# Optimizing Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency

Document Number: 356477-050US
April 2024

## Revision History

| Date | Revision | Description |
|---|---|---|
| August 2023 | 048 | Initial release of document. |
| January 2024 | 049 | • Updated fonts for accessibility.<br>• Changed title. |
| April 2024 | 050 | • Added Chapter 8: Intel® Transactional Synchronization Extensions (Intel® TSX) Optimizations.<br>• Fixed Headings. |

# CONTENTS

PAGE

# CONTENTS

PAGE

# CONTENTS

PAGE

PAGE

# TABLES

## FIGURES

# CHAPTER 1
# HASWELL MICROARCHITECTURE

# CHAPTER 2
# SANDY BRIDGE MICROARCHITECTURE

# CHAPTER 3
# INTEL® CORE™ MICROARCHITECTURE AND ENHANCED INTEL® CORE™ MICROARCHITECTURE

# CHAPTER 6
# EARLIER GENERATIONS OF INTEL ATOM® MICROARCHITECTURE AND SOFTWARE OPTIMIZATION

# TABLES

# TABLES

# EXAMPLES

## FIGURES

# 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.

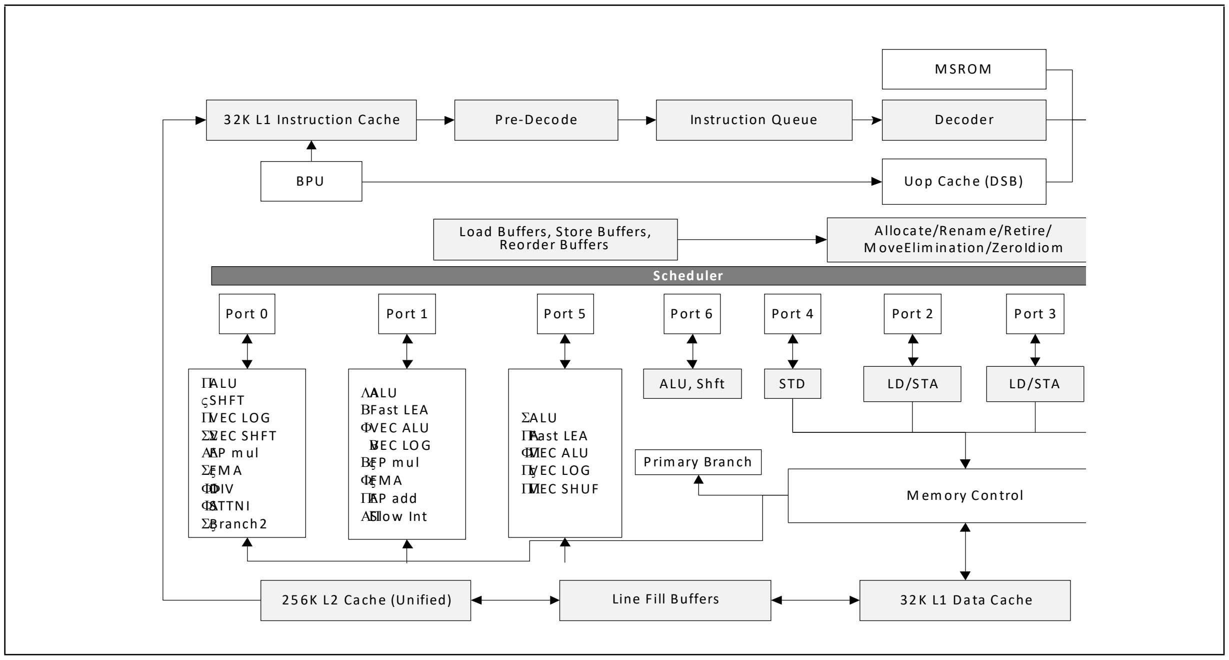


Figure 1-1. CPU Core Pipeline Functionality of the Haswell Microarchitecture

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.

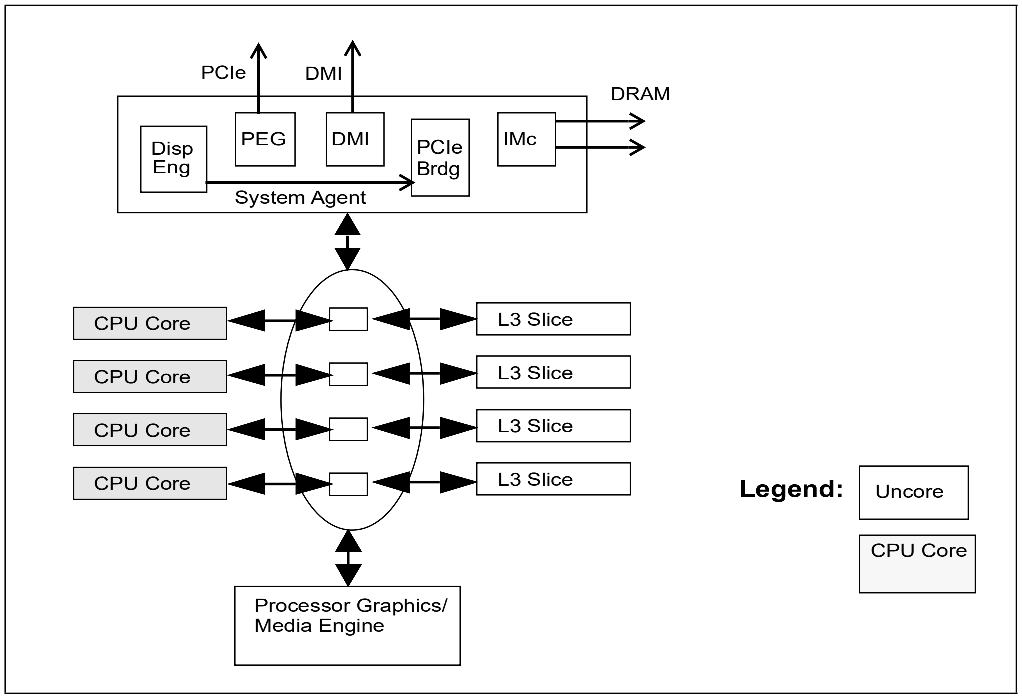


Figure 1-2. Four Core System Integration of the Haswell Microarchitecture

## 1.2 THE FRONT END

The front end of Haswell microarchitecture builds on that of the Sandy Bridge and Ivy Bridge microarchitectures, see Chapter 2, "Sandy Bridge Microarchitecture" and Section 2.3, "Ivy Bridge Microarchitecture". 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,<br>• Shift | • ALU,<br>• Fast LEA,<br>• BM | • Load_Addr,<br>• Store_addr | Store_data | • ALU,<br>• Fast LEA,<br>• BM | • ALU,<br>• Shift,<br>• JEU | • Store_addr,<br>• Simple_AGU |
| • SIMD_Log, SIMD misc, SIMD_Shifts | • SIMD_ALU, SIMD_Log | | | • SIMD_ALU,<br>• SIMD_Log, | | |
| • FMA/FP_mul<br>• Divide | • FMA/FP_mul<br>• FP_add | | | Shuffle | | |
| 2nd_Jeu | slow_int, | | | • FP mov<br>• 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 1-3. Bypass Delay Between Producer and Consumer Micro-ops (Cycles)

| From/To | INT | SSE-INT/AVX-INT | SSE-FP/AVX-P_LOW | X87/AVX-FP_High |
|---|---|---|---|---|
| INT | | • micro-op (port 5)<br>• micro-op (port 6) + 1 cycle | • micro-op (port 5)<br>• micro-op (port 6) + 1 cycle | micro-op (port 5) + 3 cycle delay |
| SSE-INT/ AVX-INT | micro-op (port 1) | | 1 cycle delay | |

Table 1-3. Bypass Delay Between Producer and Consumer Micro-ops (Cycles) (Contd.)

| From/To | INT | SSE-INT/AVX-INT | SSE-FP/AVX-P_LOW | X87/AVX-FP_High |
|---|---|---|---|---|
| SSE-FP/ AVX-FP_LOW | micro-op (port 1) | 1 cycle delay | | micro-op (port 5) + 1 cycle 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 third 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 1-4 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 1-6. Components of the Front End**

| Instruction Example | Source | Destination | Source & 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 sub-system 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.

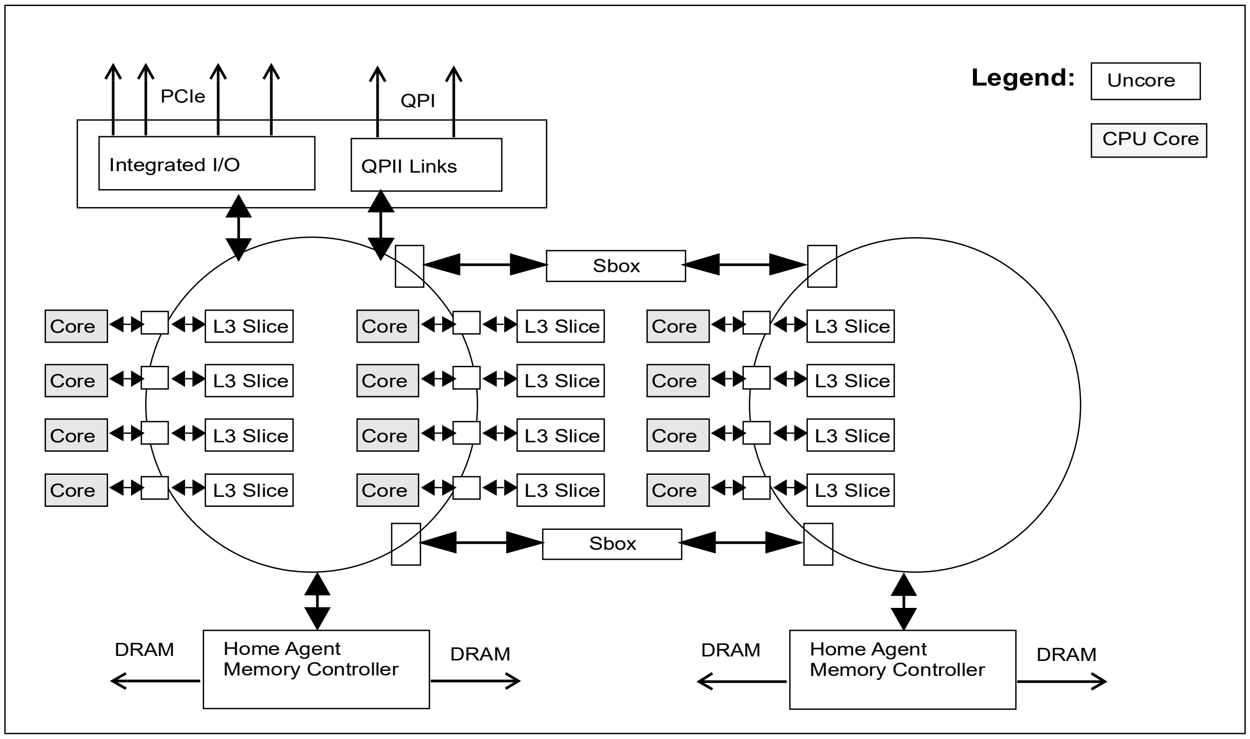


**Figure 1-3. An Example of the Haswell-E Microarchitecture Supporting 12 Processor Cores**

## 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 enhpoanced features of the Broadwell microarchitecture.

- Floating-int 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.
- 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.

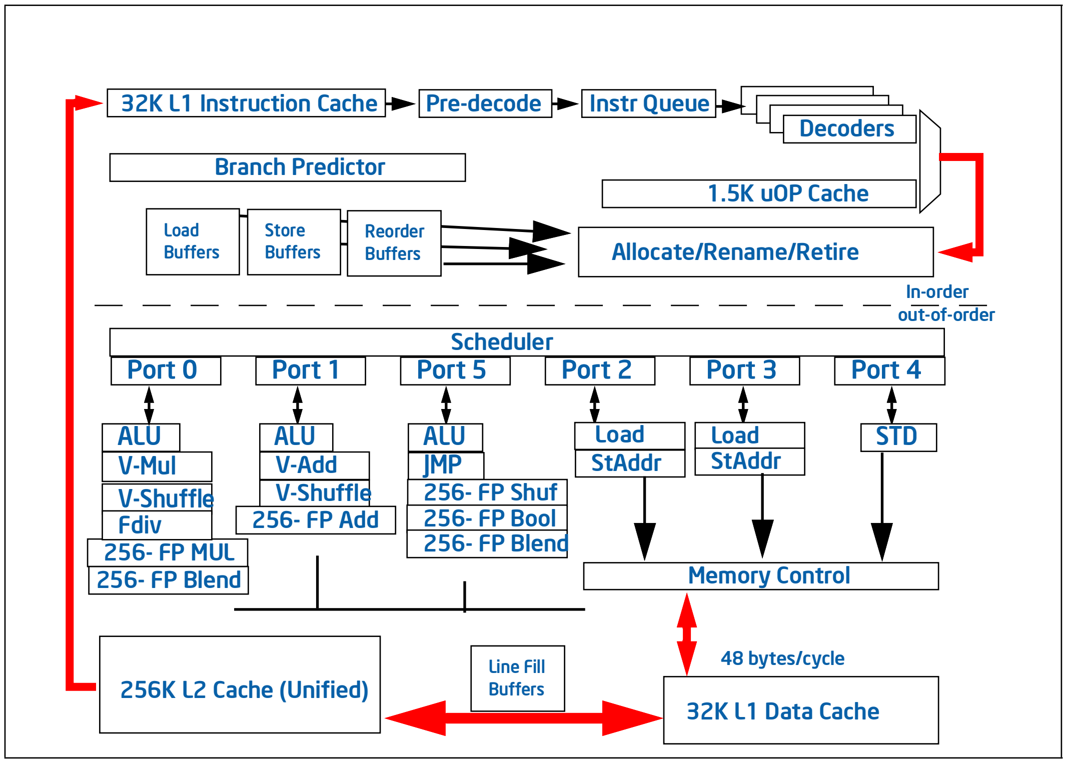


Figure 2-1. Sandy Bridge Microarchitecture Pipeline Functionality

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 2-1 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 pre-decode, 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 2-1 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.

### MacroFusion

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.**

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.

#### 2.1.1.2 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.

### 2.1.1.3 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.

### 2.1.1.4 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.

### 2.1.2 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.

#### 2.1.2.1 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.

#### 2.1.2.2 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.

### 2.1.3 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)<br>▪ micro-op (port 5) +1 cycle | 0 | 1 cycle delay | 0 |
| SSE-FP, AVX-FP_low | ▪ micro-op (port 5)<br>▪ 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 |

### 2.1.4 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 |

#### 2.1.4.1 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.

### 2.1.5 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.

#### 2.1.5.1 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 2.1.3 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.

### 2.1.5.2 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 |

### 2.1.5.3 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 |

### 2.1.5.4 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.

## 2.1.6 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.

## 2.1.7 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.

## 2.2 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 IVY BRIDGE MICROARCHITECTURE

3rd generation Intel Core processors are based on Ivy Bridge microarchitecture. Most of the features described in Section 2.1 - Section 2.2 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.

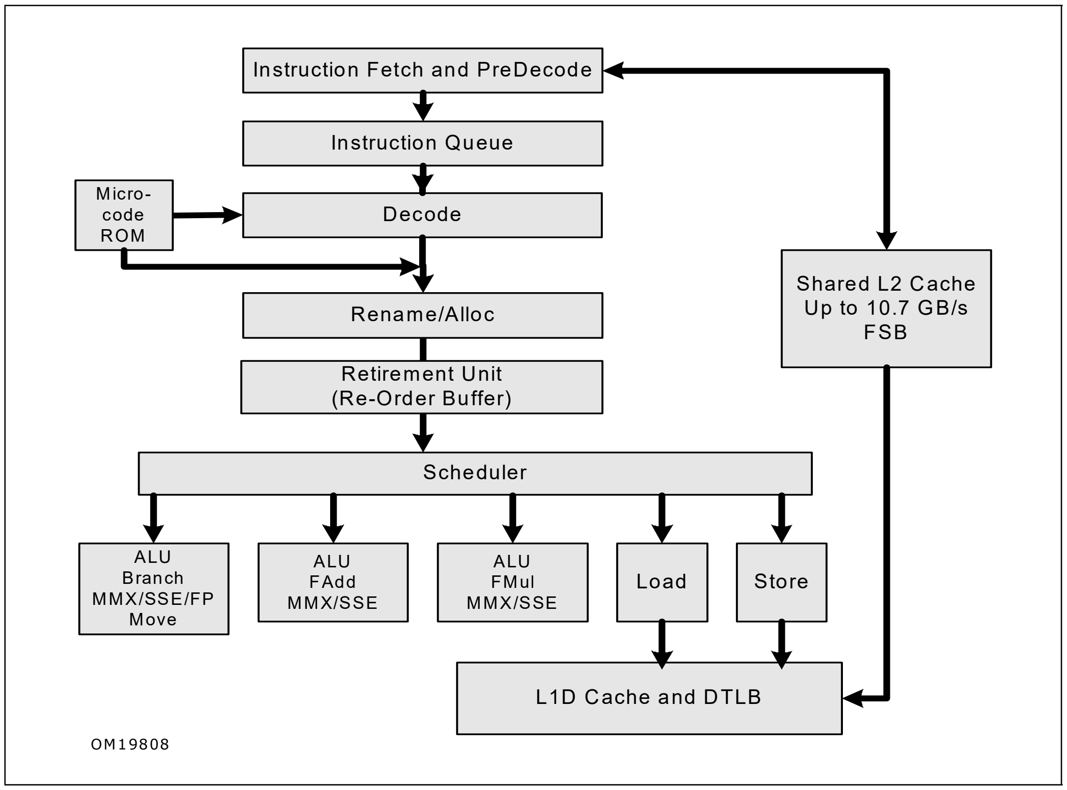

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
Shared L2 Cache Up to 10.7 GB/s FSB
OM19808

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.<br>• Uses dedicated hardware for each type. | • Enables speculative execution.<br>• 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<br>• Caches frequently-used instructions<br>• Pre-decodes and buffers instructions, maintaining a constant bandwidth despite irregularities in the instruction stream | • Variable length instruction format causes unevenness (bubbles) in decode bandwidth.<br>• 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<br>• Stack pointer tracker algorithm for efficient procedure entry and exit<br>• Implements the Macro-Fusion feature, providing higher performance and efficiency<br>• 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.<br>• Prefix adds a dimension of decoding complexity.<br>• 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.

#### 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 pre-decode logic of the instruction queue (IQ).

## 3.2 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 pre-decoder. 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.

### 3.2.1 Instruction Pre-Decode

The pre-decode 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 pre-decode unit can write up to six instructions per cycle into the instruction queue. If a fetch contains more than six instructions, the pre-decoder 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 pre-decoding after the current fetch completes.

For a fetch of seven instructions, the pre-decoder 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 pre-decoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the pre-decoder 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.

#### 3.2.1.1 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.

#### 3.2.1.2 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 micro-sequencer 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.

#### 3.2.1.3 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.

#### 3.2.1.4 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. 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.

## 3.2.2 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 3.3 for more information about the MOB.

### 3.2.2.1 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<br>Integer SIMD ALU<br>FP/SIMD/SSE2 Move and Logic | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | Includes 64-bit mode integer MUL;<br>Issue port 0; Writeback port 0; |
| Single-precision (SP) FP MUL<br>Double-precision FP MUL | 4, 1<br>5, 1 | 4, 1<br>5, 1 | Issue port 0; Writeback port 0 |
| FP MUL (X87)<br>FP Shuffle<br>DIV/SQRT | 5, 2<br>1, 1 | 5, 2<br>1, 1 | Issue port 0; Writeback port 0<br>FP shuffle does not handle QW shuffle. |
| Integer ALU<br>Integer SIMD ALU<br>FP/SIMD/SSE2 Move and Logic | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | Excludes 64-bit mode integer MUL;<br>Issue port 1; Writeback port 1; |
| FP ADD<br>QW Shuffle | 3, 1<br>1, 1[2] | 3, 1<br>1, 1[3] | Issue port 1; Writeback port 1; |
| Integer loads<br>FP loads | 3, 1<br>4, 1 | 3, 1<br>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<br>Integer SIMD ALU<br>FP/SIMD/SSE2 Move and Logic | 1, 1<br>1, 1<br>1, 1 | 1, 1<br>1, 1<br>1, 1 | Issue port 5; Writeback port 5; |
| QW shuffles<br>128-bit Shuffle/Pack/Unpack | 1, 1[2]<br>2-4, 2-4[6] | 1, 1[3]<br>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.

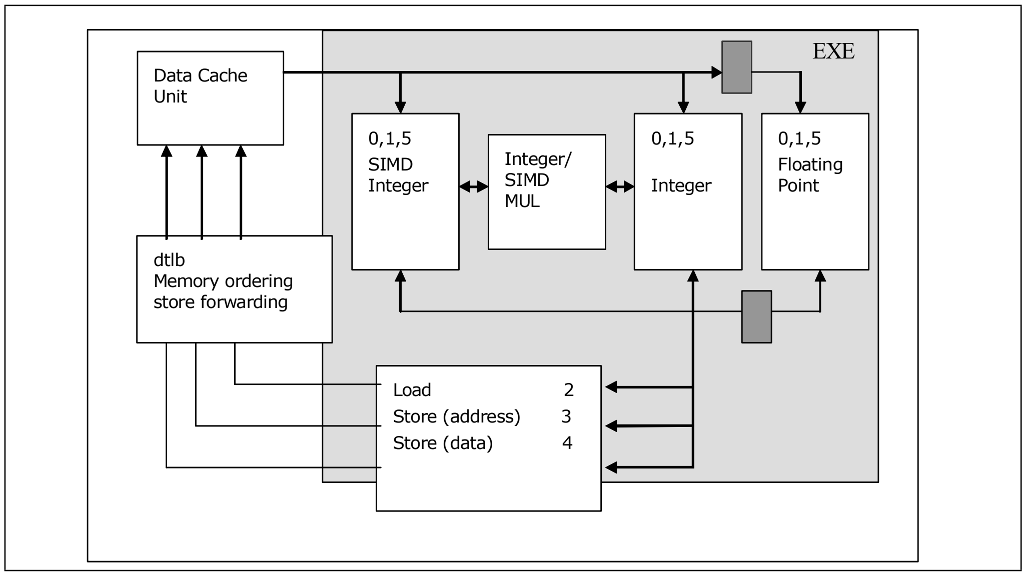


**Figure 3-2. Execution Core of Intel Core Microarchitecture**

Notice that the two dark squares inside the execution block (in the 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.

## 3.3 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).

### 3.3.1 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.

#### 3.3.1.1 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.

#### 3.3.1.2 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.

#### 3.3.1.3 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.

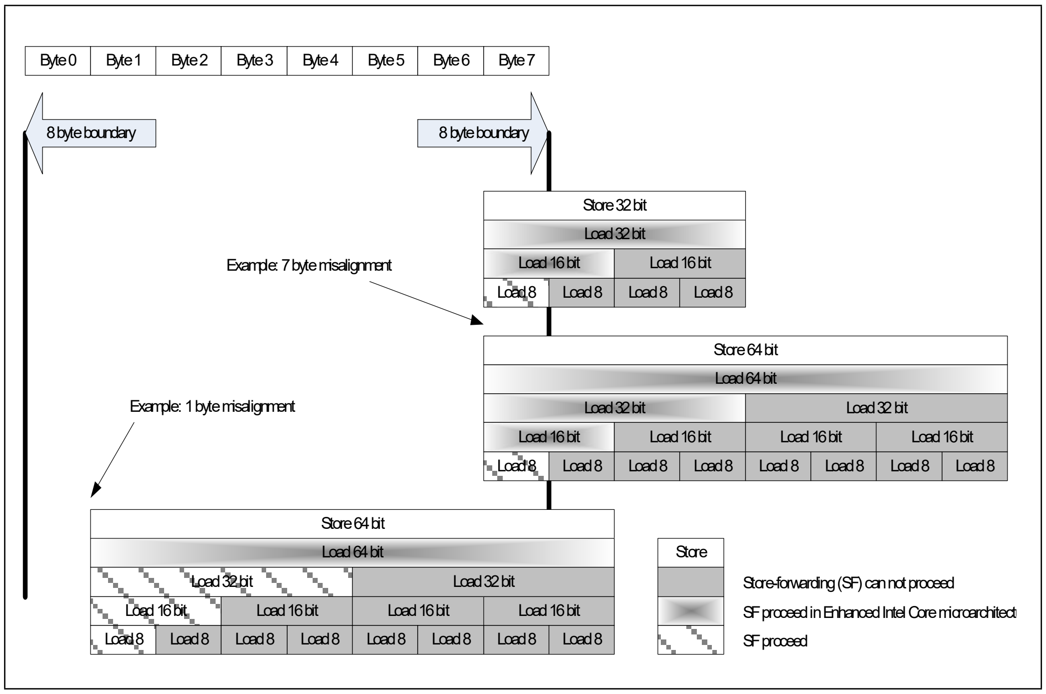


Figure 3-3. Store-Forwarding Enhancements in Enhanced Intel® Core™ Microarchitecture

#### 3.3.1.4 Memory Disambiguation

Refer to the “Memory Disambiguation” details in Section 2.1.5, “L1 DCache”.

### 3.3.2 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.

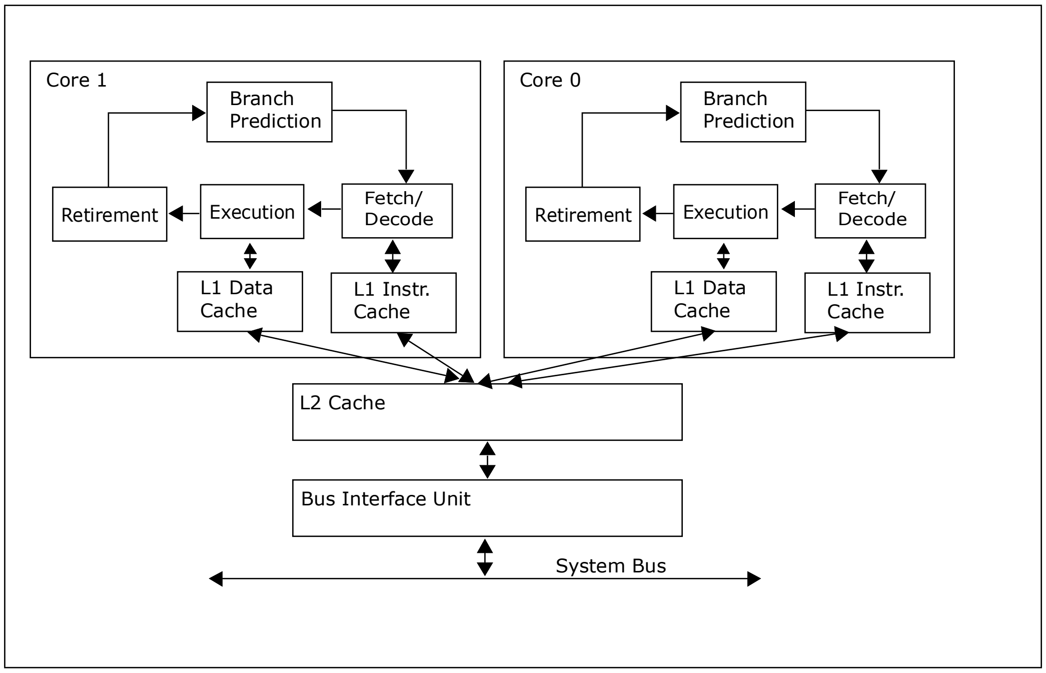


**Figure 3-4. Intel® Advanced Smart Cache Architecture**

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).

#### 3.3.2.1 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.

#### 3.3.2.2 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.

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.

# 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.

## 4.1 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.

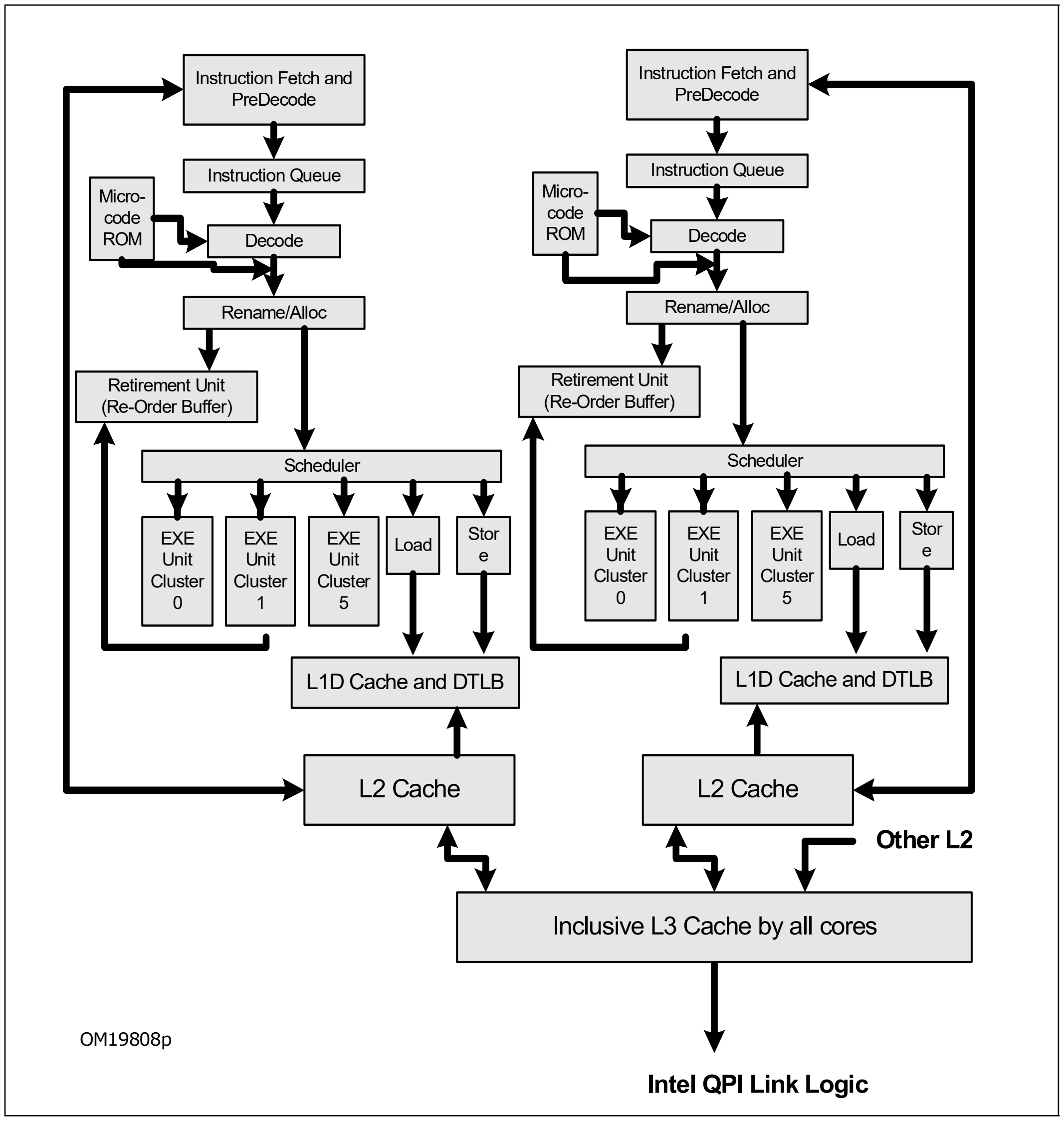


**Figure 4-1. Nehalem Microarchitecture Pipeline Functionality**

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.

### 4.1.1 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.

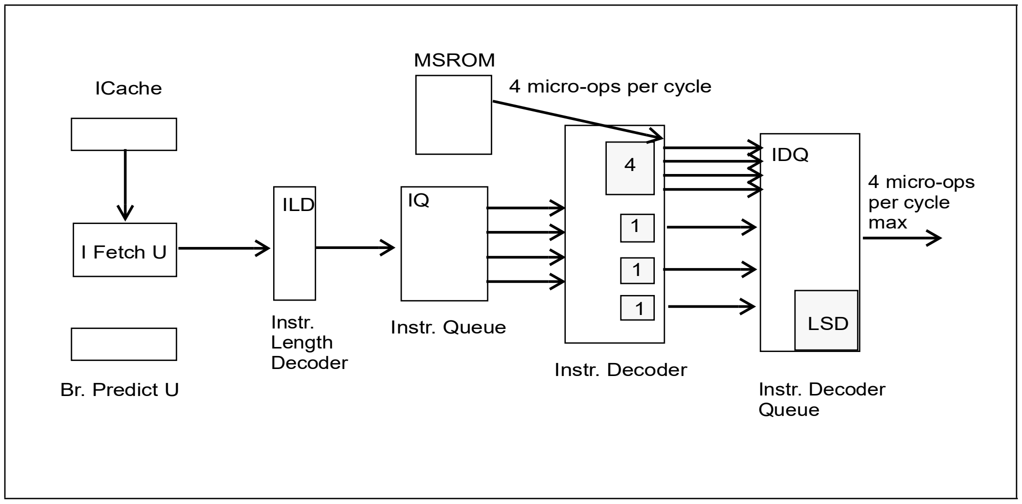


**Figure 4-2. Front End of Nehalem Microarchitecture**

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.

### 4.1.2 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 |

### 4.1.3 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<br>• Integer Shift | 1<br>1 | 1<br>1 | Integer | |
| Port 0 | • Integer SIMD ALU<br>• Integer SIMD Shuffle | 1<br>1 | 1<br>1 | SIMD | |
| Port 0 | • Single-precision (SP) FP MUL<br>• Double-precision FP MUL<br>• FP MUL (X87)<br>• FP/SIMD/SSE2 Move and Logic<br>• FP Shuffle<br>• DIV/SQRT | 4<br>5<br>5<br>1<br><br>1 | 1<br>1<br>1<br>1<br><br>1 | FP | |
| Port 1 | • Integer ALU<br>• Integer LEA<br>• Integer Mul | 1<br>1<br>3 | 1<br>1<br>1 | Integer | |
| Port 1 | • Integer SIMD MUL<br>• Integer SIMD Shift<br>• PSAD<br>• StringCompare | 1<br>1<br>3 | 1<br>1<br>1 | SIMD | |
| 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<br>• Integer Shift<br>• Jmp | 1<br>1<br>1 | 1<br>1<br>1 | Integer | |
| Port 5 | • Integer SIMD ALU<br>• Integer SIMD Shuffle | 1<br>1 | 1<br>1 | SIMD | |
| Port 5 | FP/SIMD/SSE2 Move and Logic | 1 | 1 | FP | |

**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.

### 4.1.4 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.

### 4.1.5 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 | 45nm Intel® Core™ Microarchitecture | 65nm 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.

## 4.1.6 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.

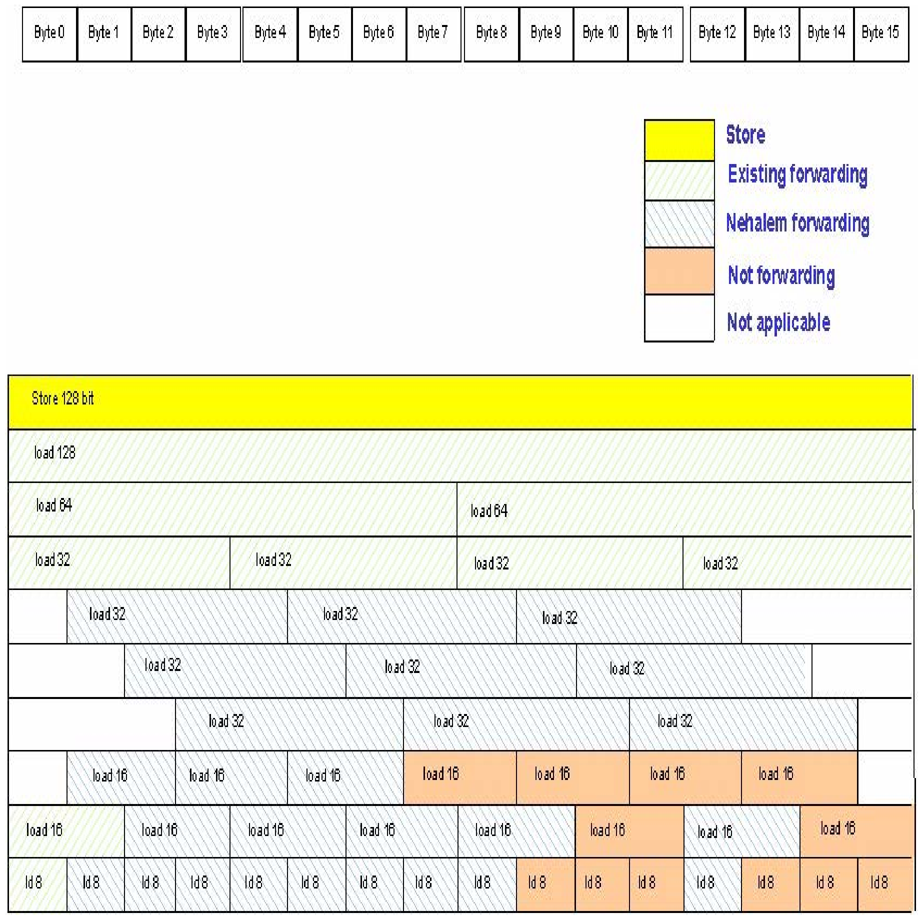


Figure 4-3. Store-Forwarding Scenarios of 16-Byte Store Operations

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.

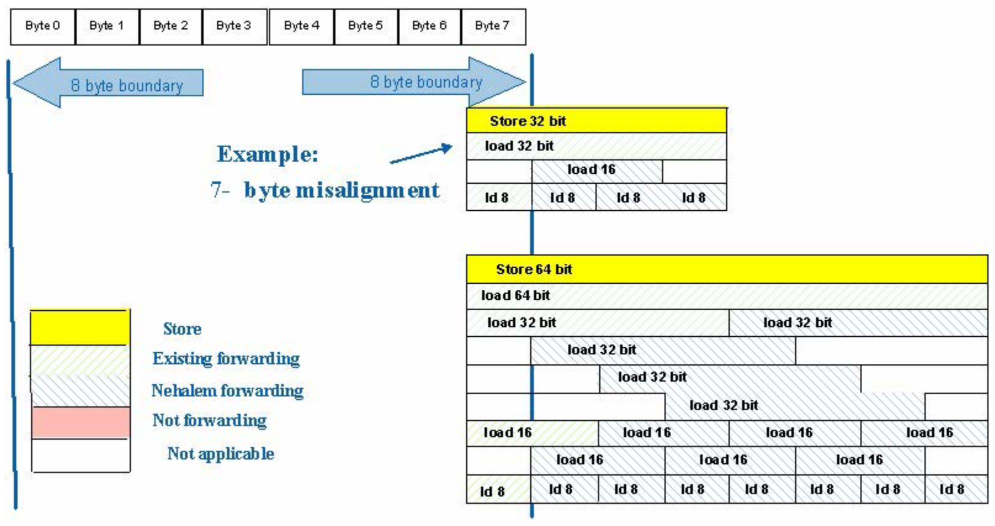


Figure 4-4. Store-Forwarding Enhancement in Nehalem Microarchitecture

## 4.2 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.

## 4.3 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.

### 4.3.1 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.

### 4.3.2 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 PhiTM 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, "Model-Specific Registers (MSRs)" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

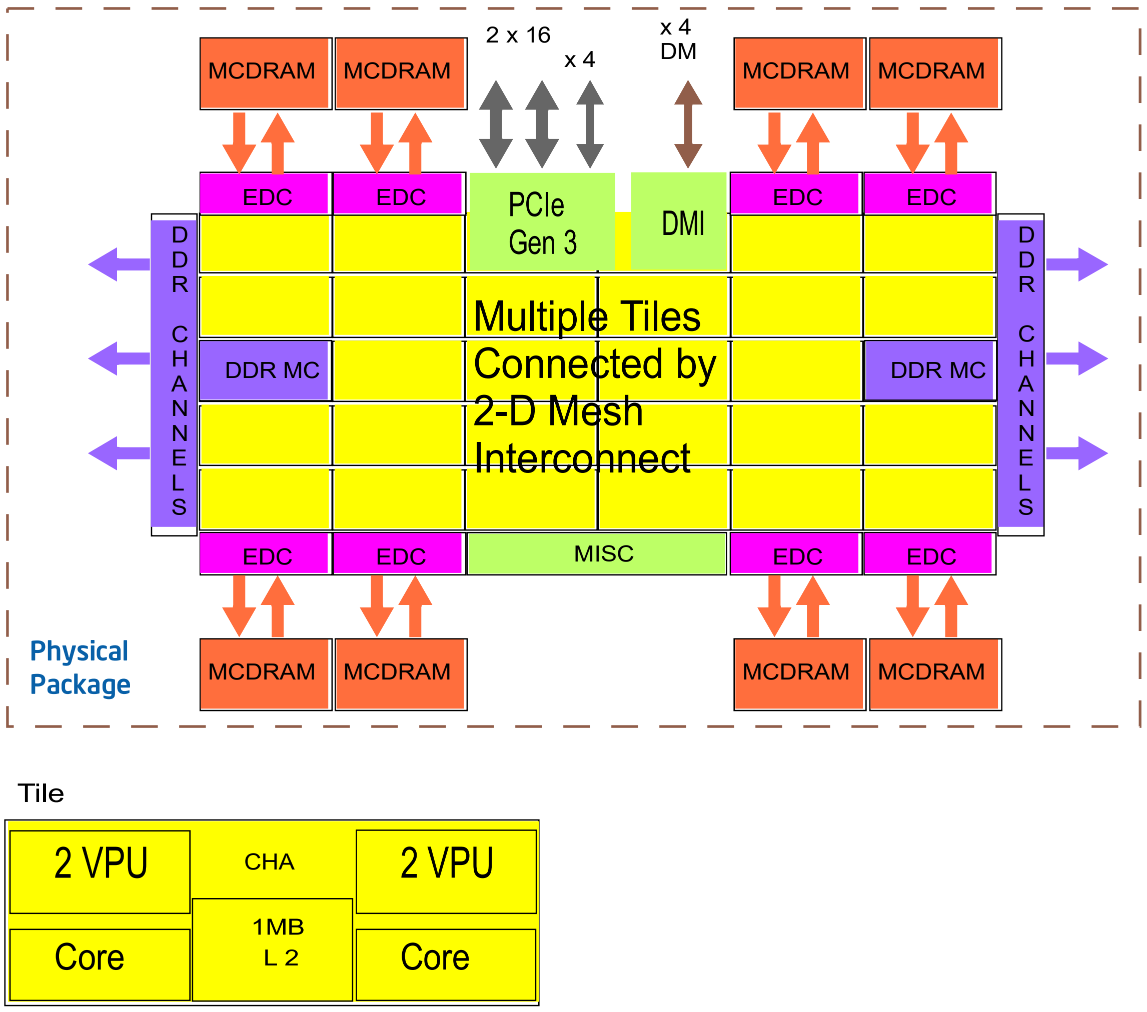


Figure 5-1. Tile-Mesh Topology of the Knights Landing Microarchitecture

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.

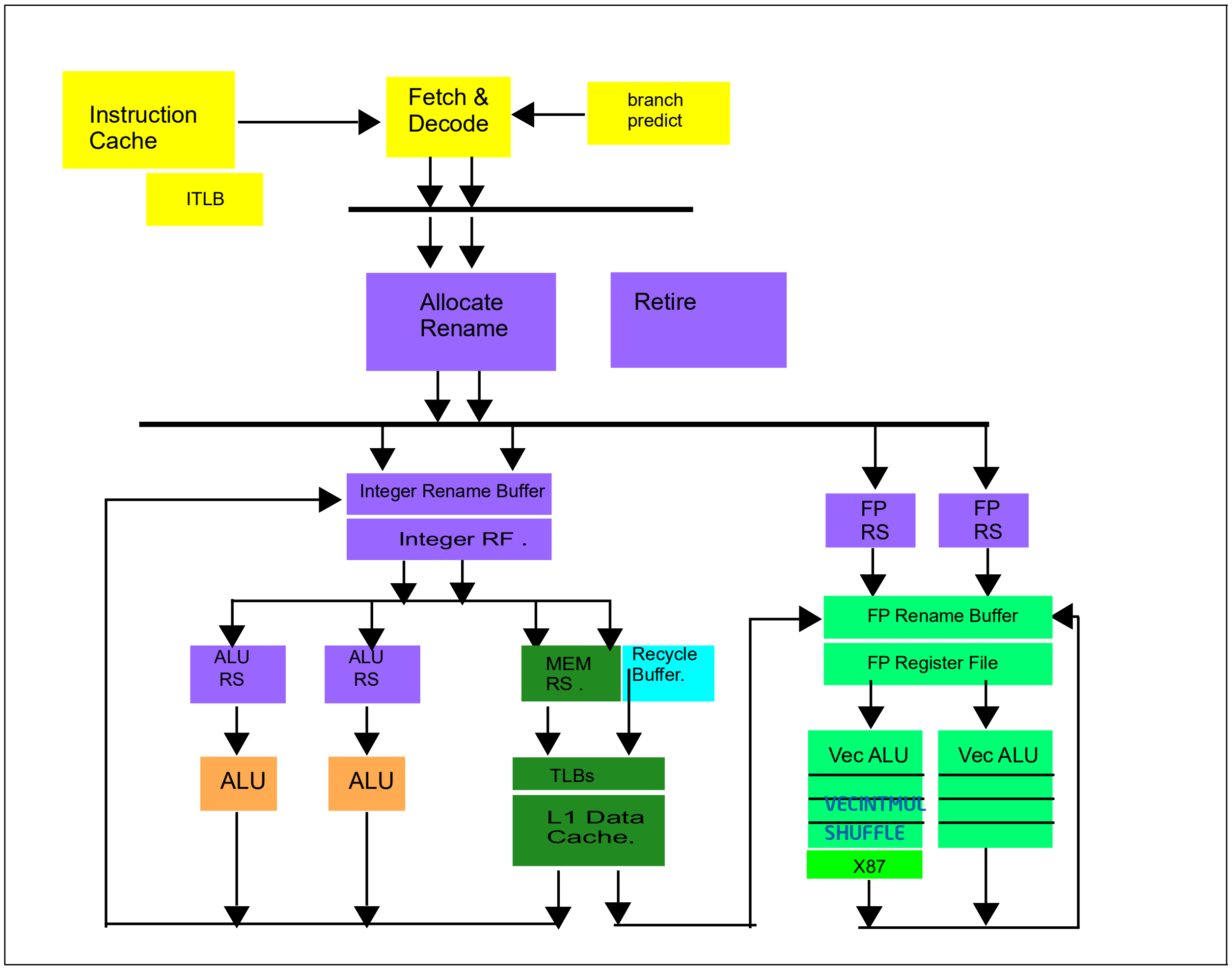


**Figure 5-2. Processor Core Pipeline Functionality of the Knights Landing Microarchitecture**

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 3-7 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 3 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 1, 2, or 4 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 |

## Table 5-2. Vector Pipeline Characteristics of the Knights Landing Microarchitecture (Contd.)

| Vector Instructions | Latency (cycle) | Throughput (cycles per instruction) |
|---|---|---|
| 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 |

Table 5-3. Characteristics of Caching Resources (Contd.)

| | Sets | Ways | Latency | Capacity/Comments |
|---|---|---|---|---|
| **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[]<br>kxnor k1,k0, k0; prepare mask<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>; compute sequence using vector register | vmovdqu ymm0, [rsp+0x1000] ; load half of index vector<br>vmovdqu ymm3, [rsp+0x1020] ; 2nd half of indirect[]<br>vpcmpeqdd ymm4, ymm4, ymm4 ; prepare mask<br>vmovdqaymm1, ymm4<br>vpgatherdd ymm2, [rax+ymm0*4], ymm1<br>vpgatherdd ymm5, [rax+ymm3*4], ymm4<br>; 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[]<br>kxnor k1,k0, k0; prepare mask<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>; compute sequence using vector register | vmovdqu zmm0, [rsp+0x1000] ; load indirect[]<br>kxnor k1,k1 ; prepare mask<br>g_loop: ; verify gathered elements are complete<br>vpgatherdd zmm2{k1}, [rax+zmm0*4]<br>jknzd k1, g_loop ; gather latency exposure<br>; 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 5.6.5, “Way, Set Conflicts” 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 four 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 two or four, 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, four 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 four logical processors. Because of this, a three-thread configuration will have fewer aggregate resources available than one, two, or four threads per core. Placing three threads on a processor core is unlikely to perform better than two or four 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 |

Table 5-4. Alternatives to MSROM Instructions (Contd.)

| | |
|---|---|
| (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.

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 5-5.

In situations where an algorithm needs to perform reduction, reduction can often be implemented without horizontal addition.

Example 5-6 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 5-6, 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 5-6 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 5-5. 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 5-6. 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.

#### 5.6.3.1 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 | 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 | zmm1, [r15+rax*8] ; executes in cycle N+4 | movq | r15, [rsp+0x48] ; cycle N+4 (load &b[0]) |
| vmovups | zmm2, [r14+rax*8] ; cycle N+5 | 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.

### 5.6.4 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.

### 5.6.5 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.

### 5.6.6 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.

### 5.6.7 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.

### 5.6.8 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 ad-dress. 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.

# 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.

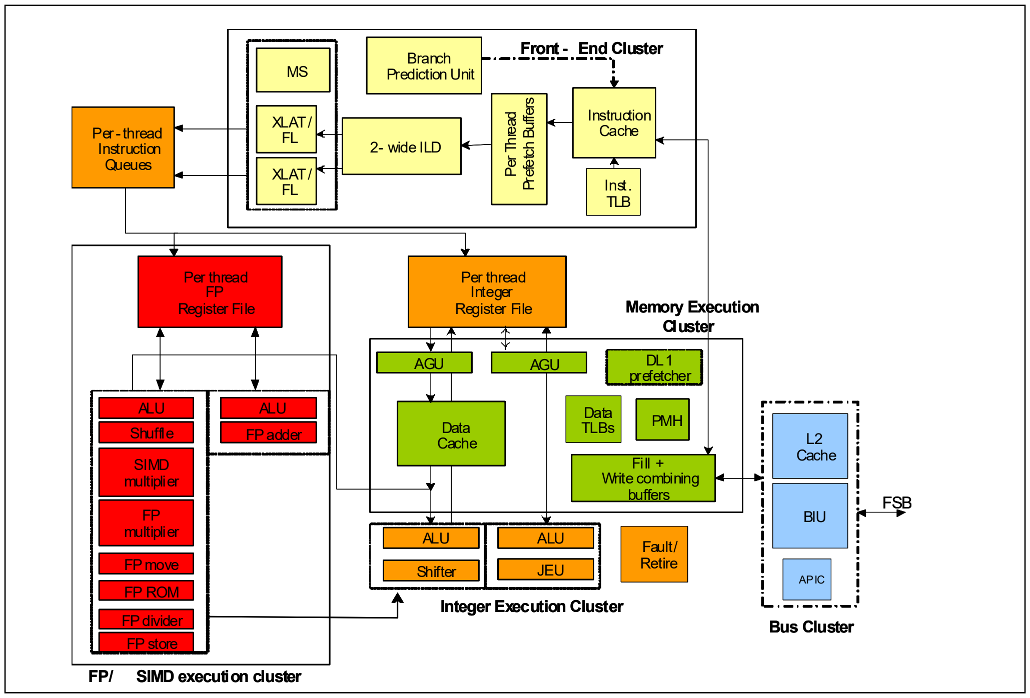


**Figure 6-1. Intel Atom® Microarchitecture Pipeline**

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 Intel® Hyper-Threading Technology (Intel® HT) 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] |
| 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 three 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 |
| 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 |

### Table 6-1. Instruction Latency/Throughput Summary of Intel Atom® Microarchitecture

| Instruction Category | Latency (cycles) | Throughput | # of Execution Unit |
|---|---|---|---|
| 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 sub-system.

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:
  - The store and load operations must be of the same size and to the same address.
  - 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
```

### Example 6-5. Examples of Dependent Multiply and Add Computation (Contd.)

```
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. Example 6-5a shows one technique to put the value of IP into the ECX register by issuing a CALL without a matching RET. Example 6-5b shows 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
```

### Example 6-6. Instruction Pointer Query Techniques (Contd.)

```
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;<br>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;<br>ADD/AND/CMP/OR/SUB/XOR[2] reg, mem; | 0 | 1 | 1 |
| ADD/AND/CMP/OR/SUB/XOR[2] reg, Imm8;<br>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 |

**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** |
| 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 |
| 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 |

### 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** |
| 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 |
| 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 |

## 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** |
| 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 |
| MOVSHDUP/MOVSLDUP/MOVUPS xmm, xmm | (0, 1) | 1 | 0.5 |

**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** |
| 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 |

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** |
| 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 |
| 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 |

## 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** |
| 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 |
| 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 |

**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** |
| 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.

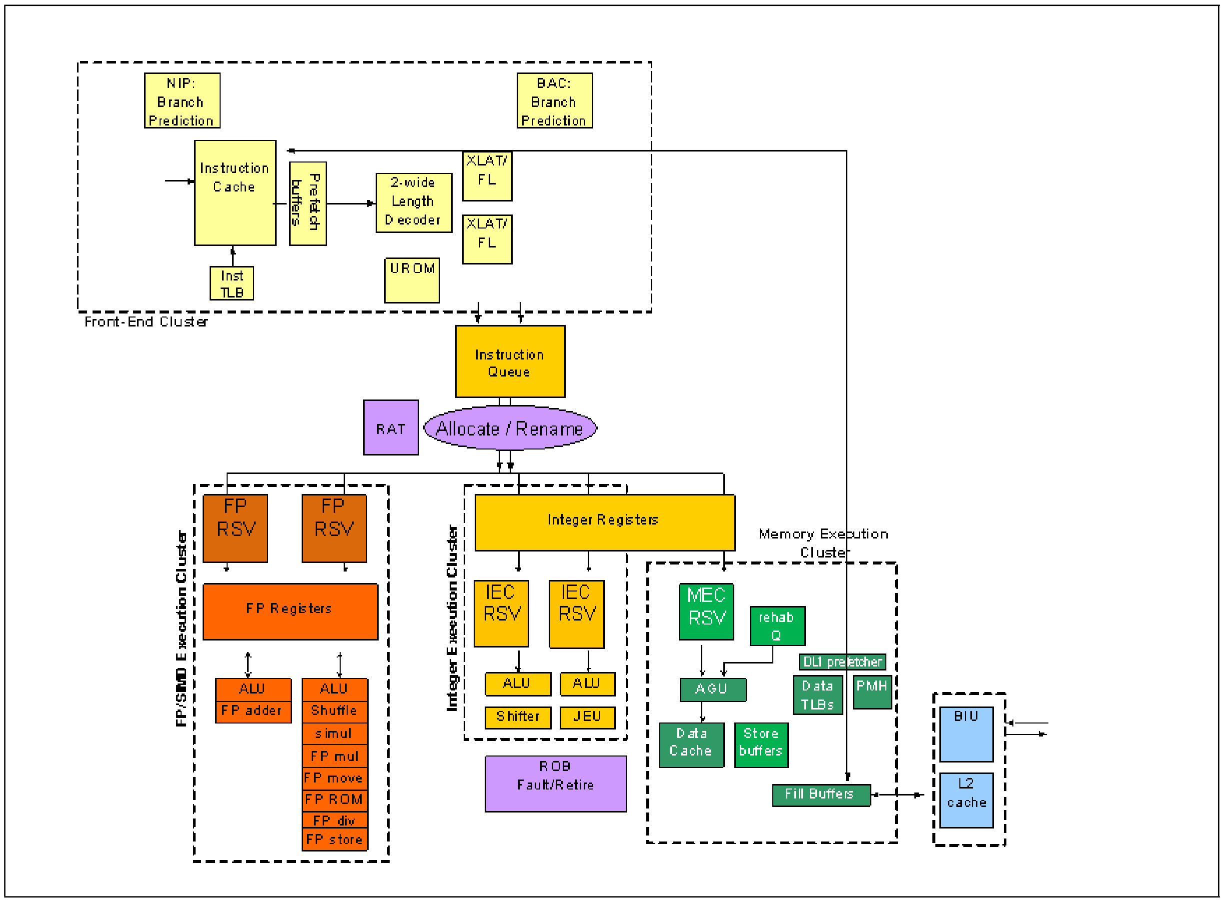


Figure 6-2.  Silvermont Microarchitecture Pipeline

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 two 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 two 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<br>• Shift/Rotate unit.<br>• LEA with no index. | • ALU1, Bit processing unit<br>• Jump unit<br>• IMUL<br>• POPCNT<br>• CRC32<br>• LEA[1] |
| **FPC** | • SIMD ALU, SIMD shift/shuffle unit.<br>• SIMD FP mul/div/cvt unit.<br>• STTNI/AESNI/PCLMULQDQ unit.<br>• RCP/RSQRT unit, F2I convert unit. | • SIMD ALU<br>• SIMD FPadd unit<br>• 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 three-wide superscalar pipeline. Specifically:
  - The decoder can decode three instructions per cycle.
  - The microcode sequencer can send three uops per cycle for allocation into the reservation stations.
  - Retirement supports a peak rate of three per cycle.
- Enhancement in branch prediction which decouples 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 one cycle throughput (versus five 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.

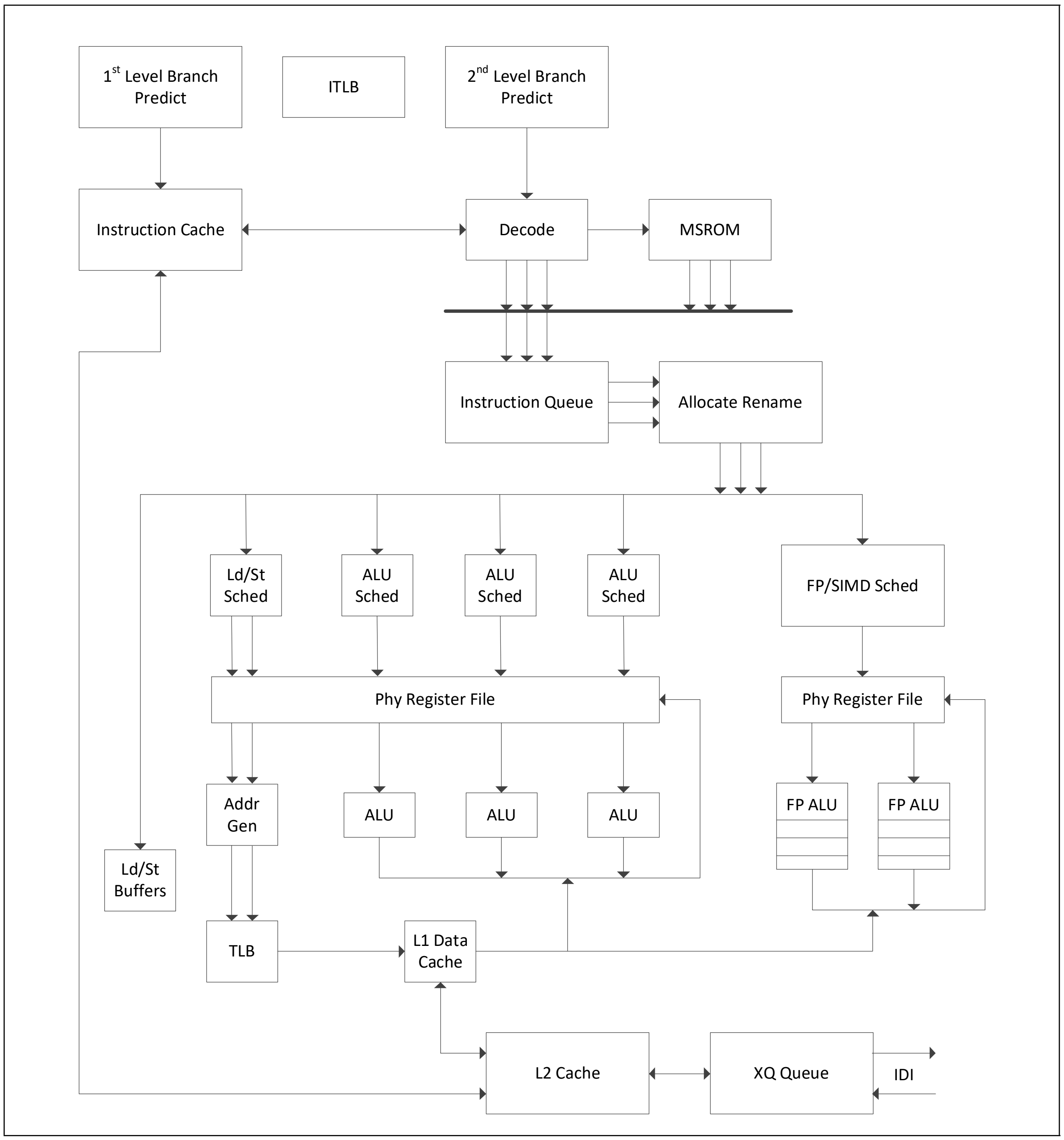


**Figure 6-3. CPU Core Pipeline Functionality of the Goldmont Microarchitecture**

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<br>• Out-of-order within each IEC RSV and between IEC, across FPC, MEC | • 2x distributed for each port<br>• Out-of-order within each IEC RSV and between IEC, across FPC, MEC |
| **FPC Reservation** | • 1x unified to ports 0, 1<br>• Out-of-order within FPC RSV and across IEC, MEC | • 2x distributed for each port<br>• In order within each FPC RSV; out-of-order between FPC, across IEC, MEC |
| **MEC Reservation** | • 1x unified to ports 0, 1<br>• Out-of-order within MEC RSV and across IEC, FPC | • 1x to port 0<br>• 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<br>• Shift/Rotate<br>• LEA with no index<br>• F2I<br>• converts/cmp, store_data | • ALU1<br>• Bit processing<br>• JEU<br>• IMUL<br>• IDIV<br>• POPCNT<br>• CRC32<br>• LEA<br>• I2F<br>• store_data | • ALU2<br>• LEA[1]<br>• I2F<br>• flag_merge |
| **FPC** | • SIMD ALU<br>• SIMD shift/Shuffle<br>• SIMD mul<br>• STTNI/AESNI/PCLMULQDQ/SHA<br>• FP_mul<br>• Converts<br>• F2I convert | • SIMD ALU<br>• SIMD shuffle<br>• FP_add<br>• 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 |

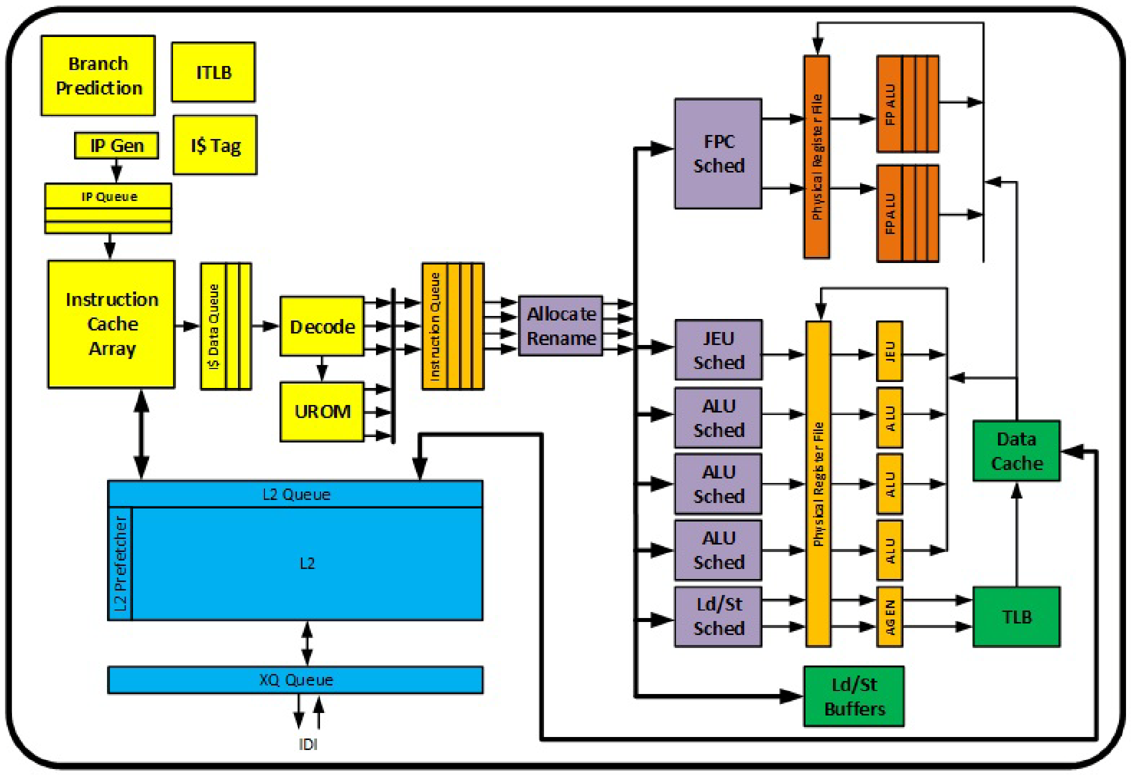
Table 6-7. Comparison of MEC Resources (Contd.)

| MEC Resource | Goldmont Microarchitecture | Silvermont Microarchitecture |
|---|---|---|
| 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<br>• Out-of-order within each IEC RSV and between IEC, across FPC, MEC | • 3x distributed for each port<br>• Out-of-order within each IEC RSV and between IEC, across FPC, MEC |
| **FPC Reservation** | • 1x unified to ports 0, 1<br>• Out-of-order within FPC RSV and across IEC, MEC | • 1x unified to ports 0, 1<br>• Out-of-order within FPC RSV and across IEC, MEC |
| **MEC Reservation** | • 1x unified to ports 0, 1<br>• Out-of-order within MEC RSV and across IEC, FPC | • 1x unified to ports 0, 1<br>• 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<br>• Shift/Rotate<br>• LEA with no index<br>• F2I<br>• converts/cmp<br>• store_data | • ALU1<br>• Bit processing<br>• IMUL<br>• IDIV<br>• POPCNT<br>• CRC32<br>• LEA<br>• I2F<br>• store_data | • ALU2<br>• LEA[1]<br>• I2F<br>• flag_merge | JEU |
| **FPC** | • SIMD ALU<br>• SIMD shift/Shuffle<br>• SIMD mul<br>• STTNI/AESNI/PCLMULQDQ/SHA<br>• FP_mul<br>• Converts<br>• F2I convert | • SIMD ALU<br>• SIMD shuffle<br>• FP_add<br>• 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 1Chapter 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<br>• (I)MUL r16<br>• r/m16 if extended precision not required, or<br>• (I)MUL r32<br>• r/m32 | Use<br>• (I)MUL r16<br>• r/m16 if extended precision not required, or<br>• (I)MUL r32<br>• r/m32 |

**Table 6-11. Alternatives to MSROM Instructions (Contd.)**

| Instruction from MSROM | Recommendation for Silvermont | Recommendation for Goldmont Plus and Goldmont |
|---|---|---|
| (I)MUL r/m32 (Result EDX:EAX) | Use<br>• (I)MUL r32<br>• r/m32 if extended precision not required, or<br>• (I)MUL r64<br>• r/m64 | Use as is (non MSROM) |
| (I)MUL r/m64 (Result RDX:RAX) | Use<br>• (I)MUL r64<br>• 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 three-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 andGoldmont 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 | EX 1 | EX 2 | EX 3 | EX 4 | EX 5 | | | | | | | | | | | | | |
| addps xmm1, xmm1 | | | | | | EX 1 | EX 2 | EX 3 | | | | | | | | | | |
| movaps mem, xmm1 | | | | | | | | | EX 1 | | | | | | | | | |
| mulps, xmm2, xmm2 | | | | | | | | | | EX 1 | EX 2 | EX 3 | EX 4 | EX 5 | | | | |
| addps xmm2, xmm2 | | | | | | | | | | | | | | | EX 1 | EX 2 | EX 3 | |
| movaps mem, xmm2 | | | | | | | | | | | | | | | | | | EX 1 |

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 1 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 0 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 1 to 3 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. See Table 15-14 for more details.

### 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 11 cycle latency and throughput of one every 11 cycles in the Silvermont microarchitecture. It has 5 cycle latency and throughput of one every 2 cycles in the Goldmont and later microarchitectures.

In general, SIMD integer multiply hardware is significantly faster (4 cycle latency) and higher throughput (1 cycle throughput) than in the Silvermont microarchitecture. Additionally, PADDQ/PSUBQ has 2 cycle latency and throughput of every 2 cycles, compared to 4 cycle latency and throughput every 4 cycles in the Silvermont microarchitecture. PSHUFB has 1 cycle latency and throughput in the Goldmont and later microarchitectures, compared to 5 cycle latency and throughput of every 5 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 six cycles with throughput of every four cycles in the Goldmont microarchitecture, compared to 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** | | | | | | |

**Table 6-14. Floating-Point and SIMD Integer Latency (Contd.)**

| | Goldmont Plus | | Goldmont | | Silvermont | |
|---|---|---|---|---|---|---|
| | Latency | Throughput | Latency | Throughput | Latency | Throughput |
| CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTTPD2DQ, CVTPD2PI, CVTPS2DQ | 4 | 1 | 4 | 1 | 5 | 2 |
| CVTPI2PS, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI | 4 | 1 | 4 | 1 | 4 | 1 |
| **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 six 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.

#### 6.8.2.17 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 | | | | | | | | | | | | |
| **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<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH,<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 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 |
| 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/XORPD/XORPS | 0.5 | 0.5 | 1 | 1 | N | N |
| AESDEC/AESDECLAST/AESENC/AESENCLAST | 2<br>1 (GLP) | 5 | 6<br>4 (GLP) | 8 | N | Y |
| AESIMC/AESKEYGEN | 2<br>1 (GLP) | 5 | 5<br>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 |

**Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)**

| Instruction | Throughput | | Latency | | MSROM | |
|---|---|---|---|---|---|---|
| | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH,<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH |
| 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 |
| 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<br>18 (GLP) | 27-69 | 13, 66<br>22 (GLP) | 27-69 | N | Y |
| DIVPS[1] | 12, 35<br>12 (GLP) | 27-39 | 13, 36<br>16 (GLP) | 27-39 | N | Y |
| DIVSD[1] | 12,33<br>10 (GLP) | 11-32 | 13, 34<br>14 (GLP) | 13-34 | N | N |
| DIVSS[1] | 12, 18<br>7 (GLP) | 11-17 | 13, 19<br>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 |

## 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 • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH, • 4DH • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH • 4DH • 5AH • 5DH |
| 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 |
| 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 |

### 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 • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH, • 4DH • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH • 4DH • 5AH • 5DH |
| 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 |
| 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 |

**Table 6-19. Instructions Latency and Throughput Recent Microarchitectures for Intel Atom® Processors (Contd.)**

| Instruction | Throughput | | Latency | | MSROM | |
|---|---|---|---|---|---|---|
| | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH,<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH |
| 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<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH,<br>• 4DH<br>• 5AH<br>• 5DH | • 06_5CH<br>• 5FH<br>• 7AH | • 06_37H<br>• 4AH<br>• 4CH<br>• 4DH<br>• 5AH<br>• 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<br>26 (GLP) | 70 | 68<br>30 (GLP) | 71 | N | Y |
| SQRTPS | 37<br>14 (GLP) | 40 | 38<br>18 (GLP) | 41 | N | Y |
| SQRTSD | 34<br>14 (GLP) | 35 | 35<br>18 (GLP) | 35 | N | Y |
| SQRTSS | 19<br>8 (GLP) | 20 | 20<br>12 (GLP) | 20 | N | Y |
| TEST r32, r32 | 0.33 | 0.5 | 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 • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH, • 4DH • 5AH • 5DH | • 06_5CH • 5FH • 7AH | • 06_37H • 4AH • 4CH • 4DH • 5AH • 5DH |
| 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)

# 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:

- **Section 7.1, "Overview"** — Provides an overview of issues related to instruction selection and scheduling.
- **Section 7.2, "Definitions"** — Presents definitions.
- **Section 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 performance issues:

- 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

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.

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.

- 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 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 |

## Table 7-1. CPUID Signature (Contd.)Values of Recent Intel Microarchitectures

| DisplayFamily_DisplayModel | Recent Intel Microarchitectures |
|---|---|
| 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)

| | DisplayFamily_DisplayModel | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **SIMD Instruction Extensions** | • 06_4EH<br>• 06_5EH | • 06_3DH<br>• 06_47H<br>• 06_56H | • 06_3CH<br>• 06_45H<br>• 06_46H<br>• 06_3FH | • 06_3AH<br>• 06_3EH | • 06_2AH<br>• 06_2DH | • 06_25H<br>• 06_2CH<br>• 06_2FH | • 06_1AH<br>• 06_1EH<br>• 06_1FH<br>• 06_2EH | • 06_17H<br>• 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<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 | ▪ 06_4E<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 |
| ADCX | 1 | 1 | 1 | 1 |
| ADOX | 1 | 1 | 1 | 1 |

**Table 7-3. BMI1, BMI2 and General Purpose Instructions (Contd.)**

| Instruction | Latency [1] | | Throughput | |
|---|---|---|---|---|
| **DisplayFamily_DisplayModel** | ▪ 06_4E<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 | ▪ 06_4E<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 |
| 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<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 | ▪ 06_3C<br>▪ 06_45<br>▪ 06_46<br>▪ 06_3F | ▪ 06_4E<br>▪ 06_5E | ▪ 06_3D<br>▪ 06_47<br>▪ 06_56 | ▪ 06_3C<br>▪ 06_45<br>▪ 06_46<br>▪ 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 |

### 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 |
| 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 |
| 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 |

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 |

**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 • 06_45 • 06_46 • 06_3F | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_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<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_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<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_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 |
| 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 |

**Table 7-8. 256-bit Intel® AVX 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_3A • 06_3E | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E |
| 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 |

**Table 7-8. 256-bit Intel® AVX 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_3A • 06_3E | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E |
| 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 |
| 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 • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_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 • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_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 • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_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 |
| 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<br>**c:** includes two 1-cycle bubbles due to bypass | | | | | | | | |

**Table 7-12. Intel® SSE3 Instructions**

| Instruction | Latency [1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| **DisplayFamily_DisplayModel** | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_2D | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_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<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_2D | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_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 |

**Table 7-13. Intel® SSE3 SIMD Floating-point Instructions (Contd.)**

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| DisplayFamily_DisplayModel | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_2D | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_2D |
| MOVSHDUP xmm1, xmm2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| MOVSLDUP xmm1, xmm2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 7-14. Intel® SIM SSE2 128-bit Integer Instructions**

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_2D | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E<br>• 06_2A<br>• 06_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 |

**Table 7-14. Intel® SIM SSE2 128-bit Integer Instructions (Contd.)**

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D |
| 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 |
| 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 |

### Table 7-14. Intel® SIM SSE2 128-bit Integer Instructions (Contd.)

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E • 06_2A • 06_2D |
| 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 • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_2A • 06_2D • (06_3A/3E) | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_2A • 06_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 |

**Table 7-15. Intel® SSE2 Double-Precision Floating-Point Instructions (Contd.)**

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_2A<br>• 06_2D<br>• (06_3A/3E) | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_2A<br>• 06_2D<br>• (06_3A/3E) |
| 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 |
| 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<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_2A<br>• 06_2D<br>• (06_3A/3E) | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_2A<br>• 06_2D<br>• (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 |
| 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) |

### Table 7-16. Intel® SSE Single-Precision Floating-Point Instructions (Contd.)

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_2A • 06_2D • (06_3 A/3E) | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_2A • 06_2D • (06_3 A/3E) |
| 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 7-17. General Purpose Instructions

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_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 |

**Table 7-17. General Purpose Instructions (Contd.)**

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| **CPUID** | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E | • 06_4E<br>• 06_5E | • 06_3D<br>• 06_47<br>• 06_56 | • 06_3C<br>• 06_45<br>• 06_46<br>• 06_3F | • 06_3A<br>• 06_3E |
| 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 |
| 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 |

Table 7-17. General Purpose Instructions (Contd.)

| Instruction | Latency[1] | | | | Throughput | | | |
|---|---|---|---|---|---|---|---|---|
| CPUID | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E | • 06_4E • 06_5E | • 06_3D • 06_47 • 06_56 | • 06_3C • 06_45 • 06_46 • 06_3F | • 06_3A • 06_3E |
| 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, "General Optimization Guidelines," 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.
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.

## 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.

### 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 |

# CHAPTER 8 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX) OPTIMIZATIONS

## 8.1 INTRODUCTION

Intel® Transactional Synchronization Extensions (Intel® TSX) aim to improve the performance of lock-protected critical sections while maintaining the lock-based programming model.

This chapter describes:

- The recommended approach for optimizing and tuning multi-threaded applications to use the Intel TSX instructions for lock elision.
- Provides guidelines focused on using the Intel TSX instructions to implement lock elision to enable concurrency of lock-protected critical sections, whether through
  - Prefix hints, as with Hardware Lock Elision (HLE)[1].
  - New instructions, as with RTM.

**Table 8-1. Additional Resources**

| Linked Title | Description |
|---|---|
| Intel® Transactional Synchronization Extension (Intel® TSX) Disable Update for Selected Processors | Provides details about Intel TSX behavior changes due to the updated microcode (including the behavior of these MSRs). |
| Intrinsics for Intel® Transactional Synchronization Extensions (Intel® TSX) | A section in the Intel® C++ Compiler Classic Developer Guide and Reference. |
| Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" | Details of the Intel TSX interface in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. |
| Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors | The latest support, product information, and documentation. |
| Intel® Transactional Synchronization Extensions (Intel® TSX) profiling with Linux* perf | Describes TSX profiling using the Linux perf) (or "perf events") profiler, that comes integrated with newer Linux systems. |

### 8.1.1 ABOUT INTEL® TSX

Intel TSX allows the processor to:

- Determine dynamically whether threads need to serialize through lock-protected critical sections.
- Perform serialization only when required.

1. Hardware Lock Elision (HLE) was deprecated in 2019.

- This lets hardware expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization through a technique known as **lock elision**.

With lock elision, the hardware executes the programmer-specified critical sections transactionally. These are called transactional regions. In this case, the lock variable is only read within the transactional region. It is neither written to nor acquired, except that the lock variable remains unchanged after the transactional region, thus exposing concurrency.

If the transactional execution completes successfully, the hardware ensures that all memory operations within the transactional region appear instantaneously when viewed from other logical processors. A processor makes architectural updates within the region visible to other logical processors only on a successful commit. This is a process called an atomic commit. Updates within the transactional region become visible to other logical processors only on an atomic commit.

Since a successful transactional execution ensures an atomic commit, the processor can optimistically execute the programmer-specified code section without synchronization. If synchronization is unnecessary, the execution can commit without any cross-thread serialization.

If the transactional execution fails, the processor cannot commit the updates atomically. When this happens, the processor will roll back the execution: called a transactional abort. During a transactional abort, the processor:

- Discards all updates performed in the region.
- Restores the architectural state, as if the optimistic execution never occurred.
- Resumes execution non-transactionally.

Depending on the policy, lock elision may be retried, or the lock may be explicitly acquired to ensure progress.

Intel TSX provides two software interfaces to programmers:

- **Hardware Lock Elision** (HLE) is a legacy-compatible instruction set extension (comprising the XACQUIRE and XRELEASE prefixes).
- **Restricted Transactional Memory** (RTM) is a new instruction set interface (comprising the XBEGIN and XEND instructions).

Programmers who want to run Intel TSX-enabled software on legacy hardware would use the HLE interface to implement lock elision. On the other hand, programmers who do not have legacy hardware requirements and who deal with more complex locking primitives would use the RTM interface of Intel TSX to implement lock elision. If the latter, when using new instructions, the programmer must:

- Provide a non-transactional path to execute after a transactional abort.
- Must not rely on the transactional execution alone.

This path should have code to acquire the lock being elided.

Intel TSX also provides the XTEST instruction to test if a logical processor executes transactionally and the XABORT instruction to abort a transactional region.

A processor performs a transactional abort for numerous reasons, but the main reason is conflicting data accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution.

- Memory addresses read from within a transactional region. They constitute:
- The read-set of the transactional region and addresses written to within the transactional region.

The write-set of the transactional region.

Intel TSX maintains the read- and write-sets at the granularity of a cache line. For lock elision using RTM, the address of the lock being elided must be added to the read-set to ensure the correct behavior of a transactionally executing thread in the presence of another thread that explicitly acquires the lock.

A conflicting data access occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. This is called a data conflict.

Since Intel TSX detects data conflicts at the cache line level, unrelated data locations in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity.

Some instructions (CPUID and IO, for example) may cause a transactional execution to abort in the implementation.

The term lock elision refers to either an HLE-based or an RTM-based implementation that elides locks.

### 8.1.2 OPTIMIZATION OUTLINE

The chapter covers:

- Application performance (See Section 8.2) improvement through Intel TSX rather than synthetic micro-kernels that tend to overlook how real applications behave after acquiring a lock.
- Enabling a synchronization library for lock elision using Intel TSX (See Section 8.3).
- Using the performance monitoring infrastructure for Intel TSX effectively (See Section 8.4) and present some performance guidelines for the first implementation (See Section 8.5).

The recommended guideline is to enable elision for all critical section locks and then identify problematic critical sections. Such a “bottoms-up” approach simplifies the evaluation and tuning of the resulting application and allows the programmer to focus on relevant critical sections.

## 8.2 APPLICATION-LEVEL TUNING AND OPTIMIZATIONS

Applications typically use **synchronization libraries** to implement the lock acquire and lock release functions associated with critical sections. The simplest way to enable these applications to take advantage of Intel TSX-based lock elision is to use an Intel TSX-enabled synchronization library. Existing libraries may be already enabled to take advantage of the Intel TSX instructions (see Section 8.2.1). If an off-the-shelf, TSX-enabled library is not yet available, Section 8.3 discusses how to extend a locking library to use the Intel TSX instructions if it has not already been enabled. TSX-enabled synchronization libraries can be interchangeably used with conventional synchronization libraries.

While applications using these libraries can use Intel TSX without application modification, some basic tuning and profiling can improve performance by increasing the commit rate of transactional execution and lowering the wasted execution cycles due to transactional aborts. The recommended first step for tuning is to use a profiling tool (see Section 16.4) to characterize the transactional behavior of the application. The profiling tool uses the performance monitoring and sampling capabilities implemented in the hardware to provide detailed information about the transactional behavior of the application. The tool uses capabilities the processor provides, such as performance monitoring counters and the Precise Event Based Sampling (PEBS) mechanism[1].

Applications using an Intel TSX-enabled synchronization library should have the same functional behavior as if using a conventional synchronization library. However, because Intel TSX changes latencies and can make cross-thread synchronization faster, latent code bugs may be exposed.

1. See Chapter 18of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

## 8.2.1 EXISTING TSX-ENABLED LOCKING LIBRARIES

This section summarizes off-the-shelf locking libraries that are already TSX-enabled for lock elision. The list represents a snapshot as of the first half of 2015. Not all libraries mentioned here may be completely tuned.

### 8.2.1.1 Libraries Allowing Lock Elision for Unmodified Programs

- On Linux, GNU glibc 2.18 added support for lock elision of pthread mutexes of PTHREAD_MUTEX_DEFAULT type. Glibc 2.19 added support for elision of read/write mutexes.
  - Whether elision is enabled depends on if the **--enable-lock-elision=yes** parameter was set at the compilation time of the library.
- Java JDK 8u20 or later supports adaptive elision for synchronized sections when the -XX:+UseRTMLocking option is enabled.
- Intel Composer XE 2013 SP1 or later supports lock elision for OpenMP **omp_lock_t**. Use "export KMP_LOCK_KIND=adaptive" to enable lock elision.

### 8.2.1.2 Libraries Requiring Program Modifications

- Intel Thread Building Blocks (TBB) 4.2 supports elision with the s**peculative_spin_rw_mutex**. The program needs to be modified to use this new lock type.
- gcc 4.8 and later supports TSX accelerating its software transactional memory implementation.
- Concurrency Kit supports lock elision of spinlocks with its **ck_elide** wrappers.
- DPDK library supports lock elision of spin locks and read-write locks (through lock/unlock calls with "**_tm**" suffix).

## 8.2.2 INITIAL CHECKS

A couple of simple sanity checks can save tuning effort later on; specifically, using a good library implementation and dealing with statistics collection inside critical sections.

- Use a good Intel TSX enabled synchronization library. The application should directly be using the TSX-enabled synchronization library. When the application implements its own custom library built on top of an Intel TSX-enabled library, it still may be missing opportunities to identify transactional regions. See Section 3 on how to enable the synchronization library for Intel TSX.
- Avoid collecting statistics inside critical sections. Critical sections (and sometimes the synchronization library itself) may employ shared global statistics counters. Such counters will cause data conflicts and transactional aborts. Applications often have flags to disable such statistics collection. Disabling such statistics in the initial tuning phase will help focus on inherent data conflicts.

## 8.2.3 RUN AND PROFILE THE APPLICATION

Visualizing synchronization-related thread interactions in multi-threaded applications is often difficult. The first step should be to run the application with an Intel TSX-enabled synchronization library and measure performance. Next, the profiling tool should be used to understand the result. First we should determine how much of the application is actually employing transactional execution, by using a profiling tool to measure the percentage of the application cycles spent in transactional execution (See Section 8.4).

Numerous causes may contribute to a low percentage of transactional execution cycles:

- The application may not be making noticeable use of critical-section based synchronization. In this case, lock elision is not going to provide benefits.

- The application's synchronization library may not use Intel TSX for all its primitives. This can occur if the application uses internal custom functions and libraries for some of the critical section locks. These lock implementations need to be identified and modified for elision (See Section 8.4.2).
- The application may be employing higher level locking constructs (referred to as meta-locks in this document) different from the one provided by the elision-enabled synchronization libraries. In these cases, the construct needs to be identified and enabled for elision (See Section 8.3.7)
- A program may be using LOCK-prefixed instructions for usages other than critical sections. TSX will not help with these typically, unless the algorithms are adapted to be transactional. Details on such non-locking usage are beyond the scope of this guide.

In the “bottom-up“ approach of Intel TSX performance tuning, the methodology can be modularized into the following tasks:

- Identify all locks.
- Run the unmodified program with a TSX synchronization library eliding all locks.
- Use a profiling tool to measure transactional execution.
- Address causes of transactional aborts if necessary.

## 8.2.4 MINIMIZE TRANSACTIONAL ABORTS

**Data conflicts** are detected through the cache coherence protocol. Data conflicts cause transactional aborts. In the initial implementation, the thread that detects the data conflict will transactionally abort.

If an HLE-based transactional execution experiences a transactional abort, then in the current implementation, the hardware will restart at the XACQUIRE prefixed instruction that initiated HLE execution but will ignore the XACQUIRE prefix. This results in the re-execution without lock elision and the lock is explicitly acquired. If an RTM-based transactional execution experiences a transactional abort, then in the current implementation, the hardware will restart at the instruction address provided by the operation of the XBEGIN instruction.

The initial TSX implementation supports a limited form of nesting. RTM supports a nesting level of 7. HLE supports a nesting level of 1. This is an implementation specific number that may change in subsequent implementations of the same generation of processor families.

The Chapter 16, “Programming with Intel® Transactional Synchronization Extensions” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 also describes the various causes for transactional aborts in detail. Details of Intel TSX instructions and prefixes can be found in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

The profiling tool can use performance monitoring to compute cycles that were spent in transactional execution that subsequently aborted. It is important to note that not all transactional aborts cause performance loss. The execution may otherwise have stalled due to waiting on a lock that had been acquired by another thread, and the transactional execution may also have a data prefetching effect.

The profiling tool can use PEBS to identify the top aborted transactional regions and provide information on the relative costs (see Section 8.4). We next discuss common causes for transactional aborts and provide mitigation strategies.

***Tuning Suggestion 9.*** *Use a profiling tool to identify the transactional aborts that contribute most to any performance loss.*

The broad categories for transactional abort causes include:

- Aborts due to conflicting data accesses.
- Aborts due to conflicts on the lock variable.

- Aborts due to exceeding resource buffering.
- Aborts due to HLE interface specific constraints.
- Miscellaneous aborts as described in Chapter 8 of the Intel® Architecture Instruction Set Extensions Programming Reference.

### 8.2.4.1 Transactional Aborts Due to Data Conflicts

A data conflict occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. In the initial implementation, data conflicts are detected through the cache coherence protocol that operates at the granularity of a cache line.

We now discuss various sources of data conflicts that can cause transactional aborts. Some are avoidable while others are inherently present in the application.

#### Conflicts due to False Sharing

False sharing occurs when unrelated variables map to the same cache line (64 bytes) and are independently written by different threads. In this case, although the addresses of the unrelated variables do not overlap, since the hardware checks data conflicts at cache-line granularity, these unrelated variables appear to have the same address and this causes unnecessary transactional aborts.

Note that negative effects of false sharing are not unique to Intel TSX. The cache coherence protocol is moving the cache line around the system with high overhead. Good software practice already recommends against placing unrelated variables on the same cache line when at least one of the variables is frequently written by different threads.

***Tuning Suggestion 10.*** *Add padding to put the two conflicting variables in separate cache line.*

***Tuning Suggestion 11.*** *Reorganize the data structure to minimize false sharing whenever possible.*

#### Conflicts due to True Sharing

These transactional aborts occur if the conflict data is actually shared and is not due to false sharing. Sometimes such conflicts can also be mitigated through software changes. We discuss how to address some of these conflicts next.

#### Conflicts due to Statistics Maintenance

Software may often use global statistics counters shared among multiple threads. Examples of such use include synchronization libraries that count the number of times a critical section lock is either successfully acquired or was found to be held. Other examples include a count in a global variable or in an object that is accessed by multiple threads. Such statistics contribute to transactional aborts. In such cases, one must first try to understand the use of such statistics.

Sometimes these statistics can be disabled or conditionally skipped as they do not affect program logic. For example, such statistics may be measuring the frequency of serialized execution of a critical section. Without lock elision, the statistic is updated inside the critical section as the execution is already serialized. However, if the lock has been elided, then counting the number of times the lock has been elided isn't particularly useful. The only time it matters is if the lock was not elided; in those situations, the software can use the statistics to track the level of serialization. The XTEST instruction can be used to update the statistics only when the execution is not eliding a lock (i.e., serialized). Sometimes these statistics are only useful during program development and can be disabled in production software.

In some cases these statistics cannot be disabled or skipped. The programmer can avoid unnecessary transactional aborts by maintaining these statistics per logical thread (while taking care to avoid false sharing). Such an approach requires results to be aggregated across all threads when read. This can also improve the performance of applications even without Intel TSX instructions by minimizing communication among various threads.

Other approaches include moving the statistic outside critical sections and using an atomic operation to update the statistic. This will reduce transactional aborts but may add additional overhead due to an additional atomic operation and will not reduce the communication overhead.

***Tuning Suggestion 12.*** *Global statistics may also be sampled rather than being updated for every operation.*

***Tuning Suggestion 13.*** *Avoid unnecessary statistics in critical sections.*

***Tuning Suggestion 14.*** *Consider maintaining statistics in critical sections on a per-thread basis.*

The programmer will have to determine the best approach for reducing transactional aborts due to shared global statistics. Disabling all global statistics during initial testing can help identify whether they are a problem.

#### Conflicts Due to Accounting in Data Structures

Another common source of data conflicts are accounting operations in data structures. For example, data structures may maintain a variable to track the number of entries present at any time. This has the same effect as a statistics counter and can cause unnecessary transactional aborts.

In some usages, it is possible to move the accounting update to outside the critical section using atomic updates (e.g., the number of entries to trigger heap reorganization).

In other scenarios, approaches may be adopted to reduce the window of time where data conflicts may occur (see Section 8.2.4.1 on Reducing the Window for Data Conflict).

#### Conflicts in Memory Allocators

Some critical sections perform memory allocations. It is recommended to use a thread-friendly memory allocation library that maintains its free list in thread local space and avoid false sharing of the allocated memory.

#### Conflict Reduction through Conditional Writes

A common software pattern involves updates to a shared variable or flag that only infrequently changes value. Such an operation (even with the same value) causes an update to the cache line, which may in turn result in the processor requesting write-permissions to the cache line. Such an operation will cause transactional aborts in other threads that are also accessing the shared variable. Software can avoid such data conflicts by performing the update only when necessary - not performing the store if the value doesn't change, see Example 8-1.

**Example 8-1. Reduce Data Conflict with Conditional Updates**

| | |
|---|---|
| state = true; // updates every time<br>var \|= flag; | if (state != true) state = true;<br>if (!(var & flag)) var \|= flag; |

#### Reducing the Window for Data Conflict

Sometimes the techniques described are insufficient to avoid transactional aborts due to frequent real data conflicts. In such cases, the goal should be to reduce the window of time where a data conflict can occur. To reduce this probability, one may move the actual conflicting memory access towards the end of the critical section.

### 8.2.4.2 Transactional Aborts Due to Limited Transactional Resources

While an Intel TSX implementation provides sufficient resources for executing common transactional regions, implementation constraints and excessive data footprint for transactional regions may cause a transactional abort.

The architecture provides neither a guarantee of the resources available for transactional execution nor that a transactional execution will ever succeed.

The processor tracks both the **read-set** addresses and the **write-set** addresses in the first level data cache (L1 cache) of the processor.

An eviction of a read set address may not always result in an immediate transactional abort since these lines may be tracked in an implementation-specific second level structure. In current implementations, the second level structure tracks evicted read-set addresses probabilistically. As a result, accesses from other threads may at times result in a false positive match thus causing an unnecessary transactional abort. The rate of such false conflicts is a function of the address stream from different threads and the precise hardware implementation. The Broadwell microarchitecture implementation has an improved second level structure. The rate of false conflicts is expected to reduce further with future implementations.

The architecture does not provide any guarantee for buffering and software must not assume any such guarantee.

With Haswell, Broadwell and Skylake microarchitectures, the L1 data cache has an associativity of 8. This means that in this implementation, a transactional execution that writes to 9 distinct locations mapping to the same cache set will abort. However, due to microarchitectural implementations, this does not mean that fewer accesses to the same set are guaranteed to never abort.

Additionally, in configurations with Intel Hyper-Threading Technology, the L1 cache is shared between the two logical processors on the same core, so operations in a sibling logical processor of the same core can cause evictions and significantly reduce the effective read and write set sizes.

Use the profiler to identify transactional regions that frequently abort due to capacity limitations (see Section 8.4.4). Software should avoid accessing excessive data within such transactional regions. Since, in general, accessing large amounts of data takes time, such aborts result in an excessive wasted execution cycles.

Sometimes, the data footprint of the critical section can be reduced by changing the algorithm. For example, for a sorted array, a binary instead of a linear search could be used to reduce the number of addresses accessed within the critical section.

If the algorithm expects certain code paths in the transactional region to access excessive data it may force an early transactional abort (through the XABORT instruction) or transition into a non-transactional execution without aborting by first acquiring the elided locks (see Section 8.2.6).

Sometimes, capacity aborts may occur due to side effects of actions inside a transactional region. For example, if an application invokes a dynamic library function for the first time the software system has to invoke the dynamic linker to resolve the symbols. If this first time happens inside a transactional region, it may result in excessive data being accessed, and thus will typically cause an abort. These types of aborts happen only the first time such a function is invoked. If this happens often, it is likely due to transactional only path not used in a non-transactional execution.

### 8.2.4.3 Lock Elision Specific Transactional Aborts

In addition to conflicts on data, transactional aborts may also occur due to conflicts on the lock itself. This is necessary to detect a transactional execution and a non-transactional execution of the critical section overlap in time. When implementing lock elision through Intel TSX, the implementation adds the lock to the read set - this occurs automatically for HLE but must be explicitly done in the software library when using RTM for lock elision. This allows checking conflicts with other threads that explicitly acquire the lock. This is a natural part of a transactional execution that aborts and re-starts and eventually acquires the lock.

For lock elision with HLE and RTM, many observed aborts occur due to such secondary conflicts on the lock variable: an aborting transactional thread transitions to a regular non-transactional execution, and as part of the transition also explicitly acquires the lock. This lock acquisition causes other transactionally executing threads to abort as they must serialize behind the thread that just acquired the lock.

For RTM, the fallback handler can potentially reduce these secondary aborts by waiting for the lock to be free before trying to acquire the lock (see Section 8.3.5).

#### 8.2.4.4 HLE Specific Transactional Aborts

Some transactional aborts only occur in HLE-based lock elision. They are described in subsequent sections.

##### Unsupported Lock Elision Patterns

For the transactional execution to commit successfully, the lock must satisfy certain properties and access to the lock must follow certain guidelines. An XRELEASE-prefixed instruction must restore the value of the elided lock to the value it had before the corresponding XACQUIRE-prefixed lock acquisition. This allows hardware to elide locks safely without adding them to the write-set. Both the data size and data address of the lock release (XRELEASE-prefixed) instruction must match that of the lock acquire (XACQUIRE-prefixed) and the lock must not cross a cache line boundary. For example, an XACQUIRE-prefixed lock acquire to an address A followed by an XRELEASE-prefixed lock release to a different address B will abort since the addresses A and B do not match.

##### Unsupported Access to Lock Variables inside HLE regions

Typically, a lock variable can be read from inside an HLE region without aborting. However, certain uncommon types of accesses may cause transactional aborts. For example, performing an unaligned access or a partially overlapping access to an elided lock variable will cause a transactional abort. Software should be changed to perform properly aligned accesses to the elided lock variable.

Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise it will cause a transactional abort.

#### 8.2.4.5 Miscellaneous Transactional Aborts

Programmers can use any instruction safely inside a transactional region and can use transactional regions at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path. Such transactional aborts will appear as Instruction Aborts in the PEBS record transactional abort status collected by the profiling tool (see Section 8.4).

The Intel SDM presents a comprehensive list of such instructions. Common examples include instructions that operate on the X87 and MMX architecture state, operations that update segment, control, and debug registers, IO instructions, and instructions that cause ring transitions, such as SYSENTER, SYSCALL, SYSEXIT, and SYSRET.

Programmers should use SSE/AVX instructions instead of X87/MMX instructions inside transactional regions. However, programmers must be careful when inter-mixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactional regions to abort. The VZEROUPPER instruction may also cause an abort, and programmers should try to move the instruction to prior to the critical section.

Certain 32-bit calling conventions may use X87 state to pass or return arguments. Programmers should consider alternate calling conventions or inline the functions. Some types such as long double may use X87 instructions and should be avoided.

In addition to the instruction-based considerations, various runtime events may cause transactional execution to abort.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. The rate of such aborts depends on the background state of the operating system. For example, operating systems with timer ticks generate interrupts that can cause transactional aborts.

Synchronous exception events (#BR, #PF, #DB, #BP/INT3, etc.) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred.

Page faults (#PF) typically occur most when a program starts up. Transactional regions will experience aborts at a higher rate during this period since pages are being mapped for the first time. These aborts will disappear as the program reaches a steady state behavior. However, for programs with very short run times, these aborts may appear to dominate. A similar behavior happens when large regions of memory were allocated in the recent past.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. These actions occur on the first access and write to the page, respectively. These operations will cause a transactional abort in the current implementation. A re-execution in non-transactional mode will cause these bits to be appropriately updated and subsequent transactional executions will typically not observe these transactional aborts. Although these transactional aborts will show up as Instruction Aborts in the PEBS record transactional abort status, special attention isn't needed unless they occur frequently.

In addition to the above, implementation-specific conditions and background system activity may cause transactional aborts. Examples include aborts as a result of the caching hierarchy of the system, subtle interactions with processor micro-architecture implementations, and interrupts from system timers among others. Aborts due to such activity are expected to be fairly infrequent for typical Intel TSX usage for lock elision.

***Tuning Suggestion 15.*** *Transactional regions during program startup may observe a higher abort rate than during steady state.*

***Tuning Suggestion 16.*** *Operating system services may cause infrequent transactional aborts due to background activity.*

### 8.2.5 USING TRANSACTIONAL-ONLY CODE PATHS

With Intel TSX, programmers can write code that is only ever executed in a transactional region and the non-transactional fallback path may be different. This is possible with RTM (through the use of the fallback handler) and with HLE in conjunction with the XTEST instruction.

Care is required if the code executed during transactional execution is significantly different than the code executed when not in transactional execution. Certain events such as page faults (instruction and data) and operations on pages that modify the accessed and dirty bits may repeatedly abort a transactional execution. Thus programmers must ensure such operations are also performed in a non-transactional fallback path, otherwise the transactional region may never succeed. This is not a problem in general since with lock elision the transactional path and non-transactional path in the application is the same and the only differences are captured in the synchronization libraries.

The XTEST instruction can be used to skip over code sequences that are unnecessary during transactional execution and likely to lead to aborts. The XTEST instruction can also be used to implement optimizations such as skipping unwind code and other error handling code (such as deadlock detection) that is only required if the lock is actually acquired.

***Tuning Suggestion 17.*** *Keep any transactional only code paths simple and inlined.*

***Tuning Suggestion 18.*** *Minimize code paths that are only executed transactionally.*

### 8.2.6 DEALING WITH TRANSACTIONAL REGIONS OR PATHS THAT ABORT AT A HIGH RATE

Some transactional regions abort at a high rate and the methods discussed so far are not effective in reducing the aborts. In such cases, the following options may be considered.

#### 8.2.6.1 Transitioning to Non-Elided Execution without Aborting

Sometimes, a transactional abort is unavoidable. Examples include system calls, and IO operations. When these are required on a transactional code path, software using RTM for lock elision can transition to a non-elided execution by attempting to acquire the lock and if successful committing the transactional execution. A simplified example is shown in Example 8-2. The actual code may need to handle nesting, etc.

**Example 8-2. Transition from Non-Elided Execution without Aborting**

```
/* … in RTM transaction, but the transactional execution will abort */
/* Acquire the lock without elision */

<original lock acquire code>
_xend();  /* Commit */

/* Do aborting operation */
```

#### 8.2.6.2 Forcing an Early Abort

Programmers should try to insert a PAUSE or XABORT instruction early in paths that lead to aborts inside transactional regions. This will force a transactional abort early and minimize work that needs to be discarded.

#### 8.2.6.3 Not Eliding Selected Locks

Sometimes if the application performance is lower with lock elision and the transactional abort reduction techniques have been exhausted, software can disable elision for the specific locks that have high and expensive transactional abort rates. This should always be validated with application level performance metrics, as even high abort rates may still result in a performance improvement.

## 8.3 DEVELOPING AN INTEL TSX-ENABLED SYNCHRONIZATION LIBRARY

This section describes how to enable a synchronization library for lock elision using the Intel TSX instructions.

### 8.3.1 ADDING HLE PREFIXES

The programmer uses the XACQUIRE prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The programmer uses the XRELEASE prefix in front of the instruction that is used to release the lock protecting the critical section. This instruction will be a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock, enabling concurrency in the absence of data conflicts.

### 8.3.2 ELISION FRIENDLY CRITICAL SECTION LOCKS

The library itself shouldn't be a source of data conflicts. Common examples of such problems include:

- Conflicts on the lock owner field.
- Conflicts on lock-related statistics.

When using HLE for lock elision, programmers must add the elision capability to the existing code path (since the code path executed with and without elision is the same with HLE). The programmer should also check that the only write operation to a shared location is through the lock-acquire/lock-release instructions on the lock variable. Any other write operation to a shared location would typically manifest itself as a data conflict among two threads using the elision library to elide a common lock. A test running multiple threads looping through an empty critical section protected by a shared lock can quickly identify such situations.

### 8.3.3 USING HLE OR RTM FOR LOCK ELISION

Software can use the CPUID information to determine whether the processor supports the HLE and RTM extensions. However, software can use the HLE prefixes (XACQUIRE and XRELEASE) without checking whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution. In contrast, software must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM. The XTEST instruction also requires a CPUID check to ensure either HLE or RTM is supported, else it will also generate a #UD exception. The CPUID information may be cached in some variable to avoid checking for CPUID repeatedly.

With HLE, if the eliding processor itself reads the value of the lock in the critical section, the value returned will appear as if the processor had acquired the lock; the read will return the non-elided value. This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The RTM interface allows programmers to write more complex synchronization algorithms and to control the retry policies following transactional aborts. The preferred way is to use the RTM-based locking implementation as a wrapper with multiple code paths within; one path exercising the RTM-based lock and the other exercising the non-RTM based lock (See Section 8.3.4). This typically does not require changes to the non-RTM based lock code. Performance may further be improved by using a try-once primitive, which allows the thread to re-attempt lock elision after the lock becomes free.

Since the RTM instructions do not have any explicit lock associated with the instructions, software using these instructions for lock elision must test the lock within the transactional region, and only if free should it continue executing transactionally. Further, the software may also define a policy to retry if the lock is not free.

In a subtle difference with HLE, if the code within the RTM-based critical section reads the lock, it will appear as if it is free and not acquired. So library functions used to return the value of locks must abort the transactional execution and return the value when executed non-transactionally (See Section 8.3.9). This situation does not exist with HLE because the HLE instructions have an explicit lock address associated with them and the hardware ensures the right value is returned.

***User/Source Coding Rule 1.*** *When using RTM for implementing lock elision, always test for lock inside the transactional region.*

***Tuning Suggestion 19.*** *Don't use an RTM wrapper if the lock variable is not readable in the wrapper.*

## 8.3.4 AN EXAMPLE WRAPPER FOR LOCK ELISION USING RTM

This section describes how to write a wrapper to implement lock elision using RTM instructions. The idea is to take the conventional lock implementation (without elision), add a wrapper around it, and then add a new path within the wrapper to implement elision. Thus, the wrapper provides separate code paths for the elided path and the non-elided paths. The non-elided lock-acquire path is executed only if the elided path was unsuccessful. Further, such an approach allows the non-elided path to remain unchanged. Such an approach works well for wide variety of locks, including ticket locks and read-write locks.

An example code sequence is shown in Example 8-3 (See Section 8.7 for a description of the intrinsics used).

### Example 8-3. Exemplary Wrapper Using RTM for Lock/Unlock Primitives

```
void rtm_wrapped_lock(lock) {
  if (_xbegin() == _XBEGIN_STARTED) {
    if (lock is free)
        /* add lock to the read-set */
        return; /* Execute transactionally */
    _xabort(0xff);
    /* 0xff means the lock was not free */
  }
  /* come here following the transactional abort */
  original_locking_code(lock);
}

void rtm_wrapped_unlock(lock) {
  /* If lock is free, assume that the lock was elided */
  if (lock is free)
    _xend();   /* commit */
  else
    original_unlocking_code(lock);
}
```

In Example 8-3, _xabort() terminates the transactional execution if the lock was not free. One can use _xend() to achieve the same effect. However, the profiling tool can easily recognize the _xabort() operation along with the 0xff abort code (which is a software convention) and determine that this is the case where the lock was unavailable. If the _xend() were used, the profiling tool would be unable to distinguish this case from the case where a lock was successfully elided.

The example above is a simplified version showing a basic policy of retrying only once and not distinguishing between various causes for transactional aborts. A more sophisticated implementation may add heuristics to determine whether to try elision on a per-lock basis based on information about the causes of transactional aborts. It may also have code to switch back to re-attempting lock elision after blocking if the lock was not free. This may require small changes to the underlying synchronization library.

Sometimes programming errors can lead to a thread releasing a lock that is already free. This error may not manifest itself immediately. However, when such a lock release function is replaced with an
RTM-enabled library using the wrapper described above, an XEND instruction will execute outside a transactional region. In this case, the hardware will signal a #GP exception. It is generally a good idea to fix the error in the original application. Alternatively, if the software wants to retain the original erroneous code path, then a XTEST can be used to guard the XEND.

## 8.3.5 GUIDELINES FOR THE RTM FALLBACK HANDLER

The fallback handler for RTM provides the code path that is executed if the RTM-based transactional execution is unsuccessful. Since the Intel TSX architecture specification does not provide any guarantee that a transactional execution will ever succeed, the RTM fallback handler must have the capability to ensure forward progress; it should not simply keep retrying the transactional execution.

***Tuning Suggestion 20.*** *When RTM is used for lock elision, forward progress is easily ensured by acquiring the lock.*

If the fallback handler explicitly acquires the lock, then all other transactionally executing threads eliding the same lock will abort and the execution serializes on the lock. This is achieved by ensuring that the lock is in the transactional region's read-set.

Software can use the abort information provided in the EAX register to develop heuristics as to when to retry the transactional execution and when to fallback and explicitly acquire the lock. For example, if the _XABORT_RETRY bit is clear, then retrying the transactional execution is likely to result in another abort. The fallback handler should distinguish this situation from cases where the lock was not free (for example, the _XABORT_EXPLICIT bit is set but the _XABORT_CODE()[1] returns a 0xff identifying the condition as a "lock busy" condition). In those cases, the fallback handler should eventually retry after waiting.

Performance may also be improved by retrying (after a delay) if the abort cause was a data conflict (_XABORT_CONFLICT) because such conditions are often transient. Such retries however should be limited and must not continually retry.

A very small number of retries for capacity aborts (_XABORT_CAPACITY) can be beneficial on configurations with Hyper Threading enabled. The L1 cache is a shared resource between HT threads and one thread may push data out of the other. On retry there is a reasonable chance to succeed. This requires ignoring the _XABORT_RETRY bit in the status code for this case. The _XABORT_RETRY bit should not be ignored for any other reason.

Generally on higher core count and multi-socket systems the number of retries should be increased.

In general, if the lock was not free, then the fallback handler should wait until the lock is free prior to retrying the transactional execution. This helps to avoid situations where the execution may persistently stay in a non-transactional execution without lock elision. This can happen because the fallback handler never had an opportunity to try a transactional execution while the lock was free (See Section 8.3.8).

***User/Source Coding Rule 2.*** *RTM abort handlers must provide a valid tested non transactional fallback path.*

***Tuning Suggestion 21.*** *Lock Busy retries should wait for the lock to become free again.*

1. _XABORT_CODE accesses the xabort status in the RTM abort code

## 8.3.6 IMPLEMENTING ELISION-FRIENDLY LOCKS USING INTEL® TSX

This section discusses strategies for implementing elision friendly versions of common locking algorithms using the Intel TSX instructions. Similar approaches can be adopted for algorithms not covered in this section.

### 8.3.6.1 Implementing a Simple Spinlock Using HLE

A spinlock is a simple yet very common locking algorithm. In this algorithm, a thread first checks to see if the lock is free and then attempts to acquire the lock through a LOCK-prefixed instruction. If not, the thread spins (using a read operation that typically completes from the local data cache holding the lock value) on the lock waiting for it to become free.

For this example, assume the lock is free when its value is zero, and held by some thread otherwise. The lock is released through a regular store instruction.

Example 8-4 uses the gcc 4.8+ **atomic intrinsics** which are similar to the C11 standard. The description here follows the recommended approach to implement a spin lock using gcc 4.8+ intrinsics. To enable HLE for this spin lock, the only change required would be the addition of the __ATOMIC_HLE_ACQUIRE and __ATOMIC_HLE_RELEASE flags. The rest of the code is the same as without using HLE.

#### Example 8-4. Spin Lock Example Using HLE in GCC 4.8 and Later

```
#include <immintrin.h> /* For _mm_pause() */
/* Lock initialized with 0 initially */
void hle_spin_lock(int *lock)
{
   while (__atomic_exchange_n(lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE) != 0)
   { int val;
      /* Wait for lock to become free again before retrying. */
      do {
         _mm_pause();  /* Abort speculation */
         __atomic_load_n(lock,  &val, __ATOMIC_CONSUME);
      } while (val == 1);
   }
}

void hle_spin_unlock(int *lock)
{
   __atomic_clear(lock, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
}
```

The following shows the same example using intrinsics for the Windows C/C++ compilers (Microsoft Visual Studio 2012 and Intel C++ Compiler 17.0).

**Example 8-5. Spin Lock Example Using HLE in Intel and Microsoft Compiler Intrinsic**

```
#include <intrin.h> /* For _mm_pause() */
#include <imminitrin.h> /* For HLE intrinsics */
/* Lock initialized with 0 initially */
void hle_spin_lock(int *lock)
{
while (_InterlockedCompareExchange_HLEAcquire(&lock, 1, 0) != 0){
        /* Wait for lock to become free again before retrying speculation */
        do {
            _mm_pause();  /* Abort speculation */
            /* prevent compiler instruction reordering and wait-loop skipping,
              no additional fence instructions are generated on IA */
            _ReadWriteBarrier();
        } while (lock == 1);
}
}

void hle_spin_unlock(int *lock)
{
_Store_HLERelease (lock, 0);
}
```

See Section 8.7 for an assembler implementation of an HLE spinlock.

### 8.3.6.2 Implementing Reader-Writer Locks Using Intel® TSX

Reader-Writer locks are common where the critical sections are mostly read-only. Such locks can avoid serializing access to the critical section for readers; however, they still require an atomic operation on a shared location (often through a LOCK prefixed XADD or CMPXCHG) and require communication among the multiple readers. Note that lock elision essentially makes all locks behave as reader-writer locks - except that, with lock elision readers and non-conflicting writers can proceed concurrently without communication.

RTM can be used to elide reader-writer locks through a wrapper approach as discussed earlier. The only difference being that, with reader-writer locks, the lock algorithm normally checks both the reader and the writer states to determine that the lock is free. When it is possible to place the reader and writer locking state on different cache lines, it is also possible to let transactional and non-transactional readers execute in parallel. The readers only need to check the writer state being free.

With HLE, the code path for the elided version and non-elided version should remain the same. Some reader-writer lock implementations use a lock to protect the reader/writer state instead of the actual critical section. In this case, the lock first needs to be changed to have a fast path with a single atomic operation. Beyond this, the path should not change the cache line with the lock variable. This can be done by combining the reader and writer counts into a single field, and then checking/updating it atomically with a LOCK- prefixed XADD or CMPXCHG instruction for the lock acquire and lock release functions. The HLE prefixes - XACQUIRE and XRELEASE - are placed on these LOCK-prefixed operations. Interestingly, this approach also improves the performance of reader-writer locks even without using Intel TSX. Alternatively, using an RTM wrapper can avoid changing lock structure since you can have different lock acquire paths for elided and non-elided versions in the synchronization library.

***Tuning Suggestion 22.*** *For Read/Write locks elide the complete lock operation, not the building block locks.*

#### 8.3.6.3 Implementing Ticket Locks Using Intel® TSX

Ticket locks are another common algorithm. A ticket lock is a variant of a spinlock where instead of spinning on a shared location and then racing to acquire the lock when the lock is free, threads use tickets to determine which thread can enter the critical section.

RTM can be used to elide ticket locks through a wrapper approach as discussed earlier (See Section 8.3.4).

Some ticket lock implementations assume an increasing ticket value and such locks do not meet HLE's requirement that the value of the lock following the lock release be the same as the value prior to the lock acquire.

***Tuning Suggestion 23.*** *Use RTM to elide ticket locks.*

#### 8.3.6.4 Implementing Queue-Based Locks Using Intel® TSX

In general, the idea of lock elision requires multiple threads to concurrently enter and try to commit a common critical section. The idea of fair locks requires threads to enter and release the critical section in a first-come first-served order. The two ideas may sometimes appear at odds, but the general objective is usually more flexible.

Queue-based locks are a form of fair locks where the threads construct a queue of lock requests. This includes different forms of ticket locks.

In some implementations the queue is formed through an initial LOCK-prefixed operation. For such implementations, the HLE XACQUIRE prefix can be added to this operation to enable lock elision. In the absence of any transactional aborts, the queue remains empty following the lock release. However, if a transactional abort occurs and the aborting thread acquires the lock explicitly (thus forming a queue), subsequent threads will add themselves to the queue, and when the lock is released, only a single thread will attempt lock elision as the other threads are not at the front of the queue. Further, if another thread arrives and adds itself to the queue, this may cause the transactionally executing thread to abort, and the execution remains in a non-eliding phase until the queue is drained.

This scenario only occurs with lock implementations that attempt lock elision as part of the queuing process. It does not apply to implementations that construct a queue only after an initial atomic operation, like an adaptive spinning-sleeping lock that elides the spinning phase but only queues for waiting after initial spinning failed. Such a problem also doesn't exist for implementations that use wrappers (such as those using RTM). In these implementations, the thread does not attempt lock elision as part of the queuing process.

***Tuning Suggestion 24.*** *Use an RTM wrapper for locks that implement queuing as part of the initial atomic operation.*

### 8.3.7 ELIDING APPLICATION-SPECIFIC META-LOCKS USING INTEL® TSX

Some applications build their own locks, called meta-locks, using an underlying synchronization library. In this approach, the application uses a lock from the underlying synchronization library to protect the data of the meta-lock. It then updates the data and releases the lock. If you recall, a similar approach was taken for the reader-writer lock implementation discussed in Section 8.3.6.2.

The application executes the critical section while holding the meta-lock, and then uses a lock from the underlying synchronization library to protect the meta-lock while it is being released. In this sequence, eliding the lock from the underlying synchronization library isn't useful; the goal should be to elide the meta-lock itself and transactionally execute the application code itself instead of the code in the synchronization library. A profiling tool can be used to identify such critical sections. An RTM wrapper (similar to one discussed in Section 8.3.4) can be used to avoid the meta-lock during lock elision.

For illustration, assume the following as an example of a meta-lock implementation.

**Example 8-6. A Meta Lock Example**

```
void meta_lock(Metalock *metalock) {
  __lock(metalock->lock);
  /* modify meta lock state for lock */
    unlock(metalock->lock);
}

void meta_unlock(Metalock *metalock) {
  lock(metalock->lock);
  /* drop metalock state */
  unlock(metalock->lock);
}

meta_lock(metalock);
  /* critical section */
meta_unlock(metalock);
```

The above example can be transformed into the following code.

**Example 8-7. A Meta Lock Example Using RTM**

```
void rtm_meta_lock(Metalock *metalock) {
    if (_xbegin() == _XBEGIN_STARTED)
          if (meta_state_is_all_free(metalock))
             return;
          _xabort(0xff);
    }
    meta_lock(metalock);
}
void rtm_meta_unlock(Metalock *metalock) {
    if (meta_state_is_all_free(metalock))
       _xend();
    else
        meta_unlock(metalock);
}

rtm_meta_lock(metalock);
/* critical section */
rtm_meta_unlock(metalock);
```

***Tuning Suggestion 25.*** *For meta-locking elide the full outer lock, not the building block locks.*

## 8.3.8 AVOIDING PERSISTENT NON-ELIDED EXECUTION

A transactional abort eventually results in execution transitioning to a non-transactional state without lock elision. This ensures forward progress. However, under certain conditions and with some lock acquire algorithms, threads may remain in a persistent non-transactional execution without attempting lock elision for an extended duration. This will limit performance opportunities.

To understand such situations, consider the following example with a simple spin lock implementation using HLE (a similar scenario can also exist with RTM). The lock value of zero means the lock is free and a value of one means it is acquired by some thread.

The HLE-enabled lock-acquire sequence can be written as shown in Example 8-8.

### Example 8-8. HLE-Enabled Lock-Acquire/ Lock-Release Sequence

```
  mov eax,$1
Retry:
  XACQUIRE; xchg LockWord,eax
  cmp eax,$0      # Was zero so lock was acquired successfully
  jz Locked
SpinWait:
  cmp LockWord, $1
  jz SpinWait# Still one
  jmp Retry# It's free, try to claim
Locked:

  XRELEASE; mov LockWord,$0
```

If a thread is unable to perform lock elision, then it acquires the lock without elision. Assume another thread arrives to acquire the lock. It executes the “XACQUIRE; xchg lockWord, eax” instruction, elides the lock operation on the lock, and enters transactional execution. However the lock at this point was held by another thread causing this thread to enter the SpinWait loop while still executing transactionally. This spin occurs during transactional execution because hardware does not have the notion of a critical section lock - it only sees the instruction to implement the atomic operation on the lock variable. The hardware doesn't have the semantic knowledge that the lock was not free.

Now, if the thread that held the lock releases it, the write operation to the lock will cause the current transactional thread spinning on the location to transactionally abort (because of the conflict between the lock release operation and the read loop of the lock by the transactional thread). Once it has aborted, the thread will restart execution without lock elision. It is easy to see how this extends to all other threads - they spin transactionally but end up executing non-transactionally and without lock elision when they actually find the lock free. This will continue until no other threads are trying to acquire the lock. The threads have thus entered a persistent non-elided execution.

A simple fix for this includes using the pause instruction (which causes an abort) as part of the spin-wait loop. This is also the recommended approach to waiting on a lock to be released, even without Intel TSX. The pause instruction will force the spin-wait loop to occur non-transactionally, thus allowing the threads to try lock elision when the lock is released.

### Example 8-9. A Spin Wait Example Using HLE

```
        mov eax,$1
Retry:
        XACQUIRE; xchg LockWord,eax
        cmp eax,$0# Was zero so we got it
        jz Locked
SpinWait:
        pause

cmp LockWord, $1
        jz SpinWait# Still one
        jmp Retry# It's free, try to claim
Locked:
```

***Tuning Suggestion 26.*** *Always include a pause instruction in the wait loop of a HLE spinlock.*

## 8.3.9 READING THE VALUE OF AN ELIDED LOCK IN RTM-BASED LIBRARIES

Some synchronization libraries provide interfaces that read the value of a lock. Libraries implementing lock elision using RTM may be unable to reliably determine if the lock variable has been acquired by the thread performing the elision since the lock was only read but not written to inside the library.

Sometimes the library interface may be as simple as a test to check whether a lock is acquired thus providing a sanity check to the software. To ensure the correct value is provided to the function using an RTM-based library, the transactional execution must be aborted and the lock explicitly acquired. This can be achieved by forcing an abort through the XABORT instruction (using _xabort(0xfe)). The 0xfe code can be used by the fallback handler to determine this situation and aid in optimizations in eliminating such a read. Alternatively, the _xtest() intrinsic can be used avoid unnecessary transactional aborts:

**assert(is_locked(my_lock)) => assert(_xtest() || is_locked(my_lock))**

A better primitive for an elided synchronization library would combine both - the lock being acquired or a lock elision in progress. For example:

**bool is_atomic(lock) { return _xtest() || is_locked(lock); }**

At other times, the lock variable may be read as part of a function with assumptions about behavior. An example is the **try-lock** interface to acquire a lock where a thread makes a single attempt to acquire the lock and returns a value indicating whether the lock was free or not. This is in contrast to a spin lock that continues to spin trying to acquire the lock. In general, this isn't a problem. But sometimes, software may make implicit assumptions about the actual value returned by a nested try-lock. With an RTM-based implementation, the value returned will be that of a free lock since the lock was elided. If software is making such implicit assumptions about the value, then the synchronization library can force a transactional abort through the XABORT instruction (using _xabort(0xfd)). This will however cause unnecessary aborts in some programs. Such implicit programming assumptions are not recommended. As such implicit programming assumptions are rare, it is recommended to not abort in the synchronization library in trylock.

## 8.3.10 INTERMIXING HLE AND RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. The behavior when HLE and RTM are nested together-HLE inside RTM or RTM inside HLE-is implementation specific. For the first implementation of the 4th generation Intel Core Processor, intermixing causes a transactional abort. This behavior may change in subsequent processor implementations but the semantics of a transactional commit will be maintained.

In general, applications should avoid intermixing HLE and RTM as they are essentially achieving the end purpose of lock elision but through different software interfaces. However, library functions implementing lock elision may be unaware of the calling function and whether the calling function is invoking the library function while eliding locks using RTM or HLE.

Software can handle such conditions by using the _xtest() operation. For example, the library may check if it was invoked within a transactional region and if the lock is free. If the call was within a transactional region, the library may avoid starting a new transactional region. If the lock was not free, the library may return an indication through the _xabort(0xff) function. This does require the function that will be invoked on a release to recognize that the acquire operation was skipped.

Example 8-10 shows a conceptual sequence.

**Example 8-10. A Conceptual Example of Intermixed HLE and RTM**

```
// Lock Acquire sequence
// Use a function local or per-thread location
bool lock_in_transactional_region = false;
if (_xtest() && my lock is free) { /* Already in a transactional region*/
  lock_in_transactional_region = true;
} else {
  // acquire lock if free, else abort
}

// the lock release sequence
if (!lock_in_transactional_region) {
    // release lock
}
```

## 8.4 USING THE PERFORMANCE MONITORING SUPPORT FOR INTEL® TSX

Application tuning using Intel TSX relies on performance counter-based profiling to understand transactional execution behavior and the causes of transactional aborts. Achieving good performance with Intel TSX often requires some tuning based on data from a profiling tool to minimize aborts. Using the performance counters is often preferable to instrumenting the application as it is usually less intrusive and easier.

Chapter 19, "Architectural Last Branch Records" of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B provides information about performance monitoring.

In general, profiling can impact transactional execution as any profiling tool generates periodic interrupts to collect information, and the interrupt will cause a transactional abort. Hence, any profiling should try to minimize the impact of this in analysis. This is not an issue if one is profiling only transactional aborts.

Program startup tends to have a large number of events that occur only once. When profiling complex programs, skipping over the startup phase can significantly reduce any noise introduced by these events.

Profilers that support TSX tuning include Linux perf, Intel Performance Counter Monitor, and Intel VTune. See Intel's Instruction Set Architecture landing page for many additional resources.

### 8.4.1 MEASURING TRANSACTIONAL SUCCESS

The first step should be to measure the transactional success in an application. This is done with the **Unhalted_Core_Cycles** event programmed in three separate configurations with three counters:

1. Use the fixed cycles counter (**IA32_FIXED_CTR0**) to measure FixedCyclesCounter.
2. Configure **IA32_PERFEVTSEL2** with the **IN_TX** and **IN_TXCP** filters set to measure CyclesInTxCP in **IA32_PMC2**.
3. Configure another **MSR IA32_PERFEVTSELx** (x= 0, 1, 3) with IN_TX filter to measure CyclesInTXOnly on the corresponding counter.

These cycle measurements should be set up to count and not sample frequently; sampling may cause additional transactional aborts. With these three values the total cycles, cycles spent in transactional execution, and cycles spent in transactional regions that eventually aborted can be computed:

**CyclesTotal = FixedCycleCounter**
**%CyclesTransactionalAborted = ((CyclesInTxOnly - CyclesInTxCP) / CyclesTotal) * 100.0**
**%CyclesTransactional = (CyclesInTx / CyclesTotal) * 100.0**
**%CyclesNonTransactional = 100.0 - %CyclesTransactional**

If CyclesTransactional is near zero then the application is either not using lock-based synchronization or not using a synchronization library enabled for lock elision through the Intel TSX instructions. In the latter case, the programmer should use an Intel TSX-enabled synchronization library (See Section 8.3).

If CyclesTransactionalAborted is small relative to CyclesTransactional, then the transactional success rate is high and additional tuning is not required.

If the CyclesTransactionalAborted is almost the same as CyclesTransactional (but not very small), then most transactional regions are aborting and lock elision is not going to be beneficial. The next step would be to identify the causes for transactional aborts and reduce them (See Section 8.2.4).

### 8.4.2 FINDING LOCKS TO ELIDE AND VERIFYING ALL LOCKS ARE ELIDED.

This step is useful if the cycles spent in transactional execution is low. This may be because few locks are being elided. The MEM_UOPS_RETIRED.LOCK_LOADS event should be counted and compared to the RTM_RETIRED.START or HLE_RETIRED.START events. If the number of lock loads is significantly higher than the number of transactional regions started, then one can usually assume that not all locks are marked for lock elision. The PEBS version of MEM_UOPS_RETIRED.LOCK_LOADS can be sampled to identify the missing locks. However, this technique isn't effective in immediately detecting missed opportunities with meta-locking (See Section 8.3.7). Additionally, a profile on the call graph of the MEM_UOPS_RETIRED.LOCK_LOADS event often identifies the high level synchronization library that needs to be TSX-enabled to allow transactional execution of the application level critical sections.

### 8.4.3 SAMPLING TRANSACTIONAL ABORTS

The hardware implementation defines PEBS precise events to sample transactional aborts **- HLE_RETIRED.ABORTED** for HLE and **RTM_RETIRED.ABORTED** for RTM. This allows programmers to perform precise profiling of all transactional aborts in the execution. The test should be run with PEBS enabled and sampled to identify the code location where the transactional aborts are occurring. The PEBS handler (a part of the profiling tool) uses the EventingIP field in the PEBS record to report the correct code location of the transactional aborts.

As a next step, the most common transactional aborts should be examined and addressed. Sampling transactional aborts does not cause any additional aborts.

### 8.4.4 CLASSIFYING ABORTS USING A PROFILING TOOL

The PEBS record generated as a result of profiling transactional aborts contains additional information on the cause of the transactional abort in the TX Abort Information field. The lower 32 bits of the TX Abort Information, called Cycles_Last_TX, also provides the cycles spent in the last transactional region prior to the abort. This approximately captures the cost of a transactional abort.

**RelativeCostOfAbortForIP = SUM(Cycles_Last_TX_For_IP)**

Not all transactional aborts are equal - some don't contribute to performance degradation while the more expensive ones can have significant impact. The programmer can use this information to decide which transactional aborts to focus on first.

- For more details on the PEBS record see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B Section 18.10.5.1

The profiling tool should display the abort cost to the user to classify the abort.

***Tuning Suggestion 27.*** *The aborts with the highest cost should be examined first.*

***Tuning Suggestion 28.*** *The TX Abort Information has additional information about the transactional abort.*

If the PEBS record **Instruction_Abort** bit (bit 34) is set, then the cause of the transactional abort can be directly associated with an instruction. For these aborts, the PEBS record captures the instruction address that was the source of the transactional abort. Exceptions, like page faults (including those that would normally terminate the program and those that fault in the working set of the program at startup) also show up as in this category.

If the PEBS record **Non_Instruction_Abort** bit (bit 35) is set, then the abort may not have been caused by the instruction reported by the instruction address in the PEBS record. An example of such an abort is one due to a data conflict with other threads. In this case, the **Data_Conflict** bit (bit 37) is also set. Another example is when transactional aborts occur due to capacity limitations for transactional write- and read-sets. This is captured by the Capacity_Write (bit 38) and the Capacity_Read (bit 39) fields.

Aborts due to data conflicts may occur at arbitrary instructions within the transactional region. Hence it is useful to concentrate on conflict causes in the whole critical section. Instead of relying on the **EventingIP** reported by PEBS for the abort, one should focus on the return IP (IP of the abort code) in conjunction with the call graphs. The return IP typically points into the synchronization library, unless the lock is inlined. The caller identifies the critical section.

For capacity it can be also useful to concentrate on the whole critical section (profiling for ReturnIP) as the whole critical section needs to be changed to access less memory.

***Tuning Suggestion 29.*** *Instruction aborts should be analyzed early, but only when they are costly and happen after program startup.*

***Tuning Suggestion 30.*** *For data conflicts or capacity aborts, concentrate on the whole critical section, not just the instruction address reported at the time of the abort.*

***Tuning Suggestion 31.*** *The profiler should support displaying the ReturnIP with callgraph for non-Instruction abort events, but display the EventingRIP for instruction abort events.*

***Tuning Suggestion 32.*** *The PEBS TX Abort Information bits should be all displayed by the profiling tool.*

## 8.4.5 XABORT ARGUMENTS FOR RTM FALLBACK HANDLERS

If the XABORT instruction is used to abort an RTM-based transactional region, the instruction operand is passed to the fallback handler through the EAX register. This information is also provided by the PEBS-based profiling tool for RTM. A profiling tool can use this information to classify various XABORT-based transactional aborts. Defining a convention can be also helpful to write sophisticated fallback handlers.

The following table presents the convention used in this document:

**Table 8-2. RTM Abort Status Definition**

| XABORT Code | Description |
|---|---|
| 0xff | XABORT-based abort because lock was not free when tested (Section 8.3.4) |
| 0xfe | XABORT-based abort because lock tested for the value of the elided lock (Section 8.3.9) |
| 0xfd | XABORT-based abort during a nested try lock (Section 8.3.9) |
| 0xfc: 0xf0 | Reserved |

***Tuning Suggestion 33.*** *The profiling tool should display the abort code to the user for RTM aborts.*

## 8.4.6 CALL GRAPHS FOR TRANSACTIONAL ABORTS

The profiling tool generates interrupts to collect performance monitoring information. Such interrupts will cause transactional aborts. This means a profiling tool can only collect information after a transactional abort happened and the tool cannot see any function calls on the stack that only happened inside the transactional region; the only view of the call graph it has was the one at the beginning of the transactional execution. When a transactional abort is sampled with PEBS the RIP field contains the instruction pointer after the abort and the EventingIP field contains the instruction pointer within the transactional region at the time of the abort. The same also applies for sampling non-abort events, as any sampling causes transactional aborts.

Depending on the type of abort, it can be useful to profile for either ReturnIP or EventingIP. The stack callgraph collected by the profiling tool is always associated with the ReturnIP. When it is combined with the EventingIP, it may appear noncontiguous (the EventingIP may not be associated with the lowest level caller), as any function calls inside the transactional region are not included. When the function calls inside the transactional region are required to understand the abort cause, Last Branch Records (LBRs, See Section 25) or the SDE software emulation (see Section 8.4.8) can be used.

***Tuning Suggestion 34.*** *The profiler should have options to display ReturnIP and EventingIP.*

***Tuning Suggestion 35.*** *The stack callgraph is always associated with the ReturnIP and may appear noncontiguous with the EventingIP.*

***Tuning Suggestion 36.*** *To see function calls inside the transactional region use LBRs or SDE.*

## 8.4.7 LAST BRANCH RECORDS AND TRANSACTIONAL ABORTS

The Last Branch Records (see section 17.4 in Volume 3 of the Intel Software Developer's Manual) provide information about transactional execution and aborts. Regular LBR usage is compatible with Intel TSX. Using LBRs can be useful to provide context inside the transaction, as the normal call graph is not visible. The lcall filter can be used to approximate a call graph. However, the LBR Call Graph Stack facility (Section 17.8 in Volume 3 of the Intel Software Developer's Manual) is not compatible with Intel TSX and may provide incomplete information.

***Tuning Suggestion 37.*** *The PEBS profiling handler should support sampling LBRs on abort and report them to the user.*

## 8.4.8 PROFILING AND TESTING INTEL TSX SOFTWARE SING THE INTEL® SDE

The Intel® Software Development Emulator (Intel® SDE) tool enables software development for planned instruction set extensions before they appear in hardware. The tool can also be used for extended testing, debugging and analysis of software that take advantage of the new instructions.

Programmers can use a number of Intel SDE capabilities for functional testing, profiling and debugging programs using the Intel TSX instructions. The tool can provide insight into common transactional aborts and additional profiling capability unavailable directly on hardware. Programmers should not use the tool to derive runtimes and absolute performance characteristics as those are a function of the inherently high overheads of the emulation the tool performs.

As described previously in Section 8.4.4, hardware reports the precise address of the instruction that caused an abort, unless the abort is due to either a data conflict or a resource limitation. The tool can provide the precise address of such an instruction and additional information about the instruction. The tool can further map this back to the application source code, providing the instruction address, source file names, line number, the call stacks, and the data address information the instruction was operating on. For victim transactions (aborted due to a conflict) the tool can also output source code locations where conflicting memory accesses have been executed.

This is achieved through the tool options:

**-tsx -hle_enabled 1 -rtm-mode full -tsx_stats 1 -tsx_stats_call_stack 1**

The fallback handler can use the contents of the EAX register to determine causes of aborts. The SDE tool can force a transactional abort with a specific EAX register value provided as an emulator parameter. This allows developers to test their fallback handler code with different EAX values. In this mode, every RTM-based transactional execution will immediately abort with the EAX register value being that provided as the parameter. This is quite effective in functionally testing for corner cases where a transactional execution aborts due to unresolved page faults or other similar operations (EAX = 0).

This is achieved through the tool options:

**-tsx -rtm-mode abort -rtm_abort_reason EAX.**

Intel SDE has instruction and memory access logging features which are useful for debugging capacity aborts. With the log data from Intel SDE, one can diagnose cache set population to determine if there is non-uniform cache set usage causing capacity overflows. A refined log data may be used to further diagnose the source of the aborts. The logging feature is enabled with the following options:

**-tsx_debug_log 3 -tsx_log_inst 1 -tsx_log_file 1**

Additionally Intel SDE allows to use a standard debugger (gdb and Microsoft Visual Studio) to perform functional debugging inside transactions.

## 8.4.9 HLE SPECIFIC PERFORMANCE MONITORING EVENTS

The Intel TSX Performance Events also include HLE-specific transactional abort conditions. These events track aborts due to causes listed in Section 8.2.4.4. These aborts often occur due to issues in synchronization library implementations. When a synchronization library is initially enabled for Intel TSX, it is useful to measure these events and improve the library until these counts are negligible.

- **TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK** counts the number of transactional aborts due to a store operation without the XRELEASE prefix operating on an elided lock in the elision buffer.
  - — This is often because the library is missing the XRELEASE prefix on the lock release instruction.
- **TX_MEM.ABORT_ELISION_BUFFER_NOT_EMPTY** counts the number of transactional aborts that occur because an XRELEASE prefixed lock release instruction that was committing the transactional execution finds the elision buffer with an elided lock.
  - — This typically occurs for code sequences where an XRELEASE occurs on a lock that wasn't elided and hence wasn't in the elision buffer.
- **TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH** counts the number of transactional aborts because the XRELEASE lock does not satisfy the address and value requirements for elision in the elision buffer.
  - — This occurs for example if the value being written by the XRELEASE operation is different from the value that was read by the earlier XACQUIRE operation to the same lock.
- **TX_MEM.ABORT_HLE_ELISION_UNSUPPORTED_ALIGNMENT** counts the number of transactional aborts if the lock in the elision buffer was accessed by a read in the transactional region but the read could not be serviced.
  - — This typically occurs if the access
    - **Was not properly aligned.**
    - **Had a partial overlap.**
    - **The read operation's linear address was different than the elided locks but the physical address was the same.**

These are fairly rare events.

### 8.4.10 COMPUTING USEFUL METRICS FOR INTEL® TSX

We now provide formulas to compute useful metrics with the performance events. While some of the counts are available as their own events, it can sometimes be useful to do a derivation with limited counters.

The following calculates the number of times a HLE or RTM transactional execution was started. This combines all nested regions into one region for counting purposes.

**#HLE Regions Started: HLE_RETIRED.COMMIT + HLE_RETIRED.ABORTED**
**#RTM Regions Started: RTM_RETIRED.COMMIT + RTM_RETIRED.ABORTED**

The following calculates the percentage of HLE or RTM transactional executions that aborted.

**%AbortedHLE = 100.0 * (HLE_RETIRED.ABORTED/HLE_RETIRED.START)**
**%AbortedRTM = 100.0 * (RTM_RETIRED.ABORTED/RTM_RETIRED.START)**

The following calculates the average number of cycles spent in a transactional region (See Section 8.4.1 for CyclesInTX computation).

**AvgCyclesInHLE = CyclesInTX/HLE_RETIRED_START**
**AvgCyclesInRTM = CyclesInTX/RTM_RETIRED.START**
**AvgCyclesInTX=CyclesInTX/(HLE_RETIRED.START + RTM_RETIRED.START)**

The following calculates the percentage of HLE or RTM transactional executions that aborted due to a data conflict.

**%AbortedHLEDataConflict = TX_MEM.ABORT_CONFLICT/HLE_RETIRED.START;**
**%AbortedRTMDataConflict = TX_MEM.ABORT_CONFLICT / RTM_RETIRED.START;**
**%AbortedTXDataConlict= TX_MEM.ABORT_CONFLICT / (HLE_RETIRED.START+RTM_RETIRED.START);**

The following calculates the number of HLE or RTM transactional executions that aborted due to limited resources for transactional stores.

**%AbortedTXStoreResource = TX_MEM.ABORT_CAPACITY_WRITE**

On processors based on the Broadwell and Skylake microarchitectures, the event "**TX_MEM.ABORT_CAPACITY_WRITE**" is replaced by **TX_MEM.ABORT_CAPACITY** that counts aborts due to either read or write.

The following calculates the total number of HLE or RTM transactional executions that aborted due to resource limitations. The distinction occurs because transactional reads that are evicted from the L1 data cache may not immediately cause an abort.

**%AbortedHLEResource = HLE_RETIRED.ABORTED_MISC1 - TX_MEM.ABORT_CONFLICT**
**%AbortedRTMResource = RTM_RETIRED.ABORTED_MISC1- TX_MEM.ABORT_CONFLICT**
**%AbortedTXResource = (HLE_RETIRED.ABORTED_MISC1+RTM_RETIRED.ABORTED_MISC5) - TX_EM.ABORT_-CONFLICT**

For **HLE, HLE_RETIRED.ABORTED_MISC1** may include some additional contributions from the events discussed in Section 8.4.9. For accurate results the lock library should be tuned first to minimize them.

#### NOTE

**HLE_RETIRED.ABORTED_MISC1** is also known with the more descriptive name **HLE_RETIRED.ABORTED_MIEM**. Similarly, **RTM_RETIRED.ABORTED_MISC1** is also known as **RTM_RETIRED.ABORTED_MEM**.

## 8.5 PERFORMANCE GUIDELINES

The 4th generation Intel Core Processor is the first implementation that support Intel TSX. Transactional execution incurs some implementation dependent overheads. Performance will improve in subsequent microarchitecture generations. The first TSX implementation is oriented towards typical usage of critical sections in applications. As a result, these overheads are amortized and do not normally manifest themselves at an application level performance.

However, some guidelines are relevant to keep in mind:

***Tuning Suggestion 38.*** *Intel TSX is designed for critical sections and thus the latency profiles of the XBEGIN/XEND instructions and XACQUIRE/XRELEASE prefixes are intended to match the LOCK prefixed instructions. These instructions should not be expected to have the latency of a regular load operation.*

There is an additional implementation-specific overhead associated with executing a transactional region. This consists of a mostly fixed cost in addition to a variable dynamic component. The overhead is largely independent of the size and memory foot print of the critical section. The additional overhead is typically amortized and hidden behind the out-of-order execution of the microarchitecture. However, on the 4th generation Intel Core Processor implementation, certain sequences may appear to exacerbate the overhead. This is particularly true if the critical section is very small and appear in tight loops (for example something typically done in microbenchmarks). Realistic applications do not normally exhibit such behavior.

The overhead is amortized in larger critical sections but will be exposed in very small critical sections. One simple approach to reduce perceived overhead is to perform an access to the transactional cache lines early in the critical section

The overhead of commits is reduced with processors based on the Broadwell microarchitecture.

## 8.6 DEBUGGING GUIDELINES

Using Intel TSX to implement Lock Elision does not change application semantics - all architectural state updated during an aborted transactional execution is automatically discarded by the hardware. Care must be taken if new code paths are added to the application and these paths are exercised only under transactional execution (See Section 8.2.5).

However, lock elision may change the timing relationships among different threads since it requires communication among threads only when required by data conflicts. Hence, locks may appear to execute much faster than normal. Such timing changes may expose latent bugs in an application. Exposure of such latent bugs is not unique to Intel TSX and can be expected with every new hardware generation.

Code instrumentation is a common technique while debugging multi-threaded software. As is the case with debugging timing related issues, care must be taken when instrumenting code to not perturb timing significantly and to not cause unnecessary aborts. A per thread buffer can be utilized to trace execution and log events of interests. The RDTSC instruction can be used to obtain a timestamp. The buffer should be printed outside the critical section.

Transactional aborts discard all memory state updated within the transactional region. This information cannot be traced without instrumentation support. Issues within transactional regions will show up in a profiling tool as a transactional abort and the Last Branch Record information can be used to reconstruct the control flow. On processors that support Intel® Processor Trace, the trace log allows reconstructing the full trace of the control flow inside transactions. The trace also contains markers indicating transaction start, commit and abort.

The regular assert() function would cause a transactional abort and its output information would not make it out of the transactional region. When using the RTM instructions, the assert functionality can be enhanced to end the transactional execution, make side effects visible, and terminate the program through the assert function. For example:

```
    assert(x) => if (!(x)) { while (_xtest()) _xend();  assert(0); }
```

## 8.7 COMMON INTRINSICS FOR INTEL® TSX

Recent assemblers (GNU binutils version 2.23, Microsoft Visual Studio 2012) include support for the Intel TSX instructions. On older tool chains it is possible to use the instructions as byte values.

### 8.7.1 RTM C INTRINSICS

Recent C/C++ compilers (gcc 4.8, Microsoft Visual Studio 2012, Intel C++ Compiler 17.0) support RTM intrinsics in the **immintrin.h** header file. RTM is a new instruction set and should be only used after checking the RTM feature flag using the CPUID instruction (See Chapter 3, “Basic Execution Environment” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

#### 8.7.1.1 _xbegin()

_xbegin() starts the transactional region and returns _XBEGIN_STARTED when in the transactional region, otherwise the abort code. It is important to check _xbegin() against _XBEGIN_STARTED which is not zero. Zero is a valid abort code. When the value is not _XBEGIN_STARTED the return code contains various status bits and an optional 8bit constant passed by _xabort().

Valid status bits are:

- **_XABORT_EXPLICIT:** Abort caused by _xabort(). _XABORT_CODE(status) contains the value passed to _xabort().
- **_XABORT_RETRY:** When this bit is set retrying the transactional region has a chance to commit. If not set retrying will likely not succeed.
- **_XABORT_CAPACITY:** The abort is related to a capacity overflow.
- **_XABORT_DEBUG:** The abort happened due to a debug trap.
- **_XABORT_NESTED:** The abort happened in a nested transaction.

#### _xend()

**_xend()** commits the transaction.

#### _xtest()

**_xtest()** returns true when the code is currently executing in a transaction. It can be also used with HLE.

#### _xabort()

**_xabort(constant)** aborts the current transaction. Constant can be only 8 bits. The constant is contained in the status code returned by _xbegin() and can be accessed with _XABORT_CODE() when the _XABORT_EXPLICIT flag is set. See Section 4.5 for a recommended convention.

On gcc 4.8 and later compilers, the -mrtm flag needs to be used to enable these intrinsics.

#### 8.7.1.2 Emulated RTM Intrinsics on Older GCC-Compatible Compilers

On older gcc compatible compilers that do not support the RTM intrinsics in immintrin.h, Example 8-11 shows the inline assembler equivalents that can be used.

### Example 8-11. Emulated RTM intrinsic for Older GCC Compilers

```
/* Not needed on newer toolchains that support this interface in immintrin.h */
#define _XBEGIN_STARTED        (~0u)
#define _XABORT_EXPLICIT       (1 << 0)
#define _XABORT_RETRY          (1 << 1)
#define _XABORT_CONFLICT       (1 << 2)
#define _XABORT_CAPACITY       (1 << 3)
#define _XABORT_DEBUG          (1 << 4)
#define _XABORT_NESTED         (1 << 5)
#define _XABORT_CODE(x)        (((x) >> 24) & 0xff)

#define __force_inline __attribute__((__always_inline__)) inline

static __force_inline int _xbegin(void)
{
        int ret = _XBEGIN_STARTED;
        asm volatile(".byte 0xc7,0xf8 ; .long 0" : "+a" (ret) :: "memory");
        return ret;
}

static __force_inline void _xend(void)
{
         asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory");
}

static __force_inline void _xabort(const unsigned int status)
{
        asm volatile(".byte 0xc6,0xf8,%P0" :: "i" (status) : "memory");
}

static __force_inline int _xtest(void)
{
        unsigned char out;
        asm volatile(".byte 0x0f,0x01,0xd6 ; setnz %0" : "=r" (out) :: "memory");
        return out;
}
```

## 8.7.2 HLE INTRINSICS ON GCC AND OTHER LINUX COMPATIBLE COMPILERS

On Linux and compatible systems HLE is implemented as an extension to gcc 4.8 and an older form of the C11 atomic primitives. **HLE XACQUIRE** can be used by setting the **__ATOMIC_HLE_ACQUIRE** flag to the memory model argument. HLE XRELEASE can be used with **__ATOMIC_HLE_RELEASE**.

- For __ATOMIC_HLE_ACQUIRE, the memory model must be **__ATOMIC_ACQUIRE** or stronger.
- For __ATOMIC_HLE_RELEASE __ATOMIC_RELEASE or stronger.
- For operations with a failure memory model (like __atomic_compare_exchange_n) the HLE flag is only supported on the non-failure memory model.

HLE is only supported on atomic operations that can be directly translated into IA atomic instructions. It is not supported with:

- 8 byte values on 32-bit targets.
- 16 byte values.
- Fetch-op or op-fetch other than add/sub when the result is accessed.
- __atomic_store and __atomic_clear only support __ATOMIC_HLE_RELEASE.

### 8.7.2.1 Generating HLE Intrinsics with GCC4.8

Due to a compiler bug in some versions of gcc 4.8 the -O2 or higher optimization level must be used to generate HLE hints using the atomic intrinsics.

### 8.7.2.2 C++11 Atomic Support

gcc 4.8 has support for the C++11 <atomic> header. The memory models defined there are extended with HLE flags similar to the C atomic interface. Two new flags __memory_order_hle_acquire and __memory_order_hle_release are defined. The constraints listed for the C atomic intrinsics apply. Example 8-12 shows a C++ example of an HLE intrinsic.

**Example 8-12. C++ Example of HLE Intrinsic**

```
#include <atomic>
#include <immintrin.h>
using namespace std;
atomic_flag lock;
for (;;) {
   if (!lock.test_and_test(memory_order_acquire|__memory_order_hle_acquire) {
      // Critical section with HLE lock elision
      lock.clear(memory_order_release|__memory_order_hle_release);
    break;
   } else {
     // Lock not acquired. Wait for lock and retry.
     while (lock.load())
        _mm_pause();   // abort transactional region on lock busy
   }
}
```

#### 8.7.2.3 Emulating HLE intrinsics with older GCC-Compatible Compilers

For older compilers that do not support these intrinsics inline assembler can be used. For example to emulate __atomic_exchange_n(&lock, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE), see Example 8-13.

**Example 8-13. Emulated HLE Intrinsic with Older GCC Compiler**

```
#define XACQUIRE ".byte 0xf2; " /* For older assemblers not supporting XACQUIRE */
#define XRELEASE ".byte 0xf3; "
static inline int hle_acquire_xchg(int *lock, int val)
{
   asm volatile(XACQUIRE “xchg %0,%1” : “+r” (val), “+m” (*lock) :: “memory”);
   return val;
}

static void hle_release_store(int *lock, int val)
{
   asm volatile(XRELEASE “mov %0,%1” : “r” (val), “+m” (*lock) :: “memory”);
}
```

### 8.7.3 HLE INTRINSICS ON WINDOWS C/C++ COMPILERS

Windows C/C++ compilers (Microsoft Visual Studio 2012 and Intel C++ Compiler 17.0) provide versions of certain atomic intrinsic with HLE prefixes; see Example 8-14.

**Example 8-14. HLE Intrinsic Supported by Intel and Microsoft Compilers**

```
Atomic compare-and-exchange operations:

long _InterlockedCompareExchange_HLEAcquire(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLEAcquire(__int64 volatile *Destination, __int64 Exchange, __int64
Comparand);
void * _InterlockedCompareExchangePointer_HLEAcquire(void * volatile     *Destination, void * Exchange, void *
Comparand);
long _InterlockedCompareExchange_HLERelease(long volatile *Destination, long Exchange, long Comparand);
__int64 _InterlockedCompareExchange64_HLERelease(__int64 volatile *Destination, __int64 Exchange, __int64
Comparand);
void * _InterlockedCompareExchangePointer_HLERelease(void * volatile *Destination, void * Exchange, void *
Comparand);


Atomic addition:

long _InterlockedExchangeAdd_HLEAcquire(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLEAcquire(__int64 volatile *Addend, __int64 Value);
long _InterlockedExchangeAdd_HLERelease(long volatile *Addend, long Value);
__int64 _InterlockedExchangeAdd64_HLERelease(__int64 volatile *Addend, __int64 Value);
```

### Example 8-14. HLE Intrinsic Supported by Intel and Microsoft Compilers

```
Intrinsics for HLE prefixed stores:

void _Store_HLERelease(long volatile *Destination, long Value);
void _Store64_HLERelease(__int64 volatile *Destination, __int64 Value);
void _StorePointer_HLERelease(void * volatile *Destination, void * Value);
```

Please consult the compiler documentation for further information on these intrinsics.