CS 4400
Computer Systems

LECTURE 12

The memory hierarchy

Locality

Cache memory

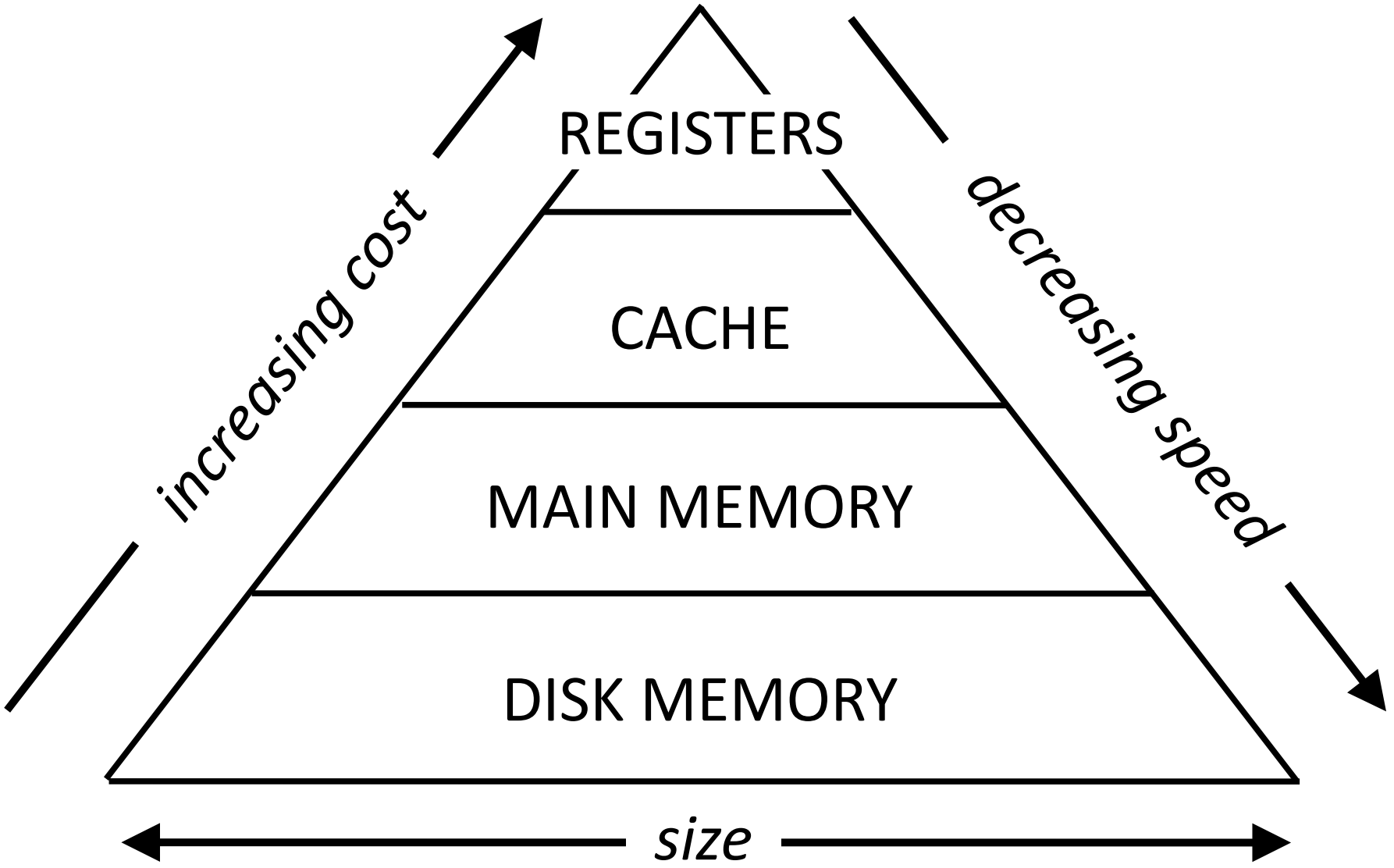
Memory Model

- Up to this point, our model of the memory system has been overly simple
  - Memory: linear array
  - Access: constant time

Memory Model

- Memory system is a ***hierarchy*** of storage devices with different capacities, costs, and access times.
- Why do we care?
  - Data movement is expensive
- 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, typically a few MB
  - Lower densities than DRAM cells
    - hence more expensive and consume more power
- Dynamic: used for main memory
  - Slower, but larger
  - Lose charge within 10-100 ms / memory must be refreshed
  - Organized as 2D arrays

Storage Technologies

- 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 large but reading/writing slow
  - Disk — size: ~1-10 TB, access: milliseconds
  - SSD — size: ~128-1024 GB, access: microseconds
  - RAM — size: ~8-128 GB, access: 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.
  - A program has good locality of reference if it can reuse data while in the upper levels of memory.

Locality

- ***Temporal locality***—accessing recently-referenced data
- ***Spatial locality***—accessing data with memory addresses near those of recently-referenced data

Exercise: Locality

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

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

  return sum;
}
```

- 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)?

Exercise: Locality

```
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.

Cache Memory

- 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 (***evicting***) another datum from the cache.

Simple Cache (fully associative cache)

- A ***fully associative*** cache contains set of storage locations that can hold any data element
- Cached data is stored in ***blocks*** or ***cache lines***
  - *Every cache has a fixed block size* ***B***
- A cache of size ***C***, contains ***E*** =***C***/***B*** cache lines
  - Each cache line stores a base address and the block of data

Eviction Policy

- When a cache is full and we need to make space, we evict another data element using an ***eviction policy*** or ***replacement policy***
- Common policies:
  - ***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.
- Is there an optimal policy?

Basics of Associative Sets (direct-mapped cache)

- A ***direct-mapped*** cache contains the same ***S =C/B*** cache lines as a fully associative
- However, now every address maps exclusively to a single cache line
- Advantages/Disadvantages?
- Example

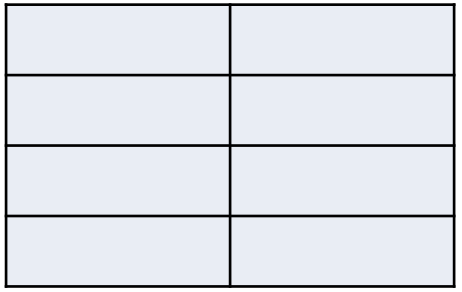
Cache Associativity

direct-mapped

E=1, *S*=8

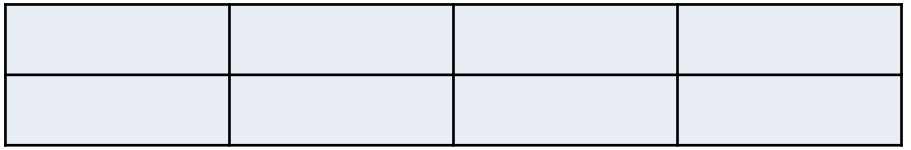
2-way associative

E=2, *S*=4



4-way associative

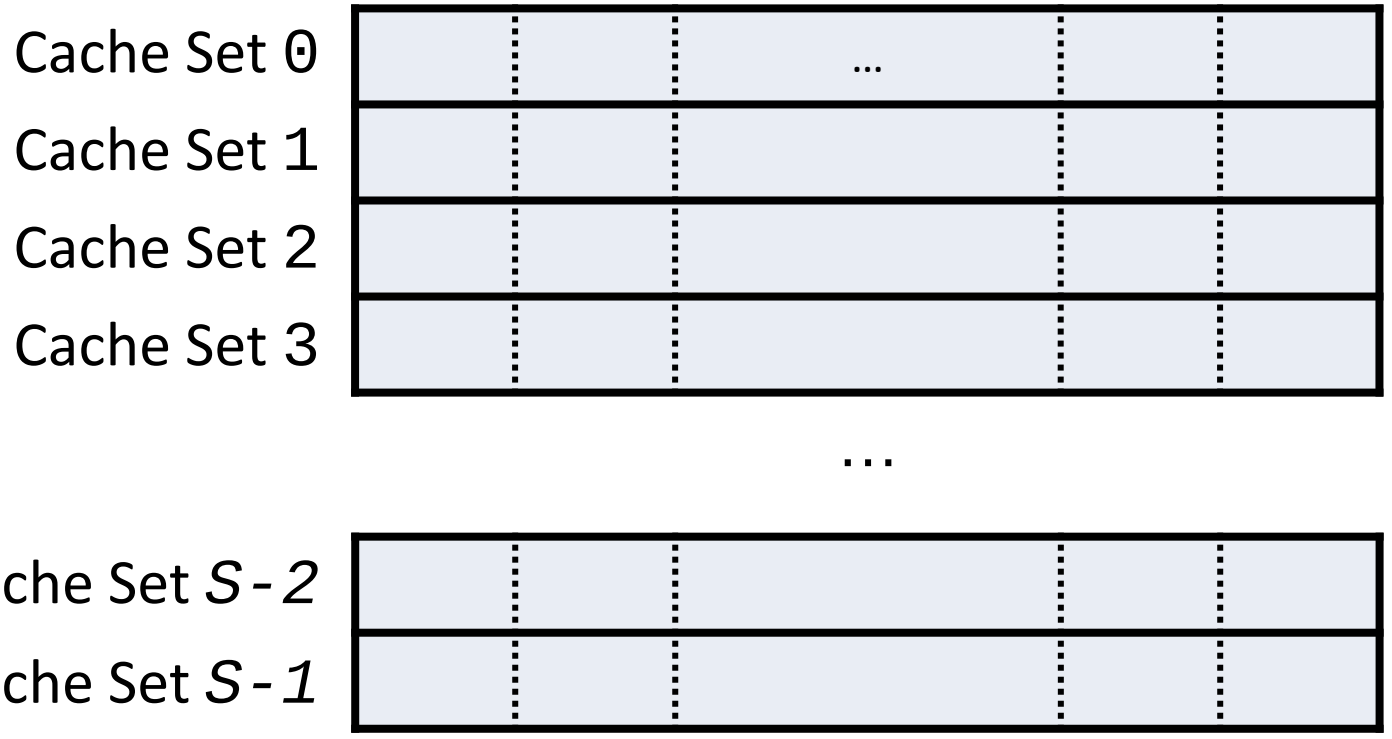
E=4, *S*=2



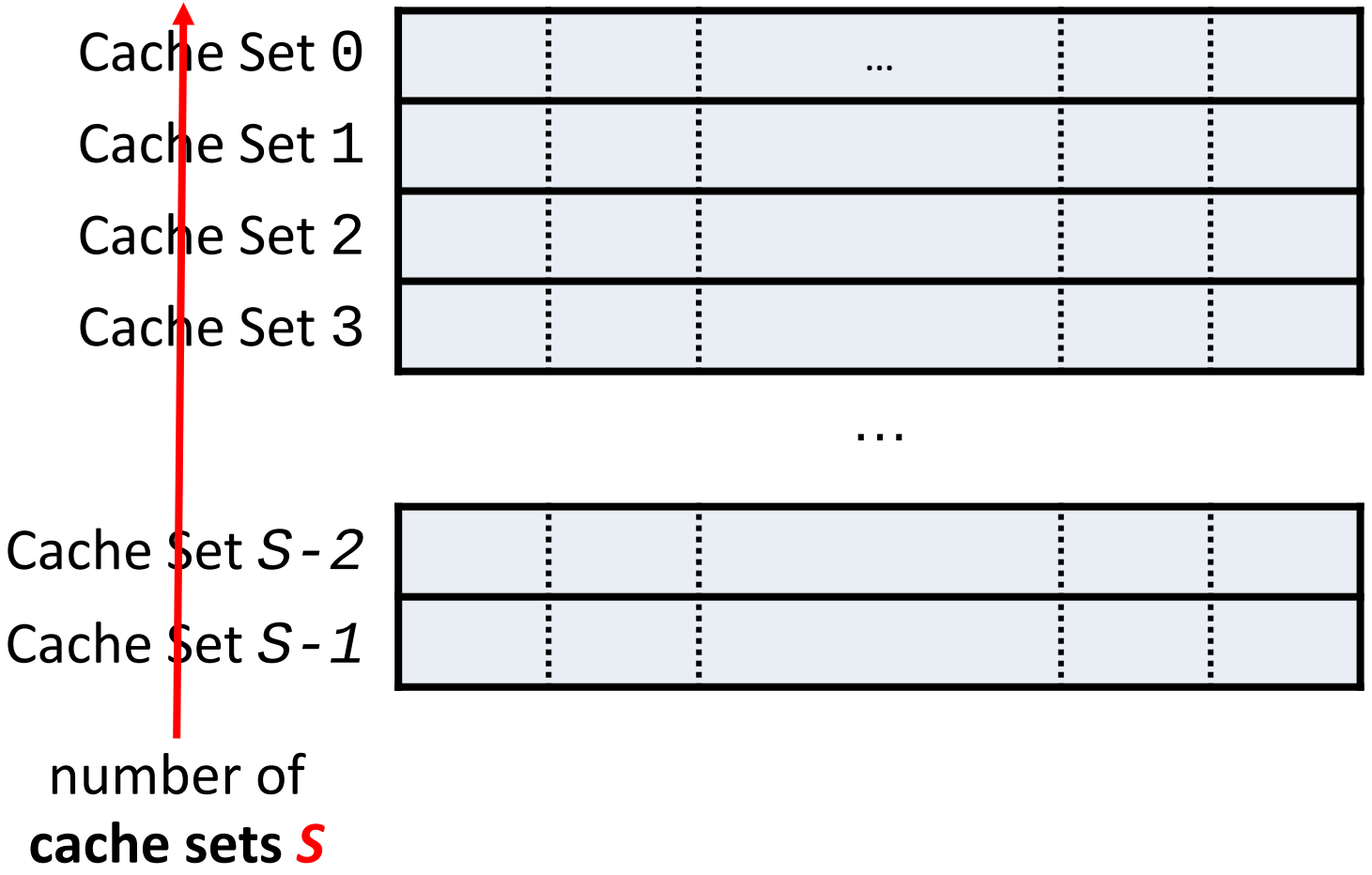
fully associative

E=8, *S*=1

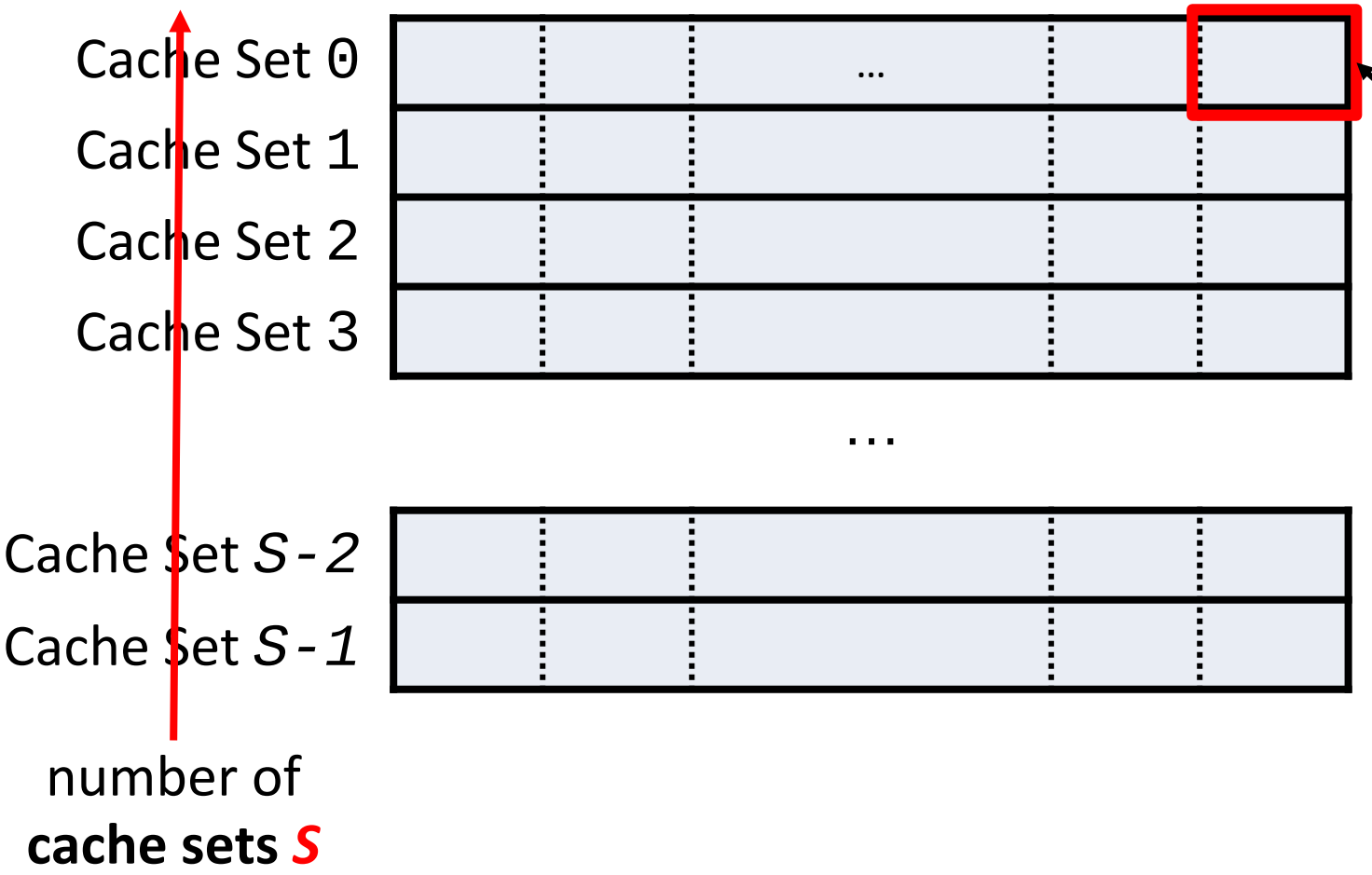
Cache Organization



Cache Organization



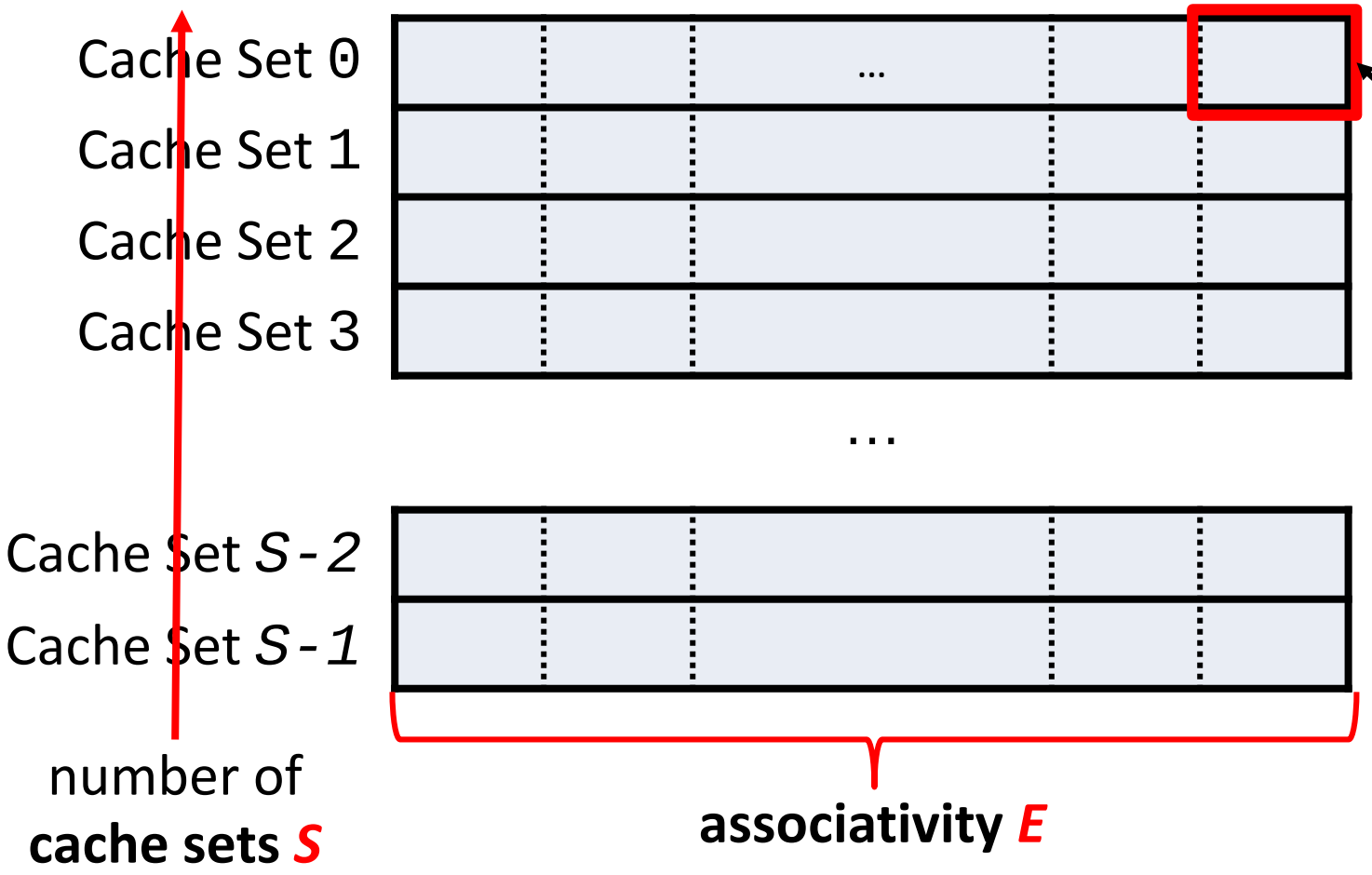
Cache Organization



Cache block with **block size** ***B*** indicates how many contiguous bytes of memory are copied to cache

Cache Organization

Cache Organization



The **capacity** of the cache is $C = E * B * S$

Question

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

Question

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

$C = 512$

$E = 1$

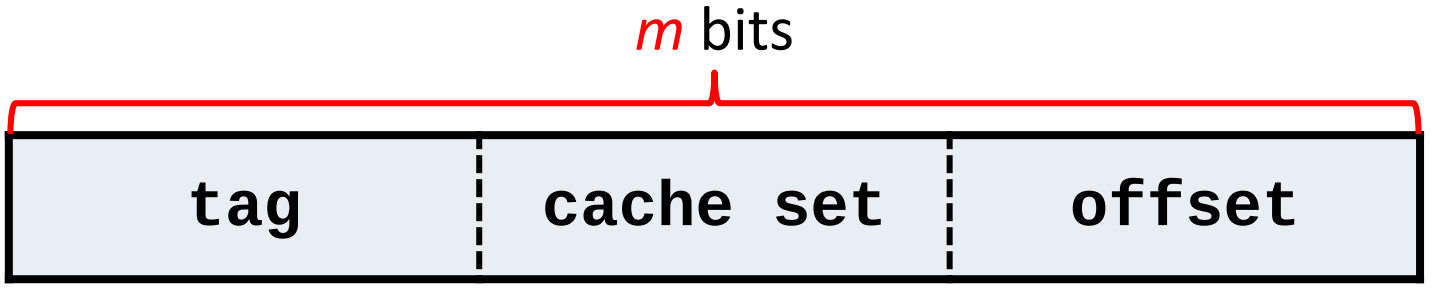
$B = 16$

$S = C / (E * B) = 32$

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

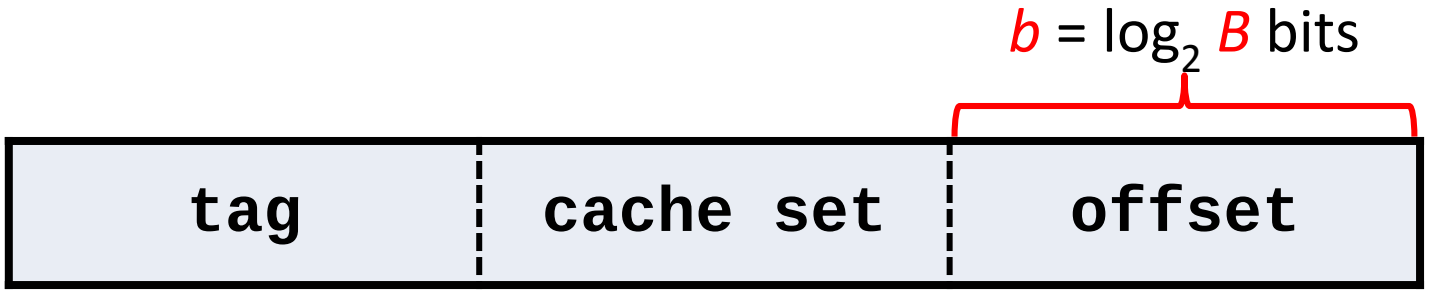
Mapping an Address to Cache

- A memory address of *m* bits is divided into 3 parts for mapping to cache



Mapping an Address to Cache

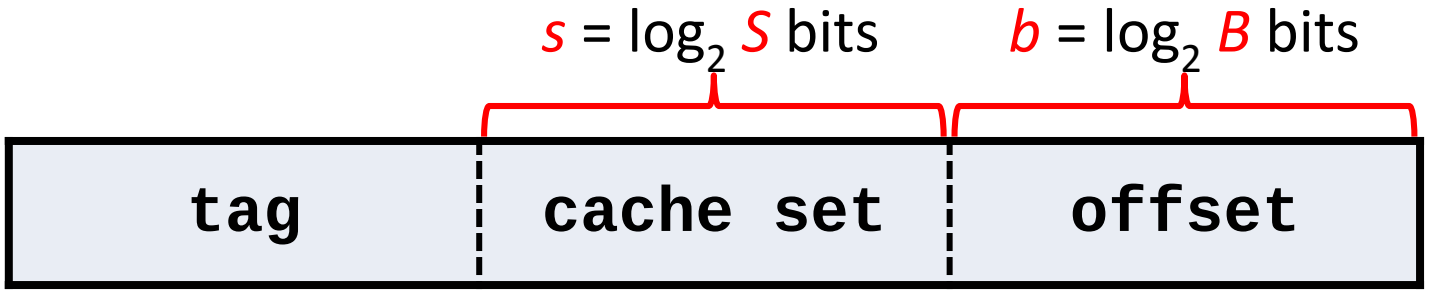
- A memory address of *m* bits is divided into 3 parts for mapping to cache



- ***Offset bits*** define the address offset into the cache block
- For example:
  - *B* = 4 bytes (*b*=2)
  - Address `0x0` (binary `0000`) and `0x1` (binary `0001`) are both part of the same cache block with different offsets

Mapping an Address to Cache

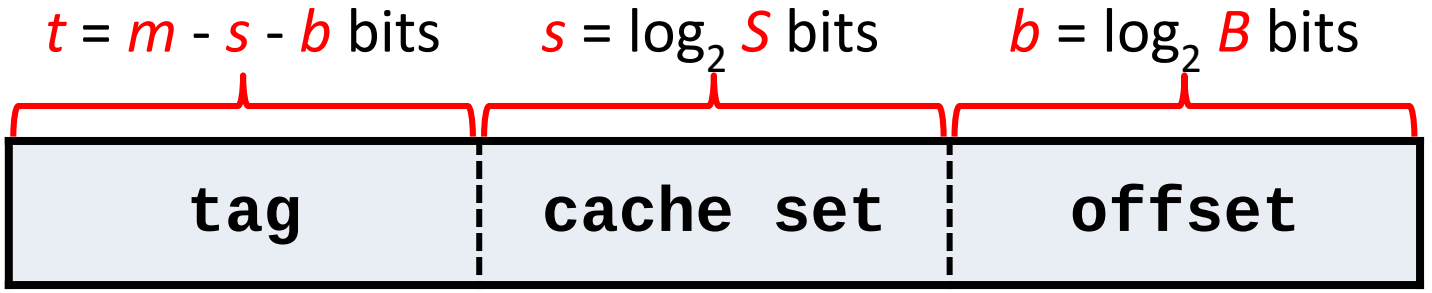
- A memory address of *m* bits is divided into 3 parts for mapping to cache



- ***Cache set bits*** tells us which set a block of data belongs to
- For example:
  - *B* = 4 bytes (*b*=2), S = 2 (*s*=1)
  - Address `0x0` (binary `0000`) and `0x4` (binary `0100`) belong to different cache sets

Mapping an Address to Cache

- A memory address of *m* bits is divided into 3 parts for mapping to cache

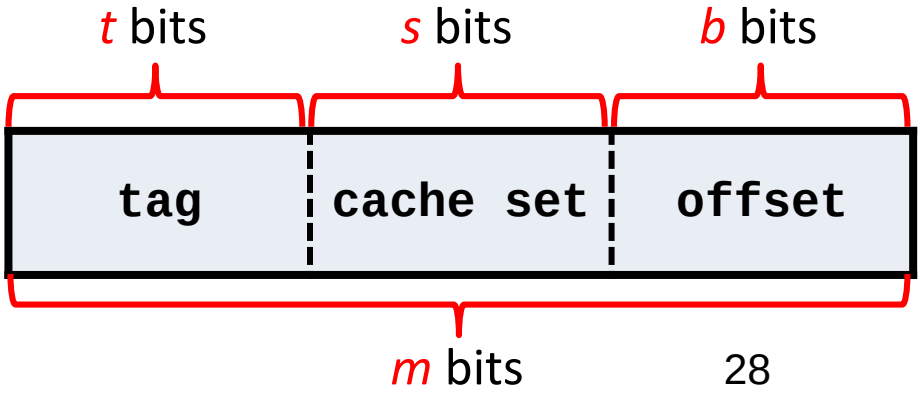


- ***Tag bits*** identify the block id that the address belongs to
- For example:
  - *B* = 4 bytes (*b*=2), S = 2 (*s*=1), *m* = 1 bit (*t*=1)
  - Address `0x0` (binary `0000`) and `0x8` (binary `1000`) belong to the same cache set (`0`), but have different block identifiers

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 ($\boldsymbol{log_2 S}$)
- ***b*** – number of block offset bits ($\boldsymbol{log_2 B}$)

- 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



Exercise: Cache Parameters

- **m** = 32, **C** = 1024, **B** = 4, **E** = 1
  - **S** = 256, **s** = 8, **b** = 2, **t** = 22
- **m** = 32, **C** = 1024, **B** = 8, **E** = 4
  - **S** = 32, **s** = 5, **b** = 3, **t** = 24
- **m** = 32, **C** = 1024, **B** = 32, **E** = 32
  - **S** = 1, **s** = 0, **b** = 5, **t** = 27

Question

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

A. 0x0836
B. 0x0845
C. 0x0877
D. 0x0900

Question

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

A. 0x0836
B. 0x0845
C. 0x0877
D. 0x0900

C = 512

E = 4

B = 16, ***b*** = 4

S = ***C*** / (***E*** * ***B***) = 8, ***s*** = 3

t = 16 - 4 - 3 = 9

Question

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

Address	Binary	Tag (9 bits)	Set (3 bits)	Offset (4 bits)
0x0836	1000 0011 0110	0x10	**0x3**	0x6
0x0845	1000 0100 0101	0x10	**0x4**	0x5
0x0877	1000 0111 0111	0x10	**0x7**	0x7
0x0900	1001 0000 0000	0x12	**0x0**	0x0

Mapping Memory to Cache

- A cache contains ***C***/***B*** total *cache lines*, and each line may be empty or may by occupied by a memory block.
- Each cache line has an additional ***valid*** bit associated with it
- No two cache lines may contain the same memory block. Why?
  - All blocks belong to only 1 set
  - The *tag* disambiguates memory blocks within the same cache set.

More on Cache-Misses

- Cache misses can actually be categorized into different types
  - Type helps to determine how to remove them

Cache-Miss Categories

- A ***compulsory* (or *cold*) *miss*** occurs on the very first access to a memory block.
  - Why?
  - Possible to avoid?

Cache-Miss Categories

- A ***capacity miss*** occurs when accessing a block that previously resided in cache, but was evicted because the cache cannot hold all of the data needed to execute a program.
  - Possible to avoid?

Cache-Miss Categories

- 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 is also a miss in an LRU fully-associative cache with capacity ***C***.
- A conflict miss in an LRU set-associative cache 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?*

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;
}
```

- ***E***=1, ***B***=16, ***C***=32, ***S***=2
- `sizeof(float)` is 4
- `x` starts at address 0
- `y` starts at address 32

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;
}
```

Loop Iteration	Element	Address	Set	Miss?	Element	Address	Set	Miss?
1	x[0]	0	0	cold	y[0]	32	0	cold
2	x[1]	4	0	conflict	y[1]	36	0	conflict
3	x[2]	8	0	conflict	y[2]	40	0	conflict
4	x[3]	12	0	conflict	y[3]	44	0	conflict
5	x[4]	16	1	cold	y[4]	48	1	cold
6	x[5]	20	1	conflict	y[5]	52	1	conflict
7	x[6]	24	1	conflict	y[6]	56	1	conflict
8	x[7]	28	1	conflict	y[7]	60	1	conflict

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 when the cache is repeatedly loading and evicting the same sets of blocks.

How can thrashing be prevented?

Loop Iteration	Element	Address	Set	Miss?	Element	Address	Set	Miss?
1	x[0]	0	0	cold	y[0]	32	0	cold
2	x[1]	4	0	conflict	y[1]	36	0	conflict
3	x[2]	8	0	conflict	y[2]	40	0	conflict
4	x[3]	12	0	conflict	y[3]	44	0	conflict
5	x[4]	16	1	cold	y[4]	48	1	cold
6	x[5]	20	1	conflict	y[5]	52	1	conflict
7	x[6]	24	1	conflict	y[6]	56	1	conflict
8	x[7]	28	1	conflict	y[7]	60	1	conflict

Exercise: Associative Cache

- ***E***=2, ***B***=4, ***C***=64, ***S***=8, ***m***=13
- Accesses are to 1-byte words
- Which bits (12 to 0) are used to determine:
  - cache block offset?
  - cache set index?
  - cache tag?
- Hit or miss (type)?
  - Reference to `0x0E34` (`111000110100`)?
  - Reference to `0x0DD5` (`110111010101`)?
  - Reference to `0x1FE4` (`1111111100100`)?

set	tag	valid	byte0	byte1	byte2	byte3	tag	valid	byte0	byte1	byte2	byte3
0	09	1	86	30	3F	10	00	0	--	--	--	--
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	--	--	--	--	0B	0	--	--	--	--
3	06	0	--	--	--	--	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	--	--	--	--
6	45	1	A0	B7	26	2D	F0	0	--	--	--	--
7	46	0	--	--	--	--	DE	1	12	C0	88	37

Exercise: Associative Cache

- ***E***=2, ***B***=4, ***C***=64, ***S***=8, ***m***=13
- Accesses are to 1-byte words
- Which bits (12 to 0) are used to determine:
  - cache block offset? (1 to 0)
  - cache set index? (4 to 2)
  - cache tag? (12 to 5)

- Hit or miss (type)?
  - Reference to `0x0E34` `1110001 101 00`
    Set = 5, Tag = 0x71
  - Reference to `0x0DD5` `1101110 101 01`
    Set = 5, Tag = 0x63
  - Reference to `0x1FE4` `11111111 001 00`
    Set = 1, Tag = 0xff

set	tag	valid	byte0	byte1	byte2	byte3	tag	valid	byte0	byte1	byte2	byte3
0	09	1	86	30	3F	10	00	0	--	--	--	--
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	--	--	--	--	0B	0	--	--	--	--
3	06	0	--	--	--	--	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	--	--	--	--
6	45	1	A0	B7	26	2D	F0	0	--	--	--	--
7	46	0	--	--	--	--	DE	1	12	C0	88	37