# HW#1 REPORT

賴御安, 110550168

***Abstract*—在硬體上設計電路配合ila來對coremark函數執行過程做出分析與討論。**

***Keywords—verilog, coremark, ILA, profile (key words)***

## I. Introduction

這次作業是撰寫硬體電路去分析coremark五個較常呼叫的函數, 配合vivado中ila的使用, 來達到即時監控板子上的資訊

## II. Profile 實作

### *A.* 概念

透過觀察coremark.objdump, 可以得到指定函數的program counter的範圍, 當從wb stage後拉出的pc(wbk_pc2csr)介於上述的範圍中時, 則可計算出對應函數總共的cycle數。另外對於cycle的種類, 我也做出一些分析, 將load、store及stall分開, 進而可以做出更進一步的討論。

### *B. profile*實作

這次電路的實作我選擇在core_top.v的最後面加上額外的電路。另外也修改原先decode、execute、memory、writeback的i/o port和部分的電路, 使得能將decode stage判斷load/store instruction的結果沿著pipeline傳到最後給profile的電路使用。

整個profile電路的架構為使用flag來紀錄coremark開始與結束和五個函數的pc範圍。Coremark開始的pc是對應coremark.objdump中start的位置, 而結束的pc是利用ila得到最後pc停留的位置。因為在經過start和end後都需要保留他們的狀態, 所以我是利用reg去存, 再用assign去改變total_cycle_flag。

而五個要分析的函數則是單純使用assign, 透過判斷pc的範圍去更新flag的值。當coremark開始且並未結束, 同時在對應的範圍, 則增加counter的值。

判斷是否為stall cycle的方式則為紀錄下前一個cycle的pc, 判斷是否跟現在這個cycle的pc一樣。將此比較結果放在stalling用來做為啟動stall cycle的flag。這也能配合load/store來算load/store stall的cycle。

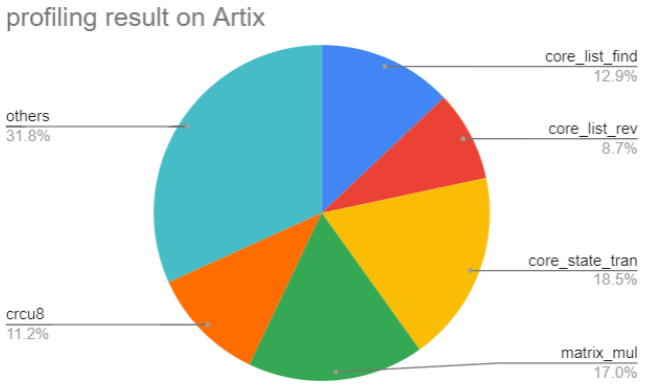
在這個profile的電路中, 對於每個函數我總共紀錄了6個counter值, 分別為

1. 函數總共的cycle數(counter_func_name)
2. 函數load cycle數(counter_load_func_name)
3. 函數load stall cycle數(stall_load_func_name)
4. 函數store cycle數(counter_store_func_name)
5. 函數store stall cycle數(stall_store_func_name)
6. 不包含load/store 的stall cycle數 (counter_stall_func_name)

以及一個計算total cycle的counter (counter_total_cycle)

## III. 分析與討論

### *C.* 比較在不同平台上*profile*的結果




上方的圓餅圖是在artix-7上分析五個指定函數cycle的比率, 對比於在pc上的結果, 會發現兩邊的分析結果不太一樣。底下幾點是我認為造成這樣現象的可能原因:

1. 分析函數的不同:在pc上profile的方式是藉由在原本函數中去呼叫分析的函數, 進而去紀錄當前函數的使用時間及呼叫次數。這會改變部分程式的執行順序, 可能會與原先直接執行函數的順序不同。而使用硬體來profile並不會造成這樣的情況, 只是在程式執行完後進行紀錄。
2. 環境差異:在pc上的gprof的環境是gcc/g++, 而在artix-7板子上則是組合語言。雖然我們是用的.elf檔案是從c code make出來的, 但是兩者在呼叫函數與執行的時間仍然會有差異。由crcu8這個函數來看比較單純, 因為它沒有進行load/store的cycle, 可是在板子上與pc上的cpu使用比率就已經相差了4%。再看到core_list_find這個有包含load的函數, 板子上得到cycle的比率卻少了十個百分比以上, 表示對於core_list_find中load的行為在pc上和板子上也有很大的差異。
3. 測量單位不同:gprof的測量單位為時間, 且為粗估值, 而透過verilog, 計算的為實際總共經過的cycle數。兩者間有差別是合理的。

### *D. cycles discuss*

| counter | core_list_find | core_list_rev | core_state_tran | matrix_mul | crcu8 |
|---|---|---|---|---|---|
| memory | 0.2662735182 | 0.4148871263 | 0.1437462699 | 0.1053902224 | 0 |
| computation | 0.7337264818 | 0.5851128737 | 0.8562537301 | 0.8946097776 | 1 |

上方表格是computation和memory cycle的比率。對於表格中memory cycle的比率是來自於load/store的cycle總和除以總共的cycle數。而computation cycle的比率是用不包含load/store 的cycle數除以總共的cycle數。

| counter | core_list_find | core_list_rev | core_state_tran | matrix_mul | crcu8 |
|---|---|---|---|---|---|
| stall | 0.6004307853 | 0.2363573166 | 0.1512496266 | 0.5601224765 | 0 |
| computation | 0.3995692147 | 0.7636426834 | 0.8487503734 | 0.4398775235 | 1 |

上方表格是computation cycle中stall和computation cycle的占比。計算方式都不包含load/store 的cycle, 前者使

用不含load/store stall cycle 的 cycle 數除以 computation cycle，而後者只計算computation期間的cycle。將兩張圖比較可以發現在core_list_find和matrix_mul_matrix_bitextract這兩個computation比率較重的函數中，stall cycle的比率相當高，對照.objdump中可以發現core_list_find有許多的branch指令和 load-use 的 hazard，在計組中我們也學到這個指令是會導致 stall cycle 的發生。即使使用branch prediction及forwarding，仍然會造成部分的stall發生，因此這項數據是合理的。而matrix_mul_matrix_bitextract中stall比率高的原因可能跟函數本身要進行許多乘法以及存在巢狀迴圈的結構有關。

| counter | core_list_find | core_list_rev | core_state_tran | matrix_mul | crcu8 |
|---|---|---|---|---|---|
| load | 1 | 0.6666666667 | 0.6553973903 | 0.963876652 | x |
| load stall | 0.3278538813 | 0.5001633987 | 0.01266968326 | 0.3331809872 | x |
| store | 0 | 0.3333333333 | 0.3446026097 | 0.03612334802 | x |
| store stall | x | 0 | 0.2581755594 | 0 | x |

上方表格是各函數load/store 與 load/store的stall cycle 比率的表格。load/store 的比率是 load/store cycle 分別除以memory cycle的比率，而 load/store的stall cycle比率為 stall cycle占load cycle的多少百分比。將此結果對照.objdump的instruction也相互呼應，core_list_find中並沒有store指令，crcu8中沒有需要load/store to memory的指令。這也驗證這些數據的合理性。

| | core_list_find | core_list_rev | core_state_tran | matrix_mul | crcu8 |
|---|---|---|---|---|---|
| 1.total | 0.129480373 | 0.08708443413 | 0.1846497779 | 0.1695443587 | 0.111505572 |
| 2.load | 0.03447719445 | 0.02408680708 | 0.01739602734 | 0.01722285421 | 0 |
| 3.load stall | 0.01130348202 | 0.01204733929 | 0.00022040215 | 0.00573832756 | 0 |
| 4.store | 0 | 0.01204340354 | 0.00914668948 | 0.00064546345 | 0 |
| 5.store stall | 0 | 0 | 0.00236145167 | 0 | 0 |
| 6.computation stall | 0.05704283308 | 0.01204340354 | 0.02391363396 | 0.08495715975 | 0 |

前述的比率都是相對的結果，方便做出他們之間的對比，而在最後附上所有有統計的cycle除以 total cycle的比率，方便判斷各項cycle實際占的比例，在下個部分對於improve Aquila能有更好的參考依據。

## IV. Discussions on how to improve Aquila

根據上方這些數據，我認為可以先從 stall cycle 的部分來下手，因為stall cycle會使得cpu的執行效率降低。從第三部分cycle discuss的部分的最後有全部有統計的cycle除以total cycle的比率，方便我們判斷哪些stall cycle的實際占比較高，進而可以優先改進。

從表格中可以挑出有較高的computation stall cycle的函數來進行改善，分別為core_list_find, matrix_mul_matrix_bitextract。在 cycle discuss 的部分也有提到觀察.objdump的指令後認為是branch指令和load-use 的 hazard的影響。根據教授上課所說的，現在使用的Aquila版本的branch prediction unit並未是較有效的bpu，可能因此造成stall cycle的比率較高，所以我認為對於branch指令優化需要依靠修改bpu中的電路，使得預測的準確率提升。對於matrix_mul_matrix_bitextract，我認為stall cycle來自於mul指令和多層for迴圈的最後一次與第一次的predict的預測錯誤，但我認為這個部分無法在single cycle cpu 得到好的解決方式，目前想到可能的作法要透過變成multithread或其他方式。