New Instruction Set Extensions

Instruction Set Innovation in Intels Processor Code Named Haswell

bob.valentine@intel.com

Agenda

- Introduction - Overview of ISA Extensions
- Haswell New Instructions
  - New Instructions Overview
  - Intel® AVX2 (256-bit Integer Vectors)
  - Gather
  - FMA: Fused Multiply-Add
  - Bit Manipulation Instructions
  - TSX/HLE/RTM
- Tools Support for New Instruction Set Extensions
- Summary/References

Instruction Set Architecture (ISA) Extensions

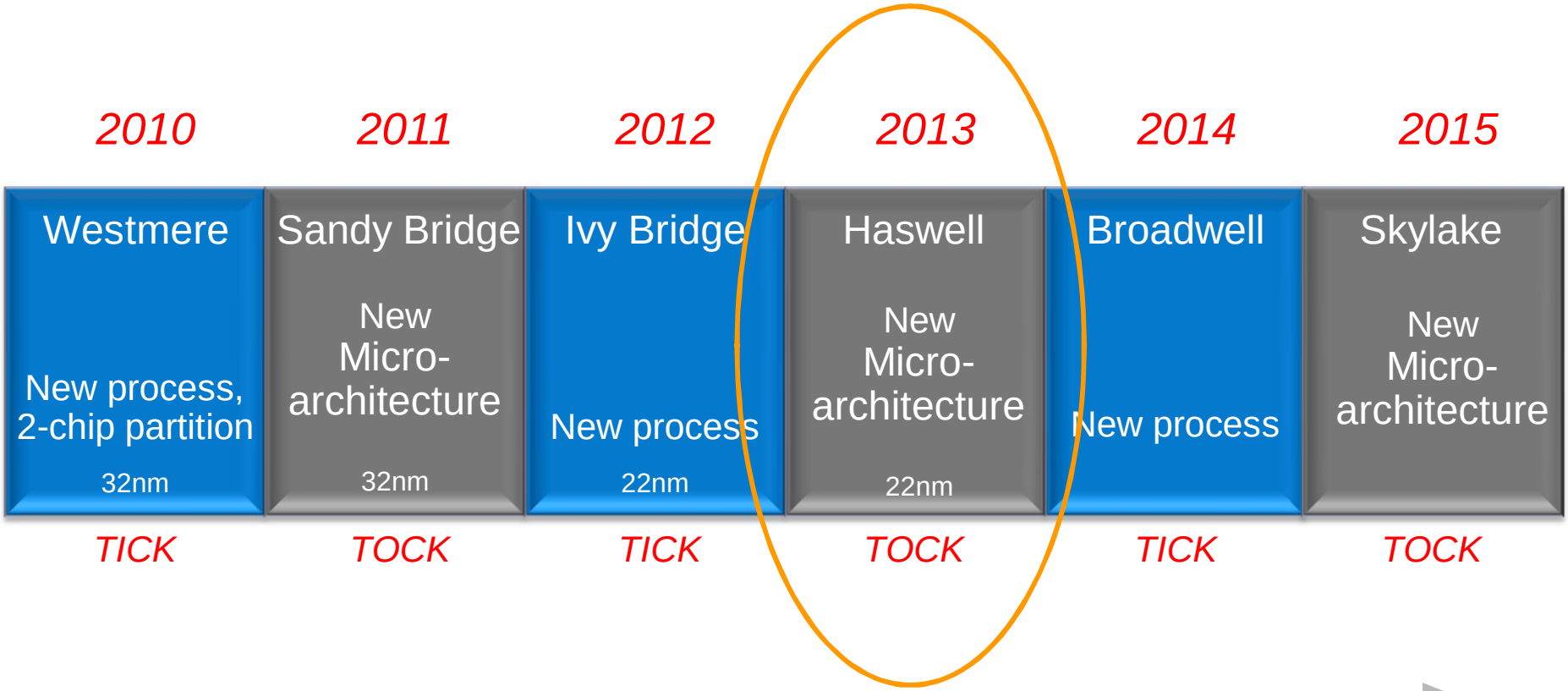
199x	MMX, CMOV, PAUSE, XCHG, …	Multiple new instruction sets added to the initial 32bit instruction set of the Intel® 386 processor
1999	Intel® SSE	70 new instructions for 128-bit single-precision FP support
2001	Intel® SSE2	144 new instructions adding 128-bit integer and double-precision FP support
2004	Intel® SSE3	13 new 128-bit DSP-oriented math instructions and thread synchronization instructions
2006	Intel SSSE3	16 new 128-bit instructions including fixed-point multiply and horizontal instructions
2007	Intel® SSE4.1	47 new instructions improving media, imaging and 3D workloads
2008	Intel® SSE4.2	7 new instructions improving text processing and CRC
2010	Intel® AES-NI	7 new instructions to speedup AES
2011	Intel® AVX	256-bit FP support, non-destructive (3-operand)
2012	Ivy Bridge NI	RNG, 16 Bit FP
2013	Haswell NI	AVX2, TSX, FMA, Gather, Bit NI

A long history of ISA Extensions !

Instruction Set Architecture (ISA) Extensions

- Why new instructions?
  - Higher absolute performance
  - More energy efficient performance
  - New application domains
  - Customer requests
  - Fill gaps left from earlier extensions
- For a historical overview see http://en.wikipedia.org/wiki/X86_instruction_listings

Intel Tick-Tock Model



Tick-tock delivers leadership through technology innovation on a reliable and predictable timeline

New Instructions in Haswell

Group		Description	Count *
AVX2	SIMD Integer Instructions promoted to 256 bits	Adding vector integer operations to 256-bit	170 / 124
	Gather	Load elements from vector of indices vectorization enabler	
	Shuffling / Data Rearrangement	Blend, element shift and permute instructions	
FMA		Fused Multiply-Add operation forms (FMA-3)	96 / 60
Bit Manipulation and Cryptography		Improving performance of bit stream manipulation and decode, large integer arithmetic and hashes	15 / 15
TSX=RTM+HLE		Transactional Memory	4/4
Others		MOVBE: Load and Store of Big Endian forms INVPCID: Invalidate processor context ID	2 / 2

* Total instructions / different mnemonics

Intel® AVX2: 256-bit Integer Vector

Extends Intel® AVX to cover integer operations

Uses same AVX (256-bit) register set

Nearly all 128-bit integer vector instructions are ‘promoted’ to 256

- Including Intel® SSE2, Intel® SSSE3, Intel® SSE4

  Exceptions: GPR moves (MOVD/Q) ; Insert and Extracts <32b, Specials (STTNI instructions, AES, PCLMULQDQ)

New 256b Integer vector operations (not present in Intel® SSE)

- Cross-lane element permutes
- Element variable shifts
- Gather

•Haswell implementation doubles the cache bandwidth

- Two 256-bit loads per cycle, fill rate and split line improvements
- Helps both Intel® AVX2 and legacy Intel® AVX performance

Intel® AVX2 ‘completes’ the 256-bit extensions started with Intel® AVX

Integer Instructions Promoted to 256

VMOVNTDQA	VPHADDSW	VPSUBQ	VPMOVZXDQ	VPMULHW
VPABSB	VPHADDW	VPSUBSB	VPMOVZXWD	VPMULLD
VPABSD	VPHSUBD	VPSUBSW	VPMOVZXWQ	VPMULLW
VPABSW	VPHSUBSW	VPSUBUSB	VPSHUFB	VPMULUDQ
VPADDB	VPHSUBW	VPSUBUSW	VPSHUFD	VPSADBW
VPADDD	VPMAXSB	VPSUBW	VPSHUFHW	VPSLLD
VPADDQ	VPMAXSD	VPACKSSDW	VPSHUFLW	VPSLLDQ
VPADDSB	VPMAXSW	VPACKSSWB	VPUNPCKHBW	VPSLLQ
VPADDSW	VPMAXUB	VPACKUSDW	VPUNPCKHDQ	VPSLLW
VPADDUSB	VPMAXUD	VPACKUSWB	VPUNPCKHQDQ	VPSRAD
VPADDUSW	VPMAXUW	VPALIGNR	VPUNPCKHWD	VPSRAW
VPADDW	VPMINSB	VPBLENDVB	VPUNPCKLBW	VPSRLD
VPAVGB	VPMINSD	VPBLENDW	VPUNPCKLDQ	VPSRLDQ
VPAVGW	VPMINSW	VPMOVSXBD	VPUNPCKLQDQ	VPSRLQ
VPCMPEQB	VPMINUB	VPMOVSXBQ	VPUNPCKLWD	VPSRLW
VPCMPEQD	VPMINUD	VPMOVSXBW	VMPSADBW	VPAND
VPCMPEQQ	VPMINUW	VPMOVSXDQ	VPCMPGTQ	VPANDN
VPCMPEQW	VPSIGNB	VPMOVSXWD	VPMADDUBSW	VPOR
VPCMPGTB	VPSIGND	VPMOVSXWQ	VPMADDWD	VPXOR
VPCMPGTD	VPSIGNW	VPMOVZXBD	VPMULDQ	VPMOVMSKB
VPCMPGTW	VPSUBB	VPMOVZXBQ	VPMULHRSW	
VPHADDD	VPSUBD	VPMOVZXBW	VPMULHUW	

Gather

"Gather" is a fundamental building block for vectorizing indirect memory accesses.

```
int a[] = {1,2,3,4,……,99,100};
int b[] = {0,4,8,12,16,20,24,28}; // indices

for (i=0; i<n; i++) {
    x[i] = a[b[i]];
}
```

Haswell introduces set of GATHER instructions to allow automatic vectorization of similar loops with non-adjacent, indirect memory accesses

- HSW GATHER not always faster – software can make optimizing assumptions

Gather Instruction -Sample

```
VPGATHERDQ    ymm1,[xmm9*8 + eax+22], ymm2
```

P	integer data (no FP data)
D	indicate index size; here double word (32bit)
Q	to indicate data size; here quad word (64 bit) integer
ymm1	destination register
ymm9	index register; here 4-byte size indices
8	scale factor
eax	general register containing base address
22	offset added to base address
ymm2	mask register; only most significant bit of each element used

Gather: Fundamental building block for nonadjacent, indirect memory accesses for either integer or floating point data

Sample – How it Works

Instruction

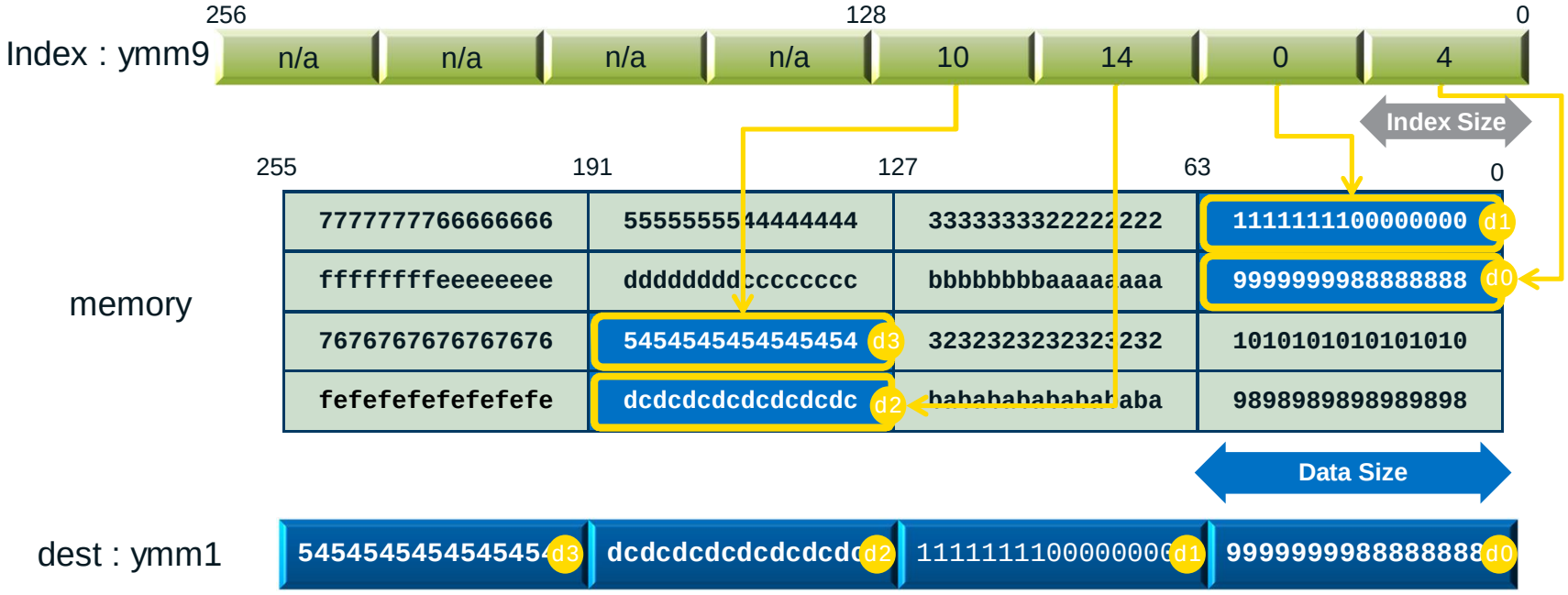
```
VPGATHERDQ  ymm1,[xmm9 * 8 + eax+8], ymm2
```

destination: ymm1 — index: xmm9 — scale: 8 — base+offset: eax+8 — write mask: ymm2

Intrinsics

```
dst = _mm_i32gather_epi64(pBase, ymm9, scale)
```

destination: dst — base: pBase — index: ymm9 — scale: scale



Gather - The whole Set

- Destination register either XMM or YMM
  - Mask register matches destination
  - No difference in instruction name
- Load of FP data or int data
  - 4 or 8 byte
- Index size 4 or 8 bytes

This results in 2x2x2x2=16 instructions

Index Size \ Data Size	4	8
4	VGATHERDPS VPGATHERDD	VGATHERDPD VPGATHERDQ
8	VGATHERQPS VPGATHERQD	VGATHERQPD VPGATHERQQ

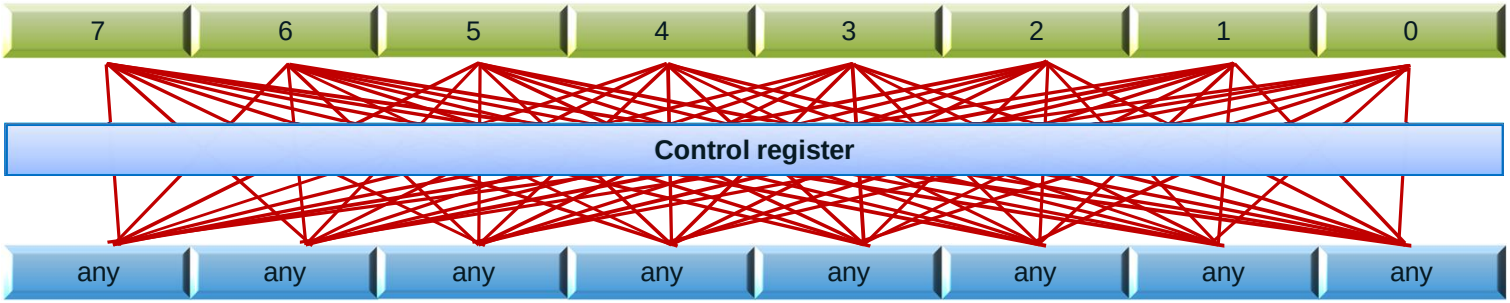
- Gather is fully deterministic
  - Same behavior from run to run even in case of exceptions
  - Mask bits are set to 0 after processing
    - Gather is complete when mask is all 0
    - The instruction can be restarted in the middle

New Data Movement Instructions

VPERMQ/PD imm	Permutes & Blends
VPERMD/PS var	
VPBLENDD imm	
VPSLLVQ & VPSRLVQ Quadword Variable Vector Shift (Left & Right Logical)	Element Based Vector Shifts
VPSLLVD, VPSRLVD, VPSRAVD Doubleword Variable Vector Shift (Left & Right Logical, Right Arithmetic)	
VPBROADCASTB/W/D/Q XMM & mem	New Broadcasts Register broadcasts requested by software developers
VBROADCASTSS/SD XMM	

Shuffling / Data Rearrangement

- Traditional IA focus was lowest possible latency
  - Many very specializes shuffles:
    - Unpacks, Packs, in-lane shuffles
  - Shuffle controls were not data driven
- New Strategy
  - **Any-to-any, data driven shuffles at slightly higher latency**



Element Width	Vector Width	Instruction	Launch
BYTE	128	PSHUFB	SSE4
DWORD	256	VPERMD VPERMPS	AVX2 New!
QWORD	256	VPERMQ VPERMPD	AVX2 New!

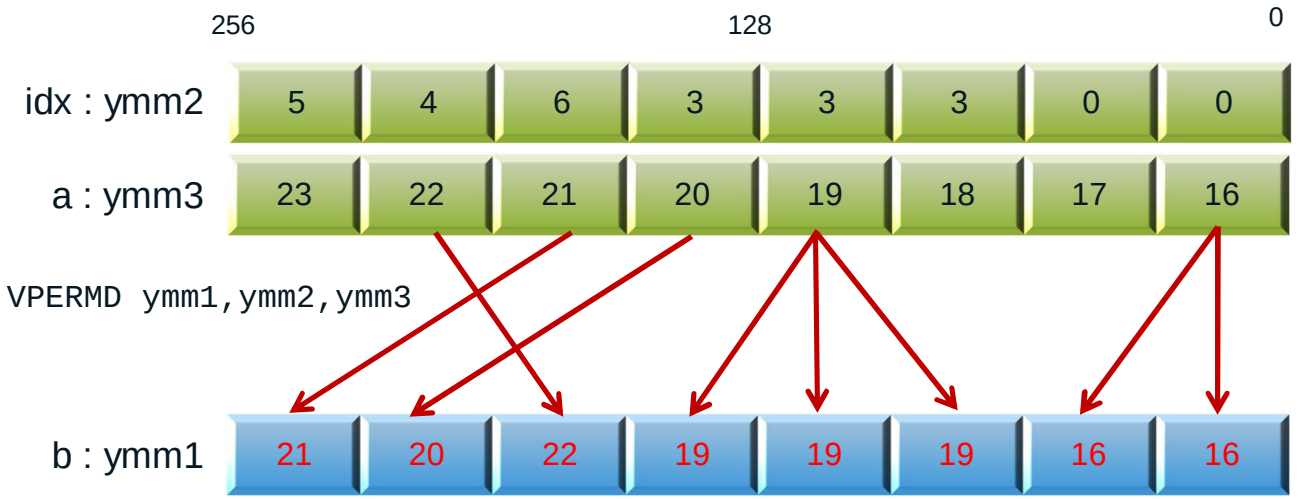
VPERMD

Instruction

```
VPERMD    ymm1, ymm2, ymm3/m256
```

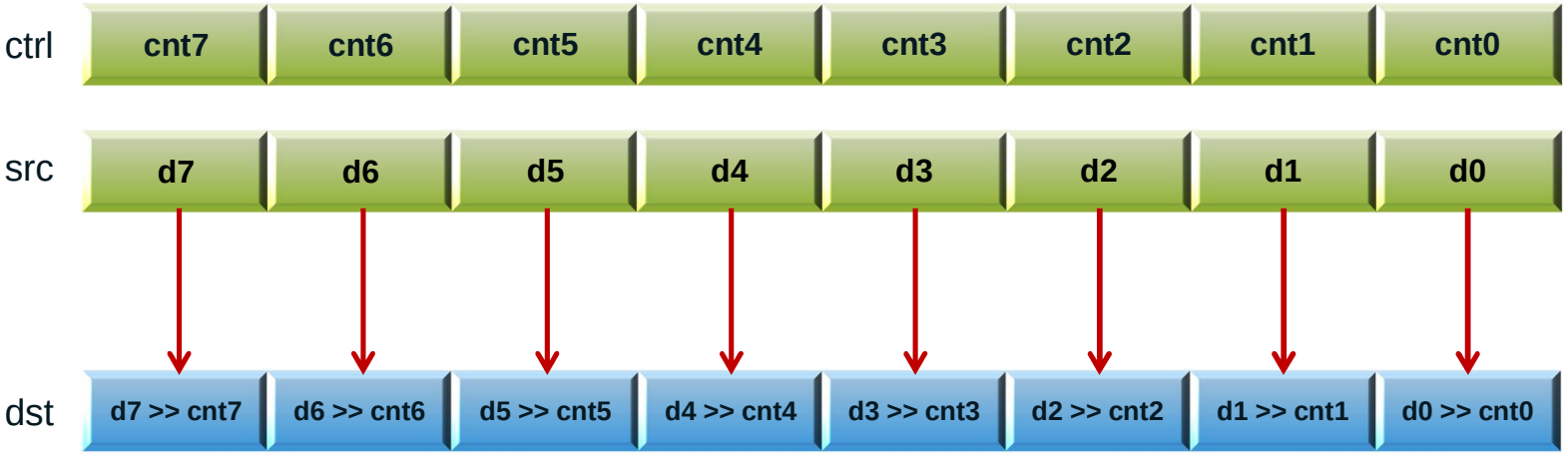
Intrinsics

```
b = _mm256_permutevar8x32_epi32(__m256i a, __m256i idx)
```



Variable Bit Shift

- Different control for each element
  - Previous shifts had one control for ALL elements



Element with	<<	>>(Logical)	>> (Arithmetic)
DWORD	VPSLLVD	VPSRLVD	VPSRAVD
QWORD	VPSLLVQ	VPSRLVQ	(not implemented)

* If the controls are greater than data width, then the destination data element are written with “0”.

FMA: Fused Multiply-Add

Computes $(a \times b) \pm c$ with only one round

- $a \times b$ intermediate result is not rounded before add/sub

Can speed up and improve the accuracy of many FP computations, e.g.,

- Matrix multiplication (SGEMM, DGEMM, etc.)
- Dot product
- Polynomial evaluation

Can perform 8 single-precision FMA operations or 4 double-precision FMA operations with 256-bit vectors per FMA unit

- Increases FLOPS capacity over Intel® AVX
- Maximum throughput of two FMA operations per cycle

FMA can provide improved accuracy and performance

20 FMA Varieties Supported

vFMAdd	a×b + c	fused mul-add	ps, pd, ss, sd
vFMSub	a×b – c	fused mul-sub	ps, pd, ss, sd
vFNMAdd	– (a×b + c)	fused negative mul-add	ps, pd, ss, sd
vFNMSub	– (a×b – c)	fused negative mul-sub	ps, pd, ss, sd
vFMAddSub	a[i]×b[i] + c[i] on odd indices a[i]×b[i] – c[i] on even indices		ps, pd
vFMSubAdd	a[i]×b[i] – c[i] on odd indices a[i]×b[i] + c[i] on even indices		ps, pd

Multiple FMA varieties support both data types and eliminate additional negation instructions

Haswell Bit Manipulation Instructions

15 overall, operate on general purpose registers (GPR):

- Leading and trailing zero bits counts
- Trailing bit manipulations and masks
- Random bit fields extract/pack
- Improved long precision multiplies and rotates

Narrowly focused instruction set arch (ISA) extension

- Partly driven by direct customers' requests
- Allows for additional performance on highly-optimized codes

Beneficial for applications *with*:

- hot spots doing bit-level operations, universal (de)coding (Golomb, Rice, Elias Gamma codes), bit fields pack/extract
- crypto algorithms using arbitrary precision arithmetic or rotates e.g.: RSA, SHA family of hashes

Bit Manipulation Instructions

CPUID bit	Name	Operation
CPUID.EAX=080000001H: ECX.**LZCNT**[bit 5] *	LZCNT	Leading Zero Count
CPUID.(EAX=07H, ECX=0H): EBX.**BMI1**[bit 3] *	TZCNT	Trailing Zero Count
	ANDN	Logical And Not Z = ~X & Y
	BLSR	Reset Lowest Set Bit
	BLSMSK	Get Mask Up to Lowest Set Bit
	BLSI	Isolate Lowest Set Bit
	BEXTR	Bit Field Extract (can also be done with SHRX + BZHI)
CPUID.(EAX=07H, ECX=0H): EBX.**BMI2**[bit 8]	BZHI	Zero High Bits Starting with Specified Position
	SHLX	Variable Shifts (non-destructive, have Load+Operation forms, no implicit CL dependency and no flags effect) Z = X << Y, Z = X >> Y (for signed and unsigned X)
	SHRX	
	SARX	
	PDEP	Parallel Bit Deposit
	PEXT	Parallel Bit Extract
	RORX	Rotate Without Affecting Flags
	MULX	Unsigned Multiply Without Affecting Flags

Note: Software needs to check for CPUID bits of *all* groups it uses instructions from

Intel® Transactional Synchronization Extensions (Intel® TSX)

Intel® TSX = HLE + RTM

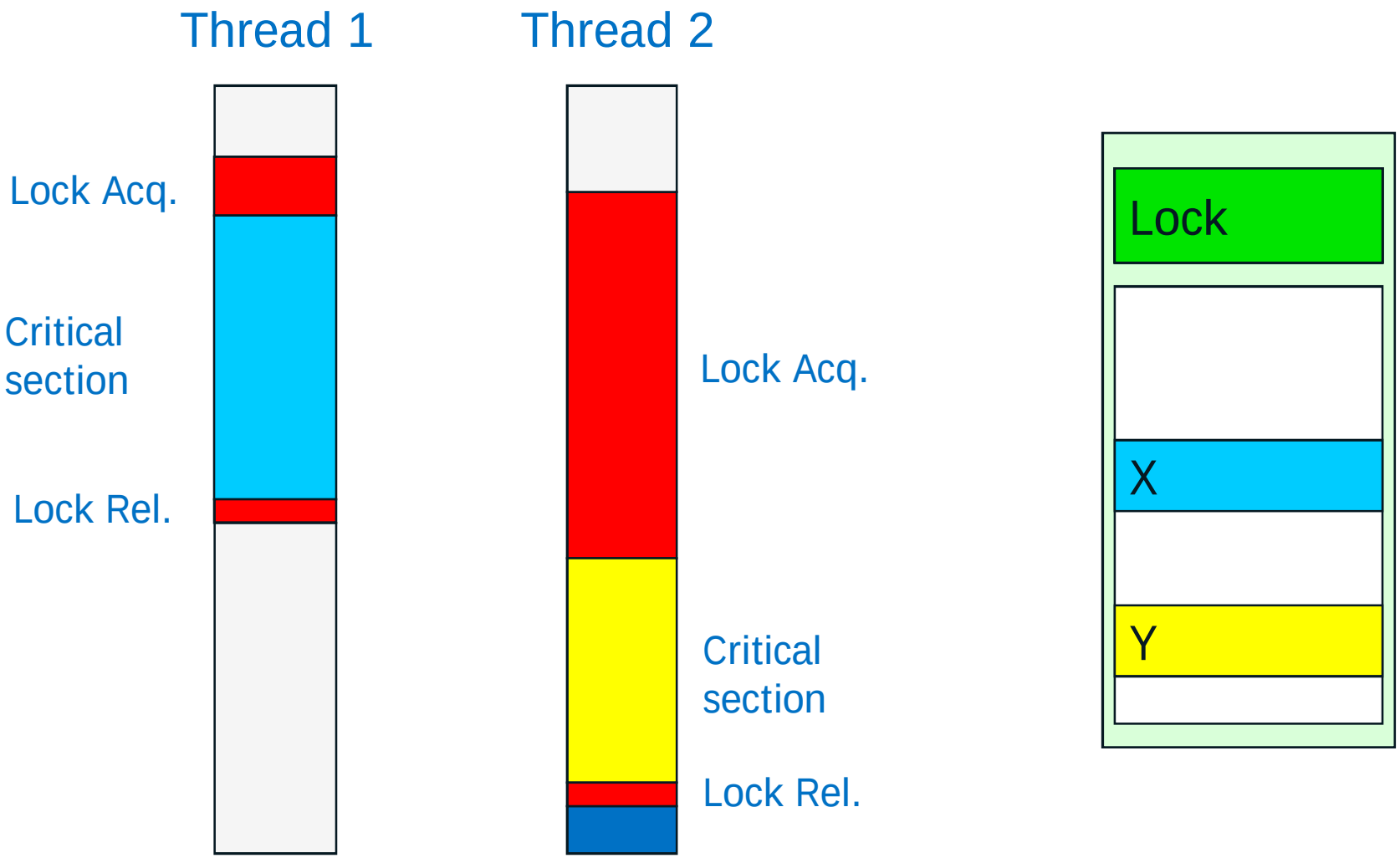
HLE (Hardware Lock Elision) is a hint inserted in front of a LOCK operation to indicate a region is a candidate for lock elision

- XACQUIRE (0xF2) and XRELEASE (0xF3) prefixes
- Don't actually acquire lock, but execute region speculatively
- Hardware buffers loads and stores, checkpoints registers
- Hardware attempts to commit atomically without locks
- If cannot do without locks, restart, execute non-speculatively

RTM (Restricted Transactional Memory) is three new instructions (XBEGIN, XEND, XABORT)

- Similar operation as HLE (except no locks, new ISA)
- If cannot commit atomically, go to handler indicated by XBEGIN
- Provides software additional capabilities over HLE

Typical Lock Use: Thread Safe Hash Table

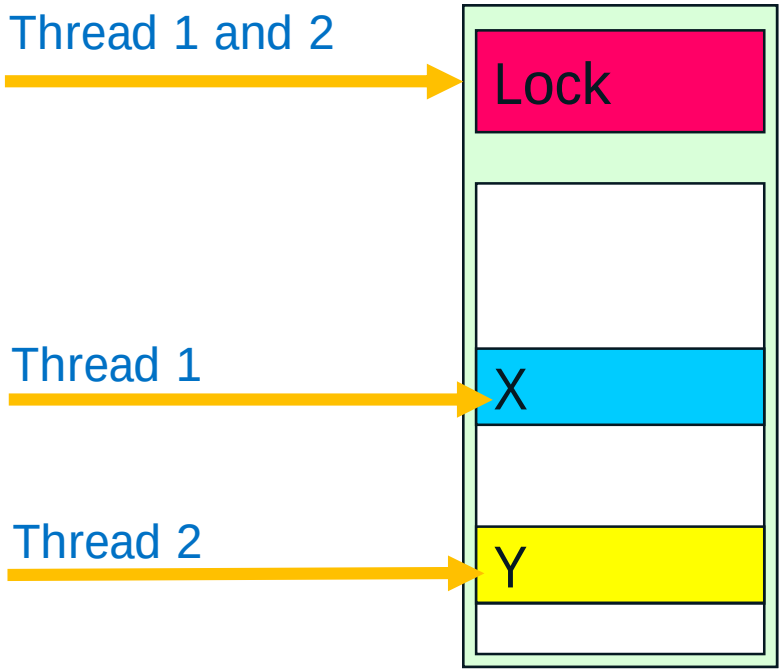


Focus on data conflicts, not lock contention

Lock Contention vs. Data Conflict

Lock contention present

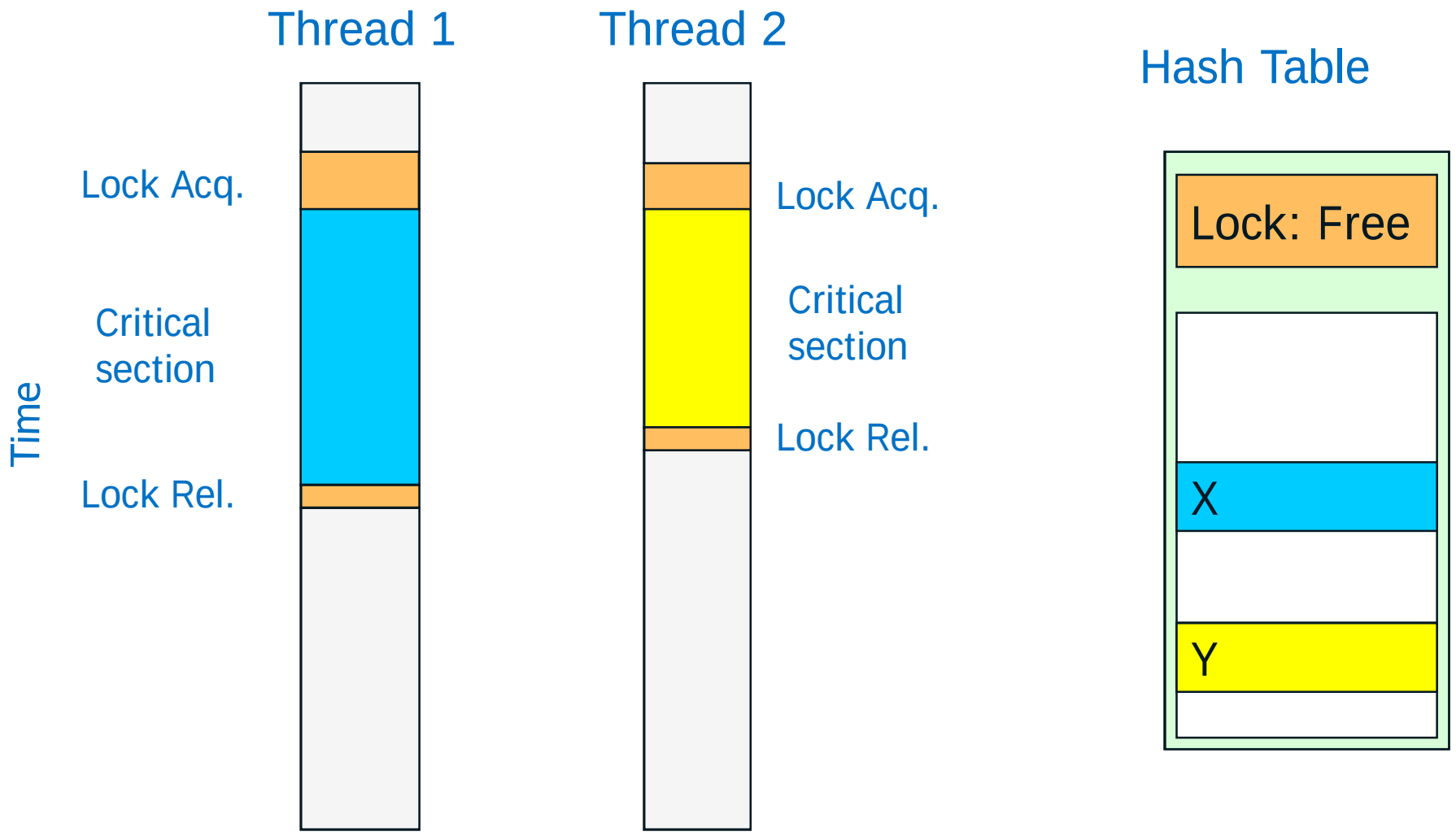
No data conflicts



Data conflicts truly limit concurrency, not lock contention

Focus on data conflicts, not lock contention

HLE Execution



No serialization if no data conflicts

HLE: Two New Prefixes

```
mov eax, 0
mov ebx, 1
F2 lock cmpxchg $semaphore, ebx
...
```

Add a prefix hint to those instructions (lock inc, dec, cmpxchg, etc.) that identify the start of a critical section. Prefix is ignored on non-HLE systems.

```
mov eax, $value1
mov $value2, eax
mov $value3, eax
...
```

Speculate

```
...
F3 mov $semaphore, 0
```

Add a prefix hint to those instructions (e.g., stores) that identify the end of a critical section.

RTM: Three New Instructions

Name	Operation
XBEGIN <rel16/32>	Transaction Begin
XEND	Transaction End
XABORT arg8	Transaction Abort

XBEGIN: Starts a transaction in the pipeline. Causes a checkpoint of register state and all following memory transactions to be buffered. Rel16/32 is the fallback handler. Nested depth support up to 7 XBEGINS. Abort if depth is 8. All abort roll-back is to outermost region.

XEND: Causes all buffered state to be atomically committed to memory. LOCK ordering semantics (even for empty transactions)

XABORT: Causes all buffered state to be discarded and register checkpoint to be recovered. Will jump to the XBEGIN labeled fallback handler. Takes imm8 argument.

There is also an XTEST instruction which can be used both for HLE and RTM to query whether execution takes place in a transaction region

SW visibility for HLE and RTM

HLE

- Execute exact same code path if HLE aborts
- Legacy compatible: transparent to software

RTM

- Execute alternative code path if RTM aborts
- Visible to software
  - Provides ‘return code’ with reason tra

XTEST: New instruction to check if in HLE or RTM

- Can be used inside HLE and RTM
- Software can determine if hardware in HLE/RTM execution

Intel Compilers – Haswell Support

- Compiler options
  - -Qxcore-avx2, -Qaxcore-avx2 (Windows*)
  - -xcore-avx2 and –axcore-avx2 (Linux)
  - -march=core-avx2 Intel compiler
  - Separate options for FMA:
    - -Qfma, -Qfma- (Windows)
    - -fma, -no-fma (Linux)
- AVX2 – integer 256-bit instructions
  - Asm, intrinsics, automatic vectorization
- Gather
  - Asm, intrinsics, automatic vectorization
- BMI – Bit manipulation instructions
  - All supported through asm/intrinsics
  - Some through automatic code generation
- INVPCID – Invalidate processor context
  - Asm only

Other Compilers – Haswell Support

- Windows: Microsoft Visual Studio* 11 will have similar support as Intel compiler
- GNU Compiler (4.7 experimental, 4.8 final) will have similar support as Intel compiler
- In particular switch -march=core-avx2 and same intrinsics

References

New Intel® AVX (AVX2) instructions specification

- http://software.intel.com/file/36945

Forum on Haswell New Instructions

- http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/

Article from Oracle engineers on how to use hardware-supported transactional memory on user level code (not Intel® TSX specific)

- http://labs.oracle.com/scalable/pubs/HTM-algs-SPAA-2010.pdf

Legal Disclaimer & Optimization Notice

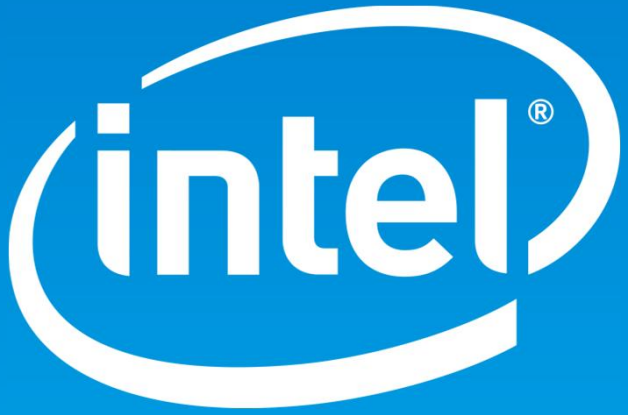
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING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

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

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


intel

VGATHERDD

Instruction

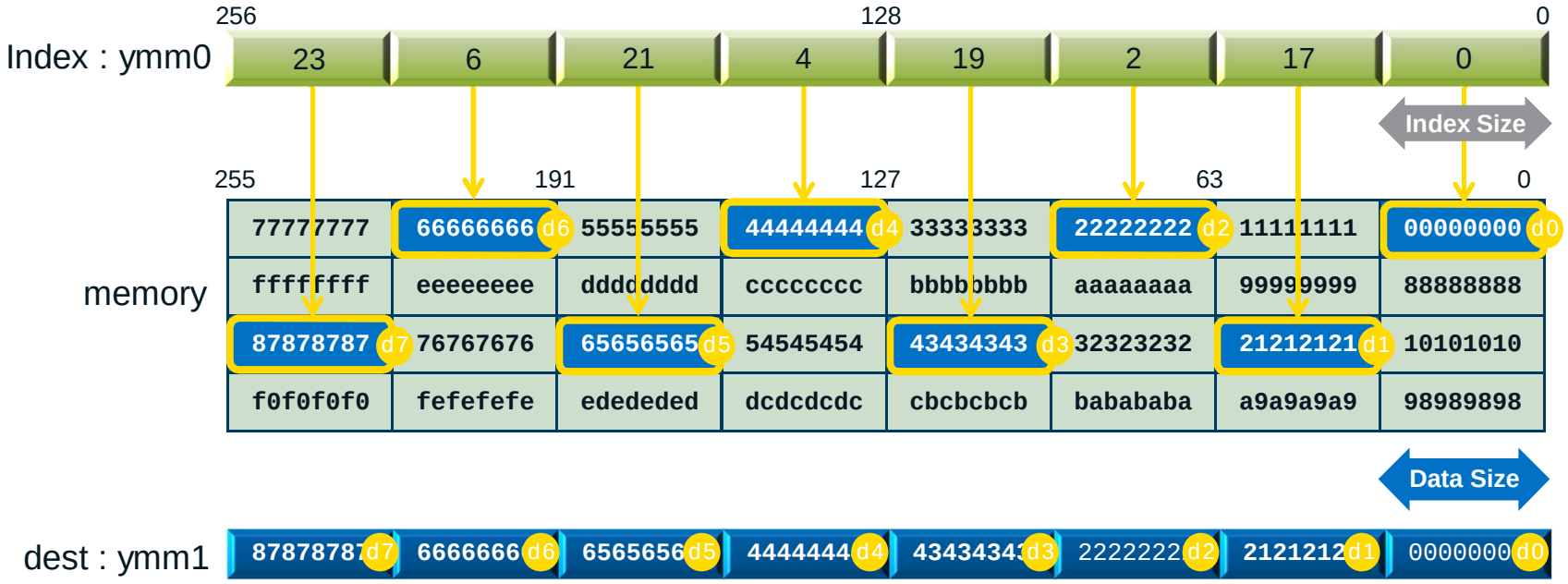
```
VPGATHERDD    ymm1,[ymm0*4 {*scale} + eax+8], ymm2
```

destination · index · base · write mask

Intrinsics

```
dst = _mm_i32gather_epi32(pBase, ymm0, scale)
```

destination · base · index



VGATHERQQ

Instruction

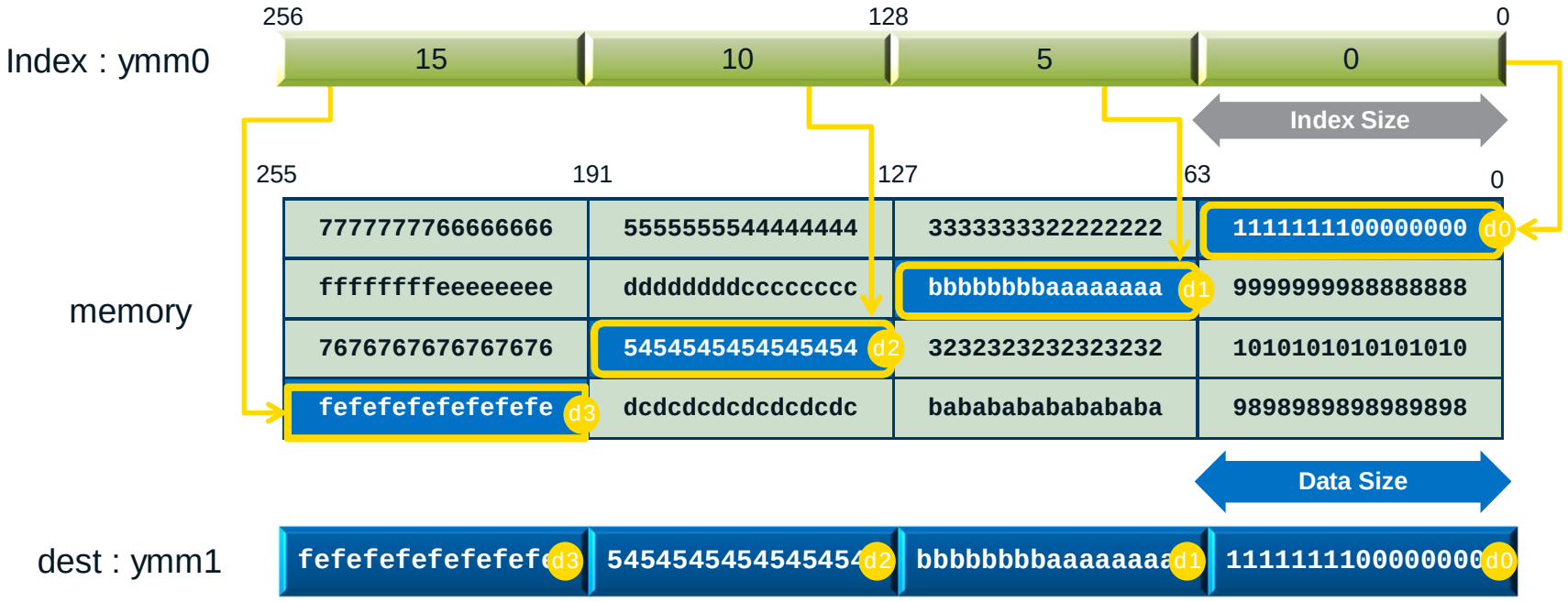
```
VPGATHERQQ     ymm1,[ymm0*8 {*scale} + eax+8], ymm2
```

destination · index · base · write mask

Intrinsics

```
dst = _mm_i64gather_epi64(pBase, ymm0, scale)
```

destination · base · index



VGATHERQD

Instruction

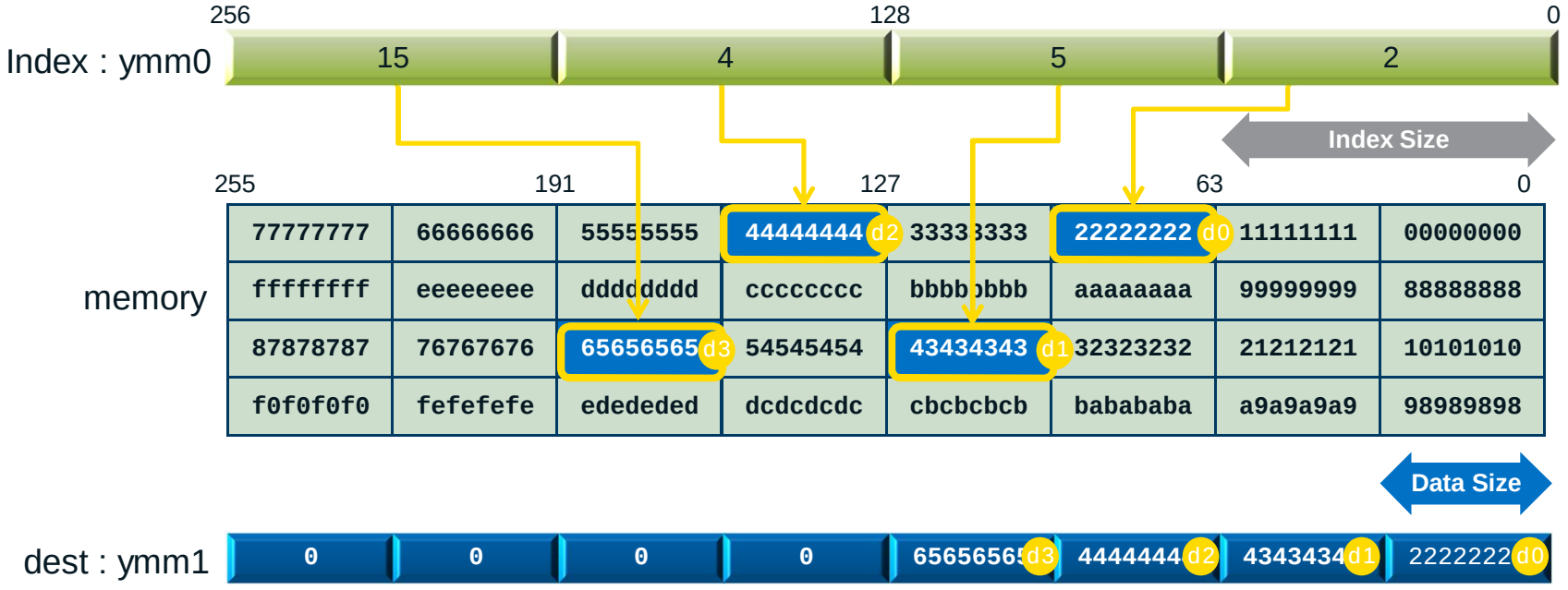
```
VPGATHERQD    ymm1,[ymm0*4 {*scale} + eax+8], ymm2
```

destination: ymm1 — index: ymm0 — base: eax+8 — write mask: ymm2

Intrinsics

```
dst = _mm_i64gather_epi32(pBase, ymm0, scale)
```

destination: dst — base: pBase — index: ymm0



All Gather Instructions

Index Size			Data Size: 4	Data Size: 8
4	4	FP	**VGATHERDPS** Gather Packed **SP FP** using signed **Dword** Indices __m128i _mm_i32gather_ps() __m128i _mm_mask_i32gather_ps() __m256i _mm256_i32gather_ps() __m256i _mm256_mask_i32gather_ps()	**VGATHERDPD** Gather Packed **DP FP** using signed **Dword** Indices __m128i _mm_i32gather_pd() __m128i _mm_mask_i32gather_pd() __m256i _mm256_i32gather_pd() __m256i _mm256_mask_i32gather_pd()
		Int	**VGATHERDD** Gather Packed **Dword** using signed **Dword** Indices __m128i _mm_i32gather_epi32() __m128i _mm_mask_i32gather_epi32() __m256i _mm256_i32gather_epi32() __m256i _mm256_mask_i32gather_epi32()	**VPGATHERDQ** Gather Packed **Qword** using signed **Dword** Indices __m128i _mm_i32gather_epi64() __m128i _mm_mask_i32gather_epi64() __m256i _mm256_i32gather_epi64() __m256i _mm256_mask_i32gather_epi64()
	8	FP	**VGATHERQPS** Gather Packed **SP FP** using signed **Qword** Indices __m128i _mm_i64gather_ps() __m128i _mm_mask_i64gather_ps() __m256i _mm256_i64gather_ps() __m256i _mm256_mask_i64gather_ps()	**VGATHERQPD** Gather Packed **DP FP** using signed **Qword** Indices __m128i _mm_i64gather_pd() __m128i _mm_mask_i64gather_pd() __m256i _mm256_i64gather_pd() __m256i _mm256_mask_i64gather_pd()
		Int	**VGATHERQD** Gather Packed **Dword** using signed **Qword** Indices __m128i _mm_i64gather_epi32() __m128i _mm_mask_i64gather_epi32() __m256i _mm256_i64gather_epi32() __m256i _mm256_mask_i64gather_epi32()	**VPGATHERQQ** Gather Packed **Qword** using signed **Qword** Indices __m128i _mm_i64gather_epi64() __m128i _mm_mask_i64gather_epi64() __m256i _mm256_i64gather_epi64() __m256i _mm256_mask_i64gather_epi64()

FMA Operand Ordering Convention

3 orders: VFMADD132, VFMADD213, VFMADD231

VFMADD*abc:* Src*a* = (Src*a* × Src*b*) + Src*c*

- Src*a* and Src*b* are numerically symmetric (interchangeable)
- Src*a* must be a register & is always the destination
- Src*b* must be a register
- Src*c* can be memory or register
- All 3 can be used in multiply or add

Example: ↙ Src*a* ↙ Src*b* ↙ Src*c*

VFMADD231 xmm8, xmm9, mem256

xmm8 = (xmm9 × mem256) + xmm8

Notes

- Combination of 20 varieties with 3 operand orderings result in 60 new mnemonics

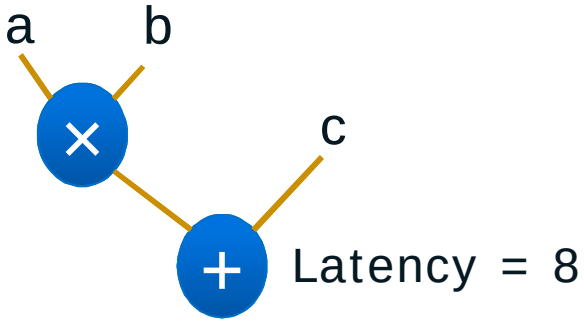
FMA operand ordering allows complete flexibility in selection of memory operand and destination

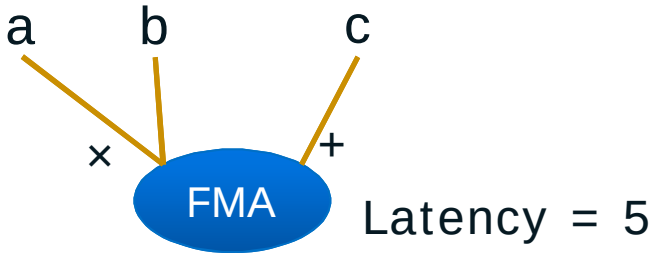
FMA Latency

Due to out-of-order, FMA latency is not always better than separate multiply and add instructions

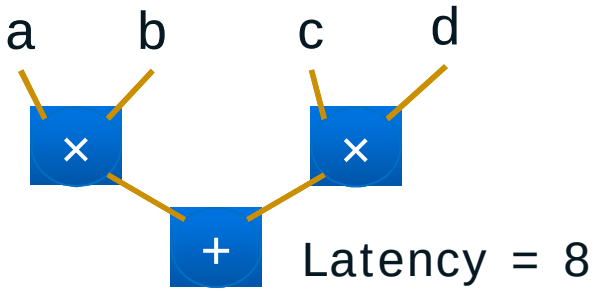
- Add latency: 3 cycles
- Multiply and FMA latencies: 5 cycles
- Will likely differ in later CPUs

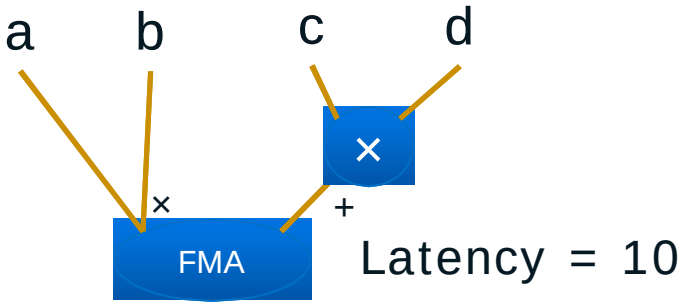
$(a \times b) + c)$





$(a \times b) + (c \times d)$





FMA can improve or reduce performance due to various factors

Fused Multiply-Add –All Combinations

	Double Precision Packed FP	Single Precision Packed FP	Double Precision Scalar FP	Single Precision Scalar FP
Fused Multiply-Add A = A x B + C C += A x B	VFMADD132PD VFMADD213PD VFMADD231PD _mm_fmadd_pd() _mm256_fmadd_pd()	VFMADD132PS VFMADD213PS VFMADD231PS _mm_fmadd_ps() _mm256_fmadd_ps()	VFMADD132SD VFMADD213SD VFMADD231SD _mm_fmadd_sd() _mm256_fmadd_sd()	VFMADD132SS, FMADD213SS VFMADD231SS _mm_fmadd_ss() _mm256_fmadd_ss()
Fused Multiply-Alternating Add/Subtract A = A x B + C | A = A x B – C C += A x B | C -= A x B	VFMADDSUB132PD VFMADDSUB213PD VFMADDSUB231PD _mm_fmaddsub_pd() _mm256_fmaddsub_pd( )	VFMADDSUB132PS VFMADDSUB213PS VFMADDSUB231PS _mm_fmaddsub_ps() _mm256_fmaddsub_ps()		
Fused Multiply-Alternating Subtract/Add A = A x B - C | A = A x B + C C -= A x B | C += A x B	VFMSUBADD132PD VFMSUBADD213PD VFMSUBADD231PD _mm_fmsubadd_pd() _mm256_fmsubadd_pd( )	VFMSUBADD132PS VFMSUBADD213PS VFMSUBADD231PS _mm_fmsubadd_pd() _mm256_fmsubadd_pd()		
Fused Multiply-Subtract A = A x B - C C -= A x B	VFMSUB132PD VFMSUB213PD VFMSUB231PD _mm_fmsub_pd() _mm256_fmsub_pd()	VFMSUB132PS VFMSUB213PS VFMSUB231PS _mm_fmsub_ps() _mm256_fmsub_ps()	VFMSUB132SD VFMSUB213SD VFMSUB231SD _mm_fmsub_sd() _mm256_fmsub_sd()	VFMSUB132SS VFMSUB213SS VFMSUB231SS _mm_fmsub_ss() _mm256_fmsub_ss()
Fused Negative Multiply-Add A = -A x B + C C += -A x B	VFNMADD132PD VFNMADD213PD VFNMADD231PD _mm_fnmadd_pd() _mm256_fnmadd_pd()	VFNMADD132PS VFNMADD213PS VFNMADD231PS _mm_fnmadd_ps() _mm256_fnmadd_ps()	VFNMADD132SD VFNMADD213SD VFNMADD231SD _mm_fnmadd_sd() _mm256_fnmadd_sd()	VFNMADD132SS VFNMADD213SS VFNMADD231SS _mm_fnmadd_ss() _mm256_fnmadd_ss()
Fused Negative Multiply-Subtract A = -A x B - C C -= -A x B	VFNMSUB132PD VFNMSUB213PD VFNMSUB231PD _mm_fnmsub_pd() _mm256_fnmsub_pd()	VFNMSUB132PS VFNMSUB213PS VFNMSUB231PS _mm_fnmsub_ps() _mm256_fnmsub_ps()	VFNMSUB132SD VFNMSUB213SD VFNMSUB231SD _mm_fnmsub_sd() _mm256_fnmsub_sd()	VFNMSUB132SS VFNMSUB213SS VFNMSUB231SS _mm_fnmsub_ss() _mm256_fnmsub_ss()