10/11/2023 @ LLVM developers' meeting

APX & AVX10

The next major evolution of Intel® architecture

intel®

AVX10: Advanced Vector Extensions

The converged vector ISA for all Intel CPUs

- Introduces generational umbrella enumeration of all vector ISA
  - Replaces current numerous disjoint vector features of AVX/AVX2/AVX-512.
  - Single CPUID for AVX10 version number and the max supported vector length (VL)
  - All future Intel CPUs will support some version of AVX10, at least AVX10.1/256
- AVX10.1/256: supported on all Intel CPUs (P/E-cores)
  - All modern AVX-512 vector instructions with a maximum VL=256
  - 32 vector register (through EVEX prefix)
  - 8 mask registers (32-bit)
  - New: embedded rounding with 256-bit instructions
- AVX10.1/512: will continue to be supported in all P-cores.
- Inclusive: AVX10.N supports all of AVX10.N-1 plus new features!

AVX10: enabling in SW

This is WIP, details can change

- -m[no-]evex512
  - A proxy option to be able to re-compile current SW and continue running it on the current HW, even not the very latest. It provides guarantees that no 512-bit instructions are generated (even through intrinsics).
- -mavx10.1[-256,-512] (default is 256)
  - Introduced for early SW enablement and supports the subset of AVX10.1:
    - all the Intel AVX512 instruction set available with P-cores codenamed Granite Rapids
    - will not include the new 256-bit vector instructions supporting embedded rounding
- -mavx10.2[-256,-512] (default is 256)
  - Include the new 256-bit vector instructions supporting embedded rounding
  - A suite of new Intel AVX10 instructions covering new AI data types and conversions, data movement optimizations, and standards support

APX: Advanced Performance Extensions

General-purpose extension of 64-bit x86 for all Intel CPUs

- +16 GPRs, for a total of 32 integer EGPRs (extended GPRs) via new REX2 prefix
- NDD: adds unique destination register for legacy GPR instructions
- XSAVE-enabled (overlays deprecated MMX state)
- New instructions/capabilities:
  - PP2: PUSH2/POP2 instructions to bundle couple of EGPR in one instruction
  - FSFP+PPX: Fast Store Forwarding Predictor optimizations in a faster and more stable manner
  - CCMP+CFCMOV: replace more branches with conditional instructions
  - NF: encode suppress of status flag writes of common instruction
  - Zero-upper SETcc: Write full register to reduce extra pre-zeroing instructions and reduce data dependency
  - JMPABS : Replace indirect branches with direct branches (at link time) for better branch prediction, along with benefits in security, and power
- Transparent interaction with legacy x86 code using a legacy-compliant ABI (new EGPRs are all caller-saved/volatile)

EGPR: 32 GPR

Design principle : least intrusive and not affecting legacy

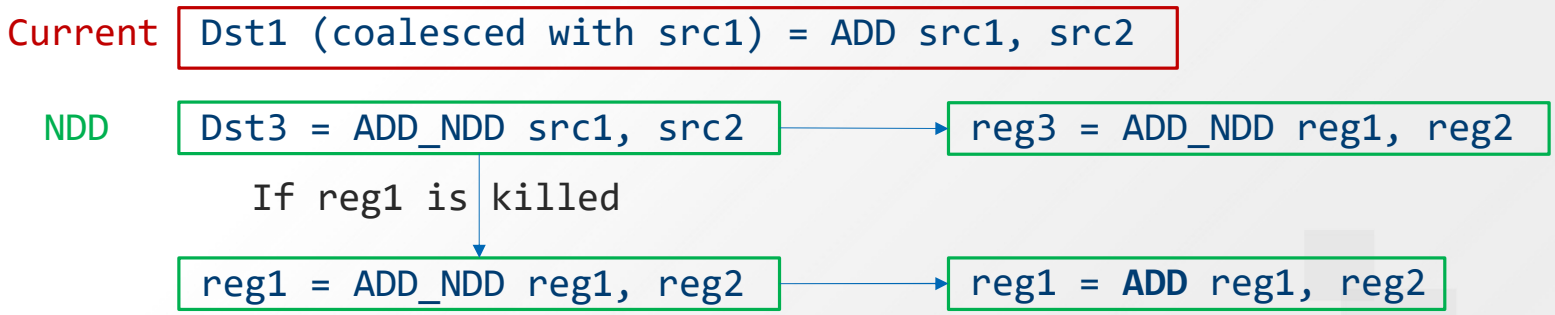
- Value: Eliminate relatively expensive memory operations keeping more state in registers
- Static register class for each instruction in the tablgen file is unchanged to not affect pass whose analysis relies on the static type of operands in TD, e.g. machine instruction schedule.
- Leverage the target hook TargetInstrInfo:getRegClass to update register class before RA
- Reserve R16-R31 for all instructions when GPR32 is not supported (X86RegisterInfo::getReservedRegs)

```
def GR64 : RegisterClass<"X86", [i64], 64, (add RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11,
           R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31,
New class                                    RBX, R14, R15, R12, R13, RBP, RSP, RIP)>;
def GR64_NOREX2 : RegisterClass<"X86", [i64], 64,  (sub GR64,  (sequence "R%u", 16, 31))>;
def A_MAP0_INST: I<..., (outs GR64:$dst), (ins GR64:$src)> Updated to GR64_NOREX2 if no APX
def A_MAP2_INST: I<..., (outs GR64:$dst), (ins GR64:$src)>
```

New destination register (NDD)

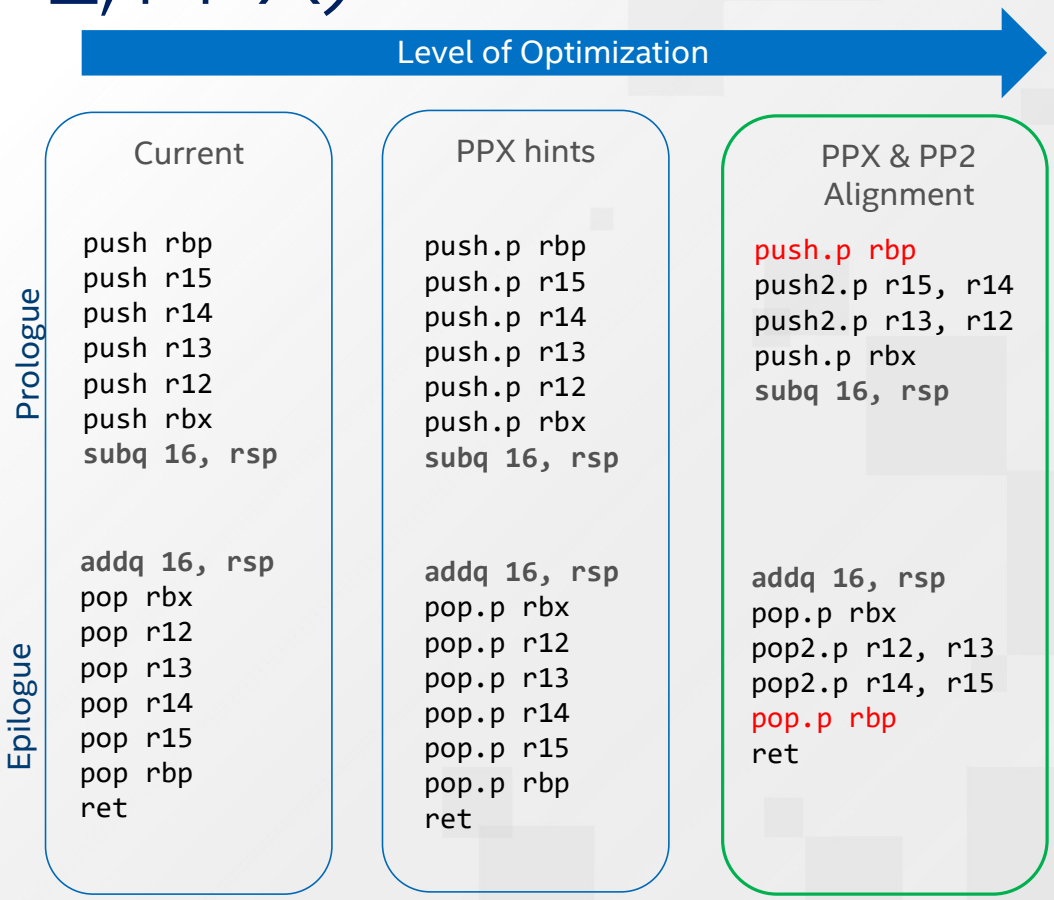
Principle : always prefer NDD over spilling

- Value: Eliminate relatively expensive memory operations keeping more state in registers
- Prefer NDD than non-NDD at instruction selection
- Give a hint to RA to make source and destination are same when it's profitable (e.g. source register is killed)
- Compress the NDD instruction to non-NDD instruction, if possible, for code size



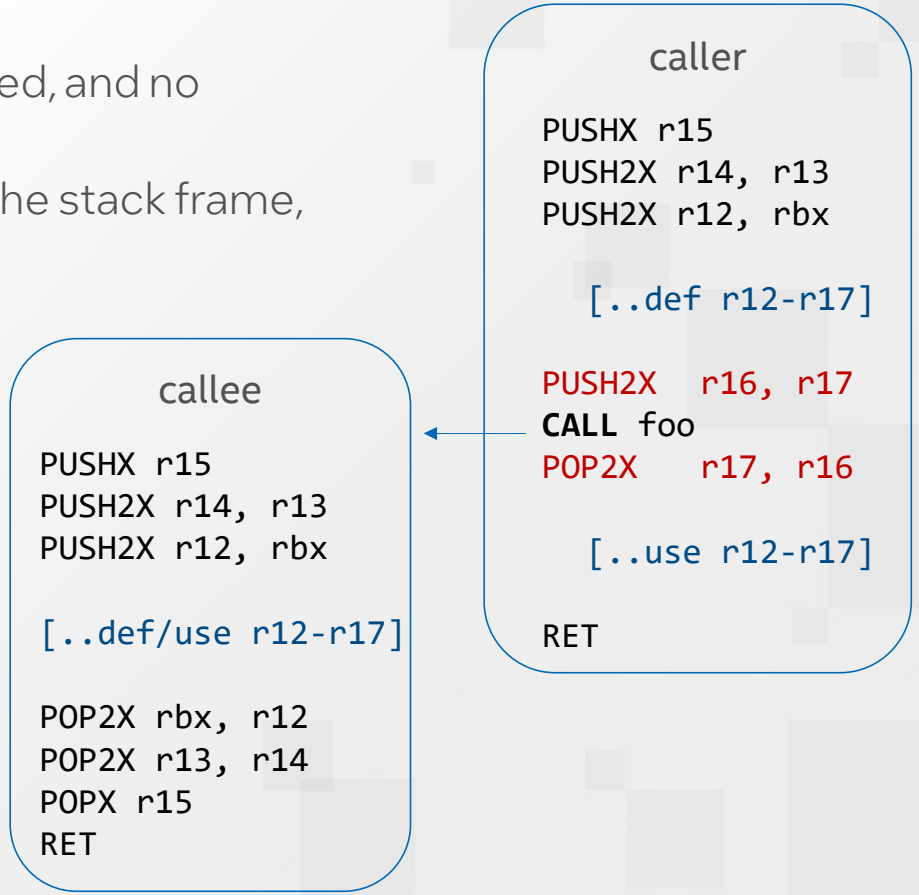
Prologue/epilogue (PP2, PPX)

- Value: reduce number of push/pop memory operations
- PPX applies FSFP optimizations for matched push/pop in a quick and stable manner
- The PUSH2/POP2 require 16B stack alignment (avoids splits in fused operations)
- Red = Pad alignment to maximize PP2 opportunities



Call-site optimization (PP2, PPX)

- Legacy compatible ABI: all new EGPRs are caller-saved, and no changes to parameter passing/returning a value.
- Instead of spilling with MOV to pre-allocated slots in the stack frame, aggressively use PUSH2/POP2 around calls.
- Value: less spill code that is also more efficient due to PPX hints.



Conditional compares (CCMP)

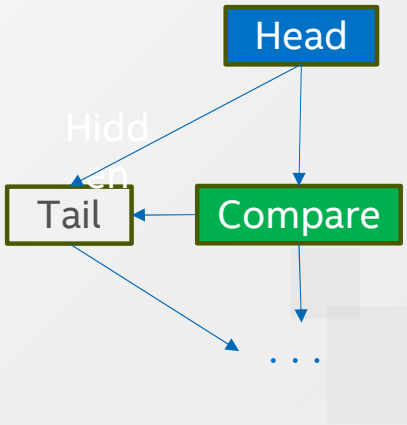
- Example:

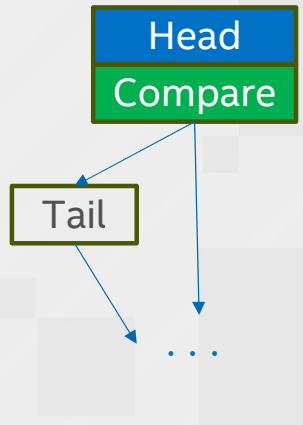
```
if (a == 5 || b == 17)
    foo();
```

```
Head:
  cmpl  $5, $edi
  je Tail
Compare:
  cmpl  $17, $esi
  je Tail
...
Tail:
  call foo
```

```
Head:
cmpl  $5, $edi
ccmpel {zf} $17, $edi
je Tail
...
Tail:
 call foo
```

- Speculatively execute compare operation based on the result of a prior compare
- Value: Eliminate conditional branches to reduce branch mis-prediction
- Update the probabilities of edges:
  - P(Tail|Compare) = P(Tail|Head) + P(Compare|Head) * P(Tail|Compare)
  - P(I|Compare) = P(Compare|Head) * P(I|Compare)





Conditional load/store (CFCMOV)

- Example:

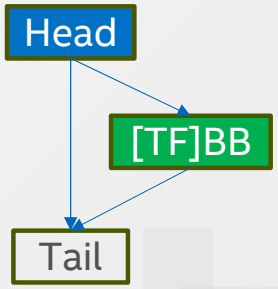
```
// int *p, *q, n;
if (*p > n)
    *p = *q;
```

```
Head:
    cmpl    %edx, (%rdi)
    jle     Tail
TBB:
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
Tail:
```

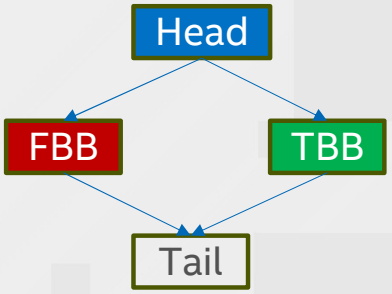
```
Head:
    cmpl    %edx, (%rdi)
    cfcmovgl  (%rsi), %eax
    cfcmovgl  %eax, (%rdi)

Tail:
```

- Load/store instructions in the conditional blocks TBB and/or FBB are spliced into the Head block.
- Increases scope of if-conversion
- Value: Eliminate conditional branches to reduce branch mis-prediction

Triangle:



Diamond:



Summary

- Intel intends to provide/enable the necessary compilers, debuggers, tools, and libraries well in advance of HW to support general APX and AVX10 SW enablement (LLVM, GCC, etc.)
- Whitepapers and further reading:
  - APX: https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html
  - AVX10: https://cdrdv2-public.intel.com/784343/356368-intel-avx10-tech-paper.pdf


intel