Memory hierarchy

Memory

- Operating system and CPU memory management unit gives each process the “illusion” of a uniform, dedicated memory space
  - i.e. 0x0 – 0xFFFFFFFF for IA32
  - Allows multitasking
  - Hides underlying non-uniform memory hierarchy

Random-Access Memory (RAM)

Non-persistent program code and data stored in two types of memory

Static RAM (SRAM)

- Each cell stores bit with a six-transistor circuit.
- Retains value as long as it is kept powered.
- Faster and more expensive than DRAM.

Dynamic RAM (DRAM)

- Each cell stores bit with a capacitor and transistor.
- Value must be refreshed every 10-100 ms.
- Slower and cheaper than SRAM.

SRAM vs DRAM Summary

	Tran. per bit	Access time	Persist?	Sensitive?	Cost	Applications
SRAM	6	1X	Yes	No	100x	cache memories
DRAM	1	10X	No	Yes	1X	Main memories, frame buffers

Memory speed over time

In 1985

- A memory access took about as long as a CPU instruction
- Memory was not a bottleneck to performance

Today

- CPUs are about 500 times faster than in 1980
- DRAM Memory is about 10 times faster than in 1980

CPU-Memory gap

SRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
access (ns)	150	35	15	3	2	1.5	*200*	*115*

DRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
access (ns)	200	100	70	60	50	40	20	*10*
typical size (MB)	0.256	4	16	64	2,000	8,000	16.000	*62,500*

CPU

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
CPU	80286	80386	Pentium	P-4	Core 2	Core i7(n)	Core i7(h)	
Clock rate (MHz)	6	20	150	3,300	2,000	2,500	3,000	500
Cycle time (ns)	166	50	6	0.30	0.50	0.4	0.33	500
Cores	1	1	1	1	2	4	4	4
Effective cycle time (ns)	166	50	6	0.30	0.25	0.10	0.08	2,075

Inflection point in computer history when designers hit the "Power Wall"

(n) Nehalem processor
(h) Haswell processor

Addressing CPU-Memory Gap

Observation

- Fast memory can keep up with CPU speeds, but cost more per byte and have less capacity
- Gap is widening

Suggest an approach for organizing memory and storage systems known as a memory hierarchy.

- For each level of the hierarchy k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

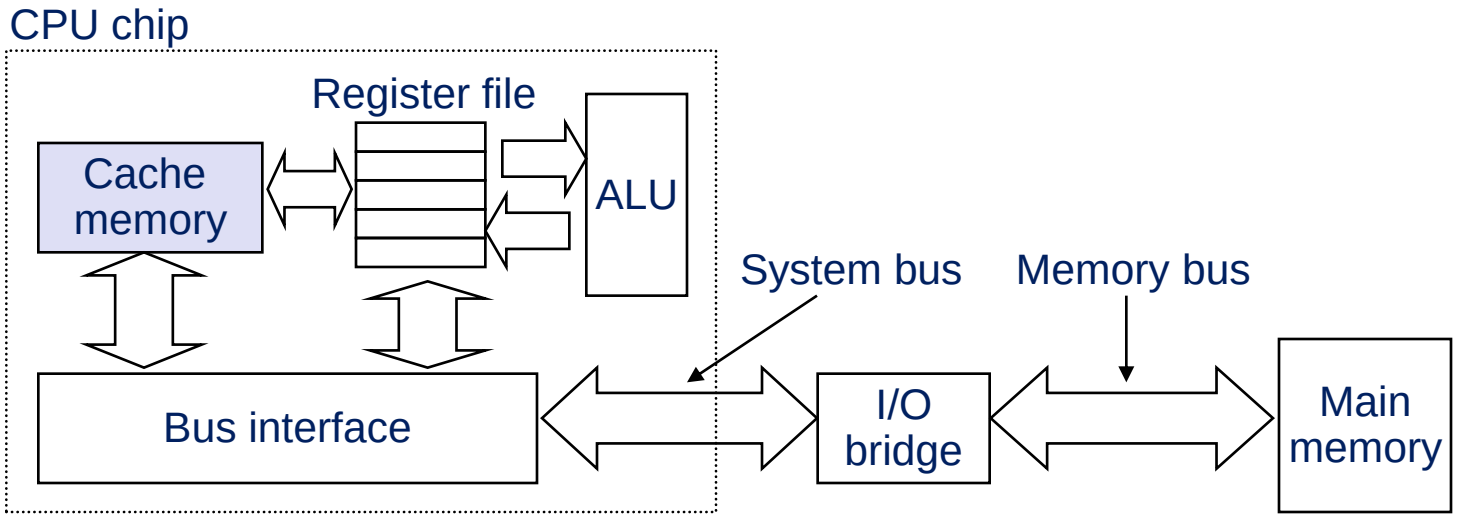
Key is for computer programs to exhibit locality

- Keep data needed in the near future in fast memory near the CPU
- Design programs to ensure data at k is accessed much more often than data at k+1.

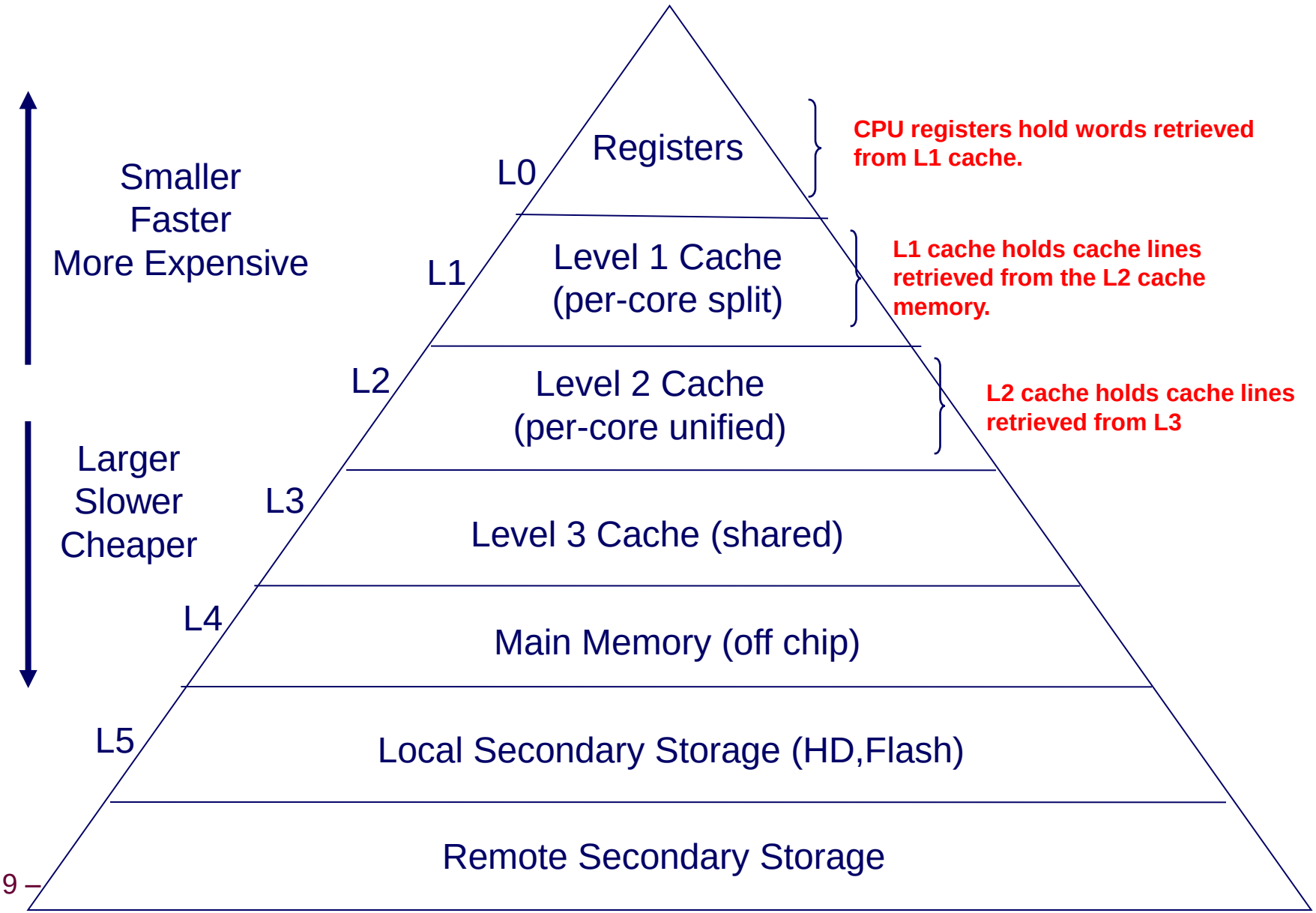
Cache Memories

A smaller, faster , SRAM-based memories managed automatically in hardware.

- **Hold frequently accessed blocks of main memory**
- **CPU looks first for data in L1, then in L2, then in main memory.**
- **Typical system structure:**



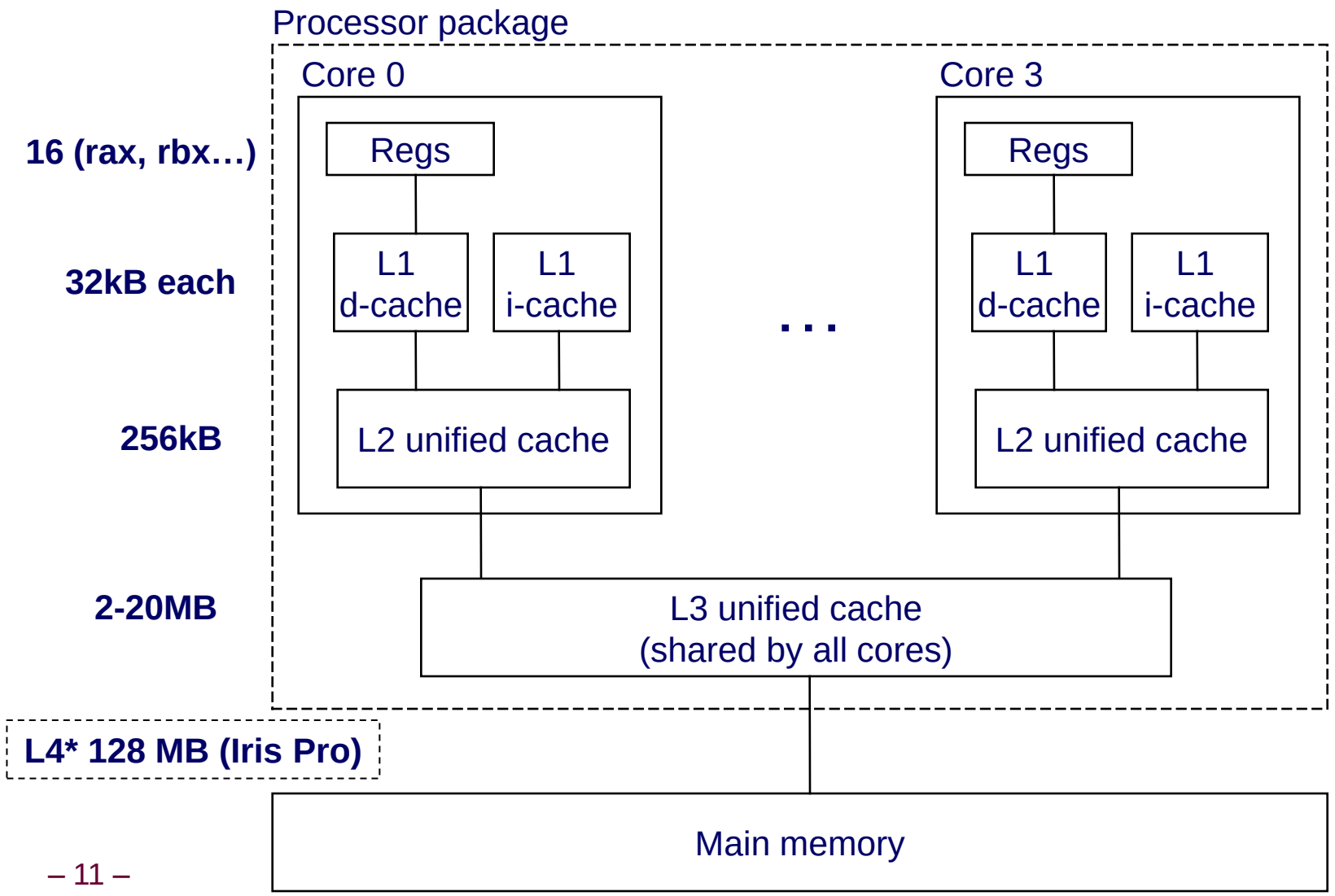
Example memory hierarchy



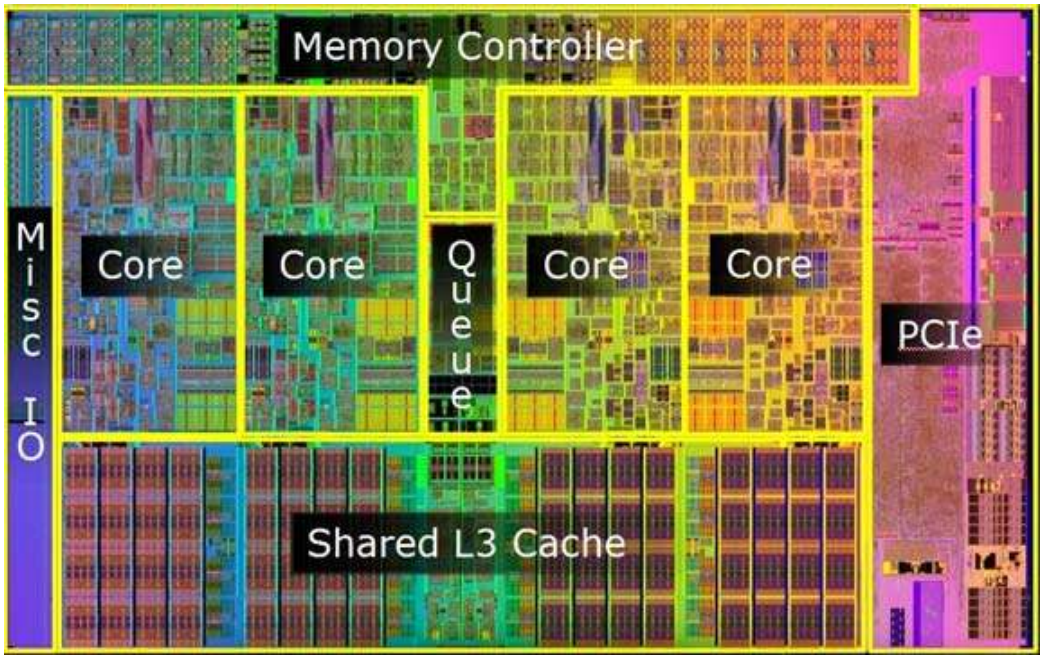
Examples of Caching in the Hierarchy

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+ OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Intel Core i7 cache hierarchy (2014)



Intel Core i7 cache hierarchy



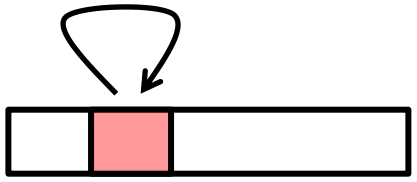
Hierarchy requires Locality

Principle of Locality:

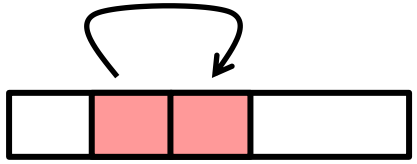
- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.

Kinds of Locality:

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



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



Locality in code

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

Give an example of spatial locality for data access

- Reference array elements in succession (stride-1 pattern)

Give an example of temporal locatity for data access

- Reference `sum` each iteration

Give an example of spatial locality for instruction access

- Reference instructions in sequence

Give an example of temporal locality for instruciton access

- Cycle through loop repeadedly

Locality example

Which function has better locality?

```
int sumarrayrows(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
}
```

```
int sumarraycols(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
}
```

Practice problem 6.7

Permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(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
}
```

Practice problem 6.7

Permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

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

Practice problem 6.8

Rank-order the functions with respect to the spatial locality enjoyed by each

```
#define N 1000
typedef struct {
   int vel[3];
   int acc[3];
} point;
point p[N];
```

Best

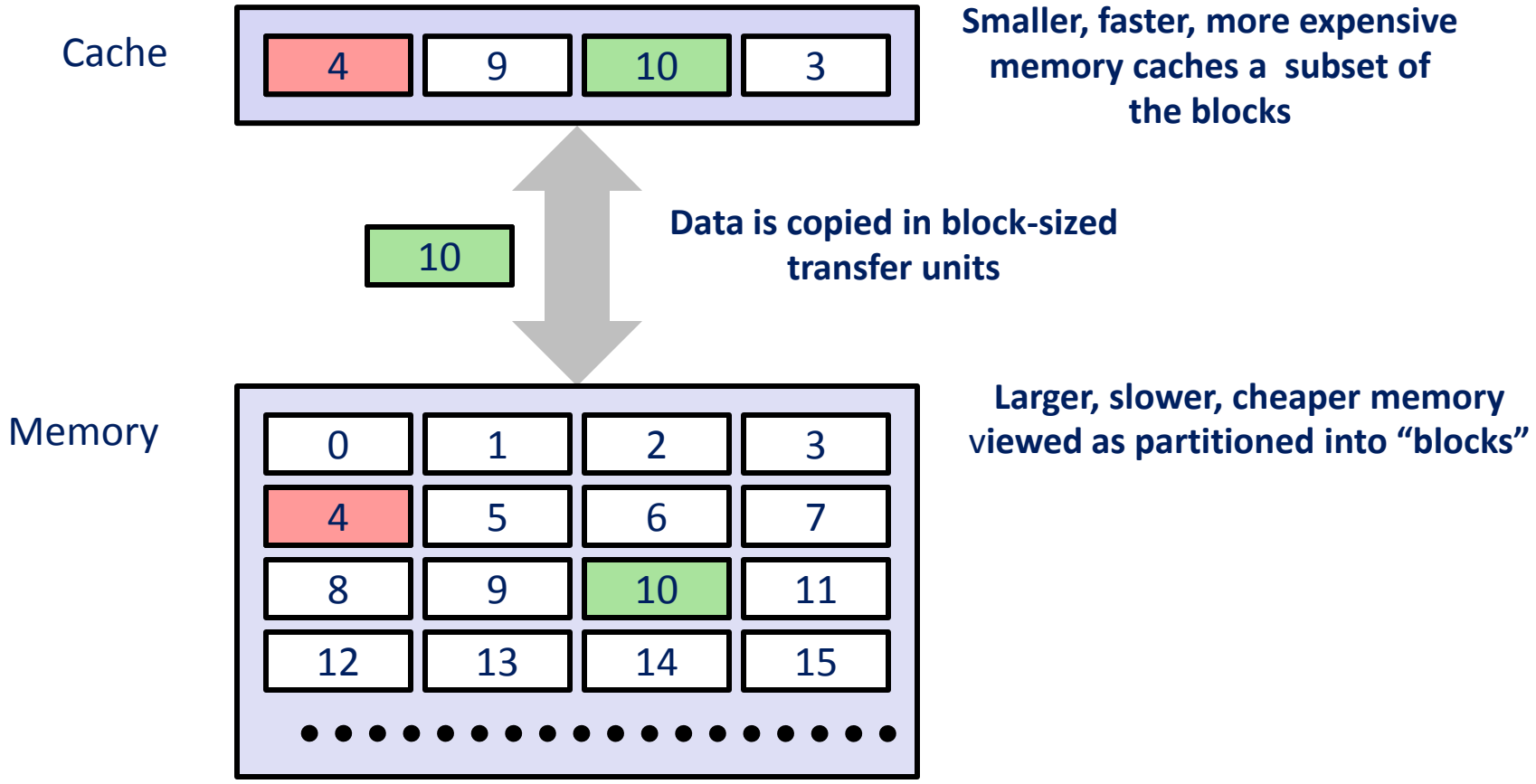
```
void clear1(point *p, int n) {
   int i,j;
   for (i=0; i<n ; i++) {
      for (j=0;j<3;j++)
         p[i].vel[j]=0;
      for (j=0; j<3 ; j++)
         p[i].acc[j]=0;
   }
```

```
void clear2(point *p, int n) {
   int i,j;
   for (i=0; i<n ; i++) {
      for (j=0; j<3 ; j++) {
         p[i].vel[j]=0;
         p[i].acc[j]=0;
      }
   }
```

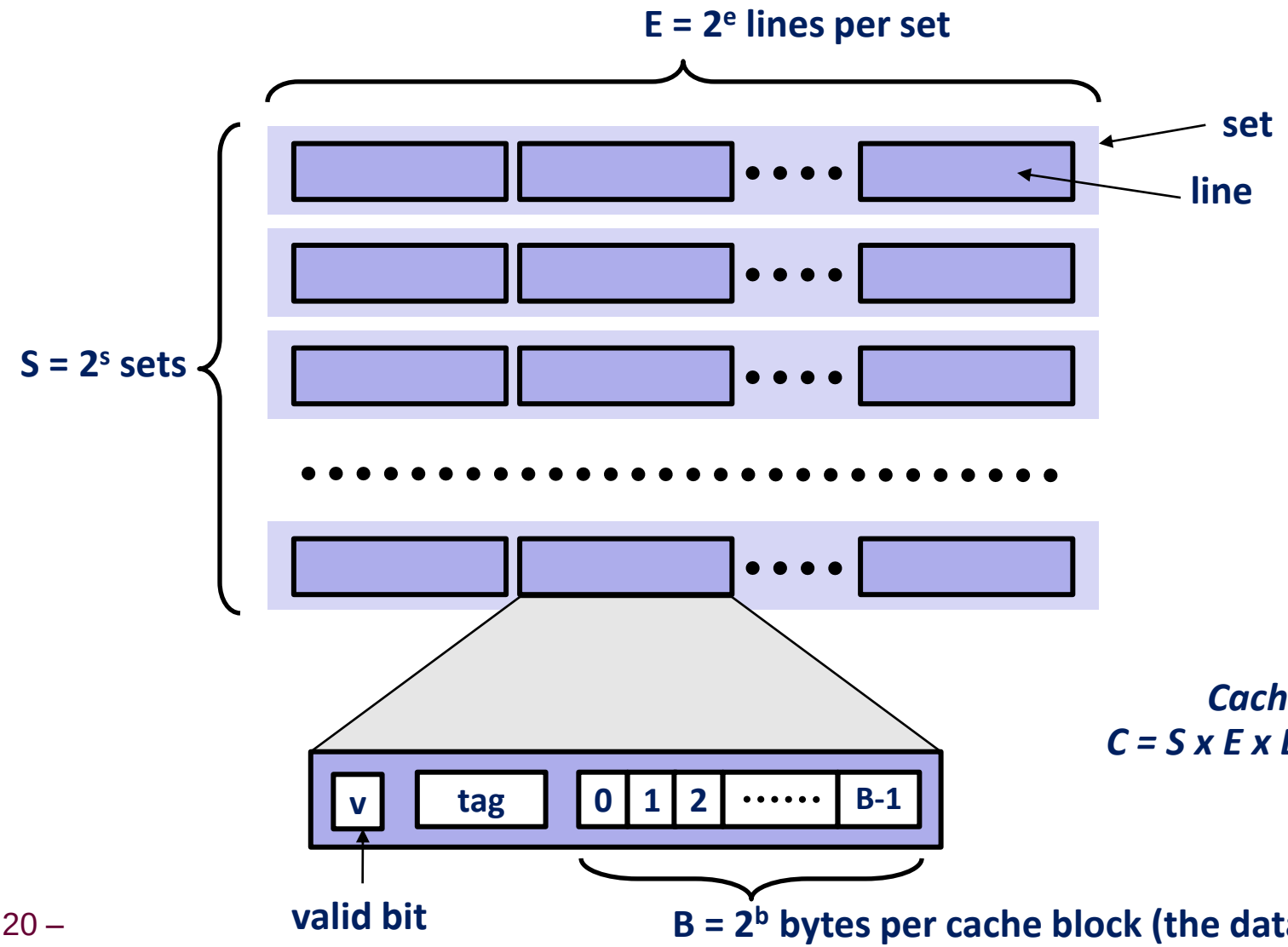
Worst

```
void clear3(point *p, int n) {
   int i,j;
   for (j=0; j<3 ; j++) {
      for (i=0; i<n ; i++)
         p[i].vel[j]=0;
      for (i=0; i<n ; i++)
         p[i].acc[j]=0;
   }
```

General Cache Operation



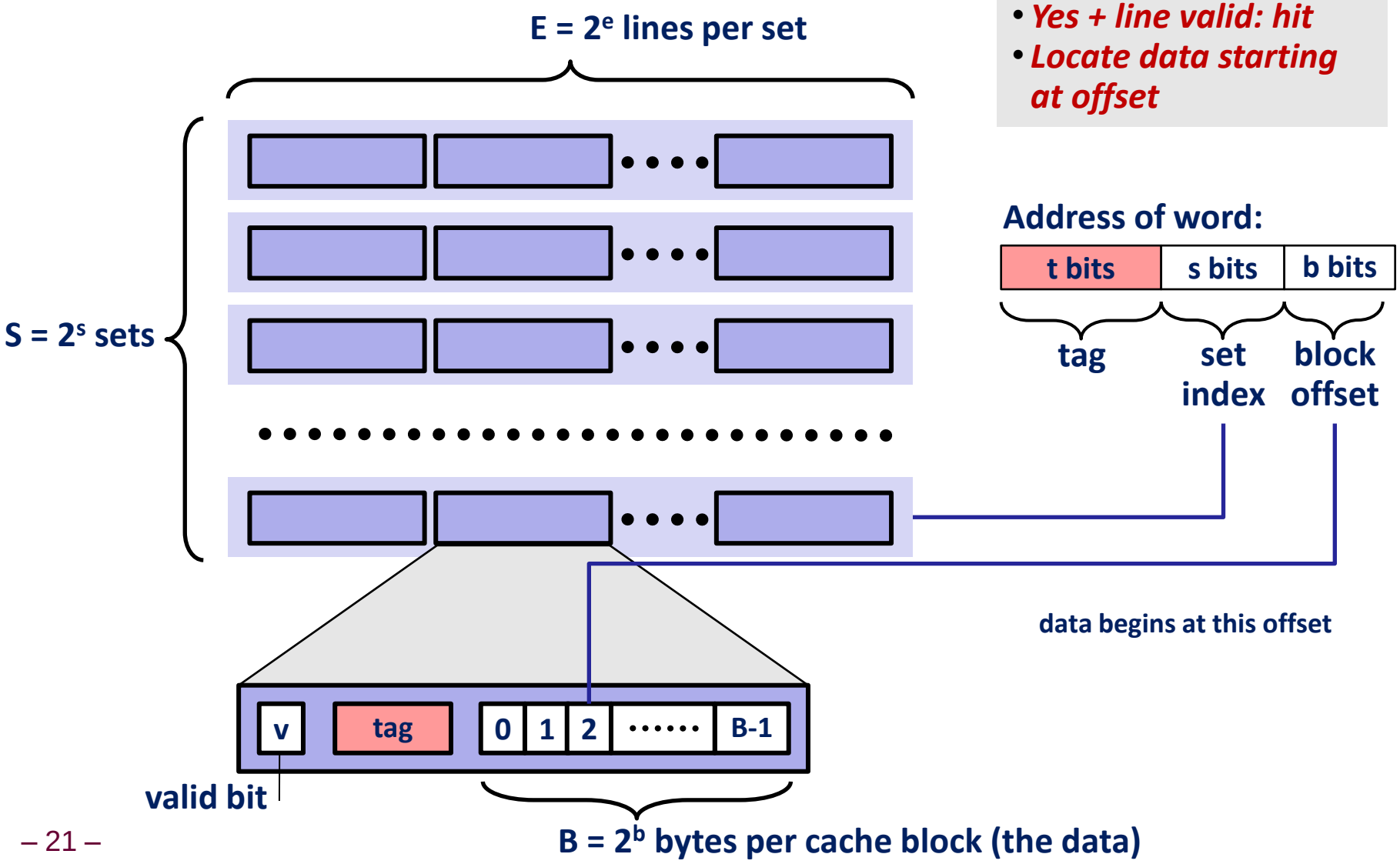
General Cache Organization (S, E, B)



Cache size:
C = S x E x B data bytes

Cache Read

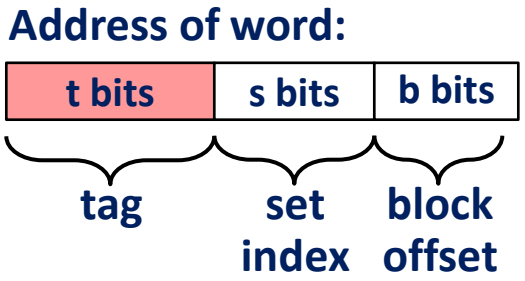
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



General Cache Organization

S sets

- s = # address bits used to select set
- $s = \log_2 S$



B blocks per line

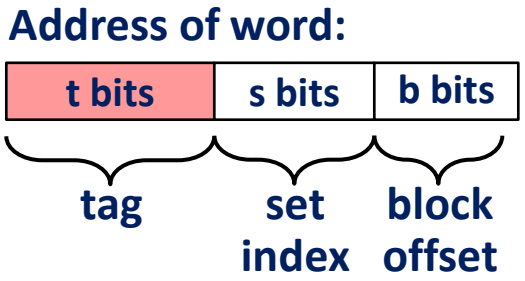
- b = # address bits used to select byte in line
- $b = \log_2 B$

How big is the tag?

General Cache Organization

M = Total size of main memory

- 2^m addresses
- m = # address bits = $\log_2 M$
- x86-64
  - m = 64
  - M = 2^{64}



Tag consists of bits not used in set index and block offset

- Tag must uniquely identify data

```
m = t + s + b
t = m - s - b
```

Practice problem 6.9

The table gives the parameters for a number of different caches. For each cache, derive the missing values

Cache size = C = S * E * B

# of tag bits = m - (s+b)

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

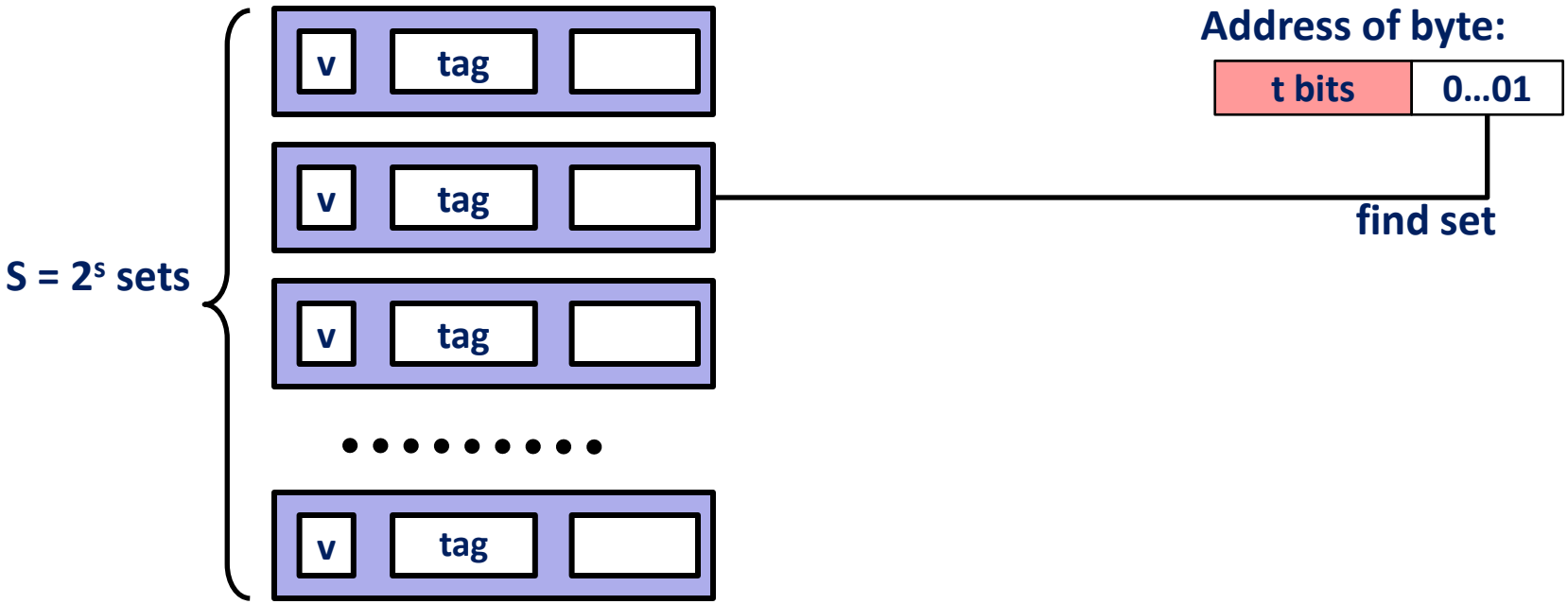
Direct mapped caches

Simplest of caches to implement

- One line per set (E = 1)
- Each location in memory maps to a single cache entry
- If multiple locations map to the same entry, their accesses will cause the cache to “thrash” from conflict misses
- Recall = (S,E,B,m)
  - C = S*E*B = S*B for direct-mapped cache

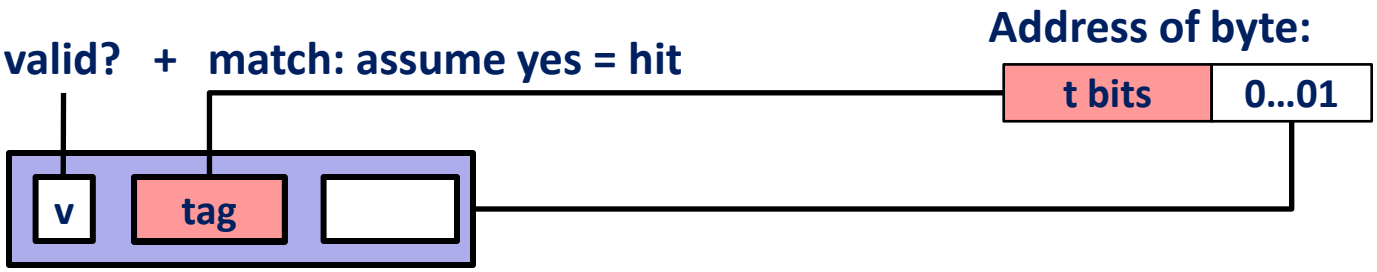
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 1 byte



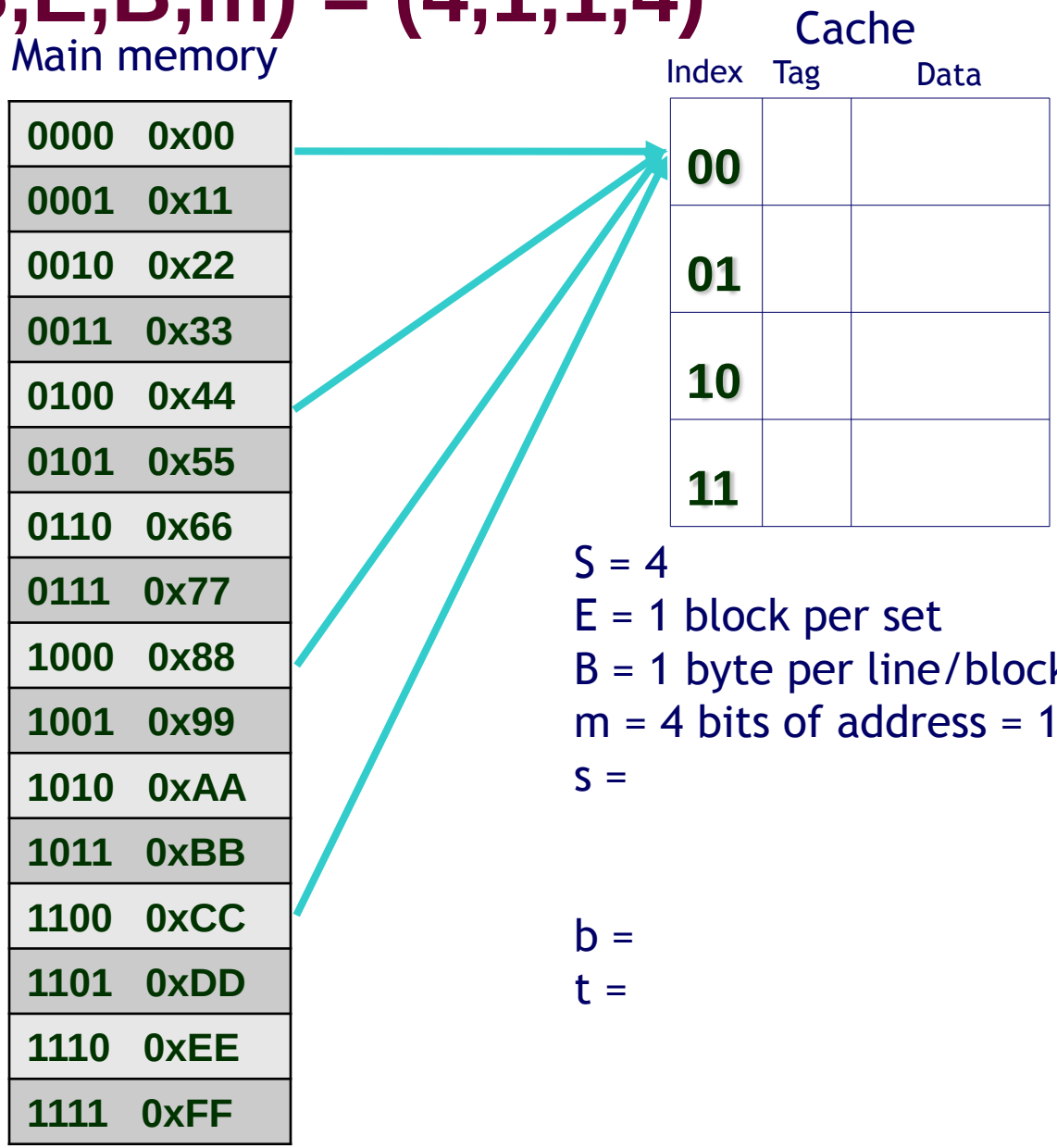
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 1 byte



If tag doesn't match: old line is evicted and replaced

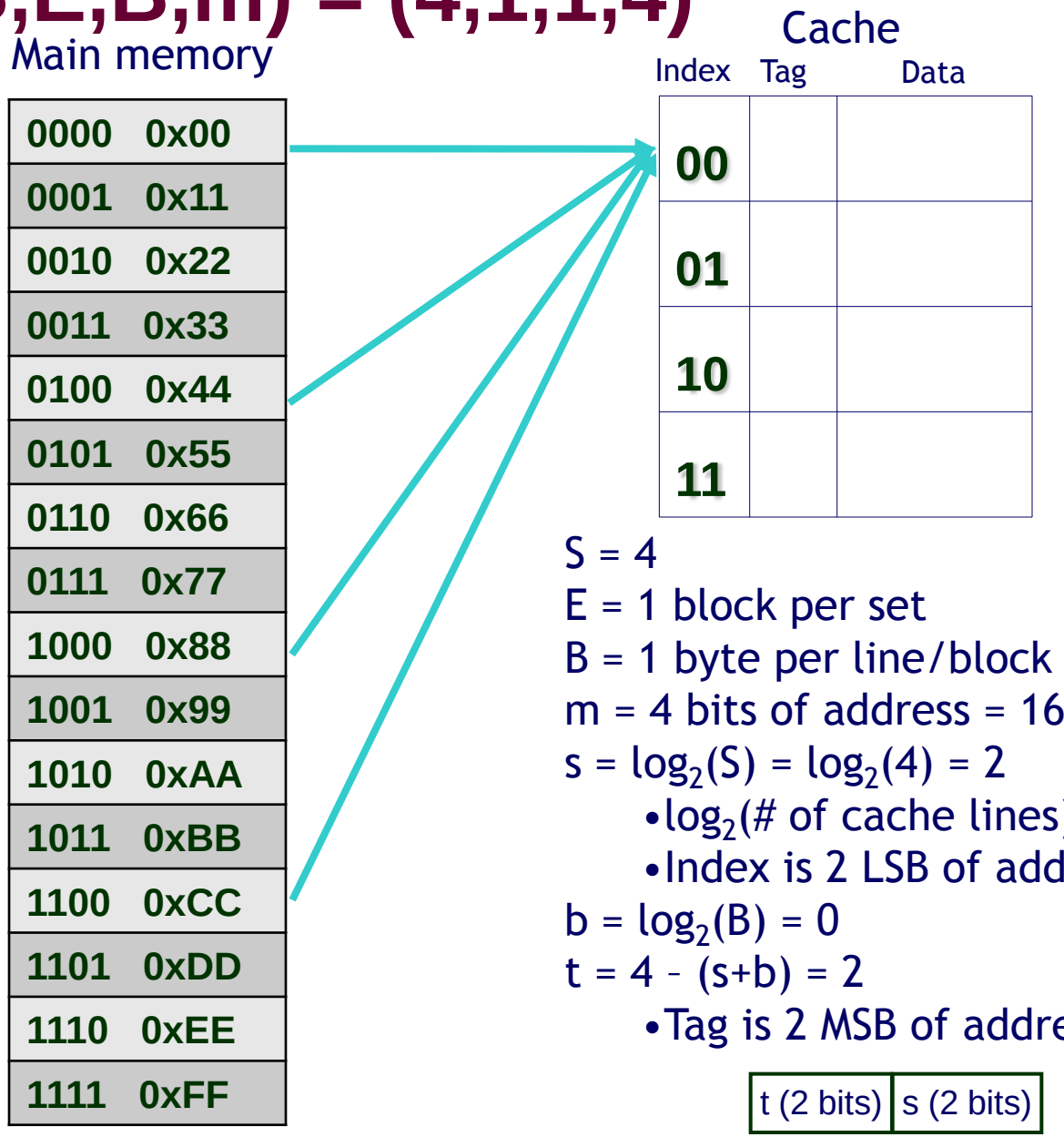
Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)



S = 4
E = 1 block per set
B = 1 byte per line/block
m = 4 bits of address = 16 bytes of memory
s =

b =
t =

Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)



S = 4

E = 1 block per set

B = 1 byte per line/block

m = 4 bits of address = 16 bytes of memory

$s = \log_2(S) = \log_2(4) = 2$

- $\log_2$(# of cache lines)
- Index is 2 LSB of address

$b = \log_2(B) = 0$

$t = 4 - (s+b) = 2$

- Tag is 2 MSB of address

t (2 bits)	s (2 bits)

Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

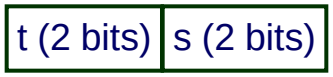
Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00		
01		
10		
11		

Access pattern:
0000
0011
1000
0011
1000



Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00	00	0x00
01		
10		
11		

Access pattern:
0000
0011
1000
0011
1000

Cache miss

t (2 bits)	s (2 bits)

Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

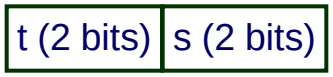
Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00	00	0x00
01		
10		
11	00	0x33

Access pattern:
0000
0011 Cache miss
1000
0011
1000



Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

Main memory

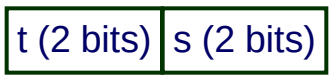
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00	00	0x00
01		
10		
11	00	0x33

Access pattern:
0000
0011
1000
0011
1000

Cache miss (Tag mismatch)



Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

Main memory

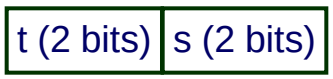
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00	10	0x88
01		
10		
11	00	0x33

Access pattern:
0000
0011
1000
0011
1000

Cache miss (Tag mismatch)



Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00	10	0x88
01		
10		
11	00	0x33

Access pattern:
0000
0011
1000
0011
1000

Cache hit

Simple direct-mapped cache example (S,E,B,m) = (4,1,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

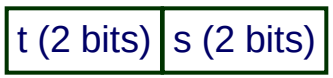
Cache

Index	Tag	Data
00	10	0x88
01		
10		
11	00	0x33

Access pattern:
0000
0011
1000
0011
1000

Cache hit

Miss rate = 3/5 = 60%



Issue #1: (S,E,B,m) = (4,1,1,4)
No spatial locality

Main memory

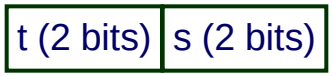
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00		
01		
10		
11		

Access pattern:
0000
0001
0010
0011
0100
0101
…

Miss rate = ?



Block size issues

Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache.

Increase hit rate for sequential access

Use least significant bits of address as block offset

- Sequential bytes in memory are stored together in a block (or cache line)

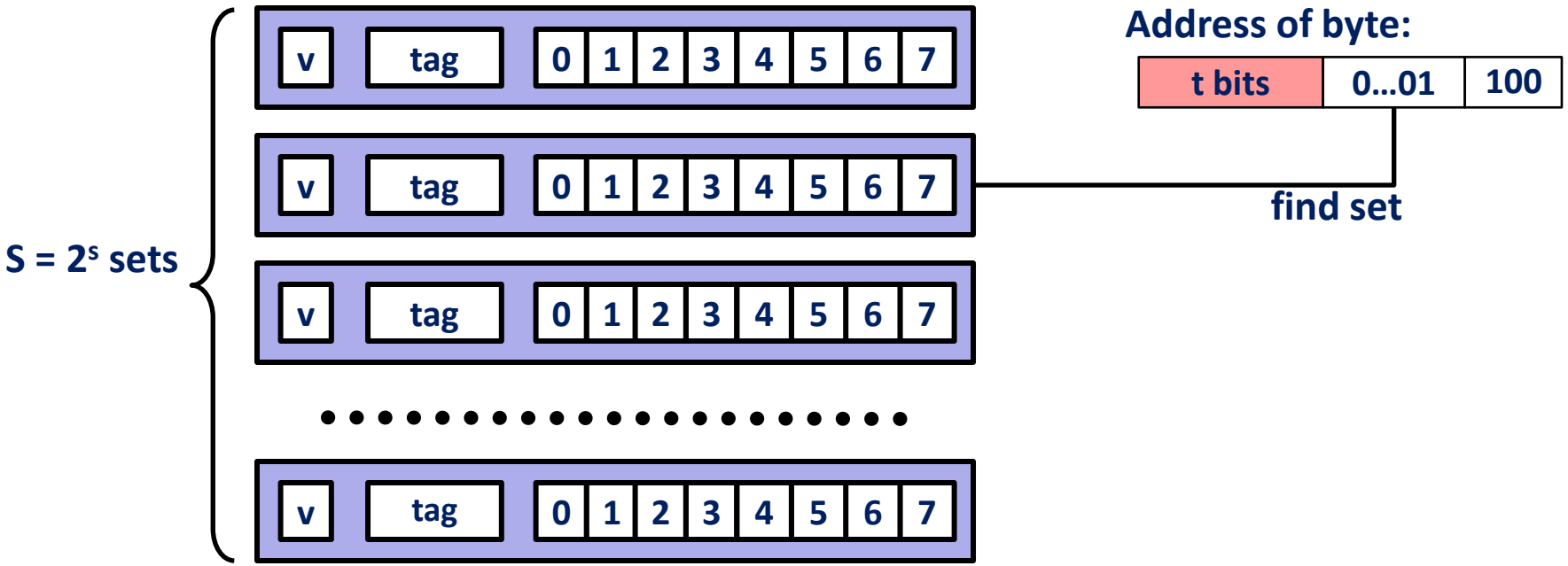
Use next significant bits (middle bits) to choose the set in the cache certain blocks map into.

- Number of bits = $\log_2$(# of sets in cache)

Use most significant bits as a tag to uniquely identify blocks in cache

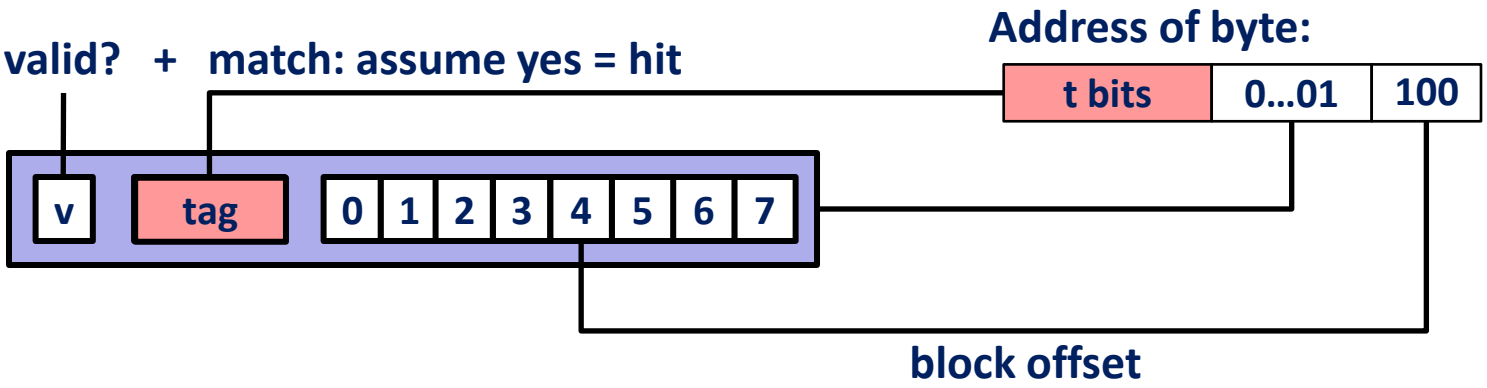
Example: Direct Mapped Cache (E = 1, B=8)

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



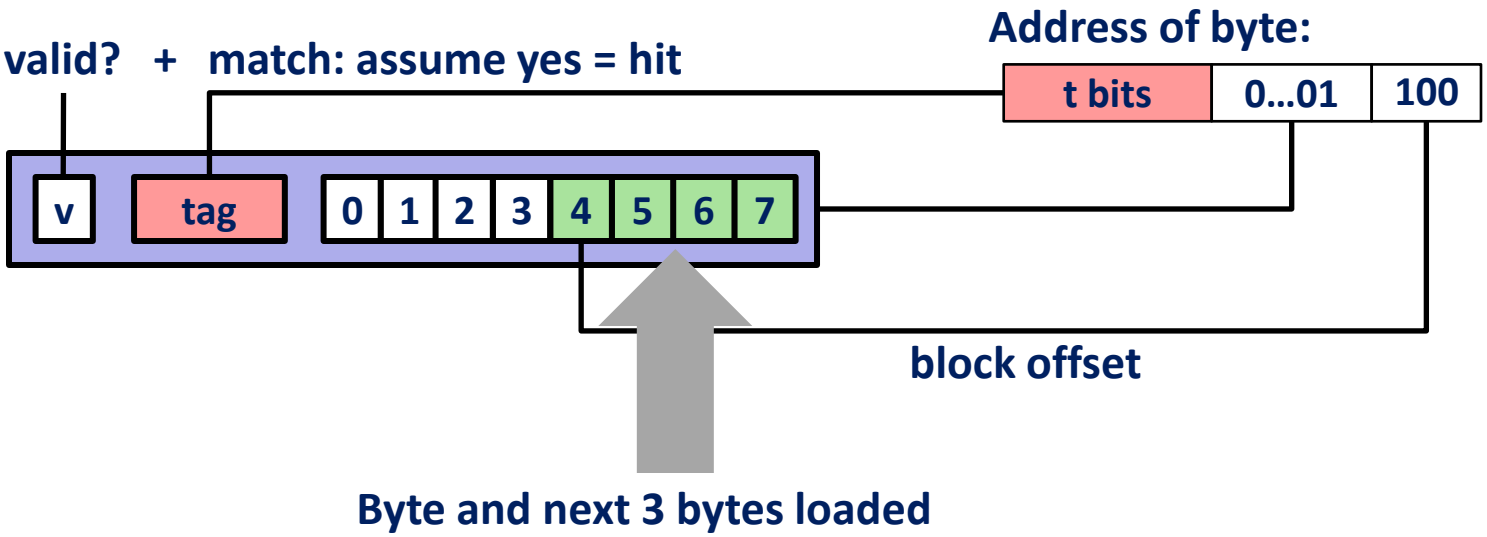
Example: Direct Mapped Cache (E = 1, B=8)

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



Example: Direct Mapped Cache (E = 1, B=8)

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



If tag doesn't match: old line is evicted and replaced

Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

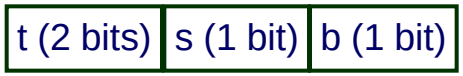
Cache

Index	Tag	Data1	Data0
0			
1			

Partition the address to identify which pieces are used for the tag, the index, and the block offset

b = 1
s = 1
t = 2

Tag = 2 MSB of address
Index = Bit 1 of address = $\log_2$(# of cache lines)
Block offset = LSB of address = $\log_2$(# blocks/line)



Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

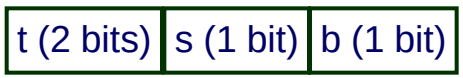
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data1	Data0
0	00	0x11	0x00
1			

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache miss



Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data1	Data0
0	00	0x11	0x00
1			

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache hit

t (2 bits)	s (1 bit)	b (1 bit)

Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

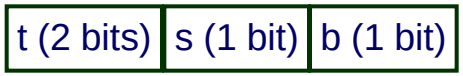
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data1	Data0
0	00	0x11	0x00
1	00	0x33	0x22

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache miss



Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

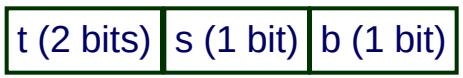
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data1	Data0
0	00	0x11	0x00
1	00	0x33	0x22

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache hit



Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

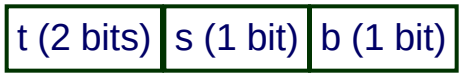
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data1	Data0
0	01	0x55	0x44
1	00	0x33	0x22

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache miss



Direct-mapped cache B=2 (S,E,B,m) = (2,1,2,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

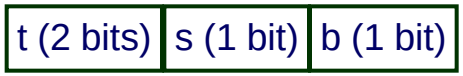
Cache

Index	Tag	Data1	Data0
0	01	0x55	0x44
1	00	0x33	0x22

Access pattern:
0000
0001
0010
0011
0100
0101
…

Cache hit

Miss rate = ?



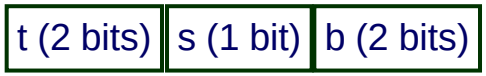
Exercise

Consider the following cache (S,E,B,m) = (2,1,4,5)

- **Derive values for number of address bits used for the tag (t), the index (s) and the block offset (b)**

  s = 1
  b = 2
  t = 2

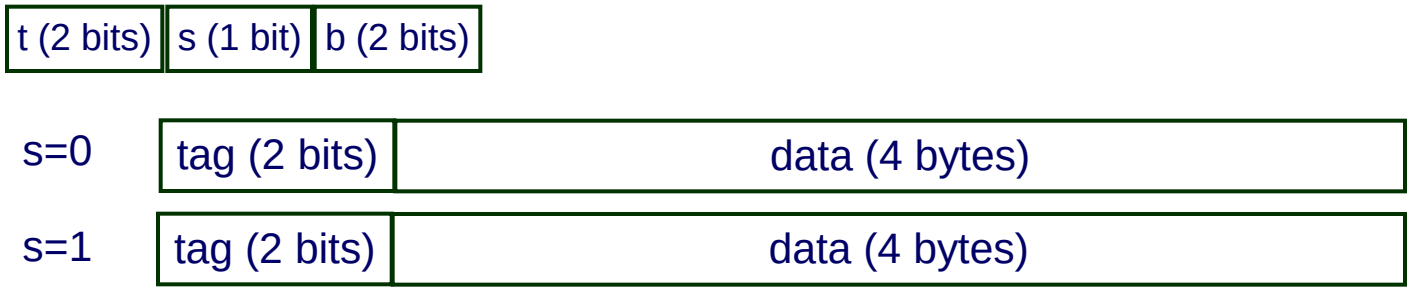
- **Draw a diagram of which bits of the address are used for the tag, the set index and the block offset**



- **Draw a diagram of the cache**

Exercise

Consider the same cache (S,E,B,m) = (2,1,4,5)

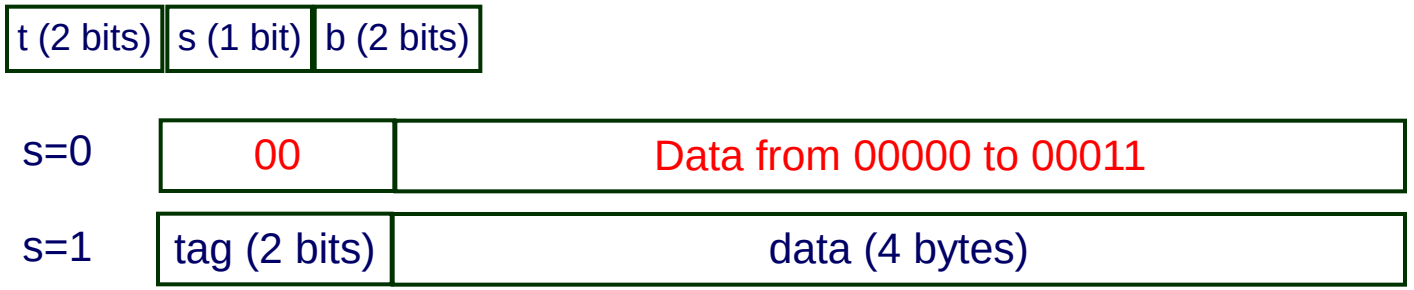


- **Derive the miss rate for the following access pattern**

00000
00001
00011
00100
00111
01000
01001
10000
10011
00000

Exercise

Consider the same cache (S,E,B,m) = (2,1,4,5)

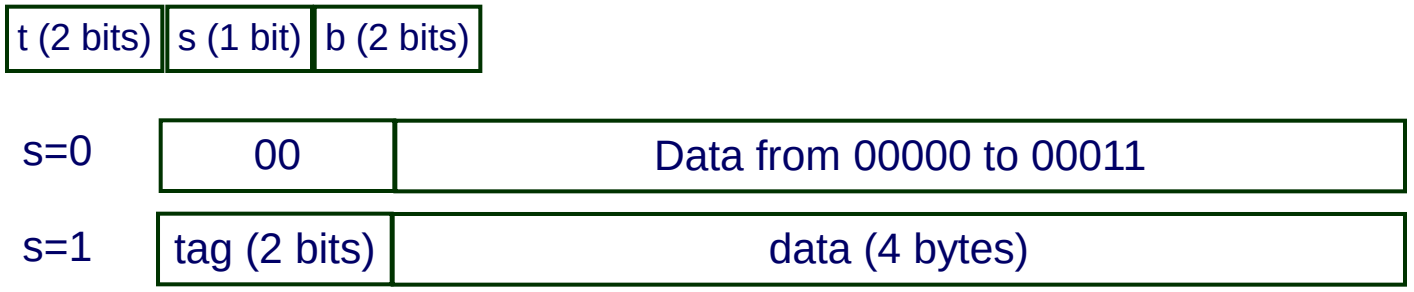


- **Derive the miss rate for the following access pattern**

  00000 Miss
  00001
  00011
  00100
  00111
  01000
  01001
  10000
  10011
  00000

Exercise

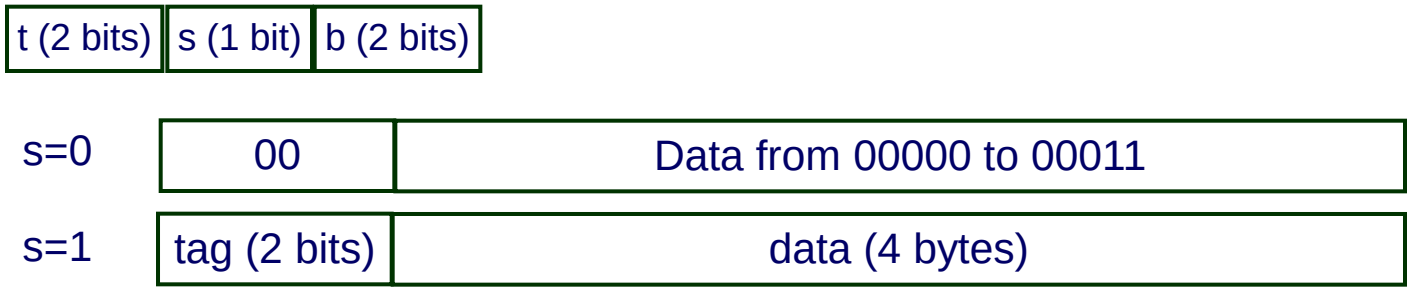
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011**
  - **00100**
  - **00111**
  - **01000**
  - **01001**
  - **10000**
  - **10011**
  - **00000**

Exercise

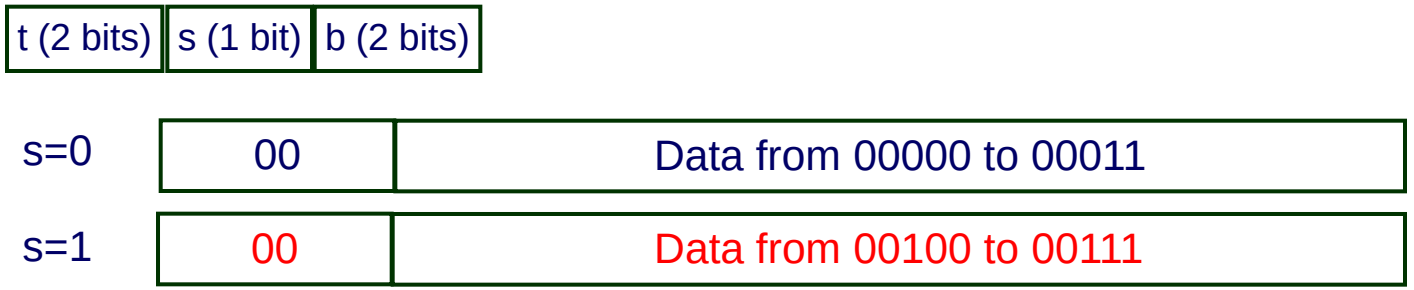
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100**
  - **00111**
  - **01000**
  - **01001**
  - **10000**
  - **10011**
  - **00000**

Exercise

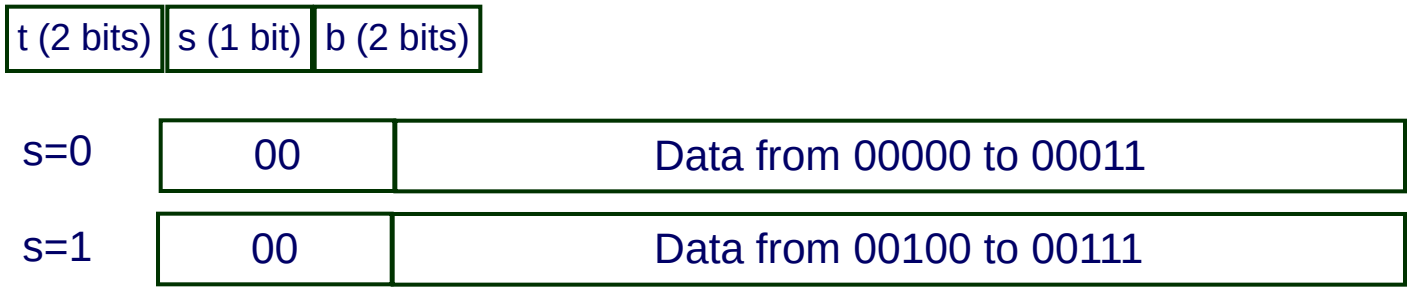
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111**
  - **01000**
  - **01001**
  - **10000**
  - **10011**
  - **00000**

Exercise

Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111 Hit**
  - **01000**
  - **01001**
  - **10000**
  - **10011**
  - **00000**

Exercise

Consider the same cache (S,E,B,m) = (2,1,4,5)

t (2 bits)	s (1 bit)	b (2 bits)

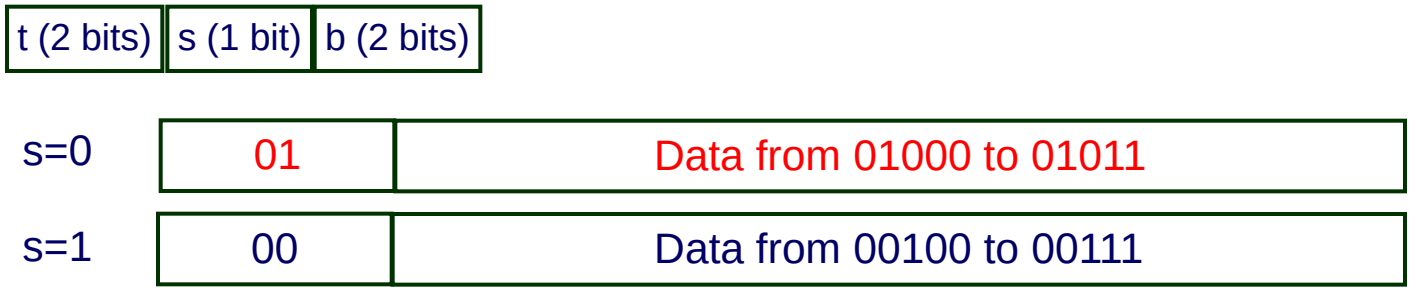
s=0	01	Data from 01000 to 01011
s=1	00	Data from 00100 to 00111

- **Derive the miss rate for the following access pattern**

00000 Miss
00001 Hit
00011 Hit
00100 Miss
00111 Hit
01000 Miss
01001
10000
10011
00000

Exercise

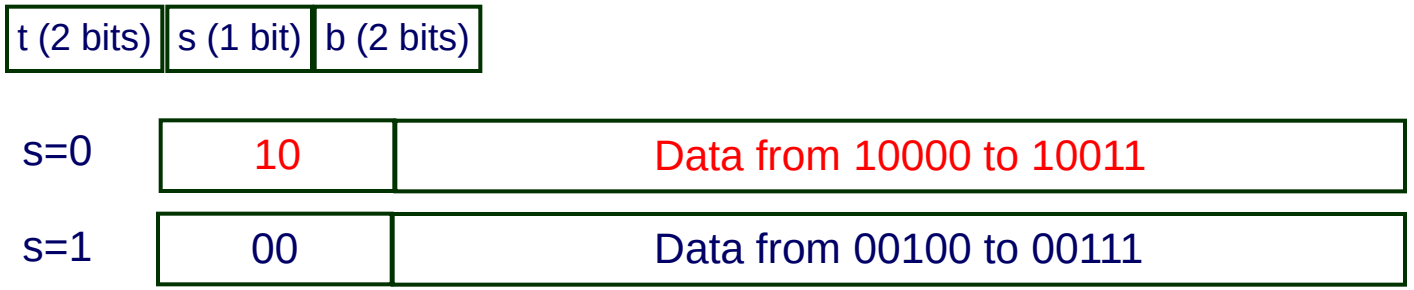
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111 Hit**
  - **01000 Miss**
  - **01001 Hit**
  - **10000**
  - **10011**
  - **00000**

Exercise

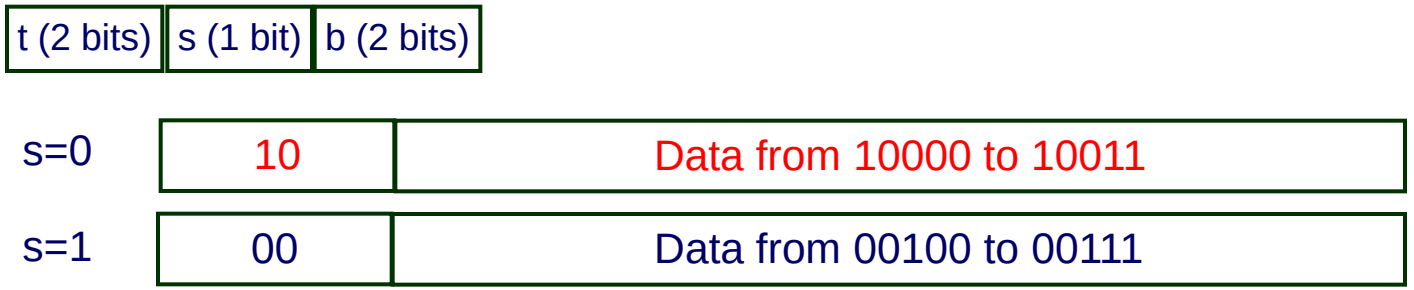
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111 Hit**
  - **01000 Miss**
  - **01001 Hit**
  - **10000 Miss**
  - **10011**
  - **00000**

Exercise

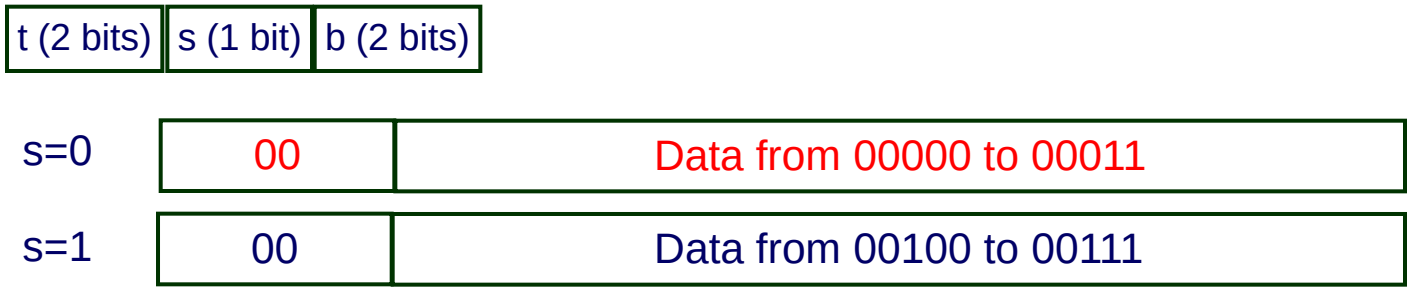
Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111 Hit**
  - **01000 Miss**
  - **01001 Hit**
  - **10000 Miss**
  - **10011 Hit**
  - **00000**

Exercise

Consider the same cache (S,E,B,m) = (2,1,4,5)



- **Derive the miss rate for the following access pattern**
  - **00000 Miss**
  - **00001 Hit**
  - **00011 Hit**
  - **00100 Miss**
  - **00111 Hit**
  - **01000 Miss**
  - **01001 Hit**
  - **10000 Miss**
  - **10011 Hit**
  - **00000 Miss**

Practice problem 6.11

Consider the following code that runs on a system with a cache of the form (S,E,B,m)=(512,1,32,32)

```
int array[4096];
for (i=0; i<4096; i++)
    sum += array[i];
```

Assuming sequential allocation of the integer array, what is the maximum number of integers from the array that are stored in the cache at any point in time?

B=32 (8 integers)
 # integers = 8*512 = 4096

Direct mapping issues

The direct-mapped cache is easy

- Each memory address belongs in exactly one block in cache
- Indices and offsets can be computed with simple bit operators
- But, not flexible

Problems when multiple blocks map to same entry

- Cause thrashing in cache

Cache misses

Types of cache misses:

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.
  - Program start-up or context switch
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
  - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

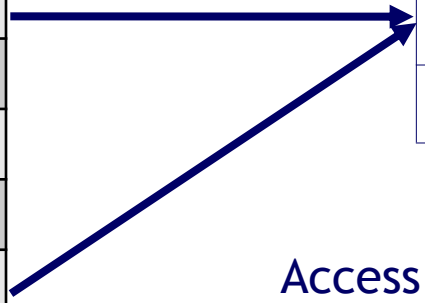
Issue #2: (S,E,B,m) = (4,1,1,4) Conflict misses

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Index	Tag	Data
00		
01		
10		
11		



Access pattern:
0010
0110
0010
0110
0010
0110
…

Each access results in a cache miss and a load into cache block 2.

Only one out of four cache blocks is used.

Miss rate = 100%

t (2 bits)	s (2 bits)

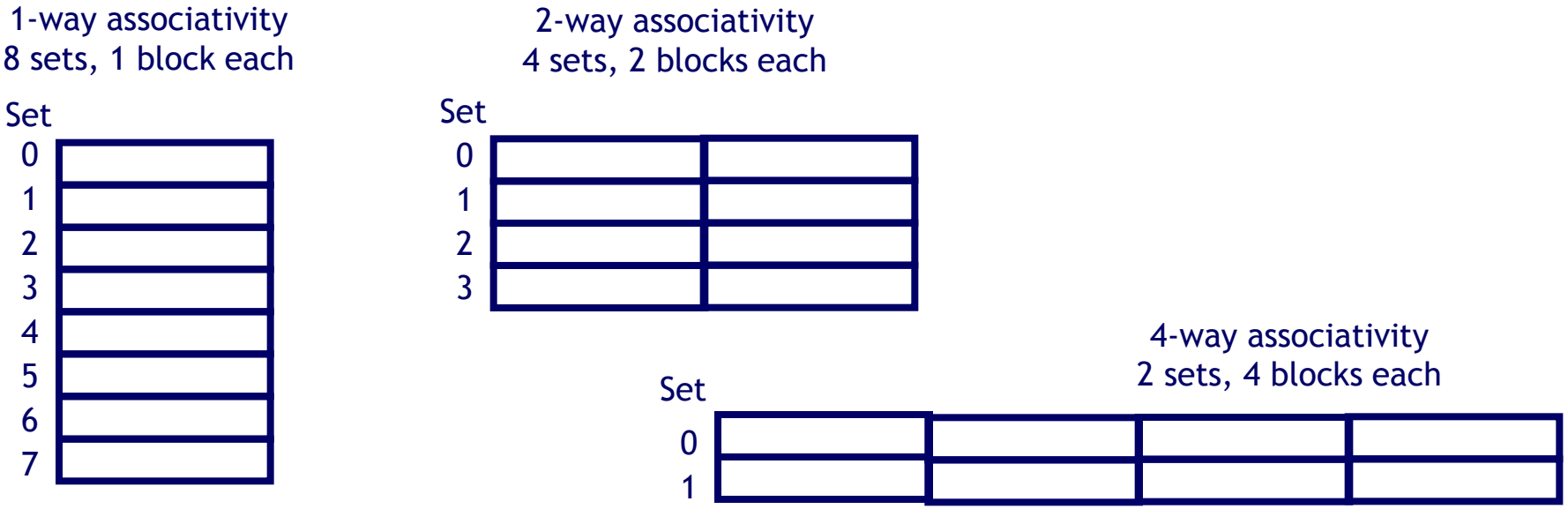
Set associativity

Set associative caches

- Each set can store multiple distinct blocks/lines
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.

If each set has 2^x blocks, the cache is an 2^x-way associative cache.

Here are several possible organizations of an eight-block cache.



Set associative caches

Each memory address is assigned to a particular set in the cache, but not to a specific block in the set.

- **Set sizes range from 1 (direct-mapped) to all blocks in cache (fully associative).**
- **Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.**

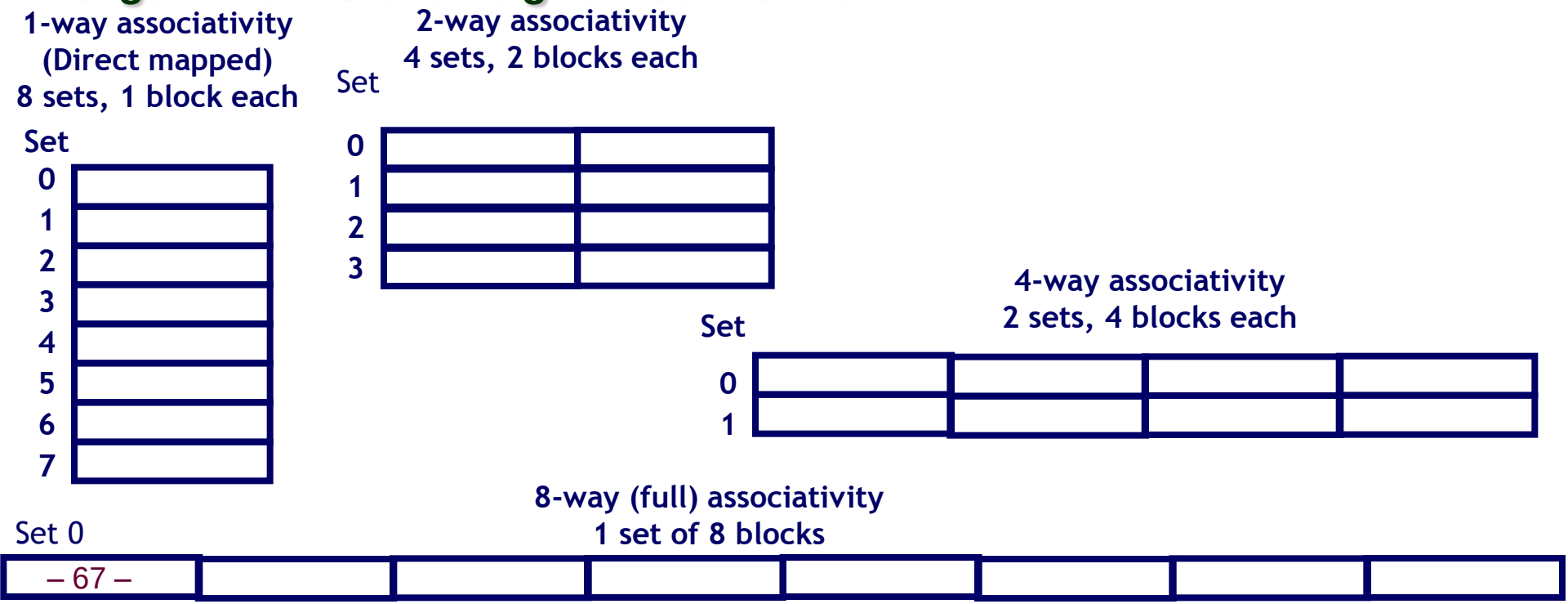
Set associative caches

Cache is divided into *groups* of blocks, called sets.

- **Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.**

If each set has 2^x blocks (E), the cache is an 2^x-way associative cache.

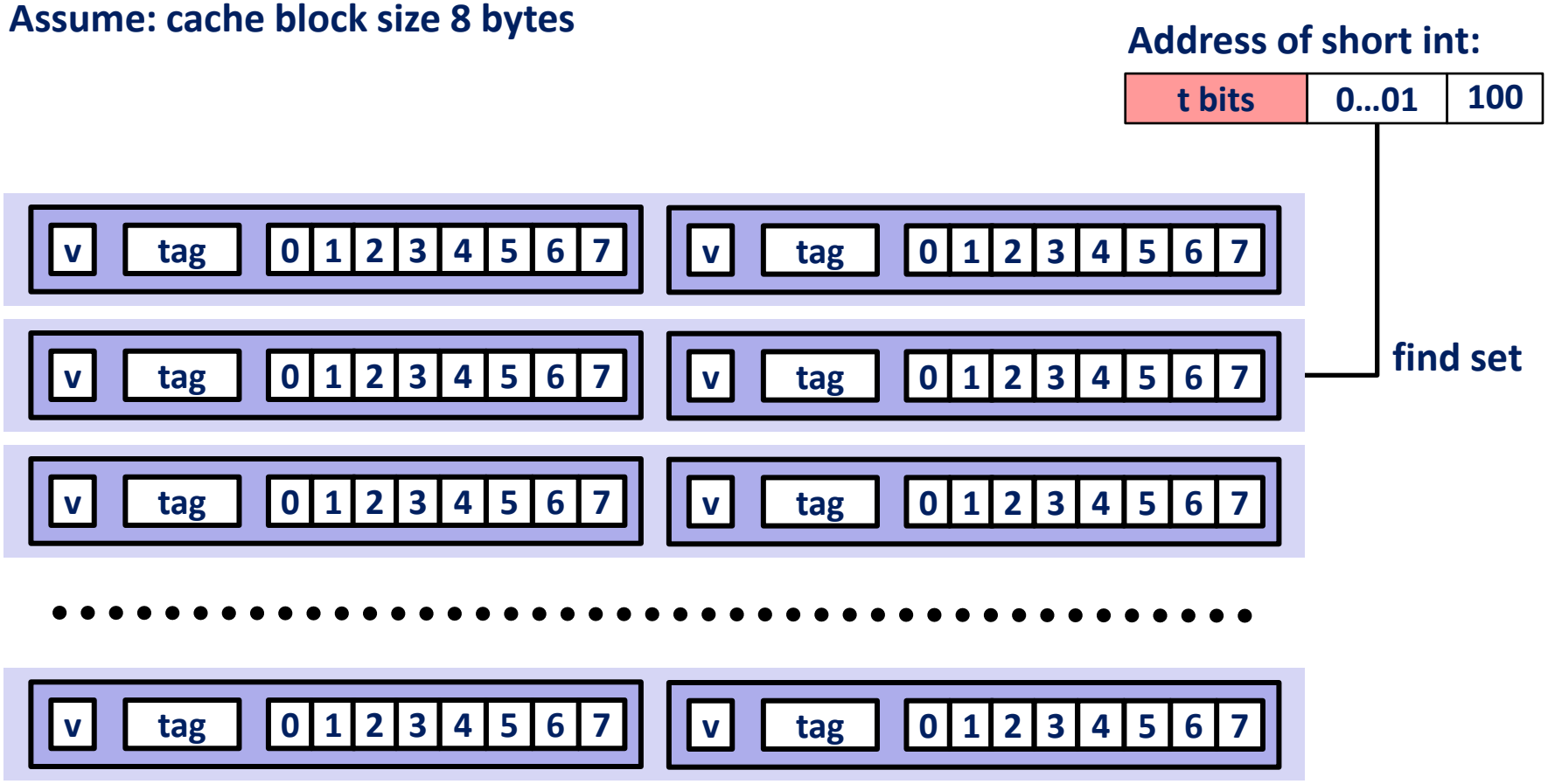
Organizations of an eight-block cache.



E-way Set Associative Cache (E = 2)

E = 2: Two lines per set

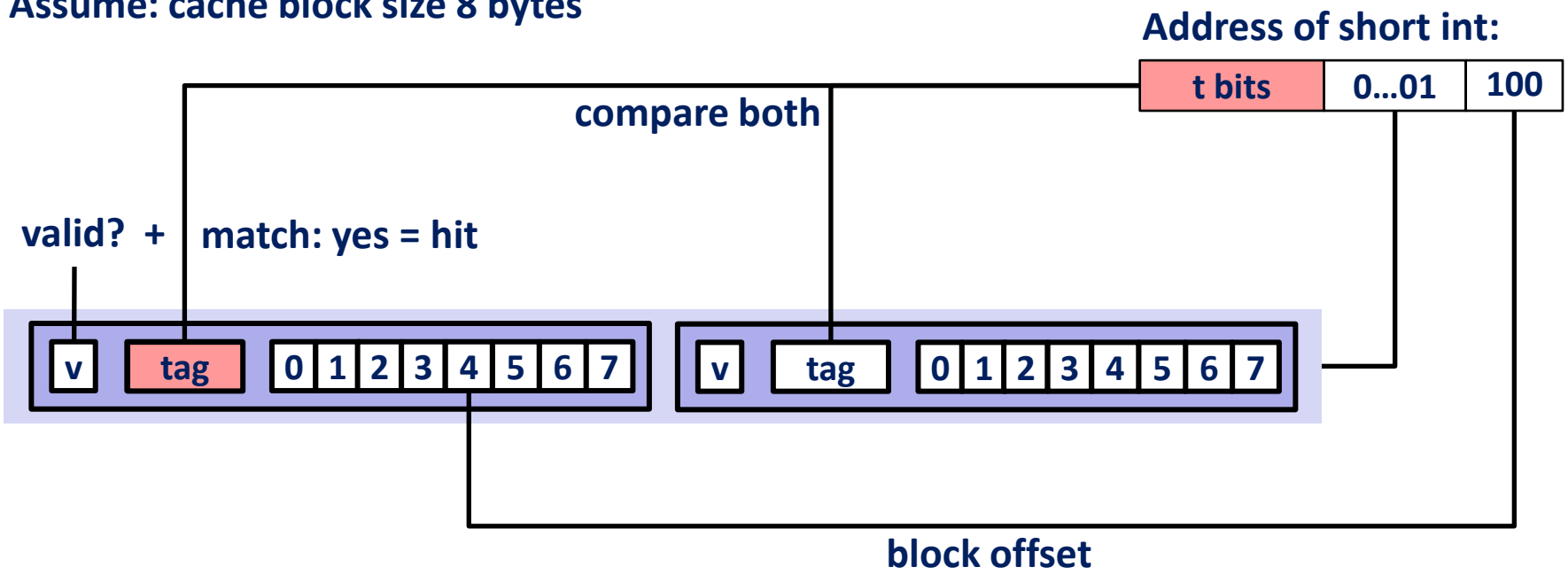
Assume: cache block size 8 bytes



E-way Set Associative Cache (E = 2)

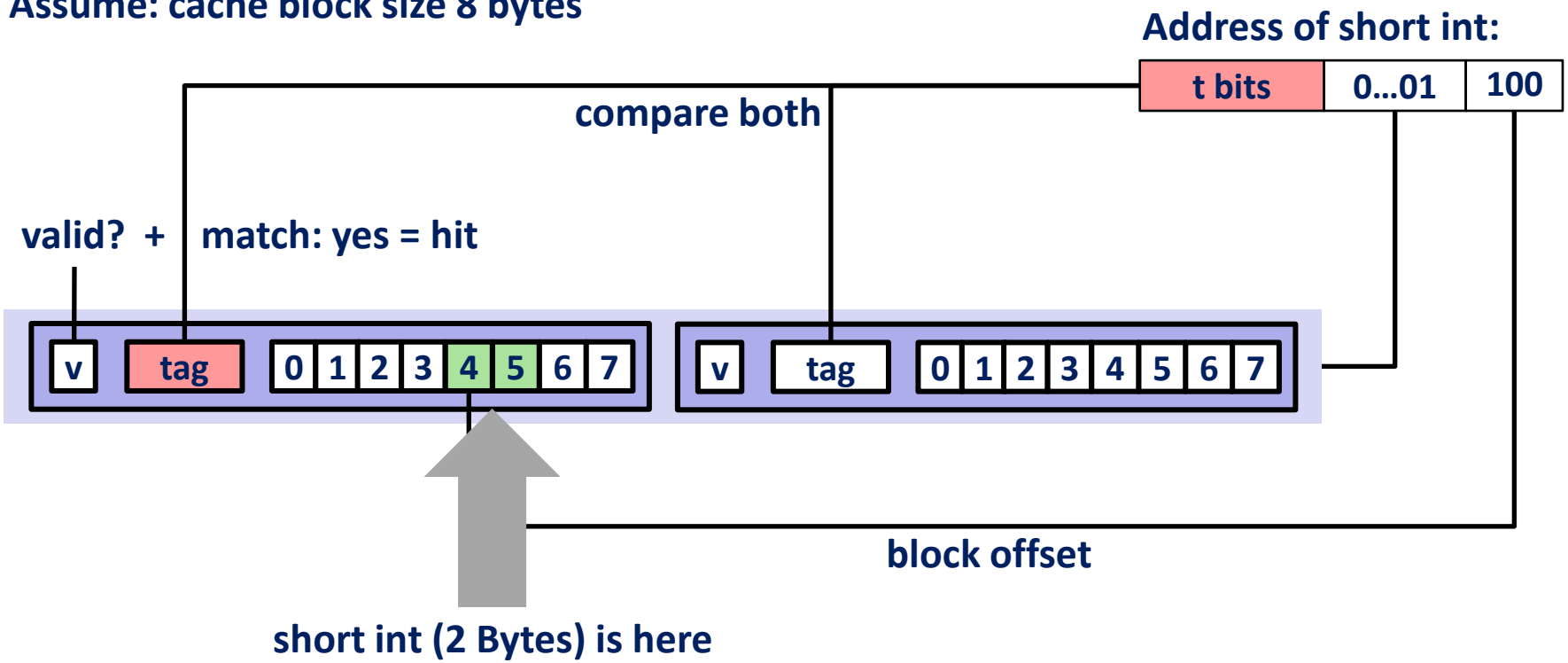
E = 2: Two lines per set

Assume: cache block size 8 bytes



E-way Set Associative Cache (E = 2)

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



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

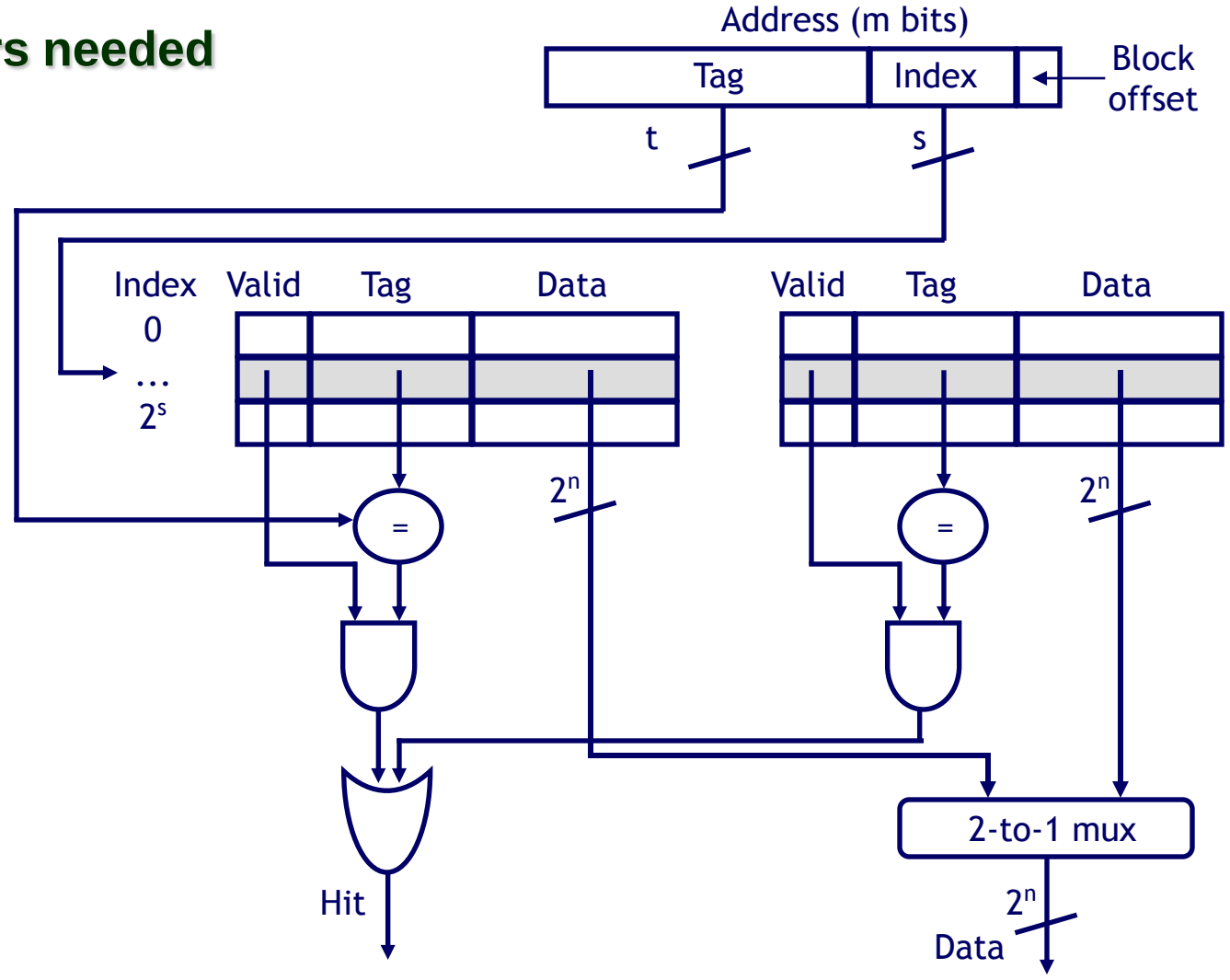
Address trace (reads, one byte per read):

0	[00$\underline{0}$0$_2$],	miss
1	[00$\underline{0}$1$_2$],	hit
7	[01$\underline{1}$1$_2$],	miss
8	[10$\underline{0}$0$_2$],	miss
0	[00$\underline{0}$0$_2$]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

2-way set associative cache implementation

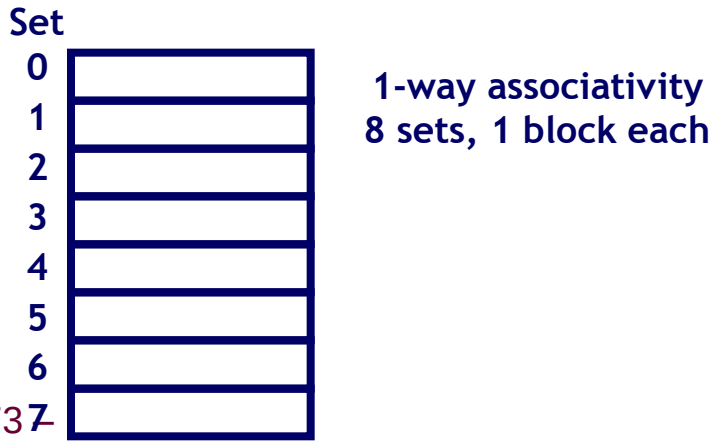
Two comparators needed



Examples

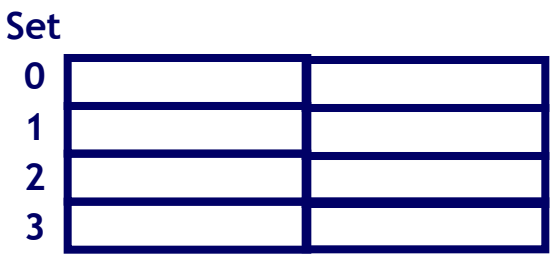
Assume 8 block cache with 16 bytes per block

- Where would data from this address go 110000 011 0011.
- B=16 bytes, so the lowest 4 bits are the block offset.

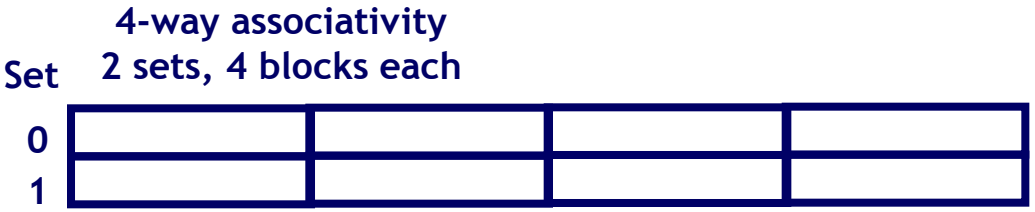


1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each



4-way associativity
2 sets, 4 blocks each



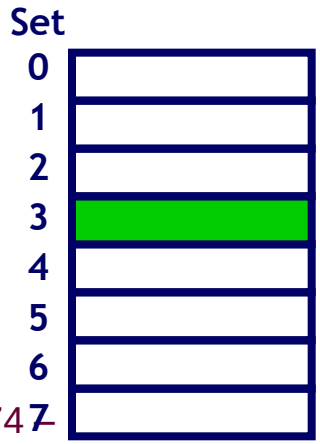
Examples

Assume 8 block cache with 16 bytes per block

- Where would data from this address go 110000 011 0011.
- B=16 bytes, so the lowest 4 bits are the block offset.
- Set index
  - 1-way (i.e. direct mapped) cache
    - » 8 sets so next three bits (011) are the set index.
  - 2-way cache
    - » 4 sets so next two bits (11) are the set index.
  - 4-way cache
    - » 2 sets so the next one bit (1) is the set index.

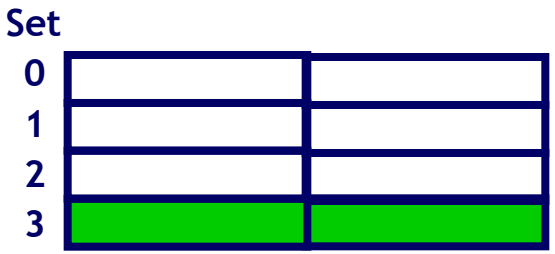
Data may go in *any* block (in green) within the correct set.

- Top bits are always for the tag

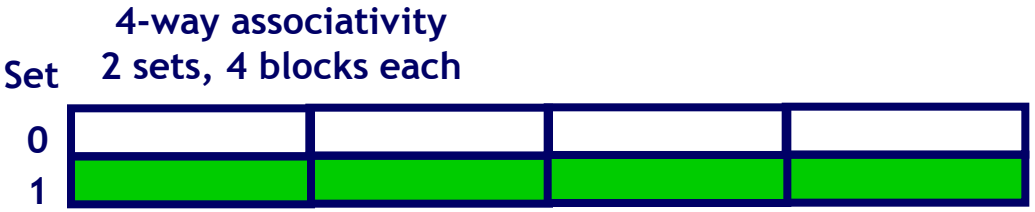


1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each



4-way associativity
2 sets, 4 blocks each

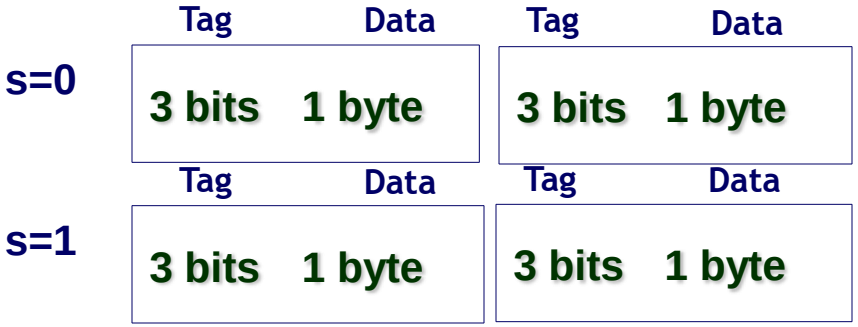


Practice problem

Consider the following set associative cache: (S,E,B,m) = (2,2,1,4)

- Derive values for number of address bits used for the tag (t), the index (s) and the block offset (b)
  - s = 1
  - b = 0
  - t = 3
- Draw a diagram of which bits of the address are used for the tag, the set index and the block offset

- Draw a diagram of the cache



Extra

Practice problem

Byte-addressable machine

- **m = 16 bits**
- **Cache = (S,E,B,m) = (?, 1, 1, 16)**
  - **Direct mapped, Block size = 1**
- **s = 5 (5 LSB of address gives index)**

How many blocks does the cache hold?

32

How many bits of storage are required to build the cache (data plus *all overhead* including tags, etc.)?

(11 + 8 + 1) * 32

Tag Byte Valid