algunas direcciones de memoria, con y sin Single Errors inyectados, y comprobar las salidas detector_single_error,
        "detector_double_error y dato_valido: "
Para i <- 0 hasta 7
    habilitador_lectura <- activado
    direccion_memoria   <- i_en_binario
    Esperar CICLO_RELOJ

    Caso i es
        Para i de 0 hasta 5 ->
            check_value(dato_valido, desactivado, "Dato no válido")
            check_value(detector_single_error, activado, "Single Error detectado")
            check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")

        Para i de 6 hasta 7 -> // Caso esquina (error inyectado en bit 0)
            check_value(dato_valido, activado, "Dato válido")
            check_value(detector_single_error, desactivado, "No se ha detectado Single Error")
            check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")

        Para OTROS_CASOS ->
            check_value(dato_valido, activado, "Dato válido")
            check_value(detector_single_error, desactivado, "No se ha detectado Single Error")
            check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")
    Fin Caso
    Esperar CICLO_RELOJ

    habilitador_lectura <- desactivado
    Esperar 20 * CICLO_RELOJ
FinPara

Escribir "Esperar 8 segundos para que el EDAC corrija los errores" // El periodo de scrubbing está inicializado a 1
Esperar 8 segundos

Escribir "Volver a leer las direcciones de memoria y comprobar que los Single Errors han sido corregidos"
operación_lectura_SRAM()
```

Ilustración Código 60. Pseudocódigo estímulos "PASO 4" SRAM SECDED Testbench

- Formas de onda:

Inyección de errores y lectura

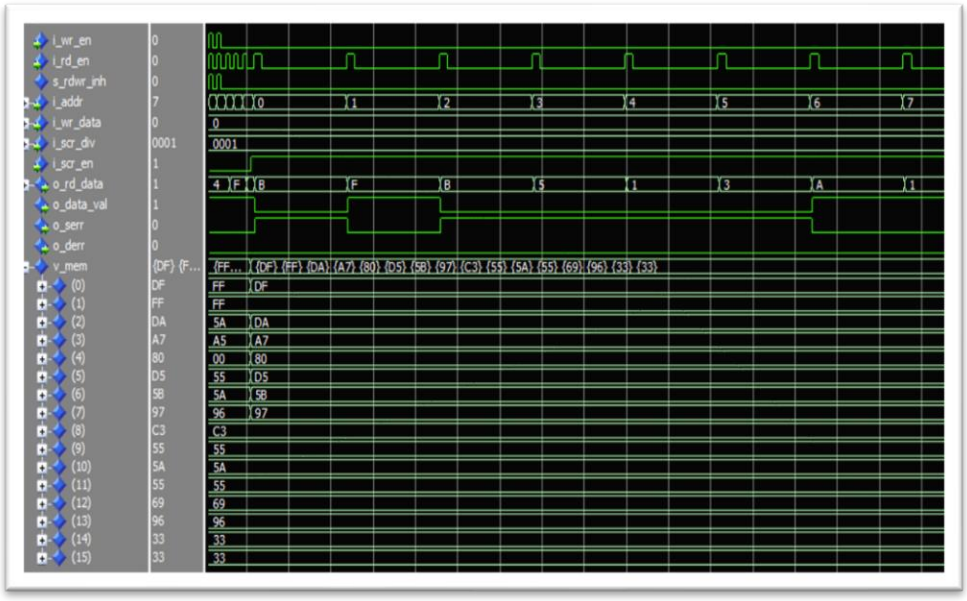

Ilustración Código 61. Formas de onda "PASO 4" RAM SECDED (1)

Máquina de estados

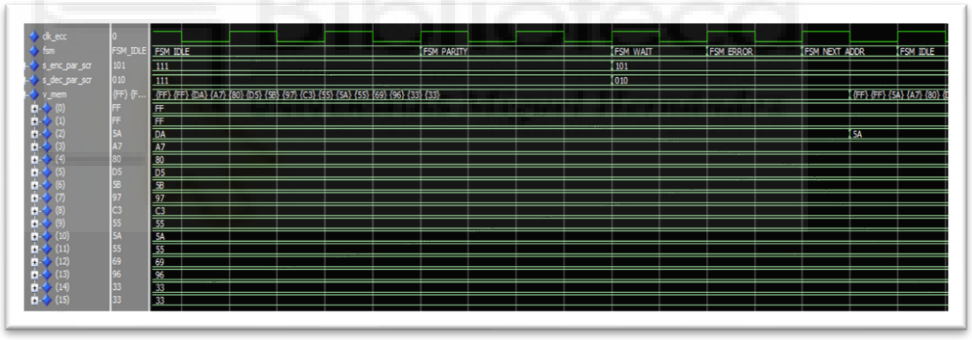

Ilustración Código 62. Formas de onda "PASO 4" RAM SECDED (2)

Corrección automática de errores

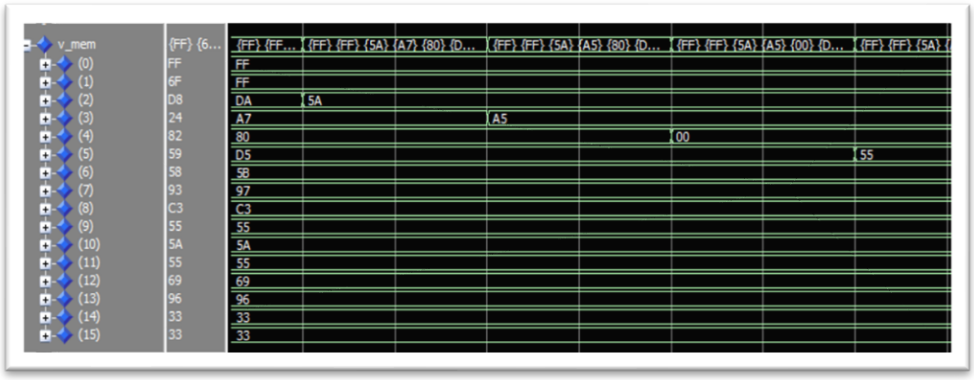

Ilustración Código 63. Formas de onda "PASO 4" RAM SECDED (3)

Lectura después de la corrección de errores

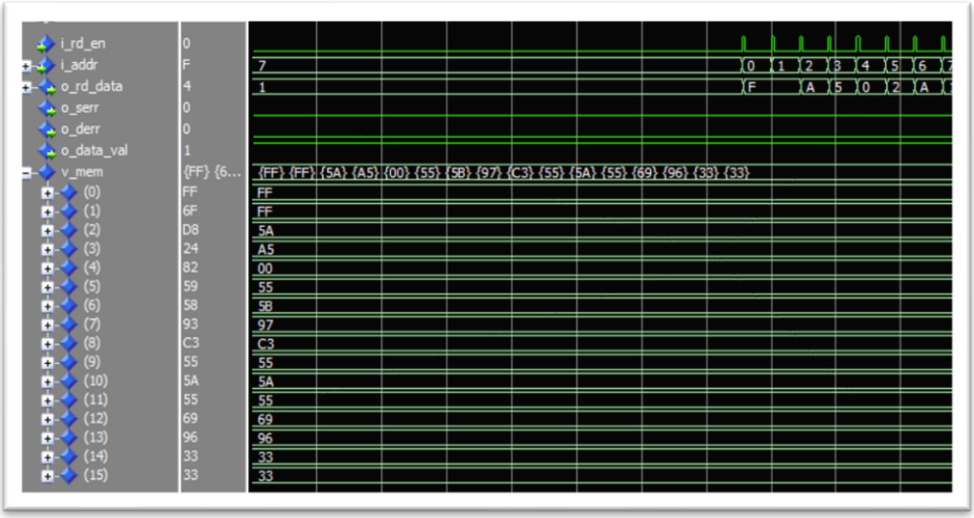

Ilustración Código 64. Formas de onda "PASO 4" RAM SECDED (4)

PASO 5: Forzar "Double Errors" en algunas direcciones de memoria y comprobar que los datos no son válidos

En este paso se inyectan "Double Errors", se comprueba que el algoritmo es capaz de detectar en qué posiciones de memoria se han producido y notificar que dichos datos no son válidos. Para llevar a cabo esta acción, se emplea el "procedure" diseñado previamente: "inject_derr".

En este paso, se inyectan "Double Errors" en los datos y se verifica la capacidad del algoritmo para identificar las posiciones de memoria afectadas. Si un dato resulta erróneo, se comprueba la notificación de dicho evento y la invalidez del dato al usuario.

```
// ---------------------------------------------------------------------------------------------
// PASO 5: Forzar "Double Errors" en algunas direcciones de memoria y comprobar que los datos no
// son válidos
// ---------------------------------------------------------------------------------------------

Escribir "Forzar "Double Errors" en algunas direcciones de memoria y comprobar que los datos no son válidos"

Escribir "Inyectando Double Errors: "

Para i ← 0 hasta 5
    Si i ≠ 3 Entonces
        Escribir "Double Error inyectado en la dirección "i_en_binario"
        inject_derr(i, valor_aleatorio, valor_aleatorio)
    FinSi
FinPara
Esperar 0.1 milisegundos

Escribir "Leer algunas direcciones de memoria, con y sin Single Errors inyectados, y comprobar las salidas detector_single_error,
        "detector_double_error y dato_valido: "
Para i ← 0 hasta 7
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    Caso i es
        Para i de (0 hasta 2) y (4 hasta 5)->
            check_value(dato_valido, desactivado, "Dato no válido")
            check_value(detector_single_error, activado, "No se ha detectado Single Error")
            check_value(detector_doble_error, desactivado, "Double Error detectado")

        Para OTROS_CASOS ->
            check_value(dato_valido, activado, "Dato válido")
            check_value(detector_single_error, desactivado, "No se ha detectado Single Error")
            check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")
    Fin Caso
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    Esperar 20 * CICLO_RELOJ
FinPara
```

Ilustración Código 65. Pseudocódigo estímulos "PASO 5" SRAM SECDED Testbench

- Formas de onda:

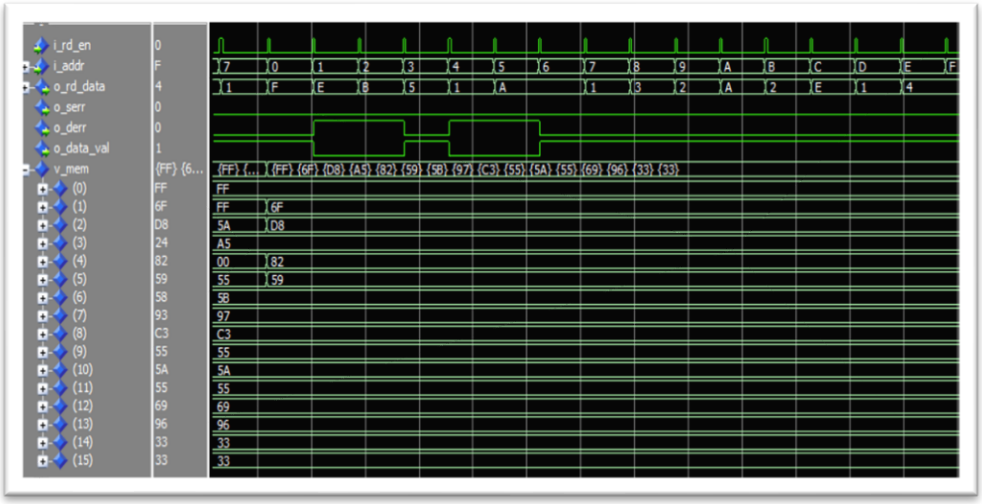

Ilustración Código 66. Formas de onda "PASO 5" RAM SECDED

PASO 6: Comprobar el comportamiento de la detección de errores cuando el bit 0 ha sufrido un SEU

En este paso se evalúa la detección de errores del algoritmo al enfrentarse a situaciones límite, en este caso, al sufrir un evento SEU en el bit 0 de la palabra de bits. Aunque el algoritmo no logra detectar ni corregir un error en esta posición específica de la secuencia, esta situación no supone un problema, dado que el dato original se mantiene inalterado. Sin embargo, si ocurre otro error en un bit diferente de la misma palabra, se deberá registrar como un “Double Error” y notificar al usuario su invalidez.

```
// ------------------------------------------------------------------------------------------------
//  PASO 6: Comprobar el comportamiento de la detección de errores cuando el bit 0 ha sufrido un SEU
// ------------------------------------------------------------------------------------------------

Escribir "Comprobar el comportamiento de la detección de errores cuando el bit 0 ha sufrido un SEU"

Escribir "Inyectando Errores: "
Escribir "Double Error inyectado en la dirección "3_en_binario"
inject_derr(3, 0, valor_aleatorio)

Escribir "Double Error inyectado en la dirección "6_en_binario"
inject_derr(6, 0, valor_aleatorio) // Esta posición ya se encuentra afectada previamente en el bit 0

Escribir "Single Error inyectado en la dirección "7_en_binario"
inject_serr(7, valor_aleatorio) // Esta posición ya se encuentra afectada previamente en el bit 0

Escribir "Leer algunas direcciones de memoria, con y sin Single Errors inyectados, y comprobar las salidas detector_single_error,
"detector_double_error y dato_valido: "
Para i ← 0 hasta 7
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    Caso i es
    Para i de (1 hasta 5) y igual que 7 ->
        check_value(dato_valido, desactivado, "Dato no válido")
        check_value(detector_single_error, activado, "No se ha detectado Single Error")
        check_value(detector_doble_error, desactivado, "Double Error detectado")

    Para i igual que 6 -> // El error en el bit 0 previamente inyectado en pasos anteriores se solventa al introducirse un error por segunda vez
        check_value(dato_valido, desactivado, "Dato no válido")
        check_value(detector_single_error, activado, "Single Error detectado")
        check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")

    Para OTROS_CASOS ->
        check_value(dato_valido, activado, "Dato válido")
        check_value(detector_single_error, desactivado, "No se ha detectado Single Error")
        check_value(detector_doble_error, desactivado, "No se ha detectado Doble Error")
    Fin Caso
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    Esperar 20 * CICLO_RELOJ
FinPara
Esperar 0.1 milisegundos
```

Ilustración Código 67. Pseudocódigo estímulos "PASO 6" SRAM SECDED Testbench

- Formas de onda:

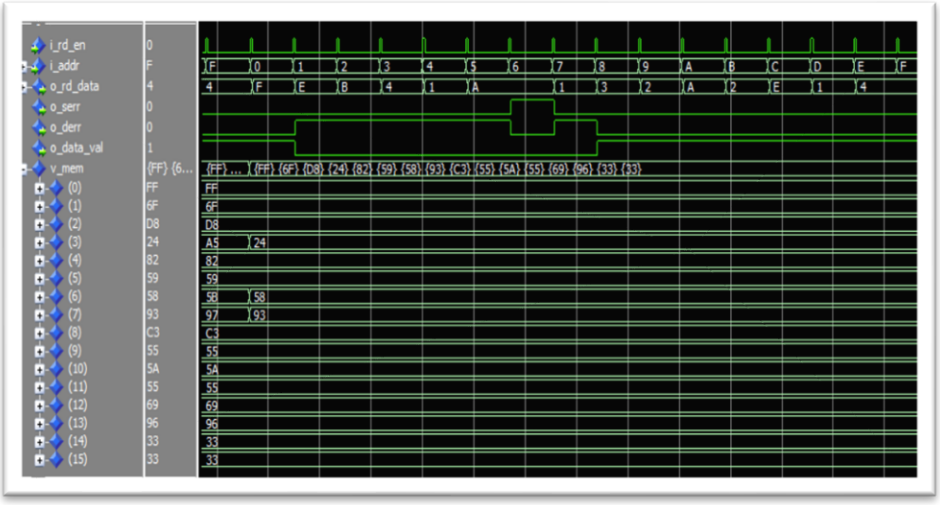

Ilustración Código 68. Formas de onda "PASO 6" RAM SECDED

PASO 7: Verificar diferentes periodos de "scrubbing"

En este paso se evalúa la configurabilidad del periodo de "scrubbing" para varios valores de división.

```
// ------------------------------------------------------------------------------------------------
// PASO 7: Verificar diferentes periodos de "scrubbing"
// ------------------------------------------------------------------------------------------------

Escribir "Verificar diferentes periodos de scrubbing"

Escribir "Ajustando divisor_scrubbing a "2_en_binario"
divisor_scrubbing ← 2_en_binario

Esperar hasta scrubbing_tick
tiempo ← tiempo_pasado
Esperar hasta scrubbing_tick
check_value(tiempo_actual - tiempo_pasado, 2 segundos, "El periodo de scrubbing es de 2 segundos")

Escribir "Ajustando divisor_scrubbing a "3_en_binario"
divisor_scrubbing ← 3_en_binario

Esperar hasta scrubbing_tick
tiempo ← tiempo_pasado
Esperar hasta scrubbing_tick
check_value(tiempo_actual - tiempo_pasado, 3 segundos, "El periodo de scrubbing es de 2 segundos")

Escribir "Ajustando divisor_scrubbing a "5_en_binario"
divisor_scrubbing ← 5_en_binario

Esperar hasta scrubbing_tick
tiempo ← tiempo_pasado
Esperar hasta scrubbing_tick
check_value(tiempo_actual - tiempo_pasado, 5 segundos, "El periodo de scrubbing es de 2 segundos")

Escribir "Ajustando divisor_scrubbing a "1_en_binario"
divisor_scrubbing ← 1_en_binario

Esperar hasta scrubbing_tick
tiempo ← tiempo_pasado
Esperar hasta scrubbing_tick
check_value(tiempo_actual - tiempo_pasado, 1 segundos, "El periodo de scrubbing es de 2 segundos")
```

Ilustración Código 69. Pseudocódigo estímulos "PASO 7" SRAM SECDED Testbench

- Formas de onda:

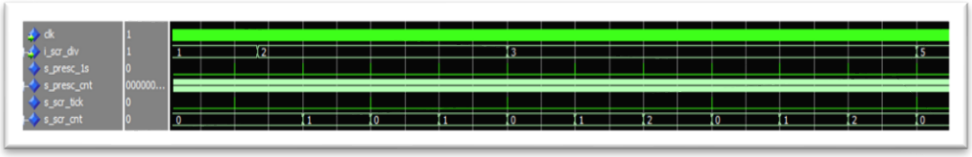

Ilustración Código 70. Formas de onda "PASO 7" RAM SECDED

Script de Python:

Para la compilación del diseño, se utiliza el siguiente script en lenguaje Python:

```
// ------------------------------------------------------------------------------------------------
//                                      SCRIPT PYTHON
// ------------------------------------------------------------------------------------------------

// Importar Librerias
importar_librerias(VUnit, VUnitCLI)

// Configurar argumentos
añadir_argumento(--g_reset_pol)         // Polaridad reset
añadir_argumento(--g_data_width)        // Ancho de datos
añadir_argumento(--g_addr_width)        // Ancho de dirección de memoria
añadir_argumento(--g_scr_div_width)     // Ancho de divisor de scrubbing
añadir_argumento(--g_main_freq)         // Frecuencia principal

// Declarar Librerias para compilación del diseño
declarar_libreria("ram_secded_lib")
declarar_libreria("hamming_pkg_lib")

// Declarar Libreria Testbench
declarar_libreria("tb_lib")

//Declarar Librerias UVVM
declarar_librerias_UVVM()

// Ajustar genéricos a los valores provistos por argumentos
Si g_reset_pol no es Nulo Entonces
    g_reset_pol = configurar_generico(--g_reset_pol)
SiNo
    g_reset_pol = Falso
FinSi

Si g_data_width no es Nulo Entonces
    g_data_width = configurar_generico(--g_data_width)
SiNo
    g_data_width = 4
FinSi

Si g_addr_width no es Nulo Entonces
    g_addr_width = configurar_generico(--g_addr_width)
SiNo
    g_addr_width = 4
FinSi

Si g_scr_div_width no es Nulo Entonces
    g_scr_div_width = configurar_generico(--g_scr_div_width)
SiNo
    g_scr_div_width = 4
FinSi

Si g_main_freq no es Nulo Entonces
    g_main_freq = configurar_generico(--g_main_freq)
SiNo
    g_main_freq = 1000000
FinSi

Algoritmo main ()

FinAlgoritmo
```

Ilustración Código 71. Pseudocódigo python script runner RAM SECDED

La librería “VUnitCLI” proporciona una interfaz de línea de comandos (CLI, Command Line Interface) para interactuar con “VUnit” desde la línea de comandos o scripts. En caso de no recibir argumentos por parte del usuario desde la consola, se establecen valores por defecto en cada genérico.

Los parámetros se transmiten consecutivos, indicando el nombre seguido por su correspondiente valor.

Ventana de comandos:

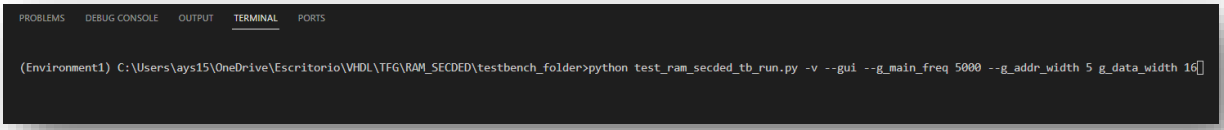

Ilustración 49. Ejemplo pasar argumentos por terminal de comandos

12.2 TESTBENCH MEMORIA SRAM CON TMR VHDL

Archivo: ram_tmr_tb.vhd

Entidad:

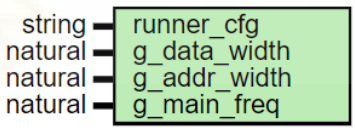

Ilustración 50. Entidad RAM TMR Testbench

Diagrama:

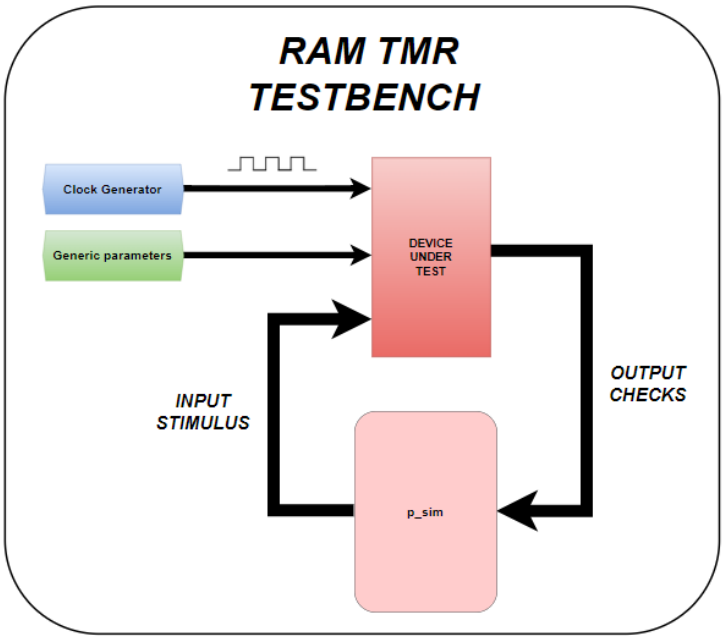

Ilustración 51. Diagrama RAM TMR Testbench

Descripción:

Este testbench testea la memoria SRAM con TMR (Triple Modular Redundancy).

Genéricos:

Nombre genérico	Tipo	Valor	Descripción
runner_cfg	string	nulo	archivo de configuración
g_data_width	natural	configurable	ancho de los datos
g_addr_width	natural	configurable	ancho de las direcciones
g_main_freq	natural	configurable	frecuencia principal de reloj

Tabla 10. Genéricos RAM TMR Testbench

Instancias:

device_under_test:

Esta instancia corresponde a la entidad de la memoria SRAM con TMR a verificar.

Generación de reloj:

Para la generación de reloj se ha hecho uso de la siguiente función de la librería UVVM:

```
clock_generation(reloj, periodo)
```

Ilustración Código 72. Generación de reloj

Donde se le asigna como primer argumento la señal de reloj, y como segundo argumento el periodo de reloj deseado.

- Formas de onda:

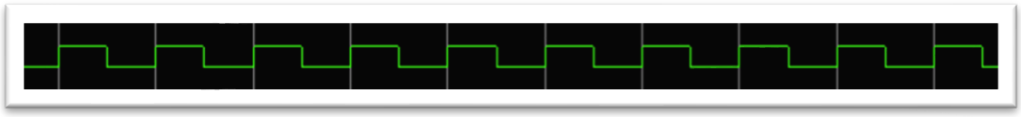

Ilustración Código 73. Forma de onda reloj RAM TMR Testbench

Procesos:

- **p_sim:**

Proceso para simular los estímulos que recibe el "device under test". En su interior se describen los pasos a testear y se verifica su comportamiento ante dichos estímulos.

Este es el cuerpo principal del test, donde se pone a prueba las características objetivo del diseño y su respuesta ante situaciones adversas.

Procedures:

En esta sección se diseñan los "procedures" que se utilizarán más adelante para la verificación.

inject_err_ram_x (0 < x < 2):

Estos "procedures" fuerzan una palabra errónea aleatoria en tantas posiciones de memoria, en orden ascendente, como desee el usuario (RAM x con [0 < x < 2]).

```
FinSubProceso inject_err_ram_0(numero_direcciones_memoria):
    Para i ← 0 hasta numero_direcciones_memoria
        memoria_0[i] ← valor_aleatorio
        Esperar 0.1 milisegundos
        Escribir "Palabra ("valor_aleatorio") inyectada en la dirección de memoria "i"
    FinPara
FinSubProceso

FinSubProceso inject_err_ram_1(numero_direcciones_memoria):
    Para i ← 0 hasta numero_direcciones_memoria
        memoria_1[i] ← valor_aleatorio
        Esperar 0.1 milisegundos
        Escribir "Palabra ("valor_aleatorio") inyectada en la dirección de memoria "i"
    FinPara
FinSubProceso

FinSubProceso inject_err_ram_2(numero_direcciones_memoria):
    Para i ← 0 hasta numero_direcciones_memoria
        memoria_2[i] ← valor_aleatorio
        Esperar 0.1 milisegundos
        Escribir "Palabra ("valor_aleatorio") inyectada en la dirección de memoria "i"
    FinPara
FinSubProceso
```

Ilustración Código 74. Pseudocódigo procedures "inject_err_ram_(0 < x < 2)

Secuencia de test:

PASO 0: Inicializar entradas y generar Tabla

En este paso se inicializan las entradas y se genera una tabla con valores aleatorios, que serán posteriormente utilizados como datos a escribir en la memoria y como referencia para las comprobaciones.

```
// ------------------------------------------------------------------------------------------------
//  PASO 0: Inicializar entradas y generar Tabla
// ------------------------------------------------------------------------------------------------

Escribir "Inicializar entradas y generar Tabla"

Escribir "Inicializando entradas"
inicializacion_entradas()

Escribir "Generando Tabla Aleatoria: "
Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    tabla_aleatoria[i] ← valor_aleatorio
    Escribir "Escribiendo valor "tabla_aleatoria[i]" en Tabla Aleatoria (posición "i")
FinPara
Esperar 0.1 milisegundos
```

Ilustración Código 75. Pseudocódigo estímulos "PASO 0" RAM TMR Testbench

PASO 1: Escribir valores en todas las direcciones de la SRAM

En este paso se testea la escritura de los datos en cada posición de memoria de la SRAM, dando los correspondientes estímulos a las entradas para realizar la operación.

```
// ------------------------------------------------------------------------------------------
//  PASO 1: Escribir valores en todas las direcciones de la SRAM
// ------------------------------------------------------------------------------------------

Escribir "Escribir valores en todas las direcciones de la SRAM"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_escritura ← activado
    direccion_memoria     ← i_en_binario
    dato_entrada          ← tabla_aleatoria[i]
    Esperar CICLO_RELOJ

    Escribir "Escribiendo valor "tabla_aleatoria[i]" en la dirección "direccion_memoria"
    Esperar CICLO_RELOJ

    habilitador_escritura ← desactivado
    Esperar 20 * CICLO_RELOJ
FinPara
```

Ilustración Código 76. Pseudocódigo estímulos "PASO 1" RAM TMR Testbench

- Formas de onda:

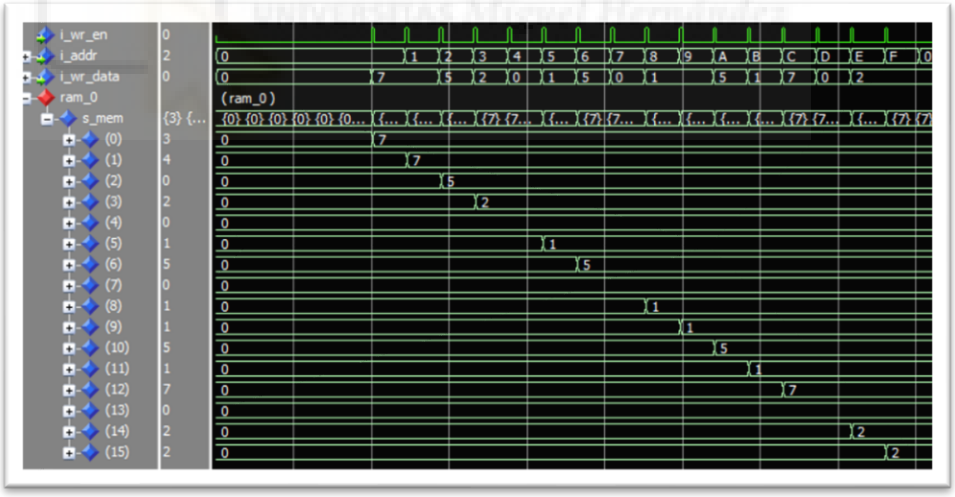

Ilustración Código 77. Formas de onda "PASO 1" RAM TMR

PASO 2: Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los escritos en el PASO 1

Este paso se comprueba la lectura de la memoria SRAM, dando los correspondientes estímulos a las entradas para realizar la operación. Una vez leído el valor, se comprueba que coincide con el escrito en el paso anterior mediante la función “check_value”.

check_value(valor_actual, valor_deseado, nivel_alerta[opcional], mensaje, [...])

```
// ------------------------------------------------------------------------------------------
//  PASO 2: Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los
//  escritos en el PASO 1
// ------------------------------------------------------------------------------------------

Escribir "Leer los valores en todas las direcciones de la SRAM y comprobar que coinciden con los escritos en el PASO 1"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección: "dirección_memoria") Lectura:  "dato_salida"      Esperado: "tabla_aleatoria[i])
    Esperar 20 * CICLO_RELOJ
FinPara
```

Ilustración Código 78. Pseudocódigo estímulos "PASO 2" RAM TMR Testbench

- Formas de onda:

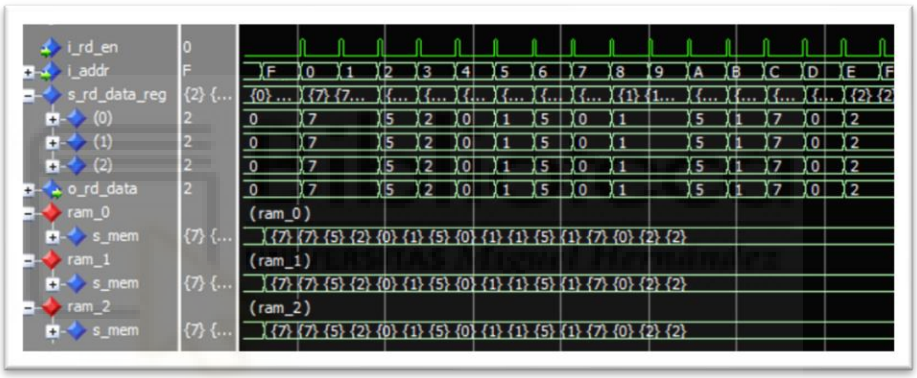

Ilustración Código 79. Formas de onda "PASO 2" RAM TMR

PASO 3: Comprobar la detección de colisión de escritura/lectura

Este paso pone a prueba la detección de colisión entre escritura/lectura. Para ello, se genera la situación descrita habilitando la escritura y lectura en un mismo instante. Se espera que tanto la memoria como la salida de lectura no se actualicen durante ese intervalo, con el fin de evitar errores.

```
// ---------------------------------------------------------------------------------------------
// PASO 3: Comprobar la detección de colisión de escritura/lectura
// ---------------------------------------------------------------------------------------------

Escribir "Comprobar la detección de colisión de escritura/lectura"

Para i ← 0 hasta 2
    Escribir "Forzar colisión habilitando la escritura y lectura, escribiendo "(todos_los_bits '0')" en la dirección "i_en_binario"

    Escribir "Comprobar que el inhibidor de lectura/escritura está activo y no se ha realizado ninguna operación"
    habilitador_escritura ← activado
    habilitador_lectura   ← activado
    direccion_memoria     ← i_en_binario
    dato_entrada          ← (todos_los_bits '0')
    Esperar CICLO_RELOJ

    habilitador_escritura ← desactivado
    habilitador_lectura   ← desactivado
    check_value(inhibidor, activado, "Inhibidor de lectura/escritura activado")
    check_value(hablitador_escritura, desactivado, "Operación de escritura desactivada")
    check_value(habilitador_lectura, desactivado, "Operación de lectura desactivada")
    Esperar CICLO_RELOJ
FinPara
Esperar 0.01 milisegundos

Para i ← 0 hasta 2
    Escribir "Comprobar dato en la dirección "i_en_binario" no ha sido modificado"
    habilitador_lectura ← activado
    dirección_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección "direccion_memoria") Lectura: "dato_salida"       Esperado: "tabla_aleatoria[i])
    Esperar CICLO_RELOJ
FinPara
```

Ilustración Código 80. Pseudocódigo estímulos "PASO 3" RAM TMR Testbench

- Formas de onda:

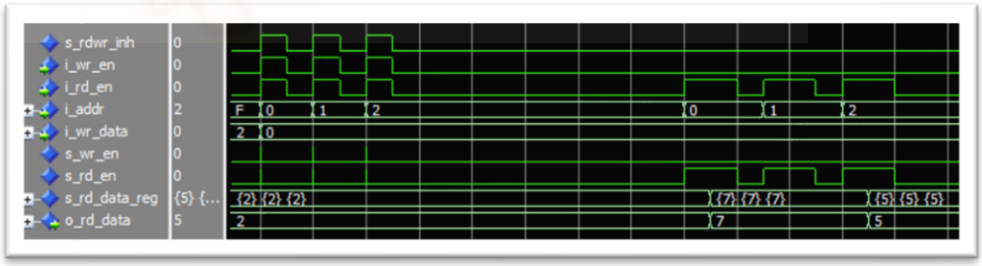

Ilustración Código 81. Formas de onda "PASO 3" RAM TMR

PASO 4: Forzar errores en una de las SRAM interna y comprobar que los datos de lectura coinciden con los escritos en el PASO 1

En este paso se testea el comportamiento del sistema de votación de la memoria SRAM con TMR. Para ello, se introducen errores en una de las SRAM internas y se comprueba que la lectura de salida sigue coincidiendo con los datos escritos originalmente. En la señal de registro de salida se puede observar que las lecturas de una de las memorias instanciadas no coinciden con las lecturas de las dos memorias restantes.

```
// ------------------------------------------------------------------------------------------
//  PASO 4: Forzar errores en una de las SRAM internas y comprobar que los datos de lectura coinciden
//  con los escritos en el PASO 1
// ------------------------------------------------------------------------------------------

Escribir "Forzar errores en una de las SRAM internas y comprobar que los datos de lectura coinciden con los escritos en el PASO 1"

Escribir "Forzando errores en varias direcciones de memoria de la SRAM_0: "
inject_err_ram_0(10)

Escribir "Leer todas las direcciones de memoria, con y sin datos erroneos inyectados, y comprobar las salidas dato_salida y dato_valido
Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar 2 * CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_valido, activado, "Dato válido")
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección: "dirección_memoria") Lectura:  "dato_salida"     Esperado: "tabla_aleatoria[i])
    Esperar CICLO_RELOJ
FinPara

Escribir "Reescribir los valores en la SRAM para sobreescribir los errores inyectados"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección: "dirección_memoria") Lectura:  "dato_salida"     Esperado: "tabla_aleatoria[i])
    Esperar 20 * CICLO_RELOJ
FinPara

Escribir "Repetir para las memorias internas SRAM_1 y SRAM_2"
```

Ilustración Código 82. Pseudocódigo estímulos "PASO 4" RAM TMR Testbench

- Formas de onda:

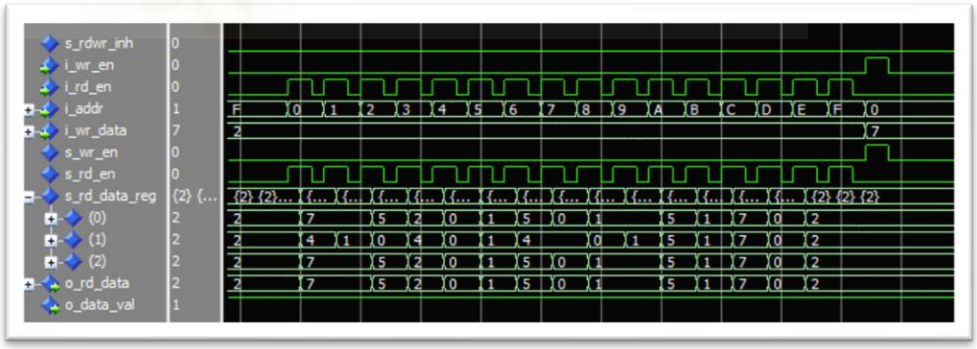

Ilustración Código 83. Formas de onda "PASO 4" RAM TMR

PASO 5: Forzar errores en varias direcciones de memoria de dos de las RAMs internas y comprobar que los datos de lectura son obsoletos

En este último paso se evalúa el comportamiento erróneo del sistema de votación de la memoria SRAM con TMR. Para ello, se introducen errores en dos de las tres SRAM internas y se comprueba que la lectura de salida no coincide con la escritura previa en la memoria, además de la activación de la salida que informa que el dato no es válido.

```
// ------------------------------------------------------------------------------------------
//  PASO 5: Forzar errores en varias direcciones de memoria de dos de las RAMs internas y comprobar
//  que los datos de lectura son obsoletos
// ------------------------------------------------------------------------------------------

Escribir "PASO 5: Forzar errores en varias direcciones de memoria de dos de las RAMs internas y comprobar que los datos de lectura son obsoletos"

Escribir "Forzando errores en varias direcciones de memoria de la SRAM_0 y SRAM_1: "
inject_err_ram_0(11)
inject_err_ram_1(11)

Escribir "Leer todas las direcciones de memoria, con y sin datos erroneos inyectados, y comprobar la salida dato_valido
Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar 2 * CICLO_RELOJ

    habilitador_lectura ← desactivado
    Escribir "(Dirección: "dirección_memoria") Lectura:  "dato_salida"    Esperado: "tabla_aleatoria[i])

    Si mem_0(i) es distinto de mem_1(i) y mem_0(i) es distinto de mem_2(i) y mem_1(i) es distinto de mem_2(i) Entonces
        check_value(dato_valido, desactivado, "Dato no válido (obsoleto)")
    SiNo
        check_value(dato_valido, activado, "Dato válido")
    FinSi
    Esperar CICLO_RELOJ
FinPara

Escribir "Reescribir los valores en la SRAM para sobreescribir los errores inyectados"

Para i ← 0 hasta DIRECCION_SRAM_MAXIMA
    habilitador_lectura ← activado
    direccion_memoria   ← i_en_binario
    Esperar CICLO_RELOJ

    habilitador_lectura ← desactivado
    check_value(dato_salida, tabla_aleatoria[i], "(Dirección: "dirección_memoria") Lectura:  "dato_salida"    Esperado: "tabla_aleatoria[i])
    Esperar 20 * CICLO_RELOJ
FinPara

Escribir "Repetir para los siguientes casos: Forzar errores en SRAM_0 y SRAM_2 / Forzar errores en SRAM_1 y SRAM_2"
```

Ilustración Código 84. Pseudocódigo estímulos "PASO 5" RAM TMR Testbench

- Formas de onda:

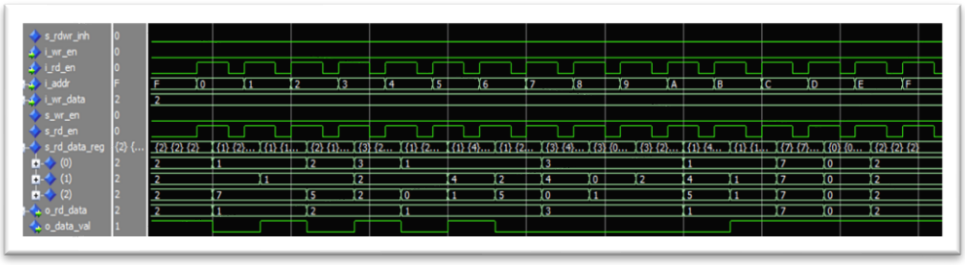

Ilustración Código 85. Formas de onda "PASO 5" RAM TMR

Script de Python:

Para la compilación del diseño, se utiliza el siguiente script en lenguaje Python:

```
// ------------------------------------------------------------------------------------------------
//                                      SCRIPT PYTHON
// ------------------------------------------------------------------------------------------------

// Importar Librerias
importar_librerias(VUnit, VUnitCLI)

// Configurar argumentos
añadir_argumento(--g_data_width)        // Ancho de datos
añadir_argumento(--g_addr_width)        // Ancho de dirección de memoria
añadir_argumento(--g_main_freq)         // Frecuencia principal

// Declarar Librerias para compilación del diseño
declarar_libreria("ram_tmr_lib")

// Declarar Libreria Testbench
declarar_libreria("tb_lib")

//Declarar Librerias UVVM
declarar_librerias_UVVM()

// Ajustar genéricos a los valores provistos por argumentos
Si g_data_width no es Nulo Entonces
    g_data_width = configurar_generico(--g_data_width)
SiNo
    g_data_width = 4
FinSi

Si g_addr_width no es Nulo Entonces
    g_addr_width = configurar_generico(--g_addr_width)
SiNo
    g_addr_width = 4
FinSi

Si g_main_freq no es Nulo Entonces
    g_main_freq = configurar_generico(--g_main_freq)
SiNo
    g_main_freq = 1000000
FinSi

Algoritmo main ()

FinAlgoritmo
```


Ilustración Código 86. Pseudocódigo python script runner RAM TMR

Al igual que el script de Python del diseño anterior, se utiliza la librería “VUnitCLI” para proveer una interfaz de línea de comandos.

13. INFORMES DE ANÁLISIS DE RECURSOS Y RENDIMIENTO

A continuación, se presentan los informes de síntesis y tiempos generados mediante Vivado. Estos informes proporcionan la información necesaria, junto con los "testbenches", para poder realizar una comparación entre las implementaciones de ambos códigos de corrección de errores.

13.1 INFORME DE SÍNTESIS DE DISEÑO (USO DE RECURSOS DE LA FPGA)

Este informe provisto por Vivado muestra el tamaño del diseño, junto con los recursos de la FPGA empleados, como LUTs (Lookup Tables), elementos biestables (flip-flops), bloques de memoria, puertas lógicas, multiplexores, entre otros. Todo ello se traduce a hardware real utilizado para la implementación del diseño.

- SRAM (SECDED):

Lógica Interna:

	Usada	Disponible	Tipo
Celdas LUT	92	3750	Lógica
Celdas Registros	77	7500	Biestables

Tabla 11. Lógica interna RAM SECDED

Información detallada del componente RTL:

	Longitud	Cantidad
Sumadores	4 bits	3
Puertas XOR	1,3,4,7 bits	11
Registros	1,3,4,8 bits	15
RAM	(16 x 8) bits	1
Multiplexores	Variados	16

Tabla 12. Componente RTL RAM SECDED

- SRAM (TMR):

Lógica Interna:

	Usada	Disponible	Tipo
Celdas LUT	6	3750	2 Lógica 4 Memoria
Celdas Registros	4	7500	Biestables

Tabla 13. Lógica interna RAM TMR

Información detallada del componente RTL:

	Longitud	Cantidad
Registros	4 bits	1
RAM	(16 x 4) bits	3
Multiplexores	Variados	2

Tabla 14. Componente RTL RAM TMR

13.2 INFORME DE TIEMPOS (LATENCIA)

Este informe muestra los detalles sobre tiempo y rendimiento del diseño. En él se pueden destacar las rutas críticas, las cuales son rutas necesarias para el correcto funcionamiento del módulo y que muestran los tiempos de acceso más largos, y más cortos, a la memoria. Junto a dicha ruta, se proporciona información sobre el tiempo transcurrido para que la señal vaya a través de esta.

Existen dos tipos de rutas:

- Rutas de retardo máximo: rutas cuyo tiempo transcurrido en viajar desde su origen (source) y su destino (destination) es máximo.

- Rutas de retardo mínimo: rutas cuyo tiempo transcurrido en viajar desde el origen (source) y su destino (destination) es mínimo.

Para observar cuál de los dos diseños es más rápido, bastará con centrarse en la ruta con mayor tiempo de retardo.

- SRAM (SECDED):

Ruta de retardo máximo:

Source:	v_mem (registro de memoria de la Dual Port RAM)
Destination:	o_serr (flag de salida indicando un Single Error)
Delay:	9.738 ns (Lógica = 4.937 ns / Ruta = 4.801 ns)

De acuerdo con el informe, la ruta con mayor retardo es la ruta desde que se realiza una lectura en la memoria hasta que se activa/desactiva la salida "o_serr". Esto es provocado por dos factores:

- Lógica (4.937 ns): Se refiere al tiempo de retraso debido a la lógica interna del circuito. Indica que la señal requiere de 4.937 nanosegundos en atravesar las puertas lógicas y otros componentes de procesamiento presentes dentro del circuito.
- Ruta (4.801 ns): Se refiere a que el retardo de la señal a lo largo de la ruta física seguida por la misma en el circuito es de 4.801 nanosegundos, incluyendo el tiempo en atravesar el enrutado, interconexiones, entre otros.

- SRAM (TMR):

Source:	s_rd_data_reg (registro de memoria de la Single Port RAM)
Destination:	o_rd_data (Salida de lectura de la RAM TMR)
Delay:	4.312 ns (Lógica = 2.758 ns / Ruta = 1.554 ns)

De acuerdo con el informe, la ruta con mayor retardo es la ruta desde que se realiza una lectura en la memorias SRAM internas hasta la salida de lectura global después del proceso de votación "o_rd_data". Al igual que en el caso anterior, esto es provocado por dos factores:

- Lógica (2.758 ns): La señal requiere de 2.758 nanosegundos en atravesar las puertas lógicas y otros componentes de procesamiento presentes dentro del circuito.

- Ruta (1.554 ns): El retardo de la señal a lo largo de la ruta física seguida por la misma en el circuito es de 1.554 nanosegundos, incluyendo el tiempo en atravesar el enrutado, interconexiones, entre otros.

13.3 REQUISITOS DE FRECUENCIA

La frecuencia de reloj de ambos módulos es configurable por el usuario, lo que proporciona una gran versatilidad para abordar los retardos propios del diseño. No obstante, es importante señalar que esta flexibilidad no garantiza automáticamente el cumplimiento de los requisitos de funcionamiento para cualquier periodo de reloj.

Con el fin de prevenir que los retardos provoquen un comportamiento no deseado en la lógica del diseño, es fundamental asegurar que la ruta de máximo retardo pueda completarse antes de la llegada del siguiente flanco de reloj. En consecuencia, la elección de la frecuencia se encuentra restringida por este retardo, siendo un factor determinante en la configuración óptima del sistema. El cálculo de la frecuencia más alta de funcionamiento para cada módulo es el siguiente:

RAM SECDED:

$$\frac{1}{9.738 * 10^{-9} segundos} = 102.69\ MHz$$

RAM TMR:

$$\frac{1}{4.312 * 10^{-9} segundos} = 231.91\ MHz$$

$$\begin{cases} Frecuencia\ Reloj_{RAM\ SECDED} \leq 102.69\ MHz \\ Frecuencia\ Reloj_{RAM\ TMR} \leq 231.91\ MHz \end{cases}$$

13.4 ESQUEMA DE POTENCIA (CONSUMO ELÉCTRICO)

Muestra la potencia consumida por la FPGA tras la implementación del módulo diseñado.

- SRAM (SECDED):

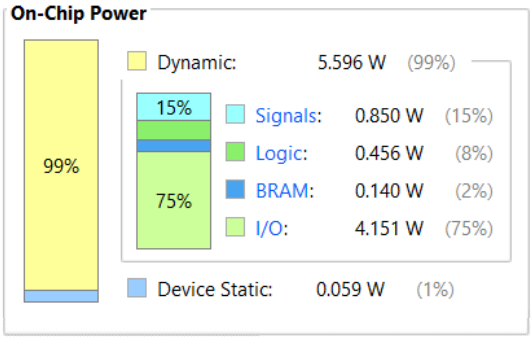

Ilustración 52. Esquema de potencia SRAM SECDED

- SRAM (TMR):

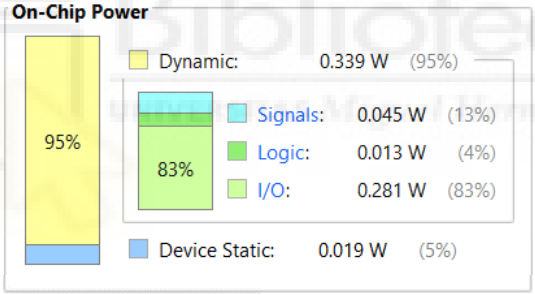

Ilustración 53. Esquema de potencia SRAM TMR

El esquema proporciona información sobre la distribución de la potencia en los diferentes componentes de la FPGA. Se dividen en dos bloques principales:

Potencia dinámica (Consumo dinámico):

- Representa la mayor parte de la potencia total consumida por la FPGA.
- Incluye subdivisiones de consumo de potencia relacionados con las señales del circuito (enrutado), la lógica interna(puertas lógicas, flip-flops, etc) y las entradas/salidas de la FPGA.
- La potencia dinámica puede aumentar o disminuir al variar la frecuencia de reloj, ya que esto afecta a la frecuencia de conmutación de los transistores, provocando cambios en el consumo de potencia.

Potencia estática (Consumo estático):

- Representa el consumo estático de la FPGA, es decir, un consumo inevitable durante el funcionamiento del módulo.
- Incluye corrientes de fuga de los transistores que componen la memoria SRAM, entre otros elementos que contribuyan al consumo de potencia cuando no hay cambios en el estado lógico.

Debido a que se pretende tener en cuenta únicamente los gastos referentes a la lógica digital empleada en cada diseño, se excluyen los consumos asociados a las conexiones con los pines de entrada/salida de la FPGA. En aplicaciones reales, las memorias SRAM generalmente no se encuentran directamente conectadas a los pines de entrada/salida, sino que actúan como bloques o buffers intermedios conectados entre dos componentes o IP. Por lo tanto, la potencia relacionada con las conexiones de entrada/salida no se tiene en cuenta en los resultados de consumo.

	RAM SECDED	RAM TMR
Estática	0.059 W	0.019 W
Dinámica	1.446 W	0.058 W
Total	1.505 W	0.077 W

Tabla 15. Consumo de potencia total SRAM SECDED y SRAM TMR

14. CONCLUSIONES

Durante el transcurso de este Trabajo de Fin de Grado se ha se ha realizado un estudio del Estado del Arte, diseño y verificación de dos memorias SRAM con diferentes técnicas para abordar la corrección de errores, tales como el algoritmo de Hamming y la Triple Redundancia Modular, con el objetivo de comparar ambas implementaciones en cuanto a efectividad y recursos consumidos de la FPGA. Tras el estudio, se ha proporcionado un pseudocódigo cada diseño, así como de test de prueba realizados, formas de onda e informes de síntesis, implementación, latencia, y potencia.

A la vista de los resultados obtenidos y posterior estudio, se puede concluir lo siguiente:

La memoria SRAM con SECDED es muy eficiente en términos de ahorro de memoria, puesto que gracias a la aplicación del Código Hamming es capaz de proteger los datos ante inducción radioactiva introduciendo un número muy reducido de bits adicionales para su identificación y posterior corrección. También se encuentra alineada de acuerdo

con el objetivo de la ley de Moore, la cual buscaba una reducción del área del hardware en los circuitos digitales. Sin embargo, su aplicación es compleja debido al diseño y control del algoritmo, así como posee un mayor tiempo de respuesta, ya que los datos han de codificarse antes de introducirse en la memoria y decodificarse, además de comprobarse, al ser leídos. En cuanto a consumo de recursos de la FPGA, utiliza una mayor cantidad de LUTs y puertas lógicas (lo que favorece dicha latencia) y su consumo en potencia es mayor. Por último, se debe tener en cuenta que su capacidad de corrección de errores está limitada a un único SEU, con posible detección de dos errores, pero sin posibilidad de modificación dado dicho caso.

La memoria SRAM con TMR es muy eficiente en términos de corrección de errores, gracias a que implica la instanciación de tres memorias SRAM internas. Esto dota a la memoria de una gran capacidad de resistencia ante radiación, ya que no importa la cantidad de SEU que ocurra en diferentes bits del dato, el sistema de votación sacará el dato correcto en el proceso de lectura. También posee una menor latencia, al no interactuar ningún algoritmo para corrección de errores y tener un menor número de lógica digital, adicionalmente su consumo en potencia es mucho menor. Sin embargo, no esta aplicación de corrección de errores no autocorrige el error, sino que permanece "oculto" hasta su nueva escritura. Esto podría requerir hardware adicional que identifique y reescriba el dato erróneo, ya sea implementando un algoritmo de corrección de errores o una reescritura cada cierto periodo de tiempo (refresco), entre otros métodos. Además, este modelo no está alineado con el objetivo de la ley de Moore, debido a que la implementación de tres memorias SRAM internas equivale a una ocupación contundente de área en comparación. Finalmente, y aunque muy improbable, en la hipótesis de que suceda un SEU en la misma posición de dos de las tres memorias SRAM, el sistema de votación no sería capaz de identificar el dato incorrecto (obsoleto) y se necesitaría de lógica adicional para detectar y sobrescribir dicho dato.

En definitiva, la utilización de uno u otro método de corrección de errores dependerá del nivel de radiación del entorno sobre el que se pretende funcionar. En niveles bajos de radiación cósmica, por ejemplo, en órbitas cercanas a la Tierra, la utilización de la memoria SRAM con SECDED sería la óptima, ya que es muy improbable que ocurran muchos SEU en diferentes bits del dato almacenado. De esta forma, se consigue un ahorro de memoria y un periodo de "scrubbing" alto, lo que no afectaría demasiado al rendimiento, aun sacrificando potencia y latencia si los requisitos de diseño de su utilización no son muy estrictos sobre el tiempo de respuesta y consumo. Por otro lado, en entornos de alta radiación cósmica, por ejemplo, en órbitas más lejanas a la Tierra, la utilización de la memoria SRAM con TMR es mejor opción. En este caso, se sacrifica área, memoria y utilización de recursos de la FPGA para la protección de los datos,

dejando abierta la posibilidad de añadir hardware adicional que permita su corrección. Esto último no sería un problema en cuestión de potencia, gracias a su bajo consumo.

14.1 LÍNEAS FUTURAS

Partiendo de la investigación realizada en este Trabajo de Fin de Grado, se plantean posibles líneas de trabajo futuras para validar el comportamiento de dichos diseños.

Se propone la realización de un test “Post-Layout” o “post-diseño”. Este test tiene el objetivo de verificar y validar el rendimiento, latencia, funcionalidad e integridad del diseño después de su implementación física. Mediante dichas pruebas, se podrá realizar un análisis más exhaustivo de temporización y latencias inducidos por la lógica digital, los efectos “parásitos” que perjudiquen el rendimiento del diseño, y permite visualizar estímulos más realistas que ayudan a reparar en un diseño más robusto en cuanto a metaestabilidad. El efecto de dichos eventos también se puede observar sobre la forma de onda.

Finalmente se concede la libertad de considerar una prueba en físico ante diferentes estímulos de radiación y comprobar con un osciloscopio la respuesta real de las señales y salidas en diferentes puntos del circuito, así como su integridad, sincronización con el reloj y análisis de temporización de acceso a la memoria.

15. REFERENCIAS

[1] **D. A. Kelleher, «Intel» 16 Febrero 2022. [En línea]. Available:** https://www.intel.la/content/www/xl/es/newsroom/opinion/moore-law-now-and-in-the-future.html. [Último acceso: 25 Febrero 2024].

[2] **M. Aguirre, «Geeknetic» 18 10 2017. [En línea]. Available:** https://www.geeknetic.es/Editorial/1406/La-realidad-sobre-los-nanometros-en-procesos-de-fabricacion-de-CPUs-y-GPUs.html. [Último acceso: 25 Febrero 2024].

[3] **E. Pérez, «Xataka» 30 Septiembre 2021. [En línea]. Available:** https://www.xataka.com/componentes/hay-fecha-para-transistores-pequenos-que-1-nanometro-grandes-fabricantes-chips-planean-pasar-a-angstroms-2030. [Último acceso: 25 Febrero 2024].

[4] **Anónimo, «Wikipedia» 27 Octubre 2023. [En línea]. Available:** https://es.wikipedia.org/wiki/Radiaci%C3%B3n_c%C3%B3smica. [Último acceso: 25 Febrero 2024].

[5] **L. Gil, «IAEA» 7 Julio 2021. [En línea]. Available:** https://www.iaea.org/es/newscenter/news/radiacion-cosmica-por-que-no-deberia-ser-motivo-de-preocupacion-en-ingles#:~:text=La%20radiaci%C3%B3n%20c%C3%B3smica%20gal%C3%A1ctica%20procede,o%20se%20destruyen%20por%20completo.. [Último acceso: 25 Febrero 2024].

[6] **J. C. López, «Xataka» 2 Abril 2021. [En línea]. Available:** https://www.xataka.com/espacio/radiacion-cosmica-que-donde-procede-que-nos-protege-ella. [Último acceso: 25 Febrero 2024].

[7] **Anónimo, «Wikipedia» 14 Noviembre 2023. [En línea]. Available:** https://es.wikipedia.org/wiki/SRAM#:~:text=Una%20memoria%20SRAM%20tiene%20tres,datos%20almacenados%20en%20la%20memoria.. [Último acceso: 25 Febrero 2024].

[8] **Anónimo, «Wikipedia» 17 Julio 2023. [En línea]. Available:** https://es.wikipedia.org/wiki/DRAM#:~:text=La%20memoria%20din%C3%A1mica%20de%20acceso,revisa%20dicha%20carga%20y%20la. [Último acceso: 25 Febrero 2024].

[9] **Anónimo, «Electricity - Magnetism». [En línea]. Available:** https://www.electricity-magnetism.org/es/sram-memoria-estatica-de-acceso-aleatorio/. [Último acceso: 25 Febrero 2024].

[10] **Anónimo, «Genera Tecnologías». [En línea]. Available:** https://www.generatecnologias.es/fpga-espacio.html. [Último acceso: 25 Febrero 2024].

[11] **J. Harris, «Planet Analog» 25 Enero 2018. [En línea]. Available:** https://www.planetanalog.com/a-quick-overview-of-radiation-effects-single-event-effects/#:~:text=The%20single%20event%20effects%20observed,event%20gate%20rupture%2

0(SEGR).. [Último acceso: 25 Febrero 2024].

[12] **L. Williams, «GURU99» 17 Febrero 2024. [En línea]. Available:** https://www.guru99.com/es/hamming-code-error-correction-example.html. [Último acceso: 25 Febrero 2024].

[13] **Anónimo, «Wolfram-Mathematica» [En línea]. Available:** https://wolfram-mathematica.website/evita-errores-en-tus-datos-con-el-codigo-de-hamming-en-mathematica/. [Último acceso: 25 Febrero 2024].

[14] **Anónimo, «Foro Histórico de las Telecomunicaciones» [En línea]. Available:** https://forohistorico.coit.es/index.php/personajes/personajes-internacionales/item/hamming-richard-w. [Último acceso: 25 Febrero 2024].

[15] **R. Invarato, «Jarroba» 5 Octubre 2016. [En línea]. Available:** https://jarroba.com/codigo-de-hamming-deteccion-y-correccion-de-errores/. [Último acceso: 25 Febrero 2024].

[16] **G. Sanderson, «3Blue1Brown» 4 Septiembre 2020. [En línea]. Available:** https://www.3blue1brown.com/lessons/hamming-codes#thanks. [Último acceso: 25 Febrero 2024].

[17] **Anónimo, «Wikipedia» 12 Septiembre 2023. [En línea]. Available:** https://en.wikipedia.org/wiki/Triple_modular_redundancy. [Último acceso: 25 Febrero 2024].

[18] **V. Elamaran, «Semantic Scholar» 2018. [En línea]. Available:** https://www.semanticscholar.org/paper/MAJORITY-VOTER-CIRCUITS-OF-TMR-CONFIGURATION-%E2%80%93-A-Elamaran-Chandrasekar/805f96b633b1bc4967c9057444b5f0911da1d47d. [Último acceso: 25 Febrero 2024].

[19] **«Git Hub» 2022. [En línea]. Available:** https://github.com/UVVM/UVVM/blob/master/uvvm_util/doc/util_quick_ref.pdf. [Último acceso: 25 Febrero 2024].

ANEXO I: TRANSCRIPCIÓN SRAM (SECDED)

Historial de simulación memoria SRAM SECDED.

```
ID_LOG_HDR                    0.0 ns  TB seq.                    STEP 0: Assert/deassert Reset after 100 us and generate Tables.
-------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER                  0.0 ns  TB seq.
ID_SEQUENCER                  0.0 ns  TB seq.                     Initializing inputs.
ID_SEQUENCER                  0.0 ns  TB seq.
ID_SEQUENCER                  0.0 ns  TB seq.                     Generating Random Table :
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xF in Random Table (position 0).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xF in Random Table (position 1).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xA in Random Table (position 2).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x5 in Random Table (position 3).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x0 in Random Table (position 4).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 5).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xA in Random Table (position 6).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 7).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x3 in Random Table (position 8).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 9).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xA in Random Table (position 10).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 11).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0xE in Random Table (position 12).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 13).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x4 in Random Table (position 14).
ID_SEQUENCER                  0.0 ns  TB seq.                     Writing value 0x4 in Random Table (position 15).
ID_SEQUENCER             100000.0 ns  TB seq.
ID_SEQUENCER             100000.0 ns  TB seq.                     Generating Hamming Table :
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x7F in Hamming Table (position 0).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x7F in Hamming Table (position 1).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2D in Hamming Table (position 2).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x52 in Hamming Table (position 3).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x00 in Hamming Table (position 4).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2A in Hamming Table (position 5).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2D in Hamming Table (position 6).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x4B in Hamming Table (position 7).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x61 in Hamming Table (position 8).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2A in Hamming Table (position 9).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2D in Hamming Table (position 10).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x2A in Hamming Table (position 11).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x34 in Hamming Table (position 12).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x4B in Hamming Table (position 13).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x19 in Hamming Table (position 14).
ID_SEQUENCER             100000.0 ns  TB seq.                     Writing value 0x19 in Hamming Table (position 15).
ID_POS_ACK               101000.0 ns  TB seq.                     check_value() => OK, for std_logic '1' (exp: '1'). 'Reset deasserted.'

ID_LOG_HDR               201000.0 ns  TB seq.                    STEP 1: Write values in every address of the SRAM.
-------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER             202000.0 ns  TB seq.                    Writing value 0xF in address 0x0.
ID_SEQUENCER             224000.0 ns  TB seq.                    Writing value 0xF in address 0x1.
ID_SEQUENCER             246000.0 ns  TB seq.                    Writing value 0xA in address 0x2.
ID_SEQUENCER             268000.0 ns  TB seq.                    Writing value 0x5 in address 0x3.
ID_SEQUENCER             290000.0 ns  TB seq.                    Writing value 0x0 in address 0x4.
ID_SEQUENCER             312000.0 ns  TB seq.                    Writing value 0x2 in address 0x5.
ID_SEQUENCER             334000.0 ns  TB seq.                    Writing value 0xA in address 0x6.
ID_SEQUENCER             356000.0 ns  TB seq.                    Writing value 0x1 in address 0x7.
ID_SEQUENCER             378000.0 ns  TB seq.                    Writing value 0x3 in address 0x8.
ID_SEQUENCER             400000.0 ns  TB seq.                    Writing value 0x2 in address 0x9.
ID_SEQUENCER             422000.0 ns  TB seq.                    Writing value 0xA in address 0xA.
ID_SEQUENCER             444000.0 ns  TB seq.                    Writing value 0x2 in address 0xB.
ID_SEQUENCER             466000.0 ns  TB seq.                    Writing value 0xE in address 0xC.
ID_SEQUENCER             488000.0 ns  TB seq.                    Writing value 0x1 in address 0xD.
ID_SEQUENCER             510000.0 ns  TB seq.                    Writing value 0x4 in address 0xE.
ID_SEQUENCER             532000.0 ns  TB seq.                    Writing value 0x4 in address 0xF.

ID_LOG_HDR               553000.0 ns  TB seq.                    STEP 2: Read values from every address of the SRAM and check that it matches with previous
                                                                 written data in STEP 1.
-------------------------------------------------------------------------------------------------------------------------------------------------------
ID_POS_ACK               554000.0 ns  TB seq.                    check_value() => OK, for slv x"F"'. '(Address 0x0) Read: 0xF   Expected: 0xF'
ID_POS_ACK               575000.0 ns  TB seq.                    check_value() => OK, for slv x"F"'. '(Address 0x1) Read: 0xF   Expected: 0xF'
ID_POS_ACK               596000.0 ns  TB seq.                    check_value() => OK, for slv x"A"'. '(Address 0x2) Read: 0xA   Expected: 0xA'
ID_POS_ACK               617000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address 0x3) Read: 0x5   Expected: 0x5'
ID_POS_ACK               638000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address 0x4) Read: 0x0   Expected: 0x0'
ID_POS_ACK               659000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address 0x5) Read: 0x2   Expected: 0x2'
ID_POS_ACK               680000.0 ns  TB seq.                    check_value() => OK, for slv x"A"'. '(Address 0x6) Read: 0xA   Expected: 0xA'
ID_POS_ACK               701000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address 0x7) Read: 0x1   Expected: 0x1'
ID_POS_ACK               722000.0 ns  TB seq.                    check_value() => OK, for slv x"3"'. '(Address 0x8) Read: 0x3   Expected: 0x3'
ID_POS_ACK               743000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address 0x9) Read: 0x2   Expected: 0x2'
ID_POS_ACK               764000.0 ns  TB seq.                    check_value() => OK, for slv x"A"'. '(Address 0xA) Read: 0xA   Expected: 0xA'
ID_POS_ACK               785000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address 0xB) Read: 0x2   Expected: 0x2'
ID_POS_ACK               806000.0 ns  TB seq.                    check_value() => OK, for slv x"E"'. '(Address 0xC) Read: 0xE   Expected: 0xE'
ID_POS_ACK               827000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address 0xD) Read: 0x1   Expected: 0x1'
ID_POS_ACK               848000.0 ns  TB seq.                    check_value() => OK, for slv x"4"'. '(Address 0xE) Read: 0x4   Expected: 0x4'
ID_POS_ACK               869000.0 ns  TB seq.                    check_value() => OK, for slv x"4"'. '(Address 0xF) Read: 0x4   Expected: 0x4'

ID_LOG_HDR               889000.0 ns  TB seq.                    STEP 3: Check read and write collision detection.
-------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER             889000.0 ns  TB seq.                    Force collision by activating write enable and read enable, write data 0x0 in address 0x2
ID_SEQUENCER             899000.0 ns  TB seq.
ID_SEQUENCER             899000.0 ns  TB seq.                    Check read/write inhibit is enabled and no operation has been done
ID_POS_ACK               899000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Read/Write inhibit is enabled'
ID_POS_ACK               899000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Write operation is disabled'
ID_POS_ACK               899000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Read operation is disabled'
ID_SEQUENCER             909000.0 ns  TB seq.
ID_SEQUENCER             909000.0 ns  TB seq.                    Check data in Address 0x2 has not been modified
ID_POS_ACK               919000.0 ns  TB seq.                    check_value() => OK, for slv x"A"'. '(Address 0x2) Read: 0xA   Expected: 0xA'
```

```
ID_LOG_HDR                939000.0 ns  TB seq.                    STEP 4: Force Single Errors in some addresses of the SRAM and check they are corrected.
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER              939000.0 ns  TB seq.
ID_SEQUENCER              939000.0 ns  TB seq.                    Injecting Single Errors :
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected in address 0x0
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected in address 0x2
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected in address 0x3
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected in address 0x4
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected in address 0x5
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected (bit 0) address 0x6
ID_SEQUENCER              940000.0 ns  TB seq.                    Single Error injected (bit 0) address 0x7
ID_SEQUENCER             1040000.0 ns  TB seq.
ID_SEQUENCER             1040000.0 ns  TB seq.                    Read from some addresses with and without Single Errors injected and check 'o_serr', 'o_derr'
                                                                  and 'o_data_val' outputs:
ID_SEQUENCER             1041000.0 ns  TB seq.
ID_SEQUENCER             1041000.0 ns  TB seq.                    (Address: 0x0) Read: 0x8   Expected: 0xF
ID_POS_ACK               1041000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid'
ID_POS_ACK               1041000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_SEQUENCER             1063000.0 ns  TB seq.
ID_SEQUENCER             1063000.0 ns  TB seq.                    (Address: 0x1) Read: 0xF   Expected: 0xF
ID_POS_ACK               1063000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK               1063000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               1063000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_SEQUENCER             1085000.0 ns  TB seq.
ID_SEQUENCER             1085000.0 ns  TB seq.                    (Address: 0x2) Read: 0x8   Expected: 0xA
ID_POS_ACK               1085000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid'
ID_POS_ACK               1085000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_SEQUENCER             1107000.0 ns  TB seq.
ID_SEQUENCER             1107000.0 ns  TB seq.                    (Address: 0x3) Read: 0x5   Expected: 0x5
ID_POS_ACK               1107000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid'
ID_POS_ACK               1107000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_SEQUENCER             1129000.0 ns  TB seq.
ID_SEQUENCER             1129000.0 ns  TB seq.                    (Address: 0x4) Read: 0x1   Expected: 0x0
ID_POS_ACK               1129000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid'
ID_POS_ACK               1129000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_SEQUENCER             1151000.0 ns  TB seq.
ID_SEQUENCER             1151000.0 ns  TB seq.                    (Address: 0x5) Read: 0x3   Expected: 0x2
ID_POS_ACK               1151000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid'
ID_POS_ACK               1151000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_SEQUENCER             1173000.0 ns  TB seq.
ID_SEQUENCER             1173000.0 ns  TB seq.                    (Address: 0x6) Read: 0xA   Expected: 0xA
ID_POS_ACK               1173000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK               1173000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               1173000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_SEQUENCER             1173000.0 ns  TB seq.                    Case corner: Error injected in bit 0 (global parity) of extended hamming encoded word
ID_SEQUENCER             1173000.0 ns  TB seq.                    Error can not be either detected nor corrected (it does not affect the data of the word, so it
                                                                  still valid)
ID_SEQUENCER             1195000.0 ns  TB seq.
ID_SEQUENCER             1195000.0 ns  TB seq.                    (Address: 0x7) Read: 0x1   Expected: 0x1
ID_POS_ACK               1195000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK               1195000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               1195000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_SEQUENCER             1195000.0 ns  TB seq.                    Case corner: Error injected in bit 0 (global parity) of extended hamming encoded word
ID_SEQUENCER             1195000.0 ns  TB seq.                    Error can not be either detected nor corrected (it does not affect the data of the word, so it
                                                                  still valid)
ID_SEQUENCER             1216000.0 ns  TB seq.
ID_SEQUENCER             1216000.0 ns  TB seq.                    Wait 8 seconds for the EDAC to correct the Single Errors

ID_POS_ACK               8001198000.0 ns  TB seq.                 check_value() => OK, for slv x"7F"'. 'Single Error corrected in address 0x0'
ID_POS_ACK               8001198000.0 ns  TB seq.                 check_value() => OK, for slv x"2D"'. 'Single Error corrected in address 0x2'
ID_POS_ACK               8001198000.0 ns  TB seq.                 check_value() => OK, for slv x"52"'. 'Single Error corrected in address 0x2'
ID_POS_ACK               8001198000.0 ns  TB seq.                 check_value() => OK, for slv x"00"'. 'Single Error corrected in address 0x4'
ID_POS_ACK               8001198000.0 ns  TB seq.                 check_value() => OK, for slv x"2A"'. 'Single Error corrected in address 0x5'
ID_SEQUENCER             8001298000.0 ns  TB seq.
ID_SEQUENCER             8001298000.0 ns  TB seq.                 Read the same addresses and check Single Errors have been corrected:
ID_SEQUENCER             8001299000.0 ns  TB seq.
ID_SEQUENCER             8001299000.0 ns  TB seq.                 Address = 0x0
ID_POS_ACK               8001299000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001299000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001299000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001321000.0 ns  TB seq.
ID_SEQUENCER             8001321000.0 ns  TB seq.                 Address = 0x1
ID_POS_ACK               8001321000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001321000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001321000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001343000.0 ns  TB seq.
ID_SEQUENCER             8001343000.0 ns  TB seq.                 Address = 0x2
ID_POS_ACK               8001343000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001343000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001343000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001365000.0 ns  TB seq.
ID_SEQUENCER             8001365000.0 ns  TB seq.                 Address = 0x3
ID_POS_ACK               8001365000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001365000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001365000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001387000.0 ns  TB seq.
ID_SEQUENCER             8001387000.0 ns  TB seq.                 Address = 0x4
ID_POS_ACK               8001387000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001387000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001387000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001409000.0 ns  TB seq.
ID_SEQUENCER             8001409000.0 ns  TB seq.                 Address = 0x5
ID_POS_ACK               8001409000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001409000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001409000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001431000.0 ns  TB seq.
ID_SEQUENCER             8001431000.0 ns  TB seq.                 Address = 0x6
ID_POS_ACK               8001431000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001431000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001431000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER             8001453000.0 ns  TB seq.
ID_SEQUENCER             8001453000.0 ns  TB seq.                 Address = 0x7
ID_POS_ACK               8001453000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK               8001453000.0 ns  TB seq.                 check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK               8001453000.0 ns  TB seq.                 check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
```

```
ID_LOG_HDR              8001474000.0 ns  TB seq.                    STEP 5: Force Double Errors in some addresses of the SRAM and check that data is not valid.
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER            8001474000.0 ns  TB seq.
ID_SEQUENCER            8001474000.0 ns  TB seq.                    Injecting Double Errors :
ID_SEQUENCER            8001474000.0 ns  TB seq.                    Double Error injected in address 0x1
ID_SEQUENCER            8001474000.0 ns  TB seq.                    Double Error injected in address 0x2
ID_SEQUENCER            8001474000.0 ns  TB seq.                    Double Error injected in address 0x4
ID_SEQUENCER            8001474000.0 ns  TB seq.                    Double Error injected in address 0x5
ID_SEQUENCER            8001574000.0 ns  TB seq.
ID_SEQUENCER            8001574000.0 ns  TB seq.                    Read values from the address of the SRAM and check if the data word is valid (o_data_value =
                                                                    '1').
ID_SEQUENCER            8001575000.0 ns  TB seq.
ID_SEQUENCER            8001575000.0 ns  TB seq.                    (Address: 0x0) Read: 0xF   Expected: 0xF
ID_POS_ACK              8001575000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001575000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001575000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001596000.0 ns  TB seq.
ID_SEQUENCER            8001596000.0 ns  TB seq.                    (Address: 0x1) Read: 0xE   Expected: 0xF
ID_POS_ACK              8001596000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001596000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8001596000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8001617000.0 ns  TB seq.
ID_SEQUENCER            8001617000.0 ns  TB seq.                    (Address: 0x2) Read: 0xB   Expected: 0xA
ID_POS_ACK              8001617000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001617000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8001617000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8001638000.0 ns  TB seq.
ID_SEQUENCER            8001638000.0 ns  TB seq.                    (Address: 0x3) Read: 0x5   Expected: 0x5
ID_POS_ACK              8001638000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001638000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001638000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001659000.0 ns  TB seq.
ID_SEQUENCER            8001659000.0 ns  TB seq.                    (Address: 0x4) Read: 0x1   Expected: 0x0
ID_POS_ACK              8001659000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001659000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8001659000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8001680000.0 ns  TB seq.
ID_SEQUENCER            8001680000.0 ns  TB seq.                    (Address: 0x5) Read: 0xA   Expected: 0x2
ID_POS_ACK              8001680000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001680000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8001680000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8001701000.0 ns  TB seq.
ID_SEQUENCER            8001701000.0 ns  TB seq.                    (Address: 0x6) Read: 0xA   Expected: 0xA
ID_POS_ACK              8001701000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001701000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001701000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001722000.0 ns  TB seq.
ID_SEQUENCER            8001722000.0 ns  TB seq.                    (Address: 0x7) Read: 0x1   Expected: 0x1
ID_POS_ACK              8001722000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001722000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001722000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001743000.0 ns  TB seq.
ID_SEQUENCER            8001743000.0 ns  TB seq.                    (Address: 0x8) Read: 0x3   Expected: 0x3
ID_POS_ACK              8001743000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001743000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001743000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001764000.0 ns  TB seq.
ID_SEQUENCER            8001764000.0 ns  TB seq.                    (Address: 0x9) Read: 0x2   Expected: 0x2
ID_POS_ACK              8001764000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001764000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001764000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001785000.0 ns  TB seq.
ID_POS_ACK              8001785000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001806000.0 ns  TB seq.
ID_SEQUENCER            8001806000.0 ns  TB seq.                    (Address: 0xB) Read: 0x2   Expected: 0x2
ID_POS_ACK              8001806000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001806000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001806000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001827000.0 ns  TB seq.
ID_SEQUENCER            8001827000.0 ns  TB seq.                    (Address: 0xC) Read: 0xE   Expected: 0xE
ID_POS_ACK              8001827000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001827000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001827000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001848000.0 ns  TB seq.
ID_SEQUENCER            8001848000.0 ns  TB seq.                    (Address: 0xD) Read: 0x1   Expected: 0x1
ID_POS_ACK              8001848000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001848000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001848000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001869000.0 ns  TB seq.
ID_SEQUENCER            8001869000.0 ns  TB seq.                    (Address: 0xE) Read: 0x4   Expected: 0x4
ID_POS_ACK              8001869000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001869000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001869000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8001890000.0 ns  TB seq.
ID_SEQUENCER            8001890000.0 ns  TB seq.                    (Address: 0xF) Read: 0x4   Expected: 0x4
ID_POS_ACK              8001890000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8001890000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8001890000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
```

```
ID_LOG_HDR              8001910000.0 ns  TB seq.                    STEP 6: Check error detection behaviour when bit 0 has suffered a bit flip.
-------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER            8001910000.0 ns  TB seq.
ID_SEQUENCER            8001910000.0 ns  TB seq.                    Injecting Errors:
ID_SEQUENCER            8001910000.0 ns  TB seq.                    Double Error injected in address 0x3, one of them in bit 0
ID_SEQUENCER            8001910000.0 ns  TB seq.                    Double Error injected in address 0x6, one of them in bit 0 (which previously suffered a SEU)
ID_SEQUENCER            8001910000.0 ns  TB seq.                    Single Error injected in address 0x7 (which previously suffered a SEU in bit 0)
ID_SEQUENCER            8002010000.0 ns  TB seq.
ID_SEQUENCER            8002010000.0 ns  TB seq.                    Read values from the addresses and check behaviour where one error affected bit 0:
ID_SEQUENCER            8002011000.0 ns  TB seq.
ID_SEQUENCER            8002011000.0 ns  TB seq.                    (Address: 0x0) Read: 0xF   Expected: 0xF
ID_POS_ACK              8002011000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002011000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002011000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002032000.0 ns  TB seq.
ID_SEQUENCER            8002032000.0 ns  TB seq.                    (Address: 0x1) Read: 0xE   Expected: 0xF
ID_POS_ACK              8002032000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002032000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002032000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002053000.0 ns  TB seq.
ID_SEQUENCER            8002053000.0 ns  TB seq.                    (Address: 0x2) Read: 0xB   Expected: 0xA
ID_POS_ACK              8002053000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002053000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002053000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002074000.0 ns  TB seq.
ID_SEQUENCER            8002074000.0 ns  TB seq.                    (Address: 0x3) Read: 0x4   Expected: 0x5
ID_POS_ACK              8002074000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002074000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002074000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002074000.0 ns  TB seq.                    Error detection still working properly
ID_SEQUENCER            8002095000.0 ns  TB seq.
ID_SEQUENCER            8002095000.0 ns  TB seq.                    (Address: 0x4) Read: 0x1   Expected: 0x0
ID_POS_ACK              8002095000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002095000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002095000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002116000.0 ns  TB seq.
ID_SEQUENCER            8002116000.0 ns  TB seq.                    (Address: 0x5) Read: 0xA   Expected: 0x2
ID_POS_ACK              8002116000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002116000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002116000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002137000.0 ns  TB seq.
ID_SEQUENCER            8002137000.0 ns  TB seq.                    (Address: 0x6) Read: 0xA   Expected: 0xA
ID_POS_ACK              8002137000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Single Error detected'
ID_POS_ACK              8002137000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002137000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002137000.0 ns  TB seq.                    A second SEU occurring in bit 0 autocorrected the previous SEU injected in STEP 4, so it
                                                                    detects a Single Error and it's able to correct it in the scrubbing
ID_SEQUENCER            8002158000.0 ns  TB seq.
ID_SEQUENCER            8002158000.0 ns  TB seq.                    (Address: 0x7) Read: 0x1   Expected: 0x1
ID_POS_ACK              8002158000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002158000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Double Error detected'
ID_POS_ACK              8002158000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (Error detected)'
ID_SEQUENCER            8002158000.0 ns  TB seq.                    Error detection still working properly
ID_SEQUENCER            8002179000.0 ns  TB seq.
ID_SEQUENCER            8002179000.0 ns  TB seq.                    (Address: 0x8) Read: 0x3   Expected: 0x3
ID_POS_ACK              8002179000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002179000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002179000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002200000.0 ns  TB seq.
ID_SEQUENCER            8002200000.0 ns  TB seq.                    (Address: 0x9) Read: 0x2   Expected: 0x2
ID_POS_ACK              8002200000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002200000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002200000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002221000.0 ns  TB seq.
ID_SEQUENCER            8002221000.0 ns  TB seq.                    (Address: 0xA) Read: 0xA   Expected: 0xA
ID_POS_ACK              8002221000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002221000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002221000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002242000.0 ns  TB seq.
ID_SEQUENCER            8002242000.0 ns  TB seq.                    (Address: 0xB) Read: 0x2   Expected: 0x2
ID_POS_ACK              8002242000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002242000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002242000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002263000.0 ns  TB seq.
ID_SEQUENCER            8002263000.0 ns  TB seq.                    (Address: 0xC) Read: 0xE   Expected: 0xE
ID_POS_ACK              8002263000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002263000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002263000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002284000.0 ns  TB seq.
ID_SEQUENCER            8002284000.0 ns  TB seq.                    (Address: 0xD) Read: 0x1   Expected: 0x1
ID_POS_ACK              8002284000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002284000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002284000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002305000.0 ns  TB seq.
ID_SEQUENCER            8002305000.0 ns  TB seq.                    (Address: 0xE) Read: 0x4   Expected: 0x4
ID_POS_ACK              8002305000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002305000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002305000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
ID_SEQUENCER            8002326000.0 ns  TB seq.
ID_SEQUENCER            8002326000.0 ns  TB seq.                    (Address: 0xF) Read: 0x4   Expected: 0x4
ID_POS_ACK              8002326000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Single Error detected'
ID_POS_ACK              8002326000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'No Double Error detected'
ID_POS_ACK              8002326000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid (No error detected)'
```

```
ID_LOG_HDR              8617522000.0 ns  TB seq.                    STEP 7: Check different scrubbing periods when i_scr_div input is modified.
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER            8617522000.0 ns  TB seq.
ID_SEQUENCER            8617522000.0 ns  TB seq.                    Setting i_scr_div to 0x2
ID_POS_ACK             12000100000.0 ns  TB seq.                    check_value() => OK, for time 2000000000000 ps. 'Scrub period is 2 s.'
ID_SEQUENCER           12000100000.0 ns  TB seq.
ID_SEQUENCER           12000100000.0 ns  TB seq.                    Setting i_scr_div to 0x3
ID_POS_ACK             18000100000.0 ns  TB seq.                    check_value() => OK, for time 3000000000000 ps. 'Scrub period is 3 s.'
ID_SEQUENCER           18000100000.0 ns  TB seq.
ID_SEQUENCER           18000100000.0 ns  TB seq.                    Setting i_scr_div to 0x5
ID_POS_ACK             28000100000.0 ns  TB seq.                    check_value() => OK, for time 5000000000000 ps. 'Scrub period is 5 s.'
ID_SEQUENCER           28000100000.0 ns  TB seq.
ID_SEQUENCER           28000100000.0 ns  TB seq.                    Setting i_scr_div to 0x1
ID_POS_ACK             30000100000.0 ns  TB seq.                    check_value() => OK, for time 1000000000000 ps. 'Scrub period is 1 s.'

    ============================================================================================================================================================
    *** FINAL SUMMARY OF ALL ALERTS ***
    ============================================================================================================================================================
                            REGARDED   EXPECTED  IGNORED      Comment?
             NOTE         :      0         0         0         ok
             TB_NOTE      :      0         0         0         ok
             WARNING      :      0         0         0         ok
             TB_WARNING   :      0         0         0         ok
             MANUAL_CHECK :      0         0         0         ok
             ERROR        :      0         0         0         ok
             TB_ERROR     :      0         0         0         ok
             FAILURE      :      0         0         0         ok
             TB_FAILURE   :      0         0         0         ok
    ============================================================================================================================================================
    >> Simulation SUCCESS: No mismatch between counted and expected serious alerts
    ============================================================================================================================================================



ID_LOG_HDR             30000200000.0 ns  TB seq.(uvvm)              SIMULATION COMPLETED
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

ANEXO II: TRANSCRIPCIÓN SRAM (TMR)

Historial de simulación memoria SRAM TMR.

```
ID_LOG_HDR                      0.0 ns  TB seq.                     STEP 0: Initialize data inputs and generate Random Data Table.
-------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER                    0.0 ns  TB seq.
ID_SEQUENCER                    0.0 ns  TB seq.                     Initializing inputs.
ID_SEQUENCER                    0.0 ns  TB seq.
ID_SEQUENCER                    0.0 ns  TB seq.                     Generating Random Table :
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x7 in Random Table (position 0).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x7 in Random Table (position 1).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x5 in Random Table (position 2).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 3).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x0 in Random Table (position 4).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 5).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x5 in Random Table (position 6).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x0 in Random Table (position 7).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 8).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 9).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x5 in Random Table (position 10).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x1 in Random Table (position 11).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x7 in Random Table (position 12).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x0 in Random Table (position 13).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 14).
ID_SEQUENCER                    0.0 ns  TB seq.                     Writing value 0x2 in Random Table (position 15).


ID_LOG_HDR                 100000.0 ns  TB seq.                     STEP 1: Write values in every address of the SRAM.
-------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER              2100000.0 ns  TB seq.                     Writing value 0x7 in address 0x0.
ID_SEQUENCER             24100000.0 ns  TB seq.                     Writing value 0x7 in address 0x1.
ID_SEQUENCER             46100000.0 ns  TB seq.                     Writing value 0x5 in address 0x2.
ID_SEQUENCER             68100000.0 ns  TB seq.                     Writing value 0x2 in address 0x3.
ID_SEQUENCER             90100000.0 ns  TB seq.                     Writing value 0x0 in address 0x4.
ID_SEQUENCER            112100000.0 ns  TB seq.                     Writing value 0x1 in address 0x5.
ID_SEQUENCER            134100000.0 ns  TB seq.                     Writing value 0x5 in address 0x6.
ID_SEQUENCER            156100000.0 ns  TB seq.                     Writing value 0x0 in address 0x7.
ID_SEQUENCER            178100000.0 ns  TB seq.                     Writing value 0x1 in address 0x8.
ID_SEQUENCER            200100000.0 ns  TB seq.                     Writing value 0x1 in address 0x9.
ID_SEQUENCER            222100000.0 ns  TB seq.                     Writing value 0x5 in address 0xA.
ID_SEQUENCER            244100000.0 ns  TB seq.                     Writing value 0x1 in address 0xB.
ID_SEQUENCER            266100000.0 ns  TB seq.                     Writing value 0x7 in address 0xC.
ID_SEQUENCER            288100000.0 ns  TB seq.                     Writing value 0x0 in address 0xD.
ID_SEQUENCER            310100000.0 ns  TB seq.                     Writing value 0x2 in address 0xE.
ID_SEQUENCER            332100000.0 ns  TB seq.                     Writing value 0x2 in address 0xF.

ID_LOG_HDR              352100000.0 ns  TB seq.                     STEP 2: Read values from every address of the SRAM and check that it matches with previous
                                                                    written data in STEP 1.
-------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_POS_ACK              354100000.0 ns  TB seq.                     check_value() => OK, for slv x"7"'. '(Address 0x0) Read: 0x7   Expected: 0x7'
ID_POS_ACK              376100000.0 ns  TB seq.                     check_value() => OK, for slv x"7"'. '(Address 0x1) Read: 0x7   Expected: 0x7'
ID_POS_ACK              398100000.0 ns  TB seq.                     check_value() => OK, for slv x"5"'. '(Address 0x2) Read: 0x5   Expected: 0x5'
ID_POS_ACK              420100000.0 ns  TB seq.                     check_value() => OK, for slv x"2"'. '(Address 0x3) Read: 0x2   Expected: 0x2'
ID_POS_ACK              442100000.0 ns  TB seq.                     check_value() => OK, for slv x"0"'. '(Address 0x4) Read: 0x0   Expected: 0x0'
ID_POS_ACK              464100000.0 ns  TB seq.                     check_value() => OK, for slv x"1"'. '(Address 0x5) Read: 0x1   Expected: 0x1'
ID_POS_ACK              486100000.0 ns  TB seq.                     check_value() => OK, for slv x"5"'. '(Address 0x6) Read: 0x5   Expected: 0x5'
ID_POS_ACK              508100000.0 ns  TB seq.                     check_value() => OK, for slv x"0"'. '(Address 0x7) Read: 0x0   Expected: 0x0'
ID_POS_ACK              530100000.0 ns  TB seq.                     check_value() => OK, for slv x"1"'. '(Address 0x8) Read: 0x1   Expected: 0x1'
ID_POS_ACK              552100000.0 ns  TB seq.                     check_value() => OK, for slv x"1"'. '(Address 0x9) Read: 0x1   Expected: 0x1'
ID_POS_ACK              574100000.0 ns  TB seq.                     check_value() => OK, for slv x"5"'. '(Address 0xA) Read: 0x5   Expected: 0x5'
ID_POS_ACK              596100000.0 ns  TB seq.                     check_value() => OK, for slv x"1"'. '(Address 0xB) Read: 0x1   Expected: 0x1'
ID_POS_ACK              618100000.0 ns  TB seq.                     check_value() => OK, for slv x"7"'. '(Address 0xC) Read: 0x7   Expected: 0x7'
ID_POS_ACK              640100000.0 ns  TB seq.                     check_value() => OK, for slv x"0"'. '(Address 0xD) Read: 0x0   Expected: 0x0'
ID_POS_ACK              662100000.0 ns  TB seq.                     check_value() => OK, for slv x"2"'. '(Address 0xE) Read: 0x2   Expected: 0x2'
ID_POS_ACK              684100000.0 ns  TB seq.                     check_value() => OK, for slv x"2"'. '(Address 0xF) Read: 0x2   Expected: 0x2'


ID_LOG_HDR              704100000.0 ns  TB seq.                     STEP 3: Check read and write collision detection.
-------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER            704100000.0 ns  TB seq.
ID_SEQUENCER            704100000.0 ns  TB seq.                     Force collision by activating write enable and read enable, write data 0x0 in address 0x0
ID_SEQUENCER            704100000.0 ns  TB seq.
ID_SEQUENCER            704100000.0 ns  TB seq.                     Check read/write inhibit is enabled and no operation has been done
ID_POS_ACK              705100000.0 ns  TB seq.                     check_value() => OK, for std_logic '1' (exp: '1'). 'Read/Write inhibit is enabled'
ID_POS_ACK              705100000.0 ns  TB seq.                     check_value() => OK, for std_logic '0' (exp: '0'). 'Write operation is disabled'
ID_POS_ACK              705100000.0 ns  TB seq.                     check_value() => OK, for std_logic '0' (exp: '0'). 'Read operation is disabled'
ID_SEQUENCER            706100000.0 ns  TB seq.
ID_SEQUENCER            706100000.0 ns  TB seq.                     Force collision by activating write enable and read enable, write data 0x0 in address 0x1
ID_SEQUENCER            706100000.0 ns  TB seq.
ID_SEQUENCER            706100000.0 ns  TB seq.                     Check read/write inhibit is enabled and no operation has been done
ID_POS_ACK              707100000.0 ns  TB seq.                     check_value() => OK, for std_logic '1' (exp: '1'). 'Read/Write inhibit is enabled'
ID_POS_ACK              707100000.0 ns  TB seq.                     check_value() => OK, for std_logic '0' (exp: '0'). 'Write operation is disabled'
ID_POS_ACK              707100000.0 ns  TB seq.                     check_value() => OK, for std_logic '0' (exp: '0'). 'Read operation is disabled'
ID_SEQUENCER            708100000.0 ns  TB seq.
ID_SEQUENCER            708100000.0 ns  TB seq.                     Force collision by activating write enable and read enable, write data 0x0 in address 0x2
ID_SEQUENCER            708100000.0 ns  TB seq.
```

```
ID_SEQUENCER              708100000.0 ns  TB seq.                   Check read/write inhibit is enabled and no operation has been done
ID_POS_ACK                709100000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Read/Write inhibit is enabled'
ID_POS_ACK                709100000.0 ns  TB seq.                   check_value() => OK, for std_logic '0' (exp: '0'). 'Write operation is disabled'
ID_POS_ACK                709100000.0 ns  TB seq.                   check_value() => OK, for std_logic '0' (exp: '0'). 'Read operation is disabled'
ID_SEQUENCER              710110000.0 ns  TB seq.
ID_SEQUENCER              710110000.0 ns  TB seq.                   Check data in Address 0x0 has not been modified
ID_POS_ACK                712110000.0 ns  TB seq.                   check_value() => OK, for slv x"7"'. '(Address 0x0) Read: 0x7   Expected: 0x7'
ID_SEQUENCER              713110000.0 ns  TB seq.
ID_SEQUENCER              713110000.0 ns  TB seq.                   Check data in Address 0x1 has not been modified
ID_POS_ACK                715110000.0 ns  TB seq.                   check_value() => OK, for slv x"7"'. '(Address 0x1) Read: 0x7   Expected: 0x7'
ID_SEQUENCER              716110000.0 ns  TB seq.
ID_SEQUENCER              716110000.0 ns  TB seq.                   Check data in Address 0x2 has not been modified
ID_POS_ACK                718110000.0 ns  TB seq.                   check_value() => OK, for slv x"5"'. '(Address 0x2) Read: 0x5   Expected: 0x5'


ID_LOG_HDR                719120000.0 ns  TB seq.                   STEP 4: Force errors in some addresses in one of the inward SRAMs and check output matches with
                                                                    previous written data in STEP 1.
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER              719120000.0 ns  TB seq.
ID_SEQUENCER              719120000.0 ns  TB seq.                   Forcing errors in some addresses in inward SRAM 0.
ID_SEQUENCER              719120000.0 ns  TB seq.                   -- SRAM 0 --
ID_SEQUENCER              719130000.0 ns  TB seq.                   Data word (0x3) injected in Address (0x0)
ID_SEQUENCER              719140000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x1)
ID_SEQUENCER              719150000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x2)
ID_SEQUENCER              719160000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x3)
ID_SEQUENCER              719170000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x4)
ID_SEQUENCER              719180000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x5)
ID_SEQUENCER              719190000.0 ns  TB seq.                   Data word (0x2) injected in Address (0x6)
ID_SEQUENCER              719200000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x7)
ID_SEQUENCER              719210000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x8)
ID_SEQUENCER              719220000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x9)
ID_SEQUENCER              719220000.0 ns  TB seq.
ID_SEQUENCER              719220000.0 ns  TB seq.                   Read values from every address of the SRAM and check 'o_rd_data' and 'o_data_val' outputs:
ID_POS_ACK                721220000.0 ns  TB seq.                   check_value() => OK, for slv x"7"'. '(Address: 0x0) Read: 0x7   Expected: 0x7'
ID_POS_ACK                721220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                724220000.0 ns  TB seq.                   check_value() => OK, for slv x"7"'. '(Address: 0x1) Read: 0x7   Expected: 0x7'
ID_POS_ACK                724220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                727220000.0 ns  TB seq.                   check_value() => OK, for slv x"5"'. '(Address: 0x2) Read: 0x5   Expected: 0x5'
ID_POS_ACK                727220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                730220000.0 ns  TB seq.                   check_value() => OK, for slv x"2"'. '(Address: 0x3) Read: 0x2   Expected: 0x2'
ID_POS_ACK                730220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                733220000.0 ns  TB seq.                   check_value() => OK, for slv x"0"'. '(Address: 0x4) Read: 0x0   Expected: 0x0'
ID_POS_ACK                733220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                736220000.0 ns  TB seq.                   check_value() => OK, for slv x"1"'. '(Address: 0x5) Read: 0x1   Expected: 0x1'
ID_POS_ACK                736220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                739220000.0 ns  TB seq.                   check_value() => OK, for slv x"5"'. '(Address: 0x6) Read: 0x5   Expected: 0x5'
ID_POS_ACK                739220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                742220000.0 ns  TB seq.                   check_value() => OK, for slv x"0"'. '(Address: 0x7) Read: 0x0   Expected: 0x0'
ID_POS_ACK                742220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                745220000.0 ns  TB seq.                   check_value() => OK, for slv x"1"'. '(Address: 0x8) Read: 0x1   Expected: 0x1'
ID_POS_ACK                745220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                748220000.0 ns  TB seq.                   check_value() => OK, for slv x"1"'. '(Address: 0x9) Read: 0x1   Expected: 0x1'
ID_POS_ACK                748220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                751220000.0 ns  TB seq.                   check_value() => OK, for slv x"5"'. '(Address: 0xA) Read: 0x5   Expected: 0x5'
ID_POS_ACK                751220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                754220000.0 ns  TB seq.                   check_value() => OK, for slv x"1"'. '(Address: 0xB) Read: 0x1   Expected: 0x1'
ID_POS_ACK                754220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                757220000.0 ns  TB seq.                   check_value() => OK, for slv x"7"'. '(Address: 0xC) Read: 0x7   Expected: 0x7'
ID_POS_ACK                757220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                760220000.0 ns  TB seq.                   check_value() => OK, for slv x"0"'. '(Address: 0xD) Read: 0x0   Expected: 0x0'
ID_POS_ACK                760220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                763220000.0 ns  TB seq.                   check_value() => OK, for slv x"2"'. '(Address: 0xE) Read: 0x2   Expected: 0x2'
ID_POS_ACK                763220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK                766220000.0 ns  TB seq.                   check_value() => OK, for slv x"2"'. '(Address: 0xF) Read: 0x2   Expected: 0x2'
ID_POS_ACK                766220000.0 ns  TB seq.                   check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              767220000.0 ns  TB seq.
ID_SEQUENCER              767220000.0 ns  TB seq.                   Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER              769220000.0 ns  TB seq.                   Writing value 0x7 in address 0x0.
ID_SEQUENCER              791220000.0 ns  TB seq.                   Writing value 0x7 in address 0x1.
ID_SEQUENCER              813220000.0 ns  TB seq.                   Writing value 0x5 in address 0x2.
ID_SEQUENCER              835220000.0 ns  TB seq.                   Writing value 0x2 in address 0x3.
ID_SEQUENCER              857220000.0 ns  TB seq.                   Writing value 0x0 in address 0x4.
ID_SEQUENCER              879220000.0 ns  TB seq.                   Writing value 0x1 in address 0x5.
ID_SEQUENCER              901220000.0 ns  TB seq.                   Writing value 0x5 in address 0x6.
ID_SEQUENCER              923220000.0 ns  TB seq.                   Writing value 0x0 in address 0x7.
ID_SEQUENCER              945220000.0 ns  TB seq.                   Writing value 0x1 in address 0x8.
ID_SEQUENCER              967220000.0 ns  TB seq.                   Writing value 0x1 in address 0x9.
ID_SEQUENCER              989220000.0 ns  TB seq.                   Writing value 0x5 in address 0xA.
ID_SEQUENCER             1011220000.0 ns  TB seq.                   Writing value 0x1 in address 0xB.
ID_SEQUENCER             1033220000.0 ns  TB seq.                   Writing value 0x7 in address 0xC.
ID_SEQUENCER             1055220000.0 ns  TB seq.                   Writing value 0x0 in address 0xD.
ID_SEQUENCER             1077220000.0 ns  TB seq.                   Writing value 0x2 in address 0xE.
ID_SEQUENCER             1099220000.0 ns  TB seq.                   Writing value 0x2 in address 0xF.
ID_SEQUENCER             1119220000.0 ns  TB seq.                   Forcing errors in some addresses in inward SRAM 1.
ID_SEQUENCER             1119220000.0 ns  TB seq.                   -- SRAM 1 --
ID_SEQUENCER             1119230000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x0)
ID_SEQUENCER             1119240000.0 ns  TB seq.                   Data word (0x1) injected in Address (0x1)
ID_SEQUENCER             1119250000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x2)
ID_SEQUENCER             1119260000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x3)
ID_SEQUENCER             1119270000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x4)
ID_SEQUENCER             1119280000.0 ns  TB seq.                   Data word (0x1) injected in Address (0x5)
ID_SEQUENCER             1119290000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x6)
ID_SEQUENCER             1119300000.0 ns  TB seq.                   Data word (0x4) injected in Address (0x7)
ID_SEQUENCER             1119310000.0 ns  TB seq.                   Data word (0x0) injected in Address (0x8)
ID_SEQUENCER             1119320000.0 ns  TB seq.                   Data word (0x1) injected in Address (0x9)
ID_SEQUENCER             1119320000.0 ns  TB seq.
```

```
ID_SEQUENCER          1119320000.0 ns  TB seq.
ID_SEQUENCER          1119320000.0 ns  TB seq.                    Read from some addresses with and without errors injected and check 'o_rd_data' and
                                                                  'o_data_val' outputs:
ID_POS_ACK            1121320000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0x0) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1121320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1124320000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0x1) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1124320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1127320000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0x2) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1127320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1130320000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0x3) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1130320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1133320000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0x4) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1133320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1136320000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x5) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1136320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1139320000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0x6) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1139320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1142320000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0x7) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1142320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1145320000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x8) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1145320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1148320000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x9) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1148320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1151320000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0xA) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1151320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1154320000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0xB) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1154320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1157320000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0xC) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1157320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1160320000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0xD) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1160320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1163320000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0xE) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1163320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1166320000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0xF) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1166320000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          1167320000.0 ns  TB seq.
ID_SEQUENCER          1167320000.0 ns  TB seq.
ID_SEQUENCER          1167320000.0 ns  TB seq.                    Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER          1169320000.0 ns  TB seq.                    Writing value 0x7 in address 0x0.
ID_SEQUENCER          1191320000.0 ns  TB seq.                    Writing value 0x7 in address 0x1.
ID_SEQUENCER          1213320000.0 ns  TB seq.                    Writing value 0x5 in address 0x2.
ID_SEQUENCER          1235320000.0 ns  TB seq.                    Writing value 0x2 in address 0x3.
ID_SEQUENCER          1257320000.0 ns  TB seq.                    Writing value 0x0 in address 0x4.
ID_SEQUENCER          1279320000.0 ns  TB seq.                    Writing value 0x1 in address 0x5.
ID_SEQUENCER          1301320000.0 ns  TB seq.                    Writing value 0x5 in address 0x6.
ID_SEQUENCER          1323320000.0 ns  TB seq.                    Writing value 0x0 in address 0x7.
ID_SEQUENCER          1345320000.0 ns  TB seq.                    Writing value 0x1 in address 0x8.
ID_SEQUENCER          1367320000.0 ns  TB seq.                    Writing value 0x1 in address 0x9.
ID_SEQUENCER          1389320000.0 ns  TB seq.                    Writing value 0x5 in address 0xA.
ID_SEQUENCER          1411320000.0 ns  TB seq.                    Writing value 0x1 in address 0xB.
ID_SEQUENCER          1433320000.0 ns  TB seq.                    Writing value 0x7 in address 0xC.
ID_SEQUENCER          1455320000.0 ns  TB seq.                    Writing value 0x0 in address 0xD.
ID_SEQUENCER          1477320000.0 ns  TB seq.                    Writing value 0x2 in address 0xE.
ID_SEQUENCER          1499320000.0 ns  TB seq.                    Writing value 0x2 in address 0xF.
ID_SEQUENCER          1519320000.0 ns  TB seq.                    Forcing errors in some addresses in inward SRAM 2.
ID_SEQUENCER          1519320000.0 ns  TB seq.                    -- SRAM 2 --
ID_SEQUENCER          1519330000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x0)
ID_SEQUENCER          1519340000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x1)
ID_SEQUENCER          1519350000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x2)
ID_SEQUENCER          1519360000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x3)
ID_SEQUENCER          1519370000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x4)
ID_SEQUENCER          1519380000.0 ns  TB seq.                    Data word (0x4) injected in Address (0x5)
ID_SEQUENCER          1519390000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x6)
ID_SEQUENCER          1519400000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x7)
ID_SEQUENCER          1519410000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x8)
ID_SEQUENCER          1519420000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x9)
ID_SEQUENCER          1519420000.0 ns  TB seq.
ID_SEQUENCER          1519420000.0 ns  TB seq.
ID_SEQUENCER          1519420000.0 ns  TB seq.                    Read from some addresses with and without errors injected and check 'o_rd_data' and
                                                                  'o_data_val' outputs:
ID_POS_ACK            1521420000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0x0) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1521420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1524420000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0x1) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1524420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1527420000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0x2) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1527420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1530420000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0x3) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1530420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1533420000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0x4) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1533420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1536420000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x5) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1536420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1539420000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0x6) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1539420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1542420000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0x7) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1542420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1545420000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x8) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1545420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1548420000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0x9) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1548420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1551420000.0 ns  TB seq.                    check_value() => OK, for slv x"5"'. '(Address: 0xA) Read: 0x5   Expected: 0x5'
ID_POS_ACK            1551420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1554420000.0 ns  TB seq.                    check_value() => OK, for slv x"1"'. '(Address: 0xB) Read: 0x1   Expected: 0x1'
ID_POS_ACK            1554420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1557420000.0 ns  TB seq.                    check_value() => OK, for slv x"7"'. '(Address: 0xC) Read: 0x7   Expected: 0x7'
ID_POS_ACK            1557420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1560420000.0 ns  TB seq.                    check_value() => OK, for slv x"0"'. '(Address: 0xD) Read: 0x0   Expected: 0x0'
ID_POS_ACK            1560420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1563420000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0xE) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1563420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_POS_ACK            1566420000.0 ns  TB seq.                    check_value() => OK, for slv x"2"'. '(Address: 0xF) Read: 0x2   Expected: 0x2'
ID_POS_ACK            1566420000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
```

```
ID_SEQUENCER            1567420000.0 ns  TB seq.
ID_SEQUENCER            1567420000.0 ns  TB seq.                    Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER            1567420000.0 ns  TB seq.                    Writing value 0x2 in address 0x0.
ID_SEQUENCER            1569420000.0 ns  TB seq.                    Writing value 0x7 in address 0x0.
ID_SEQUENCER            1589420000.0 ns  TB seq.                    Writing value 0x7 in address 0x1.
ID_SEQUENCER            1591420000.0 ns  TB seq.                    Writing value 0x7 in address 0x1.
ID_SEQUENCER            1611420000.0 ns  TB seq.                    Writing value 0x7 in address 0x2.
ID_SEQUENCER            1613420000.0 ns  TB seq.                    Writing value 0x5 in address 0x2.
ID_SEQUENCER            1633420000.0 ns  TB seq.                    Writing value 0x5 in address 0x3.
ID_SEQUENCER            1635420000.0 ns  TB seq.                    Writing value 0x2 in address 0x3.
ID_SEQUENCER            1655420000.0 ns  TB seq.                    Writing value 0x2 in address 0x4.
ID_SEQUENCER            1657420000.0 ns  TB seq.                    Writing value 0x0 in address 0x4.
ID_SEQUENCER            1677420000.0 ns  TB seq.                    Writing value 0x0 in address 0x5.
ID_SEQUENCER            1679420000.0 ns  TB seq.                    Writing value 0x1 in address 0x5.
ID_SEQUENCER            1699420000.0 ns  TB seq.                    Writing value 0x1 in address 0x6.
ID_SEQUENCER            1701420000.0 ns  TB seq.                    Writing value 0x5 in address 0x6.
ID_SEQUENCER            1721420000.0 ns  TB seq.                    Writing value 0x5 in address 0x7.
ID_SEQUENCER            1723420000.0 ns  TB seq.                    Writing value 0x0 in address 0x7.
ID_SEQUENCER            1743420000.0 ns  TB seq.                    Writing value 0x0 in address 0x8.
ID_SEQUENCER            1745420000.0 ns  TB seq.                    Writing value 0x1 in address 0x8.
ID_SEQUENCER            1765420000.0 ns  TB seq.                    Writing value 0x1 in address 0x9.
ID_SEQUENCER            1767420000.0 ns  TB seq.                    Writing value 0x1 in address 0x9.
ID_SEQUENCER            1787420000.0 ns  TB seq.                    Writing value 0x1 in address 0xA.
ID_SEQUENCER            1789420000.0 ns  TB seq.                    Writing value 0x5 in address 0xA.
ID_SEQUENCER            1809420000.0 ns  TB seq.                    Writing value 0x5 in address 0xB.
ID_SEQUENCER            1811420000.0 ns  TB seq.                    Writing value 0x1 in address 0xB.
ID_SEQUENCER            1831420000.0 ns  TB seq.                    Writing value 0x1 in address 0xC.
ID_SEQUENCER            1833420000.0 ns  TB seq.                    Writing value 0x7 in address 0xC.
ID_SEQUENCER            1853420000.0 ns  TB seq.                    Writing value 0x7 in address 0xD.
ID_SEQUENCER            1855420000.0 ns  TB seq.                    Writing value 0x0 in address 0xD.
ID_SEQUENCER            1875420000.0 ns  TB seq.                    Writing value 0x0 in address 0xE.
ID_SEQUENCER            1877420000.0 ns  TB seq.                    Writing value 0x2 in address 0xE.
ID_SEQUENCER            1897420000.0 ns  TB seq.                    Writing value 0x2 in address 0xF.
ID_SEQUENCER            1899420000.0 ns  TB seq.                    Writing value 0x2 in address 0xF.


ID_LOG_HDR              1919420000.0 ns  TB seq.                    STEP 5: Force errors in some addresses in two of the inward SRAMs and check if output data is
                                                                    deprecated.
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID_SEQUENCER            1919420000.0 ns  TB seq.
ID_SEQUENCER            1919420000.0 ns  TB seq.                    Forcing errors in some addresses in inward SRAM 0 and 1.
ID_SEQUENCER            1919420000.0 ns  TB seq.                    -- SRAM 0 --
ID_SEQUENCER            1919430000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x0)
ID_SEQUENCER            1919440000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x1)
ID_SEQUENCER            1919450000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x2)
ID_SEQUENCER            1919460000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x3)
ID_SEQUENCER            1919470000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x4)
ID_SEQUENCER            1919480000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x5)
ID_SEQUENCER            1919490000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x6)
ID_SEQUENCER            1919500000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x7)
ID_SEQUENCER            1919510000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x8)
ID_SEQUENCER            1919520000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x9)
ID_SEQUENCER            1919530000.0 ns  TB seq.                    Data word (0x1) injected in Address (0xA)
ID_SEQUENCER            1919530000.0 ns  TB seq.                    -- SRAM 1 --
ID_SEQUENCER            1919540000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x0)
ID_SEQUENCER            1919550000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x1)
ID_SEQUENCER            1919560000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x2)
ID_SEQUENCER            1919570000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x3)
ID_SEQUENCER            1919580000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x4)
ID_SEQUENCER            1919590000.0 ns  TB seq.                    Data word (0x4) injected in Address (0x5)
ID_SEQUENCER            1919600000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x6)
ID_SEQUENCER            1919610000.0 ns  TB seq.                    Data word (0x4) injected in Address (0x7)
ID_SEQUENCER            1919620000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x8)
ID_SEQUENCER            1919630000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x9)
ID_SEQUENCER            1919640000.0 ns  TB seq.                    Data word (0x4) injected in Address (0xA)
ID_SEQUENCER            1919640000.0 ns  TB seq.
ID_SEQUENCER            1919640000.0 ns  TB seq.
ID_SEQUENCER            1919640000.0 ns  TB seq.                    Read from some addresses with and without errors injected and check 'o_data_val' output:
ID_SEQUENCER            1921640000.0 ns  TB seq.                    (Address: 0x0) Read: 0x1   Expected: 0x7
ID_POS_ACK              1921640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1924640000.0 ns  TB seq.                    (Address: 0x1) Read: 0x1   Expected: 0x7
ID_POS_ACK              1924640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1927640000.0 ns  TB seq.                    (Address: 0x2) Read: 0x2   Expected: 0x5
ID_POS_ACK              1927640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1930640000.0 ns  TB seq.                    (Address: 0x3) Read: 0x2   Expected: 0x2
ID_POS_ACK              1930640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1933640000.0 ns  TB seq.                    (Address: 0x4) Read: 0x1   Expected: 0x0
ID_POS_ACK              1933640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1936640000.0 ns  TB seq.                    (Address: 0x5) Read: 0x1   Expected: 0x1
ID_POS_ACK              1936640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1939640000.0 ns  TB seq.                    (Address: 0x6) Read: 0x1   Expected: 0x5
ID_POS_ACK              1939640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1942640000.0 ns  TB seq.                    (Address: 0x7) Read: 0x3   Expected: 0x0
ID_POS_ACK              1942640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1945640000.0 ns  TB seq.                    (Address: 0x8) Read: 0x3   Expected: 0x1
ID_POS_ACK              1945640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1948640000.0 ns  TB seq.                    (Address: 0x9) Read: 0x3   Expected: 0x1
ID_POS_ACK              1948640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1951640000.0 ns  TB seq.                    (Address: 0xA) Read: 0x1   Expected: 0x5
ID_POS_ACK              1951640000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER            1954640000.0 ns  TB seq.                    (Address: 0xB) Read: 0x1   Expected: 0x1
ID_POS_ACK              1954640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1957640000.0 ns  TB seq.                    (Address: 0xC) Read: 0x7   Expected: 0x7
ID_POS_ACK              1957640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1960640000.0 ns  TB seq.                    (Address: 0xD) Read: 0x0   Expected: 0x0
ID_POS_ACK              1960640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1963640000.0 ns  TB seq.                    (Address: 0xE) Read: 0x2   Expected: 0x2
ID_POS_ACK              1963640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1966640000.0 ns  TB seq.                    (Address: 0xF) Read: 0x2   Expected: 0x2
ID_POS_ACK              1966640000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER            1967650000.0 ns  TB seq.
```

```
ID_SEQUENCER          1967650000.0 ns  TB seq.                  Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER          1969650000.0 ns  TB seq.                  Writing value 0x7 in address 0x0.
ID_SEQUENCER          1991650000.0 ns  TB seq.                  Writing value 0x7 in address 0x1.
ID_SEQUENCER          2013650000.0 ns  TB seq.                  Writing value 0x5 in address 0x2.
ID_SEQUENCER          2035650000.0 ns  TB seq.                  Writing value 0x2 in address 0x3.
ID_SEQUENCER          2057650000.0 ns  TB seq.                  Writing value 0x0 in address 0x4.
ID_SEQUENCER          2079650000.0 ns  TB seq.                  Writing value 0x1 in address 0x5.
ID_SEQUENCER          2101650000.0 ns  TB seq.                  Writing value 0x5 in address 0x6.
ID_SEQUENCER          2123650000.0 ns  TB seq.                  Writing value 0x0 in address 0x7.
ID_SEQUENCER          2145650000.0 ns  TB seq.                  Writing value 0x1 in address 0x8.
ID_SEQUENCER          2167650000.0 ns  TB seq.                  Writing value 0x1 in address 0x9.
ID_SEQUENCER          2189650000.0 ns  TB seq.                  Writing value 0x5 in address 0xA.
ID_SEQUENCER          2211650000.0 ns  TB seq.                  Writing value 0x1 in address 0xB.
ID_SEQUENCER          2233650000.0 ns  TB seq.                  Writing value 0x7 in address 0xC.
ID_SEQUENCER          2255650000.0 ns  TB seq.                  Writing value 0x0 in address 0xD.
ID_SEQUENCER          2277650000.0 ns  TB seq.                  Writing value 0x2 in address 0xE.
ID_SEQUENCER          2299650000.0 ns  TB seq.                  Writing value 0x2 in address 0xF.
ID_SEQUENCER          2319650000.0 ns  TB seq.
ID_SEQUENCER          2319650000.0 ns  TB seq.                  Forcing errors in some addresses in inward SRAM 0 and 2.
ID_SEQUENCER          2319650000.0 ns  TB seq.                  -- SRAM 0 --
ID_SEQUENCER          2319660000.0 ns  TB seq.                  Data word (0x3) injected in Address (0x0)
ID_SEQUENCER          2319670000.0 ns  TB seq.                  Data word (0x2) injected in Address (0x1)
ID_SEQUENCER          2319680000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x2)
ID_SEQUENCER          2319690000.0 ns  TB seq.                  Data word (0x4) injected in Address (0x3)
ID_SEQUENCER          2319700000.0 ns  TB seq.                  Data word (0x3) injected in Address (0x4)
ID_SEQUENCER          2319710000.0 ns  TB seq.                  Data word (0x2) injected in Address (0x5)
ID_SEQUENCER          2319720000.0 ns  TB seq.                  Data word (0x0) injected in Address (0x6)
ID_SEQUENCER          2319730000.0 ns  TB seq.                  Data word (0x3) injected in Address (0x7)
ID_SEQUENCER          2319740000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x8)
ID_SEQUENCER          2319750000.0 ns  TB seq.                  Data word (0x4) injected in Address (0x9)
ID_SEQUENCER          2319760000.0 ns  TB seq.                  Data word (0x2) injected in Address (0xA)
ID_SEQUENCER          2319760000.0 ns  TB seq.                  -- SRAM 2 --
ID_SEQUENCER          2319770000.0 ns  TB seq.                  Data word (0x0) injected in Address (0x0)
ID_SEQUENCER          2319780000.0 ns  TB seq.                  Data word (0x3) injected in Address (0x1)
ID_SEQUENCER          2319790000.0 ns  TB seq.                  Data word (0x0) injected in Address (0x2)
ID_SEQUENCER          2319800000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x3)
ID_SEQUENCER          2319810000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x4)
ID_SEQUENCER          2319820000.0 ns  TB seq.                  Data word (0x3) injected in Address (0x5)
ID_SEQUENCER          2319830000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x6)
ID_SEQUENCER          2319840000.0 ns  TB seq.                  Data word (0x1) injected in Address (0x7)
ID_SEQUENCER          2319850000.0 ns  TB seq.                  Data word (0x0) injected in Address (0x8)
ID_SEQUENCER          2319860000.0 ns  TB seq.                  Data word (0x0) injected in Address (0x9)
ID_SEQUENCER          2319870000.0 ns  TB seq.                  Data word (0x2) injected in Address (0xA)
ID_SEQUENCER          2319870000.0 ns  TB seq.
ID_SEQUENCER          2319870000.0 ns  TB seq.                  Read from some addresses with and without errors injected and check 'o_data_val' output:
ID_SEQUENCER          2321870000.0 ns  TB seq.                  (Address: 0x0) Read: 0x3   Expected: 0x7
ID_POS_ACK            2321870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2324870000.0 ns  TB seq.                  (Address: 0x1) Read: 0x2   Expected: 0x7
ID_POS_ACK            2324870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2327870000.0 ns  TB seq.                  (Address: 0x2) Read: 0x1   Expected: 0x5
ID_POS_ACK            2327870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2330870000.0 ns  TB seq.                  (Address: 0x3) Read: 0x4   Expected: 0x2
ID_POS_ACK            2330870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2333870000.0 ns  TB seq.                  (Address: 0x4) Read: 0x3   Expected: 0x0
ID_POS_ACK            2333870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2336870000.0 ns  TB seq.                  (Address: 0x5) Read: 0x2   Expected: 0x1
ID_POS_ACK            2336870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2339870000.0 ns  TB seq.                  (Address: 0x6) Read: 0x0   Expected: 0x5
ID_POS_ACK            2339870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2342870000.0 ns  TB seq.                  (Address: 0x7) Read: 0x3   Expected: 0x0
ID_POS_ACK            2342870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2345870000.0 ns  TB seq.                  (Address: 0x8) Read: 0x1   Expected: 0x1
ID_POS_ACK            2345870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2348870000.0 ns  TB seq.                  (Address: 0x9) Read: 0x4   Expected: 0x1
ID_POS_ACK            2348870000.0 ns  TB seq.                  check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER          2351870000.0 ns  TB seq.                  (Address: 0xA) Read: 0x2   Expected: 0x5
ID_POS_ACK            2351870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2354870000.0 ns  TB seq.                  (Address: 0xB) Read: 0x1   Expected: 0x1
ID_POS_ACK            2354870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2357870000.0 ns  TB seq.                  (Address: 0xC) Read: 0x7   Expected: 0x7
ID_POS_ACK            2357870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2360870000.0 ns  TB seq.                  (Address: 0xD) Read: 0x0   Expected: 0x0
ID_POS_ACK            2360870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2363870000.0 ns  TB seq.                  (Address: 0xE) Read: 0x2   Expected: 0x2
ID_POS_ACK            2363870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2366870000.0 ns  TB seq.                  (Address: 0xF) Read: 0x2   Expected: 0x2
ID_POS_ACK            2366870000.0 ns  TB seq.                  check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER          2367880000.0 ns  TB seq.
ID_SEQUENCER          2367880000.0 ns  TB seq.
ID_SEQUENCER          2367880000.0 ns  TB seq.                  Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER          2368880000.0 ns  TB seq.                  Writing value 0x7 in address 0x0.
ID_SEQUENCER          2389880000.0 ns  TB seq.                  Writing value 0x7 in address 0x1.
ID_SEQUENCER          2410880000.0 ns  TB seq.                  Writing value 0x5 in address 0x2.
ID_SEQUENCER          2431880000.0 ns  TB seq.                  Writing value 0x2 in address 0x3.
ID_SEQUENCER          2452880000.0 ns  TB seq.                  Writing value 0x0 in address 0x4.
ID_SEQUENCER          2473880000.0 ns  TB seq.                  Writing value 0x1 in address 0x5.
ID_SEQUENCER          2494880000.0 ns  TB seq.                  Writing value 0x5 in address 0x6.
ID_SEQUENCER          2515880000.0 ns  TB seq.                  Writing value 0x0 in address 0x7.
ID_SEQUENCER          2536880000.0 ns  TB seq.                  Writing value 0x1 in address 0x8.
ID_SEQUENCER          2557880000.0 ns  TB seq.                  Writing value 0x1 in address 0x9.
ID_SEQUENCER          2578880000.0 ns  TB seq.                  Writing value 0x5 in address 0xA.
ID_SEQUENCER          2599880000.0 ns  TB seq.                  Writing value 0x1 in address 0xB.
ID_SEQUENCER          2620880000.0 ns  TB seq.                  Writing value 0x7 in address 0xC.
ID_SEQUENCER          2641880000.0 ns  TB seq.                  Writing value 0x0 in address 0xD.
ID_SEQUENCER          2662880000.0 ns  TB seq.                  Writing value 0x2 in address 0xE.
ID_SEQUENCER          2683880000.0 ns  TB seq.                  Writing value 0x2 in address 0xF.
ID_SEQUENCER          2703880000.0 ns  TB seq.
```

```
ID_SEQUENCER              2703880000.0 ns  TB seq.                    Forcing errors in some addresses in inward SRAM 1 and 2.
ID_SEQUENCER              2703880000.0 ns  TB seq.                    -- SRAM 0 --
ID_SEQUENCER              2703890000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x0)
ID_SEQUENCER              2703900000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x1)
ID_SEQUENCER              2703910000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x2)
ID_SEQUENCER              2703920000.0 ns  TB seq.                    Data word (0x4) injected in Address (0x3)
ID_SEQUENCER              2703930000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x4)
ID_SEQUENCER              2703940000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x5)
ID_SEQUENCER              2703950000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x6)
ID_SEQUENCER              2703960000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x7)
ID_SEQUENCER              2703970000.0 ns  TB seq.                    Data word (0x3) injected in Address (0x8)
ID_SEQUENCER              2703980000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x9)
ID_SEQUENCER              2703990000.0 ns  TB seq.                    Data word (0x4) injected in Address (0xA)
ID_SEQUENCER              2703990000.0 ns  TB seq.                    -- SRAM 1 --
ID_SEQUENCER              2704000000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x0)
ID_SEQUENCER              2704010000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x1)
ID_SEQUENCER              2704020000.0 ns  TB seq.                    Data word (0x2) injected in Address (0x2)
ID_SEQUENCER              2704030000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x3)
ID_SEQUENCER              2704040000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x4)
ID_SEQUENCER              2704050000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x5)
ID_SEQUENCER              2704060000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x6)
ID_SEQUENCER              2704070000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x7)
ID_SEQUENCER              2704080000.0 ns  TB seq.                    Data word (0x0) injected in Address (0x8)
ID_SEQUENCER              2704090000.0 ns  TB seq.                    Data word (0x1) injected in Address (0x9)
ID_SEQUENCER              2704100000.0 ns  TB seq.                    Data word (0x3) injected in Address (0xA)
ID_SEQUENCER              2704100000.0 ns  TB seq.
ID_SEQUENCER              2704100000.0 ns  TB seq.                    Read from some addresses with and without errors injected and check 'o_data_val' output:
ID_SEQUENCER              2706100000.0 ns  TB seq.                    (Address: 0x0) Read: 0x2   Expected: 0x7
ID_POS_ACK                2706100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2709100000.0 ns  TB seq.                    (Address: 0x1) Read: 0x3   Expected: 0x7
ID_POS_ACK                2709100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2712100000.0 ns  TB seq.                    (Address: 0x2) Read: 0x1   Expected: 0x5
ID_POS_ACK                2712100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2715100000.0 ns  TB seq.                    (Address: 0x3) Read: 0x4   Expected: 0x2
ID_POS_ACK                2715100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2718100000.0 ns  TB seq.                    (Address: 0x4) Read: 0x0   Expected: 0x0
ID_POS_ACK                2718100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2721100000.0 ns  TB seq.                    (Address: 0x5) Read: 0x2   Expected: 0x1
ID_POS_ACK                2721100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2724100000.0 ns  TB seq.                    (Address: 0x6) Read: 0x2   Expected: 0x5
ID_POS_ACK                2724100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2727100000.0 ns  TB seq.                    (Address: 0x7) Read: 0x2   Expected: 0x0
ID_POS_ACK                2727100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2730100000.0 ns  TB seq.                    (Address: 0x8) Read: 0x3   Expected: 0x1
ID_POS_ACK                2730100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2733100000.0 ns  TB seq.                    (Address: 0x9) Read: 0x1   Expected: 0x1
ID_POS_ACK                2733100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2736100000.0 ns  TB seq.                    (Address: 0xA) Read: 0x4   Expected: 0x5
ID_POS_ACK                2736100000.0 ns  TB seq.                    check_value() => OK, for std_logic '0' (exp: '0'). 'Data not valid (deprecated)'
ID_SEQUENCER              2739100000.0 ns  TB seq.                    (Address: 0xB) Read: 0x1   Expected: 0x1
ID_POS_ACK                2739100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2742100000.0 ns  TB seq.                    (Address: 0xC) Read: 0x7   Expected: 0x7
ID_POS_ACK                2742100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2745100000.0 ns  TB seq.                    (Address: 0xD) Read: 0x0   Expected: 0x0
ID_POS_ACK                2745100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2748100000.0 ns  TB seq.                    (Address: 0xE) Read: 0x2   Expected: 0x2
ID_POS_ACK                2748100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2751100000.0 ns  TB seq.                    (Address: 0xF) Read: 0x2   Expected: 0x2
ID_POS_ACK                2751100000.0 ns  TB seq.                    check_value() => OK, for std_logic '1' (exp: '1'). 'Data valid'
ID_SEQUENCER              2752110000.0 ns  TB seq.
ID_SEQUENCER              2752110000.0 ns  TB seq.
ID_SEQUENCER              2752110000.0 ns  TB seq.                    Rewrite values in SRAM to overwrite errors injected.
ID_SEQUENCER              2754110000.0 ns  TB seq.                    Writing value 0x7 in address 0x0.
ID_SEQUENCER              2776110000.0 ns  TB seq.                    Writing value 0x7 in address 0x1.
ID_SEQUENCER              2798110000.0 ns  TB seq.                    Writing value 0x5 in address 0x2.
ID_SEQUENCER              2820110000.0 ns  TB seq.                    Writing value 0x2 in address 0x3.
ID_SEQUENCER              2842110000.0 ns  TB seq.                    Writing value 0x0 in address 0x4.
ID_SEQUENCER              2864110000.0 ns  TB seq.                    Writing value 0x1 in address 0x5.
ID_SEQUENCER              2886110000.0 ns  TB seq.                    Writing value 0x5 in address 0x6.
ID_SEQUENCER              2908110000.0 ns  TB seq.                    Writing value 0x0 in address 0x7.
ID_SEQUENCER              2930110000.0 ns  TB seq.                    Writing value 0x1 in address 0x8.
ID_SEQUENCER              2952110000.0 ns  TB seq.                    Writing value 0x1 in address 0x9.
ID_SEQUENCER              2974110000.0 ns  TB seq.                    Writing value 0x5 in address 0xA.
ID_SEQUENCER              2996110000.0 ns  TB seq.                    Writing value 0x1 in address 0xB.
ID_SEQUENCER              3018110000.0 ns  TB seq.                    Writing value 0x7 in address 0xC.
ID_SEQUENCER              3040110000.0 ns  TB seq.                    Writing value 0x0 in address 0xD.
ID_SEQUENCER              3062110000.0 ns  TB seq.                    Writing value 0x2 in address 0xE.
ID_SEQUENCER              3084110000.0 ns  TB seq.                    Writing value 0x2 in address 0xF.

    ================================================================================================================================================================
    *** FINAL SUMMARY OF ALL ALERTS ***
    ================================================================================================================================================================
                            REGARDED   EXPECTED  IGNORED      Comment?
            NOTE         :      0         0         0        ok
            TB_NOTE      :      0         0         0        ok
            WARNING      :      0         0         0        ok
            TB_WARNING   :      0         0         0        ok
            MANUAL_CHECK :      0         0         0        ok
            ERROR        :      0         0         0        ok
            TB_ERROR     :      0         0         0        ok
            FAILURE      :      0         0         0        ok
            TB_FAILURE   :      0         0         0        ok
    ================================================================================================================================================================
    >> Simulation SUCCESS: No mismatch between counted and expected serious alerts
    ================================================================================================================================================================



ID_LOG_HDR                3104210000.0 ns  TB seq.(uvvm)              SIMULATION COMPLETED
----------------------------------------------------------------------------------------------------------------------------------------------------------------
```