**Universidad Internacional de La Rioja**
**Máster universitario en Seguridad Informática**

# Implementación basada en co-simulación HW de un algoritmo criptográfico en una FPGA.

**Trabajo Fin de Máster**

**presentado por:** Fernández Herrera, David Guillermo

**Director/a:** Botella, Guillermo

Ciudad: Madrid

Fecha: 25 de Julio de 2014

# Resumen

Debido a la naturaleza de gran multitud de algoritmos criptográficos de realizar tareas computacionalmente intensivas y que posibilitan un alto grado de paralelismo en sus operaciones, la utilización de FPGAs para su implementación puede ser una solución óptima.

El tiempo dedicado a la verificación de los diseños en FPGA es uno de los grandes obstáculos en los proyectos que utilizan estos dispositivos, debido principalmente a la lentitud de las simulaciones realizadas en PC y por la dificultad que implica realizar un banco de pruebas complejo en un lenguaje de descripción hardware. Para solucionar estos problemas, el presente Trabajo Fin de Máster describe cómo desarrollar una sencilla aplicación genérica para gestionar la entrada/salida de datos desde el entorno Simulink y transmitirlos de forma eficiente a un algoritmo que se esté ejecutando en una FPGA mediante co-simulación Hardware. Para tal propósito, se hace uso de la herramienta System Generator de Xilinx.

**Palabras Clave:** Co-Simulación, Criptografía, System Generator, FPGA, Hardware

# Abstract

Due to the nature of a great multitude of cryptographic algorithms to perform computationally intensive tasks and exhibit high degree of parallelism, the use of FPGAs for its implementation can be an optimal solution.

The time dedicated to the verification of FPGA designs is one of the major obstacles in projects utilizing this kind of devices. It is mainly due to the slowness of the simulations running on PC and the difficulty of making complex test benches using hardware description languages. To solve these problems, this work describes how to develop a simple generic application using Simulink for managing input and output data of an algorithm running on FPGA and how to transmit data between FPGA and PC at high speed using Hardware Co-Simulation. System Generator tool is used to achieve these goals.

**Keywords:** co-simulation, cryptography, System Generator, FPGA, Hardware

# Contenido

# 1. Introducción

En la actualidad existen potentes dispositivos para desarrollar tareas computacionalmente intensivas, como pueden ser las propias de algunos algoritmos criptográficos. Dichos dispositivos utilizados como co-procesadores liberarán a los procesadores principales de la carga computacional asociada a dichas tareas.

Se propone la implementación de varios algoritmos criptográficos en una FPGA (Field Programmable Gate Array) al considerarse, tras realizar un análisis de distintas tecnologías disponibles, el dispositivo más adecuado. Existen numerosos artículos académicos sobre implementaciones de este tipo, pero el principal objetivo del presente trabajo no es la implementación en sí del algoritmo en una FPGA, sino la presentación de una arquitectura que facilite la verificación de los diseños realizados, acortando los tiempos de desarrollo de los bancos de pruebas y acelerando la realización de dichas pruebas.

Esta arquitectura se basa en el uso de herramientas de Mathworks y Xilinx trabajando conjuntamente para permitir modelar sistemas destinados a ser ejecutados en FPGA.

En el presente Trabajo Fin de Máster se desarrollará una aplicación que ejecute en FPGA un algoritmo criptográfico y en la que se intercambien datos entre este dispositivo y el PC eficientemente. Para ello, se empleará una tarjeta de evaluación (ML605 de Xilinx) conectada al PC a través de conexión Gigabit Ethernet permitiendo de esta forma, alcanzar una elevada tasa de transferencia de datos.

A este proceso, se le denomina co-simulación Hardware. El algoritmo criptográfico se tratará dentro de la aplicación como una caja negra (Black Box), de forma que pueda ser reemplazado por cualquier otro algoritmo sin alterar la estructura del Framework desarrollado.

Se comenzará describiendo el contexto de la aplicación indicando las referencias empleadas o desestimadas junto a las razones que llevaron a tales decisiones. Se definirán unos objetivos concretos y se describirá la metodología seguida en el desarrollo de la aplicación para posibilitar alcanzarlos.

Para comprender el funcionamiento de la aplicación desarrollada resulta de vital importancia realizar una descripción de las herramientas involucradas en su desarrollo, ya que éstas posibilitan la automatización de gran número de las acciones requeridas para la configuración

del dispositivo. Mathworks aporta Matlab y Simulink para simplificar el proceso de generación y análisis de datos. Por otro lado, estas dos herramientas proporcionan una enorme potencia matemática para la gestión de dichos datos. Simulink es un producto que se ejecuta sobre Matlab, que proporciona un entorno de interactivo y gráfico para el modelado, la simulación y análisis de sistemas dinámicos.

System Generator de Xilinx permite al entorno gráfico Simulink utilizar bloques que pueden ser sintetizados directamente en una FPGA facilitando su implementación. Entre una de sus principales funciones destaca el uso de “black boxes”, lo que permite importar componentes desarrollados en lenguajes de descripción hardware a Simulink, de forma que puedan tratarse como cualquier otro bloque dentro del interfaz gráfico.

Los algoritmos criptográficos que se decidan implementar deberán ser adaptados para permitir su integración en la aplicación sin que se requiera modificar la gestión de los datos de entrada/salida. La aplicación será un modelo de Simulink en el que los distintos algoritmos a verificar se tratarán como cajas negras, de manera que para utilizar el mismo modelo para distintos algoritmos baste con intercambiar los bloques que los representan. Antes de generar dichos bloques se realizarán una serie de simulaciones sobre el código fuente para verificar su funcionamiento. De estas pruebas se obtendrán unos tiempos de simulación que permitirán comparar este tipo de verificaciones con la propuesta en este Trabajo.

Para lograr el objetivo principal de este Trabajo se han ido alcanzando una serie de hitos que serán detallados en profundidad en otros apartados. En el primer Hito se desarrollará una sencilla aplicación que se ejecute íntegramente en PC haciendo uso de los componentes tipo “black box”. Su objetivo será la verificación del proceso de generación de los componentes. En el segundo Hito, sobre la aplicación anterior se realizarán una serie de modificaciones que permitan la separación de la ejecución en dos dominios, PC y FPGA. Las modificaciones realizadas se verificarán en PC para simplificar la depuración de los posibles problemas que surjan. En el tercer Hito se realizará la co-simulación HW, ejecutando el algoritmo en una tarjeta de evaluación basada en FPGA y gestionando los datos desde el PC.

Durante la consecución de los distintos hitos se obtendrán los resultados de los que se derivarán las conclusiones finales.

# 2. Marco Teórico

El presente apartado abarca el estudio de la literatura existente, donde se analiza y describe el contexto asociado a la línea de trabajo escogida. Se comentarán las referencias empleadas y las causas por las que se ha decidido emplearlas o descartarlas.

En el planteamiento inicial de este Trabajo se proponía la implementación de varios algoritmos criptográficos en una FPGA usando un lenguaje de descripción hardware. Se comenzó analizando el estado del arte en cuanto a implementaciones de este tipo, y se observó que existían multitud de publicaciones al respecto. Algunas de ellas resultan particularmente muy interesantes como en el trabajo de Song, Kawakami, Nakano e Ito (2010) [1], en el que se plantea una implementación para el algoritmo RSA (2048bits) mediante un único DSP Slice (hardware específico que incorporan las FPGAS destinado a hacer operaciones DSP, compuesto normalmente por varios multiplicadores, sumadores y registros) y un Block Ram (bloque de memoria interna de la FPGA) para la familia de FPGAs Virtex-6 de Xilinx. Kshirsagar y Vyawahe (2012) [2] proponen una implementación con gran rendimiento y una importante reducción de la lógica empleada enfocándose en un dispositivo Spartan3 de Xilinx. En el trabajo de McEvoy, Crowe, Murphy y Marnane (2006) [3] se combinan en la implementación del algoritmo SHA2 dos técnicas de optimización muy populares (pipelining y unrolling) para conseguir una gran tasa de procesamiento de datos, aunque es superada por varios trabajos más recientes como el realizado por Madhavi, Hanumantha Rao, Malyadri y Rama Krishna Prasad (2012) [4], en donde aparecen varias tablas muy interesantes comparando los resultados obtenidos con otros trabajos.

Se podrían seguir enumerando muchos otros documentos que aportan novedosas técnicas buscando aumentar la velocidad de procesado o la reducción de la lógica empleada, pero observando que existe tanta información y trabajos al respecto se descartó esta línea de trabajo, ya que se consideró que no se iba a realizar una gran aportación en este campo.

Por otro lado, se consultaron diversos artículos relacionados con implementaciones de algoritmos criptográficos empleando la herramienta System Generator y se ha observado que existe bastante literatura al respecto, ya que esta herramienta permite realizar diseños y validarlos en FPGA sin necesidad de tener conocimientos en lenguajes de descripción hardware. Por tomar un par de ejemplos, Sánchez, Alvarez y Sully Sánchez (2007) [5] proponen una arquitectura basada en filtros para procesado de la imagen y Alia Arshad,

Kanwal Aslam, Dur-e-Shahwar Kundi y Arshad Aziz (2014) en [6] realizan una implementación del algoritmo criptográfico AES que puede ser ejecutada a 288.19 MHz con una tasa de salida de 36.864 Gbps.

Respecto a la co-simulación hardware no se han encontrado muchos trabajos al respecto. En el trabajo de Panduranga, Kumar y Sharath Kumar (2013) [7] se comenta una aplicación sobre cifrado de imágenes múltiples pero no se describe cómo realizar la co-simulación. Quizás la publicación más relacionada con este Trabajo sea la de Denning, Devlin e Irvine (2004) [8], donde se discute sobre el uso de System Generator para co-simulation hardware en FPGA del algoritmo de cifrado AES-128. En este artículo se muestran datos de cómo se consigue una co-simulación tres veces más rápida usando una red TCP/IP uniendo dos nodos situados a una distancia de aproximadamente 600 kilómetros, que en una simulación convencional de Simulink sobre un PC. Otra co-simulación hardware de este estudio reduce el tiempo de simulación en un 4000% ejecutándose a través del bus PCI. En este artículo, se comenta cómo la co-simulación hardware permite liberar al PC de la carga computacional que conlleva la simulación para poder ser destinado a otras tareas.

Estos trabajos aportan datos sobre las virtudes de usar la co-simulación hardware pero en ninguno de estos trabajos se comenta cómo realizar el proceso que permite llevar a cabo tal co-simulación. Para averiguar cómo implementar esta tarea sin duda alguna la mayor ayuda se ha obtenido del documento de Xilinx “System Generator for DSP User Guide” (2010) [9] y de “System Generator for DSP Reference Guide” (2012) [23] donde se describe el funcionamiento de esta herramienta, así como un video tutorial (Xilinx. Getting Started with System Generator) [10] y la documentación asociada (Xilinx, 2010) [11]. Por otro lado, se ha tenido que consultar en reiteradas ocasiones la ayuda online que proporciona Mathworks para Simulink [12].

El código que se ha utilizado para los algoritmos criptográficos del presente trabajo, se distribuye de forma gratuita bajo licencia GPL (GNU General Public License). Los enlaces de descarga se indican en [13] (Arif Endro, 2010) y [14] (de La Piedra, 2014). La principal referencia a la hora de entender el funcionamiento de dichos algoritmos se ha extraído de [15] (Anónimo) y del trabajo de Dunkelman (2012) [16].

En cuanto a la elección de la tecnología a emplear, existen varios artículos en los que se comparan GPUs, FPGAs y CPUs. En [17], dos investigadores de Microsoft (Sirowy y Forin, 2008) analizan los mecanismos que permiten a las FPGAs sobrepasar a los

microprocesadores en determinadas tareas de cómputo siendo ejecutadas a una frecuencia de reloj mucho menor.

En el estudio de Thomas, Howes y Luk (2009) [18] se realiza una comparativa entre CPUs, GPUs, FPGAs y MPPAs (Massively Parallel Processor Arrays) para un algoritmo de generación de números aleatorios e intenta identificar para cada plataforma la implementación óptima del algoritmo. Resulta interesante este artículo por la breve pero clara definición que realiza de los distintos tipos de dispositivos que analiza y por las tablas de resultados que obtiene. En [19] Cullinan, Wyanty y Frattesi (trabajadores de Mathworks) realizan medidas sobre las tres plataformas de cómputo más importantes: CPUs, GPUs y FPGAs; realizando 66 pruebas con el objetivo de que sirvan cómo referencias para la elección del dispositivo más adecuado para las distintas aplicaciones que se tratan. Es importante destacar que en estas pruebas resulta ganadora en término de operaciones por segundo la GPU, pero debido a que no se incluyen los tiempos en transmisión y recepción de datos. Si se incluyen estos tiempos las FPGAs resultan claras vencedoras. Venugopal y Manikantan Shila (2013) [20] comentan la necesidad de procesado en tiempo real en el cifrado/descifrado de aplicaciones basadas en transmisiones sobre redes y también tratan sobre el grado de paralelismo y las tareas computacionalmente intensivas que caracterizan a los algoritmos criptográficos. Incluye una comparativa de implementaciones de los algoritmos TEA y XTEA en GPU y FPGA. Este artículo propone el uso de una herramienta llamada CHAAT (Cryptographic Hardware Acceleration and Analysis Tool) para facilitar la selección del algoritmo criptográfico óptimo para cada plataforma.

# 3. Objetivos

El principal objetivo de este Trabajo es implementar una arquitectura de comunicación PC-FPGA para la verificación de algoritmos criptográficos y demostrar las mejoras que supone este tipo de arquitectura para acelerar las etapas de diseño y verificación frente a las soluciones que existen basadas únicamente en PC o en FPGA.

Las simulaciones realizadas sobre PC de una aplicación que se ejecutará posteriormente en una FPGA suelen ser muy lentas, y los bancos de pruebas requieren un trabajo muy tedioso para abarcar todas las posibilidades.

Por otro lado, la verificación física sobre FPGA requiere el uso de pines de depuración y un analizador lógico, lo que obliga a que sólo puedan observarse unas pocas señales con cada prueba. Existe la posibilidad de instanciar dentro de la FPGA un analizador lógico que conste de una serie de búferes que almacenen las muestras que se desee observar, para posteriormente visualizarlas a través del JTAG con una herramienta en el PC (ChipScope con la familia de FPGAs de Xilinx). Esta opción presenta la limitación de que el analizador lógico usará recursos de la propia FPGA y puede incluso llegar a afectar al funcionamiento del sistema, puesto que al instanciar este componente, el diseño podría incumplir los requerimientos de frecuencia máxima impuestos.

La generación de vectores de entrada y análisis de los datos de salida haciendo uso del lenguaje de programación Matlab y del interfaz gráfico Simulink, añadirá unas ventajas importantes en cuanto a potencia computacional en el procesado de datos y también en lo relativo a la facilidad de programación.

Como objetivos específicos que se desean cumplir en las diferentes fases del presente Trabajo se pueden enumerar los siguientes:

1. Concebir la arquitectura HW más adecuada para la implementación de algoritmos criptográficos.
2. Elección e implementación de varios algoritmos criptográficos para dicha arquitectura.
3. Diseñar una aplicación genérica para verificar el funcionamiento de los algoritmos que sea ejecutada íntegramente en PC.
4. Crear un modelo para realizar co-simulación hardware con el objetivo de disminuir los tiempos de verificación obtenidos por la anterior aplicación.
5. Evaluar los resultados obtenidos.

# 4. Metodología

En el presente apartado, se realizará una descripción de la metodología seguida para la consecución de objetivos intermedios hasta lograr el principal. Se describirá el proceso seguido aunque será en el siguiente apartado donde realmente se profundizará en el funcionamiento de la aplicación.

## 4.1. Elección de la tecnología

Existen diversas alternativas para usar como co-procesador en un sistema para liberar de la carga computacional que supone un algoritmo criptográfico al procesador principal del sistema. Entre los distintos dispositivos existentes en el mercado las tres alternativas más interesantes son: DSP (Digital Signal Processor), GPU (Graphical Processing Unit) y FPGA (Field Programmable Gate Array).

El DSP es un microprocesador especializado en realizar determinadas tareas de procesado de señal digital como puede ser la compresión de video digital, de audio, implementar interfaces de comunicaciones… Su arquitectura está basada en reaprovechar el uso de un core formado por unidades aritméticas definidas (multiplicaciones de X bits, sumas…). Por ejemplo, en el caso de la arquitectura VLIW (Very Long Instruction Word) usada por Texas Instruments en la familia c64+ se dispone de dos multiplicadores y 6 ALUs (arithmetic logic units). Esta arquitectura permitiría realizar hasta 8 operaciones en un mismo ciclo de reloj (Texas Instruments, 2010) [21]. Estos recursos serán compartidos por todas las funciones que se ejecuten a lo largo del programa, de igual forma que los buses de acceso a memoria serán compartidos por el core del DSP y otros periféricos, obligando a establecer prioridades de acceso.
Se programan utilizando lenguaje C, lo que permite que resulte bastante sencillo realizar determinadas tareas, aunque para obtener un rendimiento adecuado y una optimización de recursos se requieren conocimientos sobre el uso de CACHE, DMA, los periféricos involucrados en la aplicación… en definitiva se requiere un conocimiento del hardware que compone internamente dicho DSP. En determinadas funciones críticas para el sistema en cuestiones de velocidad de ejecución, se pueden requerir conocimientos de un lenguaje de

programación específico del fabricante para optimizar el paralelismo en la ejecución de las instrucciones (Texas Instruments utiliza los denominados “intrinsics”) o el uso de lenguaje ensamblador.
La utilización del lenguaje C permite programar fácilmente rupturas del pipeline en el flujo de un programa condicional, lo cual es bastante difícil de implementar dentro de una FPGA. Esto no quiere decir que la ruptura del pipeline no afecte al rendimiento del DSP, ya que cualquiera de los 3 dispositivos planteados sufre una importante degradación de prestaciones con flujos condicionales, pero sin duda el DSP es el menos afectado y en el que resulta más sencillo implementar un código de este tipo.

La GPU al igual que el DSP presenta un set de recursos fijos que son utilizados por los programas que ejecutan, pero las GPUs están diseñadas para realizar un paralelismo de operaciones masivo, basándose su arquitectura en el uso de arrays de pequeños procesadores. Otro aspecto que optimiza la GPU es el acceso a memoria externa, puesto que su diseño está orientado a mover enormes cantidades de datos. Como su nombre indica son procesadores gráficos, por lo que son dispositivos especializados para realizar determinadas tareas. Si la aplicación que se va a desarrollar puede hacer un uso óptimo de las operaciones para las que está diseñada la GPU, podría ser utilizada para realizar tareas no relacionadas con el procesado de imagen o vídeo. Para su programación se usan principalmente dos lenguajes: CUDA (extensiones de C, C++ y Fortran para las GPUs del fabricante Nvidia) y OpenCL (u OpenGL en operaciones gráficas). La manera más sencilla de programar GPUs es mediante librerías de APIs, pero si éstas no proporcionan las operaciones necesarias para la aplicación que se desee desarrollar existe la posibilidad de programar un set de APIs propio. Esta última tarea es bastante tediosa y complicada de realizar.

Frente a los dos dispositivos anteriores que se pueden considerar realmente como aceleradores SW destinados a acelerar ciertas aplicaciones computacionalmente intensivas, la FPGA proporciona un hardware configurable por el usuario. En las arquitecturas anteriores, el programa hacía uso de unos recursos ya existentes, en la FPGA el programador configura unos elementos lógicos básicos (LUTs en Xilinx y Logic Elements/Cells para Altera) que permiten construir los recursos que se deseen utilizar y configurar cómo se interconectarán entre ellos, para ajustarse perfectamente a las necesidades de la aplicación desarrollada. Esto conlleva de forma intrínseca una importante reducción del consumo. Para que sirva como ejemplo ilustrativo, se comentó anteriormente que el DSP de Texas Instruments (familia c64+)

disponía de 6 ALUs, en el caso de la FPGA se crearían las ALUs que se requieran y se estimen oportunas para la aplicación en concreto.
Esto permite una gran versatilidad y optimizar los recursos para el uso concreto al que estén destinados, aunque se debe tener en cuenta que para cada posible configuración o condición que se de en la aplicación se deberán destinar recursos. Si bien en ciertos casos se podrán reutilizar recursos, en la gran mayoría de los casos será una tarea compleja o degradará la frecuencia máxima alcanzable. Como el hardware ha sido configurado para una misión específica, no se llevan a cabo cargas de instrucciones (fetch) para determinar la operación a realizar como en el caso de los otros procesadores. Este hecho conlleva una importante reducción en cuanto a ciclos de ejecución se refiere, como se indica en el trabajo de Sirowy y Forin (2008) [17].

Si las operaciones matemáticas a realizar son especialmente complejas, el dispositivo más adecuado suele ser el DSP ya que dispone de los recursos necesarios para realizar este tipo de operaciones. Las FPGAs trabajan de manera óptima con operaciones de punto fijo mientras que para operaciones de coma flotante normalmente se requiere el uso de gran cantidad de lógica, no llegando a ser su implementación en cuanto a prestaciones comparable al DSP.

Por otro lado, las interconexiones en las GPUs suelen realizarse por PCI Express mientras que en los DSPs se limitan a los periféricos existentes en un modelo determinado. Este hecho limita el uso de GPUs a soluciones HW basadas en PC. Tanto las GPUs como los microprocesadores actuales utilizados en PCs no tienen un buen rendimiento en cuanto a consumo se refiere, por lo que no son adecuadas para dispositivos portátiles. Los DSPs como ya se ha comentado, pueden ser utilizados siempre y cuando presenten las conexiones que vaya a necesitar la aplicación por medio de sus puertos de entrada/salida y ofrezcan unas prestaciones en cuanto a velocidad adecuadas.
Las FPGAs disponen de una gran cantidad de pines de propósito general que pueden ser interconectados a prácticamente cualquier dispositivo siempre y cuando se respeten los niveles de tensión, por lo que resultan adecuadas para la mayoría de aplicaciones.

Tras analizar todo lo comentado, se pueden obtener las siguientes conclusiones:

- Mediante el uso de una FPGA y escogiendo el modelo adecuado, se usarán únicamente los recursos necesarios para la implementación del algoritmo o

algoritmos criptográficos que se escojan. Esta reducción de recursos lleva consigo una importante reducción de costes y consumo.

- El uso de una FPGA permite la posibilidad de conectarse a prácticamente cualquier dispositivo y posteriormente, implementar el interfaz que sea necesario.
- Independientemente del interfaz que se elija para realizar la comunicación con el procesador principal, la FPGA permitirá procesar en paralelo el número de bits que se estime oportuno. Debido a que es un dispositivo programable, permitirá realizar versiones posteriores modificando el código para aumentar o reducir el número de operaciones que se realizan en paralelo.
- Debido al grado de paralelismo de operaciones que permiten, las FPGAs pueden obtener importantes tasas de procesamiento de datos a frecuencias más bajas de funcionamiento que los otros dispositivos. Esto permite que el consumo de estos dispositivos sea inferior al de GPUs, CPUs o DSPs.
- Si los programadores no cuentan con la experiencia necesaria para programar el dispositivo de una manera eficaz usando lenguajes de descripción hardware, las herramientas que se utilizarán para crear el framework de intercomunicación del presente Trabajo permiten generar código de manera automática, lo que permitirá cumplir plazos de entrega ajustados.

En la siguiente gráfica se puede observar una comparativa extraída de [18] (Thomas et al, 2009), en la que comparan algoritmos de generación aleatoria de números (RNGs) optimizados para cada plataforma.

**Table 6: Comparison of absolute performance and efficiency of RNGs across platforms.**

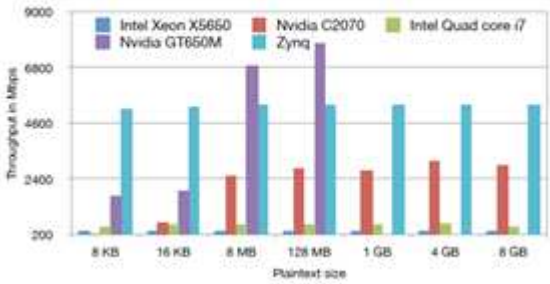
| | Performance (GSample/s) | | | | Efficiency (MSample/joule) | | | |
|---|---|---|---|---|---|---|---|---|
| | CPU | GPU | MPPA | FPGA | CPU | GPU | MPPA | FPGA |
| Uniform | 4.26 | 16.88 | 8.40 | 259.07 | 15.20 | 140.69 | 600.00 | 8635.73 |
| Gaussian | 0.89 | 12.90 | 0.86 | 12.10 | 3.17 | 107.52 | 61.48 | 403.20 |
| Exponential | 0.75 | 11.92 | 1.29 | 26.88 | 2.69 | 99.36 | 91.87 | 896.00 |
| Geo Mean | 1.42 | 13.75 | 2.10 | 43.84 | 5.07 | 114.55 | 150.21 | 1461.20 |
| | Relative Mean Performance | | | | Relative Mean Efficiency | | | |
| | CPU | GPU | MPPA | FPGA | CPU | GPU | MPPA | FPGA |
| CPU | 1.00 | 9.69 | 1.48 | 30.91 | 1.00 | 9.26 | 18.00 | 175.14 |
| GPU | 0.10 | 1.00 | 0.15 | 3.19 | 0.11 | 1.00 | 1.95 | 18.92 |
| MPPA | 0.67 | 6.54 | 1.00 | 20.85 | 0.06 | 0.51 | 1.00 | 9.73 |
| FPGA | 0.03 | 0.31 | 0.05 | 1.00 | 0.006 | 0.05 | 0.10 | 1.00 |

*Extraída de Thomas et al, 2009, pp. 9*

En dichos resultados se observa como la FPGA resulta ser la mejor opción en términos de eficiencia de consumo y prestaciones. Pero en dicha tabla no aparece ninguna métrica relacionada con el precio de los dispositivos, y la FPGA empleada en estas medidas tiene un precio superior al resto de dispositivos. Debido a esto, resulta de vital importancia escoger una FPGA que cubra las necesidades del proyecto y se ajuste al presupuesto (no sobredimensionarla escogiendo un dispositivo mucho mayor al necesario y ceñirse a los requerimientos).

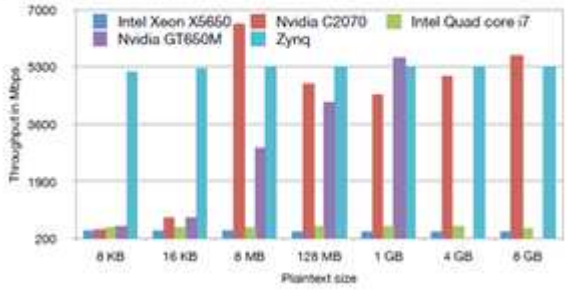
En el trabajo de Venugopal y Manikantan Shila (2013) [20], presentan las siguientes dos gráficas mostrando los resultados obtenidos para los algoritmos criptográficos TEA y XTEA:

Figure 5: Throughput (Mbps) comparison of TEA on CPU, GPU and FPGA




*Extraída de Venugopal y Manikantan Shila (2013), pp.4*

Figure 6: Throughput (Mbps) comparison of XTEA on CPU, GPU and FPGA




*Extraída de Venugopal y Manikantan Shila (2013), pp.4*

En dichas gráficas podemos observar cómo la solución basada en FPGA (Zynq de Xilinx) ofrece unos resultados más independientes del tamaño del texto aunque sea superada en prestaciones para algunos tamaños grandes. También se observa cómo dependiendo del algoritmo la GPU ganadora resulta ser un modelo u otro, mientras que la FPGA siempre mantiene unos buenos resultados.

## 4.2. Elección de los algoritmos

Los algoritmos criptográficos que se utilicen deberán ser adecuados al dispositivo que finalmente se ha elegido, la FPGA. De todo lo comentado en el apartado anterior se deduce que para que la implementación sea óptima, se deberá trabajar con algoritmos cuyas operaciones sean fácilmente paralelizables y se trate de operaciones no excesivamente complejas (descartando operaciones en coma flotante, divisiones, raíces cuadradas…). Sólo de esta forma realmente se beneficiará la implementación del uso de FPGAs.

Del algoritmo escogido se realizarán tres implementaciones. Primeramente, se establecerá el nivel en el que la seguridad de dicho algoritmo ha sido quebrantada (lo que comúnmente se conoce como que se ha "roto" el algoritmo). Una vez conocido dicho nivel, se realizará una versión en la zona límite de ruptura; otro superando dicha zona y por lo tanto, actualmente considerado inquebrantable y el último, un algoritmo con una configuración que se sabe de antemano que es vulnerable. Para las tres implementaciones se obtendrán resultados de ocupación en lógica del dispositivo, del número de operaciones por segundo que es capaz de realizar el algoritmo y del consumo correspondiente.

Como se ha comentado anteriormente, en el presente Trabajo se pretende crear un Framework que pueda ser utilizado para diversos algoritmos criptográficos de tal forma que las modificaciones a realizar para pasar de utilizar un algoritmo a otro sean mínimas. Para conseguir un framework fácilmente extrapolable, se parte del uso de algoritmos que emplean bloques de N bits en los que se divide el conjunto de datos inicial (ya sea un fichero de texto o de otro tipo). Esta condición se cumple para gran cantidad de algoritmos empleados en la criptografía como son las funciones resumen SHA1 (bloques de 512 bits), SHA256 (512 bits), SHA512 (1024 bits) o los cifradores de bloque DES (64 bits), IDEA (64bits)…

En este Trabajo se utilizarán los algoritmos SHA160, SHA256 y SHA512; ya que las modificaciones a realizar en el framework para integrar estos algoritmos serán suficientes para describir cómo incorporar otros algoritmos.

### 4.2.1. Funciones resumen SHA

Una función resumen (también llamadas funciones hash) de n-bit es una operación por la cual un mensaje de cualquier tamaño es convertido en otro mensaje de longitud n-bit. Normalmente son utilizadas para la protección de passwords y la generación de firmas digitales. Dentro de las propiedades que caracterizan una función resumen encontramos las comentadas en los apuntes de la asignatura de Criptografía y Mecanismos de Seguridad (Universidad Internacional de la Rioja, 2013) [22]:

- Debe requerir tiempos de computación muy elevados obtener dos mensajes que produzcan el mismo resumen.
- Tiene que ser computacionalmente intratable conseguir el mensaje en claro a partir del resumen.
- Igualmente, encontrar dos mensajes aleatorios que generen el mismo resumen debe ser computacionalmente intratable. Este concepto se conoce como colisión.

Actualmente se han encontrado colisiones en las implementaciones de las funciones hash MD4, MD5, SHA0 y SHA1. Para corregir dichos problemas de seguridad se han creado las versiones SHA2 y más recientemente, SHA3.

Las funciones SHA0 y SHA1 generan mensajes de 160 bits. SHA2 incluye una serie de importantes cambios respecto a sus predecesores así como el tamaño del resumen generado se incrementa hasta 224, 256, 384 o 512bits. Como ya se ha comentado los algoritmos escogidos para ser implementados en la FPGA han sido SHA1, SHA256 y SHA512. Sobre dichos algoritmos se realizará una comparativa en cuanto a velocidad, ocupación lógica y consumo. La elección de estos algoritmos viene condicionada por el hecho de que las operaciones que se realizan sobre el mensaje para obtener el resumen son muy sencillas y a que dividen los datos de entrada en bloques de longitud fija (N bits) para operar con ellos. Se trata de operaciones OR, XOR y AND a nivel de bit, desplazamientos, rotaciones y sumas que fácilmente pueden ser implementadas en una FPGA.

## 4.3. Implementación algoritmos

En este apartado se procederá a describir la implementación de los algoritmos criptográficos finalmente escogidos. Básicamente, se comentará el funcionamiento de cada algoritmo y los distintos módulos y operaciones que lo conforman. Dicha descripción será utilizada para entender el funcionamiento del modelo de referencia y de la instancia que se ejecutará en la FPGA.

### 4.3.1. SHA1

Sus siglas provienen de “Secure Hashing Algorithm”, es un algoritmo para crear resúmenes diseñado por “United States National Security Agency” (NSA) y publicado por el NIST. Supone una versión mejorada de su antecesor SHA0 y su primera publicación fue en 1995.
SHA1 genera resúmenes de 160 bits para mensajes de longitud de hasta $2^{64}$ -1 bits. Su implementación es muy similar a la de MD4 o MD5, incluso compartiendo los mismo valores de inicialización. Estas similitudes provocaron, que tras demostrar que existían colisiones en el algoritmo MD5 en 2005, SHA1 comenzase a considerarse como una función resumen poco segura, ya que podría ser vulnerada en un futuro bajo un ataque similar. Microsoft anunció el pasado año que dejaría de aceptar certificados que utilizasen SHA1 después de 2016, tras un incidente ocurrido en Irán con un malware denominado Flame. Este malware explotaba una debilidad en MD5 junto a una vulnerabilidad en el Terminal Server, la conjunción de ambas vulnerabilidades permitía al atacante crear un certificado fraudulento firmado por Microsoft. Tras conocerse este hecho, Microsoft retiró inmediatamente el uso de MD5 en sus actualizaciones.
El funcionamiento del algoritmo SHA1 es el siguiente:

- El algoritmo trabaja sobre bloques de 512 bits, por lo que para completar el mensaje hasta alcanzar una longitud múltiplo de 512bits se introduce relleno de la siguiente forma:
    - Al final del mensaje se introducen 64 bits indicando la longitud real del mensaje.

- Entre el mensaje original y los 64bits anteriores, se introducen un ‘1’ seguido de X ceros. Donde X = 448 – 1 - longitud del mensaje. 448 se obtiene de restar a 512 bits que debe ser la longitud final del mensaje de entrada al algoritmo, los 64bits destinados a la longitud real.

- Se divide el mensaje en N bloques de longitud 512bits. Donde N = Longitud del mensaje / 512.
- Se inicializan los cinco valores hash intermedios con las siguientes constantes:

A = 0x67452301
B = 0xEFCDAB89
C = 0x98BADCFE
D = 0x10325476
E = 0xC3D2E1F0

- Se aplica la función de compresión a cada bloque:

$h_i = H(h_{i-1}, M_i)$ es aplicado sobre el valor previo $h_{i-1} = (A,B,C,D,E)$ y el bloque del mensaje.

- Después de ejecutar las N iteraciones el valor de $h_n$ será el resumen final del mensaje obtenido de longitud igual a 160 bits.
- La función de compresión sigue el siguiente esquema:

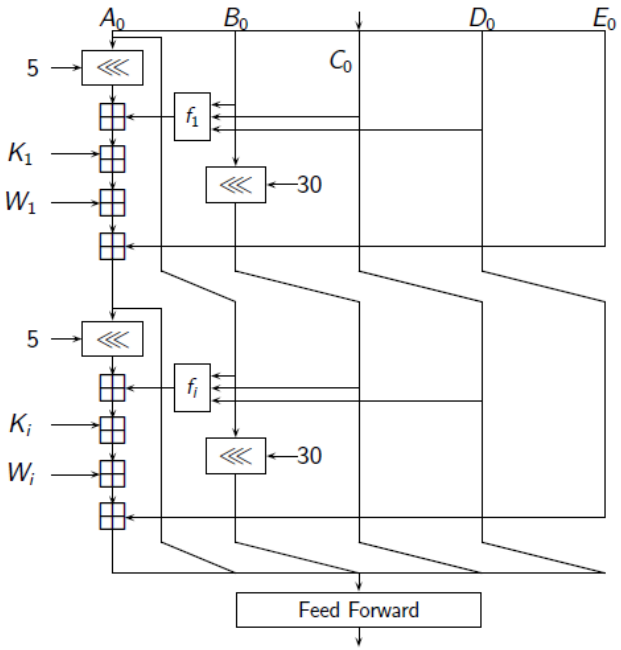



*Figura extraída de* Dunkelman *(2012), pp.7*

- Se divide el bloque Mi en 16 sub-bloques de 32bits: W0,W1…W15
- Para cada t desde 16 a 79 se realiza la siguiente operación:
  - $W_t = (W_{t-3}$ XOR $W_{t-8}$ XOR $W_{t-14}$ XOR $W_{t-16})<<<1$

*NOTA: La rotación de un bit fue incluida en SHA1, en SHA0 no se realizaba y es la única diferencia entre ambos algoritmos.*

- Se actualizan los valores A0,B0, C0, D0 y E0 con $h_{i-1}$
- Para cada t desde 0 hasta 79:
  - $T = A_t <<< 5 + f_t(B_t,C_t,D_t) + E_t + W_t + K_t$
  - $E_{t+1} = Dt$, $D_{t+1} = C_t$, $C_{t+1} = B_t <<< 30$, $B_{t+1} = A_t$
    $A_{t+1} = T$
- A = A0 + A80, B = B0 + B80, C = C0 + C80
  D = D0 + D80, E = E0 + E80 (mod $2^{32}$)
- La función ft y los valores de Kt que se pueden observar en el esquema mostrado son:

$$
\begin{array}{rll}
0 \leq t \leq 19: & f_t(X,Y,Z) = XY \vee (\neg X)Z & K_t = \texttt{5A827999} \\
20 \leq t \leq 39: & f_t(X,Y,Z) = X \oplus Y \oplus Z & K_t = \texttt{6ED9EBA1} \\
40 \leq t \leq 59: & f_t(X,Y,Z) = XY \vee XZ \vee YZ & K_t = \texttt{8F1BBCDC} \\
60 \leq t \leq 79: & f_t(X,Y,Z) = X \oplus Y \oplus Z & K_t = \texttt{CA62C1D6}
\end{array}
$$

### 4.3.2. SHA256

En el algoritmo SHA-256 la función de compresión opera sobre bloques de mensaje de 512 bits y con valores de hash intermedios de 256 bits. Esencialmente se trata de un algoritmo de cifrado el cual cifra un valor de hash intermedio utilizando el bloque del mensaje como clave.

Para comenzar, el mensaje sobre el que se aplicará la función resumen debe ser extendido hasta formar un mensaje cuya longitud sea múltiplo de 512 bits. Posteriormente, dicho mensaje será dividido en bloques de 512 bits que se irán proporcionando a la función hash de uno en uno:

H(i) = H(i-1) + Cm(i)(H(i-1))

La anterior operación se realizará N veces. Siendo N = Longitud del mensaje /512. “C” es la función de compresión del algoritmo SHA256 y la suma se realizará a nivel de palabras de 32bits mod $2^{32}$.
El valor inicial H(0) es obtenido tomando la parte fraccional de las raíces cuadradas de los 8 primeros números primos. A continuación, se muestra la secuencia de palabras de 32bits que conforma dicho valor inicial:
*H(0)1 = 0x6a09e667 , H(0)2 = 0xbb67ae85, H(0)3 = 0x3c6ef372, H(0)4 = 0xa54ff53a*
*H(0)5 = 0x510e527f, H(0)6 = 0x9b05688c, H(0)7 = 0x1f83d9ab,H(0)8 = 0x5be0cd19*

Para completar el mensaje hasta alcanzar una longitud múltiplo de 512bits se introduce relleno de la misma manera que para SHA1. Por ejemplo, para el texto “unir” la longitud del mensaje es 32bits, siendo la traducción a binario:

01110101 01101110 01101001 01110010

A continuación, añadiremos un ‘1’ seguido de X ceros, donde X = 448-1-32 = 415. Y al final se añadirá la longitud del texto de entrada (32 representado en 64bits). Así que el mensaje de entrada, tras insertar relleno sería:
01110101 01101110 01101001 01110010 1000….0000 000…100000
000….0000→ X ceros
000…100000 → Número 32 representado con 64bits.

Este algoritmo trabaja en sub-bloques de 32 bits por lo que habrá que dividir el mensaje sobre el que realizar el resumen en N-bloques de 512 bits y posteriormente, dividirlo en 16 sub-bloques de 32 bits.
Para cada bloque de 512bits se realizarán las siguientes operaciones:

- Inicializar los registros a, b, c, d, e, f, g y h con el valor intermedio hash de la iteración anterior.
  - a ← H(i-1)0; b ← H(i-1)1 … h ← H(i-1)8
- Actualizar los registros aplicando la función de compresión de SHA256. Se realizan 64 veces las siguientes operaciones:

$$T1 \leftarrow h + \sum 1(e) + Ch(e,f,g) + Kj + Wj$$
$$T2 \leftarrow \sum 0(a) + Maj(a,b,c)$$
$$h \leftarrow g$$
$$g \leftarrow f$$

f←e
e←d + T1
d←c
c←b
b←a
a←T1+ T2

La función de compresión de SHA256 se representa en la siguiente figura:

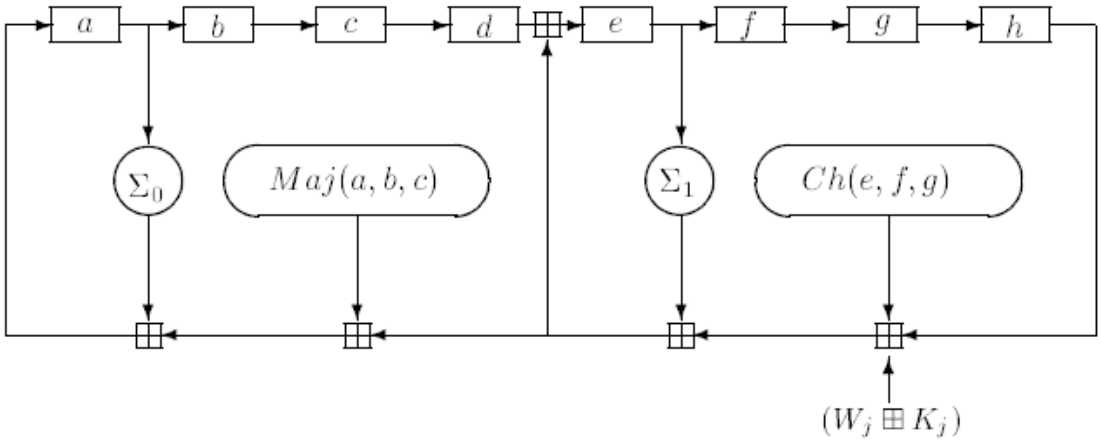



*Figura extraída de [15]* (Anónimo)*, pp. 10*

Se calcula el valor hash intermedio H(i)
H1(i) ← a + H1(i-1)…… H8(i) ← h + H8(i-1)

- El valor H8(N) = (H1(N), H2(N)… H8(N)) compone el resumen resultado del mensaje de entrada M.

Las funciones utilizadas se definen de la siguiente forma:

Ch(x,y,z ) = (x AND y) XOR (~x AND z)
Maj (x,y,z) = (x AND y) XOR (x and z) XOR (y AND z)
$\sum 0(x) = S^2(x)$ XOR $S^{13}(x)$ XOR $S^{22}(X)$
$\sum 1(x) = S^6(x)$ XOR $S^{11}(x)$ XOR $S^{25}(X)$
$\sigma 0(x) = S^7(x)$ XOR $S^{18}(x)$ XOR $R^3(X)$
$\sigma 1(x) = S^{17}(x)$ XOR $S^{19}(x)$ XOR $R^{10}(X)$

Como se puede observar en el esquema un conjunto de 64 constantes son empleadas (K0 - K63). Dichas constantes son las siguientes:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

Por otro lado la obtención de Wj se realiza siguiendo este esquema:

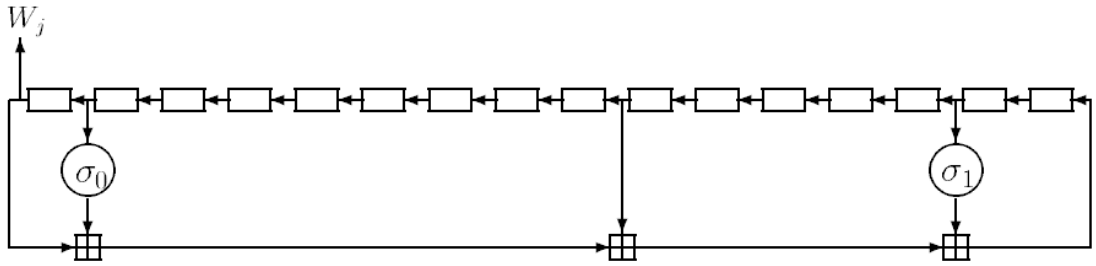



*Figura extraída de [15]* (Anónimo)*, pp.10*

Wj = Mj(i) para j = 0-15

Para j = 16 hasta j=63

$Wj = \sigma1(W_{j-2}) + W_{j-7} + \sigma0(W_{j-15}) + W_{j-16}$

### 4.3.3. SHA 512

El flujo de programa de la función hash SHA 512 es muy parecido al anterior pero dividiendo el mensaje en N bloques de 1024 bits para posteriormente, trabajar con sub-bloques de palabras de 64bits. Los valores hash intermedios son de 512 bits.

El relleno inicial se realiza de forma análoga al anterior algoritmo pero en esta ocasión, hasta conseguir un tamaño múltiplo de 1024. Las sumas se realizarán a nivel de palabras de 64 bits módulo $2^{64}$.

Los valores iniciales H(0) para la primera carga de los registros también son obtenidos mediante la parte fraccional de las raíces cuadradas de los 8 primeros números primos pero creando palabras de 64bits en lugar de 32.

$$H_1^{(0)} = \texttt{6a09e667f3bcc908}$$
$$H_2^{(0)} = \texttt{bb67ae8584caa73b}$$
$$H_3^{(0)} = \texttt{3c6ef372fe94f82b}$$
$$H_4^{(0)} = \texttt{a54ff53a5f1d36f1}$$
$$H_5^{(0)} = \texttt{510e527fade682d1}$$
$$H_6^{(0)} = \texttt{9b05688c2b3e6c1f}$$
$$H_7^{(0)} = \texttt{1f83d9abfb41bd6b}$$
$$H_8^{(0)} = \texttt{5be0cd19137e2179}$$

El relleno inicial se realiza de la misma manera que en el algoritmo anterior pero la longitud real del mensaje (sin relleno) se representa mediante 128bits. Por lo tanto, el relleno que se introducirá se conforma mediante un ‘1’ seguido de X ceros. Siendo X = 896 – 1 – Longitud del mensaje de entrada.

Las funciones que se modifican respecto al anterior algoritmo son las siguientes:

$$\Sigma_0(x) = S^{28}(x) \oplus S^{34}(x) \oplus S^{39}(x)$$
$$\Sigma_1(x) = S^{14}(x) \oplus S^{18}(x) \oplus S^{41}(x)$$
$$\sigma_0(x) = S^{1}(x) \oplus S^{8}(x) \oplus R^{7}(x)$$
$$\sigma_1(x) = S^{19}(x) \oplus S^{61}(x) \oplus R^{6}(x)$$

El bucle de la función de compresión se ejecuta 80 veces en lugar de las 64 iteraciones que se utilizan en SHA256. A su vez, el bucle para el cálculo de Wj varía para ejecutarse 64 veces en lugar de 32, quedando de la siguiente manera:

Wj = Mj(i) para j = 0-15

Para j = 16 hasta j=79:

$W_j = \sigma1(W_{j-2}) + W_{j-7} + \sigma0(W_{j-15}) + W_{j-16}$

Las constantes de 64 bits utilizadas K0-K79 son las siguientes (hex):

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

## 4.4. Elección de la plataforma

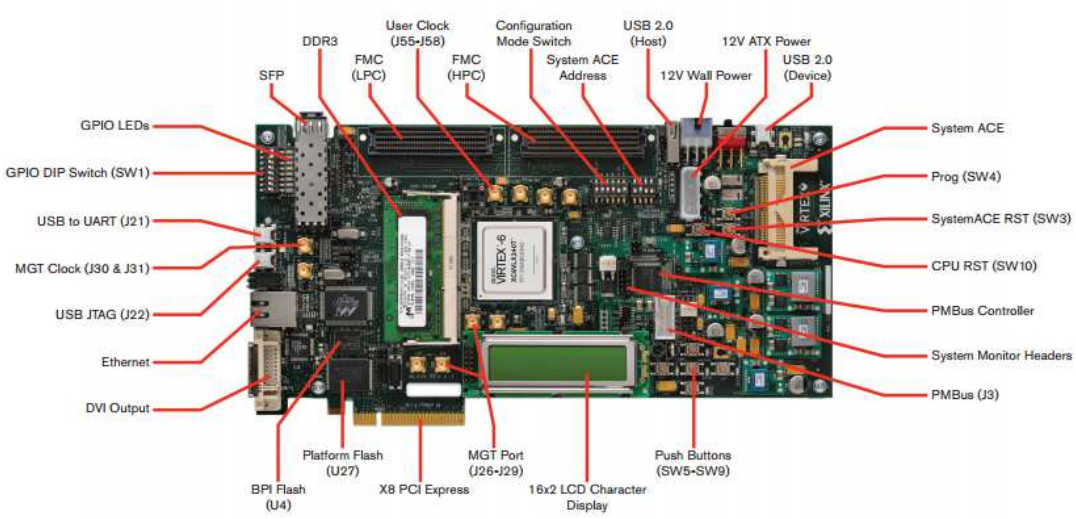
La aplicación ha sido desarrollada utilizando la tarjeta de evaluación ML605 diseñada por el fabricante de FPGAs, Xilinx. El objetivo de proporcionar este tipo de tarjetas es que posibles clientes puedan verificar el funcionamiento del dispositivo Virtex6 LX 240T y disponer de unos diseños de referencia a la hora de crear una PCB basada en tal FPGA. Dentro de las distintas familias de Xilinx, se trata de una FPGA optimizada para el uso de principalmente lógica distribuida, ya que no está destinada a tareas relacionadas con el uso de multiplicadores (familia SXT) ni a disponer de una gran conectividad serie (familia HXT). La lógica disponible en este dispositivo es enorme si consideramos los algoritmos que se van a implementar en ella y que requieren de pocos recursos, pero esto no supone ninguna limitación, ya que podremos verificar el funcionamiento del algoritmo en HW en esta tarjeta y una vez finalizada

la verificación, analizar los resultados obtenidos en la implementación en cuanto a uso de recursos y elegir una FPGA de menor tamaño para el diseño final.

Por supuesto, existen algunas diferencias entre los elementos lógicos que construyen internamente las distintas familias de dispositivos de Xilinx: Virtex-6, Spartan3, Spartan6, Serie-7… Por lo que idealmente, se debería elegir una tarjeta de evaluación de la familia que finalmente se vaya a utilizar, pero la elección de esta plataforma ha venido condicionada por la disponibilidad de conseguir una tarjeta de evaluación para el desarrollo del Trabajo Fin de Máster de manera gratuita.

Por otro lado, para facilitar la realización del presente Trabajo se buscó una tarjeta de evaluación que fuese compatible con la co-simulación HW en la herramienta System Generator, y de esta forma no vernos obligados a tener que desarrollar tareas de configuración extra.

A continuación, se muestra la tarjeta de evaluación ML605 con sus componentes:




Para la realización de la co-simulación hardware será necesario hacer uso del conector USB (a través de la cadena JTAG) para configurar a la FPGA y del conector Ethernet, para llevar a cabo toda la transmisión/recepción de datos.

# 5. Contribución

La co-simulación Hardware es un proceso que ofrece como una de sus ventajas la capacidad de poder verificar el diseño implementado en Hardware sin la necesidad de tener que esperar a tener finalizado un prototipo del equipo. Las tarjetas de evaluación que proporcionan los fabricantes de FPGAs, en muchos casos intentan interconectar la FPGA con el mayor número de periféricos posible para permitir que puedan emplearse para verificar gran cantidad de diseños. En otros casos, existen varias tarjetas de evaluación de la misma FPGA pero destinadas a distintas funcionalidades, como puede ser procesado de vídeo, de audio, interconexión PCI… Pero la realidad demuestra que en gran cantidad de diseños la funcionalidad final no puede probarse con el hardware proporcionado o para probarlo se requiere la configuración de diversos periféricos que finalmente no serán incluidos en la PCB que se desarrolle, implicando un gasto de tiempo (y consecuentemente dinero) en el desarrollo de interfaces que luego no serán utilizados en la aplicación final. Gracias a la co-simulación hardware mediante la herramienta System Generator, la configuración de los dispositivos involucrados en la transferencia de datos se realizará de manera automática y totalmente transparente para el usuario.

Otra de las ventajas de la co-simulación Hardware reside en que permite que las entradas al sistema que se va a verificar sean generadas por el PC desde el entorno Simulink-Matlab, proporcionando una potencia matemática sin igual para introducir en las pruebas gran cantidad de vectores de test. De igual manera, los resultados que se generan en la FPGA pueden alimentar funciones en el PC que permitan analizar los resultados con todos los medios disponibles en el entorno Simulink-Matlab. De todo lo dicho se extrae que una gran ventaja que proporciona esta forma de verificación es el uso de la potencia de un PC a la hora de gestionar y analizar los datos de entrada/salida que se proporcionan o reciben a o desde el diseño que se ejecuta en la FPGA.

Cuando se diseñan e implementan diseños destinados a ser ejecutados en la FPGA, una de las fases clave del ciclo de desarrollo del código es la verificación de dicho diseño. Las depuraciones se realizan viendo señales digitales que son actualizadas en instantes infinitesimales, usando el PC a modo de analizador lógico sobre una ejecución virtual. Aunque se permite trabajar con entrada/salida de ficheros y se pueden llegar a realizar bancos de prueba muy versátiles, normalmente la realización de estos bancos de pruebas suele ser

bastante compleja y requiere de personal muy cualificado y con grandes conocimientos en lenguajes de descripción hardware. Por otro lado, debido a que estas simulaciones trabajan a nivel de bit sobre una lógica virtual, la velocidad de las simulaciones suelen ser muy dependientes de la plataforma de simulación de la que se disponga, pudiendo llegar a ser muy lentas.

La operación conjunta de Matlab, Simulink y System Generator permite realizar estos test-benches en un lenguaje más sencillo de utilizar y matemáticamente hablando, mucho más potente como es Matlab. Desde este punto de vista, únicamente se proporcionan mejores vectores de entrada y se optimiza el análisis de los resultados; pero se sigue teniendo el problema de la velocidad en la ejecución. La solución a este problema reside en ejecutar la parte correspondiente al código de la FPGA realmente en el Hardware y la generación y análisis de resultados en el PC, lo cual conducirá a una importante reducción en cuanto a tiempos de simulación se refiere.

Aprovechando la cualidad existente en gran cantidad de algoritmos criptográficos de trabajar con bloques de longitud fija, se realizará un framework en el que simplemente con la sustitución del bloque que contiene al algoritmo y la modificación del valor de la variable en el que se indica el tamaño de bloque usado, se pueda ejecutar otro algoritmo dentro del mismo proyecto. Esto supone claras ventajas en cuanto al mantenimiento y reusabilidad del código en la realización de bancos de pruebas, lo cual conlleva una importante reducción del tiempo empleado en la implementación de las pruebas de verificación. Por otro lado, facilita la separación de entornos (desarrollo y pruebas), ya que el algoritmo a depurar será tratado como una caja negra sin necesidad de que la persona que realice las pruebas deba saber cómo se encuentra implementado.

## 5.1. Requisitos iniciales

Para el diseño de la aplicación se han considerado una serie de requisitos que permitan conseguir de una manera óptima los objetivos planteados.

En primer lugar, los algoritmos criptográficos que se utilicen deberán ser adecuados para su correcta implementación en una FPGA. Para llevar a cabo dicha elección, se debe optar por la lectura de la literatura existente sobre implementaciones óptimas de diversos algoritmos en FPGAs (Ver apartado 2, Marco Teórico). Para posibilitar la realización de un Framework lo más general posible, dichos algoritmos deberían trabajar sobre bloques de entrada de longitud

fija y llevar a cabo todas las operaciones que lo diferencien de otros algoritmos en su interior y de una manera transparente al Framework (ya que serán tratados como cajas negras).

Debido a que la tecnología empleada para la realización de este Trabajo no posibilita un intercambio de señales de control entre la FPGA y el PC de una manera eficiente, cuanta menos lógica de control que suponga intercambio de información entre ambos dispositivos se realice, más fácil será la integración con el Framework. Esto no es aplicable a la transferencia de grandes volúmenes de datos de entrada y salida mediante los búferes que proporcionan las herramientas y que se encuentran destinados a tal propósito.

Por otro lado, en la Aplicación que se desarrolle, deberán estar claramente identificadas desde el diseño inicial las partes que se ejecutarán en cada dispositivo (en PC o en FPGA).

## 5.2. Tecnologías empleadas

En el apartado “4.1 Elección de la tecnología” se justificó la elección de las Tecnologías empleadas en el presente Trabajo Fin de Máster, en los siguientes apartados se describen aspectos importantes de las herramientas utilizadas.

En la aplicación implementada interactúan una serie de herramientas de distintos fabricantes que han decidido colaborar para desarrollar una solución conjunta que simplifique el proceso de diseño, desarrollo y depuración de proyectos basados en FPGA.

### 5.2.1. Herramientas para el desarrollo en PC

Mathworks es una compañía líder en el desarrollo de software para cálculo matemático para ingenieros y científicos. Cuenta con 30 años de experiencia en este campo y más de 3000 trabajadores distribuidos a lo largo de 15 países. De esta compañía se utilizará el software sobre el que se cimienta todo este trabajo, las herramientas destinadas al Diseño basado en modelos Matlab y Simulink.

Matlab es un entorno interactivo para el cálculo numérico basado en un lenguaje de alto nivel propio. Se trata de una potente herramienta para analizar datos, desarrollar algoritmos y crear

modelos o aplicaciones. Es ampliamente utilizada en cálculos, principalmente en sistemas de control automáticos y en procesado de señal, ya que proporciona un lenguaje muy sencillo para elaborar programas matemáticos. Matlab posee una gran cantidad de toolboxes (librerías) específicas para distintas áreas de trabajo como puede ser procesado de señal, procesado de imagen…

Simulink es un producto que se ejecuta sobre Matlab, que proporciona un entorno de interactivo y gráfico para el modelado, la simulación y análisis de sistemas dinámicos. Permite elaborar prototipos virtuales de una manera rápida para explorar los diseños a cualquier nivel de detalle. Simulink incluye una librería de bloques predefinidos para ser utilizados en la construcción de modelos gráficos de sistemas arrastrando y soltando (drag-and-drop) los distintos elementos que los compongan. Soporta sistemas lineales y no lineales, modelado basado en muestreo, en captura continua o un híbrido de ambos. Como se trata de un producto totalmente integrado con Matlab los datos pueden ser fácilmente compartidos entre ambos programas proporcionando a la aplicación desarrollada, toda la potencia computacional que proporcionan las funciones propias de Matlab. Los bloques que utiliza Simulink son proporcionados por los denominados “BlockSets”. Existen gran cantidad de “BlockSets” que contienen elementos especializados para determinadas tareas, como puede ser “Aerospace BlockSet”, que contiene bloques para modelar y simular sistemas de propulsión, aeronáuticos y aeroespaciales.

System Generator es una herramienta de modelado a nivel de sistema que facilita el diseño hardware sobre FPGA. Puede definirse como un BlockSet que extiende la funcionalidad de Simulink al ámbito del desarrollo hardware por medio de una serie de bloques predefinidos que son directamente sintetizables en una FPGA. Dicho de otro modo, permite de una manera rápida y visual desarrollar un diseño sobre Simulink que pueda ejecutarse posteriormente de una manera directa en una FPGA. Tras validar el diseño, simplemente apretando un botón de la herramienta se generará un bitstream de configuración de la FPGA con la misma funcionalidad que la simulada. Una de las ventajas que proporciona System Generator es el uso de las denominadas, “black boxes”. Una “black box” es un módulo realizado en lenguaje de descripción hardware (VHDL o Verilog) que es importado desde System Generator para generar un bloque que pueda ser utilizado desde el entorno gráfico. Esto permite que cualquier algoritmo criptográfico que se desarrolle pueda ser convertido en un bloque y sea utilizado por un módulo mayor para crear el modelo de un sistema en Simulink. O siendo más ambiciosos, esto permite desarrollar en lenguaje de descripción hardware cualquier bloque que se necesite en la aplicación.

Realmente con los bloques básicos que proporciona System Generator, se puede llegar a crear con más o menos esfuerzo cualquier diseño digital. Pero si se cuenta con personal experimentado que conozca cómo implementar y optimizar dicha lógica en la FPGA, el resultado final respecto a la generación automática de código permitirá obtener mejores resultados en cuanto a reducción de la lógica empleada y en aumento de la frecuencia máxima de funcionamiento. Por otro lado, si el personal no tiene grandes conocimientos en programación de dispositivos lógicos, con System Generator y haciendo uso de sus componentes básicos, se conseguirá obtener un producto de una manera muy rápida gracias a que todas las operaciones se pueden realizar mediante un interfaz gráfico.

System Generator es una herramienta integrada en el entorno de desarrollo ISE del fabricante de FPGAs Xilinx. Como es obvio, esta herramienta sólo es válida para FPGAs de este fabricante. Si se opta por usar dispositivos lógicos del otro gran fabricante (Altera), se deberá usar la herramienta equivalente llamada Altera DSP Builder.

La versión finalmente empleada de Matlab es R2007a y de System Generator la asociada al entorno ISE12.3.

### 5.2.2. Herramientas para el desarrollo en FPGA

System Generator convierte los bloques empleados en el modelado del diseño a una traducción directa en una serie de ficheros escritos en lenguaje de descripción hardware que describen de forma inequívoca el funcionamiento de dichos bloques. Posteriormente, System Generator cede el control del resto de procesos que se deben realizar hasta generar el bitstream de configuración de la FPGA al entorno ISE de Xilinx.

La primera tarea a realizar será la síntesis de los ficheros de descripción hardware previamente generados para obtener a la salida de este proceso una “netlist” que indique los componentes básicos a utilizar (multiplexores, sumadores, contadores, memorias…) y las interconexiones entre estos componentes.

Posteriormente, se realizará el proceso denominado en inglés “map” que realiza una traducción de esos componentes detectados, a una implementación en la lógica que utiliza la FPGA según la tecnología que se emplee para su construcción. En otras palabras, traduce la “netlist” al conjunto de elementos lógicos de nivel más bajo que se interconectarán para realizar las funciones indicadas. Una vez determinados los elementos de bajo nivel que se emplearán en la FPGA y cómo se interconectan, se procede a situarlos dentro de la lógica de

la FPGA de tal manera que se cumplan los requisitos de frecuencia máxima de funcionamiento impuestos (timing constraints). A este proceso se le denominada “Place & Route” y una vez finalizado, si se cumplen los requisitos de frecuencia de funcionamiento, se podrá generar finalmente el bitstream de configuración de FPGA. Este bitstream de configuración será el medio por el que se programará la funcionalidad de la FPGA cada vez que se encienda de nuevo el equipo (o tarjeta de evaluación) ya que se trata de un dispositivo volátil. Cada vez que se deseen realizar modificaciones en el proyecto, lógicamente también será necesario reconfigurar el dispositivo.

Para verificar el correcto funcionamiento de los algoritmos criptográficos antes de ser integrados en el sistema de co-simulación Hardware como cajas negras de System Generator, se realizarán unas breves simulaciones en PC. Otro objetivo de estas simulaciones es proporcionar tiempos de ejecución que sirvan para compararlos con el sistema propuesto de co-simulación HW durante pruebas exhaustivas de funcionamiento.
Para la realización de estas pruebas se utilizó el entorno de simulación Modelsim SE de Mentor Graphics y el simulador ISIM. ModelSim destaca entre sus competidores (como por ejemplo, Riviera de la compañía Aldec) por su velocidad, pero también tiene un coste bastante mayor.
Existen diversas configuraciones de esta herramienta que permiten reducir su precio de adquisición a expensas de reducir la velocidad de simulación y prescindir de algunas características avanzadas, como puede ser la capacidad multi-lenguaje. Por otro lado, existen versiones específicas para los principales fabricantes de FPGAs (Altera y Xilinx) que prescinden de algunas características de la versión SE y son bastante más lentas, pero tienen una importante reducción de precio.
El simulador ISIM se encuentra integrado en el entorno de desarrollo ISE de Xilinx, y su uso no requiere el pago de ninguna licencia extra.

Las versiones utilizadas en el presente trabajo de los productos comentados en este apartado son ISEv12.3 y ModelSim SE10.d.

### 5.2.3. Interfaz de comunicación PC-FPGA

Para poder realizar la comunicación entre el PC y la FPGA se debe hacer uso de ciertos drivers en el PC y se requiere implementar en la FPGA la lógica necesaria para la recepción y transmisión de los datos. Además, se deberán configurar los periféricos de la tarjeta de evaluación que se utilicen para transmitir los datos desde los conectores de la tarjeta en sí a los pines de entrada/salida de la FPGA (en el caso de la aplicación desarrollada en este Trabajo, todo lo necesario para establecer una comunicación vía Ethernet o JTAG). Estas tareas son realizadas de forma transparente al programador por la herramienta System Generator, siempre y cuando se utilice una de las tarjetas de evaluación soportadas por la herramienta. Existe la posibilidad de utilizar otras tarjetas (siempre y cuando se cumplan ciertos requisitos) realizando algunas tareas de configuración extra.
La mayoría de estas tarjetas ofrecen dos modos de transmisión: a través del JTAG o por medio de Ethernet. El primer modo es bastante sencillo de utilizar pero la frecuencia de funcionamiento del sistema (la del JTAG) es bastante baja, por lo que no obtenemos ninguna mejora frente a la simulación convencional. El segundo modo de funcionamiento es el que realmente proporcionará una frecuencia de funcionamiento real, ya que se podrá transmitir datos hasta una frecuencia de 1Gbps que serán procesados por la FPGA a la máxima frecuencia de funcionamiento para la cual haya sido diseñado el sistema. Para permitir este modo de funcionamiento de manera continua, se requerirá el uso de unas memorias dentro de la FPGA que son configuradas como “espejo” de unas memorias equivalentes en el PC. Este tipo de memorias se conocen como “Shared Memories” y su funcionamiento será explicado con mayor detalle en otros apartados.

## 5.3. Descripción de la aplicación

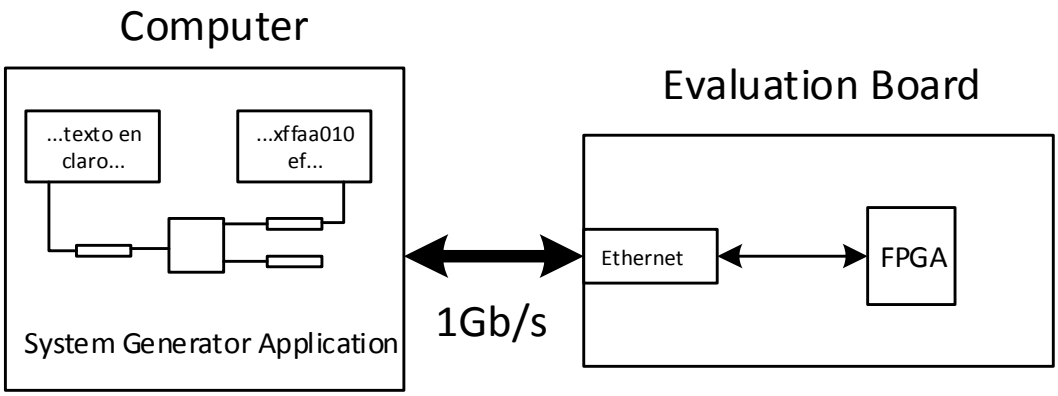
Será en este apartado donde se tratará la aplicación propiamente dicha, desglosándola en las diferentes partes que la componen y explicando la metodología de trabajo seguida.

La aplicación desarrollada consta de varias partes claramente diferenciadas y para su realización se han ido completando una serie de hitos intermedios que pueden ser utilizados según el criterio del usuario para cumplir distintos objetivos.

El primer gran hito planteado fue el diseño de una aplicación que se ejecutase íntegramente en el PC pero teniendo en mente que posteriormente diversas partes se ejecutarían en dispositivos distintos (PC y FPGA). Para asegurar este requisito, lo más sencillo es definir claramente las entradas y salidas del sistema ya que éstas serán ejecutadas en el PC. El algoritmo será siempre tratado como una “black box", de forma que para la aplicación se trate simplemente de otro bloque con una serie de entradas y salidas. En este primer hito el algoritmo se alimenta directamente de una función de Matlab que lee un fichero de texto de entrada y se lo proporciona al algoritmo en bloques de 32bits. Será cada algoritmo el que determinará mediante una señal de lectura cuántos bloques desea leer y en qué instantes. De manera equivalente, las salidas del algoritmo se proporcionan a una función de Matlab en bloques de 32 bits y se almacenan en un fichero de salida. La limitación del uso de buses de 32bits es impuesta por System Generator, ya que no permite instanciar memorias con datos de tamaño superior.

En un segundo hito, la lectura/escritura de datos entre el algoritmo y las funciones de Matlab no se realizará directamente, si no que se intercalarán entre las funciones y el algoritmo, FIFOS que almacenen los datos entre los distintos procesos. Para implementar dichas FIFOS se emplearán los componentes denominados por System Generator “Shared Memories”. Cada uno de estos componentes consiste realmente en una pareja de memorias. Cada memoria es un espejo de la otra para un dominio de trabajo distinto, pueden ser vistas como una especie de memoria doble puerto en la que cada puerto estará conectado a un dominio distinto. Un dominio será el encargado de escribir en dicha memoria y en el otro se leerán los datos. Esto posibilita que un dominio sea el entorno de trabajo en el PC y otro, la FPGA. En este hito de momento todo se ejecutará en el PC con vistas a depurar más fácilmente los fallos que puedan surgir.

En el tercer y último hito, se utilizará la división de entornos realizada en el anterior hito para asignar a cada dominio un dispositivo. Se procederá a generar un componente que sea ejecutable en la FPGA, se configurará la FPGA y se ejecutará la aplicación. La gestión de entrada/salida la realizará el PC y el algoritmo se ejecutará en la FPGA. Se realizará una primera versión ejecutando la aplicación a través del JTAG a una frecuencia reducida debido a su facilidad de configuración, y una vez verificado el funcionamiento sobre placa, se procederá a incrementar la velocidad de ejecución mediante un interfaz de comunicación de datos a través de ethernet Gigabit.




### 5.3.1. Modelos de referencia

En el presente apartado se describirán los modelos de referencia que se han utilizado para corroborar que los algoritmos criptográficos implementados funcionan correctamente. Inicialmente se consideró la implementación de los algoritmos escogidos en un lenguaje de alto nivel para verificar que la implementación de los algoritmos en lenguaje de descripción hardware se realizaba correctamente, pero viendo la gran cantidad de software libre existente se desestimó su implementación. Carece de sentido implementar un modelo de referencia sujeto a errores de codificación y tiempo de implementación y depuración asociado cuando existen soluciones libres que han sido ampliamente probadas.

Todos los pasos intermedios que han sido realizados durante la elaboración de la aplicación hasta su finalización, han sido verificados comprobando que las salidas obtenidas en cada proceso se correspondían con unas referencias iniciales. Dichas referencias fueron obtenidas del banco de pruebas implementado en lenguaje de descripción hardware que está incluido junto a cada algoritmo en los paquetes de libre distribución. En todo momento se ha asegurado que las modificaciones que se iban realizando no afectaban al correcto funcionamiento del algoritmo.

Existe código libre en lenguaje Matlab, como el encontrado en la referencia [26] (Simon, 2012), que podría ser incorporado en futuros trabajos sobre el framework desarrollado para comprobar los resultados de manera automática. También existen soluciones online que proporcionan el resumen de un texto o fichero de entrada para diferentes tipos de algoritmos [27] (hash.online-convert.com).

### 5.3.2. Algoritmos FPGA

La implementación de los algoritmos escogidos se puede realizar siguiendo las indicaciones que proporciona la descripción del apartado “4.3 Implementación de algoritmos/Funciones resumen”. Como ya se comentó, existen diversas implementaciones disponibles como código libre y se han realizado diversos trabajos sobre implementaciones óptimas de estos algoritmos, por lo que el diseño de una nueva implementación de estos algoritmos no es objetivo del presente Trabajo. Por ello, se describirán brevemente las implementaciones utilizadas y se indicará el link al sitio WEB de distribución de SW libre donde se obtuvo cada implementación. Debido a que dichas implementaciones no son directamente portables al framework de la aplicación desarrollada también se comentarán las modificaciones realizadas sobre cada implementación para permitir la integración.
Para cada algoritmo se explicarán aspectos claves de su implementación y cómo se ha realizado su verificación.

#### 5.3.2.1. SHA 256

##### 5.3.2.1.1. Implementación

Para la implementación de este algoritmo se ha utilizado un componente en lenguaje VHDL denominado “sha256_core”. Dicho componente tiene los siguientes puertos:

- clk_i: Reloj del sistema.
- ce_i: Enable del sistema. Requerido para generar un componente de tipo “black_box” en System Generator aunque no se use interiormente.
- new_msg_i: Reset del componente. Debe mantenerse a ‘1’ durante un ciclo antes del comienzo de la transformación de un nuevo mensaje.
- new_blk_i: Indica que se desea trabajar con un nuevo bloque de 512 bits.
- block0-3_i: Cuatro sub-bloques de 8 bits cada uno en los que se divide el bloque de 512 de bits que constituye el mensaje de entrada sobre el que aplicar la función

resumen. Por lo tanto, se requerirán 16 ciclos para capturar un bloque de 512 bits completo.

- a_o,b_o, c_o,d_o,e_o, f_o: Salidas que indican el valor contenido en los registros a-f y que servirán como señales de depuración en caso de algún fallo.
- valid_hash_o: Señal de validación de la salida “hash_o”, indicando que está disponible el resumen del bloque de 512 bits de entrada.
- hash_o: Resumen obtenido. Se trata de una señal 32bits, por lo que para proporcionar los 256bits que constituyen el resumen completo se requerirán 8 ciclos.

A grandes rasgos, la estructura de la implementación realizada de este algoritmo en la FPGA es la siguiente:

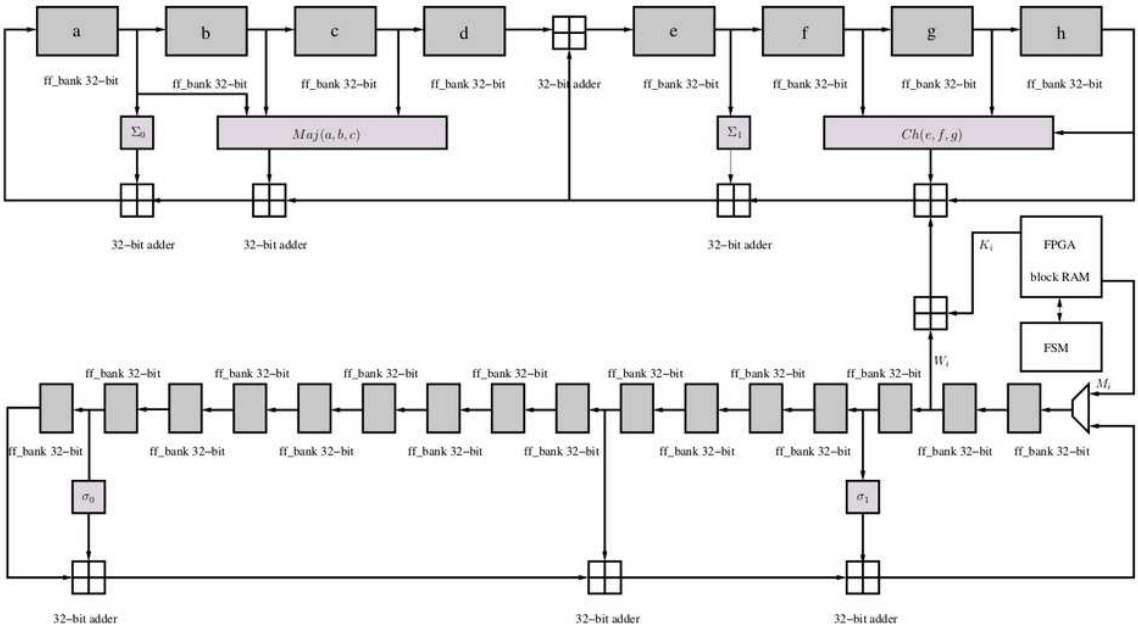
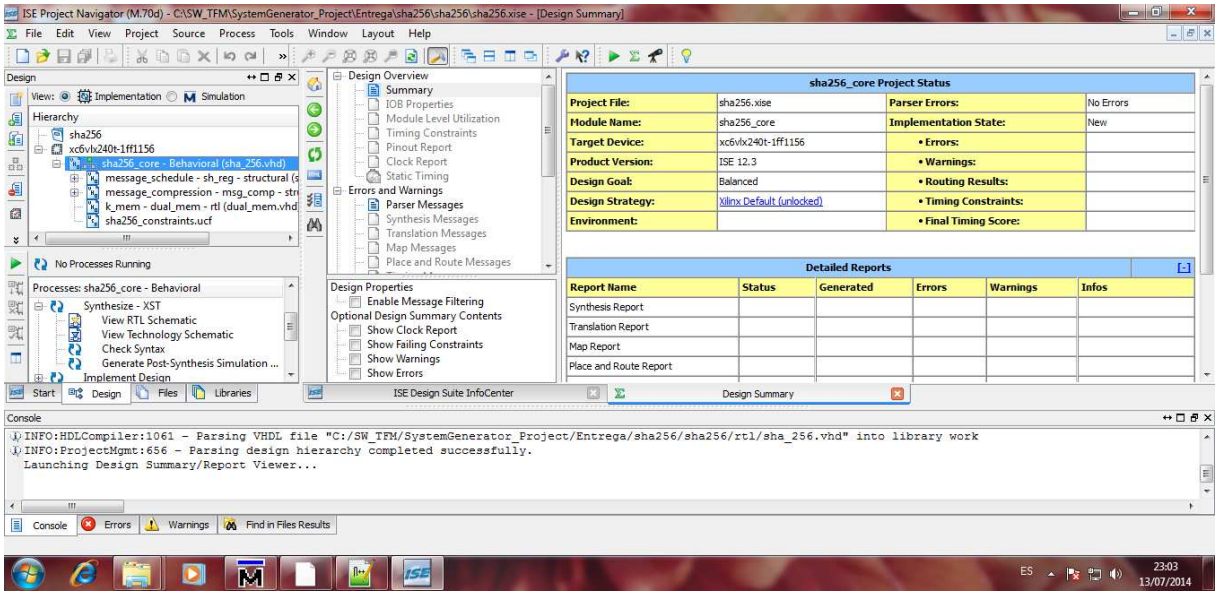



Figura extraída de De La Piedra, 2014 [14]

Las modificaciones realizadas sobre la versión original (De La Piedra, 2014) [14] están destinadas a conseguir un código que pueda ser implementado sobre una FPGA y que sea fácilmente integrable en el Framework que se ha realizado. La versión de partida aunque permitía comprobar su correcto funcionamiento mediante un sencillo test de prueba (fichero “tb_sha_256.vhd” en la carpeta “test_rtl”) no era sintetizable en una FPGA. Para poder generar a partir de dicho código un bitstream de configuración de una FPGA se creó un proyecto en ISE que contuviese todos los ficheros necesarios. Desde la aplicación que se ha realizado que podrá generar un bitstream de configuración y un proyecto ISE asociado directamente

mediante el uso de un botón, pero en este punto del desarrollo se deseaba probar el código de partida para prevenir futuros errores. Por otro lado, el proyecto autogenerado contiene una serie de ficheros que envuelven al algoritmo que dificultan el entendimiento de cualquier problema que pueda surgir por lo que es deseable realizar una primera verificación sobre un proyecto más sencillo.

El proyecto creado se ha llamado “sha256.xise” y presenta el siguiente aspecto tras su creación:




Con este proyecto se sintetizará e implementará el proyecto para comprobar la frecuencia máxima de funcionamiento del algoritmo, los recursos de lógica empleados y su consumo. Estos resultados nos servirán para estimar cómo afectan al diseño original las modificaciones que realicemos para permitir la co-simulación hardware así como los componentes añadidos por System Generator.

Una vez creado un entorno de trabajo adecuado, se procedió a realizar una primera síntesis en la que se observó que la memoria que contiene las constantes Ki no estaba siendo correctamente sintetizada. Analizando cómo dicha memoria había sido creada, se observó que se había creado como una memoria RAM que era inicializada desde el entorno de simulación asignándole un fichero de configuración. Para solucionar este problema, se implementó una memoria ROM con todos los valores de las constantes K y se observó su correcta síntesis. El fichero afectado por tal modificación es “dual_mem.vhd”. A parte de esta modificación, se corrigieron algunos warnings que proporcionó la herramienta de síntesis XST (integrada en ISE) tras su ejecución.

Por otro lado, la herramienta System Generator no contempla la posibilidad de trabajar con señales cuyo valor sea indeterminado (ni '0', ni '1') en sus bloques. En todas las salidas de la "black box" que represente al algoritmo se deberá impedir entregar valores indeterminados aunque sólo se entreguen dichos valores durante el período de inicialización. Para evitar esto se tuvo que realizar un repaso de todo el código inicializando todas las señales del interior del algoritmo.

La definición de la entidad del componente (lo que equivaldría al prototipo de una función en C, o al interfaz) fue modificada para incluir un clock enable, que aunque no sea luego utilizado en el interior del componente es obligatorio incluirlo en la definición tal y como se especifica en "System Generator for DSP Reference Guide (v12.3)" (Xilinx,2012) [23] en el apartado "Black Box/Requirements on HDL for Black Boxes".

Para integrar el componente en el Framework se hicieron diversas modificaciones:

- En lugar de recibir 16 señales de entrada de 32bits (que constituyen el bloque de 512 bits que utiliza SHA256) se recibirán 4 entradas de 8bits. El proceso original esperaba tener 512bits a su entrada cuando recibía un pulso en la señal "rst", y al ciclo siguiente de recibir tal pulso comenzaba a trabajar con el bloque si la señal "gen_hash" estaba a '1'. En la implementación actual, un pulso en la señal "new_msg_i" o "new_blk_i" origina que se lance un proceso de lectura de una memoria externa de bloques de 32 bits (en realidad, 4 bloques de 8 bits) hasta completar los 512 bits. Una vez almacenado el bloque de 512 bits se genera internamente una señal de comienzo del proceso.
- En la versión original existía una señal de habilitación del proceso llamada "gen_hash". Esta señal era proporcionada como entrada y debía mantenerse activa durante todo el proceso. Si tras finalizar el procesado de un bloque de 512 bits, se activaba dicha señal sin introducir previamente un pulso en "rst", se suponía que se trataba de otro bloque de 512 bits perteneciente al mismo mensaje por lo que el nuevo resumen generado contendría a éste y a los anteriores bloques. En la versión actual, la generación de la señal de habilitación se realiza internamente a partir de las señales "new_msg_i" o "new_blk_i". Si se desea comenzar a procesar un nuevo mensaje se debe activar la señal "new_msg_i", y si se desean procesar varios bloques de un mismo mensaje se debe introducir un pulso "new_blk_i" por cada bloque de 512 bits.
- También se modifica la forma de entregar los datos del resumen obtenido como resultado. En la versión original se entregan directamente los 256bits que lo

conforman. En la versión actual el resumen se entrega en bloques de 32 bits, afectando obviamente también a la señal de validación del resultado que durará más ciclos (8 ciclos para SHA256).

Debido a ciertos problemas que se tuvieron inicialmente en el desarrollo del Framework relacionados con la ejecución incorrecta del código del algoritmo al ejecutarse como “black box” dentro del entorno System Generator/Simulink, se tuvieron que realizar algunas modificaciones para obtener señales de depuración. Fue en este punto donde se observó la principal debilidad de la herramienta System Generator, ya que si el código exhibe un comportamiento distinto entre la simulación realizada con ModelSim o ISIM y la realizada con System Generator, desde Simulink no se obtiene ninguna visibilidad de lo que ocurre dentro de la “black box”. Para obtener dicha visibilidad se requiere modificar el código de la “black box” para entregar como salidas toda aquella señal que se quiera observar. Por este motivo, y sólo para este algoritmo (al ser el primero en utilizarse), se modificó bastante código para deshacer algunas funciones y obtener señales de depuración que posteriormente se entregaron como salidas.

Para poder estimar la frecuencia de funcionamiento máxima del algoritmo se creó un fichero de restricciones (constraints) llamado “sha256_constraints.ucf”, en el que se indica a la herramienta el período de reloj deseado. ISE intentará en la fase de “Place&Route” situar los componentes y las interconexiones del diseño de tal forma que se cumplan los requerimientos impuestos. Si no es capaz de conseguirlo, indicará un error y los paths dónde se han producido los errores.

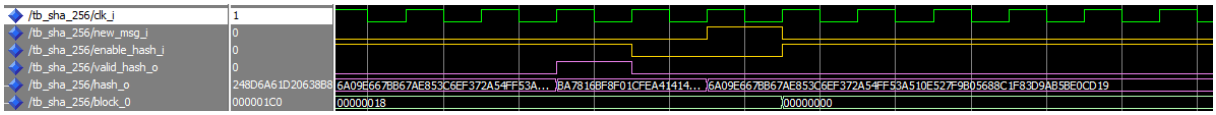
Para que ISE sea capaz de determinar la frecuencia de funcionamiento máxima de un diseño toda la lógica debe encontrarse situada entre flip-flops, para ello se ha creado un fichero que llama al algoritmo añadiendo previamente a todas las entradas flip-flops (sha256_wrapper.vhd).

#### 5.3.2.1.2. Verificación

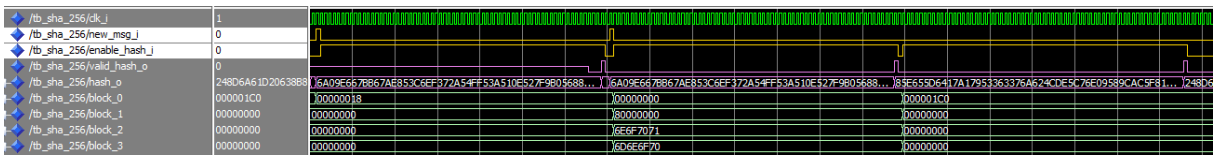
En este apartado se describirán varias simulaciones realizadas con la herramienta ModelSim SE de Mentor Graphics e ISIM de Xilinx, con el objetivo de verificar el funcionamiento de los bloques implementados en lenguajes de descripción hardware. Otro objetivo que se busca en la realización de estas simulaciones es obtener unos tiempos de referencia para poder comparar con los resultados obtenidos por la co-simulación HW.

Para comprobar el correcto funcionamiento del algoritmo SHA256 se ha utilizado un banco de pruebas (test bench) muy sencillo que se proporciona en el paquete de distribución original. En este test se introducen varios bloques de texto de 512 bits cuyo resumen es conocido y tras 0.66 us, se compara el resultado esperado con el resultado obtenido de la simulación.

Primero se realizan dos pruebas, realizando el resumen de dos textos independientes de 512 bits. Para independizarlos, entre la finalización del primer bloque (indicado por la señal "valid_hash_o") y el comienzo del segundo, se da el valor '0' a la señal "enable_hash_i" (originalmente "gen_hash") y se genera durante un ciclo la señal "new_msg_i" (originalmente rst). En la siguiente gráfica vemos con más detalle el instante en el que se cambia de un mensaje de entrada a otro:

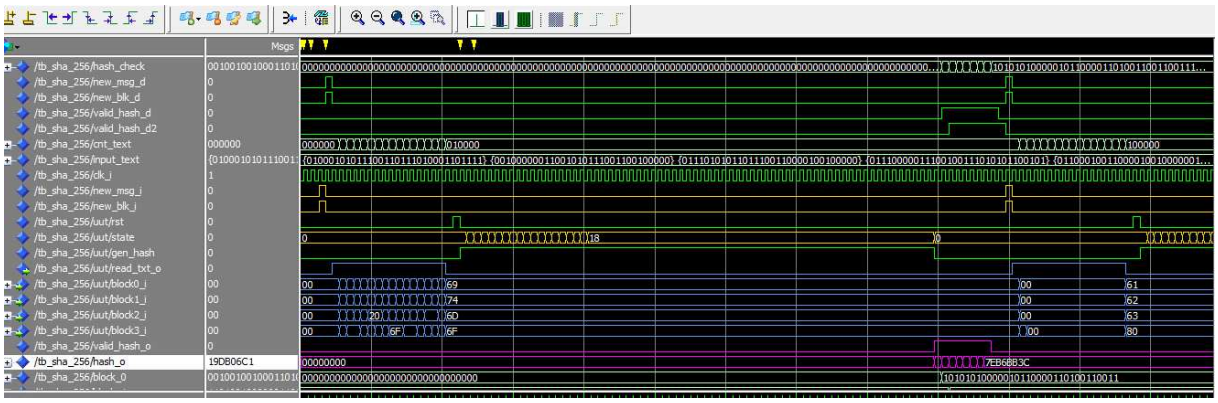



A continuación, se concatenan dos bloques de 512bits. Para ello, se introducen los primeros 512 bits del mensaje, se crea un pulso "new_msg_i" y a continuación se habilita la señal "enable_hash_i". Una vez finalizado el primer bloque de este segundo mensaje (indicado por "valid_hash_o"), se baja la señal "enable_hash_i", se introduce como entrada (dividido en 16 sub-bloques de 32 bits) el nuevo bloque de 512 bits; pero esta vez sin generar una señal "new_msg_i". De esta forma, el resumen del mensaje de 1024 bits (dividido en los dos bloques comentados) se obtiene en el instante que se genera el segundo "valid_hash_o".



En caso de que la salida del componente sea distinta del resultado esperado se indica un error mediante una cadena de texto por consola que se ha introducido en el banco de pruebas por medio de un aserto.

Tras realizar las modificaciones en el código del algoritmo para adaptarlo al Framework genérico se decidió realizar de nuevo estas pruebas para verificar que no se hubiese producido ningún fallo en la codificación. Para tal propósito hubo que adaptar el test bench a las modificaciones realizadas en el algoritmo. Por otro lado, se decidió adaptar dicho banco de pruebas para que simulase el comportamiento de la entrada/salida de datos que se utilizará en el Framework sobre System Generator.
Para conseguir esto, se implementó la lectura de los datos de entrada mediante el uso de una memoria ROM que contuviese el texto a procesar y se modificó la gestión de la salida, de manera que el resumen resultado del procesado se obtuviese en varios bloques de 32bits. En la siguiente figura podemos observar el resultado de la simulación tras las modificaciones realizadas:




El funcionamiento del “test bench” sería el siguiente:

- Se introduce una señal de comienzo por medio de un pulso de “new_msg_i”. A partir de este momento, el algoritmo empieza a funcionar de manera autónoma.
- El algoritmo genera una señal de lectura (read_txt_o) para leer los datos de entrada de la memoria que los contiene en conjuntos de 32bits (en realidad 4bytes: block0_i, block1_i, block2_i y block3_i). Como se trata de SHA256 se realizarán 16 lecturas hasta completar los 512 bits que conforman el bloque de entrada.
- Una vez finalizada dicha lectura, internamente se genera una señal de comienzo de nuevo mensaje (rst) y se activa la señal que habilita el proceso (gen_hash). Estas

señales en la versión original se introducían por puertos de entrada. Una vez finalizado el procesado del bloque, se desactiva la señal “gen_hash” y se entrega el resultado por la salida “hash_o”. Se indica que el resultado es válido por medio de la señal de validación “valid_hash_o”. El resultado se entrega en 8 bloques de 32bits (256bits).

- El algoritmo se queda a la espera de un nuevo bloque de 512bits. Para indicar que se desea procesar un nuevo bloque del mismo mensaje, se usaría la señal “new_blk_i” y para indicar un nuevo bloque de un mensaje distinto se usaría la señal “new_msg_i”.

Dentro de la memoria ROM que contiene los datos de entrada, se han introducido los valores en hexadecimal de un fichero de texto que posteriormente será utilizado para verificar el funcionamiento del Framework en System Generator. De esta manera, comprobamos que el código VHDL del que partimos realiza correctamente la operación sobre estos datos.

Si se analizan las simulaciones realizadas, se observa que se tardan 94ciclos en obtener el resumen de un bloque de 512bits.
Se realizó una segunda versión del banco de pruebas que consiste en un bucle infinito en el que la entrada son bloques de ceros. Esta prueba se ha implementado en el fichero “tb_sha_256_loop.vhd” y su objetivo se centra en medir tiempos de simulación largos, ya que para este propósito es irrelevante el dato de entrada al algoritmo.

#### 5.3.2.2. SHA1

##### 5.3.2.2.1. Implementación

Para la implementación de este algoritmo se ha utilizado un componente en lenguaje VHDL denominado “sha1.vhd”. Dicho componente tiene los siguientes puertos:

- m: Sub-bloque de 32bits en los que se divide el bloque de 512 bits de entrada.
- init: Señal que indica el comienzo de un nuevo mensaje.

- ld: Señal que indica que se están introduciendo sub-bloques de 32bits por la entrada “m”. Por lo tanto, debe estar activa durante 16 ciclos hasta completar el bloque de 512bits a procesar.
- h: Resumen obtenido. Se trata de una señal 32bits, por lo que para proporcionar los 160bits que constituyen el resumen completo se requerirán 5 ciclos.
- v: Señal de validación de la salida “hash”, indicando que está disponible el resumen del bloque de 512 bits de entrada. Se trata de un pulso de un ciclo que se activa un ciclo antes de que realmente esté disponible el resumen.
- clk: Reloj del sistema.
- rst: Reset del componente

Como se puede comprobar estos puertos son distintos de los utilizados por el componente “sha256_core”. Para integrarlo con el Framework desarrollado para el algoritmo “sha256” y no tener que realizar ninguna modificación, se implementará una envoltura (wrapper) para el componente “sha1” en el fichero denominado “sha160_wrapper.vhd”.

Dicho fichero tiene como puertos de entrada/salida los mismos que el componente “sha256_core”. La señal “init” y “rst” se han unido al puerto de entrada “new_msg_i”, para indicar que se va a procesar un nuevo mensaje. A partir de las señales “new_msg_i” y “new_blk_i” se crea la señal de lectura a la memoria que contiene el texto a procesar (read_txt_o) que debe durar 16ciclos. El contador que controla la duración de esta señal da valor al puerto de salida “read_addrs_o”. Por otro lado, para generar la señal “valid_hash_o” se crea una señal que se mantiene a ‘1’ durante 5 ciclos indicando que es válido el resumen obtenido por el puerto “hash_o” (unido a la señal “h”).

Como se puede apreciar en el fichero “sha160_wrapper.vhd” se requieren unas conversiones de tipos entre “bit_vector” y “std_logic_vector” y “bit” y “std_logic”.

Una vez realizadas estas modificaciones obtenemos un componente totalmente intercambiable con “sha256_core” como se demostrará en otros apartados.

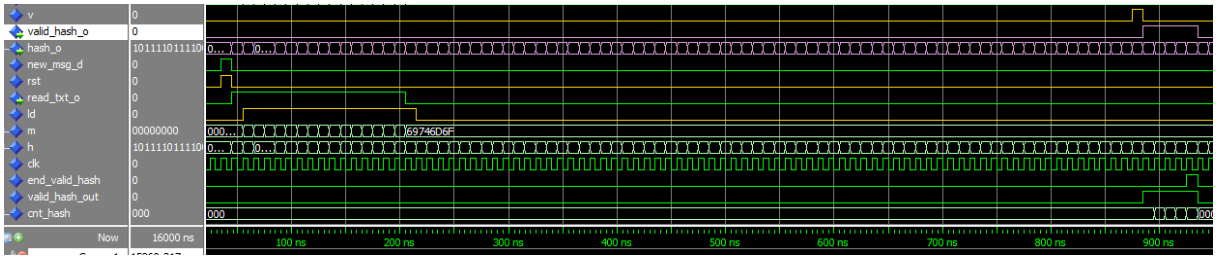
Al igual que se hizo para el caso del algoritmo anterior se creará un proyecto de ISE asociado para realizar la implementación en la FPGA y así obtener unas primeras medidas en cuanto a ocupación lógica, velocidad y consumo. El proyecto se ha creado bajo el nombre de “sha1.xise”.

#### 5.3.2.2.2. Verificación

En el paquete original de distribución de este algoritmo (Arif Endro, 2010) [13] no se proporciona un banco de pruebas para verificar el funcionamiento del algoritmo. Por ello, hubo que implementar uno para poder comprobar que las modificaciones se habían realizado correctamente.

El banco de pruebas realizado (tb_sha_160.vhd) está basado en el desarrollado para el algoritmo sha256 (tb_sha_256.vhd). Realmente lo único que se ha modificado es el componente que se prueba sustituyendo el componente “sha256_core” por “sha160_wrapper” y cambiando los valores con los que se compara la salida del algoritmo.

En la siguiente gráfica podemos ver el resultado de la simulación realizada con ModelSim SE:




En la anterior figura se puede visualizar el comportamiento descrito en el apartado anterior. A partir de la señal “new_msg_i” se realiza la lectura de la memoria (por medio de “read_txt_o”) y tras finalizar el procesado, a partir de la señal “v” se genera la señal de validación “valid_hash_o” de 5ciclos, indicando que por la salida “hash_o” se está obteniendo el resumen en sub-bloques de 32 bits. El procesado de un bloque de 512 bits se realiza en 90 ciclos.

#### 5.3.2.3. SHA 512

##### 5.3.2.3.1. Implementación

Como el código de este algoritmo se ha obtenido del mismo paquete de distribución que el algoritmo “sha160” los puertos de entrada/salida son prácticamente los mismos, aunque se han modificado los tamaños de los sub-bloques para ser de 64 bits en lugar de 32. A continuación se describen los puertos del componente “sha512”:

- m: Sub-bloque de 64bits en los que se divide el bloque de 1024 bits de entrada.
- init: Señal que indica el comienzo de un nuevo mensaje.
- ld: Señal que indica que se están introduciendo sub-bloques de 64bits por la entrada “m”. Por lo tanto, debe estar activa durante 16 ciclos hasta completar el bloque de 1024bits a procesar.
- md: Resumen obtenido. Se trata de una señal 64bits, por lo que para proporcionar los 512bits que constituyen el resumen completo se requerirán 8 ciclos.
- v: Señal de validación de la salida “hash”, indicando que está disponible el resumen del bloque de 1024 bits de entrada. Se trata de un pulso de un ciclo que se activa un ciclo antes de que realmente esté disponible el resumen.
- clk: Reloj del sistema.
- rst: Reset del componente

Debido a que la aplicación desarrollada trabaja con sub-bloques de 32 bits y este algoritmo está implementado usando bloques de 64bits habrá que realizar una serie de modificaciones para permitir la integración.

Las modificaciones se han realizado en el componente “sha512_wrapper”, en cuyo interior se adaptan los puertos de entrada/salida trabajando con sub-bloques de 32bits a los puertos del componente “sha512” preparado para trabajar a nivel de 64bits.

Para la entrada se introducen los 32bits en un puerto de escritura de una memoria interna, cuando se ha escrito una posición más de la mitad de la memoria se procede a leer por medio de dos puertos las posiciones previamente escritas. De esta forma, se entregarán dos datos de 32bits en cada ciclo para conformar los 64bits que espera el componente “sha512” como

entrada. Obviamente, se modifican las señales de control para indicar al componente que su entrada es válida cuando estén disponibles los datos de la memoria.

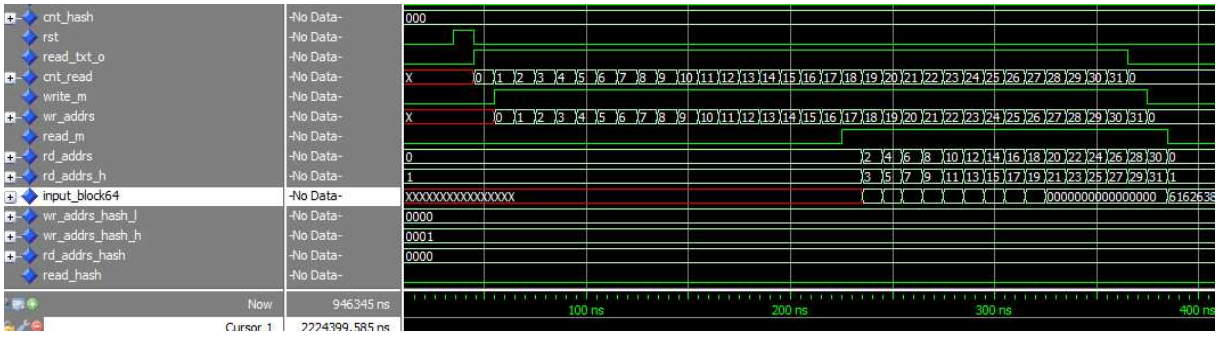
Para la adaptación de la salida se dividen los 64bits que entrega como resumen el algoritmo en dos buses de 32bits que se escriben en una memoria usando dos puertos de entrada en posiciones consecutivas. Una vez escrita la primera posición de memoria, ya se puede comenzar la lectura de la memoria ya que se irán leyendo datos de 32bits mientras se escriben datos de 64bits (dos datos de 32 en realidad), por lo que la lectura nunca alcanzará a la escritura.

En el siguiente apartado se muestra el resultado de una simulación donde se visualiza las señales comentadas.

#### 5.3.2.3.2. Verificación

Al igual que en el caso de “sha1” en el paquete de distribución original (Arif Endro, 2010) [13] no se incluía ningún banco de pruebas por lo que se decidió crear uno para validar las modificaciones. Para tal propósito se usó “tb_sha_512.vhd” que realmente es el mismo fichero que se usó para validar los anteriores algoritmos pero sustituyendo el componente a emplear (en este caso se usa “sha512_wrapper”), modificando los valores de comparación de salida del algoritmo y ampliando el número de bloques que componen la entrada y salida del algoritmo.

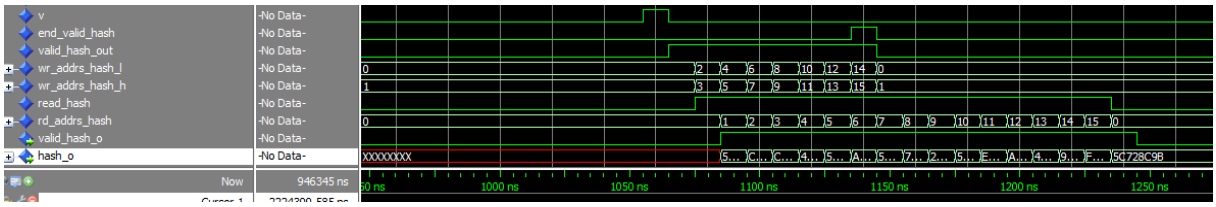
Las siguientes figuras muestran el resultado de la simulación de este algoritmo empleando el entorno de Simulación ModelSim SE:




En la simulación se puede observar el proceso de lectura comentado en el apartado anterior. Se visualiza cómo se genera la señal de lectura “read_txt_o” para leer 32 bloques de 32bits

(1024bits), y cómo a partir de ella se crea la señal de escritura en una memoria (usando las direcciones wr_addrs). Tras rellenar una posición más de la mitad de la memoria se lanza el proceso de lectura, en el que se leerán dos datos de 32bits por ciclo de dicha memoria para proporcionar al algoritmo los 64bits que espera (input_block_64).

En la siguiente gráfica se mostrará el proceso de adaptación de 64bits a 32 realizado a la salida:




Como se puede observar tras la indicación del algoritmo de finalización del proceso (señal “v”) se comienzan a escribir los datos que entrega el algoritmo en dos posiciones contiguas de una memoria (wr_addrs_hash_l y wr_addrs_hash_h) dividiendo el dato de 64bits en dos buses de 32. Una vez que se ha escrito la primera pareja de valores se comienza a leer el contenido de la memoria, entregando por la salida “hash_o” bloques de 32bits con el resultado del resumen realizado durante 16ciclos (32x16 = 512bits).

El procesado de un bloque de 1024 bits se realiza en 121ciclos de reloj.

### 5.3.3. Framework de intercomunicación

En los frameworks de intercomunicación que hacen uso de las funciones resumen la estructura que se ha seguido en los tres casos es la misma:

- Desde el entorno Simulink, se utiliza una función M que haciendo uso del lenguaje Matlab lee de un fichero el mensaje sobre el que se aplicará la función resumen. Dicho mensaje es proporcionado a una memoria FIFO de System Generator en bloques de 32 bits. El número de bloques de 32bits a proporcionar dependerá de la longitud del bloque que espera el algoritmo para comenzar el procesado (512bits en SHA256 y SHA1 y 1024 bits en SHA512).
- Los bloques se transmiten desde el PC a la FPGA por medio de los componentes de tipo “Shared Memory” proporcionados por el blockset de System Generator. Cada

bloque se transmitirá en un ciclo de ejecución de Simulink. Dichas memorias, son una forma simple de intercomunicar el dominio virtual de datos que se ejecuta sobre el PC con el dominio HW que se ejecuta en la FPGA. La función M almacenará el mensaje en la parte de escritura del primer conjunto de memorias, denominadas “To Fifo SharedFifoTxt”. La FPGA leerá los datos de la implementación “espejo” que se realiza en la FPGA, denominada “From Fifo SharedFifoTxt”.

- Una vez que se ha llenado la memoria, se debe indicar a la FPGA que dispone de nuevos datos para realizar el resumen, para este propósito se utilizará una señal de comienzo que se transmite a la FPGA por medio de las parejas “SharedRegisterStart” y “SharedRegisterStartBlock”. Si es un nuevo mensaje se indicará por medio “SharedRegisterStart” y si es un nuevo bloque dentro del mismo mensaje se indicará por medio de “SharedRegisterStartBlock”.
- En la FPGA una vez se ha finalizado la ejecución del algoritmo criptográfico en cuestión, se almacena el resumen generado. La señal “valid_hash_o” se utilizará para habilitar la escritura en el módulo “To FIFO SharedFifoHash”. Dicho módulo se instancia en la FPGA para comunicar los datos de salida a la memoria correspondiente del PC llamada “From FIFO SharedFifoHash”.
- Como se comentó anteriormente, cada bloque de N-bits (512 en SHA256 y SHA1 o 1024 en SHA512) se transmite en un ciclo de ejecución de Simulink. Por lo tanto, las memorias que se utilizan para transferir datos de un dominio a otro (PC-FPGA) serán configuradas del tamaño de ese bloque. Por otro lado, se requerirán N/32 ciclos de Simulink para llenar dichas memorias ya que el bus de datos de escritura es de 32bits.
- Desde el PC, haciendo uso de “From FIFO SharedFifoHash” se leerá el resumen obtenido. En la parte de recepción de datos del PC, como se recibirá un resumen por bloque, se contabilizarán los resúmenes de los bloques que se van generando hasta alcanzar el número total de bloques de los que está compuesto el mensaje. Una vez obtenido el último resumen se comparará con el resumen que se obtuvo previamente de un código de referencia que se ejecuta sobre PC. Cada resumen obtenido se leerá en N/2/32 ciclos, ya que el resumen es de la mitad del tamaño del bloque de entrada de N bits y el bus de lectura de la FIFO es de 32bits.
- Para utilizar el mismo Framework para los distintos algoritmos SHA implementados, basta con cambiar el bloque de co-simulación HW por el que corresponda y modificar la siguiente variable desde Matlab:

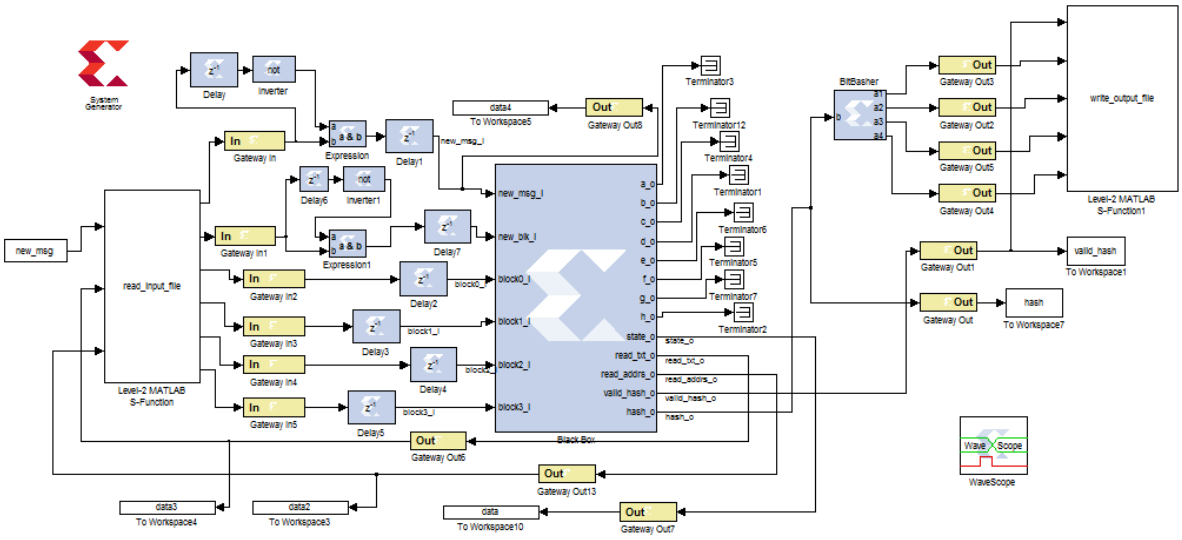
| | SHA1 | SHA256 | SHA512 |
|---|---|---|---|
| block_size | 512 | 512 | 1024 |

Para llegar a obtener esta aplicación se han ido alcanzado una serie de hitos que se comentarán en los siguientes apartados.

#### 5.3.3.1. Hito 1

##### 5.3.3.1.1.1. Implementación

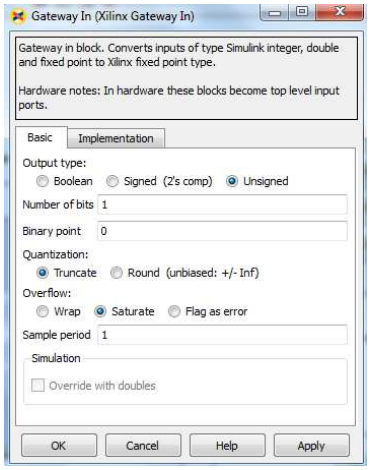
En este hito la aplicación será íntegramente ejecutada en el PC para verificar que la metodología de trabajo se está realizando adecuadamente antes de utilizar la FPGA. En la siguiente figura se puede ver la aplicación desarrollada para el algoritmo SHA256.




La entrada/salida de datos del algoritmo es gestionada por dos bloques de Simulink denominados Level-2 Matlab S-Function. Este tipo de bloques permiten instanciar en su interior código Matlab para realizar determinadas tareas. En la aplicación desarrollada, la función “read_input_file” es la encargada de leer de un fichero de texto los datos de entrada que serán proporcionados al algoritmo en bloques de 32 bits según se vayan pidiendo por medio de la señal “read_txt_o” y de la posición en memoria que indique “read_addrs_o”. Por

otro lado, la función “write_output_file” es la encargada de recibir los datos de salida del algoritmo y almacenarlos en un fichero.

Todos los bloques de color azul que aparecen en el esquema son componentes del BlockSet de Simulink, System Generator. Como se puede observar en la figura, el entorno Simulink se encuentra separado del entorno System Generator por unas cajas de color amarillo. Estas cajas se denominan “gateways”. Los “gateways” son utilizados para adaptar el tipo de datos de Simulink al tipo de datos utilizado por System Generator y para delimitar los componentes que son directamente sintetizables en FPGA (previa conversión a lenguaje de descripción hardware VHDL o Verilog). Los gateways de salida heredan el tipo de datos que entregan de su entrada, pero los gateways de entrada se configuran para indicar el tipo de dato que entregarán a su salida (tipo de representación y número de bits, realizar redondeo o truncado, tasa de muestreo…). A continuación, se puede ver el menú de configuración de un gateway de entrada:




Las cajas que contienen el nombre “To Workspace” son utilizadas para rellenar arrays que luego podrán ser visualizados desde Matlab como medida de depuración. Cada posición del array se corresponde con un dato capturado en un instante de tiempo concreto (en cada ciclo de Simulink). Por el contrario, la caja “start_msg” es del tipo “From Workspace” y nos permitirá introducir datos desde Matlab a Simulink. Dichos datos deben ser arrays de dos dimensiones. Una dimensión indica instantes de tiempo y la otra, el valor del dato para dicho instante.

El bloque denominado “Black Box” contiene al algoritmo propiamente dicho. Para cada algoritmo que se vaya a implementar en la FPGA se debe generar un modelo en System Generator equivalente. En el bloque “Black Box” hay que indicar para cada caso, el componente al que representa asignándole el fichero TOP (el de más alto nivel que incluye

al resto de componentes en lenguaje de descripción hardware). En este caso, el fichero a indicar sería sha256.vhd. Los puertos del bloque se crearán automáticamente tras indicar el fichero asociado. Es importante destacar que el “clock enable” (señal de habilitación de reloj) del componente y el reloj, deben seguir un estilo de nomenclatura apropiado para que System Generator pueda interpretar la funcionalidad de esos puertos y poder asociarles automáticamente el reloj y el clock enable del sistema sin intervención del usuario (Xilinx, 2012) [23]. Para cada “Black Box” generada se crea automáticamente un fichero de configuración *_config.m que contiene los datos que describen el interfaz (entrada/salida/genéricos) y su implementación para System Generator. Este fichero es una M-function (o P-function) escrita en lenguaje Matlab. Dicha función contiene:

- Una especificación de la entidad a la que está asociada la black box.
- El lenguaje que utiliza (VHDL o Verilog).
- Descripción de los puertos y si son dinámicos o estáticos. Puertos dinámicos pueden derivar su tamaño y tipo en función de la señal que conducen.
- Se deben especificar otros ficheros que sean utilizados por la entidad.
- Definición de relojes y “clocks enables”.
- Se debe declarar si existen realimentaciones combinacionales. Es decir, paths que no incluyen un flip-flop en su realimentación.

En el caso que nos ocupa, únicamente será necesario modificar el fichero que se generó automáticamente para incluir los ficheros auxiliares que son utilizados por sha256.vhd. A continuación, se muestra la parte del código del fichero de configuración que ha sido modificado:

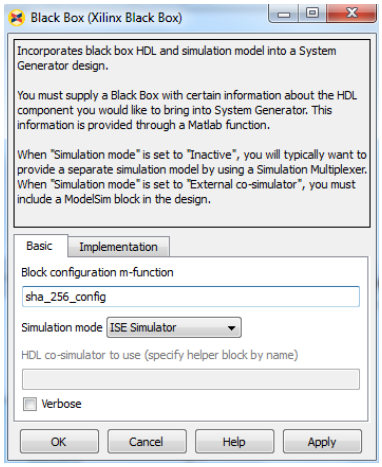
```
% Add addtional source files as needed.
%  |-------------
%  | Add files in the order in which they should be compiled.
%  | If two files "a.vhd" and "b.vhd" contain the entities
%  | entity_a and entity_b, and entity_a contains a
%  | component of type entity_b, the correct sequence of
%  | addFile() calls would be:
%  |    this_block.addFile('b.vhd');
%  |    this_block.addFile('a.vhd');
%  |-------------

%    this_block.addFile('');
%    this_block.addFile('');
this_block.addFile('..\..\rtl\sha_256.vhd');
this_block.addFile('..\..\rtl\dual_mem.vhd');
this_block.addFile('..\..\rtl\ff_bank.vhd');
this_block.addFile('..\..\rtl\msg_comp.vhd');
this_block.addFile('..\..\rtl\sh_reg.vhd');

return;
```

Una vez creada la Black Box es muy importante asignar al bloque un modo de simulación para que System Generator sepa cómo funciona este modelo. En nuestro caso, para simplificar las acciones a realizar seleccionamos el Simulador de ISE que viene en la

instalación de este producto (ISIM). De esta forma, System Generator podrá modelar el comportamiento del código que conforma el Black Box:




Las funciones “read_input_file” y “write_output_file” son también funciones Matlab contenidas en los ficheros read_input_file.m y write_output_file.m. Dichos ficheros han sido codificados íntegramente para esta aplicación siguiendo las indicaciones que aparecen en la ayuda online de Mathworks en el apartado “Implement S-Functions” [24] y utilizando como modelo la función “msfuntmpl_basic.m” que puede editarse desde Matlab con el siguiente comando:

```
edit('msfuntmpl_basic.m');
```

Básicamente este tipo de ficheros presentan una serie de funciones que se describirán brevemente a continuación:

- Setup. Función donde se indica el número de puertos de entrada y salida del bloque y de parámetros de entrada (los parámetros no se introducen como puertos). Se indica para cada puerto:
    - Dimensiones. Ya que pueden tratarse como arrays.
    - Tipo de Dato:

| Data Type | Value |
|---|---|
| 'inherited' | -1 |
| 'double' | 0 |
| 'single' | 1 |
| 'int8' | 2 |
| 'uint8' | 3 |
| 'int16' | 4 |
| 'uint16' | 5 |
| 'int32' | 6 |
| 'uint32' | 7 |
| 'boolean' or fixed-point data types | 8 |

    - Complexity. Real o complejo.
    - DirectFeedthrough. Indica que la salida es controlada directamente por un valor de un puerto de entrada. (Más información en la ayuda online de Mathworks[25])
- DoPostProSetup. Función donde se declaran las estructuras que serán utilizadas como auxiliares (block.Dwork) y que conservarán su valor durante la ejecución. Son una especie de variables estáticas.
- Start. Contiene el código que se ejecutará en el primer ciclo de la ejecución.
- Outputs. Contiene el código que se emplea para dar valor a los puertos de salida del bloque. Es donde realmente reside el código que determina el funcionamiento de la función.
- Terminate. Función que se ejecuta al finalizar la simulación.

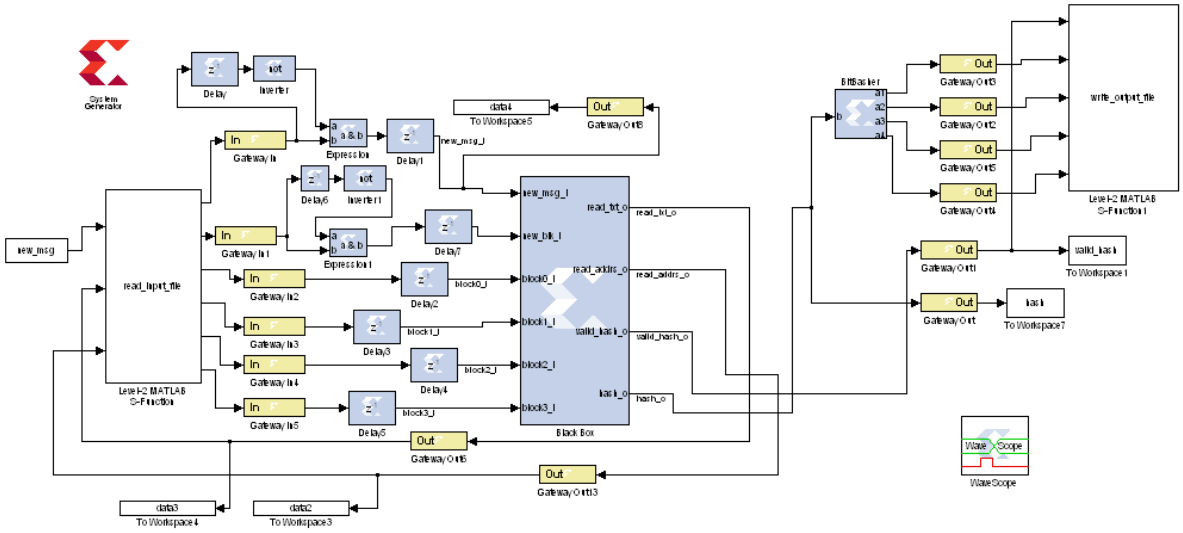
El código implementado para realizar la gestión de entrada/salida se encuentra debidamente comentado como Anexo en el apartado de “Funciones Matlab”.
Como la función de entrada “read_input_file” entrega el dato en el mismo ciclo que se active “read_txt_o”, se introducen unos registros (Delay2-5) en los datos de entrada al algoritmo, de forma que se simule el funcionamiento de una memoria síncrona.
Tanto para la señal “new_msg_i” como para la señal “new_block_i”, se incorpora una circuitería para detectar un flanco de subida de las señales que proporciona la función “read_input_file”.

Para utilizar este mismo modelo para el algoritmo SHA1 basta con sustituir la “black box” por otra que utilice el fichero “sha160_wrapper.vhd”. Al añadir este componente se generará un

nuevo fichero de configuración asociado llamado “sha160_wrapper_config.m”, en el que tendremos que añadir los ficheros que utiliza este modelo:

```
this_block.addFile('..\..\sha1_code\rtl\sha160_wrapper.vhd');
this_block.addFile('..\..\sha1_code\rtl\sha1.vhdl');
this_block.addFile('..\..\sha1_code\rtl\c6b.vhdl');
this_block.addFile('..\..\sha1_code\rtl\c4b.vhdl');
```

Una vez realizado este cambio, deberemos asignar al componente un simulador, al igual que para el caso del algoritmo SHA256 usaremos ISE Simulator. El modelo resultado tras hacer estas simples modificaciones es el siguiente:

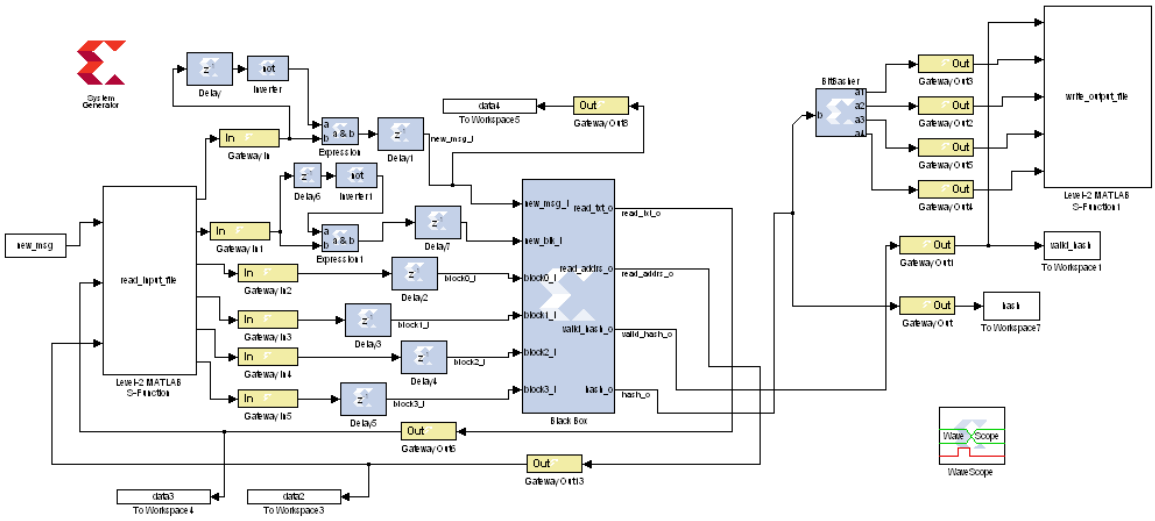


Como se puede observar a parte de la sustitución del “black box” se han eliminado algunos puertos de depuración empleados para el algoritmo SHA256.

Si se desea utilizar el modelo para SHA512, se harán las mismas modificaciones. Simplemente hay que indicar que el fichero a emplear es “sha512_wrapper.vhd” y tras la generación automática del fichero “sha512_wrapper_config.m”, se añadirán los ficheros que emplea el código del algorimo:

```
this_block.addFile('..\..\sha512_code\rtl\sha512_wrapper.vhd');
this_block.addFile('..\..\sha512_code\rtl\sha512.vhdl');
this_block.addFile('..\..\sha512_code\rtl\c8b.vhdl');
this_block.addFile('..\..\sha512_code\rtl\c4b.vhdl');
this_block.addFile('..\..\sha512_code\rtl\romk.vhdl');
```

El modelo resultante es el mismo que para SHA160 pero con el componente “black_box” modificado para usar otros ficheros:



## 5.3.3.1.1.2. Verificación

En este apartado se describirán los pasos a seguir para ejecutar la aplicación realizada y verificar su correcto funcionamiento. Para que todos los ficheros que se empleen aparezcan en el navegador de Matlab lo más conveniente es señalar como carpeta de trabajo el directorio donde se encuentren los ficheros que vayamos a ejecutar. Lo primero será inicializar las variables que provienen del entorno Matlab y que son utilizadas por el modelo de System Generator. Para ello, se ha creado el fichero “initialization.m”, así que bastará con llamar a este fichero para ejecutar el código que se halle en su interior:

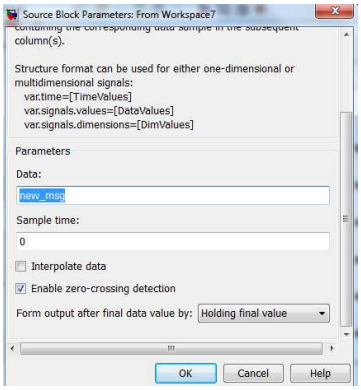
```
>> initialization
>>
```

sha512.v | initialization.m

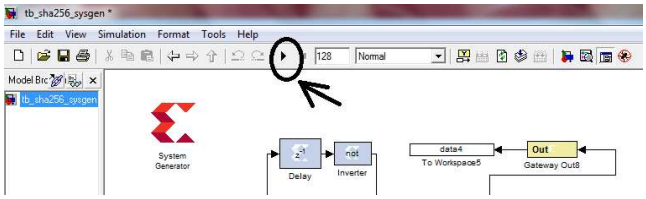
```
1  for i = 1:128,
2  new_msg(i,1)= i;
3  new_msg(i,2)= 0;
4  end
5  new_msg(10,2) = 1;
6  block_size = 512;
7
8
```

Como ya se comentó previamente, las señales que se introduzcan como entradas a Simulink desde Matlab deben tener dos dimensiones: la primera indicando un instante de tiempo y la segunda indicando el valor del dato en dicho instante. Con las anteriores líneas de código damos valor a 128 posiciones del array con un instante de tiempo igual a ‘i’. Primero damos el valor ‘0’ a todas las posiciones dentro del bucle y posteriormente, asignamos el instante donde se producirá la señal de comienzo (instante 10). Si se ejecuta Simulink durante más de

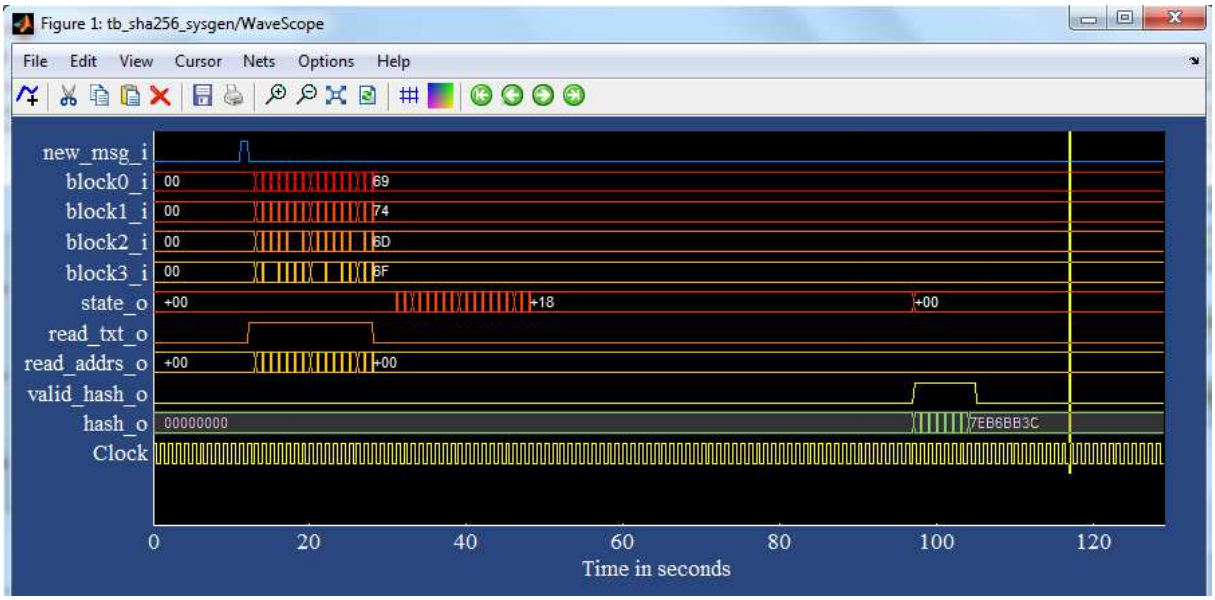
esos 128 ciclos al resto de datos se le asignará el valor de la última posición (holding final value), ya que así lo hemos indicado en las propiedades del bloque “From WorkSpace”:




La variable “block_size” sirve para identificar el tamaño del bloque de entrada que requiere el algoritmo, en el caso de sha256→512bits. Una vez, declaradas todas las variables de entrada se ejecuta el modelo pulsando el botón de “play”, indicando previamente el número de ciclos necesarios para la completa ejecución del modelo.




Dentro del modelo, se puede ver un bloque denominado “WaveScope” que servirá para depurar la aplicación desarrollada. Esta herramienta funciona a modo de analizador lógico, permitiendo añadir señales del modelo. Es importante destacar que no permite añadir señales del interior de la “Black Box”, lo cual es una limitación importante de System Generator, ya que dificulta la depuración obligando a entregar como puerto de salida de la “black box” toda aquella señal que se quiera visualizar. A continuación, se muestra una captura de pantalla en la que aparece la salida de esta herramienta junto a las señales que se visualizaron en la simulación con ModelSim SE (simulación realizada para verificar el código VHDL) tras ejecutar el modelo durante 128 ciclos para obtener el resumen de un bloque de 512bits:




Una vez finalizada la ejecución, la función “write_output_file” escribirá en el fichero hexadecimal “output_hash” el resultado del resumen realizado sobre el fichero de entrada. Dicho fichero se puede comparar con el resultado obtenido por el modelo de referencia ejecutado sobre el mismo fichero de entrada.

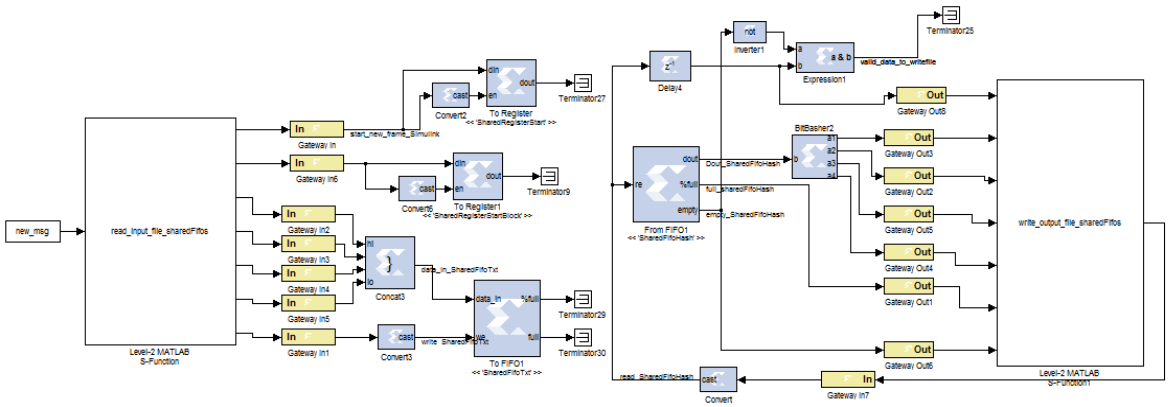
La verificación para el algoritmo SHA1 se realiza de manera idéntica a la que se acaba de comentar, con la obvia diferencia de que el resultado almacenado en el fichero de salida será distinto.

Para verificar SHA512 la única modificación a realizar será asignar el valor 1024 a la señal “block_size”. El resto del proceso es idéntico al realizado para los otros dos algoritmos.

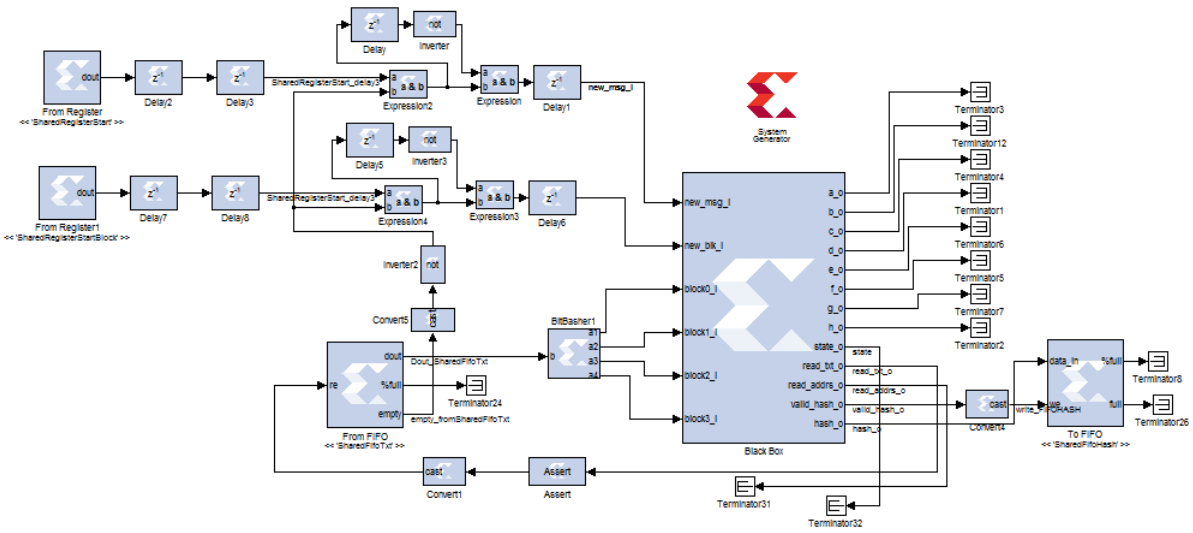
### 5.3.3.2. Hito 2a

#### 5.3.3.2.1.1. Implementación

En este hito se pretende definir plenamente las partes de la aplicación que serán ejecutadas en la FPGA y las partes de la aplicación que se ejecutarán en el PC. La primera figura representa la parte de la aplicación que será ejecutada en el entorno Simulink (PC) y cuya principal misión es gestionar la entrada/salida de datos del algoritmo.




La siguiente figura representa la parte de la aplicación que posteriormente se implementará en la FPGA:



Como puede observarse, la parte de la izquierda de la primera figura se corresponde con la entrada de datos, y la parte de la derecha con la salida de datos. Se trata de conjuntos de bloques totalmente independientes, no existen conexiones que unan la entrada o la salida con el conjunto de bloques que se implementarán en la FPGA (segunda figura).

Los datos son transmitidos/recibidos a través de las FIFOS que son denominadas en los esquemas como “SharedFifoTxt” (gestión de la entrada) y “SharedFifoHash” (gestión de la salida). Como ya se comentó anteriormente este tipo de componentes están compuestos por dos bloques, cada bloque puede ser ejecutado en un dominio distinto (FPGA y PC en nuestro caso), y los datos contenidos son los mismos. En el caso de “SharedFifoTxt” el bloque “To FIFO” es el encargado de rellenar el contenido de la memoria con los datos procedentes de fichero leídos por la M-Function “read_input_file_sharedFifos”. Los datos que se escriban en el componente “To FIFO” (SharedFifoTxt) pueden ser leídos haciendo uso de la imagen “From FIFO” (SharedFifoTxt). Será este último bloque el que suministrará datos al algoritmo.

De forma similar, la gestión de la escritura se realiza con la pareja denominada “SharedFifoHash”. El algoritmo escribe los datos que contienen el resultado en “To FIFO” (SharedFifoHash) y la función de Matlab “write_output_file_sharedFifos” lee del bloque “From FIFO” (SharedFifoHash) estos datos para escribirlos en fichero. En dicha función se introduce la señal “empty” y “%full”, para comenzar la lectura cuando la FIFO haya sido llenada y detenerla cuando se vacíe.

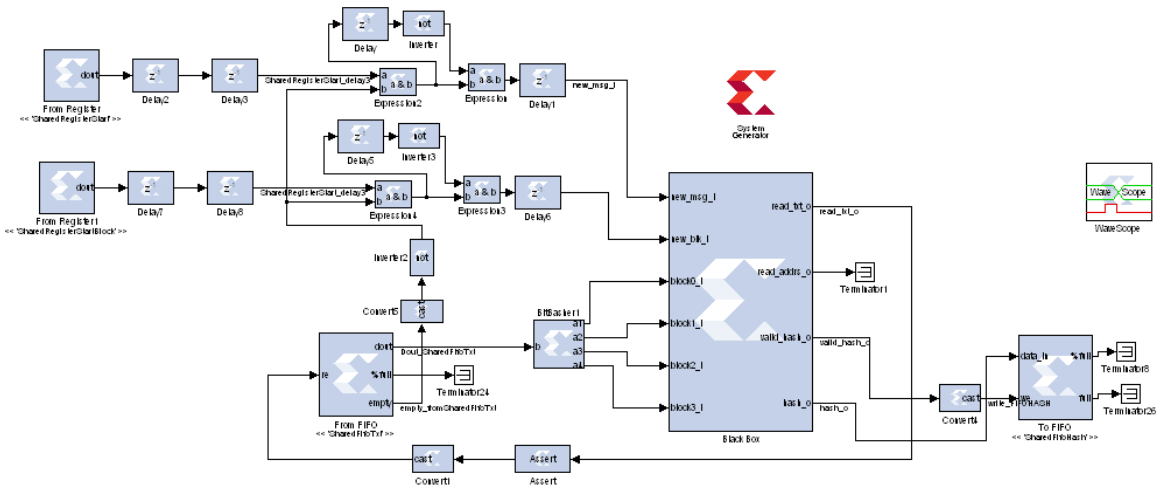
Si se analizan detenidamente ambas figuras se observa el uso de otros componentes llamados From/To Register. La pareja denominada “SharedRegisterStart” es utilizada para pasar la señal de 1 bit que indica el comienzo del proceso de un dominio a otro. En la parte que se implementará en la FPGA se observa, que a la salida del componente “From Register” se han añadido dos registros (Delays). Esto es necesario para asegurar la estabilidad de una señal que proviene de otro dominio de reloj (prevenir la denominada en diseño digital, “metaestabilidad). Sin estos registros, se podrían obtener valores de señal indeterminados (ni ‘1’ ni ‘0’). La lógica a continuación de estos registros tiene como misión detectar un flanco de subida de dicha señal para generar finalmente, la señal de comienzo para el algoritmo siempre y cuando haya datos disponibles en la FIFO (no esté vacía). La pareja anterior indica el comienzo de un nuevo mensaje, mientras que la pareja “SharedRegisterStartBlock” indica el comienzo de un nuevo bloque dentro del mismo mensaje.

Los bloques denominados “cast” sirven para adaptar tipos de datos dentro de System Generator, ya que las señales de lectura y escritura de los bloques Shared FIFOS son de tipo “Boolean” y las señales a las que se conectan son de tipo “unsigned de 1bit”.

El bloque “assert” es un aserto necesario para indicar a System Generator que la realimentación entre los componentes From FIFO (SharedFifoTxt) y la Black Box se ejecutan a la misma frecuencia (“Sample Rate”).

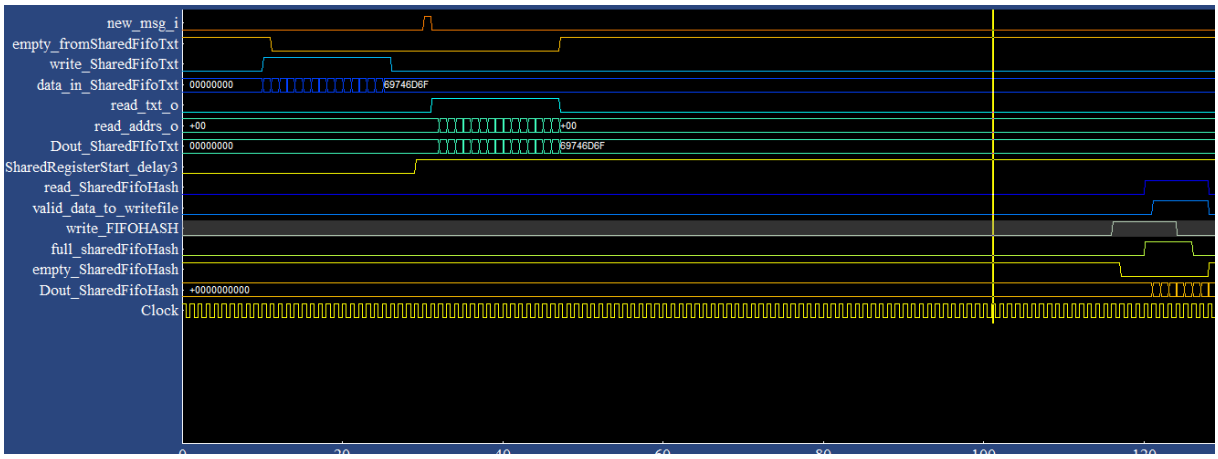
Las funciones que gestionan la entrada y salida de datos han sido modificadas para adaptarse a la separación de dominios. La función “read_input_file_sharedFifos” realiza la lectura del fichero a partir de la señal que indica el comienzo desde Matlab (new_msg). Lee automáticamente un bloque de 512bits (SHA256 y SHA1) o 1024bits (SHA512) y lo almacena en la memoria “To FIFO SharedFifoTxt” dividiendo dicho bloque en sub-bloques de 32bits y generando la señal de escritura para dicha FIFO. Una vez finalizada la escritura, genera la señal de comienzo del proceso que se transmite a través de “SharedRegisterStart”. Respecto a la salida, la función “write_output_file_sharedFifos” realiza automáticamente la lectura de la FIFO “From FIFO SharedFifoHash” y almacena el resultado en un fichero. La lectura comienza al detectar que la FIFO ha sido rellenada y finaliza al vaciarla.

Al igual que en el hito anterior, las modificaciones para incluir los algoritmos SHA1 y SHA512 son mínimas. Incluso para este modelo son más sencillas. Como ya se crearon en el hito anterior los ficheros de configuración asociados al “black box” (sha512_wrapper_config.m y sha160_wrapper_config.m) bastará con copiar estos ficheros a la carpeta de trabajo de este hito. Por otro lado, se copiará el componente “black box” del hito1 al modelo del hito2. El resultado de esta última operación es el siguiente modelo:



# 5.3.3.2.1.2. Verificación

Para verificar el correcto funcionamiento de este modelo se deben seguir exactamente los mismos pasos que se han seguido durante la verificación del Hito 1. La salida del WaveScope para la misma prueba realizada en el anterior hito sobre SHA256 puede verse en la siguiente figura:



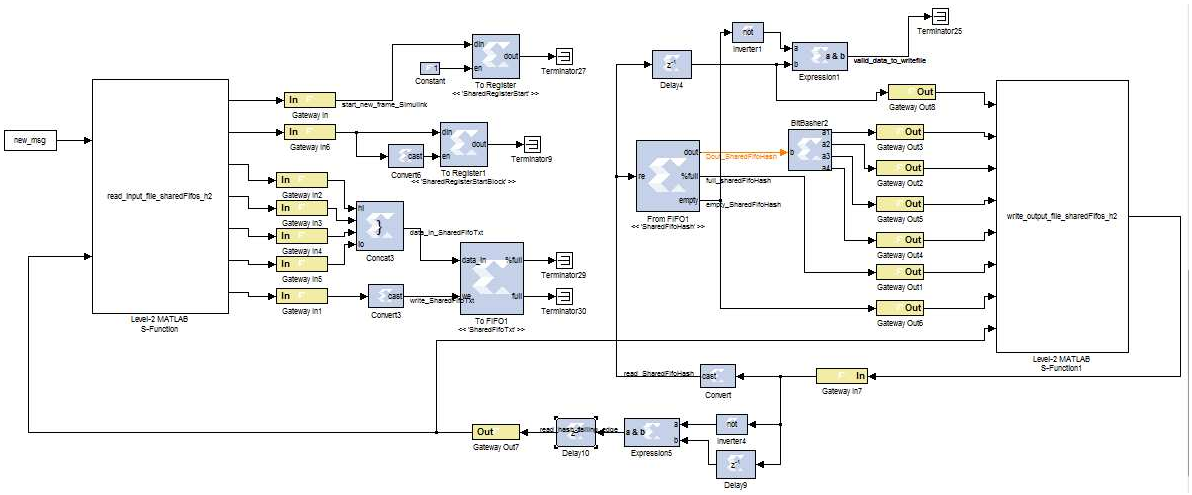

En esta figura se pueden observar cómo “SharedFifoTxt” inicialmente está vacía. Cómo tras finalizar el proceso de su escritura se genera una señal de comienzo de mensaje “new_msg_i” para el algoritmo. A continuación, el algoritmo lee la memoria hasta vaciarla nuevamente y comienza el procesado. Al finalizar el procesado, el resultado es escrito en “SharedFifoHash” y una vez que se rellena dicha memoria, se lanza una lectura de la memoria para escribir los datos en un fichero.

La verificación para SHA512 y SHA160 se realiza de la misma manera que la realizada en el Hito1.

#### 5.3.3.3. Hito2b

##### 5.3.3.3.1.1. Implementación

El objetivo de este hito es completar la implementación obtenida en el hito anterior añadiendo la posibilidad de trabajar sobre ficheros de tamaños mayores para posibilitar la realización de simulaciones largas sobre las que realizar las medidas de tiempos de ejecución. En el anterior hito se trabajaba únicamente con un bloque de 512bits para SHA256 y SHA1 y de 1024bits para SHA512. En este hito se realizan las modificaciones necesarias para procesar todos los bloques de los que conste el fichero de entrada. Las modificaciones se tienen que realizar sólo en la gestión de entrada y salida:

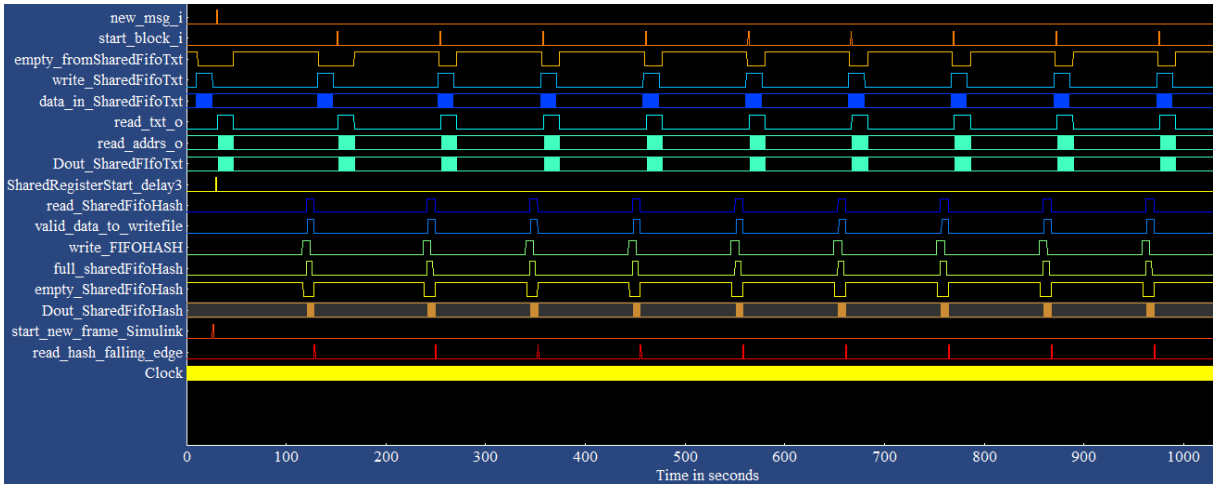



Básicamente, a partir de un flanco de bajada de la señal de lectura de la FIFO “From FIFO (SharedFifoHash) se indica el final del procesado de un bloque y el comienzo de otro. La S-Function de entrada ha sido modificada para iniciar un nuevo proceso de lectura con cada nuevo bloque y generar la señal de comienzo de bloque. La S-Function de salida también ha sido modificada para incluir un nuevo puerto de entrada que indique el comienzo de un nuevo bloque, cuya única función es permitir que en el fichero de salida que contiene el resultado de la operación sobre todo el fichero sólo se contenga el último hash obtenido (no un conjunto de resúmenes intermedios).

Este hito también sirve como ejemplo para ver cómo cambios en la gestión de la entrada/salida no afectan a la implementación del algoritmo.

# 5.3.3.3.1.2. Verificación

En este nuevo hito, para obtener el resumen de un fichero de entrada basta con ejecutar la aplicación hasta que se detenga. Para ello, en el tiempo de simulación se indicará ejecución infinita mediante la palabra “Inf”. Tras finalizar la ejecución, la herramienta WaveScope generará una gráfica en la que se podrá observar cómo evolucionan las señales implicadas en el proceso:



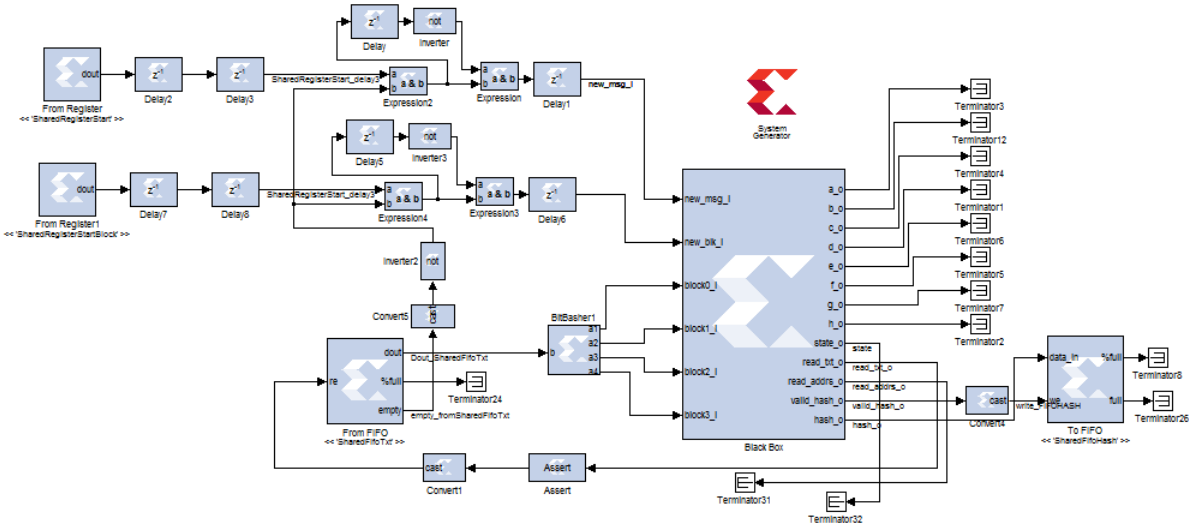

*Simulación realizada sobre SHA256*

En la gráfica se muestra la generación de un único pulso de la señal de comienzo de nuevo mensaje (new_msg_i) al principio de la simulación y cómo después se van autogenerando las señales de comienzo de nuevo bloque (new_block_i) y se van obteniendo los resúmenes intermedios.

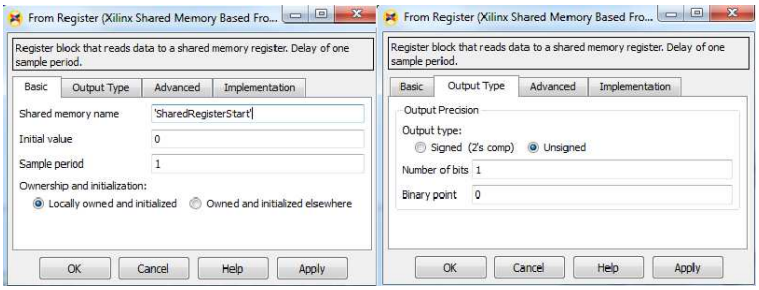
#### 5.3.3.4. Hito 3

##### 5.3.3.4.1.1. Implementación

El objetivo de este Hito es generar el componente que será ejecutado en la FPGA y comprobar el correcto funcionamiento del Framework de intercomunicación. Para la generación del componente, se aisla en un nuevo modelo (fichero *.mdl) los bloques indicados en la segunda figura del anterior apartado (Hito2b) que se corresponden con el algoritmo criptográfico y las FIFOs para la gestión de la entrada/salida de datos en la FPGA. Una vez creado el nuevo modelo independiente, se debe añadir un “token” System Generator. El resultado se muestra en la siguiente figura:

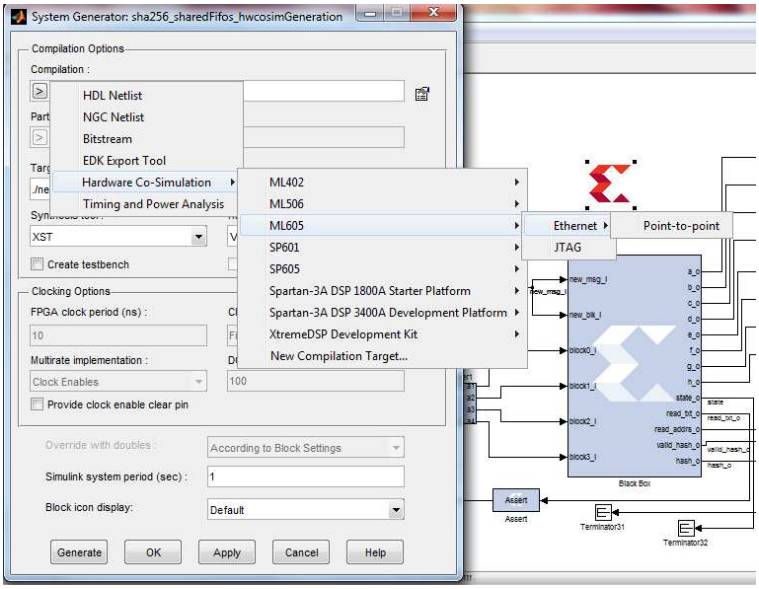



Como se ha aislado este conjunto de bloques del resto del sistema, se deben hacer unas modificaciones en los bloques From Register “SharedRegisterStart” y “SharedRegisterStartBlock”, From FIFO (SharedFifoTxt) y To FIFO “SharedFifoHash”. Estas modificaciones, se centran en declarar de manera local a este modelo el tipo de datos y el tamaño de estos componentes. Para ello, en la configuración de cada componente se selecciona “Locally Owned and initialized” y se introduce el tamaño que. Por ejemplo, para el caso del registro:



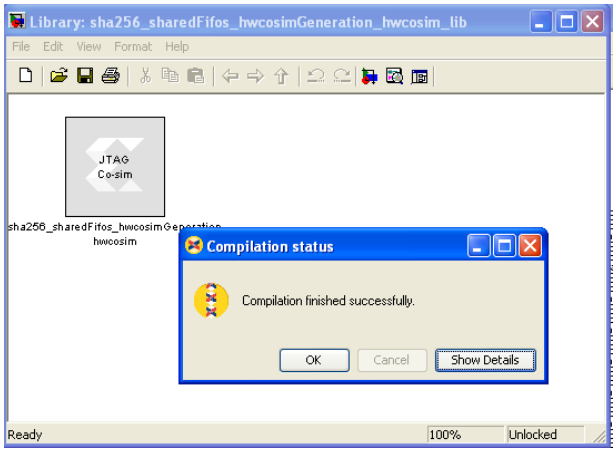

Estas modificaciones son necesarias debido a que en cada pareja de Shared Memories, FIFOs o Registers uno de los bloques de la pareja hereda el tamaño del otro. Por defecto, los "From" lo heredan de los componentes de tipo "To"; pero en este caso para el modelo no existe todavía el otro componente de la pareja.

Una vez realizados estos cambios, se debe proceder a generar el componente para realizar la co-simulación HW. Para ello, en la configuración del Token de System Generator se seleccionan las siguientes opciones:

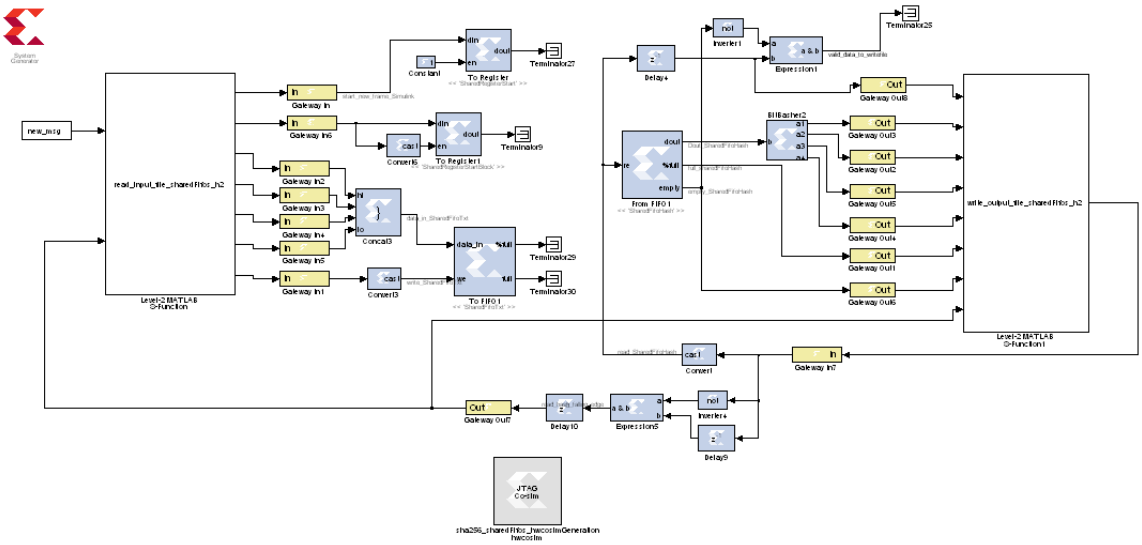



En "Compilation" se debe seleccionar Hardware Co-simulation/ML605/JTAG. Primero, se creará un componente para co-simulación a través del JTAG al ser más sencilla la depuración en este modo de algún problema que surja. ML605 es el nombre de la tarjeta de evaluación de la que se dispone, como se comentó anteriormente, si se escoge una tarjeta de evaluación que sea directamente soportada por System Generator no se deberán realizar tareas de configuración extra. Una vez finalizado este sencillo paso, se mantendrán el resto de opciones por defecto. Pulsando el botón "Generate" y si no se produce ningún error, la herramienta procederá a generar todo lo necesario para realizar la co-simulación HW sin que sean

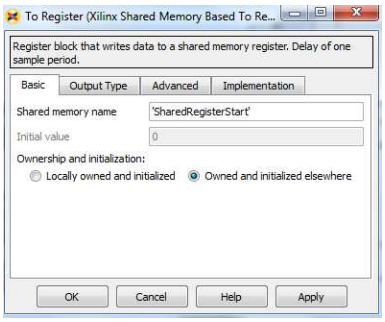
necesarias más acciones por parte del usuario. Una vez completado este proceso, aparecerá en pantalla un nuevo modelo que incluye el componente para realizar la co-simulación HW:




Con la gestión de entrada/salida que se indicó en la primera figura del Hito2b, se creará otro modelo independiente y se añadirá este nuevo componente, tal y como se muestra en la siguiente figura:



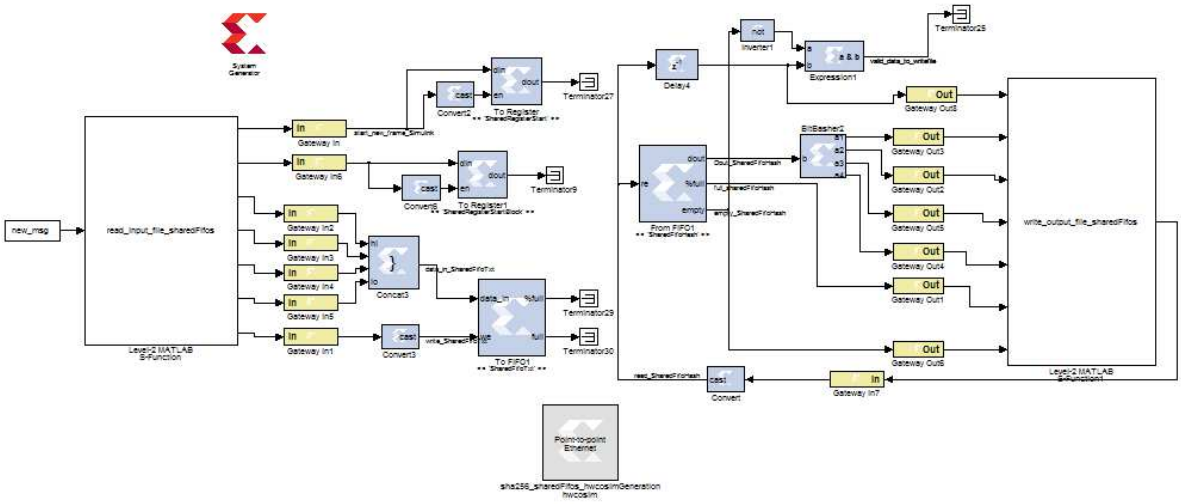
A continuación, se debe proceder a asignar los tamaños correspondientes a los bloques SharedFifos y SharedRegister, para ello se indica en cada bloque “Owned and Initialized elsewhere”. De esta forma, la configuración se heredará de los bloques que conforman la pareja de cada componente que han sido inicializados localmente dentro del bloque de co-simulación. A continuación se muestra cómo se configura el bloque To Register “SharedRegisterStart”:




Una vez realizado este paso, el modelo se encuentra en condiciones de realizar la co-simulación hardware que se explicará en el apartado de Verificación.

La generación del bloque correspondiente para la co-simulación hardware a través de Ethernet se realiza de la misma manera que lo indicado para la co-simulación basada en JTAG con la excepción de que habrá que seleccionar dicho interfaz en el token de System Generator.

Una vez finalizado el proceso, se obtendrá un nuevo bloque. Si se sustituye el bloque para JTAG por el nuevo bloque para co-simulación por Ethernet, se estará en disposición de realizar la co-simulación HW a través de este interfaz. El resultado se puede ver en la siguiente figura:

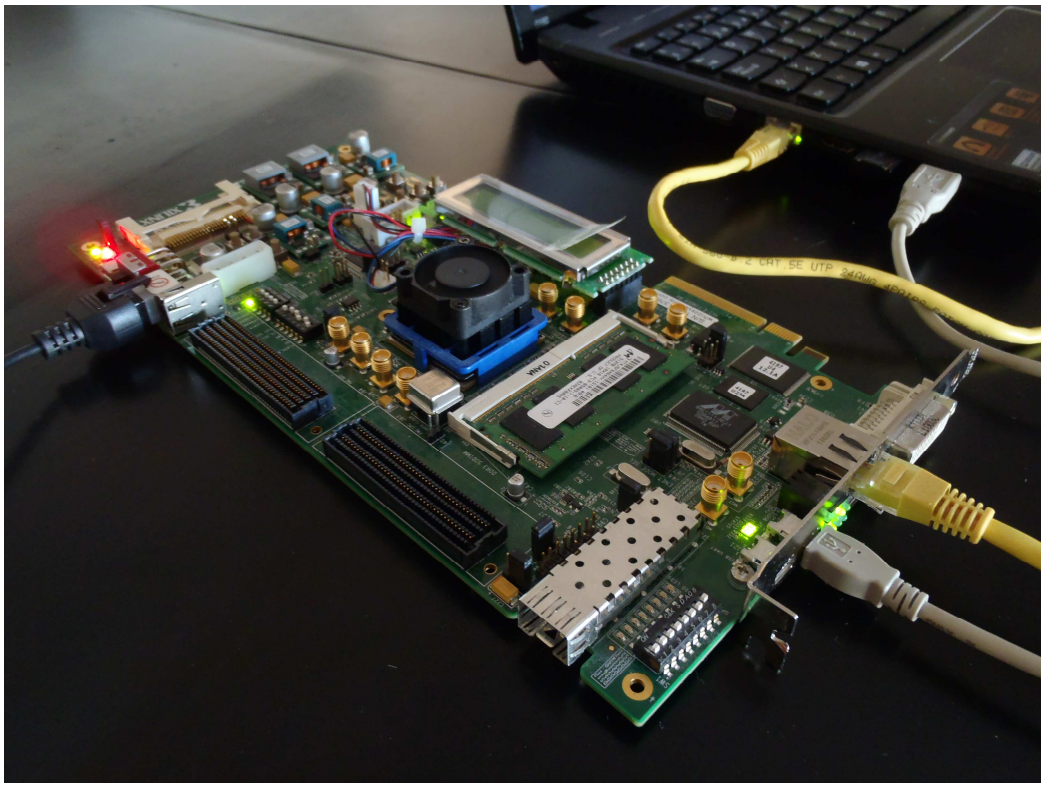



Los pasos a seguir para obtener los componentes para la co-simulación HW de los algoritmos SHA1 y SHA512 son idénticos. Obviamente en el modelo que se use para cada caso, el black box deberá estar configurado para usar dicho algoritmo.

Para utilizar un algoritmo u otro bastará con sustituir el bloque de co-simulación (caja gris de la anterior figura) por el que se desee emplear en el modelo a ejecutar.

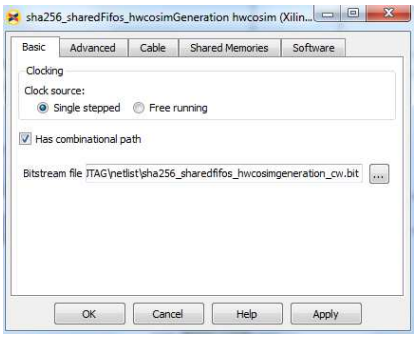
#### 5.3.3.4.1.2. Verificación

En este apartado se explicará cómo realizar la co-simulación hardware una vez realizados todos los pasos que han sido comentados en la parte de implementación. Se comenzará indicando cómo se debe de conectar el ordenador a la tarjeta de evaluación:



Como se puede observar en la fotografía anterior, la tarjeta será conectada al PC por medio de un cable USB y un cable de categoría 6 (CAT6) que permita establecer una conexión a 1Gbps. La FPGA será programada a través del USB, y una vez finalizada dicha programación se podrá ejecutar la aplicación realizando la transferencia de datos por medio de la conexión Gigabit.

Se procederá primero con la verificación de la co-simulación hardware a través del JTAG. Para ello, en el bloque “sha256_sharedfifos_hwcosimgeneration_hwcosim” se dispone de la siguiente ventana de configuración:



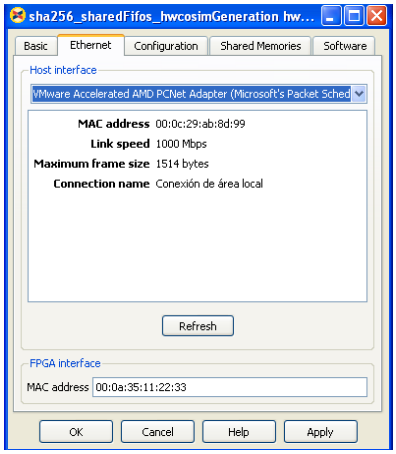

Configurando “clock source” como Single Stepped se asegura que el comportamiento de la co-simulación hardware durante la simulación será el mismo que el obtenido en el Hito2 ejecutando todo sobre PC. Con esta configuración la ejecución en la FPGA estará sincronizada con la simulación en Simulink. Si se selecciona Free Running, la FPGA se ejecutará de manera asíncrona respecto a la simulación en Simulink. Para la primera co-simulación hardware se usará el modo Single Stepped para facilitar la depuración de los problemas que surjan.

Para realizar la co-simulación hardware sobre JTAG se deberá asegurar que el bloque que se utiliza en el modelo es el adecuado, y se procederá a pulsar el botón “Play” dando el valor “Inf” al número de ciclos a emplear. Una vez realizada esta acción, la FPGA es configurada a través del JTAG y se ejecuta la co-simulación. Con la ejecución de esta co-simulación se observa que no se ha obtenido ninguna mejora respecto al tiempo de simulación usando únicamente el PC (Hito 2). Realmente se ha incrementado el tiempo de ejecución considerablemente debido a que la comunicación PC-FPGA a través del JTAG es extremadamente lenta. Con esta prueba, simplemente se ha verificado que el algoritmo funciona adecuadamente sobre la FPGA. Los ciclos empleados en la simulación para procesar un bloque son los mismos que en el caso del Hito2b, ya que como ya se ha indicado, la ejecución en la FPGA está sincronizada con la ejecución en el PC.

Para probar los otros dos algoritmos, simplemente se deberá sustituir el bloque de co-simulación HW por “sha160_sharedfifos_hwcosimgeneration_hwcosim” para SHA1 y por “sha512_sharedfifos_hwcosimgeneration_hwcosim” para SHA512.

Para acelerar la simulación y obtener realmente una ganancia en tiempos de simulación respecto a la ejecución del modelo sobre PC, se debe hacer uso de la co-simulación por Ethernet. Para ello, simplemente se sustituirá el componente “hw cosim” por el equivalente para el interfaz Ethernet. En las propiedades del bloque, se indicará Free Running. De este modo el algoritmo será ejecutado de forma asíncrona respecto a Simulink y a la frecuencia de

funcionamiento de la FPGA (10ns→100MHz). Se deberá indicar el interfaz de red que se va a utilizar para la comunicación a través de Ethernet:



# 6. Resultados

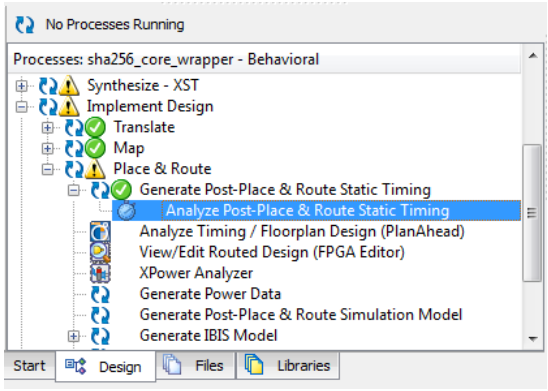
En este apartado se incluirán tablas resumiendo las prestaciones de la aplicación realizada, se presentarán los resultados de una forma objetiva, en el siguiente apartado se obtendrán las conclusiones basadas en este apartado. El objetivo de este apartado principalmente es organizar los resultados obtenidos en las etapas de verificación previas.

## 6.1. Algoritmos en lenguaje de descripción hardware

### 6.1.1. Lógica utilizada

Tras las modificaciones realizadas sobre el código de los algoritmos originales, se decidió crear un proyecto en ISE para sintetizarlo e implementarlo sobre FPGA y obtener unos datos de partida en cuanto a frecuencia de funcionamiento, ocupación y consumo. Estos resultados servirán para estimar cómo afectan al diseño las modificaciones que se han realizado tras la integración para habilitar la co-simulación hardware.

Sobre el proyecto “sha_256.xise” previamente creado, si se ejecuta “Analyze Post-Place & Route Static Timing” se realizan todos los pasos previos que permitirán obtener los ficheros necesarios para implementar el diseño en la FPGA.



Dentro de los ficheros a analizar, se encuentra “sha256_core_wrapper_map.mrp” donde se podrá encontrar un resumen de la lógica empleada para este algoritmo. A continuación, se indican los datos más relevantes de este fichero.

*Number of Slice Registers: 2,044 out of 301,440 1%*

*Number of Slice LUTs: 1,631 out of 150,720 1%*

*Number of occupied Slices: 545 out of 37,680 1%*

*Number of bonded IOBs: 73 out of 600 12%*

Para el algoritmo SHA1 se realiza el mismo proceso sobre el proyecto “sha160.xise”, obteniendo los siguientes resultados de ocupación:

*Number of Slice Registers: 805 out of 301,440 1%*

*Number of Slice LUTs: 664 out of 150,720 1%*

*Number of occupied Slices: 245 out of 37,680 1%*

*Number of bonded IOBs: 73 out of 600 12%*

Los siguientes resultados son los referentes al algoritmo SHA 512:

*Number of Slice Registers: 2,402 out of 301,440 1%*

*Number of Slice LUTs: 2,502 out of 150,720 1%*

*Number of occupied Slices: 838 out of 37,680 2%*

*Number of bonded IOBs: 74 out of 600 12%*

Los números anteriores muestran el número de registros y el número de LUTs (unidades para realizar lógica distribuida) que se utilizan del total disponible en la FPGA. Cada Slice está formado por 4 parejas compuestas cada una por una LUT y un Registro. De los datos anteriores, se deriva que se está utilizando para cada algoritmo un poco más del 1% de la lógica disponible en el dispositivo Virtex-6 Lx240T. También se indica el número de IOBs utilizados. Los IOBs son registros que se sitúan en los pines de entrada/salida para reducir el retardo desde que la señal entra en la FPGA hasta que se procesa o desde que se genera la señal hasta que abandona la FPGA.

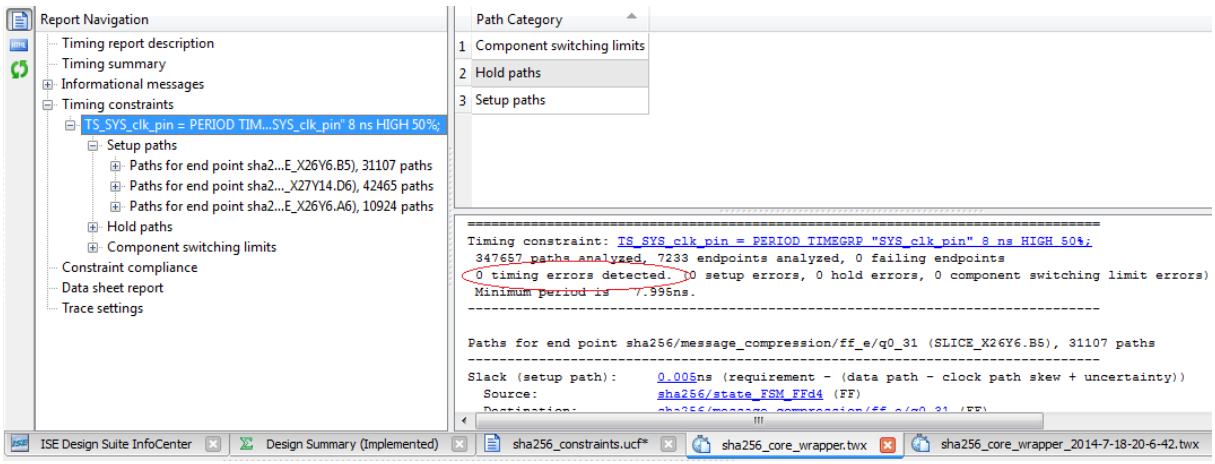
Aunque la FPGA utilizada para el desarrollo del presente Trabajo ha sido una Virtex-6 Lx 240T, en función de los resultados obtenidos se podrían utilizar dispositivos que dispongan de mucha menos lógica; con la correspondiente reducción de costes que esto supone.

### 6.1.2. Frecuencia de Funcionamiento

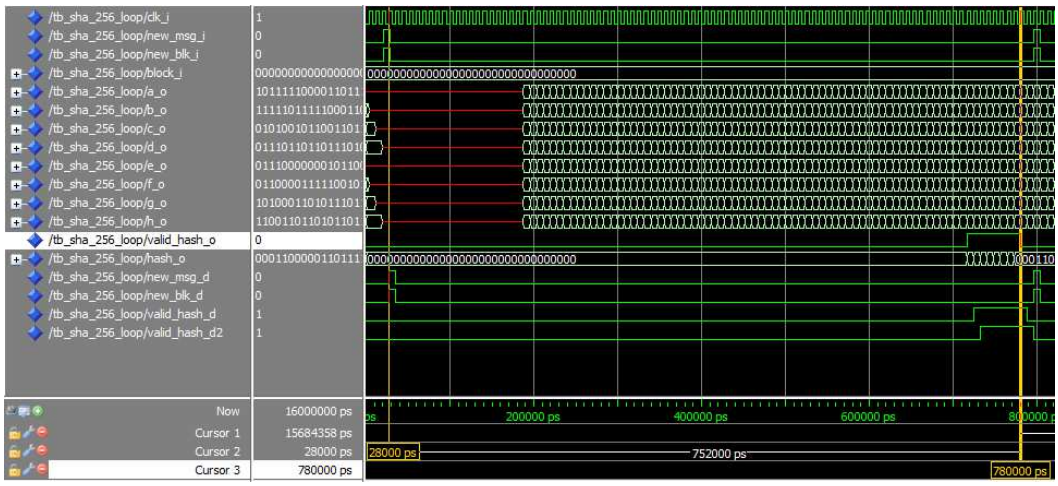
La frecuencia de operación máxima obtenida para el algoritmo SHA256 es de 125MHz. Para obtener este resultado, se indicó en el fichero de restricciones de tiempo del proyecto (sha256_constraints.ucf) un período de funcionamiento de 8ns:

```
NET "clk_i" TNM_NET = "SYS_clk_pin";
TIMESPEC "TS_SYS_clk_pin" = PERIOD "SYS_clk_pin" 8 ns HIGH 50 %;
```

Para comprobar que se ha alcanzado esta frecuencia de funcionamiento sin errores, se debe abrir el fichero “sha256_core_wrapper.twx”:




Si se analizan las simulaciones realizadas, se observa que se ejecutan 94ciclos para la obtención del resumen de un bloque de 512bits.



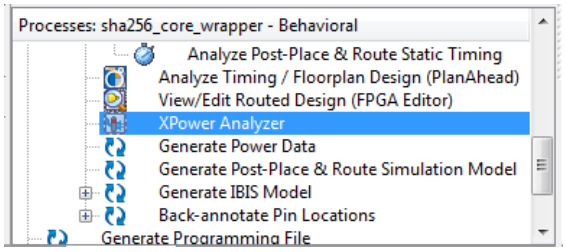

Empleando una frecuencia de 125MHz (8ns), el procesado de un bloque de 512 bits consumirá 752ns (8ns * 94ciclos). Por lo que se obtiene una tasa de procesamiento de 1329787 bloques por segundo

Para el algoritmo SHA1 se obtiene una frecuencia máxima de funcionamiento de 200 MHz. Sabiendo que el procesado de un bloque se realiza en 90ciclos, se deduce que este algoritmo es capaz de procesar en un segundo 2222222 bloques de 512bits.

En el algoritmo SHA512 la frecuencia máxima obtenida es de 142.85MHz (7ns), por lo que la tasa de procesamiento es de 1180578 bloques de 1024bits por segundo, ya que el algoritmo emplea 121ciclos para el procesamiento de cada bloque.

### 6.1.3. Consumo

Para disponer de una estimación del consumo asociado a cada algoritmo, se deberá ejecutar la herramienta XPower Analyzer:



Esta herramienta proporciona como salida el fichero “sha256_core_wrapper.pwr”, donde se puede observar el consumo del algoritmo implementado:

| On-Chip | Power (W) | Used | Available | Utilization (%) |
|---|---|---|---|---|
| Clocks | 0.019 | 1 | --- | --- |
| Logic | 0.022 | 1631 | 150720 | 1 |
| Signals | 0.021 | 3105 | --- | --- |
| IOs | 0.035 | 73 | 600 | 12 |
| Leakage | 2.708 | | | |
| Total | 2.804 | | | |

El resultado muestra que la lógica del algoritmo consume únicamente 0.022W, pero debido a que se está alimentando un dispositivo que dispone de una enorme cantidad de lógica el consumo final asciende a 2.804W. Por esta razón, y obviamente por el precio del dispositivo, es de extremada importancia elegir un dispositivo que se ajuste a los requerimientos del proyecto en cuestión. Se debe elegir el dispositivo considerando la implementación actual, futuras mejoras e inclusiones de nuevos algoritmos que se deseen realizar en próximas revisiones. El algoritmo criptográfico puede ser parte de un diseño en la FPGA de mucha mayor envergadura, por lo que se deberá contemplar el tamaño de todo el diseño. Por otro lado, esta estimación de consumo está asumiendo que todas las entradas y salidas del algoritmo serán conectadas a sus correspondientes pines, esta asunción puede no ser cierta ya que habitualmente, la entrada/salida se realizará mediante su correspondiente interfaz.

Los resultados obtenidos para los otros dos algoritmos se resumen en las siguientes tablas:

| On-Chip | Power (W) | Used | Available | Utilization (%) |
|---|---|---|---|---|
| Clocks | 0.021 | 1 | --- | --- |
| Logic | 0.001 | 664 | 150720 | 0 |
| Signals | 0.001 | 1003 | --- | --- |
| IOs | 0.005 | 73 | 600 | 12 |
| Leakage | 2.706 | | | |
| Total | 2.733 | | | |

SHA1

| On-Chip | Power (W) | Used | Available | Utilization (%) |
|---|---|---|---|---|
| Clocks | 0.025 | 1 | --- | --- |
| Logic | 0.023 | 2502 | 150720 | 2 |
| Signals | 0.012 | 3918 | --- | --- |
| BRAMs | 0.015 | * | * | * |
| IOs | 0.014 | 74 | 600 | 12 |
| Leakage | 2.708 | | | |
| Total | 2.796 | | | |

SHA512

### 6.1.4. Tiempos de Simulación

Para la realización de las medidas de los tiempos de simulación consumidos en las soluciones basadas únicamente en PC se empleará el banco de pruebas creado en lenguaje de descripción hardware del Hito1, que posibilita hacer una prueba exhaustiva mediante el uso de un bucle infinito (fichero "tb_sha_256_loop.vhd"). Las simulaciones se ejecutarán en los entornos de simulación ModelSim SE e ISIM. Por otro lado, se realizarán medidas de los tiempos de simulación que consume la aplicación creada en el Hito2b (basada en el uso de Simulink/System Generator), ya que este proyecto también se ejecuta únicamente haciendo uso del PC. En ambos casos se procesarán 1000 bloques y 10000 bloques de 512bits.
Los resultados obtenidos se muestran en la siguiente tabla:

| | **ModelSim SE** | **System Generator** | **ISIM** |
|---|---|---|---|
| **1000 bloques** | 11''86 | 1'34''70 | 16''44 |
| **10000 bloques** | 1'37''14 | 14'17''48 | 4'48''86 |

Para las pruebas se ha usado una máquina Virtual con sistema operativo Windows XP Professional a la que se ha asignado un core del microprocesador Intel i7-3520M CPU@2.9Ghz y 2GB de RAM.

## 6.2. Co-simulación Hardware

El proceso de generación del proyecto y la invocación de las distintas herramientas del entorno ISE hasta obtener un bitstream de configuración de la FPGA ya fueron realizados automáticamente por la herramienta System Generator al seguir los pasos indicados en el Hito3. Por lo que para obtener los resultados en cuanto a ocupación lógica y frecuencia de operación no se requiere generar un nuevo proyecto, basta con consultar los ficheros generados en la carpeta "XFlow". El fichero de más alto nivel del proyecto es "eth_cosim_top.vhd", dicha entidad contiene a todos los componentes que son necesarios para realizar la co-simulación hardware. En los siguientes apartados se mostrarán los resultados obtenidos para el algoritmo SHA256 y que servirán como base para la comparación con las soluciones basadas en el uso exclusivo de PC.

### 6.2.1. Lógica utilizada

La lógica empleada por el proyecto para el algoritmo SHA256 la podemos consultar en el fichero “Xflow/eth_cosim_top_map.mrp”, destacando los siguientes datos de su contenido:

*Number of Slice Registers: 2,713 out of 301,440 1%*

*Number of Slice LUTs: 2,206 out of 150,720 1%*

*Number of occupied Slices: 679 out of 37,680 1%*

*Number of RAMB36E1/FIFO36E1s: 8 out of 416 1%*

*Number of bonded IOBs: 51 out of 600 8%*

### 6.2.2. Frecuencia de Funcionamiento

La máxima frecuencia de funcionamiento obtenida para todos los algoritmos es de 100MHz como se puede observar en el fichero que contiene el resultado del proceso para cada modelo, “xflow.results”. La limitación de 100MHz viene impuesta por el uso de la tarjeta de evaluación ML605, ya que a la hora de crear el componente para la co-simulación HW no es posible aumentar este valor de frecuencia.

Por lo tanto, para el algoritmo SHA256 considerando que se consumen 94 ciclos en el procesamiento de un bloque, se obtiene una tasa teórica de procesamiento de 1063829 bloques de 512bits por segundo.

### 6.2.3. Consumo

Para obtener el consumo asociado es necesario invocar a la herramienta XPower Analyzer desde el proyecto autogenerado: “sha256_sharedfifos_hwcosimgeneration_cw.xise”. Los resultados obtenidos se muestran en la siguiente tabla:

| On-Chip | Power (W) | Used | Available | Utilization (%) |
|---|---|---|---|---|
| Clocks | 0.013 | 1 | --- | --- |
| Logic | 0.016 | 1527 | 150720 | 1 |
| Signals | 0.013 | 2894 | --- | --- |
| IOs | 0.076 | 74 | 600 | 12 |
| Leakage | 2.709 | | | |
| Total | 2.828 | | | |

Como se puede observar el consumo de la lógica empleada por toda la aplicación es inferior al consumo del algoritmo aislado, esto es debido a que la frecuencia de funcionamiento empleada es menor. Si se modifica el fichero de restricciones de timing del proyecto para aumentar la frecuencia de funcionamiento, los resultados obtenidos son los siguientes:

| On-Chip | Power (W) | Used | Available | Utilization (%) |
|---|---|---|---|---|
| Clocks | 0.020 | 1 | --- | --- |
| Logic | 0.020 | 1508 | 150720 | 1 |
| Signals | 0.018 | 2894 | --- | --- |
| IOs | 0.095 | 74 | 600 | 12 |
| Leakage | 2.710 | | | |
| Total | 2.863 | | | |

### 6.2.4. Tiempos de simulación

Para la realización de las medidas de los tiempos de simulación que consume la solución basada en el uso de PC y FPGA trabajando de manera conjunta (co-simulación Hardware) se utilizará el modelo desarrollado en el Hito3. Se realizarán medidas de los tiempos de verificación que consume la co-simulación HW por medio de JTAG y utilizando el interfaz Ethernet. En ambos casos se procesarán 1000 bloques y 10000 bloques.

Los resultados obtenidos para el algoritmo SHA256 se muestran en la siguiente tabla:

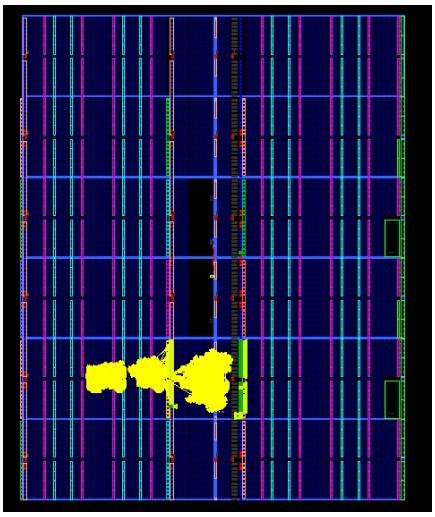
| | **JTAG (Single Stepped)** | **JTAG (Free Running)** | **Ethernet (Free Running)** |
|---|---|---|---|
| **1000 bloques** | 4’01’’45 | 18’52’’37 | 23’’36 |
| **10000 bloques** | 41’10’’03 | 3h11’01’’02 | 3’42’’57 |

Como se puede observar en la anterior tabla, los tiempos obtenidos por la co-simulación Hardware a través de JTAG son enormes para el caso del procesado de 10000 bloques. El tiempo de simulación empleado supera al obtenido en el proyecto realizado con System Generator ejecutándose exclusivamente en el PC. Esto se debe a que JTAG es un interfaz muy lento que ralentiza el proceso, al consumir gran cantidad de tiempo en la transmisión de datos entre el PC y la FPGA.

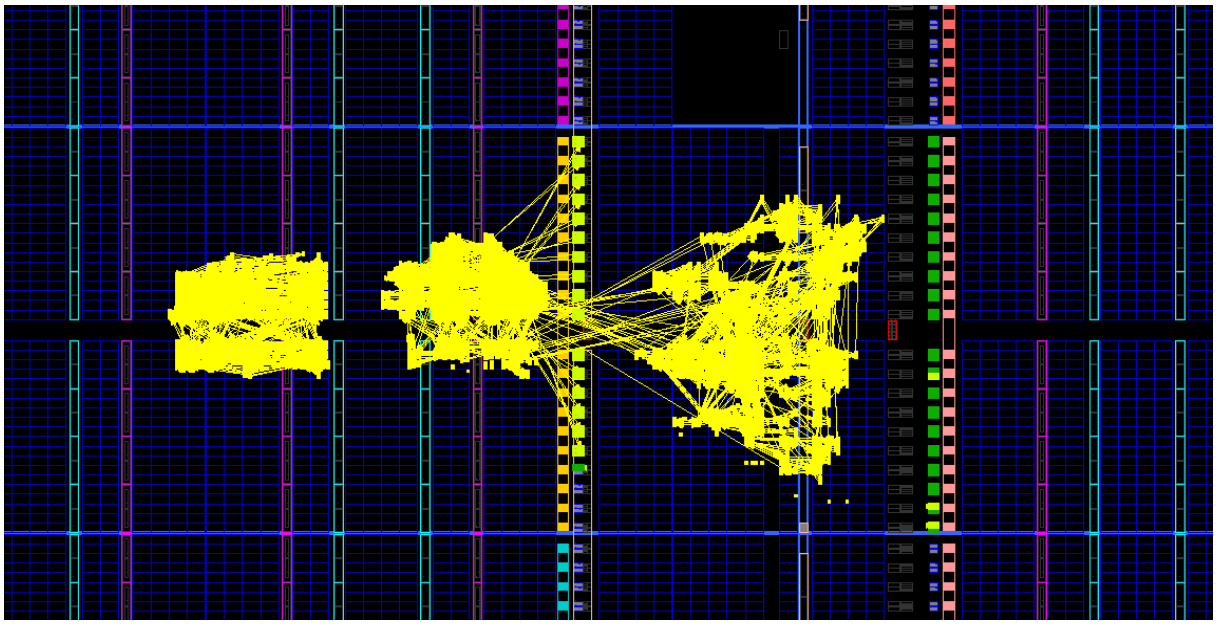
Al utilizar una transferencia de datos basada en el interfaz Gigabit por Ethernet se observa una importante mejora respecto a la ejecución en PC.

### 6.2.5. Place&Route

En la siguiente figura se puede observar cómo se ha implementado finalmente el diseño en la lógica interna de la FPGA para el algoritmo SHA256:



Esta imagen ha sido obtenida haciendo uso de la herramienta PlanAhead (incluida en el paquete de distribución del entorno ISE). Si se visualiza exclusivamente la zona en la que se ha posicionado el diseño obtenemos la siguiente imagen:



# 7. Conclusiones

La aplicación desarrollada pretende facilitar el desarrollo de proyectos en el ámbito de la criptografía que estén destinados a ser implementados sobre arquitecturas basadas en FPGA.

En el presente Trabajo Fin de Máster se ha mostrado cómo desarrollar una aplicación que ejecute un algoritmo criptográfico en una FPGA al que se le proporcionen los datos de entrada desde el PC a través de una comunicación eficiente basada en una conexión Ethernet Gigabit. El resultado obtenido tras el procesado realizado por el algoritmo es transmitido de manera equivalente al PC para realizar el análisis de dichos datos.

Mediante el uso de la aplicación propuesta, se posibilita para la generación de entradas y análisis de los resultados obtenidos la utilización de un lenguaje de cómputo sencillo y potente como es Matlab o el uso del interfaz gráfico que proporciona Simulink.

Las metodologías actuales basadas en soluciones ejecutadas sobre PC requieren amplios conocimientos en lenguajes de descripción hardware y obligan a destinar un importante esfuerzo a su codificación. Mediante el uso de Matlab y/o Simulink se facilitará la realización de bancos de prueba complejos y potentes reduciendo considerablemente los tiempos empleados en su implementación.

Los resultados demuestran que aplicando la co-simulación hardware se obtiene una importante reducción en tiempos de simulación respecto a las soluciones basadas únicamente en el uso de System Generator y Simulink ejecutándose exclusivamente en PC, tal y como demuestra la siguiente tabla comparativa:

| SHA 256 | **System Generator** | **Co-Simulación HW (Ethernet)** |
|---|---|---|
| **1000 bloques** | 1’34’’70 | 23’’36 |
| **10000 bloques** | 14’17’’48 | 3’42’’57 |

Si la comparación se realiza sobre el resultado obtenido por las simulaciones realizadas con los entornos de simulación ModelSim SE e ISIM (empleando lenguajes de descripción hardware) los resultados no son tan favorables:

| SHA256 | **ModelSim SE** | **ISIM** | **Co-Simulación HW (Ethernet)** |
|---|---|---|---|
| **1000 bloques** | 11’’86 | 16’’44 | 23’’36’ |
| **10000 bloques** | 1’37’’14 | 4’48’’86 | 3’42’’57 |

La co-simulación Hardware mejora los resultados obtenidos por ISIM para las simulaciones realizadas sobre 10000 bloques pero no es capaz de mejorar los resultados obtenidos por ModelSim SE. Esto se debe principalmente a que en el procesado de cada bloque de 512 bits se han añadido dieciséis ciclos para escribir los datos de entrada en la FIFO encargada de la comunicación entre el PC y la FPGA. Para la lectura de los datos procesados se requieren otros ocho ciclos extra. Este procesado añadido es ejecutado por Simulink provocando que en la ejecución que procesa 10000 bloques sea necesario ejecutar 240000 ciclos extra.

Para eliminar estos procesos, la escritura y lectura de las memorias destinadas a la comunicación que se ejecutan en el dominio del PC se deberían realizar por bloques de procesado no por muestras de 32bits. Para tal propósito, se deberían sustituir los componentes de tipo “Shared FIFO” por componentes de tipo “Shared Memory” ya que estos últimos permiten realizar este tipo de transferencias de bloque.

El precio del simulador ModelSim SE es bastante elevado, mientras que el simulador ISIM es proporcionado con el paquete de distribución de ISE sin ninguna licencia extra. En proyectos que no contemplen la inversión en el entorno ModelSim SE, la solución implementada puede resultar suficiente en términos de velocidad de procesamiento sin necesidad de realizar ninguna modificación.

En la siguiente tabla se aprecia la lógica incorporada por System Generator al código del algoritmo para poder llevar a cabo la co-simulación HW:

| SHA256 | **Algoritmo Base** | **Co-Simulación HW** |
|---|---|---|
| **Number of Slices** | 545 | 679 |
| **Number of BRAMs** | 0 | 8 |
| **Number of IOBs** | 73 | 51 |

Como se puede observar únicamente se añaden 134 slices al algoritmo. Se debe considerar que en diseños de mayor tamaño, la lógica que incorpora la co-simulación HW no se verá incrementada, ya que se debe en su mayoría al interfaz de comunicación Ethernet. Los datos obtenidos muestran que la Co-Simulación HW añade una serie de bloques de memoria. Estos bloques se incorporan debido a las implementaciones de los componentes de tipo “Shared Fifo” en la FPGA y una serie de memorias necesarias para implementar el interfaz Ethernet.

El número de IOBs indica el número de pines utilizado por el diseño. Como ya se comentó, en el caso del algoritmo aislado no es un dato a tener en cuenta, ya que en la implementación final que se realice dependerá del interfaz que se elija para comunicarse con la FPGA. En el caso de la co-simulación hardware si es un dato relevante, ya que indica los pines necesarios para establecer una conexión con la FPGA a través de Ethernet.

La siguiente tabla aporta una estimación del consumo añadido:

| SHA256 | **Algoritmo Base** | **Co-Simulación HW** |
|---|---|---|
| Clocks | 0.019W | 0.020W |
| Logic | 0.022W | 0.020W |
| Signals | 0.021W | 0.018W |
| IOs | 0.035W | 0.095W |
| Leakage | 2.708W | 2.710W |
| **Total** | 2.804W | 2.863W |

La tabla muestra los consumos debidos a alimentar la distribución de relojes, la lógica empleada, las señales que interconectan dicha lógica, los pines de entrada/salida y las pérdidas dentro del dispositivo. Como se puede observar las pérdidas son enormes en relación a los otros valores, esto se debe a que se está empleando una FPGA que sobredimensiona totalmente a los algoritmos implementados. Se debe alimentar a toda la FPGA aunque no se utilice gran parte de ella, por ello es fundamental escoger un dispositivo que sea congruente con el tamaño del diseño que se vaya a realizar. En dicho diseño puede que el algoritmo criptográfico sólo sea una pequeña parte de la implementación total. En el presente trabajo se ha escogido la FPGA Virtex6 240T por ser la FPGA incluida en la tarjeta de evaluación ML605.

En cuanto al consumo de la lógica asociada se observa que se ha reducido en la versión que incluye los componentes para la co-simulación HW. Esto se debe a las variaciones que existen en el posicionamiento e interconexión de componentes (Place&Route) entre dos implementaciones distintas y a que la lógica añadida para permitir la co-simulación HW es tan escasa que no compensa dichas variaciones.

Durante el desarrollo de los distintos Hitos que componen el presente Trabajo se ha demostrado la facilidad con la que la aplicación permite el uso de otros algoritmos criptográficos sin alterar su estructura. Para verificar el funcionamiento de otro algoritmo basta con sustituir el componente asociado (Black Box), permaneciendo inalterable la parte de gestión de datos de entrada/salida. De esta forma, la aplicación permite una gran reusabilidad del código facilitando además su mantenimiento.

# 8. Proyección de futuro

En este Trabajo se ha realizado una sencilla aplicación con el objetivo de mostrar las virtudes que proporciona la co-simulación HW en la verificación de diseños basados en FPGA e indicar los pasos a seguir para desarrollarla.

Nuevos algoritmos podrían ser añadidos sin realizar excesivas modificaciones. Por ejemplo, para el caso de cifradores de bloque, únicamente se requeriría modificar el tamaño del bloque a tratar e incluir unos nuevos componentes para trasmitir las claves.

En el apartado de “Conclusiones”, se propuso una mejora en la aplicación para reducir el número de ciclos de ejecución que requiere Simulink para el almacenamiento y lectura de las memorias de intercomunicación en el dominio del PC. Su implementación reduciría de forma considerable los tiempos empleados para la realización de la co-simulación HW.

La arquitectura desarrollada permite su utilización para verificar el comportamiento de diseños HW en los que se utilice una FPGA como co-procesador. De forma que el entorno en PC modele al microprocesador principal y la parte del co-procesador se ejecute en la FPGA. La co-simulación HW permite verificar un diseño basado en FPGA haciendo uso de una tarjeta de evaluación sin tener las restricciones impuestas por los periféricos que se hayan incluido en dicha tarjeta, ya que únicamente es necesario un interfaz Ethernet para realizar la transferencia de datos. De esta forma, se puede validar el diseño sin disponer de un prototipo hardware del equipo en el que se vaya a utilizar el algoritmo, reduciendo tiempos de desarrollo.

Se ha mostrado la posibilidad de generar vectores de entrada en la fase de pruebas mediante la potencia computacional de las herramientas Matlab/Simulink, así como la capacidad de analizar los resultados obtenidos. Podrían realizarse investigaciones utilizando este trabajo como referencia para añadir generación de datos y análisis de resultados de cualquier tipo. Una arquitectura como la planteada en este trabajo permitiría realizar criptoanálisis sobre los algoritmos implementados en FPGA con unos vectores de ataque creados mediante funciones desarrolladas utilizando Matlab. Esta solución además permite liberar al PC de la carga computacional asociada a la simulación del algoritmo.

# 9. Bibliografía/Referencias

[1] B. Song, K. Kawakami, K. Nakano and Y. Ito, — “An RSA Encryption Hardware Algorithm using a single DSP Block and single Block RAM on the FPGA”, In Proc. First International Conference on Networking and Computing, pp.140-147, 2010.

[2] R. V. Kshirsagar, M. V. Vyawahe. – “FPGA Implementation of High Speed VLSI Architectures for AES Algorithm”. In: ICETET, 2012

[3] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, —”Optimisation of the SHA-2 family of hash functions on FPGAs”, in Proc. IEEE Comput. Soc. Annu. Symp. Emerging VLSI Technol. Arch. (ISVLSI), pp. 317–322, 2006.

[4] V.C.Madhavi, K.Hanumantha Rao, P.Malyadri, G. Rama Krishna Prasad.-“NEW TECHNIQUES FOR HARDWARE IMPLEMENTATIONS OF SHA”, in INTERNATIONAL JOURNAL OF ENGINEERING SCIENCE & ADVANCED TECHNOLOGY Volume-2, Issue-3, 742 – 746, Jun.2012

[5] Alba M. Sánchez G., Ricardo Alvarez G., Sully Sánchez G, - “Architecture for filtering images using Xilinx System Generator”, in International journal of mathematics and computers in simulation, Issue 2, Volume 1, 2007, 101-107.

[6] Alia Arshad, Kanwal Aslam, Dur-e-Shahwar Kundi, Arshad Aziz. - “FPGA Implementation of Advance Encryption Standard Using Xilinx System Generator”, in ASIAN JOURNAL OF APPLIED SCIENCES, Vol 2, No 2 (2014).

[7] Panduranga H T, Dr. Naveen Kumar S K and Sharath Kumar H S. – “Hardware Software Co-Simulation of the Multiple Image Encryption Technique Using the Xilinx System Generator”. J Inf Process Syst, Vol.9, No.3, September 2013.

[8] Daniel Denning, Malachy Devlin, James Irvine (Institute of System Level Integration) – “Hardware Co-Simulation in System Generator of the AES-128 Encryption Algorithm”; In Proceeding FPGA '04 Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays Pages 247-247.

[9] Xilinx. System Generator for DSP User Guide. UG640 (v 12.3). September 21, 2010

[10] Xilinx. Getting Started with System Generator. Online Training.

http://www.xilinx.com/training/dsp/system-generator-training-video.htm

[11]. Xilinx. System Generator for DSP Getting Started Guide. UG639 (v 12.3) September 21, 2010.

[12] Mathworks. Documentation Center. Simulink. Recuperado el 26/05/2014 de

http://www.mathworks.es/es/help/simulink/

[13] Arif Endro Nugroho, Jun 25 2010. Nugroho Free Hash Cores :: Overview. Opencores. Recuperado el 30/05/2014 de http://opencores.org/project,nfhc

[14] Antonio de La Piedra, 2013. SHA-256 Core :: Overview. Opencores. Recuperado el 30/05/2014 de http://opencores.org/project,sha256core

[15] Descriptions of SHA-256, SHA-384, and SHA-512. Autor desconocido.

http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

[16] Orr Dunkelman. Computer Science Department. “Hash Functions — MD5 and SHA1”.

14 March, 2012.

[17] Scott Sirowy, Alessandro Forin. Microsoft Research. – “Where’s the Beef? Why FPGAs Are So Fast”, September 2008

[18] David B. Thomas, Lee Howes and Wayne Luk - A comparison of CPUs, GPUs, FPGAs and massively parallel processor arrays for random number generation. Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays. New York: ACM; 2009.

[19] Christopher Cullinan,Christopher Wyant, Timothy Frattesi – “Computing Performance Benchmarks among CPU, GPU, and FPGA”. Mathworks.

[20] Vivek Venugopal, Devu Manikantan Shila. – “High Throughput Implementations of Cryptography algorithms on GPU and FPGA”, in Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International.

[21] Texas Instruments. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. SPRU732J. July 2010..

[22] Universidad Internacional de la Rioja (2013). Tema2 de la asignatura Criptografía y Mecanismos de Seguridad del Máster Universitario en Seguridad Informática. Material no publicado.

[23] Xilinx. System Generator for DSP Reference Guide. UG639 (v 13.4) January 18, 2012.

[24] Implement S-Functions. Ayuda online de Mathworks. Recuperado el 26/05/2014 de

http://www.mathworks.es/es/help/simulink/sfg/how-to-implement-s-functions.html

[25] S-Function Concepts. Ayuda online de Mathworks. Recuperado el 26/05/2014 de

http://www.mathworks.es/es/help/simulink/sfg/s-function-concepts.html

[26] Jan Simon (2012). Matlab Central. File Exchange. Recuperado el 24/05/2014 de http://www.mathworks.com/matlabcentral/fileexchange/31272-datahash y

http://www.mathworks.com/matlabcentral/fileexchange/31795-sha-algorithms-160-224-256-384-512

[27] Generadores Online Hash. Online Convert.com. Recuperado el 24/05/2014 de

http://hash.online-convert.com/es/generador-sha512

http://hash.online-convert.com/es/generador-sha256

http://hash.online-convert.com/es/generador-sha1

# 10. Anexo I. Código.

A continuación se incluirá código que se ha considerado relevante para comprender el funcionamiento de la aplicación. El resto de ficheros utilizados por la aplicación, pueden ser consultados en el archivo “CódigoAplicacion.zip” que los contiene.

## 10.1. Funciones Matlab

### 10.1.1. “read_input_file.m”

```
function read_input_file(block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.


setup(block);
%endfunction

function setup(block)

% Register number of ports
block.NumInputPorts  = 3;
block.NumOutputPorts = 6;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%new_msg_i
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;
block.InputPort(1).SamplingMode = 0;

%read_file_i
block.InputPort(2).Dimensions  = 1;
block.InputPort(2).DatatypeID  = 0;  % double
block.InputPort(2).Complexity  = 'Real';
block.InputPort(2).DirectFeedthrough = true;
block.InputPort(2).SamplingMode = 0;
```

```
%read_addrs
block.InputPort(3).Dimensions  = 1;
block.InputPort(3).DatatypeID  = 0;  % double
block.InputPort(3).Complexity  = 'Real';
block.InputPort(3).DirectFeedthrough = true;
block.InputPort(3).SamplingMode = 0;

% Override output port properties
%new_msg_o;
block.OutputPort(1).Dimensions  = 1;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode = 0;

%new_block_o
block.OutputPort(2).Dimensions  = 1;
block.OutputPort(2).DatatypeID  = 0; % double
block.OutputPort(2).Complexity  = 'Real';
block.OutputPort(2).SamplingMode = 0;

%data0
block.OutputPort(3).Dimensions  = 1;
block.OutputPort(3).DatatypeID  = 0;
block.OutputPort(3).Complexity  = 'Real';
block.OutputPort(3).SamplingMode = 0;
%data1
block.OutputPort(4).Dimensions  = 1;
block.OutputPort(4).DatatypeID  = 0;
block.OutputPort(4).Complexity  = 'Real';
block.OutputPort(4).SamplingMode = 0;
%data2
block.OutputPort(5).Dimensions  = 1;
block.OutputPort(5).DatatypeID  = 0;
block.OutputPort(5).Complexity  = 'Real';
block.OutputPort(5).SamplingMode = 0;
%data3
block.OutputPort(6).Dimensions  = 1;
block.OutputPort(6).DatatypeID  = 0;
block.OutputPort(6).Complexity  = 'Real';
block.OutputPort(6).SamplingMode = 0;

% Register parameters
block.NumDialogPrms     = 1;

% Register sample times
%  [0 offset]        : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]         : Inherited sample time
%  [-2, 0]         : Variable sample time
block.SampleTimes = [1 0];

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);     % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction


function DoPostPropSetup(block)

block.NumDworks = 3;

block_size = block.DialogPrm(1).Data/8;
```

```
block.Dwork(1).Name = 'file';
block.Dwork(1).Dimensions      = 1;
block.Dwork(1).DatatypeID      = 0;
block.Dwork(1).Complexity      = 'Real';
block.Dwork(1).UsedAsDiscState = true;

block.Dwork(2).Name = 'input_text';
block.Dwork(2).Dimensions      = block_size; %frame_size in bytes
block.Dwork(2).DatatypeID      = 0;
block.Dwork(2).Complexity      = 'Real';
block.Dwork(2).UsedAsDiscState = true;

block.Dwork(3).Name = 'new_msg';
block.Dwork(3).Dimensions      = 1;
block.Dwork(3).DatatypeID      = 0;
block.Dwork(3).Complexity      = 'Real';
block.Dwork(3).UsedAsDiscState = true;

block.AutoRegRuntimePrms;

%endfunction
function Start(block)
block.Dwork(1).Data = fopen('input_file.txt','r'); %Open Input File
block.Dwork(3).Data = 0;
%endfunction

function Outputs(block)
%Set intermediate variables
fileID = block.Dwork(1).Data; %File identifier
new_msg_i = block.InputPort(1).Data; %Start new frame input
new_msg_o = block.Dwork(3).Data; %Start new frame output and it controls if the file is already opened
cnt_idx = (block.InputPort(3).Data*4) + 1; %Input address for memory containing input samples
read_file = block.InputPort(2).Data;

%New frame arrives
if((new_msg_i == 1 ) && (new_msg_o == 0))
        %Reading bytes from input file
block.Dwork(2).Data = fread(fileID, block.Dwork(2).Dimensions,'uint8');
%Dwork(2).Data,LengthFile] = fread(fileID,'uint8'); %Use this sentence if you want to know the number of read bytes
        new_msg_o = 1;%Set new message signal to algorithm
end

if (read_file == 1)    %algorithm is waiting for new samples
block.OutputPort(3).Data = block.Dwork(2).Data(cnt_idx);%Provide new samples to algorithm
        block.OutputPort(4).Data = block.Dwork(2).Data(cnt_idx+1);
        block.OutputPort(5).Data = block.Dwork(2).Data(cnt_idx+2);
        block.OutputPort(6).Data = block.Dwork(2).Data(cnt_idx+3);
end

%Set static variables and outputs.
block.Dwork(3).Data = new_msg_o;
block.OutputPort(1).Data = block.Dwork(3).Data;
block.OutputPort(2).Data = 0;
%endfunction


function Terminate(block)
fileID = block.Dwork(1).Data;
if fileID      %Check if file was opened to prevent execution errors
        fclose(fileID);
end
%endfunction
```

### 10.1.2. "write_output_file.m"

```
function write_output_file(block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.
setup(block);

%endfunction

function setup(block)

% Register number of ports
block.NumInputPorts  = 5;
block.NumOutputPorts = 0;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%valid_input
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;

%hash_value0
block.InputPort(2).Dimensions  = 1;
block.InputPort(2).DatatypeID  = 0; % double
block.InputPort(2).Complexity  = 'Real';
block.InputPort(2).DirectFeedthrough = true;

%hash_value1
block.InputPort(3).Dimensions  = 1;
block.InputPort(3).DatatypeID  = 0; % double
block.InputPort(3).Complexity  = 'Real';
block.InputPort(3).DirectFeedthrough = true;

%hash_value2
block.InputPort(4).Dimensions  = 1;
block.InputPort(4).DatatypeID  = 0; % double
block.InputPort(4).Complexity  = 'Real';
block.InputPort(4).DirectFeedthrough = true;

%hash_value3
block.InputPort(5).Dimensions  = 1;
block.InputPort(5).DatatypeID  = 0; % double
block.InputPort(5).Complexity  = 'Real';
block.InputPort(5).DirectFeedthrough = true;

% Register parameters
block.NumDialogPrms     = 0;
```

```
% Register sample times
%  [0 offset]            : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]               : Inherited sample time
%  [-2, 0]               : Variable sample time
block.SampleTimes = [-1 0];

block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);     % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction


function DoPostPropSetup(block)
  block.NumDworks = 1;
  block.Dwork(1).Name = 'file'; %File Identifier
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  block.AutoRegRuntimePrms;
%endfunction

function Start(block)
block.Dwork(1).Data = fopen('output_hash.txt','w'); %Open file for writing
%endfunction

function Outputs(block)

file=block.Dwork(1).Data;

if (block.InputPort(1).Data == 1)     %valid_data
fprintf(file,'%2x%2x%2x%2x', block.InputPort(2).Data, block.InputPort(3).Data,
block.InputPort(4).Data, block.InputPort(5).Data);
end
%endfunction


function Terminate(block)

file = block.Dwork(1).Data;
if (file)       %Check if file was opened to prevent execution errors
       fclose(file);
end
%endfunction
```

# 10.1.3. “read_input_file_sharedFifos.m”

```
function read_input_file_sharedFifos(block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.
setup(block);
%endfunction

function setup(block)

% Register number of ports
block.NumInputPorts  = 1;
block.NumOutputPorts = 7;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%new_msg_i
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;
block.InputPort(1).SamplingMode = 0;


% Override output port properties
%new_msg_o;
block.OutputPort(1).Dimensions  = 1;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode = 0;

%new_block_o
block.OutputPort(2).Dimensions  = 1;
block.OutputPort(2).DatatypeID  = 0; % double
block.OutputPort(2).Complexity  = 'Real';
block.OutputPort(2).SamplingMode = 0;

%data0
block.OutputPort(3).Dimensions  = 1;
block.OutputPort(3).DatatypeID  = 0;
block.OutputPort(3).Complexity  = 'Real';
block.OutputPort(3).SamplingMode = 0;
%data1
block.OutputPort(4).Dimensions  = 1;
block.OutputPort(4).DatatypeID  = 0;
block.OutputPort(4).Complexity  = 'Real';
block.OutputPort(4).SamplingMode = 0;
%data2
block.OutputPort(5).Dimensions  = 1;
block.OutputPort(5).DatatypeID  = 0;
block.OutputPort(5).Complexity  = 'Real';
```

```
block.OutputPort(5).SamplingMode = 0;
%data3
block.OutputPort(6).Dimensions  = 1;
block.OutputPort(6).DatatypeID  = 0;
block.OutputPort(6).Complexity  = 'Real';
block.OutputPort(6).SamplingMode = 0;
%valid_data
block.OutputPort(7).Dimensions  = 1;
block.OutputPort(7).DatatypeID  = 0;
block.OutputPort(7).Complexity  = 'Real';
block.OutputPort(7).SamplingMode = 0;

% Register parameters
block.NumDialogPrms     = 1;

% Register sample times
%  [0 offset]        : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]         : Inherited sample time
%  [-2, 0]         : Variable sample time
block.SampleTimes = [1 0];

block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);     % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction


function DoPostPropSetup(block)

  block.NumDworks = 5;
  block_size = (block.DialogPrm(1).Data/8);

  block.Dwork(1).Name = 'file';
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  block.Dwork(2).Name = 'input_text';
  block.Dwork(2).Dimensions      = block_size;%frame_size in bytes
  block.Dwork(2).DatatypeID      = 0;
  block.Dwork(2).Complexity      = 'Real';
  block.Dwork(2).UsedAsDiscState = true;

  block.Dwork(3).Name = 'reading';
  block.Dwork(3).Dimensions      = 1;
  block.Dwork(3).DatatypeID      = 0;
  block.Dwork(3).Complexity      = 'Real';
  block.Dwork(3).UsedAsDiscState = true;

  block.Dwork(4).Name = 'cnt_idx';
  block.Dwork(4).Dimensions      = 1;
  block.Dwork(4).DatatypeID      = 0; %uint32
  block.Dwork(4).Complexity      = 'Real';
  block.Dwork(4).UsedAsDiscState = true;

  block.Dwork(5).Name = 'opened_file';
  block.Dwork(5).Dimensions      = 1;
  block.Dwork(5).DatatypeID      = 0;
  block.Dwork(5).Complexity      = 'Real';
  block.Dwork(5).UsedAsDiscState = true;

  block.AutoRegRuntimePrms;
```

```
%endfunction

function Start(block)
block.Dwork(1).Data = fopen('input_file.txt','r'); %Open Input File
block.Dwork(3).Data = 0;
block.Dwork(4).Data = 1;
block.Dwork(5).Data = 0;
%endfunction

function Outputs(block)
%Set intermediate variables
fileID = block.Dwork(1).Data; %File identifier
new_frame = block.InputPort(1).Data;  %Start new frame input
reading = block.Dwork(3).Data;  % The process is reading input samples from memory
cnt_idx = block.Dwork(4).Data;  %Address for memory containing input samples
opened_file = block.Dwork(5).Data;%it controls if the file is already opened
length_file = block.Dwork(2).Dimensions;

%New frame
if((new_frame == 1 ) && (opened_file == 0))
        %Reading bytes from input file
block.Dwork(2).Data = fread(fileID, block.Dwork(2).Dimensions,'uint8');
    opened_file = 1;
    cnt_idx = 1;
    reading = 1;
end

if (cnt_idx >= (length_file))
    new_frame_o = 1;
    reading = 0;
else
    new_frame_o = 0;
end

if (reading == 1)      %algorithm is waiting for new samples
   block.OutputPort(3).Data = block.Dwork(2).Data(cnt_idx);
   block.OutputPort(4).Data = block.Dwork(2).Data(cnt_idx+1);
   block.OutputPort(5).Data = block.Dwork(2).Data(cnt_idx+2);
   block.OutputPort(6).Data = block.Dwork(2).Data(cnt_idx+3);
   cnt_idx = cnt_idx + 4;
end

%Set static variables and outputs.
block.OutputPort(1).Data = new_frame_o;
block.OutputPort(2).Data = 0;
block.Dwork(3).Data = reading;
block.Dwork(4).Data = cnt_idx;
block.Dwork(5).Data = opened_file;
block.OutputPort(7).Data = reading;
%endfunction

function Terminate(block)
fileID=block.Dwork(1).Data;
if fileID      %Check if file was opened to prevent execution errors
fclose(fileID);
end
%endfunction
```

## 10.1.4. “write_output_file_sharedFifos.m”

```
function write_output_file_sharedFifos(block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.
setup(block);

%endfunction

function setup(block)

% Register number of ports
block.NumInputPorts  = 7;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%valid_input
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;

%hash_value0
block.InputPort(2).Dimensions  = 1;
block.InputPort(2).DatatypeID  = 0; % double
block.InputPort(2).Complexity  = 'Real';
block.InputPort(2).DirectFeedthrough = true;

%hash_value1
block.InputPort(3).Dimensions  = 1;
block.InputPort(3).DatatypeID  = 0; % double
block.InputPort(3).Complexity  = 'Real';
block.InputPort(3).DirectFeedthrough = true;

%hash_value2
block.InputPort(4).Dimensions  = 1;
block.InputPort(4).DatatypeID  = 0; % double
block.InputPort(4).Complexity  = 'Real';
block.InputPort(4).DirectFeedthrough = true;

%hash_value3
block.InputPort(5).Dimensions  = 1;
block.InputPort(5).DatatypeID  = 0; % double
block.InputPort(5).Complexity  = 'Real';
block.InputPort(5).DirectFeedthrough = true;

%full_fifo
block.InputPort(6).Dimensions  = 1;
block.InputPort(6).DatatypeID  = 0; % double
block.InputPort(6).Complexity  = 'Real';
```

```
block.InputPort(6).DirectFeedthrough = true;

%empty_fifo
block.InputPort(7).Dimensions  = 1;
block.InputPort(7).DatatypeID  = 0; % double
block.InputPort(7).Complexity  = 'Real';
block.InputPort(7).DirectFeedthrough = true;

%read_data
block.OutputPort(1).Dimensions  = 1;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode = 0;

% Register parameters
block.NumDialogPrms     = 0;


% Register sample times
%  [0 offset]        : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]         : Inherited sample time
%  [-2, 0]         : Variable sample time
block.SampleTimes = [-1 0];

block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);      % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction


function DoPostPropSetup(block)
  block.NumDworks = 2;
  block.Dwork(1).Name = 'file'; %File Identifier
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  block.Dwork(2).Name = 'reading'; %Reading samples from output Fifo  to write them in a file
  block.Dwork(2).Dimensions      = 1;
  block.Dwork(2).DatatypeID      = 0;
  block.Dwork(2).Complexity      = 'Real';
  block.Dwork(2).UsedAsDiscState = true;

  block.AutoRegRuntimePrms;
%endfunction

function Start(block)
block.Dwork(1).Data = fopen('output_hash.txt','w');%Open file for writing
block.Dwork(2).Data = 0;
%endfunction

function Outputs(block)

file = block.Dwork(1).Data;   %File ID
fifo_full = block.InputPort(6).Data; %Full percentage
valid_data = block.InputPort(1).Data; %Input data validation
reading = block.Dwork(2).Data;%Reading samples from output Fifo  to write them in a file
fifo_empty = block.InputPort(7).Data; %Empty field from FIFO

if (valid_data == 1)   %Write input data to file
```

```
fprintf(file,'%2x%2x%2x%2x', block.InputPort(2).Data, block.InputPort(3).Data,
block.InputPort(4).Data, block.InputPort(5).Data);
end

if (fifo_full >= 0.5) %Check if FIFO is almost full
   reading = 1;       %If it is almost full then FIFO is ready to be read
else
   if (fifo_empty == 1)
    reading = 0;      %Stop reading when fifo is empty
   end
end
block.Dwork(2).Data = reading;
block.OutputPort(1).Data = reading;
%endfunction

function Terminate(block)

file = block.Dwork(1).Data;
if (file)      %Check if file was opened to prevent execution errors
    fclose(file);
end
%endfunction
```

## 10.1.5. “read_input_file_sharedFifos_h2.m”

```
function read_input_file_sharedFifos_h2(block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.

setup(block);
%endfunction

function setup(block)

% Register number of ports
block.NumInputPorts  = 2;
block.NumOutputPorts = 7;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%new_msg_i
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;
block.InputPort(1).SamplingMode = 0;

%new_blk_i
block.InputPort(2).Dimensions  = 1;
block.InputPort(2).DatatypeID  = 0;  % double
```

```
block.InputPort(2).Complexity  = 'Real';
block.InputPort(2).DirectFeedthrough = true;
block.InputPort(2).SamplingMode = 0;

% Override output port properties
%new_msg_o;
block.OutputPort(1).Dimensions  = 1;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode = 0;

%new_block_o
block.OutputPort(2).Dimensions  = 1;
block.OutputPort(2).DatatypeID  = 0; % double
block.OutputPort(2).Complexity  = 'Real';
block.OutputPort(2).SamplingMode = 0;

%data0
block.OutputPort(3).Dimensions  = 1;
block.OutputPort(3).DatatypeID  = 0;
block.OutputPort(3).Complexity  = 'Real';
block.OutputPort(3).SamplingMode = 0;
%data1
block.OutputPort(4).Dimensions  = 1;
block.OutputPort(4).DatatypeID  = 0;
block.OutputPort(4).Complexity  = 'Real';
block.OutputPort(4).SamplingMode = 0;
%data2
block.OutputPort(5).Dimensions  = 1;
block.OutputPort(5).DatatypeID  = 0;
block.OutputPort(5).Complexity  = 'Real';
block.OutputPort(5).SamplingMode = 0;
%data3
block.OutputPort(6).Dimensions  = 1;
block.OutputPort(6).DatatypeID  = 0;
block.OutputPort(6).Complexity  = 'Real';
block.OutputPort(6).SamplingMode = 0;
%valid_data
block.OutputPort(7).Dimensions  = 1;
block.OutputPort(7).DatatypeID  = 0;
block.OutputPort(7).Complexity  = 'Real';
block.OutputPort(7).SamplingMode = 0;

% Register parameters
block.NumDialogPrms     = 1;

% Register sample times
%  [0 offset]        : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]         : Inherited sample time
%  [-2, 0]         : Variable sample time
block.SampleTimes = [1 0];

block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);     % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction

function DoPostPropSetup(block)

  block.NumDworks = 6;
  block_size = (block.DialogPrm(1).Data/8);
```

```
  block.Dwork(1).Name = 'file';
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  block.Dwork(2).Name = 'input_text';
  block.Dwork(2).Dimensions      = block_size; %frame_size in bytes
  block.Dwork(2).DatatypeID      = 0;
  block.Dwork(2).Complexity      = 'Real';
  block.Dwork(2).UsedAsDiscState = true;

  block.Dwork(3).Name = 'reading';
  block.Dwork(3).Dimensions      = 1;
  block.Dwork(3).DatatypeID      = 0;
  block.Dwork(3).Complexity      = 'Real';
  block.Dwork(3).UsedAsDiscState = true;

  block.Dwork(4).Name = 'cnt_idx';
  block.Dwork(4).Dimensions      = 1;
  block.Dwork(4).DatatypeID      = 0;
  block.Dwork(4).Complexity      = 'Real';
  block.Dwork(4).UsedAsDiscState = true;

  block.Dwork(5).Name = 'opened_file';
  block.Dwork(5).Dimensions      = 1;
  block.Dwork(5).DatatypeID      = 0;
  block.Dwork(5).Complexity      = 'Real';
  block.Dwork(5).UsedAsDiscState = true;

  block.Dwork(6).Name = 'first_block';
  block.Dwork(6).Dimensions      = 1;
  block.Dwork(6).DatatypeID      = 0;
  block.Dwork(6).Complexity      = 'Real';
  block.Dwork(6).UsedAsDiscState = true;

  block.AutoRegRuntimePrms;

%endfunction

function Start(block)
block.Dwork(1).Data = fopen('input_file.txt','r');  %Open Input File
block.Dwork(3).Data = 0;
block.Dwork(4).Data = 1;
block.Dwork(5).Data = 0;
block.Dwork(6).Data = 1;
%endfunction

function Outputs(block)
%Set intermediate variables
fileID = block.Dwork(1).Data; %File identifier
new_frame = block.InputPort(1).Data;  %Start new frame input
reading = block.Dwork(3).Data;  %The process is reading input samples from memory
cnt_idx = block.Dwork(4).Data;  %Address for memory containing input samples
block_size = block.Dwork(2).Dimensions;  %Block size which can vary depending on algorithm
new_blk = block.InputPort(2).Data;  %Start new block
first_block = block.Dwork(6).Data;  %It is the first block of the message?

%New frame
if((new_frame == 1 ) || (new_blk == 1))
    %Reading bytes from input file
    block.Dwork(2).Data = fread(fileID, block.Dwork(2).Dimensions,'uint8');
    new_block_o = 0;
    cnt_idx = 1;
    reading = 1;%Start reading process
    if (new_frame == 1 )
        first_block = 1;  %It is the first block of the new message
    end
end
```

```
if (cnt_idx >= block_size) %A new block is completed
        if (first_block == 1)
            new_frame_o = 1; %Generate New Message Signal the first time
            new_block_o = 0;
        else
            new_frame_o = 0;
            new_block_o = 1; %Generate New Block Signal
        end
        first_block = 0;
    reading = 0;
    cnt_idx = 1;
else
    new_frame_o = 0;
    new_block_o = 0;
end

if (reading == 1)      %algorithm is waiting for new samples
        block.OutputPort(3).Data = block.Dwork(2).Data(cnt_idx);
        block.OutputPort(4).Data = block.Dwork(2).Data(cnt_idx+1);
        block.OutputPort(5).Data = block.Dwork(2).Data(cnt_idx+2);
        block.OutputPort(6).Data = block.Dwork(2).Data(cnt_idx+3);
        cnt_idx = cnt_idx + 4;
end

%Set static variables and outputs.
block.OutputPort(1).Data = new_frame_o;
block.OutputPort(2).Data = new_block_o;
block.Dwork(3).Data = reading;
block.Dwork(4).Data = cnt_idx;
block.OutputPort(7).Data = reading;
block.Dwork(6).Data = first_block;

%endfunction


function Terminate(block)
fileID=block.Dwork(1).Data;
if fileID      %Check if file was opened to prevent execution errors
   fclose(fileID);
end
%endfunction
```

## 10.1.6. “write_output_file_sharedFifos_h2.m”

```
function write_output_file_sharedFifos_h2 (block)
%MSFUNTMPL_BASIC A template for a Leve-2 M-file S-function
%   The M-file S-function is written as a MATLAB function with the
%   same name as the S-function. Replace 'msfuntmpl_basic' with the
%   name of your S-function.
%
%   It should be noted that the M-file S-function is very similar
%   to Level-2 C-Mex S-functions. You should be able to get more
%   information for each of the block methods by referring to the
%   documentation for C-Mex S-functions.
%
%   Copyright 2003-2007 The MathWorks, Inc.
setup(block);

%endfunction

function setup(block)
```

```
% Register number of ports
block.NumInputPorts  = 8;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
%valid_input
block.InputPort(1).Dimensions  = 1;
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
block.InputPort(1).DirectFeedthrough = true;

%hash_value0
block.InputPort(2).Dimensions  = 1;
block.InputPort(2).DatatypeID  = 0; % double
block.InputPort(2).Complexity  = 'Real';
block.InputPort(2).DirectFeedthrough = true;

%hash_value1
block.InputPort(3).Dimensions  = 1;
block.InputPort(3).DatatypeID  = 0; % double
block.InputPort(3).Complexity  = 'Real';
block.InputPort(3).DirectFeedthrough = true;

%hash_value2
block.InputPort(4).Dimensions  = 1;
block.InputPort(4).DatatypeID  = 0; % double
block.InputPort(4).Complexity  = 'Real';
block.InputPort(4).DirectFeedthrough = true;

%hash_value3
block.InputPort(5).Dimensions  = 1;
block.InputPort(5).DatatypeID  = 0; % double
block.InputPort(5).Complexity  = 'Real';
block.InputPort(5).DirectFeedthrough = true;

%full_fifo%
block.InputPort(6).Dimensions  = 1;
block.InputPort(6).DatatypeID  = 0; % double
block.InputPort(6).Complexity  = 'Real';
block.InputPort(6).DirectFeedthrough = true;

%empty_fifo
block.InputPort(7).Dimensions  = 1;
block.InputPort(7).DatatypeID  = 0; % double
block.InputPort(7).Complexity  = 'Real';
block.InputPort(7).DirectFeedthrough = true;

%Open file for rewriting process
block.InputPort(8).Dimensions  = 1;
block.InputPort(8).DatatypeID  = 0; % double
block.InputPort(8).Complexity  = 'Real';
block.InputPort(8).DirectFeedthrough = true;

%read_data
block.OutputPort(1).Dimensions  = 1;
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';
block.OutputPort(1).SamplingMode = 0;

% Register parameters
block.NumDialogPrms     = 0;
```

```
% Register sample times
%  [0 offset]        : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]          : Inherited sample time
%  [-2, 0]          : Variable sample time
block.SampleTimes = [-1 0];

block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
block.RegBlockMethod('Start', @Start);

block.RegBlockMethod('Outputs', @Outputs);      % Required
block.RegBlockMethod('Terminate', @Terminate); % Required

%end setup
%endfunction


function DoPostPropSetup(block)

  block.NumDworks = 2;
  block.Dwork(1).Name = 'file'; %File Identifier
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  block.Dwork(2).Name = 'reading'; %Reading samples from output Fifo  to write them in a file
  block.Dwork(2).Dimensions      = 1;
  block.Dwork(2).DatatypeID      = 0;
  block.Dwork(2).Complexity      = 'Real';
  block.Dwork(2).UsedAsDiscState = true;

  block.AutoRegRuntimePrms;
%endfunction

function Start(block)
block.Dwork(1).Data = fopen('output_hash.txt','w');%Open file for writing
block.Dwork(2).Data = 0;
%endfunction

function Outputs(block)

file = block.Dwork(1).Data;   %File ID
fifo_full = block.InputPort(6).Data; %Full percentage
valid_data = block.InputPort(1).Data; %Input data validation
reading = block.Dwork(2).Data;        %Reading samples from output Fifo  to write them in a file
fifo_empty = block.InputPort(7).Data; %Empty field from FIFO
new_block = block.InputPort(8).Data;

if (new_block ==1)
        fclose(file);
        %Begin the file with a new hash
        block.Dwork(1).Data = fopen('output_hash.txt','w');%Open file for writing
end

if (valid_data == 1) %Write input data to file
    fprintf(file,'%2x%2x%2x%2x', block.InputPort(2).Data, block.InputPort(3).Data,
block.InputPort(4).Data, block.InputPort(5).Data);
end

if (fifo_full >= 0.5) %Check if FIFO is almost full
        reading = 1;   %If it is almost full then FIFO is ready to be read
else
   if (fifo_empty == 1)
    reading = 0;       %Stop reading when fifo is empty
   end
```

```
end
block.Dwork(2).Data = reading;
block.OutputPort(1).Data = reading;
%endfunction


function Terminate(block)

file = block.Dwork(1).Data;
if (file)       %Check if file was opened to prevent execution errors
        fclose(file);
end
%endfunction
```

## 10.1.7. “sha256_core_config.m”

```
function sha256_core_config(this_block)

  % Revision History:
  %
  %   13-Jul-2014  (20:15 hours):
  %     Original code was machine generated by Xilinx's System Generator after parsing
  %     ..\..\rtl\sha_256.vhd
  %
  %

  this_block.setTopLevelLanguage('VHDL');

  this_block.setEntityName('sha256_core');

  % System Generator has to assume that your entity  has a combinational feed through;
  %   if it  doesn't, then comment out the following line:
  this_block.tagAsCombinational;

  this_block.addSimulinkInport('new_msg_i');
  this_block.addSimulinkInport('new_blk_i');
  this_block.addSimulinkInport('block0_i');
  this_block.addSimulinkInport('block1_i');
  this_block.addSimulinkInport('block2_i');
  this_block.addSimulinkInport('block3_i');

  this_block.addSimulinkOutport('a_o');
  this_block.addSimulinkOutport('b_o');
  this_block.addSimulinkOutport('c_o');
  this_block.addSimulinkOutport('d_o');
  this_block.addSimulinkOutport('e_o');
  this_block.addSimulinkOutport('f_o');
  this_block.addSimulinkOutport('g_o');
  this_block.addSimulinkOutport('h_o');
  this_block.addSimulinkOutport('state_o');
  this_block.addSimulinkOutport('read_txt_o');
  this_block.addSimulinkOutport('read_addrs_o');
  this_block.addSimulinkOutport('valid_hash_o');
  this_block.addSimulinkOutport('hash_o');

  a_o_port = this_block.port('a_o');
  a_o_port.setType('UFix_32_0');
  b_o_port = this_block.port('b_o');
  b_o_port.setType('UFix_32_0');
  c_o_port = this_block.port('c_o');
  c_o_port.setType('UFix_32_0');
  d_o_port = this_block.port('d_o');
  d_o_port.setType('UFix_32_0');
  e_o_port = this_block.port('e_o');
  e_o_port.setType('UFix_32_0');
```

```
f_o_port = this_block.port('f_o');
f_o_port.setType('UFix_32_0');
g_o_port = this_block.port('g_o');
g_o_port.setType('UFix_32_0');
h_o_port = this_block.port('h_o');
h_o_port.setType('UFix_32_0');
state_o_port = this_block.port('state_o');
state_o_port.setType('UFix_5_0');
read_txt_o_port = this_block.port('read_txt_o');
read_txt_o_port.setType('UFix_1_0');
read_txt_o_port.useHDLVector(false);
read_addrs_o_port = this_block.port('read_addrs_o');
read_addrs_o_port.setType('UFix_4_0');
valid_hash_o_port = this_block.port('valid_hash_o');
valid_hash_o_port.setType('UFix_1_0');
valid_hash_o_port.useHDLVector(false);
hash_o_port = this_block.port('hash_o');
hash_o_port.setType('UFix_32_0');

% ----------------------------
if (this_block.inputTypesKnown)
  % do input type checking, dynamic output type and generic setup in this code block.

  if (this_block.port('new_msg_i').width ~= 1);
    this_block.setError('Input data type for port "new_msg_i" must have width=1.');
  end

  this_block.port('new_msg_i').useHDLVector(false);

  if (this_block.port('new_blk_i').width ~= 1);
    this_block.setError('Input data type for port "new_blk_i" must have width=1.');
  end

  this_block.port('new_blk_i').useHDLVector(false);

  if (this_block.port('block0_i').width ~= 8);
    this_block.setError('Input data type for port "block0_i" must have width=8.');
  end

  if (this_block.port('block1_i').width ~= 8);
    this_block.setError('Input data type for port "block1_i" must have width=8.');
  end

  if (this_block.port('block2_i').width ~= 8);
    this_block.setError('Input data type for port "block2_i" must have width=8.');
  end

  if (this_block.port('block3_i').width ~= 8);
    this_block.setError('Input data type for port "block3_i" must have width=8.');
  end

end   % if(inputTypesKnown)
% ----------------------------

% ----------------------------
 if (this_block.inputRatesKnown)
   setup_as_single_rate(this_block,'clk_i','ce_i')
 end   % if(inputRatesKnown)
% ----------------------------

  % (!) Set the inout port rate to be the same as the first input
  %     rate. Change the following code if this is untrue.
  uniqueInputRates = unique(this_block.getInputRates);


% Add addtional source files as needed.
%  |-------------
%  | Add files in the order in which they should be compiled.
%  | If two files "a.vhd" and "b.vhd" contain the entities
```

```
  %  | entity_a and entity_b, and entity_a contains a
  %  | component of type entity_b, the correct sequence of
  %  | addFile() calls would be:
  %  |    this_block.addFile('b.vhd');
  %  |    this_block.addFile('a.vhd');
  %  |-------------

  %    this_block.addFile('');
  %    this_block.addFile('');
  this_block.addFile('..\..\rtl\sha_256.vhd');
  this_block.addFile('..\..\rtl\dual_mem.vhd');
  this_block.addFile('..\..\rtl\ff_bank.vhd');
  this_block.addFile('..\..\rtl\msg_comp.vhd');
  this_block.addFile('..\..\rtl\sh_reg.vhd');

return;


% ------------------------------------------------------------

function setup_as_single_rate(block,clkname,cename)
  inputRates = block.inputRates;
  uniqueInputRates = unique(inputRates);
  if (length(uniqueInputRates)==1 & uniqueInputRates(1)==Inf)
    block.addError('The inputs to this block cannot all be constant.');
    return;
  end
  if (uniqueInputRates(end) == Inf)
     hasConstantInput = true;
     uniqueInputRates = uniqueInputRates(1:end-1);
  end
  if (length(uniqueInputRates) ~= 1)
    block.addError('The inputs to this block must run at a single rate.');
    return;
  end
  theInputRate = uniqueInputRates(1);
  for i = 1:block.numSimulinkOutports
     block.outport(i).setRate(theInputRate);
  end
  block.addClkCEPair(clkname,cename,theInputRate);
  return;

% ------------------------------------------------------------
```

# 10.2. Código VHDL

## 10.2.1. Ficheros Implementación

### 10.2.1.1. Sha256.vhd

```
-- Copyright (c) 2013 Antonio de la Piedra

-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.

-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.

-- You should have received a copy of the GNU General Public License
-- along with this program.  If not, see <http://www.gnu.org/licenses/>.

--This file has been modified by Guillermo Fernandez to allow the integration
--with an application developed using System Generator.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sha256_core is
        port(
                clk_i  : in std_logic;
                ce_i : in std_logic;
                new_msg_i : in std_logic;
                new_blk_i : in std_logic;

                block0_i : in std_logic_vector(7 downto 0);
                block1_i : in std_logic_vector(7 downto 0);
                block2_i : in std_logic_vector(7 downto 0);
                block3_i : in std_logic_vector(7 downto 0);

                --for debugging purposes
                a_o : out std_logic_vector(31 downto 0);
                b_o : out std_logic_vector(31 downto 0);
                c_o : out std_logic_vector(31 downto 0);
                d_o : out std_logic_vector(31 downto 0);
                e_o : out std_logic_vector(31 downto 0);
                f_o : out std_logic_vector(31 downto 0);
                g_o : out std_logic_vector(31 downto 0);
                h_o : out std_logic_vector(31 downto 0);
                state_o : out std_logic_vector(4 downto 0);
                --end debug

                read_txt_o      : out std_logic;
                read_addrs_o : out std_logic_vector(3 downto 0);
                valid_hash_o : out std_logic;
                hash_o : out std_logic_vector(31 downto 0)
```

```
                --For debugging purposes
                -- m_tmp_o : out std_logic_vector(31 downto 0);
                -- w_j_tmp_o : out std_logic_vector(31 downto 0);
                -- msg_0_o         : out std_logic_vector(31 downto 0);
                -- rst_sch_tmp_o : out std_logic;
                -- k_i_tmp_o : out std_logic_vector(31 downto 0);
                -- h_0_tmp_o : out std_logic_vector(31 downto 0);
                -- cnt_s_o : out std_logic_vector(5 downto 0);

                -- d_0_tmp_o  : out std_logic_vector(31 downto 0);
                -- q_0_tmp_o  : out std_logic_vector(31 downto 0);
                -- sigma0_o   : out std_logic_vector(31 downto 0);
                -- sigma1_o   : out std_logic_vector(31 downto 0);
                -- tmp_3_o    : out std_logic_vector(31 downto 0);
                -- tmp_4_o    : out std_logic_vector(31 downto 0);
                -- tmp_5_o    : out std_logic_vector(31 downto 0)

                );
end sha256_core;

architecture Behavioral of sha256_core is

        component msg_comp is
         port(clk  : in std_logic;
                        rst : in std_logic;

                        h_0 : in std_logic_vector(31 downto 0);
                        h_1 : in std_logic_vector(31 downto 0);
                        h_2 : in std_logic_vector(31 downto 0);
                        h_3 : in std_logic_vector(31 downto 0);
                        h_4 : in std_logic_vector(31 downto 0);
                        h_5 : in std_logic_vector(31 downto 0);
                        h_6 : in std_logic_vector(31 downto 0);
                        h_7 : in std_logic_vector(31 downto 0);

                        w_i : in std_logic_vector(31 downto 0);
                        k_i : in std_logic_vector(31 downto 0);

                        a : out std_logic_vector(31 downto 0);
                        b : out std_logic_vector(31 downto 0);
                        c : out std_logic_vector(31 downto 0);
                        d : out std_logic_vector(31 downto 0);
                        e : out std_logic_vector(31 downto 0);
                        f : out std_logic_vector(31 downto 0);
                        g : out std_logic_vector(31 downto 0);
                        h : out std_logic_vector(31 downto 0));
        end component;

        component sh_reg is
         port(
                        clk  : in std_logic;
                        rst : in std_logic;
                        msg_0 : in std_logic_vector(31 downto 0);
                        msg_1 : in std_logic_vector(31 downto 0);
                        msg_2 : in std_logic_vector(31 downto 0);
                        msg_3 : in std_logic_vector(31 downto 0);
                        msg_4 : in std_logic_vector(31 downto 0);
                        msg_5 : in std_logic_vector(31 downto 0);
                        msg_6 : in std_logic_vector(31 downto 0);
                        msg_7 : in std_logic_vector(31 downto 0);
                        msg_8 : in std_logic_vector(31 downto 0);
                        msg_9 : in std_logic_vector(31 downto 0);
                        msg_10 : in std_logic_vector(31 downto 0);
                        msg_11 : in std_logic_vector(31 downto 0);
                        msg_12 : in std_logic_vector(31 downto 0);
                        msg_13 : in std_logic_vector(31 downto 0);
                        msg_14 : in std_logic_vector(31 downto 0);
                        msg_15 : in std_logic_vector(31 downto 0);
```

```
                w_j : out std_logic_vector(31 downto 0);
                d_0_tmp_o  : out std_logic_vector(31 downto 0);
                q_0_tmp_o  : out std_logic_vector(31 downto 0);
                sigma0_o   : out std_logic_vector(31 downto 0);
                sigma1_o   : out std_logic_vector(31 downto 0);
                tmp_3_o    : out std_logic_vector(31 downto 0);
                tmp_4_o    : out std_logic_vector(31 downto 0);
                tmp_5_o    : out std_logic_vector(31 downto 0)

                   );
end component;

component dual_mem is
generic (ADDR_LENGTH : integer := 6;
        DATA_LENGTH : integer := 32;
        N_ADDR      : integer := 64);
port (clk  : in std_logic;
        a    : in std_logic_vector(ADDR_LENGTH - 1 downto 0);
        spo  : out std_logic_vector(DATA_LENGTH - 1 downto 0)
);
end component;

signal w_j_tmp : std_logic_vector(31 downto 0) := (others=>'0');

signal h_0_tmp,hash_tmp0 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_1_tmp,hash_tmp1 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_2_tmp,hash_tmp2 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_3_tmp,hash_tmp3 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_4_tmp,hash_tmp4 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_5_tmp,hash_tmp5 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_6_tmp,hash_tmp6 : std_logic_vector(31 downto 0) := (others=>'0');
signal h_7_tmp,hash_tmp7 : std_logic_vector(31 downto 0) := (others=>'0');

signal k_i_tmp : std_logic_vector(31 downto 0);

signal start_cnt_tmp, rst_sch_tmp, rst_comp_tmp : std_logic :='0';
signal cnt_s : std_logic_vector(5 downto 0):= (others=>'0');

signal m_tmp : std_logic_vector(31 downto 0):= (others=>'0');

constant idle : integer := 0;
constant init : integer := 1;
constant run : integer := 2;
constant m_1 : integer := 3;
constant m_2 : integer := 4;
constant m_3 : integer := 5;
constant m_4 : integer := 6;
constant m_5 : integer := 7;
constant m_6 : integer := 8;
constant m_7 : integer := 9;
constant m_8 : integer := 10;
constant m_9 : integer := 11;
constant m_10 : integer := 12;
constant m_11 : integer := 13;
constant m_12 : integer := 14;
constant m_13 : integer := 15;
constant m_14 : integer := 16;
constant m_15 : integer := 17;
constant w_s : integer := 18;

signal state,next_state: integer range 0 to 32 := idle;

type delay_buffer_t is array(73 downto 0) of std_logic;

signal hash_delay : delay_buffer_t := (others=>'0');
signal a_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal b_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal c_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal d_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal e_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal f_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
signal g_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');
```

```
	signal h_out_tmp : std_logic_vector(31 downto 0):= (others=>'0');

	signal gen_hash_tmp : std_logic := '0';
	signal rst_cnt_s : std_logic := '0';
	signal rst : std_logic := '0';
	signal gen_hash : std_logic := '0';
	signal msg_0 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_1 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_2 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_3 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_4 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_5 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_6 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_7 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_8 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_9 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_10 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_11 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_12 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_13 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_14 : std_logic_vector(31 downto 0):= (others=>'0');
	signal msg_15 : std_logic_vector(31 downto 0):= (others=>'0');

	signal read_txt,read_txt_d,read_txt_d2 : std_logic := '0';

	signal hash_tmp : std_logic_vector(255 downto 0):= (others=>'0');
	signal cnt_rd,cnt_rd_d : std_logic_vector(3 downto 0) := (others=>'0');
	signal rst_enable,falling_edge_read_txt : std_logic := '0';
	signal block_i : std_logic_vector(31 downto 0):= (others=>'0');
	signal hash : std_logic_vector(31 downto 0):= (others=>'0');

	signal a_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal b_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal c_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal d_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal e_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal f_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal g_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal h_o0 : std_logic_vector(31 downto 0) := (others=>'0');
	signal valid_hash : std_logic := '0';

begin

	process1: process (clk_i, rst)
	begin
		if rising_edge(clk_i) then
			state <= next_state;
		end if;
	end process process1;

	--Debugging
	state_o <= conv_std_logic_vector(state,state_o'length);
	-- m_tmp_o <= m_tmp;
	-- w_j_tmp_o <= w_j_tmp;
	-- msg_0_o <= msg_0;
	-- rst_sch_tmp_o <= rst_sch_tmp;
	-- k_i_tmp_o <= k_i_tmp;
	-- h_0_tmp_o <= h_0_tmp;
	-- cnt_s_o <= cnt_s;

	block_i <= block0_i & block1_i & block2_i & block3_i;
	falling_edge_read_txt <= '1' when (read_txt_d = '0' and read_txt_d2 = '1') else
'0';

	--Generate "rst" only for the first message after a "new_msg_i" pulse
	rst  <= '1' when  falling_edge_read_txt = '1' and rst_enable = '1' else '0';
	read_addrs_o <= cnt_rd;

	process_genhash : process (clk_i)
	begin
	if rising_edge(clk_i) then
```

```
                read_txt_d <= read_txt;
                read_txt_d2 <= read_txt_d;
                cnt_rd_d <= cnt_rd;
                if hash_delay(66) = '1' then
                        gen_hash <= '0';  --Set to one when hash processing has ended
                elsif falling_edge_read_txt = '1' then
                        gen_hash <= '1';  --Set to one when reading process has ended
                end if;
                --Control read process. 16cycles.
                if new_msg_i = '1' or cnt_rd = "1111" or new_blk_i = '1' then
                        cnt_rd <= (others=>'0');
                elsif read_txt = '1' then
                        cnt_rd <= cnt_rd + 1;
                end if;

                if new_msg_i = '1' then
                        rst_enable <= '1';
                elsif rst = '1' then
                        rst_enable <= '0';
                end if;

                if new_msg_i = '1' or new_blk_i = '1' then
                        read_txt <= '1';
                elsif cnt_rd = "1111" then
                        read_txt <= '0';
                end if;

                --Divide input block in several sub-blocks of 32bits
                if read_txt_d = '1' then
                        case cnt_rd_d is
                                when "0000" =>
                                        msg_0 <= block_i;
                                when "0001" =>
                                        msg_1 <= block_i;
                                when "0010" =>
                                        msg_2 <= block_i;
                                when "0011" =>
                                        msg_3 <= block_i;
                                when "0100" =>
                                        msg_4 <= block_i;
                                when "0101" =>
                                        msg_5 <= block_i;
                                when "0110" =>
                                        msg_6 <= block_i;
                                when "0111" =>
                                        msg_7 <= block_i;
                                when "1000" =>
                                        msg_8 <= block_i;
                                when "1001" =>
                                        msg_9 <= block_i;
                                when "1010" =>
                                        msg_10 <= block_i;
                                when "1011" =>
                                        msg_11 <= block_i;
                                when "1100" =>
                                        msg_12 <= block_i;
                                when "1101" =>
                                        msg_13 <= block_i;
                                when "1110" =>
                                        msg_14 <= block_i;
                                when "1111" =>
                                        msg_15 <= block_i;
                                when others =>
                        end case;
                end if;
                end if;
        end process;
        read_txt_o <= read_txt;

process2 : process (state, gen_hash, m_tmp, msg_0, msg_1, msg_2, msg_3, msg_4, msg_5, msg_6,
msg_7, msg_8,
        msg_9,msg_10,msg_11,msg_12,msg_13,msg_14,msg_15,w_j_tmp, hash_delay(66))
```

```
begin
        next_state <= state;

        rst_sch_tmp <= '0';
        rst_comp_tmp <= '0';
        rst_cnt_s <= '0';

        start_cnt_tmp <= '0';
        m_tmp <= (others => '0');

        gen_hash_tmp <= '0';

        case state is
        when idle =>
                if gen_hash = '1' then
                        gen_hash_tmp <= '1';
                        rst_cnt_s <= '1';
                        next_state <= init;
                else
                        next_state <= idle;
                end if;
        when init =>
                rst_comp_tmp <= '1';
                start_cnt_tmp <= '1';

                next_state <= run;
        when run =>
                rst_comp_tmp <= '0';
                start_cnt_tmp <= '1';
                m_tmp <= msg_15;

                next_state <= m_1;
        when m_1 =>
                m_tmp <= msg_14;
                start_cnt_tmp <= '1';
                next_state <= m_2;
        when m_2 =>
                m_tmp <= msg_13;
                start_cnt_tmp <= '1';
                next_state <= m_3;
        when m_3 =>
                m_tmp <= msg_12;
                start_cnt_tmp <= '1';
                next_state <= m_4;
        when m_4 =>
                m_tmp <= msg_11;
                start_cnt_tmp <= '1';
                next_state <= m_5;
        when m_5 =>
                m_tmp <= msg_10;
                start_cnt_tmp <= '1';
                next_state <= m_6;
        when m_6 =>
                m_tmp <= msg_9;
                start_cnt_tmp <= '1';
                next_state <= m_7;
        when m_7 =>
                m_tmp <= msg_8;
                start_cnt_tmp <= '1';
                next_state <= m_8;
        when m_8 =>
                m_tmp <= msg_7;
                start_cnt_tmp <= '1';
                next_state <= m_9;
        when m_9 =>
                m_tmp <= msg_6;
                start_cnt_tmp <= '1';
                next_state <= m_10;
        when m_10 =>
                m_tmp <= msg_5;
                start_cnt_tmp <= '1';
                next_state <= m_11;
        when m_11 =>
```

```
                m_tmp <= msg_4;
                start_cnt_tmp <= '1';
                next_state <= m_12;
        when m_12 =>
                m_tmp <= msg_3;
                start_cnt_tmp <= '1';
                next_state <= m_13;
        when m_13 =>
                m_tmp <= msg_2;
                start_cnt_tmp <= '1';
                next_state <= m_14;
        when m_14 =>
                m_tmp <= msg_1;
                start_cnt_tmp <= '1';
                next_state <= m_15;
        when m_15 =>
                m_tmp <= msg_0;
                start_cnt_tmp <= '1';
                rst_sch_tmp <= '1';
                next_state <= w_s;
        when w_s =>
                m_tmp <= w_j_tmp;
                start_cnt_tmp <= '1';
                if hash_delay(66) = '1' then
                        next_state <= idle;
                else
                        next_state <= w_s;
                end if;
        when others=>
                next_state <= idle;
        end case;

end process;

message_schedule: sh_reg port map (clk_i,
        rst_sch_tmp,
        msg_0,
        msg_1,
        msg_2,
        msg_3,
        msg_4,
        msg_5,
        msg_6,
        msg_7,
        msg_8,
        msg_9,
        msg_10,
        msg_11,
        msg_12,
        msg_13,
        msg_14,
        msg_15,
        w_j_tmp,
        open,
        open,
        open,
        open,
        open,
        open,
        open

        -- d_0_tmp_o,
        -- q_0_tmp_o,
        -- sigma0_o ,
        -- sigma1_o ,
        -- tmp_3_o ,
        -- tmp_4_o ,
        -- tmp_5_o
        );
```

```
message_compression: msg_comp port map (clk_i,
        rst_comp_tmp,

        h_0_tmp,
        h_1_tmp,
        h_2_tmp,
        h_3_tmp,
        h_4_tmp,
        h_5_tmp,
        h_6_tmp,
        h_7_tmp,
        m_tmp,
        k_i_tmp,
        a_out_tmp,
        b_out_tmp,
        c_out_tmp,
        d_out_tmp,
        e_out_tmp,
        f_out_tmp,
        g_out_tmp,
        h_out_tmp);

a_o0 <= a_out_tmp;
b_o0 <= b_out_tmp;
c_o0 <= c_out_tmp;
d_o0 <= d_out_tmp;
e_o0 <= e_out_tmp;
f_o0 <= f_out_tmp;
g_o0 <= g_out_tmp;
h_o0 <= h_out_tmp;

a_o <= a_o0;
b_o <= b_o0;
c_o <= c_o0;
d_o <= d_o0;
e_o <= e_o0;
f_o <= f_o0;
g_o <= g_o0;
h_o <= h_o0;


k_mem: dual_mem port map(clk_i,
cnt_s, --cnt_s,
k_i_tmp);

cnt_k_pr: process(clk_i)
        variable cnt_v : unsigned(5 downto 0) := (others => '0');
        begin
        if rising_edge(clk_i) then
                if rst_cnt_s = '1' then
                        cnt_v := (others => '0');
                elsif start_cnt_tmp = '1' then
                        cnt_v := cnt_v + 1;
                end if;
        end if;
        cnt_s <= std_logic_vector(cnt_v);
end process;

hash_delay(0) <= gen_hash_tmp;

delay_chain: for i in 1 to hash_delay'high generate
        delay_ff_proc: process(clk_i)
                begin
                if rising_edge(clk_i) then
                        hash_delay(i) <= hash_delay(i-1);
                end if;
        end process delay_ff_proc;
end generate delay_chain;

final_block: process(clk_i, rst, gen_hash, hash_delay(65),
        a_out_tmp,
        b_out_tmp,
```

```
                c_out_tmp,
                d_out_tmp,
                e_out_tmp,
                f_out_tmp,
                g_out_tmp,
                h_out_tmp)
                variable h_0_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_1_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_2_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_3_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_4_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_5_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_6_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
                variable h_7_tmp_v : std_logic_vector(31 downto 0) := (others => '0');
        begin
                if rising_edge(clk_i) then
                        if rst = '1' then
                                h_0_tmp_v := X"6a09e667";
                                h_1_tmp_v := X"bb67ae85";
                                h_2_tmp_v := X"3c6ef372";
                                h_3_tmp_v := X"a54ff53a";
                                h_4_tmp_v := X"510e527f";
                                h_5_tmp_v := X"9b05688c";
                                h_6_tmp_v := X"1f83d9ab";
                                h_7_tmp_v := X"5be0cd19";
                        elsif hash_delay(65) = '1' then
                                h_0_tmp_v := (h_0_tmp_v) + (a_out_tmp);
                                h_1_tmp_v := (h_1_tmp_v) + (b_out_tmp);
                                h_2_tmp_v := (h_2_tmp_v) + (c_out_tmp);
                                h_3_tmp_v := (h_3_tmp_v) + (d_out_tmp);
                                h_4_tmp_v := (h_4_tmp_v) + (e_out_tmp);
                                h_5_tmp_v := (h_5_tmp_v) + (f_out_tmp);
                                h_6_tmp_v := (h_6_tmp_v) + (g_out_tmp);
                                h_7_tmp_v := (h_7_tmp_v) + (h_out_tmp);

                        end if;
                end if;

                h_0_tmp <= h_0_tmp_v;
                h_1_tmp <= h_1_tmp_v;
                h_2_tmp <= h_2_tmp_v;
                h_3_tmp <= h_3_tmp_v;
                h_4_tmp <= h_4_tmp_v;
                h_5_tmp <= h_5_tmp_v;
                h_6_tmp <= h_6_tmp_v;
                h_7_tmp <= h_7_tmp_v;

        end process;

        valid_hash_o <= valid_hash;
        hash_o <= hash;

        output_process : process(clk_i)
        begin
                if rising_edge(clk_i) then
                        if rst = '1' then
                                valid_hash <= '0';
                        else
valid_hash <= hash_delay(66) or hash_delay(67) or
 hash_delay(68) or hash_delay(69)
or hash_delay(70) or hash_delay(71) or hash_delay(72) or
hash_delay(73);
                        end if;
                        --Divide result in sub-blocks of 32bits
                        if hash_delay(66) = '1' then
                                hash_tmp0 <= h_0_tmp;
                                hash_tmp1 <= h_1_tmp;
                                hash_tmp2 <= h_2_tmp;
                                hash_tmp3 <= h_3_tmp;
                                hash_tmp4 <= h_4_tmp;
                                hash_tmp5 <= h_5_tmp;
                                hash_tmp6 <= h_6_tmp;
```

```
                        hash_tmp7 <= h_7_tmp;
                    end if;
                    if  hash_delay(66) = '1' then
                        hash <= h_0_tmp;
                    elsif hash_delay(67) = '1' then
                        hash <= hash_tmp1;
                    elsif hash_delay(68) = '1' then
                        hash <= hash_tmp2;
                    elsif hash_delay(69) = '1' then
                        hash <= hash_tmp3;
                    elsif hash_delay(70) = '1' then
                        hash <= hash_tmp4;
                    elsif hash_delay(71) = '1' then
                        hash <= hash_tmp5;
                    elsif hash_delay(72) = '1' then
                        hash <= hash_tmp6;
                    elsif hash_delay(73) = '1' then
                        hash <= hash_tmp7;
                    end if;
                end if;
        end process;

end Behavioral;
```

## 10.2.1.2. sha160_wrapper.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity sha160_core_wrapper is
        port(
                clk_i  : in std_logic;
                ce_i : in std_logic;
                new_msg_i : in std_logic;
                new_blk_i : in std_logic;

                block0_i : in std_logic_vector(7 downto 0);
                block1_i : in std_logic_vector(7 downto 0);
                block2_i : in std_logic_vector(7 downto 0);
                block3_i : in std_logic_vector(7 downto 0);

                read_txt_o      : out std_logic;
                read_addrs_o : out std_logic_vector(3 downto 0);
                valid_hash_o : out std_logic;
                hash_o : out std_logic_vector(31 downto 0)

                );
end sha160_core_wrapper;

architecture Behavioral of sha160_core_wrapper is

component sha1 is
  port(
  m                : in  bit_vector ( 31 downto 0); -- 32 bit data path require 16 clock to load all
                                                    -- 512 bits of each block
  init             : in  bit;                       --  initial message
  ld               : in  bit;                       --  load signal
  h                : out bit_vector ( 31 downto 0); --  5 clock after active valid signal is the
--message hash result
  v                : out bit;                       --  hash output valid signal one clock advance
  clk              : in  bit;                       --  master clock signal
```

```
  rst             : in  bit                       --  master reset signal
  );
end component;

signal read_txt        : std_logic:= '0';
signal end_read : std_logic := '0';
signal cnt_read : std_logic_vector(3 downto 0);
signal input_block : std_logic_vector(31 downto 0);
signal valid_hash : std_logic := '0';
signal read_txt_d : std_logic := '0';

signal m       : bit_vector ( 31 downto 0); -- 32 bit data path require 16 clock to load all 512 bits of
      --each block
signal init : bit;                          --  initial message
signal ld   : bit;                         --  load signal
signal h    : bit_vector ( 31 downto 0); --  5 clock after active valid signal is the message hash result
signal v    : bit;                         --  hash output valid signal one clock advance
signal clk : bit;                          --  master clock signal
signal rst : bit;                           --  master reset signal
signal end_valid_hash,valid_hash_out : std_logic:= '0';
signal cnt_hash : std_logic_vector(2 downto 0) := (others=>'0');

begin

--Read Signal Generation. 16 cycles.
end_read <= '1' when cnt_read = 15 else '0';
process (clk_i)
  begin
       if rising_edge(clk_i) then
              read_txt_d <= read_txt;
              if end_read = '1' then
                     read_txt <= '0';
              elsif new_msg_i = '1' or new_blk_i = '1' then
                     read_txt <= '1';
              end if;
              if end_read = '1' or new_msg_i = '1' or new_blk_i = '1' then
                     cnt_read <= (others=>'0');
              elsif read_txt = '1' then
                     cnt_read <= cnt_read + 1;
              end if;
              if end_valid_hash  = '1' then
                     valid_hash_out <= '0';
              elsif valid_hash = '1' then
                     valid_hash_out <= '1';
              end if;
              if end_valid_hash = '1' then
                     cnt_hash <= (others=>'0');
              else
                     if valid_hash_out = '1' then
                            cnt_hash <= cnt_hash + 1;
                     end if;
              end if;
       end if;
  end process;
--Hash Validation. 5cycles
end_valid_hash <= '1' when cnt_hash = 4 else '0';
valid_hash_o <= valid_hash_out;

read_txt_o <= read_txt;
input_block <= block0_i & block1_i & block2_i & block3_i;
read_addrs_o <= cnt_read;
init <= rst;
m <= to_bitvector(input_block); -- 32 bit data path require 16 clock to load all 512 bits of each block
ld   <= to_bit(read_txt_d);
clk <= to_bit(clk_i);
rst <= to_bit(new_msg_i);
hash_o <=  to_stdlogicvector(h);
valid_hash <=  To_StdULogic(v);

--Original algorithm
```

```
sha1_core: sha1
  port map (
  m    => m,     -- 32 bit data path require 16 clock to load all 512 bits of each block
  init => rst,     --  initial message
  ld   => ld,     --  load signal
  h    => h,    --  5 clock after active valid signal is the message hash result
  v    => v,   --  hash output valid signal one clock advance
  clk  => clk,     --  master clock signal
  rst  => rst     --  master reset signal
  );

end Behavioral;
```

## 10.2.1.3. sha512_wrapper.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sha512_core_wrapper is
        port(
               clk_i  : in std_logic;
               ce_i : in std_logic;
               new_msg_i : in std_logic;
               new_blk_i : in std_logic;

               block0_i : in std_logic_vector(7 downto 0);
               block1_i : in std_logic_vector(7 downto 0);
               block2_i : in std_logic_vector(7 downto 0);
               block3_i : in std_logic_vector(7 downto 0);

               read_txt_o      : out std_logic;
               read_addrs_o : out std_logic_vector(4 downto 0);
               valid_hash_o : out std_logic;
               hash_o : out std_logic_vector(31 downto 0)

               );
end sha512_core_wrapper;

architecture Behavioral of sha512_core_wrapper is

component sha512 is
  port(
  m               : in  bit_vector (63 downto 0);  -- 64 bit data path require 16 clock to load all
-- 1024 bits of each block
  init            : in  bit;                       --  initial message
  ld              : in  bit;                       --  load signal
  md              : out bit_vector (63 downto 0);  --  message hash result
  v               : out bit;                       --  hash output valid signal one clock advance
  clk             : in  bit;                       --  master clock signal
  rst             : in  bit                        --  master reset signal
  );
end component;
signal read_txt        : std_logic:= '0';

type MEM_TYPE is array (31 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
signal MEM :MEM_TYPE;

type MEM_TYPE_2 is array (15 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
signal MEM_O :MEM_TYPE_2;
```

```
signal end_read : std_logic := '0';
signal cnt_read : std_logic_vector(4 downto 0);
signal input_block : std_logic_vector(31 downto 0);
signal valid_hash : std_logic := '0';
signal read_txt_d : std_logic := '0';
signal hash : std_logic_vector(63 downto 0);
signal m       : bit_vector (63 downto 0); -- 64 bit data path require 16 clock to load all 1024 bits of -
     --each block
signal init : bit;                      --  initial message
signal ld   : bit;                      --  load signal
signal h    : bit_vector (63 downto 0); --  The message hash result
signal v    : bit;                      --  hash output valid signal one clock advance
signal clk : bit;                       --  master clock signal
signal rst : bit;                       --  master reset signal
signal end_valid_hash,valid_hash_out : std_logic:= '0';
signal cnt_hash : std_logic_vector(2 downto 0) := (others=>'0');
signal write_m : std_logic := '0';
signal read_m,read_m_d : std_logic := '0';
signal wr_addrs : std_logic_vector(4 downto 0) := (others=>'0');
signal rd_addrs,rd_addrs_h : std_logic_vector(4 downto 0) := (others=>'0');
signal block_h,block_l : std_logic_vector(31 downto 0);
signal input_block64 : std_logic_vector(63 downto 0);
signal wr_addrs_hash_l : std_logic_vector(cnt_hash'length downto 0) := (others=>'0');
signal wr_addrs_hash_h : std_logic_vector(wr_addrs_hash_l' range) := (others=>'0');
signal rd_addrs_hash : std_logic_vector(wr_addrs_hash_h' range) := (others=>'0');
signal read_hash : std_logic := '0';

begin

--Read Signal generation. 32cycles
end_read <= '1' when cnt_read = 31 else '0';
process (clk_i)
begin
if rising_edge(clk_i) then
        wr_addrs <= cnt_read;
        write_m <= read_txt;
        read_m_d <= read_m;

        --Reading process
        read_txt_d <= read_m;
        if end_read = '1' then
                read_txt <= '0';
        elsif new_msg_i = '1' or new_blk_i = '1' then
                read_txt <= '1';
        end if;
        if end_read = '1' or new_msg_i = '1' or new_blk_i = '1' then
                cnt_read <= (others=>'0');
        elsif read_txt = '1' then
                cnt_read <= cnt_read + 1;
        end if;

        --Output Generation
        if end_valid_hash  = '1' then
                valid_hash_out <= '0';
        elsif valid_hash = '1' then
                valid_hash_out <= '1';
        end if;
        if end_valid_hash = '1' then
                cnt_hash <= (others=>'0');
        else
                if valid_hash_out = '1' then
                        cnt_hash <= cnt_hash + 1;
                end if;
        end if;
end if;
end process;
--8cycles *64bit = 512bits
end_valid_hash <= '1' when cnt_hash = 7 else '0';

read_txt_o <= read_txt;
```

```
input_block <= block0_i & block1_i & block2_i & block3_i;

read_addrs_o <= cnt_read;

init <= rst;
m <= to_bitvector(input_block64);
ld   <= to_bit(read_m_d);
clk <= to_bit(clk_i);
rst <= to_bit(new_msg_i);

hash <=  to_stdlogicvector(h);
valid_hash <=  To_StdULogic(v);

sha512_core: sha512
  port map (
  m    => m,
  init => rst,   -- initial message
  ld   => ld,    -- load signal
  md    => h,    -- hash result
  v    => v,     -- hash output valid signal one clock advance
  clk  => clk,   -- master clock signal
  rst  => rst    -- master reset signal
  );

-------------------------------- Process to adapt 32to64bits (input) ------------
input_block64 <= block_h & block_l;
rd_addrs_h <= rd_addrs + 1;
process (clk_i)
begin
if rising_edge(clk_i) then
        if write_m = '1' then
                MEM(conv_integer(wr_addrs)) <= input_block;
        end if;
        if read_m = '1' then
                block_h <= MEM(conv_integer(rd_addrs_h));
                block_l <= MEM(conv_integer(rd_addrs));
        end if;
end if;
end process;

process (clk_i)
begin
if rising_edge(clk_i) then
        if rd_addrs_h = 31 and read_m = '1' then
                read_m <= '0';
        elsif write_m = '1' and wr_addrs = 16 then
                read_m <= '1';
        end if;
        if read_m = '1' then
                rd_addrs <= rd_addrs + 2;
        end if;
end if;
end process;

-------------------------------- Process to adapt 64to32bits (output) ------------
wr_addrs_hash_l <= cnt_hash & '0';
wr_addrs_hash_h <= cnt_hash & '1';

process (clk_i)
begin
if rising_edge(clk_i) then
        if valid_hash_out = '1' then
                MEM_O(conv_integer(wr_addrs_hash_h)) <= hash(63 downto 32);
                MEM_O(conv_integer(wr_addrs_hash_l)) <= hash(31 downto 0);
        end if;
        valid_hash_o <= read_hash;
        if read_hash = '1' then
                hash_o <= MEM_O(conv_integer(rd_addrs_hash));
        end if;
end if;
end process;
```

```
process (clk_i)
begin
if rising_edge(clk_i) then
        if rd_addrs_hash = 15 and read_hash = '1' then
                read_hash <= '0';
        elsif valid_hash_out = '1' then
                read_hash <= '1';
        end if;
        if read_hash = '1' then
                rd_addrs_hash <= rd_addrs_hash + 1;
        end if;
end if;
end process;

end Behavioral;
```

## 10.2.2. Ficheros Verificación

### 10.2.2.1. tb_sha_256.vhd

```
-- Copyright (c) 2013 Antonio de la Piedra

-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.

-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.

-- You should have received a copy of the GNU General Public License
-- along with this program.  If not, see <http://www.gnu.org/licenses/>.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY tb_sha_256 IS
END tb_sha_256;

ARCHITECTURE behavior OF tb_sha_256 IS

    -- Component Declaration for the Unit Under Test (UUT)
        component sha256_core is
                        port(
                                clk_i  : in std_logic;
                                ce_i : in std_logic;
                                new_msg_i : in std_logic;
                                new_blk_i : in std_logic;

                                block0_i : in std_logic_vector(7 downto 0);
                                block1_i : in std_logic_vector(7 downto 0);
                                block2_i : in std_logic_vector(7 downto 0);
                                block3_i : in std_logic_vector(7 downto 0);

                                a_o : out std_logic_vector(31 downto 0);
```

```
                        b_o : out std_logic_vector(31 downto 0);
                        c_o : out std_logic_vector(31 downto 0);
                        d_o : out std_logic_vector(31 downto 0);
                        e_o : out std_logic_vector(31 downto 0);
                        f_o : out std_logic_vector(31 downto 0);
                        g_o : out std_logic_vector(31 downto 0);
                        h_o : out std_logic_vector(31 downto 0);
                        state_o : out std_logic_vector(4 downto 0);
                        read_txt_o      : out std_logic;
                        valid_hash_o : out std_logic;
                        hash_o : out std_logic_vector(31 downto 0)

                );
end component;

type input_text_t is array(0 to 79) of std_logic_vector(31 downto 0);
signal cnt_text : std_logic_vector(5 downto 0):= (others=>'0');
signal input_text : input_text_t :=
(

        --Input text file
        X"4573746F",
        X"20657320",
        X"756E6120",
        X"70727565",
        X"62612064",
        X"656C2061",
        X"6C676F72",
        X"69746D6F",
        X"4573746F",
        X"20657320",
        X"756E6120",
        X"70727565",
        X"62612064",
        X"656C2061",
        X"6C676F72",
        X"69746D6F",--End file

        X"00000018",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"00000000",
        X"61626380",

        X"00000000",
        X"80000000",
        X"6E6F7071",
        X"6D6E6F70",
        X"6C6D6E6F",
        X"6B6C6D6E",
        X"6A6B6C6D",
        X"696A6B6C",
        X"68696A6B",
        X"6768696A",
        X"66676869",
        X"65666768",
        X"64656667",
        X"63646566",
        X"62636465",
        X"61626364",
```

```
                X"000001c0",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",

                X"000001c0",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000",
                X"00000000"
        );
    --Inputs
    signal clk_i : std_logic := '0';
    signal new_msg_i,new_blk_i : std_logic := '0';
    signal block_i : std_logic_vector(31 downto 0) := (others => '0');
    signal block_0,block_1,block_2,block_3 : std_logic_vector(31 downto 0) := (others => '0');
    signal block_4,block_5,block_6,block_7 : std_logic_vector(31 downto 0) := (others => '0');

        --Outputs
    signal a_o : std_logic_vector(31 downto 0):= (others => '0');
    signal b_o : std_logic_vector(31 downto 0):= (others => '0');
    signal c_o : std_logic_vector(31 downto 0):= (others => '0');
    signal d_o : std_logic_vector(31 downto 0):= (others => '0');
    signal e_o : std_logic_vector(31 downto 0):= (others => '0');
    signal f_o : std_logic_vector(31 downto 0):= (others => '0');
    signal g_o : std_logic_vector(31 downto 0):= (others => '0');
    signal h_o : std_logic_vector(31 downto 0):= (others => '0');
    signal valid_hash_o : std_logic := '0';
    signal hash_o : std_logic_vector(31 downto 0):= (others => '0');

    signal read_txt_o : std_logic := '0';
    signal cnt_hash : std_logic_vector(2 downto 0) := (others=>'0');
    signal hash_check : std_logic_vector(255 downto 0) := (others=>'0');
    signal new_msg_d,new_blk_d : std_logic := '0';
    signal valid_hash_d,valid_hash_d2 : std_logic := '0';

        -- Clock period definitions
    constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: sha256_core PORT MAP (
                clk_i => clk_i,
                ce_i => '1',
                new_msg_i => new_msg_d,
                new_blk_i => new_blk_d,

                block0_i => block_i(31 downto 24),
```

```
            block1_i => block_i(23 downto 16),
            block2_i => block_i(15 downto 8),
            block3_i => block_i(7 downto 0),
            a_o => a_o,
            b_o => b_o,
            c_o => c_o,
            d_o => d_o,
            e_o => e_o,
            f_o => f_o,
            g_o => g_o,
            h_o => h_o,
            state_o => open,
            read_txt_o => read_txt_o,
            valid_hash_o => valid_hash_o,
            hash_o => hash_o
      );

   -- Clock process definitions
   clk_process :process
   begin
            clk_i <= '0';
            wait for clk_period/2;
            clk_i <= '1';
            wait for clk_period/2;
   end process;

      process (clk_i)
        begin
            if rising_edge(clk_i) then
                  valid_hash_d <= valid_hash_o;
                  valid_hash_d2 <= valid_hash_d;
                  new_msg_d <= new_msg_i;
                  new_blk_d <= new_blk_i;

                  --read memory containing input data
                  if read_txt_o = '1' then
                        block_i <= input_text(conv_integer(cnt_text));
                        cnt_text <= cnt_text + 1;
                  end if;

                  if valid_hash_o = '1' then
                        cnt_hash <= cnt_hash + 1;
                        case cnt_hash is
                              --building a 512bits-output from 32bits blocks
                              when  "000" =>   block_0 <= hash_o;
                              when  "001"  =>  block_1 <= hash_o;
                              when  "010"  =>  block_2 <= hash_o;
                              when  "011"  =>  block_3 <= hash_o;
                              when  "100" =>   block_4 <= hash_o;
                              when  "101"  =>  block_5 <= hash_o;
                              when  "110"  =>  block_6 <= hash_o;
                              when  "111"  =>  block_7 <= hash_o;
                              when others =>  block_0 <= hash_o;
                        end case;
                  end if;
            end if;
        end process;
       hash_check <= block_0 & block_1 & block_2 & block_3 & block_4 & block_5 & block_6 &
block_7;

   -- Stimulus process
   stim_proc: process
   begin
            wait for clk_period/2 + clk_period + 1ns;

            wait for clk_period;

            new_msg_i <= '1';
            new_blk_i <= '1';
            wait for clk_period;
            new_msg_i <= '0';
            new_blk_i <= '0';
```

```
                wait until valid_hash_d2 = '1' and valid_hash_d = '0';--Wait until a new hash is
  --generated

                assert hash_check =
X"aa0b0d3338c142885487f5db611765be4fb50d7427d5532f90169ce87eb6bb3c"
                report "B.2 (Part 1) Hash output ERROR" severity FAILURE;

                wait for clk_period;

                -- Example from "APPENDIX B: SHA-256 EXAMPLES",
                -- B.1 SHA-256 Example (One-Block Message)
                -- FIPS 180-26
                new_msg_i <= '1';
                new_blk_i <= '1';
                wait for clk_period;
                new_msg_i <= '0';
                new_blk_i <= '0';

                wait until valid_hash_d2 = '1' and valid_hash_d = '0';

                assert hash_check =
X"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"
                report "B.1 Hash output ERROR" severity FAILURE;

                -- Example from "APPENDIX B: SHA-256 EXAMPLES",
                -- B.1 SHA-256 Example (One-Block Message)
                -- FIPS 180-26
                wait for clk_period;

                new_msg_i <= '1';
                new_blk_i <= '1';
                wait for clk_period;
                new_msg_i <= '0';
                new_blk_i <= '0';

                wait until valid_hash_d2 = '1' and valid_hash_d = '0';

                assert hash_check =
X"85e655d6417a17953363376a624cde5c76e09589cac5f811cc4b32c1f20e533a"
                report "B.2 (Part 1) Hash output ERROR" severity FAILURE;

                wait for clk_period;

                new_blk_i <= '1';
                wait for clk_period;
                new_blk_i <= '0';

                wait until valid_hash_d2 = '1' and valid_hash_d = '0';
                assert hash_check =
X"248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6ecedd419db06c1"
                report "B.2 (Part 2) Hash output ERROR" severity FAILURE;

                wait;
   end process;

END;
```

#### 10.2.2.2. Fichero para simulación en ModelSim: sha_256.do

```
vlib work

# libs

vcom -explicit    -93 "../rtl/dual_mem.vhd"
vcom -explicit  -93 "../rtl/ff_bank.vhd"
vcom -explicit  -93 "../rtl/sh_reg.vhd"
vcom -explicit  -93 "../rtl/msg_comp.vhd"
vcom -explicit  -93 "../rtl/sha_256.vhd"
vcom -explicit  -93 "tb_sha_256.vhd"

# Sim
vsim -lib work -t 1ps tb_sha_256

view wave
view source
view structure
view signals
add wave *

do wave_sha256.do

run 1ms
```