



Introduction to Micro- Architecture and Software Execution

David Levinthal
Mar 25, 2013

Disclaimer

This presentation merely represents how the author thinks about micro processor execution

**The model discussed is arrived at from publicly available publications, reverse engineering based on kernels and performance events and Occam's razor
Any actual similarity to reality is purely coincidental**

Micro-architecture and SW execution

- **Overview: Why know this?**
- **Cachelines and cacheline movement**
- **Caches**
 - Cache hierarchy/ Associativity
 - Cache coherence/line replacement
- **Core Pipeline**
 - Uop flow
 - Front end
 - Branch prediction
 - decoders and trace caches
 - resource allocation
 - Out of Order executions
 - Scheduling
 - Execution
 - Reorder

Why know this?

Software optimization reduces the cost of running programs

Computers are not free

- **Successful optimization requires an understanding of how a processor executes code**
 - and what it takes to do that efficiently
 - SW execution is about HW and SW
- **Execution is mostly about the movement of memory addresses for instructions and data into and out of the processors execution units**
 - Optimization is using the processors' "plumbing" effectively

Why know this?

To use HW performance events one must understand what they measure and what those measurements imply.

An understanding of how the micro architectural components can limit performance is critical to achieving this.

A mental model of the execution flow and how it manifests itself in the measurable quantities will be the basis of any understanding of performance bottlenecks.

Memory addresses and cachelines

Process memory space:

Each process has its own virtual (or linear) memory address range

Data and instructions occupy this space

Compiled code is loaded in modules by the OS using mmap

modules can be loaded, unloaded, reloaded

Data address space is usually allocated with malloc (new)

these in turn just invoke mmap

Memory addresses and cachelines

HW uses physical addresses

mmap allocates virtual addresses

- creating a contiguous sequential virtual address range

- physical address are assigned on initialization/reloading from swap space

- physical addresses can be discontinuous

- come in units of "physical pages" (4KB by default)

 - low order address bits within a page are identical for virtual and physical address

 - 12 bits for 4KB pages

DTLB translates virtual addresses to physical

HW is accessed by physical addresses

- caches, dram, addressable memory on devices (video cards, etc)

- L1D can cheat doing access with low order bits in parallel with

 - DTLB conversion to physical address

Memory addresses and cachelines

For HW the fundamental unit of address space is a cacheline

64 bytes on x86 processors

lowest 6 bits of address

64 lines/4KB page

Order of bit significance can be least to most (little endian, x86) or most to least (big endian, PPC)

Cacheline Movement

Cacheline movement is usually dominant performance limitation in large applications consuming large amounts of data

during application execution induced by:

- loads (demand read)**

- stores (read for ownership/RFO or NT store)**

 - data in cacheline will be modified by stores**

- SW prefetch (movement with cache as final target)**

- HW prefetch (movement with cache as final target)**

- Instruction fetch (driven by branch prediction HW)**

Cacheline Movement

HW prefetch looks for access patterns

4 component prefetcher in Intel processors

- High density of accesses with a increasing or decreasing trend

- Pull 2 lines at a time

- Strided access a single instruction address (L1D prefetch)

- Streaming pattern detected in L1D

Branch prediction does not prefetch instruction lines

SNB/IVB can prefetch to L3 instead of L2

freeing resources in the core

lines are pulled into core by OOO executed loads
taking advantage of larger OOO window

Caches

Microprocessors have multi level system of temporary storage for cachelines

access latency increases with size

Typically smallest caches are dedicated to data or instructions: L1D and L1I

Larger caches are unified (L2 and L3)

**Multi core processors typically have a
2 level cache hierarchy/core**

**Large shared cache in common
uncore/northbridge**

Caches

Caches are divided into sets based on address aka associative sets

ex: 16KB 4 way associative cache

$16\text{KB}/4 = 4\text{KB} = 1 \text{ page}$

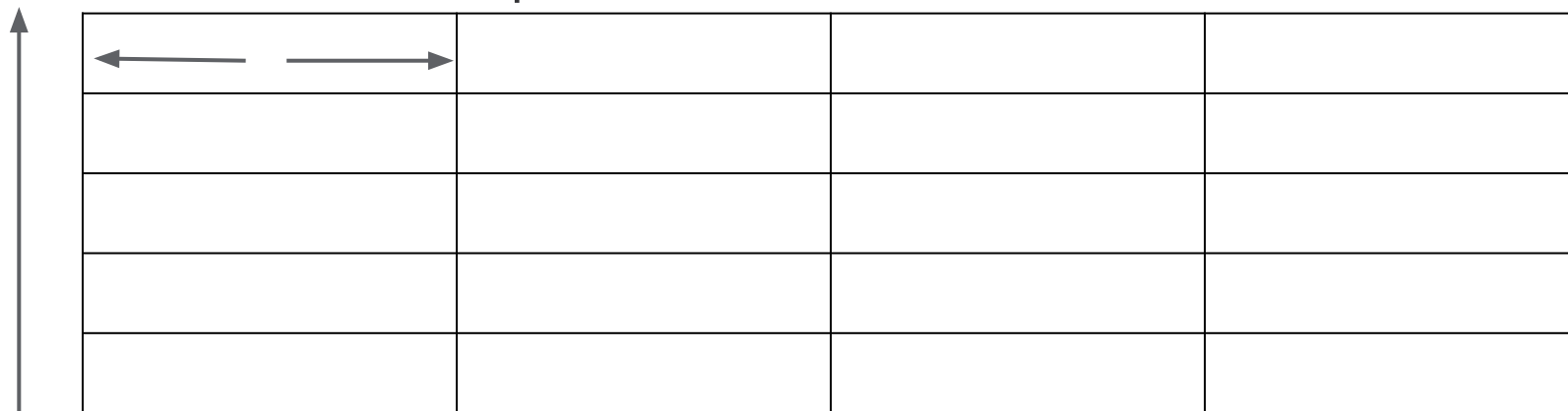
**all lines with the same offset to the page boundary
will try to be put in one of the 4 lines of a single
associative set**

bits 11:7 are identical

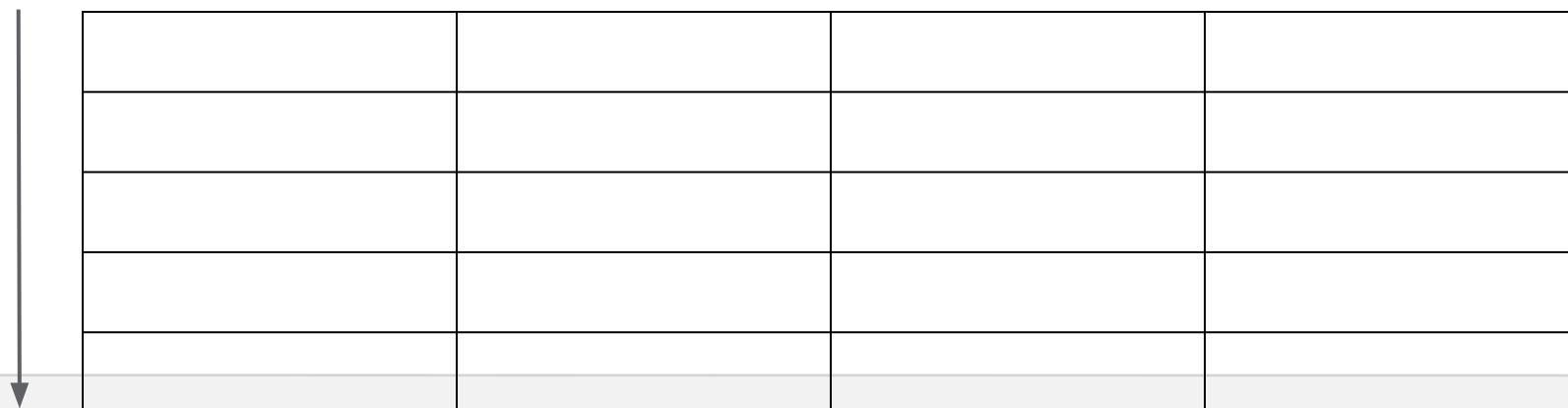
address aliasing

4 way set associative 16KB cache

bits 5:0 define position within cacheline



bits 11:6 define the row each column is one page long



Caches

Bad address usage can result in only a few associative sets being actively used

"locally" (temporally)

ex: dense matrix multiply with leading dimension that is a power of 2

This will cause access strides that are a power of 2

The stride of associative sets is also a power of 2

Causing a harmonic use of associative sets

Shared L3 Caches

Two basic styles:

Single table vs multiple components

WSM/NHM and earlier had single table caches

SNB/IVB (NHM-EX/WSM-EX) have component caches on a ring

Components called C-Boxes

C-Box assigned through hash encoding of address

Wider variability of L3 latency

Cache Coherence

Only valid copies of cachelines should be accessed: cache coherence

if parallel execution allows inconsistent copies to exist, correct/consistent results become impossible

Cache coherence is supported by MESIF protocol

Lines can only be Modified if core has Exclusive access

Other copies must be Invalidated

Shared/Forward state allows multiple readers

Core must always search for current correct copy on new read aka snoop other caches

Cacheline replacement

Lines are replaced on the basis of "least recently used" (LRU) algorithm

**Modified lines are written back to higher cache level
aka writeback on eviction**

minimizes cacheline movement

**Intel populates caches inclusively on read,
mostly**

all caches get a copy when the line is loaded (mostly)

L3 acts as a snoop filter: minimizing latency

AMD populates L1D/I on read

higher caches populated on eviction

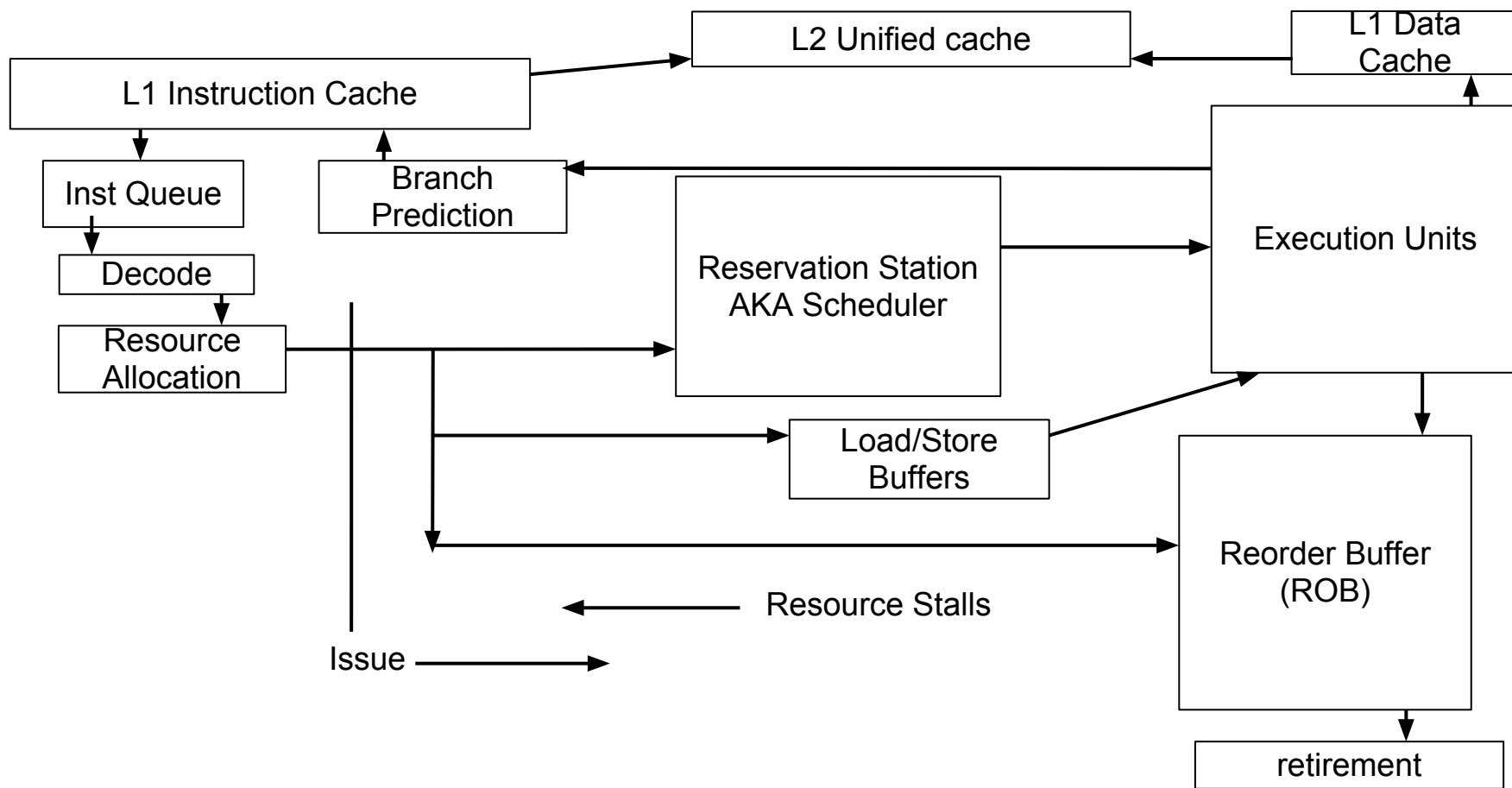
minimize cacheline duplication

Execution

- **Core Pipeline**

- **Uop flow**
- **Front end**
 - **Branch prediction**
 - **decoders and trace caches**
 - **resource allocation**
- **Out of Order executions**
 - **Scheduling**
 - **Execution**
 - **Reorder**

Artistic Out of Order Pipeline



Instruction/UOP flow I-Fetch

**HW control flow prediction drives
instruction cacheline fetching**

Branch prediction does NOT prefetch instructions

It is a fetch. Lines are pulled to Instr Decode Queue

Equivalent of a load

**Analysis of cacheline movement with
performance events suggests that 2 lines
are moved/ifetch**

Instruction/UOP flow I-Fetch

Offcore instruction lines are fetched to L1I and L2

Evicted from those caches independently

Instruction lines are not written back to L2

as they are not modified (not considering self modifying code)

L2 HW prefetcher can prefetch instructions to L2

Instruction/UOP flow I-Fetch

Branch prediction is based on record of historical path through executable

Branch history table

Branches can be taken or not taken

Branches may have multiple targets

indirect branch

target address in register

ex: function pointer

If history table contains no record of a branch within the instruction stream, it is not predicted

BACLEAR

non-prediction != misprediction

Instruction/UOP flow: Front end

On Intel instructions are fully decoded to uops on the fly

- fetches instruction blocks are 16 bytes**

- front end is common between Hyperthreads**

- front end is in order**

- fe stages alternate on hyperthreads if both have instructions**

On AMD some decoding is stored in the cachelines

- but only in copies in L1I and L2**

- fetches instruction blocks are 32 bytes**

- Bulldozer has some duplicated FE HW**

- Threading is more independent on AMD**

- Using more silicon & power**

Instruction/UOP flow: Front end

From here on this is just Intel

16 byte blocks must be broken into instructions

- instructions can span across 16 byte boundaries**
- bit strings must be checked for "length changing prefix" (LCP)**

- such instructions require extra logic**
 - and a 6 cycle FE penalty**

Instruction/UOP flow: Front end

Instructions get decoded into uops

- Allowing greater flexibility in OOO execution

- One decoder supports up to 4 uop instructions

- Longer instructions converted to uops by ucode sequencer (MS)

 - rep mov, idiv, sincos, etc

 - FP exceptions also handled by MS

Four uops are decoded per cycle

- four decoders

 - three only decode instructions that produce 1 uop

 - 4 uops produced per cycle

Instruction/UOP flow: Front end

Decoded uops are assigned required resources then shipped to back end for OOO execution (Intel:Issued, AMD: Dispatched)

- registers**

- place in scheduler/Reservation Station (RS)**

- place in re-order buffer (ROB)**

- memory operations need load/store buffers**

- etc**

**If required resources are unavailable,
uop flow from FE must be blocked
resource stall**

Instruction/UOP flow: Front end

SNB/IVB have decoded uop trace cache (DSB)

allows FE to be run in low power if control flow points to uop streams in DSB

Faster FE flow

Restrictions on what instruction streams can be cached in DSB

**If uop stream cannot be cached, switch to standard decode mode
6 cycle penalty in FE**

FE penalties don't always slow execution

Instruction and uop queues need to drain to limit BE execution

Instruction/UOP flow: Front end

Loop stream detector

Long trip count loops can be detected with the branch prediction

If they are small enough the entire loop can be held in a HW structure called the LSD

one per hyperthread

In NHM and later the LSD can hold 28 uops

cmp and branch should be microfused to 1 uop

Allows FE to power down

Can power back up very quickly

Can improve performance for discontinuous loops

Out of Order Execution

Once instructions have been:

delivered to the pipeline FE

decoded to uops

had resources assigned

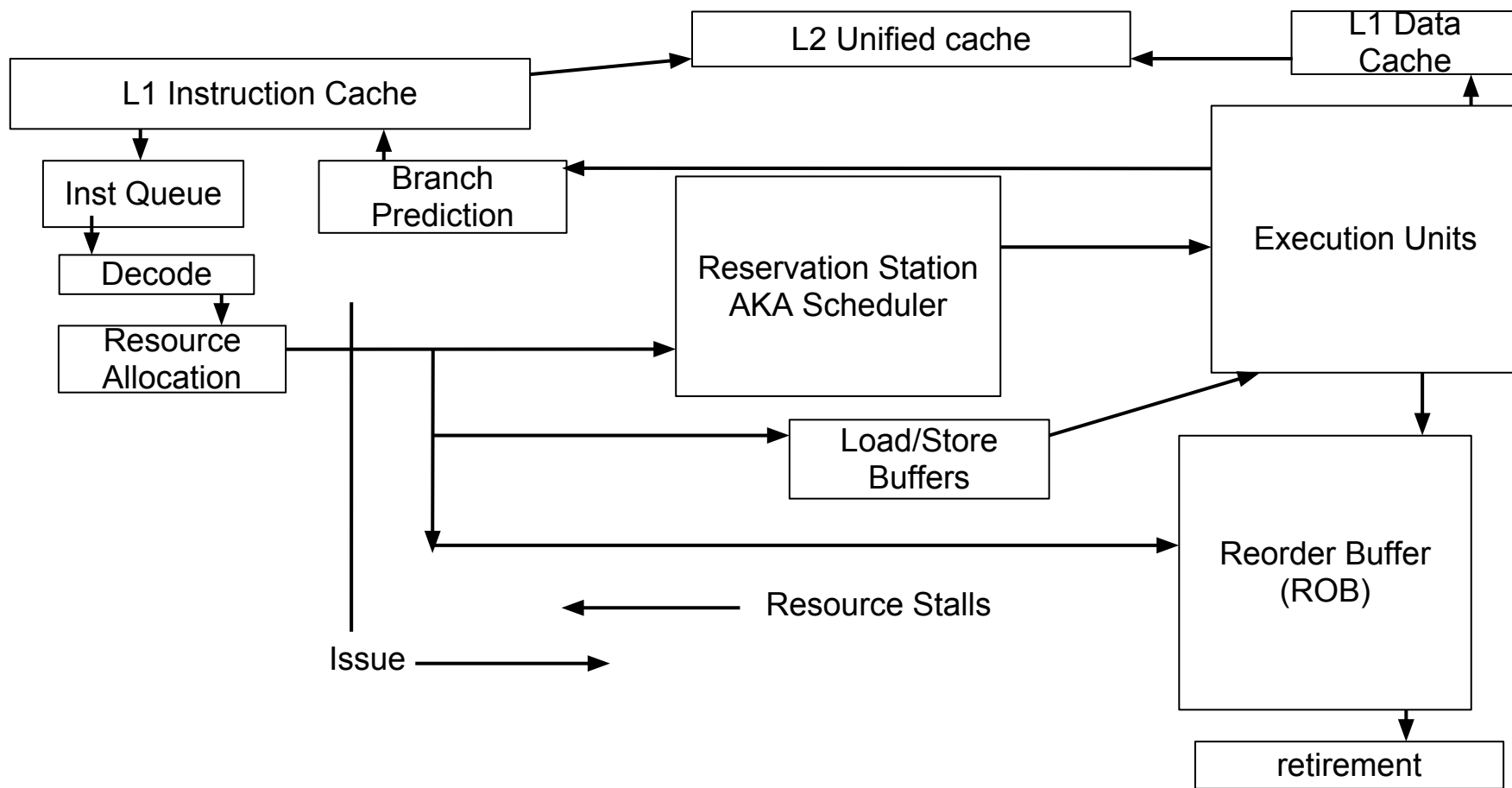
issued to BE

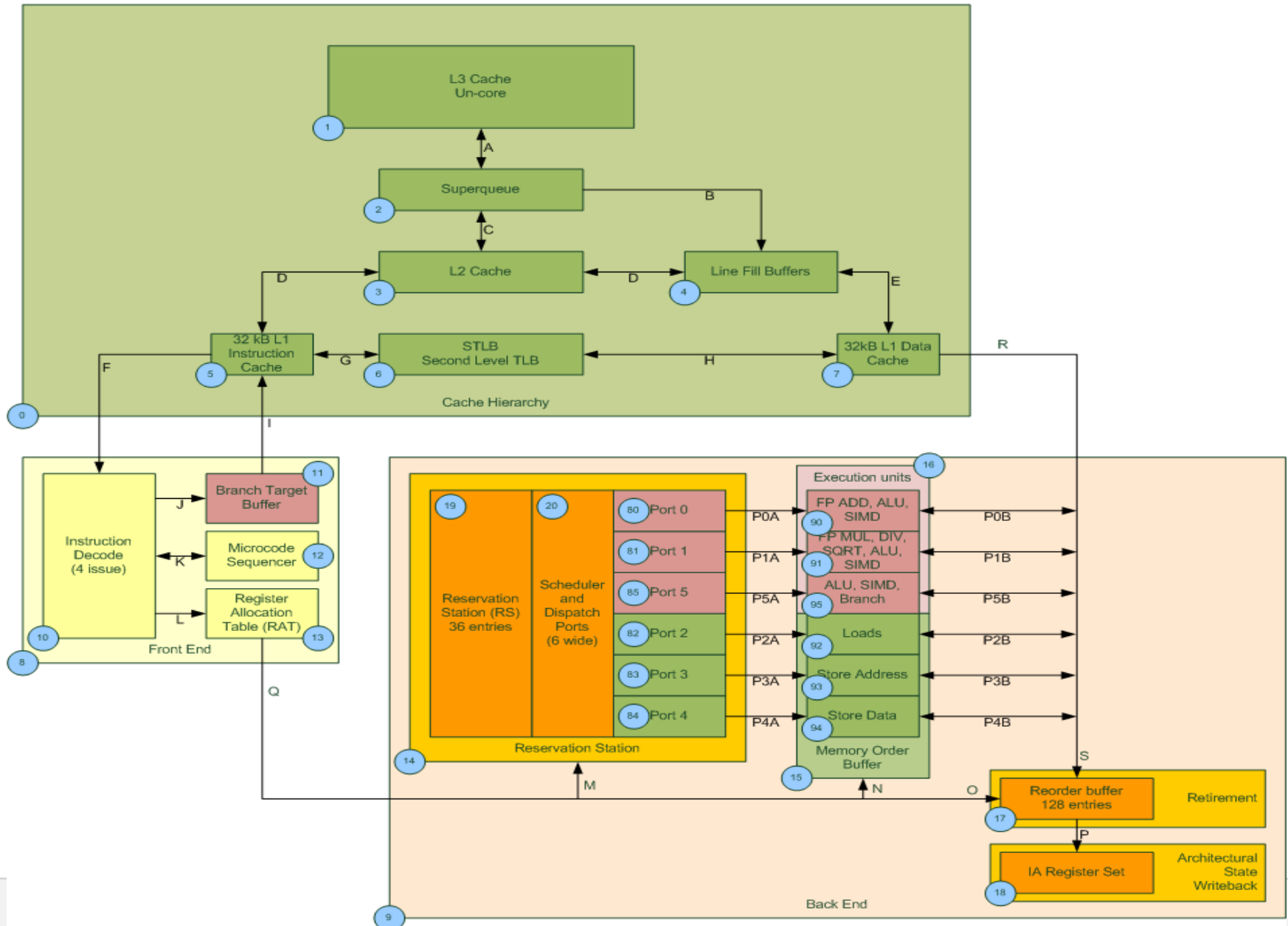
They can be considered for execution

are inputs available?

are they on a known mispredicted branch path?

Artistic Out of Order Pipeline





Scheduling uops at the RS

OOO scheduling treats uops as independent free agents to some extent

once inputs are available they can be sent to execution units (Intel: dispatch, AMD:Issue)

Scheduler/RS is monolithic on Intel

**Scheduler partitioned by function for AMD/IBM-PPC
fp/integer or fp/integer/memory**

Monolithic scheduler can apply all slots for any execution sequence

**but large scheduler is difficult to make
sort of "fully associative" to be as fast as possible**

Scheduling uops at the RS

**Since uops execute "independently",
execution of different hyperthreads is
straightforward**

**Uops can be dispatched speculatively
count uop dispatch for pointer chase as a function
of latency/buffer size**

**as latency of pointer chase loop increases,
we observe the uop dispatch count/iteration increases**

**Makes defining/observing pipeline stalls at
execution very difficult**

**on WSM and SNB
solved on IVB**

Uop execution

Intel processors (prior to HSW) have 6 ports for transferring uops to the execution units

HSW will have 8 ports (see IDF presentation)

Multiple execution units per port

Loads and stores execute on ports 2,3,4

Stores decode into 2 uops

store address, store data

loads/stores need load/store buffers to execute

Dependent uops wait for loads to complete

Dependent uops can forward store data from store buffers in many cases

Thus are not usually blocked by stores

Cacheline movement for loads/stores

Load and stores interact with L1D using load/store buffers

If line is not in L1d it must be retrieved

On first miss a line fill buffer is allocated to deliver line to L1d

Subsequent "secondary misses" will "hit the line fill buffer"

If line misses L2 a request must be sent offcore

allocate a position in "superqueue"

when line returns, it is checked into L1 and L1D

Miss in L3 will cause a request to local integrated memory controller and remote cache/memory controller

Load and store buffers

The interaction with the L1D cache requires load buffers for load operations and store buffers for store operations.

When all load or store buffers are in use, uops flow from the FE will be blocked when a load or store uop tries to issue.

For loads:

Execution is usually dependent on loads completing so execution stops in the BE due to dependency

For Stores:

Execution is not blocked, except in cases of store forward block (ex: load larger than overlapped store). Data must appear in caches in order. Consequently long latency RFO can result in all store buffers being allocated, uop issue being blocked and the RS draining to empty for the given thread.

As these resources are partitioned by the Hyperthreads, their saturation can block uop issue for only one logical core.

Uop Execution

Branch instruction mispredictions are detected at execution

ex: conditional branch depending on value that must be retrieved from dram

Branch is blocked by load

Uops on mispredicted path must be flushed
Correct instruction lines retrieved and pushed through pipeline

Uop Retirement

A uop can be "retired" when all older uops on the same correct execution path have finished

**no mispredicted branches on path to uop
retired means modifications to register values are
committed (aka can be seen in a debugger)
Multiple uops can retire/cycle**

Uop Retirement

Stores can retire prior to data having been written to L1D

data resides in store buffer until written to L1D or write combine buffer and shipped to dram

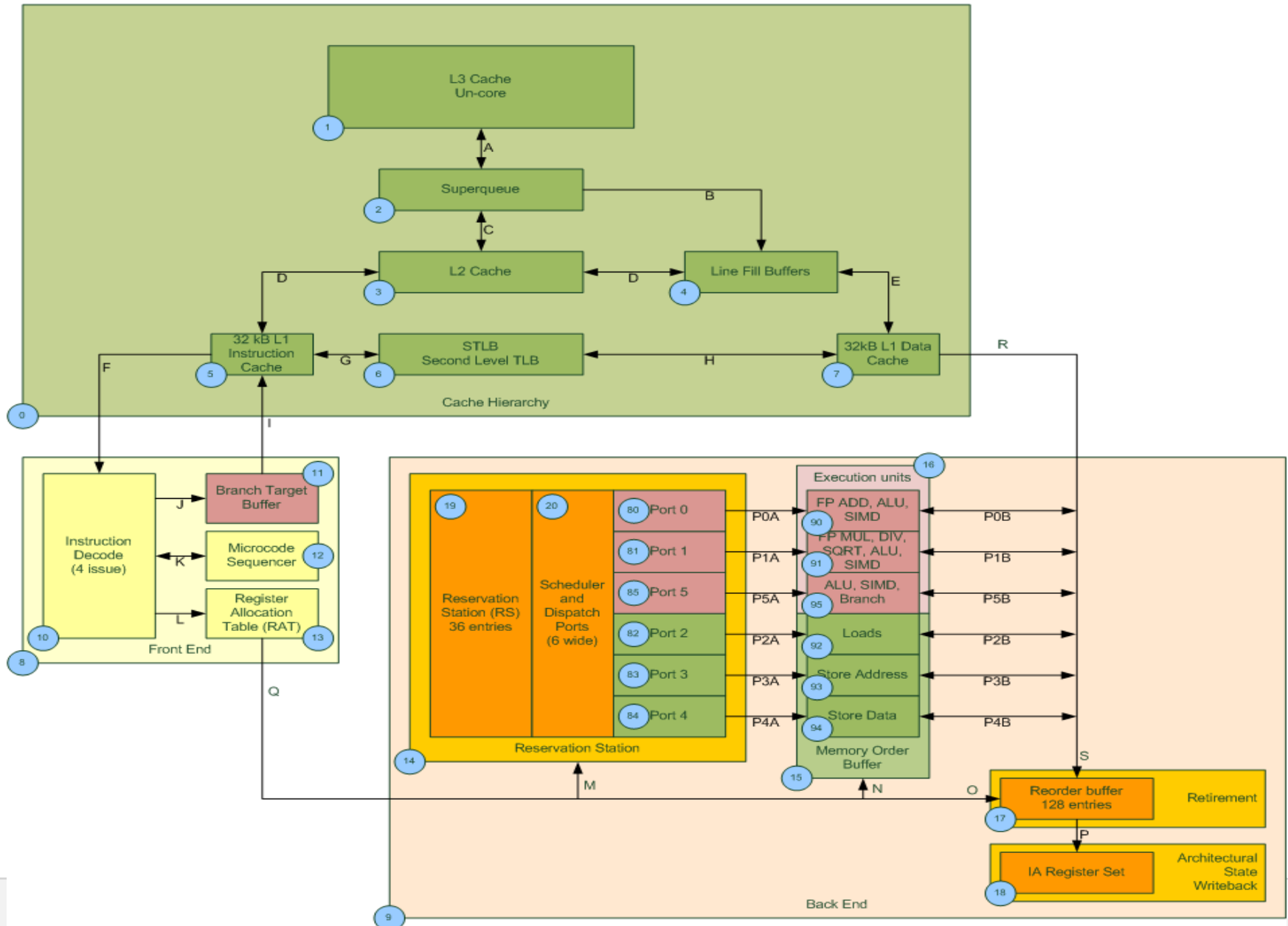
**NT stores write directly to dram
only 1 line movement (no rfo)**

Data must appear in L1D in order

**write to L1D that is blocked by L1D miss blocks
other stores from writing**

Causing store buffers to stay allocated

**when all store buffers are in use, pipeline stalls:
resource stall due to lack of free store buffers**



References

Intel Software development manual and software optimization guide

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

Agner Fog's work:

www.agner.org

<https://code.google.com/p/gooda>

[gooda-analyzer/docs](#)

[gooda-analyzer/kernels](#)

and

anything you can find on the web by some guy named David Levinthal