HW5 Report

謝翔丞, 109550025

***Abstract*—此次旨在於額外掛載一個 Floating Point IP 於 Aquila 上，嘗試利用硬體加速計算浮點數 inner-product 的過程，並分別以 8 elements 為實驗之 Input Vector 長度，最後經過驗證發現結果正確，並且相較於軟體實作有速度上的提升，展現更佳的運算表現。 (*Abstract*)**

Keywords—DAS;Inner-Product;Floating Point; (key words)

I. INTRODUCTION

本次作業根據需求利用 Xilinx 本身的 IP catalog 建立出 Floating Point 的 Inner-Product IP，並且撰寫 DataFeeder 為 Aquila 與自行 gen 出之 IP 做橋接，同時亦撰寫一份 c code for Inner-Product 的計算並編譯成 ELF file 交予硬體執行。因此後續報告會分成硬體方面予軟體方面進行整理，並在硬體方面會再分流為 DataFeeder 與自 gen 之 IP 兩者。

II. SOFTWARE

A. 整體流程

首先宣告兩個陣列，element 為另外使用 srand( ) 隨機產生好的值，接著進入 for loop 歷遍每個元素、計算兩元素乘積同時不斷做加總。而時間的計算則是包夾於該 for loop 的前後，如此才是最準確的”軟體 inner-product 計算時間”而沒有受到其他例如資料轉移之類的雜餘資訊誤導。

B. 初步想法

關於兩筆要輸入的資料陣列，我原先的計畫是利用 inline assembly，將兩個陣列的各個 element 輪流存到固定的兩個 register (eq. t0 , t1)，而硬體層面就可以輪流在每個 clock 讀取特定 register，藉此達到資料讀取的成效。但是這種作法其實相對麻煩，因為要將 register_file 的訊號層層往外拉到位於 Soc_top 的 DataFeeder，同時老師也表示，inline assembly 相對容易和 compiler 的優化產生衝突，我也遇到這個問題，因此最後捨棄這個做法改用下段提及的策略。

C. 最終版本

在請教老師以及助教過後，我採用直接給定位址的方法，將存放 data 的兩個陣列定址在 0xC200_0000 以後。之所以使用這個位置主要是因為，在觀察過 Soc_top.v 之後我發現，0xC000_0000 到 0xC0FF_FFFF 已經先被指定用於 uart device，而在 dsa_sel 的地方也已經宣告好 0xC200_0000 到 0xC2FF_FFFF。我也額外給定一個 int type 和一個 float type 的變數位址，前者目的是在 c code 內做完 data 的 memcpy 之後、要做 inner-product 之前拉起一個”資料傳輸完畢”的訊號; 後者是要給硬體部分儲存計算後的結果，用來和軟體結果做比對。

III. HARDWARE

承接 Introduction 提及，硬體方面我會以 Soc_top 為切入點，再依序介紹選用的 IP 設置以及 DataFeeder。

首先，我們可以將 Soc_top.v 中，Aquila_Soc 這個 module 視為主體，而在這份 .v 檔中其餘的 module 則可以視為掛載在 Aquila 外部的一些 device，例如原本的 uart 或是此次新增的 DSA。

A. Floating Point IP

這次我總共 gen 出兩種版本的 Floating Point IP，相同之處在於，我的設定是單個 IP 內可以完成乘法以及加法，透過這樣的 IP，只需要將每次的輸出做為下一次的 Data Stream C 輸入，並且第一次的 C data 為 0，那麼最後的 result_data 就會是所有 inner-product 的總和，一舉兩得!

相異之處在於，一個是 Blocking 另一個是 non-Blocking，我原先使用的是 non-blocking 版本，卻發現介面 port 口怎麼和講義有些許出入，後來才想到講義上的版本可能是 blocking，趕緊再 gen 第二個版本出來做修改。

1) Non-Blocking

Non-blocking 的 Floating Point IP 有幾個接口是使用者要自行接線的，分別是三個輸入 Data Stream 各自的 tvalid 和 tdata，還有輸出的 result_tvalid , result_tdata。

Input port 的 valid 是為了告訴 IP，接下來的 Data 是可以使用的。 而輸出的 Valid 是要通知使用者，也就是我們的 DataFeeder 說可以接收 result data 了。

2) Blocking

Blocking 和 Non-Blocking 的介面相差無幾，用法和用意也都相同，少數增加的是由 IP 傳給 DataFeeder 的三個 Input Data Stream 的 tready，和 DataFeeder 傳出的 tready 是一組的，用來做 handshake 確認雙方已經進入狀態。同樣的 DataFeeder 也要傳 result_tready 給 IP 做狀態確認。

B. DataFeeder

DataFeeder 的設計是此次作業的重點所在，一開始我並不清楚 MMIO 的使用和做法，因此起初我的實作是承接 II.B 提到的寫入特定 register，再將 Reg_File 的相關訊號由 Core_top 往外拉兩層到 Soc_top。但這樣的作法不但耗費工夫，同時也容易遇到前面提及的和 Compiler Optimize 衝突的狀況，因此轉而使用教授和助教建議的，利用 pointer 指向特定的位址來存取 value。

但在這樣的方向下，我遇到另一個問題，就是如何拿取軟體方存取好的內容?

原先我的想法是，將 memory.v 的訊號拉到 Soc_top，並透過類似已 address 為 index 的方式直接指定特定 address 來拿取數值，但是這樣實作的過程相當困難且怪異。

因此在請教同學後，我發現可以使用最外層 dev_addr 和 dev_data 的搭配，藉由判斷 addr 值來決定是否拿取該次 data，而這部分就是老師之所以要我們參考 Clint.v 的介面藉此以實作出 MMIO 的精隨所在，也遠遠勝於前段提及的土法煉鋼的作法。

這邊順便提及我在中後期曾經遇過 elf 在執行過中卡死的問題，後來發現是 dsa_dout 以及 dsa_ready 訊號沒有拉好，導致 Soc 及 Aquila 無法判斷，因此整個系統懸住，最後是透過重新理一次自己的整個 flow 流程圖，重新分配訊號線的拉取才解決。

以下開始介紹 DataFeeder 的各個實作細節:

1) DataBuffer

關於 DataBuffer，Input 部分我是建立兩個[32 – 1 : 0] * [VectorLength – 1 : 0] 的 register，用來暫存從 dev_din 獲取到的 a,b Input Data Stream，每筆都是 32bit，各自總共 VectorLength 筆。

Output 部分則只需要建立一個 [32-1 : 0] 的 register，之所以只要一維是因為每次從 IP 拿到的都是單個數值，且最後答案正是要加總後的值。

2) Data Output

對於資料何時要傳出給 IP 非常重要，這裡我採用的方式是:

首先我在軟體面就有另外設立一個變數，在所有 element 都隨著 clk 輪流給到預定地位址後將 send_done_flag 訊號拉起，硬體面讀到這個訊號為 1 時就知道 Input Data Buffer 已經被填滿。

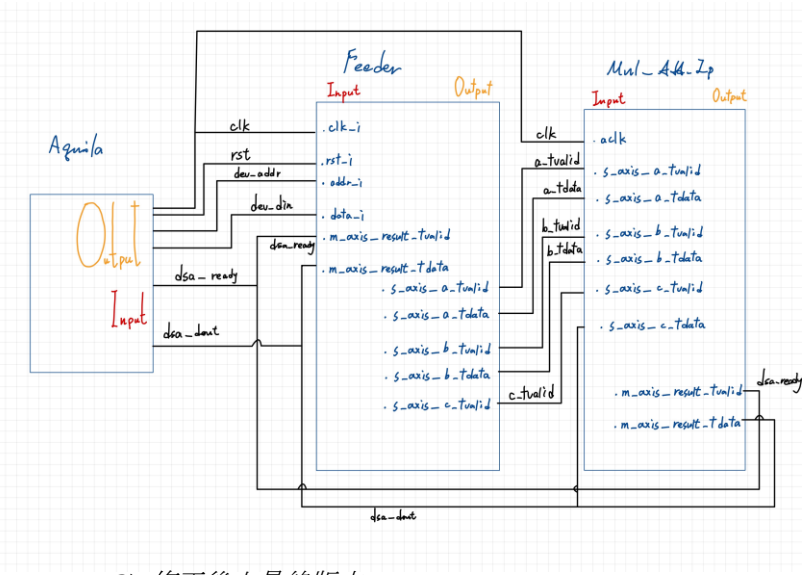
單靠”資料已填滿”的訊號並不足以驅使我們將資料傳入 Inner Product IP，這時還得將要傳到 IP 的 valid 訊號拉起，並且由 IP 傳回 DataFeeder 的 ready 訊號，在完成 handshake，確保雙方進入狀態後才能進行資料的傳遞。那麼，要在何時將 valid 拉起就顯得至關重要，我利用以下這個 counter 的計算搭配其他訊號實現 Valid 的操控。

a) Give_to_Ip_cnt

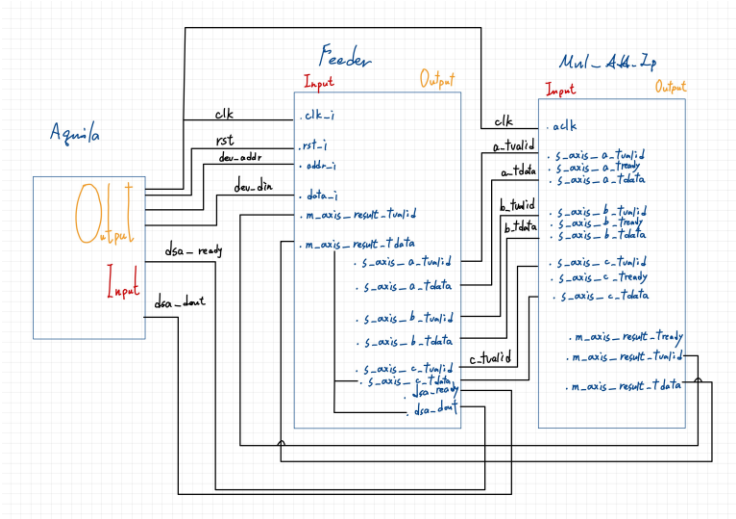
這個 counter 是為了計算 a_data、b_data 已經有幾個 element 被傳入 IP，之所以會被用於 valid 的判斷是因為在傳入所有資料後我會將 valid 訊號拉下，不再讓新的資料傳入 IP。而這個 counter 能夠被 + 1 的條件是建立在，DataBuffer 都已經被填滿，同時 IP 也已經傳來 ready 訊號，在這樣的情況下我們就可以將 valid 訊號拉高並開始計算已經有多少 element 被傳入至 IP。

C. 流程圖

1) 最初版本



2) 修正後之最終版本



IV. 實驗數據 (預期成果與實際比較)

A. 時間比較

我原先預計 IP 計算 Inner-Product 所要耗費的時間應該是 Number of Element + 1 (cycles)，以 8 個 element 為例就是 9 個 cycle。

之所以這樣假設可以參考 Figure 1.IP 流程圖隨著每個 clock 將資料餵給 IP 並在下一個 clk 得到結果，搭配我這次使用的 A * B + C，理論上應該要能達到 n+1 個 cycle 完成計算，然而我透過 ILA 觀察我資料傳給 IP 後的情形卻發現 Figure 2.資料傳輸至 IP 之波形圖的情況，最上方為 m_axis_result_data，下方為 s_axis_a_tdata 和 s_axis_b_tdata，資料傳入 IP 後並沒有在下一個 clk 就完成運算，反而中間空格了 8~9 個 cycle 才輸出。

後來我自己猜想之所以他會不如講義上在下個 clk 回傳達案的原因可能在於我 gen 的 IP 在 A 與 B 之間做的是乘法，相對的一定比加法要耗費更多時間來計算，也因此並不會每筆 data 都能相依著傳入 IP。

Time Consuming (8 element)		
Hardware		Software
FeedingData	Calculation	
0.8~ 0.10 ns	0.15 ns	
0.25 ns		27 ns

B. 計算結果驗證

我利用硬體加速器計算的結果和軟體本身計算的結果差異甚大 Figure 2.資料傳輸至 IP 之波形圖 Figure 3.ELF 執行結果圖，這部分花費我很多時間在處理與解決，我個人認為有幾點原因。

a) 我的作法是將 IP 輸出的答案馬上拉回當下一次的 C_data input，這樣的方式如果 IP 的運算過程及時間跟我預想的一樣的話應該是沒問題的，但如今面臨到前段提及的輸入與輸出間隔大幅增加的情況下，勢必會造成 IP 在計算時被運算對象混亂的結果，導致我最後的 InnerProduct 和軟體計算結果相差甚遠。

b) Handshake 處理不當

我認為另一個可能是我在傳送 a,b 資料時的處理不夠嚴謹，valid,ready 互相確認的當下就馬上傳過去又或是晚了一兩個 cycle 才傳過去，這也是其中一個可能性。

C. 程式碼修正

我在透過 ILA 觀察波型圖除錯時發現，其實我傳進去 IP 的資料並沒有錯，而 IP 回傳的 result_data 也不能說不正確，那問題究竟出在哪裡呢?

事實上查看 Figure 2.資料傳輸至 IP 之波形圖就可以發現，由最底下至上三個波行分別為: a_data, b_data, c_data (被 assign m_result_data)，而因為我在 IV.A 和 a)提到的 IP 並不會馬上回傳 result，加上我是把下一次運算要傳進去 IP 的 c_data 賦予前一次的 m_result_data，所以搭配

Figure 2.資料傳輸至 IP 之波形圖可以發現每個 element 被傳進去的時候其實 c_data stream 都是 0。仔細觀察波型圖右半部就會發現，以前兩組 input data 為例:

第一組		自行計算	IP 回傳
a	b	a * b	Result_data
0.586846	0.586846	0.34438822	
0x3f163b8a	0x3f163b8a	0x3eb053a7	0x3eb053a7

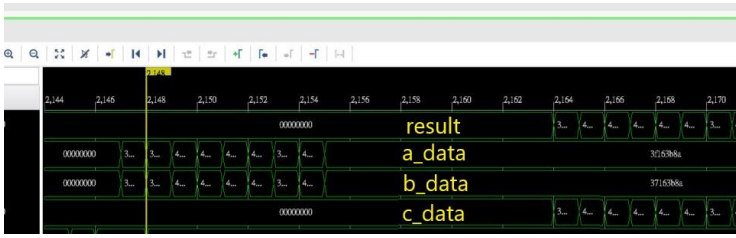


Figure 2.資料傳輸至 IP 之波形圖

第二組		自行計算	IP 回傳
a	b	a * b	Result_data
1.877920	1.877920	3.5265835264	
0x3ff05faf	0x3ff5faf	0x4061b38c	0x4061b38c

IP 傳回來的值和單純的 a * b 相同，證實了我上段提及的想法，因此其實把 IP return 的這些值加總起來就是正確的答案！

根據上方的結論，我將回傳得到的值輪流進行加總，答案卻仍然不正確。最後發現是因為 int 與 floating point 的儲存方式不同，所以不能直接把所有值加起來，可見下方表格為此次其中兩筆資料為例，需要透過自行撰寫或是連接到 Xilinx 內建的 FP IP。很可惜我最後雖然已經將所有元素的正確乘積拿到手，卻因為前期花費太久的時間在進行錯誤的方向，導致我已經找到問題點卻未能在時間內修正成最完整正確的版本。

	0x4142_5add	0xc0b3_2ee2
Int Add	0x1_01f5_89bf (last 32bit) = 9.0196 e-38 (dec)	
Floating Point Add	0x40d1_86d8 = 6.5477(dec)	

D. Figures and Tables

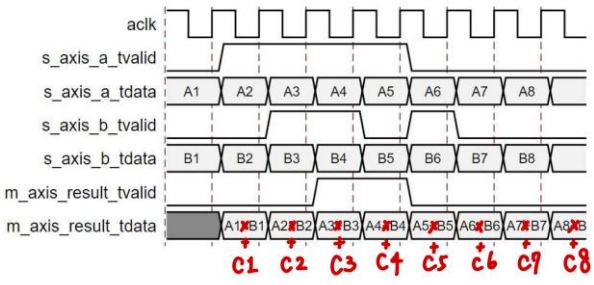


Figure 1.IP 流程圖

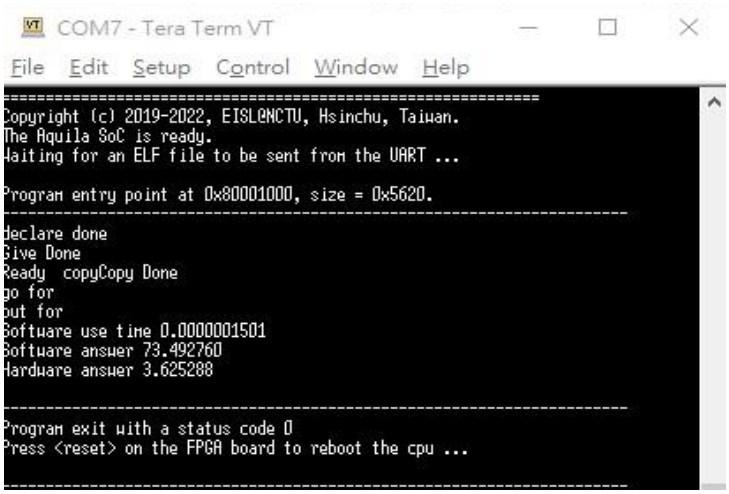


Figure 3.ELF 執行結果圖

I. 心得

這學期修習這門課是我上大學以來收穫最多的一門課，每一次作業難度的增加，都意味著更多時間的投入，同時也代表著更多知識的汲取。

從一開始的 profiling、Branch Predictor 的撰寫、硬體實作新的 Cache 演算法、RTOS 的分析 最後到特殊領域加速器，縱使難度不斷提升，卻也高漲著我的興趣及熱情。之所以當初選修這門課是因為大二上時修習 DLab，發覺自己從頭開始規劃、實作出一塊電路是如此的燒腦以及有趣，也因此毅然決然將”軟硬體整合”學程設定為自己的畢業學程。

這個學期以來，不敢說自己在這堂課的修課成果很亮眼，甚至可以說是偏不理想，但卻無愧於自己，因為這是我在這學期最投入的一門課，無論上課的專心程度、抑或是花費在作業上的時間。

撰寫這份報告及心得時已經考完期末考，或許是緊張、也或許是自身準備的仍不夠充分，很遺憾的在期末上機考未能拿到任何分數，即使考試內容和作業相關，卻

也因為自己未能在考試前將作業的 bug 找出來，導致期末上機考亦十分不理想。雖然搞砸期末考意味著學期成績必定跟著挫敗，更是一種對自己能力的否定，但對我自己而言更重要的是去搞懂老師希望我們學習到的內容，因此考試結束後也特地留下來向助教請教我的問題可能出現的癥結點與參考的解法，我想這是彰顯我自己對於這門課堂投入程度的最佳證明吧!

回想當初選擇這門課時，無論是同學或是去年有耳聞但也沒實際修過這門課的學長姊都很訝異我的選擇，不斷告訴我這門課的 loading 偏重、很硬，現在回想起來，他們好像也沒騙人 XD

但就如同老師在開學前幾堂課所說的，要培養實力、成為未來企業看重的人才，就必須接受相對應強度的訓練，即使我可能最後未能在這門課拿到亮眼的成績，卻無法抹滅這一個學期以來的訓練以及收穫。

很幸運這學期能參與教授的這門課，上大學以來幾乎沒有遇過一位教授對於學生如此傾囊相授、給予幫助並鼓勵我們多多提問，從期初到期末始終如一總是很即時的回覆我的信件並提供建議、指引方向。也很感謝助教學長姐們，每每我寄信去問很笨的問題時都不厭其煩的回答我、幫助我，尤其是毅澤學長應該跟我來回將近有三、四十封信，卻依然保持著莫大的耐心。對於教授和學長姊在這門課上的付出與幫助實在是十分、十分的感激，同時也很難過自己未能在考試或甚至作業上拿出更好的表現來展示教授和助教努力指導的成果，但我想，這一切都會成為我未來求學路上的養分，促使我更加努力、更加投入這個領域。

最後，除了感謝教授和助教，也要感謝當初那個勇敢選課、堅持不退選的自己，謝謝您們。