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 de Sonora

aaron.villegas@hotmail.com

Eduardo Romero Aguirre  
Instituto Tecnológico de Sonora

Cd. Obregón de Sonora

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 . 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].

OFDM (Multiplexación por División de Frecuencia Ortogonal) es una de las técnicas propuestas para ser usada en los siguientes sistemas de comunicación inalámbricas [12]. La idea básica de OFDM es dividir un flujo de datos de alta velocidad en flujos de tasa más baja para después ser transmitidos sobre subportadoras [13]. Algunos de los beneficios son: alta velocidad de datos, alta eficiencia espectral, alta calidad de servicio y robustez contra la interferencia en banda estrecha y desvanecimiento selectivo en frecuencia [14].

La factibilidad de un sistema de comunicación depende gran parte de la complejidad de la etapa de detección de datos. El detector óptimo de máxima verosimilitud (Maximum Likihood, ML) presenta una complejidad de , se observa que la complejidad del detector optimo aumenta con el tamaño de la constelación y la cantidad de subportadoras , por lo tanto, su implantación no es viable en comparación con los detectores lineales cuya complejidad es acotada por [15].

En la literatura se encuentran algoritmos con desempeños en términos de tasa de error de bit semejantes al ML, uno de los más relevantes es el detector esférico [16]. Existen esquemas de detección no lineales que son basados en el algoritmo M y la descomposición QR de la matriz de canal [17],[18].Uno de ellos el reciente detector Near-ML, presenta baja complejidad, mejor desempeño de BER(Bit Error Rate) y se acerca al desempeño del detector optimo ML en comparación con los sistemas OFDM convencionales [15]. La etapa fundamental del detector es el ordenamiento de una cantidad n de distancias de acuerdo con el tamaño de la constelación, por ende, el algoritmo de ordenamiento y su correspondiente implementación en hardware determina en gran medida el rendimiento y consumo de hardware del detector y del receptor.

Este trabajo se organiza de la siguiente manera: sección II presenta los algoritmos más conocidos para el ordenamiento de datos; en la sección III las pruebas y resultados 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 más 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 [19].Tal comportamiento queda reflejado en el pseudocódigo de la Figura 1.

**INSERTION-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 1. Algoritmo Insertion-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 [20]. El algoritmo de selección se muestra como seudocódigo en la Figura 2

**SELECTION-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 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 [21].

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].

**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.

## Ordenamiento de burbuja

El algoritmo funciona analizando un arreglo de elementos de izquierda a derecha. Se compara cada par de elementos adyacentes y se intercambian si el elemento de la izquierda es mayor que el de la derecha com. El algoritmo realiza este proceso continuamente has que pasa por el arreglo de elementos sin realizar ninguna operación de intercambio, esto significa que todos los elementos del arreglo han sido ordenados [22].

**BUBBLE-SORT(A)**

1 **for** i = A.length - 1 **to** 0

2 **for** j = 1 **to** i

3 if (A[j - 1] > A[j])

4 swap (A[j - 1], A[j])

Figura 6. Algoritmo Bubble Sort.

# Pruebas y resultados

Con el objetivo de obtener el rendimiento de los distintos algoritmos de ordenamiento en términos de tiempo y recursos de hardware consumidos del FPGA, se establecieron las siguientes condiciones.

* Se modelaron en lenguaje C.
* Se eligió la tarjeta de desarrollo Nexys 4 Artix-7 con el chip xc7a100tcsg324-1.
* Se usó la herramienta Vivado-HLS versión 2016.1 con las opciones de optimización desactivadas.
* Se sintetizó para 4, 16, 64, y 256 elementos de acuerdo con la cantidad de distancias que se producen en las constelaciones QPSK, QAM 16, QAM 64 y QAM 256.
* Se estableció la longitud de 16 bits para cada elemento a ordenar con base al análisis de punto fijo .
* Se utilizó el periodo de reloj por detecto de 10ns y tolerancia de 1.25ns.

En la Tabla 1 se muestran los resultados de velocidad de cada uno de los algoritmos de ordenamiento para las distintas cantidades de distancia que se producen en la modulación QAM. Tomando como referencia el algoritmo de inserción con N = 4 donde N es la cantidad de elementos a ordenar se obtiene una estimación del periodo de reloj de 5.16. La tolerancia por defecto de 1.25ns permite asegurar la existencia de un margen de tiempo para retrasos desconocidos que se puedan presentan durante el proceso de enrutamiento. Por ejemplo, si se resta el periodo de reloj por defecto de 10ns menos la tolerancia de 1.25ns se obtiene 8.75ns, este resultado es mayor que la estimación de 5.16, lo cual cumple con los requerimientos de tiempo.

Tabla 1. Desempeño de velocidad

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | N | Periodo  del reloj  (ns) | Estimado  (ns) | Toleracia  (ns) |
| Bubble  sort | 4 | 10 | 5.16 | 1.25 |
| 16 | 10 | 5.16 | 1.25 |
| 64 | 10 | 5.48 | 1.25 |
| 256 | 10 | 5.48 | 1.25 |
| Selection sort | 4 | 10 | 6.53 | 1.25 |
| 16 | 10 | 6.53 | 1.25 |
| 64 | 10 | 6.85 | 1.25 |
| 256 | 10 | 6.85 | 1.25 |
| Insertion sort | 4 | 10 | 5.16 | 1.25 |
| 16 | 10 | 5.16 | 1.25 |
| 64 | 10 | 5.48 | 1.25 |
| 256 | 10 | 5.48 | 1.25 |
| Heap  sort | 4 | 10 | 6.53 | 1.25 |
| 16 | 10 | 6.53 | 1.25 |
| 64 | 10 | 6.85 | 1.25 |
| 256 | 10 | 6.85 | 1.25 |

En la Tabla 2 se presenta el desempeño en términos de latencia para cada conjunto de elementos a ordenar. La cantidad máxima de latencia en cada uno de los algoritmos representa los ciclos necesarios para lograr ordenar los N elementos y el intervalo se refiere a la cantidad de ciclos de reloj que deben de transcurrir para empezar a ordenar un nuevo bloque de N elementos desordenados. Por ejemplo, el algoritmo de selección le toma 34 ciclos para logar ordenar la cantidad de 4 elementos, en el ciclo 35 es capaz de volver a empezar con la etapa de ordenación. Esto significa que el comportamiento del bloque es completamente secuencial y técnicas como la de “pipline” no son implementadas.

Tabla 2.Desempeño de latencia

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmo | N | Latencia | Intervalo |
| Bubble  sort | 4 | 34 | 35 |
| 16 | 514 | 515 |
| 64 | 8194 | 8195 |
| 256 | 131074 | 131075 |
| Selection sort | 4 | 35 | 36 |
| 16 | 527 | 528 |
| 64 | 8255 | 8256 |
| 256 | 131327 | 131328 |
| Insertion sort | 4 | 26 | 27 |
| 16 | 482 | 483 |
| 64 | 8066 | 8067 |
| 256 | 130562 | 130563 |
| Heap sort | 4 | 201 | 202 |
| 16 | 2811 | 2812 |
| 64 | 43491 | 43492 |
| 256 | 690051 | 690052 |

En la Tabla 3 se muestran los resultados del consumo de hardware de cada algoritmo de ordenamiento para los distintos casos de elementos a ordenar. Se observa que todos consumen FF (Flip Flops) y LUTs (Lookup Tables) para su implementación en el FPGA y ninguno de ellos requiere bloques de RAM o DSPs. Estos últimos no son consumidos debido a que los elementos a ordenar son almacenados en una memoria externa y porque todos algoritmos de ordenamiento presentados en este trabajo se basan en comparaciones y asignaciones, por lo cual las operaciones aritméticas no son necesarias.

Tabla 3. Recursos consumidos del FPGA

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algoritmo | N | BRAM | DSP | FF | LUT |
| Bubble  sort | 4 | 0 | 0 | 33 | 39 |
| 16 | 0 | 0 | 43 | 51 |
| 64 | 0 | 0 | 53 | 64 |
| 256 | 0 | 0 | 63 | 76 |
| Selection sort | 4 | 0 | 0 | 115 | 151 |
| 16 | 0 | 0 | 125 | 158 |
| 64 | 0 | 0 | 135 | 165 |
| 256 | 0 | 0 | 145 | 171 |
| Insertion sort | 4 | 0 | 0 | 33 | 41 |
| 16 | 0 | 0 | 45 | 54 |
| 64 | 0 | 0 | 55 | 66 |
| 256 | 0 | 0 | 65 | 79 |
| Heap  sort | 4 | 0 | 0 | 193 | 247 |
| 16 | 0 | 0 | 215 | 272 |
| 64 | 0 | 0 | 237 | 296 |
| 256 | 0 | 0 | 259 | 321 |

Con base a los resultados de las tablas 1, 2 y 3 se determinó que el mejor algoritmo en términos de velocidad, latencia y consumo de hardware es el algoritmo de ordenamiento por inserción. En la Figura 7 se muestra los puertos de entrada y salida del bloque de hardware generado por Vivado HLS. Se compone de las señales con prefijo “ap” las cuales son generadas de manera automática para cualquier bloque digital y una interfaz para acceder a una memoria RAM (Random Access Memory) de doble puerto donde están almacenados los elementos a ordenar. Cuando la señal “ap\_start” se vuelve un uno lógico se da inicio con el ordenamiento de datos y termina cuando la bandera “ap\_done” se vuelve verdadero indicando que el proceso de ordenamiento ha finalizado. Para leer los datos ya ordenados se debe de colocar la dirección que se desea leer en la señal de entrada “indexOutputData” para que posteriormente el dato de esa dirección se vea reflejado en la señal de salida “ap\_return”. El diagrama RTL (Register Transfer Level) que se genera con la herramienta de Vivado se muestra en la Figura 8.

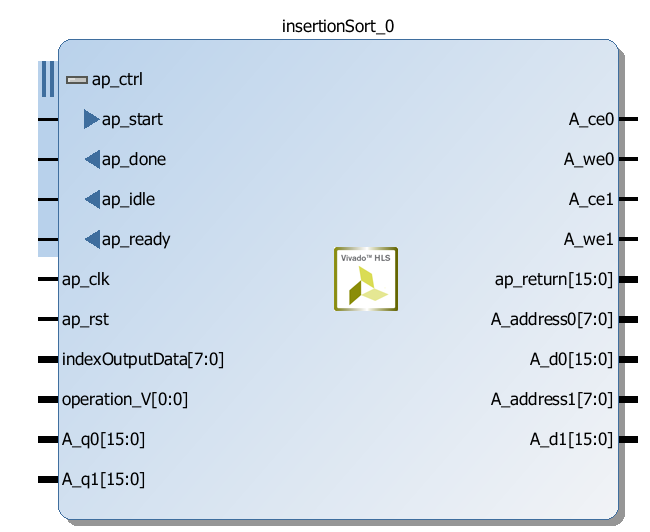


Figura 7. Bloque del algoritmo de inserción.

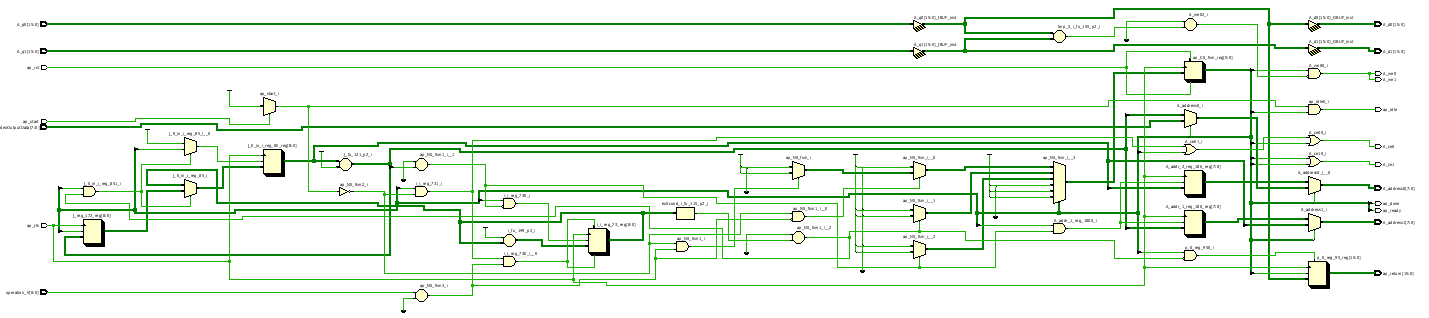


Figura 8. RTL del algoritmo de inserción.

En la Figura 9 se presenta el diagrama de tiempos del algoritmo de inserción. La ejecución empieza cuando la señal “ap\_start” se vuelve un uno lógico, en ese momento el bloque de hardware inicia con el proceso de ordenamiento generando direcciones para leer o escribir en la memoria externa. Después de una cantidad de ciclos de reloj la bandera “ap\_done” indica que los elementos están completamente ordenados y es posible seguir con la etapa de lectura como se muestra en la Figura 10.

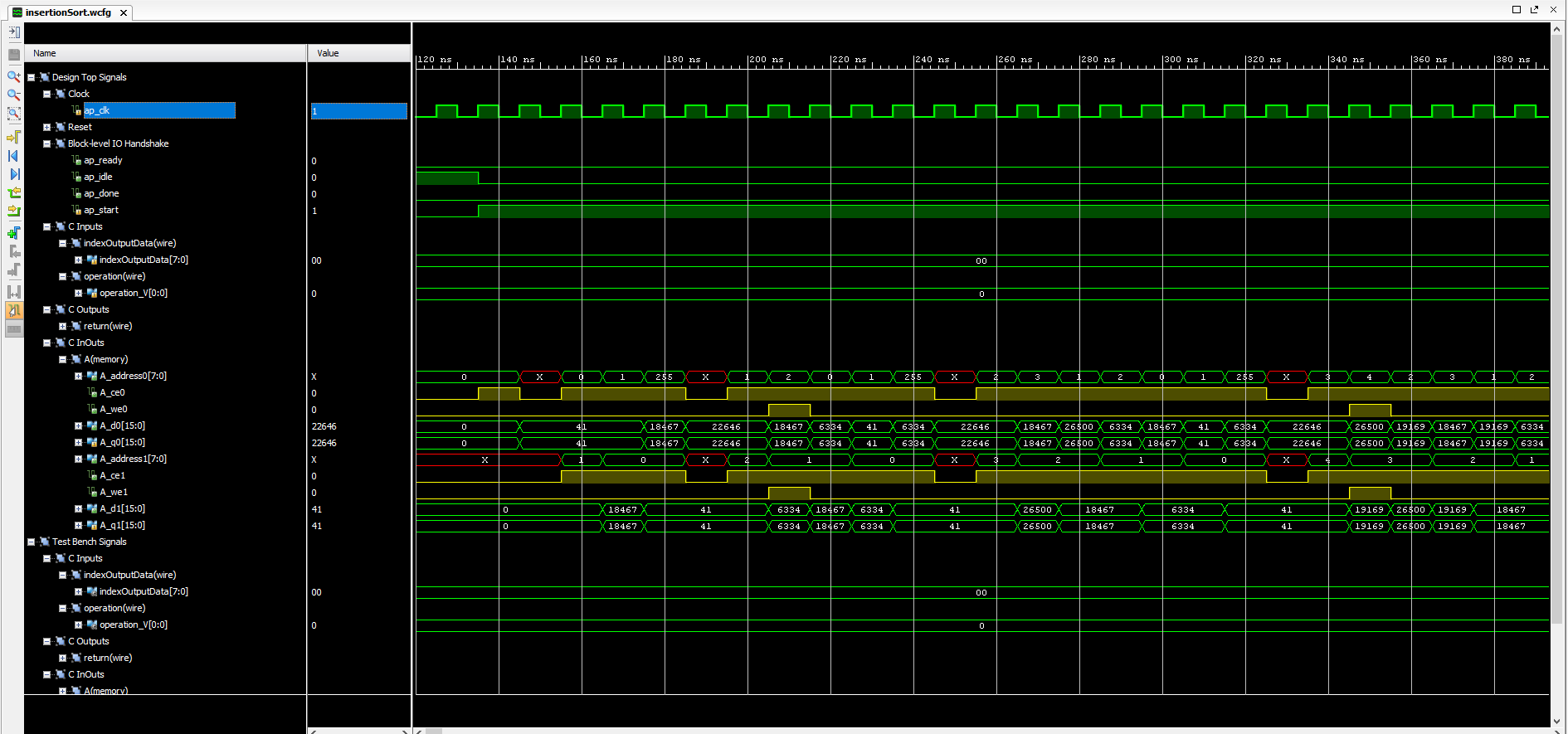


Figura 9. Inicio del proceso de ordenamiento.

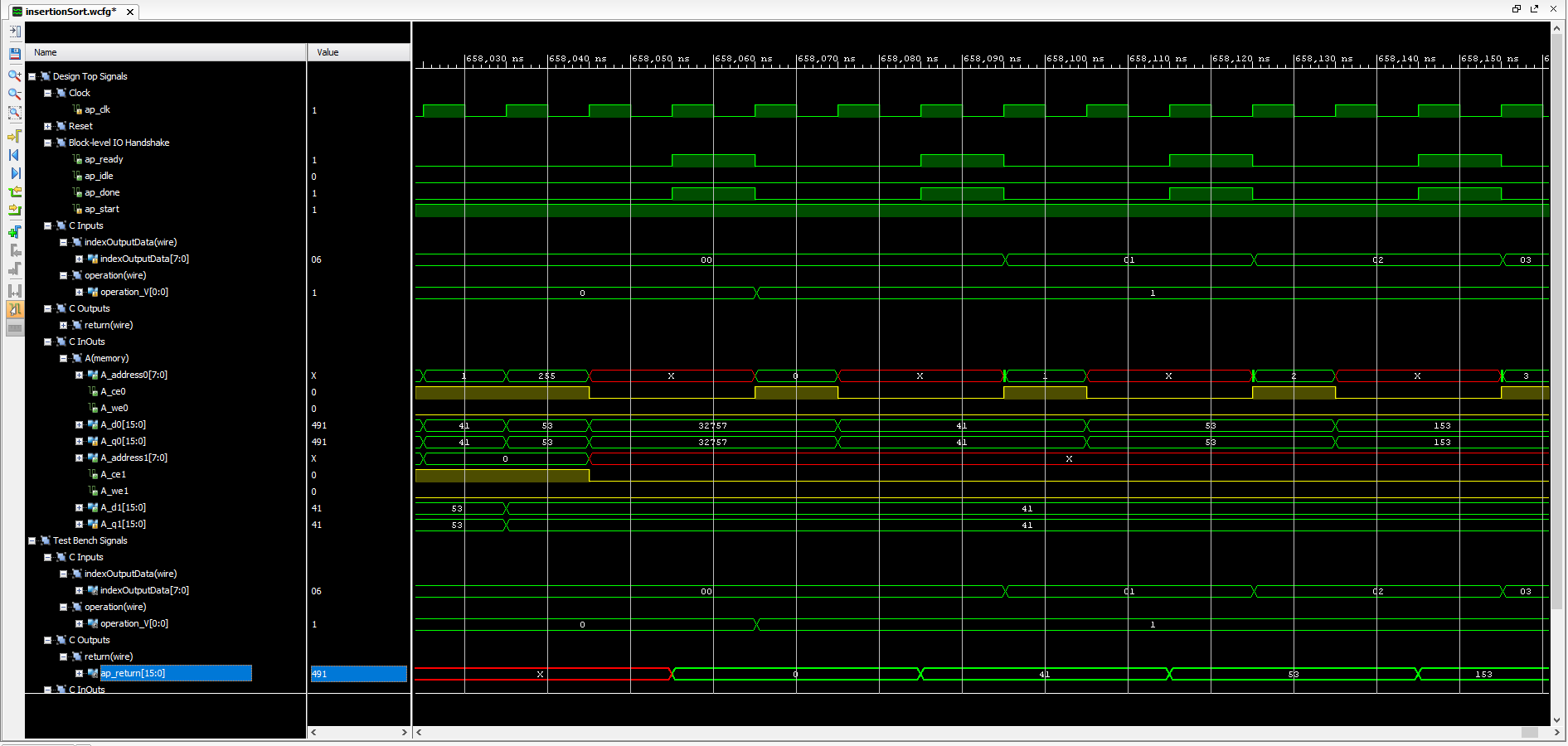


Figura 10. Finalización del proceso de ordenamiento.

# 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 realizó la síntesis de cuatro algoritmos de ordenamiento utilizando Vivado HLS y del mejor se obtuvo su correspondiente RTL y diagrama de tiempos. Se encontró que el algoritmo de inserción presenta mejores resultados en términos de velocidad, latencia y recursos de hardware de un FPGA. Desde un punto de vista de implementación el algoritmo de inserción es el mejor candidato para ser incorporado al detector Near-ML cuya parte fundamental es el ordenamiento N distancias de acuerdo con la constelación a utilizar. Como futuro trabajo se pueden aplicar técnicas de optimización con el fin de reducir complejidad, recursos consumidos y latencia del algoritmo de ordenamiento.

##### Referencias

[1] A. R. Lipu, R. Amin, M. N. I. Mondal, and M. Al Mamun, “Exploiting parallelism for faster implementation of Bubble sort algorithm using FPGA,” ICECTE 2016 - 2nd Int. Conf. Electr. Comput. Telecommun. Eng., no. December, pp. 8–10, 2017, doi: 10.1109/ICECTE.2016.7879576.

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

[3] G. Graefe, “Implementing sorting in database systems,” ACM Comput. Surv., vol. 38, no. 3, p. 4, 2006, doi: 10.1145/1132960.1132964.

[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] M. M. Kamruzzaman, “Performance of Turbo coded wireless link for SISO ­ o MRMr,” no. Iccit, pp. 22–24, 2011.

[13] K. R. Rao, Z. S. Bojkovic, and D. A. Milovanovic, “OFDM for Wireless Multimedia Communications,” Wirel. Multimed. Commun., 2018, doi: 10.1201/9781420008227.

[14] I. Baig and V. Jeoti, “A new ZCT precoded OFDM system with pulse shaping: PAPR analysis,” IEEE Asia-Pacific Conf. Circuits Syst. Proceedings, APCCAS, pp. 1131–1134, 2010, doi: 10.1109/APCCAS.2010.5775063.

[15] J. Alberto, D. Aldrete, P. Flores, and D. Tesis, “Sistema de comunicación multiportadora para el estándar Metodologías TCAD para diseñar diodos precodificación frecuencial epitaxiales de recuperación rápida y de silicio cancelación no lineal de interferencia usando una estructura con contacto tipo mosaic,” 2019.

[16] M. O. Damen, H. El Gamal, and G. Caire, “On maximum-likelihood detection and the search for the closest lattice point,” IEEE Trans. Inf. Theory, vol. 49, no. 10, pp. 2389–2402, 2003, doi: 10.1109/TIT.2003.817444.

[17] H. Moroga, T. Yamamoto, and F. Adachi, “Overlap QRM-ML Block Signal Detection for Single-Carrier Transmission without CP Insertion,” Wirel. Pers. Commun., vol. 74, no. 4, pp. 1163–1177, 2014, doi: 10.1007/s11277-013-1570-5.

[18] K. Temma, T. Yamamoto, and F. Adachi, “Improved 2-step QRM-ML block signal detection for single-carrier transmission,” IEEE Veh. Technol. Conf., no. 0, 2011, doi: 10.1109/VETECF.2011.6093118.

[19] 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.

[20] “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.”

[21] 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.

[22] S. M. Cheema, N. Sarwar, and F. Yousaf, “Contrastive analysis of bubble & merge sort proposing hybrid approach,” 2016 6th Int. Conf. Innov. Comput. Technol. INTECH 2016, pp. 371–375, 2017, doi: 10.1109/INTECH.2016.7845075.