# AVX-512 Opmask register allocation and optimization for masked operations

Jatin Bhateja

Intel

## Agenda

- Introduction to AVX-512 opmask registers.
- C2 register allocation.
- Opmask register allocation support (PoC).
- Improvements in existing masked operation support (PoC).
- Remaining items.

# AVX-512 opmask registers:

- AVX-512 added support for 8 new opmask registers (k0 through k7) on both 32 and 64-bit platforms.
- These registers are used for conditional execution and efficient merging of destination operands.
- The opmask register state is managed by the operating system using the XSAVE/XRSTOR/XSAVEOPT instructions.
- The width of each opmask register is architecturally defined as size MAX_KL (64 bits). Seven of the eight opmask registers (k1-k7) can be used in conjunction with EVEX-encoded AVX-512 Foundation instructions.
- The encoding of opmask register k0 is typically used when all data elements (unconditional processing) are desired.

- AVX-512 vectors composed of multiple lanes or elements of various sizes (8bit/16bits/32bits/64bits).
- Each bit of an opmask register corresponds to the lane/element in a vector register. For a 8bit lane we use all 64 mask bits, for 16bit lane 32 mask bits are used, for 32bit lane mask comprises of 16bits.

## C2 Register allocator (quick recap):

- Based on Chaitin Briggs' algorithm graph coloring algorithm.
- Uses register masks for constraining allocation set of a def and while creating interferences b/w LRGs during IFG construction.
- Initial step computes set of livein and liveout registers for each block and create a degenerated graph with node count same as number of LRGs.
- Register mask overlap aware interferences edges are created during iterative backward dataflow liveness analysis over blocks.
- INTPRESSURE and FLOATPRESSURE threshold and block register pressures are used while making spilling decisions.

# Opmask register allocation support (PoC).

- AD file changes:
  1. New register definitions for K0-K7 registers.
  2. New register classes for opmask registers.
  3. New operand definition kReg for mask operands, this is later used in instruction patterns.
  4. Ideal type binding of opmask operand to type returned by new target dependent API. For X86 this API returns Type::LONG which is sufficient to hold 64bit value of opmask register.
- Other changes:
  1. Extensions to spilling code to move values b/w opmask reg, mem and general-purpose registers appearing in src or dst positions.
  2. State storing and restoration of opmask register during transitions from compiler to VM or during de-optimization and exception handling.
  3. Removal of hard coded uses of K2 register from instruction patterns and macro assembly routines and introduce bound opmask operands in the patterns.

## Improvements in existing masked operation support (PoC).

- RA support for opmask registers and new kReg operands facilitates propagating mask values between instruction patterns.
- AVX-512 facilitates conditional execution of vector operations under the influence of a mask operand.
- Vector API facilitates creation and storage of mask through various APIs declared in VectorMask class e.g. VectorMask.fromArray, VectorMask.fromValues, VectorMask.intoArray etc.
- Currently masks are propagated using vector registers.

# Masked operation support cont..

- Disecting VectorMask.fromArray:
  - VectorMask object encapsulates a boolean array to hold mask values corresponding to vector lanes.
  - As per JVM specification boolean arrays are stored as byte arrays. Thus, for boolean element having a true value LSB bit is set in corresponding byte element.

  - For each VectorMask.fromArray call in java source compiler creates two ideal nodes **`VectorMaskLoad (LoadVector mem).`** First the contents of byte array is loaded into vector followed by unpacking and vector subtract operation, this is needed since blend instruction looks at the MSB bit to select between sources.

  - Similarly, for VectorMask.intoArray() compiler created IR comprise of **`StoreVector (VectorStoreMask vec) mem.`** Mask vector is packed into byte vector which is stored into boolean array.

  - For AVX-512 masked operations, masks are rematerialized into opmask registers K2 from mask vector though a vector comparison. This instruction is emitted before each mask consuming operation. Clearly, we can save these instructions using new mask operands.
  - Alternatively, emitting a **vector blend instruction after an operation** can facilitate masking of operation but its **not energy efficient** since original operation was performed for each lane element before blending it.

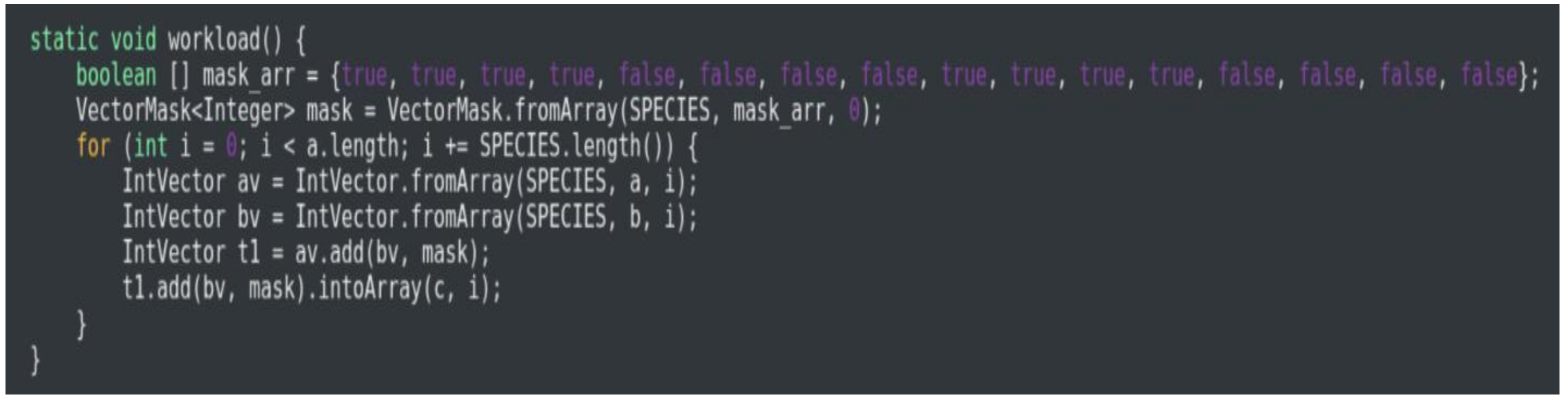
# Masked operation support cont..

- PoC implementation changes:
    1. Main objective was to make minimal modification in target independent IR code.
    2. With respect to masked operation Ideal IR nodes can be divided into three categories:
        A. Mask Generating Nodes : There are 3 ideal nodes which computes mask values i.e. **VectorLoadMask , VectorMaskGen, VectorMaskCmp.** Ideal nodes capture Type and Ideal register information. Mapping b/w Ideal reg and Register mask is created during matcher initializations. Overloaded two routines ***bottom_type*** and ***ideal_reg*** for mask generating nodes to return predicate reg type and new Op_RegVMask ideal registers for AVX-512. Bottom type of Mach Node is currently being used during generic vector operands resolution.

        B. Mask Consuming Nodes : These corresponds to Vector Mask Operations. Added ideal transformation to create **only one new VectorMaskOper Ideal node** which folds various graph patterns involving different kinds of operations i.e. binary (AddV,SubV,MulV,DivV), unary (AbsV) and ternary (Fma).

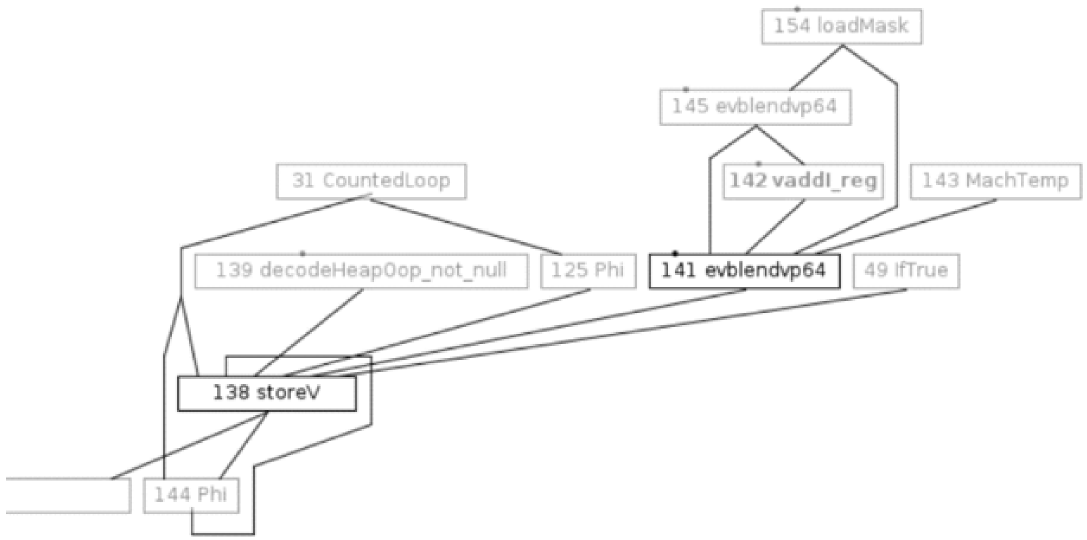
        A. Mask Propagating Nodes: Data convergence nodes (Phi). A scan over IR after instruction selection is done to set special flag in Phi nodes if one of its incoming definition is a Mask operand. This mask is used to return Op_RegVMask ideal register for such Phi nodes.
    3. With a view to introduce minimal number of new instruction selection patterns for masked operation in AD file, mask operation opcode is propagated from Ideal node to newly added MachMetaData node. This opcode information facilitated creation of one generic macro assembly routing which accept opcode.
    4. Added generic type accepting leaf level assembly routines for few vector assembly instructions accepting Kregisters.

## Example:

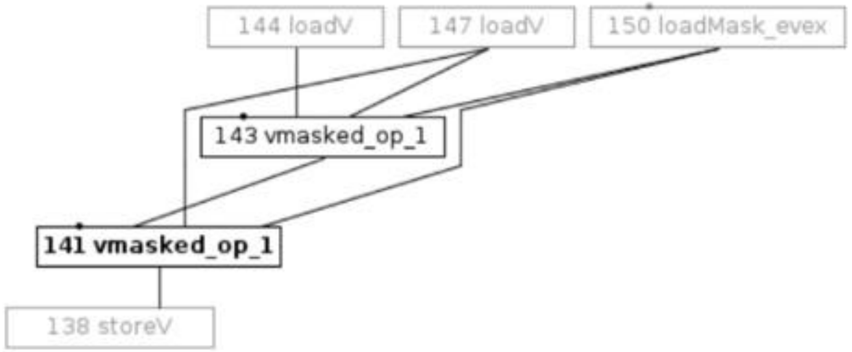


```java
static void workload() {
    boolean [] mask_arr = {true, true, true, true, false, false, false, false, true, true, true, true, false, false, false, false};
    VectorMask<Integer> mask = VectorMask.fromArray(SPECIES, mask_arr, 0);
    for (int i = 0; i < a.length; i += SPECIES.length()) {
        IntVector av = IntVector.fromArray(SPECIES, a, i);
        IntVector bv = IntVector.fromArray(SPECIES, b, i);
        IntVector t1 = av.add(bv, mask);
        t1.add(bv, mask).intoArray(c, i);
    }
}
```

Post matcher original Graph:



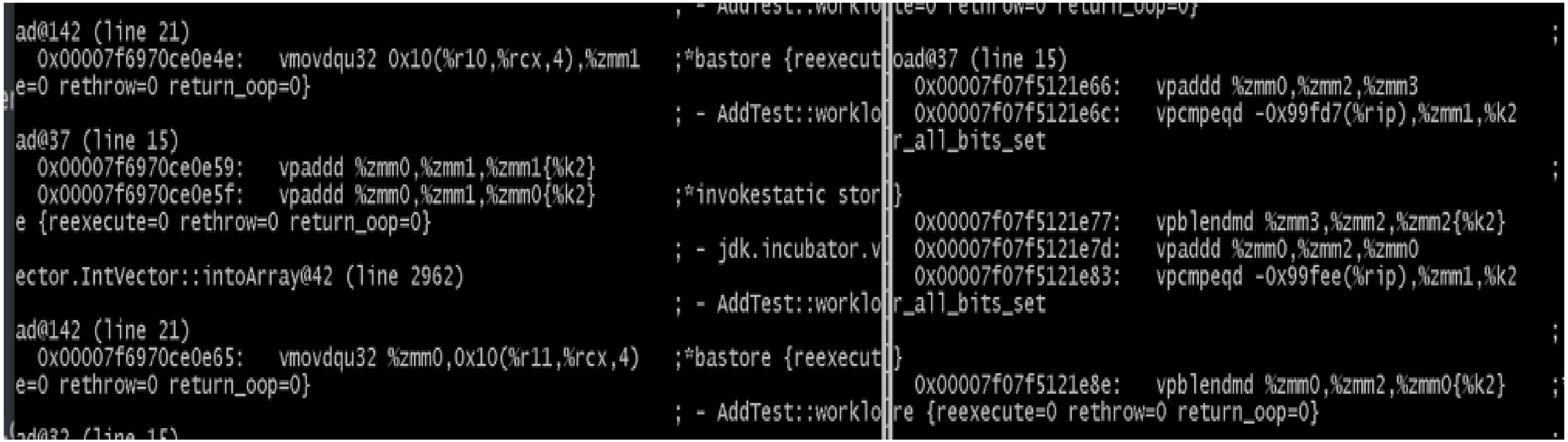
Post matcher new Graph:



JIT Assembly sequence:  Saves extra vector compare and blend per masked operation.

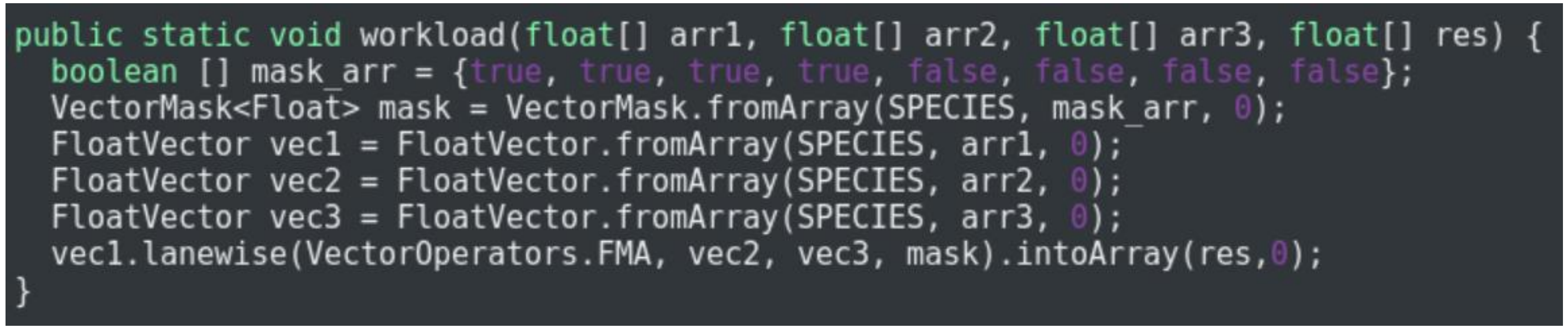
New

Original



```
                                                                 ; - AddTest::worklo
ad@142 (line 21)
  0x00007f6970ce0e4e:   vmovdqu32 0x10(%r10,%rcx,4),%zmm1   ;*bastore {reexecut
e=0 rethrow=0 return_oop=0}
                                                                 ; - AddTest::worklo
ad@37 (line 15)
  0x00007f6970ce0e59:   vpaddd %zmm0,%zmm1,%zmm1{%k2}
  0x00007f6970ce0e5f:   vpaddd %zmm0,%zmm1,%zmm0{%k2}       ;*invokestatic stor
e {reexecute=0 rethrow=0 return_oop=0}
                                                                 ; - jdk.incubator.v
ector.IntVector::intoArray@42 (line 2962)
                                                                 ; - AddTest::worklo
ad@142 (line 21)
  0x00007f6970ce0e65:   vmovdqu32 %zmm0,0x10(%r11,%rcx,4)   ;*bastore {reexecut
e=0 rethrow=0 return_oop=0}
                                                                 ; - AddTest::worklo
ad@37 (line 15)
```

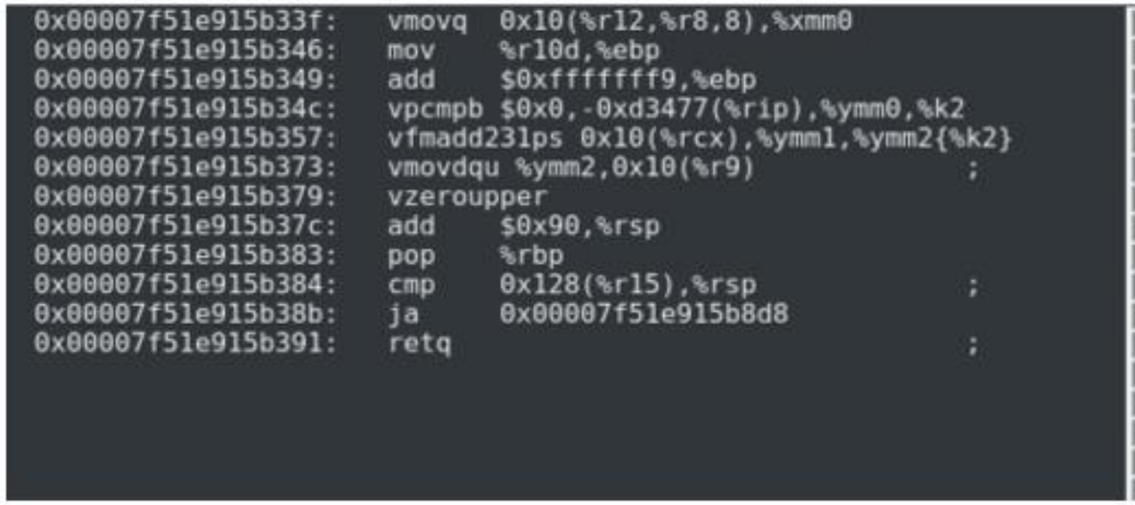
```
te=0 rethrow=0 return_oop=0}
                                                                        ;
oad@37 (line 15)
  0x00007f07f5121e66:   vpaddd %zmm0,%zmm2,%zmm3
  0x00007f07f5121e6c:   vpcmpeqd -0x99fd7(%rip),%zmm1,%k2
r_all_bits_set
                                                                        ;
}
  0x00007f07f5121e77:   vpblendmd %zmm3,%zmm2,%zmm2{%k2}
  0x00007f07f5121e7d:   vpaddd %zmm0,%zmm2,%zmm0
  0x00007f07f5121e83:   vpcmpeqd -0x99fee(%rip),%zmm1,%k2
r_all_bits_set
                                                                        ;
}
  0x00007f07f5121e8e:   vpblendmd %zmm0,%zmm2,%zmm0{%k2}    ;
re {reexecute=0 rethrow=0 return_oop=0}
```



```java
public static void workload(float[] arr1, float[] arr2, float[] arr3, float[] res) {
  boolean [] mask_arr = {true, true, true, true, false, false, false, false};
  VectorMask<Float> mask = VectorMask.fromArray(SPECIES, mask_arr, 0);
  FloatVector vec1 = FloatVector.fromArray(SPECIES, arr1, 0);
  FloatVector vec2 = FloatVector.fromArray(SPECIES, arr2, 0);
  FloatVector vec3 = FloatVector.fromArray(SPECIES, arr3, 0);
  vec1.lanewise(VectorOperators.FMA, vec2, vec3, mask).intoArray(res,0);
}
```

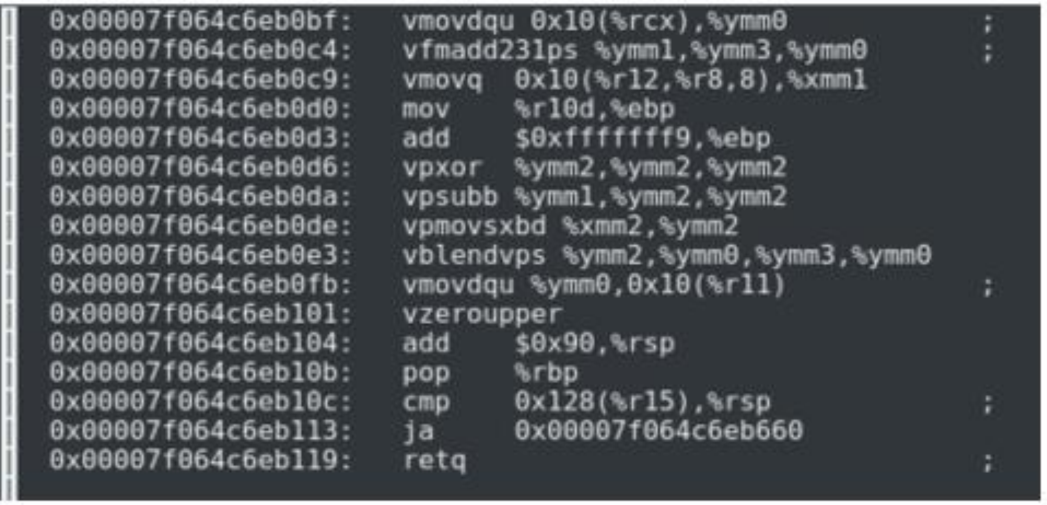
Masking optimization output for Ternary Operation (FMA)

New JIT'ed snippet



```
0x00007f51e915b33f:   vmovq  0x10(%r12,%r8,8),%xmm0
0x00007f51e915b346:   mov    %r10d,%ebp
0x00007f51e915b349:   add    $0xfffffff9,%ebp
0x00007f51e915b34c:   vpcmpb $0x0,-0xd3477(%rip),%ymm0,%k2
0x00007f51e915b357:   vfmadd231ps 0x10(%rcx),%ymm1,%ymm2{%k2}
0x00007f51e915b373:   vmovdqu %ymm2,0x10(%r9)                  ;
0x00007f51e915b379:   vzeroupper
0x00007f51e915b37c:   add    $0x90,%rsp
0x00007f51e915b383:   pop    %rbp
0x00007f51e915b384:   cmp    0x128(%r15),%rsp                  ;
0x00007f51e915b38b:   ja     0x00007f51e915b8d8
0x00007f51e915b391:   retq                                     ;
```

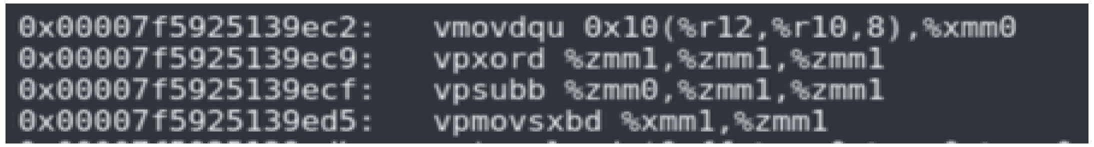
Original JIT'ed snippet



```
0x00007f064c6eb0bf:   vmovdqu 0x10(%rcx),%ymm0                           ;
0x00007f064c6eb0c4:   vfmadd231ps %ymm1,%ymm3,%ymm0                      ;
0x00007f064c6eb0c9:   vmovq  0x10(%r12,%r8,8),%xmm1
0x00007f064c6eb0d0:   mov    %r10d,%ebp
0x00007f064c6eb0d3:   add    $0xfffffff9,%ebp
0x00007f064c6eb0d6:   vpxor  %ymm2,%ymm2,%ymm2
0x00007f064c6eb0da:   vpsubb %ymm1,%ymm2,%ymm2
0x00007f064c6eb0de:   vpmovsxbd %xmm2,%ymm2
0x00007f064c6eb0e3:   vblendvps %ymm2,%ymm0,%ymm3,%ymm0
0x00007f064c6eb0fb:   vmovdqu %ymm0,0x10(%r11)                           ;
0x00007f064c6eb101:   vzeroupper
0x00007f064c6eb104:   add    $0x90,%rsp
0x00007f064c6eb10b:   pop    %rbp
0x00007f064c6eb10c:   cmp    0x128(%r15),%rsp                            ;
0x00007f064c6eb113:   ja     0x00007f064c6eb660
0x00007f064c6eb119:   retq                                               ;
```

**Mask loading API VectorMask.fromArray() instruction sequence latency and size:**

**Original:**



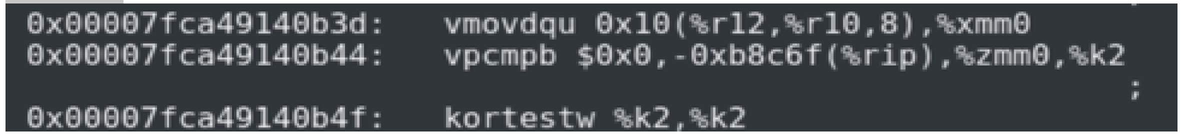
```
0x00007f5925139ec2:   vmovdqu 0x10(%r12,%r10,8),%xmm0
0x00007f5925139ec9:   vpxord %zmm1,%zmm1,%zmm1
0x00007f5925139ecf:   vpsubb %zmm0,%zmm1,%zmm1
0x00007f5925139ed5:   vpmovsxbd %xmm1,%zmm1
```

| | |
|---|---|
| Ec9 -> VPXOR | Zeroing idiom. (not issued to execution ports). |
| Ecf -> VPSUB | 1 cycle |
| Ed5 -> VPMOVSXBD | 3 cycle one port. |

Latency : 4 cycles

Size : 18 bytes

**New:**



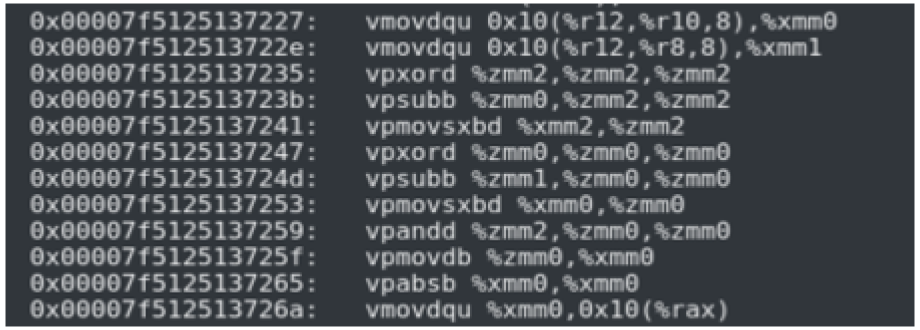
```
0x00007fca49140b3d:   vmovdqu 0x10(%r12,%r10,8),%xmm0
0x00007fca49140b44:   vpcmpb $0x0,-0xb8c6f(%rip),%zmm0,%k2
                                                       ;
0x00007fca49140b4f:   kortestw %k2,%k2
```

| | |
|---|---|
| B44 -> VPCMP | 3 cycles one port. |

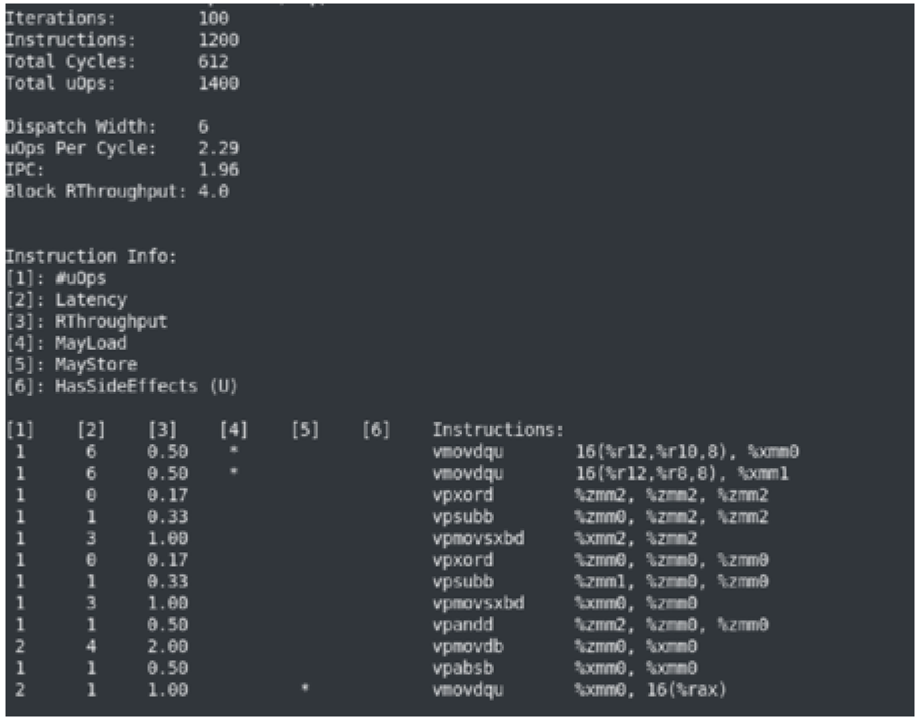
Latency : 3 cycles

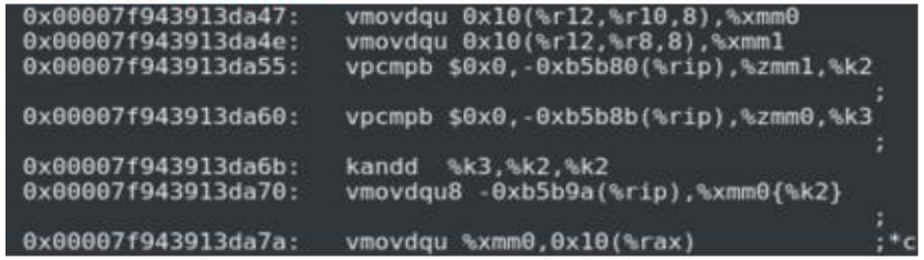
Size : 11 bytes

***VectorMask jdk.incubator.vector.VectorMask.and(VectorMask , VectorMask)***

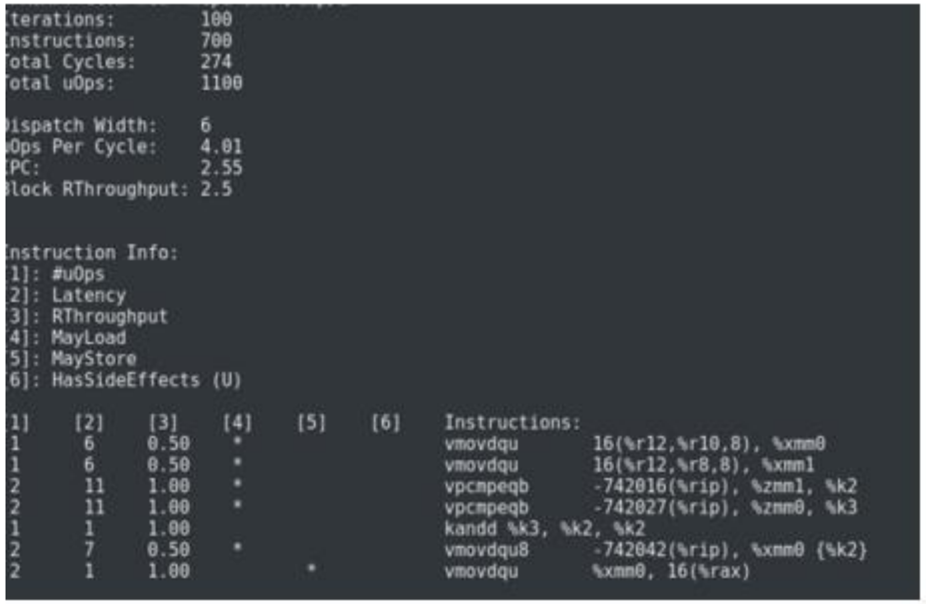

```
0x00007f5125137227:   vmovdqu 0x10(%r12,%r10,8),%xmm0
0x00007f512513722e:   vmovdqu 0x10(%r12,%r8,8),%xmm1
0x00007f5125137235:   vpxord %zmm2,%zmm2,%zmm2
0x00007f512513723b:   vpsubb %zmm0,%zmm2,%zmm2
0x00007f5125137241:   vpmovsxbd %xmm2,%zmm2
0x00007f5125137247:   vpxord %zmm0,%zmm0,%zmm0
0x00007f512513724d:   vpsubb %zmm1,%zmm0,%zmm0
0x00007f5125137253:   vpmovsxbd %xmm0,%zmm0
0x00007f5125137259:   vpandd %zmm2,%zmm0,%zmm0
0x00007f512513725f:   vpmovdb %zmm0,%xmm0
0x00007f5125137265:   vpabsb %xmm0,%xmm0
0x00007f512513726a:   vmovdqu %xmm0,0x10(%rax)
```

Micro-arch analysis (SKX)


```
Iterations:        100
Instructions:      1200
Total Cycles:      612
Total uOps:        1400

Dispatch Width:    6
uOps Per Cycle:    2.29
IPC:               1.96
Block RThroughput: 4.0


Instruction Info:
[1]: #uOps
[2]: Latency
[3]: RThroughput
[4]: MayLoad
[5]: MayStore
[6]: HasSideEffects (U)

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 1      6     0.50    *                   vmovdqu      16(%r12,%r10,8), %xmm0
 1      6     0.50    *                   vmovdqu      16(%r12,%r8,8), %xmm1
 1      0     0.17                        vpxord       %zmm2, %zmm2, %zmm2
 1      1     0.33                        vpsubb       %zmm0, %zmm2, %zmm2
 1      3     1.00                        vpmovsxbd    %xmm2, %zmm2
 1      0     0.17                        vpxord       %zmm0, %zmm0, %zmm0
 1      1     0.33                        vpsubb       %zmm1, %zmm0, %zmm0
 1      3     1.00                        vpmovsxbd    %xmm0, %zmm0
 1      1     0.50                        vpandd       %zmm2, %zmm0, %zmm0
 2      4     2.00                        vpmovdb      %zmm0, %xmm0
 1      1     0.50                        vpabsb       %xmm0, %xmm0
 2      1     1.00           *            vmovdqu      %xmm0, 16(%rax)
```


```
0x00007f943913da47:   vmovdqu 0x10(%r12,%r10,8),%xmm0
0x00007f943913da4e:   vmovdqu 0x10(%r12,%r8,8),%xmm1
0x00007f943913da55:   vpcmpb $0x0,-0xb5b80(%rip),%zmm1,%k2
                                                           ;
0x00007f943913da60:   vpcmpb $0x0,-0xb5b8b(%rip),%zmm0,%k3
                                                           ;
0x00007f943913da6b:   kandd  %k3,%k2,%k2
0x00007f943913da70:   vmovdqu8 -0xb5b9a(%rip),%xmm0{%k2}
                                                           ;
0x00007f943913da7a:   vmovdqu %xmm0,0x10(%rax)             ;*c
```

Micro-arch analysis (SKX)


```
terations:        100
nstructions:      700
otal Cycles:      274
otal uOps:        1100

ispatch Width:    6
Ops Per Cycle:    4.01
PC:               2.55
lock RThroughput: 2.5


nstruction Info:
1]: #uOps
2]: Latency
3]: RThroughput
4]: MayLoad
5]: MayStore
6]: HasSideEffects (U)

1]    [2]    [3]    [4]    [5]    [6]    Instructions:
1      6     0.50    *                   vmovdqu      16(%r12,%r10,8), %xmm0
1      6     0.50    *                   vmovdqu      16(%r12,%r8,8), %xmm1
2      11    1.00    *                   vpcmpeqb     -742016(%rip), %zmm1, %k2
2      11    1.00    *                   vpcmpeqb     -742027(%rip), %zmm0, %k3
1      1     1.00                        kandd %k3, %k2, %k2
2      7     0.50    *                   vmovdqu8     -742042(%rip), %xmm0 {%k2}
2      1     1.00           *            vmovdqu      %xmm0, 16(%rax)
```

## Remaining Items:

- Split the patch in two parts for easy review
  1. RA Opmask support + replace existing hardcoded K2/K1/K7 instantiations in macro assembly routines with bound opmask operands in corresponding instruction patterns.
  2. Optimizing existing vector masked operations handling for AVX-512, also VectorAPI's VectorMask class defines various operation on masks, allTrue/anyTrue/firstTrue/lastTrue/and/not/or etc which can now be optimized to generate efficient JIT sequence.
- SLP does not infer masked patterns, vector blend patterns (strech goal).

## Backup

## C2 Register allocation:

- Compiler uses Chaitin’s graph coloring algorithm for register allocation.
- Allocation works over Machine IR(SSA), this is tightly coupled with the target architecture.
- Each machine instruction has following attributes:
    1. Machine Opcode : This opcode propagated from the IDEAL IR.
    2. Machine Instruction Encoding is generated through associated emit routines which are generated by ADLC.
    3. Machine Operands: Operand can be a DEF, USE or a TEMP.  Operands facilitate data flow b/w instructions i.e. DEF operand of an instruction feeds the USE operands of its user instruction.

## C2 Register allocation cont..

- Machine Operand carry a register mask, which determines the physical register allocation set for that operand.
- Register mask is a bit vector generated by ADCL after parsing reg_class clause in AD files.
- Each bit of register mask corresponds to a 32-bit value. Since as per JVM specification local variable/operand size is 4 bytes. Thus, for a 64 bit register like RAX,RDX two bits are set in register mask.
- Fundamental unit of allocation in Chaitin's textbook algorithm is a web.
- For non-SSA IR a web comprises of a USE with all its definitions. For an SSA IR each use has only one associated DEF. For data convergence nodes (Phi) web comprises of DEF of Phi along with DEFs of all the incoming DEF operands from different control paths.
- Algorithm begins by computing the LRGs (live ranges) for each DEF node (web). RA maintains a map b/w LRG and Machine Node Index.

# C2 register allocation cont..

- Once live ranges are computed next stage is to create an IFG (interference graph). C2 internally uses adjacency matrix representation to store it.
- Edge exists b/w two LRGs if they overlap.
- On completion of IFG construction, physical registers are allocated (coloring phase) to each LRG. Allocator checks the physical registers allocated to neighboring nodes, subtracts them from the register mask associated current LRG and assigns a physical register from the remaining available registers.
- In case number of intersecting LRGs are greater than number of physical registers, allocator SPILLs certain high degree (number of adjacent LRGs) LRG. Spilling causes reserving a slot on stack and creating MachSpillCopyNodes which emits  relevant instructions to move the contents of registers into memory at DFE site and loads contents from stack into register at USE sites.
- RA also does special handling for two address instruction, CICS promotions and aggressive coalescing to reduce number of allocatable LRGs.