Implementación de un acelerador para árboles de decisión tipo Gradient Boosting en plataformas RISC-V

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Ángel Rubio Jorge

Trabajo de fin de grado de Ingeniería Informática
Universidad de Zaragoza

Directores del trabajo:
Samuel Pérez Pedrajas y Aldrián Alcolea Moreno
Ponente:
Javier Resano Ezcaray

27 de noviembre de 2024

Abstract

This project explores the design, implementation, and integration of a hardware accelerator based on Gradient Boosting Decision Trees (GBDT) on a RISC-V arquitecture. GBDTs are a popular model used for classification and regression tasks. They provide interpretability and also are really easy to follow up. Unlike other complex models such as neural based networks, GBDTs use simple operations, such as comparisons and additions. This makes them computationally efficient while offering insights into the decision-making process. This study aims to implement an accelerator in a processor which manages and classifies pixels in hyperspectral images using GBDTs. These pixels pass through different nodes inside the trees. Each tree has different features which decide which way the pixel takes, either the right or the left path. Each leaf node contains a prediction, which is added to the ones obtained on other trees. This aggregated prediction allows the program to classify the node in one of the different existing classes.

The study has been divided into several phases. It began by identifying the computational requirements to start the new functional unit. A detailed analysis was made in order to find a commmon instruction pattern in different parts of the code. By finding it, it was possible to make a more exhaustive study on its computational cost. These instructions, were executed in different sections in the original code. The main idea of this project was to fuse these instructions into a single one. By doing this, the computational time is reduced as this new instruction only has to pass through each phase of the processor once instead of three or four, depending on the pattern. This approach aimed to reduce the number of cycles required, thereby improving the execution speed.

The new instruction and its corresponding functional unit were designed and implemented in a 32-bit RISC-V 5-staged processor. This new hardware unit, combines bitwise logic and, shifting and addition all in once. It also allows to choose different constants from a predefined set to use. The unit has been added to the processor in the execution stage, creating a new module and modifying the control signals in order to work properly. It was created using SystemVerilog. This unit has passed several validation phases. The first one consisted on simple test cases which did not involve the whole processor, only the new module. Simple trees were used in order to invest less time in the initial validation case. During the second and last phase, the hyperspectral images, which are IP_data, PU_data and KSC_data, were proven. This images contain thousands of pixels and nodes. This phase involved the entire processor as it was necessary to check the correct attachment of the new unit into the processor.

Performance results have proved a significant improvement in the execution cicle-time. The processor with the new unit attached achieved a speedup of up to 27 % compared to the original processor. Also, a hardware cost analysis has been performed. The new design was implemented on an FPGA to evaluate these hardware costs. The results revealed an 8.8 % increase in logic usage and a 3 % rise in power consumption. Combining both of these analysis shows a decrease in energy consumption. The new module reduces the energy consumption doing the same task by almost 20 %, which is quite impressive. Therefore, the new unit desgin and implementation has been successful.

This project provides a clear example of how small hardware changes can significantly improve execution speed while maintaining low hardware costs and power consumption. The arquitecture of the processor, divided in different modules, allows scalability by easing the addition of new instructions or modules.

Resumen

Este proyecto se ha basado en el diseño, implementación e integración de un acelerador de hardware basado en *Gradient Boosting Decision Trees* (GBDT) en una arquitectura RISC-V. Los GBDT son un modelo utilizado para tareas de clasificación y regresión. Son fácilmente interpretables y fáciles de seguir. A diferencia de otros modelos más complejos como las redes neuronales, los GBDT utilizan operaciones simples, como comparaciones y sumas. Esto hace que sean más eficientes a nivel computacional al tiempo que ofrecen información sobre el proceso de toma de decisiones. Este estudio busca implementar un acelerador en un procesador que sea capaz de gestionar y clasificar píxeles en imágenes hiperespectrales utilizando los GBDT. Estos píxeles pasan por los diferentes nodos dentro de los árboles. Cada árbol tiene diferentes características que deciden qué camino toma el píxel, ya sea el camino derecho o el izquierdo. Cada nodo hoja contiene una predicción, que se suma a las obtenidas en otros árboles. Esta predicción agregada permite al programa clasificar el nodo en una de las diferentes clases existentes.

El estudio se ha dividido en varias fases. Se comenzó identificando los requisitos computacionales para iniciar la nueva unidad funcional. A continuación tuvo lugar un análisis detallado para encontrar un patrón de instrucción común en diferentes partes del código. Al encontrarlo, fue posible realizar un estudio más exhaustivo sobre el coste computacional. Estas instrucciones se ejecutaron en diferentes secciones del código original. La idea principal de este proyecto es fusionar estas instrucciones en una sola. Al hacer esto, se reduce el tiempo computacional ya que esta nueva instrucción solo tiene que pasar por cada fase del procesador una vez en lugar de dos o hasta cuatro veces, dependiendo del patrón. Este enfoque tenía como objetivo reducir el número de ciclos requeridos, mejorando así la velocidad de ejecución.

La nueva instrucción y su correspondiente unidad funcional han sido diseñadas e implementadas en un procesador RISC-V de 32 bits y 5 etapas. Esta nueva unidad de hardware combina lógica de bit a bit, desplazamiento y suma todo en uno. También permite elegir diferentes constantes de un conjunto predefinido para usar. La unidad se ha añadido al procesador en la etapa de ejecución, creando un nuevo módulo y modificando las señales de control para que funcione correctamente. Fue creada usando SystemVerilog. Esta unidad ha pasado por varias fases de validación. La primera consistió en casos de prueba simples que no involucraban todo el procesador, tan solo el nuevo módulo. Se utilizaron árboles sencillos para invertir menos tiempo en el caso de validación inicial. Durante la segunda y última fase, se probaron las imágenes hiperespectrales, que son IP_data, PU_data y KSC_data. Estas imágenes contienen miles de píxeles y nodos. Esta etapa involucró todo el procesador ya que era necesario verificar la correcta integración de la nueva unidad en el procesador.

El análisis de rendimiento ha demostrado una mejora significativa en el tiempo de ciclo de ejecución. El procesador, con la nueva unidad adjunta, ha logrado una aceleración de hasta un 27 % en comparación con el procesador original. El nuevo diseño se implementó en una FPGA para evaluar los costes de hardware. Los resultados revelaron un aumento del 8.8 % en el uso de la memoria lógica y un aumento del 3 % en el consumo de energía. Combinando estos dos datos, se puede obtener la variación de consumo energético. Los resultados muestran que existe un descenso del consumo de casi el 20 % realizando la misma actividad, un dato muy positivo. Por lo tanto, el diseño e implementación de la nueva unidad ha sido un éxito.

Este proyecto muestra cómo pequeños cambios en el hardware pueden mejorar significativamente la velocidad de ejecución mientras se mantienen bajos costes de hardware y consumo de energía. Por otro lado, la arquitectura del procesador, dividida en diferentes módulos, permite la escalabilidad al facilitar la adición de nuevas instrucciones o módulos.

Índice general

Capítulo 1

Introducción

Los árboles de decisión son una herramienta muy útil y eficiente a la hora de abordar problemas predictivos de clasificación de clases y regresión. Sin embargo, un solo árbol no es capaz de realizar y resolver tareas muy complejas. Para resolver problemas de clasificación complejos, existe una variante de los árboles de decisión conocida como *Gradient Boosting Decision Trees* (GBDT) [17]. En esta variante se combinan múltiples árboles, de forma que cada árbol intenta mejorar los errores de árboles anteriores.

Actualmente, los GBDTs son una de las soluciones más populares. Entre sus ventajas cabe destacar que permiten soluciones flexibles capaces de resolver tanto problemas de regresión como de clasificación, y que son capaces de combinar entradas con características muy diversas sin necesidad de aplicar estrategias de normalización [6]. Además, proporcionan métricas que favorecen la interpretabilidad, indicando qué características han sido las más importantes para la toma de decisiones que realiza un modelo de GBDT.

Estos árboles de decisión son árboles binarios en los que el avance desde un nodo padre a uno de sus dos hijos (izquierda o derecha) se decide mediante un valor de comparación. Estos valores se comparan con el píxel de entrada, el cual es el que queremos clasificar en una de las N clases que existen. Este píxel hace un recorrido del árbol en función de sus valores. Finalmente, llegará a un nodo hoja que corresponderá al valor de la predicción de ese árbol en concreto.

La principal característica distintiva de estos árboles de decisión es la manera en la que son entrenados. El entrenamiento se realiza de manera secuencial, haciendo que el error de un árbol se minimice con el valor del siguiente. Cada nodo hoja nos indica el nodo raíz del siguiente árbol (o si es el árbol final que ya ha acabado) y la predicción del mismo. De esta manera, la predicción final del modelo GBDT es la suma de las predicciones de cada uno de los árboles por los que ha ido pasando. En la figura 1.1 se muestra un diagrama de un GBDT.

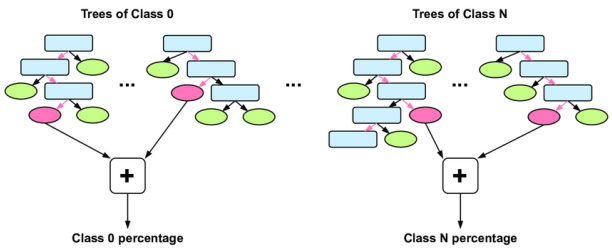

Figura 1.1: Diagrama de GBDT [1]

Existen herramientas como XGBoost [14], LightGBM [7] y CatBoost [3] que optimizan la implementación de los GBDT para que sean rápidos y escalables, ideales para tareas complejas. LightGBM está diseñada para procesadores de alto rendimiento. Una vez entrenados, se usa una librería de software para

ejecutarlos en procesadores empotrados. Esta librería fue diseñada inicialmente para procesar imágenes hiperespectrales, como las usadas en este estudio, pero puede utilizarse en cualquier otro problema cuya entrada sean valores enteros [12]. Existen otros modelos predictivos como por ejemplo las redes neuronales, con una complejidad superior. Frecuentemente, los GBDT alcanzan resultados comparables usando modelos más ligeros en cuanto a los cálculos necesarios. Si la entrada de los árboles son números enteros, la operación básica será una comparación de números enteros, mientras que otros modelos de aprendizaje automático requieren realizar operaciones aritméticas en punto flotante mucho más complejas. También se han realizado investigaciones en los que se estudia y observa en qué contextos se obtiene mejores resultados empleando GBDT y en cuáles usando redes neuronales [8]. La principal ventaja de estos árboles es que se basan en comparaciones y sumas, que son operaciones computacionalmente muy simples. Esto favorece una mejor explicabilidad, es decir, poder observar de manera sencilla qué está ocurriendo a lo largo del todo proceso. Por todo lo anterior, se ha decidido acelerar la ejecución de modelos GBDT. De esta manera, el seguimiento de los distintos pasos a la hora de diseñar el acelerador es más claro y fácil de entender.

Este trabajo de fin de grado propone modificar un procesador de arquitectura RISC-V para aumentar la eficiencia de la inferencia de los GBDT. A lo largo de este estudio se va a trabajar con imágenes hiperespectrales, buscando clasificar los píxeles que la componen. RISC-V es una arquitectura que proporciona un conjunto de instrucciones de libre uso y abierta [11]. Esto permite poder desarrollar nuevos diseños de procesadores sin tener que pagar licencias y añadir instrucciones a medida al conjunto original. El proceso de desarrollo de la extensión RISC-V se divide en cinco grandes bloques.

El primero de ellos consiste en hacer un análisis de la librería de los GBDT. La comprensión de la misma, su funcionamiento e implementación en el programa desarrollado son imprescindibles para entender qué instrucciones o partes del código hay que modificar para desarrollar una nueva unidad funcional de bajo coste y que proporcione una mejora tanto de rendimiento como de consumo de energía. Para ello, hay que comprender el entorno de trabajo, las instrucciones en lenguaje ensamblador de bajo nivel y los ciclos que tarda cada una.

Una vez analizado y comprendido el problema, se debe diseñar e implementar la nueva unidad funcional, de acuerdo a los estudios previos. Esta unidad debe de ser probada y validada tanto con bancos de prueba específicos, como con problemas y datos reales, para demostrar su fiabilidad.

El siguiente paso es integrar la nueva unidad diseñada en el procesador base. Éste es un procesador RISC-V segmentado de cinco etapas desarrollado por el investigador Samuel Pérez [9], dentro de un proyecto de investigación del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza. Al ser un procesador segmentado, la nueva unidad debe de ser integrada de manera que no altere su funcionamiento previo en ninguna de las fases.

El cuarto bloque consiste en la validación de la unidad funcional dentro del procesador base. En esta etapa debe de pasar una nueva serie de pruebas y validaciones de manera que se pueda corroborar el correcto funcionamiento del procesador en su conjunto. Para estas pruebas, se ha empleado el simulador Verilator [10].

Finalmente, y para concluir el trabajo, se hará un análisis del coste hardware y energético utilizando una FPGA y de la aceleración obtenida comparando el algoritmo original con el algoritmo optimizado utilizando la nueva instrucción desarrollada.

Capítulo 2

Metodología

El trabajo ha sido realizado en distinta etapas. Éstas están reflejadas en cada uno de los capítulos del trabajo. El trabajo y esfuerzo han sido constantes a lo largo de los últimos 4 meses, compaginándolo con el horario laboral. Ha existido un seguimiento semanal de los avances para comprobar cómo iba el trabajo y solucionar posibles problemas y dudas. En cada una de estas reuniones se especificaban los objetivos a lograr en la siguiente sesión. El Anexo A refleja las horas invertidas en cada una de las etapas así como en las reuniones semanales.

Las herramientas usadas para el desarrollo del trabajo son muy variadas. El programa base, a partir del cual se ha basado la predicción y clasificación ha sido desarrollado en C y puede verse en el Anexo B. Este lenguaje posee numerosas ventajas en las que destaca el poder trabajar con instrucciones y funciones a bajo nivel. Esto permite un gran control del hardware y poder así compaginar instrucciones a nivel ensamblador con funciones de más alto nivel.

El procesador de partida está implementado en SystemVerilog. Es un lenguaje de descripción hardware que es utilizado para diseñar y verificar sistemas electrónicos. Permite desde el uso de puertas lógicas hasta diseños mucho más complejos que incluyan varios módulos. Para la simulación del procesador, se ha empleado un simulador llamado Verilator. Este simulador realiza cálculos de manera muy rápida y eficiente. Las pruebas usadas por este simulador se llevan a cabo en un banco de pruebas el cual está desarrollado en código C++, que permite una fácil gestión de la memoria así como un control directo sobre el hardware y una fácil depuración. También permite una gran compatibilidad con otras herramientas como las FPGA que se van a mencionar a continuación.

La última de las herramientas a utilizar son los *"Field-Programmable Gate Array* (FPGA) [5]. Son unos circuitos integrados que ofrecen una gran flexibilidad ya que permiten implementar múltiples diseños hardware definidos en *Hardware Description Language* (HDL). Son reprogramables lo que permite que sean configuradas y programadas varias veces. Permite diseñar hardware de forma sencilla, sin necesidad de tener que hacer un circuito integrado a medida. De esta forma, se puede realizar el diseño del circuito con la herramienta VIVADO [15], y posteriormente programar la FPGA para que implemente el circuito diseñado. Todo esto se puede llevar a cabo desde un ordenador convencional. Esta herramienta se va a emplear para el análisis de coste hardware a la hora de implementar el acelerador dentro del procesador base. En la figura 2 se puede observar la estructura interna de una FPGA. Los bloques lógicos (Logic Block en la figura) son la unidad fundamental dentro de la FPGA. Son capaces de implementar una o varias funciones combinacionales o secuenciales. Si se unen varios bloques se pueden realizar diseños tan complejos como se desee. Las interconexiones (líneas que unen los distintos bloques) son las rutas que conectan los distintos bloques. Por último, los bloques de entrada y salida (I/O Block en la figura) comunican la FPGA con el mundo exterior controlando las señales que entran y salen.

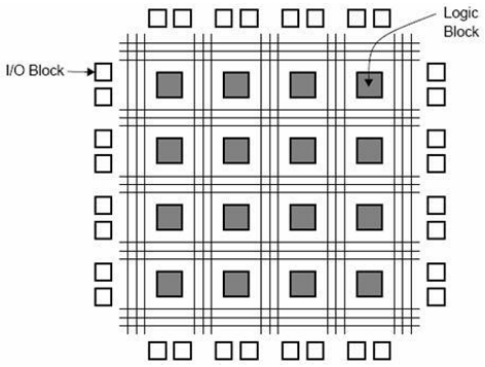

Figura 2.1: Arquitectura de una FPGA [13]

2.1. Descripción del entorno de trabajo

Para llevar a cabo esta tarea de diseñar el acelerador, primero se debe conocer y analizar las herramientas y entornos en los que se trabaja. Principalmente se va a hablar de tres grandes bloques de elementos.

2.1.1. Conjunto de datos de prueba

Para este trabajo, se ha trabajado con varios bancos de pruebas: `IP_data`, `PU_data` y `KSC_data`. Estos archivos contienen píxeles que conforman una imagen hiperespectral. Una imagen hiperespectral como la de la figura 2.2 está compuesta por bandas espectrales. Estas bandas representan diferentes tipos de energía luminosa, cada una con una longitud de onda diferente. Cada imagen hiperespectral está compuesta por muchos píxeles. El problema de clasificación consiste en asociar cada píxel a una clase concreta. Los píxeles son generados por un sensor espectral. Este sensor asocia a cada uno de los píxeles un valor por cada banda espectral. Al conjunto de estos valores se le llama firma espectral. La idea del acelerador es poder clasificar los píxeles en una de las clases a la misma velocidad en la que son generados y medidos por el sensor. Por ello, se busca acelerar el proceso de clasificación lo máximo posible y hacerlo en tiempo de ejecución. En cada nodo, uno de los valores que compone la firma es comparado para decidir la dirección que toma ese píxel dentro del árbol y así determinar su clasificación. En la figura 2.3 se observa lo descrito anteriormente.

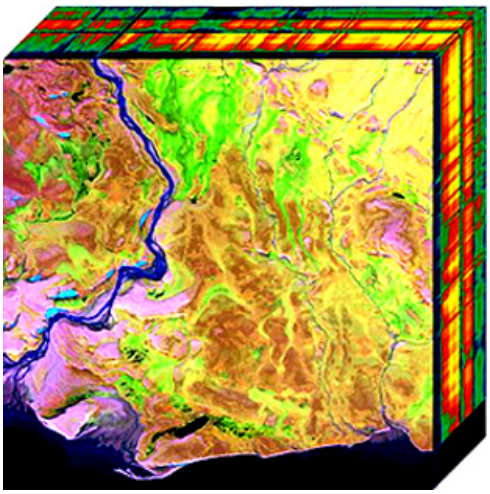

Figura 2.2: Ejemplo de imagen hiperespectral [4]

Usar GBDTs es una opción viable, pero hay otras. Los requisitos computacionales de distintos modelos para el problema de clasificación de estas imágenes pueden consultarse en [2].

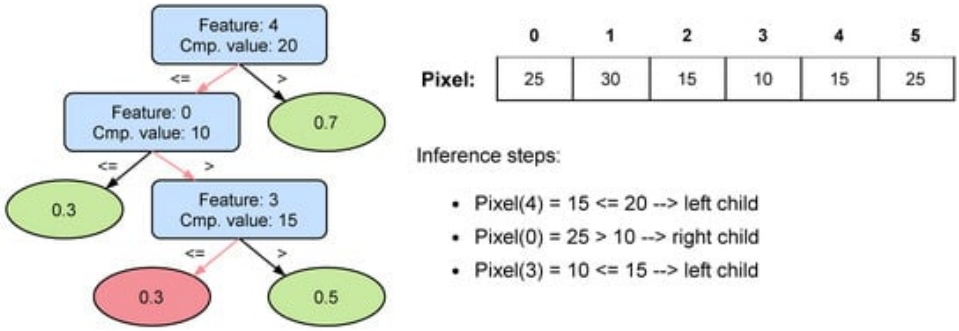

Figura 2.3: Ejemplo de píxel y su camino a través de un árbol [1]

Cada vez que se analiza un píxel en cada una de las clases, se almacena un valor de predicción que se agrega a las predicciones obtenidas en cada uno de los árboles que ha atravesado. La clase predicha será aquella que posea el valor de predicción agregado final más alto. Cada una de las clases a analizar poseerá un máximo número de nodos que puede contener la concatenación de árboles previamente entrenados que conforman la clase. Por último, a la hora de validar, se usarán las tres imágenes mencionadas en esta sección.

2.1.2. Programa de pruebas simplificado

En este trabajo, se van a utilizar los árboles entrenados [16] que pueden tener hasta 8192 nodos. Por ello, en las primeras fases de diseño simular estos sistemas paso a paso requeriría un tiempo excesivo. Su simulación paso a paso es muy lenta, se invertiría excesivo tiempo. Por lo tanto, para las pruebas rápidas iniciales de validación, se han creado tres subcasos para corroborar el correcto funcionamiento:

1. Un árbol compuesto por un solo nodo.
2. Un árbol compuesto por el nodo raíz, hijo izquierdo y derecho.
3. Tres árboles concatenados pero muy sencillos para comprobar el correcto pase entre árboles.

Cada uno de estos casos está compuesto por tan solo un píxel y dos valores de comparación. Son árboles simplificados para comprobar el correcto avance a través de los árboles.

2.1.3. Programa para llevar a cabo la predicción

Una vez obtenidos los árboles GBDT entrenados, es necesario usarlos para realizar la predicción. Para ello, se ha obtenido del trabajo desarrollado por Adrián Alcolea [1] un programa desarrollado en C que permite, a partir de los árboles previamente entrenados, clasificar cada uno de los píxeles de una imagen hiperespectral en distintas clases. Este programa posee una función llamada `predict()` la cual se encarga de obtener la predicción de cada uno de los píxeles en cada una de las clases. Se puede observar la función original en el Anexo B. Dentro de esta función, el píxel comienza en el primero de los árboles. Desde aquí, dependiendo del valor del píxel a tratar y los valores de comparación de los árboles de cada clase, irá avanzando al nodo izquierdo o derecho de cada uno de los árboles. Al final de cada árbol, en uno de sus nodos hoja, añadirá la predicción de ese mismo nodo a la ya agregada hasta llegar al nodo final. La función `main()` compara cada una de las predicciones finales obtenidas en cada una de las clases y almacena tan solo la de mayor valor. Cuando ya se ha tratado ese píxel por cada una de las clases, se decidirá que la predicción final es aquella clase que haya obtenido el valor de predicción máximo. En la figura 2.4, se puede observar en qué campos se dividen los 32 bits que conforman cada uno de los nodos, dependiendo si es nodo hoja o no. Se puede observar que los nodos que no son hoja

poseen la dirección de la constante con el que se van a comparar, su valor de comparación y la dirección relativa de su hijo derecho. Por otro lado, los nodos hoja poseen un valor de predicción y la dirección del próximo árbol, en el caso de que no sea el árbol final.

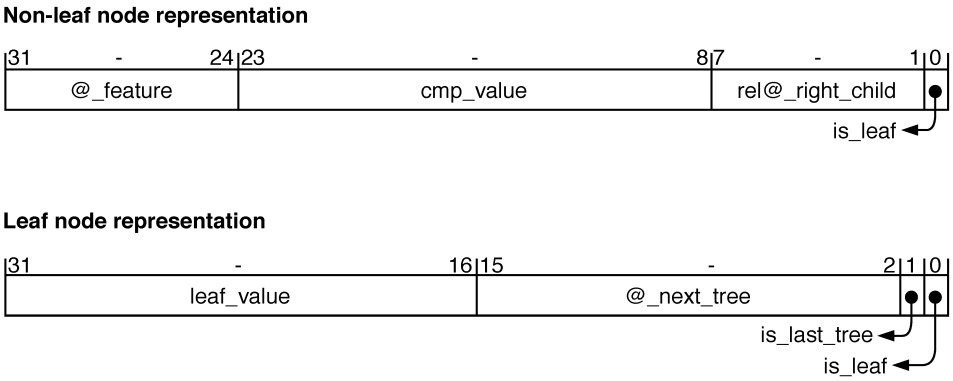

Figura 2.4: Campos dentro de cada tipo de nodo [1]

Los árboles utilizados, poseen la particularidad de que los nodos hijo izquierdo y derecho de cada nodo padre no están colocados de manera arbitraria. Para cada uno de los nodos padres, el nodo de a continuación en el vector de nodos será su hijo izquierdo, por lo tanto, el cálculo de la posición de éste es inmediata. Sin embargo, el hijo derecho se encuentra situado a una distancia x que es indicada por el padre en caso de que tenga que avanzar por el nodo hijo derecho. Esta operación ya requiere de más instrucciones y por lo tanto de más ciclos. Estos detalles que se comentan pueden observarse claramente en la figura 2.5.

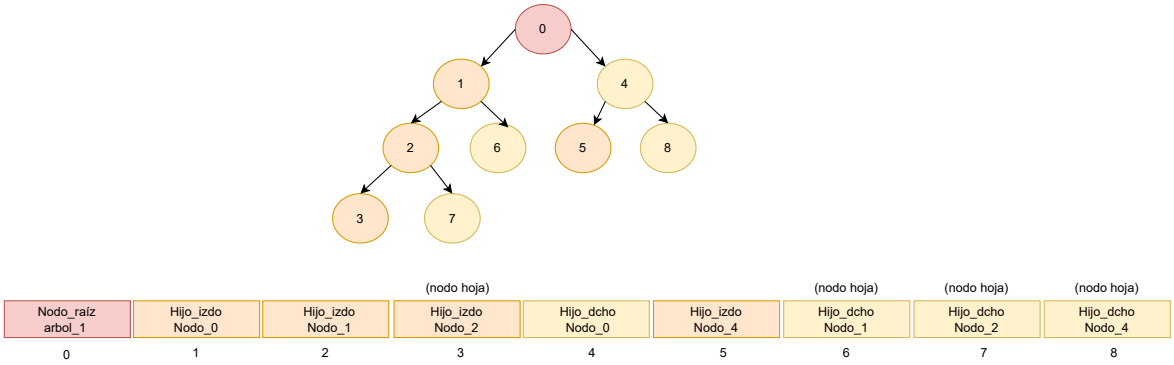

Figura 2.5: Representación de un árbol en memoria

Análisis del programa en C

En esta sección se van a tratar las necesidades observadas dentro del programa y cómo pueden ser optimizadas gracias a la inclusión de una nueva unidad funcional.

En primer lugar, se ha realizado una medición de ciclos dentro del programa C. Para ello, se ha hecho uso de unos contadores externos ya incluidos en el banco de pruebas de C++. El banco incluye 264 contadores que se pueden utilizar, modificando el código del simulador, para que monitoricen distintos eventos, y proporcionen métricas precisas a nivel de ciclo de rendimiento. Han servido para medir dentro de la propia función los costes en cuanto a ciclos de cada una de las fases o etapas. Se han colocado distintos contadores a lo largo de la función y se ha observado que la fase de la función que más impacto tiene en el rendimiento es la comparación para decidir el camino dentro del árbol. Se han realizado diversas

mediciones, como por ejemplo, los ciclos que tarda la función desde que se ha elegido el camino derecho hasta que termina la iteración del bucle principal. Todas estas mediciones han servido para hacer patente los gastos en ciclos de cada una de las partes de la función. De esta manera, se puede abordar el problema sabiendo qué secciones consumen más recursos. En el Anexo C, se observan las mediciones realizadas. Todas estas mediciones se han realizado sobre el programa en C del Anexo B.

2.1.4. Procesador base

El procesador base sobre el que se sustenta el trabajo ha sido desarrollado SystemVerilog utilizando el simulador Verilator. Es un procesador RISC-V con tamaño de palabra de 32 bits. La ejecución de instrucciones está segmentada en cinco etapas: *fetch, decode, execution, memory* y *wirteback*. Cada una de estas etapas ejecuta las siguientes tareas:

1. ***Fetch***: Se encarga de coger la nueva instrucción de memoria en función del PC así como de actualizar este valor.
2. ***Decode***: En esta etapa se recoge la instrucción y se dividen los campos de la misma en función del tipo de instrucción. También se leen los registros del banco de registros (en este procesador hasta 3) y se generan las señales de control que serán cruciales para las etapas posteriores.
3. ***Execution***: Esta tercera etapa es la de ejecución de la instrucción. El procesador ya posee todos los datos necesarios así como de las señales de control. Por lo tanto, es capaz de realizar la tarea. La nueva unidad a implementar sobre todo va a afectar a esta etapa del procesador.
4. ***Memory***: En la etapa de memoria, se accede a memoria si es necesario. En caso contrario, no se hace nada y tan solo se transmiten los datos y señales de control.
5. ***Writeback***: Está última etapa es la de almacenamiento de datos de vuelta al banco de registros. De esta manera, los resultados obtenidos de la instrucción ejecutada están disponibles para futuras instrucciones. La unidad a implementar realizará esta etapa.

También es capaz de gestionar paradas provocadas por dependencias de datos para así funcionar de manera correcta. Es un procesador de bajo coste. La figura 2.6 muestra un diagrama de su ruta de datos.

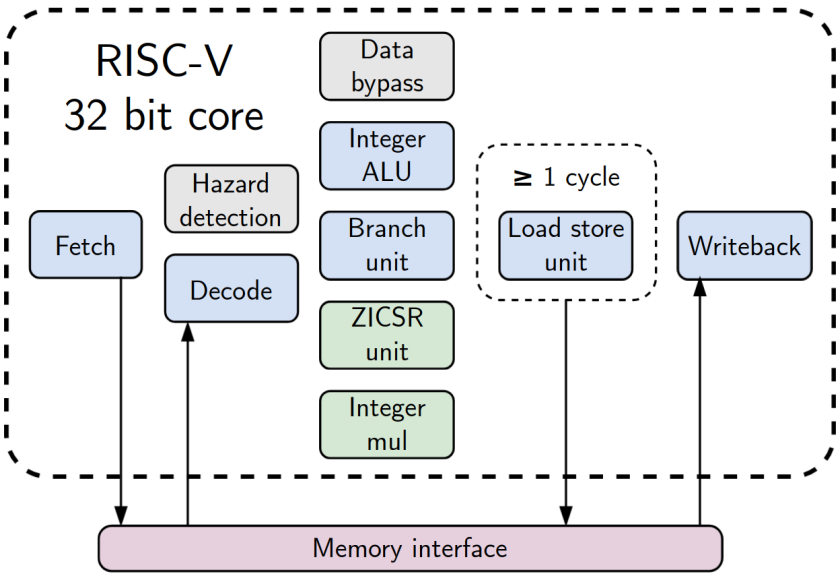

Figura 2.6: Ruta de datos del procesador base [9].

Capítulo 3

Análisis de los requisitos funcionales de los GBDT

En este capítulo se abordan los primeros pasos a la hora de diseñar, el acelerador. De esta manera, y explicado paso a paso, se podrá comprender el contexto, las ideas y análisis de la nueva unidad funcional, así como la conclusión para determinar el diseño final de la misma.

3.1. Análisis funcional

3.1.1. Desarrollo en pseudocódigo del ensamblador

Para llevar a cabo el estudio de requisitos, es necesario profundizar en la función `predict()` mencionada. Para ello, se ha desarrollado en pseudocódigo de ensamblador cada uno de los pasos e instrucciones que se llevan a cabo dentro de la función `predict()`. Una imagen del programa a nivel ensamblador ha servido como base para poder desarrollar el pseudocódigo. Los procesos llevados a cabo durante el capítulo de metodología 2 han permitido observar que hay dos patrones bastante repetitivos en la función `predict()` bajo estudio. En concreto, los siguientes:

AND	r2	r1	CONST
SRL	r3	r2	DESP

Cuadro 3.1: Primer patrón

AND	r2	r1	CONST
SRL	r3	r2	DESP
ADD	r5	r4	r3

Cuadro 3.2: Segundo patrón

Cada uno de los `ri` es un registro del banco de registros y tanto `CONST` como `DESP` son valores constantes. Esta secuencia de instrucciones se repiten a lo largo del bucle principal de la función 3 y 2 veces respectivamente. También se puede observar que las dos primeras instrucciones se repiten en los dos patrones. En el fragmento 3.1 se observa la aparición de los dos patrones mencionados en el pseudocódigo en ensamblador.

```
1  NON_LEAF:
2      and $t4, $t3, 0xff000000    # r4 = r3 & 0xff000000 || Patron 2
3      srl $t5, $t4, 0x18          # r5 = r4 >> 24        ||
4      add $t6, $t11, $t5          # r6 = r11 + r5        ||
5      lw $t13, 0($t6)             # r13 = pixel[r6]
6      and $t7, $t3, 0x00ffff00    # r7 = r3 & 0x00ffff00 || Patron 1
7      srl $t8, $t7, 8             # r8 = r7 >> 8         ||
```

Figura 3.1: Fragmento de pseudocódigo en ensamblador

Dentro del programa de C, estos patrones se reflejan en varios fragmentos del código 3.2 3.3. En el segundo fragmento es donde se decide si el próximo nodo es el hijo izquierdo o el hijo derecho, en el caso de que el nodo actual no sea hoja.

```
#define IS_LEAF(x)      ((x & 0x00000001) == 0x00000001)
#define PRED_VALUE(x)   ((x & 0xffff0000) >> 16)
#define NEXT_TREE(x)    ((x & 0x0000fffC) >> 2)
#define NUM_FEATURE(x)  ((x & 0xff000000) >> 24)
#define CMP_VALUE(x)    ((x & 0x00ffff00) >> 8)
#define RIGHT_CHILD(x)  ((x & 0x000000fe) >> 1)
#define CLASS_END(x)    ((x & 0x00000001) == 0x00000001)
```

Figura 3.2: Fragmento de código que incluye los patrones

```
feature = pixel[NUM_FEATURE(class[curr_addr])];
cmp_value = CMP_VALUE(class[curr_addr]);
bool bigger = feature <= cmp_value
```

Figura 3.3: Fragmento de código en el que se emplean los define

En una misma iteración del bucle, puede repetirse alguno de estos dos patrones hasta 4 veces y como mínimo 2. Teniendo en cuenta que cada uno de los fragmentos mostrados se ejecutan en cada una de las iteraciones del bucle dentro de la función `predict()`, si se lograse reducir el consumo en ciclos de esta sección del código, se obtendría una mejora de rendimiento.

3.2. Conclusión del análisis

Con todos los datos de las etapas anteriores ya se puede realizar un análisis sobre un posible diseño de la nueva unidad funcional. Lo primero reseñable es que los dos patrones mencionados en la sección anterior se encuentran en las etapas de la función que más impacto en el rendimiento tienen, la fase de comparación. Los patrones no solo aparecen en este fragmento si no en varios más. Por lo tanto, el objetivo es reducir el número de ciclos de estos patrones de instrucciones.

Como ya se ha mencionado, estamos ante un procesador que ejecuta las instrucciones de manera segmentada y por lo tanto cada una de ellas debe de pasar por cada una de las cinco fases. Si por ejemplo hubiera una instrucción que englobase las funciones de las instrucciones anteriores, tan solo tendría que pasar un sola vez por cada una de las cinco fases y al pasar de dos instrucciones en dos, se produciría un ahorro de como mínimo un ciclo. Esto sería el caso del primero de los patrones. Si englobásemos las tres instrucciones del segundo patrón, se lograría una mejora de al menos dos ciclos por cada ve que se ejecutase ese patrón. Por otro lado, como ya se mencionó antes, las dos primeras instrucciones son idénticas (con diferentes constantes) en los dos patrones. Por lo tanto, es lógico intentar agrupar los dos patrones en una misma instrucción. De esta manera se puede ejecutar esta nueva instrucción indiferentemente al patrón. Ese es el objetivo e idea base del acelerador: crear esta nueva instrucción multifuncional.

Capítulo 4

Diseño e implementación del acelerador

En el capítulo anterior, se ha dejado clara la idea principal: crear una nueva instrucción capaz de ejecutar los dos patrones encontrados con un mayor impacto potencial en el rendimiento. Por lo tanto, ahora toca realizar el diseño de esta nueva instrucción con sus correspondientes componentes hardware que permitirán su ejecución.

4.1. Ideas básicas de diseño

En este capítulo, se busca realizar un primer diseño de la nueva unidad funcional. En la figura 4.1 se puede observar una versión simplificada del nuevo componente. De esta manera, se podrá seguir con mayor facilidad las ideas de a continuación.

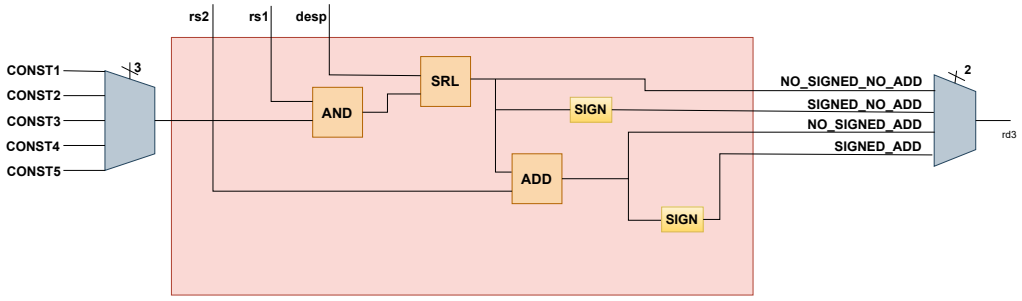

Figura 4.1: Diseño simplificado de la nueva unidad funcional

La primera idea a tener en cuenta, es en qué tipo de código se va a programar. Como el procesador base ha sido programado en SystemVerilog, la nueva unidad debe de ser implementada con el mismo lenguaje ya que luego ha de añadirse al mismo. A continuación, se debe pensar qué tipo de componentes hardware son necesarios para poder cumplir los patrones. Al analizar el programa en C en el Anexo B, se puede observar que existen cinco constantes distintas que son usadas en la operación `and ri, rj, CONST`. La nueva instrucción debe de ser capaz de elegir una de entre las cinco constantes por lo que es necesario un `MUX` con 3 bits de control. A continuación se puede observar una tabla con cada una de las entradas `MUX`:

Nombre constante	Valor	Bits de control
CONST1	0xFFFF0000	000
CONST2	0x0000FFFC	001
CONST3	0XFF000000	010
CONST4	0X00FFFF00	011
CONST5	0X000000FE	100

Cuadro 4.1: Datos de las CONST para la operación AND

Una vez elegida la constante, se debe poder elegir el desplazamiento de la instrucción `srl ri, rj, desp`. El desplazamiento máximo posible en un dato de 32 bits es obviamente 32. Por lo tanto, son necesarios cinco bits para poder indicar el desplazamiento. Analizando la función `predict()`, ver Anexo B, se puede ver que alguna de las salidas requiere una salida con signo. Los valores están codificados en complemento a 2, esto significa que el bit de más peso indica el signo del número en binario. Si es 1, el número es negativo, positivo en caso contrario. Por lo tanto, es necesario distinguir entre las operaciones con signo y sin signo. En la línea 7 del programa en C del Anexo B, se puede observar que el valor que se utiliza es de 16 bits. Sin embargo, el resto de vectores con los que se trabaja es de 32. Por lo tanto, hay que realizar una conversión a un vector de 16 bits, pasarlo de un valor sin signo a un valor con signo y a continuación trasladar ese signo de nuevo a un vector de 32 bits. Con toda esta información, se puede avanzar al siguiente paso que es la ejecución de las dos instrucciones. Esto se logra gracias a los operadores `&` para la operación `AND` y `<<<` para el desplazamiento. De esta manera, las dos primeras instrucciones de los dos patrones ya han sido realizadas. Hay que tener en cuenta que el segundo de los patrones señalados en el capítulo anterior, posee una instrucción final de suma. Por lo tanto, es necesario un segundo `MUX` de 2 bits de control para poder elegir entre el resultado con suma, con signo y sin signo y el resultado sin suma con signo y sin signo. En la siguiente tabla se puede observar las posibles entradas del `MUX`:

Nombre constante	Descripción funcionamiento	Bits de control
NEW_M_NO_ADD	NO suma sin signo	00
NEW_M_SIGNED_NO_ADD	NO suma con signo	01
NEW_M_ADD	Suma sin signo	10
NEW_M_SIGNED_AD	Suma con signo	11

Cuadro 4.2: Salida final de la instrucción

4.2. Banco de pruebas para validación

Para finalizar esta fase de diseño del acelerados se realizaron pruebas sencillas de validación. De esta manera, se comprueba que la nueva instrucción funciona en una serie de casos específicos. La tabla 7.1 muestra las primeras pruebas de validación. Estas pruebas representan casos reales y válidos en la función `predict()`. Estos serán los valores que se le darán a la función en cada una de las llamadas dentro del programa. En estas pruebas se toman como valores de entrada en rs_1 y rs_2 valores sencillos que no cubren todos los casos.

CONST = 1	desp = 16	MUX1 = 0	MUX2 = 3
CONST = 2	desp = 2	MUX1 = 1	MUX2 = 0
CONST = 3	desp = 24	MUX1 = 2	MUX2 = 0
CONST = 4	desp = 8	MUX1 = 3	MUX2 = 0
CONST = 5	desp = 1	MUX1 = 4	MUX2 = 3

Cuadro 4.3: Pruebas iniciales en la nueva unidad

Capítulo 5

Integración del acelerador en el procesador base

En este capítulo se va a integrar la nueva unidad en el procesador ya existente de tal manera que el funcionamiento y rendimiento del mismo sea válido. Para ello, no vale solo con añadir el hardware recién creado al procesador sino que hay que tener en cuenta las distintas fases del procesador así como sus señales de control.

Cómo ya se ha mencionado anteriormente, el procesador está segmentado en 5 etapas. Esta nueva instrucción, realiza la primera fase de manera normal. En la fase de decodificación, se vieron modificadas las señales de control. En la etapa de ejecución se va a añadir un nuevo módulo y otras componentes. Se explican todos estos cambios a continuación.

5.1. Creación del nuevo módulo

Lo primero de todo, es asignar a la nueva instrucción un código de operación. De esta manera, es reconocido por el procesador y es decodificada correctamente. Es necesario elegir uno de los códigos de operación que están libres dentro de RISC-V para instrucciones. En la figura 5.1, se pueden observar los `OPCODE` ocupados y libres.

Inst (4:2) / Inst (6:5)	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NAMDD	OP-FP	reserved	custom2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom3/rv128	≥80b

Figura 5.1: tabla de `OPCODE` en RISC-V

En este caso, el nuevo `OPCODE` será `1011011`. El `10` corresponde a la tercera fila, el `110` a la séptima columna y `11` a una de las posibles combinaciones de bits libres. Todo esto corresponde al valor de *custom2*.

El código donde está definido el comportamiento del procesador, concretamente en la etapa de *execution* está dividido en módulos. En esta etapa, cada uno de los grandes bloques de componentes corresponde a un módulo. Por ejemplo la ALU posee un módulo concreto. Por lo tanto, en este caso también es necesario crear un nuevo módulo o unidad para esta instrucción ya que posee unas características que ningún otro módulo puede cumplir.

Cada uno de estos módulos está conectado al módulo principal de la fase de *execution*. Cada módulo posee unos valores de entrada y otros de salida. En este caso, los valores de entrada necesarios son:

1. Un dato de 32 bits que será llamado `rs1`. Esto se debe a que en codificación RISC-V, al primer registro fuente se le llama así. Es usado para la operación de `AND` con una de las constantes.

2. Otro vector de 32 bits, `rs2`, que es el segundo registro fuente en codificación RISC-V. Este segundo vector es el que se suma al resultado de los pasos anteriores en caso de requerirse por la operación.

3. 5 bits que serán usados para indicar el desplazamiento requerido en la parte del <<<.

4. 3 bits que servirán como bits de control para el primero de los multiplexores indicado en el capítulo anterior. Ésto indica que constante se elige para aplicarla en el `AND`.

5. 2 bits para el control del segundo multiplexor. Éste indica si la salida es con suma o sin suma además si es con signo o sin signo.

Como valor de salida se empleará otro dato de 32 bits llamado `rd` que es el nombre que recibe el registro destino en codificación RISC-V.

Las instrucciones que son leídas de memoria para ser decodificadas poseen 32 bits. Cada uno de los valores de entrada anteriores es codificado en el formato de una instrucción de RISC-V y en esta instrucción concreta, están distribuidos como se observa en la figura 5.2.

Los cinco bits de `rs1` y `rs2` van a pasar durante la fase de decodificación por el banco de registros el cual posee 31 registros de 32 bits cada uno. Existe un registro extra que siempre tiene valor cero. Estos cinco bits servirán para indicar en qué posición del banco de registros se encuentra el deseado y de esta manera, tener los dos vectores de 32 bits disponibles para la siguiente etapa. Los nuevos campos `desp`, `mux1` y `mux2`; se van a definir utilizando los campos `funct3` y `funct7` definidos en el formatos de instrucciones de RISC-V para incluir información adicional.

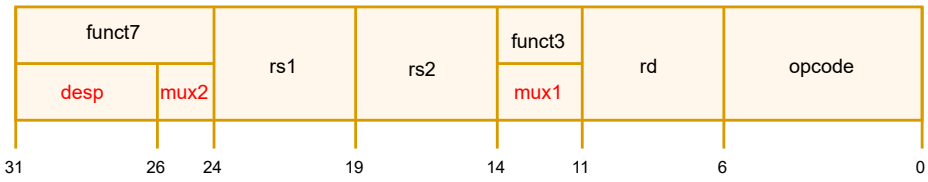

Figura 5.2: Codificación de la instrucción implementada de tipo R

5.2. Modificación del procesador base

Una vez creado el nuevo módulo, es necesario añadir los cambios necesarios dentro del procesador. De esta manera, comprenderá qué necesita esta nueva instrucción. Los principales cambios son las señales de control. Existen diversas señales dentro del procesador pero solo se mencionan las relevantes para esta nueva instrucción.

1. `t = INSTR_R_TYPE`. De esta manera se indica al procesador que está trabajando con una instrucción de tipo R. Existen multitud de tipos de instrucciones y cada una posee una distribución de los 32 bits distintas. De esta manera, se le indica que la distribución de tipo R es la correcta y que debe de tratar la nueva instrucción como tal.

2. La señal de `register_wb = 1`. Indica que es necesario guardar de vuelta en el banco de registros el resultado de la instrucción. De esta manera estará disponible para instrucciones posteriores.

3. `WB_SOURCE = WB_NEW_MODULE`. Esta segunda señal indica la procedencia de aquello que se va a escribir de vuelta en el banco de registros. Como se ha comentado antes, existen instrucciones extras y ALUs por lo que es necesario indicar que lo que se quiere guardar en el banco es la salida de nuestro módulo, el cual estará guardado en el vector `data_result[0]`.

4. `use_rs[0] = 1` y `use_rs[1] = 1`. El banco de registros de este procesador posee tres salidas. Sin embargo, para la nueva instrucción implementada solo son necesarias dos de la salidas. De esta manera se le indica que solo se usarán las dos primeras. Estas dos señales también son útiles para la detección de riesgo de datos y adelantamiento de operandos. La detección de riesgos consiste en evitar que el procesador funcione incorrectamente en determinadas situaciones. Se está trabajando en un procesador segmentado por lo que se están ejecutando hasta cinco instrucciones de manera paralela. Hay ocasiones en los que una instrucción necesita datos de una instrucción previa que no han sido procesados aún. Ejecutar esta instrucción de manera normal provocaría errores dentro del procesador. Por esto, el sistema de detección es vital.

Además de las señales, se ha añadido el nuevo módulo a la etapa de ejecución del procesador. Para ello, se han añadido nuevas estructuras de datos que permiten poder integrarlo correctamente. El archivo `rv32_types.sv` contiene todos los tipos de estructuras de datos que existen dentro del procesador y sus posibles valores. Dentro de este archivo se han incluido las nuevas estructuras para los dos multiplexores de la nueva unidad funcional y sus valores. También se ha añadido el nuevo `OPCODE` y un nuevo bus de salida correspondientes a la nueva instrucción.

5.3. Integración con el compilador gcc

Una vez integrada la nueva unidad al procesador base, es momento de ponerlo a prueba con el programa en C del Anexo B encargado de la clasificación de píxeles. Sin embargo, este programa no es capaz de reconocer el nuevo módulo e instrucción por sí solo. Es necesario indicarle qué es lo que tiene que hacer. Por ello, se ha creado una nueva función llamada `extract_field()`. Esta función indica al compilador de C qué es lo que debe de hacer exactamente. Para ello, se debe declarar esta función como *inline*. Ésto indica al compilador que debe de insertar directamente el código de la función en vez de llamarla. De esta manera, se reduce el *overhead* de la ida y vuelta de la llamada. Como la función creada es pequeña y es usada frecuentemente, es un método muy recomendable a utilizar. Dentro de esta misma función, se realiza la llamada a la instrucción de bajo nivel. Para ello, se emplea un tipo especial de instrucción llamada `.insn`. Este tipo de instrucciones forman parte del grupo *Register Transfer Language* (RTL) que permiten al compilador GCC generar el código máquina necesario para llevar a cabo la instrucción requerida. Esta instrucción `.insn` posee varios campos los cuales son rellenados con los parámetros que se pasan a la función de más alto nivel creada `extract_field()`. En el primer campo de la instrucción se indica el tipo de instrucción desarrollada (en este caso tipo R). Los siguientes campos son rellenados con el número de bits necesarios de acuerdo al tipo de instrucción indicado en el primero de los campos. Pueden existir un número de parámetros distinto al de campos. En esta nueva instrucción, alguno de los campos, como por ejemplo el de `funct7` está compuesto por la concatenación de dos parámetros el `desp` y la señal de control del segundo multiplexor. La función en C desarrollada se puede observar en la figura 5.3.

```
inline int extract_field(int rs1, int rs2, const int desp, const int
    ctrl_mux1, const int ctrl_mux2){
    int rd3;
    const char funct3 = ctrl_mux:1 & 0b111;
    const char funct7 = (desp | (ctrl_mux_2 << 5)) & 0b1111111;
    __asm__ volatile (
        ".insn r 0x5B, %[func3], %[func7], %[rd3], %[rs1], %[rs2]"
        : [rd3] "=r" (rd3)
        : [rs1] "r" (rs1), [rs2] "r" (rs2), [func3] "i" (func3), [func7]
          "i" (func7)
    );
    return rd3;
}
```

Figura 5.3: Instrucción `extract_field()` implementada

Capítulo 6

Diseño de pruebas y validación del acelerador

La realización de pruebas se ha dividido en dos grandes bloques. El primero de ellos ha consistido en ejecutar los árboles de prueba. Como se ha mencionado en apartados anteriores, son árboles muy básicos que obviamente no cubren todo el espectro de casos. Sin embargo, ha servido para comprobar que el acelerador iba por buen camino. En el segundo bloque, una vez superado el primero, se ha probado con los árboles originales. Estos árboles contienen una gran cantidad de nodos y casuísticas por lo que son muy útiles para corroborar el correcto funcionamiento.

6.1. Comparación de resultados con programa original

Para la validación de las pruebas, todos los resultados obtenidos con la nueva función han ido siendo comparados con el programa original. De esta manera, si los resultados diferían, el programa paraba y se indicaba el nodo donde había ocurrido el fallo. Esto ha sido muy conveniente ya que ha permitido profundizar en el error y ver las diferencias paso a paso entre nuevo y original. De nuevo, en esta segunda fase de pruebas, se han repetido las pruebas de la sección 4.2. Así se comprueba que el programa sigue siendo correcto y válido.

6.2. Validación de pruebas aisladas

El primer bloque de pruebas no tuvo excesiva complicación. Al ser pocos casos (aproximadamente 10 nodos) y muy básicos, todo fue bien. Sin embargo, al pasar al segundo bloque, varios casos fueron erróneos. Al calcular las nuevas predicciones, el programa original realiza una conversión a un entero de 16 bits con signo. Sin embargo, en el procesador base se trabaja con enteros de 32 bits por lo que el signo no se trasladaba correctamente. Haciendo una conversión de datos de 32 a 16 bits, extendiendo el signo y volviendo a 32 bits, se logró solucionar el problema.

Otro problema que surgió durante las pruebas fue el manejo de los signos negativos. Por defecto, Verilator trabaja con vectores sin signo. Sin embargo, el programa en C requiere que el desplazamiento sea con signo pero el resultado sin signo. Para poder solucionar este apartado se realizo una simulación de las funciones. Se daba como valor predeterminado a las llamadas de la función *curr_addr = -1* en la función `predict()` del programa en C del Anexo B. De esta manera, se comprobaba el manejo de números negativos en cada una de la llamadas. Una vez solucionado este caso, el resto de números negativos eran iguales por lo que el problema se solucionó.

6.2.1. Trazas de logs

Para solucionar los casos anteriores, tuvo lugar un proceso de depuración. Para llevarlo a cabo, se requirió del uso de los bancos de pruebas en C++ sobre los que se apoya verilator así como de impresiones por pantalla de los distintos resultados. El banco de pruebas permite sacar por pantalla las distintas señales y valores que conforman el procesador. Para ello, previamente había que crear una serie de funciones que permitiesen obtener las señales del procesador. Se puede observar un ejemplo en la figura 6.1. Esta función en concreto, permite obtener las señales internas correspondientes a la etapa de ejecución. Esta misma función existe para cada una de las etapas. También hay otras que permiten extraer señales de unidades concretas como la que se está diseñando.

```
1 inline ExecutionStageData get_exec_stage_data(const Vrv32_top* rvtop) {
2     ExecutionStageData d;
3     d.set(rvtop->rv32_top->core->exec_stage->internal_data);
4     return d;
5 }
```

Figura 6.1: Ejemplo función para el banco de pruebas en C++

Gracias a estas funciones, se pudo observar los *inputs* y *outputs* en cada una de las llamadas a la instrucción además de los valores de los buses y señales dentro de la misma en cada una de las etapas.

Por otro lado, también se han usado las impresiones por pantalla. A la hora de comparar el procesador nuevo y original, cada vez que existía un fallo, se han introducido `printf` para mostrar por pantalla las diferencias entre los dos casos. De esta manera, se ha podido comprobar cuál de los parámetros usados era el que difería. Así se ha podido poner el foco en aquello que fallaba.

Capítulo 7

Análisis de rendimiento y coste

La nueva unidad funcional ya ha sido implementada de manera exitosa. No hay problemas con las señales y ha superado las pruebas de validación. Sin embargo, resta el paso más importante. Comprobar que esta nueva unidad realmente acelera el tiempo de ejecución y merece la pena. No sería útil implementar algo que incremente el número de ciclos necesarios para ejecutar el programa de clasificación. Para medir si ha sido un éxito o no, hay que tener en cuenta varios factores y pasos que se explican a continuación.

7.1. Análisis del *speed up*

El primero de los pasos es calcular el *speed up*. Esta médida, muy utilizada en arquitectura de computadores mide la mejora de velocidad. Se compara la ejecución de una tarea en dos arquitecturas similares pero que emplean recursos distintos. Para que el acelerador sea efectivo, el valor del *speed up* debe de ser superior a 1, y cuanto más alto mejor. Este valor se calcula mediante la Ecuación 7.1.

$$Speed\ Up = \frac{T_{\text{old}}}{T_{\text{new}}} \tag{7.1}$$

En esta ecuación, T_{old} es el tiempo en número de ciclos original y T_{new} es el tiempo en número de ciclos de la nueva versión.

Para calcular esta factor de rendimiento, es necesario poder conocer el número de ciclos que ha tardado cada ejecución. Para ello, se van a utilizar los contadores que se explicaron anteriormente. En la fase de exploración de requisitos, fueron utilizados para poder ver qué etapas en el programa requerían mayor cantidad de ciclos. En este caso, se van a utilizar para calcular el coste en ciclos total de la ejecución y predicción de todos y cada uno de los hasta 8721 píxeles en las 16 clases posibles para cada una de las imágenes.

7.1.1. Árboles de prueba

Primeramente, esta métrica se va a probar en los árboles de prueba. Es un entorno de "juguete"por lo que los valores son útiles para ver si la nueva unidad funciona. No están todas las clases y tan solo hay un píxel. En este caso, se obtiene un *speed up* de aproximadamente 1,4. Es un resultado bastante bueno teniendo en cuenta que sólo se ha modificado una pequeña parte del código para que utilice la nueva instrucción.

7.1.2. Árboles de decisión originales

En este caso, se va a probar con las imágenes de prueba al completo. Para el cálculo, se han creado dos funciones `main()` diferentes. En la primero de ellas, se emplea la función `predict()` original, sin modificaciones. En la segunda, la función `predict()` se ha visto modificada, añadiendo la nueva función personalizada. Creando dos programas independientes, se evita que, a la hora de compilar, el propio

compilador entremezcle los resultados de las dos funciones para ahorrar registros y variables. De esta manera los cálculos obtenidos no se ven alterados.

Con los datos de *IP_data*, se obtiene en la ejecución del programa con la arquitectura original 3.915.036.124 ciclos. Por otro lado, con la nueva arquitectura se obtienen 3.092.770.718 ciclos. Se puede observar de manera gráfica en la figura 7.1.

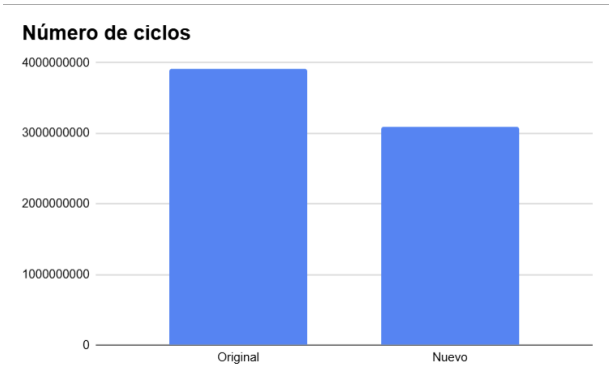

Figura 7.1: Gráfica del *speed up obtenido en la imagen IP_data*

A continuación, se calcula con los datos anteriores el valor del *speed up*:

$$Speed\ Up\ IP\_data = \frac{T_{\text{og}}}{T_{\text{new}}} = \frac{3{,}915{,}036{,}124}{3{,}092{,}770{,}718} = 1{,}26586691384 \tag{7.2}$$

Se observa que el valor del *speed up* no es 1.4 pero aún así es un valor superior a 1 y con una mejora cercana al 27 %. Por lo tanto, ya se ha comprobado que en cuanto a ciclos, existe una mejora sustancial. También se ha llevado a cabo el cálculo del *speedup* con las otras dos imágenes: `PU_data` y `KSC_data`. Los resultados han sido los siguientes:

$$Speed\ Up\ PU\_data = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{775{,}733{,}420}{639{,}315{,}250} = 1{,}21313130259 \tag{7.3}$$

$$Speed\ Up\ KSC\_data = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1{,}242{,}505{,}172}{992{,}556{,}540} = 1{,}25182305431 \tag{7.4}$$

En estas dos nuevas imágenes se observa un *speed up* en torno al 20-25 %. Éstos siguen siendo valores razonables y para los que merece la pena implantar la nueva unidad funcional en el caso que los costes de Hardware sean bajos.

7.2. Análisis de coste Hardware

La implementación de la nueva unidad requiere añadir nuevas unidades hardware al procesador base. Estas nuevas unidades tienen un coste asociado. Por lo tanto, es necesario comprobar que estos nuevos costes no penalizan en exceso y por lo tanto es conveniente implantar la unidad.

Para analizar este coste, se ha empleado la herramienta VIVADO para implementar el diseño en una FPGA que está explicado en el capítulo de metodología 2 de manera más extendida. Esta herramienta es la que proporciona el fabricante de la placa usada (Xilinx). En este análisis se tienen en cuenta diversas métricas:

1. **LUTS:** Son uno de los bloques principales de las FPGA. Son las tablas usadas para implementar las funciones lógicas. Un símil de las mismas serían las tablas de verdad. Permiten el diseño de los circuitos lógicos.

2. **FFS:** Son los *Flip-Flops*. Son circuitos de un bit de datos que permiten la creación de registros y otros elementos secuenciales.

3. **DSP:** Son bloques de mucho más peso que los LUTS o FFS previos. Se utilizan para realizar operaciones más complicadas como la multiplicación de manera más eficiente.

4. **BUFG:** Son los distribuidores de las señales del reloj. Permiten que ésta señal llegue a cada parte del circuito que se ha diseñado.

5. **Estimated power:** Es una estimación del consumo de potencia. Te permite aproximar el consumo del circuito si se pusiera en marcha. Es clave para entender la eficiencia energética.

Los resultados obtenidos se pueden observar en la siguiente tabla.

Medida	**Original**	**Nuevo**
LUTS	2295	2496
FFS	1701	1701
DSP	12	12
BUFG	5	5
Estimated power	0.033 W	0.034 W

Cuadro 7.1: Análisis de las medidas hardware

Se puede ver que existe un incremento del 8,8 % en cuanto a LUTS y del 3 % en cuanto a *estimated power*. Estos incrementos no son muy grandes, sobre todo si se pone en comparación con el aumento de *speed up* de la sección anterior. Además, los bloques más costosos, que serían los DSP o BUFG permanecen idénticos.

En la figura 7.2 se puede observar el diagrama de la FPGA del procesador con el nuevo módulo ya implementado. En verde se encuentra la memoria, en rosa el procesador y en azul las conexiones de entrada y salida. A simple vista, el procesador no ocupa una gran cantidad de espacio. Estamos trabajando con una memoria de 64 Kb y un reloj con una frecuencia de 100 MHz.

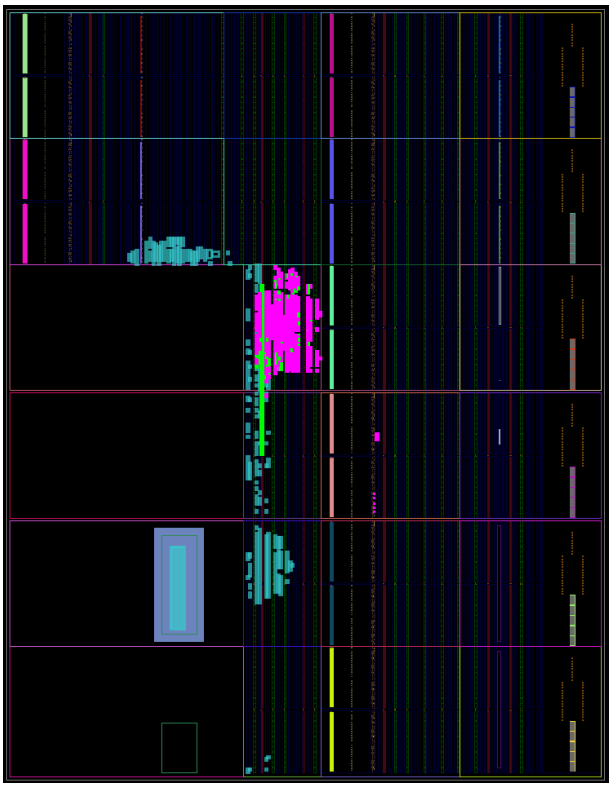

Figura 7.2: Representación de los recursos utilizados para implementar el procesador base con la nueva unidad

7.3. Análisis de consumo de energía

Al realizar una medición de consumo de una actividad, se tiene en cuenta el tiempo dedicado y la potencia que es la energía por unidad de tiempo. En este estudio, la unidad de tiempo son los ciclos. En esta sección, se va a realizar una comparación del consumo al realizar la tarea de predicción en cada una de las imágenes primero con el procesador original y después añadiendo el nuevo módulo. La fórmula para comparar los consumos de energía entre los dos procesos es la siguiente:

$$\frac{E_{\text{new}}}{E_{\text{old}}} = \frac{T_{\text{new}} * P_{\text{new}}}{T_{\text{old}} * P_{\text{old}}} = \frac{T_{\text{new}}}{T_{\text{old}}} * \frac{P_{\text{new}}}{P_{\text{old}}} \tag{7.5}$$

En esta fórmula, E_x es el consumo de energía, T_x el número de ciclos y P_x la potencia empleada. A cada una estas fracciones se le denomina incremento de energía, ciclos y potencia respectivamente. Se expresa con Δx.

Para la imagen `IP_data`, el incremento de energía es el siguiente:

$$\Delta E = \frac{3{,}092{,}770{,}718}{3{,}915{,}036{,}124} * \frac{34}{33} = 0{,}814 \tag{7.6}$$

Este resultado es menor que 1, e indica que el procesador con la nueva unidad consume un 81 % de la energía que se consumía con el procesador original. Esto quiere decir que el consumo de energía a descendido con la implementación del acelerador. Este cambio es positivo ya que indica que el procesador requiere de menos recursos energéticos para realizar la misma actividad. Para las imágenes `KSC_data` y `PU_data` los resultados del consumo energético han sido de 0.823 y 0,849 respectivamente. De nuevo, son resultados positivos que indican una mejora notable en el consumo.

Capítulo 8

Conclusiones

Una vez analizados los resultados, se puede concluir que la adición del acelerador ha sido un éxito. El coste hardware es mínimo teniendo en cuenta el poco espacio que ocupa dentro de un procesador tan pequeño. Ésto, sumado a los buenos resultados obtenidos en la métrica de *speed up* y la reducción del consumo energético en aproximadamente un 18 %, hacen que el acelerador sea rentable. A la hora de hacer el trabajo, se ha seguido una estructura modular, de acuerdo a las instrucciones previamente implementadas. Sin embargo, si se buscase obtener un coste menor de LUTS, podría optarse por romper esta modularidad. Dentro del procesador original ya existían componentes capaces de realizar las operaciones tanto de suma, and y desplazamiento. Podrían haberse reutilizado dentro del trabajo para reducir costes. Sin embargo, la implementación elegida permite la adición y supresión de este nuevo módulo sin afectar en sobremanera al resto de operaciones e instrucciones del procesador original.

La realización de este trabajo a supuesto alguna que otra dificultad que no estaba planeada en un principio. El año pasado, mi último año de carrera, estuve de Erasmus en Italia. Por lo tanto, algunas asignaturas como proyecto Hardware no las he podido cursar dentro del marco de la Universidad de Zaragoza. Ésto ha provocado que algunas nociones que pueden considerarse básicas y que son adquiridas en esta asignatura, no hayan sido tan sencillas para mí. Tanto el lenguaje ensamblador como el trabajo con las funciones inline me han supuesto un gran reto. El tener que aprender estos conocimientos prácticamente desde cero o con lo poco viso hace varios años ha provocado que el número de horas invertidas en estas partes hayan aumentado considerablemente. De la misma manera, su validación y depuración para comprobar que funcionase correctamente. Esta misma falta de conocimientos propició que la distinción entre enteros con signo y sin signo dentro de verilog haya sido otro gran dolor de cabeza. Obviamente, la extensión de signo no es igual en C que en system verilator. Al ser la primera vez que usaba este lenguaje, el entender los conceptos básicos ha supuesto una gran inversión de horas. Y el poder entender los detalles más complejos como puede ser la extensión de signo y el truncamiento de vectores ha requerido aún más.

En cuanto al resto del trabajo, las horas invertidas han sido bastante fructíferas. Ha sido un trabajo que se salía de mi idea inicial al tener pensado enlazarlo con algo relacionado con el *Machine Learning* y las redes neuronales, que es de lo que trató mi trabajo de fin de grado anterior. Sin embargo, descender hasta el más bajo nivel con el ensamblador y trabaja con circuitos combinacionales después de varios años ha sido una buena experiencia. El desarrollo paso a paso de la nueva unidad funcional ha sido un proceso lento pero gratificante. Desde analizar el entorno de trabajo y los requisitos hasta llegar a los resultados esperados pasando por todas las tareas intermedias. Trabajar con diferentes lenguajes y herramientas me ha servido, para descubrir que cualquier tarea, por pequeña que sea, requiere una gran cantidad de conocimientos varios y el entrelazado de los mismos para poder sacar la tarea adelante. También, el apoyo de otras personas ayuda a deshacer bloqueos que trabajando solo pueden parecer montañas. Por ello y para terminar, agradecer a Samuel, Javier y Adrián la colaboración semanal para sacar adelante este trabajo. El apoyo constante ha supuesto una ayuda inmensa para poder terminar todo a tiempo y con criterio.

Bibliografía

[1] Adrián Alcolea y Javier Resano. "FPGA Accelerator for Gradient Boosting Decision Trees". En: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030314. URL: https://www.mdpi.com/2079-9292/10/3/314.

[2] Adrián Alcolea et al. "Inference in Supervised Spectral Classifiers for On-Board Hyperspectral Imaging: An Overview". En: *Remote Sensing* 12.3 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12030534. URL: https://www.mdpi.com/2072-4292/12/3/534.

[3] CatBoost Developers. *CatBoost Documentation*. 2024. URL: https://catboost.ai/.

[4] Wikipedia contributors. *Hyperspectral imaging — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/wiki/Hyperspectral_imaging.

[5] Wikipedia contributors. *Matriz de puerta programable en campo — Wikipedia, La enciclopedia libre*. 2024. URL: https://es.wikipedia.org/wiki/Matriz_de_puerta_programable_en_campo.

[6] Lei Huang et al. "Normalization Techniques in Training DNNs: Methodology, Analysis and Application". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.8 (2023), págs. 10173-10196. DOI: 10.1109/TPAMI.2023.3250241.

[7] LightGBM Developers. *LightGBM Documentation*. 2024. URL: https://lightgbm.readthedocs.io/en/stable/.

[8] Duncan McElfresh et al. *When Do Neural Nets Outperform Boosted Trees on Tabular Data?* 2024. arXiv: 2305.02997 [cs.LG]. URL: https://arxiv.org/abs/2305.02997.

[9] Samulix20. *riscv_system_verilog*. 2024. URL: https://github.com/Samulix20/riscv_system_verilog.

[10] Veripool. *Verilator: The Fastest Free Verilog HDL Simulator*. 2024. URL: https://www.veripool.org/verilator/.

[11] Andrew Waterman y Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Ver. 2.2. Document Version 2.2. RISC-V Foundation. Mayo de 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[12] Dingkai Wei, Yazi Li y Mengxiao Li. "An innovative multi-factor prediction algorithm construction and importance analysis model based on GBDT-LightGBM". En: *2023 IEEE 3rd International Conference on Data Science and Computer Application (ICDSCA)*. 2023, págs. 854-858. DOI: 10.1109/ICDSCA59871.2023.10392448.

[13] Wikipedia, la enciclopedia libre. *Matriz de puerta programable en campo*. 2024. URL: https://es.wikipedia.org/wiki/Matriz_de_puerta_programable_en_campo.

[14] XGBoost Developers. *XGBoost Documentation*. 2024. URL: https://xgboost.readthedocs.io/en/stable/.

[15] Xilinx, Inc. *Xilinx Support Downloads*. 2024. URL: https://www.xilinx.com/support/download.html.

[16] Universidad de Zaragoza. *FPGA Accelerator for GBDT*. 2024. URL: https://github.com/universidad-zaragoza/FPGA_accelerator_for_GBDT/tree/main.

[17] Zg104. *Gradient Boosting Decision Trees*. URL: https://zg104.github.io/ML/GBDT (visitado 21-11-2024).

Anexos

Anexo A.

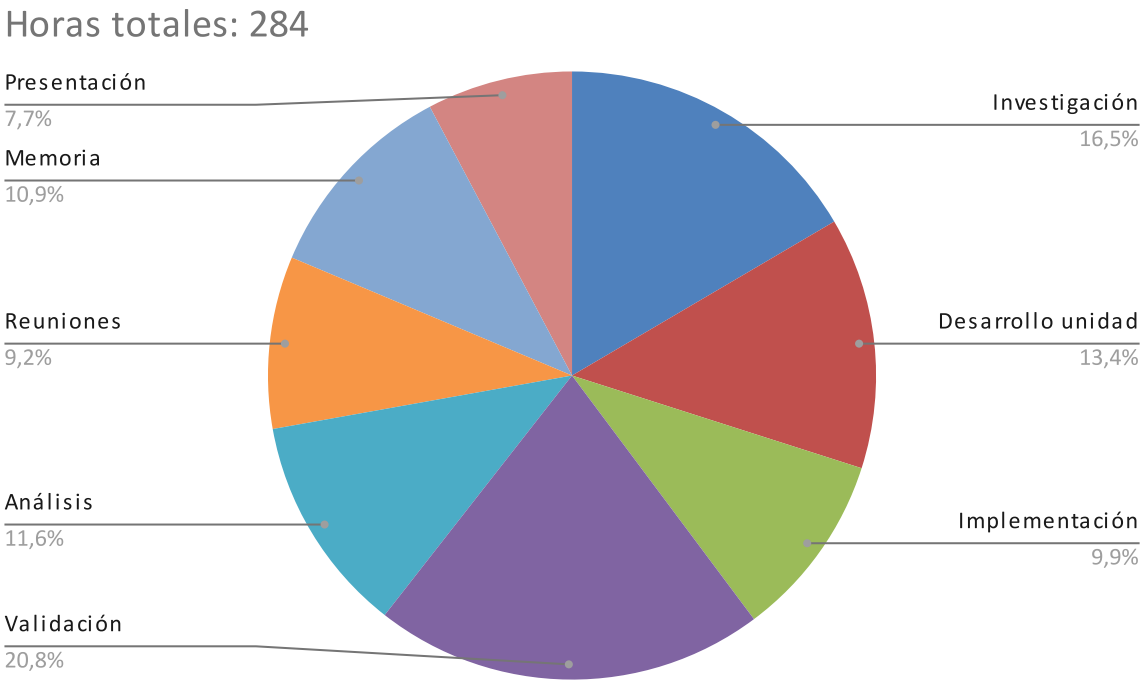

Figura 1: Gráfica de horas trabajadas desglosadas

Anexo B.

```c
#include <stdio.h>
#include "stdint.h"
#include "xtime_l.h"
#include "platform.h"
#include "xil_printf.h"

/* Datasets
//#include "IP_data.h"
//#include "KSC_data.h"
//#include "PU_data.h"

#define IS_LEAF(x)     ((x & 0x00000001) == 0x00000001)
#define PRED_VALUE(x)  ((x & 0xffff0000) >> 16)
#define NEXT_TREE(x)   ((x & 0x0000fffc) >> 2)
#define NUM_FEATURE(x) ((x & 0xff000000) >> 24)
#define CMP_VALUE(x)   ((x & 0x00ffff00) >> 8)
#define RIGHT_CHILD(x) ((x & 0x000000fe) >> 1)
#define CLASS_END(x)   ((x & 0x00000003) == 0x00000003)
```

Figura 2: Programa en C (parte 1)

```
int predict(u32 class[], u32 pixel[]){
    int prediction = 0.0;
    int curr_addr = 0;
    int end = 0;
    while(!end){
        if(IS_LEAF(class[curr_addr])){ // Leaf node
            prediction += (int16_t)PRED_VALUE(class[curr_addr]);
            if(CLASS_END(class[curr_addr])){
                end = 1;
            } else{
                curr_addr = NEXT_TREE(class[curr_addr]);
            }
        } else{ // Non leaf node
            if(pixel[NUM_FEATURE(class[curr_addr])]
                   <= CMP_VALUE(class[curr_addr])){ // Left child
                curr_addr++;
            } else{ // Right child
                curr_addr += RIGHT_CHILD(class[curr_addr]);
            }
        }
    }
    return prediction;
}

int main(){

    int pixels;

    printf("IMAGE: %s\n", IMAGE_NAME);

        int predictions[NUM_PIXELS] = {0};
        int max_prediction_value;
        int prediction_value;
        int hits = 0;

        for(int p = 0; p < NUM_PIXELS; p++){
            max_prediction_value = 0.0;
            for(int c = 0; c < NUM_CLASSES; c++){
                prediction_value = predict(class[c], pixel[p]);
                if(prediction_value > max_prediction_value){
                    max_prediction_value = prediction_value;
                    predictions[p] = c;
                }
            }
    return 0;
}
```

Figura 3: Programa en C (parte 2)

Anexo C.

DESCRIPTION	NUM_PIXELS	NUM_CLASSES	NUM_FEATURES	SIM_TIME
Complete_loop_ w_prints	40	16	200	15019980
Complete_loop_ w_prints	2	16	200	761284
Complete_loop_ w_prints	4	16	200	1508793
Complete_loop_ w_prints	4	8	200	722270
Complete_loop_ w_prints	4	4	200	352868
Complete_loop_ wo_prints	40	16	200	14185405
Complete_loop_ wo_prints	4	16	200	1423065
Complete_loop_ wo_prints	4	8	200	674933
only loop	40	16	200	14185406
predict_function	40	16	200	354924
comp_pred_w_max	40	16	200	16
inside_predict_function_wo_init	2	16	200	359086
update_curr_addr_NEXT_TREE (x179)	1	16	200	3
leaf_node (x180)	1	1	200	7 (11 END)
non_leaf_node (x558)	1	1	200	18-13
update_curr_addr_LEFT_CHILD (x83)	1	1	200	2
update_curr_addr_RIGHT_CHILD (x475)	1	1	200	3
update_prediction (x179)	1	1	200	1
if_comparator(only boolean part) (x475)	1	1	200	8
if+curr_addr_LEFT_CHILD (x83)	1	1	10	2
comparator+if	1	1	10	12
comparator+if+update_curr_addr_LEFT_CHILD	1	1	10	13
comparator+if+update+else	1	1	10	12
comparator+i+update+else+update	1	1	10	15
bool terminar (x180)	1	1	10	2
Toy_total	1	1	2	218
Toy_predict_function				205
Toy_inside_prediction				198
Toy_leaf_loop!=END (counters for non_leaf)				13
Toy_leaf_loop==END (counters for non_leaf)				15
Toy_leaf_loop!=END (no extra counters)				17
Toy_leaf_loop==END (no extra counters)				19
Toy_non_leaf_LEFT				20
Toy_non_leaf_RIGHT				22

Cuadro 1: Tabla de mediciones