Design of Parallel and High-Performance Computing

Fall 2017

Lecture: Refresher on Caches

Instructor: Torsten Hoefler & Markus Püschel

TA: Salvatore Di Girolamo

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organization

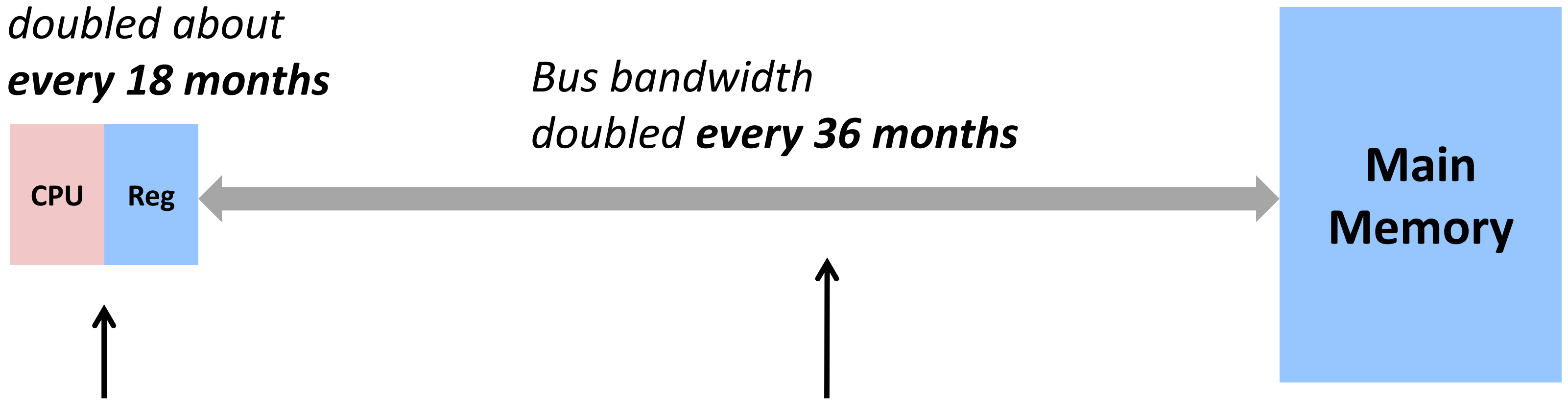
- **Temporal and spatial locality**
- **Memory hierarchy**
- **Caches**

*Chapter 5 in **Computer Systems: A Programmer's Perspective**, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010*
Part of these slides are adapted from the course associated with this book

Problem: Processor-Memory Bottleneck

Processor performance doubled about ***every 18 months***

Bus bandwidth doubled ***every 36 months***

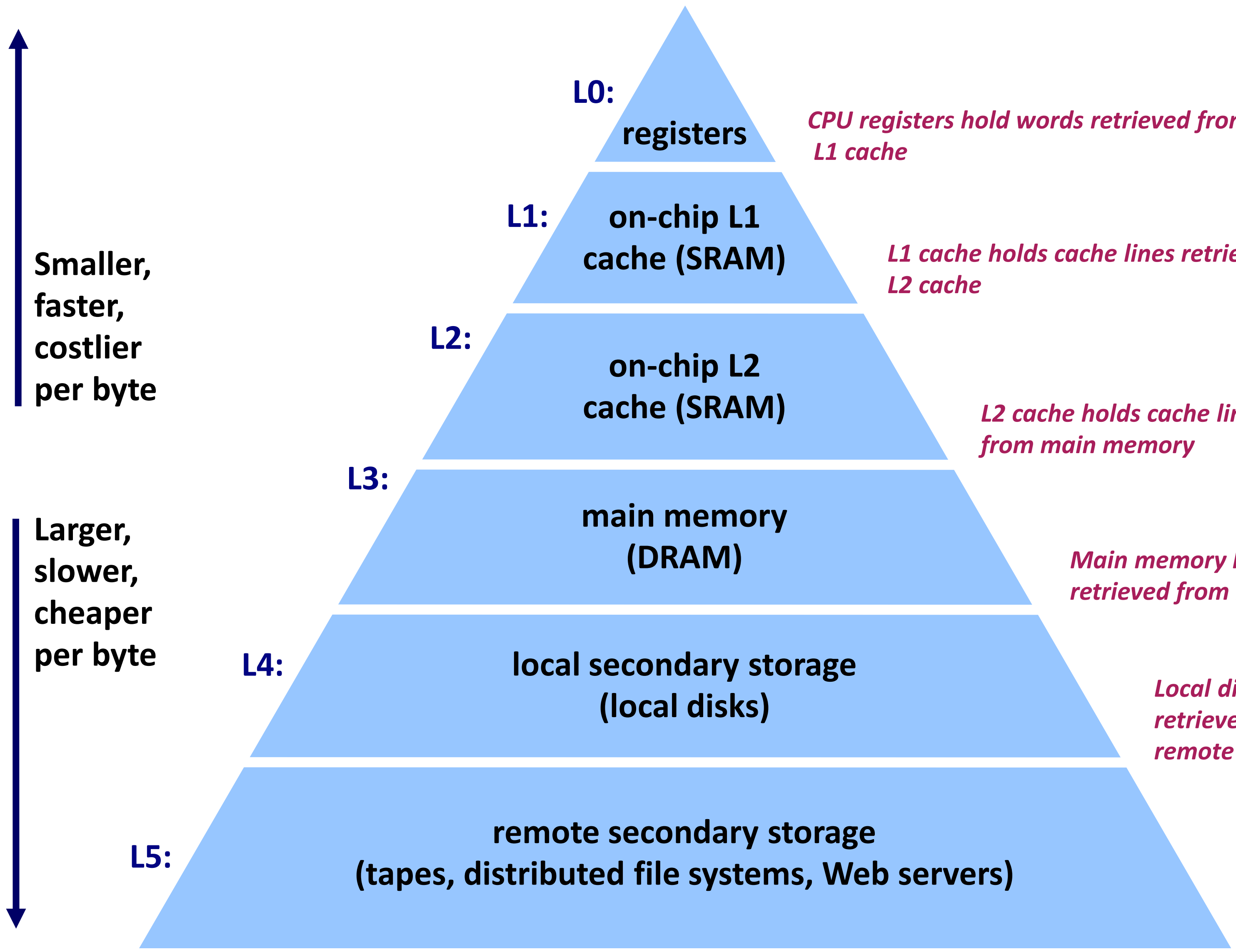


Core i7 Haswell:
Peak performance:
2 AVX three operand (FMA) ops/cycles
consumes up to 192 Bytes/cycle

Core i7 Haswell:
Bandwidth
16 Bytes/cycle

Solution: Caches/Memory hierarchy

Typical Memory Hierarchy



CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

The next slide is from the course “How to Write Fast Numerical Code”
http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring17/course.html
It contains additional information on latency and throughput of caches

Abstracted Microarchitecture: Example Core i7 Haswell (2013) and *Sandybridge (2011)*

Throughput (tp) is measured in doubles/cycle. For example: 4 *(2)*

Latency (lat) is measured in cycles

1 double floating point (FP) = 8 bytes

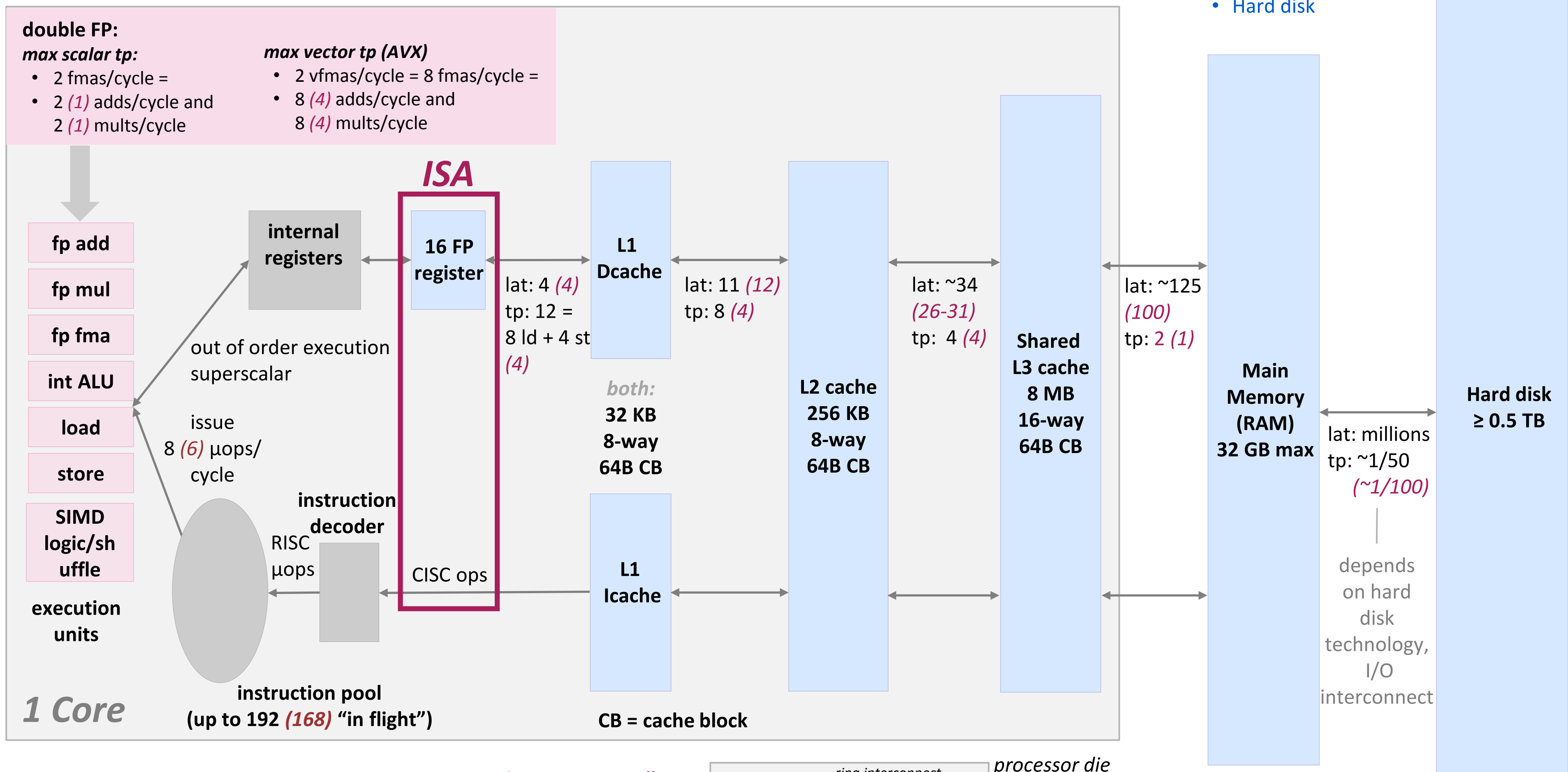
fma = fused multiply-add

Rectangles not to scale

Haswell *Sandy Bridge*

Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk



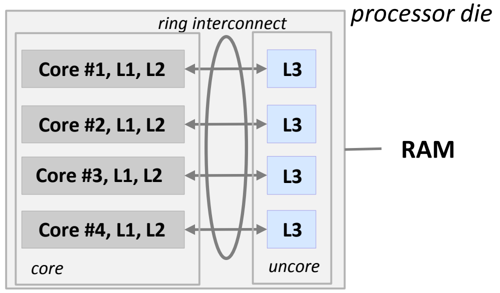
Core i7-4770 Haswell:

4 cores, 8 threads

3.4 GHz

(3.9 GHz max turbo freq)

2 DDR3 channels 1600 MHz

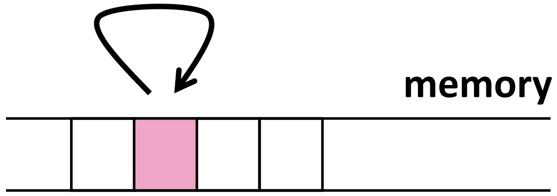


Source: Intel manual (chapter 2), http://www.7-cpu.com/cpu/Haswell.html

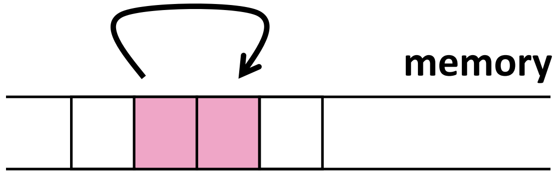
Why Caches Work: Locality

- ***Locality:*** **Programs tend to use data and instructions with addresses near or equal to those they have used recently**
  History of locality

- ***Temporal locality:***
  Recently referenced items are likely
  to be referenced again in the near future



- ***Spatial locality:***
  Items with nearby addresses tend
  to be referenced close together in time



Example: Locality?

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

- **Data:**
  - Temporal: **`sum`** referenced in each iteration
  - Spatial: array **`a[]`** accessed consecutively
- **Instructions:**
  - Temporal: loops cycle through the same instructions
  - Spatial: instructions referenced in sequence

- ***Being able to assess the locality of code is a crucial skill for a performance programmer***

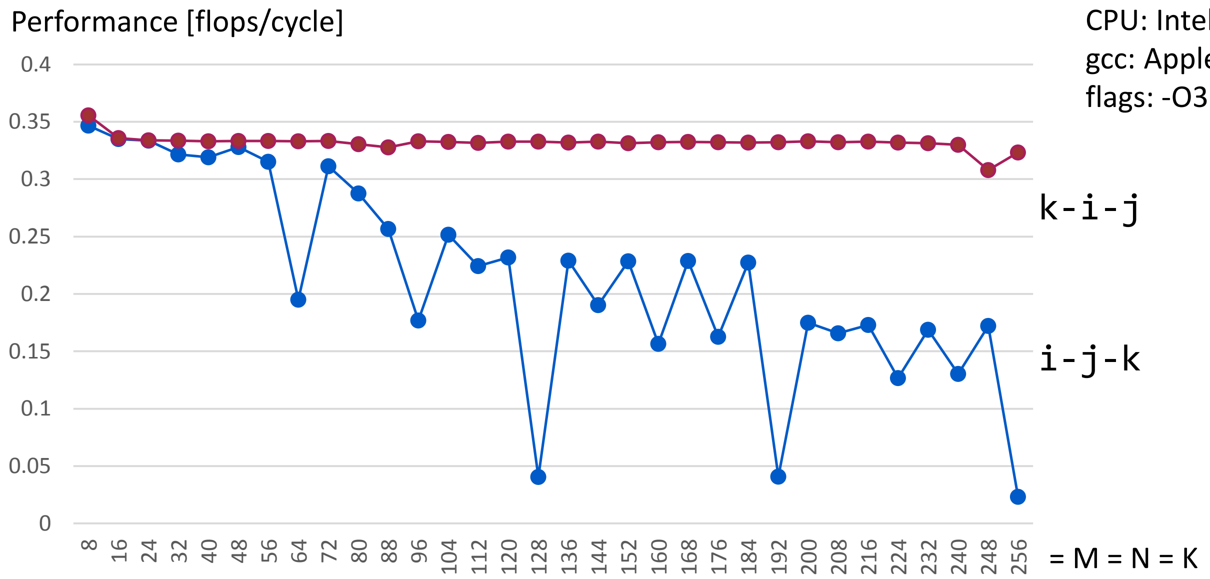
Locality Example

```
int sum_array_3d(double a[M][N][K])
{
  int i, j, k, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < K; k++)
        sum += a[k][i][j];
  return sum;
}
```

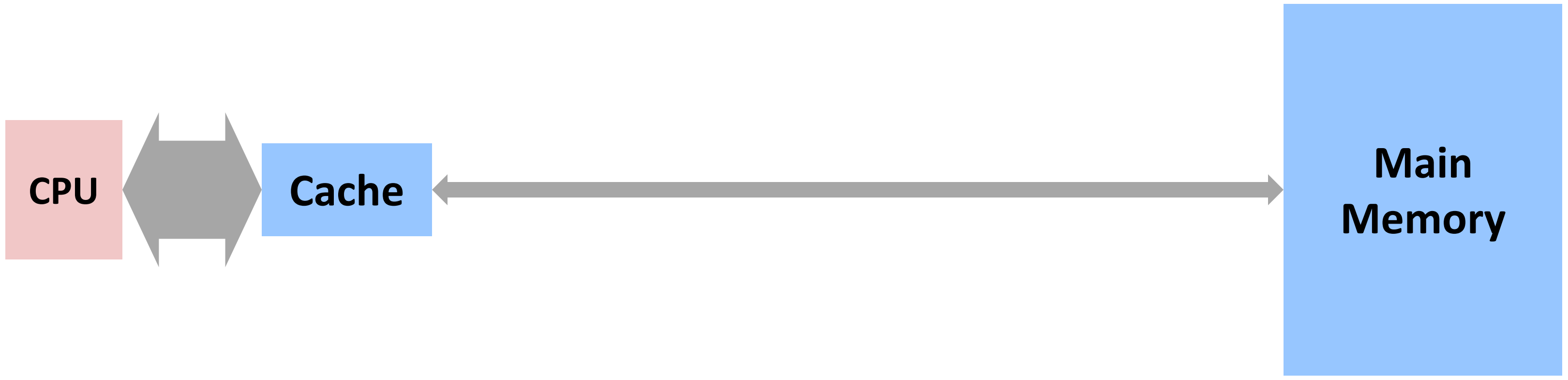
How to improve locality?

CPU: Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz
gcc: Apple LLVM version 8.0.0 (clang-800.0.42.1)
flags: -O3 -fno-vectorize



Cache

- *Definition:* **Computer memory with short access time used for the storage of frequently or recently used instructions or data**



- **Naturally supports** ***temporal locality***
- ***Spatial locality*** **is supported by transferring data in blocks**
  - Core family: one block = 64 B = 8 doubles

Cache Structure

- Add associativity (E = 2, B = 32 bytes, S = 8)
- Show how elements are mapped into cache

Example (S=4, E=2)

Ignore the variables sum, i, j

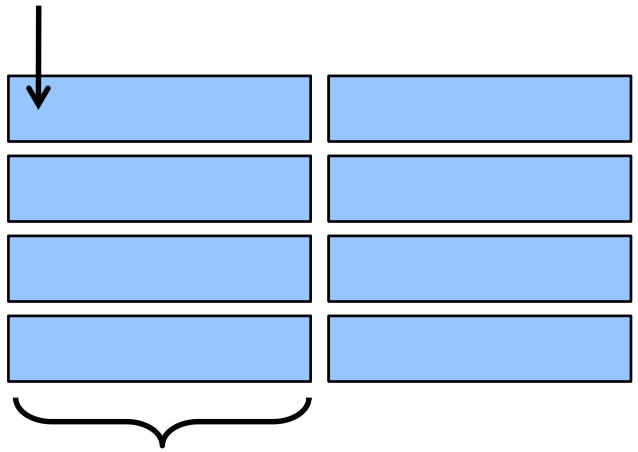
```
int sum_array_rows(double a[16][16])
{
  int i, j;
  double sum = 0;

  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)
      sum += a[i][j];
  return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
  int i, j;
  double sum = 0;

  for (j = 0; j < 16; i++)
    for (i = 0; i < 16; j++)
      sum += a[i][j];
  return sum;
}
```

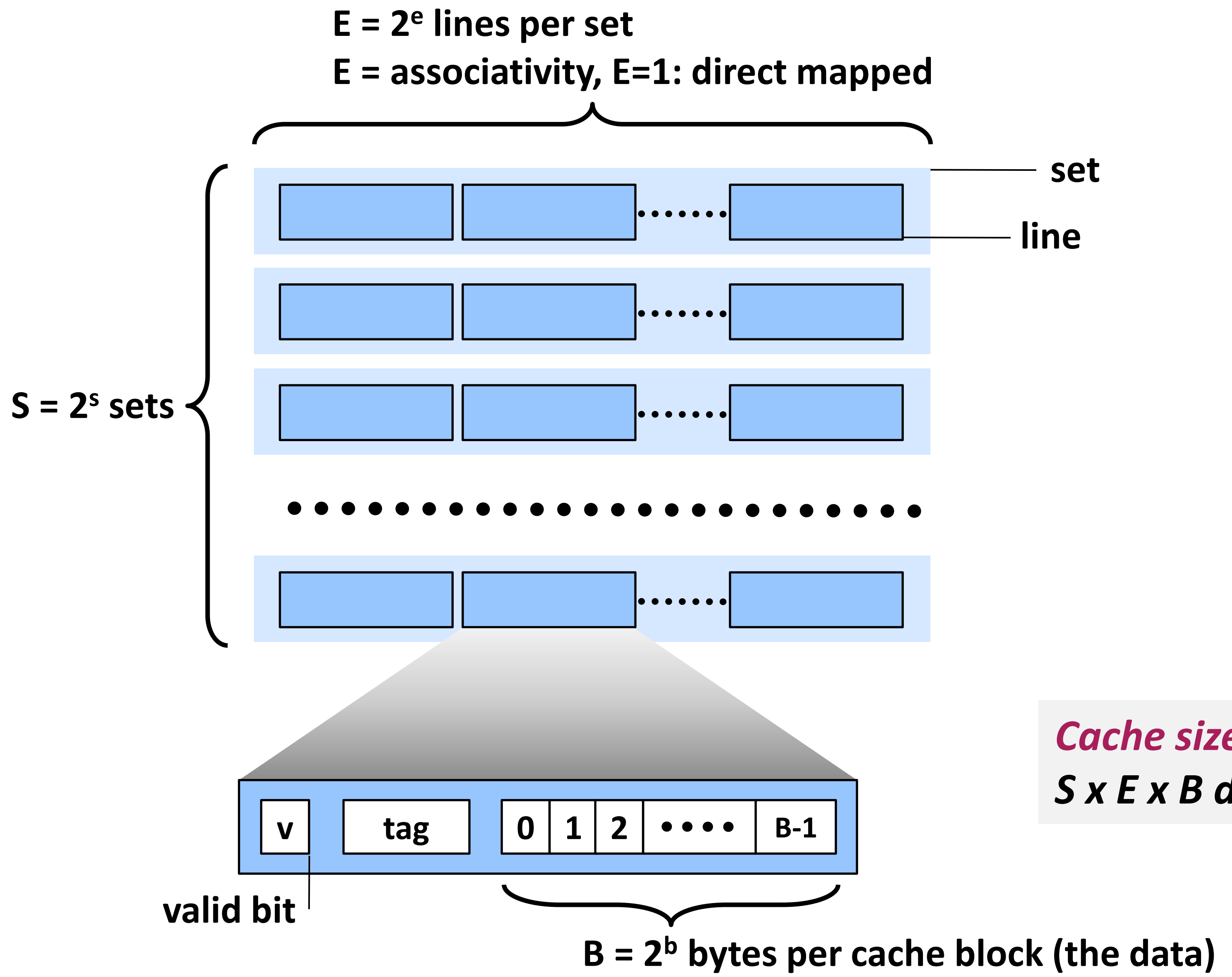
assume: cold (empty) cache,
a[0][0] goes here



B = 32 byte = 4 doubles

blackboard

General Cache Organization (S, E, B)

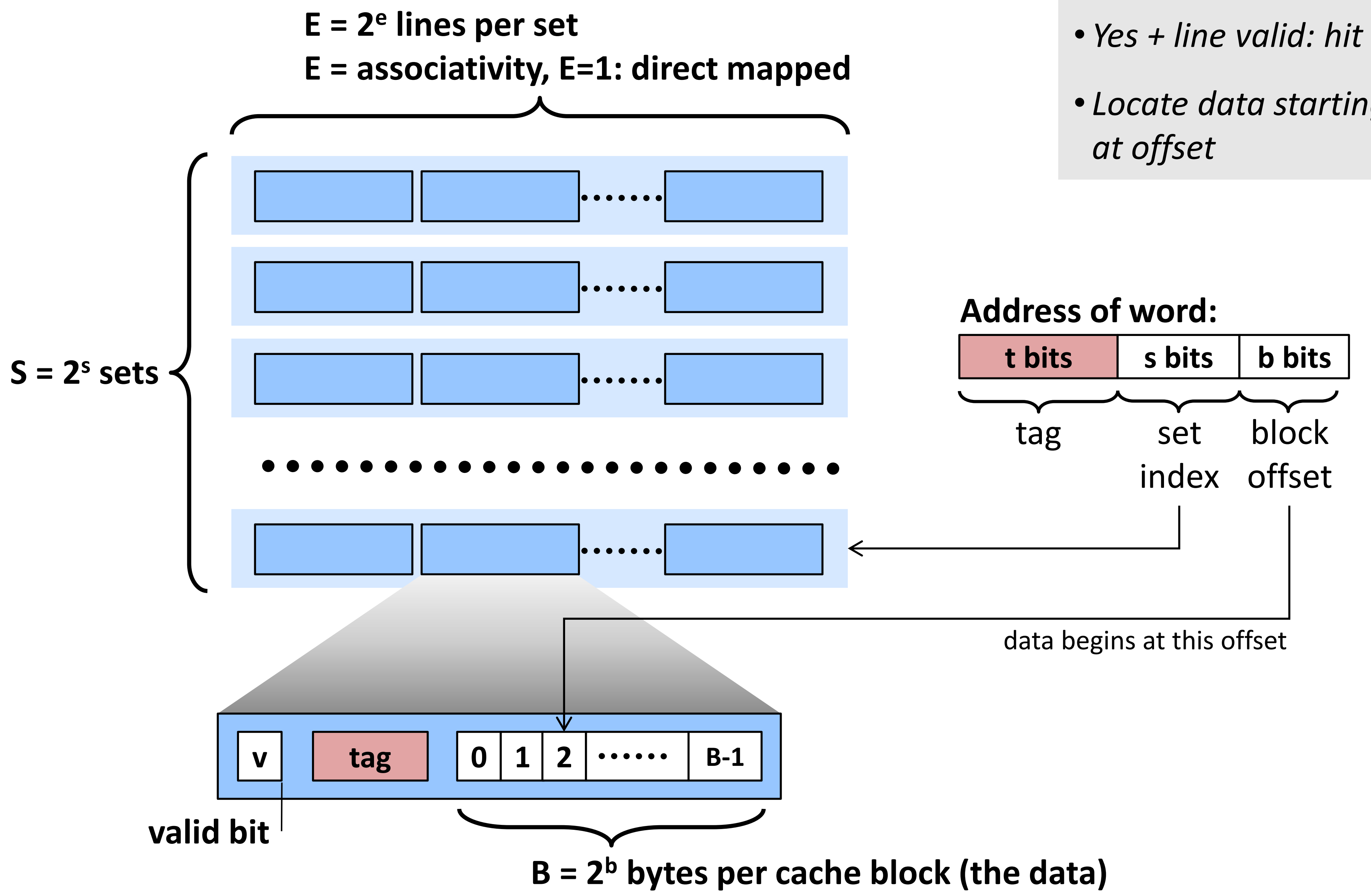


$E = 2^e$ lines per set
E = associativity, E=1: direct mapped
set
line
$S = 2^s$ sets
v
tag
0
1
2
B-1
valid bit
$B = 2^b$ bytes per cache block (the data)

Cache size:
S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Terminology

- **Direct mapped cache:**
  - Cache with E = 1
  - Means every block from memory has a unique location in cache
- **Fully associative cache**
  - Cache with S = 1 (i.e., maximal E)
  - Means every block from memory can be mapped to any location in cache
  - In practice to expensive to build
  - One can view the register file as a fully associative cache
- **LRU (least recently used) replacement**
  - when selecting which block should be replaced (happens only for E > 1), the least recently used one is chosen

Types of Cache Misses (The 3 C’s)

- ***Compulsory (cold)* miss**

  Occurs on first access to a block

- ***Capacity* miss**

  Occurs when working set is larger than the cache

- ***Conflict* miss**

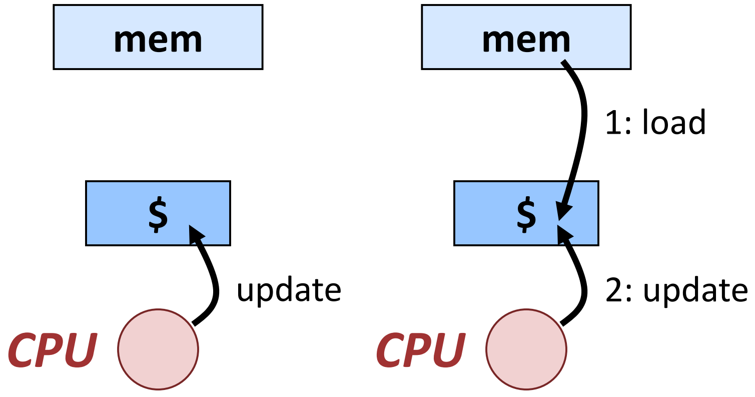
  Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

- **Not a clean classification but still useful**

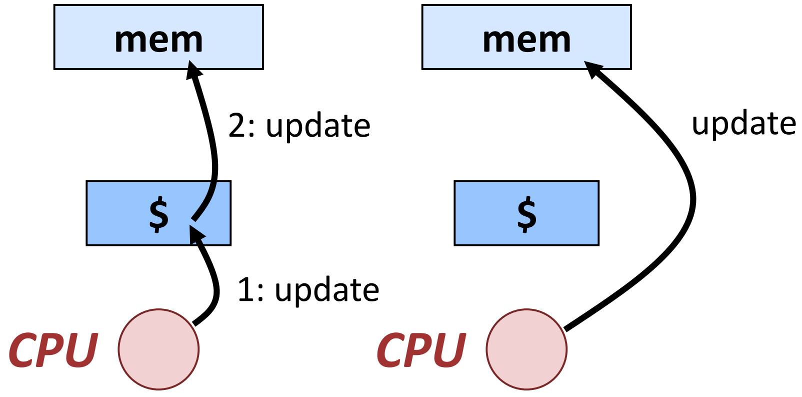
What about writes?

- **What to do on a write-hit?**
  - ***Write-through:*** write immediately to memory
  - ***Write-back:*** defer write to memory until replacement of line
- **What to do on a write-miss?**
  - ***Write-allocate:*** load into cache, update line in cache
  - ***No-write-allocate:*** writes immediately to memory

Write-back/write-allocate (Core) ***Write-through/no-write-allocate***



Write-hit **Write-miss**



Write-hit **Write-miss**

Example: (Blackboard)

- z = x + y, x, y, z vector of length n
- assume they fit jointly in cache + cold cache
- memory traffic Q(n)?
- operational intensity I(n)?

Summary

- **It is important to assess temporal and spatial locality in the code**
- **Cache structure is determined by three parameters**
  - block size
  - number of sets
  - associativity
- **You should be able to roughly simulate a computation on paper**