Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Final

SIMD sobre algoritmos de hash SHA-1 *y* SHA-256

- **Autor**: Jorge Daniel Pino
- **LU**: 556/07
- **Email**: JDanielPino@gmail.com
- **Fecha**: 2016.02.24

Resumen

Entre las aplicaciones típicas de la *Streaming SIMD Extensions* (*SSE*) de los procesadores *Intel* se destacan, entre otras, el *procesamiento de señales* (*DSP*), el *procesamiento de imágenes y gráficos*, y la *simulación de sistemas físicos*. Existe, sin embargo, otro área de aplicación probablemente menos popular: los *algoritmos de hash*. De esto nos ocupamos en este trabajo. Más precisamente, analizamos en detalle dos de los algoritmos de hash que continúan siendo los de mayor uso hoy día: *SHA-1* y un miembro de la familia de *SHA-2*: *SHA-256*. El objetivo consistió en un análisis pormenorizado de estos algoritmos con la idea de obtener una implementación *vectorizada* de los mismos, aprovechando las capacidades y características de la extensión *SSE*. Evaluamos luego la performance de cada una de estas implementaciones comparándolos con sus contraparte escalares. Si bien en ambos casos las versiones vectorizadas obtienen los mejores resultados, comprobaremos que el proceso de vectorización no es para nada trivial. Veremos también que estos dos algoritmos tienen distintas capacidades de adaptación al modelo *SIMD*.

Índice

Nomenclatura

$A \vee B$: operación *o* a nivel de bit entre los operandos A y B.

$A \wedge B$: operación *y* a nivel de bit entre los operandos A y B.

$A \oplus B$: operación *o-exclusiva* a nivel de bit entre los operandos A y B.

$\neg A$: operación *negación* a nivel de bit sobre el operando A.

$A \hookleftarrow B$: operación *rotación a izquierda* en B bits del operando A.

$A \hookrightarrow B$: operación *rotación a derecha* en B bits del operando A.

$A \ll B$: operación *desplazamiento a izquierda* en B bits del operando A.

$A \gg B$: operación *desplazamiento a derecha* en B bits del operando A.

1. Introducción

Entre los usos más frecuentes de la extensión *SSE* incluida en los procesadores *Intel* a partir del *Pentium III*, se encuentran, entre otros, el procesamiento de señales, el procesamiento de imágenes, y la simulación de sistemas físicos. Pero también existe otro área de aplicación menos conocido: los *algoritmos de hash*. De ello nos ocupamos en este trabajo.

Son dos los algoritmos de hashing que estudiaremos: el *Secure Hash Algorithm 1* (*SHA-1*), y un miembro de la denominada familia de algoritmos *SHA-2*: el *Secure Hash Algorithm 256* (*SHA-256*). La elección de estos dos algoritmos obedece al hecho de que ambos continúan siendo los algoritmos de hashing de mayor uso hoy día.

Ahora bien, ocurre que ni *SHA-1*, ni *SHA-256* son algoritmos precisamente *vectorizables*: como veremos, los datos computados en una ronda determinada dependen de los datos de la ronda anterior (propiedad implementada a partir de la *Construcción de Merkle-Damgård*[1]), de modo que no existe independencia entre ellos. Esto impide, como consecuencia, un procesamiento paralelo directo, como sí suele suceder en otros casos (por ejemplo, el procesamiento de imágenes).

Sin embargo, veremos que distintas propiedades matemáticas y algunos mecanismos muy rebuscados, combinados con una serie de optimizaciones, nos permitirán vectorizar gran parte del proceso estos dos algoritmos de hash.

El presente trabajo consiste, por lo tanto, en el análisis detallado de cada elemento (propiedades, optimizaciones, estrategias, etc.) que conducirán a la obtención de la versión cuasi-vectorizada de cada uno de los algoritmos de hashing estudiados, para luego evaluar el desempeño de estas implementaciones con sus contraparte escalares.

1.1. Software y hardware empleados en el proyecto

Este trabajo originalmente se inició en entorno *Debian Linux Squeeze* de 64 bits, corriendo en una máquina con procesador *Pentium D*, y memoria $2GB$ de RAM *DDR2*. Parte de la implementación de los algoritmos se limitó, por lo tanto, a la extensión *SSE3*.

Posteriormente, el desarrollo continuó en otra máquina con *Debian Linux Wheezy* de 64 bits, procesador *Intel Core I-5 3210M*, $2GB$ de RAM *DDR3*. Como consecuencia del traspaso, optimizamos un tanto más las implementaciones, incorporando nuevas instrucciones incluidas en la mejora *SSSE3* (véase la sección 3.5, *Instrucciones SSSE3 especiales*). De modo que todas las implementaciones de este proyecto se limitan a la extensión suplementaria *SSSE3*.

Todos los algoritmos fueron implementados en *ensamblador*, utilizando *NASM*[2], a excepción de aquellos casos donde se incluyen implementaciones adicionales en lenguaje *C*.

1.2. Recorrido del trabajo

Describimos a continuación las distintas partes que componen este trabajo:

- **Técnicas algorítmicas de optimización**:

  Como paso previo al análisis de los algoritmos de hashing, la sección 2 se ocupa de la descripción de dos de las primera técnicas algorítmicas que utilizaremos con frecuencia en las implementaciones de los algoritmos del proyecto: *loop unrolling* y *software pipelining*. La primera la aplicamos en todos los casos (esto es, algoritmos escalares y vectorizados); la segunda, solo en el caso de los algoritmos vectorizados. En el caso del *loop unrolling*, se incluye también una primera experiencia que permite evaluar qué tanto puede mejorar el desempeño de un algoritmo que implementa esta técnica, con respecto a otro que no

[1] La *Construcción de Merkle-Damgård*, también conocida como *Función de hash de Merkle-Damgård*, tiene como objetivo incrementar la resistencia a colisiones de los algoritmos de hash empleando funciones de *compresión* de los datos de *una vía* (que son, por definición, resistente a colisiones en sí mismas). Básicamente, estas funciones utilizan como entrada un nuevo bloque de datos y la salida de la aplicación de la misma función sobre el bloque de datos anterior.

[2] Página web oficial de *NASM*: `www.nasm.us`

la implementa. Si bien para este primer ensayo el algoritmo empleado es el *SHA-1*, no entramos en detalles con respecto al mismo, sino hasta la sección que sigue.

- **Algoritmo de hashing SHA-1**:

  La sección 3 analiza el primer algoritmo de hash que estudiaremos: el algoritmo SHA-1. Este análisis lo componen múltiples secciones, que van desde la definición formal del algoritmo de acuerdo al *fips-180*[3], pasando por el análisis detallado de las optimizaciones, propiedades matemáticas y estrategias que permiten su cuasi-vectorización, siguiendo por los detalles a nivel de código de la implementaciones vectorizada, y finalizando con los detalles de la experiencia y los resultados correspondientes.

- **Algoritmo de hashing SHA-256**:

  En la sección 4 repetimos el recorrido anterior sobre el segundo algoritmo de interés: el algoritmo *SHA-256*.

- **Detalles de las implementaciones y la generación de datos**:

  En esta se explican todos los detalles acerca de las aplicaciones desarrolladas y la generación de los datos: estructura de los programas implementados, detalles de los módulos que los componen, el modo de uso de las aplicaciones, los datos de entrada que reciben y la salida que generan, y el modo en que se procesa toda la información generada para obtener los datos de las distintas experiencias.

- **Apéndice - Extensiones especiales de los procesadores Intel**:

  Por último, la sección 6 es una apéndice con toda la información acerca de las nuevas extensiones que facilitan las operaciones de los algoritmos de hash que analizamos, cómo así también aquellas que expanden las capacidades de cómputo y que también podemos aprovechar sobre estos algoritmos. Más precisamente, describiremos:

  - La *Intel SHA Extensions*: se trata de un nuevo conjunto de instrucciones que permiten el cómputo de los hash SHA-1 y SHA-256 mediante la aceleración por hardware. Veremos en detalle y con ejemplos, cómo operan cada una de las 7 instrucciones de conforman esta extensión.
  - Las extensiones *AVXs*, en sus variantes *AVX1*, *AVX2* y la *AVX-512*[4]: entre sus características más sobresalientes se encuentran la duplicación y cuadruplicación del tamaño de los registros `XMM` y la incorporación de instrucciones vectoriales avanzadas. Repasaremos rápidamente cada uno de estos elementos, y veremos la manera en que podemos aprovechar estas nuevas características para mejorar los algoritmos de hash estudiados.
  - Extensión *BMI2*: incopora un número importante de instrucciones que operan a nivel de bit sobre registros de propósito general, especialmente útiles en las operaciones de los algoritmos de hash.

[3] *fips* son las siglas de *Federal Information Processing Standard*, que puede traducirse como *Estándares Federales para el Procesamiento de la Información*. Se trata de documentos emitidos por el *NIST* de los Estados Unidos (*National Institute of Standards and Technology* o, en español, *Instituto Nacional de Estándares y Tecnología*), donde básicamente se definen los estándares y regulaciones para el manejo de la información en computadoras.

[4] Se espera que la extensión *AVX-512* sea lanzada en algún momento del presente año.

2. Técnicas algorítmicas de uso frecuente en las implementaciones

En esta sección se describen dos técnicas algorítmicas aplicadas con mucha frecuencia en los distintos algoritmos implementados: *loop unrolling* y *software pipelining*.

El *loop unrolling* se aplica tanto en las versiones escalares de los algoritmos estudiados, como en las vectorizadas. El *software pipelining*, en cambio, solo en estas últimas.

2.1. *Loop Unrolling*

El *loop unrolling*, o también *loop unwinding*, que en español se suele traducir como *desenrosado de ciclos*, o *despliegue de ciclos*, es una técnica algorítmica que busca optimizar la velocidad de los ciclos basados en iteraciones (por ejemplo, el bucle `for`).

Consiste sencillamente en el despliegue secuencial del bloque de código correspondiente a cada iteración del ciclo. En términos más prácticos, si tenemos un ciclo como el siguiente:

Para $i = 0$ hasta 4, **hacer**
  $f(i)$
Fin Para

Donde f es cualquier bloque de código que puede o no requerir al contador i, lo reemplazamos por esto otro:

$f(0)$
$f(1)$
$f(2)$
$f(3)$
$f(4)$

Como resultado del despliegue, nos ahorramos todas las instrucciones relativas al control del ciclo, tales como el incremento de la variable que actúa como contador, como así también las instrucciones de comparación que evalúan la condición de finalización del ciclo.

Esta técnica, que a las claras es muy simple, suele mejorar considerablemente la performance de un ciclo, o incluso la aplicación entera. Esto es algo que comprobaremos en la primera experiencia del trabajo, que se describe en la sección 2.1.1 siguiente. Tiene la desventaja, sin embargo, de que puede incrementar significativamente el tamaño del ejecutable final. Pero en los casos donde el tamaño del programa no resulte un problema, o el incremento no sea significativo o, sobre todo, cuando la performance sea prioridad, como puede ser el caso del cómputo de un valor de hash, es altamente probable que el *loop unrolling* resulte conveniente.

En general, este mecanismo suele implementarse mediante el uso de *macros*; en particular, aquellas que admiten parámetros. Las macros simplifican y automatizan el proceso de despliegue al permitirnos definir el bloque de código a desplegar en un solo lugar, y en función de algún parámetro que represente el número de iteración. Con este valor, el ensamblador realiza el reemplazo correctamente en cada lugar. De otra forma, la tarea puede volverse demasiado tediosa y altamente susceptible a errores, dado que el proceso de despliegue para cada iteración habría que realizarlo de forma manual.

Veamos un ejemplo sencillo. Consideremos el caso de un ciclo de 20 iteraciones en el cual se leen secuencialmente en el registro `RAX` 20 valores `QWORD` desde un área de memoria apuntada por `BUFFER_PTR`. El código en ensamblador, sin *loop unrolling*, podría ser algo como lo siguiente:

```
      ...

   XOR RCX, RCX

loop:

   ;
   ; Actualización del puntero al buffer basado en el
   ; contador de la iteración.
   ;
   MOV RBX, RCX
   SHL RBX, 3   ; Multiplicamos por 8.

   ;
   ; Leemos el QWORD utilizando a RBX como offset.
   ;
   MOV RAX, [BUFFER_PTR + RBX]

   ;
   ; Instrucciones de control del ciclo.
   ;
   INC RCX
   CMP RCX, 20
   JLE loop

      ...
```

El mismo código, con *loop unrolling* implementado a través de macros, se convierte en esto otro:

```
   ;
   ; Definición de la macro para el despliegue.
   ;
   ; Recibe dos parámetros:
   ;
   ; %1: dirección del buffer.
   ; %2: número de ronda.
   ;
   %macro UNROLLED_LOOP 2

      MOV RAX, [%1 + %2*8]

   %endmacro

       ...

   ;
   ; El ciclo se transformó en esto:
   ;
   UNROLLED_LOOP BUFFER_PTR, 0
   UNROLLED_LOOP BUFFER_PTR, 1
   UNROLLED_LOOP BUFFER_PTR, 2
   UNROLLED_LOOP BUFFER_PTR, 3

       ...

   UNROLLED_LOOP BUFFER_PTR, 19
```

Como resultado, el ensamblador traduce el código anterior en lo siguiente antes del proceso de ensamblaje:

```
MOV RAX, [BUFFER_PTR]
MOV RAX, [BUFFER_PTR + 8]
MOV RAX, [BUFFER_PTR + 16]
MOV RAX, [BUFFER_PTR + 24]

   ...

MOV RAX, [BUFFER_PTR + 152]
```

Donde antes había 6 instrucciones, ahora hay una sola, y esto tendrá como resultado una mejora considerable en la velocidad de ejecución.

2.1.1. Experiencia

Con el objeto de verificar qué tanto puede mejorar el desempeño de un algoritmo al emplear la técnica del *loop unrolling*, realizamos una experiencia muy sencilla: escribimos un programa en *C* que implementa el primero de los algoritmos analizados en este trabajo: el algoritmo *SHA-1*.

El programa puede ejecutar el algoritmo en una de dos versiones: la primera, sin *loop unrolling*, y la segunda, con *loop unrolling*. Recibe, por lo tanto, dos parámetros: el primero, con el tipo de versión a ejecutar, y el segundo, con la ruta absoluta al archivo con los datos para los que se pretende calcular el valor de hash SHA-1. Al finalizar, la aplicación devuelve, entre otras cosas, el valor del hash SHA-1 asociado al archivo, y el *tiempo de CPU* que fue requerido por el cálculo. Estos datos también se guardan en un archivo de salida[5].

Dado que el algoritmo SHA-1 es tema de la sección 3, donde se lo analiza en detalle, omitimos ahora cualquier descripción, y nos enfocamos en el resultado de la comparación entre las dos implementaciones con y sin *loop unrolling*, respectivamente [6].

La experiencia consistió en la ejecución de ambas versiones del algoritmo SHA-1 sobre un grupo de archivos cuyos tamaños varían en el rango 0,5 *GB* y 3,0 *GB*, en saltos de 0,5 *GB* (es decir, 6 archivos de tamaños 0,5, 1, 1,5, 2, 2,5 y 3,0 *GB*, respectivamente). Cada ejecución se repitió media docena de veces, y de ahí se obtuvo el tiempo de CPU promedio para cada caso.

El esquema 1 ilustra el resultado de la prueba.

Del análisis de los datos, se desprende que la versión con *loop unrolling* resultó ser casi un 50 % más rápida que la implementación sin *loop unrolling*! (47, 1 %, para ser precisos) La diferencia es muy notable.

A la luz de estos resultados, todos los algoritmos implementados en este trabajo, tanto en las versiones escalares como en las vectorizadas, incluirán esta optimización.

[5] La sección 5 describe todos los detalles acerca del modo de uso, código fuente, generación y procesamiento de los datos, etc., de la aplicación implementada para esta experiencia.

[6] Sin embargo, la lectura del código fuente del programa para esta primera experiencia debería resultar lo suficientemente claro para su comprensión.

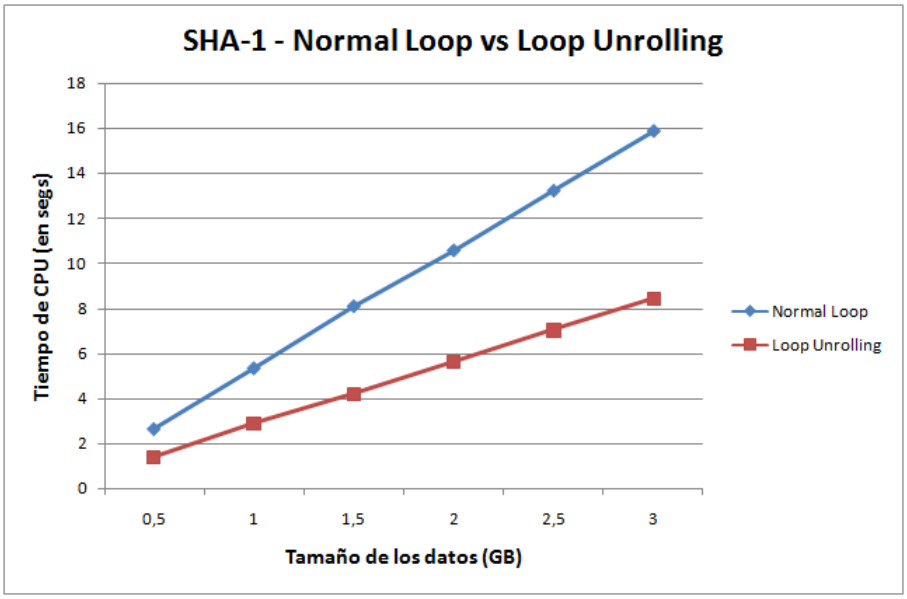

Figura 1: implementaciones de SHA-1 con y sin *loop unrolling*

2.2. *Software Pipelining*

Software pipelining es otra técnica algorítmica de optimización, que consiste básicamente en la ejecución de múltiples ductos o secuencias de instrucciones independientes entre sí. Tiene como resultado el aprovechamiento de la *ejecución fuera de orden*, propiedad de los procesadores capaz de optimizar el número de operaciones que éstos realizan en paralelo durante los ciclos de instrucción *fetch-decode-execute*. Veamos la idea con un ejemplo. Sea el siguiente ciclo:

Para $i = 0$ hasta N, **hacer**

$a = f(i)$
$b = g(a)$
$h(b)$

Fin Para

Es evidente la dependencia $h \to g \to f$, donde el nexo son, en este caso, las variables a y b. Este código, por lo tanto, debe ejecutarse de forma secuencial, y la ejecución fuera de orden esencialmente no es posible. Notemos, sin embargo, que la secuencia de instrucciones en un ciclo i es independiente a la del ciclo $i + 1$.

El *software pipelining* consiste en el reordenamiento de las sentencias de modo tal que las instrucciones de la secuencia resulten independientes entre sí. En nuestro caso, este reordenamiento podría ser algo como lo siguiente:

$a = f(0)$
$b = g(a)$
$a = f(1)$

Para $i = 2$ hasta N, **hacer**

$h(b)$
$b = g(a)$
$a = f(i)$

Fin Para

$b = g(a)$
$h(b)$

Si observamos el cuerpo del ciclo, las instrucciones que lo componen ahora son independientes, puesto que son las mismas que en la secuencia original, pero en el orden inverso. Notemos también que este reordenamiento implicó la ejecución de ciertas instrucciones *de arranque* antes del ciclo, y la ejecución de otras ciertas instrucciones que *cierran* el cómputo al finalizar dicho ciclo; lo primero se denomina *prólogo*, y lo segundo, *epílogo*.

Depleguemos el ciclo algunas iteraciones para hacer otra observación:

```
//
// Prólogo
//
a = f(0)    <- "Ducto" 0
b = g(a)    <- "Ducto" 0
a = f(1)    <- "Ducto" 1

//
// Ciclo desplegado.
//
h(b)        <- "Ducto" 0 - Completado
b = g(a)    <- "Ducto" 1
a = f(2)    <- "Ducto" 2

h(b)        <- "Ducto" 1 - Completado
b = g(a)    <- "Ducto" 2
a = f(3)    <- "Ducto" 3

h(b)        <- "Ducto" 2 - Completado
b = g(a)    <- "Ducto" 3
a = f(4)    <- "Ducto" 4

     ...

//
// Epílogo
//
b = g(a)    <- "Ducto" N
h(b)        <- "Ducto" N - Completado
```

Las etiquetas marcan cada ducto o secuencia independiente de instrucciones que se generan como resultado de la aplicación de esta técnica: es como que los datos en cada iteración se desplazan a la siguiente etapa de procesamiento en cada ducto paralelo.

La desventaja del software pipelining radica fundamentalmente en la manera que dificulta la comprensión del código: el prólogo, el cuerpo del ciclo, y el epílogo pueden fácilmente dar la impresión de no tener ninguna relación entre sí.

El *software pipelining* se suele combinar con el *loop unrolling* precisamente para favorecer el procesamiento de los múltiples *ductos*, a lo que se le suma las ventajas en el ahorro de las instrucciones de control del ciclo.

En el marco de este trabajo, el *software pipelinig* se aplica únicamente a las implementaciones vectorizadas de los algoritmos analizados.

3. El algoritmo de hashing *SHA-1*

3.1. Descripción

El algoritmo SHA-1 es una función de hash que toma un conjunto de datos de entrada, y devuelve un valor de hash de 160 bits de longitud.

Si bien el algoritmo está definido para datos a nivel de bit, en el contexto de este trabajo, en el que los datos de entrada en realidad son archivos, la menor unidad de almacenamiento pasa a ser el *Byte*. De manera que todos los algoritmos estudiados en este trabajo operan a nivel de *Byte*.

Con el objeto de describir el algoritmo SHA-1 de la manera más comprensible, a continuación haremos una descripción general del mismo, para luego especificar un pseudocódigo con todos los detalles.

En términos generales, podemos dividir el algoritmo SHA-1 en dos etapas: (1) el *padding* o agregado de datos, y (2) el procesamiento de los datos que generan el valor de hash:

1. Padding o relleno de los datos de entrada:
   - El algoritmo realiza un padding (agregado de datos) en los datos de entrada, con el objeto de que la longitud total de esos datos sea siempre múltiplo de 64 bytes.
   - Básicamente, el padding consiste en agregar algunos bytes al final de los datos de entrada.
   - Los detalles acerca de este proceso se describen en la sección 3.2.

2. Procesamiento de los datos:
   - Inmediatamente después del padding, los datos de entrada se parten en bloque de 64 bytes, que se procesan de forma secuencial.
   - El procesamiento de cada bloque tiene como resultado la actualización del valor de hash devuelto por el algoritmo.
   - Cada uno de los bloques de 64 bytes de los datos de entrada se considera un vector de 16 valores `DWORD`.
   - El cómputo realizado durante la actualización del valor de hash utiliza esos 16 `DWORDs` de cada bloque.

El siguiente pseudocódigo es la descripción formal del algoritmo SHA-1, de acuerdo al *fips-180*[7], para un conjunto de datos de entrada `m` de cualquier longitud:

[7] Véase nota 3.

Función `SHA-1`

- Entrada: conjunto de datos m de cualquier longitud en bytes
- Salida: vector $h[5]$ de `DWORD` con el valor del hash SHA-1

```
//
// Variables locales.
//
h[5]:   vector de DWORD       // Valor de hash.
w[80]: vector de DWORD        // Vector denominado Message Schedule.
a, b, c, d, e, f, k: DWORD    // Variables denominadas de estado.
aux: DWORD                    // Variable auxiliar.

//
// Inicialización del vector h.
//
h[0] = 0x67452301
h[1] = 0xEFCDAB89
h[2] = 0x98BADCFE
h[3] = 0x10325476
h[4] = 0xC3D2E1F0

//
// Padding o relleno. Después del padding, los datos de entrada m
// se extenderán en algunos bytes, y su tamaño total será múltiplo
// de 64 bits.
//
m = Padding(m)

//
// Procesamiento de cada bloque de 64 bytes de los datos
// de entrada m para generar el hash.
//
Para cada bloque b de 64 bytes en los datos m, hacer

    a = h[0]
    b = h[1]
    c = h[2]
    d = h[3]
    e = h[4]

    //
    // Copia directa de los 16 DWORDs correspondientes al bloque b
    // en los primeros 16 elementos del vector w.
    //
    Para i = 0 hasta 15 hacer
        w[i] = b[i]
    Fin Para

    //
    // Extensión del vector w: los elementos del vector w a partir
    // del 16 se obtienen mediante la siguiente función, expresada
    // en términos de elementos previos del vector w.
    //
    Para i = 16 hasta 79 hacer
        w[i] = (w[i − 3] ⊕ w[i − 8] ⊕ w[i − 14] ⊕ w[i − 16]) ↩ 1
    Fin Para
```

//
// Actualización de las variables de estado a, b, c, d y e.
//
Para $i = 0$ hasta 79, **hacer**

Si $(0 \leq i \leq 19)$ **entonces**
$f = (b \wedge c) \vee ((\neg\, b) \wedge d)$
$k =$ `0x5A827999`

Sino Si $(20 \leq i \leq 39)$ **entonces**
$f = b \oplus c \oplus d$
$k =$ `0x6ED9EBA1`

Sino Si $(40 \leq$ `i` $\leq 59)$ **entonces**
$f = (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$
$k =$ `0x8F1BBCDC`

Sino
$f = b \oplus c \oplus d$
$k =$ `0xCA62C1D6`

Fin Si

$aux = (a \hookleftarrow 5) + f + e + k + w[i]$
$e = d$
$d = c$
$c = b \hookleftarrow 30$
$b = a$
$a = aux$

Fin Para

//
// Actualización del valor de hash.
//
$h[0] = h[0] + a$
$h[1] = h[1] + b$
$h[2] = h[2] + c$
$h[3] = h[3] + d$
$h[4] = h[4] + e$

Fin Para

Devolver h

Fin Función

3.2. Padding en el algoritmo SHA-1

El proceso de padding se puede resumir de la siguiente forma:

1. Inmediatamente después del último byte de los datos de entrada m, se agrega el byte `0x80`.
2. A continuación, se agrega una cantidad n de bytes `0x0` (que se describe más adelante), que puede ser nula.
3. Se computa el tamaño t en bits de los datos de entrada m originales (esto es, sin el padding).
4. El valor obtenido t se expresa en un `QWORD`, en formato *Big-Endian* (esto es, el byte más significativo se ubica en la posición de memoria más significativa).
5. Asumiento que el tamaño en bytes de los datos de entrada m originales es s, el nuevo tamaño después del padding queda definido por:

$$s + 1 + n + 8$$

6. El valor n debe ser tal que la suma total produzca el menor múltiplo de 64.

Ejemplo:

- Sea `f` un archivo de tamaño 0 bytes.
- El tamaño de los datos después del padding queda definido como:

$$0 + 1 + n + 8 = 9 + n$$

- El menor múltiplo de 64 que puede obtenerse es precisamente 64, con lo cual:

$$9 + n = 64 \Rightarrow n = 55$$

- Luego, los datos de entrada adquieren la siguiente estructura tras el padding:

`0x80 | 0x0 ... 0x0 | 0x0000000000000000`

  donde:

  1. `0x80`: es el byte agregado al final de los datos. Dado que el archivo `f` del ejemplo no tiene datos, este byte ocupa el primer lugar.
  2. `0x0 ... 0x0`: es la cadena de 55 bytes de valores `0x0` correspondientes al padding.
  3. `0x0000000000000000`: es el valor `QWORD` expresado en *Big-Endian* con el tamaño en bits de los datos originales, en este caso 0.

Otro ejemplo:

- Sea `f` un archivo de tamaño 55 bytes.
- El tamaño de los datos después del padding queda definido por:

$$55 + 1 + n + 8 = 64$$

- El menor múltiplo de 64 que puede obtenerse es precisamente 64, con lo cual:

$$64 + n = 64 \Rightarrow n = 0$$

- De modo que no es necesario agregar ningún byte `0x0`, y los datos de entrada adquieren la siguiente estructura tras el padding:

55 bytes de datos de `f | 0x80 | 0x00000000000001B8`

  Donde `0x00000000000001B8` es el `QWORD` en *Big-Endian* con el tamaño en bits de los datos originales de `f` (esto es, 55 Bytes $\times$ 8 = 440 bits, que en hexadecimal es `0x01B8`).

Otro ejemplo más:

- Sea f un archivo de tamaño 56 bytes.
- El tamaño de los datos después del padding queda definido como:

$$56 + 1 + n + 8 = 65$$

- El menor múltiplo de 64 que puede obtenerse es 128, con lo cual:

$$65 + n = 128 \Rightarrow n = 63$$

- Luego, será necesario agregar 63 bytes `0x0`, y los datos de entrada se extienden de la siguiente forma:

  56 bytes de datos de `f` | `0x80` | `0x0 0x0 ... 0x0` | `0x00000000000001C0`

El siguiente pseudocódigo es la descripción formalmente el proceso de padding del algoritmo SHA-1:

Función `Padding`

- **Entrada**: conjunto de datos m de cualquier longitud
- **Salida**: conjunto de datos m extendido

//
// Variables locales.
//
r, s, t: entero
q: `QWORD`

$t =$ `tamaño`(m)
$s = t + 9$

//
// Determinamos el tamaño total de los datos m cuando le
// sumamos los 9 bytes correspondientes al byte `0x80`, seguido
// por el valor `QWORD` asociado al tamaño en bits de los datos m.
//
$r = s \mod 64$

//
// Si al sumar los 9 bytes al tamaño de m no obtuvimos un
// múltiplo de 64, extenderemos el número de bytes que
// agregaremos para alcanzar el primer múltiplo de 64.
//
Si $(r \neq 0)$ **entonces**
  $s = s + (64 - r)$
Fin Si

Extender m en s bytes

//
// La variable t almacena el tamaño de los datos m originales antes
// de la extensión. La utilizamos como índice para insertar el primer
// byte del padding.
//
$m[t] =$ `0x80`

//
// Rellenamos con la cantidad de ceros que correspondan.
//
Para $i = t + 1$ hasta $($`tamaño`$(m) - 9)$, **hacer**
  $m[i] =$ `0x0`
Fin Para

//
// Obtenemos el valor `QWORD` en formato *Big-Endian* del tamaño
// en bits de los datos m originales.
//
$q =$ `BigEndian_QWORD`$(t \times 8)$

//
// Copiamos cada byte del `QWORD` en los últimos 8 bytes de m.
//
Para $i = 0$ hasta $i = 7$, **hacer**
$m[\texttt{tamaño}(m) - 8 + i] = q[i]$
Fin Para

Devolver m

Fin Función

3.3. Análisis del algoritmo SHA-1 y estrategia de vectorización

3.3.1. Primera optimización del algoritmo SHA-1

Si repasamos la estructura general del algoritmo SHA-1 y abstraemos los ciclos que lo componen, obtenemos algo como lo siguiente:

...

Para cada bloque b de 64 bytes en los datos m, **hacer**

...

Para $i = 0$ hasta 15, **hacer**
$w[i] = b[i]$
Fin Para

...

Para $i = 16$ hasta 79, **hacer**
$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$
Fin Para

...

Para $i = 0$ hasta 79, **hacer**
...
$aux = ... + w[i]$
...
Fin Para

...

Fin Para

...

Es fácil ver que el cuerpo del ciclo externo lo componen, en realidad, otros dos ciclos que van de 0 a 79; el primero de ellos partido en dos etapas (la primera, de 0 a 15, y la segunda de 16 a 79).

Por otra parte, se observa que en las rondas:

- De 0 a 15, el cómputo del $w[i]$ se obtiene de forma directa accediendo a los datos.
- De 16 a 79, el cómputo del $w[i]$ requiere de otros w previos, en el rango $w[i-16]$ a $w[i-3]$. Pero estos elementos, siempre existen.
- De 0 a 79, se efectúa una lectura sobre el elemento $w[i]$.

por lo que no existe ningún tipo de dependencia especial que impida que los cómputos en el cuerpo de cada ciclo puedan realizarse en un único ciclo anidado, que cubra el rango 0 a 79. De esta forma, obtenemos la primera optimización del algoritmo mediante la siguiente transformación:

...

Para cada bloque b de 64 bytes en los datos m, **hacer**

...

Para $i = 0$ hasta 79, **hacer**

Si $(i < 16)$ **entonces**

$w[i] = b[i]$

Sino

$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$

Fin Si

...

$aux = ... + w[i]$

...

Fin Para

...

Fin Para

...

Esta nueva versión del algoritmo SHA-1 es la que emplearemos como base tanto en la versión escalar como en la vectorizada. El pseudocódigo que sigue es la descripción completa de nuestro nuevo algoritmo SHA-1 optimizado:

Función `SHA-1`

- Entrada: conjunto de datos m de cualquier longitud en bytes
- Salida: vector $h[5]$ de `DWORD` con el valor del hash SHA-1

$h[5]$: vector de `DWORD` // Valor de hash.
$w[80]$: vector de `DWORD` // Vector denominado *Message Schedule*.
a, b, c, d, e, f, k: `DWORD` // Variables denominadas de *estado*.
aux: `DWORD` // Variable auxiliar.

//
// Inicialización del vector h.
//
$h[0] =$ `0x67452301`
$h[1] =$ `0xEFCDAB89`
$h[2] =$ `0x98BADCFE`
$h[3] =$ `0x10325476`
$h[4] =$ `0xC3D2E1F0`

//
// Padding o relleno. Después del padding, los datos de entrada m
// se extenderán en algunos bytes, y su tamaño total será múltiplo
// de 64 bits.
//
$m =$ `Padding`(m)

//
// Procesamiento de cada bloque de 64 bytes de los datos
// de entrada m para generar el hash.
//
Para cada bloque b de 64 bytes en los datos m, **hacer**

$a = h[0]$
$b = h[1]$
$c = h[2]$
$d = h[3]$
$e = h[4]$

Para $i = 0$ hasta 79, **hacer**

//
// Procesamiento del vector w (Message Schedule).
//
Si $(i < 16)$ **entonces**
$w[i] = b[i]$
Sino
$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$
Fin Si

//
// Actualización de las variables a, b, c, d y e.
//
Si $(0 \leq \texttt{i} \leq 19)$ **entonces**
$f = (b \wedge c) \vee ((\neg\ b) \wedge d)$
$k = \texttt{0x5A827999}$
Sino Si $(20 \leq i \leq 39)$ **entonces**
$f = b \oplus c \oplus d$
$k = \texttt{0x6ED9EBA1}$
Sino Si $(40 \leq i \leq 59)$ **entonces**
$f = (b \wedge c) \vee (b \wedge d) \wedge (c \wedge d)$
$k = \texttt{0x8F1BBCDC}$
Sino
$f = b \oplus c \oplus d$
$k = \texttt{0xCA62C1D6}$
Fin Si

$aux = (a \hookleftarrow 5) + f + e + k + w[i]$
$e = d$
$d = c$
$c = b \hookleftarrow 30$
$b = a$
$a = aux$

Fin Para

//
// Actualización del valor de hash.
//
$h[0] = h[0] + a$
$h[1] = h[1] + b$
$h[2] = h[2] + c$
$h[3] = h[3] + d$
$h[4] = h[4] + e$

Fin Para

Devolver h
Fin Función

3.3.2. Análisis de la factibilidad de la vectorización

Si por un momento nos abstraemos del ciclo anidado, notaremos que el ciclo externo no hace más que inicializar las variables estado al comienzo, y actualizar el vector con el valor de hash al finalizar. No hay optimización posible en estas operaciones.

Analicemos ahora el ciclo anidado. Se distinguen dos partes esenciales en la nueva versión optimizada del algoritmo SHA-1:

1. El procesamiento del vector w (message schedule).
2. La actualización de las variables de estado a, b, c, d y e.

donde la segunda depende de la primera.

En el caso de la actualización de las variables de estado, es fácil ver que existe una circularidad que obliga indefectiblemente al procesamiento secuencial. Esta circularidad queda definida en el último bloque de código del ciclo anidado:

$aux = (a \hookleftarrow 5) + f + e + k + w[i]$
$e = d$
$d = c$
$c = b \hookleftarrow 30$
$b = a$
$a = aux$

y está dada por el siguiente esquema:

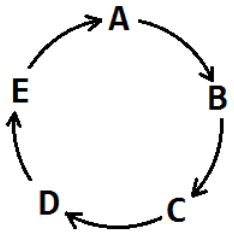

Figura 2: dependencia circular en el algoritmo SHA-1

De esta forma, cualquier posibilidad de vectorización en esta etapa queda descartada.

Analicemos ahora la operatoria sobre el vector `w`:

Para $i = 0$ hasta 79, **hacer**

Si $(i < 16)$ **entonces**
$w[i] = b[i]$
Sino
$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$
Fin Si
...
Fin Para

Para el primer caso en la sentencia de decisión, la vectorización es inmediata: basta con acceder al buffer con los datos de entrada y cargar 4 `DWORDs` en cualquier registro `XMM`. Este proceso podrá repetirse unas 4 veces. A partir del elemento 16, entramos en la sección alternativa de la sentencia **Si**, y los elementos del vector w se computan siguiendo la expresión más compleja, dependiente de cuatro elementos del previos: $w[i-3]$, $w[i-8]$, $w[i-4]$ y $w[i-16]$.

Veamos qué ocurre si pretendemos repetir el procedimiento anterior, procesando de forma simultánea los cuatro elementos $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$:

- $w[i]$ requiere de: $w[i-3]$, $w[i-8]$, $w[i-14]$ y $w[i-16]$
- $w[i+1]$ requiere de: $w[i-2]$, $w[i-7]$, $w[i-13]$ y $w[i-15]$
- $w[i+2]$ requiere de: $w[i-1]$, $w[i-6]$, $w[i-12]$ y $w[i-14]$
- $w[i+3]$ requiere de: $w[i]$, $w[i-5]$, $w[i-11]$ y $w[i-13]$

Como puede observarse, el problema es $w[i+3]$: este elemento requiere de $w[i]$, que se encuentra, al igual que el propio $w[i+3]$, en pleno proceso de cómputo en nuestro cálculo vectorizado.

Una posibilidad es reducir el cómputo vectorizado a los primeros 3 elementos $w[i]$, $w[i+1]$ y $w[i+2]$, y luego repetir el procedimiento para w[i+3] de forma individual, utilizando el ya calculado $w[i]$. Esta alternativa claramente es poco performante, ya que fuerza a la repetición del cómputo y perjudica la idea de la vectorización.

Afortunadamente, existe otra alternativa mucho más eficiente, que se basa en la siguiente propiedad:

Propiedad: *Distributividad de la operación* rotación *a nivel de bit.*

Sean:

- A y B dos valores binarios de igual longitud n en bits.
- L un operador lógico binario cualquiera a nivel de bit.
- r un valor entero en el rango $0 \leq r \leq n$
- $\hookleftarrow$ la operación *rotación a izquierda* a nivel de bit.

Entonces:

$$L(A,B) \hookleftarrow r = L(A \hookleftarrow r, B \hookleftarrow r)$$

Demostración:

Expresemos A y B a nivel de bit:

$$A = a_0\ a_1\ a_2\ \ldots\ a_n$$
$$B = b_0\ \ b_1\ b_2\ \ldots\ b_n$$

donde a_i y b_j corresponden al bit $i-$ésimo y $j-$ésimo de A y B, $0 \leq i, j \leq n$, respectivamente.

Dado que L es un operador lógico binario a nivel de bit, $L(A,B)$ puede escribirse como:

$$L(A,B) = L(a_0,b_0)\ L(a_1,b_1)\ \ldots\ L(a_n,b_n) = \ c_0\ c_1 \ldots\ c_n$$

Con lo cual:

$$L(A,B) \hookleftarrow r = c_r\ c_{r+1}\ c_{r+2}\ \ldots\ c_n\ c_0\ c_1\ \ldots\ c_{r-1} \tag{1}$$

Por otra parte:

$$\begin{aligned} L(A \hookleftarrow r,\ B \hookleftarrow r) &= L(a_r\ a_{r+1}\ \ldots\ a_n\ a_0\ a_1\ \ldots\ a_{r-1},\ b_r\ b_{r+1}\ \ldots\ b_n\ b_0\ b_1\ \ldots\ b_{r-1}) = \\ &= L(a_r,b_r)\ L(a_{r+1},b_{r+1})\ \ldots\ L(a_n,b_n)\ L(a_0,b_0)\ L(a_1,b_1)\ \ldots\ L(a_{r-1},b_{r-1}) = \\ &= c_r\ c_{r+1}\ \ldots\ c_n\ c_0\ c_1\ \ldots c_{r-1} \end{aligned} \tag{2}$$

De donde resulta que (1) y (2) son iguales, como queríamos demostrar. ■

Volviendo a nuestro problema, la propiedad anterior posibilita reescribir la expresión:

$$w[i+3] = (w[i] \oplus w[i-5] \oplus w[i-11] \oplus w[i-13]) \hookleftarrow 1$$

de la siguiente forma:

$$w[i+3] = (w[i] \hookleftarrow 1) \oplus ((w[i-5] \oplus w[i-11] \oplus w[i-13]) \hookleftarrow 1)$$

A su vez, la conmutatividad de la operación lógica $\oplus$ permite modificar el orden de los términos:

$$w[i+3] = ((w[i-5] \oplus w[i-11] \oplus w[i-13]) \hookleftarrow 1) \oplus (w[i] \hookleftarrow 1)$$

Finalmente, aprovechamos la identidad:

$$A \oplus 0 = 0 \oplus A = A$$

Y reescribimos la expresión anterior como:

$$w[i+3] = [(0 \oplus w[i-5] \oplus w[i-11] \oplus w[i-13]) \hookleftarrow 1] \oplus (w[i] \hookleftarrow 1)$$

Con todo, podemos ahora realizar el cómputo los cuatro elementos $w[i]$, $w[i+1]$, $w[i+2]$ y el problemático $w[i+3]$ de forma simultánea, reemplazando al elemento $w[i]$ requerido y no disponible para el cómputo de $w[i+3]$ por el valor 0, y realizando un ajuste al final para completar el cómputo del $w[i+3]$. Este ajuste consiste en realizar una rotación a izquierda en un 1 bit sobre una copia del elemento $w[i]$ ya obtenido, y luego ejecutar la $\oplus$ entre este último y el $w[i+3]$ parcial para obtener el valor definitivo.

El siguiente esquema ilustra la propuesta. Cada grupo horizontal de elementos `w` puede ser almacenado en registros `XMM`, y las operaciones $\oplus$ sobre cada elemento se realizan fácilmente mediante la instrucción `PXOR` (los detalles se describen en la sección 3.6):

$w[i-3]$	$w[i-2]$	$w[i-1]$	**0**
$\oplus$	$\oplus$	$\oplus$	$\oplus$
$w[i-8]$	$w[i-7]$	$w[i-6]$	$w[i-5]$
$\oplus$	$\oplus$	$\oplus$	$\oplus$
$w[i-14]$	$w[i-13]$	$w[i-12]$	$w[i-11]$
$\oplus$	$\oplus$	$\oplus$	$\oplus$
$w[i-16]$	$w[i-15]$	$w[i-14]$	$w[i-13]$
$\Vert$	$\Vert$	$\Vert$	$\Vert$
$w'[i]$	$w'[i+1]$	$w'[i+2]$	$w'[i+3]$
$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
$(w'[i] \hookleftarrow 1)$	$(w'[i+1] \hookleftarrow 1)$	$(w'[i+2] \hookleftarrow 1)$	$(w'[i+3] \hookleftarrow 1)$
$\Vert$	$\Vert$	$\Vert$	$\Vert$
w[i]	**w[i+1]**	**w[i+2]**	$w''[i+3]$
$\oplus$	$\oplus$	$\oplus$	$\oplus$
0	0	0	**(w[i]** $\hookleftarrow 1$**)**
$\Vert$	$\Vert$	$\Vert$	$\Vert$
w[i]	**w[i+1]**	**w[i+2]**	**w[i+3]**

3.3.3. Propiedad de *Max Locktyukhin*

Max Locktyukhin[8], ingeniero ruso de Intel, se ocupó del algoritmo SHA-1, y observó la siguiente propiedad:

> El cómputo de la extensión de w del algoritmo SHA-1, para toda ronda i, $i \geq 32$, puede calcularse como:
>
> $$w[i] = (w[i-6] \oplus w[i-16] \oplus w[i-28] \oplus w[i-32]) \hookleftarrow 2$$

Demostración:

La demostración se basa en la siguiente idea:

Propiedad

Sean a y b dos valores enteros positivos cualesquiera, tales que $i - a \geq 0$ e $i - b \geq 0$.

Si:

$$w[i] = w[i-a] \oplus w[i-b] \tag{1}$$

entonces:

$$w[i] = w[i-2a] \oplus w[i-2b] \tag{2}$$

Demostración:

Según (1), $w[i-a]$ y $w[i-a]$ pueden obtenerse como:

$$w[i-a] = w[i-a-a] \oplus w[i-b-a] = w[i-2a] \oplus w[i-b-a] \tag{3}$$

$$w[i-b] = w[i-a-b] \oplus w[i-b-b] = w[i-a-b] \oplus w[i-2b] \tag{4}$$

Reemplazando (3) y (4) en (1), resulta:

$$w[i] = (w[i-2a] \oplus w[i-b-a]) \oplus (w[i-a-b] \oplus w[i-2b]) \tag{5}$$

Por la conmutatividad de la operación $\oplus$, podemos reacomodar los términos de (5) de la siguiente forma:

$$w[i] = (w[i-2a] \oplus w[i-2b]) \oplus (w[i-b-a] \oplus w[i-a-b]) \tag{6}$$

Se observa que el último paréntesis de (6) vale 0, puesto que se trata de una $\oplus$ entre elementos iguales ($w[i-a-b]$). Con lo cual:

$$w[i] = (w[i-2a] \oplus w[i-2b]) \oplus 0 = w[i-2a] \oplus w[i-2b] \tag{7}$$

De donde puede verse que (7) es igual a (2), como queríamos demostrar. ■

Lema:

Sea B un valor binario cualquiera de longitud n en bits, y r y s dos enteros positivos tales que $0 \leq r, s \leq n$. Entonces, vale la identidad:

$$(B \hookleftarrow r) \hookleftarrow s = B \hookleftarrow (r+s)$$

La demostración es trivial.

[8]Max Locktyukhin es un *Software Performance Engineer* de la *Intel Corp.*, especialista en procesos de optimización tanto a nivel de hardware como de software.

Corolario:

Sea:

$$w[i] = (w[i-a] \oplus w[i-b]) \hookleftarrow r \tag{1}$$

Entonces:

$$w[i] = (w[i-2a] \oplus w[i-2b]) \hookleftarrow 2r \tag{2}$$

Demostración:

$$w[i-a] = (w[i-2a] \oplus w[i-b-a]) \hookleftarrow r \tag{3}$$

$$w[i-b] = (w[i-a-b] \oplus w[i-2b]) \hookleftarrow r \tag{4}$$

Reemplazando (3) y (4) en (1):

$$w[i] = (((w[i-2a] \oplus w[i-b-a]) \hookleftarrow r) \oplus ((w[i-a-b] \oplus w[i-2b])) \hookleftarrow r)) \hookleftarrow r$$

Por la propiedad de distributividad de la operación de rotación:

$$w[i] = (((w[i-2a] \oplus w[i-b-a]) \oplus (w[i-a-b] \oplus w[i-2b])) \hookleftarrow r) \hookleftarrow r$$

Aplicando el *Lema 1* y la conmutatividad de la operación $\oplus$:

$$w[i] = ((w[i-2a] \oplus w[i-2b]) \oplus (w[i-a-b] \oplus w[i-b-a])) \hookleftarrow r+r = (w[i-2a] \oplus w[i-2b]) \hookleftarrow 2r$$

como queríamos demostrar. ■

De esta forma, extendiendo el corolario anterior al caso de 4 elementos w en lugar de 2, la propiedad de Max Locktyukhin se demuestra de forma inmediata.

La propiedad de Max Locktyukhin permite, ahora sí, el procesamiento simultáneo de 4 elementos del vector w sin operaciones adicionales. Tiene la desventaja, sin embargo, de que funciona para valores de índice $i \geq 32$, dado que el elemento más chico del cual depende es $w[i-32]$.

Este inconveniente de alguna forma se ve compensado por una situación bastante afortunada: los elementos $w[i-16]$, $w[i-28]$ y $w[i-32]$ quedan alineados correctamente para el caso del procesamiento vectorizado en registros `XMM`. El elemento $w[i-6]$ es el único que requiere ser reubicado en el registro, pero existen instrucciones que realizan esa operación (los detalles se describen en la sección 3.5, *Instrucciones SSSE3 especiales*).

3.4. Resumen de la estrategia de vectorización del algoritmo *SHA-1*

Del análisis anterior del algoritmo SHA-1, resultó que solo el procesamiento del vector w, denominado *Message Schedule*, está libre de circularidades y dependencias especiales que impidan la posibilidad de vectorización.

Apuntamos, por tanto, a esta etapa, y como resultado del análisis y de las disintas propiedades lógicas que hemos visto, encontramos 3 situaciones que dan lugar al procesamiento paralelo:

1. Rondas 0 a 15:
   - El proceso es inmediato.
   - Basta con cargar 16 bytes en algún registro `XMM` directo del buffer con los datos.
   - Esos 16 bytes corresponden a 4 elementos `DWORD` del vector w.
2. Rondas 16 a 31:
   - A partir de la ronda 16 comienza el cómputo complejo de los elementos del vector w.
   - Por las características del cálculo, donde el elemento más adelantado depende del primero en el mismo proceso de cómputo, el tratamiento vectorizado capaz de procesar 4 elementos de forma simultánea debe, en principio, acotarse a solo 3.

- Sin embargo, diversas propiedades lógicas mostraron que es posible avanzar con el cómputo vectorizado de los 4 elementos hasta la última operación.
- En ese momento, el valor requerido por el elemento más adelantado es reemplazado por un valor neutro en el marco de la operación lógica en cuestión.
- Producto de este truco, se obtienen los primeros tres elementos vectorizados, y con ajuste de bajo de costo, es posible obtener el restante.

3. Rondas 32 a 79:
   - La propiedad de *Max Locktyukhin*, permite el procesamiento vectorizado de todos elementos del vector w sin inconveniente.

3.5. Detalles de la implementación del algoritmo SHA-1 vectorizado

A continuación se describen las ideas más importantes incluidas en la implementación de nuestra versión vectorizada del algoritmo SHA-1. Los detalles específicos a nivel de código se detallan en la sección 3.6.

1. *Loop unrolling* y *software pipelining*:

   Todos los algoritmos en este trabajo han sido implementados siguiendo la técnica del *loop unrolling*, y adicionalmente en el caso vectorizado, el *pipelining*, que se describen en la sección 3.6. Como se explicó, el objetivo es favorecer la performance evitando el *overhead* asociado a las instrucciones requeridas en el chequeo de las condiciones de los ciclos. En el caso de nuestra implementación, desplegaremos el ciclo `for` anidado, que se desplaza en el rango 0 a 79. De modo que cada una de las 80 rondas se computan secuencialmente. Además, el *software pipelining* se implementa con un *prólogo* y *epílogo* que procesan 16 elementos del vector w. Tal como se explicará en la sección correspondiente, esto es posible gracias a las macros.

2. Vector w almacenado en búffer circular que emplea registros `XMM`:

   El vector w es de uso frecuente en el cómputo del valor del hash SHA-1. Resultaría muy conveniente mantenerlo de alguna forma en registros del CPU, en lugar de la memoria. En principio, no disponemos de registros suficientes para almacenarlo completamente, dada su extensión de 80 elementos `DWORDs`.

   Sin embargo, no es necesario mantener la totalidad de sus elementos en memoria: de los análisis previos observamos que la dependencia más lejana para un $w[i]$ determinado, es $w[i-32]$, situación que aparece en la propiedad de Max Locktyukhin. De modo que para toda ronda basta con almacenar los 32 elementos de w precedentes. Con esa idea, implementamos un buffer circular compuesto por 8 registros `XMM`:

```
XMM0 =    w[i] |  w[i+1] |  w[i+2] |  w[i+3] -> W0
XMM1 =  w[i+4] |  w[i+5] |  w[i+6] |  w[i+7] -> W1
XMM2 =  w[i+8] |  w[i+9] | w[i+10] | w[i+11] -> W2
XMM3 = w[i+12] | w[i+12] | w[i+13] | w[i+14] -> W3
XMM4 = w[i+16] | w[i+17] | w[i+18] | w[i+19] -> W4
XMM5 = w[i+20] | w[i+21] | w[i+22] | w[i+23] -> W5
XMM6 = w[i+24] | w[i+25] | w[i+26] | w[i+27] -> W6
XMM7 = w[i+28] | w[i+29] | w[i+30] | w[i+31] -> W7
```

   Las cotas del buffer están dadas por los elementos $w[i]...w[i+31]$. Esta es precisamente la distancia de dependencia más grande entre dos elementos de w, en las fórmulas vectorizadas que analizamos en las secciones anteriores.

   El buffer es circular porque los elementos a partir de $w[i+32]$ se almacenan comenzando desde el principio, sobreescribiendo los valores anteriores.

3. Cómputo de $w + k$:

   La participación de los elementos de w en el cómputo del valor de hash SHA-1 aparece en el cálculo de una de las variables de estado del algoritmo:

   $$aux = (a \hookleftarrow 5) + f + e + k + w[i]$$

   Notemos que el valor de k es uno de cuatro constantes predefinidas, que se selecciona según el número de ronda:

   Rondas 0 a 19: $k = k0 =$ `0x5A827999`
   Rondas 20 a 39: $k = k1 =$ `0x6ED9EBA1`
   Rondas 40 a 59: $k = k2 =$ `0x8F1BBCDC`
   Rondas 60 a 79: $k = k3 =$ `0xCA62C1D6`

   Es fácil ver que el k asociado a cualquier grupo de elementos $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$ en el cómputo vectorizado es siempre el mismo para los cuatro:

$w[0] \to k0$	...	$w[16] \to k0$	$w[20] \to k2$	...	$w[40] \to k3$	...	$w[76] \to k4$
$w[1] \to k0$	...	$w[17] \to k0$	$w[21] \to k2$	...	$w[41] \to k3$	...	$w[77] \to k4$
$w[2] \to k0$	...	$w[18] \to k0$	$w[22] \to k2$	...	$w[42] \to k3$	...	$w[78] \to k4$
$w[3] \to k0$	...	$w[19] \to k0$	$w[23] \to k2$	...	$w[43] \to k3$	...	$w[79] \to k4$

   Con todo, resulta conveniente aprovechar esta propiedad y realizar la suma también de forma vectorizada. Almacenaremos para ello las constantes k en otros cuatro registros `XMM`:

   ```
   XMM8  = 0x5A827999 | 0x5A827999 | 0x5A827999 | 0x5A827999 -> K0
   XMM9  = 0x6ED9EBA1 | 0x6ED9EBA1 | 0x6ED9EBA1 | 0x6ED9EBA1 -> K1
   XMM10 = 0x8F1BBCDC | 0x8F1BBCDC | 0x8F1BBCDC | 0x8F1BBCDC -> K2
   XMM11 = 0xCA62C1D6 | 0xCA62C1D6 | 0xCA62C1D6 | 0xCA62C1D6 -> K3
   ```

4. Vector del hash y variables de estado en registros:

   Tanto el vector h donde se computa el valor de hash SHA-1, como las 5 variables de estado a, b, c, d y e, también son de uso frecuente. De modo que para favorecer la performance los almacenamos en registros escalares. La sección 3.6 muestra los detalles.

5. Optimización del proceso de actualización de las variables de estado:

   En la sección 3.3.2 comprobamos que el proceso de actualización de las variables de estado a, b, c, d, y e no puede vectorizarse debido a una dependencia circular que existe entre ellas. Sin embargo, esa circularidad da lugar a otra forma de optimización.

   Repasemos las operaciones en el proceso de actualización de estas variables:

   $$\begin{aligned} aux &= (a \hookleftarrow 5) + f + e + k + w[i] \\ e &= d \\ d &= c \\ c &= b \hookleftarrow 30 \\ b &= a \\ a &= aux \end{aligned}$$

   La implementación más inmediata de este proceso consistiría, en primer lugar, del cómputo de la variable aux, para luego copiar el valor de d en e, el de c en d, etc. Es decir, existe un overhead importante producto del movimiento de datos entre las variables.

   Sin embargo, ese overhead no es necesario: basta con aplicar las operaciones sobre las variables que sí se ven modificadas, y luego reinterpretarlas a todas como si hubieran sido rotadas (esto es, la variable a pasa a interpretarse como b, la variable b como c, y así sucesivamente). Veamos con más detalle esta idea:

- Ronda 0:
  Solo modificamos las variables b y e:

$$b = b \hookleftarrow 30$$
$$e \mathrel{+}= (a \hookleftarrow 5) + f + e + k + w[i]$$

- Ronda 1:
  Reinterpretamos el orden de las variables:

a ahora es la que antes era e
b ahora es la que antes era a
c ahora es la que antes era b
d ahora es la que antes era c
e ahora es la que antes era d

  Con lo cual, la actualización ahora es:

$$a = a \hookleftarrow 30$$
$$d \mathrel{+}= (e \hookleftarrow 5) + ...$$

- Ronda 2:

a ahora es la que antes era d
b ahora es la que antes era e
c ahora es la que antes era a
d ahora es la que antes era b
e ahora es la que antes era c

  La actualización ahora es:

$$e = e \hookleftarrow 30$$
$$c \mathrel{+}= (d \hookleftarrow 5) + ...$$

y así sucesivamente. Como resultado, el código de actualización de las variables se simplifica considerablemente. De modo que en la implementación, como veremos, la macro que se ocupa de este proceso computa de a dos rondas por vez, en lugar de una, para que este proceso no se vuelva demasiado atómico.

6. *Interleaving* o entrelazado:

   Cuando en la sección 3.3.2 analizamos las posibilidades de vectorización del algoritmo SHA-1, encontramos dos partes o etapas bien diferenciadas: el procesamiento del vector w, y la actualización de las variables de estado (que son las que, en definitiva, generan el valor de hash).

   Vimos que la segunda depende de la primera, dado que para cualquier ronda i la segunda etapa requiere del elemento $w[i]$. Vimos además que la segunda etapa no puede vectorizarse debido a una dependencia circular que existe entre sus variables.

   Como consecuencia, la segunda etapa solo puede avanzar de a un paso a la vez vez, mientras que la primera puede hacerlo de a cuatro, producto de la vectorización.

   Esta situación fuerza algún tipo de coordinación entre el avance de una y otra etapa. El objetivo es evitar que el cómputo de los elementos de w se adelante lo suficiente como para sobreescribir un elemento aun no consumido por la segunda etapa.

   Son varias las alternativas para coordinar este avance: computar 4 elementos de w, consumir 4, computar otros 4 w, etc.; computar los 32 w, consumir los 32, computar otros 32, etc.

   Sin embargo, existe otra estrategia que, si bien es más compleja, no solo da lugar a un esquema de coordinación mucho más homogéneo, sino que también favorece considerablemente el interleaving o entrelazado entre las instrucciones escalares e instrucciones *SIMD*: se trata de la partición del cómputo vectorizado en cuatro fases, donde cada fase quedará determinada por el número de ronda.

   Es importante destacar que, a partir de esta idea, el número de ronda cumple dos funciones en el marco del cómputo vectorizado: sirve como discriminante entre las tres formas de cómputo del vector w (copia directa (rondas 0 a 15), computo cuasi-vectorizado (rondas 16

a 31); cómputo vectorizado (*Max Locktyukhin*) (rondas 32 a 79)), y además funciona como selector de una de las cuatro fases en el proceso de cálculo.

Veamos, entonces, cómo se desarrolla el cómputo vectorizado de w para las primeras 8 rondas:

- Ronda 0: fase 0 del cómputo vectorizado de $w[0]$, $w[1]$, $w[2]$ y $w[3]$.
- Ronda 1: fase 1 del cómputo vectorizado de $w[0]$, $w[1]$, $w[2]$ y $w[3]$.
- Ronda 2: fase 2 del cómputo vectorizado de $w[0]$, $w[1]$, $w[2]$ y $w[3]$.
- Ronda 3: fase 3 del cómputo vectorizado de $w[0]$, $w[1]$, $w[2]$ y $w[3]$ (completado).

  El registro `XMM0`, alias `W0`, incluye ahora el valor de $w[0]$, $w[1]$, $w[2]$ y $w[3]$.

- Ronda 4: fase 0 del cómputo vectorizado de w[4], w[5], w[6] y w[7].
- Ronda 5: fase 1 del cómputo vectorizado de w[4], w[5], w[6] y w[7].
- Ronda 6: fase 2 del cómputo vectorizado de w[4], w[5], w[6] y w[7].
- Ronda 7: fase 3 del cómputo vectorizado de w[4], w[5], w[6] y w[7] (completado).

  El registro `XMM1`, alias `W1`, incluye ahora el valor de $w[4]$, $w[5]$, $w[6]$ y $w[7]$.

  ...

Como podemos observar, este esquema requiere que el cómputo vectorizado se inicie lo suficientemente antes que el cómputo escalar que actualiza las variables de estado, de manera tal que al iniciar este último ya existan elementos de w calculados. En la sección 3.6 veremos que en la implementación del algoritmo SHA-1 vectorizado nos adelantamos al cómputo escalar tanto como los primeros 16 elementos del vector w (aquellos que obtenemos directamente del buffer), como resultado de la aplicación del *software pipelining*.

7. Instrucciones *SSSE3* especiales.

   Se describen a continuación dos instrucciones incluidas en la extensión SSSE3 que resultan de especial utilidad para la implementación del cómputo vectorizado, como así también la desventaja de no contar con una instrucción que permita realizar una rotación empaquetada, y el modo en que solventamos el problema.

   - Instrucción `PSHUFB`

     Hemos visto que los datos para los primeros 16 elementos del vector w (rondas 0 a 15) se obtienen directamente desde el buffer. En nuestro caso vectorizado, cargamos de 4 DWORDs de forma simultánea en un registro XMM, mediante una instrucción como la siguiente:

     ```
     MOVDQU XMM0, [BUFFER_PTR]
     ```

     Ocurre que la lectura directa del buffer acarrea un inconveniente: los bytes en el buffer figuran de manera secuencial siguiendo el mismo orden original que tenían en el archivo. Podemos decir, por lo tanto, que el contenido del archivo se carga en memoria en formato 'Big-Endian': el byte menos significativo se ubica en la posición más baja de memoria, mientras que el más significativo en la más alta. Veamos un ejemplo:

     Sea `BUFFER_PTR` un puntero a datos en memoria:

     ```
       ...

     BUFFER_PTR:      0x00
     BUFFER_PTR + 1:  0x01
     BUFFER_PTR + 2:  0x02
     BUFFER_PTR + 3:  0x03
       ...

     BUFFER_PTR + 13: 0x0D
     BUFFER_PTR + 14: 0x0E
     BUFFER_PTR + 15: 0x0F
     ```

Sea la instrucción:

```
MOVDQU XMM0, [BUFFER_PTR]
```

El contenido del registro XMM0 es exactamente el siguiente:

```
XMM0: 0x00010203 | 0x04050607 | 0x08090A0B | 0x0C0D0E0F
```

Cada `DWORD` figura en formato *Big-Endian*, cuando en realidad la arquitectura Intel representa los valores en formato *Little-endian* (el byte más significativo se ubica en la posición más baja, mientras que el menos significativo en la más alta).

Afortunadamente, la extensión SSSE3 incluye la instrucción `PSHUFB`. `PSHUFB` ejecuta un *shuffling* o reordenamiento empaquetado a nivel de byte especialmente útil en situaciones como estas.

Trabaja con dos operandos: un registro `XMM` sobre el cual llevará a cabo el reordenamiento, y otro registro `XMM` (o una posición de memoria) donde se especifica la reubicación de cada byte del primero.

Para comprender la mecánica de esta instrucción, consideramos que cada byte de un registro `XMM` se enumera de izquierda a derecha, y de 0 a 7. Entonces, la constante de reubicación debería ser la siguiente:

```
little_endian_bytes_shuffling_values DB 0x03, 0x02, 0x01, 0x00,\
0x07, 0x06, 0x05, 0x04,\
0x0b, 0x0a, 0x09, 0x08,\
0x0f, 0x0e, 0x0d, 0x0c
```

Esta constante establece tiene el siguiente efecto: el byte en la posición 0 (el primero) en el registro `XMM`, lo reubica al byte en la posición 3; el byte en la posición 1 lo reubica a la posición 2, y así sucesivamente.

Con lo cual, retomando el ejemplo anterior, debería incluirse la siguiente instrucción para reubicar los `DWORDs` en formato Little-endian:

```
PSHUFB XMM0, [little_endian_bytes_shuffling_values]
```

El formato de cada valor DWORD ahora es el correcto:

```
XMM0: 0x03020100 | 0x07060504 | 0x0B0A0908 | 0x0F0D0E0C
```

El resultado es que convertimos 4 `DWORDs` en *Little-endian* empleando una sola instrucción.

- Instrucción `PALIGNR`

  En la sección 3.3.2 vimos que el cómputo vectorizado complejo de w se divide en dos partes: la primera, abarca las rondas 16 a 31, y ejecuta el cómputo siguiendo la definición original del algoritmo SHA-1:

  $$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$$

  la segunda, abarca las rondas 32 a 79, y opera con la expresión de *Max Locktyukhin*:

  $$w[i] = (w[i-6] \oplus w[i-16] \oplus w[i-28] \oplus w[i-32]) \hookleftarrow 2$$

  Teniendo en cuenta que el vector w es almacenado en el buffer descrito en el ítem 2 anterior, donde cada registro `XMM` del buffer almacena cuatro elementos de w, comenzando siempre por índices múltiplo de 4, encontraremos la dificultad de que los elementos $w[i]$ no quedan correctamente alineadas en los cómputos paralelos. Veamos como ejemplo el segundo caso, y consideremos el cómputo vectorizado en la ronda 32:
  Nos interesa obtener $w[32]$, $w[33]$, $w[34]$, $w[35]$:

	$w[32]$	$w[33]$	$w[34]$	$w[35]$	$\leftarrow w[i]$
XMM0:	$w[0]$	$w[1]$	$w[2]$	$w[3]$	$\leftarrow w[i-32]$
XMM1:	$w[4]$	$w[5]$	$w[6]$	$w[7]$	$\leftarrow w[i-28]$
		...			
XMM4:	$w[16]$	$w[17]$	$w[18]$	$w[19]$	$\leftarrow w[i-16]$
		...			
XMM6:	$w[24]$	$w[25]$	$w[26]$	$w[27]$	
XMM7:	$w[28]$	$w[29]$	$w[30]$	$w[31]$	

Observamos que tres de los cuatro términos en el cómputo paralelo quedan perfectamente alineados: $w[i-32]$, $w[i-28]$ y $w[i-16]$. El término restante, $w[i-6]$, se encuentra, en cambio, repartido entre los registros `XMM6`, y `XMM7`. En concreto, necesitamos almacenar en algún registro `XMM` auxiliar lo siguiente:

$$w[26] \quad w[27] \quad w[28] \quad w[29] \quad \leftarrow w[i-6]$$

Podemos obtener este resultado mediante copias de los registos `XMM6` y `XMM7`, seguido por desplazamientos a derecha y a izquierda, y una operación lógica como la `XOR`. Sin embargo, la extensión *SSSE3* incluye la instrucción `PALIGNR`, que realiza esta operación en un solo paso. `PALIGNR` requiere de 3 operandos: dos registros `XMM`, y un valor inmediato que indica el número de bytes que se desplaza un puntero ficticio a derecha. Veamos cómo funcionaría en el caso de nuestro ejemplo:

```
MOVDQU  XMM10, XMM6     ; Hacemos una copia del registro XMM6
                        ; para no destruir el buffer.

PALIGNR XMM10, XMM7, 8  ; XMM10 = w[26]  w[27]  w[28]  w[29]
```

`PALIGNR` opera así:

a) Concatena los registros `XMM10` y `XMM7`:

$$w[24] \quad w[25] \quad w[26] \quad w[27] \quad w[28] \quad w[29] \quad w[30] \quad w[31]$$

b) Considera al elemento de la izquierda ($w[24]$) como el primero, y se desplaza a la derecha tantos bytes como lo indique el tercer operando. En nuestro caso, el tercer operando vale 8. Si tenemos en cuenta que cada $w[i]$ es un `DWORD` (es decir, tienen longitud de 4 bytes), el resultado es que el el puntero ficticio se desplaza al elemento $w[26]$:

$$\underset{}{w[24]} \quad \underset{}{w[25]} \quad \underset{\uparrow}{w[26]} \quad \underset{}{w[27]} \quad \underset{}{w[28]} \quad \underset{}{w[29]} \quad \underset{}{w[30]} \quad \underset{}{w[31]}$$

c) Finalmente, la operación extrae los 16 bytes siguientes a partir del puntero ficticio, y los copia en el registro `XMM` de destino (en nuestro caso, `XMM10`).

$$\texttt{XMM10} = w[26] \quad w[27] \quad w[28] \quad w[29]$$

- Ausencia de instrucciones de rotación empaquetada.

  Lamentablemente la extensión SSSE3 no incluye una instrucción que permita ejecutar rotaciones empaquetadas sobre los registros SIMD, operación requerida como último paso en la obtención de los w. Sin embargo, la siguiente identidad nos permitirá emularla:

  Sea A un valor binario de longitud n en bits, y r un valor entero, $0 \leq r \leq n$. Entonces:

$$A \leftrightarrow r = (A \ll r) \vee (A \gg (n-r))$$

  o también:

$$A \leftrightarrow r = (A \ll r) \oplus (A \gg (n-r))$$

  Al igual que las anteriores, la demostración es inmediata, operando a nivel de bit. De forma análoga, se prueba el caso para la rotación a derecha.

  De manera que empleando desplazamientos a izquierda y derecha, y la operación lógica `OR` (o alternativamente `XOR`), podemos ejecutar una operación de rotación.

  Como ejemplo, supongamos que debemos rotar a izquierda en 20 bits a cada `DWORD` (32 bits) empaquetado en el registro `XMM0`. El procedimiento es el siguiente:

```
MODQU XMM1, XMM0
PSLLD XMM0, 20
PSRLD XMM1, 12    ; 12 = 32 - 20
POR   XMM0, XMM1
```

3.6. Análisis del código

Habiendo descripto las estrategias para la vectorización del algoritmo SHA-1, los detalles relativos a la implementación de estas ideas, las técnicas algorítmicas incluidas que favorecen la performance, las características especiales de la extensión SSSE3, el modo en que resolvemos algunos inconvenientes, pasamos ahora al análisis del código ensamblador del módulo que implementa el cómputo vectorizado del algoritmo, incluyendo los comentarios que correspondan para facilitar su comprensión en el código:

- Vector de hash `H` y variables de estado `A`, `B`, `C`, `D`, y `E` en registros:

```
%xdefine H0 RDX
%xdefine H1 RDX + 4
%xdefine H2 RDX + 8
%xdefine H3 RDX + 12
%xdefine H4 RDX + 16

  ...

%xdefine A R11D
%xdefine B R12D
%xdefine C R13D
%xdefine D R14D
%xdefine E R15D
```

- Buffer circular para el vector w:

```
%xdefine W0 XMM0  ; w[i]    | w[i+1] | w[i+2] | w[i+3]
%xdefine W1 XMM1  ; w[i+4]  |       ...       | w[i+7]
%xdefine W2 XMM2  ; w[i+8]  |       ...       | w[i+11]
%xdefine W3 XMM3  ; w[i+12] |       ...       | w[i+15]
%xdefine W4 XMM4  ; w[i+16] |       ...       | w[i+19]
%xdefine W5 XMM5  ; w[i+20] |       ...       | w[i+23]
%xdefine W6 XMM6  ; w[i+24] |       ...       | w[i+27]
%xdefine W7 XMM7  ; w[i+28] |       ...       | w[i+31]
```

- Constantes k también almacenadas en registros `XMM`:

```
%xdefine K0 XMM8
%xdefine K1 XMM9
%xdefine K2 XMM10
%xdefine K3 XMM11

  ...

k1 DD 0x5A827999, 0x5A827999, 0x5A827999, 0x5A827999
k2 DD 0x6ED9EBA1, 0x6ED9EBA1, 0x6ED9EBA1, 0x6ED9EBA1
k3 DD 0x8F1BBCDC, 0x8F1BBCDC, 0x8F1BBCDC, 0x8F1BBCDC
k4 DD 0xCA62C1D6, 0xCA62C1D6, 0xCA62C1D6, 0xCA62C1D6

  ...

MOVDQU K0, [k1]
MOVDQU K1, [k2]
MOVDQU K2, [k3]
MOVDQU K3, [k4]
```

- Macro para la actualización de las variables de estado: actualiza las variables de estado a, b, c, d y e para dos rondas consecutivas:

```
%macro UPDATE_ABCDE 6

  ;
  ; Rondas 0 a 19
  %if (%1 <= 19)
     %xdefine F F1
  ;
  ; Rondas 20 a 39
  %elif (%1 <= 39)
    %xdefine F F2

    ...

  %endif

    ...

  ;
  ; Actualizamos la variable 'E', que en la próxima ronda será
  ; 'A'.
  ;
  ; E = E + f + ROTATE_LEFT(A, 5) + w[i] + k
  ;
  MOV EBX, %2           ; EBX = A1
  ADD %6, EAX           ; E1  = E1 + f
  ROL EBX, 5            ; EBX = ROTATE_LEFT(A1, 5)
  ADD %6, [wk + 4*%1]   ; E1  = E1 + f + ROTATE_LEFT(A1, 5) + w[i] + k
  ADD %6, EBX           ; E1  = E1 + f + ROTATE_LEFT(A1, 5)

    ...

%endmacro
```

- Macros para el cómputo del vector w y $w + k$:

```
;
; Parámetros:
;
; %1: número de ronda (solo válida para i = 0..15).
; %2: dirección del buffer.
;
%macro CALCULATE_WK 2

  %if (%1 < 16)
    CALCULATE_WK_FOR_ROUNDS_0_TO_15 %1, %2
  %elif (%1 < 32)
    CALCULATE_WK_FOR_ROUNDS_16_TO_31 %1
  %elif (%1 < 80)
    CALCULATE_WK_FOR_ROUNDS_32_TO_79 %1
  %endif

%endmacro

  ...
```

```
  ...

;
; Parámetros:
;
; %1: número de ronda (solo válida para i = 0..15).
; %2: dirección del buffer.
;
%macro CALCULATE_WK_FOR_ROUNDS_0_TO_15 2

  ;
  ; Aplicamos una 'and' al índice 'i'. Esto nos permitirá
  ; distinguir el número de etapa.
  %assign i (%1 & 3)

  ;
  ; Etapa 1: rondas 0, 4, 8, 12. Simplemente leemos del buffer.
  ;
  %if (i == 0)
      ...

  %elif (i == 1)
      ...

    PSHUFB W, LITTLE_ENDIAN_BYTES_SHUFFLING
      ...

  %endif
      ...

%endmacro

;
; Parámetro:
;
; %1: número de ronda.
;
%macro CALCULATE_WK_FOR_ROUNDS_16_TO_31 1
  ...

%endmacro

;
; Parámetro:
;
; %1: dirección del buffer con los datos.
;
%macro CALCULATE_WK_FOR_ROUNDS_32_TO_79 1
    ...

  MOVDQU  XMM_AUX0, W_MINUS_4     ; XMM_AUX0 = w[i-1] | ... | w[i-4]
  PALIGNR XMM_AUX0, W_MINUS_8, 8  ; XMM_AUX0 = w[i-3] | ... | w[i-6]

  PXOR W, XMM_AUX0 ; W = ...| w[i-32] XOR w[i-28] XOR w[i-16] XOR w[i-6]

    ...

%endmacro
```

- Entrada del programa:

  Notemos el *prólogo* del *software pipelining*, en el que se procesan los primeros 16 elementos del vector w, el *despliegue* del ciclo (*loop unrolling*) combinado con el entrelazado entre las macros `CALCULATE_WK` y `UPDATE_ABCDE`, y el *epílogo* del *software pipelining* que involucra únicamente a la macro `UPDATE_ABCDE` hacia el final.

```
    ...

;
; Calculamos por adelantado 16 elementos del vector W (64
; bytes).
CALCULATE_WK 0, BUFFER_PTR
CALCULATE_WK 1, BUFFER_PTR
CALCULATE_WK 2, BUFFER_PTR
CALCULATE_WK 3, BUFFER_PTR  ; Obtenemos W0: w[0] | w[1] | w[2] | w[3]

    ...

CALCULATE_WK 12, BUFFER_PTR
CALCULATE_WK 13, BUFFER_PTR
CALCULATE_WK 14, BUFFER_PTR
CALCULATE_WK 15, BUFFER_PTR ; Obtenemos W3: w[12] | ... | w[15]

    ...

;
; Ciclo del algoritmo desplegado (loop unrolling).
;
loop:

    CALCULATE_WK 16, BUFFER_PTR
    UPDATE_ABCDE 0,  A, B, C, D, E
    CALCULATE_WK 17, BUFFER_PTR

    CALCULATE_WK 18, BUFFER_PTR
    UPDATE_ABCDE 2,  D, E, A, B, C
    CALCULATE_WK 19, BUFFER_PTR     ; Obtenemos w[16] ... w[19].

    CALCULATE_WK 20, BUFFER_PTR
    UPDATE_ABCDE 4,  B, C, D, E, A
    CALCULATE_WK 21, BUFFER_PTR

    CALCULATE_WK 22, BUFFER_PTR
    UPDATE_ABCDE 6,  E, A, B, C, D
    CALCULATE_WK 23, BUFFER_PTR     ; Obtenemos w[20] ... w[23].

      ...

    ;
    ; Cómputo de cierre.
    ;
    UPDATE_ABCDE 64, B, C, D, E, A
    UPDATE_ABCDE 66, E, A, B, C, D
    UPDATE_ABCDE 68, C, D, E, A, B
    UPDATE_ABCDE 70, A, B, C, D, E
    UPDATE_ABCDE 72, D, E, A, B, C
    UPDATE_ABCDE 74, B, C, D, E, A
    UPDATE_ABCDE 76, E, A, B, C, D
    UPDATE_ABCDE 78, C, D, E, A, B

      ...
```

Notemos que la rotación de las variables de estado cada vez que se invoca a la macro `UPDATE_ABCDE` se efectúa en saltos de a dos elementos. Como se explicó en secciones anteriores, esto se debe a que la macro `UPDATE_ABCDE` actualiza el valor de las variables para dos rondas consecutivas.

3.7. Experiencia y resultados

La experiencia sobre el algoritmo de hash SHA-1 consistió en la contrastación de la performance de la versión vectorizada que obtuvimos, frente a dos versiones escalares: la primera, también en *ensamblador*; la segunda, en lenguaje *C*. Estas últimas se caracterizan por el procesamiento de un elemento por vez en cada iteración (esto es, un elemento del vector w, seguido por una ronda que actualiza las variables de estado). Sin embargo, tanto una como otra emplea la técnicas de optimización del *Loop Unrolling*, como así también todas las optimizaciones escalares posibles (como por ejemplo, la rotación de las variables de estado (véase sección 3.5, *Optimización del proceso de actualización...*). De modo que el objetivo en este trabajo también incluyó la optimización de las versiones escalares de los algoritmos estudiados.

Las pruebas para evaluar la performance de cada implementación consistieron en la medición del tiempo de CPU de la ejecución de cada una de las tres versión sobre un grupo de 6 archivos. El tamaño de estos archivos varía en el rango de 0,5 a 3,0 *GB*, en saltos de 0,5 *GB*. Sobre cada archivo, cada implementación se ejecutó media docena de veces, y de todas ellas se computó el promedio.

El gráfico de la figura 3 muestra los resultados de la experiencia.

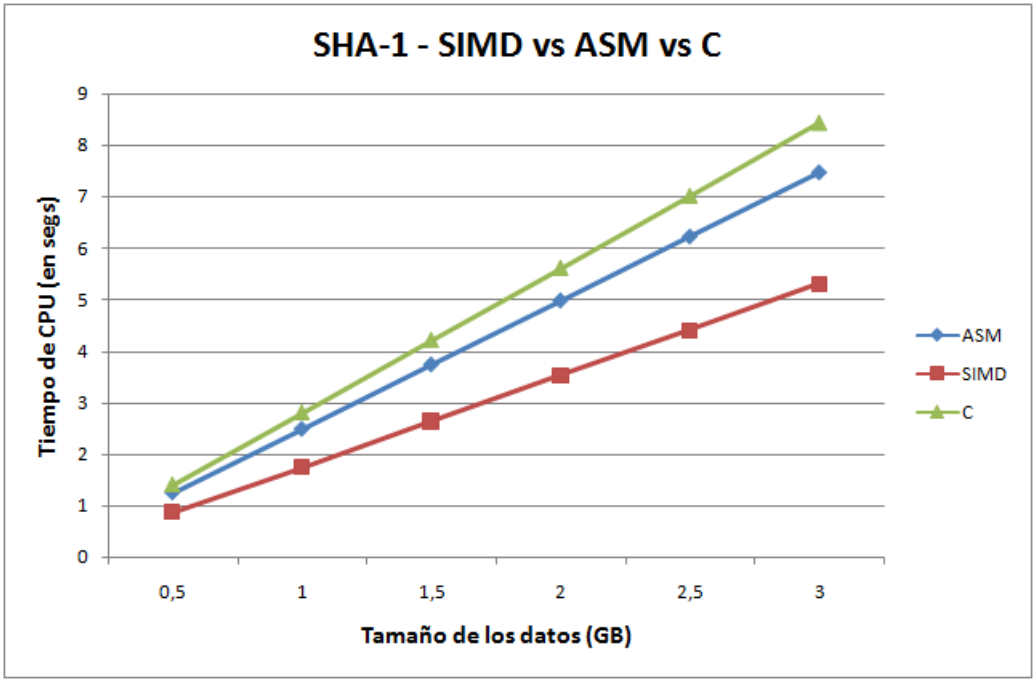

Figura 3: SHA-1 - *SIMD* vs *ASM* vs *C*

El análisis de los datos produjo los siguientes resultados:

- La versión escalar *ASM* resultó un 11,5 % más rápida que la versión escalar en *C*.
- La versión vectorizada *SIMD* resultó un 29,6 % más rápida que la versión escalar en *ASM*.
- La versión vectorizada *SIMD* resultó ser 37,4 % más rápida que la versión escalar en *C*.

3.8. Conclusión

Es importante volver a recordar ahora que SHA-1 no es un algoritmo diseñado para la paralelización. Por el contrario, la vectorización que obtuvimos es, de hecho, parcial, y fue el resultado de un análisis pormenorizado de las partes que componen este algoritmo, proceso durante el cual buscamos todos los elementos posibles susceptibles a la operatoria *SIMD*.

En este marco, afirmar que la implementación cuasi-vectorizada superó a la escalar -también optimizada- en un 30 %, resulta un logro muy considerable. Pensemos, por ejemplo, que lo que a la versión escalar en ensamblador le llevaría 5 minutos, la versión vectorizada lo resuelve en $3,5$, lo que significa una diferencia importante.

Por todo esto, consideramos el resultado obtenido como *muy satisfactorio*.

Respecto a los resultados de las versiones escalares, *ASM* vs *C*, ambas optimizadas por igual, y considerando que la compilación realizada incluyó el máximo nivel de optimización para el caso de *C* (opción `-O3` de `gcc`), encontramos que la pequeña diferencia de casi el 12 % en favor de la versión en ensamblador resulta razonable. Fundamentalmente porque, exceptuando las diferencias del lenguaje, las dos implementaciones son esencialmente idénticas en cuanto a la estructura y a las operaciones realizadas.

4. Algoritmo de hash *SHA-256*

4.1. Descripción

SHA-2 es una familia de algoritmos que incluye a los algoritmos SHA-224, SHA-256, SHA-384 y SHA-512. Se tratan de funciones de hash que reciben un conjunto de datos de cualquier longitud, y devuelve un valor de hash de 224, 256, 384 y 512 bits, respectivamente.

SHA-256 es el algoritmo que hemos seleccionado para realizar la segunda experiencia de contrastación entre la versión escalar y una vectorizada.

La estructura del algoritmo SHA-256 es exactamente igual que la del algoritmo SHA-1, lo que nos permitirá simplificar considerablemente las descripciones. Sin embargo, diferen notablemente en el cómputo de sus elementos. A continuación se listan las similitudes y diferencias entre uno y otro:

- **Similitudes de SHA-256 con SHA-1**:
  - Recibe como entrada cualquier conjunto de datos.
  - Está definido a nivel de bit, pero a los efectos de este trabajo, donde los datos de entrada lo contituyen los archivos, trabajaremos a nivel de byte.
  - Realiza exactamente el mismo proceso de padding que en SHA-1 sobre los datos de entrada.
  - Tiene la misma estructura de ciclos.
  - Parte los datos de entrada, inmediatamente después del padding, en bloques de 64 bytes, los cuales procesa de forma secuencial.
  - El procesamiento de cada bloque de 64 bytes tiene como resultado la actualización del valor de hash, y consiste en:
    1. Copiar directamente los 16 `DWORDs` que componen al bloque en los primeros 16 elementos del vector w (*Message Schedule*).
    2. Realizar cómputos que extienden la cantidad de elementos del vector w.
    3. Utilizar el valor del elemento w de la ronda actual, junto al valor de una constante k, para actualizar variables de estado.
    4. Actualizar las variables de estado en sentido circular.
    5. Actualizar el valor de hash.

- **Diferencias de SHA-256 con SHA-1**:
  - El valor de hash es de mayor longitud: lo conforma un vector de 8 `DWORDs`.
  - Las rondas van de 0 a 63, y no de 0 a 79 como en SHA-1.
  - El vector w, por lo tanto, se extiende a 63 elementos.
  - El cómputo de los elementos extendidos de w es bastante más complejo.
  - Son 64 las constantes[9] k: una asociada a cada $w[i]$, con $0 \leq i \leq 63$.
  - Las variables de estado ahora son 8: a, b, c, d, e, f, g y h.
  - La actualización de las variables de estado utiliza a otras variables auxiliares previas, cuyo cómputo también tiene una complejidad importante.

El *fips-180* describe al algoritmo SHA-256 siguiendo la misma estructura que a SHA-1. Eso implica la inclusión de ciclos anidados inncesarios. Por ese motivo, el pseudocódigo que sigue constituye la versión pulida del algoritmo SHA-256:

[9] Estas constantes k se obtienen de la siguiente manera: se tratan de los primeros 32 bits de la parte decimal de la raíz cúbica de los primeros 64 números primos (es decir, los números primos en el rango 2 a 311, incluyendo a ambas cotas). Por ejemplo, en el caso de 2, el primero de estos valores, procedemos así: computamos la raíz cúbica de 2, conservamos únicamente la parte decimal, y empezamos a convertirla a binario hasta completar un total de 32 bits. Finalmente, convertimos ese valor en hexadecimal. Comprobaremos que el resultado es `0x428A2F98`

Función `SHA-256`

- Entrada: conjunto de datos m de cualquier longitud en bytes
- Salida: vector $hash[8]$ de `DWORD` con el valor del hash SHA-256

```
//
// Variables locales.
//
hash[8]: vector de 8 DWORDs   // Valor de hash.
w[63]:   vector de 80 DWORDs  // Vector Message Schedule.
a, b, c, d, e, f, g, h: DWORD // Variables de estado.
k[63]: vector de DWORD        // Vector para las constantes k.
s0, s1: DWORD                 // Variables auxiliares para el cómputo de w.
S0, S1, maj, ch: DWORD        // Variables auxiliares para la actualización
temp1, temp2 :   DWORD        // de las variables de estado.

k ={0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,
    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,
    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,
    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,
    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,
    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,
    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,
    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,
    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,
    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2}

//
// Inicialización del vector hash.
//
hash[0] =0x6A09E667
hash[1] =0xBB67AE85
hash[2] =0x3C6EF372
hash[3] =0xA54FF53A
hash[4] =0x510E527F
hash[5] =0x9B05688C
hash[6] =0x1F83D9AB
hash[7] =0x5BE0CD19

//
// Padding o relleno. Después del padding, los datos de entrada
// m se extenderán en algunos bytes, y su tamaño total será
// múltiplo de 64 bits.
//
m =Padding(m)
```

//
// Procesamiento de cada bloque de 64 bytes de los datos de
// entrada m para generar el hash.
//
Para cada bloque b de 64 bytes en los datos m, **hacer**

$a = hash[0]$
$b = hash[1]$
$c = hash[2]$
$d = hash[3]$
$e = hash[4]$
$f = hash[5]$
$g = hash[6]$
$h = hash[7]$

Para $i = 0$ hasta 63, **hacer**

//
// Procesamiento del vector w (*Message Schedule*).
//
Si $(i < 16)$ **entonces**
$w[i] = b[i]$
Sino
$s0 = (w[i-15] \hookrightarrow 7) \oplus (w[i-15] \hookrightarrow 18) \oplus (w[i-15] \gg 3)$
$s1 = (w[i-2] \hookrightarrow 17) \oplus (w[i-2] \hookrightarrow 19) \oplus (w[i-2] \gg 10)$
$w[i] = w[i-16] + s0 + w[i-7] + s1$
Fin Si

//
// Actualización de las variables de estado a, b, c,
// d, f, g y h.
//
$S1 = (e \hookrightarrow 6) \oplus (e \hookrightarrow 11) \oplus (e \hookrightarrow 25)$
$ch = (e \wedge f) \oplus ((\neg e) \wedge g)$
$temp1 = h + S1 + ch + k[i] + w[i]$
$S0 = (a \hookrightarrow 2) \oplus (a \hookrightarrow 13) \oplus (a \hookrightarrow 22)$
$maj = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
$temp2 = S0 + maj$

$h = g$
$g = f$
$f = e$
$e = d + temp1$
$d = c$
$c = b$
$b = a$
$a = temp1 + temp2$

Fin Para

//
// Actualización del valor de hash.
//
$hash[0] = hash[0] + a$
$hash[1] = hash[1] + b$
$hash[2] = hash[2] + c$
$hash[3] = hash[3] + d$
$hash[4] = hash[4] + e$
$hash[5] = hash[5] + f$

$hash[6] = hash[6] + g$
$hash[7] = hash[7] + h$

Fin Para

Devolver *hash*

Fin Función

4.2. Análisis del algoritmo SHA-256 y estrategia de vectorización

De la experiencia previa con el algoritmo SHA-1, sabemos que no hay vectorización posible sobre las variables de estado, dada la dependencia circular que existe entre ellas. De modo que el foco lo ponemos, una vez más, en el procesamiento del vector w, que en SHA-256 se computa de la siguiente forma:

Si $(i < 16)$ **entonces**
$w[i] = b[i]$
Sino
$s0 = (w[i-15] \hookrightarrow 7) \oplus (w[i-15] \hookrightarrow 18) \oplus (w[i-15] \gg 3)$
$s1 = (w[i-2] \hookrightarrow 17) \oplus (w[i-2] \hookrightarrow 19) \oplus (w[i-2] \gg 10)$
$w[i] = w[i-16] + s0 + w[i-7] + s1$
Fin Si

El objetivo es computar en paralelo los elementos $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$. Ya vimos que el caso de $0 \leq i < 16$ es trivial. Basta con acceder al buffer y copiar de 4 `DWORDs` de forma simultánea en algún registro `XMM`. Veamos lo que ocurre para el cómputo extendido.

En concreto, $w[i]$ depende de los elementos previos $w[i-15]$ (en $s0$), $w[i-2]$ (en $s1$), y $w[i-16]$ y $w[i-7]$. Es fácil ver que el obstáculo para una vectorización inmediata se encuentra en s1: se define en términos de $w[i-2]$, cuya proximidad con $w[i]$ impide el procesamiento de 4 elementos simultáneos, puesto que el cómputo de los w más adelantados, $w[i+2]$ y $w[i+3]$, requieren, respectivamente, de $w[i]$ y $w[i+1]$, que forman parte del mismo proceso de cálculo vectorizado y, por lo tanto, no existen aún.

A diferencia del algoritmo SHA-1, para el que encontramos soluciones simples, claras y directas frente al mismo problema, veremos a continuación que, en el caso del algoritmo SHA-256, esta situación lamentablemente es insalvable. La prueba es muy sencilla:

En primer lugar, debemos persuadirnos de que la expresión de $w[i]$ no puede simplificarse. La razón de esto es que $w[i]$ se define en términos de una suma de varios elementos de w anteriores, en lugar de una operación $\oplus$ como era el caso de SHA-1. No existe, por lo tanto, la posibilidad de anular términos en sustituciones sucesivas. Por el contrario, las sustituciones expanden considerablemente la expresión, incorporando varios nuevos elementos en cada reemplazo.

De modo que la única forma de computar $w[i-2]$ es sencillamente aplicando la definición anterior, resultando:

$$s0 = (w[i-17] \hookrightarrow 7) \oplus (w[i-17] \hookrightarrow 18) \oplus (w[i-17] \gg 3)$$
$$s1 = (w[i-4] \hookrightarrow 17) \oplus (w[i-4] \hookrightarrow 19) \oplus (w[i-4] \gg 10)$$
$$w[i-2] = w[i-18] + s0 + w[i-9] + s1$$

Como resultado, duplicamos el cómputo que estamos intentado vectorizar, puesto que el cálculo anterior debemos repetirlo para los otros tres elementos: $w[i+1]$, $w[i+2]$ y $w[i+3]$.

Teniendo en cuenta lo anterior, la estrategia más conveniente resulta ser aquella que aplicamos en el algoritmo SHA-1, durante la vectorización de la primera extensión de w (rondas 16 a 31). Esta idea consistía simplemente en realizar el cómputo simultáneo de los cuatro elementos $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$, hasta el punto en que los dos más adelantados no pueden continuar debido al problema de las dependencias. A partir de allí, completábamos el cálculo de los dos primeros y, una vez obtenidos, los utilizábamos para finalizar el cálculo parcial de los dos restantes.

En nuestro caso, el cómputo simultáneo de $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$ abarcará la obtención del $s0$ de cada uno, y la suma parcial $w[i]' = w[i-16] + s0 + w[i-7]$, que no incluye a $s1$. A partir de allí, calculamos los $s1$ asociados a los dos primeros, y con ellos completamos la suma parcial que produce finalmente a $w[i]$ y $w[i+1]$. Por último, empleamos los recién calculados $w[i]$ y $w[i+1]$ para generar los s1 asociados a $w[i+2]$ y $w[i+3]$, y con ellos completamos la suma parcial que finalmente los genera.

El resultado es que solo duplicamos el cómputo de s1, a lo que se le agrega una suma adicional. Esto es por lejos mucho más conveniente que duplicar la totalidad del cómputo de la extensión de w. Esta será, por lo tanto, la rebuscada estrategia que utilizaremos para intentar vectorizar este segundo algoritmo de hashing.

4.3. Detalles de la implementación del algoritmo SHA-256 vectorizado

4.3.1. Operaciones de rotación

Como puede observarse del pseudocódigo descripto en la sección 4, el algoritmo SHA-256 utiliza la operación de rotación (a derecha) con mucha frecuencia en sus expresiones. Tanto en aquellas que intentamos vectorizar, y que involucra al vector w, como en las operaciones escalares que emplean a las variables de estado y auxiliares.

Ya hemos mencionado que la extensión SSSE3 no cuenta con ninguna instrucción que realice rotaciones empaquetada sobre los registros SIMD. Sin embargo, vimos que podemos emularla combinando desplazamientos a izquierda y derecha empaquetados, y operaciones `OR` o `XOR`.

Veamos, por ejemplo, el caso de $s0$:

$$s0 = (w[i-15] \hookrightarrow 7) \oplus (w[i-15] \hookrightarrow 18) \oplus (w[i-15] \gg 3)$$

Operemos sobre la expresión y reemplacemos las rotaciones con la combinación de desplazamiento y operaciones $\oplus$:

$$s0 = [(w[i-15] \gg 7) \oplus (w[i-15] \ll 25)] \oplus [(w[i-15] \gg 18) \oplus (w[i-15] \ll 14)] \oplus (w[i-15] \gg 3)$$

Los términos no solo casi se duplican, sino que, además, el cómputo de esa expresión implica un *overhead* muy importante solo en copias intermedias de $w[i-15]$.

Existe una propiedad análoga a la presentada en 3.3.2 (*Distributividad de la operación rotación..*), solo que ahora la aplicaremos para las operaciones de desplazamiento a izquierda y derecha, y que nos permitirá reducir el número de copias de $w[i-15]$ a solo dos:

Propiedad:

Sean A y B dos valores binarios cualesquiera de longitud n en bits, y r y s dos enteros, $0 \leq r, s \leq n$, con $r \leq s$. Entonces:

$$(A \ll r) \oplus (B \ll s) = [(A \oplus (B \ll (s-r))] \ll r$$

La demostración es inmediata, y se obtiene operando a nivel de bit sobre cada lado de la igualdad.

Retomando la expresión de $s0$, si reacomodamos los términos y aplicamos la propiedad, resulta:

$$
\begin{aligned}
s0 =&[(w[i-15] \ll 14) \oplus (w[i-15] \ll 25)] \oplus \\
&\oplus[(w[i-15] \ll 7) \oplus (w[i-15] \gg 18) \oplus (w[i-15] \gg 3)] = \\
\\
=&[(w[i-15] \oplus (w[i-15] \ll 11)) \ll 14] \oplus \\
&\oplus[((w[i-15] \oplus (w[i-15] \gg 18)) \gg 7) \oplus (w[i-15] \gg 3)] = \\
\\
=&[(w[i-15] \oplus (w[i-15] \ll 11)) \ll 14] \oplus \\
&\oplus[(((w[i-15] \oplus (w[i-15] \gg 18)) \gg 4) \oplus w[i-15]) \gg 3] \blacksquare
\end{aligned}
$$

Como puede observarse, y a pesar de la falsa mala apariencia que podría generar la extensión de la nueva expresión, ahora solo necesitaremos dos copias de $w[i-15]$: la primera, para computar $w[i-15] \ll 11$, y la segunda, para hacer lo propio con $w[i-15] \gg 18$. Con estos dos valores, y el original $w[i-15]$, podemos obtener cada término encerrado entre corchetes, para luego completar el cálculo con la operación $\oplus$.

Por ejemplo, en el caso de la expresión entre corchetes del lado derecho, deberíamos hacer lo siguiente:

1. Hacer una copia de $w[i-15]$.
2. Sobre esa copia, hacer $w[i-15] \gg 18$.
3. Computar una `XOR` sobre la copia ya modificada, y el $w[i-15]$ original.
4. Desplazar ese resultado a derecha 4 bits.
5. Aplicar al resultado una `XOR` utilizando nuevamente a $w[i-15]$ como segundo operando.
6. Aplicar sobre el resultado otro desplazamiento a derecha en 3 bits.

Notemos que solo empleamos una copia del elemento $w[i-15]$.

Este método lo aplicamos tanto en el cómputo vectorizado, como en el cómputo escalar que actualiza a las variables de estado.

4.3.2. Otras optimizaciones

La implementación vectorizada del algoritmo SHA-256 incluye todas las optimizaciones aplicadas en el caso de SHA-1, que son:

1. *Loop unrolling* y *software pipelining*: la situación es análoga al caso de SHA-1: desplegamos las 63 rondas asociadas al procesamiento de cada bloque de 64 bytes y calculamos por adelantado los 16 primeros elementos del vector w, utilizando macros.
2. Vector w almacenado en búffer circular que emplea registros `XMM`: el vector w se alamcena en registros `XMM` para evitar el acceso a memoria. A diferencia de SHA-1, este buffer es mucho más reducido, puesto que la dependencia más lejana entre los elementos de w se da con $w[i-16]$. Serán necesarios, por lo tanto, 4 registros `XMM` para implementar el buffer circular. La sección 4.4 describe los detalles.
3. Cómputo de $w + k$: al igual que en SHA-1, en el algoritmo SHA-256 cada w calculado se utiliza en una suma que involucra una constante k. Esta suma parcial se utiliza para actualizar el valor de una de las variables auxiliares. Por ese motivo, esta operación también se vectoriza.
4. Vector del hash y variables de estado en registros: con fines de optimización, el vector con el valor de hash y las variables de estado se almacenan en registros.

5. Optimización del proceso de actualización de las variables de estado: las rotamos repitiendo la idea aplicada sobre las variables de estado del algoritmo SHA-1 (véase sección 3.5, *Optimización del proceso de actualización...*). De esta forma, evitamos un número considerable de copias innecesarias de los datos.

6. *Interleaving* o entrelazado: nuevamente, dividimos el cómputo vectorizado en 4 etapas, con el objeto de intercalarlo con el cálculo escalar, y de esta forma favorecer la *ejecución fuera de orden*.

4.4. Análisis del código

A continuación describimos cómo y dónde se implementa en el código en esamblador para el módulo vectorizado, cada uno de los aspectos detallados en secciones anteriores:

- Vector de hash Hx, donde x es un valor entre 0 y 7, y variables de estado A, B, C, D, E, F, G y H se almacenan en registros:

```
%xdefine H0 RDX
%xdefine H1 RDX + 4
%xdefine H2 RDX + 8
%xdefine H3 RDX + 12
%xdefine H4 RDX + 16
%xdefine H5 RDX + 20
%xdefine H6 RDX + 24
%xdefine H7 RDX + 28

  ...

%xdefine A R8D
%xdefine B R9D
%xdefine C R10D
%xdefine D R11D
%xdefine E R12D
%xdefine F R13D
%xdefine G R14D
%xdefine H R15D
```

- Buffer circular para el vector w y para $w + k$:

```
%xdefine W0 XMM0  ; w[i]    | w[i+1] | w[i+2] | w[i+3]
%xdefine W1 XMM1  ; w[i+4]  |       ...       | w[i+7]
%xdefine W2 XMM2  ; w[i+8]  |       ...       | w[i+11]
%xdefine W3 XMM3  ; w[i+12] |       ...       | w[i+15]

  ...

%xdefine WK0 XMM4  ; w[i]    + k[i]    |     ...     | w[i+3]  + k[i+3]
%xdefine WK1 XMM5  ; w[i+4]  + k[i+4]  |     ...     | w[i+7]  + k[i+7]
%xdefine WK2 XMM6  ; w[i+8]  + k[i+8]  |     ...     | w[i+11] + k[i+11]
%xdefine WK3 XMM7  ; w[i+12] + k[i+12] |     ...     | w[i+15] + k[i+15]
```

- Constantes k en memoria:

  El algoritmo SHA-256 emplea 64 constantes k asociadas a cada $w[i]$, $0 \leq i \leq 63$. No disponemos de suficientes registros `XMM` libres, por lo que estas constantes se almacenan en memoria, y se recuperan luego en la suma vectorizada.

```
k DD 0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5, 0x3956C25B,\
     0x59F111F1, 0x923F82A4, 0xAB1C5ED5, 0xD807AA98, 0x12835B01,\
     0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7,\
     0xC19BF174, 0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,\
     0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA, 0x983E5152,\
     0xA831C66D, 0xB00327C8, 0xBF597FC7, 0xC6E00BF3, 0xD5A79147,\
     0x06CA6351, 0x14292967, 0x27B70A85, 0x2E1B2138, 0x4D2C6DFC,\
     0x53380D13, 0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,\
     0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3, 0xD192E819,\
     0xD6990624, 0xF40E3585, 0x106AA070, 0x19A4C116, 0x1E376C08,\
     0x2748774C, 0x34B0BCB5, 0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F,\
     0x682E6FF3, 0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,\
     0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2
```

- Macro para la actualización de las variables de estado:

```
%macro UPDATE_ABCDEFGH 9
  %assign s (%1 & 12)/4  ; Selector del elemento WK del
                         ; buffer (WK0, WK1, WK2, WK3).
  %assign i (%1 & 3)     ; Selector del elemento DWORD
                         ;en el WK anterior.
  %if   (s == 0)
    %xdefine WK WK0

  %elif (s == 1)
    %xdefine WK WK1

    ...

  %endif

    ...

  ;
  ; Obtenemos 'S1':
  ;
  ;  S1 = ROTATE_RIGHT(E, 6)^ROTATE_RIGHT(E, 11)^ROTATE_RIGHT(E, 25)
  ;
  ; Que puede escribirse:
  ;
  ;  S1 = ROTATE_RIGHT(E^ROTATE_RIGHT(E XOR ROTATE_RIGHT(E, 14), 5), 6)
  ;
  MOV EBX, %6  ; EBX = E
  ROR EBX, 14
  XOR EBX, %6
  ROR EBX, 5
  XOR EBX, %6
  ROR EBX, 6   ; EBX = S1
    ...

%endmacro
```

- Macros para el cómputo del vector w y $w + k$:

```
;
; Esta macro centraliza las llamadas a las otras macros que
; calculan los elementos del vector WK de forma vectorizada.
%macro CALCULATE_WK 2

  %if (%1 < 16)
  CALCULATE_WK_FOR_ROUNDS_0_TO_15 %1, %2
  %else
    CALCULATE_WK_FOR_ROUNDS_16_TO_63 %1
  %endif

%endmacro

  ...

;
; Parámetros:
;
;   %1: número de ronda (solo válida para i = 0..15).
;   %2: dirección del buffer.
%macro CALCULATE_WK_FOR_ROUNDS_0_TO_15 2

  ;
  ; Aplicamos una 'and' al índice 'i'. Esto nos permitirá
  ; distinguir el número de etapa.
  %assign i (%1 & 3)

  ;
  ; Etapa 1: rondas 0, 4, 8, 12. Simplemente leemos del buffer.
  ;
  %if (i == 0)

    ...

  %else
    MOVDQU WK, [k + (%1 - 3)*4] ; WK = k[i] | ... | k[i+3]
    PADDD  WK, W                ; WK = k[i] + w[i] | ...
      ...
  %endif

%endmacro

;
; Parámetro:
;
;   %1: número de ronda.
;
%macro CALCULATE_WK_FOR_ROUNDS_16_TO_63 1

  ;
  ; Aplicamos una 'and' al índice 'i'. Esto nos permitirá
  ; distinguir el número de etapa.
  %assign i (%1 & 3)

  ;
  ; Selector. Nos permitirá movernos de forma circular por los
  ; búfferes W  (w0, w1, w2 y w3) y WK (WK0, WK1, WK2 y WK3).
  %assign s (%1 & 12)/4
```

```
    ;
    ; Etapa 1: rondas 16, 20, 24, 28, 32, ... 60
    ;
    %if (i == 0)

      ...

    %endif

      ...
    ;
    ; s0 = [(w[i-15] XOR (w[i-15] << 11) << 14] XOR
    ;      XOR  [(((w[i-15] XOR (w[i-15] >> 11)) >> 4) XOR w[i-15]) >> 3]
    ;
    MOVDQU  XMM_AUX1, W_MINUS_12      ; XMM_AUX1 = w[i-9]  | ... | w[i-12]
    PALIGNR XMM_AUX1, W_MINUS_16, 4   ; XMM_AUX1 = w[i-12] | ... | w[i-15]

      ...
  %endmacro
```

- Entrada del programa:

```
    ...

  ;
  ; Calculamos por adelantado 16 elementos del vector W (64
  ; bytes).
  CALCULATE_WK 0, BUFFER_PTR
  CALCULATE_WK 1, BUFFER_PTR
  CALCULATE_WK 2, BUFFER_PTR
  CALCULATE_WK 3, BUFFER_PTR  ; Obtenemos W0: w[0] | w[1] | w[2] | w[3]

    ...

  CALCULATE_WK 12, BUFFER_PTR
  CALCULATE_WK 13, BUFFER_PTR
  CALCULATE_WK 14, BUFFER_PTR
  CALCULATE_WK 15, BUFFER_PTR  ; Obtenemos W3: w[12] | ... | w[15]

    ...

;
; Ciclo del algoritmo desplegado (loop unrolling).
;
loop:

  UPDATE_ABCDEFGH 0,  A, B, C, D, E, F, G, H
  CALCULATE_WK 16, BUFFER_PTR

  UPDATE_ABCDEFGH 1,  H, A, B, C, D, E, F, G
  CALCULATE_WK 17, BUFFER_PTR

  UPDATE_ABCDEFGH 2,  G, H, A, B, C, D, E, F
  CALCULATE_WK 18, BUFFER_PTR

  UPDATE_ABCDEFGH 3,  F, G, H, A, B, C, D, E
  CALCULATE_WK 19, BUFFER_PTR     ; Obtenemos w[16], w[17], w[18] y w[19].

    ...
```

Notemos que, a diferencia de SHA-1, la rotación de las variables de estado se realiza en saltos

de a un elemento. Esto es así porque el proceso de actualización de estas variables en SHA-256 es mucho más complejo, e involucra un número importante de operaciones. No es necesario, por lo tanto, realizar un procesamiento extendido que sume más instrucciones escalares que favorezcan el *interleaving*.

4.5. Experiencia y resultados

En esta segunda experiencia repetimos la idea de evaluar la performance de nuestra versión cuasi-vectorizada del algoritmo SHA-256 con una versión escalar. Sin embargo, en esta oportunidad prescindimos de la versión en *C*, y solo empleamos la versión escalar en ensamblador (*ASM*). Como ya ha sido mencionado en reiteradas oportunidades, la versión escalar incluye todas las optimización posibles.

Las pruebas son exactamente las mismas que las efectuadas para la experiencia del algoritmo SHA-1: media docena de ejecuciones repetidas, empleando cada una de las implementaciones, sobre 6 archivos de datos, que van desde 0,5 *GB* a los 3,0 *GB*, en saltos de 0,5 *GB*. Sobre el total de ejecuciones de cada versión, sobre cada archivo, computamos el promedio. La figura 4 ilustra el resultado de la experiencia.

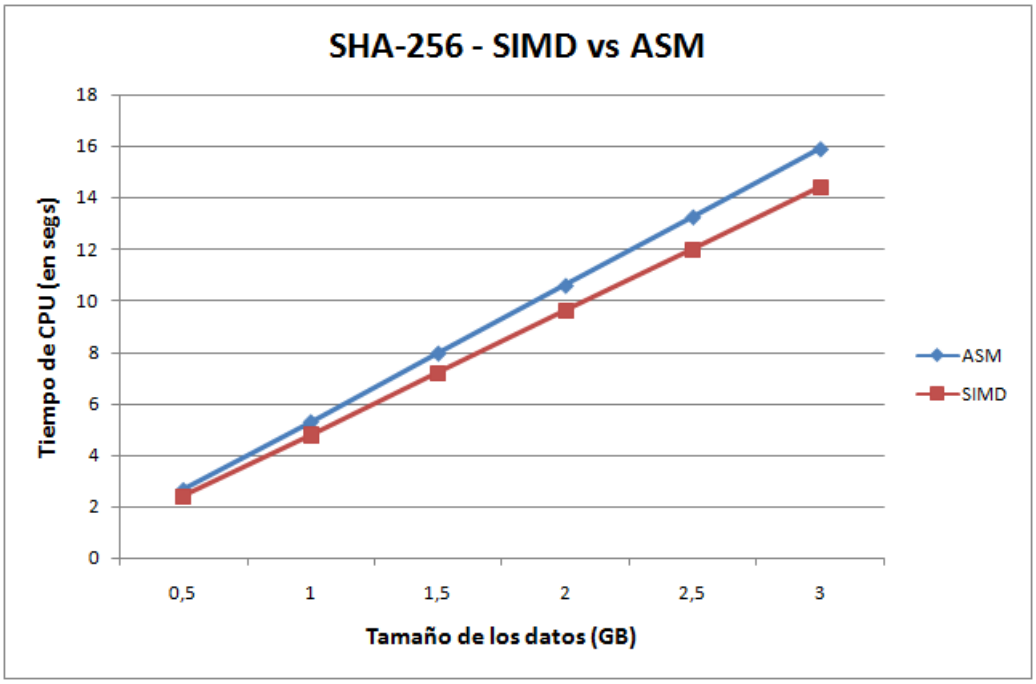

Figura 4: SHA-256 - *SIMD* vs *ASM*

Del análisis de los datos, se desprende que la versión vectorizada *SIMD* resultó, en promedio, un 8,85 % más rápida que la escalar en ensamblador. Es decir, no llega al 10 %.

4.6. Conclusión

A diferencia de lo ocurrido en la experiencia con el algoritmo SHA-1, los resultados obtenidos en el proceso de cuasi-vectorización del algoritmo SHA-256 son bastante pobres. Y esto es algo que, a decir verdad, se anunciaba conforme avanzábamos con el análisis de cada componente de este algoritmo.

La naturaleza compleja de sus expresiones, la insalvable proximidad entre las dependencias, y sobre todo el número de operaciones adicionales requeridas para el tratamiento adecuado de los datos, entre otras dificultades, generaron un *overhead* que claramente afectó la performance de la implementación. De hecho, como consecuencia de lo último, no hubiera sido motivo de sorpresa que la versión escalar obtuviera igual o incluso mejor performance que la versión vectorizada.

En este marco, y considerando las tecnologías a nivel procesador con las que trabajamos, el

hecho de que la intrincada versión vectorizada del algoritmo SHA-256 superara a su contraparte escalar en casi un 10 % es considerado como un resultado *aceptable*.

5. Detalles de las implementaciones y la generación de datos

Se describen a continuación todas las aplicaciones desarrolladas, la estructura de las mismas, el modo de uso, los datos que reciben como entrada y los que generan como salida, el procesamiento de toda esta información para producir los datos de las experiencias, y los *scripts* empleados para automatizar las tareas.

5.1. Estructura y modo de uso de los programas implementados

Como ya fue descripto al comienzo de este documento, el trabajo consistió en tres proyectos: la experiencia correspondiente a la primera optimización algorítimica (*Loop Unrolling*), y las experiencias asociadas al análisis de los algoritmos SHA-1 y SHA-256, respectivamente.

Los programas implementados para cada una de esas experiencias consisten en un módulo principal escrito en lenguaje *C*, desde el cual se invocan a los otros módulos que implementan la distintas versiones de los algoritmos.

Por ejemplo, en el caso de la experiencia SHA-1, el módulo principal es `sha1.c`, y los módulos que implementan al algoritmo en versión escalar (en ensamblador y en *C*), y en versión vectorizada (*SIMD* en ensamblador) son, respectivamente, `sha1_asm.asm`, `sha1_c.c` y `sha1_simd.asm`.

Todos estos proyectos, junto a los respectivos archivos, se encuentran organizados adecuadamente en directorios. Además, cada aplicación incluye el archivo `makefile` correspondientes. Las compilaciones realizadas con `gcc` incluye la optimización de máximo nivel (opción `-O3`).

A continuación se describen los datos de cada proyecto y el modo de cada programa:

1. Experiencia *Loop Unrolling*:
   - Ruta: `fuentes/sha1/sha1_c_vs_c/`
   - Nombre del ejecutable: `sha1`
   - Modo de uso:

     ```
     ./sha1 -opción ruta_absoluta_al_archivo
     ```

     donde opción puede ser:

     `n`: ejecuta la versión *normal* (esto es, sin *loop unrolling*) del algoritmo SHA-1.
     `u`: ejecuta la versión *optimizada* con *loop unrolling* del algoritmo SHA-1.

   - Ejemplo:

     ```
     ./sha1_c_vs_c -u /home/pedrito/un_archivo.dat
     ```

2. Experiencia *SHA-1*:
   - Ruta: `fuentes/sha1/sha1/`
   - Nombre del ejecutable: `sha1`
   - Modo de uso:

     ```
     ./sha1 -opción ruta_absoluta_al_archivo
     ```

     donde opción puede ser:

     `a`: ejecuta la versión escalar en ensamblador del algoritmo SHA-1.
     `c`: ejecuta la versión escalar en C del algoritmo SHA-1.
     `s`: ejecuta la versión vectorizada en ensamblador (*SIMD*) del algoritmo SHA-1.

   - Ejemplo:

     ```
     ./sha1 -s /home/pedrito/otro_archivo.dat
     ```

3. Experiencia *SHA-256*:
   - Ruta: `fuentes/sha256/`
   - Nombre del ejecutable: `sha256`
   - Modo de uso:

     `./sha1 -opción ruta_absoluta_al_archivo`

     donde opción puede ser:

     `a`: ejecuta la versión escalar en ensamblador del algoritmo SHA-256.
     `s`: ejecuta la versión vectorizada en ensamblador (*SIMD*) del algoritmo SHA-256.

   - Ejemplo:

     `./sha256 -s /home/pedrito/otro_archivo_mas.dat`

La ejecución de cada uno de estos programas tiene como resultado la impresión del valor de hash correspondiente, el tamaño del archivo procesado, el tiempo de CPU en segundos, y el nombre del archivo de salida generado. La figura 5 muestra la salida para la aplicación `sha256`, empleando la versión vectorizada:

```
- Cálculo de hash SHA-256 mediante implementación vectorizada en ASM con SIMD:

  da87281c9f9ab6cef8f9362935f4fc864db94606d52212614894f1253461a762

- Tiempo de CPU (en segs): 4.780000
- Tamaño del archivo: 1048576000B
- Salida agregada al archivo 'sha256_simd_out.dat'
```

Figura 5: salida de ejemplo de la aplicación `sha256`

Los datos en el archivo de salida incluye al tamaño del archivo procesado y el tiempo de CPU en segundos que fueron requeridos para computar el valor de hash; ambos campos se separan mediante un *;*. Si un archivo de salida ya existe, los datos se agregan al final.

Las distintas implementaciones para un algoritmo determinado generan archivos de salida diferentes. Por ejemplo, en el caso del algoritmo SHA-1, las ejecuciones con las opciones `a`, `c` y `s` generan, respectivamente, los siguientes archivos de salida: `sha1_asm.out`, `sha1_c.out` y `sha1_simd.out`.

5.2. Generación y procesamiento de los datos

Para automatizar el proceso de generación de archivos, ejecución de las aplicaciones y procesamiento de los datos de salida correspondientes a cada experiencia, empleamos *scripts* para *Bash*. El conjunto de estas herramientas, junto a los archivos generados, puede encontrarse en el directorio `experiencias`.

Si bien cada proyecto (*Loop Unrolling*, SHA-1 y SHA-256) incluye su propio *script* para la generación y procesamiento automático de los datos de salida, se incluye también un *script* general que ejecuta la totalidad de las experiencias en un solo paso. Se trata del *script* `ejecutar_experiencias.sh`, ubicado en el mencionado directorio `experiencias`.

En concreto, los *scripts* generan para cada experiencia archivos temporales que varían en tamaño desde 0,5 *GB* a 3,0 *GB* en saltos de 0,5 *GB*. Eso da un total de 6 archivos de tamaños 0,5 *GB*, 1,0 *GB*, 1,5 *GB*, 2,0 *GB*, 2,5 *GB* y 3,0 *GB*, respectivamente. Estos archivos se generan de a uno por vez, y sobre cada uno de ellos se ejecutan los algoritmos, con sus distintas implementaciones, media docena de veces. Todos y cada uno de estos archivos temporales son eliminados automáticamente cuando finaliza el procesamiento de los datos correspondientes.

Puesto que los *scripts* para *Bash* carecen de capacidad para operar con valores flotantes (requeridos para manipular los tiempos de CPU, los promedios, etc.), los datos generados son procesados por separado en *Excel*.

Se incluyen, por lo tanto, el volcado de estos datos en las distintas planillas de *Excel* asociadas a cada experiencia. En estas planillas se computan promedios, ventajas en términos de porcentajes de una implementación con respecto a otra, gráficos, etc. Las planillas se encuentran en el subdirectorio `Datos procesados (Excel)` de `experiencias`, separadas, a su vez, en otros directorios que se distinguen por el nombre de los algoritmos analizados.

6. Apéndice - Extensiones especiales de los procesadores de *Intel*

6.1. Intel SHA Extensions

Como contraparte a todos nuestros esfuerzos en la optimización por software de los algoritmos de hashing que estudiamos, *Intel* lanza en 2015 la microarquitectura *Goldmont*, la primera en incluir la denominada *Intel SHA Extensions*. Se trata de un conjunto de 7 instrucciones que tienen por objeto la aceleración por hardware en el cómputo de los valores de hash para los dos algoritmos que analizamos precisamente en este trabajo: SHA-1 y SHA-256.

Vale destacar que, contrariamente a lo que pudiera suponerse, este nuevo conjunto de instrucciones no realizan la totalidad de los cómputos, sino que, más bien, facilitan diversos cómputos parciales en ambos algoritmos.

Desafortunadamente, el empleo de estas instrucciones es bastante rebuscado, dado que al hecho de tener que lidiar con los aspectos de por sí intrincados de estos dos algoritmos, se le suman algunas presunciones respecto de al contenido de los operandos, muchas de la cuales involucran las optimizaciones descriptas en este trabajo (como por ejemplo, la suma de $w + k$, o la propiedad circular de las variables de estado a, b, c, etc., que deberán especificarse de forma alternada y en registros diferentes, etc.).

El *Intel SHA Extensions* incluye un total de 7 instrucciones, divididas en dos grupos: 4 instrucciones para SHA-1, y 3 instrucciones para SHA-256. A su vez, cada grupo se divide en dos categorías de instrucciones: instrucciones para el cómputo de la extensión del vector w, e instrucciones para el cómputo de rondas (actualización de las variables a, b, c, etc.). Todas las instrucciones utilizan como operandos los registros `XMM`, direcciones de memorias, y operandos inmediatos de 1 byte. Existe además el caso insólito de una instrucción perteneciente al grupo de SHA-1 que requiere como tercer operando al registro explítico `XMM0`.

La siguiente lista enumera las instrucciones que componen a la extensión SHA de *Intel*:

- SHA-1:
  - Instrucciones para la extensión de w:
    1. `SHA1MSG1` (*SHA-1 - Message 1*)
    2. `SHA1MSG2` (*SHA-1 - Message 2*)
  - Instrucciones para la actualización de las variables de estado (a, b, c, d y e) en las rondas:
    1. `SHA1RNDS4` (*SHA-1 - Rounds 4*)
    2. `SHA1NEXTE` (*SHA-1 - Next E*)
- SHA-256:
  - Instrucciones para la extensión de w:
    1. `SHA256MSG1` (*SHA-256 - Message 1*)
    2. `SHA256MSG2` (*SHA-256 - Message 2*)
  - Instrucciones para la actualización de las variables de estado (a, b, c, d ... h) en las rondas:
    1. `SHA256RNDS2` (*SHA-256 - Rounds 2*)

Veamos en detalle como funcionan.

6.1.1. Instrucciones del *Intel SHA Extensions* para SHA-1

- Cómputo de la extensión del *Message Schedule*

Recordemos antes la expresión para el cómputo de la extensión del vector w (*Message Schedule*) en SHA-1:

$$w[i] = (w[i-3] \oplus w[i-8] \oplus w[i-14] \oplus w[i-16]) \hookleftarrow 1$$

Instrucción `SHA1MSG1`

- **Definición**:

  `SHA1MSG1 XMM1, XMM2/m128`

- **Detalles**:

  Ejecuta el cómputo parcial $w[i-14] \oplus w[i-16]$ de forma vectorizada, en el cálculo para la extensión del vector w, y almacena el resultado en `XMM1`.

- **Operandos**:

  1. `XMM1`: almacena 4 valores consecutivos del vector w a partir de $w[i-16]$, que son: $w[i-16]$, $w[i-15]$, $w[i-14]$ y $w[i-13]$.
  2. `XMM2/m128`: almacena 4 valores consecutivos del vector w a partir de $w[i-14]$, que son: $w[i-14]$, $w[i-13]$, $w[i-12]$ y $w[i-11]$ (los dos últimos no se utilizan).

- **Resultado**:

  `XMM1: w[i-14] XOR w[i-16] | ... | w[i-11] XOR w[i-13]`

Instrucción `SHA1MSG2`

- **Definición**:

  `SHA1MSG2 XMM1, XMM2/m128`

- **Detalles**:

  Completa el cómputo vectorizado que se inició con `SHA1MSG1`, para obtener finalmente los nuevos elementos del vector w: $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$. Sin embargo, su aplicación no es directa, como se describe a continuación.

- **Operandos**:

  1. `XMM1`: almacena 4 valores consecutivos con el cómputo $w[i-8] \oplus w[i-14] \oplus w[i-16]$. Es decir:

     ```
     XMM1: w[i-8] XOR w[i-14] XOR w[i-16] |
           w[i-7] XOR w[i-13] XOR w[i-15] |
           w[i-6] XOR w[i-12] XOR w[i-14] |
           w[i-5] XOR w[i-11] XOR w[i-11]
     ```

     Notemos que el cómputo vectorizado parcial $w[i-14] \oplus w[i-16]$ lo obtuvimos de la intrucción `SHA1MSG1` descripta anteriormente. A ese resultado falta computarle $w[i-8] \oplus ...$ para obtener los valores requeridos en este operando. Esa parte debe realizarse de forma explícita por separado, mediante la instrucción `PXOR`.

  2. `XMM2/m128`: almacena 4 valores consecutivos del vector w a partir de $w[i-4]$, que son: $w[i-4]$, $w[i-3]$, $w[i-2]$ y $w[i-1]$.

- **Resultado**:

  `XMM: w[i], w[i+1], w[i+2] y w[i+3]`

- Actualización de las variables de estado en las rondas

Repasemos el cómputo para la actualización de las variables de estado a, b, c, d y e en cada ronda del algoritmo SHA-1:

Si $(0 \leq \texttt{i} \leq 19)$ **entonces**

$f = (b \wedge c) \vee ((\neg\ b) \wedge d)$
$k =$ `0x5A827999`

Sino Si $(20 \leq i \leq 39)$ **entonces**

$f = b \oplus c \oplus d$
$k =$ `0x6ED9EBA1`

Sino Si $(40 \leq i \leq 59)$ **entonces**

$f = (b \wedge c) \vee (b \wedge d) \wedge (c \wedge d)$
$k =$ `0x8F1BBCDC`

Sino

$f = b \oplus c \oplus d$
$k =$ `0xCA62C1D6`

Fin Si

$aux = (a \hookleftarrow 5) + f + e + k + w[i]$
$e = d$
$d = c$
$c = b \hookleftarrow 30$
$b = a$
$a = aux$

Instrucción `SHA1RNDS4`

- **Definición**:

  `SHA1RNDS4 XMM1, XMM2/m128, imm8`

- **Detalles**: Actualiza las variables de estado a, b, c y d por el equivalente a 4 rondas consecutivas del algoritmo SHA-1. (Nótese la omisión de la variable e. Esto se explica en la instrucción que sigue.)

- **Operandos**:

  1. `XMM1`: almacena las variables de estado a, b, c y d.
  2. `XMM2/m128`: almacena los siguientes valores: $w[i] + e$, $w[i+1]$, $w[i+2]$ y $w[i+3]$.
  3. `imm8`: valor de tamaño `BYTE` que varía entre 0 y 3, y que se utiliza como selector de valores para el par f y k, donde 0 equivale al rango $0 \leq i \leq 19$, 1 al rango $20 \leq i \leq 39$, 2 al rango $40 \leq i \leq 59$, y 3 al rango $60 \leq i \leq 79$ (véase la sentencia de decisión en el framento de código incluido al comienzo de esta sección).

- **Resultado**:

  Registro `XMM1` con el valor de las variables a, b, c y d actualizadas por el equivalente a 4 rondas consecutivas de SHA-1.

Instrucción `SHA1NEXTE`

- **Definición**:

  `SHA1NEXTE XMM1, XMM2/m128`

- **Detalles**: Devuelve los valores $w[i] + e$, $w[i+1]$, $w[i+2]$ y $w[i+3]$.

- **Operandos**: 1) `XMM1`: almacena las variables de estado a, b, c y d. 2) `XMM2/m128`: almacena los siguientes valores: $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$.

- **Resultado**:

  `XMM1: w[i] + E | w[i+1] | w[i+2] | w[i+3]`

- **Observación**:

  La variable e no justifica un cómputo particular, puesto que hacia la cuarta ronda, su valor es el equivalente a la variable a original (es decir, la variable a en la ronda 0), rotada a la izquierda en 30 bits. Esto es consecuencia de la circularidad que vimos que existe entre las variables de estado (véase la sección 3.5, *Optimización del proceso de actualización...*), en donde el valor de la variable a en la ronda 0 pasa a ser el valor de b en la ronda 1, el de c en la ronda 2 con una modificación (es acá que se le aplica la rotación a izquierda en 30 bits), el de d en la ronda 3 (con la modificación anterior) y finalmente el de e en la ronda 4. Por ese motivo, la instrucción realiza este cálculo internamente y ahorra unos pasos más sumándola directamente al elemento $w[i]$, que es, en definitiva, para lo que se usa. De modo la instrucción `SHA1NEXTE` es complentaria a `SHA1RNDS4`.

6.1.2. Instrucciones del *Intel SHA Extensions* para SHA-256

- Cómputo de la extensión del *Message Schedule*

Recordemos antes el cómputo de la extensión del vector w (*Message Schedule*) en SHA-256:

$$s0 = (w[i-15] \hookrightarrow 7) \oplus (w[i-15] \hookrightarrow 18) \oplus (w[i-15] \gg 3)$$
$$s1 = (w[i-2] \hookrightarrow 17) \oplus (w[i-2] \hookrightarrow 19) \oplus (w[i-2] \gg 10)$$
$$w[i] = w[i-16] + s0 + w[i-7] + s1$$

Instrucción `SHA256MSG1`

- **Definición**:

  `SHA256MSG1 XMM1, XMM2/m128`

- **Detalles**:

  Ejecuta el cómputo parcial $s0 + w[i-16]$, de forma vectorizada, en el cálculo para la extensión del vector w, y almacena el resultado en `XMM1`.

- **Operandos**:

  1. `XMM1`: almacena 4 valores consecutivos del vector w a partir de $w[i-16]$, que son: $w[i-16]$, $w[i-15]$, $w[i-14]$ y $w[i-13]$.
  2. `XMM2/m128`: almacena 4 valores consecutivos del vector w a partir de $w[i-12]$, que son: $w[i-12]$, $w[i-11]$, $w[i-10]$ y $w[i-9]$ (los 3 últimos no se utilizan).

- **Resultado**:

  `XMM1: s0 + w[i-16] | s0 + w[i-15] | s0 + w[i-14] | s0 + w[i-13]`

Instrucción `SHA256MSG2`

- **Definición**:

  `SHA256MSG2 XMM1, XMM2/m128`

- **Detalles**:

  Completa el cómputo vectorizado que se inició con `SHA256MSG1`, para obtener finalmente los nuevos elementos del vector w: $w[i]$, $w[i+1]$, $w[i+2]$ y $w[i+3]$. Sin embargo, su aplicación no es directa, como se describe a continuación.

- **Operandos**:

  1. `XMM1`: almacena 4 valores consecutivos con el cómputo $s0 + w[i-16] + w[i-7]$. Es decir:

     ```
     s0 + w[i-16] + w[i-7] | ... | s0 + w[i-13] + w[i-4]
     ```

     Observemos que el cómputo vectorizado parcial $s0 + w[i-16]$ lo obtuvimos de la intrucción `SHA256MSG1` que se decribió antes. Ese resultado es incompleto, y falta sumarle $w[i-7]$ para obtener los valores requeridos en este operando. Esa parte debe realizarse de forma explícita por separado, mediante la instrucción `PADDD`. Pero incluso antes de esa suma empaquetada, también será necesario una instrucción PALIGNR, dado que el elemento `w[i-7]` no es múltiplo de 4. A continuación se muestra un ejemplo donde se realiza esta alineación requerida:

     Asumiendo:

     ```
     XMM1: w[i-8] | w[i-7] | w[i-6] | w[i-5]
     XMM2: w[i-4] | w[i-3] | w[i-2] | w[i-1]
     ```

     Entonces:

     ```
     PALIGNR XMM1, XMM2, 4  ; XMM1: w[i-7] | ... | w[i-4]
     ```

  2. `XMM2/m128`: almacena 4 valores consecutivos del vector w a partir de $w[i-4]$, que son: $w[i-4]$, $w[i-3]$, $w[i-2]$ y $w[i-1]$ (los dos primeros valores no se utilizan).

- **Resultado**:

  `XMM1: w[i], w[i+1], w[i+2] y w[i+3]`

- Actualización de las variables de estado en las rondas

Repasemos el cómputo para la actualización de las variables de estado a, b, c, d, e, f, g y h en cada ronda del algoritmo SHA-256:

$S1 = (e \hookrightarrow 6) \oplus (e \hookrightarrow 11) \oplus (e \hookrightarrow 25)$
$ch = (e \wedge f) \oplus ((\neg e) \wedge g)$
$temp1 = h + S1 + ch + k[i] + w[i]$
$S0 = (a \hookrightarrow 2) \oplus (a \hookrightarrow 13) \oplus (a \hookrightarrow 22)$
$maj = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$
$temp2 = S0 + maj$

$h = g$
$g = f$
$f = e$
$e = d + temp1$
$d = c$
$c = b$
$b = a$
$a = temp1 + temp2$

Instrucción `SHA256RNDS2`

- **Definición**:

  `SHA256RNDS2 XMM1, XMM2/m128, XMM0`

- **Detalles**: Actualiza las variables de estado a, b, c, d, d, e, f, g y h por el equivalente a 2 rondas consecutivas del algoritmo SHA-256.

- **Operandos**:

  1. `XMM1`: almacena las variables de estado c, d, g y h.
  2. `XMM2/m128`: almacena las variables de estado a, b, e y f.
  3. `XMM0`: almacena los valores $w[i] + k[i]$ y $w[i+1] + k[i+1]$; el valor de los dos restantes `DWORDs` se ignoran.

- **Resultado**:

  Registro `XMM1` con el valor de las variables a, b, e y f actualizadas por el equivalente a dos rondas de SHA-256.

- **Más detalles**:

  Esta instrucción puede parecer, en principio, la más intrincada de todo el conjunto. La confusión aparece a raíz de una optimización que se aprovecha, la cual involucra, una vez más, a la propiedad de circularidad de las variables de estado. Veamos en detalle el porqué de la estructura de los operandos, y el modo en que debe interpretarse el resultado.

  En primer lugar, observemos que cuando proyectamos la circularidad de las variables de estado hacia la ronda 2, resulta que la variable c se convierte en la variable a, la d en b, la g en e, y la h en f; todas ellas sin modificación alguna:

Ronda 0:	a	b	c	d	e	f	g	h
Ronda 1:	h'	a	b	c	d'	e	f	g
Ronda 2:	g'	h'	$\underset{\uparrow}{a}$	$\underset{\uparrow}{b}$	c'	d'	$\underset{\uparrow}{e}$	$\underset{\uparrow}{f}$

  Ahora bien, ocurre que los valores de las variables a, b, e y f en los que se convierten, respectivamente, las variables c, d, g y h en la ronda 2, se almacenan precisamente en el segundo operando (`XMM2/m128`). De modo esos valores intactos en el segundo operando se interpretan como los nuevos valores de las variables c, d, g y h para la ronda 2. Por otra parte, sobre el primer operando, que es el operando de destino, se realizan efectivamente cómputos que generan los nuevos valores de a, b, e y f correspondientes a la segunda ronda.

  Este comportamiento explica lo que al principio parecía una extraña distribución de las variables de estado en los operandos de la instrucción: simplemente se separan las variables que se modifican de las que no. Notemos además que el procesamiento secuencial de las rondas, en saltos de a dos debido a la operatoria de la instrucción, se reduce a una alternancia en la especificación de los operandos:

```
SHA256RNDS2 XMM1, XMM2, XMM0
        ...
SHA256RNDS2 XMM2, XMM1, XMM0
        ...
SHA256RNDS2 XMM1, XMM2, XMM0
        ...
SHA256RNDS2 XMM2, XMM1, XMM0
        ...
```

6.2. Extensiones *AVXs* y *BMI2*

6.2.1. Extensiones *AVXs*

AVX son las siglas de *Advanced Vector Extensions*, un conjunto de extensiones y mejoras que involucran la expansión en el tamaño de los registros SIMD y la incorporación de nuevas instrucciones vectoriales, entre otras cosas. Actualmente existen tres versiones, la última próxima a ser lanzada en el mercado: *AVX*, *AVX2* y *AVX-512*

La primera versión de *AVX* fue incorporada en la microarquitectura *Sandy Bridge*, lanzada en 2011 en la línea de procesadores *Core i3*. Uno de las características más especiales de *AVX*

no tiene que ver precisamente con la típica expansión en las capacidades y la incorporación de nuevas operaciones. Más bien, es lo que podría considerarse como un cambio en el paradigma en la programación en ensamblador *x86* moderno: la idea de *instrucciones no destructivas*. En concreto, las nuevas instrucciones incorporan un operando adicional que actúa como destino en una operación determinada. De esta forma, se evita la clásica destrucción del primero de los operandos al ejecutar una instrucción, motivo por el cual frecuentemente es obligación hacer copia del valor en otro registro, en la pila o en la memoria.

Con respecto a la expansión de las capacidades de los registros `XMM`, éstos duplican el tamaño: pasan de 128 a 256 bits. Estos nuevos registros `XMM` ahora se denominan `YMM`.

La primera versión de *AVX* es muy limitada en cuanto al número de instrucciones vectoriales capaces de operar sobre los nuevos registros `YMM`. La gran mayoría de las nuevas instrucciones continúan empleando a los registros `XMM` o a valores inmediatos `QWORD` como operandos.

El cambio verdadero se produce con la aparición de *AVX2*, incluida en la microarquitectura *Haswell*, lanzada al mercado en 2013 en la línea de los nuevos procesadores *Core i3*.

La nueva extensión *AVX2* aumenta considerablemente el soporte de las instrucciones *SSE* sobre *AVX*, a la vez que incorpora un número importante de otras nuevas. Pero además incluye optimizaciones de bajo nivel para muchas de estas operaciones.

Las nuevas instrucciones *SSE* para *AVX* implementan el formato no destructivo mediante un operando adicional, y el nombre de estas operaciones es el mismo que en *SSE*, solo que ahora se debe anteponer el prefijo `'V'`. A continuación ejemplo muy sencillo:

Con *SSE*:

```
   ...
valores1 DD 1, 2, 3, 4
valores2 DD 5, 6, 7, 8
   ...

MOVDQU XMM0, [valores1]  ; XMM0:  1 |  2 |  3 |  4
MODQU  XMM1, [valores2]  ; XMM1:  5 |  6 |  7 |  8
PADDD  XMM0, XMM1        ; XMM0:  6 |  8 | 10 | 12
```

Con *AVX2*:

```
   ...
valores1 DD 1,  2,  3,  4,  5,  6,  7,  8
valores2 DD 9, 10, 11, 12, 13, 14, 15, 16
   ...

VMOVDQU YMM0, [valores1]        ; YMM0:   1,  2,  3,  4,  5,  6,  7,  8
VPADDD  YMM1, YMM0, [valores2]  ; YMM1:  10, 12, 14, 16, 18, 20, 22, 24
```

Notemos el doble del tamaño de los nuevos registros, y la manera en que se agiliza la suma al no tener que hacer copias para evitar la destrucción del registro `YMM0`.

Con todo, son esencialmente tres las mejoras que podemos aplicar sobre los algoritmos de hash SHA-1 y SHA-256 que hemos estudiado:

- Cómputo paralelo de 4 elementos consecutivos de la extensión de w (*Message Schedule*) para 2 bloques de datos: los 256 bits en el tamaño de los registros `YMM` permiten procesar simultáneamente 8 valores `DWORDs`. Considerando que el cómputo de estos elementos es exactamente el mismo para todos los bloques, resulta visiblemente conveniente aprovechar una misma secuencia de cálculo de w operando de forma simultánea sobre los elementos de dos bloques de datos. Sin embargo, esta estrategia requiere operaciones adicionales para detectar y actuar en consecuencia en aquellos casos donde el total de bloques de los datos no seá múltiplo de dos.

- Cómputo paralelo de 1 elemento de la extensión de w (*Message Schedule*) para 8 bloques de datos: la estrategia es la misma que la anterior, solo que ahora operamos sobre un único elemento de w para cada bloque.
- Aprovechar al máximo la ventaja de las operaciones no destructivas, en particular para los desplazamientos a nivel de bit: esto permitirá ahorrarnos las copias de los datos requeridas en el procesamiento en paralelo de los elementos extendidos del vector w.

La próxima mejora aparecerá de la mano de *AVX-512*. Esta nueva versión de la extensión *AVX* incluye, además de nuevos conjuntos de instrucciones, la duplicación en el número y tamaño de los registros `YMM`: se trata de los 32 registros `ZMM`, con un tamaño de 512 bits.

En este caso, las tres estrategias anteriores podrían volver a emplearse sobre los algoritmos de hash estudiados, solo que ahora de manera mucho más extendida, por ejemplo en el procesamiento de archivos de gran tamaño. Quizás sea posible otra variante orientada a aprovechar la capacidad de los registros `ZMM`, que involucre la carga completa de los bloques de datos en estos registros, y emplee combinaciones del repertorio de las nuevas instrucciones para los cómputos vectorizados. Veremos lo que ocurre cuando esta extensión finalmente esté disponible.

6.2.2. Extensión *BMI2*

Además de *AVX2*, la ya mencionada arquitectura *Haswell* introdujo la denominada extensión BMI2. Esta extensión incorpora un número importante de nuevas instrucciones no destructivas con operaciones a nivel de bit, pero en este caso para registros de propósito general. Entre ellas, la instrucción `RORX`, que puede aprovecharse en el algoritmo de hash SHA-256.

La intrucción `RORX` realiza rotaciones a derecha, y devuelve el resultado en el primero de los tres operandos. Ejemplo:

```
MOV  RAX, 2       ; RAX: 8
RORX RBX, RAX, 2  ; RBX: 0xC000000000000000 ; RAX: 8
```

En SHA-256 podemos aprovechar esta instrucción en los cómputos de los elementos $S0$ y $S1$, respectivamente. En concreto, nos ahorraríamos las copias de los datos. Veamos, por ejemplo, el caso de $S1$. La expresión para obtenerlo era la siguiente:

$$S1 = (e \hookrightarrow 6) \oplus (e \hookrightarrow 11) \oplus (e \hookrightarrow 25)$$

El código en ensamblador que emplea a `RORX` podría ser algo como esto:

```
RORX EAX, E, 6    ; EAX: (E >>> 6)
RORX EBX, E, 11   ; EBX: (E >>> 11)
XOR  EAX, EBX     ; EAX: (E >>> 6) XOR (E >>> 11)
RORX EBX, E, 25   ; EBX: (E >>> 25)
XOR  EAX, EBX     ; EAX: (E >>> 6) XOR (E >>> 11) XOR (E >>> 25) = S1
```

Referencias

[1] *FIPS PUB 180-4 - Secure Hash Standard (SHS)*, NIST
http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[2] *Merkle–Damgård Construction*, Wikipedia
https://en.wikipedia.org/wiki/Merkle-Damgård_construction

[3] *SHA: A Design for Parallel Architectures?*, Antoon Bosselaers, René Govaerts and Joos Vandewalle
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.4047&rep=rep1&type=pdf

[4] *Analysis of SIMD Applicability to SHA Algorithms*, O. Aciicmez
https://software.intel.com/sites/default/files/m/b/9/b/aciicmez.pdf

[5] *SHA-1 using SIMD techniques*, Dean Gaudet
http://arctic.org/∼dean/crypto/sha1.html

[6] *Improving the Performance of the Secure Hash Algorithm (SHA-1)*, Max Locktyukhin
https://software.intel.com/en-us/articles/improving-the-performance

[7] *SHA-2*, Wikipedia
https://en.wikipedia.org/wiki/SHA-2

[8] *Advanced Vector Extensions*, Wikipedia
http://en.wikipedia.org/wiki/Advanced_Vector_Extensions

[9] *Fast SHA-512 Implementation on Intel Architecture Processors*, Intel
http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-sha512-implementations-ia-processors-paper.pdf

[10] *Intel SHA Extensions*, Intel
https://software.intel.com/en-us/articles/intel-sha-extensions

[11] *Intel Architecture Instruction Set Extensions Programming Reference*, Intel
https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf

[12] *Haswell New Instruction Descriptions*, Mark Buxton
http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/

[13] *Introduction to Intel Advanced Vector Extensions*, Chris Lomont
https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[14] *Haswell Cryptographic Performance*, Intel
http://www.intel.com/content/www/us/en/communications/haswell-cryptographic-performance-paper.html

[15] *The Netwide Assembler*, NASM
http://www.nasm.us

[16] *Documentation and tutorials*, Flat Assembler
http://flatassembler.net/docs.php?article=manual