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)

# 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。

# Software

## 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。

## 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。

## 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。

# 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。

## 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。

### 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。

### 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。

## 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:

### 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。

### 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。

#### 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。

## 流程圖

### 最初版本

### 修正後之最終版本

### 

# *實驗數據 (預期成果與實際比較)*

## 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 |

## 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。

#### 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。

#### 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。

## 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 |

|  |  |  |  |
| --- | --- | --- | --- |
| 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) | |

## Figures and Tables

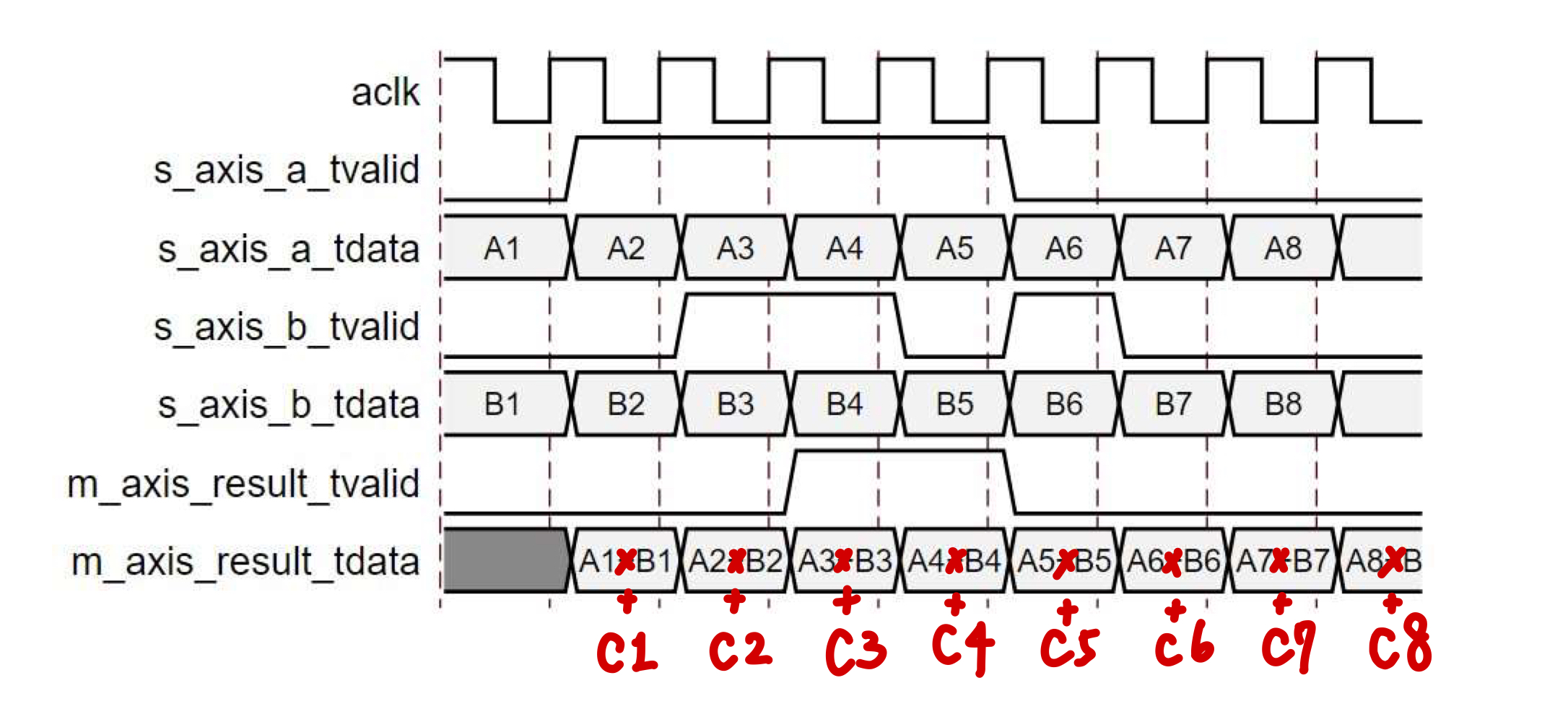


Figure .IP流程圖

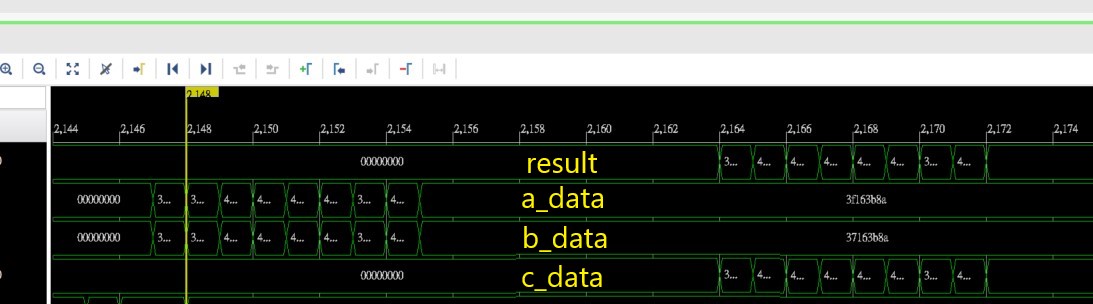


Figure .資料傳輸至IP之波形圖

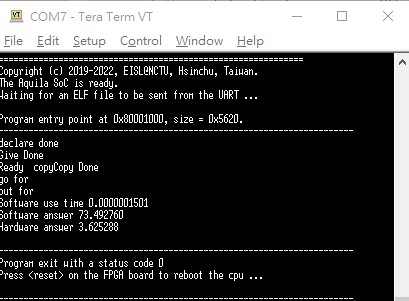


Figure .ELF 執行結果圖

# 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。