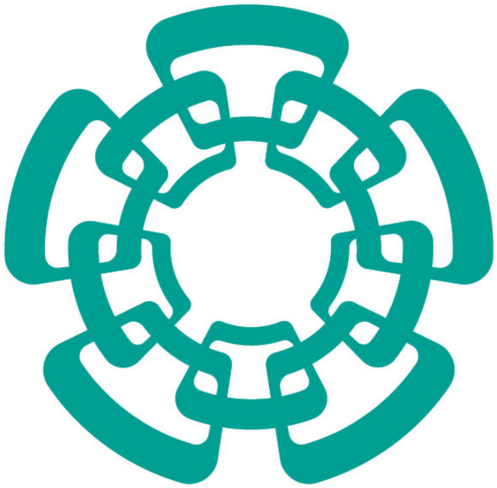
Diseño de un Coprocesador de convolución discreta

Proyecto 1

Diego Alejandro Sánchez Kelly



Metodología de diseño con SoC's

Universidad de Guadalajara || CINVESTAV

Índice

Índice de figuras

Índice de cuadros

1. Problema

El proyecto consiste en diseñar e implementar un coprocesador de convolución discreta en hardware utilizando la Metodología de diseño Top-Down.

1.1. Descripción

Se pretende diseñar un procesador de convolución que cuente con 2 interfaces de entrada, X y Y, y una interfaz de salida Z, que corresponden a las señales discretas X, Y y Z de la Ecuación 1.

Recordando,

$$Z[n] = \sum_{k=-\infty}^{\infty} x[n] \cdot y[n-k] \tag{1}$$

Es decir, la convolución en tiempo discreto $Z[n]$ de dos señales $X[n]$ y $Y[n]$ es la suma de menos infinito a infinito de los productos de todos los términos de X y de Y desplazada k-veces.

1.2. Definicion de Requisitos

Una descripción de alto nivel puede encontrarse en la Figura 1.

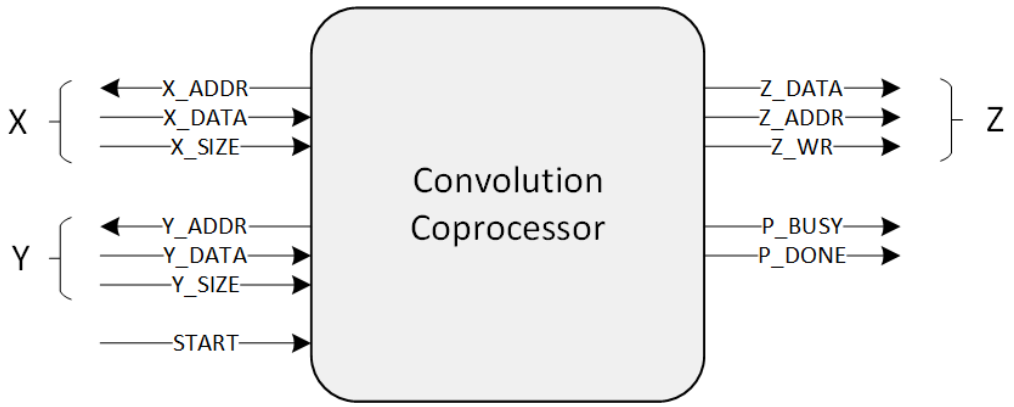


Figura 1: Descripción de Caja Negra

1.3. Interfaces de Memoria

El coprocesador deben implementar con 3 interfaces de memoria denominadas X, Y y Z.

Las interfaces de memoria X, Y y Z deben implementar una señal de salida correspondiente a la dirección de memoria de sus datos, denominadas `INT_ADDR`. Cada interfaz debe ser capaz de direccionar hasta 32 direcciones de memoria distintas.

Las interfaces de memoria X y Y deben implementar una señal de entrada de datos con un ancho de 8 bits, denominadas `INT_DATA`.

La interfaz de memoria Z debe implementar una señal de salida de datos con ancho de 16 bits, denominada `Z_DATA`.

Las interfaces de memoria X y Y deben implementar una señal de entrada denominada `INT_SIZE`, con ancho de 5 bits, utilizada para determinar el número de muestras presentes en cada señal. [1]

Finalmente, la interfaz de memoria Z debe implementar una señal de salida denominada `Z_WR`, utilizada para habilitar la escritura a la memoria correspondiente.

1.4. Señales de control

El coprocesador debe contar con tres señales de control denominadas `START`, `P_BUSY` y `P_DONE`.

La señal `START` debe ser una señal asíncrona de entrada utilizada para iniciar la máquina de estados interna al coprocesador.

Las señales `P_BUSY` y `P_DONE` deben ser señales síncronas de salida utilizadas para compartir el estado del módulo al resto del SoC.

La señal `P_BUSY` debe ser afirmada al iniciarse el procesamiento, y debe permanecer así hasta finalizarse, momento en el que debe negarse. Asimismo, la señal

[1] INT refiere al nombre de la interfaz.

P_DONE debe afirmarse al finalizar el procesamiento durante un solo ciclo de reloj [2], posteriormente, debe negarse.

2. Algoritmo

Para calcular el resultado de la convolución discreta de dos señales, se ha seleccionado un algoritmo tabular basado en el cálculo individual del producto de cada una de las muestras de las señales X y Y, almacenado en una matriz temporal denominada ZTemp.

Así,

$$ZTemp_{mn} = \prod_{m=0}^{size(x)} \prod_{n=0}^{size(y)} \tag{2}$$

tal que ZTemp contenga una tabla de todos todos los productos $X \cdot Y$. Visualmente, el algoritmo está representado en la Figura 2.

Posteriormente, la suma de los elementos de la diagonal k de la matriz ZTemp es el resultado $Z\,[k]$, tal que,

$$Z\,[k] = diag(Z_{mn}) \tag{3}$$

este algoritmo tiene como desventaja el requerir una memoria interna para almacenar los elementos de $ZTemp$ previos al cálculo final de $Z\,[k]$, por lo que para cubrir todos los casos $size(X) = size(Y) = 32$, serían necesarios 32^2 diferentes ubicaciones temporales de memoria, equivalentes a 1kB.

La complejidad temporal del algoritmo es de $O\,(n^2)$. Lo mismo puede decirse de su complejidad de almacenamiento. La Figura 3 presenta una gráfica de la relación del tamaño de entrada con la cantidad de operaciones necesarias.

Obviamente, el algoritmo requiere de optimización, la cual recae en la propia naturaleza del algoritmo.

[2] One-shot

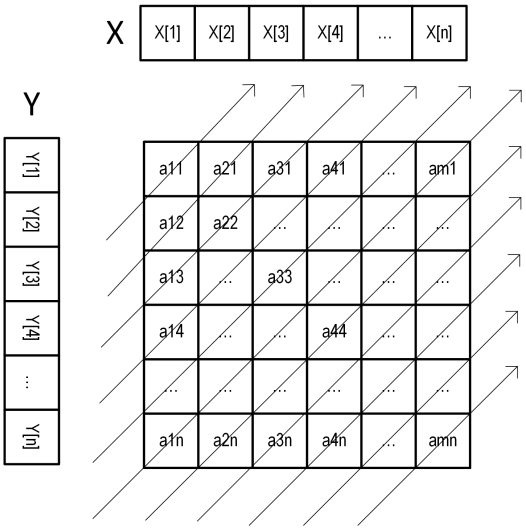


Figura 2: Representación del algoritmo.

Actualmente, el algoritmo depende de una memoria para almacenar $m \cdot n$ productos, sin embargo, estos productos pueden calcularse en demanda y sumarse a un registro temporal.

Esto permite mantener la complejidad temporal $O\left(n^2\right)$, pero reduciendo la complejidad de almacenamiento a $O(1)$, es decir, serán necesarias $m \cdot n$ operaciones para calcular el resultado de la convolución, pero con una cantidad fija de almacenamiento.

En un FPGA, el almacenamiento es más relevante que la complejidad temporal ya que este debe ser implementado en el hardware, de igual manera, al ser un coprocesador, será más rápido que realizar los cálculos utilizando solo software.

Por tanto, el algoritmo modificado es mucho más adecuado.

2.1. Descripción

Comúnmente, para calcular la convolución de dos señales se utiliza un algoritmo recursivo de desplazamiento, producto y suma, o un algoritmo basado en una transformada rápida de Fourier.

El algoritmo recursivo es útil, pero complejo de implementar en un diseño digital,

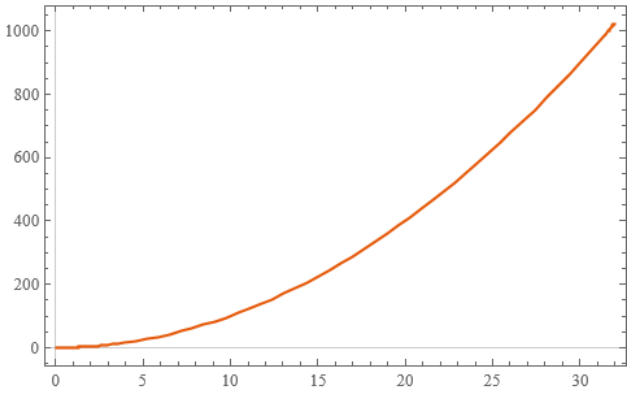


Figura 3: Complejidad algorítmica.

manteniendo complejidad en $O\left(n^2\right)$ sin ganancias particulares en velocidad.

El algoritmo de FFT es mucho más veloz, con complejidad $O(n \log n)$, pero es aún más complejo de implementar en un diseño digital.

El algoritmo tabular es similar al algoritmo recursivo, pero ahorrando el desplazamiento de las señales y la necesidad de re-calcular o almacenar los productos anteriores para calcular las sumas de productos $n - k$ cuando $k \to \infty$.

Dado que este diseño está pensado para implementarse en par con un núcleo de procesamiento y su respectiva memoria, es posible eliminar el desplazamiento y utilizar las direcciones de memoria de los vectores X y Y como índices para el cálculo de las diagonales.

Esto asegura 3 cosas:

- Cada índice a_{mn} de la matriz Z será calculado y utilizado una sola vez.
- No se requieren ciclos de reloj extra para desplazar Y k veces.
- El almacenamiento temporal requerido es estático y no cambia con el tamaño de X y Y.

Por lo tanto, es óptimo para resolver el problema de convolución cuando se conocen los tamaños máximos de las señales de entrada. La Figura 4 muestra el código del algoritmo implementado en C.

Dada la presencia de dos ciclos anidados, es evidente que la complejidad es $O(n^2)$ a través de todo el rango de tamaños posibles, con un límite en 1024 operaciones para 2 vectores de un máximo de 32 elementos.

```
1   void conv(sizeX, sizeY, dataX, dataY, dataZ) {
2     int nDiag, cDiag, xAddr, yAddr;
3     long int temp, product;
4     nDiag = sizeX + sizeY - 1;
5     for (int cDiag = 0; cDiag < nDiag; cDiag++) {
6       if ( cDiag < sizeY ){
7         xAddr = 0; yAddr = cDiag;
8       }
9       else {
10        yAddr = cDiag - sizeY + 1; xAddr = sizeY - 1;
11      }
12
13      while (xAddr > sizeX && yAddr >= 0) {
14        product = dataX[xAddr] * dataY[yAddr];
15        temp += product;
16        xAddr++; yAddr--;
17      }
18      dataX[cDiag] = temp;
19    }
20  }
```

Figura 4: Cálculo de convolución por diagonales.

En la Figura 4 se muestra el cálculo, que primero calcula el tamaño total del cálculo. Posteriormente, utiliza un ciclo (for) que corresponde a las diagonales, mientras que utiliza otro ciclo (while) para calcular los productos y sumas de cada diagonal.

Antes del inicio del segundo ciclo anidado, las variables de dirección de los vectores de entrada se cargan con los valores respectivos, que dependen de la diagonal que actualmente se está calculando. Esto es debido a que en un cálculo, para vectores de tamaño n y m existen n diagonales superiores, y $m-1$ diagonales

inferiores.

Cuando se calculan las diagonales superiores, se debe comenzar en la columna 0, en la fila con índice igual a la diagonal.

3. Algorithm State Machine

La descripción del algoritmo a nivel RTL requiere agregar algo de lógica al manejo de señales externas, como es el caso del estado inicial y la entrada `start_i` así como las salidas `done_o` y `busy_o`.

La descripción inicial, vista en la Figura 5, cuenta con 24 eventos diferentes.

El núcleo permanecerá en el estado inicial hasta recibir la señal de inicio. Posteriormente, al recibir dicha señal, se inicializan todas las señales utilizadas durante el cálculo, así como las señales de estatus del núcleo.

Un comparador externo a la máquina determina cuando el cálculo ha finalizado, mientras este no sea el caso, otro comparador verifica si se están calculando diagonales superiores e inferiores. Con esta infomación, se cargan los diferentes valores de x mencionados en la Sección 2.

Los valores de x y y en las respectivas direcciones de memoria se obtienen y almacenan, y posteriormente el producto es calculado y almacenado, y las variables correspondientes a direcciones de memoria y diagonales son incrementadas (o decrementadas) finalmente las comparaciones determinan si la diagonal se ha terminado y si el cálculo completo ha finalizado.

Si la diagonal ha finalizado, se escriben los contenidos a la memoria Z, posteriormente, si ha finalizado el cálculo, las señales de estatus externas se actualizan, de otra manera, se calcula la siguiente diagonal y el ciclo se repite.

3.1. Optimizaciones

Dada la naturaleza del algoritmo, algunas optimizaciones son posibles. Asignando una bandera a cada estado como salida de la FSM, es posible utilizar dicha

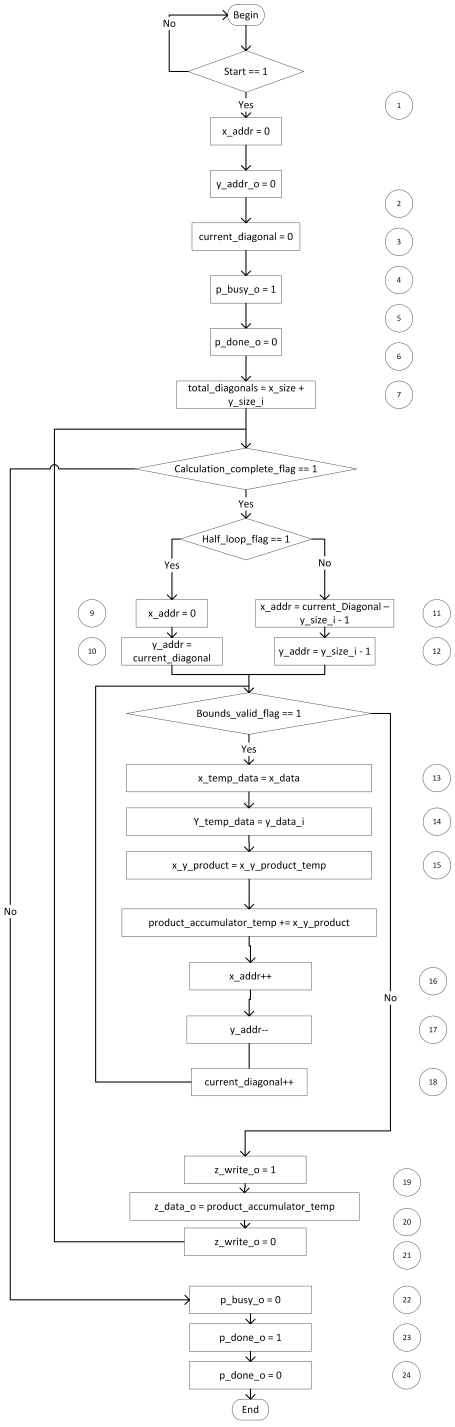


Figura 5: Algorithm State Machine

bandera en diferentes módulos con diferentes efectos (clear, enable, etc.), de tal manera que los estados pueden optimizarse fácilmente. La Figura 3.1 muestra la descripción de la máquina de estados finalizada.

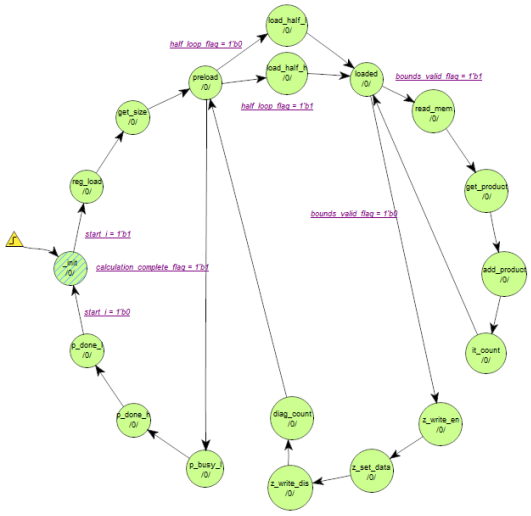


Figura 6: Máquina de estados optimizada.

En este caso, los estados 2-7 se optimizan en un solo estado llamado `reg_load`. Se dedica un estado a calcular y asegurar que el tamaño de la operación sea estable en memoria, a este estado se le denomina `get_size`.

Se añade un estado *dummy* antes y después de la carga de valores para la operación, denominados `preload` y `loaded`. Entre estos estados, según el estaus de la bandera `half_loop_load_flag`, se cargan los valores de `x_address` y `y_address` respectivamente. A estos estados se les llama `load_half_h` y `load_half_l`.

Posteriormente, se leen las memorias X y Y, en el estado `read_mem`, luego, se obtiene el producto, en el estado `get_product`, se suma el producto al acumulador, en el estado `add_product` y finalmente se cuenta la iteración del cálculo, en el estado `it_count`.

Se regresa al estado loaded, en el cuál se define si se ha terminado la diagonal actual, en cuyo caso, se pasa a los estados `z_write_en`, `z_set_data` y

z_write_dis, que se encargan de habilitar la escritura a Z, colocar los datos en la salida Z, y desactivar la escritura en Z, respectivamente.

Al finalizar el proceso de escritura, se incrementa la diagonal en el estado diag_count y se regresa al estado preload, en donde, según el estatus del cálculo, se procede al estado p_busy_l y se desactiva la bandera de estatus p_busy_o.

Finalmente, en los estados p_done_h y p_done_l la bandera de estatus p_done_o se activa y desactiva, respectivamete, para lograr una salida de tipo *one-shot*.

3.2. Mejoras

Posibles mejoras al diseño incluyen el eliminar los estados *dummy* del diseño en aras de reducir el tiempo de procesamiento, así como optimizar el flujo de algunas señales, como es el de z_write_o que está activa innecesariamente durante 3 ciclos de reloj.

Ya que los estados preload y loaded se ejecutan múltiples ocasiones dentro del flujo del núcleo, es posible que eliminar dichos estados resulte en una mejora clara en rendimiento.

Algo similar aplica para el caso de la bandera de control de escritura de la memoria Z.

4. Datapath

El *datapath* del diseño es concebido en 3 etapas, en primera instancia, haciendo un análisis de los bloques esenciales asociados con cada estado, tal como lo menciona el método. La Figura 4 muestra este paso del diseño.

Debido al algoritmo que se escogió depende mucho de comparaciones, varios de los pasos implementan comparadores dentro para generar señales que posteriormente se utilizan como entradas a la máquina de estados, o como entrada a otros de los bloques del núcleo.

Otro bloque utilizado en varias ocasiones dentro del núcleo es el de adición, en

algunos casos utilizando números negativos. Estos bloques son los que permiten que la máquina de estados sea capaz de conocer el progreso del core a través de ciertas banderas de estatus.

En la Figura 4 se observan las primeras optimizaciones realizadas al datapath anteriormente mostrado en la Figura 4, donde esencialmente se combinan algunos estados y re-utilizan algunas banderas de estados.

Aunque no forman parte del resultado final, ni parte del modelo utilizado para la síntesis final del core del SoC, se muestran varios flip-flops que corresponden a las banderas de salida del core hacia el IP interface (definidas en la Sección 1). Más adelante en el proceso de diseño, estos elementos se eliminaron, ya que la instancia de la máquina de estados del core ya implementa estas salidas.

Asimismo, se realizó una serie de optimizaciones finales al diseño:

- Los bloques de conteo/almacenamiento de las direcciones de X y Y se unifican en contadores ascendentes o descendentes, según el caso. Estos contadores se implementan también con salidas de overflow/underflow, que actuan como banderas, eliminando la necesidad de comparadores externos. Lo mismo aplica para el contador de diagonales y de direcciones de Z.
- El cálculo de algunas sumas/restas se hace de manera comportamental, dejando al sintetizador inferir el hardware implementado. Esto redució bastante la complejidad de conexiones y flujo de datos interno, haciendo más facil la depuración.
- La máquina de estados se re-diseña con 1 flag por estado, con la excepción de los estados dummy que no tienen banderas.

5. Simulaciones

El núcleo se simuló con las memorias RAM Y y Z implementadas en un testbench con un reloj a 200MHz, el código de dicha cama de pruebas puede encontrarse en la Figura 5.

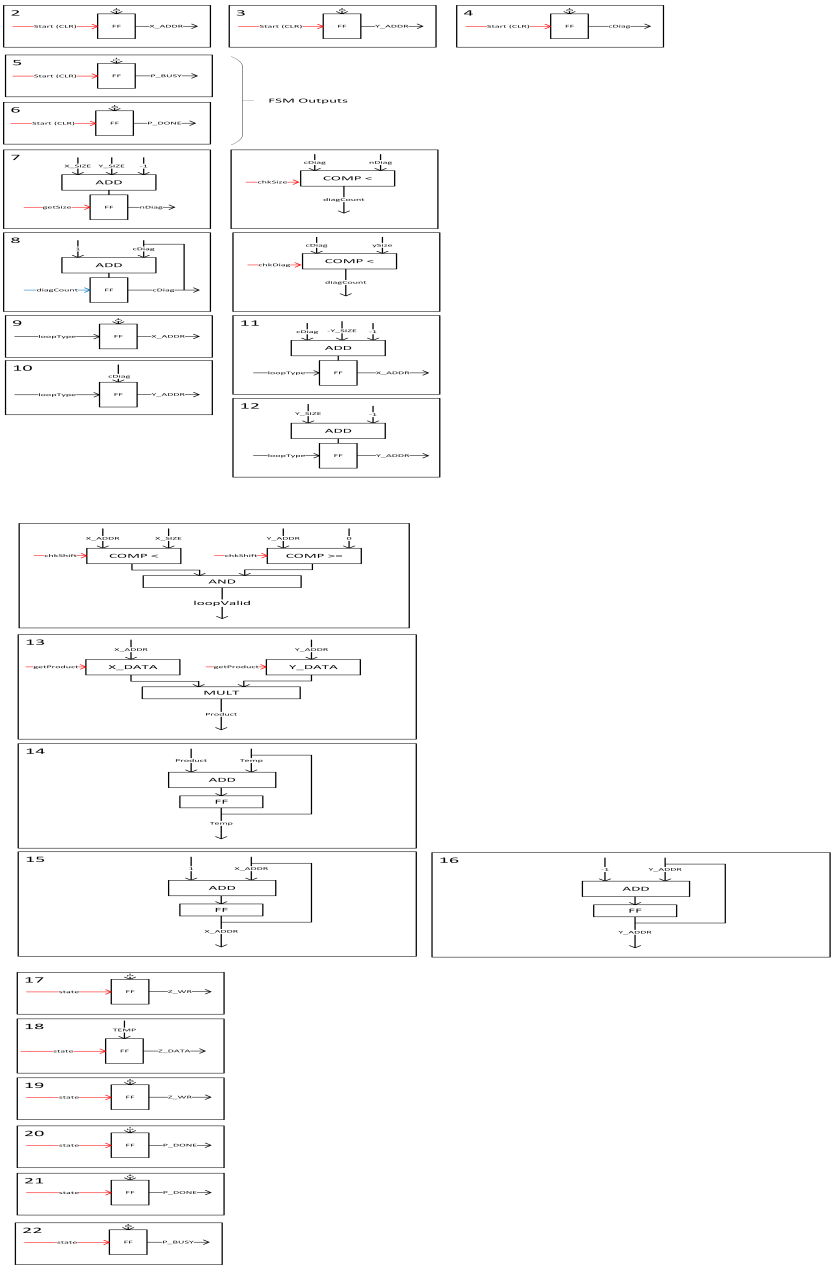


Figura 7: Datapath sin optimizacion

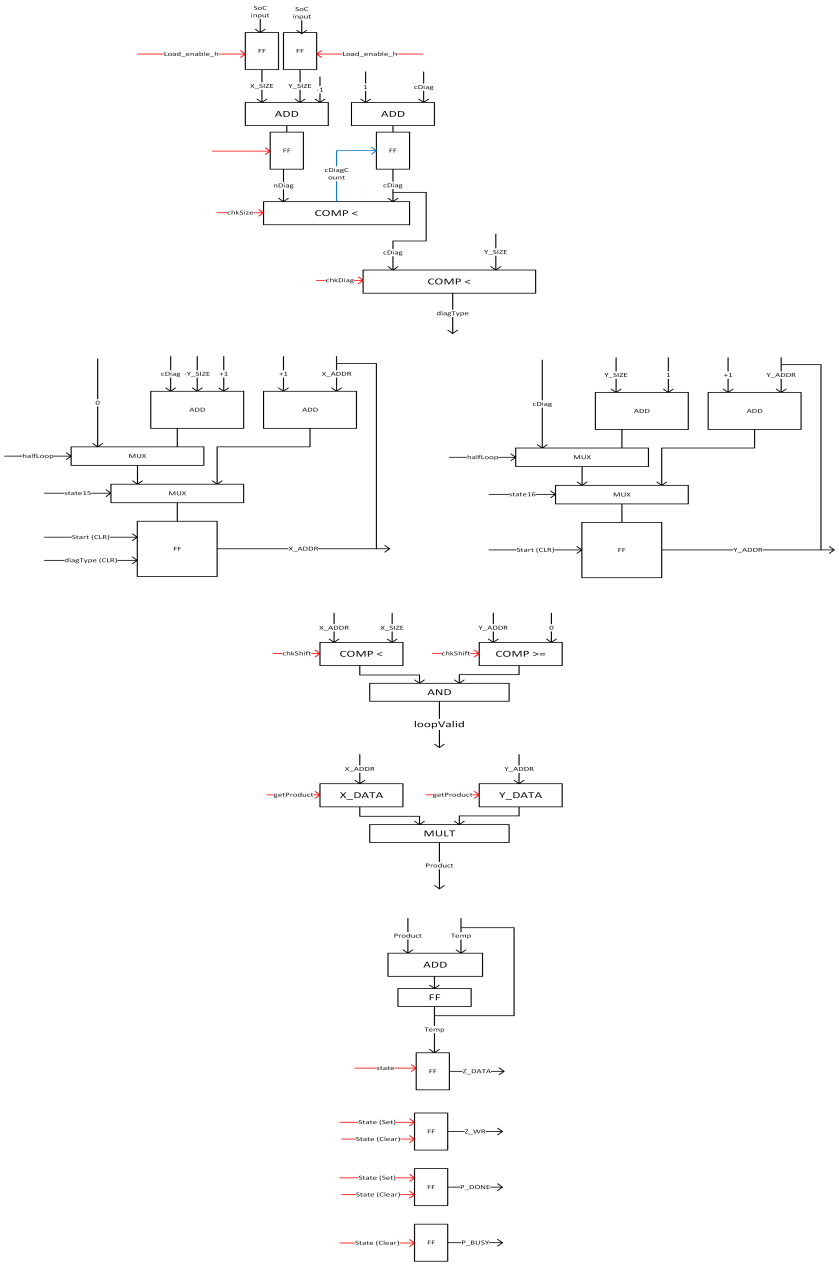


Figura 8: Datapath tras la primera ronda de optimización.

En esta configuración, el núcleo toma 278 ciclos de reloj para calcular la convolución de dos vectores de dimensiones 10 y 5, lo cual es equivalente a 2.785µs.

Esta medida se toma en el *falling-edge* de la señal de estatus `done` interna a la FSM, como se puede apreciar en la Figura 5.

6. Síntesis

El reporte de la herramienta se encuentra en la Figura 6. El diseño utiliza menos de 100 registros en total y menos del 1 % de los elementos lógicos del sistema.

Asimismo, utiliza 46 pines y un bloque de DSP.

La frecuencia máxima reportada es de 274.2MHz para el modelo lento a 1.1V 85C, mientras que es de 267.17MHz para 0C. Los modelos rápidos no reportan una frecuencia máxima.

Igualmente, tampoco hay información sobre el área utilizada.

El Esquemático RTL se puede encontrar en el Apéndice A.

A. RTL

```
module convolutor_tb ();

  int i;
  logic clk, rst_n, start_i, busy, zwr, done, z_dout;
  logic [DATA_WIDTH-1:0] y_data; logic [ADDR_WIDTH-1:0] y_addr;
  logic [2*DATA_WIDTH-1:0] z_data; logic [ADDR_WIDTH:0] z_addr;

  convolutor conv_top (
    .clk       (clk),
    .rst_n     (rst_n),
    .start_i   (start_i),
    .y_data_i  (y_data),
    .y_addr_o  (y_addr),
    .y_size_i  (5'd5),
    .z_data_o  (z_data),
    .z_addr_o  (z_addr),
    .p_busy_o  (busy),
    .p_done_o  (done),
    .z_write_en(zwr)
    );

  convolutor_ram_dual_port # (
    .DATA_WIDTH(DATA_WIDTH),
    .DEPTH(32),
    .ADDR_WIDTH(ADDR_WIDTH),
    .MEMFILE   ("./Sequential/RAM/MEMY.mem")
    ) Y_RAM (
    .clk            (clk),
    .read_address_i (y_addr),
    .data_output_o  (y_data),
    .write_enable_i (1'b0),
    .write_address_i(5'b0),
    .write_data_i   (8'b0)
    );

  convolutor_ram_dual_port # (
    .DATA_WIDTH(16),
    .DEPTH(64),
    .ADDR_WIDTH(6),
    .MEMFILE   ("./Sequential/RAM/MEMZ.mem")
    ) Z_RAM (
    .clk            (clk),
    .read_address_i (6'b0),
    .data_output_o  (z_dout),
    .write_enable_i (zwr),
    .write_address_i(z_addr),
    .write_data_i   (z_data)
    );

forever #5 clk = ~clk;

  initial begin
    start_i = 0; clk = 1'b0; rst_n = 1'b0; #1;
    rst_n = 1'b1; start_i = 1'b1; #3000;
    $finish;
  end

endmodule : convolutor_tby
```

Figura 9: Cama de pruebas del sistema.

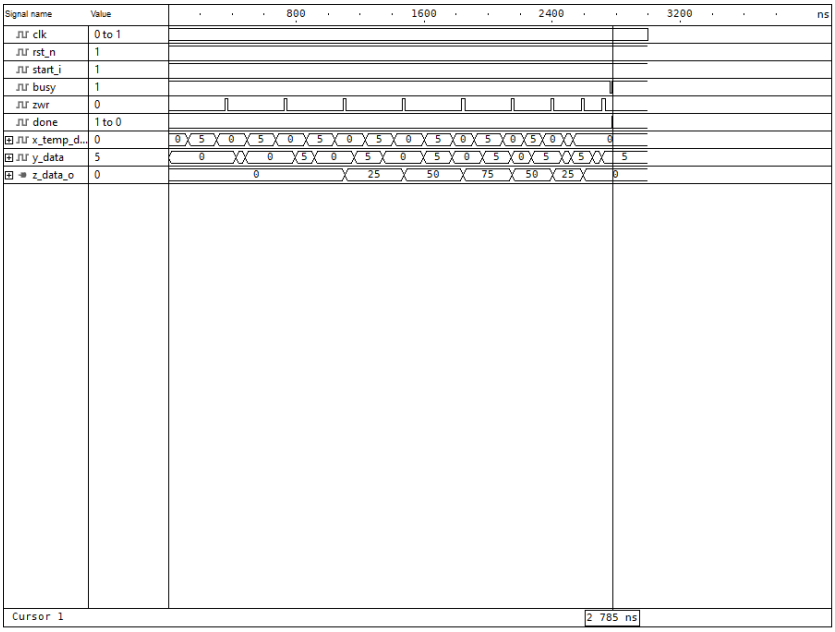


Figura 10: Simulación, convolución 10x5.

Flow Status	Analyzed - Fri May 10 00:13:33 2024
Quartus Prime Version	23.1std.0 Build 991 11/28/2023 SC Lite Edition
Revision Name	CONVOLUTOR
Top-level Entity Name	convolutor
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	58 / 32,070 (< 1 %)
Total registers	97
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 %)

Figura 11: Reporte de síntesis de la herramienta.