

# CS 4400

## Computer Systems

---

### LECTURE 12

*The memory hierarchy*

*Locality*

*Cache memory*

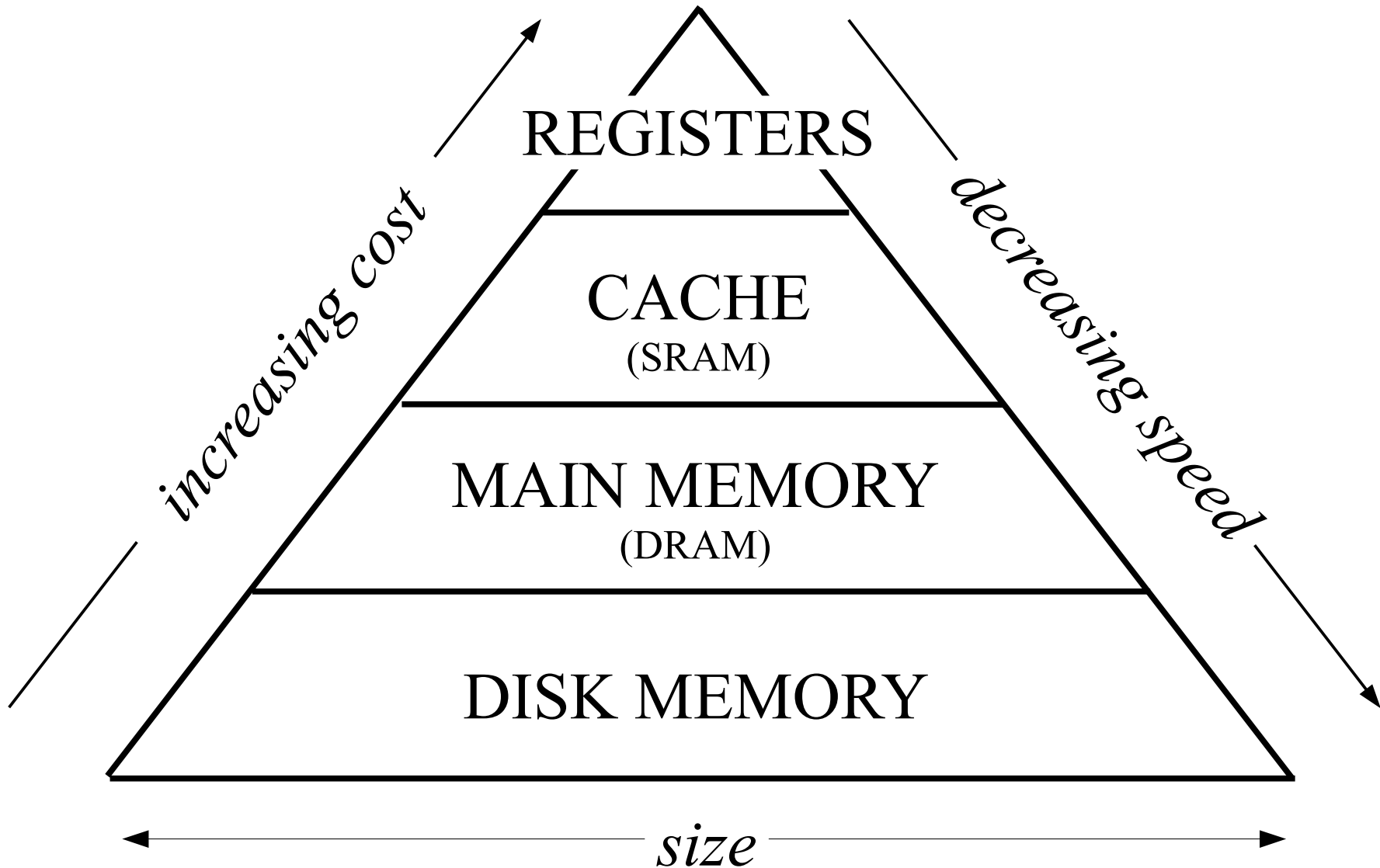
# Memory Model

---

- Up to this point, our model of the memory system has been overly simple.
  - Memory is a linear array of bytes.
  - The CPU accesses each location in constant amount of time.
- In practice, the memory system is a *hierarchy* of storage devices with different capacities, costs, and access times.
- Why do we care?
  - If we understand how data moves up and down the hierarchy, we can write programs whose data is stored higher in hierarchy.
- Our programs must have good *locality*.

# The Memory Hierarchy

---



# Storage Technologies

---

- Two varieties of random-access memory (RAM)
  - *Static*—used for on-and off-chip caches (fast, expensive)
  - *Dynamic*—used for main memory
- Typically, a desktop will have only a few megabytes of SRAM and thousands of megabytes of DRAM.
- SRAM cells have lower densities than DRAM cells (hence more expensive and consume more power).
- DRAM cells lose charge within 10-100 ms.
  - retention time is long compared to a clock cycle (ns)
  - memory system must refresh every bit of memory

# Storage Technologies

---

- DRAMs are organized as 2-d arrays.
  - row and column addresses are broadcast separately
  - increases access time
- DRAMs and SRAMs are *volatile*—they lose their information if the supply voltage is turned off.
- Nonvolatile memories are called read-only memory (ROM), even though some types can be written to.
- Disks hold huge amounts of data, but reading is slow.
  - *disk*: 10-100 GB, milliseconds
  - *RAM*: 100-1000 MB,  $\leq$  nanoseconds

# Locality

---

- Well-written programs tend to access items that are “near” other recently-accessed items.
  - This *principle of locality* has enormous impact on the design and performance of hardware and software systems.
- *Temporal locality*—accessing recently-referenced data.
- *Spatial locality*—accessing data with memory addresses near that of recently-referenced data.
- A program has *good locality* of reference if it can reuse data while it is still in the upper levels of memory.

# Exercise: Locality

---

- Does the reference to `sum` have good locality? If so, what kind(s)?
- Does the reference to `v` have good locality? If so, what kind(s)?

```
int sumvec(int v[N]) {
    int i, sum = 0;

    for(i = 0; i < N; i++)
        sum += v[i];

    return sum;
}
```

```
int sumarrayrows(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;
}
```

- Does the reference to `a` have good locality? If so, what kind(s)?
- What if we interchange the `i`- and `j`-loops?

# Cache Memory

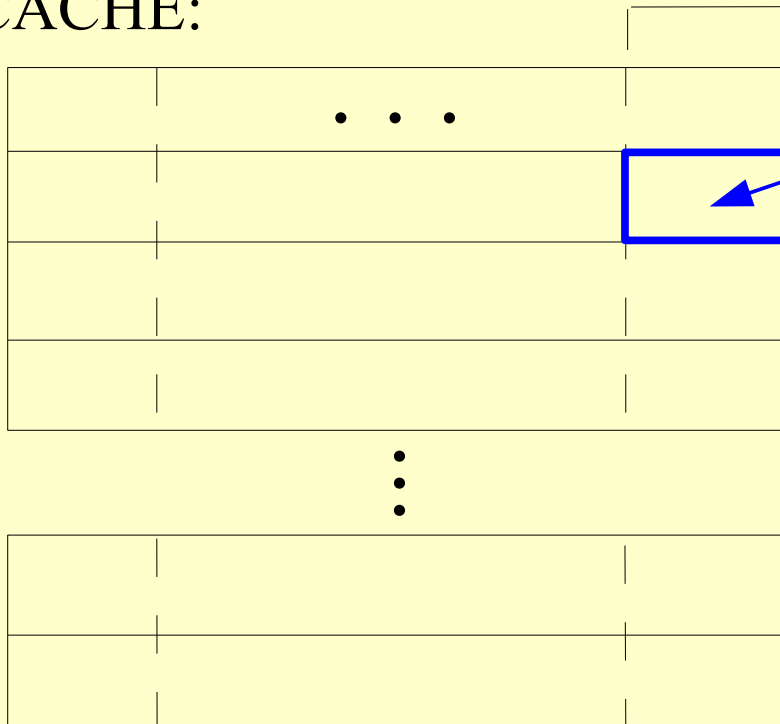
---

- *Caches* are small fast memories that hold blocks of the most recently accessed instructions and data.
- When the processor requires the datum at memory address  $x$ , it first looks in the cache.
  - If  $x$  is in the cache, a *cache hit* occurs.
  - If  $x$  is not in the cache, a *cache miss* occurs. The processor fetches  $x$  from main memory, placing a copy of  $x$  in the cache.
- Placing  $x$  in the cache may mean displacing another datum from the cache.



# Cache Organization

CACHE:

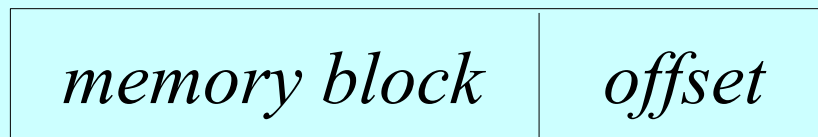


blocksize  $B$

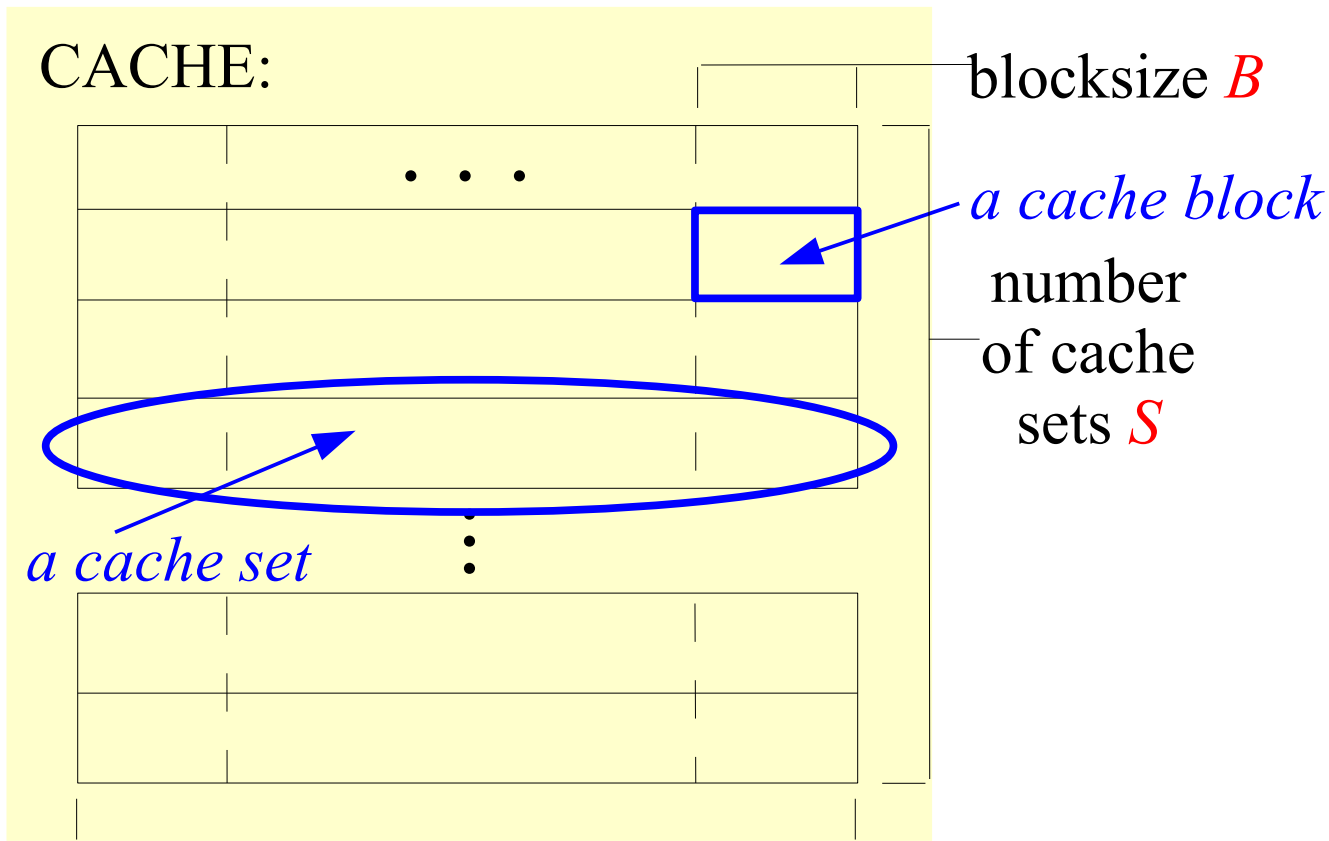
*a cache block*

- The binary expansion of a memory address is divided into a memory block address and an offset within the block.
- The cache blocksize  $B$  indicates how many contiguous bytes of memory are copied to cache.

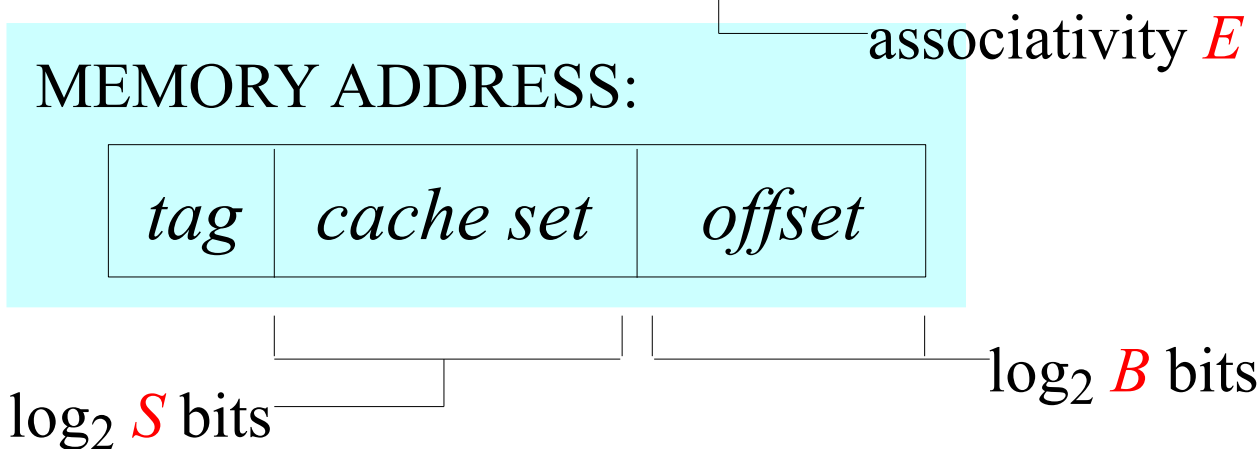
MEMORY ADDRESS:



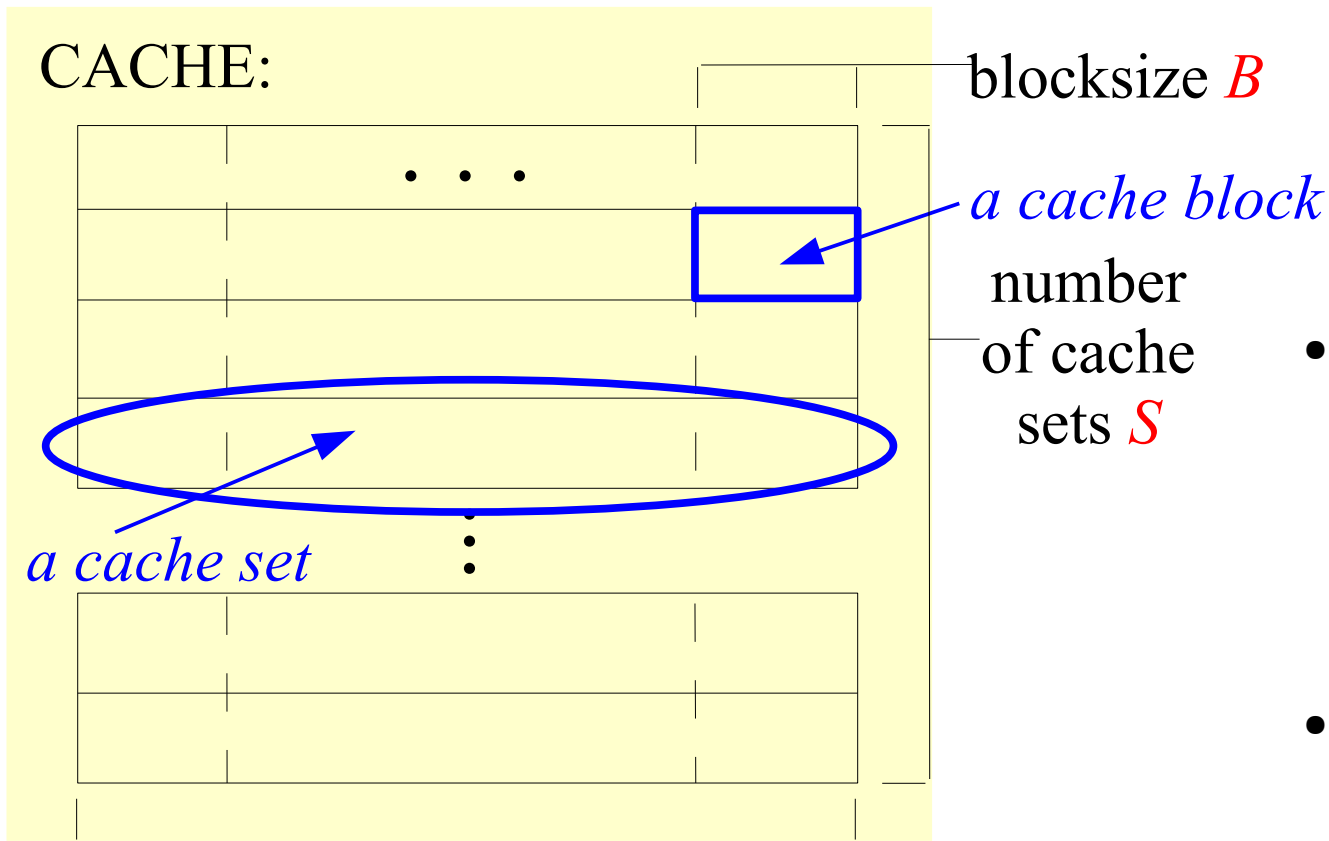
# Cache Organization



- A memory address is further divided into a cache set and a tag.
- The number of cache sets  $S$  indicates how many sets are in the entire cache.
- What is  $E$ ?



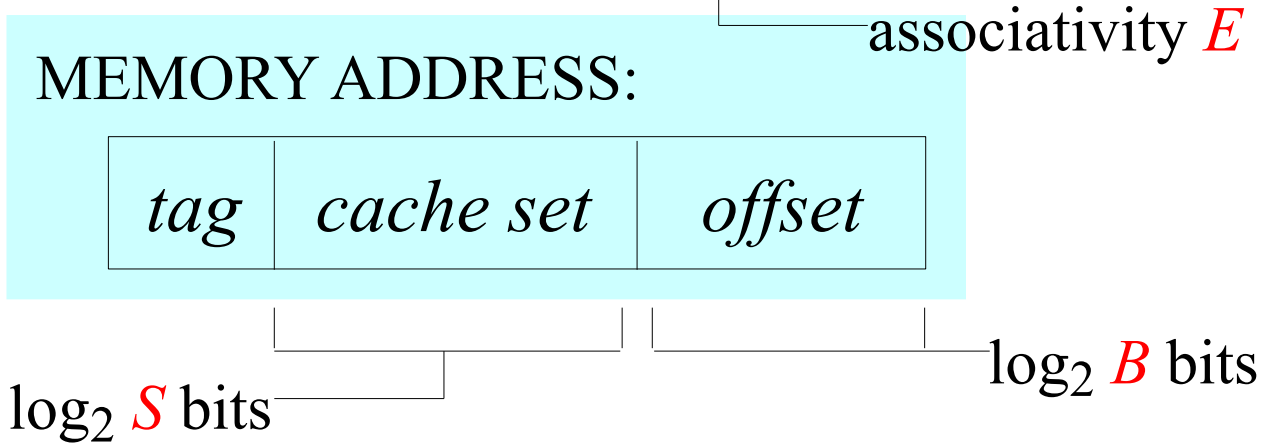
# Cache Organization



- The capacity of the entire cache is

$$C = E * B * S.$$

- Why is the organization of the cache so seemingly complex?



# Mapping Memory to Cache

---

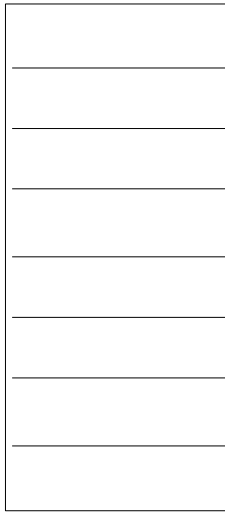
- A cache contains  $C/B$  *cache lines*, and each line may be empty or may be occupied by a memory block.
- No two cache lines may contain the same memory block. Why?
- A *cache set* is a collection of  $E$  lines that a particular memory block may occupy in cache.
- The *tag* disambiguates memory blocks within the same cache set.

# Cache Associativity

---

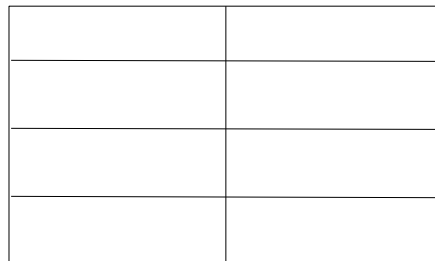
direct-mapped

$E=1, S=8$



2-way associative

$E=2, S=4$



4-way associative

$E=4, S=2$



fully associative

$E=8, S=1$



# Cache-Replacement Policy

---

- On a cache miss, the *replacement policy* selects which frame in a cache set to update with the new memory block. Optimal policy?
- *Random*: Choose randomly which block to replace.
- *First-in first-out (FIFO)*: Choose to replace the block that has resided in the cache set the longest.
- *Least-recently used (LRU)*: Choose to replace the block that has been unused the longest.

# Cache-Miss Categories

---

1. A *compulsory* (or cold) miss occurs on the very first access to a memory block. Why? Possible to avoid?
2. A *capacity* miss occurs when accessing a block that previously resided in cache, but was replaced because the cache cannot hold all of the data needed to execute a program. Possible to avoid?
3. A *conflict* miss occurs when accessing a block that previously resided in cache, but was replaced because too many blocks map to the same cache set. Possible to avoid?

# Cache-Miss Categories

---

- A capacity miss in an LRU set-associative cache with capacity  $C$  is also a miss in an LRU fully-associative cache with capacity  $C$ .
- A conflict miss in an LRU set-associative cache with capacity  $C$  is a hit in an LRU fully-associative cache with capacity  $C$ .
- *Why wouldn't we make all caches fully-associative, to avoid conflict misses?*



# Clicker Question

---

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

Consider a direct-mapped cache with 512 total bytes and 16-byte blocks. How many cache sets are there?

- A. 8 sets
- B. 16 sets
- C. 32 sets
- D. 64 sets
- E. There is not information to determine the number of sets.
- F. None of the above

# Clicker Question

---

Consider a fully-associative cache with 512 total bytes and 16-byte blocks. How many cache sets are there?

- A. 8 sets
- B. 16 sets
- C. 32 sets
- D. 64 sets
- E. There is not information to determine the number of sets.
- F. None of the above

# Exercise: Cache Parameters

$C$  – cache capacity (total number of bytes)

$B$  – cache block size

$E$  – cache associativity (number of blocks per set)

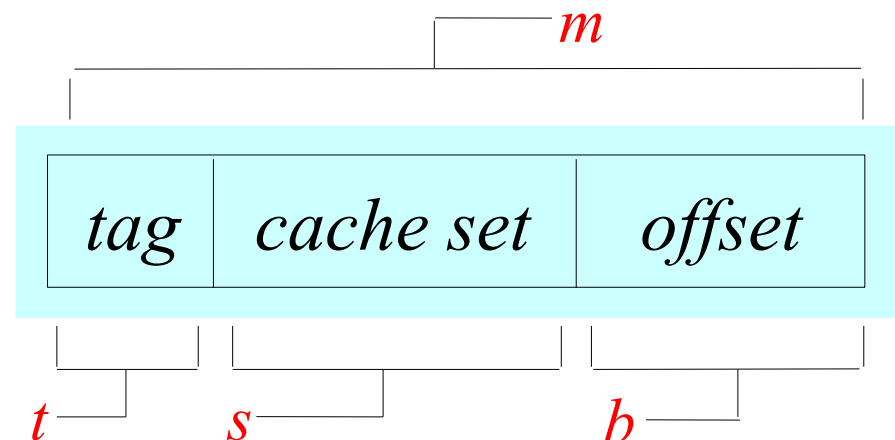
$S$  – number of cache sets

$m$  – number of main memory address bits

$t$  – number of tag bits ( $m - s - b$ )

$s$  – number of set index bits ( $\log_2 S$ )

$b$  – number of block offset bits ( $\log_2 B$ )



What are  $S$ ,  $t$ ,  $s$ , and  $b$ ?

- $m = 32$ ,  $C = 1024$ ,  $B = 4$ ,  $E = 1$
- $m = 32$ ,  $C = 1024$ ,  $B = 8$ ,  $E = 4$
- $m = 32$ ,  $C = 1024$ ,  $B = 32$ ,  $E = 32$

# Clicker Question

---

Consider a 4-way set-associative cache with 512 total bytes and 16-byte blocks. Which of the following addresses of memory blocks map to cache set 5?

- A. 0x0836
- B. 0x0845
- C. 0x0877
- D. 0x0900
- E. Two or more of the above
- F. None of the above

# Example: DM Cache

```
float dotprod(float x[8], float y[8]) {
    float sum = 0.0;
    int i;

    for(i = 0; i < 8; i++)
        sum += x[i] + y[i];

    return sum;
}
```

“Thrashing” occurs—  
the cache is repeatedly  
loading and evicting  
the same sets of blocks.

How can thrashing be  
prevented?

$E=1, B=16, C=32, S=2$

`sizeof(float)` is 4, `x` starts at address 0, `y` starts at address 32

element	addr	set	miss?	element	addr	set	miss?
x[0]	0	0	cold	y[0]	32	0	cold
x[1]	4	0	conflict	y[1]	36	0	conflict
x[2]	8	0	conflict	y[2]	40	0	conflict
x[3]	12	0	conflict	y[3]	44	0	conflict
x[4]	16	1	cold	y[4]	48	1	cold
x[5]	20	1	conflict	y[5]	52	1	conflict
x[6]	24	1	conflict	y[6]	56	1	conflict
x[7]	28	1	conflict	y[7]	60	1	conflict

# Exercise: Associative Cache

---

- $E=2, B=4, C=64, S=8, m=13$
- Memory accesses are to 1-byte words.
- Which bits (12 to 0) are used to determine the cache block offset?  
     cache set index?                      cache tag?
- Suppose address `0x0E34` is referenced. Hit or miss?
- Reference to `0x0DD5`?                      Reference to `0x1FE4`?

LINE (or FRAME) 1

set	tag	valid	byte0	byte1	byte2	byte3
0	09	1	86	30	3F	10
1	45	1	60	4F	E0	23
2	EB	0	--	--	--	--
3	06	0	--	--	--	--
4	C7	1	06	78	07	C5
5	71	1	0B	DE	18	4B
6	45	1	A0	B7	26	2D
7	46	0	--	--	--	--

LINE (or FRAME) 2

set	tag	valid	byte0	byte1	byte2	byte3
0	00	0	--	--	--	--
1	38	1	00	BC	0B	37
2	0B	0	--	--	--	--
3	32	1	12	08	7B	AD
4	05	1	40	67	C2	3B
5	6E	0	--	--	--	--
6	F0	0	--	--	--	--
7	DE	1	12	C0	88	37