Diseño HLS en FPGA de algoritmos de ordenamiento para receptor OFDM con detector de simbolos Near-ML

Aarón Escoboza Villegas

Instituto Tecnológico de Sonora

Cd. Obregón, México

aaron.villegas@hotmail.com

Eduardo Romero Aguirre

Instituto Tecnológico de Sonora

Cd. Obregón, México

Eduardo.romero@itson.edu.mx

*Resumen*—En un sistema receptor OFDM, la detección de símbolos mediante el algoritmo Near-ML, la etapa de ordenamiento de datos es parte esencial en el desempeño y por ende del sistema receptor. Por tal motivo, la motivación detrás de este trabajo es evaluar la implementación en FPGA de diversos algoritmos de ordenamiento y de evaluar su desempeño en términos de métricas tales como: Tiempo de ejecución, consumo de recursos de hardware y frecuencia de operación, para la cantidad de datos requerida por el detector. Todo esto para determinar su factibilidad de implementación en la arquitectura digital de un sistema receptor OFDM con algoritmo de detección Near-ML.

Palabras clave—Diseño HLS, FPGA, Algoritmos de ordenamiento, OFDM.

# Introducción

Hoy en día, gran parte de las aplicaciones de cómputo científico y/o de ejecución de algoritmos de procesamiento digital de señales (PDS), requieren que los datos de entrada se alimenten den un orden especifico [1].Este procedimiento denominado ordenamiento (“sorting”), se ha convertido en una de las operaciones fundamentales en el área de computación [2],[3]. Dicho proceso puede modelarse con el siguiente enunciado, dado una secuencia de números el problema de ordenamiento se define como la permutación tal que [4]. En gran parte de los procesos de la vida real y en una variedad de aplicaciones tales como: procesamiento digital de señales e imágenes, compresión de datos, sistemas de adquisición de datos [5], [6],[7].

Por otro lado, en las últimas décadas la industria semiconductora ha crecido de manera exponencial a pesar de su complejidad y el rendimiento del hardware. FPGAs ha sido la tecnología con mayor desarrollo en comparación con el resto de industria [8]. Dado el vasto campo de aplicación de algoritmos PDS que requieren etapas de ordenamiento y el crecimiento de los FPGAs como tecnología de desarrollo digital el trabajo en conjunto de estas dos se ha convertido en un tema interesante de investigación.

Sin embargo, aún existen retos, son pocos los arquitectos en diseño digital por medio de lenguajes de descripción de hardware como VHDL y/o Verilog. Es preponderante que los no-expertos deben alcanzar un alto nivel de especialización para el desarrollo de modelos descritos a nivel comportamental y trasladarlos a su contraparte a nivel transferencia de registros (RTL), para asegurar su correcta síntesis. Además, VHDL y Verilog no son tan poderosos como los lenguajes de alto nivel lo cual conlleva a códigos significativamente largos incrementando la probabilidad de errores de codificación y tiempo en realizar modificaciones una vez el diseño está hecho.

La síntesis de alto nivel (HLS) puede ayudar a superar los retos anteriormente mencionados [9]. HLS ha estado creciendo rápidamente en los últimos 30 años. Esta tendencia es debido principalmente por la necesidad de una plataforma rápida y confiable que, empezando de un modelo en alto nivel descrito en C o C++ genere su contraparte de bajo nivel, que luego se pueda implementar en diferentes tecnologías de carácter programable, como lo son los FPGAs [10]. Entonces, las herramientas HLS han experimentado un aumento de popularidad debido a que se especializan en generar modelos RTL de alta calidad de producción basados en modelos de alto nivel. Uno de los HLS más populares es Vivado HLS de la compañía Xilinx Inc [11].

El ordenamiento es crucial para el rendimiento general de algunos detectores, uno de ellos es el algoritmo de baja complejidad cerca de la máxima verosimilitud (Near-ML) cuya eficiencia depende completamente del ordenar de 48 y 64 datos [12].

Este trabajo se organiza de la siguiente manera: sección II presenta los algoritmos más conocidos para el ordenamiento de datos; en la sección III , simulación en Matlab e implementación en Vivado son mostrados; finalmente, en la sección IV se realizan conclusiones del trabajo y discusiones sobre futuras investigaciones en el área de estudio.

# Algoritmos de ordenamiento

En esta sección se presentarán los algoritmos mas conocidos para el ordenamiento de datos y su respectivo seudocódigo.

## Algoritmo de inserción

Para lograr el ordenamiento el algoritmo de inserción mantiene dos sub arreglos en el arreglo original: un arreglo desordenado y otro desordenado. El algoritmo funciona de la siguiente manera: Se toma el primer elemento del arreglo desordenado y se compara con cada uno de los elementos del arreglo desordenado, si el elemento es mayor entonces se realiza un intercambio de posiciones. De otra manera se mantiene en su lugar. Este procedimiento se repite veces ,donde es la longitud del arreglo a ordenar [13].Tal comportamiento queda reflejado en el pseudocódigo de la figura 1.

**INSERTION-SORT (A)**

1 **for** j = 1 **to** n – 1

2 min = j

3 **for** i = j + 1 **to** n

4 if A[i] < A[min]

5 min = i

6 swap A[j], A[min])

Figura 1. Algoritmo Inseriton-Sort

## Algoritmo de selección

El algoritmo de selección funciona de la siguiente manera: el primer elemento del arreglo se selecciona para compararse con todos los elementos restantes, si el elemento seleccionado es más grande que el elemento más pequeño, se intercambia la posición de los dos elementos, por lo tanto, en la primera iteración el elemento más pequeño se coloca en la primera posición del arreglo. Después, se repite el mismo procedimiento con los n – 1 elementos restantes hasta conseguir un arreglo ordenado [14]. El algoritmo de selección se muestra como pseudocodigo en la figura 2.

**SELECTION-SORT (A)**

1 **for** j = 2 to A.length

2 key = A[J]

3 i = j - 1

4 **while** i > 0 **and** A[i] > key

5 A[i + 1] = A[i]

6 i = i - 1

7 A[i + 1] = key

Figura 2. Algoritmo Selection-Sort

## Algoritmo de ordenamiento por montículos

El ordenamiento por montículos es un árbol binario completo que satisface una condición, la condición de montículo. Existen dos tipos de montículos: mínimo y máximo. El montículo mínimo contiene el valor más pequeño en el nodo raíz y el montículo máximo contiene el valor más grande [15].

El algoritmo heapsort mostrando en la figura 3 funciona de la siguiente manera. Primero, se empieza construyendo un montículo máximo con la entrada de un arreglo . Como es el elemento de valor más grande se cambia a su posición final y se descarta del montículo decrementando la variable A.heap-size que representa el tamaño del montículo. La nueva raíz puede no cumplir con la condición del montículo, por tal razón se llama a la función MAX-HEAPIFY(A,1), la cual construye un montículo máximo A[1]. El procedimiento se repite hasta lograr un montículo de tamaño 2 [4]. B,l

**HEAPSORT** **(A)**

1 BUILD-MAX-HEAP (A)

2 **for** i = A.length **downto** 2

3 exchange A[1] with A[i]

4 A.heapSize = A.heapSize - 1

5 MAX-HEAPIFY (A,1)

Figura 3. Algoritmo Heapsort

**BUILD-MAX-HEAP (A)**

1 A.heapSize = A.length

2 **for** i = A.length/2 **downto** 1

3 MAX-HEAPIFY(A,i)

Figura 4. Subfunción BUILD-MAX-HEAP

**MAX-HEAPIFY(A,i)**

1 l = 2i

2 r = 2i + 1

3 **if** l ≤ A.heapSize **and** A[l] > A[i]

4 largest = l

5 **else** largest = i

6 **if** r ≤ A.heapSize **and** A[r] > A[largest]

7 largest = r

8 **if** largest i

9 exchange A[i] with A[largest]

10 **MAX-HEAPIFY**(A,largest)

Figura 5. Subfunción MAX-HEAPIFY

## Algoritmo de ordenamiento por mezcla

El algoritmo por mezcla fue inventado por John Von Neumann, el cual está basado en el enfoque dividir y conquistar. La idea principal es dividir una secuencia de elementos , en dos secuencias individuales y . Después, cada conjunto de datos es ordenado utilizando recursividad. Finalmente, las dos secuencias ordenada son mezcladas para obtener una sola secuencia de elementos ordenados [16].

**MERGE(A,p,q,r)**

1 = q - p + 1

2 = r - q

3 **for** i = 1 **to**

4 L[i] = A[p + i - 1]

5 **for** j = 1 **to**

6 R[j] = A[q + j]

7 L[ + 1] = infinito

8 R[ + 1] = infinito

9 i = 1

10 j = 1

11 **for** k = p **to** r

12 **if** L[i] ≤ R[j]

13 A[k] = L[i]

14 i = i + 1

15 **else** A[k] = R[j]

16 j = j + 1

Figura 6. Algoritmo MERGE

**MERGE-SORT(A,p,r)**

1 **if** p < r

2 q = **lowest**((p + r)/2)

3 MERGE-SORT(A,p,q)

4 MERGE-SORT(A,q + 1,r)

5 MERGE(A,p,q,r)

Figura 7. Subfunción MERGE-SORT

# Pruebas y resultados

Con el objetivo de identificar el algoritmo con menor tiempo de ejecución, se procedió a modelar cada uno de ellos en MATLAB. Para esto se definieron las siguientes consideraciones.

* Se realizaron 100 pruebas, cada una genera un arreglo de números aleatorios que se desea ordenar
* La longitud de cada arreglo es de 48 elementos, correspondientes al numero de subportadoras en un símbolo OFDM.
* El tiempo se midió con las funciones tic y toc de MATLAB.

En la figura 1 se resume el resultado del experimento anterior, como se puede notar los dos mejores algoritmos para el ordenamiento del arreglo de 48 datos son el de selección y el de inserción, por tal razón son elegidos para ser sintetizados en Vivado. En la figura 2 se muestra los recursos consumidos.

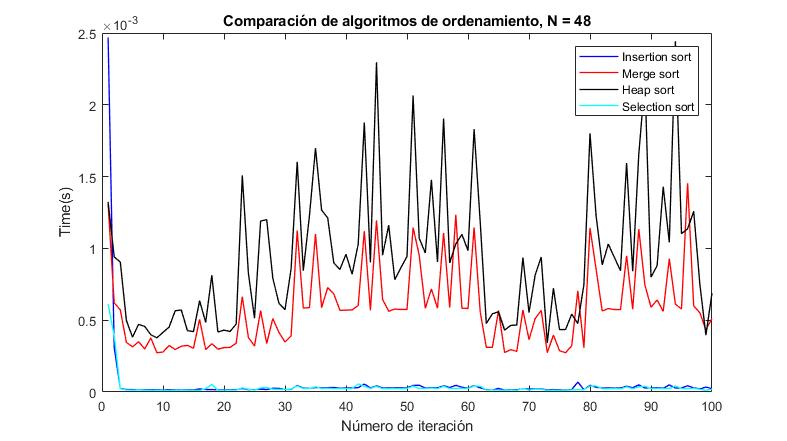


Figura 1. Comparación de algoritmos de ordenamiento

Para el caso de obtener el rendimiento en términos de frecuencia de operación y consumo de hardware. Se establecieron las siguientes condiciones para ambos algoritmos.

* Se modelaron en lenguaje C
* Se eligió la tarjeta de desarrollo Nexys 4
* Se usó la herramienta Vivado-HLS con las opciones de optimización desactivadas.

\*Poner figuras con ambas síntesis y frecuencia de operación

\*Poner discusión de los resultados

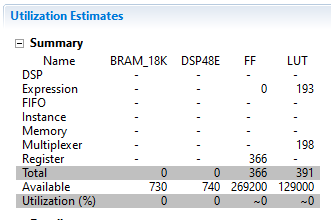


Figura 2. Resultado de síntesis del algoritmo de selección

# Conclusiones

El ordenamiento es una importante operación para una gran cantidad de aplicaciones y puede ser la parte crucial para el rendimiento general de un sistema. En este artículo se evaluó tiempo de ejecución de los algoritmos de ordenamiento más populares hasta el momento. Se encontró que el algoritmo de selección es el mejor para ser incorporado al detector Near-ML y se obtuvo los recursos consumidos de hardware para ser implementado en un FPGA.

##### Referencias

[1] “Lipu, A. R., Amin, R., Mondal, M. N. I.,& Al Mamun, M. (2016, December). Exploiting parallelism for faster implementat ion of Bubble sort algorithm using FPGA. In 2016 2nd Internat ional Conference on Elect rical, Computer & Telecommunication Engineering .”

[2] “D.E Knuth. The Art of computer programming, Sort ing and Searching volume II, Addison-Wesley, 2011.”

[3] “G. Goetz, ‘Implementing sorting in database systems,’ ACM Comput. Surv., vol. 38, pp. 10, 2006.”

[4] S. Radhakrishnan, D. Kolippakkam, and V. S. Mathura, *Introduction to algorithms*. 2007.

[5] “S. Lukáš, ‘Evolutionary Design Space Exploration for Median Circuits’, Lecture Notes in Computer Science, Vol. 2004, No. 3005, DE, pp. 240-249, 2004.”

[6] “E. Jamro, M. Wielgosz, and K. Wiatr, “FPGA Implementation of the Dynamic Huffman Encoder," Proc. Workshop of Programmable Devices and Embedded Systems, pages 60-65, February 2006.”

[7] “C. C. W. Robson and C. Bohm, ‘A high speed data acquisition collector for merging and sorting data,’ Nuclear Science Symposium Conference Record, 2008. NSS ’08, 2008.”

[8] “D. Chen, J. Cong, and P. Pan, ‘FPGA Design Automation: A Survey,’ Foundations and Trends in Electronic Design Automation, vol. 1, no. 3, pp. 139–169, 2006.” .

[9] “D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, Algorithmic and Register-Transfer Level Synthesis: The System Architects Workbench. The Kluwer International Series in Engineering and Computer Science 85, Springer.”

[10] “O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, ‘An empirical evaluation of high-level synthesis languages and tools for database acceleration,’ in 24th International Conference on.”

[11] “Xilinx Inc., Vivado Design Suite User Guide v2015.1, 2015.”

[12] Hector Eduardo Aldrete Vidrio, “Sistema de comunicación multiportadora para el estándar 802.11p utilizando precodificación frecuencial y cancelación no lineal de interferencia,” 2019.

[13] K. Nenwani, V. Mane, and S. Bharne, “Enhancing adaptability of Insertion sort through 2-Way expansion,” *Proc. 5th Int. Conf. Conflu. 2014 Next Gener. Inf. Technol. Summit*, pp. 843–847, 2014, doi: 10.1109/CONFLUENCE.2014.6949294.

[14] “Jadoon, S., Solehria, S. F., Rehman, S., & Jan, H. (2011). Design and analysis of optimized selection sort algorithm. International Journal of Electric & Computer Sciences (IJECS-IJENS), pp. 16–22.”

[15] M. Shaikh and R. Vadivel, “A Comparative Study of Well Known Sorting Algorithms,” *Int. J. Adv. Res. Comput. Sci.*, vol. 9, no. 2, pp. 742–744, 2018.

[16] “Jmaa, Y. B., Ali, K., Duvivier, D., Jemaa, M. B., & Atitallah, R. B. (2017). An efficient hardware implementation of timsort and mergesort algorithms using high level synthesis. IEEE International Conference on High Performance Computing & Simulation (HPC.”