FMA benchmark

Бенчмарк направлен на тестирование операций FMA (Fused-Multiply Add, умножение-сложение): fmadd и fmsub на разных платах в двух вариантах: как функция на языке C и как ассемблерная инструкция (с помощью ассемблерных вставок) Платы: LicheePi 4A; Banana Pi BPI-F3

Lichee

```
4 cores

cpu-freq        : 1.848Ghz
cpu-icache      : 64KB
cpu-dcache      : 64KB
cpu-l2cache     : 1MB
cpu-tlb         : 1024 4-ways
cpu-cacheline   : 64Bytes
cpu-vector      : 0.7.1
```

Banana

```
8 cores

CPU(s) scaling MHz:  100%
  CPU max MHz:         1600.0000
  CPU min MHz:         614.4000
Caches (sum of all):
  L1d:                 256 KiB (8 instances)
  L1i:                 256 KiB (8 instances)
  L2:                  1 MiB (2 instances)
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                4
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                4
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  524288
LEVEL2_CACHE_ASSOC                 16
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  0
LEVEL3_CACHE_ASSOC                 0
LEVEL3_CACHE_LINESIZE              0
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Основное тело цикла:

```
double fmadd(int32_t marg, volatile double darg, double aarg, double barg,
             double carg) {
    for (int32_t i = 0; i < marg; i++) {
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
        darg = aarg + barg * carg;
    }
    return darg;
}
```

Ассемблерная инструкция:

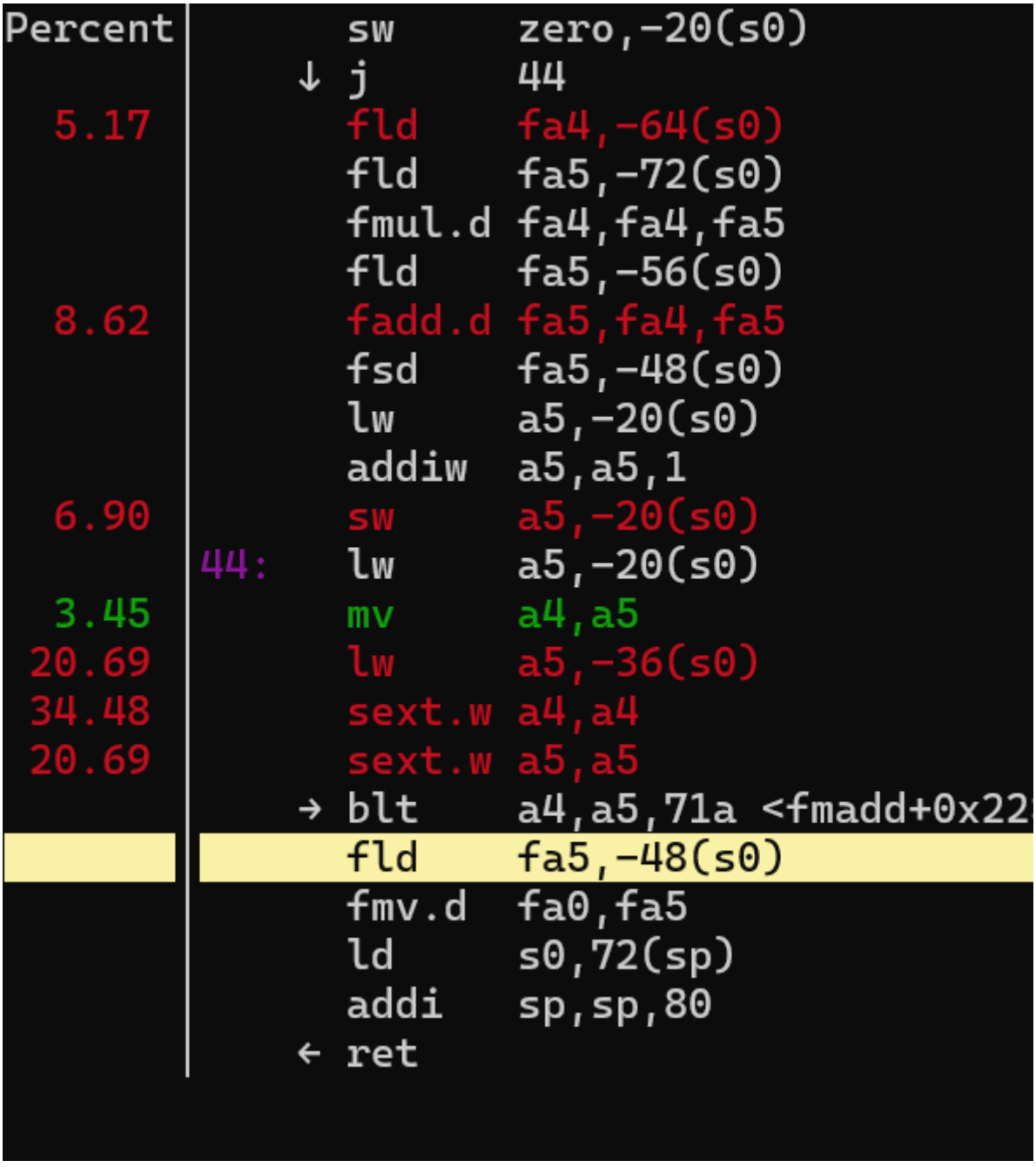
```
double fmadd(int32_t marg, volatile double darg, double aarg, double barg,
             double carg) {
    for (int32_t i = 0; i < marg; i++) {
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
                     : "f"(barg), "f"(carg), "f"(aarg));
        asm volatile("fmadd.d %0, %1, %2, %3"
                     : "=f"(darg)
```

```
                    : "f"(barg), "f"(carg), "f"(aarg));
    }
    return darg;
}
```

(аналогично для fmsub)

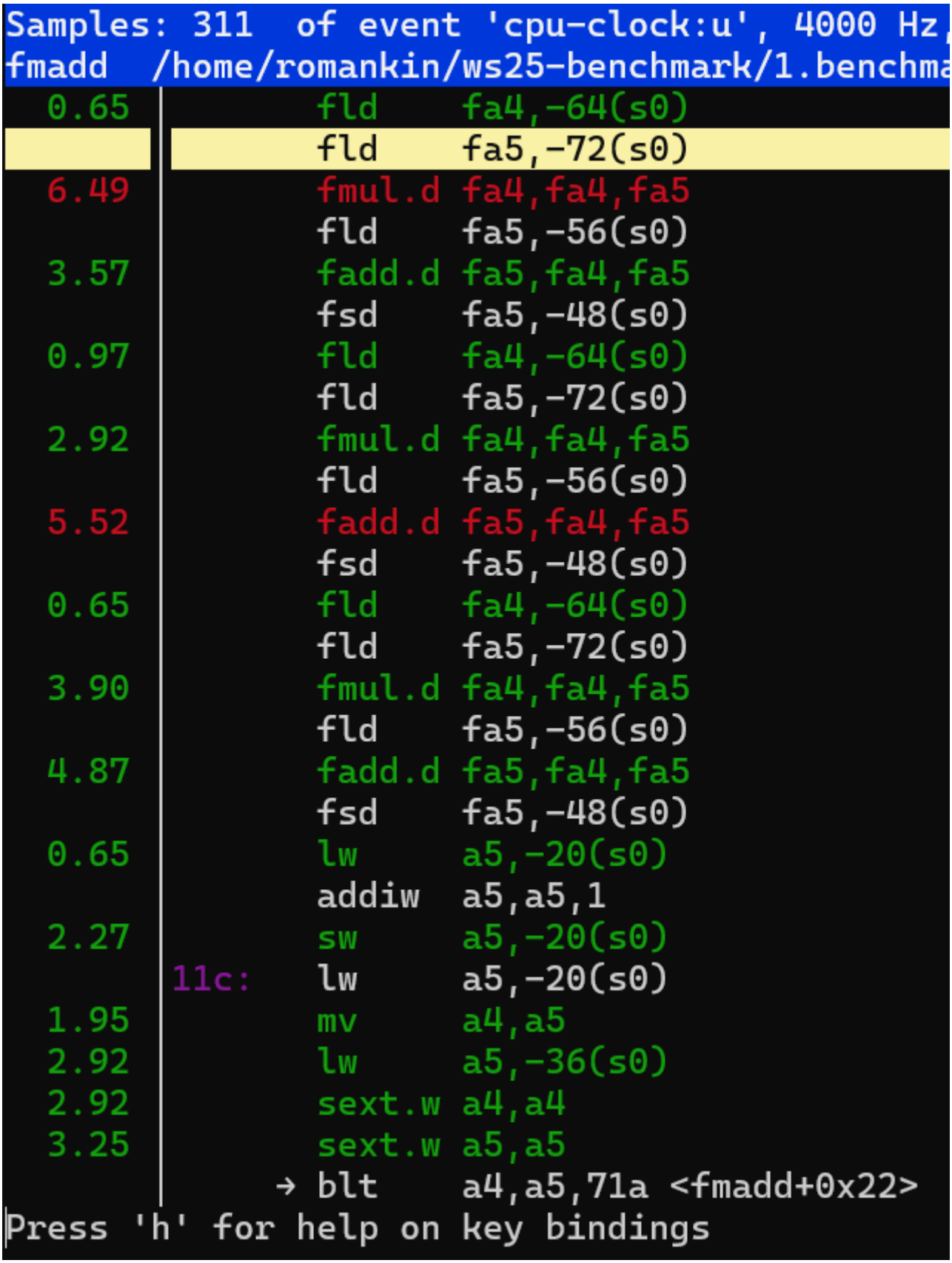
Дублирование кода инструкций в телах циклов необходимо для того, чтобы наиболее "горячими" были инструкции, действительно относящиеся к функции (т.н. loop unrolling) (в данном случае - `fadd.d` и `fmul.d`).

Рассмотрим на примере функции fmadd: Так выглядит профилировка с одной операцией сложения в цикле:



Видно, что большую часть работы занимают инструкции `sext.w` и `lw`, в то время как нужные нам инструкции `fadd.d` и `fmul.d` занимают менее 9% от всего времени работы

Если же продублировать операцию сложения:



Инструкции `sext.w` и `lw` выполняются меньшее кол-во времени, а инструкции сложения и умножения в сумме занимают ~77%, поэтому такой код можно использовать для тестирования

Горячий код (с ассемблерными вставками) (~92%):

```
Percent |       fld     fa2,8(sp)
        |       sext.w  a4,a5
        |     ↓ blez    a5,d0
        |       auipc   a5,0x0
        |       fld     fa3,916(a5) # c18 <_IO_stdin_used+0x30>
        |       auipc   a5,0x0
        |       fld     fa4,916(a5) # c20 <_IO_stdin_used+0x38>
        |       auipc   a5,0x0
        |       fld     fa5,916(a5) # c28 <_IO_stdin_used+0x40>
        |       li      a5,0
  2.61  |       fmsub.d fa2,fa3,fa4,fa5
  6.54  |       fmsub.d fa2,fa3,fa4,fa5
 13.07  |       fmsub.d fa2,fa3,fa4,fa5
  6.54  |       fmsub.d fa2,fa3,fa4,fa5
  7.84  |       fmsub.d fa2,fa3,fa4,fa5
 11.11  |       fmsub.d fa2,fa3,fa4,fa5
  6.54  |       fmsub.d fa2,fa3,fa4,fa5
 13.07  |       fmsub.d fa2,fa3,fa4,fa5
 11.11  |       fmsub.d fa2,fa3,fa4,fa5
 12.42  |       fmsub.d fa2,fa3,fa4,fa5
  9.15  |       addiw   a5,a5,1
        |     → bne     a4,a5,8a0 <main+0xa0>
        | d0:   fsd     fa2,8(sp)
        | d4:   ld      ra,40(sp)
```

Окружение:

- Сборка и ключи компиляции

```
gcc -g main.c -o main -O0 -pg

gcc  -fno-verbose-asm -march=rv64id main_asm.c -o main_asm -O3 -pg
```

Ключи `-g`, `-pg` и `-fno-verbose-asm` создают дополнительную информацию, полезную при профилировке и отладке (`-pg`, в частности, создаёт `gmon.out` для профилировщика `gprof`)

Ключ `-O` устанавливает степень оптимизации компилятора

Ключ `-march=rv64id` подключает расширение RISC-V для работы с double

- Скрипт для просмотра горячего кода (аналогично для бенчмарка с ассемблерными вставками)

```
cd ../
perf record -e cpu-clock ./main $1 $2
perf report
```

- Скрипт для замеров и записи значений в отдельный файл (аналогично для бенчмарка с ассемблерными вставками)

```
#!/bin/bash
cd ../
rep=20
iter=100000
n=10000000
gcc -g main.c -o main -O0 -pg
if [[ $1 == 1 ]]; then
    for ((i = 1; i < $n + 1; i+=$iter))
    do
    perf stat -o out_fmadd.txt -r $rep --table ./main $1 $i
    grep ") #" out_fmadd.txt > out2_fmadd.txt
    cat out2_fmadd.txt | cut -d"(" -f1 >> out3_fmadd.txt
    done
else
    for ((i = 1; i < $n + 1; i+=$iter))
    do
    perf stat -o out_fmsub.txt -r $rep --table ./main $1 $i
    grep ") #" out_fmsub.txt > out2_fmsub.txt
    cat out2_fmsub.txt | cut -d"(" -f1 >> out3_fmsub.txt
    done
fi
```

Программа для построения графиков:

```
import matplotlib.pyplot as plt
import numpy as np
import sys

CONST = int(sys.argv[1])
file = open("out3_fmadd.txt", "r")
file2 = open("banana_out3_fmadd.txt", "r")
x = []
y = []
y2 = []
buf = []
min_val = 0
count = 1;
c = 0

for line in file:
    buf.append(float(line))
    c += 1
    if c == CONST:
        min_val = min(buf)
        y.append(min_val)
        x.append(count)
        count += 1000000
```

```
            c = 0
            buf = []
    buf = []
    c = 0
    min_val = 0
    for line in file2:
        buf.append(float(line))
        c += 1
        if c == CONST:
            min_val = min(buf)
            y2.append(min_val)
            c = 0

            buf = []


    plt.figure().set_figwidth(15)
    plt.plot(x, y, 'ro-', label='lichee')
    plt.plot(x, y2, 'go-', label='banana')
    plt.title("fmadd", fontsize = 20)
    plt.grid(True)
    plt.ylabel('time(s)', fontsize = 15)
    plt.xlabel('cycle iterations', fontsize = 15)

    plt.ylim([0, 1])
    plt.legend()
    plt.savefig('out_fmadd.png')
    file.close()
    file2.close()
```

Запуск: `./bench.sh <номер функции: 1 - fmadd, 2 - fmsub>` (для функций с ассемблерными вставками - `bench_asm.sh`)

Скрипт создаёт файлы вида `out3_<имя_функции>_<ассемблер>.txt`. Файлы с платы Lichee должны иметь вид `out3_fmadd.txt`, с Banana - `banana_out3_fmadd.txt`.

Эти файлы нужны для запуска программы построения графиков `test.py`

Запуск: `python test.py <значение переменной rep в скрипте для замеров>`

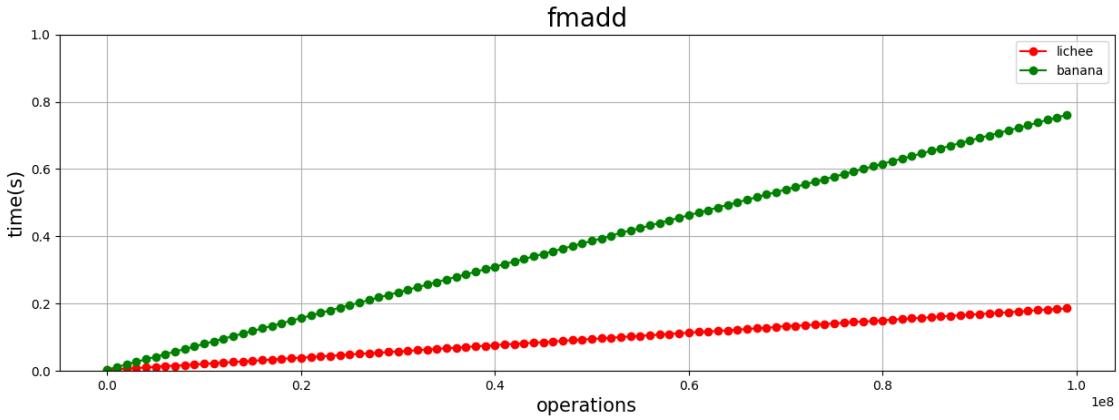
Например: `python test.py 20`

Переменная `rep` - количество запусков на некотором значении количества операций, среди которых выбирается минимальное

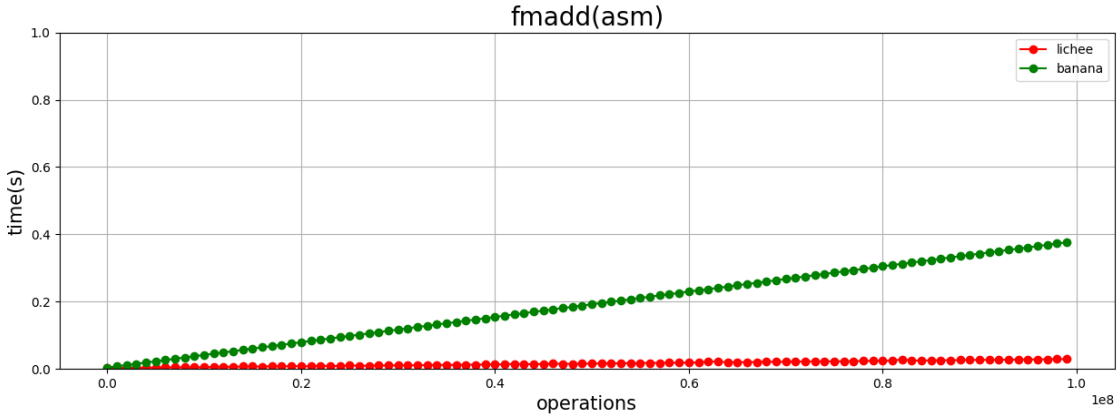
Примечание: для запуска нужна библиотека `matplotlib`

Результаты работы бенчмарка

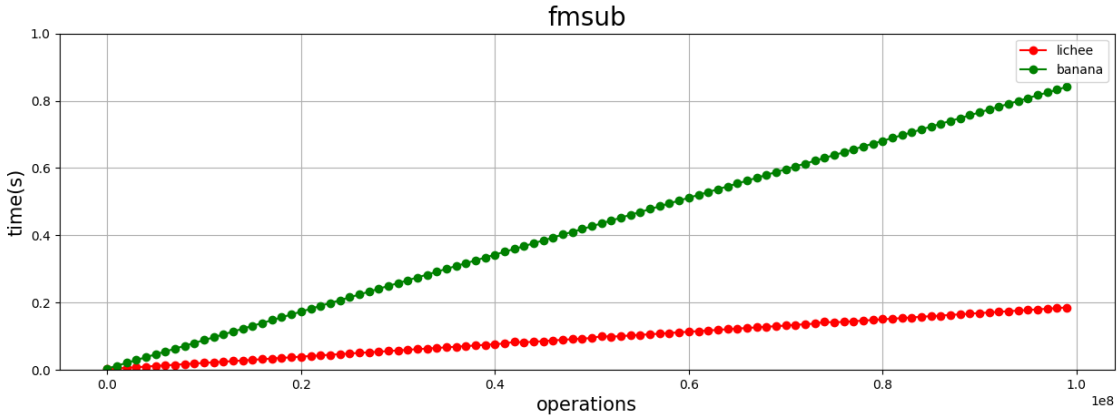
Fmadd



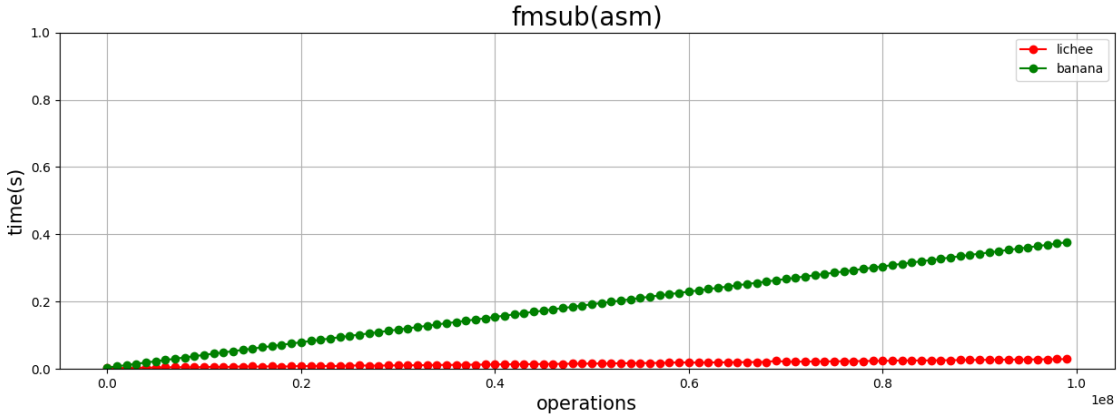
Fmadd(ассемблер)



Fmsub



Fmsub(ассемблер)


fmsub(asm)
lichee
banana
time(s)
operations
1e8
0.0
0.2
0.4
0.6
0.8
1.0