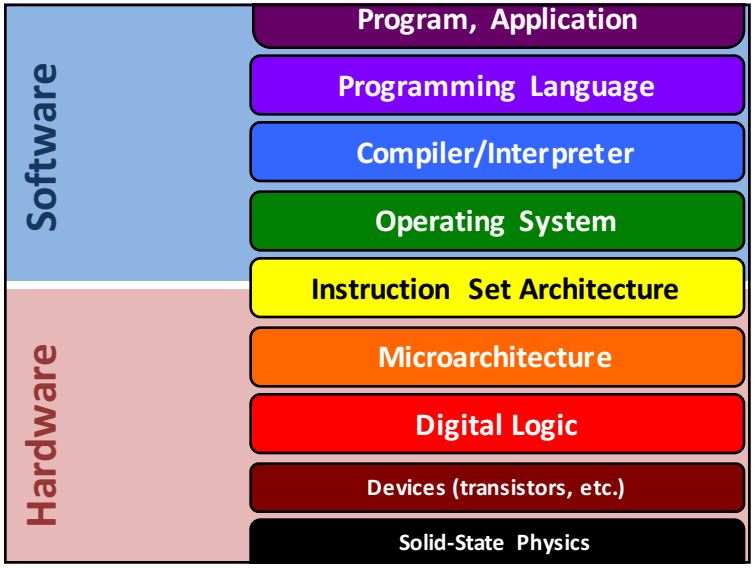
Memory Hierarchy: Cache

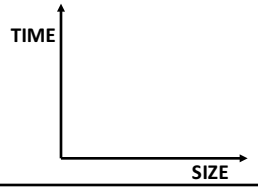
Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming



How does execution time grow with SIZE?

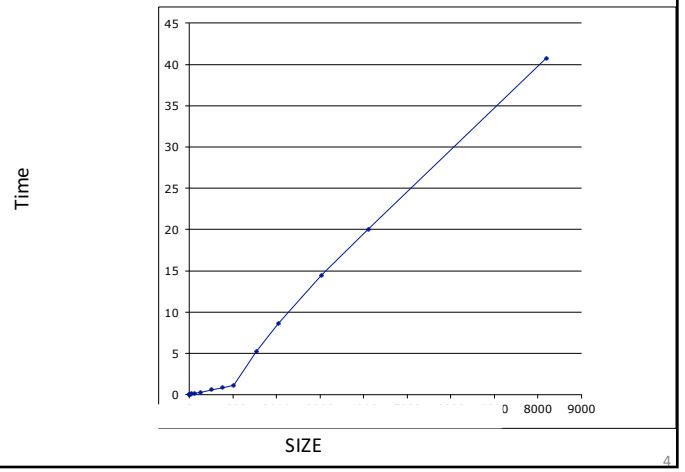
```
int[] array = new int[SIZE];
fillArrayRandomly(array);
int s = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    s += array[j];
  }
}
```



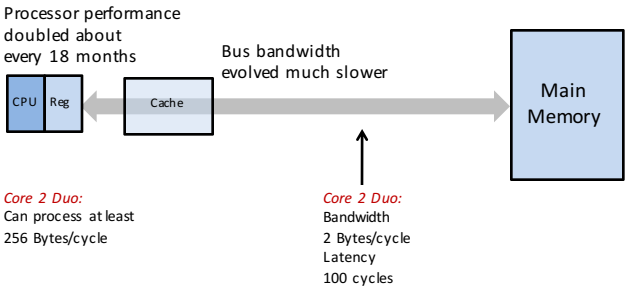
3

reality beyond O(...)



4

Processor-Memory Bottleneck



Solution: caches

5

Cache

English:

n. a hidden storage space for provisions, weapons, or treasures
v. to store away in hiding for future use

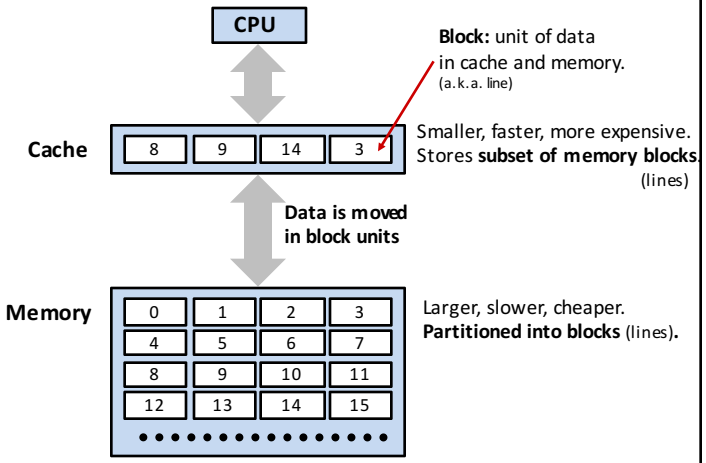
Computer Science:

n. a computer memory with short access time used to store frequently or recently used instructions or data
v. to store [data/instructions] temporarily for later quick retrieval

Also used more broadly in CS: software caches, file caches, etc.

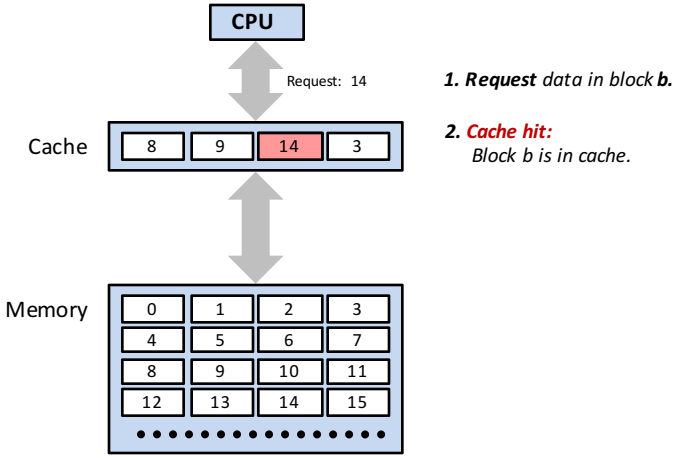
6

General Cache Mechanics



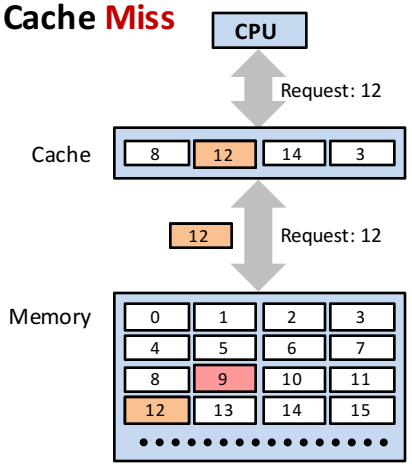
7

Cache Hit



8

Cache Miss



1. ***Request*** *data in block* ***b.***
2. *Cache miss:* *block is not in cache*
3. *Cache eviction:* *Evict a block to make room, maybe store to memory.*
4. *Cache fill:* *Fetch block from memory, store in cache.*

Placement Policy: where to put block in cache

***R*eplacement Policy:** which block to evict

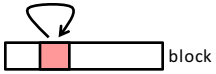
9

Locality: why caches work

Programs tend to use data and instructions at addresses near or equal to those they have used recently.

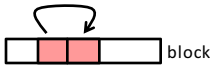
Temporal locality:

Recently referenced items are *likely* to be referenced again in the near future.



Spatial locality:

Items with nearby addresses are *likely* to be referenced close together in time.



How do caches exploit temporal and spatial locality?

10

Example: Locality?

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

What is stored in memory?

Data:

Temporal: **sum** referenced in each iteration

Spatial: array **a[]** accessed in ***stride-1*** pattern

Instructions:

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

Assessing locality in code is an important programming skill.

11

Locality Example #1

row-major M x N 2D array in C

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

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

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

12

Locality Example #2

row-major M x N 2D array in C

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

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

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

13

Locality Example #3

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

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

What is "wrong" with this code?

How can it be fixed?

14

Cost of Cache Misses

Huge difference between a hit and a miss

Could be 100x, if just L1 and main memory

99% hits could be twice as good as 97%. How?

Cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:

97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles

99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

hit/miss rate

This is why "miss rate" is used instead of "hit rate"

15

Cache Performance Metrics

Miss Rate

Fraction of memory accesses to data not in cache (misses / accesses)

Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

Hit Time

Time to find and deliver a block in the cache to the processor.

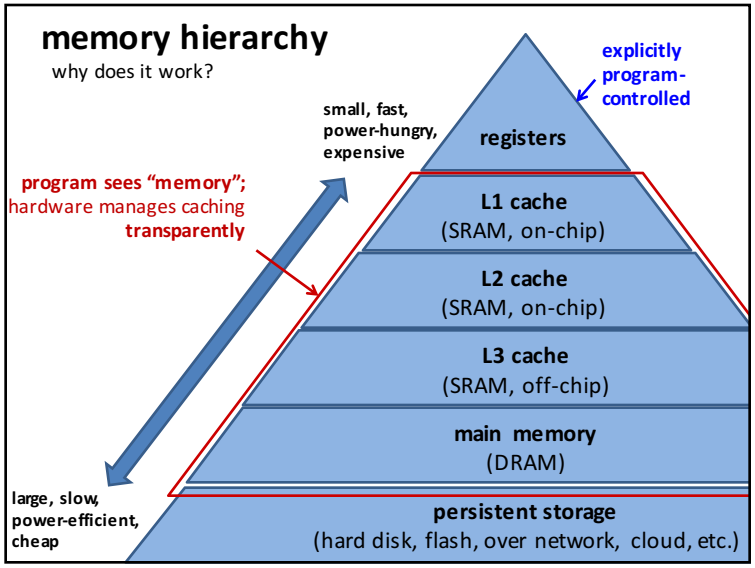
Typically: **1 - 2 clock cycles** for L1; **5 - 20 clock cycles** for L2

Miss Penalty

Additional time required on cache miss = main memory access time

Typically **50 - 200 cycles** for L2 *(trend: increasing!)*

16



Cache Organization: Key Points

Block

Fixed-size **unit of data** in memory/cache

Placement Policy

Where should a given block be stored in the cache?

- direct-mapped, set associative

Replacement Policy

What if there is no room in the cache for requested data?

- least recently used, most recently used

Write Policy

When should writes update lower levels of memory hierarchy?

- write back, write through, write allocate, no write allocate

Blocks

Divide memory into fixed-size aligned blocks.
power of 2

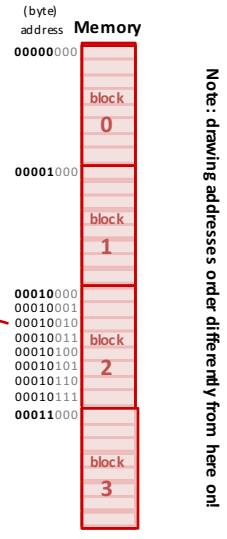
Example: block size = 8

full byte address

00010010

Block ID
address bits - offset bits

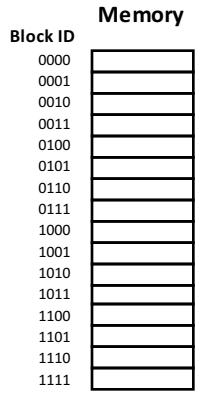
offset within block
$\log_2$(block size)



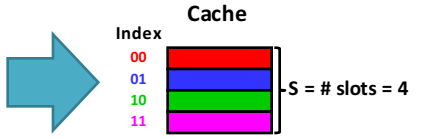
Note: drawing addresses order differently from here on!

remember withinSameBlock? (Pointers Lab)

Placement Policy



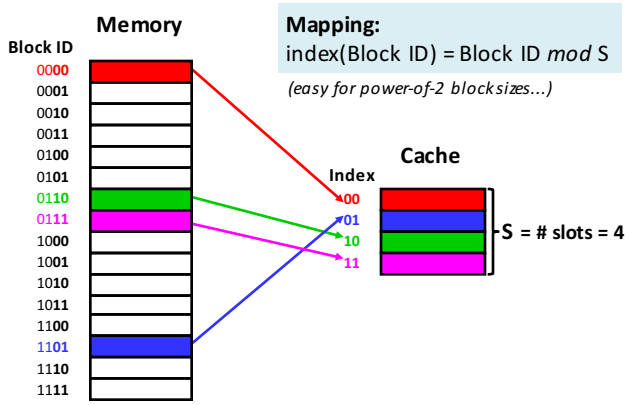
Mapping:
index(Block ID) = **???**



Small, fixed number of block slots.

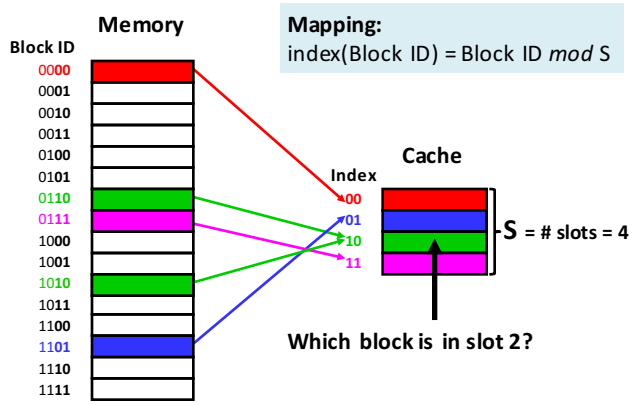
Large, fixed number of block slots.

Placement: *Direct-Mapped*



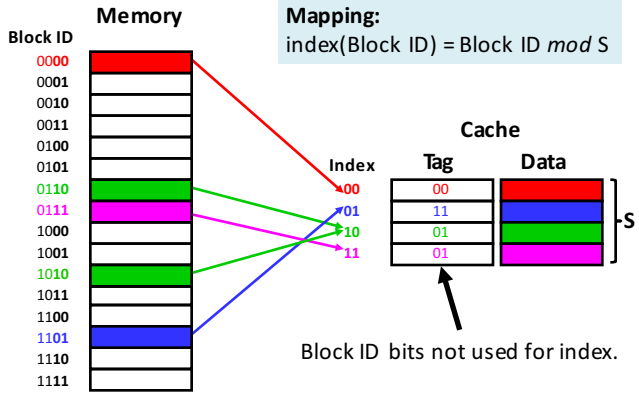
21

Placement: mapping ambiguity



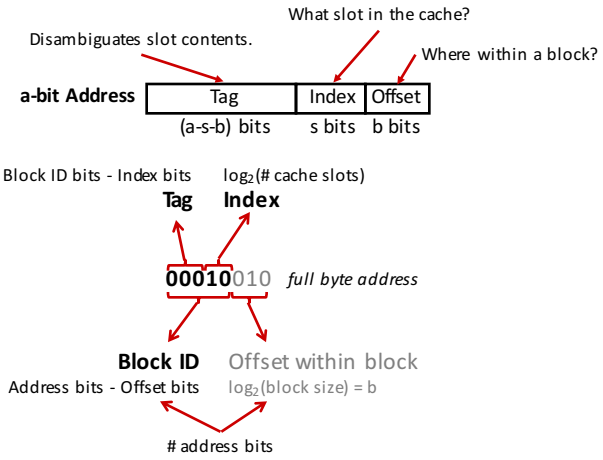
22

Placement: Tags resolve ambiguity

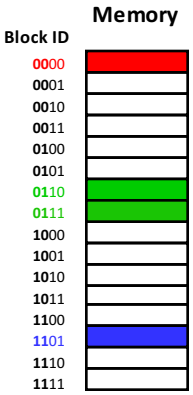


23

Address = Tag, Index, Offset

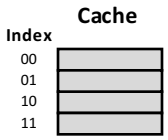


Placement: ~~Direct-Mapped~~



Why not this mapping?
index(Block ID) = Block ID / S

(still easy for power-of-2 block sizes…)

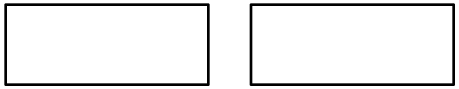


25

A puzzle.

Cache starts *empty.*
Access (address, hit/miss) stream:

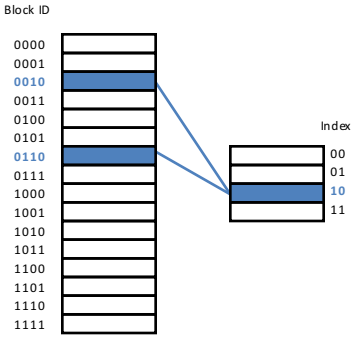
(10, miss), (11, hit), (12, miss)



What could the block size be?

26

Placement: direct mapping conflicts



What happens when accessing in repeated pattern:
0010, 0110, 0010, 0110, 0010…?

cache conflict
Every access suffers a miss, evicts cache line needed by next access.

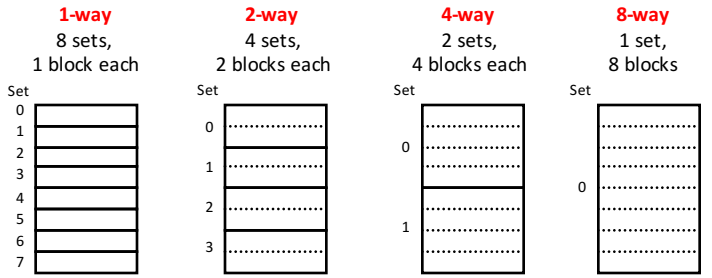
27

Placement: *Set Associative*

sets
S = # ~~slots~~ in cache

Index per ***set*** of block slots.
Store block in ***any*** slot within set.

Mapping:
index(Block ID) = Block ID *mod* S



28

Example: Tag, Index, Offset?

4-bit Address | Tag | Index | Offset |

Direct-mapped
4 slots
2-byte blocks

tag bits ____
set index bits ____
block offset bits ____

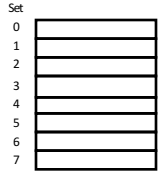
index(1101) = ____

Example: Tag, Index, Offset?

E-way set-associative
S slots
16-byte blocks

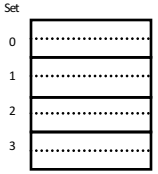
16-bit Address | Tag | Index | Offset |

E = 1-way
S = 8 sets



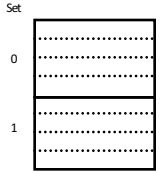
tag bits ____
set index bits ____
block offset bits ____
index(0x1833) ____

E = 2-way
S = 4 sets



tag bits ____
set index bits ____
block offset bits ____
index(0x1833) ____

E = 4-way
S = 2 sets



tag bits ____
set index bits ____
block offset bits ____
index(0x1833) ____

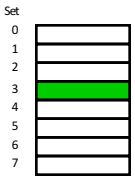
Replacement Policy

If set is full, what block should be replaced?

Common: **least recently used (LRU)**

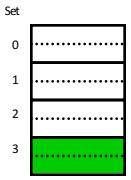
(hardware usually implements "not most recently used")

1-way associativity
8 sets, 1 block each

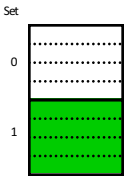


direct mapped

2-way associativity
4 sets, 2 blocks each



4-way associativity
2 sets, 4 blocks each

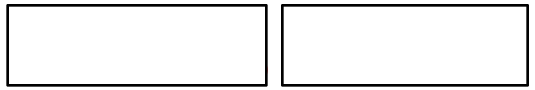


31

Another puzzle.

Cache starts *empty*, uses LRU.
Access (address, hit/miss) stream

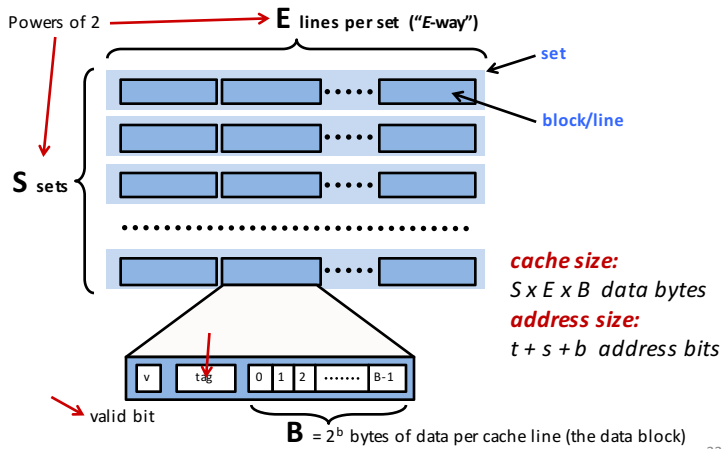
(10, miss); (12, miss); (10, miss)



associativity of cache?

32

General Cache Organization (S, E, B)

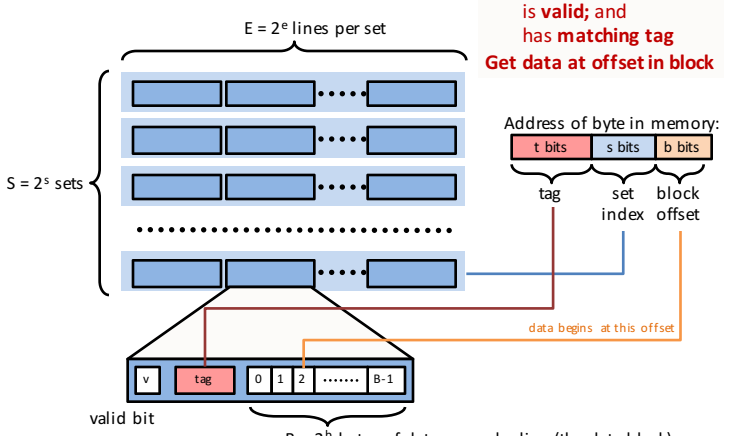


cache size:
S x E x B data bytes
address size:
t + s + b address bits

33

Cache Read

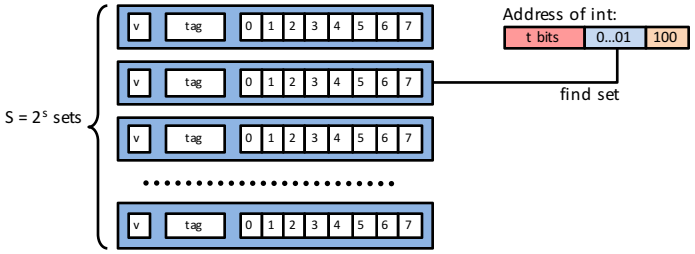
Locate set by index
Hit if any line in set:
is **valid;** and
has **matching tag**
Get data at offset in block



34

Read: Direct-Mapped (E = 1)

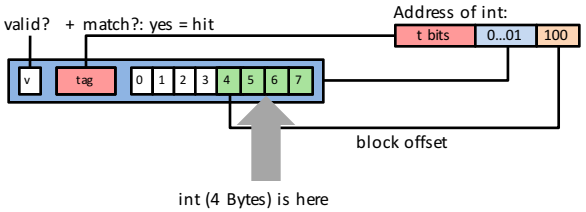
Direct-mapped: One line per set
Assume: cache block size 8 bytes



35

Read: Direct-Mapped (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes



If no match: old line is evicted and replaced

36

Example (E = 1)

Assume sum, i, j in registers
Address of an aligned element of a: `aa...arrrrcccc000`

```
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;
}
```

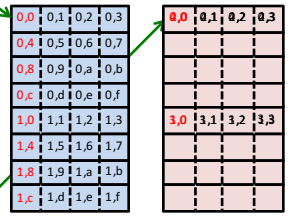
```
int sum_array_cols(double a[16][16]){
    int i, j;
    double sum = 0;

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

Assume: cold (empty) cache
3-bit set index, 5-bit offset

`aa...arrr rcc cc000`

0,0: aa...a000 000 00000



32 bytes = 4 doubles

4 misses per row of array
4*16 = 64 misses

32 bytes = 4 doubles

every access a miss
16*16 = 256 misses

37

Example (E = 1)

block = 16 bytes; 8 sets in cache
How many block offset bits?
How many set index bits?

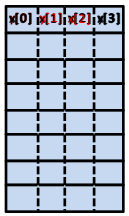
```
int dotprod(int x[8], int y[8]) {
    int sum = 0;
    int i;

    for (i = 0; i < 8; i++)
sum += x[i]*y[i];
    return sum;
}
```

Address bits:
B =
S =

0:
128:
160:

if x and y have aligned starting addresses,
e.g., &x[0] = 0, &y[0] = 128

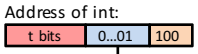


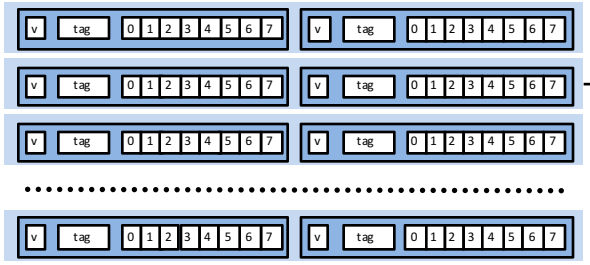
if x and y have unaligned starting addresses,
e.g., &x[0] = 0, &y[0] = 160

38

Read: Set-Associative (Example: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

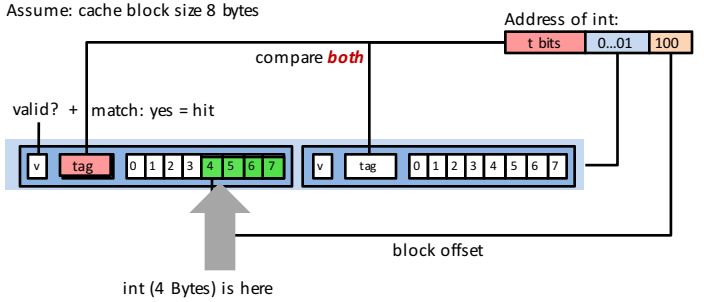




39

Read: Set-Associative (Example: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



If no match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

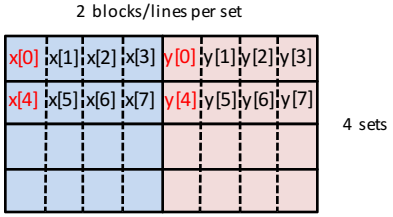
40

Example (E = 2)

```
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
sum += x[i]*y[i];
    return sum;
}
```

If x and y aligned,
e.g. &x[0] = 0, &y[0] = 128,
can still fit both because each set
has space for two blocks/lines



41

Types of Cache Misses

Cold (compulsory) miss
first access to a block

Conflict miss
cache has space for all needed blocks, but multiple blocks map to same slot
e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
increasing associativity can reduce conflict misses

Capacity miss
working set of active cache blocks is larger than the cache

42

What about writes?

Multiple copies of data exist:
L1, L2, possibly L3, main memory

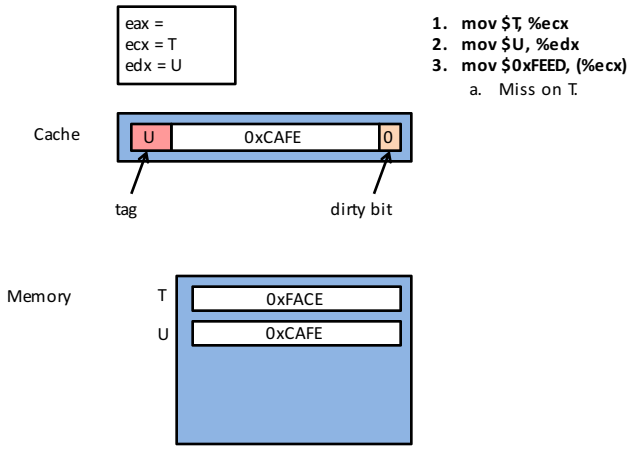
Write-hit policy
Write-through: write immediately to memory, all caches in between.
Write-back: defer write to memory until line is evicted (replaced)
Need a ***dirty bit*** to indicate if line is different from memory or not

Write-miss policy
Write-allocate: load into cache, update line in cache.
Good if more nearby writes or reads follow
No-write-allocate: just write immediately to memory.

Typical caches:
Write-back + Write-allocate, usually
Write-through + No-write-allocate, occasionally

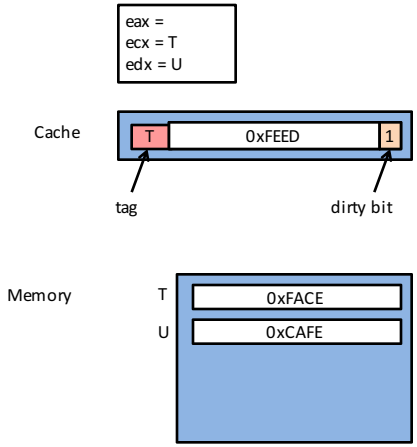
43

Write-back, write-allocate example



44

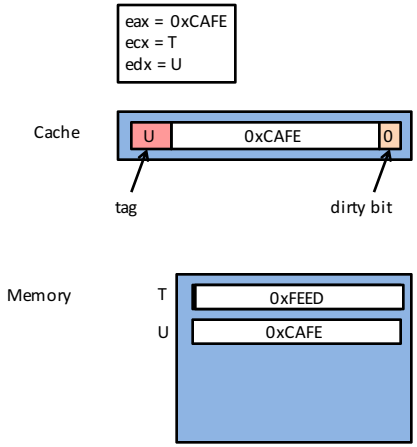
Write-back, write-allocate example



1. **mov $T, %ecx**
2. **mov $U, %edx**
3. **mov $0xFEED, (%ecx)**
   a. Miss on T.
   b. Evict U (clean: discard).
   c. Fill T (write-allocate).
   d. Write T in cache (dirty).
4. **mov (%edx), %eax**
   a. Miss on U.

45

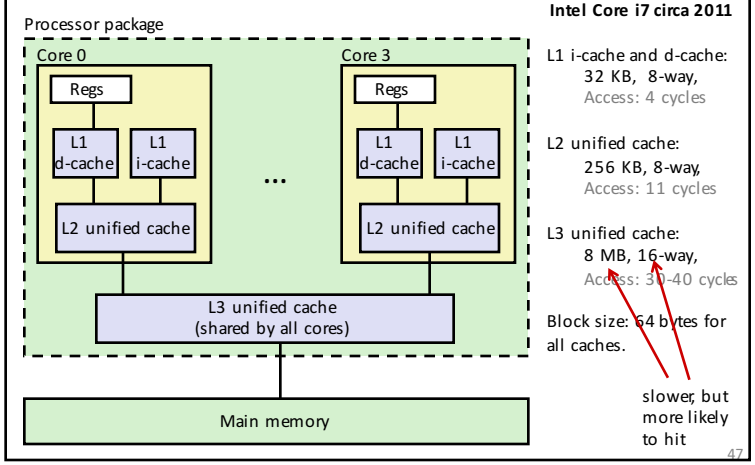
Write-back, write-allocate example



1. **mov $T, %ecx**
2. **mov $U, %edx**
3. **mov $0xFEED, (%ecx)**
   a. Miss on T.
   b. Evict U (clean: discard).
   c. Fill T (write-allocate).
   d. Write T in cache (dirty).
4. **mov (%edx), %eax**
   a. Miss on U.
   b. Evict T (dirty: write back).
   c. Fill U.
   d. Set %eax.
5. **DONE.**

46

Example Memory Hierarchy



Intel Core i7 circa 2011

L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

slower, but more likely to hit

47

Aside: software caches

Examples

File system buffer caches, web browser caches, database caches, network CDN caches, etc.

Some design differences

- Almost always fully-associative
  - so, no placement restrictions
  - index structures like hash tables are common (for placement)
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single "block" transfers
  - may fetch or write-back in larger units, opportunistically

48

Cache-Friendly Code

Locality, locality, locality. ← locality

Programmer can optimize for cache performance

- Data structure organization
- Data access patterns
  - Nested loops
  - Blocking (see CSAPP 6.5)

All systems favor "cache-friendly code"

- Performance is hardware-specific
  - Cache size, line size, associativity, etc.
- Generic rules still capture most of advantages
  - Keep working set small (temporal locality)
  - Use small strides (spatial locality)
  - Focus on inner loop code

57