Развивая Интринсики

Владимир Кемпик, Илья Гаврилин, Syntacore, 2024

О себе

Владимир:

- JVM Инженер в Oracle 2013-2017
- JVM Инженер в Zulu JDK в Azul 2019-2022
  - JEP 391: macOS/AArch64 Port со-автор
- JVM Инженер в JetBrains Runtime 2022
- Ведущий Инженер по Runtimes в Syntacore с 2022

Илья:

- Стажер в отделе Runtimes в Syntacore с 2023. МФТИ

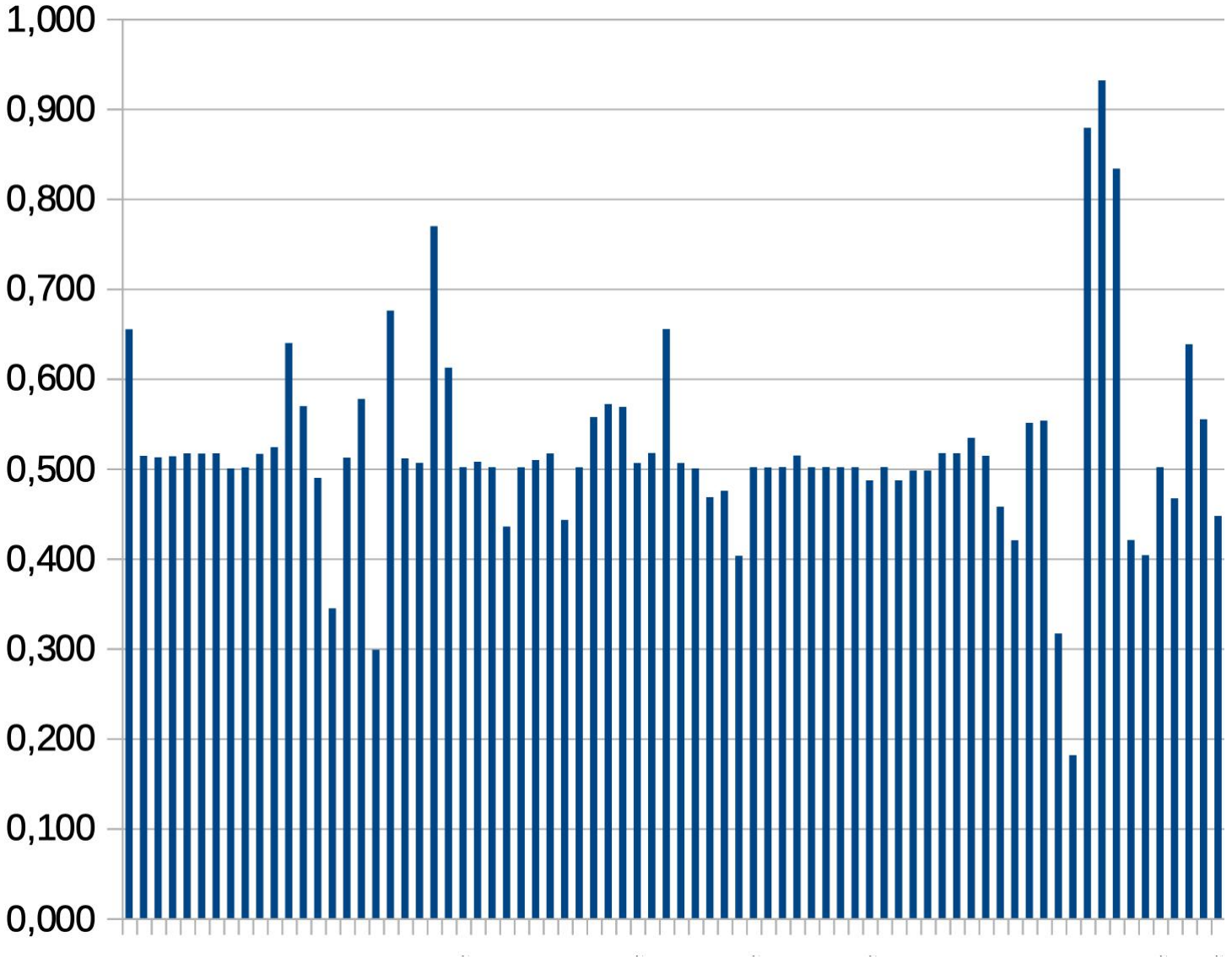
В предыдущих сериях

- RISC-V - свободная архитектура команд (ISA), родом из университета Беркли
- Развивается с 2010 года
- В 2015 для её развития был основан RISC-V Foundation
- Syntacore - Founding member в RISC-V Foundation
- Цель нашей группы - улучшение производительность managed runtimes (в т.ч. OpenJDK) под платформу RISC-V

Более подробно про RISC-V в докладе:

JVM для RISC-V - https://www.youtube.com/watch?v=_v7ddWj8buk

Сравнение производительности на микротестах



Производительность кода RISC-V в долях от производительности кода ARM64

Чего-то не хватает

Часть провалов на диаграмме объясняется отсутствием интринсиков под RISC-V

Интринсик - функция, которую JIT-компилятор может встроить вместо вызова Java-кода с целью **оптимизации**.

Два типа интринсиков в JVM

- **Library intrinsic** methods may be replaced with **hand-crafted assembly code**, with hand-crafted compiler IR, or with a combination of the two (String.Compare, String.indexOf)
- **Bytecode intrinsic** methods are not replaced by special code, but they are treated in some other special way by the compiler. For example, the compiler may delay inlining for some String-related intrinsic methods (intValue, StringBuffer_append_String)

Типы Library интринсиков

- IR/C2 nodes (remainderUnsigned_i/l, divideUnsigned_i/l, compareUnsigned_i/l, Math.ceil/floor/rint, signumF/signumD, unsignedMultiplyHigh)
- Сгенерированные функции - RuntimeStubs (poly1305.processBlocks, updateBytesCRC32)
- Вызовы в нативный (сгенерированный gcc/clang) код (sin,cos,dpow)

Поговорим о типах

- Стандартные целочисленные типы:

```
byte, short, int, long - знаковые
```

- Иногда нужен больший диапазон: криптография, CRC checksums
- Методы для беззнаковых чисел:

```
compareUnsigned(...), unsignedMultiplyHigh(...)
```

- Как реализовать `compareUnsigned` на pure java, используя знаковые типы?

compareUnsigned

```
1)public static int compareUnsigned(int x, int y);
2)public static int compareUnsigned(long x, long y);
```

JavaDoc:

Compares two **int** values numerically treating the values as unsigned.

return the value **`0`** if **`x == y`**; a value **`less than 0`** if **`x < y`** as unsigned values; and a value **`greater than 0`** if **`x > y`** as unsigned values

Как сравнить два unsigned int-а ?

```
int x = getunsignedint();
int y = getunsignedint();
long lx = x;
long ly = y;
int res = (lx < ly) ? -1 : ((lx == ly) ? 0 : 1);
```

- Нам достаточно одного **знакового** сравнения.
- А если надо сравнить два `unsigned long` ?

compareUnsigned

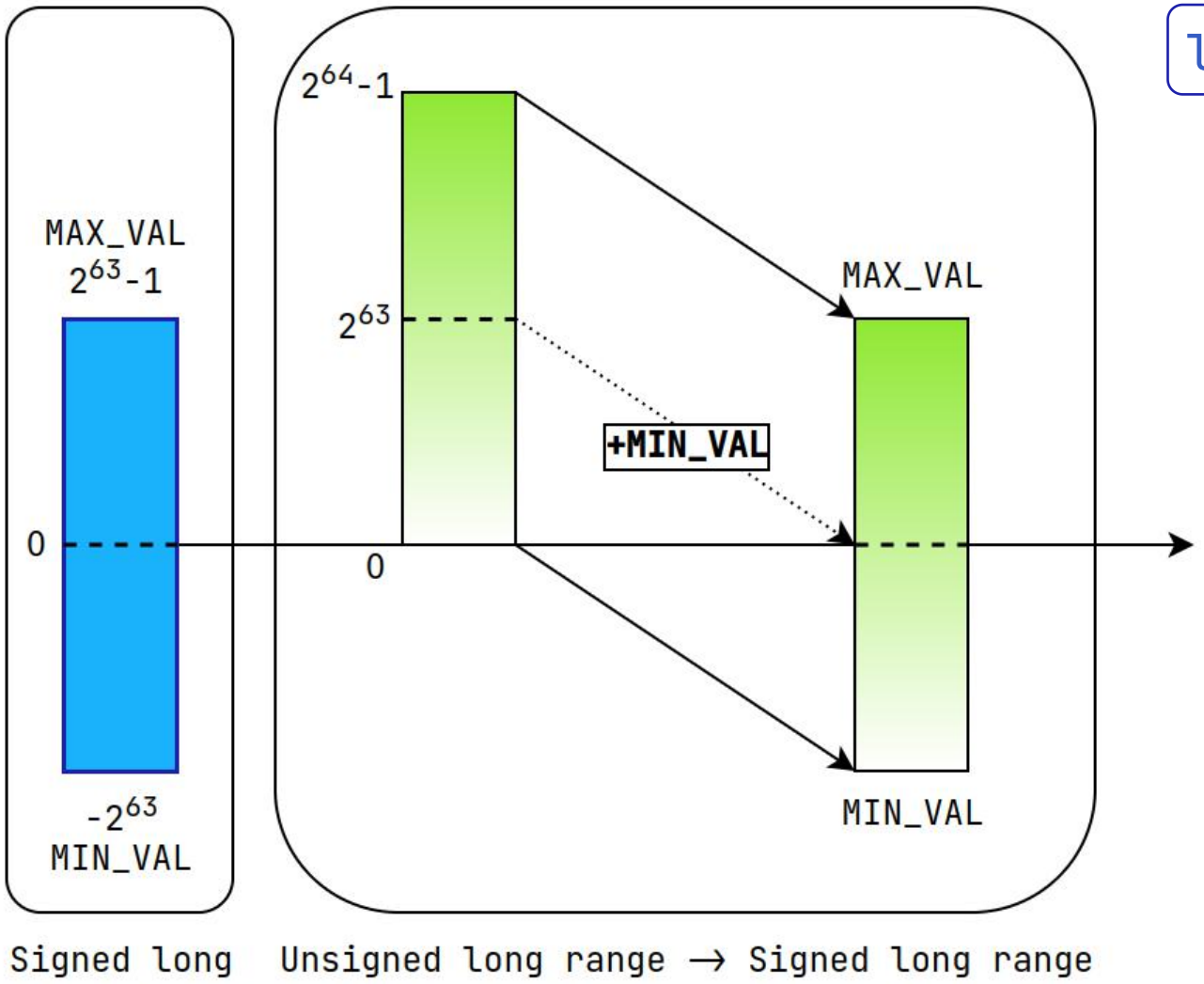
```
1)public static int compareUnsigned(int x, int y);
2)public static int compareUnsigned(long x, long y);
```

JavaDoc:

Compares two **long** values numerically treating the values as unsigned.

return the value **`0`** if **`x == y`**; a value **`less than 0`** if **`x < y`** as unsigned values; and a value **`greater than 0`** if **`x > y`** as unsigned values

Case study: Сравниваем два unsigned long




```
long MIN_VAL = 0b10....00;//-2^63
```

Unsigned long (binary) → **Signed long** (binary)	
$2^{64}-1$ (0b111...11)	$2^{63}-1$ (0b011...11)
2^{63} (0b100...00)	0 (0b000...00)
0 (0b000...00)	-2^{63} (0b100...00)

Compare unsigned - java реализация

```
long MIN_VAL = 0x8000000000000000L; //-2^63 (0b1...0)




//simple signed comparison
public static int compare(long x, long y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

Compare unsigned - java реализация

```
long MIN_VAL = 0x8000000000000000L; //-2^63 (0b1...0)
@IntrinsicCandidate
public static int compareUnsigned(long x, long y) {
    return compare(x + MIN_VAL, y + MIN_VAL);
}
//simple signed comparison
public static int compare(long x, long y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

Compare unsigned - assembly

```
compareUnsigned(Register a1, Register a2) -> Register a0
```

```
0x38:   li      a0,-1
0x3a:   slli    a0,a0,0x3f
```

a0 = Longs.MIN_VAL (-2^{63})

Compare unsigned - assembly

```
compareUnsigned(Register a1, Register a2) -> Register a0
```

```
0x38:   li      a0,-1          ┐
0x3a:   slli    a0,a0,0x3f     ┘  a0 = Longs.MIN_VAL (-2^63)
-------------------------------------------------------------
0x3c:   add     t2,a1,a0       ┐  t2 = a1 + MIN_VAL;
0x40:   add     a0,a0,a2       ┘  a0 = a2 + MIN_VAL;
```

Compare unsigned - assembly

```
compareUnsigned(Register a1, Register a2) -> Register a0
```

```
  0x38:   li      a0,-1
  0x3a:   slli    a0,a0,0x3f
  0x3c:   add     t2,a1,a0
  0x40:   add     a0,a0,a2
  0x42:   slt     a1,a0,t2
  0x46:   bnez    a1,done[0x52]
  0x4a:   slt     a1,t2,a0
  0x4e:   neg     a1,a1
done:0x52:   mv      a0,a1
```

a0 = Longs.MIN_VAL (-2^{63})

t2 = a1 + MIN_VAL;
a0 = a2 + MIN_VAL;

a1 = compare(t2, a0);

Compare unsigned - assembly

```
compareUnsigned(Register a1, Register a2) -> Register a0
```

```
  0x38:   li      a0,-1
  0x3a:   slli    a0,a0,0x3f
  0x3c:   add     t2,a1,a0
  0x40:   add     a0,a0,a2
  0x42:   slt     a1,a0,t2
  0x46:   bnez    a1,done[0x52]
  0x4a:   slt     a1,t2,a0
  0x4e:   neg     a1,a1
done:0x52:   mv      a0,a1
```

```
a0 = Longs.MIN_VAL (-2^63)

t2 = a1 + MIN_VAL;
a0 = a2 + MIN_VAL;

a1 = compare(t2, a0);
```

Note: `slt` **`rd, rs1, rs2`**
Place the value **`1`** in register **`rd`** if register **`rs1`** is **`less than`** register **`rs2`** when both are treated as signed numbers, else **`0`** is written to **`rd`**.

Compare unsigned - assembly

```
compareUnsigned(Register a1, Register a2) -> Register a0
```

```
    0x42:   sltu    t1,a2,a1
    0x46:   bnez    t1,done[0x52]
    0x4a:   sltu    a1,a1,a2
    0x4e:   neg     a1,a1
done:0x52:  mv      a0,a1
```

```
a1 = compareUnsigned(a1, a2);
```

Note: `sltu rd, rs1, rs2`
Place the value **1** in register **rd** if register **rs1** is **less than** register **rs2** when both are treated as unsigned numbers, else **0** is written to **rd**.

C2 nodes

В процессе работы C2 использует промежуточное представление (SeaOfNodes IR):

- граф, совмещающий в себе

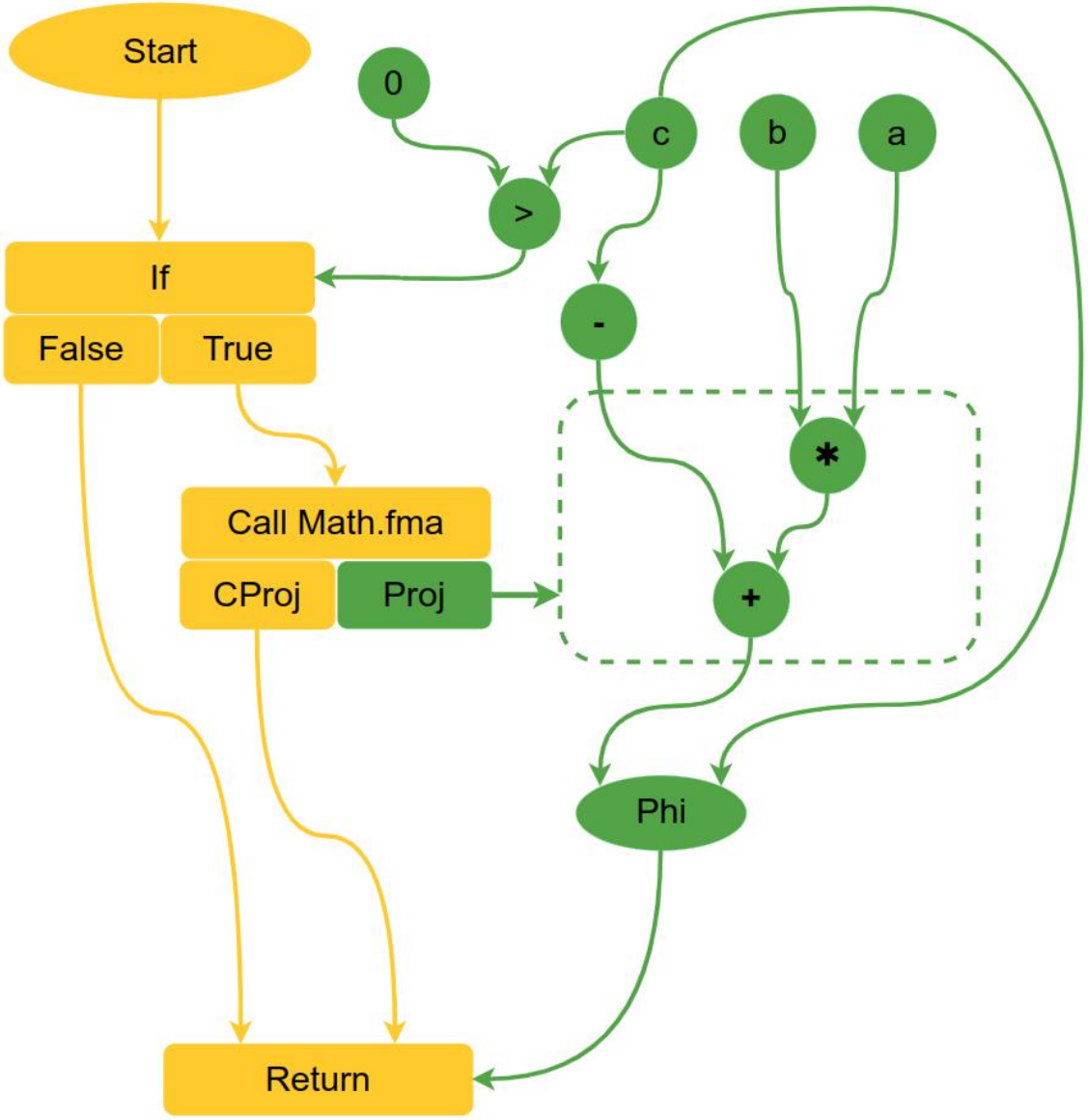
Control Flow и Data Flow.

- Data Flow имеет SSA форму.
- Sea Of Nodes. Cliff Click, Joker 2019*

Потрогаем “Море Нод”

*https://www.youtube.com/watch?v=9epgZ-e6DUU

Пример графа SeaOfNodes



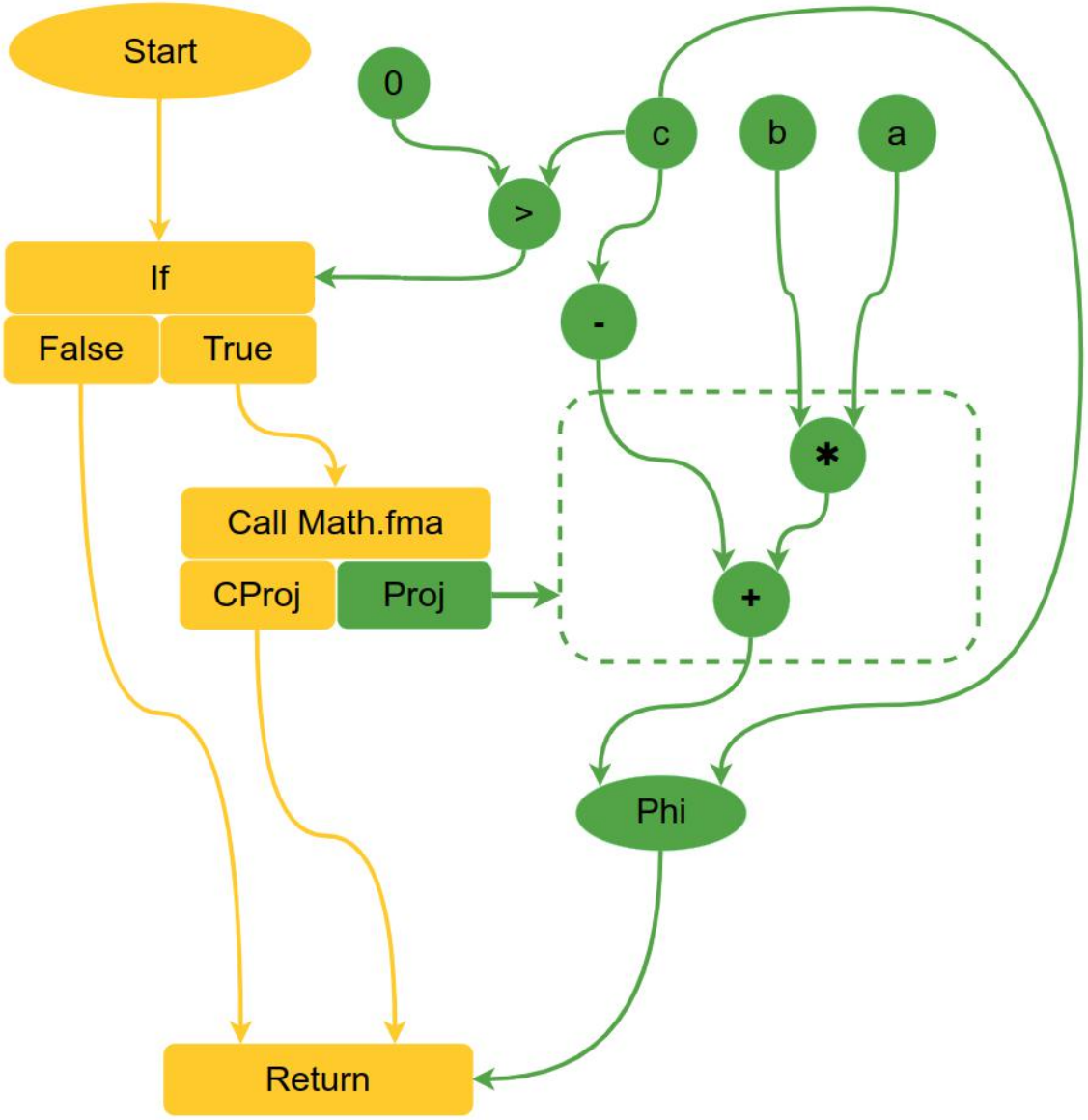
SeaOfNodes

Желтое - **Control** Flow Graph

+

Зеленое - **Data** Flow Graph

Пример графа SeaOfNodes




```
private static double
Dfma(double a, double b, double c){
    double res = c;
    if (c < 0) {
      res = Math.fma(a, b, -c);
    }
    return res;
}
```

```
Note:
res = Math.fma(a, b, -c);
// fused multiply-add
res = a * b + (-c);
```

IGV: реальный IR

```
private static double
Dfma(double a, double b, double c){
    return Math.fma(a, b, -c);
}
```

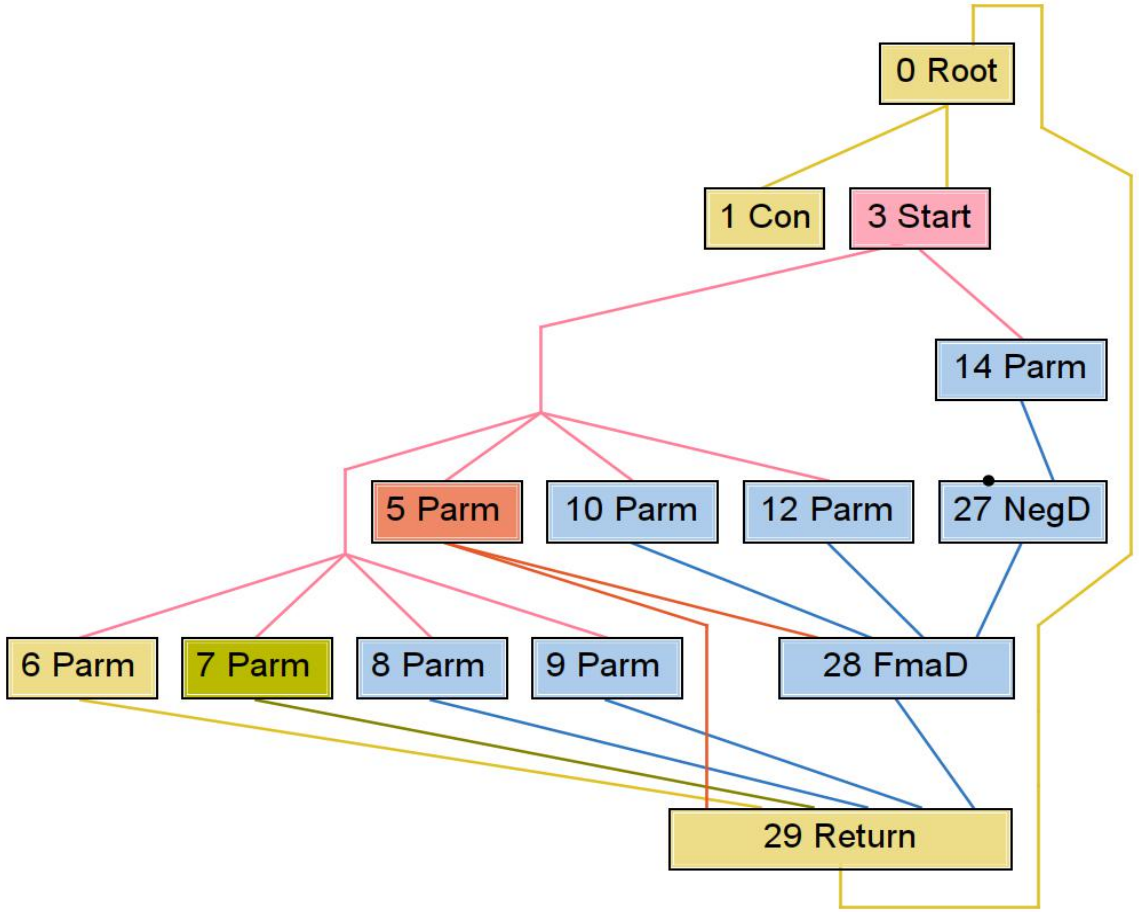
IGV: реальный IR

```
$ java -XX:PrintIdealGraphLevel=[1-5] -XX:PrintIdealGraphFile=filename.xml ...
```

```
private static double
Dfma(double a, double b, double c){
    return Math.fma(a, b, -c);
}
```

IGV: реальный IR

```
$ java -XX:PrintIdealGraphLevel=[1-5] -XX:PrintIdealGraphFile=filename.xml ...
```



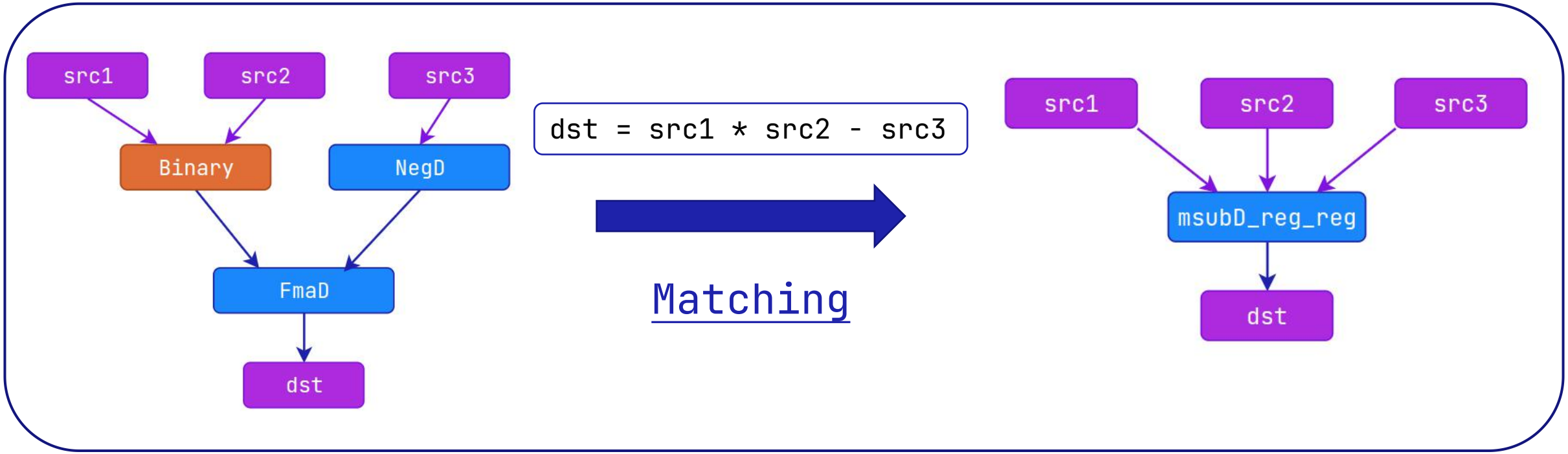

```
private static double
Dfma(double a, double b, double c){
    return Math.fma(a, b, -c);
}
```

Основные фазы:

- After parsing
- Before **matching**
- After **matching**
- Final code

C2 nodes - matching

- C2 в процессе работы отображает машинно-независимые ноды на платформенные ноды
- Правила отображения описываются на DSL(ADL)
- ADL описывает отображение N машинно-независимых нод в M платформенных

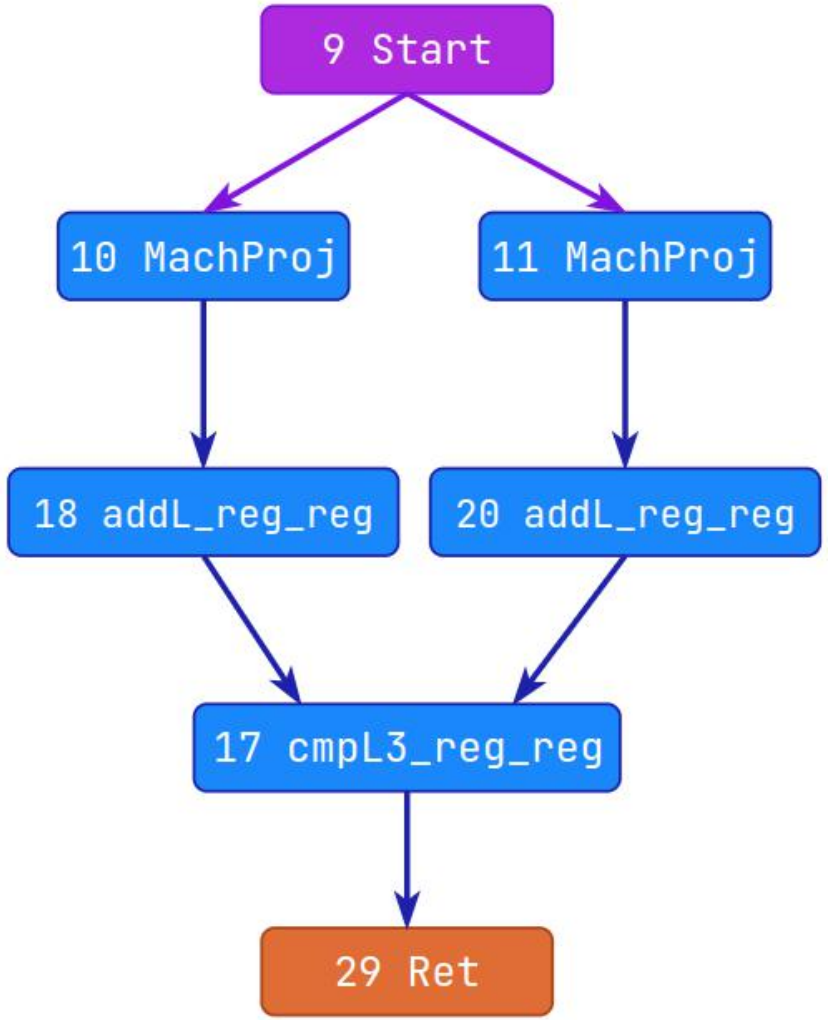


Compare unsigned без интринсика

```
@IntrinsicCandidate
public static int
compareUnsigned(long x, long y)
{

   return compare(x+MIN_VAL,y+MIN_VAL);

}
```

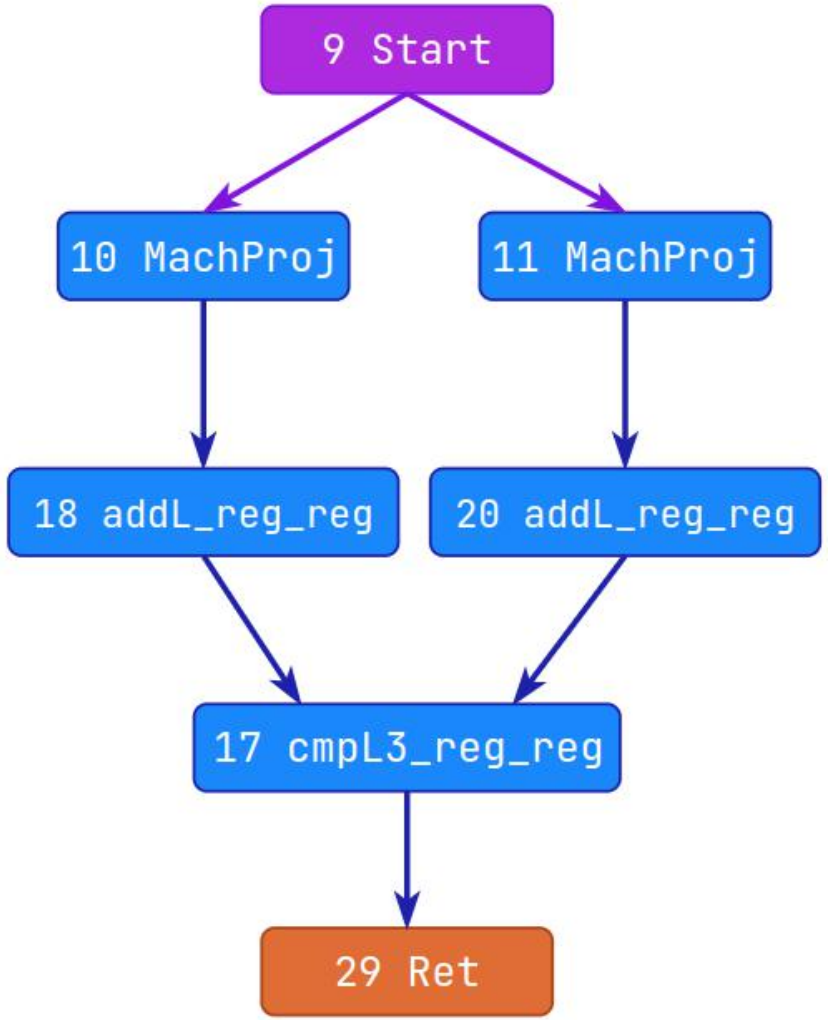


Compare unsigned без интринсика

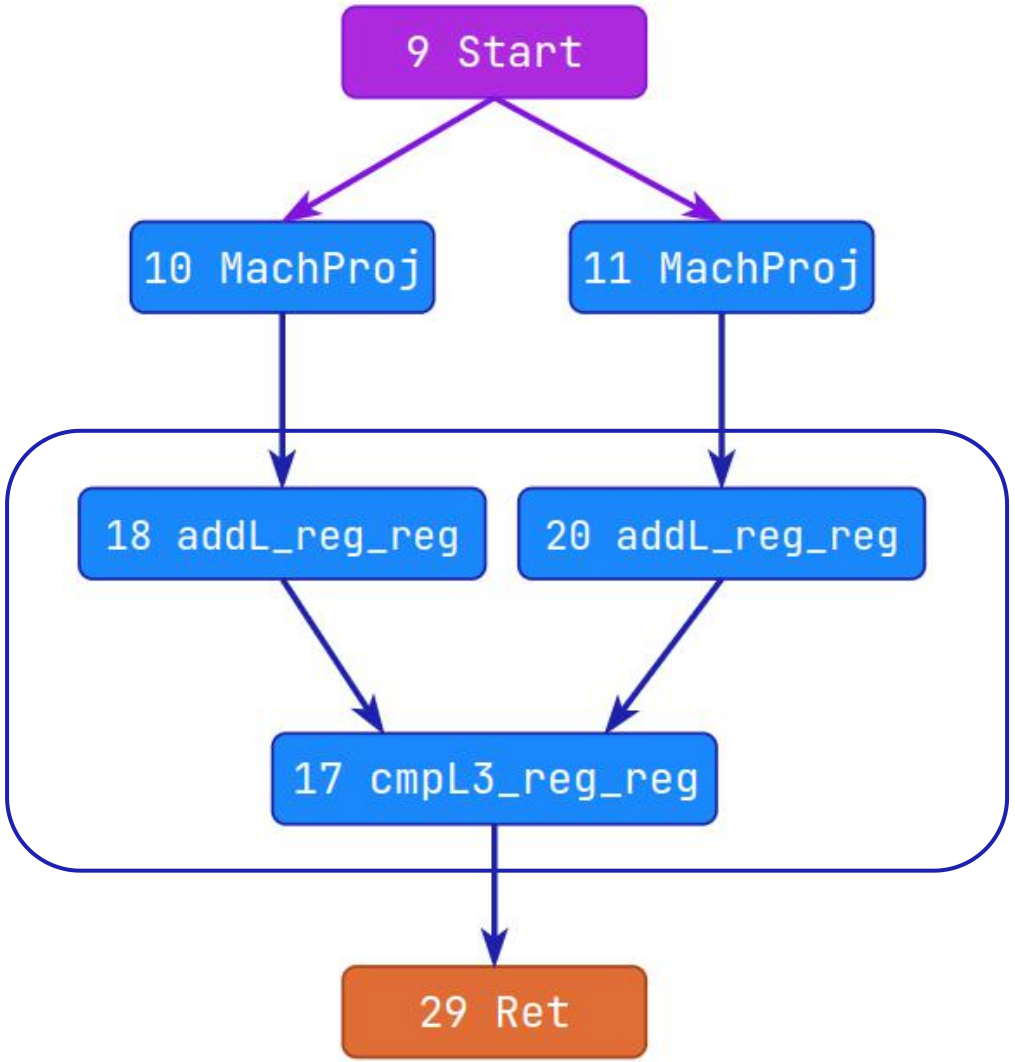
```
@IntrinsicCandidate
public static int
compareUnsigned(long x, long y)
{

   return compare(x+MIN_VAL,y+MIN_VAL);

}
```



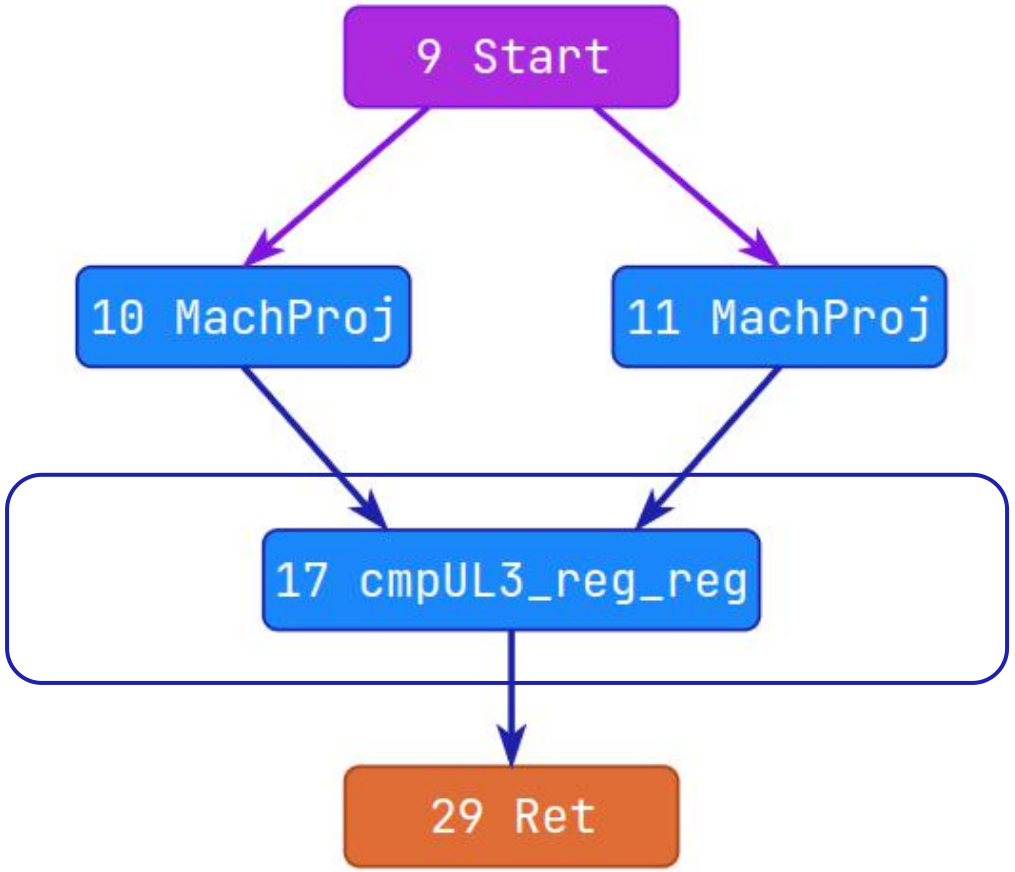
Compare unsigned без интринсика



Правило для подстановки:

```
match(Set dst (CmpUL3 op1 op2));
```

Compare unsigned с интринсиком



Правило для подстановки:

```
match(Set dst (CmpUL3 op1 op2));
```

C2 nodes: CmpUL3

```
instruct cmpUL3_reg_reg(iRegINoSp dst, iRegL op1, iRegL op2)
%{
  match(Set dst (CmpUL3 op1 op2));

  ins_cost(ALU_COST * 3 + BRANCH_COST);
  ins_encode %{
    __ cmp_ul2i(t0, as_Register($op1$$reg),
                    as_Register($op2$$reg));
    __ mv(as_Register($dst$$reg), t0);
  %}
%}
```

C2 nodes: CmpUL3

```
instruct cmpUL3_reg_reg(iRegINoSp dst, iRegL op1, iRegL op2)
%{
  match(Set dst (CmpUL3 op1 op2));

  ins_cost(ALU_COST * 3 + BRANCH_COST);
  ins_encode %{
    __ cmp_ul2i(t0, as_Register($op1$$reg),
                    as_Register($op2$$reg));
    __ mv(as_Register($dst$$reg), t0);
  %}
%}
```

C2 nodes: CmpUL3

```
instruct cmpUL3_reg_reg(iRegINoSp dst, iRegL op1, iRegL op2)
%{
  match(Set dst (CmpUL3 op1 op2));

  ins_cost(ALU_COST * 3 + BRANCH_COST);
  ins_encode %{
    __ cmp_ul2i(t0, as_Register($op1$$reg),
                    as_Register($op2$$reg));
    __ mv(as_Register($dst$$reg), t0);
  %}
%}
```

C2 nodes: CmpUL3

```
instruct cmpUL3_reg_reg(iRegINoSp dst, iRegL op1, iRegL op2)
%{
  match(Set dst (CmpUL3 op1 op2));

  ins_cost(ALU_COST * 3 + BRANCH_COST);
  ins_encode %{
    __ cmp_ul2i(t0, as_Register($op1$$reg),
                    as_Register($op2$$reg));
    __ mv(as_Register($dst$$reg), t0);
  %}
%}
```

Макроассемблер

```
instruct simple_node(iRegINoSp dst, iRegINoSp src1, iRegINoSp src2)
{
  ins_encode %{
__add(as_Register($dst$$reg),
      as_Register($src1$$reg),as_Register($src2$$reg));
  %}
}
```

```
Register dst  = x4;
Register src1 = x5;
Register src2 = x6;
```

```
add(dst, src1, src2);
        ↓
add(x4, x5, x6);
```

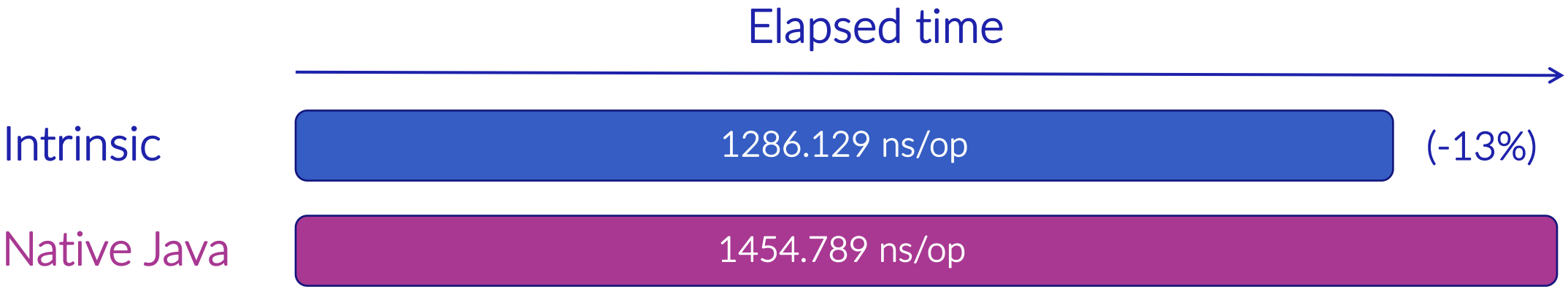
```
add(x4, x5, x6);
        ↓
0x00628233
```

Выбор регистров

Подстановки внутри макроассемблера

Генерация исполняемого фрагмента

Compare unsigned Benchmarks



С интринсиком (Intrinsic)

```
Benchmark                              Score      Error Units
Longs.compareUnsignedDirect         1286.129 ±    8.441 ns/op
```

Без интринсика (Native Java)

```
Benchmark                              Score      Error Units
Longs.compareUnsignedDirect         1454.789 ±  129.557 ns/op
```

Умножим два числа

- А как умножить два unsigned long, и получить 128-битное unsigned значение ?

Попробуем просто перемножить их:

```
static void printMultiply(long i, long j) {
    long ij = i * j;
    System.out.println
      ("Result of " + i + " * " + j + " = " + ij);
}
```

Упражнение

1. `printMultiply(10, 10);`

2. `printMultiply(-10, -10);`

```
long unsLong = 33 + Long.MIN_VALUE;
```

3. `printMultiply(2, unsLong);`

Упражнение: ответы

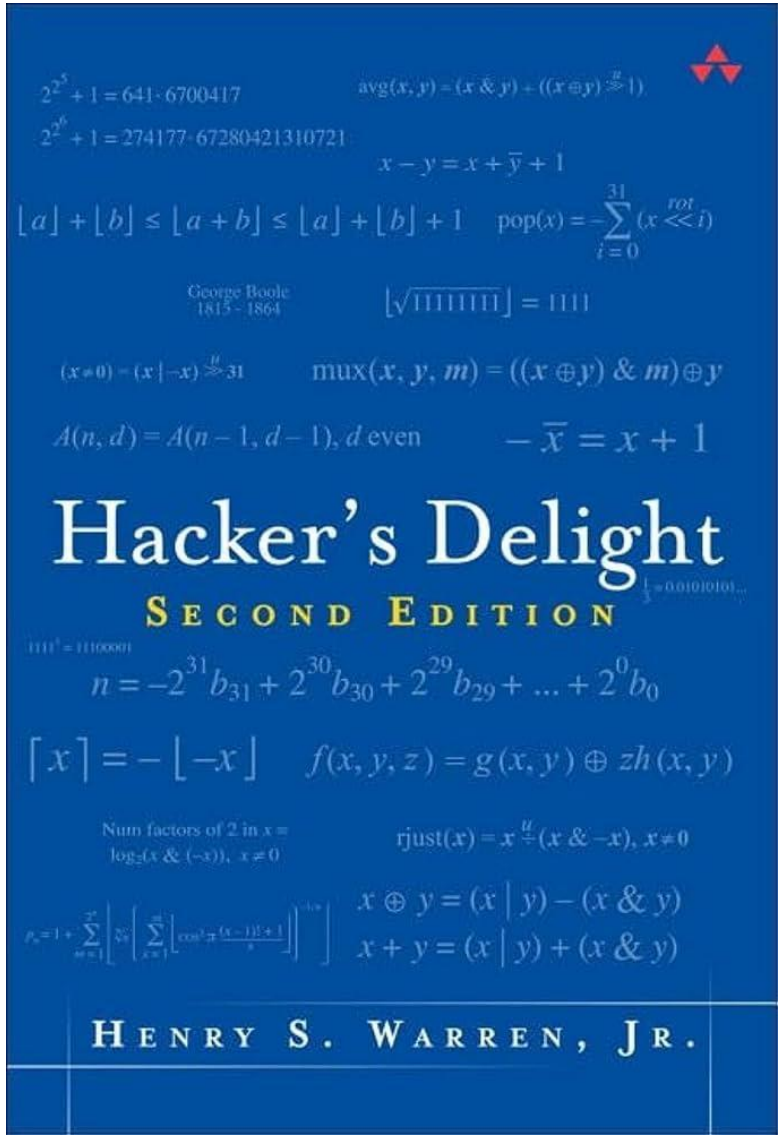
```
1. printMultiply(10, 10);
>>> Result of 10 * 10 = 100
2. printMultiply(-10, -10);
>>> Result of -10 * -10 = 100
long unsLong = 33 + Long.MIN_VALUE;
3. printMultiply(2, unsLong);
>>> Result of 2 * -9223372036854775775 = 66
```

Упражнение: причины

```
long unsLong = 33 + Long.MIN_VALUE;
3. printMultiply(2, unsLong);
>>> Result of 2 * -9223372036854775775 = 66
```

```
unsLong = 33 + 0x800...00L  == 0x8000000000000021;
// a*2 == a << 1
//0b10...100001 << 1 = 0b00...1000010
2 * unsLong = 0x0000000000000042; // = 66
>>> Result of 2 * unsLong = 66
```

Hacker`s Delight: окунемся в алгоритмы

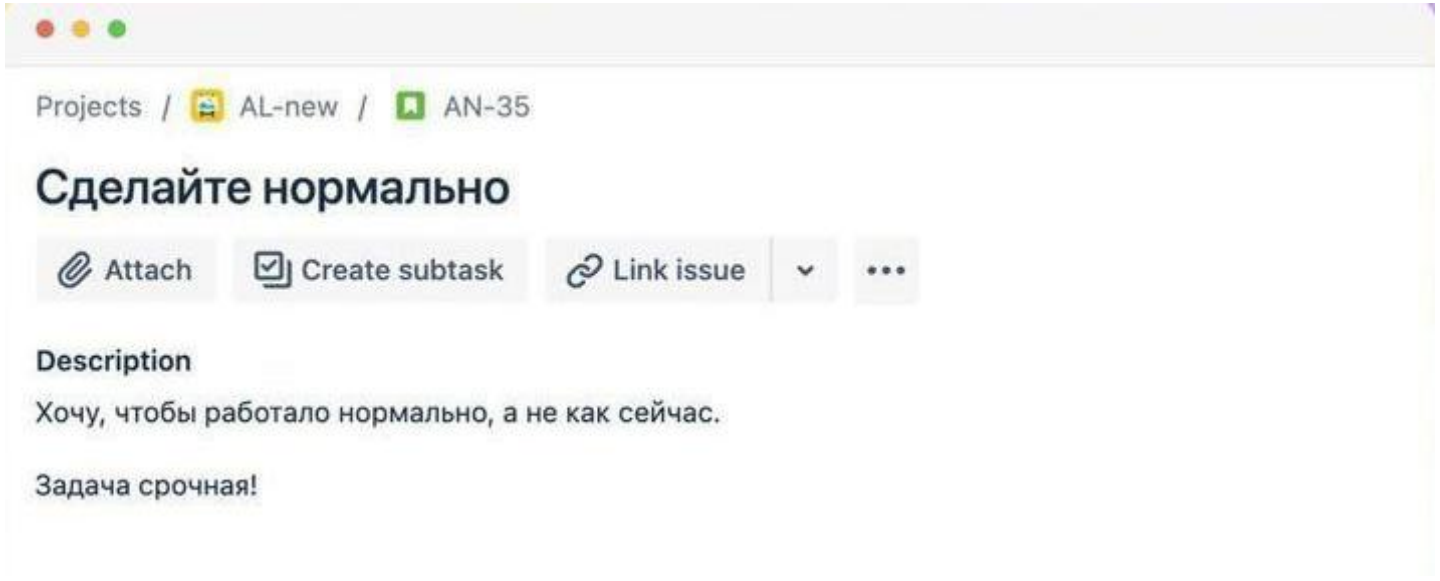


Немного о книге:

- Подсчет числа ведущих и замыкающих нулей
- Методы подсчета CRC
- Различные битовые манипуляции
- Низкоуровневые арифметические алгоритмы
- Вычисление квадратных и кубических корней

Wiki

UnsignedMultiplyHigh: сделайте нормально




```
@IntrinsicCandidate
static long unsignedMultiplyHigh(long x, long y);
@IntrinsicCandidate
static long multiplyHigh(long x, long y);
```

UnsignedMultiplyHigh: а как уже сделано?

```
@IntrinsicCandidate
static long unsignedMultiplyHigh(long x, long y) {
    long result = Math.multiplyHigh(x, y);
    // more operations under the result
    return result;
}
@IntrinsicCandidate
static long multiplyHigh(long x, long y){
    // Hacker's Delight (2nd ed.) (Addison Wesley, 2013), 173-174.
}
```

UnsignedMultiplyHigh: делаем куда ни шло

```
instruct mulHiL_rReg(iRegLNoSp dst, iRegL src1, iRegL src2)%{
  match(Set dst (MulHiL src1 src2));
  ins_encode %{
    __ mulh($dst$$reg, $src1$$reg,$src2$$reg);
  %}
%}
```

```
@IntrinsicCandidate
static long unsignedMultiplyHigh(long x, long y) {
    long result = _asm_mulh(x, y);
    // more operations under the result
    return result;
}
```

UnsignedMultiplyHigh: оценим результат

Как было
(Java методы)

```
and  t4,t3,a0
mul  t5,t5,t6
add  t2,t2,t4
srai t3,t3,0x20
srai t2,t2,0x20
add  t3,t3,t5
add  t2,t2,t3
srai t4,a4,0x3f
srai t3,a5,0x3f
and  t4,t4,a5
and  t3,t3,a4
add  t2,t2,t4
```

Куда ни шло
(+multiplyHigh)

```
mulh  t4,t2,t3
srai  t5,t2,0x3f
and   t5,t5,t3
srai  t3,t3,0x3f
and   t2,t3,t2
add   t4,t4,t5
add   t2,t2,t4
```

UnsignedMultiplyHigh: делаем нормально

```
instruct umulHiL_rReg(iRegLNoSp dst, iRegL src1, iRegL src2)%{
  match(Set dst (UMulHiL src1 src2));
  ins_encode %{
    __ mulhu($dst$$reg, $src1$$reg,$src2$$reg);
  %}
%}
```

```
@IntrinsicCandidate
static long unsignedMultiplyHigh(long x, long y) {
  return _asm_mulhu(x, y);
}
```

UnsignedMultiplyHigh: оценим результат

Как было
(Java методы)

```
and  t4,t3,a0
mul  t5,t5,t6
add  t2,t2,t4
srai t3,t3,0x20
srai t2,t2,0x20
add  t3,t3,t5
add  t2,t2,t3
srai t4,a4,0x3f
srai t3,a5,0x3f
and  t4,t4,a5
and  t3,t3,a4
add  t2,t2,t4
```

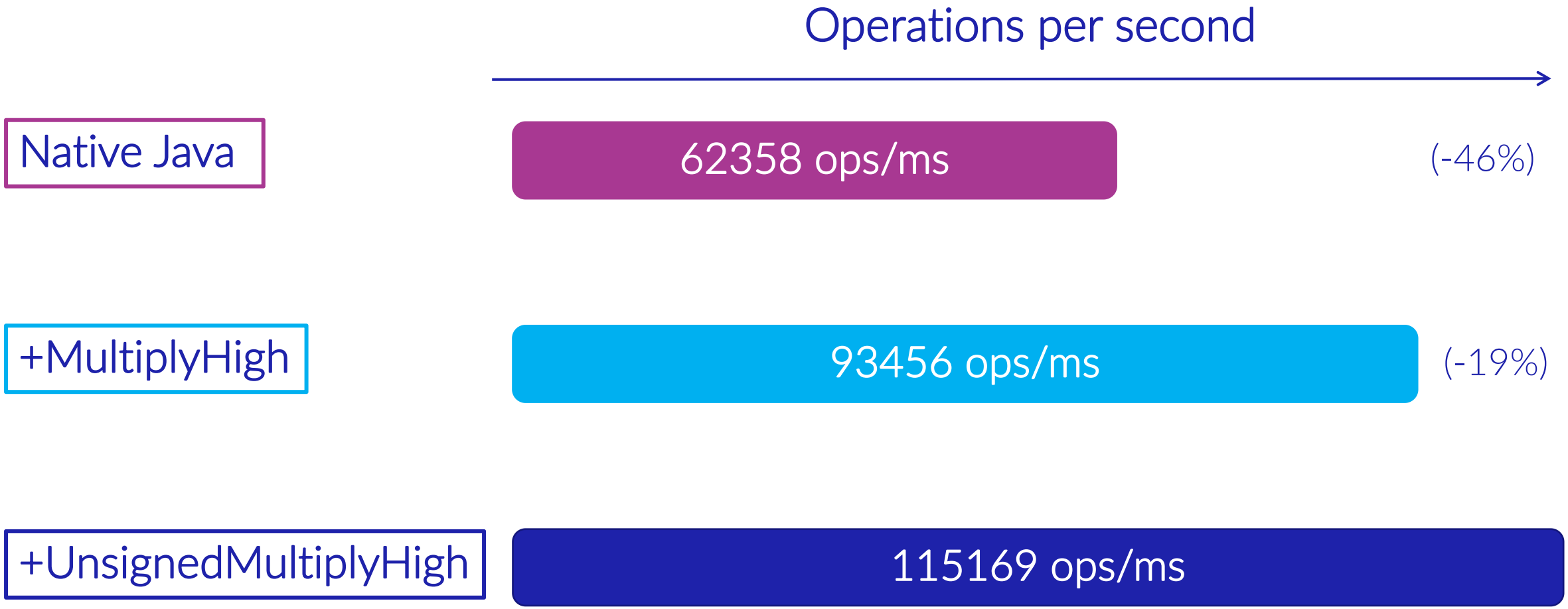
Куда ни шло
(+multiplyHigh)

```
mulh  t4,t2,t3
srai  t5,t2,0x3f
and   t5,t5,t3
srai  t3,t3,0x3f
and   t2,t3,t2
add   t4,t4,t5
add   t2,t2,t4
```

Сделали нормально
(+unsignedMultiplyHigh)

```
mulhu t2,t2,t3
```

UnsignedMultiplyHigh замеры производительности



C2 nodes итоги

Применяется, если для интринсика выполняются условия:

- Нужен только в C2, не нужен в C1
- Относительно небольшой размер машинного кода (код интринсика вполне может заинлайниться)
- Относительно небольшое количество используемых регистров

Последние два условия иногда могут нарушаться

Еще в C2 есть трюки

Shortcuts в коде C2 компилятора:

```
@IntrinsicCandidate
public static double pow(double a, double b);
```

Если:

b - const double

```
b == 2.0d
```

или

```
b == 0.5d
sqrt supported
```

вид `pow(a, b)`:

```
return a * a;
```

```
return asm_sqrt(a);
 (RISC-V: fsqrt.d)
```

C2 трюки: реализация

```
Node* exp = round_double_node(argument(2));
const TypeD* d =
    _gvn.type(exp)->isa_double_constant();
if (d != nullptr) { // b - const double?
if (d->getd() == 2.0) {
    ...
} else if (d->getd() == 0.5 &&
Matcher::match_rule_supported(Op_SqrtD)) {
    ...
```

Library интринсики

- Их можно поделить на несколько типов:
  - IR/C2 nodes (remainderUnsigned_i/l, divideUnsigned_i/l, compareUnsigned_i/l, Math.ceil/floor/rint, signumF/signumD, unsignedMultiplyHigh)
  - Сгенерированные функции - RuntimeStubs (poly1305.processBlocks, updateBytesCRC32)
  - Вызовы в нативный (сгенерированный gcc/clang) код (sin,cos,dpow)

Сгенерированные стабы

- Могут использоваться и в C1 и в C2
- Обычно это много машинного кода, который использует немалое количество регистров и может обращаться к памяти JVM (например таблицы crc32)
- В машинном коде выглядит как обычный вызов функции, не JNI: т.е. достаточно быстро, но медленнее, чем заинлайненная нода C2
- Машинный код генерируется во время старта JVM, до запуска приложения и может быть оптимизирован под ваш CPU

Например шифрование

- Poly1305 алгоритм хеширования.
- Часто используется в связке с ChaCha
- ChaCha20-Poly1305 authenticated cipher deployed in TLS on the internet
- TL;DR: хорошо, когда эта связка быстрая, если вы делаете TLS.

Poly1305

- При старте JVM генерируем функции (пишем ее на макроассемблере), которые делают ChaCha20Cipher.implChaCha20Block и poly1305.processBlocks и умеют это делать быстрее, чем pure java code
- Учим C1 и C2 вызывать сгенерированные функции, вместо java кода

Сгенерированные стабы, пример

```
address generate_poly1305_processBlocks() {
 __ align(CodeEntryAlignment);
 address start = __ pc();
 __ enter();
```

Сгенерированные стабы, пример

```
address generate_poly1305_processBlocks() {
  __ align(CodeEntryAlignment);
  address start = __ pc();
  __ enter();
  RegSetIterator<Register> regs = (RegSet::range(x14,
x31) - RegSet::range(x22, x27)).begin();
```

Сгенерированные стабы, пример

```
  address generate_poly1305_processBlocks() {
    __ align(CodeEntryAlignment);
    address start = __ pc();
    __ enter();
    RegSetIterator<Register> regs = (RegSet::range(x14,
x31) - RegSet::range(x22, x27)).begin();
    RegSet saved_regs = RegSet::range(x18, x21);
    __ push_reg(saved_regs, sp);
//сохраним на стек Save On Entry (SOE) регистры
//по контракту (ABI) их нельзя изменять
```

Сгенерированные стабы, пример

```
  address generate_poly1305_processBlocks() {
    __ align(CodeEntryAlignment);
    address start = __ pc();
    __ enter();
    RegSetIterator<Register> regs = (RegSet::range(x14,
x31) - RegSet::range(x22, x27)).begin();
   RegSet saved_regs = RegSet::range(x18, x21);
    __ push_reg(saved_regs, sp); //сохраним на стек
    const Register input_start = c_rarg0, length =
c_rarg1, acc_start = c_rarg2, r_start = c_rarg3;
// Arguments
```

Poly1305 %improvements over pure java

Benchmark	dataSize	Improvement, %
digestBuffer	64	9.7
	256	32.3
	1024	101.1
	16384	250.4
digestBytes	64	13.2
	256	54.9
	1024	134.6
	16384	252.1
updateBytes	64	145.6
	256	255.1
	1024	320.0
	16384	277.0

Вызовы нативного кода aka natives

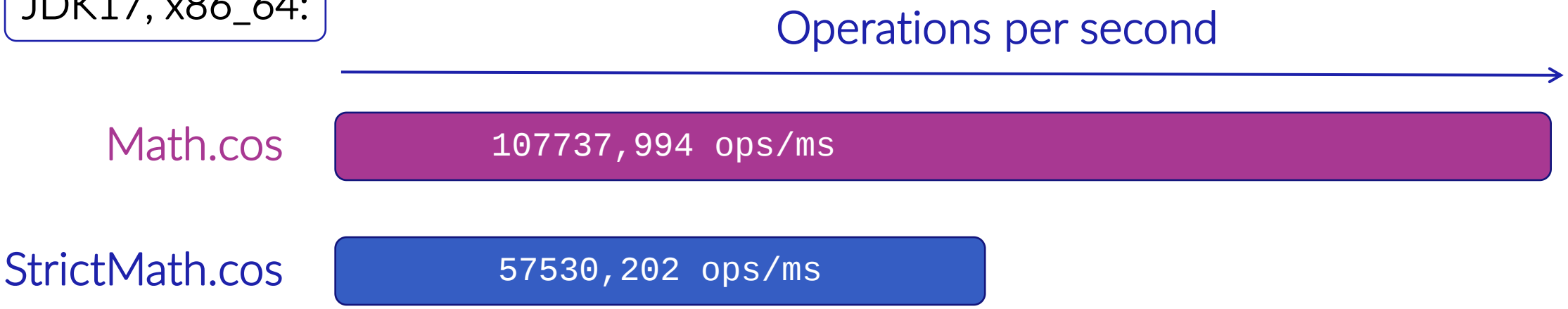
- Спорно - интринсик ли это
- Как ни странно, это тоже НЕ JNI вызовы (leaf call)
- Зачастую являются дефолтной реализацией интринсика, общей для всех платформ, при отсутствии более оптимизированных версий
- Удобно для новых платформ
- Производительность может зависеть от типа/версии компилятора, которым была собрана openJDK

Разница RuntimeStub vs. Native

RuntimeStub	Native (SharedRuntime)
Генерируется **при старте** JVM	Гененируется **при компиляции** OJDK
Пишется на **макроассемблере**	Написан на **C++**
Учитывает **набор расширений**	**Легко заменить** на оптимизированный вариант

К чему это приводит?

JDK17, x86_64:




```
Benchmark                          Score        Error   Units
MathBench.cosDouble          107737,994 ± 2414,245   ops/ms
StrictMathBench.cosDouble  57530,202 ± 1250,267   ops/ms
```

На JDK17 Math.cos - runtimeStub
StrictMath.cos - native

Легкая оптимизация

macOS/clang15/i9-9880H

опции по умолчанию

```
StrictMathBench.cosDouble  57530,202 ± 1250,267  ops/ms
```

--with-extra-cflags="-march=native" + 5%

```
StrictMathBench.cosDouble  60488,424 ± 1255,490  ops/ms
```

Легкая оптимизация

macOS/clang15/i9-9880H

--with-extra-cflags="-march=native" + 5%

```
StrictMathBench.cosDouble  60488,424 ± 1255,490  ops/ms
```

linux/gcc11/i9-9880H

опции по умолчанию, еще + 8%

```
StrictMathBench.cosDouble  65255.648 ± 454.490   ops/ms
```

А на java21?

JDK21, x86_64:

```
Benchmark                          Score          Error   Units
MathBench.cosDouble           100076,171 ± 3862,189   ops/ms
StrictMathBench.cosDouble     103812,277 ± 4591,225   ops/ms
```

На JDK21

`Math.cos` - runtimeStub

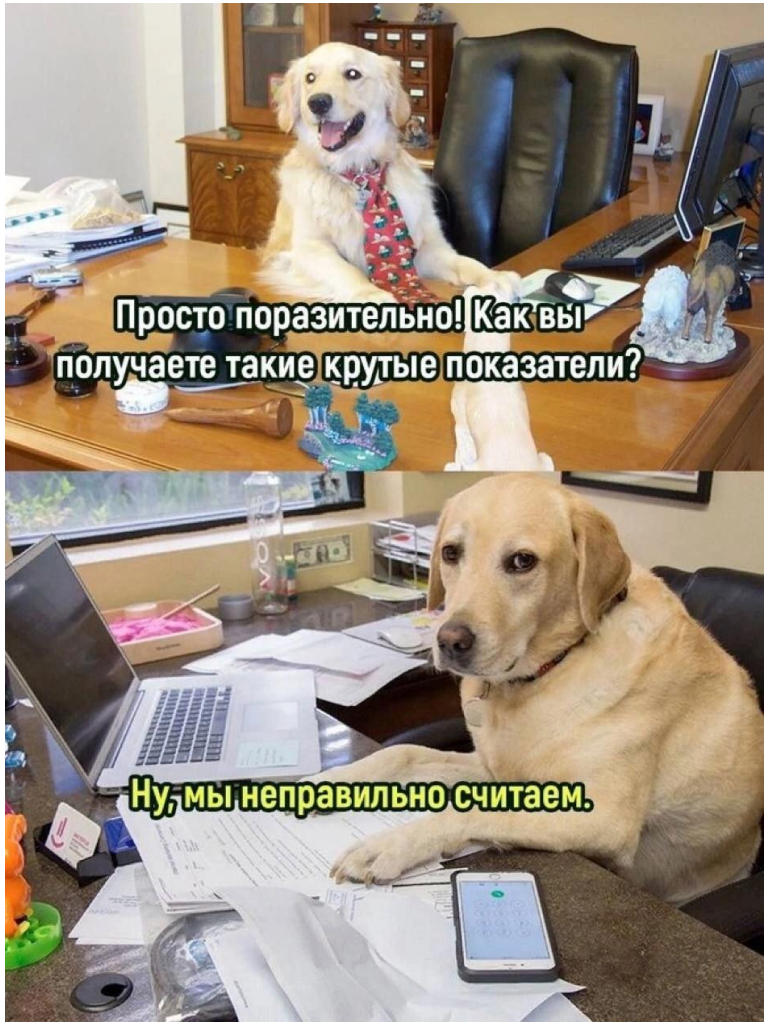
`StrictMath.cos` - pure java (порт fdlibm на java)

StrictMathBench

```
public double double1 = 1.0d;
@Benchmark
public double  cosDouble() {
  return StrictMath.cos(double1);
}
```

Особенности StrictMathBench (JMH):

- Подставляется постоянное значение
- Хорошо подходит для проверки RuntimeStubs или Natives
- Компилятор агрессивно оптимизирует код
- Результат может быть обманчив



StrictMathBench: добавим случайности

```
public double double1 = 1.0d;
@Benchmark
public double  cosDouble() {
  return StrictMath.cos(double1);
}
```

Рандомизируем входные данные:

```
public double[] DA;
@Benchmark
public void trigcos(Blackhole bh) {
  for (int i = 0; i < TESTSIZE; i++)
    Res[i] = StrictMath.cos(DA[i]);
}
```

На рандомизированном наборе

JDK21

```
Benchmark.mathCos          97,610 ± 1,893  ops/ms
Benchmark.strictMathCos    57,204 ± 1,948  ops/ms
```

JDK17

```
Benchmark.mathCos          96,803 ± 2,020  ops/ms
Benchmark.strictMathCos    39,588 ± 1,102  ops/ms
```

Предыдущий тест всегда делал Math.cos(1.0d), текущий работает на рандомизированном наборе данных

FDLIBM на java

Две больших фазы:

JDK9: JDK-8134780: Port fdlibm to Java, part 1

```
cbrt, exp, pow, hypot
```

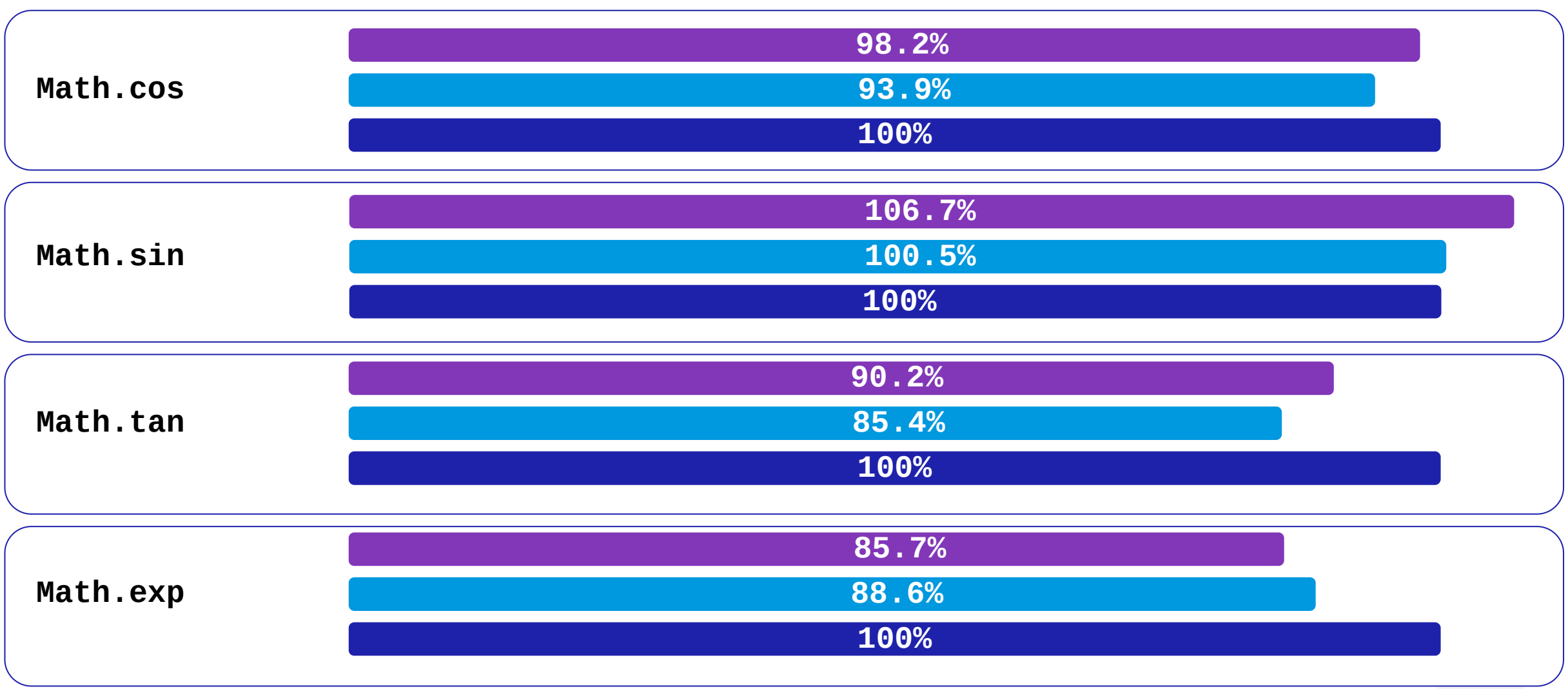
JDK21: JDK-8171407: Port fdlibm to Java, part 2

```
sin,   cos,  tan,  asin, acos, atan,
sinh, cosh, tanh, atan2
log, log10, expm1, log1p
```

А на RISC-V?

- RuntimeStubs для части математики отсуствуют
- Вместо них используются natives
- Они всё ещё способны выиграть по перформансу у pure java

Зависимость от компилятора (op/s)



Как узнать, что используется

https://chriswhocodes.com/hotspot_intrinsics_openjdk17.html

- Настроить фильтр: показать только Library
- Флаг N - native, последний из описанных типов.
- Без флага - один из первых двух

Вернемся к RISC-V

- За год было реализовано множество интринсиков, в том числе векторные версии

- Перформанс векторных версий пока не тестировался (за неимением аппаратуры), но эта ситуация начинает меняться

Краткий обзор по прогрессу RISC-V

Реализованные за год Library интринсики:

- ExpandBits, CompressBits, MD5.implCompress, remainderUnsigned_i/l, divideUnsigned_i/l, compareUnsigned_i/l, ChaCha20, roundD/roundF, VectorizedHashCode, ensureMaterializedForStackWalk, poly1305.processBlocks, Math.ceil/floor/rint, copySignF/copySignD, signumF/signumD, unsignedMultiplyHigh
- Улучшены String.Compare (до 23x в некоторых случаях), String.indexOf (до 11x в некоторых случаях)

Краткий обзор по прогрессу RISC-V

- Появился аналог cpuid
(системный вызов hwprobe, linux 6.5+)
- OpenJDK17 поддерживает RISC-V

Текущие задачи RISC-V

Список интринсиков: https://bugs.openjdk.org/browse/JDK-8318216

Зеленое - сделано, синее - в процессе, серое - еще не начато

1.	RISC-V: C2 VectorizedHashCode	RESOLVED	Vladimir Kempik
2.	RISC-V: C2 CompressBits	RESOLVED	Hamlin Li
3.	RISC-V: C2 ExpandBits	RESOLVED	Hamlin Li
4.	RISC-V: C2 ReverseI	IN PROGRESS	Hamlin Li
5.	RISC-V: C2 ReverseL	IN PROGRESS	Hamlin Li
6.	RISC-V: C2 CmpU3	RESOLVED	Hamlin Li
7.	RISC-V: C2 CmpUL3	RESOLVED	Hamlin Li
8.	RISC-V: C2 UDivI	RESOLVED	Hamlin Li
9.	RISC-V: C2 UModI	RESOLVED	Hamlin Li
10.	RISC-V: C2 UModL	RESOLVED	Hamlin Li
11.	RISC-V: C2 ConvHF2F	RESOLVED	Hamlin Li
12.	RISC-V: C2 ConvF2HF	RESOLVED	Hamlin Li
13.	RISC-V: C2 GetAndAddB	OPEN	Hamlin Li
14.	RISC-V: C2 GetAndAddS	OPEN	Hamlin Li
15.	RISC-V: C2 GetAndSetB	OPEN	Hamlin Li
16.	RISC-V: C2 GetAndSetS	OPEN	Hamlin Li
17.	RISC-V: C2 OverflowAdd/Sub/Mul I/L	OPEN	Unassigned
18.	RISC-V: C2 Digit	OPEN	Unassigned
19.	RISC-V: C2 LowerCase	OPEN	Unassigned
20.	RISC-V: C2 UpperCase	OPEN	Unassigned
21.	RISC-V: C2 Whitespace	OPEN	Unassigned
22.	RISC-V: C2 CopySignD	CLOSED	Ilya Gavrilin
23.	RISC-V: C2 CopySignF	CLOSED	Ilya Gavrilin
24.	RISC-V: C2 UDivL	RESOLVED	Hamlin Li
25.	RISC-V: C2 VectorCastHF2F	RESOLVED	Hamlin Li
26.	RISC-V: C2 VectorCastF2HF	RESOLVED	Hamlin Li
27.	RISC-V: C2 DivModI	OPEN	Unassigned
28.	RISC-V: C2 DivModL	OPEN	Unassigned
29.	RISC-V: C2 UDivModI	OPEN	Unassigned
30.	RISC-V: C2 UDivModL	OPEN	Unassigned
31.	RISC-V: C2 LoadD_unaligned	OPEN	Unassigned
32.	RISC-V: C2 LoadL_unaligned	OPEN	Unassigned
34.	RISC-V: C2 NegI	OPEN	Unassigned
35.	RISC-V: C2 NegL	OPEN	Unassigned
36.	RISC-V: C2 OverflowAddI	OPEN	Unassigned
37.	RISC-V: C2 OverflowSubI	OPEN	Unassigned
38.	RISC-V: C2 OverflowMulI	OPEN	Unassigned
39.	RISC-V: C2 OverflowAddL	OPEN	Unassigned
40.	RISC-V: C2 OverflowSubL	OPEN	Unassigned
41.	RISC-V: C2 OverflowMulL	OPEN	Unassigned
42.	RISC-V: C2 PopCountVI	IN PROGRESS	Hamlin Li
43.	RISC-V: C2 PopCountVL	IN PROGRESS	Hamlin Li
44.	RISC-V: C2 ReverseV	IN PROGRESS	Hamlin Li
45.	RISC-V: C2 ReverseBytesV	OPEN	Hamlin Li
46.	RISC-V: C2 RoundDoubleModeV	OPEN	Unassigned
47.	RISC-V: C2 RotateLeftV	IN PROGRESS	Hamlin Li
48.	RISC-V: C2 RotateRightV	IN PROGRESS	Hamlin Li
49.	RISC-V: C2 SignumVF	RESOLVED	Hamlin Li
50.	RISC-V: C2 SignumVD	RESOLVED	Hamlin Li
51.	RISC-V: C2 MulReductionVI	OPEN	Unassigned
52.	RISC-V: C2 MulReductionVL	OPEN	Unassigned
53.	RISC-V: C2 MulReductionVF	OPEN	Unassigned
54.	RISC-V: C2 MulReductionVD	OPEN	Unassigned
55.	RISC-V: C2 MulAddVS2VI	OPEN	Unassigned
56.	RISC-V: C2 VectorCmpMasked	OPEN	Unassigned
57.	RISC-V: C2 RoundVF	IN PROGRESS	Hamlin Li
58.	RISC-V: C2 RoundVD	IN PROGRESS	Hamlin Li
59.	RISC-V: C2 ExtractUB	OPEN	Hamlin Li
60.	RISC-V: C2 VectorLoadShuffle	IN PROGRESS	Hamlin Li
61.	RISC-V: C2 VectorCastF2HF	CLOSED	Hamlin Li
62.	RISC-V: C2 VectorCastHF2F	CLOSED	Hamlin Li
63.	RISC-V: C2 VectorUCastB2X	RESOLVED	Hamlin Li
64.	RISC-V: C2 VectorUCastS2X	RESOLVED	Hamlin Li
65.	RISC-V: C2 VectorUCastI2X	RESOLVED	Hamlin Li

Наш вклад

roundD/roundF

VectorizedHashCode

ensureMaterializedForStackWalk

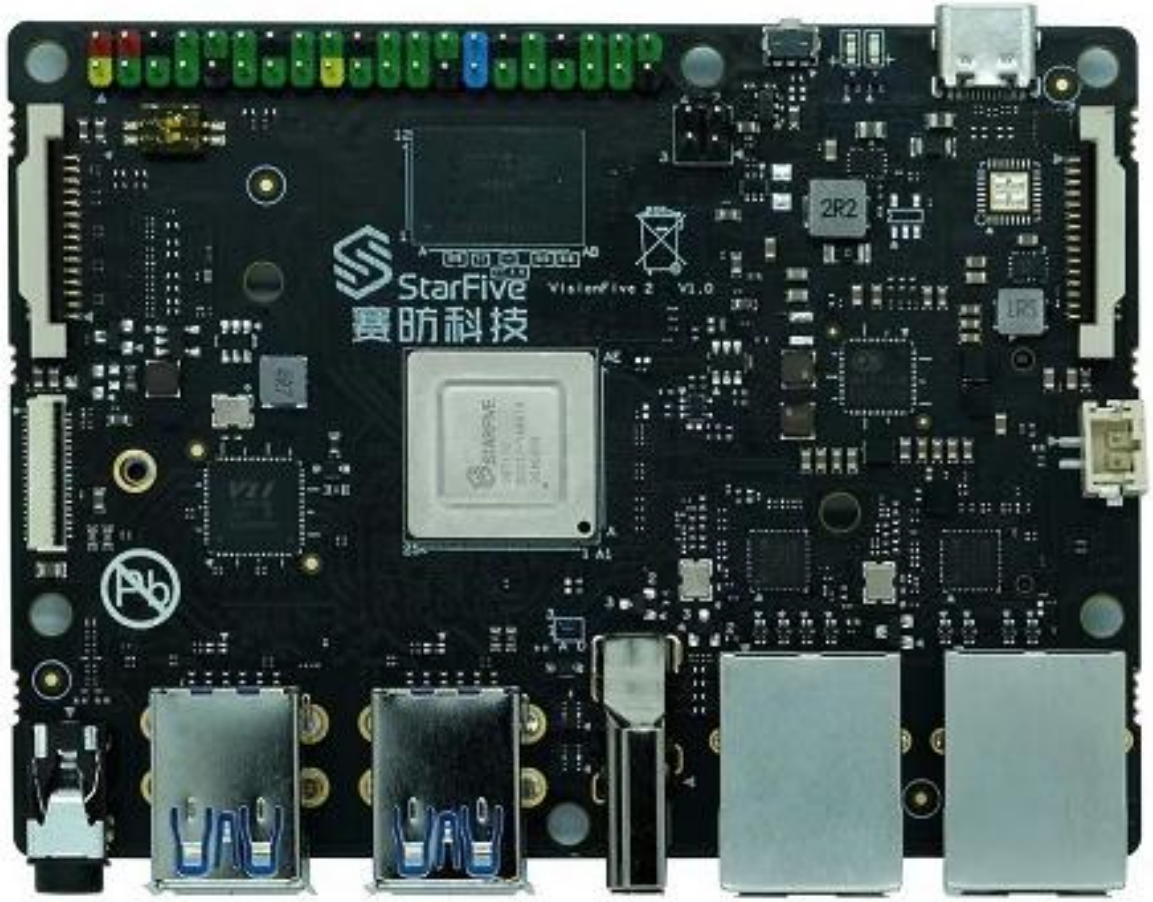
unsignedMultiplyHigh

poly1305.processBlocks

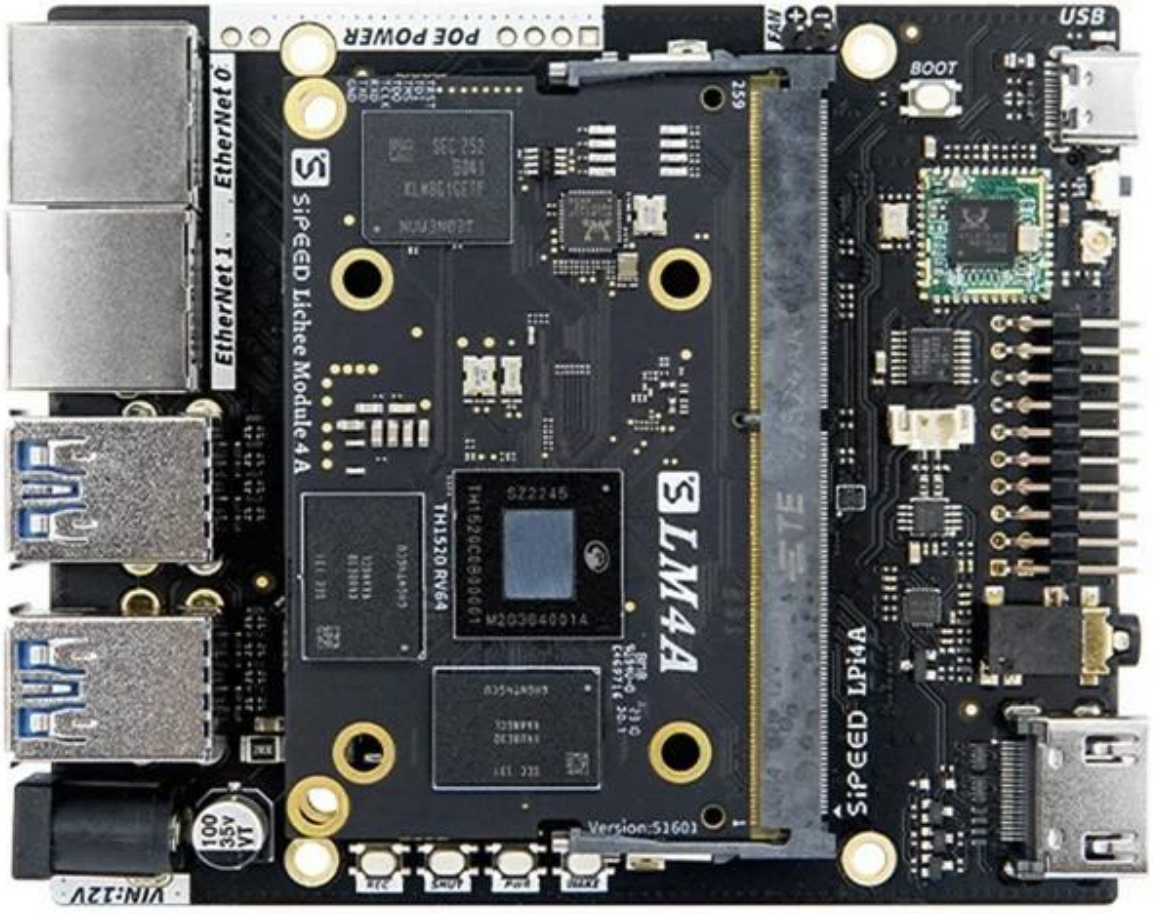
Math.ceil/floor/rint, copySignF/copySignD, signumF/signumD

RISC-V общедоступные платы RV64GC

VisionFive 2



Licheepi 4a



Выводы

- OpenJDK может по-разному реализовывать то, что мы считаем интринсиками

Выводы

- OpenJDK может по разному реализовывать то, что мы считаем интринсиками

- Производительность интринсика может зависеть от типа и версии компилятора, которым собрал openJDK ваш вендор

Выводы

- OpenJDK может по разному реализовывать то, что мы считаем интринсиками
- Производительность интринсика может зависеть от типа и версии компилятора, которым собрал openJDK ваш вендор
- У RISC-V большой прогресс за год по количеству поддерживаемых интринсиков, low hanging fruits уже не осталось

Полезные ссылки

- Sea Of Nodes. Cliff Click, Joker 2019, https://www.youtube.com/watch?v=9epgZ-e6DUU
- Volker Simonis — HotSpot Intrinsics, https://youtu.be/76gyiCteq-I
- JDK Intrinsics list - https://chriswhocodes.com/hotspot_intrinsics_openjdk17.html
- JDK21 + вектора 0.7.1, работает на licheepi4A: https://github.com/syntacore/syntaj21/tree/rvv0.7.1

Контакты

Владимир Кемпик: vladimir.kempik@syntacore.com
Илья Гаврилин: ilya.gavrilin@syntacore.com