HW 1 – Program Profiling

沈昱宏, 110705013

***Abstract*—In this homework assignment, I added new hardware components on Aquila, making it capable of collecting profiling data for runtime analysis. Using the CoreMark benchmark, I collected profiling data, which included execution cycles for each function and the ratio between computation and memory access cycles. This data is crucial in identifying performance bottlenecks and evaluating the system's efficiency.**

Keywords—Aquila, CoreMark, Profiling

I. INTRODUCTION

In this homework assignment, the objective is to analyze the execution behavior of the Aquila System-on-Chip (SoC) by using hardware profiling components. Profiling is an important on understanding performance bottlenecks and optimizing software running on hardware platforms. By modifying the Aquila SoC, I enabled the system to collect runtime profiling data that count the execution cycles spent in different functions. Additionally, I calculated the ratio between computation cycles and memory cycles for each function, which provides insights into the efficiency of the program's memory usage versus computational workload. This analysis helps identify hotspots in the code and optimize the performance of the system by balancing between memory access and computation.

II. PROFILING MECHANISM

A. Counting Total Cycle

To achieve the goal, I created an additional module in the core of Aquila. The input of the profiling module is clk, rst, and the address in EX stage, which is the variable "exe2mem_pc" in core_top.v. I created a register as a counter and reset it to 0 when the signal rst is 0. The counter is incremented by one for each cycle, starting when the address 0x1088 (the starting address of main) was read and continuing until the address 0x1798 (the return address of main) was reached. The result shows that the number of total cycles is 2563196 (decimal).

B. Counting the Cycle of Top-5 Hotspots

In the module I created in part A, I created 5 registers that initialized when rst to count the cycle of each function. I used the memory range of the functions to decide which register I should increment. For example, the instruction address of core_list_find() is 0x1cfc to 0x1d4c. When the input address lies in the range, I will add the register for counting core_list_find() by one. The result is shown as follows:

TABLE I. TOP-5 HOTSPOTS

Function	Metric	
	Cycle Count	***Percentage***
core_list_find()	327671	12.8 %
core_list_reverse()	221621	8.6 %
core_state_transition()	450902	17.6 %
matrix_mul_matrix_bitextract()	427164	16.7 %
crcu8()	304624	11.9 %
main()	2563195	100 %

C. Counting the Memory Cycle of Top-5 Hotspots

To count the memory cycles of the 5 functions, I used the signal that indicates the memory r/w in exe stage, which is the variable "exe_we" and "exe_re". If either of these signals is true, the memory will be accessed. Therefore, I perform a bitwise OR on these two signals, create an additional input, and count the number of times it becomes true when entering each function. I used the signal in exe stage because it can also take the stall cycles into consideration. When entering the stall cycle, the signal will be kept true.

TABLE II. MEMORY CYCLES / TOTAL CYCLES

Function	Metric		
	Memory Cycle Count	***Cycle Count***	***Percentage***
core_list_find()	175200	327671	53.5 %
core_list_reverse()	122400	221621	55.2 %
core_state_transition()	121760	450902	27 %
matrix_mul_matrix_bitextract()	61640	427164	14.4 %
crcu8()	0	304624	0 %

D. Counting the Stall Cycles related to memory

To count the stall cycles, I do bitwise OR on all the stall signals and count the number of signals when performing memory instructions. Note that I only count the number of stall cycle created by memory instructions.

TABLE III. STALL CYCLES / MEMORY CYCLES

Function	Metric		
	Stall Cycle Count	***Memory Cycle Count***	***Percentage***
core_list_find()	87600	175200	50 %
core_list_reverse()	61200	122400	50 %
core_state_transition()	63800	121760	52.4 %
matrix_mul_matrix_bitextract()	30820	61640	50 %
crcu8()	0	0	- %

III. Discussion

In this section, I will briefly discuss the following topics.

A. Result of Aquila vs. PC

Fron the result, you can tell that the hotspot is quite different from that of PC. The distribution of the function in Aquila is more uniform than that of PC. The result might be caused by different ISA between PC and Aquila. Aquila use RISC-V, whereas PC normally uses ARM or x86.

B. Computation vs. Memory Cycles

In the result, you can clearly see that the core_list_find() and core_list_reverse() contain lots of memory cycles, whereas core_state_transition contains some, matrix_mul…() contains a little but do more computation, and crcu8() does not contain any memory cycle. This result is not suprising because this is what the function does, finding an item in the list will ends up with higher memory cycles because you are using the memroy more.

C. Stall cycles

As you can see in the table III, stall cycles related to memory account for about 50% of memory cycles, which means that the software should be improved. In addition to stall cycles caused by memory access, there are also stall cycles related to multiplication. In the matrix_mul… function, you can see lots of stall cycles caused by multiplication.

IV. Future Directions of Aquila

There might be several directions to improve Aquila. First, as mentioned above, the mul instruction takes several cycles to complete. Adding a hardware to handle multiplication might be a good idea. Secondly, the percentage of stall cycles caused by memory operations is too high. Adding a cache mechanism or add a prediction mechanism to prevent data hazard by executing independent instructions might solve the issue. Finally, Aquila currently does not support vector operations, using a SIMD architecture might help when the code has lots of computations.