Final Project Report

MCU System with CNN Accelerator

楊子賡, DCS031, 111550076

***Abstract*—本次作業實作了一個整合 MCU、CNN 加速器與 DMA 控制器的 Domain-Specific Accelerator。透過專屬的 CNN 指令觸發後，CNN 模組利用 DMA 控制器，透過 AXI 協定從 DRAM 擷取 input feature、權重與激活函數模式，並執行運算，最後將結果回傳給 MCU。**

MCU 採用 5-stage pipelined 處理器架構，並設計了 Instruction Cache 以提升指令擷取效率。在 6.1 ns 的 clock period 下，本系統以 562,826 的電路面積完成範例測資，於 gate level simulation 所需時脈數為 123,941 cycles。

***Index Terms*—Floating Point IP, CNN, MMIO, RISC-V, Aquila**

I. Introduction

本作業實作了一套整合 MCU、CNN 加速器與 DMA 控制器的 Domain-Specific Accelerator 系統，旨在提升嵌入式平台於深度學習推論任務中的運算效能。系統架構中，MCU 負責指令執行、任務分派與整體控制，CNN 模組透過專屬指令觸發後啟動計算流程，並由 DMA 控制器依據 AXI 協定從 DRAM 擷取所需的 input feature 、weight 與 activation function mode。Fig. 1展示了系統的 top-level 架構圖，概述各模組間的資料與控制流程。

從圖中可見，MCU 搭載了 Instruction Cache，用以緩存指令資料，當發生 cache miss 時，則透過 AXI bus 以 AXI burst 模式向 Instruction DRAM 發送請求。而在資料存取方面，系統並未配置 Data Cache，MCU 需直接透過 AXI bus 與 Data DRAM 進行讀寫操作。由於 CNN 加速器中的 DMA 控制器與 MCU 的記憶體存取單元會同時競爭對 Data DRAM 的存取權，而系統僅提供單一組 AXI 接口連接至該記憶體，因此需設計 Arbiter 模組以調度兩者的請求並避免衝突。整體系統採模組化設計，具備良好的可擴充性與可維護性，便於未來的功能增強與效能調整。

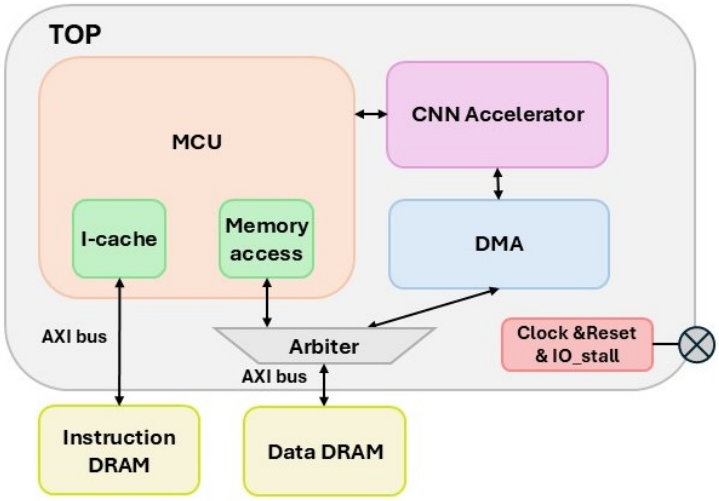

Fig. 1: CNN Accelerator System Architecture

II. Implementation of 5-stage Pipelined CPU

本次作業所使用的 CPU 參考自我在計算機組織課程中實作過的 MIPS 5-stage pipelined CPU，具備 forward unit 與 load-use stall 的處理邏輯，採用標準的五級流水線架構，並針對本次作業指定的指令架構進行相應的調整與擴充，包括加入 Instruction Cache 的改良設計。Fig. 2 為處理器的 microarchitecture，以下將逐一說明各階段、pipeline 與功能單元的設計細節。

A. The Instruction Fetch (IF) Stage

在 IF 階段，透過 AXI protocol 從 Instruction DRAM 取出指令，並傳遞至 Decode 階段。Instruction Address 由 Program Counter 控制，當指令依序執行時，Program Counter 將自動遞增；遇到 branch 或 jump 指令時，Program Counter 則會被更新為目標位址，進而繼續進行指令提取。

我將 Instruction Cache 的 block size 設為 128-bit（即八筆指令），採用 direct-mapped 的方式實作，並設計為擁有兩個 entry。此外，entry 數量與整體 cache size 均為 configurable，可根據需求彈性調整 cache 參數。Fig. 3為 Instruction Cache Controller 的示意圖。當發生 cache miss 時，系統會利用 AXI burst 一次向 Instruction DRAM 提取八筆指令，期間 pipeline 將進入 stall 狀態。

針對 cache block size 的設計，我觀察到此次指令中經常在八筆指令以內即發生 branch 行為，因此若設定過大的 cache size，可能導致不必要的指令提取浪費。若發生 conflict miss，亦將對效能造成負面影響，並增加額外的指令擷取時間。此外，觀察多數開源的 RISC-V 32-bit CPU，其 Instruction Cache 設計多為 128-bit 或 256-bit（對應四筆或八筆指令），此設定在實務中能在提取效率與複雜度之間取得平衡，尤其在處理較為複雜的指令流程時，能有效發揮效能。

B. The Instruction Decode (ID) Stage

在 ID 階段中，將獲取的指令根據 spec 規則進行編碼，產生不同的指令類型控制訊號，並傳遞至 Execution 階段。除了指令解碼之外，也要從 ID 階段中的 register file 讀取暫存器中的數值。

同時在這個階段也會將 jump 信號及 jump address 傳至 Program Counter Unit，讓 PC 跳到指定的位置，由於這個步驟是在 decode stage 做，所以本系統中的 jump 指令會產生一個 bubble。

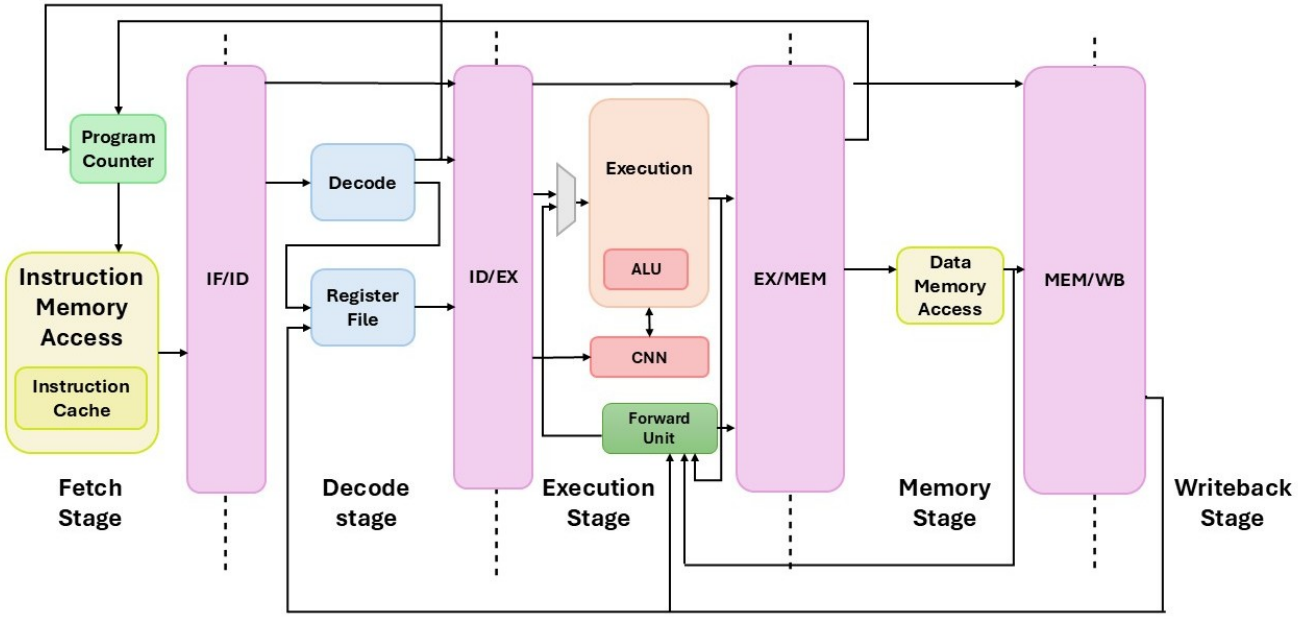

Fig. 2: pipelined cpu microarchitecture

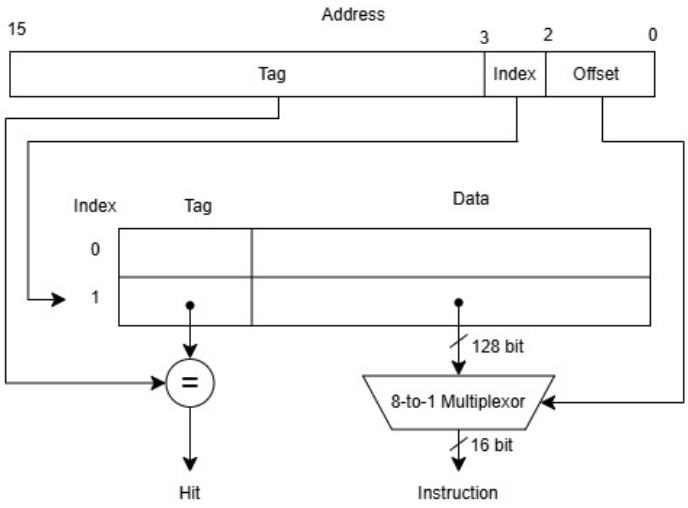

Fig. 3: Instruction Cache 示意圖

另外 load use hazard 的偵測也在這個 stage 執行，當傳給 execution stage 的 rd 和 fetch stage 傳過來的 rs 或 rt 相同，且傳給 execution stage 的指令是 load、fetch stage 傳過來的指令會使用 rs 或 rd，就會觸發 load use hazard，這時就會讓 pc、fetch stage 進行 stall，並產生一個 bubble。

C. The Execution (EXE) Stage

EXE 階段會執行 add, sub, mult 指令以及 beq 指令的 target address，另外也會運算 load, store 指令的 target address。

此階段使用一個簡化的 ALU，根據不同的指令類型選擇適當的兩個運算元及對應的運算操作。運算元來源可能為 PC、immediate 或 register file 中的資料，而運算操作則包含加法、減法與乘法。最終的運算結果會傳遞至 Memory 階段。

當遇到 CNN 指令時，EXE 階段亦負責偵測並進行 stall，直到 CNN 加速單元完成運算，再將結果傳遞至後續階段。

D. The memory (MEM) Stage

在 MEM 階段，處理器負責執行 Load 與 Store 指令，透過 AXI protocol 與 Data DRAM 進行資料的讀寫操作。本設計的 AXI 操作邏輯主要參考作業四的實作方式。

由於指令執行模式中會定期檢查 DRAM 的資料內容，因此無法實作 Write-Back Cache；雖然可以考慮 Write-Through Cache，但評估後發現效益有限，且觀察指令存取行為後並未顯示明顯的空間或時間區域性（locality），因此本階段並未設計 Data Cache。

在等待 DRAM 操作完成期間，整體 pipeline 將進入 stall 狀態，以確保資料存取的正確性與操作的同步性。當資料成功從 DRAM 讀出後，MEM 階段會將該資料，連同來自 EXE 階段的 ALU 運算結果、register destination 以及相關控制信號，一併傳遞至 Writeback 階段，寫入目標暫存器。

E. The Write Back (WB) Stage

在 Write Back 階段，處理器將執行結果寫回至 register file。根據控制信號判斷資料來源，可能為 MEM 階段讀出的資料，亦可能為 EXE 階段產生的 ALU 運算結果。系統會依據資料路徑與控制信號，將正確的資料寫入指定的目的暫存器，完成整條指令的執行流程。

F. Bypass & Hazard Detection

由於指令間常存在 data dependency，本設計針對 ALU 的兩個運算元與 Store 指令的資料皆實作了 bypass 機制。當需要的運算元尚未寫回暫存器、而位於 Memory 階段或 Write Back 階段時，系統會判斷其來源並從對應階段直接轉送資料，以避免不必要的 stall。

唯一需要引入 data hazard stall 的是 Load-Use Hazard，即當一條 Load 指令後立即跟隨使用該資料的指令。此情況會在 Decode 階段即被偵測，並且發生 stall 以延遲該指令執行，確保資料已載入完成。

透過這些 bypass 與 hazard detection 機制，可確保整體運算過程的正確性與管線流暢度。

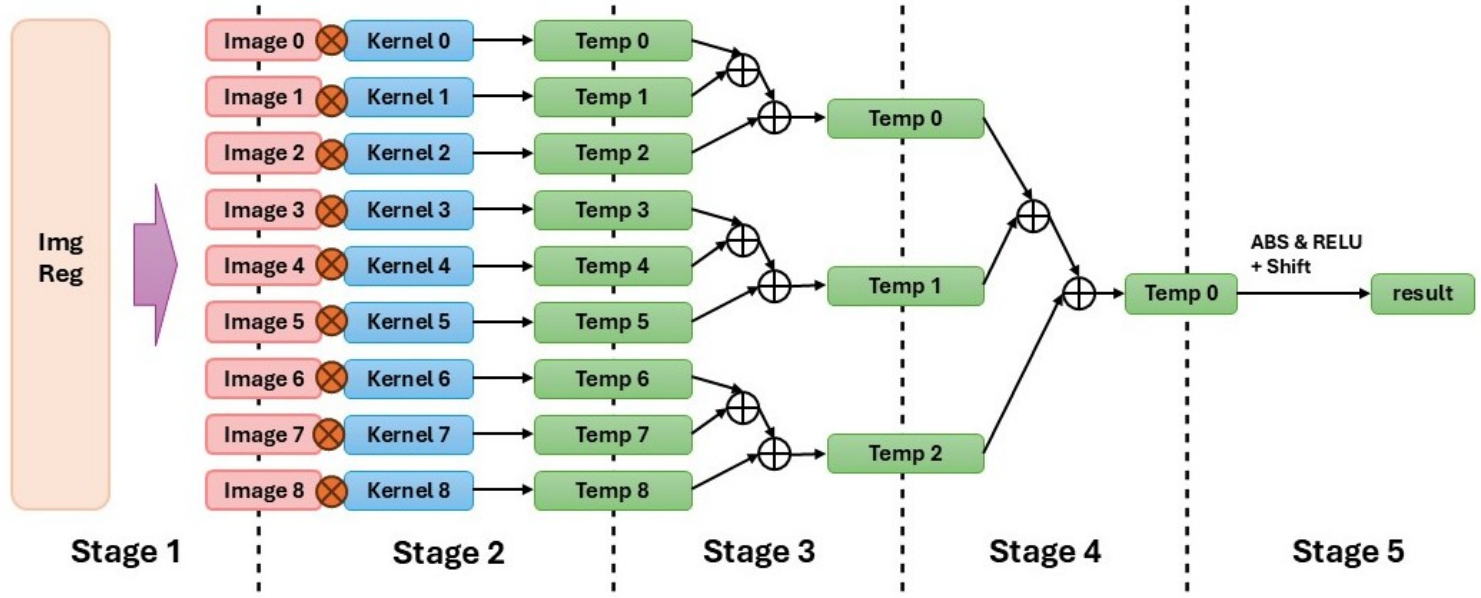

Fig. 4: Convolution Layer Pipeline Diagram

III. CNN Accelerator Design

本次設計中，所有計算單元皆採用 pipelined 架構，目的在於將 critical path 縮短至單一乘法延遲，並維持計算資源的高利用率。透過管線化設計，不僅提升整體效能，也有效降低了單一 cycle 的延遲時間。

A. DMA Design

當接收到 CNN 指令後，需自 Data DRAM 中提取所需資料。為了充分發揮 AXI burst 傳輸的效率，實作時按照 kernel、weight 與兩張 image 的順序進行資料讀取。首先，使用 burst length 為 17 的 AXI burst 讀取兩組 kernel 資料並暫存於 register；接著，讀取 Fully Connected Layer 所需的 weight，採用 burst length 為 31 的設定，一次讀取 32 筆資料，同樣存入 register 中；最後依序讀取 imgA 與 imgB，每張圖的讀取使用 burst length 為 71。為提升效率，於 image 尚未完全讀取完成時，即可啟動後續的運算流程。

B. Convolution Layer

本層採用五級 pipeline 架構以實現高效捲積運算。Fig. 4為管線示意圖，每個 stage 使用 valid register 維護，當 valid 信號傳至下一個 stage 時，就會進行該階段的運算。Stage 1 根據 index 計算欲取出的 image 上 3×3 patch 的九個像素值；Stage 2 將九個像素分別與 kernel 對應相乘，實現九組乘法的平行運算；Stage 3 將上述九筆乘積分為三組進行加總，產出三筆中間結果；Stage 4 再進一步將三筆中間結果進行加總；Stage 5 根據 activation mode 執行 Activation 函數（支援取絕對值與 ReLU），並在同一階段執行 quantization（右移捨去九個 bit）。最終結果將儲存於暫存器供後續 Pooling Layer 使用。

C. Pooling Layer

當 Convolution Layer 完成後，立即進行 Max Pooling 操作。在一個 clock cycle 內，完成四組 2×2 window 的最大值擷取，並將這四筆資料儲存並傳遞至 Fully Connected Layer。

D. Fully Connected Layer

本層同樣採用 pipeline 架構設計。由於 Convolution Layer 結束後仍需等待資料存取，因此設計為一次處理四筆輸入資料。在第一輪中，這四筆輸入與 weight 的前四筆進行乘法計算；第二輪則與後四筆 weight 計算。Stage 1 完成四個輸入與八個 weight 的乘法，共產生 16 筆資料；Stage 2 進行兩兩相加，產生 8 筆輸出；Stage 3 再次兩兩相加，得 4 筆結果。將兩輪產生的四筆結果加總後，即為最終的四筆 feature map。最後進行 argmax 運算以找出最大值對應的 index，輸出至 Execution Stage，並同時解除 Execution Stall，完成 CNN 指令運算。

TABLE I: stall overhead 分析

	cycle	ratio
CNN stall	781	0.63%
load use stall	184	0.15%
instr fetch stall	10814	8.73%
data fetch stall	109452	88.31%
total stall cycle	119334	96.28%
total cycle	123941	100%

IV. Evaluation

A. Overhead Analysis

TABLE I 顯示範例測資中的 stall cycle 分析結果，其中 stall cycle 佔總運行時間高達 96.28%，比例相當可觀。雖然四種類型的 stall 可能會部分重疊，但仍可從整體比例判斷：效能影響最大的為 data fetch stall。此類 stall 的主因在於資料缺乏 locality，且因 Pattern 檢查規則限制，無法採用 write-back cache 策略，導致 memory access overhead 難以優化，成為系統的主要瓶頸。

至於 CNN 運算部分，範例測資中僅執行一次 CNN 指令，約耗時 781 cycles，與整體系統執行時間相比影響極小。另一方面，load-use stall 僅發生 184 cycles，一樣沒有造成很大的效能影響。

B. Discussion on Out-of-Order CNN Operation

在本次作業的設計條件下，不適合採用 Out-of-Order 的 CNN 執行策略，原因如下：

首先，後續指令會不斷更新 Data Memory 的內容，可能覆寫掉 CNN 指令需要存取的資料，因此在缺乏嚴格記憶體一致性保障的情況下，Out-of-Order 架構無法確保資料正確性。

其次，CNN 的 DMA 模組與 MCU 的 memory stage 共用同一組 Data AXI bus。當後續指令持續進行 store 操作時，若 CNN 運算同時向 DRAM 發起大量資料請求，將進一步造成 MCU memory fetch 的額外 stall，進而拖累整體效能。因此，考量本作業的指令執行 pattern 與資源共享架構，並不適合導入 Out-of-Order CNN 執行機制。

TABLE II: Instruction Cache 對效能影響的分析

	no instruction cache	2-entry cache	4-entry cache
instr fetch stall	722640 (97.25%)	10814 (8.73%)	10814 (8.73%)
data fetch stall	109452 (14.73%)	109452 (88.31%)	109452 (88.31%)
total stall	738501 (99.38%)	119334 (96.28%)	119334 (96.28%)
total cycle	743108 (100%)	123941 (100%)	123941 (100%)
cell area	543071	562826	584512
cycle time	6.1	6.1	6.1

C. Performance Improvement from Instruction Cache

TABLE II 比較了無 Instruction Cache、2-entry 與 4-entry Cache 設計下的效能差異，分析項目包含 stall cycle、cell area 與 cycle time。結果顯示，加入 Instruction Cache 雖增加了電路面積，但 instruction stall cycle 數下降了 66.82 倍，整體執行 cycle 數亦減少約 6 倍，系統的主要效能瓶頸由 instruction fetch 轉移至 data fetch，可見 Instruction Cache 對整體效能具明顯提升效果，是一項值得投入的資源。

進一步比較 2-entry 與 4-entry 設計，在範例測資中兩者的 stall cycle 與總執行週期幾乎相同，而 4-entry 的面積較大，顯示對於本次範例測資而言，2-entry Cache 已具備足夠效能，不需浪費額外面積增加至 4 entry。

D. Resource Utilization

合成結果顯示，目前僅保留 Convolution Layer 為獨立模組，無法分析其他階段之資源使用。然而 Convolution Layer 合成後的面積佔整體面積達 39.7%，顯示其為資源最密集的部分。若能進一步壓縮此模組的電路面積，將對整體系統之面積與功耗具有顯著優化潛力。

V. Conclusion

本專題實作了一個整合 MCU、CNN 加速器與 DMA 控制器的嵌入式系統，具備完整的 pipeline 架構與指令快取機制。實驗結果顯示，Instruction Cache 大幅降低了 stall cycle 並提升整體效能，CNN 模組亦成功完成平行化加速設計。

雖然系統在 data fetch 上仍有明顯瓶頸，但本設計已提供一個高效、模組化且具可擴展性的基礎平台，為未來在架構優化與模型泛化方面奠定良好基礎。

VI. Future Work

A. Scalability and CNN Optimization

本次作業中的 CNN 架構為固定設計，CNN Accelerator 亦採取針對特定網路進行最佳化的實作方式。若欲將本設計擴展為更具通用性的加速器，未來工作應聚焦於提升架構的可擴展性與可參數化能力，使其支援多層數、可變 kernel size 與不同層類型（如 depthwise separable、residual 等）之 CNN 模型。

此外，現階段的實作以效能為優先，採用高度平行化的運算方式。未來可考慮改採 FMA（Fused Multiply-Add）指令結構來進行運算，不僅可減少邏輯資源使用，也有望降低面積與 cycle time，提升整體設計的面積效能比。

B. MCU Architecture Optimization

本作業中設計的特殊 ISA 與傳統 RISC 架構在 load/store address 計算方式上有所差異：其記憶體位址計算需先將 register value 與 immediate 相乘，再加上另一個 immediate。此設計導致 critical path 包含一個乘法與一個加法，難以進一步縮短 cycle time。

從合成結果觀察，critical path 主要集中於 execution stage。若能將 execution stage 拆分為兩個 pipeline stage，使乘法與加法分別分攤在不同 clock cycle 中執行，將有助於降低 critical path 延遲。由於此優化不涉及功能變更，預期對整體效能影響不大，卻能顯著改善 timing 收斂與時序可靠度。