**FP16 Support (lworld+fp16) :-**

**Implementation details: -**

1/ Newly defined Float16 class encapsulate a short value holding IEEE 754 binary16 encoded value.

2/ Float16 is a primitive class which in future will be aligned with other enhanced primitive wrapper classes proposed by [JEP-402.](https://openjdk.org/jeps/402)

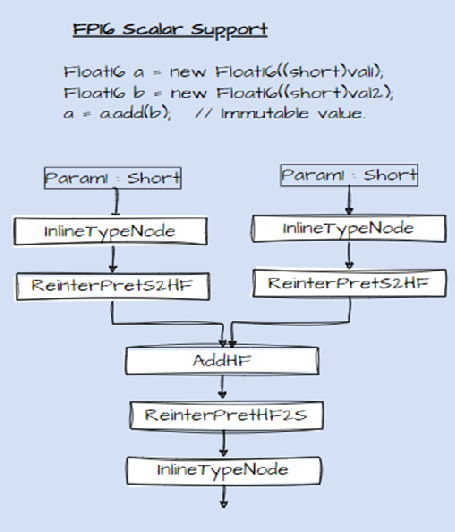
3/ Float16 to support all the operations supported by corresponding Float class.

4/ Java implementation of each API will internally perform floating point operation at FP32 granularity.

5/ API which can be directly mapped to an Intel AVX512FP16 instruction will be a candidate for intensification by C2 compiler.

6/ With Valhalla, C2 compiler always creates an InlineType IR node for a value class instance.  
Total number of inputs of an InlineType node match the number of non-static fields. In this case node will have one input of short type TypeInt::SHORT.

7/ Since all the scalar AVX512FP16 instructions operate on floating point registers and Float16 backing storage is held in a general-purpose register hence we need to introduce appropriate conversion IR which moves a 16-bit value from GPR to a XMM register and vice versa.

[](https://private-user-images.githubusercontent.com/59989778/261691956-192fca7e-6b7e-4e62-9b09-677e33eca48d.png?jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..SSFTYLyWKrJUDMy6wxtXdKvRxjkRn4uwt4n1tfkN43o)

8/ Current plan is to introduce a new IR node for each operation which is a subclass of its corresponding single precision IR node. This will allow leveraging idealization routines (Ideal/Identity/Value) of its parent operation.

9/ All the single/double precision IR nodes carry a Type::FLOAT/DOUBLE ideal type. This represents entire FP32/64 value range and is different from integral types which explicitly record lower and upper bounds of value ranges. Value resolution routines operating over integral types calibrates the lattice associated with IR node based on flow function depicting the semantics of the operation.

10/ There are two separate ideal type for floating point constants TypeF and TypeD, these are singleton types and represents a unique single/double precision floating point value

11/ Ideal types are linked to specific register classes and determine the allocation set for register allocator.

12/ Since FP16 value range is a proper subset of FP32 value range hence all the newly created IR nodes will still carry Type::FLOAT type.

13/ To handle constant folding scenarios value routines associated with newly introduced conversion IR nodes convert an FP16 constant to a FP32 constant (TypeF) value using existing runtime helpers. Constant folding is performed by existing idealization routines of parent IR nodes.

14/ Auto-vectorizer scans through packs of scalar IR to create a vector IR.

15/ Functional and performance test for creation for each newly supported operation.

Above design is generic and should work for any reduced precision types like FP8 values.

Please let me know your views on this.

Best Regards,  
Jatin

PS: Test instantiating Float16 class must be compiled with -XDenablePrimitiveClasses currently, we need to relax this constraint for known primitive classes in language compiler, JEP-402 talks about converting existing primitive box classes to value classes should address this limitation.

Clang Test :-

\_Float16 add\_constantfp16 () {

\_Float16 temp1 = 15360;

\_Float16 temp2 = 15360;

return temp1 + temp2;

}

LLVM IR for AVX512FP16 targets



LLVM IR on non-AVX512F targets

CPROMPT>clang -mavx512f fp16add.c -emit-llvm -Xclang -disable-O0-optnone -S

CPROMPT>opt -passes=mem2reg fp16add.ll -o fp16add\_opt.ll -S



InstCombiner Pass folds the values of APFloat operands as per IEEE floating point semantics for half, single or double precision constants.

MOVD MS switch over penalty: -

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

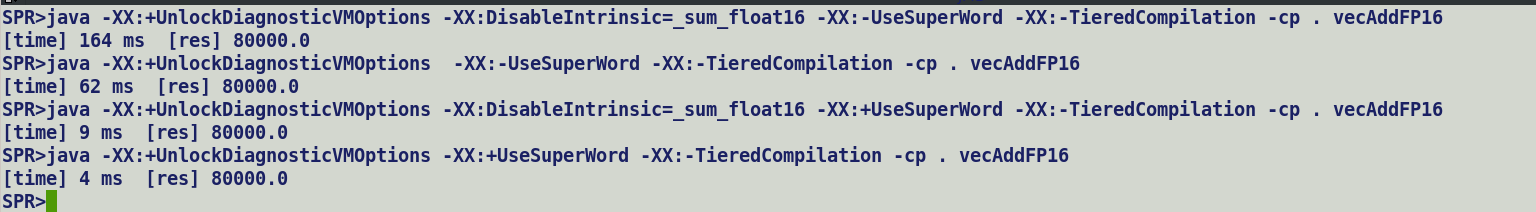
Float16 is a primitive class wrapping a short field holding binary16 encoded value.

Thus, a short field is used meagrely as a backing storage and raw value needs to be converted into required destination type.

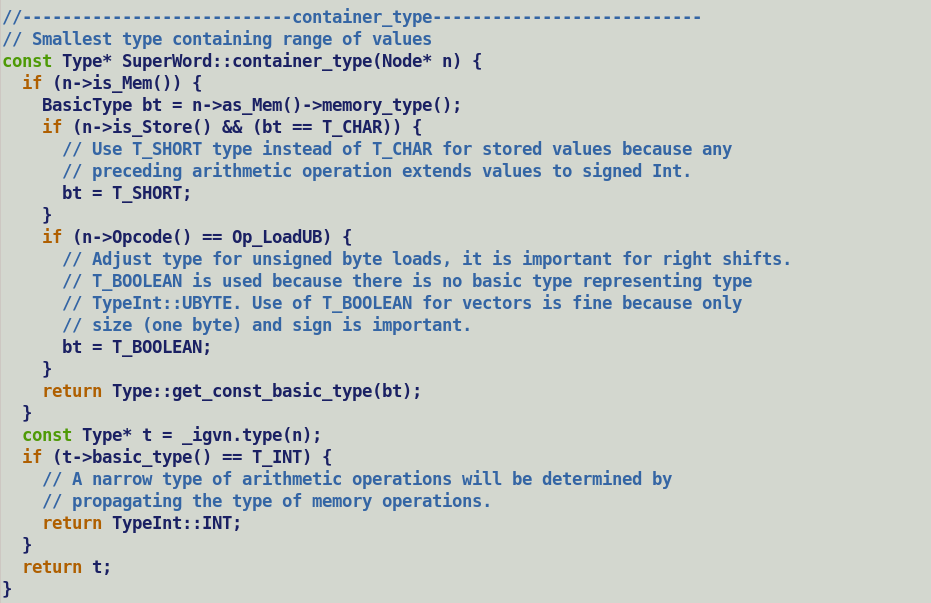
However, given that JVM local variable is a 32-bit value, any sub-word value is internally sign extended to a 32-bit integral value. Hence ReinterpretHF2S should sign extend 16-bit value.

One may argue that this extension should be done at use site but given that JVM sign extends short value while loading it into a local variable and operations over operand stack are always performed at integral granularity, hence de-facto implementation is that compiler should sign-extend the results at definition site i.e., the moment a sub-word value is produced to match JVM semantics.

Performance data for FP16 operation



FP16 Super Word Level Parallelism Support: Carrier type of Float16 generating nodes is set a T\_SHORT based on Opcodes, on the similar lines to existing implementation.



**Double rounding for Float16 Operations: [1] :-**

A close up of text

Description automatically generated

For Float16

p = 11 (Float16 precision)

q = 24 (Float32 precision bits)

F16 => F32 OPER FP32 => FP16

(Double Rounding)

=> a) Intermediate computation happens at float32 granularity. (one rounding)

=> b) Second rounding down casting.

p (11) <= (q(24) -1) / 2 i.e. 11 <= 11

So double rounding is effective.

================================

Q. Why do we inject ReinterpretS2HF and HF2S instead of casting nodes to FP32?

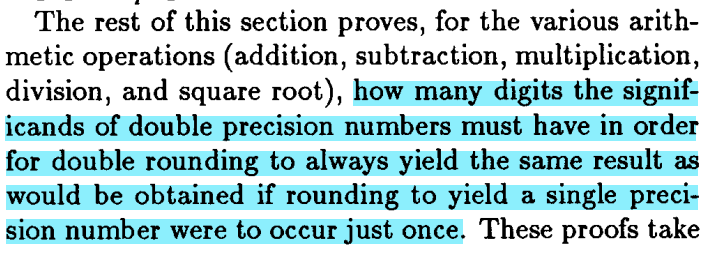
A. We have crafted specialized IR nodes for Float16 operations, which translates into FP16 ISA. All FP16 operations expects to receive operands in 16-bit IEEE FP16 format, thus a re-interpretation node meagrely moves the short value from GPR to floating point register.

In case inputs were constants, we translate them to equivalent Floating point constants (TypeF), these floating-point constants are then constant folded by existing value routines of floating-point parent IR nodes of new FP16 IR nodes.

Folded FP32 constants are then translated back to FP16 constants by runtime helpers called by values routines associated with HF2S reinterpret node.

All the new Float16 intrinsics perform operations at FP32 granularity in their fallback implementation. Following proofs verify the precision correctness of performing FP16 operations at FP32 granularity and effect of double rounding on final result.

Note: This is only relevant for picking correct floating-point type for fallback implementation of a FP16 operation. In Float16 class we chose to perform most of the Float16 operations at FP32 granularity.



A text on a white background

Description automatically generated

Holds good for Float16 + Float16 => fp32res = float16ToFloat + float16ToFloat => fp16res = floatTofloat16. Thus, double rounding due to performing operations at FP32 granularity will not induce rounding error more than what a single rounding of an inexact result of operation at FP16 granularity could have caused.

Float16 : Precision => 11 bits

Float32 : Precision => 24 bits

A text on a white background

Description automatically generated

Holds good for Float16 - Float16 => fp32res = float16ToFloat - float16ToFloat => fp16res = floatTofloat16. Thus, double rounding due to performing operations at FP32 granularity will not induce rounding error more than what a single rounding of an inexact result of operation at FP16 granularity could have caused.

A text on a white background

Description automatically generated

Holds good for Float16 \* Float16 => fp32res = float16ToFloat \* float16ToFloat => fp16res = floatTofloat16. Thus, double rounding due to performing operations at FP32 granularity will not induce rounding error more than what a single rounding of an inexact result of operation at FP16 granularity could have caused.

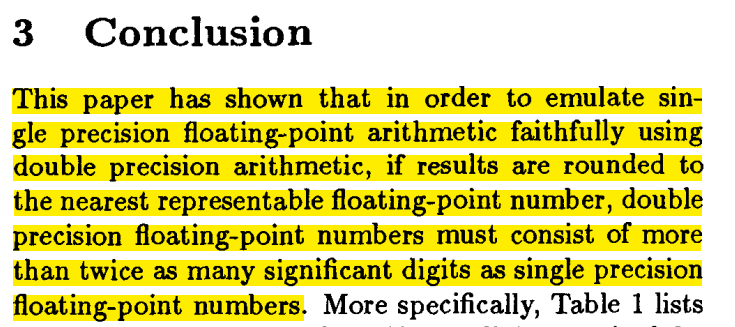
A text on a page

Description automatically generated

Holds good for Float16 / Float16 => fp32res = float16ToFloat / float16ToFloat => fp16res = floatTofloat16. Thus, double rounding due to performing operations at FP32 granularity will not induce rounding error more than what a single rounding of an inexact result of operation at FP16 granularity could have caused.

A text on a white background

Description automatically generated



References:

[1] <https://dl.acm.org/doi/pdf/10.1145/221332.221334>

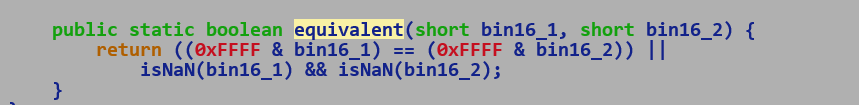
Using AVX512-FP VMOVW for moving FP16 value from / to GRP

Upper bits [16:32) of GPR are zeroed out, it will violate following assumption.

A screenshot of a computer

Description automatically generated

Existing functional test equivalence routines consider upper 16 bits while comparing integer promoted short (FP16) values, this looks incorrect and comparison should be done after masking out upper word.



This is specially useful with VMOVW.

Pending Scalar Operations:

* Conversions
  + To / From FP16 to primitive types.
  + Floating point special value APIs (NaN, Inf, IsFinite)
    - FPCLASS instructions.
* Developer Productivity: -
  + Template based assembler layer.
  + Avoid repetition (ARM has a similar assembler.)
* Primary and secondary opcodes.
  + BinOpNode -> Op\_BinOpNode
    - Op\_BinOpAddHF
    - Op\_BinOpSubHF
    - Op\_BinOpMulHF
    - Op\_BinOpDivHF
    - Op\_BinOpMaxHF
    - Op\_BinOpMinHF
  + Classes.hpp
    - Define new macro for secondary opcodes
      * macro(primary, secondary)
      * It should perform IR check for primary opcode, but may use secondary opcode for match\_rule\_supported checking.

bool match\_rule\_supported\_shared\_vector() {

}

bool match\_rule\_supported\_shared (primary\_opcode, secondary\_opcode) {

bool retVal = Match\_rule\_supported(primary\_opcode);

switch(secondary\_opcode) {

}

}

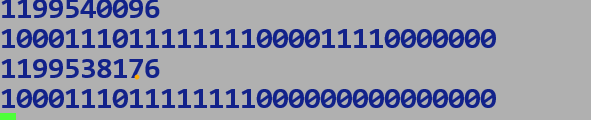
A screenshot of a computer program

Description automatically generated



Round towards nearest policy for float to float16.





High level planning Float16:-

1. Port existing Halffloat Java side implementation to lworld+fp16 using Float16 backing storage.
2. Validate and checkin into lworld+fp16.
3. Enable Intrinsic layer changes to test unified backend implementation in lworld+fp16
4. Enhance lworld+vector to support non-primitive type multifields.
5. Merge lworld+fp16 into lworld+vector.
6. Merger lworld+vector into lworld.

LaneType.asFloating / asIntegral methods return the size compatible floating-point type for a given integral lane.

These are queried by viewAsIntegralLanes() and viewAsFloatingLanes() APIs.

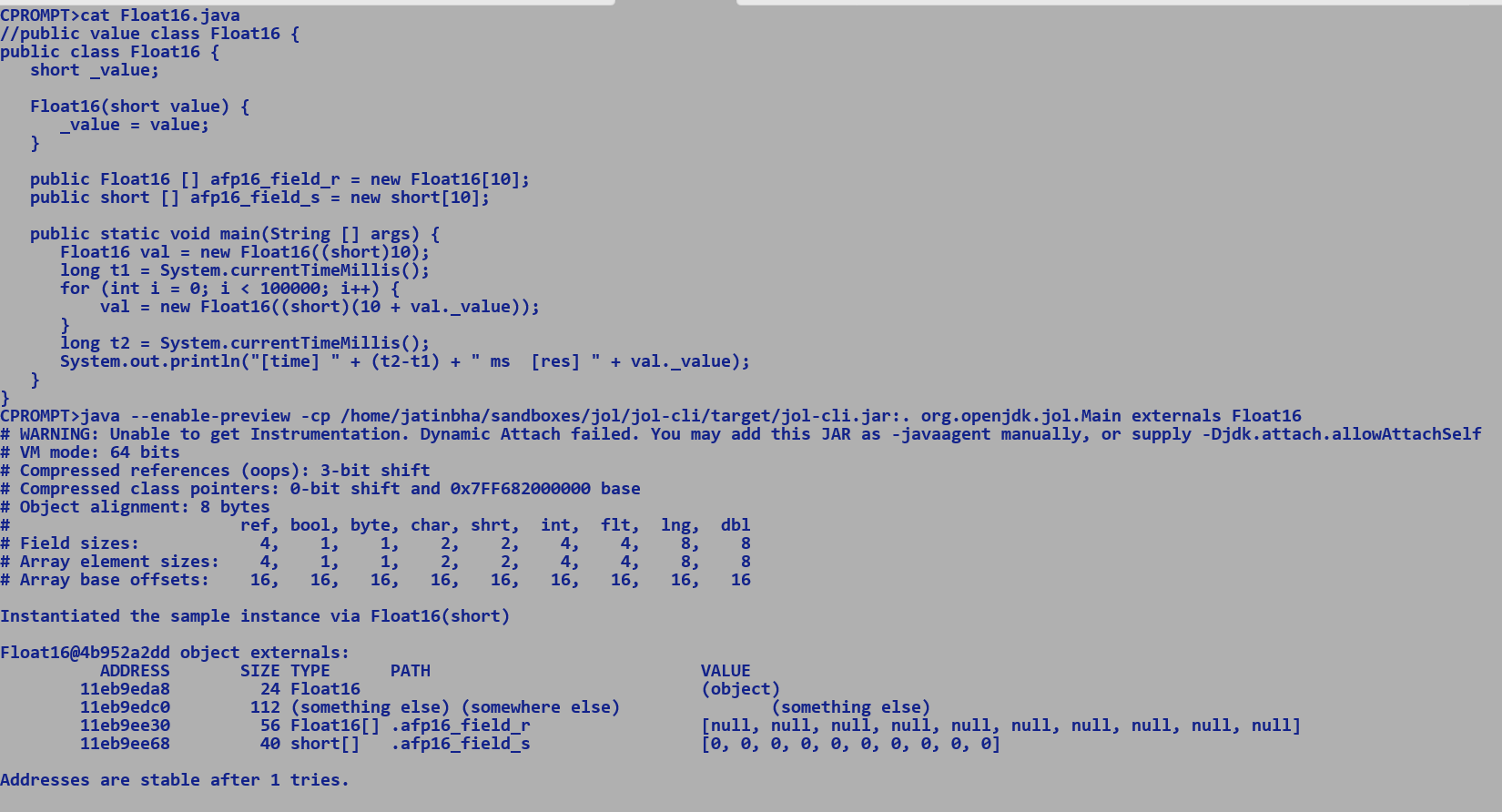
* For Byte currently we do not have a size compatible floating-point type, hence we throw an exception, in future FP8 will be size compatible with 8-bit integral type (byte).
* For Short, this can be agnostic to FP16 target support, we should interpret short lane as a float16. It will allow seamless translation b/w ShortVector and Float16Vector, giving user much needed flexibility.

Vector API fallback implementation can handle such scenario on a non-FP16 target.

Comparison operations

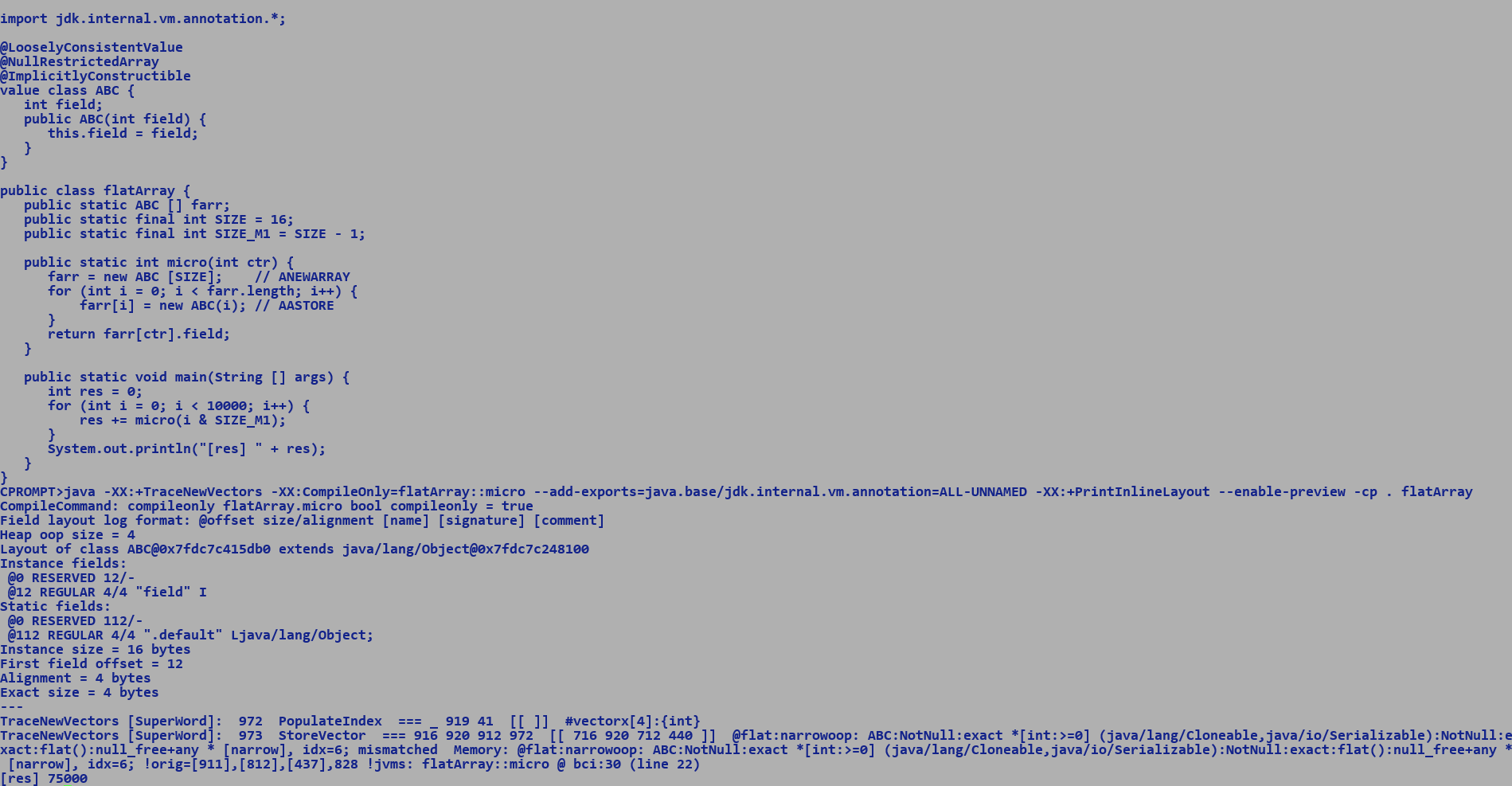
* Float16 being a value type follows value comparison and not referential comparison semantics.

Object Layout of Flat vs Non-flat arrays:-





Flat layout of arrays of values:-



Flat array layout of simple value types promotes vectorization since address of consecutively laid of elements differ by primitive type size.

Removing NullRestrictedArray annotation prevent flat layout.

A screenshot of a computer

Description automatically generated

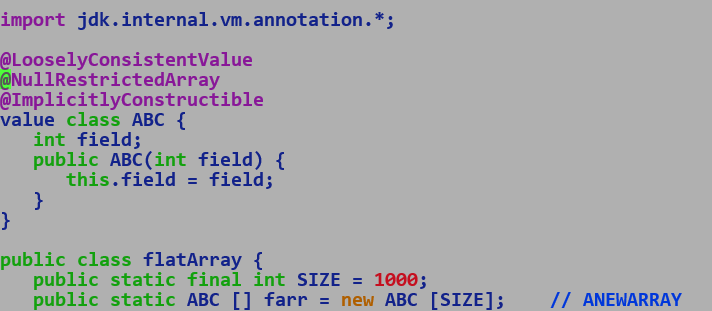
Other method is to dump layout using -XX:+PrintFlatArrayLayout (currently broken)

For flat arrays, null restriction and implicit construction is mandatory, which not only ensures non nullity on creation but also during the lifetime of array. This simplifies the JVM model for arrays of float16, since there is no means to assign null values to elements, as it simplifies the complexity in determining the layout and ensuring flatness.! ensures null restriction, ? indicates may be null i.e. either null or not null.

Float16 is null restricted and implicitly constructible and arrays of float16 are flat arrays…**enforced through jdk internal annotations**. Flat layout is enforced at the time array allocation and layout is preserved across all the execution engines. Interpreter, C1 and C2 all handler flat array accesses.

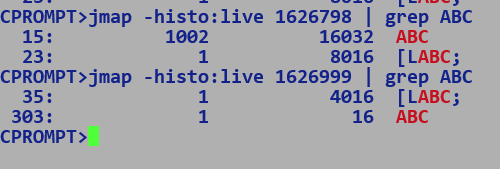
Thus, a flat array of Float16 will always be backed by short array.

e.g. Consider following static array of value objects and its heap sizes with and without flat array layout (no NullRestrictedArray annotation).



java --enable-preview -XX:-UseCompressedOops -Xint flatArray [Non Flat Array Layout, comment out @NullRestricredArray annotation]

java --enable-preview -XX:-UseCompressedOops -Xint flatArray [ Flat Array Layout]



Entry corresponding to array [LABC;

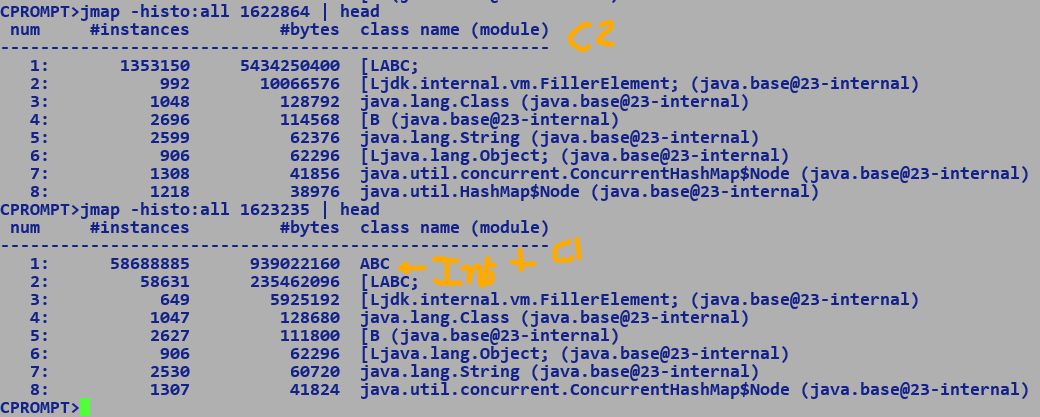
* Case 1 = 1000 x 8 byte reference + 12 byte header + 4 byte length = 8016
* Case 2 = 1000 x 4 byte fields + 12 byte header + 4 byte length = 4016

MemorySegment Refresher:-

* On Heap :- Array backed.
* Off Heap :- Native
  + Get / set APIs only operate at primitive type granularity OR AddressLayout which is an address of another memory segement.
  + FromMemorySegment / intoMemorySegment handles vectors load / store.

C1 + Int -> allocates on value objects on Buffer (Heap)

C2 -> Creates an implicitly scalarized representation of value objects (InlineTypeNode) and only buffer them iff needed.



Thus value object are help allocated but are still immutable and identity less, identity comparison with == is a deep comparison.

Arrays of values have similar behaviour to usual arrays in terms of allocation, still allocated over heap, elements may have flat layout.

Q. Why do we not allocates regular objects / store addresses to class instances into memory segments?

A. Native memory segments cannot hold and an address to heap memory, since native memory is outside GC purview and object movement will invalidate location references.

Whereas heap backed memory segments are based over primitive type arrays, GC only iterates of arrays of objects to refresh liveness of pointees.

BigDecimal

* BigInteger -> significand
* Scale -> scale by which significand is divided (if +ve) or multiplied if (-ve)
* Precision -> number of significands

floatValue()

* intCompact / scale if scale is +ve
* intCompact \* scale if scale is -ve
* intCompact if scale is 0.

A screen shot of a computer program

Description automatically generated

* fullfloat16Value

BigI q = < >

BigI r = < >

**-Inf < -MAX\_NORMAL < -MIN\_NORMAL < -MIN\_VALUE < -0.0 == +0.0 < MIN\_VALUE < MIN\_NORMAL < MAX\_VALUE < +Inf**

* Minimum +ve floating point value greater than +0.0 is MIN\_VALUE, and +ve value less than MIN\_VALUE becomes +0.0, even some targets assume sub-normal values as 0.0, compilers in fast math mode assumes sub-normals as zeros.

Float16.ZERO -> ULP

* Float16.MIN\_VALUE – Float16.ZERO

A screenshot of a computer program

Description automatically generated

Precision

Scale

Value / Scale if scale > 0 else Value \* Scale

Rounding

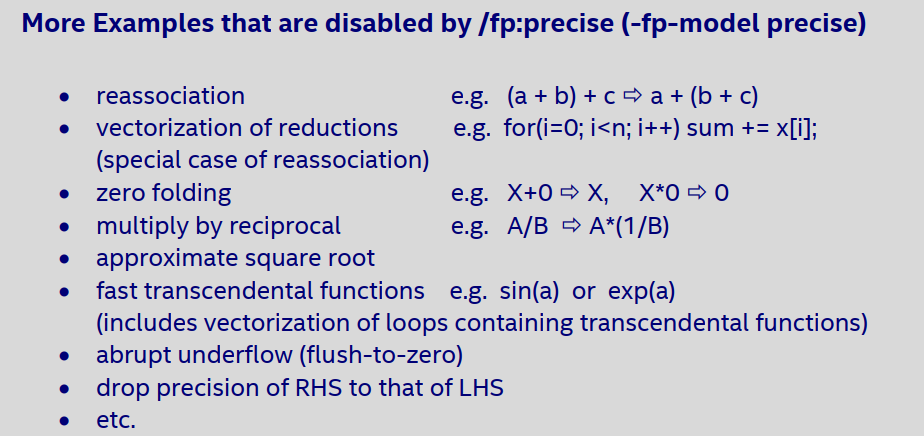
* To nearest / even
* To zero

A screenshot of a computer program

Description automatically generated

A white background with blue text

Description automatically generated



A screenshot of a math problem

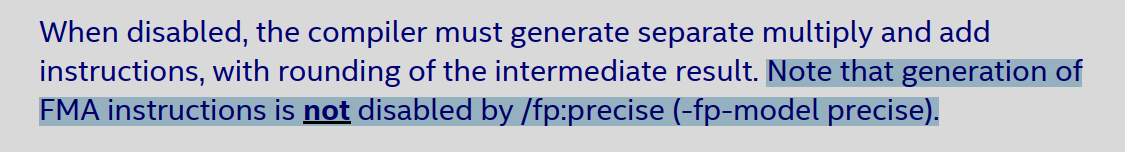
Description automatically generated



A screenshot of a computer

Description automatically generated





A close-up of a white background

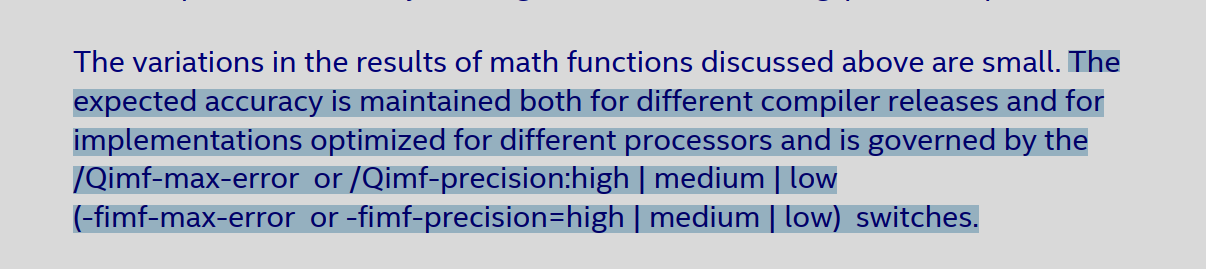
Description automatically generated

A close-up of a computer program

Description automatically generated

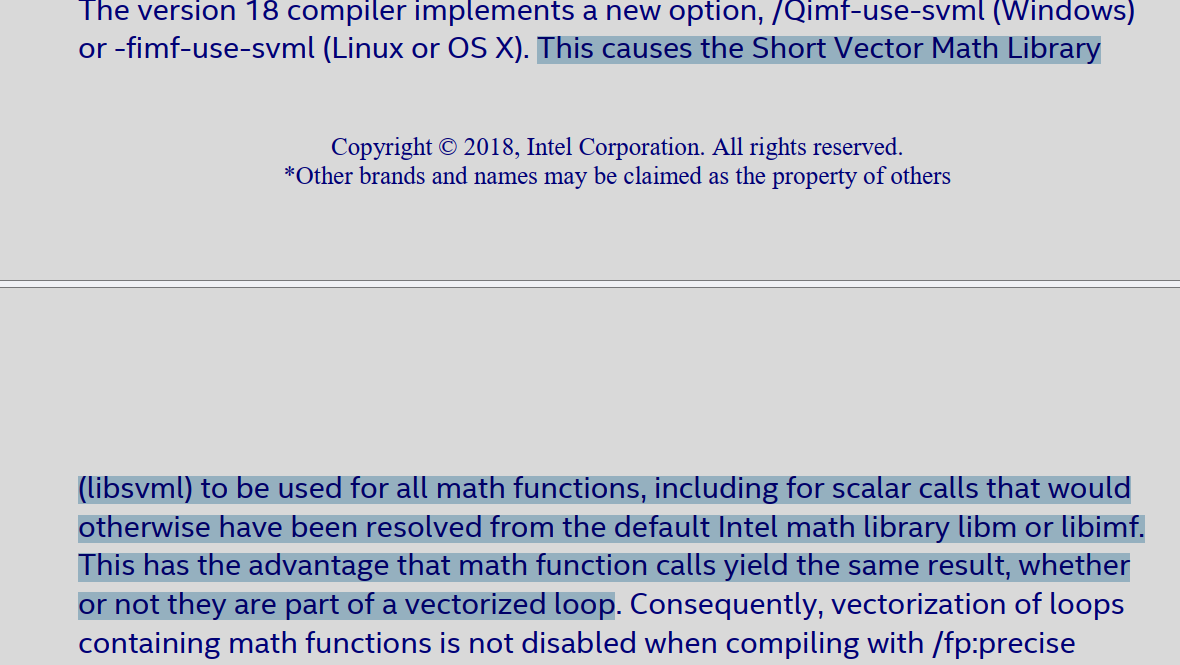
A close-up of a text

Description automatically generated



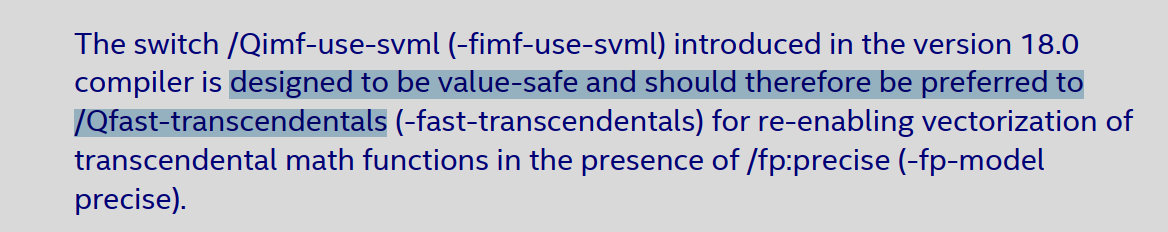
A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated



Float16.fma encoding:-

vfmadd132ph xmm1, xmm0, xmm1

E.V’ VVVV - NDS (opr2)

MODRM.REG : DST (opr1)

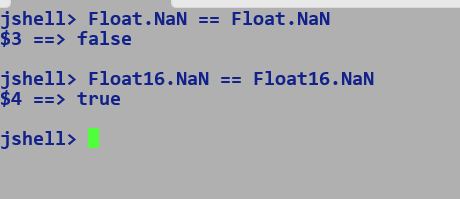
MODRM.R : SRC2 (opr3)

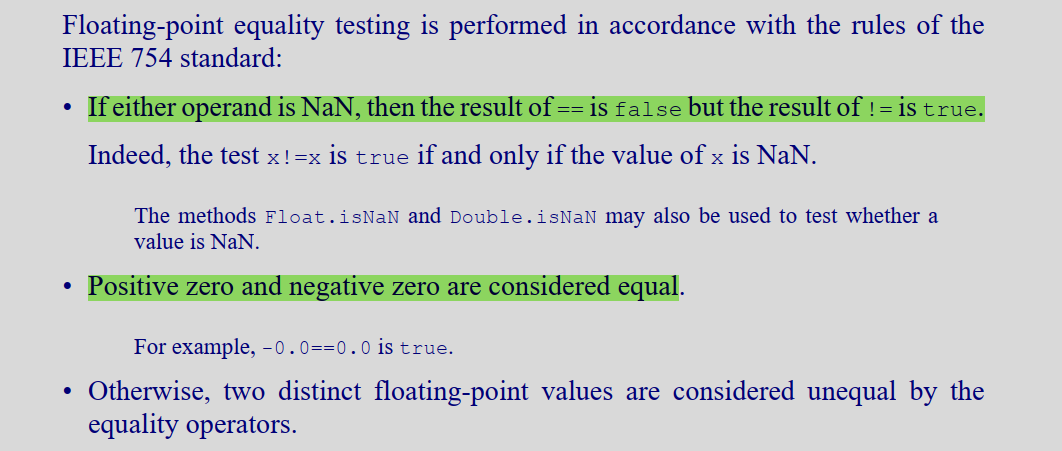
Xmm0 : 00000 : bit inverted 11111

Xmm1 : 00001 : bit inverted 11110

* EVEX : 62 [1111 0110] [0111 1101] [1000 1000]
* 62 F6 7D 88
* OPCODE
* MODRM 1100 1 001
* C9
* SIB [ NA]
* IMM / DISP / DISP8 (compressed)

[NA]

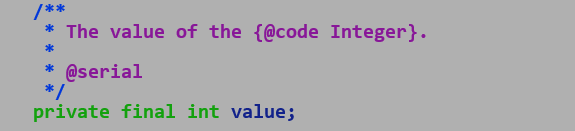




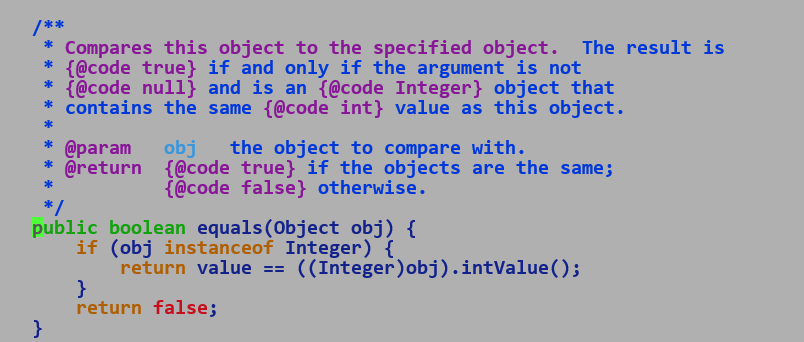
**Value-based Classes**

Some classes, such as java.lang.Integer and java.time.LocalDate, are *value-based*. A value-based class has the following properties:

* the class declares only final instance fields (though these may contain references to mutable objects);



* the class's implementations of equals, hashCode, and toString compute their results solely from the values of the class's instance fields (and the members of the objects they reference), not from the instance's identity;



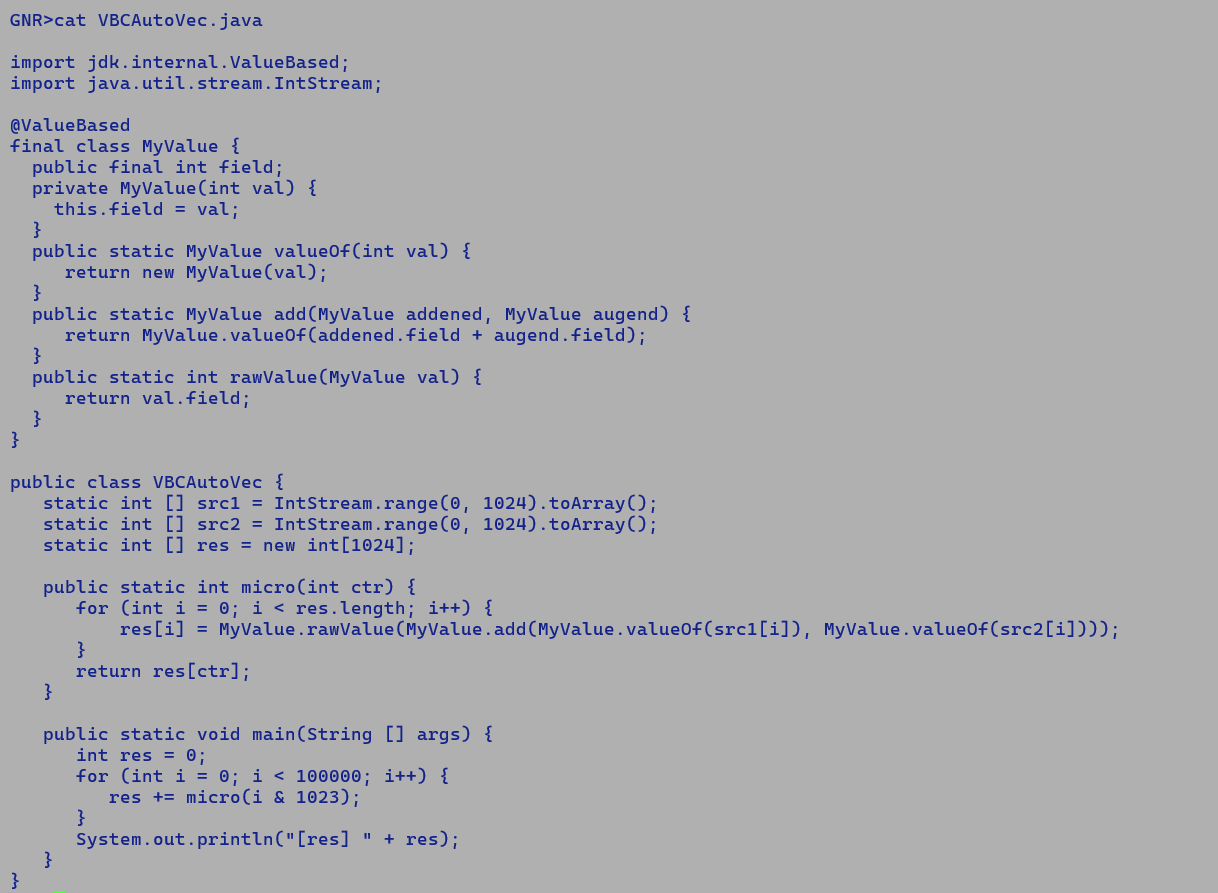
* the class's methods treat instances as *freely substitutable* when equal, meaning that interchanging any two instances x and y that are equal according to equals() produces no visible change in the behavior of the class's methods;
* the class performs no synchronization using an instance's monitor;
* the class does not declare (or has deprecated any) accessible constructors;
* the class does not provide any instance creation mechanism that promises a unique identity on each method call—in particular, any factory method's contract must allow for the possibility that if two independently-produced instances are equal according to equals(), they may also be equal according to ==;
* the class is final, and extends either Object or a hierarchy of abstract classes that declare no instance fields or instance initializers and whose constructors are empty.

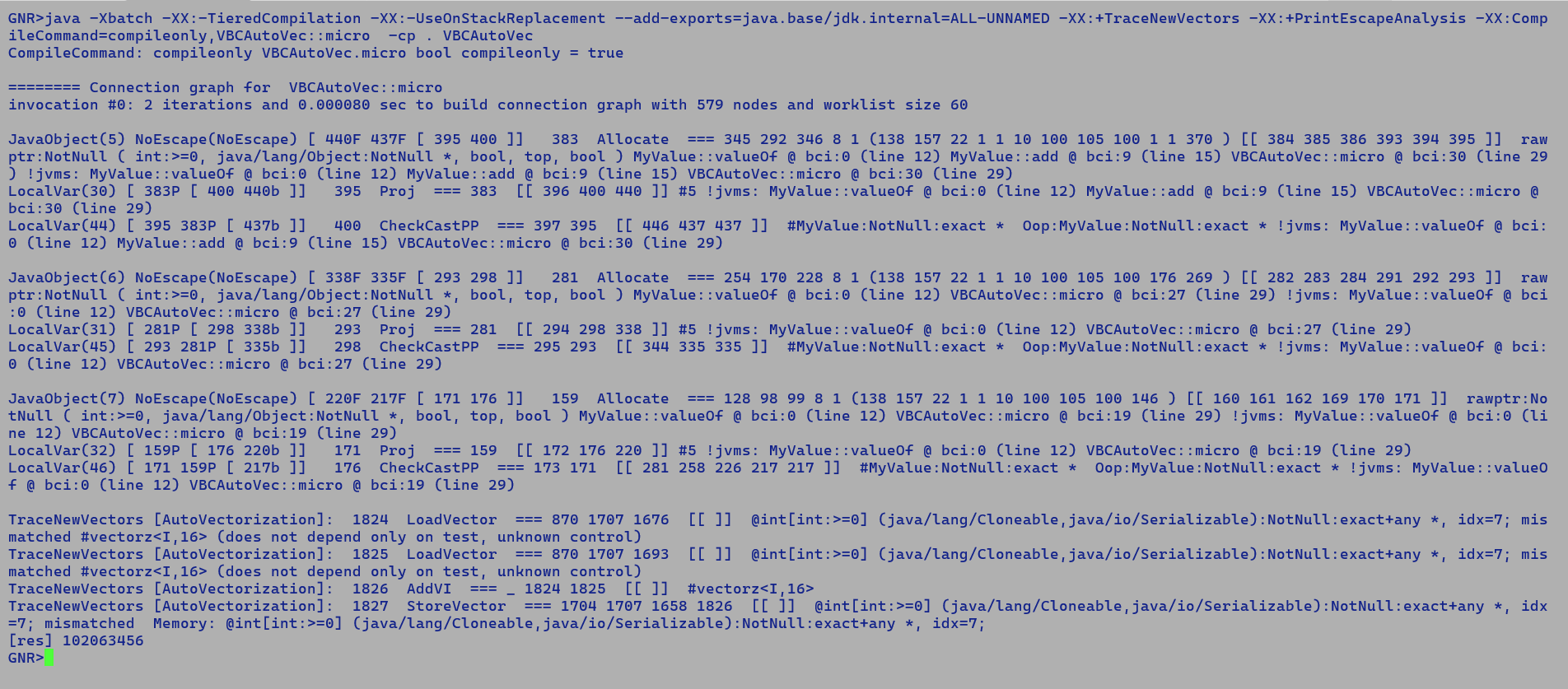
When two instances of a value-based class are equal (according to `equals`), a program should not attempt to distinguish between their identities, whether directly via reference equality or indirectly via an appeal to synchronization, identity hashing, serialization, or any other identity-sensitive mechanism.

Synchronization on instances of value-based classes is strongly discouraged, because the programmer cannot guarantee exclusive ownership of the associated monitor.

Identity-related behavior of value-based classes may change in a future release. For example, synchronization may fail.

Auto-Vectorizing ValueBased classes instance arrays.





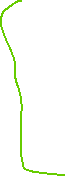
FP16 FP16 FP16 FP16

| | | |

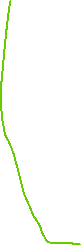
ConvHF2F ConvHF2F ReinterpretS2HF ReinterpretS2HF

| | | |

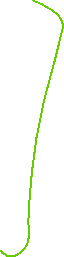
\ / => \ /



AddF AddHF



| |



| ReinterpretHF2S

| |

| ConvHF2F

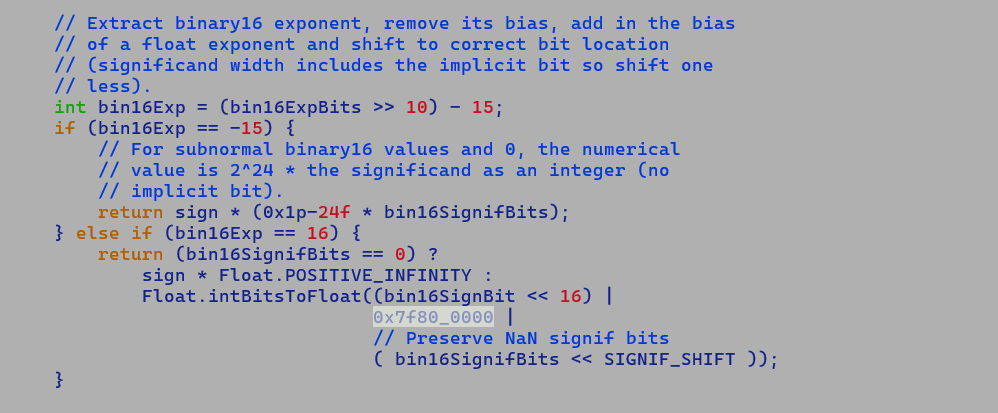
ConvF2HF |

| ConvF2HF

FP16 |

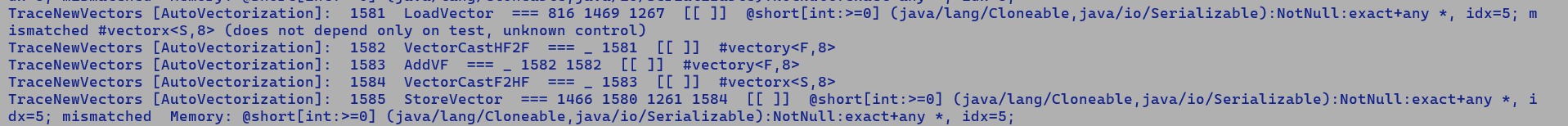
| FP16

float16ToFloat :-

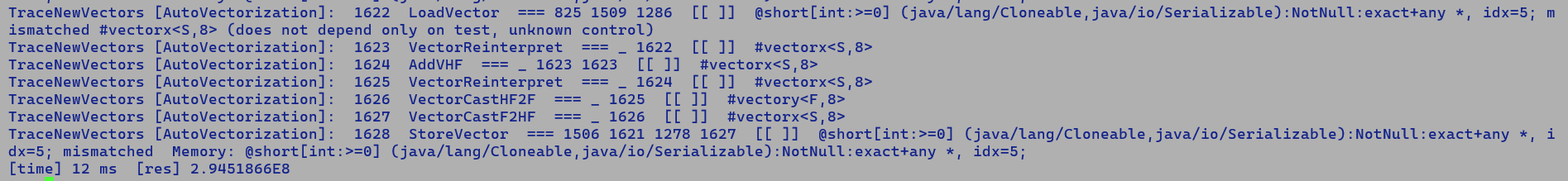


SNaN => 0x7c01 => Converted to FP32 QNaN with significand preserved, thereby making CastHF2F + CastF2HF infeasible.

User’s code : Float.floatToFloat16(Float.float16ToFloat(a) + Float.float16ToFloat(a))



We intend to replace : ConvF2HF (AddF (ConvHF2F a) (ConvHF2F a)).

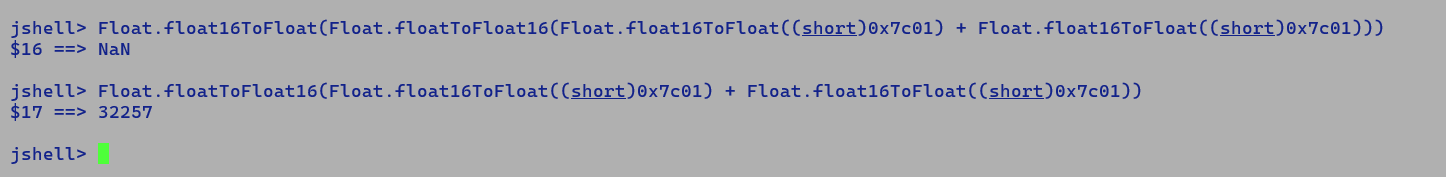


Replacing this at AddF, will need injection of additional ReinterpretHF2S + ConvHF2F to preserver IR type semantics.

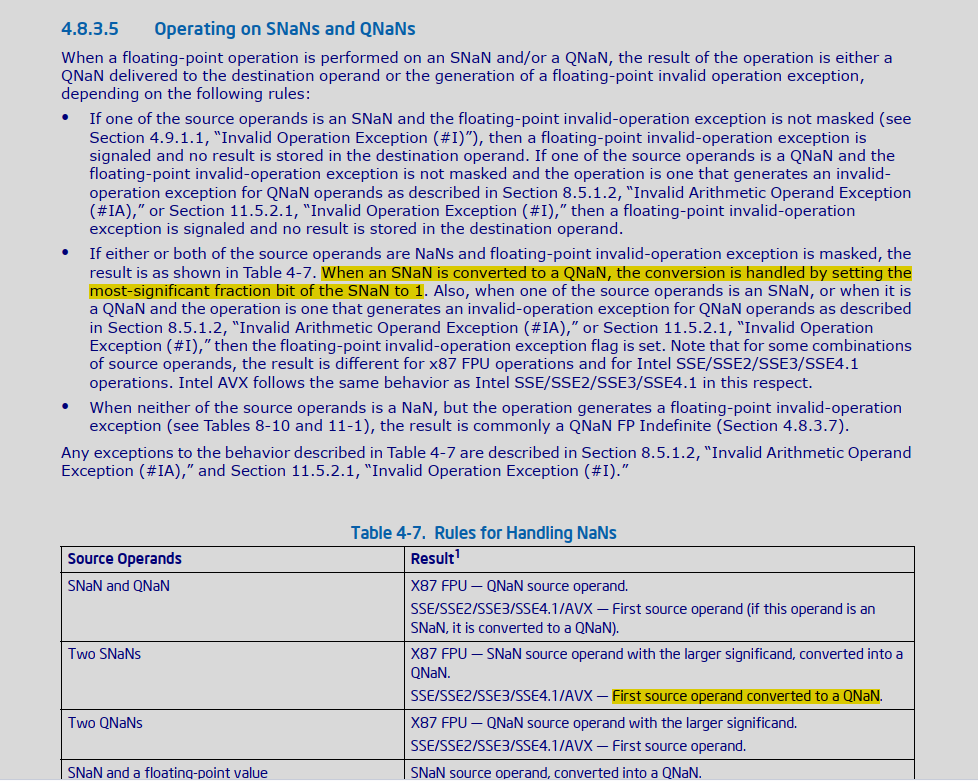
Replacing it at (VectorCastF2HF (AddF/MulF/SubF/DivF/MaxF/MinF) VectorCastHF2F) => Should be ok.

floatToFloat16((float)Math.sqrt(Float16ToFloat(in)))

ConvHF2F -> ConvF2D -> SqrtD -> ConvD2F -> ConvF2HF



So does intrinsic, VPADDHF if any of its operand is SNaN or QNaN



Float16 Format:-

S EEEEE MMMMMMMMMM

* Normalized 1.0 representation.
* Min Normal exponent: -14
* Max Normal exponent: 15
* Min sub-normal exponent: -24

Full exponent range : 15 – (-24) + 1 = 40 bits

Rounding Rules:-

EXP > MAX\_EXP = Result = +Inf

Inexact result, rounded using rounding control bits (embedded or MXCSR)

* RC = 00 : round to even, out of two exact (representable) bracketing values surrounding the inexact result, pick the value with LSB set to zero
* RC = 01 : round towards -ve inf , pick minimum value greater than -ve inf.
* RC = 10 : round toward +ve inf, pick maximum value less than inf.
* RC = 11 : truncate the extra bits that cannot be accommodated into significant bits.

Example:-

<https://github.com/jbhateja/playground_snippets/blob/master/c/round/test.c>

In the fallback implementation, Float16 FMA is performed using double precision type.

Double has 53 significand bits. We first compute the product, followed by addition.

The maximum exponent range needed to accommodate an infinitely precise product of two Float16 values is 80 bits. This is because, in the extreme case, one of the inputs could be a minimum sub-normal value 0x1.0P-24, whereas the other could be a maximum normal value 0x1.ffcP+15. Thus, we need 40 bits (15 – (-24) + 1) to accommodate the precise value range of each input.

* A \* B
* 40-bit x 40 bit => 80 bits

Clearly, the product cannot always be accommodated into 53 significand bits of a double-precision number. Therefore, we need to justify that such cases will not impact the precision after final rounding to a Float16 value.

There are 3 cases that we need to address

1. The exponent of the precise product is greater than MAX\_EXPONENT (15), in such a case we expect and +ve Infinite result, and double rounding will be innocuous e.g. if the exact product needs 60 bits which is greater than 53 significand bits of double precision number then first we need to round off the result to fit into 53 significand bits and second rounding is needed to represent the result into Float16 format.
2. A precise product is accommodated in 53 significand bits of the double-precision number.
3. Inputs are in the sub-normal range and the exponent of precise produce is less than -24.

Case A)

In this case, also double rounding is innocuous.

0.0x0000000000 000000000000001 (X)

A value below a sub-normal range is represented as zeros.

Next Up Value (B) in this case will set the second LSB of the Float16 mantissa.

0.0x0000000000 000000000000010

Next down will be (A)

0.0x0000000000 (extras - ignored) 000000000000001

A < X < B

jshell> Float16.float16ToRawShortBits(Float16.valueOf(0x1.0P-23f - 0x1.0P-24f))

$25 ==> 1

jshell> Float16.float16ToRawShortBits(Float16.ulp(Float16.valueOf(0x1.0P-24f)))

$26 ==> 1

jshell> Float16.float16ToRawShortBits(Float16.nextUp(Float16.valueOf(0x1.0P-24f)))

$28 ==> 2

jshell> Float16.float16ToRawShortBits(Float16.valueOf(0x1.0P-23f))

$29 ==> 2

Case B)

0.0x0000,0000,10 000000000000001 (X)

nextUp precise value : (B)

0.0x0000000011 extra: 00000000000001

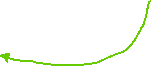
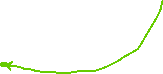
Lower Value: (A)

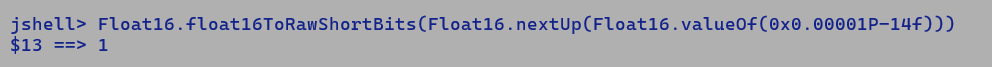
0x0x0000000010 extra (ignored) 00000000000001

A < X < B

A screenshot of a computer code

Description automatically generated

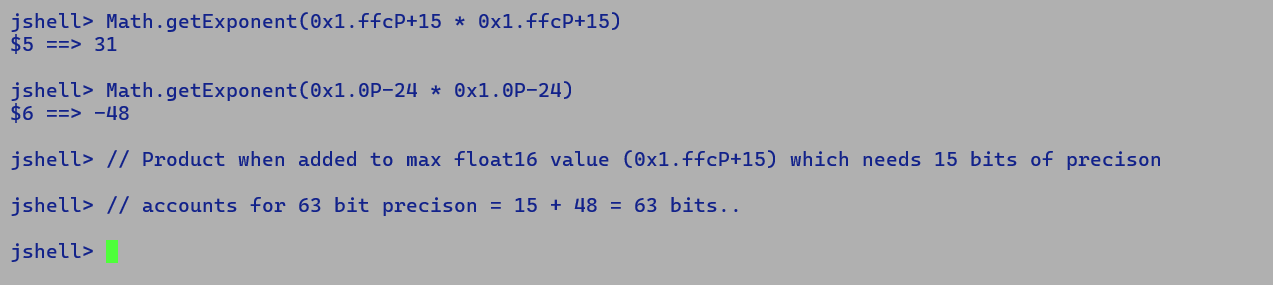




A screenshot of a computer

Description automatically generated

TBD : Consider other rounding modes (towards inf, away from inf, truncation)



**Auto-vectorization: -**

On targets supporting FP16 ISA, we leverage the full width of a vector with short (16-bit) lanes e.g. on x86 AVX512-FP16 targets we process 32 Float16 values with one SIMD instruction.

In all other cases, we promote a Float16 lane value to a Float value thereby only processing half the number of packed values in one SIMD operation.

**Intrinsics fallback inlining:**

VectorAPIs with default JVM flags are intrinsified incrementally, i.e. lazy intrinsification during optimizations, therefore compiler does not get an opportunity to perform procedure inlining in case of an inline expansion failure. This is because, LateInlineCallGenerator encapsulates only one CallGenerator i.e. LibraryCallKit, and does not include ParseGenerator as a fallback call generator to facilitate inlining. For FP16 we attempt a parse time intrinsification which facilitates inlining in the fall-through scenario. Thus, Sqrt, Fma, intrinsics should get inlined if are not inline expanded.

Thus, inline expansion and FP16 scalar IR creation are sensitive to the existence of necessary target features.

**Use case for Float16.isInfinite/isNan/isFinite**

Data parallel algorithms primarily have a single control flow, and any conditional divergence is handled through predicated operations under the influence of a mask.

Consider following the scalar kernel

for (int i = 0 ; i < 1024; i++) {

if (Float16.isFinite(src[i])) {

dst[i] = Float16.mul(src[i], src[i]);

} else {

dst[i] = Float16.add(src[i], src[i]);

}

}

When parallelized, we intend to stratify the control path under predication conditions computed for a range of packed float16 values.

SLP only operates at a basic block level, thus a prior if-conversion inferring conditional moves (cmov) appearing within a basic block can still be vectorized using VectorMaskCmp + VectorBlend combination.

Above use case is parallelized using explicit vectorization flow (VectorAP).

var vsrc = FloatVector.fromArray(SPECIES, src, index);

VectorMask<Float16> mask = vsrc.test(VectorOperators.IS\_FINITE);

vsrc.lanewise(VectorOperators.MUL, vsrc, mask)

vsrc.lanewise(VectorOperators.ADD, vsrc, mask.not());

Java Snippet:

if (Float16.isFinite(src[i])) {

dst[i] = float16ToShortBits(Float16.mul(src[i], src[i])); // Unwrap Boxes

} else {

dst[i] = float16ToShortBits(Float16.add(src[i], src[i])); // Unwrap boxes.

}

C2 Ideal IR: After intrinsification of isFinite.

IfFinite + (Additional Compare)

IsFiniteHF (TypeInt::BOOL) == 0 / 1 / Other bool vs CmpI

| AddHF

Bool (eq) MulHF |

| Bool | /

If | | |

/ \ Conditional Move Detection is only applicable to CMov .

IfTrue IfFlase integral register under influence of EFLAGS |

| |

\ /

\ /

Region MulHF AddHF

\ / /

\ / /

\ / /

PhiNode

|

StoreS

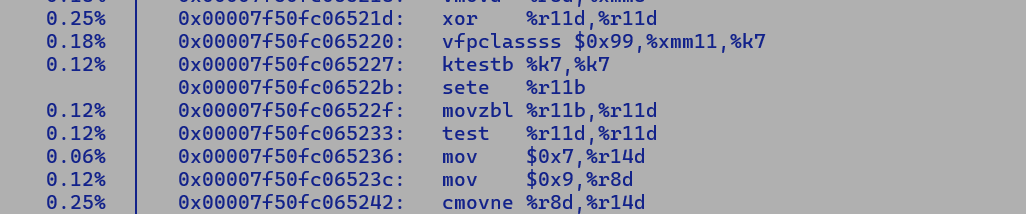
|

ifFinite => ABS<F> > LONG\_Max => Compare…

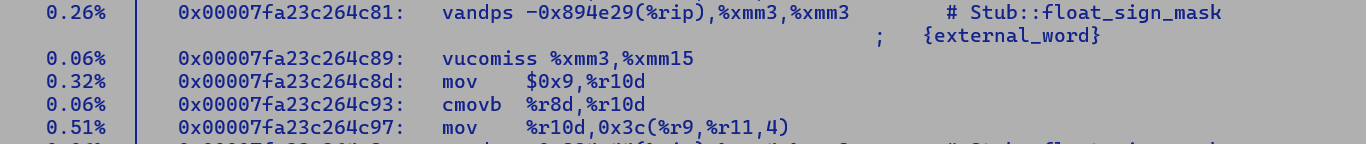
PhaseIdealLoop::conditional\_move modifications.

* + Existing non-intrinsic is implemented in terms of floating-point comparison there by automatically resulting into a Compare feeding into Bool and hence facilitates conditional moves detection.
  + With Intrinsic we end up generating addition compare post infinite/finite detection.

SuperWord level changes to pack scalar FPClass nodes into Vector IR of VectorMask type.



ifeq + invokestatic => Bool CmpI IsFiniteF SRC => CmpI I



+ NaN and Inf detection special case handling…

Float16:

S(1) EEEEE(5) MMMMMMMMMM (10)

* Float16 to Float to Integer => Float16 to Integer.

Float’s 24 bit precision can accommodate 10 precision bits of Float16, so does its exponent range.. (-127, 126).

**Half floating Point Constant folding: -**

1. TypeF – Holds floating-point constant value
2. Type::FLOAT – Represent full floating point range.
   1. We don’t do bit-level data flow analysis on floating point types, since FP32 or FP64 or FP16 numbers have a fixed format defined by IEEE.

There are two options:-

Option 1: (Hacky)

Use existing TypeF to hold FP constant and define new Value routines which perform operations at FP32 granularity and bring the result of computation into FP16 representable value range including NaN , +/- Inf encoded using TypeF.

i.e. FP16.MAX\_VALUE \* FP16.MAX\_VALUE is Inf in FP16 value range but it’s a non-inf value on the FP32 value range. Thus, the folded FP32 value should lie in the FP16 value range.

Matcher:

New selection patterns with constant operands, ConH (newly defined FP16 IR constant of TypeF with values strictly in FP16 range)

PhaseOutput:

ConF IR corresponding to FP32 constant when dumped occupies 32 bits corresponding to IEEE 754 FP32 value in constant section. Again, we need to perform FP32 to FP16 conversion and only save 16-bit value into the constant section. ConH will only dump 16-bit IEEE 754 FP16 value into the constant section.

Pros:

TypeF is well exposed to the compiler, we need minimum changes, TypeF associated with ConH will always represent a value in the FP16 value range (MAX\_NORMAL, MIN\_SUB\_NORMAL, +/-Inf, NaN).

Option 2:

Define new FP16 constant and non-constant types, namely TypeH and Type::HALF\_FLOAT with Base Types (HalfFloatConst, HalfFloat) and type convergence routines meet/join.

(Theory: lattice is a partial order where each pair has a LUB (lowest upper bound) and GLB (greatest lower bound), the lattice has two special elements, TOP which is a superset of all the values below it in the Hasses diagram, and BOTTOM is the lowest value representing a null value. A meet operation b/w two lattice points selects the GLB of two elements and takes us down towards the BOTTOM in the lattice, whereas a join operation selects the LUB of participating values and takes us higher in the lattice towards the TOP. BOTTOM and TOP values are necessary for convergence while monotonicity ensures unidirectional flow in lattice with each data flow analysis iteration i.e. application of flow function over participating lattice points)

New FP16 constant node, namely ConH of type TypeH. The ideal type associated with TypeH will be Op\_RegF, unlike TypeF, which is backed by a float value, the new type will use 16-bit short backing storage. The basic type associated with the new TypeH type is T\_SHORT, thus optimization passes consider it as a 16-bit value.

Define new runtime helpers for different operations (add, sub, mul, div, sqrt, fma) for the folding TypeH constants. New routines will perform operations at FP32 granularity and then downcast them back to the FP16 range.

New matcher patterns with ConH operands and constant table dumps will write 16-bit value in the constant section.

Option 2 also removes special handling based on opcode checks to infer lane types of packed half-float operations, currently, scalar FP16 IR carries a 32-bit float type, to pack the maximum number of FP16 operations in a vector, super word optimizer internally maps their container type to short. This kludgy fix is no longer needed with the newly defined HALF\_FLOAT type.

Q. Problem with ConF ?

A. During Auto-Vectorization, ConF replication constrains the operational vector lane count to half of what can be used for regular Float16 operation. Since, only 16 floats can be accommodated into 512 bit vector, hence all the vectors in use-def chain also operate on narrower vectors while ISA is capable of operating at double than inferred vector width. As a result we process only 16 Float16 values even though we can accommodate 32 Float16 in a 512 bit vector.

With ConH, we create a 16-bit constant during constant folding, hence auto-vectorization is able to pack 32 short into one vector, thereby realizing full vector width…

Q. Why do we not use existing TypeInt::SHORT instead of creating a new TypeH type ?

A. New TypeH is a special type, since its ideal type is RegF while its basic type is T\_SHORT, thus C2 always view it as a 16 bit value while register allocator assigns it a floating point register..

Q. Do we need to add explicit matcher patterns for immediate operands.

A. No, just one new loadConH pattern should be able to load the values into a floating point register, all existing scalar half float operation patterns have floating point operands.

Q. What makes TypeInt::SHORT different from Type::HALF\_FLOAT?

A. Ideal type associated with these two types differ because of which the former is assigned a GPR while the latter is assigned a Floating point register during register allocation.

**FP16 ISA Extension GAP**

Floating point values are always loaded into XMM register or x87 floating point registers.

Existing ISA have explicit instructions to broadcast single and double precision values





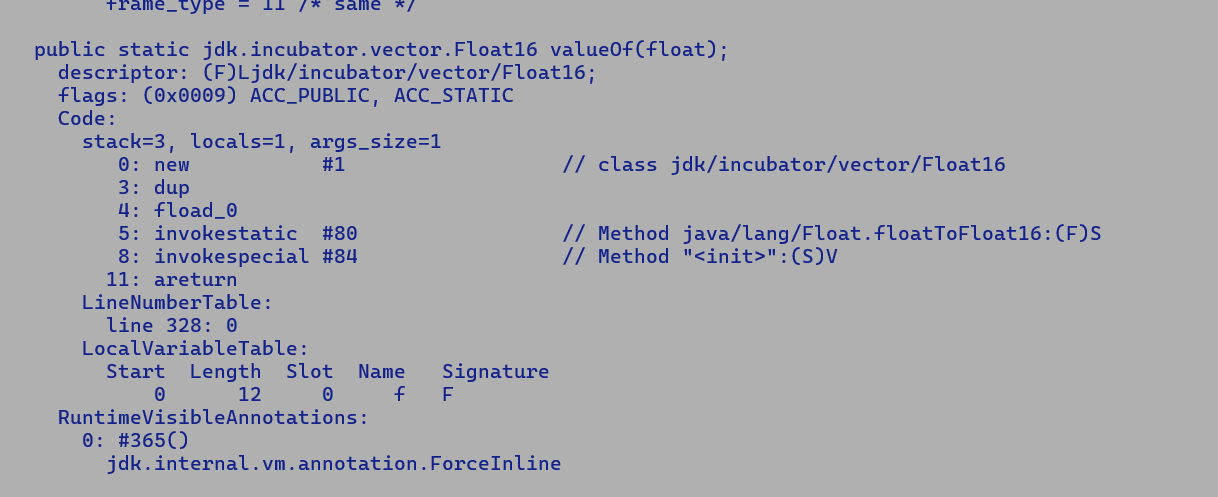
Integral broadcast instruction replicates the values held in GPR register across vector lanes.

Since FP16 arithmetic instructions operates over X/Y/ZMM registers, hence it will help our users to have a special broadcasting instruction “VBROADCASTSH” which will replicate lower 16 bits of xmm register.

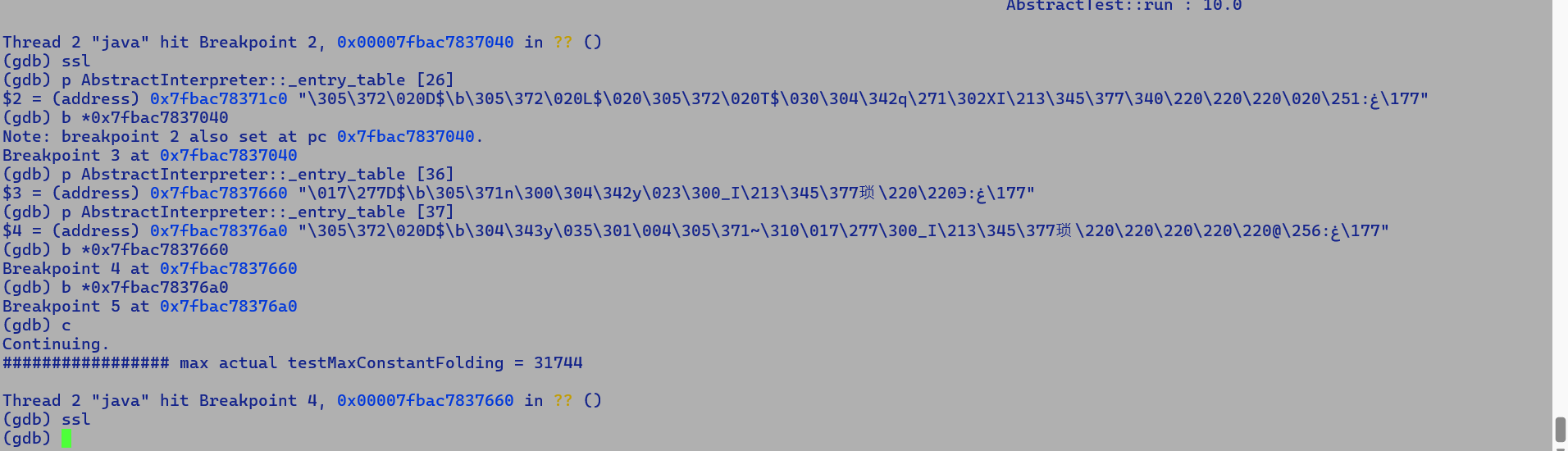
Calling convention: -

Float16 is passed as oop argument and not unboxed, so no change in calling convention.

Float16.valueOf(float)



valueOf(max(a.floatValue(), b.floatValue()))



Breakpoint : For PS2PH never hit , we directly land into PH2PS which passed infinity..

// static short Float16.float16ToShortBits() { return this.value;}

// Translates into getfield and must return short, these constants are later converted to half float constants during value transforms associated with ReinterpretS2HF IR.

// This preservers the program semantics at IR level.

**Primitive Box Type flow**

Float F1 = Float.valueOf(f1);

Float F2 = Float.valueOf(f2);

Float F3 = Float.valueOf(f3);

Float r1 = F1.floatValue() + F2.floatValue();

Float r2 = r1.floatValue() + F3.floatValue();

return r1.floatValue() + r2.floatValue();

Regular primitive type boxing APIs are lazily intrinsified, thereby generating an intrinsifiable Call IR during parsing.

LoadNode’s idealization can fetch a boxed value from the input of boxing call IR and directly forward it to users.

Q. What is the problem in directly passing Float16 boxes to FMA and SQRT intrinsic entries points?

A. Compiler will have to unbox them before the actual operation, there are multiple schemes to preform unboxing like: name based, offset based and index-based field lookup.

Vector API unbox expansion uses an offset based payload field lookup, for this it bookkeeps the payload’s offset over runtime representation of VectorPayload class created as part of VM initialization.

However, VM can only bookkeep this information for classes which are part of java.base module, Float16 being part of incubation module cannot use offset based field lookup. Thus only viable alternative is to unbox using field name / index based lookup.

For this compiler will first verify that incoming oop is of Float16 type and then use a hardcoded name-based lookup to Load the field value. This looks fragile as it establishes an unwanted dependency b/w Float16 field names and compiler implementation, same applies to index based lookup as index value are dependent on combined field count of class and instance specific fields, thus any addition or deletion of a class level static helper fields before the field of interest can invalidate any hardcoded index value used by compiler.

All in all for safe and reliable unboxing by compiler its necessary to create an upfront VM representation like vector\_VectorPayload.

Q. What are pros and cons of passing both the unboxed value and boxed values to intrinsic entry point?

A.

Pros:

* This will save unsafe unboxing implementation if holder class is not part of java.base module.
* We can leverage existing box intrinsification infrastructure which directly forwards the embedded values to its users.
* Also, it will minimize the changes in the Java side implementation.

Cons:

- Its suboptimal in case the call is neither intrinsified or inlined, as it will add additional spills before the call.

Q. Primitive box class boxing API “valueOf” accept an argument of corresponding primitive type. How different are Float16 boxing APIs.

A. Unlike primitive box classes, Float16 has multiple boxing APIs and none of them accept a short type argument.

public static Float16 valueOf(int value)

public static Float16 valueOf(long value)

public static Float16 valueOf(float f)

public static Float16 valueOf(double d)

public static Float16 valueOf(String s) throws NumberFormatException

public static Float16 valueOf(BigDecimal v)

public static Float16 valueOf(BigInteger bi)

Thus, we need to add special handling to first downcast the parameter value to short type carrier otherwise it will pose problems in forwarding the boxed values. Existing LoadNode idealization directly forwards the input of unboxed Call IR to its users. To use existing idealization, we need to massage the input of unboxed call to exact carrier size, so it’s not a meagre one-line change in following methods to enable seamless intrinsification of Float16 boxing APIs.

bool ciMethod::is\_boxing\_method() const

bool ciMethod::is\_unboxing\_method() const

In view of above observations passing additional 3 box argument to intrinsic and returning a box value needs additional changes in compiler while minor re-structuring in Java implementation packed with in the glue logic looks like a reasonable approach. BTW this is not the first instance where we are performing this intrinsic friendly restructuring.

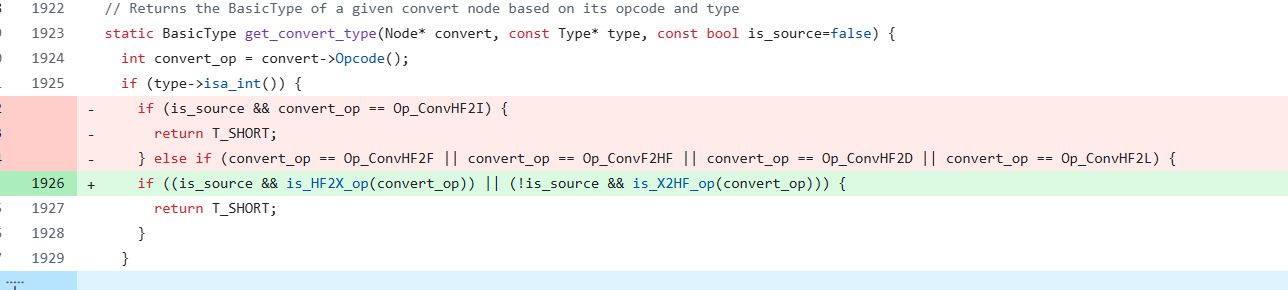
Double to float16 conversion.

There are very good chances of double to float16 conversion resulting into an inexact value, given that double number has 54 precision bits while float16 has 11 precision bits, thus any significand demanding more than 11 bits of precision with its exponent range within float16 exponent range needs to be rounded off to a representable number using prescribed rounding mode, default RC is round to nearest / even i.e. out of two float16 numbers bracketing the inexact number we select the one which has zero as its LSB bit.

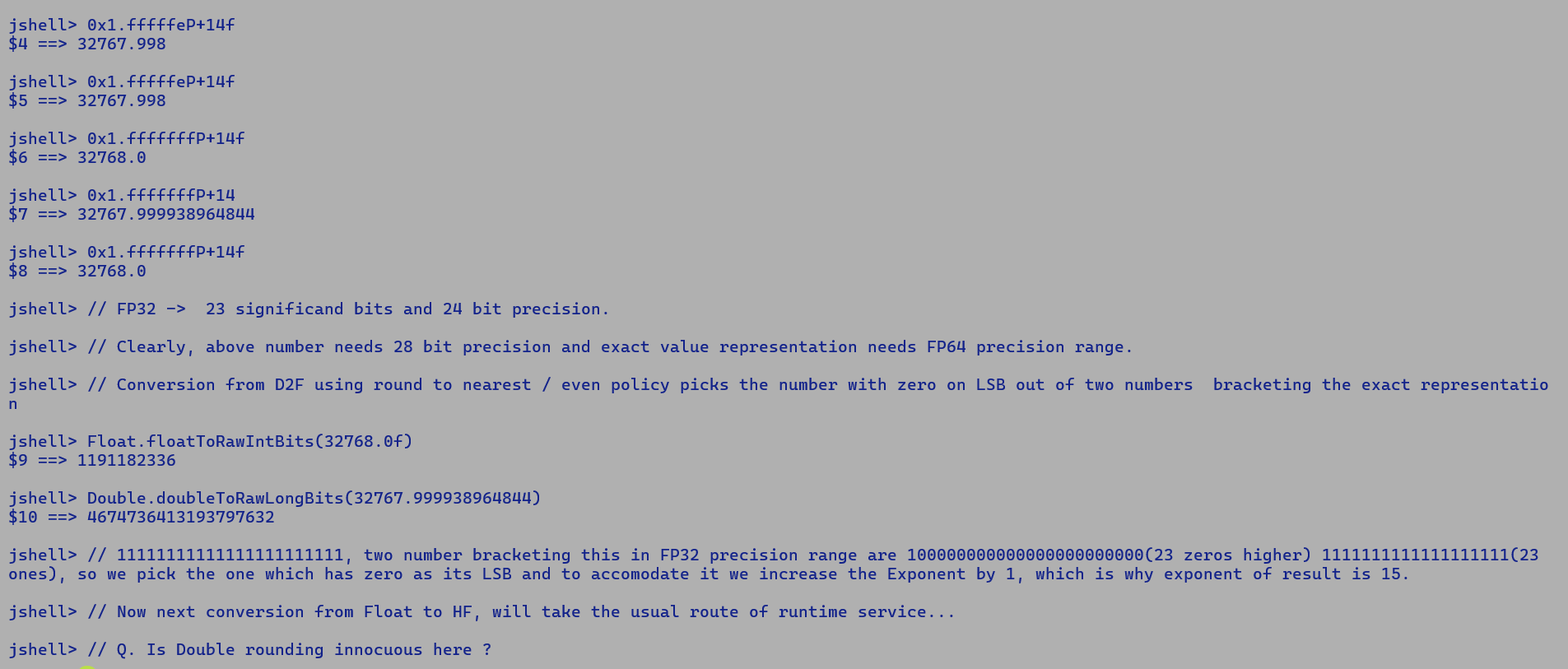
Float16 to Signed integer conversion.

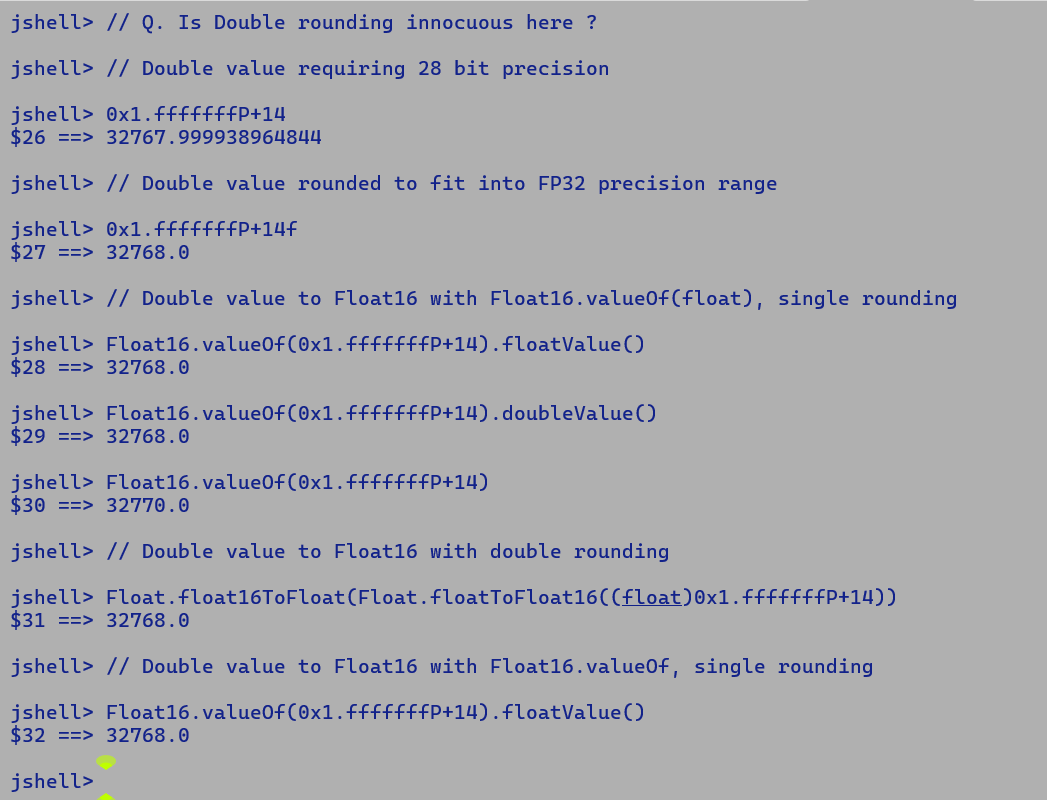
Since Float16 has 11 significand bits hence numbers withing +/- 2^11 are directly transferrable to 32 bit result, exponent range of Float16 varies from -14 to 15, largest possible float16 number of 0x1.ffcP+15 is 65504.0, thus entire value range of Float16 can be accommodated into integral value range, this is different from Float to integer conversion where exponent of float varies from -126 to 127, thus only a subset of floating point range can be accommodated into signed integral range.

Double to integral conversion is again narrowing, since only a subset of double numbers can be accommodated into singed integer value range.



This routine is only used by PhiNode idealization to move compatible conversion operations across phi nodes, thereby reducing the emission of expensive conversion operations in each converging branch. Some conversions especially Floating point to integral ones emit additional code to handle special cases and to avoid code bloating we try to move conversions across the PhiNode.





As per Amdahl’s law.

The estimated speedup with 50% parallelism is less than 2x in the best case.

Speedup = X / (0.5X / degree of parallelism) + 0.5X (scalar))

Even if the first term in the denominator becomes close to zero with infinite parallel resources, we could still achieve less than 2x performance.

Dot-Product

FP32 : 137 ops/ms

FP16 : 680 ops/ms

4.96x speedup.

**24/2/2025:**

**Quick Note VectorAPI FP16:-**

commit 7276a1bec0d90f63e9e433fdcdfd6564b70dc9bb

Author: Quan Anh Mai <qamai@openjdk.org>

Date: Fri Oct 11 15:28:15 2024 +0000

8341784: Refactor TypeVect to use a BasicType instead of a const Type\*

Reviewed-by: kvn, jkarthikeyan

The above patch replaced the element type (Type\*) with BasicType. We cannot have a new BasicType for the Float16 type, as basic types represent Java primitive type.

There are two options:-

1. Create a specialized IR (VectorMaskCmpHF) in applicable scenarios.
2. Partially revert the above patch.

This problem will not hurt us until VectorAPI FP16 implementation as direct comparisons are only possible through VectorOperators.EQ, LT, GT, etc. Existing scalar float16 APIs IsFiniter, isInfinite, etc are based on integral logical operations.

Float16.cmpareT(Float16) -> can be intrinsified to generate a new CmpHF node on the lines of CmpF and CmpD.

Auto-vectorizer can then generate a new VectorMaskCmpHF IR for the same and backed can emit FP16 comparison instruction VCMPPH.

Similarly, VectorOperator.EQ/LT etc. can leverage VectorMaskCmpHF.

* IsNaN -> with two compares is a good use case for APX CCMP
  + Conditional moves – not inferred.
  + In absence of CMove branches are not absorbed preventing any auto-vectorization which is applicable to single basic block.
  + Intrinsification candidate.
* To be taken in follow-up PR.
  + Existing Float/Double Floating point classification (isInFinite) is also not getting auto-vectorized.

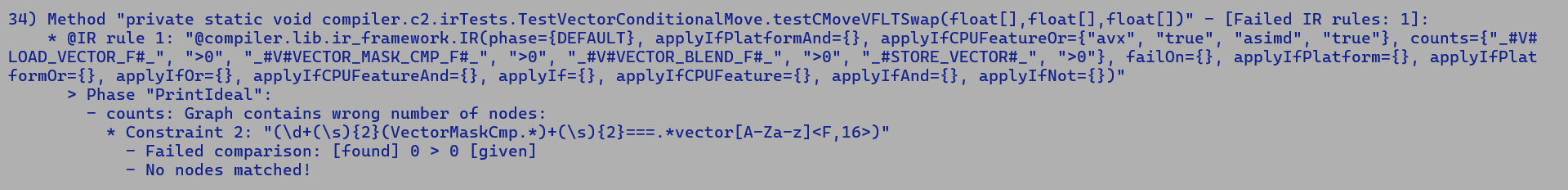
Existing Conditional Move handling is incorrectly always generating VectorMaskCmpNode of TypeVect,

This is sub-optimal and can be improved upon by setting the type to TypeVectMask on predicated targets.

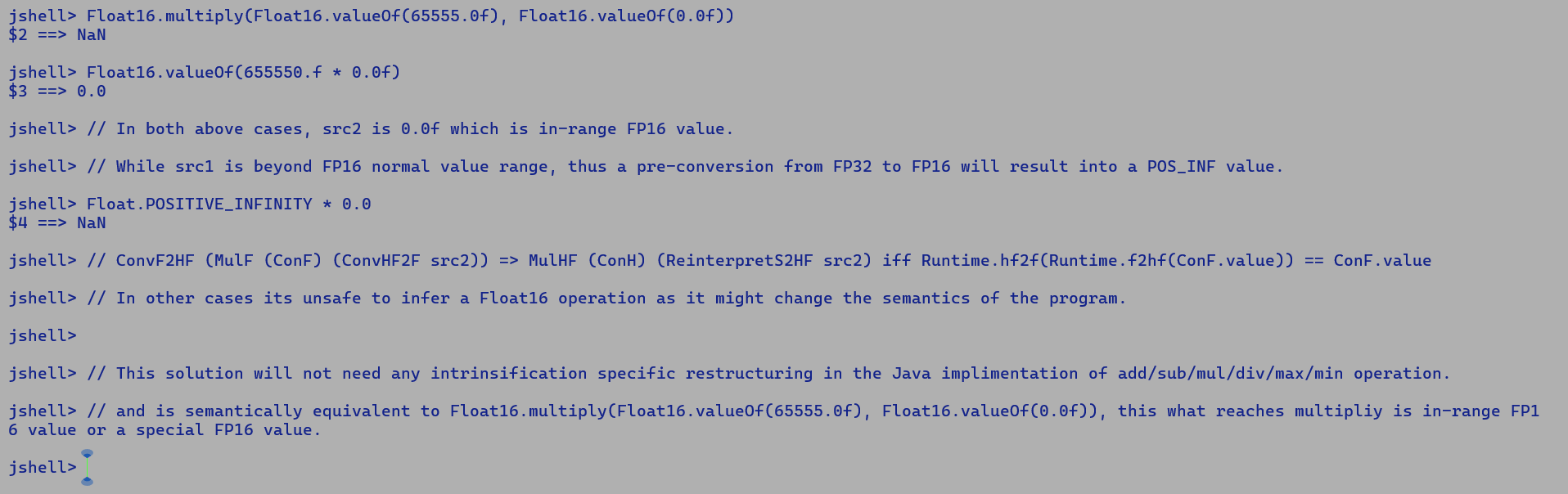
A screenshot of a computer

AI-generated content may be incorrect.

IR Framework expects type of Vector IR should be TypeVect, VectorMaskCmp has TypeVectMask as its ideal type. Vectmask and this causes IR verification failures.



[JDK-8352635](https://bugs.openjdk.org/browse/JDK-8352635) :Improve inferencing of Float16 operations with constant inputs.



Random values testing for Float16.isNaN

jshell> var f1 = Float16.shortBitsToFloat16((short)-860)

jshell> var f1 = Float16.shortBitsToFloat16((short)-860)

f1 ==> NaN

jshell> var f1 = Float16.shortBitsToFloat16((short)-860)

f1 ==> NaN

jshell> var f1 = Float16.shortBitsToFloat16((short)-390)

f1 ==> NaN

jshell>

Panama-Vector – FP16 support

* Payload : <https://github.com/openjdk/valhalla/blob/lworld/src/java.base/share/classes/jdk/internal/value/ValueClass.java#L65C35-L65C66>

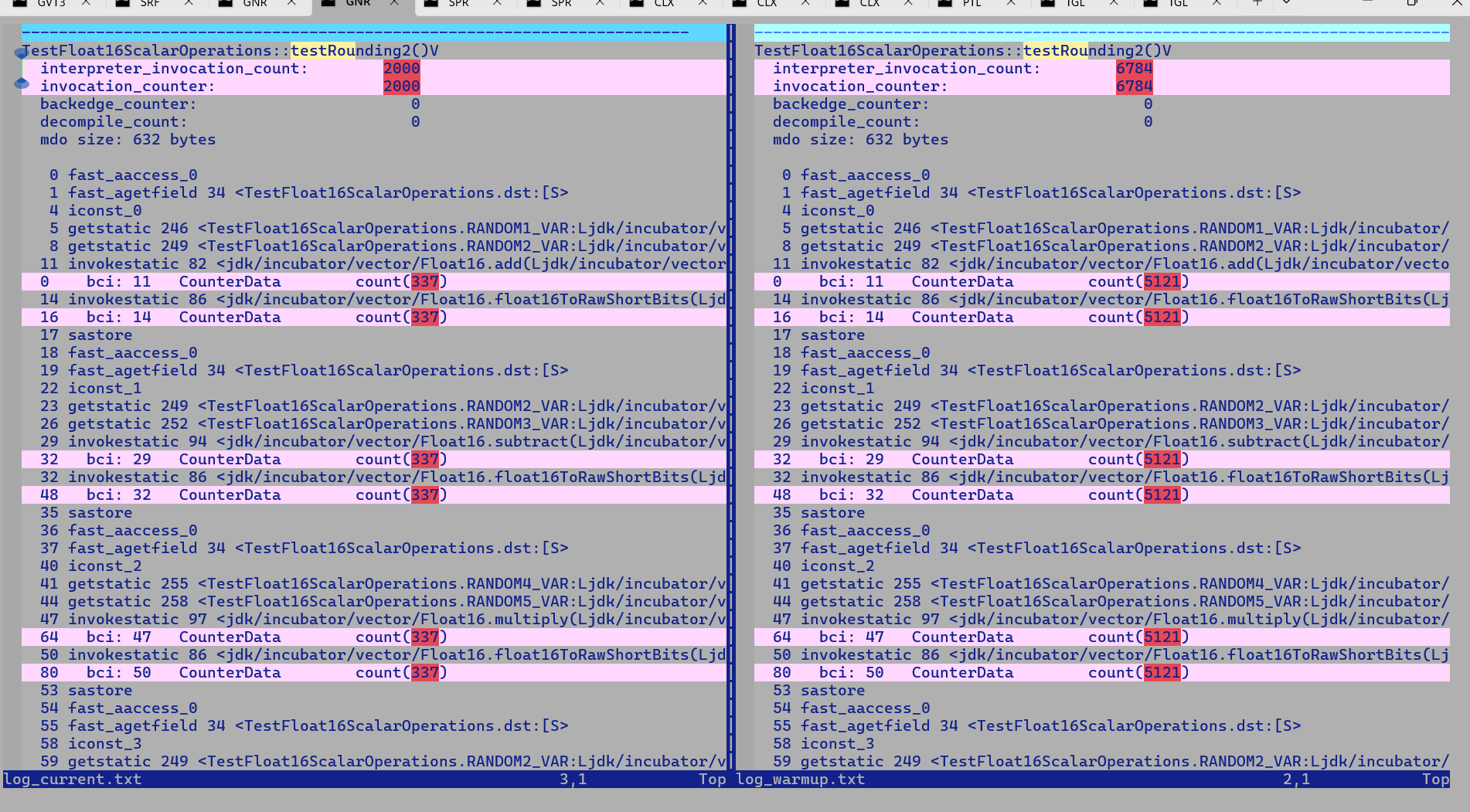
**Bug Analysis:**

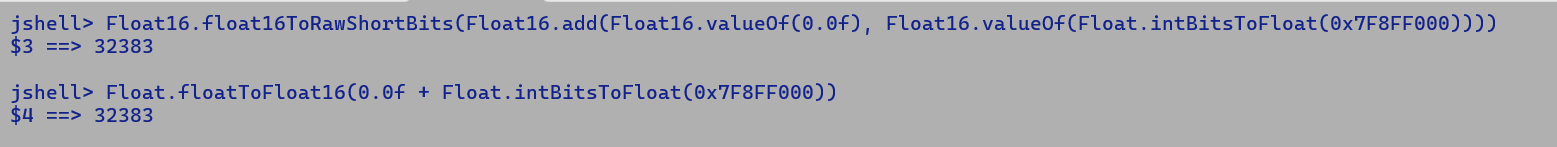
<https://bugs.openjdk.org/browse/JDK-8355708>

MDO dump with -XX:+PrintMethodData

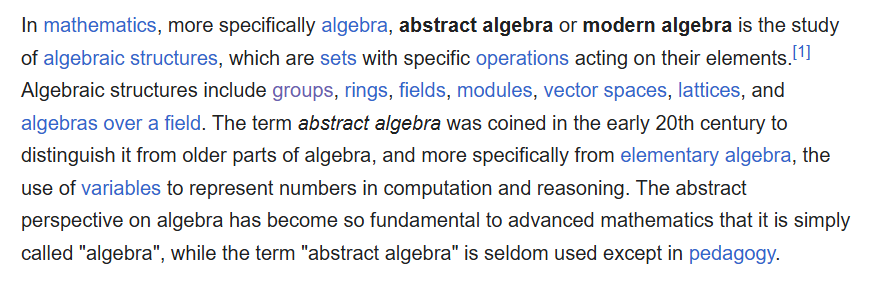
- Insufficient profile information prevents the inlining of Float16.add/multiply/subtract/divide/sqrt/fma

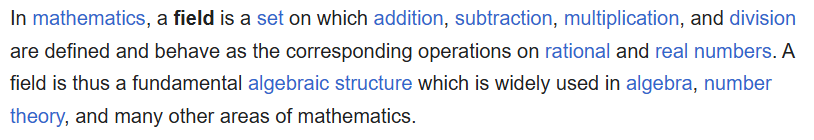
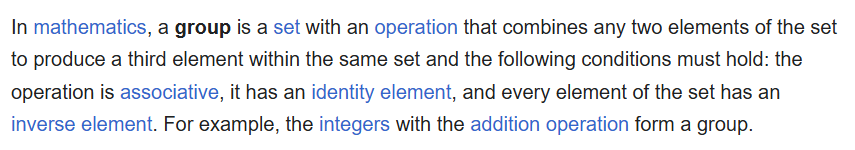
Thereby, C2 emits a Call to these static routines, resulting in non-inference of Float16 IRs

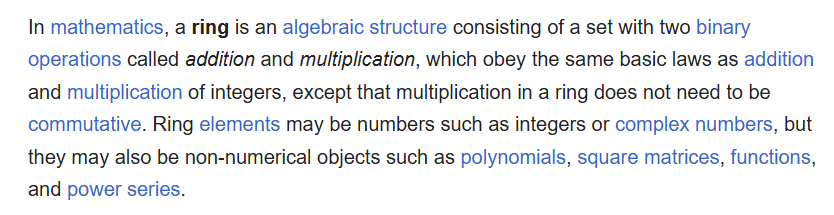


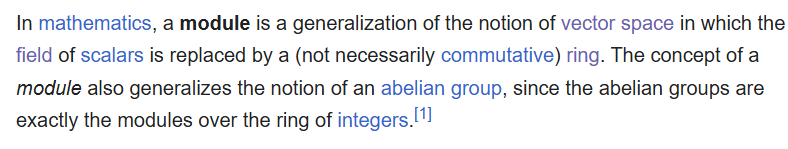


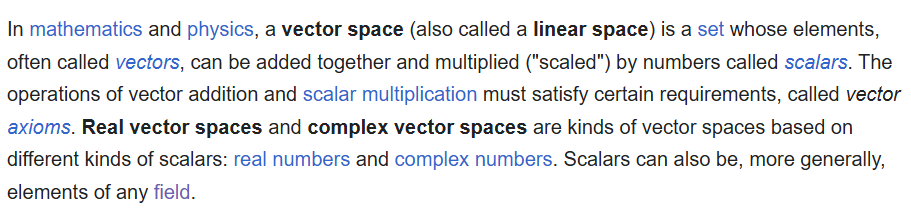
**6/8/2025**

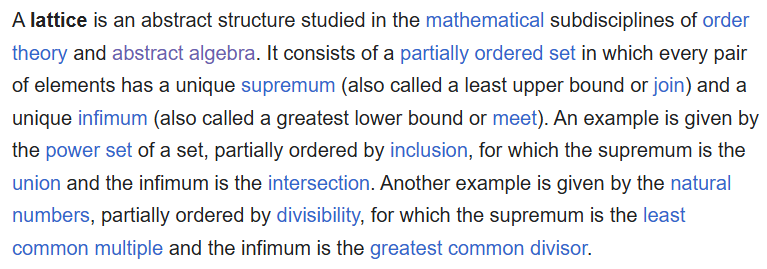


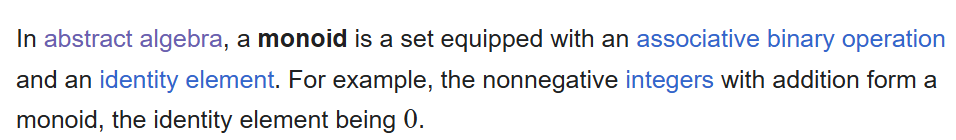












**Float16 Vector API FP16 dotproduct example:-**

static final VectorSpecies<Float16> FSP = Halffloat.SPECIES\_PREFERRED;

short [] fp16buf1 = readFloat16Buffer1();

short [] fp16buf2 = readFloat16Buffer2();

HalffloatVector res = HalffloatVector.broadcast(FSP, 0);

for (int I = 0; I < FSP.loopBound(fp16buf1.length) ; I += FSP.length()) {

HalffloatVector vec1 = HalffloatVector.fromArray(FSP, fp16buf1, idx);

HalffloatVector vec2 = HalffloatVector.fromArray(FSP, fp16buf2, idx);

res = res.lanewise(VectorOperators.FMA, vec1, vec2, res);

}

float dp\_res = 0.0f;

for (int I = 0 ; I < FSP.length(); i++) {

dp\_res += Float.float16ToFloat(res.lane(i));

}

return dp\_res;

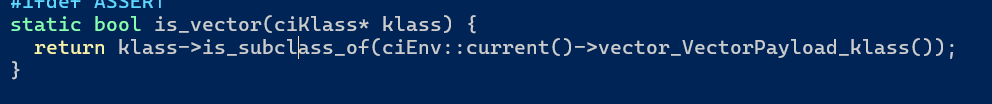
**C2 side changes for Halffloat APIs:-**

1. Vector.fromArray :-

* entry : VectorSupport.load
  + vectortype/boxtype = Halffloat256Vector.class
  + elemtype = short.class
  + length = 16
* expander
  + LoadVector
    - TypeVect : {element\_basic\_type = T\_SHORT, num\_elem = 16}
* Matcher
  + Pattens for short vector loads exists.

1. Vector.lanewise :-

* entry : VectorSupport.unaryOp/binaryOp/ternaryOp
  + vectortype/boxtype = Halffloat256Vector.class
  + elemtype = short.class
  + length = 16
* Use existing vector IR nodes for which backend implementation is already in place. C2 currently creates a specialized IR nodes for FP16 vector operations.



Lanewise operations for both ShortVector and HalffloatVector operate over short lanes i.e. element type passed to expander is short.class and *VectorNode::make* accepts a basic type and scalar opcode to determine a vector IR opcode, we need a special check to infer HalfFloatVector IR opcode. Since HalffloatVector and it’s concrete derivates are part of incubation module, therefore unlike *VectorSupport.\** classes they cannot directly be exposed to VM. For now, we have to resort to name-based check to differentiate b/w Short and HalfFloat vector operations. Alternatively, passing an additional argument to VectorSupport entry points will save costly name-based resolutions.

Auto-Vectorization operates over packs of dependency free scalar IR nodes while Vector API inline expander uses the opcode information to infer vector IR.

Robust solution is to pass an additional int argument which defines operation type while the element type is seen as a carrier type. Various operation types can be VECTOR\_TYPE\_FP16, VECTOR\_TYPE\_FP8, VECTOR\_TYPE\_INT8 while corresponding carrier type is short or byte. Thus, element\_basic\_type of TypeVect will be based on the carrier type while Vector IR opcode inferencing will depend on operation type and operation kind.

1. Vector.compare

* Entry: VectorSupport.compare
  + vectortype/boxtype = Halffloat256Vector.class
  + elemtype/carriertype = short.class
  + length = 16
  + Vector operation type: VECTOR\_TYPE\_FP16
* Expander
  + Based on operation type create a specialized vector IR node VectorMaskCmpHFNode.

1. Vector.intoArray

* Entry: VectorSupport.store
  + vectortype/boxtype = Halffloat256Vector.class
  + elemtype/carriertype = short.class
  + length = 16
  + Vector operation type: VECTOR\_TYPE\_FP16
* Expander
  + No special handling needed on expander side; carrier type is sufficient for generating StoreVectorNode.
  + No special handling needed for predicated variant either.

**Jave Side changes: -**

* Abstract and Concreate vector class nomenclature changes
  + HalffloatVector
    - Halffloat64Vector
    - Halffloat128Vector
    - Halffloat256Vector
    - Halffloat512Vector
  + ElementType
    - short.class
  + BoxType: Float16
    - To simplify the implementation, it looks appealing to comply BoxType with carrier type. i.e. Short.

Q. How will user differentiate if Vector<Short> holds a FP16 or Short value ?

A. Vector<Short> is a receiver of the lanewise API, function dispatch of virtual functions is very much tied to its receiver type, if we pick BoxType as Short, then it will create ambiguity in virtual dispatch, hence we need to pick Float16 as the box type.

* + Fallback implementation of unary, binary and ternary operations should promote input short lanes to floating point values using existing Float16.floatValue() API, perform operation are FP32 granularity and then down case the result to FP16 value.