

**CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL IPN,
UNIDAD GUADALAJARA**

**PROGRAMA DE TALENTO ALTAMENTE ESPECIALIZADO EN SISTEMAS
EMBEBIDOS**

METODOLOGÍA DE DISEÑO DE SYSTEM-ON-CHIP

Instructor: Dr. Vidkar Anibal Delgado Gallardo

Elaborado por: José Antonio Rodríguez Velázquez

09/05/2024

INTRODUCCIÓN

El presente documento detalla el proceso llevado a cabo para implementar un coprocesador de convolución unidimensional mediante la metodología de diseño Top-Down para System-on-Chip (SoC).

El proceso de diseño inicia con la obtención de los requerimientos del problema, los cuales sirven de base para describir las características de funcionamiento del sistema a desarrollar. Este proceso se realiza desde niveles de abstracción elevados hasta llegar a la implementación en un lenguaje de descripción de hardware que modele el circuito.

Partiendo de los requerimientos y la definición del problema, se propone un modelo de caja negra que guía el desarrollo de un modelo de oro, el cual ofrece una solución al problema planteado. Posteriormente, se valida este modelo y se utilizan herramientas como los diagramas de máquinas de estado algorítmicas (ASM), bloques lógicos básicos y diagramas de estados finitos.

En este punto, se procede a la implementación en lenguaje de descripción de hardware, teniendo en cuenta que este consta de dos partes principales: el data path y la máquina de estados que lo controla.

Una vez generado el módulo a partir de los análisis descritos anteriormente, el documento presenta los resultados de simulación y síntesis. Esto permite evaluar el funcionamiento y rendimiento (en términos de área y tiempo) del modelo generado.

1. DESCRIPCIÓN DEL PROBLEMA

Implementar un coprocesador de convolución utilizando la metodología de diseño *top-down*.

Características de la implementación

- El sistema cuenta con una entrada correspondiente al valor de la primera señal a la que se le aplicará la operación de convolución (señal *Y*).
- La señal *Y* es almacenada en una memoria, para acceder a un valor discreto específico de la señal es necesario que el sistema genere una señal de salida que corresponda a la dirección de memoria de la señal.
- El tamaño de la señal *Y* también es una entrada al sistema.
- La segunda señal de entrada para aplicar la convolución (señal *H*), junto con su tamaño, estarán almacenados dentro del sistema y estarán previamente definidos a la ejecución.
- Para iniciar el cálculo de la convolución, es necesario que una señal de entrada, denominada *Start*, sea habilitada.
- La señal de salida es almacenada en una memoria, por lo que el circuito debe generar la lógica de escritura de esta.
- El sistema genera una señal de ocupado una vez se haya iniciado el cálculo, y esta es desactivada para indicar que se ha completado el cálculo.

2. DIAGRAMA DE CAJA NEGRA Y DEFINICIÓN DE SEÑALES DE ENTRADA Y SALIDA

Definición de entradas y salidas

Entradas:

- Entradas de control:
 - *start* (1 bit): Especifica el inicio del proceso.
- Entradas de datos:
 - *dataY* (8 bits): Señal de datos de entrada, corresponde al valor de la dirección de memoria seleccionada por la salida *memY_addr*.
 - *size* (5 bits): Especifica el tamaño de la señal de entrada *Y*.

Salidas:

- Salidas de control:
 - *busy* (1 bit): Señal que indica que el proceso está en ejecución, pero que no se ha completado.
 - *done* (1 bit): Señal que indica que el proceso se ha completado.
 - *writeZ* (1 bit): Señal para escribir el valor actual de la salida *dataZ* en la dirección de la memoria *Z*, determinada por la señal *memZ_addr*.

- Salidas de datos:
 - *dataZ* (16 bits): Entrega el dato calculado de la convolución en el índice actual del proceso. Este dato se almacenará en la dirección de memoria indicada por la señal *memZ_addr*.
 - *memZ_addr* (6 bits): Dirección de memoria en donde se almacenará el dato calculado de la convolución actual.

Relación entre las entradas y las salidas

- Cuando la señal *start* tenga un estado lógico 0, las señales *busy* y *done* también tendrán un valor bajo.
- Cuando la señal *start* tenga un estado lógico alto, el proceso deberá de iniciar, y la señal *busy* tendrá un estado alto hasta que se termine el proceso, una vez se haya terminado, está tendrá un estado bajo y la señal *done* pasará a tener uno alto.
- La señal *done* permanecerá en alto exclusivamente en un ciclo de reloj.

Sistema de caja negra

La **Figura-1** muestra el sistema de caja negra propuesto a partir de las señales de entrada y de salida del sistema. La **Figura-1** nos permite visualizar el funcionamiento general del sistema: mediante el uso de una memoria, una señal (*Y*) alimenta el sistema, la cuál será utilizada, junto con una señal (*H*) almacenada en el módulo, para aplicar la operación de convolución, y guardar los resultados en una memoria.



Figura-1. Diagrama de caja negra del coprocesador de convolución.

3. PSEUDOGÓDIGO

Recordando la expresión matemática de la operación de convolución en tiempo discreto

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k]g[n - k]$$

es posible identificar los elementos que la componen, descritos a continuación:

$f[n] \rightarrow$ Señal en tiempo discreto f

$g[n] \rightarrow$ Señal en tiempo discreto g

Existen diferentes métodos analíticos para el cálculo de la convolución, los cuales pueden resultar muy convenientes al momento de analizar la implementación algorítmica de la operación de convolución en tiempo discreto. Suponiendo las señales $f[n] = \{5, 7, 4, 2\}$ y $g[n] = \{3, 2, 1\}$, y recordando que en los instantes de tiempo en los que estas no están definidas de forma explícita, estas cuentan con un valor de cero, es posible utilizar algoritmos como el que se muestra en la **Tabla-1**.

k	-2	-1	0	1	2	3	4	5
$f[k]$			5	7	4	2		
$g[0-k]$	1	2	3	0	0	0	0	0
$g[1-k]$	0	1	2	3	0	0	0	0
$g[2-k]$	0	0	1	2	3	0	0	0
$g[3-k]$	0	0	0	1	2	3	0	0
$g[4-k]$	0	0	0	0	1	2	3	0
$g[5-k]$	0	0	0	0	0	1	2	3

n	$y[n]$
0	15
1	31
2	31
3	21
4	8
5	2

Tabla-1. Método tabular para el cálculo de la convolución de dos señales discretas.

Del método de la tabla anterior, es posible deducir dos cosas importantes:

- Número de muestras de la señal de salida. Para garantizar que el resultado de $\sum_{k=-\infty}^{\infty} f[k]g[n-k]$ pueda ser diferente de cero, es necesario que el desplazamiento de la señal $g[n-k]$ hacia la izquierda sea igual al número de muestras de la señal $g[n-k]$ menos uno ($N_g - 1$).

Por otra parte, desde que el origen de las dos señales se alinea a la izquierda, y durante el desplazamiento de g hacia la derecha hasta que el último elemento de la señal $f[k]$ se superponga con el primer elemento de la señal $g[n-k]$, es posible realizar N_f operaciones.

Combinando los dos razonamientos anteriores, para obtener el número de muestras de la señal de salida, basta con sumar las muestras desde que el último elemento de $g[n-k]$ coincide con el primer elemento de $f[k]$, hasta que el primer elemento de $g[n-k]$ coincida con el último elemento de $f[k]$, o lo que es lo mismo

$$N_{(f*g)} = N_f + N_g - 1$$

- El proceso puede realizarse mediante el uso de ciclos anidados. Si el proceso se analiza bajo el uso de variables de iteración, es posible observar que la primera de ellas corresponde a una variable que recorra todos los índices de la señal de salida, por lo tanto, esta debe ir desde 0 hasta $N_f + N_g - 1$. Esta variable puede tomar el nombre de i .

En contraparte, para poder asegurar de multiplicar todos los elementos, es necesario iterar a través del número de elementos de una de las dos señales con otra variable de nombre j , para cada una de las iteraciones de la variable anterior. Después se calcula

el índice o la muestra de la segunda señal para realizar el producto, y posteriormente se suman los productos resultados de cada iteración.

La **Figura-2** muestra el pseudocódigo propuesto para la implementación de la solución del coprocesador de convolución, a partir del análisis anterior.

```
1. While start = 0
2. End while
3. busy = 1
4. i = 0
5. While i < (sizeY + sizeH - 1)
6.     currentZ = 0
7.     j = 0
8.     While j < sizeY
9.         If ((i-j >= 0) && (i-j < sizeH))
10.            currentZ = currentZ + MemH[i-j] * dataY
11.        j++
12.    End While
13.    writeZ = 1
14.    writeZ = 0
15.    i++
16. End while
17. busy = 0
18. done = 1
19. done = 0
20. If start = 0
21. goto 1
22. Else goto 20
```

Figura-2. Pseudocódigo propuesto para el coprocesador de convolución.

La complejidad temporal del algoritmo, al contar con los dos ciclos anidados es de $O(n^2)$, mientras que la complejidad espacial es de $O(1)$, ya que el espacio de memoria no depende del tamaño de los datos de entrada.

a. VALIDACIÓN

A partir del pseudocódigo anterior, y, mediante el uso del lenguaje de programación C, se validó el algoritmo propuesto. La **Figura-3** muestra el código generado.

```
#include <stdio.h>

int main(){
    int i, j, currentZ;
    // Memoria Y
    int MemY[] = {5, 4, 3, 2, 1};
    int sizeY = sizeof(MemY) / sizeof(int);
    // Memoria H
    int MemH[] = {2, 4, 7};
    int sizeH = sizeof(MemH) / sizeof(int);
    // Memoria Z
    int MemZ[sizeH + sizeY - 1];

    i = 0;
    while(i < (sizeY + sizeH - 1)){
        currentZ = 0;
        j = 0;
        while(j < sizeY){
            if(((i - j) >= 0) && ((i - j) < sizeH)){
                currentZ += MemH[i - j] * MemY[j];
            }
            j++;
        }
        MemZ[i] = currentZ;
        i++;
    }

    // Imprime resultado
    printf("Z = ");
    for (int i = 0; i < sizeH + sizeY - 1; i++){
        printf("%d ", MemZ[i]);
    }

    return 0;
}
```

Figura-3. Código generado en lenguaje C para corroborar el algoritmo propuesto.

Para validar el funcionamiento, se ejecutaron diferentes pruebas del código propuesto. A su vez, se realizaron las mismas pruebas haciendo uso de la función *conv(u, v)* de MATLAB, la cual retorna un vector con el resultado de la convolución entre los vectores ingresados como parámetros *u* y *v*. La **Tabla-2** muestra las pruebas propuestas y sus resultados obtenidos.

Señales de entrada		Resultados obtenidos	
$Y(n)$	$H(n)$	Código en C	MATLAB
{5, 4, 3, 2, 1}	{2, 4, 7}	10 28 57 44 31 18 7	10 28 57 44 31 18 7
{5, 4, 3, 2, 1}	{5, 4, 3, 2, 1}	25 40 46 44 35 20 10 4 1	25 40 46 44 35 20 10 4 1
{7, 5, 2, 7, 1, 4}	{5, 8, 4, 1, 2, 7, 8}	35 81 78 78 88 117 138 85 71 71 36 32	35 81 78 78 88 117 138 85 71 71 36 32
{5, 4, 3}	{1, 2, 3}	5 14 26 18 9	5 14 26 18 9
{15, 1, 12, 13, 14}	{6, 4, 5, 7}	90 66 151 236 203 205 161 98	90 66 151 236 203 205 161 98

Tabla-2. Pruebas propuestas y resultados obtenidos.

De los resultados obtenidos de la *Tabla-2*, podemos concluir que el algoritmo planteado funciona de forma satisfactoria.

4. DIAGRAMA ASM

La **Figura-4** muestra el diagrama de la máquina de estados algorítmica propuesta para el diseño del circuito a implementar, cabe resaltar que esta se obtuvo a partir del pseudocódigo mostrado en la **Figura-2**.

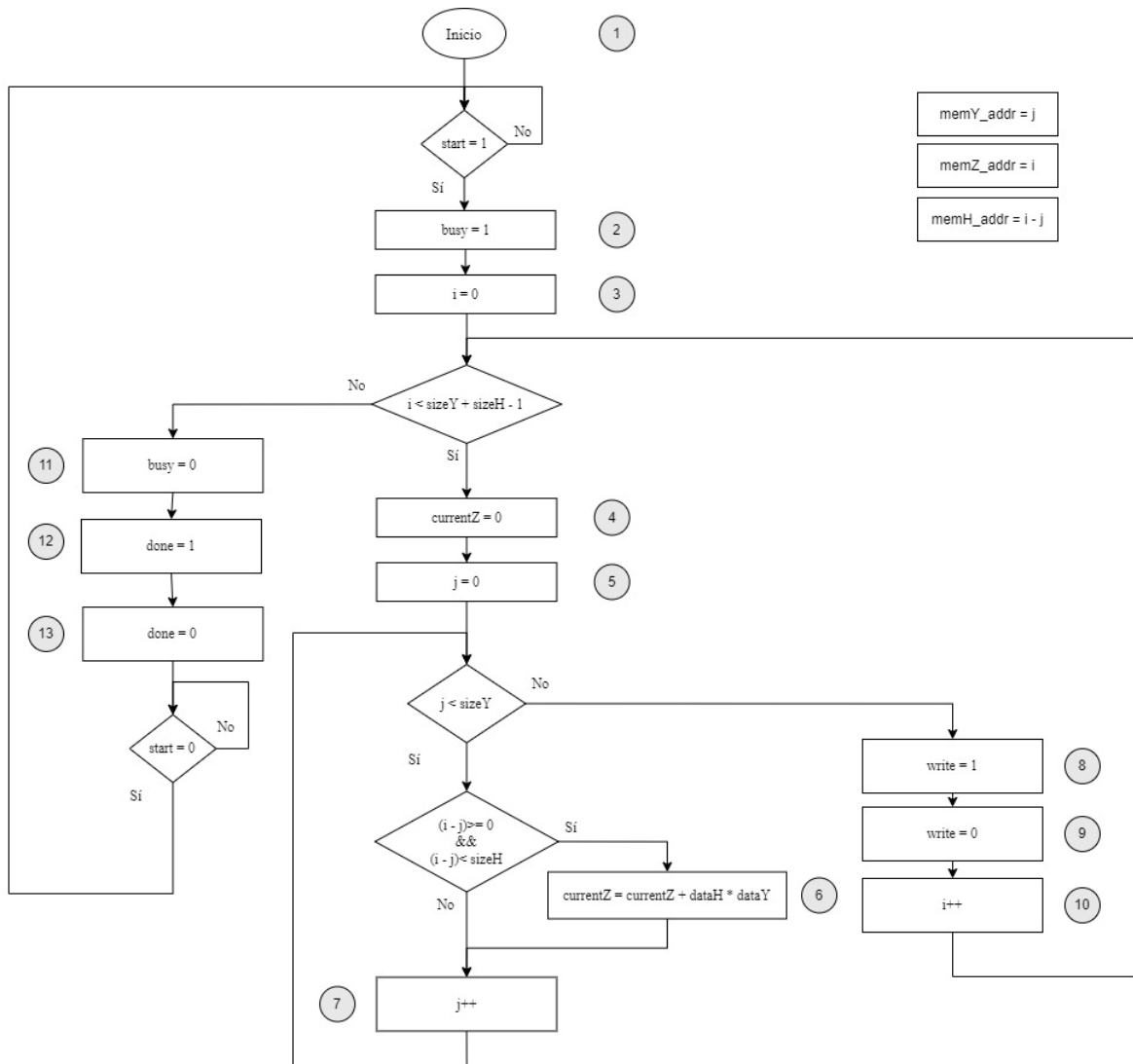
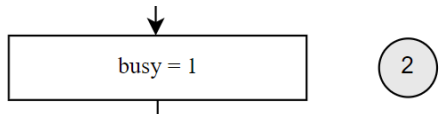
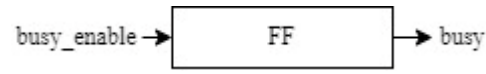
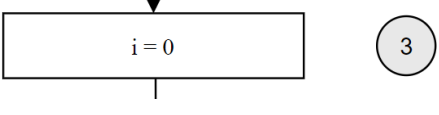
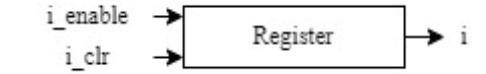
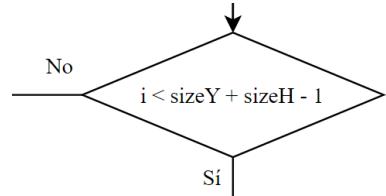
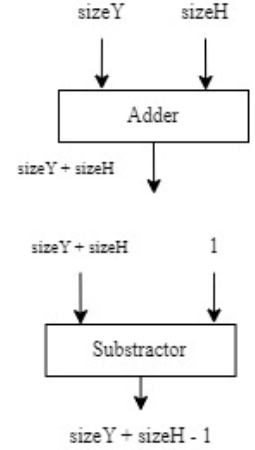


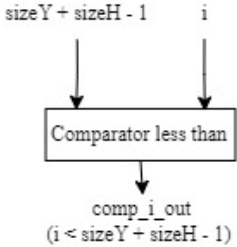
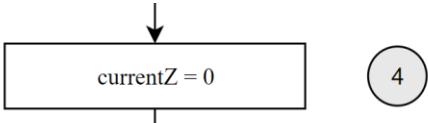

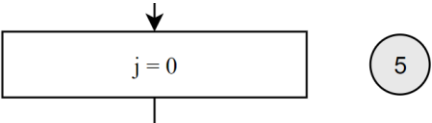

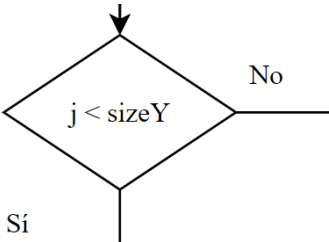
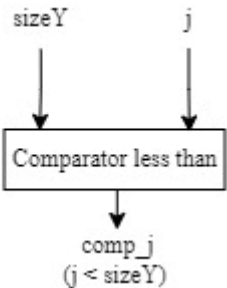
Figura-4. Diagrama ASM del algoritmo generado para el coprocesador de convolución.

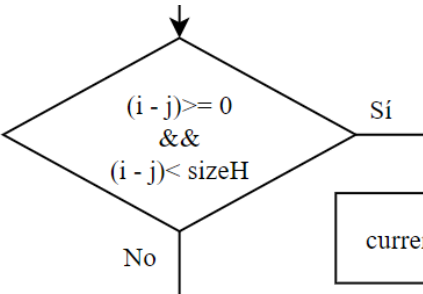
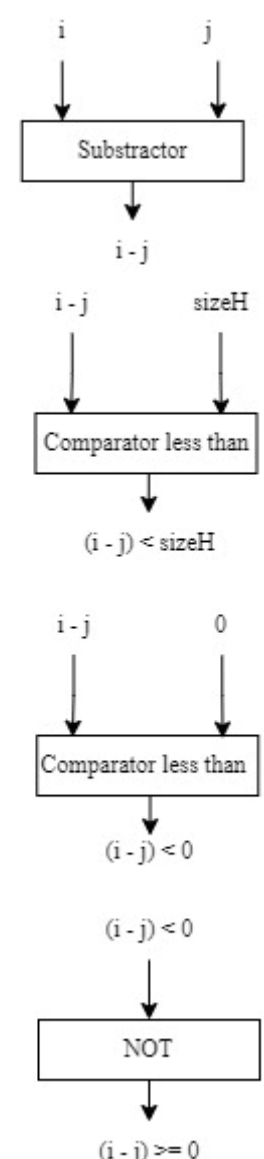
5. DIAGRAMA DEL *DATA PATH* Y MÁQUINA DE ESTADOS

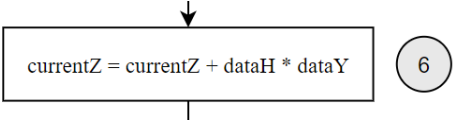
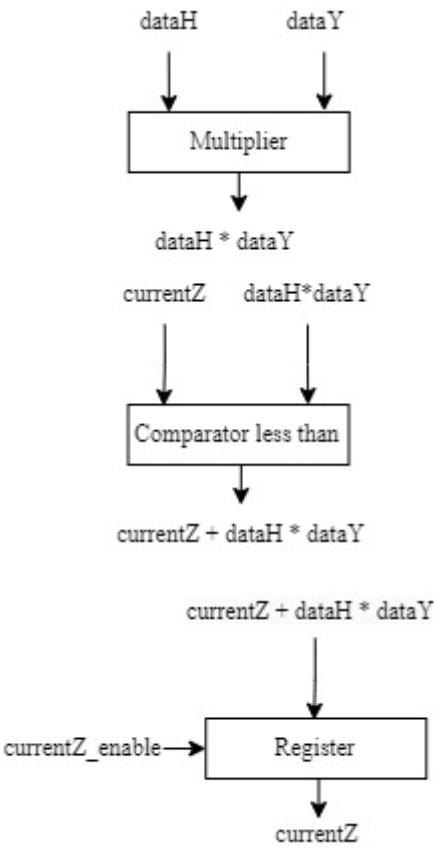
Bloques lógicos

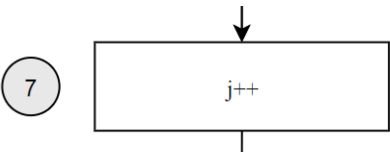
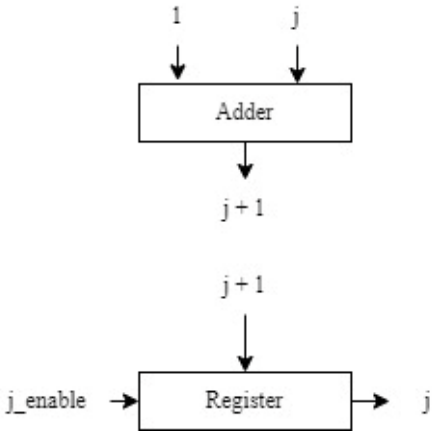
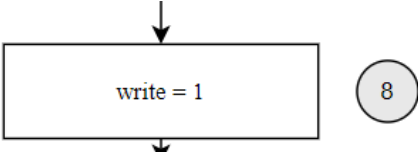
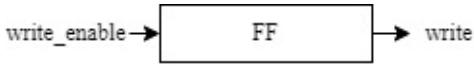
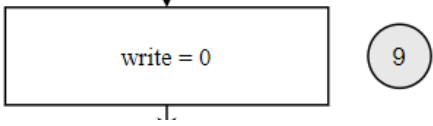
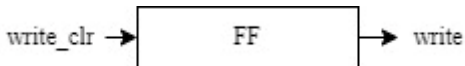
Utilizando el diagrama ASM de la **Figura-4**, identificamos cada uno de los bloques lógicos necesarios para cada elemento del diagrama. La **Tabla-3** resume la elección de bloques para cada estado.

No. Estado	Nombre de bloque	Bloque ASM	Bloque lógico	Detalles de selección
2	<i>busy</i>			Salida registrada para evitar <i>glitches</i> causados por lógica combinacional.
3	<i>i</i>			Registro con capacidad de limpiar el valor de salida y ponerlo en 0.
-	<i>comp_i</i>			<p>Sumador combinacional de con entradas <i>sizeY</i> y <i>sizeH</i>.</p> <p>Restador combinacional con entradas <i>sizeY + sizeH</i> y 1.</p> <p>Comparador menor qué entre <i>sizeY + sizeH - 1</i> y el registro <i>i</i>.</p>

				
4	<i>currentZ</i>			Registro con capacidad de limpiar el valor de salida y ponerlo en 0.
5	<i>j</i>			Registro con capacidad de limpiar el valor de salida y ponerlo en 0.
-	<i>comp_j</i>			Comparador menor que entre <i>j</i> y <i>sizeY</i> .

-	<i>comp_indexH</i>	 <pre> graph TD Entry(()) --> Decision{ "(i - j) >= 0 && (i - j) < sizeH" } Decision -- Sí --> current[current] Decision -- No --> Exit(()) </pre>	 <pre> graph TD i1[i] --> Subtractor[Subtractor] j1[j] --> Subtractor Subtractor --> i_j1["i - j"] i_j1 --> Comp1[Comparator less than] sizeH[sizeH] --> Comp1 Comp1 --> i_j_less_sizeH["(i - j) < sizeH"] i_j1 --> Comp2[Comparator less than] zero[0] --> Comp2 Comp2 --> i_j_less_zero["(i - j) < 0"] i_j_less_zero --> NOT[NOT] NOT --> i_j_ge_zero["(i - j) >= 0"] </pre>	<p>Restador entre i y j para obtener el índice o dirección de la memoria H a evaluar.</p> <p>Comparador menor que para verificar si $(i - j) < \text{sizeH}$ y determinar si la dirección de memoria es correcta.</p> <p>Comparador menor que y su respectiva negación para obtener la expresión lógica $(i - j) \geq 0$.</p>
---	--------------------	--	--	---

6	<i>currentZ</i>			<p>Bloques para obtener, y registrar el valor de $Z[n]$.</p> <p>Sumador entre el valor registrado y el producto de los datos actuales de H y Y.</p> <p>Multiplicador entre los datos actuales de dataH y dataY.</p> <p>Registro para almacenar el valor de currentZ.</p>
---	-----------------	---	--	---

7	<i>Incrementa j</i>			Sumador con entradas 1 y j, para obtener el valor a guardar en el registro j.
8	<i>Set_write</i>			Salida registrada, para poner a uno la señal de salida.
9	<i>Clr_write</i>			Salida registrada, para poner a cero la señal de salida.

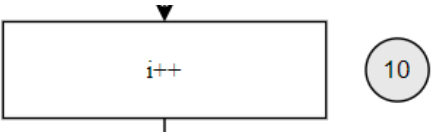
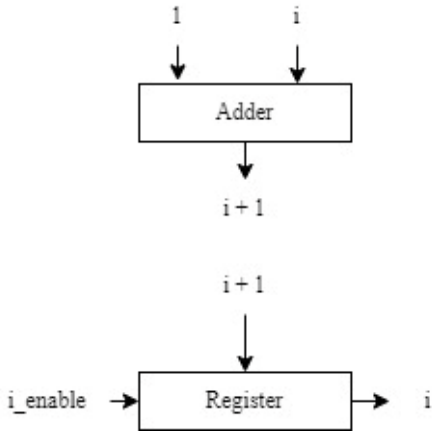
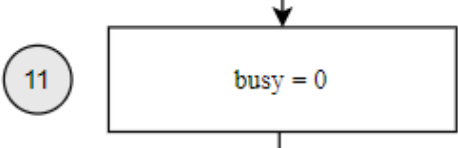
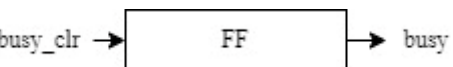
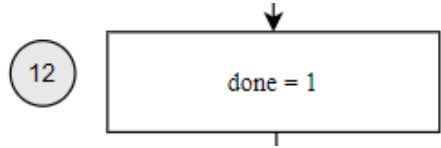
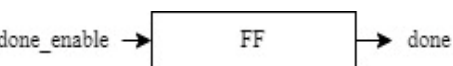
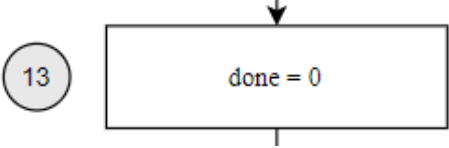
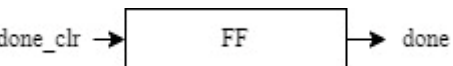
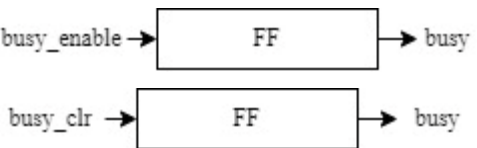
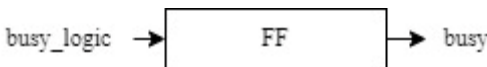
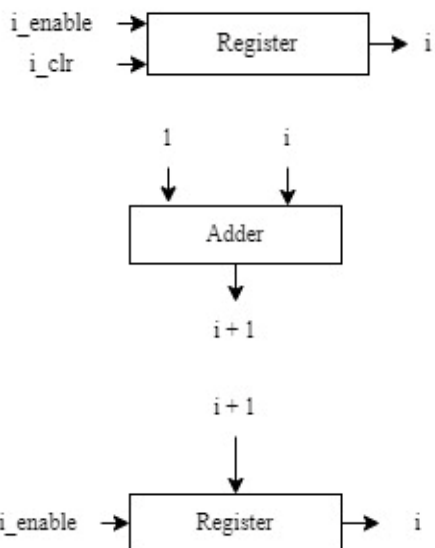
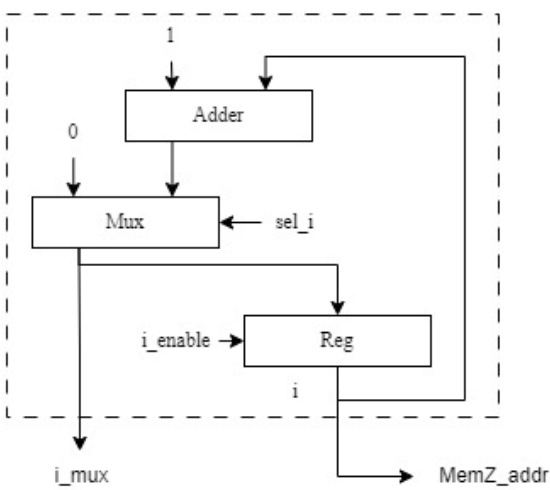
10	<i>Increment i</i>			Sumador con entradas 1 e i, para obtener el valor a guardar en el registro i.
11	<i>clr_busy</i>			Salida registrada, para poner a cero la señal de salida.
12	<i>Done_enable</i>			Salida registrada, para poner a uno la señal de salida.
13	<i>Done_clr</i>			Salida registrada, para poner a cero la señal de salida.

Tabla-3. Implementación de bloques lógicos a partir del diagrama ASM.

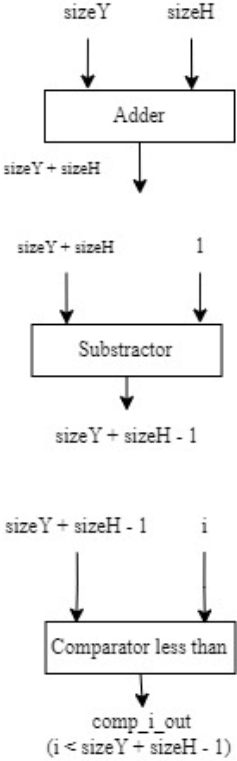
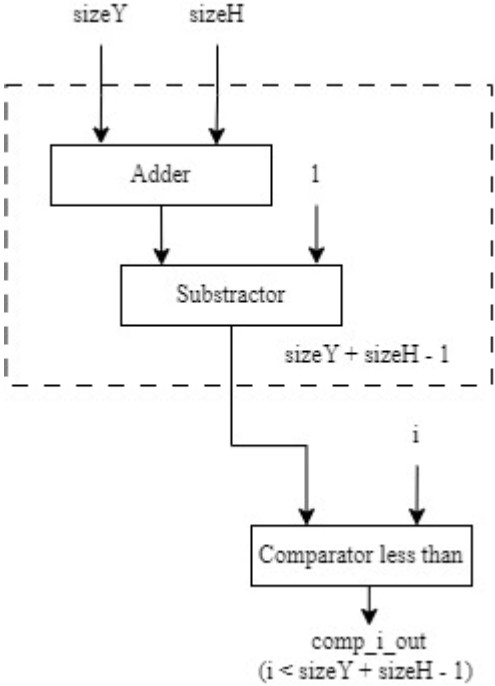
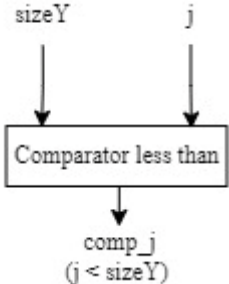
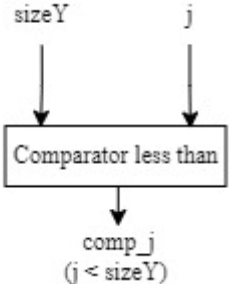
Combinación y optimización de bloques

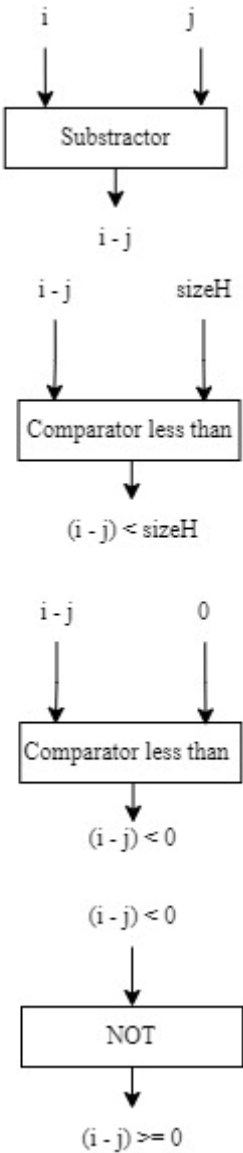
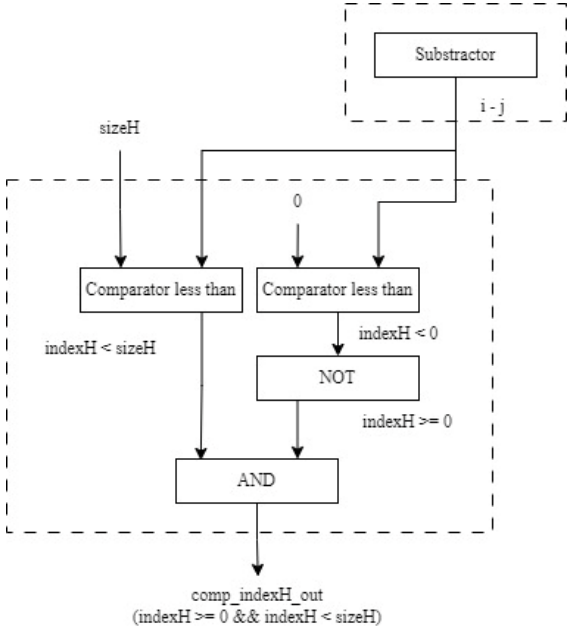
La **Tabla-4** muestra la combinación de estados y sus respectivos bloques lógicos, para poder llegar a la obtención de los bloques óptimos para cada elemento de la implementación.

Estados combinados	Nombre de bloque	Bloques separados	Bloque óptimo	Descripción
2 y 11	<i>busy</i>			Este bloque será optimizado al estar presente en la FSM del sistema como una salida registrada.
3 y 10	<i>i</i>			<p>Este bloque opera como un contador con cuenta de uno en uno. Además, es reinicializable mediante el uso del multiplexor, que selecciona entre el valor de reinicio y el resultado de la suma $i + 1$.</p> <p>El multiplexor es utilizado para ahorrar ciclos de reloj.</p>

4 y 6	<i>currentZ</i>	<pre> graph TD currentZ_enable1[currentZ_enable] --> Register1[Register] currentZ_clr1[currentZ_clr] --> Register1 Register1 --> currentZ1[currentZ] dataH1[dataH] --> Multiplier1[Multiplier] dataY1[dataY] --> Multiplier1 Multiplier1 --> product[dataH * dataY] currentZ1 --> Comparator[Comparator less than] product --> Comparator Comparator --> sum[currentZ + dataH * dataY] currentZ_enable2[currentZ_enable] --> Register2[Register] sum --> Register2 Register2 --> currentZ2[currentZ] </pre>	<pre> graph TD dataH2[dataH] --> Multiplier2[Multiplier] dataY2[dataY] --> Multiplier2 Multiplier2 --> Adder[Adder] Register2[Register] --> Adder Adder --> Register2 currentZ_en2[currentZ_en] --> Register2 currentZ_clr2[currentZ_clr] --> Register2 Register2 --> currentZ3[currentZ] </pre>	<p>Este bloque permite registrar las sumas de las multiplicaciones de los datos de las dos señales de entrada para el cálculo de convolución.</p> <p>Note que las señales de control nos permiten registrar el valor del sumador actual, y reiniciar la cuenta cada que tengamos una nueva iteración del cálculo de una muestra de salida.</p>
-------	-----------------	---	--	--

5 y 7	<i>j</i>			Este bloque cuenta con el mismo funcionamiento que el del contador <i>i</i> , sin embargo, note cómo la dirección de la memoria de salida es tomada desde el valor combinacional, pero ahorrar un estado de la fsm, y apuntar a la dirección de memoria correcta.
8 y 9	<i>write</i>			Este bloque será optimizado al estar presente en la FSM del sistema como una salida registrada.
12 y 13	<i>donw</i>			Este bloque será optimizado al estar presente en la FSM del sistema como una salida registrada.

-	<i>Comp_i</i>	 <pre> graph TD sizeY --> Adder sizeH --> Adder Adder -- "sizeY + sizeH" --> Subtractor Subtractor -- "1" --> Subtractor Subtractor -- "sizeY + sizeH - 1" --> Comparator i --> Comparator Comparator -- "comp_i_out (i < sizeY + sizeH - 1)" --> out </pre>	 <pre> graph TD sizeY --> Adder sizeH --> Adder Adder -- "sizeY + sizeH" --> Subtractor Subtractor -- "1" --> Subtractor Subtractor -- "sizeY + sizeH - 1" --> Comparator i --> Comparator Comparator -- "comp_i_out (i < sizeY + sizeH - 1)" --> out </pre>	Este bloque de configuración utiliza elementos puramente combinacionales para agilizar el flujo de la fsm.
-	<i>Comp_j</i>	 <pre> graph TD sizeY --> Comparator j --> Comparator Comparator -- "comp_j (j < sizeY)" --> out </pre>	 <pre> graph TD sizeY --> Comparator j --> Comparator Comparator -- "comp_j (j < sizeY)" --> out </pre>	Este bloque no tiene optimización.

-	Comp_indexH	 <pre> graph TD i --> Subtractor j --> Subtractor Subtractor --> ij["i - j"] ij --> Comp1["Comparator less than"] sizeH --> Comp1 Comp1 --> cond1["(i - j) < sizeH"] ij --> Comp2["Comparator less than"] 0 --> Comp2 Comp2 --> cond2["(i - j) < 0"] cond2 --> NOT NOT --> cond3["(i - j) >= 0"] </pre>	 <pre> graph TD subgraph DashedBox [] direction TB sizeH --> Comp1["Comparator less than"] ij["i - j"] --> Comp1 Comp1 --> cond1["indexH < sizeH"] 0 --> Comp2["Comparator less than"] ij --> Comp2 Comp2 --> cond2["indexH < 0"] cond2 --> NOT NOT --> cond3["indexH >= 0"] cond1 --> AND cond3 --> AND AND --> out["comp_indexH_out (indexH >= 0 && indexH < sizeH)"] end subgraph SubtractorBox [] direction TB Subtractor end ij --> Subtractor Subtractor --> ij </pre>	<p>Este bloque de configuración utiliza elementos puramente combinacionales para agilizar el flujo de la fsm.</p> <p>Para el uso de únicamente un comparador en la condición:</p> <p style="text-align: center;">$\text{indexH} \geq 0$</p> <p>se utiliza un comparador menor qué y una compuerta NOT.</p>
---	-------------	---	--	---

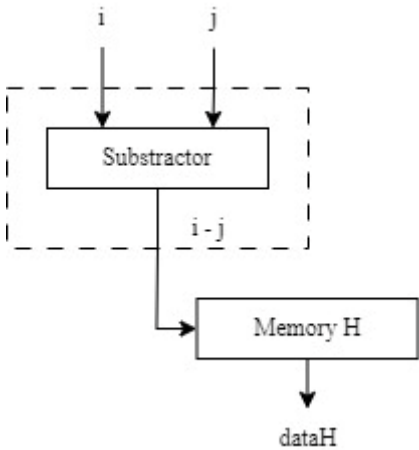
-	<i>MemoryH</i>		 <pre> graph TD i[i] --> Subtractor j[j] --> Subtractor Subtractor -- "i - j" --> MemoryH[Memory H] MemoryH --> dataH[dataH] subgraph DashedBox [] Subtractor end </pre>	<p>Este bloque permite acceder a la segunda señal de entrada del cálculo de la convolución, almacenada dentro de una memoria ROM, siendo $i - j$, la dirección de la memoria.</p>
---	----------------	--	--	--

Tabla-4. *Combinación de estados y obtención de bloques óptimos.*

Diagrama del data path

La **Figura-5** muestra el *data path* generado a partir de la máquina de estados algorítmica de la **Figura-4**, y de los bloques básicos y optimizaciones anteriores.

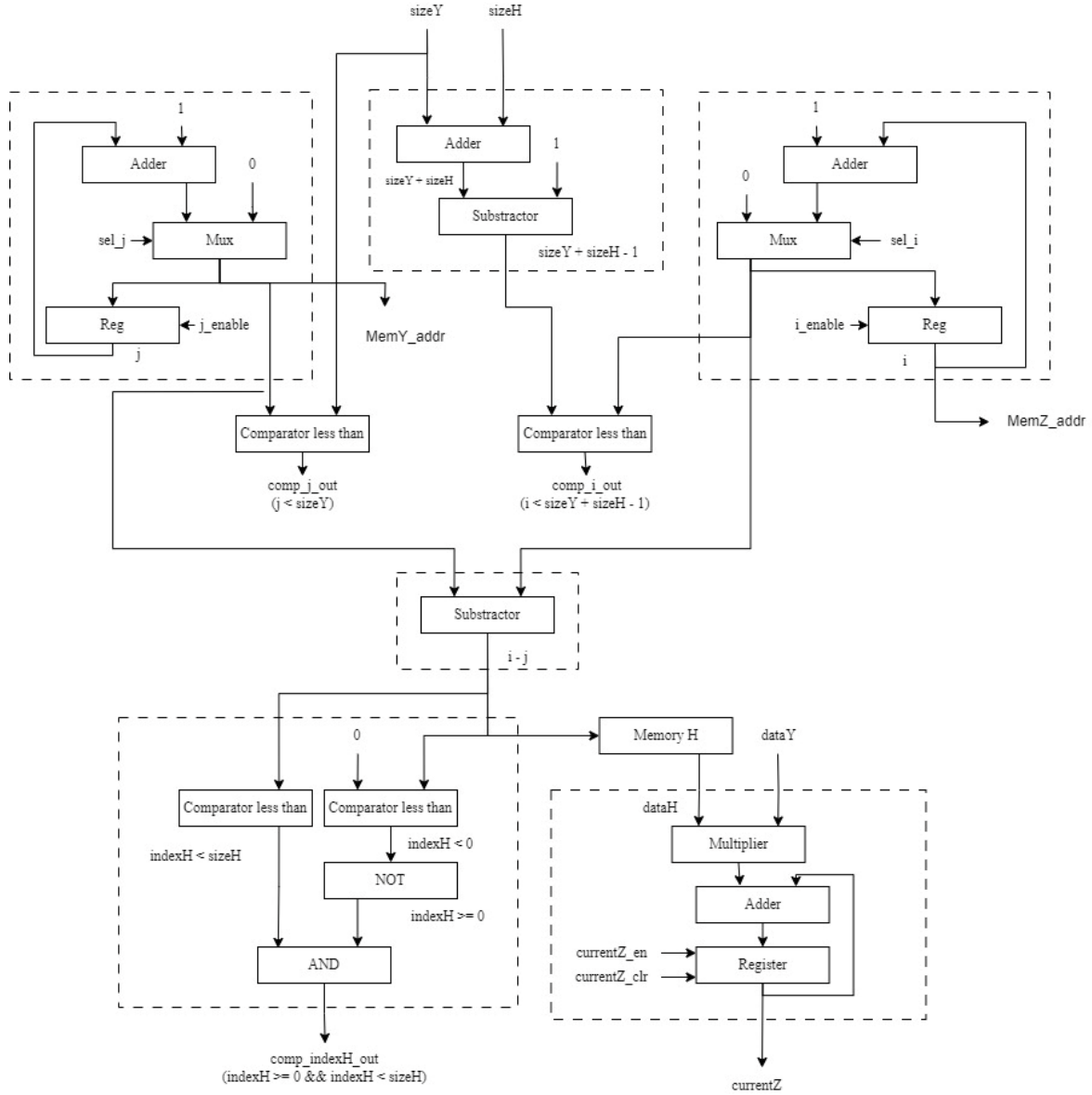


Figura-5. Data path propuesto para la implementación del coprocesador de convolución.

Máquina de estados

A partir del *data path* de la **Figura-5** podemos determinar las señales de entrada y de salida de la máquina de estados que se encargará de controlar el flujo de ejecución del programa, generando las señales correspondientes para poder implementar el algoritmo de convolución. La **Figura-6** muestra el bloque de la máquina de estados finitos.

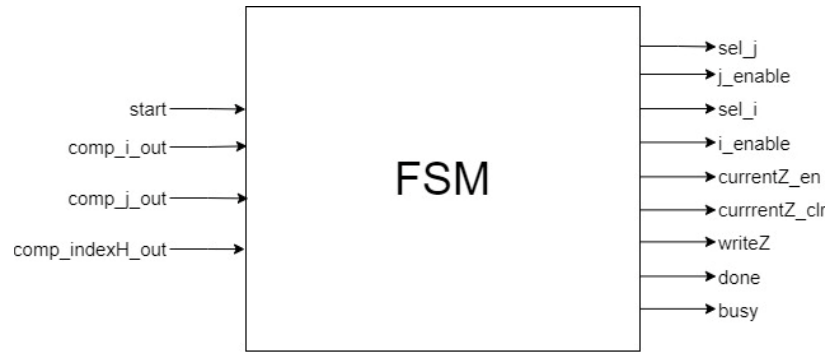


Figura-6. Señales de entrada y de salida de la máquina de estados.

La **Tabla-5** muestra la relación de las entradas y salidas de la máquina de estados, para cada uno de los estados de esta, obtenidos del diagrama ASM de la **Figura-4**.

State	Inputs/Value	Outputs/Value
1	start/0 (S1) start/0 (S13)	busy = 0
2	start/1 (S1)	busy = 1
3		sel_i = 1 i_enable = 1
4	(comp_i_out)/1 (S3) (comp_i_out)/1 (S10)	currentZ_clr = 1
5		sel_j = 1 j_enable = 1
6	(comp_j_out/1 & comp_indexH_out/1) (S5) (comp_j_out/1 & comp_indexH_out/1) (S7)	currentZ_en= 1
7	- (S6) (comp_j_out/1 & comp_indexH_out/0) (S5) (comp_j_out/1 & comp_indexH_out/0) (S7)	sel_j = 0 (default) j_enable = 1
8	(comp_j_out)/0 (S5) (comp_j_out)/0 (S7)	write = 1
9		write = 0
10		sel_i = 0 (default) i_enable = 1
11	(comp_i_out)/0 (S3) (comp_i_out)/0 (S10)	busy = 0
12		done = 1
13	- (S12) - start/1 (S13)	done = 0

Tabla-5. Relación de entradas y salidas de la máquina de estados.

Haciendo uso de la **Tabla-4** es posible determinar el diagrama de estados, el cual se muestra en la **Figura-7**.

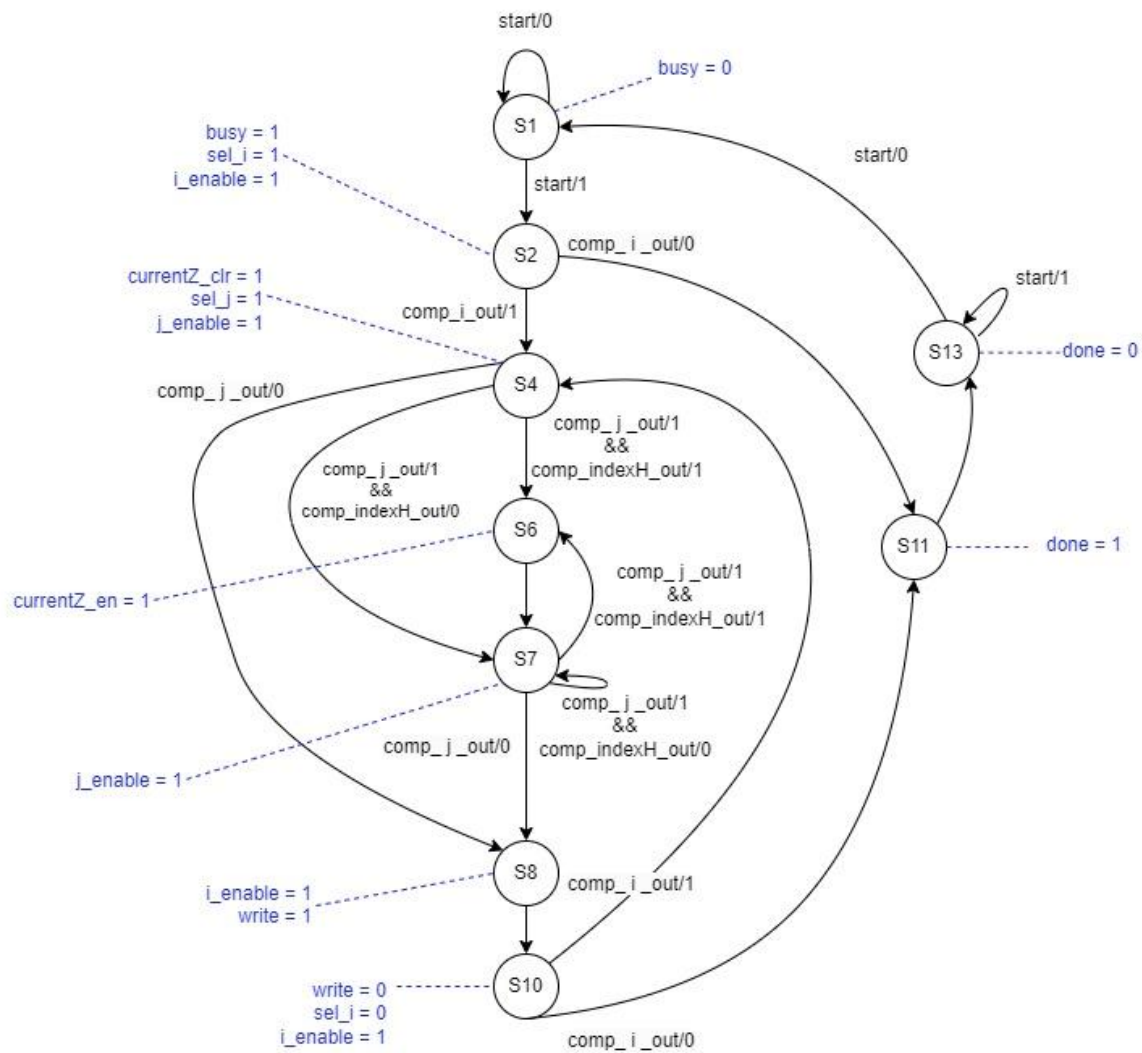


Figura-7. Diagrama de estados.

6. RESULTADOS DE SIMULACIÓN Y SÍNTESIS EN FPGA

En este apartado se muestra la información más relevante de los resultados de simulación y síntesis al utilizar la herramienta de síntesis *Quartus* para el FPGA MAX10DAF484C7G.

a. Esquemático Top Level generado por la herramienta de síntesis

La **Figura-8** muestra el esquemático Top Level generado por la herramienta, en él es posible observar todos los elementos que componen el *data path* propuesto.

Por otro lado, la **Figura-9** muestra el diagrama de la máquina de estados de control de flujo de datos obtenida de la herramienta.

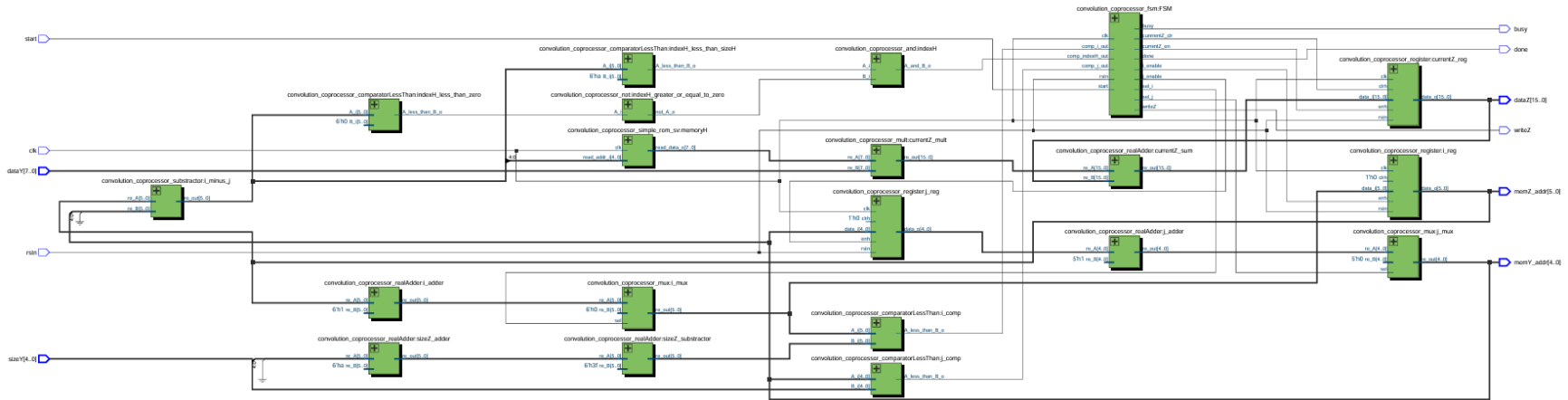


Figura-8. Esquemático Top Level generado por la herramienta de síntesis *Quartus*

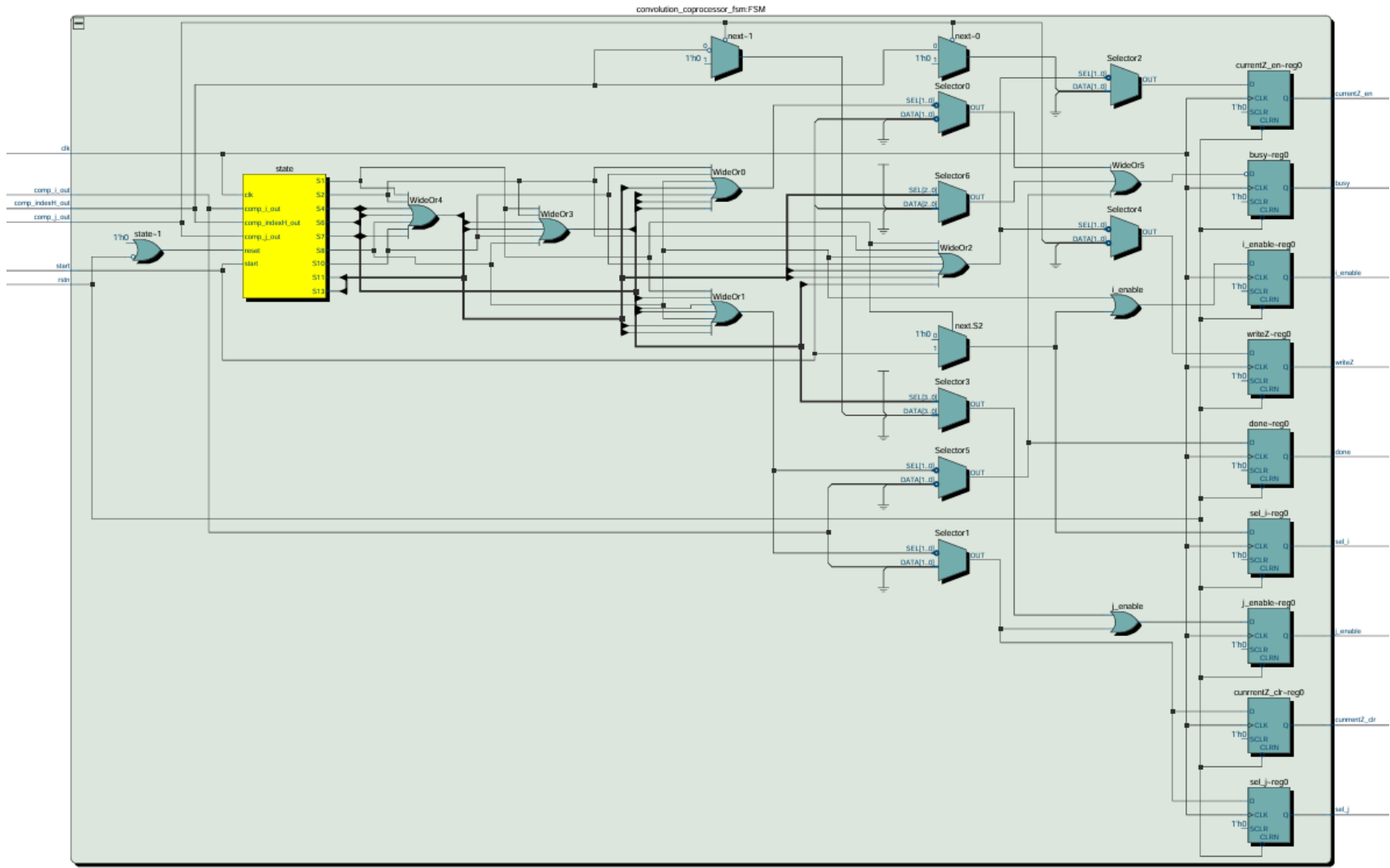


Figura-9. Esquemático FSM generado por la herramienta de síntesis Quartus.

b. Programa para generación de pruebas automáticas

Para facilitar el proceso de verificación, se generó el código mostrado en la **Figura-9**, el cual toma dos parámetros de entrada: (1) el tamaño de la señal Y ; y (2) el tamaño de la señal H . El programa genera los archivos con extensión .txt necesarios para cargar las memorias $MemY$ y $MemH$, las cuáles son utilizadas por la cama de pruebas para evaluar el funcionamiento del coprocesador. Además, el archivo utiliza el modelo de oro y muestra el resultado correcto de la convolución, para hacer la evaluación más sencilla.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define UPPER_LIMIT 50
#define LOWER_LIMIT 0
#define TWO_DIGITS_HEX_NUMMER 0x10

int* get_convolution(int *Y, int *H, int sizeY, int sizeH);
int* get_signal(int sizeSignal);
int get_random_number();
void printSignal(const char* signalName, int *signal, int sizeSignal);
void create_file(const char* fileName, int* signal, int sizeSignal);

int main(){
    int sizeY = 5;
    int sizeH = 10;
    int sizeZ = sizeY + sizeH - 1;
    int* y = NULL;
    int* h = NULL;
    int* z = NULL;

    // Initialize seed
    srand(time(0));
    // Get random signals
    y = get_signal(sizeY);
    h = get_signal(sizeH);
    // Calculate convolution
    z = get_convolution(y, h, sizeY, sizeH);
    // Generate files
    create_file("MemoryY.txt", y, sizeY);
    create_file("MemoryH.txt", h, sizeH);
    // Print Signals
    printSignal("Y", y, sizeY);
    printSignal("H", h, sizeH);
    printSignal("Z", z, sizeZ);
    // Free memory
    free(y);
    free(h);
    free(z);

    return 0;
}
```

```

int* get_convolution(int *Y, int *H, int sizeY, int sizeH){
    int i, j, currentZ;
    int* Z = (int*) malloc(sizeof(int) * (sizeY + sizeH - 1));

    i = 0;
    while(i < (sizeY + sizeH - 1)){
        currentZ = 0;
        j = 0;
        while(j < sizeY){
            if(((i - j) >= 0) && ((i - j) < sizeH)){
                currentZ += H[i - j] * Y[j];
            }
            j++;
        }
        Z[i] = currentZ;
        i++;
    }
    return Z;
}

int* get_signal(int sizeSignal){
    int* signal = (int*) malloc(sizeof(int) * sizeSignal);
    for(int i = 0; i < sizeSignal; i++){
        signal[i] = get_random_number();
    }
    return signal;
}

int get_random_number(){
    return (rand() % (UPPER_LIMIT - LOWER_LIMIT + 1));
}

void printSignal(const char* signalName, int *signal, int sizeSignal){
    printf("%s = [", signalName);
    for(int i = 0; i < sizeSignal; i++){
        printf("%X ", signal[i]);
    }
    printf("]\n");
}

void create_file(const char* fileName, int* signal, int sizeSignal){
    FILE* file = fopen(fileName, "wb+");
    for(int i = 0; i < sizeSignal; i++){
        if(signal[i] < TWO_DIGITS_HEX_NUNMER){
            fprintf(file, "%X", 0);
        }
        fprintf(file, "%X\n", signal[i]);
    }
}

```

Figura-9. Programa para generación automática de pruebas.

c. Forma de onda de la simulación

Para comprobar el funcionamiento del circuito, se propone una simulación que cuente con las características adecuadas para probar la mayor cantidad de situaciones posibles.

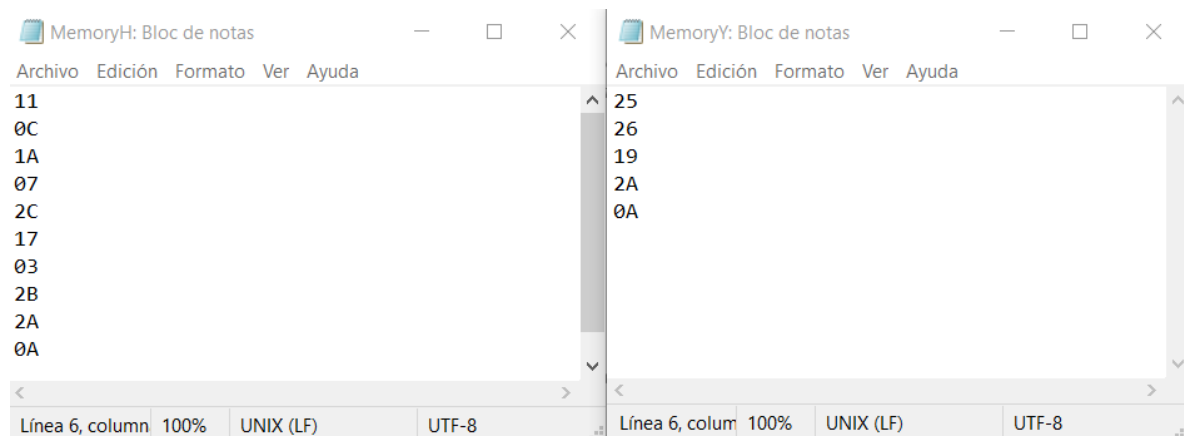
Características de simulación:

- Tamaño de la memoria H = 10
- Tamaño de la memoria Y = 5
- Período señal de reloj = 10 ns
- La señal de *start* inicia en 0.
- La señal de *start* cambia su estado a 1, después de unos ciclos de reloj.
- La señal de *start* permanece en uno hasta que se genere la interrupción de flanco de la señal *done*.
- Una vez obtenido el flanco que indica el proceso terminado, la señal *start* esperará un tiempo en su estado actual (1) y después cambiará a 0.
- Después de un tiempo, se enviará otra condición de inicio (*start* = 1).

La **Figura-10a** muestra las señales generadas de forma aleatoria, mientras que la **Figura-10b** muestra la forma en la que los archivos de datos de las memorias son generados.

```
Y = [25 26 19 2A A ]  
H = [11 C 1A 7 2C 17 3 2B 2A A ]  
Z = [275 442 733 8D5 C92 F46 A4F 1066 123D D45 CC2 98C 348 64 ]
```

a)



b)

Figura-10. a) Valores de las señales de entrada obtenidos de forma aleatoria, y el resultado de la convolución con el modelo de oro. b) Formato de los archivos de datos de memoria generados.

Forma de onda de la simulación

La **Figura-11** muestra el resultado obtenido de la simulación propuesta. A este nivel de detalle, es importante evaluar como parámetro principal la memoria de salida. Comparando los resultados obtenidos en la simulación, con los obtenidos por el modelo de oro, podemos concluir que el algoritmo está funcionando de forma satisfactoria.

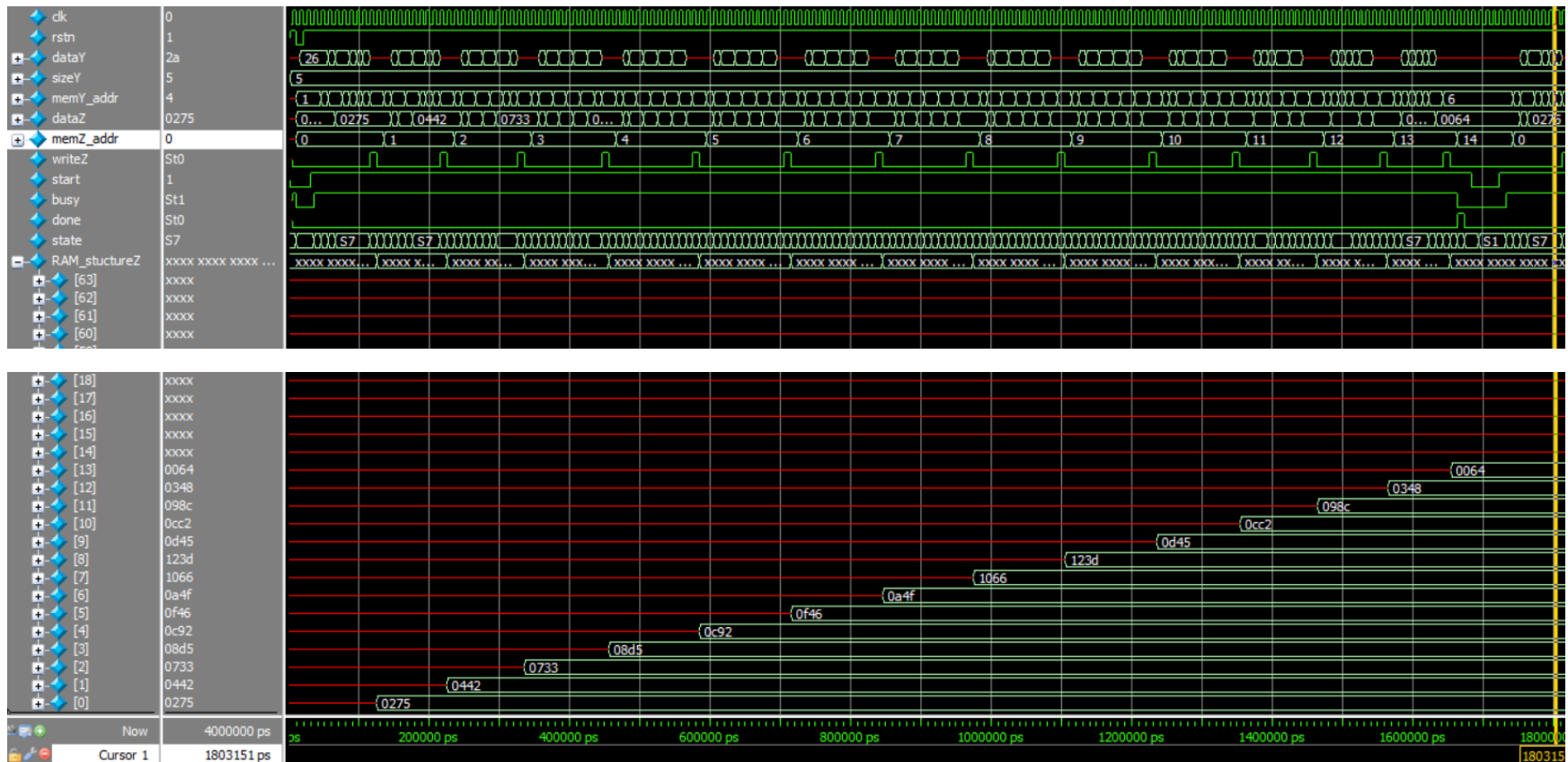


Figura-11. Forma de onda obtenida de la simulación propuesta.

<i>n</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>	<i>11</i>	<i>12</i>	<i>13</i>
<i>Modelo de oro</i>	275	442	733	8D5	C92	F46	A4F	1066	123D	D45	CC2	98C	348	64
<i>Simulación</i>	275	442	733	8D5	C92	F46	A4F	1066	123D	D45	CC2	98C	348	64

Tabla-6. Comparación de los resultados obtenidos.

Condición de inicio

La **Figura-12** muestra cómo después del *reset* asíncrono, el estado de la FSM es S1, las señales de *busy* y *done* son puestas en 0, y como condición de simulación, la señal de *start* permanece en cero. Estos estados prevalecen hasta que la señal de *start* da el pulso de inicio, por lo que el estado de la FSM cambia a S2 y la señal de *busy* es activada. Además, es posible ver cómo el coprocesador comienza a trabajar.

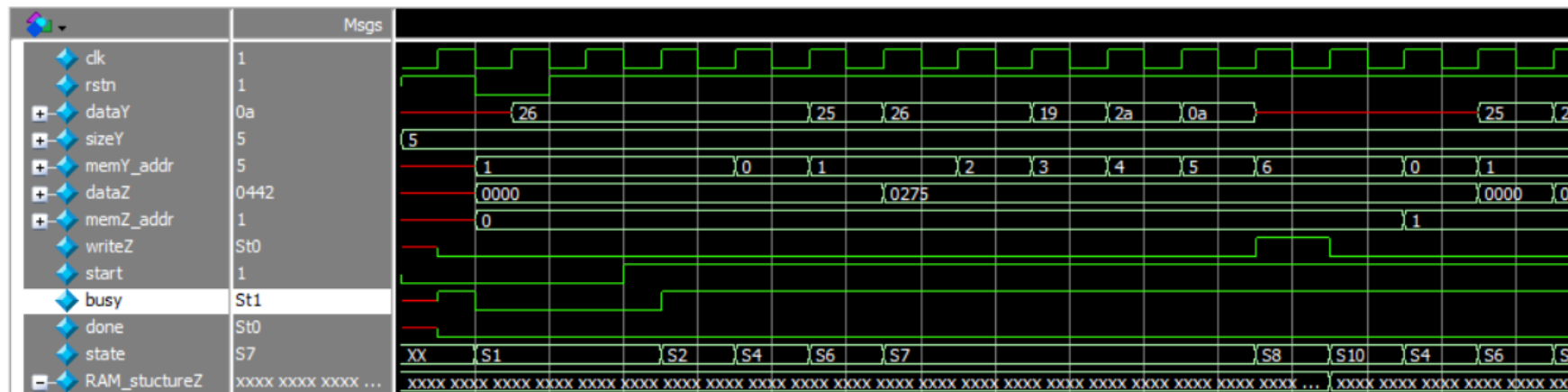


Figura-12. Evaluación de condición de inicio.

Condición de paro

Una vez el coprocesador a terminado el proceso, la señal *busy* cambia su estado a cero, mientras que la señal *done* genera el pulso necesario para indicar que el proceso se ha terminado. Note cómo la señal de *start* permanece en 1. El estado de la FSM se mantiene en S13 hasta que la señal de *start* sea igual a 0, lo que hace que el estado de la FSM ahora sea S1 y el coprocesador esté listo para iniciar su proceso de nuevo. De igual forma a la condición de inicio mostrada anteriormente, es posible identificar que la condición de paro funciona de forma correcta, y el proceso vuelve a iniciar, una vez se haya generado la condición de inicio.

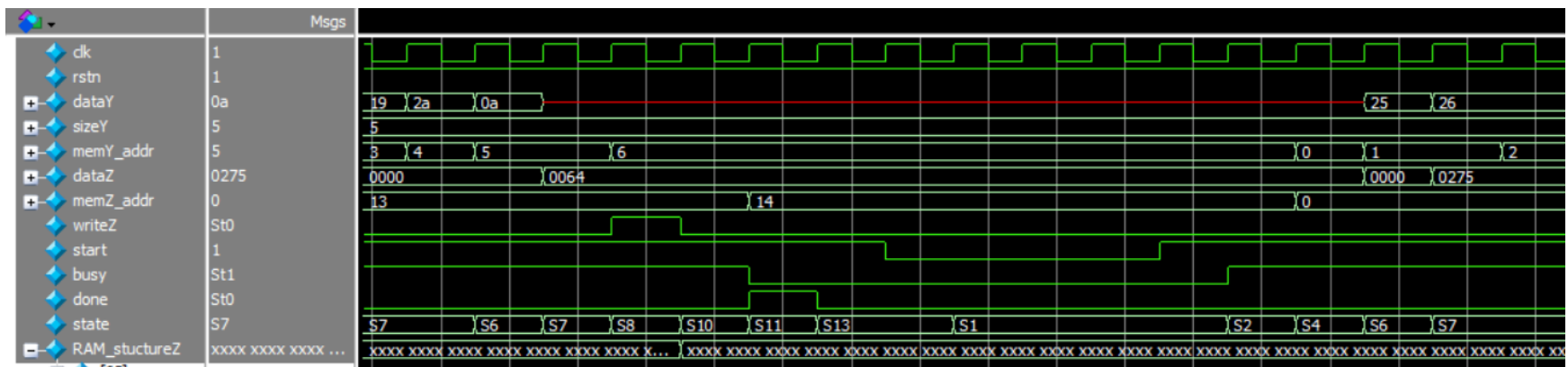


Figura-13. Evaluación de condición de paro.

d. Área

Mediante el uso de la herramienta de síntesis, es posible obtener el área o el número de elementos lógicos utilizados en la implementación del proyecto. La **Figura-14** resume los resultados obtenidos.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri May 10 00:09:38 2024
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	convolution_coprocessor
Top-level Entity Name	convolution_coprocessor
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	97 / 49,760 (< 1 %)
Total registers	41
Total pins	46 / 360 (13 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	1 / 288 (< 1 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Análisis de Área	
Parámetro	Resultado
Total logic elements	97 / 49,760 (< 1%)
Total registers	41
Total pins	46 / 360 (13 %)
Embedded Multiplier 9-bit elements	1 / 288 (< 1%)

Figura-14. Análisis de área.

e. Frecuencia Máxima

Hacemos uso de *Quartus*, para observar el analizador de tiempo y obtener la frecuencia máxima a la que se garantiza el correcto funcionamiento del coprocesador. La **Figura-15** muestra los resultados obtenidos.

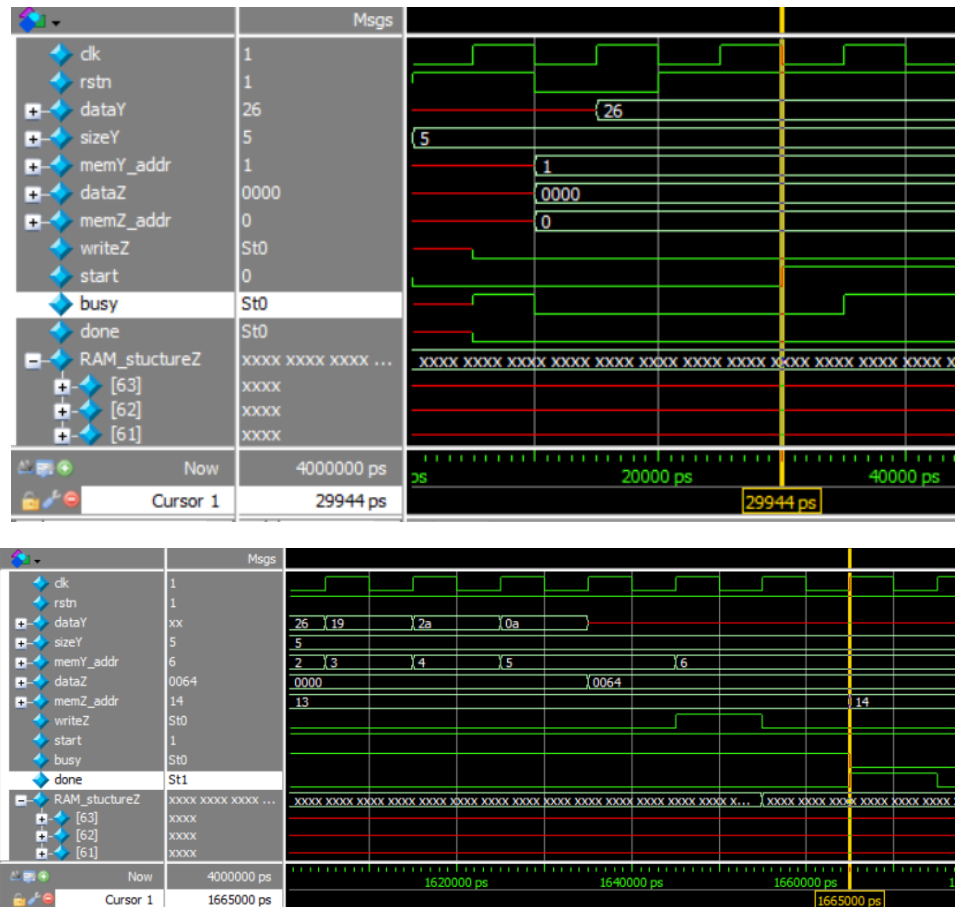
Slow 1200mV 85C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	173.58 MHz	173.58 MHz	clk	

Frecuencia máxima	
clk	173.58 MHz

Figura-15. Análisis de frecuencia máxima.

f. Número de ciclos de reloj para señal conocida

Para determinar el número de ciclos de reloj para la simulación con las características mostradas en el apartado *Forma de onda de la simulación*, utilizamos los cursores, posicionándolos en la transición de subida de la señal de *start* (condición de inicio) y en la transición de subida de la señal *done* (interrupción de completado). La **Figura-16** muestra los resultados obtenidos de este proceso.



Análisis número de ciclos de reloj		
Tiempo de inicio	t_{start}	29.944 us
Tiempo de completado	t_{done}	1665 us
Período clk	T_{clk}	10 us

Figura-16. Uso de cursores para la obtención de los tiempos de inicio y paro.

A partir de los resultados anteriores, y utilizando el período de la señal de reloj, es posible determinar el número de ciclos que le toma al coprocesador para obtener la convolución de dos señales de 10 y 5 muestras respectivamente.

$$Ciclos_{clk} = \frac{t_{done} - t_{start}}{T_{clk}} = \frac{1,665 \text{ us} - 29.944 \text{ us}}{10 \text{ us}} = 163.5056 \text{ ciclos} \approx \mathbf{167 \text{ ciclos}}$$

Análisis ciclos de reloj	
Ciclos	167

Tabla-7. Ciclos de reloj necesarios para procesar dos señales de 5 y 10 muestras respectivamente.

7. CONCLUSIONES

En este documento se presenta el desarrollo, análisis y verificación de un coprocesador de convolución unidimensional utilizando la metodología *Top-Down* de diseño para *System-on-chip*. Se logró obtener un coprocesador funcional y sintetizable que realiza la convolución entre dos señales de entrada, a partir de un conjunto de requerimientos solicitados.

La implementación en lenguaje de descripción de hardware requirió el uso de dos unidades funcionales principales: el *data path* y la máquina de estados que lo controla. Estos dos módulos garantizan la correcta inferencia de hardware, además de facilitar los procesos de depuración. Esta implementación redujo el tiempo de análisis de problemas, siendo la sincronización entre módulos del *data path* el principal reto en su desarrollo.

Mediante el uso de la metodología *Top-Down* contamos con un plan de trabajo y un conjunto de herramientas previamente definidas, para facilitar los procesos de desarrollo, verificación y documentación del diseño de módulos de sistemas en chip. Debido a que la integración de módulos, inherente a estas tecnologías, puede llegar a ser complicada, el uso de las prácticas de diseño y este tipo de metodologías es crucial para garantizar un correcto desarrollo.

La propuesta de trabajo futuro consiste en optimizar el tiempo de ejecución, haciendo que uno de los iteradores del algoritmo utilice como límites de iteración el tamaño de la señal de entrada más pequeña, lo que garantiza una optimización en el tiempo de ejecución, vital en el procesamiento de algoritmos de la complejidad temporal $O(n^2)$.