國立臺灣師範大學理學院

資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Science

National Taiwan Normal University

Master's Thesis

以 Chipyard 為基礎的 SoC 設計平台 FPGA 實現之研究

Research on FPGA Implementation of Chipyard-based SoC Design Platform

黃維熙

Wei-Hsi Huang

指導教授：黃文吉 博士

Advisor: Wen-Jyi Hwang, Ph.D.

中華民國111年7月

July 2022

致謝

本論文得以完成，首先要感謝我的指導教授黃文吉博士不厭其煩的指導，除了給予我論文題目的方向，更在研究與實驗過程中提供莫大的協助，讓我能快速的掌握研究的核心方向。教授在治學充滿熱忱與嚴謹，不只對於我的研究給予實質的幫忙，更在論文的撰寫方面，從研究貢獻、核心研究方法、最後的實驗數據與結論，對於每個細節花費相當多的時間來批閱，並且給予建議來做修正，使本論文更加完整。

在求學過程中，感謝我實驗室的同學們，非常感謝鄭博升以及蔡佳諭同學在我迷茫時協助我很快地找會狀態，並且在 Debug 時也提供了非常多的幫助。也非常感謝古佳像同學在我研究遇上問題時，提供了一些解決問題的手段，最後感謝曾靖庭同學在最後論文撰寫階段的陪伴，沒有上述的同學，相信本論文的撰寫不會如現在那麼順利，最後感謝我的家人，因為有你們的關懷與支持，讓我能全新全意的完成我的學業，由衷的感謝你們。

最後僅將這篇論文獻給最關心我的家人及朋友，願你們一起與我分享這份喜悅與榮耀。

摘要

近年來在軟體上的 AI 加速器發展越來越多元化，並且在硬體上也有一些的發展及實現，而硬體 AI 加速器的優勢在於對特定資料格式做運算可以大幅提升速度，僅需使用資料流的方式就可以實現。

本論文針對柏克萊大學提出的硬體開源框架 Chipyard，提出一個硬體建構的流程，將 RISC-V 為基礎的 CPU 搭配 AI 硬體加速器整合於 FPGA 平台，並且完善 RISC-V 軟體開機流程，讓我們可以通過硬體建構流程調整所需的硬體資源，做出客製化的硬體電路，快速的去對 CPU 及 AI 硬體加速器於 FPGA 開發板上做有效的效能評估。

關鍵字：FPGA、SoC、Chipyard

目錄

表目錄

圖目錄

第1章　緒論

本章節主要內容是對本論文做簡單的介紹，1-1 節會針對開源系統晶片發展平台做介紹。1-2 節會提出本論文的動機。1-3 節則是說明本論文的研究時所遇到的困難。1-4 節則是說明本論文主要的研究目的。1-5 節會提出本論文在建構以 Chipyard 為基礎的 SoC 設計平台在 FPGA 上實現所做的貢獻。

1-1 研究背景

1-1-1 Chipyard

Chipyard[1]是柏克萊大學(UC Berkeley)所推出的一個開源(Open-Source)的系統晶片開發平台(SoC development platform)，最主要使用的是基於 RISC-V 指令集[2]的中央處理器(Central Processing Unit, CPU)，Chipyard 提供了很多部件讓我們來圍繞著這個 CPU 建構一個屬於自己的 SoC(System on Chip)，也提供了許多環境供我們做測試及開發。在 Chipyard 的部件其中基於 RISC-V 指令集的 CPU[3]有:Rocket core、BOOM(Berkeley Out-of-Order Machine)[4]、CVA6 Core[5]以及 32 bit 的 Ibex core[6]。在硬體加速器方面，Chipyard 支援: Hwacha[7]、Gemmini[8]、SHA3[9]及 NVDLA[10]等用途各不相同的硬體加速器。

系統部件則是提供有: Icenet[11]、Sifive-blocks、AWL (Analog Widget Library)和 testchipip 等多種 IP 讓我們可以在測試或實際需要時去呼叫來進行使

用，而這些 IP 都是開源的，不需要額外付費即可使用。

開發工具與工具鏈則是提供:Chisel(Constructing Hardware in a Scala Embedded Language) [12]、FIRRTL(A Flexible Intermediate Representation for RTL) [13]、Bartools、Dsptools、Dromajo、riscv-tools、esp-tools 等的一些利於我們上手的工具，包括可讀性高的硬體描述語言，還有工具通過 IR(intermediate representation)去轉成通用的硬體描述語言，還有一些交叉編譯的工具鏈，讓我們的開發更加輕鬆。

模擬器的部分，在軟體分面提供了 Spike[14] (RISC-V ISA Simulator)，RTL(Register-Transfer Level)方面提供了 Verilator[15](Open-Source)以及 VCS[16](license needed)，以及柏克萊大學提供的雲端 FPGA 模擬器:FireSim[17]，通過各種模擬器，可以更快速的驗證模擬出的硬體的正確性。

Chipyard [1]也支援兩種 VLSI(Very-Large-Scale integration) 設計方式: FPGA(Field Programmable Gate Array) Prototyping 以及 ASIC(Application Specific Integrated Circuit)，讓我們可以通過這兩種方法，去客製化我們自己的系統晶片。

1-1-2 Existing SoC Design Platforms

表 1-1 本論文比較了現有的 SoC 設計發展平台，包括提供者、提供可使用處理器 IP、Design Style、開源情況、開機程序以及產出結果，可以看到這邊是將 Chipyard[1]與 Quartus、Vivado 做比較，最主要的差異是 Chipyard 裡面的 IP 都是開源可以供我們免費使用及調整的。而 Quartus 及 Vivado 則是沒有開源的情況，所以我們無法針對特定需求去客製化一顆 CPU，來使資源有最少的浪費，效能可以最大化的被利用。

那在這邊還有一個比較特別的點是，Chipyard 產出的是合適於 RTL 設計的 Verilog 硬體描述語言(Hardware Desciption Language, HDL)程式碼，而另外兩個則是做完後可以直接產出一個完整的 FPGA 電路，供我們的 FPGA 板做使用，所以就意味著 Chipyard 產出的硬體描述語言需要和這些開發平台做結合，也意味著我們可以在這幾個平台上各取所需，取出各自的優點，並且加以結合，生成我們需要的完整 FPGA 電路。

表 1- 1 比較現有 SoC 設計平台

	Chipyard	Quartus Qsys	Vivado IP-Integrator
提供者	UC Berkeley	Altera/Intel	Xilinx
中央處理器	RISC-V only	NIOS II, NIOS V(RISC-V) ARM(Hardcore)	MicroBlaze, ARM(Hardcore)
Design Style	FPGA,ASIC	FPGA	FPGA
開源情況	Yes	No	No
Boot Procedure	First-Stage boot loader	Proprietary (Dependent on Development board)	Proprietary (Dependent on Development board)
產出	Verilog code for RTL Design	完整FPGA 電路	完整FPGA 電路

1-2 研究動機

由於 Chipyard[1]是開源讓我們使用的，所以一定會有相對應不夠完整的地方，下面本論文會將 Chipyard 的限制分為軟硬體去做說明。

軟體方面: Chipyard 在 RISC-V 軟體層面 Boot 的流程，只支援到 First-Stage 的部分，並沒有完整 Linux 支援，在我們做完 FPGA 電路，並且燒路上開發板後，沒有 OS(Operating System)會讓我們後續的驗證程序受阻。

硬體方面: Chipyard 在建構 SoC 的硬體流程上，僅有提供到 RTL level 的設計，並沒有完成 Physical level 的電路，讓我們可以直接應用到 FPGA 上面做使用。

1-3 研究困難

在 Chipyard[1]平台下，主要有兩個研究困難，其一是 Chipyard 平台在硬體設計裡面只有做到 CPU 的 RTL 設計，沒辦法有效的整合記憶體，在對 FPGA 開發板做開發時，這樣會有一些介面及資料傳遞上的問題需要去解決，若是沒有很好的解決辦法，就會導致完整的 SoC 建置以及評估會較為困難。

其二是 Chipyard 在軟體開發層面，只有提供 First-Stage Bootloader，沒有後續的 Bootloader 以及 OS 的導入，對於我們 SoC 的效能評估會有很大的影響，其中，相關的 AI(Artificial Intelligence)應用程式無法執行，是對整個流程最大的阻礙。

1-4 研究目的

本論文主要目的是以 Chipyard[1]為基礎，提出軟硬體建構的流程，並且通過使用 Chipyard 平台提供的一些硬體元件、中央處理器、加速器及快取記憶體等的部件，協助我們快速的建構出一個完整的 SoC 系統，在硬體方面，提供從頂層元件設計，到底層 FPGA 實際電路的完整流程。在軟體方面，提供從底層 First-Stage Bootloader 到頂層 AI 應用程式開發、驗證以及效能評估的完整流程，通過軟體方面的 OS，也可以讓我們在 FPGA 開發板上建立環境，開發我們的 AI 應用程式。

1-5 研究貢獻

由於現有的系統並沒有提供非常完善的軟硬體流程讓我們去使用及燒錄到 FPGA 開發板上面，只能在模擬層面或往 ASIC 方向滿足我們，所以基於 Chipyard[1]平台所提供的硬體元件及工具鏈等的部件，通過 Chipyard 的便利性結合其他 SoC 發展平台的優點，本論文在硬體方面提供 Chipyard SoC 發展平台完整硬體設計流程，自 RTL 元件產生、RTL 元件整合到完整 FPGA 電路設計，簡化 SoC 的 Fast Prototyping 驗證以及評估。

本論文在軟體方面，提供 Chipyard SoC 平台完整軟體設計流程，自 First-Stage Bootloader 至 Linux 作業系統佈署，提供完整 AI 應用程式在 SoC 的發展平台。

第 2 章 理論基礎與背景

本章節的主要內容是為了討論與本論文相關的軟硬體開發流程以及技術，其中包含 Chipyard 平台的設計流程、Chisel、FIRRTL、硬體生成器、Rocket core、Gemmini Accelerator。

2-1 Chipyard Design Flow

圖 2-1 表示的是 Chipyard[1]平台現有的流程，從最一開始的選擇我們所需 Chipyard 開源的硬體核心(CPU)，加上使用 Chisel[12]來表達的設計、外部的接口、外部設備、快取記憶體又或者 AI 硬體加速器等，組合好我們選擇或所需的硬體配置後，通過 FIRRTL[13]這種硬體編譯器(Hardware Compiler)，將 Chisel 換為 Verilog ，給各種用途使用。例如在本機端通過 Verilator[15]、VCS[16]這種 RTL 模擬工具進行模擬，或者在柏克萊大學建立的雲端系統上面跑 FPGA 模擬，也能通過 Chipyard 平台提供的工具，再經過很多複雜、繁瑣的 EDA (Electronic Design Automation)設計流程，建立一顆晶片出來。

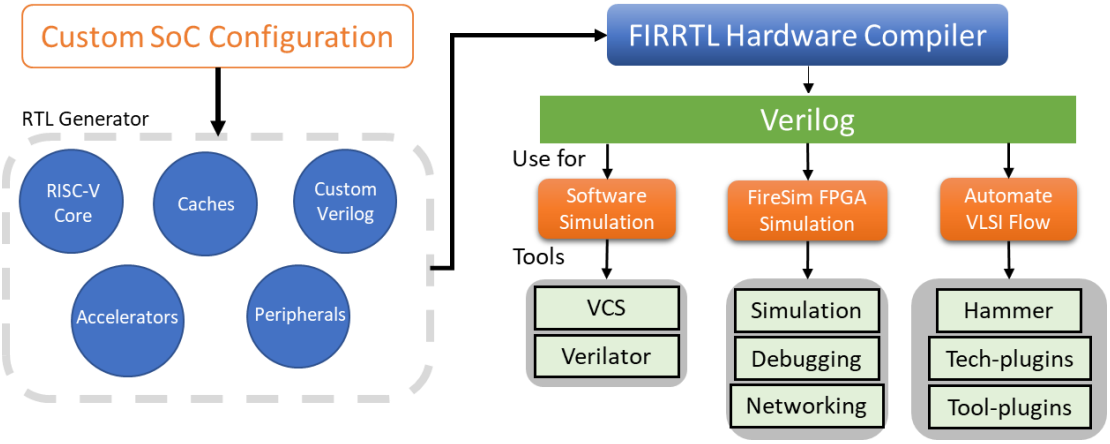


圖 2- 1 Chipyard 平台設計流程

圖 2-2 則是本論文預計通過這一種方式來進行 FPGA 硬體電路的構建，通過一系列流程將要 Chisel 組合而成的電路通過 FIRRTL 轉換為 Verilog，之後將電路燒上 FPGA 開發板後，搭配軟體在開發板上做驗證與效能評估。

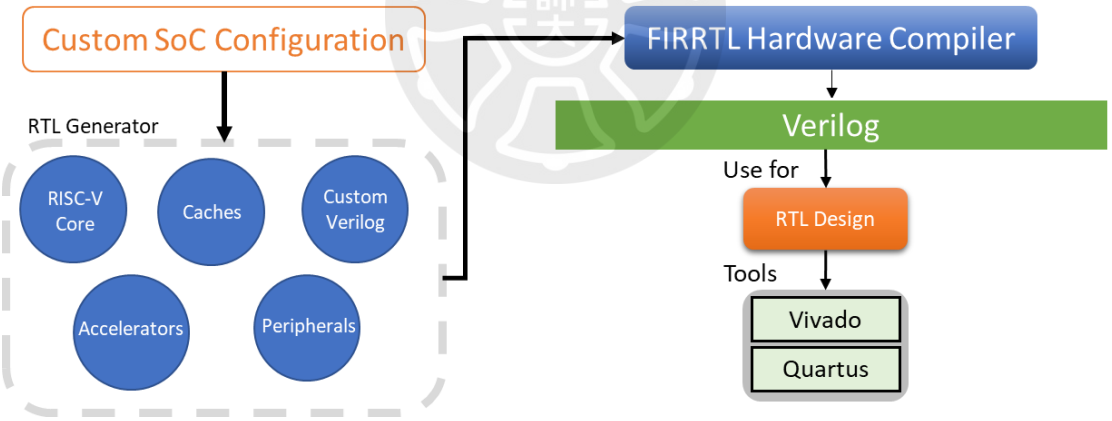


圖 2- 2 本論文使用之設計流程

2-2 Chisel

Chisel[12]是柏克萊大學開發的一種開源的、嵌入在 Scala 中的高階硬體描述語言，Chisel 最主要用於描述 RTL 中的數位電路以及一些高度參數化的電路生成器，例如快取記憶體、處理器或具有特定功能的 Block 等，有助於 ASIC 和 FPGA 數位邏輯設計的電路生成和設計的重複使用。

Chisel 在 Scala 語言中加上構建硬體的語法，為硬體電路設計人員提供了類似於現在主流的軟體程式語言，例如:Python, C 等語言的強大功能，通過硬體編譯器，從而生成可合成的 Verilog 電路。這種方法可以幫助我們去建構可重複使用的組件和函式庫，例如 Chisel Standard Library 中的 FIFO(First In First Out)，我們通過 include 這個函式庫，就可以直接對其進行呼叫並且通過設定的參數使用，讓設計更加直觀且容易。雖然 Chisel 與 RISC-V 沒有必然聯繫，但柏克萊大學是利用 Chisel 設計出基於 RISC-V 指令集的 CPU[3]，所以大部分的 RISC-V 的 CPU 都是採用 Chisel 實作而成，且皆具備參數化調整硬體資源的功能。

簡單來說，Chisel 可以理解為一種 Verilog 生成器(需要搭配硬體編譯器 FIRRTL)，通過利用 Scala 的語法，來快速高效的開發硬體設計。

2-3 FIRRTL

FIRRTL[13]是一種基於 Chisel[12]而產生出的硬體編譯器，最主要的功能就是將 Chisel 編譯為 Verilog。2-2 章有提到，Chisel 是高階的硬體描述語言，主要解決了傳統硬體描述語言不易將硬體電路參數化的問題，但如果我們要將 Chisel 電路功能實作出來，很多現有的硬體開發工具並不支援，因為這些現有的工具只支援到 Verilog 及 VHDL 這種傳統的硬體描述語言，所以通過 FIRRTL 將 Chisel 轉為 Verilog，就可以有效落實電路設計及模擬，並解決 Chisel 在實作時支援工具不足的問題。

FIRRTL 在做硬體編譯器的工作時，會通過演算法、資源共享、重複使用特定模組及減少不必要暫存器等方式，對電路進行優化，也會在中間確認 Chisel 的語法是否有誤。FIRRTL 也支援在硬體編譯器裡面加入特定的規則或條件，讓 FIRRTL 能夠更好的符合我們的需要的電路轉換模式。

圖 2-3 可以看到 Chisel 轉變為 Verilog 的過程，簡單來說，Chisel 通過 FIRRTL 硬體編譯器後，可以產生出我們需要的 Verilog ，並且在通過 FIRRTL 硬體編譯器時會幫助我們檢查語法以及優化程式碼。

圖 2-3 FIRRTL 轉換流程

2-4 Rocket Chip Generator

Rocket Chip 生成器[3]是柏克萊大學開發用於生成 SoC 的電路生成器，Chipyard[1]使用 Rocket Chip 生成器作為產生 RISC-V SoC 的基礎。

Rocket Chip 生成器裡面包含很多 SoC 使用的組件，Rocket Chip 一般使用 Rocket Core 作為其預設 CPU，但也可以通過參數設定的方式，去使用 BOOM[4]亂序核心或其他一些自定義的 CPU，接下來會對 Rocket Chip 生成器(圖 2-4)的每個部件逐一做說明:

A. Core Generator，用於生成核心，支援 Rocket-core 和 BOOM 兩種，又或是自行編寫的核心，Chipyard 提供的核心皆可以額外加入 FPU。

B. Cache，可自定義大小和類型的 L1 Cache 和 L2 Cache。

C. RoCC，The Rocket Custom Coprocessor interface，可以自行透過 Chisel 編寫加速器，或使用柏克萊大學提供的加速器，掛載到 Rocket-chip 中。

D. Tile: 核心和加速器和 L1 Cache（包括 Instruction Cache 和 Data Cache）構成一個 Tile，而核心和加速器的數量和類型可以通過參數自行修改。

E. TileLink[18]: 為柏克萊大學自行開發的 Bus，主要有，用於連接和溝通處理器、Cache 等 Rocket Chip 裡面有的組件。

F. Peripheral: Rocket Chip 對外部溝通的主要 Bus，使用 AMBA[19] (Advanced Microcontroller Bus Architecture）架構，例如: AXI，AHB-Lite 和 APB 等。

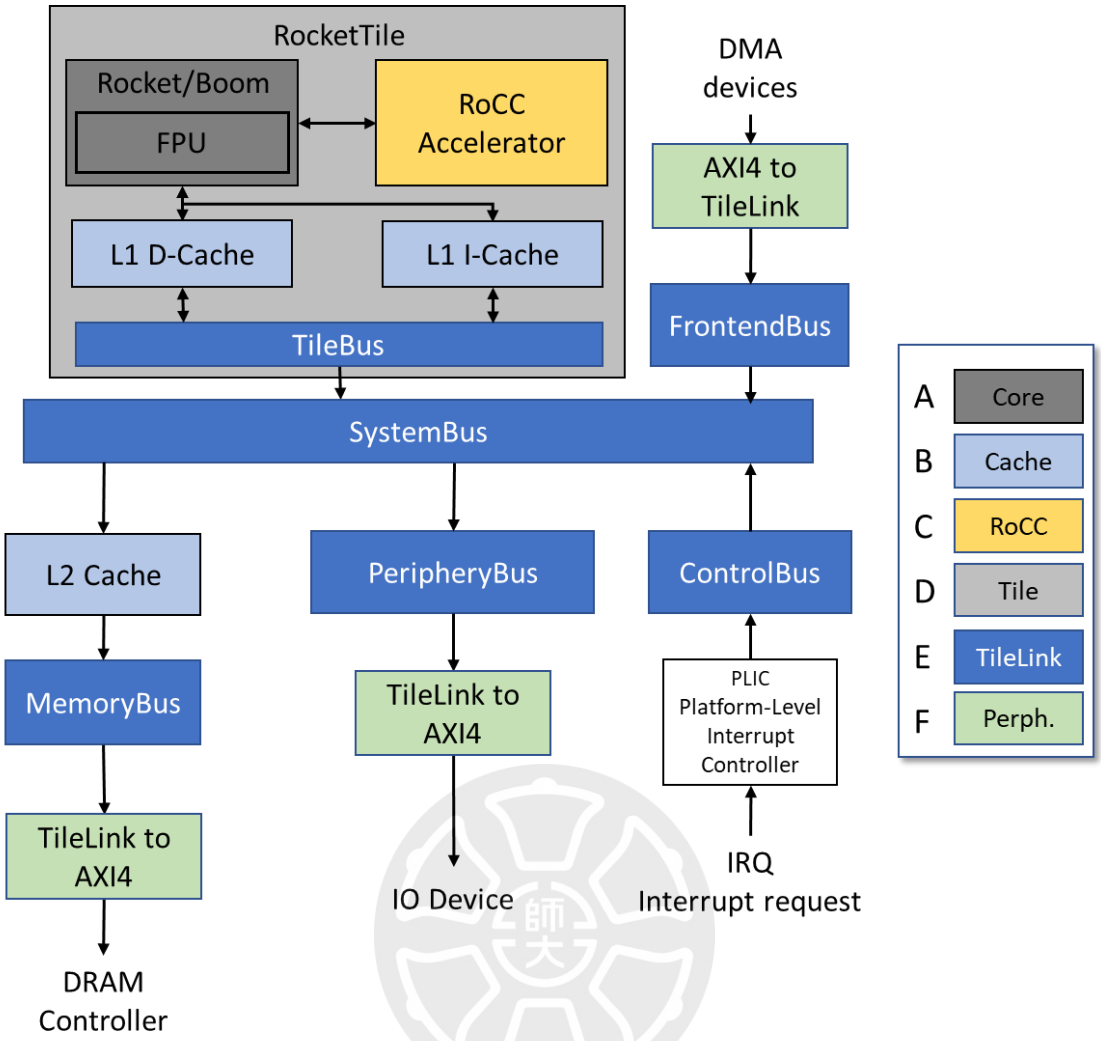


圖 2- 4 Rocket Chip 生成器，其中 bus 的部分有 SystemBus、TileBus、FrontendBus、ControlBus、PeripheryBus 及 MemoryBus，這邊較為特殊的是 PLIC[20](Platform-Level Interrupt Controller)，這是一個 I/O 設備使用的硬體的 interrupt unit，專門處理 I/O 設備的 IRQ(Interrupt ReQuest)。

2-5 Rocket Core

Rocket Core 是一個標準的 5-stage pipeline 處理器(圖 2-5)，用作 Rocket Chip SoC 生成器[3]中核心部分的一個組件。

Rocket Core 支持開源的 RV64GC RISC-V 指令集[2]，採用 Chisel[12]硬體描述語言編寫。Rocket Core 具有支援 page-based 的虛擬內存的記憶體管理單元(Memory Management Unit, MMU)、Non-blocking 的資料快取記憶體(Data Cache)和帶有分支預測(Branch Prediction)的前端。分支預測這個功能是通過三個組件分別是分支目標緩衝區 (Branch Target Buffer, BTB)、分支歷史表(Branch History Table, BHT) 和 Return Address Stack (RAS)去實現的，也可以通過參數對其進行調整。而對於浮點數運算，Rocket Core 中使用了柏克萊大學開發的 Chisel 浮點單元(Floating Point Unit, FPU)實現。

Rocket Core 還支持 RISC-V 的 Machine、Supervisor 和 User 權限等級，可以透過權限等級啟動 Linux 操作系統。Rocket Core 也公開了許多參數，包括對特定的指令集架構擴展的支援、浮點 pipeline 階段的數量以及 Cache 大小都可以做調整。

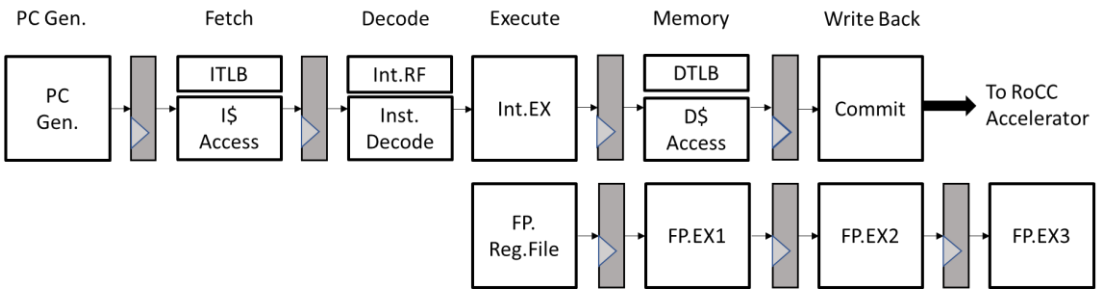


圖 2- 5 Rocket Core pipeline

2-6 Gemmini 加速器

Gemmini[8]加速器的核心是一個執行矩陣乘法的 Systolic Array 所構成的電路，也是 Chipyard [1]生態系統的一部分，Gemmini 是通過 Chisel[12]硬體描述語言進行開發的，通過此電路我們可以實現以資料流的方式去對矩陣乘法、卷積運算等需要大量 CPU 資源的運算做加速，目前 Gemmini 支援的資料流方式有 Weight Stationary[21] (WS)及 Output Stationary[21] (OS)。

指令部分，Gemmini 為使用 RISC-V 自定義指令[2]的 RoCC (Rocket Custom Coprocessor Interface) 加速器，使用 Rocket Core[3]或 BOOM [4]核心的 RoCC 接口去對其下達指令或接收回傳地址。資料部分，通過 DMA(Direct Memory Access)控制器對記憶體中我們運算需要使用的資料做搬運，以及回傳運算結果。

圖 2-6 描述的是 Gemmini 的硬體架構，Gemmini 硬體裡面支援有: DMA Controller、Scratchpad、Activation Function Unit、Scaling Function Unit、Transposer 及 Systolic Array 等。

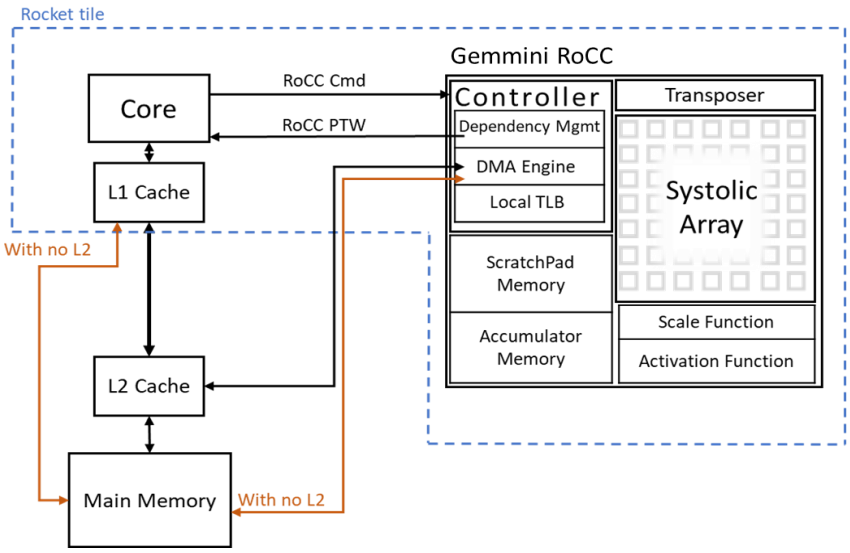


圖 2- 6 Gemmini 加速器硬體架構

第 3 章 研究方法

本章節主要會分為兩節來說明本論文的研究方法，第一節硬體設計流程是通過第二章所述的 SoC 設計流程(圖 2-2)以及 SoC 的周邊組件去合成出我們所需的硬體電路。第二節軟體建構流程是接續 Chipyard 沒提供的 Bootloader，去將作業系統燒錄至 SD 卡上，通過 SD 卡上的程序讓我們硬體燒錄的 FPGA 開發板可以正常開機使用。

3-1 Hardware Design Flow

這一節主要介紹本論文的硬體設計流程，在這一節中，本論文會詳細介紹 Chipyard 中本論文所使用的各個硬體元件，並且通過 Vivado SoC 設計平台中的 IP-Integrator 實際操作將 Rocket Chip 連接至外部記憶體以及 I/O，製作出一個可以燒至 FPGA 的 SoC，接下來本論文將通過圖 3-1 中描述的四個流程，詳細說明本論文的硬體設計流程。

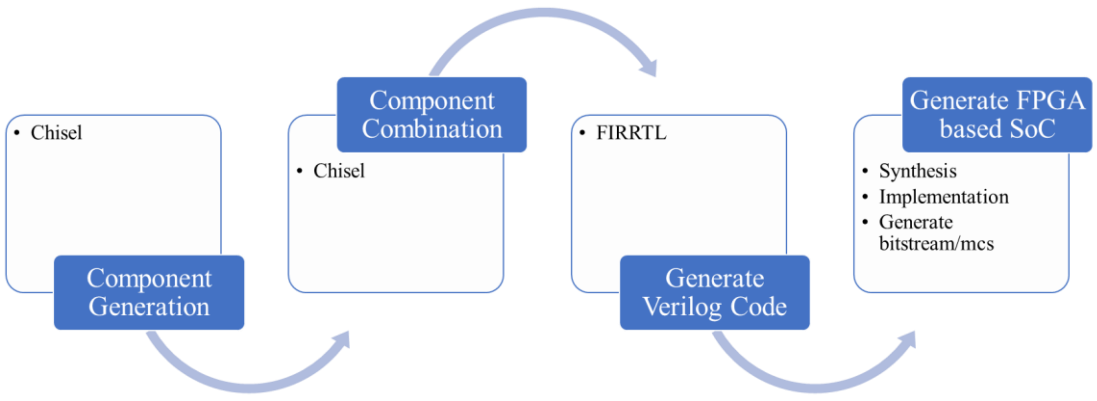


圖 3- 1 硬體設計流程

(一) 元件產生

在元件產生這一步會特別針對本論文主要使用的五種元件做詳細的討論，五種元件分別是 Rocket Core & L1 Cache、L2 Cache、Rocket Base Configuration、Gemmini 加速器以及 Rocket Chip chiptop，並且指出通過 Chisel，能夠讓我們在硬體電路上面透過參數調整達到我們預期使用的硬體資源。

1. L1 Cache & Rocket Core

在 Chipyard 中生成 Rocket Core 時，會有 L1 Cache 隨之而生，但事實上 L1 Cache 與 Rocket Core 屬於兩種不同的元件，並不是綁定在一起的。本論文首先介紹 L1 Cache，L1 Cache 分為 Data Cache 和 Instruction Cache 兩種，可調整參數以及對應到 Rocket Chip(圖 2-4)中的架構如圖 3-2。

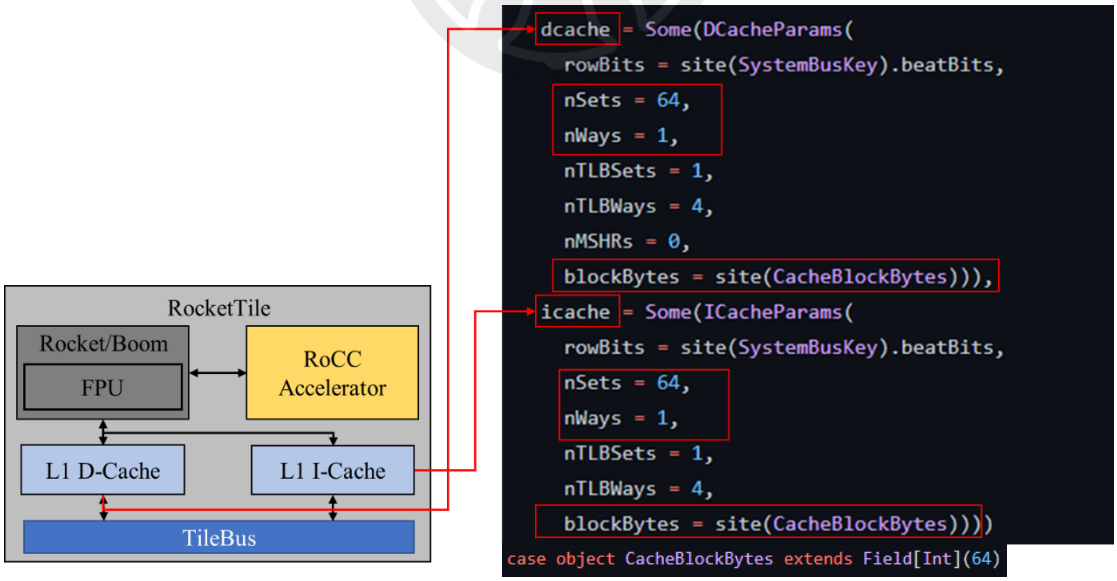


圖 3- 2 L1 Cache 參數，CacheBlockBytes 預設為 64 [22]，nWays=1 代表 Cache 存取方式為 Direct mapping，nSets=64 表示 Cache 有 64 個 block，Cache 通過公式 nSets*nWays *blockBytes 計算後，I-Cache[23]及 D-Cache[24]皆為 4KB(4096 Bytes)，由於 Rocket Core 支援虛擬記憶體，所以 Cache 裡面有 TLB 的參數，nTLBSets=1 表示這邊 TLB 有一個 Block，nTLBWays=4 代表這是 4-way set associative virtual memory。

Rocket Core 是一個 64 位元 RISC-V 指令集的核心，也是 Rocket Chip Generator 預設使用的核心，他有許多核心參數可以做調整，在 Chipyard 中柏克萊大學提供了四種不同功能的 Rocket Core，分別是 Big Core、Med Core、Small Core 和 Tiny Core，本論文以圖 3-3 Med Rocket Core[25]及圖 3-4 Tiny Rocket Core[25]為例子對 Rocket Core 主要可調整參數做介紹。

圖 3-3 中白色虛線為 Rocket Core 對應於 Rocket Chip 的架構，紅框中我們可以看到有 fpu(floating poing unit)、btb(branch target buffer)及 L1 Cache 的參數，有 fpu 表示 Rocket Core 支援浮點數運算，有 btb 代表此核心支援 branch prediction，且在調整 Rocket Core 參數時，也可以一起對 L1 Cache 做調整。

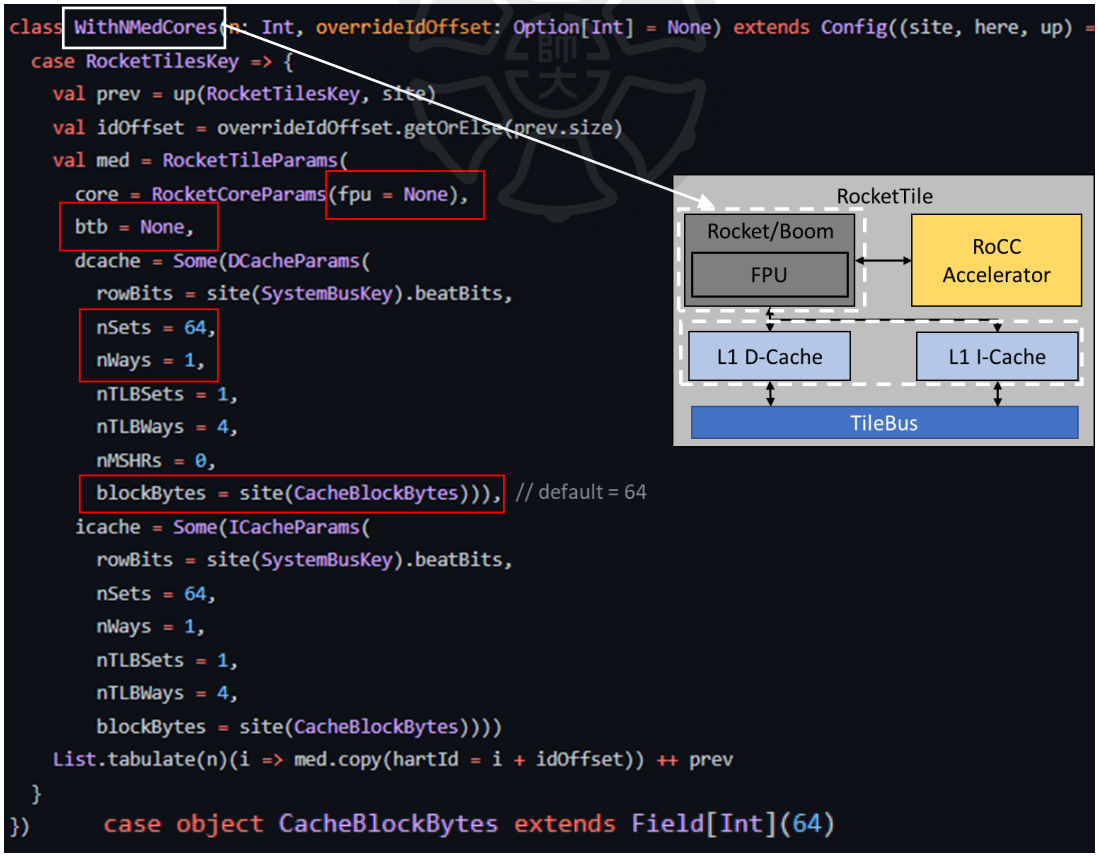


圖 3- 3 Med Rocket Core，藉由參數 fpu=None, btb=None 我們可以知道，這是一顆不支援浮點運算、branch prediction 的核心，且通過公式 nSets*nWays*blockBytes 計算後可以得知此核心擁有 4KB I-Cache 和 4KB D-Cache 存取方式為 Direct mapping。

在圖 3-4 的紅框中，我們可以看到 XLen、useVM、fpu 及 btb，其中 XLen 的意義為核心支援指令集的位元數，Rocket Core 中 XLen 預設值為 64，而 Rocket Core 中的 Tiny Core 為支援整數運算，不支援虛擬記憶體以及使用 32 位元的小 CPU，所以在下圖 3-4 中可以看到 Xlen 為 32 代表使用 32 位元指令集，useVM=false 代表不支援虛擬記憶體，fpu=None 為不支援浮點數運算，代表只支援整數運算，而最後的 btb=None 為不支援 branch prediction。

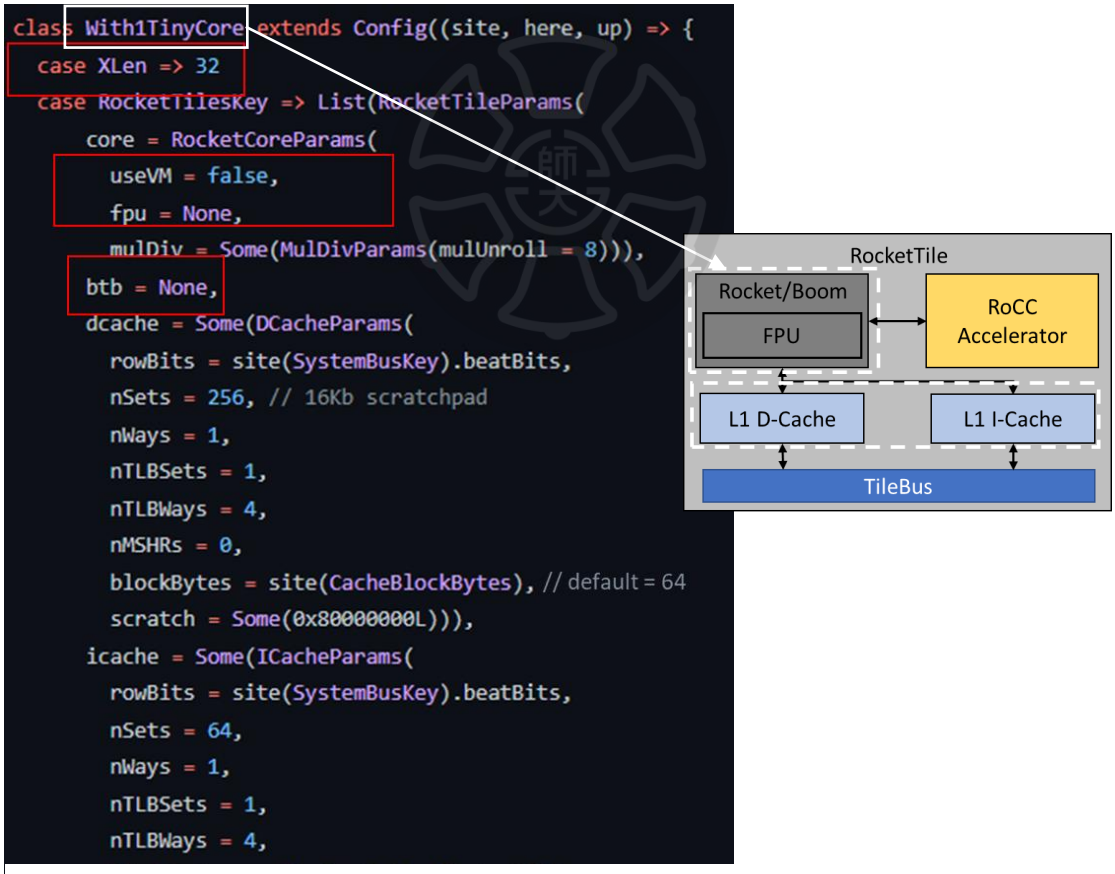


圖 3- 4 Tiny Rocket Core，通過紅框所示的參數 XLen=32 可以得知，這是一顆 32 位元的核心，useVM=false, fpu=None, btb=None 了解此核心不支援記憶體管理單元、浮點數運算以及 branch prediction，並且通過公式 nWays*nSets*blockBytes 可以算出此核心有一個 256KB 大的 D-Cache 以及 4KB 大的 I-Cache，存取方式為 Direct mapping，而白線所示為 Tiny Rocket Core 對應圖 2-4 的整體架構。

前面提到了柏克萊大學開發的 Rocket Core 有四種類型可以讓我們做使用，這四種核心主要硬體功能上的差異整理於表 3-1，所以這也說明了，通過修改 Chisel 中的參數，就能生成符合我們需求的核心，並且還可以通過編寫 Chisel 生成特定功能元件，整合到核心使用，當然最後這步並不是非常容易實現，其最大挑戰是需要與 Rocket Core 做整合，既不破壞現有功能，又增加新的設定。

表 3- 1 Difference of Different Rocket Cores，比較不同 Rocket Core 在硬體上的差別。

核心類型	Big Core	Med Core	Small Core	Tiny Core
位元數	64位元	64位元	64位元	32位元
浮點數運算單元(FPU)	有	無	無	無
記憶體管理單元	有	有	無	無
Instruction Cache	16KB	4KB	4KB	16KB
Data Cache	16KB	4KB	4KB	4KB
Cache Mapping	4-Way Set Associative	Direct Mapped	Direct Mapped	Direct Mapped

2. L2 Cache

Chipyard 中的 L2 Cache 使用 SiFive 開源的 Cache 模組[26]，在圖 3-5 我們可以看到 sets 以及 ways 可以實作 Set Associative Cache，capacityKB 設定 Cache 大小，memCycles 可以設定 latency，writeBytes 設定寫入資料大小。

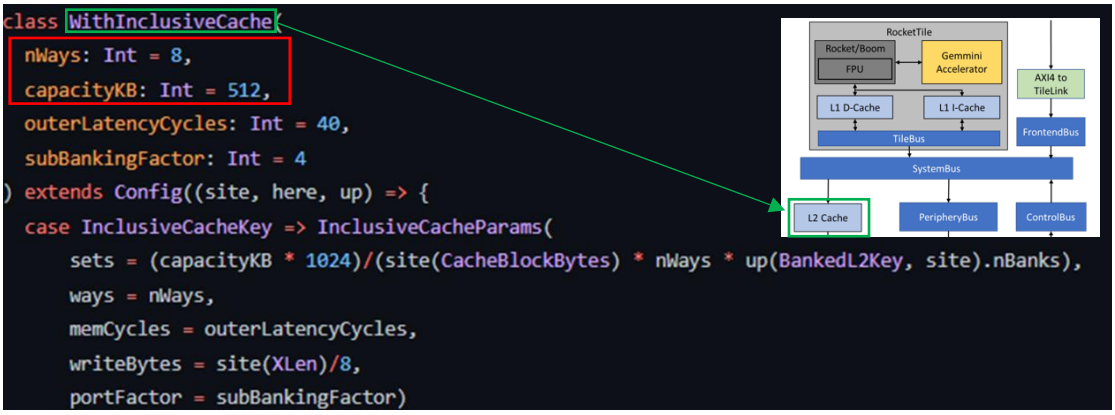


圖 3- 5 L2 Cache 參數，上圖中綠線為 L2 Cache 對應到圖 2-4 整體架構，而紅框為設定 L2 Cache 為 512 KB 的 8-way set associatve Cache。

3. Rocket Base Configuration

Rocket Core 以及 L1 Cache 生成時，我們還需要做基本的設定，例如我們需要設定 Bus 的大小、一次資料傳輸的量及設定 Rocket Core 中的資料型態、各個接口及 Bus 等等，而圖 3-6 以 Chisel 的程式碼來說明 Rocket Core 基本的設定。

```
class BaseConfig extends Config(
  new WithDefaultMemPort() ++
  new WithDefaultMMIOPort() ++
  new WithDefaultSlavePort() ++
  new WithTimebase(BigInt(1000000)) ++ // 1 MHz
  new WithDTS("freechips,rocketchip-unknown", Nil) ++
  new WithNExtTopInterrupts(2) ++
  new BaseSubsystemConfig()
)
```

圖 3- 6 Rocket Base Configuration [32]，在上圖中我們可以看到 WithDefaultMemPort()，這是在設定 Memory port 的起始位址、Bus 的大小及一次資料傳輸的量。而 WithDefaultMMIOPort()為設定 Memory Map I/O 的起始位址、Bus 的大小及一次傳輸資料的量。WithDefaultSlavePort()主要接收外部 DMA 來的資料，所以設定傳輸資料量。WithTimebase()為設定 device tree 的基礎時間，WithDTS()會讓合成電路產生 device tree，WithNExtTopInterrupts()則是設定外部 interrupt 接口的數量，BaseSubsystemConfig()是在設定 Rocket Core 裡的 Bus 和資料型態等等，而上述的 Function 可以於 Rocket Chip subsystem[25]裡面修改設定內容。

4. Gemmini 加速器

Gemmini 加速器主要核心是一個執行矩陣乘法的 Systolic Array，資料會通過硬體電路做運算，運算後的值也要有記憶體可以保存，所以

Gemmini 有很多跟資料型態及資料儲存相關且參數化的硬體電路，詳細運算流程及應用請參考以矩陣乘法為基礎應用硬體加速器於一維卷積計算之研究[38]以及基於 RISC-V 架構之脈動陣列一維卷積運算研究[39]。

圖 3-7 為 Gemmini 架構與參數對應圖，本論文將 Gemmini 加速器分為四部分做說明:

(1) 資料型態:

首先資料型態的部分分為三種，inputType 對應到輸入，accType 對應到 Accumulator Memory 裡累加後的資料型態，spatialArrayOutputType 對應到 Systolic Array 運算中間的輸出，三種資料型態都可以做修改，但由於要跑的應用程式使用 C 語言，所以 inputType 以及 accType 必須符合 C 語言的標準做更改，否則沒辦法生成，而運算中間的資料型態則可以自由修改。

(2) Systolic Array:

Systolic Array 內的區塊可以通過 tileRows 及 tileColumns 做調整，Systolic Array 內每個區塊的大小通過 meshRows 以及 meshColumns 做調整。接著是 Systolic Array 支援的資料流方式，透過 dataflow 就可以進行修改，可以選擇支援 Output Stationary、Weight Stationary 或兩者都支援。

(3) 資料儲存:

資料儲存有兩部分，第一部分是 Scratch Pad 的大小及區塊，可以通過 sp_capacity 指定大小，sp_banks 指定切分區塊數目。第二部分為 Accumulator memory 的大小及區塊，透過 acc_capacity 指定大小，acc_banks 指定切分區塊數目。

(4) Function Unit:

搬資料進出 Gemmini 加速器可以做 scaling，不需要也可以將 mvin_scale_arg 及 acc_scale_arg 設定為 None 關閉這個功能。資料運算完可以通過硬體電路的激活函數，若是不需要就將 non_linear_activation 設為 false 即可。

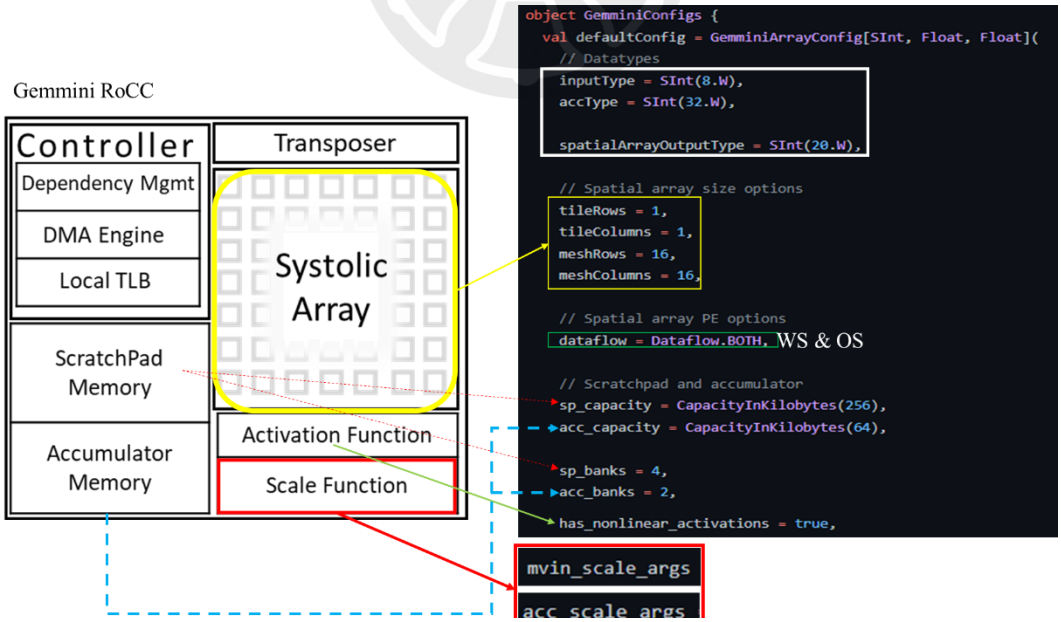


圖 3- 7 Gemmini 架構與參數對應圖，由上圖參數會產生一個具有大小為 16*16 的 Systolic Array，input 大小為 8-bit，累加器中資料大小為 32-bit，Systolic Array 運算中間輸出為 20-bit，4 塊 Scratchpad 大小總和為 256KB，2 塊 Accumulator Memory 大小總和為 64KB，支援 WS 以及 OS 資料流，支援激活函數和 Scaling Function 的 Gemmini 加速器。

5. Rocket Chip chiptop

Rocket Chip 的 Chiptop[27]我們可以通過參數調整外部接口，也可以定義 interrupt 信號，讓我們能更好地通過外部的信號來控制整個 Rocket Chip，並且對其進行資料傳輸和交換等操作，而 Rocket Chip chiptop 對應到 Rocket Chip(圖 2-4)的架構及其詳細參數和功能設定請參閱附錄一。

(二) 元件組合

在元件組合這裡，我們要將元件產生提到的元件組合起來，並且使用 Rocket Chip chiptop(附圖 1)作為 top level module，因為 Rocket Chip chiptop 提供了 AXI4 協議的接口可以讓我們使用，有了 AXI4 協議的接口可以做使用，我們就可以將資料通過 AXI Bus 傳進 Rocket Chip 裡面做運算，也可以從 Rocket Chip 把資料丟到主記憶體做儲存，可以說只要有了 AXI Bus 我們就可以對這個 SoC 進行操控。

圖 3-8 是透過 Chisel 組合的元件，我們只要將需要使用的元件通過 **new *元件名稱*** ++ 這種形式，即可加入至組合電路，黃色英文字代表著該元件所對應的功能或該元件產生的硬體電路，其餘的各個方框及線條則代表該 Chisel 元件在整個 Rocket Chip(圖 2-4)中的位置。

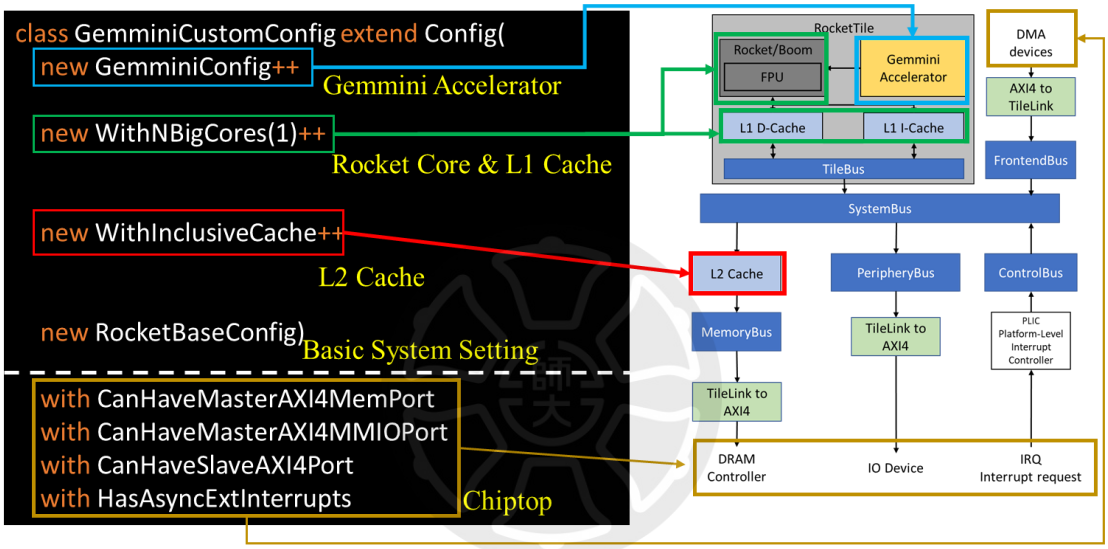


圖 3- 8　元件組合對應到 Rocket Chip，藍線為 Gemmini 加速器，綠線為 Rocket Core 以及 L1 Cache，紅線對應到 L2 Cache，橘黃色的線為附圖 1 中 ExampleRocketSystem 的 AXI 接口對應到整體架購。

(三) Generate Verilog Code

生成 Verilog 這一個步驟非常簡單，將組合好的 Chisel(圖 3-8)透過使用 FIRRTL 硬體編譯器編譯、轉換及優化(圖 3-9)後，我們就可以獲得客製化出的 Verilog 程式碼，並且可以將此電路用在各個支援完整 FPGA 電路發展平台上繼續做發展，圖 3-10 為附圖 1 ExampleRocketSystem 通過 FIRRTL 轉換後的三個 AXI4 接口的 Verilog 程式碼。

圖 3- 9 FIRRTL 優化流程，分別是 Dead expression elimination、Statistics collection 及 Netlist manipulation

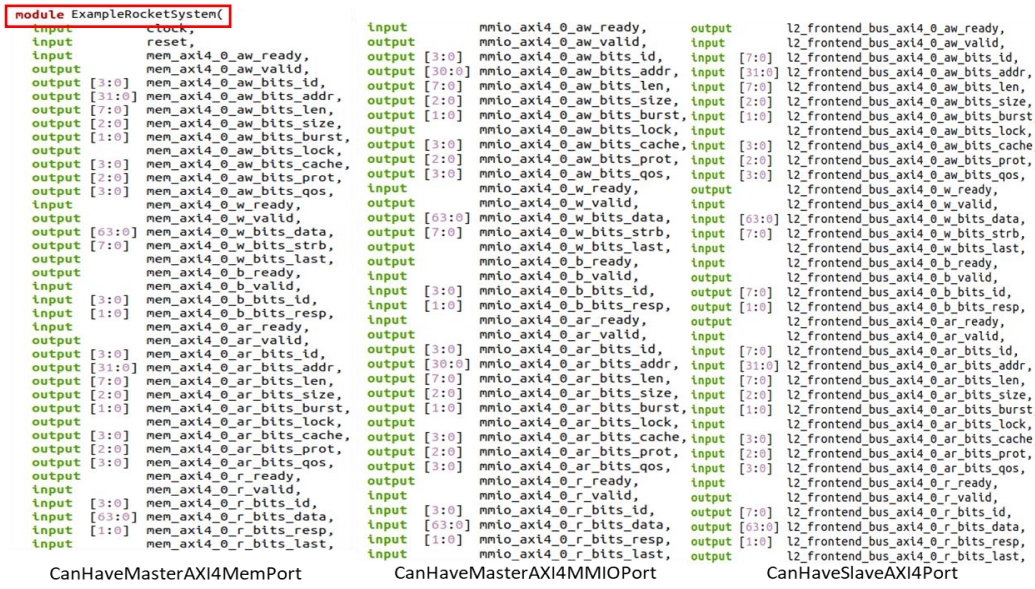


圖 3- 10 附圖 1 ExampleRocketSystem 經 FIRRTL 轉換後的 Verilog 程式碼，可以在紅框看到有一個 module 為 ExampleRocketSystem，由此圖得知即便做了元件組合，FIRRTL 也是會將每個元件分別轉換後再進行組合。

(四) Generate FPGA based SoC

通過 FIRRTL 硬體編譯器後，我們已經有生成好的 Verilog 可以讓我們進行使用，藉由各個 FPGA 設計平台(本論文使用 Vivado 做為開發平台)，我們將其導入到新的專案，在導入之前要先決定使用哪一塊開發版，選定之後開啟專案時要將板子接口的 constraint 檔加入到專案中。由於沒有整理過的 Rocket Chip 外部接線的信號(圖 3-10)非常雜亂，所以先將其封裝成為一個 IP Core，本論文使用 Vivado 來對 Rocket Chip 做封裝[29]，讓我們在與其他元件做連接以及看上去都比較輕鬆，之後通過 AXI Bus 接到我們使用的 IO，例如 UART、GPIO 及 Ethernet 等外部接口，以及接到我們 FPGA 開發板的 DRAM 控制器，並且設定 Clock 以及一些需要使用到的 IP Core，整理好信號線確認沒問題後，通過 wrapper 將 block design 具體化為硬體描述語言，就可以開始進行 Synthesis 及 Implementation，最後產生 bitstream 及 memory config file 後就可以燒錄至 FPGA 了，整體流程如圖 3-11。

圖 3-12 中紅框所示為封裝後的 Rocket Chip，黃線所示為 Rocket Chip AXI Master 連接到 DRAM 控制器的 AXI4 Slave Port，在附圖 1 中的參數表示為 CanHaveMasterAXI4MemPort。綠線所示為 I/O AXI Master 連接到 Rocket Chip 的 AXI4 Slave Port(附圖 1 中的 CanHaveSlaveAXI4Port)，紫線為 Rocket Chip AXI Master 連接到 I/O AXI Slave(附圖 1 中的 CanHaveMasterAXI4MMIOPort)，而橘線所示為 I/O 傳遞 interrupt 進入 Rocket Chip 的 Bus(附圖 1 中的

HasAsyncExtInterrupts)，最後藍框中的名字為圖 3-10 中紅框所示的

ExampleRocketSystem。

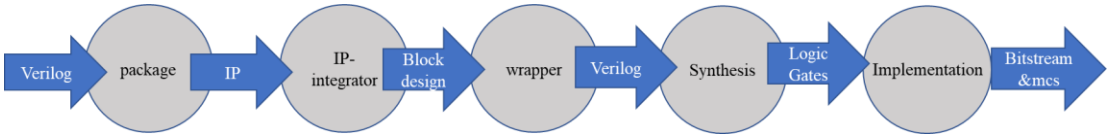


圖 3- 11　FPGA 開發流程，1.將 Verilog 封裝為 IP。2.通過 IP-integrator 設計生成 Block Design。3.使用 wrapper 將 Block Design 轉為 Verilog 做後續使用。4.Synthesis。5.Implementation 完成後生成 Bitstream 以及 memory config file 就完成生成 FPGA based SoC 的任務。

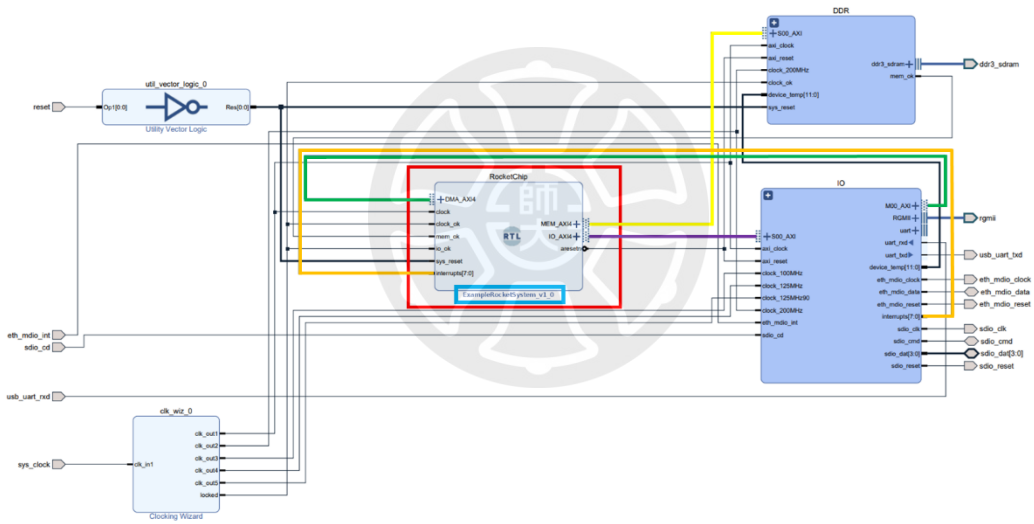


圖 3- 12　Vivado IP-Integrator 上整體的 FPGA 硬體電路架構，上圖中有的電路由五個部件組成，左上的為 reset 信號通過一個反向器，左下為 Clock Wizard，右上為 DRAM 的控制器，右下為 IO Block，而中間紅框標示的為通過圖 3-8 組合元件以及附圖 1 的 Rocket chiptop 合成後的 Rocket Chip。只有一個 Rocket Chip 時，因為缺少記憶體以及 I/O，所以沒辦法形成一個完整的計算機系統，將 Rocket Chip 連接記憶體、I/O 及 Clock 後，我們就能得到一個完整的計算機系統。黃、橘、紫及綠線為附圖 1 紅框標示出的四個接口。

3-2 Software Building Flow

近年來，各大開源社群以及使用 RISC-V 指令集架構的公司都在推動使用 Linux 的 OS 對在 RISC-V 指令集架構下 CPU 的開機流程[30]，最主要的原因是，RISC-V 為開源的指令及架構，未來的發展性是可以預見的。RISC-V 的 CPU 及指令集跟現有常見的都不一樣，最開始沒有 Bootloader 以及 Interface 可以使用，所以即便基於 RISC-V 指令集的 CPU 有支援多種特權等級的模式，也很難有辦法可以把 Linux 放在 RISC-V 的 CPU 上啟動。

而 CPU 支援特權等級重要的地方就在於，要想啟動 Linux，需要支援虛擬記憶體，而特權等級是通過虛擬記憶體實現，若是不支援虛擬記憶體，意味著我們會一直在最高權限的狀態，這時候我們可以直接對物理地址裡面的值做修改，系統安全性不足，也就沒有特權等級的區別，所以若是 CPU 不支援特權等級，意味著不支援作業系統。

RISC-V 的特權等級又分為三種，分別是 ：

(一) M-mode(machine mode):

M-mode 是最底層的模式，也是每一個標準 RISC-V 處理器必須要實現的模式，此模式擁有最高權限，這意味著他可以直接存取實體地址，除此之外，M-mode 還具備攔截和處理異常的能力，並且可以訪問所有其他模式下的控制狀態暫存器（Control Status Register, CSR）。

(二) S-mode(supervisor mode):

S-mode 在表 3-2 可以看出，是 M-mode 上兩層的模式，作業系統在 S-mode 執行如作業系統核心的軟體時，會通過 OpenSBI[31](Open Supervisor Binary Interface)介面呼叫底層裝置。

(三) U-mode(user mode):

U-mode 就是我們常見的 User space，在這模式我們可以做的就是安全的開發或執行 AI 應用程式。

Rocket Core 支援 MMU，並且也支援特權等級，再加上近年來各大公司及開源社群在 RISC-V 指令集 CPU 上面發展的開機流程，透過上述的種種原因，我們也能架設一個作業系統讓我們生成出的 SoC 在 FPGA 平台上做使用。

表 3- 2 RISC-V 特權等級，三個等級分別對應到的編碼為 User mode(00)、Supervisor mode(01)及 Machine mode(11)，中間的 10 編碼為保留，做為未來擴充先的 mode 時使用。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

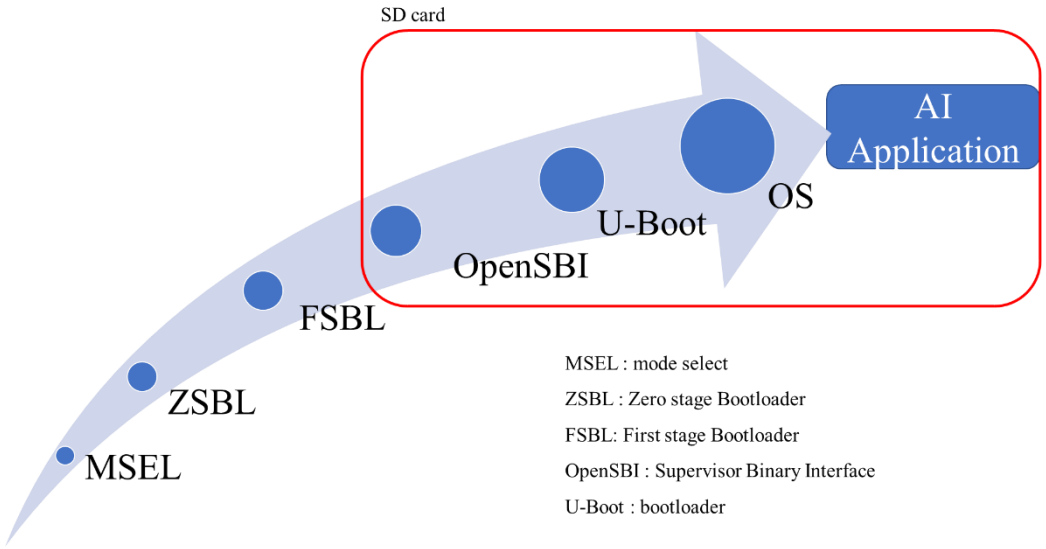


圖 3- 83 Software Boot Flow，從 Mode select 開始，經過 Zero stage Bootloader、First stage Bootloader、Open Supervisor Binary Interface 及 U-Boot 到 Linux OS 上執行程式。

圖 3-13 為本論文的軟體架構流程，本論文將圖 3-13，分為 4 個步驟做說明:

1. **Mode Select(MSEL)**

   SoC 系統燒上 FPGA 開發板開機後，會先經過 Mode Select 這個過程，將模式設定於 Machine Mode。

2. **Zero Stage Bootloader(ZSBL)**

   當 FPGA 開發版開機，經過 MSEL 將模式設定於 M-Mode 後，系統時脈開始產生，CPU 收到時脈訊號後，便會自動從 ROM 的位址開始讀取啟動程式碼，開始讀取即進入下一階段。

3. **First Stage Bootloader(FSBL)**

   FSBL 是通過 ZSBL 讀取 bootrom 後進入的階段，bootrom 檔為 FSBL 最重要的檔案，通過此檔案才能連接後面階段的程序。我們在做硬體時，會先將 bootrom 檔與 device tree 編譯後放進 ROM 中，通過 Chisel

函式 **WithBootROMFile(path/to/bootrom)**[25](附圖 1 內綠框)，指定我們的 bootrom 檔。Rocket Chip 有提供 bootrom 讓我們做使用，但他並沒有支援到可以連結上外部設備的功能，所以本論文使用開源的 bootrom[33]，將我們 device tree 中 SD controller 外部設備 MMIO 的位址(圖 3-14 中紅框前段位址)上去，與 device tree 一起編譯生成 bootrom 檔，再重新編譯硬體，讓 Zero Stage Bootloader 能夠成功抓到就完成這一階段。

```
uart0: uart@60010000 {
    compatible = "riscv,axi-uart-1.0";
    reg = <0x60010000 0x10000>;
    interrupt-parent = <&{/soc/interrupt-controller@c000000}>;
    interrupts = <1>;
    port-number = <0>;
};

mmc: mmc0@60000000 {
    compatible = "riscv,axi-sd-card-1.0";
    clock = <100000000>;
    reg = <0x60000000 0x10000>;
    bus-width = <4>;
    fifo-depth = <256>;
    interrupt-parent = <&{/soc/interrupt-controller@c000000}>;
    interrupts = <2>;
    max-frequency = <50000000>;
    cap-sd-highspeed;
};

eth: eth0@60020000 {
    compatible = "riscv,axi-ethernet-1.0";
    reg = <0x60020000 0x10000>;
    phy-mode = "rgmii-rxid";
    local-mac-address = [00 0a 35 00 00 02];
    interrupt-parent = <&{/soc/interrupt-controller@c000000}>;
    interrupts = <3>;
```

圖 3- 14 I/O device tree，由上圖可以得知有三個外部元件，uart 為使用 UART 外部接口，mmc 為使用 MultiMediaCard interface 意思為配置 SD 卡的外部接口，eth 為使用網路的外部接口。紅框中 reg 的第一個地址為該接口在 MMIO 的起始地址，第二串數字為該接口配置的大小，而 MMIO 起始位址及大小於圖 3-6 可以設定。在 interrupt 上面可以看到使用的數字為 1、2 和 3，在 RISC-V 指令集[20]中有定義數字越小優先級越高，所以以優先級來看 uart0 > mmc > eth。

4. SD card

通過 bootrom 連接進入 SD 卡後，RISC-V 指令集的核心需要有 OpenSBI 將我們的模式由 Machine Mode 切換到 Supervisor Mode，並跳轉到 U-Boot[34]，圖 3-15 為 OpenSBI 設定跳轉至 U-Boot 部分程式。跳轉至 U-Boot 後，U-Boot 會將 linux kernel[35]加載至 DRAM 中，並且啟動，而編譯 U-boot 以及將有使用到的外部設備的 Driver 放入 kernel 並重新編譯的部分在附錄二中有詳細說明。

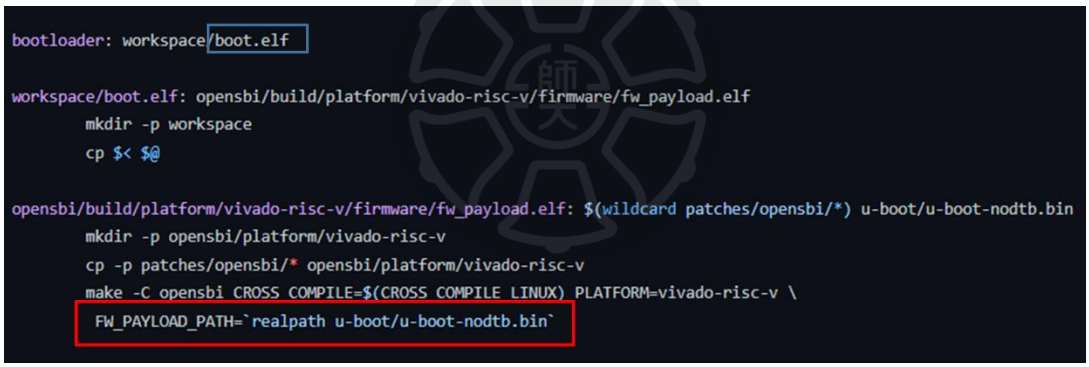


圖 3- 15 OpenSBI 設定跳轉路徑[36]，在上圖紅框中我們可以看到在編譯 OpenSBI 時需要將 U-Boot 跳轉路徑放入到 OpenSBI 裡一起編譯，完成後會產生藍框中的 boot.elf 讓 bootrom 讀取。

最後將上述的資料燒錄進 SD 卡，將 SD 卡分為兩個區塊(圖 3-16)，bootrom 以及後續 Bootloader 需要使用檔案放在第一個區塊，Debian rootfs[37] 放在第二個區塊，接著燒錄至 SD 卡上的詳細流程請參考附錄三，即可完成軟體架構流程，並且啟動作業系統執行應用程式。

```
SD_LOOP=$(
losetup -a | grep `pwd`/ | while IFS=':' read -ra LINE ; do
  echo ${LINE[0]}
done
)

echo "SD image device: ${SD_LOOP}"

sudo sfdisk --no-tell-kernel ${SD_LOOP} <<-__EOF__
1M,${SD_BOOT_SIZE}M,0xE,*
,,,-
__EOF__

sudo partprobe ${SD_LOOP}

UUID=68d82fa1-1bb5-435f-a5e3-862176586eec
sudo mkfs.vfat -F 16 -n BOOT ${SD_LOOP}p1
sudo mkfs.ext4 -E nodiscard -L rootfs -U $UUID ${SD_LOOP}p2
```

圖 3- 16 SD 卡分區，在上圖紅框部分可以觀察到將 SD 卡分為兩個區塊，分別是 BOOT 以及 rootfs。

第 4 章 實驗數據與效能分析

經過第二、三章的討論，描述本論文所使用之軟硬體架構以及構建方式，並且通過本論文的構建流程實作於 FPGA 開發板上。在了解以 RISC-V 為基礎的硬體架構後，在此章節本論文會通過調整部分硬體參數並實際燒錄於 FPGA 開發板，來對個別功能影響的硬體資源使用量做討論，並且在最後會測試在不同的 Systolic Array 下，對於矩陣乘法加速效能比較。第一節會介紹開發及實驗所使用的軟硬體環境，第二節會介紹本論文實驗設計所變更的硬體架構參數以及矩陣乘法進行效能測試的資料，第三節介紹本論文實驗效能評估方法，第四節的實驗結果為不同硬體配置的硬體資源使用量以及大小矩陣在不同大小的 Gemmini 加速器中的加速效果。

4-1 實驗環境介紹

本論文實驗環境會分為兩部份做介紹，第一部份實驗的軟硬體環境如表 4-1。個人電腦在硬體方面，配置有 Intel® Core™ i7-11700 的 CPU 以及 32G 的 DRAM。軟體方面，作業系統使用是 Ubuntu 20.04。而在開發平台上使用的是柏克萊大學開源的 Chipyard 以及 Xilinx Vivado Suite 2021.1 版，在效能測試程式的編寫上面主要使用 C 語言，而在效能測試程式的編譯方面，所使用編譯器為 riscv64-linux-gnu-gcc。

而第二部分為 FPGA 開發板上的環境如表 4-2 所示，FPGA 開發版選用 Digilent 公司使用 Xilinx FPGA 所推出的 Genesys2，CPU 及加速器為

Chipyard 提供開源的 Rocket Core 以及 Gemmini 加速器，OS 為 Debian 5.16.8，DRAM 配置有 1GB，FPGA 為 Xilinx Kintex-7 family XC7K325T-2FFG900C，有 50950 個邏輯單元，每個邏輯單元可以轉換為 4 個 6-input 的 LookUp Table(LUT)以及 8 個 Flip Flop(FF)，所以共有 203800 個 LUTs 和 407600 個 FFs 可以讓我們使用於合成電路，有 16Mbits 的 Block RAM 供我們合成所需使用之記憶體，最後還配置有 16MB 的 Quad-SPI Flash 供我們在燒錄 FPGA 時做使用。

表 4- 1　軟硬體實驗環境介紹

實驗環境	
OS	Ubuntu 20.04
CPU	Intel® Core™ i7-11700
DRAM	32GB
Development Platform	Chipyard, Vivado Suite 2021.1
程式語言	C
編譯工具	riscv64-linux-gnu-gcc

表 4- 2　FPGA 開發版環境配置

Genesys2 開發版環境	
OS	Debian 5.16.8
CPU	64-bit RISC-V Rocket Core
Hardware Accelerator	Gemmini v6.0.3
DRAM	1GB
FPGA Part#	Xilinx XC7K325T-2FFG900C
Logic Cell	50950 logic slice (4 6-input LUTs & 8 flip-flops each)
Block RAM	16Mbits
Quad-SPI Flash	16MB
Picture	

4-2 實驗設計

本論文實驗設計主要會分為硬體以及軟體做討論，硬體會針對 Gemmini 加速器的硬體配置做調整，並且比較其中硬體資源使用量。而軟體會針對輸入至 Gemmini 加速器之測試資料做設計，並且對測試資料做效能比較。

(一)硬體：

本論文針對 Gemmini 加速器中導致硬體資源消耗增加的參數做調整，例如 Systolic Array 支援的資料流、Systolic Array 大小、SRAM 大小、Scaling Function、資料型態以及外部的 L2 Cache 做調整，讓我們可以通過實驗了解各元件硬體資源消耗情況。在本論文的硬體實驗設計中，會使用 Systolic Array 為 4*4 以及 8*8 來比較其硬體資源消耗。在圖 4-1 可以看到 Systolic Array 為 4*4 以及 8*8 的 Default 配置，在表 4-3 可以看到對使用 4*4 Systolic Array 的 Gemmini 加速器硬體參數做調整的部份，而在表 4-4 可以觀察到使用 8*8 Systolic Array 的 Gemmini 加速器硬體參數做調整的部份。

- Systolic Array 8x8硬體配置為
  - Dataflow = Weight Stationary
  - Scratchpad size = 64KB
  - Accumulator Memory size = 16KB
  - Without mvin_scale function
  - Has 128KB L2 cache
  - Data Type: input: 8-bit SInt、output: 20-bit Sint、accumulator: 32-bit Sint
- Systolic Array 4x4 硬體配置為
  - Dataflow = Weight Stationary以及Output Stationary
  - Scratchpad size=256KB
  - Accumulator Memory size=64KB
  - With mvin_scale function
  - Has 512KB L2 cache
  - Data Type: input: 8-bit SInt、output: 20-bit Sint、accumulator: 32-bit Sint

圖 4- 1 Systolic Array 為 4*4 以及 8*8 的設定 1(Default)硬體配置

表 4- 3 Systolic Array 為 4*4 的 Gemmini 加速器硬體參數調整，可以從上圖中紅字看到設定 2~5 分別對 Dataflow、Scratchpad size、Accumulator size、move in scaling 以及 L2 Cache 做調整，而 Default 實際參數在圖 4-1 可以得知。

4x4	Dataflow	Spad Size	AccMem Size	Mvin_scale	L2 Cache
設定1(Default)	WS & OS	256KB	64KB	Yes	512KB
設定2	WS	256KB	64KB	Yes	512KB
設定3	WS & OS	64KB	16KB	Yes	512KB
設定4	WS & OS	256KB	64KB	No	512KB
設定5	WS & OS	256KB	64KB	Yes	No

表 4- 4 Systolic Array 為 8*8 的 Gemmini 加速器硬體參數調整，可以由上圖紅字看到設定 2~3 調整的參數有將 Accumulator 中的資料型態改為 16-bit 的 Signed integer，以及將 L2 Cache 移除，而 Default 實際參數在圖 4-1 可以得知。

8x8	Accumulator DataType	L2 Cache
設定1(Default)	32-bit SInt	128KB
設定2	16-bit SInt	128KB
設定3	32-bit SInt	No

(二)軟體：

軟體部份，本論文會使用一組小矩陣以及一組大矩陣對 31.25Mhz 4*4 Systolic Array 的 Gemmini 加速器配合 Rocket Core、25Mhz 8*8 Systolic Array 的 Gemmini 加速器配合 Rocket Core、25Mhz 的 Rocket Core 以及 50Mhz 的 Rocket Core 在 FPGA 開發板上做效能以及執行時間的評估，之所以選擇兩組不同的矩陣資料原因是通過小矩陣我們可能會很難看出在 Gemmini 加速器上面的加速效果，因為需要運算的資料過小，甚至可能 CPU 算的都比加速器效果好，而在大矩陣上，我們能更好的看出實際上的加速效能提升，通過大小矩陣運算的效能比較，會比單純大矩陣測試效能

有著更較明顯的對比，圖 4-2 為小矩陣測資，圖 4-3 為大矩陣測資。

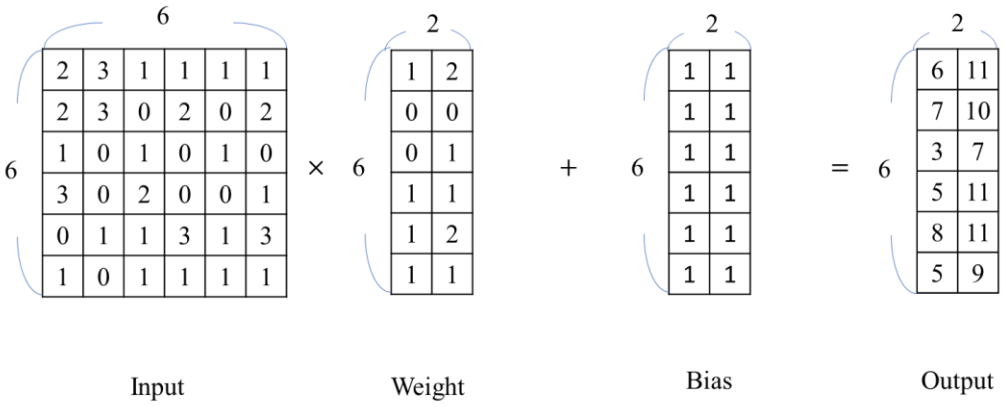


圖 4- 2　小矩陣測資，小矩陣測資為一個 6*6 的 input 與一個 6*2 的 weight 相乘後，加上 6*2 的 bias，最終得到 6*2 的 output，其中內容如上圖各矩陣中所呈現。

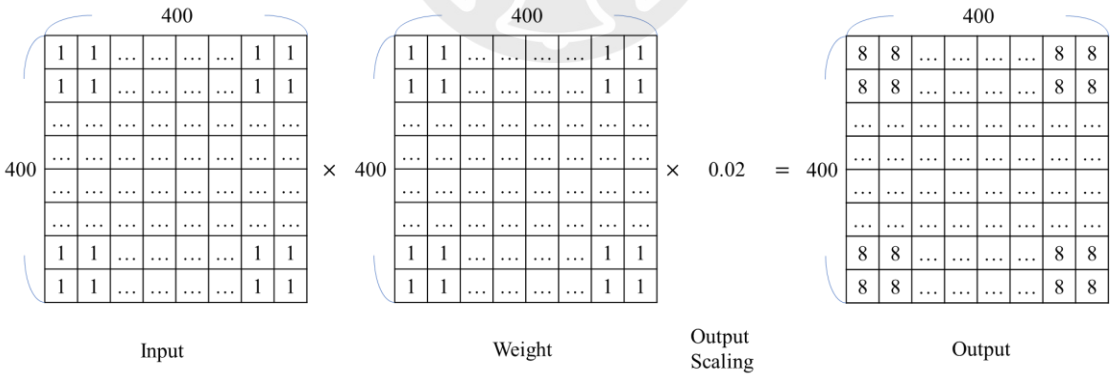


圖 4- 3　大矩陣測資，大矩陣測資為一個 400*400 且內容全部為 1 的 input 與一個 400*400 且內容全部為 1 的 weight 相乘後，再乘上一個為 0.02 的 scaling 值，最後可以得到結果為 400*400 且內容全部為 8 的 output。

4-3 實驗效能評估

本章節會對實驗效能的評估方法做說明，與上一個章節相同，會分為硬體以及軟體做討論。

(一)硬體：

硬體分面，我們調整 Gemmini 加速器硬體配置，以及 L2 Cache 大小，在做完硬體設計流程後，通過 Vivado 合成電路後產生的硬體資源占比做硬體資源消耗的比較。

(二)軟體：

軟體方面，本論文使用圖 4-4 的架構來計算通過 Gemmini 加速器執行後的 Clock Cycle，並且在執行的效能及時間比較上使用公式表示 B 執行加速效果為 A 的 N 倍。

$$A(\text{Clock Cycle or Second}) \div B(\text{Clock Cycle or Second}) = N \qquad (4\text{-}1)$$

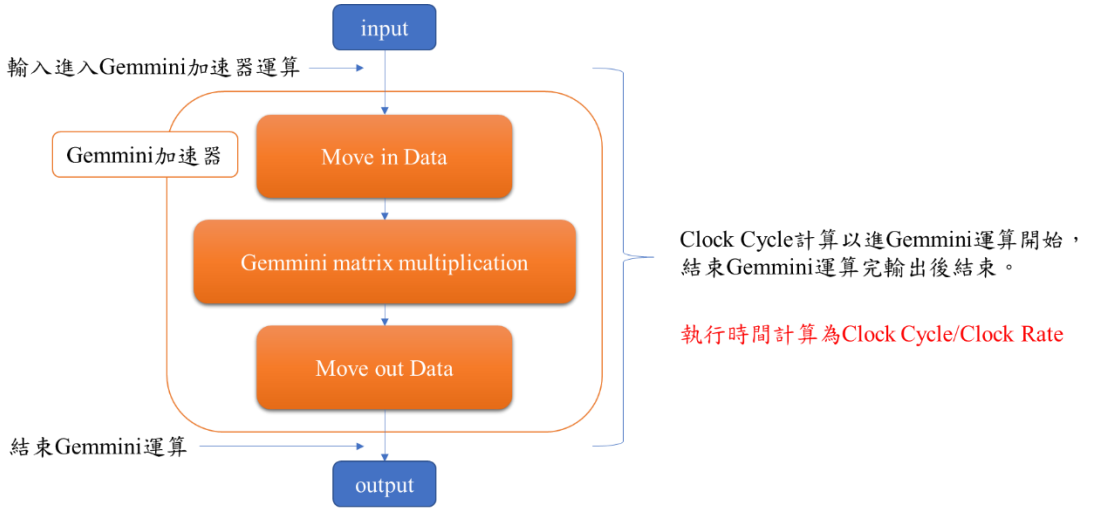


圖 4- 4 Gemmini 加速器運算 Clock Cycle 統計範圍

4-4 實驗結果

本章節由表 4-3 及表 4-4 去對硬體做使用資源比較，並且使用 Default(圖 4-1)的硬體配置對 Gemmini 加速器與 Rocket Core 執行矩陣乘法的加速效果做效能比較及分析，這裡分別對硬體及軟體做相對應討論。

(一)硬體：

首先，在沒有 Gemmini 加速器時，只使用 64-bit 支援浮點數運算、Branch prediction、記憶體管理單元和 16KB 的 4-way set-associative Instruction and Data Caches 的 Default Rocket Core 時的硬體資源使用量如圖 4-5。而圖 4-6 為 Rocket Core 硬體參數加入擁有 4*4 Systolic Array 的 Gemmini 加速器後，硬體資源使用量與單一核心比較，可以看到在加入 Gemmini 加速器後，LookUp Table(LUT)、Flip-Flop(FF)以及 Block RAM(BRAM)使用量急劇增加，這代表 Gemmini 加速器使用最多的是這三個硬體資源。接下來本論文會以 LUT、FF 以及 BRAM 做硬體資源消耗比較，表 4-5 為調整表 4-3 中紅字所示的參數，並且在紅字的部份為調整該參數節省最多的硬體資源。而表 4-6 為調整表 4-4 中紅字所示的參數，在紅字的部份為調整該參數節省最多的硬體資源。

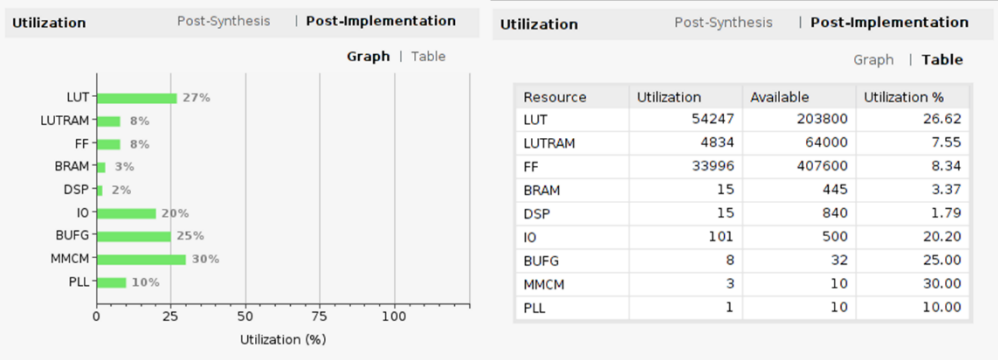
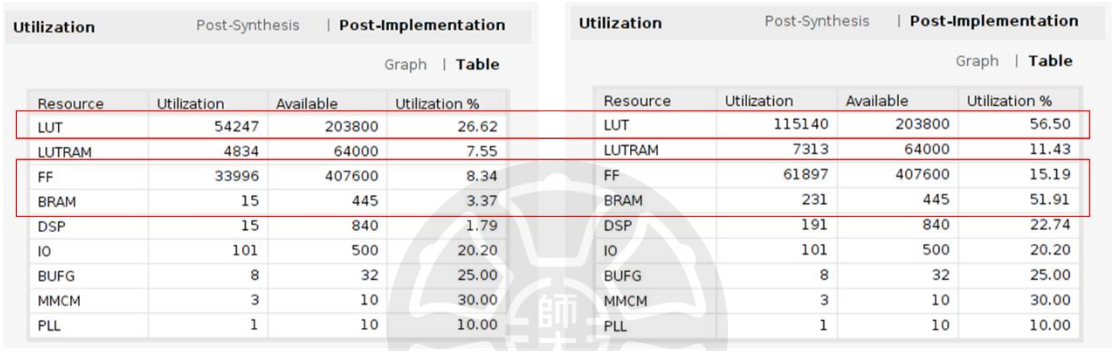


圖 4- 5 Rocket Core 硬體資源使用量，通過左邊的橫條圖以及右邊的詳細圖表可以清楚知道單一個 Rocket Core 在沒有 Gemmini 加速器的情況，硬體資源使用占比並不高，LUT 只使用了 27%，Flip Flop 使用了 8%。



Rocket Core　　Default Gemmini 4*4

圖 4- 6 Rocket Core 與 4*4 Systolic Array Gemmini 加速器硬體資源比較，通過上圖紅框可以得知加入 Gemmini 加速器後，硬體資源 LUT 部份多了 30%使用量，Flip-Flop 多了約 7%，Block RAM 多了約 48%。

表 4- 5 Systolic Array 為 4*4 的 Gemmini 加速器硬體資源使用比較，在表中我們可以看到將 Dataflow 調整為 Weight Stationary 時，能夠為 LUT 節省將近 5%的資源，而將 L2 Cache 移除，可以為 FF 減少 0.4%的消耗，為 BRAM 減少 30%的消耗。通過此表也可以清楚了解調整特定硬體架構，能夠優化硬體資源使用量。

4x4	LUT (203800個)	FF (407600個)	BRAM (445個, 16Mbits)
Rocket Core	26.62%	8.34%	3.37%
Rocket Core + 設定1(Default)	56.5%	15.19%	51.91%
Rocket Core + 設定2	51.62%	14.89%	51.91%
Rocket Core + 設定3	55.83%	15.09%	38.43%
Rocket Core + 設定4	54.52%	14.81%	51.91%
Rocket Core + 設定5	54.07%	14.71%	21.35%

表 4- 6 Systolic Array 為 8*8 的 Gemmini 加速器硬體資源使用比較，由於 FPGA 開發版硬體資源的限制，因此並沒有辦法將 Systolic Array 為 8*8 的 Gemmini 加速器的 Default 配置與 4*4 相同(圖 4-1)。藉由此表可以看出 Default 的硬體資源使用量分別高達了 77.37%、47.16%以及 14.61%，而經過調整 L2 Cache 我們可以獲得低消耗的 BRAM，通過調整 Accumulator 裡面的資料型態更是可以將 LUT 及 FF 的消耗壓低至 55%和 16%，但由於運算精確度的需求，基本上不會對資料型態做調整。

8x8	LUT (203800個)	FF (407600個)	BRAM (445個, 16Mbits)
Rocket Core	26.62%	8.34%	3.37%
Rocket Core + 設定1(Default)	77.37%	47.16%	14.61%
Rocket Core + 設定2	75.43%	46.69%	6.97%
Rocket Core + 設定3	55.19%	15.91%	36.63%

(二)軟體：

本論文在第四章第二節實驗設計有提到，使用一組小矩陣一組大矩陣做效能和執行時間的評估及比較，因此，在軟體的實驗結果中主要討論以圖 4-1 的硬體架構以及 Rocket Core 執行大小矩陣乘法的結果，通過圖 4-4 所圈出範圍來計算執行矩陣乘法的 Clock Cycle 和時間，並且使用公式計算各種硬體架構的加速效果。表 4-7 及表 4-8 為大小矩陣在同 Clock Rate 下執行矩陣運算的效能比較，表 4-7 可以看到擁有 4*4 Systolic Array 的 Gemmini 加速器對比 Rocket Core 能夠有著 1.2 倍的加速效果，而 8*8 Systolic Array 的 Gemmini 加速器對比 Rocket Core 能夠有著 1.42 倍的加速效果，但由於矩陣資料量不夠大，所以很難看出 Gemmini 加速器的加速效果。表 4-8 為大矩陣執行的效能比較，可以看到擁有 4*4 Systolic Array 的 Gemmini 加速器對比 Rocket Core 能夠有著 331 倍的

加速效果，而 8*8 Systolic Array 的 Gemmini 加速器對比 Rocket Core 能夠有著 1298 倍的加速效果，通過大矩陣我們就可以很明顯看出 Gemmini 加速器與 Rocket Core 運算矩陣乘法 Clock Cycle 的差異。

表 4-7 小矩陣效能比較，在小矩陣運算的 Clock Cycle 中，並沒有很明顯的差異，代表在資料量較小時，加速效果並不理想，但 8*8 Systolic Array 的 Gemmini 加速器還是能比 4*4 Systolic Array 的 Gemmini 加速器快 1.17 倍。

小矩陣	Rocket Core	Rocket Core with Gemmini 4*4	Rocket Core with Gemmini 8*8
Clock Cycle	1474	1220	1040
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini)	-	1.21	1.42
Speedup (Rocket Core with Gemmini 4*4 over Rocket Core with Gemmini 8*8)	-	-	1.17

表 4-8 大矩陣效能比較，在大矩陣的運算中，我們就可以很明顯的看出效能差異，並且在 4*4 Systolic Array 的 Gemmini 加速器以及 8*8 Systolic Array 的 Gemmini 加速器都有著很好的加速表現，而 8*8 的 Gemmini 加速器還比 4*4 的 Gemmini 加速器快了 3.9 倍。

大矩陣	Rocket Core	Rocket Core with Gemmini 4*4	Rocket Core with Gemmini 8*8
Clock Cycle	2029497435	6117790	1562758
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini)	-	332	1299
Speedup (Rocket Core with Gemmini 4*4 over Rocket Core with Gemmini 8*8)	-	-	3.9

前面對於 Clock Cycle 的效能比較及討論在現實上有一定的限制，特別是當我們想要使用 Rocket Core 搭配 Gemmini 加速器時，事實上需要將 Clock Rate 降低才有辦法成功避免硬體 setup/hold time 的問題，並且成功編譯及合成電

路，所以在上述有關效能的討論中，都是預設在相同 Clock Rate 下的效能比較，而在表 4-9 及表 4-10 中，我們會討論到不同的 Clock Rate 對於執行時間的影響及差異。在表 4-9 中，我們可以觀察到 Rocket Core 於 25Mhz 及 50Mhz 時運算小矩陣的時間，也可以在表中看到 Rocket Core 搭配 4*4 以及 8*8 Systolic Array 的 Gemmini 加速器於 25Mhz 執行小矩陣的時間，通過這張表我們可以得知在小矩陣中，Gemmini 加速器的加速效果並不是非常理想，甚至在 Rocket Core 的 Clock Rate 為 50Mhz 時，4*4 以及 8*8 Systolic Array Gemmini 加速器的執行時間還高於 Rocket Core，這可以說明若只是要進行一般簡單的運算，Gemmini 加速器沒辦法提供非常有效的幫助。在表 4-10 大矩陣執行時間比較中我們可以很明顯的觀察到，在同個 Clock Rate 時，執行時間的加速效果與表 4-8 的效能比較相同，而在不同的 Clock Rate 時，也可以很清楚的在表中紅字上看到加速效果相比於小矩陣來說也是非常好的，所以我們可以得出一個結論是，Gemmini 加速器適合使用於資料量大的運算，若只是執行資料量較小的運算，不僅加速效果不如我們預想的，還讓 Rocket Core 的 Clock Rate 會因為增加了 Gemmini 加速器而下降。

表 4- 9 小矩陣執行時間比較，通過藍字可以得知，在同頻率(25Mhz)下加速的倍率在 4*4 及 8*8 Systolic Array 的 Gemmini 搭配 Rocket Core 對比單 Rocket Core 分別是 1.21 倍以及 1.42 倍，而單 Rocket Core 於 50Mhz 時，在紅字中的執行時間可以看到，反而沒有加速效果，甚至還比 Rocket Core 還慢。

小矩陣	Rocket Core (25Mhz)	Rocket Core (50Mhz)	Rocket Core with Gemmini 4*4 (25MHz)	Rocket Core with Gemmini 8*8 (25Mhz)
Clock Cycles	1474		1220	1040
執行時間(ms)	0.05896	0.02948	0.0488	0.0416
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini) (25Mhz)			1.21	1.42
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini) (50Mhz)			0.60	0.71

表 4- 10 大矩陣執行時間比較，通過藍字可以得知，在同頻率(25Mhz)下加速的倍率在 4*4 及 8*8 Systolic Array 的 Gemmini 搭配 Rocket Core 對比單 Rocket Core 分別是 332 倍以及 1299 倍，而單 Rocket Core 於 50Mhz 時，在紅字中的執行時間可以看到，在資料量夠大的情況，加速效果還是很明顯的，分別達到 166 倍以及 649 倍。

大矩陣	Rocket Core (25Mhz)	Rocket Core (50Mhz)	Rocket Core with Gemmini 4*4 (25MHz)	Rocket Core with Gemmini 8*8 (25Mhz)
Clock Cycles	2029497435		6117790	1562758
執行時間(s)	81.1798974	40.5899487	0.2447116	0.06251032
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini) (25Mhz)			332	1299
Speedup (Rocket Core with Gemmini over Rocket Core without Gemmini) (50Mhz)			166	649

第 5 章 結論

通過 Chipyard 提供的開源元件、Vivado 以及 FPGA 開發板，我們可以對 Chipyard 提供的 Rocket Chip 做硬體資源使用率比較，例如只有 Rocket Core，沒有加速器時，使用到的 LUT 為 26.62%，FF 為 8.34%，BRAM 為 3.37%，而加入配置為 4*4 Systolic Array 的 Gemmini 加速器後，LUT 使用量上升至 56.5%，FF 上升至 15.19%，BRAM 上升至 51.91%。而加入配置為 8*8 Systolic Array 的 Gemmini 加速器後，LUT 使用量上升至 77.37%，FF 上升至 47.16%，BRAM 上升 14.61%，通過硬體資源使用量可以得知，只要 FPGA 開發板硬體資源足夠，即適用本論文的流程。而軟體部分在 FPGA 開發板的效能比較，小矩陣上，4*4 Systolic Array 的 Gemmini 加速器比 Rocket Core 快了 1.21 倍，而 8*8 Systolic Array 的 Gemmini 加速器比 Rocket Core 快了 1.42 倍。大矩陣上，4*4 Systolic Array 的 Gemmini 加速器比 Rocket Core 快了 332 倍，而 8*8 Systolic Array 的 Gemmini 加速器比 Rocket Core 快了 1299 倍，比 4*4 Systolic Array 的 Gemmini 加速器快了 3.9 倍。而在不同 Clock Rate 的執行時間比較上面，能看到小矩陣使用 Gemmini 加速器跟 50Mhz 的 Rocket Core 比較並沒有加速的效果，執行時間還更長，大矩陣方面則可以了解到使用資料量大的運算時 Gemmini 可以很有效地對運算過程做加速。而通過討論大小矩陣的效能及執行時間比較和評估，本論文流程也能夠更有效的證明 Gemmini AI 硬體加速器實際在硬體上的加速效果。

參考文獻

[1] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs." *in IEEE Micro*, **2020**, *vol. 40, no. 4, pp. 10-21*, doi: 10.1109/MM.2020.2996616.

[2] A. Waterman, K. Asanovic, "The RISC-V instruction set manual, volume II: Privileged architecture." *RISC-V Foundation*, **2019**.

[3] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, A. Waterman, "The rocket chip generator." *EECS Department, University of California, Berkeley*, **2016**, Tech. Rep. UCB/EECS-2016-17 4.

[4] C. Celio, P. -F. Chiu, K. Asanović, B. Nikolić and D. Patterson, "BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS," in *IEEE Micro*, **2019**, *vol. 39, no. 2, pp. 52-60,* doi: 10.1109/MM.2019.2897782.

[5] F. Zaruba, L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* **2019**, *27(11), 2629-2640.*

[6] lowRISC. LowRISC/IBEX: Ibex is a small 32 bit RISC-v CPU core, previously known as Zero-riscy. GitHub. Retrieved July 5, 2022, from https://github.com/lowRISC/ibex

[7] Y. Lee, C. Schmidt, A. Ou, A. Waterman, K. Asanović, "The Hwacha vector-fetch architecture manual." *EECS Department, University of California, Berkeley,* **2015**, Tech. Rep. UCB/EECS-2015-262.

[8] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, Y. Sophia Shao, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration." *2021 58th ACM/IEEE Design Automation Conference (DAC)*, **2021**, *pp. 769-774*, doi: 10.1109/DAC18074.2021.9586216.

[9] Schmidt, C., & Izraelevitz, A. A fast parameterized sha3 accelerator. *In tech. rep.. EECS Department*, *University of California,* **2015**.

[10] NVIDIA. *NVDLA*. NVIDIA Deep Learning Accelerator. Retrieved July 5, 2022, from http://nvdla.org/, **2015.**

[11] IceNet. IceNet - Chipyard main documentation. (n.d.). Retrieved July 5, 2022,from https://chipyard.readthedocs.io/en/latest/Generators/IceNet.html, **2019.**

[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, K. Asanović, "Chisel: Constructing hardware in a Scala embedded language." *DAC Design Automation Conference 2012*, **2012**, *pp. 1212-1221*, doi: 10.1145/2228360.2228584.

[13] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations." *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, **2017**, *pp. 209-216*, doi: 10.1109/ICCAD.2017.8203780.

[14] Riscv-Software-Src. *Spike, a RISC-V ISA Simulator*. GitHub. Retrieved July 5, 2022, from https://github.com/riscv/riscv-isa-sim

[15] W. Snyder, "Verilator and systemperl. " *In: North American SystemC Users' Group, Design Automation Conference*. **2004**.

[16] Synopsys. *VCS Functional Verification Solution*. Functional Verification Solution. Retrieved July 5, 2022, from https://www.synopsys.com/verification/simulation/vcs.html

[17] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, **2018**, *pp. 29-42*, doi: 10.1109/ISCA.2018.00014.

[18] H. Cook, W. Terpstra, Y. Lee, Diplomatic design patterns: A TileLink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, **2017**.

[19] D. Flynn, "AMBA: enabling reusable on-chip designs," *in IEEE Micro*, **1997**, *vol. 17, no. 4, pp. 20-27*, doi: 10.1109/40.612211.

[20] A. Waterman, K. Asanovic, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10.", **2017**.

[21] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, T. Krishna, "Scale-sim: Systolic cnn accelerator simulator." *arXiv preprint arXiv:1811.02883*, **2018**.

[22] Chipsalliance. *Rocket-chip/BankedL2Params.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/subsystem/BankedL2Params.scala, 2**020.**

[23] Chipsalliance. *Rocket-chip/icache.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/rocket/ICache.scala, **2021.**

[24] Chipsalliance. *Rocket-chip/hellacache.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/rocket/HellaCache.scala, **2021**.

[25] Chipsalliance. *Rocket-chip/configs.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/subsystem/Configs.scala, **2021**.

[26] Sifive. *Sifive Cache*. GitHub. Retrieved July 5, 2022, from https://github.com/sifive/block-inclusivecache-sifive/tree/e3a3000cc1fd4cdf3a4e638e4d081b8aae94ebf0, **2020**.

[27] Chipsalliance. *Rocket-chip/examplerocketsystem.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/system/ExampleRocketSystem.scala, **2020**.

[28] S. Math, R. Manjula, S. Manvi, P. Kaunds, "Data transactions on system-on-chip bus using AXI4 protocol," *2011 INTERNATIONAL CONFERENCE ON RECENT ADVANCEMENTS IN ELECTRICAL, ELECTRONICS AND CONTROL ENGINEERING*, **2011**, *pp. 423-427*, doi: 10.1109/ICONRAEeCE.2011.6129797.

[29] Xilinx. *Packaging custom IP for using in Ip Integrator*. Xilinx. Retrieved July 5, 2022, from https://www.xilinx.com/video/hardware/packaging-custom-ip-integrator.html, **2016**.

[30] Western Digital Corporation. An Introduction to RISC-V Boot Flow. Retrieved July 5, 2022, from https://riscv.org/wp-content/uploads/2019/12/Summit_bootflow.pdf, **2019**.

[31] Western Digital Corporation. *RISC-V open source supervisor binary interface*. GitHub. Retrieved July 5, 2022, from https://github.com/riscv-software-src/opensbi, **2019**.

[32] Chipsalliance. *Rocket-chip/configs.scala at 114325B27CFE5312C86A8A325B187BE9455A62AF · chipsalliance/rocket-chip*. GitHub. Retrieved July 5, 2022, from https://github.com/chipsalliance/rocket-chip/blob/114325b27cfe5312c86a8a325b187be9455a62af/src/main/scala/system/Configs.scala, **2021**.

[33] E. Tarassov, Bootrom. GitHub. Retrieved July 5, 2022, from https://github.com/eugene-tarassov/vivado-risc-v/blob/master/bootrom/bootrom.c, **2021**.

[34] U-Boot. *U-Boot*. GitHub. Retrieved July 5, 2022, from https://github.com/u-boot/u-boot, **2021**.

[35] Torvalds , L. Linux kernel. Kernel/Git/Stable/linux.git - linux kernel stable Tree. Retrieved July 5, 2022, from https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/, **2022**.

[36] E. Tarassov, Build software. GitHub. Retrieved July 5, 2022, from https://github.com/eugene-tarassov/vivado-risc-v/blob/master/Makefile, **2022.**

[37] M. Montecelo, K. Merker, RISC-V - Debian Wiki. Retrieved July 5, 2022, from https://wiki.debian.org/RISC-V, **2021**.

[38]鄭博升, “以矩陣乘法為基礎應用硬體加速器於一維卷積計算之研究”, *國立臺灣師範大學資訊工程研究所碩士論文*, **2022**.

[39]蔡佳諭, “基於 RISC-V 架構之脈動陣列一維卷積運算研究”, *國立臺灣師範大學資訊工程研究所碩士論文*, **2022**.

附錄

本論文附錄分為附錄一、附錄二及附錄三主要用於補充說明在本文第三章研究方法中的一些細節資料，並且對資料做詳細的說明，讓讀者能夠更加深入的了解本論文的流程及內容。

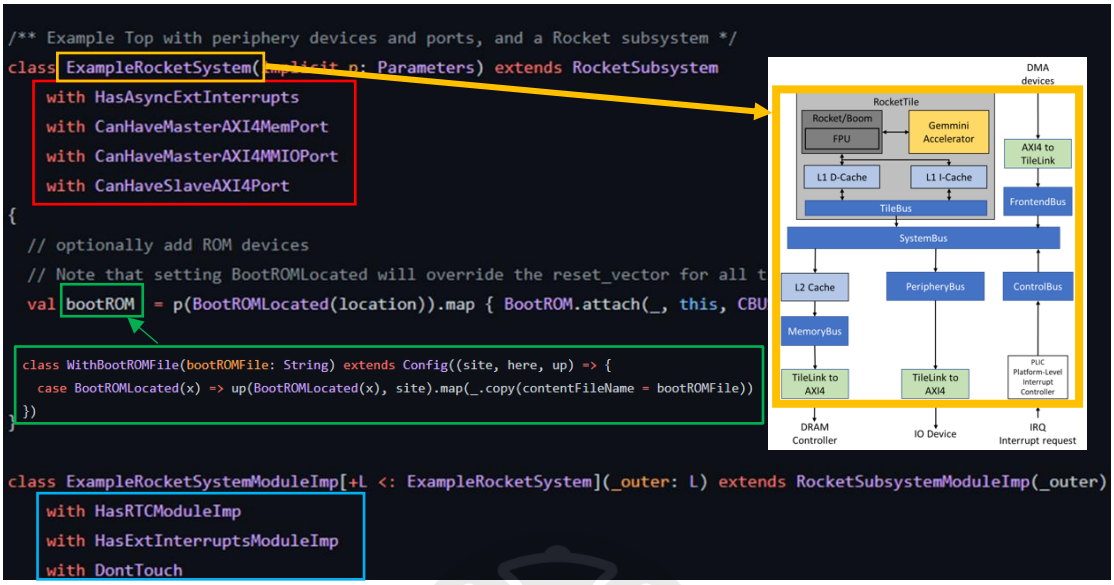
附錄一 Rocket Chip chiptop

附錄一的內容為詳細介紹本論文於 Rocket Chip chiptop 所使用的 Bus 接口及模組等，並且對應至 Rocket Chip 架構中的位置，在本文中所對應的章節為第三章第一節。

附圖 1 紅框部分可以看到有四個外部接口:HasAsyncExtInterrupts 是允許有外部非同步的 interrupt request(IRQ)傳進 Rocket Chip 的 interrupt 接口，CanHaveMasterAXI4MemPort 為在系統上加入一個 AXI4[28]協議的記憶體接口，通過這個接口去控制 DRAM Controller，而 CanHaveMasterAXI4MMIOPort 以及 CanHaveSlaveAXI4Port 為通過 AXI4 協議與 Memory Map I/O 互相溝通，橘框為 ExampleRocketSystem 整體對應圖 2-4 的架構，可以看到除外部設備外，都被包含在 ExampleRocketSystem 底下，所以橘框部份為硬體頂層設計。

而附圖 1 中綠框部份我們可以通過 WithBootRomFile 去將 bootloader 放進 CPU 的 ROM 裡面，詳細說明會於 3-3 做說明。最後藍框部份 HasRTCModuleImp 為加入一個 Real-time Clock 的模組，

HasExtInterruptsModuleImp 則是通過模組去轉換進入到 Rocket Chip 的 interrupt 訊息型態，DontTouch 就是這邊的接口不能做更動及使用。



附圖 1　Rocket Chip chiptop

附錄二 編譯 U-Boot 及 Linux Kernel

附錄二為編譯 U-Boot 使其與作業系統做連結以及將本論文使用到的 driver 加入至 linux kernel 後重新編譯，在本文中所對應的章節為第三章第二節。

在下圖中紅框部分為跟 linux kernel 做連結並且啟動，藍框部分為將開發板接口資訊加入 U-Boot，黃框部分為編譯 U-Boot 產生 binary 檔供 OpenSBI 使用。

```
u-boot/configs/vivado_riscv64_defconfig: patches/u-boot/vivado_riscv64_defconfig Makefile
	cp patches/u-boot/vivado_riscv64_defconfig u-boot/configs
ifeq ($(ROOTFS),NFS)
	echo 'CONFIG_USE_BOOTARGS=y' >>u-boot/configs/vivado_riscv64_defconfig
	echo 'CONFIG_BOOTCOMMAND="booti $${kernel_addr_r} - $${fdt_addr}"' >>u-boot/configs/vivado_riscv64_defconfig
	echo 'CONFIG_BOOTARGS="root=/dev/nfs rootfstype=nfs rw nfsroot='$(ROOTFS_URL)',nolock,vers=4,tcp ip=dhcp earlycon console=ttyAU0,115200n8 locale.LANG=en_US.UTF-8"' >>u
else ifeq ($(JTAG_BOOT),1)
	echo 'CONFIG_USE_BOOTARGS=y' >>u-boot/configs/vivado_riscv64_defconfig
	echo 'CONFIG_BOOTCOMMAND="booti $${kernel_addr_r} $${ramdisk_addr_r} $${fdt_addr}"' >>u-boot/configs/vivado_riscv64_defconfig
	echo 'CONFIG_BOOTARGS="ro root=UUID=68d82fa1-1bb5-435f-a5e3-862176586eec earlycon initramfs.runsize=24M locale.LANG=en_US.UTF-8"' >>u-boot/configs/vivado_riscv64_defco
endif

u-boot-patch: u-boot/configs/vivado_riscv64_defconfig
	if [ -s patches/u-boot.patch ] ; then cd u-boot && ( git apply -R --check ../patches/u-boot.patch 2>/dev/null || git apply ../patches/u-boot.patch ) ; fi
	cp -p -r patches/u-boot/vivado_riscv64 u-boot/board/xilinx
	cp -p patches/u-boot/vivado_riscv64.h u-boot/include/configs

u-boot/u-boot-nodtb.bin: u-boot-patch $(U_BOOT_SRC)
	make -C u-boot CROSS_COMPILE=$(CROSS_COMPILE_LINUX) BOARD=vivado_riscv64 vivado_riscv64_config
	make -C u-boot \
	  BOARD=vivado_riscv64 \
	  CC=$(CROSS_COMPILE_LINUX)gcc-8 \
	  CROSS_COMPILE=$(CROSS_COMPILE_LINUX) \
	  KCFLAGS='-O1 -gno-column-info' \
	  all
```

附圖 2 U-Boot 編譯部分程式碼[36]

下圖紅框中我們將 Driver 複製到 kernel 裡面，並使用藍框中指令重新編譯 linux kernel。

```
# --- build Linux kernel ---

.PHONY: linux linux-patch

linux: linux-stable/arch/riscv/boot/Image

CROSS_COMPILE_LINUX = /usr/bin/riscv64-linux-gnu-

linux-patch: patches/linux.patch patches/fpga-axi-sdc.c patches/fpga-axi-eth.c patches/linux.config
	if [ -s patches/linux.patch ] ; then cd linux-stable && ( git apply -R --check ../patches/linux.patch 2>/dev/null || git apply ../patches/linux.patch ) ; fi
	cp -p patches/fpga-axi-eth.c  linux-stable/drivers/net/ethernet
	cp -p patches/fpga-axi-sdc.c  linux-stable/drivers/mmc/host
	cp -p patches/fpga-axi-uart.c linux-stable/drivers/tty/serial
	cp -p patches/linux.config linux-stable/.config

linux-stable/arch/riscv/boot/Image: linux-patch
	make -C linux-stable ARCH=riscv CROSS_COMPILE=$(CROSS_COMPILE_LINUX) oldconfig
	make -C linux-stable ARCH=riscv CROSS_COMPILE=$(CROSS_COMPILE_LINUX) all
```

附圖 3 重新編譯 linux kernel [36]

附錄三 SD 卡燒錄

附錄三為將作業系統以及 RISC-V 指令集開機所需的 Bootloader 分別通過掛載的方式燒錄製 SD 卡上，在本文中所對應的章節為第三章第二節。

首先在紅框部分指令為開兩個資料夾並且分別掛載到 SD 卡上的 BOOT 分區以及 rootfs 分區，到藍框的部分後將開機需要使用的檔案推送至 BOOT 分區，Debian root file system 推送至 rootfs 分區，最後黃框部分設定 rootfs 的權限。

```
mkdir -p debian-riscv64/boot
mkdir -p debian-riscv64/rootfs

sudo mount ${SD_LOOP}p1 debian-riscv64/boot
sudo mount ${SD_LOOP}p2 debian-riscv64/rootfs

pushd debian-riscv64/rootfs
  if [ -z "$SD_SWAP_SIZE" ] ; then
    sudo tar xzf ../rootfs.tar.gz
  else
    sudo tar --exclude=swapfile -xzf ../rootfs.tar.gz
    sudo fallocate -l ${SD_SWAP_SIZE}M swapfile
    sudo chmod 600 swapfile
    sudo mkswap swapfile
  fi
popd

pushd debian-riscv64/boot
  sudo mkdir extlinux
  sudo cp ../extlinux.conf extlinux
  sudo cp ../initrd extlinux/initrd-$KERNEL_VER.img
  sudo cp ../../linux-stable/arch/riscv/boot/Image extlinux/image-$KERNEL_VER
  sudo cp ../../workspace/boot.elf boot.elf
popd

sudo chown root:root debian-riscv64/rootfs
sudo chmod 755 debian-riscv64/rootfs
```

附圖 4 SD 卡燒錄