Diseño De Un Acelerador Con Vitis HLS (PARTE II)

UNIVERSIDAD DE MÁLAGA

Autores:

Ivan Iroslavov Petkov
Álvaro Bermúdez Gámez
Ignacy Grzymlot Borzestowski
Iván Lago Guerreira

Pertenecientes a los grados de

Ingeniería Informática e Ingeniería del Software

GRUPO NEGRO

UNIVERSIDAD DE MÁLAGA

E.T.S.I INGENIERÍA INFORMÁTICA

Abril 2024

Visión general del proyecto

Esto es un documento para la realización de un acelerador en Vitis HLS, que decidimos llamar 'Acelerador II', que comenzó a diseñarse acorde al siguiente enunciado:

Dada una señal de entrada, S, de M=1024 muestras reales entre (-1 y 1), y un trozo, T, de dicha señal (o muy similar) de 16 muestras, encontrar la posición de dicho trozo en la señal original. Para ello iremos desplazando T sobre S, de muestra en muestra, y calculando la energía del error, siendo dicha energía, la suma de la diferencia entre las muestras correspondientes al cuadrado

$$Energía = (S(i+n) - T(n))^2$$

(para todo n entre 0 y 15). La salida de la función será el valor de i que minimiza dicha energía (idealmente cero), junto con dicha energía. La función recibe como parámetros dos arrays con las muestras de S y T, un entero para escribir la posición i, y otro escalar para devolver la energía del error correspondiente a dicha posición.

Sin embargo, a este Informe vamos a añadirle una actualización, y haremos un 'Acelerador II.2' de la siguiente forma:

Dado el 'Acelerador II', considerar que los valores tanto de S como de T, son valores aleatorios entre -1 y 1 generados mediante la función rand(). Reducir aún más el área del circuito usando la librería de punto fijo (fixed point) para representar los valores y hacer los cálculos. Para ello tener en cuenta que se permite resultados erróneos del acelerador siempre que estos se produzcan menos del 1 % de todos los casos calculados. Por otro lado, diseñar otra arquitectura para la misma función pero que maximice el throughput sin un uso desproporcionado de recursos. Es decir, que maximice el valor de la función throughput/área, siendo el área:

$$\text{Área} = \#FF + \#LUT + (\#DSPs \times 200)$$

Índice general

Introducción

Contents

1.1 Introducción al informe

En este informe vamos a modificar el acelerador desarrollado anteriormente en el ejercicio "Diseño de un acelerador II". Trabajaremos con la misma señal **S** y el trozo **T** con valores entre -1 y 1 generados mediante la función **rand()** y con la misma formula:

$$((S(i+n)-T(n))^2),$$

para todo n en el rango de 0 a 15.

La salida de nuestra función será el valor de **i** que minimiza la energía del error (idealmente sería cero), junto con el valor de la energía del error correspondiente a dicha posición. Los parámetros de entrada para la función incluyen dos arrays que contienen las muestras de S y T, un entero para escribir la posición i, y un escalar para devolver la energía del error en dicha posición.

Ahora se buscará disminuir el área usando la librería de punto fijo (fixed point) para representar los valores y hacer los cálculos. Además solo consideraremos correcto a los aceleradores donde se produzcan menos de un 1 % de errores de todos los calculados. Es decir, en nuestro caso realizaremos 1000 pruebas y solo consideraremos correctos los aceleradores que tengan menos de 10 casos erróneos.

Implementación Base

Contents

2.1 Descripción de Pragmas

En el contexto de la optimización de hardware con Vitis HLS, el uso de pragmas para la partición de arrays es una técnica crucial para mejorar el acceso paralelo a los datos y optimizar el rendimiento del diseño. La partición **completa** de un array T en este caso, asigna cada elemento del array T a una memoria independiente, facilitando el acceso simultáneo a todos los elementos. Esto es particularmente útil para procesar los elementos de un array en paralelo.

```
#pragma HLS ARRAY_PARTITION variable=T type=complete
```

Por otra parte, la partición **cíclica**, organiza los elementos del array S en un conjunto de bancos de memoria, distribuyendo los elementos de manera que cada treintaidosavo elemento consecutivo caiga en el mismo banco. Este enfoque balancea el acceso entre los bancos de memoria, lo cual es beneficioso en situaciones donde múltiples operaciones necesitan acceder a partes diferentes del array de manera eficiente, evitando cuellos de botella en el acceso a un único banco de memoria.

```
#pragma HLS ARRAY_PARTITION variable=S type=cyclic factor=32
```

Para acabar, con un pragma para el **pipeline**, optimizaremos aún más el diseño al permitir la ejecución en paralelo de un bucle (también se puede aplicar a funciones). Este pragma intenta que cada iteración del bucle se ejecute en un ciclo de reloj, lo que **aumenta considerablemente el throughput del sistema**.

```
#pragma HLS PIPELINE
```

2.2 Explicación de los códigos

El código A.1 que está en la página 14 representa la cabecera del algoritmo. Este código define una interfaz para un algoritmo que encuentra la mejor coincidencia de un "trozo"de señal dentro de una señal más grande. Utiliza tipos de datos de **punto fijo** para representar las señales y opera sobre arrays de tamaño fijo (1024 para la señal principal y 16 para el trozo). La función **findBestMatch** busca la posición dentro de la señal principal (**TSignal**) donde el trozo de señal (**TTrozo**) minimiza la energía del error, la cual es calculada como la **suma de las diferencias al cuadrado** entre los elementos correspondientes de TSignal y TTrozo.

Por otro lado, el código A.2 situado en la página 15 representa la implementación del algoritmo del cálculo de energías completo. Este código implementa la función **findBestMatch** que busca la posición en una señal grande (**S**) donde un trozo de señal (**T**) minimiza el error de ajuste. El error se calcula como la **suma de las diferencias al cuadrado entre los elementos de S y T en cada posible posición de S**. La función itera sobre la señal **S**, comparando cada segmento de longitud igual a **T**, calcula la energía del error, y mantiene registro de la posición donde esta energía es mínima. Al final del proceso, devuelve la posición y el valor mínimo de la energía del error encontrados.

Para terminar, el código A.3 situado en la página 16 representa el "test bench", que sería nuestro entorno de pruebas para comprobar la funcionalidad del algoritmo. Este código sirve para evaluar la función **findBestMatch** que implementa un algoritmo de búsqueda para encontrar la posición en una señal grande donde un segmento más pequeño minimiza el error de ajuste. El test bench **genera 1000 señales aleatorias como datos de prueba**, asignando cada una a una estructura de señal S y extrayendo un 'trozo' T para comparar. Luego, se utiliza la función **findBestMatch** para determinar la posición en S

que mejor se ajusta a T. El test evalúa si la energía del error es menor a 0.5 y si la posición está dentro de un rango válido, registrando los resultados como correctos o incorrectos. Al final, se imprime un resumen de los tests correctos e incorrectos.

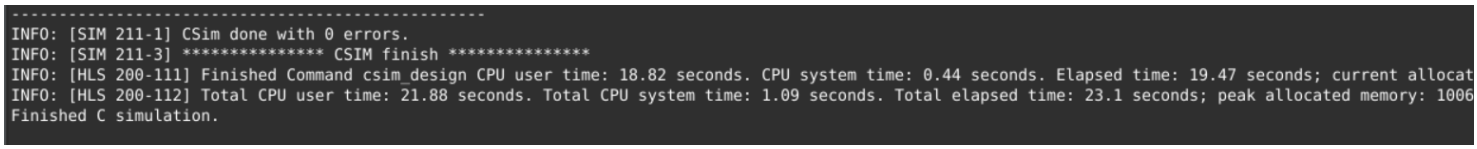

Figura 2.1: Aquí se puede observar la impresión por pantalla que produce el Test Bench, en el caso de que todas las pruebas hayan salido bien

2.3 Camino de datos de la implementación 'Base'

Podemos observar que hemos divido la señal en 32 memorias (en la imagen se muestran 2 memorias y 1 registro (trozo) y se dice que se repite 16 veces) y se aplica el pipeline añadiendo los registros entre los componentes.

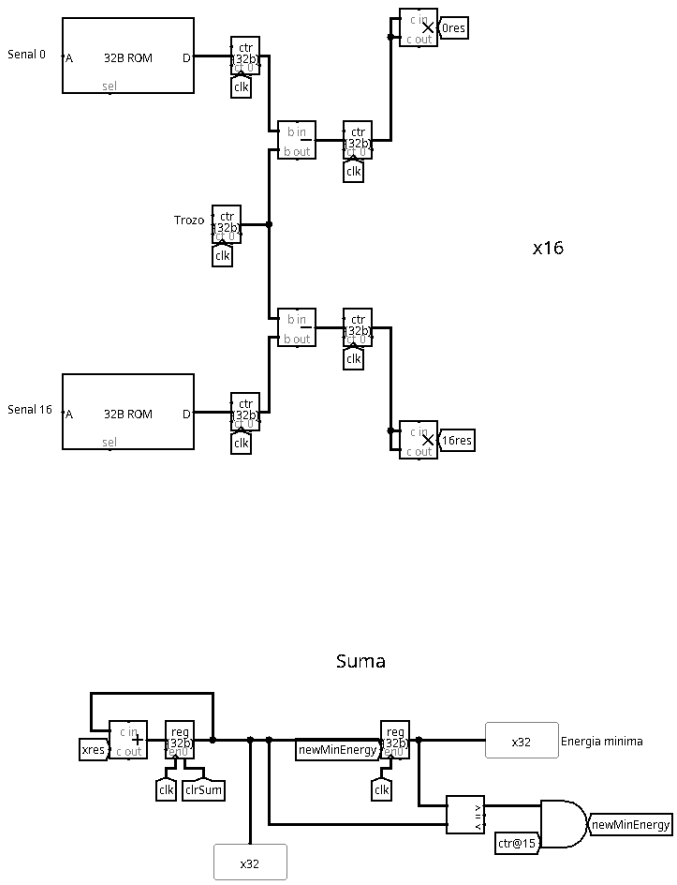

Figura 2.2: Implementación de la arquitectura base en Logisim

2.4 Resultados de la síntesis

Como podemos ver en la tabla, los resultados de esta tabla no son muy eficientes respecto al throughput. En cuanto al área, vemos que no es necesariamente grande. En las siguientes implementaciones, el objetivo será **minimizar el área** (Arquitectura 1) y luego **maximizar el throughput** (Arquitectura 2).

Latencia (ciclos)	Latencia (ns)	FF	LUT	DSP	RAM
1013	1.013E4	797	7351	0	0

Cuadro 2.1: Resultados de la síntesis de la arquitectura 'Base'

Arquitectura 1 (Minimizando Área)

Contents

3.1 Descripción de la 'Arquitectura 1'

Esta arquitectura enfocada en minimizar el área que la 'Arquitectura Base' nos ofrece, ya que esa arquitectura implementa múltiples técnicas de optimización para aumentar la velocidad de procesamiento.

En contraste, la aproximación de la 'Arquitectura 1' elimina esas optimizaciones de hardware con el fin de reducir el consumo de recursos. Al **deshabilitar** funciones que normalmente amplían el uso del área, como el **particionamiento avanzado de datos** y el **procesamiento en pipeline**, logramos un diseño más **compacto**. Para ello, hemos decidido especificar de forma implícita que Vitis no haga pipeline.

```
#HLS pipeline off
```

Este enfoque sería esencial en una aplicación donde la **eficiencia del espacio** y la **reducción de costos** son prioritarias sobre el throughput. En esencia, sacrifica la velocidad de ejecución en favor de una implementación más económica y menos demandante en términos de recursos de hardware.

3.2 Camino de datos de la 'Arquitectura 1'

Es un camino de datos básico, ya que no se le aplica ninguna directiva especial. Mientras que en la 'Arquitectura Base' observamos una división avanzada de la señal en 32 memorias, optimizando el acceso y procesamiento mediante un esquema de pipeline que intercala registros entre los componentes para mejorar el rendimiento, la 'Arquitectura 1' opta por un enfoque más **simplificado**.

En esta segunda configuración, no se aplica ninguna directiva de optimización, resultando en un camino de datos más fundamental y lineal. **Sin el uso de particionamiento o pipeline**, esta arquitectura puede no igualar la eficiencia de la Base, pero **reduce significativamente la complejidad del diseño y los recursos utilizados**, priorizando la simplicidad y menor uso de hardware.

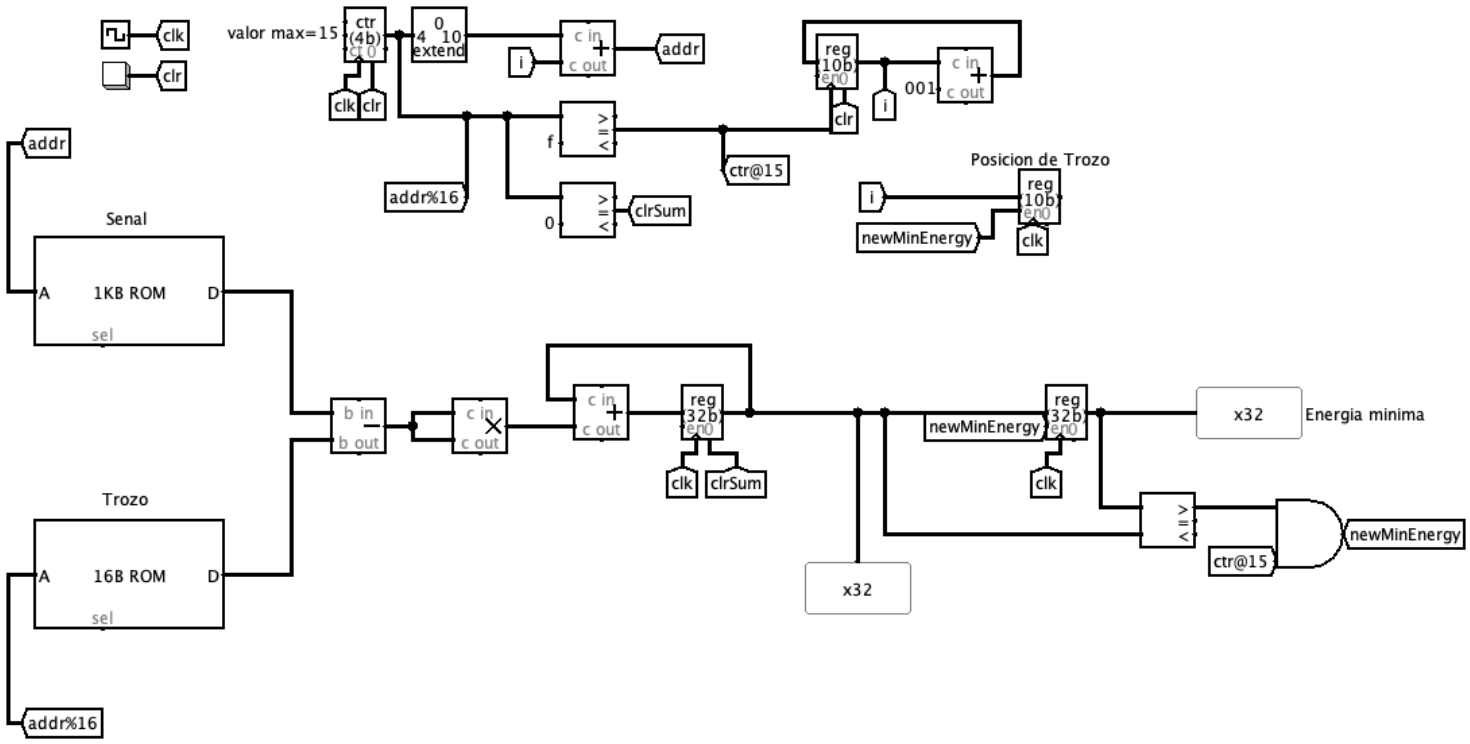

Figura 3.1: Implementación de la arquitectura 1 en Logisim

3.3 Resultados de la síntesis

En comparación con la 'Arquitectura Base', la **latencia se dispara** siendo 33 veces la latencia base. En este caso no nos molesta este resultado ya que estamos buscando minimizar el área lo máximo posible, por lo que es normal perder algo de latencia. Podemos comprobar que **los FF se han reducido bastante** (se han dividido entre 8) y **los LUT también** (se han dividido entre 9).

Latencia (ciclos)	Latencia (ns)	FF	LUT	DSP	RAM
34307	3.430E5	93	789	0	0

Cuadro 3.1: Resultados de la síntesis de la 'Arquitectura 1'

Arquitectura 2 (Maximizando Throughput)

Contents

4.1 Descripción de la 'Arquitectura 2'

La 'Arquitectura 2' **mejora el throughput** mediante ajustes en los pragmas utilizados. A diferencia de la 'Arquitectura Base', que usa particionamiento cíclico con un factor de 32 para S y completo para T, la 'Arquitectura 2' **reduce el factor cíclico a 16**. Este cambio **incrementa** el **paralelismo** y la **eficiencia en el procesamiento de datos**. Adicionalmente, se aplica **unroll** con un **intervalo de iniciación** (II=10) en el pipeline. El unroll incrementa la ejecución de **múltiples iteraciones del bucle en un solo ciclo de reloj**, mientras que el intervalo de iniciación controla el número de ciclos antes de que inicie una nueva operación, permitiendo una mayor **superposición de operaciones** y **acelerando la velocidad de procesamiento**. Esta combinación optimiza el uso del hardware al maximizar el rendimiento operacional.

```
#pragma HLS ARRAY_PARTITION variable=S type=cyclic factor=16
#pragma HLS ARRAY_PARTITION variable=T type=complete
#pragma HLS PIPELINE II=10
#pragma HLS UNROLL factor=16
```

4.2 Camino de datos de la 'Arquitectura 2'

La 'Arquitectura 2' conserva el diseño general de la parte superior del circuito de la 'Arquitectura Base', pero con una modificación clave: **se ha eliminado una memoria** debido a que el particionamiento del array se ha ajustado a un **factor de 16**. En la parte inferior del circuito, hemos implementado tanto el **pipeline** como el **unroll**, **optimizando** así la **velocidad de procesamiento**. Estas mejoras permiten un manejo más eficiente de los datos y una ejecución más rápida de las operaciones en comparación con la arquitectura base, haciendo de esta configuración una solución más eficaz para aplicaciones que requieren alto rendimiento, debido a su alto nivel de **throughput**.

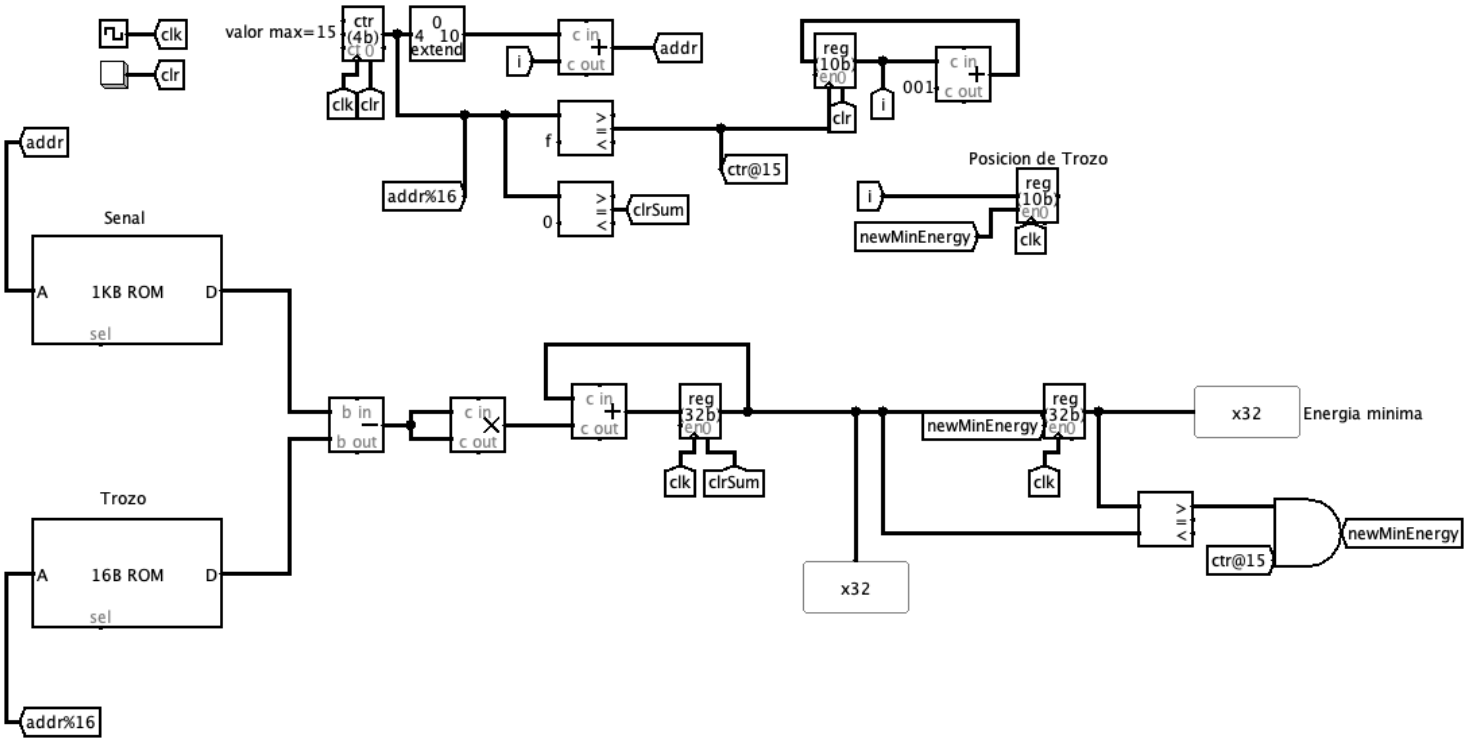

Figura 4.1: Implementación de la arquitectura 2 en Logisim

4.3 Resultados de la síntesis

La Arquitectura 2 muestra un **incremento** significativo en el uso de recursos de **hardware**, con un aumento del 74.4 % en los Flip-Flops y del 212.4 % en los Look-Up Tables, reflejando una **complejidad mayor** en comparación con la arquitectura base.

Sin embargo, este incremento se traduce en **mejoras notables en el rendimiento**, como la evidente **reducción** del 37 % en la **latencia**. Este avance demuestra una **optimización efectiva del throughput**, lo que sugiere que la Arquitectura 2 está diseñada para **maximizar la eficiencia del procesamiento**, a pesar del mayor consumo de área de hardware.

Latencia (ciclos)	Latencia (ns)	FF	LUT	DSP	RAM
640	6.400E3	1390	22969	0	0

Cuadro 4.1: Resultados de la síntesis de la 'Arquitectura 2'

Estudio comparativo de las arquitecturas

Contents

5.1 Comparación de los resultados

Los **porcentajes de mejora** calculados anteriormente se han hecho en base a las siguientes fórmulas:

$$MejoraLatencia = (ResultadoBase - ResultadoNuevo)/ResultadoBase * 100\,\%$$

$$IncrementoArea = (ResultadoNuevo - ResultadoBase)/ResultadoBase * 100\,\%$$

	Latencia (ciclos)	Latencia (ns)	FF	LUT	DSP	RAM
Arquitectura Base	1013	1.013E4	797	7351	0	0
Arquitectura 1	34307	3.430E5	93	789	0	0
Arquitectura 2	640	6.400E3	1390	22969	0	0

Cuadro 5.1: Resultados de la síntesis comparativa

Ahora, para comparar el área general respecto a la 'Arquitectura Base', vamos a seguir la fórmula para **calcular el área** que aparece a continuación. Hemos puesto los valores de la 'Arquitectura Base' a 0, ya que será nuestro punto de referencia para comparar el resto de arquitecturas

$$Area = \#FF + \#LUT + (\#DSPs * 200)$$

Para la **Arquitectura 1**, la latencia es de 34,307 ciclos, con 93 FF y 789 LUT, manteniendo los valores para DSP y BRAM en cero. Al calcular la mejora de latencia, se obtiene un resultado de -3286.67 %, indicando un aumento significativo en la latencia respecto a la arquitectura base. El incremento en área muestra una reducción del -88.33 % en FF y

del -89.27 % en LUT, resultando en un incremento general en área del -0.39 % tras aplicar la media ponderada.

En la **Arquitectura 2**, la latencia es mucho menor, con 640 ciclos. Los componentes del área incluyen 1,390 FF y 22,969 LUT, mientras que DSP y BRAM permanecen en cero. La mejora de latencia con respecto a la arquitectura base es de 36.82 %, lo que muestra una clara mejora en el rendimiento. Sin embargo, el incremento en área para FF es del 74.40 % y para LUT del 212.46 %, lo que da un incremento general en área del 0.63 % usando la media ponderada.

En resumen, la **Arquitectura 1** tiene una latencia considerablemente mayor pero reduce el uso de recursos, mientras que la 'Arquitectura 2' mejora significativamente la latencia, pero con un notable incremento en el uso de Flip-Flops y Look-Up Tables, aunque el impacto general en área es bajo.

	Mejora en latencia (%)	Incremento de FF (%)	Incremento de LUT (%)	Incremento de DSP (%)	Incremento de BRAM (%)	Incremento en área (general) (%)
Arquitectura Base	0	0	0	0	0	0
Arquitectura 1	-3286.67	-88.33	-89.27	0	0	-0.39
Arquitectura 2	36.82	74.40	212.46	0	0	0.63

Cuadro 5.2: Resultados de la síntesis comparativa en porcentajes

Apéndice de los códigos utilizados en el Acelerador II.2

A.1 Header del algoritmo

```
#ifndef ALGORITMOACELERADOR_ALGORITMO_H
#define ALGORITMOACELERADOR_ALGORITMO_H
#include <iostream>
#include <limits> // Para poner el valor mas grande
#include <ap_fixed.h> // libreria punto fijo
using namespace std;

const int M = 1024;
const int N = 16;

//typedef ap_fixed<9,4> fixed;

typedef struct TSignal{
    ap_fixed<9,4> vec[M];
} TSignal;

typedef struct TTrozo{
    ap_fixed<9,4> vec[N];
} TTrozo;

void findBestMatch(const TSignal &S, const TTrozo &T, int &
    ↪ position, ap_fixed<9,4> &minEnergy);

#endif // ALGORITMOACELERADOR_ALGORITMO_H
```

A.2 Algoritmo completo

```
#include "algoritmo.h"

/**
 Encuentra la posicion de una pieza de señal (T) dentro de una señal mas grande (S) que minimiza el error
 **/
void findBestMatch(const TSignal &S, const TTrozo &T, int &position, ap_fixed<9,4> &minEnergy)
{
        ap_fixed<9,4> diff, currentEnergy;
        ap_fixed<9,4> aux;
    minEnergy = numeric_limits<float>::max(); // Inicializado con el maximo valor posible
    //cout<<"numero inicial"<<minEnergy<<"\n";
    position = -1;

    // Loop por el array S
    for (int i = 0; i <= M - N; i++)
    {
        //suma cuadrados
        currentEnergy = 0.0;

        // Calcula la energia del error para esta posicion
        for (int n = 0; n < N; n++)
        {
            diff = S.vec[i + n] - T.vec[n];
            //cout<<"S.vec[i+n]: "<<S.vec[i+n]<<"\n";
            //cout<<"T.vec[n]: "<<T.vec[n]<<"\n";
            //cout<< "diff: "<<diff<<"\n";
            aux = diff * diff;
            //cout<<"aux: "<<aux<<"\n";
            currentEnergy = currentEnergy + aux;
            //cout<< "Suma energías= "<< currentEnergy <<"\n";
        }
```

```
32
33
34
35          // Si la energia actual es menor que la minima, la
               ↪ actualizo
36          if (currentEnergy < minEnergy)
37          {
38               minEnergy = currentEnergy;
39               position = i;
40          }
41      } // end for
42  } // end findBestMatch(..)
```

A.3 Test Bench

```
1   #include "algoritmo.h"
2   int main()
3   {
4       // Inicializado con unos valores random ideales como PoC (
           ↪ Proof of Concept) con un error de 0.
5       int position;
6       ap_fixed<9,4> minEnergy;
7       int contadorCorrectos=0;
8       int contadorIncorrectos=0;
9
10      // TSignal
11      TSignal S;
12      // TTrozo
13      TTrozo T;
14      int posicion;
15      ap_fixed<9,4> energia;
16      float randomFloat;
17
18  // inicializamos la entrada
```

```
19 bucleGrande:
20     for (int i = 0; i < 1000; i++)
21     {
22     bucleEntrada:
23 
24         //cout<<"array de entrada:"<<"\n";
25         for (int i = 0; i < M; i++)
26         {
27             randomFloat = (rand()) / static_cast<float>(
                   ↪ RAND_MAX);
28             randomFloat = randomFloat * 2.0 - 1.0;
29             S.vec[i] = randomFloat;
30             //cout<<"Números aleatorios): "<< S.vec[i]<<"
                   ↪ posición: "<<i <<"\n";
31         }
32 
33         int j = rand() % 1008 + 0;
34         //cout<<"posición de inicio del trozo = "<< j<<"\n";
35     bucleMuestra:
36                 //cout<<"trozo:" <<"\n";
37                 int z=0;
38         for (int i = j; i < j+N; i++)
39         {
40             T.vec[z] = S.vec[i];
41             //cout<<"T.vec[i] parte main"<<T.vec[i]<<"\n";
42             z++;
43         }
44 
45         findBestMatch(S, T, posicion, energia);
46 
47         if (energia < 0.5 && (posicion < 1008 && posicion >=
               ↪ 0))
48         {
49             cout << "---------------------------------- \n";
50             cout << "TEST NUMERO " << i << " CORRECTO \n";
```

```
51              cout << "---------------------------------- \n";
52              contadorCorrectos++;
53          }
54          else
55          {
56              cout << "---------------------------------- \n";
57              cout << "TEST NUMERO " << i << " INCORRECTO \n";
58              cout << "Energia mínima: "<<energia<<"\n";
59              cout << "---------------------------------- \n";
60              contadorIncorrectos++;
61          }
62
63      }
64
65      cout << "---------------------------------- \n";
66      cout << "HAY " << contadorCorrectos << " TEST CORRECTOS \n
           ↪ ";
67      cout << "TEST NUMERO " << contadorIncorrectos << " TEST
           ↪ INCORRECTOS \n";
68      cout << "---------------------------------- \n";
69  }
```

Índice de cuadros

Índice de figuras