Metodología de diseño de SoCs TAE 2024 26 de mayo de 2024
Hernández Carrazco Isaac Giovanni
Fuentes Avalos Luis Angel

Coprocesador de convolución

Descripción del problema

Implementar un coprocesador de convolución usando la metodología top-down.

El coprocesador hace una operación de convolución de una señal X con una señal Y. Estas señales estarán guardadas en memorias, por lo que el coprocesador debe de hacer la lógica necesaria para acceder a cada una de las muestras de las señales en las memorias.

Entradas y salidas

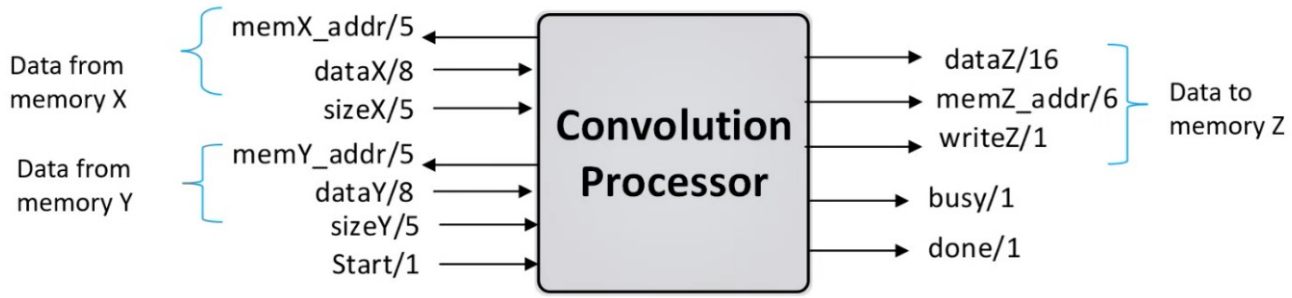

Figure 1: Diagrama de caja negra de entradas y salidas del módulo

Entradas

- ***dataX*** es una señal de 8 bits que contendrá la muestra de la señal X para convolucionar.
- ***sizeX*** es una señal de 5 bits que representará la cantidad de muestras dentro de la memoria X
- ***dataY*** es una señal de 8 bits que contendrá la muestra de la señal Y para convolucionar
- ***sizeY*** es una señal de 5 bits que representará la cantidad de muestras dentro de la memoria Y
- ***start*** es una señal de 1 bit que controlará el inicio del proceso de convolución

Salidas

- ***memX_addr*** es una señal de 5 bits que contendrá la dirección de la memoria X que se desea leer para obtener la muestra de X
- ***memY_addr*** es una señal de 5 bits que contendrá la dirección de la memoria X que se desea leer para obtener la muestra de Y
- ***dataZ*** es una señal de 16 bits que contendrá el resultado final de la convolución de las dos señales X y Y

- ***memZ_addr*** es una señal de 6 bits representará la dirección donde se escribirá el resultado final de la convolución en la memoria Z
- ***writeZ*** es una señal de 1 bit que controlará la memoria Z para habilitar la escritura
- ***busy*** es una señal de 1 bit que estará activa mientras se esté realizando el proceso de convolución
- ***done*** es una señal de 1 bit tipo *One Shot* que se activará cuando el proceso de convolución haya terminado

Algoritmo de resolución

El algoritmo diseñado e implementado es el siguiente:

```
   1. while start sea 0
   2. end while
// Inicializar variables
   3. busy = 1
   4. writeZ = 0
   5. dataZ = 0
   6. memZ_addr = 0
   7. memX_addr = 0
   8. memY_addr = 0
// Recorrer una vez todos los valores de Y con X estática
   9. for i desde 0 hasta sizeY, con paso de +1:
         10.      dataZ = 0
            // Obtener los valores de las multiplicaciones de cada elemento
         11.      for j desde 0 hasta sizeX e i, con paso de +1:
               12.      memX_addr = j
               13.      memY_addr = i – j
               14.      dataZ = dataZ + (dataX * dataY)
         15.      end for
         16.      writeZ = 1
         17.      writeZ = 0
         18.      memZ_addr = memZ_addr + 1
   19.      end for
// Recorrer los valores de X con Y estática
   20.      for i desde 1 hasta sizeX, con pasos de +1:
         21.      dataZ = 0
         22.      for j desde i, k desde sizeY-1, j hasta sizeX mientras k sea
            positivo, con pasos de j+1 y k-1:
               23.      memX_addr = j
               24.      memY_addr = k
               25.      dataZ = dataZ + (dataX * dataY)
         26.      end for
         27.      writeZ = 1
         28.      writeZ = 0
         29.      memZ_addr = memZ_addr + 1
   30.      end for
```

```
31.       busy = 0
32.       done = 1
33.       done = 0
```

Para explicar de mejor manera el algoritmo, véase las figuras 2, 3, y 4.

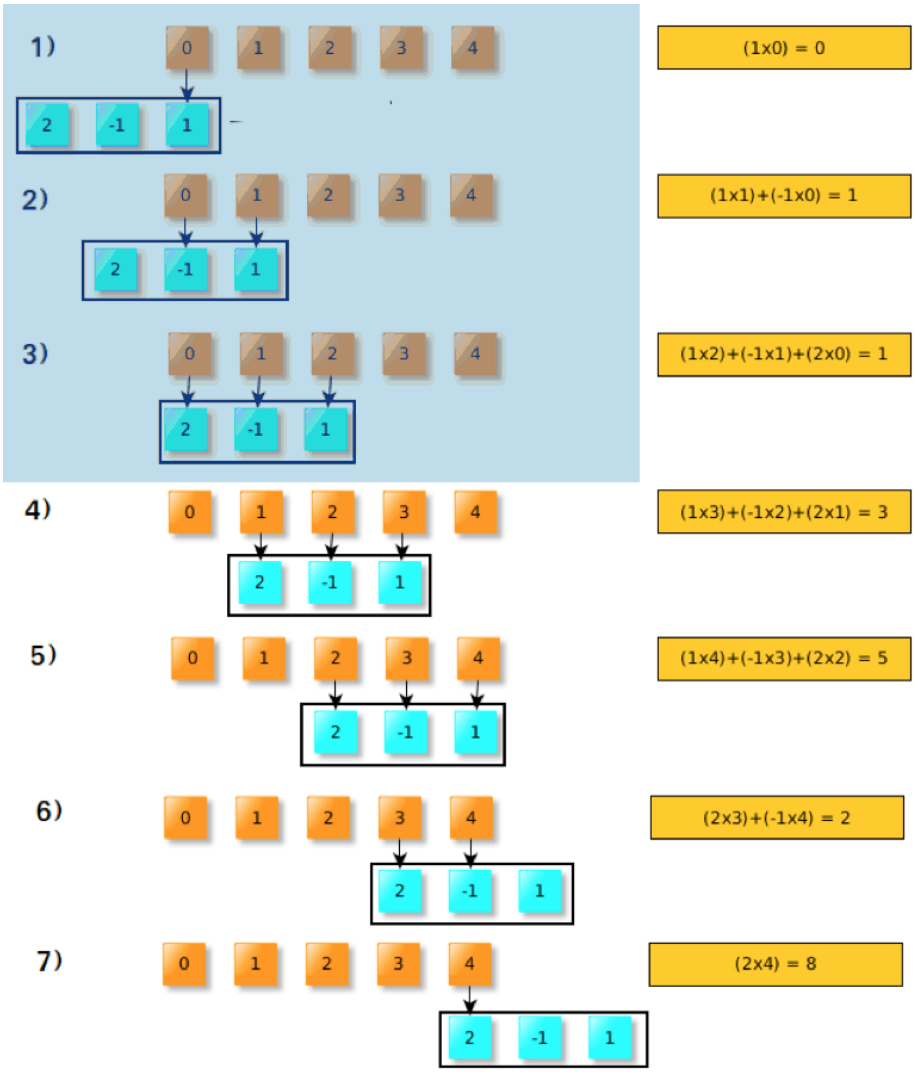

Figure 2: Primera parte del algoritmo de convolución. Y se recorre a la derecha en X que se queda estática.

En la primera parte del algoritmo, que corresponde de la línea 9 a la 19, se recorren todos los valores de Y hasta X, donde el bit más significativo de Y corresponde con el bit menos significativo de X. En esta parte se puede ver como X se queda estática mientras Y se desplaza sobre X.

En la segunda parte del algoritmo, que corresponde de la línea 20 a la 30, se recorren todos los valores de X hasta no estar “encima” de algún valor de Y, donde al final el bit menos significativo de X corresponde con el bit más significativo de Y.

En la figura 4 se observa como es que estos dos procesos ocurren internamente, en cada uno de sus estados, donde se itera cada muestra de X con cada muestra de Y.

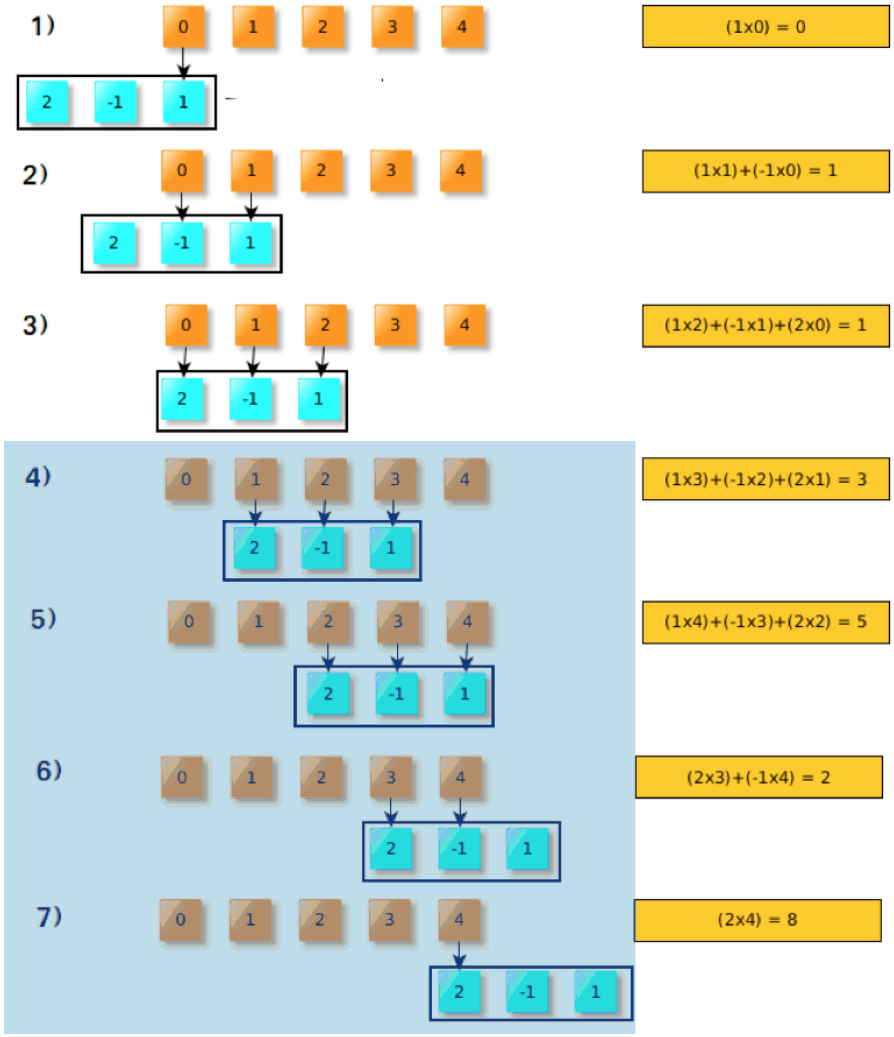

Figure 3: Segunda parte del algoritmo. Y se queda estática y X se recorre a la izquierda.

Figure 4: Proceso de la suma de productos de cada muestra.

Validación del algoritmo

Para comprobar el funcionamiento del algoritmo creado, se escribió un código en C que lo implementa:

```c
#include <stdio.h>
#include <stdint.h>

int main(void)
{
    uint16_t dataZ;
    uint8_t dataX, dataY;
    uint8_t sizeX, sizeY;
    uint8_t memX_addr, memY_addr, memZ_addr;

    uint8_t memX [32] = {5, 10, 15, 20, 25, 30};
    uint8_t memY [32] = {3,  6,  9, 12, 15, 18};
    uint16_t memZ [32] = {};

    sizeX = 2;
    sizeY = 6;
    memX_addr = 0;
    memY_addr = 0;
    memZ_addr = 0;

    for (int i = 0; i < sizeY; i++){
        dataZ = 0;
        for (int j = 0; (j<=i) && (j<sizeX); j++){
            // Cargar valores de memorias
            dataX = memX[j];
            dataY = memY[i-j];
            dataZ += dataX * dataY;
        }
        memZ[memZ_addr] = dataZ;
        memZ_addr++;
    }

    for (int i = 1; i < sizeX; i++){
        dataZ = 0;
        for (int j = i, k = sizeY-1; j < sizeX; j++, k--){
            dataX = memX[j];
            dataY = memY[k];
            dataZ += dataX * dataY;
        }
        memZ[memZ_addr] = dataZ;
        memZ_addr++;
    }
```

Figure 5: Código en C para comprobar el funcionamiento del algoritmo

Al final del código se muestran el resultado de la convolución en la consola o terminal.

Para las pruebas se usaron dos vectores que representan X y Y, los mismos de la figura 5. En cada prueba se cambiaron los valores de sizeX y sizeY para el proceso de convolución. Además se comprobó con Matlab el resultado de la operación para comparar los resultados. Las pruebas realizadas fueron las siguientes:

- sizeX = 2, sizeY = 6

```
[0] -- 15
[1] -- 60
[2] -- 105
[3] -- 150
[4] -- 195
[5] -- 240
[6] -- 180
Press <RETURN> to close this window...
```

```
>> convo
    15    60   105   150   195   240   180
```

- sizeX = 6, sizeY = 6

```
[0] -- 15
[1] -- 60
[2] -- 150
[3] -- 300
[4] -- 525
[5] -- 840
[6] -- 1050
[7] -- 1140
[8] -- 1095
[9] -- 900
[10] -- 540
Press <RETURN> to close this window...
```

```
>> convo
    15    60   150   300   525   840  1050  1140  1095   900   540
```

- sizeX = 5, sizeY = 4

```
[0] -- 15
[1] -- 60
[2] -- 150
[3] -- 300
[4] -- 450
[5] -- 510
[6] -- 465
[7] -- 300
Press <RETURN> to close this window...
```

```
>> convo
    15    60   150   300   450   510   465   300
```

Al analizar fácilmente los resultados obtenidos podemos concluir con el correcto funcionamiento del algoritmo para realizar la operación de convolución.

Table 1: Comparación de resultados obtenidos

sizeX	sizeY	Resultados iguales y correctos
2	6	√
6	6	√
5	4	√

Diagrama ASM

La figura 6 muestra el diagrama ASM (*Algorithm State Machine*) del proceso de convolución, con la implementación de varios ciclos for anidados, como se comprendió con los puntos anteriores.

En la figura 7 vemos a detalle la primera parte del algoritmo después de la inicialización de variables, que es el pedazo correspondiente de la figura 2 del proceso de convolución.

En la figura 8 se observa la segunda parte del algoritmo, con su figura 3 correspondiente del proceso de convolución.

Los estados se diferencian por número y color, donde varias operaciones dentro de un estado representa que pueden ser operaciones concurrentes, esto quiero decir que, no codepende alguna operación de de las demás, y no afecta el resultado esperado.

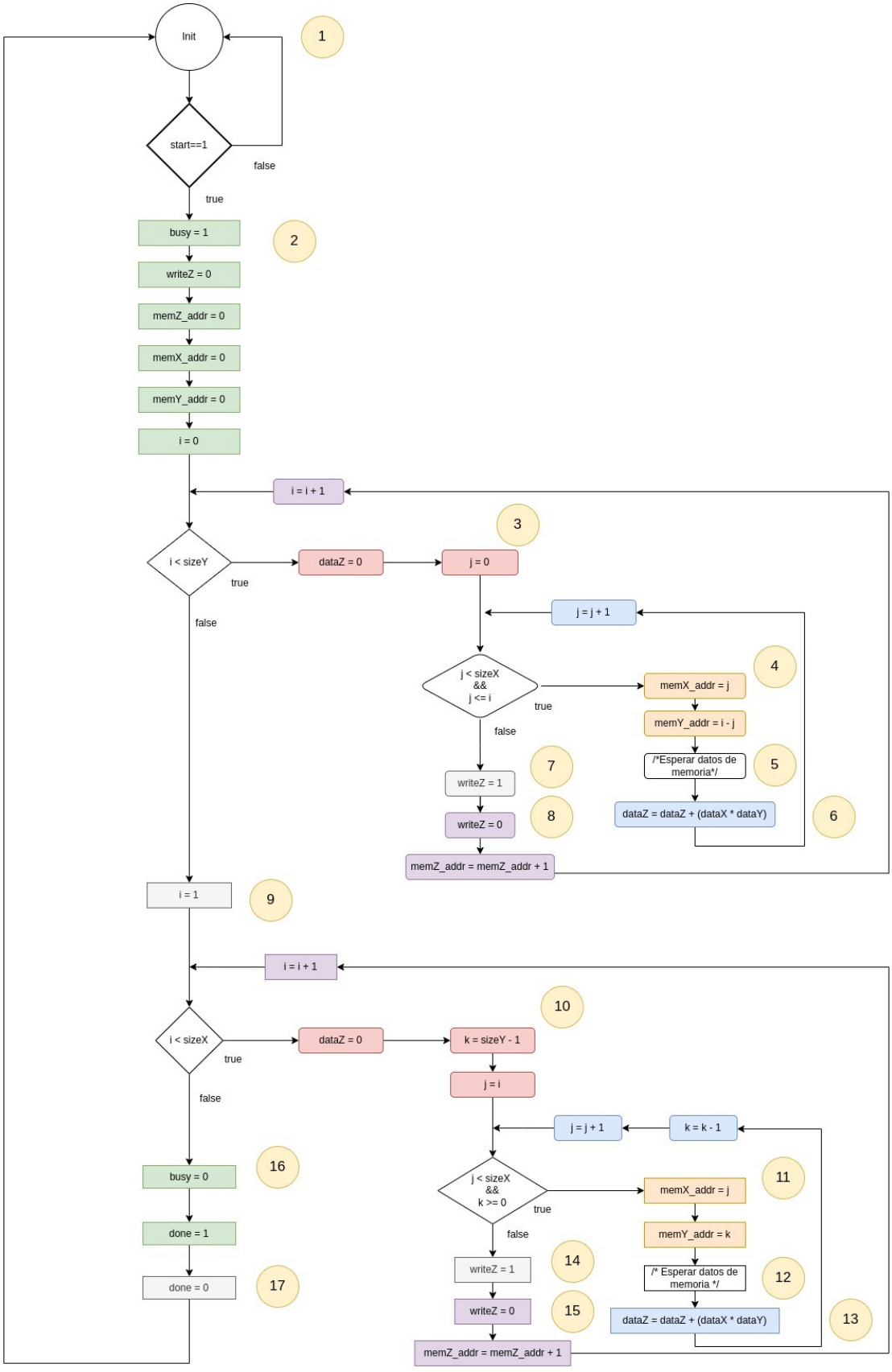

Figure 6: Diagrama ASM del módulo de convolución

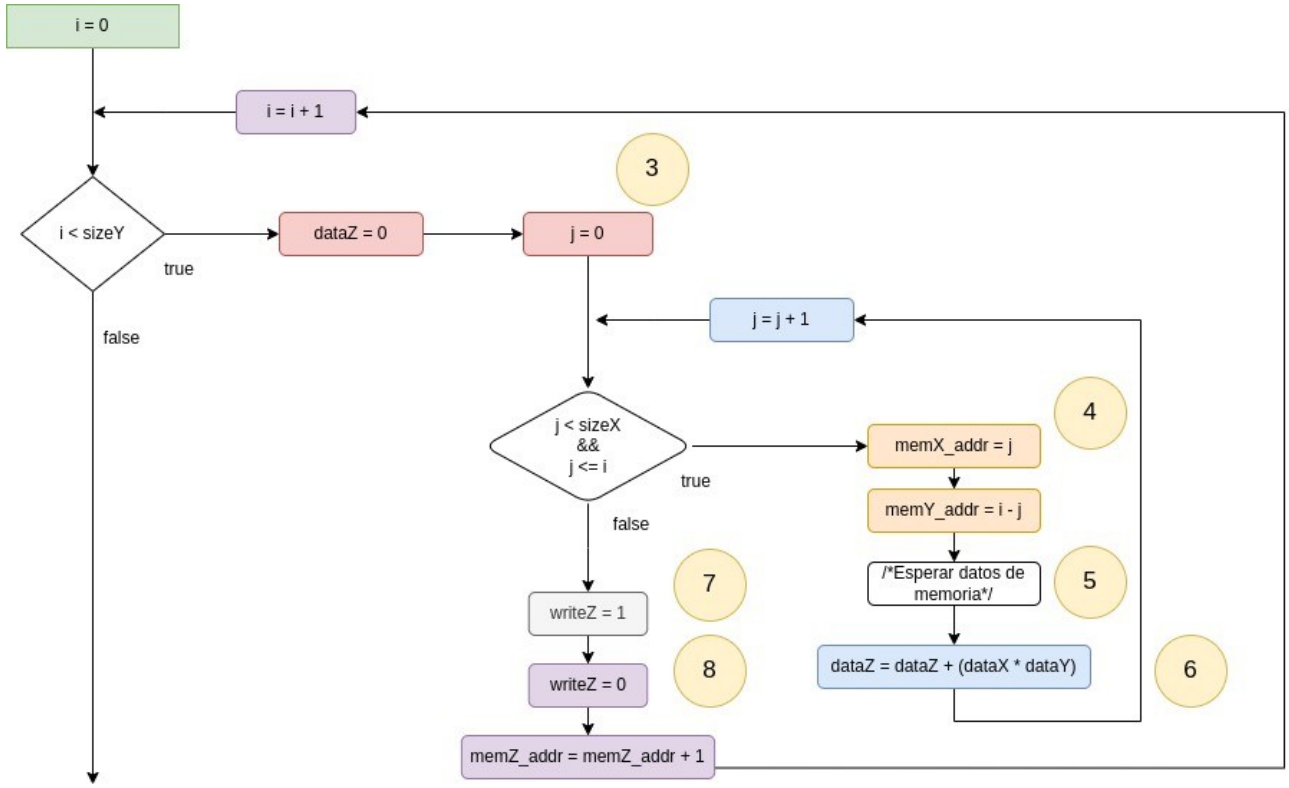

Figure 7: Diagrama ASM de la primera parte del algoritmo de convolución

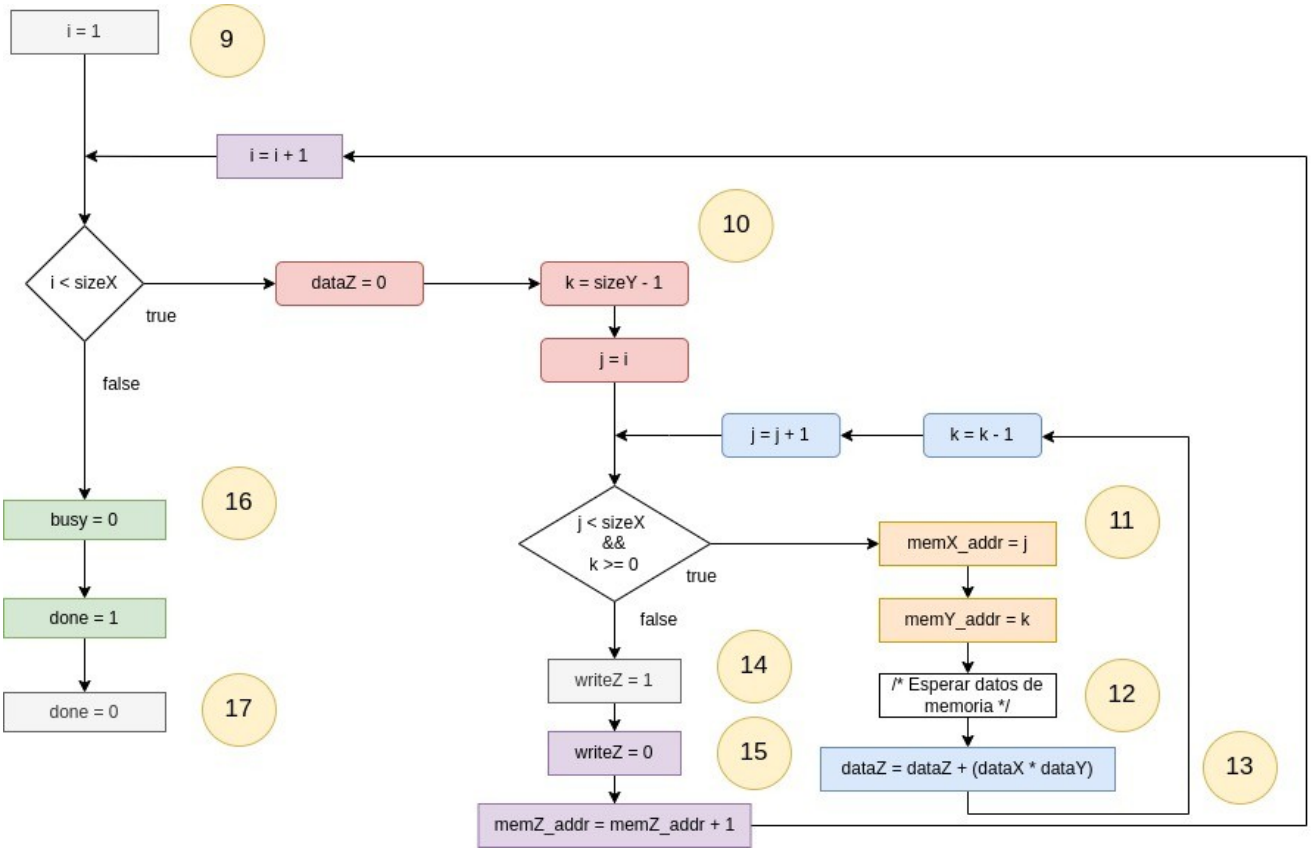

Figure 8: Diagrama ASM de la segunda parte del algoritmo de convolución

Datapath

Las figuras 9 y 11 representan el datapath a partir de los estados del diagrama ASM de las figura 6. Este no es el datapath final, debido a que se pueden optimizar algunos bloques.

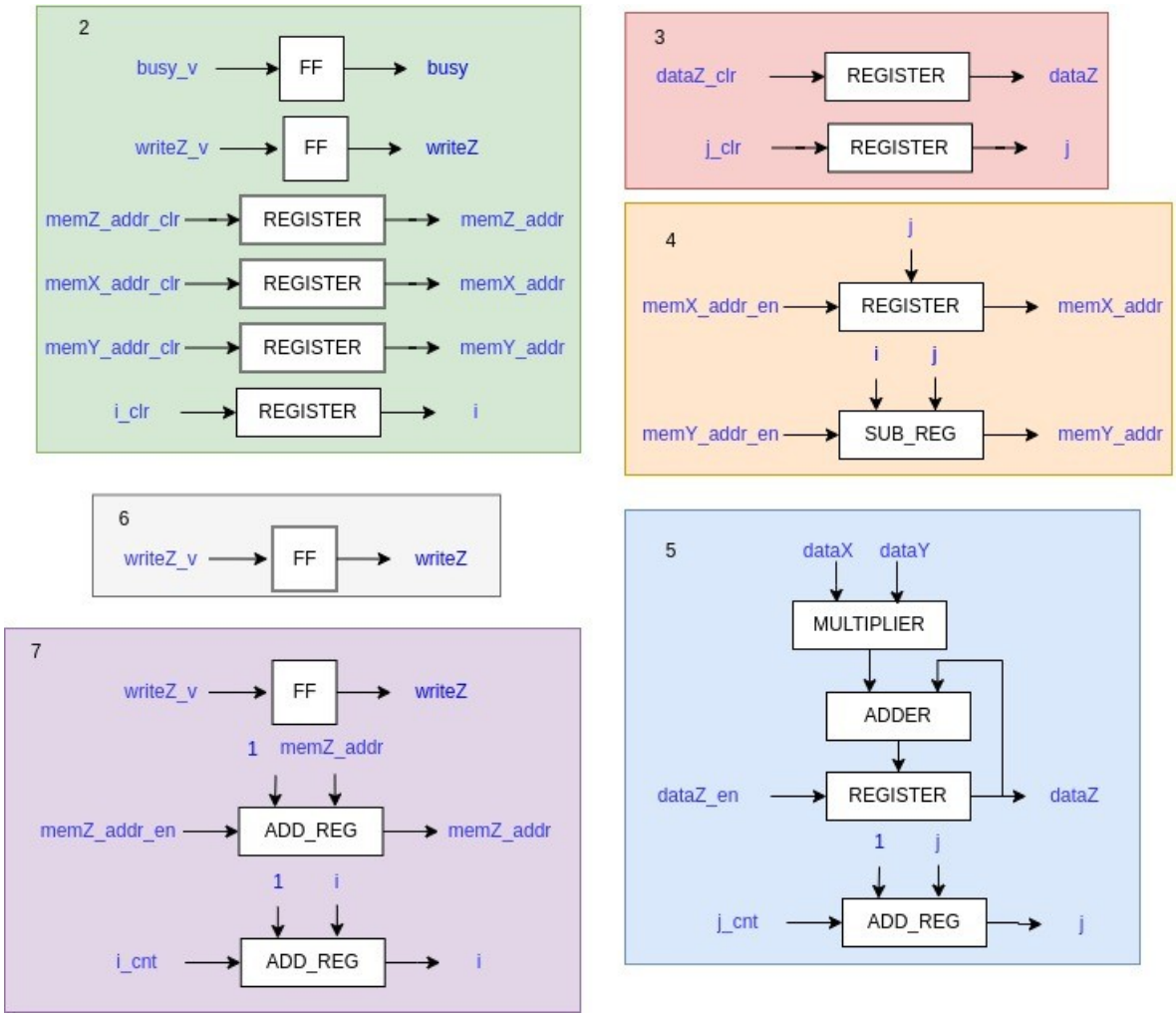

Figure 9: Datapath de los estados 2 al 7, que representan la primera parte del algoritmo

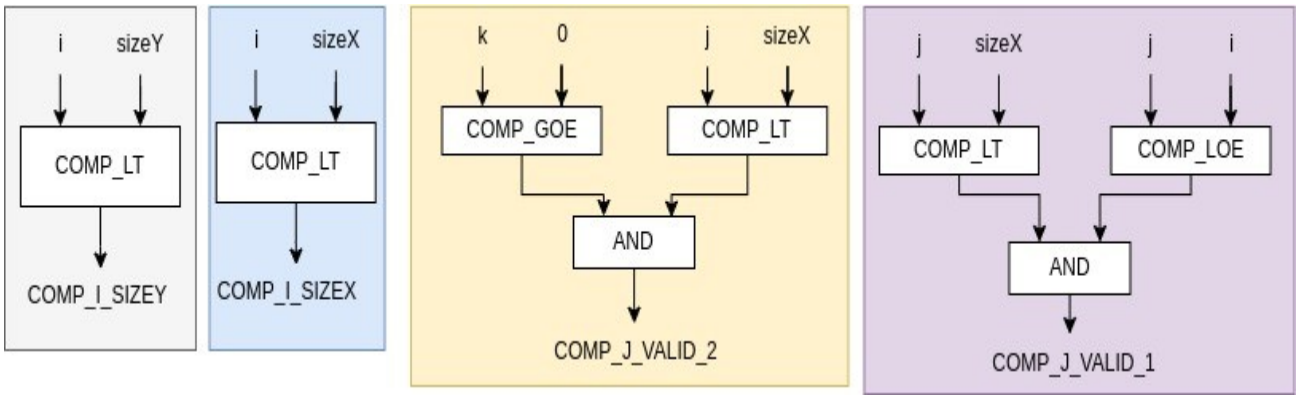

Figure 10: Bloques de comparación dentro del algoritmo

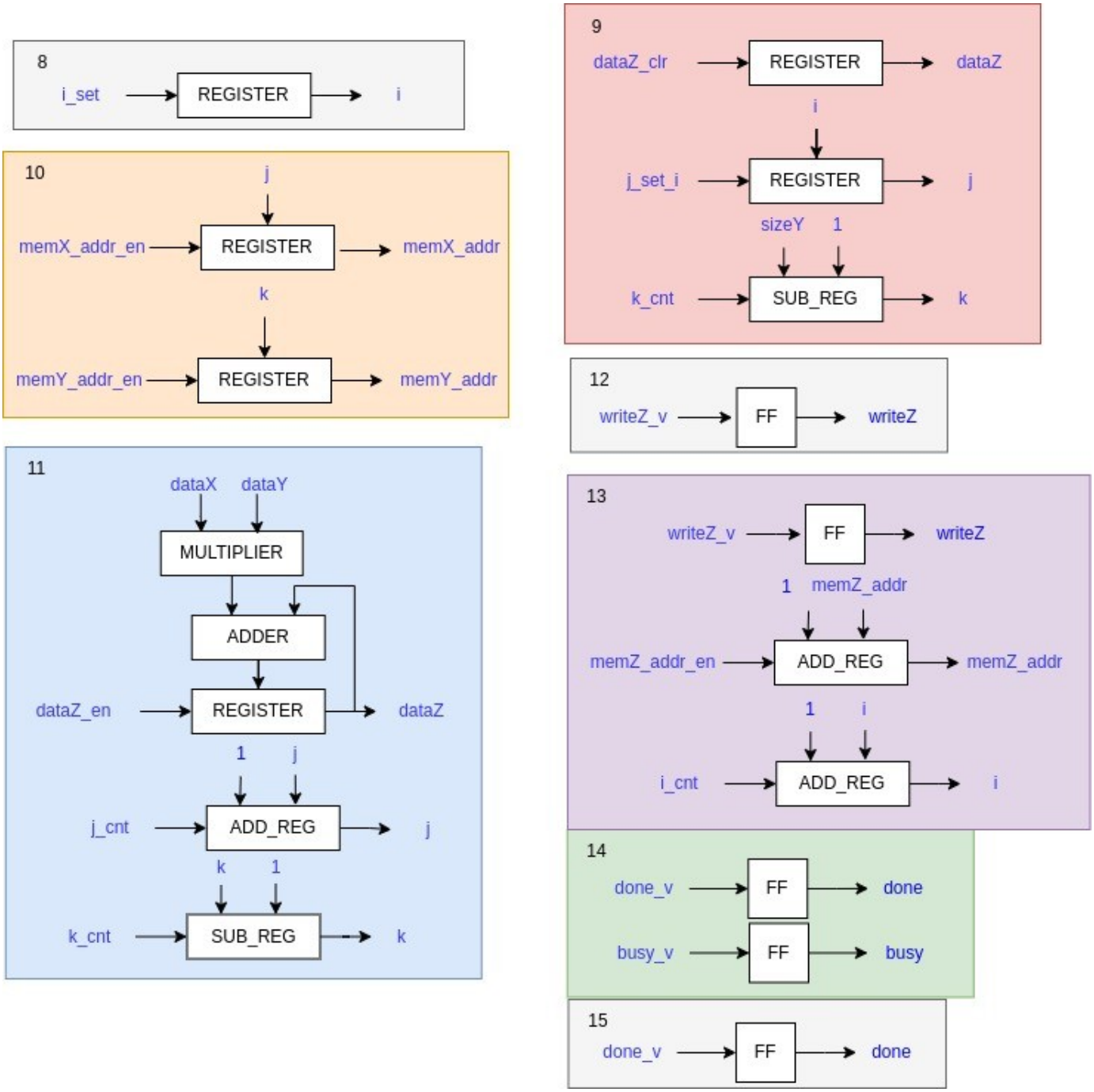

Figure 11: Datapath de los estados 8 al 15, que representan la segunda parte del algoritmo

Optimización del datapath

Tras analizar los bloques de estado y su datapath, podemos ver redundancias y aspectos que podrían ser mejorados. Por ejemplo, en la figura 12 observamos la optimización para el registro memZ_addr, que se compartía en los estados 13, 7 y 2. Este nuevo bloque “mejorado” nos facilita analizar el datapath, puesto que ahora existe un único bloque que lo modifica.

Otro ejemplo de optimización se puede tomar con la figura 13, donde el registro memY_addr es cargado con un valor correspondiente al selector del multiplexor cuando la señal de control memY_addr_en esté activa. Del mismo modo, cuenta con una señal para limpiar el registro llamada memY_addr_clr.

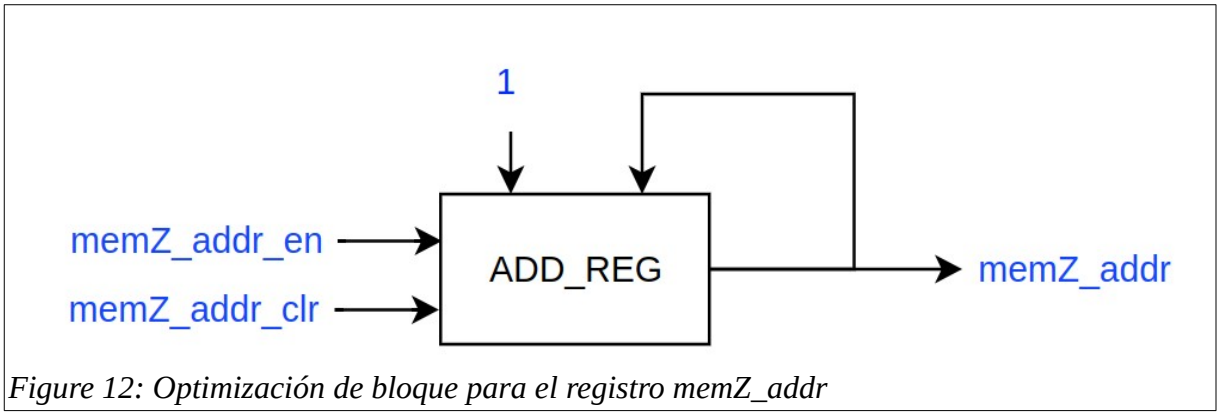

Figure 12: Optimización de bloque para el registro memZ_addr

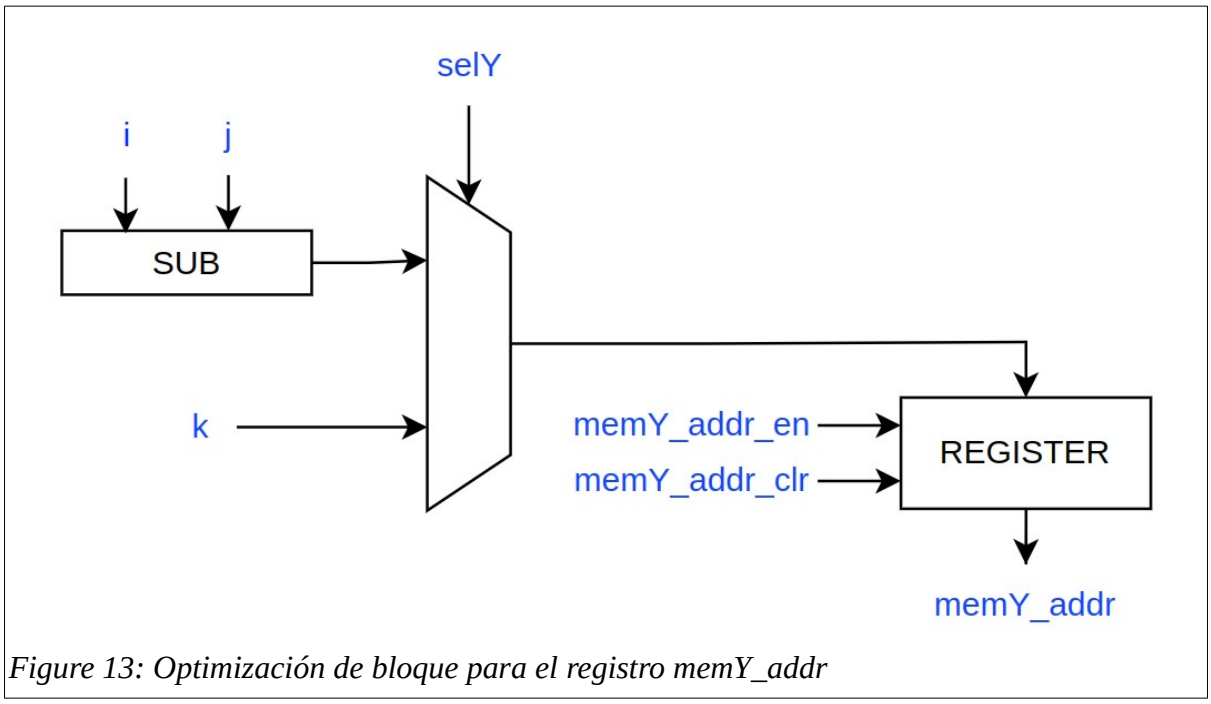

Figure 13: Optimización de bloque para el registro memY_addr

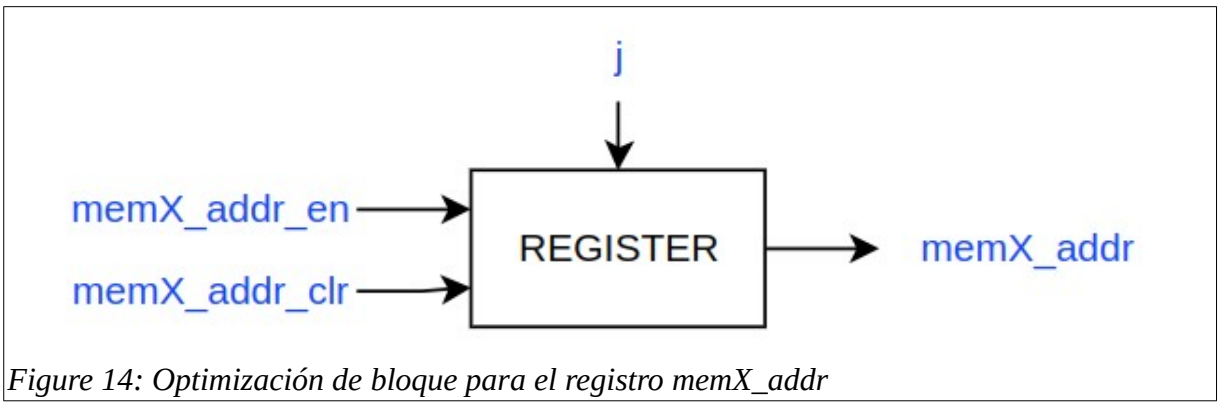

Figure 14: Optimización de bloque para el registro memX_addr

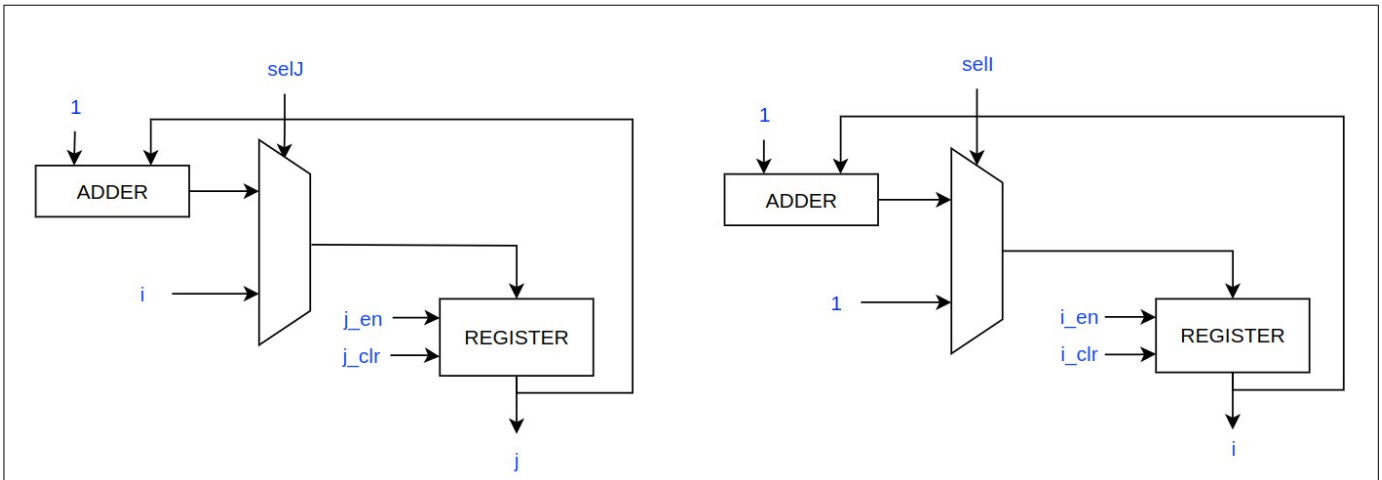

Figure 15: Optimización de bloque para los registros j e i

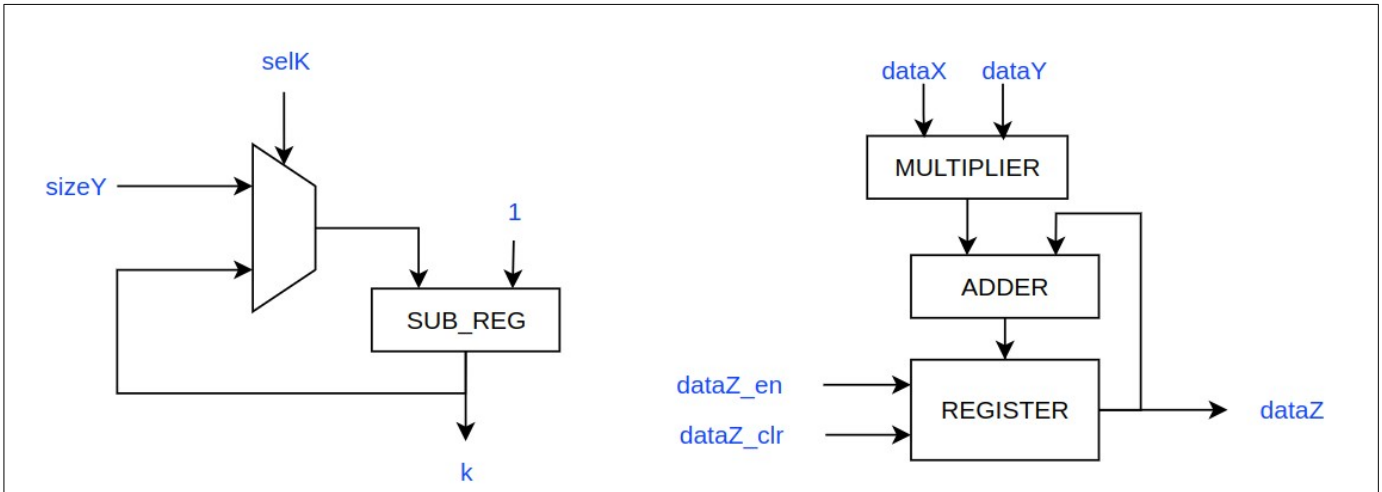

Figure 16: Optimización de bloque para los registros k y dataZ

Es importante destacar que los registros *j* e *i* (figura 15) son casi idénticos, por lo que se podría inferir un módulo común, ya que se comportan como contadores capaces de ser cargados con un valor externo, refiriéndose a los valores de *i* y *1* en cada bloque respectivamente.

El datapath desarrollado a lo largo del tema se observa en la figura 17. Existen bloques que requieren de registros provenientes de otros bloques, por lo que ocurre unan interconexión entre estos. Las conexiones de bloques del datapath se pueden explicar de la siguiente forma: el bloque del registro *j* de depende del registro *i*, que a su vez conecta con el bloque del registro memX_addr y el bloque de memY_addr. Y por último el bloque de memY_addr depende de los registros *i, j* y *k*.

Cada uno de los bloque previamente optimizados están coloreados de un color que los diferencia de los circundantes a él.

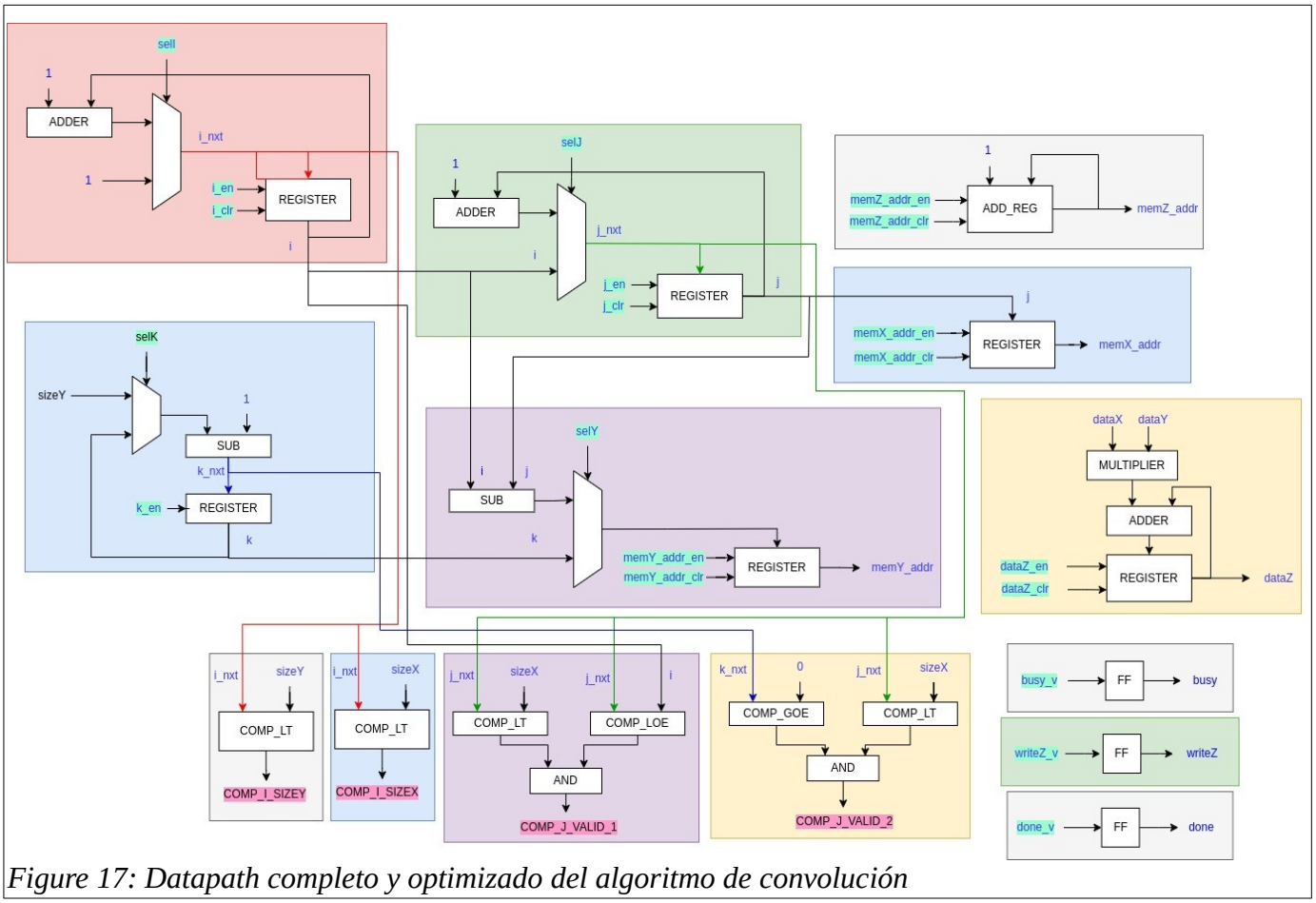

Figure 17: Datapath completo y optimizado del algoritmo de convolución

Máquina de estados

Los estados de la máquina de estados pueden observarse en la figura 17 y 18, donde las señales remarcadas de verde, representan señales de control las cuales se requieren en los bloques para determinar la transmisión de datos correctamente, infiriendo así la operación que debe realizar un multiplexor, o estableciendo un momento para limpiar o establecer un valor en un registro. Las señales remarcadas de rojo, representan señales de control de salida de varios comparadores. A diferencia de las señales remarcadas de verde, las señales remarcadas de rojo no infieren el comportamiento de un bloque, sino que sirven para controlar la máquina de estados, que obtiene su siguiente estado a partir de estas señales de control.

A partir del datapath y de definición de las señales de control, se puede definir la tabla de señales de control para cada uno de los estados posibles (tabla 2). Las señales de control del lado izquierdo significan que son necesarias de cumplirse para ingresar o llegar a ese estado. Si no hay alguna señal de control para ingresar al estado, a éste solo se podrá ingresar a partir del termino del estado anterior a éste. Del lado derecho de la tabla de señales, se encuentran las señales que se producirán en ese estado.

Se agregaron dos estados auxiliares, de manera que su función es esperar a que se lean los datos dataX y dataY de las memorias X y Y respectivamente, siendo los estados 5 y 12 los estados en cuestión.

FSM		
State	**Inputs/Value**	**Outputs/Value**
1	START/0 (S1)	
2	START/1 (S1)	busy_v/1 writeZ/0 memZ_addr_clr/1 memX_addr_clr/1 memY_addr_clr/1 i_clr/1
3	COMP_I_SIZEY/1 (S2) COMP_I_SIZEY/1 (S8)	dataZ_clr/1 j_clr/1
4	COMP_J_VALID_1/1 (S3) COMP_J_VALID_1/1 (S6)	memX_addr_en/1 memY_addr_en/1 SelY/0
5		
6		dataZ_en/1 j_en/1 SelJ/0
7	COMP_J_VALID_1/0 (S3) COMP_J_VALID_1/0 (S6)	writeZ_v/1
8		writeZ_v/0 memZ_addr_en/1 i_en/1 SelI/0
9	COMP_I_SIZEY/0 (S2) COMP_I_SIZEY/0 (S8)	i_en/1 SelI/1
10	COMP_I_SIZEX/1 (S9) COMP_I_SIZEX/1 (S15)	dataZ_clr/1 k_en/1 SelK/0 j_en/1 SelJ/1
11	COMP_J_VALID_2/1 (S10) COMP_J_VALID_2/1 (S13)	memX_addr_en/1 memY_addr_en/1 SelY/1
12		
13		dataZ_en/1 k_en/1 SelK/1 j_en/1 SelJ/0
14	COMP_J_VALID_2/0 (S10) COMP_J_VALID_2/0 (S13)	writeZ_v/1
15		writeZ_v/0 memZ_addr_en/1 i_en/1 SelI/0
16	COMP_I_SIZEX/0 (S9) COMP_I_SIZEX/0 (S15)	busy_v/0 done_v/1
17		done_v/0

Table 2. Tabla de la FSM del convolucionador

En la figura 18 se muestra el diagrama de estados de la máquina de estados del convolucionador. Se observan de color azul y letras grandes las señales de control que debe de recibir la FSM para cambiar al siguiente estado, y existen cuadros grises con las salidas que deberá reflejar la máquina para cada uno de los estados.

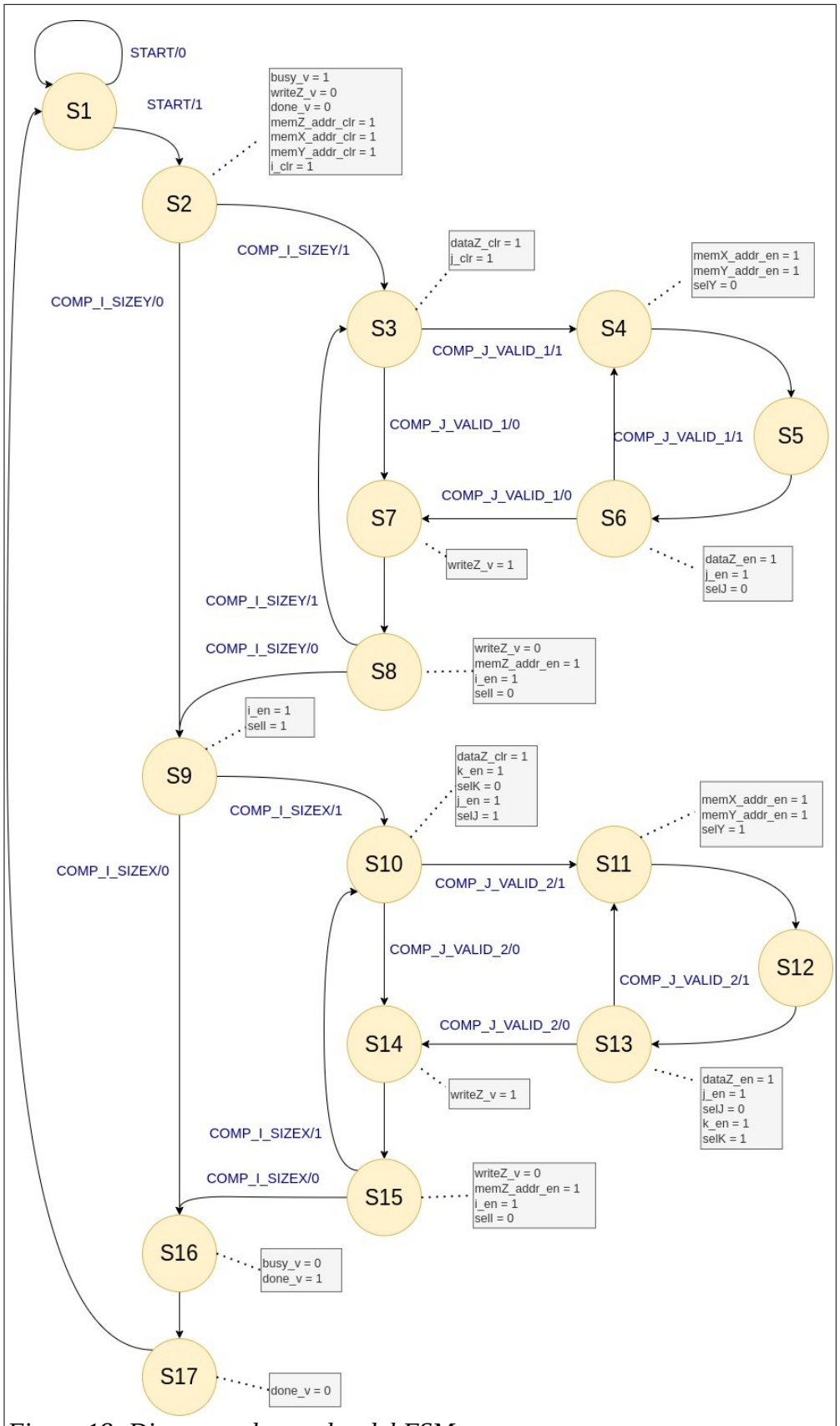

Figure 18: Diagrama de estados del FSM

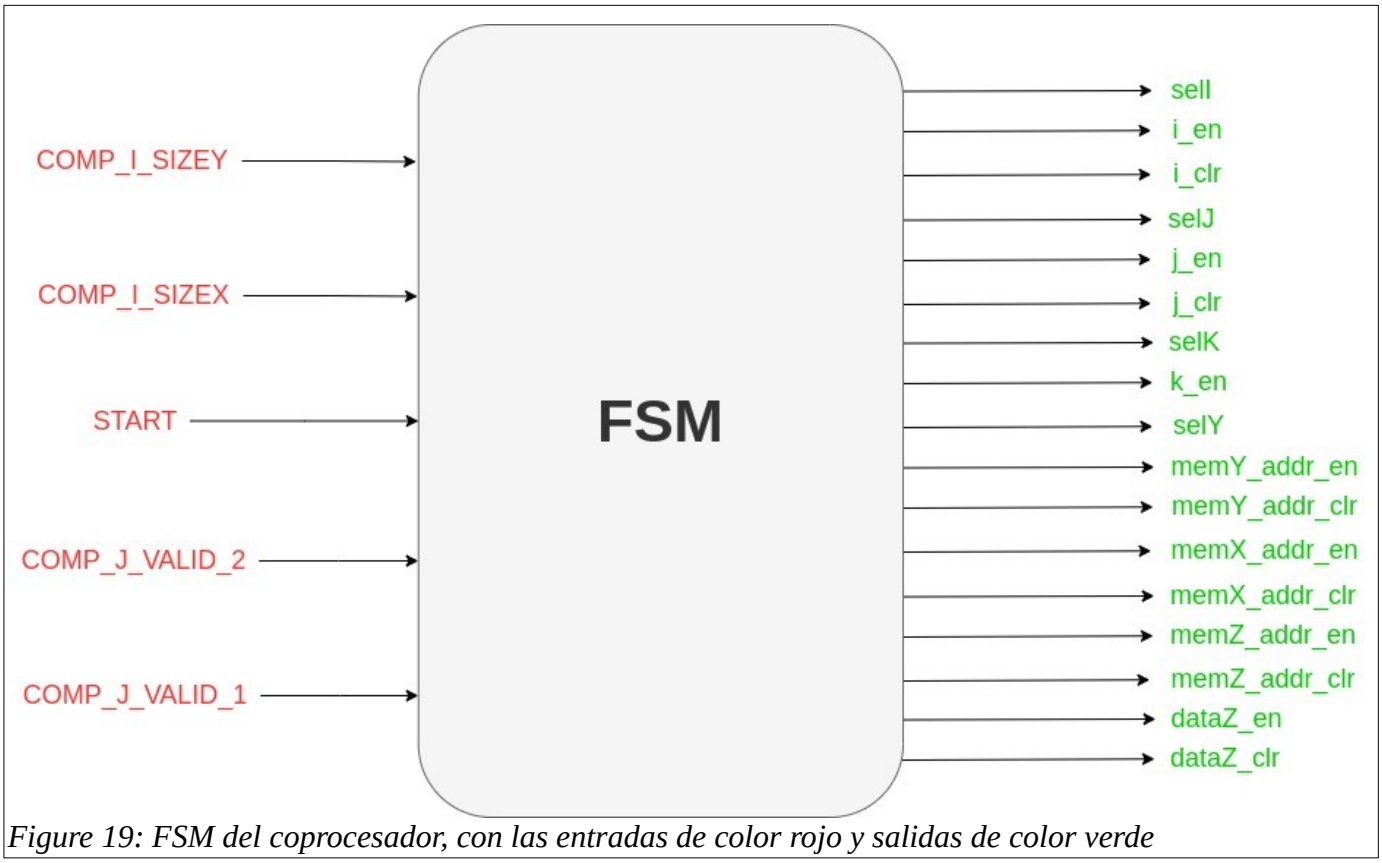

Figure 19: FSM del coprocesador, con las entradas de color rojo y salidas de color verde

Resultados de simulación

Se creo una cama de prueba para verificar el funcionamiento del convolucionador. Las instancias de módulos utilizadas en la cada de prueba fueron las siguientes:

- convolucionador (DUT)
- memX (memoria RAM síncrona)
- memY (memoria RAM síncrona)
- memZ (memoria RAM síncrona)

Las memorias X e Y, fueron cargadas con valores iniciales a través de archivos de textos precargados en el sistema. Los archivos de texto contenían 32 valores de 8 bits que fueron cargados hacia las memorias. Estos archivos de texto fueron generados a partir de un código en C que generaba números al azar y los escribía en los archivos.

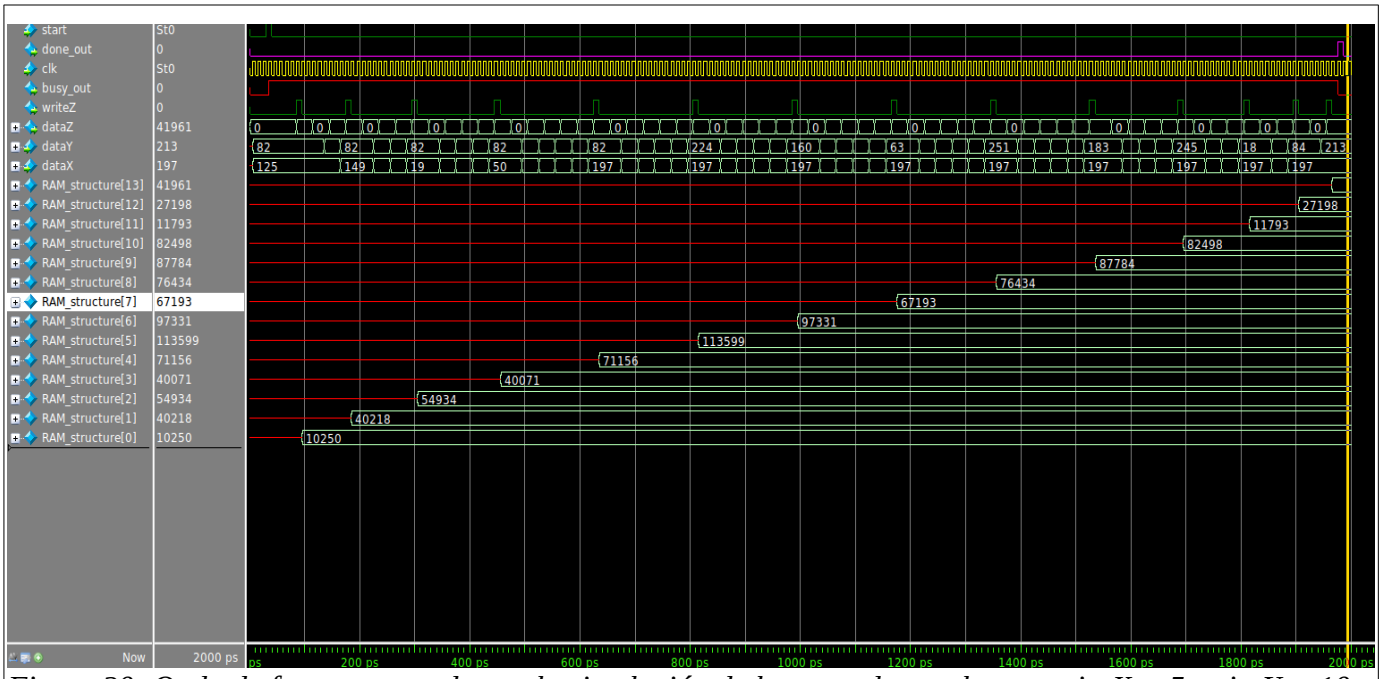

Figure 20: Onda de forma generada por la simulación de la cama de prueba, con sizeX = 5 y sizeY = 10

En la forma de onda de la figura 20 se encuentra la señal de start a mero arriba de las demás; de color rosa se ve la señal de done; de rojo la señal de busy y de amarillo la señal de reloj. Por último tenemos las 14 primeras localidades de la memoria Z, donde se puede notar como se escribe el resultado de la convolución en cada una de ellas según corresponda.

Dentro de la simulación, la seña de start se establece activa a partir del tiempo: 35 ps. El tiempo en el que done es puesta activa es: 1975 ps. Si el período de la señal de reloj es de 10 ps, podemos calcular el número de ciclos que tuvieron que pasar entre la una y la otra a partir de $\frac{1975-35}{10}=194$.

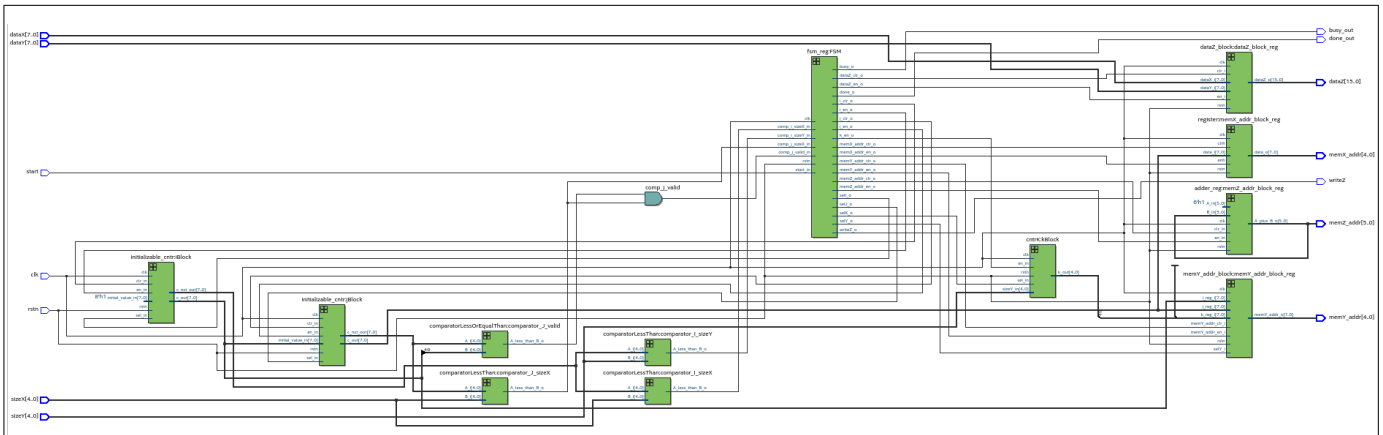

Figure 21: Top level generado por quartus

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	69
2		
3	Combinational ALUT usage for logic	110
1	-- 7 input functions	0
2	-- 6 input functions	27
3	-- 5 input functions	9
4	-- 4 input functions	11
5	-- <=3 input functions	63
4		
5	Dedicated logic registers	74
6		
7	I/O pins	64
8		
9	Total DSP Blocks	1
10		
11	Maximum fan-out node	clk~input
12	Maximum fan-out	74
13	Total fan-out	846
14	Average fan-out	2.70

Figure 22: Área. Recursos utilizados por el diseño

	Fmax	Restricted Fmax	Clock Name	Note
1	205.04 MHz	205.04 MHz	clk	

Figure 23: Frecuencia máxima a la cual puede operar el diseño

194 ciclos de reloj
para terminar el proceso de convolución

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

void writeToFile(FILE* archivo, const uint8_t* values);

int main (void){

   srand(time(NULL));
   uint8_t  valuesX[32] = {};
   uint8_t  valuesY[32] = {};

   for (int i=0; i<32; i++){
      valuesX[i] = rand() % 256;
      valuesY[i] = rand() % 256;
   }

   FILE* archivoX = NULL;
   FILE* archivoY = NULL;

   archivoX = fopen("memX_values.txt", "w");
   writeToFile(archivoX, valuesX);
   fclose(archivoX);

   archivoY = fopen("memY_values.txt", "w");
   writeToFile(archivoY, valuesY);
   fclose(archivoY);

   return 0;
}

void writeToFile(FILE* archivo, const uint8_t* values){
   for(int i=0; i<32; i++){
      fprintf(archivo, "%02X\n", values[i]);
   }
}
```

Figure 24: Código en C implementado para la generación de archivos con valores aleatorios