Diseño e Implementación de un ASIP para interpolación de imágenes

Jose Ignacio Granados
Juan Ignacio Navarro
Jose David Sánchez
Mónica Waterhouse

19 de mayo de 2022

Índice

1. Introducción

Este documento corresponde a la explicación de los principales aspectos de diseño e implementación de un procesador que sea capaz de realizar la interpolación de imágenes. El objetivo del proyecto corresponde a realizar la arquitectura y microarquitectura desde cero de tal forma que se pueda realizar el análisis a la imagen. A continuación se presentará con más detalle los requerimientos del sistema, las soluciones propuestas, su respectiva comparación y la solución final implementada.

2. Requerimientos del sistema

Para el desarrollo de este sistema se especificaron sus requerimientos previo a su desarrollo. Estos corresponden a los aspectos a considerar como resultado del sistema y que serán útiles para conocer si se hizo una correcta implementación. A continuación se detallan estos requerimientos y su resultado esperado:

1. Se debe elaborar un script en lenguaje de alto nivel que ejecute el algoritmo de interpolación bilineal que genere interpolación de imágenes. Esto se hace para justificar las características del ISA. De esta manera se puede analizar cuáles instrucciones son necesarias y cuáles operaciones se pueden ejecutar a base de otras.

2. Elaborar una Green Sheet para que se identifiquen las proporciones y divisiones de cada instrucción en cuanto a bits asociados a registros, códigos de operación, funciones e inmediatos.

3. Elaborar un Memory Map que contenga las divisiones respectivas de la memoria tomando en cuenta aspectos como la cantidad de instrucciones, cantidad de datos de lectura y la cantidad total de datos de escritura.

4. Se debe construir un diagrama de bloques especificando todas las partes por las que va a ser compuesta la microarquitectura del proyecto. Tomar en cuenta que este requerimiento se debe desarrollar en conjunto con los requerimientos anteriores para especificar módulos, entradas, salidas y sus tamaños basándose en un procesador pipeline.

5. Elaborar un código ensamblador para la arquitectura diseñada con el set de instrucciones específico para el procesador que se va a desarrollar. La idea es hacer este requerimiento basándose en el script del lenguaje de alto nivel previamente mencionado.

6. Se debe desarrolla un compilador capaz de interpretar un archivo de entrada conteniendo el algoritmo de interpolación elaborado en el lenguaje de ensamblador. Dicho compilador debe tomar como base las instrucciones definidas del ISA en la Green Sheet.

7. Elaborar un procesador con microarquitectura pipeline programado en Quartus Prime utilizando el lenguaje HDL llamado System Verilog para crear todos los módulos necesarios especificados en el diagrama de la microarquitectura.

8. Una vez completa la microaquitectura se debe hacer un reporte de consumo de recursos del FPGA para saber aproximadamente qué tan caro es para el hardware procesar el código de ensamblador y ejecutar el procesador en general.

9. Se debe crear un pequeño programa en alto nivel que utilice los datos generados por el procesador de Quartus Prime para elaborar la imagen procesada y que es requerida en la etapa final de visualización.

Además, la implementación necesita de otro script para su funcionamiento. Este es el que se encarga del almacenamiento de los datos de la imagen e instrucciones en memoria.

3. Algoritmo de interpolación

Para este proyecto se va a desarrollar un proceso de interpolación bilineal. La idea es implementar un algoritmo que de cierta manera le haga una especie de zoom a un sector de una imagen original sin duplicar píxeles. Dicho algoritmo va a constar de tres grandes partes las cuales se muestran a continuación.

3.1. Espaciado entre píxeles originales

La idea con esta etapa es procesar los datos de la imagen original para generar una cuadrícula con espacios en blanco entre dichos píxeles. En las etapas posteriores se va a proceder a calcular el valor que se le va a asignar a cada espacio en blanco. En la figura 1 se puede observar el agrandamiento de la cuadrícula original rellenando entre píxeles con dichos espacios en blanco.

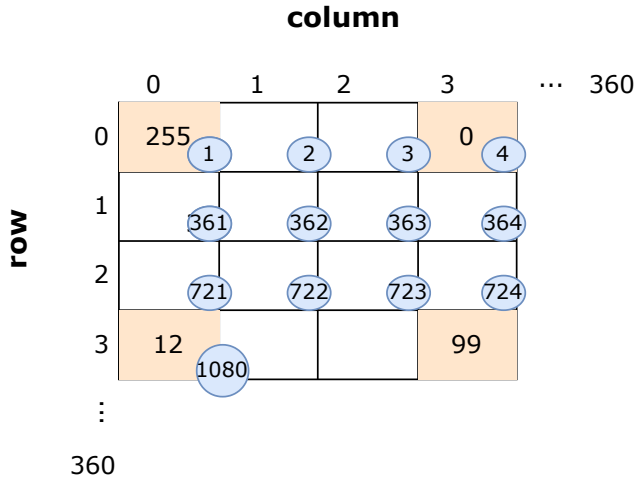

Figura 1: Agrandamiento de la cuadrícula original mediante el uso de espacios en blanco.

3.2. Cálculo de los píxeles en las columnas

Para este paso, se va a calcular los píxeles de la primera y última columna de cada sector que fue separado. Como se observa en la figura 2, dichos píxeles que se van a calcular son los asignados con las letras a, b, c y d en ese orden respectivamente.

3.3. Cálculo de los píxeles en las filas

Para esta tercera y última etapa del algoritmo implementado en ensamblador, se deben calcular los valores de las casillas restantes. Dichas casillas se pueden observar en la figura 3 cuyas letras asignadas son f, g, h, i, j, k, l y m en ese orden respectivamente.

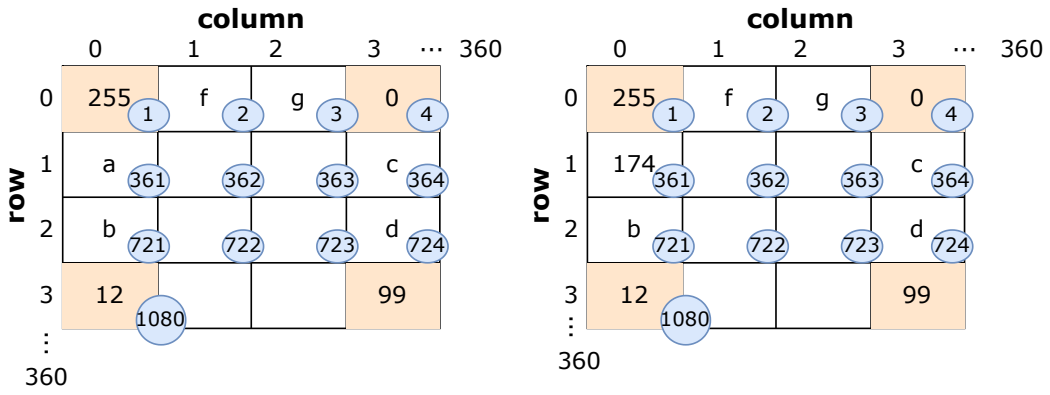

Figura 2: Cálculo de los píxeles en las columnas.

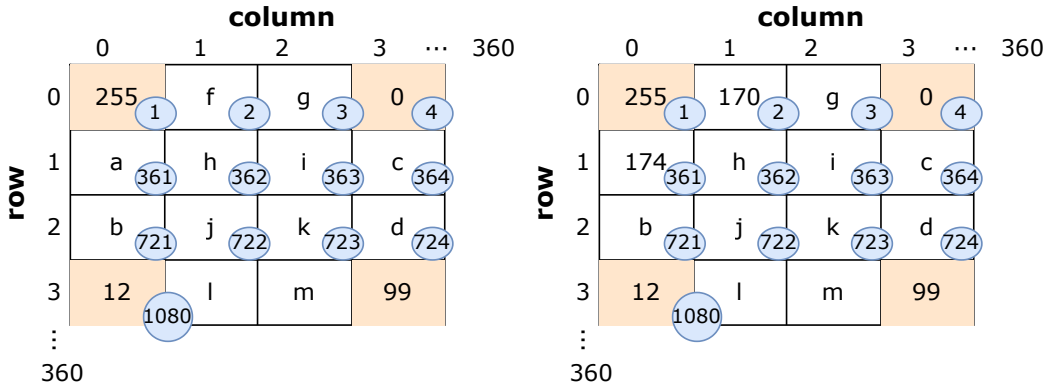

Figura 3: Cálculo de los píxeles en las filas.

4. Posibles soluciones

4.1. Soluciones en relación con el diseño de la arquitectura

1. Primer diseño de arquitectura

   El primer set de instrucciones propuesto se muestra en la figura 4 y en el cual, se pueden destacar las siguientes características:

   - Cada una de las instrucciones es de 20 bits.
   - Se poseen 5 tipos de instrucciones diferentes.
   - Para las instrucciones de control y memoria, se emplea un opcode de 2 bits, mientras que, en las instrucciones de datos, se utiliza un opcode de 5 bits.
   - En cada tipo de instrucción, los registros fuente y destino se encuentran colocados en diferentes posiciones de bits.
   - Existen diversos tamaños de valores inmediatos entra las instrucciones de control, memoria y datos con inmediato.

Control (condicional)					
tipo	OPCODE		Reg 1 (Src)	Reg 2 (Src)	dir. Tag
	branch	ins			
19-18	17	16	15-13	12-10	9-0

Control (incondicional)			
tipo	OPCODE		Inmediato
	branch	ins	
19-18	17	16	15-0

Memoria				
tipo	OPCODE	Reg 1 (Destino)	Reg 2 (Src)	Inmediato
	Instrucción			
19-18	17	16-14	13-9	10-0

Datos (sin inmediato)					
tipo	OPCODE		Reg 1 (Destino)	Reg 2 (Src)	Reg 3 (Src)
	Flag Inm	ins			
19-18	17	16-13	12-9	8-5	4-0

Datos (con inmediato)				
tipo	OPCODE		Reg 1 (Destino)	Inmediato
	Flag Inm	ins		
19-18	17	16-13	12-10	9-0

Figura 4: Primera propuesta del set de instrucciones.

2. Segundo diseño de arquitectura

   El segundo set de instrucciones propuesto se muestra en la figura 5 y en el cual, se pueden destacar las siguientes características:

   - Cada una de las instrucciones es de 32 bits.
   - Se poseen 5 tipos de instrucciones diferentes.
   - Para las instrucciones de control se emplea un opcode de 2 bits.
   - Para las instrucciones de memoria se emplea un opcode de 4 bits.
   - Para las instrucciones de datos se emplea un opcode de 5 bits.
   - En cada tipo de instrucción, los registros fuente y destino se encuentran colocados en diferentes posiciones de bits.
   - Existen diversos tamaños de valores inmediatos entra las instrucciones de control, memoria y datos con inmediato.
   - Las instrucciones de memoria y datos sin inmediato, requieren de un relleno de 3 y 13 bits respectivamente para completar los espacios restantes.

Control (condicional)					
tipo	OPCODE		Reg 1 (Src)	Reg 2 (Src)	dir. Tag
	branch	ins			
31-30	29	28	27-24	23-20	19-0

Control (incondicional)			
tipo	OPCODE		Inmediato
	branch	ins	
31-30	29	28	27-0

Memoria					
tipo	OPCODE		Reg 1 (Destino)	Reg 2 (Src)	Inmediato
	Instrucción	Relleno			
31-30	29	000	25-22	21-18	17-0

Datos (sin inmediato)						
tipo	OPCODE		Reg 1 (Destino)	Reg 2 (Src)	Reg 3 (Src)	Relleno
	Flag Inm	ins				
31-30	29	28-25	24-21	20-17	16-13	12-0

Datos (con inmediato)					
tipo	OPCODE		Reg 1 (Destino)	Reg 2 (Src)	Inmediato
	Flag Inm	ins			
31-30	29	28-25	24-21	20-17	16-0

Figura 5: Segunda propuesta del set de instrucciones.

4.2. Soluciones en relación con la cantidad de registros

1. Total de 8 registros

   La primera cantidad de registros propuesta se muestra en la figura 6 y en la cual, se pueden destacar las siguientes características:

   - Al ser 8 registros en total, cada uno de ellos es de 3 bits.
   - El registro R0 siempre tendrá un valor fijo que no puede ser reescrito.
   - Se disponen de 5 registros de propósito general.
   - El registro RR contiene la dirección del PC respecto a la última función llamada.
   - El registro RS se encarga de llevar el puntero de la memoria del stack.

2. Total de 16 registros

   La segunda cantidad de registros propuesta se muestra en la figura 7 y en la cual, se pueden destacar las siguientes características:

   - Al ser 16 registros en total, cada uno de ellos es de 4 bits.
   - El registro R0 siempre tendrá un valor fijo que no puede ser reescrito.
   - Se disponen de 15 registros de propósito general.

R0	Su valor siempre es cero
R1	Registros de propósito general
R2	
R3	
R4	
R5	
RR	Registro para retornar a una dirección de una función
RS	Registro que guarda la dirección del stack

Figura 6: Propuesta inicial utilizando 8 registros.

R0	Registro cuyo valor siempre es cero
R1	Registros de propósito general
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

Figura 7: Propuesta final utilizando 16 registros.

4.3. Soluciones en relación con el uso de la memoria

1. Utilización de los módulos de memoria de Quartus Prime

   Una de las funcionalidades brindadas por la aplicación de Quartus Prime es la creación de memorias RAM y ROM dentro del proyecto que se esté desarrollando. Para el caso de la microarquitectura previamente diseñada, estos módulos son necesarios para almacenar las instrucciones que se deben probar, la imagen a procesar y la respectiva imagen procesada. Es decir, se debe guardar el código a ejecutar y los píxeles de ambas imágenes.

La memoria de instrucciones ha sido asignada con 400 espacios, la imagen de entrada consume 8100 espacios de memoria (90x90 píxeles) y la imagen de salida requiere de 129600 espacios de memoria (360x360 píxeles). Sin embargo, cada uno de los módulos, mencionados anteriormente, son capaces de almacenar únicamente 65535 datos en su interior. Por lo que, no es posible guardar toda la información requerida. Por esta razón, se tomó la decisión de almacenar 4 datos en un solo espacio de memoria, reduciendo así la cantidad de información por guardar. Esta propuesta implica realizar una modificación en las instrucciones de memoria, dado que se debe conocer en cuál sección del espacio de memoria, se debe almacenar el dato en cuestión. Dicha corrección se muestra en la figura 8.

Memoria					
tipo	OPCODE		Reg 1 (Destino)	Reg 2 (Src)	Inmediato
	Instrucción	Sección			
31-30	29	28-26	25-22	21-18	17-0

Figura 8: Distribución de bits en la instrucción de memoria en la propuesta inicial.

2. Utilización de archivos .txt como módulos de memoria

   Tal y como su nombre lo indica, se utilizarán archivos externos a Quartus Prime, en formato .txt, para almacenar cada una de las instrucciones a ejecutar, los pixeles de la imagen a procesar y los pixeles de la imagen procesada. Como se mencionó anteriormente, se requieren de 400 espacios memoria para las instrucciones, 8100 espacios de memoria para la imagen de entrada y 129600 espacios de memoria para la imagen de salida, dando un total de 138100 datos por almacenar. Dicha división de espacio se aprecia en la imagen a continuación.

   Los archivos en formato .txt técnicamente no poseen un límite de almacenamiento, solo el que el sistema de archivos les provee. Sin embargo, el valor máximo es mucho más grande que la cantidad de datos calculada anteriormente. Por lo que, es posible almacenar cada valor por separado dentro del archivo en cuestión.

5. Comparación de soluciones

5.1. Comparación de las soluciones en relación con el diseño de la arquitectura

La principal diferencia entre ambos sets de instrucciones es la cantidad de bits que cada una de ellas requiere. Por un lado, se utilizan de 20 bits y el por el otro, se emplean 32 bits. Sin embargo, la arquitectura de instrucciones más amplias (solución 2) permite almacenar valores inmediatos más grandes, dado que se disponen de más bits para dicha información. A su vez, la cantidad de registros se amplía debido a que se asigna un bit de más, permitiendo así, duplicar la cantidad de los mismos. Además, se presenta una mayor simetría entre los diversos tipos de instrucciones. No obstante, la arquitectura de instrucciones de menor tamaño (solución 1) no requiere de bits de relleno (0's) para cumplir con los requerimientos de longitud mientras que el otro set, necesita dicho ajuste en las instrucciones de memoria y datos sin inmediato.

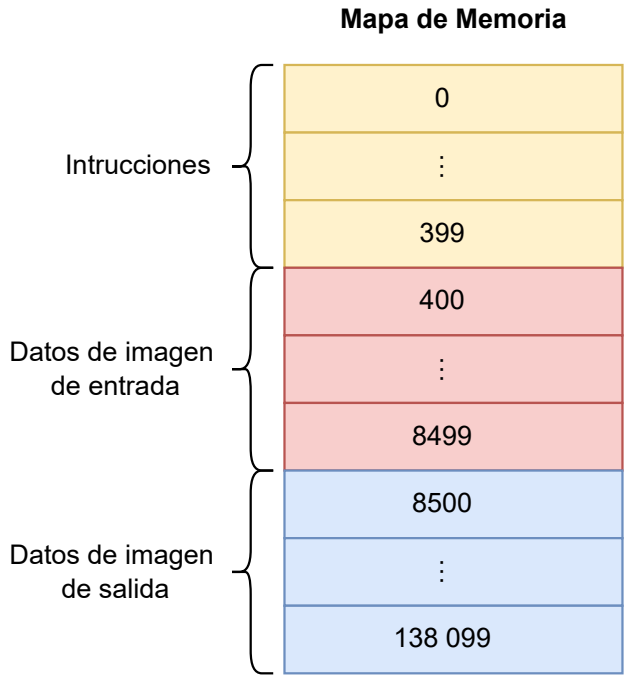

Figura 9: Mapa de memoria de la propuesta final.

5.2. Comparación de las soluciones en relación con la cantidad de registros

En este caso particular, la propuesta de menor tamaño (solución 1) presenta 4 tipos de registros: fijo (R0), propósito general (R1-R5), retorno (RR) y stack (RS). Dicho set implica el diseño de instrucciones complejas de control y memoria. Por el otro lado, la propuesta de mayor tamaño (solución 2) presenta la mitad de tipos de registros: fijo (R0) y propósito general (R1-R15), permitiendo así una mayor libertad en el almacenamiento de datos temporales y una considerable simplicidad en las instrucciones por diseñar.

5.3. Comparación de las soluciones en relación con el uso de la memoria

La implementación de la primera propuesta, puede considerarse relativamente más complicada que la segunda debido a las siguientes razones:

- Utilizar la memoria por defecto de la aplicación de Quartus Prime, involucra una revisión más profunda y cuidadosa de los resultados arrojados por la simulación debido a que al almacenar 4 valores en un solo espacio de memoria, se generarán datos muy grandes los cuales pueden llegar a generar malas interpretaciones de los mismos mientras que, en los archivos con formato .txt, dicha problemática no se ocasionará dado que cada uno de los pixeles se almacenarán por separado, evitando así, cualquier complicación en la lectura de dicha información.

- Almacenar 4 valores en un único espacio de memoria involucra el desarrollo de una unidad aritmética lógica (ALU) capaz de diferenciar las secciones de dicho bloque de almacenamiento. Así como también, el diseño de una arquitectura que pueda realizar dichas distinciones mientras que, en la otra solución, la situación actual no se presentará.

- Al momento realizar la simulación con las memorias RAM y ROM de Quartus Prime, los resultados de dichas ejecuciones no son almacenados en los módulos mencionados debido que el proyecto no se ejecuta directamente en una FPGA. Por lo que, dicho instrumento de hardware es necesario para conservar los datos de salida. Por el contrario, la segunda solución es capaz de almacenar toda la información de salida por medio de la ejecución de las pruebas.

6. Propuesta final

Para comenzar con el desarrollo del proyecto, se decidió implementar el algoritmo de interpolación en Python (el funcionamiento de este algoritmo se puede observar con detalle en la sección 3) de tal forma que se pudiera comprobar de una manera más sencilla el correcto funcionamiento del algoritmo y de igual forma poder tener una base para crear el código ensamblador, con el cual, se pudieron determinar las instrucciones necesarias para que el procesador desarrollado por el grupo sea capaz de realizar la interpolación de imágenes. Basándose en las instrucciones identificadas en el código ensamblador se pudo elaborar la arquitectura del sistema, la cual, estaría compuesta por un set de instrucciones que incluye cinco tipos diferentes: de control (condicionales y no condicionales), de memoria y de datos (con y son inmediato). Cada una de estas instrucciones estaría formada por 32 bits que corresponden al tipo de instrucción, OPCODE, registros fuente y destino, así como el valor de los inmediatos en caso de que vayan a ser utilizados. El detalle del set de instrucciones y de los registros se puede observar en las secciones 4.1.2 y 4.2.2 respectivamente.

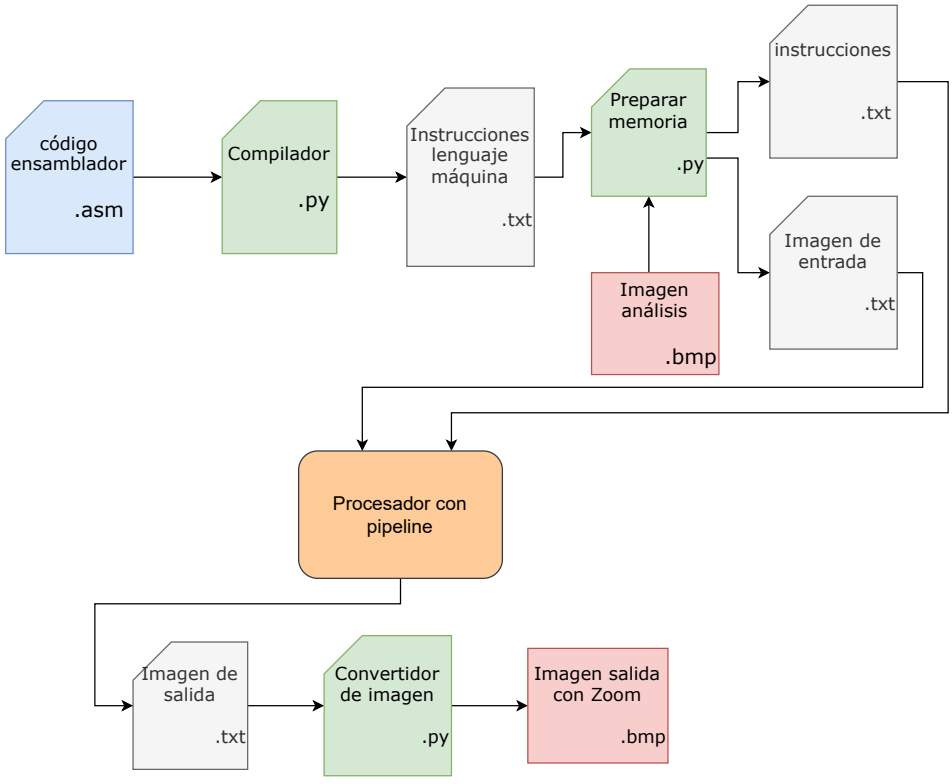

Figura 10: Flujo de información entre los archivos del proyecto.

La figura 10 muestra el flujo de la información entre las diferentes etapas de la solución final del proyecto. Como se puede observar, una vez que se creó el código ensamblador, el cual permitió obtener la arquitectura del procesador, se comenzó a desarrollar el compilador utilizando Python. Este compilador se encarga de tomar las instrucciones del algoritmo en lenguaje ensamblador y crear un archivo .txt con dichas instrucciones en lenguaje de máquina. Para esto, resultaba esencial que la arquitectura del sistema estuviera lista ya que el compilador se basa en esta para la asignación de los valores de cada uno de los 32 bits que conforman una instrucción en lenguaje de máquina. Además, el compilador fue desarrollado de tal forma que se pudieran corregir los riesgos de datos de manera estática mediante el uso de *stalls*.

La memoria utilizada en la microarquitectura es de dos canales, es decir, se utiliza un canal para manejar el bus de instrucciones y otro para manejar los datos. Por lo tanto, se creó un script en Python para preparas la memoria. Este script recibe como entrada el archivo .txt con las instrucciones en lenguaje de máquina así como la imagen que se quiere procesar en formato .bmp para generar dos archivos .txt: uno de instrucciones y otro con los datos de de los pixeles de la imagen de entrada. Como se menciona en la sección 4.3.2, se requiere de 400 espacios memoria para las instrucciones, 8100 espacios de memoria para la imagen de entrada y 129600 espacios de memoria para la imagen de salida por lo que se utilizan archivos en formato .txt para el manejo de memoria ya que, a diferencia de la memoria proporcionada por Quartus Prime, estos no poseen límite de almacenamiento.

El procesador con pipeline recibe como entrada externa los archivos .txt correspondientes a la memoria de instrucciones y la memoria de datos. Este procesador, se desarrolló en Quartus Prime mediante un diseño modular basado en la microarquitectura elaborada por el equipo de trabajo, la cual, puede observarse en el archivo *Microarquitectura.png* anexado a este documento. A grandes rasgos, la microarquitectura está compuesta por los siguientes módulos:

- **PC Register:** Este módulo es un registro que guarda el valor de PC y que como entrada recibe el siguiente, el cual se carga solamente si la entrada **load** está en 1.
- **Multiplexores:** se utilizaron varios multiplexores de 2x1 y 4x1 para poder seleccionar bits específicos de la instrucción para realizar las operaciones y el valor de salida va a depender de las señales obtenidas de la unidad de control según el *OPCODE* de la instrucción que se esté ejecutando.
- **Sign Extend:** este módulo recibe como estrada los bits correspondientes al inmediato. Como se puede observar en el set de instrucciones de la figura 5 dependiendo del tipo de operación se pueden necesitar todos los bits de entrada o un recorte de este dependiendo de la señal de la unidad de control denominada *ImmSrc*. Además, este módulo permite también extender el bit de signo lo cual es importante para las operaciones en la ALU.
- **Sumador:** este es un sumador aparte de la ALU que recibe 2 entradas de 32 bits y se utiliza para sumar el valor del PC actual con un inmediato para los casos en los que se requiere un salto a otra instrucción.

- **Register file:** este módulo se encarga de guardar y obtener los datos de los 15 registros de propósito general. El registro R0 siempre tiene que tener un valor de 0. Además, es importante recalcar que se diseñó de tal forma que se escriben valores en los semiciclos positivos y se leen en los negativos.
- **ALU:** este módulo tiene dos entradas de 32 bits que corresponden a los operandos y una entrada de 3 bits para indicar la operación que se quiere realizar. Este módulo se encarga de realizar operaciones aritméticas especificadas en la arquitectura: suma, resta, multiplicación, división entera y residuo. Además, tiene como salida la bandera de cero en caso de que un resultado tenga como valor 0 (esta se usa principalmente para determinar si se debe hacer un salto condicional cuando los operandos son iguales).
- **Acceso a memoria:** este módulo se encarga de acceder y escribir datos e instrucciones en archivos txt que funcionarán como memoria. Se tienen dos canales para obtener o escribir datos, uno para las instrucciones y otro para los datos. Esto va a permitir que el pipeline obtenga información de las instrucciones y los datos de las imágenes.
- **Control de saltos:** corresponde a una serie de compuertas lógicas que permiten identificar cuando se debe hacer un salto de instrucción.
- **Unidad de control:** este módulo recibe el tipo de instrucción y el *OPCODE* para así poder setear las banderas que controlan los módulos descritos anteriormente.
- **Registros de segmentación:** estos registros se colocan entre cada una de las etapas de las instrucciones para así poder implementar pipeline en el diseño propuesto.

Una vez que se ejecuta el código del procesador, se obtiene como salida un archivo .txt que contiene los pixeles de la imagen de salida en hexadecimal. Posteriormente, este último archivo .txt se procesa en un script de Python que utiliza la librería image.io para leer los valores en hexadecimal de los pixeles en escala de grises de la imagen de salida y la genera en formato .bmp. Esta es una imagen de 360x360 pixeles que representa un zoom de uno de los 16 cuadrantes de la imagen de entrada sin duplicar pixeles.