Oblig 1 INF-2200

Sigurd Uhre and Sammol Buljo

September 2021

1 Info

Sigurd Uhre. Email: suh000@uit.no. GitHub usernames: siguhr.

Sommol Buljo. Email: sbu028@uit.no. GitHub usernames: SammolA.

GitHub repository: https://github.com/uit-inf-2200-f21/assignment-1-random

2 Introduction

In this assignment we will compare our assembly code against a compiler. To do this we will implement a micro-benchmark in C and x86 assembly and then compare their performance. We have selected to micro-benchmark open source merge-sort program[1]. We selected this program because it represent a realistic simplified version of a larger program, containing functions and several loops. Merge-sort is a comparison-based sorting algorithm that divides the input array into two parts, then divides the two parts again until all the numbers are split. and then merges the two sorted halves together in order. To stress the cpu and makeinclude ¡sys/time.h¿ a realistic representation of a program we altered the sorting to make 1 million numbers from 1 to 2 million.

C is generally considered a language close to the computer. Assembly, on the other hand, is a language that is closer to the computer compared to C. This is something to take into account when assessing our expectations of the results. It is to be expected that the hotspot written in assembly will run faster compared to the C function if implemented correctly.

2.1 Technical background

Assembly has things in common from other languages such as operations. In C there are operations like if, while, for and so on.. The operations used in assembly are just more technical. You can move and store values and addresses from registers with the movl and leal operators. Add and subtract with the addl and subl operators and so on. There are too many of these operations to touch on that these will not be included in the report, but rather merely mentioned so that you can understand the next paragraph better.

The first thing to understand when diving into x86 assembly is what registers are and how they work. Registers are internal memory storage in the processor to speed up the process. This is there to not have the need to fetch the memory externally which would take more time. There are limited amount of registers in the processor and they are following: Below is a picture [2] of what kind of registers there are, the most commonly used registers, seem to be the 4 first ones. However all the general-purpose registers work the same way for the most part. The two last registers are special cases, one is the stack pointer and the other is the base pointer. ESP (stack pointer) is a indirect memory operand and points to the top of the stack at any time. EBP (base pointer) provides a base pointer to the current function so that all the parameters and local variables are kept at a offset from the base pointer.

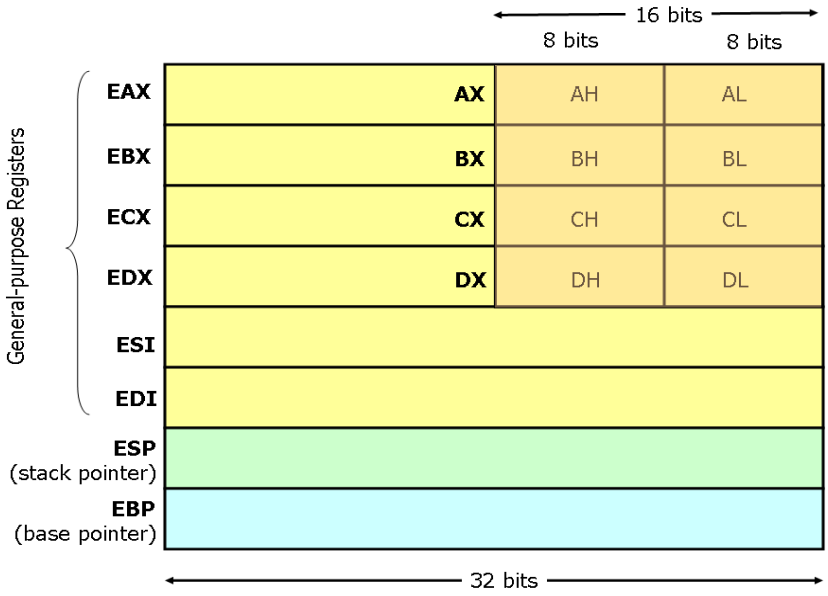

Illustration 1: Registers in the processor. [2].

3 Implementation

The C implementation is just below 30 lines with three integer declarations, three while loops and one if check. By this point of the program the array and all sub arrays are already split up, what the hotspot part does is it merges the the sub arrays into larger arrays, until there's only a single array in a sorted order.

To convert this C implementation into the assembly language, the first step would be to think about the layout of the function. Going through the C implementation sequentially and figuring out, step by step, what comes next. The way assembly reads the script is it starts from the top and goes down line by line. The few first lines of the assembly script sets up the stack frame, after this the actual code that is going to be converted starts. The three first lines of the C code are integer declarations, to do this in assembly we have to set the corresponding values into different addresses in the base pointer.

After that comes the loop so the sequence jumps to the next function/section of the script. Before the condition of the loop can be checked the value of the correct base pointer address has to be moved to one of the registers (eax, ebx, exc or edx) and then compare that value to the value of another base pointer address. The line below this comparison is the condition itself. If the condition

doesn't meet then the line is skipped. If the condition is met then this line breaks out of the loop and jumps to the next while loop lower in the script. Since this loop has two conditions this process is repeated again with different values from different base pointers. If neither of the conditions are met then the sequence just keeps going to the next function.

Next few parts work with arrays, all the operations inside every loop are very similar. To set up the arrays we have to move the arrays that are saved in memory from base pointer to one of the general purpose registers, this does not index the arrays. To index an array lea (load effective address) operation is used on the integer that is being set as the index to the array and is also moved to one of the registers. The registers with the array and the index are then added together with addl operation and the array is then fully set up. This needs to be done separately with all arrays. From there on the arrays can be compared through the if check and be added to the output array, the same way it was set up to be compared. This is the last part of the loop and after this the sequence will jump back to the start of the loop.

Since all the loops are similar, the process from the previous paragraph just repeats. After all the loops have finished the sequence reaches the end of the script and is about to return to the caller. Before that is done the stack frame needs to be re-positioned by adding x amount of bytes to the stack pointer.

4 Methodology

We will locate the sorting algorithms "hotspot" and write this specific function in assembly. The main hotspot in the micro-benchmark is the portion of the code most of the execution time is spent. We re-implement and re-structured the code to simplify and shorten the location of the hotspot. The method used to locate the hotspot was to profile the original code and locate the function that used most execution time.

The specs of the computers used were different. And the results of the profiling were quite different because of the hardware gap. The cpu's used are:
Ryzen FX-6350 - 6 core processor
i9 10850K - 8 core processor

The hotspot measuring was done by gprof-profiling. The profiling did not show any time used on small samples to relatively larger samples like arrays up to a 100 thousand, so the sample size for the arrays had to be very large. However, there are some problems trying to use very large arrays and that is stack overflow. The largest array that could be used without segmentation fault was just above a million, so the array size we ended up using was a rounded million. There are ways to work around this but the sample size was big enough to experiment.

In the profile layout we are interested in time spent for each function. The functions name is located to the rightmost side of the figure. They are arranged so that the functions that spend the most time on execution will end up at the top of the list. We are particularly interested in measuring the time category "percent time".

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 63.64      3.08     3.08 29999970     0.00     0.00  c_function
 27.07      4.39     1.31 29999970     0.00     0.00  merge
  4.55      4.61     0.22       60     3.67     3.67  printArray
  3.93      4.80     0.19       30     6.33   152.67  mergeSort
  0.41      4.82     0.02       30     0.67     0.67  CreateArray
  0.41      4.84     0.02                             __x86.get_pc_thunk.bx
```

Illustration 2: Profile for hotspot.

Illustration 2 above shows which function spends the most time when we profiled it. The c-function uses 63.6 percent of the time. This feature is our hotspot and the code we translated into assembly. To get a better measuring point for time, we use the library-function gettimeofday() to measure the time. The main loop is iterated 100 times, but only the time before and after the target functions are recorded and added to a global variable. In this way, each function takes some time to complete and the differences in both processors and functions is better displayed.

5 Results

When we run the programs there will be small variations depending on how much work the cpu is loaded with from other programs. To adjust for such variations, we have terminated other processes and run the program several times on each processor to control for out-layers. The profiles we have presented in "results" are the ones that are most representative of the average of the runs, excluding out-layers. The main loop is iterated 100 times, which amounts to one hundred million array numbers sorted. The denomination is displayed in seconds.

Ryzen FX-6350 - 6 core processor

C-function:
Total: 22.020257 s
Average: 0.220202 s

Assembly-function:
Total: 21.095936 s
Average: 0.210959 s

i9 10850K - 8 core processor

C-function:
Total: 12.060899 s
Average: 0.120609 s

Assembly-function:
Total: 11.625739 s
Average: 0.116257 s

6 Discussion

We selected merge-sort as our program to micro-benchmark. Location of the programs hotspot was found by profiling the sorting code and find the function that took most time to execute. This function was split into two functions (mrg2 and mrg3) and profiled again. Function "mrg2" was the now the "final" hotspot and the function we wrote in assembly, however, mrg3 was so insignificant to the whole program that we ended up adding it back to mrg2 to write a bit longer assembly code.

As mentioned in the introductory part, we expect the assembly function to be faster than the C function due to the fact that it is a language that is closer to the machine. The result shown above tells us that the assembly function runs marginally faster than the C-function on both processors. This was also the case for the "control" runs, adjusting for out-layers. All the measured numbers here on are the average time over multiple runs for each iteration of the the hotspot function. For the Ryzen 6 core processor the difference is 10 ms favoring the assembly code. For the i9 -8 core processor the difference is 4 ms favoring the assembly code.

It may be appropriate to have a margin of error in milliseconds as there are small margins that affect performance. If we set the margin of error at 10 ms, then we must conclude that there is no clear difference between the two programs for both processors. However, if we put the margin of error at 5 ms, we

can say that for Ryzen - 6 core processor there is a difference in time between assembly function and C-function. Both these suggested margin of error will make the claim that there is no difference in speed for assembly- and C-function when processed through the i9 - 8 core processor.

In conclusion, the assembly-function run 10 ms faster than the C-function in Ryzen -6 core processor. For the i9 -8 core processor the difference of 4 ms favoring the assembly code, which is not enough to draw any conclusions for this processor. Both timers give marginally time-difference in this benchmark, and are most depending on how much stress the cpu is exposed to from other programs. Involving only one function of a code, the difference will most likely increase if we swapped more features with assembly language or had a larger function translated to assembly.

7 Reference

1. [Internet]. 2021 [cited 14 September 2021]. Available from:
https://www.citethisforme.com/cite/sources/websiteautociteconfirm

2. [Internet]. Cs.virginia.edu. 2021 [cited 15 September 2021]. Available from:
https://www.cs.virginia.edu/ evans/cs216/guides/x86-registers.png