Desarrollo e implementación de cores para compresión y descompresión de datos usando LZW sobre la tarjeta Zynq-7000 SoC para SDR

Manuel Hernández Calviño, Jorge Flores Troncoso, Salvador Ibarra Delgado, Remberto Sandoval Aréchiga, Hamurabi Gamboa Rosales, Rodrigo Soulé de Castro

Universidad Autónoma de Zacatecas, Unidad Académica de Ingeniería Eléctrica.
Av. López Velarde 801, Col. Centro, Zacatecas, Zac., México, 98000.
hecm001534@uaz.edu.mx, jflorest@uaz.edu.mx

2014 Published by $\mathcal{DIFU}_{100}ci@$ http://www2.uaz.edu.mx/web/www/publicaciones

Resumen

En este trabajo, se describe el desarrollo, puesta a punto y prueba de 2 cores implementados en hardware sobre FPGA, para la compresión y descompresión de datos, utilizando el conocido algoritmo Lempel-Ziv Welch (LZW). Estos cores serán utilizados posteriormente junto a otros, para crear sistemas de transmisión de datos que utilicen eficientemente el ancho de banda disponible. El algoritmo LZW se basa esencialmente en crear en el compresor una tabla o diccionario donde se almacenan las secuencias de datos de entrada (bytes) que se repiten. Su ventaja reside en que no es necesario transferir la tabla al descompresor, porque éste reconstruye la tabla en la medida que se van recibiendo los códigos enviados por el compresor. Al implementarlos en hardware, se logra un sistema de compresión-descompresión muy eficiente que puede funcionar a alta velocidad, como parte de un sistema de transmisión de datos que así lo requiera.

Palabras clave: Compresión, Descompresión, LZW, FPGA.

1. Introducción

En la actualidad los sistemas de comunicaciones demandan tanto una alta capacidad de almacenamiento de datos al igual que el incremento de la velocidad de procesamiento de los dispositivos. Además los nuevos diseños de dispositivos de comunicaciones tienden a disminuir los costos de memoria así como también el aumento de la tasa de transmisión. En consecuencia surge la necesidad de emplear algoritmos para la compresión de datos.

Para comprimir los datos, en general se busca redundancia en los datos a ser almacenados/transmitidos, y se intenta removerlas aplicando un método o algoritmo de compresión que elimine o remueva dicha redundancia. La redundancia depende del tipo de datos (texto,

imágenes, audio, etc), por tanto, no existe un método de compresión universal que pueda ser óptimo para todos los tipos de datos. El desempeño de los métodos de compresión se mide en base a dos criterios: i) *la razón de compresión* que se obtiene al dividir el total de bytes de los datos comprimido entre el total de bytes de los datos originales, y ii) *el factor de compresión*, siendo este el inverso del primero. Entre mayor redundancia exista en los datos, mejor razón (factor) de compresión será obtenido.

Entre las diferentes técnicas para comprimir datos se encuentran las que usan métodos basados en diccionario, los cuales usan el principio de reemplazar cadenas de datos con codewords que identifican a esa cadena dentro de un diccionario [1]. El diccionario contiene una lista de subcadenas y codewords asociados a cada una de ellas. El diccionario que contiene las cadenas de símbolos puede ser estático o adaptable (dinámico). Prácticamente, todos los métodos de compresión sustitucionales están basados en los métodos de compresión desarrollado por Jacob Ziv y Abraham Lempel en los 70s, los métodos LZ77 y LZ78 [2], [3].

En este trabajo se implementa al algoritmo Lempel-Ziv Welch (LZW) [4] que proviene de una mejora propuesta por Terry Welch (1984) a los algoritmos propuestos por Abrahm Lempel y Jacob Ziv. Se trata de un método de compresión sin pérdidas, ya que los datos cifrados pueden ser perfectamente reconstruidos en el receptor. La implementación se realiza sobre una tarjeta de desarrollo Xilinx Zynq-7000 All Programmable SoC ARM dual-core Cortex-A9.

2. Puesta a punto

2.1. Compresión LZW

La ventaja de LZW radica en su dinamismo, ya que a la vez realiza la codificación de los datos y la generación de nuevas entradas, creando un diccionario de tipo semi-adaptativo que no requiere ser conocido y además es reconstruido por el receptor. Los patrones repetitivos forman una tabla para codificar, donde para cada entrada podrán agruparse varios símbolos concatenados. Este arreglo llamado diccionario o tabla de patrones y es necesario para asignar el código correspondiente a cada salida en el algoritmo. Con lo anterior se puede asignar código de menor cantidad de bits a una mayor cantidad de símbolos que como se designaría normalmente (sin alterar el orden de la constelación y provocar penalizaciones en los errores de bit). La longitud final de código propicia que se reduzca la cantidad de símbolos iniciales.

La Figura 1 muestra el diagrama de flujo para el proceso de codificación de los símbolos (bytes). Como se observa en la Figura 1, se deben inicializar las primeras 256 entradas de la tabla con los 256 caracteres diferentes. El nuevo byte de entrada se asigna a una variable B y se concatena con el índice del anterior, y se busca este en el diccionario. Si existe, índice toma el valor de esta concatenación, de lo contrario se añade esta concatenación al diccinario. Esto ocurre hasta encontrar el último byte.

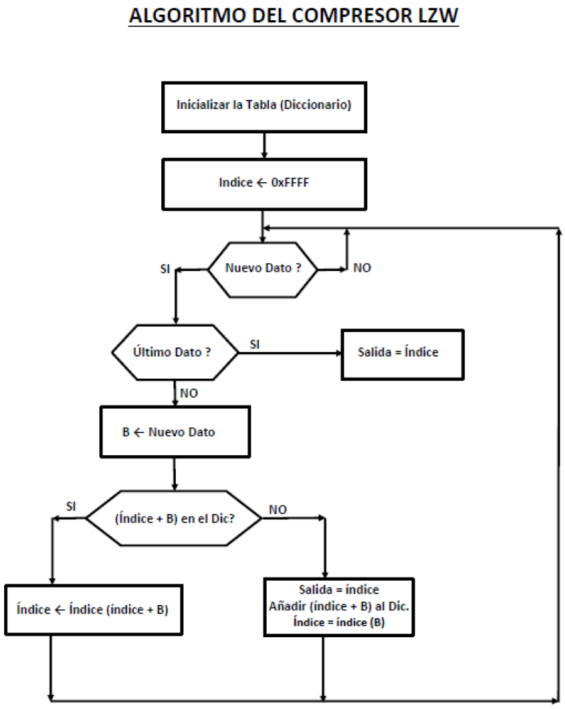

Figura 1. Diagrama de flujo del algoritmo de compresión LZW.

2.2. Descompresión LZW

El descompresor LZW va creando su propio diccionario en la medida que recibe los códigos creados por el compresor. Este diccionario se mantiene actualizado y perfectamente sincronizado. El descompresor es más sencillo y más rápido que el compresor, ya que no tiene que hacer una búsqueda por contenidos en el diccionario, sólo saber si una cadena ya fue almacenada y eso se averigua si índice <tope de la tabla o diccionario. Los pasos que sigue el algoritmo de descompresión son:

1. Inicializar las primeras 256 entradas de la tabla con los 256 caracteres diferentes.
2. índice ← primer código a la entrada
3. Salida ← cadena almacenada en <índice>
4. old ← índice

5. índice ← siguiente código
6. ¿Existe ya la entrada <indice>en el diccionario?
   a) SI
      i. Salida ← cadena almacenada en <índice>
      ii. B ← primer byte de la cadena almacenada en <índice>
      iii. Añadir <old>B al diccionario
   b) NO
      i. B ← primer byte de la cadena almacenada en <old>
      ii. Añadir <old>B al diccionario
      iii. Salida ← cadena <old>B
7. <old>← <índice>

Donde <old>es la posición en el diccionario de la cadena anteriormente recibida. La Figura 2 es un diagrama de flujo del algoritmo de descompresión.

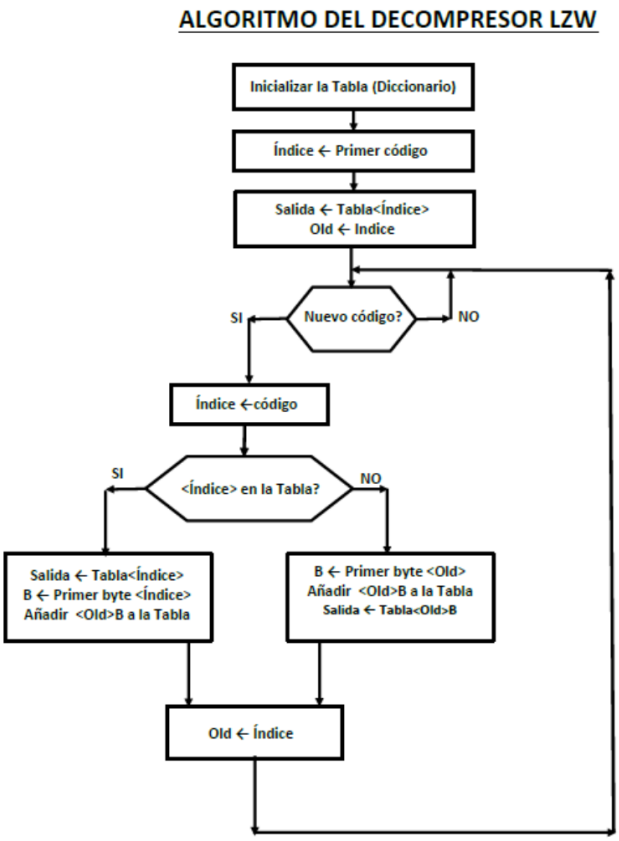

Figura 2. Diagrama de flujo del descompresor LZW.

3. Resultados

3.1. Implementación del compresor LZW

La Figura 3 muestra el diagrama de bloques del core implementado que ejecuta la compresión siguiendo el algoritmo LZW, El diccionario ha sido implementado utilizando una memoria RAM de 4K x 32. El byte más significativo no ha sido utilizado por el momento. El proceso de búsqueda en el diccionario es lineal a partir de la dirección cero la primera vez y posteriormente a partir de la dirección 256. En el futuro se hará una optimización del core y se utilizarán otros métodos para agilizar la búsqueda. La máquina de estados FSM se encarga de realizar todo el proceso, a partir de la señal de entrada New_Data y del resultado del comparador que decide si la cadena <prefijo>B ya está en el diccionario. Al finalizar el proceso genera la señal EOP y si hay un nuevo código, activa Data_Ready. El código de salida tiene 12 bits. Por el momento, la política que se utiliza cuando la RAM se llena es no seguir almacenando datos en ella. Otras políticas de remplazo serán investigadas e implementadas en el futuro. El core del compresor ha sido probado exitosamente a una frecuencia de reloj de 100 MHz, integrándolo mediante un puerto de entrada y otro de salida a un sistema configurado por un procesador ARM Cortex 9 con bus AXI, todo ello incluido en el chip Zynq.7000, que trae la placa de desarrollo ZedBoard de Avnet. El sistema fue adicionalmente configurado para comunicarse serie con una PC, utilizando la hiperterminal.

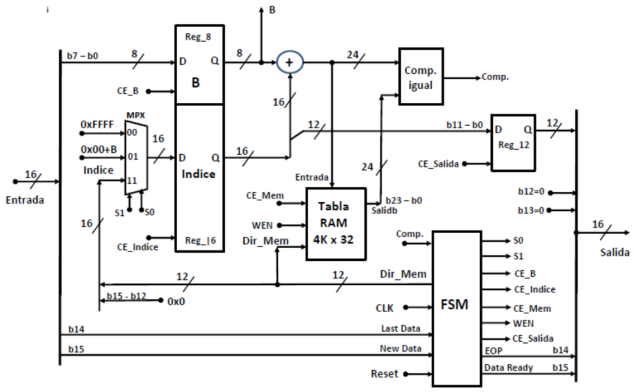

Figura 3. Diagrama de bloques del core para el compresor LZW.

La Tabla 1 muestra la secuencia de datos que se le enviaron, para comprobar el correcto funcionamiento.

3.2. Implementación del descompresor LZW

La Figura 4 es un diagrama de bloques del core de descompresión. Fue probado a una frecuencia de reloj de 100 MHz, utilizando la secuencia de códigos de entrada que muestra la Tabla 2, obteniéndose el resultado esperado.

Posteriormente se procedió a probar ambos cores a la vez conectándolos en serie.

Tabla 1. Secuencia de datos de prueba y códigos de salida del compresor.

Item	Entrada	Índice	B	Salida codif.
0	a	97	---	---
1	b	98	b	97
2	a	97	a	98
3	b	256	b	---
4	c	99	c	256
5	b	98	b	99
6	a	257	a	---
7	b	98	b	257
8	a	257	a	---
9	b	260	b	---
10	a	97	a	260
11	a	97	a	97
12	a	262	a	---
13	a	97	a	262
14	a	262	a	---
15	a	263	a	---
16	a	97	a	263
17	a	262	a	---
18	a	263	a	---
19	a	264	a	---
20	a	97	a	264

Tabla 2. Reconstrucción por el descompresor de la data recibida.

Entrada codif.	old	índice	B	Salida
97	97	--	--	97=a
98	98	97	98	98=b
256	256	98	97	256 =ab
99	99	256	99	99=c
257	257	99	98	257=ba
260	260	257	98	257.98=bab
97	97	260	97	97=a
262	262	97	97	97,97=aa
263	263	262	97	262,97=aaa
264	264	263	97	263,97=aaaa
265	265	264	97	264,97=aaaaa

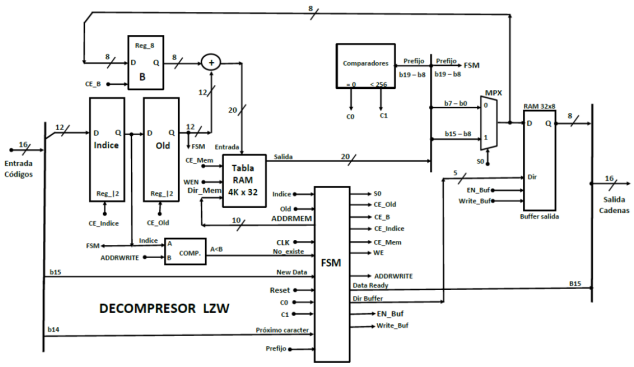

Figura 4. Diagrama de bloques del core para el descompresor LZW.

Fue necesario hacer ligeras modificaciones en el hardware de ambos cores, para garantizar el hands-hake con solo 1 puerto de entrada-salida de 16 bits del ARM Cortex 9.

Se aplicó el código de entrada de la Tabla 1 y a la salida del descompresor se obtuvo la cadena de salida que muestra la Tabla 2.

4. Conclusiones

En este trabajo se presenta una implementación de un algoritmo LZW y se resaltan las características principales del diseño propuesto el cual presenta un excelente desempeño con la ventaja de la tecnología utilizada. Los detalles de arquitectura presentados en este artículo muestran que el diseño de optimización en términos de área y velocidad juegan un papel importante para la simplicidad y eficiencia en alta compresión del algoritmo LZW. La implementación en FPGA del algoritmo discutido en este trabajo ofrece una posibilidad de explorar técnicas de optimización en la parte de búsqueda que realiza el algoritmo tal que se supere en desempeño a otras arquitecturas comerciales disponibles.

Referencias

[1] P. M. Nishad and R. M. Chezian, "Optimization of LZW Algorithm to Reduce Time Complexity for Dictionary Creation in Encoding and Decoding". Asian J. of Computer Science and Information Technology, pp. 114–118, 2012.

[2] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression". *IEEE Trans. Information Theory*, pp. 337–343, May 1977.

[3] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding". *IEEE Trans. Information Theory*, pp. 5306, Sept 1978.

[4] T. A. Welch, "A Technique for High-Performance Data Compression" *IEEE J. Selected Areas in Comput.*, pp. 8–19, June 1961.