國立清華大學 電機工程學系

實作專題研究成果摘要

Workload Optimized SoC Design

系統單晶片之工作負載最佳化設計

專題領域：系統領域

組　　別：第 A393 組

指導教授：賴瑾(Jiin Lai)

組員姓名：張育碩

研究期間：112 年 6 月 30 日至 113 年 5 月 8 日止，共 11 個月

一、報告摘要

系統單晶片(System on a Chip，SOC)為一將許多電子系統整合到單一晶片的積體電路。相較於傳統個別封裝技術，SOC 擁有小體積、成本低，低耗電量及高運算速度等優點。並且由於 SOC 可以充分利用已有的設計積累，提高了 ASIC 的設計能力。因此 SOC 是積體電路發展的必然趨勢，也是 IC 產業未來的發展。

本專題利用 Efabless Caravel "Harness" SoC 平台設計電路並在 Xilinx Vivado 環境下分析、合成，最終於遠端 FPGA 驗證。我們學習系統設計及軟硬體共同設計等相關知識，使用硬體加速三個工作負載 (FIR、MM 以及 Q-Sort) 比較其與使用韌體執行下的效率。

二、報告內容

1. 背景/動機

WLOS (Workload Optimized SoC) 為基於 Google Open-Source Silicon Program 實驗，使用 Efabless Caravel "Harness" SoC 平台開發之專題。於此專題中，我們將設計數個 IP 放在 User Project Area 中，旨在將三個 workload (FIR、MM 以及 Q-Sort) 使用硬體加速，以達最佳的效能。此外我們學習嵌入編程並使用 RISC-V 指令集啟動 IP。於驗證階段，我們可以使用 Verilog Simulator 在 Caravel SoC Harness 系統架構 Caravel SoC 的驗證，但執行效率不佳。因此我們將 Verilog 的驗證環境 (即 testbench) 實現在 FPGA，使整個驗證跑在硬體的速度，加速驗證的流程。最後我們於 Caravel SoC 中的 User Project Wrapper 整合所有設計的 IP，將 Caravel SoC 與 FPGA 整合，進行系統的驗證。

本研究為基於賴瑾老師開設科號：11210EE525100 系統晶片設計 SOC Design 課程期末專題之延續。我們參考來自台清交老師開設同名課程優秀的期末專題，結合不同的演算法及硬體架構，優化我們當時期末專題的成果。開發來源皆來自 GitHub 網站："Bridge of Life Education", https://github.com/bol-edu。

2. 研究目的

此專題旨在運用 Verilog HDL 及 C 語言進行軟硬體共同設計 (software and hardware codesign)，將三項 workload 設計成硬體並實現於 FPGA 上，加速三項 workload 執行在 Caravel SoC 上的效率。最終與使用韌體編程，執行在 RISC-V 上的 workload 進行比較，量化其優化成效。

3. 研究方法

3.1 WLOS 硬體架構及演算法

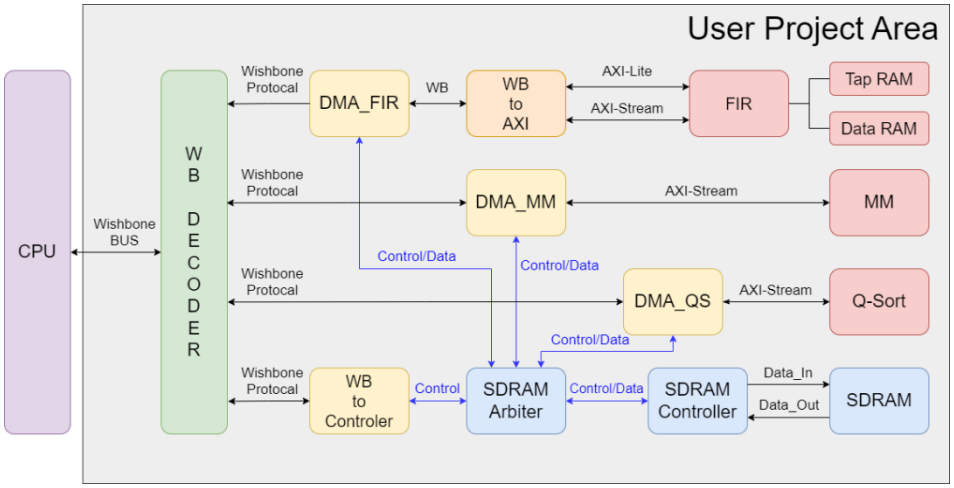

Fig. 1 WLOS Block Diagram

於 Caravel SoC 中，CPU 透過 Wishbone (WB) Interface 與 User Project Area 溝通。然而我們在 User Project Area 中則使用 AXI-Lite 及 AXI-Stream protocol 傳輸資料。因此，我們必須先根據 configuration register 的地址 decode WB 訊號到三個 workload 及 SDRAM。為了達到更好的效能，我們為三個 workload 設計了個別的 DMA，以及在 SDRAM 系統加上 SDRAM Arbiter，減少資料傳輸的路徑。

3.1.1 Finite Impulse Response (FIR)

$$y[t] = \sum_{i=0}^{n} h[i] * x[t-i]$$

為了將 FIR 實現到電路上，我們運用一個 Data_RAM 儲存 data 的資料 x[t]，一個 Tap_RAM 儲存係數 h[t]。於運算階段，我們設計 pipeline stage 以及 FSM 使得在只有一個加法器、乘法器的硬體資源下，得以用 11 個 cycle 完成 11 筆資料的運算。

(1) 開始/完成 FIR:

開始:外界 (DMA_FIR and WB to AXI) 經由 AXI-Lite 拉起 ap_start 訊號。

完成: FIR 完成運算且輸出結果後拉起 ap_done 訊號由 AXI-Lite 通知外界。

(2) 輸入/輸出:

FIR 透過 AXI-Lite 以及 AXI-Stream protocols 與外界溝通，其中:

AXI-Lite 處理: coefficient: h[t]、data length: n、ap_start、ap_done 的讀/寫。

AXI-Stream 處理: input data: x[t]、output data: y[t]的讀/寫。

(3) FIR 硬體演算法:

如果我們按照 FIR 的數學公式直接實現在電路上，我們勢必得重複讀寫 Data_RAM 以改變 x[i-t]的數值。然而因為我們是用 synchronous block RAM (BRAM) 作為記憶體的電路，考效率最低的情形，每改變一輪 x[t]至少需要 11 個 clock cycle。這樣頻繁的讀寫記憶體必然會嚴重地影響電路的效率。因此我們使用 address

pointer 指向每一輪運算 Data_RAM 的起點，減少讀取 block RAM 的次數。

3.1.2 Matrix-Matrix Multiplication

於此專題之矩陣乘法，我們打算加速 4x4 方陣之矩陣乘法。

(1) 開始/完成 MM:

開始:外界 (DMA_MM) 拉起 MM_start 訊號。

完成: MM 完成運算且輸出結果後拉起 MM_down 訊號。

(2) 輸入/輸出:

MM 只透過 AXI-Stream protocol 與 DMA 的控制訊號外界溝通，其中:

DMA 的控制訊號: 寫 MM_start 訊號、讀 MM_done 訊號。

AXI-Stream 處理: input data: Matrix A and B、output data: Matrix C 的讀/寫。

(3) MM 硬體演算法:

若我們將一筆一筆資料匯入 Matrix A、B 中，直到全部匯完再執行矩陣乘法運算，這樣的算法顯然不能達到最好的效率。但若一次匯入 Matrix A 及 B 全部的資料，則需要在 DMA_MM 中加入不小的 buffer 才能一次匯入 32 筆 32bits 的資料，因此我們打算以矩陣 row/column 為單位，配合 SDRAM 一次 prefetch 4 筆資料的特性，持續 feed data 與輸出 data 達到更好的效能。

3.1.3 Quick Sort

於此專題之矩陣乘法，我們打算加速 4x4 方陣之矩陣乘法。

(1) 開始/完成 QS:

開始:外界 (DMA_QS) 拉起 QS_start 訊號。

完成: QS 完成運算且輸出結果後拉起 QS_down 訊號。

(2) 輸入/輸出:

QS 只透過 AXI-Stream protocol 與 DMA 的控制訊號外界溝通，其中:

DMA 的控制訊號: 寫 QS_start 訊號、讀 QS_done 訊號。

AXI-Stream 處理: input data: Array[0:9]、output data: Array[0:9]的讀/寫。

(3) QS 硬體演算法:

根據參考資料，我們可以用最快速的排序方法完成 10 筆資料的排序。

3.1.4 Synchronous Dynamic Random-Access Memory (SDRAM)

我們將原本所使用的 BRAM 改成使用 SDRAM (底層仍用 FPGA 所提供的 BRAM 來合成)。SDRAM 具有更快的讀取速度以及可運用 muti-bank 的特性進行 user code/data address 的分配，達到 bank interleave 的好處，以下為其原理。

若使用 BRAM，其並沒有 bank 的差異，因此當時 code 和 data 皆放在同一

個 bank。此時若要執行：

```
Return_data1 = read_data(0x38000000);
Return_data2 = read_data(0x38000004);
```

這類存取 data memory 的指令時，CPU 會需要先到 instruction memory (code 區) 拿此指令，接著發現此指令會需要存取 data memory (data 區)，因此去 access 到 data 區。執行完後要再到 code 區，拿取下一個指令，而下一個指令也需要到 data 區存取，如此一來 BRAM 會不斷需要存取不同 row。

然而於 FPGA 提供的 BRAM 中，若想要存取與上次存取時不同的 row 時，須將舊的 row 做 precharge，再將欲存取新的 row 做 activation。其中 state 的轉換皆需要操作時間，因此存取與上次不同 row 時會耗費大量的時間。

綜上所述，我們希望能盡量存取與上次相同 row，以節省時間。因 BRAM 將 code 和 data 放在相同 bank，會導致在資料要進 BRAM 的時候需要頻繁 precharge & activate 而浪費掉許多時間。因此我們利用 SDRAM 的 module 中具有 muti-bank 的特性，並設定 linker，在 compile 時將 user code 與 user data 分開放在不同的 bank。

由 code 及 data 在 access 的 locality (即通常我們會存取位於下一個位址的 code/data)，對於 code/data bank，會不斷存取同一個 row 的 data。如此一來就可以不需頻繁的 precharge & activate，進而造成時間上的浪費。

3.1.5 SDRAM with burst mode & prefetch scheme

為了加速 MM、QS 從 SDRAM 獲取資料的速度，我們設計了 burst 模式及 prefetch 機制，使得 SDRAM 可以連續傳出 4 筆資料，其原理如下：

(1) 讀取 SDRAM 過程中，此時將每個 cycle 給 SDRAM request 的 address 增加一定量

   i. +1：Matrix A in MM, Array[0:10] in QS；

   ii. +4：Matrix B in MM。

(2) 因 SDRAM 即可連續傳出 4 筆相同間隔 address 的 data，間隔由 prefetch_step 此訊號所決定，所以我們可以在 1 次的 request 中輸出 4 筆 data。相較於無此設計的機制，減少了 SDRAM 與 MM、QS request 的次數，僅需多花費 3 個 cycles 即可額外獲得 3 筆 data。

3.1.6 SDRAM Arbiter

由於 CPU、DMA_FIR、DMA_MM、DMA_QS 皆會對 SDRAM 有 request。但 SDRAM 只有一個，一次只能處理其中的一個 request，故需要有 arbiter 作為仲裁

者，為各 request 排序。

在考量效率以及實作難易度，對於不同的 workload 有不同的優先級，應該就硬體執行的步驟安排 request 的順序，但我們並沒有研究這方面的演算法，所以我們決定使用最簡單的 FIFO，先 request 先處理，若有其他 request 則先存入 FIFO。

我們設計一組四個位置的 FIFO[0:3]，其中順序為 FIFO[3] →FIFO[2] → FIFO[1] → FIFO[0] → 進入 SDRAM request 階段，因此越早來的 request 會被放在 FIFO[0]，越晚來的會排在 FIFO[1]、FIFO[2]或 FIFO[3]。若同時有 request 則依照 workload 的量由 CPU → FIR → MM → QS 的順序排入 FIFO 中，並且每個 FIFO 位置中有 4 個 bits，分別代表是 CPU, FIR, MM or QS，用以判斷 request 的來源。

以 CPU request 為例，當 CPU 對 SDRAM_Arbiter 給出 read address 的 request 時，arbiter 會先確認 prefetch buffer 中有無此 address，若有則可直接回傳值；若無則將此 request 存入 FIFO，且當輪到此 request 時，將此 request 傳給 controller，controller 會回傳 4 筆 data 給 arbiter，arbiter 會回傳第一筆 data 給 CPU 的 request，並將剩下三筆 data 存在 prefetch buffer，若未來有這幾個 address 的 read request 就可直接輸出，不須再 request 到底層的 SDRAM 中。

3.1.7 Diract Memory Accsess (DMA)

在沒有使用 DMA 時，當 workload 要存取 SDRAM 的 data 時，必須通過 CPU 來回在 workload 及 SDRAM 之間傳遞資料。這不僅會增加 CPU 的負擔，並且因為 CPU 通常並不是閒置的，還會需要許多時間等待 CPU，增加得到資料的時間。

因此，我們為三個 workload，設計了三個 DMA。其中透過 I/O buffer 存取 workload 的 input/output 以及兩個 FSM。其一為 input buffer 空閒時，與 SDRAM 溝通的介面；其二為 I/O buffer valid = 1 時，與 workload 硬體溝通的介面。

(1) DMA_FIR

以下為 DMA_FIR 其設計原理：

i. 內部有一個 input_buffer，對應到 input_buffer_valid：當 input buffer valid=0 時表示沒有準備好輸入的值了，因此 DMA_FIR 就會透過 FIR request 向 SDRAM_arbiter 提出 request，並取回下一個 input data，再將 input buffer valid 拉起來為 1，即代表下一筆 input data 已準備好。當 DMA_FIR 中的 FSM 偵測到 input buffer valid=1 後，即可透過 AXI-Stream ss 將 input 傳給下層的 FIR engine。

ii. 另外，內部還有一個 output_buffer，對應到 output_buffer_valid：當偵測到 sm_tvalid 時表示有新的 output 值計算完畢，此時將此值透過 AXI-Stream sm 送出到 DMA_FIR 中的 output_buffer，並將 output buffer valid 設為 1，

表示有新的一筆 output data 並送出 request 將 output data write 至 SDRAM 中所對應的 array 位置。

(2) DMA_MM

設計原理同 DMA_FIR 但改變部分設計如下：

i. 我們設計了 4 個 32bits 的 input buffer，對應到 prefetch buffer 的大小，也對應到矩陣乘法一個 row/column 的大小。

ii. 16 個 32bits 的 output buffer，用來存取輸出 4x4 的矩陣。

iii. Input valid 仍維持 1 bit，因為每次會從 SDRAM 去 request 4 筆資料，且輸入至 MM 也是一次 4 筆，當 input buffer 全空時 input valid 才變為 0。

(3) DMA_QS

設計原理同 DMA_MM 但改變部分設計如下：

i. 為了優化 MM，我們設計 SDRAM 一次可以 prefetch 4 筆 data 的功能，並為 DMA_MM 設計 4 個 32bits 的 input buffer。但對 QS 而言，我們只設計 1 個 32bits 的 input buffer，其原因為除了資料數少，不必為了 QS 設計 prefetch 10 筆 data 的功能外。因為 QS 的數據是連續 10 筆儲存在 SDRAM 相鄰的位置上， SDRAM prefetch 時可以直接 prefetch 4 筆連續位置的 data，並將其中 3 筆存在 prefetch buffer 中，所以當 DMA_QS 跟 SDRAM Arbiter request 時，大都可以快速的存取到 prefetch buffer 中的 data。

ii. 10 個 32bits 的 output buffer，用來存取輸出 10 個數字的陣列。

3.2 開發環境

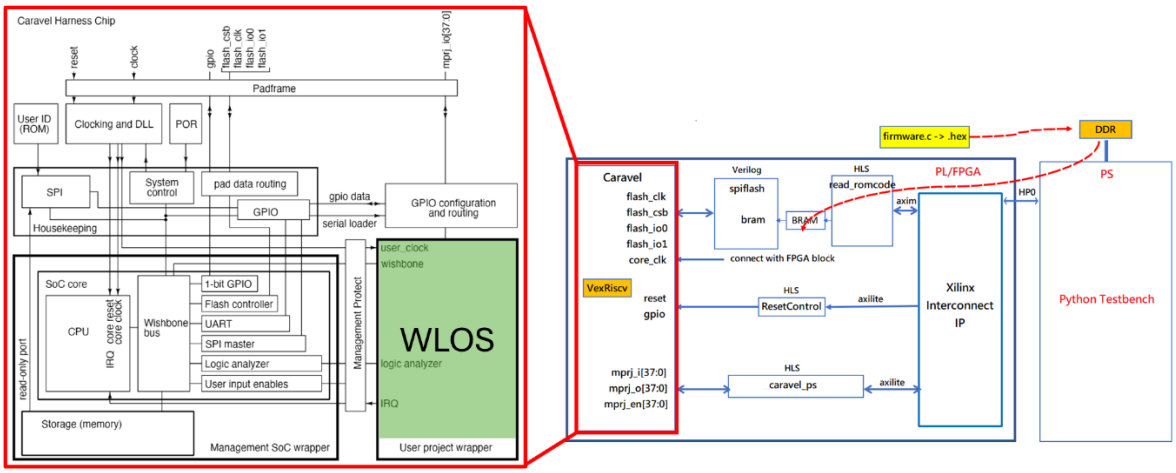

Fig. 2 Caravel SoC FPGA Development Environment

Caravel Soc 環境的開發主要由兩個部分組成：

1. Caravel SoC 中 RSIC-V CPU 的韌體是由 C 語言編寫。當我們在跑模擬以及驗證時，透過 make file 編譯 C 語言，產生 assembly code 以及 machine code 給 RSIC-V CPU。並且我們經由 testbench 或手動匯入 machine code 到 Caravel SoC 中(透過 mprj_io pin)，並根據 configuration register mapping 寫入到 SDRAM 中，其位置為 0x380XXXX。

2. 硬體加速則根據上述驗算法，使用 Verilog 撰寫 RTL code 開發個別硬體 IP。最終在 PYNQ-Z2 此 FPGA 上執行。這些硬體在 user project area 中使用 AXI-Lite 與 AXI-Stream 傳輸資料，並透過 WB_Decoder 轉換 AXI interface 至 Wishbone protocol 與 Caravel SoC 中的 CPU 溝通，最終我們結合 Vitis HLS 寫出的硬體與 Caravel SoC 透過 Xilinx Vivado 的環境生成 bitstream。並將 bitstream、machine code 以及 Python testbench 上傳到 Jupyter Notebook 上，將電路實現在 online FPGA 上，進行驗證。

3.3 實作流程

我們根據第二點介紹之演算法，將使用硬體的方始加速同時執行三種 workload 的效率。以下為實作流程：

1. 使用 Verilog HDL 實做個別硬體 IP，並透過 FSM 控制系統狀態及設計 pipeline stage 在有限的資源下優化 throughput。

2. 考量三種 workload 的硬體演算法，設計出個別的 DMA 以及 SDRAM 系統。

3. 經由 Testbench 跑模擬並透過 Check bits 以及觀察 Waveform 驗證其結果。

   我們在軟體中，根據不同的狀態下寫入不同的 Check bits。並且在硬體的 TB 中等待對應的 Check bits，並根據三種 workload 的 base address 來確認執行完的結果是否正確。

4. 將 RTL code 透過寫好的執行檔呼叫 Xilinx Vivado 合成並實踐電路在 PYNQ-Z2 此 FPGA 上。

5. 將 Vivado 合成出的 Bitstream, firmware 產生的 machine code 及 Python code 透過 Jupyter Notebook 連線到遠端的 PYNQ-Z2 並驗證。

4. 研究結果

4.1 測試數據來源

於驗證環節，我們會在軟體的 head file 中宣告三個 workload 的 base address，以及測試的資料。

4.2 成果比較

我們透過 GTKWave 打開模擬的波形圖，找出 check bits、主要的訊號及 I/O

buffer 等關係，進而判斷模擬結果是否正確。並且我們也能透過觀察 waveform，理解 Caravel SoC 是如何完成作業的。

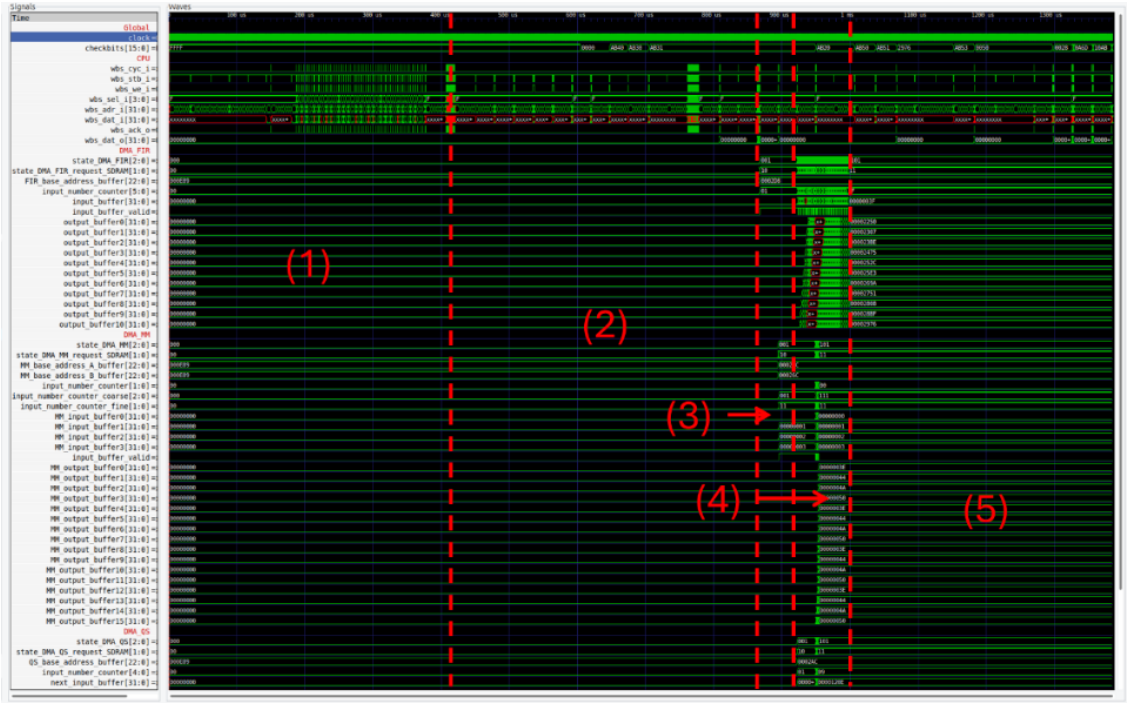

Fig. 3 Simulation Waveform

Fig. 3 可分為 5 個區段，表示 Caravel SoC 完成 WLOS 的五個步驟：

(1) 從 PS 端搬移 mprjram (user code)區段至 user_SDRAM。
(2) 從 PS 端搬移 mprj_user_data 區段至 user_SDRAM。
(3) Workload Initialization：進行 hardware accelerator 的初始設定。
(4) Workload Computation：同時執行 FIR、MM 及 Q-sort。
(5) Check results：輸出並檢查結果的正確性。

並且我們可以透過計算三個 workload 在 simulation waveform 執行的時間，來計算三個 workload 執行的時間。(Simulation clock cycle = 25ns = 25000ps)

(1) FIR：922,762,500ps – 999,862,500ps，共 3084 clock cylces。
(2) MM：949,387,500ps – 954,537,500ps，共 206 clock cylces。
(3) QS：949,937,500ps – 953,012,500ps，共 123 clock cylces。

4.3 分析與討論

4.3.1 Quality of Service (QOS)

從 Fig. 4 中，我們比較 baseline 執行所需要的 cycle，可以發現我們 FIR 為此次加速的瓶頸。並且因其資料數較大，需要執行的時間也遠大於其他兩個 workload，使得總共執行的時間及加速成效被 FIR 限制。

#Cycles	Baseline	Our Work	Improvement
FIR	1,510,415	3084	x489
MM	203,852	206	x990
Q-Sort	97,082	123	x789
Total	1,811,349	3413	x530

Fig. 4 Quality of Service

此外我們可以計算在理想狀態下三個 workload 所需要的 clock cycles：

(1) FIR：約 64 (data number) x 11 (computation) = 704。

(2) MM：16x2 (read Matrix A/B) +16 (computation) +16 (write Matrix C) = 48。

(3) QS：10 (read Array) + 6 (computation) +10 (write Array) = 26。

與理想的電路 (通過 testbench 測量) 不同，當我們同時在系統執行三種 workload 時，測量出來會比理想的電路還要長 (約為 5 倍 clock cycle)。這是因為會需要等待 request 經過 arbiter 仲裁至 SDRAM 以及 SDRAM 本身運作的時間。

雖然 DMA 中有 input buffer，但仍要先確認 input buffer valid 才能開始輸入。並且因為有 SDRAM burst mode，一次會輸入 4 筆 data，導致 input buffer 來不及補充，故 computation engine 並無法像理想情況下有源源不絕的 input 等著被輸入其中，而是需要等待 DMA 的 request 被 SDRAM_Arbiter 選取以及 SDRAM 處理的時間。

4.3.2 Utilization Comparison

從 Fig. 5 中，我們比較 baseline 實現在 FPGA 上所需要的資源。因為使用硬體加速，其中加入了許多的 combinational logic 以及 pipeline FF，必然在 LUT 及 FF 的使用率上有顯著的提升。

Utilization	Baseline	Our Work	Increment
LUT	5,344(10%)	9,784(18%)	+83%
LUTRAM	188(1%)	252(1%)	+34%
FF	6,173(6%)	11,518(11%)	+86%
BRAM	10(7%)	10(7%)	0
DSP	0(0%)	51(23%)	-

Fig. 5 Utilization Comparison

然而相較於 Baseline 沒有 DSP 的實用，我們十分好奇，當硬體實現在 FPGA 時，DSP 是運用在何部分。從 implementation report 中，我們可以看到 FIR 使用了 3 個 DSP，而 MM 使用了 48 個 DSP。因為 FIR 使用了一個乘法器，而 MM 使用了 16 個乘法器，我們可以合理推斷在 PYNQ-Z2 此 FPGA 上，執行一個 32bits x 32bits 的乘法需要 3 個 DSP。

4.4 未來優化方向

優化各個硬體的演算法外(bottle neck 為 FIR)，還應該更深入理解每個 bus protocol、設計出更好的 FSM 等，使得硬體不要有 stall cycle，達到最佳化的效能。

5. 總結

本專題除了要清楚 Efabless Caravel “Harness” SoC 的硬體架構外，更需要開發三個 workload 的硬體演算法並想辦法加入優化電路加速整個系統，以減少硬體 computation cycle 以及 stall cycle。此外，我們可以從分析中得到：使用越多的硬體資源通常越可以使得硬體加速到接近理想的狀態，然而這也意味著我們會生成更大面積的電路、產生更大的能耗以及更有可能超出 FPGA 提供的資源。這時候我們便需要考慮加速成效與硬體資源的 trade off，達到良好的平衡。

本次專題我們成功將三個 workload 用硬體的方式實做出來，並且加入 DMA、SDRAM Arbiter 以及 SDRAM Prefetch 等功能來盡可能地優化我們的電路，使我們用了不到一倍的資源，便可以加速三種 workload 到 500 至接近 1000 倍的速度。我們學習到如何開發硬體演算法及應用各種硬體優化系統行為，最終我們找到可能導致結果不如預期的原因，未來將進一步持續優化。

三、心得感想

為了這兩學期的實作專題，我們從去年暑假開始學習相關知識，從各種線上課程與老師上課的錄影到實作各個 Lab，最後整合從 Lab1- Lab6 學到的知識以及其他硬體優化的技巧才完成 WLOS 這個專題，一路上我們克服了許多障礙。首先理解 Caravel SoC 的架構以及各種 bus protocol handshake 的模式就是一大困難。大三上的我連計算機結構都還沒修過，光要理解要 FIFO 及 pipeline stage 等優化原理就需要不少的時間，更何況還要用 Verilog 實作出來。跑模擬時，一開始也是問題百出，從資料無法從 mprj_io 傳進去 Caravel SoC 中到因電路本身設計等問題導致硬體算出的結果根本是錯的。我們花了需多時間觀察每一個訊號的波形圖，找出電路設計的瑕疵，才把各種問題解決。經過了持續的努力，我們從一開始只從邏輯設計實驗學會最基礎的 Verilog 設計，到最終讓整個電路放到 FPGA 上並成功運行，當看到模擬結果成功的那一刻是很欣慰的，覺得努力總算有了成果，也讓我們得到許多成就感。

這一年來，我學到很多，也進步很多。除了學習到將許多上課知識實際應用到專題上，包括 C++程式、計算機結構、邏輯設計實驗等知識的結合，也讓我了解做好一個專題不單單只是上課學習理論而已，並且發現自己還有許多能力不足之處。儘管過程常常碰到困難，到繳交期限前還以為自己要做不出來了，最後也都一一克服。雖然成果還有很多進步空間，但這一年來是很充實的，學到各種不同的硬體演算法以及加速系統的技巧，希望未來能夠帶著所學持續進步。

最後非常謝謝老師、貴人們一年來的指教及協助。