Нововведения в RISC-V

МФТИ
Осень 2024

Что мы хотим от процессора?

- Максимальную производительность

Что мы хотим от процессора?

- Максимальную производительность
- При, по возможности, минимальном энергопотреблении

Наиболее распространенные методы увеличения производительности

- Оптимизация набора команд
  - Специфические инструкции
  - Аппаратные кодеки
- Использование параллельности по данным
  - Суперскалярная архитектура
    - Несколько конвейеров в рамках одного ядра, которые могут параллельно вычислять инструкции, не зависящие по данным
  - Мультиядерная архитектура
    - Несколько независимых вычислительных ядер с общей памятью и механизмами синхронизации
  - Векторная архитектура
    - Широкий конвейер, выполняющий одну операцию сразу над большим количеством элементов

Мультиядерная архитектура

- Процессор состоит из отдельных вычислительных ядер
- У ядер есть приватные ресурсы:
  - Регистры
  - Конвейер
- У ядер есть общие ресурсы:
  - Память
  - Периферия
- Для обеспечения консистентности доступа к общим ресурсам используются разные механизмы и алгоритмы

Обсуждение: инкремент структуры

- Допустим стоит задача реализовать функцию, выполняющую инкремент полей структуры

```
struct Counters { int a; int b; };

void increment(atomic<Counters> &cnt) {
  Counters val = cnt.load();
  val.a += 1; val.b += 1;
  cnt.store(val);
}
```

- `atomic<T>` это что-то вроде класса-обертки, обеспечивающей атомарные `load` и `store` операции
- Что может пойти *не так*?

Инкремент структуры: проблема

```
struct Counters { int a; int b; };

void increment(atomic<Counters> &cnt) {
  Counters val = cnt.load();
  // ядро #1 модифицирует cnt
  val.a += 1; val.b += 1;
  // ядро #2 модифицирует cnt
  cnt.store(val);
}
```

- Другие потоки модифицировали структуру, пока мы инкрементировали поля

Инкремент структуры: попытка решения

- Введем допущение: инкремент может не получиться с первого раза

```
struct Counters { int a; int b; };

bool increment(atomic<Counters> &cnt) {
  Counters oldval = cnt.load();
  Counters newval { oldval.a + 1, oldval.b + 1 };
  if (cnt.load() == oldval) {
    cnt.store(val); return true;
  }
  return false;
}
```

- Все ли хорошо?

Инкремент структуры: попытка решения

```
struct Counters { int a; int b; };

bool increment(atomic<Counters> &cnt) {
  Counters oldval = cnt.load();
  Counters newval { oldval.a + 1, oldval.b + 1 };
  if (cnt.load() == oldval) {
    // другой поток все еще может попортить cnt здесь
    cnt.store(val); return true;
  }
  return false;
}
```

- Увы, кажется нам надо усилить требования на атомарность

Инкремент структуры: не блокирующий CAS

```
struct Counters { int a; int b; };

bool increment(atomic<Counters> &cnt) {
  Counters oldval = cnt.load();
  Counters newval { oldval.a + 1, oldval.b + 1 };
  return cnt.compare_and_swap(oldval, newval);
}
```

- `compare_and_swap` записывает `newval` в `cnt`, если `cnt == oldval`, и возвращает `true`, иначе записывает `newval` в `oldval` и возвращает `false`

Инкремент структуры: блокирующий CAS

- Если нельзя допустить неудачи инкремента, то это решается с помощью CAS-цикла

```
struct Counters { int a; int b; };

void increment(atomic<Counters> &cnt) {
  Counters oldval = cnt.load();
  do {
    Counters newval { oldval.a + 1, oldval.b + 1 };
  } while (!cnt.compare_exchange_strong(oldval, newval));
}
```

Инкремент структуры: блокирующий CAS

- Если нельзя допустить неудачи инкремента, то это решается с помощью CAS-цикла

```
struct Counters { int a; int b; };

void increment(atomic<Counters> &cnt) {
  Counters oldval = cnt.load();
  do {
    Counters newval { oldval.a + 1, oldval.b + 1 };
  } while (!cnt.compare_exchange_strong(oldval, newval));
}
```

- Теперь все хорошо?

CAS: ABA problem

- Метод pop неблокирующегося стека:

```
bool pop(T &Data) {
  Node *Old = Head.load();
  if (!Old)
    return false;
  while (!Head.compare_exchange_weak(Old, Old-
>Next)) // race
  …
}
```

- В этом коде есть весьма неочевидная ошибка

CAS: ABA проблема

- Метод pop неблокирующегося стека:

```
bool pop(T &Data) {
  Node *Old = Head.load();
  if (!Old)
    return false;
  // ядро 2 тоже делает pop и успевает раньше
  // ядро 7 сделало push и так получилось, что
  // Old побитово имеет то же значение
  while (!Head.compare_exchange_weak(Old, Old->Next)) //
race
  …
}
```

Суть ABA проблемы

```
Head.compare_exchange_weak(Old, Old->Next))
```

- Ожидание: проверь, что Head не изменился
- Реальность: проверь, что Head побитово не изменился
- Эта проблема называется ABA, потому что произошел переход состояний A -> B -> A, но мы считаем, что объект никак не изменялся

Решение ABA проблемы

- Прямой метод: храним указатель вместе со счетчиком циклов

```
struct Pointer {
  NodeTy *Node;
  unsigned long Num;
  bool operator==(const Pointer &) const = default;
};

std::atomic<Pointer> Head = Pointer{ nullptr, 0ul };
```

- Попробуйте покритиковать такой метод

Решение ABA проблемы

- Прямой метод: храним указатель вместе со счетчиком циклов

```
struct Pointer {
  NodeTy *Node;
  unsigned long Num;
  bool operator==(const Pointer &) const = default;
};

std::atomic<Pointer> Head = Pointer{ nullptr, 0ul };
```

- Попробуйте покритиковать такой метод
  - Занимает больше места
  - Потенциально медленно

Решение ABA проблемы: RISC-V

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
cas:
  lr.w t0, (a0) # Load original value
  bne t0, a1, fail # Doesn’t match, so fail
  sc.w t0, a2, (a0) # Try to update
  bnez t0, cas # Retry if store-conditional failed
fail:
```

Решение ABA проблемы: RISC-V

```
cas:
  lr.w t0, (a0) # Load original value
  bne t0, a1, fail # Doesn’t match, so fail
  sc.w t0, a2, (a0) # Try to update
  bnez t0, cas # Retry if store-conditional failed
```

- Когда выполняется `lr`, вместе с load происходит аппаратная резервация ячейки памяти (load reserved)
- `sc` выполняет store только если резервация не была снята
- Резервация снимается после исполнения любой инструкции, за исключением разрешенного списка
- Так же по спецификации между `lr` и `sc` разрешено иметь не более 16 инструкций

Обсуждение: мультиядерная архитектура

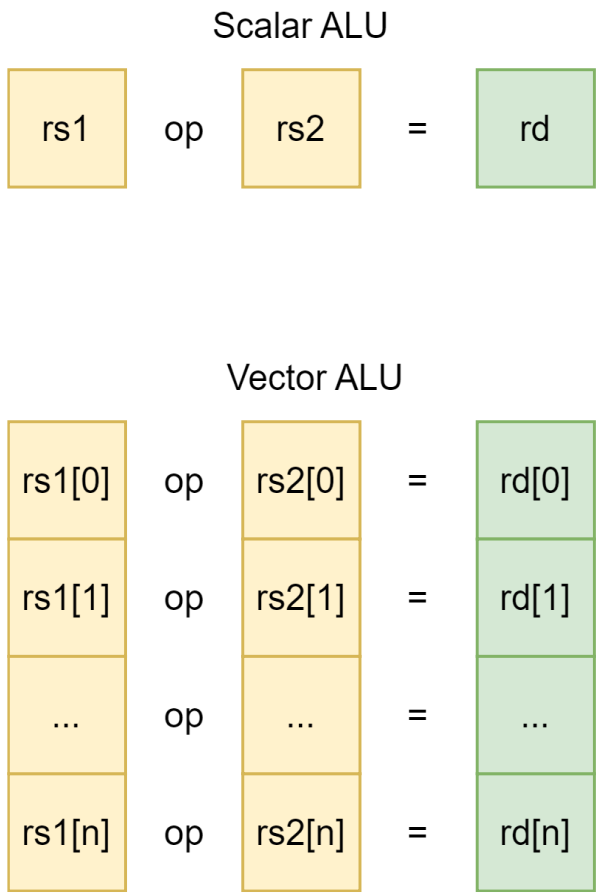
- Мультиядерная архитектура хороша для распараллеливания большого количества задач с *довольно разным* потом данных
- Рассмотрим пример:

```
int find(const int *a, int n, int x) {
  for (int i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

- Будет ли этот цикл выполнен быстрее, если его распараллелить на аппаратные потоки?

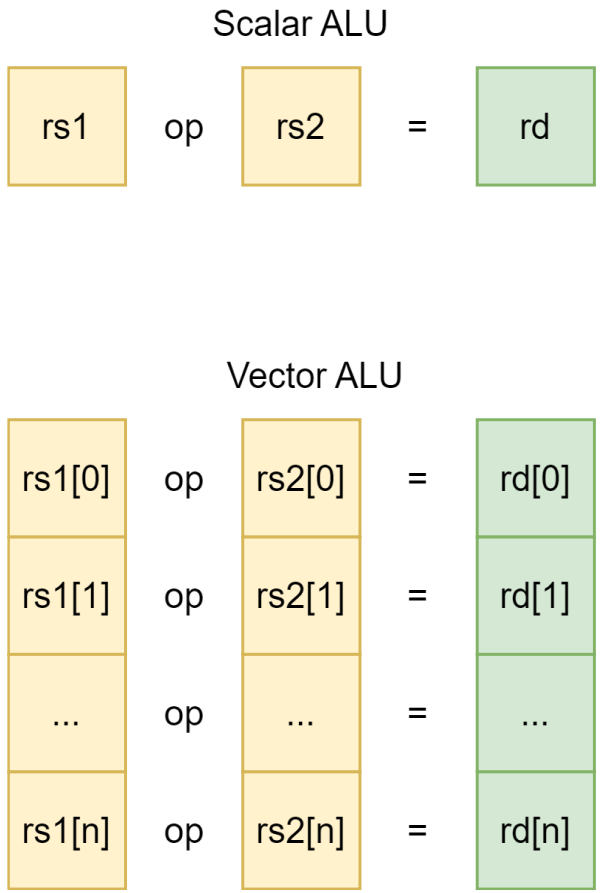
Векторная архитектура

- Подобные циклы принято ускорять другим методом – векторизацией
- Основная идея – выполнение **одной и той же** операции над широкими векторными регистрами, состоящими из элементов



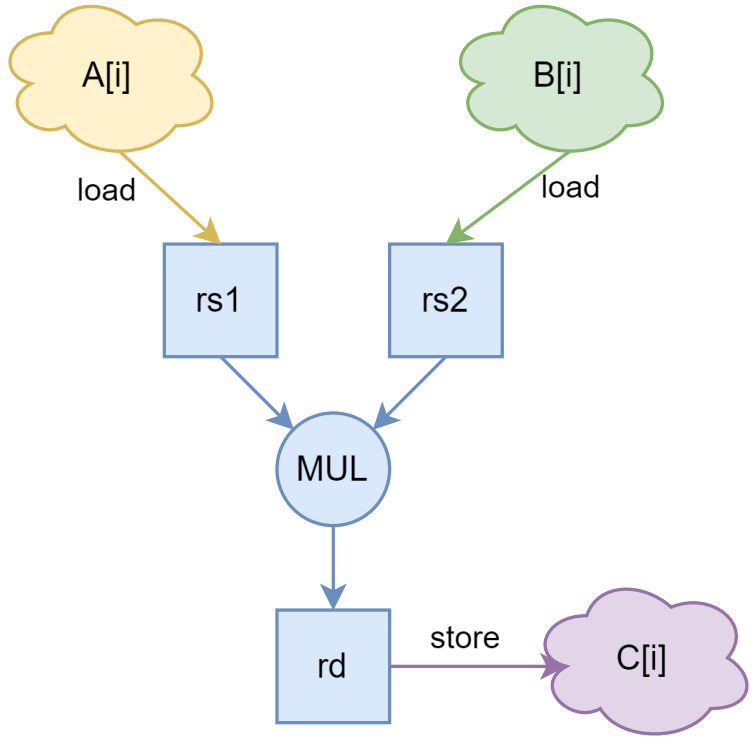
Векторная архитектура

- Подобные циклы принято ускорять другим методом – векторизацией
- Основная идея – выполнение **одной и той же** операции над широкими векторными регистрами, состоящими из элементов
- Иногда векторную архитектуру называют **SIMD** (**S**ingle **I**nstruction **M**ultiple **D**ata) архитектурой



Пример: поэлементное перемножение массива

```
for (int32_t i = 0; i < 1024; i++)
  C[i] = A[i] * B[i];
```




```
la x1, &A     # la – load address
la x2, &B
la x3, &C
li x4, 0      # li – load immediate
li x5, 1024
loop:
  lw x6, (x1)
  lw x7, (x2)
  mul x8, x6, x7
  sw x8, (x3)
  addi x1, x1, 4
  addi x2, x2, 4
  addi x3, x3, 4
  addi x4, x4, 1
  blt x4, x5, loop
```

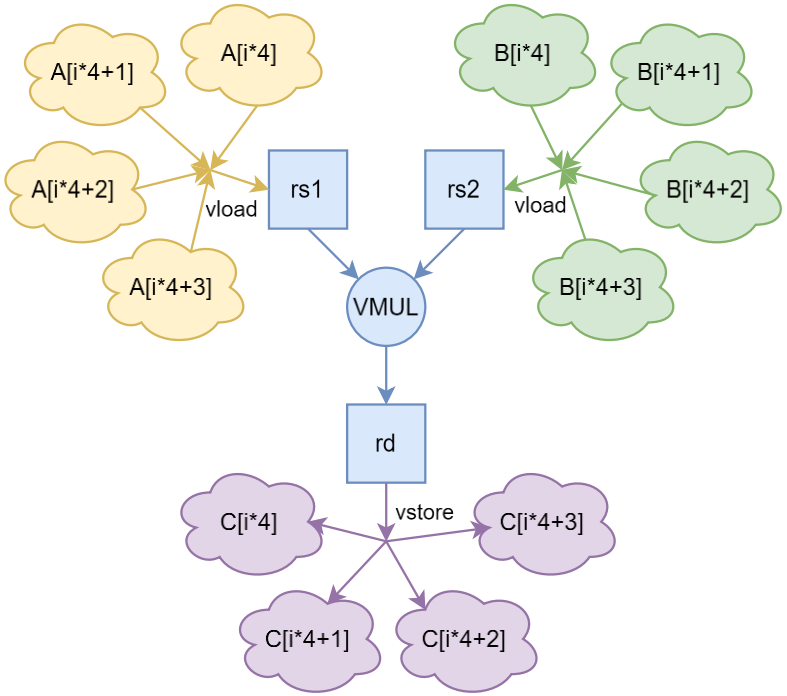
В скалярном случае будет выполнено 1024 итерации, по одному умножению за итерацию

Пример: поэлементное перемножение массива

```
for (int32_t i = 0;
     i < 1024;
     i+=4) {
  C[i*4] =
     A[i*4] * B[i*4];
  C[i*4+1] =
     A[i*4+1] * B[i*4+1];
  C[i*4+2] =
     A[i*4+2] * B[i*4+2];
  C[i*4+3] =
     A[i*4+3] * B[i*4+3];
}
```

```
la x1, &A     # la – load address
la x2, &B
la x3, &C
li x4, 0      # li – load immediate
li x5, 1024
loop:
  vlw v1, (x1) # vector load
  vlw v2, (x2) # vector load
  vmul v3, v1, v2 # vector mul
  vsw v3, (x3) # vector store
  addi x1, x1, 16
  addi x2, x2, 16
  addi x3, x3, 16
  addi x4, x4, 4
  blt x4, x5, loop
```

В векторном случае будет выполнено 256 итераций, по четыре умножения за итерацию



Векторная архитектура: пример

- Примером векторной архитектуры являются векторные расширения Intel:
  - SSE – 128-бит регистры XMM
  - AVX/AVX2 – 256-бит регистры YMM
  - AVX-512 – 512-бит регистры ZMM
- Такой подход, когда расширение добавляет регистры фиксированной длины и операции над ними называется **Fixed SIMD**

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

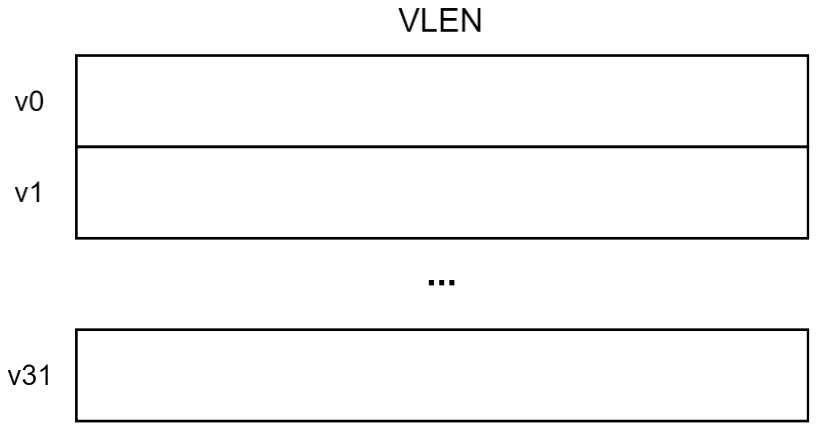
511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

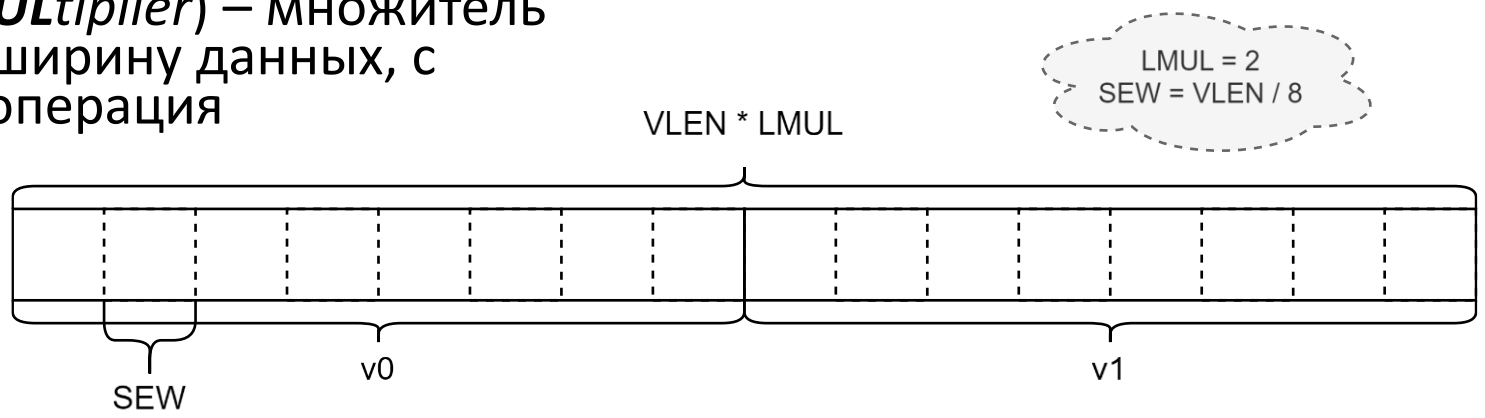
Проблемы fixed SIMD расширений

- Допустим в первом поколении было реализован набор инструкций, работающих с 256-битными регистрами
- В дальнейшем было принято решение об увеличении размера векторного регистра до 512 бит
- Несмотря на то, что инструкции будут выполнять те же операции, кодировка у них будет другой
- Fixed SIMD расширения не имеют бинарной совместимости
  - В случае автовекторизации компилятором вам надо пересобрать приложение
  - В случае ручной векторизации вам нужно заново векторизовать алогитм

RISC-V: Scalable Vectorization

- Размер векторного регистра становится деталью реализации микроархитектуры
- Абстракция с помощью нескольких параметров:
  - **VLEN** (***V****ector* ***LEN****gth*) – количество бит в векторном регистре, $2^3 \le VLEN \le 2^{16}$ (65’536)
  - **SEW** (***S****elected* ***E****lement* ***W****idth*) – количество бит в одном элементе
  - **LMUL** (*vector* ***L****ength* ***MUL****tiplier*) – множитель VLEN, определяющий ширину данных, с которой выполняется операция



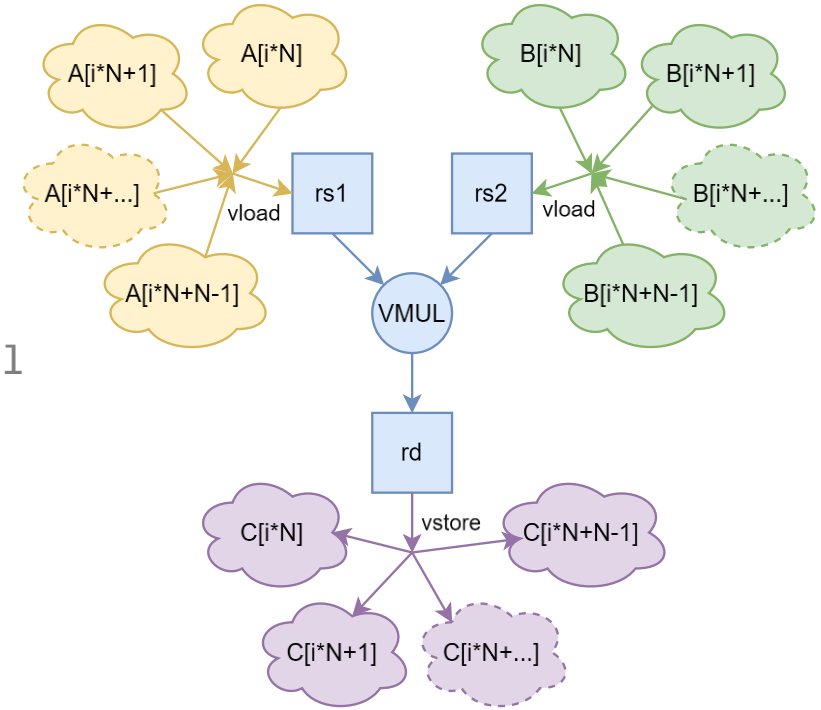


Пример: поэлементное перемножение массива

```
for (int32_t i = 0;
     i < 1024;
     i+=N) {
  C[i*N] =
     A[i*N] * B[i*N];
  C[i*N+1] =
     A[i*N+1] * B[i*N+1];
  ...
  C[i*N+N-1] =
     A[i*N+N-1] * B[i*N+N-1];
}
```

```
la x1, &A     # la – load address
la x2, &B
la x3, &C
li x4, 0      # li – load immediate
li x5, 1024
loop:
  # set vector mode
  vsetvli x6, x5, e32, m8
  vle32.v v1, (x1) # vector load
  vle32.v v2, (x1) # vector load
  vmul.vv v3, v2, v1 # vector mul
  vse v3, (x3) # vector store
  add x1, x1, x6 * 4
  add x2, x2, x6 * 4
  add x3, x3, x6 * 4
  add x4, x4, x6
  blt x4, x5, loop
```

Будет выполнено 1024/N итераций, по N умножений за итерацию в зависимости от реализации процессора



To be continued …

На следующем занятии

- Поговорим детальнее о верификации
- Разберемся, когда можно остановиться и сказать, что мы написали достаточно тестов
- Основные практики по обеспечению качественной верификации