Intel® 64 and IA-32 Architectures Optimization Reference Manual

Documentation Changes

August 2023

Document Number: 355308-002

Contents

Revision History

Revision	Description	Date
-001	Initial release	May 2023
-002	Q3 Release	August 2023

Preface

This document is an update to the optimization recommendations contained in the Intel® 64 and IA-32 Architectures Optimization Reference Manual, also known as the Software Optimization Manual. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Nomenclature

Documentation Changes include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 Architecture software optimization topics covered by this reference manual.

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 2
2	Updates to Chapter 3
3	Updates to Chapter 5
4	Updates to Chapter 20
5	Updates to Appendix E
6	Updates to Appendix F
7	Updates to Appendix D

Documentation Changes

Changes to the Intel® 64 and IA-32 Architectures Optimization Reference Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 2

Change bars and **violet** text show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Introduction.

Changes to this chapter:

- Section 2.3
  - Updated Figure 2-1 to correct a typo
- Section 2.4:
  - Updated Figure 2-3 to match style of 2-1
- Section 2.7
  - Removed section Relating to Knights Landing: “Intel Xeon Phi processors based on the Knights Landing microarchitecture support 4 logical processors in each processor core; see Chapter 23 for detailed information of Intel HT Technology that is implemented in the Knights Landing microarchitecture.”
  - Updated Figure 2-9 to match style of other Figures.

CHAPTER 2 INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel® 64 and IA-32 processors[1]. These features are:

- Microarchitectures that enable executing instructions with high throughput at high clock speeds, a high-speed cache hierarchy, and high-speed system bus.
- Intel® Hyper-Threading Technology[2] (Intel® HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- Single Instruction Multiple Data (SIMD) instruction extensions: MMX™ technology, Streaming SIMD Extensions (Intel® SSE), Streaming SIMD Extensions 2 (Intel® SSE2), Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® SSE4.1, and Intel® SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
- Intel® Thread Director.

2.1 SAPPHIRE RAPIDS MICROARCHITECTURE

Intel processors based on Sapphire Rapids microarchitecture use Golden Cove cores and support the following additional features:

- Intel® Advanced Matrix Extensions (Intel® AMX) (Chapter 20).
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) (Chapter 19).
- Intel® Data Streaming Accelerator (Intel® DSA)[3].
- Intel® In-Memory Analytics Accelerator (Intel® IAA)[4].
- Intel® Quick Assist Technology (Intel® QAT)(Chapter 22)

2.1.1 4th Generation Intel® Xeon® Scalable Family of Processors

Intel's fourth generation Xeon® Scalable Family of Processors changes from a single-die monolithic design to multi-die Tiles.

The server products are scalable from dual-socket to eight-socket configurations (Section 3.11).

The I/O is increased with PCI Express 5.0, DDR5 memory, and Compute Express Link 1.1.

Packaging includes a multi-die chip with up to 4 tiles. Each tile is a 400mm2 SoC, providing both compute cores and I/O.

1. Intel Atom® processors are covered in Chapter 4, “Intel Atom® Processor Architectures.”
2. Intel HT Technology requires a computer system with an Intel processor supporting hyper-threading and an Intel HT Technology-enabled chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.
3. Please see the Intel® DSA Specification and Intel® DSA User Guide.
4. Please see the Intel® IAA Specification.

Each tile contains 15 Golden Cove cores (see Section 2.3). Its memory controller provides two channels of DDR5 with a maximum of eight channels across 4 tiles, and 28 PCIe 5.0 lanes for a maximum of 112 across 4 tiles.

2.2 ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE

The Alder Lake performance hybrid architecture combines two Intel architectures, bringing together the Golden Cove performant cores and the Gracemont efficient Atom cores onto a single SoC. For details on the Golden Cove microarchitecture, see Section 2.3 For details on the Gracemont microarchitecture, see Section 4.1

2.2.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

12th Generation Intel® Core™ processors supporting performance hybrid architecture consist of up to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MB Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has symmetrical ISA and comes in variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. P-cores on these processors can also have Intel Hyper-Threading Technology enabled. All cores can be active simultaneously when the operating system (OS) decides to schedule on all processors.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-cores and E-cores. In order to maintain symmetric ISA, the E-cores do not support the following features: Intel AVX-512, Intel AVX-512 FP-16, and Intel® TSX. The E-cores do support Intel AVX2 and Intel AVX-VNNI.

2.2.2 Hybrid Scheduling

2.2.2.1 Intel® Thread Director

Intel® Thread Director continually monitors software in real-time giving hints to the operating system's scheduler allowing it to make more intelligent and data-driven decisions on thread scheduling. With Intel Thread Director, hardware provides runtime feedback to the OS per thread based on various IPC performance characteristics, in the form of:

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director is first introduced in desktop and mobile variants of the 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.

A processor containing both P-cores and E-cores with different performance characteristics creates a challenge for the operating system’s scheduler. Additionally, different software threads see different performance ratios between the P-cores and E-cores. For example, the performance ratio between the P-cores and E-cores for highly vectorized floating-point code is higher than the performance ratio for scalar integer code. So, when the operating system needs to make an optimal scheduling decision it needs to be aware of the characteristics of the software threads that are candidates for scheduling. If not enough P-cores are available and there is a mix of software threads with different characteristics, the operating system should schedule those threads that benefit most from the P-cores onto those cores and schedule the others on the E-cores.

Intel Thread Director provides the necessary hint to the operating system about the characteristics of the software thread executing on each of the logical processors. The hint is dynamic and reflects the recent characteristics of the thread, i.e., it may change over time based on the dynamic instruction mix of the thread. The processor also considers microarchitecture factors to define the dynamic software thread characteristics.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the operating system via writing to configuration MSRs. The Intel Thread Director implementation on processors based on Alder Lake performance hybrid architecture defines four thread classes:

0. Non-vectorized integer or floating-point code.
1. Integer or floating-point vectorized code, excluding Intel® Deep Learning Boost (Intel® DL Boost) code.
2. Intel DL Boost code.
3. Pause (spin-wait) dominated code.

The dynamic code does not have to be 100% of the class definition. It should be large enough to be considered belonging to that class. Also, dynamic microarchitectural metrics such as consumed memory bandwidth or cache bandwidth may move software threads between classes. Example pseudo-code sequences for the Intel Thread Director classes available on processors based on Alder Lake performance hybrid architecture are provided in the Examples 2-1 through 2-4.

Intel Thread Director also provides a table in system memory, only accessible to the operating system, that defines the P-core vs. E-core performance ratio per class. This allows the operating system to pick and choose the right software thread for the right logical processor.

In addition to the performance ratio between P-cores and E-cores, Intel Thread Director provides the energy efficiency ratio between those cores. The operating system can then use this information when it prefers energy savings over maximum performance. For example, a background task such as indexing can be scheduled on the most energy efficient core since its performance is less critical.

Example 2-1. Class 0 Pseudo-code Snippet

```
while (1)
{
    asm("xor rax, rax;"
        "add rax, 5;"
        "inc rax;"
    );
}
```

Example 2-2. Class 1 Pseudo-code Snippet

```
while (1)
{
    asm("vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm3;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm4;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm5;"
        "vfmaddsub213ps %ymm0, %ymm1, %ymm6;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm7;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm8;"

        "vfmaddsub213ps %ymm0, %ymm1, %ymm9;"
        "vfmaddsub231ps %ymm0, %ymm1, %ymm10;"
        "vfmaddsub132ps %ymm0, %ymm1, %ymm2;"
    );
}
```

Example 2-3. Class 2 Pseudo-code Snippet

```
while (1)
{
    __asm(
            vpdpbusd ymm2, ymm0, ymm1
            vpdpbusd ymm3, ymm0, ymm1
            vpdpbusd ymm4, ymm0, ymm1
            vpdpbusd ymm5, ymm0, ymm1
            vpdpbusd ymm6, ymm0, ymm1
            vpdpbusd ymm7, ymm0, ymm1
            vpdpbusd ymm8, ymm0, ymm1
            vpdpbusd ymm9, ymm0, ymm1
            vpdpbusd ymm10, ymm0, ymm1
            vpdpbusd ymm11, ymm0, ymm1
            vpdpbusd ymm12, ymm0, ymm1
            vpdpbusd ymm13, ymm0, ymm1
    );
}
```

Example 2-4. Class 3 Pseudo-code Snippet

```
while (1)
{
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
            asm("PAUSE;")
    );
}
```

For more detailed information on this technology, refer to the Intel® 64 and IA-32 Architectures Software Developer's Manual.

2.2.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-threads busy.

2.2.2.3 Scheduling with a Multi-E-Core Module

E-cores within an idle module help provide better performance than E-cores in a busy module.

2.2.2.4 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage scalability and multithread efficiency of E-cores.

2.2.3 Recommendations for Application Developers

The following are recommendations when using processors supporting performance hybrid architecture:

- Stay up to date on updates on operating systems and optimized libraries.
- Software needs to avoid setting hard affinities on either threads or processes in order to allow the operating system to provide the optimal core selection for Intel Hybrid.
- Software should replace active spin-waits with lightweight waits ideally using the new UMWAIT/TPAUSE and older PAUSE instructions which will allow for better hints to the scheduler on time spinning.
- Software can utilize the Windows Power Throttling information using process information and thread information APIs, to give hints to the scheduler on the Quality of Service (QoS) required for a particular thread or process to improve both performance and energy efficiency.
- Leverage Windows frameworks and media APIs for multimedia application development. Windows Media Foundation framework is optimized for hybrid architecture and enables media applications to run efficiently while preventing glitches.
- The Windows IrqPolicyMachineDefault policy enables Windows to optimally target interrupts to the right core, and more so on hybrid architecture.

For additional recommendations and information on performance hybrid architecture, refer to the white papers on the Performance Hybrid Architecture page.

2.3 GOLDEN COVE MICROARCHITECTURE

The Golden Cove microarchitecture is the successor of Ice Lake microarchitecture. The Golden Cove microarchitecture introduces the following enhancements:

- Wider machine: 5→6 wide allocation, 10→12 execution ports, and 4→8 wide retirement.
- Significant increases in the size of key structures enable deeper OOO execution and expose more instruction level parallelism.
- Greater capabilities per execution port, e.g., 5th integer ALU execution ports with expanded capability and a new fast floating-point adder.
- Intel® Advanced Matrix Extensions (Intel® AMX)[1]: Built-in integrated Tiled Matrix Multiplication / Machine Learning Accelerator.
- Improved branch prediction.
- Improvements for large code footprint workloads, e.g., larger branch prediction structures, enhanced code prefetcher, and larger instruction TLB.
- Wider fetch: legacy decode pipeline fetch bandwidth increase to 32B/cycles, 4→6 decoders, increased micro-op cache size, and increased micro-op cache bandwidth.
- Maximum load bandwidth increased from 2 loads/cycle to 3 loads/cycle.
- Larger 4K Pages DTLB, increase in the number of outstanding Page Miss handlers.
- Increased number of outstanding misses (16 FB, 32→48 Deeper MLC miss queues).

1. Intel AMX are not available on client parts.

- Enhanced data prefetchers for increased memory parallelism.
- Mid-level cache size increased to 2MB on server parts; remains 1.25MB on client parts.

2.3.1 Golden Cove Microarchitecture Overview

The basic pipeline functionality of the Golden Cove microarchitecture is depicted in Figure 2-1.

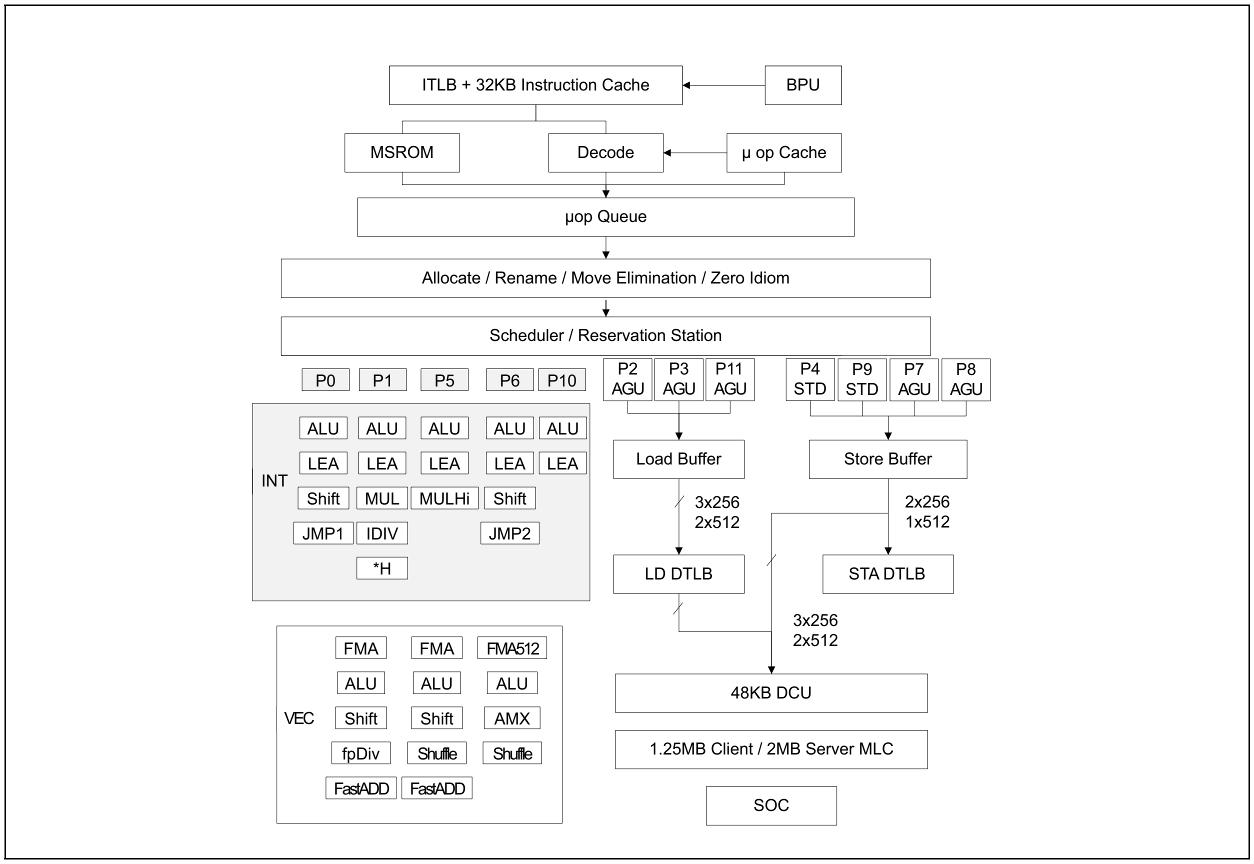


Figure 2-1. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture

The Golden Cove front end is depicted in Figure 2-2. The front end is built to feed the wider and deeper out-of-order core:

- Legacy decode pipeline fetch bandwidth increased from 16 to 32 bytes/cycle.
- The number of decoders increased from four to six, allowing decode of up to 6 instructions per cycle.
- The micro-op cache size increased, and its bandwidth increased to deliver up to 8 micro-ops per cycle.
- Improved branch prediction.

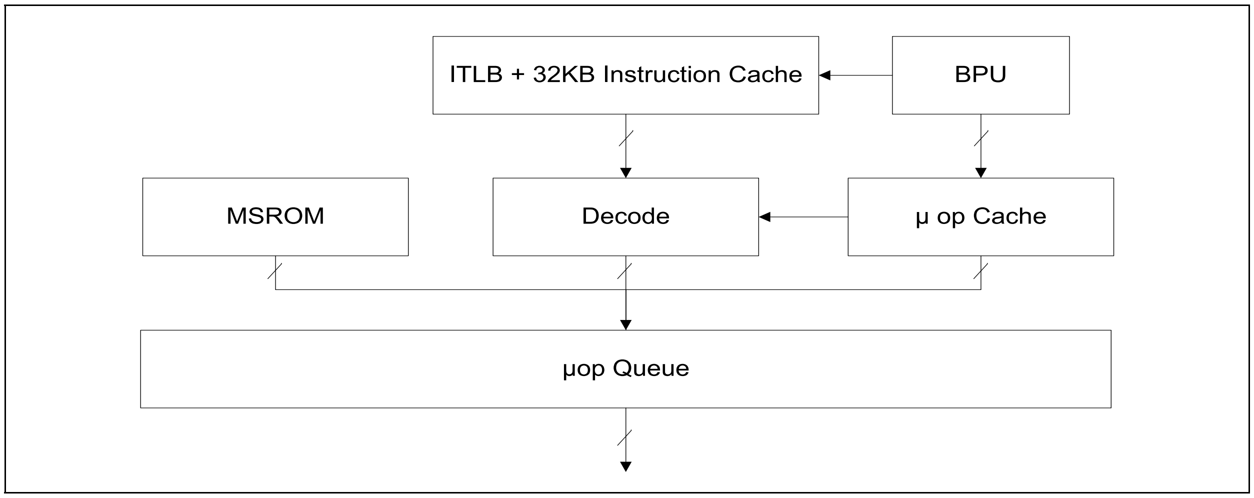


Figure 2-2. Processor Front End of the Golden Cove Microarchitecture

Improvements for large code footprint workloads:

- Double the size of the instruction TLB: 128→256 entries for 4K pages, 16→32 entries for 2M/4M pages.
- Bigger branch prediction structures.
- Enhanced code prefetcher.
- Improved LSD coverage.
- The IDQ can hold 144 uops per logical processor in single thread mode, or 72 uops per thread when SMT is active.

Additional improvements include:

- Significant increase in size of key buffer structures to enable deeper OOO execution and expose more instruction level parallelism.
- Wider machine:
  - — Wider allocation (5→6 uops per cycle) and retirement (4→8 uops per cycle) width.
  - — Increase in number of execution ports (10→12).
  - — Greater capabilities per execution port.

Table 2-1 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-1. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU[3] LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle*				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. "*" in this table indicates that these features are not available for 512-bit vectors.
2. "**" in this table indicates that these features are not available for 512-bit vectors in Client parts.
3. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

Table 2-2 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel® SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-2. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	5[2]	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2[3]	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	2x256-bit 1x512-bit	(v)add, (v)cmp. (v)max, (v)min, (v)sub, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2sl, (v)cvtss2sl
	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si

Table 2-2. Golden Cove Microarchitecture Execution Units and Representative Instructions[1] (Contd.)

Execution Unit	# of Unit	Instructions
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)
Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.
2. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.
3. *Ibid.*

Table 2-3 describes bypass delay in cycles between producer and consumer operations.

Table 2-3. Bypass Delay Between Producer and Consumer Micro-Ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	**SIMD/0,1/1**	**FMA/0,1/4**	**MUL/0,1/4**	**Fast Adder/1,5/3**	**SIMD/5/1,3**	**SHUF/ 1,5/1, 3**	**V2I/0/3**
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to a 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.
- "V2I/0/3" applies to a 3-cycle vector-to-integer uop dispatched to port 0.

- "I2V/5/1" applies to a 1-cycle integer-to-vector uop dispatched to port 5.
- "Fast Adder/1,5/3" applies to either a 3-cycle 256-bit uop dispatched to either port 1 or port 5, or a 512-bit uop dispatched to port 5. This operation supports two cycles back-to-back between a pair of Fast Adder operations.

A new Fast Adder[1] unit is added as 512-bit on port 5 in VEC stack, and as 256-bit on ports 1 and 5. The Fast Adder performs floating-point ADD/SUB operations in 3 cycles.

Back-to-back ADD/SUB operations that are both executed on the Fast Adder unit perform the operations in two cycles.

- In 128/256-bit, back-to-back ADD/SUB operations executed on the Fast Adder unit perform the operations in two cycles.
- In 512-bit, back-to-back ADD/SUB operations are executed in two cycles if both operations use the Fast Adder unit on port 5.

The following instructions are executed by the Fast Adder unit:

- (V)ADDSUBSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.3.1.1 Cache Subsystem and Memory Subsystem

The cache subsystem and memory subsystem changes in the Golden Cove microarchitecture are:

- Maximum load bandwidth increased from 2 to 3 loads per cycle. Bandwidth of Intel AVX-512 loads, Intel AMX loads, and MMX/x87 loads remain at a maximum of 2 loads per cycle.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Number of entries for 4K pages in the load DTLB increased from 64 to 96.
- Page Miss handler can handle up to four D-side page walks in parallel instead of two.
- Increased number of outstanding DCU and MLC misses.
- Enhanced data prefetchers for increased memory parallelism.
- Partial store forwarding allowing forwarding data from store to load also when only part of the load was covered by the store (in case the load's offset matches the store's offset).

2.3.1.2 Avoiding Destination False Dependency

Some SIMD instructions incur false dependency on the destination operand. The following instructions are affected:

- VFMULCSH, VFMULCPH
- VFCMULCSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

1. The Fast Adder unit is not available on 512-bit vectors in Client parts.

***Recommendation:** Use dependency breaking zero idioms on the destination register before the affected instructions to avoid potential slowdown from the false dependency.*

Example 2-5. Breaking False Dependency through Zero Idiom

Code with False Dependency Impact	Mitigation: Break False Dependency with Zero Idiom
vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vfmulcph zmm3, zmm2, zmm1 ;False dependency on zmm3. Will not execute out-of-order until vaddps writes zmm3.	vaddps zmm3, zmm4, zmm5 vmovaps [rsi], zmm3 vpxord zmm3, zmm3, zmm3 ;Dependency-breaking zero idiom. vfmulcph zmm3, zmm2, zmm1 ;Execute out-of-order without waiting for vaddps result.

2.4 ICE LAKE CLIENT MICROARCHITECTURE

The Ice Lake client microarchitecture introduces the following new features that allow optimizations of applications for performance and power consumption:

- Targeted vector acceleration.
- Crypto acceleration.
- Intel® Software Guard Extensions (Intel® SGX) enhancements.
- Cache line writeback instruction (CLWB).

2.4.1 Ice Lake Client Microarchitecture Overview

The Ice Lake client microarchitecture builds on the successes of the Skylake client microarchitecture. The basic pipeline functionality of the Ice Lake Client microarchitecture is depicted in Figure 2-3.

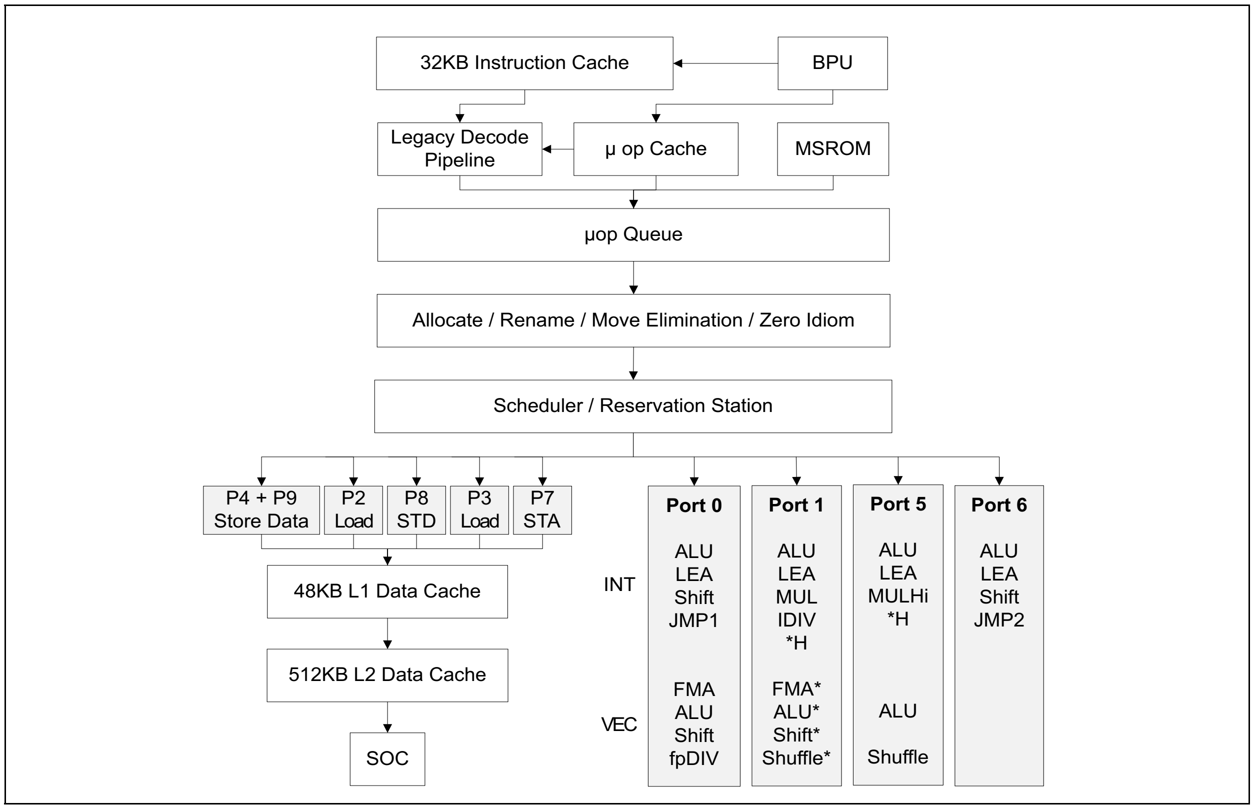


Figure 2-3. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

NOTES:

1. "*" in the figure above indicates these features are not available for 512-bit vectors.
2. "INT" represents GPR scalar instructions.
3. "VEC" represents floating-point and integer vector instructions.
4. "MULHi" produces the upper 64 bits of the result of an iMul operation that multiplies two 64-bit registers and places the result into two 64-bits registers.
5. The "Shuffle" on port 1 is new, and supports only in-lane shuffles that operate within the same 128-bit sub-vector.
6. The "IDIV" unit on port 1 is new, and performs integer divide operations at a reduced latency.
7. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

The Ice Lake client microarchitecture introduces the following new features:

- Significant increase in size of key structures enable deeper OOO execution.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Intel AVX-512 (new for client processors): 512-bit vector operations, 512-bit loads and stores to memory, and 32 new 512-bit registers.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA), reduced latency Integer Divider.
- 2×BW for AES-NI peak throughput for existing binaries (microarchitectural).
- Rep move string acceleration.
- 50% increase in size of the L1 data cache.
- Reduced effective load latency.
- 2×L1 store bandwidth: 1 → 2 stores per cycle.
- Enhanced data prefetchers for increased memory parallelism.
- Larger 2nd level TLB.
- Larger uop cache.
- Improved branch predictor.
- Large page ITLB size in single thread mode doubled.
- Larger L2 cache.

The Ice Lake client microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

2.4.1.1 The Front End

The front end changes in Ice Lake Client microarchitecture include:

- Improved branch predictor.
- Large page ITLB in single thread mode increased from 8 to 16 entries.
- Larger uop cache.
- The IDQ can hold 70 uops per logical processor vs. 64 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2×70 vs. 2×64 per core). If only one logical processor is active in the core, the IDQ can hold 70 uops vs. 64 uops.
- The LSD in the IDQ can detect loops of up to 70 uops per logical processor irrespective single thread or multi thread operation.

2.4.1.2 The Out of Order and Execution Engines

The Out of Order and execution engines changes in Ice Lake client microarchitecture include:

- A significant increase in size of reorder buffer, load buffer, store buffer, and reservation stations enable deeper OOO execution and higher cache bandwidth.
- Wider machine: 4 → 5 wide allocation, 8 → 10 execution ports.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA).
- Reduced latency Integer Divider.
- A new iDIV unit was added that significantly reduces the latency and improves the of throughput of integer divide operations.

Table 2-4 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-4. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

NOTES:

1. "*" in this table indicates these features are not available for 512-bit vectors.

Table 2-5 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-5. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si

Table 2-5. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
Shuffle	2	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

Table 2-6 describes bypass delay in cycles between producer and consumer operations.

Table 2-6. Bypass Delay Between Producer and Consumer Micro-ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	**SIMD/0,1/1**	**FMA/0,1/4**	**VIMUL/0,1/4**	**SIMD/5/1,3**	**SHUF/5/1, 3**	**V2I/0/3**	**I2V/5/1**
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.
- "V2I/0/3" applies to a 3-cycle vector-to-integer uop dispatched to port 0.
- "I2V/5/1" applies to a 1-cycle integer-to-vector uop to port 5.

2.4.1.3 Cache and Memory Subsystem

The cache hierarchy changes in Ice Lake Client microarchitecture include:

- 50% increase in size of the L1 data cache.
- 2×L1 store bandwidth: 3 → 4 AGUs, 1 → 2 store data.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Higher cache bandwidth compared to previous generations.
- Larger 2nd level TLB: 1.5K entries → 2K entries.
- Enhanced data prefetchers for increased memory parallelism.
- L2 cache size increased from 256KB to 512KB.
- L2 cache associativity increased from 4 ways to 8 ways.
- Significant reduction in effective load latency.

Table 2-7. Cache Parameters of the Ice Lake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Latency[1] (cycles)	Peak Bandwidth (bytes/cycles)	Sustained Bandwidth (bytes/cycles)	Update Policy
First Level (DCU)	48KB/8	64	5	2×64B loads + 1x64B or 2x32B stores	Same as peak	Writeback
Second Level (MLC)	512KB/8	64	13	64	48	Writeback
Third Level (LLC)	Up to 2MB per core/up to 16 ways	64	xx[2]	32	21	Writeback

NOTES:

1. Software-visible latency/bandwidth will vary depending on access patterns and other factors.
2. This number depends on core count.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, shared L2 TLB for 4K and 4MB pages and a dedicated L2 TLB for 1GB pages.

Table 2-8. TLB Parameters of the Ice Lake Client Microarchitecture

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Instruction	4KB	128	64	8
Instruction	2MB/4MB	16	8	8
First Level Data (loads)	4KB	64	64 competitively shared	4
First Level Data (loads)	2MB/4MB	32	32 competitively shared	4
First Level Data (loads)	1GB	8	8 competitively shared	8
First Level Data (stores)	Shared for all page sizes	16	16 competitively shared	16
Second Level	Shared for all page sizes	2048[1]	2048 competitively shared	16

NOTES:

1. 4K pages can use all 2048 entries. 2/4MB pages can use 1024 entries (in 8 ways), sharing them with 4K pages. 1GB pages can use the other 1024 entries (in 8 ways), also sharing them with 4K pages.

Paired Stores

Ice Lake Client microarchitecture includes two store pipelines in the core, with the following features:

- Two dedicated AGU for LDs on ports 2 and 3.
- Two dedicated AGU for STAs on ports 7 and 8.
- Two fully featured STA pipelines.
- Two 256-bit wide STD pipelines (Intel AVX-512 store data takes two cycles to write).
- Second senior store pipeline to the DCU via store merging.

Ice Lake Client microarchitecture can write two senior stores to the cache in a single cycle if these two stores can be paired together. That is:

- The stores must be to the same cache line.
- Both stores are of the same memory type, WB or USWC.
- None of the stores cross cache line or page boundary.

In order to maximize performance from the second store port try to:

- Align store operations whenever possible.
- Place consecutive stores in the same cache line (not necessarily as adjacent instructions).

As seen in Example 2-6, it is important to take into consideration all stores, explicit or not.

Example 2-6. Considering Stores

Stores are Paired Across Loop Iterations	Stores Not Paired Due to Stack Update in Between

Loop:
    compute reg
    ...
    store [X], reg
    add X, 4
    jmp Loop   ; stores from different iterations of the loop can be paired all together because they usually would be same line
	Loop:
	    call function to compute reg
	    ...
	    store [X], reg
	    add X, 4
	    jmp Loop ; stores from different iterations of the loop cannot be paired anymore because of the call store to stack
	; the call is disturbing pairing

In some cases it is possible to rearrange the code to achieve store pairing. Example 2-7 provides details.

Example 2-7. Rearranging Code to Achieve Store Pairing

Stores to Different Cache Lines - Not Paired	Unrolling May Solve the Problem

Loop:
    ... compute ymm1 ...
    vmovaps [x], ymm1
    ... compute ymm2 ...
    vmovaps [y], ymm2
    add x, 32
    add y, 32
    jmp Loop ; this loop cannot pair any store because of alternating store to different cache lines [x] and [y]
	Loop:
	    ... compute ymm1 ...
	    vmovaps [x], ymm1
	    ... compute new ymm1 ...
	    vmovaps [x+32], ymm1
	    ... compute ymm2 ...
	    vmovaps [y], ymm2
	    ... compute new ymm2 ...
	    vmovaps [y+32], ymm2
	    add x, 64
	    add y, 64
	    jmp Loop ; the loop was unrolled 2 times and stores re-arranged to make sure two stores to the same cache line are placed one after another. Now stores to addresses [x] and [x+32] are to the same cache line and could be paired together and executed in same cycle

2.4.1.4 New Instructions

New instructions and architectural changes in Ice Lake Client microarchitecture are listed below. Actual support may be product dependent.

- Crypto acceleration
  - SHA NI for acceleration of SHA1 and SHA256 hash algorithms.
  - Big-Number Arithmetic (IFMA): VPMADD52 - two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance.
  - Galois Field New Instructions (GFNI) for acceleration of various encryption algorithms, error correction algorithms, and bit matrix multiplications.
  - Vector AES and Vector Carry-less Multiply (PCLMULQDQ) instructions to accelerate AES and AES-GCM.
- Security Technologies
  - Intel® SGX enhancements to improve usability and applicability: EDMM, multi-package server support, support for VMM memory oversubscription, performance, larger secure memory.
- Sub Page protection for better performance of security VMMs.
- Targeted Acceleration
  - Vector Bit Manipulation Instructions: VBMI1 (permutes, shifts) and VBMI2 (Expand, Compress, Shifts)- used for columnar database access, dictionary based decompression, discrete mathematics, and data-mining routines (bit permutation and bit-matrix-multiplication).
  - VNNI with support for integer 8 and 16 bits data types- CNN/ML/DL acceleration.
  - Bit Algebra (POPCNT, Bit Shuffle).
  - Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Platform analysis features for more efficient performance software tuning and debug.
  - AnyThread removal.

— 2x general counters (up to 8 per-thread).
— Fixed Counter 3 for issue slots.
— New performance metrics for built-in support for Level 1 Top-Down method (% of Issue slots that are front-end bound, back-end bound, bad speculation, retiring) while leaving the 8 general purpose counters free for software use.

2.4.1.5 Ice Lake Client Microarchitecture Power Management

Processors based on Ice Lake client microarchitecture are the first client processors whose cores may execute at a different frequency from one another. The frequency is selected based on the specific instruction mix; the type, width and number of vector instructions of the program that executes on each core, the ratio between active time and idle time of each core, and other considerations such as how many cores share similar characteristics.

Most of the power management features of Skylake Server Microarchitecture (see Section 2.5) is applicable to Ice Lake Client microarchitecture as well. The main differences are the following:

- The typical P0n max frequency difference between Intel® Advanced Vector Extensions (Intel® AVX-512) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) on Ice Lake Client microarchitecture is much lower than on Skylake Server microarchitecture. Therefore, the negative impact on overall application performance is much smaller.
- All processors based on Ice Lake Client microarchitecture contain a single 512-bit FMA unit, whereas some of the processors based on Skylake Server microarchitecture contain two such units. Both processors contain two 256-bit FMA units. The power consumed by Ice Lake Client FMA units is the same, whereas on Skylake Server the 512-bit units consume twice as much.

Compute heavy workloads, especially those that span multiple Ice Lake client cores, execute at a lower frequency than P0n, both under Intel AVX-512 and under Intel AVX2 instruction sets, due to power limitations. In this scenario, Intel AVX-512 architecture, which requires less dynamic instructions to complete the same task than Intel AVX2 architecture, consumes less power and thus may achieve higher frequency. The net result may be higher performance due to the shorter path length and a bit higher frequency.

There are still some cases where coding to the Intel AVX-512 instruction set yields lower performance than when coding to the Intel AVX2 instruction set. Sometimes it is due to microarchitecture artifacts of longer vectors, in other cases the natural vectors are just not long enough. Most compilers are still maturing their Intel AVX-512 support, and it may take them a few more years to generate optimal code.

The general recommendation in the Skylake Server Power Management section (see Section 2.5.3) still holds. Developers should code to the Intel AVX-512 instruction set and compare the performance to their Intel AVX2 workload on Ice Lake client microarchitecture, before making the decision to proceed with a complete port.

2.5 SKYLAKE SERVER MICROARCHITECTURE

The Intel® Xeon® Processor scalable processor family is based on the Skylake Server microarchitecture. Processors based on the Skylake microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of CHAPTER 2 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

The Skylake Server microarchitecture introduces the following new features[1] that allow you to optimize your application for performance and power consumption.

- A new core based on the Skylake Server microarchitecture with process improvements based on the Kaby Lake microarchitecture.
- Intel AVX-512 support.
- More cores per socket (max 28 vs. max 22).

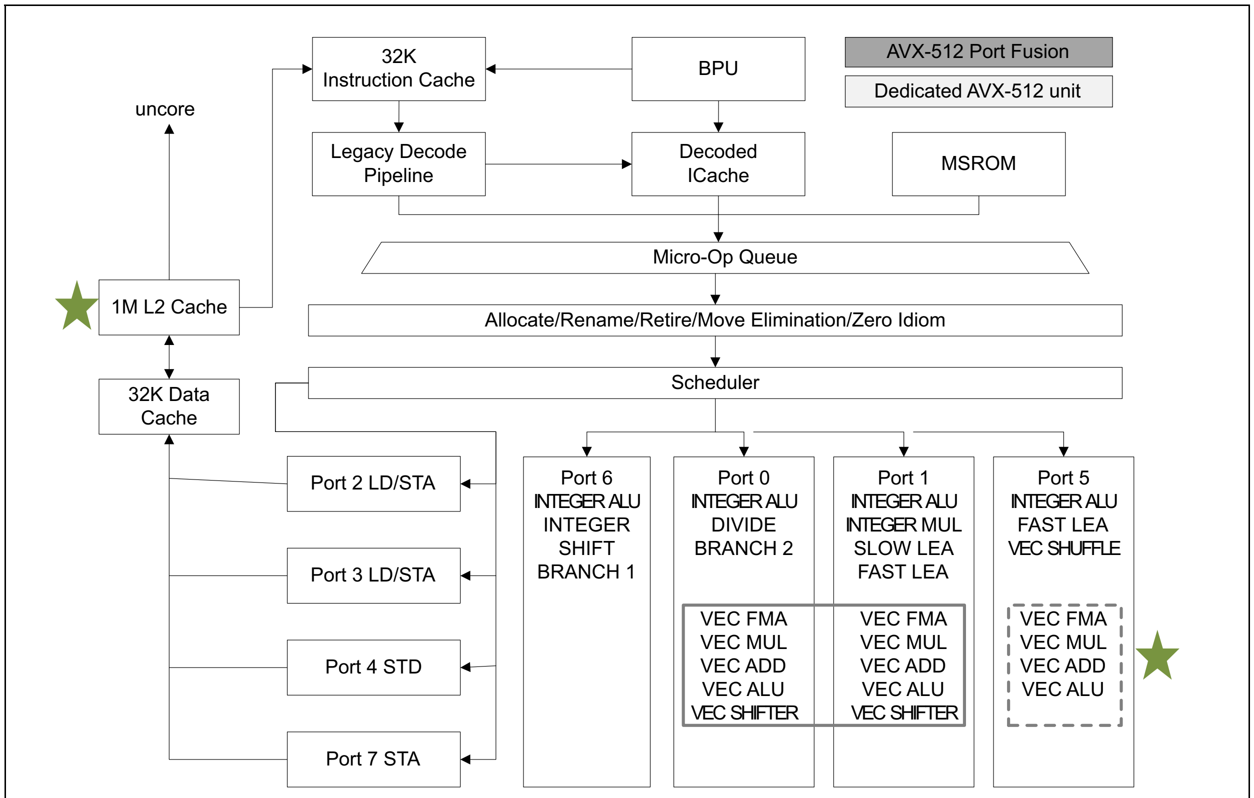
1. Some features may not be available on all products.

- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.
- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA).
- Sub-NUMA Clustering (SNC) support.

The green stars in Figure 2-4 represent new features in Skylake Server microarchitecture compared to Skylake microarchitecture for client; a 1MB L2 cache and an additional Intel AVX-512 FMA unit on port 5 which is available on some parts.

Since port 0 and port 1 are 256-bits wide, Intel AVX-512 operations that will be dispatched to port 0 will execute on both port 0 and port 1; however, other operations such as *lea* can still execute on port 1 in parallel. See the red block in Figure 2-8 for the fusion of ports 0 and 1.

Notice that, unlike Skylake microarchitecture for client, the Skylake Server microarchitecture has its front end loop stream detector (LSD) disabled.


32K Instruction Cache
BPU
AVX-512 Port Fusion
Dedicated AVX-512 unit
uncore
Legacy Decode Pipeline
Decoded ICache
MSROM
Micro-Op Queue
1M L2 Cache
Allocate/Rename/Retire/Move Elimination/Zero Idiom
Scheduler
32K Data Cache
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 6 INTEGER ALU INTEGER SHIFT BRANCH 1
Port 0 INTEGER ALU DIVIDE BRANCH 2 VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 1 INTEGER ALU INTEGER MUL SLOW LEA FAST LEA VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 5 INTEGER ALU FAST LEA VEC SHUFFLE VEC FMA VEC MUL VEC ADD VEC ALU

INTEGER represent GPR scalar instructions.
VEC represent floating point and integer vector instructions.
SLOW LEA represent a lea with two registers and displacement, all other lea versions considered as FAST LEA
BRANCH1 is primary branch and more capable than BRANCH2

Figure 2-4. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture

2.5.1 Skylake Server Microarchitecture Cache

Intel Xeon scalable processors based on Skylake server microarchitecture has significant changes in core and uncore architecture to improve performance and scalability of several components compared with the previous generation of the Intel Xeon processors based on the Broadwell microarchitecture.

2.5.1.1 Larger Mid-Level Cache

Skylake server microarchitecture implements a mid-level (L2) cache of 1 MB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64B and it is 16-way associative. The mid-level cache is private to each core.

Software that has been optimized to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake server microarchitecture.

2.5.1.2 Non-Inclusive Last Level Cache

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MB per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

2.5.1.3 Skylake Server Microarchitecture Cache Recommendations

A high-level comparison between Skylake server microarchitecture cache and the previous generation Broadwell microarchitecture cache is available in the table below.

Table 2-9. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16

Table 2-9. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375[1] per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	32
	Sustained bandwidth [bytes/cycles]	14	15

NOTES:

1. Some Skylake server parts have some cores disabled and hence have more than 1.375 MB per core of L3 cache.

The figure below shows how Skylake server microarchitecture shifts the memory balance from shared-distributed with high latency, to private-local with low latency.

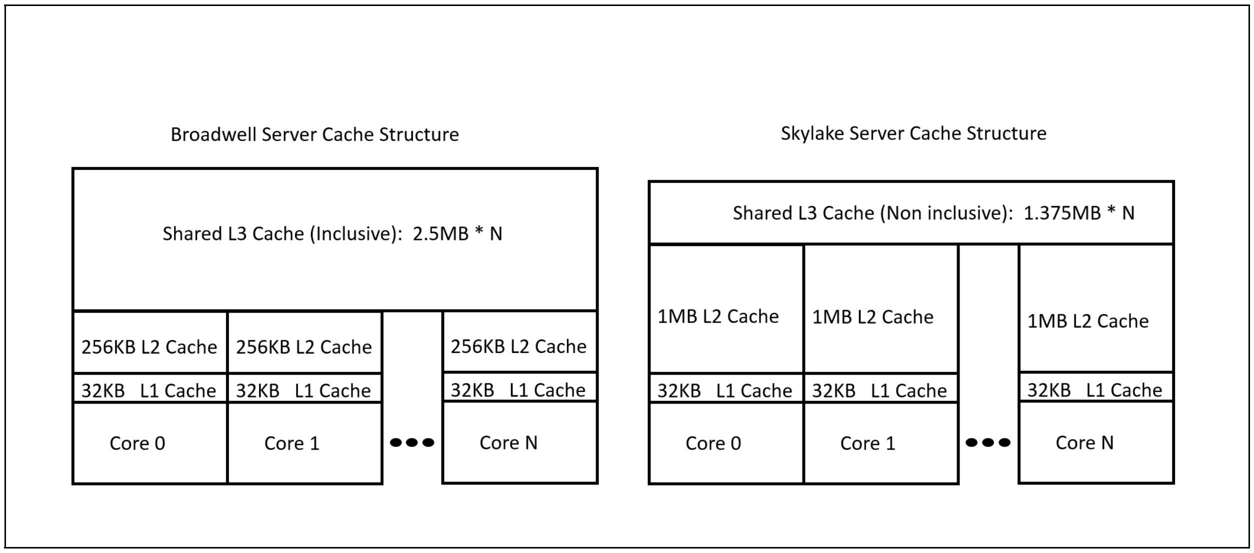

Broadwell Server Cache Structure
Shared L3 Cache (Inclusive): 2.5MB * N
256KB L2 Cache
32KB L1 Cache
Core 0
Core 1
Core N
Skylake Server Cache Structure
Shared L3 Cache (Non inclusive): 1.375MB * N
1MB L2 Cache
32KB L1 Cache
Core 0
Core 1
Core N

Figure 2-5. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures

The potential performance benefit from the cache changes is high, but software will need to adapt its memory tiling strategy to be optimal for the new cache sizes.

***Recommendation**: Rebalance application shared and private data sizes to match the smaller, non-inclusive L3 cache, and larger L2 cache.*

Choice of cache blocking should be based on application bandwidth requirements and changes from one application to another. Having four times the L2 cache size and twice the L2 cache bandwidth compared to the previous generation Broadwell microarchitecture enables some applications to block to L2 instead of L1 and thereby improves performance.

***Recommendation**: Consider blocking to L2 on Skylake Server microarchitecture if L2 can sustain the application's bandwidth requirements.*

The change from inclusive last level cache to non-inclusive means that the capacity of mid-level and last level cache can now be added together. Programs that determine cache capacity per core at run time should now use a combination of mid-level cache size and last level cache size per core to estimate the effective cache size per core. Using just the last level cache size per core may result in non-optimal use of available on-chip cache; see Section 2.5.2 for details.

***Recommendation:** In case of no data sharing, applications should consider cache capacity per core as L2 and L3 cache sizes and not only L3 cache size.*

2.5.2 Non-Temporal Stores on Skylake Server Microarchitecture

Because of the change in the size of each bank of last level cache on Skylake server microarchitecture, if an application, library, or driver only considers the last level cache to determine the size of on-chip cache-per-core, it may see a reduction with Skylake server microarchitecture and may use non-temporal store with smaller blocks of memory writes. Since non-temporal stores evict cache lines back to memory, this may result in an increase in the number of subsequent cache misses and memory bandwidth demands on Skylake Server microarchitecture, compared to the previous Intel Xeon processor family.

Also, because of a change in the handling of accesses resulting from non-temporal stores by Skylake Server microarchitecture, the resources within each core remain busy for a longer duration compared to similar accesses on the previous Intel Xeon processor family. As a result, if a series of such instructions are executed, there is a potential that the processor may run out of resources and stall, thus limiting the memory write bandwidth from each core.

The increase in cache misses due to overuse of non-temporal stores and the limit on the memory write bandwidth per core for non-temporal stores may result in reduced performance for some applications.

To avoid the performance condition described above with Skylake server microarchitecture, include mid-level cache capacity per core in addition to the last level cache per core for applications, libraries, or drivers that determine the on-chip cache available with each core. Doing so optimizes the available on-chip cache capacity on Skylake server microarchitecture as intended, with its non-inclusive last level cache implementation.

2.5.3 Skylake Server Power Management

This section describes the interaction of Skylake Server's Power Management and its Vector ISA.

Skylake Server microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics.

Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake Server microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions.

The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution.

Table 2-10 includes information about the maximum Intel® Turbo Boost technology core frequency for each type of instruction executed. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. The more cores belonging to a category at any given time, the lower the maximum frequency.

Table 2-10. Maximum Intel® Turbo Boost Technology Core Frequency Levels

Level	Category	Frequency Level	Max Frequency (P0n)	Instruction Types
0	Intel® AVX2 light instructions	Highest	Max	Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
1	Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions	Medium	Max Intel® AVX2	Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
2	Intel® AVX-512 heavy instructions	Lowest	Max Intel® AVX-512	Intel® AVX-512 FP + INT MUL/FMA

For per SKU max frequency details (reference figure 1-15), refer to the Intel® Xeon® Scalable Processor Family Technical Resources page.

Figure 2-6 is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload.

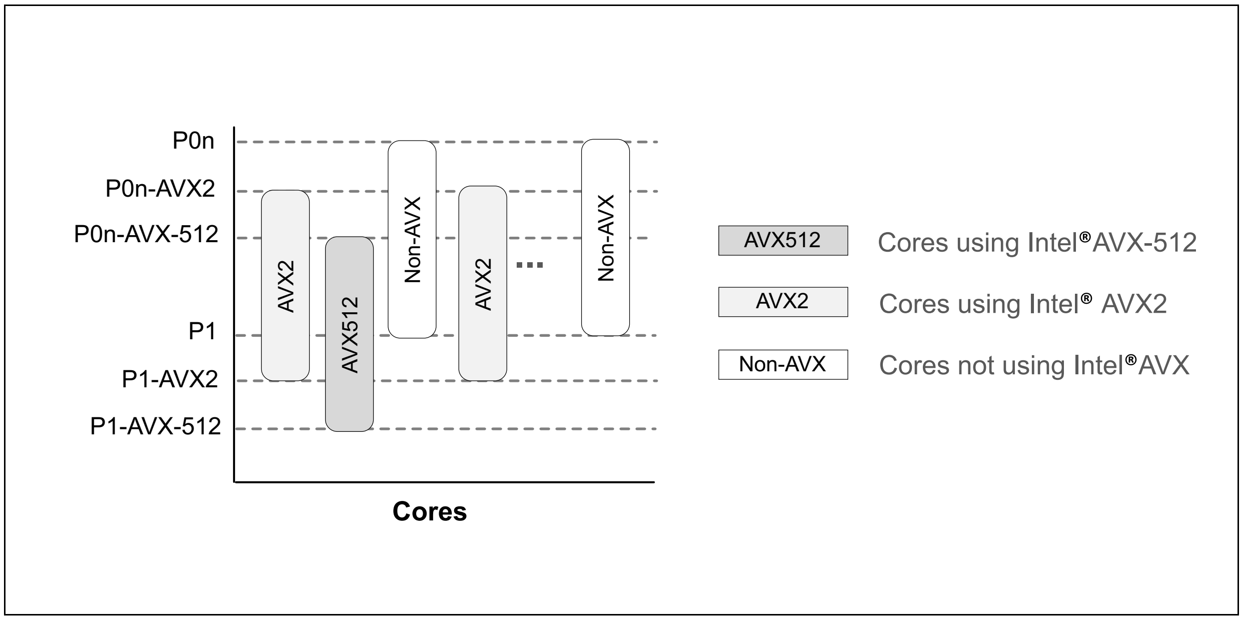


Figure 2-6. Mixed Workloads

The following performance monitoring events can be used to determine how many cycles were spent in each of the three frequency levels.

- CORE_POWER.LVL0_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n.
- CORE_POWER.LVL1_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX2.
- CORE_POWER.LVL2_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX-512.

When the core requests a higher license level than its current one, it takes the PCU up to 500 micro-seconds to grant the new license. Until then the core operates at a lower peak capability. During this time period the PCU evaluates how many cores are executing at the new license level and adjusts their frequency as necessary, potentially lowering the frequency. Cores that execute at other license levels are not affected.

A timer of approximately 2ms is applied before going back to a higher frequency level. Any condition that would have requested a new license resets the timer.

NOTES

A license transition request may occur when executing instructions on a mis-speculated path.

A large enough mix of Intel AVX-512 light instructions and Intel AVX2 heavy instructions drives the core to request License 2, despite the fact that they usually map to License 1. The same is true for Intel AVX2 light instructions and Intel SSE heavy instructions that may drive the core to License 1 rather than License 0. For example, The Intel® Xeon® Platinum 8180 processor moves from license 1 to license 2 when executing a mix of 110 Intel AVX-512 light instructions and 20 256-bit heavy instructions over a window of 65 cycles.

Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the LINPACK benchmark is power limited and does not reach the processor's maximum frequency. The following graph shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

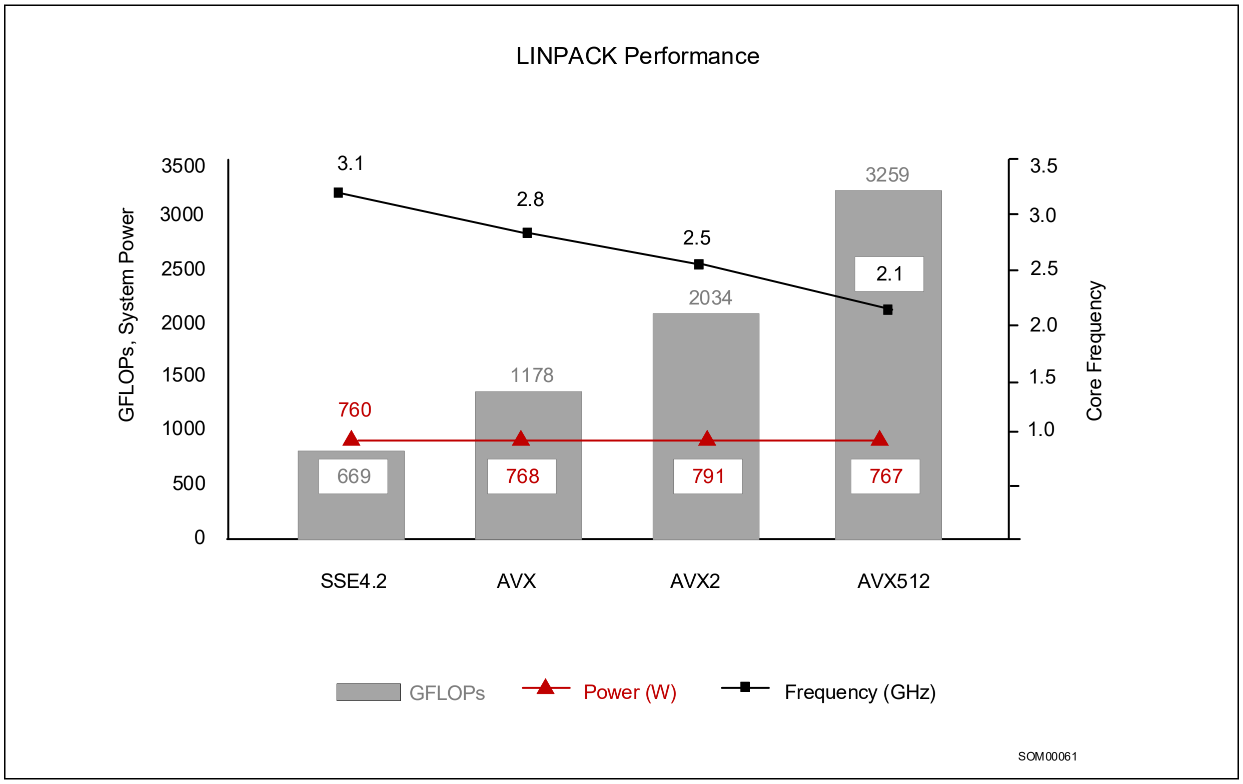


Figure 2-7. LINPACK Performance

Workloads that execute Intel AVX-512 instructions as a large proportion of their whole instruction count can gain performance compared to Intel AVX2 instructions, even though they may operate at a lower frequency. For example, maximum frequency bound Deep Learning workloads that target Intel AVX-512 heavy instructions at a very high percentage can gain 1.3x-1.5x performance improvement vs. the same workload built to target Intel AVX2 (both operating on Skylake Server microarchitecture).

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to the use of zmm registers. However, some programs that use zmm registers may not gain as much, or may even lose performance. It is recommended to try multiple build options and measure the performance of the program.

Recommendation: To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -xCORE-AVX2 -mtune=skylake-avx512 (Linux* and macOS*)
  /QxCORE-AVX2 /tune=skylake-avx512 (Windows*)
- -xCORE-AVX512 -qopt-zmm-usage=low (Linux* and macOS*)
  /QxCORE-AVX512 /Qopt-zmm-usage:low (Windows*)
- -xCORE-AVX512 -qopt-zmm-usage=high (Linux* and macOS*)
  /QxCORE-AVX512 /Qopt-zmm-usage:high (Windows*)

See Section 18.26 for more information about these options.

The GCC Compiler has the option -mprefer-vector-width=none|128|256|512 to control vector width preference. While -march=skylake-avx512 is designed to provide the best performance for the Skylake Server microarchitecture some programs can benefit from different vector width preferences. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance. -mprefer-vector-width=256 is the default for skylake-avx512.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512
- -march=skylake-avx512 -mprefer-vector-width=512

Clang/LLVM is currently implementing the option -mprefer-vector-width=none|128|256|512, similar to GCC. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512 (plus -mprefer-vector-width=256, if available)
- -march=skylake-avx512 (plus -mprefer-vector-width=512, if available)

2.6 SKYLAKE CLIENT MICROARCHITECTURE

The Skylake client microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake client microarchitecture is depicted in Figure 2-8.

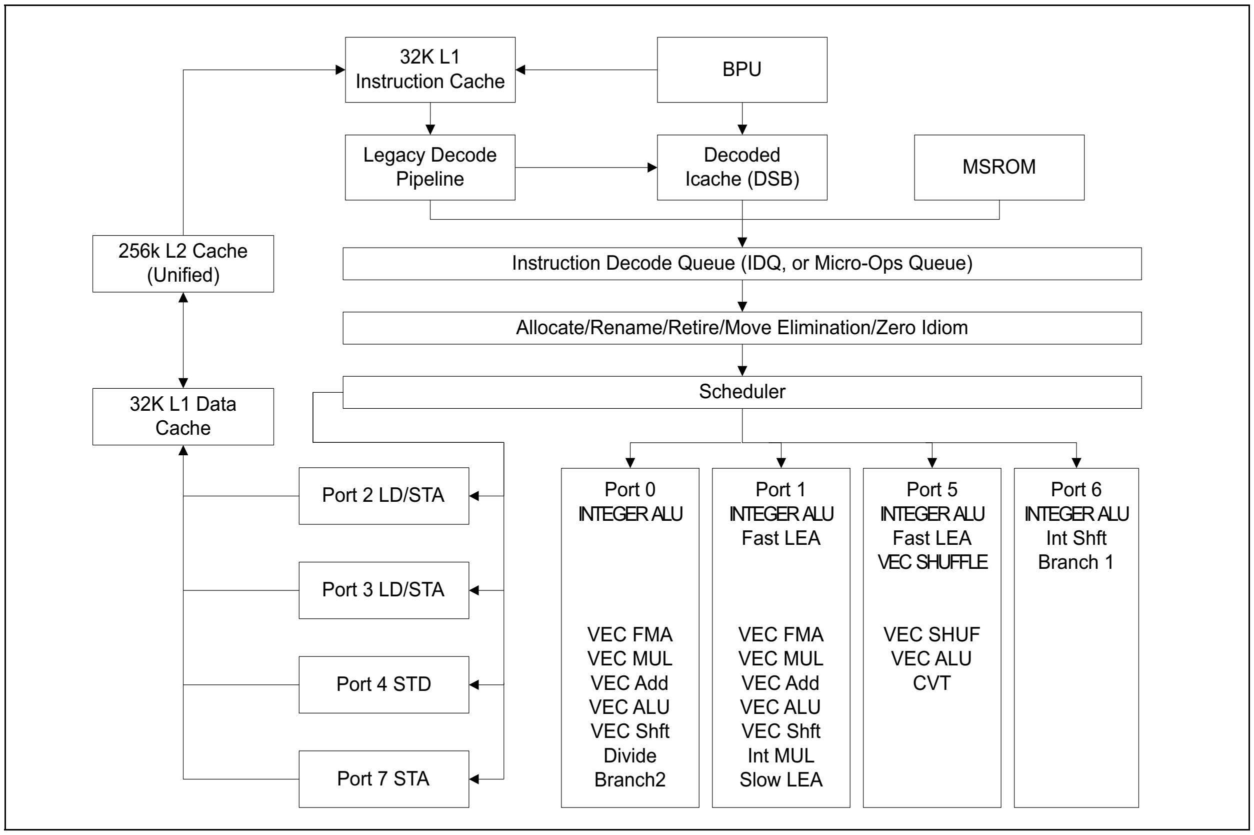

32K L1 Instruction Cache
BPU
Legacy Decode Pipeline
Decoded Icache (DSB)
MSROM
256k L2 Cache (Unified)
Instruction Decode Queue (IDQ, or Micro-Ops Queue)
Allocate/Rename/Retire/Move Elimination/Zero Idiom
32K L1 Data Cache
Scheduler
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 0 INTEGER ALU VEC FMA VEC MUL VEC Add VEC ALU VEC Shft Divide Branch2
Port 1 INTEGER ALU Fast LEA VEC FMA VEC MUL VEC Add VEC ALU VEC Shft Int MUL Slow LEA
Port 5 INTEGER ALU Fast LEA VEC SHUFFLE VEC SHUF VEC ALU CVT
Port 6 INTEGER ALU Int Shft Branch 1

Figure 2-8. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture

The Skylake Client microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 cache (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc.

2.6.1 The Front End

The front end in the Skylake Client microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The DSB delivers 6 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The IDQ can hold 64 uops per logical processor vs. 28 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 uops (64 vs. 56 uops in ST operation).
- The LSD in the IDQ can detect loops up to 64 uops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.6.2 The Out-of-Order Execution Engine

The Out of Order and execution engine changes in Skylake Client microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

Table 2-11 summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-11. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

Table 2-12 lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-12. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

A significant portion of the Intel SSE, Intel AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two-uop instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

Table 2-13 describes the bypass delay in cycles between a producer uop and the consumer uop. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-13. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "VIMUL/0,1/4" applies to 4-cycle vector integer multiply uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.

2.6.3 Cache and Memory Subsystem

The cache hierarchy of the Skylake Client microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-14. Cache Parameters of the Skylake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:
1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of Table 2-15 indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-15. TLB Parameters of the Skylake Client Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.6.4 Pause Latency in Skylake Client Microarchitecture

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake Client microarchitecture are listed below.
  — Front end slots in the Decode ICache, LSD and IDQ.
  — Execution slots in the RS.
- Save power consumed by the processor core compared with executing equivalent spin loop instruction sequence in the following configurations.
  — One logical processor is inactive (e.g., entering a C-state).
  — Both logical processors in the same core execute the PAUSE instruction.

— HT is disabled (e.g. using BIOS options).

The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake Client microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

The following is an example of how to use the PAUSE instruction with a dynamic loop iteration count.

Notice that in the Skylake Client microarchitecture the RDTSC instruction counts at the machine's guaranteed P1 frequency independently of the current processor clock (see the INVARIANT TSC property), and therefore, when running in Intel® Turbo-Boost-enabled mode, the delay will remain constant, but the number of instructions that could have been executed will change.

Use Poll Delay function in your lock to wait a given amount of guaranteed P1 frequency cycles, specified in the “clocks” variable.

Example 2-8. Dynamic Pause Loop Example

```
#include <x86intrin.h>
#include <stdint.h>

/* A useful predicate for dealing with timestamps that may wrap.
 Is a before b? Since the timestamps may wrap, this is asking whether it's
 shorter to go clockwise from a to b around the clock-face, or anti-clockwise.
 Times where going clockwise is less distance than going anti-clockwise
 are in the future, others are in the past. e.g. a = MAX-1, b = MAX+1 (=0),
 then a > b (true) does not mean a reached b; whereas signed(a) = -2,
 signed(b) = 0 captures the actual difference */

static inline bool before(uint64_t a, uint64_t b)
{
   return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
   uint64_t endTime = _rdtsc()+ clocks;

   for (; before(_rdtsc(), endTime); )
      _mm_pause();
}
```

For contended spinlocks of the form shown in the baseline example below, we recommend an exponential back off when the lock is found to be busy, as shown in the improved example, to avoid significant performance degradation that can be caused by conflicts between threads in the machine. This is more important as we increase the number of threads in the machine and make changes to the architecture that might aggravate these conflict conditions. In multi-socket Intel server processors with shared memory, conflicts across threads take much longer to resolve as the number of threads contending for the same lock increases. The exponential back off is designed to avoid these conflicts between the threads thus avoiding the potential performance degradation. Note that in the example below, the number of PAUSE instructions are increased by a factor of 2 until some MAX_BACKOFF is reached which is subject to tuning.

Example 2-9. Contended Locks with Increasing Back-off Example

```
/*******************/
/*Baseline Version */
/*******************/

// atomic {if (lock == free) then change lock state to busy}
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
      __asm__ ("pause");
   }
}
```

```
/*******************/
/*Improved Version */
/*******************/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
      for (int i=mask; i; --i){
         __asm__ ("pause");
      }
        mask = mask < max ? mask<<1 : max;
   }
}
```

2.7 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT TECHNOLOGY)

Intel® Hyper-Threading Technology (Intel® HT Technology) enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package, or within each processor core in a physical processor package. In its first implementation in the Intel® Xeon® processor, Intel HT Technology makes a single physical processor (or a processor core) appear as two or more logical processors.

Most Intel Architecture processor families support Intel HT Technology with two logical processors in each processor core, or in a physical processor in early implementations. The rest of this section describes features of the early implementation of Intel HT Technology. Most of the descriptions also apply to later implementations supporting two logical processors. The microarchitecture sections in this chapter provide additional details to individual microarchitecture and enhancements to Intel HT Technology.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an Intel HT Technology-capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, Intel HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

Figure 2-9 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting Intel HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an "add" operation from logical processor 0 and another "add" operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of Intel HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing Intel HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

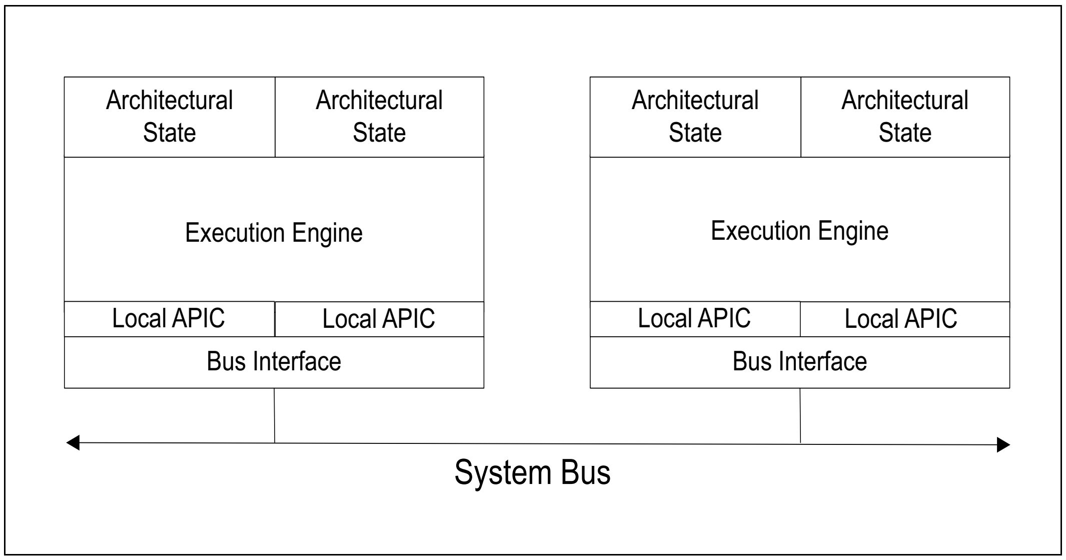


Figure 2-9. Intel® Hyper-Threading Technology on an SMP System

The performance potential due to Intel HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.7.1 Processor Resources and Intel® HT Technology

The majority of microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

2.7.1.1 Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D.

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the two-entry instruction streaming buffers) were replicated to reduce complexity.

2.7.1.2 Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include µop queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

2.7.1.3 Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

2.7.2 Microarchitecture Pipeline and Intel® HT Technology

This section describes the Intel HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage. Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.7.3 Execution Core

The core can dispatch up to six µops per cycle, provided the µops are ready to execute. Once the µops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned such that each uses half the entries.

2.7.4 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor needs to write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.8 SIMD TECHNOLOGY

SIMD computations (see Figure 2-10) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-11).

The Pentium III processor extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-11). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-10 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

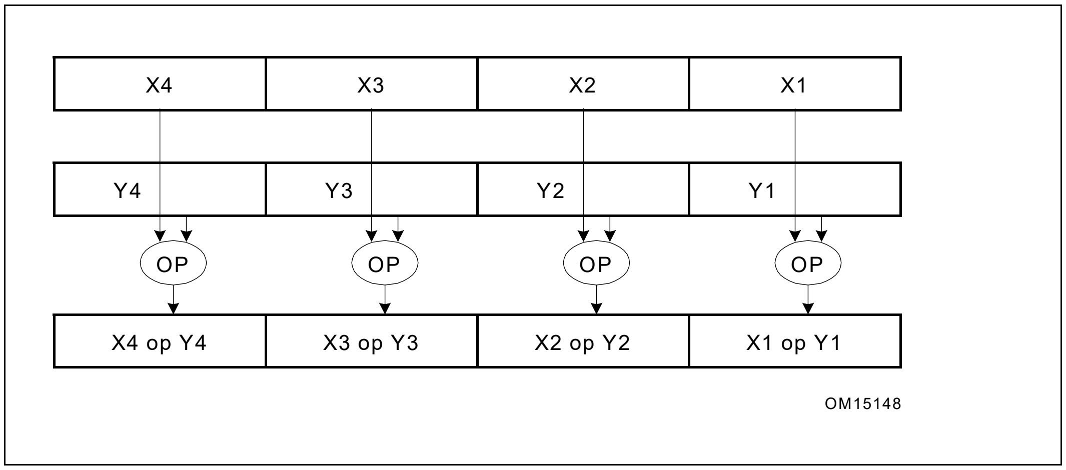


Figure 2-10. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-11.

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

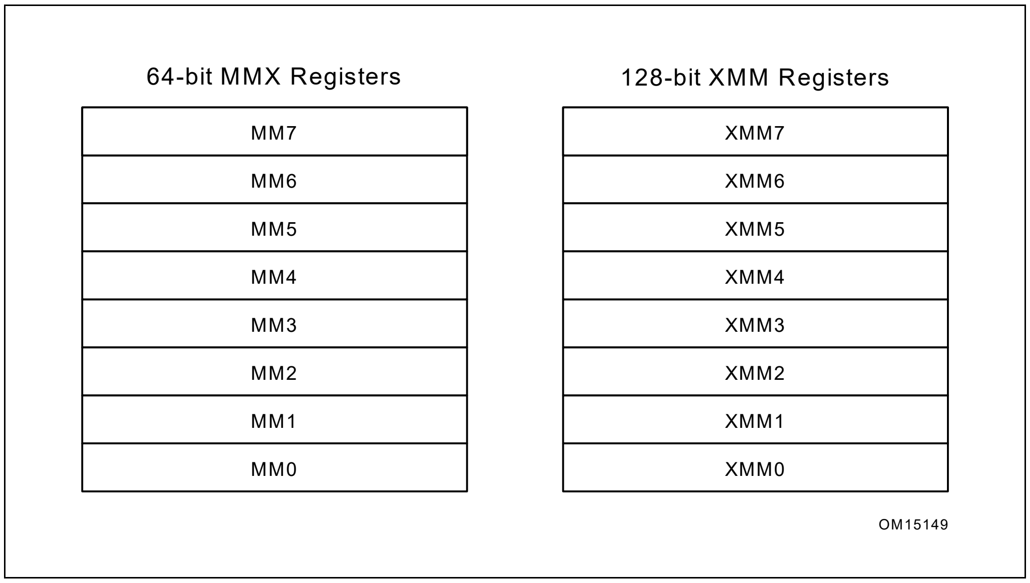


Figure 2-11. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.9 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1

- Chapter 9, "Programming with Intel® MMX™ Technology."
- Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."
- Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."
- Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI."
- Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2."
- Chapter 15, "Programming with Intel® AVX-512."
- Chapter 16, "Programming with Intel® Transactional Synchronization Extensions."

2.9.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

Recommendation: Integer SIMD code written using MMX instructions should consider more efficient implementations using SSE/Intel AVX instructions.

2.9.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.9.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.
- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.9.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).
- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.9.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.
- 6 instructions that evaluate the absolute values.
- 2 instructions that perform multiply and add operations and speed up the evaluation of dot products.
- 2 instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- 2 instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- 6 instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- 2 instructions that align data from the composite of two operands.

2.9.6 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction provides a streaming hint for WC loads.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations of word integers.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

2.9.7 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
  - — Basic and advanced string library functions from strlen, strcmp, to strcspn.
  - — Delimiter processing, token extraction for lexing of text streams.
  - — Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.9.8 AESNI and PCLMULQDQ

AESNI introduces seven new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES teration.

AES encryption involves processing 128-bit input data (plain text) through a finite number of iterative operation, referred to as "AES round", into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the "equivalent inverse cipher" instead of the "inverse cipher".

The cryptographic processing at each round involves two input data, one is the "state", the other is the "round key". Each round uses a different "round key". The round keys are derived from the cipher key using a "key schedule" algorithm. The "key schedule" algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.9.9 Intel® Advanced Vector Extensions (Intel® AVX)

Intel® Advanced Vector Extensions (Intel® AVX) offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

Intel AVX instruction set and 256-bit register state management detail are described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D. Optimization techniques for Intel AVX are discussed in Chapter 15, "Optimizations for Intel® AVX, Intel® AVX2, and Intel® FMA."

2.9.10 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.9.11 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.9.12 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.9.13 Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, Intel AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.9.14 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. The majority of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

There instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 and chapters 3, 4 and 5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

2.9.15 Intel® Transactional Synchronization Extensions (Intel® TSX)

The fourth generation Intel Core processor family introduces Intel® Transactional Synchronization Extensions (Intel® TSX), which aim to improve the performance of lock-protected critical sections of multithreaded applications while maintaining the lock-based programming model.

For background and details, see Chapter 16 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Software tuning recommendations for using Intel TSX on lock-protected critical sections of multithreaded applications are described in Chapter 16, "Intel® TSX Recommendations."

2.9.16 RDSEED

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.9.17 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics.

2. Updates to Chapter 3

Change bars and **violet** text show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Intel® 64 and IA-32 Processor Architectures.

Changes to this chapter:

- Corrected branding and style across chapter.
- Section 3.4
  - Removed section referring to the updates in Intel® Core Duo.
- Section 3.5
  - Updated Figure 3-1 to match style of those in Chapter 2
- Section 3.6:
  - 3.6.1: added- Bank conflicts may occur with the introduction due to the third load port in the Golden Cove microarchitecture. In this microarchitecture, conflicts happen between three loads with the same bits 2-5 of their linear address even if they access the same set of the cache. Up to two loads can access the same cache bank without a conflict; however, a third load accessing the same bank must be delayed. The bank conflicts do not apply to 512-bit wide loads because their bandwidth is limited to two per cycle.

    Recommendation: In the Golden Cove microarchitecture, bank conflicts often happen when multiple loads access the same memory location. Whenever possible, avoid reading the same memory location within a tight loop or using multiple load operations. Commonly used memory locations are better kept in the registers to prevent potential bank conflict penalty.
- Section 3.11:
  - Added Section: 3.11.5: False Sharing.

CHAPTER 3 GENERAL OPTIMIZATION GUIDELINES

This chapter discusses general optimization techniques that can improve the performance of applications running on Intel® processors. These techniques take advantage of microarchitectural features described in Chapter 2, "Intel® 64 and IA-32 Processor Architectures." Optimization guidelines focusing on Intel multi-core processors, Hyper-Threading Technology, and 64-bit mode applications are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (Intel® HT)," and Chapter 13, "64-bit Mode Coding Guidelines."

Practices that optimize performance focus on three areas:

- Tools and techniques for code generation.
- Analysis of the performance characteristics of the workload and its interaction with microarchitectural sub-systems.
- Tuning code to the target microarchitecture (or families of microarchitecture) to improve performance.

Some hints on using tools are summarized first to simplify the first two tasks. The rest of the chapter will focus on recommendations for code generation or code tuning to the target microarchitectures.

This chapter explains optimization techniques for the Intel® C++ Compiler, the Intel® Fortran Compiler, and other compilers.

3.1 PERFORMANCE TOOLS

Intel offers several tools to help optimize application performance, including compilers, performance analysis, and multithreading tools.

3.1.1 Intel® C++ and Fortran Compilers

Intel compilers support multiple operating systems (Windows*, Linux*, Mac OS*, and embedded). The Intel compilers optimize performance and give application developers access to advanced features, including:

- Flexibility to target 32-bit or 64-bit Intel processors for optimization.
- Compatibility with many integrated development environments or third-party compilers.
- Automatic optimization features to take advantage of the target processor's architecture.
- Automatic compiler optimization reduces the need to write different code for different processors.
- Common compiler features that are supported across Windows, Linux, and Mac OS include:
  - General optimization settings.
  - Cache-management features.
  - Interprocedural optimization (IPO) methods.
  - Profile-guided optimization (PGO) methods.
  - Multithreading support.
  - Floating-point arithmetic precision and consistency support.
  - Compiler optimization and vectorization reports.

3.1.2 General Compiler Recommendations

Generally speaking, a compiler tuned for a target microarchitecture can be expected to match or outperform hand-coding. However, if performance problems are noted with the compiled code, some compilers (like Intel C++ and Fortran compilers) allow the coder to insert intrinsics or inline assembly to exert control over generated code. If inline assembly is used, the user must verify that the code generated is high quality and yields good performance.

Default compiler switches are targeted for common cases. An optimization may be made to the compiler default if it benefits most programs. If the root cause of a performance problem is a poor choice on the part of the compiler, using different switches or compiling the targeted module with a different compiler may be the solution. See the "Quick Reference Guide to Optimization with Intel® C++ and Fortran Compilers" for additional suggestions on compiler Optimization Options, including processor-specific ones.

3.1.3 VTune™ Performance Analyzer

VTune uses performance monitoring hardware to collect statistics and coding information about your application and its interaction with the microarchitecture. This allows software engineers to measure performance characteristics of the workload for a given microarchitecture. VTune supports all current and past Intel processor families.

The VTune Performance Analyzer provides two kinds of feedback:

- Indication of a performance improvement gained by using a specific coding recommendation or microarchitectural feature.
- Information on whether a change in the program has improved or degraded performance with respect to a particular metric.

The VTune Performance Analyzer also provides measures for a number of workload characteristics, including:

- Retirement throughput of instruction execution as an indication of the degree of extractable instruction-level parallelism in the workload.
- Data traffic locality as an indication of the stress point of the cache and memory hierarchy.
- Data traffic parallelism as an indication of the degree of effectiveness of amortization of data access latency.

NOTE

Improving performance in one part of the machine does not necessarily bring significant gains to overall performance. It is possible to degrade overall performance by improving performance for some particular metric.

Where appropriate, coding recommendations in this chapter include descriptions of the VTune Performance Analyzer events that provide measurable data on the performance gain achieved by following the recommendations. For more on using the VTune analyzer, refer to the application's online help.

3.2 PROCESSOR PERSPECTIVES

Many coding recommendations work well across current microarchitectures. However, there are situations where a recommendation may benefit one microarchitecture more than another.

3.2.1 CPUID Dispatch Strategy and Compatible Code Strategy

When optimum performance on all processor generations is desired, applications can take advantage of the CPUID instruction to identify the processor generation and integrate processor-specific instructions into the source code. The Intel C++ Compiler supports the integration of different versions of the code for different target processors. The selection of which code to execute at runtime is made based on the CPU identifiers. Binary code targeted for different processor generations can be generated under the control of the programmer or by the compiler. Refer to the "Intel® C++ Compiler Classic Developer Guide and Reference" cpu_dispatch and cpu_specific sections for more information on CPU dispatching (a.k.a function multi-versioning).

For applications that target multiple generations of microarchitectures, and where minimum binary code size and single code path is important, a compatible code strategy is the best. Optimizing applications using techniques developed for the Intel Core microarchitecture combined with Nehalem microarchitecture are likely to improve code efficiency and scalability when running on processors based on current and future generations of Intel 64 and IA-32 processors.

3.2.2 Transparent Cache-Parameter Strategy

If the CPUID instruction supports function leaf 4, also known as deterministic cache parameter leaf, the leaf reports cache parameters for each level of the cache hierarchy in a deterministic and forward-compatible manner across Intel 64 and IA-32 processor families.

For coding techniques that rely on specific parameters of a cache level, using the deterministic cache parameter allows software to implement techniques in a way that is forward-compatible with future generations of Intel 64 and IA-32 processors, and cross-compatible with processors equipped with different cache sizes.

3.2.3 Threading Strategy and Hardware Multithreading Support

Intel 64 and IA-32 processor families offer hardware multithreading support in two forms: multi-core technology and HT Technology.

To fully harness the performance potential of hardware multithreading in current and future generations of Intel 64 and IA-32 processors, software must embrace a threaded approach in application design. At the same time, to address the widest range of installed machines, multithreaded software should be able to run without failure on a single processor without hardware multithreading support and should achieve performance on a single logical processor that is comparable to an unthreaded implementation (if such comparison can be made). This generally requires architecting a multithreaded application to minimize the overhead of thread synchronization. Additional guidelines on multithreading are discussed in Chapter 11, "Multicore and Intel® Hyper-Threading Technology (Intel® HT)."

3.3 CODING RULES, SUGGESTIONS, AND TUNING HINTS

This section includes rules, suggestions, and hints. They are targeted for engineers who are:

- Modifying source code to enhance performance (user/source rules).
- Writing assemblers or compilers (assembly/compiler rules).
- Doing detailed performance tuning (tuning suggestions).

Coding recommendations are ranked in importance using two measures:

- Local impact (high, medium, or low) refers to a recommendation's affect on the performance of a given instance of code.
- Generality (high, medium, or low) measures how often such instances occur across all application domains. Generality may also be thought of as "frequency."

These recommendations are approximate. They can vary depending on coding style, application domain, and other factors.

The purpose of the high, medium, and low (H, M, and L) priorities is to suggest the relative level of performance gain one can expect if a recommendation is implemented.

Because it is not possible to predict the frequency of a particular code instance in applications, priority hints cannot be directly correlated to application-level performance gain. In cases in which application-level performance gain has been observed, we have provided a quantitative characterization of the gain (for information only). In cases in which the impact has been deemed inapplicable, no priority is assigned.

3.4 OPTIMIZING THE FRONT END

Optimizing the front end covers two aspects:

- Maintaining steady supply of micro-ops to the execution engine — Mispredicted branches can disrupt streams of micro-ops, or cause the execution engine to waste execution resources on executing streams of micro-ops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit. Common techniques are covered in Section 3.4.1
- Supplying streams of micro-ops to utilize the execution bandwidth and retirement bandwidth as much as possible. In Sandy Bridge microarchitecture, this aspect focuses on keeping the hot code running from Decoded ICache. Techniques to maximize decode throughput for Intel microarchitecture are covered in Section 3.4.2

3.4.1 Branch Prediction Optimization

Branch optimizations have a significant impact on performance. By understanding the flow of branches and improving their predictability, you can increase the speed of code significantly.

Optimizations that help branch prediction are:

- It is critical to keep code and data on separate pages. See Section 3.6 for more information.
- Eliminate branches whenever possible.
- Arrange code to be consistent with the static branch prediction algorithm.
- Use the PAUSE instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase).
- Avoid putting multiple conditional branches in the same 8-byte aligned code block (i.e, have their last bytes' addresses within the same 8-byte aligned code) if the lower 6 bits of their target IPs are the same. This restriction has been removed in Ice Lake Client and later microarchitectures.

3.4.1.1 Eliminating Branches

Eliminating branches improves performance because:

- It reduces the possibility of mispredictions.
- It reduces the number of required branch target buffer (BTB) entries. Conditional branches that are never taken do not consume BTB resources.

There are four principal ways of eliminating branches:

- Arrange code to make basic blocks contiguous.
- Unroll loops, as discussed in Section 3.4.1.6
- Use the CMOV instruction.
- Use the SETCC instruction.

The following rules apply to branch elimination:

Assembly/Compiler Coding Rule 1. (MH impact, M generality) *Arrange code to make basic blocks contiguous and eliminate unnecessary branches.*

Assembly/Compiler Coding Rule 2. (M impact, ML generality) *Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.*

Consider a line of C code that has a condition dependent upon one of the constants:

```
X = (A < B) ? CONST1 : CONST2;
```

This code conditionally compares two values, A and B. If the condition is true, X is set to CONST1; otherwise it is set to CONST2. An assembly code sequence equivalent to the above C code can contain branches that are not predictable if there are no correlation in the two values.

Example 3-1 shows the assembly code with unpredictable branches. The unpredictable branches can be removed with the use of the SETCC instruction. Example 3-2 shows optimized code that has no branches.

Example 3-1. Assembly Code with an Unpredictable Branch

```
    cmp a, b                    ; Condition
    jbe L30                     ; Conditional branch
    mov ebx const1              ; ebx holds X
    jmp L31                     ; Unconditional branch
L30:
    mov ebx, const2
L31:
```

Example 3-2. Code Optimization to Eliminate Branches

```
xor   ebx, ebx      ; Clear ebx (X in the C code)
cmp   A, B
setge bl            ; When ebx = 0 or 1
                    ; OR the complement condition
sub   ebx, 1        ; ebx=11...11 or 00...00
and   ebx, CONST3; CONST3 = CONST1-CONST2
add   ebx, CONST2; ebx=CONST1 or CONST2
```

The optimized code in Example 3-2 sets EBX to zero, then compares A and B. If A is greater than or equal to B, EBX is set to one. Then EBX is decreased and AND'd with the difference of the constant values. This sets EBX to either zero or the difference of the values. By adding CONST2 back to EBX, the correct value is written to EBX. When CONST2 is equal to zero, the last instruction can be deleted.

Another way to remove branches is to use the CMOV and FCMOV instructions. Example 3-3 shows how to change a TEST and branch instruction sequence using CMOV to eliminate a branch. If the TEST sets the equal flag, the value in EBX will be moved to EAX. This branch is data-dependent, and is representative of an unpredictable branch.

Example 3-3. Eliminating Branch with CMOV Instruction

```
    test ecx, ecx
    jne  1H
    mov  eax, ebx


1H:
; To optimize code, combine jne and mov into one cmovcc instruction that checks the equal flag
    test     ecx, ecx              ; Test the flags
    cmoveq   eax, ebx              ; If the equal flag is set, move
                                   ; ebx to eax- the 1H: tag no longer needed
```

An extension to this concept can be seen in the AVX-512 masked operations, as well as in some instructions such as VPCMP which can be used to eliminate data dependent branches; see Section 18.4.

3.4.1.2 Static Prediction

Branches that do not have a history in the BTB (see Section 3.4.1) are predicted using a static prediction algorithm:

- Predict forward conditional branches to be NOT taken.
- Predict backward conditional branches to be taken.
- Predict indirect branches to be NOT taken.

The following rule applies to static prediction:

Assembly/Compiler Coding Rule 3. (M impact, H generality) *Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.*

Example 3-4 illustrates the static branch prediction algorithm. The body of an IF-THEN conditional is predicted.

Example 3-4. Static Branch Prediction Algorithm

```
//Forward condition branches not taken (fall through)
    IF<condition> {....
    ↓
    }

IF<condition> {...
    ↓
    }

//Backward conditional branches are taken
    LOOP {...
    ↑ — }<condition>

//Unconditional branches taken
     JMP
    ------→
```

Example 3-5 and Example 3-6 provide basic rules for a static prediction algorithm. In Example 3-5, the backward branch (JC BEGIN) is not in the BTB the first time through; therefore, the BTB does not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

Example 3-5. Static Taken Prediction

```
Begin:  mov     eax,  mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
```

The first branch instruction (JC BEGIN) in Example 3-6 is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through. The static prediction algorithm correctly predicts that the CALL CONVERT instruction will be taken, even before the branch has any branch history in the BTB.

Example 3-6. Static Not-Taken Prediction

```
        mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc      Begin
        mov     eax, 0
Begin:  call    Convert
```

The Intel Core microarchitecture does not use the static prediction heuristic. However, to maintain consistency across Intel 64 and IA-32 processors, software should maintain the static prediction heuristic as the default.

3.4.1.3 Inlining, Calls, and Returns

The return address stack mechanism augments the static and dynamic predictors to optimize specifically for calls and returns. It holds 16 entries, which is large enough to cover the call depth of most programs. If there is a chain of more than 16 nested calls and more than 16 returns in rapid succession, performance may degrade.

To enable the use of the return stack mechanism, calls and returns must be matched in pairs. If this is done, the likelihood of exceeding the stack depth in a manner that will impact performance is very low.

The following rules apply to inlining, calls, and returns:

Assembly/Compiler Coding Rule 4. (MH impact, MH generality) *Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns.*

Calls and returns are expensive; use inlining for the following reasons:

- Parameter passing overhead can be eliminated.
- In a compiler, inlining a function exposes more opportunity for optimization.
- If the inlined routine contains branches, the additional context of the caller may improve branch prediction within the routine.
- A mispredicted branch can lead to performance penalties inside a small function that are larger than those that would occur if that function is inlined.

Assembly/Compiler Coding Rule 5. (MH impact, MH generality) *Selectively inline a function if doing so decreases code size or if the function is small and the call site is frequently executed.*

Assembly/Compiler Coding Rule 6. (ML impact, ML generality) *If there are more than 16 nested calls and returns in rapid succession; consider transforming the program with inline to reduce the call depth.*

Assembly/Compiler Coding Rule 7. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred.*

Assembly/Compiler Coding Rule 8. (L impact, L generality) *If the last statement in a function is a call to another function, consider converting the call to a jump. This will save the call/return overhead as well as an entry in the return stack buffer.*

Assembly/Compiler Coding Rule 9. (M impact, L generality) *Do not put more than four branches in a 16-byte chunk.*

Assembly/Compiler Coding Rule 10. (M impact, L generality) *Do not put more than two end loop branches in a 16-byte chunk.*

3.4.1.4 Code Alignment

Careful arrangement of code can enhance cache and memory locality. Likely sequences of basic blocks should be laid out contiguously in memory. This may involve removing unlikely code, such as code to handle error conditions, from the sequence. See Section 3.7 on optimizing the instruction prefetcher.

Assembly/Compiler Coding Rule 11. (M impact, H generality) *When executing code from the Decoded ICache, direct branches that are mostly taken should have all their instruction bytes in a 64B cache line and nearer the end of that cache line. Their targets should be at or near the beginning of a 64B cache line.*

When executing code from the legacy decode pipeline, direct branches that are mostly taken should have all their instruction bytes in a 16B aligned chunk of memory and nearer the end of that 16B aligned chunk. Their targets should be at or near the beginning of a 16B aligned chunk of memory.

Assembly/Compiler Coding Rule 12. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, it should be placed on a different code page.*

3.4.1.5 Branch Type Selection

The default predicted target for indirect branches and calls is the fall-through path. Fall-through prediction is overridden if and when a hardware prediction is available for that branch. The predicted branch target from branch prediction hardware for an indirect branch is the previously executed branch target.

The default prediction to the fall-through path is only a significant issue if no branch prediction is available, due to poor code locality or pathological branch conflict problems. For indirect calls, predicting the fall-through path is usually not an issue, since execution will likely return to the instruction after the associated return.

Placing data immediately following an indirect branch can cause a performance problem. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations and this can cause resource conflicts and slow down branch recovery. Also, data immediately following indirect branches may appear as branches to the branch predication hardware, which can branch off to execute other data pages. This can lead to subsequent self-modifying code problems.

Assembly/Compiler Coding Rule 13. (M impact, L generality) *When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path.*

Indirect branches resulting from code constructs (such as switch statements, computed GOTOs or calls through pointers) can jump to an arbitrary number of locations. If the code sequence is such that the target destination of a branch goes to the same address most of the time, then the BTB will predict accurately most of the time. Since only one taken (non-fall-through) target can be stored in the BTB, indirect branches with multiple taken targets may have lower prediction rates.

The effective number of targets stored may be increased by introducing additional conditional branches. Adding a conditional branch to a target is fruitful if:

- The branch direction is correlated with the branch history leading up to that branch; that is, not just the last target, but how it got to this branch.
- The source/target pair is common enough to warrant using the extra branch prediction capacity. This may increase the number of overall branch mispredictions, while improving the misprediction of indirect branches. The profitability is lower if the number of mispredicting branches is very large.

User/Source Coding Rule 1. (M impact, L generality) *If an indirect branch has two or more common taken targets and at least one of those targets is correlated with branch history leading up to the branch, then convert the indirect branch to a tree where one or more indirect branches are preceded by conditional branches to those targets. Apply this "peeling" procedure to the common target of an indirect branch that correlates to branch history.*

The purpose of this rule is to reduce the total number of mispredictions by enhancing the predictability of branches (even at the expense of adding more branches). The added branches must be predictable for this to be worthwhile. One reason for such predictability is a strong correlation with preceding branch history. That is, the directions taken on preceding branches are a good indicator of the direction of the branch under consideration.

Example 3-7 shows a simple example of the correlation between a target of a preceding conditional branch and a target of an indirect branch.

Example 3-7. Indirect Branch With Two Favored Targets

```
function ()
{
int n = rand();             // random integer 0 to RAND_MAX
    if ( ! (n & 0x01) ) {   // n will be 0 half the times
        n = 0;              // updates branch history to predict taken
    }
    // indirect branches with multiple taken targets
    // may have lower prediction rates

 switch (n) {
    case 0: handle_0(); break;     // common target, correlated with
                                   // branch history that is forward taken
    case 1: handle_1(); break;     // uncommon
    case 3: handle_3(); break;     // uncommon
    default: handle_other();       // common target
     }
}
```

Correlation can be difficult to determine analytically, for a compiler and for an assembly language programmer. It may be fruitful to evaluate performance with and without peeling to get the best performance from a coding effort.

An example of peeling out the most favored target of an indirect branch with correlated branch history is shown in Example 3-8.

Example 3-8. A Peeling Technique to Reduce Indirect Branch Misprediction

```
function ()
{
   int n = rand();                                    // Random integer 0 to RAND_MAX
     if( ! (n & 0x01) ) THEN
            n = 0;                                    // n will be 0 half the times
     if (!n) THEN
           handle_0();                                // Peel out the most common target
                                                      // with correlated branch history

      {
        switch (n) {
           case 1: handle_1(); break;                 // Uncommon
           case 3: handle_3(); break;                 // Uncommon

           default: handle_other();                   // Make the favored target in
                                                      // the fall-through path
           }
     }
}
```

3.4.1.6 Loop Unrolling

Benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows one to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependence chain to expose the critical path.
- Unrolling exposes the code to various other optimizations, such as removal of redundant loads, common subexpression elimination, and so on.

The potential costs of unrolling loops are:

- Unrolling loops whose bodies contain branches increases demand on BTB capacity. If the number of iterations of the unrolled loop is 16 or fewer, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

Assembly/Compiler Coding Rule 14. (H impact, M generality) *Unroll small loops until the overhead of the branch and induction variable accounts (generally) for less than 10% of the execution time of the loop.*

Assembly/Compiler Coding Rule 15. (M impact, M generality) *Unroll loops that are frequently executed and have a predictable number of iterations to reduce the number of iterations to 16 or fewer. Do this unless it increases code size so that the working set no longer fits in the instruction cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches).*

Example 3-9 shows how unrolling enables other optimizations.

Example 3-9. Loop Unrolling

```
Before unrolling:
        do i = 1, 100
                if ( i mod 2 == 0 ) then a( i ) = x
                        else a( i ) = y
        enddo
After unrolling
        do i = 1, 100, 2
                a( i ) = y
                a( i+1 ) = x
        enddo
```

In this example, the loop that executes 100 times assigns X to every even-numbered element and Y to every odd-numbered element. By unrolling the loop you can make assignments more efficiently, removing one branch in the loop body.

3.4.2 Fetch and Decode Optimization

Intel Core microarchitecture provides several mechanisms to increase front end throughput. Techniques to take advantage of some of these features are discussed below.

3.4.2.1 Optimizing for Microfusion

An Instruction that operates on a register and a memory operand decodes into more micro-ops than its corresponding register-register version. Replacing the equivalent work of the former instruction using the register-register version usually require a sequence of two instructions. The latter sequence is likely to result in reduced fetch bandwidth.

Assembly/Compiler Coding Rule 16. (ML impact, M generality) *For improving fetch/decode throughput, Give preference to memory flavor of an instruction over the register-only flavor of the same instruction, if such instruction can benefit from micro-fusion.*

The following examples are some of the types of micro-fusions that can be handled by all decoders:

- All stores to memory, including store immediate. Stores execute internally as two separate micro-ops: store-address and store-data.
- All "read-modify" (load+op) instructions between register and memory, for example:
```
ADDPS   XMM9, OWORD PTR [RSP+40]
FADD    DOUBLE PTR [RDI+RSI*8]
XOR     RAX, QWORD PTR [RBP+32]
```
- All instructions of the form "load and jump," for example:
```
JMP     [RDI+200]
RET
```
- CMP and TEST with immediate operand and memory.

An Intel 64 instruction with RIP relative addressing is not micro-fused in the following cases:

- When an additional immediate is needed, for example:

```
CMP    [RIP+400], 27
MOV    [RIP+3000], 142
```

- When an RIP is needed for control flow purposes, for example:

```
JMP    [RIP+5000000]
```

In these cases, Intel Core microarchitecture and Sandy Bridge microarchitecture provide a 2 micro-op flow from decoder 0, resulting in a slight loss of decode bandwidth since 2 micro-op flow must be steered to decoder 0 from the decoder with which it was aligned.

RIP addressing may be common in accessing global data. Since it will not benefit from micro-fusion, compiler may consider accessing global data with other means of memory addressing.

3.4.2.2 Optimizing for Macrofusion

Macrofusion merges two instructions to a single micro-op. Intel Core microarchitecture performs this hardware optimization under limited circumstances.

The first instruction of the macro-fused pair must be a CMP or TEST instruction. This instruction can be REG-REG, REG-IMM, or a micro-fused REG-MEM comparison. The second instruction (adjacent in the instruction stream) should be a conditional branch.

Since these pairs are common ingredient in basic iterative programming sequences, macrofusion improves performance even on un-recompiled binaries. All of the decoders can decode one macro-fused pair per cycle, with up to three other instructions, resulting in a peak decode bandwidth of 5 instructions per cycle.

Each macro-fused instruction executes with a single dispatch. This process reduces latency, which in this case shows up as a cycle removed from branch mispredict penalty. Software also gain all other fusion benefits: increased rename and retire bandwidth, more storage for instructions in-flight, and power savings from representing more work in fewer bits.

The following list details when you can use macrofusion:

- CMP or TEST can be fused when comparing:

  REG-REG. For example: CMP EAX,ECX; JZ label
  REG-IMM. For example: CMP EAX,0x80; JZ label
  REG-MEM. For example: CMP EAX,[ECX]; JZ label
  MEM-REG. For example: CMP [EAX],ECX; JZ label

- TEST can fused with all conditional jumps.
- CMP can be fused with only the following conditional jumps in Intel Core microarchitecture. These conditional jumps check carry flag (CF) or zero flag (ZF). jump. The list of macrofusion-capable conditional jumps are:

  JA or JNBE
  JAE or JNB or JNC
  JE or JZ
  JNA or JBE
  JNAE or JC or JB
  JNE or JNZ

CMP and TEST can not be fused when comparing MEM-IMM (e.g. CMP [EAX],0x80; JZ label). Macrofusion is not supported in 64-bit mode for Intel Core microarchitecture.

- Nehalem microarchitecture supports the following enhancements in macrofusion:
  - CMP can be fused with the following conditional jumps (that was not supported in Intel Core microarchitecture):
    - JL or JNGE
    - JGE or JNL

    - JLE or JNG
    - JG or JNLE
  — Macrofusion is supported in 64-bit mode.
- Enhanced macrofusion support in Sandy Bridge microarchitecture is summarized in Table 3-1 with additional information in Example 3-14:

Table 3-1. Macro-Fusible Instructions in Sandy Bridge Microarchitecture

Instructions	TEST	AND	CMP	ADD	SUB	INC	DEC
JO/JNO	Y	Y	N	N	N	N	N
JC/JB/JAE/JNB	Y	Y	Y	Y	Y	N	N
JE/JZ/JNE/JNZ	Y	Y	Y	Y	Y	Y	Y
JNA/JBE/JA/JNBE	Y	Y	Y	Y	Y	N	N
JS/JNS/JP/JPE/JNP/JPO	Y	Y	N	N	N	N	N
JL/JNGE/JGE/JNL/JLE/JNG/JG/JNLE	Y	Y	Y	Y	Y	Y	Y

- Enhanced macrofusion support in Haswell microarchitecture is summarized in Table 3-2. Macrofusion is supported CMP/TEST/OP with reg-imm, reg-mem, and reg-reg addressing but not mem-imm addressing.

Table 3-2. Macro-Fusible Instructions in Haswell Microarchitecture

Opcode		JCC	ADD / SUB / CMP	INC / DEC	TEST / AND
70	0F 80	Jo	N	N	Y
71	0F 81	Jno	N	N	Y
72	0F 82	Jc / Jb	Y	N	Y
73	0F 83	Jae / Jnb	Y	N	Y
74	0F 84	Je / Jz	Y	Y	Y
75	0F 85	Jne / Jnz	Y	Y	Y
76	0F 86	Jna / Jbe	Y	N	Y
77	0F 87	Ja / Jnbe	Y	N	Y
78	0F 88	Js	N	N	Y
79	0F 89	Jns	N	N	Y
7A	0F 8A	Jp / Jpe	N	N	Y
7B	0F 8B	Jnp / Jpo	N	N	Y
7C	0F 8C	Jl / Jnge	Y	Y	Y
7D	0F 8D	Jge / Jnl	Y	Y	Y
7E	0F 8E	Jle / Jng	Y	Y	Y
7F	0F 8F	Jg / Jnle	Y	Y	Y

Assembly/Compiler Coding Rule 17. (M impact, ML generality) *Employ macrofusion where possible using instruction pairs that support macrofusion. Prefer TEST over CMP if possible. Use unsigned variables and unsigned jumps when possible. Try to logically verify that a variable is non-negative at the time of comparison. Avoid CMP or TEST of MEM-IMM flavor when possible. However, do not add other instructions to avoid using the MEM-IMM flavor.*

Example 3-10. Macrofusion, Unsigned Iteration Count

	Without Macrofusion	With Macrofusion

C code	for (int[1] i = 0; i < 1000; i++)
	    a++;
		for (unsigned int[2] i = 0; i < 1000; i++)
		    a++;

Disassembly	for (int i = 0; i < 1000; i++) mov dword ptr [i], 0 jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp dword ptr [i], 3E8H[3] jge End a++; mov eax, dword ptr [a] addqq eax,1 mov dword ptr [a], eax jmp Loop End:	for (unsigned int i = 0; i < 1000; i++) xor eax, eax mov dword ptr [i], eax jmp First Loop: mov eax, dword ptr [i] add eax, 1 mov dword ptr [i], eax First: cmp eax, 3E8H [4] jae End a++; mov eax, dword ptr [a] add eax, 1 mov dword ptr [a], eax jmp Loop End:

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.
4. CMP REG-IMM, JAE permits macrofusion.

Example 3-11. Macrofusion, If Statement

	Without Macrofusion	With Macrofusion

C code	int[1] a = 7;
	if (a < 77)
	    a++;
	else
	    a--;
		unsigned int[2] a = 7;
		if (a < 77)
		    a++;
		else
		    a--;

Disassembly	int a = 7; mov dword ptr [a], 7 if (a < 77) cmp dword ptr [a], 4DH [3] jge Dec	unsigned int a = 7; mov dword ptr [a], 7 if (a < 77) mov eax, dword ptr [a] cmp eax, 4DH jae Dec

Example 3-11. Macrofusion, If Statement (Contd.)

	Without Macrofusion	With Macrofusion
	a++; mov eax, dword ptr [a] add eax, 1 mov dword ptr [a], eax else jmp End a--; Dec: mov eax, dword ptr [a] sub eax, 1 mov dword ptr [a], eax End::	a++; add eax,1 mov dword ptr [a], eax else jmp End a--; Dec: sub eax, 1 mov dword ptr [a], eax End::

NOTES:

1. Signed iteration count inhibits macrofusion.
2. Unsigned iteration count is compatible with macrofusion.
3. CMP MEM-IMM, JGE inhibit macrofusion.

Assembly/Compiler Coding Rule 18. (M impact, ML generality) *Software can enable macro fusion when it can be logically determined that a variable is non-negative at the time of comparison; use TEST appropriately to enable macrofusion when comparing a variable with 0.*

Example 3-12. Macrofusion, Signed Variable

Without Macrofusion	With Macrofusion
test ecx, ecx jle OutSideTheIF cmp ecx, 64H jge OutSideTheIF <IF BLOCK CODE> OutSideTheIF:	test ecx, ecx jle OutSideTheIF cmp ecx, 64H jae OutSideTheIF <IF BLOCK CODE> OutSideTheIF:

For either signed or unsigned variable 'a'; "CMP a,0" and "TEST a,a" produce the same result as far as the flags are concerned. Since TEST can be macro-fused more often, software can use "TEST a,a" to replace "CMP a,0" for the purpose of enabling macrofusion.

Example 3-13. Macrofusion, Signed Comparison

C Code	Without Macrofusion	With Macrofusion
if (a == 0)	cmp a, 0 jne lbl ... lbl:	test a, a jne lbl ... lbl:
if (a >= 0)	cmp a, 0 jl lbl; ... lbl:	test a, a jl lbl ... lbl:

Sandy Bridge microarchitecture enables more arithmetic and logic instructions to macro-fuse with conditional branches. In loops where the ALU ports are already congested, performing one of these macrofusions can relieve the pressure, as the macro-fused instruction consumes only port 5, instead of an ALU port plus port 5.

In Example 3-14, the "add/cmp/jnz" loop contains two ALU instructions that can be dispatched via either port 0, 1, 5. So there is higher probability of port 5 might bind to either ALU instruction causing JNZ to wait a cycle. The “sub/jnz” loop, the likelihood of ADD/SUB/JNZ can be dispatched in the same cycle is increased because only SUB is free to bind with either port 0, 1, 5.

Example 3-14. Additional Macrofusion Benefit in Sandy Bridge Microarchitecture

Add + cmp + jnz alternative	Loop control with sub + jnz
lea rdx, buff xor rcx, rcx xor eax, eax loop: add eax, [rdx + 4 * rcx] add rcx, 1 cmp rcx, LEN jnz loop	lea rdx, buff - 4 xor rcx, LEN xor eax, eax loop: add eax, [rdx + 4 * rcx] sub rcx, 1 jnz loop

3.4.2.3 Length-Changing Prefixes (LCP)

The length of an instruction can be up to 15 bytes in length. Some prefixes can dynamically change the length of an instruction that the decoder must recognize. Typically, the pre-decode unit will estimate the length of an instruction in the byte stream assuming the absence of LCP. When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle. Normal queuing throughout of the machine pipeline generally cannot hide LCP penalties.

The prefixes that can dynamically change the length of a instruction include:

- Operand size prefix (0x66).
- Address size prefix (0x67).

The instruction MOV DX, 01234h is subject to LCP stalls in processors based on Intel Core microarchitecture, and in Intel Core Duo and Intel Core Solo processors. Instructions that contain imm16 as part of their fixed encoding but do not require LCP to change the immediate size are not subject to LCP stalls. The REX prefix (4xh) in 64-bit mode can change the size of two classes of instruction, but does not cause an LCP penalty.

If the LCP stall happens in a tight loop, it can cause significant performance degradation. When decoding is not a bottleneck, as in floating-point heavy code, isolated LCP stalls usually do not cause performance degradation.

Assembly/Compiler Coding Rule 19. (MH impact, MH generality) *Favor generating code using imm8 or imm32 values instead of imm16 values.*

If imm16 is needed, load equivalent imm32 into a register and use the word value in the register instead.

Double LCP Stalls

Instructions that are subject to LCP stalls and cross a 16-byte fetch line boundary can cause the LCP stall to trigger twice. The following alignment situations can cause LCP stalls to trigger twice:

- An instruction is encoded with a MODR/M and SIB byte, and the fetch line boundary crossing is between the MODR/M and the SIB bytes.
- An instruction starts at offset 13 of a fetch line references a memory location using register and immediate byte offset addressing mode.

The first stall is for the 1st fetch line, and the 2nd stall is for the 2nd fetch line. A double LCP stall causes a decode penalty of 11 cycles.

The following examples cause LCP stall once, regardless of their fetch-line location of the first byte of the instruction:

```
ADD DX, 01234H
ADD word ptr [EDX], 01234H
ADD word ptr 012345678H[EDX], 01234H
ADD word ptr [012345678H], 01234H
```

The following instructions cause a double LCP stall when starting at offset 13 of a fetch line:

```
ADD word ptr [EDX+ESI], 01234H
ADD word ptr 012H[EDX], 01234H
ADD word ptr 012345678H[EDX+ESI], 01234H
```

To avoid double LCP stalls, do not use instructions subject to LCP stalls that use SIB byte encoding or addressing mode with byte displacement.

False LCP Stalls

False LCP stalls have the same characteristics as LCP stalls, but occur on instructions that do not have any imm16 value.

False LCP stalls occur when (a) instructions with LCP that are encoded using the F7 opcodes, and (b) are located at offset 14 of a fetch line. These instructions are: not, neg, div, idiv, mul, and imul. False LCP experiences delay because the instruction length decoder can not determine the length of the instruction before the next fetch line, which holds the exact opcode of the instruction in its MODR/M byte.

The following techniques can help avoid false LCP stalls:

- Upcast all short operations from the F7 group of instructions to long, using the full 32 bit version.
- Ensure that the F7 opcode never starts at offset 14 of a fetch line.

Assembly/Compiler Coding Rule 20. (M impact, ML generality) *Ensure instructions using 0xF7 opcode byte does not start at offset 14 of a fetch line; and avoid using these instruction to operate on 16-bit data, upcast short data to 32 bits.*

Example 3-15. Avoiding False LCP Delays with 0xF7 Group Instructions

A Sequence Causing Delay in the Decoder	Alternate Sequence to Avoid Delay
neg word ptr a	movsx eax, word ptr a neg eax mov word ptr a, AX

3.4.2.4 Optimizing the Loop Stream Detector (LSD)

The LSD detects loops that have many iterations and fit into the µop-queue. The µop-queue streams the loop until a branch miss-prediction inevitably ends it.

LSD improves fetch bandwidth. In single thread mode, it saves power by allowing the front-end to sleep. In multi-thread mode, front-resource can better serve the other thread.

Loops qualify for LSD replay if all the following conditions are met:

- Loop body size up to 60 µops, with up to 15 taken branches, and up to 15 64-byte fetch lines.
- No CALL or RET.
- No mismatched stack operations (e.g., more PUSH than POP).
- More than ~20 iterations.

Many calculation-intensive loops, searches, and software string moves match these characteristics. These loops exceed the BPU prediction capacity and always terminate in a branch misprediction.

Assembly/Compiler Coding Rule 21. (MH impact, MH generality) *Break up a loop body with a long sequence of instructions into loops of shorter instruction blocks of no more than the size of the LSD.*

Allocation bandwidth in Ice Lake Client microarchitecture increased from 4 µops per cycle to 5 µops per cycle.

Assume a loop that qualifies for LSD has 23 µops in the loop body. The hardware unrolls the loop such that it still fits into the µop-queue, in this case twice. The loop in the µop-queue thus takes 46 µops.

The loop is sent to allocation 5 µops per cycle. After 45 out of the 46 µops are sent, in the next cycle only a single µop is sent, which means that in that cycle, 4 of the allocation slots are wasted. This pattern repeats itself, until the loop is exited by a misprediction. Hardware loop unrolling minimizes the number of wasted slots during LSD.

3.4.2.5 Optimization for Decoded ICache

The decoded ICache is a new feature in Sandy Bridge microarchitecture. Running the code from the Decoded ICache has two advantages:

- Higher bandwidth of micro-ops feeding the out-of-order engine.
- The front end does not need to decode the code that is in the Decoded ICache; this saves power.

There is overhead in switching between the Decoded ICache and the legacy decode pipeline. If your code switches frequently between the front end and the Decoded ICache, the penalty may be higher than running only from the legacy pipeline.

To ensure "hot" code is feeding from the decoded ICache:

- Make sure each hot code block is less than about 750 instructions. Specifically, do not unroll to more than 750 instructions in a loop. This should enable Decoded ICache residency even when hyper-threading is enabled.
- For applications with very large blocks of calculations inside a loop, consider loop-fission: split the loop into multiple loops that fit in the Decoded ICache, rather than a single loop that overflows.
- If an application can be sure to run with only one thread per core, it can increase hot code block size to about 1500 instructions.

Dense Read-Modify-Write Code

The Decoded ICache can hold only up to 18 micro-ops per each 32 byte aligned memory chunk. Therefore, code with a high concentration of instructions that are encoded in a small number of bytes, yet have many micro-ops, may overflow the 18 micro-op limitation and not enter the Decoded ICache. Read-modify-write (RMW) instructions are a good example of such instructions.

RMW instructions accept one memory source operand, one register source operand, and use the source memory operand as the destination. The same functionality can be achieved by two or three instructions: the first reads the memory source operand, the second performs the operation with the second register source operand, and the last writes the result back to memory. These instructions usually result in the same number of micro-ops but use more bytes to encode the same functionality.

One case where RMW instructions may be used extensively is when the compiler optimizes aggressively for code size.

Here are some possible solutions to fit the hot code in the Decoded ICache:

- Replace RMW instructions with two or three instructions that have the same functionality. For example, "adc [rdi], rcx" is only three bytes long; the equivalent sequence "adc rax, [rdi]" + "mov [rdi], rax" has a footprint of six bytes.
- Align the code so that the dense part is broken down among two different 32-byte chunks. This solution is useful when using a tool that aligns code automatically, and is indifferent to code changes.
- Spread the code by adding multiple byte NOPs in the loop. Note that this solution adds micro-ops for execution.

Align Unconditional Branches for Decoded ICache

For code entering the Decoded ICache, each unconditional branch is the last micro-op occupying a Decoded ICache Way. Therefore, only three unconditional branches per a 32 byte aligned chunk can enter the Decoded ICache.

Unconditional branches are frequent in jump tables and switch declarations. Below are examples for these constructs, and methods for writing them so that they fit in the Decoded ICache.

Compilers create jump tables for C++ virtual class methods or DLL dispatch tables. Each unconditional branch consumes five bytes; therefore up to seven of them can be associated with a 32-byte chunk. Thus jump tables may not fit in the Decoded ICache if the unconditional branches are too dense in each 32Byte-aligned chunk. This can cause performance degradation for code executing before and after the branch table.

The solution is to add multi-byte NOP instructions among the branches in the branch table. This may increases code size and should be used cautiously. However, these NOPs are not executed and therefore have no penalty in later pipe stages.

Switch-Case constructs represents a similar situation. Each evaluation of a case condition results in an unconditional branch. The same solution of using multi-byte NOP can apply for every three consecutive unconditional branches that fits inside an aligned 32-byte chunk.

Two Branches in a Decoded ICache Way

The Decoded ICache can hold up to two branches in a way. Dense branches in a 32 byte aligned chunk, or their ordering with other instructions may prohibit all the micro-ops of the instructions in the chunk from entering the Decoded ICache. This does not happen often. When it does happen, you can space the code with NOP instructions where appropriate. Make sure that these NOP instructions are not part of hot code.

Assembly/Compiler Coding Rule 22. (M impact, M generality) *Avoid putting explicit references to* ESP *in a sequence of stack operations (*POP, PUSH, CALL, RET*).*

3.4.2.6 Other Decoding Guidelines

Assembly/Compiler Coding Rule 23. (ML impact, L generality) *Use simple instructions that are less than eight bytes in length.*

Assembly/Compiler Coding Rule 24. (M impact, MH generality) *Avoid using prefixes to change the size of immediate and displacement.*

Long instructions (more than seven bytes) may limit the number of decoded instructions per cycle. Each prefix adds one byte to the length of instruction, possibly limiting the decoder's throughput. In addition, multiple prefixes can only be decoded by the first decoder. These prefixes also incur a delay when decoded. If multiple prefixes or a prefix that changes the size of an immediate or displacement cannot be avoided, schedule them behind instructions that stall the pipe for some other reason.

3.5 OPTIMIZING THE EXECUTION CORE

The superscalar, out-of-order execution core(s) in recent generations of microarchitectures contain multiple execution hardware resources that can execute multiple micro-ops in parallel. These resources generally ensure that micro-ops execute efficiently and proceed with fixed latencies. General guidelines to make use of the available parallelism are:

- Follow the rules (see Section 3.4) to maximize useful decode bandwidth and front end throughput. These rules include favoring single micro-op instructions and taking advantage of micro-fusion, Stack pointer tracker and macrofusion.
- Maximize rename bandwidth. Guidelines are discussed in this section and include properly dealing with partial registers, ROB read ports and instructions which causes side-effects on flags.
- Scheduling recommendations on sequences of instructions so that multiple dependency chains are alive in the reservation station (RS) simultaneously, thus ensuring that your code utilizes maximum parallelism.

- Avoid hazards, minimize delays that may occur in the execution core, allowing the dispatched micro-ops to make progress and be ready for retirement quickly.

3.5.1 Instruction Selection

Some execution units are not pipelined, this means that micro-ops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle.

It is generally a good starting point to select instructions by considering the number of micro-ops associated with each instruction, favoring in the order of: single micro-op instructions, simple instruction with less than 4 micro-ops, and last instruction requiring microsequencer ROM (micro-ops which are executed out of the microsequencer involve extra overhead).

Assembly/Compiler Coding Rule 25. (M impact, H generality) *Favor single-micro-operation instructions. Also favor instruction with shorter latencies.*

A compiler may be already doing a good job on instruction selection. If so, user intervention usually is not necessary.

Assembly/Compiler Coding Rule 26. (M impact, L generality) *Avoid prefixes, especially multiple non-0F-prefixed opcodes.*

Assembly/Compiler Coding Rule 27. (M impact, L generality) *Do not use many segment registers.*

Assembly/Compiler Coding Rule 28. (M impact, M generality) *Avoid using complex instructions (for example, enter, leave, or loop) that have more than four µops and require multiple cycles to decode. Use sequences of simple instructions instead.*

Assembly/Compiler Coding Rule 29. (MH impact, M generality) *Use push/pop to manage stack space and address adjustments between function calls/returns instead of enter/leave. Using enter instruction with non-zero immediates can experience significant delays in the pipeline in addition to misprediction.*

Theoretically, arranging instructions sequence to match the 4-1-1-1 template applies to processors based on Intel Core microarchitecture. However, with macrofusion and micro-fusion capabilities in the front end, attempts to schedule instruction sequences using the 4-1-1-1 template will likely provide diminishing returns.

Instead, software should follow these additional decoder guidelines:

- If you need to use multiple micro-op, non-microsequenced instructions, try to separate by a few single micro-op instructions. The following instructions are examples of multiple micro-op instruction not requiring micro-sequencer:

  ```
  ADC/SBB
  CMOVcc
  Read-modify-write instructions
  ```

- If a series of multiple micro-op instructions cannot be separated, try breaking the series into a different equivalent instruction sequence. For example, a series of read-modify-write instructions may go faster if sequenced as a series of read-modify + store instructions. This strategy could improve performance even if the new code sequence is larger than the original one.

3.5.1.1 Integer Divide

Typically, an integer divide is preceded by a CWD or CDQ instruction. Depending on the operand size, divide instructions use DX:AX or EDX:EAX for the dividend. The CWD or CDQ instructions sign-extend AX or EAX into DX or EDX, respectively. These instructions have denser encoding than a shift and move would be, but they generate the same number of micro-ops. If AX or EAX is known to be positive, replace these instructions with:

```
xor dx, dx
```

or

```
xor edx, edx
```

Modern compilers typically can transform high-level language expression involving integer division where the divisor is a known integer constant at compile time into a faster sequence using IMUL instruction instead. Thus programmers should minimize integer division expression with divisor whose value can not be known at compile time.

Alternately, if certain known divisor value are favored over other unknown ranges, software may consider isolating the few favored, known divisor value into constant-divisor expressions.

Section 13.2.4 describes more detail of using MUL/IMUL to replace integer divisions.

3.5.1.2 Using LEA

In Sandy Bridge microarchitecture, there are two significant changes to the performance characteristics of LEA instruction:

- LEA can be dispatched via port 1 and 5 in most cases, doubling the throughput over prior generations. However this apply only to LEA instructions with one or two source operands.

Example 3-16. Independent Two-Operand LEA Example

```
    mov     edx, N
    mov     eax, X
    mov     ecx, Y


loop:
    lea     ecx, [ecx + ecx]        // ecx = ecx*2
    lea     eax, [eax + eax *4]     // eax = eax*5
    and     ecx, 0xff
    and     eax, 0xff
    dec     edx
    jg      loop
```

- For LEA instructions with three source operands and some specific situations, instruction latency has increased to 3 cycles, and must dispatch via port 1:
  — LEA that has all three source operands: base, index, and offset.
  — LEA that uses base and index registers where the base is EBP, RBP, or R13.
  — LEA that uses RIP relative addressing mode.
  — LEA that uses 16-bit addressing mode.

Example 3-17. Alternative to Three-Operand LEA

3 operand LEA is slower	Two-operand LEA alternative	Alternative 2

#define K 1
uint32 an = 0;
uint32 N= mi_N;
mov ecx, N
xor esi, esi;
xor edx, edx;
cmp ecx, 2;
jb finished;
dec ecx;
 	
loop1:
  mov edi, esi;
  lea esi, [K+esi+edx];
  and esi, 0xFF;
  mov edx, edi;
  dec ecx;
  jnz loop1;
finished:
  mov [an] ,esi;
	#define K 1
	uint32 an = 0;
	uint32 N= mi_N;
	mov ecx, N
	xor esi, esi;
	xor edx, edx;
	cmp ecx, 2;
	jb finished;
	dec ecx;
	
	loop1:
	  mov edi, esi;
	  lea esi, [K+edx];
	lea esi, [esi+edx];
	and esi, 0xFF;
	  mov edx, edi;
	  dec ecx;
	  jnz loop1;
	finished:
	  mov [an] ,esi;
		#define K 1
		uint32 an = 0;
		uint32 N= mi_N;
		mov ecx, N
		xor esi, esi;
		mov edx, K;
		cmp ecx, 2;
		jb finished;
		mov eax, 2
		dec ecx;
		loop1:
		  mov edi, esi;
		  lea esi, [esi+edx];
		  and esi, 0xFF;
		  lea edx, [edi +K];
		  dec ecx;
		  jnz loop1;
		finished:
		  mov [an] ,esi;

The LEA instruction or a sequence of LEA, ADD, SUB and SHIFT instructions can replace constant multiply instructions. The LEA instruction can also be used as a multiple operand addition instruction, for example:

```
LEA ECX, [EAX + EBX*4 + A]
```

Using LEA in this way may avoid register usage by not tying up registers for operands of arithmetic instructions. This use may also save code space.

If the LEA instruction uses a shift by a constant amount then the latency of the sequence of µops is shorter if adds are used instead of a shift, and the LEA instruction may be replaced with an appropriate sequence of µops. This, however, increases the total number of µops, leading to a trade-off.

Assembly/Compiler Coding Rule 30. (ML impact, L generality) *If an LEA instruction using the scaled index is on the critical path, a sequence with ADDs may be better.*

3.5.1.3 ADC and SBB in Sandy Bridge Microarchitecture

The throughput of ADC and SBB in Sandy Bridge microarchitecture is 1 cycle, compared to 1.5-2 cycles in the prior generation. These two instructions are useful in numeric handling of integer data types that are wider than the maximum width of native hardware.

Example 3-18. Examples of 512-bit Additions

```
//Add 64-bit to 512 Number
    lea         rsi, gLongCounter
    lea         rdi, gStepValue
    mov         rax, [rdi]
    xor         rcx, rcx
loop_start:
    mov         r10, [rsi+rcx]
    add         r10, rax
    mov         [rsi+rcx], r10

    mov         r10, [rsi+rcx+8]
    adc         r10, 0
    mov         [rsi+rcx+8], r10


    mov         r10, [rsi+rcx+16]
    adc         r10, 0
    mov         [rsi+rcx+16], r10
    mov         r10, [rsi+rcx+24]
    adc         r10, 0
    mov         [rsi+rcx+24], r10

    mov         r10, [rsi+rcx+32]
    adc         r10, 0
    mov         [rsi+rcx+32], r10
    mov        r10, [rsi+rcx+40]
    adc        r10, 0
    mov        [rsi+rcx+40], r10



mov     r10, [rsi+rcx+48]
 adc    r10, 0
 mov    [rsi+rcx+48], r10

 mov    r10, [rsi+rcx+56]
 adc    r10, 0
 mov    [rsi+rcx+56], r10
 add    rcx, 64
 cmp    rcx, SIZE
 jnz    loop_start
```

```
// 512-bit Addition
loop1:
    mov          rax, [StepValue]
    add         rax, [LongCounter]
    mov         LongCounter, rax
    mov         rax, [StepValue+8]
    adc         rax, [LongCounter+8]
    mov         LongCounter+8, rax
    mov         rax, [StepValue+16]
    adc         rax, [LongCounter+16]




    mov          LongCounter+16, rax
    mov         rax, [StepValue+24]
    adc         rax, [LongCounter+24]

    mov         LongCounter+24, rax
    mov         rax, [StepValue+32]
    adc         rax, [LongCounter+32]

    mov         LongCounter+32, rax
    mov         rax, [StepValue+40]
    adc         rax, [LongCounter+40]

    mov         LongCounter+40, rax
    mov         rax, [StepValue+48]
    adc         rax, [LongCounter+48]



    mov         LongCounter+48, rax
    mov         rax, [StepValue+56]
    adc         rax, [LongCounter+56]

    mov         LongCounter+56, rax
    dec         rcx
    jnz         loop1
```

3.5.1.4 Bitwise Rotation

Bitwise rotation can choose between rotate with count specified in the CL register, an immediate constant and by 1 bit. Generally, The rotate by immediate and rotate by register instructions are slower than rotate by 1 bit. The rotate by 1 instruction has the same latency as a shift.

Assembly/Compiler Coding Rule 31. (ML impact, L generality) *Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction.*

In Sandy Bridge microarchitecture, ROL/ROR by immediate has 1-cycle throughput, SHLD/SHRD using the same register as source and destination by an immediate constant has 1-cycle latency with 0.5 cycle throughput. The "ROL/ROR reg, imm8" instruction has two micro-ops with the latency of 1-cycle for the rotate register result and 2-cycles for the flags, if used.

In Ivy Bridge microarchitecture, The "ROL/ROR reg, imm8" instruction with immediate greater than 1, is one micro-op with one-cycle latency when the overflow flag result is used. When the immediate is one, dependency on the overflow flag result of ROL/ROR by a subsequent instruction will see the ROL/ROR instruction with two-cycle latency.

3.5.1.5 Variable Bit Count Rotation and Shift

In Sandy Bridge microarchitecture, The "ROL/ROR/SHL/SHR reg, cl" instruction has three micro-ops. When the flag result is not needed, one of these micro-ops may be discarded, providing better performance in many common usages. When these instructions update partial flag results that are subsequently used, the full three micro-ops flow must go through the execution and retirement pipeline, experiencing slower performance. In Ivy Bridge microarchitecture, executing the full three micro-ops flow to use the updated partial flag result has additional delay. Consider the looped sequence below:

```
loop:
        shl eax, cl
        add ebx, eax
        dec edx ; DEC does not update carry, causing SHL to execute slower three micro-ops flow
        jnz loop
```

The DEC instruction does not modify the carry flag. Consequently, the SHL EAX, CL instruction needs to execute the three micro-ops flow in subsequent iterations. The SUB instruction will update all flags. So replacing DEC with SUB will allow SHL EAX, CL to execute the two micro-ops flow.

3.5.1.6 Address Calculations

For computing addresses, use the addressing modes rather than general-purpose computations. Internally, memory reference instructions can have four operands:

- Relocatable load-time constant.
- Immediate constant.
- Base register.
- Scaled index register.

Note that the latency and throughput of LEA with more than two operands are slower in Sandy Bridge microarchitecture (see Section 3.5.1.2). Addressing modes that uses both base and index registers will consume more read port resource in the execution engine and may experience more stalls due to availability of read port resources. Software should take care by selecting the speedy version of address calculation.

In the segmented model, a segment register may constitute an additional operand in the linear address calculation. In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

3.5.1.7 Clearing Registers and Dependency Breaking Idioms

Code sequences that modifies partial register can experience some delay in its dependency chain, but can be avoided by using dependency breaking idioms.

In processors based on Intel Core microarchitecture, a number of instructions can help clear execution dependency when software uses these instruction to clear register content to zero. The instructions include:

```
XOR REG, REG
SUB REG, REG
XORPS/PD XMMREG, XMMREG
PXOR XMMREG, XMMREG
SUBPS/PD XMMREG, XMMREG
PSUBB/W/D/Q XMMREG, XMMREG
```

In processors based on Sandy Bridge microarchitecture, the instruction listed above plus equivalent AVX counter parts are also zero idioms that can be used to break dependency chains. Furthermore, they do not consume an issue port or an execution unit. So using zero idioms are preferable than moving 0's into the register. The AVX equivalent zero idioms are:

```
VXORPS/PD XMMREG, XMMREG
VXORPS/PD YMMREG, YMMREG
VPXOR XMMREG, XMMREG
VSUBPS/PD XMMREG, XMMREG
VSUBPS/PD YMMREG, YMMREG
VPSUBB/W/D/Q XMMREG, XMMREG
```

Microarchitectures that support Intel AVX-512 have the equivalent of zero idioms for the 512-bit registers using the unmasked versions of the instructions:

```
VXORPS/PD ZMMREG, ZMMREG
VPXOR ZMMREG, ZMMREG
VSUBPS/PD ZMMREG, ZMMREG
VPSUBB/W/D/Q ZMMREG, ZMMREG
```

The XOR and SUB instructions can be used to clear execution dependencies on the zero evaluation of the destination register.

Assembly/Compiler Coding Rule 32. (M impact, ML generality) *Use dependency-breaking-idiom instructions to set a register to* 0, *or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move* 0 *into the register instead. This requires more code space than using* XOR *and* SUB*, but avoids setting the condition codes.*

Example 3-19 of using pxor to break dependency idiom on a XMM register when performing negation on the elements of an array.

```
int a[4096], b[4096], c[4096];
For ( int i = 0; i < 4096; i++ )
        C[i] = - ( a[i] + b[i] );
```

Example 3-19. Clearing Register to Break Dependency While Negating Array Elements

Negation (-x = (x XOR (-1)) - (-1) without breaking dependency	Negation (-x = 0 -x) using PXOR reg, reg breaks dependency

```
lea      eax, a
lea      ecx, b
lea      edi, c
xor      edx, edx
movdqa   xmm7, allone
lp:

movdqa   xmm0, [eax + edx]
paddd    xmm0, [ecx + edx]
pxor     xmm0, xmm7
psubd    xmm0, xmm7
movdqa   [edi + edx], xmm0
add      edx, 16
cmp      edx, 4096
jl       lp
```
	```
	lea   eax, a
	lea   ecx, b
	lea   edi, c
	xor   edx, edx
	lp:

	movdqa   xmm0, [eax + edx]
	paddd    xmm0, [ecx + edx]
	pxor     xmm7, xmm7
	psubd    xmm7, xmm0
	movdqa   [edi + edx], xmm7
	add      edx,16
	cmp      edx, 4096
	jl       lp
	```

Assembly/Compiler Coding Rule 33. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using MOVZX.*

Sometimes sign-extended semantics can be maintained by zero-extending operands. For example, the C code in the following statements does not need sign extension, nor does it need prefixes for operand size overrides:

```
static short INT a, b;
IF (a == b) {
  . . .
}
```

Code for comparing these 16-bit operands might be:

```
MOVZW  EAX, [a]
MOVZW  EBX, [b]
CMP    EAX, EBX
```

These circumstances tend to be common. However, the technique will not work if the compare is for greater than, less than, greater than or equal, and so on, or if the values in eax or ebx are to be used in another operation where sign extension is required.

Assembly/Compiler Coding Rule 34. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.*

The trace cache can be packed more tightly when instructions with operands that can only be represented as 32 bits are not adjacent.

Assembly/Compiler Coding Rule 35. (ML impact, L generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as sign-extended 16-bit immediates near each other. Try to schedule µops that have no immediate immediately before or after µops with 32-bit immediates.*

3.5.1.8 Compares

Use TEST when comparing a value in a register with zero. TEST essentially ANDs operands together without writing to a destination register. TEST is preferred over AND because AND produces an extra result register. TEST is better than CMP ..., 0 because the instruction size is smaller.

Use TEST when comparing the result of a logical AND with an immediate constant for equality or inequality if the register is EAX for cases such as:

```
IF (AVAR & 8) { }
```

The TEST instruction can also be used to detect rollover of modulo of a power of 2. For example, the C code:

```
IF ( (AVAR % 16) == 0 ) { }
```

can be implemented using:

```
TEST    EAX, 0x0F
JNZ     AfterIf
```

Using the TEST instruction between the instruction that may modify part of the flag register and the instruction that uses the flag register can also help prevent partial flag register stall.

Assembly/Compiler Coding Rule 36. (ML impact, M generality) *Use the* TEST *instruction instead of* AND *when the result of the logical AND is not used. This saves µops in execution. Use a TEST of a register with itself instead of a CMP of the register to zero, this saves the need to encode the zero and saves encoding space. Avoid comparing a constant to a memory operand. It is preferable to load the memory operand and compare the constant to a register.*

Often a produced value must be compared with zero, and then used in a branch. Because most Intel architecture instructions set the condition codes as part of their execution, the compare instruction may be eliminated. Thus the operation can be tested directly by a JCC instruction. The notable exceptions are MOV and LEA. In these cases, use TEST.

Assembly/Compiler Coding Rule 37. (ML impact, M generality) *Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a* TEST *instruction instead of a compare. Be certain that any code transformations made do not introduce problems with overflow.*

3.5.1.9 Using NOPs

Code generators generate a no-operation (NOP) to align instructions. Examples of NOPs of different lengths in 32-bit mode are shown in Table 3-3.

Table 3-3. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

These are all true NOPs, having no effect on the state of the machine except to advance the EIP. Because NOPs require hardware resources to decode and execute, use the fewest number to achieve the desired padding.

The one byte NOP:[XCHG EAX,EAX] has special hardware support. Although it still consumes a µop and its accompanying resources, the dependence upon the old value of EAX is removed. This µop can be executed at the earliest possible opportunity, reducing the number of outstanding instructions, and is the lowest cost NOP.

The other NOPs have no special hardware support. Their input and output registers are interpreted by the hardware. Therefore, a code generator should arrange to use the register containing the oldest value as input, so that the NOP will dispatch and release RS resources at the earliest possible opportunity.

Try to observe the following NOP generation priority:

- Select the smallest number of NOPs and pseudo-NOPs to provide the desired padding.
- Select NOPs that are least likely to execute on slower execution unit clusters.
- Select the register arguments of NOPs to reduce dependencies.

3.5.1.10 Mixing SIMD Data Types

Previous microarchitectures (before Intel Core microarchitecture) do not have explicit restrictions on mixing integer and floating-point (FP) operations on XMM registers. For Intel Core microarchitecture, mixing integer and floating-point operations on the content of an XMM register can degrade performance. Software should avoid mixed-use of integer/FP operation on XMM registers. Specifically:

- Use SIMD integer operations to feed SIMD integer operations. Use PXOR for idiom.
- Use SIMD floating-point operations to feed SIMD floating-point operations. Use XORPS for idiom.
- When floating-point operations are bitwise equivalent, use PS data type instead of PD data type. MOVAPS and MOVAPD do the same thing, but MOVAPS takes one less byte to encode the instruction.

3.5.1.11 Spill Scheduling

The spill scheduling algorithm used by a code generator will be impacted by the memory subsystem. A spill scheduling algorithm is an algorithm that selects what values to spill to memory when there are too many live values to fit in registers. Consider the code in Example 3-20, where it is necessary to spill either A, B, or C.

Example 3-20. Spill Scheduling Code

```
LOOP
    C := …
    B := …
    A := A + …
```

For modern microarchitectures, using dependence depth information in spill scheduling is even more important than in previous processors. The loop-carried dependence in A makes it especially important that A not be spilled. Not only would a store/load be placed in the dependence chain, but there would also be a data-not-ready stall of the load, costing further cycles.

Assembly/Compiler Coding Rule 38. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies.*

A possibly counter-intuitive result is that in such a situation it is better to put loop invariants in memory than in registers, since loop invariants never have a load blocked by store data that is not ready.

3.5.1.12 Zero-Latency MOV Instructions

In processors based on Ivy Bridge microarchitecture, a subset of register-to-register move operations are executed in the front end (similar to zero-idioms, see Section 3.5.1.7). This conserves scheduling/execution resources in the out-of-order engine. Most forms of register-to-register MOV instructions can benefit from zero-latency MOV. Example 3-21 list the details of those forms that qualify and a small set that do not.

Example 3-21. Zero-Latency MOV Instructions

MOV instructions latency that can be eliminated	MOV instructions latency that cannot be eliminated
MOV reg32, reg32 MOV reg64, reg64 MOVUPD/MOVAPD xmm, xmm MOVUPD/MOVAPD ymm, ymm MOVUPS?MOVAPS xmm, xmm MOVUPS/MOVAPS ymm, ymm MOVDQA/MOVDQU xmm, xmm MOVDQA/MOVDQU ymm, ymm MOVDQA/MOVDQU zmm, zmm MOVZX reg32, reg8 (if not AH/BH/CH/DH) MOVZX reg64, reg8 (if not AH/BH/CH/DH)	MOV reg8, reg8 MOV reg16, reg16 MOVZX reg32, reg8 (if AH/BH/CH/DH) MOVZX reg64, reg8 (if AH/BH/CH/DH) MOVSX

Example 3-22 shows how to process 8-bit integers using MOVZX to take advantage of zero-latency MOV enhancement. Consider

X = (X * 3^N) MOD 256;

Y = (Y * 3^N) MOD 256;

When “MOD 256” is implemented using the “AND 0xff” technique, its latency is exposed in the result-dependency chain. Using a form of MOVZX on a truncated byte input, it can take advantage of zero-latency MOV enhancement and gain about 45% in speed.

Example 3-22. Byte-Granular Data Computation Technique

Use AND Reg32, 0xff	Use MOVZX
mov rsi, N mov rax, X mov rcx, Y loop: lea rcx, [rcx+rcx*2] lea rax, [rax+rax*4] and rcx, 0xff and rax, 0xff lea rcx, [rcx+rcx*2] lea rax, [rax+rax*4] and rcx, 0xff and rax, 0xff sub rsi, 2 jg loop	mov rsi, N mov rax, X mov rcx, Y loop: lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rbx, [rcx+rcx*2] movzx, rcx, bl lea rdx, [rax+rax*4] movzx, rax, dl llea rdx, [rax+rax*4] movzx, rax, dl sub rsi, 2 jg loop

The effectiveness of coding a dense sequence of instructions to rely on a zero-latency MOV instruction must also consider internal resource constraints in the microarchitecture.

Example 3-23. Re-ordering Sequence to Improve Effectiveness of Zero-Latency MOV Instructions

Needing more internal resource for zero-latency MOVs	Needing less internal resource for zero-latency MOVs
mov rsi, N	mov rsi, N
mov rax, X	mov rax, X
mov rcx, Y	mov rcx, Y
loop:	loop:
lea rbx, [rcx+rcx*2]	lea rbx, [rcx+rcx*2]
movzx, rcx, bl	movzx, rcx, bl
lea rdx, [rax+rax*4]	lea rbx, [rcx+rcx*2]
movzx, rax, dl	movzx, rcx, bl
lea rbx, [rcx+rcx*2]	lea rdx, [rax+rax*4]
movzx, rcx, bl	movzx, rax, dl
llea rdx, [rax+rax*4]	llea rdx, [rax+rax*4]
movzx, rax, dl	movzx, rax, dl
sub rsi, 2	sub rsi, 2
jg loop	jg loop

In Example 3-23, RBX/RCX and RDX/RAX are pairs of registers that are shared and continuously over-written. In the right-hand sequence, registers are overwritten with new results immediately, consuming less internal resources provided by the underlying microarchitecture. As a result, it is about 8% faster than the left-hand sequence where internal resources could only support 50% of the attempt to take advantage of zero-latency MOV instructions.

3.5.2 Avoiding Stalls in Execution Core

Although the design of the execution core is optimized to make common cases executes quickly. A micro-op may encounter various hazards, delays, or stalls while making forward progress from the front end to the ROB and RS. The significant cases are:

- ROB Read Port Stalls.
- Partial Register Reference Stalls.
- Partial Updates to XMM Register Stalls.
- Partial Flag Register Reference Stalls.

3.5.2.1 Writeback Bus Conflicts

The writeback bus inside the execution engine is a common resource needed to facilitate out-of-order execution of micro-ops in flight. When the writeback bus is needed at the same time by two micro-ops executing in the same stack of execution units, the younger micro-op will have to wait for the writeback bus to be available. This situation typically will be more likely for short-latency instructions experience a delay when it might have been otherwise ready for dispatching into the execution engine.

Consider a repeating sequence of independent floating-point ADDs with a single-cycle MOV bound to the same dispatch port. When the MOV finds the dispatch port available, the writeback bus can be occupied by the ADD. This delays the MOV operation.

If this problem is detected, you can sometimes change the instruction selection to use a different dispatch port and reduce the writeback contention.

3.5.2.2 Bypass Between Execution Domains

Floating-point (FP) loads have an extra cycle of latency. Moves between FP and SIMD stacks have another additional cycle of latency.

Example:

```
ADDPS  XMM0, XMM1
PAND  XMM0, XMM3
ADDPS  XMM2, XMM0
```

The overall latency for the above calculation is 9 cycles:

- 3 cycles for each ADDPS instruction.
- 1 cycle for the PAND instruction.
- 1 cycle to bypass between the ADDPS floating-point domain to the PAND integer domain.
- 1 cycle to move the data from the PAND integer to the second floating-point ADDPS domain.

To avoid this penalty, organize code to minimize domain changes. Sometimes bypasses cannot be avoided.

Account for bypass cycles when counting the overall latency of your code. If your calculation is latency-bound, you can execute more instructions in parallel or break dependency chains to reduce total latency.

Code that has many bypass domains and is completely latency-bound may run slower on the Intel Core microarchitecture than it did on previous microarchitectures.

3.5.2.3 Partial Register Stalls

Beginning with the Skylake microarchitecture, Partial Register Stalls are no longer treated using micro-operation (UOP) insertions. The hardware takes care of merging the partial register (for instance any of AL, AH or AX is merged into the RAX destination register). This eliminates the special allocation window used to insert merge micro-operation.

From Skylake to Ice Lake microarchitectures, operations that access *H registers (i.e., AH, BH, CH, DH) are executed exclusively on ports 1 and 5.

The *H micro-ops are executed with one cycle latency; however, one cycle of *additional* delay is required for ensuing UOPs because they depend on the results of the *H operation. This additional delay is required due to potential data swapping. A swap might happen, for example, with the instruction "Add AH, BL", or "ADD AL, BH." The pipeline functionality is illustrated in Figure 2-3.

Beginning with the Golden Cove Microarchitecture, the *H operations are limited to Port 1 (port1) with three cycles of latency. This penalty on *H operations helped performance improvement and timing requirements of the Golden Cove microarchitecture.

For more information about Golden Cove microarchitecture, see Section 2.3.1. Figure 2-1 shows the flow.

A closer look at the INT execution ports in Figure 3-1 shows the *H operation limited to Port 1:

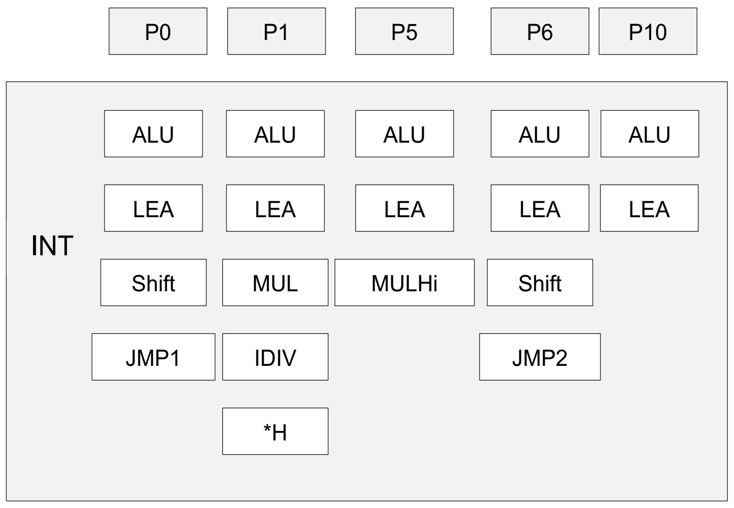


Figure 3-1. INT Execution Ports Within the Processor Core Pipeline

3.5.2.4 Partial XMM Register Stalls

Partial register stalls can also apply to XMM registers. The following SSE and SSE2 instructions update only part of the destination register:

MOVL/HPD XMM, MEM64
MOVL/HPS XMM, MEM32
MOVSS/SD between registers

Using these instructions creates a dependency chain between the unmodified part of the register and the modified part of the register. This dependency chain can cause performance loss.

Example 3-24 illustrates the use of MOVZX to avoid a partial register stall when packing three byte values into a register.

Follow these recommendations to avoid stalls from partial updates to XMM registers:

- Avoid using instructions which update only part of the XMM register.
- If a 64-bit load is needed, use the MOVSD or MOVQ instruction.
- If 2 64-bit loads are required to the same register from non continuous locations, use MOVSD/MOVHPD instead of MOVLPD/MOVHPD.
- When copying the XMM register, use the following instructions for full register copy, even if you only want to copy some of the source register data:

MOVAPS
MOVAPD
MOVDQA

Example 3-24. Avoiding Partial Register Stalls in SIMD Code

Using movlpd for memory transactions and movsd between register copies Causing Partial Register Stall	Using movsd for memory and movapd between register copies Avoid Delay

```
        mov      edx, x
        mov      ecx, count
        movlpd   xmm3,_1_
        movlpd   xmm2,_1pt5_
        align 16
    lp:
        movlpd   xmm0, [edx]
        addsd    xmm0, xmm3
        movsd    xmm1, xmm2
        subsd    xmm1, [edx]
        mulsd    xmm0, xmm1
        movsd    [edx], xmm0
        add      edx, 8
        dec      ecx
        jnz      lp
```
	```
	        mov      edx, x
	        mov      ecx, count
	        movsd    xmm3,_1_
	        movsd    xmm2, _1pt5_
	        align 16
	    lp:
	        movsd    xmm0, [edx]
	        addsd    xmm0, xmm3
	        movapd   xmm1, xmm2
	        subsd    xmm1, [edx]
	        mulsd    xmm0, xmm1
	        movsd    [edx], xmm0
	        add      edx, 8
	        dec      ecx
	        jnz      lp
	```

3.5.2.5 Partial Flag Register Stalls

A "partial flag register stall" occurs when an instruction modifies a part of the flag register and the following instruction is dependent on the outcome of the flags. This happens most often with shift instructions (SAR, SAL, SHR, SHL). The flags are not modified in the case of a zero shift count, but the shift count is usually known only at execution time. The front end stalls until the instruction is retired.

Other instructions that can modify some part of the flag register include CMPXCHG8B, various rotate instructions, STC, and STD. An example of assembly with a partial flag register stall and alternative code without the stall is shown in Example 3-25.

In processors based on Intel Core microarchitecture, shift immediate by 1 is handled by special hardware such that it does not experience partial flag stall.

Example 3-25. Avoiding Partial Flag Register Stalls

Partial Flag Register Stall	Avoiding Partial Flag Register Stall

```
xor     eax, eax
mov     ecx, a
sar     ecx, 2
setz al ;SAR can update carry causing a stall
```
	```
	or      eax, eax
	mov     ecx, a
	sar     ecx, 2
	test    ecx, ecx ; test always updates all flags
	setz al ;No partial reg or flag stall,
	```

In Sandy Bridge microarchitecture, the cost of partial flag access is replaced by the insertion of a micro-op instead of a stall. However, it is still recommended to use less of instructions that write only to some of the flags (such as INC, DEC, SET CL) before instructions that can write flags conditionally (such as SHIFT CL).

Example 3-26 compares two techniques to implement the addition of very large integers (e.g., 1024 bits). The alternative sequence on the right side of Example 3-26 will be faster than the left side on Sandy Bridge microarchitecture, but it will experience partial flag stalls on prior microarchitectures.

Example 3-26. Partial Flag Register Accesses in Sandy Bridge Microarchitecture

Save partial flag register to avoid stall	Simplified code sequence

```
    lea     rsi, [A]
    lea     rdi, [B]
    xor     rax, rax
    mov     rcx, 16 ; 16*64 =1024 bit

lp_64bit:
    add     rax, [rsi]
    adc     rax, [rdi]
    mov     [rdi], rax
    setc al ;save carry for next iteration
    movzx rax, al
    add     rsi, 8
    add     rdi, 8
    dec     rcx
    jnz     lp_64bit
```
	```
	    lea  rsi, [A]
	    lea  rdi, [B]
	    xor  rax, rax
	    mov rcx, 16

	lp_64bit:
	    add     rax, [rsi]
	    adc     rax, [rdi]
	    mov     [rdi], rax
	    lea     rsi, [rsi+8]
	    lea     rdi, [rdi+8]
	    dec     rcx
	    jnz     lp_64bit
	```

3.5.2.6 Floating-Point/SIMD Operands

Moves that write a portion of a register can introduce unwanted dependences. The MOVSD REG, REG instruction writes only the bottom 64 bits of a register, not all 128 bits. This introduces a dependence on the preceding instruction that produces the upper 64 bits (even if those bits are not longer wanted). The dependence inhibits register renaming, and thereby reduces parallelism.

Use MOVAPD as an alternative; it writes all 128 bits. Even though this instruction has a longer latency, the μops for MOVAPD use a different execution port and this port is more likely to be free. The change can impact performance. There may be exceptional cases where the latency matters more than the dependence or the execution port.

Assembly/Compiler Coding Rule 39. (M impact, ML generality) *Avoid introducing dependences with partial floating-point register writes, e.g. from the* MOVSD XMMREG1, XMMREG2 *instruction. Use the* MOVAPD XMMREG1, XMMREG2 *instruction instead.*

The MOVSD XMMREG, MEM instruction writes all 128 bits and breaks a dependence.

3.5.3 Vectorization

This section provides a brief summary of optimization issues related to vectorization. There is more detail in the chapters that follow.

Vectorization is a program transformation that allows special hardware to perform the same operation on multiple data elements at the same time. Successive processor generations have provided vector support through the MMX technology, Intel Streaming SIMD Extensions (Intel SSE), Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3) and Intel Supplemental Streaming SIMD Extensions 3 (Intel SSSE3).

Vectorization is a special case of SIMD, a term defined in Flynn's architecture taxonomy to denote a single instruction stream capable of operating on multiple data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating-point data elements in Intel SSE and two double-precision floating-point data elements in Intel SSE2 to sixteen byte operations in a 128-bit register in Intel SSE2. Thus, vector length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

The Intel C++ Compiler supports vectorization in three ways:

- The compiler may be able to generate SIMD code without intervention from the user.

- The can user insert pragmas to help the compiler realize that it can vectorize the code.
- The user can write SIMD code explicitly using intrinsics and C++ classes.

To help enable the compiler to generate SIMD code, avoid global pointers and global variables. These issues may be less troublesome if all modules are compiled simultaneously, and whole-program optimization is used.

User/Source Coding Rule 2. (H impact, M generality) *Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.*

User/Source Coding Rule 3. (M impact, ML generality) *Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.*

The integer part of the SIMD instruction set extensions cover 8-bit,16-bit and 32-bit operands. Not all SIMD operations are supported for 32 bits, meaning that some source code will not be able to be vectorized at all unless smaller operands are used.

User/Source Coding Rule 4. (M impact, ML generality) *Avoid the use of conditional branches inside loops and consider using SSE instructions to eliminate branches.*

User/Source Coding Rule 5. (M impact, ML generality) *Keep induction (loop) variable expressions simple.*

3.5.4 Optimization of Partially Vectorizable Code

Frequently, a program contains a mixture of vectorizable code and some routines that are non-vectorizable. A common situation of partially vectorizable code involves a loop structure which include mixtures of vectorized code and unvectorizable code. This situation is depicted in Figure 3-2.

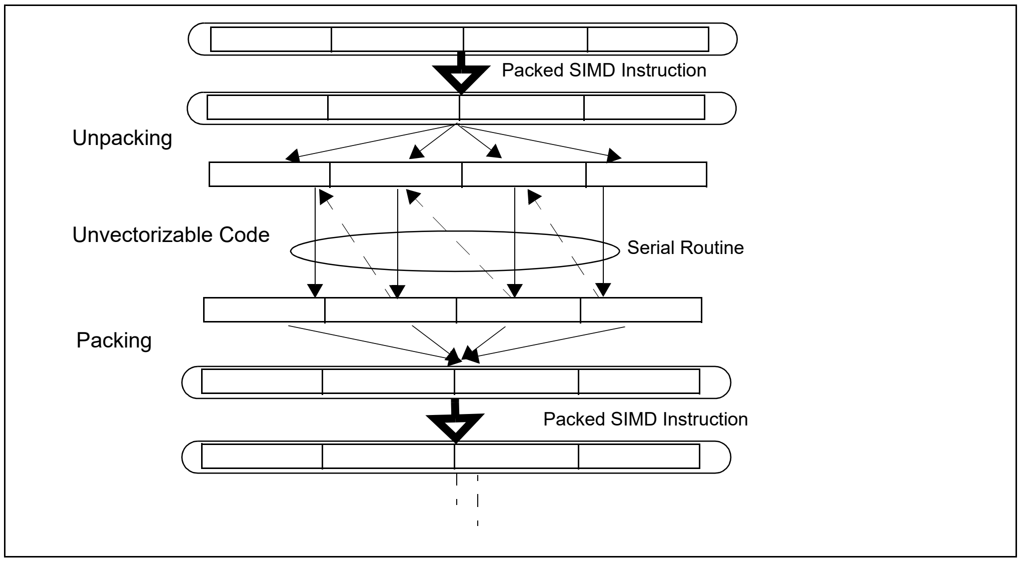


Figure 3-2. Generic Program Flow of Partially Vectorized Code

It generally consists of five stages within the loop:

- Prolog.
- Unpacking vectorized data structure into individual elements.
- Calling a unvectorizable routine to process each element serially.
- Packing individual result into vectorized data structure.
- Epilogue.

This section discusses techniques that can reduce the cost and bottleneck associated with the packing/unpacking stages in these partially vectorize code.

Example 3-27 shows a reference code template that is representative of partially vectorizable coding situations that also experience performance issues. The unvectorizable portion of code is represented generically by a sequence of calling a serial function named "foo" multiple times. This generic example is referred to as "shuffle with store forwarding", because the problem generally involves an unpacking stage that shuffles data elements between register and memory, followed by a packing stage that can experience store forwarding issue.

There are more than one useful techniques that can reduce the store-forwarding bottleneck between the serialized portion and the packing stage. The following sub-sections presents alternate techniques to deal with the packing, unpacking, and parameter passing to serialized function calls.

Example 3-27. Reference Code Template for Partially Vectorizable Program

```
// Prolog   ///////////////////////////////
push ebp
mov ebp, esp

// Unpacking  ////////////////////////////
sub ebp, 32
and ebp, 0xfffffff0
movaps [ebp], xmm0


// Serial operations on components ///////
sub ebp, 4

mov eax, [ebp+4]
mov [ebp], eax
call foo
mov [ebp+16+4], eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
mov [ebp+16+4+4], eax


mov eax, [ebp+12]
mov [ebp], eax
call foo
mov [ebp+16+8+4], eax


mov eax, [ebp+12+4]
mov [ebp], eax
call foo
mov [ebp+16+12+4], eax


// Packing ///////////////////////////////
movaps xmm0, [ebp+16+4]

// Epilog ////////////////////////////////
pop ebp
ret
```

3.5.4.1 Alternate Packing Techniques

The packing method implemented in the reference code of Example 3-27 will experience delay as it assembles 4 doubleword result from memory into an XMM register due to store-forwarding restrictions.

Three alternate techniques for packing, using different SIMD instruction to assemble contents in XMM registers are shown in Example 3-28. All three techniques avoid store-forwarding delay by satisfying the restrictions on data sizes between a preceding store and subsequent load operations.

Example 3-28. Three Alternate Packing Methods for Avoiding Store Forwarding Difficulty

Packing Method 1	Packing Method 2	Packing Method 3
movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] punpckldq xmm0, xmm1 punpckldq xmm2, xmm3 punpckldq xmm0, xmm2	movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] psllq xmm3, 32 orps xmm2, xmm3 psllq xmm1, 32 orps xmm0, xmm1movlhps xmm0, xmm2	movd xmm0, [ebp+16+4] movd xmm1, [ebp+16+8] movd xmm2, [ebp+16+12] movd xmm3, [ebp+12+16+4] movlhps xmm1,xmm3 psllq xmm1, 32 movlhps xmm0, xmm2 orps xmm0, xmm1

3.5.4.2 Simplifying Result Passing

In Example 3-27, individual results were passed to the packing stage by storing to contiguous memory locations. Instead of using memory spills to pass four results, result passing may be accomplished by using either one or more registers. Using registers to simplify result passing and reduce memory spills can improve performance by varying degrees depending on the register pressure at runtime.

Example 3-29 shows the coding sequence that uses four extra XMM registers to reduce all memory spills of passing results back to the parent routine. However, software must observe the following conditions when using this technique:

- There is no register shortage.
- If the loop does not have many stores or loads but has many computations, this technique does not help performance. This technique adds work to the computational units, while the store and loads ports are idle.

Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing

```
mov eax, [ebp+4]
mov [ebp], eax
call foo
movd xmm0, eax

mov eax, [ebp+8]
mov [ebp], eax
call foo
movd xmm1, eax
```

Example 3-29. Using Four Registers to Reduce Memory Spills and Simplify Result Passing (Contd.)

```
mov eax, [ebp+12]
mov [ebp], eax
call foo
movd xmm2, eax

mov eax, [ebp+12+4]
mov [ebp], eax
call foo
movd xmm3, eax
```

3.5.4.3 Stack Optimization

In Example 3-27, an input parameter was copied in turn onto the stack and passed to the unvectorizable routine for processing. The parameter passing from consecutive memory locations can be simplified by a technique shown in Example 3-30.

Example 3-30. Stack Optimization Technique to Simplify Parameter Passing

```
call foo
mov [ebp+16], eax

add ebp, 4
call foo
mov [ebp+16], eax


add ebp, 4
call foo
mov [ebp+16], eax

add ebp, 4
call foo
```

Stack Optimization can only be used when:

- The serial operations are function calls. The function "foo" is declared as: INT FOO(INT A). The parameter is passed on the stack.
- The order of operation on the components is from last to first.

Note the call to FOO and the advance of EDP when passing the vector elements to FOO one by one from last to first.

3.5.4.4 Tuning Considerations

Tuning considerations for situations represented by looping of Example 3-27 include:

- Applying one of more of the following combinations:
  - — Choose an alternate packing technique.
  - — Consider a technique to simply result-passing.
  - — Consider the stack optimization technique to simplify parameter passing.
- Minimizing the average number of cycles to execute one iteration of the loop.
- Minimizing the per-iteration cost of the unpacking and packing operations.

The speed improvement by using the techniques discussed in this section will vary, depending on the choice of combinations implemented and characteristics of the non-vectorizable routine. For example, if the routine “foo” is short (representative of tight, short loops), the per-iteration cost of unpacking/packing tend to be smaller than situations where the non-vectorizable code contain longer operation or many dependencies. This is because many iterations of short, tight loop can be in flight in the execution core, so the per-iteration cost of packing and unpacking is only partially exposed and appear to cause very little performance degradation.

Evaluation of the per-iteration cost of packing/unpacking should be carried out in a methodical manner over a selected number of test cases, where each case may implement some combination of the techniques discussed in this section. The per-iteration cost can be estimated by:

- Evaluating the average cycles to execute one iteration of the test case.
- Evaluating the average cycles to execute one iteration of a base line loop sequence of non-vectorizable code.

Example 3-31 shows the base line code sequence that can be used to estimate the average cost of a loop that executes non-vectorizable routines.

Example 3-31. Base Line Code Sequence to Estimate Loop Overhead

```
push ebp
mov ebp, esp
sub ebp, 4

mov [ebp], edi
call foo


mov [ebp], edi
call foo

mov [ebp], edi
call foo


mov [ebp], edi
call foo

add ebp, 4
pop ebp
ret
```

The average per-iteration cost of packing/unpacking can be derived from measuring the execution times of a large number of iterations by:

((Cycles to run TestCase) - (Cycles to run equivalent baseline sequence)) / (Iteration count).

For example, using a simple function that returns an input parameter (representative of tight, short loops), the per-iteration cost of packing/unpacking may range from slightly more than 7 cycles (the shuffle with store forwarding case, Example 3-27) to ~0.9 cycles (accomplished by several test cases). Across 27 test cases (consisting of one of the alternate packing methods, no result-simplification/simplification of either 1 or 4 results, no stack optimization or with stack optimization), the average per-iteration cost of packing/unpacking is about 1.7 cycles.

Generally speaking, packing method 2 and 3 (see Example 3-28) tend to be more robust than packing method 1; the optimal choice of simplifying 1 or 4 results will be affected by register pressure of the runtime and other relevant microarchitectural conditions.

Note that the numeric discussion of per-iteration cost of packing/packing is illustrative only. It will vary with test cases using a different base line code sequence and will generally increase if the non-vectorizable routine requires longer time to execute because the number of loop iterations that can reside in flight in the execution core decreases.

3.6 OPTIMIZING MEMORY ACCESSES

This section discusses guidelines for optimizing code and data memory accesses. The most important recommendations are:

- Execute load and store operations within available execution bandwidth.
- Enable forward progress of speculative execution.
- Enable store forwarding to proceed.
- Align data, paying attention to data layout and stack alignment.
- Place code and data on separate pages.
- Enhance data locality.
- Use prefetching and cacheability control instructions.
- Enhance code locality and align branch targets.
- Take advantage of write combining.

3.6.1 Load and Store Execution Bandwidth

Typically, loads and stores are the most frequent operations in a workload, up to 40% of the instructions in a workload carrying load or store intent are not uncommon. Each generation of microarchitecture provides multiple buffers to support executing load and store operations while there are instructions in flight. These buffers were comprised of 128-bit wide entries for the Sandy Bridge and Ivy Bridge microarchitectures. The size was increased to 256-bit in Haswell, Broadwell and Skylake Client microarchitectures; and to 512-bit in Skylake Server, Cascade Lake, Cascade Lake Advanced Performance, and Ice Lake Client microarchitectures. To maximize performance, it is best to use the largest width available in the platform.

3.6.1.1 Making Use of Load Bandwidth in Sandy Bridge Microarchitecture

While prior microarchitecture has one load port (port 2), Sandy Bridge microarchitecture can load from port 2 and port 3. Thus two load operations can be performed every cycle and doubling the load throughput of the code. This improves code that reads a lot of data and does not need to write out results to memory very often (Port 3 also handles store-address operation). To exploit this bandwidth, the data has to stay in the L1 data cache or it should be accessed sequentially, enabling the hardware prefetchers to bring the data to the L1 data cache in time.

Consider the following C code example of adding all the elements of an array:

```
int buff[BUFF_SIZE];
int sum = 0;

for (i=0;i<BUFF_SIZE;i++){
    sum+=buff[i];
}
```

Alternative 1 is the assembly code generated by the Intel compiler for this C code, using the optimization flag for Nehalem microarchitecture. The compiler vectorizes execution using Intel SSE instructions. In this code, each ADD operation uses the result of the previous ADD operation. This limits the throughput to one load and ADD operation per cycle. Alternative 2 is optimized for Sandy Bridge microarchitecture by enabling it to use the additional load bandwidth. The code removes the dependency among ADD operations, by using two registers to sum the array values. Two load and two ADD operations can be executed every cycle.

Example 3-32. Optimizing for Load Port Bandwidth in Sandy Bridge Microarchitecture

Register dependency inhibits PADD execution	Reduce register dependency allow two load port to supply PADD execution
xor eax, eax	xor eax, eax
pxor xmm0, xmm0	pxor xmm0, xmm0
lea rsi, buff	pxor xmm1, xmm1
	lea rsi, buff
loop_start:	loop_start:
paddd xmm0, [rsi+4*rax]	paddd xmm0, [rsi+4*rax]
paddd xmm0, [rsi+4*rax+16]	paddd xmm1, [rsi+4*rax+16]
paddd xmm0, [rsi+4*rax+32]	paddd xmm0, [rsi+4*rax+32]
paddd xmm0, [rsi+4*rax+48]	paddd xmm1, [rsi+4*rax+48]
paddd xmm0, [rsi+4*rax+64]	paddd xmm0, [rsi+4*rax+64]
paddd xmm0, [rsi+4*rax+80]	paddd xmm1, [rsi+4*rax+80]
paddd xmm0, [rsi+4*rax+96]	paddd xmm0, [rsi+4*rax+96]
paddd xmm0, [rsi+4*rax+112]	paddd xmm1, [rsi+4*rax+112]
add eax, 32	add eax, 32
cmp eax, BUFF_SIZE	cmp eax, BUFF_SIZE
jl loop_start	jl loop_start
sum_partials:	sum_partials:
movdqa xmm1, xmm0	paddd xmm0, xmm1
psrldq xmm1, 8	movdqa xmm1, xmm0
paddd xmm0, xmm1	psrldq xmm1, 8
movdqa xmm2, xmm0	paddd xmm0, xmm1
psrldq xmm2, 4	movdqa xmm2, xmm0
paddd xmm0, xmm2	psrldq xmm2, 4
movd [sum], xmm0	paddd xmm0, xmm2
	movd [sum], xmm0

3.6.1.2 L1D Cache Latency in Sandy Bridge Microarchitecture

Load latency from L1D cache may vary. The best case if 4 cycles, which apply to load operations to general purpose registers using one of the following:

- One register.
- A base register plus an offset that is smaller than 2048.

Consider the pointer-chasing code example in Example 3-33.

Example 3-33. Index versus Pointers in Pointer-Chasing Code

Traversing through indexes	Traversing through pointers

`// C code example`
`index = buffer.m_buff[index].next_index;`
`// ASM example`
`loop:`
`    shl rbx,  6`
`    mov rbx, 0x20(rbx+rcx)`
`    dec rax`
`    cmp rax, -1`
`jne loop`
	`// C code example`
	`    node = node->pNext;`
	`// ASM example`
	`loop:`
	`    mov rdx, [rdx]`
	`    dec rax`
	`    cmp rax, -1`
	`    jne loop`

The left side implements pointer chasing via traversing an index. Compiler then generates the code shown below addressing memory using base+index with an offset. The right side shows compiler generated code from pointer de-referencing code and uses only a base register.

The code on the right side is faster than the left side across Sandy Bridge microarchitecture and prior microarchitecture. However the code that traverses index will be slower on Sandy Bridge microarchitecture relative to prior microarchitecture.

3.6.1.3 Handling L1D Cache Bank Conflict

In the Sandy Bridge microarchitecture, the internal organization of the L1D cache may manifest a situation when two load micro-ops whose addresses have a bank conflict. When a bank conflict is present between two load operations, the more recent one will be delayed until the conflict is resolved. A bank conflict happens when two simultaneous load operations have the same bit 2-5 of their linear address but they are not from the same set in the cache (bits 6 - 12).

Bank conflicts should be handled only if the code is bound by load bandwidth. Some do not cause any performance degradation since they are hidden by other performance limiters. Eliminating such bank conflicts does not improve performance.

The L1D cache bank conflict issue does not apply to Haswell microarchitecture.

The following example demonstrates bank conflict and how to modify the code and avoid them. It uses two source arrays with a size that is a multiple of cache line size. When loading an element from A and the counterpart element from B the elements have the same offset in their cache lines; therefore, a bank conflict may happen.

Example 3-34. Example of Bank Conflicts in L1D Cache and Remedy

```
int A[128];
int B[128];
int C[128];
for (i=0;i<128;i+=4){
    C[i]=A[i]+B[i];        the loads from A[i] and B[i] collide
    C[i+1]=A[i+1]+B[i+1];
    C[i+2]=A[i+2]+B[i+2];
    C[i+3]=A[i+3]+B[i+3];
}
```

```
// Code with Bank Conflicts
    xor rcx, rcx
    lea r11, A
    lea r12, B
    lea r13, C
loop:
    lea esi, [rcx*4]
    movsxd rsi, esi
    mov edi, [r11+rsi*4]
    add edi, [r12+rsi*4]
    mov r8d, [r11+rsi*4+4]
    add r8d, [r12+rsi*4+4]
    mov r9d, [r11+rsi*4+8]
    add r9d, [r12+rsi*4+8]
    mov r10d, [r11+rsi*4+12]
    add r10d, [r12+rsi*4+12]


    mov [r13+rsi*4], edi
    inc ecx
    mov [r13+rsi*4+4], r8d
    mov [r13+rsi*4+8], r9d
    mov [r13+rsi*4+12], r10d
    cmp ecx, LEN
    jb loop
```

```
// Code without Bank Conflicts
    xor rcx, rcx
    lea r11, A
    lea r12, B
    lea r13, C
loop:
    lea esi, [rcx*4]
    movsxd rsi, esi
    mov edi, [r11+rsi*4]
    mov r8d, [r11+rsi*4+4]
    add edi, [r12+rsi*4]
    add r8d, [r12+rsi*4+4]
    mov r9d, [r11+rsi*4+8]
    mov r10d, [r11+rsi*4+12]
    add r9d, [r12+rsi*4+8]
    add r10d, [r12+rsi*4+12]


    inc ecx
    mov [r13+rsi*4], edi
    mov [r13+rsi*4+4], r8d
    mov [r13+rsi*4+8], r9d
    mov [r13+rsi*4+12], r10d
    cmp ecx, LEN
    jb loop
```

Bank conflicts may occur with the introduction of the third load port in the Golden Cove microarchitecture. In this microarchitecture, conflicts happen between three loads with the same bits 2-5 of their linear address even if they access the same set of the cache. Up to two loads can access the same cache bank without a conflict; however, a third load accessing the same bank must be delayed. The bank conflicts do not apply to 512-bit wide loads because their bandwidth is limited to two per cycle.

Recommendation: In the Golden Cove microarchitecture, bank conflicts often happen when multiple loads access the same memory location. Whenever possible, avoid reading the same memory location within a tight loop or using multiple load operations. Commonly used memory locations are better kept in the registers to prevent potential bank conflict penalty.

3.6.2 Minimize Register Spills

When a piece of code has more live variables than the processor can keep in general purpose registers, a common method is to hold some of the variables in memory. This method is called register spill. The effect of L1D cache latency can negatively affect the performance of this code. The effect can be more pronounced if the address of register spills uses the slower addressing modes.

One option is to spill general purpose registers to XMM registers. This method is likely to improve performance also on previous processor generations. The following example shows how to spill a register to an XMM register rather than to memory.

Example 3-35. Using XMM Register in Lieu of Memory for Register Spills

Register spills into memory	Register spills into XMM
loop:	movq xmm4, [rsp+0x18]
mov rdx, [rsp+0x18]	mov rcx, 0x10
movdqa xmm0, [rdx]	movq xmm5, rcx
movdqa xmm1, [rsp+0x20]	loop:
pcmpeqd xmm1, xmm0	movq rdx, xmm4
pmovmskb eax, xmm1	movdqa xmm0, [rdx]
test eax, eax	movdqa xmm1, [rsp+0x20]
jne end_loop	pcmpeqd xmm1, xmm0
movzx rcx, [rbx+0x60]	pmovmskb eax, xmm1
	test eax, eax
	jne end_loop
	movzx rcx, [rbx+0x60]
add qword ptr[rsp+0x18], 0x10	padd xmm4, xmm5
add rdi, 0x4	add rdi, 0x4
movzx rdx, di	movzx rdx, di
sub rcx, 0x4	sub rcx, 0x4
add rsi, 0x1d0	add rsi, 0x1d0
cmp rdx, rcx	cmp rdx, rcx
jle loop	jle loop

3.6.3 Enhance Speculative Execution and Memory Disambiguation

Prior to Intel Core microarchitecture, when code contains both stores and loads, the loads cannot be issued before the address of the older stores is known. This rule ensures correct handling of load dependencies on preceding stores.

The Intel Core microarchitecture contains a mechanism that allows some loads to be executed speculatively in the presence of older unknown stores. The processor later checks if the load address overlapped with an older store whose address was unknown at the time the load executed. If the addresses do overlap, then the processor re-executes the load and all succeeding instructions.

Example 3-36 illustrates a situation that the compiler cannot be sure that “Ptr->Array” does not change during the loop. Therefore, the compiler cannot keep “Ptr->Array” in a register as an invariant and must read it again in every iteration. Although this situation can be fixed in software by a rewriting the code to require the address of the pointer is invariant, memory disambiguation improves performance without rewriting the code.

Example 3-36. Loads Blocked by Stores of Unknown Address

C code	Assembly sequence

struct AA {
AA ** array;
};
void nullify_array (AA *Ptr, DWORD Index, AA *ThisPtr)
{
while (Ptr->Array[--Index] != ThisPtr)
    {
    Ptr->Array[Index] = NULL ;
    } ;
} ;
	nullify_loop:
	mov dword ptr [eax], 0
	mov edx, dword ptr [edi]
	sub ecx, 4
	cmp dword ptr [ecx+edx], esi
	lea eax, [ecx+edx]
	jne nullify_loop

It is possible to disable speculative store bypass with the IA32_SPEC_CTRL.SSBD MSR.

Additional information on this topic can be found on the Software Security Guidance page.

3.6.4 Store Forwarding

The processor's memory system only sends stores to memory (including cache) after store retirement. However, store data can be forwarded from a store to a subsequent load from the same address to give a much shorter store-load latency.

There are two kinds of requirements for store forwarding. If these requirements are violated, store forwarding cannot occur and the load must get its data from the cache (so the store must write its data back to the cache first). This incurs a penalty that is largely related to pipeline depth of the underlying micro-architecture.

The first requirement pertains to the size and alignment of the store-forwarding data. This restriction is likely to have high impact on overall application performance. Typically, a performance penalty due to violating this restriction can be prevented. The store-to-load forwarding restrictions vary from one microarchitecture to another. Several examples of coding pitfalls that cause store-forwarding stalls and solutions to these pitfalls are discussed in detail in Section 3.6.4.1 The second requirement is the availability of data, discussed in Section 3.6.4.2 A good practice is to eliminate redundant load operations.

It may be possible to keep a temporary scalar variable in a register and never write it to memory. Generally, such a variable must not be accessible using indirect pointers. Moving a variable to a register eliminates all loads and stores of that variable and eliminates potential problems associated with store forwarding. However, it also increases register pressure.

Load instructions tend to start chains of computation. Since the out-of-order engine is based on data dependence, load instructions play a significant role in the engine's ability to execute at a high rate. Eliminating loads should be given a high priority.

If a variable does not change between the time when it is stored and the time when it is used again, the register that was stored can be copied or used directly. If register pressure is too high, or an unseen function is called before the store and the second load, it may not be possible to eliminate the second load.

Assembly/Compiler Coding Rule 40. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible. Passing arguments on the stack requires a store followed by a reload. While this sequence is optimized in hardware by providing the value to the load directly from the memory order buffer without the need to access the data cache if permitted by store-forwarding restrictions, floating-point values incur a significant latency in forwarding. Passing floating-point arguments in (preferably XMM) registers should save this long latency operation.*

Parameter passing conventions may limit the choice of which parameters are passed in registers which are passed on the stack. However, these limitations may be overcome if the compiler has control of the compilation of the whole binary (using whole-program optimization).

3.6.4.1 Store-to-Load-Forwarding Restriction on Size and Alignment

Data size and alignment restrictions for store-forwarding apply to processors based on Intel Core microarchitecture, Intel Core 2 Duo, Intel Core Solo and Pentium M processors. The performance penalty for violating store-forwarding restrictions is less for shorter-pipelined machines.

Store-forwarding restrictions vary with each microarchitecture. The following rules help satisfy size and alignment restrictions for store forwarding:

Assembly/Compiler Coding Rule 41. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data.*

Assembly/Compiler Coding Rule 42. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data.*

A load that forwards from a store must wait for the store's data to be written to the store buffer before proceeding, but other, unrelated loads need not wait.

Assembly/Compiler Coding Rule 43. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. This is better than incurring the penalties of a failed store-forward.*

Assembly/Compiler Coding Rule 44. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed.*

Example 3-37 depicts several store-forwarding situations in which small loads follow large stores. The first three load operations illustrate the situations described in Rule 44. However, the last load operation gets data from store-forwarding without problem.

Example 3-37. Situations Showing Small Loads After Large Store

```
mov [EBP],'abcd'
mov AL, [EBP]        ; Not blocked - same alignment
mov BL, [EBP + 1]    ; Blocked
mov CL, [EBP + 2]    ; Blocked
mov DL, [EBP + 3]    ; Blocked
mov AL, [EBP]        ; Not blocked - same alignment
                     ; n.b. passes older blocked loads
```

Example 3-38 illustrates a store-forwarding situation in which a large load follows several small stores. The data needed by the load operation cannot be forwarded because all of the data that needs to be forwarded is not contained in the store buffer. Avoid large loads after small stores to the same area of memory.

Example 3-38. Non-forwarding Example of Large Load After Small Store

```
mov [EBP], 'a'
mov [EBP + 1], 'b'
mov [EBP + 2], 'c'
mov [EBP + 3], 'd'
mov EAX, [EBP]    ; Blocked
    ; The first 4 small store can be consolidated into
    ; a single DWORD store to prevent this non-forwarding
    ; situation.
```

Example 3-39 illustrates a stalled store-forwarding situation that may appear in compiler generated code. Sometimes a compiler generates code similar to that shown in Example 3-39 to handle a spilled byte to the stack and convert the byte to an integer value.

Example 3-39. A Non-forwarding Situation in Compiler Generated Code

```
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h]  ; Stall
and eax, 0xff                 ; Converting back to byte value
```

Example 3-40 offers two alternatives to avoid the non-forwarding situation shown in Example 3-39.

Example 3-40. Two Ways to Avoid Non-forwarding Situation in Example 3-39

```
; A. Use MOVZ instruction to avoid large load after small
; store, when spills are ignored.
movz eax, bl                    ; Replaces the last three instructions
; B. Use MOVZ instruction and handle spills to the stack
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h]    ; Not blocked
```

When moving data that is smaller than 64 bits between memory locations, 64-bit or 128-bit SIMD register moves are more efficient (if aligned) and can be used to avoid unaligned loads. Although floating-point registers allow the movement of 64 bits at a time, floating-point instructions should not be used for this purpose, as data may be inadvertently modified.

As an additional example, consider the cases in Example 3-41.

Example 3-41. Large and Small Load Stalls

```
; A. Large load stall
mov     mem, eax        ; Store dword to address “MEM”
mov     mem + 4, ebx    ; Store dword to address “MEM + 4”
fld     mem             ; Load qword at address “MEM”, stalls
; B. Small Load stall
fstp  mem               ; Store qword to address “MEM”
mov   bx, mem+2         ; Load word at address “MEM + 2”, stalls
mov   cx, mem+4         ; Load word at address “MEM + 4”, stalls
```

In the first case (A), there is a large load after a series of small stores to the same area of memory (beginning at memory address MEM). The large load will stall.

The FLD must wait for the stores to write to memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

In the second case (B), there is a series of small loads after a large store to the same area of memory (beginning at memory address MEM). The small loads will stall.

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible.

Store forwarding restrictions for processors based on Intel Core microarchitecture is listed in Table 3-4.

Table 3-4. Store Forwarding Restrictions of Processors Based on Intel Core Microarchitecture

Store Alignment	Width of Store (bits)	Load Alignment (byte)	Width of Load (bits)	Store Forwarding Restriction
To Natural size	16	word aligned	8, 16	not stalled
To Natural size	16	not word aligned	8	stalled
To Natural size	32	dword aligned	8, 32	not stalled
To Natural size	32	not dword aligned	8	stalled
To Natural size	32	word aligned	16	not stalled
To Natural size	32	not word aligned	16	stalled
To Natural size	64	qword aligned	8, 16, 64	not stalled
To Natural size	64	not qword aligned	8, 16	stalled
To Natural size	64	dword aligned	32	not stalled
To Natural size	64	not dword aligned	32	stalled
To Natural size	128	dqword aligned	8, 16, 128	not stalled
To Natural size	128	not dqword aligned	8, 16	stalled
To Natural size	128	dword aligned	32	not stalled
To Natural size	128	not dword aligned	32	stalled
To Natural size	128	qword aligned	64	not stalled
To Natural size	128	not qword aligned	64	stalled
Unaligned, start byte 1	32	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	32	not byte 0 of store	8, 16	stalled
Unaligned, start byte 1	64	byte 0 of store	8, 16, 32	not stalled
Unaligned, start byte 1	64	not byte 0 of store	8, 16, 32	stalled
Unaligned, start byte 1	64	byte 0 of store	64	stalled
Unaligned, start byte 7	32	byte 0 of store	8	not stalled
Unaligned, start byte 7	32	not byte 0 of store	8	not stalled
Unaligned, start byte 7	32	don't care	16, 32	stalled
Unaligned, start byte 7	64	don't care	16, 32, 64	stalled

3.6.4.2 Store-Forwarding Restriction on Data Availability

The value to be stored must be available before the load operation can be completed. If this restriction is violated, the execution of the load will be delayed until the data is available. This delay causes some execution resources to be used unnecessarily, and that can lead to sizable but non-deterministic delays. However, the overall impact of this problem is much smaller than that from violating size and alignment requirements.

In modern microarchitectures, hardware predicts when loads are dependent on and get their data forwarded from preceding stores. These predictions can significantly improve performance. However, if a load is scheduled too soon after the store it depends on or if the generation of the data to be stored is delayed, there can be a significant penalty.

There are several cases in which data is passed through memory, and the store may need to be separated from the load:

- Spills, save and restore registers in a stack frame.
- Parameter passing.
- Global and volatile variables.

- Type conversion between integer and floating-point.
- When compilers do not analyze code that is inlined, forcing variables that are involved in the interface with inlined code to be in memory, creating more memory variables and preventing the elimination of redundant loads.

Assembly/Compiler Coding Rule 45. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction - for example, MUL or DIV. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain.*

Example 3-42 shows an example of a loop-carried dependence chain.

Example 3-42. Loop-Carried Dependence Chain

```
for ( i = 0; i < MAX; i++ ) {
    a[i] = b[i] * foo;
    foo = a[i] / 3;
}                    // foo is a loop-carried dependence.
```

Assembly/Compiler Coding Rule 46. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads.*

3.6.5 Data Layout Optimizations

User/Source Coding Rule 6. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary.*

If the operands are packed in a SIMD instruction, align to the packed element size (64-bit or 128-bit).

Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding. However, compilers might not have this freedom. The C programming language, for example, specifies the order in which structure elements are allocated in memory. For more information, see Section 5.4.

Example 3-43 shows how a data structure could be rearranged to reduce its size.

Example 3-43. Rearranging a Data Structure

```
struct unpacked { /* Fits in 20 bytes due to padding */
    int     a;
    char    b;
    int     c;
    char    d;
    int     e;
};

struct packed {  /* Fits in 16 bytes */
    int     a;
    int     c;
    int     e;
    char    b;
    char    d;
}
```

Cache line size of 64 bytes can impact streaming applications (for example, multimedia). These reference and use data only once before discarding it. Data accesses which sparsely utilize the data within a cache line can result in less efficient utilization of system memory bandwidth. For example, arrays of structures can be decomposed into several arrays to achieve better packing, as shown in Example 3-44.

Example 3-44. Decomposing an Array

```
struct {        /* 1600 bytes */
    int    a, c, e;
    char b, d;
} array_of_struct [100];

struct {        /* 1400 bytes */
    int    a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;


struct {        /* 1200 bytes */
    int   a, c, e;
} hybrid_struct_of_array_ace[100];


struct {        /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

The efficiency of such optimizations depends on usage patterns. If the elements of the structure are all accessed together but the access pattern of the array is random, then ARRAY_OF_STRUCT avoids unnecessary prefetch even though it wastes memory.

However, if the access pattern of the array exhibits locality (for example, if the array index is being swept through) then processors with hardware prefetchers will prefetch data from STRUCT_OF_ARRAY, even if the elements of the structure are accessed together.

When the elements of the structure are not accessed with equal frequency, such as when element A is accessed ten times more often than the other entries, then STRUCT_OF_ARRAY not only saves memory, but it also prevents fetching unnecessary data items B, C, D, and E.

Using STRUCT_OF_ARRAY also enables the use of the SIMD data types by the programmer and the compiler.

Note that STRUCT_OF_ARRAY can have the disadvantage of requiring more independent memory stream references. This can require the use of more prefetches and additional address generation calculations. It can also have an impact on DRAM page access efficiency. An alternative, HYBRID_STRUCT_OF_ARRAY blends the two approaches. In this case, only 2 separate address streams are generated and referenced: 1 for HYBRID_STRUCT_OF_ARRAY_ACE and 1 for HYBRID_STRUCT_OF_ARRAY_BD. The second alterative also prevents fetching unnecessary data — assuming that (1) the variables A, C and E are always used together, and (2) the variables B and D are always used together, but not at the same time as A, C and E.

The hybrid approach ensures:

- Simpler/fewer address generations than STRUCT_OF_ARRAY.
- Fewer streams, which reduces DRAM page misses.
- Fewer prefetches due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

Assembly/Compiler Coding Rule 47. (H impact, M generality) *Try to arrange data structures such that they permit sequential access.*

If the data is arranged into a set of streams, the automatic hardware prefetcher can prefetch data that will be needed by the application, reducing the effective memory latency. If the data is accessed in a non-sequential manner, the automatic hardware prefetcher cannot prefetch the data. The prefetcher can recognize up to eight concurrent streams. See Chapter 9 for more information on the hardware prefetcher.

User/Source Coding Rule 7. (M impact, L generality) *Beware of false sharing within a cache line (64 bytes).*

3.6.6 Stack Alignment

Performance penalty of unaligned access to the stack happens when a memory reference splits a cache line. This means that one out of eight spatially consecutive unaligned quadword accesses is always penalized, similarly for one out of 4 consecutive, non-aligned double-quadword accesses, etc.

Aligning the stack may be beneficial any time there are data objects that exceed the default stack alignment of the system. For example, on 32/64bit Linux, and 64bit Windows, the default stack alignment is 16 bytes, while 32bit Windows is 4 bytes.

Assembly/Compiler Coding Rule 48. (H impact, M generality) *Make sure that the stack is aligned at the largest multi-byte granular data type boundary matching the register width.*

Aligning the stack typically requires the use of an additional register to track across a padded area of unknown amount. There is a trade-off between causing unaligned memory references that spanned across a cache line and causing extra general purpose register spills.

The assembly level technique to implement dynamic stack alignment may depend on compilers, and specific OS environment. The reader may wish to study the assembly output from a compiler of interest.

Example 3-45. Examples of Dynamical Stack Alignment

```
// 32-bit environment
    push     ebp ; save ebp
    mov      ebp, esp ; ebp now points to incoming parameters
    andl     esp, $-<N> ;align esp to N byte boundary
    sub      esp, $<stack_size>; reserve space for new stack frame
    .         ; parameters must be referenced off of ebp
    mov      esp, ebp ; restore esp
    pop      ebp ; restore ebp


// 64-bit environment
    sub      esp, $<stack_size +N>
    mov      r13, $<offset_of_aligned_section_in_stack>
    andl     r13, $-<N> ; r13 point to aligned section in stack
    .         ;use r13 as base for aligned data
```

If for some reason it is not possible to align the stack for 64-bits, the routine should access the parameter and save it into a register or known aligned storage, thus incurring the penalty only once.

3.6.7 Capacity Limits and Aliasing in Caches

There are cases in which addresses with a given stride will compete for some resource in the memory hierarchy.

Typically, caches are implemented to have multiple ways of set associativity, with each way consisting of multiple sets of cache lines (or sectors in some cases). Multiple memory references that compete for the same set of each way in a cache can cause a capacity issue. There are aliasing conditions that apply to specific microarchitectures. Note that first-level cache lines are 64 bytes. Thus, the least significant 6 bits are not considered in alias comparisons.

3.6.8 Mixing Code and Data

The aggressive prefetching and pre-decoding of instructions by Intel processors have two related effects:

- Self-modifying code (SMC) works correctly, according to the Intel architecture processor requirements, but incurs a significant performance penalty. Avoid self-modifying code if possible.
- Placing writable data in the code segment might be impossible to distinguish from self-modifying code. Writable data in the code segment might suffer the same performance penalty as self-modifying code.

Assembly/Compiler Coding Rule 49. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch.*

Tuning Suggestion 1. *In rare cases, a performance problem may be caused by executing data on a code page as instructions. This is very likely to happen when execution is following an indirect branch that is not resident in the trace cache. If this is clearly causing a performance problem, try moving the data elsewhere, or inserting an illegal opcode or a* PAUSE *instruction immediately after the indirect branch. Note that the latter two alternatives may degrade performance in some circumstances.*

Assembly/Compiler Coding Rule 50. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate 4-KByte pages or on separate aligned 1-KByte subpages.*

3.6.8.1 Self-Modifying Code (SMC)

Self-modifying code (SMC) that ran correctly on Pentium III processors and prior implementations will run correctly on subsequent implementations. SMC and cross-modifying code (when multiple processors in a multiprocessor system are writing to a code page) should be avoided when high performance is desired.

Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written. In addition, sharing a page containing directly or speculatively executed code with another processor as a data page can trigger an SMC condition causing the entire pipeline of the machine and the trace cache to be cleared.

Dynamic code need not cause the SMC condition if the code written fills up a data page before that page is accessed as code. Dynamically-modified code (for example, from target fix-ups) is likely to suffer from the SMC condition and should be avoided where possible. Avoid the condition by introducing indirect branches and using data tables on data pages (not code pages) using register-indirect calls.

3.6.8.2 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. Example 3-46a shows one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 3-46b shows an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 3-46. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP does not corrupt the RSB
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX


b) Using matched call/ret pair


        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

3.6.9 Write Combining

Write combining (WC) improves performance in two ways:

- On a write miss to the first-level cache, it allows multiple stores to the same cache line to occur before that cache line is read for ownership (RFO) from further out in the cache/memory hierarchy. Then the rest of line is read, and the bytes that have not been written are combined with the unmodified bytes in the returned line.
- Write combining allows multiple writes to be assembled and written further out in the cache hierarchy as a unit. This saves port and bus traffic. Saving traffic is particularly important for avoiding partial writes to uncached memory.

Processors based on Intel Core microarchitecture have eight write-combining buffers in each core. Beginning with Nehalem microarchitecture, there are 10 buffers available for write-combining. Beginning with Ice Lake Client microarchitecture, there are 12 buffers available for write-combining.

Assembly/Compiler Coding Rule 51. (H impact, L generality) *If an inner loop writes to more than four arrays (four distinct cache lines), apply loop fission to break up the body of the loop such that only four arrays are being written to in each iteration of each of the resulting loops.*

Write combining buffers are used for stores of all memory types. They are particularly important for writes to uncached memory: writes to different parts of the same cache line can be grouped into a single, full-cache-line bus transaction instead of going across the bus (since they are not cached) as several partial writes. Avoiding partial writes can have a significant impact on bus bandwidth-bound graphics applications, where graphics buffers are in uncached memory. Separating writes to uncached memory and writes to writeback memory into separate phases can assure that the write combining buffers can fill before getting evicted by other write traffic. Eliminating partial write transactions has been found to have performance impact on the order of 20% for some applications. Because the cache lines are 64 bytes, a write to the bus for 63 bytes will result in partial bus transactions.

When coding functions that execute simultaneously on two threads, reducing the number of writes that are allowed in an inner loop will help take full advantage of write-combining store buffers. For write-combining buffer recommendations for Intel® Hyper-Threading Technology (Intel® HT), see Chapter 11.

Store ordering and visibility are also important issues for write combining. When a write to a write-combining buffer for a previously-unwritten cache line occurs, there will be a read-for-ownership (RFO). If a subsequent write happens to another write-combining buffer, a separate RFO may be caused for that cache line. Subsequent writes to the first cache line and write-combining buffer will be delayed until the second RFO has been serviced to guarantee properly ordered visibility of the writes. If the memory type for the writes is write-combining, there will be no RFO since the line is not cached, and there is no such delay. For details on write-combining, see Chapter 9, “Optimizing Cache Usage”

3.6.10 Locality Enhancement

Locality enhancement can reduce data traffic originating from an outer-level sub-system in the cache/memory hierarchy. This is to address the fact that the access-cost in terms of cycle-count from an outer level will be more expensive than from an inner level. Typically, the cycle-cost of accessing a given cache level (or memory system) varies across different microarchitectures, processor implementations, and platform components. It may be sufficient to recognize the relative data access cost trend by locality rather than to follow a large table of numeric values of cycle-costs, listed per locality, per processor/platform implementations, etc. The general trend is typically that access cost from an outer sub-system may be approximately 3-10X more expensive than accessing data from the immediate inner level in the cache/memory hierarchy, assuming similar degrees of data access parallelism.

Thus locality enhancement should start with characterizing the dominant data traffic locality. Appendix A, “Application Performance Tools” describes some techniques that can be used to determine the dominant data traffic locality for any workload.

Even if cache miss rates of the last level cache may be low relative to the number of cache references, processors typically spend a sizable portion of their execution time waiting for cache misses to be serviced. Reducing cache misses by enhancing a program’s locality is a key optimization. This can take several forms:

- Blocking to iterate over a portion of an array that will fit in the cache (with the purpose that subsequent references to the data-block [or tile] will be cache hit references).
- Loop interchange to avoid crossing cache lines or page boundaries.
- Loop skewing to make accesses contiguous.

Locality enhancement to the last level cache can be accomplished with sequencing the data access pattern to take advantage of hardware prefetching. This can also take several forms:

- Transformation of a sparsely populated multi-dimensional array into a one-dimension array such that memory references occur in a sequential, small-stride pattern that is friendly to the hardware prefetch.
- Optimal tile size and shape selection can further improve temporal data locality by increasing hit rates into the last level cache and reduce memory traffic resulting from the actions of hardware prefetching (see Section 9.5.11).

It is important to avoid operations that work against locality-enhancing techniques. Using the lock prefix heavily can incur large delays when accessing memory, regardless of whether the data is in the cache or in system memory.

User/Source Coding Rule 8. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing, and packing are best done by the compiler. Optimize data structures either to fit in one-half of the first-level cache or in the second-level cache; turn on loop optimizations in the compiler to enhance locality for nested loops.*

Optimizing for one-half of the first-level cache will bring the greatest performance benefit in terms of cycle-cost per data access. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.

3.6.11 Non-Temporal Store Bus Traffic

Peak system bus bandwidth is shared by several types of bus activities, including reads (from memory), reads for ownership (of a cache line), and writes. The data transfer rate for bus write transactions is higher if 64 bytes are written out to the bus at a time.

Typically, bus writes to Writeback (WB) memory must share the system bus bandwidth with read-for-ownership (RFO) traffic. Non-temporal stores do not require RFO traffic; they do require care in managing the access patterns in order to ensure 64 bytes are evicted at once (rather than evicting several chunks).

Although the data bandwidth of full 64-byte bus writes due to non-temporal stores is twice that of bus writes to WB memory, transferring several chunks wastes bus request bandwidth and delivers significantly lower data bandwidth. This difference is depicted in Examples 3-47 and 3-48.

Example 3-47. Using Non-Temporal Stores and 64-byte Bus Write Transactions

```
#define STRIDESIZE 256
lea ecx, p64byte_Aligned
mov edx, ARRAY_LEN
xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0
movntps XMMWORD ptr [ecx + eax+48], xmm0
; 64 bytes is written in one bus transaction
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

Example 3-48. On-temporal Stores and Partial Bus Write Transactions

```
#define STRIDESIZE 256
Lea ecx, p64byte_Aligned
Mov edx, ARRAY_LEN
Xor eax, eax
slloop:
movntps XMMWORD ptr [ecx + eax], xmm0
movntps XMMWORD ptr [ecx + eax+16], xmm0
movntps XMMWORD ptr [ecx + eax+32], xmm0


; Storing 48 bytes results in several bus partial transactions
add eax, STRIDESIZE
cmp eax, edx
jl slloop
```

3.7 PREFETCHING

Recent Intel processor families employ several prefetching mechanisms to accelerate the movement of data or code and improve performance:

- Hardware instruction prefetcher.
- Software prefetch for data.
- Hardware prefetch for cache lines of data or instructions.

3.7.1 Hardware Instruction Fetching and Software Prefetching

Software prefetching requires a programmer to use PREFETCH hint instructions and anticipate some suitable timing and location of cache misses.

Software PREFETCH operations work the same way as do load from memory operations, with the following exceptions:

- Software PREFETCH instructions retire after virtual to physical address translation is completed.
- If an exception, such as page fault, is required to prefetch the data, then the software prefetch instruction retires without prefetching data.
- Avoid specifying a NULL address for software prefetches.

3.7.2 Hardware Prefetching for First-Level Data Cache

Example 3-49 depicts a technique to trigger hardware prefetch. The code demonstrates traversing a linked list and performing some computational work on two members of each element that reside in two different cache lines. Each element is of size 192 bytes. The total size of all elements is larger than can be fitted in the L2 cache.

Example 3-49. Using DCU Hardware Prefetch

Original code	Modified sequence benefit from prefetch
mov ebx, DWORD PTR [First] xor eax, eax scan_list: mov eax, [ebx+4] mov ecx, 60 do_some_work_1: add eax, eax and eax, 6 sub ecx, 1 jnz do_some_work_1 mov eax, [ebx+64] mov ecx, 30 do_some_work_2: add eax, eax and eax, 6 sub ecx, 1 jnz do_some_work_2	mov ebx, DWORD PTR [First] xor eax, eax scan_list: mov eax, [ebx+4] mov eax, [ebx+4] mov eax, [ebx+4] mov ecx, 60 do_some_work_1: add eax, eax and eax, 6 sub ecx, 1 jnz do_some_work_1 mov eax, [ebx+64] mov ecx, 30 do_some_work_2: add eax, eax and eax, 6 sub ecx, 1 jnz do_some_work_2
mov ebx, [ebx] test ebx, ebx jnz scan_list	mov ebx, [ebx] test ebx, ebx jnz scan_list

The additional instructions to load data from one member in the modified sequence can trigger the DCU hardware prefetch mechanisms to prefetch data in the next cache line, enabling the work on the second member to complete sooner.

Software can gain from the first-level data cache prefetchers in two cases:

- If data is not in the second-level cache, the first-level data cache prefetcher enables early trigger of the second-level cache prefetcher.
- If data is in the second-level cache and not in the first-level data cache, then the first-level data cache prefetcher triggers earlier data bring-up of sequential cache line to the first-level data cache.

There are situations that software should pay attention to a potential side effect of triggering unnecessary DCU hardware prefetches. If a large data structure with many members spanning many cache lines is accessed in ways that only a few of its members are actually referenced, but there are multiple pair accesses to the same cache line. The DCU hardware prefetcher can trigger fetching of cache lines that are not needed. In Example 3-50, references to the "Pts" array and "AltPts" will trigger DCU prefetch to fetch additional cache lines that won't be needed. If significant negative performance impact is detected due to DCU hardware prefetch on a portion of the code, software can try to reduce the size of that contemporaneous working set to be less than half of the L2 cache.

Example 3-50. Avoid Causing DCU Hardware Prefetch to Fetch Unneeded Lines

```
while ( CurrBond != NULL )
    {
    MyATOM  *a1 = CurrBond->At1 ;
    MyATOM  *a2 = CurrBond->At2 ;


    if ( a1->CurrStep <= a1->LastStep &&
       a2->CurrStep <= a2->LastStep
        )
         {
         a1->CurrStep++ ;
         a2->CurrStep++ ;

         double  ux = a1->Pts[0].x - a2->Pts[0].x ;
         double  uy = a1->Pts[0].y - a2->Pts[0].y ;
         double  uz = a1->Pts[0].z - a2->Pts[0].z ;
         a1->AuxPts[0].x += ux ;
         a1->AuxPts[0].y += uy ;
         a1->AuxPts[0].z += uz ;


         a2->AuxPts[0].x += ux ;
         a2->AuxPts[0].y += uy ;
         a2->AuxPts[0].z += uz ;
         } ;
    CurrBond = CurrBond->Next ;
    } ;
```

To fully benefit from these prefetchers, organize and access the data using one of the following methods:

Method 1:

- Organize the data so consecutive accesses can usually be found in the same 4-KByte page.
- Access the data in constant strides forward or backward IP Prefetcher.

Method 2:

- Organize the data in consecutive lines.
- Access the data in increasing addresses, in sequential cache lines.

Example 3-51 demonstrates accesses to sequential cache lines that can benefit from the first-level cache prefetcher.

Example 3-51. Technique for Using L1 Hardware Prefetch

```
unsigned int *p1, j, a, b;
for (j = 0; j < num; j += 16)
{
a = p1[j];
b = p1[j+1];
// Use these two values
}
```

By elevating the load operations from memory to the beginning of each iteration, it is likely that a significant part of the latency of the pair cache line transfer from memory to the second-level cache will be in parallel with the transfer of the first cache line.

The IP prefetcher uses only the lower 8 bits of the address to distinguish a specific address. If the code size of a loop is bigger than 256 bytes, two loads may appear similar in the lowest 8 bits and the IP prefetcher will be restricted. Therefore, if you have a loop bigger than 256 bytes, make sure that no two loads have the same lowest 8 bits in order to use the IP prefetcher.

3.7.3 Hardware Prefetching for Second-Level Cache

The Intel Core microarchitecture contains two second-level cache prefetchers:

- **Streamer** — Loads data or instructions from memory to the second-level cache. To use the streamer, organize the data or instructions in blocks of 128 bytes, aligned on 128 bytes. The first access to one of the two cache lines in this block while it is in memory triggers the streamer to prefetch the pair line. To software, the L2 streamer's functionality is similar to the adjacent cache line prefetch mechanism found in processors based on Intel NetBurst microarchitecture.
- **Data prefetch logic (DPL)** — DPL and L2 Streamer are triggered only by writeback memory type. They prefetch only inside page boundary (4 KBytes). Both L2 prefetchers can be triggered by software prefetch instructions and by prefetch request from DCU prefetchers. DPL can also be triggered by read for ownership (RFO) operations. The L2 Streamer can also be triggered by DPL requests for L2 cache misses.

Software can gain from organizing data both according to the instruction pointer and according to line strides. For example, for matrix calculations, columns can be prefetched by IP-based prefetches, and rows can be prefetched by DPL and the L2 streamer.

3.7.4 Cacheability Instructions

SSE2 provides additional cacheability instructions that extend those provided in SSE. The new cacheability instructions include:

- New streaming store instructions.
- New cache line flush instruction.
- New memory fencing instructions.

For more information, see Chapter 9

3.7.5 REP Prefix and Data Movement

The REP prefix is commonly used with string move instructions for memory related library functions such as MEMCPY (using REP MOVSD) or MEMSET (using REP STOS). These STRING/MOV instructions with the REP prefixes are implemented in MS-ROM and have several implementation variants with different performance levels.

The specific variant of the implementation is chosen at execution time based on data layout, alignment and the counter (ECX) value. For example, MOVSB/STOSB with the REP prefix should be used with counter value less than or equal to three for best performance.

String MOVE/STORE instructions have multiple data granularities. For efficient data movement, larger data granularities are preferable. This means better efficiency can be achieved by decomposing an arbitrary counter value into a number of doublewords plus single byte moves with a count value less than or equal to 3.

Because software can use SIMD data movement instructions to move 16 bytes at a time, the following paragraphs discuss general guidelines for designing and implementing high-performance library functions such as MEMCPY(), MEMSET(), and MEMMOVE(). Four factors are to be considered:

- **Throughput per iteration** — If two pieces of code have approximately identical path lengths, efficiency favors choosing the instruction that moves larger pieces of data per iteration. Also, smaller code size per iteration will in general reduce overhead and improve throughput. Sometimes, this may involve a comparison of the relative overhead of an iterative loop structure versus using REP prefix for iteration.
- **Address alignment** — Data movement instructions with highest throughput usually have alignment restrictions, or they operate more efficiently if the destination address is aligned to its natural data size. Specifically, 16-byte moves need to ensure the destination address is aligned to 16-byte boundaries, and 8-bytes moves perform better if the destination address is aligned to 8-byte boundaries. Frequently, moving at doubleword granularity performs better with addresses that are 8-byte aligned.
- **REP string move vs. SIMD move** — Implementing general-purpose memory functions using SIMD extensions usually requires adding some prolog code to ensure the availability of SIMD instructions, preamble code to facilitate aligned data movement requirements at runtime. Throughput comparison must also take into consideration the overhead of the prolog when considering a REP string implementation versus a SIMD approach.
- **Cache eviction** — If the amount of data to be processed by a memory routine approaches half the size of the last level on-die cache, temporal locality of the cache may suffer. Using streaming store instructions (for example: MOVNTQ, MOVNTDQ) can minimize the effect of flushing the cache. The threshold to start using a streaming store depends on the size of the last level cache. Determine the size using the deterministic cache parameter leaf of CPUID.

  Techniques for using streaming stores for implementing a MEMSET()-type library must also consider that the application can benefit from this technique only if it has no immediate need to reference the target addresses. This assumption is easily upheld when testing a streaming-store implementation on a micro-benchmark configuration, but violated in a full-scale application situation.

When applying general heuristics to the design of general-purpose, high-performance library routines, the following guidelines can are useful when optimizing an arbitrary counter value N and address alignment. Different techniques may be necessary for optimal performance, depending on the magnitude of N:

- When N is less than some small count (where the small count threshold will vary between microarchitectures -- empirically, 8 may be a good value when optimizing for Intel NetBurst microarchitecture), each case can be coded directly without the overhead of a looping structure. For example, 11 bytes can be processed using two MOVSD instructions explicitly and a MOVSB with REP counter equaling 3.
- When N is not small but still less than some threshold value (which may vary for different micro-architectures, but can be determined empirically), an SIMD implementation using run-time CPUID and alignment prolog will likely deliver less throughput due to the overhead of the prolog. A REP string implementation should favor using a REP string of doublewords. To improve address alignment, a small piece of prolog code using MOVSB/STOSB with a count less than 4 can be used to peel off the non-aligned data moves before starting to use MOVSD/STOSD.

- When N is less than half the size of last level cache, throughput consideration may favor either:
  - An approach using a REP string with the largest data granularity because a REP string has little overhead for loop iteration, and the branch misprediction overhead in the prolog/epilogue code to handle address alignment is amortized over many iterations.
  - An iterative approach using the instruction with largest data granularity, where the overhead for SIMD feature detection, iteration overhead, and prolog/epilogue for alignment control can be minimized. The trade-off between these approaches may depend on the microarchitecture.

  An example of MEMSET() implemented using stosd for arbitrary counter value with the destination address aligned to doubleword boundary in 32-bit mode is shown in Example 3-52.
- When N is larger than half the size of the last level cache, using 16-byte granularity streaming stores with prolog/epilog for address alignment will likely be more efficient, if the destination addresses will not be referenced immediately afterwards.

Example 3-52. REP STOSD with Arbitrary Count Size and 4-Byte-Aligned Destination

A 'C' example of Memset()	Equivalent Implementation Using REP STOSD

```
void memset(void *dst,int c,size_t size)
{
char *d = (char *)dst;
size_t i;
for (i=0;i<size;i++)
    *d++ = (char)c;
}
```
	```
	push edi
	movzx eax, byte ptr [esp+12]
	mov ecx, eax
	shl ecx, 8
	or ecx, eax
	mov ecx, eax
	shl ecx, 16
	or eax, ecx
	mov edi, [esp+8]          ; 4-byte aligned
	mov ecx, [esp+16]         ; byte count
	shr ecx, 2                ; do dword
	cmp ecx, 127
	jle _main
	test edi, 4
	jz _main
	stosd                  ;peel off one dword
	dec ecx
	```

	```
	_main:                 ; 8-byte aligned
	rep stosd
	mov ecx, [esp + 16]
	and ecx, 3              ; do count <= 3
	rep stosb               ; optimal with <= 3
	pop edi
	ret
	```

Memory routines in the runtime library generated by Intel compilers are optimized across a wide range of address alignments, counter values, and microarchitectures. In most cases, applications should take advantage of the default memory routines provided by Intel compilers.

In some situations, the byte count of the data is known by the context (as opposed to being known by a parameter passed from a call), and one can take a simpler approach than those required for a general-purpose library routine. For example, if the byte count is also small, using REP MOVSB/STOSB with a count less than four can ensure good address alignment and loop-unrolling to finish the remaining data; using MOVSD/STOSD can reduce the overhead associated with iteration.

Using a REP prefix with string move instructions can provide high performance in the situations described above. However, using a REP prefix with string scan instructions (SCASB, SCASW, SCASD, SCASQ) or compare instructions (CMPSB, CMPSW, SMPSD, SMPSQ) is not recommended for high performance. Consider using SIMD instructions instead.

3.7.6 Enhanced REP MOVSB and STOSB Operation

Beginning with processors based on Ivy Bridge microarchitecture, REP string operation using MOVSB and STOSB can provide both flexible and high-performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.[bit 9] = 1.

3.7.6.1 Fast Short REP MOVSB

Beginning with processors based on Ice Lake Client microarchitecture, REP MOVSB performance of short operations is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. Support for fast-short REP MOVSB is enumerated by the CPUID feature flag: CPUID [EAX=7H, ECX=0H).EDX.FAST_SHORT_REP_MOVSB[bit 4] = 1. There is no change in the REP STOS performance.

3.7.6.2 Memcpy Considerations

The interface for the standard library function memcpy introduces several factors (e.g. length, alignment of the source buffer and destination) that interact with microarchitecture to determine the performance characteristics of the implementation of the library function. Two of the common approaches to implement memcpy are driven from small code size vs. maximum throughput. The former generally uses REP MOVSD+B (see Section 3.7.5), while the latter uses SIMD instruction sets and has to deal with additional data alignment restrictions.

For processors supporting enhanced REP MOVSB/STOSB, implementing memcpy with REP MOVSB will provide even more compact benefits in code size and better throughput than using the combination of REP MOVSD+B. For processors based on Ivy Bridge microarchitecture, implementing memcpy using Enhanced REP MOVSB and STOSB might not reach the same level of throughput as using 256-bit or 128-bit AVX alternatives, depending on length and alignment factors.

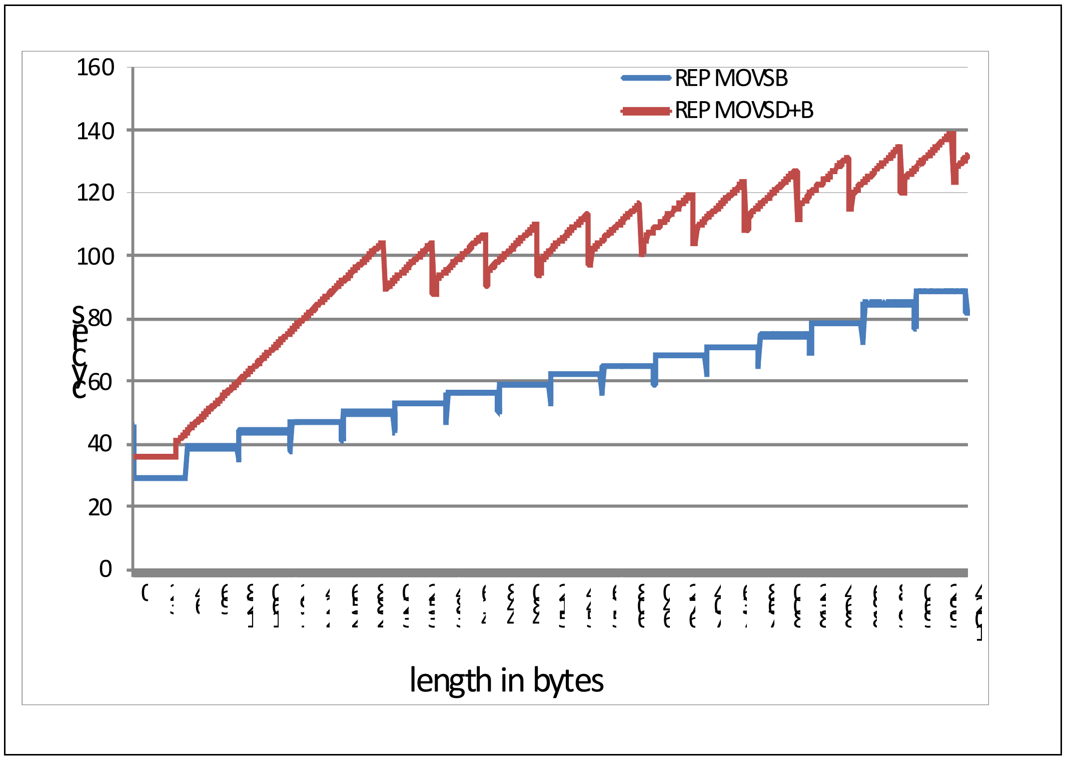


Figure 3-3. Memcpy Performance Comparison for Lengths up to 2KB

Figure 3-3 depicts the relative performance of memcpy implementation on a third-generation Intel Core processor using Enhanced REP MOVSB and STOSB versus REP MOVSD+B, for alignment conditions when both the source and destination addresses are aligned to a 16-Byte boundary and the source region does not overlap with the destination region. Using Enhanced REP MOVSB and STOSB always delivers better performance than using REP MOVSD+B. If the length is a multiple of 64, it can produce even higher performance. For example, copying 65-128 bytes takes 40 cycles, while copying 128 bytes needs only 35 cycles.

If an application wishes to bypass standard memcpy library implementation with its own custom implementation and have freedom to manage the buffer length allocation for both source and destination, it may be worthwhile to manipulate the lengths of its memory copy operation to be multiples of 64 to take advantage the code size and performance benefit of Enhanced REP MOVSB and STOSB.

The performance characteristic of implementing a general-purpose memcpy library function using a SIMD register is significantly more colorful than an equivalent implementation using a general-purpose register, depending on length, instruction set selection between SSE2, 128-bit AVX, 256-bit AVX, relative alignment of source/destination, and memory address alignment granularities/boundaries, etc.

Hence comparing performance characteristics between a memcpy using Enhanced REP MOVSB and STOSB versus a SIMD implementation is highly dependent on the particular SIMD implementation. The remainder of this section discusses the relative performance of memcpy using Enhanced REP MOVSB and STOSB versus unpublished, optimized 128-bit AVX implementation of memcpy to illustrate the hardware capability of Ivy Bridge microarchitecture.

Table 3-5. Relative Performance of Memcpy() Using Enhanced REP MOVSB and STOSB Vs. 128-bit AVX

Range of Lengths (bytes)	**<128**	**128 to 2048**	**2048 to 4096**
Memcpy_ERMSB/Memcpy_AVX128	0x7X	1X	1.02X

Table 3-5 shows the relative performance of the Memcpy function implemented using enhanced REP MOVSB versus 128-bit AVX for several ranges of memcpy lengths, when both the source and destination addresses are 16-byte aligned and the source region and destination region do not overlap. For memcpy length less than 128 bytes, using Enhanced REP MOVSB and STOSB is slower than what’s possible using 128-bit AVX, due to internal start-up overhead in the REP string.

For situations with address misalignment, memcpy performance will generally be reduced relative to the 16-byte alignment scenario (see Table 3-6).

Table 3-6. Effect of Address Misalignment on Memcpy() Performance

Address Misalignment	**Performance Impact**
Source Buffer	The impact on Enhanced REP MOVSB and STOSB implementation versus 128-bit AVX is similar.
Destination Buffer	The impact on Enhanced REP MOVSB and STOSB implementation can be 25% degradation, while 128-bit AVX implementation of memcpy may degrade only 5%, relative to 16-byte aligned scenario.

Memcpy() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit SIMD integer data-path in Haswell microarchitecture. See Section 15.16.3.

3.7.6.3 Memmove Considerations

When there is an overlap between the source and destination regions, software may need to use memmove instead of memcpy to ensure correctness. It is possible to use REP MOVSB in conjunction with the direction flag (DF) in a memmove() implementation to handle situations where the latter part of the source region overlaps with the beginning of the destination region. However, setting the DF to force REP MOVSB to copy bytes from high towards low addresses will experience significant performance degradation.

When using Enhanced REP MOVSB and STOSB to implement memmove function, one can detect the above situation and handle first the rear chunks in the source region that will be written to as part of the destination region, using REP MOVSB with the DF=0, to the non-overlapping region of the destination. After the overlapping chunks in the rear section are copied, the rest of the source region can be processed normally, also with DF=0.

3.7.6.4 Memset Considerations

The consideration of code size and throughput also applies for memset() implementations. For processors supporting Enhanced REP MOVSB and STOSB, using REP STOSB will again deliver more compact code size and significantly better performance than the combination of STOSD+B technique described in Section 3.7.5.

When the destination buffer is 16-byte aligned, memset() using Enhanced REP MOVSB and STOSB can perform better than SIMD approaches. When the destination buffer is misaligned, memset() performance using Enhanced REP MOVSB and STOSB can degrade about 20% relative to aligned case, for processors based on Ivy Bridge microarchitecture. In contrast, SIMD implementation of memset() will experience smaller degradation when the destination is misaligned.

Memset() implemented with Enhanced REP MOVSB and STOSB can benefit further from the 256-bit data path in Haswell microarchitecture. see Section 15.16.3.3.

3.8 REP STRING OPERATIONS

Several REP string performance enhancements are available beginning with processors based on Golden Cove microarchitecture.

3.8.1 Fast Zero Length REP MOVSB

REP MOVSB performance of zero length operations is enhanced. The latency of a zero length REP MOVSB is now the same as the latency of lengths 1 to 128 bytes. When both Fast Short REP MOVSB and Fast Zero Length REP MOVSB features are enabled, REP MOVSB performance is flat 9 cycles per operation, for all strings 0-128 byte long whose source and destination operands reside in the processor first level cache.

Support for fast zero-length REP MOVSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_ZERO_LENGTH_REP_MOVSB[bit 10] = 1.

3.8.2 Fast Short REP STOSB

REP STOSB performance of short operations is enhanced. The enhancement applies to string lengths between 0 and 128 bytes long. When Fast Short REP STOSB feature is enabled, REP STOSB performance is flat 12 cycles per operation, for all strings 0-128 byte long whose destination operand resides in the processor first level cache.

Support for fast-short REP STOSB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_STOSB[bit 11] = 1.

3.8.3 Fast Short REP CMPSB and SCASB

REP CMPSB and SCASB performance is enhanced. The enhancement applies to string lengths between 1 and 128 bytes long. When the Fast Short REP CMPSB and SCASB feature is enabled, REP CMPSB and REP SCASB performance is flat 15 cycles per operation, for all strings 1-128 byte long whose two source operands reside in the processor first level cache.

Support for fast short REP CMPSB and SCASB is enumerated by the CPUID feature flag:

CPUID.07H.01H:EAX.FAST_SHORT_REP_CMPSB_SCASB[bit 12] = 1.

3.9 FLOATING-POINT CONSIDERATIONS

When programming floating-point applications, it is best to start with a high-level programming language such as C, C++, or Fortran. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code, the compiler may need some assistance.

3.9.1 Guidelines for Optimizing Floating-Point Code

User/Source Coding Rule 9. (M impact, M generality) *Enable the compiler's use of Intel SSE, Intel SSE2, Intel AVX, Intel AVX2, and possibly more advanced SIMD instruction sets (Intel AVX-512) with appropriate switches. Favor scalar SIMD code generation to replace x87 code generation.*

Follow this procedure to investigate the performance of your floating-point application:

- Understand how the compiler handles floating-point code.
- Look at the assembly dump and see what transforms are already performed on the program.
- Study the loop nests in the application that dominate the execution time.
- Determine why the compiler is not creating the fastest code.
- See if there is a dependence that can be resolved.
- Determine the problem area: bus bandwidth, cache locality, trace cache bandwidth, or instruction latency. Focus on optimizing the problem area. For example, adding PREFETCH instructions will not help if the bus is already saturated. If trace cache bandwidth is the problem, added prefetch µops may degrade performance.

Also, in general, follow the general coding recommendations discussed in this chapter, including:

- Blocking the cache.
- Using prefetch.
- Enabling vectorization.
- Unrolling loops.

User/Source Coding Rule 10. (H impact, ML generality) *Make sure your application stays in range to avoid denormal values, underflows.*

Out-of-range numbers cause very high overhead.

When converting floating-point values to 16-bit, 32-bit, or 64-bit integers using truncation, the instructions CVTTSS2SI and CVTTSD2SI are recommended over instructions that access x87 FPU stack. This avoids changing the rounding mode.

User/Source Coding Rule 11. (M impact, ML generality) *Usually, math libraries take advantage of the transcendental instructions (for example, FSIN) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider an alternate, software-based approach, such as a look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision and the size of the look-up table, and by taking advantage of the parallelism of the Intel SSE and the Intel SSE2 instructions.*

3.9.2 Floating-Point Modes and Exceptions

When working with floating-point numbers, high-speed microprocessors frequently must deal with situations that need special handling in hardware or code.

3.9.2.1 Floating-Point Exceptions

The most frequent cause of performance degradation is the use of masked floating-point exception conditions such as:

- Arithmetic overflow.
- Arithmetic underflow.
- Denormalized operand.

Refer to Chapter 4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for definitions of overflow, underflow and denormal exceptions.

Denormalized floating-point numbers impact performance in two ways:

- Directly when are used as operands.
- Indirectly when are produced as a result of an underflow situation.

If a floating-point application never underflows, the denormals can only come from floating-point constants.

User/Source Coding Rule 12. (H impact, ML generality) *Denormalized floating-point constants should be avoided as much as possible.*

Denormal and arithmetic underflow exceptions can occur during the execution of x87 instructions or Intel SSE/Intel SSE2/Intel SSE3 instructions. Processors based on Intel NetBurst microarchitecture handle these exceptions more efficiently when executing Intel SSE/Intel SSE2/Intel SSE3 instructions and when speed is more important than complying with the IEEE standard. The following paragraphs give recommendations on how to optimize your code to reduce performance degradations related to floating-point exceptions.

3.9.2.2 Dealing with Floating-Point Exceptions in x87 FPU Code

Every special situation listed in Section 3.9.2.1 is costly in terms of performance. For that reason, x87 FPU code should be written to avoid these situations.

There are basically three ways to reduce the impact of overflow/underflow situations with x87 FPU code:

- Choose floating-point data types that are large enough to accommodate results without generating arithmetic overflow and underflow exceptions.
- Scale the range of operands/results to reduce as much as possible the number of arithmetic overflow/underflow situations.
- Keep intermediate results on the x87 FPU register stack until the final results have been computed and stored in memory. Overflow or underflow is less likely to happen when intermediate results are kept in the x87 FPU stack (this is because data on the stack is stored in double extended-precision format and overflow/underflow conditions are detected accordingly).
- Denormalized floating-point constants (which are read-only, and hence never change) should be avoided and replaced, if possible, with zeros of the same sign.

3.9.2.3 Floating-Point Exceptions in SSE/SSE2/SSE3 Code

Most special situations that involve masked floating-point exceptions are handled efficiently in hardware. When a masked overflow exception occurs while executing Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 code, processor hardware can handles it without performance penalty.

Underflow exceptions and denormalized source operands are usually treated according to the IEEE 754 specification[1], but this can incur significant performance delay. If a programmer is willing to trade pure IEEE 754 compliance for speed, two non-IEEE 754 compliant modes are provided to speed situations where underflows and input are frequent: FTZ mode and DAZ mode.

1. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229.

When the FTZ mode is enabled, an underflow result is automatically converted to a zero with the correct sign. Although this behavior is not compliant with IEEE 754, it is provided for use in applications where performance is more important than IEEE 754 compliance. Since denormal results are not produced when the FTZ mode is enabled, the only denormal floating-point numbers that can be encountered in FTZ mode are the ones specified as constants (read only).

The DAZ mode is provided to handle denormal source operands efficiently when running a SIMD floating-point application. When the DAZ mode is enabled, input denormals are treated as zeros with the same sign. Enabling the DAZ mode is the way to deal with denormal floating-point constants when performance is the objective.

If departing from the IEEE 754 specification is acceptable and performance is critical, run Intel SSE/Intel SSE2/Intel SSE3/Intel AVX/Intel AVX2/Intel AVX-512 applications with FTZ and DAZ modes enabled.

NOTE

The DAZ mode is available with both the Intel SSE and Intel SSE2 extensions, although the speed improvement expected from this mode is fully realized only in SSE code and later.

3.9.3 Floating-Point Modes

For x87 code, using the FLDCW instruction to change floating modes can be an expensive operation in many cases.

Recent processor generations provide hardware optimization for FLDCW that allows programmers to alternate between two constant values efficiently. For the FLDCW optimization to be effective, the two constant FCW values are only allowed to differ on the following 5 bits in the FCW:

```
FCW[8-9]      ; Precision control
FCW[10-11]    ; Rounding control
FCW[12]       ; Infinity control
```

If programmers need to modify other bits (for example: mask bits) in the FCW, the FLDCW instruction is still an expensive operation.

In situations where an application cycles between three (or more) constant values, FLDCW optimization does not apply, and the performance degradation occurs for each FLDCW instruction.

One solution to this problem is to choose two constant FCW values, take advantage of the optimization of the FLDCW instruction to alternate between only these two constant FCW values, and devise some means to accomplish the task that requires the 3rd FCW value without actually changing the FCW to a third constant value. An alternative solution is to structure the code so that, for periods of time, the application alternates between only two constant FCW values. When the application later alternates between a pair of different FCW values, the performance degradation occurs only during the transition.

It is expected that SIMD applications are unlikely to alternate between FTZ and DAZ mode values. Consequently, the SIMD control word does not have the short latencies that the floating-point control register does. A read of the MXCSR register has a fairly long latency, and a write to the register is a serializing instruction.

There is no separate control word for single and double precision; both use the same modes. Notably, this applies to both FTZ and DAZ modes.

Assembly/Compiler Coding Rule 52. (H impact, M generality) *Minimize changes to bits 8-12 of the floating-point control word. Changes for more than two values (each value being a combination of the following bits: precision, rounding and infinity control, and the rest of bits in FCW) leads to delays that are on the order of the pipeline depth.*

3.9.3.1 Rounding Mode

Many libraries provide float-to-integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. With the Pentium 4 processor, one can use the CVTTSD2SI and CVTTSS2SI instructions to convert operands with truncation without ever needing to change rounding modes. The cost savings of using these instructions over the methods below is enough to justify using Intel SSE and Intel SSE2 wherever possible when truncation is involved.

For x87 floating-point, the FIST instruction uses the rounding mode represented in the floating-point control word (FCW). The rounding mode is generally "round to nearest", so many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the FLDCW instruction. For a change in the rounding, precision, and infinity bits, use the FSTCW instruction to store the floating-point control word. Then use the FLDCW instruction to change the rounding mode to truncation.

In a typical code sequence that changes the rounding mode in the FCW, a FSTCW instruction is usually followed by a load operation. The load operation from memory should be a 16-bit operand to prevent store-forwarding problem. If the load operation on the previously-stored FCW word involves either an 8-bit or a 32-bit operand, this will cause a store-forwarding problem due to mismatch of the size of the data between the store operation and the load operation.

To avoid store-forwarding problems, make sure that the write and read to the FCW are both 16-bit operations.

If there is more than one change to the rounding, precision, and infinity bits, and the rounding mode is not important to the result, use the algorithm in Example 3-53 to avoid synchronization issues, the overhead of the FLDCW instruction, and having to change the rounding mode. Note that the example suffers from a store-forwarding problem which will lead to a performance penalty. However, its performance is still better than changing the rounding, precision, and infinity bits among more than two values.

Example 3-53. Algorithm to Avoid Changing Rounding Mode

```
_fto132proc
    lea     ecx, [esp-8]
    sub     esp, 16             ; Allocate frame
    and     ecx, -8             ; Align pointer on boundary of 8
    fld     st(0)               ; Duplicate FPU stack top

    fistp   qword ptr[ecx]
    fild    qword ptr[ecx]
    mov     edx, [ecx+4]        ; High DWORD of integer
    mov     eax, [ecx]          ; Low DWIRD of integer
    test    eax, eax
    je      integer_QnaN_or_zero

arg_is_not_integer_QnaN:
    fsubp   st(1), st           ; TOS=d-round(d), { st(1) = st(1)-st & pop ST}
    test    edx, edx            ; What's sign of integer
    jns     positive            ; Number is negative
    fstp    dword ptr[ecx]      ; Result of subtraction
    mov     ecx, [ecx]          ; DWORD of diff(single-precision)
    add     esp, 16
    xor     ecx, 80000000h
    add     ecx,7fffffffh       ; If diff<0 then decrement integer
    adc     eax,0               ; INC EAX (add CARRY flag)
    ret
positive:
```

Example 3-53. Algorithm to Avoid Changing Rounding Mode (Contd.)

```
    positive:
    fstp      dword ptr[ecx]     ; 17-18 result of subtraction
    mov       ecx, [ecx]         ; DWORD of diff(single precision)
    add       esp, 16
    add       ecx, 7fffffffh     ; If diff<0 then decrement integer
    sbb       eax, 0             ; DEC EAX (subtract CARRY flag)
    ret
integer_QnaN_or_zero:
    test      edx, 7fffffffh
    jnz       arg_is_not_integer_QnaN
    add  esp, 16
    ret
```

Assembly/Compiler Coding Rule 53. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision, and infinity bits.*

3.9.3.2 Precision

If single precision is adequate, use it instead of double precision. This is true because:

- Single precision operations allow the use of longer SIMD vectors, since more single precision data elements can fit in a register.
- If the precision control (PC) field in the x87 FPU control word is set to single precision, the floating-point divider can complete a single-precision computation much faster than either a double-precision computation or an extended double-precision computation. If the PC field is set to double precision, this will enable those x87 FPU operations on double-precision data to complete faster than extended double-precision computation. These characteristics affect computations including floating-point divide and square root.

Assembly/Compiler Coding Rule 54. (H impact, L generality) *Minimize the number of changes to the precision mode.*

3.9.4 x87 vs. Scalar SIMD Floating-Point Trade-Offs

There are a number of differences between x87 floating-point code and scalar floating-point code (using Intel SSE and Intel SSE2). The following differences should drive decisions about which registers and instructions to use:

- When an input operand for a SIMD floating-point instruction contains values that are less than the representable range of the data type, a denormal exception occurs. This causes a significant performance penalty. An SIMD floating-point operation has a flush-to-zero mode in which the results will not underflow. Therefore subsequent computation will not face the performance penalty of handling denormal input operands. For example, in the case of 3D applications with low lighting levels, using flush-to-zero mode can improve performance by as much as 50% for applications with large numbers of underflows.
- Scalar floating-point SIMD instructions have lower latencies than equivalent x87 instructions. Scalar SIMD floating-point multiply instruction may be pipelined, while x87 multiply instruction is not.
- Although x87 supports transcendental instructions, software library implementation of transcendental function can be faster in many cases.
- x87 supports 80-bit precision, double extended floating-point. SSE support a maximum of 32-bit precision. SSE2 supports a maximum of 64-bit precision.
- Scalar floating-point registers may be accessed directly, avoiding FXCH and top-of-stack restrictions.

- The cost of converting from floating-point to integer with truncation is significantly lower with Intel SSE and Intel SSE2 in the processors based on Intel NetBurst microarchitecture than with either changes to the rounding mode or the sequence prescribed in the Example 3-53.

***Assembly/Compiler Coding Rule 55. (M impact, M generality)** Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Most SSE2 arithmetic operations have shorter latency then their X87 counterpart and they eliminate the overhead associated with the management of the X87 register stack.*

3.9.4.1 Scalar Intel® SSE/Intel® SSE2

In code sequences that have conversions from floating-point to integer, divide single-precision instructions, or any precision change, x87 code generation from a compiler typically writes data to memory in single-precision and reads it again in order to reduce precision. Using Intel SSE/Intel SSE2 scalar code instead of x87 code can generate a large performance benefit using Intel NetBurst microarchitecture and a modest benefit on Intel Core Solo and Intel Core Duo processors.

Recommendation: Use the compiler switch to generate scalar floating-point code using XMM rather than x87 code.

When working with Intel SSE/Intel SSE2 scalar code, pay attention to the need for clearing the content of unused slots in an XMM register and the associated performance impact. For example, loading data from memory with MOVSS or MOVSD causes an extra micro-op for zeroing the upper part of the XMM register.

3.9.4.2 Transcendental Functions

If an application needs to emulate math functions in software for performance or other reasons (see Section 3.9.1), it may be worthwhile to inline math library calls because the CALL and the prologue/epilogue involved with such calls can significantly affect the latency of operations.

3.10 MAXIMIZING PCIE PERFORMANCE

PCIe performance can be dramatically impacted by the size and alignment of upstream reads and writes (read and write transactions issued from a PCIe agent to the host's memory). As a general rule, the best performance, in terms of both bandwidth and latency, is obtained by aligning the start addresses of upstream reads and writes on 64-byte boundaries and ensuring that the request size is a multiple of 64-bytes, with modest further increases in bandwidth when larger multiples (128, 192, 256 bytes) are employed. In particular, a partial write will cause a delay for the following request (read or write).

A second rule is to avoid multiple concurrently outstanding accesses to a single cache line. This can result in a conflict which in turn can cause serialization of accesses that would otherwise be pipelined, resulting in higher latency and/or lower bandwidth. Patterns that violate this rule include sequential accesses (reads or writes) that are not a multiple of 64-bytes, as well as explicit accesses to the same cache line address. Overlapping requests—those with different start addresses but with request lengths that result in overlap of the requests—can have the same effect. For example, a 96-byte read of address 0x00000200 followed by a 64-byte read of address 0x00000240 will cause a conflict—and a likely delay—for the second read.

Upstream writes that are a multiple of 64-byte but are non-aligned will have the performance of a series of partial and full sequential writes. For example, a write of length 128-byte to address 0x00000070 will perform similarly to 3 sequential writes of lengths 16, 64, and 48 to addresses 0x00000070, 0x00000080, and 0x00000100, respectively.

For PCIe cards implementing multi-function devices, such as dual or quad port network interface cards (NICs) or dual-GPU graphics cards, it is important to note that non-optimal behavior by one of those devices can impact the bandwidth and/or latency observed by the other devices on that card. With respect to the behavior described in this section, all traffic on a given PCIe port is treated as if it originated from a single device and function.

For the best PCIe bandwidth:

1. Align start addresses of upstream reads and writes on 64-byte boundaries.
2. Use read and write requests that are a multiple of 64-bytes.
3. Eliminate or avoid sequential and random partial line upstream writes.
4. Eliminate or avoid conflicting upstream reads, including sequential partial line reads.

Techniques for avoiding performance pitfalls include cache line aligning all descriptors and data buffers, padding descriptors that are written upstream to 64-byte alignment, buffering incoming data to achieve larger upstream write payloads, allocating data structures intended for sequential reading by the PCIe device in such a way as to enable use of (multiple of) 64-byte reads. The negative impact of unoptimized reads and writes depends on the specific workload and the microarchitecture on which the product is based.

3.10.1 Optimizing PCIe Performance for Accesses Toward Coherent Memory and MMIO Regions (P2P)

In order to maximize performance for PCIe devices in the processors listed in Table 3-7 the software should determine whether the accesses are toward coherent (system) memory or toward MMIO regions (P2P access to other devices). If the access is toward MMIO region, then software can command HW to set the RO bit in the TLP header, as this would allow hardware to achieve maximum throughput for these types of accesses. For accesses toward coherent memory, software can command HW to clear the RO bit in the TLP header (no RO), as this would allow hardware to achieve maximum throughput for these types of accesses.

Table 3-7. Intel Processor CPU RP Device IDs for Processors Optimizing PCIe Performance

Processor	CPU RP Device IDs
Intel® Xeon processors based on Broadwell microarchitecture	6F01H-6F0EH
Intel® Xeon processors based on Haswell microarchitecture	2F01H-2F0EH

3.11 SCALABILITY WITH CONTENDED LINE ACCESS IN 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

A two-socket system as found in the Sapphire Rapids microarchitecture can have up to 224 (2 sockets x 56 cores/socket x 2 threads/core) hardware threads. Scalability and performance bottlenecks may happen when all of these hardware threads compete for the same address.

3.11.1 Causes of Performance Bottlenecks

When multiple hardware threads go after the same address (for example, AA), this address is queued in the Ingress Queue, with one entry for each hardware thread. Due to the resource limitation of the Ingress Queue, the CPU core is throttled to slow the rate of requests when this queue overflows. This usually occurs with contention for a lock.

3.11.2 Performance Bottleneck Detection

When multiple cores are contending on the same lock, several outstanding requests are mapped to that same address. The Phys_addr_match event can count as such an event. This CHA event increments by one every other cycle when there is more than one outstanding request to the same address.

Here are the PMU event id and Umask for the 2 CHA events that are very useful for detecting contention:

1. Phys_addr_match event: Event id: 0x19, Umask: 0x80

2. CHA_clockticks event: Event id: 0x01, Umask: 0x01

These events have to be measured on a per-CHA basis, and if the ratio of the counts between phys_addr_match to CHA_clockticks is more than 0.15 on any CHA that indicates > 30% of the CHA cycles (2x the ratio as this event can count only once every two cycles) are spent with multiple requests outstanding to the same address.

Here is the recipe to measure these events with Linux Perf:

```
$ sudo perf stat -a -e 'uncore_cha/event=0x19,umask=0x80/,uncore_cha/event=0x1,umask=0x1/' --per-socket
--no-merge -- sleep 30
```

Once confirmed that the ratio of phys_addr_match events to the CHA clockticks is more than 0.15, the next step is figuring out where this may be happening in the code. Intel CPUs provide a PMU mechanism wherein a load operation is randomly selected and tracked through completion, and the true latency is recorded if it is over a given threshold. The threshold value is specified in cycles and must be in the power of 2. In the following "perf mem record" command, define a command to sample all loads that take more than 128 cycles to complete.

```
$ sudo perf mem record -a --ldlat 128 sleep 1
```

Once the above data is collected, execute the following command to process the data collected:

```
$ sudo perf mem report
```

Information similar to the table below will be generated. Such information will include details on hot loads along with data linear address and the actual latency that the load experienced. This can be used to identify the necessary fixes to the code.

Table 3-8. Samples: 365K of Events 'anon group{cpu/mem-loads-aux/,cpu/mem-loads,ldat=128/pp}', Event Count (a--r0x): 67900852

Overhead	Samples	Local Weight	Memory Access	Symbol	Shared Object	Data Symbol	Data Object	Snoop	TLB Access	Locked	Blocked	Local INSTR Latency
0.22% 0.07%	1	1 38060	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	47251
0.18% 0.06%	1	1 31338	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	40411
0.17% 0.06%	1	1 29572	L3 or L3 hit	[.]asm_mutex	lockcontention	[.]0x0000556db14282a0	[heap]	HitM	L1 or L2 hit	Yes	N/A	36652

3.11.3 Solutions for Performance Bottlenecks

The following is a list of suggested solutions:

1. Run multiple instances of the workload with a scale-out approach instead of a single instance with scale-up so that the contention for per instance hot variables (including locks) is reduced.
2. Guard the cmpxchg by checking that the destination memory is expected with a load, test, and branch beforehand.

3. Implement a backoff mechanism so that the cmpxchg is issued less. For example, in locks, exponential backoff is a common and effective method to prevent all cores from being in lockstep. In the case of contention for a lock, checking to see if it is accessible by a load before trying to write to it through a cmpxchg will help.

The code in Example 3-54 provides an example:

Example 3-54. Locking Algorithm for the Sapphire Rapids Microarchitecture

```
 lock_loop:
 while (lock is not free) // just a load operation
 execute pause;

// now the lock is free, so try to acquire it.
 Exponential Backoff spin // so all the cores don't come back at the same time
 Execute cmpxchg on the lock
 if the lock is not successfully acquired, goto lock_loop
```

Additionally, as the core counts continue to increase, exploring other algorithmic fixes that dissolve or reduce contention on memory variables (including locks) is essential. For example, instead of frequently updating a hot statistical variable from all threads, consider updating a copy of it per thread (without contention) and later aggregate the updated per-thread copies on a less frequent basis or use some existing atomic-free concurrency methods such as rseq[1]. As another example, restructure locking algorithms to use hierarchical locking when excessive contention is detected on a global lock.

3.11.4 Case Study: SysBench/MariaDB

With SysBench/MariaDB 10.3.34[2], the workload's throughput drops as the number of threads increases. Another metric we can use is the CHA% Cycles Fast Asserted. It is a signal to slow down the cores when the Ingress Queue fills up. This is another way to identify scalability issues. The graph below plots the number of active client threads representing the work intensity on the horizontal axis. The percentage of Fast Asserts is plotted on the vertical axis.

The baseline case (blue line) had a sharp throughput with increased thread count, as all cores reduced their throughput as they suffered from the increasing percent of Fast Asserts. With the same work distributed instances (red line), Fast asserts dropped. Similarly, with a software fix (gray line), again, the Fast Asserts dropped even though only one instance was in execution.

1. https://git.kernel.org/pub/scm/libs/librseq/librseq.git/tree/doc/man/rseq.2
2. The most current version is MariaDB 10.3.39

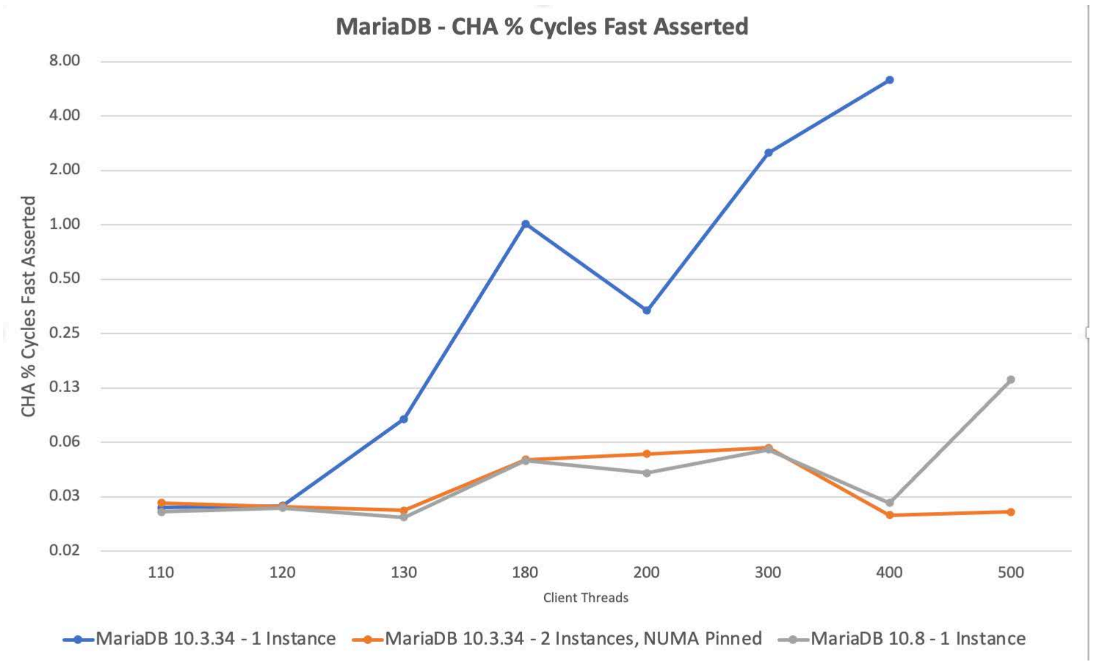


Figure 3-4. MariaDB - CHA % Cycles Fast Asserted

3.11.5 Scalability With False Sharing

A two-socket 4th Generation Intel® Xeon® Scalable Processors 8480 system can support up to 224 hardware threads (2 sockets x 56 cores per socket x 2 threads per core). However, when multiple threads concurrently access different variables in a structure that happen to reside in the same cache line, it can result in false sharing leading to scalability issues. False sharing can cause unnecessary cache invalidations and updates leading to significant performance degradation in multi-threaded programs that utilize all the hardware threads. Therefore, it is essential to avoid false sharing by designing data structures and memory layouts that minimize contention on shared cache lines to achieve optimal performance in multi-threaded environments.

3.11.5.1 Causes of False Sharing

False sharing is a performance problem that can occur in multi-threaded programming when threads access different variables sharing the same cache line. Cache lines are units of memory that are loaded into the processor's cache. When multiple threads write different variables in the same cache line, they end up competing for access to the cache line. This results in cache invalidations and updates that are unnecessary, which can lead to a significant performance degradation. This problem gets worse when many threads are contending for the same cache line.

3.11.5.2 Detecting False Sharing

The perf c2c is a profiling tool available in Linux that detects false sharing issues by analyzing cache-to-cache (c2c) transfers between threads. It works by intercepting the cache coherence messages sent between threads and identifying the specific cache lines that are involved in false sharing. The perf c2c approach generates a report that shows the amount of time spent on c2c transfers, the number of bytes transferred, and the specific cache lines that are affected by false sharing. This approach provides a more precise and accurate method of detecting false sharing issues compared to traditional profiling tools, as it directly measures the cache coherence overhead caused by false sharing. The perf c2c approach is particularly useful for detecting subtle false sharing issues that may not be visible using other profiling tools.

Hardware Invalidation Tracking Modified (HITM) is a counter in the perf c2c output that represents the number of cache lines that were modified in one cache and then invalidated in another cache due to both false and true sharing. The HITM counter provides insight into the performance impact of false sharing by measuring the number of unnecessary cache invalidations and the resulting traffic between caches. By reducing false sharing, the HITM counter can be reduced, leading to better performance and scalability in multi-threaded programs.

Steps for perf c2c analysis:

1. Collect perf c2c data on the target system (this example is for the full system):

   ```
   "perf c2c record  -a -u --ldlat 50 -- sleep 30
   ```

2. Generate report (this can take considerable time to process)

   ```
   "perf c2c report -NN -g --call-graph --full-symbols -c pid,iaddr --stdio >perf_report.txt
   ```

3. Check the generated perf_report.txt for "Shared Data Cache Line Table" (see Table 3-9). This table is sorted by the HITM. Pay attention to the topped "CacheLine address". See Example 3-55.
4. Read the perf_report.txt for the "Shared Cache Line Distribution Pareto" (see Table 3-10). Check the "Offset" column to see if there are multiple offset within single cache line. If there are multiple offset, that points to a potential false s haring issue. See Example 3-56.

The blog, https://joemario.github.io/blog/2016/09/01/c2c-blog/ , provides a nice introduction to perf c2c in Linux.

3.11.5.3 Fixing False Sharing and Additional Resources

The following is a list of suggested solutions:

1. Add padding so the fields are not on the same cache line. Example 3-56 shows to prevent the false sharing between the **full** and **empty lfstack** variables padding is added between them. This is the fix detailed in Section 3.11.5.4. This has the additional effect of increasing the memory sizes and may create other false sharing for other variables.
2. Run multiple instances of the workload instead of a single instance so that the false sharing for per in-stance false sharing variables is reduced as fewer hardware threads are allocated per instances.
3. Change other parameters to prevent the false sharing. In the case of Go, the GOGC variable can be tuned to reduce this.
4. In some environments, it may not be desirable to increase data structure sizes. In this case there may be other patterns to follow such as splitting up a data structure or changing writes for some global variable to use compare(read)-then-write instead of unconditional write. However, this will require further code refactoring.

The Linux kernel has documented some kernel specific False Sharing issues and how to mitigate them.

A blog by a Netflix engineer details how they used a variety of tools including the Intel PMCs (Performance Monitoring Counters) to find and fix False Sharing in JVM.

Example 3-55. Perf Annotation for runtime.getempty

```
next :* atomic.Load64(&node.next)
6290    425fb6: mov      (%rcx) ,%rdx              // lfstack.go.48
        425fb9 lea       0x87fb88(%rip) ,%rbx
9703    425fc0 lock      cmpxchng %rdx,(%rbx)      // lfstack.go.49
        425fc5 sete      %dl
        425fc8 test      %dl,%dl
        425fca je        425f9e <runtime.getempty+0x19e>
        425fcc jmp       425fde <runtime.getempty+0x1d0>
        425fce xor       %ecx,%ecx
```

Example 3-56. Padding Insertion in Go Runtime

```
src/runtime/mgc.go
@@ -285,8 + @@ func pollFractionalWorkerExit() bool {

 var work struct {
    full l-stack                  // lock-free list of full blocks workbuf.
  + pad0 cpu.CacheLinePad         // prevents false-sharing between full and empty.
    empty l-stack                 // lock-free list of empty blocks workbuf.
```

3.11.5.4 Case Study: DeathStarBench/hotelReservation

DeathStarBench is an open-source benchmark suite for microservice workloads, originally developed by Cornell University. It represents different applications written in modern cloud native architecture. The hotelReservation workload in DeathStarBench mimics a typical microservice workload: a hotel booking system. It is written in Golang and uses gRPC-go for inter-microservice communication.

When running with the default parameters and on a single instance of the workload, perf c2c shows false sharing issue with the DSB HR workload. Table 3-10 shows that there are two different offsets being modified by different threads/functions for the specific cache line.

Table 3-9. Shared Data Cache Line Table

Cache Line				Total Hitm	Load Hitm	
Index	Address	Node	PA cnt		Total	LclHitm
0	**0xca5b40**	**1**	**19364**	**3.25%**	**9083**	**9083**
1	**0xd9a840**	0	10918	1.66%	4652	4652
2	**0xce1140**	1	10613	1.56%	4352	4352
3	**0xd9a080**	0	8300	1.14%	3181	3181
4	**0xd9a8c0**	0	4274	0.87%	2448	2448
5	**0xd95900**	0	5346	0.83%	2334	2334
6	**0xd9d800**	1	5440	0.83%	2324	2324
7	**0xce0980**	1	6129	0.83%	2319	2319
8	**0xd98800**	1	5117	0.77%	2160	2160

Table 3-10. Shared Cache Line Distribution Pareto

HTTM		Data Address						
RmtHitm	LclHitm	Offset	Node	Total Records	cpu cnt	Symbol	Shared Object	Souce:Line
0.00%	15.26%	0x0	1	8272	112	[.] runtime.gcDrainN	frontend	mgmark.go:1186
0.00%	8.99%	0x0	1	7276	112	[.] runtime.gcDrain	frontend	mgmark.go:1028
0.00%	7.99%	0x0	1	2850	112	[.] runtime.trygetfull	frontend	lfstack.go.49
0.00%	3.36%	0x0	1	2827	112	[.] runtime.trygetfull	frontend	mgcwork.go.421
0.00%	3.01%	0x0	1	1324	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.35
0.00%	1.94%	0x0	1	885	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.33
0.00%	37.84%	0x8	1	9239	112	[.] runtime.getempty	frontend	lfstack.go.49
0.00%	6.90%	0x8	1	2875	112	[.] runtime.(*lfstack).push	frontend	fstack.go.35
0.00%	5.92%	0x8	1	7188	112	[.] runtime.getempty	frontend	lfstack.go.43
0.00%	4.81%	0x8	1	1947	112	[.] runtime.(*lfstack).push	frontend	lfstack.go.33
0.00%	2.87%	0x8	1	1012	112	[.] runtime.getempty	frontend	mgcwork.go.350

To find the root causes, perf annotate target function:

```
perf annotate --tui -l -n "runtime.(*lfstack).push
```

and review the source code to identify false sharing. In this case the update of full and empty **lfstack** variables by hardware threads on different cores causes the false sharing.

After identifying and fixing the false sharing problem in the Golang runtime (it is in the Go Runtime), releasing in and recompiling the workload binary with the modified Golang runtime improved the throughput metric by 12%. As the following table shows, other metrics such as the CPI and the CHA Fast Asserts also improve significantly. The perf c2c report also shows no additional false sharing.

Table 3-11. False Sharing Improvements

Metric	False Sharing Fix/Base
TPS	1.12
CPI	0.84
Metric CHA % cycles Fast Asserted	0.42

3.11.6 Instruction Sequence Slowdowns

The Golden Cove CPU microarchitecture upon which the Sapphire Rapids microarchitecture is based has increased the cost of mixing Legacy SSE and VEX without clearing the state of upper registers for power efficiency reasons.

3.11.6.1 Causes of Instruction Sequence Slowdowns

The Golden Cove CPU microarchitecture eliminated some hardware speed paths for power efficiency and replaced them with microcode. The instruction sequence in Table 3-12 mixes VEX and Legacy SSE. It has, for example, higher core cycles than on the previous generation Sunny Cove CPU microarchitecture for the Ice Lake version of the 3rd Generation of Intel® Xeon® Scalable processors. The higher core cycles are due to the execution of additional micro-operations.

Table 3-12. Instruction Sequence Mixing VEX on the Sapphire Rapids and Ice Lake Server Microarchitectures

Intel Assembly Code Syntax	Ice lake Server Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)	
	Inst Retired	Core Cycles	Inst Retired	Core Cycles
VPXOR XMM3, XMM3, XMM3; VEXTRACTI128 XMM3, YMM3, 1; PXOR XMM3, XMM3	3.00	1	3.00	388.04

3.11.6.2 Detecting Instruction Sequence Slowdowns

The event ASSISTS.SSE_AVX_MIX can be used to determine if there are VEX to legacy SSE transitions. The following Linux perf command-line can be used while the workload is running:

```
$ sudo perf stat -e 'assists.sse_avx_mix'1 <workload>
```

With the Intel® TMA (Topdown Methodology) (there is a metric called Mixing_Vectors which gives the percentage of injected blend uops out of all the uops issued. Usually, a Mixing_Vectors metric over 5% is worth investigating. You can find more details in Appendix B1 of the Optimizations Guide.

3.11.6.3 Fixing Instruction Sequence Slowdowns

The following is a list of suggested solutions:

1. When possible, use VEX-encoded instructions for all the SIMD instructions when possible.
2. Insert a VZEROUPPER to tell the hardware that the state of the higher registers is clean between the VEX and the legacy SSE instructions. Often the best way to do this is to insert a VZEROUPPER before returning from any function that uses VEX (that does not produce a VEX register) and before any call to an unknown function.

VZEROUPPER was inserted in the code sequence below and there are no SSE_AVX_MIX assists. With this change, the Core Cycles do not have a performance inversion relative to the previous generation.

Table 3-13. Fixed Instruction Sequence with Improved Performance on Sapphire Rapids Microarchitecture

Intel Assembly Code Syntax	Ice lake Microarchitecture (Sunny Cove Cores)		Sapphire Rapids Microarchitecture (Golden Cove Cores)		ASSISTS.SSE_AVX_MIX
	Inst Retired	Core Cycles	Inst Retired	Core Cycles	
VPXOR XMM3, XMM3, XMM3; VEXTRACTI128 XMM3, YMM3, 1; PXOR XMM3, XMM3	4.00	2.00	4.00	1.00	0

3.11.7 Misprediction for Branches >2GB

The Golden Cove CPU is a wider machine and might exhibit a higher Top-down Microarchitecture Analysis (TMA) Bad Speculation percentage. See Section B.1.1 for additional information about TMA. Some sources of Bad Speculation are branch prediction misses. In this case, however, Bad Speculation is due to the wider machine and less efficient branch prediction for certain indirect branches.

1. Using upstream perf. If OS doesn't have support for the event use cpu/event=0xc1,umask=0x10,name=assists_sse_avx_mix/

3.11.7.1 Causes of Branch Misprediction >2GB

For a near absolute indirect JMP/CALL branch instruction (opcodes FF /4 and FF /2), the branch distance (ADDR_TARGET - ADDR_BRANCH) affects the performance of the branch predictor. The branch predictor uses fewer resources to predict the branch if its distance can be specified with a 32-bit signed displacement (JMP/CALL imm32). If the distance is larger (>2GB), the predictor uses more resources to predict the branch and performance may suffer.

3.11.7.2 Detecting Branch Mispredictions >2GB

You can use the Last Branch Record (LBR) to identify jumps greater than 2GB. The collection of performance analysis tools based on perf on Linux supports this. The following is an example output from the tool. It shows that 21% of the call/jumps of >2GB offset are mispredicted. The histogram of one of the indirect branches at address 0x555555603664 shows that it is to one target and in a library. The profile mask is to use LBR, and the duration is 10 seconds. It does a system-level profile.

```
% ./do.py profile --profile-mask=0x100 -s 10

count of indirect call/jump of >2GB offset: 93200
count of mispredicted indirect call/jump of >2GB offset: 19943
misprediction ratio for indirect branch at address 0x7ffff577eca4: 4.23%
misprediction ratio for indirect branch at address 0x5555556030c4: 32.23%
misprediction ratio for indirect branch at address 0x555555603664: 22.30%
misprediction ratio for indirect branch at address 0x555555603c24: 13.84%
...
indirect_0x555555603664 histogram:
0x7ffff7af2670:  50501 100.0%
```

Figure 3-5. Identifying >2GB Branches

3.11.7.3 Fixing Branch Mispredictions >2GB

Arrange the code so the jumps don't span the >2GB range. This can be done through a variety of approaches:

1. If possible, statically link all the libraries into the executable.
2. For **.text** to library code, use the Glibc environment variable LD_PREFER_MAP_32BIT_EXEC=1 to restrict the addresses into the 4GB range.
3. For dynamically compiled code, keep it close to the .text address or copy the frequently called entries into the dynamically compiled code address region. See the Google V8 Blog.

In a case study with WordPress/PHP running eight containers with and without the 2GB fix, the CPI and performance scores improve by 6%.

Table 3-14. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction

		WP4.2 / PHP7.4.29 - NO FIX	WP4.2 / PHP7.4.29 - 2G FIX in Glibc	2G FIX/ NO FIX
Config	**Workers**	8c x 42	8c x 42	-
	Cores Per socket	56	56	1.00
	Sockets	2	2	1.00
	Total Cores	112	112	1.00
	Total Thread Count	224	224	1.00

Table 3-14. WordPress/PHP Case Study: With and Without a 2GB Fix for Branch Misprediction

		WP4.2 / PHP7.4.29 - NO FIX	WP4.2 / PHP7.4.29 - 2G FIX in Glibc	2G FIX/ NO FIX
Performance	Throughput	1.00	1.06	1.06
	CPI	1.12	1.05	0.96
Path Length	Instructions per Unit of Work	33,789,862.68	33,730,155.10	1.00
Cycles per Transaction	Cycles per Unit of Work	37,803,310.48	35,359,628.33	0.94

3.12 OPTIMIZING COMMUNICATION WITH PCI DEVICES ON INTEL® 4TH GENERATION INTEL® XEON® SCALABLE PROCESSORS

The Sapphire Rapids microarchitecture introduced a new set of instructions designed to optimize communication between SW running on IA cores and PCI devices on the platform.

3.12.1 Signaling Devices with Direct Move

Most software-to-device interaction follows a producer-to-consumer relationship where the software creates work for the device and then signals it to inform the device that work is available. Descriptor rings are the ubiquitous pattern here and once descriptors are added to the ring, the signal (or "doorbell") consists of an update to the tail pointer register on the device. This is a write to an MMIO-mapped BAR register.

Such writes tend to be relatively expensive operations –the latency to complete the write to the device is high relative to the CPU operating speed. Since writes are ordered by default, this creates a bubble during which subsequent writes cannot be drained from store buffers. Signaling can therefore affect performance via store backpressure.

As a result, some software libraries avoid frequent signaling by batching relatively large quantities of work descriptors with each doorbell update. However, this is not always possible, and it introduces latency.

The Sapphire Rapids microarchitecture introduces "Direct Store" instructions to optimize signaling; there are two instructions in the family:

- MOVDIRI: 4/8B direct store.
- MOVDIR64B: 64B atomic direct copy.

Direct Stores are weakly ordered (like non-temporal or USWC-mapped memory writes) regardless of the underlying memory type (which is usually UC for MMIO-mapped locations). Since they do not order subsequent writes the performance issue described above does not occur.

Since they are intended for signaling, direct stores will never combine with other stores to the same address as can happen with non-temporal or USWC writes. Each write is guaranteed to occur as issued. In the case of MOVDIR64B, the full 64B will be delivered as a single write to the device. This is the only ISA that carries an architectural guarantee of >8B atomicity.

These instructions benefit from the fact that signaling use cases typically do not care if subsequent writes are observed before the doorbell itself because the ordering is relaxed. However, since typically the door-bell must not be observable before earlier writes (such writes are creating the work descriptors), SW should insert a store fence immediately before the direct store.

Having a fence before the direct store does not normally limit performance– except when many direct stores are issued. If there is an SFENCE before each, the fence on direct store N+1 imposes an order on direct store N, which can remove some of the benefits. The guideline is to avoid this where possible. One technique that may work if multiple doorbells to different addresses are being issued (such as for a NIC driver that is handling multiple descriptor rings), is to group the direct stores to different locations together and insert a single SFENCE before the group.

It is also worth noting that the device write latency can vary widely with the address being written. This is especially true on large CPUs implemented as multiple tiles. So if SW has the luxury of choosing between multiple addresses, it is possible to envisage adaptive schemes that "match" an address to a SW thread (especially if that thread is pinned to a single core) by selecting the best performing such address during an initialization stage.

3.12.1.1 MOVDIR64B: Additional Considerations

As noted above MOVDIR64B is a copy operation; it moves data from one 64B-aligned address to another. Typical usage is that the source address is a memory location, and the destination is MMIO mapped to a device, whereupon it confers the benefits described above. However, since the source data is usually written immediately before the MOVDIR64B, additional considerations include:

- It is unnecessary to fence to ensure the source data is written before the MOVDIR64B since the source data is written to the same address that the MOVDIR64B reads. In some scenarios, no store fence is needed in conjunction with MOVDIR64B. The correct operation of the system depends on being observed before the MOVDIR64B if no other data is written to memory.
- It is critical to allow store forwarding of the source data for the best performance.
- The source data should be aligned to 64B and written at the same granularity that the MOVDIR64B reads. For the Sapphire Rapids microarchitecture, this is 64B: the source data should, therefore, be written using 64B Intel® AVX-512 Instructions for the best performance.

3.12.1.2 Streaming Data

MOVDIR64B can also be used to stream data to a device by copying a block of memory because it is weakly ordered. This is similar behavior to mapping the destination memory locations as USWC, except:

- The destination address can remain mapped UC.
- The writes are guaranteed to arrive at the device as 64B writes, which is not guaranteed with any other method.

3.13 SYNCHRONIZATION

3.13.1 User-Level Monitor, User-Level MWAIT, and TPAUSE

New instructions for user-level monitor and MWAIT act like legacy monitor and MWAIT instructions with additional functionality identified as the timeout and ring-3 (user space) application support. TPAUSE is similar to legacy pause instruction but is designed to accept time interval and sleep state parameters. User-level MWAIT and TPAUSE support the same C0.1 light sleep and C0.2 deeper sleep states. These instructions are helpful in user space applications that support a busy poll, synchronization, or asynchronous IO, such as waiting for an event. A minor code modification yields power benefits along with low latency wake-up.

3.13.1.1 Checking for User-Level Monitor, MWAIT, and TPAUSE Support

This section describes how to check whether a processor supports user-level monitor, user-level MWAIT, or TPAUSE; if user-level monitor, user-level MWAIT, or TPAUSE instruction is supported, then CPUID.(EAX=07H, ECX=0): ECX [bit 5] is enumerated as 1.

Example 3-57. Identification of WAITPKG with CPUID

```
…identify the existence of cpuid instruction
… ;
… ;
Identify signature is genuine Intel …;
mov eax, 7; Request for feature flags
mov ecx, 0; Request for feature flags
cpuid; 0FH, A2H CPUID instruction
test ecx, 00000020h;
Is waitpkg bit (bit 5) in feature flags equal to 1 jnz Found
```

3.13.1.2 User-Level Monitor, User-Level MWAIT, and TPAUSE Operations

User-level monitor initializes the monitor hardware in such a way that, after execution of the user-level MWAIT, a store to a monitored address acts as a wakeup event. So, the User level monitor and the user-level MWAIT work together to obtain a sleep state. TPAUSE is a single instruction request to enter one of the same two sleep states for a defined time

There are possibilities of a "false wake-up" because of other events, notably interrupts or timeouts. The application may re-execute user-level MWAIT/TPAUSE if it has been falsely woken. If the application needs to determine the source of the predefined OS sleep wakeup, RFLAGS.CF is set Otherwise it is assumed that the application can detect changes at the monitored address (MWAIT) or poll for activity (TPAUSE).

3.13.1.3 Recommended Usage of Monitor, MWAIT, and TPAUSE Operations

A frequent paradigm in packet processing applications is to have dedicated HW threads polling a NIC receive descriptor ring for ingress traffic. This kind of "busy polling" arrangement wastes energy when the traffic rates are low. Changing the polling loop to perform user-level Monitor/MWAIT on the next descriptor to be written can save substantial power in periods of low traffic. The same scheme could be used with any "work distributor," assigning work by writing to selected memory locations.

Accelerators frequently offload tasks from SW in an asynchronous manner. For example, the Intel® Data Streaming Accelerator (Intel® DSA) performs copy operations and can return the status of the completed operation by writing to memory. If an application uses the user-level monitor/MWAIT at a memory location where the status field will be written, it can be woken when the task is complete. Instead of monitoring, the device may issue an interrupt that can act as a wake-up event.

Alternatively, applications may decide to choose TPAUSE as a wait event. This has the advantage of being independent of the number of event sources.

In all cases, a small change in the user space application is needed to convert a busy poll application to something more energy efficient with low latency wake-up.

Synchronous application: when two hardware threads from the same core use user-level monitor and user-level MWAIT, it can progress effectively as some of the hardware resources are available to the other thread when a hyperthread issues the user-level MWAITs.

To achieve the best performance using user-level monitor and user-level MWAIT:

- The entire contents of monitored locations must be verified after user-level MWAIT to avoid a false wake-up.
- It is the developer's responsibility to check the contents of monitored locations:
  - Before issuing monitor.
  - Before issuing user-level MWAIT.
  - After user-level MWAIT. See Example 3-58.
- If an application expects a store to a monitored location, the timeout value should be as high as it is supported.

Since user-level MWAIT and TPAUSE are a hint to a processor, a user should selectively identify locations in the application.

Example 3-58. Code Snippet in an Asynchronous Example

```
void * m_address;  // it is expected device will update m_address to 1
unsigned char ret;
while (1) {
        if (*m_address != 0) // if device already finished operation, no need to user monitor/user mwait
                break;
        if (*m_address == 0) { // check monitored location before issuing umonitor instruction
                _umonitor (m_address);
                if (*m_address == 0) {         // check monitored location before issuing umwait instruction
                        ret = _umwait(0, 0x186A0);    // some high value in timeout
                }
        }
}
```

5. Updates to Chapter 5

Change bars and **violet** text show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual:* Coding for SIMD Architectures.

Changes to this chapter:

- Section 5.3.1
  - Typo correction in Figure 5-4 (Instrinsics to Intrinsics)

CHAPTER 5 CODING FOR SIMD ARCHITECTURES

- Processors based on Intel Core microarchitecture support MMX™, Intel® SSE, Intel® SSE2, Intel® SSE3, and Intel® SSSE3.
- Processors based on Enhanced Intel Core microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.
- Processors based on Nehalem microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, Intel SSE4.1, and Intel SSE4.2.
- Processors based Westmere microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, Intel SSE4.1, Intel SSE4.2, and AESNI.
- Processors based on Sandy Bridge microarchitecture support MMX, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, Intel SSE4.1, Intel SSE4.2, AESNI, PCLMULQDQ, and Intel® AVX.
- Intel® Pentium® 4, Intel® Xeon® and Intel® Pentium® M processors include support for Intel SSE2, Intel SSE, and MMX technology. Intel SSE3 was introduced with the Intel Pentium 4 processor supporting Intel® Hyper-Threading Technology at 90 nm technology.
- Intel® Core™ Solo and Intel® Core™ Duo processors support MMX, Intel SSE, Intel SSE2, and Intel SSE3.

Single-instruction, multiple-data (SIMD) technologies enable the development of advanced multimedia, signal processing, and modeling applications.

SIMD techniques can be applied to text/string processing, lexing and parser applications. This is covered in Chapter 14, "Intel® SSE4.2 and SIMD Programming For Text-Processing/Lexing/Parsing." Techniques for optimizing AESNI are discussed in Section 6.10.

To take advantage of the performance opportunities presented by these capabilities, do the following:

- Ensure that the processor supports MMX technology, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.
- Ensure that the operating system supports MMX technology and Intel SSE (OS support for Intel SSE2, Intel SSE3 and Intel SSSE3 is the same as OS support for Intel SSE).
- Employ the optimization and scheduling strategies described in this book.
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Utilize the cacheability instructions offered by Intel SSE and Intel SSE2, where appropriate.

5.1 CHECKING FOR PROCESSOR SUPPORT OF SIMD TECHNOLOGIES

This section shows how to check whether a processor supports MMX technology, Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.1.

SIMD technology can be included in your application in three ways:

1. Check for the SIMD technology during installation. If the desired SIMD technology is available, the appropriate DLLs can be installed.
2. Check for the SIMD technology during program execution and install the proper DLLs at runtime. This is effective for programs that may be executed on different machines.
3. Create a "fat" binary that includes multiple versions of routines; versions that use SIMD technology and versions that do not. Check for SIMD technology during program execution and run the appropriate versions of the routines. This is especially effective for programs that may be executed on different machines.

5.1.1 Checking for MMX Technology Support

If MMX technology is available, then CPUID.01H:EDX[BIT 23] = 1. Use the code segment in Example 5-1 to test for MMX technology.

Example 5-1. Identification of MMX Technology with CPUID

```
…identify existence of cpuid instruction
…                            ;
…                            ; Identify signature is genuine Intel
…                            ;
mov eax, 1                   ; Request for feature flags
cpuid                        ; 0FH, 0A2H CPUID instruction
test edx, 00800000h          ; Is MMX technology bit (bit 23) in feature flags equal to 1
jnz      Found
```

See CPUID Information for Intel® Processors for more information.

5.1.2 Checking for Intel® Streaming SIMD Extensions (Intel® SSE) Support

Checking for processor support of Intel Streaming SIMD Extensions (SIntel SE) on your processor is similar to checking for MMX technology. However, operating system (OS) must provide support for Intel SSE states save and restore on context switches to ensure consistent application behavior when using Intel SSE instructions.

To check whether your system supports Intel SSE, follow these steps:

1. Check that your processor supports the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSE existence.

Example 5-2 shows how to find the SSE feature bit (bit 25) in CPUID feature flags.

Example 5-2. Identification of Intel® SSE with CPUID

```
…Identify existence of cpuid instruction
                             ; Identify signature is genuine intel
mov eax, 1                   ; Request for feature flags
cpuid                        ; 0FH, 0A2H cpuid instruction
test EDX, 002000000h         ; Bit 25 in feature flags equal to 1
jnz          Found
```

5.1.3 Checking for Intel® Streaming SIMD Extensions 2 (Intel® SSE2) Support

Checking for support of Intel SSE2 is like checking for Intel SSE support. The OS requirements for Intel SSE2 Support are the same as the OS requirements for Intel SSE.

To check whether your system supports Intel SSE2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSE2 technology existence.

Example 5-3 shows how to find the SSE2 feature bit (bit 26) in the CPUID feature flags.

Example 5-3. Identification of Intel® SSE2 with cpuid

```
     …identify existence of cpuid instruction
     …                                  ; Identify signature is genuine intel
mov eax, 1                              ; Request for feature flags
cpuid                                   ; 0FH, 0A2H CPUID instruction
test EDX, 004000000h                    ; Bit 26 in feature flags equal to 1
jnz       Found
```

5.1.4 Checking for Intel® Streaming SIMD Extensions 3 (Intel® SSE3) Support

Intel SSE3 includes 13 instructions, 11 of those are suited for SIMD or x87 style programming. Checking for support of Intel SSE3 instructions is similar to checking for Intel SSE support. The OS requirements for Intel SSE3 Support are the same as the requirements for Intel SSE.

To check whether your system supports the x87 and SIMD instructions of Intel SSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the ECX feature bit 0 of CPUID for Intel SSE3 technology existence.

Example 5-4 shows how to find the SSE3 feature bit (bit 0 of ECX) in the CPUID feature flags.

Example 5-4. Identification of Intel® SSE3 with CPUID

```
     …identify existence of cpuid instruction
     …                            ; Identify signature is genuine intel
mov eax, 1                        ; Request for feature flags
cpuid                             ; 0FH, 0A2H CPUID instruction
test ECX, 000000001h              ; Bit 0 in feature flags equal to 1
jnz       Found
```

Software must check for support of MONITOR and MWAIT before attempting to use MONITOR and MWAIT.Detecting the availability of MONITOR and MWAIT can be done using a code sequence similar to Example 5-4. The availability of MONITOR and MWAIT is indicated by bit 3 of the returned value in ECX.

5.1.5 Checking for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE) Support

Checking for support of Intel SSSE3 is similar to checking for Intel SSE support. The OS requirements for Intel SSSE3 support are the same as the requirements for Intel SSE.

To check whether your system supports Intel SSSE3, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bits of CPUID for Intel SSSE3 technology existence.

Example 5-5 shows how to find the Intel SSSE3 feature bit in the CPUID feature flags.

Example 5-5. Identification of SSSE3 with cpuid

```
     …Identify existence of CPUID instruction
     …                            ; Identify signature is genuine intel
mov eax, 1                        ; Request for feature flags
cpuid                             ; 0FH, 0A2H CPUID instruction
test ECX, 000000200h              ; ECX bit 9
jnz       Found
```

5.1.6 Checking for Intel® SSE4.1 Support

Checking for support of SSE4.1 is similar to checking for Intel SSE support. The OS requirements for Intel SSE4.1 support are the same as the requirements for Intel SSE.

To check whether your system supports Intel SSE4.1, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for Intel SSE4.1.

Example 5-6 shows how to find the Intel SSE4.1 feature bit in the CPUID feature flags.

Example 5-6. Identification of Intel® SSE4.1 with CPUID

```
     …Identify existence of CPUID instruction
     …                          ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000080000h            ; ECX bit 19
jnz      Found
```

5.1.7 Checking for Intel® SSE4.2 Support

Checking for support of Intel SSE4.2 is similar to checking for Intel SSE support. The OS requirements for SSE4.2 support are the same as the requirements for Intel SSE.

To check whether your system supports SSE4.2, follow these steps:

1. Check that your processor has the CPUID instruction.
2. Check the feature bit of CPUID for Intel SSE4.2.

Example 5-7 shows how to find the INtel SSE4.2 feature bit in the CPUID feature flags.

Example 5-7. Identification of SSE4.2 with cpuid

```
     …Identify existence of CPUID instruction
     …                          ; Identify signature is genuine intel
mov eax, 1                      ; Request for feature flags
cpuid                           ; 0FH, 0A2H CPUID instruction
test ECX, 000100000h            ; ECX bit 20
jnz      Found
```

5.1.8 DetectiON of PCLMULQDQ and AESNI Instructions

Before an application attempts to use the following AESNI instructions: AESDEC/AESDE-CLAST/AESENC/AESENCLAST/AESIMC/AESKEYGENASSIST, it must check that the processor supports the AESNI extensions. AESNI extensions is supported if CPUID.01H:ECX.AESNI[bit 25] = 1.

Prior to using PCLMULQDQ instruction, application must check if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling SSE state will also support applications that use AESNI extensions and PCLMULQDQ instruction. This is the same requirement for Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.

Example 5-8. Detection of AESNI Instructions

```
     …Identify existence of CPUID instruction
     …                             ; Identify signature is genuine intel
mov eax, 1                         ; Request for feature flags
cpuid                              ; 0FH, 0A2H CPUID instruction
test ECX, 002000000h               ; ECX bit 25
jnz       Found
```

Example 5-9. Detection of PCLMULQDQ Instruction

```
     …Identify existence of CPUID instruction
     …                             ; Identify signature is genuine intel
mov eax, 1                         ; Request for feature flags
cpuid                              ; 0FH, 0A2H CPUID instruction
test ECX, 000000002h               ; ECX bit 1
jnz       Found
```

5.1.9 Detection of Intel® AVX Instructions

Intel AVX operates on the 256-bit YMM register state. Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 5-1.

Prior to using AVX, the application must identify that the operating system supports the XGETBV instruction, the YMM register state, in addition to processor's support for YMM state management using XSAVE/XRSTOR and AVX instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use[1])

2) Issue XGETBV and verify that XFEATURE_ENABLED_MASK[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Detect CPUID.1:ECX.AVX[bit 28] = 1 (AVX instructions supported).

Note: Step 3 can be done in any order relative to 1 and 2.

1.If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XFEATURE_ENALBED_MASK register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

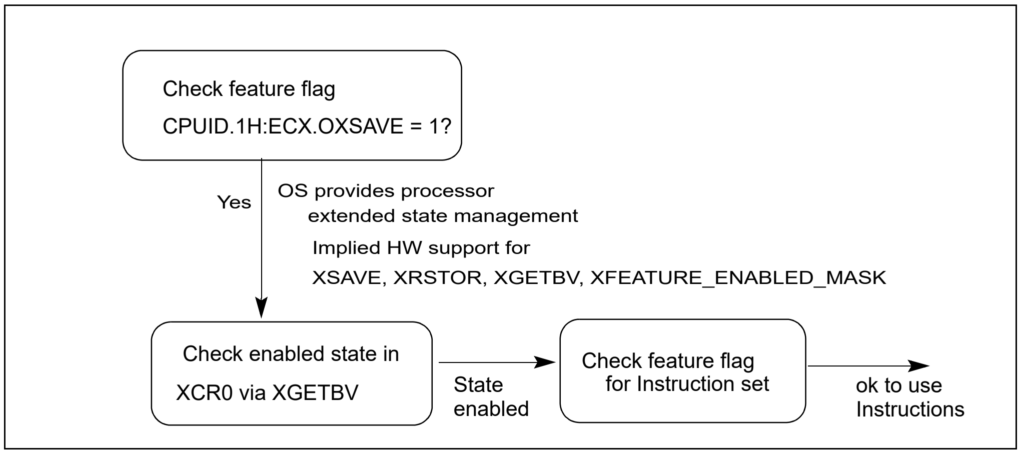


Figure 5-1. General Procedural Flow of Application Detection of Intel® AVX

The following pseudocode illustrates this recommended application Intel AVX detection process:

Example 5-10. Detection of Intel® AVX Instruction

```
INT supports_AVX()
{   mov     eax, 1
    cpuid
    and     ecx, 018000000H
    cmp     ecx, 018000000H; check both OSXSAVE and AVX feature flags
     jne    not_supported
    ; processor supports AVX instructions and XGETBV is enabled by OS
    mov     ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
    XGETBV              ; result in EDX:EAX
    and     eax, 06H
    cmp     eax, 06H; check OS has enabled both XMM and YMM state support
    jne     not_supported
    mov     eax, 1
    jmp     done
NOT_SUPPORTED:
    mov     eax, 0
done:
```

NOTE

It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating systems, AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. "CPUID.1:ECX.XSAVE[bit 26] = 1" does not guarantee the OS actually uses the XSAVE process for state management.

5.1.10 Detection of VEX-Encoded AES and VPCLMULQDQ

VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for Intel AVX.

Example 5-11. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VAESNI()
{    mov       eax, 1
     cpuid
     and       ecx, 01A000000H
     cmp       ecx, 01A000000H; check OSXSAVE AVX and AESNI feature flags
      jne      not_supported
     ; processor supports AVX and VEX-encoded AESNI and XGETBV is enabled by OS
     mov       ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV              ; result in EDX:EAX
     and       eax, 06H
     cmp       eax, 06H; check OS has enabled both XMM and YMM state support
     jne       not_supported
     mov       eax, 1
     jmp       done
NOT_SUPPORTED:
     mov       eax, 0
done:
```

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 and the sequence for detection application support for AVX.

This is shown in the pseudocode:

Example 5-12. Detection of VEX-Encoded AESNI Instructions

```
INT supports_VPCLMULQDQ)
{    mov       eax, 1
     cpuid

     and       ecx, 018000002H
     cmp       ecx, 018000002H; check OSXSAVE AVX and PCLMULQDQ feature flags
      jne      not_supported
     ; processor supports AVX and VEX-encoded PCLMULQDQ and XGETBV is enabled by OS
     mov       ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
     XGETBV              ; result in EDX:EAX
     and       eax, 06H
     cmp       eax, 06H; check OS has enabled both XMM and YMM state support
     jne       not_supported
     mov       eax, 1
     jmp       done
NOT_SUPPORTED:
     mov       eax, 0
done:
```

5.1.11 Detection of F16C Instructions

Application using float 16 instruction must follow a detection sequence similar to Intel AVX to ensure:

- The OS has enabled YMM state management support.
- The processor support Intel AVX as indicated by the CPUID feature flag, i.e. CPUID.01H:ECX.AVX[bit 28] = 1.
- The processor support 16-bit floating-point conversion instructions via a CPUID feature flag (CPUID.01H:ECX.F16C[bit 29] = 1).

Application detection of Float-16 conversion instructions follow the general procedural flow in Figure 5-2.

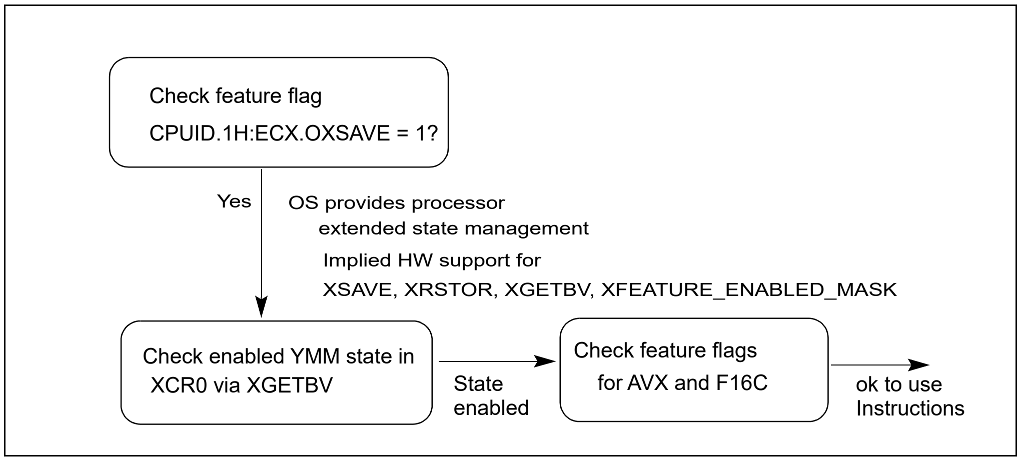

Check feature flag
CPUID.1H:ECX.OXSAVE = 1?
Yes
OS provides processor
extended state management
Implied HW support for
XSAVE, XRSTOR, XGETBV, XFEATURE_ENABLED_MASK
Check enabled YMM state in XCR0 via XGETBV
State enabled
Check feature flags for AVX and F16C
ok to use Instructions

Figure 5-2. General Procedural Flow of Application Detection of Float-16

```
--------------------------------------------------------------------------------------
INT supports_f16c()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 038000000H
        cmp ecx, 038000000H; check OSXSAVE, AVX, F16C feature flags
         jne not_supported
        ; processor supports AVX,F16C instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
-----------------------------------------------------------------------------
```

5.1.12 Detection of FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:

```
----------------------------------------------------------------------------------------
INT supports_fma()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018001000H
        cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
         jne not_supported
        ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
-------------------------------------------------------------------------------
```

5.1.13 Detection of Intel® AVX2

Hardware support for Intel AVX2 is indicated by CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1.

Application Software must identify that hardware supports Intel AVX, after that it must also detect support for AVX2 by checking CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]. The recommended pseudo-code sequence for detection of Intel AVX2 is:

```
----------------------------------------------------------------------------------------
INT supports_avx2()
{       ; result in eax
        mov eax, 1
        cpuid
        and ecx, 018000000H
        cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
         jne not_supported
        ; processor supports AVX instructions and XGETBV is enabled by OS
        mov eax, 7
        mov ecx, 0
        cpuid
        and ebx, 20H
        cmp ebx, 20H; check AVX2 feature flags
        jne not_supported
        mov ecx, 0; specify 0 for XFEATURE_ENABLED_MASK register
        XGETBV; result in EDX:EAX
        and eax, 06H
        cmp eax, 06H; check OS has enabled both XMM and YMM state support
        jne not_supported
        mov eax, 1
        jmp done
        NOT_SUPPORTED:
        mov eax, 0
        done:
}
------------------------------------------------------------------------------
```

5.2 CONSIDERATIONS FOR CODE CONVERSION TO SIMD PROGRAMMING

The VTune Performance Enhancement Environment CD provides tools to aid in the evaluation and tuning. Before implementing them, you need answers to the following questions:

1. Will the current code benefit by using MMX technology, Intel SSE, Intel SSE2, Intel SSE3, or Intel SSSE3?
2. Is this code integer or floating-point?
3. What integer word size or floating-point precision is needed?
4. What coding techniques should I use?
5. What guidelines do I need to follow?
6. How should I arrange and align the datatypes?

Figure 5-3 provides a flowchart for the process of converting code to MMX technology, Intel SSE, Intel SSE2, Intel SSE3, or Intel SSSE3.

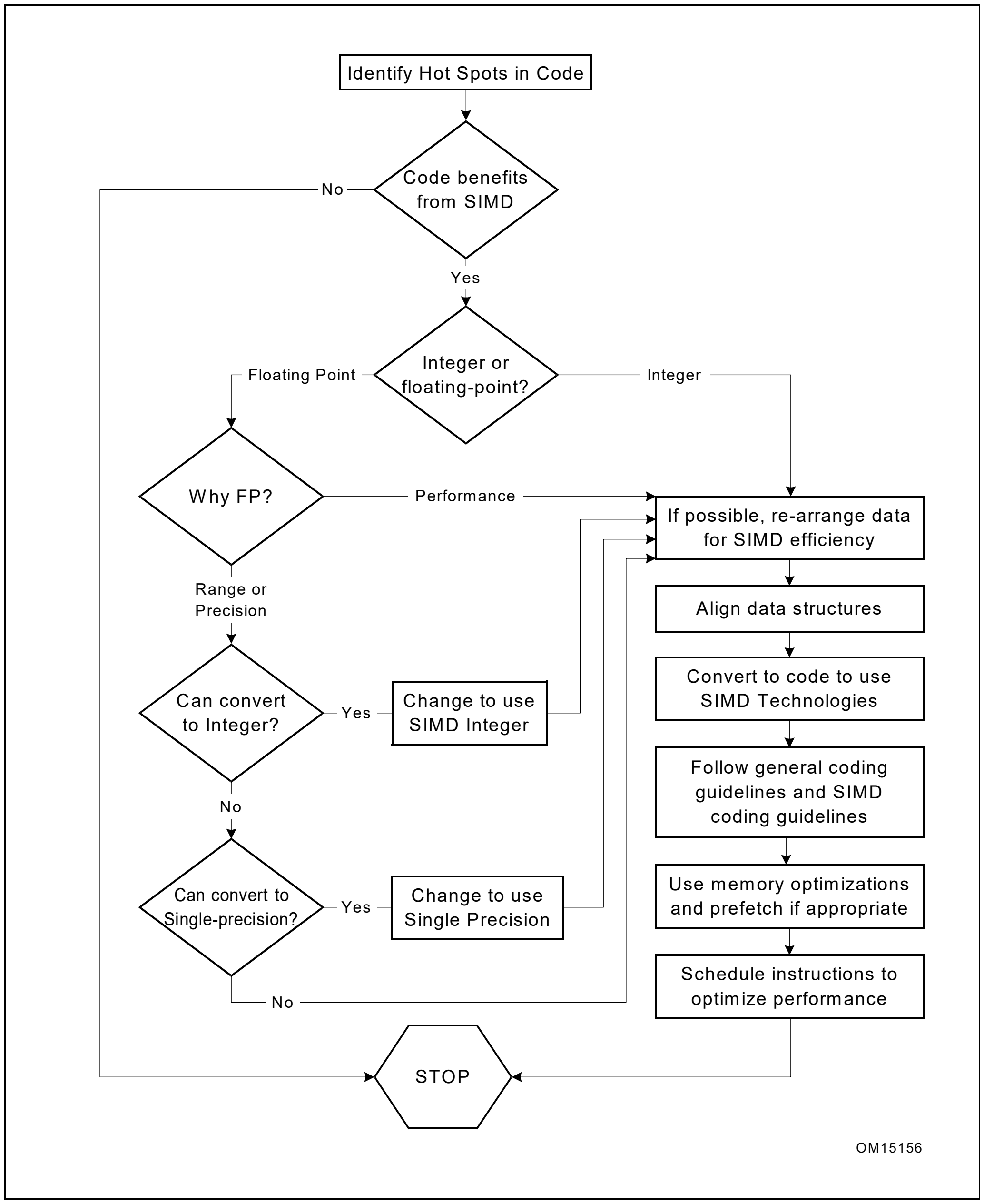

Identify Hot Spots in Code
Code benefits from SIMD
No
Yes
Integer or floating-point?
Floating Point
Integer
Why FP?
Performance
Range or Precision
If possible, re-arrange data for SIMD efficiency
Align data structures
Convert to code to use SIMD Technologies
Follow general coding guidelines and SIMD coding guidelines
Use memory optimizations and prefetch if appropriate
Schedule instructions to optimize performance
Can convert to Integer?
Yes
Change to use SIMD Integer
No
Can convert to Single-precision?
Yes
Change to use Single Precision
No
STOP
OM15156

Figure 5-3. Converting to Intel® Streaming SIMD Extensions Chart

To use any of the SIMD technologies optimally, you must evaluate the following situations in your code:

- Fragments that are computationally intensive.
- Fragments that are executed often enough to have an impact on performance.
- Fragments that with little data-dependent control flow.
- Fragments that require floating-point computations.
- Fragments that can benefit from moving data 16 bytes at a time.
- Fragments of computation that can coded using fewer instructions.
- Fragments that require help in using the cache hierarchy efficiently.

5.2.1 Identifying Hot Spots

To optimize performance, use the VTune Performance Analyzer to find sections of code that occupy most of the computation time. Such sections are called the hotspots. See Appendix A, "Application Performance Tools."

The VTune analyzer provides a hotspots view of a specific module to help you identify sections in your code that take the most CPU time and that have potential performance problems. The hotspots view helps you identify sections in your code that take the most CPU time and that have potential performance problems.

The VTune analyzer enables you to change the view to show hotspots by memory location, functions, classes, or source files. You can double-click on a hotspot and open the source or assembly view for the hotspot and see more detailed information about the performance of each instruction in the hotspot.

The VTune analyzer offers focused analysis and performance data at all levels of your source code and can also provide advice at the assembly language level. The code coach analyzes and identifies opportunities for better performance of C/C++, Fortran and Java* programs, and suggests specific optimizations. Where appropriate, the coach displays pseudo-code to suggest the use of highly optimized intrinsics and functions in the Intel® Performance Library Suite. Because VTune analyzer is designed specifically for Intel architecture (IA)-based processors, including the Pentium 4 processor, it can offer detailed approaches to working with IA. See Appendix A.1.1 for details.

5.2.2 Determine If Code Benefits by Conversion to SIMD Execution

Identifying code that benefits by using SIMD technologies can be time-consuming and difficult. Likely candidates for conversion are applications that are highly computation intensive, such as the following:

- Speech compression algorithms and filters.
- Speech recognition algorithms.
- Video display and capture routines.
- Rendering routines.
- 3D graphics (geometry).
- Image and video processing algorithms.
- Spatial (3D) audio.
- Physical modeling (graphics, CAD).
- Workstation applications.
- Encryption algorithms.
- Complex arithmetics.

Generally, good candidate code is code that contains small-sized repetitive loops that operate on sequential arrays of integers of 8, 16 or 32 bits, single-precision 32-bit floating-point data, double precision 64-bit floating-point data (integer and floating-point data items should be sequential in memory). The repetitiveness of these loops incurs costly application processing time. However, these routines have potential for increased performance when you convert them to use one of the SIMD technologies.

Once you identify your opportunities for using a SIMD technology, you must evaluate what should be done to determine whether the current algorithm or a modified one will ensure the best performance.

5.3 CODING TECHNIQUES

The SIMD features of Intel SSE3, Intel SSE2, Intel SSE, and MMX technology require new methods of coding algorithms. One of them is vectorization. Vectorization is the process of transforming sequentially-executing, or scalar, code into code that can execute in parallel, taking advantage of the SIMD architecture parallelism. This section discusses the coding techniques available for an application to make use of the SIMD architecture.

To vectorize your code and thus take advantage of the SIMD architecture, do the following:

- Determine if the memory accesses have dependencies that would prevent parallel execution.
- "Strip-mine" the inner loop to reduce the iteration count by the length of the SIMD operations (for example, four for single-precision floating-point SIMD, eight for 16-bit integer SIMD on the XMM registers).
- Re-code the loop with the SIMD instructions.

Each of these actions is discussed in detail in the subsequent sections of this chapter. These sections also discuss enabling automatic vectorization using the Intel C++ Compiler.

5.3.1 Coding Methodologies

Software developers need to compare the performance improvement that can be obtained from assembly code versus the cost of those improvements. Programming directly in assembly language for a target platform may produce the required performance gain, however, assembly code is not portable between processor architectures and is expensive to write and maintain.

Performance objectives can be met by taking advantage of the different SIMD technologies using high-level languages as well as assembly. The new C/C++ language extensions designed specifically for Intel SSE3, Intel SSE2, Intel SSE, and MMX technology help make this possible.

Figure 5-4 illustrates the trade-offs involved in the performance of hand-coded assembly versus the ease of programming and portability.

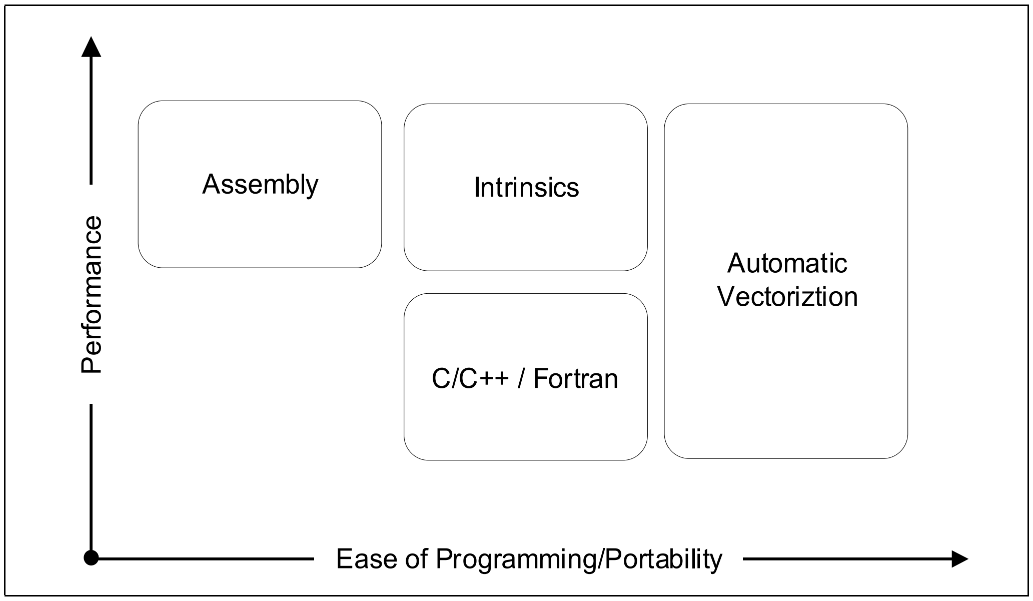


Figure 5-4. Hand-Coded Assembly and High-Level Compiler Performance Trade-Offs

The examples that follow illustrate the use of coding adjustments to enable the algorithm to benefit from the Intel SSE. The same techniques may be used for single-precision floating-point, double-precision floating-point, and integer data under Intel SSE3, Intel SSE2, Intel SSE, and MMX technology.

As a basis for the usage model discussed in this section, consider a simple loop shown in Example 5-13.

Example 5-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
   c[i] = a[i] + b[i];
 }
}
```

Note that the loop runs for only four iterations. This allows a simple replacement of the code with Streaming SIMD Extensions.

For the optimal use of the Intel SSE that need data alignment on the 16-byte boundary, all examples in this chapter assume that the arrays passed to the routine, *A*, *B*, *C*, are aligned to 16-byte boundaries by a calling routine. For the methods to ensure this alignment, please refer to the application notes for the Intel Pentium 4 processor.

The sections that follow provide details on the coding methodologies: inlined assembly, intrinsics, C++ vector classes, and automatic vectorization.

5.3.1.1 Assembly

Key loops can be coded directly in assembly language using an assembler or by using inlined assembly (C-asm) in C/C++ code. The Intel compiler or assembler recognize the new instructions and registers, then directly generate the corresponding code. This model offers the opportunity for attaining greatest performance, but this performance is not portable across the different processor architectures.

Example 5-14 shows the Intel SSE inlined assembly encoding.

Example 5-14. Intel® Streaming SIMD Extensions (Intel® SSE) Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
 __asm {
   mov    eax, a
   mov    edx, b
   mov    ecx, c
   movaps  xmm0, XMMWORD PTR [eax]
   addps   xmm0, XMMWORD PTR [edx]
   movaps  XMMWORD PTR [ecx], xmm0
 }
}
```

5.3.1.2 Intrinsics

Intrinsics provide the access to the ISA functionality using C/C++ style coding instead of assembly language. Intel has defined three sets of intrinsic functions that are implemented in the Intel C++ Compiler to support the MMX technology, Intel SSE, Intel SSE2. Four new C data types, representing 64-bit and 128-bit objects are used as the operands of these intrinsic functions. __M64 is used for MMX integer SIMD, __M128 is used for single-precision floating-point SIMD, __M128I is used for Streaming SIMD Extensions 2 integer SIMD, and __M128D is used for double precision floating-point SIMD. These types enable the programmer to choose the implementation of an algorithm directly, while allowing the compiler to perform register allocation and instruction scheduling where possible. The intrinsics are portable among all Intel architecture-based processors supported by a compiler.

The use of intrinsics allows you to obtain performance close to the levels achievable with assembly. The cost of writing and maintaining programs with intrinsics is considerably less. For a detailed description of the intrinsics and their use, refer to the Intel C++ Compiler documentation.

Example 5-15 shows the loop from Example 5-13 using intrinsics.

Example 5-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

The intrinsics map one-to-one with actual Intel SSE assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an mm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, "+", ">>").
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the Intel Architecture MMX Technology Developer's Guide. For a description of data types, see the Intel® 64 and IA-32 Architectures Software Developer's Manual.

5.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Intel SSE and Intel SSE2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance of applications using this methodology can approach that of one using the intrinsics. Further details on the use of these classes can be found in the Intel C++ Class Libraries for SIMD Operations page.

Example 5-16 shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries.

Example 5-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Here, fvec.h is the class definition file and F32vec4 is the class representing an array of four floats. The "+" and "=" operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundary.

5.3.1.4 Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which loops, such as in Example 5-13 can be automatically vectorized, or converted into Intel SSE code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

- The layout of the loop and the data structures used.
- Dependencies amongst the data accesses in each iteration and across iterations.

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

Example 5-17 shows the code for automatic vectorization for the simple four-iteration loop (from Example 5-13).

Example 5-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
                float *restrict b,
                float *restrict c)
{
        int i;
        for (i = 0; i < 4; i++) {
                c[i] = a[i] + b[i];
        }
}
```

Compile this code using the -QAX and -QRESTRICT switches of the Intel C++ Compiler, version 4.0 or later.

The RESTRICT qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used, provides the only means of accessing the memory in question in the scope in which the pointers live. Without the restrict qualifier, the compiler will still vectorize this loop using runtime data dependence testing, where the generated code dynamically selects between sequential or vector execution of the loop, based on overlap of the parameters. The restrict keyword avoids the associated overhead altogether.

See Intel® C++ Compiler Classic Developer Guide and Reference for details.

5.4 STACK AND DATA ALIGNMENT

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

5.4.1 Alignment and Contiguity of Data Access Patterns

The 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Intel SSE and Intel SSE2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and SSE. Several techniques for improving data access, such as padding, organizing data elements into arrays, etc. are described below. Intel SSE3 provides a special-purpose instruction LDDQU that can avoid cache line splits is discussed in Section 6.7.3

5.4.1.1 Using Padding to Align Data

However, when accessing SIMD data using SIMD operations, access to data can be improved simply by a change in the declaration. For example, consider a declaration of a structure, which represents a point in space plus an attribute.

```
typedef struct {short x,y,z; char a} Point;
Point pt[N];
```

Assume we will be performing a number of computations on X, Y, Z in three of the four elements of a SIMD word; see Section 5.5.1 for an example. Even if the first element in array PT is aligned, the second element will start 7 bytes later and not be aligned (3 shorts at two bytes each plus a single byte = 7 bytes).

By adding the padding variable PAD, the structure is now 8 bytes, and if the first element is aligned to 8 bytes (64 bits), all following elements will also be aligned. The sample declaration follows:

```
typedef struct {short x,y,z; char a; char pad;} Point;
Point pt[N];
```

5.4.1.2 Using Arrays to Make Data Contiguous

In the following code,

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

the second dimension Y needs to be multiplied by a scaling value. Here, the FOR loop accesses each Y dimension in the array PT thus disallowing the access to contiguous data. This can degrade the performance of the application by increasing cache misses, by poor utilization of each cache line that is fetched, and by increasing the chance for accesses which span multiple cache lines.

The following declaration allows you to vectorize the scaling operation and further improve the alignment of the data access patterns:

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty[i] *= scale;
```

With the SIMD technology, choice of data organization becomes more important and should be made carefully based on the operations that will be performed on the data. In some applications, traditional data arrangements may not lead to the maximum performance.

A simple example of this is an FIR filter. An FIR filter is effectively a vector dot product in the length of the number of coefficient taps.

Consider the following code:

```
(data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1]),
```

If in the code above the filter operation of data element I is the vector dot product that begins at data element J, then the filter operation of data element I+1 begins at data element J+1.

Assuming you have a 64-bit aligned data vector and a 64-bit aligned coefficients vector, the filter operation on the first data element will be fully aligned. For the second data element, however, access to the data vector will be misaligned. For an example of how to avoid the misalignment problem in the FIR filter, refer to Intel application notes on Streaming SIMD Extensions and filters.

Duplication and padding of data structures can be used to avoid the problem of data accesses in algorithms which are inherently misaligned. Section 5.5.1 discusses trade-offs for organizing data structures.

NOTE

The duplication and padding technique overcomes the misalignment problem, thus avoiding the expensive penalty for misaligned data access, at the cost of increasing the data size. When developing your code, you should consider this tradeoff and use the option which gives the best performance.

5.4.2 Stack Alignment for 128-bit SIMD Technologies

For best performance, the Streaming SIMD Extensions and Streaming SIMD Extensions 2 require their memory operands to be aligned to 16-byte boundaries. Unaligned data can cause significant performance penalties compared to aligned data. However, the existing software conventions for IA-32 (STDCALL, CDECL, FASTCALL) as implemented in most compilers, do not provide any mechanism for ensuring that certain local data and certain parameters are 16-byte aligned. Therefore, Intel has defined a new set of IA-32 software conventions for alignment to support the new __M128* datatypes (__M128, __M128D, and __M218I). These meet the following conditions:

- Functions that use Streaming SIMD Extensions or Streaming SIMD Extensions 2 data need to provide a 16-byte aligned stack frame.
- __M128* parameters need to be aligned to 16-byte boundaries, possibly creating "holes" (due to padding) in the argument block.

The new conventions presented in this section as implemented by the Intel C++ Compiler can be used as a guideline for an assembly language code as well. In many cases, this section assumes the use of the __M128* data types, as defined by the Intel C++ Compiler, which represents an array of four 32-bit floats.

5.4.3 Data Alignment for MMX™ Technology

Many compilers enable alignment of variables using controls. This aligns variable bit lengths to the appropriate boundaries. If some of the variables are not appropriately aligned as specified, you can align them using the C algorithm in Example 5-18.

Example 5-18. C Algorithm for 64-bit Data Alignment

```
/* Make newp a pointer to a 64-bit aligned array of NUM_ELEMENTS 64-bit elements. */
double *p, *newp;
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));
newp = (p+7) & (~0x7);
```

The algorithm in Example 5-18 aligns an array of 64-bit elements on a 64-bit boundary. The constant of 7 is derived from one less than the number of bytes in a 64-bit element, or 8-1. Aligning data in this manner avoids the significant performance penalties that can occur when an access crosses a cache line boundary.

Another way to improve data alignment is to copy the data into locations that are aligned on 64-bit boundaries. When the data is accessed frequently, this can provide a significant performance improvement.

5.4.4 Data Alignment for 128-bit data

Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers used by Intel SSE, Intel SSE2, Intel SSE3, and Intel SSSE3. This must be done to avoid severe performance penalties and, at worst, execution faults.

There are MOVE instructions (and intrinsics) that allow unaligned data to be copied to and out of XMM registers when not using aligned data, but such operations are much slower than aligned accesses. If data is not 16-byte-aligned and the programmer or the compiler does not detect this and uses the aligned instructions, a fault occurs. So keep data 16-byte-aligned. Such alignment also works for MMX technology code, even though MMX technology only requires 8-byte alignment.

The following describes alignment techniques for Pentium 4 processor as implemented with the Intel C++ Compiler.

5.4.4.1 Compiler-Supported Alignment

The Intel C++ Compiler provides the following methods to ensure that the data is aligned.

Alignment by F32vec4 or __m128 Data Types

When the compiler detects F32VEC4 or __M128 data declarations or parameters, it forces alignment of the object to a 16-byte boundary for both global and local data, as well as parameters. If the declaration is within a function, the compiler also aligns the function's stack frame to ensure that local data and parameters are 16-byte-aligned. For details on the stack frame layout that the compiler generates for both debug and optimized ("release"-mode) compilations, refer to Intel's compiler documentation.

__declspec(align(16)) specifications

These can be placed before data declarations to force 16-byte alignment. This is useful for local or global data declarations that are assigned to 128-bit data types. The syntax for it is

```
__declspec(align(integer-constant))
```

where the INTEGER-CONSTANT is an integral power of two but no greater than 32. For example, the following increases the alignment to 16-bytes:

```
__declspec(align(16)) float buffer[400];
```

The variable BUFFER could then be used as if it contained 100 objects of type __M128 or F32VEC4. In the code below, the construction of the F32VEC4 object, X, will occur with aligned data.

```
void foo() {
    F32vec4 x = *(__m128 *) buffer;
    ...
}
```

Without the declaration of __DECLSPEC(ALIGN(16)), a fault may occur.

Alignment by Using a UNION Structure

When feasible, a UNION can be used with 128-bit data types to allow the compiler to align the data structure by default. This is preferred to forcing alignment with __DECLSPEC(ALIGN(16)) because it exposes the true program intent to the compiler in that __M128 data is being used. For example:

```
union {
    float f[400];
    __m128 m[100];
} buffer;
```

Now, 16-byte alignment is used by default due to the __M128 type in the UNION; it is not necessary to use __DECLSPEC(ALIGN(16)) to force the result.

In C++ (but not in C) it is also possible to force the alignment of a CLASS/STRUCT/UNION type, as in the code that follows:

```
struct __declspec(align(16)) my_m128
{
     float f[4];
};
```

If the data in such a CLASS is going to be used with the Intel SSE or Intel SSE2, it is preferable to use a UNION to make this explicit. In C++, an anonymous UNION can be used to make this more convenient:

```
class my_m128 {
    union {
        __m128 m;
        float f[4];
    };
};
```

Because the UNION is anonymous, the names, M and F, can be used as immediate member names of MY__M128. Note that __DECLSPEC(ALIGN) has no effect when applied to a CLASS, STRUCT, or UNION member in either C or C++.

Alignment by Using __m64 or DOUBLE Data

In some cases, the compiler aligns routines with __M64 or DOUBLE data to 16-bytes by default. The command-line switch, -QSFALIGN16, limits the compiler so that it only performs this alignment on routines that contain 128-bit data. The default behavior is to use -QSFALIGN8. This switch instructs the complier to align routines with 8- or 16-byte data types to 16 bytes.

See Intel® C++ Compiler Classic Developer Guide and Reference for details.

5.5 IMPROVING MEMORY UTILIZATION

Memory performance can be improved by rearranging data and algorithms for Intel SSE, Intel SSE2, and MMX technology intrinsics. Methods for improving memory performance involve working with the following:

- Data structure layout.
- Strip-mining for vectorization and memory utilization.
- Loop-blocking.

Using the cacheability instructions, prefetch and streaming store, also greatly enhance memory utiliza-tion. See also: Chapter 9, “Optimizing Cache Usage.”

5.5.1 Data Structure Layout

For certain algorithms, like 3D transformations and lighting, there are two basic ways to arrange vertex data. The traditional method is the array of structures (AoS) arrangement, with a structure for each vertex (Example 5-19). However this method does not take full advantage of SIMD technology capabilities.

Example 5-19. AoS Data Structure

```
typedef struct{
    float x,y,z;
    int a,b,c;
    . . .
} Vertex;
Vertex Vertices[NumOfVertices];
```

The best processing method for code using SIMD technology is to arrange the data in an array for each coordinate (Example 5-20). This data arrangement is called structure of arrays (SoA).

Example 5-20. SoA Data Structure

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int a[NumOfVertices];
    int b[NumOfVertices];
    int c[NumOfVertices];
    . . .
} VerticesList;
VerticesList Vertices;
```

There are two options for computing data in AoS format: perform operation on the data as it stands in AoS format, or re-arrange it (swizzle it) into SoA format dynamically. See Example 5-21 for code samples of each option based on a dot-product computation.

Example 5-21. AoS and SoA Code Samples

```
; The dot product of an array of vectors (Array) and a fixed vector (Fixed) is a
; common operation in 3D lighting operations, where Array = (x0,y0,z0),(x1,y1,z1),...
; and Fixed = (xF,yF,zF)
; A dot product is defined as the scalar quantity d0 = x0*xF + y0*yF + z0*zF.

;
; AoS code
; All values marked DC are "don't-care."

; In the AOS model, the vertices are stored in the xyz format
movaps  xmm0, Array        ; xmm0 = DC, x0, y0, z0
movaps  xmm1, Fixed        ; xmm1 = DC, xF, yF, zF
mulps   xmm0, xmm1         ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhlps xmm, xmm0          ; xmm = DC, DC, DC, x0*xF

addps   xmm1, xmm0         ; xmm0 = DC, DC, DC,
                           ; x0*xF+z0*zFmovaps  xmm2, xmm1
shufps  xmm2, xmm2,55h     ; xmm2 = DC, DC,    DC,    y0*yF
addps   xmm2, xmm1         ; xmm1 = DC, DC,    DC,
                           ; x0*xF+y0*yF+z0*zF
```

Example 5-21. AoS and SoA Code Samples (Contd.)

```
; SoA code
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3
; A = xF,xF,xF,xF
; B = yF,yF,yF,yF
; C = zF,zF,zF,zF

movaps xmm0, X          ; xmm0 = x0,x1,x2,x3
movaps xmm1, Y          ; xmm0 = y0,y1,y2,y3
movaps xmm2, Z          ; xmm0 = z0,z1,z2,z3
mulps  xmm0, A          ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps  xmm1, B          ; xmm1 = y0*yF, y1*yF, y2*yF, y3*xF
mulps  xmm2, C          ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps  xmm0, xmm1
addps  xmm0, xmm2       ; xmm0 = (x0*xF+y0*yF+z0*zF), ...
```

Performing SIMD operations on the original AoS format can require more calculations and some operations do not take advantage of all SIMD elements available. Therefore, this option is generally less efficient.

The recommended way for computing data in AoS format is to swizzle each set of elements to SoA format before processing it using SIMD technologies. Swizzling can either be done dynamically during program execution or statically when the data structures are generated. See Chapter 6, "Optimizing for SIMD Integer Applications" and Chapter 7, "Optimizing for SIMD Floating-Point Applications" for examples. Performing the swizzle dynamically is usually better than using AoS, but can be somewhat inefficient because there are extra instructions during computation. Performing the swizzle statically, when data structures are being laid out, is best as there is no runtime overhead.

As mentioned earlier, the SoA arrangement allows more efficient use of the parallelism of SIMD technologies because the data is ready for computation in a more optimal vertical manner: multiplying components X0,X1,X2,X3 by XF,XF,XF,XF using 4 SIMD execution slots to produce 4 unique results. In contrast, computing directly on AoS data can lead to horizontal operations that consume SIMD execution slots but produce only a single scalar result (as shown by the many "don't-care" (DC) slots in Example 5-21).

Use of the SoA format for data structures can lead to more efficient use of caches and bandwidth. When the elements of the structure are not accessed with equal frequency, such as when element x, y, z are accessed ten times more often than the other entries, then SoA saves memory and prevents fetching unnecessary data items a, b, and c.

Example 5-22. Hybrid SoA Data Structure

```
NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];

} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    . . .
```

Example 5-22. Hybrid SoA Data Structure (Contd.)

```
} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];
```

Note that SoA can have the disadvantage of requiring more independent memory stream references. A computation that uses arrays X, Y, and Z (see Example 5-20) would require three separate data streams. This can require the use of more prefetches, additional address generation calculations, as well as having a greater impact on DRAM page access efficiency.

There is an alternative: a hybrid SoA approach blends the two alternatives (see Example 5-22). In this case, only 2 separate address streams are generated and referenced: one contains XXXX, YYYY,ZZZZ, ZZZZ,... and the other AAAA, BBBB, CCCC, AAAA, DDDD,... . The approach prevents fetching unnecessary data, assuming the variables X, Y, Z are always used together; whereas the variables A, B, C would also be used together, but not at the same time as X, Y, Z.

The hybrid SoA approach ensures:

- Data is organized to enable more efficient vertical SIMD computation.
- Simpler/less address generation than AoS.
- Fewer streams, which reduces DRAM page misses.
- Use of fewer prefetches, due to fewer streams.
- Efficient cache line packing of data elements that are used concurrently.

With the advent of the SIMD technologies, the choice of data organization becomes more important and should be carefully based on the operations to be performed on the data. This will become increasingly important in the Pentium 4 processor and future processors. In some applications, traditional data arrangements may not lead to the maximum performance. Application developers are encouraged to explore different data arrangements and data segmentation policies for efficient computation. This may mean using a combination of AoS, SoA, and Hybrid SoA in a given application.

5.5.2 Strip-Mining

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure by:

- Increasing the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- Reducing the number of iterations of the loop by a factor of the length of each "vector," or number of operations being performed per SIMD operation. In the case of Intel SSE, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

Consider Example 5-23:

Example 5-23. Pseudo-Code Before Strip Mining

```
typedef struct _VERTEX {
        float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;

main()
{
        Vertex_rec v[Num];
        ....
        for (i=0; i<Num; i++) {
          Transform(v[i]);
        }
        for (i=0; i<Num; i++) {
          Lighting(v[i]);
        }
        ....
}
```

The main loop consists of two functions: transformation and lighting. For each object, the main loop calls a transformation routine to update some data, then calls the lighting routine to further work on the data. If the size of array V[NUM] is larger than the cache, then the coordinates for V[I] that were cached during TRANSFORM(V[I]) will be evicted from the cache by the time we do LIGHTING(V[I]). This means that V[I] will have to be fetched from main memory a second time, reducing performance.

In Example 5-24, the computation has been strip-mined to a size STRIP_SIZE. The value STRIP_SIZE is chosen such that STRIP_SIZE elements of array V[NUM] fit into the cache hierarchy. By doing this, a given element V[I] brought into the cache by TRANSFORM(V[I]) will still be in the cache when we perform LIGHTING(V[I]), and thus improve performance over the non-strip-mined code.

Example 5-24. Strip Mined Code

```
MAIN()
{
   Vertex_rec v[Num];
   ....
   for (i=0; i < Num; i+=strip_size) {
     FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
         TRANSFORM(V[J]);
    }
     FOR (J=I; J < MIN(NUM, I+STRIP_SIZE); J++) {
         LIGHTING(V[J]);
     }
   }
}
```

5.5.3 Loop Blocking

Loop blocking is another useful technique for memory performance optimization. The main purpose of loop blocking is also to eliminate as many cache misses as possible. This technique transforms the memory domain of a given problem into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse. In fact, one can treat loop blocking as strip mining in two or more dimensions.

Consider the code in Example 5-23 and access pattern in Figure 5-5. The two-dimensional array A is referenced in the J (column) direction and then referenced in the I (row) direction (column-major order); whereas array B is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array A and B for the code in Example 5-25 would be 1 and MAX, respectively.

Example 5-25. Loop Blocking

```
A. Original Loop
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
        A[i,j] = A[i,j] + B[j, i];
    }
}

B. Transformed Loop after Blocking
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```

For the first iteration of the inner loop, each access to array B will generate a cache miss. If the size of one row of array A, that is, A[2, 0:MAX-1], is large enough, by the time the second iteration starts, each access to array B will always generate a cache miss. For instance, on the first iteration, the cache line containing B[0, 0:7] will be brought in when B[0,0] is referenced because the float type variable is four bytes and each cache line is 32 bytes. Due to the limitation of cache capacity, this line will be evicted due to conflict misses before the inner loop reaches the end.

For the next iteration of the outer loop, another cache miss will be generated while referencing B[0, 1]. In this manner, a cache miss occurs when each element of array B is referenced, that is, there is no data reuse in the cache at all for array B.

This situation can be avoided if the loop is blocked with respect to the cache size. In Figure 5-5, a BLOCK_SIZE is selected as the loop blocking factor. Suppose that BLOCK_SIZE is 8, then the blocked chunk of each array will be eight cache lines (32 bytes each). In the first iteration of the inner loop, A[0, 0:7] and B[0, 0:7] will be brought into the cache. B[0, 0:7] will be completely consumed by the first iteration of the outer loop. Consequently, B[0, 0:7] will only experience one cache miss after applying loop blocking optimization in lieu of eight misses for the original algorithm.

As illustrated in Figure 5-5, arrays A and B are blocked into smaller rectangular chunks so that the total size of two blocked A and B chunks is smaller than the cache size. This allows maximum data reuse.

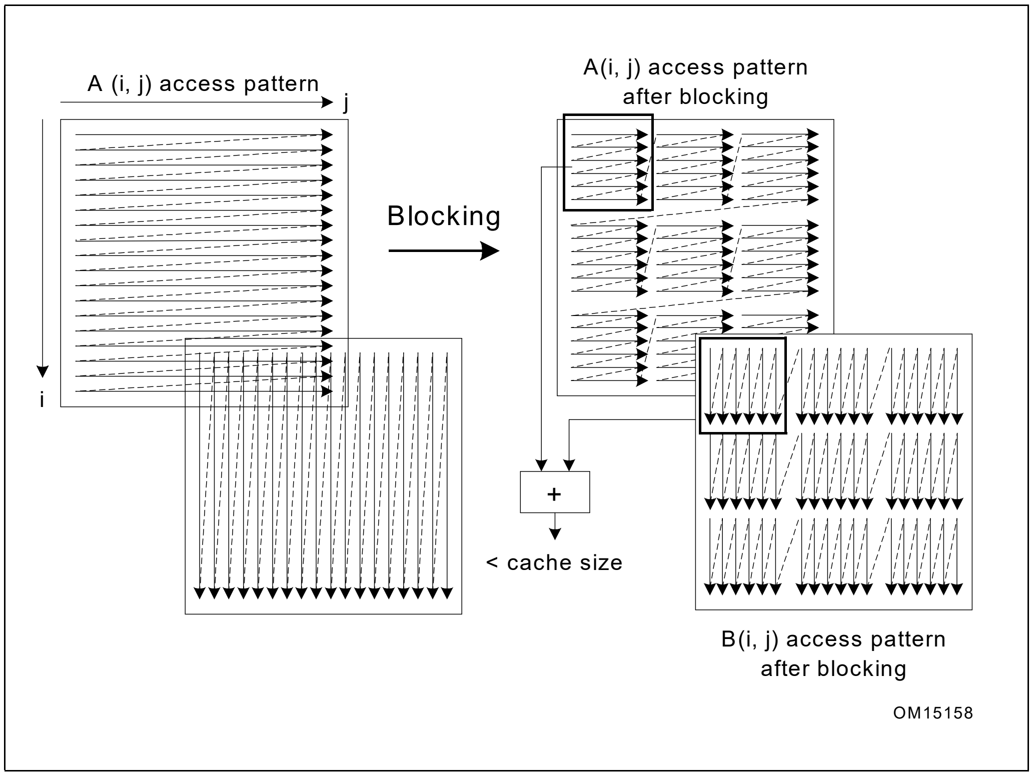

A (i, j) access pattern
j
i
Blocking
A(i, j) access pattern after blocking
+
< cache size
B(i, j) access pattern after blocking
OM15158

Figure 5-5. Loop Blocking Access Pattern

As one can see, all the redundant cache misses can be eliminated by applying this loop blocking technique. If MAX is huge, loop blocking can also help reduce the penalty from DTLB (data translation lookaside buffer) misses. In addition to improving the cache/memory performance, this optimization technique also saves external bus bandwidth.

5.6 INSTRUCTION SELECTION

The following section gives some guidelines for choosing instructions to complete a task.

One barrier to SIMD computation can be the existence of data-dependent branches. Conditional moves can be used to eliminate data-dependent branches. Conditional moves can be emulated in SIMD computation by using masked compares and logicals, as shown in Example 5-26. SSE4.1 provides packed blend instruction that can vectorize data-dependent branches in a loop.

Example 5-26. Emulation of Conditional Moves

```
High-level code:
__declspec(align(16)) short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT], D[MAX_ELEMENT],
E[MAX_ELEMENT];

for (i=0; i<MAX_ELEMENT; i++) {
    if (A[i] > B[i]) {
        C[i] = D[i];
    } else {
        C[i] = E[i];
    }
```

Example 5-26. Emulation of Conditional Moves (Contd.)

```
}
MMX assembly code processes 4 short values per iteration:
    xor       eax, eax

top_of_loop:
    movq      mm0, [A + eax]
    pcmpgtwxmm0, [B + eax]; Create compare mask
    movq      mm1, [D + eax]
    pand      mm1, mm0; Drop elements where A<B
    pandn     mm0, [E + eax] ; Drop elements where A>B

    por       mm0, mm1; Crete single word
    movq      [C + eax], mm0
    add       eax, 8
    cmp       eax, MAX_ELEMENT*2
    jle       top_of_loop

SSE4.1 assembly processes 8 short values per iteration:
    xor       eax, eax

top_of_loop:
    movdqq    xmm0, [A + eax]
    pcmpgtwxmm0, [B + eax]; Create compare mask
    movdqa    xmm1, [E + eax]
    pblendv   xmm1, [D + eax], xmm0;
    movdqa    [C + eax], xmm1;
    add       eax, 16
    cmp       eax, MAX_ELEMENT*2
    jle       top_of_loop
```

If there are multiple consumers of an instance of a register, group the consumers together as closely as possible. However, the consumers should not be scheduled near the producer.

5.7 TUNING THE FINAL APPLICATION

The best way to tune your application once it is functioning correctly is to use a profiler that measures the application while it is running on a system. Intel VTune Amplifier XE can help you determine where to make changes in your application to improve performance. Using Intel VTune Amplifier XE can help you with various phases required for optimized performance. See Appendix A.3.1 for details. After every effort to optimize, you should check the performance gains to see where you are making your major optimization gains.

4. Updates to Chapter 20

Change bars and **violet** text show changes to Chapter 20 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual:* Multicore and Hyper-Threading Technology.

Changes to this chapter:

- Section 20.5.3:
  - Figures 20-3 and 20-4 were changed into tables due to illegibility. These tables are 20-3, 20-4, 20-5, and 20-6.

CHAPTER 20 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

This chapter aims to help low-level DL programmers optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP microarchitecture. It extends the public documentation on Optimizing DL code with DL Boost instructions in Section 20.8.

It explains how to detect processor support in Intel® Advanced Matrix Extensions (Intel® AMX) Architecture (Section 20.1). It provides an overview of Intel AMX architecture (Section 20.2) and presents Intel AMX instruction throughput and latency (Section 20.3). It also discusses software optimization opportunities for Intel AMX (Section 20.5 through Section 20.17), TileConfig/TileRelease and compiler ABI (Section 20.18), Intel AMX state management and system software aspects (Section 20.19), and the use of Intel AMX for higher precision GEMMs (Section 20.20).

Table 20-1. Intel® AMX-Related Links

Description	URL
Intel® AMX architecture definitions in the Intel® 64 and IA-32 Architecture Software Developer's Manual	https://www.intel.com/sdm
Buildable and executable templates of code examples for this chapter.	https://github.com/intel/optimization-manual
Open VINO™ Optimization Guide	https://docs.openvino.ai/latest/openvino_docs_optimization_guide_dldt_optimization_guide.html
oneDNN GitHub	https://github.com/oneapi-src/oneDNN
oneDNN documentation	https://oneapi-src.github.io/oneDNN/
Intel® Optimization TensorFlow Installation Guide	https://www.intel.com/content/www/us/en/developer/articles/guide/optimization-for-tensorflow-installation-guide.html
PyTorch Landing Page	https://pytorch.org/
PyTorch GitHub	https://github.com/pytorch/pytorch
Intel® Neural Compressor (INC) GitHub	https://github.com/intel/neural-compressor
Tips for measuring the performance of matrix multiplication using Intel® MKL	https://www.intel.com/content/www/us/en/developer/articles/technical/a-simple-example-to-measure-the-performance-of-an-intel-mkl-function.html
Intel® AMX ABI	https://gitlab.com/x86-psABIs/x86-64-ABI/-/wikis/home
GitHub Repository	https://github.com/intel/optimization-manual
Using dynamically enabled XSTATE features in Linux user space applications	https://www.kernel.org/doc/html/latest/x86/xstate.html

Table 20-1. Intel® AMX-Related Links

Description	URL
Using dynamically enabled XSTATE features in Windows user space applications	https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getenabledxstatefeatures
	https://docs.microsoft.com/es-es/windows/win32/api/winbase/nf-winbase-enableprocessoptionalxstatefeatures
	https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getthreadenabledxstatefeaturesv
	https://docs.microsoft.com/en-us/windows/win32/api/process-threadsapi/nf-processthreadsapi-updateprocthreadattribute

20.1 DETECTING INTEL® AMX SUPPORT

Use the CPUID instruction described in Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual to find out whether the processor you are executing on supports Intel AMX at the hardware level.

Specifically, when issuing the CPUID instruction with EAX register set to 7 and ECX register set to 0, the instruction returns in the EDX register an indication on Intel AMX support of bits 22, 24, 25. They are all set to 0 if Intel AMX is not supported and all set to 1 if it is supported by the processor.

Next step is check whether the OS has enabled Intel AMX state. For that you first need to issue the CPUID instruction again to check whether the OS supports the XGETBV instruction, then use it to check whether the OS has enabled the Intel AMX state save/restore.

When issuing the CPUID instruction with EAX register set to 1, the instruction returns an indication of XGETBV support in bit 26 of the ECX register. If bit 26 is set, when issuing the XGETBV instruction with ECX register set to 0, the instruction returns an indication on OS support in saving and restoring Intel AMX state in bits 17 and 18 of the EAX register. Both bits should be set in order to use the Intel AMX instructions. For additional CPUID information about Intel AMX, see Chapter 3.3 of the Intel® 64 and IA-32 Architecture Software Developer's Manual

Operating systems may require calling an OS API to allocate Intel AMX state. Visit LinuxAPI and Windows APIs for more detailed information. Please see Section 20.19 for more information about Intel AMX state management.

20.2 INTEL® AMX MICROARCHITECTURE OVERVIEW

General Intel AMX microarchitecture overview is available in Chapter 18 of Volume 1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual.

20.2.1 INTEL® AMX FREQUENCIES

Discussion on the connection between max frequency, frequency license, and Instruction Set Architecture covering Intel AVX technologies up to Intel® AVX-512 Instruction Set, is available in Section 2.5.3. Intel AMX adds yet another license level whose max frequency is usually lower than that of the Intel AVX-512 license.

When the Intel AMX unit utilization is lower than 15%, the processor may exceed the nominal max frequency associated with Intel AMX license.

20.3 INTEL® AMX INSTRUCTIONS THROUGHPUT AND LATENCY

Several Intel AMX instructions are available. Two instructions (TileLoad*) load data from the memory hierarchy into the tile registers and one instruction (TileStore) stores the contents of a tile register into the DCU (Data Cache Unit–first level cache). Other instructions (TDP*) execute the matrix multiplication, operating on two input tile registers and writing the result into a third tile register. Additionally, there are some less-frequently used instructions. The following table provides the instruction throughput and latency counted in cycles.

Table 20-2. Intel® AMX Instruction Throughput and Latency

Instruction	Throughput	Latency
LDTILECFG	Not Relevant	204
STTILECFG	Not Relevant	19
TILETRELEASE	Not Relevant	13
TDP/*	16	52
TILELOADD	8	45
TILELOADDT1	33	48
TILESTORED	16	
TILEZERO	0	16

NOTE

Due to the high latency of the LDTILECFG instruction we recommend issuing a single pair of LDTILECFG and TILERELEASE operations per Intel AMX-based DL layer implementation.

20.4 DATA STRUCTURE ALIGNMENT

GEMM and Convolution input/output data structures must be 64-byte aligned for optimal performance but should not be aligned to 128-byte, 256-byte, etc. For more details, see Tip 6 in *Tips for Measuring the Performance of Matrix Multiplication Using Intel® MKL*.

20.5 GEMMS / CONVOLUTIONS

20.5.1 NOTATION

The following notation is used for the matrices (A, B, C) and the dimensions (M, K, N) in matrix multiplication (GEMM).

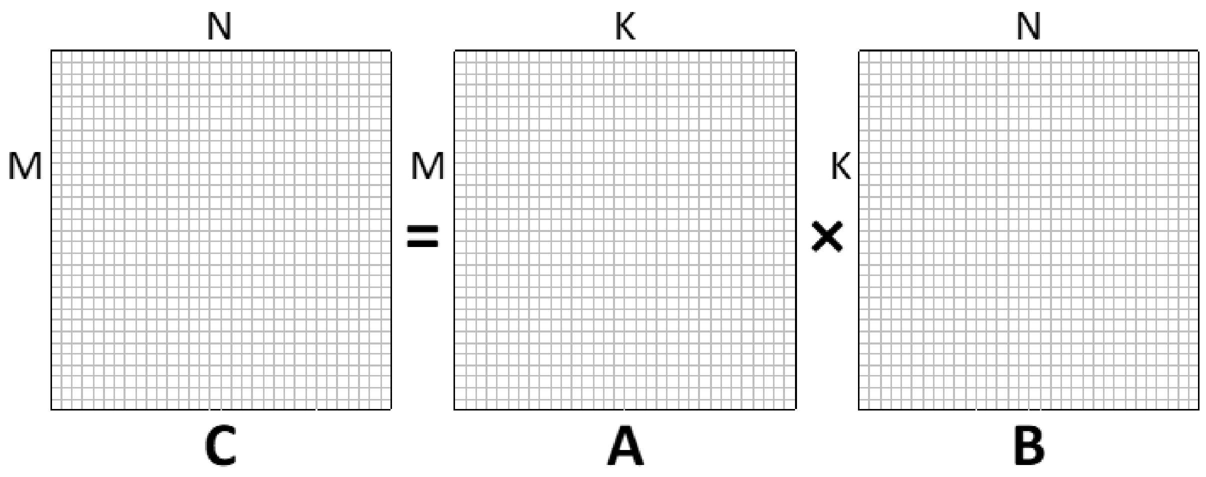


Figure 20-1. Matrix Notation

20.5.2 TILES IN THE INTEL® AMX ARCHITECTURE

The Intel AMX instruction set operates on tiles: large two-dimensional registers with configurable dimensions. The configuration is dependent on the type of tile.

- A-tiles can have between 1-16 rows and 1-MAX_TILE_K columns.
- B-tiles can have between 1-MAX_TILE_K rows and 1–16 columns.
- C-tiles can have between 1-16 rows and 1–16 columns.

MAX_TILE_K=64/sizeof(type_t), and type_t is the type of the data being operated on. Therefore, MAX_TILE_K=64 for (u)int8 data, and MAX_TILE_K=32 for bfloat16 data. The dimensions here are mathematical/logical. For mapping to tile register configuration parameters, see the Intel® Architecture Instruction Set Extensions Programming Reference.

The type of data residing in the tiles also varies depending on the type of tile.

A tiles and B tiles contain data of `type_t`, which can be (u)int8 or bfloat16.

- C tiles contain data of type res_type_t:
- int32 if type_t=(u)int8
- float if type_t=bfloat16

Thus, a maximum-sized tile multiplication operation for (u)int8 data type looks this way:

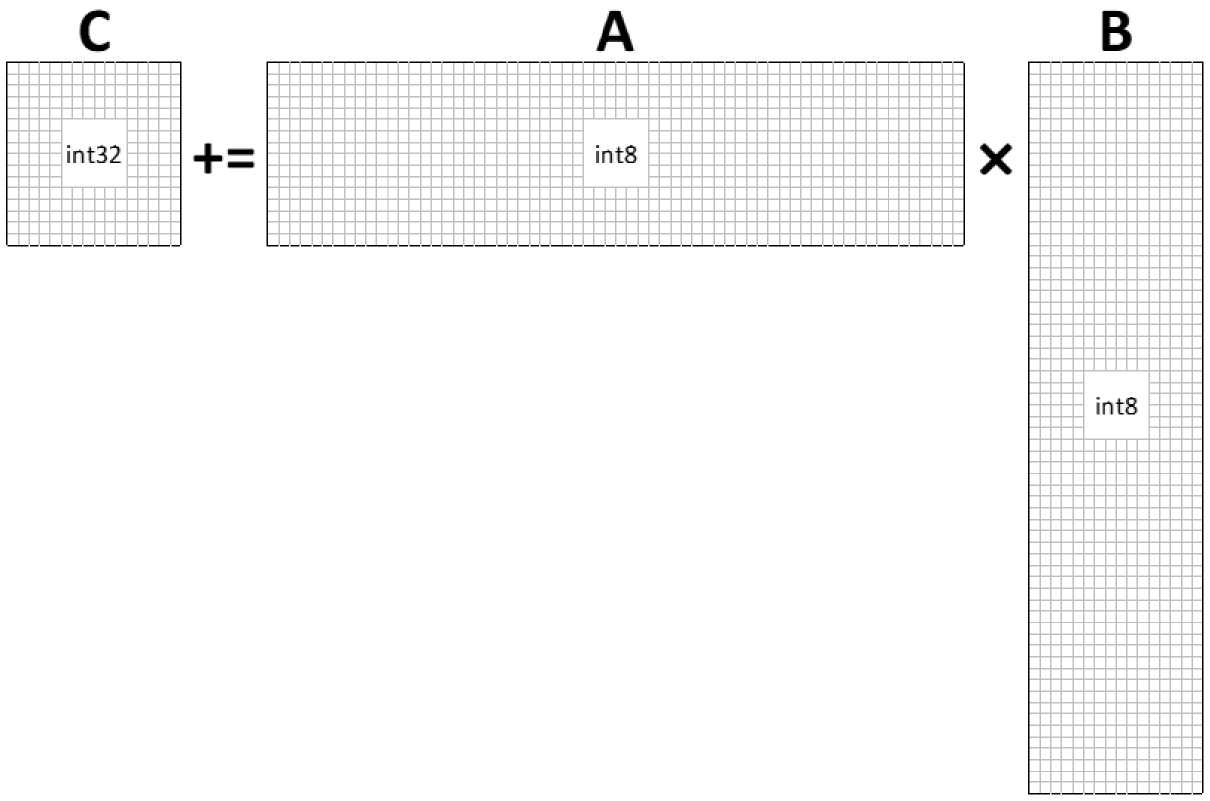


Figure 20-2. Intel® AMX Multiplication with Max-sized int8 Tiles

TileLoad and TileStore Instructions

The tiles are loaded from memory with the TileLoad instruction and stored to memory with a TileStore instruction. The TileLoad/TileStore instructions receive the following parameters:

- The destination/source tile of the TileLoad/TileStore.
- The source/destination location in memory for the TileLoad/TileStore.
- The stride (bytes) in memory between subsequent rows of the tile.

Lines 6—10 in Example 20-1 illustrate how a tile is loaded from memory.

Example 20-1. Pseudo-Code for the Tilezero, TileLoad, and TileStore Instructions

```
template<size_t rows, size_t bytes_cols> class tile {
public:
  friend void tilezero(tile& t) {
    memset(t.v, 0, sizeof(v));
  }
  friend void tileload(tile& t, void* src, size_t bytes_stride) {
    for (size_t row = 0; row < rows; ++row)
      for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
        t.v[row][bcol] = static_cast<int8_t*>(src)[row*bytes_stride + bcol];
  }
friend void tilestore(tile& t, void* dst, size_t bytes_stride) {
    for (size_t row = 0; row < rows; ++row)
      for (size_t bcol = 0; bcol < bytes_cols; ++bcol)
        static_cast<int8_t*>(dst)[row*bytes_stride + bcol] = t.v[row][bcol];
  }
template <class TC, class TA, class TB>
friend void tdp(TC &tC, TA &tA, TB &tB);
private:
  int8_t v[rows][bytes_cols];
};

// clang-format on

template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB)
}
```

For the sake of readability, a tile template class abstraction is introduced. The number of rows in the tile and the number of column bytes per row parametrizes the abstraction.

20.5.3 B MATRIX LAYOUT

Like the Intel® DL Boost use case, the B matrix must undergo a re-layout before it can be used within the corresponding Intel AMX multiply instruction. The re-layout procedure is as follows:

Example 20-2. B Matrix Re-Layout Procedure

```
#define KPACK (4/sizeof(type_t))                  // Vertical K packing into Dword

type_t B_mem_orig[K][N];                          // Original B matrix
type_t B_mem[K/KPACK][N][KPACK];                  // Re-laid B matrix

for (int k = 0; k < K; ++k)
  for (int n = 0; n < N; ++n)
    B_mem[k/KPACK][n][k%KPACK] = B_mem_orig[k][n];
```

The following tables illustrate the data re-layout process for a 64x16 int8 B matrix and a 32x16 bfloat16 B matrix (corresponding to the maximum-sized B-tile):

Table 20-3. Original Layout of 32x16 bfloat16 B-Matrix

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	95	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271
272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303
304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319
320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335
336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351
352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367
368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383
384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399
400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415
416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431
432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447
448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463
464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479
480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495
496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511

Table 20-4. Re-Layout of 32x16 bfloat16 B-Matrix

	0	16	1	17	2	18	3	19	4	20	5	21	6	22	7	23	8	24	9	25	10
	32	48	33	49	34	50	35	51	36	52	37	53	38	54	39	55	40	56	41	57	42
	64	80	65	81	66	82	67	83	68	84	69	85	70	86	71	87	72	88	73	89	74
	96	112	97	113	98	114	99	115	100	116	101	117	102	118	103	119	104	120	105	121	106
	128	114	129	145	130	146	131	147	132	148	133	149	134	150	135	151	136	152	137	153	138
	160	176	161	177	162	178	163	179	164	180	165	181	166	182	167	183	168	184	169	185	170
	192	208	193	209	194	210	195	211	196	212	197	213	198	214	199	215	200	216	201	217	202
	224	240	225	241	226	242	227	243	228	244	229	245	230	246	231	247	232	248	233	249	234
	256	272	257	273	258	274	259	275	260	276	261	277	262	278	263	279	264	280	265	281	266
	288	304	289	305	290	306	291	307	292	308	293	309	294	310	295	311	296	312	297	313	298
	320	336	321	337	322	338	323	339	324	340	325	341	326	342	327	343	328	344	329	345	330
	352	368	353	369	354	370	355	371	356	372	357	373	358	374	359	375	360	376	361	377	362
	384	400	385	401	386	402	387	403	388	404	389	405	390	406	391	407	392	408	393	409	394
	416	432	417	433	418	434	419	435	420	436	421	437	422	438	423	439	424	440	425	441	426
	448	464	449	465	450	466	451	467	452	468	453	469	454	470	455	471	456	472	457	473	458
	480	496	481	497	482	498	483	499	484	500	485	501	486	502	487	503	488	504	489	505	490

Table 20-4. (Contd.)Re-Layout of 32x16 bfloat16 B-Matrix

	26	11	27	12	28	13	29	14	30	15	31
	58	43	59	44	60	45	61	46	62	47	63
	90	75	91	76	92	77	93	78	95	79	95
	122	107	123	108	124	109	125	110	126	111	127
	154	139	155	140	156	141	157	142	158	143	159
	186	171	187	172	188	173	189	174	190	175	191
	218	203	219	204	220	205	221	206	222	207	223
	250	235	251	236	252	237	253	238	254	239	255
	282	267	283	268	284	269	285	270	286	271	287
	314	299	315	300	316	301	317	302	318	303	319
	346	331	347	332	348	333	349	334	350	335	351
	378	363	379	364	380	365	381	366	382	367	383
	410	395	411	396	412	397	413	398	414	399	415
	442	427	443	428	444	429	445	430	446	431	447
	474	459	475	460	476	461	477	462	478	463	479
	506	491	507	492	508	493	509	494	510	495	511

Table 20-5. Original Layout of 64 x 16 unt8 B-Matrix

4	5	6	7	8	9	10	11	12	13	14	15
20	21	22	23	24	25	26	27	28	29	30	31
36	37	38	39	40	41	42	43	44	45	46	47
52	53	54	55	56	57	58	59	60	61	62	63
68	69	70	71	72	73	74	75	76	77	78	79
84	85	86	87	88	89	90	91	92	93	94	95
100	101	102	103	104	105	106	107	108	109	110	111
116	117	118	119	120	121	122	123	124	125	126	127
132	133	134	135	136	137	138	139	140	141	142	143
148	149	150	151	152	153	154	155	156	157	158	159
164	165	166	167	168	169	170	171	172	173	174	175
180	181	182	183	184	185	186	187	188	189	190	191
196	197	198	199	200	201	202	203	204	205	206	207
212	213	214	215	216	217	218	219	220	221	222	223
228	229	230	231	232	233	234	235	236	237	238	239
244	245	246	247	248	249	250	251	252	253	254	255
260	261	262	263	264	265	266	267	268	269	270	271
276	277	278	279	280	281	282	283	284	285	286	287

Table 20-5. Original Layout of 64 x 16 unt8 B-Matrix

292	293	294	295	296	297	298	299	300	301	302	303
308	309	310	311	312	313	314	315	316	317	318	319
324	325	326	327	328	329	330	331	332	333	334	335
340	341	342	343	344	345	346	347	348	349	350	351
356	357	358	359	360	361	362	363	364	365	366	367
372	373	374	375	376	377	378	379	380	381	382	383
388	389	390	391	392	393	394	395	396	397	398	399
404	405	406	407	408	409	410	411	412	413	414	415
420	421	422	423	424	425	426	427	428	429	430	431
436	437	438	439	440	441	442	443	444	445	446	447
452	453	454	455	456	457	458	459	460	461	462	463
468	469	470	471	472	473	474	475	476	477	478	479
484	485	486	487	488	489	490	491	492	493	494	495
500	501	502	503	504	505	506	507	508	509	510	511
516	517	518	519	520	521	522	523	524	525	526	527
532	533	534	535	536	537	538	539	540	541	542	543
548	549	550	551	552	553	554	555	556	557	558	559
564	565	566	567	568	569	570	571	572	573	574	575
580	581	582	583	584	585	586	587	588	589	590	591
596	597	598	599	600	601	602	603	604	605	606	607
612	613	614	615	616	617	618	619	620	621	622	623
628	629	630	631	632	633	634	635	636	637	638	639
644	645	646	647	648	649	650	651	652	653	654	655
660	661	662	663	664	665	666	667	668	669	670	671
676	677	678	679	680	681	682	683	684	685	686	687
692	693	694	695	696	697	698	699	700	701	702	703
708	709	710	711	712	713	714	715	716	717	718	719
724	725	726	727	728	729	730	731	732	733	734	735
740	741	742	743	744	745	746	747	748	749	750	751
756	757	758	759	760	761	762	763	764	765	766	767
772	773	774	775	776	777	778	779	780	781	782	783
788	789	790	791	792	793	794	795	796	797	798	799
804	805	806	807	808	809	810	811	812	813	814	815
820	821	822	823	824	825	826	827	828	829	830	831
836	837	838	839	840	841	842	843	844	845	846	847
852	853	854	855	856	857	858	859	860	861	862	863
868	869	870	871	872	873	874	875	876	877	878	879
884	885	886	887	888	889	890	891	892	893	894	895
900	901	902	903	904	905	906	907	908	909	910	911
916	917	918	919	920	921	922	923	924	925	926	927

Table 20-5. Original Layout of 64 x 16 unt8 B-Matrix

932	933	934	935	936	937	938	939	940	941	942	943
948	949	950	951	952	953	954	955	956	957	958	959
964	965	966	967	968	969	970	971	972	973	974	975
980	981	982	983	984	985	986	987	988	989	990	991
996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007
1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

Table 20-6. Re-Layout of 64 x 16 int8 B-Matrix

0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51	4	20	36	52	5
64	80	96	112	65	81	97	113	66	82	98	114	67	83	99	115	68	84	100	116	69
128	144	160	176	129	145	161	177	130	146	162	178	131	147	163	179	132	148	164	180	133
192	208	224	240	193	209	225	241	194	210	226	242	195	211	227	243	196	212	228	244	197
256	272	288	304	257	273	289	305	258	274	290	306	259	275	291	307	260	276	292	308	261
320	336	352	368	321	337	353	369	322	338	354	370	323	339	355	371	324	340	356	372	325
384	400	416	432	385	401	417	433	386	402	418	434	387	403	419	435	388	404	420	436	389
448	464	480	496	449	465	481	497	450	466	482	498	451	467	483	499	452	468	484	500	453
512	528	544	560	513	529	545	561	514	530	546	562	515	531	547	563	516	532	548	564	517
576	592	608	624	577	593	609	625	578	594	610	626	579	595	611	627	580	596	612	628	581
640	656	672	688	641	657	673	689	342	658	674	690	643	659	675	691	644	660	676	692	645
704	720	736	752	705	721	737	753	706	722	738	754	707	723	739	755	708	724	740	756	709
768	784	800	816	769	785	801	817	770	786	802	818	771	787	803	819	772	788	804	820	773
832	848	864	880	833	849	865	881	834	850	866	882	835	851	867	883	836	852	868	884	837
896	912	928	944	897	913	929	945	898	914	930	946	899	915	931	947	900	916	932	948	901
960	976	992	1008	961	977	993	1009	962	978	994	1010	963	979	995	1011	964	980	996	1012	965

Table 20-6. (Contd.)Re-Layout of 64 x 16 int8 B-Matrix

21	37	53	6	22	38	54	7	23	39	55	8	24	40	56	9	25	41	57	10	26	42	58	11	27
85	101	117	70	86	102	118	71	87	103	119	72	88	104	120	73	89	105	121	74	90	106	122	75	91
149	165	181	134	150	166	182	135	151	167	183	136	152	168	184	137	153	169	185	138	154	170	186	139	155
213	229	245	198	150	230	246	199	215	231	249	200	296	232	248	201	217	233	249	202	218	234	250	203	219
277	293	309	262	278	294	310	263	279	295	311	264	360	296	312	265	281	297	313	266	282	298	314	267	283
341	357	373	326	342	358	374	327	343	359	375	328	424	260	376	329	345	361	377	330	346	362	378	331	347
405	421	437	390	406	422	438	391	407	423	439	392	488	424	440	393	409	425	441	394	410	426	442	395	411
469	485	501	454	470	486	502	455	571	487	503	456	552	488	504	457	473	489	505	458	474	490	506	459	475
533	549	656	518	534	550	566	519	535	551	567	520	616	552	568	521	537	553	569	522	538	554	570	523	539
597	613	629	582	598	614	630	583	599	615	631	584	600	616	632	585	601	617	633	586	602	618	638	587	603
661	677	693	646	662	678	694	647	663	679	695	648	664	680	696	649	665	681	697	650	666	682	698	651	667
725	741	757	710	726	742	758	711	727	743	759	712	728	744	760	713	729	745	761	714	730	746	762	715	731
789	805	821	774	790	806	822	775	791	807	823	776	792	808	824	777	793	809	825	778	794	810	826	779	795
853	869	885	838	854	870	886	839	855	871	887	840	856	872	888	841	857	873	889	842	858	874	890	843	859
917	933	949	902	918	934	950	903	919	935	951	904	920	936	952	921	921	937	953	906	922	938	954	907	923
981	997	1013	966	982	998	1014	967	983	999	1015	968	984	1000	1016	969	985	1001	1017	970	986	1002	1018	971	987

Table 20-6. (Contd.)Re-Layout of 64 x 16 int8 B-Matrix

43	59	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63
107	123	76	92	108	124	77	93	109	125	78	94	110	126	79	95	111	127
171	187	140	156	172	188	141	157	173	189	142	158	174	190	143	159	175	191
235	251	204	220	236	252	205	221	237	253	206	222	238	254	207	223	239	255
299	315	268	284	300	316	269	285	301	317	270	286	302	318	271	287	303	319
363	379	332	348	364	380	333	349	365	381	334	350	366	382	335	351	367	383
427	443	396	412	428	444	397	413	429	445	398	414	430	446	399	415	431	447
491	507	560	476	492	508	461	477	493	509	462	478	494	510	463	479	495	511
555	571	524	540	556	572	525	541	557	573	526	542	558	574	527	543	559	575
619	635	588	604	620	636	589	504	621	637	590	606	622	638	591	607	623	639
683	699	652	668	684	700	653	669	685	701	654	670	686	702	655	671	687	703
747	763	716	732	748	764	717	733	749	765	718	734	750	766	719	735	751	767
811	827	780	796	812	828	781	797	813	829	782	798	814	830	783	799	815	831
875	891	844	860	876	892	845	861	877	893	846	862	878	894	847	863	879	895
939	955	908	924	940	956	909	925	941	957	910	926	942	958	911	927	943	959
1003	1019	972	988	1004	1020	973	989	1005	1021	974	990	1006	1022	975	991	1007	1023

20.5.4 STRAIGHTFORWARD GEMM IMPLEMENTATION

This is GEMM reference code. Its performance is sub-optimal. Please refer to Section 20.5.5.3 for optimal GEMM code. Begin implementation by defining the following:

Example 20-3. Common Defines

```
#define M ...                          // Number of rows in the A or C matrices
#define K ...                          // Number of columns in the A or rows in the B matrices
#define N ...                          // Number of columns in the B or C matrices
#define M_ACC ...                      // Number of C accumulators spanning the M dimension
#define N_ACC ...                      // Number of C accumulators spanning the N dimension
#define TILE_M ...                     // Number of rows in an A or C tile
#define TILE_K ...                     // Number of columns in an A tile or rows in a B tile
#define TILE_N ...                     // Number of columns in a B or C tile

typedef ... type_t;                    // The type of data being operated on
typedef ... res_type_t;                // The data type of the result

#define KPACK (4/sizeof(type_t))       // Vertical K packing into Dword

type_t A_mem[M][K];                    // A matrix
type_t B_mem[K/KPACK][N][KPACK];       // B matrix
res_type_t C_mem[M][N];                // C matrix

template<size_t rows, size_t bytes_cols> class tile;
template<class T> void tilezero (T& t);
template<class T> void tileload (T& t, void* src, size_t stride);
template<class T> void tilestore(T& t, void* dst, size_t stride);
template <class TC, class TA, class TB> void tdp(TC &tC, TA &tA, TB &tB) {
    int32_t v;
    for (size_t m = 0; m < TILE_M; m++) {
        for (size_t k = 0; k < TILE_K / KPACK; k++) {
            for (size_t n = 0; n < TILE_N; n++) {
                memcpy(&v, &tC.v[m][n * 4], sizeof(v));
                v += tA.v[m][k * 4] * tB.v[k][n * 4];
                v += tA.v[m][k * 4 + 1] * tB.v[k][n * 4 + 1];
                v += tA.v[m][k * 4 + 2] * tB.v[k][n * 4 + 2];
                v += tA.v[m][k * 4 + 3] * tB.v[k][n * 4 + 3];
                memcpy(&tC.v[m][n * 4], &v, sizeof(v));
            }
        }
    }
}
```

Data type_t is the type being operated upon, i.e., signed/unsigned int8 or bfloat16. For the description of KPACK, see Section 20.5.5. The tile template class and the three functions that operate on it are the same as the ones introduced in Example 20-3. tilezero (t) resets the contents of tile t to 0, tileload(t, src, stride) and loads tile t with the contents of data at src with a stride of stride between consecutive rows. tilestore(t, dst, stride) stores the contents of tile t to dst with a stride of stride between consecutive rows. Additionally, tdp(tC,tA,tB) performs a matrix multiplication equivalent of tC=tC+tA×tB. In reality, tiles are defined by known compile-time integers, and the actual code operating on tiles looks slightly different. Please visit the GitHub Repository for proper usage.

The following is a simple implementation of GEMM of the matrices stored in A_mem and B_mem.

Example 20-4. Reference GEMM Implementation

```
for (int n = 0; n < N; n += N_ACC*TILE_N) {
  for (int m = 0; m < M; m += M_ACC*TILE_M) {
    tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
      for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
        tilezero(tC[m_acc][n_acc]);

    for (int k = 0; k < K; k += TILE_K) {
      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
        tile<TILE_K/KPACK, TILE_N*KPACK> tB;
        tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
        for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
          tile<TILE_M, TILE_K*sizeof(type_t)> tA;
          tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
          tdp(tC[m_acc][n_acc], tA, tB);
        }
      }
    }
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
      for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
        int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
        tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
      }
    }
  }
}
```

This implementation is the reference point in the following discussions.

20.5.5 OPTIMIZATIONS

20.5.5.1 Minimizing Tile Loads

Redundant tile loads may severely impact performance due to the large size of the data loaded into the tiles, unnecessary cache evictions, etc. To minimize tile loads, it is essential to utilize the data as completely as possible once it has been loaded into the tile.

Location of the K Loop: Outside of the M_ACC and N_ACC Loops

The three loops in lines 8–18 of Example 20-4 could also have been written this way:

Example 20-5. K-Dimension Loop as Innermost Loop–A, a Highly Inefficient Approach

```
for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
 tile<TILE_K/KPACK, TILE_N*KPACK> tB;
 for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
  tile<TILE_M, TILE_K*sizeof(type_t)> tA;
  for (int k = 0; k < K; k += TILE_K) {
   tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
   tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
   tdp(tC[m_acc][n_acc], tA, tB);
  }
 }
}
```

While both approaches yield correct results, there are K/TILE_K×N_ACC B tile loads in the reference implementation. Additionally, K/TILE_K×N_ACC×M_ACC B tile loads in the implementation presented in this section. The number of A tile loads is identical.

This approach is also characterized by excessive pressure on the memory along with an increased number of tile loads.

Suppose the B_mem data resides in main memory. In the reference implementation, a new chunk of TILE_K×TILE_N B data is read every M_ACC iteration of the inner loop. The inner loop then reuses the read data. In the current implementation, when n_acc == m_acc == 0, a new chunk of TILE_K×TILE_N B data is read every iteration of the inner loop. Then the same data is read (presumably from caches) on subsequent iterations of n_acc, m_acc. This burst access pattern of reads from main memory results in increased data latency and decreased performance.

Hence, keeping the K-dimension loop outside the M_ACC and N_ACC loops is recommended.

Pre-Loading Innermost Loop Tiles

Consider the following replacement code for the code in lines 8–18 of Example 20-4:

Example 20-6. Innermost Loop Tile Pre-Loading

```
1    for (int k = 0; k < K; k += TILE_K) {
2     tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
3     for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
4      tileload(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
5     for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
6      tile<TILE_K/KPACK, TILE_N*KPACK> tB;
7      tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
8      for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
9       tdp(tC[m_acc][n_acc], tA[m_acc], tB);
10     }
11    }
12   }
```

The A-tile has been extended to an array of A-tiles (line 2) and pre-read the A tiles for the current K-loop iteration (lines 3–4). A pre-read A-tile is used in the tile multiplication (line 9). There were

K/TILE_K×N_ACC×M_ACC A-tile reads in the reference implementation, while there are only K/TILE_K×M_ACC A-tile reads in the current implementation.

Hence, preallocation and pre-reading the tiles of the innermost loop (tA[M_ACC] in this case) is recommended. The maximum number of tiles used at any given time in this scenario is N_ACC×M_ACC+M_ACC+1 as opposed to N_ACC×M_ACC+2 in the reference implementation. Since this optimization requires preallocation of an additional M_ACC-1 tiles, and since tiles are a scarce resource, if N_ACC<M_ACC, it might prove beneficial to switch the order of the N_ACC and M_ACC loops. This way, it is possible to allocate N_ACC-1<M_ACC-1 additional tiles:

Example 20-7. Switched Order of M_ACC and N_ACC Loops

```
for (int k = 0; k < K; k += TILE_K) {
  tile<TILE_K/KPACK, TILE_N*KPACK> tB[N_ACC];
  for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
    tileload(tB[n_acc], B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
    tile<TILE_M, TILE_K*sizeof(type_t)> tA;
    tileload(tA, &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
    for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
      tdp(tC[m_acc][n_acc], tA, tB[n_acc]);
    }
  }
}
```

2D Accumulator Array vs. 1D Accumulator Array

Consider Example 20-6 with the following scenarios:

- `N_ACC=2,M_ACC=2`
- `N_ACC=4,M_ACC=1`

As stated before, the number of A tile loads in lines 3–11 is M_ACC, and the number of B tile loads is N_ACC. Thus, the total number of tile loads (M_ACC+N_ACC) is 4 in the first scenario vs. 5 in the second one (an increase of 25%), even though both scenarios perform the same amount of work.

Hence, using 2D accumulator arrays is recommended. Selecting dimensions close to square is particularly recommended (since x=y minimizes f(x,y)=x+y under the constraint x×y=const).

20.5.5.2 Software Pipelining of Tile Loads and Stores

It is a best practice to interleave instructions using different resources so they may be executed in parallel, preventing a bottleneck involving a specific resource. Therefore, preventing sequential TileLoads and TileStores (see lines 19–23 of Example 20-4 and lines 3–4 of Example 20-6) is recommended. Instead, interleave them with the tdp instructions (see Example 20-8).

20.5.5.3 Optimized GEMM Implementation

Below is the original code from Example 20-4, augmented with the insights from Example 20-6, with tile loads and stores interleaved with tdps:

Example 20-8. Optimized GEMM Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2  for (int m = 0; m < M; m += M_ACC*TILE_M) {
3    tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4    tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5    tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7    for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8     for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9      tilezero(tC[m_acc][n_acc]);
10
11    for (int k = 0; k < K; k += TILE_K) {
12     for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
13      tileload(tB, B_mem[k/KPACK][n + n_acc*TILE_N], N*sizeof(type_t)*KPACK);
14      for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
15       if (n_acc == 0)
16        tileload(tA[m_acc], &A_mem[m + m_acc*TILE_M][k], K*sizeof(type_t));
17       tdp(tC[m_acc][n_acc], tA[m_acc], tB);
18       if (k == K - TILE_K) {
19        int mc = m + m_acc*TILE_M, nc = n + n_acc*TILE_N;
20        tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
21       }
22      }
23     }
24    }
25  }
26}
```

While placing the tile loads and stores under conditions inside the main loop (lines 13, 16, 20), conditions can be eliminated by sufficiently unrolling the loops.

The rest of this section presents a specific example of GEMM, implemented in low-level Intel AMX instructions. This is to show a full performance potential from using Intel AMX extensions.

Example 20-9. Dimension of Matrices, Data Types, and Tile Sizes

```
#define M 32
#define K 128
#define N 32
#define M_ACC 2
#define N_ACC 2
#define TILE_M 16
#define TILE_K 64
#define TILE_N 64

typedef int8_t type_t
typedef int32_t res_type_t
```

The following code is a specific example of the algorithm outlined in Example 20-8.

Example 20-10. Optimized GEMM Assembly Language Implementation

```
/*1 of 2*/
typedef struct {
        uint8_t palette_id;
        uint8_t startRow;
        uint8_t reserved[14];
        uint16_t cols[16];
        uint8_t rows[16];
} __attribute__ ((__packed__)) tileconfig_t;

static const tileconfig_t tc = {
        1,                                                      // palette_id
        0,                                                      // startRow
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,               // reserved - must be
        64, 64, 64, 64, 64, 64, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // calls for 7 tiles used
       16, 16, 16, 16, 16, 16, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0    // rows for 7 tiles used
};


_asm {
ldtilecfg tc                            # Load tile config
mov r8, A_mem                           # Initialize register for A
mov r9, B_mem                           # Initialize register for B
mov r10, C_mem                          # Initialize register for C

mov r11, 128                            #  Initialize register for strides
tileloadd tmm6, [r9 + r11*1]            #  Load B for n_acc = 0, k_acc = 0
tileloadd tmm4, [r8 + r11*1]            #  Load A for m_acc = 0, k_acc = 0
tilezero tmm0                           #  Zero accumulator tile
tdpbssd tmm0, tmm4, tmm6                #  Multiply-add tmm0 += tmm4 * tmm6
tileloadd tmm5, [r8 + r11*1 + 2048]     #  Load A for m_acc = 1, k_acc = 0
tilezero tmm1                           #  Zero accumulator tile
/*2 of 2*/
31   tdpbssd tmm1, tmm5, tmm6                   # Multiply-add tmm1 += tmm5 * tmm6
32   tileloadd tmm6, [r9 + r11*1 + 64 ]         # Load B for n_acc = 1, k_acc = 0
33   tilezero tmm2                              # Zero accumulator tile
34   tdpbssd tmm2, tmm4, tmm6                   # Multiply-add tmm2 += tmm4 * tmm6
35   tilezero tmm3                              # Zero accumulator tile
36   tdpbssd tmm3, tmm5, tmm6                   # Multiply-add tmm3 += tmm5 * tmm6
37   tileloadd tmm6, [r9 + r11*1 + 2048]        # Load B for n_acc = 0, k_acc = 1
38   tileloadd tmm4, [r8 + r11*1 + 64]          # Load A for m_acc = 0, k_acc = 1
39   tdpbssd tmm0, tmm4, tmm6                   # Multiply-add tmm0 += tmm4 * tmm6
40   tilestored [r10 + r11*1], tmm0             # Store C for m_acc = 0, n_acc = 0
41   tileloadd tmm5, [r8 + r11*1 + 2112]        # Load A for m_acc = 1, k_acc = 1
42   tdpbssd tmm1, tmm5, tmm6                   # Multiply-add tmm1 += tmm5 * tmm6
43   tilestored [r10 + r11*1 + 2048], tmm1      # Store C for m_acc = 1, n_acc = 0
44   tileloadd tmm6, [r9 + r11*1 + 2112]        # Load B for n_acc = 1, k_acc = 1
45   tdpbssd tmm2, tmm4, tmm6                   # Multiply-add tmm2 += tmm4 * tmm6
46   tilestored [r10 + r11*1 + 64], tmm2        # Store C for m_acc = 0, n_acc = 1
47   tdpbssd tmm3, tmm5, tmm6                   # Multiply-add tmm3 += tmm5 * tmm6
48   tilestored [r10 + r11*1 + 2112], tmm3      # Store C for m_acc = 1, n_acc = 1
49   }
```

Lines 1-12 in Example 20-10 define the tile configuration for this example, and contain information about tile sizes. Tile configuration should be loaded prior to any execution of Intel AMX instructions (line 16). Tile sizes are defined by the configuration at the load time and can't be changed dynamically (unless TileRelease is called). The 'palette_id' field in the configuration specifies the number of logical tiles available for use; palette_id == 1 means 8 logical tiles are available, named tmm0 through tmm7. This particular example uses 7 logical tiles (tmm4, tmm5 for A, tmm6 for B, tmm0-tmm3 for C).

According to the dimensions specified, K-loop consists of 2 iterations (cf. code listing 8.1, line 11) according to the dimensions specified in the example. Lines 23-34 implement the first iteration and lines 35-46 the second iteration. Note the interleaving of tdp and TileStore instructions to hide the high cost of TileStore operation.

Variable Input Dimensions

The code in Example 20-8 and 20-10 process an entire matrix of inputs of size MxK. Sometimes, only part of the input is significant, so it is beneficial to adapt the computation to the actual input size. Often, topologies that use self-attention it is enough to process only the first m rows of the input that are significant, where m < M. For example, taking the GEMM dimensions described above with the choice of a 1D accumulator array of N_ACC=2,M_ACC=1, when accepting data as input with at most sixteen significant rows, we can degenerate the m loop (line 2 in Example 20-8) so as to effectively reduce the computation by half.

It is worth noting that in variable M dimension use cases there is an advantage to 1D accumulators. Up to N_ACC=6, M_ACC=1 dimensions are possible if N is 96 or larger, one tile for A, one tile for B and six tiles for the accumulator.

20.5.5.4 Direct Convolution with Intel® AMX

Direct convolution is performed directly on the input data; no data replication is required. However, there are some layout considerations.

Activations Layout

Similar to the Intel DL Boost use case, the activations are laid out in a layout obtained from the original layout by the following procedure:

Example 20-11. Activations Layout Procedure

```
#define K C                              // K-dimension of the A matrix = channels
#define M H*W                            // M-dimension of the A matrix = spatial
type_t A_mem_orig[C][H][W];              // Original activations tensor
type_t A_mem[H][W][K];                   // Re-laid A matrix7

for (int c = 0; c < C; ++c)
  for (int h = 0; h < H; ++h)
    for (int w = 0; w < W; ++w)
      A_mem[h][w][c] = A_mem_orig[c][h][w];
```

This procedure on the left side of the diagram below shows the conversion of a 3-dimensional tensor into a 2-dimensional matrix:

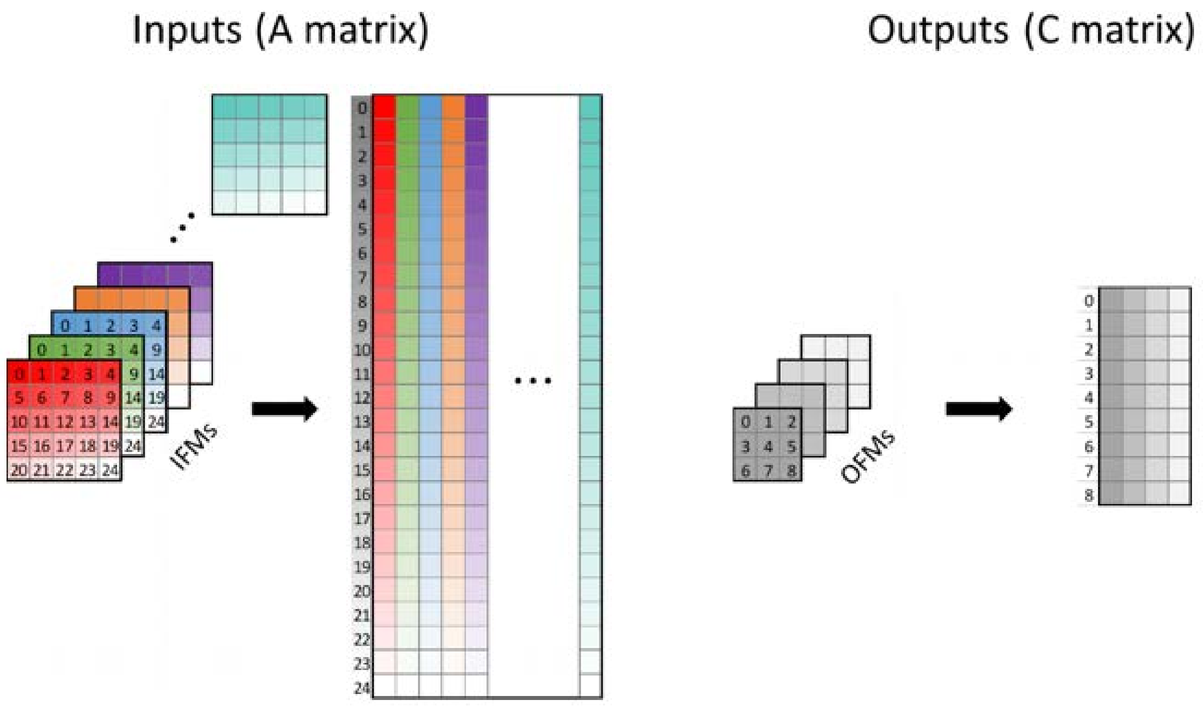


Figure 20-3. Activations layout

The procedure shown on the right is identical for the outputs, e.g., the activations of the next layer in the topology).

Weights Layout

Similar to the Intel DL Boost use case, the weights are re-laid by the following procedure:

Example 20-12. Weights Re-Layout Procedure

```
#define KH ...                                  // Vertical dimension of the weights
#define KW ...                                  // Horizontal dimension of the weights
#define KPACK (4/sizeof(type_t))                // Vertical K packing into Dword

type_t B_mem_orig[K][N][KH][KW];                // Original weights
type_t B_mem[KH][KW][K/KPACK][N][KPACK];        // Re-laid B matrices

for (int kh = 0; kh < KH; ++kh)
  for (int kw = 0; kw < KW; ++kw)
    for (int k = 0; k < K; ++k)
      for (int n = 0; n < N; ++n)
        B_mem[kh][kw][k/KPACK][n][k%KPACK] = B_mem_orig[k][n][kh][kw];
```

The procedure transforms the original 4-dimensional tensor into a series of 2-dimensional matrices (a single matrix is highlighted in orange in Example 20-12) as illustrated in the following diagram for KH=KW=3, resulting in a series of 9 B-matrices:

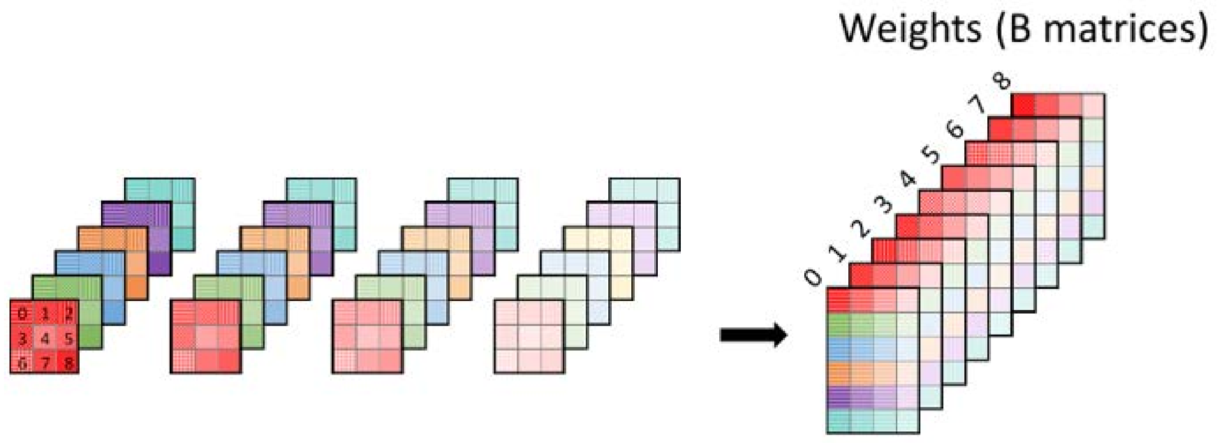


Figure 20-4. Weights Re-Layout

20.5.5.5 Convolution - Matrix-like Multiplications and Summations Equivalence

Figure 20-5 illustrates the equivalence between convolution and summation of a series of matrix-like multiplications between subsets of the 2-dimensional A-matrix representing the 3-dimensional activations tensor. The 2-dimensional B-matrices correspond to the various spatial elements of the weights filter.

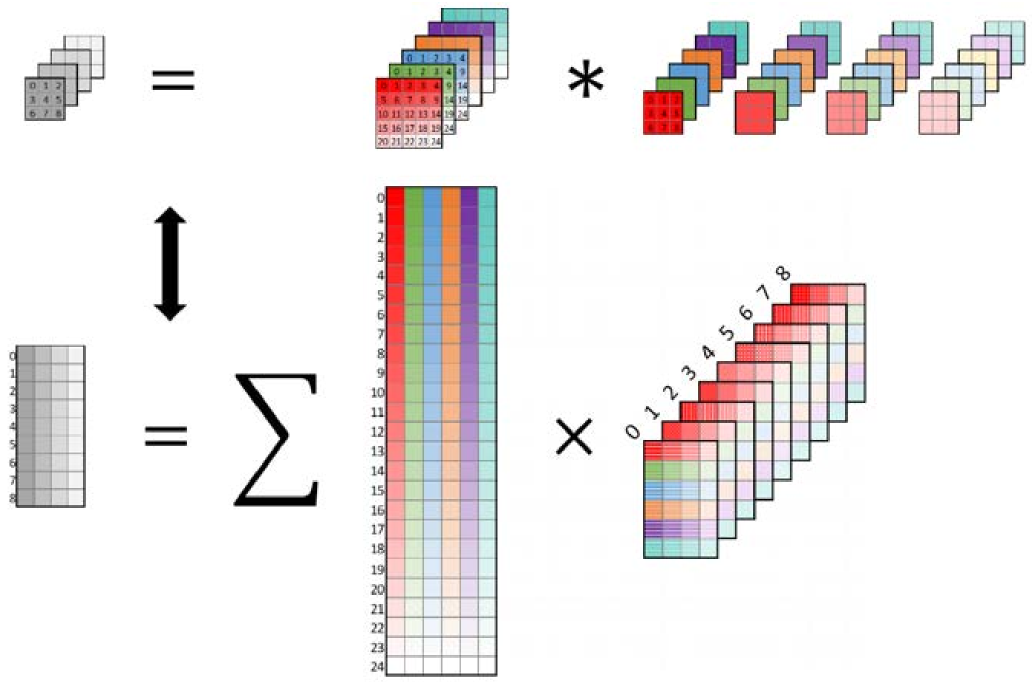


Figure 20-5. Convolution-Matrix Multiplication and Summation Equivalence

The A-matrix subset participating in the matrix-like multiplication depends on the spatial weight element in question (i.e., the kh,kw coordinates, or the index in the range 0–8 in the previous example). For each weight element, the A-matrix’s participating rows will interact with the weight element when the filter is slid over the activations. For example, when sliding the filter over the activations in the previous example, weight element 0 will only interact with activation elements 0, 1, 2, 5, 6, 7, 10, 11, and 12. For example, it will not interact with activation element four because when the filter is applied in such a manner (i.e., weight element 0 interacts with activation element 4), weight elements 2, 5, and 8 leave the activation frame entirely. The A-matrix subsets for several weight elements are illustrated in the following figure.

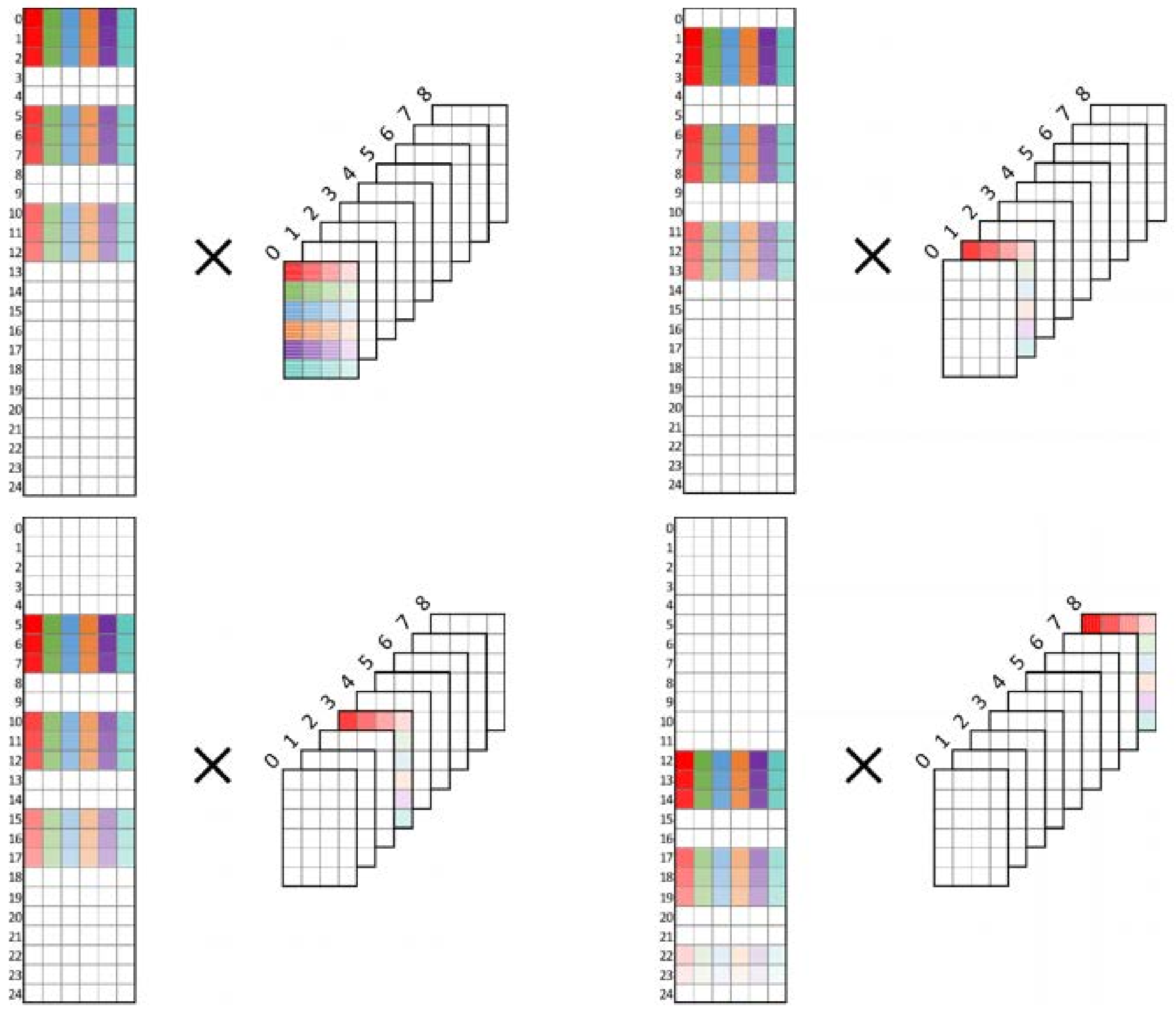


Figure 20-6. Matrix-Like Multiplications Part of a Convolution

20.5.5.6 Optimized Convolution Implementation

Replace the common defines in Example 20-3 with the following:

Example 20-13. Common Defines for Convolution

```
#define H ...                          // The height of the activation frame
#define W ...                          // The width of the activation frame
#define MA (H*W)                       // The M dimension (rows) of the A matrix
#define K ...                          // Number of activation channels
#define N ...                          // Number of output channels
#define KH ...                         // The height of the weights kernel
#define KW ...                         // The width of the weights kernel
#define SH ...                         // The vertical stride of the convolution
#define SW ...                         // The horizontal stride of the convolution
#define M_ACC ...                      // Number of C accumulators spanning the M dimension
#define N_ACC ...                      // Number of C accumulators spanning the N dimension
#define TILE_M ...                     // Number of rows in an A or C tile
#define TILE_K ...                     // Number of columns in an A tile or rows in a B tile
#define TILE_N ...                     // Number of columns in a B or C tile

#define HC ((H-KH)/SH+1)               // The height of the output frame
#define WC ((W-KW)/SW+1)               // The width of the output frame
#define MC (HC*WC)                     // The M dimension (rows) of the C matrix

typedef ... type_t;                    // The type of the data being operated on
typedef... res_type_t;                 // The data type of the result

#define KPACK (4/sizeof(type_t))                      // Vertical K packing into Dword

type_t A_mem[H][W][K];                                // A matrix (equivalent to A_mem[H*W][K])
type_t B_mem[KH][KW][K/KPACK][N][KPACK];              // B matrices
res_type_t C_mem[MC][N];                              // C matrix

template<size_t rows, size_t cols> class tile;

template<class T> void tilezero (T& t);
template<class T> void tileload (T& t, void* src, size_t stride);
template<class T> void tilestore(T& t, void* dst, size_t stride);
template<class TC, class TA, class TB> void tdp(TC& tC, TA& tA, TB& tB);

int mc_to_ha(int mc) {return mc / HC * SH;}        // C matrix M -> A tensor h coord
int mc_to_wa(int mc) {return mc % HC * SW;}        // C matrix M -> A tensor w coord
```

Replace the implementation in Example 20-8 with the following:

Example 20-14. Optimized Direct Convolution Implementation

```
1 for (int n = 0; n < N; n += N_ACC*TILE_N) {
2  for (int m = 0; m < MC; m += M_ACC*TILE_M) {
3   tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
4   tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
5   tile<TILE_K/KPACK, TILE_N*KPACK> tB;
6
7   for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
8    for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
9     tilezero(tC[m_acc][n_acc]);
10
11   for (int k = 0; k < K; k += TILE_K) {
12    for (int kh = 0; kh < KH; ++kh) {
13     for (int kw = 0; kw < KW; ++kw) {
14      for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
15       int nc = n + n_acc*TILE_N;
16       tileload(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
17       for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
18        int mc = m + m_acc*TILE_M;
19        if (n_acc == 0) {
20         int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
21         tileload(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
22        }
23        tdp(tC[m_acc][n_acc], tA[m_acc], tB);
24        if (k + kh + kw == K - TILE_K + KH + KW - 2)
25         tilestore(tC[m_acc][n_acc], &C_mem[mc][nc], N*sizeof(res_type_t));
26       }
27      }
28     }
29    }
30   }
31  }
32 }
```

The divergences highlighted in yellow in Example 20-8 include:

- The loop over M-dimension (line 2) references the M-dimension of the C-matrix (since the M-dimensions of A and C no longer have to be the same). To get the corresponding A-matrix m index from a C-matrix m index, one must employ the conversion functions mc_to_ha() and mc_to_wa() (line 20).
- There are additional loops over the weights kernel dimensions KH and KW (lines 12–13), which define the B-matrix to be used (line 16), enter into the condition for accumulator tile storing (line 24) and computation of A-matrix coordinates (line 20).
- The stride of the A tile load must account for the convolutional horizontal stride (line 21).

Note that care should be taken to define TILE_M*M_ACC in such a way that it cleanly divides WC (the width of the output frame), i.e., WC%(TILE_M*M_ACC)==0. Otherwise, some tiles will end up loading data that should not be multiplied by the corresponding weight element (see Figure 20-6). Possible mitigations of this issue:

- An M_ACC loop with a dynamic upper limit depending on the current position in A.

- Use different sized A tiles (and correspondingly C tiles) depending on the current position in A (if there are enough free tiles, performing TileConfig during the convolution is highly discouraged).
- Define TILE_M without consideration for WC and remove/disregard the "junk" data from the results at the post-processing stage (code not shown). Care should be taken in this case concerning the advancement of the m index (line 2) since the current assumption is that every row of every tile is valid (corresponds to a row in the C matrix). If "junk" data is loaded, this is no longer the case: a C-tile will have less than TILE_M rows of C.

Location of the KH, KW Loops

As shown in Example 20-5, it is ill-advised to put the loop over the K-dimension inside an inner M_ACC or N_ACC loop. The same considerations hold in the case of the kh,kw loops. While there is no functional obstacle precluding the positioning of the kh,kw loops further up (before lines 12-13), it is recommended to keep them under the K loop and above the M_ACC, N_ACC loops because, during the traversal of kh,kw with the same k value, the TileLoad of A-data (line 21) will have much overlap with A-data loaded for previous values of kh,kw (with the same k value). This data will likely reside in the lowest-level cache. Moving the kh,kw loops upward will reduce that likelihood.

20.6 CACHE BLOCKING

Data movement costs vary greatly depending on where the data lies in the cache hierarchy. When the matrices involved in a GEMM or convolution are larger than the available cache, computations must proceed in such a manner as to optimize data reuse from the cache. Here a simple cache-blocking scheme is implemented to simultaneously process partial blocks of the A, B, and C matrices.

20.6.1 OPTIMIZED CONVOLUTION IMPLEMENTATION WITH CACHE BLOCKING

In the following example, the focus is on implementing cache blocking for the optimized convolution implementation described in the Optimized Convolution Implementation <XREF> section. However, note that similar changes can also be made to the optimized GEMM implementation. Alternatively, the GEMM implementation can be derived as a special case of convolution with KH=KW=1 and SH=SW=1.

In addition to the common defines in Example 20-13, add the following:

Example 20-15. Additional Defines for Convolution with Cache Blocking

```
#define MC_CACHE ...                          // Extent of cache block along the M dimension of the C matrix
#define K_CACHE ...                           // Extent of cache block along the K dimension
#define N_CACHE ...                           // Extent of cache block along the N dimension
typedef ... acc_type_t;                       // The accumulation data type (either int32 or float)
acc_type_t aC_mem[M_ACC][N_ACC][TILE_M][TILE_N];                 // Accumulator buffers of C
```

Replace the implementation in Example 20-14 with the following:

Example 20-16. Optimized Convolution Implementation with Cache Blocking

```
for (int nb = 0; nb < N; nb += N_CACHE) {
  for (int mb = 0; mb < MC; mb += MC_CACHE) {
    for (int kb = 0; kb < K; kb += K_CACHE) {
      for (int n = nb; n < nb + N_CACHE; n += N_ACC*TILE_N) {
        for (int m = mb; m < mb + MC_CACHE; m += M_ACC*TILE_M) {
          tile<TILE_M, TILE_N*sizeof(res_type_t)> tC[M_ACC][N_ACC];
          tile<TILE_M, TILE_K*sizeof(type_t)> tA[M_ACC];
          tile<TILE_K/KPACK, TILE_N*KPACK> tB;

          for (int n_acc = 0; n_acc < N_ACC; ++n_acc)
            for (int m_acc = 0; m_acc < M_ACC; ++m_acc)
              if (kb == 0)
                tilezero(tC[m_acc][n_acc]);
              else {
                int m_aC = (m - mb) / TILE_M + m_acc;
                int n_aC = (n - nb) / TILE_N + n_acc;
                tileload(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
                       TILE_N*sizeof(acc_type_t));
              }

          for (int k = kb; k < kb + K_CACHE; k += TILE_K) {
            for (int kh = 0; kh < KH; ++kh) {
              for (int kw = 0; kw < KW; ++kw) {
                for (int n_acc = 0; n_acc < N_ACC; ++n_acc) {
                  int nc = n + n_acc*TILE_N;
                  tileload(tB, B_mem[kh][kw][k/KPACK][nc], N*sizeof(type_t)*KPACK);
                  for (int m_acc = 0; m_acc < M_ACC; ++m_acc) {
                    int mc = m + m_acc*TILE_M;
                    if (n_acc == 0) {
                      int ha = mc_to_ha(mc)+kh, wa = mc_to_wa(mc)+kw;
                      tileload(tA[m_acc], &A_mem[ha][wa][k], K*SW*sizeof(type_t));
                    }
                    tdp(tC[m_acc][n_acc], tA[m_acc], tB);
                    if (k + kh + kw == K - TILE_K + KH + KW - 2)
                      tilestore(tC[m_acc][n_acc], &C_mem[mc][nc],
                            N*sizeof(res_type_t));
                    else if (k + kh + kw == kb + K_CACHE - TILE_K + KH + KW - 2) {
                      int m_aC = (m - mb) / TILE_M + m_acc;
                      int n_aC = (n - nb) / TILE_N + n_acc;
                      tilestore(tC[m_acc][n_acc], &aC_mem[m_aC][n_aC],
                            TILE_N*sizeof(acc_type_t));
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The loops over the N, MC, and K dimensions are replaced by loops over cache blocks of N, MC, and K.

Additional loops over the entire N, MC, and K-dimensions are added at the outermost level. These loops have a step size equal to the cache blocks of N, MC, and K.

In the case of cache blocking along the K-dimension, additional calls to `TileLoad` and TileStore are required to load and store intermediate accumulation results. Note that this adds additional memory traffic, especially for int8 output data types (as Accumulation data type is either int32_t or float). For this reason, it is generally not advisable to block along the K dimension.

For simplicity, assume the following relationships:

- N is an integer multiple of N_CACHE: an integer multiple of N_ACC*TILE_N.
- MC is an integer multiple of MC_CACHE: an integer multiple of M_ACC*TILE_M. As before, the condition WC%(TILE_M*M_ACC)==0 still holds.
- K is an integer multiple of K_CACHE: an integer multiple of TILE_K.

Define the following set of operations as the compute kernel of the optimized convolution implementation. First, initialize the accumulation tiles to zero (line 13) for an M_ACC*TILE_M x N_ACC*TILE_N chunk of the C-matrix. Next, for each of the KH*KW B-matrices, the matrix multiplication of the corresponding M_ACC*TILE_M x K chunk of the A-matrix by a K x N_ACC*TILE_N chunk of the B-matrix is performed, each time accumulating to the same set of accumulation tiles (lines 18–30). Finally, the results are stored in the C-matrix (line 32).

Continue with the computation **of a full cache block** of C-matrix, ignoring any blocking along the K-dimension. First, the kernel is performed for the first chunks of the A, B, and C cache blocks. Next, the chunks of A and C advance along the M dimension, and the kernel is repeated with the same chunk set of the B-matrices. The above step is repeated until the last chunks of A and C in the current cache block have been accessed. Next, the chunks of B and C are advanced along the N-dimension by N_ACC*TILE_N and the chunk of A returns to the beginning of its cache block.

Observe the following from the above description of the computation of a **full cach**e block of the C-matrix:

- For each kernel iteration, it is better if the current chunk of matrix A (roughly KH*M_ACC*TILE_M*K*sizeof(type_t)) fits into the DCU. This allows for maximal data reuse between the partially overlapping regions of A that need to be accessed by the different B matrices.
- Advancing from one chunk of matrix A to the next, it is better if the current chunk set of the B matrices (in total, KH*KW*K*N_ACC*TILE_N*sizeof(type_t)) fits into the DCU.
- Advancing from one chunk set of the B matrices to the next, it is better if the current cache block of matrix A fits into the MLC.
- Advancing from one cache block of matrix A to the next, it is better if the current cache block of the B matrices (in total, KH*KW*K*N_CACHE*sizeof(type_t)) fits into the MLC.

From these observations, a general cache blocking strategy is choosing `MC_CACHE` and `N_CACHE` to be as large as possible while keeping the A, B, and C cache blocks in the MLC.

Intel® AMX-Specific Considerations

A specific feature of Intel AMX-accelerated kernels to keep in mind when applying the previous cache-blocking recommendations is any post-processing of results from the Intel AMX unit (e.g., adding bias, dequantizing, converting between data types) must occur by way of vector registers. Thus, a buffer is needed to store results from the accumulation tiles and load them into vector registers for post-processing. Note that if acc_type_t is the same as res_type_t, the C matrix itself can be used to store intermediate results. However, the buffer is small (at most 4KB for the accumulation strategies described in “2D Accumulator Array vs. 1D Accumulator Array”) and easily fits into the DCU. While it should still be considered when determining the optimal cache block partitioning, it is unlikely to influence kernel performance strongly.

20.7 MINI-BATCHING IN LARGE BATCH INFERENCE

Layers have different sizes and shapes, which require different cache and memory-blocking strategies. There are layers with a small spatial dimension (M) and relatively larger shared dimension (K) and SIMD dimension (N). In such layers, the weights are significantly larger than the inputs. Therefore, most of the load operations are weights matrix loads whose cost is high when the weights reside in memory or the last level cache.

Running a large batch allows employing an optimization that amortizes the cost of loading the weight matrix. The idea is to use the same weights for multiple inputs, e.g., execute the same layer with multiple images. This optimization is highly applicable in CNNs where the inputs of the first layers are large while the weights are relatively small but end with small input images and large weight matrices. Optimal execution of the topology starts in the instance or image affinity, where a single input goes through one layer after another before the next input is retrieved. At some point, the topology execution switches to layer affinity, where the same layer processes several inputs (mini-batch) before moving forward to the next layer.

For example, in ResNet-50, the conv-1 to conv-4 layers have relatively large IFMs and smaller weight matrices. However, many weight matrices are larger than MLC size (mid-level cache) in the conv-5 layers. The switchover point from image affinity to layer affinity on a 4th Generation Intel® Xeon® Processor microarchitecture is the first layer of conv-5.

The diagram below illustrates six layers with four instances per thread (mini-batch of four). Boxes with identical colors identify the same layers in each column. Arrows flowing downward through each column’s layers represent the data flow of a particular instance. Translucent red arrows identify the execution order of layers with corresponding instances. The first four layers of the diagram have instance (aka image) affinity, and the last two have layer affinity.

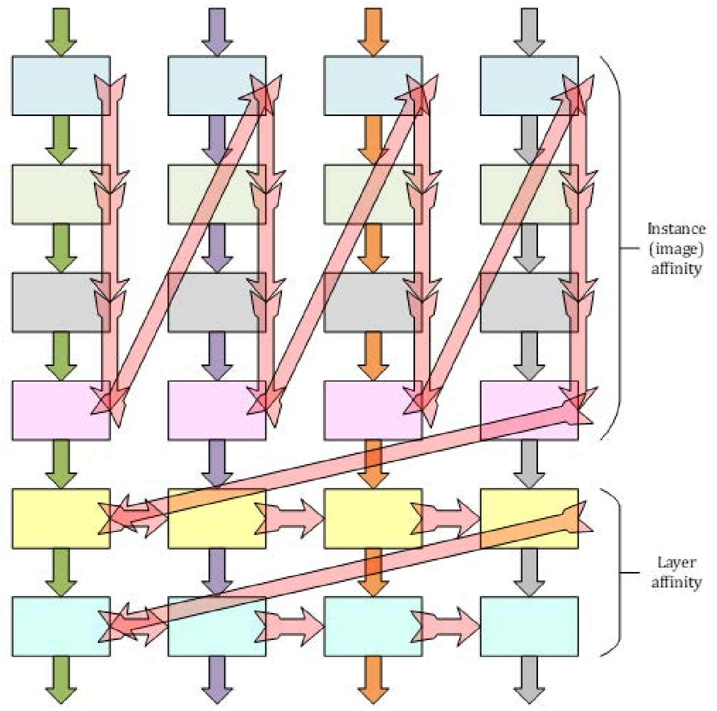


Figure 20-7. Batching Execution Using Six Layers with Four Instances Per Thread

On Resnet-50, this optimization can yield a 17% performance gain.

20.8 NON-TEMPORAL TILE LOADS

When a regular tile load is issued, the data for the tile are placed in L2, L1, and then in the tile register (DRAM/L3->L2->L1->tile register), as with any other register load. This has the well-known benefit of reduced data read latency due to data proximity when recently accessed data are reaccessed after a short time. However, indiscriminate application of this approach might sometimes prove detrimental.

Consider the code in Example 20-4, referring to the unoptimized, unblocked implementation for simplicity. The five loops in the code listing alongside the total input (A) matrix data and weights (B) matrix data accessed at each loop level is shown in the following table. The original row in the code listing is provided for convenience:

Table 20-7. Five Loops in Example 20-4

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	M×K	K×N
2	m	[0:M:M_ACC×TILE_M]	M×K	K×N_ACC×TILE_N
8	k	[0:K:TILE_K]	MC_CACHE×K	
9	n acc	[0:N_ACC:1]	M_ACC×TILE_M×TILE_K	TILE_K×N_ACC×TILE_N
12	m ac	[0:M_ACC:1]		

20.8.1 PRIORITY INVERSION SCENARIOS WITH TEMPORAL LOADS

For the following discussion, assume:

- The data type is int8 (i.e., each element in the table above takes 1 byte).
- TILE_M=16, TILE_K=64, TILE_N=16 (i.e., all tiles are of size 1kB).
- L1 cache size is 32kB.
- M_AC=N_ACC=2.

Scenario One:

Consider the following scenario, including M=256, K=1024, and N=256.

Table 20-8 illustrates accessed data sizes:

Table 20-8. Accessed Data Sizes: Scenario One

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	256kB	256kB
2	m	[0:M:M_ACC×TILE_M]		32kB
8	k	[0:K:TILE_K]	32kB	
9	n acc	[0:N_ACC:1]	32kB	2kB
12	m ac	[0:M_ACC:1]		

At the k loop level, the combined sizes of A and B accessed data will overflow the L1 cache by a factor of two. Proceeding to the m-level since m is progressing, new A-data are constantly read (a total of 256kB-32kB=224kB new A data), while the same 32kB of B data are being accessed repeatedly. Thus, a priority inversion occurs: new A-data placed in the L1 cache repeatedly are accessed only once. They evict the 32kB of B data that are accessed eight times. Placement of A data in the L1 cache is not beneficial: the next time the same data are accessed will be in the n loop after 256kB (x8 L1 cache size) of A data has been read. Additionally, it is detrimental because it causes repeated eviction of 32kB of B data that could have been read from the L1 cache eight times.

Scenario Two:

Consider the following scenario, including M=32, K=1024, and N=256. Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Here, the M-dimension is covered in the m_acc loop, and the loop over m is redundant. The priority inversion is: as n advances, new B-data (accessed only once) repeatedly evict 32kB of A-data that could have been read (8 times) from the L1 cache had it not been pushed out by B-data.

Table 20-9. Accessed Data Sizes: Scenario Two

Row	Var	Variable Range	A Data Size	B Data Size
1	n	[0:N:N_ACC×TILE_N]	32kB	256kB
2	m	[0:M:M_ACC×TILE_M]		32kB
8	k	[0:K:TILE_K]		
9	n acc	[0:N_ACC:1]	2kB	2kB
12	m ac	[0:M_ACC:1]		

These two basic scenarios can be readily extended to the blocked code in Example 20-16.

Table 20-10. Accessed Data Sizes Extended to Blocked Code

Row	Var	Variable Range	A Data Size	B Data Size
1	nb	[0:N:N_CACHE]	M×K	
2	mb	[0:MC:MC_CACHE]	M×K	
3	kb	[0:K:K_CACHE]	MC_CACHE×K	
4	n	[nb:nb+N_CACHE:N_ACC×TILE_N]	MC_CACHE×K_CACHE	K_CACHE×KH×KW×N_ACC×TILE_N
5	m	[mb:mb+MC_CACHE:M_ACC×TILE_M]		
18	k	[kb:kb+K_CACHE:TILE_K]		
19	kh	[0:KH:1]	/*/*	TILE_K×KH×KW×N_ACC×TILE_N
20	kw	[0:KW:1]	M_ACC×TILE_M×TILE_K	
21	n acc	[0:N_ACC:1]		TILE_K×N_ACC×TILE_N
24	m ac	[0:M_ACC:1]		

NOTE

Due to the nature of convolution, the loops over kh, kw reuse most of the A-data.

The innermost loops m_acc, n_acc, kh,kw will access at most M_ACC kB of A data and KH×KW×N_ACC kB of B-data, which, in some cases (e.g., KH=KW=3, N_ACC=4) might already overflow the L1 cache size. Thus, several opportunities for priority inversions exist in this more complex loop structure, depending on the parameters in the table above:

- B-data evicting reusable A-data at the kh,kw loops level.
- A-data evicting reusable B-data at the m loop level.
- B-data evicting reusable A-data at the n loop level.
- A-data evicting reusable B-data at the mb loop level.
- B-data evicting reusable A-data at the nb loop level.

Solution to Priority Inversions: Non-Temporal Loads

Intel AMX architecture introduces a way to load tile registers bypassing the L1 cache via non-temporal tile loads (TILELOADDT1). This allows the user to deal with priority inversions such as those described above by loading the large, non-reusable data chunk with non-temporal loads. Thus, the larger chunk is prevented from evicting the smaller, frequently used data chunk. In Table 20-8, the A-tiles are loaded with non-temporal loads while loading B-tiles with temporal loads. This ensures the B-tile loads at the m loop level will all come from the L1 cache. In Table 20-9, the B-tiles are loaded with non-temporal loads while loading A-tiles with temporal loads, thus ensuring that the A-tile loads at the n loop level will all come from the SL1 cache.

20.9 USING LARGE TILES IN SMALL CONVOLUTIONS TO MAXIMIZE DATA REUSE

A convolution with a small-sized input frame can make the Intel AMX computation inefficient.

Consider the following example: a 7x7 input frame, with padding of 1 (size including padding is 9x9), convolved with a 3x3 filter to produce a 7x7 output frame.

Figure 20-8 shows the pieces participating in the convolution (in yellow) interacting with the khaki=0,0 weight element.

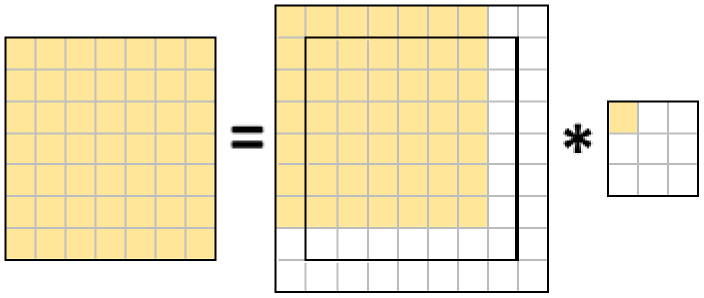


Figure 20-8. A Convolution Example

Thus, the yellow parts of the input frame are the only ones that should be loaded into A-tiles when processing weight element kh,kw=0,0. The white parts of the input frame should be ignored. This requires the number of tile rows to be set at seven, utilizing less than half of the A-tile, reducing B (weights) data reuse by a factor of two. Each A-tile is now half the size, and seven tiles are required to cover the spatial dimension. Because there are not seven tiles, B-tiles must be loaded twice as many times, potentially leading to significant performance degradation, depending on the size of the weights. This is usually inversely proportional to the spatial size of the input frame).

Figure 20-9 shows three A-tiles with sixteen rows and one tile with seven rows to cover the entire spatial dimension of the convolution.

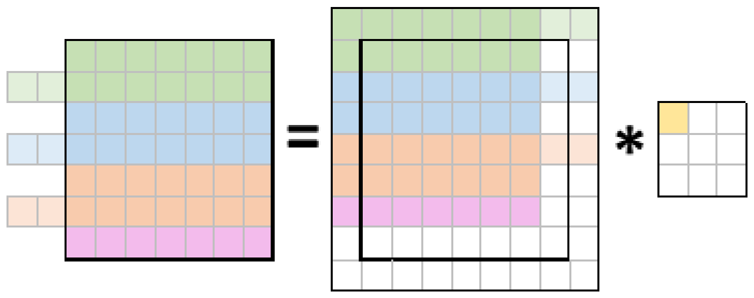


Figure 20-9. A Convolution Example with Large Tiles.

Each tile is highlighted differently. The green, blue, and orange tiles now load those two “extra” pieces previously ignored. Those pieces will waste compute resources and take up two rows in the accumulator tiles. The user may choose to ignore those rows in subsequent computations (e.g., int8-quantization, RELU, etc.), complicating the implementation. The potential benefit of increased B-data reuse could be dramatic, however.

20.10 HANDLING INCONVENIENTLY-SIZED ACTIVATIONS

Occasionally, the spatial dimensions of an activation might be ill-suited for efficient tiling with tiles. Consider a GEMM with activations’ M=100. This poses a challenge: while the M dimension can be neatly tiled by ten tiles, each with ten rows, this approach is inefficient since a larger M dimension of 112 requires only seven tiles with sixteen rows. This means that the data reuse for M=100 is 30% worse than for M=112.

The following solutions will be useful:

1. Define two types of A- and C-tiles – tiles with 16 rows and one tile with four. Use tiles of the first type for M=0..9 and the second type tile for M=96..99.
2. Allocate extra space in A and C buffers, as if M=112, and use tiles with 16 rows exclusively. The extra space need not be zeroed out or otherwise prepared in any way. In this case, the last (seventh) tile will load four meaningful rows (M=96..99) and twelve “garbage” rows (M=100..111). At the output, tile C will have four meaningful rows (M=96..99) and twelve “garbage” rows (M=100..111) which the user can then ignore.

The first solution does not require tampering with the A and C buffers and computes 100 tile rows, producing a clean result. Still, it requires additional A- and C-tiles. unused throughout the computation except at the very end. Since only eight tiles are available, this requirement can be costly and might **reduce** the data reuse (e.g., to use a 2D accumulator array, you would need three x2 C-tiles, two A-tiles, and two B-tiles, equaling ten tiles). The second solution avoids this requirement by complicating buffer handling and paying with additional loads, compute, and storing (it loads, computes, and stores 112 tile rows).

20.11 POST-CONVOLUTION OPTIMIZATIONS

Most Intel AMX-friendly applications are from the Deep Learning domain, where the data flows through multiple layers. It is often necessary to process the convolution output before passing it as an input to the next layer (processing operations depend on a specific application). This stage is called **post-convolution**.

20.11.1 POST-CONVOLUTION FUSION

As with Intel AVX-512 code, a critical optimization is the "fusion" of post-convolutional operations to the convolutional data they operate upon. Fusion reduces the memory hierarchy thrashing. Additionally, fusing the quantization step gains x2 (for bfloat16 data type) or x4 (for int8 data type) compute bandwidth, and reduces memory bandwidth by x2 or x4, respectively.

Consider the code in Example 20-8. Lines 7-24 contain the entire GEMM operation for any M, N coordinate in the output. Thus, the optimal location to post-process the data computed in lines 7-24 is right before line 24 while it is still in the low-level cache.

In Example 20-17, blue code illustrates a fully unrolled example from line 7 through 24, for int8 GEMM with K=192, N_ACC=M_ACC=2, TILE_M=2, TILE_K=64, TILE_N=16. The convolution code is fused with post-convolution code (blue) that quantizes the output and ReLU. To keep the post-convolution code in the example short, an unrealistically low value of TILE_M=2 was chosen.

In that example, an additional buffer, temporary_C, contains the convolutional results of M_ACCxN_ACC tiles. The results are stored at the end of the convolutional part and loaded during the post-convolutional part. A temporary buffer is required because the size of the post-processed data is four times smaller. Hence, the convolutional output cannot be written directly to the output buffer.

The GPRs r8, r9, r10, r11, and r14 point to the current location in the A, B, C, temporary_C, and q_bias (which holds the quantization factors and biases) buffers, respectively.

The macros A_OFFSET(m,k), B_OFFSET(k,n), C_OFFSET(m,n), C_TMP_OFFSET(m,n), Q_OFFSET(n), and BIAS_OFFSET(n) receive as arguments m,k,n tile indices and return the offset of the data from r8,r9,r10, r11, and r14, respectively.

Example 20-17. Convolution Code Fused with Post-Convolution Code

```
/*1 of 2*/
#define TILE_N_B              (N)
#define A_OFFSET(m,k)         ((m)*K*TILE_M + (k)*TILE_K)
#define B_OFFSET(k,n)         ((k)*N*TILE_N*4 + (n)*TILE_N*4)
#define C_OFFSET(m,n)         ((m)*N*TILE_M + (n)*TILE_N)
#define C_TMP_OFFSET(m,n)     ((m)*N*TILE_M*4 + (n)*TILE_N*4)
#define Q_OFFSET(n)           ((n)*TILE_N*4)
#define BIAS_OFFSET(n)        ((n)*TILE_N*4 + N*4)

static const tileconfig_t tc = {
  1,                                                    // Palette ID
  0,                                                    // Start row
    0,  0,  0,  0,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0,    // Reserved – must be 0
  64, 64, 64, 64, 64, 64, 64, 0, 0, 0, 0, 0, 0, 0, 0, 0, // Cols for 7 tiles used
  2, 2, 2, 2, 2, 16, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0      // Rows for tiles used: 2 for A, C,
                                                        // 16 for B
};

ldtilecfg tc                                            // Load tile config
mov         r12, 192                                    // A stride
mov         r13, 128                                    // B, C_TMP stride
tileloadd   tmm5, [r9 + r13*1 + B_OFFSET(0,0)]          // Load B [k,n] = [0,0]
tileloadd   tmm4, [r8 + r12*1 + A_OFFSET(0,0)]          // Load A [m,k] = [0,0]
tilezero    tmm0                                        // Zero acc [m,n] = [0,0]
tdpbusd     tmm0, tmm4, tmm5
tileloadd   tmm6, [r9 + r13*1 + B_OFFSET(0,1)]          // Load B [k,n] = [0,1]
tilezero    tmm2                                        // Zero acc [m,n] = [0,1]
tdpbusd     tmm2, tmm4, tmm6
tileloadd   tmm4, [r8 + r12*1 + A_OFFSET(1,0)]          // Load A [m,k] = [1,0]
tilezero    tmm1                                        // Zero acc [m,n] = [1,0]
tdpbusd     tmm1, tmm4, tmm5
tilezero    tmm3                                        // Zero acc [m,n] = [1,1]
tdpbusd     tmm3, tmm4, tmm6
tileloadd   tmm5, [r9 + r13*1 + B_OFFSET(1,0)]          // Load B [k,n] = [1,0]
tileloadd   tmm4, [r8 + r12*1 + A_OFFSET(0,1)]          // Load A [m,k] = [0,1]
tdpbusd     tmm0, tmm4, tmm5
tileloadd   tmm6, [r9 + r13*1 + B_OFFSET(1,1)]          // Load B [k,n] = [1,1]
tdpbusd     tmm2, tmm4, tmm6
tileloadd   tmm4, [r8 + r12*1 + A_OFFSET(1,1)]          // Load A [m,k] = [1,1]
tdpbusd     tmm1, tmm4, tmm5
tdpbusd     tmm3, tmm4, tmm6
tileloadd   tmm5, [r9 + r13*1 + B_OFFSET(2,0)]          // Load B [k,n] = [2,0]
tileloadd   tmm4, [r8 + r12*1 + A_OFFSET(0,2)]          // Load A [m,k] = [0,2]
tdpbusd     tmm0, tmm4, tmm5
tilestored  [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0     // Store C tmp [m,n] = [0,0]
tileloadd   tmm6, [r9 + r13*1 + B_OFFSET(2,1)]          // Load B [k,n] = [2,1]
```

```
/*2 of 2*/
tdpbusd        tmm2, tmm4, tmm6
tilestored     [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2          // Store C tmp [m,n] = [0,1]
tileloadd      tmm4, [r8 + r12*1 + A_OFFSET(1,2)]               // Load A [m,k] = [1,2]
tdpbusd        tmm1, tmm4, tmm5
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1          // Store C tmp [m,n] = [1,0]
tdpbusd        tmm3, tmm4, tmm6
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3          // Store C tmp [m,n] = [1,1]

vcvtdq2ps      zmm0 , [r11 + C_TMP_OFFSET(0,0) + 0*TILE_N_B]    // int32 -> float
vmovups        zmm1 , [r14 + Q_OFFSET(0)]                       // q-factors for N=0
vmovups        zmm2 , [r14 + BIAS_OFFSET(0)]                    // biases    for N=0
vfmadd213ps    zmm0 , zmm1 , zmm2                               // zmm0  = zmm0  * q + b
vcvtps2dq      zmm0 , zmm0                                      // float -> int32
vpxord         zmm3 , zmm3 , zmm3                               // Prepare zero ZMM
vpmaxsd        zmm0 , zmm0 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0         // uint32 -> uint8
vcvtdq2ps      zmm4 , [r11 + C_TMP_OFFSET(0,0) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm4 , zmm1 , zmm2                               // zmm4  = zmm4  * q + b
vcvtps2dq      zmm4 , zmm4                                      // float -> int32
vpmaxsd        zmm4 , zmm4 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4         // uint32 -> uint8
vcvtdq2ps      zmm5 , [r11 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm5 , zmm1 , zmm2                               // zmm5  = zmm5  * q + b
vcvtps2dq      zmm5 , zmm5                                      // float -> int32
vpmaxsd        zmm5 , zmm5 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5         // uint32 -> uint8
vcvtdq2ps      zmm6 , [r11 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm6 , zmm1 , zmm2                               // zmm6  = zmm6  * q + b
vcvtps2dq      zmm6 , zmm6                                      // float -> int32
vpmaxsd        zmm6 , zmm6 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6         // uint32 -> uint8
vcvtdq2ps      zmm7 , [r11 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]    // int32 -> float
vmovups        zmm8 , [r14 + Q_OFFSET(1)]                       // q-factors for N=1
vmovups        zmm9 , [r14 + BIAS_OFFSET(1)]                    // biases    for N=1
vfmadd213ps    zmm7 , zmm8 , zmm9                               // zmm7  = zmm7  * q + b
vcvtps2dq      zmm7 , zmm7                                      // float -> int32
vpmaxsd        zmm7 , zmm7 , zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7         // uint32 -> uint8
vcvtdq2ps      zmm10, [r11 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm10, zmm8 , zmm9                               // zmm10 = zmm10 * q + b
vcvtps2dq      zmm10, zmm10                                     // float -> int32
vpmaxsd        zmm10, zmm10, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10        // uint32 -> uint8
vcvtdq2ps      zmm11, [r11 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm11, zmm8 , zmm9                               // zmm11 = zmm11 * q + b
vcvtps2dq      zmm11, zmm11                                     // float -> int32
vpmaxsd        zmm11, zmm11, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11        // uint32 -> uint8
vcvtdq2ps      zmm12, [r11 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm12, zmm8 , zmm9                               // zmm12 = zmm12 * q + b
vcvtps2dq      zmm12, zmm12                                     // float -> int32
vpmaxsd        zmm12, zmm12, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12        // uint32 -> uint8
```

20.11.2 INTEL® AMX AND INTEL® AVX-512 INTERLEAVING (SW PIPELINING)

A modern CPU has multiple functional units that can execute different instructions simultaneously. For example, a load instruction and an arithmetic instruction can execute in parallel. A commonly used approach for maximizing the utilization of various resources in parallel is the out-of-order execution, where the CPU might alter the order of the instructions to achieve higher resource utilization.

Intel AMX compute instructions are prime candidates for optimization because they utilize resources very lightly (1/2 of the available ALU ports, 1/TILE_M of the time).

The blue post-convolutional code of one iteration could, theoretically, execute in parallel to the **Bold** code in lines 3 through 25 (before the first TileStore) of the next iteration, where iteration is the execution of the code in Example 20-17. Unfortunately, this cannot be done automatically and efficiently by the CPU: since the convolution (**Bold**) and post-convolution (blue) parts of the code tend to be sizable, the CPU can only overlap small portions of them efficiently before it runs out of resources in the out-of-order machine. Thus, a manual (SW) solution is required.

As previously written, the blue code before the first TileStore can be run in parallel with the green code of the next iteration. This would overwrite temporary_C memory, which the post-convolution code reads from. To remove this dependency and maximize parallel execution, use double-buffering on temporary_C. Temporary_C would thus contain two buffers, interchanged every iteration.

In Example 20-28, the content deviates from the previous example by interleaving the current iteration’s convolutional code with the previous iteration’s post-convolutional code. Temporary_C is double-buffered, with r11 pointing to the buffer of the current iteration and r12 pointing to the previous iteration’s buffer. They are exchanged at the end of the iteration.

Example 20-18. An Example of a Short GEMM Fused and Pipelined with Quantization and ReLU

```
/*1 of 3*/
1  ldtilecfg     tc                                          // Load tile config
2  mov           r15, 192                                    // A stride
3  mov           r13, 128                                    // B, C_TMP stride
4  tileloadd     tmm5, [r9 + r13*1 + B_OFFSET(0,0)]          // Load B [k,n] = [0,0]
5  tileloadd     tmm4, [r8 + r15*1 + A_OFFSET(0,0)]          // Load A [m,k] = [0,0]
6  tilezero      tmm0                                        // Zero acc [m,n] = [0,0]
7  vcvtdq2ps     zmm0, [r12 + C_TMP_OFFSET(0,0) + 0*TILE_N_B] // int32 -> float
8  vmovups       zmm1, [r14 + Q_OFFSET(0)]                   // q-factors for N=0
9  vmovups       zmm2, [r14 + BIAS_OFFSET(0)]                // biases    for N=0
10 vfmadd213ps   zmm0, zmm1, zmm2                            // zmm0 = zmm0 * q + b
11 vcvtps2dq     zmm0, zmm0                                  // float -> int32
12 vpxord        zmm3, zmm3, zmm3                            // Prepare zero ZMM
13 vpmaxsd       zmm0, zmm0, zmm3                            // RELU (int32)
14 tdpbusd       tmm0, tmm4, tmm5
15 tileloadd     tmm6, [r9 + r13*1 + B_OFFSET(0,1)]          // Load B [k,n] = [0,1]
16 tilezero      tmm2                                        // Zero acc [m,n] = [0,1]
17 vpmovusdb     [r10 + C_OFFSET(0,0) + 0*TILE_N_B], zmm0    // uint32 -> uint8
18 vcvtdq2ps     zmm4 , [r12 + C_TMP_OFFSET(0,0) + 4*TILE_N_B] // int32 -> float
19 vfmadd213ps   zmm4 , zmm1 , zmm2                          // zmm4  = zmm4  * q + b
20 tdpbusd       tmm2, tmm4, tmm6
21 tileloadd     tmm4, [r8 + r15*1 + A_OFFSET(1,0)]          // Load A [m,k] = [1,0]
22 tilezero      tmm1                                        // Zero acc [m,n] = [1,0]
23 vcvtps2dq     zmm4 , zmm4                                 // float -> int32
24 vpmaxsd       zmm4 , zmm4 , zmm3                          // RELU (int32)
25 vpmovusdb     [r10 + C_OFFSET(0,0) + 1*TILE_N_B], zmm4    // uint32 -> uint8
26 tdpbusd       tmm1, tmm4, tmm5
27 tilezero      tmm3                                        // Zero acc [m,n] = [1,1]
```

```
/*2 of 3*/
vcvtdq2ps      zmm5 , [r12 + C_TMP_OFFSET(1,0) + 0*TILE_N_B]     // int32 -> float
vfmadd213ps    zmm5 , zmm1 , zmm2                                // zmm5  = zmm5  * q + b
vcvtps2dq      zmm5 , zmm5                                       // float -> int32
vpmaxsd        zmm5 , zmm5 , zmm3                                // RELU (int32)
tdpbusd        tmm3, tmm4, tmm6
tileloadd      tmm5 , [r9 + r13*1 + B_OFFSET(1,0)]               // Load B [k,n] = [1,0]
tileloadd      tmm4 , [r8 + r15*1 + A_OFFSET(0,1)]               // Load A [m,k] = [0,1]
vpmovusdb      [r10 + C_OFFSET(1,0) + 0*TILE_N_B], zmm5          // uint32 -> uint8
vcvtdq2ps      zmm6 , [r12 + C_TMP_OFFSET(1,0) + 4*TILE_N_B]     // int32 -> float
vfmadd213ps    zmm6 , zmm1 , zmm2                                // zmm6  = zmm6  * q + b
tdpbusd        tmm0, tmm4, tmm5
tileloadd      tmm6, [r9 + r13*1 + B_OFFSET(1,1)]                // Load B [k,n] = [1,1]
vcvtps2dq      zmm6 , zmm6                                        // float -> int32
vpmaxsd        zmm6 , zmm6 , zmm3                                // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,0) + 1*TILE_N_B], zmm6          // uint32 -> uint8
tdpbusd        tmm2 , tmm4, tmm6
tileloadd      tmm4 , [r8 + r15*1 + A_OFFSET(1,1)]               // Load A [m,k] = [1,1]
vcvtdq2ps      zmm7 , [r12 + C_TMP_OFFSET(0,1) + 0*TILE_N_B]     // int32 -> float
vmovups        zmm8 , [r14 + Q_OFFSET(1)]                        // q-factors for N=1
vmovups        zmm9 , [r14 + BIAS_OFFSET(1)]                     // biases    for N=1
vfmadd213ps    zmm7 , zmm8 , zmm9                                // zmm7  = zmm7  * q + b
vcvtps2dq      zmm7 , zmm7                                       // float -> int32
vpmaxsd        zmm7 , zmm7 , zmm3                                // RELU (int32)
tdpbusd        tmm1 , tmm4, tmm5
vpmovusdb      [r10 + C_OFFSET(0,1) + 0*TILE_N_B], zmm7          // uint32 -> uint8
vcvtdq2ps      zmm10 , [r12 + C_TMP_OFFSET(0,1) + 4*TILE_N_B]    // int32 -> float
vfmadd213ps    zmm10 , zmm8 , zmm9                               // zmm10 = zmm10 * q + b
tdpbusd        tmm3 , tmm4, tmm6
tileloadd      tmm5 , [r9 + r13*1 + B_OFFSET(2,0)]               // Load B [k,n] = [2,0]
tileloadd      tmm4 , [r8 + r15*1 + A_OFFSET(0,2)]               // Load A [m,k] = [0,2]
vcvtps2dq      zmm10 , zmm10                                      // float -> int32
vpmaxsd        zmm10 , zmm10, zmm3                               // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(0,1) + 1*TILE_N_B], zmm10         // uint32 -> uint8
tdpbusd        tmm0, tmm4, tmm5
tilestored     [r11 + r13*1 + C_TMP_OFFSET(0,0)], tmm0           // Store C tmp [m,n] = [0,0]
tileloadd      tmm6, [r9 + r13*1 + B_OFFSET(2,1)]                // Load B [k,n] = [2,1]
vcvtdq2ps      zmm11, [r12 + C_TMP_OFFSET(1,1) + 0*TILE_N_B]     // int32 -> float
vfmadd213ps    zmm11, zmm8 , zmm9                                // zmm11 = zmm11 * q + b
vcvtps2dq      zmm11, zmm11                                      // float -> int32
vpmaxsd        zmm11, zmm11, zmm3                                // RELU (int32)
tdpbusd        tmm2, tmm4, tmm6
tilestored     [r11 + r13*1 + C_TMP_OFFSET(0,1)], tmm2           // Store C tmp [m,n] = [0,1]
tileloadd      tmm4, [r8 + r15*1 + A_OFFSET(1,2)]                // Load A [m,k] = [1,2]
vpmovusdb      [r10 + C_OFFSET(1,1) + 0*TILE_N_B], zmm11         // uint32 -> uint8
vcvtdq2ps      zmm12, [r12 + C_TMP_OFFSET(1,1) + 4*TILE_N_B]     // int32 -> float
vfmadd213ps    zmm12, zmm8 , zmm9                                // zmm12 = zmm12 * q + b
tdpbusd        tmm1, tmm4, tmm5
tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,0)], tmm1           // Store C tmp [m,n] = [1,0]
vcvtps2dq      zmm12, zmm12                                      // float -> int32
vpmaxsd        zmm12, zmm12, zmm3                                // RELU (int32)
vpmovusdb      [r10 + C_OFFSET(1,1) + 1*TILE_N_B], zmm12         // uint32 -> uint8
tdpbusd        tmm3, tmm4, tmm6
```

```
/*3 of 3*/
80   tilestored     [r11 + r13*1 + C_TMP_OFFSET(1,1)], tmm3        // Store C tmp [m,n] = [1,1]
81
82   xchg           r11, r12                                       // Swap buffers for current/next iter
```

With the exception of a larger TILE_M (N_ACC=M_ACC=2, TILE_M=16, TILE_K=64, TILE_N=16) on a [256x192] x [192x256] GEMM, application of this algorithm with the parameters laid out in section Section 20.8.1 yielded an 18.5% improvement in running time vs. the non-interleaved code described in Section 20.11.1.

20.11.3 AVOIDING THE H/W OVERHEAD OF FREQUENT OPEN/CLOSE OPERATIONS IN PORT FIVE

When the processor executes Intel AMX compute instructions (TDP*), it usually closes port five (one of the two Intel AVX-512 FMA ports) to conserve power. When the processor senses no more Intel AMX compute instructions in the pipeline, it opens port five. This open/close operation stalls the pipeline for a few cycles. Up to 20% performance degradation may be observed when the Intel AVX-512 instruction block contains 100 to 300 Intel AVX-512 instructions.

We recommend adding one or two TileZero instructions in the middle of the green block, roughly one hundred Intel AVX-512 instructions apart. Such an addition ensures that port five remains closed during blocks of up to three hundred Intel AVX-512 instructions. For longer blocks, it is preferable not to insert TileZero since longer blocks execute faster on two open FMA ports. The processor does not open port five for blocks shorter than one hundred Intel AVX-512 instructions, so no special handling is necessary.

NOTE

The TileZero instruction is considered an Intel AMX compute instruction for that matter.

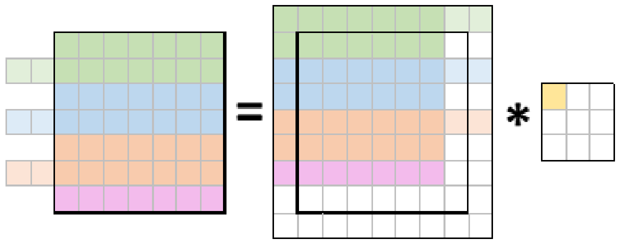


Figure 20-10. Using TileZero to Solve Performance Degradation

20.11.4 POST-CONVOLUTION MULTIPLE OFM ACCUMULATION AND EFFICIENT DOWN-CONVERSION

An important question arises concerning fused post-convolution optimization. What is the optimal block of accumulators processed in a single post-convolution iteration? As a post-processing unit, it is convenient to consider the M_ACC * N_ACC block of tiles accumulated in loops starting at lines 7-8 and 10-11 in Example 20-14 and Example 20-16, respectively. For simplicity, consider only multiples of these accumulation blocks. There is a trade-off between using smaller and larger post-convolution blocks:

Using small post-convolution blocks may have a negative impact by interrupting the convolution flow too often. Conversely, using big post-convolution blocks may also negatively impact by evicting part of the accumulated tiles out of DCU.

The optimal size, therefore, depends very much on the DL network topology and convolution-blocking parameters. Performance studies show that the number of iterations of M_ACC * N_ACC blocks before proceeding to post-convolution iteration may vary from 1 to 7.

As AMX instructions generate a higher precision output (32-bit integers or 32-bit floats) from lower precision inputs (8-bit integers or 16-bit bfloats, respectively), there is a need to convert 32-bit outputs to 8- or 16-bit inputs to be fed to the next DL network layer.

Suppose a single high-precision cache line (512-bit) is processed for conversion at a time. In that case, there will be two or four rounds of processing until a single low-precision cache line is generated for 8- or 16-bit inputs. Potential problems include:

- the number of loads and stores of the same cache line increases 4X or 2X, respectively.
- the next round of processing of the same cache line may occur after this cache line is evicted from DCU.

One of the optimizations mitigating these performance issues is to collect enough high-precision outputs to convert the full low-precision cache line in a single round.

The following drawing shows the conversion flow of 32-bit integers to 8-bit integers. Each colored block at the top represents a single **full** TILE output. The horizontal dimension is OFMs the vertical dimension is spatial).

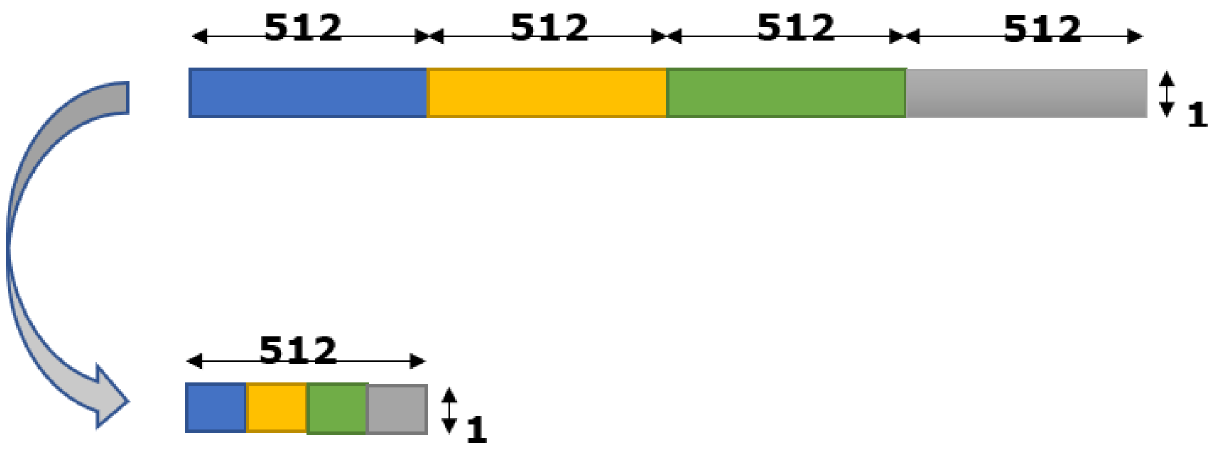


Figure 20-11. A Conversion Flow of 32-bit Integers to 8-bit Integers

To generate full 512-bit cache lines of 8-bit inputs (bottom), a multiple of 64 OFMs should be collected before conversion. Accordingly, to generate full cache lines with 16-bit inputs, a multiple of 32 OFMs should be collected. This often produces better performance results, though it may be viewed as a restriction to convolution blocking parameters (in particular, N_ACC).

Example 20-19 shows the conversion code for two blocks of sixteen cache lines of 32-bit floats converted to a single block of sixteen cache lines of 16-bit bfloats. TMUL outputs are assumed to be placed into a scratchpad *spad,* and the conversion result is placed in the next_inputs buffer.

Example 20-19. Two Blocks of 16 Cache Lines of 32-bit Floats Converted to One Block of 16 Cache Lines of 16-bit BFloat

```
float* spad;
bfloat_16* next_inputs;
inline unsigned inputs_spatial_dim( void ) {
  return /* number of pixels in map */
}
for (int i = 0; i < 16; i++)
{
__m512 f32_0 = _mm512_load_ps(spad);
     __m512 f32_1 = _mm512_load_ps(spad + 16*16);
     __m512 bf16 = _mm512_castsi512_ps(_mm512_cvtne2ps_pbh(f32_1, f32_0));
     _mm512_store_ps(next_inputs, bf16);

    spad += 16; /* Next TILE row */
    next_inputs += 32 * inputs_spatial_dim();
}
```

Example 20-20. Using Unsigned Saturation

```
const int32_t db_sel[16] = { 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15 };
inline __m512i Pack_DwordsToBytes(__m512i dwords[4])
{
  const __m512i sel_reg     = _mm512_load_si512(db_sel);
  const __m512i words_0     = _mm512_packs_epi32(dwords[0], dwords[1]);
  const __m512i words_1     = _mm512_packs_epi32(dwords[2], dwords[3]);
  __m512i bytes             = _mm512_packus_epi16(words_0, words_1);
  bytes                     = _mm512_permutexvar_epi32(sel_reg, bytes);

  return bytes;
}
```

20.12 INPUT AND OUTPUT BUFFERS REUSE (DOUBLE BUFFERING)

Due to the significant computational speedup achieved by the Intel AMX instructions, the performance bottleneck of Intel AMX-enabled applications is usually memory access. The most straightforward way to improve memory utilization is to reduce an application’s memory footprint. An application with a smaller memory footprint will keep more of its essential data in the caches while reducing the number of costly cache evictions. This usually improves performance.

In Deep Learning (DL), a simple, efficient way to reduce the memory footprint is to reuse the input and output buffers of various layers in the topology.

The following simple topology illustrates where the previous layer feeds the next layer (left):

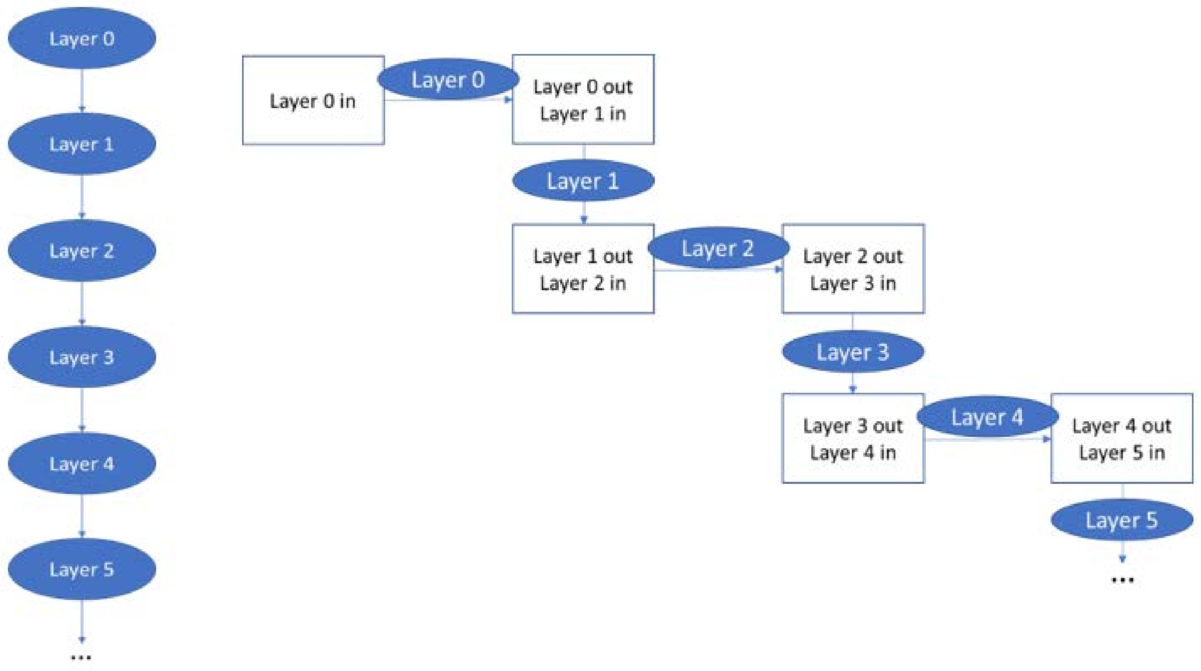


Figure 20-12. Trivial Deep Learning Topology with Naive Buffer Allocation

A straightforward buffer allocation scheme is illustrated on the in Figure 20-12, in which the output of layer N is placed into a dedicated memory buffer which is then consumed as input by layer N+1. In this scheme, such topology with L-layers would require L+1 memory buffers, of which only the last is valuable (containing the final results). The rest of the L memory buffers are single-use and disposable, significantly increasing the application's memory footprint.

The allocation scheme in Figure 20-13 offers an improved scheme whereby the entire topology only requires two reusable memory buffers.

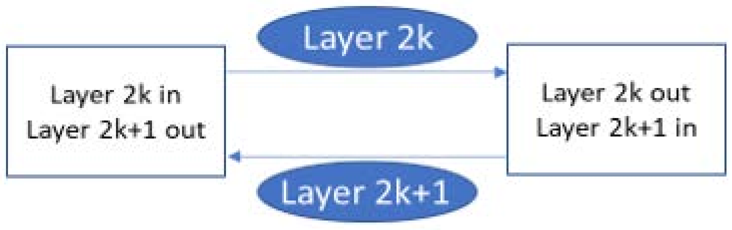


Figure 20-13. Minimal Memory Footprint Buffer Allocation Scheme for Trivial Deep Learning Topology

A more complex topology would require more reusable buffers, but this number is significantly smaller than the naïve approach. ResNet-50, for example, requires only three reusable buffers (instead of 55). Inception-ResNet-V2 requires only five reusable buffers (instead of over 250). This optimization resulted in a 25% improved performance on the int8 end-to-end large batch throughput run of Resnet50 v1.5.

20.13 SOFTWARE PREFETCHES

The CPU employs sophisticated HW prefetchers that predict future access and provide relevant data. This works best when most memory accesses are sequential. For more details on processor hardware prefetchers, see Section 20.13.1.2.

20.13.1 SOFTWARE PREFETCH FOR CONVOLUTION AND GEMM LAYERS

Since the Conv/GEMM kernel is centered around loops over the M, K, and N dimensions of the involved matrices, the access will often be sequential. However, memory blocking, also recommended in this guide, causes the CPU to re-use the same block in the A or B matrices (or both) multiple times during the kernel execution. This means that sometimes the HW prefetcher cannot predict the subsequent access correctly. This opens the opportunity for an SW prefetch algorithm tightly integrated into the Conv/GEMM kernel and can bring in cache lines from future blocks based on the blocking strategy.

While the SW prefetch instruction enables selecting the target cache hierarchy level for the prefetch, this document assumes that the prefetch will go to the MLC. The DCU is too small to prevent the prefetched lines from being evicted before they can be used, and prefetching to LLC may not yield significant improvement.

20.13.1.1 The Prefetch Strategy

The prefetch strategy is highly dependent on the Conv/GEMM kernel method of operation. Assuming the "loops and blocking" design discussed earlier, the kernel operation can probably be split into multiple phases where each phase manages a different part of the matrices (corner, middle, etc.). The developer is encouraged to reduce the program's size by reusing sections for repeatable matrix patterns to avoid overflowing the instruction cache. This can be done by having each section work on relative addresses. The SW prefetch instruction can be integrated into these sections and work on relative addresses. This means that while one section of the code loads addresses for its use, it also prefetches memory for a future section. The future section can be determined by looking at the future indices of any of the M/K/N loop levels.

20.13.1.2 Prefetch Distance

One of the most important decisions when using SW prefetching is the distance between the current and prefetched addresses. Supposing some blocking strategy is employed, it is more complex than adding an offset to the current address. The prefetched address must be set based on the target block of the matrix. If the target block is too close, the prefetched memory might still be in transit when the memory is required, and the CPU will stall, waiting for it to arrive. The data might be evicted if prefetched memory is too distant before it is used. The developer must tune the distance based on the layer/blocking parameters.

As an example heuristic:

- One or two loads for each TMUL operation.
- Where one matrix is already in a register.
- When two registers must be loaded.
- The recommended range between the prefetch time and the consumption time is between 100 and 500 TMUL operations.
- 100 TMUL operations should take about 1600 cycles.
- The maximum number of bytes loaded between prefetch and consumption is 1MB (500 TMUL ops /* 2 loads per ops /* 1K per tile).
- The optimum is probably closer to 100 TMUL ops. At any rate, the developer must check the current CPU architecture and make sure that the MLC will not overflow.

20.13.1.3 To Prefetch A or Prefetch B?

Whether to prefetch A, B, or both depends on the order of layer execution.

In general, the following approaches are available:

- Image affinity.
- Execute the next layer of the same image.
- Complete a single image end-to-end before continuing to the next image in the same mini-batch.

Layer affinity:

- Execute the same layer of the following image.
- Complete a layer for all images in the mini-batch before continuing to the next layer.

The activations (the result of the previous layer) in the CPU caches are seen when image affinity is used. The weights in the caches are found when layer affinity is used. Generally, image affinity is recommended when sizeof(A)>sizeof(B) and layer affinity otherwise. To maximize performance, the developer should tune the switch point between the two methods. The choice between these two methods is also affected by the target matrix for prefetching. If the developer is confident that one of the matrices will already be present in the cache when the Conv/GEMM kernel begins execution, the potential benefit of SW prefetching decreases dramatically.

The size of the A-matrix, B-matrix, and cache.

The developer should sum up the memory requirements of the Conv/GEMM kernel for the current layer and compare it to the size of the cache (MLC). Combined with the previous step, it can indicate whether SW prefetching can yield any performance benefit. When large matrices are involved, there is a greater chance for improvement when prefetching the A- and the B-matrices.

20.13.1.4 To Prefetch or Not to Prefetch C?

It is not the C-matrix we might want to prefetch but rather the final output matrix of the layer, after its post-convolution or post-GEMM phase, including quantization to a lower precision data type. Generally, prefetch those cache lines ahead of time since, with double buffering, these might have been read by previous layers, possibly executed in other cores.

Use the PREFETCHW instruction to read those cache lines into the DCU just in time for the store operations to find them in the DCU ready to be written, avoiding Read For Ownership latency that otherwise delays store completion. The exact timing of issuing the PREFETCHW instruction depends on multiple factors and requires careful tuning to get it right.

20.13.2 SOFTWARE PREFETCH FOR EMBEDDING LAYER

When the memory access pattern is semi-random, it is often impossible for the HW prefetcher to predict since it is based on application logic. In this case, the application may benefit from “proactive” prefetching using the SW prefetch instructions of addresses the application knows it will access soon.

An excellent example is Deep Learning, wherein the recommendation systems often use the embedding layer. The core loop of the embedding algorithm loads indices from an index buffer, and for each index, it loads the corresponding row from the embedding table for further processing. While the index buffer may contain duplicate indices that benefit from CPU caching, the pattern is often considered random or semi-random. This can make the HW prefetcher less efficient. Since the entire content of the index buffer is already known, rows soon to be encountered can be prefetched to the DCU.

Example 20-21. Prefetching Rows to the DCU

```
void prefetched_embedding(uint32_t *a, float *e, float *c, size_t num_indices,
                float scale, float bias, size_t lookahead)
{
    __m512 s = _mm512_set1_ps(scale);
    __m512 b = _mm512_set1_ps(bias);

    for (size_t i = 0; i < num_indices; i++) {
        _mm_prefetch(
            (char const *)&e[a[i + lookahead] * FLOATS_PER_CACHE_LINE],
            _MM_HINT_T0);
        __m512 ereg =
          _mm512_load_ps(&e[((size_t)a[i]) * FLOATS_PER_CACHE_LINE]);
        __m512 creg = _mm512_fmadd_ps(ereg, s, b);
        _mm512_store_ps(&c[i * FLOATS_PER_CACHE_LINE], creg);
    }
}
```

20.14 STORE TO LOAD FORWARDING

Before it gets written to the DCU (first-level cache), store instructions copy data from general purpose, vector, or tile registers into store buffers. All load instructions, other than TileLoad, can load the data they are looking for from the store buffers and memory hierarchy.

The TileLoad instruction can't load data from store buffers. It can only detect that the data is there and must wait until it is written to the memory hierarchy. Once written, TileLoad can read it from the memory hierarchy. This incurs a significant slowdown.

TileStore forwarding to non-TileLoad instructions via store buffers is supported under one restriction: they must both be of cache line size (64 bytes).

Forwarding is generally not advised because this mechanism has outliers. To avoid store-to-load forwarding, ensure enough distance between those two operations in the order of 10s of cycles in time.

20.15 MATRIX TRANSPOSE

This section describes the best-known SW implementations for several matrix transformations of BF16 data.

In this context, **flat format** means:

- Normal (i.e., non-VNNI).
- Unblocked rows (rows of matrices occupy a consecutive region in memory).

The first condition is essential. The second could be relaxed by changing the code in Example 20-22 accordingly. **VNNI format** implies only the second condition (non-blocking of rows). It is important to note that the MxN matrix in flat format will be represented by a (M/2)x(N/*2) matrix in VNNI format.

20.15.1 FLAT-TO-FLAT TRANSPOSE OF BF16 DATA

The primitive block transposed in this algorithm is 32x8 (i.e., 32 rows, eight BF16 numbers each), which is transformed into an 8x32 block (i.e., eight rows of 32 BF16 numbers each).

The implementation uses sixteen ZMM registers and three mask registers.

Input parameters: MxN, sizes of the rectangular block to be transposed. Assuming M is a multiple of 32, and N is a multiple of eight, we may also assume in Example 20-22:

- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains starting address of the input matrix.
- r9 contains starting address of the output buffer.

Example 20-22. BF16 Matrix Transpose (32x8 to 8x32)

```
/*1 of 2 */
mov        r10,      0xf0
kmovd      k1,       r10d
mov        r10,      0xf00
kmovd      k2,       r10d
mov        r10,      0xf000
kmovd      k3,       r10d
mov        rax,      N / 8
L.N:
mov        rdx,      M / 32
L.M:
vbroadcasti32x4 zmm0,              xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1},          xmmword ptr [r8+I_STRIDE*8]
vbroadcasti32x4 zmm0{k2},          xmmword ptr [r8+I_STRIDE*16]
vbroadcasti32x4 zmm0{k3},          xmmword ptr [r8+I_STRIDE*24]
vbroadcasti32x4 zmm1,              xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1},          xmmword ptr [r8+I_STRIDE*9]
vbroadcasti32x4 zmm1{k2},          xmmword ptr [r8+I_STRIDE*17]
vbroadcasti32x4 zmm1{k3},          xmmword ptr [r8+I_STRIDE*25]
vbroadcasti32x4 zmm2,              xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm2{k1},          xmmword ptr [r8+I_STRIDE*10]
vbroadcasti32x4 zmm2{k2},          xmmword ptr [r8+I_STRIDE*18]
vbroadcasti32x4 zmm2{k3},          xmmword ptr [r8+I_STRIDE*26]
vbroadcasti32x4 zmm3,              xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm3{k1},          xmmword ptr [r8+I_STRIDE*11]
vbroadcasti32x4 zmm3{k2},          xmmword ptr [r8+I_STRIDE*19]
vbroadcasti32x4 zmm3{k3},          xmmword ptr [r8+I_STRIDE*27]
vbroadcasti32x4 zmm4,              xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm4{k1},          xmmword ptr [r8+I_STRIDE*12]
vbroadcasti32x4 zmm4{k2},          xmmword ptr [r8+I_STRIDE*20]
vbroadcasti32x4 zmm4{k3},          xmmword ptr [r8+I_STRIDE*28]
vbroadcasti32x4 zmm5,              xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm5{k1},          xmmword ptr [r8+I_STRIDE*13]
vbroadcasti32x4 zmm5{k2},          xmmword ptr [r8+I_STRIDE*21]
vbroadcasti32x4 zmm5{k3},          xmmword ptr [r8+I_STRIDE*29]
vbroadcasti32x4 zmm6,              xmmword ptr [r8+I_STRIDE*6]
```

```
/*2 of 2 */
vbroadcasti32x4 zmm6{k1},          xmmword ptr [r8+I_STRIDE*14]
vbroadcasti32x4 zmm6{k2},          xmmword ptr [r8+I_STRIDE*22]
vbroadcasti32x4 zmm6{k3},          xmmword ptr [r8+I_STRIDE*30]
vbroadcasti32x4 zmm7,              xmmword ptr [r8+I_STRIDE*7]
vbroadcasti32x4 zmm7{k1},          xmmword ptr [r8+I_STRIDE*15]
vbroadcasti32x4 zmm7{k2},          xmmword ptr [r8+I_STRIDE*23]
vbroadcasti32x4 zmm7{k3},          xmmword ptr [r8+I_STRIDE*31]
vpunpcklwd     zmm8,     zmm0,     zmm1
vpunpckhwd     zmm9,     zmm0,     zmm1
vpunpcklwd     zmm10,    zmm2,     zmm3
vpunpckhwd     zmm11,    zmm2,     zmm3
vpunpcklwd     zmm12,    zmm4,     zmm5
vpunpckhwd     zmm13,    zmm4,     zmm5
vpunpcklwd     zmm14,    zmm6,     zmm7
vpunpckhwd     zmm15,    zmm6,     zmm7
vpunpckldq     zmm0,     zmm8,     zmm10
vpunpckhdq     zmm1,     zmm8,     zmm10
vpunpckldq     zmm2,     zmm9,     zmm11
vpunpckhdq     zmm3,     zmm9,     zmm11
vpunpckldq     zmm4,     zmm12,    zmm14
vpunpckhdq     zmm5,     zmm12,    zmm14
vpunpckldq     zmm6,     zmm13,    zmm15
vpunpckhdq     zmm7,     zmm13,    zmm15
vpunpcklqdq    zmm8,     zmm0,     zmm4
vpunpckhqdq    zmm9,     zmm0,     zmm4
vpunpcklqdq    zmm10,    zmm1,     zmm5
vpunpckhqdq    zmm11,    zmm1,     zmm5
vpunpcklqdq    zmm12,    zmm2,     zmm6
vpunpckhqdq    zmm13,    zmm2,     zmm6
vpunpcklqdq    zmm14,    zmm3,     zmm7
vpunpckhqdq    zmm15,    zmm3,     zmm7
vmovdqu16      zmmword ptr [r9],                zmm8
vmovdqu16      zmmword ptr [r9+O_STRIDE],       zmm9
vmovdqu16      zmmword ptr [r9+O_STRIDE*2],     zmm10
vmovdqu16      zmmword ptr [r9+O_STRIDE*3],     zmm11
vmovdqu16      zmmword ptr [r9+O_STRIDE*4],     zmm12
vmovdqu16      zmmword ptr [r9+O_STRIDE*5],     zmm13
vmovdqu16      zmmword ptr [r9+O_STRIDE*6],     zmm14
vmovdqu16      zmmword ptr [r9+O_STRIDE*7],     zmm15

add       r9, 0x40
add       r8, I_STRIDE*32
dec       rdx
jnz       L.M

add       r9, (O_STRIDE*8 – (M/32) * 0X40)
sub       r8, (I_STRIDE*M-0x10)
dec       rax
jnz       L.N
```

Implementation discussion:

- Lines 1-6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9-72 implement the transpose of a primitive block 32x8. It uses 16 ZMM registers (zmm0-zmm15).
- Lines 9-40 implement loading 32 quarter-cache lines into 8 ZMM registers, according to the following picture (numbers are in **bytes**):

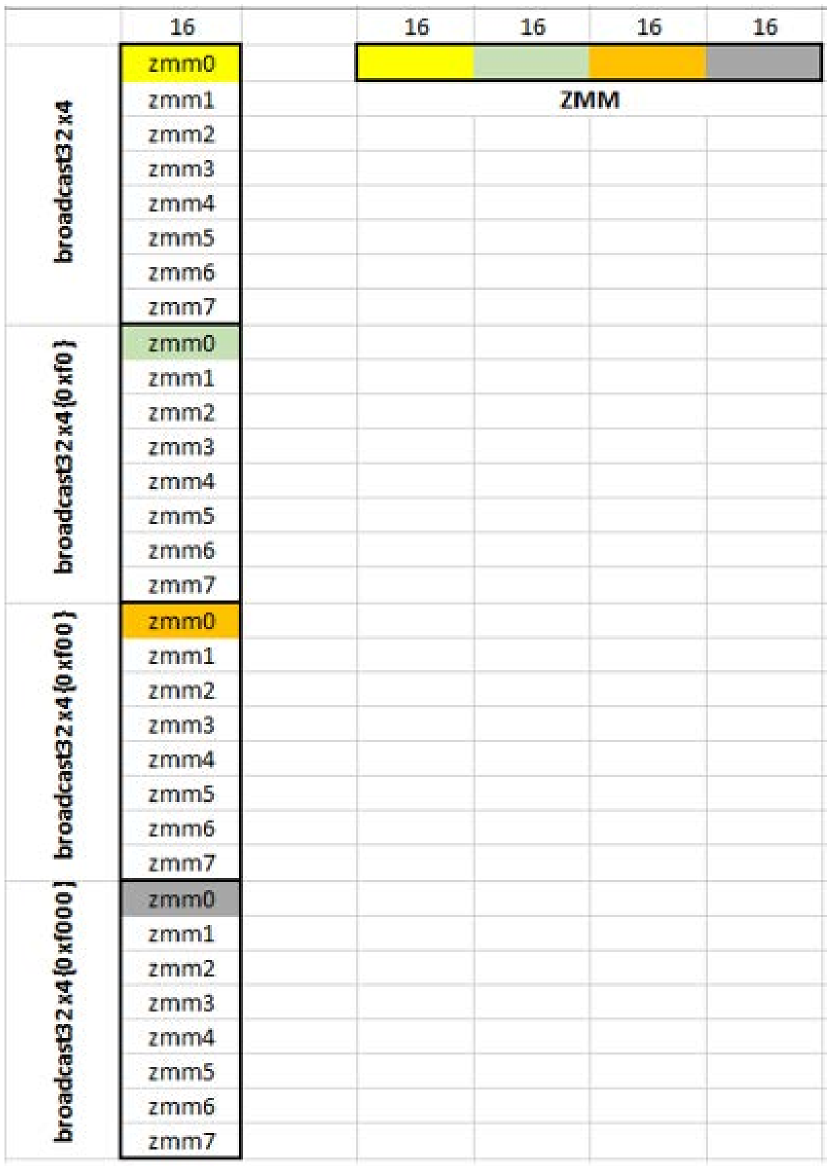


Figure 20-14. Loading 32 Quarter-Cache Lines into 8 ZMM Registers

- Lines 41-64 are transpose flow proper. It creates a transposed block 8x32 in registers zmm8-zmm15.
- Lines 65-72 store transposed block 8x32 to the output buffer.

20.15.2 VNNI-TO-VNNI TRANSPOSE

The primitive block transposed in this algorithm is 8x8 (i.e., eight rows, eight BF16 numbers each), which is transformed into a2x32 block (i.e., two rows of 32 BF16 numbers each).

The implementation uses five ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed (*in VNNI format*); it is assumed that M, N are multiples of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains starting address to the input matrix.
- r9 contains starting address to the output buffer.
- zmm31 is preloading with four copies of the following constant: unsigned int shuflle_cntrl[4] = {0x05040100, 0x07060302, 0x0d0c0908, 0x0f0e0b0a};

Example 20-23. BF16 VNNI-to-VNNI Transpose (8x8 to 2x32)

```
mov r10, 0xf0
kmovd k1, r10d
mov r10, 0xf00
kmovd k2, r10d
mov r10, 0xf000
kmovd k3, r10d
mov rax, N / 8
L.N:
mov rdx, M / 8
L.M:
vbroadcasti32x4 zmm0, xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*6]
vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*7]

vpshufb zmm2, zmm0, zmm31
vpshufb zmm3, zmm1, zmm31
vpunpcklqdq zmm0, zmm2, zmm3
vpunpckhqdq zmm1, zmm2, zmm3

vmovdqu16 zmmword ptr [r9], zmm0
vmovdqu16 zmmword ptr [r9+O_STRIDE], zmm1

add r9, 0x40
add r8, I_STRIDE*8
dec rdx
jnz L.M

add r9, (O_STRIDE*2 - (M/8) * 0x40)
sub r8, (I_STRIDE*M-0x10)
dec rax
jnz L.N
```

BF16 VNNI-to-VNNI Transpose Implementation Discussion

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–22 implement the transpose of a primitive block 32x8. It uses five ZMM registers (zmm0-zmm3, zmm31).
- Lines 9–16 implement loading eight quarter-cache lines into two ZMM registers, according to Figure 20-15 (numbers are in bytes):

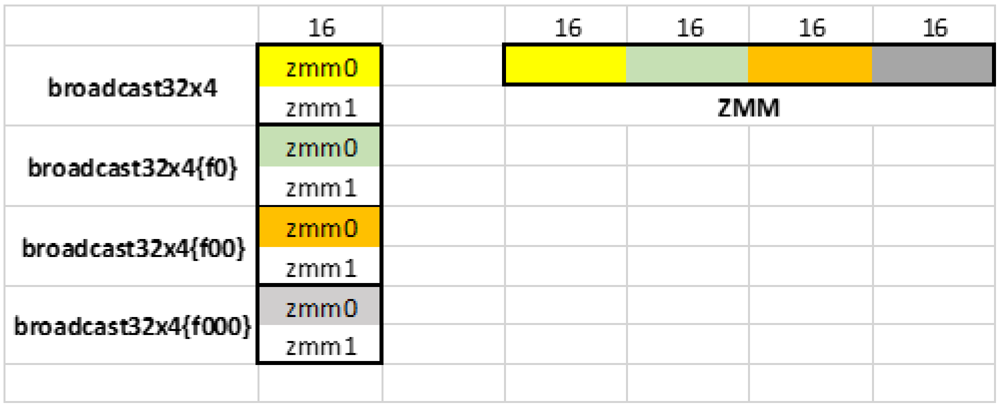
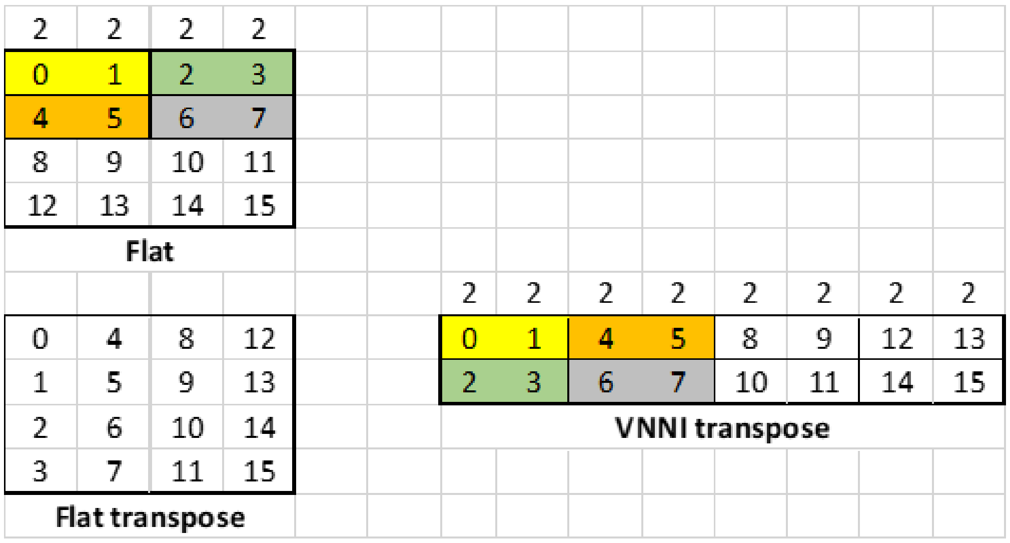


Figure 20-15. Loading Eight Quarter-Cache Lines into Two ZMM Registers

- Lines 17–20 implement simultaneous transpose of four 2x2 blocks of QWORDs (i.e., 2x8 blocks of BF16). It creates a transposed block 2x32 in registers zmm2-zmm3.
- Lines 21–22 store transposed block 2x32 to the output buffer.

20.15.3 FLAT-TO-VNNI TRANSPOSE

The algorithm below is based on: Flat-to-VNNI transpose of WORDs is equivalent to Flat-to-Flat transpose of DWORDs. This is illustrated below (the header numbers are bytes):



2	2	2	2
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Flat

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Flat transpose

2	2	2	2	2	2	2	2
0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15

VNNI transpose

Figure 20-16. Flat-to-VNNI Transpose of WORDs Equivalence to Flat-to-Flat Transpose of DWORDs

The primitive block transposed in this algorithm is 16x8 (i.e., 16 rows, 8 BF16 numbers each), which is transformed into a 4x32 block (i.e., four rows of 32 BF16 numbers each).

The implementation uses eight ZMM registers and three mask registers.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that M multiple of 16, N multiple of eight.
- I_STRIDE is the row size of the input matrix in bytes.
- O_STRIDE is the row size of the output buffer in bytes.
- r8 contains the starting address for the input matrix.
- r9 contains the starting address for the output buffer.

Example 20-24. BF16 Flat-to-VNNI Transpose (16x8 to 4x32)

```
mov r10, 0xf0
kmovd k1, r10d
mov r10, 0xf00
kmovd k2, r10d
mov r10, 0xf000
kmovd k3, r10d
mov rax, N / 8
L.N:
mov rdx, M / 16
L.M:
vbroadcasti32x4 zmm0, xmmword ptr [r8]
vbroadcasti32x4 zmm0{k1}, xmmword ptr [r8+I_STRIDE*4]
vbroadcasti32x4 zmm0{k2}, xmmword ptr [r8+I_STRIDE*8]
vbroadcasti32x4 zmm0{k3}, xmmword ptr [r8+I_STRIDE*12]
vbroadcasti32x4 zmm1, xmmword ptr [r8+I_STRIDE*1]
vbroadcasti32x4 zmm1{k1}, xmmword ptr [r8+I_STRIDE*5]
vbroadcasti32x4 zmm1{k2}, xmmword ptr [r8+I_STRIDE*9]
vbroadcasti32x4 zmm1{k3}, xmmword ptr [r8+I_STRIDE*13]
vbroadcasti32x4 zmm2, xmmword ptr [r8+I_STRIDE*2]
vbroadcasti32x4 zmm2{k1}, xmmword ptr [r8+I_STRIDE*6]
vbroadcasti32x4 zmm2{k2}, xmmword ptr [r8+I_STRIDE*10]
vbroadcasti32x4 zmm2{k3}, xmmword ptr [r8+I_STRIDE*14]
vbroadcasti32x4 zmm3, xmmword ptr [r8+I_STRIDE*3]
vbroadcasti32x4 zmm3{k1}, xmmword ptr [r8+I_STRIDE*7]
vbroadcasti32x4 zmm3{k2}, xmmword ptr [r8+ I_STRIDE*11]
vbroadcasti32x4 zmm3{k3}, xmmword ptr [r8+ I_STRIDE*15]

vpunpckldq zmm4, zmm0, zmm1
vpunpckhdq zmm5, zmm0, zmm1
vpunpckldq zmm6, zmm2, zmm3
vpunpckhdq zmm7, zmm2, zmm3
vpunpcklqdq zmm0, zmm4, zmm6
vpunpckhqdq zmm1, zmm4, zmm6
vpunpcklqdq zmm2, zmm5, zmm7
vpunpckhqdq zmm3, zmm5, zmm7

vmovups zmmword ptr [r9], zmm0
vmovups zmmword ptr [r9+O_STRIDE], zmm1
vmovups zmmword ptr [r9+O_STRIDE*2], zmm2
vmovups zmmword ptr [r9+O_STRIDE*3], zmm3

add r9, 0x40
add r8, I_STRIDE*16
dec rdx
jnz L.M

add r9, (O_STRIDE*4 - (M/16)*0x40)
sub r8, (I_STRIDE*M-0x10)
dec rax
jnz L.N
```

Implementation Discussion

- Lines 1–6 set mask registers k1, k2, k3.
- Lines 7 and 8 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 9–36 implement the transpose of a primitive block 16x8. It uses eight ZMM registers (zmm0–zmm7).
- Lines 9–24 implement loading 16 quarter-cache lines into four ZMM registers zmm0-zmm3, according to Figure 20-17 (numbers are in bytes):

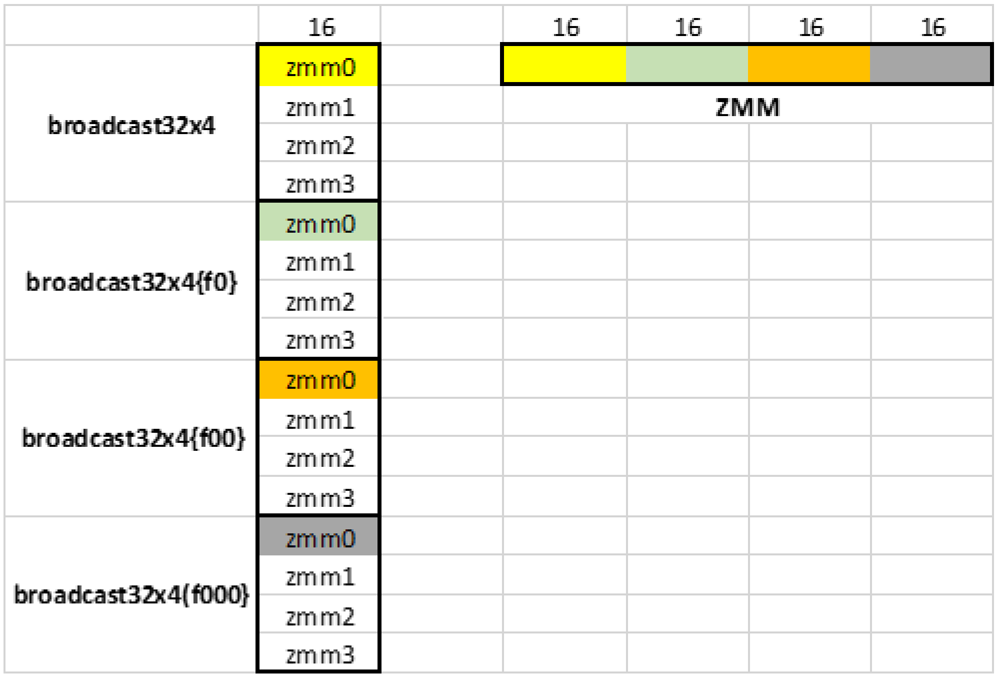


Figure 20-17. BF16 Flat-to-VNNI Transpose

- Lines 25–32 are the transpose flow proper. It creates a transposed block 4x32 in registers zmm0–zmm3.
- Lines 33–36 store transposed block 4x32 to the output buffer.

20.15.4 FLAT-TO-VNNI RE-LAYOUT

The primitive block which is being re-layout in this algorithm is 2x32 (i.e., 2 rows, 32 BF16 numbers each), which is transformed into a 1x64 block (i.e., 1 rows of 64 BF16 numbers each).

The implementation uses **5 ZMM registers and no mask registers**.

Input parameters:

- MxN, sizes of the rectangular block to be transposed; it is assumed that **M multiple of 2, N multiple of 32.**
- I_STRIDE is the row size of input matrix in **bytes**.
- O_STRIDE is the row size of output buffer in **bytes**.
- r8 contains starting address to input matrix.
- r9 contains starting address to output buffer.
- zmm30, zmm31 are preloaded with following constants, respectively:

- const short perm_cntl_1[32] = {0x00, 0x20, 0x01, 0x21, 0x02, 0x22, 0x03, 0x23, 0x04, 0x24, 0x05, 0x25, 0x06, 0x26, 0x07, 0x27, 0x08, 0x28, 0x09, 0x29, 0x0a, 0x2a, 0x0b, 0x2b, 0x0c, 0x2c, 0x0d, 0x2d, 0x0e, 0x2e, 0x0f, 0x2f};
- const short perm_cntl_2[32] = {0x30, 0x10, 0x31, 0x11, 0x32, 0x12, 0x33, 0x13, 0x34, 0x14, 0x35, 0x15, 0x36, 0x16, 0x37, 0x17, 0x38, 0x18, 0x39, 0x19, 0x3a, 0x1a, 0x3b, 0x1b, 0x3c, 0x1c, 0x3d, 0x1d, 0x3e, 0x1e, 0x3f, 0x1f};

Example 20-25. BF16 Flat-to-VNNI Re-Layout

```
1     mov rdx, M / 2
L.M:
 2    mov rax, N / 32
L.N:
 3    vmovups zmm0, zmmword ptr [r8]
 4    vmovups zmm1, zmmword ptr [r8+I_STRIDE]

 5    vmovups zmm2, zmm0
 6    vpermt2w zmm2, zmm30, zmm1
 7    vpermt2w zmm1, zmm31, zmm0

 8    vmovups zmmword ptr [r9], zmm2
 9    vmovups zmmword ptr [r9+0x40], zmm1

10    add r9, 0x80
11    add r8, 0x40
12    dec rax
13    jnz L.N

14    add r9, (O_STRIDE - (N/32)*0x80)
15    add r8, (I_STRIDE*2 – (N/32)*0x40)
16    dec rdx
17    jnz L.M
```

BF16 Flat-to-VNNI Re-Layout Implementation Discussion

- Lines 1, 2 put trip counts for primitive blocks in N- and M-dimensions, respectively.
- Lines 3, 4 implement loading two full cache lines into two ZMM registers zmm0-zmm1, from consecutive rows of the input matrix.
- Lines 5 through 7 implement the re-layout of a primitive block 2x32. It uses five ZMM registers (zmm0–zmm2, zmm30-zmm31).
- Lines 8, 9 implement storing two full cache lines in two ZMM registers zmm1-zmm2, into consecutive columns of the output matrix.

20.16 MULTI-THREADING CONSIDERATIONS

20.16.1 THREAD AFFINITY

As with Intel AVX-512 code, it is advised to fully define thread affinity and object affinity to process a single object in the same physical core, thus keeping the activations in core caches (unless larger than the size of the caches). This advice becomes imperative with Intel AMX code since those applications are more sensitive to memory-related issues.

20.16.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT)

Running more than one Intel AMX thread on the same physical core may result in overall performance loss due to the two threads competing for the same hardware resources. Scheduling a non-Intel AMX thread next to an Intel AMX thread on the same core may decrease the thread performance more than one expects due to normal competition for resources.

For optimum performance, please choose one of the following options in priority order:

1. Schedule one Intel AMX thread per physical core on one of its logical processors, while leaving the other logical processors idle.
2. Affintize a software thread that executes an endless TPAUSE CO.2 loop next to the Intel AMX thread.
    a. This prevents other threads from being scheduled on that logical processor.
        1) All hardware resources within the physical core are therefore allocated to the Intel AMX thread.
        2) This endless loop thread must terminate when the Intel AMX thread is about to terminate.
3. Code pause loops of thread pool threads that are waiting for the next task to be assigned to them with UMWAIT or TPAUSE C0.2 rather than with PAUSE, TPAUSE C0.1, or a non-pausing spin.

20.16.3 WORK PARTITIONING BETWEEN CORES

Deep Learning (DL) applications must often adhere to latency requirements that cannot be fulfilled within a single core. In these cases, a single object's processing must be partitioned between multiple cores.

Additionally, often the output of one layer is the input of the next layer. Due to the nature of the computations in DL applications, partitioning over different dimensions (N, M, K) will have different implications for the data locality in the core's caches. Minimize importing data from a different core's caches if possible as this can hamper performance.

20.16.3.1 Partitioning Over M

Partitioning a DL layer over the M dimension has the advantage of nearly complete data locality. The layer's output is also partitioned by M between the cores and is, therefore, already in the cache of the corresponding core at the beginning of the next layer. Figure 20-18 shows this schematically.

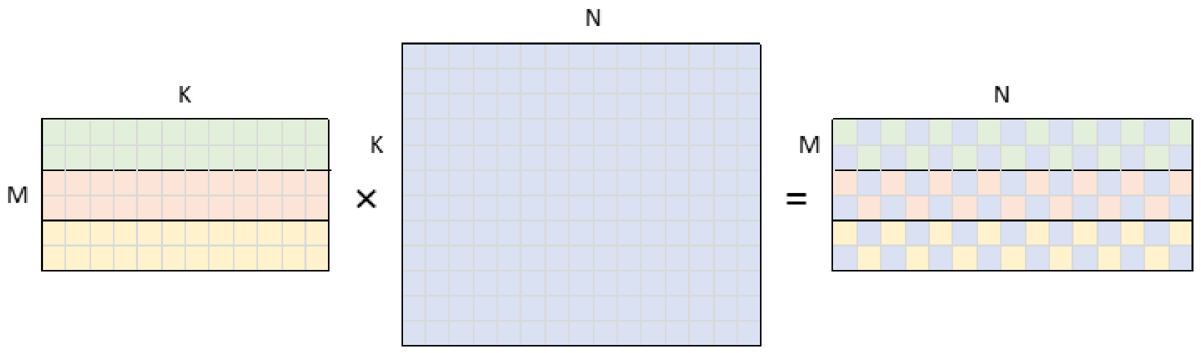


Figure 20-18. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the M-Dimension

Here the data read and written by each of the three cores is bound by a black rectangle.
It should be noted that in the case of convolutions, limited overlap in the M-dimension of the activations occurs between neighboring cores. Due to the convolutions, a finite-sized filter is slid over the activations. Thus, the M-dimension overlaps (KH-1)/*W (refer to Example 20-13) between the two neighboring cores.

- **Advantages:** When multiple layers in a chain are partitioned by the M-dimension between the same number of cores, each core has its data in its local cache.
- **Disadvantages:** All the cores read the B-matrix (or weights in convolutions) entirely, which might pose a bandwidth problem if the B-matrix is large.

20.16.3.2 Partitioning Over N

Partitioning a DL layer over the N-dimension reduces the read bandwidth in GEMMs with large B-matrices or large weights in convolutions. Each core reads a portion of the B-matrix in this scenario:

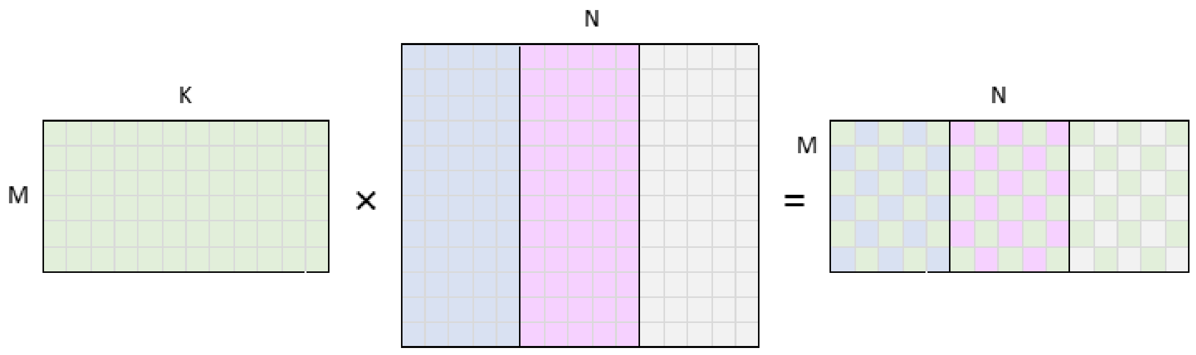


Figure 20-19. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the N-Dimension

Unfortunately, the output of the layer is also partitioned by the N-dimension between the cores, which is incompatible with M and N partitioning of the subsequent layer. For visualization, compare the right side of Figure 20-19 to the left side of Figures 20-18 and 20-19. In this scenario, a core in the subsequent layer is guaranteed to have most of its data from outside its local caches. This is not the case in K-dimension partitioning (see Section 20.16.3.3), but it also comes at a price.

- **Advantages:** It may reduce read bandwidth significantly in case of large B / large weights.
- **Disadvantages:** If the next layer is partitioned by M or by N, most of the activations in the next layer will not reside in the local caches of the corresponding cores.

20.16.3.3 Partitioning Over K

Partitioning a DL layer over the K-dimension reduces the read bandwidth in GEMMs with large K-dimensions by reducing the amount of data being read from the A- and B-matrices (activations and weights in convolutions). Each core reads a portion of the matrices in this scenario, as illustrated in Figure 20-20.

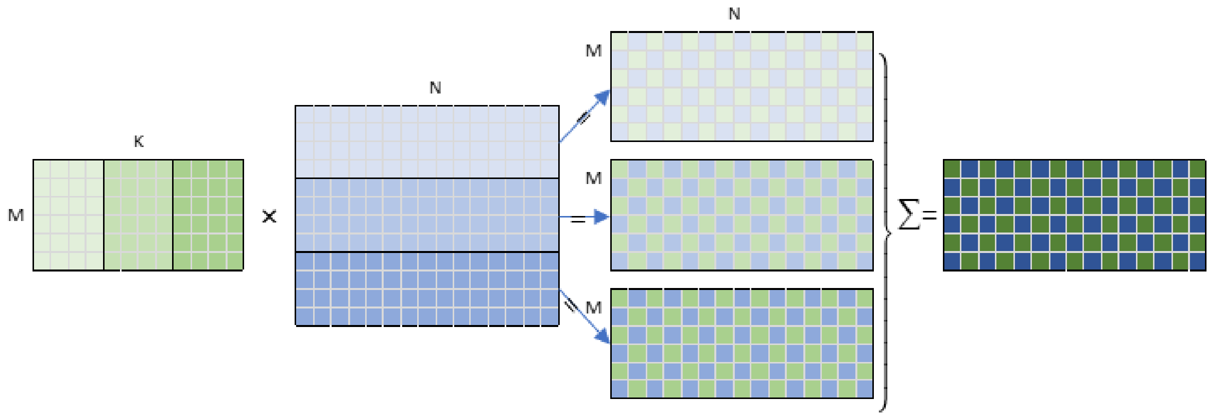


Figure 20-20. GEMM Data Partitioning Between Three Cores in a Layer Partitioned by the K-Dimension

Additionally, if a layer is partitioned by the N-dimension and the subsequent layer is partitioned by the K-dimension, the activation data will reside in the local caches of the cores in layer partitioned by the K-dimension. For visualization, compare the right side of Figure 20-19 with the left side of Figure 20-20. Unfortunately, this comes at a price: each core prepares partial results of the entire C-matrix. To obtain final results, either a mutex (or several mutexes) is required to guard the write operations into the C-matrix, or a reduction operation is needed at the end of the layer. The mutex solution is not advised because threads will be blocked for a significant time. A reduction runs the risk of being costly since it entails the following:

- A synchronization barrier is required before the reduction.
- Reading a potentially large amount of data during the reduction:
  - There are T copies of the C-matrix, where T is the number of threads (the example has three).
  - The size of the matrices before the reduction is x2 (in case of a bfloat16 datatype) or x4 (in case of int8 datatype) times larger than the output C-matrix.
  - During the reduction, most of the cores’ data will come outside their local cache hierarchy.

20.16.3.4 Memory Bandwidth Implications of Work Partitioning Over Multiple Dimensions

OpenMP offers a convenient interface for nested loop parallelization. For example, one could partition the N, M, and K dimensions can be partitioned automatically between threads using Example 20-26.

Example 20-26. GEMM Parallelized with omp Parallel for Collapse

```
#pragma omp parallel for collapse(2)

for (int n = 0; n < N; n += N_ACC*TILE_N) {
  for (int m = 0; m < M; m += M_ACC*TILE_M) {
    ...
  }
}
```

The collapse clause specifies how many loops within a nested loop should be collapsed into a single iteration space and divided between the threads. The order of the iterations in the collapsed iteration space is the same as though they were executed sequentially.

If there is no specified schedule, OpenMP automatically uses schedule(static,1), resulting in the sequential assignment of loop iterations to threads.

If we assume `N=4*N_ACC*TILE_N` and `M=4*M_ACC*TILE_M` wherein the K-dimension is deliberately excluded from consideration due to its problematic nature, there would be 4*4=16 iterations in the two nested loops. Now assume the division of iterations between three threads. As shown in Table 20-11, the code in Example 20-26 would result in a partition of the iterations between threads.

Table 20-11. A Simple Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.3	1.2	2.1	3.0	3.3	100%	100%	38%
Thread 1:	0.1	1.0	1.3	2.2	3.1		100%	100%	100%
Thread 2:	0.2	1.1	2.0	2.3	3.2		100%	100%	100%

Where every cell of the form n',m' contains the n'=n/N_ACC*TILE_N and m'=m/M_ACC*TILE_M indices from the loops in Example 20-19.

It is clear from Table 20-11 that each of the three threads executes at least one iteration with n'=0,1,2,3 and at least one iteration with m'=0,1,2,3. This means that every thread reads all of A and all of B.

By rearranging the work between threads in the following partitioning, the size of the B read is reduced by each thread by 50%, which might be significant in workloads where B is large. Similarly, the size of A can be reduced by 50% by swapping m' and n' indices for workloads with a large A.

Table 20-12. An Optimized Partition of Work Between Three Threads

							A	B	C
Thread 0:	0.0	0.1	0.2	0.3	3.0	3.1	100%	50%	38%
Thread 1:	1.0	1.1	1.2	1.3	3.2	3.3	100%	50%	38%
Thread 2:	2.0	2.1	2.2	2.3			100%	25%	25%

20.16.4 RECOMMENDATION SYSTEM EXAMPLE

Many recommendation systems are built from a few GEMM layers that follow each other, an Embedding layer, and a layer connecting them. They are generally split into four distinct tasks:

1. Bottom GEMMs (MLPs).
2. Embedding.
3. Bottom MLP + Embedding Concat, GEMM, and Reshape.
4. Top GEMMs (MLPs).

The first two are independent so that they can execute in parallel. Their output feeds into the third task, whose output, in turn, feeds into the fourth task.

A few notes:

- Recommendation systems usually use a large batch to rank a reasonably large set of options.
- The GEMM layers are usually compute- or cache-bandwidth limited, whereas the Embedding layer is memory-bandwidth limited.
- Recommendation systems are real-time and therefore limited to a specific latency.

When the latency requirement is a few milliseconds, the recommendation system topology has to be multi-threaded across several cores. The previous section discussed GEMM partitioning across multiple cores. This section deals with work partition between the four different tasks.

Figure 20-21 proposes a way to split the three tasks across machine cores. The block sizes in the chart are for illustration purposes only and do not represent any specific recommendation system.

Those three tasks can then be split into two tasks due to Bottom MLPs and Embedding independence. Those two tasks feed the other tasks: Bottom MLP + Embedding Concat, GEMM, Reshape, and Top MLPs. The latter tasks are merged into a single task. Choosing the number of cores for each task is a trial-and-error exercise. It may involve a phase for analyzing time required to execute each task across different cores.

Because of a dependency between the Bottom MLPs, Embedding tasks, and the third task, a barrier exists between them, implying a potential wait-time immediately following the faster layers.

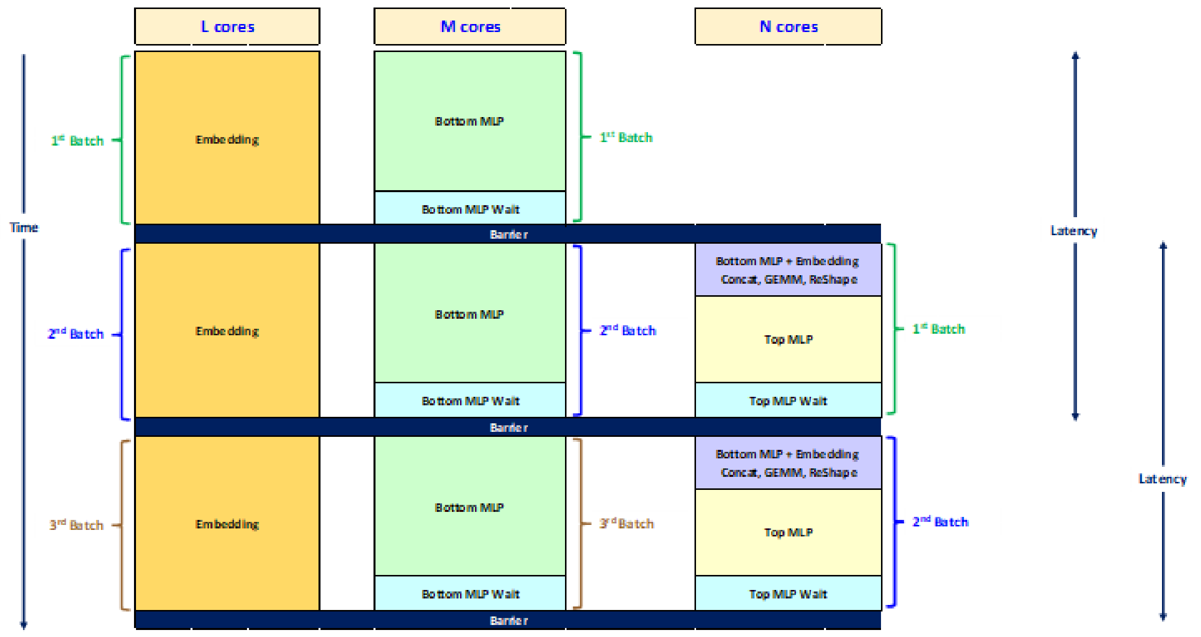


Figure 20-21. A Recommendation System Multi-Threading Model

20.17 SPARSITY OPTIMIZATIONS FOR INTEL® AMX

This section describes how Intel AMX can be further optimized for operations on sparse matrices. An example use case can be the inference of sparse neural networks, where the sparse weights are known to initially reside in DRAM due to the "online" usage model or large model capacity. In those cases, the primary performance bottleneck would be bringing the weights from DRAM. A helpful optimization technique for this case is to get the weights from DRAM in a compressed format, decompress them into the local caches using Intel AVX-512, and perform Intel AMX computations on the decompressed data.

The compressed matrix format can consist of the following components:

- **compressed[]:** an array of non-zero matrix entries.
- **mask[]:** a bit-per-element array for the full matrix. 0 signifies the corresponding element is 0. 1 signifies a non-zero value that exists in the **compressed[]** array mentioned above.

The compressed format can be computed off-line. The sparsity bitmask **mask[]** can be generated using the Intel AVX-512 *VPTESTMB* instruction on the sparse data. The **compressed[]** array can be generated using the Intel AVX-512 *VPCOMPRESS* instruction on the sparse data using the sparsity bitmask.

The code in Example 20-27 uses Intel AVX-512 to generate ***num*** rows of decompressed data, assuming 8-bit elements and 64 elements per tile row.

Example 20-27. Byte Decompression Code with Intel® AVX-512 Intrinsics

```
// uint8_t* compressed_ptr is a pointer to compressed data array
// __mmask64* compression_masks_ptr is a pointer to bitmask array
// uint8_t* decompressed_ptr is a pointer to decompressed data array

for (int i=0; i < num ; i++) {
  __m512i compressed = _mm512_loadu_epi32(compressed_ptr);
  __mmask64 mask = _load_mask64(compression_masks_ptr);
  __m512i decompressed_vec = _mm512_maskz_expand_epi8(mask, compressed);
  _mm512_store_epi32(decompressed_ptr, decompressed_vec);
  decompressed_ptr += 64; // 64 bytes per decompressed row
  compressed_ptr += _mm_countbits_64(mask); // advance compressed pointer by number of non-zero elements
  compression_masks_ptr ++; //64 bitmask bits per decompressed row
}
```

The matrix multiplication code will load the decompressed matrix to tiles from **decompressed[]**, an array containing the decompressed matrix data.

The decompression code makes use of the Intel AVX-512 date expand operation is shown in Figure 20-22.

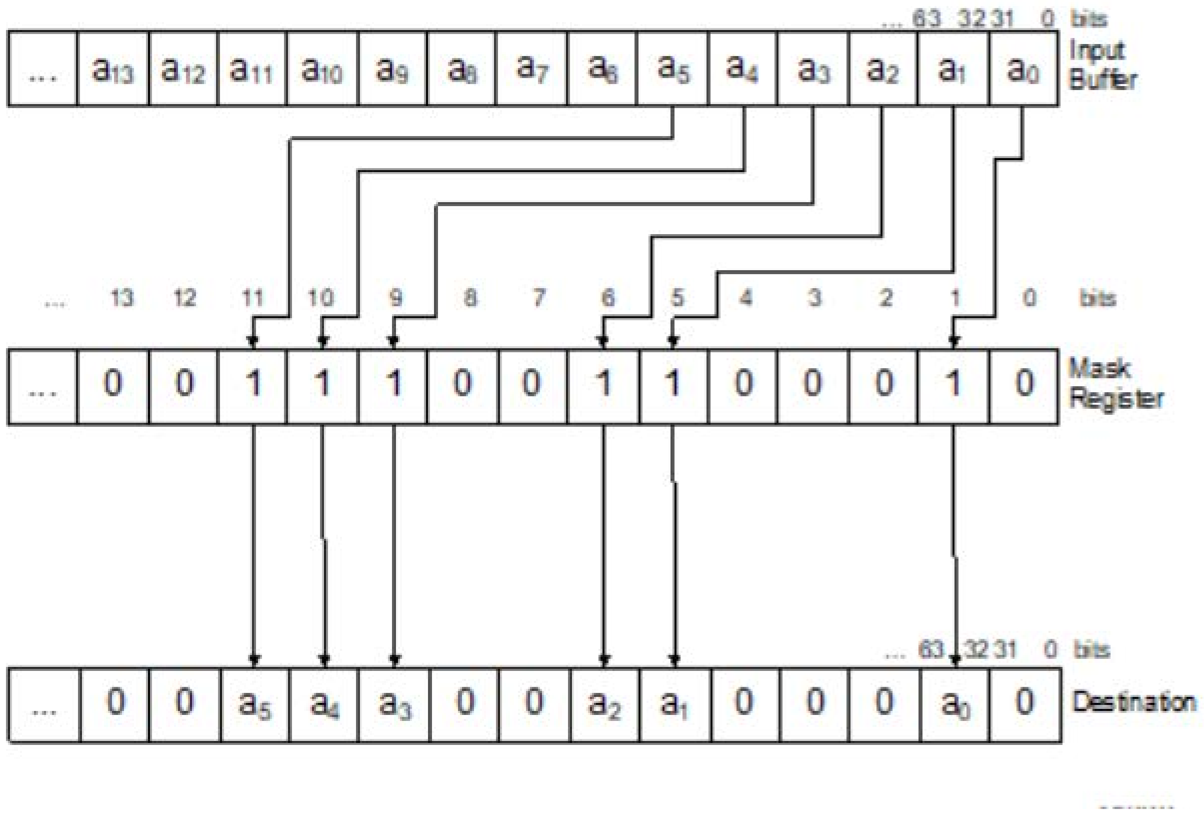

... 63 32 31 0 bits
Input Buffer
... a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
... 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bits
Mask Register
... 0 0 1 1 1 0 0 1 1 0 0 0 1 0
... 63 32 31 0 bits
Destination
... 0 0 a5 a4 a3 0 0 a2 a1 0 0 0 a0 0

Figure 20-22. Data Expand Operation

Decompression code for 16-byte elements can be designed in the same way.

For the best performance, apply the following optimizations:

- **Interleaving:** Fine-grained interleaving of decompression code and matrix multiplication to overlap Intel AVX-512 decompression with Intel AMX computation.
- **Decompress Early:** Prepare the decompressed buffer before immediate Intel AMX use to avoid store forwarding issues.
- **Buffer Reuse:** Decompressing the full sparse matrix could overflow the CPU caches. For best cache reuse, it is recommended to have a decompressed buffer that can hold two decompressed panels of the sparse matrix. While matrix is multiplying with one panel, decompress the next panel for the subsequent iteration. In the subsequent iteration, decompress the next panel into the first half of the decompressed buffer that is no longer used, and so on.
- **Decompress Once:** Coordinate the matrix multiplication blocking and loop structure with the decompression scheme to minimize the number of times the same portion of the sparse matrix is decompressed. For example, if the B-matrix is sparse, traversing the entire vertical M-dimension will compress every vertical panel of the B-matrix only once.

20.18 TILECONFIG/TILERELEASE, CORE C-STATE, AND COMPILER ABI

For a function to use tile registers, it needs to configure them. For the LDTILECFG instruction definition, see Section 20.2. LDTILECFG creates an Intel AMX state which is kept valid until the TILERELEASE instruction is issued. TILERELEASE resets the Intel AMX state back to INIT. When the Intel AMX state is valid, and the OS issues the MWAIT instruction trying to move the physical processor, it executes on to Core C6 State. The 4th Generation Intel® Xeon® Scalable processor based on the Sapphire Rapids microarchitecture will not enter Core C6 even if the sibling logical processor is idle. This is because it lacks the dedicated backing store to keep the Intel AMX state until waking up. The Core C-State is demoted to C1 instead.

This is not an issue in Linux and Windows, where only the idle process issues the MWAIT instruction. The Idle Process in both operating systems does not use the Intel AMX ISA, so its Intel AMX tile state is always invalid (INIT). If still valid, the Intel AMX tile state will have previously been saved in an OS-defined area in memory while context-switching between a thread that uses Intel AMX and the Idle Process thread.

20.18.1 ABI

The tile data registers (tmm0 – tmm7) are volatile. Their contents are passed back and forth between functions through memory. No tile register is saved and restored by the callee. Tile configuration is also volatile. The compiler saves and restores tile configuration and tile register contents if the register(s) need to live across the function call. The compiler eliminates the save instruction because its content remains the same on the stack. The compiler reuses the configured content saved on the stack before the call. All functions need to configure the tile registers themselves; however, tile registers may not be configured across functions.

Please download the System V Application Binary Interface: Intel386 Architecture Processor Supplement, Version1.0.

20.18.2 INTRINSICS

Example 20-28. Identification of Tile Shape Using Parameter m, n, k

```
typedef int _tile1024i __attribute__((__vector_size__(1024), __aligned__(64)));
_tile1024i _tile_loadd_internal(unsigned short m, unsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_loaddt1_internal(unsigned short m, uunsigned short n, const void*base, __SIZE_TYPE__ stride);
_tile1024i _tile_dpbssd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbsud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbusd_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbuud_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
_tile1024i _tile_dpbf16ps_internal(unsigned short m, unsigned short n, unsigned short k, _tile1024i dst, _tile1024i
src1, _tile1024i src2);
void_tile_stored_internal(unsigned short m, unsigned short n, void*base, __SIZE_TYPE__ stride, _tile1024i tile);
```

The parameter m, n, k identifies the shape of the tile.

20.18.3 USER INTERFACE

Example 20-29. Intel® AMX Intrinsics Header File

```
/* 1 of 2 */
typedef struct __tile1024i_str {
  const unsigned short row;
const unsigned short col;
  _tile1024i tile;
} __tile1024i;

/// Load tile rows from memory specified by "base" address and "stride" into
/// destination tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILELOADD </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_loadd(__tile1024i *dst, const void *base, __SIZE_TYPE__ stride);
/// Load tile rows from memory specified by "base" address and "stride" into
/// destination tile "dst". This intrinsic provides a hint to the implementation
/// that the data will likely not be reused in the near future and the data
/// caching can be optimized accordingly.
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILELOADDT1 </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be loaded in memory.
void __tile_stream_loadd(__tile1024i* dst, const void* base, __SIZE_TYPE__ stride);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with
/// corresponding signed 8-bit integers in src1, producing 4 intermediate 32-bit
/// results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
```

```
/* 2 of 3 */
/// This intrinsic corresponds to the <c> TDPBSSD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbssd(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of signed 8-bit integers in src0 with
/// corresponding unsigned 8-bit integers in src1, producing 4 intermediate
/// 32-bit results. Sum these 4 results with the corresponding 32-bit integer
/// in "dst", and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBSUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbsud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Compute dot-product of bytes in tiles with a source/destination accumulator.
/// Multiply groups of 4 adjacent pairs of unsigned 8-bit integers in src0 with
/// corresponding signed 8-bit integers in src1, producing 4 intermediate 32-bit
/// results. Sum these 4 results with the corresponding 32-bit integer in "dst",
/// and store the 32-bit result back to tile "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBUUD </c> instruction.
///
/// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbuud(__tile1024i *dst, __tile1024i src1, __tile1024i src2);
/// Zero the tile specified by "dst".
///
/// \headerfile <immintrin.h>
///
```

```
/* 2of 2 */
/// This intrinsic corresponds to the <c> TILEZERO </c> instruction.
///
/// \param dst
///    The destination tile to be zero. Max size is 1024 Bytes.
void __tile_zero(__tile1024i* dst);
/// Compute dot-product of BF16 (16-bit) floating-point pairs in tiles src0 and
/// src1, accumulating the intermediate single-precision (32-bit) floating-point
/// elements with elements in "dst", and store the 32-bit result back to tile
/// "dst".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TDPBF16PS </c> instruction.
////// \param dst
///    The destination tile. Max size is 1024 Bytes.
/// \param src0
///    The 1st source tile. Max size is 1024 Bytes.
/// \param src1
///    The 2nd source tile. Max size is 1024 Bytes.
void __tile_dpbf16ps(__tile1024i* dst, __tile1024i src0, __tile1024i src1);
/// Store the tile specified by "src" to memory specified by "base" address and
/// "stride".
///
/// \headerfile <immintrin.h>
///
/// This intrinsic corresponds to the <c> TILESTORED </c> instruction.
///
/// \param dst
///    A destination tile. Max size is 1024 Bytes.
/// \param base
///    A pointer to base address.
/// \param stride
///    The stride between the rows' data to be stored in memory.
void __tile_stored(void *base, __SIZE_TYPE__ stride, __tile1024i src);
```

20.18.4 INTEL® AMX INTRINSICS EXAMPLE

In Example 20-30, function foo is called in line 18, and the tile variable 'a' written in line 17 needs to live up to line 21 across the function call. The compiler needs to save the tile data register allocated to 'a' before calling foo, then restore the tile configure register and tile data registers after calling foo. Lines 39, 42, and 46 in Example 20-31 are the save/restore code. Since the configure register doesn't change, the configure register in the stack does not require saving.

Example 20-30. Intel® AMX Intrinsics Usage

```
1 #include <immintrin.h>
2
3 char buf[1024];
4 #define STRIDE 32
5
6 int count = 0;
7 __attribute__((noinline))
8 void foo() {
9   count++;
10 }
11
12 void test_api(int cond, unsigned short row, unsigned short col) {
13   __tile1024i a = {row, col};
14   __tile1024i b = {row, col};
15   __tile1024i c = {row, col};
16
17   __tile_loadd(&a, buf, STRIDE);
18   foo();
19   __tile_loadd(&b, buf, STRIDE);
20   __tile_loadd(&c, buf, STRIDE);
21   __tile_dpbssd(&c, a, b);
22   __tile_stored(buf, STRIDE, c);
23 }
```

```
clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables.
```

Notice the ldtilecfg instruction at the beginning of the function (line 34 in Example 20-31), which sets the Intel AMX registers configuration within the CPU and the TileRelease instruction towards the end of the function. This placement ensures that the Intel AMX state is initialized, thus avoiding the expensive Intel AMX state save/restore in case of a software thread context-switch outside of the Intel AMX function.

Example 20-31. Compiler-Generated Assembly-Level Code from Example 20-30

```
test_api:                           # @test_api
# %bb.0:                                # %entry
        pushq   %rbp
        pushq   %r15
        pushq   %r14
        pushq   %rbx
        subq    $1096, %rsp                 # imm = 0x448
        movl    %edx, %ebx
        movl    %esi, %ebp
        vpxord  %zmm0, %zmm0, %zmm0
        vmovdqu64       %zmm0, (%rsp)
        movb    $1, (%rsp)
        movw    %bx, 20(%rsp)
        movb    %bpl, 50(%rsp)
        movw    %bx, 18(%rsp)
        movb    %bpl, 49(%rsp)
        movw    %bx, 16(%rsp)
        movb    %bpl, 48(%rsp)
        ldtilecfg       (%rsp)
        movl    $buf, %r14d
        movl    $32, %r15d
        tileloadd       (%r14,%r15), %tmm0
        movabsq $64, %rax
        tilestored      %tmm0, 64(%rsp,%rax)    # 1024-byte Folded Spill
        vzeroupper
        callq   foo
        ldtilecfg       (%rsp)
        tileloadd       (%r14,%r15), %tmm0
        tileloadd       (%r14,%r15), %tmm1
        movabsq $64, %rax
        tileloadd       64(%rsp,%rax), %tmm2    # 1024-byte Folded Reload
        tdpbssd %tmm0, %tmm2, %tmm1
        tilestored      %tmm1, (%r14,%r15)
        addq    $1096, %rsp                 # imm = 0x448
        popq    %rbx
        popq    %r14
        popq    %r15
        popq    %rbp
        tilerelease
        retq
```

20.18.5 COMPILATION OPTION

The save/restore is sometimes unnecessary, e.g., when foo does not clobber any tile register. To avoid unnecessary save/restore, compile with "-mllvm -enable-ipra", which does an IPO analysis to get the information on what physical registers are clobbered during the function call. Example 20-32 shows no tile register save/restore across calling foo.

clang -O2 -S amx-across-func.c -mamx-int8 -mavx512f -fno-asynchronous-unwind-tables -mllvm -enable-ipra

Example 20-32. Compiler-Generated Assembly-Level Code Where Tile Register Save/Restore is Optimized Away

```
        .type   test_api,@function
test_api:                        # @test_api
# %bb.0:                         # %entry
        subq    $72, %rsp
        vpxord  %zmm0, %zmm0, %zmm0
        vmovdqu64       %zmm0, 8(%rsp)
        movb    $1, 8(%rsp)
        movw    %dx, 28(%rsp)
        movb    %sil, 58(%rsp)
        movw    %dx, 26(%rsp)
        movb    %sil, 57(%rsp)
        movw    %dx, 24(%rsp)
        movb    %sil, 56(%rsp)
        ldtilecfg       8(%rsp)
        movl    $buf, %eax
        movl    $32, %ecx
        tileloadd       (%rax,%rcx), %tmm0
        callq   foo
        tileloadd       (%rax,%rcx), %tmm1
        tileloadd       (%rax,%rcx), %tmm2
        tdpbssd %tmm1, %tmm0, %tmm2
        tilestored      %tmm2, (%rax,%rcx)
        addq    $72, %rsp
        tilerelease
        vzeroupper
        retq
.Lfunc_end1:
        .size   test_api, .Lfunc_end1-test_api
```

20.19 INTEL® AMX STATE MANAGEMENT

Intel AMX is XSAVE supported, meaning that it defines processor registers that can be saved and restored using instructions of the XSAVE feature set. Intel AMX is also XSAVE enabled, meaning that system software must enable it before it can be used.

The XSAVE feature set operates on state components, each a discrete set of processor registers (or parts of registers). Intel AMX is associated with two state components, XTILECFG and XTILEDATA. The XSAVE feature set organizes state components using state-component bitmaps. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. Intel AMX defines bits 18:17 for its state components (collectively, these are called AMX state):

- State component 17 is used for the 64-byte TILECFG register (XTILECFG state).
- State component 18 is used for the 8192 bytes of tile data (XTILEDATA state).

These are both user-state components, meaning the entire XSAVE feature set can manage them. In addition, it implies that setting bits 18:17 of extended control register XCR0 by system software enables Intel AMX. If those bits are zero, an Intel AMX instruction execution results in an invalid-opcode exception (#UD).

About the XSAVE feature set's INIT optimization, the Intel AMX state is in its initial configuration if the TILECFG register is zero and all tile data are zero.

Enumeration and feature-enabling documentation can be found in Section 20.2.

An execution of XRSTOR or XRSTORS initializes the TILECFG register (resulting in TILES_CONFIGURED = 0) in response to an attempt to load it with an illegal value. Moreover, an execution of XRSTOR or XRSTORS that is not directed to load XTILEDATA leaves it unmodified, even if the execution is loading XTILECFG.

It is highly recommended that developers execute TILERELEASE to initialize the tiles at the end of the Intel AMX instructions code region. More on this is in Section 20.18.

If the system software does not initialize the Intel AMX state first (by executing TILERELEASE, for example), it may disable Intel AMX by clearing XCR0[18:17], by clearing CR4.OSXSAVE, or by setting IA32_XFD[18].

20.19.1 EXTENDED FEATURE DISABLE (XFD)

The XTILEDATA state component size is 8 KBytes, and an operating system may, by default, prefer not to allocate memory for the XTILEDATA state for every user thread. An operating system that enables Intel AMX might select a fault when user threads use the feature. That way, it can allocate a large enough state save area only for the user threads using the feature. An operating system may offer an API for the user threads to declare their intention to use Intel AMX and allow the OS to preallocate the state and avoid an exception when Intel AMX is used for the first time.

See Linux API and Windows API for more details.

Extended feature disable (XFD) is added to the XSAVE feature set to support such usage. See the Intel® AMX Architecture Definition for XFD documentation.

20.19.2 ALTERNATE SIGNAL HANDLER STACK IN LINUX OPERATING SYSTEM

When programs use an alternate signal handler stack, the stack size should be adjusted to accommodate the additional Intel AMX state. See Using XSTATE Features in User-Space Applications for more details.

20.20 USING INTEL® AMX TO EMULATE HIGHER PRECISION GEMMS

Intel AMX/TMUL has instructions that enable matrix-matrix operations such as multiplication on small precision elements. This section considers how to use the low-precision Intel AMX instructions to approximate the answers to matrix-matrix multiplication of higher-precision terms. Even if low-precision inputs are Bfloat16 or Integer8, one can still combine the transforms to approximate matrix-matrix multiplication in higher precisions.

Pay attention to the exponent range and mantissa bits when approximating higher precisions. There are IEEE-754 double precision numbers (FP64) that aren't representable as single precision (FP32) or lower precisions. These are typically range-based issues in the exponent bits. FP64 has more exponent bits than FP32. However, scaling factors can overcome most range-based problems. If A is a matrix of FP64 values, then A (as a sum of Bfloat16 matrices) cannot generally be represented. Scaling factors can, however, be used to get around most issues. The A-matrix as s1*A1 + s2*A2 + … + sn*An can be written where each matrix A_i is lower precision, and each si is a constant scaling factor.

For Bfloat16 decomposition of FP32, consider the following:

- Let A be a matrix of FP32 values.
- Let A1 = bfloat16(A), a matrix containing RNE-rounded Bfloat16 conversions of A.
- Let A2 = bfloat16(A – fp32(A1)).
- Let A3 = bfloat16(A – fp32(A1) – fp32(A2)).
- Now A is approximately A1 + A2 + A3.

Once one has written two matrices as a sum of lower precision matrices, one can run AMX/TMUL on the product to approximate the higher precision. But to do this effectively, one needs to have higher precision accumulation. There are tricks in the literature for doing higher precision all in a lower precision, such as works on so-called double-double arithmetic. Still, these tend to vary too much from standard matrix-matrix multiplication to be helpful with TMUL. In the case of Bfloat16, having 32-bit accumulation in the product allows one to use Bfloat16 to approximate FP32 accuracy.

Therefore, if A = s1*A1 + s2*A2 + s3*A3, and B = t1*B1 + t2*B2 + t3*B3, then A*B can be computed using AMX/TMUL on the projects Ai*Bj for 1<=i,j<=3, assuming scaling is done carefully to avoid denormals. Assuming FP32 accumulation, the FP32 approximation of A*B can be made by writing out these lower precision multiplies. Scaling factors can be chosen to avoid denormals at times, but they can also be picked in a way that simplifies further steps in the algorithm. In some cases, scaling factors can be chosen to be a power of two, for instance, without significantly reducing the accuracy of the resulting matrix-matrix multiply.

The number of matrices for A or B are picked depending on the mantissa range to cover. If trying to emulate FP32 which has 24 bits of mantissa (including the implicit mantissa bit), it is possible with three Bfloat16 matrices (because each of the triples has 8 bits of mantissa, including the implicit bit.). Here the range is less important because Bfloat16 and FP32 have the same exponent range. Use three Bfloat16 matrices to approximate FP32 precision by BF16x3. Range issues may still come up for BF16x3 cases where A has values close to the maximum or minimum exponent for FP32, but that too can be circumvented by scaling constants. Scaling factors of 2^24 or 2^(-24) suffice to push it far enough away from the boundary to make the computation feasible again. This is dependent upon the closest end of the spectrum.

A few terms from an expansion can also be dropped. For instance, in the BF16x3 case, where there are three As and three Bs, nine products may result. That is:

A*B = (A1+A2+A3)*(B1+B2+B3) = (A1*B1) + (A1*B2 + A2*B1) + (A1*B3 + A2*B2 + A3*B1) + (A2*B3 + A3*B2) +(A3*B3).

The parentheses in the last equation are intentionally derived so that all entries in the same “bin” are put together, and there are nine entries of the form Ai*Bj. This example has five bins, each with its own set of parentheses. In the Bfloat16 case, |Ai| <= |A_i-1}| / 256. This shows the last two bins (with A2*B3,A3*B2,A3*B3) are too small to contribute significantly to the answer, which is why if there are Y terms on each side of A*B, only (Y+1)*Y/2 multiplies are required, not Y*Y multiplies. In this case, dropping the last three (also the difference between Y*Y – (Y+1)*Y/2 when Y=3.) from the nine multiplies. The last three multiplies in the last two bins have terms less than 2^(-24) as big as the first term. So, A*B can be approximated (ignoring the scaling terms for now) as the sum of the first three most significant bins: A1*B1 + (A1*B2+A2*B1)+(A1*B3+A2*B2*A3*B1). In this case, adding from the least significant bin to the most significant bin (A1*B1) is recommended.

Whenever A and B are each expanded out to Y-terms, computing only Y*(Y+1)/2 products works under the condition that each term has the same number of mantissa bits. If some terms have a different number of bits, then this guideline no longer applies. But for BF16x3, each term covers eight mantissa bits and Y=3, so six products are needed.

Regarding accuracy, the worst-case relative error for BF16x3 may be worse than FP32. However, BF16x3 tends to cover a larger mantissa range due to implicit bits, which can be more accurate in many cases. Nevertheless, accuracy is not offered by matrix-matrix multiplication. Even FP64 or FP128 can be bad for component-wise relative errors. Take A = [1, -1] and B = [1; 1]. A*B is zero. Let eps be a small perturbation to A and/or B. The solution may now be arbitrarily bad in terms of relative error. In general, assume that the same mantissa range and exponent range is covered as a given higher-precision floating point format, and the accumulation is at least as good as the higher-precision format. With such an assumption, the answer will be approximately the same as the higher-precision floating point format. It may or may not be identical. Performing the same operation in the higher precision format but changing the order of the computations could yield slightly different results. In terms of matrix-matrix multiplication, it could yield vast differences in relative error.

Things get slightly more complicated if low precision is used to approximate matrix-matrix at FP64 accuracy or FP128 precision. Here the scalars aren't just for avoiding denormals but are necessary to do the initial matrix conversion. Nevertheless, converting to an integer is recommended in this case because the FP32-rounded errors in each of the seven or fewer bins may introduce too many errors. An integer is easier to get right because there are no floating-point errors in each bin.

Conversion to Integer functions in the same way as all of the previous Bfloat16 examples. The quantization literature explains how to map floating point numbers into integers. The only difference is that these integers are further broken down into 8-bit pieces for the use of Intel AMX. Constant factors are still needed, but in this case they are primarily defined in the conversion itself.

One difficulty with quantization to integers is the notion of a shared exponent. All the numbers quantized together with shared exponents must share the same range. The assumption is that all of A shares a joint exponent range. Since this will also be true for B, each row of A and column of B can be quantized separately.

Assuming that there is Integer32 accumulation with the Integer8 multiplies, a matrix may be broken down into far more bits than required. This may significantly reduce the inaccuracy impact of picking a shared exponent. Because Integer32 arithmetic will be precise, modulo overflow/underflow concerns, then one can break up A or B into a huge number of 8-bit integer matrices, then do all the matrix-matrix work with Intel AMX, and then convert back all the results to even get accuracies up to quad-precision.

Considering an extreme case of trying to get over 100-bits of accuracy in a matrix-matrix multiply. All A-values can be quantified into 128-bit integers. The same holds true with B. Once broken down into 8-bit quantities, this will have a significant expansion like: A = s1*A1 + s2*A2 + … + s14*A14 for when attempting 112-bits of mantissa. The same can be done with B = t1*B1 + t2*B2 + … + t14*B14. A*B is potentially 14*14=196 products, but only 105 products are needed because the last few products may have scaling factors less than 2^(-112) times the most important terms. Each product term should be added separately and computing into C from the least significant bits forward.

C15 = (s1*t14)*A1*B14 + (s2*t13)*A2*B13 + … + (s14*t1)*A14*B1

C14 = (s1*t13)*A1*B13 + (s2*t12)*A2*B12 + … + (s13*t1)*A13*B1

C13 = (s1*t12)*A1*B12 + (s2*t11)*A2*B11 + … + (s12*t1)*A12*B1

…

C02 = (s1*t1)*A1*B1

Sometimes choosing scalers is possible such that all the products in a given row can be computed with the same scratch array. The converted sum of C02 gives the final product through C15, where terms like C15 should be computed first.

Writing matrix-matrix multiplies in terms of an expansion like (A1+A2+A3)*(B1+B2+B3) is referred to as "cascading GEMM." Performance will vary depending on the TMUL/Intel AMX specification, and may vary from generation to generation. Note that some computations may become bandwidth-bound. Since there is no quad floating-point precision in hardware for Intel Architecture, the above algorithm may be competitive performance-wise with other approaches like doing software double-double optimizations or software-based quad precision.

5. Updates to Appendix E

Change bars and **violet** text show changes to Appendix E of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: The Optimization of Earlier Generations of Intel 64 and IA32 Processor Architectures.

Changes to this chapter:

- This chapter has been updated to comprise Chapters 1 through 5 of the new Volume 2 of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.
- Updated capitalization of headings throughout chapter.
- Updated branding throughout chapter.
- Typo and punctuation corrections as necessary.

CHAPTER 1 HASWELL MICROARCHITECTURE

1.1 INTRODUCTION

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell microarchitecture is depicted in Figure 1-1. In general, most of the features described in Section 1.2 - Section 1.4 also apply to the Broadwell microarchitecture. Enhancements of the Broadwell microarchitecture are summarized in Section 1.6.

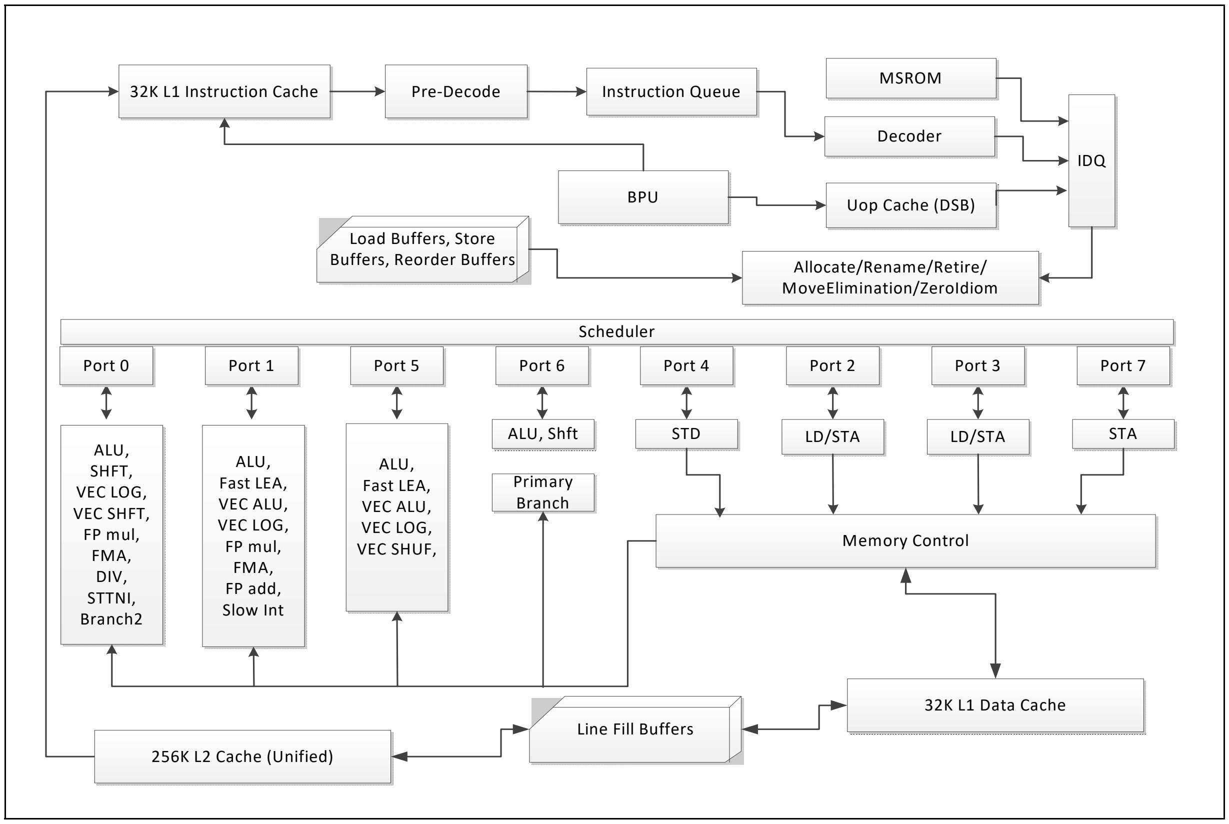

32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
IDQ
BPU
Uop Cache (DSB)
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure 1-1. CPU Core Pipeline Functionality of the Haswell Microarchitecture

The Haswell microarchitecture offers the following innovative features:

- Support for Intel® Advanced Vector Extensions 2 (Intel® AVX2), FMA.
- Support for general-purpose, new instructions to accelerate integer numeric encryption.
- Support for Intel® Transactional Synchronization Extensions (Intel® TSX).
- Each core can dispatch up to 8 micro-ops per cycle.
- 256-bit data path for memory operation, FMA, AVX floating-point and AVX2 integer execution units.
- Improved L1D and L2 cache bandwidth.
- Two FMA execution pipelines.
- Four arithmetic logical units (ALUs).
- Three store address ports.

- Two branch execution units.
- Advanced power management features for IA processor core and uncore sub-systems.
- Support for optional fourth level cache.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. An example of the system integration view of four CPU cores with uncore components is illustrated in Figure 1-2.

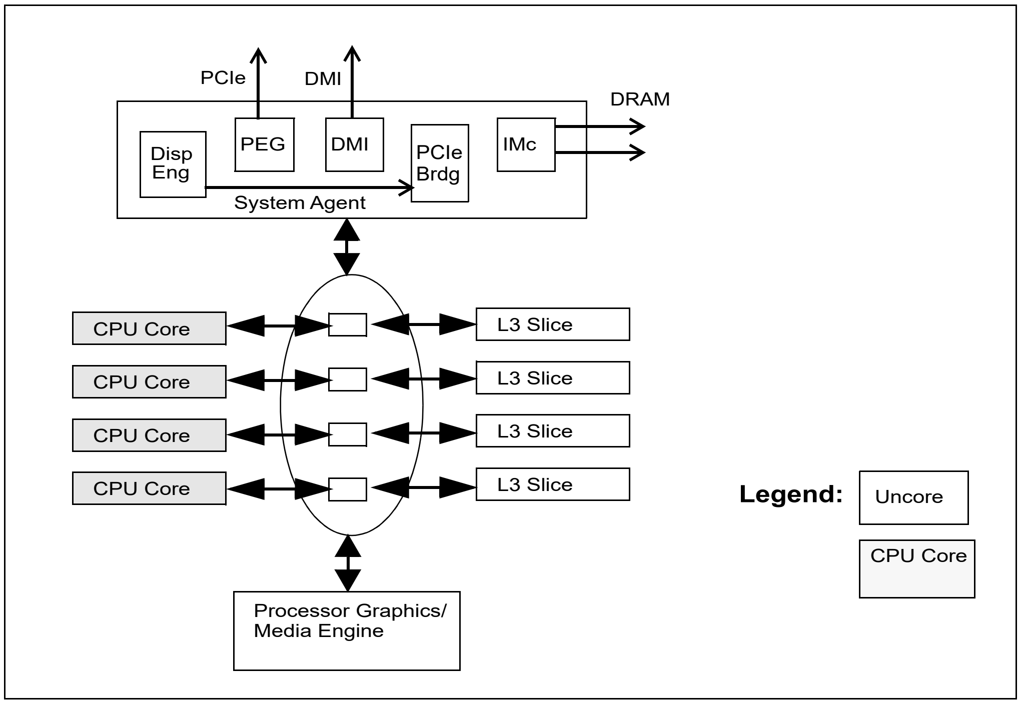


Figure 1-2. Four Core System Integration of the Haswell Microarchitecture

1.2 THE FRONT END

The front end of Haswell microarchitecture builds on that of the Sandy Bridge and Ivy Bridge microarchitectures, see Section A.2.2 and Section A.2.7. Additional enhancements in the front end include:

- The uop cache (or decoded ICache) is partitioned equally between two logical processors.
- The instruction decoders will alternate between each active logical processor. If one sibling logical processor is idle, the active logical processor will use the decoders continuously.
- The LSD in the micro-op queue (or IDQ) can detect small loops up to 56 micro-ops. The 56-entry micro-op queue is shared by two logical processors if Hyper-Threading Technology is active (Sandy Bridge microarchitecture provides duplicated 28-entry micro-op queue in each core).

1.3 THE OUT-OF-ORDER ENGINE

The key components and significant improvements to the out-of-order engine are summarized below:

Renamer: The Renamer moves micro-ops from the micro-op queue to bind to the dispatch ports in the Scheduler with execution resources. Zero-idiom, one-idiom and zero-latency register move operations are performed by the Renamer to free up the Scheduler and execution core for improved performance.

Scheduler: The Scheduler controls the dispatch of micro-ops onto the dispatch ports. There are eight dispatch ports to support the out-of-order execution core. Four of the eight ports provided execution resources for computational operations. The other 4 ports support memory operations of up to two 256-bit load and one 256-bit store operation in a cycle.

Execution Core: The scheduler can dispatch up to eight micro-ops every cycle, one on each port. Of the four ports providing computational resources, each provides an ALU, two of these execution pipes provided dedicated FMA units. With the exception of division/square-root, STTNI/AESNI units, most floating-point and integer SIMD execution units are 256-bit wide. The four dispatch ports servicing memory operations consist with two dual-use ports for load and store-address operation. Plus a dedicated 3rd store-address port and one dedicated store-data port. All memory ports can handle 256-bit memory micro-ops. Peak floating-point throughput, at 32 single-precision operations per cycle and 16 double-precision operations per cycle using FMA, is twice that of Sandy Bridge microarchitecture.

The out-of-order engine can handle 192 uops in flight compared to 168 in Sandy Bridge microarchitecture.

1.3.1 Execution Engine

Table 1-1 summarizes which operations can be dispatched on which port.

Table 1-1. Dispatch Port and Execution Stacks of the Haswell Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Shift	ALU, Fast LEA, BM	Load_Addr, Store_addr	Store_data	ALU, Fast LEA, BM	ALU, Shift, JEU	Store_addr, Simple_AGU
SIMD_Log, SIMD misc, SIMD_Shifts	SIMD_ALU, SIMD_Log			SIMD_ALU, SIMD_Log,		
FMA/FP_mul, Divide	FMA/FP_mul, FP_add			Shuffle		
2nd_Jeu	slow_int,			FP mov, AES		

Table 1-2 lists execution units and common representative instructions that rely on these units. Table 1-2 also includes some instructions that are available only on processors based on the Broadwell microarchitecture.

Table 1-2. Haswell Microarchitecture Execution Units and Representative Instructions

Execution Unit	# of Ports	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa
SHFT	2	sal, shl, rol, adc, sarx, (adcx, adox)[1] etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
SIMD Log	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)blendp*, vpblendd
SIMD_Shft	1	(v)psl*, (v)psr*
SIMD ALU	2	(v)padd*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)pcmpgt*
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)pblendw
SIMD Misc	1	(v)pmul*, (v)pmadd*, STTNI, (v)pclmulqdq, (v)psadw, (v)pcmpgtq, vpsllvd, (v)bendv*, (v)plendw,
FP Add	1	(v)addp*, (v)cmpp*, (v)max*, (v)min*,
FP Mov	1	(v)movap*, (v)movup*, (v)movsd/ss, (v)movd gpr, (v)andp*, (v)orp*
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Only available in processors based on the Broadwell microarchitecture and support CPUID ADX feature flag.

The reservation station (RS) is expanded to 60 entries deep (compared to 54 entries in Sandy Bridge microarchitecture). It can dispatch up to eight micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, arranged in several stacks to handle specific data types or granularity of data.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. Table A-25 describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

Table 1-3. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

From/To	INT	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
INT		• micro-op (port 5) • micro-op (port 6) + 1 cycle	• micro-op (port 5) • micro-op (port 6) + 1 cycle	micro-op (port 5) + 3 cycle delay
SSE-INT/ AVX-INT	micro-op (port 1)		1 cycle delay	
SSE-FP/ AVX-FP_LOW	micro-op (port 1)	1 cycle delay		micro-op (port 5) + 1cycle delay
X87/ AVX-FP_High	micro-op (port 1) + 3 cycle delay		micro-op (port 5) + 1cycle delay	
Load		1 cycle delay	1 cycle delay	2 cycle delay

1.4 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy is similar to prior generations, including an instruction cache, a first-level data cache and a second-level unified cache in each core, and a 3rd-level unified cache with size dependent on specific product configuration. The 3rd-level cache is organized as multiple cache slices, the size of each slice may depend on product configurations, connected by a ring interconnect. The exact details of the cache topology is reported by CPUID leaf 4. The 3rd level cache resides in the "uncore" sub-system that is shared by all the processor cores. In some product configurations, a fourth level cache is also supported. Table A-23 provides more details of the cache hierarchy.

Table 1-4. Cache Parameters of the Haswell Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Throughput (clocks)	Peak Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	0.5[2]	64 (Load) + 32 (Store)	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/8	64	11 cycle	Varies	64	Writeback
Third Level (Shared L3)	Varies	64	~34	Varies		Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors. L3 latency can vary due to clock ratios between the processor core and uncore.
2. First level data cache supports two load micro-ops each cycle; each micro-op can fetch up to 32-bytes of data.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 1-5. TLB Parameters of the Haswell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1024	8	fixed

1.4.1 Load and Store Operation Enhancements

The L1 data cache can handle two 256-bit load and one 256-bit store operations each cycle. The unified L2 can service one cache line (64 bytes) each cycle. Additionally, there are 72 load buffers and 42 store buffers available to support micro-ops execution in-flight.

1.4.2 Unlamination

Some micro-fused instructions cannot be allocated as a single uop, and therefore they break into two uops in the micro-op queue. The process of breaking a fused instruction into its uops is called unlamination.

Unlamination will take place if the number of fused instruction sources is greater than three.

Instruction sources in the context of unlamination are considered to be one of the following: memory address base, memory address index, source register, destination register (including flags), or a source and destination register.

A memory operand in the context of unlamination can have up to two sources. A memory address in the x86 instruction set is constructed from: base + index*scale + displacement.

Only a base and an index are counted as instruction sources. Notice that if an index exists, the base is counted as a source even if it's not present.

In addition, source and destination registers are counted as two sources; this is also true in the case where the source and destination register are the same.

The following table shows examples of micro-fused instructions and details of unlamination.

Table A-6. Components of the Front End

Instruction Example	Source	Destination	Source and Destination	Index	Base	Number of Sources[1]	Unlaminated
mulss xmm1, [4*rax+100]	-	-	xmm1	rax	0	3	no
vmulss xmm1, xmm1, [rax +100]	xmm1	xmm1	-	-	rax	3	no
vmulss xmm1, xmm1, [4*rax+100]	xmm1	xmm1	-	rax	0	4	yes
cmp rax, [rbx+4*rax+4]	rax	flags	-	rax	rbx	4	yes
cmp rax, [rbx+4]	rax	flags	-	-	rbx	3	no

NOTES:

1. Recommendation: to avoid unlamination, keep the number of micro-fused instruction sources under 4.

1.5 HASWELL-E MICROARCHITECTURE

Intel processors based on the Haswell-E microarchitecture comprises the same processor cores as described in the Haswell microarchitecture, but provides more advanced uncore and integrated I/O capabilities. Processors based on the Haswell-E microarchitecture support platforms with multiple sockets.

The Haswell-E microarchitecture supports versatile processor architectures and platform configurations for scalability and high performance. Some of capabilities provided by the uncore and integrated I/O subsystem of the Haswell-E microarchitecture include:

- Support for multiple Intel QPI interconnects in multi-socket configurations.
- Up to two integrated memory controllers per physical processor.
- Up to 40 lanes of Intel® PCI Express* 3.0 links per physical processor.
- Up to 18 processor cores connected by two ring interconnects to the L3 in each physical processor.

An example of a possible 12-core processor implementation using the Haswell-E microarchitecture is illustrated in Figure 1-3. The capabilities of the uncore and integrated I/O sub-system vary across the processor family implementing the Haswell-E microarchitecture. For details, please consult the data sheets of respective Intel Xeon E5 v3 processors.

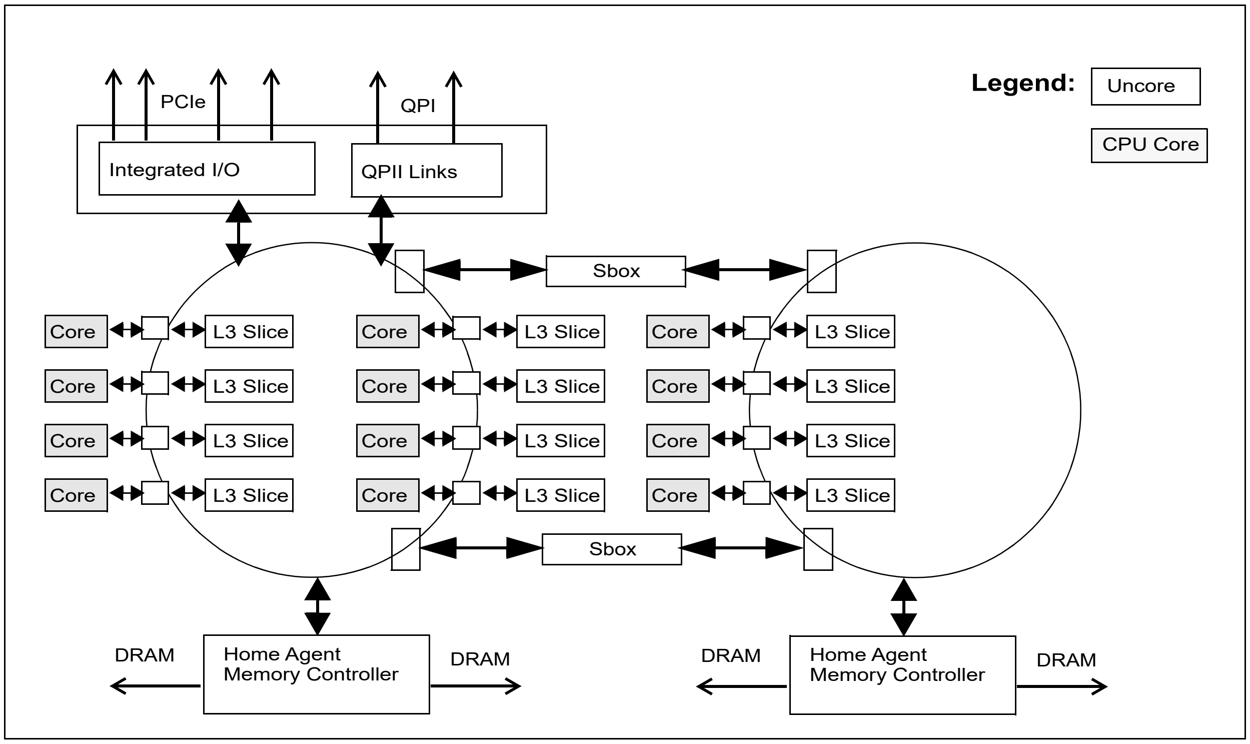


Figure 1-3. An Example of the Haswell-E Microarchitecture Supporting 12 Processor Cores

1.6 BROADWELL MICROARCHITECTURE

Intel Core M processors are based on the Broadwell microarchitecture. The Broadwell microarchitecture builds from the Haswell microarchitecture and provides several enhancements. This section covers enhanced features of the Broadwell microarchitecture.

- Floating-point multiply instruction latency is improved from five cycles in prior generation to three cycles in the Broadwell microarchitecture. This applies to Intel AVX, Intel SSE and FP instruction sets.
- The throughput of gather instructions has been improved significantly, see Table D-5.
- The PCLMULQDQ instruction implementation is a single uop in the Broadwell microarchitecture with improved latency and throughput.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 1-7. TLB Parameters of the Broadwell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2MB pages	1536	6	fixed
Second Level	1GB pages	16	4	fixed

CHAPTER 2
SANDY BRIDGE MICROARCHITECTURE

Sandy Bridge microarchitecture builds on the successes of Intel® Core™ microarchitecture and Nehalem microarchitecture. It offers the following innovative features:

- Intel Advanced Vector Extensions (Intel AVX)
  - 256-bit floating-point instruction set extensions to the 128-bit Intel SSE, providing up to 2X performance benefits relative to 128-bit code.
  - Non-destructive destination encoding offers more flexible coding techniques.
  - Supports flexible migration and co-existence between 256-bit AVX code, 128-bit AVX code and legacy 128-bit SSE code.
- Enhanced front end and execution engine
  - New decoded ICache component that improves front end bandwidth and reduces branch misprediction penalty.
  - Advanced branch prediction.
  - Additional macro-fusion support.
  - Larger dynamic execution window.
  - Multi-precision integer arithmetic enhancements (ADC/SBB, MUL/IMUL).
  - LEA bandwidth improvement.
  - Reduction of general execution stalls (read ports, writeback conflicts, bypass latency, partial stalls).
  - Fast floating-point exception handling.
  - XSAVE/XRSTORE performance improvements and XSAVEOPT new instruction.
- Cache hierarchy improvements for wider data path
  - Doubling of bandwidth enabled by two symmetric ports for memory operation.
  - Simultaneous handling of more in-flight loads and stores enabled by increased buffers.
  - Internal bandwidth of two loads and one store each cycle.
  - Improved prefetching.
  - High bandwidth low latency LLC architecture.
  - High bandwidth ring architecture of on-die interconnect.
- System-on-a-chip support
  - Integrated graphics and media engine in second generation Intel Core processors.
  - Integrated Intel® PCIe controller.
  - Integrated memory controller.
- Next generation Intel Turbo Boost Technology
  - Leverage TDP headroom to boost performance of CPU cores and integrated graphic unit.

2.1 SANDY BRIDGE MICROARCHITECTURE PIPELINE OVERVIEW

Figure 2-1 depicts the pipeline and major components of a processor core that’s based on Sandy Bridge microarchitecture. The pipeline consists of:

- An in-order issue front end that fetches instructions and decodes them into micro-ops (micro-operations). The front end feeds the next pipeline stages with a continuous stream of micro-ops from the most likely path that the program will execute.
- An out-of-order, superscalar execution engine that dispatches up to six micro-ops to execution, per cycle. The allocate/rename block reorders micro-ops to "dataflow" order so they can execute as soon as their sources are ready and execution resources are available.
- An in-order retirement unit that ensures that the results of execution of the micro-ops, including any exceptions they may have encountered, are visible according to the original program order.

The flow of an instruction in the pipeline can be summarized in the following progression:

1. The Branch Prediction Unit chooses the next block of code to execute from the program. The processor searches for the code in the following resources, in this order:
   a. Decoded ICache.
   b. Instruction Cache, via activating the legacy decode pipeline.
   c. L2 cache, last level cache (LLC) and memory, as necessary.

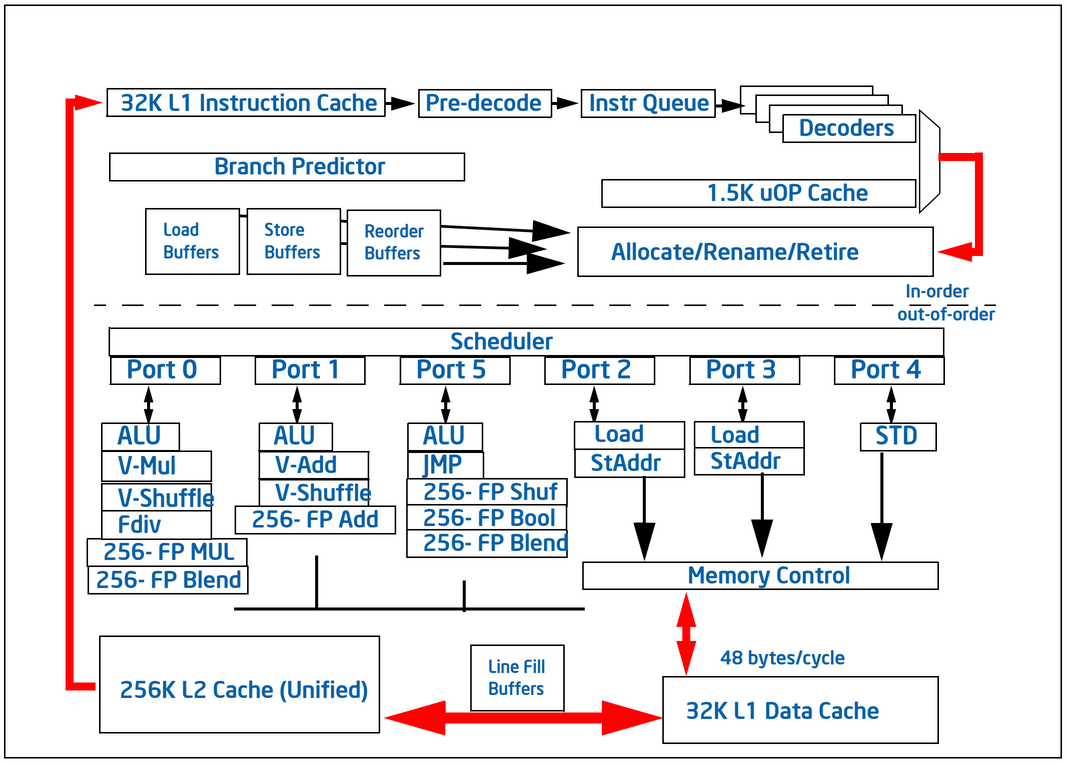


Figure 2-1. Sandy Bridge Microarchitecture Pipeline Functionality

2. The micro-ops corresponding to this code are sent to the Rename/retirement block. They enter into the scheduler in program order, but execute and are de-allocated from the scheduler according to data-flow order. For simultaneously ready micro-ops, FIFO ordering is nearly always maintained.

   Micro-op execution is executed using execution resources arranged in three stacks. The execution units in each stack are associated with the data type of the instruction.

   Branch mispredictions are signaled at branch execution. It re-steers the front end which delivers micro-ops from the correct path. The processor can overlap work preceding the branch misprediction with work from the following corrected path.

3. Memory operations are managed and reordered to achieve parallelism and maximum performance. Misses to the L1 data cache go to the L2 cache. The data cache is non-blocking and can handle multiple simultaneous misses.
4. Exceptions (Faults, Traps) are signaled at retirement (or attempted retirement) of the faulting instruction.

Each processor core based on Sandy Bridge microarchitecture can support two logical processor if Intel® Hyper-Threading Technology (Intel® HT) is enabled.

2.1.1 The Front End

This section describes the key characteristics of the front end. Table B-1 lists the components of the front end, their functions, and the problems they address.

Table 2-1. Components of the Front End of Sandy Bridge Microarchitecture

Component	Functions	Performance Challenges
Instruction Cache	32-Kbyte backing store of instruction bytes	Fast access to hot code instruction bytes
Legacy Decode Pipeline	Decode instructions to micro-ops, delivered to the micro-op queue and the Decoded ICache.	Provides the same decode latency and bandwidth as prior Intel processors. Decoded ICache warm-up
Decoded ICache	Provide stream of micro-ops to the micro-op queue.	Provides higher micro-op bandwidth at lower latency and lower power than the legacy decode pipeline
MSROM	Complex instruction micro-op flow store, accessible from both Legacy Decode Pipeline and Decoded ICache	
Branch Prediction Unit (BPU)	Determine next block of code to be executed and drive lookup of Decoded ICache and legacy decode pipelines.	Improves performance and energy efficiency through reduced branch mispredictions.
Micro-op queue	Queues micro-ops from the Decoded ICache and the legacy decode pipeline.	Hide front end bubbles; provide execution micro-ops at a constant rate.

2.1.1.1 Legacy Decode Pipeline

The Legacy Decode Pipeline comprises the instruction translation lookaside buffer (ITLB), the instruction cache (ICache), instruction predecode, and instruction decode units.

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB and into the instruction cache. The instruction cache can deliver every cycle 16 bytes to the instruction pre-decoder. Table B-1 compares the ICache and ITLB with prior generation.

Table 2-2. ICache and ITLB of Sandy Bridge Microarchitecture

Component	Sandy Bridge Microarchitecture	Nehalem Microarchitecture
ICache Size	32-Kbyte	32-Kbyte
ICache Ways	8	4
ITLB 4K page entries	128	128
ITLB large page (2M or 4M) entries	8	7

Upon ITLB miss there is a lookup to the Second level TLB (STLB) that is common to the DTLB and the ITLB. The penalty of an ITLB miss and a STLB hit is seven cycles.

Instruction PreDecode

The predecode unit accepts the 16 bytes from the instruction cache and determines the length of the instructions.

The following length changing prefixes (LCPs) imply instruction length that is different from the default length of instructions. Therefore they cause an additional penalty of three cycles per LCP during length decoding. Previous processors incur a six-cycle penalty for each 16-byte chunk that has one or more LCPs in it. Since usually there is no more than one LCP in a 16-byte chunk, in most cases, Sandy Bridge microarchitecture introduces an improvement over previous processors.

- Operand Size Override (66H) preceding an instruction with a word/double immediate data. This prefix might appear when the code uses 16 bit data types, unicode processing, and image processing.
- Address Size Override (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. This prefix may appear in boot code sequences.
- The REX prefix (4xh) in the Intel® 64 instruction set can change the size of two classes of instructions: MOV offset and MOV immediate. Despite this capability, it does not cause an LCP penalty and hence is not considered an LCP.

Instruction Decode

There are four decoding units that decode instruction into micro-ops. The first can decode all IA-32 and Intel 64 instructions up to four micro-ops in size. The remaining three decoding units handle single-micro-op instructions. All four decoding units support the common cases of single micro-op flows including micro-fusion and macro-fusion.

Micro-ops emitted by the decoders are directed to the micro-op queue and to the Decoded ICache. Instructions longer than four micro-ops generate their micro-ops from the MSROM. The MSROM bandwidth is four micro-ops per cycle. Instructions whose micro-ops come from the MSROM can start from either the legacy decode pipeline or from the Decoded ICache.

MicroFusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core as many times as it would if it were not micro-fused.

Micro-fusion enables you to use memory-to-register operations, also known as the complex instruction set computer (CISC) instruction set, to express the actual program operation without worrying about a loss of decode bandwidth. Micro-fusion improves instruction bandwidth delivered from decode to retirement and saves power.

Coding an instruction sequence by using single-uop instructions will increases the code size, which can decrease fetch bandwidth from the legacy pipeline.

The following are examples of micro-fused micro-ops that can be handled by all decoders.

- All stores to memory, including store immediate. Stores execute internally as two separate functions, store-address and store-data.
- All instructions that combine load and computation operations (load+op), for example:
    - ADDPS XMM9, OWORD PTR [RSP+40]
    - FADD DOUBLE PTR [RDI+RSI*8]
    - XOR RAX, QWORD PTR [RBP+32]
- All instructions of the form "load and jump," for example:
    - JMP [RDI+200]
    - RET
- CMP and TEST with immediate operand and memory

An instruction with RIP relative addressing is not micro-fused in the following cases:

- An additional immediate is needed, for example:
    - CMP [RIP+400], 27
    - MOV [RIP+3000], 142
- The instruction is a control flow instruction with an indirect target specified using RIP-relative addressing, for example:
    - JMP [RIP+5000000]

In these cases, an instruction that can not be micro-fused will require decoder 0 to issue two micro-ops, resulting in a slight loss of decode bandwidth.

In 64-bit code, the usage of RIP Relative addressing is common for global data. Since there is no micro-fusion in these cases, performance may be reduced when porting 32-bit code to 64-bit code.

Macro-Fusion

Macro-fusion merges two instructions into a single micro-op. In Intel Core microarchitecture, this hardware optimization is limited to specific conditions specific to the first and second of the macro-fusable instruction pair.

- The first instruction of the macro-fused pair modifies the flags. The following instructions can be macro-fused:
    - — In Nehalem microarchitecture: CMP, TEST.
    - — In Sandy Bridge microarchitecture: CMP, TEST, ADD, SUB, AND, INC, DEC
    - — These instructions can fuse if
        - The first source / destination operand is a register.
        - The second source operand (if exists) is one of: immediate, register, or non RIP-relative memory.
- The second instruction of the macro-fusable pair is a conditional branch. Table 3-1 describes, for each instruction, what branches it can fuse with.

Macro fusion does not happen if the first instruction ends on byte 63 of a cache line, and the second instruction is a conditional branch that starts at byte 0 of the next cache line.

Since these pairs are common in many types of applications, macro-fusion improves performance even on non-recompiled binaries.

Each macro-fused instruction executes with a single dispatch. This reduces latency and frees execution resources. You also gain increased rename and retire bandwidth, increased virtual storage, and power savings from representing more work in fewer bits.

2.1.2 Decoded ICache

The Decoded ICache is essentially an accelerator of the legacy decode pipeline. By storing decoded instructions, the Decoded ICache enables the following features:

- Reduced latency on branch mispredictions.
- Increased micro-op delivery bandwidth to the out-of-order engine.
- Reduced front end power consumption.

The Decoded ICache caches the output of the instruction decoder. The next time the micro-ops are consumed for execution the decoded micro-ops are taken from the Decoded ICache. This enables skipping the fetch and decode stages for these micro-ops and reduces power and latency of the Front End. The Decoded ICache provides average hit rates of above 80% of the micro-ops; furthermore, "hot spots" typically have hit rates close to 100%.

Typical integer programs average less than four bytes per instruction, and the front end is able to race ahead of the back end, filling in a large window for the scheduler to find instruction level parallelism. However, for high performance code with a basic block consisting of many instructions, for example, Intel

SSE media algorithms or excessively unrolled loops, the 16 instruction bytes per cycle is occasionally a limitation. The 32-byte orientation of the Decoded ICache helps such code to avoid this limitation.

The Decoded ICache automatically improves performance of programs with temporal and spatial locality. However, to fully utilize the Decoded ICache potential, you might need to understand its internal organization.

The Decoded ICache consists of 32 sets. Each set contains eight Ways. Each Way can hold up to six micro-ops. The Decoded ICache can ideally hold up to 1536 micro-ops.

The following are some of the rules how the Decoded ICache is filled with micro-ops:

- All micro-ops in a Way represent instructions which are statically contiguous in the code and have their EIPs within the same aligned 32-byte region.
- Up to three Ways may be dedicated to the same 32-byte aligned chunk, allowing a total of 18 micro-ops to be cached per 32-byte region of the original IA program.
- A multi micro-op instruction cannot be split across Ways.
- Up to two branches are allowed per Way.
- An instruction which turns on the MSROM consumes an entire Way.
- A non-conditional branch is the last micro-op in a Way.
- Micro-fused micro-ops (load+op and stores) are kept as one micro-op.
- A pair of macro-fused instructions is kept as one micro-op.
- Instructions with 64-bit immediate require two slots to hold the immediate.

When micro-ops cannot be stored in the Decoded ICache due to these restrictions, they are delivered from the legacy decode pipeline. Once micro-ops are delivered from the legacy pipeline, fetching micro-ops from the Decoded ICache can resume only after the next branch micro-op. Frequent switches can incur a penalty.

The Decoded ICache is virtually included in the Instruction cache and ITLB. That is, any instruction with micro-ops in the Decoded ICache has its original instruction bytes present in the instruction cache. Instruction cache evictions must also be evicted from the Decoded ICache, which evicts only the necessary lines.

There are cases where the entire Decoded ICache is flushed. One reason for this can be an ITLB entry eviction. Other reasons are not usually visible to the application programmer, as they occur when important controls are changed, for example, mapping in CR3, or feature and mode enabling in CR0 and CR4. There are also cases where the Decoded ICache is disabled, for instance, when the CS base address is NOT set to zero.

2.1.3 Branch Prediction

Branch prediction predicts the branch target and enables the processor to begin executing instructions long before the branch true execution path is known. All branches utilize the branch prediction unit (BPU) for prediction. This unit predicts the target address not only based on the EIP of the branch but also based on the execution path through which execution reached this EIP. The BPU can efficiently predict the following branch types:

- Conditional branches.
- Direct calls and jumps.
- Indirect calls and jumps.
- Returns.

2.1.4 Micro-op Queue and the Loop Stream Detector (LSD)

The micro-op queue decouples the front end and the out-of order engine. It stays between the micro-op generation and the renamer as shown in Figure 2-1. This queue helps to hide bubbles which are introduced between the various sources of micro-ops in the front end and ensures that four micro-ops are delivered for execution, each cycle.

The micro-op queue provides post-decode functionality for certain instructions types. In particular, loads combined with computational operations and all stores, when used with indexed addressing, are represented as a single micro-op in the decoder or Decoded ICache. In the micro-op queue they are fragmented into two micro-ops through a process called un-lamination, one does the load and the other does the operation. A typical example is the following "load plus operation" instruction:

```
ADD                    RAX, [RBP+RSI]; rax := rax + LD( RBP+RSI )
```

Similarly, the following store instruction has three register sources and is broken into "generate store address" and "generate store data" sub-components.

```
MOV                    [ESP+ECX*4+12345678], AL
```

The additional micro-ops generated by unlamination use the rename and retirement bandwidth. However, it has an overall power benefit. For code that is dominated by indexed addressing (as often happens with array processing), recoding algorithms to use base (or base+displacement) addressing can sometimes improve performance by keeping the load plus operation and store instructions fused.

The Loop Stream Detector (LSD)

The Loop Stream Detector was introduced in Intel® Core microarchitectures. The LSD detects small loops that fit in the micro-op queue and locks them down. The loop streams from the micro-op queue, with no more fetching, decoding, or reading micro-ops from any of the caches, until a branch mis-prediction inevitably ends it.

The loops with the following attributes qualify for LSD/micro-op queue replay:

- Up to eight chunk fetches of 32-instruction-bytes.
- Up to 28 micro-ops (~28 instructions).
- All micro-ops are also resident in the Decoded ICache.
- Can contain no more than eight taken branches and none of them can be a CALL or RET.
- Cannot have mismatched stack operations. For example, more PUSH than POP instructions.

Many calculation-intensive loops, searches and software string moves match these characteristics.

Use the loop cache functionality opportunistically. For high performance code, loop unrolling is generally preferable for performance even when it overflows the LSD capability.

2.2 THE OUT-OF-ORDER ENGINE

The Out-of-Order engine provides improved performance over prior generations with excellent power characteristics. It detects dependency chains and sends them to execution out-of-order while maintaining the correct data flow. When a dependency chain is waiting for a resource, such as a second-level data cache line, it sends micro-ops from another chain to the execution core. This increases the overall rate of instructions executed per cycle (IPC).

The out-of-order engine consists of two blocks, shown in Figure 2-1: Core Functional Diagram, the Rename/retirement block, and the Scheduler.

The Out-of-Order (OOO) engine contains the following major components:

Renamer. The Renamer component moves micro-ops from the front end to the execution core. It eliminates false dependencies among micro-ops, thereby enabling out-of-order execution of micro-ops.

Scheduler. The Scheduler component queues micro-ops until all source operands are ready. Schedules and dispatches ready micro-ops to the available execution units in as close to a first in first out (FIFO) order as possible.

Retirement. The Retirement component retires instructions and micro-ops in order and handles faults and exceptions.

2.2.1 Renamer

The Renamer is the bridge between the in-order part in Figure 2-1, and the dataflow world of the Scheduler. It moves up to four micro-ops every cycle from the micro-op queue to the out-of-order engine. Although the renamer can send up to 4 micro-ops (unfused, micro-fused, or macro-fused) per cycle, this is equivalent to the issue port can dispatch six micro-ops per cycle. In this process, the out-of-order core carries out the following steps:

- Renames architectural sources and destinations of the micro-ops to micro-architectural sources and destinations.
- Allocates resources to the micro-ops. For example, load or store buffers.
- Binds the micro-op to an appropriate dispatch port.

Some micro-ops can execute to completion during rename and are removed from the pipeline at that point, effectively costing no execution bandwidth. These include:

- Zero idioms (dependency breaking idioms).
- NOP.
- VZEROUPPER.
- FXCHG.

The renamer can allocate two branches each cycle, compared to one branch each cycle in the previous microarchitecture. This can eliminate some bubbles in execution.

Micro-fused load and store operations that use an index register are decomposed to two micro-ops, hence consume two out of the four slots the Renamer can use every cycle.

Dependency Breaking Idioms

Instruction parallelism can be improved by using common instructions to clear register contents to zero. The renamer can detect them on the zero evaluation of the destination register.

Use one of these dependency breaking idioms to clear a register when possible.

- XOR REG,REG
- SUB REG,REG
- PXOR/VPXOR XMMREG,XMMREG
- PSUBB/W/D/Q XMMREG,XMMREG
- VPSUBB/W/D/Q XMMREG,XMMREG
- XORPS/PD XMMREG,XMMREG
- VXORPS/PD YMMREG, YMMREG

Since zero idioms are detected and removed by the renamer, they have no execution latency.

There is another dependency breaking idiom - the "ones idiom".

- CMPEQ XMM1, XMM1; "ones idiom" set all elements to all "ones"

In this case, the micro-op must execute, however, since it is known that regardless of the input data the output data is always "all ones" the micro-op dependency upon its sources does not exist as with the zero idiom and it can execute as soon as it finds a free execution port.

2.2.2 Scheduler

The scheduler controls the dispatch of micro-ops onto their execution ports. In order to do this, it must identify which micro-ops are ready and where its sources come from: a register file entry, or a bypass directly from an execution unit. Depending on the availability of dispatch ports and writeback buses, and the priority of ready micro-ops, the scheduler selects which micro-ops are dispatched every cycle.

2.2.3 The Execution Core

The execution core is superscalar and can process instructions out of order. The execution core optimizes overall performance by handling the most common operations efficiently, while minimizing potential delays.

The out-of-order execution core improves execution unit organization over prior generation in the following ways:

- Reduction in read port stalls.
- Reduction in writeback conflicts and delays.
- Reduction in power.
- Reduction of SIMD FP assists dealing with denormal inputs and underflow outputs.

Some high precision FP algorithms need to operate with FTZ=0 and DAZ=0, i.e. permitting underflow intermediate results and denormal inputs to achieve higher numerical precision at the expense of reduced performance on prior generation microarchitectures due to SIMD FP assists. The reduction of SIMD FP assists in Sandy Bridge microarchitecture applies to the following Intel SSE instructions (and Intel AVX variants): ADDPD/ADDPS, MULPD/MULPS, DIVPD/DIVPS, and CVTPD2PS.

The out-of-order core consist of three execution stacks, where each stack encapsulates a certain type of data. The execution core contains the following execution stacks:

- General purpose integer.
- SIMD integer and floating-point.
- X87.

The execution core also contains connections to and from the cache hierarchy. The loaded data is fetched from the caches and written back into one of the stacks.

The scheduler can dispatch up to six micro-ops every cycle, one on each port. The following table summarizes which operations can be dispatched on which port.

Table 2-3. Dispatch Port and Execution Stacks

	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
Integer	ALU, Shift	ALU, Fast LEA, Slow LEA, MUL	Load_Addr, Store_addr	Load_Addr Store_addr	Store_data	ALU, Shift, Branch, Fast LEA
SSE-Int, AVX-Int, MMX	Mul, Shift, STTNI, Int-Div, 128b-Mov	ALU, Shuf, Blend, 128b-Mov			Store_data	ALU, Shuf, Shift, Blend, 128b-Mov
SSE-FP, AVX-FP_low	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov
X87, AVX-FP_High	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov

After execution, the data is written back on a writeback bus corresponding to the dispatch port and the data type of the result. Micro-ops that are dispatched on the same port but have different latencies may need the write back bus at the same cycle. In these cases the execution of one of the micro-ops is delayed until the writeback bus is available. For example, MULPS (five cycles) and BLENDPS (one cycle) may collide if both are ready for execution on port 0: first the MULPS and four cycles later the BLENDPS. Sandy Bridge microarchitecture eliminates such collisions as long as the micro-ops write the results to different stacks. For example, integer ADD (one cycle) can be dispatched four cycles after MULPS (five cycles) since the integer ADD uses the integer stack while the MULPS uses the FP stack.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a one- or two-cycle delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. The following table describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

Table 2-4. Execution Core Writeback Latency (cycles)

	Integer	**SSE-Int, AVX-Int, MMX**	**SSE-FP, AVX-FP_low**	**X87, AVX-FP_High**
Integer	0	micro-op (port 0)	micro-op (port 0)	micro-op (port 0) + 1 cycle
SSE-Int, AVX-Int, MMX	micro-op (port 5) or micro-op (port 5) +1 cycle	0	1 cycle delay	0
SSE-FP, AVX-FP_low	micro-op (port 5) or micro-op (port 5) +1 cycle	1 cycle delay	0	micro-op (port 5) +1 cycle
X87, AVX-FP_High	micro-op (port 5) +1 cycle	0	micro-op (port 5) +1 cycle	0
Load	0	1 cycle delay	1 cycle delay	2 cycle delay

2.3 CACHE HIERARCHY

The cache hierarchy contains a first level instruction cache, a first level data cache (L1 DCache) and a second level (L2) cache, in each core. The L1D cache may be shared by two logical processors if the processor support Intel HT. The L2 cache is shared by instructions and data. All cores in a physical processor package connect to a shared last level cache (LLC) via a ring connection.

The caches use the services of the Instruction Translation Lookaside Buffer (ITLB), Data Translation Lookaside Buffer (DTLB) and Shared Translation Lookaside Buffer (STLB) to translate linear addresses to physical address. Data coherency in all cache levels is maintained using the MESI protocol. For more information, see the Intel® 64 IA-32 Architectures Software Developer's Manual, Volume 3. Cache hierarchy details can be obtained at run-time using the CPUID instruction. see the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

Table 2-5. Cache Parameters

Level	**Capacity**	**Associativity (ways)**	**Line Size (bytes)**	**Write Update Policy**	**Inclusive**
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

2.3.1 Load and Store Operation Overview

This section provides an overview of the load and store operations.

Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for it in the caches and memory. Table 2-6 shows the access lookup order and best case latency. The actual latency can vary depending on the cache queue occupancy, LLC ring occupancy, memory components, and their parameters.

Table 2-6. Lookup Order and Load Latency

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1 Data	4[1]	2 x16 bytes
L2 (Unified)	12	1 x 32 bytes
Third Level (LLC)	26-31[2]	1 x 32 bytes
L2 and L1 DCache in other cores if applicable	43- clean hit; 60 - dirty hit	

NOTES:

1. Subject to execution core bypass restriction shown in Table 2-4.
2. Latency of L3 varies with product segment and sku. The values apply to second generation Intel Core processor families.

The LLC is inclusive of all cache levels above it - data contained in the core caches must also reside in the LLC. Each cache line in the LLC holds an indication of the cores that may have this line in their L2 and L1 caches. If there is an indication in the LLC that other cores may hold the line of interest and its state might have to modify, there is a lookup into the L1 DCache and L2 of these cores too. The lookup is called "clean" if it does not require fetching data from the other core caches. The lookup is called "dirty" if modified data has to be fetched from the other core caches and transferred to the loading core.

The latencies shown above are the best-case scenarios. Sometimes a modified cache line has to be evicted to make space for a new cache line. The modified cache line is evicted in parallel to bringing the new data and does not require additional latency. However, when data is written back to memory, the eviction uses cache bandwidth and possibly memory bandwidth as well. Therefore, when multiple cache misses require the eviction of modified lines within a short time, there is an overall degradation in cache response time. Memory access latencies vary based on occupancy of the memory controller queues, DRAM configuration, DDR parameters, and DDR paging behavior (if the requested page is a page-hit, page-miss or page-empty).

Stores

When an instruction writes data to a memory location that has a write back memory type, the processor first ensures that it has the line containing this memory location in its L1 DCache, in Exclusive or Modified MESI state. If the cache line is not there, in the right state, the processor fetches it from the next levels of the memory hierarchy using a Read for Ownership request. The processor looks for the cache line in the following locations, in the specified order:

1. L1 DCache
2. L2
3. Last Level Cache
4. L2 and L1 DCache in other cores, if applicable
5. Memory

Once the cache line is in the L1 DCache, the new data is written to it, and the line is marked as Modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of store instruction retirement. Therefore, the store latency usually does not affect the store instruction itself. However, several sequential stores that miss the L1 DCache may have cumulative latency that can affect performance. As long as the store does not complete, its entry remains occupied in the store buffer. When the store buffer becomes full, new micro-ops cannot enter the execution pipe and execution might stall.

2.3.2 L1 DCache

The L1 DCache is the first level data cache. It manages all load and store requests from all types through its internal data structures. The L1 DCache:

- Enables loads and stores to issue speculatively and out of order.
- Ensures that retired loads and stores have the correct data upon retirement.
- Ensures that loads and stores follow the memory ordering rules of the IA-32 and Intel 64 instruction set architecture.

Table 2-7. L1 Data Cache Components

Component	Sandy Bridge Microarchitecture	Nehalem Microarchitecture
Data Cache Unit (DCU)	32KB, 8 ways	32KB, 8 ways
Load buffers	64 entries	48 entries
Store buffers	36 entries	32 entries
Line fill buffers (LFB)	10 entries	10 entries

The DCU is organized as 32 KBytes, eight-way set associative. Cache line size is 64-bytes arranged in eight banks.

Internally, accesses are up to 16 bytes, with 256-bit Intel AVX instructions utilizing two 16-byte accesses. Two load operations and one store operation can be handled each cycle.

The L1 DCache maintains requests which cannot be serviced immediately to completion. Some reasons for requests that are delayed: cache misses, unaligned access that splits across cache lines, data not ready to be forwarded from a preceding store, loads experiencing bank collisions, and load block due to cache line replacement.

The L1 DCache can maintain up to 64 load micro-ops from allocation until retirement. It can maintain up to 36 store operations from allocation until the store value is committed to the cache, or written to the line fill buffers (LFB) in the case of non-temporal stores.

The L1 DCache can handle multiple outstanding cache misses and continue to service incoming stores and loads. Up to 10 requests of missing cache lines can be managed simultaneously using the LFB.

The L1 DCache is a write-back write-allocate cache. Stores that hit in the DCU do not update the lower levels of the memory hierarchy. Stores that miss the DCU allocate a cache line.

Loads

The L1 DCache architecture can service two loads per cycle, each of which can be up to 16 bytes. Up to 32 loads can be maintained at different stages of progress, from their allocation in the out of order engine until the loaded value is returned to the execution core.

Loads can:

- Read data before preceding stores when the load address and store address ranges are known not to conflict.
- Be carried out speculatively, before preceding branches are resolved.
- Take cache misses out of order and in an overlapped manner.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access uncacheable memory.

The common load latency is five cycles. When using a simple addressing mode, base plus offset that is smaller than 2048, the load latency can be four cycles. This technique is especially useful for pointer-chasing code. However, overall latency varies depending on the target register data type due to stack bypass. See Section 2.2.3 for more information.

The following table lists overall load latencies. These latencies assume the common case of flat segment, that is, segment base address is zero. If segment base is not zero, load latency increases.

Table 2-8. Effect of Addressing Modes on Load Latency

Data Type/Addressing Mode	Base + Offset > 2048; Base + Index [+ Offset]	Base + Offset < 2048
Integer	5	4
MMX, SSE, 128-bit AVX	6	5
X87	7	6
256-bit AVX	7	7

Stores

Stores to memory are executed in two phases:

- Execution phase. Fills the store buffers with linear and physical address and data. Once store address and data are known, the store data can be forwarded to the following load operations that need it.
- Completion phase. After the store retires, the L1 DCache moves its data from the store buffers to the DCU, up to 16 bytes per cycle.

Address Translation

The DTLB can perform three linear to physical address translations every cycle, two for load addresses and one for a store address. If the address is missing in the DTLB, the processor looks for it in the STLB, which holds data and instruction address translations. The penalty of a DTLB miss that hits the STLB is seven cycles. Large page support include 1G byte pages, in addition to 4K and 2M/4M pages.

The DTLB and STLB are four way set associative. The following table specifies the number of entries in the DTLB and STLB.

Table 2-9. DTLB and STLB Parameters

TLB	Page Size	Entries
DTLB	4KB	64
	2MB/4MB	32
	1GB	4
STLB	4KB	512

Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the data can forward directly from the store operation to the load. This process, called store to load forwarding, saves cycles by enabling the load to obtain the data directly from the store operation instead of through memory. You can take advantage of store forwarding to quickly move complex structures without losing the ability to forward the subfields. The memory control unit can handle store forwarding situations with less restrictions compared to previous micro-architectures.

The following rules must be met to enable store to load forwarding:

- The store must be the last store to that address, prior to the load.
- The store must contain all data being loaded.
- The load is from a write-back memory type and neither the load nor the store are non-temporal accesses.

Stores cannot forward to loads in the following cases:

- Four byte and eight byte loads that cross eight byte boundary, relative to the preceding 16- or 32-byte store.
- Any load that crosses a 16-byte boundary of a 32-byte store.

Table 2-10 to Table 2-13 detail the store to load forwarding behavior. For a given store size, all the loads that may overlap are shown and specified by 'F'. Forwarding from 32 byte store is similar to forwarding from each of the 16 byte halves of the store. Cases that cannot forward are shown as 'N'.

Table 2-10. Store Forwarding Conditions (1 and 2 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	F														
	2	F	N														

Table 2-11. Store Forwarding Conditions (4-16 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	F	F	F												
	2	F	F	F	N												
	4	F	N	N	N												
8	1	F	F	F	F	F	F	F	F								
	2	F	F	F	F	F	F	F	N								
	4	F	F	F	F	F	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-12. 32-byte Store Forwarding Conditions (0-15 byte alignment)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-13. 32-byte Store Forwarding Conditions (16-31 byte alignment)

		Load Alignment															
Store Size	Load Size	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Memory Disambiguation

A load operation may depend on a preceding store. Many microarchitectures block loads until all preceding store addresses are known. The memory disambiguator predicts which loads will not depend on any previous stores whose addresses aren't yet known. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from an earlier store to the same address. This hides the load latency. Eventually, the prediction is verified. If the load did indeed depend on a store whose address was unknown at the time the load executed, this conflict is detected and the load and all succeeding instructions are re-executed.

The following loads are not disambiguated. The execution of these loads is stalled until addresses of all previous stores are known.

- Loads that cross the 16-byte aligned boundary, other than 32-byte loads.
- 32-byte Intel AVX loads that are not 32-byte aligned.

Bank Conflict

Since 16-byte loads can cover up to three banks, and two loads can happen every cycle, it is possible that six of the eight banks may be accessed per cycle, for loads. A bank conflict happens when two load accesses need the same bank (their address has the same 2-4 bit value) in different sets, at the same time. When a bank conflict occurs, one of the load accesses is recycled internally.

In many cases two loads access exactly the same bank in the same cache line, as may happen when popping operands off the stack, or any sequential accesses. In these cases, conflict does not occur and the loads are serviced simultaneously.

2.3.2.1 Ring Interconnect and Last Level Cache

The system-on-a-chip design provides a high bandwidth bi-directional ring bus to connect between the IA cores and various sub-systems in the uncore. In the second generation Intel Core processor 2xxx series, the uncore subsystem include a system agent, the graphics unit (GT) and the last level cache (LLC).

The LLC consists of multiple cache slices. The number of slices is equal to the number of IA cores. Each slice has logic portion and data array portion. The logic portion handles data coherency, memory ordering, access to the data array portion, LLC misses and writeback to memory, and more. The data array portion stores cache lines. Each slice contains a full cache port that can supply 32 bytes/cycle.

The physical addresses of data kept in the LLC data arrays are distributed among the cache slices by a hash function, such that addresses are uniformly distributed. The data array in a cache block may have 4/8/12/16 ways corresponding to 0.5M/1M/1.5M/2M block size. However, due to the address distribution among the cache blocks from the software point of view, this does not appear as a normal N-way cache.

From the processor cores and the GT view, the LLC act as one shared cache with multiple ports and bandwidth that scales with the number of cores. The LLC hit latency, ranging between 26-31 cycles, depends on the core location relative to the LLC block, and how far the request needs to travel on the ring.

The number of cache-slices increases with the number of cores, therefore the ring and LLC are not likely to be a bandwidth limiter to core operation.

The GT sits on the same ring interconnect, and uses the LLC for its data operations as well. In this respect it is very similar to an IA core. Therefore, high bandwidth graphic applications using cache bandwidth and significant cache footprint, can interfere, to some extent, with core operations.

All the traffic that cannot be satisfied by the LLC, such as LLC misses, dirty line writeback, non-cacheable operations, and MMIO/IO operations, still travels through the cache-slice logic portion and the ring, to the system agent.

In the Intel Xeon Processor E5 Family, the uncore subsystem does not include the graphics unit (GT). Instead, the uncore subsystem contains many more components, including an LLC with larger capacity and snooping capabilities to support multiple processors, Intel® QuickPath Interconnect interfaces that can support multi-socket platforms, power management control hardware, and a system agent capable of supporting high-bandwidth traffic from memory and I/O devices.

In the Intel Xeon processor E5 2xxx or 4xxx families, the LLC capacity generally scales with the number of processor cores with 2.5 MBytes per core.

2.3.2.2 Data Prefetching

Data can be speculatively loaded to the L1 DCache using software prefetching, hardware prefetching, or any combination of the two.

You can use the four Streaming SIMD Extensions (SSE) prefetch instructions to enable software-controlled prefetching. These instructions are hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is intended for prefetching data, but not for prefetching code.

The rest of this section describes the various hardware prefetching mechanisms provided by Sandy Bridge microarchitecture and their improvement over previous processors. The goal of the prefetchers is to automatically predict which data the program is about to consume. If this data is not close-by to the execution core or inner cache, the prefetchers bring it from the next levels of cache hierarchy and memory. Prefetching has the following effects:

- Improves performance if data is arranged sequentially in the order used in the program.
- May cause slight performance degradation due to bandwidth issues, if access patterns are sparse instead of local.
- On rare occasions, if the algorithm's working set is tuned to occupy most of the cache and unneeded prefetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

Data Prefetch to L1 Data Cache

Data prefetching is triggered by load operations when the following conditions are met:

- Load is from writeback memory type.
- The prefetched data is within the same 4K byte page as the load instruction that triggered it.
- No fence is in progress in the pipeline.
- Not many other load misses are in progress.
- There is not a continuous stream of stores.

Two hardware prefetchers load data to the L1 DCache:

- **Data cache unit (DCU) prefetcher**. This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.
- **Instruction pointer (IP)-based stride prefetcher**. This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to 2K bytes.

Data Prefetch to the L2 and Last Level Cache

The following two hardware prefetchers fetched data from memory to the L2 cache and last level cache:

Spatial Prefetcher: This prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk.

Streamer: This prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 DCache requests initiated by load and store operations and by the hardware prefetchers, and L1 ICache requests for code fetch. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. Prefetched cache lines must be in the same 4K page.

The streamer and spatial prefetcher prefetch the data to the last level cache. Typically data is brought also to the L2 unless the L2 cache is heavily loaded with missing demand requests.

Enhancement to the streamer includes the following features:

- The streamer may issue two prefetch requests on every L2 lookup. The streamer can run up to 20 lines ahead of the load request.
- Adjusts dynamically to the number of outstanding requests per core. If there are not many outstanding requests, the streamer prefetches further ahead. If there are many outstanding requests it prefetches to the LLC only and less far ahead.
- When cache lines are far ahead, it prefetches to the last level cache only and not to the L2. This method avoids replacement of useful cache lines in the L2 cache.
- Detects and maintains up to 32 streams of data accesses. For each 4K byte page, you can maintain one forward and one backward stream can be maintained.

2.3.3 System Agent

The system agent implemented in the second generation Intel Core processor family contains the following components:

- An arbiter that handles all accesses from the ring domain and from I/O (PCIe* and DMI) and routes the accesses to the right place.
- PCIe controllers connect to external PCIe devices. The PCIe controllers have different configuration possibilities the varies with product segment specifics: x16+x4, x8+x8+x4, x8+x4+x4+x4.
- DMI controller connects to the PCH chipset.
- Integrated display engine, Flexible Display Interconnect, and Display Port, for the internal graphic operations.
- Memory controller.

All main memory traffic is routed from the arbiter to the memory controller. The memory controller in the second generation Intel Core processor 2xxx series support two channels of DDR, with data rates of 1066MHz, 1333MHz and 1600MHz, and 8 bytes per cycle, depending on the unit type, system configuration and DRAMs. Addresses are distributed between memory channels based on a local hash function that attempts to balance the load between the channels in order to achieve maximum bandwidth and minimum hotspot collisions.

For best performance, populate both channels with equal amounts of memory, preferably the exact same types of DIMMs. In addition, using more ranks for the same amount of memory, results in somewhat better memory bandwidth, since more DRAM pages can be open simultaneously. For best performance, populate the system with the highest supported speed DRAM (1333MHz or 1600MHz data rates, depending on the max supported frequency) with the best DRAM timings.

The two channels have separate resources and handle memory requests independently. The memory controller contains a high-performance out-of-order scheduler that attempts to maximize memory bandwidth while minimizing latency. Each memory channel contains a 32 cache-line write-data-buffer. Writes to the memory controller are considered completed when they are written to the write-data-buffer. The write-data-buffer is flushed out to main memory at a later time, not impacting write latency.

Partial writes are not handled efficiently on the memory controller and may result in read-modify-write operations on the DDR channel if the partial-writes do not complete a full cache-line in time. Software should avoid creating partial write transactions whenever possible and consider alternative, such as buffering the partial writes into full cache line writes.

The memory controller also supports high-priority isochronous requests (such as USB isochronous, and Display isochronous requests). High bandwidth of memory requests from the integrated display engine takes up some of the memory bandwidth and impacts core access latency to some degree.

2.3.4 Ivy Bridge Microarchitecture

3rd generation Intel Core processors are based on Ivy Bridge microarchitecture. Most of the features described in Section 2.1 - Section 2.3.3 also apply to Ivy Bridge microarchitecture. This section covers feature differences in microarchitecture that can affect coding and performance.

Support for new instructions enabling include:

- Numeric conversion to and from half-precision floating-point values.
- Hardware-based random number generator compliant to NIST SP 800-90A.
- Reading and writing to FS/GS base registers in any ring to improve user-mode threading support.

For details about using the hardware based random number generator instruction RDRAND, please refer to the article available from Intel Software Network at https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide/.

A small number of microarchitectural enhancements that can be beneficial to software:

- Hardware prefetch enhancement: A next-page prefetcher (NPP) is added in Ivy Bridge microarchitecture. The NPP is triggered by sequential accesses to cache lines approaching the page boundary, either upwards or downwards.
- Zero-latency register move operation: A subset of register-to-register MOV instructions are executed at the front end, conserving scheduling and execution resource in the out-of-order engine.
- Front end enhancement: In Sandy Bridge microarchitecture, the micro-op queue is statically partitioned to provide 28 entries for each logical processor, irrespective of software executing in single thread or multiple threads. If one logical processor is not active in Ivy Bridge microarchitecture, then a single thread executing on that processor core can use the 56 entries in the micro-op queue. In this case, the LSD can handle larger loop structure that would require more than 28 entries.
- The latency and throughput of some instructions have been improved over those of Sandy Bridge microarchitecture. For example, 256-bit packed floating-point divide and square root operations are faster; ROL and ROR instructions are also improved.

CHAPTER 3 INTEL® CORE™ MICROARCHITECTURE AND ENHANCED INTEL® CORE™ MICROARCHITECTURE

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multi-threaded workloads:

- **Intel® Wide Dynamic Execution** enables each processor core to fetch, dispatch, execute with high bandwidths and retire up to four instructions per cycle. Features include:
  - Fourteen-stage efficient pipeline.
  - Three arithmetic logical units.
  - Four decoders to decode up to five instruction per cycle.
  - Macro-fusion and micro-fusion to improve front end throughput.
  - Peak issue rate of dispatching up to six micro-ops per cycle.
  - Peak retirement bandwidth of up to four micro-ops per cycle.
  - Advanced branch prediction.
  - Stack pointer tracker to improve efficiency of executing function/procedure entries and exits.
- **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, optimal performance and flexibility for single-threaded and multi-threaded applications. Features include:
  - Optimized for multicore and single-threaded execution environments.
  - 256 bit internal data path to improve bandwidth from L2 to first-level data cache.
  - Unified, shared second-level cache of 4 Mbyte, 16 way (or 2 MByte, 8 way).
- **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution. Features include:
  - Hardware prefetchers to reduce effective latency of second-level cache misses.
  - Hardware prefetchers to reduce effective latency of first-level data cache misses.
  - Memory disambiguation to improve efficiency of speculative execution engine.
- **Intel® Advanced Digital Media Boost** improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations. Features include:
  - Single-cycle throughput of most 128-bit SIMD instructions (except 128-bit shuffle, pack, unpack operations)
  - Up to eight floating-point operations per cycle
  - Three issue ports available to dispatching SIMD instructions for execution.

The Enhanced Intel Core microarchitecture supports all of the features of Intel Core microarchitecture and provides a comprehensive set of enhancements.

- **Intel® Wide Dynamic Execution** includes several enhancements:
  - A radix-16 divider replacing previous radix-4 based divider to speedup long-latency operations such as divisions and square roots.
  - Improved system primitives to speedup long-latency operations such as RDTSC, STI, CLI, and VM exit transitions.
- **Intel® Advanced Smart Cache** provides up to 6 MBytes of second-level cache shared between two processor cores (quad-core processors have up to 12 MBytes of L2); up to 24 way/set associativity.

- **Intel® Smart Memory Access** supports high-speed system bus up 1600 MHz and provides more efficient handling of memory operations such as split cache line load and store-to-load forwarding situations.
- **Intel® Advanced Digital Media Boost** provides 128-bit shuffler unit to speedup shuffle, pack, unpack operations; adds support for forty-seven Intel SSE4.1 instructions.

In the sub-sections of 2.1.x, most of the descriptions on Intel Core microarchitecture also applies to Enhanced Intel Core microarchitecture. Differences between them are note explicitly.

3.1 INTEL® CORE™ MICROARCHITECTURE PIPELINE OVERVIEW

The pipeline of the Intel Core microarchitecture contains:

- An in-order issue front end that fetches instruction streams from memory, with four instruction decoders to supply decoded instruction (micro-ops) to the out-of-order execution core.
- An out-of-order superscalar execution core that can issue up to six micro-ops per cycle (see Table 3-2) and reorder micro-ops to execute as soon as sources are ready and execution resources are available.
- An in-order retirement unit that ensures the results of execution of micro-ops are processed and architectural states are updated according to the original program order.

Intel Core 2 Extreme processor X6800, Intel Core 2 Duo processors and Intel Xeon processor 3000, 5100 series implement two processor cores based on the Intel Core microarchitecture. Intel Core 2 Extreme quad-core processor, Intel Core 2 Quad processors and Intel Xeon processor 3200 series, 5300 series implement four processor cores. Each physical package of these quad-core processors contains two processor dies, each die containing two processor cores. The functionality of the subsystems in each core are depicted in Figure 3-1.

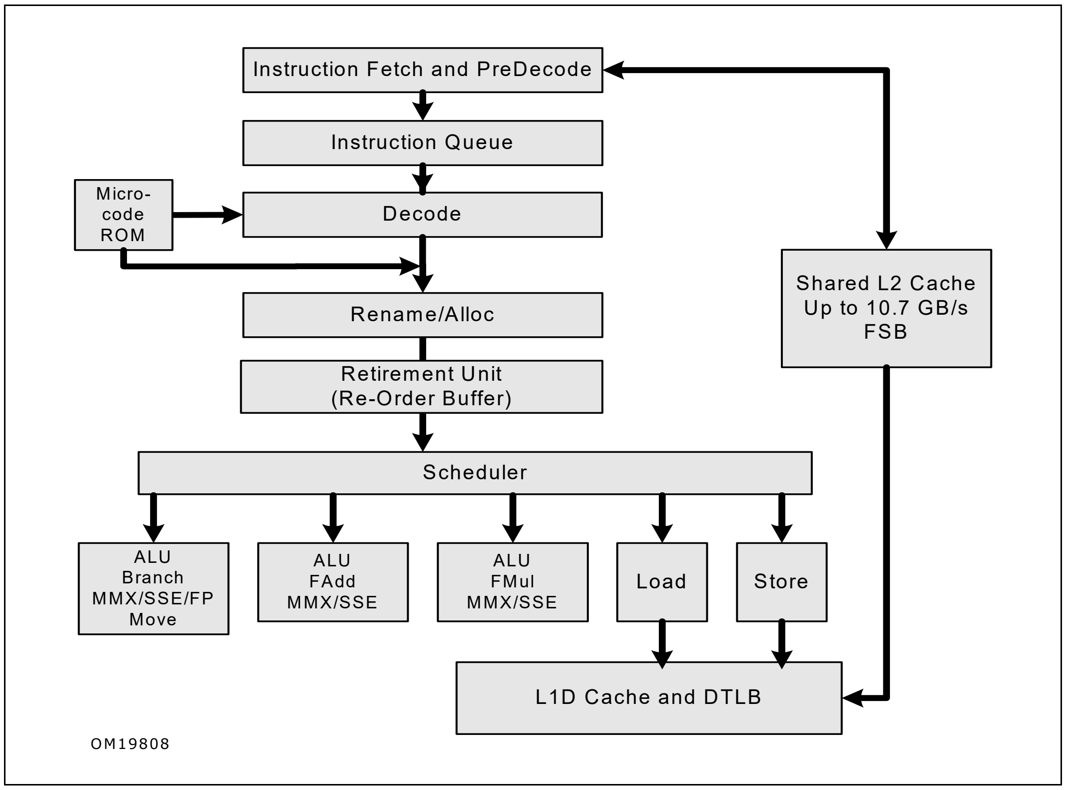

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
Shared L2 Cache Up to 10.7 GB/s FSB
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
OM19808

Figure 3-1. Intel® Core™ Microarchitecture Pipeline Functionality

3.1.1 Front End

The front ends needs to supply decoded instructions (micro-ops) and sustain the stream to a six-issue wide out-of-order engine. The components of the front end, their functions, and the performance challenges to microarchitectural design are described in Table 3-1.

Table 3-1. Components of the Front End

Component	Functions	Performance Challenges
Branch Prediction Unit (BPU)	• Helps the instruction fetch unit fetch the most likely instruction to be executed by predicting the various branch types: conditional, indirect, direct, call, and return. Uses dedicated hardware for each type.	• Enables speculative execution. • Improves speculative execution efficiency by reducing the amount of code in the "non-architected path"[1] to be fetched into the pipeline.
Instruction Fetch Unit	• Prefetches instructions that are likely to be executed • Caches frequently-used instructions • Predecodes and buffers instructions, maintaining a constant bandwidth despite irregularities in the instruction stream	• Variable length instruction format causes unevenness (bubbles) in decode bandwidth. • Taken branches and misaligned targets causes disruptions in the overall bandwidth delivered by the fetch unit.
Instruction Queue and Decode Unit	• Decodes up to four instructions, or up to five with macro-fusion • Stack pointer tracker algorithm for efficient procedure entry and exit • Implements the Macro-Fusion feature, providing higher performance and efficiency • The Instruction Queue is also used as a loop cache, enabling some loops to be executed with both higher bandwidth and lower power	• Varying amounts of work per instruction requires expansion into variable numbers of micro-ops. • Prefix adds a dimension of decoding complexity. • Length Changing Prefix (LCP) can cause front end bubbles.

NOTES:

1. Code paths that the processor thought it should execute but then found out it should go in another path and therefore reverted from its initial intention.

3.1.1.1 Branch Prediction Unit

Branch prediction enables the processor to begin executing instructions long before the branch outcome is decided. All branches utilize the BPU for prediction. The BPU contains the following features:

- 16-entry Return Stack Buffer (RSB). It enables the BPU to accurately predict RET instructions.
- Front end queuing of BPU lookups. The BPU makes branch predictions for 32 bytes at a time, twice the width of the fetch engine. This enables taken branches to be predicted with no penalty.

  Even though this BPU mechanism generally eliminates the penalty for taken branches, software should still regard taken branches as consuming more resources than do not-taken branches.

The BPU makes the following types of predictions:

- Direct Calls and Jumps. Targets are read as a target array, without regarding the taken or not-taken prediction.
- Indirect Calls and Jumps. These may either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior.
- Conditional branches. Predicts the branch target and whether or not the branch will be taken.

For information about optimizing software for the BPU, see Section 3.4, "Optimizing the Front End."

3.1.1.2 Instruction Fetch Unit

The instruction fetch unit comprises the instruction translation lookaside buffer (ITLB), an instruction prefetcher, the instruction cache and the predecode logic of the instruction queue (IQ).

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB into the instruction cache and instruction prefetch buffers. A hit in the instruction cache causes 16 bytes to be delivered to the instruction predecoder. Typical programs average slightly less than 4 bytes per instruction, depending on the code being executed. Since most instructions can be decoded by all decoders, an entire fetch can often be consumed by the decoders in one cycle.

A misaligned target reduces the number of instruction bytes by the amount of offset into the 16 byte fetch quantity. A taken branch reduces the number of instruction bytes delivered to the decoders since the bytes after the taken branch are not decoded. Branches are taken approximately every 10 instructions in typical integer code, which translates into a "partial" instruction fetch every 3 or 4 cycles.

Due to stalls in the rest of the machine, front end starvation does not usually cause performance degradation. For extremely fast code with larger instructions (such as Intel SSE2 integer media kernels), it may be beneficial to use targeted alignment to prevent instruction starvation.

Instruction PreDecode

The predecode unit accepts the sixteen bytes from the instruction cache or prefetch buffers and carries out the following tasks:

- Determine the length of the instructions.
- Decode all prefixes associated with instructions.
- Mark various properties of instructions for the decoders (for example, "is branch.").

The predecode unit can write up to six instructions per cycle into the instruction queue. If a fetch contains more than six instructions, the predecoder continues to decode up to six instructions per cycle until all instructions in the fetch are written to the instruction queue. Subsequent fetches can only enter predecoding after the current fetch completes.

For a fetch of seven instructions, the predecoder decodes the first six in one cycle, and then only one in the next cycle. This process would support decoding 3.5 instructions per cycle. Even if the instruction per cycle (IPC) rate is not fully optimized, it is higher than the performance seen in most applications. In general, software usually does not have to take any extra measures to prevent instruction starvation.

The following instruction prefixes cause problems during length decoding. These prefixes can dynamically change the length of instructions and are known as length changing prefixes (LCPs):

- Operand Size Override (66H) preceding an instruction with a word immediate data.
- Address Size Override (67H) preceding an instruction with a mod R/M in real, 16-bit protected or 32-bit protected modes.

When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle.

Normal queuing within the processor pipeline usually cannot hide LCP penalties.

The REX prefix (4xh) in the Intel 64 architecture instruction set can change the size of two classes of instruction: MOV offset and MOV immediate. Nevertheless, it does not cause an LCP penalty and hence is not considered an LCP.

3.1.1.3 Instruction Queue (IQ)

The instruction queue is 18 instructions deep. It sits between the instruction predecode unit and the instruction decoders. It sends up to five instructions per cycle, and supports one macro-fusion per cycle. It also serves as a loop cache for loops smaller than 18 instructions. The loop cache operates as described below.

A Loop Stream Detector (LSD) resides in the BPU. The LSD attempts to detect loops which are candidates for streaming from the instruction queue (IQ). When such a loop is detected, the instruction bytes are locked down and the loop is allowed to stream from the IQ until a misprediction ends it. When the loop plays back from the IQ, it provides higher bandwidth at reduced power (since much of the rest of the front end pipeline is shut off).

The LSD provides the following benefits:

- No loss of bandwidth due to taken branches.
- No loss of bandwidth due to misaligned instructions.
- No LCP penalties, as the pre-decode stage has already been passed.
- Reduced front end power consumption, because the instruction cache, BPU and predecode unit can be idle.

Software should use the loop cache functionality opportunistically. Loop unrolling and other code optimizations may make the loop too big to fit into the LSD. For high performance code, loop unrolling is generally preferable for performance even when it overflows the loop cache capability.

3.1.1.4 Instruction Decode

The Intel Core microarchitecture contains four instruction decoders. The first, Decoder 0, can decode Intel 64 and IA-32 instructions up to 4 micro-ops in size. Three other decoders handle single micro-op instructions. The microsequencer can provide up to 3 micro-ops per cycle, and helps decode instructions larger than 4 micro-ops.

All decoders support the common cases of single micro-op flows, including: micro-fusion, stack pointer tracking and macro-fusion. Thus, the three simple decoders are not limited to decoding single micro-op instructions. Packing instructions into a 4-1-1-1 template is not necessary and not recommended.

Macro-fusion merges two instructions into a single micro-op. Intel Core microarchitecture is capable of one macro-fusion per cycle in 32-bit operation (including compatibility sub-mode of the Intel 64 architecture), but not in 64-bit mode because code that uses longer instructions (length in bytes) more often is less likely to take advantage of hardware support for macro-fusion.

3.1.1.5 Stack Pointer Tracker

The Intel 64 and IA-32 architectures have several commonly used instructions for parameter passing and procedure entry and exit: PUSH, POP, CALL, LEAVE and RET. These instructions implicitly update the stack pointer register (RSP), maintaining a combined control and parameter stack without software intervention. These instructions are typically implemented by several micro-ops in previous microarchitectures.

The Stack Pointer Tracker moves all these implicit RSP updates to logic contained in the decoders themselves. The feature provides the following benefits:

- Improves decode bandwidth, as PUSH, POP and RET are single micro-op instructions in Intel Core microarchitecture.
- Conserves execution bandwidth as the RSP updates do not compete for execution resources.
- Improves parallelism in the out of order execution engine as the implicit serial dependencies between micro-ops are removed.
- Improves power efficiency as the RSP updates are carried out on small, dedicated hardware.

3.1.1.6 Micro-fusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core. Micro-fusion provides the following performance advantages:

- Improves instruction bandwidth delivered from decode to retirement.

- Reduces power consumption as the complex micro-op represents more work in a smaller format (in terms of bit density), reducing overall "bit-toggling" in the machine for a given amount of work and virtually increasing the amount of storage in the out-of-order execution engine.

Many instructions provide register flavors and memory flavors. The flavor involving a memory operand will decode into a longer flow of micro-ops than the register version. Micro-fusion enables software to use memory to register operations to express the actual program behavior without worrying about a loss of decode bandwidth.

3.1.2 Execution Core

The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order. When a dependency chain causes the machine to wait for a resource (such as a second-level data cache line), the execution core executes other instructions. This increases the overall rate of instructions executed per cycle (IPC).

The execution core contains the following three major components:

- **Renamer** — Moves micro-ops from the front end to the execution core. Architectural registers are renamed to a larger set of microarchitectural registers. Renaming eliminates false dependencies known as read-after-read and write-after-read hazards.
- **Reorder buffer** (ROB) — Holds micro-ops in various stages of completion, buffers completed micro-ops, updates the architectural state in order, and manages ordering of exceptions. The ROB has 96 entries to handle instructions in flight.
- **Reservation station** (RS) — Queues micro-ops until all source operands are ready, schedules and dispatches ready micro-ops to the available execution units. The RS has 32 entries.

The initial stages of the out of order core move the micro-ops from the front end to the ROB and RS. In this process, the out of order core carries out the following steps:

- Allocates resources to micro-ops (for example: these resources could be load or store buffers).
- Binds the micro-op to an appropriate issue port.
- Renames sources and destinations of micro-ops, enabling out of order execution.
- Provides data to the micro-op when the data is either an immediate value or a register value that has already been calculated.

The following list describes various types of common operations and how the core executes them efficiently:

- **Micro-ops with single-cycle latency** — Most micro-ops with single-cycle latency can be executed by multiple execution units, enabling multiple streams of dependent operations to be executed quickly.
- **Frequently-used μops with longer latency** — These micro-ops have pipelined execution units so that multiple micro-ops of these types may be executing in different parts of the pipeline simultaneously.
- **Operations with data-dependent latencies** — Some operations, such as division, have data dependent latencies. Integer division parses the operands to perform the calculation only on significant portions of the operands, thereby speeding up common cases of dividing by small numbers.
- **Floating-point operations with fixed latency for operands that meet certain restrictions** — Operands that do not fit these restrictions are considered exceptional cases and are executed with higher latency and reduced throughput. The lower-throughput cases do not affect latency and throughput for more common cases.
- **Memory operands with variable latency, even in the case of an L1 cache hit** — Loads that are not known to be safe from forwarding may wait until a store-address is resolved before executing. The memory order buffer (MOB) accepts and processes all memory operations. See Section 3.1.3 for more information about the MOB.

3.1.2.1 Issue Ports and Execution Units

The scheduler can dispatch up to six micro-ops per cycle through the issue ports. The issue ports of Intel Core microarchitecture and Enhanced Intel Core microarchitecture are depicted in Table 3-2, the former is denoted by its CPUID signature of DisplayFamily_DisplayModel value of 06_0FH, the latter denoted by the corresponding signature value of 06_17H. The table provides latency and throughput data of common integer and floating-point (FP) operations for each issue port in cycles.

Table 3-2. Issue Ports of Intel® Core™ and Enhanced Intel® Core™ Microarchitectures

Executable operations	Latency, Throughput		Comment[1]
	Signature = 06_0FH	Signature = 06_17H	

Integer ALU
Integer SIMD ALU
FP/SIMD/SSE2 Move and Logic
	1, 1
	1, 1
	1, 1
		1, 1
		1, 1
		1, 1
			Includes 64-bit mode integer MUL;
			Issue port 0; Writeback port 0;

Single-precision (SP) FP MUL Double-precision FP MUL	4, 1 5, 1	4, 1 5, 1	Issue port 0; Writeback port 0
FP MUL (X87) FP Shuffle DIV/SQRT	5, 2 1, 1	5, 2 1, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.

Integer ALU
Integer SIMD ALU
FP/SIMD/SSE2 Move and Logic
	1, 1
	1, 1
	1, 1
		1, 1
		1, 1
		1, 1
			Excludes 64-bit mode integer MUL;
			Issue port 1; Writeback port 1;

FP ADD QW Shuffle	3, 1 1, 1[2]	3, 1 1, 1[3]	Issue port 1; Writeback port 1;
Integer loads FP loads	3, 1 4, 1	3, 1 4, 1	Issue port 2; Writeback port 2;
Store address[4]	3, 1	3, 1	Issue port 3;
Store data[5].			Issue Port 4;
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Issue port 5; Writeback port 5;
QW shuffles 128-bit Shuffle/Pack/Unpack	1, 1[2] 2-4, 2-4[6]	1, 1[3] 1-3, 1[7]	Issue port 5; Writeback port 5;

NOTES:
1. Mixing operations of different latencies that use the same port can result in writeback bus conflicts; this can reduce overall throughput.
2. 128-bit instructions executes with longer latency and reduced throughput.
3. Uses 128-bit shuffle unit in port 5.
4. Prepares the store forwarding and store retirement logic with the address of the data being stored.
5. Prepares the store forwarding and store retirement logic with the data being stored.
6. Varies with instructions; 128-bit instructions are executed using QW shuffle units.
7. Varies with instructions, 128-bit shuffle unit replaces QW shuffle units in Intel Core microarchitecture.

In each cycle, the RS can dispatch up to six micro-ops. Each cycle, up to 4 results may be written back to the RS and ROB, to be used as early as the next cycle by the RS. This high execution bandwidth enables execution bursts to keep up with the functional expansion of the micro-fused micro-ops that are decoded and retired.

The execution core contains the following three execution stacks:

- SIMD integer.
- Regular integer.
- x87/SIMD floating-point.

The execution core also contains connections to and from the memory cluster. See Figure 3-2.

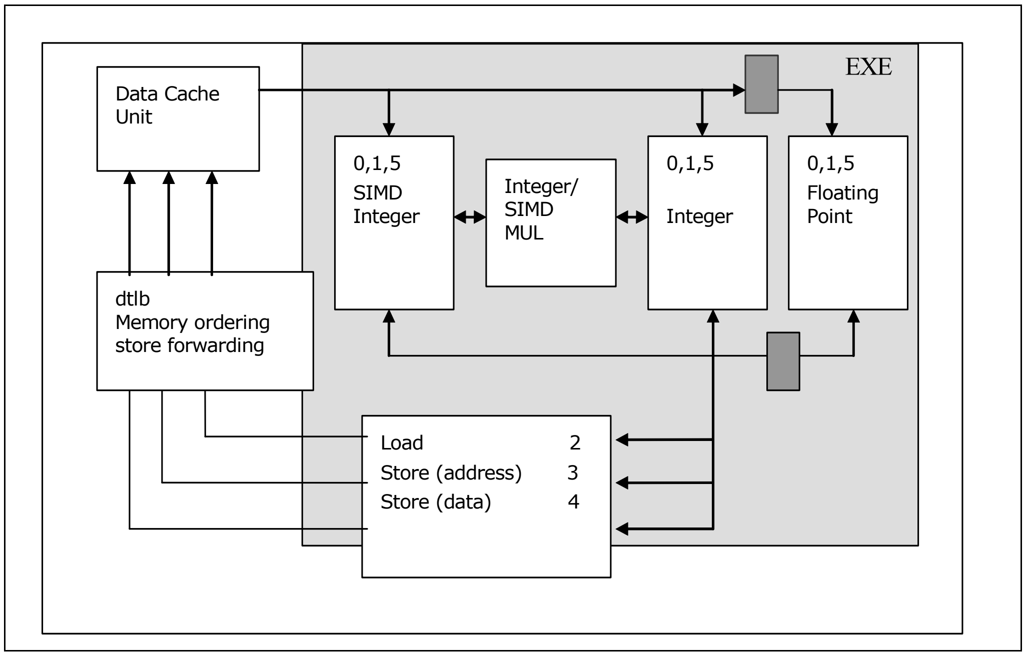


Figure 3-2. Execution Core of Intel Core Microarchitecture

Notice that the two dark squares inside the execution block (in grey color) and appear in the path connecting the integer and SIMD integer stacks to the floating-point stack. This delay shows up as an extra cycle called a bypass delay. Data from the L1 cache has one extra cycle of latency to the floating-point unit. The dark-colored squares in Figure 3-2 represent the extra cycle of latency.

3.1.3 Intel® Advanced Memory Access

The Intel Core microarchitecture contains an instruction cache and a first-level data cache in each core. The two cores share a 2 or 4-MByte L2 cache. All caches are writeback and non-inclusive. Each core contains:

- **L1 data cache, known as the data cache unit (DCU)** — The DCU can handle multiple outstanding cache misses and continue to service incoming stores and loads. It supports maintaining cache coherency. The DCU has the following specifications:
  - 32-KBytes size.
  - 8-way set associative.
  - 64-bytes line size.
- **Data translation lookaside buffer (DTLB)** — The DTLB in Intel Core microarchitecture implements two levels of hierarchy. Each level of the DTLB have multiple entries and can support either 4-KByte pages or large pages. The entries of the inner level (DTLB0) is used for loads. The entries in the outer level (DTLB1) support store operations and loads that missed DTLB0. All entries are 4-way associative. Here is a list of entries in each DTLB:

  - DTLB1 for large pages: 32 entries.
  - DTLB1 for 4-KByte pages: 256 entries.
  - DTLB0 for large pages: 16 entries.
  - DTLB0 for 4-KByte pages: 16 entries.

  An DTLB0 miss and DTLB1 hit causes a penalty of 2 cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the DTLB1 and PMH are largely non-blocking due to the design of Intel Smart Memory Access.

- **Page miss handler (PMH)**
- **A memory ordering buffer (MOB)** — Which:
  - Enables loads and stores to issue speculatively and out of order.
  - Ensures retired loads and stores have the correct data upon retirement.
  - Ensures loads and stores follow memory ordering rules of the Intel 64 and IA-32 architectures.

The memory cluster of the Intel Core microarchitecture uses the following to speed up memory operations:

- 128-bit load and store operations.
- Data prefetching to L1 caches.
- Data prefetch logic for prefetching to the L2 cache.
- Store forwarding.
- Memory disambiguation.
- 8 fill buffer entries.
- 20 store buffer entries.
- Out of order execution of memory operations.
- Pipelined read-for-ownership operation (RFO).

For information on optimizing software for the memory cluster, see Section 3.6, "Optimizing Memory Accesses."

3.1.3.1 Loads and Stores

The Intel Core microarchitecture can execute up to one 128-bit load and up to one 128-bit store per cycle, each to different memory locations. The microarchitecture enables execution of memory operations out of order with respect to other instructions and with respect to other memory operations.

Loads can:

- Issue before preceding stores when the load address and store address are known not to conflict.
- Be carried out speculatively, before preceding branches are resolved.
- Take cache misses out of order and in an overlapped manner.
- Issue before preceding stores, speculating that the store is not going to be to a conflicting address.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access the uncacheable memory type.

Faulting or uncacheable loads are detected and wait until retirement, when they update the programmer visible state. x87 and floating-point SIMD loads add 1 additional clock latency.

Stores to memory are executed in two phases:

- **Execution phase** — Prepares the store buffers with address and data for store forwarding. Consumes dispatch ports, which are ports 3 and 4.

- **Completion phase** — The store is retired to programmer-visible memory. It may compete for cache banks with executing loads. Store retirement is maintained as a background task by the memory order buffer, moving the data from the store buffers to the L1 cache.

3.1.3.2 Data Prefetch to L1 caches

Intel Core microarchitecture provides two hardware prefetchers to speed up data accessed by a program by prefetching to the L1 data cache:

- **Data cache unit (DCU) prefetcher** — This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.
- **Instruction pointer (IP)- based strided prefetcher** — This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to half of a 4KB-page, or 2 KBytes.

Data prefetching works on loads only when the following conditions are met:

- Load is from writeback memory type.
- Prefetch request is within the page boundary of 4 Kbytes.
- No fence or lock is in progress in the pipeline.
- Not many other load misses are in progress.
- The bus is not very busy.
- There is not a continuous stream of stores.

DCU Prefetching has the following effects:

- Improves performance if data in large structures is arranged sequentially in the order used in the program.
- May cause slight performance degradation due to bandwidth issues if access patterns are sparse instead of local.
- On rare occasions, if the algorithm's working set is tuned to occupy most of the cache and unneeded prefetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

In contrast to hardware prefetchers relying on hardware to anticipate data traffic, software prefetch instructions relies on the programmer to anticipate cache miss traffic, software prefetch act as hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is intended for prefetching data, but not for prefetching code.

C.1.3.3 Data Prefetch Logic

Data prefetch logic (DPL) prefetches data to the second-level (L2) cache based on past request patterns of the DCU from the L2. The DPL maintains two independent arrays to store addresses from the DCU: one for upstreams (12 entries) and one for down streams (4 entries). The DPL tracks accesses to one 4K byte page in each entry. If an accessed page is not in any of these arrays, then an array entry is allocated.

The DPL monitors DCU reads for incremental sequences of requests, known as streams. Once the DPL detects the second access of a stream, it prefetches the next cache line. For example, when the DCU requests the cache lines A and A+1, the DPL assumes the DCU will need cache line A+2 in the near future. If the DCU then reads A+2, the DPL prefetches cache line A+3. The DPL works similarly for “downward” loops.

The Intel Pentium M processor introduced DPL. The Intel Core microarchitecture added the following features to DPL:

- The DPL can detect more complicated streams, such as when the stream skips cache lines. DPL may issue 2 prefetch requests on every L2 lookup. The DPL in the Intel Core microarchitecture can run up to 8 lines ahead from the load request.

- DPL in the Intel Core microarchitecture adjusts dynamically to bus bandwidth and the number of requests. DPL prefetches far ahead if the bus is not busy, and less far ahead if the bus is busy.
- DPL adjusts to various applications and system configurations.

Entries for each core in a multi-core processor are handled separately.

3.1.3.4 Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the Intel Core microarchitecture can forward the data directly from the store to the load. This process, called store to load forwarding, saves cycles by enabling the load to obtain the data directly from the store operation instead of through memory.

The following rules must be met for store to load forwarding to occur:

- The store must be the last store to that address prior to the load.
- The store must be equal or greater in size than the size of data being loaded.
- The load cannot cross a cache line boundary.
- The load cannot cross an 8-Byte boundary. 16-Byte loads are an exception to this rule.
- The load must be aligned to the start of the store address, except for the following exceptions:
  - An aligned 64-bit store may forward either of its 32-bit halves.
  - An aligned 128-bit store may forward any of its 32-bit quarters.
  - An aligned 128-bit store may forward either of its 64-bit halves.

Software can use the exceptions to the last rule to move complex structures without losing the ability to forward the subfields.

In Enhanced Intel Core microarchitecture, the alignment restrictions to permit store forwarding to proceed have been relaxed. Enhanced Intel Core microarchitecture permits store-forwarding to proceed in several situations that the succeeding load is not aligned to the preceding store. Figure 3-3 shows six situations (in gradient-filled background) of store-forwarding that are permitted in Enhanced Intel Core microarchitecture but not in Intel Core microarchitecture. The cases with backward slash background depicts store-forwarding that can proceed in both Intel Core microarchitecture and Enhanced Intel Core microarchitecture.

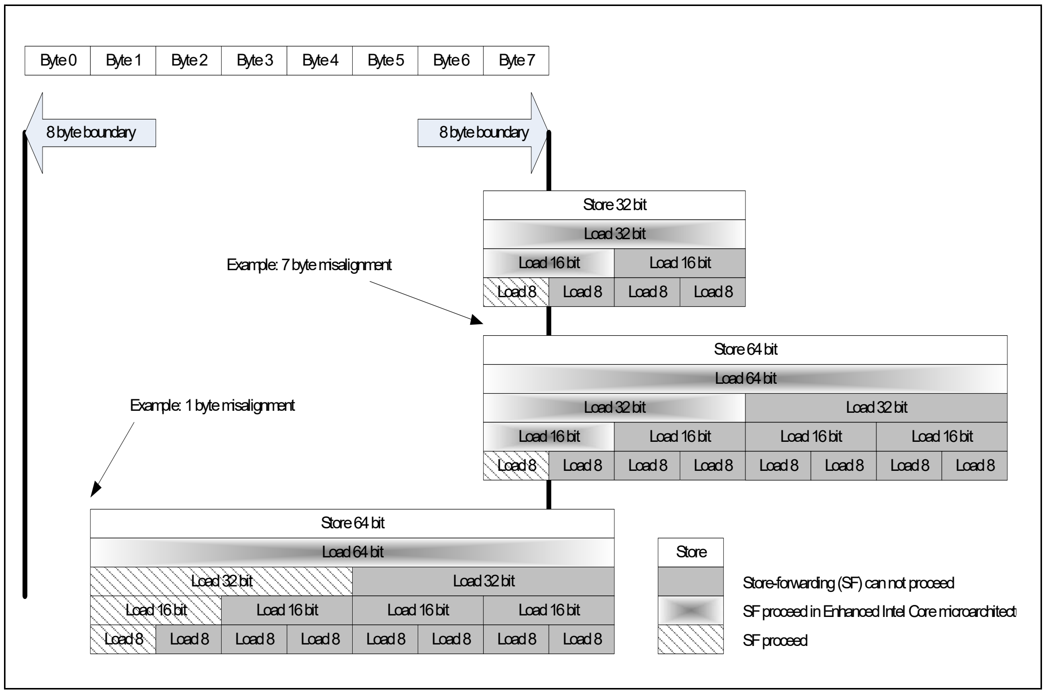


Figure 3-3. Store-Forwarding Enhancements in Enhanced Intel Core Microarchitecture

3.1.3.5 Memory Disambiguation

Refer to the "Memory Disambiguation" details in Section 2.3.2.

3.1.4 Intel® Advanced Smart Cache

The Intel Core microarchitecture optimized a number of features for two processor cores on a single die. The two cores share a second-level cache and a bus interface unit, collectively known as Intel Advanced Smart Cache. This section describes the components of Intel Advanced Smart Cache. Figure 3-4 illustrates the architecture of the Intel Advanced Smart Cache.

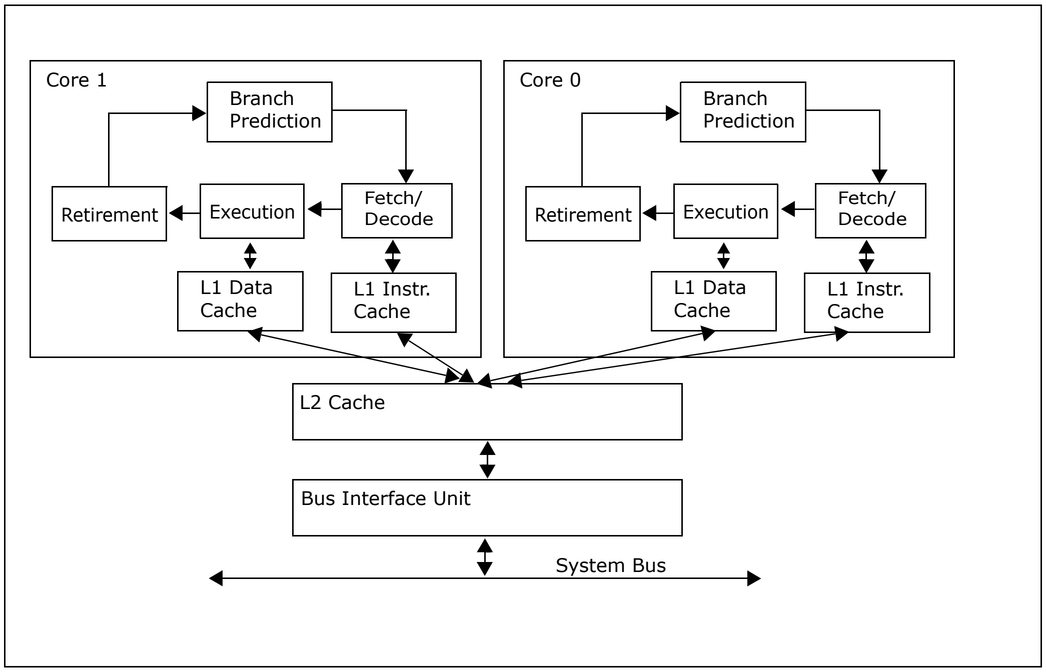


Figure 3-4. Intel Advanced Smart Cache Architecture

Table 3-3 details the parameters of caches in the Intel Core microarchitecture. For information on enumerating the cache hierarchy identification using the deterministic cache parameter leaf of CPUID instruction, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 3-3. Cache Parameters of Processors based on Intel Core Microarchitecture

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level	32 KB	8	64	3	1	Writeback
Instruction	32 KB	8	N/A	N/A	N/A	N/A
Second Level (Shared L2)[1]	2, 4 MB	8 or 16	64	14[2]	2	Writeback
Second Level (Shared L2)[3]	3, 6MB	12 or 24	64	15[2]	2	Writeback
Third Level[4]	8, 12, 16 MB	16	64	~110	12	Writeback

NOTES:

1. Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 0FH).
2. Software-visible latency will vary depending on access patterns and other factors.
3. Enhanced Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 17H or 1DH).
4. Enhanced Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 1DH).

3.1.4.1 Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for the cache line that contains this data in the caches and memory in the following order:

1. DCU of the initiating core.

2. DCU of the other core and second-level cache.
3. System memory.

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache.

Table 3-4 shows the characteristics of fetching the first four bytes of different localities from the memory cluster. The latency column provides an estimate of access latency. However, the actual latency can vary depending on the load of cache, memory components, and their parameters.

Table 3-4. Characteristics of Load and Store Operations in Intel Core Microarchitecture

Data Locality	Load		Store	
	Latency	Throughput	Latency	Throughput
DCU	3	1	2	1
DCU of the other core in modified state	14 + 5.5 bus cycles	14 + 5.5 bus cycles	14 + 5.5 bus cycles	
2nd-level cache	14	3	14	3
Memory	14 + 5.5 bus cycles + memory	Depends on bus read protocol	14 + 5.5 bus cycles + memory	Depends on bus write protocol

Sometimes a modified cache line has to be evicted to make space for a new cache line. The modified cache line is evicted in parallel to bringing the new data and does not require additional latency. However, when data is written back to memory, the eviction uses cache bandwidth and possibly bus bandwidth as well. Therefore, when multiple cache misses require the eviction of modified lines within a short time, there is an overall degradation in cache response time.

3.1.4.2 Stores

When an instruction writes data to a memory location that has WB memory type, the processor first ensures that the line is in Exclusive or Modified state in its own DCU. The processor looks for the cache line in the following locations, in the specified order:

1. DCU of initiating core.
2. DCU of the other core and L2 cache.
3. System memory.

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache. After reading for ownership is completed, the data is written to the first-level data cache and the line is marked as modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of retirement. Therefore, the store latency does not effect the store instruction itself. However, several sequential stores may have cumulative latency that can affect performance. Table 3-4 presents store latencies depending on the initial cache line location.

CHAPTER 4 NEHALEM MICROARCHITECTURE

Nehalem microarchitecture provides the foundation for many innovative features of Intel Core i7 processors and Intel Xeon processor 3400, 5500, and 7500 series. It builds on the success of 45 nm enhanced Intel Core microarchitecture and provides the following feature enhancements:

- **Enhanced processor core**
  - Improved branch prediction and recovery from misprediction.
  - Enhanced loop streaming to improve front end performance and reduce power consumption.
  - Deeper buffering in out-of-order engine to extract parallelism.
  - Enhanced execution units to provide acceleration in CRC, string/text processing and data shuffling.
- **Hyper-Threading Technology**
  - Provides two hardware threads (logical processors) per core.
  - Takes advantage of 4-wide execution engine, large L3, and massive memory bandwidth.
- **Smart Memory Access**
  - Integrated memory controller provides low-latency access to system memory and scalable memory bandwidth.
  - New cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic.
  - Two level TLBs and increased TLB size.
  - Fast unaligned memory access.
- **Dedicated Power management Innovations**
  - Integrated microcontroller with optimized embedded firmware to manage power consumption.
  - Embedded real-time sensors for temperature, current, and power.
  - Integrated power gate to turn off/on per-core power consumption.
  - Versatility to reduce power consumption of memory, link subsystems.

Westmere microarchitecture is a 32 nm version of Nehalem microarchitecture. All of the features of latter also apply to the former.

4.0.1 Microarchitecture Pipeline

Nehalem microarchitecture continues the four-wide microarchitecture pipeline pioneered by the 65nm Intel Core microarchitecture. Figure 4-1 illustrates the basic components of the pipeline of Nehalem microarchitecture as implemented in Intel Core i7 processor, only two of the four cores are sketched in the Figure 4-1 pipeline diagram.

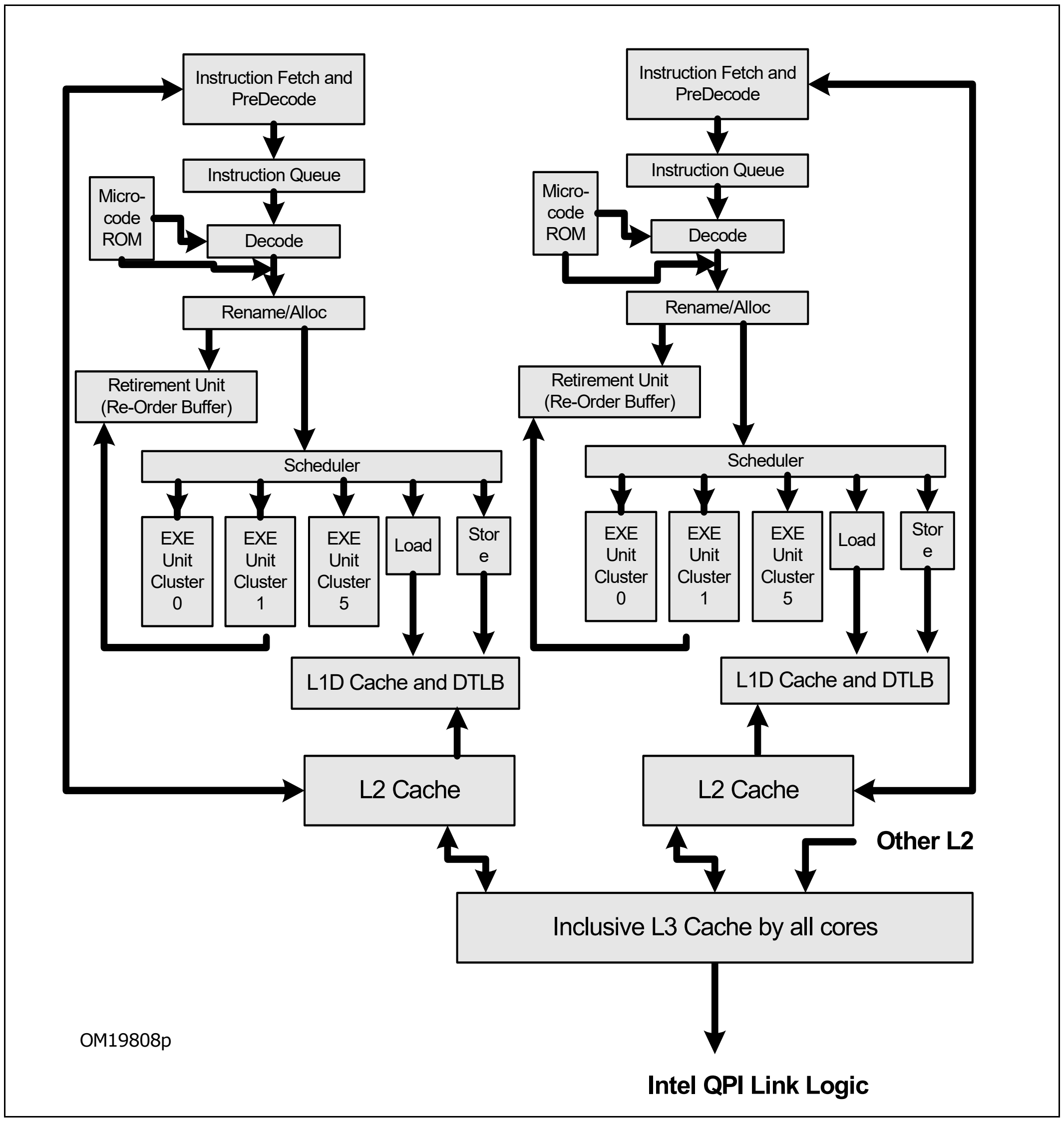


Figure 4-1. Nehalem Microarchitecture Pipeline Functionality

The length of the pipeline in Nehalem microarchitecture is two cycles longer than its predecessor in the 45 nm Intel Core 2 processor family, as measured by branch misprediction delay. The front end can decode up to 4 instructions in one cycle and supports two hardware threads by decoding the instruction streams between two logical processors in alternate cycles. The front end includes enhancement in branch handling, loop detection, MSROM throughput, etc. These are discussed in subsequent sections.

The scheduler (or reservation station) can dispatch up to six micro-ops in one cycle through six issue ports (five issue ports are shown in Figure 4-1; store operation involves separate ports for store address and store data but is depicted as one in the diagram).

The out-of-order engine has many execution units that are arranged in three execution clusters shown in Figure 4-1. It can retire four micro-ops in one cycle, same as its predecessor.

D.1 FRONT END OVERVIEW

Figure 4-2 depicts the key components of the front end of the microarchitecture. The instruction fetch unit (IFU) can fetch up to 16 bytes of aligned instruction bytes each cycle from the instruction cache to the instruction length decoder (ILD). The instruction queue (IQ) buffers the ILD-processed instructions and can deliver up to four instructions in one cycle to the instruction decoder.

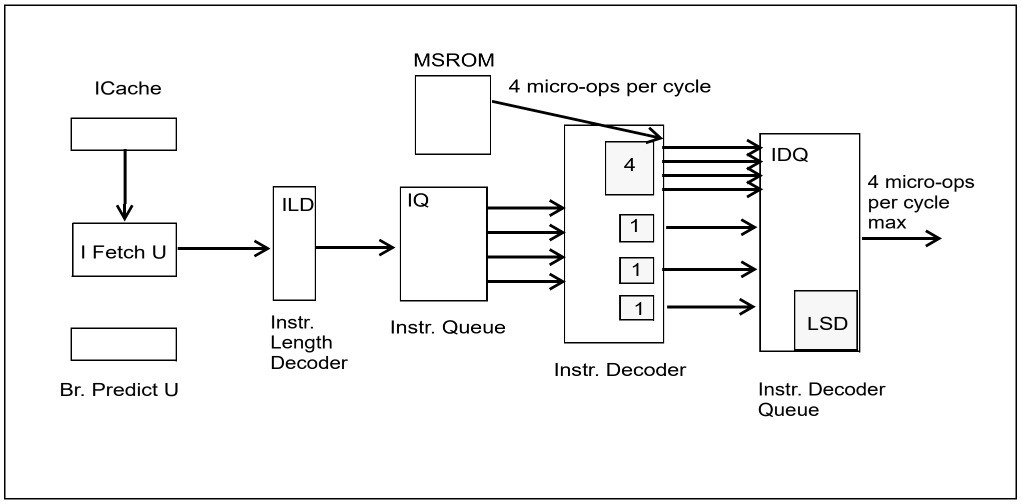


Figure 4-2. Front End of Nehalem Microarchitecture

The instruction decoder has three decoder units that can decode one simple instruction per cycle per unit. The other decoder unit can decode one instruction every cycle, either simple instruction or complex instruction made up of several micro-ops. Instructions made up of more than four micro-ops are delivered from the MSROM. Up to four micro-ops can be delivered each cycle to the instruction decoder queue (IDQ).

The loop stream detector is located inside the IDQ to improve power consumption and front end efficiency for loops with a short sequence of instructions.

The instruction decoder supports micro-fusion to improve front end throughput, increase the effective size of queues in the scheduler and re-order buffer (ROB). The rules for micro-fusion are similar to those of Intel Core microarchitecture.

The instruction queue also supports macro-fusion to combine adjacent instructions into one micro-ops where possible. In previous generations of Intel Core microarchitecture, macro-fusion support for CMP/Jcc sequence is limited to the CF and ZF flag, and macro-fusion is not supported in 64-bit mode.

In Nehalem microarchitecture, macro-fusion is supported in 64-bit mode, and the following instruction sequences are supported:

- CMP or TEST can be fused when comparing (unchanged):

  REG-REG. For example: CMP EAX,ECX; JZ label
  REG-IMM. For example: CMP EAX,0x80; JZ label
  REG-MEM. For example: CMP EAX,[ECX]; JZ label
  MEM-REG. For example: CMP [EAX],ECX; JZ label

- TEST can fused with all conditional jumps (unchanged).

- CMP can be fused with the following conditional jumps. These conditional jumps check carry flag (CF) or zero flag (ZF). The list of macro-fusion-capable conditional jumps are (unchanged):

  JA or JNBE
  JAE or JNB or JNC
  JE or JZ
  JNA or JBE
  JNAE or JC or JB
  JNE or JNZ

- CMP can be fused with the following conditional jumps in Nehalem microarchitecture (this is an enhancement):

  JL or JNGE
  JGE or JNL
  JLE or JNG
  JG or JNLE

The hardware improves branch handling in several ways. Branch target buffer has increased to increase the accuracy of branch predictions. Renaming is supported with return stack buffer to reduce mispredictions of return instructions in the code. Furthermore, hardware enhancement improves the handling of branch misprediction by expediting resource reclamation so that the front end would not be waiting to decode instructions in an architected code path (the code path in which instructions will reach retirement) while resources were allocated to executing mispredicted code path. Instead, new micro-ops stream can start forward progress as soon as the front end decodes the instructions in the architected code path.

4.2 EXECUTION ENGINE

The IDQ (Figure 4-2) delivers micro-op stream to the allocation/renaming stage (Figure 4-1) of the pipeline. The out-of-order engine supports up to 128 micro-ops in flight. Each micro-ops must be allocated with the following resources: an entry in the re-order buffer (ROB), an entry in the reservation station (RS), and a load/store buffer if a memory access is required.

The allocator also renames the register file entry of each micro-op in flight. The input data associated with a micro-op are generally either read from the ROB or from the retired register file.

The RS is expanded to 36 entry deep (compared to 32 entries in previous generation). It can dispatch up to six micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, each cluster may contain a collection of integer/FP/SIMD execution units.

The result from the execution unit executing a micro-op is written back to the register file, or forwarded through a bypass network to a micro-op in-flight that needs the result. Nehalem microarchitecture can support write back throughput of one register file write per cycle per port. The bypass network consists of three domains of integer/FP/SIMD. Forwarding the result within the same bypass domain from a producer micro-op to a consumer micro is done efficiently in hardware without delay. Forwarding the result across different bypass domains may be subject to additional bypass delays. The bypass delays may be visible to software in addition to the latency and throughput characteristics of individual execution units. The bypass delays between a producer micro-op and a consumer micro-op across different bypass domains are shown in Table 4-1.

Table 4-1. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

	FP	Integer	SIMD
FP	0	2	2
Integer	2	0	1
SIMD	2	1	0

4.2.1 Issue Ports and Execution Units

Table 4-2 summarizes the key characteristics of the issue ports and the execution unit latency/throughputs for common operations in the microarchitecture.

Table 4-2. Issue Ports of Nehalem Microarchitecture

Port	Executable operations	Latency	Throughput	Domain	Comment
Port 0	Integer ALU	1	1	Integer	
	Integer Shift	1	1		
Port 0	Integer SIMD ALU	1	1	SIMD	
	Integer SIMD Shuffle	1	1		
Port 0	Single-precision (SP) FP MUL	4	1	FP	
	Double-precision FP MUL	5	1		
	FP MUL (X87)				
	FP/SIMD/SSE2 Move and Logic	5 1	1 1		
	FP Shuffle				
	DIV/SQRT	1	1		
Port 1	Integer ALU	1	1	Integer	
	Integer LEA	1	1		
	Integer Mul	3	1		
Port 1	Integer SIMD MUL	1	1	SIMD	
	Integer SIMD Shift	1	1		
	PSAD	3	1		
	StringCompare				
Port 1	FP ADD	3	1	FP	
Port 2	Integer loads	4	1	Integer	
Port 3	Store address	5	1	Integer	
Port 4	Store data			Integer	
Port 5	Integer ALU	1	1	Integer	
	Integer Shift	1	1		
	Jmp	1	1		
Port 5	Integer SIMD ALU	1	1	SIMD	
	Integer SIMD Shuffle	1	1		

Table 4-2. Issue Ports of Nehalem Microarchitecture (Contd.)

Port	Executable operations	Latency	Throughput	Domain	Comment
Port 5	FP/SIMD/SSE2 Move and Logic	1	1	FP	

4.3 CACHE AND MEMORY SUBSYSTEM

Nehalem microarchitecture contains an instruction cache, a first-level data cache and a second-level unified cache in each core (see Figure 4-1). Each physical processor may contain several processor cores and a shared collection of sub-systems that are referred to as "uncore". Specifically in Intel Core i7 processor, the uncore provides a unified third-level cache shared by all cores in the physical processor, Intel QuickPath Interconnect links and associated logic. The L1 and L2 caches are writeback and non-inclusive.

The shared L3 cache is writeback and inclusive, such that a cache line that exists in either L1 data cache, L1 instruction cache, unified L2 cache also exists in L3. The L3 is designed to use the inclusive nature to minimize snoop traffic between processor cores. Table 4-3 lists characteristics of the cache hierarchy. The latency of L3 access may vary as a function of the frequency ratio between the processor and the uncore sub-system.

Table 4-3. Cache Parameters of Intel Core i7 Processors

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level Data	32 KB	8	64	4	1	Writeback
Instruction	32 KB	4	N/A	N/A	N/A	N/A
Second Level	256KB	8	64	10[1]	Varies	Writeback
Third Level (Shared L3)[2]	8MB	16	64	35-40+[2]	Varies	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.

Nehalem microarchitecture implements two levels of translation lookaside buffer (TLB). The first level consists of separate TLBs for data and code. DTLB0 handles address translation for data accesses, it provides 64 entries to support 4KB pages and 32 entries for large pages. The ITLB provides 64 entries (per thread) for 4KB pages and 7 entries (per thread) for large pages.

The second level TLB (STLB) handles both code and data accesses for 4KB pages. It support 4KB page translation operation that missed DTLB0 or ITLB. All entries are 4-way associative. Here is a list of entries in each DTLB:

- STLB for 4-KByte pages: 512 entries (services both data and instruction look-ups).
- DTLB0 for large pages: 32 entries.
- DTLB0 for 4-KByte pages: 64 entries.

An DTLB0 miss and STLB hit causes a penalty of 7cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the STLB and PMH are largely non-blocking.

4.3.1 Load and Store Operation Enhancements

The memory cluster of Nehalem microarchitecture provides the following enhancements to speed up memory operations:

- Peak issue rate of one 128-bit load and one 128-bit store operation per cycle.
- Deeper buffers for load and store operations: 48 load buffers, 32 store buffers and 10 fill buffers.
- Fast unaligned memory access and robust handling of memory alignment hazards.
- Improved store-forwarding for aligned and non-aligned scenarios.
- Store forwarding for most address alignments.

4.3.1.1 Efficient Handling of Alignment Hazards

The cache and memory subsystems handles a significant percentage of instructions in every workload. Different address alignment scenarios will produce varying performance impact for memory and cache operations. For example, 1-cycle throughput of L1 (see Table 4-4) generally applies to naturally-aligned loads from L1 cache. But using unaligned load instructions (e.g. MOVUPS, MOVUPD, MOVDQU, etc.) to access data from L1 will experience varying amount of delays depending on specific microarchitectures and alignment scenarios.

Table 4-4. Performance Impact of Address Alignments of MOVDQU from L1

Throughput (cycle)	**Intel Core i7 Processor**	**45 nm Intel Core Microarchitecture**	**65 nm Intel Core Microarchitecture**
Alignment Scenario	**06_1AH**	**06_17H**	**06_0FH**
16B aligned	1	2	2
Not-16B aligned, not cache split	1	~2	~2
Split cache line boundary	~4.5	~20	~20

Table 4-4 lists approximate throughput of issuing MOVDQU instructions with different address alignment scenarios to load data from the L1 cache. If a 16-byte load spans across cache line boundary, previous microarchitecture generations will experience significant software-visible delays.

Nehalem microarchitecture provides hardware enhancements to reduce the delays of handling different address alignment scenarios including cache line splits.

4.3.1.2 Store Forwarding Enhancement

When a load follows a store and reloads the data that the store writes to memory, the microarchitecture can forward the data directly from the store to the load in many cases. This situation, called store to load forwarding, saves several cycles by enabling the load to obtain the data directly from the store operation instead of through the memory system.

Several general rules must be met for store to load forwarding to proceed without delay:

- The store must be the last store to that address prior to the load.
- The store must be equal or greater in size than the size of data being loaded.
- The load data must be completely contained in the preceding store.

Specific address alignment and data sizes between the store and load operations will determine whether a store-forward situation may proceed with data forwarding or experience a delay via the cache/memory sub-system. The 45 nm Enhanced Intel Core microarchitecture offers more flexible address alignment and data sizes requirement than previous microarchitectures. Nehalem microarchitecture offers additional enhancement with allowing more situations to forward data expeditiously.

The store-forwarding situations for with respect to store operations of 16 bytes are illustrated in Figure 4-3.

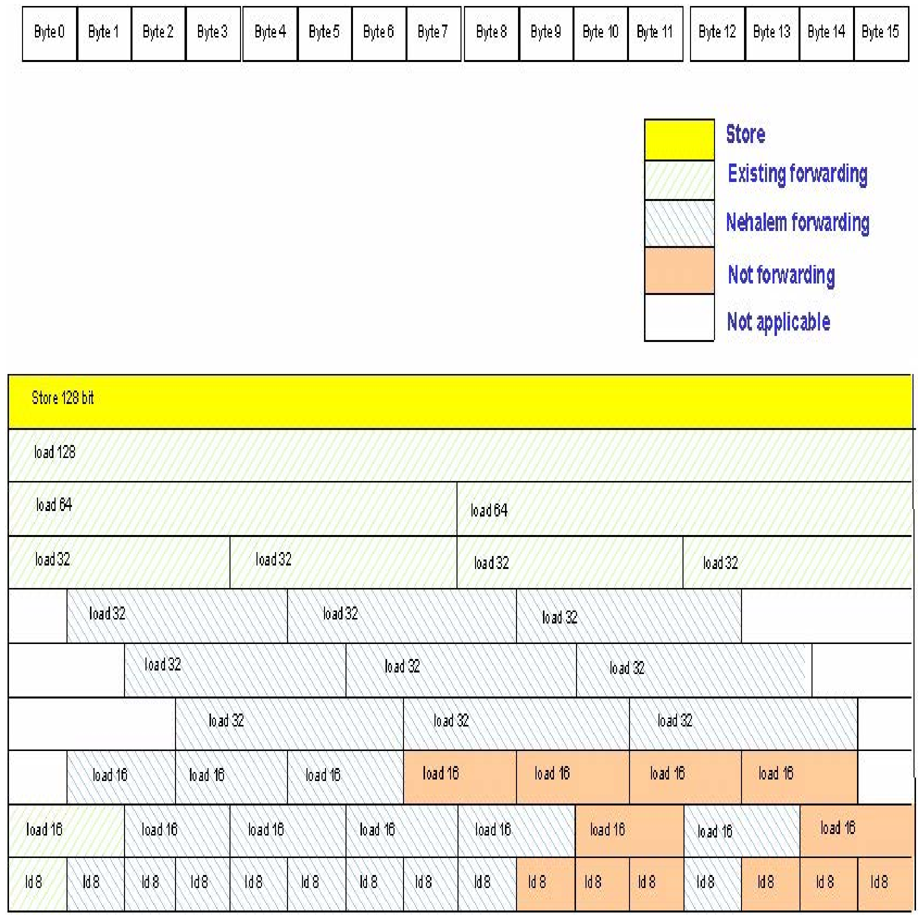


Figure 4-3. Store-Forwarding Scenarios of 16-Byte Store Operations

Nehalem microarchitecture allows store-to-load forwarding to proceed regardless of store address alignment (The white space in the diagram does not correspond to an applicable store-to-load scenario). Figure 4-4 illustrates situations for store operation of 8 bytes or less.

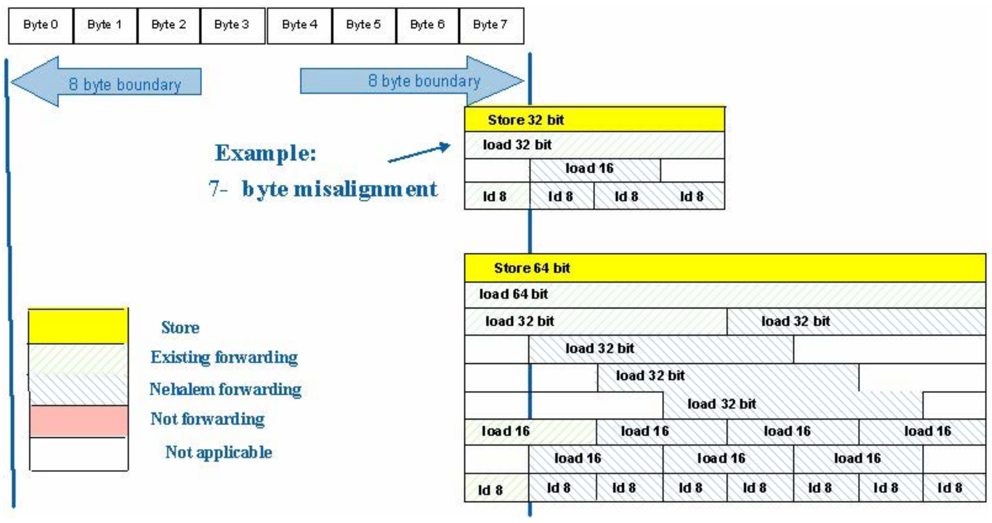


Figure 4-4. Store-Forwarding Enhancement in Nehalem Microarchitecture

4.4 REP STRING ENHANCEMENT

REP prefix in conjunction with MOVS/STOS instruction and a count value in ECX are frequently used to implement library functions such as memcpy()/memset(). These are referred to as "REP string" instructions. Each iteration of these instruction can copy/write constant a value in byte/word/dword/qword granularity The performance characteristics of using REP string can be attributed to two components: startup overhead and data transfer throughput.

The two components of performance characteristics of REP String varies further depending on granularity, alignment, and/or count values. Generally, MOVSB is used to handle very small chunks of data. Therefore, processor implementation of REP MOVSB is optimized to handle ECX < 4. Using REP MOVSB with ECX > 3 will achieve low data throughput due to not only byte-granular data transfer but also additional startup overhead. The latency for MOVSB, is 9 cycles if ECX < 4; otherwise REP MOVSB with ECX >9 have a 50-cycle startup cost.

For REP string of larger granularity data transfer, as ECX value increases, the startup overhead of REP String exhibit step-wise increase:

- Short string (ECX <= 12): the latency of REP MOVSW/MOVSD/MOVSQ is about 20 cycles.
- Fast string (ECX >= 76: excluding REP MOVSB): the processor implementation provides hardware optimization by moving as many pieces of data in 16 bytes as possible. The latency of REP string latency will vary if one of the 16-byte data transfer spans across cache line boundary:
  - Split-free: the latency consists of a startup cost of about 40 cycles and each 64 bytes of data adds 4 cycles.
  - Cache splits: the latency consists of a startup cost of about 35 cycles and each 64 bytes of data adds 6cycles.
- Intermediate string lengths: the latency of REP MOVSW/MOVSD/MOVSQ has a startup cost of about 15 cycles plus one cycle for each iteration of the data movement in word/dword/qword.

Nehalem microarchitecture improves the performance of REP strings significantly over previous microarchitectures in several ways:

- Startup overhead have been reduced in most cases relative to previous microarchitecture.
- Data transfer throughput are improved over previous generation.

- In order for REP string to operate in “fast string” mode, previous microarchitectures requires address alignment. In Nehalem microarchitecture, REP string can operate in “fast string” mode even if the address is not aligned to 16 bytes.

4.4.1 Enhancements for System Software

In addition to microarchitectural enhancements that can benefit both application-level and system-level software, Nehalem microarchitecture enhances several operations that primarily benefit system software.

Lock primitives: Synchronization primitives using the Lock prefix (e.g. XCHG, CMPXCHG8B) executes with significantly reduced latency than previous microarchitectures.

VMM overhead improvements: VMX transitions between a Virtual Machine (VM) and its supervisor (the VMM) can take thousands of cycle each time on previous microarchitectures. The latency of VMX transitions has been reduced in processors based on Nehalem microarchitecture.

4.4.2 Efficiency Enhancements for Power Consumption

Nehalem microarchitecture is not only designed for high performance and power-efficient performance under wide range of loading situations, it also features enhancement for low power consumption while the system idles. Nehalem microarchitecture supports processor-specific C6 states, which have the lowest leakage power consumption that OS can manage through ACPI and OS power management mechanisms.

4.4.3 Intel® Hyper-Threading Technology (Intel® HT) Support in Nehalem Microarchitecture

Nehalem microarchitecture supports Intel® Hyper-Threading Technology (Intel® HT). Its implementation of Intel HT provides two logical processors sharing most execution/cache resources in each core. The HT implementation in Nehalem microarchitecture differs from previous generations of HT implementations using Intel NetBurst microarchitecture in several areas:

- Nehalem microarchitecture provides four-wide execution engine, more functional execution units coupled to three issue ports capable of issuing computational operations.
- Nehalem microarchitecture supports integrated memory controller that can provide peak memory bandwidth of up to 25.6 GB/sec in Intel Core i7 processor.
- Deeper buffering and enhanced resource sharing/partition policies:
  - Replicated resource for HT operation: register state, renamed return stack buffer, large-page ITLB.
  - Partitioned resources for HT operation: load buffers, store buffers, re-order buffers, small-page ITLB are statically allocated between two logical processors.
  - Competitively-shared resource during HT operation: the reservation station, cache hierarchy, fill buffers, both DTLB0 and STLB.
  - Alternating during Intel HT operation: front end operation generally alternates between two logical processors to ensure fairness.
  - HT unaware resources: execution units.

CHAPTER 5 KNIGHTS LANDING MICROARCHITECTURE OPTIMIZATION

Intel® Xeon Phi^{TM} Processors 7200/5200/3200 Series are based on the Knights Landing microarchitecture. Coding techniques for software targeting the Knights Landing microarchitecture are described in this chapter. Processors based on the Knights Landing microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of Chapter 2, "Intel® 64 and IA-32 Processor Architectures" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

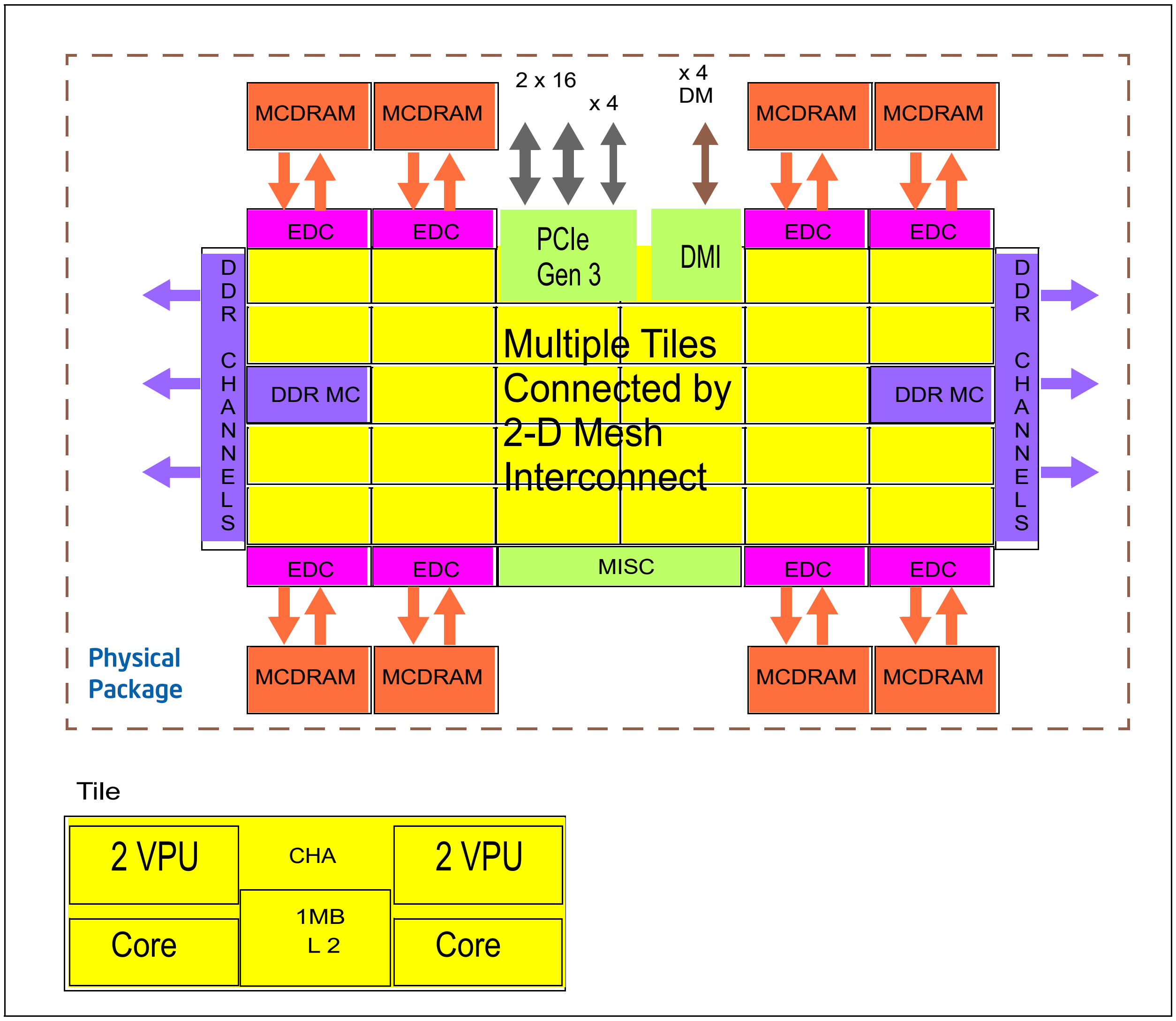


Figure 5-1. Tile-Mesh Topology of the Knights Landing Microarchitecture

The Knights Landing microarchitecture is designed for processors and co-processor product families that target highly-parallel, high-performance applications. An Intel Xeon Phi processor based on the Knights Landing microarchitecture is comprised of:

- A large number of tiles.
- A two-dimensional mesh interconnect connecting the tiles.

- An advanced memory sub-system supplying data to all the tiles containing IA-compatible processor cores and cache hierarchy.

Figure 5-1 depicts a collection of "tile" units (or pairs of processor cores) connected by a two-dimensional mesh network, offering I/O capabilities via PCIe and DMI interfaces, a memory sub-system supporting high-bandwidth optimized MCDRAM, and capacity-optimized DDR memory channels.

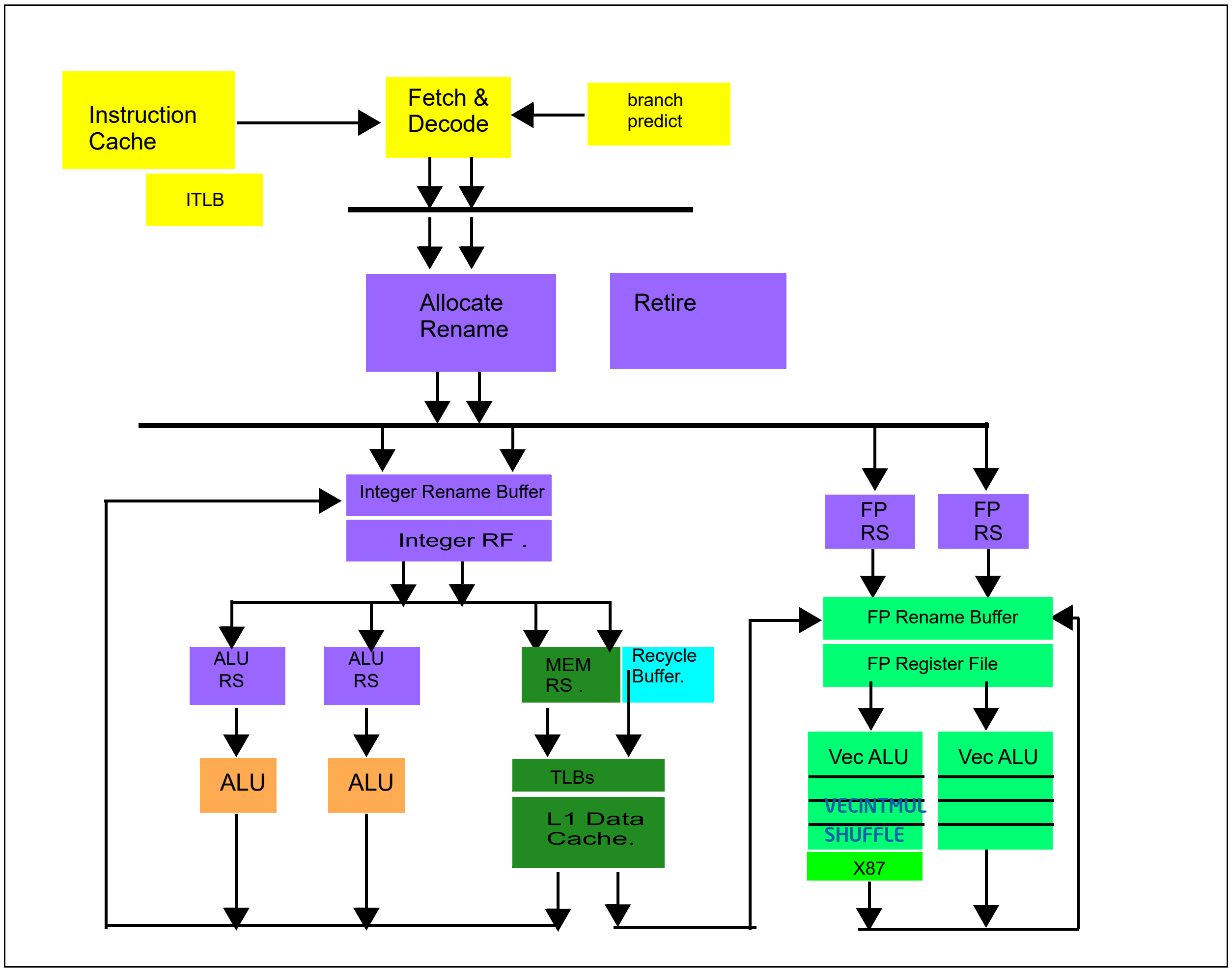


Figure 5-2. Processor Core Pipeline Functionality of the Knights Landing Microarchitecture

Figure 5-1 also illustrates each tile comprising:

- Two out-of-order IA processor cores supporting Intel® Hyper-Threading Technology (Intel® HT)with 4 logical processors per core.
- A 1 MByte L2 cache shared between the two processor cores in the tile.
- A Caching Homing Agent (CHA) connecting each tile to the 2-D mesh interconnect.
- Each processor core also provides a dedicated vector processing unit (VPU) capable of executing 512-bit, 256-bit, 128-bit and scalar SIMD instructions.

Figure 5-2 illustrates the microarchitectural pipelines of a processor core (including the VPU pipelines) inside a tile.

The processor core in the Knights Landing microarchitecture provides the following features:

- An out-of-order (OOO) execution engine with 6-wide execution (2 VPU, 2 memory, 2 integer) pipeline. Specifically, the out-of-order engine is supported by:

 - — The front end can decode two instructions per-cycle into micro-ops (uops).
 - — The allocate/rename stage is also two-wide.
 - — The out-of-order engine has distributed reservation stations (72-entry deep) feeding the integer, memory, and VPU pipelines.

- The VPU can execute Intel AVX-512F, Intel AVX-512CD, Intel AVX-512ER, Intel AVX-512PF, Intel AVX, and 128-bit SIMD/FP instructions.
- The VPU can perform two 512-bit FMA operations per cycle; x87 and MMX instructions throughput is limited to one per cycle.
- Each processor core supports 4 logical processors via Intel HT.
- Two processor cores share a 1 MByte L2 cache and form a tile.

5.1 FRONT END

The front end can fetch 16 bytes of instructions per cycle. The decoders can decode up to two instructions of not more than 24 bytes in a cycle. The decoders can only provide a single uop per instruction. If an instruction decodes into multiple uops (e.g., VSCATTER*), the microcode sequencer (MS) will supply the uop flow with a performance bubble of three to seven cycles, depending on instruction alignment in the decoder and length of the MS flow. The decoder will also have a small delay if a taken branch is encountered. If an instruction has more than three prefixes, there will be a multi-cycle bubble.

The front end is connected to the OOO execution engine through the Allocation, Renaming and Retirement cluster. Scheduling of uops is handled with distributed reservation stations across the integer, memory and VPU pipelines.

5.1.1 Out-of-Order Engine

The reorder buffer (ROB) is 72 uops deep. There are 16 store buffers (for both address and data). Distributed scheduling of uops include (see Figure 5-2):

- Two integer reservation stations (one per dispatch port) are 12 entries each.
- The single MEC reservation station has 12 entries, and dispatches up to 2 uops per cycle.
- The two VPU reservation stations (one per dispatch port) are 20 entries each.

The reservation stations, ROB, and store data buffers are hard partitioned per logical processor (depending on the processor core operating with one, two, or four active logical processors). Hard partitioning of resources changes as logical processors wake up and go to sleep. The store address buffers have two entries reserved per logical processor, with the remaining entries shared among the logical processors.

The integer reservation stations can dispatch 2 uops per cycle each, and are able to do so out-of-order. The memory execution reservation station dispatches 2 uops from its scheduler in-order, but uops can complete in any order. The data cache can read two 64B cache lines and write one cache line per cycle. The VPU reservation stations can dispatch 2 uops per cycle each and complete out-of-order.

The OOO engine in the Knights Landing microarchitecture is optimized to favor execution throughput over latency. Loads to integer registers (e.g., RAX) are 4 cycles, and loads to VPU registers (e.g., XMM0, YMM1, ZMM2, or MM0) are 5 cycles. Only one integer load is possible per cycle, but the other memory operations (store address, vector load, and prefetch) can dispatch two per cycle. Stores commit post-retirement, at a rate of 1 per cycle. The data cache and instruction caches are each 32 KB in size.

Most commonly-used integer math instructions (e.g. add, sub, cmp, test) have a throughput of 2 per cycle with latency of a single cycle. The integer pipeline has only one integer multiplier with a latency of 3 or 5 cycles depending on the operand size. Latency of integer division will vary depending on the operand size and input value; its throughput is expected to be not faster than one every ~20 cycles. Store to load forwarding has a cost of 2 cycles and can forward one per cycle if the store-forwarding restrictions are met.

Table 5-1. Integer Pipeline Characteristics of the Knights Landing Microarchitecture

Integer Instruction/operations	**Latency (cycle)**	**Throughput (cycles per instruction)**
Simple Integer	1	0.5
Integer Multiply	3 or 5	1
Integer Divide	Varies	> 20
Store to Load Forward	2	1
Integer Loads	4	1

Many VPU math operations can dispatch on either VPU port with a latency of either 2 cycles or 6 cycles; see Table 5-2. The following instructions can only dispatch on a single port:

- All x87 math operations.
- FP divisions and square roots.
- Intel AVX-512ER.
- Vector permute / shuffle operations.
- Vector to integer moves.
- Intel AVX-512CD conflict instructions.
- AESNI.
- The store data operation of a vector instruction with store semantics.

The above operations are limited to one of the two VPU dispatch pipes. Vector store data and vector to integer moves are on one dispatch pipe. The remaining single pipe instructions are on the other dispatch pipe.

Table 5-2. Vector Pipeline Characteristics of the Knights Landing Microarchitecture

Vector Instructions	Latency (cycle)	Throughput (cycles per instruction)
Simple Integer	2	0.5
Most Vector Math (including FMA)	6	0.5
Mask Instructions (operating on opmask)	2	0.5
AVX-512ER (64-bit element)	7	2
AVX-512ER (32-bit element)	8	3
Vector Loads	5	0.5
Store to Load Forward	2	0.5
Gather (8 elements)	15	5
Gather (16 elements)	19	10
Register Move (GPR -> XMM/YMM/ZMM)	2	1
Register Move (XMM/YMM/ZMM -> GPR)	4	1
DIVSS/SQRTSS[1]	25	~20
DIVSD/SQRTSD[1]	40	~33
DIVP*/SQRTP*[1]	38	~10
Shuffle/Permute (1 source operand)[1]	2	1
Shuffle/Permute (2 source operands)[1]	3	2
Convert (from/to same width)[1]	2	1
Convert (from/to different width)[1]	6	5
Common x87/MMX Instructions[1]	6	1

NOTES:

1. The physical units executing these instructions may experience additional scheduling delay due to the physical layout of the units in the VPU.

Additionally, some instructions in the Knights Landing microarchitecture will be decoded as one uop by the front end but need to expand to two operations for execution. These complex uops will have an allocation throughput of one per cycle. Examples of these instructions are:

- POP: integer load data + ESP update
- PUSH: integer store data + ESP update
- INC: add to register + update partial flags
- Gather: two VPU uops
- RET: JMP + ESP update
- CALL, DEC, LEA with 3 sources

Table 5-3 lists characteristics of the caching resources in the Knights Landing microarchitecture.

Table 5-3. Characteristics of Caching Resources

	Sets	Ways	Latency	Capacity/Comments
uTLB	8	8	1	64 4KB pages (fractured)[1]
DTLB (4KB page)	32	8	4	256 4KB pages
DTLB (2M/4M page)	16	8	4	128 2MB/4MB pages
DTLB (1GB page)	1	16	4	16 1GB pages
ITLB	1	48	4	48 4KB pages (fractured)
PDE	8	4	1	Page descriptors
L1 Data Cache	64	8	4 or 5	32 KB
Instruction Cache	64	8	4	32 KB
Shared L2 Cache	1024	16	13+L1 latency	1 MB

NOTES:

1. The uTLB and ITLB can only hold translations for 4 KB memory regions. If the relevant page is larger than 4 KB (such as 2MB or 1 GB), then the buffer holds the translation for the portion of the page that is being accessed. This smaller translation is referred to as a fractured page.

5.1.2 UnTile

In the Knights Landing microarchitecture, many tiles are connected by a mesh interconnect into a physical package; see Figure 5-1. The mesh and associated on-package components are referred to as "untile". At each mesh stop, there is a connection to the tile and a tag directory that identifies which L2 cache (if any) holds a particular cache line. There is no shared L3 cache within a physical package. Memory accesses that miss in the tile must go over the mesh to the tag directory to identify any cached copies in another tile. Cache coherence uses the MESIF protocol. If the cache line is not cached in another tile, then a request goes to memory.

MCDRAM is an on-package, high bandwidth memory subsystem that provides peak bandwidth for read traffic, but lower bandwidth for write traffic (compared to reads). The aggregate bandwidth provided by MCDRAM is higher than the off-package memory subsystem (i.e., DDR memory). DDR memory bandwidth can potentially be saturated by writes or reads alone. The achievable memory bandwidth for MCDRAM is approximately 4x - 6x of what DDR can do, depending on the mix of read and write traffic.

MCDRAM capacity supported by the Knights Landing microarchitecture is either 8 or 16 GB, depending on product-specific features. The peak MCDRAM bandwidth will vary according to the size of the installed MCDRAM. MCDRAM has higher bandwidth but lower capacity than DDR. The Maximum DDR capacity is 384 GB for the Knights Landing microarchitecture.

The physical memory in a platform comprises both MCDRAM and DDR memory; they can be partitioned in a number of different modes of operation. The commonly-used modes are summarized below.

- Cache mode: MCDRAM as a direct mapped cache and DDR is used as system memory addressable by software.
- Flat mode: MCDRAM and DDR map to disjoint addressable, system memory.
- Hybrid mode: MCDRAM is partitioned; parts of MCDRAM act as direct mapped cache, the rest of MCDRAM is directly addressable. DDR map to addressable system memory.

The configuration between tiles, tag directories and the mesh support the following modes of clustering operation for cache coherent traffic:

- All-to-All: the requesting core, tag directory and memory controller for a cache line can be anywhere in the mesh.
- Quadrant: the tag directory and memory that it monitors are in the same quadrant of the mesh, but the requesting core can be anywhere in the mesh.
- Sub-NUMA Clustering (SNC): In SNC mode, BIOS expose each quadrant as a NUMA node. This requires software to recognize the NUMA domains and co-locate the requesting core, tag directory, and memory controller in the same quadrant of the mesh to realize the benefit of optimal cache miss latency.

If critical portions of an application working set fit in the capacity of MCDRAM, performance could benefit greatly by allocating it into the MCDRAM and using flat or hybrid mode. Cache mode is generally best for code that has not yet been optimized for the Knights Landing microarchitecture, and has a working set that MCDRAM can cache.

In general, cache miss latency in All-to-All mode will be worse than it is in Quadrant mode; SNC mode can achieve the best latency. Quadrant mode is the default mesh configuration. SNC clustering requires some support from software to recognize the different NUMA nodes. If DDR is not populated evenly (e.g., missing DIMMs), the mesh will need to use the All-to-All clustering mode.

When multiple tiles read the same cache line, each tile might have a copy of the cache line. If both cores in the same tile read a cache line, there will only be a single copy in the L2 cache of that tile.

If MCDRAM is configured as a cache, it can hold data or instructions accessed by the cores in a single place. If multiple tiles request the same line, only one MCDRAM cacheline will be used.

L1 data cache has higher bandwidth and lower latency than L2 cache. Cache line access from L2 has higher bandwidth and lower latency than access from memory.

MCDRAM and DDR memory have different latency and throughput profiles. This becomes important when choosing between cache vs. flat or other memory modes. In most memory configurations, the DDR capacity will be substantially larger than MCDRAM capacity. Likewise, MCDRAM capacity should be much larger than the combined L2 cache.

Working sets that fit in MCDRAM capacity, but not in the L2 cache, should be in MCDRAM. Large or rarely accessed structures should migrate to DDR. In Knights Landing microarchitecture, hardware will try to do this dynamically if MCDRAM is put in cache or hybrid memory modes. If memory is in the flat memory mode, data structures are bound to one memory or the other (MCDRAM or DDR) at allocation time. The programmer should strive to maximize the number of memory access that go to MCDRAM. One possible algorithm would allocate data structures into MCDRAM if they are frequently accessed, and have working sets that do not fit into the tile caches.

In cache memory mode, the MCDRAM access is done first. If the cacheline is not in MCDRAM, the DDR access begins. Because of this, the perceived memory access latency of DDR in cache memory mode is higher than in flat memory mode.

5.2 INTEL® AVX-512 CODING RECOMMENDATIONS FOR KNIGHTS LANDING MICROARCHITECTURE

The Intel AVX-512 family comprises a collection of instruction set extensions. For an overview and detailed features (EVEX prefix encoding, opmask support, etc.) of the Intel AVX-512 family of instructions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. Intel Xeon Phi processors (7200, 5200, 3100 series) based on the Knights Landing microarchitecture support AVX-512 Foundation (AVX-512F), AVX-512 Exponential and Reciprocal (AVX-512ER), AVX-512 Conflict (AVX-512CD), and AVX-512 Prefetch extensions. Intel AVX and Intel AVX2 instructions are also supported on processors based on the Knights Landing microarchitecture. Prior generation Intel Xeon Phi processors (7100, 5100, 3100 series) do not support Intel AVX-512, Intel AVX2, nor Intel AVX instructions.

5.2.1 Using Gather and Scatter Instructions

Gather instructions in Intel AVX-512F are enhanced over those in Intel AVX2, performing 512-bit operations (either 16 elements of 32-bit data or 8 elements of 64-bit data) and using an opmask register as writemask for conditional updates of fetched elements to the destination ZMM register.

Scatter instructions in Intel AVX-512F selectively store elements in a ZMM register to memory locations expressed via an index vector. Conditional store to the destination location is selected using an opmask register. Scatter instructions are not supported in Intel AVX or Intel AVX2.

Consider the following C code fragment:

```
    for (uint32 i = 0; i < 16; i ++) {
     b[i] = a[indirect[i]];
     // vector compute sequence
}
```

Example 5-1. Gather Comparison Between Intel® AVX-512F and Intel® AVX2

AVX-512F	AVX2
vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k0, k0; prepare mask vpgatherdd zmm2{k1}, [rax+zmm0*4] ; compute sequence using vector register	vmovdqu ymm0, [rsp+0x1000] ; load half of index vector vmovdqu ymm3, [rsp+0x1020] ; 2nd half of indirect[] vpcmpeqdd ymm4, ymm4, ymm4 ; prepare mask vmovdqa ymm1, ymm4 vpgatherdd ymm2, [rax+ymm0*4], ymm1 vpgatherdd ymm5, [rax+ymm3*4], ymm4 ; compute sequence using vector register

When using VGATHER and VSCATTER, you often need to set a mask to all ones. An efficient instruction to do this is KXNOR of a mask register with itself. Since VSCATTER and VGATHER clear their mask as the last thing they do, a loop carried dependence from the VGATHER to KXNOR can be generated. Because of this, it is wise to avoid using the same mask for source and destination in KXNOR. Since it is rare for the k0 mask to be used as a destination, it is likely that "KXNORW k1, k0, k0" will be faster than "KXNOR k1, k1, k1".

Gather and Scatter instructions in AVX-512F are different from those in prior generation Intel Xeon Phi processors (abbreviated by "Previous Generation" in Example 5-2).

Example 5-2. Gather Comparison Between Intel® AVX-512F and Previous Generation Equivalent

AVX-512F	Previous Generation Equivalent Sequence
vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k0, k0; prepare mask vpgatherdd zmm2{k1}, [rax+zmm0*4] ; compute sequence using vector register	vmovdqu zmm0, [rsp+0x1000] ; load indirect[] kxnor k1,k1 ; prepare mask g_loop: ; verify gathered elements are complete vpgatherdd zmm2{k1}, [rax+zmm0*4] jknzd k1, g_loop ; gather latency exposure ; compute sequence using vector register

5.2.2 Using Enhanced Reciprocal Instructions

The Intel AVX-512ER instructions provide high precision approximations of exponential, reciprocal, and reciprocal square root functions. The approximate math instructions in Intel AVX-512ER provide 28 bits of accuracy, compared to 11 bits in RCPSS or 14 bits with VRCP14SS. Intel AVX-512ER can reduce execution time for iterative algorithms like Newton-Raphson. Example 5-3 contains sample code using the Newton-Raphson algorithm to compute a single 32b float division with VRCP28SS. Both values are read off the stack. Note the use of rounding mode overrides on some of the math operations.

Example 5-3. Using VRCP28SS for 32-bit Floating-Point Division

```
vgetmantss    xmm18, xmm18, [rsp+0x10], 0
vgetmantss    xmm20, xmm20, [rsp+0x8], 0
vrcp28ss      xmm19, xmm18, xmm18
vgetexpss     xmm16, xmm16, [rsp+0x8]
vgetexpss     xmm17, xmm17, [rsp+0x10]
vsubss        xmm22, xmm16, xmm17
vmulss        xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss  xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss   xmm21, xmm19, xmm20
vscalefss     xmm0, xmm21, xmm22
```

5.3 USING AVX-512CD INSTRUCTIONS

Refer to Section 18.16, "Conflict Detection" for details on using the Intel AVX-512 Conflict Detection instructions.

5.3.1 Using Intel® Hyper-Threading Technology (Intel® HT)

The Knights Landing microarchitecture supports 4 logical processors with each processor core. There are choices that highly-threaded software may need to consider with respect to:

- Maximizing per-thread performance by providing maximum per-core resources to one logical processor per core.
- Maximizing per-core throughput by allowing multiple logical processors to execute on a processor core.

As thread count per core grows to 2 or 4, some applications will have higher per core performance, but lower per thread performance. If an application can perfectly scale its performance to an arbitrary number of threads, 4 threads per core is likely to have the highest instruction throughput. Practical limitations on memory capacity or parallelism may limit the number of threads per core.

In Knights Landing microarchitecture, some per core resources (like the ROB or scheduler) are partitioned to one for each of 4 logical processors. Because of this, a 3 thread configuration will have fewer aggregate resources available than 1, 2, or 4 threads per core. Placing 3 threads on a processor core is unlikely to perform better than 2 or 4 threads per core.

5.3.2 Front End Considerations

To ensure front end restrictions are not typically a performance limiter, software should consider the following:

- MSROM instructions should be avoided if possible. A good example is the memory form of CALL near indirect. It will often be better to perform a load into a register and then perform the register version of CALL. Additional examples are shown in Table 5-4.
- The total length of the instruction bytes that can be decoded each cycle is at most 16 bytes per cycle with instructions not more than 8 bytes in length. For instruction length exceeding 8 bytes, only one instruction per cycle is decoded on decoder 0. Vector instructions which address memory using 32-bit displacement can cause the decoder to limit performance.
- Instructions with multiple prefixes can restrict decode throughput. The restriction is on the length of bytes combining prefixes and escape bytes. There is a 3 cycle penalty when the escape/prefix count

exceeds 3 with the Knights Landing microarchitecture. Only decoder 0 can decode an instruction exceeding the limit of a prefix/escape byte restriction.

- Maximum number of branches that can be decoded each cycle is 1.

5.3.3 Instruction Decoder

Some IA instructions require a lookup in the microcode sequencer ROM (MSROM) to decode into a multiple uop flow. Choosing an alternative sequence of instructions which does not require MSROM will improve performance.

Table 5-4 provides alternate non-MSROM instruction sequences that can replace an instruction that decodes from MSROM.

Table 5-4. Alternatives to MSROM Instructions

Instruction from MSROM	Recommendation for Knights Landing
CALL m16/m32/m64	Load + CALL reg
PUSH m16/m32/m64	Store + RSP update
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required

5.3.4 Branching Indirectly Across a 4GB Boundary

Another important performance consideration from a front end standpoint is branch prediction for indirect branches (indirect branch or call, or ret). For 64-bit applications, indirect branch prediction fails when the target of a branch is in a different 4GB chunk of the address space from the source. (I.e. the top 32 bits of the virtual addresses of the source and target are different). This is more likely to happen when the application is split into shared libraries. Developers can build statically to improve the locality in their code, particularly for latency-sensitive library calls that are accessed frequently. Another option is to use glibc 2.23 or later, and set the LD_PREFER_MAP_32BIT_EXEC environment variable which requests that the dynamic linker place all shared libraries at the bottom of the address space.

5.4 INTEGER EXECUTION CONSIDERATIONS

5.4.1 Flags usage

Many instructions have an implicit data result that is captured in a flags register. These results can be consumed by a variety of instructions such as conditional moves (cmovs), branches, and even a variety of logic/arithmetic operations (such as rcl). The most common instructions used in computing branch conditions are compare instructions (CMP). Branches dependent on the CMP instruction can execute in the next cycle. The same is true for branch instructions dependent on ADD or SUB instructions.

INC and DEC instructions require an additional uop to merge the flags as they are partial flag writers. As a result, an INC or a DEC instruction should be replaced by “ADD reg, 1” or “SUB reg, 1” to avoid a partial flag penalty.

Instructions that operate on 8-bit or 16-bit registers are not optimized in hardware in the Knights Landing microarchitecture. In general, it is faster to use integer instructions operating on 32-bit or 64-bit general purpose registers than 8-bit or 16-bit registers.

5.4.2 Integer Division

Integer division can be a common operation in some mathematical expressions. However, using hardware integer divide instructions is often less than optimal in performance. If the divisor is known to be relatively small (16 bits or less), there are fast SW sequences to emulate the division. If the divisor is known to be a power of 2, use SHR (division) and/or AND (remainder) instead of DIV. Division by a constant can be replaced by MUL with a constant. If the input values are highly constrained, a pre-computed lookup table is likely to provide better performance. Some examples of the techniques can be found in Section 13.2.4, "Replace 128-bit Integer Division with 128-bit Multiplication," and Section 14.5, "Numerical Data Conversion to ASCII Format."

Division instructions should be aggressively minimized by the compiler, either using the techniques mentioned earlier, or by hoisting redundant divisions out of inner loops.

5.5 OPTIMIZING FP AND VECTOR EXECUTION

5.5.1 Instruction Selection Considerations

In general, using 512-bit instructions are more favorable to achieve higher throughput than 256-bit instructions. The same applies relative to 256-bit vs. 128-bit vector instructions. 128-bit SSE instructions are likely to achieve higher throughput than using X87 instruction equivalents. Often, X87 instruction functionality (transcendental) not present in vector instruction extensions natively can be replaced by library implementations using vector instructions.

In the Knights Landing microarchitecture, COMIS* and UCOMIS* instructions (legacy, VEX, or EVEX encoding) that update EFLAGS are slow. These should be replaced by a more optimal sequence of the Intel AVX-512F version of VCMPS* and KORTEST.

Example 5-4. Replace VCOMIS* with VCMPSS/KORTEST

```
vcmpss k1, xmm1, xmm2, imm8 ; specify imm8 according to desired primitive
kortest k1, k1
```

Some instructions, like VCOMPRESS*, are single uop when writing a register, but an MS flow when writing memory. Where possible, it is much better to do a VCOMPRESS to register and then store it. Similar optimizations apply to all vector instructions that do some sort of operation followed by a store (e.g., PEXTRACT).

In the Knights Landing microarchitecture, mixing SSE instructions and Intel AVX instructions require a different set of considerations to avoid loss of performance due to intermixing of SSE and Intel AVX instructions. Replace SSE code with AVX-128 equivalents, whenever possible.

Situations that can result in a performance penalty are:

- If an Intel AVX instruction encoded with a vector length of more than 128 bits is allocated before the retirement of previous in-flight SSE instructions.
- VZEROUPPER instruction throughput is slow, and is not recommended to preface a transition to AVX code after SEE code execution. The throughput of VZEROALL is also slow. Using either the VZEROUPPER or the VZEROALL instruction is likely to result in performance loss.

Conditional packed load/store instructions, like MASKMOVDQU and VMASKMOV, use a vector register for element selection. AVX-512F instructions provide alternatives using an opmask register for element selection and are preferred over using a vector register for element selection.

Some vector math instructions require multiple uops to implement in the VPU. This increases the latency of the individual instruction beyond the standard math latencies of 2 and 6. In general, instructions that alter output/input element width (e.g., VCVTSD2SI) fall into this category. Many Intel AVX2 instructions that operate on byte and word quantities have reduced performance compared to the equivalents that operate on 32b or 64b quantities.

Some execution units in the VPU may incur scheduling delay if a sequence of dependent uop flow needs to use these execution units. When this happens, it will have an additional cost of a 2-cycle bubble. Code that frequently transition between the outlier units with other units in the VPU can experience a performance issue due to these bubbles.

Most of the Intel AVX-512 instructions support using an opmask register to make conditional updates to the destination. In general, using an opmask with all 1's will be the fastest relative to using an opmask with other non-zero values. Using a non-zero opmask value, the instruction will be similar in speed relative to an opmask with all 1s, if zeroing-the-non-updated element is selected. Using a non-zero opmask value with merging (preserving) non-updated elements of the destination will likely be slower.

Horizontal add/subtraction instructions in Intel AVX2 do not have promoted equivalents in Intel AVX-512. Horizontal reduction is best implemented using software sequences; see Example E-5.

In situations where an algorithm needs to perform reduction, reduction can often be implemented without horizontal addition.

Example E-6 shows code fragment for the inner loop of a DGEMM matrix multiplication routine, which computes the dense matrix operation of C = A * B.

In Example E-6, there are 16 partial sums. The sequence of FMA instructions make use of the two VPU capability of 2 FMAs per cycle throughput, 6 cycles latency. The FMA code snippet in Example E-6 is presented using uncompressed addressing form for the memory operand. It is important for code generators to ensure optimal code generation will make use of compressed disp8 addressing form, so that the length of each FMA instruction will be less than 8 bytes. At the end of the inner loop, the partial sums will need to be aggregated and store the result matrix C to memory.

Example E-5. Using Software Sequence for Horizontal Reduction

```
    vextractf64x4 ymm1, zmm6, 1; reduction of 16
elements
    vaddps    ymm1, ymm6, ymm1
    vpermpd ymm4, ymm1,0xff
    vpermpd ymm5, ymm1,0xaa
    vpermpd ymm3, ymm1,0x44
    vaddps      xmm1, xmm1, xmm4
    vaddps      xmm3, xmm5, xmm3
    vaddps      xmm3, xmm1, xmm3
    vpsrlq    xmm1, xmm3, 32
    vaddss xmm3, xmm1, xmm3
```

```
    vextractf64x4 ymm1, zmm6, 1; reduction of 8
elements
    vaddps    ymm1, ymm6, ymm1
    valignq ymm4, ymm1,0x3
    valignq ymm5, ymm1,0x2
    valignq ymm3, ymm1,0x1
    vaddsd ymm1, ymm1, ymm4
    vaddsd ymm3, ymm5, ymm3
    vaddsd ymm3, ymm1, ymm3
```

Example E-6. Optimized Inner Loop of DGEMM for Knights Landing Microarchitecture

```
;; matrix - matrix dense multiplication
prefetcht0  [rdi+0x400] ;; get A matrix element into L1$
vmovapd     zmm30, [rdi]
prefetcht0  [rsi+0x400] ;; get B matrix element into L1$
vfmadd231pd zmm1, zmm30, [rsi+r12]{b} ;; broadcast B elements
vfmadd231pd zmm2, zmm30, [rsi+r12+0x08]{b} ;; displacement shown in un-compressed form
vfmadd231pd zmm3, zmm30, [rsi+r12+0x10]{b}
vfmadd231pd zmm4, zmm30, [rsi+r12+0x18]{b}
vfmadd231pd zmm5, zmm30, [rsi+r12+0x20]{b}
vfmadd231pd zmm6, zmm30, [rsi+r12+0x28]{b}
vfmadd231pd zmm7, zmm30, [rsi+r12+0x30]{b}
vfmadd231pd zmm8, zmm30, [rsi+r12+0x38]{b}

prefetcht0  [rsi+0x440]     ;; pull line into the L1$
vfmadd231pd zmm9, zmm30, [rsi+r12+0x40]{b}
vfmadd231pd zmm10, zmm30, [rsi+r12+0x48]{b}
vfmadd231pd zmm11, zmm30, [rsi+r12+0x50]{b}
vfmadd231pd zmm12, zmm30, [rsi+r12+0x58]{b}
vfmadd231pd zmm13, zmm30, [rsi+r12+0x60]{b}
vfmadd231pd zmm14, zmm30, [rsi+r12+0x68]{b}
vfmadd231pd zmm15, zmm30, [rsi+r12+0x70]{b}
vfmadd231pd zmm16, zmm30, [rsi+r12+0x78]{b}
```

5.5.2 Porting Intrinsics from Previous Generation

Most intrinsics map to individual instructions of the native hardware. Some 512-bit intrinsics may provide syntax that hides the difference between AVX-512F and the 512-bit incompatible previous generation instruction set.

However, intrinsic code that is optimized to run on previous generations will likely not run optimized on the Knights Landing microarchitecture, due to differences in the underlying microarchitecture (e.g., unaligned memory access, cost differences of permutes, limitations of previous generations).

It is likely that coding an algorithm in a high level language (C/Fortran) to compile with Intel Compilers supporting Intel AVX-512F will generate more optimal code than using previous generation intrinsics.

5.5.3 Vectorization Trade-Off Estimation

Profitability of vectorization of loops written in a high-level language to use AVX-512 is an important part of optimization for compilers as well as for hand coding assembly. Estimating this for the simplest type of loop construct can be based on trip count alone. For example, a trip count of 4 or less may be difficult to realize performance gain over scalar code. With Intel AVX-512, a trip count of 16 may be the minimum to consider vectorization.

Estimation of vectorization trade-off for more elaborate loop construct requires more sophistication. The rest of this section provides an analytic approach of examining the composition within the loop body and makes use of a table of cost estimates of basic operations, Table 5-5,to derive the trade-off comparison between vectorization versus scalar code.

Table 5-5. Cycle Cost Building Blocks for Vectorization Estimate for Knights Landing Microarchitecture

Operation	Cost (cycles)	Example Code Construct
Simple scalar math	1	A*B+C, or A+B, or A*B
Load (split cacheline)	1 (2)	A[i] /* load reference to an array element */
Store (split cacheline)	1(2)	A[i] = 2;
Gather (Scatter) 8 elements	15 (20)	A[key[i]]
Gather (Scatter) 16elements	20 (25)	A[key[i]] ;
Horizontal reduction	30	sum += A[i]
Division or Square root	15	A/B

To illustrate the cost build-up approach, consider the simple loop:

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

Within the loop body, the basic operations consist of:

- Two loads (a[i], b[i]) per iteration.
- An FMA per iteration.
- For scalar version: an accumulate per loop iteration; for vectorization: a horizontal reduction at the end of the loop.

The total cost of N trips for scalar code is 4N. By comparison, the total cost for vectorized code using AVX-512 on a 64-bit data element would be 3 * Ceiling(N/8) + 30, assuming both the main loop and remainder loop (if N is not multiples of 8) are vectorized. Therefore, profitable vectorization will need a trip count of at least 9.

Consider another example involving fetching data from irregular access patterns which might take advantage of GATHER instructions:

```
for (i=0; i<N; i++) {c[i] = a[indir[i]] * K + b[i]; }
```

Within the loop body, the basic operations consist of:

- Two loads (indir[i], b[i]) per iteration.
- An FMA per iteration.
- A store per iteration.
- For scalar version: a 3rd load per loop iteration; for vectorization: one GATHER per 8 iteration.

The total cost of N trips for scalar code is 5N. By comparison, the total cost for vectorized code would be 19* Ceiling(N/8). Scalar would be faster if N < 4.

Consider an example involving fetching data from twice irregular access patterns than the previous example:

```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

- One load (ind[i]) per iteration.
- An FMA per iteration.
- A store per iteration.
- For scalar version: two more loads per loop iteration; for vectorization: two GATHERs per 8 iteration.

The total cost of N trips for scalar code is still 5N. By comparison, the total cost for vectorized code would be (15*2 + 3)* Ceiling(N/8) = 33* Ceiling(N/8). Even a relatively small profitability of vectorization will require a significantly larger trip count.

Consider the next example involving fetching data from one irregular access pattern and horizontal reduction:

```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

Scalar cost is still 5N. Cost of vectorization is now 19*Ceiling(N/8) + 30. Scalar code would be faster for N <= 13.

Consider an example of scatter with division:

```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

The scalar cost is (15+4)*N. Cost of vectorization would be (15+20+3)*Ceiling(N/8). Vectorization would be profitable for N > 2.

In the case of gather followed by scatter:

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

The cost of scalar code is 3*N, and vector code will cost (15+20+1)*Ceiling(N/8). Vectorization will not be profitable.

For a loop body that is more complex, consider the code below from a workload known as miniMD:

```
for (int k = 0; k < numneigh; k++) {
    int j = neighs[k];
    double rsq = (xtmp - x[3*j])^2 +
           (ytmp - x[3*j+1])^2 +
           (ztmp - x[3*j+2])^2;
    if (rsq < cutforcesq) {
       double sr2 = 1.0/rsq;
       double sr6 = sr2*sr2*sr2;
       double force = sr6*(sr6-0.5)*sr2;
       res1 += delx*force;
       res2 += dely*force;
       res3 += delz*force;
    }
}
```

Before considering the IF clause, there is one load, 3 gathers (strided loads of x[]), 3 subtractions and 3 multiplies. Inside the IF clause, there is one division, 8 math operations, and 3 horizontal reductions. The scalar cost is 10*numneigh + 23 * numneigh * percent_rsq_less_than_cutforcesq. The vector cost is (52+23) * Ceiling(numneigh / 8) + 3 * 30. Scalar code makes sense if numneigh < 6 or if the compiler is highly confident that the if clause is almost never taken.

For many compilers, a vectorized loop is generated, and a remainder loop is used to take care of the rest of the operations. In other words, the vectorized loop is executed floor(N/8) times, and the remainder loop is executed N mod 8 times. In that case, modify the equations above to use floor instead of ceiling to determine whether the primary loop should be vectorized. For the remainder loop, the maximum value of the loop trip count is known. If N is unknown, it is simplest to set N to half the maximum value (4 for a ZMM vector of doubles).

More sophisticated analysis is possible. For example, the building block simple math operation of 1-cycle cost in Table 5-5 covers common instruction sequences that are not blocked by a dependency chain or long latency operations. Expanding entries of the cost table can cover more complex situations.

5.6 MEMORY OPTIMIZATION

5.6.1 Data Alignment

Data access to address spanning a cache line boundary will experience a small performance hit. Access patterns that stream through memory can avoid cache line splits to make sure each 64-byte access is aligned to a cache line boundary. When loading 32-bytes of memory to YMM, do not access 64-bytes of memory with an opmask value to mask off the high 32 bytes.

Memory references crossing a 4-Kbytes boundary will incur significant cost in performance. Access patterns that stream throughput memory using 512-bit instructions have a higher rate of crossing a 4-KBytes boundary. So alignment to 64 byte will also avoid the penalty of a page split.

If possible to predict the distance in code space of the next crossing of page boundary, it can be helpful to insert a PREFETCHT1 (to L2) a few iterations ahead of the current read stream. This can also start the page translation early and permit the L2 hardware prefetcher to start fetching on the next page.

Some access patterns which might intend to use gather and scatter will always have pairs of consecutive addresses. One common example is complex numbers, where the real and imaginary parts are laid out contiguously. It is also common when w, x, y, and z information is contiguous. If the values are 32b, it is faster to gather and scatter the 32-bit elements as half as many 64-bit elements. If the numbers are 64 bits, then it is usually faster to load and insert a 128-bit element instead of gathering 64-bit elements.

5.6.2 Hardware Prefetcher

There are two types of HW prefetchers in a tile. The Instruction Pointer Prefetcher (IPP) resides in a processor core and analyzes all the accesses in the data cache and the instructions that generated the access. The prefetcher will then attempt to insert HW prefetches to the L1 cache if a strided access pattern is detected on a cacheable page. The IPP will not cross a 4k page boundary. The IPP uses the instruction address and logical processor to index into a table. For this reason, the compiler may insert NOPs into large loops (>256 B) to make instructions that access memory go into different table entries.

The L2 HW prefetcher tries to identify streaming access patterns, and can track up to 48 access patterns. A streaming access pattern touches consecutive cache lines in increasing or decreasing order - the stride detected in the L2 is always +/-1 cacheline. The 48 detectors are allocated independently of the logical processor that originated the request. Each detector looks at the accesses done within a 4 KB region. If a stream is detected, HW prefetches for later elements of the stream will be sent to the L2 cache, and if they miss, to memory. The HW prefetcher will not stream across a 4 KB boundary. If multiple access patterns are done within the same 4 KB region, the detector can get confused, and fail to detect the stream.

5.6.3 Software Prefetch

Knights Landing microarchitecture supports out-of-order execution. In general, it can hide cache miss latency better than previous generation in-order microarchitecture. Hence, programmers should not use the same aggressive approach to insert software prefetches.

With the two hardware prefetchers described in Section 5.6.2, most streaming and short stride access patterns should be detected by the hardware prefetchers. If the access pattern is streaming, a programmer might benefit from adding software prefetches beyond the current 4-KBytes page. If the access pattern is known, but non-streaming, software prefetches can be beneficial in some situations. This is especially true if the access pattern is a relatively large stride (>256 bytes), since the IPP will not fetch across a 4 KB boundary. The software prefetch will do the PMH walk to fill the TLB, and to start the memory reference early.

Generally, software prefetching into the L2 will show more benefit than L1 prefetches. A software prefetch into L1 will consume critical hardware resources (fill buffer) until the cacheline fill completes. A software prefetch into L2 does not hold those resources, and it is less likely to have a negative performance impact. If you do use L1 software prefetches, it is best if the software prefetch is serviced by hits in the L2 cache, so the length of time that the hardware resources are held is minimized.

Software prefetch instructions that are dropped will have a negative performance impact due to consuming retirement slots from an invalid address. The performance penalty of prefetching an invalid address or requiring OS privilege from user code can be very large. The performance monitoring event NUKE.ALL provides an indication of when this might be affecting your code.

5.6.4 Memory Execution Cluster

The MEC has limited capability in executing uops out-of-order. Specifically, memory uops are dispatched from the scheduler in-order, but can complete in any order. By re-arranging the order of memory instructions, performance may be improved if they make good use of the MEC’s capability.

Example 5-7 illustrates the effect of ordering the sequence of memory instructions of two read streams accessing two arrays, a[] and b[]. The left side of Example 5-7 is the optimal sequence with the 2nd vector load from b[] dispatched on cycle N+5, assuming an L1 cache hit. The right side of Example 5-7 is a naive ordering of the memory instructions, resulting in the second vector load dispatched on cycle N+8.

The right side sequence uses one more register than the left side. If the pointer loads would miss L1, the benefit of left side will be greater than what is shown in the comment.

Example 5-7. Ordering of Memory Instruction for MEC

movq r15, [rsp+0x40] ; cycle N (load &a[0]) movq r14, [rsp+0x48] ; cycle N+1 (load &b[0]) vmovups zmm1, [r15+rax*8] ; executes in cycle N+4 vmovups zmm2, [r14+rax*8] ; cycle N+5	movq r15, [rsp+0x40] ; cycle N (load &a[0]) vmovups zmm1, [r15+rax*8] ; executes in cycle N+4 movq r15, [rsp+0x48] ; cycle N+4 (load &b[0]) vmovups zmm2, [r15+rax*8] ; cycle N+8

If there are many loads in the machine, it might be possible to hoist up the pointer loads, so that there are several memory references between the pointer load and de-reference, without requiring more integer registers to be reserved.

5.6.5 Store Forwarding

Store forwarding restriction for integer execution and the MEC in the Knights Landing microarchitecture is similar to those of the Silvermont microarchitecture. The following paragraphs describes the forwarding restrictions with the VPU.

Vector, X87, and MMX loads and stores can forward (ZMM0, YMM1, XMM2, MM3, and ST4) if the stores and loads have the same memory address and the load is not larger than the store. VPU stores cannot forward to integer loads, and integer stores cannot forward to VPU loads. In either case, the load must wait until the store is post-retirement to get the value from memory.

Vector stores that use an opmask cannot be forwarded from. If your algorithm requires such behavior, you may benefit if you merge the value in a register, and then store to memory without a conditional opmask. Later loads can then forward from the merged value.

5.6.6 Way, Set Conflicts

The memory hierarchy determines forwarding requirements based on the address of the access. The L1 data cache uses address bits 11:6 to identify which cache set to use. Forwarding logic uses bits 11:0 and the size of the access to identify potential forwarding or conflicts between loads and stores. If there are many conflicts, performance could be degraded.

Many dynamic memory allocation routines (may vary by OS and compiler) will start large memory regions with the same pattern in the least significant 12 bits. If your access patterns touch many arrays with identical shapes (element size and dimensions) and similar indices, performance could degrade significantly due to set conflict. To void these set conflicts, it is beneficial for bits [11..6] of memory accesses to be different. For example, consider:

```
a = malloc(sizeof(double) * 10000);

b = malloc(sizeof(double) * 10000);

for (i=0; i < 10000; i++) {

  a[i] = b[i] + 0.5 * b[i-1]);

}
```

Very likely, in most OSes, the effective address of a[] and b[] will have identical lowest 12 bits, i.e., (a & 0xfff) == (b & 0xfff). Some intra-loop conflict may occur with:

- a[i] and b[i] of iteration N collide.
- a[i] of iteration N-1 and b[i-1] of iteration N collide.

There are multiple ways to offset dynamic arrays. Examples include:

- Offset the working base pointer from the malloc result by an amount of several cache lines,
- Use customized malloc() routine,
- Use posix_memalign() routine with alignment directives for each dynamic allocation to have different alignments (powers of 2 bytes: 64, 128, 256, 512, etc.) .

The HPC workload known as Leslie3D can be affected by alignment issue.

5.6.7 Streaming Store Versus Regular Store

When writing to memory and data is not expected to be consumed by loads immediately, it may be desirable to choose between streaming stores or regular stores (writeback). On Knights Landing microarchitecture, streaming stores may be preferable if in flat memory mode; see Section 5.1.2.

If MCDRAM is configured as cache mode, and the data being written fits in the MCDRAM cache, it is likely that standard stores will perform better. Experimenting with both options may yield non-trivial performance for your application.

5.6.8 Compiler Switches and Directives

When using Fortran 90 syntax, Fortran programmers should use the CONTIGUOUS attribute when appropriate. If not, the compiler may assume that incoming arrays are not contiguous, and will (potentially) replace vector load and store instructions with VGATHER and VSCATTER instructions. This can have a negative impact on performance.

Expert coders compiling with the Intel compiler can annotate their code with various pragmas. Some of the more useful ones are LOOP_COUNT, SIMD, and UNROLL. Read the documentation for these pragmas, and use them where appropriate. The compiler can produce better code when it is given more information to evaluate the cost of vectorization.

When using the Intel compilers, the compiler switch "-xMIC-AVX512" targets Knights Landing microarchitecture.

5.6.9 Direct Mapped MCDRAM Cache

When MCDRAM is configured in cache mode, the MCDRAM cache is a convenient way to increase memory bandwidth. As a memory side cache, it can automatically cache recently used data, and provide much higher bandwidth than what DDR memory can achieve.

The MCDRAM cache is a direct mapped cache. This means that multiple memory locations can map to a single place in the cache. Because of this, a simple optimization for a program to evaluate its memory bandwidth sensitivity is to turn on the MCDRAM cache. Some applications that heavily utilize only a few GBytes of memory footprint could see performance improvements of up to 4x. Because of the simplicity of this - no source code changes, and the large possible performance benefits, moving from DDR only to MCDRAM cache mode should be one of the first performance optimizations to try.

There are a few scenarios where enabling the cache could reduce performance. One case is when the MCDRAM cache is not able to hold the accessed working set. If an application streams through 64 GB of memory without reuse, the cost of memory access will increase due to checking the MCDRAM cache (and missing), relative to accessing DDR memory.

The caching of data in the MCDRAM direct mapped cache uses the physical address, not the linear address. Even if an address is contiguous in the linear/virtual address space, the physical addresses that the OS allocates and manages are not required to be. This can cause cache contention when a significant portion of the MCDRAM cache are used. These contentions are likely to reduce the peak memory bandwidth achievable, and vary from run to run; as how the OS allocates pages can change from run to run. The performance monitoring hardware in the Knights Landing microarchitecture provides the UNC_E_EDC_ACCESS event to compute the MCDRAM cache hit rate. It can be instructive in diagnosing this problem.

If MCDRAM cache is enabled, every modified line in the tile caches (L1 or L2 cache) must have an entry in the MCDRAM cache. If a line is evicted from the MCDRAM cache, any modified version of that line in the tile caches will writeback its data to memory, and transition to a shared state. There is a very small probability that a pair of lines that are frequently read and written will alias to the same MCDRAM set. This could cause a pair of writes that would normally hit in the tile caches to generate extra mesh traffic when using MCDRAM in cache mode. Due to this, a pair of threads could become substantially slower than the other threads in the chip. Linear to physical mapping can vary from run to run, making it difficult to diagnose.

One case in point is when two threads read and write their private stacks. Conceptually, any data location that is commonly read and written to would work, but register spills to the stack are the most frequent case. If the stacks are offset by a multiple of 16 GB (or the total MCDRAM cache size) in physical memory, they would collide into the same MCDRAM cache set. A run-time that forced all thread stacks to allocate into a contiguous physical memory region would avoid this case from occurring.

There is hardware in the Knights Landing microarchitecture to reduce the frequency of set conflicts from occurring. The probability of hitting this scenario on a given node is extremely small. The best clue to detecting this, is that a pair of threads on the same chip are significantly slower than all other threads during a program phase. Which exact threads cores in a package would experience set collision should vary from run to run, happen rarely, and only when the cache memory mode is enabled. It is very likely that a user may never encounter this on their system.

6. Updates to Appendix F

Change bars and **violet** text show changes to Appendix F of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Earlier Generations of Intel Atom® Microarchitecture and Software Optimization.

Changes to this chapter:

- This chapter has been updated to be Volume 2, Chapter 6.
- Updated capitalization of headings throughout chapter.
- Updated branding throughout chapter.
- Typo and punctuation corrections as necessary.

CHAPTER 6
EARLIER GENERATIONS OF INTEL ATOM® MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

6.1 OVERVIEW

45 nm Intel Atom processors introduced Intel Atom microarchitecture. The same microarchitecture also used in 32 nm Intel Atom processors. This chapter covers a brief overview the Intel Atom microarchitecture, and specific coding techniques for software whose primary targets are processors based on the Intel Atom microarchitecture. The key features of Intel Atom processors to support low power consumption and efficient performance include:

- Enhanced Intel SpeedStep® Technology enables operating system (OS) to program a processor to transition to lower frequency and/or voltage levels while executing a workload.
- Support deep power down technology to reduces static power consumption by turning off power to cache and other sub-systems in the processor.
- Intel Hyper-Threading Technology providing two logical processor for multi-tasking and multi-threading workloads.
- Support Single-instruction multiple-data extensions up to SSE3 and SSSE3.
- Support for Intel 64 and IA-32 architecture.

The Intel Atom microarchitecture is designed to support the general performance requirements of modern workloads within the power-consumption envelop of small form-factor and/or thermally-constrained environments.

6.2 INTEL ATOM® MICROARCHITECTURE

Intel Atom microarchitecture achieves efficient performance and low power operation with a two-issue wide, in-order pipeline that support Hyper-Threading Technology. The in-order pipeline differs from out-of-order pipelines by treating an IA-32 instruction with a memory operand as a single pipeline operation instead of multiple micro-operations.

The basic block diagram of the Intel Atom microarchitecture pipeline is shown in Figure 6-1.

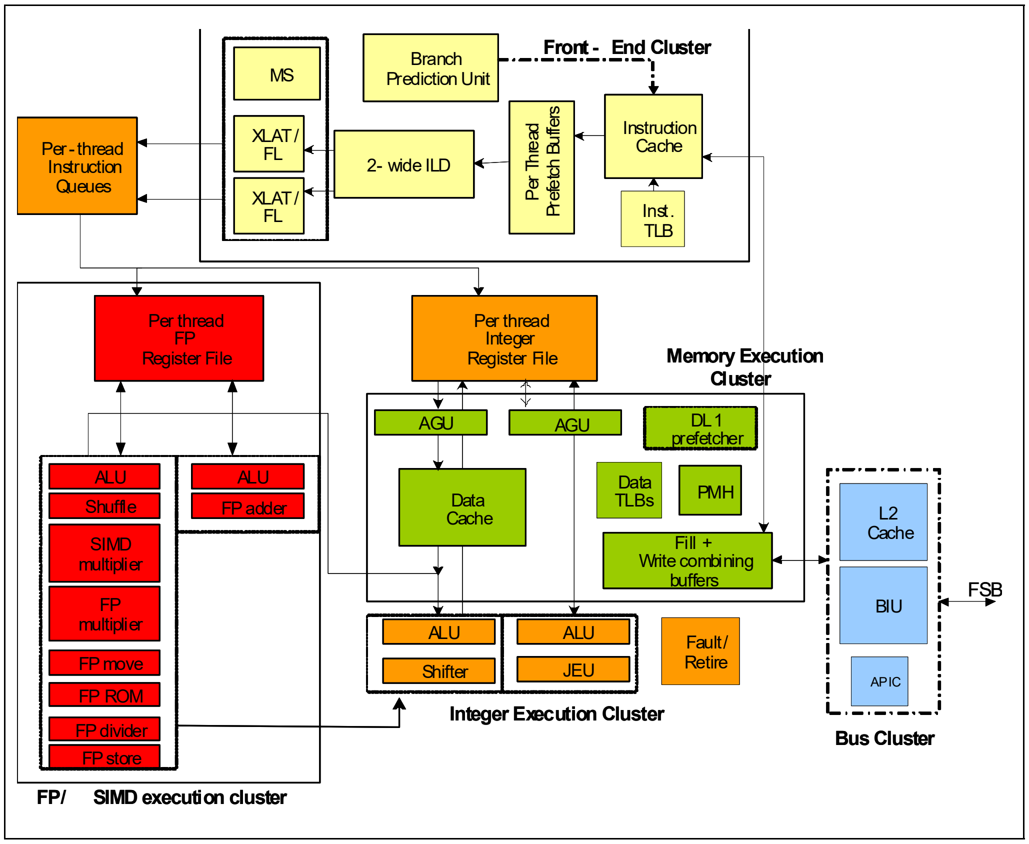


Figure 6-1. Intel Atom® Microarchitecture Pipeline

The front end features a power-optimized pipeline, including:

- 32KB, 8-way set associative, first-level instruction cache.
- Branch prediction units and ITLB.
- Two instruction decoders, each can decode up to one instruction per cycle.

The front end can deliver up to two instructions per cycle to the instruction queue for scheduling. The scheduler can issue up to two instructions per cycle to the integer or SIMD/FP execution clusters via two issue ports.

Each of the two issue ports can dispatch an instruction per cycle to the integer cluster or the SIMD/FP cluster to execute. The port-bindings of the integer and SIMD/FP clusters have the following features:

- Integer execution cluster:
  — Port 0: ALU0, Shift/Rotate unit, Load/Store.
  — Port 1: ALU1, Bit processing unit, jump unite and LEA.
  — Effective “load-to-use” latency of 0 cycle.
- SIMD/FP execution cluster:
  — Port 0: SIMD ALU, Shuffle unit, SIMD/FP multiply unit, Divide unit, (support IMUL, IDIV).
  — Port 1: SIMD ALU, FP Adder.
  — The two SIMD ALUs and the shuffle unit in the SIMD/FP cluster are 128-bit wide, but 64-bit integer SIMD computation is restricted to port 0 only.
  — FP adder can execute ADDPS/SUBPS in 128-bit data path, data path for other FP add operations are 64-bit wide.

  - — Safe Instruction Recognition algorithm for FP/SIMD execution allow younger, short-latency integer instruction to execute without being blocked by older FP/SIMD instruction that might cause exception.
  - — FP multiply pipe also supports memory loads.
  - — FP ADD instructions with memory load reference can use both ports to dispatch.

The memory execution sub-system (MEU) can support 48-bit linear address for Intel 64 Architecture, either 32-bit or 36-bit physical addressing modes. The MEU provides:

- 24KB first level data cache.
- Hardware prefetching for L1 data cache.
- Two levels of DTLB for 4KByte and larger paging structure.
- Hardware pagewalker to service DTLB and ITLB misses.
- Two address generation units (port 0 supports loads and stores, port 1 supports LEA and stack operations).
- Store-forwarding support for integer operations.
- 8 write combining buffers.

The bus logic sub-system provides:

- 512KB, 8-way set associative, unified L2 cache.
- Hardware prefetching for L2 and interface logic to the front side bus.

6.2.1 Hyper-Threading Technology Support in Intel Atom® Microarchitecture

The instruction queue is statically partitioned for scheduling instruction execution from two threads. The scheduler is able to pick one instruction from either thread and dispatch to either of port 0 or port 1 for execution. The hardware makes selection choice on fetching/decoding/dispatching instructions between two threads based on criteria of fairness as well as each thread’s readiness to make forward progress.

6.3 CODING RECOMMENDATIONS FOR INTEL ATOM® MICROARCHITECTURE

Instruction scheduling heuristics and coding techniques that apply to out-of-order microarchitectures may not deliver optimal performance on an in-order microarchitecture. Likewise instruction scheduling heuristics and coding techniques for an in-order pipeline like Intel Atom microarchitecture may not achieve optimal performance on out-of-order microarchitectures. This section covers specific coding recommendations for software whose primary deployment targets are processors based on Intel Atom microarchitecture.

6.3.1 Optimization for Front End of Intel Atom® Microarchitecture

The two decoders in the front end of Intel Atom microarchitecture can handle most instructions in the Intel 64 and IA-32 architecture. Some instructions dealing with complicated operations require the use of an MSROM in the front end. Instructions that go through the two decoders generally can be decoded by either decoder unit of the front end in most cases. Instructions the must use the MSROM or conditions that cause the front end to re-arrange decoder assignments will experience a delay in the front end.

Software can use specific performance monitoring events to detect instruction sequences and/or conditions that cause front end to re-arrange decoder assignment.

Assembly/Compiler Coding Rule 1. (MH impact, ML generality) *For Intel Atom processors, minimize the presence of complex instructions requiring MSROM to take advantage the optimal decode bandwidth provided by the two decode units.*

Using the performance monitoring events "MACRO_INSTS.NON_CISC_DECODED" and "MACRO_INSTS.CISC_DECODED" can be used to evaluate the percentage instructions in a workload that required MSROM.

Assembly/Compiler Coding Rule 2. (M impact, H generality) *For Intel Atom processors, keeping the instruction working set footprint small will help the front end to take advantage the optimal decode bandwidth provided by the two decode units.*

Assembly/Compiler Coding Rule 3. (MH impact, ML generality) *For Intel Atom processors, avoiding back-to-back X87 instructions will help the front end to take advantage the optimal decode bandwidth provided by the two decode units.*

Using the performance monitoring events "DECODE_RESTRICTION" can count the number of occurrences in a workload that encountered delays causing reduction of decode throughput.

In general the front end restrictions are not typical a performance limiter until the retired "cycle per instruction" becomes less than unity (maximum theoretical retirement throughput corresponds to CPI of 0.5). To reach CPI below unity, it is important to generate instruction sequences that go through the front end as instruction pairs decodes in parallel by the two decoders. After the front end, the scheduler and execution hardware do not need to dispatch the decode pairings through port 0 and port 1 in the same order.

The decoders cannot decode past a jump instruction, so jumps should be paired as the second instruction in a decoder-optimized pairing. The front end can only handle one X87 instruction per cycle, and only decoder unit 0 can request a transfer to use MSROM. Instructions that are longer than 8 bytes or having more than three prefixes will results in a MSROM transfer, experiencing two cycles of delay in the front end.

Instruction lengths and alignment can impact decode throughput. The prefetching buffers inside the front end imposes a throughput limit that if the number of bytes being decoded in any 7-cycle window exceeds 48 bytes, the front end will experience a delay to wait for a buffer. Additionally, every time an instruction pair crosses 16 byte boundary, it requires the front end buffer to be held on for at least one more cycle. So instruction alignment crossing 16 byte boundary is highly problematic.

Instruction alignment can be improved using a combination of an ignore prefix and an instruction.

Example 6-1. Instruction Pairing and Alignment to Optimize Decode Throughput on Intel Atom® Microarchitecture

Address	Instruction Bytes	Disassembly
7FFFFDF0	0F594301	mulps xmm0, [ebx+ 01h]
7FFFFDF4	8341FFFF	add dword ptr [ecx-01h], -1
7FFFFDF8	83C2FF	add edx, , -1
7FFFFDFB	64	; FS prefix override is ignored, improves code alignment
7FFFFDFC	F20f58E4	add xmm4, xmm4
7FFFFE00	0F594B11	mulps xmm1, [ebx+ 11h]
7FFFFE04	8369EFFF	sub dword ptr [ecx- 11h], -1
7FFFFE08	83EAFF	sub edx, -1
7FFFFE0B	64	; FS prefix override is ignored, improves code alignment
7FFFFE0C	F20F58ED	addsd xmm5, xmm5
7FFFFE10	0F595301	mulps xmm2, [ebx +1]

Example 6-1. Instruction Pairing and Alignment to Optimize Decode Throughput on Intel Atom® Microarchitecture

Address	Instruction Bytes	Disassembly
7FFFFE14	8341DFFF	add dword ptr [ecx-21H], -1
7FFFFE18	83C2FF	add edx, -1
7FFFFE1B	64	; FS prefix override is ignored, improves code alignment
7FFFFE1C	F20F58F6	addssd xmm6, xmm6
7FFFFE20	0F595B11	mulps xmm3, [ebx+ 11h]
7FFFFE24	8369CFFF	sub dword ptr [ecx- 31h], -1
7FFFFE28	83EAFF	sub edx, -1

When a small loop contains some long-latency operation inside, loop unrolling may be considered as a technique to find adjacent instruction that could be paired with the long-latency instruction to enable that adjacent instruction to make forward progress. However, loop unrolling must also be evaluated on its impact to increased code size and pressure to the branch target buffer.

The performance monitoring event "BACLEARS" can provide a means to evaluate whether loop unrolling is helping or hurting front end performance. Another event "ICACHE_MISSES" can help evaluate if loop unrolling is increasing the instruction footprint.

Branch predictors in Intel Atom processor do not distinguish different branch types. Sometimes mixing different branch types can cause confusion in the branch prediction hardware.

The performance monitoring event "BR_MISSP_TYPE_RETIRED" can provide a means to evaluate branch prediction issues due to branch types.

6.3.2 Optimizing the Execution Core

This section covers several items that can help software use the two-issue-wide execution core to make forward progress with two instructions more frequently.

6.3.2.1 Integer Instruction Selection

In an in-order machine, instruction selection and pairing can have an impact on the machine's ability to discover instruction-level-parallelism for instructions that have data ready to execute. Some examples are:

- **EFLAG**: The consumer instruction of any EFLAG flag bit can not be issued in the same cycle as the producer instruction of the EFLAG register. For example, ADD could modify the carry bit, so it is a producer; JC (or ADC) reads the carry bit and is a consumer.
  - Conditional jumps are able to issue in the following cycle after the consumer.
  - A consumer instruction of other EFLAG bits must wait one cycle to issue after the producer (two cycle delay).

Assembly/Compiler Coding Rule 4. (M impact, H generality) *For Intel Atom processors, place a MOV instruction between a flag producer instruction and a flag consumer instruction that would have incurred a two-cycle delay. This will prevent partial flag dependency.*

- **Long-latency Integer Instructions**: They will block shorter latency instruction on the same thread from issuing (required by program order). Additionally, they will also block shorter-latency instruction on both threads for one cycle to resolve writeback resource.
- **Common Destination**: Two instructions that produce results to the same destination can not issue in the same cycle.

- **Expensive Instructions**: Some instructions have special requirements and become expensive in consuming hardware resources for an extended period during execution. It may be delayed in execution until it is the oldest in the instruction queue; it may delay the issuing of other younger instructions. Examples of these include FDIV, instructions requiring execution units from both ports, etc.

6.3.2.2 Address Generation

The hardware optimizes the general case of instruction ready to execute must have data ready, and address generation precedes data being ready. If address generation encounters a dependency that needs data from another instruction, this dependency in address generation will incur a delay of 3 cycles.

The address generation unit (AGU) may be used directly in three situations that affect execution throughput of the two-wide machine. The situations are:

- **Implicit ESP updates**: When the ESP register is not used as the destination of an instruction (explicit ESP updates), an implicit ESP update will occur with instructions like PUSH, POP, CALL, RETURN. Mixing explicit ESP updates and implicit ESP updates will also lead to dependency between address generation and data execution.
- **LEA**: The LEA instruction uses the AGU instead of the ALU. If one of the source register of LEA must come from an execution unit. This dependency will also cause a 3 cycle delay. Thus, LEA should not be used in the technique of adding two values and produce the result in a third register. LEA should be used for address computation.
- **Integer-FP/SIMD transfer**: Instructions that transfer integer data to the FP/SIMD side of the machine also uses AGU. Examples of these instructions include MOVD, PINSRW. If one of the source register of these instructions depends on the result of an execution unit, this dependency will also cause a delay of 3 cycles.

Example 6-2. Alternative to Prevent AGU and Execution Unit Dependency

```
a) Three cycle delay when using LEA in ternary operations
        mov eax, 0x01
        lea eax, 0x8000[eax+ebp]; values in eax comes from execution of previous instruction
        ; 3 cycle delay due to lea and execution dependency

b) Dependency handled in execution, avoiding AGU and execution dependency
        mov eax, 0x01
        add eax, 0x8000
        add eax, ebp
```

Assembly/Compiler Coding Rule 5. (MH impact, H generality) *For Intel Atom processors, LEA should be used for address manipulation; but software should avoid the following situations which creates dependencies from ALU to AGU: an ALU instruction (instead of LEA) for address manipulation or ESP updates; a LEA for ternary addition or non-destructive writes which do not feed address generation. Alternatively, hoist producer instruction more than 3 cycles above the consumer instruction that uses the AGU.*

6.3.2.3 Integer Multiply

Integer multiply instruction takes several cycles to execute. They are pipelined such that an integer multiply instruction and another long-latency instruction can make forward progress in the execution phase. However, integer multiply instructions will block other single-cycle integer instructions from issuing due to requirement of program order.

Assembly/Compiler Coding Rule 6. (M impact, M generality) *For Intel Atom processors, sequence an independent FP or integer multiply after an integer multiply instruction to take advantage of pipelined IMUL execution.*

Example 6-3. Pipeling Instruction Execution in Integer Computation

```
a) Multi-cycle Imul instruction can block 1-cycle integer instruction
        imul eax, eax
        add ecx, ecx ; 1 cycle int instruction blocked by imul for 4 cycles
        imul ebx, ebx ; instruction blocked by in-orer issue

b) Back-to-back issue of independent imul are pipelined
        imul eax, eax
        imul ebx, ebx ; 2nd imul can issue 1 cycle later
        add ecx, ecx ; 1 cycle int instruction blocked by imul
```

6.3.2.4 Integer Shift Instructions

Integer shift instructions that encodes shift count in the immediate byte have one-cycle latency. In contrast, shift instructions using shift count in the ECX register may need to wait for the register count are updated. Thus shift instruction using register count has 3-cycle latency.

Assembly/Compiler Coding Rule 7. (M impact, M generality) *For Intel Atom processors, hoist the producer instruction for the implicit register count of an integer shift instruction before the shift instruction by at least two cycles.*

6.3.2.5 Partial Register Access

Although partial register access does not cause additional delay, the in-order hardware tracks dependency on the full register. Thus 8-bit registers like AL and AH are not treated as independent registers. Additionally some instructions like LEA, vanilla loads, and pop are slower when the input is smaller than 4 bytes.

Assembly/Compiler Coding Rule 8. (M impact, MH generality) *For Intel Atom processors, LEA, simple loads and POP are slower if the input is smaller than 4 bytes.*

6.3.2.6 FP/SIMD Instruction Selection

Table 6-1 summarizes the characteristics of various execution units in Intel Atom microarchitecture that are likely used most frequently by software.

Table 6-1. Instruction Latency/Throughput Summary of Intel Atom® Microarchitecture

Instruction Category	Latency (cycles)	Throughput	# of Execution Unit
SIMD Integer ALU			
128-bit ALU/logical/move	1	1	2
64-bit ALU/logical/move	1	1	2
SIMD Integer Shift			
128-bit	1	1	1
64-bit	1	1	1

Table 6-1. Instruction Latency/Throughput Summary of Intel Atom® Microarchitecture (Contd.)

Instruction Category	Latency (cycles)	Throughput	# of Execution Unit
SIMD Shuffle			
128-bit	1	1	1
64-bit	1	1	1
SIMD Integer Multiply			
128-bit	5	2	1
64-bit	4	1	1
FP Adder			
X87 Ops (FADD)	5	1	1
Scalar SIMD (addsd, addss)	5	1	1
Packed single (addps)	5	1	1
Packed double (addpd)	6	5	1
FP Multiplier			
X87 Ops (FMUL)	5	2	1
Scalar single (mulss)	4	1	1
Scalar double (mulsd)	5	2	1
Packed single (mulps)	5	2	1
Packed double (mulpd)	9	9	1
IMUL			
IMUL r32, r/m32	5	1	1
IMUL r12, r/m16	6	1	1

SIMD/FP instruction selection generally should favor shorter latency first, then favor faster throughput alternatives whenever possible. Note that packed double-precision instructions are not pipelined, using two scalar double-precision instead can achieve higher performance in the execution cluster.

Assembly/Compiler Coding Rule 9. (MH impact, H generality) *For Intel Atom processors, prefer SIMD instructions operating on XMM register over X87 instructions using FP stack. Use Packed single-precision instructions where possible. Replace packed double-precision instruction with scalar double-precision instructions.*

Assembly/Compiler Coding Rule 10. (M impact, ML generality) *For Intel Atom processors, library software performing sophisticated math operations like transcendental functions should use SIMD instructions operating on XMM register instead of native X87 instructions.*

Assembly/Compiler Coding Rule 11. (M impact, M generality) *For Intel Atom processors, enable DAZ and FTZ whenever possible.*

Several performance monitoring events may be useful for SIMD/FP instruction selection tuning: "SIMD_INST_RETIRED.{PACKED_SINGLE, SCALAR_SINGLE, PACKED_DOUBLE, SCALAR_DOUBLE}" can be used to determine the instruction selection in the program. "FP_ASSIST" and "SIR" can be used to see if floating exceptions (or false alarms) are impacting program performance.

The latency and throughput of divide instructions vary with input values and data size. Intel Atom microarchitecture implements a radix-2 based divider unit. So, divide/sqrt latency will be significantly longer than other FP operations. The issue throughput rate of divide/sqrt will be correspondingly lower.

The divide unit is shared between two logical processors, so software should consider all alternatives to using the divide instructions.

Assembly/Compiler Coding Rule 12. (H impact, L generality) *For Intel Atom processors, use divide instruction only when it is absolutely necessary, and pay attention to use the smallest data size operand.*

The performance monitoring events “DIV” and “CYCLES_DIV_BUSY” can be used to see if the divides are a bottleneck in the program.

FP operations generally have longer latency than integer instructions. Writeback of results from FP operation generally occur later in the pipe stages than integer pipeline. Consequently, if an instruction has dependency on the result of some FP operation, there will be a two-cycle delay. Examples of these type of instructions are FP-to-integer conversions CVTxx2xx, MOVD from XMM to general purpose registers.

In situations where software needs to do computation with consecutive groups 4 single-precision data elements, PALIGNR+MOVAPS is preferred over MOVUPS. Loading 4 data elements with unconstrained array index *k*, such as MOVUPS xmm1, _pArray[k], where the memory address _pArray is aligned on 16-byte boundary, will periodically causing cache line split, incurring a 14-cycle delay.

The optimal approach is for each k that is not a multiple of 4, round down k to multiples of 4 with j = 4*(k/4), do a MOVAPS MOVAPS xmm1, _pArray[j] and MOVAPS xmm1, _pArray[j+4], and use PALIGNR to splice together the four data elements needed for computation.

Assembly/Compiler Coding Rule 13. (MH impact, M generality) *For Intel Atom processors, prefer a sequence MOVAPS+PALIGN over MOVUPS. Similarly, MOVDQA+PALIGNR is preferred over MOVDQU.*

6.3.3 Optimizing Memory Access

This section covers several items that can help software optimize the performance of the memory subsystem.

Memory access to system memory of cache access that encounter certain hazards can cause the memory access to become an expensive operation, blocking short-latency instructions to issue even when they have data ready to execute.

The performance monitoring events “REISSUE” can be used to assess the impact of re-issued memory instructions in the program.

6.3.3.1 Store Forwarding

In a few limited situations, Intel Atom microarchitecture can forward data from a preceding store operation to a subsequent load instruction. The situations are:

- Store-forwarding is supported only in the integer pipeline, and does not apply to FP nor SIMD data. Furthermore, the following conditions must be met:
  a. The store and load operations must be of the same size and to the same address.
  b. Data size larger than 8 bytes do not forward from a store operation.
- When data forwarding proceeds, data is forwarded base on the least significant 12 bits of the address. So software must avoid the address aliasing situation of storing to an address and then loading from another address that aliases in the lowest 12-bits with the store address.

6.3.3.2 First-level Data Cache

Intel Atom microarchitecture handles each 64-byte cache line of the first-level data cache in 16 4-byte chunks. This implementation characteristic has a performance impact to data alignment and some data access patterns.

Assembly/Compiler Coding Rule 14. (MH impact, H generality) *For Intel Atom processors, ensure data are aligned in memory to its natural size. For example, 4-byte data should be aligned to 4-byte boundary, etc. Additionally, smaller access (less than 4 bytes) within a chunk may experience delay if they touch different bytes.*

6.3.3.3 Segment Base

In Intel Atom microarchitecture, the address generation unit assumes that the segment base will be 0 by default. Non-zero segment base will cause load and store operations to experience a delay.

- If the segment base isn’t aligned to a cache line boundary, the max throughput of memory operations is reduced to one very 9 cycles.

If the segment base is non-zero but cache line aligned the penalty varies by segment base.

- DS will have a max throughput of one every two cycles.
- FS, and GS will have a max throughput of one every two cycles. However, FS and GS are anticipated to be used only with non-zero bases and therefore have a max throughput of one every two cycles even if the segment base is zero.
- ES:
  - — If used as the implicit segment base for the destination of string operation, will have a max throughput of one every two cycles for non-zero but cacheline aligned bases.
  - — Otherwise, only do one operation every nine cycles.
- CS and SS will always have a max throughput of one every nine cycles if its segment base is non-zero but cache line aligned.

Assembly/Compiler Coding Rule 15. (H impact, ML generality) *For Intel Atom processors, use segments with base set to 0 whenever possible; avoid non-zero segment base address that is not aligned to cache line boundary at all cost.*

Assembly/Compiler Coding Rule 16. (H impact, L generality) *For Intel Atom processors, when using non-zero segment bases, Use DS, FS, GS; string operation should use implicit ES.*

Assembly/Compiler Coding Rule 17. (M impact, ML generality) *For Intel Atom processors, favor using ES, DS, SS over FS, GS with zero segment base.*

6.3.3.4 String Moves

Using MOVS/STOS instruction and REP prefix on Intel Atom processor should recognize the following items:

- For small count values, using REP prefix is less efficient than not using REP prefix. This is because the hardware does have small REP count optimization.
- For small count values, using REP prefix is less efficient than not using REP prefix. This is because the hardware does have small REP count optimization.
- For large count values, using REP prefix will be less efficient than using 16-byte SIMD instructions.
- Incrementing address in loop iterations should favor LEA instruction over explicit ADD instruction.
- If data footprint is such that memory operation is accessing L2, use of software prefetch to bring data to L1 can avoid memory operation from being re-issued.
- If string/memory operation is accessing system memory, using non-temporal hints of streaming store instructions can avoid cache pollution.

Example 6-4. Memory Copy of 64-byte

```
T1:    prefetcht0 [eax+edx+0x80] ; prefetch ahead by two iterations
       movdqa  xmm0, [eax+ edx] ; load data from source (in L1 by prefetch)
       movdqa  xmm1, [eax+ edx+0x10]
       movdqa  xmm2, [eax+ edx+0x20]
       movdqa  xmm3, [eax+ edx+0x30]
       movdqa  [ebx+ edx], xmm0; store data to destination
       movdqa  [ebx+ edx+0x10], xmm1
       movdqa  [ebx+ edx+0x30], xmm2
       movdqa  [ebx+ edx+0x30], xmm3
       lea     edx, 0x40 ; use LEA to adjust offset address for next iteration
       dec     ecx
       jnz     T1
```

6.3.3.5 Parameter Passing

Due to the limited situations of load-to-store forwarding support in Intel Atom microarchitecture, parameter passing via the stack places restrictions on optimal usage by the callee function. For example, "bool" and "char" data usually are pushed onto the stack as 32-bit data, a callee function that reads "bool" or "char" data off the stack will face store-forwarding delay and causing the memory operation to be re-issued.

Compiler should recognize this limitation and generate prolog for callee function to read 32-bit data instead of smaller sizes.

Assembly/Compiler Coding Rule 18. (MH impact, M generality) *For Intel Atom processors, "bool" and "char" value should be passed onto and read off the stack as 32-bit data.*

6.3.3.6 Function Calls

In Intel Atom microarchitecture, using PUSH/POP instructions to manage stack space and address adjustment between function calls/returns will be more optimal than using ENTER/LEAVE alternatives. This is because PUSH/POP will not need MSROM flows and stack pointer address update is done at AGU.

When a callee function need to return to the caller, the callee could issue POP instruction to restore data and restore the stack pointer from the EBP.

Assembly/Compiler Coding Rule 19. (MH impact, M generality) *For Intel Atom processors, favor register form of PUSH/POP and avoid using LEAVE; Use LEA to adjust ESP instead of ADD/SUB.*

6.3.3.7 Optimization of Multiply/Add Dependent Chains

Computations of dependent multiply and add operations can illustrate the usage of several coding techniques to optimize for the front end and in-order execution pipeline of the Intel Atom microarchitecture.

Example 6-5a shows a code sequence that may be used on out-of-order microarchitectures. This sequence is far from optimal on Intel Atom microarchitecture. The full latency of multiply and add operations are exposed and it is not very successful at taking advantage of the two-issue pipeline.

Example 6-5b shows an improved code sequence that takes advantage of the two-issue in-order pipeline of Intel Atom microarchitecture. Because the dependency between multiply and add operations are present, the exposure of latency are only partially covered.

Example 6-5. Examples of Dependent Multiply and Add Computation

a) Instruction sequence that encounters stalls

```
; accumulator xmm2 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        movaps xmm1, [edi] ; vector stored in 16-byte aligned memory
        mulps xmm0, xmm1
        addps xmm2, xmm0 ; dependency and branch exposes latency of mul and add
        add esi, 16 ;
        add edi, 16
        sub ecx, 1
        jnz top
```

b) Improved instruction sequence to increase execution throughput

```
; accumulator xmm4 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm0, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm4, xmm0 ; latency exposures partially covered by independent instructions
        dec ecx ;
        jnz top
```

c) Improving instruction sequence further by unrolling and interleaving

```
; accumulator xmm0, xmm1, xmm2, xmm3 initialized
Top:    movaps xmm0, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm0, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm5, xmm1 ; dependent multiply hoisted by unrolling and interleaving
        movaps xmm1, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm1, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm6, xmm2 ; dependent multiply hoisted by unrolling and interleaving
```

(continue)

```
        movaps xmm2, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm2, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm7, xmm3 ; dependent multiply hoisted by unrolling and interleaving
        movaps xmm3, [esi] ; vector stored in 16-byte aligned memory
        lea esi, [esi+16] ; can schedule in parallel with load
        mulps xmm3, [edi] ;
        lea edi, [edi+16] ; can schedule in parallel with multiply
        addps xmm4, xmm0 ; dependent multiply hoisted by unrolling and interleaving
        sub ecx, 4;
        jnz top
        ; sum up accumulators xmm0, xmm1, xmm2, xmm3 to reduce dependency inside the loop
```

Example 6-5c illustrates a technique that increases instruction-level parallelism and further reduces latency exposures of the multiply and add operations. By unrolling four times, each ADDPS instruction can be hoisted far from its dependent producer instruction MULPS. Using an interleaving technique, non-dependent ADDPS and MULPS can be placed in close proximity. Because the hardware that executes MULPS and ADDPS is pipelined, the associated latency can be covered much more effectively by this technique relative to Example 6-5b.

6.3.3.8 Position Independent Code

Position independent code often needs to obtain the value of the instruction pointer. a show one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 6-5b show an alternative technique to put the value of IP into the ECX register using a matched pair of CALL/RET.

Example 6-6. Instruction Pointer Query Techniques

```
a) Using call without return to obtain IP
        call _label; return address pushed is the IP of next instruction
_label:
        pop ECX; IP of this instruction is now put into ECX
```

```
b) Using matched call/ret pair

        call _lblcx;
        ... ; ECX now contains IP of this instruction
        ...
_lblcx
        mov ecx, [esp];
        ret
```

6.4 INSTRUCTION LATENCY

This section lists the port-binding and latency information of Intel Atom microarchitecture. The port-binding information for each instruction may show one of 3 situations:

- ‘Single digit’ - the specific port that must be issued.
- (0, 1) - either port 0 or port 1.
- ‘B’ - both ports are required.

In the “Instruction” column:

- If different operand syntax of the same instruction have the same port-binding and latency, operand syntax is omitted.
- When different operand syntax may produce different latency or port binding, the operand syntax is listed; but instruction syntax of different operand sizes may be compacted and abbreviated with a footnote.

Instruction that required decoder assistance from MSROM are marked in the “Comment” column (should be used minimally if more decode-efficient alternatives are available).

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
ADD/AND/CMP/OR/SUB/XOR/TEST[1] (E)AX/AL, imm;	(0, 1)	1	0.5
ADD/AND/CMP/OR/SUB/XOR[2] mem, Imm8; ADD/AND/CMP/OR/SUB/XOR/TEST[4] mem, imm; TEST m8, imm8	0	1	1
ADD/AND/CMP/OR/SUB/XOR/TEST[2] mem, reg; ADD/AND/CMP/OR/SUB/XOR[2] reg, mem;	0	1	1
ADD/AND/CMP/OR/SUB/XOR[2] reg, Imm8; ADD/AND/CMP/OR/SUB/XOR[4] reg, imm	(0, 1)	1	0.5
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, mem	B	7	6
ADDPD/ADDSUBPD/MAXPD/MAXPS/MINPD/MINPS/SUBPD xmm, xmm	B	6	5
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, mem	B	5	1
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/SUBSS xmm, xmm	1	5	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, mem	0	1	1
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XORPD/XORPS xmm, xmm	(0, 1)	1	1
BSF/BSR r16, m16	B	17	16
BSF/BSR[3] reg, mem	B	16	15
BSF/BSR[4] reg, reg	B	16	15
BT m16, imm8; BT[3] mem, imm8	(0, 1)	2; 1	1
BT m16, r16; BT[3] mem, reg	B	10, 9	8
BT[4] reg, imm8; BT[4] reg, reg	1	1	1
BTC m16, imm8; BTC[3] mem, imm8	B	3; 2	2
BTC/BTR/BTS m16; r16	B	12	11
BTC/BTR/BTS[3] mem, reg	B	11	10
BTC/BTR/BTS[4] reg, imm8; BTC/BTR/BTS[4] reg, reg	1	1	1
CALL mem	(0, 1)	2	2
CALL reg; CALL rel16; CALL rel32	B	1	1
CMOV[4] reg, mem; MOV[1] (E)AX/AL, MOFFS; MOV[2] mem, imm	0	1	1
CMOV[4] reg, reg; MOV[2] reg, imm; MOV[2] reg, reg; ; SETcc r8	(0, 1)	1	0.5
CMPPD/CMPPS xmm, mem, imm; CVTTPS2DQ xmm, mem	B	7	6
CMPPD/CMPPS xmm, xmm, imm; CVTTPS2DQ xmm, xmm	B	6	5
CMPSD/CMPSS xmm, mem, imm	B	5	1
CMPSD/CMPSS xmm, xmm, imm	1	5	1
(U)COMISD/(U)COMISS xmm, mem;	B	10	9
(U)COMISD/(U)COMISS xmm, xmm;	B	9	8
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, mem	B	8	7
CVTDQ2PD/CVTPD2DQ/CVTPD2PS xmm, xmm	B	7	6
CVTDQ2PS/CVTSD2SS/CVTSI2SS/CVTSS2SD xmm, mem	B	7	6

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
CVTDQ2PS/CVTSD2SS/CVTSS2SD xmm, xmm	B	6	5
CVT(T)PD2PI mm, mem; CVTPI2PD xmm, mem	B	8	7
CVT(T)PD2PI mm, xmm; CVTPI2PD xmm, mm	B	7	6
CVTPI2PS/CVTSI2SD xmm, mem;	B	5	4
CVTPI2PS xmm, mm;	1	5	1
CVTPS2DQ xmm, mem;	B	7	6
CVTPS2DQ xmm, xmm;	B	6	5
CVT(T)PS2PI mm, mem;	B	5	5
CVT(T)PS2PI mm, xmm;	1	5	1
CVT(T)SD2SI[3] reg, mem; CVT(T)SS2SI r32, mem	B	9	8
CVT(T)SD2SI[3] reg, xmm; CVT(T)SS2SI r32, xmm	B	8	7
CVTSI2SD xmm, r32; CVTSI2SS xmm, r32	B	7; 6	5
CVTSI2SD xmm, r64; CVTSI2SS xmm, r64	B	6; 7	5
CVT(T)SS2SI r64, mem; RCPPS xmm, mem	B	10	9
CVT(T)SS2SI r64, xmm; RCPPS xmm, xmm	B	9	8
CVTTPD2DQ xmm, mem	B	8	7
CVTTPD2DQ xmm, xmm	B	7	6
DEC/INC[2] mem; MASKMOVQ; MOVAPD/MOVAPS mem, xmm	0	1	1
DEC/INC[2] reg; FLD ST; FST/FSTP ST; MOVDQ2Q mm, xmm	(0, 1)	1	0.5
DIVPD; DIVPS	B	125; 70	124; 69
DIVSD; DIVSS	B	62; 34	61; 33
EMMS; LDMXCSR	B	5	4
FABS/FCHS/FXCH; MOVQ2DQ xmm, mm; MOVSX/MOVZX r16, r16	(0, 1)	1	0.5
FADD/FSUB/FSUBR[3] mem	B	5	4
FADD/FADDP/FSUB/FSUBP/FSUBR/FSUBRP ST;	1	5	1
FCMOV	B	6	5
FCOM/FCOMP[3] mem	B	1	1
FCOM/FCOMP/FCOMPP/FUCOM/FUCOMP ST; FTST	1	1	1
FCOMI/FCOMIP/FUCOMI/FUCOMIP ST	B	9	8
FDIV/FSQRT[3] mem; FDIV/FSQRT ST	0	25-65	24-64
FIADD/FIMUL[5] mem	B	11	10
FICOM/FICOMP mem	B	7	6
FILD[4] mem	B	5	4
FLD[3] mem; FXAM; MOVAPD/MOVAPS/MOVD xmm, mem	0	1	1
FLDCW	B	5	4
FMUL/FMULP ST; FMUL[3] mem	0	5	1

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
FNSTSW AX; FNSTSW m16	B	10; 14	9; 13
FST/FSTP[3] mem	B	2	1
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, mem	B	9	8
HADDPD/HADDPS/HSUBPD/HSUBPS xmm, xmm	B	8	7
IDIV r/m8; IDIV r/m16; IDIV r/m32; IDIV r/m64;	B	33;42;57;197	32;41;56;196
IMUL/MUL[6] EAX/AL, mem; IMUL/MUL AX, m16	B	7; 8	6; 7
IMUL/MUL[7] AX/AL, reg; IMUL/MUL EAX, r32	B	7; 6	6; 5
IMUL m16, imm8/imm16; IMUL r16, m16	B	7;	6
IMUL r/m32, imm8/imm32; IMUL r32, r/m32	0	5	1
IMUL r/m64, imm8/imm32;	B	14	13
IMUL r16, r16; IMUL r16, imm8/imm16	B	6	5
IMUL r64, r/m64; IMUL/MUL RAX, r/m64	B	11; 12	10; 11
JCC[1]; JMP[4] reg; JMP[1]	1	1	1
JCXZ; JECXZ; JRCXZ	B	4	1
JMP mem[4];	B	2	1
LDDQU; MOVDQU/MOVUPD/MOVUPS xmm, mem;	B	3	2
LEA r16, mem; MASKMOVDQU; SETcc m8	(0, 1)	2	1
LEA, reg, mem	1	1	1
LEAVE;	B	2;	2
MAXSD/MAXSS/MINSD/MINSS xmm, mem	B	5	1
MAXSD/MAXSS/MINSD/MINSS xmm, xmm	1	5	1
MOV[2] MOFFS, (E)AX/AL; MOV[2] reg, mem; MOV[2] mem, reg	0	1	1
MOVD mem[3], mm; MOVD xmm, reg[3]; MOVD mm, mem[3]	0	1	1
MOVD reg[3], mm; MOVD reg[3], xmm; PMOVMSK reg[3], mm	0	3	1
MOVDQA/MOVQ xmm, mem; MOVDQA/MOVD mem, xmm;	0	1	1
MOVDQA/MOVDQU/MOVUPD xmm, xmm; MOVQ mm, mm	(0, 1)	1	0.5
MOVDQU/MOVUPD/MOVUPS mem, xmm;	B	2	2
MOVHLPS;MOVLHPS;MOVHPD/MOVHPS/MOVLPD/MOVLPS	0	1	1
MOVMSKPD/MOVSKPS/PMOVMSKB reg[3], xmm	0	3	1
MOVNTI[3] mem, reg; MOVNTPD/MOVNTPS; MOVNTQ	0	1	1
MOVQ mem, mm; MOVQ mm, mem; MOVDDUP	0	1	1
MOVSD/MOVSS xmm, xmm; MOVSXD[5] reg, reg	(0, 1)	1	0.5
MOVSD/MOVSS xmm, mem; PALIGNR	0	1	1
MOVSD/MOVSS mem, xmm; PINSRW	0	1	1
MOVSHDUP/MOVSLDUP xmm, mem	0	1	1

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
MOVSHDUP/MOVSLDUP/MOVUPS xmm, xmm	(0, 1)	1	0.5
MOVSX/MOVZX r16, m8; MOVSX/MOVZX r16, r8	0	3; 2	1
MOVSX/MOVZX reg[3], r/m8; MOVSX/MOVZX reg[3], r/m16	0	1	1
MOVSXD[5] reg, mem; MOVSXD r64, r/m32	0	1	1
MULPS/MULSD xmm, mem; MULSS xmm, mem;	0	5; 4	2
MULPS/MULSD xmm, xmm; MULSS xmm, xmm	0	5; 4	2
MULPD	B	5; 4	2
NEG/NOT[2] mem; PREFETCHNTA; PREFETCHTx	0	10	9
NEG/NOT[2] reg; NOP	(0, 1)	1	0.5
PABSB/D/W mm, mem; PABSB/D/W xmm, mem	0	1	1
PABSB/D/W mm, mm; PABSB/D/W xmm, xmm	(0, 1)	1	0.5
PACKSSDW/WB mm, mem; PACKSSDW/WB xmm, mem	0	1	1
PACKSSDW/WB mm, mm; PACKSSDW/WB xmm, xmm	0	1	1
PACKUSWB mm, mem; PACKUSWB xmm, mem	0	1	1
PACKUSWB mm, mm; PACKUSWB xmm, xmm	0	1	1
PADDB/D/W/Q mm, mem; PADDB/D/W/Q xmm, mem	0	1	1
PADDB/D/W/Q mm, mm; PADDB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PADDSB/W mm, mem; PADDSB/W xmm, mem	0	1	1
PADDSB/W mm, mm; PADDSB/W xmm, xmm	(0, 1)	1	0.5
PADDUSB/W mm, mem; PADDUSB/W xmm, mem	0	1	1
PADDUSB/W mm, mm; PADDUSB/W xmm, xmm	(0, 1)	1	0.5
PAND/PANDN/POR/PXOR mm, mem; PAND/PANDN/POR/PXOR xmm, mem	0	1	1
PAND/PANDN/POR/PXOR mm, mm; PAND/PANDN/POR/PXOR xmm, xmm	(0, 1)	1	0.5
PAVGB/W mm, mem; PAVGB/W xmm, mem	0	1	1
PAVGB/W mm, mm; PAVGB/W xmm, xmm	(0, 1)	1	0.5
PCMPEQB/D/W mm, mem; PCMPEQB/D/W xmm, mem	0	1	1
PCMPEQB/D/W mm, mm; PCMPEQB/D/W xmm, xmm	(0, 1)	1	0.5
PCMPGTB/D/W mm, mem; PCMPGTB/D/W xmm, mem	0	1	1
PCMPGTB/D/W mm, mm; PCMPGTB/D/W xmm, xmm	(0, 1)	1	0.5
PEXTRW;	B	4	1
PHADDD/PHSUBD mm, mem; PHADDD/PHSUBD xmm, mem	B	4	3
PHADDD/PHSUBD mm, mm; PHADDD/PHSUBD xmm, xmm	B	3	2
PHADDW/PHADDSW mm, mem; PHADDW/PHADDSW xmm, mem	B	6; 8	5;7
PHADDW/PHADDSW mm, mm; PHADDW/PHADDSW xmm, xmm	B	5; 7	M
PHSUBW/PHSUBSW mm, mem; PHSUBW/PHSUBSW xmm, mem	B	6; 8	M
PHSUBW/PHSUBSW mm, mm; PHSUBW/PHSUBSW xmm, xmm	B	5; 7	M

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW mm, mem	0	4	1
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm; PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, mem	0	5	1
PMAXSW/UB mm, mem; PMAXSW/UB xmm, mem	0	1	1
PMAXSW/UB mm, mm; PMAXSW/UB xmm, xmm	(0, 1)	1	0.5
PMINSW/UB mm, mem; PMINSW/UB xmm, mem	0	1	1
PMINSW/UB mm, mm; PMINSW/UB xmm, xmm	(0, 1)	1	0.5
PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mm; PMULHUW/PMULHW/PMULLW/PMULUDQ mm, mem	0	4	1
PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, xmm; PMULHUW/PMULHW/PMULLW/PMULUDQ xmm, mem	0	5	1
POP mem[5]; PSLLD/Q/W mm, mem; PSLLD/Q/W xmm, mem	B	3	2
POP r16; PUSH mem[4]; PSLLD/Q/W mm, mm; PSLLD/Q/W xmm, xmm	B	2	1
POP reg[3]; PUSH reg[4]; PUSH imm	B	1	1
POPA ; POPAD	B	9	8
PSHUFB mm, mem; PSHUFD; PSHUFHW; PSHUFLW; PSHUFW	0	1	1
PSHUFB mm, mm; PSLLD/Q/W mm, imm; PSLLD/Q/W xmm, imm	0	1	1
PSHUFB xmm, mem	B	5	4
PSHUFB xmm, xmm	B	4	3
PSIGNB/D/W mm, mem; PSIGNB/D/W xmm, mem	0	1	1
PSIGNB/D/W mm, mm; PSIGNB/D/W xmm, xmm	(0, 1)	1	0.5
PSRAD/W mm, imm; PSRAD/W xmm, imm;	0	1	1
PSRLD/Q/W mm, mem; PSRLD/Q/W xmm, mem	B	3	2
PSRLD/Q/W mm, mm; PSRLD/Q/W xmm, xmm	B	2	1
PSRLD/Q/W mm, imm; PSRLD/Q/W xmm, imm;	0	1	1
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	0	1	1
PSUBB/D/W/Q mm, mem; PSUBB/D/W/Q xmm, mem	0	1	1
PSUBB/D/W/Q mm, mm; PSUBB/D/W/Q xmm, xmm	(0, 1)	1	0.5
PSUBSB/W mm, mem; PSUBSB/W xmm, mem	0	1	1
PSUBSB/W mm, mm; PSUBSB/W xmm, xmm	(0, 1)	1	0.5
PSUBUSB/W mm, mem; PSUBUSB/W xmm, mem	0	1	1
PSUBUSB/W mm, mm; PSUBUSB/W xmm, xmm	(0, 1)	1	0.5
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	0	1	1
PUNPCKHQDQ; PUNPCKLQDQ	0	1	1
PUSHA ; PUSHAD	B	8	7
RCL mem[2], 1; RCL reg[2], 1	0	1	1

Table 6-2. Intel Atom® Microarchitecture Instructions Latency Data (Contd.)

Instruction	Ports	Latency	Throughput
DisplayFamily_DisplayModel	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**	**06_1CH, 06_26H, 06_27H**
RCL m8, CL; RCL m16, CL; RCL mem[3], CL;	B	18;16; 14	17;15;13
RCL m8, imm; RCL m16, imm; RCL mem[3], imm;	B	18; 17; 14	17;16;13
RCL r8, CL; RCL r16, CL; RCL reg[3], CL;	B	17; 16; 14	16;15;14
RCL r8, imm; RCL r16, imm; RCL reg[3], imm;	B	18;16; 14	17;15;13
RCPSS	0	4	1
RCR mem[2], 1; RCR reg[2], 1	B	7; 5	6;4
RCR m8, CL; RCR m16, CL; RCR mem[3], CL;	B	15; 13; 12	14;12;11
RCR m8, imm; RCR m16, imm; RCR mem[3], imm;	B	16,;14; 12	15;13;11
RCR r8, CL; RCR r16, CL; RCR reg[3], CL;	B	14; 13; 12	13;12;11
RCR r8, imm; RCR r16, imm; RCR reg[3], imm;	B	15, 14, 12	14;13;11
RET imm16	B	1	1
RET (far)	B	79	
ROL; ROR; SAL; SAR; SHL; SHR	0	1	1
SETcc		1	1
SHLD[8] mem, reg, imm; SHLD r64, r64, imm; SHLD m64, r64, CL	B	11	10
SHLD m32, r32; SHLD r32, r32	B	4; 2	3; 1
SHLD m16, r16, CL; SHLD r16, r16, imm; SHLD r64, r64, CL	B	10	9
SHLD r16, r16, CL; SHRD m64, r64; SHRD r64, r64, imm	B	9	8
SHRD m32, r32; SHRD r32, r32	B	4; 2	3; 1
SHRD m16, r16; SHRD r16, r16	B	6	5
SHRD r64, r64, CL	B	8	7
STMXCSR	B	15	14
TEST[2] reg, reg; TEST[4] reg, imm	(0, 1)	1	0.5
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	0	1	1

Notes on operand size (osize) and address size (asize):
1. osize = 8, 16, 32 or asize = 8, 16, 32
2. osize = 8, 16, 32, 64
3. osize = 32, 64
4. osize = 16, 32, 64 or asize = 16, 32, 64
5. osize = 16, 32
6. osize = 8, 32
7. osize = 8, 16
8. osize = 16, 64

6.5 SILVERMONT MICROARCHITECTURE

The Intel Atom processor E3000 and C2000 Series are based on the Silvermont microarchitecture. The Silvermont microarchitecture spans a wide range of computing devices from tablets, phones, and PCs to microservers. In addition to support for Intel 64 and IA-32 architecture, major enhancements of the Silvermont microarchitecture include:

- Out-of-order execution for integer instructions and de-coupled ordering between non-integer and memory instructions. In contrast, the 45nm and 32nm Intel Atom microarchitecture was strictly in-order with limited ability to exploit available instruction-level parallelism.
- Non-blocking memory instructions allowing multiple (8) outstanding misses. In previous generation processors, problems in a single memory instruction (for example, a cache miss) caused all subsequent instructions to stall until the problem was resolved. The new microarchitecture allows up to 8 unique outstanding references.
- Modular system design with two cores sharing an L2 cache connected to a new integrated memory controller using a point-to-point interface instead of the Front Side Bus.
- Instruction set enhancements to include SSE 4.1, SSE 4.2, AESNI and PCLMULQDQ.

The block diagram for the Silvermont microarchitecture is depicted in Figure 6-1. While the memory and execute clusters were significantly redesigned for improved single thread performance, the primary focus is still a highly efficient design in a small form factor power envelope. Each pipeline is accompanied with a dedicated scheduling queue called a reservation station. While floating-point and memory instructions schedule from their respective queues in program order, integer execution instructions schedule from their respective queues out of order.

Integer instructions can be scheduled from their queues out of order in contrast to in-order execution in previous generations. Out of order scheduling allows these instructions to tolerate stalls caused by unavailable (re)sources. Memory instructions must generate their addresses (AGEN) in-order and schedule from the scheduling queue in-order but they may complete out-of-order.

Non-integer instructions (including SIMD integer, SIMD floating-point, and x87 floating-point) also schedule from their respective scheduling queue in program order. However, these separate scheduling queues allow their execution to be decoupled from instructions in other scheduling queues.

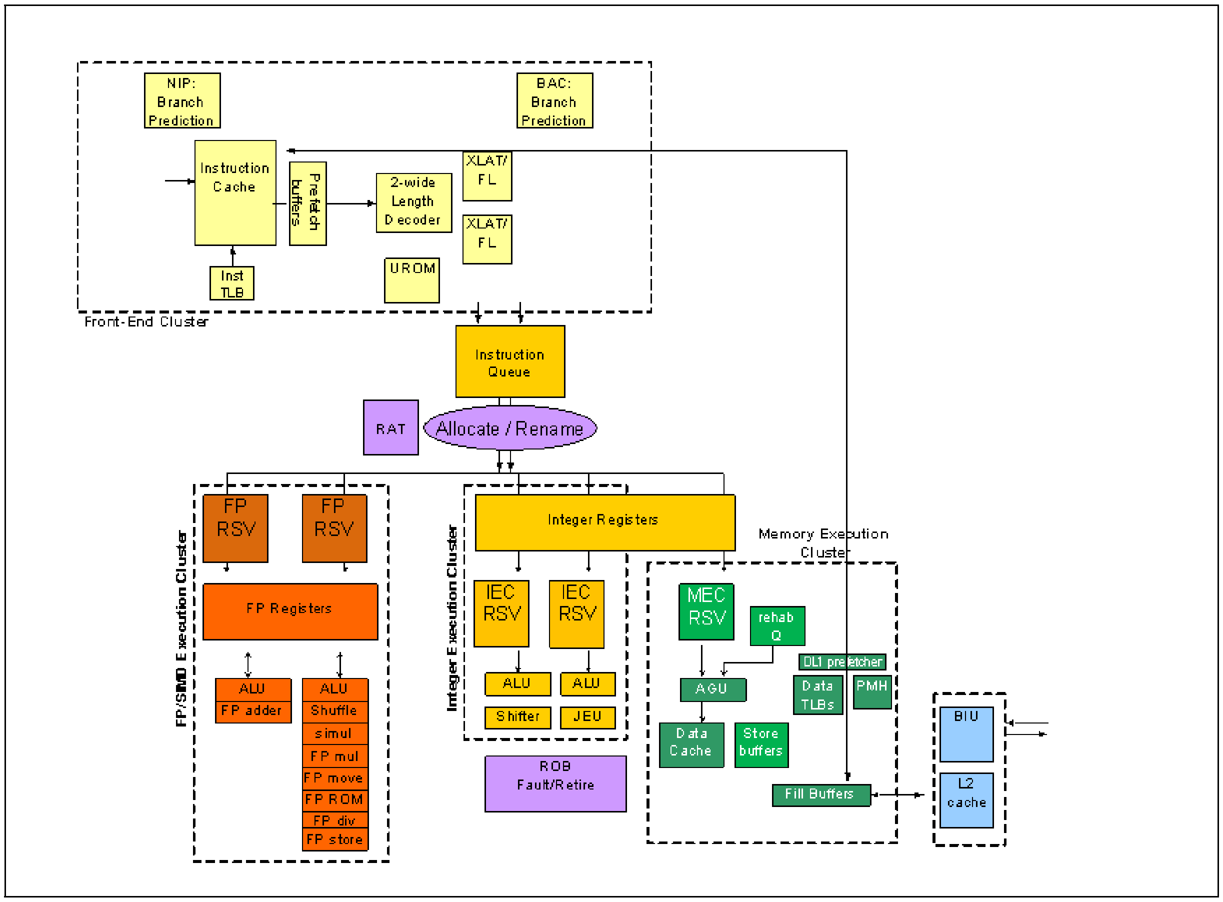


Figure 6-2. Silvermont Microarchitecture Pipeline

The design of the microarchitecture takes into account maximizing platform performance of multiple form factors (e.g. phones, tablets, to micro-servers) and minimizing the power and area cost due to out of order scheduling (i.e. maximizing performance/power/cost efficiency). Intel Hyper-Threading Technology is not supported in favor of a multi-core architecture with a shared L2 cache. The rest of this section will cover some of the cluster-level features in more detail.

The front end cluster (FEC), shown in yellow in Figure 6-1, features a power optimized 2-wide decode pipeline. FEC is responsible for fetching and decoding instructions from instruction memory. FEC utilizes predecode hints from the icache to avoid costly on-the-fly instruction length determination. The front end contains a Branch Target Buffer (BTB), plus advanced branch predictor hardware.

The front end is connected to the execution units through the Allocation, Renaming and Retirement (ARR) cluster (lavender color in Figure 6-1). ARR receives uops from the FEC and is responsible for resource checks. The Register Alias Table (RAT) renames the logical registers to the physical registers. The Reorder Buffer (ROB) puts the operations back into program order and completes (retires) them. It also stops execution at interrupts, exceptions and assists and runs program control over microcode.

Scheduling in the Silvermont microarchitecture is distributed, so after renaming, uops are sent to various clusters (IEC: integer execution cluster; MEC: memory execution cluster; FPC: floating-point cluster) for scheduling (shown as RSV for FP, IEC, and MEC in Figure 6-1).

There are 2 sets of reservation stations for FPC and IEC (one for each port) and a single set of reservation stations for MEC. Each reservation station is responsible for receiving up to 2 ops from the ARR cluster in a cycle and selecting one ready op for dispatching to execution as soon as the op becomes ready.

To support the distributed reservation station concept, load-op and load-op-store macro-instructions requiring integer execution must be split into a memory sub-op that is sent to the MEC and resides in the memory reservation station and an integer execution sub-op that is sent to the integer reservation station. The IEC schedulers pick the oldest ready instruction from each of its RSVs while the MEC and the FPC schedulers only look at the oldest instruction in their respective RSVs. Even though the MEC and FPC clusters employ in-order schedulers, a younger instruction from a particular FPC RSV can execute before an older instruction in the other FPC RSV for example (or the IEC or MEC RSVs).

Each execution port has specific functional units available. Table 6-3 shows the mapping of functional units to ports for IEC (the orange units in Figure 6-1), MEC (the green units in Figure 6-1), and the FPC (the red units in Figure 6-1). Compared to the previous Intel Atom microarchitecture, the Silvermont microarchitecture adds an integer multiply unit (IMUL) in IEC.

Table 6-3. Function Unit Mapping of the Silvermont Microarchitecture

Cluster	Port 0	Port 1
IEC	ALU0, Shift/Rotate Unit, LEA with no index	ALU1, Bit processing unit, Jump unit, IMUL, POPCNT, CRC32, LEA[1]
FPC	SIMD ALU, SIMD shift/Shuffle unit, SIMD FP mul/div/cvt unit, STTNI/AESNI/PCLMULQDQ unit, RCP/RSQRT unit, F2I convert unit	SIMD ALU, SIMD FPadd unit, F2I convert unit
MEC	Load/Store	

NOTES:

1. LEAs with valid index and displacement are split into multiple UOPs and use both ports. LEAs with valid index execute on port 1.

The Memory Execution Cluster (MEC) (shown in green in Figure 6-1) can support both 32-bit and 36-bit physical addressing modes. The Silvermont microarchitecture has a 2 level Data TLB hierarchy with support for both large (2MB or 4MB) and small page structures. A small micro TLB (referred to as uTLB) is backed up by a larger 2nd level TLB (referred to as DTLB). A hardware page walker services misses from both the Instruction and Data TLBs.

The MEC also owns the MEC RSV, which is responsible for scheduling of all loads and stores. Load and store instructions go through addresses generation phase in program order to avoid on-the-fly memory ordering later in the pipeline. Therefore, an unknown address will stall younger memory instructions. Memory operations that incur problems (e.g. uTLB misses, unavailable resources, etc.) are put in a separate queue called the RehabQ. This allows younger instructions (that do not incur problems) to continue execution rather than stalling all younger instructions. The problematic instruction is later reissued from the RehabQ when the problem is resolved. Note that load misses are not considered problematic as the Silvermont microarchitecture features a non-blocking data cache that can sustain 8 outstanding misses.

The Bus Cluster (BIU) includes the second-level cache (L2) and is responsible for all communication with components outside the processor core. The L2 cache supports up to 1MB with an optimized latency less than the previous Intel Atom microarchitecture. The Front-Side Bus from earlier Intel Atom processors has been replaced by an intra-die interconnect (IDI) fabric connecting to a newly optimized memory controller. The BIU also houses the L2 data prefetcher.

The new core level multi-processing (or CMP) system configuration features two processor cores making requests to a single BIU, which will handle the multiplexing between cores. This basic CMP module can be replicated to create a quad-core configuration, or one core chopped off to create a single-core configuration.

6.5.1 Integer Pipeline

Load pipeline stages are no longer inlined with the rest of the integer pipeline. As a result, non-load ops can reach execute faster, and the branch misprediction penalty is effectively 3 cycles less compared to earlier Intel Atom processors. Front end pipe stages are the same as earlier Intel Atom processors (3 cycles for fetch, 3 cycles for decode). ARR pipestages perform out-of-order allocation and register renaming, split the uop into parts if necessary, and send them to the distributed reservation stations. RSV stage is where the distributed reservation station performs its scheduling. The execution pipelines are very similar to earlier Intel Atom processors. When all parts of a uop are marked as finished, the ROB handles final completion in-order.

6.5.2 Floating-Point Pipeline

Compared to the INT pipeline, the FP pipeline is longer. The execution stages can vary between one and five depending on the instruction. Like other Intel microarchitectures, the Silvermont microarchitecture needs to limit the number of FP assists (when certain floating-point operations cannot be handled natively by the execution pipeline, and must be performed by microcode) to the bare minimum to achieve high performance. To do this the processor should be run with exceptions masked and the DAZ (denormal as zero) and FTZ (flush to zero) flags set whenever possible.

As mentioned, while each FPC RSV schedules instructions in-order, the RSVs can get out of order with respect to each other.

6.6 GOLDMONT MICROARCHITECTURE

The Goldmont microarchitecture builds on the success of the Silvermont microarchitecture (see Section 6.5), and provides the following enhancements:

- An out-of-order execution engine with a 3-wide superscalar pipeline. Specifically:
  - — The decoder can decode 3 instructions per cycle.
  - — The microcode sequencer can send 3 uops per cycle for allocation into the reservation stations.
  - — Retirement supports a peak rate of 3 per cycle.
- Enhancement in branch prediction which de-couples the fetch pipeline from the instruction decoder.
- Larger out-of-order execution window and buffers that enable deeper out-of-order execution across integer, FP/SIMD, and memory instruction types.
- Fully out-of-order memory execution and disambiguation. The Goldmont microarchitecture can execute one load and one store per cycle (compared to one load or one store per cycle in the Silvermont microarchitecture). The memory execution pipeline also includes a second level TLB enhancement with 512 entries for 4KB pages.
- Integer execution cluster in the Goldmont microarchitecture provides three pipelines and can execute up to three simple integer ALU operations per cycle.
- SIMD integer and floating-point instructions execute in a 128-bit wide engine. Throughput and latency of many instructions have improved, including PSHUFB with 1-cycle throughput (versus 5 cycles for Silvermont microarchitecture) and many other SIMD instructions with doubled throughput; see Table 6-19 for details.
- Throughput and latency of instructions for accelerating encryption/description (AES) and carry-less multiplication (PCLMULQDQ) have been improved significantly in the Goldmont microarchitecture.
- The Goldmont microarchitecture provides new instructions with hardware accelerated secure hashing algorithm, SHA1 and SHA256.
- The Goldmont microarchitecture also adds support for the RDSEED instruction for random number generation meeting the NIST SP800-90C standard.
- PAUSE instruction latency is optimized to enable better power efficiency.

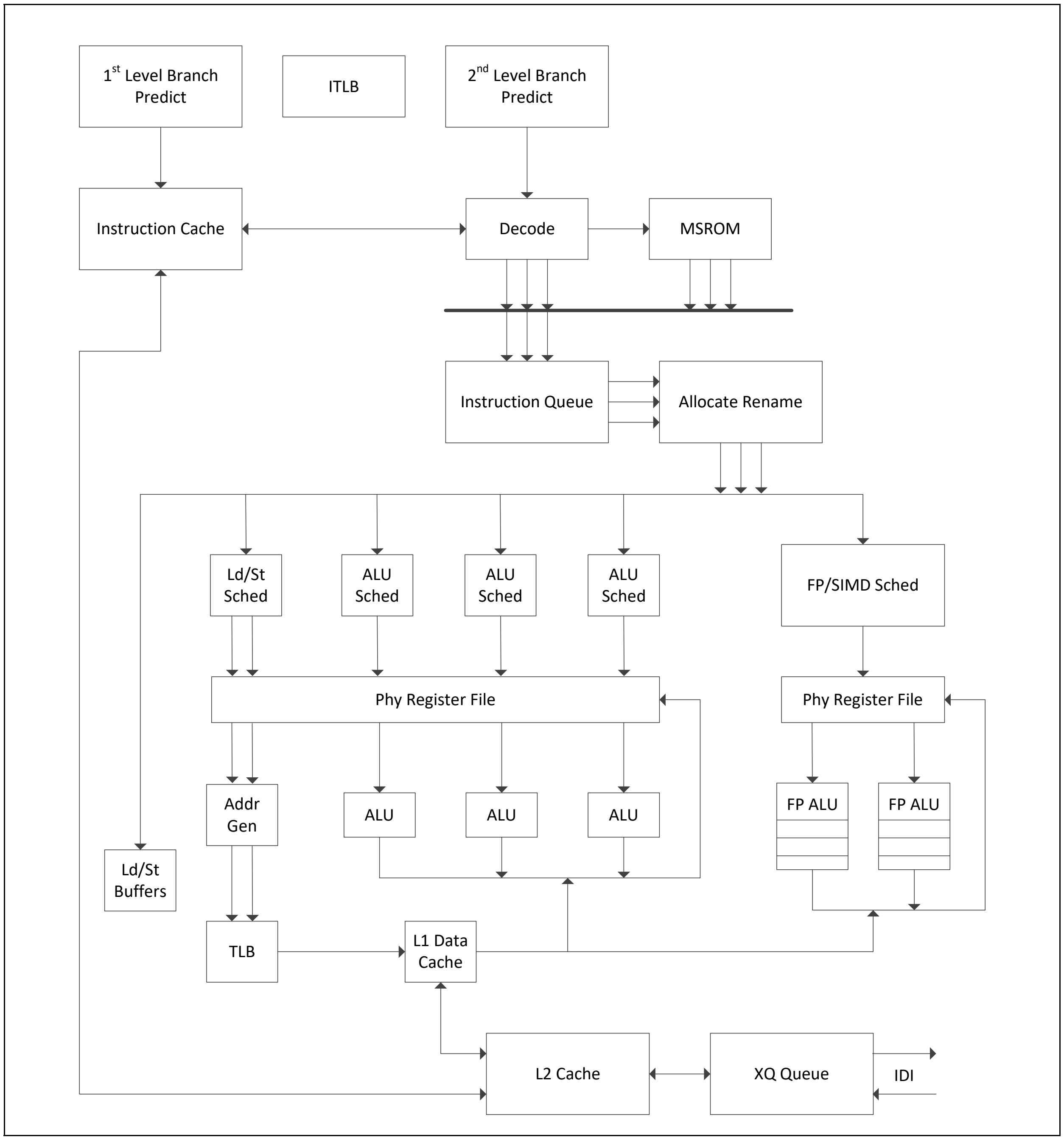


Figure 6-3. CPU Core Pipeline Functionality of the Goldmont Microarchitecture

The front end cluster (FEC) of the Goldmont microarchitecture provides a number of enhancements over the FEC of the Silvermont microarchitecture. The enhancements are summarized in Table 6-4.

Table 6-4. Comparison of Front End Cluster Features

Feature	Goldmont Microarchitecture	Silvermont Microarchitecture
Number of Decoders	3	2
Max Throughput of Decoders	20 Bytes per cycle	16 Bytes per cycle
Fetch and Icache Pipeline	Decoupled	Coupled
ITLB	48 entries, large page support	48 entries
Branch Mispredict Penalty	12 cycles	10 cycles
L2 Predecode Cache	16K	NA

The FEC is connected to the execution units through the Allocation, Renaming and Retirement (ARR) cluster. Scheduling of uops is handled with distributed reservation stations across different clusters (IEC, FPC, MEC). Each cluster has its own reservations for receiving multiple uops from the ARR. Table 6-5 compares the out-of-order scheduling characteristics between the Goldmont microarchitecture and Silvermont microarchitecture.

Table 6-5. Comparison of Distributed Reservation Stations on Scheduling Uops

Cluster	Goldmont Microarchitecture	Silvermont Microarchitecture
IEC Reservation	3x distributed for each port	2x distributed for each port
	Out-of-order within each IEC RSV and between IEC, across FPC, MEC	Out-of-order within each IEC RSV and between IEC, across FPC, MEC
FPC Reservation	1x unified to ports 0, 1	2x distributed for each port
	Out-of-order within FPC RSV and across IEC, MEC	In order within each FPC RSV; out-of-order between FPC, across IEC, MEC
MEC Reservation	1x unified to ports 0, 1	1x to port 0
	Out-of-order within MEC RSV and across IEC, FPC	In order within each MEC RSV; out-of-order across IEC, FPC

An instruction that references memory and requires integer/FP resources will have the memory uop sent to the MEC cluster and the integer/FP uop sent to the IEC/FPC cluster. Then out-of-order execution can commence according to the heuristic described in Table 6-5 and when resources are available. Table 6-6 shows the mapping of execution units across each port for respective clusters.

Table 6-6. Function Unit Mapping of the Goldmont Microarchitecture

Cluster	Port 0	Port 1	Port 2
IEC	ALU0, Shift/Rotate, LEA with no index, F2I, converts/cmp, store_data	ALU1, Bit processing, JEU, IMUL, IDIV,POPCNT, CRC32, LEA, I2F, store_data	ALU2, LEA[1], I2F, flag_merge
FPC	SIMD ALU, SIMD shift/Shuffle, SIMD mul, STTNI/AESNI/PCLMULQDQ/SHA ; FP_mul, Converts, F2I convert	SIMD ALU, SIMD shuffle, FP_add, F2I compare	
MEC	Load_addr	Store_addr	

NOTES:

1. LEAs without index can execute on port 0, 1, or 2. LEA with valid index and displacement are split into multiple UOPs and use both port 1 and 2. LEAs with valid index execute on port 1.

The MEC owns the MEC RSV and is responsible for scheduling all load and stores via ports 0 and 1. Load and store instructions can go through the address generation phase in order or out-of-order. When out-of-order address generation scheduling is available, memory execution pipeline is de-coupled from the address generation pipeline using the load buffers and store buffers.

With out-of-order execution, situations where loads can pass an unknown store may cause memory order issues if the load eventually depended on the unknown store and would require a pipeline flush when the store ad-dress is known. The Goldmont microarchitecture keeps track of and minimizes such potentially problematic load executions.

Memory operations that experienced problems (for example, uTLB misses and unavailable resources) go back to load or store buffer for re-execution. This allows younger instructions (that do not incur problems) to continue execution rather than stalling all younger instructions. The problematic instruction is later re-issued (in some cases, re-issued at retirement) from the load/store buffer when the problem is resolved. Note that load misses are considered problematic as the data cache is non-blocking and can sustain multiple outstanding misses using write-combining buffers (WCB).

Table 6-7. Comparison of MEC Resources

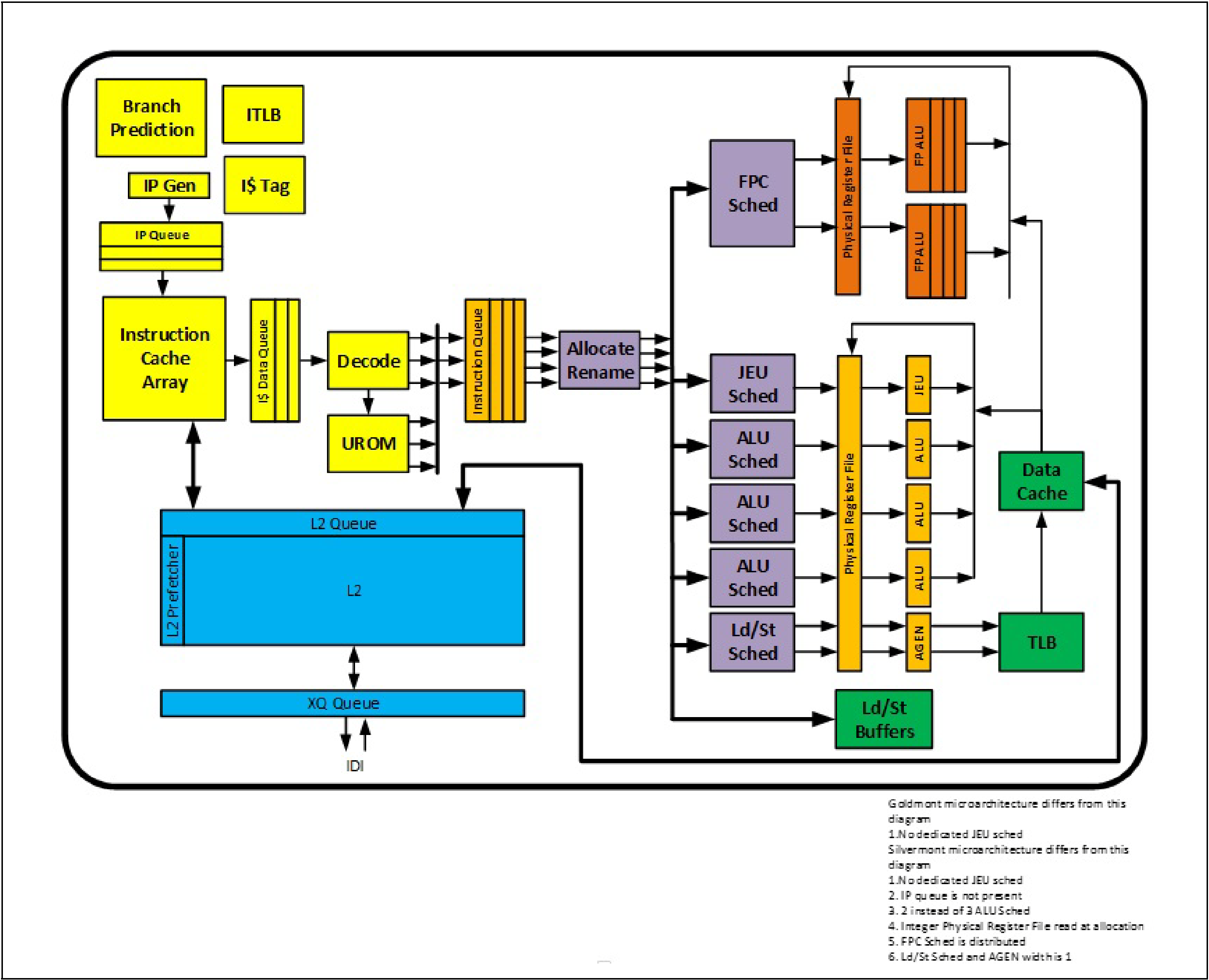
MEC Resource	**Goldmont Microarchitecture**	**Silvermont Microarchitecture**
L1 Data Cache	24KB	24 KB
uTLB	32 entries	32 entries
DTLB (4KB page)	512 entries	128 entries
DTLB (2M/4M page)	32 entries	16 entries
Load-use Latency	3 cycles	3 cycles
Pipeline	1x load + 1x store	1x share by load/store
AGEN	Out-of-order	In order
WCBs	8	8
Addressing	39-bit physical, 48-bit linear	36-bit physical, 48-bit linear

6.7 GOLDMONT PLUS MICROARCHITECTURE

The Goldmont Plus microarchitecture builds on the success of the Goldmont microarchitecture (see Section 6.6), and provides the following enhancements:

- Widen previous generation Intel Atom processor back-end pipeline to 4-wide allocation to 4-wide retire, while maintaining 3-wide fetch and decode pipeline.
- Enhanced branch prediction unit.
- 64KB shared second level pre-decode cache (16KB in Goldmont microarchitecture).
- Larger reservation station and ROB entries to support large out-of-order window.
- Wider integer execution unit. New dedicated JEU port with support for faster branch redirection.
- Radix-1024 floating point divider for fast scalar/packed single, double and extended precision floating point divides.
- Improved AES-NI instruction latency and throughput.
- Larger load and store buffers. Improved store-to-load forwarding latency store data from register.
- Shared instruction and data second level TLB. Paging Cache Enhancements (PxE/ePxE caches).
- Modular system design with four cores sharing up to 4MB L2 cache.

- Support for Read Processor ID (RDP) new instruction.



Goldmont microarchitecture differs from this diagram
1.No dedicated JEU sched
Silvermont microarchitecture differs from this diagram
1.No dedicated JEU sched
2. IP queue is not present
3. 2 instead of 3 ALU Sched
4. Integer Physical Register File read at allocation
5. FPC Sched is distributed
6. Ld/St Sched and AGEN width is 1

Figure 6-4. CPU Core Pipeline Functionality of the Goldmont Plus Microarchitecture

The front end cluster (FEC) of the Goldmont Plus microarchitecture provides a number of enhancements over the FEC of the Goldmont microarchitecture. The enhancements are summarized in Table 6-8.

Table 6-8. Comparison of Front End Cluster Features

Feature	Goldmont Plus Microarchitecture	Goldmont Microarchitecture
Number of Decoders	3	3
Max. Throughput Decoders	20 Bytes per cycle	20 Bytes per cycle
Fetch and Icache Pipeline	Decoupled	Decoupled
ITLB	48 entries, large page support	48 entries, large page support
2nd Level ITLB	Shared with DTLB	
Branch Mispredict Penalty	13 cycles (12 cycles for certain Jcc)	12 cycles
L2 Predecode Cache	64K	16K

The FEC is connected to the execution units through the Allocation, Renaming and Retirement (ARR) cluster. Scheduling of uops is handled with distributed reservation stations across different clusters (IEC, FPC, MEC). Each cluster has its own reservations for receiving multiple uops from the ARR. Table 6-9 compares the out-of-order scheduling characteristics between the Goldmont Plus microarchitecture and Goldmont microarchitecture.

Table 6-9. Comparison of Distributed Reservation Stations on Scheduling Uops

Cluster	Goldmont Plus Microarchitecture	Goldmont Microarchitecture
IEC Reservation	4x distributed for each port	3x distributed for each port
	Out-of-order within each IEC RSV and between IEC, across FPC, MEC	Out-of-order within each IEC RSV and between IEC, across FPC, MEC
FPC Reservation	1x unified to ports 0, 1	1x unified to ports 0, 1
	Out-of-order within FPC RSV and across IEC, MEC	Out-of-order within FPC RSV and across IEC, MEC
MEC Reservation	1x unified to ports 0, 1	1x unified to ports 0, 1
	Out-of-order within MEC RSV and across IEC, FPC	Out-of-order within MEC RSV and across IEC, FPC

An instruction that references memory and requires integer/FP resources will have the memory uop sent to the MEC cluster and the integer/FP uop sent to the IEC/FPC cluster. Then out-of-order execution can commence according to the heuristic described in Table 6-9 when resources are available. Table 6-10 shows the mapping of execution units across each port for respective clusters.

Table 6-10. Function Unit Mapping of the Goldmont Plus Microarchitecture

Cluster	Port 0	Port 1	Port 2	Port 3
IEC	ALU0, Shift/Rotate, LEA with no index, F2I, converts/cmp, store_data	ALU1, Bit processing, IMUL, IDIV,POPCNT, CRC32, LEA, I2F, store_data	ALU2, LEA[1], I2F, flag_merge	JEU
FPC	SIMD ALU, SIMD shift/Shuffle, SIMD mul, STTNI/AESNI/PCLMULQDQ/ SHA ; FP_mul, Converts, F2I convert	SIMD ALU, SIMD shuffle, FP_add, F2I compare		
MEC	Load_addr	Store_addr		

NOTES:

1. LEAs without index can execute on port 0, 1, or 2. LEA with a valid index and displacement are split into multiple UOPs and use both port 1 and 2. LEAs with a valid index execute on port 1.

The MEC owns the MEC RSV and is responsible for scheduling all load and stores via ports 0 and 1. Load and store instructions can go through the address generation phase in order or out-of-order. When out-of-order address generation scheduling is available, memory execution pipeline is de-coupled from the address generation pipeline using the load buffers and store buffers.

With out-of-order execution, situations where loads can pass an unknown store may cause memory order issues if the load eventually depended on the unknown store and would require a pipeline flush when the store address is known. The Goldmont Plus microarchitecture keeps track of and minimizes such potentially problematic load executions.

Memory operations that experienced problems (for example, uTLB misses and unavailable resources) go back to the load or store buffer for re-execution. This allows younger instructions (that do not incur problems) to continue execution rather than stalling all younger instructions. The problematic instruction is later re-issued from the load/store buffer when the problem is resolved. Note that load misses are considered problematic as the data cache is non-blocking and can sustain multiple outstanding misses using write-combining buffers (WCB).

Goldmont Plus microarchitecture includes secondary level TLB changes to support both data and instruction side translations (Goldmont microarchitecture secondary level TLB only supports data).

6.8 CODING RECOMMENDATIONS

Most of the general coding recommendations described in Volume 1, Chapter 3, "General Optimization Guidelines" also apply to the Intel Atom microarchitectures. The rest of this chapter describes techniques that
supplement the general recommendations and are specific to the Intel Atom microarchitectures.

6.8.1 Optimizing The Front End

6.8.1.1 Instruction Decoder

Some IA instructions that perform complex tasks require a lookup in the microcode sequencer ROM (MSROM) to decode them into a multiple uop flow. To determine which instructions require an MSROM lookup, see the instruction latency/bandwidth table in Section 6.9.

Fewer instructions require MSROM lookup in the Goldmont Plus and Goldmont microarchitecture than in the Silvermont microarchitecture, though the Silvermont microarchitecture also improved significantly over prior generations in this area; Section 6.9 provides more details. It is advisable to avoid ucode flows where possible. Table 6-11 provides alternate non-MSROM instruction sequences that can replace an instruction that decodes from MSROM.

Table 6-11. Alternatives to MSROM Instructions

Instruction from MSROM	Recommendation for Silvermont	Recommendation for Goldmont Plus and Goldmont
CALL m16/m32/m64	Load + CALL reg	Load + CALL reg
PUSH m16/m32/m64	Load + PUSH reg	Use as is (non MSROM)
LEAVE	No recommended replacement	Use as is (non MSROM)
FLD/FST/FSTP m80fp	No recommended replacement	Use as is (non MSROM)
FCOM+FNSTSW	FCOMI	FCOMI
(I)MUL r/m16 (Result DX:AX)	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32	Use (I)MUL r16, r/m16 if extended precision not required, or (I)MUL r32, r/m32
(I)MUL r/m32 (Result EDX:EAX)	Use (I)MUL r32, r/m32 if extended precision not required, or (I)MUL r64, r/m64	Use as is (non MSROM)
(I)MUL r/m64 (Result RDX:RAX)	Use (I)MUL r64, r/m64 if extended precision not required	Use as is (non MSROM)
PEXTRB/D/Q	No recommended replacement	Use as is (non MSROM)
PMULLD	No recommended replacement	Use as is (non MSROM)

Tuning Suggestion 1. *Use the perfmon counter MS_DECODED.MS_ENTRY to find the number of instructions that need the MSROM (the count will include any assist or fault that occurred).*

Assembly/Compiler Coding Rule 1. (M impact, M generality) *Try to keep the I-footprint small to get the best reuse of the predecode bits.*

Avoid I-cache aliasing/thrashing since the *incorrect* predecode bits result in reduction of decode throughput in one instruction every 3 cycles.

Tuning Suggestion 2. *Use the perfmon counter DECODE_RESTRICTION.PREDECODE_WRONG to count the number of times that a decode restriction reduced instruction decode throughput because predecoded bits are incorrect.*

6.8.1.2 Front End High IPC Considerations

In general front end restrictions are not typically a performance limiter until you reach higher (>1) Instructions Per Cycle (IPC) levels.

The decode restrictions that must be followed to get full decode bandwidth per cycle through the decoders include:

- MSROM instructions should be avoided if possible. A good example is the memory form of CALL near indirect. It will often be better to perform a load into a register and then perform the register version of CALL.
- The total length of the instruction bytes that can be decoded each cycle varies by microarchitecture.
  - Silvermont microarchitecture: up to 16 bytes per cycle with instruction not more than 8 bytes in length. For an instruction length exceeding 8 bytes, only one instruction per cycle is decoded on decoder 0.
  - Goldmont and later microarchitecture: up to 20 bytes per cycle depending on alignment (for example, if the first instruction of three consecutive instructions is aligned on 4-Byte boundary and the 3 instruction sequence meets decode restrictions. For an instruction length exceeding 8 bytes, it is not restricted to decoder 0 or one per cycle.
- An instruction with multiple prefixes can restrict decode throughput. The restriction is on the length of bytes combining prefixes and escape bytes. There is a 3 cycle penalty when the escape/prefix count exceeds the following limits as specified per microarchitectures.
  - Silvermont microarchitecture: the limit is 3 bytes.
  - Goldmont and later microarchitecture: the limit is 4 bytes. Thus, SSE4 or AES instruction that accesses one of the upper 8 registers do not incur a penalty.
  - Only decoder 0 can decode an instruction exceeding the limit of prefix/escape byte restriction on the Silvermont and Goldmont microarchitectures.
- The maximum number of branches that can be decoded each cycle is 1 for the Silvermont microarchitecture and 2 for the Goldmont microarchitecture. Prevent a re-steer penalty by avoiding back-to-back conditional branches.

Unlike the previous generation, the Silvermont and later microarchitectures can decode two x87 instructions in the same cycle without incurring a 2-cycle penalty. Branch decoder restrictions are also relaxed. In earlier Intel Atom processors, decoding past a conditional or indirect branch in decoder 0 resulted in a 2-cycle penalty.

The Silvermont microarchitecture can decode past conditional and indirect branch instructions in decoder 0. However, if the next instruction (on decoder 1) is also a branch, there is a 3-cycle penalty for the second branch instruction.

The Goldmont and later microarchitecture can decode one predicted not-taken branches in decoder 0 or decoder 1, plus another branch in decoder 2 without the 3-cycle re-steer penalty. However, if there are two predicted not-taken branches at decoder 0 and 1, the second branch at decoder 1 will incur a 3-cycle penalty.

For a branch target that is a predicted taken conditional branch or unconditional branch, it is decoded with a one cycle bubble across all generations of Intel Atom processors.

Assembly/Compiler Coding Rule 2. (MH impact, H generality) *Minimize the use of instructions that have the following characteristics to achieve more than one instruction per cycle throughput: (i) using the MSROM, (ii) exceeding the limit of escape/prefix bytes, (iii) more than 8 bytes long, or (iv) have back to back branches.*

For example, an instruction with 3 bytes of prefix/escape and accessing the lower 8 registers can decode normally in the Silvermont, Goldmont and later microarchitectures. For instance:

```
PCLMULQDQ 66 0F 3A 44 C7 01   pclmulqdq xmm0, xmm7, 0x1
```

To access any of the upper 8 XMM registers, XMM8-15, an additional byte with REX prefix is necessary. Consequently, it will decode normally in the Goldmont and later microarchitecture, but incur a decode penalty in the Silvermont microarchitecture. For instance:

```
PCLMULQDQ 66 41 0F 3A 44 C0 01 pclmulqdq xmm0, xmm8, 0x1
```

(Note the REX byte 41, in between the 66 and the 0F 3A.)

The 3-cycle penalty applies whenever the combined prefix/escape bytes exceed the decode restriction limit. Also, it forces the instruction to be decoded on decoder 0. Additionally, when decoding an instruction exceeding the prefix/escape length limit, not on decoder 0, there is an extra delay to re-steer to decoder 0 (for a total of a 6 cycle penalty for the decoder). Therefore, when hand writing high performance assembly, be aware of these cases. It would be beneficial to pre-align these cases to decoder 0 if they occur infrequently using a taken branch target or MS entry point as a decoder 0 alignment vehicle. NOP insertion should be used only as a last resort as NOP instructions consume resources in other parts of the pipeline. Similar alignment is necessary for MS entry points which suffer the additional 3 cycle penalty if they align originally to decoder 1. The penalty associated with a prefix/escape length limit and re-steer apply to both Silvermont, Goldmont and later microarchitectures.

Table 6-12 compares decoder capabilities between microarchitectures.

Table 6-12. Comparison of Decoder Capabilities

	Goldmont Plus and Goldmont Microarchitecture	Silvermont Microarchitecture
Width	3	2
Max Throughput	20 bytes per cycle (1st instr. aligned to 4B boundary and decoder 1 and 2 restrictions)	16 bytes per cycle (1st instr. <= 8 bytes))
Prefix/Escape Limit	4 bytes	3 bytes
Branch	2	1

6.8.1.3 Branching Across 4GB Boundary

Another important performance consideration from a front end standpoint is branch prediction. For 64-bit applications, branch prediction performance can be negatively impacted when the target of a branch is more than 4GB away from the branch. This is more likely to happen when the application is split into shared libraries. Newer glibc versions can put the shared libraries into the first 2GB to avoid this problem (since 2.23). The environment variable LD_PREFER_MAP_32BIT_EXEC=1 has to be set. Developers can build statically to improve the locality in their code. Building with LTO should further improve performance.

6.8.1.4 Loop Unrolling and Loop Stream Detector

The Silvermont and later microarchitectures include a Loop Stream Detector (LSD) that provides the back end with uops that are already decoded. This provides performance and power benefits. When the LSD is engaged, front end decode restrictions, such as number of prefix/escape bytes and instruction length, no longer apply.

One way to reduce the overhead of loop maintenance code and increase the amount of independent work in a loop is software loop unrolling. Unfortunately care must be taken on where it is utilized because loop unrolling has both positive and negative performance effects. The negative performance effects are caused by the increased code size and increased BTB and register pressure. Furthermore, loop unrolling can increase the loop size beyond the limits of the LSD. The LSD loop size limit varies with microarchitecture; it is 27 for the Goldmont and later microarchitecture with a three-wide decoder, and 28 for the Silvermont microarchitecture. Care must be taken to keep the loop size under the LSD limit.

User/Source Coding Rule 1. (M impact, M generality) *Keep per-iteration instruction count below 28 when considering loop unrolling technique on short loops with high iteration count.*

Tuning Suggestion 3. *Use the BACLEARS.ANY perfmon counter to see if the loop unrolling is causing too much pressure. Use the ICACHE.MISSES perfmon counter to see if loop unrolling is having an excessive negative effect on the instruction footprint.*

6.8.1.5 Mixing Code and Data

Intel Atom processors perform best when code and data are on different pages. Software should avoid sharing code and data in the same page to avoid false SMC conditions. This recommendation applies to all page sizes.

6.8.2 Optimizing The Execution Core

6.8.2.1 Scheduling

The Silvermont microarchitecture is less sensitive to instruction ordering than its predecessors due to the introduction of out-of-order execution for integer instructions. FP instructions have their own reservation stations but still execute in order with respect to each other. Memory instructions also issue in order but with the addition of the Rehab Queue, they can complete out of order and memory system delays are no longer blocking.

The Goldmont and later microarchitecture features fully out-of-order execution across the IEC, FPC, and MEC pipelines, and is supported by enhancements ranging from 3 ports for IEC, 128-bit data path of FPC units, dedicated load address and store address pipelines.

Tuning Suggestion 4. *Use the perfmon counter UOPS_NOT_DELIVERED.ANY (NO_ALLOC_CYCLE.ANY on Silvermont microarchitecture) as an indicator of performance bottlenecks in the back end. This includes delays in the memory system and execution delays.*

6.8.2.2 Address Generation

The Silvermont microarchitecture eliminated address generation limitations in previous generations. As such, using LEA or ADD instructions to generate addresses are equally effective on the Silvermont and later microarchitectures.

The rule of thumb for ADDs and LEAs is that it is justified to use LEA with a valid index and/or displacement for non-destructive destination purposes (especially useful for stack offset cases), or to use a SCALE. Otherwise, ADD(s) are preferable.

6.8.2.3 FP Multiply-Accumulate-Store Execution

With Goldmont and later microarchitectures, a unified FPC reservation station eliminates the performance issue that can happen in the Silvermont microarchitecture due to intra-port dependence of in-order scheduling of FPC uops. The paragraphs below and Example 6-7 illustrate the problem.

FP arithmetic instructions executing on different ports can execute out-of-order with respect to each other in the Silvermont microarchitecture. As a result, in unrolled loops with multiplication results feeding into add instructions which in turn produce results for store instructions, grouping the store instructions at the end of the loop will improve performance. This allows it to overlap the execution of the multiplies and the adds. Consider the example shown in Example 6-7.

Example 6-7. Unrolled Loop Executes In-Order Due to Multiply-Store Port Conflict

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5													
addps xmm1, xmm1						EX1	EX2	EX3										
movaps mem, xmm1									EX1									
mulps, xmm2, xmm2										EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2															EX1	EX2	EX3	
movaps mem, xmm2																		EX1

Due to the data dependence, the add instructions cannot start executing until the corresponding multiply instruction is executed. Because multiplies and stores use the same port, they have to execute in program order. This means the second multiply instruction cannot start execution even though it is independent from the first multiply and add instructions. If you group the store instructions together at the end of the loop as shown below, the second multiply instruction can execute in parallel with the first multiply instruction (note the one-cycle bubble when multiplies are overlapped).

Example 6-8. Grouping Store Instructions Eliminates Bubbles and Improves IPC

Instruction	1	2	3	4	5	6	7	8	9	10	11
mulps, xmm1, xmm1	EX1	EX2	EX3	EX4	EX5						
addps xmm1, xmm1						EX1	EX2	EX3			
mulps, xmm2, xmm2		bubble	EX1	EX2	EX3	EX4	EX5				
addps xmm2, xmm2								EX1	EX2	EX3	
movaps mem, xmm1									EX1		
movaps mem, xmm2											EX1

6.8.2.4 Integer Multiply Execution

The Silvermont and later microarchitectures have a dedicated integer multiplier to accelerate commonly-used forms of integer multiply flows. Table 6-13 shows the latency and instruction forms of mul/imul instructions that are accelerated and not using MSROM.

Table 6-13. Integer Multiply Operation Latency

Integer Multiply Operations	Output	Goldmont Plus and Goldmont Latency	Silvermont Latency
imul/mul r/m8	16	4^u	5^u
imul/mul r/m16	32	4^u	5^u
imul/mul r/32	64	3	4^u
imul/mul r/m64	128	5	7^u
imul/mul r16, r/m16; r16, r/m16, imm	16	4^u	4^u
imul/mul r32, r/m32; r32, r/m32, imm	32	3	3
imul/mul r64, r/m64; r64, r/m64, imm8	64	5	5
u: ucode flow from MSROM			

The multiply forms with microcode flows should be avoided.

6.8.2.5 Zeroing Idioms

XOR / PXOR / XORPS / XORPD instructions are commonly used to force register values to zero when the source and the destination register are the same (e.g. XOR eax, eax).

This method of zeroing is preferred by compilers instead of the equivalent MOV eax, 0x0 instructions as the MOV encoding is larger than the XOR in code bytes.

The Silvermont and later microarchitectures have special hardware support to recognize these cases and mark both the sources as valid in the architectural register file. This helps the XOR execute faster since any value XORed with itself will accomplish the necessary zeroing.

The logic will also support PXOR, XORPS, and XORPD idioms.

In Silvermont microarchitecture, zero-idiom, a 64-bit general purpose operand using REX.W, will experience delay. However, zero-idiom is supported with XMM8-XMM15 or the upper 8 general purpose registers without REX.W. To clear r8, it is sufficient to use XOR r8d, r8d.

Goldmont and later microarchitecture supports these zero-idioms for 64-bit operands.

6.8.2.6 NOP Idioms

NOP instruction is often used for padding or alignment purposes. The Goldmont and later microarchitecture has hardware support for NOP handling by marking the NOP as completed without allocating it into the reservation station. This saves execution resources and bandwidth. Retirement resource is still needed for the eliminated NOP.

6.8.2.7 Move Elimination and ESP Folding

Move elimination is supported in Goldmont and later microarchitecture. When move elimination is in effect, these instructions can execute with higher throughput in addition to zero-cycle latency. Specifically, 32-bit and 64-bit operand size of MOV, and MOVAPS/MOVAPD/MOVDQA/MOVDQU/MOVUPS/MOVUPD with XMM are supported and have throughput of 0.33 cycle if move elimination is in effect. MOVSX and MOVZX do not support move elimination.

Stack operation using PUSH/POP/CALL/RET is more efficient with the Goldmont and later microarchitecture than with the Silvermont microarchitecture. Computing the stack pointer address does not consume allocation and execution resources in the Goldmont and later microarchitecture. Additionally, throughput of PUSH/POP is increased from one to three per cycle.

6.8.2.8 Stack Manipulation Instruction

The memory forms of indirect CALL m16/m32/m64 are decoded into a uop flow from MSROM. Indirect CALL with target specified in a register can avoid the delays. Thus, loading the target address to a register, followed by an indirect CALL via register operand is recommended.

In the Goldmont and later microarchitecture, PUSH m16/m32/m64 do not require MSROM to decode. The same is also true with the LEAVE instruction.

In the Silvermont microarchitecture, PUSH m16/m32/m64 and LEAVE require MSROM to decode.

6.8.2.9 Flags usage

Many instructions have an implicit data result that is captured in a flags register. These results can be consumed by a variety of instructions such as conditional moves (cmovs), branches and even a variety of logic/arithmetic operations (such as rcl). The most common instructions used in computing branch conditions are compare instructions (CMP). Branches dependent on the CMP instruction can execute in the next cycle. The same is true for branch instructions dependent on ADD or SUB instructions.

INC and DEC instructions require an additional uop to merge the flags as they are partial flag writers. As a result, a branch instruction depending on an INC or a DEC instruction incurs a 1 cycle penalty.
Note that this penalty only applies to branches that are directly dependent on the INC or DEC instruction.

Assembly/Compiler Coding Rule 3. (M impact, M generality) *Use CMP/ADD/SUB instructions to compute branch conditions instead of INC/DEC instructions whenever possible.*

6.8.2.10 SIMD Floating-Point and X87 Instructions

In the Silvermont microarchitecture, only a subset of the SIMD FP execution units are implemented with a 128-bit wide data path. In Goldmont and later microarchitecture, SIMD FP units are implemented with a 128-bit data path. In general, packed SIMD instructions complete with one cycle less in latency and twice the throughput in the Goldmont and later microarchitecture, compared to the Silvermont microarchitecture.

In particular, MULPD latency is accelerated from 7 to 4 cycles, with 4-fold throughput from every 4 cycles to 1 per cycle.

Latency and throughput of X87 extended precision load and store, FLD m80fp, and FST/FSTP m80fp are also improved in the Goldmont and later microarchitecture. See Table 6-19 for more details.

In the Goldmont Plus microarchitecture, Floating point divider is upgraded to radix-1024 based design. Floating point divide and square root latency and bandwidth are significantly improved.

6.8.2.11 SIMD Integer Instructions

In the Silvermont microarchitecture, a relatively small subset of the SIMD integer instructions can execute with throughput of two instructions per cycle. In the Goldmont and later microarchitecture, many more SIMD integer instructions can complete at a rate of two instructions per cycle.

Latency and/or throughput improvements in the Goldmont and later microarchitecture include other SIMD integer instructions that execute only one port. For example, PMULLD has an eleven-cycle latency and throughput of one every eleven cycles in the Silvermont microarchitecture. It has five-cycle latency and throughput of one every two cycles in the Goldmont and later microarchitectures.

In general, SIMD integer multiply hardware is significantly faster (four-cycle latency) and higher throughput (one cycle throughput) than in the Silvermont microarchitecture. Additionally, PADDQ/PSUBQ has two-cycle latency and throughput of every two cycles, compared to four-cycle latency and throughput every four cycles in the Silvermont microarchitecture. PSHUFB has one-cycle latency and throughput in the Goldmont and later microarchitectures, compared to five-cycle latency and throughput of every five cycles. See Table 6-19 for more details.

6.8.2.12 Vectorization Considerations

In the Silvermont microarchitecture, opportunity for profitable vectorization may be limited by the availability of high-throughput implementation SIMD execution units or SIMD instructions that require MSROM to decode into longer flows.

The Goldmont and later microarchitectures allows compiler, as well as direct programming, to profit from vectorization due to improvement in latency and throughput across a wide variety of SIMD instructions.

Assembly/Compiler Coding Rule 4. (M impact, M generality) *Avoid MSROM instructions for code vectorization.*

6.8.2.13 Other SIMD Instructions

The Silvermont microarchitecture supports AESNI and PCLMULQDQ to accelerate performance of various cryptographic algorithms like AES and AES-GCM for block encryption/decryption.

In the Goldmont and later microarchitectures, the execution hardware is improved from execution latency, throughput to decode throughput. For example, PCLMULQDQ has latency of 6 cycles with throughput of every four cycles in the Goldmont microarchitecture, compared to ten-cycle latency and throughput of every ten cycles in the Silvermont microarchitecture.

Additionally, the Goldmont and later microarchitecture supports SHANI to accelerate the performance of secure hashing algorithms like SHA1 and SHA256. More details about the secure hashing algorithms and SHANI can be found at

https://software.intel.com/en-us/articles/intel-sha-extensions.

Examples and reference implementation of using the Intel SHA extensions can be found at:

https://software.intel.com/en-us/articles/intel-sha-extensions-implementations.

6.8.2.14 Instruction Selection

Table 6-14 summarizes the latency for floating-point and SIMD integer operations in the Silvermont microarchitecture. The throughput column is expressed in number of cycles per instruction that execution can complete with all available execution units employed (for example, 4 indicates the same instruction can complete execution every four cycles; 0.33 indicates three identical instructions can complete execution each cycle).

Table 6-14. Floating-Point and SIMD Integer Latency

	Goldmont Plus		Goldmont		Silvermont	
	Latency	Throughput	Latency	Throughput	Latency	Throughput
SIMD integer ALU						
128-bit ALU/logical/move	1	0.5	1	0.5	1	0.5
64-bit ALU/logical/move	1	0.5	1	0.5	1	0.5
SIMD integer shift						
128-bit	1	0.5	1	0.5	1	1
64-bit	1	0.5	1	0.5	1	1
SIMD shuffle						
128-bit	1	0.5	1	0.5	1	1
64-bit	1	0.5	1	0.5	1	1
SIMD integer multiplier						
128-bit	4	1	4	1	5	2
64-bit	4	1	4	1	4	1
FP Adder						
x87 (fadd)	3	1	3	1	3	1
scalar (addsd, addss)	3	1	3	1	3	1
packed (addpd, addps)	3	1	3	1	4	2
FP Multiplier						
x87 (fmul)	5	2	5	2	5	2
scalar single-precision (mulss)	4	1	4	1	4	1
scalar double-precision (mulsd)	4	1	4	1	5	2
packed single-precision (mulps)	4	1	4	1	5	2
packed double-precision (mulpd)	4	1	4	1	7	4
Converts						
CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ	4	1	4	1	5	2
CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI	4	1	4	1	4	1

Table 6-14. Floating-Point and SIMD Integer Latency (Contd.)

	Goldmont Plus		Goldmont		Silvermont	
	Latency	**Through put**	**Latency**	**Through put**	**Latency**	**Throughp ut**
FP Divider						
x87 fdiv (extended-precision)	15	11	39	39	39	39
x87 fdiv (double-precision)	14	10	34	34	34	34
x87 fdiv (single-precision)	11	7	19	19	19	19
scalar single-precision (divss)	11	7	19	18	19	17
scalar double-precision (divsd)	14	10	34	33	34	32
packed single-precision (divps)	16	12	36	35	39	39
packed double-precision (divpd)	22	18	66	65	69	69

Note that scalar SSE single precision multiples are one cycle faster than most FP operations. From inspection of the table you can also see that packed SSE doubles have a slightly larger latency and smaller throughput compared to their scalar counterparts.

Assembly/Compiler Coding Rule 5. (M impact, M generality) *Favor SSE floating-point instructions over x87 floating point instructions.*

Assembly/Compiler Coding Rule 6. (MH impact, M generality) *Run with exceptions masked and the DAZ and FTZ flags set (whenever possible).*

Tuning Suggestion 5. *Use the perfmon counters MACHINE_CLEARS.FP_ASSIST to see if floating exceptions are impacting program performance.*

6.8.2.15 Integer Division

In Silvermont microarchitecture, integer division requires microcode flows that are relatively long and slow. Its latency can vary profoundly on the input value and data sizes. In Goldmont and later microarchitecture, there is hardware enhancement for short-precision forms of DIV/IDIV without using MSROM. DIV/IDIV forms needing higher precision do use MSROM, but are also accelerated from the hardware enhancement. Table 6-15 and Table 6-16 show the latency range for divide instructions, and the instructions that require MSROM are noted with the superscript ‘u’.

Table 6-15. Unsigned Integer Division Operation Latency

	Dividend	**Divisor**	**Quotient**	**Remainder**	**Silvermont^u**	**Goldmont Plus/Goldmont**
DIV r8	AX	r8	AL	AH	25	11-12
DIV r16	DX:AX	r16	AX	DX	26-30	12-17^u
DIV r32	EDX:EAX	r32	EAX	EDX	26-38	12-25^u
DIV r64	RDX:RAX	r64	RAX	RDX	38-123	12-41^u

Table 6-16. Signed Integer Division Operation Latency

	Dividend	Divisor	Quotient	Remainder	Silvermont[u]	Goldmont Plus/Goldmont
IDIV r8	AX	r8	AL	AH	34	11-12
IDIV r16	DX:AX	r16	AX	DX	35-40	12-17[u]
IDIV r32	EDX:EAX	r32	EAX	EDX	35-47	12-25[u]
IDIV r64	RDX:RAX	r64	RAX	RDX	49-135	12-41[u]

User/Source Coding Rule 2. (M impact, L generality) *Use divides only when really needed and take care to use the correct data size and sign so that you get the most efficient execution.*

Tuning Suggestion 6. *Use the perfmon counter CYCLES_DIV_BUSY.ANY to see if the divides are a bottleneck in the program.*

If one needs unaligned groups of packed singles where the whole array is aligned, the use of PALIGNR is recommend over MOVUPS. For instance, load A[x+y+3:x+y] where x and y are loop variables; it is better to calculate x+y, round down to a multiple of 4 and use a MOVAPS and PALIGNR to get the elements (rather than a MOVUPS at x+y). While this may look longer, the integer operations can execute in parallel to FP ones. This will also avoid the periodic MOVUPS that splits a line at the cost of approximately 6 cycles.

User/Source Coding Rule 3. (M impact, M generality) *Use PALIGNR when stepping through packed single elements*

6.8.2.16 Integer Shift

When using an integer shift instruction with shift count in a register (i.e., CL), there is a one cycle bubble for scheduling if the count register is produced by the preceding instruction in the execution pipeline. Thus, the instruction producing the shift count should be hoisted whenever possible.

Additionally, double shift instructions (SHLD/SHRD) operating on 64-bit input data require long MSROM flows. In the Silvermont microarchitecture, SHRD with a 32-bit destination register and immediate shift count is decoded from the MSROM but the corresponding SHLD is not. In the Goldmont and later microarchitecture, SHLD/SHRD with 32-bit destination register and immediate shift count are not decoded from the MSROM. SHLD/SHRD with 32-bit destination memory operand or with CL shift count are decoded from the MSROM on both Silvermont and Goldmont.

F.8.2.17 Pause Instruction

In the Goldmont and later microarchitecture, the latency of the PAUSE instruction is similar to that of the Skylake microarchitecture to achieve better power saving with thread synchronization primitives.

6.8.3 Optimizing Memory Accesses

6.8.3.1 Reduce Unaligned Memory Access with PALIGNR

When working with single-precision FP or dword data arrays, loading 4 consecutive elements often encounter memory accesses that are not 16-Byte aligned. For example, a nested loop iteration with an array using two iterating indices, 'i', 'j' in A[i + j]. When loading 16 bytes from memory using "i+j" as the effective index that increments by 1 in an inner loop, unaligned access will occur 3 of 4 accesses.

These unaligned memory access can be avoided. Assuming the base of the array is 16-Bytes aligned, loading 16 bytes should be done with an effective index that is a multiple of 4, followed by PALIGNR with two consecutive 16-byte chunks already loaded in XMM, with the imm8 constant derived from 4* remainder of the original "i+j".

Assembly/Compiler Coding Rule 7. (M impact, M generality) *Use PALIGNR when stepping through packed single-precision FP or dword elements.*

6.8.3.2 Minimize Memory Execution Issues

In the Goldmont and later microarchitecture, fully out-of-order execution in the MEC allows loads to pass older stores which have not yet resolved their address. If the load did depend on the older store, the hardware detects this situation and the load and subsequent operations need to be re-executed. The programmer can use a performance counter event to assess and locate the cause of such re-execution.

In the Silvermont microarchitecture, its RehabQ needs to deal with several types of execution problems in the MEC. The issues include: load blocks, load/store splits, locks, TLB misses, unknown addresses, and too many stores. *The perfmon counter's REHABQ* in the Silvermont microarchitecture can be used to assess problems specific to the Silvermont microarchitecture.

Tuning Suggestion 7. *Use the perfmon counters MACHINE_CLEAR.DISAMBIGUATION to assess the impact of loads passing older unknown stores on application performance with the Goldmont microarchitecture and its descendants.*

6.8.3.3 Store Forwarding

Forwarding is significantly improved in the Silvermont and later microarchitectures compared to prior generations. A store instruction will forward its data to a receiving load instruction if the following are true:

- The forwarding store and the receiving load start at the same address.
- The receiving load is smaller than or equal to the forwarding store in terms of width.
- The forwarding store or the receiving load do not incur cache line splits.

Table 6-17 and Table 6-18 illustrate various situations of successful forwarding versus situations where preceding stores cannot be forwarded.

Table 6-17. Store Forwarding Conditions (1 and 2 Byte Stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	N														
	2	F	N														

Table 6-18. Store Forwarding Conditions (4-16 Byte Stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	N	F	N												
	2	F	N	F	N												
	4	F	N	N	N												

Table 6-18. Store Forwarding Conditions (4-16 Byte Stores)

		Load Alignment															
8	1	F	N	N	N	N	N	N	N								
	2	F	N	N	N	N	N	N	N								
	4	F	N	N	N	N	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	2	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	4	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	8	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

If one (or more) of these conditions is not satisfied, the load is blocked and put into the RehabQ to reissue again.

To eliminate/avoid store forwarding problems, use the guidelines below (in order of preference):

- Use registers instead of memory.
- Hoist the store as early as possible (stores happen later in the pipeline than loads, so the store needs to be hoisted many instructions earlier than the load).

The cost of a successful forwarding varies with microarchitectures. The cost is 3 cycles in the Silvermont microarchitecture (that is, if the store executes at cycle n, the load will execute at cycle n+3). The cost is 4 cycles in the Goldmont microarchitecture. Intel Goldmont Plus microarchitecture optimizes certain store data from register operation to reduce store to load forwarding latency to 3 cycle.

6.8.3.4 PrefetchW Instruction

The Silvermont and later microarchitectures support the PrefetchW instruction (0f 0d /1). This instruction is a hint to the hardware to prefetch the specified line into the cache with a read-for-ownership request. This can allow later stores to that line to complete faster than they would if the line was not prefetched or was prefetched with a different instruction. All prefetch instructions may cause performance loss if misused. Care should be used to ensure that prefetch instructions, including PrefetchW, actually improve performance. The instruction opcode 0f 0d /0 continues to be a NOP. It does not prefetch the indicated line.

6.8.3.5 Cache Line Splits and Alignment

Cache line splits cause load and store instructions to operate at reduced bandwidth. As a result, they should be avoided where possible.

Tuning Suggestion 8. *Use the REHABQ.ST_SPLIT and REHABQ.LD_SPLIT perfmon counters to locate splits, and to count the number of split operations.*

While aligned accesses are preferred, the Silvermont microarchitecture has hardware support for unaligned references. As such, MOVUPS/MOVUPD/MOVDQU instructions are all single UOP instructions in contrast to previous generation Intel Atom processors.

6.8.3.6 Segment Base

For simplicity, the AGU in the Silvermont microarchitecture assumes that the segment base will be zero. However, while studies have shown that this is overwhelmingly true, there are times when a non-zero segment base (NZB) must be used. When using NZBs, keep the segment base cache line (0x40) aligned if at all possible. NZB address generation involves a 1 cycle penalty in the Silvermont microarchitecture. In Goldmont and later microarchitecture, NZB address generation can maintain one per cycle.

6.8.3.7 Copy and String Copy

Compilers typically provide libraries with memcpy/memset routines that provide good performance while managing code size and alignment issues.

Memcpy and memset operation can be accomplished using REP MOVS/STOS instructions with length of operation decomposed for optimized byte/dword granular operations and alignment considerations. This usually provides a decent copy/set solution for the general case. The REP MOVS/STOS instructions have a fixed overhead. REP STOS should be able to cope with line splits for long strings; but REP MOVS cannot due to the complexity of the possible alignment matches between source and destination.

For specific copy/set needs, macro code sequence using SIMD instruction can provide modest gains (on the order of a dozen clocks or so), depending on the alignment, buffer length, and cache residency of the buffers. Large memory copies with cache line splits are a notable exception to this rule, where careful macrocode might avoid the cache lines splits and substantially improve on REP MOV.

Processors based on the Silvermont microarchitecture support the Enhanced REP MOVSB and STOSB operation feature. REP string operations using MOVSB and STOSB can provide the smallest code size with both flexible and high performance REP string operations for software in common situations like memory copy and set operations. Processors that provide enhanced MOVSB/STOSB operations are enumerated by the CPUID feature flag: CPUID:(EAX=7H, ECX=0H):EBX.[bit 9] = 1.

Software wishing to have a simple default string copy or store routine that will work well on a range of implementations (including future implementations) should consider using REP MOVSB or REP STOSB on implementations that support Enhanced REP MOVSB and STOSB. Although these instructions may not be as fast on a specific implementation as a more specialized copy/store routine, such specialized routines may not perform as well on future processors and may not take advantage of future enhancements. REP MOVSB and REP STOSB will continue to perform reasonably well on future processors.

6.9 INSTRUCTION LATENCY AND THROUGHPUT

This section lists the throughput and latency information of recent microarchitectures for Intel Atom processor generations. Instructions that require decoder assistance from MSROM are marked in the "Comment" column (instructions marked with 'Y' should be used minimally if more decode-efficient alternatives are available). Throughput and latency values for various instructions are grouped by the respective microarchitecture according to its CPUID DisplayFamily_DisplayModel. When a large number of DisplayModels of the same DisplayFamily have the same time timing characteristics, the DisplayFamily may be listed only once.

The microarchitectures and corresponding DisplayFamily_DisplayModel signature covered in this section are:

- Goldmont Plus microarchitecture: 06_7AH. Note that if Goldmont Plus microarchitecture differs from Goldmont in value, this will be indicated by the addition of "(GLP)" next to the value in the table below.
- Goldmont microarchitecture: 06_5CH, 06_5FH.
- Silvermont or Airmont microarchitecture: 06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
ADC/SBB r32, imm8	1	2	2	2	N	N
ADC/SBB r32, r32	1	2	2	2	N	N
ADC/SBB r64, r64	1	2	2	2	N	N

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
ADD/AND/CMP/OR/SUB/XOR/TEST r32, r32	0.33	0.5	1	1	N	N
ADD/AND/CMP/OR/SUB/XOR/TEST r64, r64	0.33	0.5	1	1	N	N
ADDPD/ADDSUBPD/MAXPD/MINPD/SUBPD xmm, xmm	1	2	3	4	N	N
ADDPS/ADDSD/ADDSS/ADDSUBPS/SUBPS/SUBSD/ SUBSS	1	1	3	3	N	N
MAXPS/MAXSD/MAXSS/MINPS/MINSD/MINSS xmm, xmm	1	1	3	3	N	N
ANDNPD/ANDNPS/ANDPD/ANDPS/ORPD/ORPS/XO RPD/XORPS	0.5	0.5	1	1	N	N
AESDEC/AESDECLAST/AESENC/AESENCLAST	2 1 (GLP)	5	6 4 (GLP)	8	N	Y
AESIMC/AESKEYGEN	2 1 (GLP)	5	5 4 (GLP)	8	N	Y
BLENDPD/BLENDPS xmm, xmm, imm8	0.5	1	1	1	N	N
BLENDVPD/BLENDVPS xmm, xmm	4	4	4	4	Y	Y
BSF/BSR r32, r32	8	10	10	10	Y	Y
BSWAP r32	1	1	1	1	N	N
BT/BTC/BTR/BTS r32, r32	1	1	1	1	N	N
CBW	4	4	4	4	Y	Y
CDQ/CLC/CMC	1	1	1	1	N	N
CMOVxx r32; r32	1	1	2	2	N	N
CMPPD xmm, xmm, imm	1	2	3	4	N	N
CMPSD/CMPPS/CMPSS xmm, xmm, imm	1	1	3	3	N	N
CMPXCHG r32, r32	5	6	5	6	Y	Y
CMPXCHG r64, r64	5	6	5	6	Y	Y
(U)COMISD/(U)COMISS xmm, xmm;	1	1	4	4	N	N
CPUID	58	60	58	60	Y	Y
CRC32 r32, r32	1	1	3	3	N	N
CRC32 r64, r64	1	1	3	3	N	N
CVTDQ2PD/CVTDQ2PS/CVTPD2DQ/CVTPD2PS xmm, xmm	1	2	4	5	N	N
CVT(T)PD2PI/CVT(T)PI2PD	1	2	4	5	N	N
CVT(T)PS2DQ/CVTPS2PD xmm, xmm;	1	2	4	5	N	N
CVT(T)SD2SS/CVTSS2SD xmm, xmm	1	1	4	4	N	N
CVTSI2SD/SS xmm, r32	1	1	7	6	N	N
CVTSD2SI/SS2SI r32, xmm	1	1	4	4	N	N
DEC/INC r32	1	1	1	1	N	N

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)

Instruction	**Throughput**		**Latency**		**MSROM**	
	06_5CH, 5FH, 7AH	**06_37H, 4AH,4CH ,4DH,5A H,5DH**	**06_5CH, 5FH, 7AH**	**06_37H, 4AH,4CH, 4DH,5AH, 5DH**	**06_5CH ,5FH, 7AH**	**06_37H, 4AH,4C H,4DH,5 AH,5DH**
DIV r8	11-12	25	11-12	25	N	Y
DIV r16	12-17	26-30	12-17	26-30	Y	Y
DIV r32	12-25	26-38	12-25	26-38	Y	Y
DIV r64	12-41	38-123	12-41	38-123	Y	Y
DIVPD[1]	12, 65 18 (GLP)	27-69	13, 66 22 (GLP)	27-69	N	Y
DIVPS[1]	12,35 12 (GLP)	27-39	13, 36 16 (GLP)	27-39	N	Y
DIVSD[1]	12,33 10 (GLP)	11-32	13,34 14 (GLP)	13-34	N	N
DIVSS[1]	12,18 7 (GLP)	11-17	13,19 11 (GLP)	13-19	N	N
DPPD xmm, xmm, imm	5	8	8	12	Y	Y
DPPS xmm, xmm, imm	11	12	14	15	Y	Y
EMMS	23	10	23	10	Y	Y
EXTRACTPS	1	4	4	5	N	Y
F2XM1	87	88	87	88	Y	Y
FABS/FCHS	0.5	1	1	1	N	N
FCOM	1	1	4	4	N	N
FADD/FSUB	1	1	3	3	N	N
FCOS	154	168	154	168	Y	Y
FDECSTP/FINCSTP	0.5	0.5	1	1	N	N
FDIV	39 11 (EP GLP)	39	39 15 (EP GLP)	39	N	N
FLDZ	280	277	280	277	Y	Y
FMUL	2	2	5	5	N	N
FPATAN/FYL2X/FYL2XP1	303	296	303	296	Y	Y
FPTAN/FSINCOS	287	281	287	281	Y	Y
FRNDINT	41	25	41	25	Y	Y
FSCALE	32	74	32	74	Y	Y
FSIN	140	150	140	150	Y	Y
FSQRT	40	40	40	40	N	N
HADDPD/HSUBPD xmm, xmm	5	5	5	6	Y	Y
HADDPS/HSUBPS xmm, xmm	6	6	6	6	Y	Y
IDIV r8	11-12	34	11-12	34	N	Y
IDIV r16	12-17	35-40	12-17	35-40	Y	Y

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
IDIV r32	12-25	35-47	12-25	35-47	Y	Y
IDIV r64	12-41	49-135	12-41	49-135	Y	Y
IMUL r32, r32 (single dest)	1	1	3	3	N	N
IMUL r32 (dual dest)	2	5	3 (4, EDX)	4	N	Y
IMUL r64, r64 (single dest)	2	2	5	5	N	N
IMUL r64 (dual dest)	2	4	5 (6,RDX)	5 (7,RDX)	N	Y
INSERTPS	0.5	1	1	1	N	N
MASKMOVDQU	4	5	4	5	Y	Y
MOVAPD/MOVAPS/MOVDQA/MOVDQU/MOVUPD/MOVUPS xmm, xmm;	0.33[2]/0.5	0.5	0/1	1	N	N
MOVD r32, xmm; MOVQ r64, xmm	1	1	4	4	N	N
MOVD xmm, r32 ; MOVQ xmm, r64	1	1	4	3	N	N
MOVDDUP/MOVHLPS/MOVLHPS/MOVSHDUP/MOVSLDUP	0.5	1	1	1	N	N
MOVDQ2Q/MOVQ/MOVQ2DQ	0.5	0.5	1	1	N	N
MOVSD/MOVSS xmm, xmm;	0.5	0.5	1	1	N	N
MPSADBW	4	5	5	7	Y	Y
MULPD	1	4	4	7	N	N
MULPS; MULSD	1	2	4	5	N	N
MULSS	1	1	4	4	N	N
NEG/NOT r32	0.33	0.5	1	1	N	N
PACKSSDW/WB xmm, xmm; PACKUSWB xmm, xmm	0.5	1	1	1	N	N
PABSB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDB/D/W xmm, xmm; PSUBB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PADDQ/PSUBQ/PCMPEQQ xmm, xmm	1	4	2	4	N	Y
PADDSB/W; PADDUSB/W; PSUBSB/W; PSUBUSB/W	0.5	0.5	1	1	N	N
PALIGNR xmm, xmm	0.5	1	1	1	N	N
PAND/PANDN/POR/PXOR xmm, xmm	0.5	0.5	1	1	N	N
PAVGB/W xmm, xmm	0.5	0.5	1	1	N	N
PBLENDW xmm, xmm, imm	0.5	0.5	1	1	N	N
PBLENDVB xmm, xmm	4	4	4	4	Y	Y
PCLMULQDQ xmm, xmm, imm	4	10	6	10	Y	Y
PCMPEQB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPESTRI xmm, xmm, imm	13	21	19(C)/26(F)[3]	21(C)/28(F)	Y	Y
PCMPESTRM xmm, xmm, imm	14	17	15(X)/25(F)[1]	17(X)/24(F)	Y	Y

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH ,4DH,5A H,5DH	06_5CH, 5FH, 7AH	06_37H, 4AH,4CH, 4DH,5AH, 5DH	06_5CH ,5FH, 7AH	06_37H, 4AH,4C H,4DH,5 AH,5DH
PCMPGTB/D/W xmm, xmm	0.5	0.5	1	1	N	N
PCMPGTQ/PHMINPOSUW xmm, xmm	2	2	5	5	N	N
PCMPISTRI xmm, xmm, imm	8	17	14(C)/ 21(F)[1]	17(C)/ 24(F)	Y	Y
PCMPISTRM xmm, xmm, imm	7	13	10(X)/ 20(F)[1]	13(X)/ 20(F)	Y	Y
PEXTRB/WD r32, xmm, imm	1	4	4	5	N	Y
PINSRB/WD xmm, r32, imm	1	1	4	3	N	N
PHADDD/PHSUBD xmm, xmm	4	6	4	6	Y	Y
PHADDW/PHADDSW xmm, xmm	6	9	6	9	Y	Y
PHSUBW/PHSUBSW xmm, xmm	6	9	6	9	Y	Y
PMADDUBSW/PMADDWD/PMULHRSW/PSADBW xmm, xmm	1	2	4	5	N	N
PMAXSB/W/D xmm, xmm; PMAXUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMINSB/W/D xmm, xmm; PMINUB/W/D xmm, xmm	0.5	0.5	1	1	N	N
PMOVMSKB r32, xmm	1	1	4	4	N	N
PMOVSXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMOVZXBW/BD/BQ/WD/WQ/DQ xmm, xmm	0.5	1	1	1	N	N
PMULDQ/PMULUDQ xmm, xmm	1	2	4	5	N	N
PMULHUW/PMULHW/PMULLW xmm, xmm	1	2	4	5	N	N
PMULLD xmm, xmm	2	11	5	11	N	Y
POPCNT r32, r32	1	1	3	3	N	N
POPCNT r64, r64	1	1	3	3	N	N
PSHUFB xmm, xmm	1	5	1	5	N	Y
PSHUFD xmm, mem, imm	0.5	1	1	1	N	N
PSHUFHW; PSHUFLW; PSHUFW	0.5	1	1	1	N	N
PSIGNB/D/W xmm, xmm	0.5	1	1	1	N	N
PSLLDQ/PSRLDQ xmm, imm; SHUFPD/SHUFPS	0.5	1	1	1	N	N
PSLLD/Q/W xmm, xmm	1	2	2	2	N	N
PSRAD/W xmm, imm;	0.5	1	1	1	N	N
PSRAD/W xmm, xmm;	1	2	2	2	N	N
PSRLD/Q/W xmm, imm;	0.5	1	1	1	N	N
PSRLD/Q/W xmm, xmm	1	2	2	2	N	N
PTEST xmm, xmm	1	1	4	4	N	N
PUNPCKHBW/DQ/WD; PUNPCKLBW/DQ/WD	0.5	1	1	1	N	N
PUNPCKHQDQ; PUNPCKLQDQ	0.5	1	1	1	N	N

Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)

Instruction	Throughput		Latency		MSROM	
	06_5CH, 5FH, 7AH	**06_37H, 4AH,4CH ,4DH,5A H,5DH**	**06_5CH, 5FH, 7AH**	**06_37H, 4AH,4CH, 4DH,5AH, 5DH**	**06_5CH ,5FH, 7AH**	**06_37H, 4AH,4C H,4DH,5 AH,5DH**
RCPPS/RSQRTPS	6	8	9	9	Y	Y
RCPSS/RSQRTSS	1	1	4	4	N	N
RDTSC	20	30	20	30	Y	Y
ROUNDPD/PS	1	2	4	5	N	N
ROUNDSD/SS	1	1	4	4	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in CL)	1	1	1 (2 for CL source)	1 (2 for CL source)	N	N
ROL; ROR; SAL; SAR; SHL; SHR (count in imm8)	1	1	1	1	N	N
SAHF	1	1	1	1	N	N
SHLD r32, r32, imm	2	2	2	2	N	N
SHRD r32, r32, imm	2	4	2	4	N	Y
SHLD/SHRD r64, r64, imm	12	10	12	10	Y	Y
SHLD/SHRD r64, r64, CL	14	10	14	10	Y	Y
SHLD/SHRD r32, r32, CL	4	4	4	4	Y	Y
SHUFPD/SHUFPS xmm, xmm, imm	0.5	1	1	1	N	N
SQRTPD	67 26 (GLP)	70	68 30 (GLP)	71	N	Y
SQRTPS	37 14 (GLP)	40	38 18 (GLP)	41	N	Y
SQRTSD	34 14 (GLP)	35	35 18 (GLP)	35	N	Y
SQRTSS	19 8 (GLP)	20	20 12 (GLP)	20	N	Y
TEST r32, r32	0.33	0.5	1	1	N	N
UNPCKHPD; UNPCKHPS; UNPCKLPD, UNPCKLPS	0.5	1	1	1	N	N
XADD r32, r32	2	5	4	5	Y	Y
XCHG r32, r32	2	5	4	5	Y	Y
XCHG r64, r64	2	5	4	5	Y	Y
SHA1MSG1/SHA1MSG2/SHA1NEXTE	1	NA	3	NA	N	NA
SHA1RNDS4 xmm, xmm, imm	2	NA	5	NA	N	NA
SHA256MSG1/SHA256MSG2	1	NA	3	NA	N	NA
SHA256RNDS2	4	NA	7	NA	N	NA

NOTES:

1. DIVPD/DIVPS/DIVSD/DIVSS list early-exit value first and common-case value second. Early-exit case applies to a special input value such as QNAN. Common case applies to normal numeric values.
2. Throughput is 0.33 cycles if move elimination is effect, otherwise 0.5 cycle.
3. Latency values are for ECX/EFLAGS/XMM0 dependency: (C/F/X)

7. Updates to Appendix D

Change bars and **violet** text show changes to Appendix D of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: Instruction Latency and Throughput.

Changes to this chapter:

- This chapter has been updated to be Volume 2, Chapter 7: Instruction Latency and Throughput.
- Updated capitalization of headings throughout chapter.
- Updated branding throughout chapter.
- Typo and punctuation corrections as necessary.

CHAPTER 7
INSTRUCTION LATENCY AND THROUGHPUT

NOTE

All recent processors have latency and throughput information posted on the Intel® 64 and IA-32 Architectures Software Developer Manuals Page.

This appendix contains tables showing the latency and throughput are associated with commonly used instructions[1]. The instruction timing data varies across processors family/models. It contains the following sections:

- **Chapter 7.1, "Overview"** — Provides an overview of issues related to instruction selection and scheduling.
- **Chapter 7.2, "Definitions"** — Presents definitions.
- **Chapter 7.3, "Latency and Throughput"** — Lists instruction throughput, latency associated with commonly-used instructions.

7.1 OVERVIEW

This appendix provides information to assembly language programmers and compiler writers. The information aids in the selection of instruction sequences (to minimize chain latency) and in the arrangement of instructions (assists in hardware processing). The performance impact of applying the information has been shown to be on the order of several percent. This is for applications not dominated by other performance factors, such as:

- Cache miss latencies.
- Bus bandwidth.
- I/O bandwidth.

Instruction selection and scheduling matters when the programmer has already addressed the performance issues discussed in Chapter 2:

- Observe store forwarding restrictions.
- Avoid cache line and memory order buffer splits.
- Do not inhibit branch prediction.
- Minimize the use of xchg instructions on memory locations.

While several items on the above list involve selecting the right instruction, this appendix focuses on the following issues. These are listed in priority order, though which item contributes most to performance varies by application:

- Maximize the flow of μops into the execution core. Instructions which consist of more than four μops require additional steps from microcode ROM. Instructions with longer micro-op flows incur a delay in the front end and reduce the supply of micro-ops to the execution core.

  In Pentium 4 and Intel Xeon processors, transfers to microcode ROM often reduce how efficiently μops can be packed into the trace cache. Where possible, it is advisable to select instructions with four or fewer μops. For example, a 32-bit integer multiply with a memory operand fits in the trace cache without going to microcode, while a 16-bit integer multiply to memory does not.

1. Although instruction latency may be useful in some limited situations (e.g., a tight loop with a dependency chain that exposes instruction latency), software optimization on super-scalar, out-of-order microarchitecture, in general, will benefit much more on increasing the effective throughput of the larger-scale code path. Coding techniques that rely on instruction latency alone to influence the scheduling of instruction is likely to be sub-optimal as such coding technique is likely to interfere with the out-of-order machine or restrict the amount of instruction-level parallelism.

- Avoid resource conflicts. Interleaving instructions so that they don't compete for the same port or execution unit can increase throughput. For example, alternate PADDQ and PMULUDQ (each has a throughput of one issue per two clock cycles). When interleaved, they can achieve an effective throughput of one instruction per cycle because they use the same port but different execution units. Selecting instructions with fast throughput also helps to preserve issue port bandwidth, hide latency and allows for higher software performance.
- Minimize the latency of dependency chains that are on the critical path. For example, an operation to shift left by two bits executes faster when encoded as two adds than when it is encoded as a shift. If latency is not an issue, the shift results in a denser byte encoding.

In addition to the general and specific rules, coding guidelines and the instruction data provided in this manual, you can take advantage of the software performance analysis and tuning toolset available at http://developer.intel.com/software/products/index.htm. The tools include the Intel VTune Performance Analyzer, with its performance-monitoring capabilities.

7.2 DEFINITIONS

The data is listed in several tables. The tables contain the following:

- **Instruction Name** — The assembly mnemonic of each instruction.
- **Latency** — The number of clock cycles that are required for the execution core to complete the execution of all of the μops that form an instruction.
- **Throughput** — The number of clock cycles required to wait before the issue ports are free to accept the same instruction again. For many instructions, the throughput of an instruction can be significantly less than its latency.
- The case of RDRAND instruction latency and throughput is an exception to the definitions above, because the hardware facility that executes the RDRAND instruction resides in the uncore and is shared by all processor cores and logical processors in a physical package. The software observable latency and throughput using the sequence of "rdrand followby jnc" in a single-thread scenario can be as low as ~100 cycles. In third generation Intel Core processors based on Ivy Bridge microarchitecture, the total bandwidth to deliver random numbers via RDRAND by the uncore is about 500 MBytes/sec. Within the same processor core microarchitecture and different uncore implementations, RDRAND latency/throughput can vary across Intel Core and Intel Xeon processors.

7.3 LATENCY AND THROUGHPUT

This section presents the latency and throughput information for commonly-used instructions including: MMX technology, Streaming SIMD Extensions, subsequent generations of SIMD instruction extensions, and most of the frequently used general-purpose integer and x87 floating-point instructions.

Due to the complexity of dynamic execution and out-of-order nature of the execution core, the instruction latency data may not be sufficient to accurately predict realistic performance of actual code sequences based on adding instruction latency data.

- Instruction latency data is useful when tuning a dependency chain. However, dependency chains limit the out-of-order core's ability to execute micro-ops in parallel. Instruction throughput data are useful when tuning parallel code unencumbered by dependency chains.
- Numeric data in the tables is:
  - — Approximate and subject to change in future implementations of the microarchitecture.
  - — Not meant to be used as reference for instruction-level performance benchmarks. Comparison of instruction-level performance of microprocessors that are based on different microarchitectures is a complex subject and requires information that is beyond the scope of this manual.

Comparisons of latency and throughput data between different microarchitectures can be misleading.

Chapter 7.3.1 provides latency and throughput data for the register-to-register instruction type.

Chapter 7.3.3 discusses how to adjust latency and throughput specifications for the register-to-memory and memory-to-register instructions.

In some cases, the latency or throughput figures given are just one half of a clock. This occurs only for the double-speed ALUs.

7.3.1 Latency and Throughput with Register Operands

Instruction latency and throughput data are presented in Table 7-4 through Table 7-18. Tables include AESNI, SSE4.2, SSE4.1, Supplemental Streaming SIMD Extension 3, Streaming SIMD Extension 3, Streaming SIMD Extension 2, Streaming SIMD Extension, MMX technology and most common Intel 64 and IA-32 instructions. Instruction latency and throughput for different processor microarchitectures are in separate columns.

Processor instruction timing data is implementation specific; it can vary between model encodings within the same family encoding (e.g. model = 3 vs model < 2). Separate sets of instruction latency and throughput are shown in the columns for CPUID signature 0xF2n and 0xF3n. The column represented by 0xF3n also applies to Intel processors with CPUID signature 0xF4n and 0xF6n. The notation 0xF2n represents the hex value of the lower 12 bits of the EAX register reported by CPUID instruction with input value of EAX = 1; 'F' indicates the family encoding value is 15, '2' indicates the model encoding is 2, 'n' indicates it applies to any value in the stepping encoding.

Intel Core Solo and Intel Core Duo processors are represented by 06_0EH. Processors bases on 65 nm Intel Core microarchitecture are represented by 06_0FH. Processors based on Enhanced Intel Core microarchitecture are represented by 06_17H and 06_1DH. CPUID family/Model signatures of processors based on Nehalem microarchitecture are represented by 06_1AH, 06_1EH, 06_1FH, and 06_2EH. Processors based on Westmere microarchitecture are represented by 06_25H, 06_2CH and 06_2FH. Processors based on Sandy Bridge microarchitecture are represented by 06_2AH, 06_2DH. Processors based on Ivy Bridge microarchitecture are represented by 06_3AH, 06_3EH. Processors based on Haswell microarchitecture are represented by 06_3CH, 06_45H and 06_46H.

Table 7-1. CPUID Signature Values of Of Recent Intel Microarchitectures

DisplayFamily_DisplayModel	Recent Intel Microarchitectures
06_4EH, 06_5EH	Skylake microarchitecture
06_3DH, 06_47H, 06_56H	Broadwell microarchitecture
06_3CH, 06_45H, 06_46H, 06_3FH	Haswell microarchitecture
06_3AH, 06_3EH	Ivy Bridge microarchitecture
06_2AH, 06_2DH	Sandy Bridge microarchitecture
06_25H, 06_2CH, 06_2FH	Intel microarchitecture Westmere
06_1AH, 06_1EH, 06_1FH, 06_2EH	Intel microarchitecture Nehalem
06_17H, 06_1DH	Enhanced Intel Core microarchitecture
06_0FH	Intel Core microarchitecture

Instruction latency varies by microarchitectures. Table 7-2 lists SIMD extensions introduction in recent microarchitectures. Each microarchitecture may be associated with more than one signature value given by the CPUID's "display_family" and "display_model". Not all instruction set extensions are enabled in all processors associated with a particular family/model designation. To determine whether a given instruction set extension is supported, software must use the appropriate CPUID feature flag as described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

.

Table 7-2. Instruction Extensions Introduction by Microarchitectures (CPUID Signature)

SIMD Instruction Extensions	DisplayFamily_DisplayModel							
	06_4EH, 06_5EH	06_3DH, 06_47H, 06_56H	06_3CH, 06_45H, 06_46H, 06_3FH	06_3AH, 06_3EH	06_2AH, 06_2DH	06_25H, 06_2CH, 06_2FH	06_1AH, 06_1EH, 06_1FH, 06_2EH	06_17H, 06_1DH
CLFLUSHOPT	Yes	No	No	No	No	No	No	No
ADX, RDSEED	Yes	Yes	No	No	No	No	No	No
AVX2, FMA, BMI1, BMI2	Yes	Yes	Yes	No	No	No	No	No
F16C, RDRAND, RWFSGSBASE	Yes	Yes	Yes	Yes	No	No	No	No
AVX	Yes	Yes	Yes	Yes	Yes	No	No	No
AESNI, PCLMULQDQ	Yes	Yes	Yes	Yes	Yes	Yes	No	No
SSE4.2, POPCNT	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
SSE4.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSSE3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SSE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MMX	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 7-3. BMI1, BMI2 and General Purpose Instructions

Instruction	Latency [1]		Throughput	
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_4E, 06_5E	06_3D, 06_47, 06_56
ADCX	1	1	1	1
ADOX	1	1	1	1
RESEED	Similar to RDRAND	Similar to RDRAND	Similar to RDRAND	Similar to RDRAND

Table 7-4. 256-bit Intel® AVX2 Instructions

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
VEXTRACTI128 xmm1, ymm2, imm	1	1	1	1	1	1
VMPSADBW	4	6	6	2	2	2
VPACKUSDW/SSWB	1	1	1	1	1	1
VPADDB/D/W/Q	1	1	1	0.33	0.5	0.5
VPADDSB	1	1	1	0.5	0.5	0.5
VPADDUSB	1	1	1	0.5	0.5	0.5
VPALIGNR	1	1	1	1	1	1
VPAVGB	1	1	1	0.5	0.5	0.5
VPBLENDD	1	1	1	0.33	0.33	0.33
VPBLENDW	1	1	1	1	1	1
VPBLENDVB	1	2	2	1	2	2
VPBROADCASTB/D/SS/SD	3	3	3	1	1	1
VPCMPEQB/W/D	1	1	1	0.5	0.5	0.5
VPCMPEQQ	1	1	1	0.5	0.5	0.5
VPCMPGTQ	3	5	5	1	1	1
VPHADDW/D/SW	3	3	3	2	2	2
VINSERTI128 ymm1, ymm2, xmm, imm	3	3	3	1	1	1
VPMADDWD	5[b]	5	5	0.5	1	1
VPMADDUBSW	5[b]	5	5	0.5	1	1
VPMAXSD	1	1	1	0.5	0.5	0.5
VPMAXUD	1	1	1	0.5	0.5	0.5
VPMOVSX	3	3	3	1	1	1
VPMOVZX	3	3	3	1	1	1
VPMULDQ/UDQ	5[b]	5	5	0.5	1	1
VPMULHRSW	5[b]	5	5	0.5	1	1
VPMULHW/LW	5[b]	5	5	0.5	1	1
VPMULLD	10[b]	10	10	1	2	2
VPOR/VPXOR	1	1	1	0.33	0.33	0.33
VPSADBW	3	5	5	1	1	1
VPSHUFB	1	1	1	1	1	1
VPSHUFD	1	1	1	1	1	1
VPSHUFLW/HW	1	1	1	1	1	1
VPSIGNB/D/W/Q	1	1	1	0.5	0.5	0.5
VPERMD/PS	3	3	3	1	1	1
VPSLLVD/Q	2	2	2	0.5	2	2

Table 7-4. 256-bit Intel® AVX2 Instructions (Contd.)

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**	**06_4E, 06_5E**	**06_3D, 06_47, 06_56**	**06_3C, 06_45, 06_46, 06_3F**
VPSRAVD	2	2	2	0.5	2	2
VPSRAD/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLDQ ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm1, ymm2, imm8	1	1	1	1	1	1
VPSLLQ/D/W ymm, ymm, ymm	4	4	4	1	1	1
VPUNPCKHBW/WD/DQ/QDQ	1	1	1	1	1	1
VPUNPCKLBW/WD/DQ/QDQ	1	1	1	1	1	1
ALL VFMA	4	5	5	0.5	0.5	0.5
VPMASKMOVD/Q mem, ymm[d], ymm				1	2	2
VPMASKMOVD/Q NUL, msk_0, ymm				>200[e]	2	2
VPMASKMOVD/Q ymm, ymm[d], mem	11	8	8	1	2	2
VPMASKMOVD/Q ymm, msk_0, [base+index][f]	>200	~200	~200	>200	~200	~200

b: includes 1-cycle bubble due to bypass.
c: includes two 1-cycle bubbles due to bypass
d: MASKMOV instruction timing measured with L1 reference and mask register selecting at least 1 or more elements.
e: MASKMOV store instruction with a mask value selecting 0 elements and illegal address (NUL or non-NUL) incurs delay due to assist.
f: MASKMOV Load instruction with a mask value selecting 0 elements and certain addressing forms incur delay due to assist.

Table 7-5. Gather Timing Data from L1D*

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F
VPGATHERDD/PS xmm, [vi128], xmm	~20	~17	~14	~4	~5	~7
VPGATHERQQ/PD xmm, [vi128], xmm	~18	~15	~12	~3	~4	~5
VPGATHERDD/PS ymm, [vi256], ymm	~22	~19	~20	~5	~6	~10
VPGATHERQQ/PD ymm, [vi256], ymm	~20	~16	~15	~4	~5	~7

* Gather Instructions fetch data elements via memory references. The timing data shown applies to memory references that reside within the L1 data cache and all mask elements selected

Table 7-6. BMI1, BMI2 and General Purpose Instructions

Instruction	Latency [1]			Throughput		
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F
ANDN	1	1	1	0.5	0.5	0.5
BEXTR	2	2	2	0.5	0.5	0.5
BLSI/BLSMSK/BLSR	1	1	1	0.5	0.5	0.5
BZHI	1	1	1	0.5	0.5	0.5
MULX r64, r64, r64	4	4	4	1	1	1
PDEP/PEXT r64, r64, r64	3	3	3	1	1	1
RORX r64, r64, r64	1	1	1	0.5	0.5	0.5
SALX/SARX/SHLX r64, r64, r64	1	1	1	0.5	0.5	0.5
LZCNT/TZCNT	3	3	3	1	1	1

Table 7-7. F16C,RDRAND Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_3A/3E	06_4E, 06_5E	06_3D, 06_47, 06_56	06_3C/45/46/3F	06_3A/3E
RDRAND* r64	Varies	Varies	Varies	<200	<300	~250	~250	<200
VCVTPH2PS ymm1, xmm2	7	6	6	7	1	1	1	1
VCVTPH2PS xmm1, xmm2	5	4	4	6	1	1	1	1
VCVTPS2PH ymm1, xmm2, imm	7	6	6	10	1	1	1	1
VCVTPS2PH xmm1, xmm2, imm	5	4	4	9	1	1	1	1

* See Section 7.2

Table 7-8. 256-bit Intel® AVX Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E
VADDPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VADDSUBPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VANDNPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VANDPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VBLENDPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	0.33	0.33	0.33	0.5
VBLENDVPD/PS ymm1, ymm2, ymm3, ymm	1	2	2	1	1	2	2	1
VCMPPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1

Table 7-8. 256-bit Intel® AVX Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VCVTDQ2PD ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTDQ2PS ymm1, ymm2	4	3	3	3	0.5	1	1	1
VCVT(T)PD2DQ ymm1, ymm2	7	6	6	4	1	1	1	1
VCVTPD2PS ymm1, ymm2	7	6	6	4	1	1	1	1
VCVT(T)PS2DQ ymm1, ymm2	4	3	3	3	1	1	1	1
VCVTPS2PD ymm1, xmm2	7	4	4	2	1	1	1	1
VDIVPD ymm1, ymm2, ymm3	14	16-23	25-35	27-35	8	16	27	28
VDIVPS ymm1, ymm2, ymm3	11	13-17	17-21	18-21	5	10	13	14
VDPPS ymm1, ymm2, ymm3	13	12	14	12	1.5	2	2	2
VEXTRACTF128 xmm1, ymm2, imm	3	3	3	3	1	1	1	1
VINSERTF128 ymm1, xmm2, imm	3	3	3	3	1	1	1	1
VMAXPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMINPD/PS ymm1, ymm2, ymm3	4	3	3	3	0.5	1	1	1
VMOVAPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMOVDDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVDQA/U ymm1, ymm2	1	1	1	1	0.25	0.25	0.25	0.5
VMOVMSKPD/PS ymm1, ymm2	2	2	2	1	1	1	1	1
VMOVQ xmm1, xmm2	1	1	1	1	0.33	0.33	0.33	0.33
VMOVD/Q xmm1, r32/r64	2	1	1	1	1	1	1	1
VMOVD/Q r32/r64, xmm	2	1	1	1	1	1	1	1
VMOVNTDQ/PS/PD					1	1	1	1
VMOVSHDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVSLDUP ymm1, ymm2	1	1	1	1	1	1	1	1
VMOVUPD/PS ymm1, ymm2	1	1	1	1	0.25	0.5	0.5	1
VMULPD/PS ymm1, ymm2, ymm3	4	3	5	5	0.5	0.5	0.5	1
VORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VPERM2F128 ymm1, ymm2, ymm3, imm	3	3	3	2	1	1	1	1
VPERMILPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VRCPPS ymm1, ymm2	4	7	7	7	1	2	2	2
VROUNDPD/PS ymm1, ymm2, imm	8	6	6	3	1	2	2	1
VRSQRTPS ymm1, ymm2	4	7	7	7	1	2	2	2
VSHUFPD/PS ymm1, ymm2, ymm3, imm	1	1	1	1	1	1	1	1
VSQRTPD ymm1, ymm2	<18	19-35	19-35	19-35	<12	16-27	16-27	28
VSQRTPS ymm1, ymm2	12	18-21	18-21	18-21	<6	13	13	14
VSUBPD/PS ymm1, ymm2, imm	4	3	3	3	0.5	1	1	1

Table 7-8. 256-bit Intel® AVX Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
VTESTPS ymm1, ymm2	3	2	2	2	1	1	1	1
VUNPCKHPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VUNPCKLPD/PS ymm1, ymm2, ymm3	1	1	1	1	1	1	1	1
VXORPD/PS ymm1, ymm2, ymm3	1	1	1	1	0.33	1	1	1
VZEROUPPER	0	0	0	0	1	1	1	1
VZEROALL					12	8	8	9
VEXTRACTPS reg, xmm2, imm	3	2	2	2	1	1	1	1
VINSERTPS xmm1, xmm2, reg, imm	1	1	1	1	1	1	1	1
VMASKMOVPD/PS mem[a], ymm, ymm					1	2	2	2
VMASKMOVPD/PS NUL, msk_0, ymm					>200[b]	2	2	2
VMASKMOVPD/PS ymm, ymm[a], mem	11	8	8	9	1	2	2	2
VMASKMOVPD/PS ymm, msk_0, [base+index][c]	>200	~200	~200	~200	>200	~200	~200	~200

Latency and Throughput data for CPUID signature 06_3AH are generally the same as those of 06_2AH, only those that differ from 06_2AH are shown in the 06_3AH column.
a: MASKMOV instruction timing measured with L1 reference and mask register selecting at least 1 or more elements.
b: MASKMOV store instruction with a mask value selecting 0 elements and illegal address (NUL or non-NUL) incurs delay due to assist.
c: MASKMOV Load instruction with a mask value selecting 0 elements and certain addressing forms incur delay due to assist.

Latency of VEX.128 encoded AVX instructions should refer to corresponding legacy 128-bit instructions.

Table 7-9. AESNI and PCLMULQDQ Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E	06_4E, 06_5E	06_3D/ 47/56	06_3C/4 5/46/3F	06_3A /3E
AESDEC/AESDECLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESENC/AESENCLAST xmm1, xmm2	4	7	7	8	1	1	1	1
AESIMC xmm1, xmm2	8	14	14	14	2	2	2	2
AESKEYGENASSIST xmm1, xmm2, imm	12	10	10	10	12	8	8	8
PCLMULQDQ xmm1, xmm2, imm	7[b]	5	7	14	1	1	2	8

b: includes 1-cycle bubble due to bypass.

Table 7-10. Intel® SSE4.2 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C /45/46 /3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
CRC32 r32, r32	3	3	3	3	1	1	1	1
PCMPESTRI xmm1, xmm2, imm	15	10	10	11	5	4	4	4
PCMPESTRM xmm1, xmm2, imm	10	10	10	11	6	5	5	4
PCMPISTRI xmm1, xmm2, imm	15	10	10	11	3	3	3	3
PCMPISTRM xmm1, xmm2, imm	15	11	11	11	3	3	3	3
PCMPGTQ xmm1, xmm2	3	5	5	5	0.33	1	1	1
POPCNT r32, r32	3	3	3	3	1	1	1	1
POPCNT r64, r64	3	3	3	3	1	1	1	1

Table 7-11. Intel® SSE4.1 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
BLENDPD/S xmm1, xmm2, imm	1	1	1	1	0.33	0.33	0.33	0.5
BLENDVPD/S xmm1, xmm2	1	2	2	2	1	2	2	1
DPPD xmm1, xmm2	9	7	9	9	1	1	1	1
DPPS xmm1, xmm2	13	12	14	13	2	2	2	2
EXTRACTPS xmm1, xmm2, imm	3	2	2	2	1	1	1	1
INSERTPS xmm1, xmm2, imm	1	1	1	1	1	1	1	1
MPSADBW xmm1, xmm2, imm	4	6	6	6	2	2	2	1
PACKUSDW xmm1, xmm2	1	1	1	1	1	1	1	0.5
PBLENVB xmm1, xmm2	2	2	2	2	2	2	2	1
PBLENDW xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PCMPEQQ xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRB/W/D reg, xmm1, imm	3	3	3	3	1	1	1	1
PHMINPOSUW xmm1,xmm2	4	5	5	5	1	1	1	1
PINSRB/W/D xmm1,reg, imm	2	2	2	2	1	1	1	1
PMAXSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMAXUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINSB/SD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMINUW/UD xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PMOVSXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVSXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5

Table 7-11. Intel® SSE4.1 Instructions (Contd.)

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D /47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A /3E/2A /2D
PMOVZXBD/BW/BQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMOVZXWD/WQ/DQ xmm1, xmm2	1	1	1	1	1	1	1	0.5
PMULDQ xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULLD xmm1, xmm2	10[c]	10	10	5	2	2	2	1
PTEST xmm1, xmm2	3	2	2	2	1	1	1	1
ROUNDPD/PS xmm1, xmm2, imm	6	6	6	3	2	2	2	1
ROUNDSD/SS xmm1, xmm2, imm	6	6	6	3	2	2	2	1

b: includes 1-cycle bubble due to bypass
c: includes two 1-cycle bubbles due to bypass

Table 7-12. Intel® SSE3 Instructions

Instruction	Latency [1]				Throughput			
DisplayFamily_DisplayModel	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A/ 3E/2A/ 2D	06_4E, 06_5E	06_3D/ 47/56	06_3C/ 45/46/ 3F	06_3A/ 3E/2A/ 2D
PALIGNR xmm1, xmm2, imm	1	1	1	1	1	1	1	0.5
PHADDD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHADDSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBD xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PHSUBSW xmm1, xmm2	3	3	3	3	2	2	2	1.5
PMADDUBSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PMULHRSW xmm1, xmm2	5[b]	5	5	5	0.5	1	1	1
PSHUFB xmm1, xmm2	1	1	1	1	1	1	1	0.5
PSIGNB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5
PABSB/D/W xmm1, xmm2	1	1	1	1	0.5	0.5	0.5	0.5

b: includes 1-cycle bubble due to bypass

Table 7-13. Intel® SSE3 SIMD Floating-point Instructions

Instruction	Latency[1]				Throughput			
DisplayFamily_DisplayModel	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
ADDSUBPD/ADDSUBPS	4	3	3	3	0.5	1	1	1
HADDPD xmm1, xmm2	6	5	5	5	2	2	2	2
HADDPS xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPD xmm1, xmm2	6	5	5	5	2	2	2	2
HSUBPS xmm1, xmm2	6	5	5	5	2	2	2	2
MOVDDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSHDUP xmm1, xmm2	1	1	1	1	1	1	1	1
MOVSLDUP xmm1, xmm2	1	1	1	1	1	1	1	1

Table 1-14. Intel® SIM SSE2 128-bit Integer Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
CVTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
CVTTPS2DQ xmm, xmm	3	3	3	3	1	1	1	1
MASKMOVDQU xmm, xmm					7	6	6	6
MOVD xmm, r64/r32	2	1	1	1	1	1	1	1
MOVD r64/r32, xmm	2	1	1	1	1	1	1	1
MOVDQA xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVDQU xmm, xmm	1	1	1	1	0.25	0.33	0.33	0.5
MOVQ xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PACKSSWB/PACKSSDW/ PACKUSWB xmm, xmm	1	1	1	1	1	1	1	0.5
PADDB/PADDW/PADDD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PADDSB/PADDSW/ PADDUSB/PADDUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PADDQ/ PSUBQ[3] xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PAND xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PANDN xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PAVGB/PAVGW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPEQB/PCMPEQD/ PCMPEQW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PCMPGTB/PCMPGTD/PCMPGTW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PEXTRW r32, xmm, imm8	3	3	3	3	1	1	1	1
PINSRW xmm, r32, imm8	2	2	2	2	2	2	2	1

Table 1-14. Intel® SIM SSE2 128-bit Integer Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A/3E/2A/2D
PMADDWD xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMAX xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMIN xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PMOVMSKB[3] r32, xmm	2	2	2	2	1	1	1	1
PMULHUW/PMULHW/ PMULLW xmm, xmm	5[b]	5	5	5	0.5	1	1	1
PMULUDQ xmm, xmm	5[b]	5	5	5	0.5	1	1	1
POR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33
PSADBW xmm, xmm	3	5	5	5	1	1	1	1
PSHUFD xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFHW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSHUFLW xmm, xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSLLW/PSLLD/PSLLQ xmm, imm8	1	1	1	1	1	1	1	1
PSLL/PSRL xmm, xmm	2	2	2	2	1	1	1	1
PSRAW/PSRAD xmm, imm8	1	1	1	1	1	1	1	1
PSRAW/PSRAD xmm, xmm	2	2	2	2	1	1	1	1
PSRLDQ xmm, imm8	1	1	1	1	1	1	1	0.5
PSRLW/PSRLD/PSRLQ xmm, imm8	1	1	1	1	1	1	1	1
PSUBB/PSUBW/PSUBD xmm, xmm	1	1	1	1	0.33	0.5	0.5	0.5
PSUBSB/PSUBSW/PSUBUSB /PSUBUSW xmm, xmm	1	1	1	1	0.5	0.5	0.5	0.5
PUNPCKHBW/PUNPCKHWD/ PUNPCKHDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKHQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLBW/PUNPCKLWD/ PUNPCKLDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PUNPCKLQDQ xmm, xmm	1	1	1	1	1	1	1	0.5
PXOR xmm, xmm	1	1	1	1	0.33	0.33	0.33	0.33

b: includes 1-cycle bubble due to bypass

Table 7-15. Intel® SSE2 Double-Precision Floating-Point Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
ADDPD xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSD xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPD xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPD xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
CMPSD xmm, xmm, imm8	4	3	3	3	0.5	1	1	1
COMISD xmm, xmm	2	2	2	2	1	1	1	1
CVTDQ2PD xmm, xmm	5	4	4	4	1	1	1	1
CVTDQ2PS xmm, xmm	4	3	3	3	1	1	1	1
CVTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTPD2PS xmm, xmm	5	4	4	4	1	1	1	1
CVT[T]PS2DQ xmm, xmm	4	3	3	3	1	1	1	1
CVTPS2PD xmm, xmm	5	2	2	2	1	1	1	1
CVT[T]SD2SI r64/r32, xmm	6	4	4	5	1	1	1	1
CVTSD2SS xmm, xmm	5	4	4	4	1	1	1	1
CVTSI2SD xmm, r64/r32	5	3	3	4	1	1	1	1
CVTSS2SD xmm, xmm	5	2	2	2	1	1	1	1
CVTTPD2DQ xmm, xmm	5	4	4	4	1	1	1	1
CVTTSD2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPD xmm, xmm[1]	14	<14	14-20	16-22 (15-20)	4	8	13	22(14)
DIVSD xmm, xmm	14	<14	14-20	16-22 (15-20)	4	5	13	22(14)
MAXPD xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSD xmm, xmm	4	3	3	3	0.5	1	1	1
MINPD xmm, xmm	4	3	3	3	0.5	1	1	1
MINSD xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MOVMSKPD r64/r32, xmm	2	2	2	2	1	1	1	1
MOVSD xmm, xmm	1	1	1	1	1	1	1	1
MOVUPD xmm, xmm	1	1	1	1	0.33	0.5	0.5	1
MULPD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
MULSD xmm, xmm	3	5	5	5	0.5	0.5	0.5	1
ORPD xmm, xmm	1	1	1	1	0.33	1	1	1
SHUFPD xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPD xmm, xmm[2]	18	20	20	22(21)	6	13	13	22(14)
SQRTSD xmm, xmm	18	20	20	22(21)	6	7	13	22(14)
SUBPD xmm, xmm	4	3	3	3	0.5	1	1	1

Table 7-15. Intel® SSE2 Double-Precision Floating-Point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
SUBSD xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISD xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPD xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPD xmm, xmm	1	1	1	1	1	1	1	1
XORPD[3] xmm, xmm	1	1	1	1	0.33	1	1	1

NOTES:

1. The latency and throughput of DIVPD/DIVSD can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles.
2. The latency throughput of SQRTPD/SQRTSD can vary with input value. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than10 cycles.

Table 7-16. Intel® SSE Single-Precision Floating-Point Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
ADDPS xmm, xmm	4	3	3	3	0.5	1	1	1
ADDSS xmm, xmm	4	3	3	3	0.5	1	1	1
ANDNPS xmm, xmm	1	1	1	1	0.33	1	1	1
ANDPS xmm, xmm	1	1	1	1	0.33	1	1	1
CMPPS xmm, xmm	4	3	3	3	0.5	1	1	1
CMPSS xmm, xmm	4	3	3	3	0.5	1	1	1
COMISS xmm, xmm	2	2	2	2	1	1	1	1
CVTSI2SS xmm, r32	6	4	4	5	1	1	1	1
CVTSS2SI r32, xmm	6	4	4	5	1	1	1	1
CVT[T]SS2SI r64, xmm	6	4	4	5	1	1	1	1
CVTTSS2SI r32, xmm	6	4	4	5	1	1	1	1
DIVPS xmm, xmm[1]	11	<11	<13	10-14	3	4	6	14(6)
DIVSS xmm, xmm	11	<11	<13	10-14	3	2.5	6	14(6)
MAXPS xmm, xmm	4	3	3	3	0.5	1	1	1
MAXSS xmm, xmm	4	3	3	3	0.5	1	1	1
MINPS xmm, xmm	4	3	3	3	0.5	1	1	1
MINSS xmm, xmm	4	3	3	3	0.5	1	1	1
MOVAPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MOVHLPS xmm, xmm	1	1	1	1	1	1	1	1
MOVLHPS xmm, xmm	1	1	1	1	1	1	1	1
MOVMSKPS r64/r32, xmm	2	2	2	2	1	1	1	1

Table 7-16. Intel® SSE Single-Precision Floating-Point Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)	06_4E, 06_5E	06_3D/47/56	06_3C/45/46/3F	06_2A/2D(06_3A/3E)
MOVSS xmm, xmm	1	1	1	1	1	1	1	1
MOVUPS xmm, xmm	1	1	1	1	0.25	0.5	0.5	1
MULPS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
MULSS xmm, xmm	4	3	5	5	0.5	0.5	0.5	1
ORPS xmm, xmm	1	1	1	1	0.33	1	1	1
RCPPS xmm, xmm	4	5	5	5	1	1	1	1
RCPSS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTPS xmm, xmm	4	5	5	5	1	1	1	1
RSQRTSS xmm, xmm	4	5	5	5	1	1	1	1
SHUFPS xmm, xmm, imm8	1	1	1	1	1	1	1	1
SQRTPS xmm, xmm[2]	13	13	13	14	3	7	7	14(7)
SQRTSS xmm, xmm	13	13	13	14	3	4	7	14(7)
SUBPS xmm, xmm	4	3	3	3	0.5	1	1	1
SUBSS xmm, xmm	4	3	3	3	0.5	1	1	1
UCOMISS xmm, xmm	2	2	2	2	1	1	1	1
UNPCKHPS xmm, xmm	1	1	1	1	1	1	1	1
UNPCKLPS xmm, xmm	1	1	1	1	1	1	1	1
XORPS xmm, xmm	1	1	1	1	1	1	1	1
LFENCE[3]					6	5	5	4
MFENCE[3]					~40	~35	~35	~35
SFENCE[3]					7	6	6	5
STMXCSR[3]					1	1	1	1
FXSAVE[3]					~90	~71	~75	~78

NOTES:

1. The latency and throughput of DIVPS/DIVSS can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles.
2. The latency and throughput of SQRTPS/SQRTSS can vary with input values. For certain values, hardware can complete quickly, throughput may be as low as ~ 6 cycles. Similarly, latency for certain input values may be as low as less than 10 cycles
3. The throughputs of FXSAVE/LFENCE/MFENCE/SFENCE/STMXCSR are measured with the destination in L1 Data Cache.

Table 1-17. General Purpose Instructions

Instruction	Latency[1]				Throughput			
CPUID	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E
ADC/SBB reg, reg	1	2	2	2	0.5	1	1	1
ADC/SBB reg, imm	1	2	2	2	0.5	1	1	1
ADD/SUB	1	1	1	1	0.25	0.25	0.25	0.33
AND/OR/XOR	1	1	1	1	0.25	0.25	0.25	0.33
BSF/BSR	3	3	3	3	1	1	1	1
BSWAP	2	2	2	2	0.5	0.5	0.5	1
BT	1	1	1	1	0.5	0.5	0.5	0.5
BTC/BTR/BTS	1	1	1	1	0.5	0.5	0.5	0.5
CBW/CWDE/CDQE	1	1	1	1	1	1	1	1
CDQ	1	1	1	1	1	1	1	1
CQO	1	1	1	1	0.5	0.5	0.5	0.5
CLC					0.25	0.33	0.33	0.33
CMC					0.25	0.33	0.33	0.33
STC					0.25	0.33	0.33	0.33
CLFLUSH[12]					~2 to 50	~3 to 50	~3 to 50	~5 to 50
CLFLUSHOPT[13]					~2to 10	NA	NA	NA
CMOVE/CMOVcc	1	1	2	2	0.5	0.5	0.5	0.5
CMOVBE/NBE/A/NA	2	2	3	3	1	1	1	1
CMP/TEST	1	1	1	1	0.25	0.25	0.25	0.33
CPUID (EAX = 0)					~100	~100	~100	~95
CPUID (EAX != 0)					>200	>200	>200	>200
CMPXCHG r64, r64	5	5	5	5	5	5	5	5
CMPXCHG8B m64	15	8	8	8	15	8	8	8
CMPXCHG16B m128	19	10	10	10	19	10	10	10
Lock CMPXCHG8B m64	22	19	19	24	22	19	19	24
Lock CMPXCHG16B m128	32	28	28	29	32	28	28	29
DEC/INC	1	2	2	2	0.25	0.25	0.25	0.33
IMUL r64, r64	3	3	3	3	1	1	1	1
IMUL r64[10]	4, 5	3, 4	3, 4	3, 4	1	1	1	1
IMUL r32	5	4	4	4	1	1	1	1
IDIV r64 (RDX!= 0)[8]					~85-100	~85-100	~85-100	~85-100
IDIV r32[9]					~20-26	~20-26	~20-26	~19-25
LEA	1	1	1	1	0.5	0.5	0.5	0.5
LEA [base+index]disp	3	3	3	3	1	1	1	1
MOVSB/MOVSW	1	1	1	1	0..25	0..25	0..25	0.33
MOVZB/MOVZW	1	1	1	1	0.25	0.25	0.25	0.33
DIV r64 (RDX!= 0)[8]					~80-95	~80-95	~80-95	~80-95
DIV r32[9]					~20-26	~20-26	~20-26	~19-25

Table 1-17. General Purpose Instructions (Contd.)

Instruction	Latency[1]				Throughput			
CPUID	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E
MUL r64[10]	4, 5	3, 4	3, 4	3, 4	1	1	1	1
NEG/NOT	1	2	2	2	0.25	0.25	0.25	0.33
PAUSE					~140	~10	~10	~10
RCL/RCR reg, 1	2	2	2	2	2	1.5	1.5	1.5
RCL/RCR	6	6	6	6	6	6	6	6
RDTSC					~13	~10	~10	~20
RDTSCP					~20	~30	~30	~30
ROL/ROR reg 1	1 (2 flg)	1 (2 flg)	1 (2 flg)	1 (2 flg)	1	1	1	1
ROL/ROR reg imm	1	1	1	1	0.5	0.5	0.5	0.5
ROL/ROR reg, cl	2	2	2	2	1.5	1.5	1.5	1.5
LAHF/SAHF	3	2	2	2				
SAL/SAR/SHL/SHR reg, imm	1	1	1	1	0.5	0.5	0.5	0.5
SAL/SAR/SHL/SHR reg, cl	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
SETBE	2	2	2	2	1	1	1	1
SETE	1	1	1	1	0.5	0.5	0.5	0.5
SHLD/RD reg, reg, cl	6	4	4	2 (4 flg)	1.5	1	1	1.5
SHLD/RD reg, reg, imm	3	3	3	1	0.5	0.5	0.5	0.5
XSAVE[11]					~98	~100	~100	~100
XSAVEOPT[11]					~86	~90	~90	~90
XADD	2	2	2	2	1	1	1	1
XCHG reg, reg	1	1	1	2	1	1	1	1
XCHG reg, mem	22	19	19	19	22	19	19	19

7.3.2 Table Footnotes

The following footnotes refer to all tables in this appendix.

1. Latency information for many instructions that are complex (> 4 μops) are estimates based on conservative (worst-case) estimates. Actual performance of these instructions by the out-of-order core execution unit can range from somewhat faster to significantly faster than the latency data shown in these tables.
2. Latency and Throughput of transcendental instructions can vary substantially in a dynamic execution environment. Only an approximate value or a range of values are given for these instructions.
3. It may be possible to construct repetitive calls to some Intel 64 and IA-32 instructions in code sequences to achieve latency that is one or two clock cycles faster than the more realistic number listed in this table.
4. The FXCH instruction has 0 latency in code sequences. However, it is limited to an issue rate of one instruction per clock cycle.
5. The load constant instructions, FINCSTP, and FDECSTP have 0 latency in code sequences.
6. Selection of conditional jump instructions should be based on the recommendation of Section 3.4.1, "Branch Prediction Optimization," to improve the predictability of branches. When branches are predicted successfully, the latency of jcc is effectively zero.

7. RCL/RCR with shift count of 1 are optimized. Using RCL/RCR with shift count other than 1 will be executed more slowly. This applies to the Pentium 4 and Intel Xeon processors.
8. The throughput of "DIV/IDIV r64" varies with the number of significant digits in the input RDX:RAX. The throughput is significantly higher if RDX input is 0, similar to those of "DIV/IDIV r32". If RDX is not zero, the throughput is significantly lower, as shown in the range. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input RDX:RAX (relative to the number of significant bits of the divisor) or the output quotient. The latency of "DIV/IDIV r64" also varies with the significant bits of input values. For a given set of input values, the latency is about the same as the throughput in cycles.
9. The throughput of "DIV/IDIV r32" varies with the number of significant digits in the input EDX:EAX and/or of the quotient of the division for a given size of significant bits in the divisor r32. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input EDX:EAX or the output quotient. The latency of "DIV/IDIV r32" also varies with the significant bits of the input values. For a given set of input values, the latency is about the same as the throughput in cycles.
10. The latency of MUL r64 into 128-bit result has two sets of numbers, the read-to-use latency of the low 64-bit result (RAX) is smaller. The latency of the high 64-bit of the 128 bit result (RDX) is larger.
11. The throughputs of XSAVE and XSAVEOPT are measured with the destination in L1 Data Cache and includes the YMM states.
12. CLFLUSH throughput is representative from clean cache lines for a range of buffer sizes. CLFLUSH throughput can decrease significantly by factors including: (a) the number of back-to-back CLFLUSH being executed, (b) flushing modified cache lines incurs additional cost than cache lines in other coherent state. See Section 9.4.6.
13. CLFLUSHOPT throughput is representative from clean cache lines for a range of buffer sizes. CLFLUSHOPT throughput can decrease by factors including: (a) flushing modified cache lines incurs additional cost than cache lines in other coherent state, (b) the number of cache lines back-to-back. See Section 9.4.7.

7.3.3 Instructions with Memory Operands

The latency of an Instruction with memory operand can vary greatly due to a number of factors, including data locality in the memory/cache hierarchy and characteristics that are unique to each microarchitecture. Generally, software can approach tuning for locality and instruction selection independently. Thus Table 7-4 through Table 7-18 can be used for the purpose of instruction selection. Latency and throughput of data movement in the cache/memory hierarchy can be dealt with independent of instruction latency and throughput. Load-to-use Latency of the cache hierarchy can be found in Chapter 2.

7.3.3.1 Software Observable Latency of Memory References

When measuring latency of memory references of individual instructions, many factors can influence the observed latency exposure. Aside from access patterns, cache locality, effect of the hardware prefetchers, different microarchitectures may expose variability such register domains of the destination or memory addressing form with respect to the instruction encoding.

Table 7-18 gives a few selected sampling of the variability of L1D cache hit latency that software may observe using pointer-chasing constructs, due to memory reference encoding details, on recent Intel microarchitectures.

Table 7-18. Pointer-Chasing Variability of Software Measurable Latency of L1 Data Cache Latency

Pointer Chase Construct	L1D latency Observation
MOV rax, [rax]	4
MOV rax, disp32[rax] , disp32 < 2048	4
MOV rax, [rcx+rax]	5
MOV rax, disp32[rcx+rax] , disp32 < 2048	5