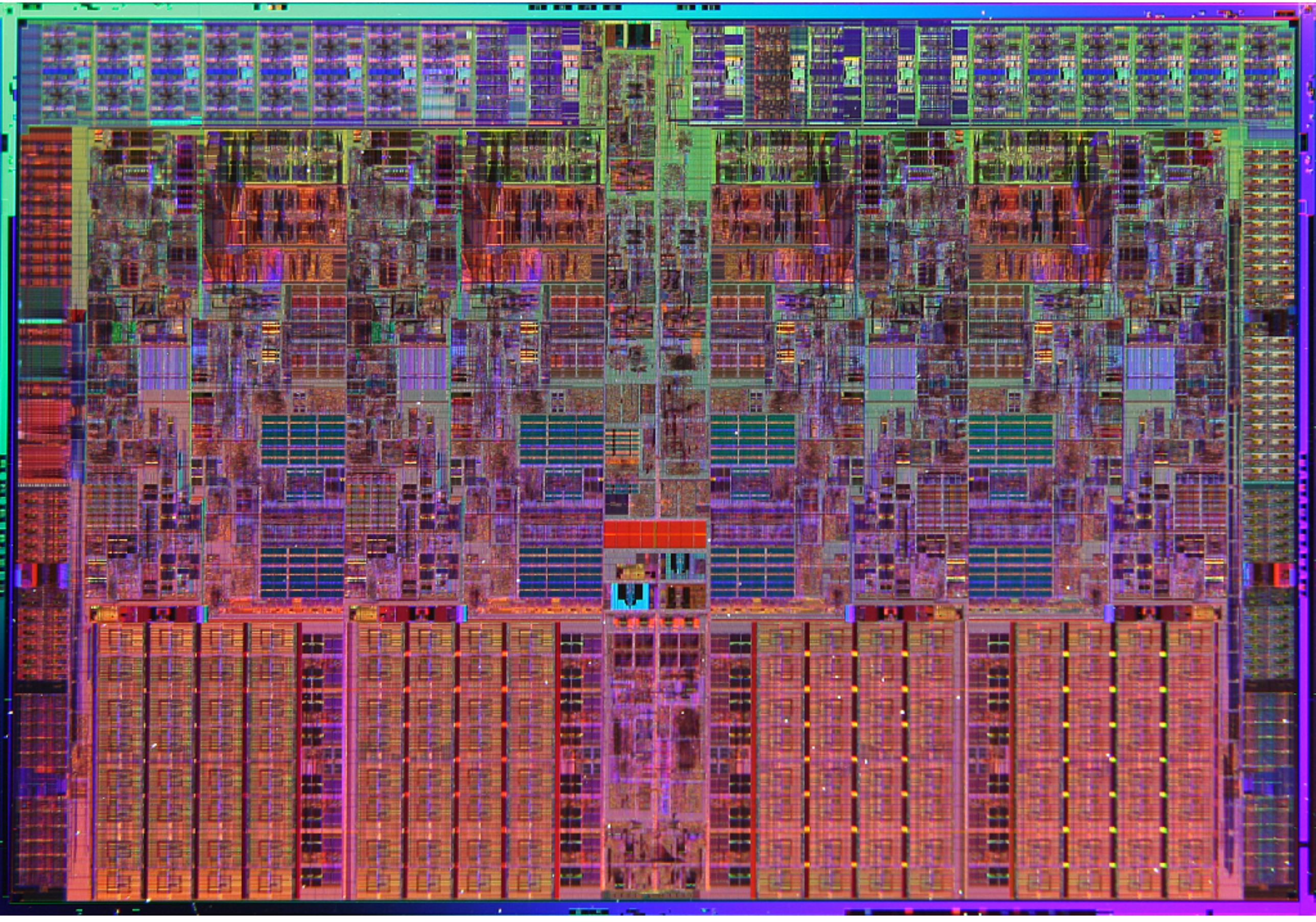
Nehalem - Part 1



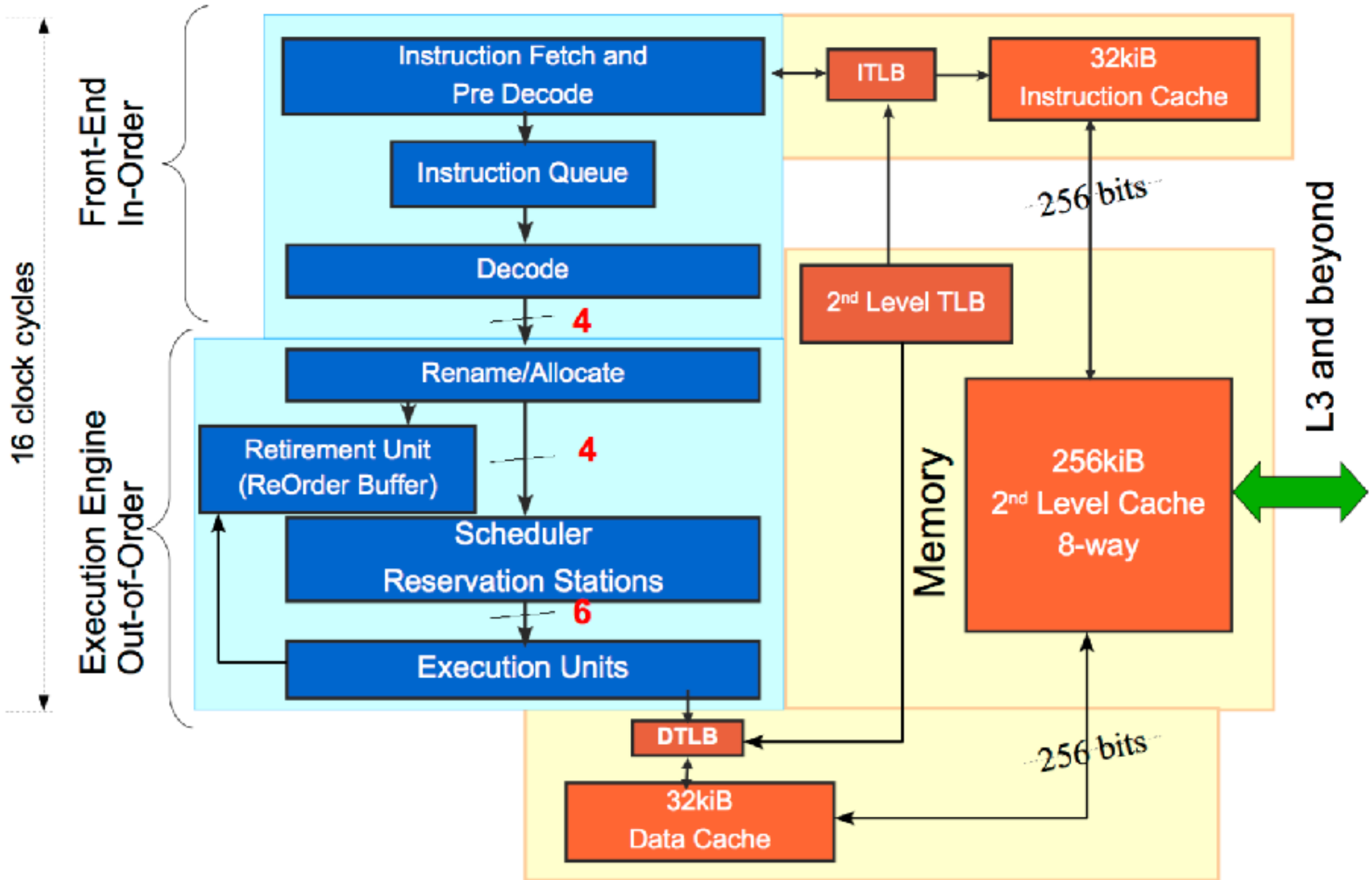
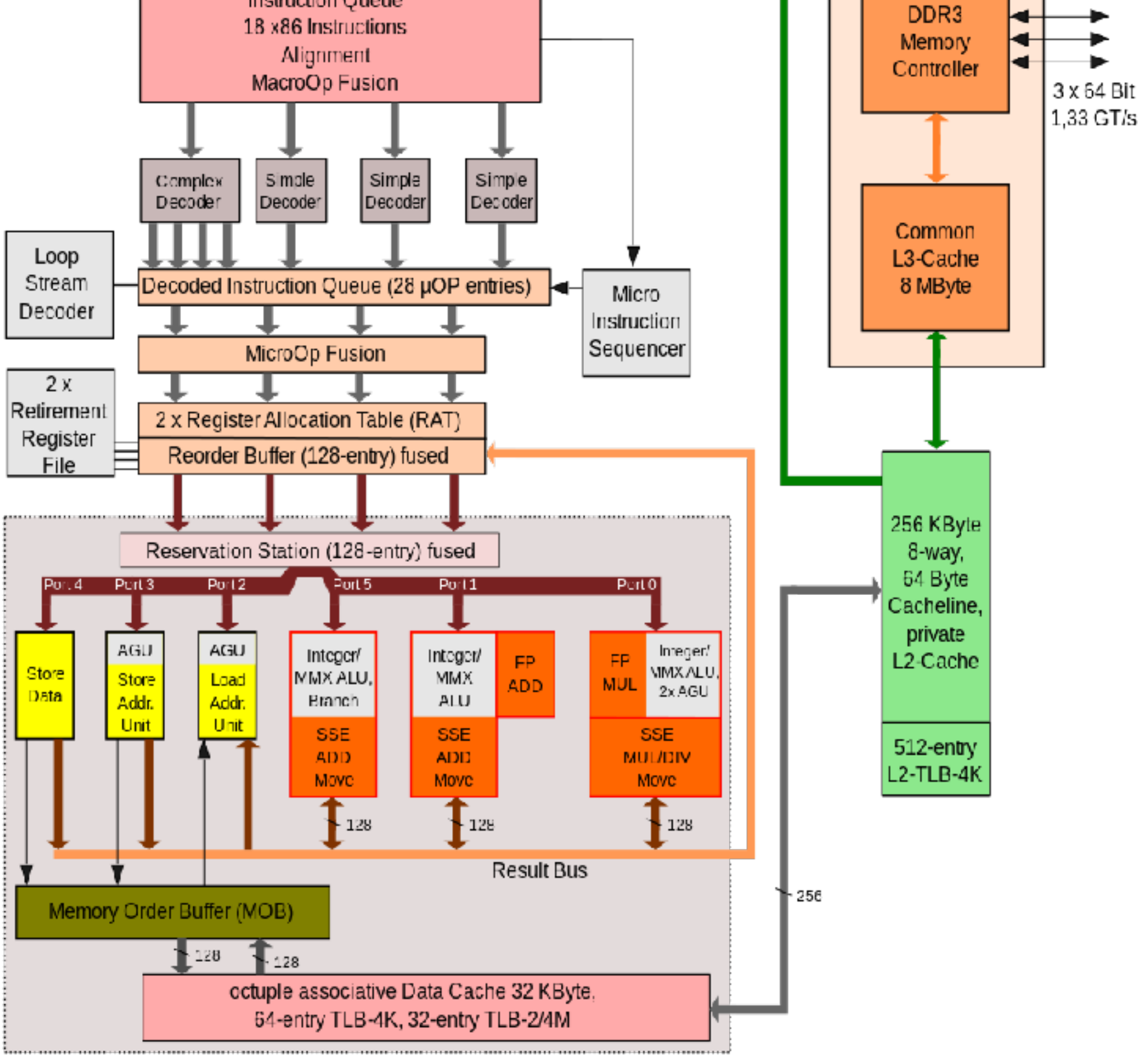


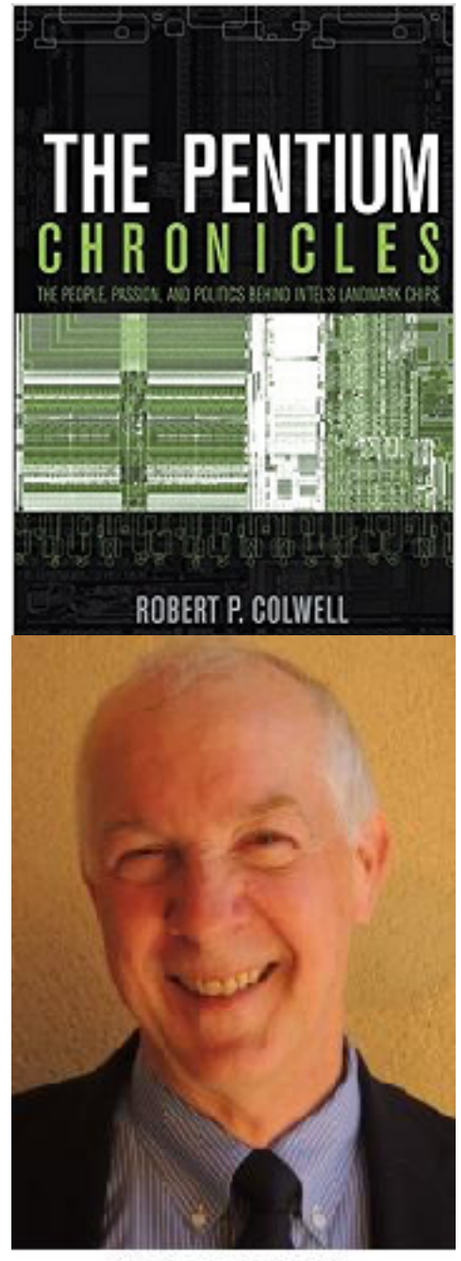
Figure 3: High-level diagram of a Nehalem core pipeline.



GT/s: gigatransfers per second

What is uOp Cracking?

- uOps are components of larger macro ops.
- uOp cracking is taking CISC like instructions to RISC like instructions
- it would be good to crack CISC ops in parallel
- dynamically adjust how we crack a sequence
- memorize the result with a cache
- and re-order uOps into program order by making that cache store *traces*.



Robert Colwell

What is uOp Fusion?

- Fuse certain uOps into sequences.
- MUL r1, r2 -> r3 ; ADD r4, r4, r3
  - w/o Fuse: input { r1, r2, r4 } output { r3, r4 }
  - w/Fuse: input { r1, r2, r4 } output { r4 }

May 10th - Computer History Museum visit

Tentative but *highly likely*

What is Memory Dependence Prediction?

- The problem
  - MUL r10,r11, -> r4 ; r10 == 0
    STORE r1, @(r4)
    BLAH
    LOAD_CONSTANT R5, #0
    LOAD r3, @(r5) ;; but r4 == r5!
    ADD r3, r6 -> r7
- Bloom filter solution

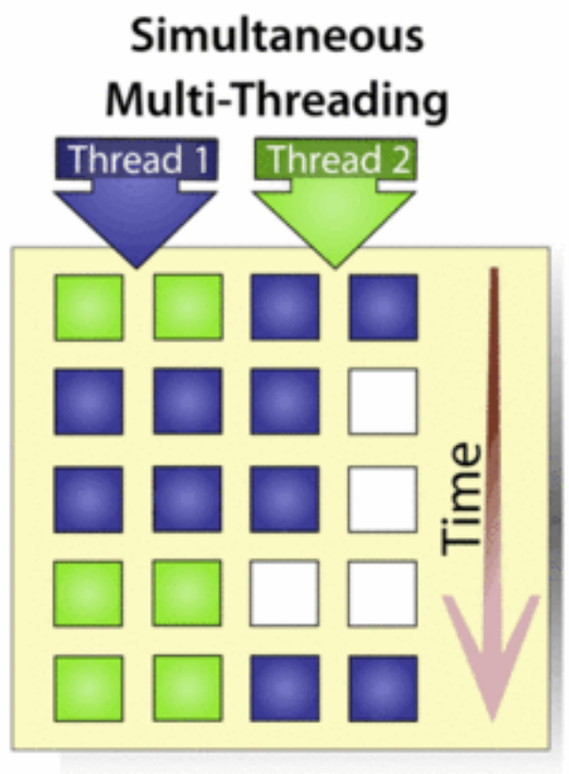
FP

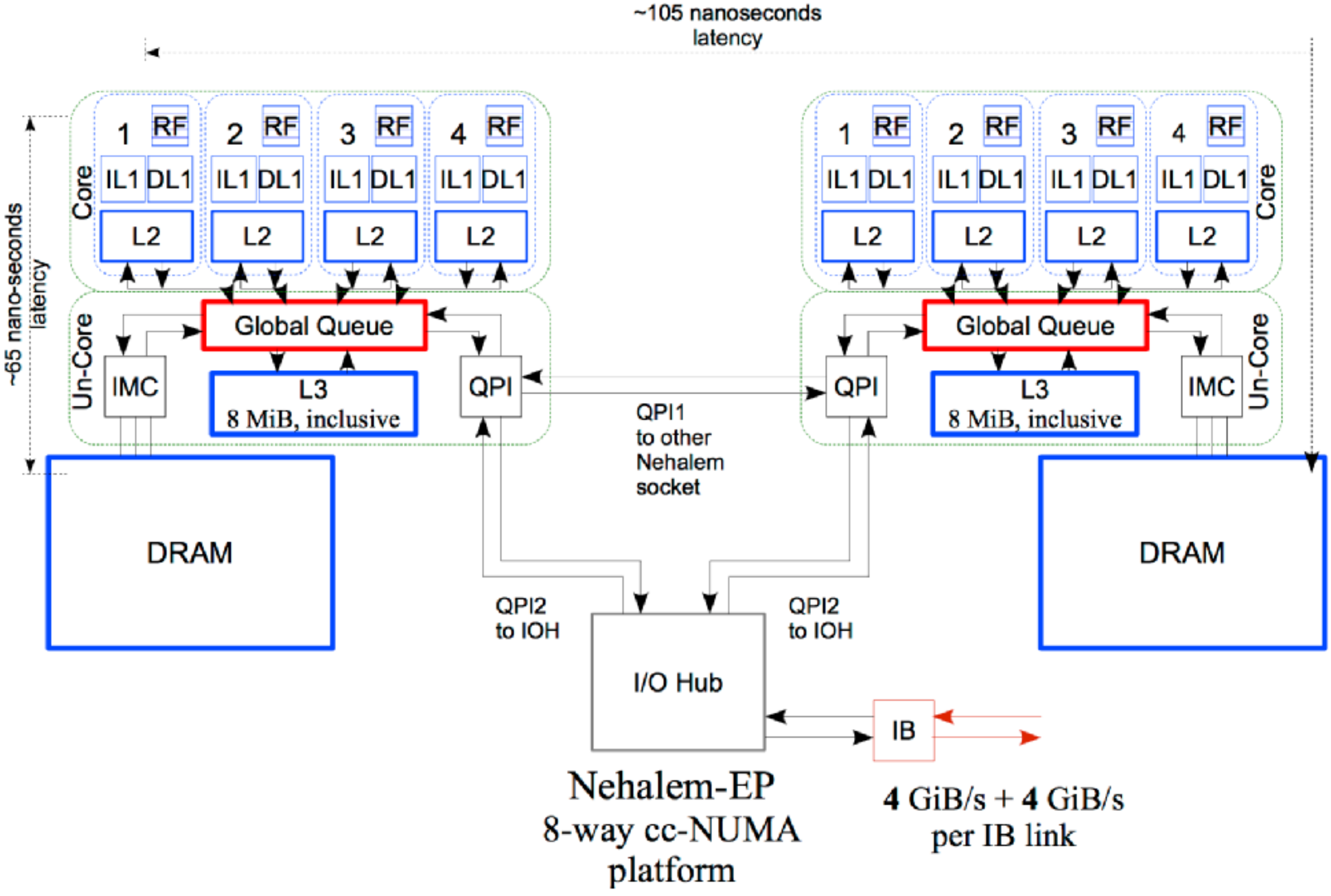
- IEEE

- SIMD (XMMS/MMX/blah blah)

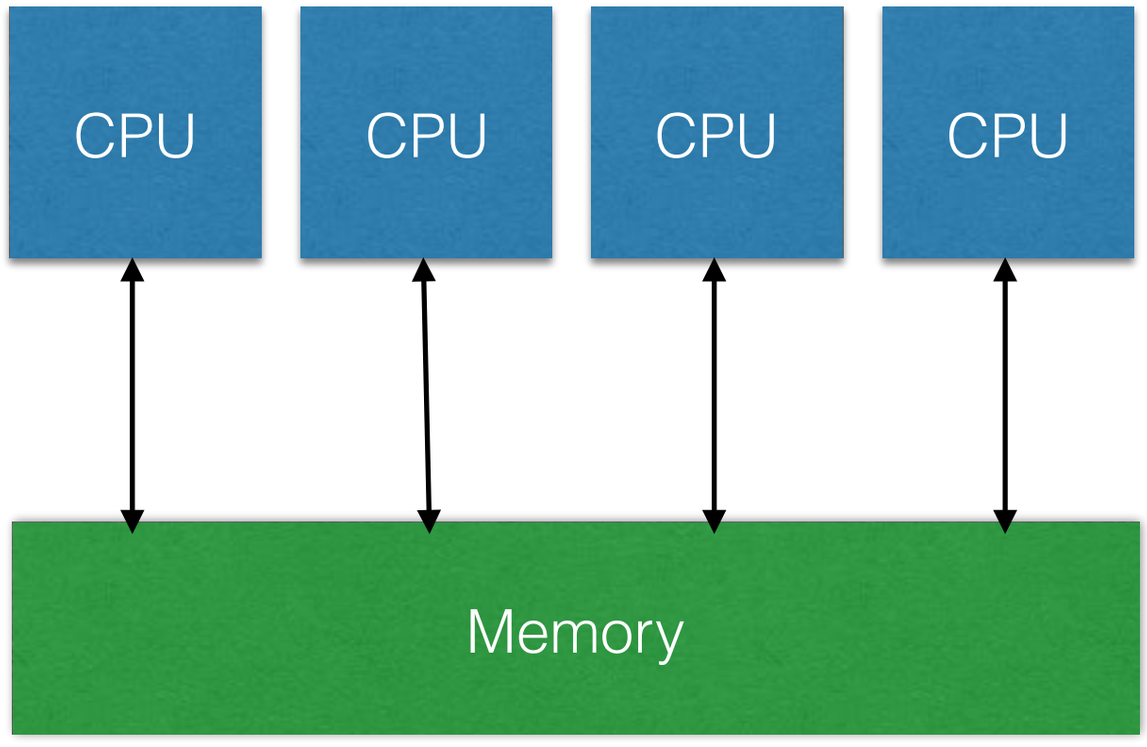
How does SMT work?

- Multiprocessing within one core
- Two program counters
- Two register sets?
  - Two maps
- Two branch history registers
- Sharing support for lots of stuff
  - Cache ports, branch predictor, completion, etc, etc

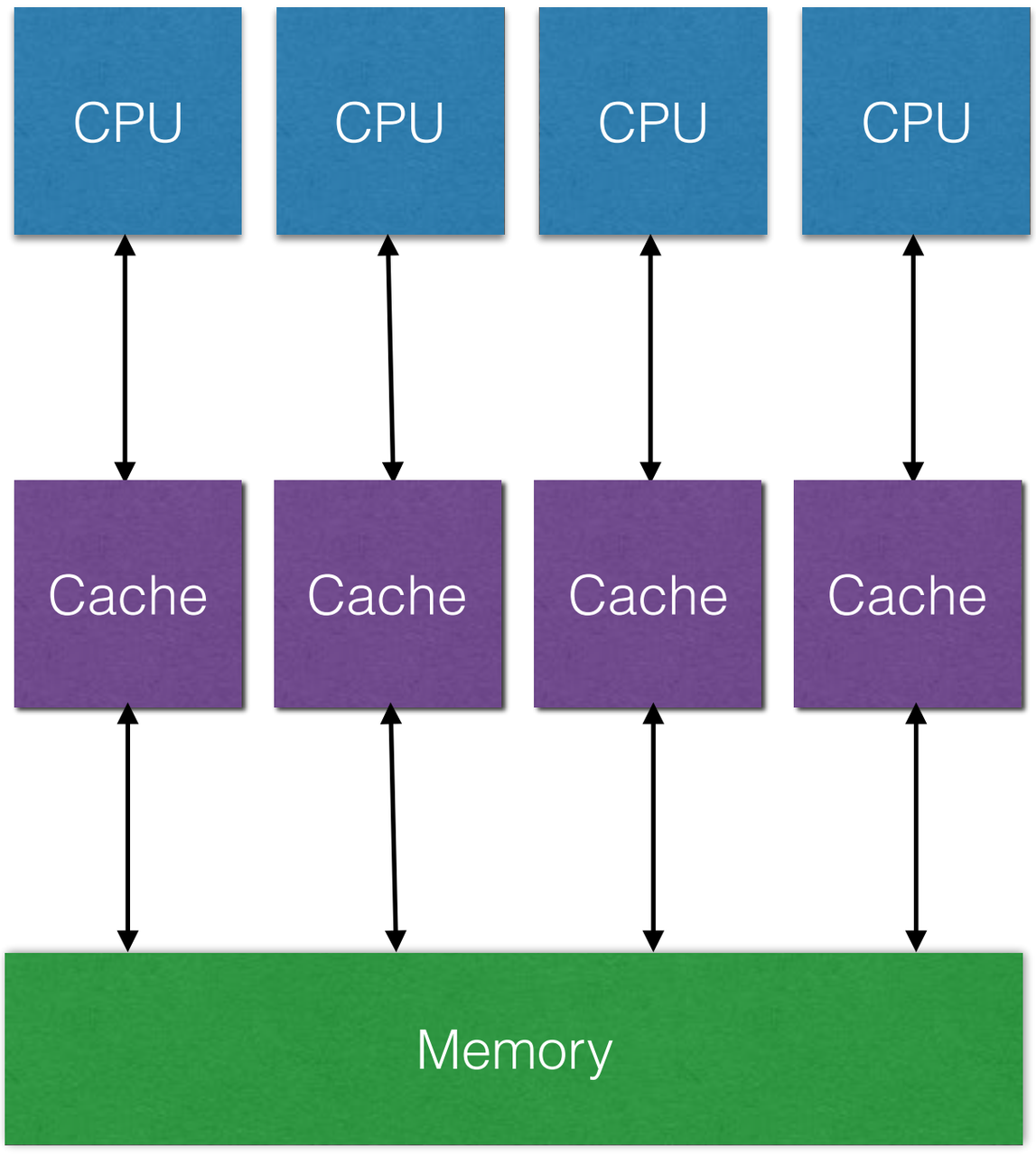



~105 nanoseconds
latency
~65 nano-seconds
latency
Core
1
RF
IL1
DL1
L2
2
3
4
Un-Core
Global Queue
IMC
L3
8 MiB, inclusive
QPI
QPI1
to other
Nehalem
socket
DRAM
QPI2
to IOH
I/O Hub
IB
Nehalem-EP
8-way cc-NUMA
platform
4 GiB/s + 4 GiB/s
per IB link

How programmers want to see the world



How programmers need to see the world



Basic MESI Protocol

Local Read

Remote Read

Local Write

Remote Write

Invalid

Exclusive

Modified

Shared

Consistency

- Unfortunately for you, hardware doesn't behave like you think.
- Up until now, we've been writing code that will execute correctly on x86, ARM, etc.
- Broadly speaking, there are four consistency models in the world:
    - sequential, how you think concurrency works until you're told it's not so
    - processor, how x86 works, except for where it doesn't
    - release, how ARM works and a few other esoteric ISAs
    - scope (used for GPUs).
- The consistency model presented to the programmer is a function of the **language.** It is more or less easy to implement that consistency model on different hardware.
    - Easy for an academic to say. In practice, it's the wild west out there and you have to know what you are working with.
- C11 and C++11 are what is known as SC for DRF programs; or sequentially consistent for "data race free" programs.

Say what?

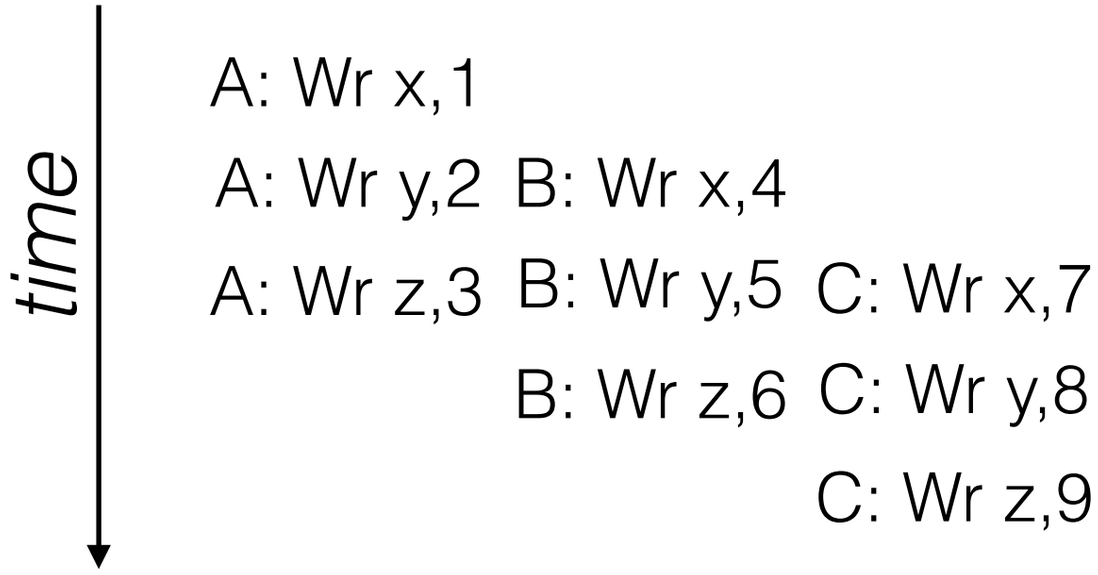
```
int a = 1;
int b = 2;

thread1() {
  a = 3;
  b = 4;
}

thread 2 {
  printf(“%d”, b);
  printf(“%d”, a);
}
```

What are the possible outputs?

Sequential Consistency

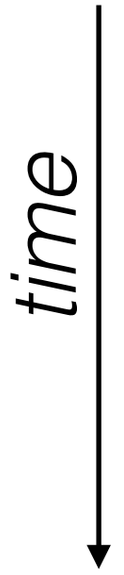


A: Wr x,1
A: Wr y,2
B: Wr x,4
B: Wr y,5
A: Wr z,3
C: Wr x,7
C: Wr y,8
B: Wr z,6
C: Wr z,9

All processors see all writes in the *same* order

This is just one possible SC order.

Processor Consistency



A: Wr x,1
A: Wr y,2
A: Wr z,3

B: Wr x,4
B: Wr y,5
B: Wr z,6

C: Wr x,7
C: Wr y,8
C: Wr z,9

A: Wr x,1
A: Wr y,2
B: Wr x,4
B: Wr y,5
A: Wr z,3
C: Wr x,7
C: Wr y,8
B: Wr z,6
C: Wr z,9

How A can see the world

A: Wr x,1
B: Wr x,4
A: Wr y,2
B: Wr y,5
C: Wr x,7
A: Wr z,3
B: Wr z,6
C: Wr y,8
C: Wr z,9

How B can see the world

A processor sees *all* writes from another processor in the order that processor performs them

Release Consistency

time

A: Wr x,1
A: Wr y,2
A: Wr z,3

B: Wr x,4
B: Wr y,5
B: Wr z,6

C: Wr x,7
C: Wr y,8
C: Wr z,9

A: Wr x,1
A: Wr y,2
B: Wr x,4
B: Wr y,5
A: Wr z,3
C: Wr x,7
C: Wr y,8
B: Wr z,6
C: Wr z,9

How A can see the world

A: Wr x,1
B: Wr x,4
A: Wr z,3
A: Wr y,2
B: Wr y,5
C: Wr x,7
B: Wr z,6
C: Wr z,9
C: Wr y,8

How B can see the world

More or less, release consistency promises nothing without a fence

What would Brian Boitano do?

- **Don’t write code that synchronizes outside of existing synchronization primitives**
    - Or if you must, wrap your shared memory accesses with fences: MFENCE (fence it all), LFENCE (load fence), or SFENCE (store-fence).
- In my experience, this kind of stuff will not make your code run fast. The problems that hinder parallel code bases are always much bigger than the micro-scale synchronization that is used.
    - **Go for maintainability and don’t check in nor allow anyone else to check in code that uses subtle synchronization.**

Parallel Programming Primer

Why do we have more than one CPU?

- Only so many transistors
- So we can be lazier programmers
- So we can do more than one thing at a time
- Heat
- Cheaper to scale horizontally than vertically
- Speeds of a single core stopped increasing

Topics for today/next week

- Threads
- Locks (Mutex)
- Semaphore
- Reader/Writer lock
- Condition variables
- Barrier
- Monitors
- Lock free (“Live free or Die!”)
- Consistency
- User-mode threads
- OpenMP
- Transactions

What is a thread?

- Lightweight process
- Stack + IP
- Sequential execution of a list of instructions
- Something you can map onto a processor

- Hardware: IP, register set, misc context, address space
- Language: (hopefully) a well defined execution context
- OS: something you can schedule and run

Common bugs with threads

- Failure to join
- Race on start state
- Failure to synchronize on library calls (POSIX 1 vs. POSIX 1c (1996))
- Using shared memory incorrectly (much more on this)

Common performance problems with threads

- Too fine
- Too coarse
- Too few
- Too many
- Move around too much (no affinity)

What is a lock?

- Mechanism to prevent other threads from using a resource
- Piece of shared memory to achieve mutual exclusion
- Meeting point for threads
- data-structure to maintain ownership
- Something that can be “owned” by only one thread

A (broken) lock implementation

```
void lock(int *the_lock) {
 while (*the_lock == 1)
  ;
 *the_lock = 1;
}



void unlock(int *the_lock) {
 *the_lock = 0;
}
```

How are locks implemented?

- Core necessity: *Either* a bit of private state per acquirer (Petersen lock), ***Or***, an **atomic read/write** operation (how we tend to do it in SM systems)

- Processors tend to support a variety of atomic operations useful for constructing locks:

  - LOCK XCHG # Exchange

    - Swap a register and memory location value

  - LOCK CMPXCHG # Compare and xchange

    - Write to memory a register value if and only if the memory is equal to a given value

A lock implementation

```
void lock(int *the_lock) {
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1)
  ;
}
```

```
void unlock(int *the_lock) {
 *the_lock = 0;
}
```

A slightly better implementation

```
void lock(int *the_lock) {
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1)
  asm volatile ("pause");
}
```

A slightly better implementation

```
#define MAX_BACK_OFF    (1<<12)

void lock(int *the_lock) {
 int back_off = 1, i;
 while (__sync_val_compare_and_swap(the_lock, 0, 1)
  == 1) {

  for (i = 0; i < back_off; i++)
    asm volatile ("pause");

  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

A slightly slightly better implementation

```
#define MAX_BACK_OFF   (1<<12)

void lock(int  *the_lock) {
 int back_off = 1, i;
 while (1) {
  // TEST
  while (*the_lock != 1) {
   asm volatile (“pause"); // Tell the CPU we are spinning
   asm volatile ( ::: "memory"); // address expose
  }
  // TEST AND SET
  if (__sync_val_compare_and_swap(the_lock, 0, 1) == 0)
     return;
  // BACK OFF
  for (i = 0; j < back_off; i++)
    asm volatile ("pause");
  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

An even slightly slightly better implementation

```
#define MAX_BACK_OFF   (1<<12)
#define CACHE_LINE_SIZE (64)

typedef union _lock {
    int lock_state;
    char padding[CACHE_LINE_SIZE];
} lock;

void lock(lock *the_lock) {
 int back off = 1, i;
 while (1) {
  while (the_lock->lock_state != 1) {
   asm volatile ("pause");
    asm volatile ( ::: “memory");
  }
  if (__sync_val_compare_and_swap(the_lock->lock_state, 0, 1) == 0)
   return;
  for (i = 0; j < back_off; i++)
    asm volatile ("pause");
  if (back_off <= MAX_BACK_OFF)
    back_off = back_off << 1;
}
```

Common bugs with locks

- Failure to use one
- Failure to initialize them
- Failure to acquire all of them that you need
- Failure to unlock
- Releasing them too early
- Failure to acquire in a consistent order (deadlock)
- Acquiring them twice
- Releasing them when a thread ends abnormally
- Holding a lock over blocking I/O that may block “indefinitely”

You should never deadlock

- It's possible to write a piece of code known as a "lock witness".
- Group locks into classes.
- Classes can only be acquired in order
- An acquisition of locks out of order, *regardless of whether a deadlock occurred,* is detectable and should be signaled to the developer.
- Every large project needs a lock witness. Go take it from FreeBSD.

Common performance issues with locks

- Holding them for too long a time
- Holding them for too short a time
  - (acquiring/release the same lock in an inner loop)
- Holding more of them than you actually need
- Spinning when you should be queueing
- Queuing when you should be spinning
- Multiple locks in the same cache-line

Reader/Writer locks

- Basic idea: some data-structures support concurrent reads, but not concurrent writes.
  - hash-tables, trees, maps, whatev's.

- A reader/writer lock permits multiple readers but only one writer.
  pthread_rwlock_rdlock(pthread_rwlock_t *);
  pthread_rwlock_wrlock(pthread_rwlock_t *);
  pthread_rwlock_unlock(pthread_rwlock_t *);

A good interview question…

Implement a *fair* reader/writer lock

Why is this a hard question?

How would you solve it?

Condition Variables

- Probably the least understood but most important synchronization primitive there is.

- pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *lock);
    - Thread **must** hold the specified lock. The thread **releases** the lock and is **blocked** until the condition is **signaled**.

- pthread_cond_signal(pthread_cond_t *cond);

- pthread_cond_broadcast(pthread_cond_t *cond);

Condition variables

- Condition variables are important because they all you to synchronize without spinning, and hence wasting resources.

- Condition variables can be extremely challenging for programmers to use but if you follow two simple rules then you will be fine:

  - 1) **A signal before wait is lost.** Always acquire the mutex the wait will wait on, and then signal.

  - 2) When you are signaled, you are **not necessarily assured the condition you really want to be true is true**; it depends on the condition, the number of waits, what they do, etc.

Barriers

- Barriers synchronize a *group* of threads.
- Threads are stopped until all threads that should enter the barrier have done so.
- pthread_barrier_init(pthread_barrier_t *barrier, attr, unsigned int count);
- pthread_barrier_wait(barrier);

Barriers, continued

- Should barriers support a barrier_signal or barrier_broadcast, “early release” mechanism?

- Should barriers support a time-out?

Common bugs/issues with barriers

- A wayward thread never enters
  - Perhaps it's waiting on a lock or data to be produced from a thread that has already entered the barrier!
- *Advice:* Barriers are useful at the mesoscale of your code.
  - If you enter the barrier from different points in your code, think hard, but it still might be ok
  - If you enter a barrier after executing a giant section of code, think hard, but it still might be ok.

Lock-Free Algorithms

- There is an important concept in parallel programing, or rather, a concept some people think is important, known as lock-free algorithms

  - *Lock-free does not mean synchronization free*

- Lock-free generally means a data-structure is designed such that synchronization becomes inherent in the manipulation of the data.

- I'm pushing 1MM lines+ in my lifetime so far. I do not write lock-free data-structures. I do write "lock free" flags now and then but less so as I get older and hopefully wiser.

Lock-free issues

- Just being lock-free does not make you fast
- Lock-free is very challenging to get correct. Think of it as the extreme end of super-fine-grained locking
- Lock-freedom *does not mean* wait-freedom. Wait-free algorithms (which are necessarily lock-free) do exist for some things
- People still write papers that appear in top places on lock free algorithms. Frankly I think they are bit like doing algorithmic research on quantum computing. Hard, and maybe relevant some day, but not so much now. (But then again, I'm starting to get back into QC…)