**АРХИТЕКТУРА РАЧУНАРА**

**ПРВИ ПРОЈЕКТНИ ЗАДАТАК**

**АСЕМБЛЕРСКИ ПРОГРАМ ЗА ОБРАДУ ПОДАТАКА**

ТЕМА: Израчунавање укупне количине простих бројева у опсезима из специфицираног скупа опсега

Аутор: Гордан Летић

Верзија: 1.0

Датум: 31.12.2022

Садржај:

Увод ………………………………………………………………………………………………………………… 3

Поступак и услови тестирања ………………………………………………………………………… 3

Поређење времена извршавања …………………………………………………………………… 4

Поређење времена извршавања-Оптимизације gcc компајлера ………………… 8

Сликовити приказ оптимизације употребом SSE инструкцијског скупа ……… 11

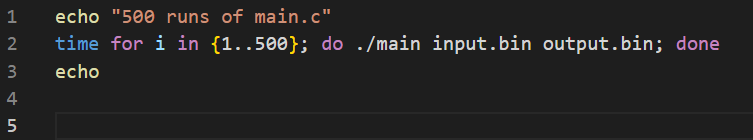
Закључак …………………………………………………………………………………………………………. 12

**УВОД**

Пројектни задатак на тему асемблерске обраде података реализован је на алгоритму за израчунавање укупне количине простих бројева у опсезима из специфицираног скупа опсега. Алгоритам је реализован, како је задатком дефинисано, у асемблерском језику са стандардним инструкцијским скупом х86\_64 намјењеним за линукс оперативни систем, након тога је извршена оптимизација алгоритма увођењем SSE инструкцијског скупа за паралелизацију, те на крају је реализован алгоритам у С програмском језику. Сва три рјешења су тестирана и упоређена на неколико различитих улазних података, као и различитог броја извршавања, те различитих врста компајлерских оптимизација за С програм.

**ПОСТУПАК И УСЛОВИ ТЕСИРАЊА**

Све реализације алгоритама су тесиране помоћу *schell* скрипте која покреће дати програм задати број пута.

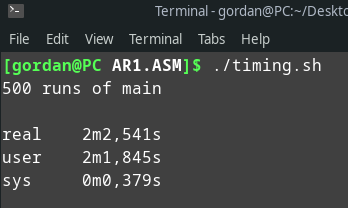


*Слика 1. Примјер садржаја schell скрипте за мјерење времена извршавања*

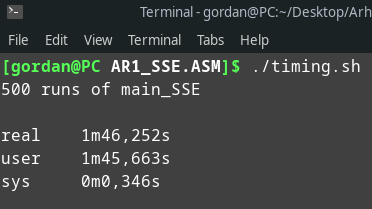
Алгоритми су извршавани на процесору Intel i7-4510U, на оперативном систему Manjaro Linux (Arch дистрибуција). Дати процесор има 2 физичке језгре и 4 логичке (Hyper Theading), а основна фреквенција рада је 2.00GHz, а може да иде и до 3.10GHz при већем оптерећењу.

**ПОРЕЂЕЊЕ ВРЕМЕНА ИЗВРШАВАЊА**

Поређење времена извршавања је представљено у виду screenshot-ова, на којима се јасно виде разлике у временима извршавања над различитим реализацијама алгоритама, као и различитих улазних података.

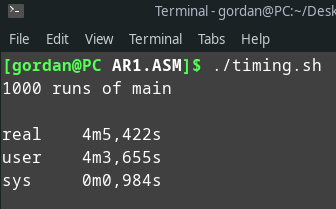


*Слика 2. Х86\_64: У улазном фајлу 100 опсега од 1 до 1000*

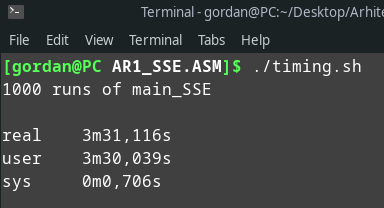


*Слика 3. SSE: У улазном фајлу 100 опсега од 1 до 1000*

Очигледно је убрзање од приближно 16 секунди кориштењем SSE инструкцијског скупа.

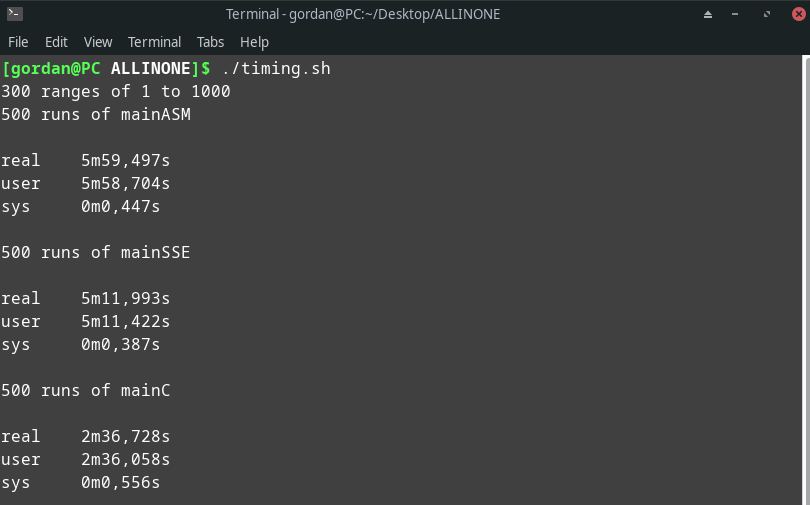


*Слика 4. Х86\_64: У улазном фајлу 100 опсега од 1 до 1000*

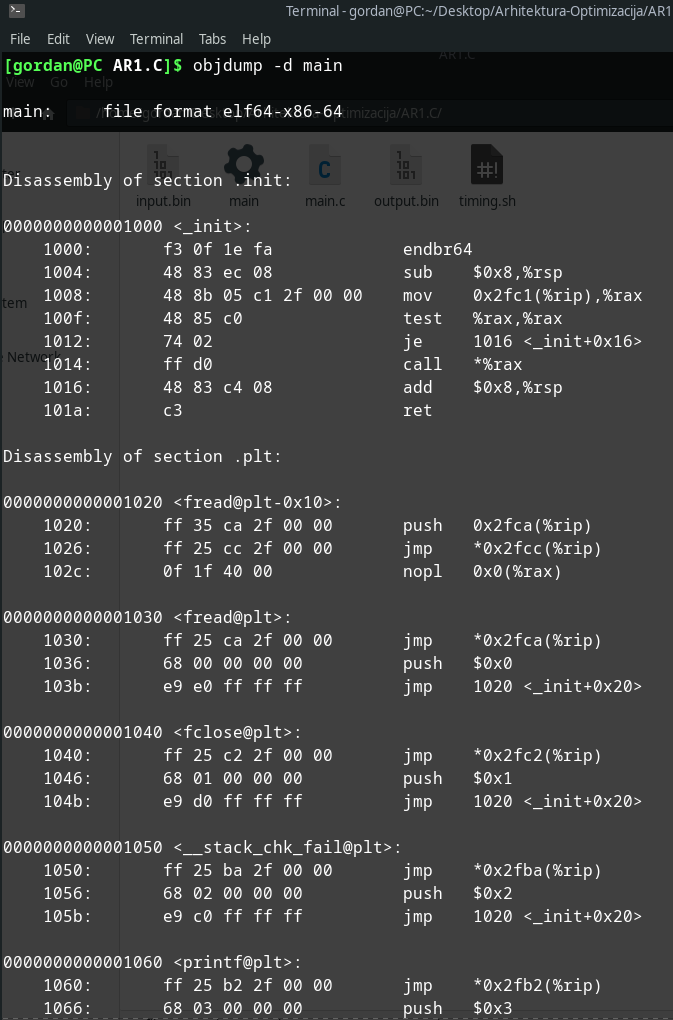


*Слика 5. SSE: У улазном фајлу 100 опсега од 1 до 1000*

Очигледно је убрзање од приближно 34 секунде кориштењем SSE инструкцијског скупа, гдје видимо да је резултат пропорционалан првом мјерењу што је и очекивано.

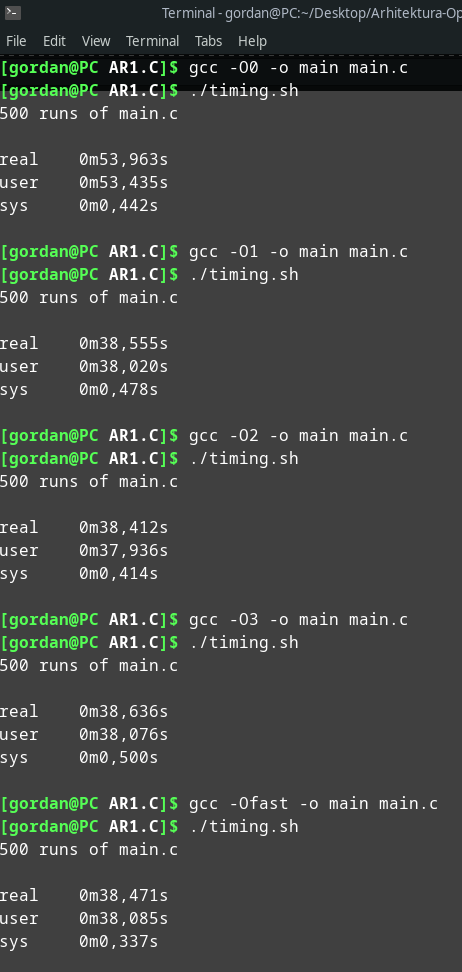
*Слика 6. Поређење: У улазном фајлу 300 опсега од 1 до 1000*

Повећањем броја улазних елементата (опсега) на 300 уочавамо приближно убрзање од 48 секунди са кориштењем SSE инструкцијског скупа. Занимљиво да када погледамо вријеме извршавања С програма он се извршавао најбрже у свим случајевима. Када деасемблирамо С програм схватамо да је компајлер генерисао асемблерски код који је доста другачији од нашег, те је то највјероватније разлог за брже извршавање овог програма у односу на нашу имплементацију асемблерског кода.

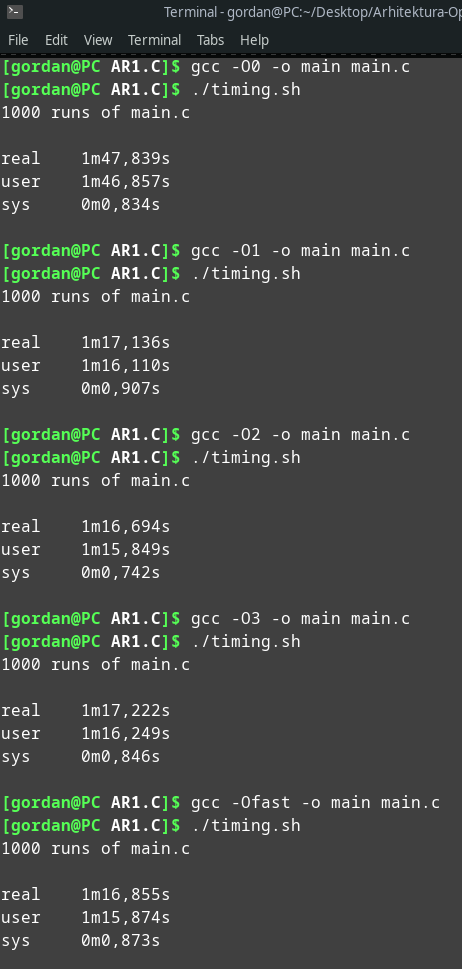
*Слика 7. Деасемблирани С програм*

**ПОРЕЂЕЊЕ ВРЕМЕНА ИЗВРШАВАЊА – ОПТИМИЗАЦИЈЕ GCC КОМПАЈЛЕРА**

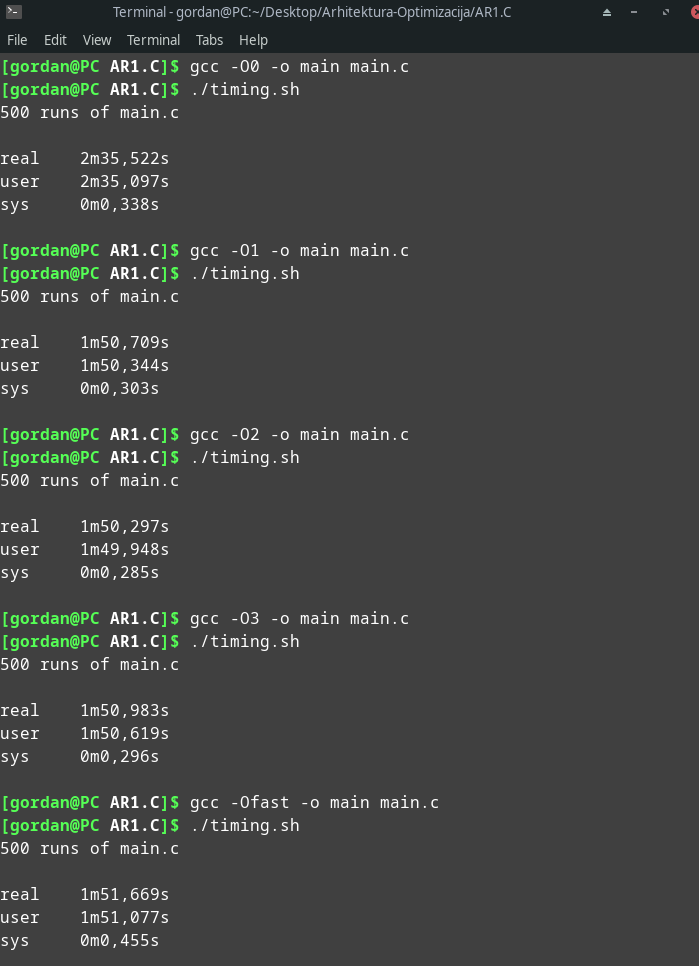
Такође и за поређење времана извршавања при различитим компајлерским оптимизацијама, резултате извршавања ћу приложити у виду screenshot-ова на којима се јасно види које оптимизације су искориштене.



*Слика 8. С програм: У улазном фајлу 100 опсега од 1 до 1000*



*Слика 9. С програм: У улазном фајлу 100 опсега од 1 до 1000*

*Слика 10. С програм: У улазном фајлу 300 опсега од 1 до 1000*

Као што видимо са приложених слика увођењем компајлерских оптимизација добијам о мање вријеме извршавања. С тим да након -О1 оптимизације вријеме извршавања је готово идентично. Разлог за то је вјероватно сам алгоритам у коме компајлер нема више шта да оптимизује, а за неки други програм ово не мора бити случај.

**СЛИКОВИТИ ПРИКАЗ ОПТИМИЗАЦИЈЕ УПОТРЕБОМ SSE ИНСТРУКЦИЈСКОГ СКУПА**

Узмимо за примјер да је број 15 број који провјеравамо да ли је прост.

Нпр. Xmm0:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | 15 | 15 | 15 |

Xmm1:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 4 | 5 |

Xmm0 mod Xmm1:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 3 | 0 |

Итерацијом кроз резултат провјеравамо да ли је остатак при дјељењу једнак нули, уколико јесте инкрементујемо бројач. Овакав поступак понављамо скроз до n/2 елементата. Уколико је бројач једанк нули број је прост. У овом случају сљедећа итерација ће изгледати овако:

Xmm0:

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | 15 | 15 | 15 |

Xmm1:

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 7 | X | X |

Xmm0 mod Xmm1:

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 1 | X | X |

У овом случају нас вриједности поља означених са Х не занимају.

**ЗАКЉУЧАК**

Након свих тестирања и упоређивања резултата долазимо до закључка да су компајлери много напредовали, те да је наш програм само скуп инструкција којима указујемо шта желимо постићи само листа жеља. Док ће копајлер узети наш програм, пресложити и оптимизовати на начин да омогући што боље вријеме извршавања. Овакав случај се често сусреће и у пракси, гдје поједине дистрибуције линукса (Arch) компајлирају комплетан програм на нашем хардверу како би се омогућиле и постигле што боље перформансе. Осим тога закључујемо да употребом векторског инструкцијског скупа можемо ефикасно оптимизовати извршавање нашег програма, што у суштини и сам компајлер ради.