UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

DESARROLLO DE UN PROCESADOR DE DOMINIO ESPECÍFICO ORIENTADO A LA IMPLEMENTACIÓN DE REDES NEURONALES ARTIFICIALES MEDIANTE INSTRUCCIONES PERSONALIZADAS DEL ISA DE RISC-V

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

DANIEL IGNACIO VÁSQUEZ PARRA

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
CRISTIÁN J. FIGUEROA SEPÚLVEDA
ANDRÉS CABA RUTTE

SANTIAGO DE CHILE
2022

Resumen

Este trabajo consta del desarrollo de un procesador de dominio específico orientado a redes neuronales convolucionales, usando instrucciones personalizadas y basado en un procesador de RISC-V, con el fin de reducir el tiempo de ejecución de la red.

El trabajo se desarrolló en varias etapas:

- Primero, un análisis de cada una de las etapas de una red CNN con el fin de comprender mejor los puntos donde se puede optimizar el tiempo de ejecución.
- Después, se plantean en detalle cada una de las instrucciones nuevas necesarias y se adaptan con el fin de que no interfieran en las instrucciones creadas ya en un procesador de RISC-V.
- En la siguiente etapa, se crea el circuito necesario para las simulaciones y se programa en el lenguaje de descripción de hardware (HDL) Verilog
- Finalmente, se realizaron las simulaciones y comparaciones con un procesador de RISC-V, para conocer en cuánto tiempo se reduce la operación de la red CNN.

Todas las implementaciones se realizaron en el software *ACTIVE HDL*, de la empresa ALDEC , y las comparaciones se realizaron con las mismas configuraciones para que ambos procesadores estuvieran en igualdad de operación.

Los resultados obtenidos en el desarrollo de este trabajo son alentadores ya que se logra reducir el tiempo de operación considerablemente, sobretodo, en las etapas de multiplicación matricial y convolución.

A mi familia en especial

a mi polola que estuvo en todo

el proceso conmigo

||

Tabla de Contenido

1. Introducción **1**

1.1. Motivación 2

1.2. Formalización del problema 2

1.3. Resultados esperados 2

1.4. Objetivos 3

2. Marco teórico y Estado del Arte **4**

2.1. Marco teórico 4

2.1.1. Redes Neuronales Artificiales (ANN) 4

2.1.2. Redes Neuronales Convolucionales (CNN) 5

2.1.3. *Assembler y RISC-V* 7

2.2. Estado del Arte 9

3. Diseño e implementación **11**

3.1. Metodología de trabajo 11

3.1.1. Parámetros de trabajo 11

3.1.2. Etapas de trabajo 11

3.2. Análisis de las principales funciones de una red CNN 12

3.2.1. Convolución 12

3.2.2. *Pooling* 13

3.2.3. Capa *Fully-conected* 14

3.3. Diseño de nuevas instrucciones 15

3.4. Diagrama de bloques 16
3.4.1. Bloque counter 17
3.4.2. Bloque cnn_decoder 17
3.4.3. Bloque cnn_memory 18
3.4.4. Bloque cnn_aluoperation 19
3.4.5. Bloque cnn_register 20
3.4.6. Bloque cnn_savemem 20
3.5. Implementación de una rutina con las instrucciones personalizadas 21
3.5.1. Ejemplo de codificación de instrucciones en hexadecimal 21

4. Resultados y Discusión **24**
4.1. Factores de configuración 24
4.1.1. Consideraciones y simplificaciones para medir tiempo efectivo 24
4.2. Resultados de las simulaciones 24
4.2.1. Resultados Convolución 25
4.2.2. Resultados MaxPool 25
4.2.3. Resultados Multiplicación matricial 26
4.2.4. Simulaciones ACTIVE HDL 27
4.3. Discusión 30
4.3.1. Rendimiento Convolución 30
4.3.2. Rendimiento MaxPool 30
4.3.3. Rendimiento Multiplicación Matricial 31

5. Conclusiones **32**
5.1. Trabajo futuro 32

Bibliografía **34**

Anexos **35**

A. Códigos verilog del procesador **36**

B. RISC-V ISA **43**

Índice de Tablas

3.1. Tabla resumen de las instrucciones creadas 16

3.2. Tabla de distribución de los bits de instrucciones, el opcode se ocupó uno sin uso para RISC-V, el 1010011. 16

4.1. Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para la convolución 25

4.2. Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para MaxPool 26

4.3. Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para multiplicación matricial 26

Índice de Ilustraciones

2.1. Diagrama de un perceptrón 4
2.2. Diagrama de una red ANN 5
2.3. Representación gráfica de la convolución 6
2.4. Máx-pool para reducir a la mitad una imagen de 4x4 6
2.5. Extensiones estándares de RISC-V 8
2.6. Copresador planteado en el paper[11] 9
2.7. Coprocesador con crossbar planteado en el paper [8] 10

3.1. Diagrama de bloques del procesador 17
3.2. Bloque counter 17
3.3. Bloque decoder 18
3.4. Bloque memory 19
3.5. Bloque aluoperation 20
3.6. Bloque register 20
3.7. Bloque Save_mem 21

4.1. Gráfico comparativo de tiempos de ejecución entre RISC-V y la propuesta de solución para la convolucion 25
4.2. Gráfico comparativo de tiempos de procesamiento entre RISC-V y la propuesta de solución para MaxPool 26
4.3. Gráfico comparativo de tiempos de procesamiento entre RISC-V y la propuesta de solución para la multiplicación matricial 27
4.4. Simulación en active HDL: en rojo se observa la instrucción 0, que es el reset del registro interno. 27

4.5. Simulación en active HDL: en rojo se observa la instrucción 1, que es multiplicipación entre input 1 y 2 más lo que contiene el registro. 28

4.6. Simulación en active HDL: en rojo se observa la instrucción 6, que guarda el valor que contine el registro interno. 28

4.7. Simulación en active HDL: en rojo se observa que el registro guardo el valor esperado. 28

4.8. Simulación en active HDL: en rojo se observa la instrucción 0, que es el reset del registro interno. 29

4.9. Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 11-20 29

4.10. Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 21-30 29

4.11. Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 31-40 29

Capítulo 1

Introducción

No es nuevo que la tecnología cada día crece a pasos agigantados, considerando que hace 40 años, para la mayor parte de la población, era impensado un teléfono inteligente o cámaras que pudieran reconocer nuestro rostro e interpretar quién es la persona que está en pantalla.

Actualmente, esta tecnología está basada en redes neuronales convolucionales (CNN), lo cual es relativamente nuevo, considerando que la primera red CNN fue diseñada hace 40 años solamente. Las redes CNN tienen múltiples usos, siendo uno de los más importantes el reconocimiento de imágenes. Para el reconocimiento de imágenes, estas aplican filtros extrayendo las características más importantes y finalmente clasificando estas imágenes, por ejemplo, reconocer qué animal se encuentra en una imagen.

Si bien esta tecnología existe y funciona correctamente, en ocaciones puede tomar mucho tiempo el procesamiento de estas, sobretodo cuando se presenta un gran número de imágenes, lo cual frecuentemente ocurre, por lo que este trabajo se enfocará en tratar de reducir el tiempo de operación de estas redes.

Las opciones a considerar para mejorar la *performance* es, primeramente, la opción de trabajar con un procesador más rápido, lo cual no es viable, principalmente por 2 razones: primero, el costo de este procesador sería muy elevado en comparación con uno estándar del mercado y, segundo, no es físicamente posible ya que no se puede implementar un procesador mucho más rápido debido a problemas con la disipación de calor.

Esto por esto que la mejor opción es realizar un procesador de dominio específico para estas redes; utilizando como base un procesador de RISC-V.

RISC-V es un procesador que tiene la particularidad de que su ISA(set de instrucciones) son móviles y de código abierto, es decir, se pueden agregar instrucciones si el usuario lo requiere y son libres y gratuitas para usar por cualquier desarrollador.

Este trabajo consta de 5 capítulos principales:

1. Introducción
2. Marco teórico y estado del arte

3. Diseño e Implementación
4. Resultados y discusión
5. Conclusiones

Cada uno de estos capítulos se mencionan en detalle más adelante y cómo fue el proceso creativo para este trabajo. Finalmente, están los resultados y la discusión donde se analiza la solución propuesta y cómo se ve afectado el tiempo de procesamiento de una red, además de propuestas para continuar este trabajo a futuro.

1.1. Motivación

La motivación para realizar esta memoria es agilizar procesos que involucren procesamientos de imágenes. El gran beneficio que traería consigo conseguir esto es mejorar procesos que involucren tecnología de procesamientos de imágenes cómo la IA de los autos (como la asistencia de estacionamiento, o la asistencia de piloto automático), tratamientos médicos modernos que involucran tecnología para asistir al médico, mejorar el tiempo de procesamiento de redes con gran volumen de imágenes, entre otras.

1.2. Formalización del problema

Como ya se ha mencionado con anterioridad, el objetivo principal de este trabajo es mejorar la *performance* de una red CNN, de tal manera que la ejecución sea en el menor tiempo posible. Todo esto, utilizando instrucciones personalizadas de la ISA de un procesador basado en RISC-V.

En particular, la finalidad es crear instrucciones específicas para el procesador, de tal manera que estas sean de uso exclusivo para redes CNN.

Para la realización de esta memoria, se plantea realizar un procesador de una instrucción por ciclo (*single-cycle*). Realizar un procesador simplificado desde cero en Verilog, nos permite ingresar instrucciones personalizadas, muy específicas, que mejoren la *performance*.

Como se explica en mayor detalle en el capítulo de Diseño e Implementación, a un procesador basado en RISC-V se le agrega una memoria que retroalimenta a la ALU con la finalidad de guardar registros de operaciones para luego entregarlas todas en el momento necesario.

1.3. Resultados esperados

Lo que se espera de este trabajo, es lograr una reducción considerable del tiempo de operación de una red CNN, en especial, de las subrutinas de la red como la convolución o la

multiplicación matricial. Todo esto, manteniendo la misma estructura de una procesador de RISC-V y aprovechando los espacios disponibles para nuevas instrucciones de tal forma que se pueda indexar, posteriormente, a un procesador completo.

1.4. Objetivos

El objetivo principal de esta memoria es reducir el tiempo de procesamiento de una o más etapas de una red neuronal convolucional. Para lograr el objetivo principal se plantean los siguientes objetivos específicos.

- Diseñar instrucciones personalizadas para un procesador basado en RISC-V
- Implementar las instrucciones en un procesador simplificado basado en RISC-V
- Implementar códigos en assembler con las nuevas instrucciones para probar el correcto funcionamiento del procesador

Capítulo 2

Marco teórico y Estado del Arte

2.1. Marco teórico

2.1.1. Redes Neuronales Artificiales (ANN)

Las Redes Neuronales Artificiales, ANN (Artificial Neural Networks), tienen este nombre, porque tratan de emular al cerebro humano, en específico, a las neuronas biológicas [10].

Las ANN están muy lejos de igualar a un cerebro humano, sin embargo, presentan varias características en común con este. Por ejemplo, las ANN aprenden mediante ejemplos, son capaces de generalizar en base a estos y abstraen las características principales.

Así como en el cerebro humano su unidad básica es una neurona, para una ANN, su unidad básica es el perceptrón (PE). El PE tiene varias entradas, y una combinación matemática de estas, nos entrega una salida que a su vez se conecta con otros PE

La Figura 2.1 representa un perceptrón de una red neuronal artificial implementada.

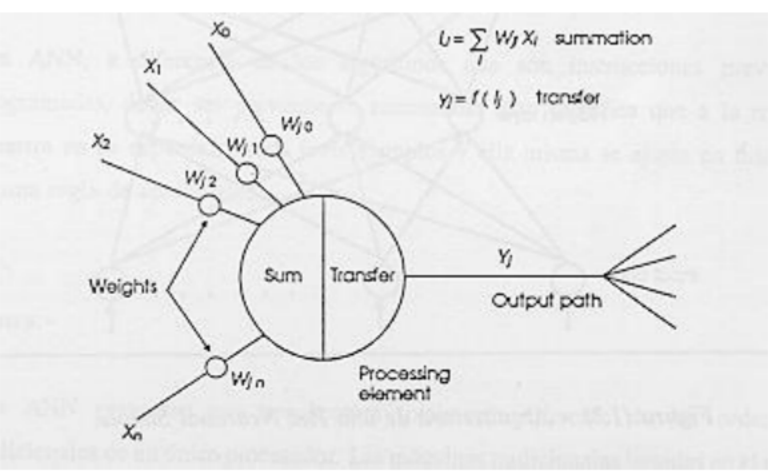

Figura 2.1: Diagrama de un perceptrón

Una red neuronal consta de una gran cantidad de PE, asociados principalmente en capas, las cuales van interconectadas entre sí [3]. Las capas que más destacan son:

- Capa de entrada: es donde ingresa la información
- Capas ocultas: una red ANN tiene muchas capas ocultas las que son encargadas de interpretar esta información.
- Capa de salida: es la última capa de la red, es donde se entrega la información clasificada.

El la figura 2.2 muestra una red ANN

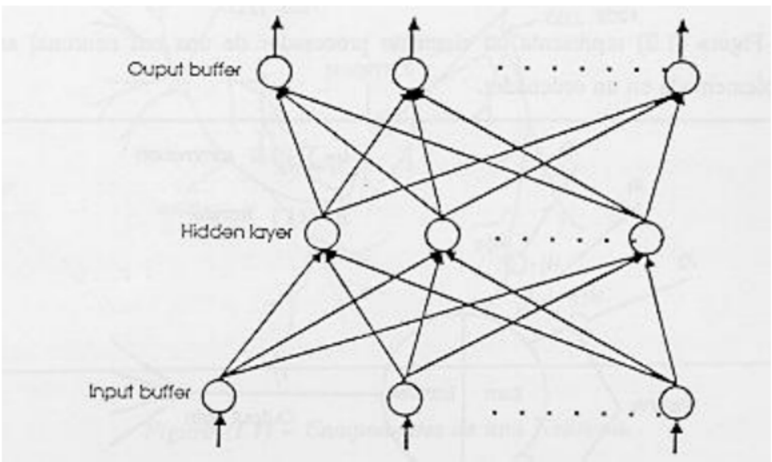

Figura 2.2: Diagrama de una red ANN

2.1.2. Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (*convolutional neural networks*, CNN) es un tipo de ANN, pero, más enfocada a objetos que necesitan convolución como imágenes. La mayor diferencia entre una CNN y una ANN, es que, para la CNN cambia el concepto de perceptrón por la convolución y capas de *subsampling*

Uno de los primeros desarrollos de CNN, fue por parte de Kunihiko Fukushima en el año 1982. Kunihiko desarrolló el *neocognitron* una red neuronal de tipo *backpropagation* que imita el proceso del *cortex* visual [5]. Posteriormente, en 1998, Yann LeCun entrenó una de las primeras CNN con éxito denominada LeNet la cual emplea una estructura de red neuronal convolucional y consiguiendo clasificar imágenes de dígitos escritos a mano con una precisión del 99,3 %.

Pero fue en 2012, con la creación de la red AlexNet (Geoffrey Hinton, Ilya Sutskever y Alex Krizhevsky) para el concurso anual ImageNet Large Scale Visual Recognition Challenge (ILSVRC), cuando se observó la potencia de las redes neuronales para la clasificación de imágenes.

Estas redes constan de varias etapas las cuales se detallarán a continuación:

Convolución

La convolución es la etapa más peculiar de las redes, ya que las distingue de cualquier otra red que se pueda usar. Esta consiste en tomar un *pixel* y los que están a su alrededor y hacer un producto punto con un *kernel*. El *kernel* funciona como filtro de la imagen y, dependiendo de los valores que este tenga, es como se verá la salida.

La figura 2.3 muestra un ejemplo de convolución

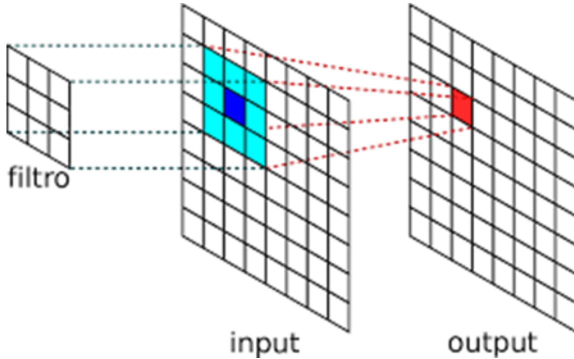

Figura 2.3: Representación gráfica de la convolución

Posterior a la convolución, generalmente, se aplica una función de activación para recalcar la información más importante. Las principales funciones que se usan son RELU y sigmodal.

Pooling

Luego de la etapa de convolución, viene una etapa de *pooling*. Aquí es necesario disminuir las dimensiones de la imagen, dejando la información más importante. El *pooling* más común es *max-pooling*, donde se subdivide la imagen en pequeñas matrices y se elige el mayor valor de ellos.

En la figura 2.4 se ve un ejemplo de *max-pool*

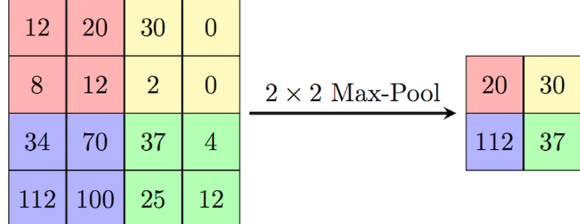

Figura 2.4: Máx-pool para reducir a la mitad una imagen de 4x4

Etapa *fully conected*

Las etapas de convolución y *pooling* se aplican varias veces hasta, finalmente, aplicar una capa de una red neuronal tradicional, donde se cargan ciertos pesos y, mediante una multiplicación matricial, tenemos los valores previos a la salida.

Softmax

Etapa final donde se normalizan los valores de la última capa y entrega la probabilidad de que cada imagen pertenezca a una clase. También, se ocupa con código one-hot, donde el 1 indica pertenencia a la clase.

2.1.3. *Assembler y RISC-V*

Assembler

El lenguaje *Assembler*, es un lenguaje de programación de bajo nivel. El lenguaje Assembler puede cambiar según el procesador que se este usando, ya que las instrucciones dependen exclusivamente del tipo de procesador con el que se trabaje.

Los lenguajes de alto nivel, como Python, JAVA, entre otros, son llamados así, ya que, la mayor parte de sus instrucciones son en inglés, por lo cual es más simple y rápido que un lenguaje *Assembler*. Es por esto, que los lenguajes de alto nivel son dependientes de un interprete que traduzca sus instrucciones a lenguaje máquina.

RISC-V

RISC-V es un procesador con una ISA de código abierto con 3 extensiones básicas: RV32I, RV32E y RV64I. Estas extensiones poseen las instrucciones correspondientes para el manejo de números enteros, donde RV32I y RV32E utilizan (y por tanto requieren de) respectivamente, 32 y 16 registros de 32 bits, incluyendo el registro de constante cero x0, donde RV32E es usado particularmente en procesadores más pequeños. Por otro lado, la extensión RV64I contiene las mismas instrucciones que RV32I, sin embargo, se amplia de 32 a 64 bits.

RISC-V se caracteriza por tener una ISA modular, característica que le permite agregar o no, las distintas extensiones según requiera el usuario. Las extensiones que dispone un procesador son definidas por la arquitectura física de este. Las extensiones estándares son las indicadas en la tabla de la Figura 2.5

Extensión	Descripción breve
I/E	Es la extensión base de RISC-V. Posee operaciones basicas de numeros enteros y trabaja sobre 32 y 16 registros respectivamente
M	Operaciones de multiplicación y división de enteros
C	Instrucciones "Compactas" utilizadas para operaciones de 16 bits y a menor costo de memoria
F	Instrucciones para el manejo de Punto Flotante de simple precisión
D	Instrucciones para el manejo de Punto Flotante de doble precisión
A	Instrucciones para operaciones en memoria atómica

Figura 2.5: Extensiones estándares de RISC-V

ISA

ISA, es el conjunto de instrucciones que contiene un procesador. Estas instrucciones son utilizadas, principalmente, para indicar al procesador las operaciones que debe realizar de acuerdo al orden que se le asigne.

La ISA de RISV-V tiene la particularidad de ser del tipo modular, es por esto que las instrucciones que contiene no son fijas, se pueden añadir o retirar instrucciones, inclusive, se pueden agregar instrucciones nuevas si así lo requiere el usuario.

Aunque existe una convención sobre la ISA de un procesador, cada procesador cuenta con su ISA única y es importante verificar el ISA con el que trabaja el procesador.

ALU

La ALU (*arithmetic logic unit*) es la unidad principal de procesamiento que contiene cualquier procesador. Es la encargada de realizar todas las operaciones matemáticas necesarias para el correcto funcionamiento del procesador.

Las ALU se caracterizan por tener al menos los siguientes elementos:

- Dos entradas de múltiples bits para los *inputs* a operar
- Selector de operaciones, que le indica a la ALU que operación realizar

- Banderas, que indican alguna anomalía en las operaciones como *overflow*, *underflow*, entre otras.
- *Output*, que muestra la salida de la operación entre ambos *inputs*

La ALU, sólo realiza operaciones, no toma decisiones, las entradas deben contener toda la información necesaria para la operación (signo, magnitud,etc), y la ALU requiere un mecanismo de operación que generalmente se trata de un *decoder*.

2.2. Estado del Arte

EL implementar instrucciones personalizadas en RISC-V enfocadas en redes CNN, es un problema relativamente nuevo y poco documentado, sin embargo, ya hay acercamientos a estos.

El paper "*A Heterogeneous Processor Design for CNN-Based AI Applications on IoT Devices*"[11] fue la principal introducción para este tema, debido a que los puntos más importantes de optimización están en las etapas que se verán más adelante. Sin embargo, la gran diferencia de es que se plantea un co-procesador que ayude en algunas instrucciones especificas al procesador principal, como se muestra en la figura 2.6.

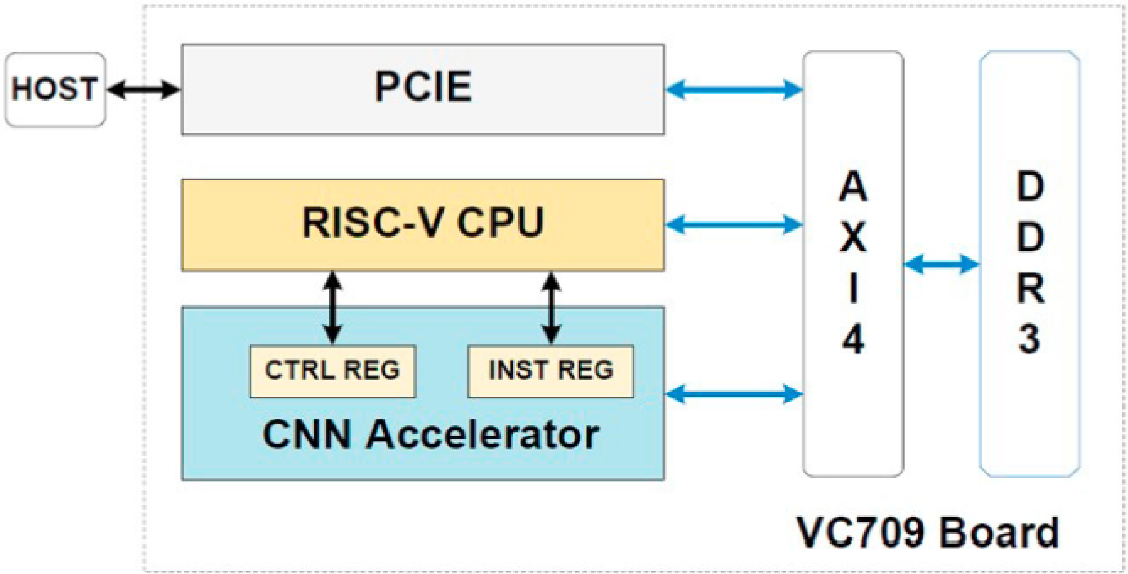

Figura 2.6: Copresador planteado en el paper[11]

La idea que se rescata de este procesador es el uso de registros inmediatos, para agilizar el proceso y verificar más la parte funcional que la parte estructural del problema.

Del paper "*A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set*"[8], se plantea una idea similar al paper antes citado, sin embargo, la diferencia se presenta en cómo se muestran las instrucciones, con el uso de un *crossbar* que funciona como selección de funciones, como se ve en la figura 2.7.

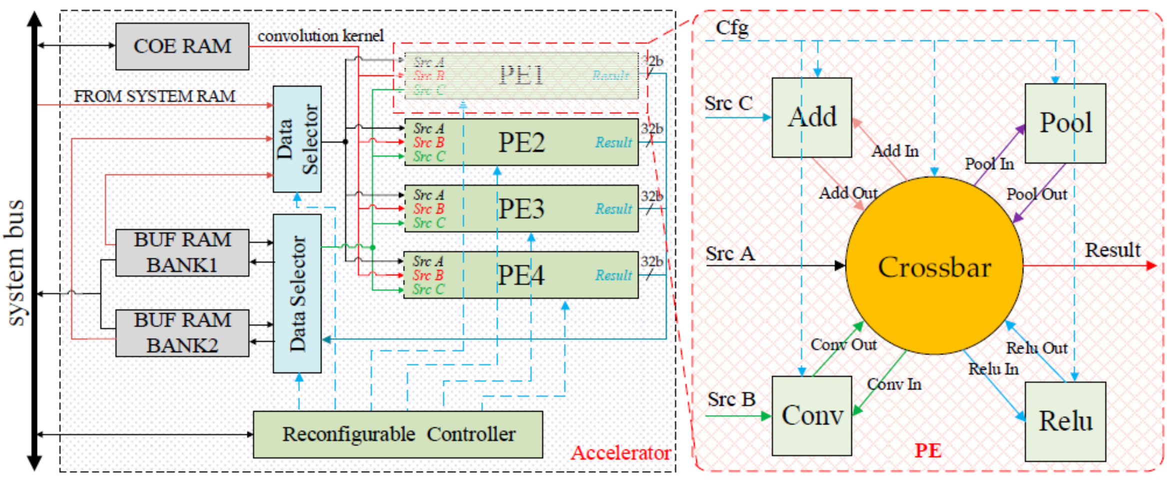

Figura 2.7: Coprocesador con crossbar planteado en el paper [8]

De este paper más el anterior, es de donde se encontró la inspiración para escribir esta memoria, y se plantearon los puntos importantes a trabajar, que es crear una ALU similar al *crossbar*, y trabajar con registros rápidos.

Capítulo 3

Diseño e implementación

3.1. Metodología de trabajo

3.1.1. Parámetros de trabajo

El desarrollo de este trabajo se realizó con el software ACTIVE-HDL de la empresa ALDEC en la versión 11.1. En este software se diseñó, en lenguaje Verilog, cada uno de los bloques necesarios para el correcto funcionamiento del procesador propuesto y mediante un texto con las instrucciones en hexadecimal, cada una en una linea diferente que se agregan como memoria al programa para que pueda realizar la rutina propuesta por el usuario. Al ser instrucciones personalizadas, el proceso de traducir el lenguaje assembler a hexadecimal se realizó de manera manual, instrucción por instrucción.

Para comprobar el correcto funcionamiento del procesador, es necesario observar la memoria y cómo esta cambia cuando se aplican las distintas funciones.

En cuanto a la configuración del programa, se utilizó una frecuencia de reloj de 10MHz para todas las simulaciones realizadas.

3.1.2. Etapas de trabajo

Para cumplir el objetivo principal, que es construir un procesador que sea capaz de procesar una red CNN a una velocidad mayor a la que se ejecuta actualmente, es necesario seguir una serie de pasos los cuales se describen a continuación:

- Análisis de las principales funciones de una red CNN: esta etapa consiste en observar el código fuente y ver en qué puntos se pueden cambiar una operación para obtener mejoras y, así, optimizar el tiempo de ejecución. De aquí, se desprenderán las nuevas instrucciones.
- Diseño de nuevas instrucciones: a partir de lo anterior, es necesario diseñar las nuevas

instrucciones que serán la base del procesador propuesto. Estas instrucciones deben estar al detalle, de tal forma que se puedan indexar a un procesador de RISC-V de manera fácil, por lo que se ocupará un *opcode* disponible.

- Diagrama de bloques: en esta etapa se diseña el circuito necesario para el correcto funcionamiento del procesador, donde posteriormente el circuito será implementado en lenguaje descriptivo (Verilog) y posteriormente simulado.
- Implementación de la rutina con las instrucciones personalizadas: en esta etapa se implementa una rutina en assembler, utilizando las instrucciones nuevas, para comprobar el correcto funcionamiento de las nuevas instrucciones y medir el tiempo en que se completa esta rutina.

3.2. Análisis de las principales funciones de una red CNN

3.2.1. Convolución

La convolución discretizada para las imágenes, no es más que el producto punto entre una sección de la imagen y un filtro o *kernel*. El *kernel* es una matriz cuadrada y sus dimensiones más comunes son 3x3 y 5x5. El código de cada una de las convoluciones de la imagen se puede resumir según el código del *listing* 3.1

```
1       for i in range(0,len(kernel)):
2       for j in range(0,len(kernel(0)):
3            Acumulador =image[i][j]*kernel[i][j]
4            Convolucion[i][j]=Convolucion[i][j]+Acumulador
```

Listing 3.1: Pseudocódigo convolución

Del código es posible observar que la instrucción donde se adhiere la multiplicación con el resultado previo de la casilla de convolución, es repetitiva y se podría encontrar una mejora en este punto. Para esto, se proponen las instrucciones cnn.mult y cnn.show que se explican, con mayor detalle, más adelante. En el *listing* 3.2 se da un ejemplo de convolución, para un kernel de 3x3, en *assembler* y en el *listing* 3.3, se muestra el mismo ejemplo, pero, con las instrucciones personalizadas.

```
1 #Assembler
2 #Primero se realizan las multiplicaciones, de la casilla de la matriz por
3 #la casilla correspondiende del kernel, y se guardan en variables
4 #temporales
5 
6 mult t1,a0,ker00
7 mult t2,a1,ker01
8 mult t3,a2,ker02
9 mult t4,a3,ker10
10 mult t5,a4,ker11
11 mult t6,a5,ker12
12 mult t7,a6,ker20
13 mult t8,a7,ker21
14 mult t9,a8,ker22
```

```
15
16 #Se hace la suma de las variables temporales
17
18 add t1,t1,t2
19 add t1,t1,t3
20 add t1,t1,t4
21 add t1,t1,t5
22 add t1,t1,t6
23 add t1,t1,t7
24 add t1,t1,t8
25
26 #Se guarda el valor de la operacion
27
28 add s1,t1,t9
```

Listing 3.2: Ejemplo de código para convolución en assembler

```
1 #Instrucciones personalizadas
2 #Se resetea el registro extra para que parta en 0
3
4 cnn.reset
5
6 #Se realizan las multiplicaciones y la nueva ALU, hace las sumas
7 #de manera automatica
8
9 cnn.mul a0i,a0j,ker0i,ker0j
10 cnn.mul a1i,a1j,ker0i,ker1j
11 cnn.mul a2i,a2j,ker0i,ker2j
12 cnn.mul a3i,a3j,ker1i,ker0j
13 cnn.mul a4i,a4j,ker1i,ker1j
14 cnn.mul a5i,a5j,ker1i,ker2j
15 cnn.mul a6i,a6j,ker2i,ker0j
16 cnn.mul a7i,a7j,ker2i,ker1j
17 cnn.mul a8i,a8j,ker2i,ker2j
18
19 #Se guarda el resultado de la suma de todas las multiplicaciones y,
20 #se reinicia el registro para realizar nuevamente las operaciones
21
22 cnn.show s1,1
```

Listing 3.3: Ejemplo de código para convolución en assembler usando instrucciones personalizadas

3.2.2. *Pooling*

La etapa de *pooling*, permite principalmente, reducir la imagen. Para esto, se elige cuánto se quiere reducir y se plantean micro-matrices donde se aplica alguna transformación para obtener el nuevo número correspondiente a la imagen. La transformada más común es *max-pool*, donde se selecciona el mayor valor posible dentro de las micro-matrices. Otra forma común de hacer *pooling*, es tomar el promedio de los valores de esta micro-matriz. Para estas dos funciones, se plantean las instrucciones cnn.max, cnn.min, cnn.sum y cnn.prom que se detallan en la sección de diseño de nuevas instrucciones. En el *listing* 3.4 se da un ejemplo de

maxpool, reduciendo a la mitad, en assembler y en el *listing* 3.5, se muestra el mismo ejemplo, pero, con las instrucciones personalizadas

```
#Assembler
#Se busca encontrar que numero es mayor entre a1,a2,a3,a4
#Se elige el mayor de a0,a1

sgt t1,a0,a1

#Se elige el mayor del resultado anterior con a2

sgt t1,t1,a2

#Se guarda el mayor entre el resultado anterior y a3

sgt s1,t1,a3
```

Listing 3.4: Ejemplo de código para pooling en assembler

```
#Propuesta solucion
#Se busca encontrar que numero es mayor entre a1,a2,a3,a4
#Se compara el maximo entre el registro extra, a0 y a1 y se guarda en el
#registro

cnn.max a0,a1

#Se compara el maximo entre el registro extra, a2 y a3

cnn.max a2,a3

#Se muestra el maximo de los 4 numeros

cnn.show t1,1
```

Listing 3.5: Ejemplo de código para pooling en assembler usando instrucciones personalizadas

3.2.3. Capa *Fully-conected*

La multiplicación matricial al igual que la convolución tiene muchas multiplicaciones y sumas de estos resultados acumulados, por lo que, para poder realizarlas se pueden ocupar las mismas instrucciones que para la convolución. En el *listing* 3.6 se da un ejemplo de una parte de la multiplicación matricial en assembler y en el *listing* 3.7, se muestra el mismo ejemplo, pero, con las instrucciones personalizadas.

```
#Assembler, se repite este codigo 9 veces para una matriz de 3x3
#Se calcula casilla por casilla
#Se realizan las multiplicaciones de la primera matriz por la segunda

mult t1,m00,n00
mult t2,m10,n01
mult t3,m20,n02

#Se suman estos valores
add s00,t1,t2
```

```
#Se guarda el valor final
add s00,s00,t2
```

Listing 3.6: Ejemplo de código para una parte de la multiplicación matricial en assembler

```
#Propuesta solucion
#Se realizan las multiplicaciones

cnn.mul m0i,m0j,n0i,n0j
cnn.mul m1i,m0j,n0i,n1j
cnn.mul m2i,m0j,n0i,n2j

#Se guarda el resultado
cnn.show s00,1
```

Listing 3.7: Ejemplo de código para una parte de la multiplicación matricial en assembler con instrucciones personalizadas

3.3. Diseño de nuevas instrucciones

Lo principal para el uso de nuevas aplicaciones es crear una ALU de 3 entradas donde 2 son valores desde la memoria y el tercero proviene de un registro conectado a la salida de la misma ALU. Esto, para que la ALU tenga retroalimentación instantánea de la misma, con la finalidad de que pueda recordar lo que hizo en el ejercicio anterior y evitar tener variables acumuladoras.

Para esto es necesario entonces crear nuevas instrucciones que cumplan tanto con la ALU de 3 entradas como con las instrucciones mencionandas con anterioridad. Estas son:

- cnn.reset: operación que deja el registro de destino en 0, no necesita guardar en la memoria el resultado de la operación.
- cnn.mult: operación que multiplica los 2 *inputs* y suma este valor a lo que contiene el registro; no hay necesidad de guardar este resultado en la memoria.
- cnn.sum: operación que suma los 2 inputs y suma este valor a lo que contiene el registro; no hay necesidad de guardar este resultado en la memoria.
- cnn.max: operación que compara el valor de ambos inputs y lo que hay en el registro y deja el mayor valor de estos; no hay necesidad de guardar este resultado en la memoria.
- cnn.min: operación que compara el valor de ambos inputs y lo que hay en el registro y deja el menor valor de estos; no hay necesidad de guardar este resultado en la memoria.
- cnn.prom: recibe sólo un input como entrada y entrega, como salida, el cuociente entre el registro y el input1. Este valor se guarda.
- cnn.show: recibe un entero como entrada y entrega, como salida, la multiplicación del entero con el registro. Este valor se guarda.

- cnn.div: recibe sólo un input como entrada y entrega, como salida, el cuociente entre el input1 y el registro. Este valor se guarda.

El detalle de estas operaciones se resumen en la tabla 3.1. Donde cada una de las líneas indica lo siguiente:

- Primero, están las instrucciones.
- Segundo, se encuentra el tipo de instrucción; esto se ve al detalle en la tabla 3.2.
- Tercero, se encuentra el código que tendrá esta instrucción en la ALU.
- Cuarto, quinto, se muestra con una x si no es necesario un input como en la operación reset, e in1, in2 para índicar que sí necesita un valor para operar. En el caso de la función show, se le entrega un entero en vez de un input de un registtro.
- Sexto, aquí se muestra que pasa con la memoria, donde el valor reg es el valor que tuvo en el ejercicio anterior.
- Y por último, se encuentra la salida de la ALU, donde no siempre tiene un valor ya que ese valor se está acumulando para luego mostrarse.

	instrucción	código	Input1	Input2	reg	cnn.output
cnn.reset	UC	00000	x	x	0	x
cnn.mult	RC	00001	in1	in2	reg+in1*in2	x
cnn.sum	RC	00010	in1	in2	reg+in1+in3	x
cnn.max	RC	00011	in1	in2	max(reg,in1,in2)	x
cnn.min	RC	00100	in1	in2	min(reg,in1,in2)	x
cnn.prom	IC	00101	in1	x	reg	reg/in1
cnn.show	IC	00110	int		reg	int*reg
cnn.div	IC	00111	in1	x	reg	in1/reg

Tabla 3.1: Tabla resumen de las instrucciones creadas

	31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
RC	funct5	rs1i	rs1j	rs2i/rdi	rs2j/rdj	opcode
IC	funct5	int10		rdi	rdj	opcode
UC	funct5	imm[26:7]				opcode

Tabla 3.2: Tabla de distribución de los bits de instrucciones, el opcode se ocupó uno sin uso para RISC-V, el 1010011.

3.4. Diagrama de bloques

Para la realización del procesador fue necesario trabajar en bloques el problema, con el fin de dividir el trabajo y poder ir probando por partes su correcto funcionamiento.

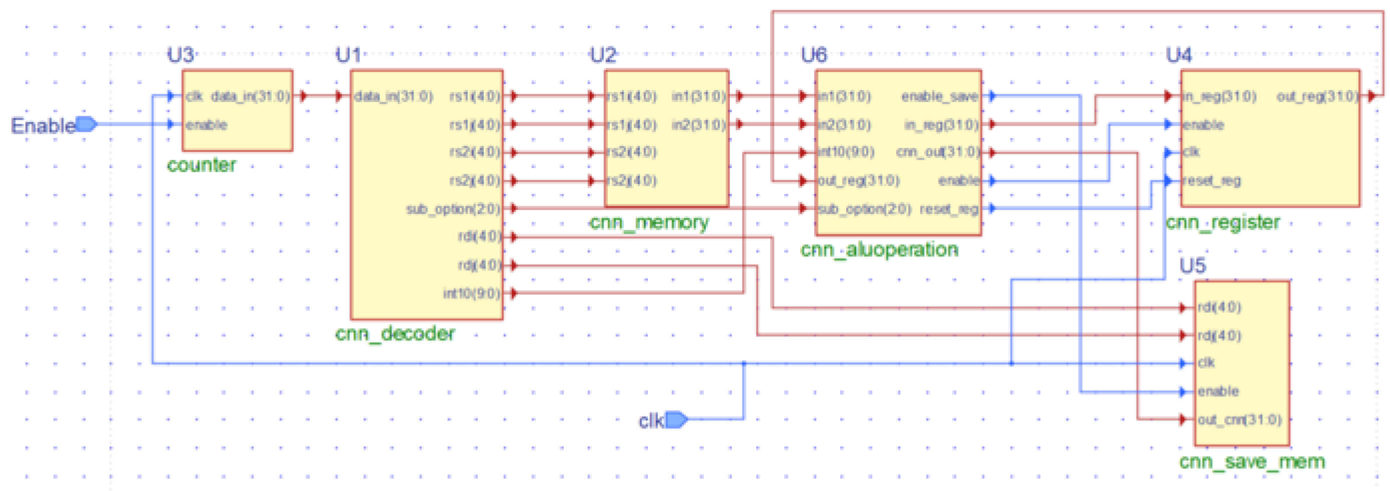

Figura 3.1: Diagrama de bloques del procesador

Como se observa en la figura 3.1 consta de 6 bloques prinipales los cuales se detallan a continuación.

3.4.1. Bloque counter

El bloque counter es el encargado de leer la memoria de instrucciones y entregarlas una a una. Tiene los siguientes I/O:

- Enable: input de 1 bit, que activa y desactiva la cuenta del counter
- Clk: input de 1 bit, es el reloj del sistema
- Data_in[31:0]: output de 32 bits que entrega la instrucción que tiene que hacer el procesador

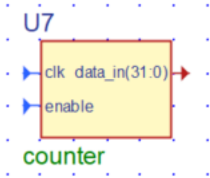

Figura 3.2: Bloque counter

3.4.2. Bloque cnn_decoder

El bloque cnn_decoder es el encargado de leer las instrucciones y entregar por parte las distintas señales para que funcione el sistema. Cuenta con las siguientes I/O:

- Data_in[31:0]: input de 32 bits que entrega la instrucción que tiene que hacer el procesador
- rs1i[4:0]: output de 5 bits que entrega la posición i de la dirección del primer registro a operar en la ALU
- rs1j[4:0]: output de 5 bits que entrega la posición j de la dirección del primer registro a operar en la ALU
- rs2i[4:0]: output de 5 bits que entrega la posición i de la dirección del segundo registro a operar en la ALU
- rs2j[4:0]: output de 5 bits que entrega la posición j de la dirección del segundo registro a operar en la ALU
- sub_option[2:0]: output de 3 bits que entrega la instrucción a realizar por el procesador.
- rdi[4:0]: output de 5 bits que entrega la posición i de la dirección del registro donde se guardará la operación de la ALU.
- rdj[4:0]: output de 5 bits que entrega la posición j de la dirección del registro donde se guardará la operación de la ALU.
- int10[9:0]: output de 10 bits que entrega un número entero de hasta 10 bits.

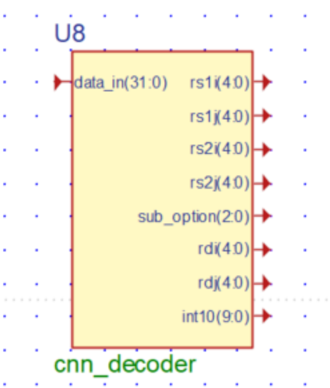

Figura 3.3: Bloque decoder

3.4.3. Bloque cnn_memory

Bloque encargado de leer la memoria según los registros y entregar las entradas para operar en la ALU. Consta de las siguientes I/O:

- rs1i[4:0]: input de 5 bits que entrega la posición i de la dirección del primer registro a operar en la ALU
- rs1j[4:0]: input de 5 bits que entrega la posición j de la dirección del primer registro a operar en la ALU

- rs2i[4:0]: input de 5 bits que entrega la posición i de la dirección del segundo registro a operar en la ALU
- rs2j[4:0]: input de 5 bits que entrega la posición j de la dirección del segundo registro a operar en la ALU
- in1[31:0]: output de 32 bits que entrega el primer operando de la ALU
- in2[31:0]: output de 32 bits que entrega el segundo operando de la ALU

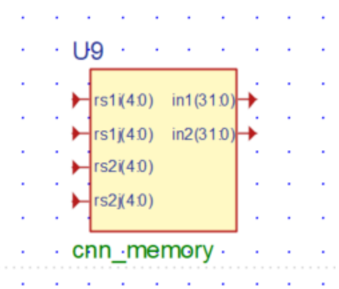

Figura 3.4: Bloque memory

3.4.4. Bloque cnn_aluoperation

Bloque encargado de realizar las operaciones correspondientes, es la *arithmetic logic unit* (ALU). Tienes los siguientes I/O:

- in1[31:0]: input de 32 bits que entrega el primer operando de la ALU
- in2[31:0]: input de 32 bits que entrega el segundo operando de la ALU
- int10[9:0]: input de 10 bits que entrega un número entero de hasta 10 bits.
- out_reg[31:0]: input de 32 bits que entregar el valor que contiene el registro de operación.
- sub_option[2:0]: input de 3 bits que entrega la instrucción a realizar por el procesador.
- enable_save: output de 1 bit que indica si el debe o no escribir en la memoria
- in_reg[31:0]: output de 32 bits que actualiza el resultado del registro de operación
- cnn_out[31:0]: output de 32 bits que tiene el valor de salida de la cnn
- enable: output de 1 bit que índica si debe o no escribir el registro de operación
- reset_reg: output de 1 bit que indica si se debe reiniciar el valor del registro de operación.

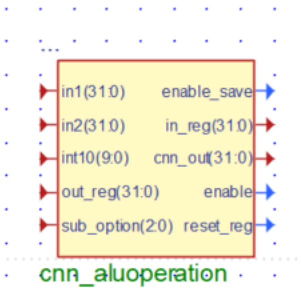

Figura 3.5: Bloque aluoperation

3.4.5. Bloque cnn_register

Es el registro de operaciones, sirve para acumular variables para algunos procesos útiles. Consta de las siguientes I/O:

- in_reg[31:0]: input de 32 bits que actualiza el resultado del registro de operación
- enable: input de 1 bit que índica si debe o no escribir el registro de operación
- Clk: input de 1 bit, es el reloj del sistema
- reset_reg: input de 1 bit que indica si se debe reiniciar el valor del registro de operación.
- out_reg[31:0]: output de 32 bits que entrega el valor que contiene el registro de operación.

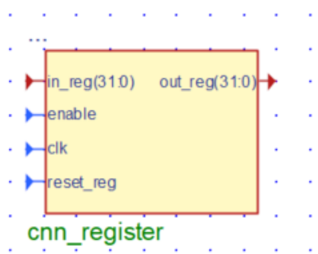

Figura 3.6: Bloque register

3.4.6. Bloque cnn_savemem

Bloque encargado de guardar en la memoria los datos obtenidos de la operación de la ALU. Solo tiene Inputs.

- rdi[4:0]: input de 5 bits que entrega la posición i de la dirección del registro donde se guardará la operación de la ALU.

- rdj[4:0]: input de 5 bits que entrega la posición j de la dirección del registro donde se guardará la operación de la ALU.
- Clk: input de 1 bit, es el reloj del sistema
- enable_save: Input de 1 bit que indica si el debe o no escribir en la memoria
- cnn_out[31:0]: input de 32 bits que tiene el valor de salida de la cnn

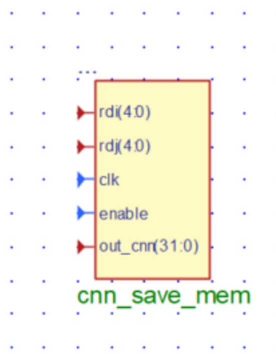

Figura 3.7: Bloque Save_mem

3.5. Implementación de una rutina con las instrucciones personalizadas

Para la implementación del programa es necesario crear un archivo *.data*. Este archivo contendrá cada una de las instrucciones requeridas por el usuario, separadas por una línea de código. Cómo estas instrucciones no están definidas en lenguaje assembler, es necesario que el usuario traduzca cada una de las instrucciones a hexadecimal.

3.5.1. Ejemplo de codificación de instrucciones en hexadecimal

Para este ejemplo utilizaremos la instrucción del *listing* 3.8.

```
cnn.mult ai,aj,ki,kj
```

Listing 3.8: Instrucción de prueba

Identificar la instrucción

La instrucción *cnn.mult* esta contenida en la tabla 3.2 y en la tabla 3.1 se ven el valor para la instrucción.

- cnn.mult = 00001

- ai = 00001 (dirección del registro ai)
- aj = 00010 (dirección del registro aj)
- ki = 00100 (dirección del registro ki)
- kj = 01000 (dirección del registro kj)
- opcode = 1010011

Concatenación de las instrucciones

Las instrucciones se concatenan según la tabla 3.2 quedando para el ejemplo como en el *listing* 3.9

```
00001000010001000100010001010011
```

Listing 3.9: Instrucción de ejemplo en binario

Transformación a hexadecimal

Finalmente el ultimo paso es pasar el código binario a hexadecimal quedando para este ejemplo como en el *listing* 3.10

```
08444453
```

Listing 3.10: Instrucción de ejemplo en hexadecimal

Ejemplo de código

En el *listing* 3.11 se muestra un ejemplo de código de convolución

```
# Convolucion de la casilla 0,0
00000053 # cnn.reset
080080D3 # cnn.mult 0,0,8,1
08028153 # cnn.mult 0,1,8,2
084090D3 # cnn.mult 1,0,9,1
08429153 # cnn.mult 1,0,9,2
3002A053 # cnn.show 1,10,0

# Convolucion de la casilla 0,1
00000053 # cnn.reset
08008053 # cnn.mult 0,0,8,0
080280D3 # cnn.mult 0,1,8,1
08048153 # cnn.mult 0,2,8,2
08409053 # cnn.mult 1,0,9,0
084290D3 # cnn.mult 1,1,9,1
08449153 # cnn.mult 1,2,9,2
3002A0D3 # cnn.show 1,10,1

# Convolucion de la casilla 1,0
```

```
20 00000053 # cnn.reset
21 080070D3 # cnn.mult 0,0,7,1
22 08027153 # cnn.mult 0,1,7,2
23 084080D3 # cnn.mult 1,0,8,1
24 08428153 # cnn.mult 1,1,8,2
25 088090D3 # cnn.mult 2,0,9,1
26 08829153 # cnn.mult 2,1,9,2
27 3002B053 # cnn.show 1,11,0
28
29 # Convolucion de la casilla 1,1
30 00000053 # cnn.reset
31 08007053 # cnn.mult 0,0,7,0
32 080270D3 # cnn.mult 0,1,7,1
33 08047153 # cnn.mult 0,2,7,2
34 08408053 # cnn.mult 1,0,8,0
35 084280D3 # cnn.mult 1,1,8,1
36 08448153 # cnn.mult 1,2,8,2
37 08809053 # cnn.mult 2,0,9,0
38 088290D3 # cnn.mult 2,1,9,1
39 08849153 # cnn.mult 2,2,9,2
40 3002B0D3 # cnn.show 1,11,1
```

Listing 3.11: Ejemplo de código completo

Capítulo 4

Resultados y Discusión

En este capítulo se mostrarán los resultados que se obtuvieron de las distintas pruebas realizas y los códigos necesarios para la correcta realización de estos experimentos, además de las distintas comparaciones de tiempos y las configuraciones con las cuales se midieron estos tiempos de respuesta.

4.1. Factores de configuración

Para la configuración del procesador realizado en esta memoria se simuló totalmente en el software ACTIVE-HDL de *Aldec*. Se configuró con una velocidad de reloj de 10Mhz.

Para la configuración del procesador con el que se comparó los tiempos de procesamiento, se ocupó una simulación de un *socket de RISC-V* con las mismas condiciones que el procesador anterior. La razón principal de ocupar un *socket* límitado, es poner en igualdad de condiciones el trabajo realizado, ya que, el trabajo tiene varias limitaciones, las que se pueden mejorar en un trabajo a futuro.

4.1.1. Consideraciones y simplificaciones para medir tiempo efectivo

Ya que ambos procesadores tienen la capacidad de realizar solamente una instrucción por ciclo de reloj y ambos tienen la misma configuración en cuanto a software y reloj, las comparaciones se harán por cantidad de instrucciones.

4.2. Resultados de las simulaciones

Las simulaciones se dividieron en 3 grandes grupos; cada uno asociado a una de las etapas que tienen las redes CNN.

4.2.1. Resultados Convolución

Para la convolución se aplicaron distintos kernels, todos cuadrados, simétricos e impares en sus dimensiones. Esta comparación se observa en la tabla 4.1, y de manera gráfica, en la figura 4.1, donde en azul, se muestra el tiempo de procesamiento de el *socket*, y en naranja, el tiempo de procesamiento de la propuesta solución.

Tamaño kernel	RISC-V	Propuesta
3x3	17	11
5x5	49	27
7x7	97	51
9x9	161	83

Tabla 4.1: Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para la convolución

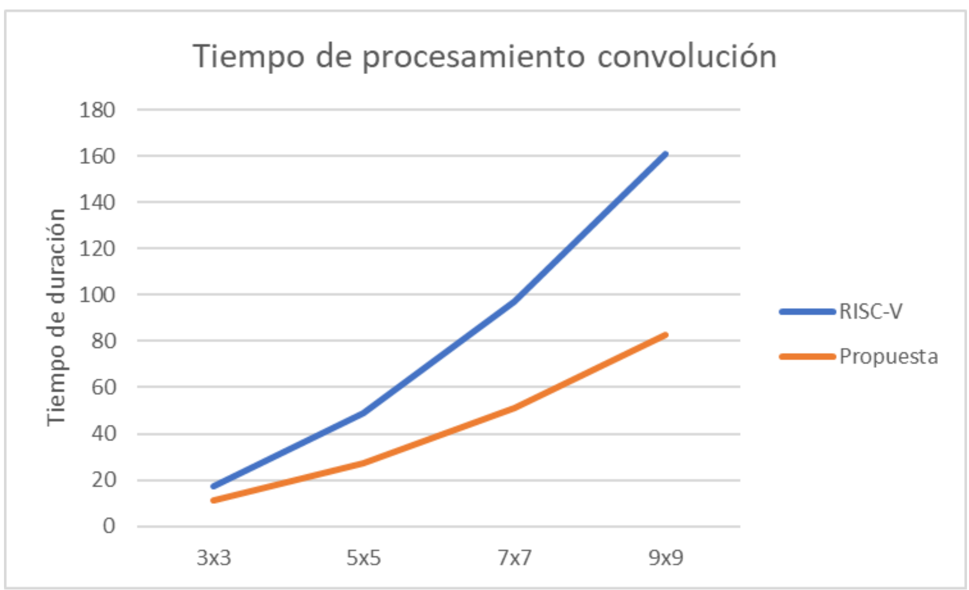

Figura 4.1: Gráfico comparativo de tiempos de ejecución entre RISC-V y la propuesta de solución para la convolucion

4.2.2. Resultados MaxPool

Para obtener los resultados de MaxPool se ocuparon matrices cuadradas y de distintas dimensiones. Esta comparación se observa en la tabla 4.2, y de manera gráfica, en la figura 4.2, donde en azul, se muestra el tiempo de procesamiento de el *socket*, y en naranja, el tiempo de procesamiento de la propuesta solución.

Tamaño matriz	RISC-V	Propuesta
2x2	3	3
3x3	8	5
4x4	15	8
5x5	24	13

Tabla 4.2: Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para MaxPool

Figura 4.2: Gráfico comparativo de tiempos de procesamiento entre RISC-V y la propuesta de solución para MaxPool

4.2.3. Resultados Multiplicación matricial

Para la multiplicación matricial se usaron matrices cuadradas multiplicadas por otra de la misma dimensión y no usaron optimizaciones para la multiplicación entre estas. Esta comparación se observa en la tabla 4.3, y de manera gráfica, en la figura 4.3, donde en azul, se muestra el tiempo de procesamiento de el *socket*, y en naranja, el tiempo de procesamiento de la propuesta solución.

Tamaño matriz	RISC-V	Propuesta
3x3	45	36
4x4	112	80
5x5	225	150
6x6	396	252
7x7	637	392

Tabla 4.3: Comparación de tiempos de procesamientos entre RISC-V y la propuesta de solución para multiplicación matricial

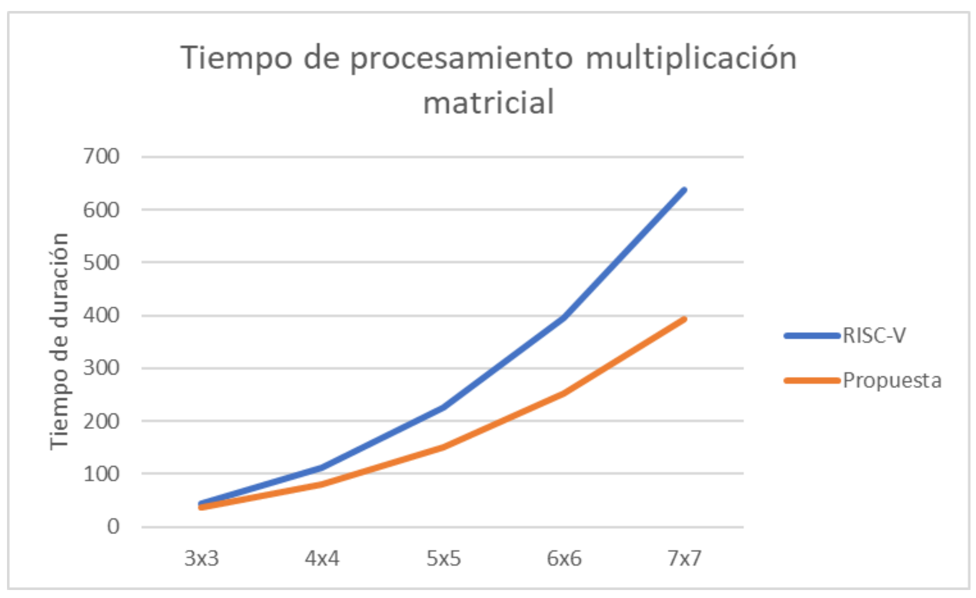

Figura 4.3: Gráfico comparativo de tiempos de procesamiento entre RISC-V y la propuesta de solución para la multiplicación matricial

4.2.4. Simulaciones ACTIVE HDL

Para la simulación en ACTIVE HDL se usó el código del *listing* 3.11. En las figuras 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10 y 4.11, el BUS710, muestra la operación que esta realizando la ALU, el BUS641 y 645, muestran las entradas a los registros y, mem, muestra la memoria que se esta utilizando.

En la figura 4.4 muestra en rojo la instrucción número 2 del código del *listing* 3.11 cnn.reset, donde se reinicia el registro interno.

Figura 4.4: Simulación en active HDL: en rojo se observa la instrucción 0, que es el reset del registro interno.

En la figura 4.5 muestra en rojo las instrucciones 3-6 del código del *listing* 3.11. Donde se aplica cnn.mult entre inputs y se guarda el resultado en el registro interno.

Figura 4.5: Simulación en active HDL: en rojo se observa la instrucción 1, que es multiplicipación entre input 1 y 2 más lo que contiene el registro.

En la figura 4.6 muestra en rojo la instrucción 7 del código del *listing* 3.11. Donde se aplica cnn.show y se guarda el valor en la dirección 10,0.

Figura 4.6: Simulación en active HDL: en rojo se observa la instrucción 6, que guarda el valor que contine el registro interno.

En la figura 4.7 se muestra como se actualiza el valor de la memoria con este cambio.

Figura 4.7: Simulación en active HDL: en rojo se observa que el registro guardo el valor esperado.

En la figura 4.8 muestra en rojo la instrucción número 10 del código del *listing* 3.11 cnn.reset, donde se reinicia el registro interno, para iniciar una nueva convolución de otra casilla

Figura 4.8: Simulación en active HDL: en rojo se observa la instrucción 0, que es el reset del registro interno.

Las figuras 4.9,4.10 y 4.11, muestran nuevos ciclos de la convolución de las instrucciónes del *listing* 3.11. Las instrucciónes son: de la 11 a la 20 para la figura 4.9, de la 21 a la 30 para la figura 4.10 y de la 31 a la 40 para la figura 4.11

Figura 4.9: Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 11-20

Figura 4.10: Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 21-30

Figura 4.11: Simulación en active HDL: en rojo se observa un nuevo ciclo instrucciones 31-40

4.3. Discusión

4.3.1. Rendimiento Convolución

A partir de los valores obtenidos en la tabla 4.1 es posible encontrar una generalización del tiempo obtenido según n (dimensión del *kernel*). Obteniendo las siguientes ecuaciones:

$$Conv_{RISC-V}(n) = 2n^2 - 1 \tag{4.1}$$

$$Conv_{propuesta}(n) = n^2 + 2 \tag{4.2}$$

Para obtener el tiempo de mejora estimado, se calcula el límite de n tendiendo a infinito del cuociente entre la ecuación 4.2 y la ecuación 4.1.

$$\frac{Conv_{propuesta}}{Conv_{RISC-V}} = \frac{n^2+2}{2n^2-1} \tag{4.3}$$

$$\lim_{n\to\infty} \frac{n^2+2}{2n^2-1} = \frac{1}{2} \tag{4.4}$$

Como se observa en la ecuación 4.4 el tiempo de ejecución de la propuesta para grandes números tiene una mejora considerable ya que le toma la mitad del tiempo versus su contraparte en RISC-V.

4.3.2. Rendimiento MaxPool

A partir de los valores obtenidos en la tabla 4.2 es posible encontrar una generalización del tiempo obtenido según n (tamaño de la disminución). Obteniendo las siguientes ecuaciones:

$$Pool_{RISC-V}(n) = n^2 - 1 \tag{4.5}$$

$$Pool_{propuesta}(n) = \frac{n^2}{2} \tag{4.6}$$

Para obtener el tiempo de mejora estimado, se calcula el límite de n tendiendo a infinito del cuociente entre la ecuación 4.6 y la ecuación 4.5.

$$\frac{Pool_{propuesta}}{Pool_{RISC-V}} = \frac{n^2/2}{n^2-1} \tag{4.7}$$

$$\lim_{n\to\infty} \frac{n^2/2}{n^2-1} = \frac{1}{2} \tag{4.8}$$

Como se muestra en la ecuación 4.8 el tiempo de ejecución de la propuesta es menor, en particular para grandes números, tendiendo a la mitad de tiempo que su contraparte en RISC-V.

4.3.3. Rendimiento Multiplicación Matricial

A partir de los valores obtenidos en la tabla 4.3 es posible encontrar una generalización del tiempo obtenido según n (dimensión de la matriz). Obteniendo las siguientes ecuaciones:

$$Mult_{RISC-V}(n) = 2n^3 - n^2 \tag{4.9}$$

$$Mult_{propuesta}(n) = n^3 + n^2 \tag{4.10}$$

Para obtener el tiempo de mejora estimado, se calcula el límite de n tendiendo a infinito del cuociente entre la ecuación 4.10 y la ecuación 4.9.

$$\frac{Mult_{propuesta}}{Mult_{RISC-V}} = \frac{n^3 + n^2}{2n^3 - n^2} \tag{4.11}$$

$$\lim_{n \to \infty} \frac{n^3 + n^2}{2n^3 - n^2} = \frac{1}{2} \tag{4.12}$$

Según la ecuación 4.12 el tiempo de ejecución de la propuesta, cuando se tiende a números grandes, es de la mitad que en su contraparte en RISC-V.

Capítulo 5

Conclusiones

El objetivo principal de esta memoria se cumplió totalmente, primeramente, porque se logran crear instrucciones personalizadas y muy específicas para un propósito muy bien definido que es la operación de una red CNN, implementando un microprocesador que cumpliera estas exigencias.

Las instrucciones personalizadas se diseñan al detalle, ya que el trabajo de esta memoria rompe el esquema tradicional de la arquitectura de un computador. Las diferencias más significativas entre el procesador tradicional de RISC-V y el propuesto está en que la ALU es de 3 entradas y el output no siempre otorga un valor a guardar.

El procesador propuesto, funciona para las 3 etapas importantes de la convolución, guardando los valores en la memoria del sistema cuando se necesitan y ocupando las instrucciones personalizadas propuestas en este trabajo.

La mejora en cuanto al tiempo de operación de cada una de las etapas es bastante prometedora, obteniendo mejoras de hasta un 50 %. Para la convolución, considerando que los *kernel* más comunes son de 3x3 y 5x5, la mejora es de 37 % y 45 %, respectivamente. Para la etapa de *pooling* la disminución más común es de a la mitad, por lo tanto, no hay una mejora considerable, sin embargo, si se quisiera reducir a un tercio la imagen, se obtendría una mejora de un 37 % y a mayor reducción se obtendrían resultados de hasta un 50 %. Finalmente, para la multiplicación matricial, las matrices que se trabajan son de gran volumen, considerando que normalmente sus dimensiones son mayores a 10x10, la mejora es prácticamente de un 50 %, siendo la etapa que tiene la más grande mejoría. Se obtiene un desempeño bastante considerable y logra el objetivo de reducir el tiempo de operación de una red CNN.

5.1. Trabajo futuro

Como trabajo futuro queda integrar este trabajo, a un procesador con arquitectura completa de RISC-V, en vez de la arquitectura simplificada y funcional que se presenta en este trabajo. Obteniendo una arquitectura completa se robustece el procesador quedando mejor preparado para inconvenientes y se actualiza la ISA, para no tener que convertir las instruc-

ciones de manera manual.

Además, como trabajos futuros queda adaptar este trabajo a un lenguaje de alto nivel. Con un lenguaje de alto nivel, las instrucciones personalizadas serán muy simples de usar, se podrá *debugear* de manera sencilla y la programación se hará de manera rápida y automatizada.

También, queda propuesto realizar una extensa documentación de las instrucciones nuevas y un manual de usuario con ejemplos de rutinas.

Finalmente, queda propuesto ampliar la arquitectura de este trabajo, para otros tipos de redes neuronales, agregar funciones de activación y trabajar las operaciones en punto flotante.

Bibliografía

[1] *ISA card refence.* RISC-V, https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files /docs/RISCVGreenCardv8-20151013.pdf.

[2] Gianluca D'Agostino. *Diseño e implementación de un SOC de RISC-V en una FPGA basado en el ISA de RISC-V.* Universidad de Chile, Chile, 2021.

[3] J Dayhoff. *Quimeras del Conocimiento. Mitos y Realidades de la Inteligencia Artificial.* Ediciones Fundesco, Barcelona, 1992.

[4] J Dayhoff. *Quimeras del Conocimiento. Mitos y Realidades de la Inteligencia Artificial.* Ediciones Fundesco, Barcelona, 1992.

[5] KUNIHIKO FUKUSHIMA. *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.* Springer, volumen 36, Japón, 1980.

[6] José Giralt. *Desarrollo de técnicas de procesamiento paralelo a nivel de lenguaje ensamblador para el procesador de RISC-V.* Universidad de Chile, Chile, 2021.

[7] Josip Knezovic Luka Strizic, Branimir Pervan. *Deep Learning Accelerator on Programmable Heterogeneous System with RISC-V Processor.* University of Zagreb, Zagreb, Croatia, 2019.

[8] Lei Zhang Fang Zhou Ning Wu, Tao Jiang and Fen Ge. *A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set.* Nanjing University of Aeronautics and Astronautics,, China, 2020.

[9] D. A. Patterson and J. L. Hennessy. *Arquitectura de computadores: Un enfoque cuantitativo.* MacGraw-Hill,, Madrid, 2002.

[10] Inés M. Galván León Pedro Isasi Viñuela. *Redes de Neuronas Artificiales. Un enfoque Práctico.* Pearson Prentice Hall, Madrid, 2003.

[11] Guoqing Lei Kai Chen Buyue Qin Xiaoqiang Zhao Zhiqiang Liu, Jingfei Jiang. *A Heterogeneous Processor Design for CNN-Based AI Applications on IoT Devices.* ELSEVIER, China, 2019.

Anexos

Anexo A

Códigos verilog del procesador

Se presentan los modulos ocupados para el correcto funcionamiento del procesador

```
/*
autor: Daniel Vasquez
Santiago de Chile, Noviembre 2021
*/
`timescale 1ns / 1ps

module cnn_aluoperation(
  input [31:0] in1,
    input [31:0] in2,
  input [9:0] int10,
  input [31:0] out_reg,
  input [2:0] sub_option,
  output reg enable_save,
  output reg [31:0] in_reg,
  output reg [31:0] cnn_out,
  output reg enable,
  output reg reset_reg);
  always_comb begin
        case(sub_option)
            3'b000: // operation=000: cnn.reset
      begin
          in_reg = 0 ;
        cnn_out = 0 ;
        enable=  1 ;
        reset_reg = 1;
        enable_save=0;
      end
            3'b001: // operation=001: cnn.mult
      begin
                in_reg = in1*in2+out_reg;
        cnn_out = in1*in2+out_reg;
        enable=  1;
        reset_reg = 0;
        enable_save=0;
      end
            3'b010: // operation=010: cnn.sum
      begin
```

```
38                in_reg = in1+in2+out_reg;
39        cnn_out = in1+in2+out_reg;
40        enable=  1;
41        reset_reg = 0;
42        enable_save=0;
43      end
44            3'b011: // operation=011: cnn.max
45      begin
46                in_reg = (signed'(in1) >=signed'(in2)) ? in1 : (signed'(
    in2)>=signed'(out_reg)? in2:out_reg)  ;
47        cnn_out = (signed'(in1) >=signed'(in2)) ? in1 : (signed'(in2)>=
    signed'(out_reg)? in2:out_reg) ;
48        enable=  1;
49        reset_reg = 0;
50        enable_save=0;
51      end
52            3'b100: // operation=100: cnn.min
53      begin
54                in_reg = (signed'(in1) >=signed'(in2)) ? in1 : (signed'(
    in2)>=signed'(out_reg)? in2:out_reg)   ;
55        cnn_out = (signed'(in1) >=signed'(in2)) ? in1 : (signed'(in2)>=
    signed'(out_reg)? in2:out_reg)  ;
56        enable=  1;
57        reset_reg = 0;
58        enable_save=0;
59      end
60            3'b101: // operation=101: cnn.prom
61      begin
62          in_reg = out_reg    ;
63        cnn_out = out_reg/int10 ;
64        enable=  1;
65        reset_reg = 0;
66        enable_save=1;
67      end
68      3'b110: // operation=110: cnn.show
69      begin
70          in_reg = out_reg*int10     ;
71        cnn_out = out_reg*int10 ;
72        enable=  1;
73        reset_reg = 0;
74        enable_save=1;
75      end
76      3'b111: // operation=111: cnn.div
77      begin
78                in_reg = out_reg     ;
79        cnn_out = in1/out_reg  ;
80        enable=  1;
81        reset_reg = 0;
82        enable_save=1;
83      end
84            default:
85      begin
86                in_reg = out_reg     ;
87        cnn_out = out_reg    ;
88        enable=  0;
89        reset_reg = 0;
```

```
90         enable_save=0;
91       end
92         endcase
93     end
94 endmodule
95
96 module cnn_register(
97   input [31:0] in_reg,
98   input enable,
99   input clk,
100   input reset_reg,
101   output reg [31:0] out_reg);
102
103   always @(negedge clk)
104     begin
105       if (reset_reg==1)
106         out_reg=0;
107       else
108         if (enable==1)
109           out_reg=in_reg;
110     end
111 endmodule
112
113
114 module counter (
115   input clk,
116   input enable,
117   output reg [31:0]data_in,);
118
119   reg [7:0]count;
120   reg [31:0]mem[108];
121   initial
122     begin
123      $readmemh ("data_fin.mem",mem);
124      end
125
126   always @(posedge clk)
127     begin
128       if(enable==1)
129           count<=count+1;
130       else
131         count<=0;
132     end
133   assign data_in=mem[count];
134
135
136 endmodule
137
138 module cnn_decoder (input [31:0]data_in,
139   output reg [4:0]rs1i,
140   output reg [4:0]rs1j ,
141   output reg [4:0]rs2i ,
142   output reg [4:0]rs2j ,
143   output reg [2:0]sub_option,
144   output reg [4:0]rdi ,
145   output reg [4:0]rdj ,
```

```
146   output reg [9:0]int10 );
147 
148   assign sub_option= data_in[29:27];
149   assign rs1i=data_in[26:22];
150   assign rs1j=data_in[21:17];
151   assign rs2i=data_in[16:12];
152   assign rs2j=data_in[11:7];
153   assign rdi=data_in[16:12];
154   assign rdj=data_in[11:7];
155   assign int10=data_in[26:17];
156 
157 
158 endmodule
159 
160 module cnn_read_mem();
161   reg [31:0]mem[0:18][0:6];
162   initial
163     begin
164      $readmemh ("mem_mem.mem",mem);
165      end
166 
167 endmodule
168 
169 module cnn_memory (
170   input [4:0]rs1i,
171   input [4:0]rs1j,
172   input [4:0]rs2i,
173   input [4:0]rs2j,
174     output reg [31:0]in1,
175   output reg [31:0]in2);
176   reg [31:0]mem[0:18][0:6];
177     initial
178     begin
179      $readmemh ("mem_mem.mem",mem);
180      end
181     assign in1 = (mem[rs1i][rs1j]);
182   assign in2 = (mem[rs2i][rs2j]);
183 endmodule
184 
185 module cnn_save_mem(
186   input [4:0]rdi,
187   input [4:0]rdj,
188   input clk,
189   input enable,
190   input [31:0] out_cnn);
191   reg [31:0]mem[0:18][0:6];
192   wire [4:0] rdi,rdj;
193   initial
194     begin
195      $readmemh ("mem_mem.mem",mem);
196      end
197   always @(posedge clk) begin
198   if (enable==1)
199     mem[rdi][rdj]=out_cnn;
200   $writememh ("mem_mem.mem",mem);
201   end
```

```
endmodule
```

A continuación se muestra el código que muestra las conexiones entre los módulos

```
// ---------- Design Unit Header ---------- //
`timescale 1ps / 1ps

module total2 (clk,Enable) ;

// ------------ Port declarations --------- //
input clk;
wire clk;
input Enable;
wire Enable;

// ----------- Signal declarations -------- //
reg [31:0]mem[0:18][0:6];
wire NET664;
wire NET744;
wire NET760;
wire [31:0] BUS1032;
wire [4:0] BUS622;
wire [4:0] BUS626;
wire [4:0] BUS630;
wire [4:0] BUS634;
wire [31:0] BUS641;
wire [31:0] BUS645;
wire [9:0] BUS649;
wire [31:0] BUS679;
wire [31:0] BUS687;
wire [2:0] BUS710;
wire [4:0] BUS714;
wire [4:0] BUS722;
wire [31:0] BUS752;

// -------- Component instantiations -------//

cnn_decoder U1
(
  .data_in(BUS1032),
  .rs1i(BUS622),
  .rs1j(BUS626),
  .rs2i(BUS630),
  .rs2j(BUS634),
  .sub_option(BUS710),
  .rdi(BUS714),
  .rdj(BUS722),
  .int10(BUS649)
);



cnn_memory U2
(
  .rs1i(BUS622),
  .rs1j(BUS626),
  .rs2i(BUS630),
```

```
  .rs2j(BUS634),
  .in1(BUS641),
  .in2(BUS645)
);



counter U3
(
  .clk(clk),
  .enable(Enable),
  .data_in(BUS1032)
);



cnn_register U4
(
  .in_reg(BUS679),
  .enable(NET664),
  .clk(clk),
  .reset_reg(NET744),
  .out_reg(BUS687)
);



cnn_save_mem U5
(
  .rdi(BUS714),
  .rdj(BUS722),
  .clk(clk),
  .enable(NET760),
  .out_cnn(BUS752)
);



cnn_aluoperation U6
(
  .in1(BUS641),
  .in2(BUS645),
  .int10(BUS649),
  .out_reg(BUS687),
  .sub_option(BUS710),
  .enable_save(NET760),
  .in_reg(BUS679),
  .cnn_out(BUS752),
  .enable(NET664),
  .reset_reg(NET744)
);





```

```
110  always @(posedge clk)
111     begin
112      $readmemh ("mem_mem.mem",mem);
113      end
114
115
116 endmodule
```

Anexo B

RISC-V ISA

A continuación se encuentra el set de instrucciones (ISA) de un procesador de RISC-V que se encuentra en [1]

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd′,rs1′,imm`	`LW rd′,rs1′,imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd′,rs1′,imm`	`LD rd′,rs1′,imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd′,rs1′,imm`	`LQ rd′,rs1′,imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1′,rs2′,imm`	`SW rs1′,rs2′,imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1′,rs2′,imm`	`SD rs1′,rs2′,imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1′,rs2′,imm`	`SQ rs1′,rs2′,imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1′,imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1′,imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats

	31–25	24–20	19–15	14–12	11–7	6–0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1] imm[11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20] imm[10:1]	imm[11]	imm[19:12]		rd	opcode

16-bit (RVC) Instruction Formats

	15–13	12	11–10	9–7	6–5	4–2	1–0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm				rd′	op
CL	funct3	imm		rs1′	imm	rd′	op
CS	funct3	imm		rs1′	imm	rs2′	op
CB	funct3	offset		rs1′	offset		op
CJ	funct3	jump target					op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*