Intel’s Haswell CPU Microarchitecture

Haswell Instruction Set and Front-end

Haswell introduces a huge number of new instructions for the x86 ISA, that fall into four general families. The first is AVX2, which promotes integer SIMD instructions from 128-bits wide in SSE to 256-bits wide. The original AVX was a 256-bit extension using the YMM registers, but largely for floating point instructions. AVX2 is the complement and brings integer SIMD to the full YMM registers, along with some enhancements for 128-bit operation. AVX2 also adds more robust and generalized support for vector permutes and shifts. Perhaps more importantly, AVX2 includes 16 new gather instructions, loads that can fetch 4 or 8 non-contiguous data elements using special vector addressing for both integer and floating point (FP) SIMD. Gather is crucial for wider SIMD and substantially simplifies vectorizing code. Note that AVX2 does not include scatter instructions (i.e., vector addressed stores), because of complications with the x86 memory ordering model and the load/store buffers.

While AVX2 emphasizes integer SIMD, Haswell has huge benefits for floating point code. In addition to gather, Intel’s Fused Multiply Add (FMA) includes 36 FP instructions for performing 256-bit computations and 60 instructions for 128-bit vectors. As announced in early 2008, Intel’s FMA was originally architected for 4-operand instructions. However, the 22nm Ivy

Bridge can perform register move instructions in the front-end through register renaming tricks, without issuing any uops. Intel’s architects determined that MOV elimination with FMA3 provides about the same performance as FMA4, but using denser and easier to decode instructions; hence the abrupt about face in late 2008.

The third extension is 15 scalar bit manipulation instructions (known as BMI) that operate on general integer registers. These instructions fall into three general areas: bit field manipulations such as insert, shift and extract; bit counting such as leading zero count; and arbitrary precision integer multiply and rotation. The latter is particularly useful for cryptography. As an aside, Haswell also adds a big-endian move instruction (MOVBE) that can convert to and from traditional x86 little-endian format. MOVBE was introduced for Intel’s Atom, and is quite useful for embedded applications that deal with storage and networking, since TCP/IP is big-endian.

The last and most powerful of Intel’s ISA extensions is TSX, which has been extensively discussed in a previous article on Haswell’s transactional memory. In short, TSX enables programmers to write parallel code that focuses on using synchronization for correctness, while the hardware optimizes the execution for performance and concurrency. Hardware Lock Elision (HLE) transparently provides the performance and throughput of fine-grained locking, even when programmers use coarse-grained locks. Most importantly, the hint prefixes are compatible with older processors.

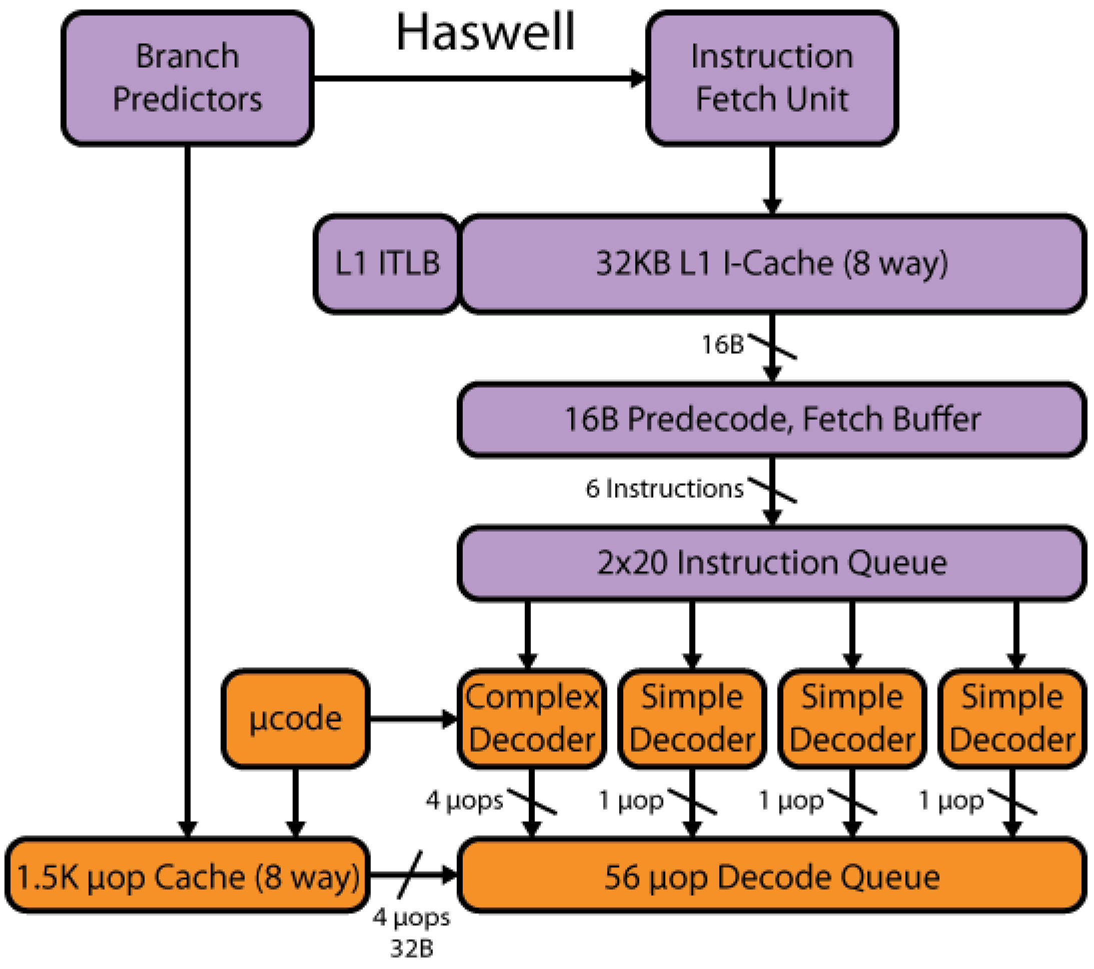
Restricted Transactional Memory (RTM) is an entirely new programmer interface that provides transactional memory to x86 developers. TM is far more useful than traditional lock-based synchronization, because transactions can protect more complex data structures and be composed across functions, modules and even applications. However, it does require linking new libraries using RTM and possibly rewriting software to get the full benefits.

Both variants of TSX are tracked at 64B cache line granularity. Excessive conflicts due to transaction limits, false sharing, or data races can actually harm performance, so developers must judiciously adopt TSX. However, future implementations will most likely have fewer conflicts and be more flexible.

Of these new instructions, the vast majority are simple instructions that decode into a single uop. However, the more complex ones such as gather and the TSX commit and abort are microcoded.

Fundamentally, x86 is quite similar to RISC architectures in a number of dimensions. ALU operations are largely the same; there are only so many ways to do addition, subtraction and multiplication. However, the front-end is quite different and one of the most challenging aspects of modern x86 CPUs. The instruction caches are kept coherent with data caches, and the variable length instructions make decoding quite complex. x86 instructions range in size from 1-15 bytes, with length-changing prefixes, inconsistent operand positions and complex microcoded instructions. Since the P6, these instructions have been transformed into more tractable fixed length uops that can be tracked by an out-of-order core. As with all architectures, the instruction stream is frequently interrupted by control flow such as conditional branches, jumps, calls and returns, which potentially redirect the instruction fetching and introduce bubbles into the pipeline.

Sandy Bridge made tremendous strides in improving the front-end and ensuring the smooth delivery of uops to the rest of the pipeline. The biggest improvement was a uop cache that essentially acts as an L0 instruction cache, but contains fixed length decoded uops. The uop cache is virtually addressed and included in the L1 instruction cache. Hitting in the uop cache has several benefits, including reducing the pipeline length by eliminating power hungry instruction decoding stages and enabling an effective throughput of 32B of instructions per cycle. For newer SIMD instructions, the 16B fetch limit was problematic, so the uop cache synergizes nicely with extensions such as AVX.


Haswell
Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8 way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
µcode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 µops
1 µop
1 µop
1 µop
1.5K µop Cache (8 way)
56 µop Decode Queue
4 µops
32B

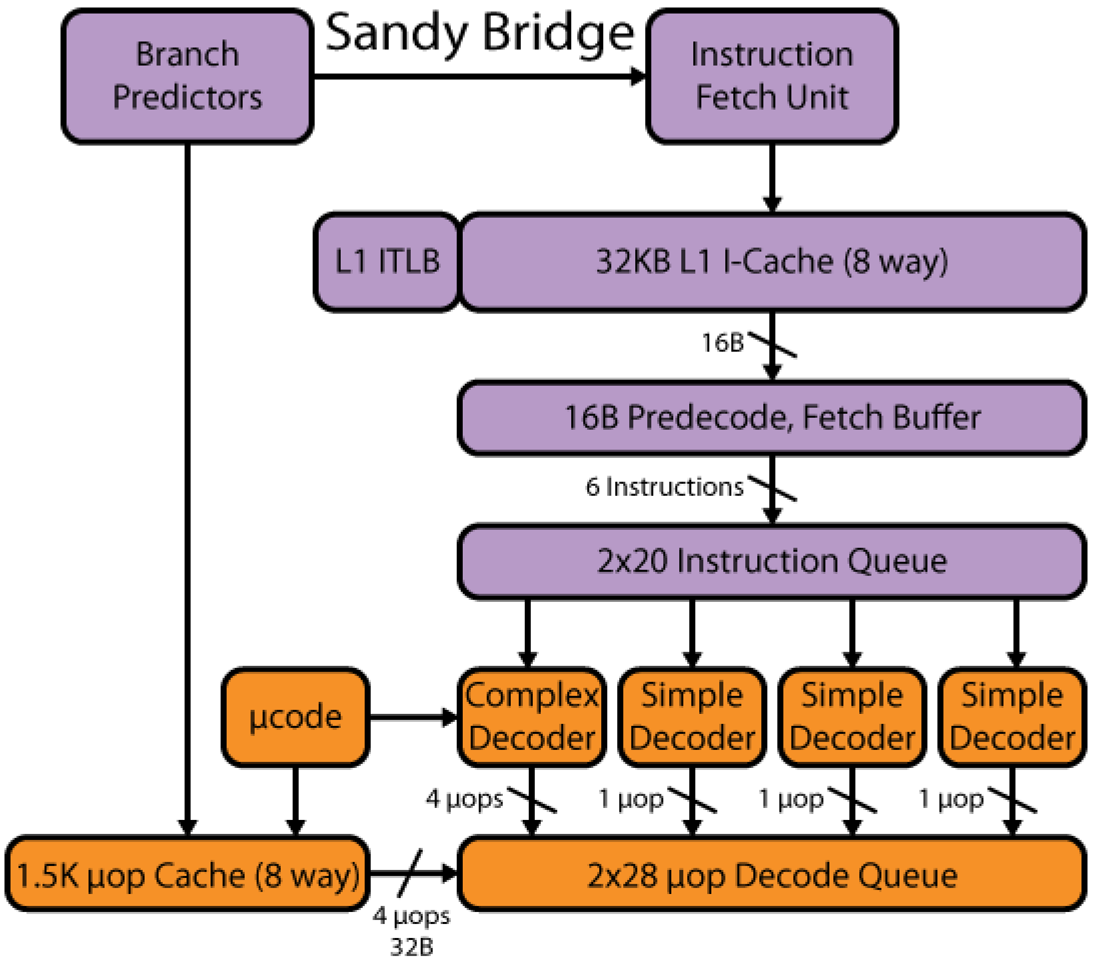

Sandy Bridge
Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8 way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
µcode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 µops
1 µop
1 µop
1 µop
1.5K µop Cache (8 way)
2x28 µop Decode Queue
4 µops
32B

Figure 1. Haswell and Sandy Bridge Front-end

The most significant difference in Haswell's front-end is undoubtedly support for the various instruction set extensions outlined above. At a high level, instruction fetch and decode microarchitecture is largely similar to Sandy Bridge, as shown in Figure 1. There are a number of subtle enhancements, but the concepts and many of the details are the same.

As to be expected, the branch prediction for Haswell has improved. Unfortunately, Intel was unwilling to share the details or results of these optimizations. The instruction cache is still 8-way associative, 32KB and dynamically shared by the two threads. The instruction TLBs are also unchanged, with 128 entries for 4KB page translations; the 4KB translations are statically partitioned between the two threads and 4-way associative. The 2MB page array is 8 entries, fully associative and replicated for each thread. Instruction fetching from the instruction cache continues to be 16B per cycle. The fetched instructions are deposited into a 20 entry instruction queue that is replicated for each thread, in both Sandy Bridge and Haswell.

The Haswell instruction cache was optimized for handling misses faster. Speculative ITLB and cache accesses are supported with better timing to improve the benefits of prefetching, and the cache controller is much more efficient about handling instruction cache misses in parallel.

As with previous generations, the decoding is performed by a complex decoder that emits 1-4 fused uops and three simple decoders that emit a single fused uop in parallel. Alternatively, instructions requiring more than 4 uops are handled by microcode and block the conventional decoders. Macro-fusion can combine adjacent compare and branch instructions into a single uop, improving the potential throughput to 5 instructions per cycle. The decoding also contains the stack engine, which resolves push/pop and call/return pairs without sending uops further down the pipeline.

The Haswell uop cache is the same size and organization as in Sandy Bridge. The uop cache lines hold upto 6 uops, and the cache is organized into 32 sets of 8 cache lines (i.e., 8 way associative). A 32B window of fetched x86 instructions can map to 3 lines within a single way. Hits in the uop cache can deliver 4 uops/cycle and those 4 uops can correspond to 32B of instructions, whereas the traditional front-end cannot process more than 16B/cycle. For performance, the uop cache can hold microcoded instructions as a pointer to microcode, but partial hits are not supported. As with the instruction cache, the decoded uop cache is shared by the active threads.

One difference in Haswell's decoding path is the uop queue, which receives uops from the decoders or uop cache and also functions as a loop cache. In Sandy Bridge, the 28 entry uop queue was replicated for each thread. However, in Ivy Bridge the uop queue was combined into a single 56 entry structure that is statically partitioned when two threads are active. The distinction is that when a single thread is executing on Ivy Bridge or Haswell, the entire 56 entry uop buffer is available for loop caching and queuing, making better use of the available resources.