Metodología de Diseño de Sistemas On-Chip

Desarrollo de un coprocesador de convolución

Pablo Daniel Pérez Montes

TAE Sistemas Embebidos 2024
Cinvestav Unidad Guadalajara

Descripción del problema

Para este proyecto se planteo el desarrollo de un Core que realiza la convolución discreta de dos señales. A continuación, se puede apreciar el diagrama de caja negra que se plantea:

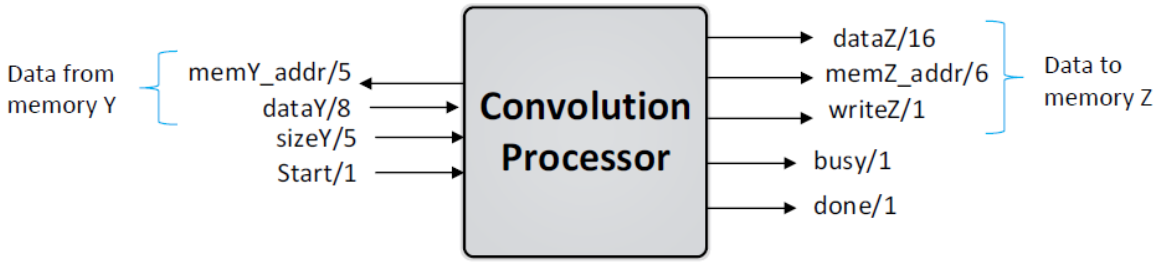

Ilustración 1: Diagrama de caja negra

El problema nos limita a trabajar con datos con un tamaño de palabra de 1 byte, por lo cual *dataY* como entrada será un bus de 8 bits.

Se establece que la señal Y o kernel, será una señal proveniente de una memoria RAM, por lo cual se requiere que la misma pueda ser accedida pidiendo el dato en su respectiva dirección, por lo que se requiere la señal *memY_addr* con un bus de 5 bits.

El usuario debe proporcionar el tamaño del vector con anticipación para configurar la operación, por lo que se pide que entregue el tamaño en la señal *sizeY* de tamaño de palabra de 5 bits.

El core ha de funcionar una vez que la señal *start* tenga el valor lógico alto. Entonces el procesador comenzara a calcular el vector Z realizando la convolución con los datos que extraiga de la memoria RAM externa y una ROM interna con un vector de tamaño y de datos no especificados.

Los datos son guardados en una memoria RAM externa, la maquina escribira el resultado constantemente, sin embargo hasta que la señal *done* no tenga el valor lógico alto, no se considerara como “bueno” el resultado. Despues de un ciclo de reloj, la señal *done* vuelve a estado lógico bajo asi como la señal *busy*

Algoritmo de la convolución

La convolución discreta se define por la siguiente sumatoria:

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k]$$

Ilustración 2: la operación de convolución discreta

El resultado de y[n] es una señal de tamaño M+N-1, donde M es la longitud de la señal H y N el tamaño de la señal X (en el diagrama de bloques, X y Y respectivamente).

Para atacar el problema hay que entender que la sumatoria tiene el tamaño M+N-1 que se menciona arriba, por lo que lo conveniente es tomar el valor de la longitud de ambas señales para que el core pueda trabajar. El algoritmo propuesto es el siguiente:

```
while start = 0
end while
busy = 1
done = 0
write = 0
size_temp = sizeY
sizeZ = size_temp + sizeH - 1
i = 0
while i < sizeZ

   temp_z = 0 /
   j = 0
   while j < sizeH
      temp_h = h[j]
      k = 0
      while k < size_temp
         temp_y = dataY [k]

         if (j+k) == i
            result = temp_z + (temp_h * temp_y)
            temp_z = result
            write = 1
            memZ_addr = i
            dataY = temp_z
            write = 0
         end if

         k = k+1
      end while

      j = j+1
   end while

   i = i+1
end while
done = 1
done = 0

goto 1
```

Ilustración 3: algoritmo en pseudo codigo

En el algoritmo se propone que la maquina debe esperar a la señal start a que sea de nivel lógico alto

. Una vez siendo esto correcto se procede con la toma del valor que este en el bus de *sizeY* que será necesario para calcular el numero de iteraciones que se deben realizar para poder guardar el valor del resultado de la convolución. Para entonces la señal *busy* ha sido levantada avisando al dispositivo maestro que se esta trabajando y no se puede atender mas operaciones de momento.

El valor de *sizeY* es guardado en un registro nombrado *size_temp* con el valor de este registro se realiza la operación *sizeH = sizeH + size_temp – 1* donde *sizeH* es un valor almacenado en un flop interno de tamaño de bits de 5 y *sizeH* mide 6 bits. Entonces se inicia el contador i en cero lo que da comienzo a la iteración de las operaciones.

Para el contador i desde el valor 0 hasta el valor de *sizeZ-1* primero se borra cualquier contenido de temp_z, un registro que tiene tamaño 16 bits. Este registro almacena la operación de convolución. Después se establece el valor de 0 al contador j, que va a contar del 0 al tamaño *sizeH-1* entonces se carga el valor de temp_h al valor que este almacenado en la memoria ROM en la dirección *i*, solo entonces cuando se hallan cargado se realiza la operación de la convolución, y asi hasta que se terminen las iteraciones. Los resultados se almacenan en la memoria externa en un momento de las iteraciones donde los contadores j y k en suma sean iguales a i, recordemos que los contadores se conectan a las direcciones de la memoria, por lo que justo las iteraciones correspondientes a $j + k = i$ o se puede entender como $sizeY + sizeH = sizeZ$.

En esta tabla ejemplifica el procedimiento del calculo, supongamos un vector de tamaño 4 y otro de tamaño 3:

$$y = \{1,2,3,4\} \quad h = \{1,2,3\}$$

El vector Z tendrá una longitud 6. Para z[0] los índices de y y h deben sumar cero, para el índice 1, deben sumar 1 tanto j y k.

0	1	2	3	4	5
(0,0)	(0,1)	(0,2)	(0,3)	(1,3)	(3,2)
	(1,0)	(1,1)	(1,2)	(2,2)	
		(2,0)	(2,1)		

Ilustración 4: Tabla de explicacion de localizacion de iteraciones

En este apartado se expone el código en C del modelo del algoritmo que recién se explicó:

```c
#include <stdio.h>

int main (void){
    int temp_z; //register for result of z vector
    int temp_h, temp_y; //registers for values of dataY and dataH from memories, repectively
    int dataZ;  // net name
    int resultado;  //register to save operation

    int start; // fsm input
    int busy,done,write; //fsm output

    int addrY,addrH,addrZ;  //net names
    int sizeH,sizeY; //sizes of kernel Y and vector H
    int size_temp; //size temp is a register that stores the input sizeY
    int i,j,k; // iteration constants

    sizeH = 7; //size H is a hardwired value
    sizeY = 4;  //sizeY comes from imput
    size_temp = sizeY;  //size_temp stores sizeY after 1 clk cycle
    int sizeZ = (sizeH+size_temp-1); // sizeZ will store the operation of sizeH + size_temp - 1
    int z[sizeZ];// = {0};

    int h[7] = {1,2,3,4,5,6,7}; //simulates a vector stored in a ROM
    int y[4] = {4,3,2,1};   //simulate a vector stored in a RAM

    //data flow
    //while start = 0
    start = 0;   //wait the input

    //end while, then
    done = 0;
    start = 1;
    busy = 1;
    addrY = 0; addrH = 0;

    for(i = 0; i < sizeZ; i++){
        temp_z = 0;

        for(j = 0; j < sizeH; j++, k++){
            addrH = j; //net named addrH drives the value of j
            temp_h = h[addrH];

           for(k = 0; k < size_temp; k++){
                addrY = k; //net named addrY (memY_addr)
                temp_y = y[addrY];

                //if the sum of k + j equals to the current data of Z
                if( (j+k) == i ){
                    resultado = temp_z + (temp_h * temp_y);
                    temp_z = resultado;
                    //begin the writing to mem Z
                    write = 1;
                    addrZ = i;
                    z[addrZ] = temp_z;
```

```
57                      //printf("y[%i] = %i * h[%i] =
58 %i\n",k,temp_y,j,temp_h);
59                      write = 0;
60                  }
61              }
62          }
63
64      }
65
66      //show the result output in the Z memory
67      for(int i = 0; i < (sizeH + sizeY -1); i++){
68         printf("[%i] = {%i}\n", i,z[i]);
69      }

        return 0;
    }
```

Ilustración 5: Codigo en C

Como resultado, en terminal apreciamos el siguiente resultado

```
[0] = {4}
[1] = {11}
[2] = {20}
[3] = {30}
[4] = {40}
[5] = {50}
[6] = {60}
[7] = {38}
[8] = {20}
[9] = {7}

Process returned 0 (0x0)   execution time : 0.577 s
```

Ilustración 6: Terminal de C

Realizando en Matlab, utilizando los siguientes comandos obtenemos en terminal el resultado siguiente:

```
>> x = [1 2 3 4 5 6 7]

x =

     1     2     3     4     5     6     7

>> y = [4 3 2 1]

y =

     4     3     2     1

>> z = conv(x,y)

z =

     4    11    20    30    40    50    60    38    20     7
```

Ilustración 7: Terminal de Matlab

Pudiendo comprobar el resultado del algoritmo con una herramienta de calculo como Matlab se puede proceder con el desarrollo del diagrama ASM con expresiones RTL.

Nota: en el código de C, a partir de la línea 66 se encuentra una línea de código simplemente ilustrativa para poder ver el resultado de la operación, por lo mismo el arreglo dataY no es un elemento que se considera en el datapath siguiente, asi como la memoria dataY no es un elemento propio del core, si no un elemeno externo.

Diagrama ASM

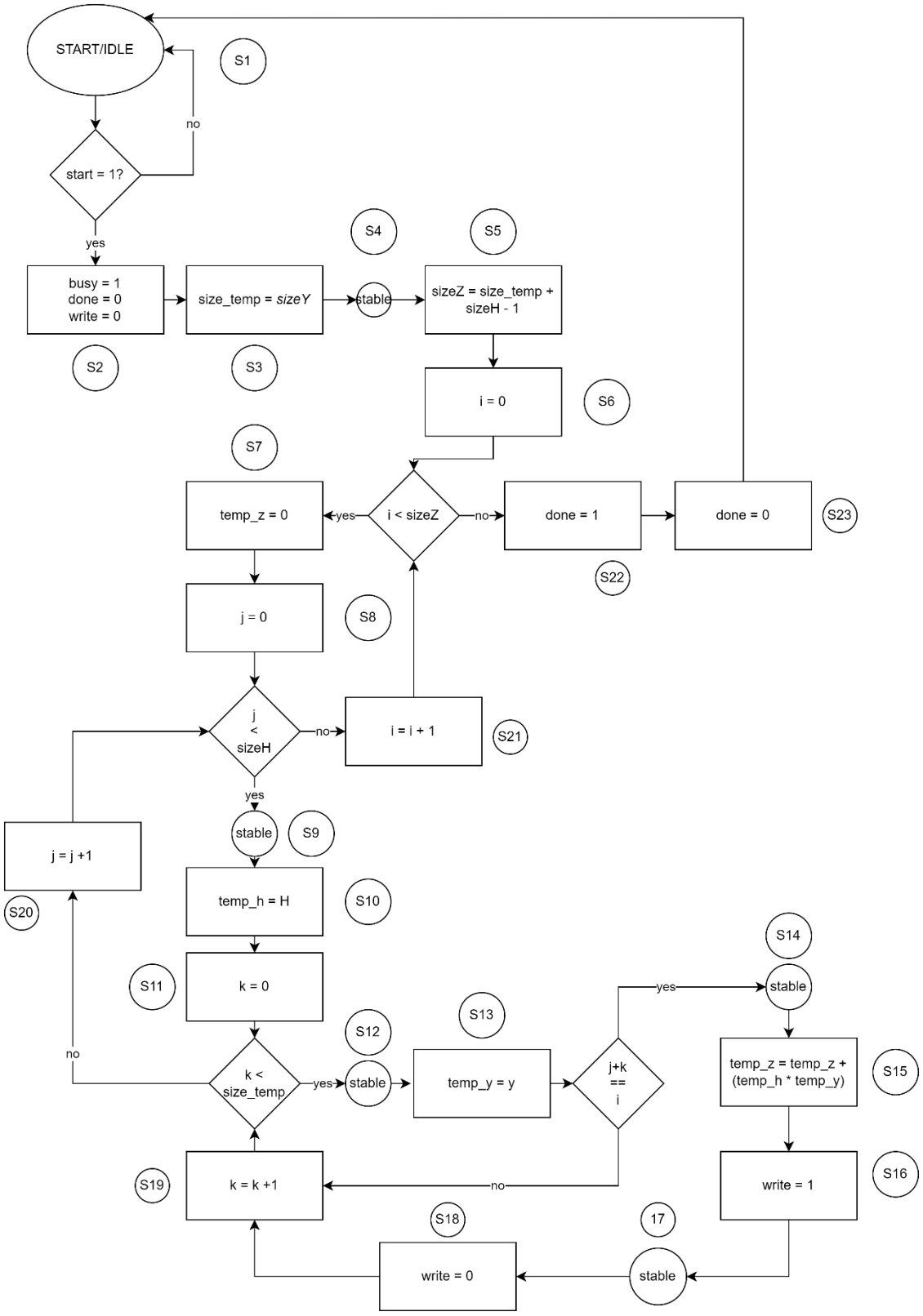

Ilustración 8: Diagrama ASM con expresiones RTL

Diagrama Datapath y diagrama de estados

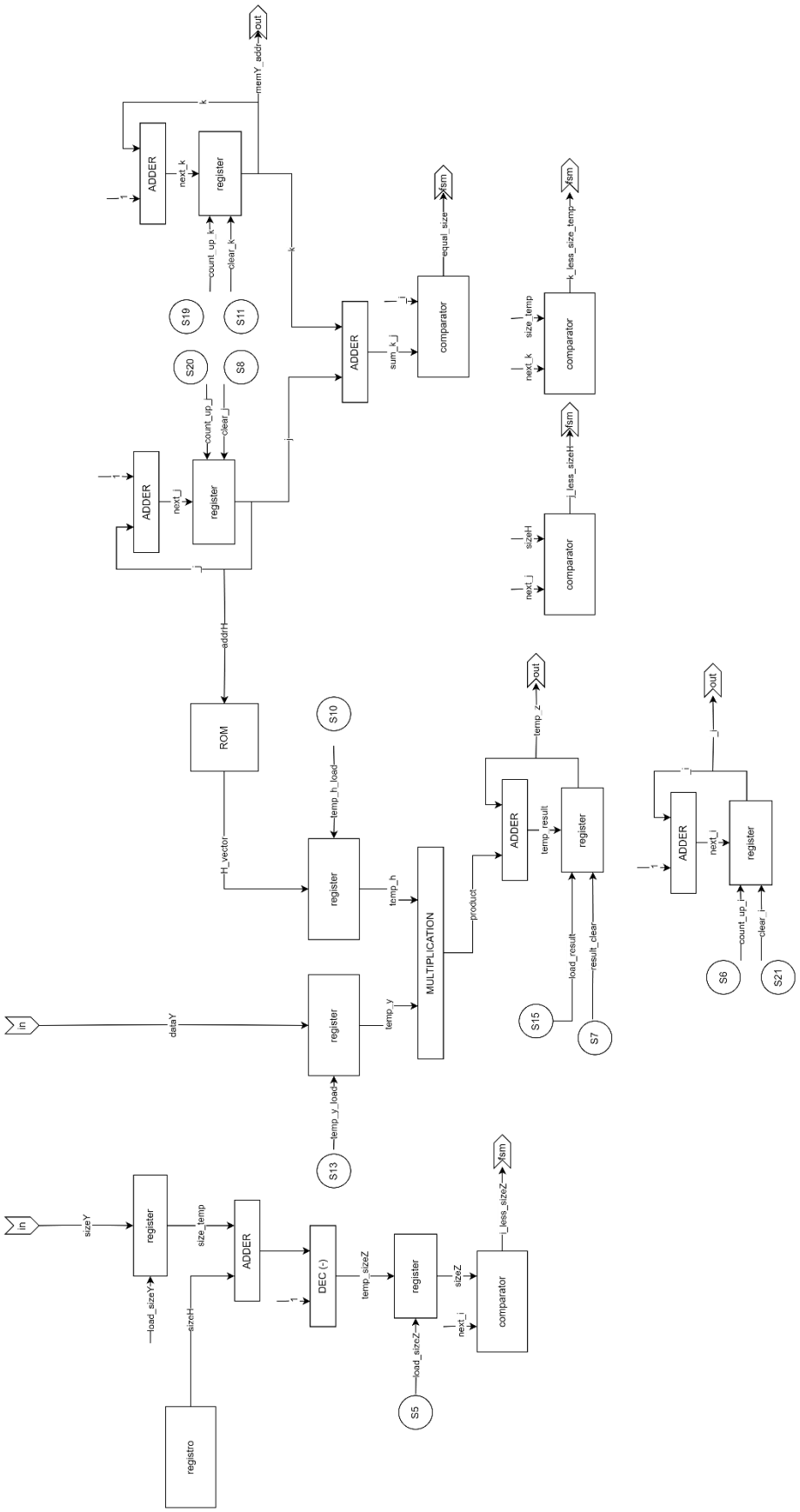

Ilustración 9: data path del diagrama ASM

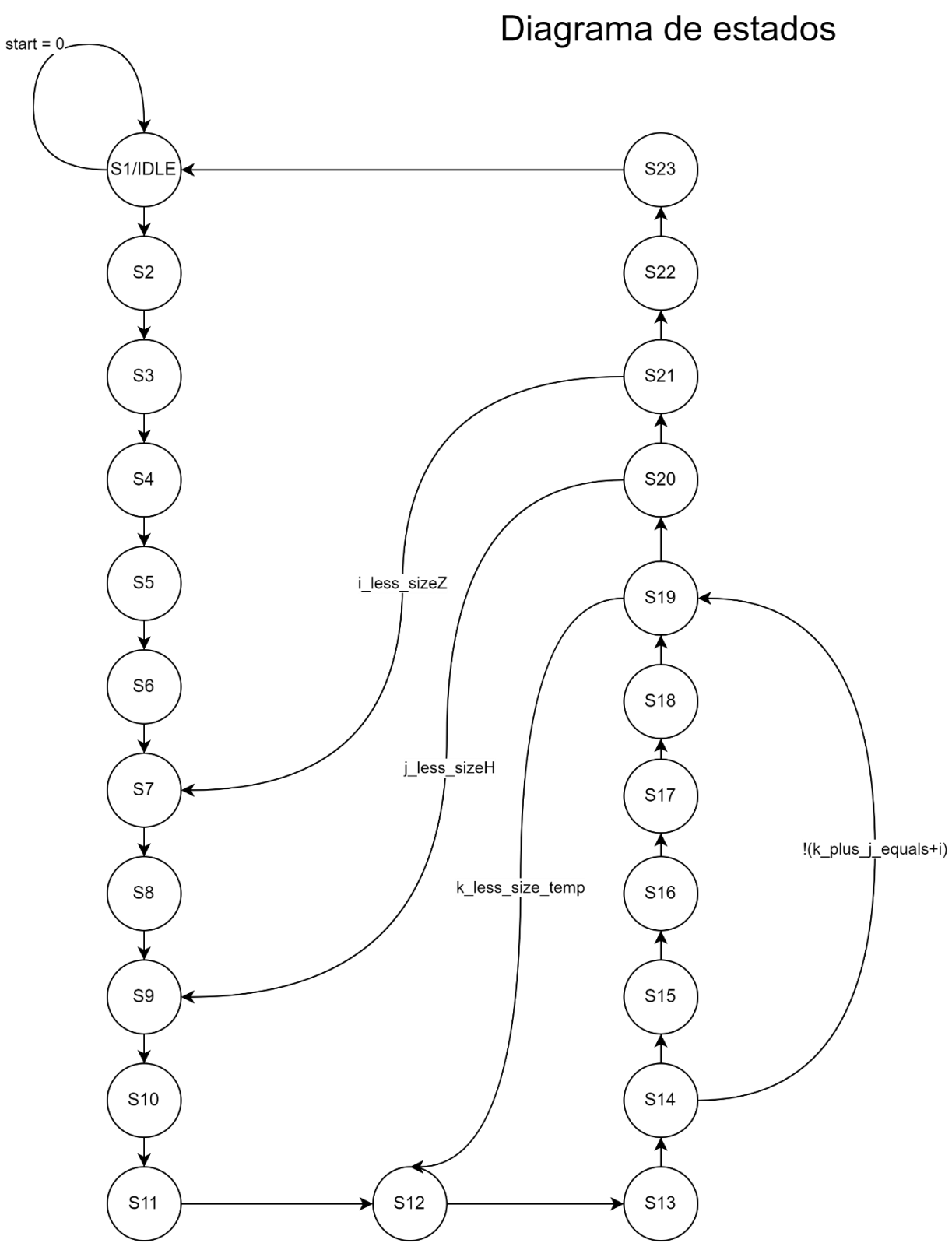

Ilustración 10: diagrama de estados

Explicación de los diagramas

Entrando en detalles de ambos diagramas, en el primero se llega a los bloques optimos a partir de las expresiones RTL. En el datapath se señalan las señales de entrada, señales de salida y las señales de entrada a la máquina de estados. Para acceder a los elementos del vector, se decidió dividir las expresiones de arreglos en C en dos partes, un registro para almacenar aquel valor que se accede de la memoria externa o la ROM interna del core. La segunda expresión se describe como un cable conectado entre el bus de address para las memorias y los registros de los contadores, mas específicamente las señales _i, _j y _k están directamente conectadas a los buses de dirección de memoria dataY, dataH y dataZ, respectivamente.

```
addrY = k; //net named addrY (memY_addr)
temp_y = y[addrY];
```

Ilustración 11: ilustracion de una expresion incongruente en el dominio RTL

El diagrama de estados se basa en el diagrama ASM, donde se cabe destacar que los estados se optimizaron considerando los delays de la transferencia entre registros, estos estados son llamados “stable” en el diagrama ASM, junto a cada expresión RTL se agrego un dibujo que indica el estado al que corresponde. En el diagrama de estados hay un total de 4 ramas derivadas de las decisiones que se encuentran en el diagrama ASM de la figura.

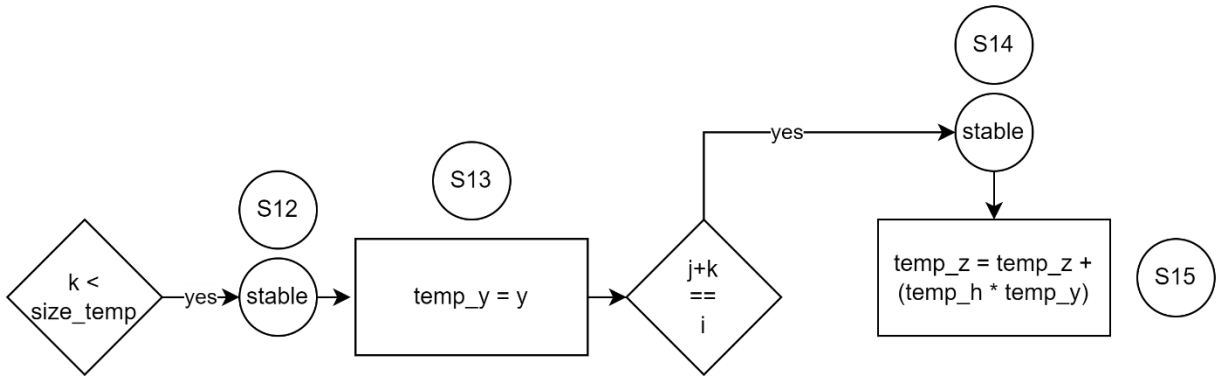

Ilustración 12: estados estables en el diagrama ASM

Bloques optimos

El datapath consiste principalmente en registros, circuitos aritméticos como sumadores, restadores y comparadores (*menor que* e *igual que*). No se decidió agregar un bloque optimo propio para los contadores, los cuales se cuentan en el repertorio de código de Verilog/SystemVerilog, esto es por que se requieren las señales next_i, next_j y next_k para las comparaciones. Esto justamente ahorra al diseño el uso de registros, los cuales proporcionarían más retrasos en la transferencia de los datos. Otro bloque optimo ahorrado fue la decisión de convertir en cables directos al bus a las salidas ya registradas de los bloques de contadores para acceder a las

memorias externas y la memoria ROM interna. Por último, la salida dataY es un puente directo entre la salida del registro temp_z, decisión tomada por los argumentos mencionados.

Resultados y Sintesis

Esquematico del circuito generado en Quartus Prime

El esquematico que se genera por Quartus realizando modelado estructural (a excepción de modulos muy sencillos en el diseño) fue el siguiente:

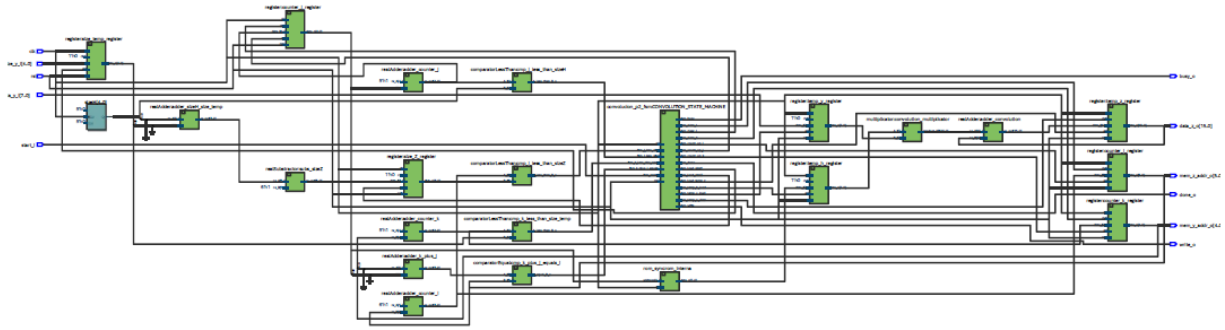

Ilustración 13: esquematico generado por la herramienta Quartus Prime Lite

El gran bloque rectangular en el centro es el control path o maquina de estados finitos descrita por el diagrama de estados de la sección anterior. El bloque color azul de la extrema izquierda se trata de un flipflop que guarda el valor numerico del tamaño de los elementos en la ROM interna, es decir la longitud del vector H.

Waveform de la simulación del diseño

La forma de onda del diseño es la siguiente, en las siguientes imágenes se muestra el contenido de las memorias Y y H, tanto como el resultado que quedo almacenado en la memoria Z.

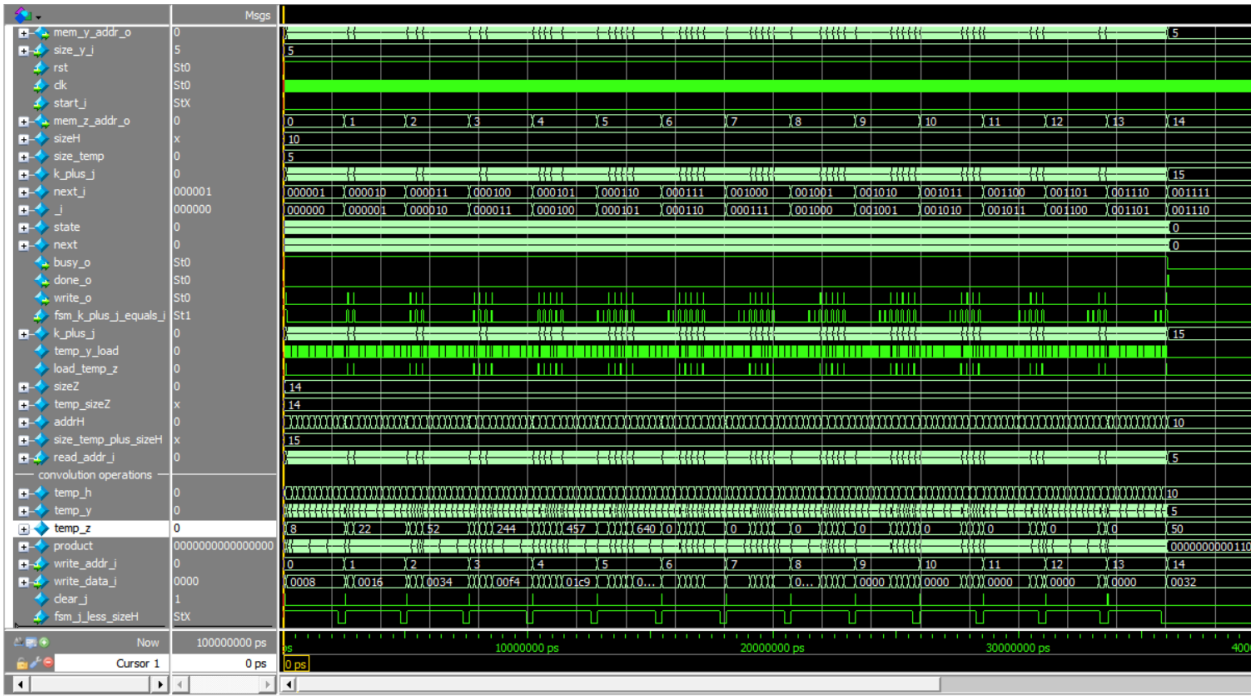

Ilustración 14: waveform expandido de las señales del coprocesador

Los valores de los vectores fueron los siguientes:

$$h = \{8,6,16,162,21,18,36,22,9,10\}$$

$$y = \{1,2,3,4,5\}$$

El contenido de la memoria H:

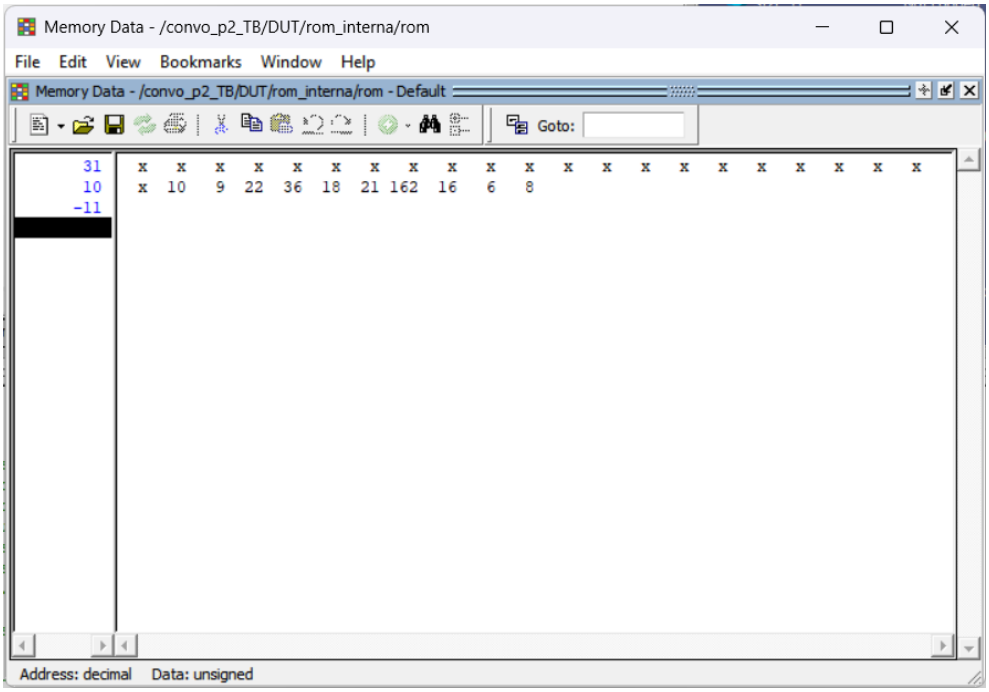

Ilustración 15: memoria H

Para Y:

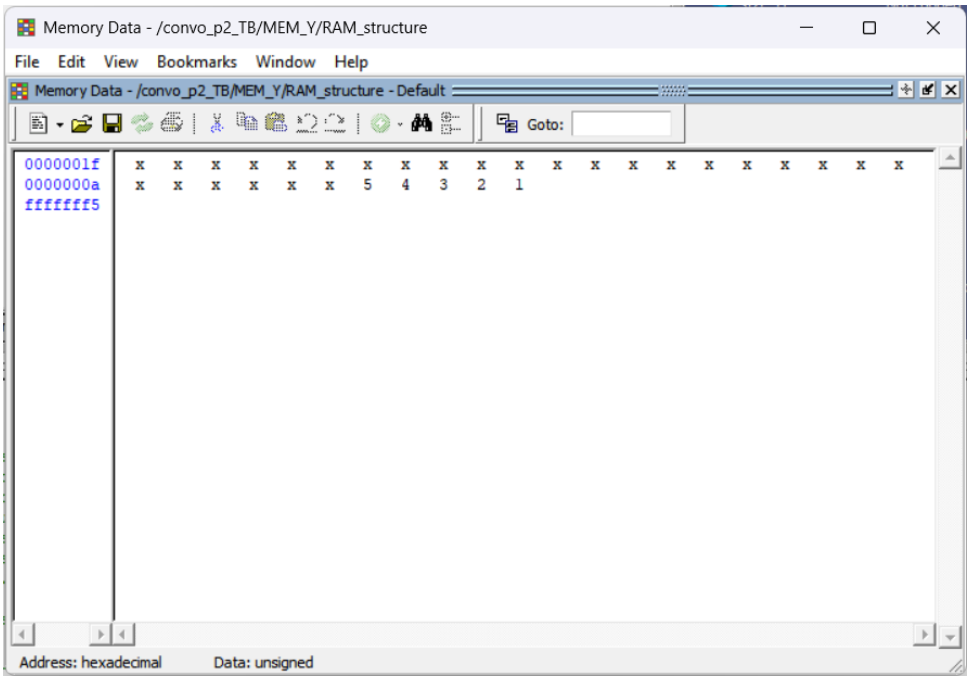

Ilustración 16: Memoria Y

El resultado almacenado en la memoria Z:

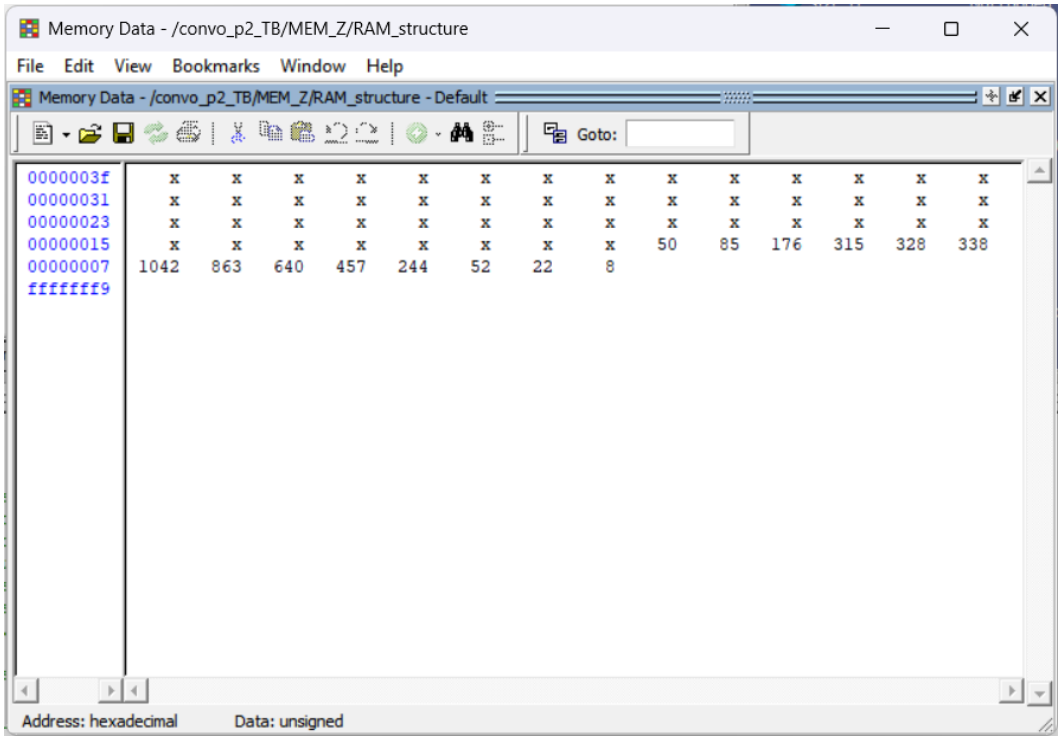

Ilustración 17: Memoria Z

Area

Revision Name	convolucion_p2
Top-level Entity Name	convolucion_p2
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	46 / 32,070 (< 1 %)
Total registers	94
Total pins	46 / 457 (10 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	1 / 87 (1 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Ilustración 18: Area del diseño en la placa de la implementacion

Maxima frecuencia de trabajo

	Fmax	Restricted Fmax	Clock Name	Note
1	146.5 MHz	146.5 MHz	clk	

Ilustración 19: frecuencia maxima

Ciclos de reloj empleados en la operación

La medición se realiza desde el flanco de subida para la señal start hasta el flanco de subida del flanco done:

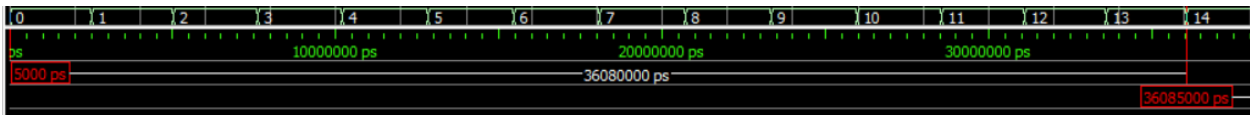

Ilustración 20: Rango desde start hasta done

Para la señal de prueba, el periodo de la señal *clk* fue de $10000\ ps$ o de $10\ ns$ por lo que la operación nos entrega de resultado:

$$\frac{5000\ ps - 36085000\ ps}{10000\ ps} = 3608\ cycles$$