國立台北大學資訊工程學系專題報告

A Hardware-Accelerated Post-Quantum Signature System on FPGA

專題組員:張皓宇、曾責晏

專題編號:PRJ-NTPUCSIE-114-005
執行期間:114年9月 至115年6月

1. 摘要

後量子密碼學(Post-Quantum Cryptography,PQC)是因應未來量子電腦可能破解傳統公開金鑰密碼系統而發展的重要技術。ML-DSA是 NIST 標準化的後量子數位簽章演算法之一, 可用於訊息完整性、來源認證與不可否認性。在 ML-DSA 的運算流程中, 多項式乘法是重要核心, 而 Number Theoretic Transform(NTT)與 Inverse Number Theoretic Transform(INTT)能有效提升多項式運算效率, 因此會在演算法中被重複執行。

本專題旨在製作一個應用ML-DSA 的 FPGA NTT / INTT 硬體加速系統, 並將加速器整合至 PQC library 中。系統透過 Linux kernel driver 與 ioctl() 介面, 使 library 內部能在不改變原本 API 的情況下呼叫 FPGA 執行核心運算。使用者仍可依照原本流程進行金鑰產生、簽章與驗章, 而底層 NTT / INTT計算則由硬體加速器負責完成。本專題完成 HLS 硬體設計、FPGA 實作、driver 控制與 PQC library 整合, 展示後量子密碼演算法於軟硬體協同設計中的應用可行性。

2. 簡介

隨著量子運算技術的發展, 傳統 RSA、ECC 等公開金鑰密碼系統未來可能受到量子演算法威脅, 因此能抵抗量子攻擊的後量子密碼逐漸受到重視。ML-DSA 是一種基於格密碼問題的後量子數位簽章演算法, 可用於確認訊息來源與防止資料遭到竄改。由於其具有抵抗量子攻擊的特性, 因此被視為未來資訊安全系統的重要技術之一。

在 ML-DSA 中, 多項式運算佔有相當重要的比例, 其中NTT與INTT是加速多項式乘法的核心方法。若所有 NTT / INTT 運算皆由 CPU 執行, 當系統需要大量簽章、驗章或同時處理多個任務時, 可能增加 CPU 的計算負擔。FPGA具有可客製化硬體架構與平行運算能力, 適合處理規則性高且重複執行的

運算, 因此本專題選擇將 ML-DSA 中的 NTT / INTT 運算卸載至 FPGA。

本專題目標是製作一個可整合至實際 ML-DSA 軟體流程的硬體加速系統, 而非僅完成獨立電路測試。系統由PQClibrary、Linux kernel driver 與 FPGA hardware 組成, 透過 ioctl() 傳遞資料與運算模式, 使硬體能依需求執行 NTT 或 INTT。藉由此設計, 上層應用程式不需改變原本 API, 即可使用硬體加速後的 ML-DSA 簽章與驗章功能。

3. 專題進行方式

3.1 成員配置與職責

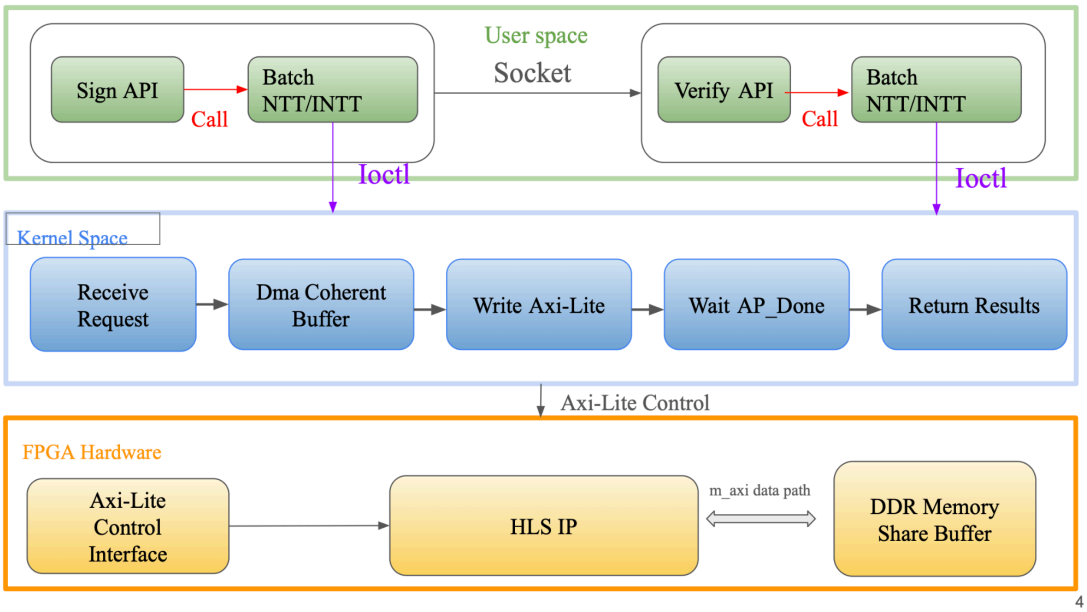
曾責晏:
整體系統架構設計、電路設計及優化
、驅動程式開發
張皓宇:
軟體應用建置、軟硬體系統整合實作、效能測試與評估

3.2 開發過程

一、開發階段

1. 基礎研究與軟體建置
   了解 ML-DSA, 完成 PQC library 與 Socket 測試。
2. 硬體加速架構設計
   規劃 NTT / INTT offload 與系統資料流程。
3. 硬體設計與優化
   使用 Vitis HLS 設計 NTT / INTT 加速器。
4. **Driver** 與 **Library** 整合
   透過 Linux driver 與 ioctl 控制 FPGA 硬體。
5. 整體測試與應用展示
   完成簽驗章測試與 Socket 端對端展示

二、系統整體架構



本專題的系統整體架構主要分為三個層次：User Space、Kernel Space 與 FPGA Hardware。User Space 負責執行上層應用程式與 PQC library，包含 ML-DSA 的金鑰產生、簽章與驗章流程；Kernel Space負責執行 Linux kernel driver，作為軟體與硬體之間的控制橋接層；FPGAHardware則負責執行 NTT / INTT 的實際硬體加速運算。

在原本的 ML-DSA 軟體流程中，KeyGen、Sign、Verify 會呼叫 PQC library 內部的多項式運算函式，其中 NTT / INTT 是多項式乘法的重要核心。本專題的設計重點，是將 library 中原本由 CPU 執行的 NTT / INTT 函式替換為 FPGA 硬體加速版本，但保留上層原本的 ML-DSA API。因此，使用者在應用程式中仍然以原本方式呼叫金鑰產生、簽章與驗章功能，不需要知道底層 NTT / INTT 已經改由 FPGA 執行。

整體資料流程如下：當 PQC library 內部需要執行 NTT 或 INTT 時，會先將多項式係數整理成一段連續的Buffer，並建立 Request structure。Request 中包含係數資料指標、欲處理的多項式數量，以及運算模式 mode。接著 User space 透過 ioctl() 將 Request 傳送給 Kernel driver。Driver 會將資料搬移至 DMA coherent buffer，透過 AXI-Lite 設定硬體控制暫存器，啟動 FPGA 硬體運算，並等待硬體完成。當硬體完成後，Driver 將結果複製回 User space，讓 PQC library 繼續後續的 ML-DSA 流程。

三、硬體設計與優化

硬體部分的主要目標，是設計一個可支援 NTT 與 INTT 的 FPGA 加速器，並能透過 Driver 被 PQC library 呼叫。由於 NTT / INTT 的演算法結構明確，主要由固定 Stage 與 Butterfly 運算組成，因此本專題使用 HLS 設計硬體電路，再將產生的 HLS IP 匯入 Vivado 進行 FPGA 系統整合。

本專題的硬體加速器透過 AXI-Lite 接收控制參數，包括 Buffer address、mode、num_polys 與 ap_start。Mode 用來決定本次硬體要執行 NTT 或 INTT，num_polys 則用來支援 Batch 運算，使硬體可以一次處理多個多項式。資料傳輸部分則透過 m_axi 介面存取 DDR，硬體從 DDR 讀取輸入係數，完成 NTT / INTT 計算後，再將結果寫回 DDR。

1.DDR Burst 與 Local Buffer

第一個硬體優化方向是記憶體 I/O。若硬體每次 butterfly 運算都直接從 DDR 讀取或寫入資料，會造成大量記憶體存取延遲，降低加速效果。因此本專題採用 DDR burst 搭配 Local buffer 的方式，先將一整個多項式的 256 個係數由 DDR 連續讀入 FPGA 內部的 BRAM local buffer，再於 Local buffer 中完成計算，最後將結果連續寫回 DDR。

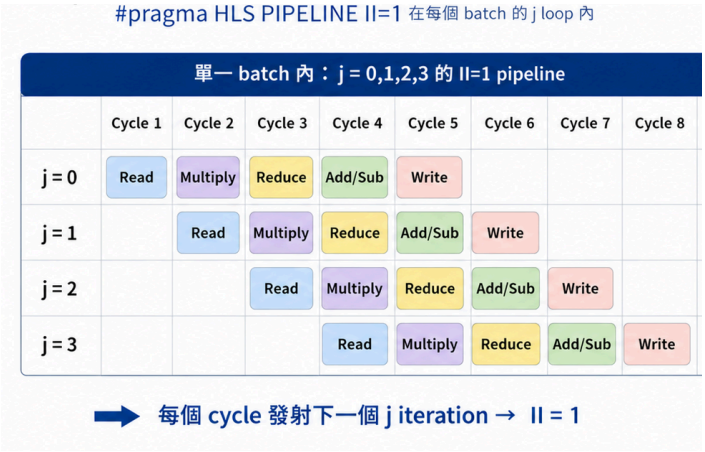
這種方式的優點是可以減少零散的 DDR 存取，將多筆連續資料合併成較有效率的 Burst transaction。對 NTT / INTT 這種資料長度固定、位址連續的演算法而言，Burst 傳輸能有效降低資料搬移的等待時間。

2. Pipeline 優化

第二個優化方向是 Pipeline。NTT / INTT 的 Butterfly 運算包含讀取資料、乘法、模數化簡、加減法與寫回等步驟。若每一次 Iteration 都必須等前一次完全完成後才能開始，整體執行時

間會很長。因此本專題在 Butterfly loop 中加入 Pipeline，希望讓不同 Iteration 可以重疊執行，使下一筆資料能盡快進入運算流程。

理想情況下，Pipeline 的 Initiation Interval 可以接近 1，也就是每個 Clock cycle 都能啟動下一次 Iteration。不過在實際 HLS 設計中，即使加入 #pragma HLS PIPELINE II=1，若存在 Memory dependency、記憶體 Port 不足或運算路徑過長，HLS 編譯器仍可能將 II 拉長。因此 Pipeline 優化不只是加入 pragma，而是必須配合 Buffer 架構、資料流與運算路徑一起調整。



3.Ping-Pong Buffer

第三個優化方向是 Ping-Pong buffer，此設計概念參考相關論文而來。NTT / INTT 在每一個 Stage 都會讀取前一階段的資料，並寫入更新後的結果。如果讀取與寫入都發生在同一個 Buffer 中，HLS 可能會判斷不同 Iteration 之間存在 Memory dependency，進而拉長 Pipeline II。

為了解決這個問題，本專題使用兩組 Buffer 交替作為讀取端與寫入端。例如某一個 Stage 從 Buffer 0 讀取資料時，將結果寫入 Buffer 1；下一個 Stage 則反過來從 Buffer 1 讀取，再寫回 Buffer 0。透過此方式，可以避免同一個 Buffer 在相近時間內同時承擔讀取與寫入，降低 Memory dependency 對 Pipeline 的影響。

四、**Driver** 設計與內部流程(圖一)

Driver 是本專題中連接 PQC library 與 FPGA 硬體加速器的關鍵模組。由於 User space 無法直接操作 FPGA 的實體暫存器，也無法直接管理適合硬體存取的實體記憶體，因此本專題透過 Linux kernel driver 負責硬體控制、資料搬移與結果回傳。

本專題的 Driver 以 Character device 形式實作，提供 /dev/ntt_hw 作為 User space 與 Kernel space 的溝通介面。PQC library 在需要執行硬體加速 NTT / INTT 時，會先開啟此裝置，再透過 ioctl() 傳送 Batch request 給 Driver。

Driver 的內部流程主要包含以下步驟。首先，Driver 從 User space 接收 Request structure，並檢查其中的參數是否正確，例如資料指標、num_polys 與 mode。接著，Driver 配置 DMA coherent buffer，將 User space 中的多項式係數複製到此 Buffer 中。使用 DMA coherent buffer 的原因，是讓 CPU 與 FPGA 可以存取同一段實體連續且快取一致的記憶體區域，避免資料同步錯誤。

完成資料準備後，Driver 透過 AXI-Lite 寫入 HLS IP 的控制暫存器，包括 buffer address、mode、num_polys 與 ap_start。當 ap_start 被寫入後，FPGA 硬體開始從 DDR 中讀取資料並執行 NTT / INTT。Driver 接著透過 Polling 方式等待 ap_done 訊號，確認硬體運算是否完成。當ap_done 被設為 1 後，代表硬體已完成計算並將結果寫回 DMA buffer。最後，Driver 將 DMA buffer 中的結果複製回 User space，並釋放相關資源，完成一次硬體加速呼叫。

在控制路徑方面，CPU 透過 AXI-Lite 寫入 Buffer address、mode、num_polys 與 ap_start，並讀取 ap_done 判斷硬體是否完成。在資料路徑方面，HLS IP 透過 m_axi 介面作為 AXI master，經由 AXI Interconnect 與 Zynq PS 的 HP port 存取 DDR memory，完成資料讀取與寫回。

4. 主要成果與評估

4.1 成果

(1)系統規模:

系統涵蓋軟體應用層、kernel　driver、硬體電路層，並做到三個層級的整合系統。

(2)測試成果:

- 上層軟體端的多項式資料能夠透果driver傳輸給硬體電路計算並將結果回傳至上層應用端，完成簽驗章流。
- 波形圖展示:
  波形圖中16進制的資料在valid和ready同時高位才有效
  圖(二):NTT讀取波形圖
  圖(三):NTT寫回buffer
  圖(四):INTT讀取波形圖
  圖(五):INTT寫回buffer

(3)效能評估與分析:

純NTT/INTT計算:

CPU計算時間:

```
=========  CPU 純軟體效能測試 (跑 10000 次)
CPU 正向 NTT 總耗時 : 323924.65 us
 平均單次 CPU NTT 耗時: 32.392 us

CPU 逆向 INTT 總耗時 : 379527.99 us
 平均單次 CPU INTT 耗時: 37.953 us
```

CPU時脈:625MHz

NTT: 21666 cpu cycles

INTT: 25266 cpu cycles

硬體電路計算時間(從軟體端送資料開始計時至資料回到上層):

```
===== 執行 Batch NTT 運算 (批次量: 4) =====
測試次數: 10000 次
總共耗時: 4076423.94 us (微秒)
平均單次 NTT 耗時: 101.911 us (微秒)
===== 執行 INTT 運算 (還原) =====
測試次數: 10000 次
總共耗時: 4099604.58 us (微秒)
平均單次 INTT 耗時: 102.490 us (微秒)
```

另外測量在電路內部純計算不包含資料傳回上層的時間:
FPGA core time =15.87us
FPGA 時脈:125MHZ
NTT clock cycle : 1984
INTT clock cycle : 2046

分析:
1.優化差異比較:

- 有pipeline :

```
===== 執行 Batch NTT 運算 (批次量: 4) =====
測試次數: 10000 次
總共耗時: 4076423.94 us (微秒)
平均單次 NTT 耗時: 101.911 us (微秒)
===== 執行 INTT 運算 (還原) =====
測試次數: 10000 次
總共耗時: 4099604.58 us (微秒)
平均單次 INTT 耗時: 102.490 us (微秒)
```

- 無pipeline :

```
xilinx@pynq:~/project/driver$ sudo ./test
===== 執行 Batch NTT 運算 (批次量: 4) =====
測試次數: 10000 次
總共耗時: 13662606.63 us (微秒)
平均單次 NTT 耗時: 341.565 us (微秒)

===== 執行 INTT 運算 (還原) =====
測試次數: 10000 次
總共耗時: 15639616.23 us (微秒)
平均單次 INTT 耗時: 390.990 us (微秒)
```

2.效能瓶頸分析:
從測量的的時間可發現, 純電路的計算不包含從上層應用到Driver的完整流程。其實從資料送到電路到計算完成速度其實相較Cpu的計算快上一倍。但即便如此, 我們縮短的計算時間, 也遠不及Driver的時間開銷。當我們將完整上到下的計算時間扣掉純硬體電路計算時間可發現, 上層系統的時間開銷佔了整體84%的時間。為此未來的優化重點將會著重在Driver的部分。

4.2未來方向

1.將更多演算法放入硬體加速中
2.預先配置 Dma Coherent buffer。原因以我們Driver流程來說, 配置給硬體的Buffer是在每次需要計算時才配置, 且計算完後又會釋放掉。其實這對於時間的開銷會非常大, 且當更多演算法需要硬體加速時此做法會把這個缺點放大, 當大量計算同時需要配置Buffer時就會需要等待大量時間。而預先配置的想法, 是參考作業系統中Thread pool的概念, 省去動態建立與銷毀, 使資源可重複利用。
3.提高ML-DSA的安全層級
4.提高硬體電路處理資料的吞吐量

5. 結語與展望

5.1 總結

本專題完成了一個應用於 ML-DSA 後量子數位簽章演算法的 FPGA-based NTT / INTT 硬體加速器。

專題的成果不僅是完成單獨的硬體電路測試, 而是將 FPGA 加速器實際整合至 PQC library 中。透過 Linux kernel driver 與 ioctl() 介面, PQC library 能在內部呼叫硬體執行 NTT / INTT 運算, 同時保留原本 ML-DSA 的 API 使用方式。使用者仍可依照原本流程進行金鑰產生、簽章與驗章, 而底層核心運算則改由硬體加速器完成。

整體而言, 本專題完成了從演算法分析、HLS 硬體設計、FPGA 系統整合、Linux driver 控制, 到 PQC library 修改與端對端簽驗章測試的完整流程。此成果展示了 FPGA 應用於後量子密碼系統運算卸載的可行性, 也提供了一個軟硬體協同設計應用於實際密碼函式庫的實作案例。

5.2 未來展望

雖然本專題已完成 ML-DSA 中 NTT / INTT的 FPGA 硬體加速與 PQC library 整合, 但在系統效能與應用完整性上仍有進一步改善空間。首先, 目前整體執行時間仍受到 Software、Driver 與 Kernel system overhead 影響, 因此未來可針對 Driver 流程進行優化, 例如預先配置 DMA coherent buffer, 或建立類似 Buffer pool 的機制, 減少每次呼叫硬體時重複配置、釋放記憶體與資料複製所造成的成本。

其次, 未來可以進一步改善 Batch 處理方式。目前系統已能將多個多項式打包後一次送入 FPGA執行NTT/ INTT, 但仍可研究更大的批次量、更有效率的資料排列方式, 以及更完整的資料流設計, 使硬體加速器能在一次啟動後處理更多連續運算, 降低User space 與 Kernel space 之間頻繁切換造成的開銷。

在硬體設計方面, 未來可持續優化 NTT / INTT 電路, 例如改善 Memory access pattern、提高 Pipeline 效率、降低 Memory dependency, 或進一步調整 Ping-pong Buffer 與 Burst 傳輸方式, 以提升 FPGA core 的吞吐量。同時, 也可以評估硬體資源使用量與效能之間的平衡, 使設計更適合部署在資源有限的嵌入式 FPGA 平台上。

此外, 本專題目前主要針對 ML-DSA-44 進行

實作與驗證, 未來可延伸至更高安全層級的 ML-DSA, 例如 ML-DSA-65 或 ML-DSA-87, 觀察不同參數組對硬體資源、執行時間與系統整合成本的影響。除NTT/INTT外, 也可進一步分析 ML-DSA 中其他重複性高的運算, 評估是否能一併進行硬體加速, 以提升整體簽章與驗章流程的效能。

整體而言, 本專題已建立一個可運作的 FPGA-based PQC 加速系統。未來若能進一步降低系統Overhead、提升硬體利用率, 並支援更多安全層級與更多核心運算, 將能使此系統更接近實際應用需求, 也可作為後續發展後量子密碼硬體加速平台的基礎。

6. 銘謝

感謝指導教授於專題的製作過程中給予許多技術上的建議與幫助, 引導我們進行方向的同時, 保留足夠多的空間讓我們嘗試各種可能的方法, 使我們能夠順利完成本次專題研究。

7. 參考文獻

[1] 陈朝晖, 马原, 荆继武, “格密码关键运算模块的硬件实现优化与评估,” 北京大学学报（自然科学版）, vol. 57, no. 4, pp. 595–603, 2021. [Online]. Available: https://xbna.pku.edu.cn/fileup/0479-8023/HTML/2021-4-595.html. [Accessed: Jun. 7, 2026].

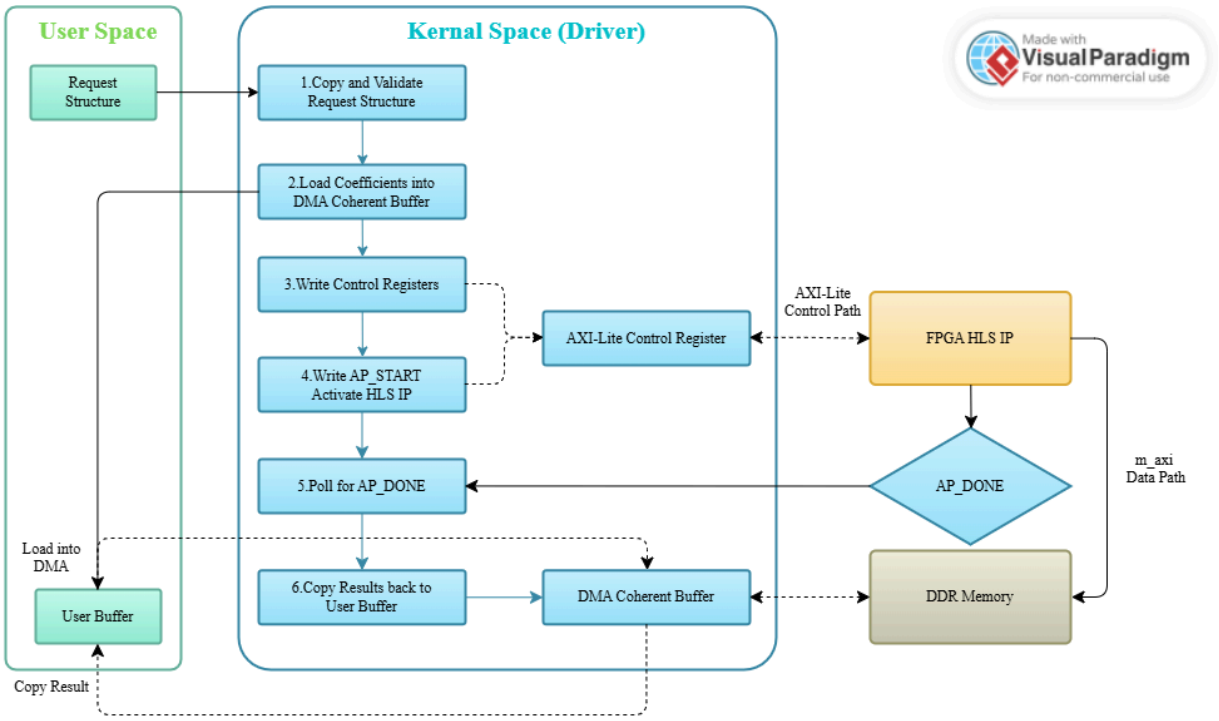
[2] Open Quantum Safe Project, “liboqs: Open source C library for quantum-safe cryptographic algorithms,” GitHub repository. [Online]. Available: https://github.com/open-quantum-safe/liboqs. [Accessed: Jun. 7, 2026].

[3] “數論轉換介紹：Number Theoretic Transform (NTT),” course slides, National Taiwan University, Apr. 9, 2025. [Online]. Available: https://djj.ee.ntu.edu.tw/ADSP_NTT_2025.pdf. [Accessed: Jun. 7, 2026].
參考 NTT 的基本定義、模運算、原根與 NTT / INTT 概念說明。

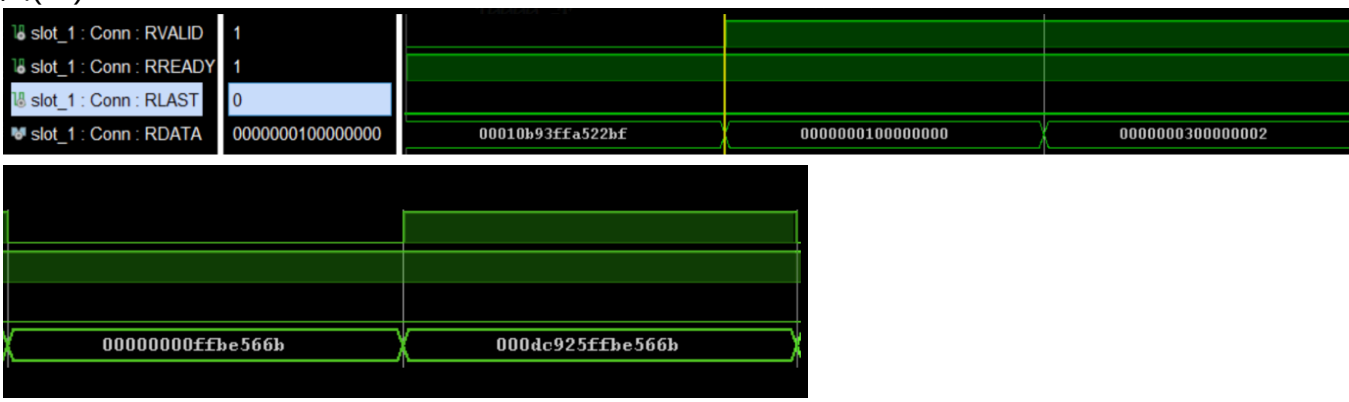
[4] AMD, “Specifying Arrays as Ping-Pong Buffers or FIFOs,” *Vitis High-Level Synthesis User Guide (UG1399)*, 2021.2 English, Dec. 15, 2021. [Online]. Available: https://docs.amd.com/r/2021.2-English/ug1399-vitis-hls/Specifying-Arrays-as-Ping-Pong-Buffers-or-FIFOs. [Accessed: Jun. 7, 2026].

本專題在 HLS 設計中參考 AMD Vitis HLS 官方文件, 將陣列資料通道設計為 ping-pong buffer 或 FIFO, 以改善資料搬移與運算階段之間的銜接。

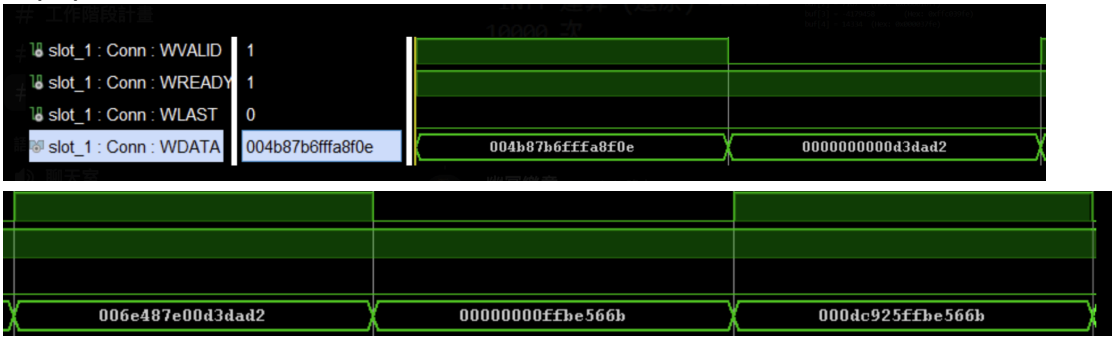
圖(一)


User Space
Kernal Space (Driver)
Request Structure
1.Copy and Validate Request Structure
2.Load Coefficients into DMA Coherent Buffer
3.Write Control Registers
4.Write AP_START Activate HLS IP
5.Poll for AP_DONE
6.Copy Results back to User Buffer
AXI-Lite Control Register
AXI-Lite Control Path
FPGA HLS IP
AP_DONE
m_axi Data Path
DDR Memory
DMA Coherent Buffer
Load into DMA
User Buffer
Copy Result
Made with Visual Paradigm
For non-commercial use

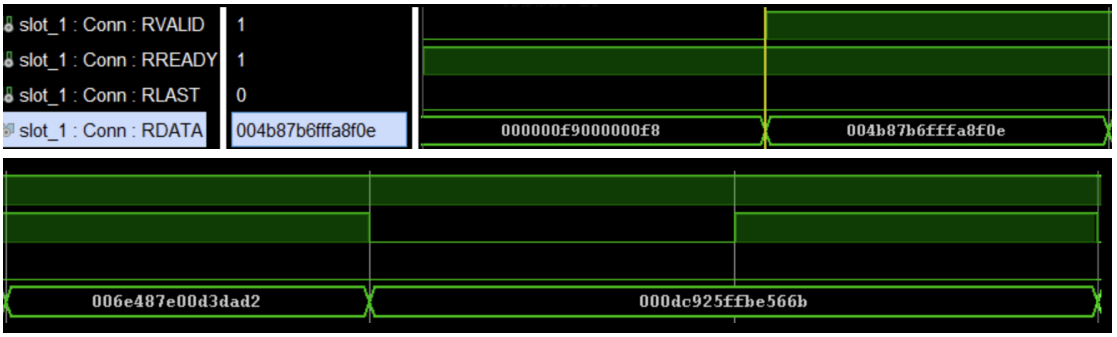
圖(二)


slot_1 : Conn : RVALID
slot_1 : Conn : RREADY
slot_1 : Conn : RLAST
slot_1 : Conn : RDATA
1
1
0
0000000100000000
00010b93ffa522bf
0000000100000000
0000000300000002
00000000ffbe566b
000dc925ffbe566b

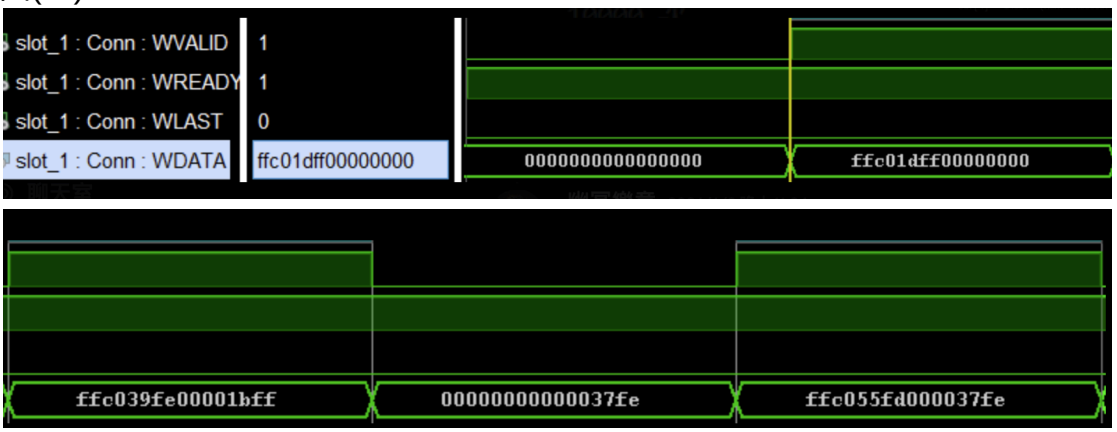
圖(三)


slot_1 : Conn : WVALID
slot_1 : Conn : WREADY
slot_1 : Conn : WLAST
slot_1 : Conn : WDATA
1
1
0
004b87b6fffa8f0e
004b87b6fffa8f0e
0000000000d3dad2
006e487e00d3dad2
00000000ffbe566b
000dc925ffbe566b

圖(四)


slot_1 : Conn : RVALID
slot_1 : Conn : RREADY
slot_1 : Conn : RLAST
slot_1 : Conn : RDATA
1
1
0
004b87b6fffa8f0e
000000f9000000f8
004b87b6fffa8f0e
006e487e00d3dad2
000dc925ffbe566b

圖(五)


slot_1 : Conn : WVALID
slot_1 : Conn : WREADY
slot_1 : Conn : WLAST
slot_1 : Conn : WDATA
1
1
0
ffc01dff00000000
0000000000000000
ffc01dff00000000
ffc039fe00001bff
00000000000037fe
ffc055fd000037fe