Laboratory 2

The following graph shows the difference in execution time in cycles for different types of operations in assembly. I took 10 experiments.



The operations for adding(ADD) two values from local registers or adding a register and variable have alike values.

The multiplication (MUL) takes a bit longer than ADD, because the operation is a bit more complex.

Fdiv and Fsub take almost double the time than MUL, with Fdiv taking a bit more time.

I implemented the algorithms in the C file. First, I filled the static and dynamic arrays with 50.000 random values from 0 to 100. I used bubble sort for both.

The classic bubblesort is the one with the 2 fors,one from 0 to n and another from 0 to n-1.

The optimized bubblesort has 2 fors, one from 0 to n-i and one flag. This parts optimize the code a lot because the bubblesort doesn’t do so many useless comparisons and the flag saves time by stopping the algorithm when it is already sorted.

The frequency of my processor is (unfortunately not so great) 1Ghz which means 1 000 000 000 hertz.

I took 10 experiments and calculated the average for execution time in cycles and seconds.

**Bubblesort classic static array**

21314105362 cycles-21,314105362 seconds

21284820067 cycles-21,284820067 seconds

21529752630 cycles-21,529752630 seconds

21195566194 cycles-21,195566194 seconds

21439340796 cycles-21,439340796 seconds

21142774700 cycles-21,142774700 seconds

21303293333 cycles-21,303293333 seconds

21567940576 cycles-21,567940576 seconds

21988437396 cycles-21,988437396 seconds

21207128159 cycles-21,207128159 seconds

average execution time

cycles: 21397315921

seconds: 21,397315921

**Bubblesort classic dynamic array**

21509538234 cycles-21,509538234 seconds

21216719686 cycles-21,216719686 seconds

21051251728 cycles-21,051251728 seconds

21107551105 cycles-21,107551105 seconds

21064109454 cycles-21,064109454 seconds

21148912947 cycles-21,148912947 seconds

21140745435 cycles-21,140745435 seconds

21140745435 cycles-21,140745435 seconds

21158620019 cycles-21,158620019 seconds

21162637522 cycles-21,162637522 seconds

average execution time:

cycles:21170083157

seconds:21,170083157

**Bubblesort optimized static array**

16172520277 cycles-16,2520277 seconds

16233157959 cycles-16,233157959 seconds

16304860019 cycles-16,304860019 seconds

16323351640 cycles-16,323351640 seconds

16320826909 cycles-16,320826909 seconds

16188175725 cycles-16,188175725 seconds

16212766159 cycles-16,212766159 seconds

16336866752 cycles-16,336866752 seconds

16472786859 cycles-16,472786859 seconds

16190532059 cycles-16,190532059 seconds

average execution time

cycles:16275584436

seconds:16,275584436

**Bubblesort optimized dynamic array**

16396448383 cycles-16,396448383 seconds

16315727007 cycles-16,315727007 seconds

16340475040 cycles-16,340475040 seconds

16210347426 cycles-16,210347426 seconds

16355848391 cycles-16,355848391 seconds

16293490700 cycles-16,293490700 seconds

16544793450 cycles-16,544793450 seconds

16274780060 cycles-16,274780060 seconds

16267458593 cycles-16,267458593 seconds

16498589206 cycles-16,498589206 seconds

average executon time

cycles:16349795826

seconds:16,349795826

The exact execution time is written in the lab – ‘To make an exact measurement, the overhead of CPUID can be measured and subtracted from the cycle count obtained for the measured code. The CPUID instruction takes longer to execute the first two times it is called. It is better to measure the execution of its third call, and use this value for all future measurements.’

**Bubblesort classic static array**

total cycles:20945146315 cpuid\_time: 331 exact execution time:20945145984

total cycles:21012565571 cpuid\_time: 550 exact execution time:21012565021

total cycles:21062736473 cpuid\_time: 535 exact execution time:21062735938

total cycles:20998406029 cpuid\_time: 483 exact execution time:20998405546

total cycles:21367164933 cpuid\_time: 554 exact execution time:22367164379

**Bubblesort optimized static array**

total cycles:16391255444 cpuid\_time:423 exact execution time:16391255021

total cycles:16522694424 cpuid\_time: 350 exact execution time:16522694074

total cycles:16272110148 cpuid\_time: 525 exact execution time:16272109623

total cycles:16216852680 cpuid\_time: 698 exact execution time:16216851982

total cycles:16361219648 cpuid\_time: 451 exact execution time:16361219197

total cycles:16284004527 cpuid\_time: 422 exact execution time:16284004105

**Bubblesort classic dynamic array**

total cycles:21011090129 cpuid\_time: 498 exact execution time:21011089631

total cycles:21154870188 cpuid\_time: 486 exact execution time:21154869702

total cycles:20963036820 cpuid\_time: 539 exact execution time:20963036281

total cycles:21031048857 cpuid\_time: 444 exact execution time:21031048413

total cycles:21030929395 cpuid\_time: 484 exact execution time:21030928911

**Bubblesort optimized dynamic array**

total cycles:16302966205 cpuid\_time: 428 exact execution time:16302965777

total cycles:16336248168 cpuid\_time: 656 exact execution time:16336247512

total cycles:16383498876 cpuid\_time: 517 exact execution time:16383498359

total cycles:16370537651 cpuid\_time: 558 exact execution time:16370537093

total cycles:16312475306 cpuid\_time: 1750 exact execution time:16312473556

Whatever the original or optimized version of the algorithm, the difference between the static and dynamic arrays are not so big.

The optimized version however is superior to the classic algorithm. It saves approximately 5 seconds.