CS 433 Mini-Project Presentation

Jaemin Choi (jchoi157)
Senthil Kumar Karthik (skk3)
Vipul Harsh (vharsh2)

12/02/2016
# Intel® Core™ i7 - Haswell

<table>
<thead>
<tr>
<th>Intel Core i7</th>
<th>Bloomfield, Nehalem, Clarksfield, Clarksfield XM, Lynnfield, Sandy Bridge, Sandy Bridge-E, Ivy Bridge, Ivy Bridge-E, Haswell, Haswell Refresh, Devil’s Canyon, Broadwell, Skylake, Kaby Lake</th>
<th>2008 - present</th>
</tr>
</thead>
</table>

- **Intel Core i7**
  - Brand name for various mid-range to high-end home and business microprocessors
- **Focus will be on microarchitecture codename Haswell**
Processor Core
Microarchitecture
Haswell

- 4th generation core microprocessor architecture
  - Predecessor “Sandy Bridge”
- Dual-threaded, out-of-order execution
- Decode up to 5 instructions per cycle
- Fundamentally x86 is similar to RISC
  - Instruction -> Micro-ops (u-ops)
- Majority of challenges are in front end

Image source: Figure 2-2, Intel® 64 and IA-32 Architectures Optimization Reference Manual
Pipeline Overview

- Fourteen-stage efficient pipeline
- Consists of 3 major components:
  1. In-order issue front end
  2. Out-of-order superscalar execution engine
  3. In-order retirement unit (similar to commit stage in simple pipeline)
Pipeline Overview

- Fourteen-stage efficient pipeline
- Consists of 3 major components:
  1. In-order issue front end
     - Fetch instructions and decode into micro-ops (u-ops)
     - Feed pipeline with continuous stream of u-ops from most likely execution path
  2. Out-of-order superscalar execution engine
  3. In-order retirement unit (similar to commit stage in simple pipeline)
Pipeline Overview

- Fourteen-stage efficient pipeline
- Consists of 3 major components:
  1. In-order issue front end
  2. Out-of-order superscalar execution engine
    - Reorder u-ops in “dataflow” order
    - Dispatches upto 8 u-ops to execution core per cycle
  3. In-order retirement unit (similar to commit stage in simple pipeline)
Pipeline Overview

- Fourteen-stage efficient pipeline
- Consists of 3 major components:
  1. In-order issue front end
  2. Out-of-order superscalar execution engine
  3. In-order retirement unit (similar to commit stage)
     - Ensures program order in execution of u-ops
     - Handles faults and any exceptions during execution
Components of Front End

- Legacy Decode Pipeline
- Decoded I-Cache
- Branch Prediction Unit (BPU)
- Micro-op queue

Image source: http://www.realworldtech.com/haswell-cpu/2/haswell-1.png
Components of Front End

- Legacy Decode Pipeline
  - Decodes instructions to u-ops
  - Includes ITLB, I-Cache, I-Predescode and I-Decode units
- Decoded I-Cache
- Branch Prediction Unit (BPU)
- Micro-op queue

Image source: http://www.realworldtech.com/haswell-cpu/2/haswell-1.png
Components of Front End

- Legacy Decode Pipeline
- Decoded I-Cache
  - An accelerator for the decode pipeline
  - Caches output of instruction decoder
- Branch Prediction Unit (BPU)
- Micro-op queue

Image source: http://www.realworldtech.com/haswell-cpu/2/haswell-1.png
Components of Front End

- Legacy Decode Pipeline
- Decoded I-Cache
- Branch Prediction Unit (BPU)
  - 16-entry Return Stack Buffer
  - Front end queuing of BPU lookups
  - Makes predictions for 32 bytes at a time
  - 15 cycle misprediction penalty
- Micro-op queue

Image source: http://www.realworldtech.com/haswell-cpu/2/haswell-1.png
Components of Front End

- Legacy Decode Pipeline
- Decoded I-Cache
- Branch Prediction Unit (BPU)
- Micro-op queue
  - Between u-op generation and renamer
  - Un-lamination (fragment 1 u-op to multiple)

Image source: http://www.realworldtech.com/haswell-cpu/2/haswell-1.png
Out of Order Execution Engine

- The module that makes Haswell interesting with many new visible changes

Components:
- Renamer
- Scheduler
- Execution core

Out of Order Execution Engine

- **Renamer**
  - Moves u-ops from front-end to the execution core
  - Renames architectural source and destination x86 registers
  - Allocates resources like load or store buffers and reorder buffer to the u-ops
  - **Optimization**: Some u-ops can complete to execution in renamer itself and thus removed from pipeline. Examples include NOP, Zero idioms, One idioms, VZEROUPPER

- **Scheduler**

- **Execution core**
Out of Order Execution Engine

- **Renamer**
- **Scheduler**
  - Queues u-ops till all operands are ready and execution resources are available
  - Sources of operands for a u-op
    - Register file entry
    - Bypass directly from an execution unit (data forwarding)
  - It is similar to the concept of Reservation Stations discussed in Tomasulo’s Algorithm
  - Expanded to hold 60 entries and can dispatch up to 8 u-ops in one cycle
  - Possible causes for stalls
    - RAW dependencies
    - Transitions between Intel SSE integer and Intel SSE FP operations
- **Execution core**
Out of Order Execution Engine

- **Renamer**
- **Scheduler**
- **Execution core**
  - 8 ports for dispatch
    - 4 memory
    - 4 computations
  - Of the 4 ports for computations, 2 provide dedicated execution to FMA units
  - Breakdown of the 4 memory ports
    - 2 dual-use ports for load and store-address operations
    - 1 dedicated store-address port & 1 dedicated store-data port
  - Execution units are arranged into 3 stacks: integer, SIMD integer and FP (scalar and SIMD)

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
<th>Port 2, 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Port 6</th>
<th>Port 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU, Shift</td>
<td>ALU, Fast LEA, BM</td>
<td>Load_addr, Store_addr</td>
<td>Store_data</td>
<td>ALU, Fast LEA, BM</td>
<td>ALU, Shift, JEU</td>
<td>Store_addr, Simple_AGU</td>
</tr>
<tr>
<td>SIMD_Log, SIMD misc, SIMD_Shifts</td>
<td>SIMD_ALU, SIMD_Log</td>
<td></td>
<td></td>
<td>SIMD_ALU, SIMD_Log,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMA/FP_mul, Divide</td>
<td>FMA/FP_mul, FP_add</td>
<td></td>
<td></td>
<td>Shuffle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd_Jeu</td>
<td>slow_int,</td>
<td></td>
<td></td>
<td>FP_mov, AES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Image source: Table 2-6, Intel® 64 and IA-32 Architectures Optimization Reference Manual
Innovations in Haswell microarchitecture

4 major architectural enhancements from Sandy Bridge

1. AVX2 - Advanced Vector Extensions
2. FMA - Fused multiply-add unit
3. BMI (Bit Manipulation Instructions)
4. TSX - Transactional Memory
Innovations in Haswell microarchitecture

4 major architectural enhancements from Sandy Bridge

1. AVX2 - Advanced Vector Extensions
   ○ Original AVX was for 256-bit FP operations, AVX2 brings in integer SIMD for 256-bit vectors
   ○ AVX2 also includes new gather instructions and loads for fetching non-contiguous data
2. FMA - Fused multiply-add unit
3. BMI (Bit Manipulation Instructions)
4. TSX - Transactional Memory
Innovations in Haswell microarchitecture

4 major architectural enhancements from Sandy Bridge

1. AVX2 - Advanced Vector Extensions
2. FMA - Fused multiply-add unit
   - Each FMA operation is effectively 2 FP ops i.e. multiply and add in same execution unit
   - Handy for media processing and 3D rendering work
3. BMI (Bit Manipulation Instructions)
4. TSX - Transactional Memory
Innovations in Haswell microarchitecture

4 major architectural enhancements from Sandy Bridge

1. AVX2 - Advanced Vector Extensions
2. FMA - Fused multiply-add unit
3. BMI (Bit Manipulation Instructions)
   - 15 new scalar BMI, powerful for cryptography-related operations
   - MOVBE: big-endian move instruction
     - can convert to and from little-endian (example use-case: TCP/IP)
4. TSX - Transactional Memory
Innovations in Haswell microarchitecture

4 major architectural enhancements from Sandy Bridge

1. AVX2 - Advanced Vector Extensions
2. FMA - Fused multiply-add unit
3. BMI (Bit Manipulation Instructions)
4. TSX - Transactional Memory
   ○ A programming model for concurrency and multithreading
References

- Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - Chapter 2.2 The Haswell Microarchitecture
  - Chapter 2.3 Intel Microarchitecture Code Name Sandy Bridge
  - Chapter 2.4 Intel Core Microarchitecture and Enhanced Intel Core Microarchitecture
Memory Hierarchy
Cache Structure
Cache Structure

Image source: Figure 11-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
Cache Structure

Image source: Figure 11-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
Cache Structure

Image source: Figure 11-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
Cache Structure

Image source: Figure 11-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer's Manual
**Cache Structure**

- 3 levels of cache
  - L1 cache: split (data/instruction)
  - L2 cache: unified
  - Each core has own L1 and L2 caches
  - L3 cache: unified, inclusive, and shared by all processors
  - Peak bandwidth doubled from last generation (Sandy Bridge)

- Load & store buffers

*Image source: Table 2-9, Intel® 64 and IA-32 Architectures Optimization Reference Manual*
Methods of Caching

- Default: **Write Back (WB)**
  - Writes/reads to/from memory cached
  - Speculative reads
  - Write misses cause cache line fills
  - WB operations triggered by cache line replacements & mechanisms for cache coherency
  - Best performance, but requires all devices that access system memory to be able to snoop memory accesses

---

**Table 11-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual**
TLB Structure

- 2 levels of TLB
- Instruction TLB
  - 64 entries per thread (128 total)
  - Due to HyperThreading
- Data TLB
- Second level TLB
  - Unified
  - Available only on Core i7

<table>
<thead>
<tr>
<th>Level</th>
<th>Page Size</th>
<th>Entries</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>4KB</td>
<td>128</td>
<td>4 ways</td>
</tr>
<tr>
<td>Instruction</td>
<td>2MB/4MB</td>
<td>8 per thread</td>
<td></td>
</tr>
<tr>
<td>First Level Data</td>
<td>4KB</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>First Level Data</td>
<td>2MB/4MB</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>First Level Data</td>
<td>1GB</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Second Level</td>
<td>Shared by 4KB and 2/4MB pages</td>
<td>1024</td>
<td>8</td>
</tr>
</tbody>
</table>

Image source: Table 2-10, Intel® 64 and IA-32 Architectures Optimization Reference Manual
Main Memory

- Dependent on actual hardware
  - Maximum physical memory size: 32GB
  - Maximum memory bandwidth: 25.6GB/s
  - Supports DIMMs
  - Supports DDR3-1300/1600, DDR3L-1300/1600
Virtual Memory

- Modes of operation
  - IA-32 architecture (32-bit)
    - Protected mode, ...
  - Intel 64 architecture (64-bit)
    - All modes of IA-32
    - And IA-32e mode
Protected Mode (32-bit)

- 2 components
  - Segmentation
  - Paging

- Segmentation
  - Mandatory
  - Used to isolate multiple programs on one processor
  - Logical address -> linear address

- Paging
  - Optional
  - Conventional demand-paging, virtual memory
  - Linear address -> physical address

Image source: Figure 3-1, Volume 3, Intel® 64 and IA-32 Architectures Software Developer's Manual
Protected Mode (32-bit)

- 2 components
  - Segmentation
  - Paging

- Segmentation
  - Mandatory
  - Used to isolate multiple programs on one processor
  - Logical address -> linear address

- Paging
  - Optional
  - Conventional demand-paging, virtual memory
  - Linear address -> physical address
Protected Mode (32-bit)

- 2 components
  - Segmentation
  - Paging
- Segmentation
  - Mandatory
  - Used to isolate multiple programs on one processor
  - Logical address -> linear address
- Paging
  - Optional
  - Conventional demand-paging, virtual memory
  - Linear address -> physical address

Image source: Figure 3-1, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
IA-32e Mode (64-bit)

- No segmentation
- 2 sub-modes
  - 64-bit mode
  - Compatibility mode
- 64-bit mode
  - Enables 64-bit applications to run
- Compatibility mode
  - Enables 32-bit applications to run
- Mode enabled on an individual code basis
  - Both 64-bit & 32-bit applications can run simultaneously

Image source: Figure 4-8, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
Paging Modes

- Translate linear addresses into physical addresses
  - Using hierarchical paging structures (page directory, page table, …)
  - Every page structure is 4KB
  - Upper portion of linear address used to identify paging structures
  - Lower portion of linear address identifies page offset
  - Last entry identifies page frame

Image source: Table 4-1, Volume 3, Intel® 64 and IA-32 Architectures Software Developer's Manual
32-bit Paging

- Translates 32-bit linear address to 32-bit physical address
  - Supports up to 4GB of physical memory
  - Register CR3 points to page directory
- With 4MB pages, only page directory is used (without page tables)

Image source: Figure 4-2, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
IA-32e Paging

- Translates 48-bit linear address to 52-bit physical address
  - Default paging mode in Intel 64 architectures
  - Backward compatible with 32-bit programs
  - 4 levels of paging structures
- Caches for PML4, PDPTE, and PDE

Image source: Figure 4-8, Volume 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual
Prefetching

- Mechanisms
- Hardware prefetching of data (to L1 cache)
- Hardware prefetching of data (to L2 cache)
- Software prefetching
Prefetching

- **Mechanisms**
  - Hardware prefetch for cache lines of data or instructions
  - Software prefetch for data
- **Hardware prefetching of data (to L1 cache)**
- **Hardware prefetching of data (to L2 cache)**
- **Software prefetching**
Prefetching

- **Mechanisms**
  - **Hardware prefetching of data (to L1 cache)**
    - Data Cache Unit (DCU) prefetcher
      - Triggered by an ascending access to very recently loaded data
      - Processor assumes this is part of a streaming algorithm
    - Instruction Pointer (IP) based strided prefetcher
      - Keeps track of individual load instructions
      - If a load is detected to have regular stride, then a prefetch is sent to next address
  - **Hardware prefetching of data (to L2 cache)**
  - **Software prefetching**
Prefetching

- **Mechanisms**
- **Hardware prefetching of data (to L1 cache)**
- **Hardware prefetching of data (to L2 cache)**
  - Data Prefetch Logic (DPL)
    - Prefetches data to L2 based on past request patterns of the DCU to L2
    - Maintains an array structure to store addresses from the DCU
    - Tracks accesses to one 4KB page in each entry
- **Software prefetching**
Prefetching

- Mechanisms
- Hardware prefetching of data (to L1 cache)
- Hardware prefetching of data (to L2 cache)
- Software prefetching
  - Programmer should use PREFETCH hint instructions
References

- Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - Chapter 2.2 The Haswell Microarchitecture
  - Chapter 3.7 Prefetching
- Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3
  - Chapter 3 Protected-Mode Memory Management
  - Chapter 4 Paging
  - Chapter 11 Memory Cache Control
Multicore/thread-level Parallelism Support
Overview

- Thread Synchronization
- Network interconnect
- Cache Coherence Protocol
- Hyper-Threading
- Multicore guidelines for OS
Thread Synchronization: TSX

- Transactional Synchronization Extensions (TSX)
- Detects conflicting memory accesses in a “Transaction” code dynamically at runtime in hardware
- “Transaction” code boundary - specified by the programmer
- If conflicting memory access is detected, whole transaction is aborted and program is notified.
Thread Synchronization: TSX (Example)

**SERIAL VERSION**

```c
for(int i = 0; i < N; i++)
{
    int acc_num = acc[i];
    sums[acc_num] += 10;
}
```

**WITH MULTI-THREADING**

```c
mutex lock;
#pragma omp parallel for num_threads(8)
for(int i = 0; i < N; i++)
{
    int acc_num = acc[i];
    lock.acquire();
    sums[acc_num] += 10;
    lock.release();
}
```

Example borrowed from:
Thread Synchronization: TSX (Example)

SERIAL VERSION

```c
for(int i = 0; i < N; i++){
    int acc_num = acc[i];
    sums[acc_num] += 10;
}
```

WITH TSX: transactions

```c
mutex lock;
#pragma omp parallel for num_threads(8)
for(int i = 0; i < N; i++){
    int acc_num = acc[i];
    if(_xbegin()==-1)
    {
        sums[acc_num] += 10;
        _xend();
    }
}
```

Example borrowed from:
Thread Synchronization: TSX (Example)

**SERIAL VERSION**

```c
for(int i = 0; i < N; i++){
    int acc_num = acc[i];
    sums[acc_num] += 10;
}
```

**WITH TSX: transactions with fallback code**

```c
mutex lock;
#pragma omp parallel for num_threads(8)
for(int i = 0; i < N; i++){
    int acc_num = acc[i];
    if(_xbegin()==-1) {
        if( !lock.is_acquired())
            sums[acc_num] += 10;
        else
            _xabort(1);
    }else {
        lock.acquire();
        sums[acc_num] += 10;
        lock.release();
    }
}
```

Network interconnect: Ring topology

Components

- CPU Cores
- Last level cache (L3)
- Graphics Engine
- System agents (Uncore)
Ring interconnect


Figure 2-3. Four Core System Integration of the Haswell Microarchitecture
Ring interconnect


CPU Cores
- L1, L2 cache
- ALU, FPU

Figure 2-3. Four Core System Integration of the Haswell Microarchitecture
Ring interconnect


L3 cache:
- L3 divided into slices, equal to the number of CPU cores
- Cores can access any slice via ring topology

Legend:
- Uncore
- CPU Core

Figure 2-3. Four Core System Integration of the Haswell Microarchitecture
Ring interconnect


Figure 2-3. Four Core System Integration of the Haswell Microarchitecture
Ring interconnect


System agent
- memory controller
- thunderbolt - interface to external device
- QPI controllers

Legend:
- Uncore
- CPU Core
Cache Coherence Protocol: MESIF

- M - Modified
- E - Exclusive
- S - Shared
- I - Invalid
- F - Forward
  - Don’t go to memory on a cache miss, if block in other caches
  - Forwards data block to other processors

Image borrowed from: http://www.texample.net/media/tikz/examples/PNG/mesif.png
Hyper-Threading (…not to be confused with multi threading)

- Intel’s technology for simultaneous multithreading (SMT)
- For every physical processor core, there are two virtual (logical) cores

Image borrowed from: https://i.ytimg.com/vi/lMdMa-VaVRc/maxresdefault.jpg
Hyper-Threading (...not to be confused with multi threading)

Advantages
- Increases the number of independent instructions on a physical core
- Shared execution resources, one thread can use execution resources when another thread using the same resources is stalled

Disadvantages
- Complex design
- High overhead because of hyperthreading, gains only in specific cases
- More cache misses since threads on same physical core share the same cache
Hyper-Threading (...not to be confused with multi threading)

**Advantages**
- Increases the number of independent instructions on a physical core
- Shared execution resources, one thread can use execution resources when another thread using the same resources is stalled

**Disadvantages**
- Complex design
- High overhead because of hyperthreading, gains only in specific cases
- More cache misses since threads on same physical core share the same cache
General Multicore Guidelines for OS

- Advance prefetching mechanisms provided in the hardware
- Software prefetching and overlapping memory reads can improve performance if queue size is large for bus memory access
- TLB shootdown - invalidating Page Table entries (PTE) in TLB’s of all other processors; propagation of changes in paging structure
References

- Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - Chapter 2.2 The Haswell Microarchitecture
  - Chapter 8 Multicore and Hyper-Threading Technology
- Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3
  - Chapter 4 Paging
  - Chapter 11 Memory Cache Control
What After Haswell?

- **Tick-Tock** model
  - Tick: die shrink of process technology
  - Tock: new microarchitecture
  - Haswell was the last microarchitecture in this model
- **“process-architecture-optimization”** model
  - 5th generation - Broadwell (process)
  - 6th generation - Skylake (architecture)
  - 7th generation - Kaby Lake (optimization)
- Reduced power consumption drastically
- More performance improvements targeted towards on-chip GPU
Thank you! Questions?
Backup Slides
Progress flow of instruction (remove?)

1. Branch Prediction Unit (BPU) chooses next block of code to execute in program
2. Corresponding u-ops sent to Rename/Retirement unit
3. Micro-op dispatched and executed using available execution resources
4. Branch mispredictions handled during branch execution; front end signaled to deliver u-ops from correct execution path
5. Reordering of memory operations to increase parallelism
6. Exceptions(faults, traps) handled at retirement of faulting instruction
Load & Store Buffers

- Temporarily stores reads/writes to memory
- Improves processor performance
  - Allows processor to continue executing instructions without having to wait for completion
  - Allows memory accesses to be delayed for more efficient use of memory bus
- Transparent to software
  - Even in systems with multiple processors
- Contents drained to memory when:
  1. An exception or interrupt is generated
  2. When a serializing instruction is executed
  3. When an I/O instruction is executed
  4. When a LOCK operation is performed
Ring interconnect

Components

- CPU Cores
  - L1, L2 cache
  - ALU, FPU
- Last level cache (L3)
- Graphics Engine
- System agents (Uncore)
Ring interconnect

Components

- CPU Cores
- Last level cache (L3)
  - L3 cache is divided into slices - one per each core
  - Each core is directly connected to its L3 slice
  - Each core can access any slice via a ring topology
  - Accessing another slice is slower - more number of hops
- Graphics Engine
- System agents (Uncore)
Ring interconnect

Components

- CPU Cores
- Last level cache (L3)
- Graphics Engine
- System agents (Uncore): Functions that aren’t part of core
  - memory controller
  - thunderbolt - hardware interface to external peripherals
  - QPI Controllers
    - QPI (Quick Path interconnect): Intel’s protocol used for point to point connections