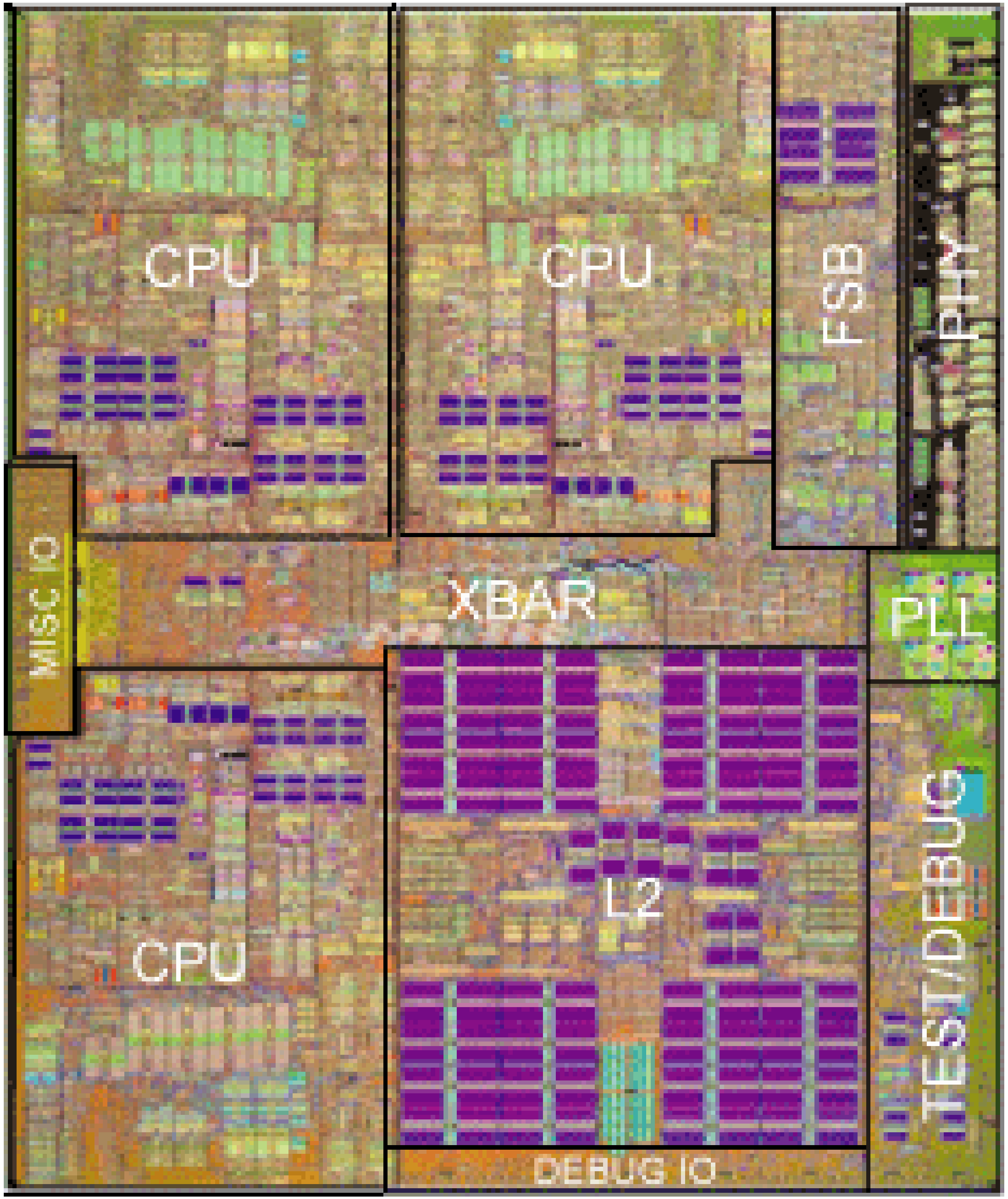


XBox 360

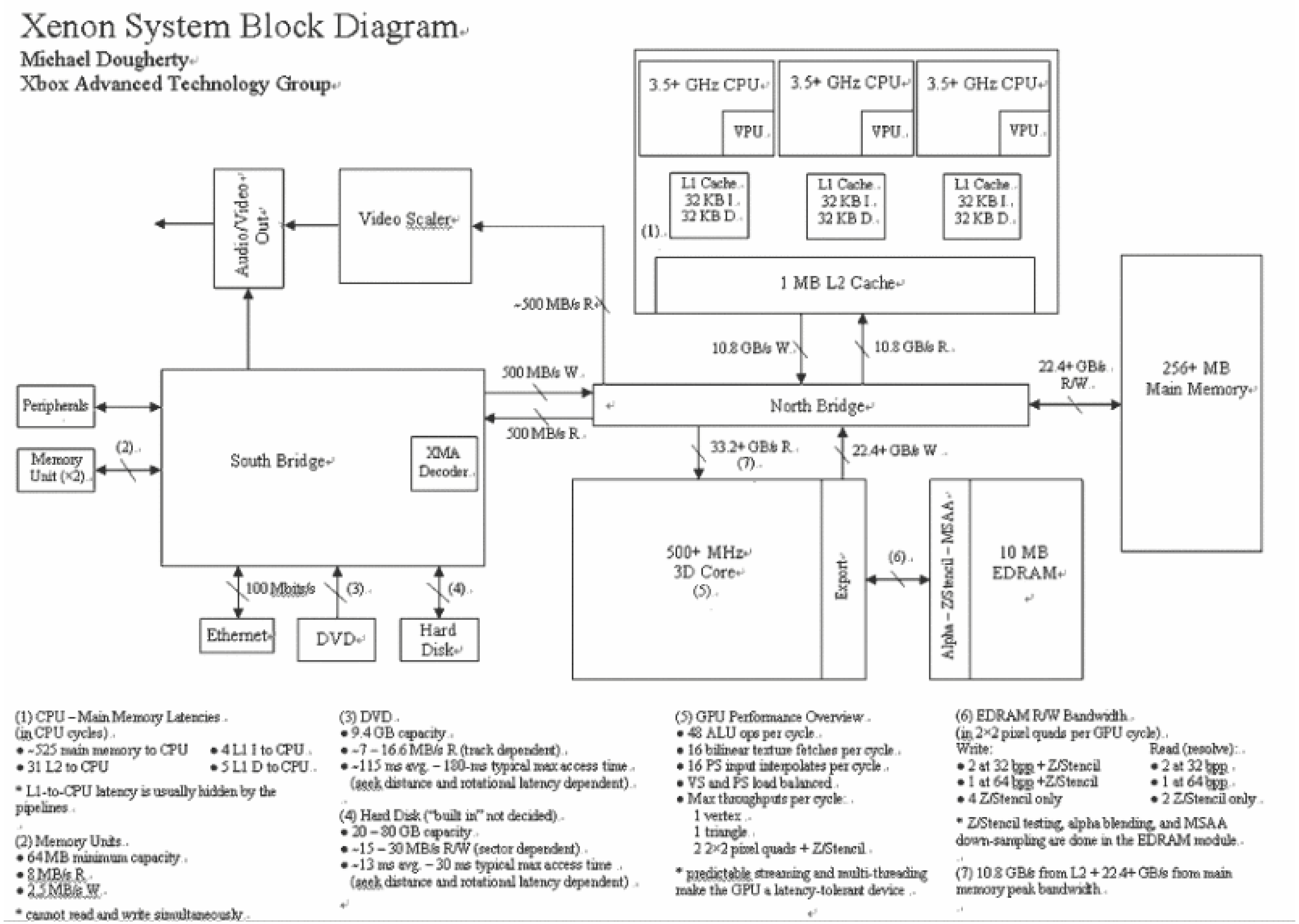
Components

- 3 PowerPC cores
  - 2 threads pr core
  - 3.2 GHz
- R500 Graphics Processor
  - 500MHz
  - 48 Pipelines
- 512 MB RAM

Xenon

- 3-Way Symmetric Multi-Processor
  - IBM PowerPC Architecture®
  - Specialized Function VMX
  - 3.2GHz
    - Shared 1 MByte L2
    - Front Side Bus / PHY – 21.6 GB/sec
    - Phase Locked Loops
- 165 M Transistors
  - IBM – 90nm SOI

Layout



Xenon System Block Diagram

Michael Dougherty
Xbox Advanced Technology Group

(1) CPU – Main Memory Latencies
(in CPU cycles)

- ~525 main memory to CPU
- 31 L2 to CPU
- 4 L1 I to CPU
- 5 L1 D to CPU

* L1-to-CPU latency is usually hidden by the pipelines

(2) Memory Units

- 64 MB minimum capacity
- 8 MB/s R
- 2.5 MB/s W

* cannot read and write simultaneously

(3) DVD

- 9.4 GB capacity
- ~7 – 16.6 MB/s R (track dependent)
- ~115 ms avg. – 180-ms typical max access time (seek distance and rotational latency dependent)

(4) Hard Disk ("built in" not decided)

- 20 – 80 GB capacity
- ~15 – 30 MB/s R/W (sector dependent)
- ~13 ms avg. – 30 ms typical max access time (seek distance and rotational latency dependent)

(5) GPU Performance Overview

- 48 ALU ops per cycle
- 16 bilinear texture fetches per cycle
- 16 PS input interpolates per cycle
- VS and PS load balanced
- Max throughputs per cycle:
  1 vertex
  1 triangle
  2 2×2 pixel quads + Z/Stencil

* predictable streaming and multi-threading make the GPU a latency-tolerant device

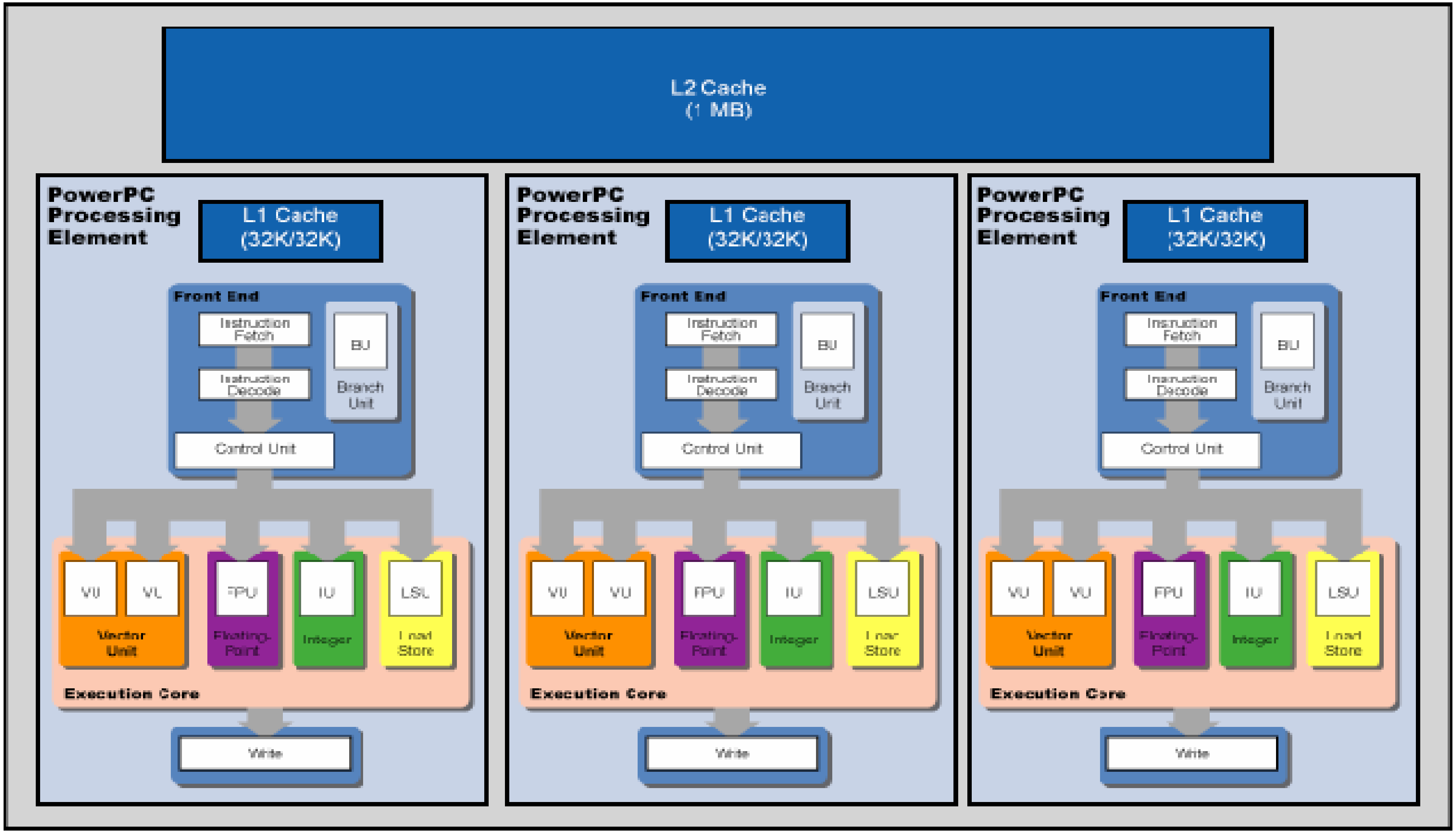
(6) EDRAM R/W Bandwidth
(in 2×2 pixel quads per GPU cycle)

Write:	Read (resolve):
• 2 at 32 bpp + Z/Stencil	• 2 at 32 bpp
• 1 at 64 bpp +Z/Stencil	• 1 at 64 bpp
• 4 Z/Stencil only	• 2 Z/Stencil only

* Z/Stencil testing, alpha blending, and MSAA down-sampling are done in the EDRAM module

(7) 10.8 GB/s from L2 + 22.4+ GB/s from main memory peak bandwidth

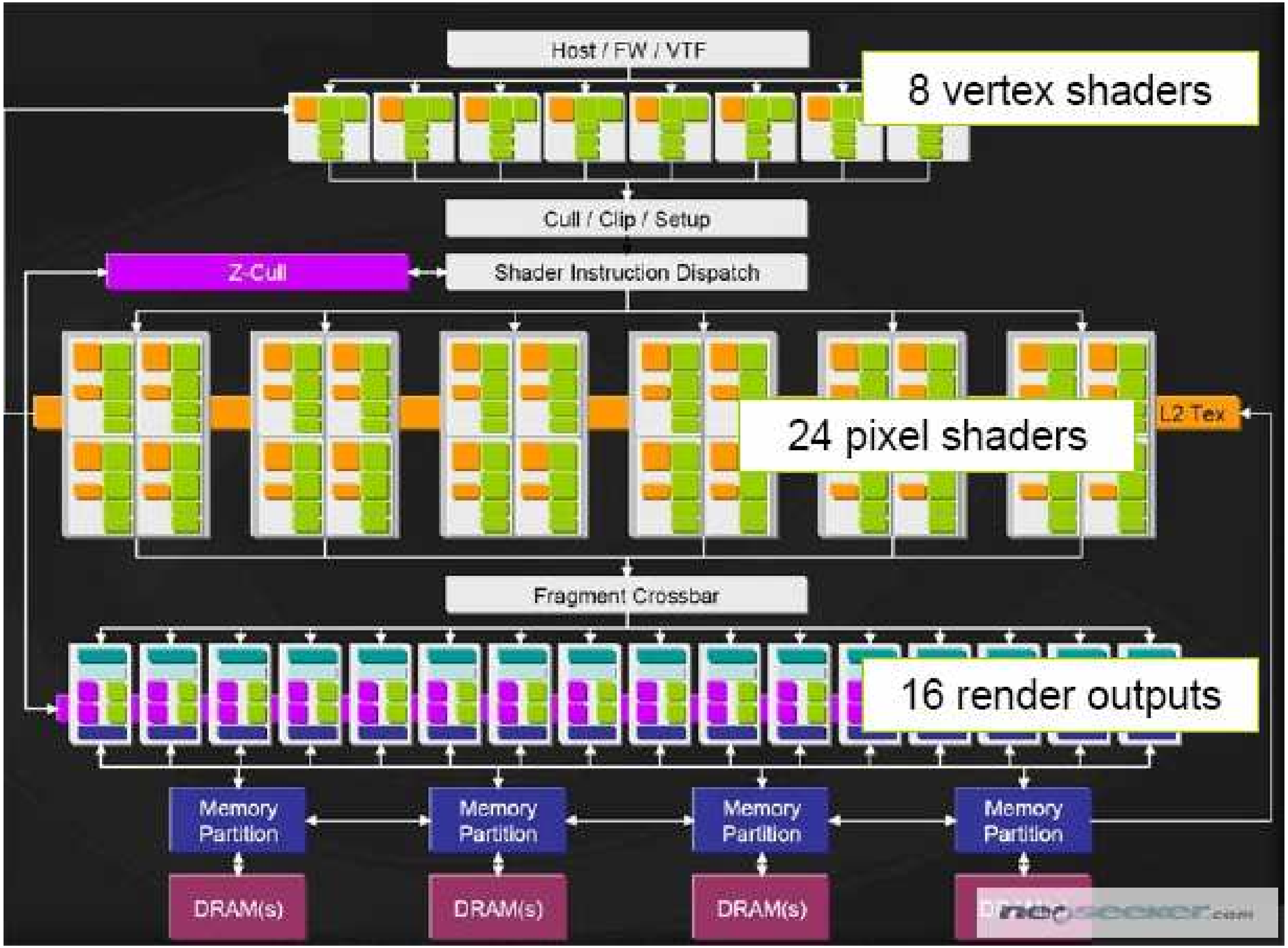
Layout of the Xenon



GPU

- Custom ATI Graphics Processor
  - 10MB DRAM
  - 48-way parallel floating point
  - Unified shader architecture
  - 500 million triangles per sec
  - 16 gigasamples/sec
  - 48 billion shader operations/sec

GPU


Host / FW / VTF
8 vertex shaders
Cull / Clip / Setup
Z-Cull
Shader Instruction Dispatch
24 pixel shaders
L2 Tex
Fragment Crossbar
16 render outputs
Memory Partition
Memory Partition
Memory Partition
Memory Partition
DRAM(s)
DRAM(s)
DRAM(s)
DRAM(s)

Memory

- 512 MB of 700MHz GDDR3 RAM – unified memory architecture
- 22.4 GB/s interface bus bandwidth
- 256 GB/s memory bandwith to EDRAM
- 21.6 GB/s front-side bus

Multithreading Tips & Tricks

Jonathan Haas

Software Design Engineer

Xbox Advanced Technology Group

Why Multithread?

- Necessary to take full advantage of Xbox 360 CPU
- Necessary to take full advantage of modern PC CPUs
- Other platforms might benefit from multithreading as well
- What do all these things have in common?
  - 2D sprite-based graphics
  - Waveform synthesized audio
  - 16-bit pointers
  - Single-threaded games

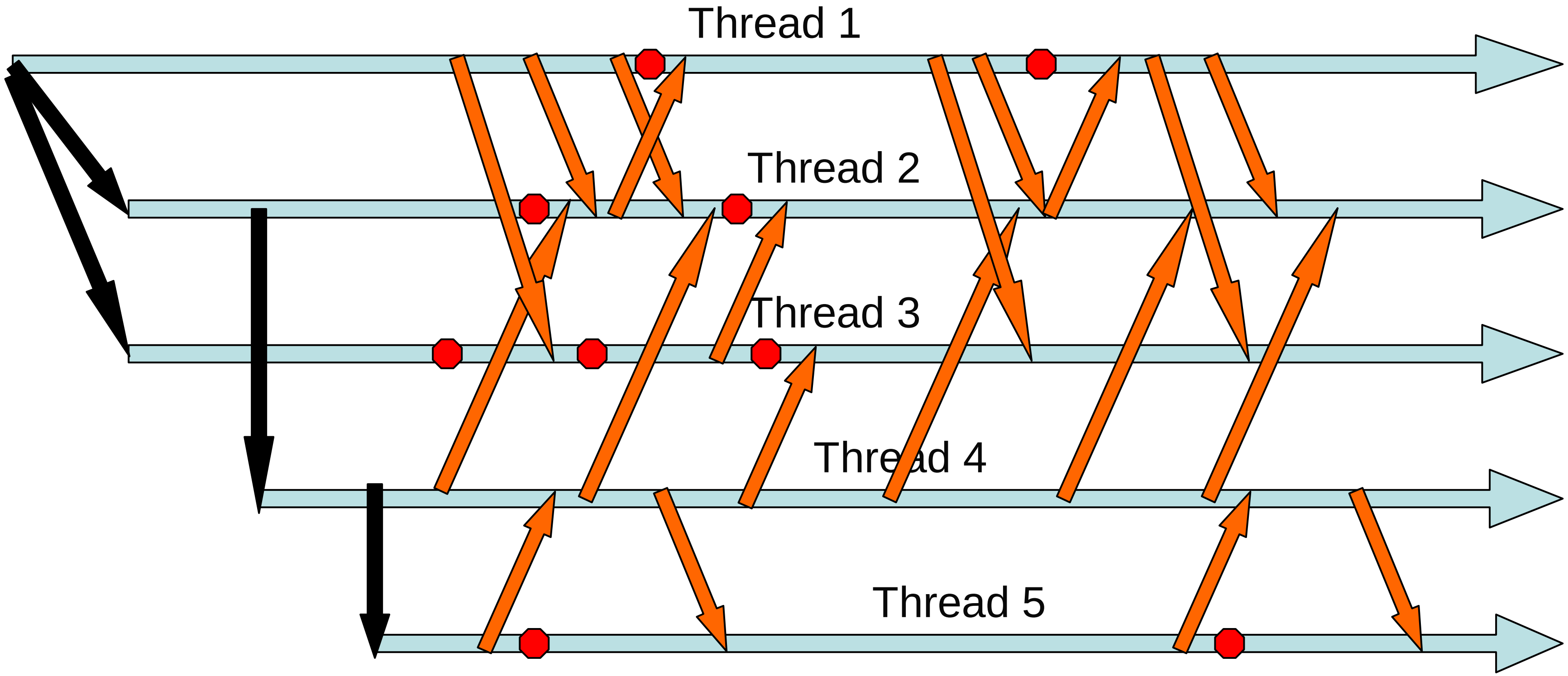
Agenda

- Designing for multiple threads
- Thread basics
- Synchronization
- Lockless programming

Design for Multithreading

- Bad multithreading can be worse than no multithreading, so design intelligently
- Two major paradigms:
  - Symmetric threads
    - Job queues
  - Asymmetric threads
    - Task-oriented threading
- Well-designed systems use both

Bad Multithreading



Thread 1

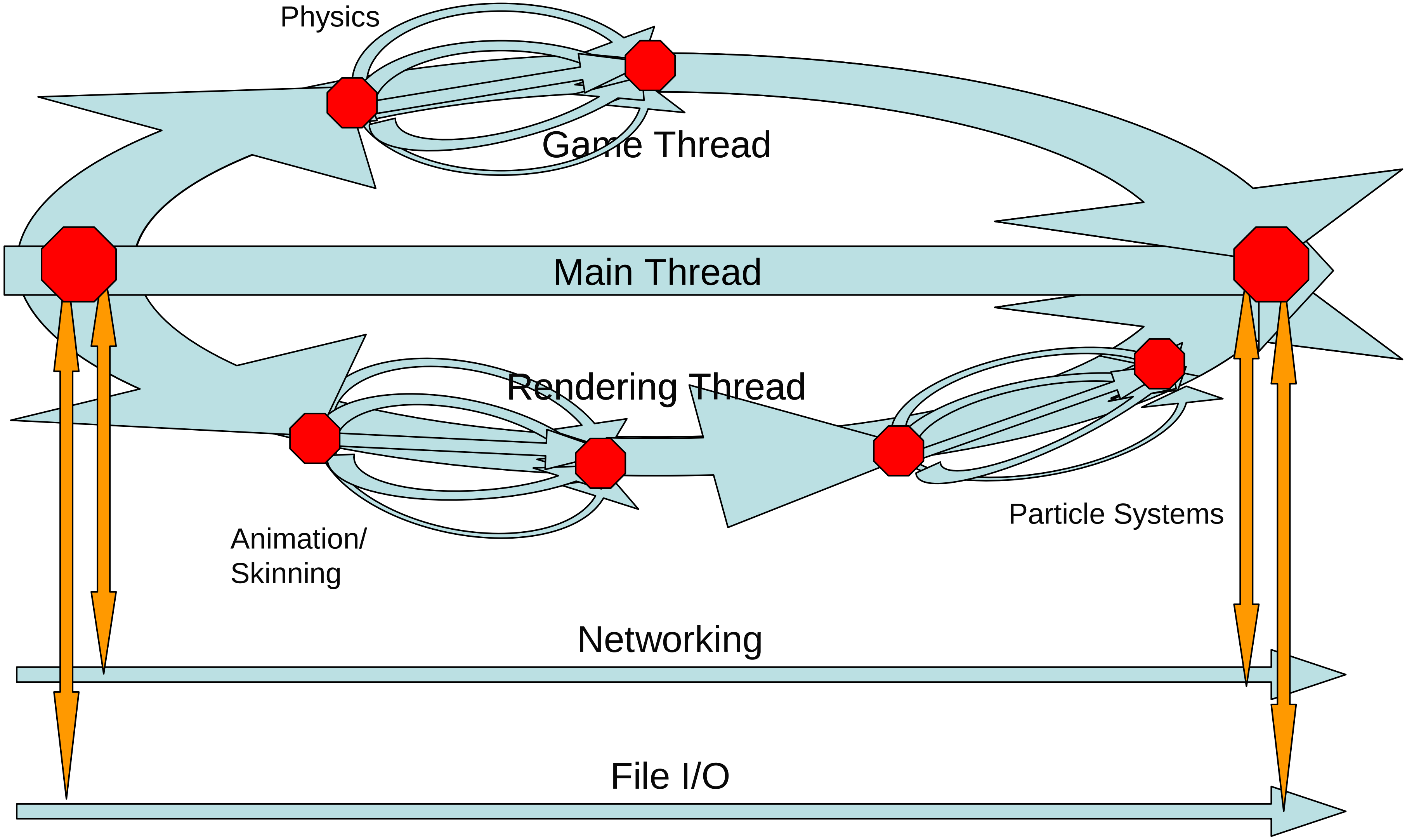
Thread 2

Thread 3

Thread 4

Thread 5

Good Multithreading



Another Paradigm: Cascades

- Advantages:
  - Synchronization points are few and well-defined
- Disadvantages:
  - Increases latency

Basic Thread Management

- CreateThread()
  - ⚠Watch *dwStackSize*
- SuspendThread(), ResumeThread()
  - ⚠Probably a bad idea—can lead to deadlocks
- TerminateThread() – not available
- XSetThreadProcessor()
  - proc / 2 = core
  - proc % 2 = hw thread
- WaitForSingleObject(), CloseHandle()

Thread Local Storage

- TlsAlloc()
  - Allocates up to 64 DWORDs
  - Returns index
- TlsSetValue()/GetValue()/Free()
- __declspec(thread)
  - Not for use with massive arrays
  - Earlier docs discouraged—ignore

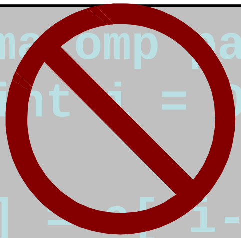
Heap APIs

- In general, system libraries are thread-safe
- Memory management APIs let you turn thread-safety off
  - Pass HEAP_NO_SERIALIZE
  - Use **only** when certain multiple, simultaneous access is not a problem
  - **Never** use on main system heap
- Best use: heaps that are read-only to all but one thread

Open MP

- Set of compiler directives for easy parallelization
- #pragma omp *

```
#pragma omp parallel for
for( int i = 0; i < 1000000; i++ )
{
  a[i] = i;
}
```



```
#pragma omp parallel for
for( int i = 0; i < 1000000; i++ )
{
  a[i] = a[ i-1 ];
}
```

Controlling OpenMP

- Default is to create a thread on each processor
  - OpenMP considers each hardware thread to be a processor
- xomp_set_cpu_order()

```
xomp_cpu_order_t orderNew;
orderNew.order[ 0 ] = 4;
orderNew.order[ 1 ] = 2;
xomp_cpu_order_t orderOld = xomp_set_cpu_order( orderNew );
#pragma omp parallel for num_threads( 2 )
// loop goes here
// reset CPU order to orderOld when done
```

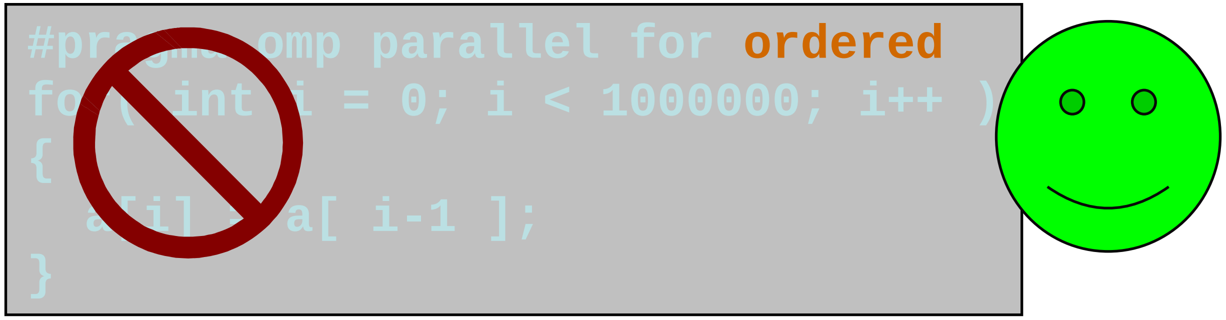
More fun with OpenMP

- #pragma omp parallel…
  - – sections
    - Follow with #pragma omp section
  - – if(expr)
  - – shared(varName)
  - – private(varName)
  - – reduction(op : varName)

```
int nSum = 0;
#pragma omp parallel reduction ( + : nSum ) for
for( int i = 0; i <= 10; i++ )
  nSum += i;
printf( "%d", nSum );
```

Still more fun with OpenMP

- #pragma omp…
  - critical
  - barrier
  - flush
  - master
  - ordered



```
#pragma omp parallel for ordered
for( int i = 0; i < 1000000; i++ )
{
  a[i] = a[ i-1 ];
}
```

- For more information, read the Visual Studio® 2005 docs

Fibers

- Cooperative software pseudothreads
- Do not preempt
- ConvertThreadToFiber()
- CreateFiber()
- SwitchToFiber()
- Context switches are 7-9× faster than software threads

Overview: Synchronization

- Necessary to control access to shared resources
  - Primarily memory
- A **lock** is a construct designed to stall a thread until needed resources are available.
- **Lockless** programming uses non-locking constructs to achieve synchronized access
  - Lockless programming is subtle and can lead to very hard-to-find bugs
  - Unpredictable memory latency can lead to strange results ⚠

Synch: The Goldilocks Problem

- Not enough synch
  - Can lead to fatal, hard-to-find bugs
- Too much synch
  - Wastes time acquiring unneeded locks
  - Wastes even more time if you have to wait unnecessarily!
- Just right, but…
  - Better to eliminate contention, if possible

Locks: Critical Sections

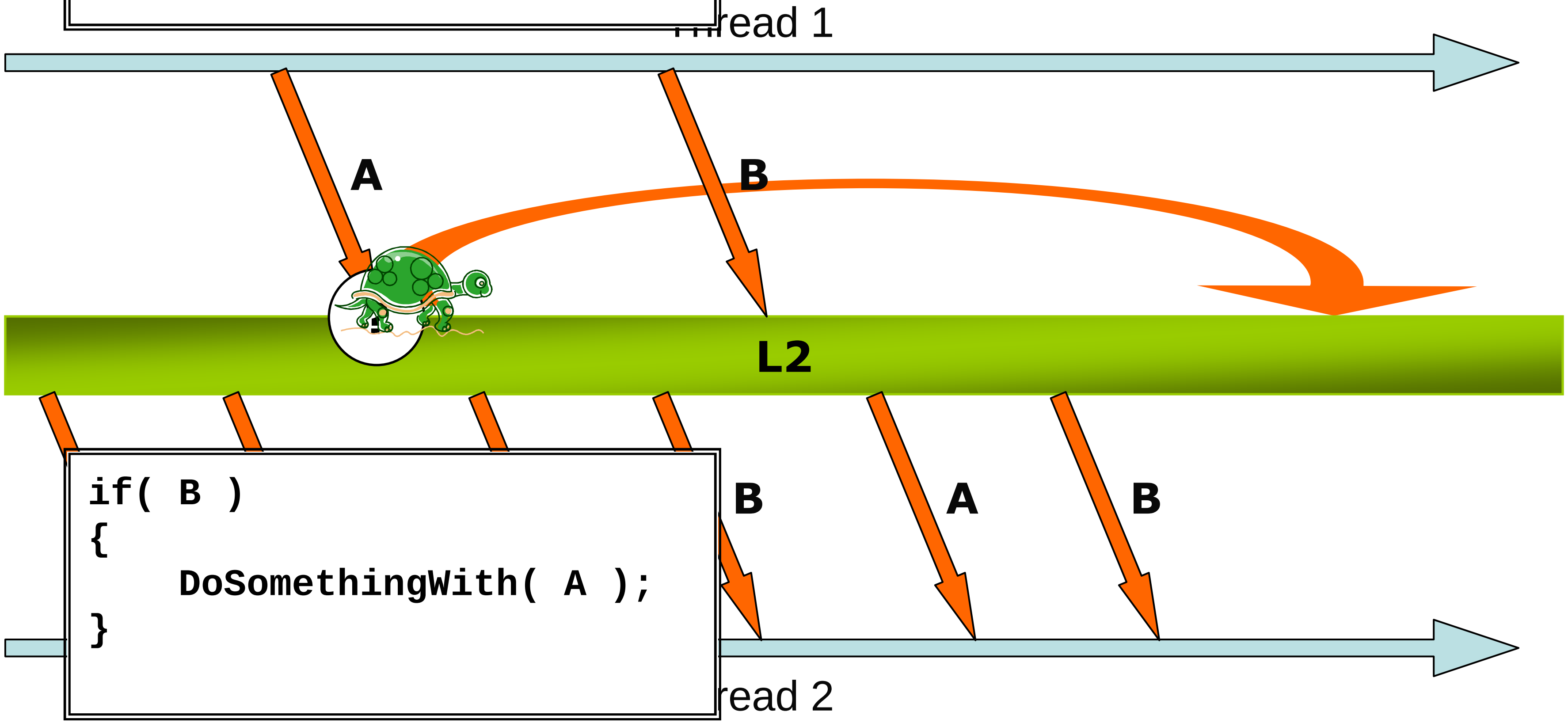
- Prevent contention by ensuring only a single thread can use a resource
  - InitializeCriticalSection()
  - EnterCriticalSection() / TryEnterCriticalSection()
- Cheap, but not free
  - ≈700 cycles when not blocking
  - Rolling your own doesn’t help

Locks: Objects

- Events
  - Single trigger
  - Great for letting threads sleep while waiting for another thread
- Semaphores
  - Have a count that can be incremented
  - Count decrements when waiting thread is released
  - Great for job queues
- Mutexes
  - Allow single thread access to resources

The Joys of Lockless Synch

```
A = new SomeObject();
B = true;
```



```
if( B )
{
    DoSomethingWith( A );
}
```

Memory Access Reordering

- Order of completing of memory accesses is not guaranteed
  - Compiler may reorder instructions
  - CPU may reorder instructions

Memory Access Reordering

- Order of completing of memory accesses is not guaranteed
  - Compiler may reorder instructions
  - CPU may reorder instructions
  - CPU may reorder reads and writes
- Needed: *memory barrier*
  - Ensures that prior memory accesses complete before future memory accesses

Enter lwsync

- lwsync CPU instruction
- __lwsync() compiler intrinsic
- Creates a memory barrier
  - All memory accesses before lwsync must complete before memory accesses after lwsync
  - Works across threads

lwsync semantics

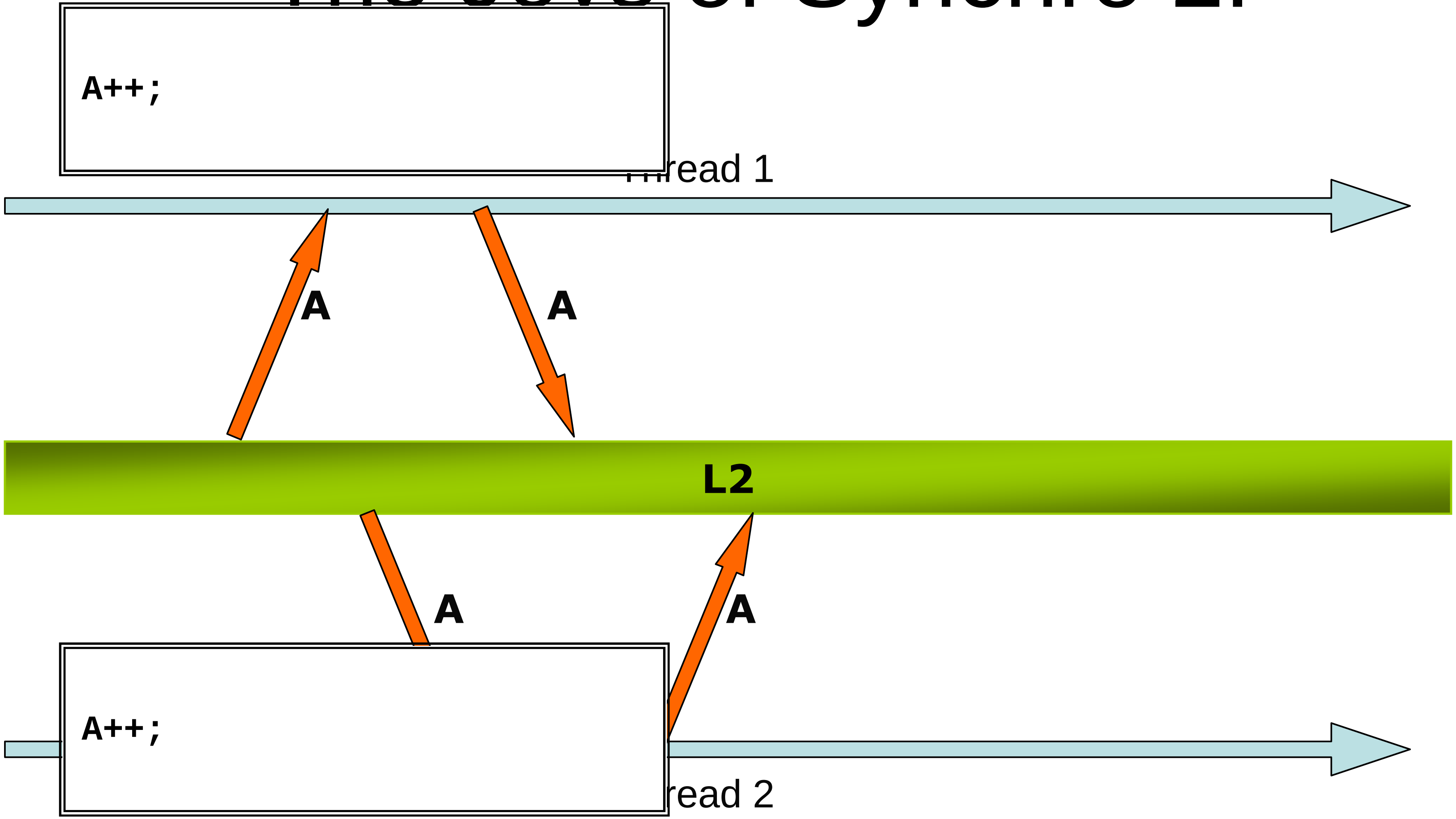
```
A = new SomeObject();
__lwsync() // release A
B = true;
```

- after an operation, results ture operations
  - Put __lwsync() after operation
- *Release*: Ensure that before doing an operation, all previous operations have completed
  - Put __lwsync() before operation
- *Fence*: A combination of the above

```
if( B )
{
    __lwsync() // acquire A
    DoSomethingWith( A );
}
```

- jects are implemented

The Joys of Synchro 2:



Interlocked IO

- Math
  - InterlockedIncrement/Decrement
  - InterlockedAnd/Or/Xor
- Conditionals
  - InterlockedExchange/CompareExchange
- Stacks
  - InterlockedPush/Pop/FlushSList
- Cheap
- Do **NOT** create a memory barrier; you must use __lwsync() in an appropriate location

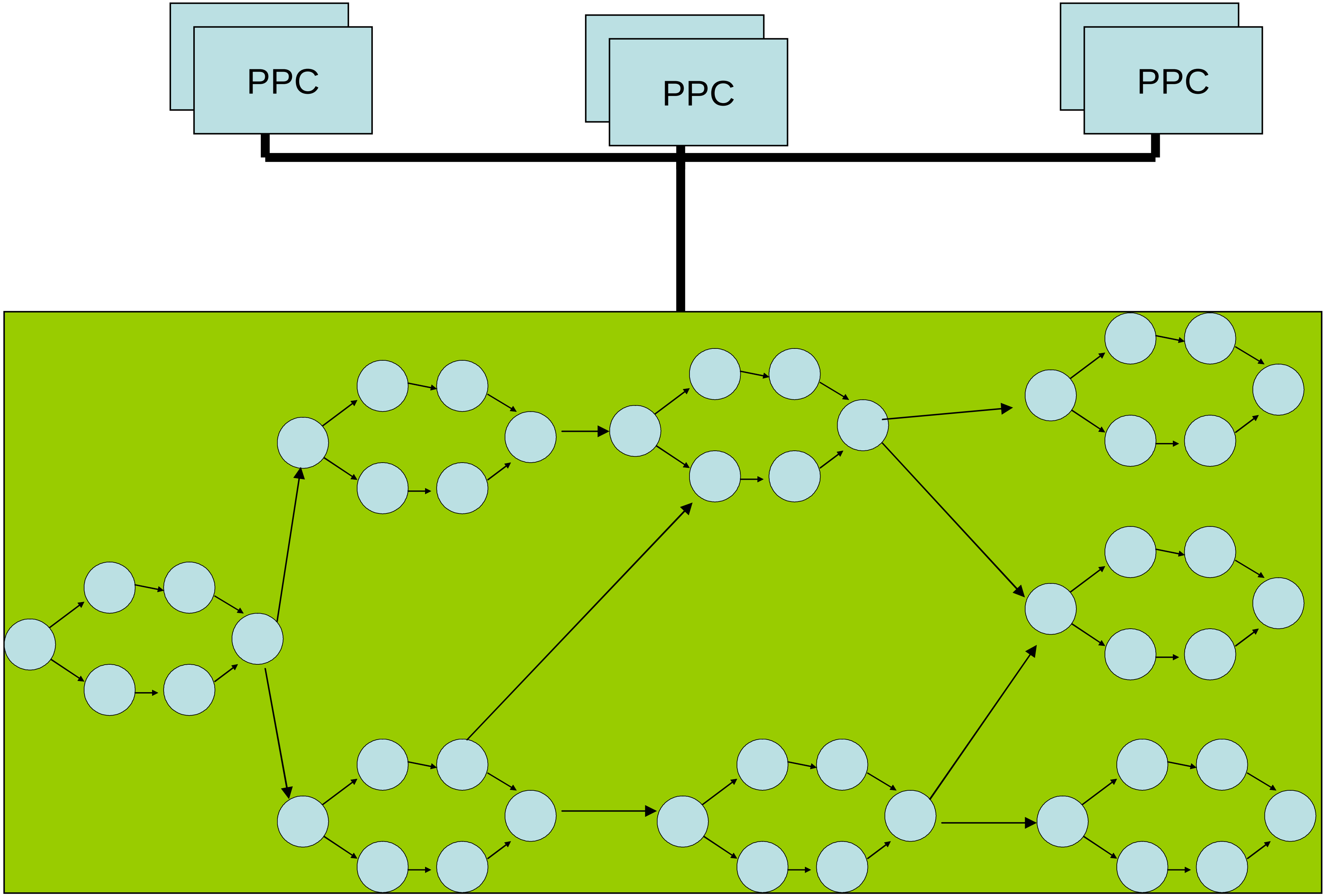
Conclusions

- Multithreading is important
- Design a multithreaded architecture that works for you
- Use locks judiciously
- Use lockless programming with extreme caution

CSP

- Program your application as a CSP network
  - Make sure you have enough processes
    - Enough >> 6
- With shared cache we can let all 6 HW threads use the same scheduler
  - But we can also let L1 dictate a 2x3 scheduler

CSP



PPC

PPC

PPC

Advantages of CSP

- No consideration of the underlying architecture when determining parallelism
  - Porting to other architectures is easy
- Dynamic load balancing

Problems with CSP

- No tools exists
- CPS kernel must be implemented with knowledge of the architecture
  - This should be really easy on this architecture though