# COMP60001/COMP70086
# Advanced Computer Architecture
# Chapter 1.4

## Caches: a quick review of introductory *memory system* architecture

Objective: bring everyone up to speed, and also establish some key ideas that will come up later in the course in more complicated contexts

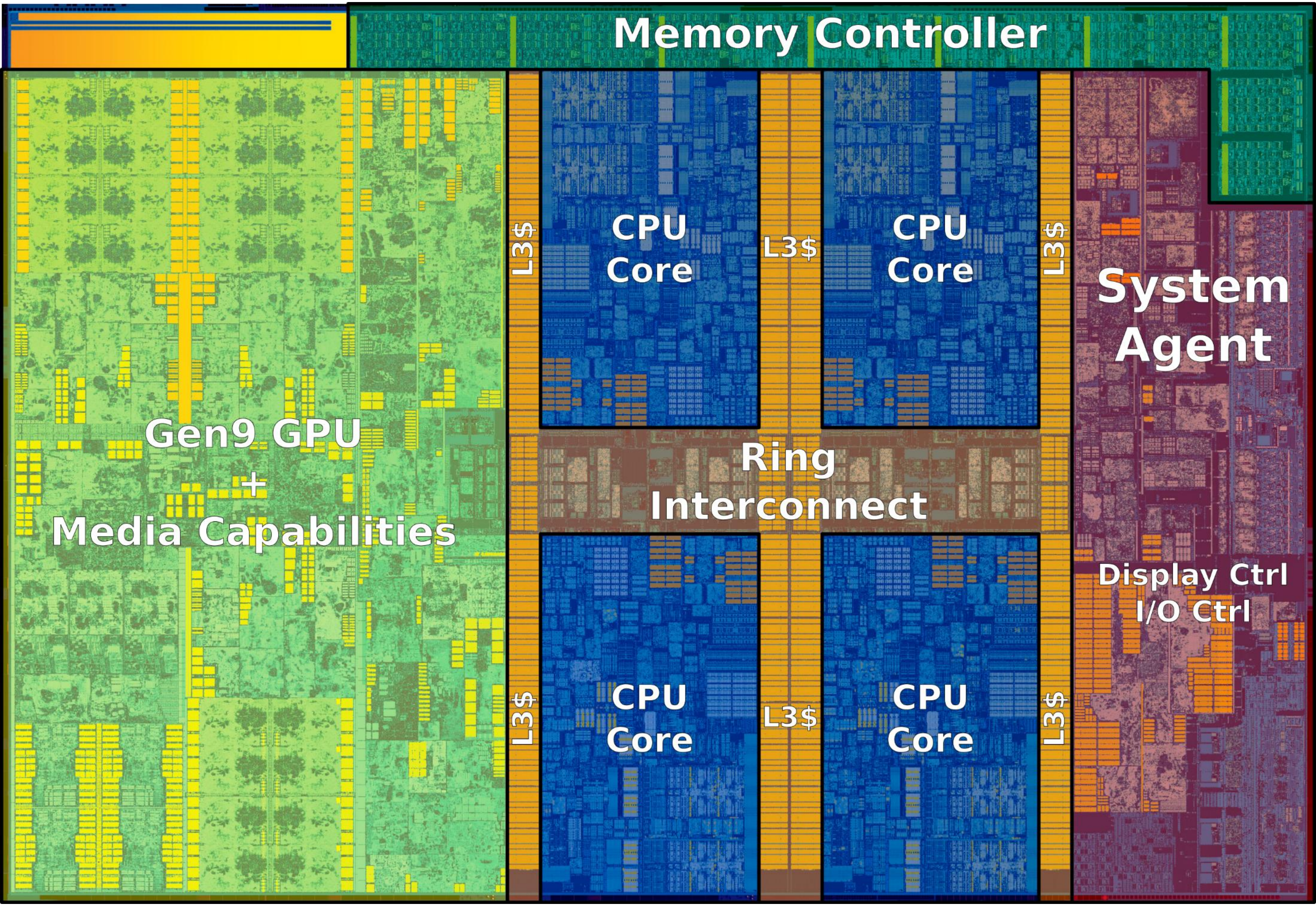
October 2025

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (6th ed), and on the lecture slides of David Patterson's Berkeley course (CS252)*

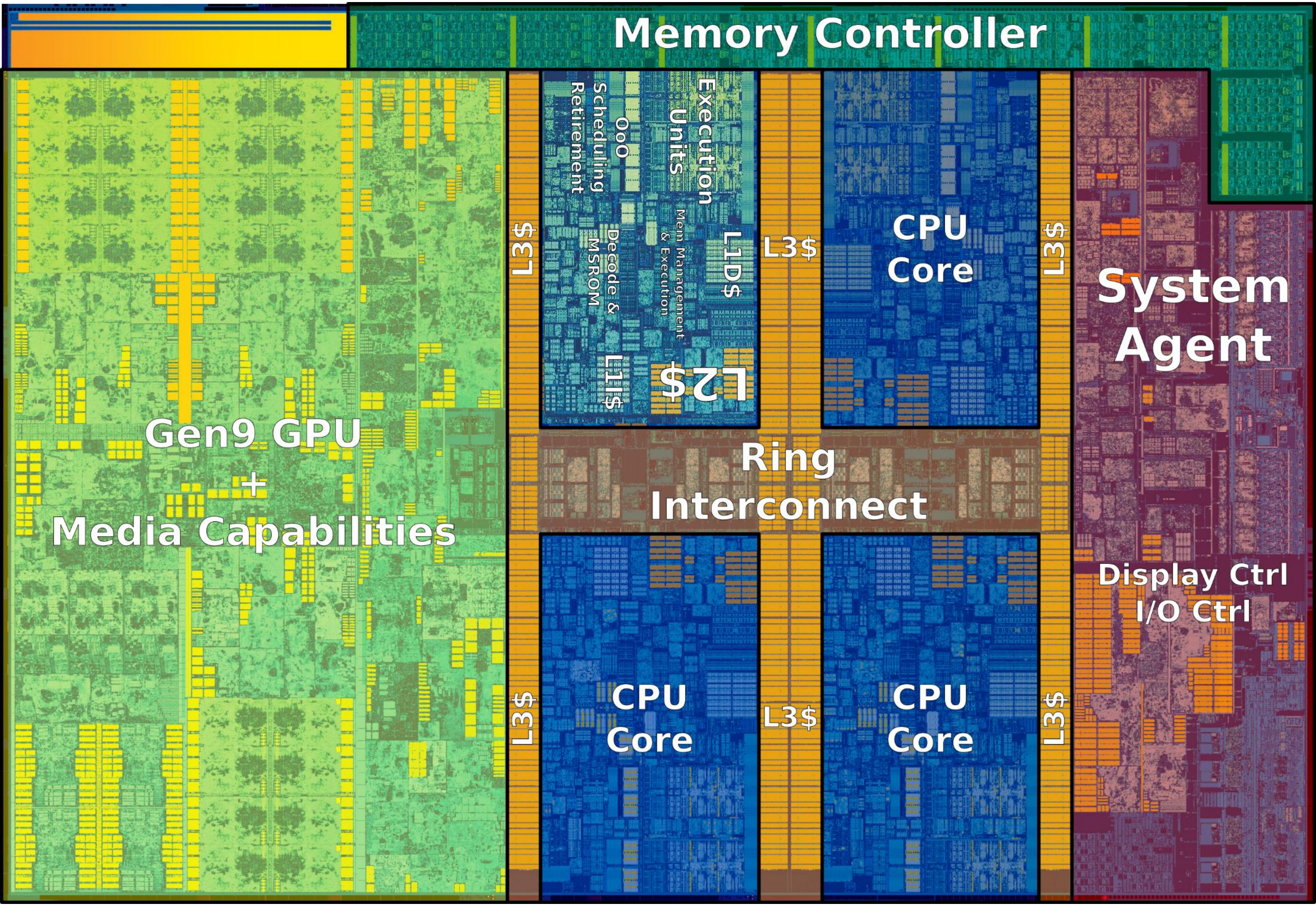
Course materials online on
https://scientia.doc.ic.ac.uk/2526/modules/60001/materials and
https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/

# Intel Skylake quad-core die photo



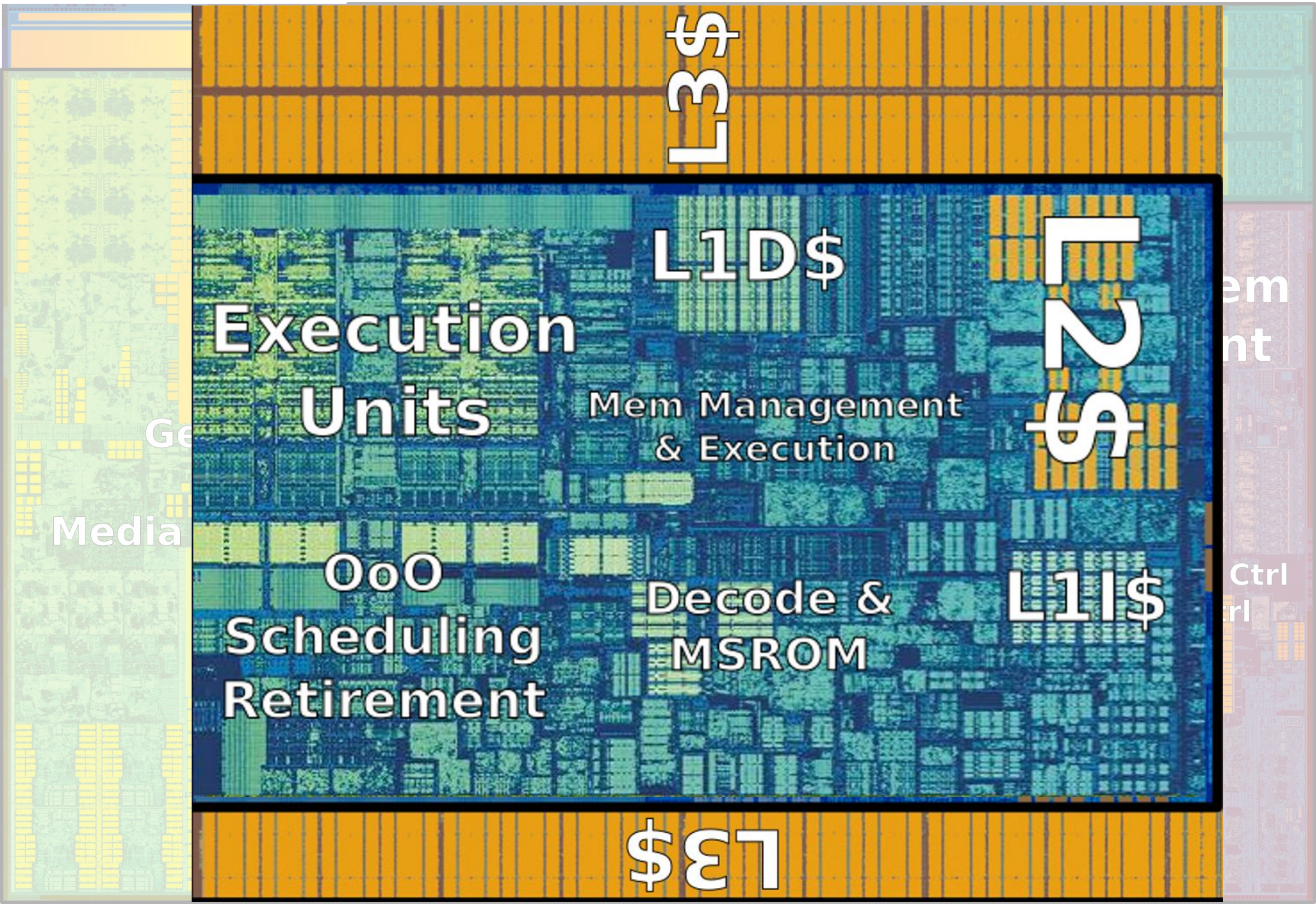
https://en.wikichip.org/wiki/File:skylake_(quad-core)_(annotated).png

Intel Skylake quad-core die photo



https://en.wikichip.org/wiki/File:skylake_(quad-core)_(annotated).png

# Intel Skylake quad-core die photo



https://en.wikichip.org/wiki/File:skylake_(quad-core)_(annotated).png

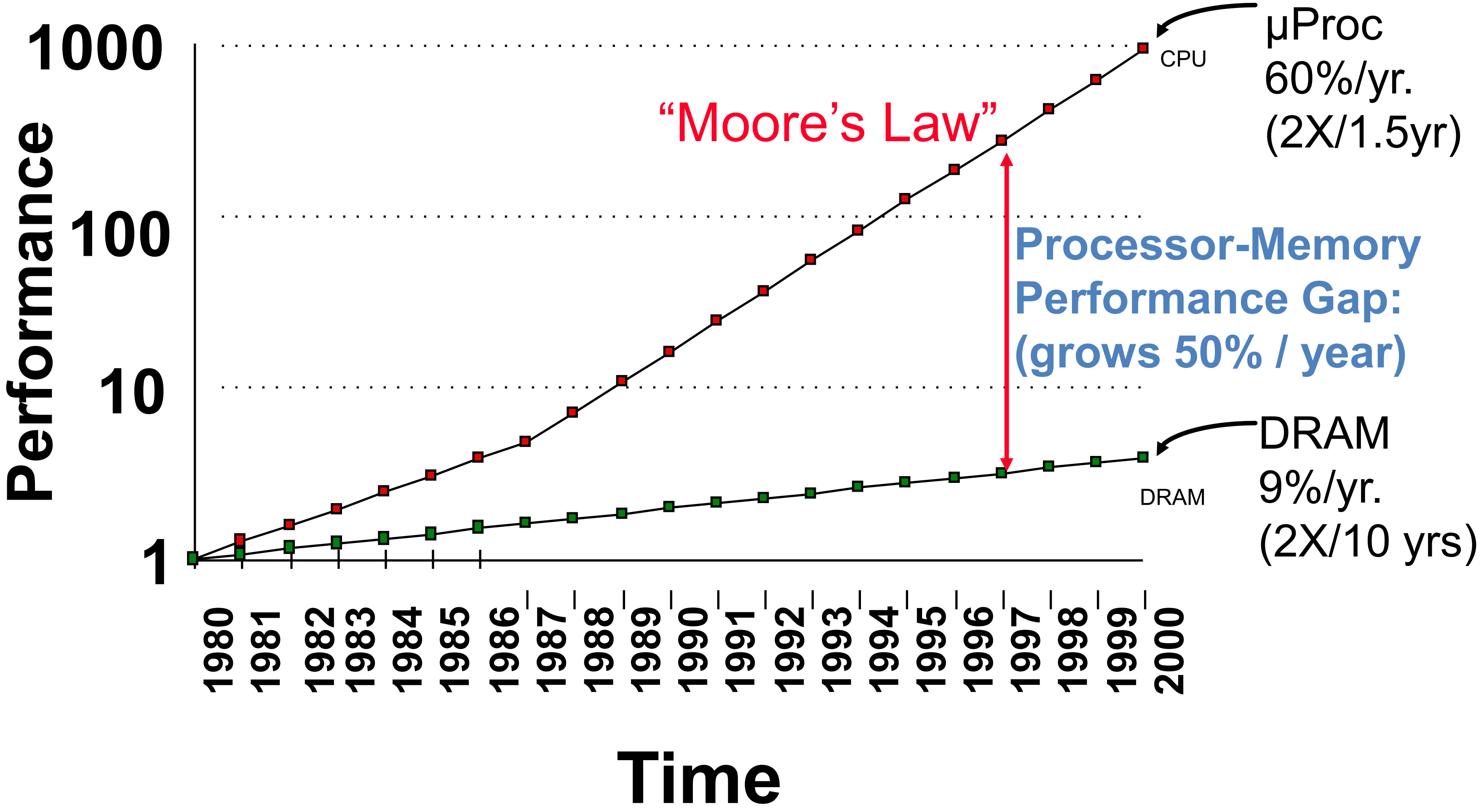
# We finished the last lecture by asking how fast a pipelined processor can go?

- A simple 5-stage pipeline can run at 5-9GHz
- Limited by critical path through slowest pipeline stage logic
- Tradeoff: do more per cycle?  Or increase clock rate?
  - Or do more per cycle, in parallel…
- At 3GHz, clock period is 330 picoseconds.
  - The time light takes to go about four inches
  - About 10 gate delays
    - for example, the Cell BE is designed for 11 FO4 (“fan-out=4”) gates per cycle: www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf
    - Pipeline latches etc account for 3-5 FO4 delays leaving only 5-8 for actual **work**

- **How can we build a RAM that can implement our MEM stage in 5-8 FO4 delays?**

Life used to be so easy

# Processor-DRAM Memory Gap (latency)



In 1980 a large RAM's access time was close to the CPU cycle time. 1980s machines had little or no need for cache. Life is no longer quite so simple.

# Levels of the Memory Hierarchy

*Capacity*
*Access Time*
*Cost*

*CPU Registers*
100s Bytes
<1ns

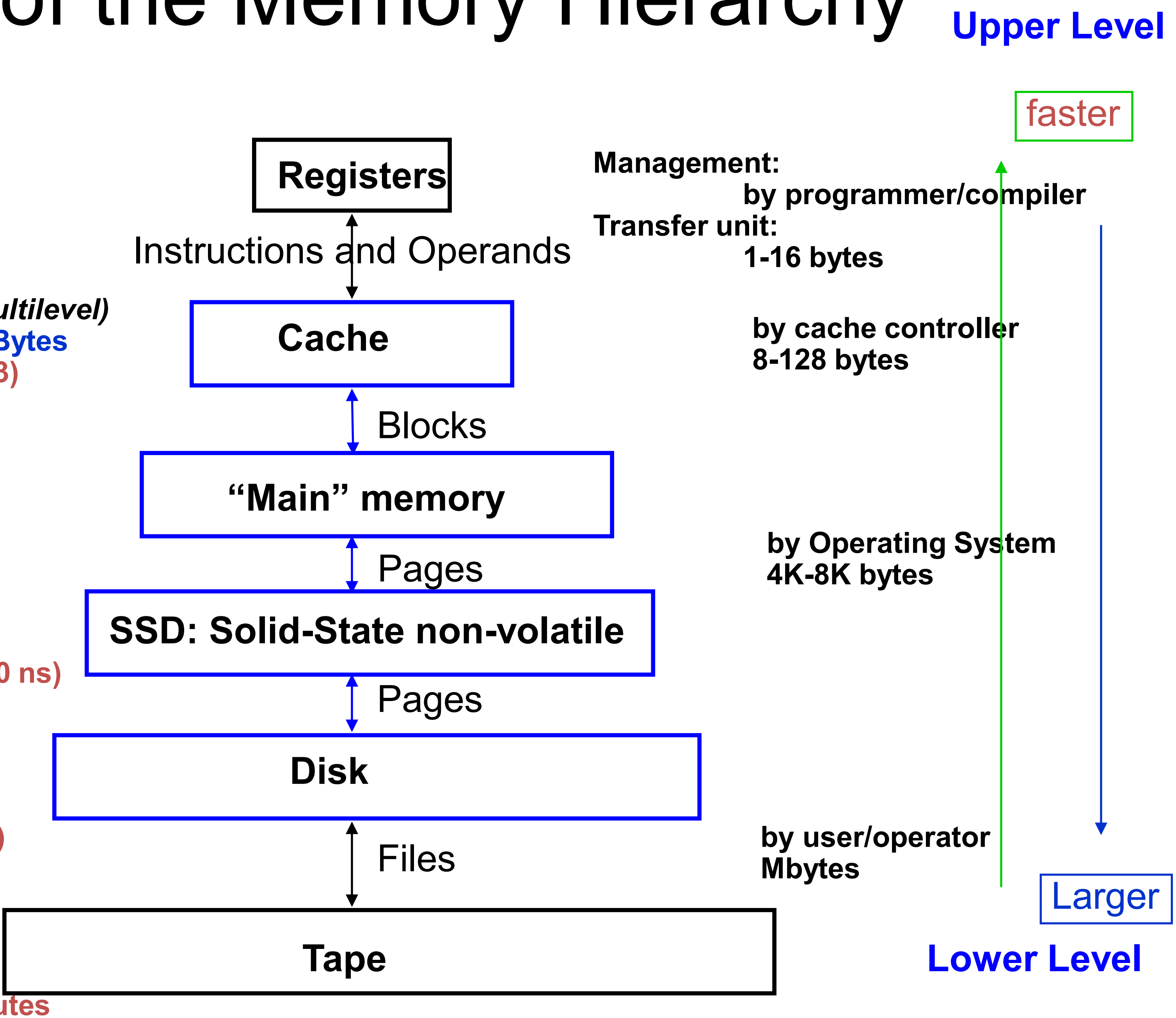
*Cache (perhaps multilevel)*
10s-1000s K Bytes
1-10 ns (L1-L3)
~$1 / MByte

*Main Memory*
G Bytes
100ns- 200ns
$1 / GByte

*SSD*
Terabytes,
10-500us
(10,000-500,000 ns)
$50-70 / TB)

*Disk*
Terabytes,
10 ms
(10,000,000 ns)
$10-20 / TB)

*Tape*
infinite
Seconds-minutes
$5-10 / TB



Upper Level

faster

Registers

Instructions and Operands

Management:
by programmer/compiler
Transfer unit:
1-16 bytes

Cache

by cache controller
8-128 bytes

Blocks

"Main" memory

Pages

by Operating System
4K-8K bytes

SSD: Solid-State non-volatile

Pages

Disk

Files

by user/operator
Mbytes

Larger

Tape

Lower Level

~ Exponential increase in access latency, block size, capacity

- The Principle of Locality:
  - Programs access a relatively small portion of the address space at any instant of time.

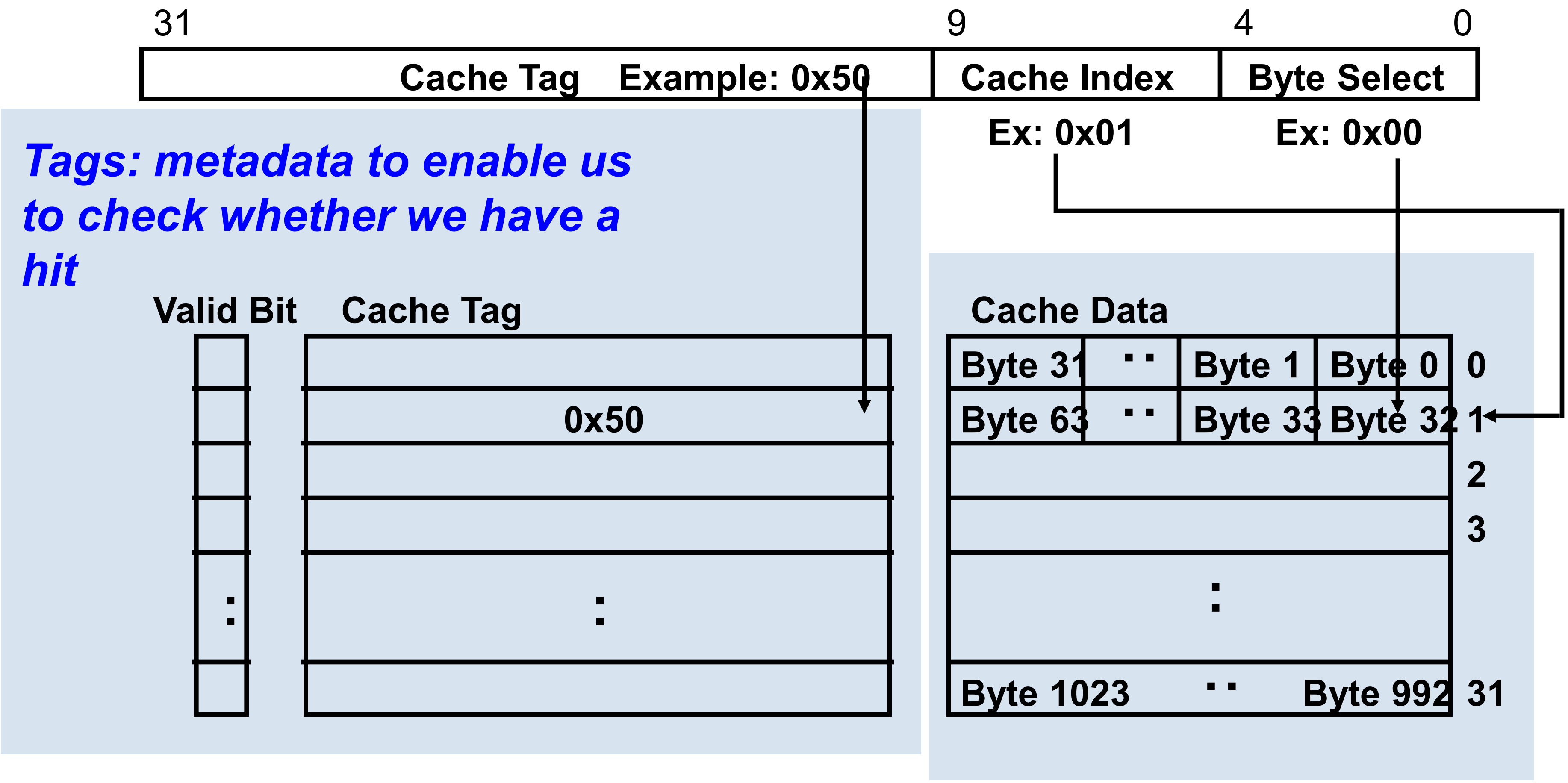
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)

  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon
    (e.g., straightline code, array access)

- Most modern architectures are heavily reliant (totally reliant?) on locality for speed

# 1 KB "Direct Mapped" Cache, 32B blocks

- For a $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)



31 9 4 0

Cache Tag Example: 0x50 | Cache Index | Byte Select

Ex: 0x01 Ex: 0x00

*Tags: metadata to enable us to check whether we have a hit*

Valid Bit Cache Tag

0x50

:

Cache Data

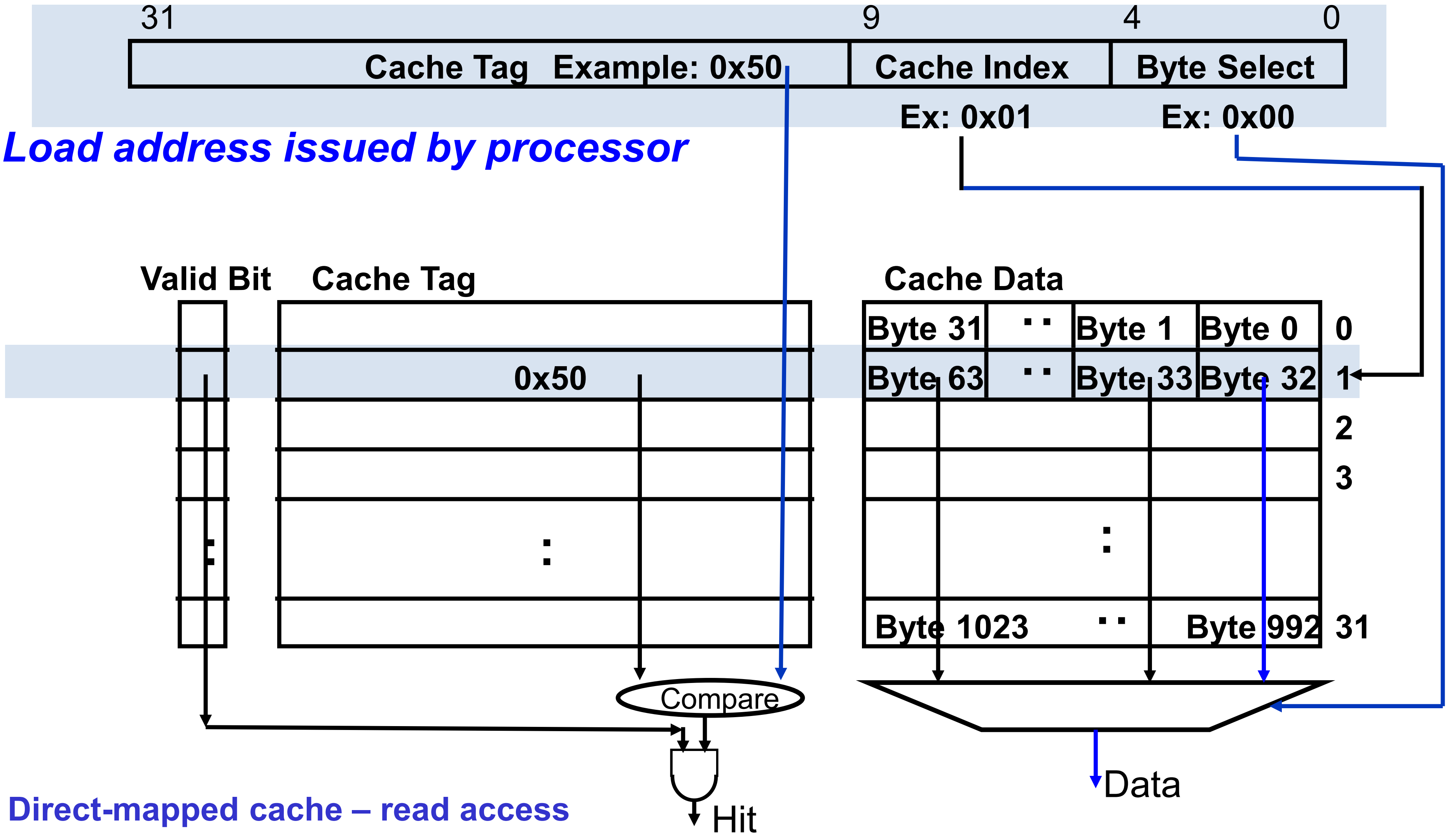
| | | | | |
|---|---|---|---|---|
| Byte 31 | : : | Byte 1 | Byte 0 | 0 |
| Byte 63 | : : | Byte 33 | Byte 32 | 1 |
| | | | | 2 |
| | | | | 3 |
| | : | | | |
| Byte 1023 | : : | | Byte 992 | 31 |

*Data: the cached data itself, arranged in cache lines/blocks*

**Direct-mapped cache - storage**

# 1 KB “Direct Mapped” Cache, 32B blocks

- For a $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)



31 9 4 0

Cache Tag Example: 0x50

Cache Index

Byte Select

Ex: 0x01

Ex: 0x00

*Load address issued by processor*

Valid Bit

Cache Tag

Cache Data

0x50

Byte 31 : : Byte 1 Byte 0 0

Byte 63 : : Byte 33 Byte 32 1

2

3

:

Byte 1023 : : Byte 992 31

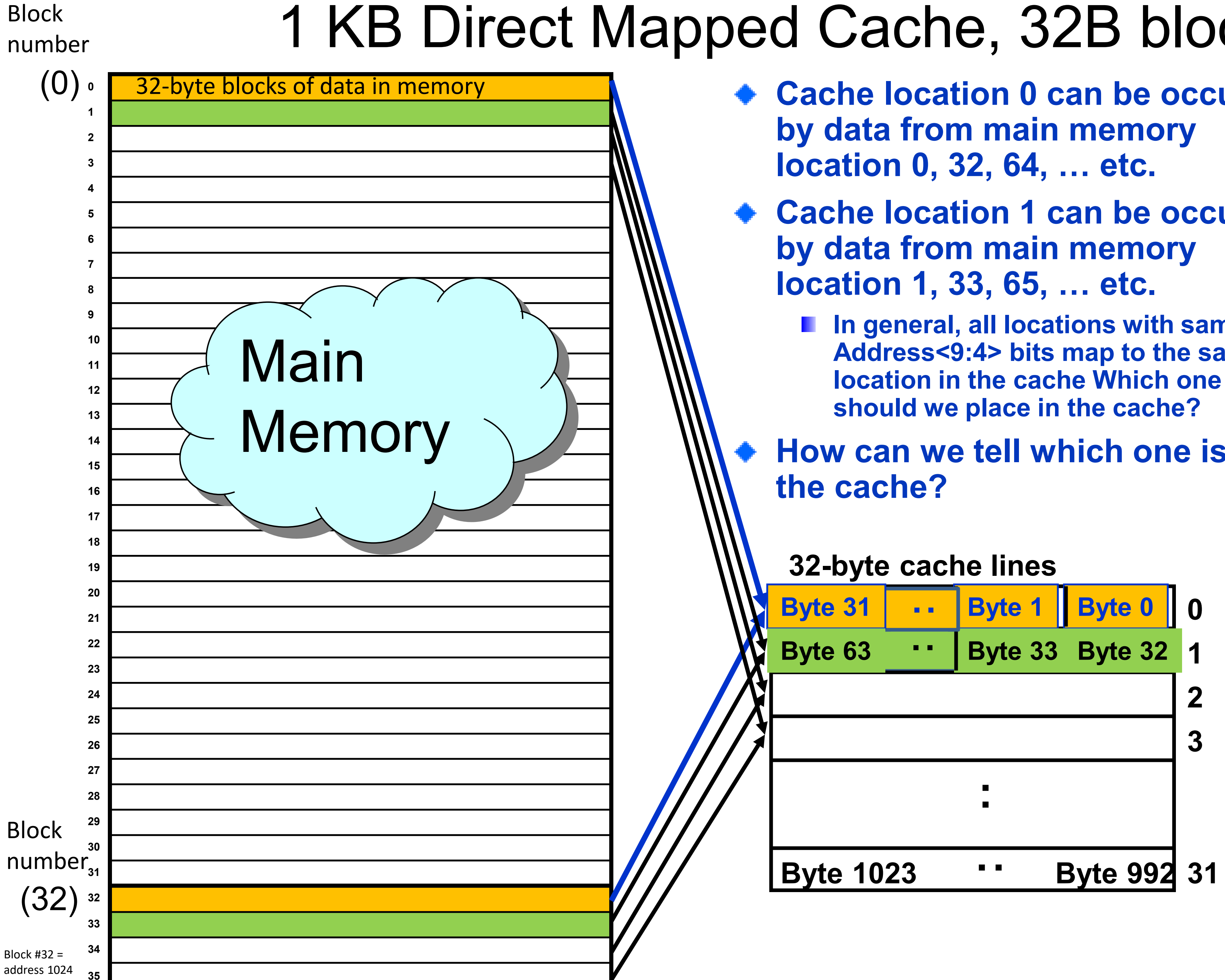
Compare

Hit

Data

**Direct-mapped cache – read access**

# 1 KB Direct Mapped Cache, 32B blocks



Block #32 = address 1024

- Cache location 0 can be occupied by data from main memory location 0, 32, 64, … etc.
- Cache location 1 can be occupied by data from main memory location 1, 33, 65, … etc.
  - In general, all locations with same Address<9:4> bits map to the same location in the cache Which one should we place in the cache?
- How can we tell which one is in the cache?

# Associativity conflicts in a direct-mapped cache

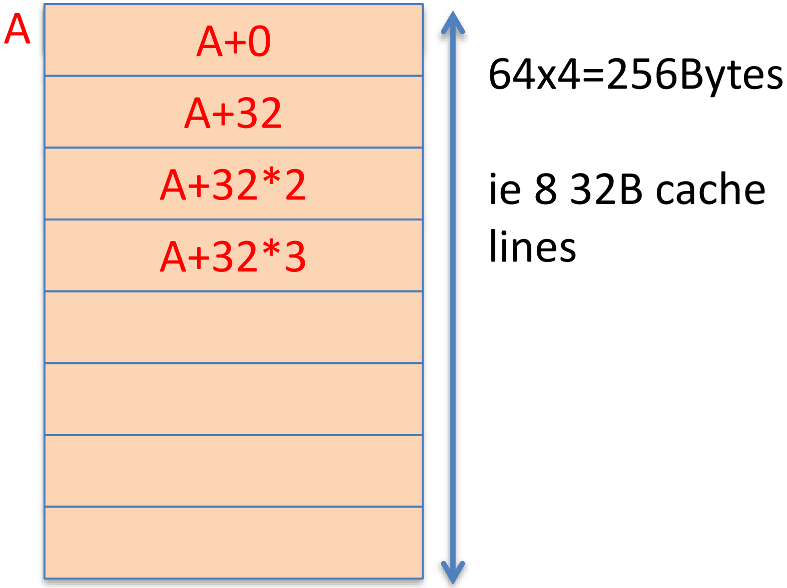
- **Consider a loop that repeatedly reads part of two different arrays:**

```
int A[256];
int B[256];
int r = 0;
for (int i=0; i<10; ++i) {
  for (int j=0; j<64; ++j) {
    r += A[j] + B[j];
  }
}
```

Repeatedly re-reads 64 values from both A and B

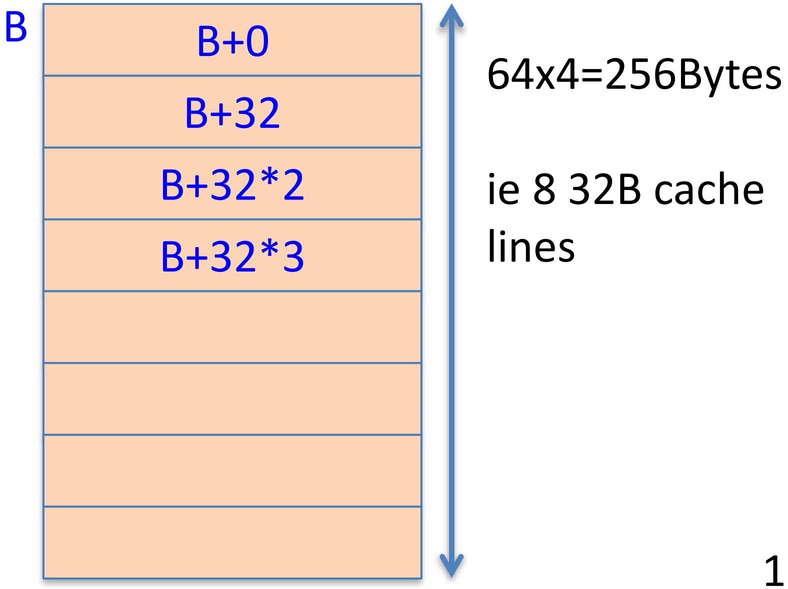
**For the accesses to A and B to be mostly cache hits, we need a cache big enough to hold 2x64 ints, ie 512B**

**Consider the 1KB direct-mapped cache on the previous slide -** what might go wrong?



64x4=256Bytes

ie 8 32B cache lines



64x4=256Bytes

ie 8 32B cache lines

# Associativity conflicts in a direct-mapped cache

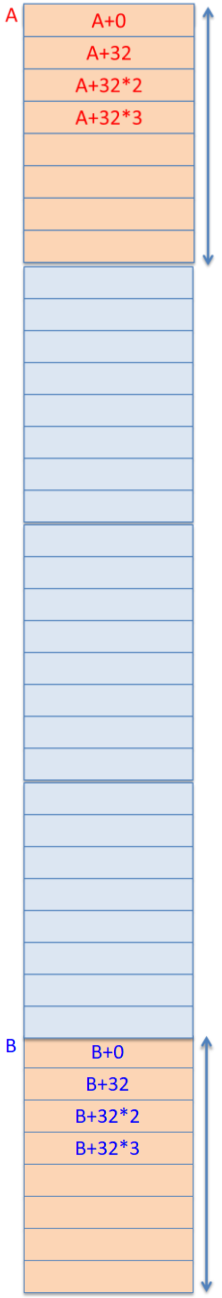
- **Consider a loop that repeatedly reads part of two different arrays:**

```
int A[256];
int B[256];
int r = 0;
for (int i=0; i<10; ++i) {
 for (int j=0; j<64; ++j) {
   r += A[j] + B[j];
 }
}
```

Repeatedly re-reads 64 values from both A and B

**For the accesses to A and B to be mostly cache hits, we need a cache big enough to hold 2x64 ints, ie 512B**

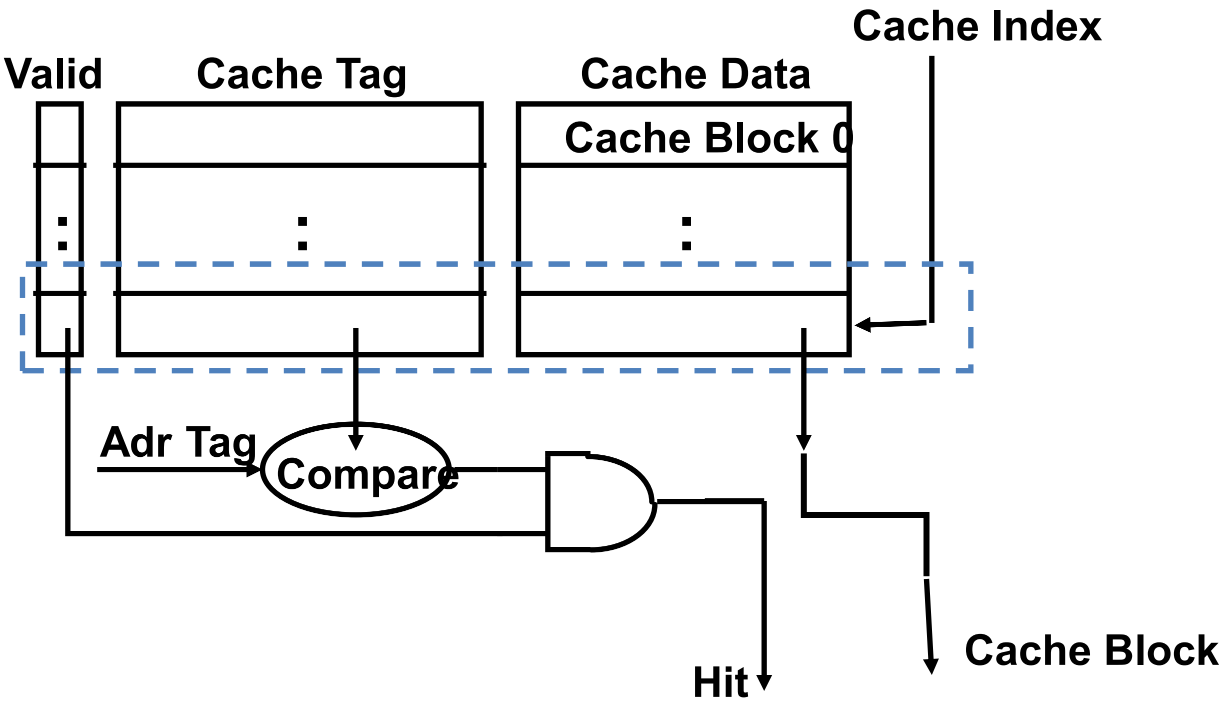
**Consider the 1KB direct-mapped cache on the previous slide - what might go wrong?**



Array B is located exactly 1024 bytes after array A

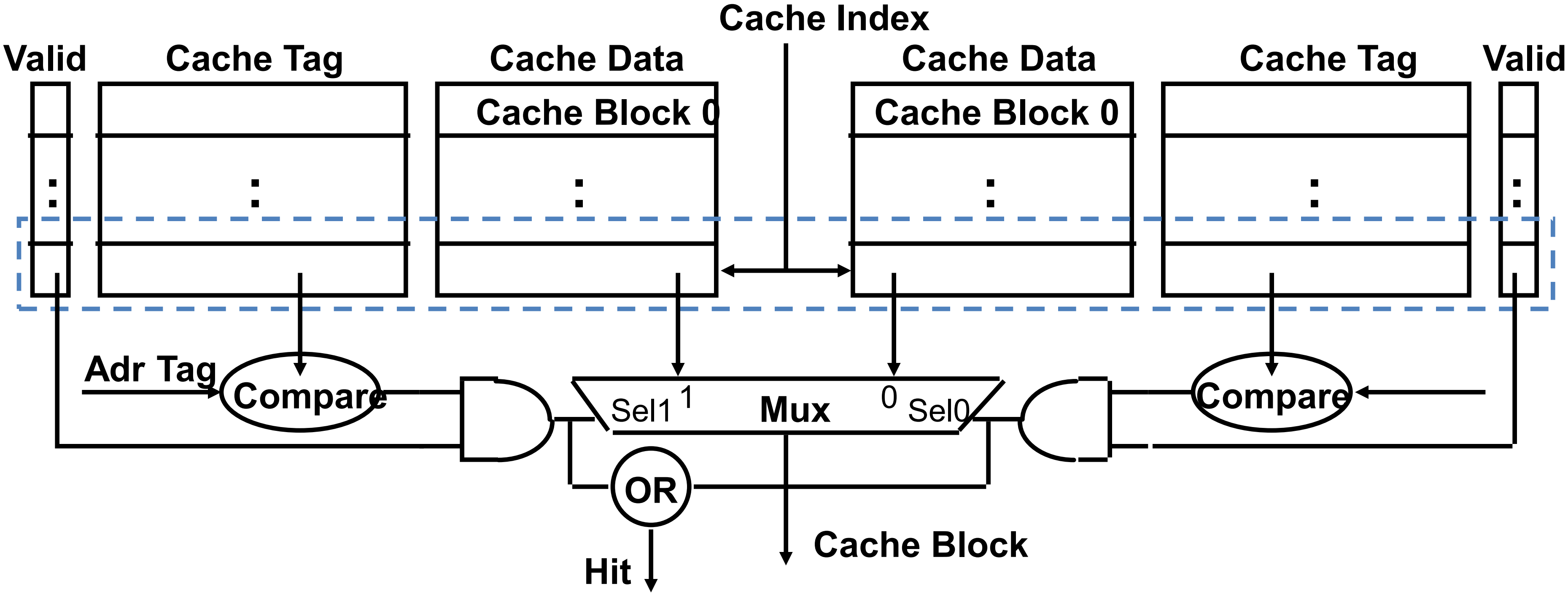
# Direct-mapped Cache - structure

- Capacity: C bytes (eg 1KB)
- Blocksize: B bytes (eg 32)
- Byte select bits: 0..log(B)-1 (eg 0..4)
- Number of blocks: C/B (eg 32)
- Address size: A (eg 32 bits)
- Cache index size: I=log(C/B) (eg log(32)=5)
- Tag size: A-I-log(B) (eg 32-5-5=22)



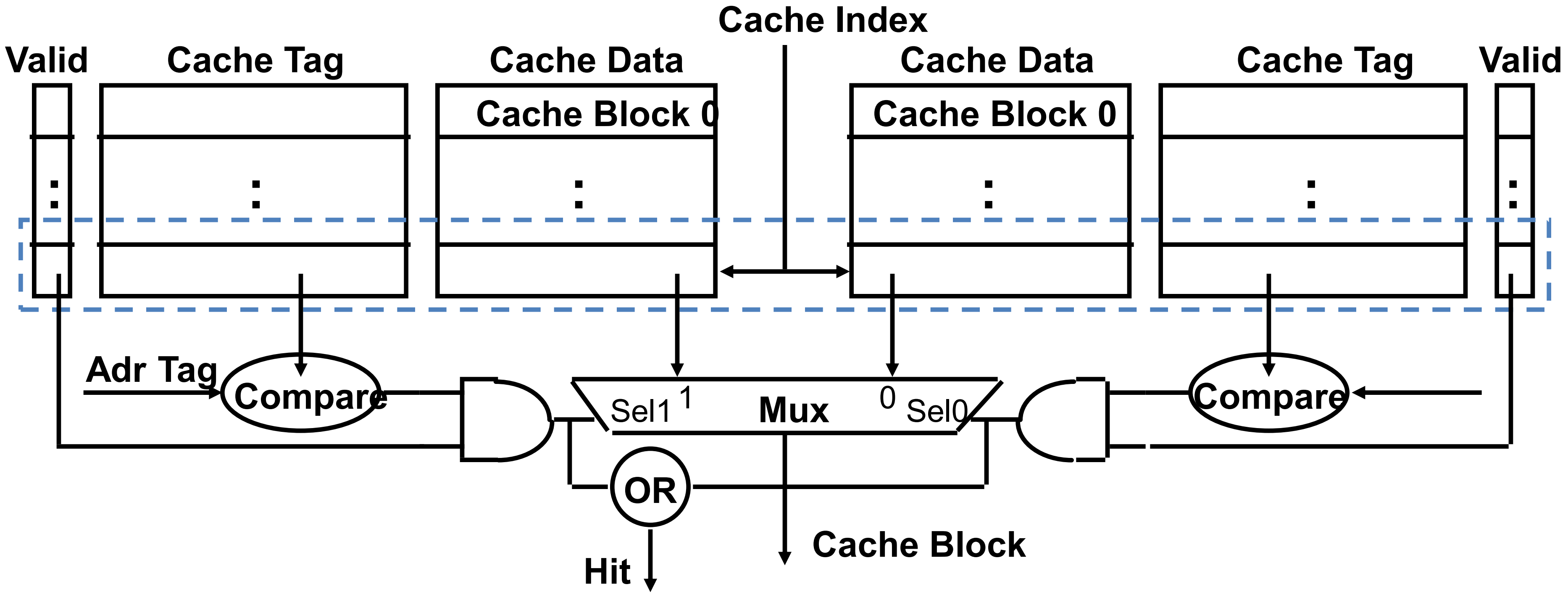
# Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operated in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



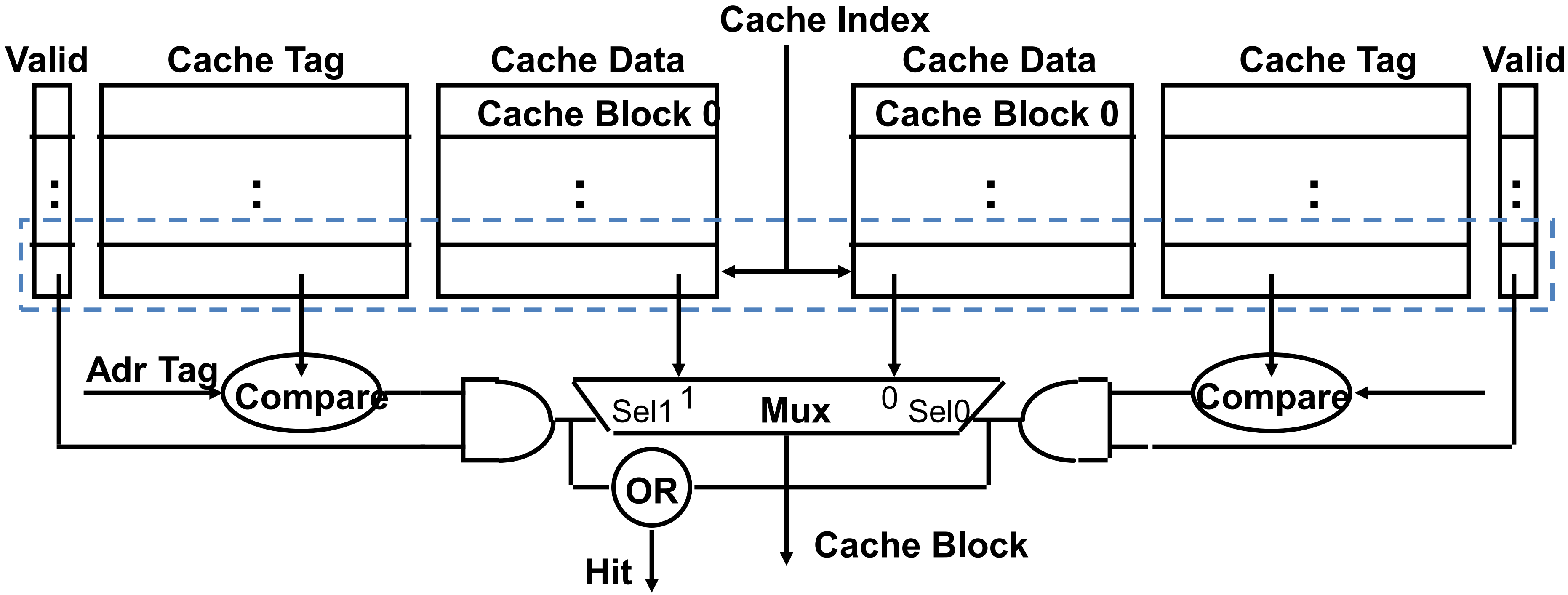
# Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue.  Recover later if miss.



# Example: Intel Pentium 4 Level-1 cache (pre-Prescott)

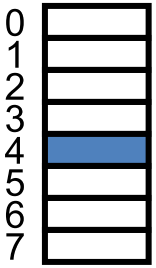
- **Capacity: 8K bytes** (total amount of data cache can store)
- **Block: 64 bytes** (so there are 8K/64=128 blocks in the cache)
- **Ways: 4** (addresses with same index bits can be placed in one of 4 ways)
- **Sets: 32** (=128/4, that is each RAM array holds 32 blocks)
- **Index: 5 bits** (since $2^5$=32 and we need index to select one of the 32 ways)
- **Tag: 21 bits** (=32 minus 5 for index, minus 6 to address byte within block)
- **Access time: 2 cycles,** (.6ns at 3GHz; pipelined, dual-ported [load+store])



# 4 Questions for Memory Hierarchy

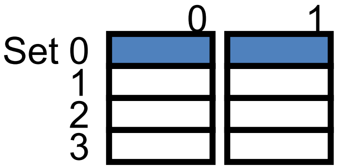
- Q1: Where can a block be placed in the upper level?
  - *Block placement*
- Q2: How is a block found if it is in the upper level?
  - *Block identification*
- Q3: Which block should be replaced on a miss?
  - *Block replacement*
- Q4: What happens on a write?
  - *Write strategy*

# Q1: Where can a block be placed in the upper level?



In a direct-mapped cache, block 12 can only be placed in one cache location, determined by its low-order address bits –

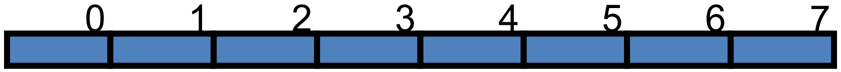
(12 mod 8) = 4



In a two-way set-associative cache, the set is determined by its low-order address bits –

(12 mod 4) = 0

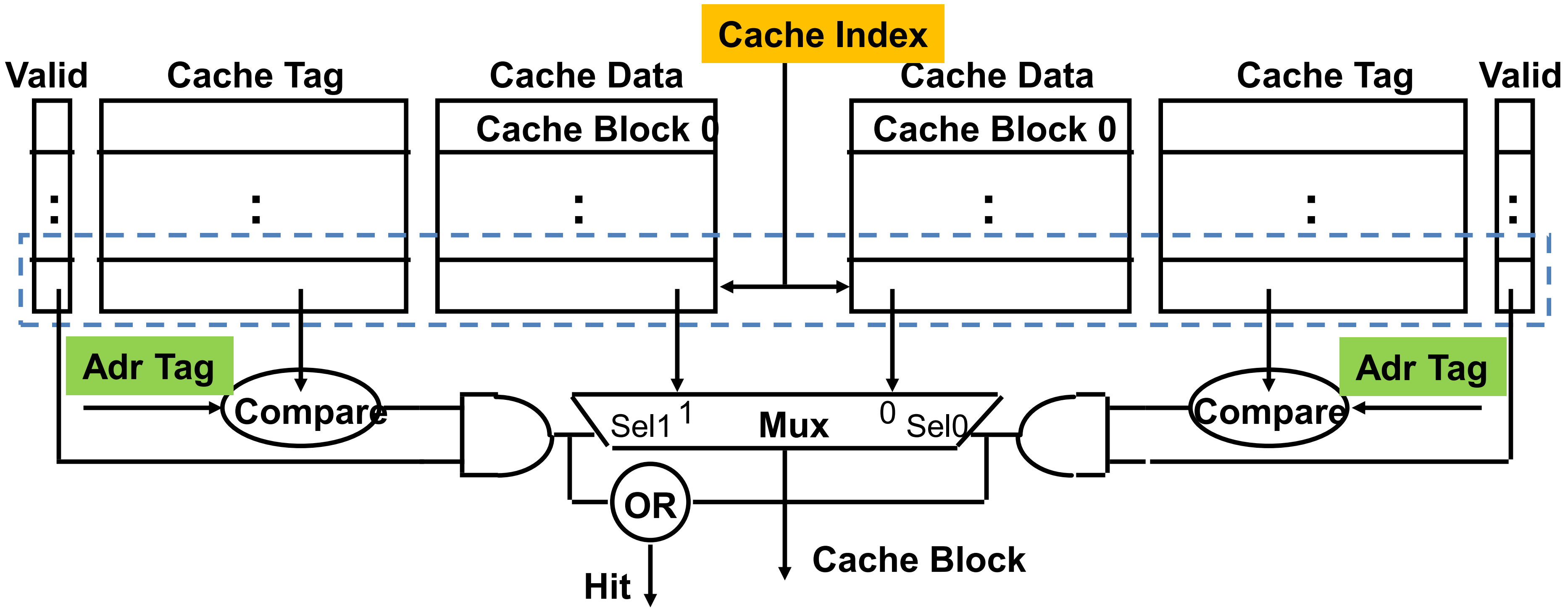
Block 12 can be placed in either of the two cache locations in set 0



In a fully-associative cache, block 12 can be placed in any location in the cache

- **More associativity:**
  - **More comparators – larger, more energy**
  - **Better hit rate (diminishing returns)**
  - **Reduced storage layout sensitivity – more predictable**

# Q2: How is a block found if it is in the upper level?



- Tag on each block
  - No need to check index or block offset

| Block Address | | Block Offset |
|---|---|---|
| Tag | Index | |

- Increasing associativity shrinks index, expands tag

# Q3: Which block should be replaced on a miss?

- With Direct Mapped there is no choice
- With Set Associative or Fully Associative we want to choose
  - Ideal: least-soon re-used
  - LRU (Least Recently Used) is a popular approximation
  - Random is remarkably good in large caches

| Assoc: | 2-way | | 4-way | | 8-way | |
|---|---|---|---|---|---|---|
| Size | LRU | Ran | LRU | Ran | LRU | Ran |
| 16 KB | 5.2% | 5.7% | 4.7% | 5.3% | 4.4% | 5.0% |
| 64 KB | 1.9% | 2.0% | 1.5% | 1.7% | 1.4% | 1.5% |
| 256 KB | 1.15% | 1.17% | 1.13% | 1.13% | 1.12% | 1.12% |

Benchmark studies show that LRU beats random only with small caches

LRU can be pathologically bad....... there are better strategies

# Q4: What happens on a write?

- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory

- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - **is block clean or dirty?**

- Pros and Cons of each?
  - **WT: read misses cannot result in writes**
  - **WB: absorbs repeated writes to same location**

- WT always combined with write buffers so that we don’t wait for lower level memory

# Caches are a *big* topic

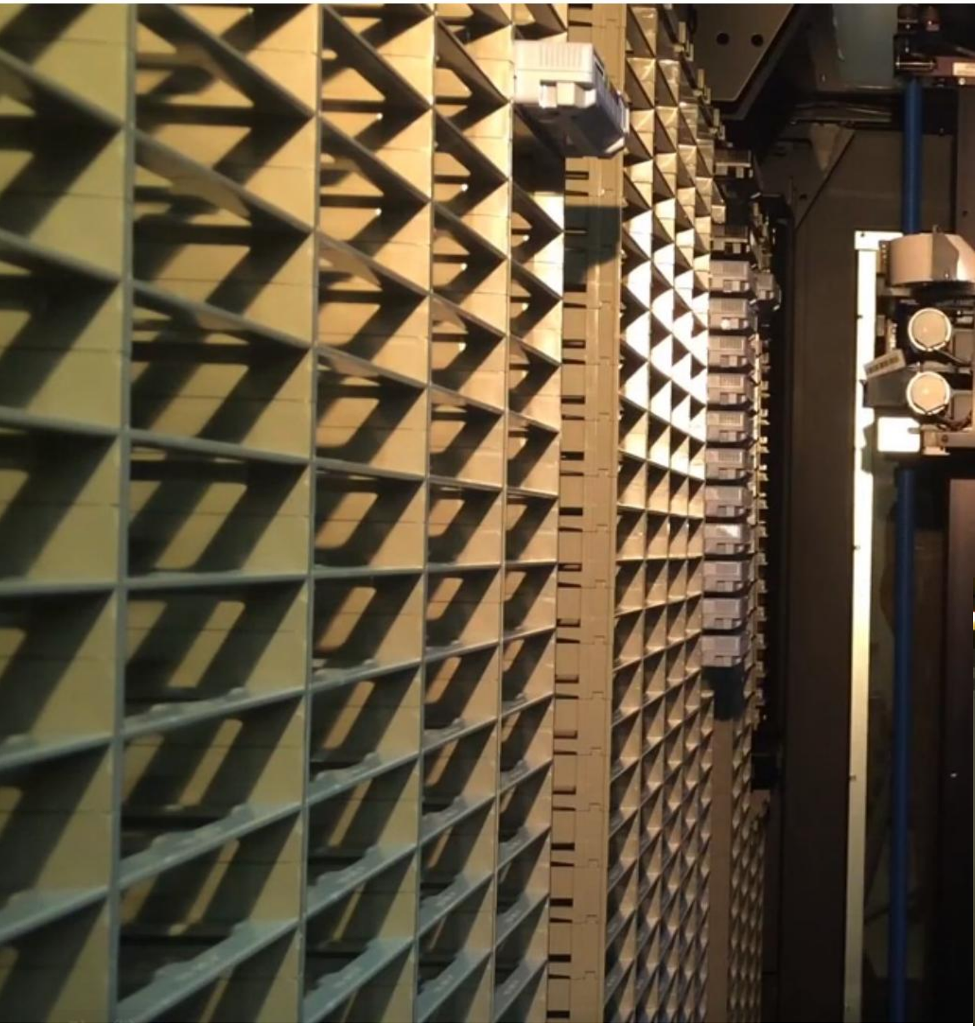
- Cache coherency
  - If your data can be in more than one cache, how do you keep the copies consistent?
- Victim caches
  - Stash recently-evicted blocks in a small fully-associative cache (a “competitive strategy”)
- Prefetching
  - Use a predictor to guess which block to fetch next – *before* the processor requests it
- And much much more........

# What’s at the bottom of the memory  hierarchy?

- StorageTek STK 9310 (“Powderhorn”)
  - 2,000, 3,000, 4,000, 5,000, or 6,000 cartridge slots per library storage module (LSM)
  - Up to 24 LSMs per library (144,000 cartridges)
  - 120 TB (1 LSM) to 28,800 TB capacity (24 LSM)
  - Each cartridge holds 300GB, readable up to 40 MB/sec
- Up to 28.8 petabytes
- Ave 4s to load tape

- 2017 product: Oracle SL8500
- Up to 1.2 Exabyte per unit
- Combine up to 32 units into single robot tape drive system
- http://www.oracle.com/us/products/servers-storage/storage/tape-storage/034341.pdf

https://www.itnews.com.au/gallery/inside-suns-multi-storey-colorado-data-centre-135385/page1

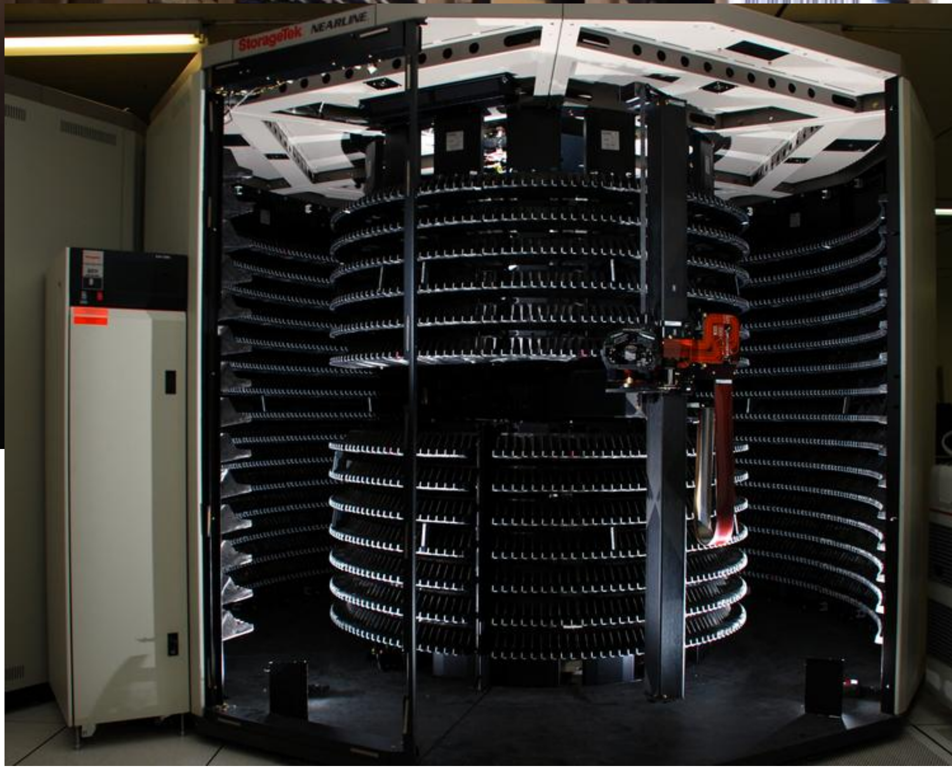


## IBM System Storage Tape Library ts3500 ts4500

From
https://www.youtube.com/watch?v=CVN93H6EuAU&list=PLp5rLKqrfZu_EvvnFM1HDptl_n0k5Th_q

30-50TB/cartridge
~400MB/s throughput per drive
Up to 128 drives/library
Up to 17,550 cartridges/library
Up to 877.5 exabytes/library

Datasheet: https://www.ibm.com/downloads/documents/us-en/10c31775c6d400a0



StorageTek Powderhorn before disassembly, CERN 2007
http://www.flickriver.com/photos/naezmi/2074280052/#large

# Can we live without cache?

http://www.karo.com

- Interesting exception: Cray/Tera MTA, first delivered June 1999:
  - www.cray.com/products/systems/mta/
- Each CPU switches every cycle between 128 threads
- Each thread can have up to 8 outstanding memory accesses
- 3D toroidal mesh interconnect
- Memory accesses hashed to spread load across banks
- MTA-1 fabricated using Gallium Arsenide, not silicon
- “nearly un-manufacturable” (wikipedia)
- Third-generation Cray XMT:
  - http://www.cray.com/Products/XMT.aspx
  - YarcData's uRiKA (http://www.yarcdata.com/products.html)

# Summary:

- **Without caches we are in trouble**
  - **DRAM access times are commonly >100 cycles**
- **Without locality caches won't help**
- **Spatial vs temporal locality**

We will look at various techniques to exploit memory parallelism to overcome this – especially in GPUs

- **Direct-mapped**
- **Set-associative**
- **Associativity conflicts**

We will see similar structures, and issues, in branch predictors, prefetching etc

- **Policy questions:**
  - **Write-through**
  - **Write-back**
  - **Many more – see next chapter!**

We will see similar choices in cache coherency protocols for multicore

Next:

Discussion exercise – the “Turing Tax”

Then dynamic scheduling

Then a deeper dive into caches and the memory hierarchy

# In response to a student question:

- There is a tag for each 32-byte cache block (and in the 1KB cache, there would, as you say, be 32 blocks, since 1024=32x32).
- Two adjacent cache blocks could (normally will) hold 32-byte blocks from different parts of the memory.
- In a fully-associative cache we would have a tag and a tag comparator for every 32-byte block.
- In a direct-mapped cache, we have a tag for every block, but only one tag comparator.
- This is cheaper, faster and lower-power. But in order to make it work, we use some of the low-order address bits to index the cache - to select just one cache block. If its tag matches, we have a hit. If not, we don't. Similarly, when data is allocated into the cache. the same index bits are used to select the cache block that will be used (perhaps displacing whatever was there before).
- This means that different addresses that happen to have the same index bits map to the same cache block. So only one of them can be in the cache at the same time.

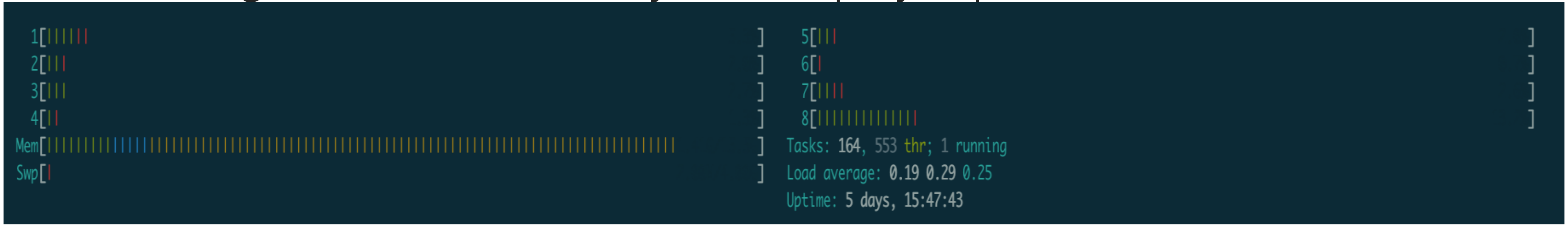
# Running long simulations… (helpful student’s edstem post)

Hello! Here is a quick list of tips I’ve picked up when running remote workloads on the lab machines.

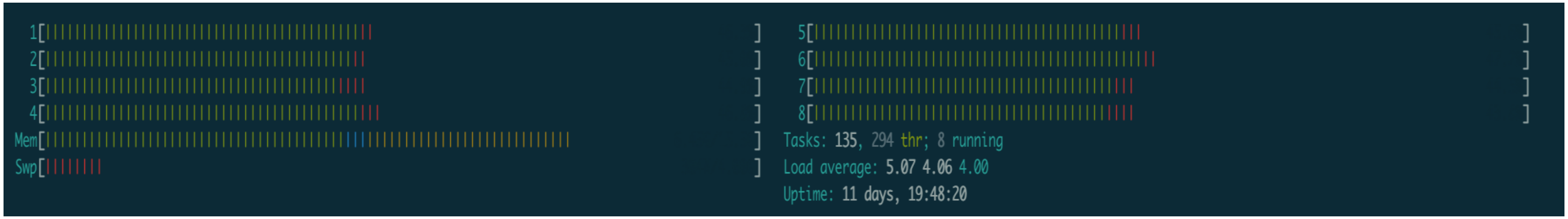
- I presume you are familiar with SSH-ing into the machines, normally I SSH with 2 hops (you should not run any apps on the shell servers, as far as I know they are meant only for accessing the DoC network): my laptop -> shell[1-5].doc.ic.ac.uk -> [your chosen lab machine]. Also, I have found that at rare times the shell servers can be slow. Either try another one or use the Imperial VPN and skip the first hop. Here is a list of lab machines for your convenience: link.

- Now, to find an empty lab machine, try to run `htop`.

- Here is some sample output, you can see in the first image an empty machine (low ram/CPU usage) and after that a busy one. To quit just press `q`.



```
  1[||||||                                                                                      ]   5[|||                                                                                                ]
  2[|||                                                                                         ]   6[|                                                                                                  ]
  3[|||                                                                                         ]   7[||||                                                                                               ]
  4[||                                                                                          ]   8[||||||||||||||                                                                                     ]
Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||         ]  Tasks: 164, 553 thr; 1 running
Swp[|                                                                                           ]  Load average: 0.19 0.29 0.25
                                                                                                   Uptime: 5 days, 15:47:43
```



```
  1[|||||||||||||||||||||||||||||||||||||||||||||                                               ]   5[||||||||||||||||||||||||||||||||||||||||||||||||                                                   ]
  2[||||||||||||||||||||||||||||||||||||||||||||                                                ]   6[||||||||||||||||||||||||||||||||||||||||||||||||||                                                 ]
  3[||||||||||||||||||||||||||||||||||||||||||||                                                ]   7[|||||||||||||||||||||||||||||||||||||||||||||||                                                    ]
  4[||||||||||||||||||||||||||||||||||||||||||||||                                              ]   8[|||||||||||||||||||||||||||||||||||||||||||||||                                                    ]
Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||                    ]  Tasks: 135, 294 thr; 8 running
Swp[||||||||                                                                                    ]  Load average: 5.07 4.06 4.00
                                                                                                   Uptime: 11 days, 19:48:20
```

# Running long simulations… (helpful student's edstem post)

- To make your session persistent you can use GNU Screen.

- Just type `screen` in your terminal and it should open up a new bash session.
- You can do your work there, and when you are ready to leave just type `CTRL-A-D` to minimise the session - it should now persist even if you log out!
- To list your sessions, type `screen -ls`.
- When you are ready to reconnect, just SSH into the same machine and run `screen -r` (potentially pass the name of the session as well if you have multiple). Don't forget to close your sessions when you are done (a simple `exit` will do).

Finally, a caveat: checkpoint your work - if your machine gets reset for whatever reason you will loose your sessions.

# Feeding curiosity

- Does LRU have pathological worst-case behaviour? How much worse might it be than an optimal replacement policy?
  - See: *Some Mathematical Facts About Optimal Cache Replacement*, Pierre Michaud, ACM TACO 2016 https://dl.acm.org/doi/pdf/10.1145/3017992
- What could be better than LRU?
  - See: *High performance cache replacement using re-reference interval prediction (RRIP),* Aamer Jaleel et al, ISCA'10 https://dl.acm.org/doi/abs/10.1145/1815961.1815971
- Can we reason about the efficiency of programs in a way that takes into account how well they will use the memory hierarchy – can we reason about the asymptotic complexity of programs as the distance to memory grows?
  - See: *The Uniform Memory Hierarchy Model of Computation*, Alpern et al, Algorithmica 1994 https://link.springer.com/content/pdf/10.1007/BF01185206.pdf
- As memory gets bigger, latency gets worse. But we could pipeline it? Under what circumstances does an algorithm's time complexity depend on memory latency?
  - See: *On approximating the ideal random access machine by physical machines,* Bilardi et al JACM 2009 *https://dl.acm.org/doi/10.1145/1552285.1552288*