AVX-512 MASK REGISTERS CODE GENERATION CHALLENGES

Guy Blank

Intel Corporation, Israel

March 27-28, 2017 European LLVM Developers Meeting

Saarland Informatics Campus, Saarbrücken, Germany

Motivation

C

```
extern void f();
extern int j;
void foo (bool b) {
  if (j && b )
    f();
}
```

AVX2

```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        xorb        $1, %dil
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

AVX512

```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        kmovw       %edi, %k0
        kxnorw      %k0, %k0, %k1
        kshiftrw    $15, %k1, %k1
        kxorw       %k1, %k0, %k0
        kmovw       %k0, %eax
        andl        $1, %eax
        testb       %al, %al
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

- New instructions utilized!
- Scalar performance worse than AVX2
- Why are mask registers used is scalar code?

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Outline

- Introduction
- Scalar Code Issues
- Memory Representation

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Introduction

- Intel Advanced Vector Extension 512 (AVX-512) is an extension to AVX and AVX2
- Introduces 32 64-byte wide SIMD registers (zmm0-31)
  - “old” xmm and ymm registers are aliased to the lower part of zmms
- Introduces 8 Mask registers (k0-7)
- Mask registers’ width is architecturally defined, up to 64 bits
  - Each bit controls the operation on a single element of the vector register
- Mask registers provide conditional execution and efficient merging of data elements

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Masked Operations

- An operation is not performed for an element if the corresponding mask bit is not set
  - No exceptions can be caused by a masked-off element
- A destination element is not updated if the corresponding mask bit is not set
  - The element value is either preserved or zeroed

```
vpaddb    %zmm1, %zmm2, %zmm0{%k1}
```

- Packed byte operation, 64 mask register bits are used, masked-off elements are preserved

```
vpaddq    %zmm1, %zmm2, %zmm0{%k1}{z}
```

- Packed quadword operation, only 8 bits from the mask register are used, masked-off elements are zeroed

```
vaddss    %xmm1, %xmm2, %xmm0{%k1}
```

- Scalar operation, only 1 bit from the mask register are used, masked-off elements are preserved

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Mask Registers

How are mask registers born?

- Vector compare

```
vpcmpeqb  %zmm1, %zmm0, %k0
```

- Scalar Floating-Point compare

```
vcmpeqss  %xmm1, %xmm0, %k0
```

- Copy from GPR / Load from memory

```
kmovw   %edi, %k1             kmovw   (%rdi), %k1
```

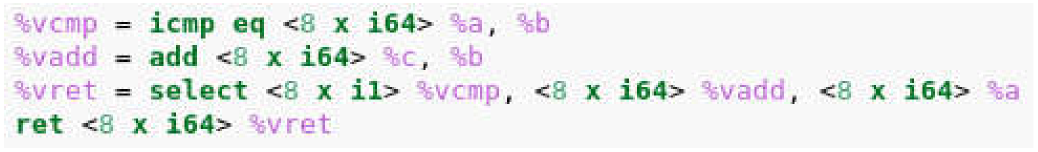
- Mask-to-mask operations

```
kandw     %k1, %k0, %k2       korw      %k1, %k0, %k2
```

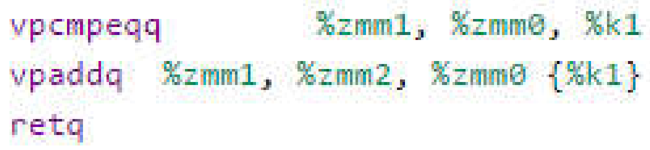
Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Masks in LLVM IR

- No special representation in IR
- Naturally map to <N x i1> data types
  - As the result of vector compares
  - As the condition operand of vector selects



```
%vcmp = icmp eq <8 x i64> %a, %b
%vadd = add <8 x i64> %c, %b
%vret = select <8 x i1> %vcmp, <8 x i64> %vadd, <8 x i64> %a
ret <8 x i64> %vret
```



```
vpcmpeqq        %zmm1, %zmm0, %k1
vpaddq  %zmm1, %zmm2, %zmm0 {%k1}
retq
```

- X86 C intrinsics use scalar integer types for masks
  - Bitcasted to i1 vector types in IR/DAG

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Masks in the X86 Backend

Prior to AVX512

- <N x i1> types are illegal in the X86 Backend
  - Promoted to fit into XMM registers
- i1 type is illegal
  - Promoted to i8, mapped to a GPR class

With AVX512

- X86 Backend declares <N x i1> types legal
  - Mapping them to registers classes containing mask registers
- X86 Backend declares i1 type legal
  - Mapped to mask registers as well
  - Supporting scalar masked operations
  - Supporting <N x i1> related DAG nodes: build vector, extract vector element, …

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

AVX-512 Scalar Code

C

```
extern void f();
extern int j;
void foo (bool b) {
  if (j && b )
    f();
}
```

AVX2

```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        xorb        $1, %dil
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

AVX512

```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        kmovw       %edi, %k0
        kxnorw      %k0, %k0, %k1
        kshiftrw    $15, %k1, %k1
        kxorw       %k1, %k0, %k0
        kmovw       %k0, %eax
        andl        $1, %eax
        testb       %al, %al
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

C bool condition is computed using mask register instructions

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

AVX-512 Scalar Code

LLVM IR

```
define void @_Z3foob(i1 zeroext %b) #0 {
entry:
  %0 = load i32, i32* @j, align 4, !tbaa !1
  %tobool = icmp eq i32 %0, 0
  %b.not = xor i1 %b, true
  %brmerge = or i1 %tobool, %b.not
  br i1 %brmerge, label %if.end, label %if.then

if.then:
  tail call void @_Z1fv()
  br label %if.end

if.end:
  ret void
}
```

AVX2

```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        xorb        $1, %dil
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

AVX512

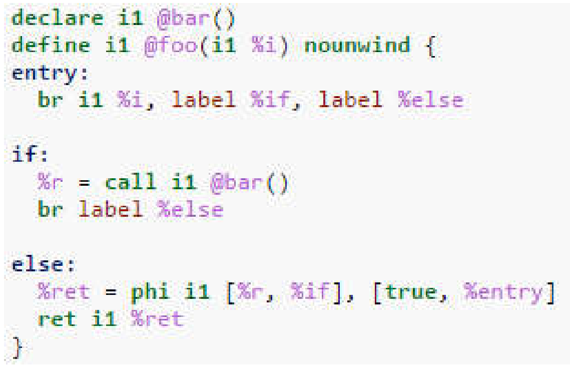
```
_Z3foob:
        .cfi_startproc
# BB#0:
        cmpl        $0, j(%rip)
        je          .LBB0_2
# BB#1:
        kmovw       %edi, %k0
        kxnorw      %k0, %k0, %k1
        kshiftrw    $15, %k1, %k1
        kxorw       %k1, %k0, %k0
        kmovw       %k0, %eax
        andl        $1, %eax
        testb       %al, %al
        jne         .LBB0_2
# BB#3:
        jmp         _Z1fv
.LBB0_2:
        retq
```

- AVX2 – i1 is illegal, promoted to i8 and assigned to a GPR
- AVX512 – i1 is legal, assigned to a Mask register
  - The i1 data type has different use cases
  - scalar integer vs. scalar mask
  - Each use case has a different appropriate register class
- Isn't this an instruction selection bug? Yes, But…

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

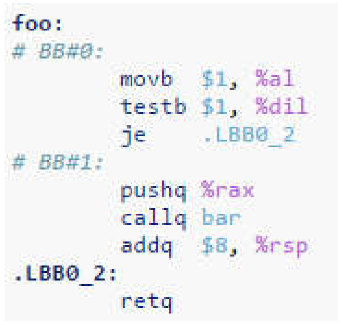
Cross Basic Block Code

LLVM IR



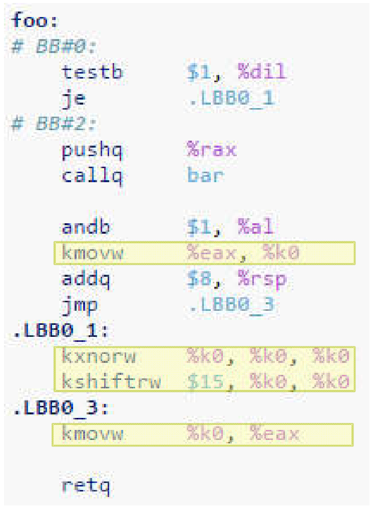
```
declare i1 @bar()
define i1 @foo(i1 %i) nounwind {
entry:
  br i1 %i, label %if, label %else

if:
  %r = call i1 @bar()
  br label %else

else:
  %ret = phi i1 [%r, %if], [true, %entry]
  ret i1 %ret
}
```

AVX2



```
foo:
# BB#0:
        movb  $1, %al
        testb $1, %dil
        je    .LBB0_2
# BB#1:
        pushq %rax
        callq bar
        addq  $8, %rsp
.LBB0_2:
        retq
```

AVX512



```
foo:
# BB#0:
    testb      $1, %dil
    je         .LBB0_1
# BB#2:
    pushq      %rax
    callq      bar

    andb       $1, %al
    kmovw      %eax, %k0
    addq       $8, %rsp
    jmp        .LBB0_3
.LBB0_1:
    kxnorw     %k0, %k0, %k0
    kshiftrw   $15, %k0, %k0
.LBB0_3:
    kmovw      %k0, %eax

    retq
```

Cross-BB

- Instruction Selection does not look beyond the scope of a basic block
- Default register class is used for live in/out values – Mask registers are selected
- With GlobalISel, there should be enough information to make the right choice

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Solution A: Implement a Fixup pass

- Post instruction selection machine function pass
- Replace mask-based instructions with GPR-based ones, when profitable

LLVM IR

```
declare i1 @bar()
define i1 @foo(i1 %i) nounwind {
entry:
  br i1 %i, label %if, label %else

if:
  %r = call i1 @bar()
  br label %else

else:
  %ret = phi i1 [%r, %if], [true, %entry]
  ret i1 %ret
}
```

AVX2

```
foo:
# BB#0:
        movb  $1, %al
        testb $1, %dil
        je    .LBB0_2
# BB#1:
        pushq %rax
        callq bar
        addq  $8, %rsp
.LBB0_2:
        retq
```

AVX512

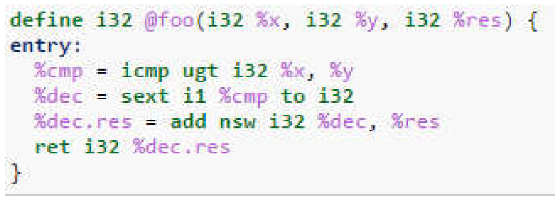
```
foo:
# BB#0:
    testb      $1, %dil
    je         .LBB0_1
# BB#2:
    pushq      %rax
    callq      bar

    andb       $1, %al
    ~~kmovw      %eax, %k0~~
    addq       $8, %rsp
    jmp        .LBB0_3
.LBB0_1:
    kxnorw     %k0, %k0, %k0
    kshiftrw   $15, %k0, %k0     →   movb  $1, %al
.LBB0_3:
    ~~kmovw      %k0, %eax~~

    retq
```

- We could miss out on some optimizations
- Mask-based ISA is limited, resulting in long sequences
  - Could be difficult to replace with optimal GPR-based code

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

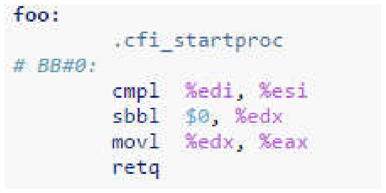
Solution A: Implement a Fixup pass

LLVM IR



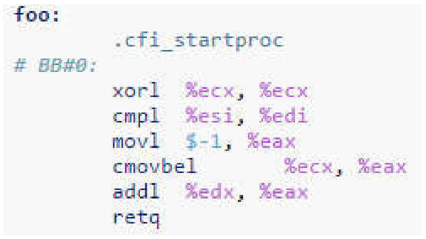
```
define i32 @foo(i32 %x, i32 %y, i32 %res) {
entry:
  %cmp = icmp ugt i32 %x, %y
  %dec = sext i1 %cmp to i32
  %dec.res = add nsw i32 %dec, %res
  ret i32 %dec.res
}
```

AVX2



```
foo:
        .cfi_startproc
# BB#0:
        cmpl  %edi, %esi
        sbbl  $0, %edx
        movl  %edx, %eax
        retq
```

AVX512



```
foo:
        .cfi_startproc
# BB#0:
        xorl  %ecx, %ecx
        cmpl  %esi, %edi
        movl  $-1, %eax
        cmovbel       %ecx, %eax
        addl  %edx, %eax
        retq
```

- No mask registers present, nothing to be fixed by the pass
- The legality of i1 affects optimizations even without mask registers

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Solution B: Choose GPR by default

The core issue is the cross basic block default register class

- Instruction Selection phase does not have all the information to make the best choice

Solution: Change the default register class of i1 to a GPR

- Make i1 illegal in the X86 Backend
- i1 will be promoted to i8, and assigned to GPRs
- Aligns with AVX2
- A solution for scalar masked operation will be needed

- Issues could arise in masked code
  - Fixup pass may still be required

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

i1 Vectors Memory Representation

Memory operations on i1 Vectors

- AVX512 introduces memory load/store operations on mask registers
  - Loading and storing i1 vectors is straightforward

```
%val = load <8 x i1>, <8 x i1>* %src
store <8 x i1> %val, <8 x i1>* %dst, align 1
```

```
kmovb   (%rdi), %k0
kmovb   %k0, (%rsi)
```

  - Memory representation is bit-packed

- In AVX2 i1 vectors are promoted to fit into xmm registers.
  - Bit packing will require an effort.

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

i1 – a bit or a byte?

There were several discussions over the years about the memory representation of i1 vectors. Quite a few bugs are still open

Option 1
Byte packed

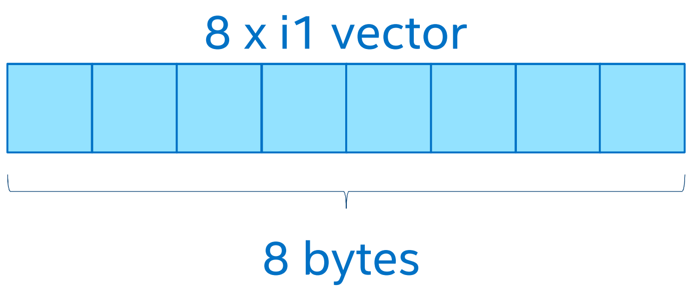
Each vector element stored in a unique byte

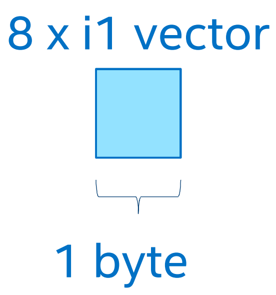
Consecutive vector elements stored in consecutive bytes

Option 2
Bit packed

Each vector element stored in a unique bit

Consecutive vector elements stored in consecutive bits





Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Possible Directions

- Option A
  Byte-packed on all X86 subtargets
  - Not optimal for AVX512
  - Does not align with bitcast semantics
- Option B
  Bit-packed on all X86 subtargets
  - Not optimal for AVX2
- Option C
  Most performant option, per-subtarget
  - Byte-packed on AVX2 and older
  - Bit-packed on AVX512
  - No memory layout consistency within the same target

Optimization Notice
Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

intel®

experience what's inside™