HW#1 Report Title
Real-time Analysis of a HW-SW Platform

邱振源, 111550100

***Abstract*—對 Aquila 進行實時分析，並使用 Vivado 的 ILA 工具分析 CoreMark 中五個主要函數的記憶體使用狀況、佔用的周期數以及周期的停滯情況。我們設計了 profile 電路，並分析記憶體相關指令與 stall cycle 提出 Aquila 的修改建議。**

Keywords—real-time analysis; RISC-V; Aquila; CoreMark; memory usage; stall cycles; profile circuit; Vivado ILA

I. INTRODUCTION

這次的作業主要是透過分析 RISC-V 處理器 Aquila 的電路設計狀況，從而在其中加入 profile 的程式碼，並藉由這樣的方式配合 Vivado 的 ILA 分析 CoreMark 裡面的五個 hotspot 函式的 memory 使用狀況，佔有的 cycle 數與 cycle 的 stall 狀況。

II. PROFILE IMPLEMENTATION

A. Concept

先從 program counter 下手，從 coremark.objdump 可以找到每個 function 的起始與結束位子，再來，透過追蹤一個 instruction 在 Aquila 的傳遞運作流程後，在合適的地方加上 profile 電路，藉由 counter 的累加與適當的判斷條件來得到所需的計算資料。

```
000019b4 <crcu8>:
    19b4:       ffffa637                lui     a2,0xffffa
    19b8:       00050693                mv      a3,a0
    19bc:       00800713                li      a4,8
    19c0:       00058513                mv      a0,a1
    19c4:       00160613                addi    a2,a2,1 # ffffa001 <  stack top+0xfffed6a1>
    19c8:       00a6c7b3                xor     a5,a3,a0
    19cc:       0017f793                andi    a5,a5,1
    19d0:       40f007b3                neg     a5,a5
    19d4:       00155513                srli    a0,a0,0x1
    19d8:       00c7f7b3                and     a5,a5,a2
    19dc:       fff70713                addi    a4,a4,-1
    19e0:       00a7c7b3                xor     a5,a5,a0
    19e4:       01079513                slli    a0,a5,0x10
    19e8:       0ff77713                zext.b  a4,a4
    19ec:       0016d693                srli    a3,a3,0x1
    19f0:       01055513                srli    a0,a0,0x10
    19f4:       fc071ae3                bnez    a4,19c8 <crcu8+0x14>
    19f8:       00008067                ret
```

Fig. 1. Example of a function in coremark.objdump

B. Profile Method

我選擇在 core_top.v 的後面加上 profile 的電路，因為一個 instruction 在 execution stage 結束後才算是真正被執行到，所以我用 exe2mem_pc 來當作抓 pc 的地方，另外在 trace code 之後可以發現 execution stage 的兩個 signal: exe_re 代表這個這個 instruction 是 load instruction，而 exe_we 則是代表 store instruction。因此我在 profile 的地方用 assign 設定兩個 flag 把 load 和 store 判定完成。

接著，我對不同的 hotspot 函是設定不同的 flag，而這些 falg 的起始與結束位置就是根據 objdump 檔中該函式的開始與結束的地方。同時，我也設定 profile 開始 counting 與結束的地方，開始的部分我設定成 main() 在 objdump 檔中開始的地方，而結束的部分，我一開始設定成 main()結束的地方，然而在觀察 ILA 之後發現，這樣在計算 cycle 的時候會造成在跑數值的時候出現很多的不同，因此我觀察了 pc 在 ILA 顯示結束的地方，並將結束 counting 的地方設定成 uartboot 的 exit(0) 跑無窮迴圈的位置。

而在判斷 stall 的部分，在計算 counting 的區域內，我記錄了 pc 在上一個 clock cycle 的值(pc_test)，比較 pc_test 和當前的 pc 的值是否相等，若兩者相等則代表 stall cycle 發生。在實作的部分我一樣用一個 flag 將這樣的情況 assign 給這個 flag。

最後，我們就可以藉由在 if 裡面排列上述這些 flag 的布林運算式，讓我們可以在指定的範圍內設定一個 counter 來計算我們所想要測試的 instruction cycle 數，而除了用一個 total_cycle 來計算運行 coremark 的 cycle 數之外，針對五個 hotspot 函式我也計算了下列六種情況的 cycle 數：.

a) Function 的總 cycle 數(_counter)
b) Function load cycle 數(_load_counter)
c) Function store cycle 數(_store_counter)
d) Function load stall cycle 數(_load_stall_counter)
e) Function store stall cycle 數(_store_stall_counter)
f) Function 的其他 stall cycle 數(_stall_counter)

III. RESULT AND ANALYSIS

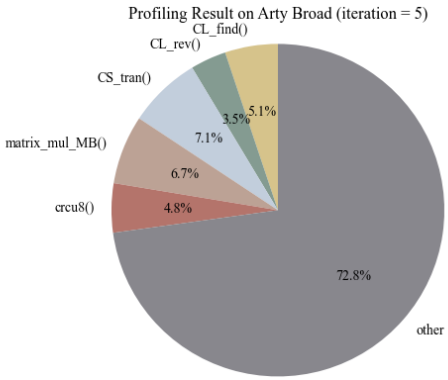

Fig. 2. 各 function 在 CoreMark 的 cycle 數比例(-DITERATIONS=5)

在完成 profile 的電路之後，我透過 arty board 測試結果，將 counter 計算到的 cycle 數做成如上圖(Fig. 2)的圓餅圖之後發現，這樣的結果與 pc 的結果差距甚遠，尤其是計算 other 的地方明顯比例比 hotspot 函式多很多。在閱讀 CoreMark 的 Makefile 之後，我發現在-DITERATIONS 的地

方設成 5，我推測可能是因為 iteration 的次數太少，因為在板子上面會有其他的 cycle 數來讓程式碼在板子上運行，而如果 iteration 的次數越高就可以讓板子上真正跑 CoreMark 的比例變高，所以在觀察 core_main.c 之後，我發現可以將 -DITERATIONS 的地方設成 0，同時觀察這樣的 iteration 次數，發現他會跑 2000 次。

A. *Function Cycle*

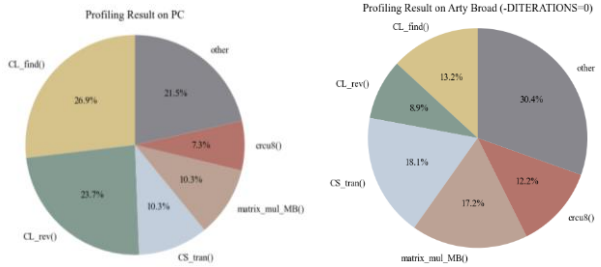

Fig. 3. 各 function 在 pc 與 arty board 的 cycle 數比例

再來我將 CoreMark 再 pc 和 arty board 的結果整理成上方的圓餅圖(Fig. 3)，在 pc 是用老師提供的函示比例，而 arty board 的部分則是使用 -DITERATIONS=0 的部分，然而根據上圖，可以發現在 pc 上的結果與在 arty board 的結果還是有不小的差距，我推測可能是因為以下的原因：

a) 執行環境差異：

因為我們是透過 gprof 來在 pc 上分析比例，而在 pc 上的 gprof 是透過 gcc 運行的，反觀 arty board 是透過組合語言運行的，儘管 elf 檔也是從 c make 出來，然而兩者在運行函數與執行的時間可能有所差異。像是比較單純(沒有 load/store)的 crcu8()在兩者的比例差異上就沒有很明顯，但 core_list_find() 就有明顯得差異，就可以推測出 load 執行在 pc 與板子上所需的時間有很大不同。

b) 編譯器優化：

就像上面提到的，gprof 會透過 gcc 運行，而根據老師上課的時候所提，gcc 編譯器在編譯時可能自己針對部分的函示進行優化，而這樣子優化的結果可能讓 cycle 數量不同，進而使的比例有所不同。

c) 硬體層面：

因為 CPU 的設計架構上可能會透過亂序執行、平行運算來追求更好的效能表現，相較板子可能會有更多為了優化而進行的運算進而使的運行的 cycle 比例有所不同。

B. *Memory Cycle*

TABLE I. MEMORY CYCLE V.S. COMPUTAION CYCLE

Instr.	Function Name				
	cl_find()	**cl_rev()**	**cs_tran()**	**matrix()**	**crcu8()**
Laod	53.46%	27.61%	19.36%	13.66%	0%
Store	0%	27.61%	7.65%	0.76%	0%
memory	53.46%	52.23%	27%	14.43%	0%
compute	46.53%	44.77%	73%	85.57%	100%

TABLE I 記錄了五個 hotspot 函式中 load, store, memory, computation instruction 的 cycle 數佔該函式總 cycle 數的比例。可以發現 core_list_find()和 core_list_reverse()的 memory instruction 比例較高，其中 core_list_find()沒有使用到 store instruction，而像是 matrix_mul_matrix_bitextract() 的 computaion instruction 比例較高，可能是因為在該函式中進行較多的乘法運算與巢狀迴圈所致。

C. *Stall Cycle*

TABLE II. STALL CYCLE IN EACH FUNCTION

Stall Instr.	Function Name				
	cl_find()	**cl_rev()**	**cs_tran()**	**matrix()**	**crcu8()**
load	35.50%	13.81%	9.68%	6.83%	0%
store	0%	13.81%	3.82%	0.38%	0%
memory	35.50%	27.62%	13.5%	7.22%	0%
compute	17.81%	0%	2.08%	46.73%	0%

TABLE II 紀錄了 load stall, store stall, memory stall, computation stall 的 cycle 數佔該函式總 cycle 數的比例。跟 TABLE I 比較可以發現，memory cycle 大概是 memory stall cycle 的兩倍，觀察 core_top.v 可以看到老師在 D memory 的 FSM 下了 one-cycle delay 的註解，讓每次的 memory instruction 都必然會有一個 stall cycle，這樣在需要較多 memory instruction 的函式就容易有較多的 stall cycle。

此外 matrix_mul_matrix_bitextract()的 computation stall cycle 較多，推測可能與該函示要進行乘法與巢狀迴圈的運算有關。再者，core_list_find()也是 stall cycle 比例較高的函式，我們可以觀察.objdump 檔案，core_list_find()有較多的 branch 指令和 hazard 的部分，而在計算機組織中也有提到，這樣的指令也是讓函式有 stall cycle 的原因之一。

IV. IMPROVEMENT OF AQUILA

在可以改進 Aquila 的地方，我認為主要可以從減少 stall cycle 的地方著手，來讓 Aquila 處理器的效率更高。

A. *Branch and Data Dependence*

像是想優化 core_list_find()這類的函式，就可以先從優化 branch 的 stall cycle 著手，觀察現在的 bpu 可以發現現在是用 1-level 的 predict，這樣並不是最有效率的，可以透過優化 bpu 來減少 stall cycle。另外就是 data hazard，可以在 Aquila 裡面設計可以重排指令的地方，來減少由於 hazard 造成的 stall cycle。

B. *Optimize muldiv Unit*

如果是想優化 matrix_mul_matrix_bitextract()的函式，我發現這個函式在計算時需要透過 muldiv unit，如果可以優化這個單元，讓他在計算時可以減少 stall cycle，或是透過 multithread 的方式讓這類函式進行平行運算，便可以在這類函式的 computation stall cyele 大幅度的降低。

C. *FSM Of Memory System*

就像上面提過的，處理 D memory 時的 fsm 會讓 Aquila 不管遇到怎樣的狀況都會 stall 一個 cycle，如果可以重新處理這樣的 fsm 便可以降低 memory 類的 stall cycle。