Estudio de la influencia de la arquitectura y de la organización interna en el rendimiento

Diseño de Procesadores

Carlos Pérez Fino alu0101340333@ull.edu.es
Cheuk Kelly Ng Pante alu0101364544@ull.edu.es

12 de julio de 2024

Índice general

1. Introducción

The aim of this project is to address the hardware implementation of a processor, in this case the Nios II, which optimises performance in solving a specific problem, in this case that of applying a convolution filter on an original image. in the resolution of a specific problem, in this case that of applying a convolution filter on an original image.

To start the project, a previous activity was the tutorial on the development of Nios II, and once the tutorial was finished, we proceeded to the realisation of the project. tutorial was completed, some test programs were created in order to familiarise ourselves with the development environment and with the Nios II processor. After testing that the development environment and the processor work correctly, we proceeded to improve the design by measuring the performance of each improvement. by measuring the performance of each improvement. It was found that introducing different types of memory took about the same amount of time for the different enhancements. for the different improvements. To check the times, the following program was run:

```
#include <stdio.h>
#include "alt_types.h"
#include "sys/alt_timestamp.h"

#define PALABRAS 1024

int memoria[PALABRAS];

void escritura(char* mem_base, char dato) {
  alt_u32 time1;
  alt_u32 time2;
  int i;
  if (alt_timestamp_start() < 0) {
    printf("No timestamp device available\n");
  } else {
    time1 = alt_timestamp();
    for (i = 0; i < PALABRAS; i++) {
      dato = *mem_base;
      mem_base++;
    }
    time2 = alt_timestamp();
    printf("time memory = %u\n", (unsigned int)(time2 - time1));
  }
}

int main(void) {
  register dato = 0;

  int i;
  printf("\nsram\n\n");
  int* buffer = memoria;
  for (i = 0; i < 4; i++) {
    escritura(buffer, dato);
  }

  buffer = (int*)0x01108000;
  printf("\non chip\n\n");
  for (i = 0; i < 4; i++) {
```

```
    escritura(buffer, dato);
  }

  buffer = (int*)0x00800000;
  printf("\nsdram\n\n");
  for (i = 0; i < 4; i++) {
    escritura(buffer, dato);
  }

  return 0;
}
```

2. Desarrollo del filtro de convolución, filtro Gaussiano

Para el desarrollo del filtro de convolución se ha utilizado el filtro Gaussiano. El filtro Gaussiano es un filtro que se utiliza para suavizar una imagen. Se ha utilizado este filtro porque es un filtro que se utiliza mucho en el procesamiento de imágenes y es un filtro que se puede aplicar a cualquier imagen. El filtro Gaussiano se aplica a una imagen para suavizarla y eliminar el ruido de la imagen.

2.1. Implementación del filtro Gaussiano

Para la implementación del filtro Gaussiano se ha utilizado el siguiente código:

```
int kernel[3][3] = {{1, 2, 1},
                    {2, 4, 2},
                    {1, 2, 1}};

void gaussian_filter(char* input_image, char* output_image) {
  int row, col;
  unsigned char** data = pgmread2(input_image, &row, &col);
  if (!data) {
    printf("Error abriendo imagen\n");
    return;
  }

  unsigned char** output = (unsigned char**)malloc((row - 2) * sizeof(unsigned char*));
  int i, j, k, l;
  for (i = 0; i < row - 2; i++) {
    output[i] = (unsigned char*)malloc((col - 2) * sizeof(unsigned char));
  }

  for (i = 1; i < row - 1; i++) {
    for (j = 1; j < col - 1; j++) {
      int sum = 0;
      for (k = 0; k < 3; k++) {
        for (l = 0; l < 3; l++) {
          sum += data[i - 1 + k][j - 1 + l] * kernel[k][l];
        }
      }
      output[i - 1][j - 1] = sum / 16;
    }
  }

  pgmwrite2(output_image, row - 2, col - 2, output, "Filtro Gaussiano aplicado", 1);
```

```
32
33    for (i = 0; i < row - 2; i++) {
34      free(data[i]);
35      free(output[i]);
36    }
37    free(data);
38    free(output);
39  }
```

Este algoritmo se ha implementado en C y lo primero lo que se hace es definir el kernel del filtro Gaussiano. El kernel del filtro Gaussiano es una matriz de 3x3 que se utiliza para aplicar el filtro a la imagen. Tiene la siguiente forma:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Este kernel se utiliza para aplicar el filtro Gaussiano a la imagen. Para aplicar el filtro Gaussiano a la imagen se recorre la imagen y se aplica el filtro a cada píxel de la imagen. Para aplicar el filtro a un píxel se recorre la matriz del kernel y se multiplica cada elemento de la matriz por el píxel de la imagen correspondiente. Una vez se han multiplicado todos los elementos de la matriz del kernel por el píxel de la imagen se suman todos los resultados y se divide entre 16. El resultado de esta operación es el nuevo valor del píxel de la imagen que se está procesando.

La función `gaussian_filter` recibe como parámetros la imagen de entrada y la imagen de salida. La imagen de entrada es la imagen original que se quiere suavizar y la imagen de salida es la imagen suavizada. Esta función lee la imagen de entrada y crea una imagen de con la función `pgmread2`. Una vez se ha leído la imagen de entrada lo que se hace es reservar memoria para la imagen de salida, que tendrá dimensiones (row - 2) x (col - 2) debido al borde que no se puede procesar completamente. Luego, reserva memoria para cada fila de la imagen de salida. A continuación, se hace la convolución de la imagen con el kernel del filtro Gaussiano. Para ello, recorre cada pixel excluyendo los bordes de la imagen, excluyendo los bordes de (1 a row - 1) y (1 a col - 1). Para cada pixel en la posición (i, j), calcula la suma de la multiplicación de cada pixel de la imagen con el kernel del filtro Gaussiano. Finalmente, divide la suma entre 16 y guarda el resultado en la imagen de salida 'output'.

A continuación, se escribe la imagen de salida en un archivo con la función `pgmwrite2`. Esta función es la encargada de escribir la imagen en un archivo con el formato PGM y se le pasa como parámetros el nombre del archivo, las dimensiones de la imagen y la imagen en sí. Finalmente, se libera la memoria de la imagen de entrada y de la imagen de salida.

3. Estudios y resultados

A la hora de realizar el estudio, se ha optado por hacer cinco pruebas diferentes para poder comparar los resultados y ver cuál es la mejor opción para mejorar el rendimiento del procesador. Para hacer las pruebas los tiempos se han obtenido haciendo un promedio de 5 ejecuciones de cada prueba. Las pruebas que se han realizado son las siguientes:

Prueba	**Tiempo obtenido en segundos**
Sin cambios	18.96
CPU con mayores prestaciones (Nios II/f)	18.98
Hardware divide	18.92
Nios II/f y Hardware divide	18.97
Caché para datos	18.90

3.1. CPU con mayores prestaciones

Analizando los resultados obtenidos, al cambiar la CPU a uno con mayores prestaciones (Nios II/f) no obtenemos mucha diferencia pero observamos que tarda un poquillo más, por algún motivo que desconocemos. En cambio, en la ejecución del programa si es verdad que la diferencia es casi imperceptible pero si mejor algo.

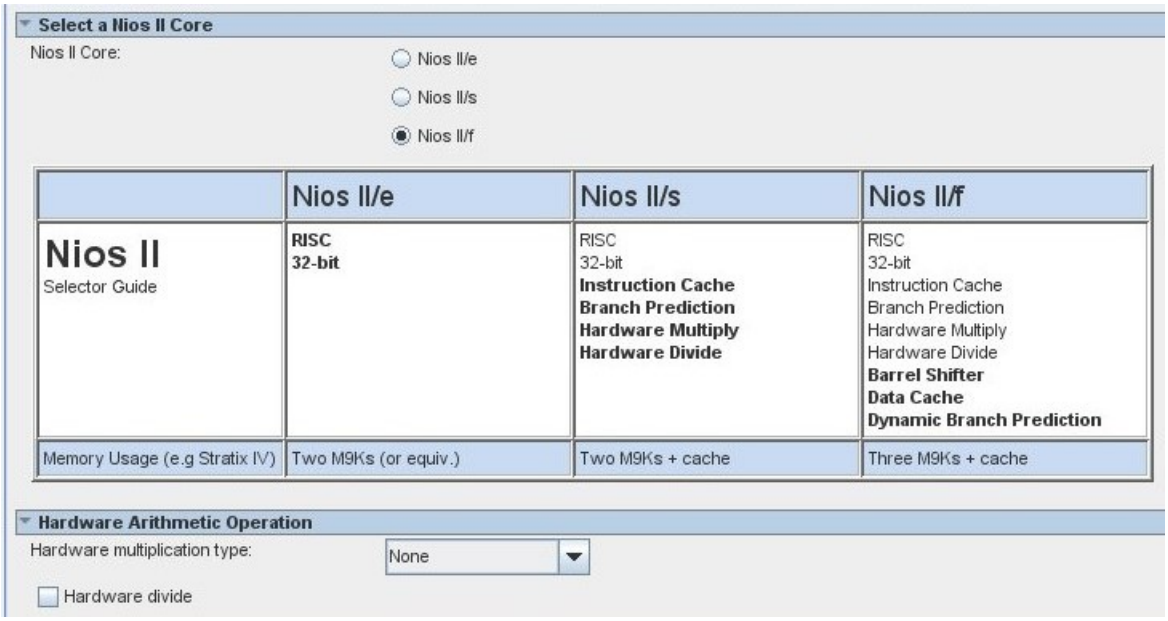

Figura 3.1: Configuración de la CPU con mayores prestaciones

3.2. Hardware para multiplicación y división

En la siguiente prueba se ha añadido hardware para la multiplicación y la división. En esta prueba se ha obtenido un tiempo de 18.92 segundos, lo que supone una mejora de 0.04 segundos respecto a la prueba anterior. Aunque la mejora no es muy significativa, se puede observar que el hardware para la multiplicación y la división ha mejorado el rendimiento del procesador.

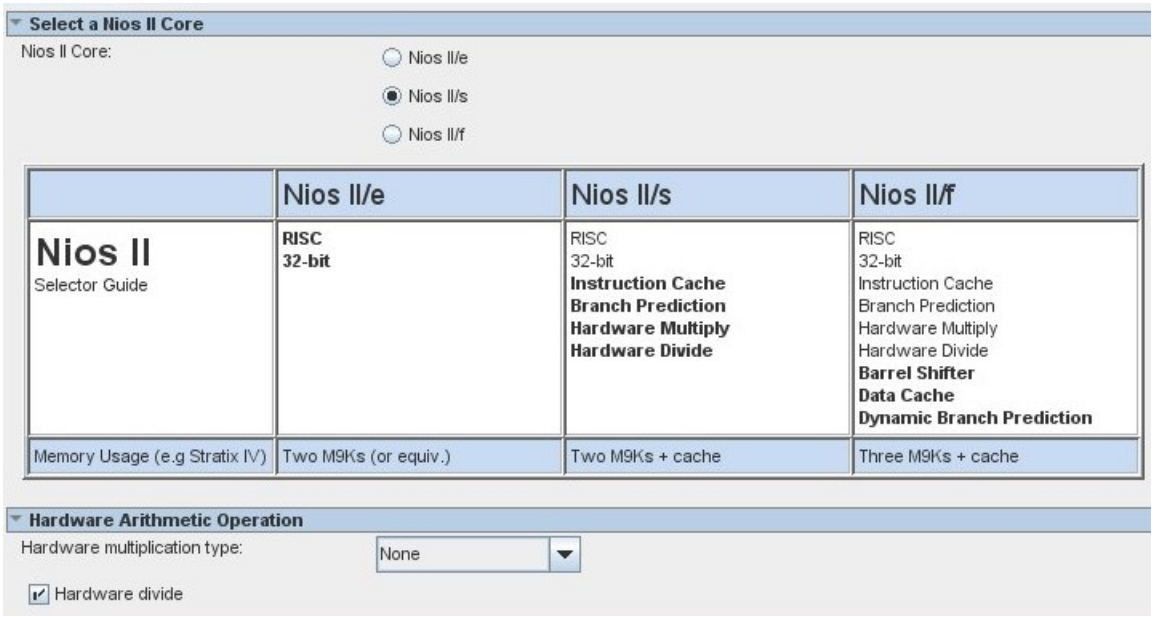

Figura 3.2: Configuración del hardware para la multiplicación y la división

3.3. CPU con mayores prestaciones y hardware para multiplicación y división

En la siguiente prueba se han combinado las dos mejoras anteriores, es decir, se ha cambiado la CPU a una con mayores prestaciones y se ha añadido hardware para la multiplicación y la división. En esta prueba ha supuesto un empeoramiento respecto a la prueba original. No sabemos por qué ha ocurrido esto, pero se puede deber a que la CPU con mayores prestaciones no es compatible con el hardware para la multiplicación y la división.

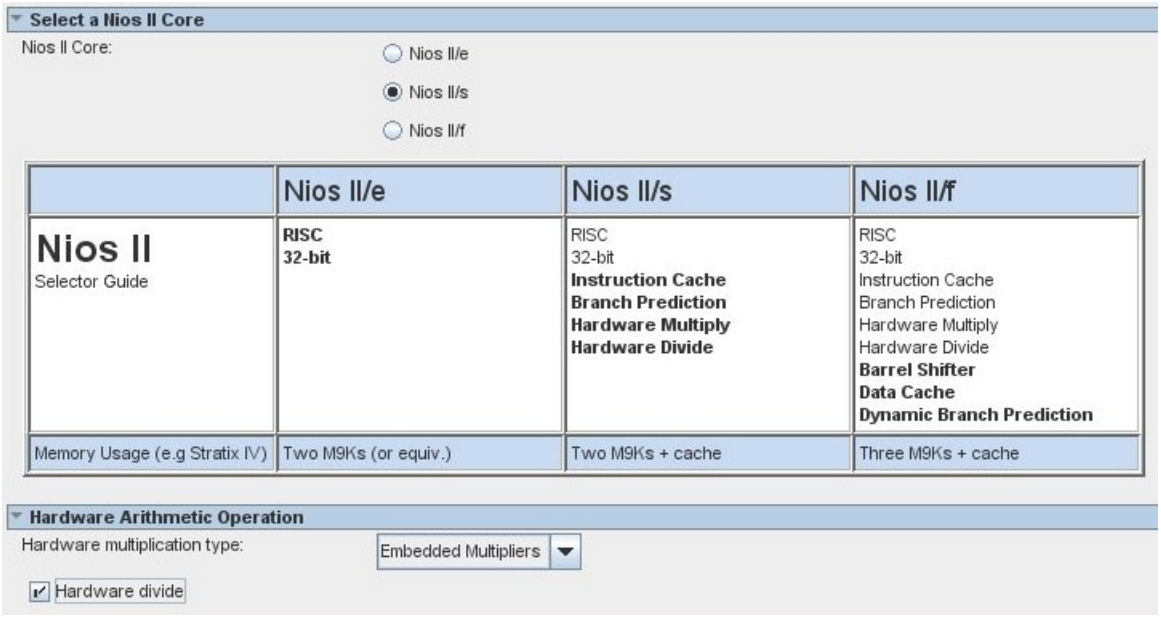

Figura 3.3: Configuración de la CPU con mayores prestaciones y hardware para la multiplicación y la división

3.4. Caché para datos

En la última prueba se ha añadido una caché de instrucciones de datos. En esta prueba se ha obtenido un tiempo de 18.90 segundos, lo que supone una mejora de 0.06 segundos respecto a la prueba original. Aunque la mejora no es muy significativa, se puede observar que la caché de instrucciones de datos ha mejorado el rendimiento del procesador.

Primero se ha asignado el tamaño de la caché de datos, llevando a un conflicto con la memoria onchip. Para solucionar este problema, se redujo el tamaño de la memoria onchip pasando de los 20KB a 10KB.

Haciendo esto permitía crear una caché de unos 8KB, lo cual tuvo un impacto positivo en el rendimiento del procesador y se redujo un poco el tiempo de ejecución.

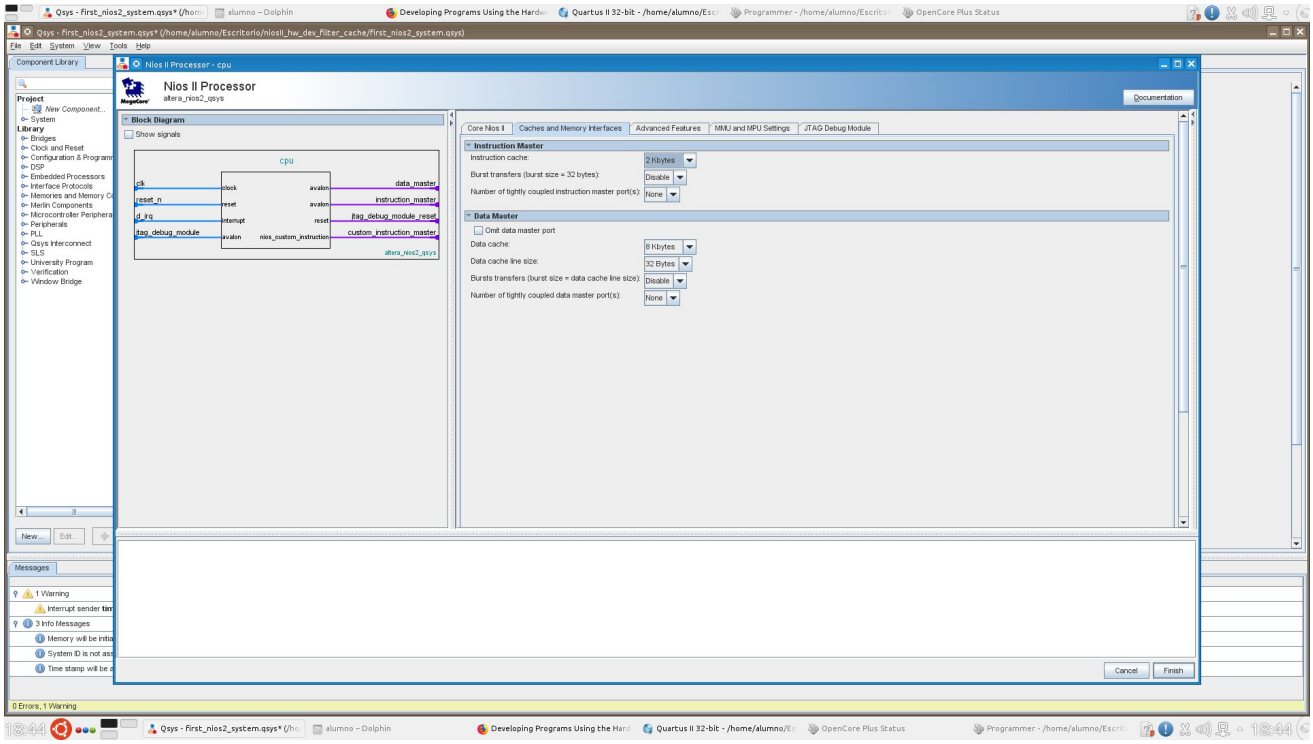

Figura 3.4: Configuración de la caché de datos

4. Tightly coupled memory

Como útlima mejora se ha añadido una memoria tightly coupled memory. La tightly coupled memory es una memoria que se conecta directamente al procesador y se utiliza para almacenar datos que se utilizan con frecuencia.

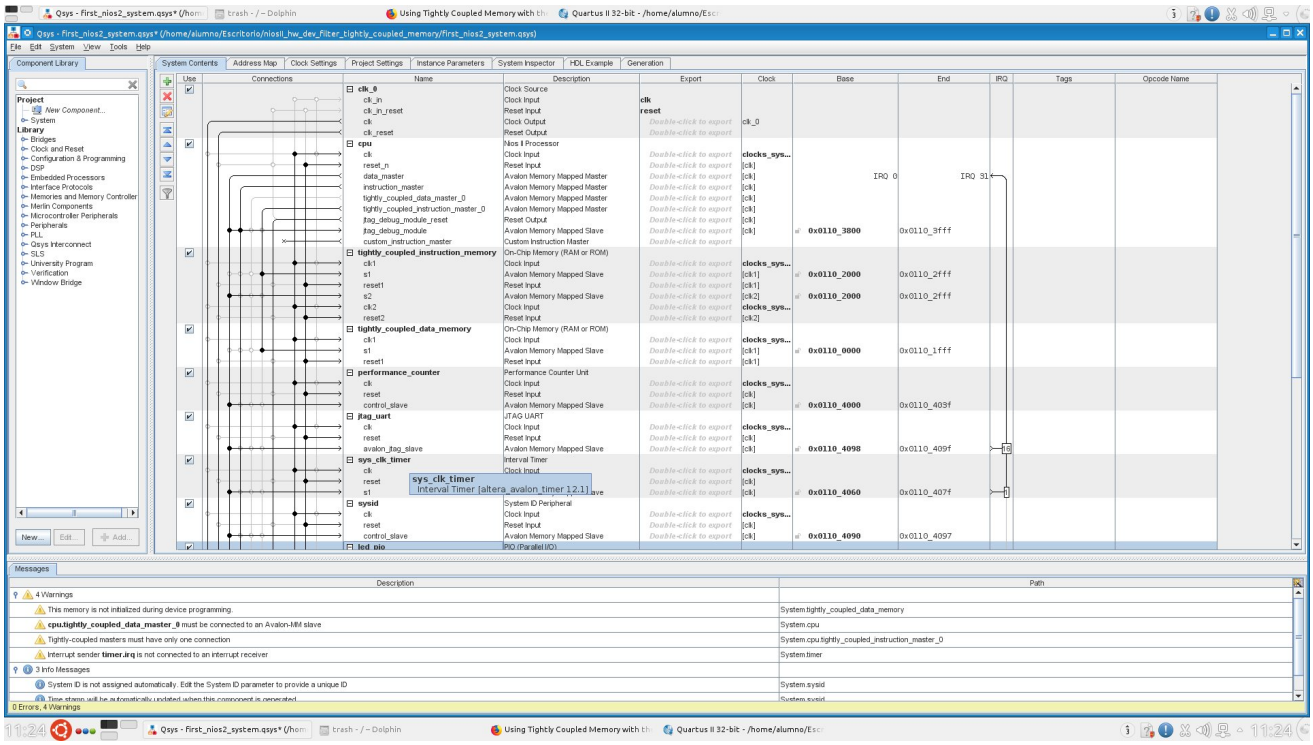

Figura 4.1: Configuración de la tightly coupled memory

Se ha añadido los módulos de memoria habría que programar el modelo y hacer la mediciones de tiempo para ver si ha mejorado el rendimiento del procesador. No obstante, no se ha podido realizar esta mejora debido a problemas con la implementación de la tightly coupled memory. Se ha seguido el tutorial de Altera para la implementacion de esta memoria pero no se ha podido realizar correctamente. Uno de los problemas era abrir la terminal de Nios II y localizar donde se encontraba.

5. Conclusiones

En este proyecto ha servido para aprender a utilizar el procesador Nios II y a mejorar su rendimiento. Se ha realizado un estudio para ver cómo mejorar el rendimiento del procesador y se han realizado varias pruebas para comparar los resultados. Se ha observado que la caché de instrucciones de datos ha mejorado el rendimiento del procesador y se ha obtenido una mejora de 0.06 segundos en el tiempo de ejecución.

Por otro lado a la hora de realizar las mediciones, los resultados no eran los esperados por el resultado dado del `clock()` ya que al realizar 5 repeticiones para aplicar el filtro estabamos aproximadamente 12 minutos esperando a que se ejecutara el programa y luego poniamos temporizadores al inicio final de la ejecucion del programa y daba tiempos de 25 segundos aproximadamente. Por lo que no se ha podido realizar correctamente las mediciones de tiempo pero se ha podido observar que con algunas mejoras se ha podido mejorar el rendimiento del procesador.

Además, otros problemas que se han encontrado a lo largo del proyecto han sido con el propio Quartus, al ser una version del programa bastante antiguo y no tener mucha experiencia con el programa y con el poco soporte online, ha sido complicado realizar algunas tareas. También, había veces que se hacía pequeños cambios en la configuración del proyecto BSP y saltaban errores que no se sabía por qué ocurrían y habia que empezar de nuevo.