HW1 REPORT

謝翔丞, 109550025

*Abstract*—撰寫 profiling 電路同時使用 Integrated Logic Analyzer 為 benchmark-Coremark 之執行過程做函示分析與解剖，報告中會提及過程電路設計架構以及對於結果之探討總結。

Keywords—component; Coremark , ILA , Profile, Verilog (key words)

# Introduction

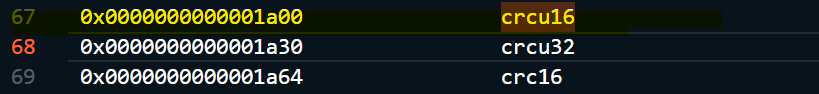
本次作業前提在以更換原先使用之 benchmark Dhrystone 為 Coremark 之前提下，撰寫一份 profiling 電路分析 Coremark 在執行過程中五個較常呼叫的 hotspot function ，同時學習利用 uart 與開發板作互動，傳入需要執行的程式與檔案，並且嘗試使用 Vivado 的內建工具 ILA 查看訊號執行結果。

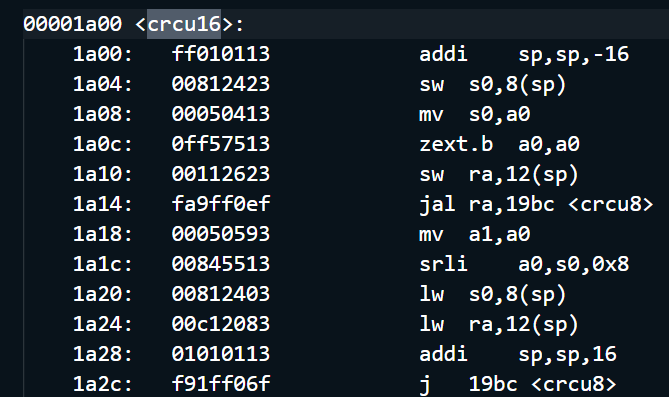
# Profiling 電路設計與架構

## 概念

要進行程式分析，必須從 program counter下手。我們首先觀察\*map 與 \*.objdump從中獲得該函示執行時的pc值起始點與其範圍，接著利用抓取之pc訊號進行範圍判讀並設計counter累加計算cycle，以達到分析各函示使用時長之成果。

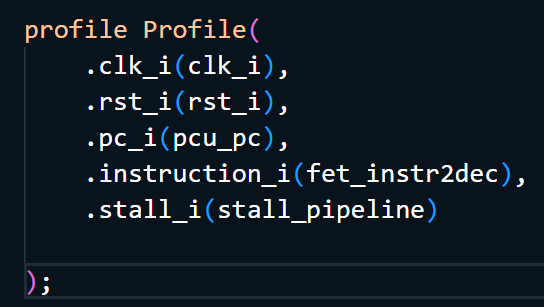
(圖一為以 .map 查找函示對應圖二.objdump之實例)



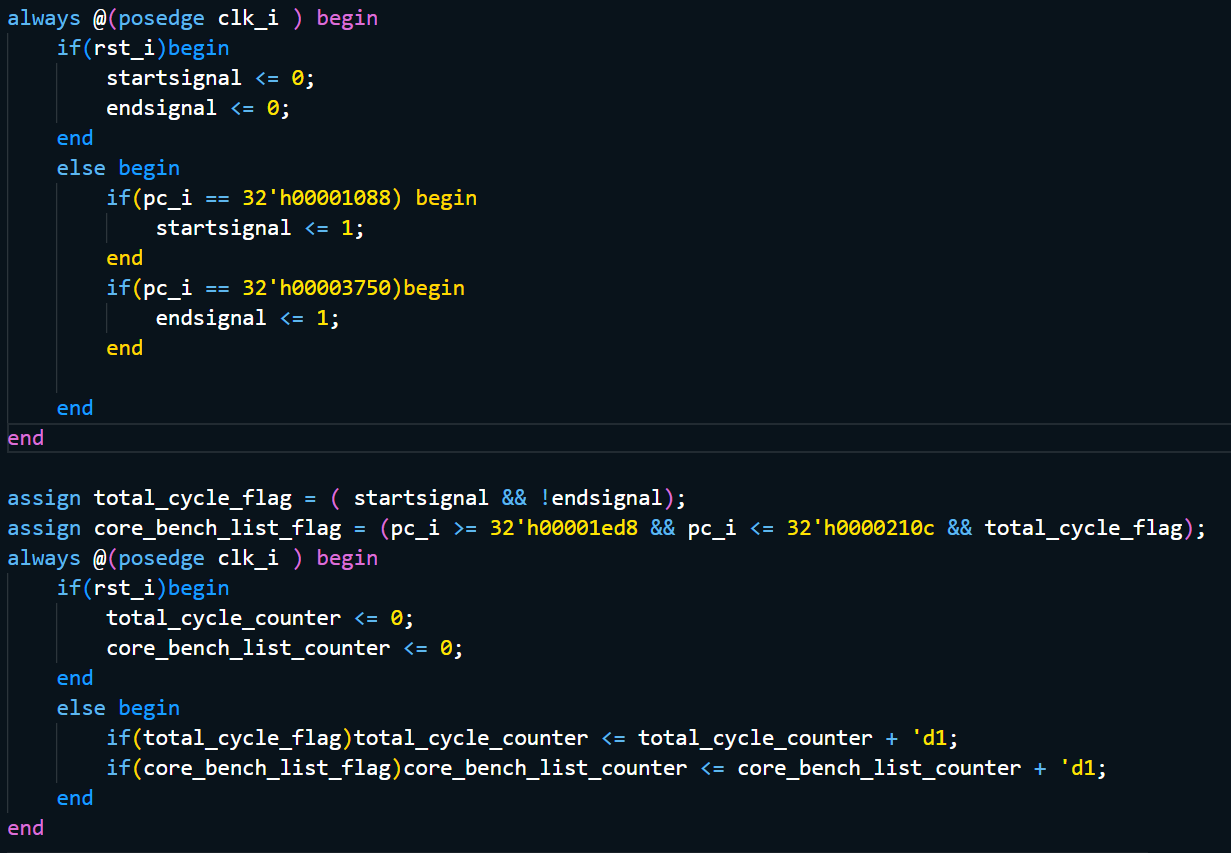


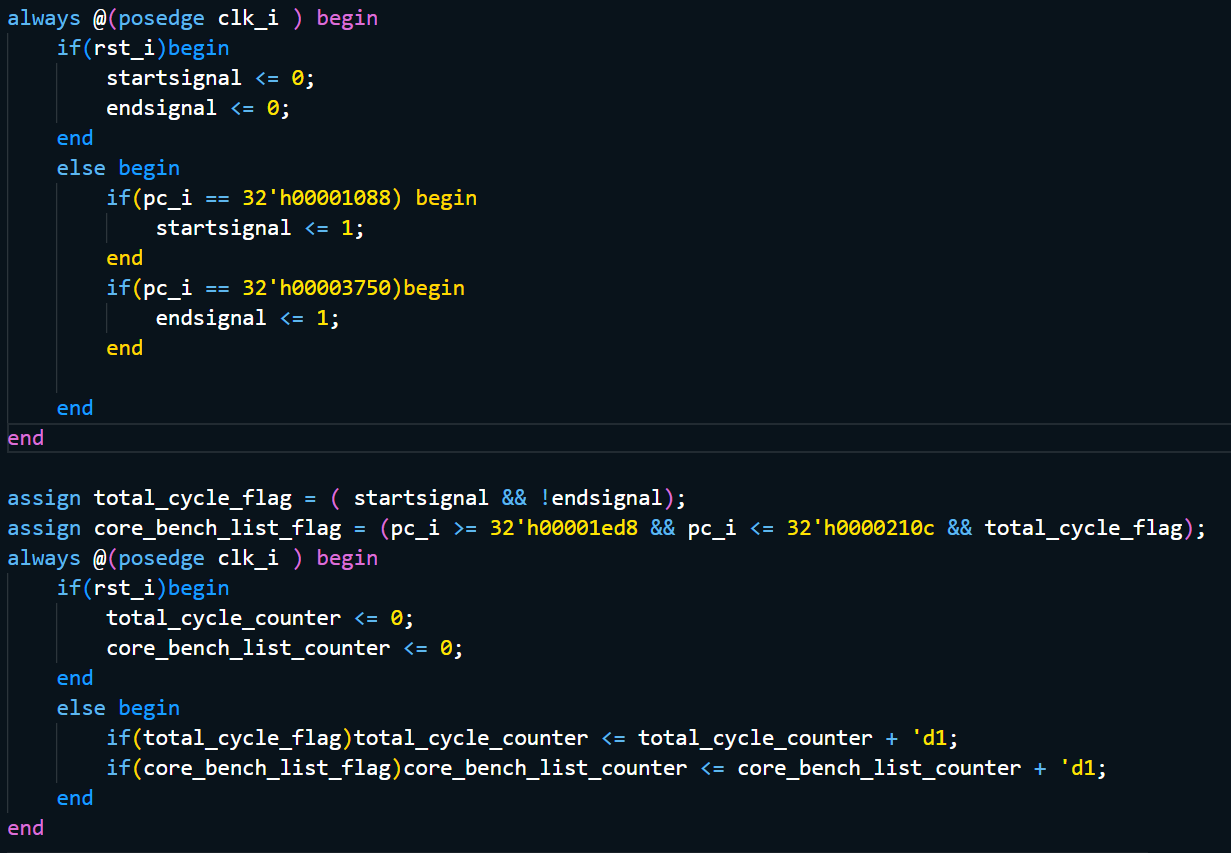
## 電路實作

接續前段所提，在這塊電路中我們必須利用到pc 訊號，同時為了考慮到計算memory computation的cycle數量，我們也需要使用instruction以及stall訊號作為input，因此我將自己撰寫的profile.v =>profile module放置於coretop.v中，如此一來可以使用到現有設置好的 stall\_pipeline 訊號。

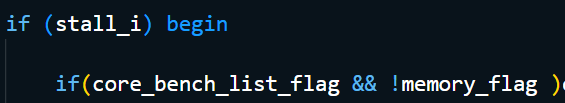


接著介紹profile module內部設計，首先我根據.map檔案設立計算cycle的起始點與結束點，也就是對應到c檔案的main()與exit()，並額外增設total\_cycle\_flag，唯有在這段時間內的cycle才會被計入total\_cycle。另外針對五個欲檢測的function我都分別設立一個flag，這個flag的值是取決於當前pc值是否落於該函示的位址範圍(可於.objdump查找)內，一旦滿足前述條件且total\_cycle\_flag值為1時，便計入該函示之counter計算。(下圖為其一範例與部分截圖)





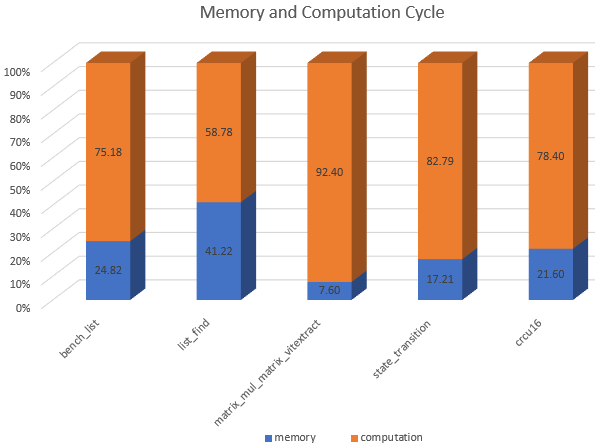
另外在計算是否為memory cycle時，則多利用一個memory flag，這個flag的判斷是利用opcode的[6:5]以及[4:2]來分辨，只要在上段的cycle counter再加上此flag就能計算出該函示的memory cycle 進而也能推出computation cycle。同理，在計算cycle是否stall時，也是利用早前input就拉入的stall\_i作為flag計算。(例:在附圖之condition下，會計算出core\_bench\_list在computation cycle下的stall cycle數量。)



# Profiling結果之探討分析

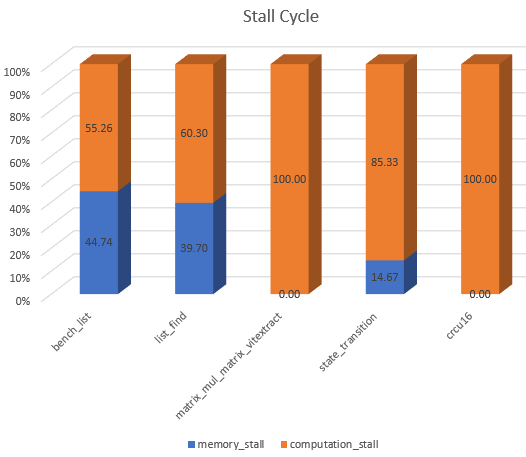
上圖是我的profiling計算結果做成圓盤圖表，可發現coremark在Artix-7 XC7A100T FPGA 開發板上五個函示之執行cycle數量與在PC上有滿大的差異，原先佔有25%左右的core\_bench\_list竟下降至1.8% 附近，其餘函示也有極大的差異，可從兩圖做對比。

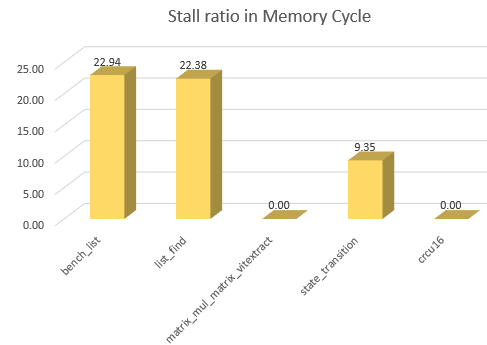
關於導致不同硬體裝置上，程式運行結果有如此差異的情形，我個人有以下幾點想法:

1. 硬體層面:不同CPU在設計上有不同的架構，而部分CPU可能會利用亂序執行來追求更高效能的表現，PC為了應付較大負荷量的處理程序，相對於開發板較有可能實行亂序執行，這有可能是導致如此差異的其中一個原因。
2. Compiler層面:早前老師曾經有點出過一個問題是riscV-gcc在編譯時有可能自行將部份函示優化、轉成inline function，由此可以發現在對compiler下不同參數的情況下，我們的程式可能會編出不同的.mem file，導致在實際執行程式時有不同路徑選擇，這也是造成結果差異的可能原因。

上圖是這五個函示分別進行memory 以及computation cycle的比例呈現。

下兩張附圖則是stall分別在computation以及memory cycle發生的分布狀況，以及stall佔所有memory cycle內的比例。





# Discussions on how to improve Aquila

根據上方圖例可以發現，此次程式在執行時不論是在memory cycle或是 computation cycle ，stall cycle 發生十分頻繁。我針對stall發生率最高的core\_bench\_list觀察其反組譯後的程式碼，發現其中他在s2不斷被使用在lw並接著做bnez，但因為load的關係必須stall兩個cycle才能得到正確的值，這是導致大量stall cycle 出現的原因之一。

針對上述的現象，我認為我們可以透過自行排序assembly code，在不影響程式邏輯的前提下將接續load/store之後的指令改為其他不造成hazard的指令，如此可以避免空無作用的stall cycle，或許會是一個有效的解決辦法。