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)

I. 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。

II. PROFILING 電路設計與架構

A. 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)

```
0x0000000000001a00          crcu16
0x0000000000001a30          crcu32
0x0000000000001a64          crc16
```

```
00001a00 <crcu16>:
    1a00:   ff010113            addi    sp,sp,-16
    1a04:   00812423            sw  s0,8(sp)
    1a08:   00050413            mv  s0,a0
    1a0c:   0ff57513            zext.b  a0,a0
    1a10:   00112623            sw  ra,12(sp)
    1a14:   fa9ff0ef            jal ra,19bc <crcu8>
    1a18:   00050593            mv  a1,a0
    1a1c:   00845513            srli    a0,s0,0x8
    1a20:   00812403            lw  s0,8(sp)
    1a24:   00c12083            lw  ra,12(sp)
    1a28:   01010113            addi    sp,sp,16
    1a2c:   f91ff06f            j   19bc <crcu8>
```

B. 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。

```
profile Profile(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .pc_i(pcu_pc),
    .instruction_i(fet_instr2dec),
    .stall_i(stall_pipeline)
```

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)

```
    if(pc_i == 32'h00001088) begin
        startsignal <= 1;
    end
    if(pc_i == 32'h00003750)begin
        endsignal <= 1;
    end
```

```
assign total_cycle_flag = ( startsignal && !endsignal);
assign core_bench_list_flag = (pc_i >= 32'h00001ed8 && pc_i <= 32'h0000210c && tota
always @(posedge clk_i ) begin
    if(rst_i)begin
        total_cycle_counter <= 0;
        core_bench_list_counter <= 0;
    end
    else begin
        if(total_cycle_flag)total_cycle_counter <= total_cycle_counter + 'd1;
        if(core_bench_list_flag)core_bench_list_counter <= core_bench_list_counter
    end
end
```

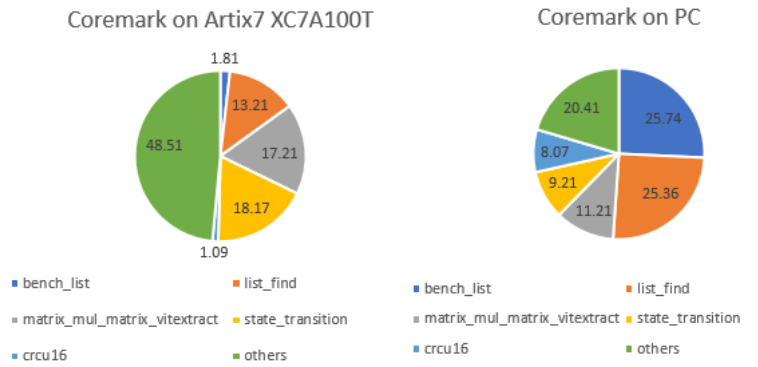
另外在計算是否為 memory cycle 時，則多利用一個 memory flag，這個 flag 的判斷是利用 opcode 的[6:5]以及[4:2]來分辨，只要在上段的 cycle counter 再加上此 flag 就能計算出該函示的 memory cycle 進而也能推出

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。)

```
if (stall_i) begin

    if(core_bench_list_flag && !memory_flag )
```

III. PROFILING 結果之探討分析

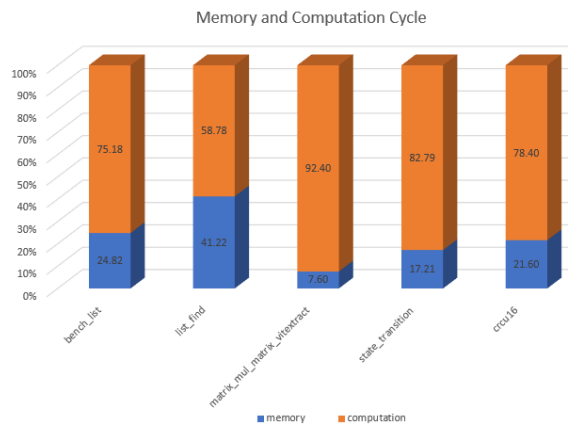


上圖是我的 profiling 計算結果做成圓盤圖表，可發現 coremark 在 Artix-7 XC7A100T FPGA 開發板上五個函示之執行 cycle 數量與在 PC 上有滿大的差異，原先佔有 25%左右的 core_bench_list 竟下降至 1.8% 附近，其餘函示也有極大的差異，可從兩圖做對比。

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:

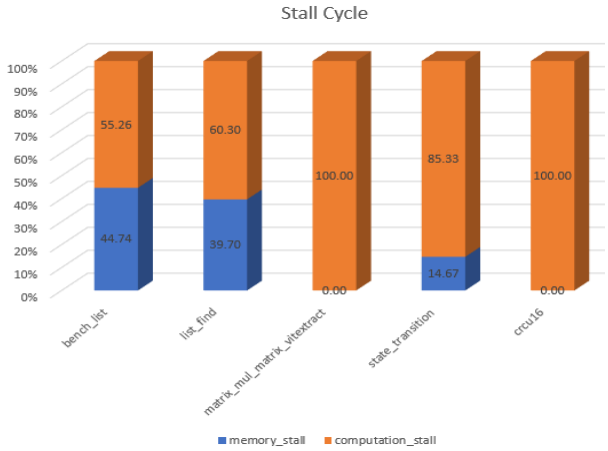
A. 硬體層面:不同 CPU 在設計上有不同的架構，而部分 CPU 可能會利用亂序執行來追求更高效能的表現，PC 為了應付較大負荷量的處理程序，相對於開發板較有可能實行亂序執行，這有可能是導致如此差異的其中一個原因。

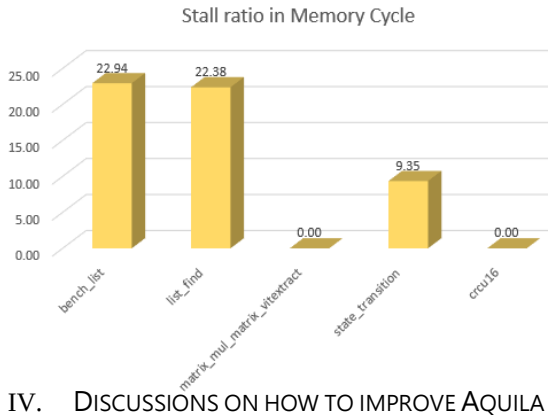
B. Compiler 層面:早前老師曾經有點出過一個問題是 riscV-gcc 在編譯時有可能自行將部份函示優化、轉成 inline function，由此可以發現在對 compiler 下不同參數的情況下，我們的程式可能會編出不同的.mem file，導致在實際執行程式時有不同路徑選擇，這也是造成結果差異的可能原因。



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。





IV. 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。