Announcement

- Lab 9 (graded)
  - To be returned this afternoon
- Assignment 6
  - Due November 11
  - Sorting: Combining C and IA-32 assembly

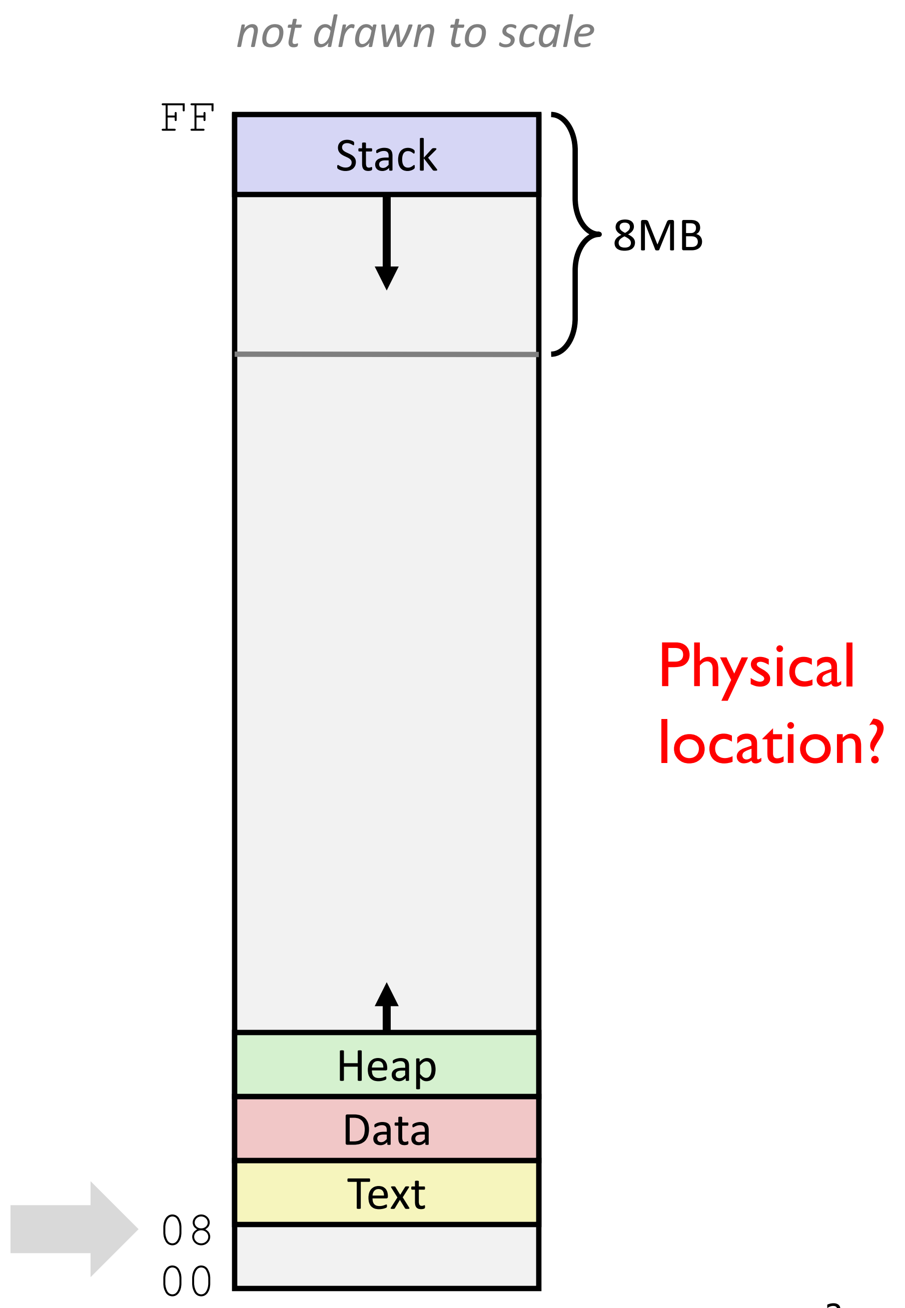
Lecture 25

Locality

CPSC 275
Introduction to Computer Systems

IA-32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - e. g., local variables
- Heap
  - Dynamically allocated storage
  - When call `malloc`, `calloc`, `new`
- Data
  - Statically allocated data
  - e.g., globals & strings declared in code
- Text
  - Executable machine instructions
  - Read-only (*re-entrant*)



Upper 2 hex digits
= 8 bits of address

Random-Access Memory (RAM)

- Key features
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.
- RAM comes in two varieties:
  - DRAM (dynamic RAM)
  - SRAM (static RAM)

SRAM vs DRAM

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

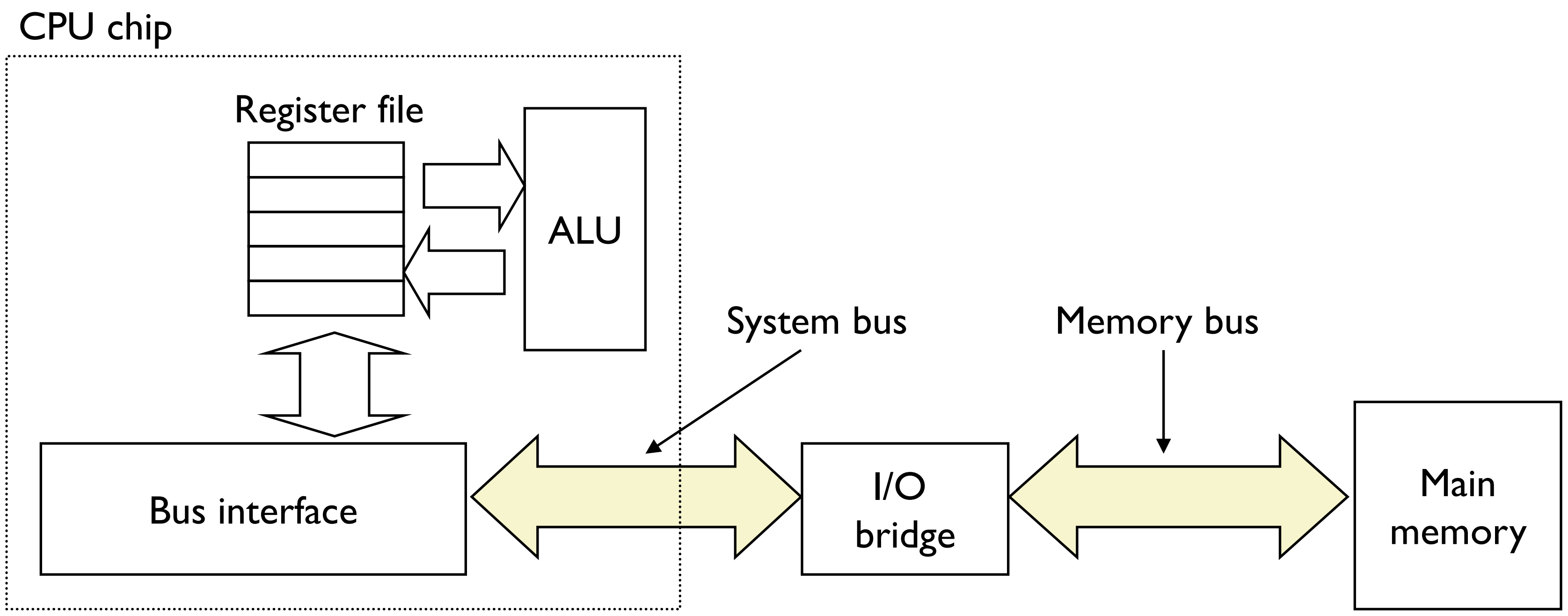
DRAM and SRAM are *volatile* memories
(lose information if powered off)

Nonvolatile Memories

- *Nonvolatile* memories retain value even if powered off
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability
  - Flash memory: EEPROMs. with partial (block-level) erase capability
- Uses for nonvolatile memories
  - *Firmware* programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Solid state disks (SSD)
  - Disk caches

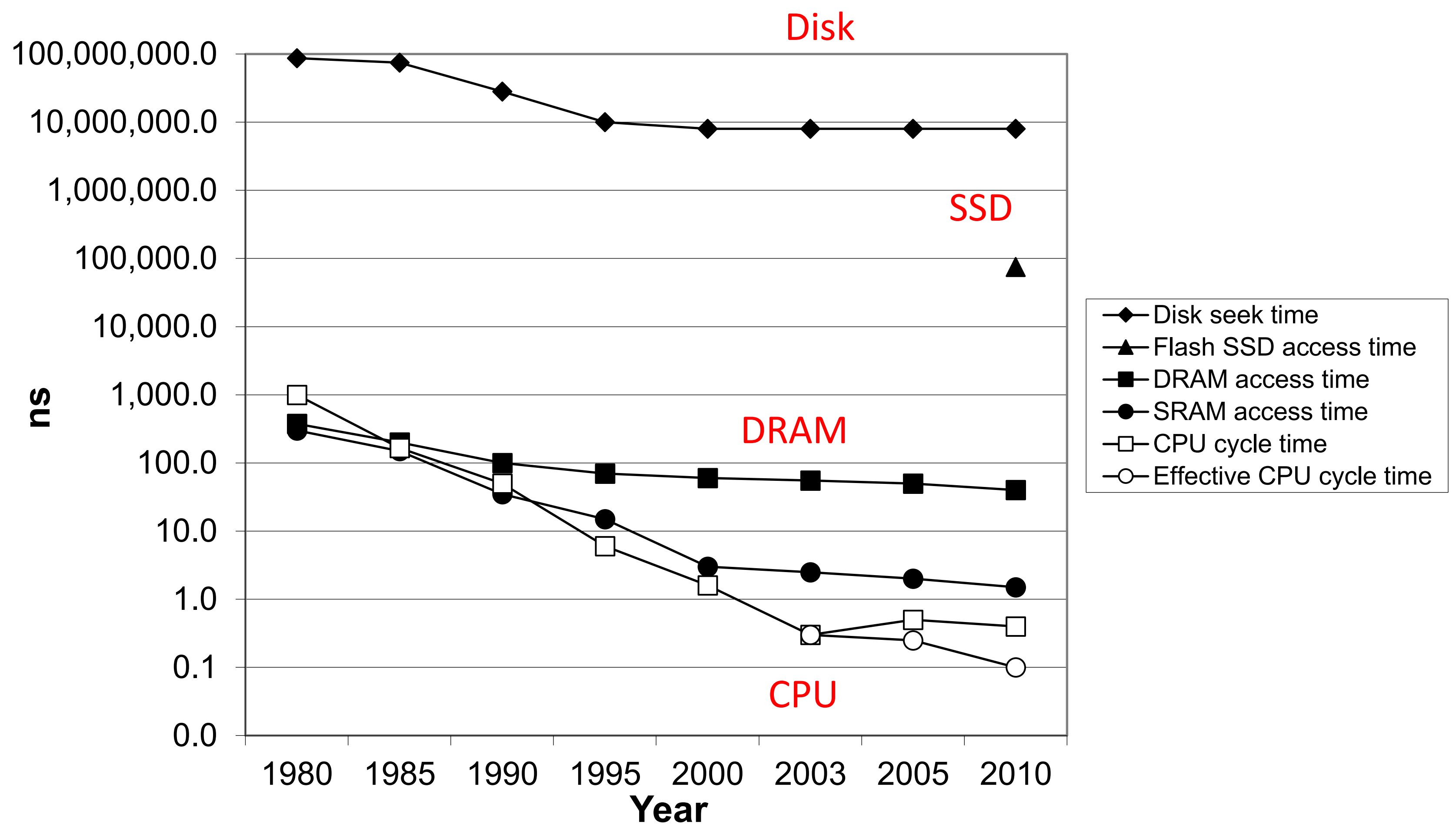
Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



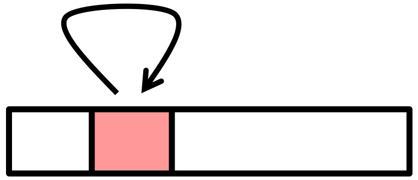
Locality to the rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality.

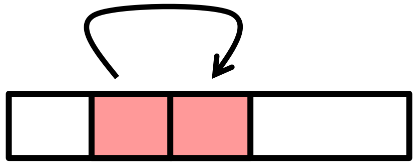
What is Locality?

- Principle of Locality: Programs tend to use data and instructions with 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 tend to be referenced close together in time



Locality Example

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

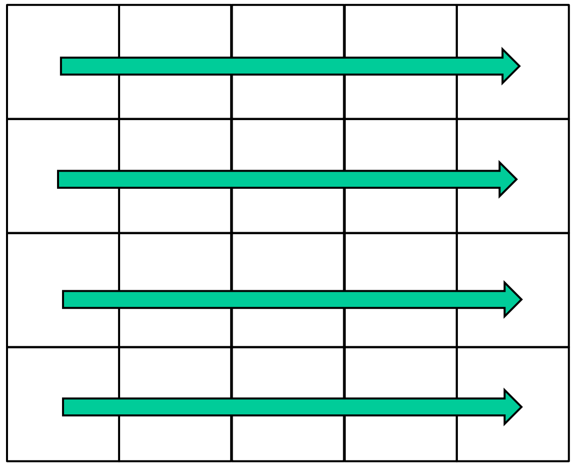
- data references
  - Reference array elements in succession (stride-1 reference pattern). spatial locality
  - Reference variable `sum` each iteration. temporal locality
- instruction references
  - Reference instructions in sequence. spatial locality
  - Cycle through loop repeatedly. temporal locality

Qualitative Estimates of Locality

- Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- *Q*: Does this function exhibit good locality with respect to array `a`?

```
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;
}
```



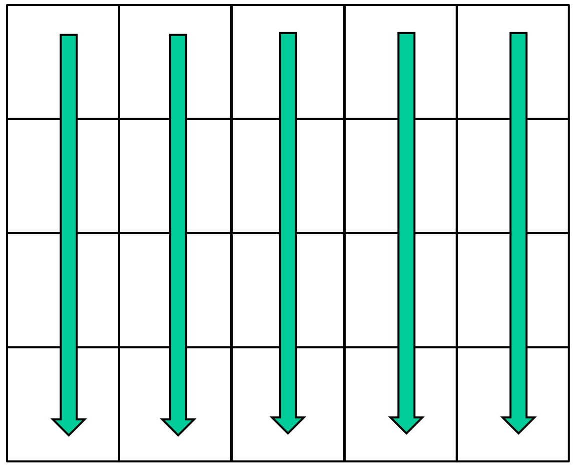
Q: Stride? 1

Locality Example

- *Q*: Does this function exhibit good locality with respect to array `a`?

```
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;
}
```



Q: Stride? `N`

- *Q*: Which one is better?

Locality Example

- *Q*: Can you 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? (Homework)

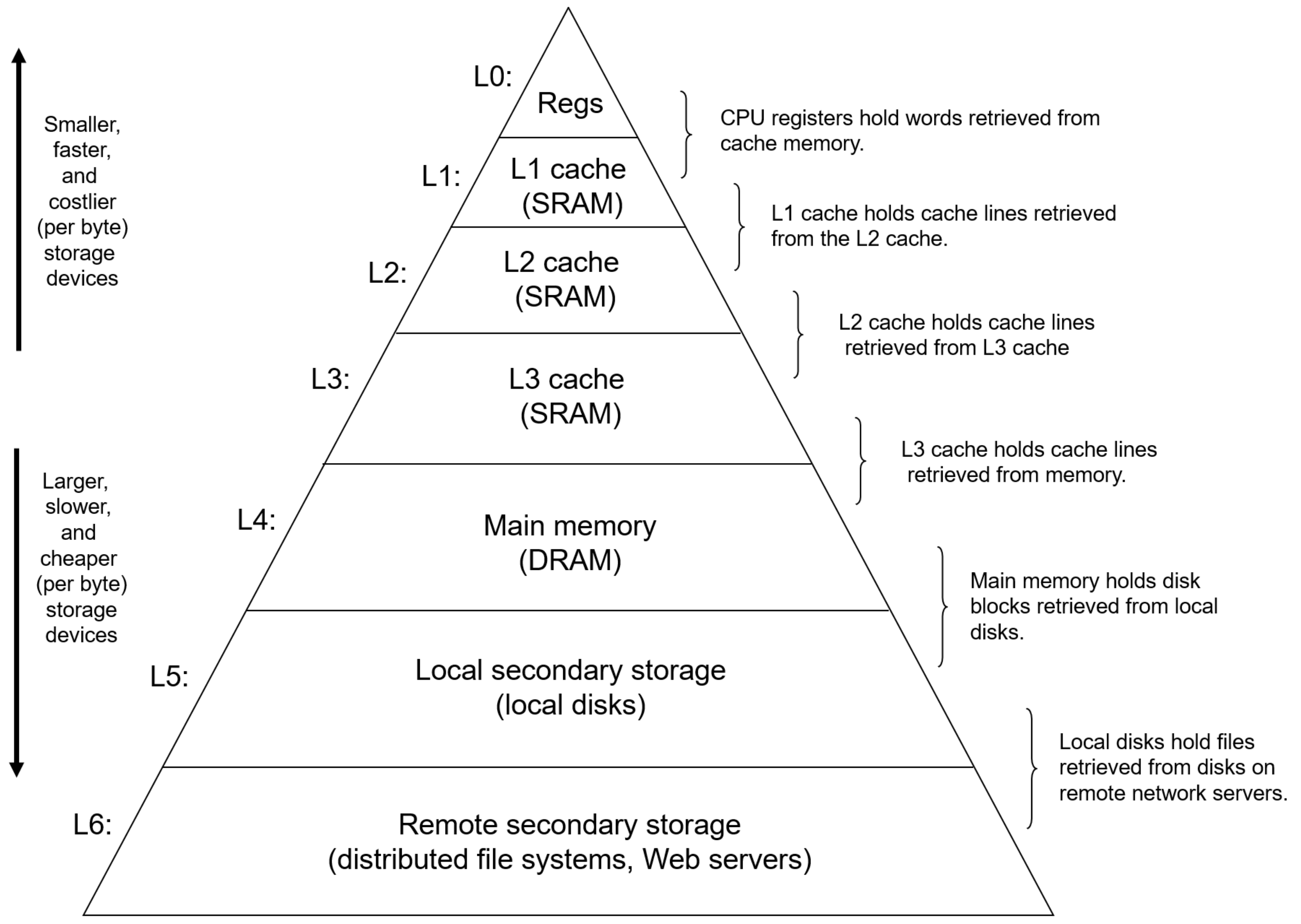
```
int sum_array_3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

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

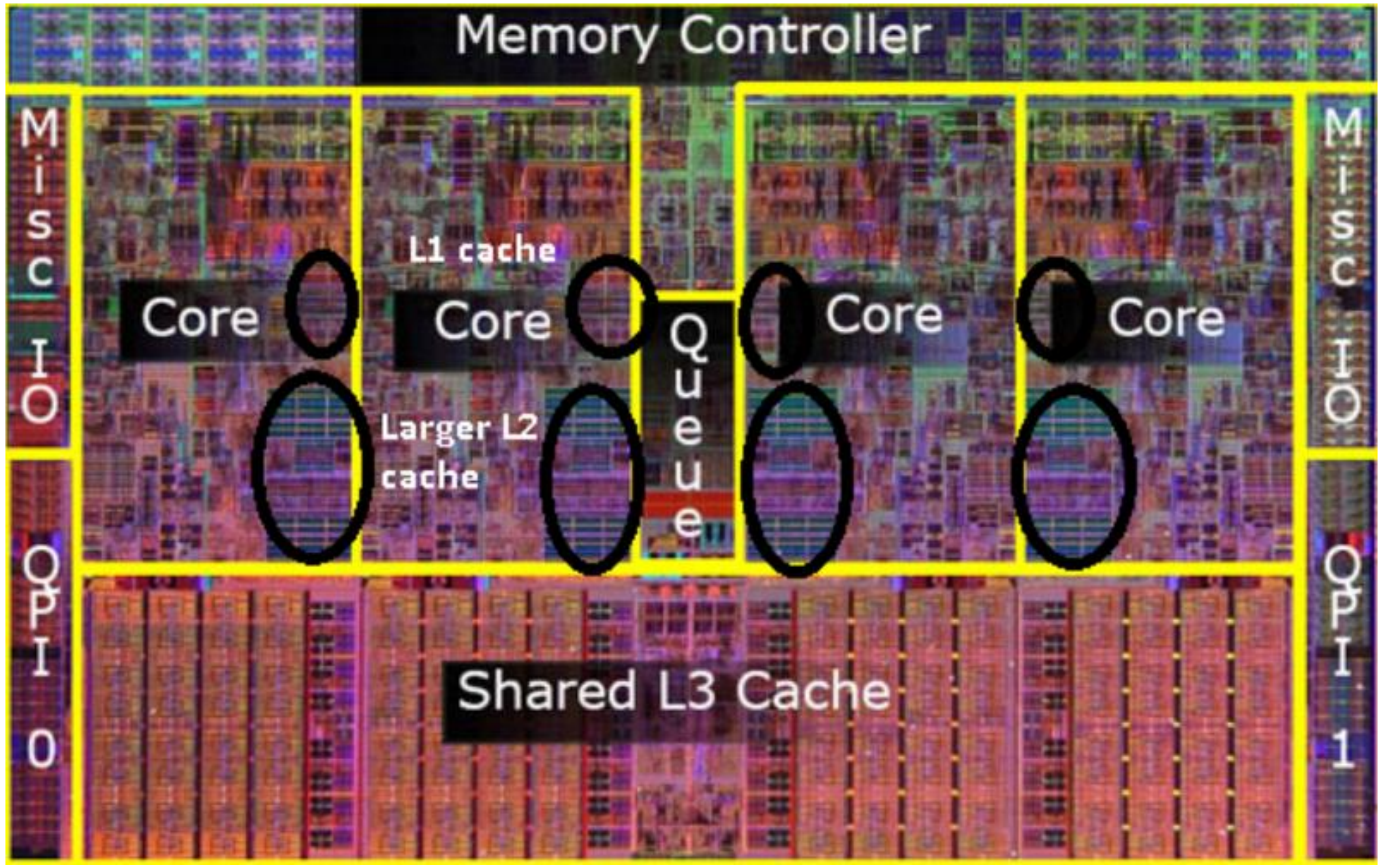
Memory Hierarchy

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - **Well-written** programs tend to exhibit **good locality**.
- This led to an approach of organizing memory and storage systems known as a memory hierarchy.

Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Where is cache?

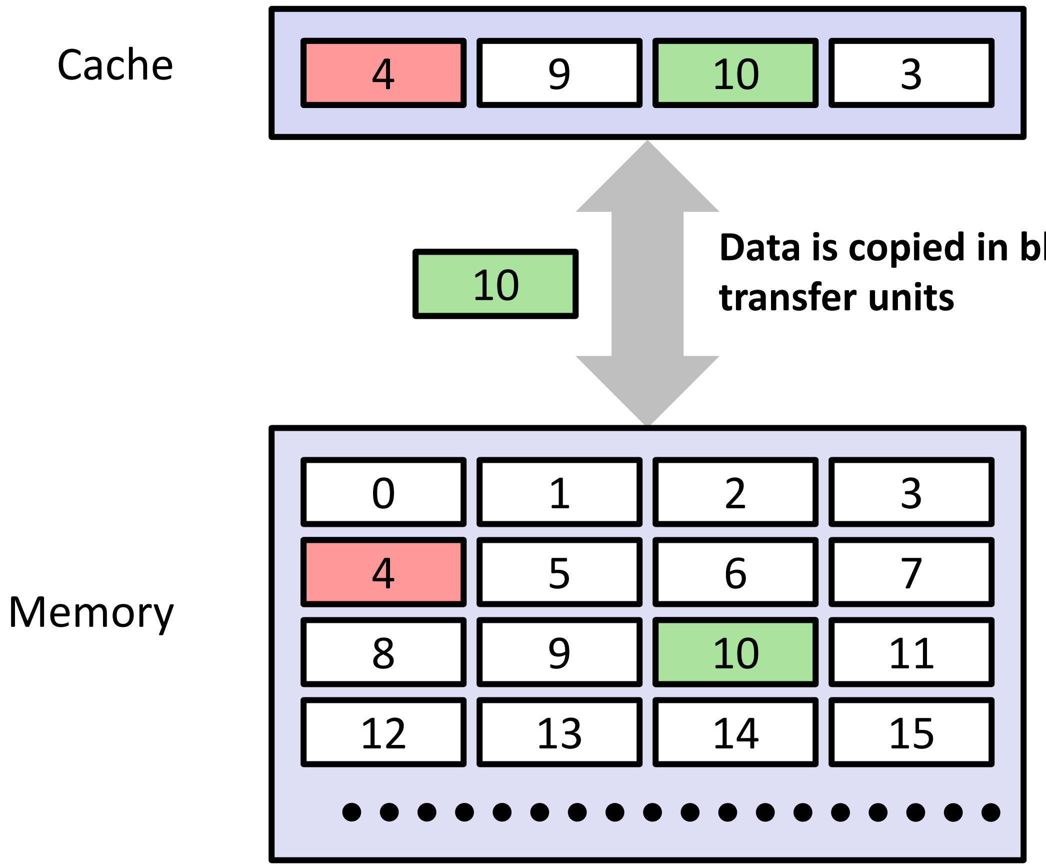


A typical quad-core processor (Intel i7)

Cache

- *Cache*: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each *k*, the faster, smaller device at level *k* serves as a cache for the larger, slower device at level *k*+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level *k* more often than they access the data at level *k*+1.
  - Thus, the storage at level *k*+1 can be slower, and thus larger and cheaper per bit.

General Cache Concepts

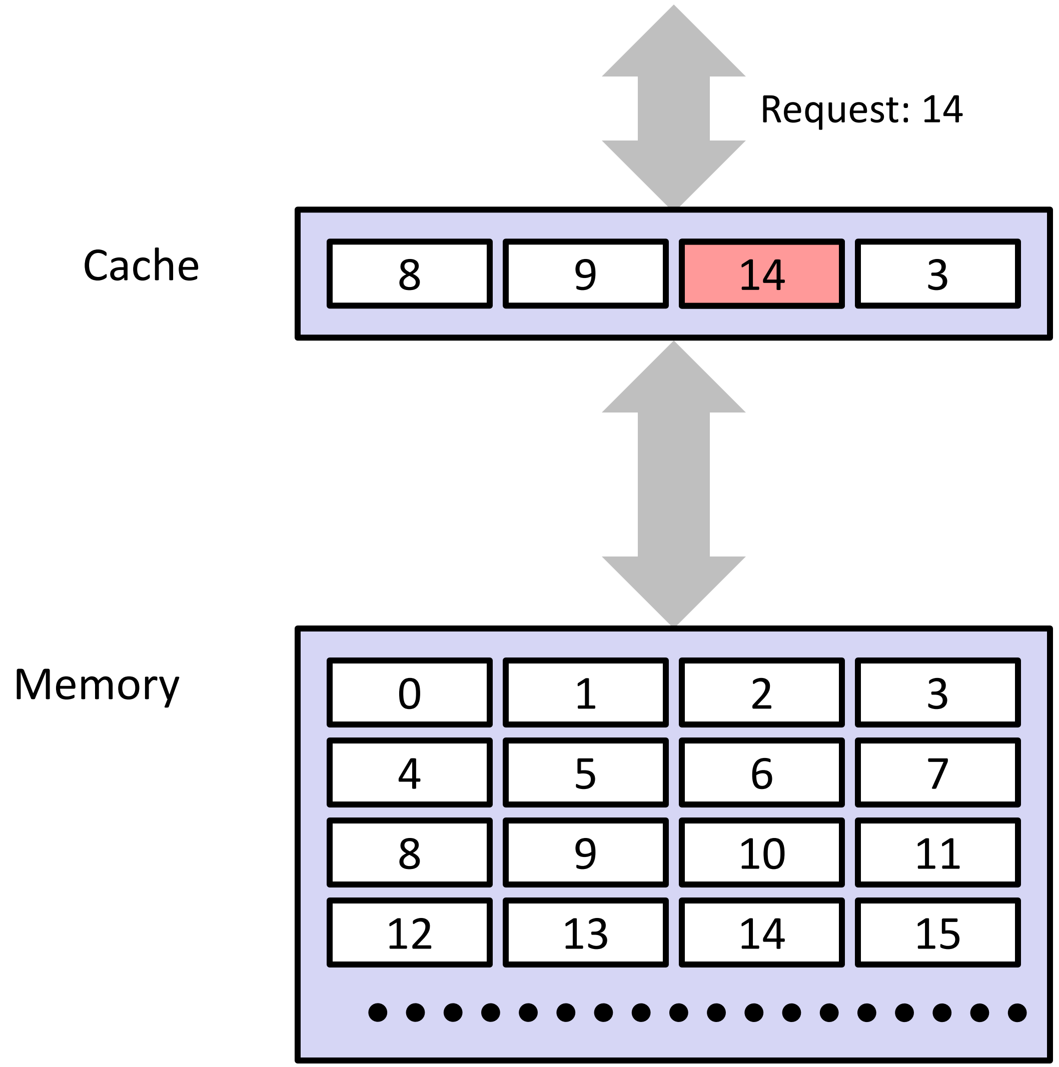


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "blocks"

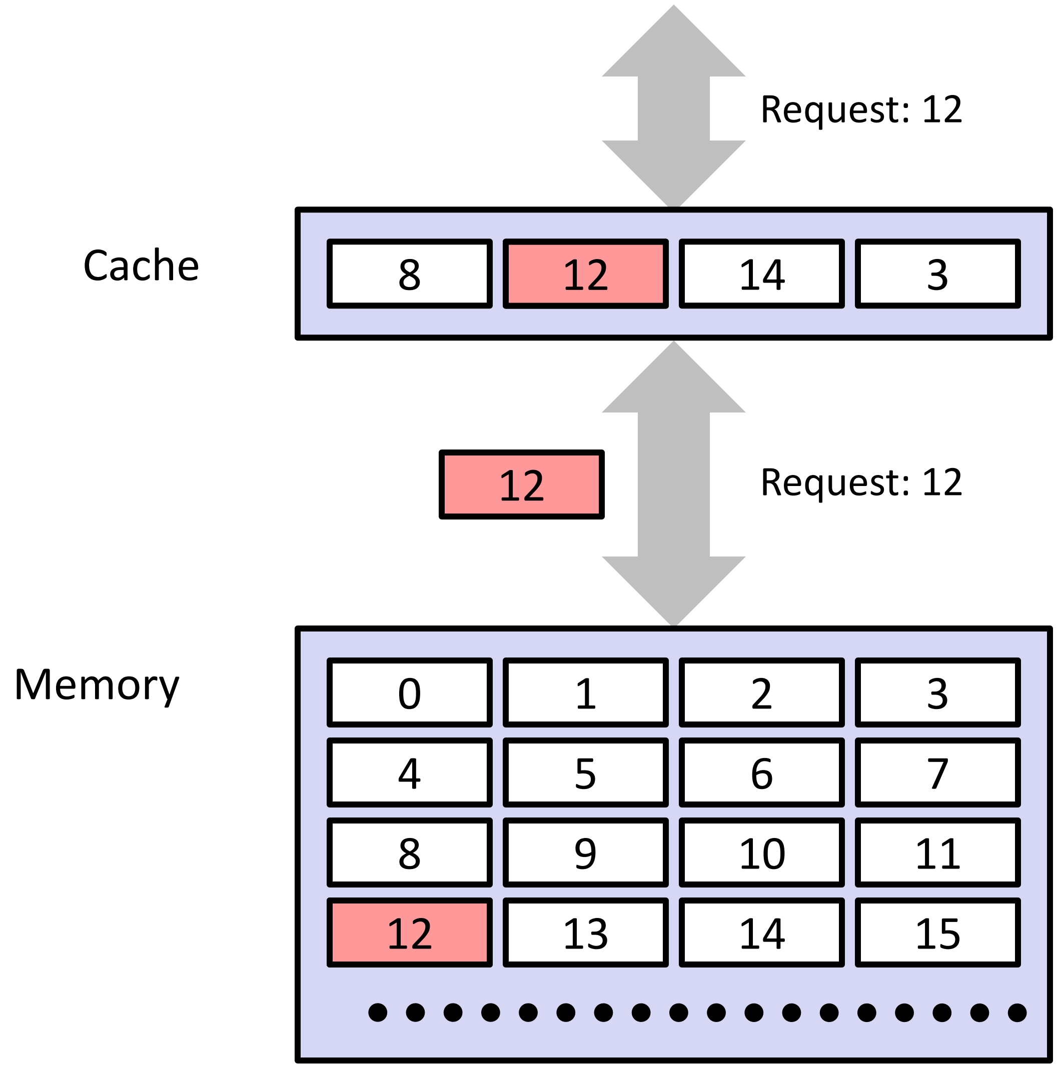
General Cache Concepts



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Misses



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

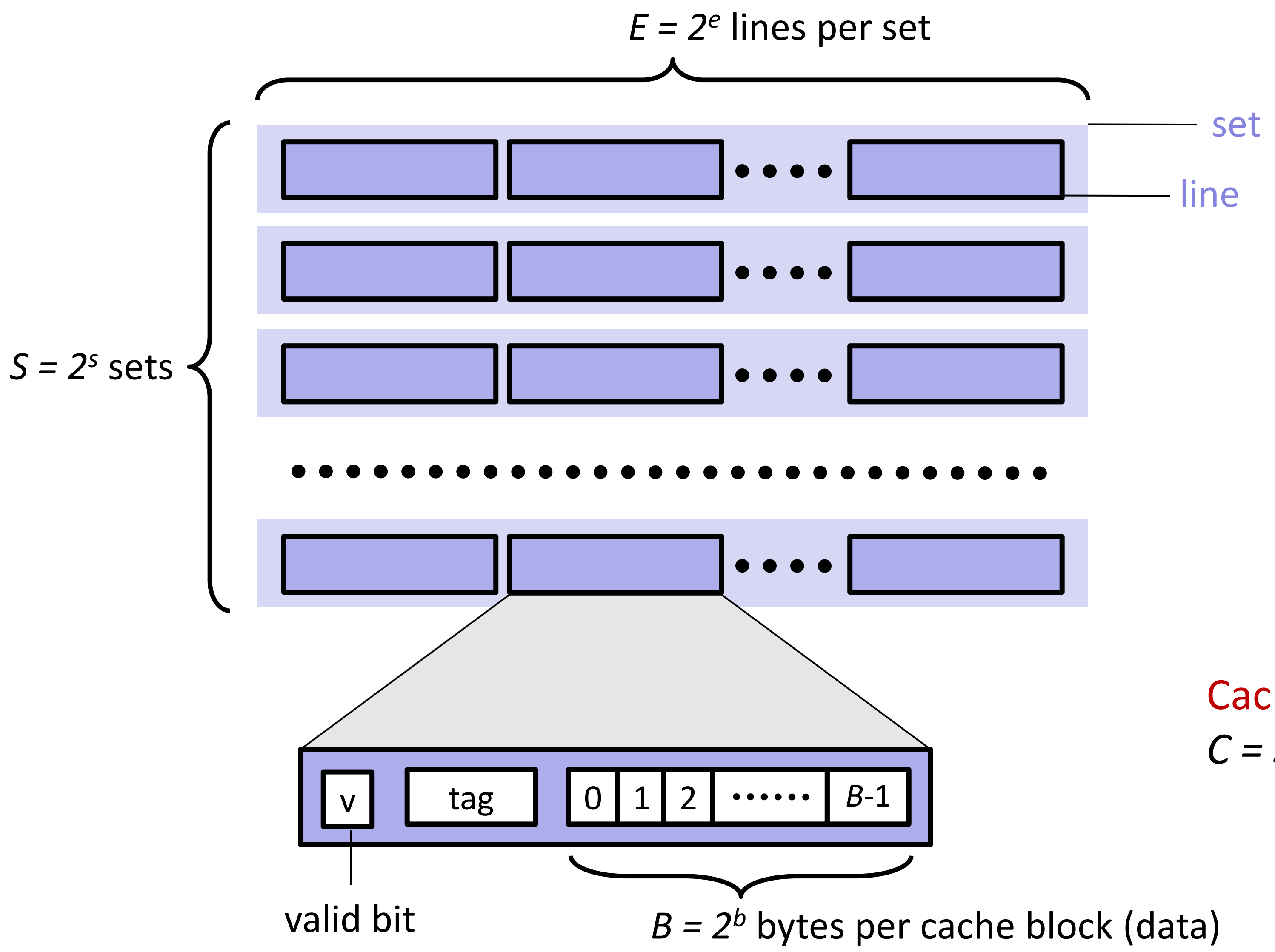
Block b is stored in cache

- Placement policy: determines where *b* goes
- Replacement policy: determines which block gets evicted (*victim*)

Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Conflict miss
  - Most caches limit blocks at level *k*+1 to a small subset of the block positions 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 (known as the *working set*) is larger than the cache.

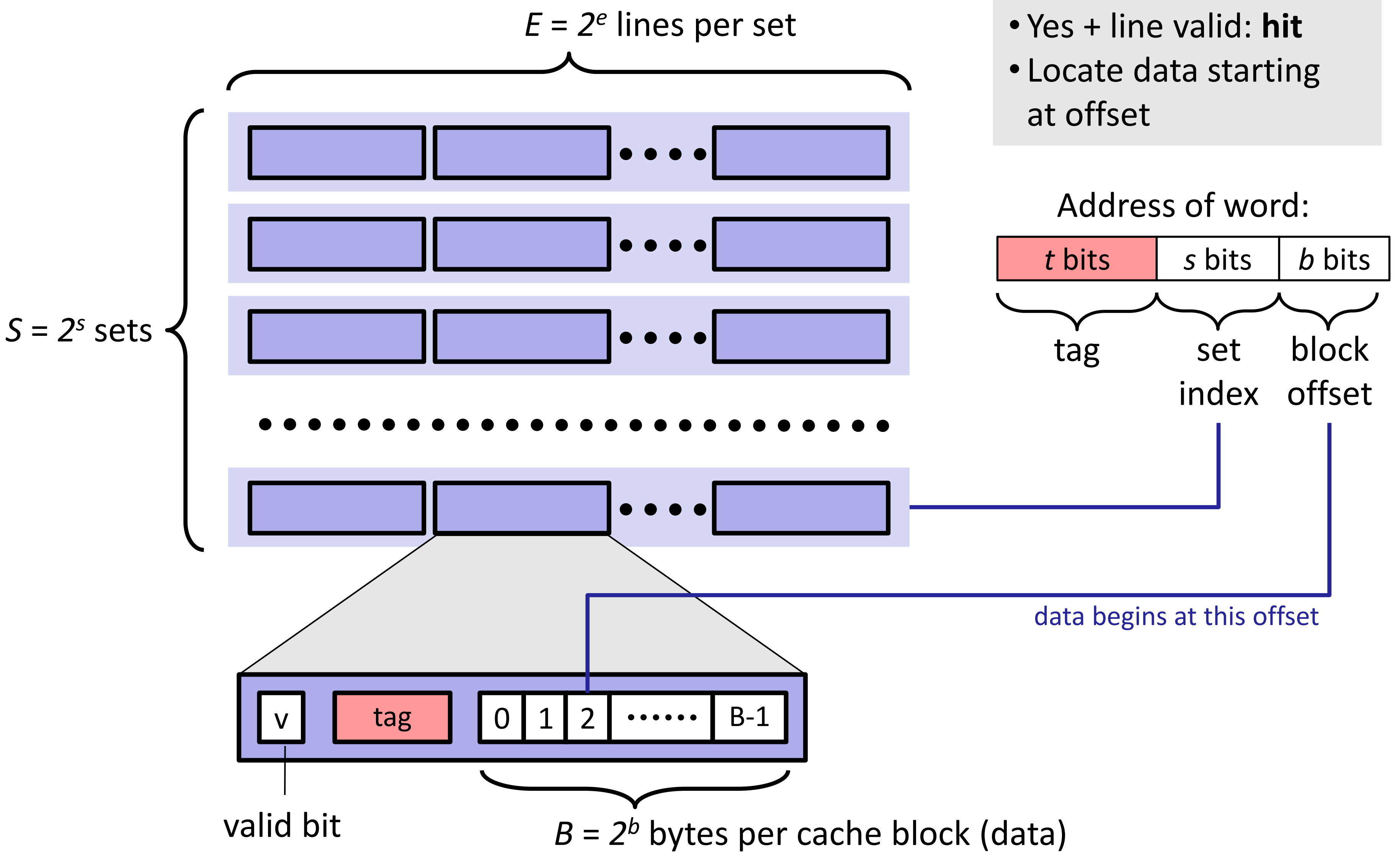
General Cache Organization (*S*, *E*, *B*)



Cache size?
$C = S \times E \times B$ bytes

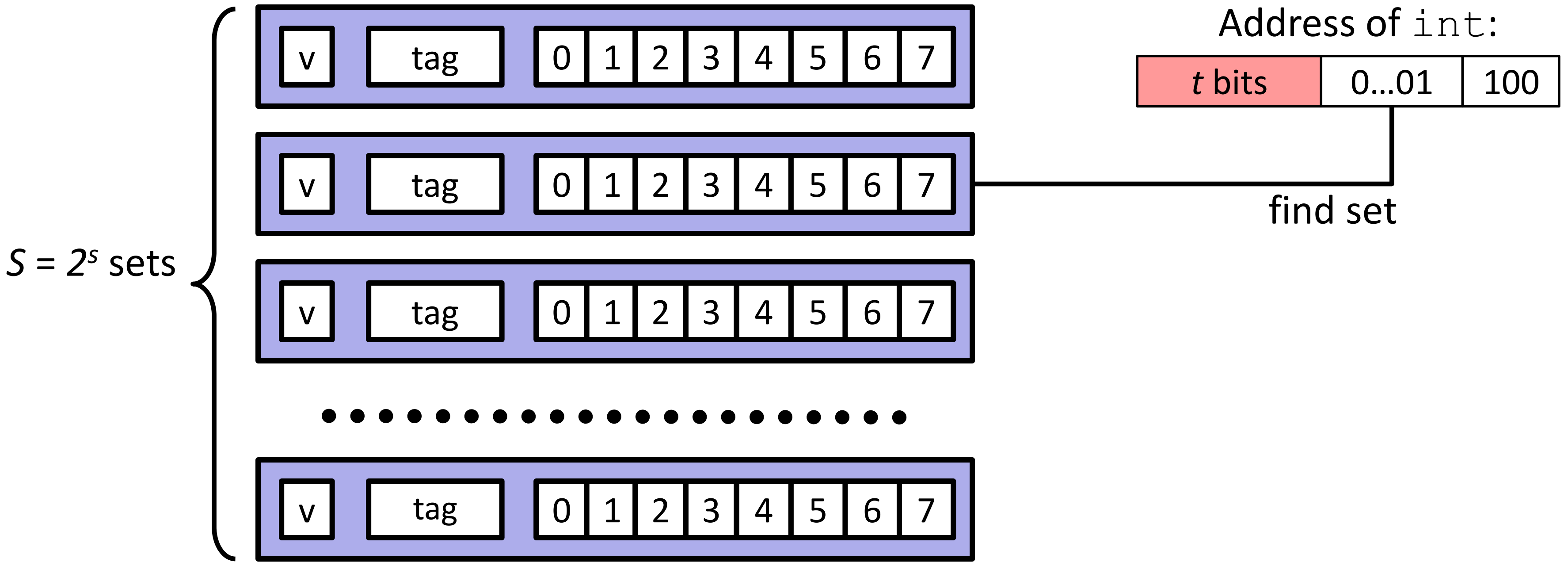
Cache Read

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



Example: Direct Mapped Cache (E = 1)

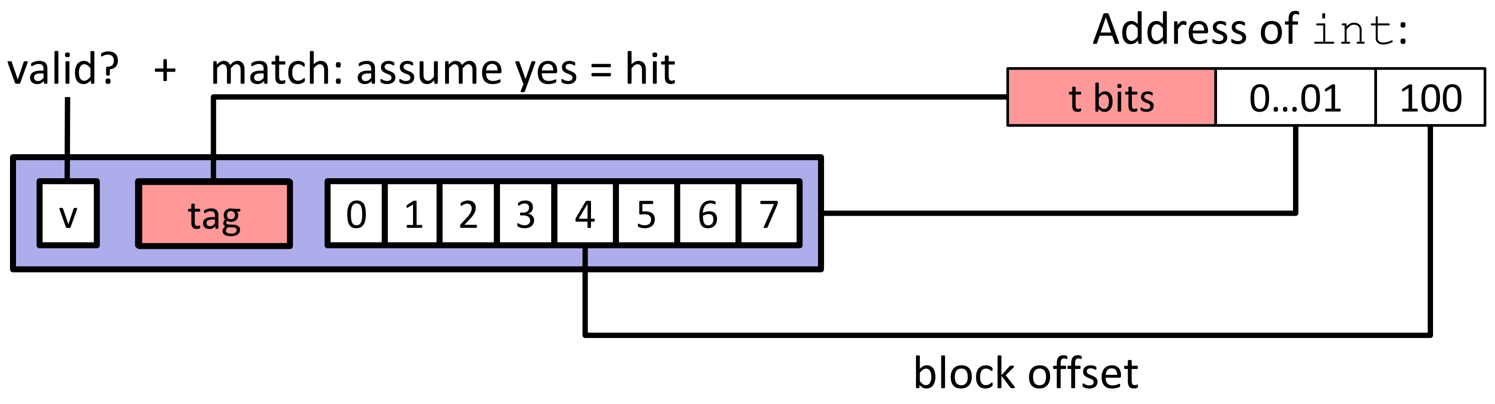
Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

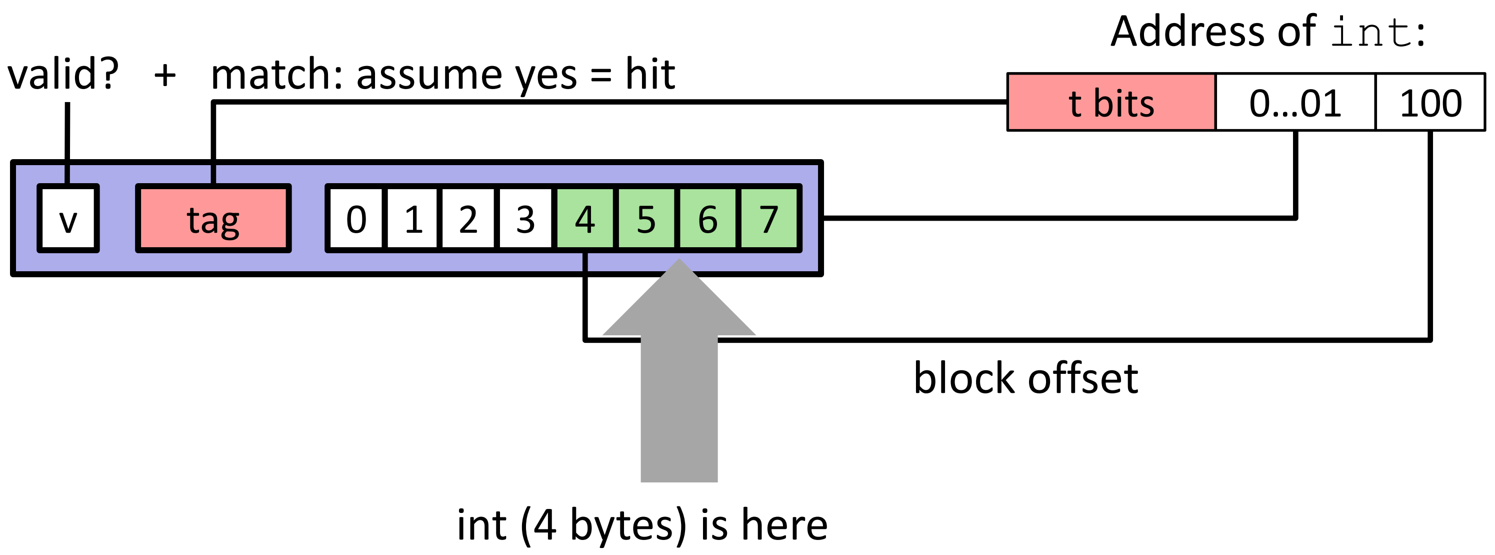
Direct mapped: One line per set

Assume: cache block size 8 bytes



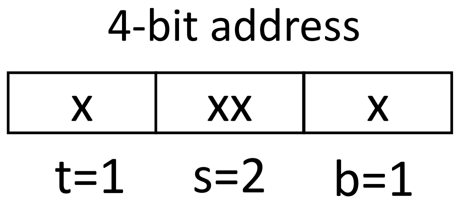
Example: Direct Mapped Cache (E = 1)

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



If no match, an old line is evicted and replaced.

Direct-Mapped Cache Simulation



M=16 byte addresses (total size of memory)
B=2 bytes/block
S=4 sets
E=1 line/set

Direct-Mapped Cache Simulation

4-bit address

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

M=16 byte addresses (total size of memory)
B=2 bytes/block
S=4 sets
E=1 line/set

Address trace:

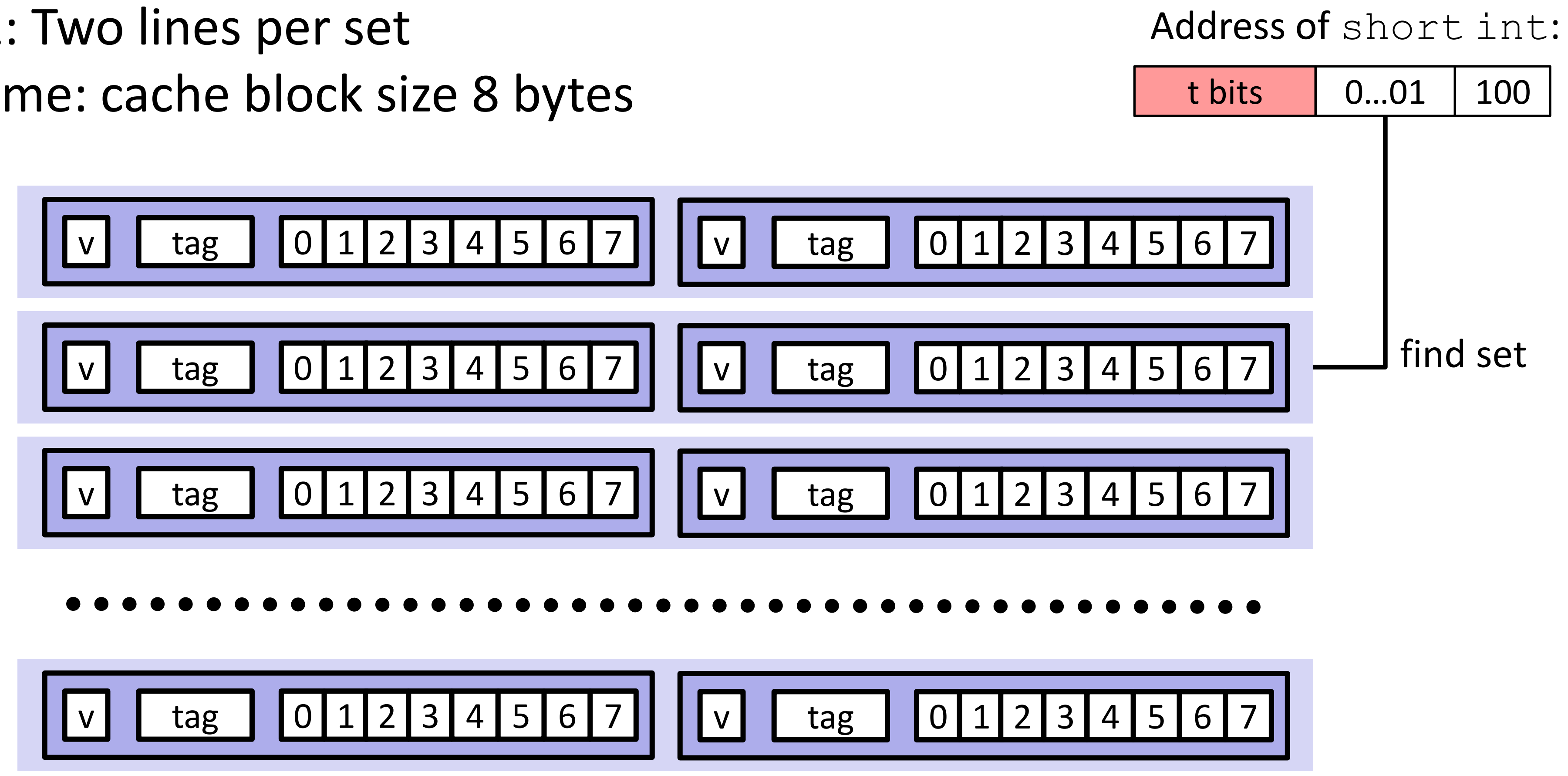
0	$[0\underline{00}0_2]$	miss
1	$[0\underline{00}1_2]$	hit
7	$[0\underline{11}1_2]$	miss
8	$[1\underline{00}0_2]$	miss
0	$[0\underline{00}0_2]$	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

2-way Set Associative Cache

E = 2: Two lines per set

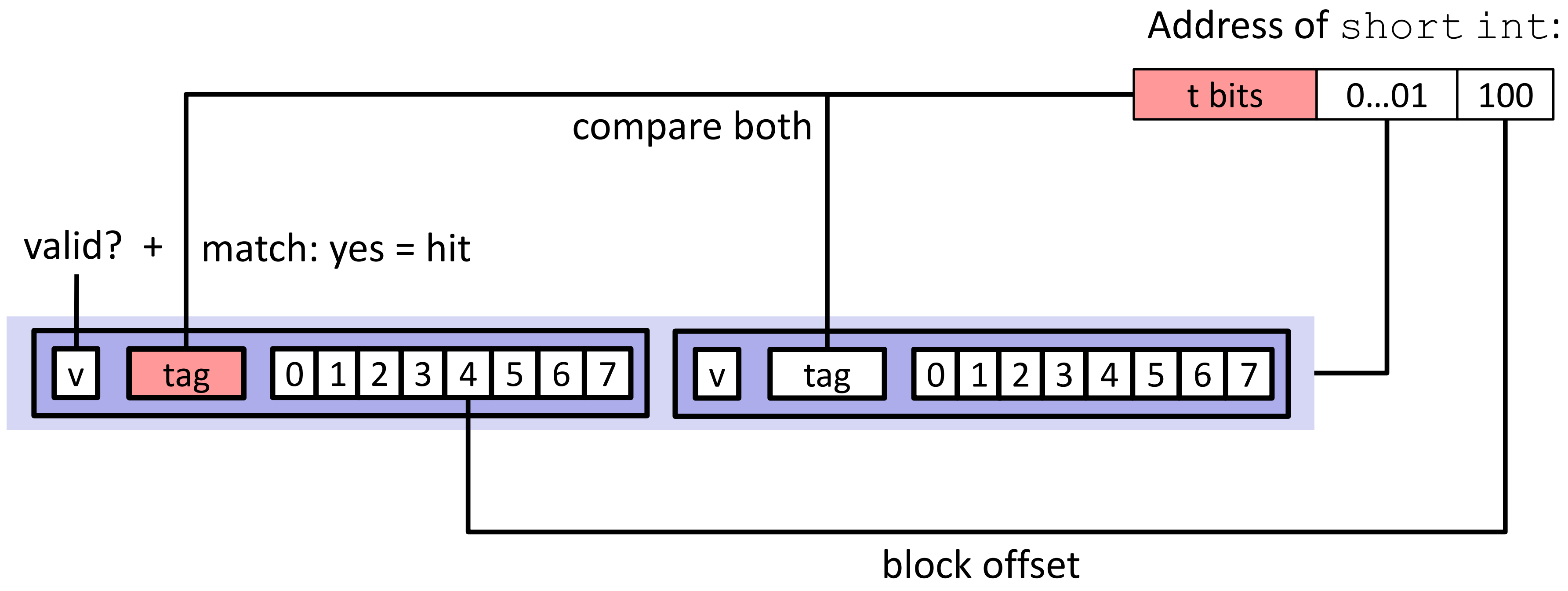
Assume: cache block size 8 bytes



2-way Set Associative Cache

E = 2: Two lines per set

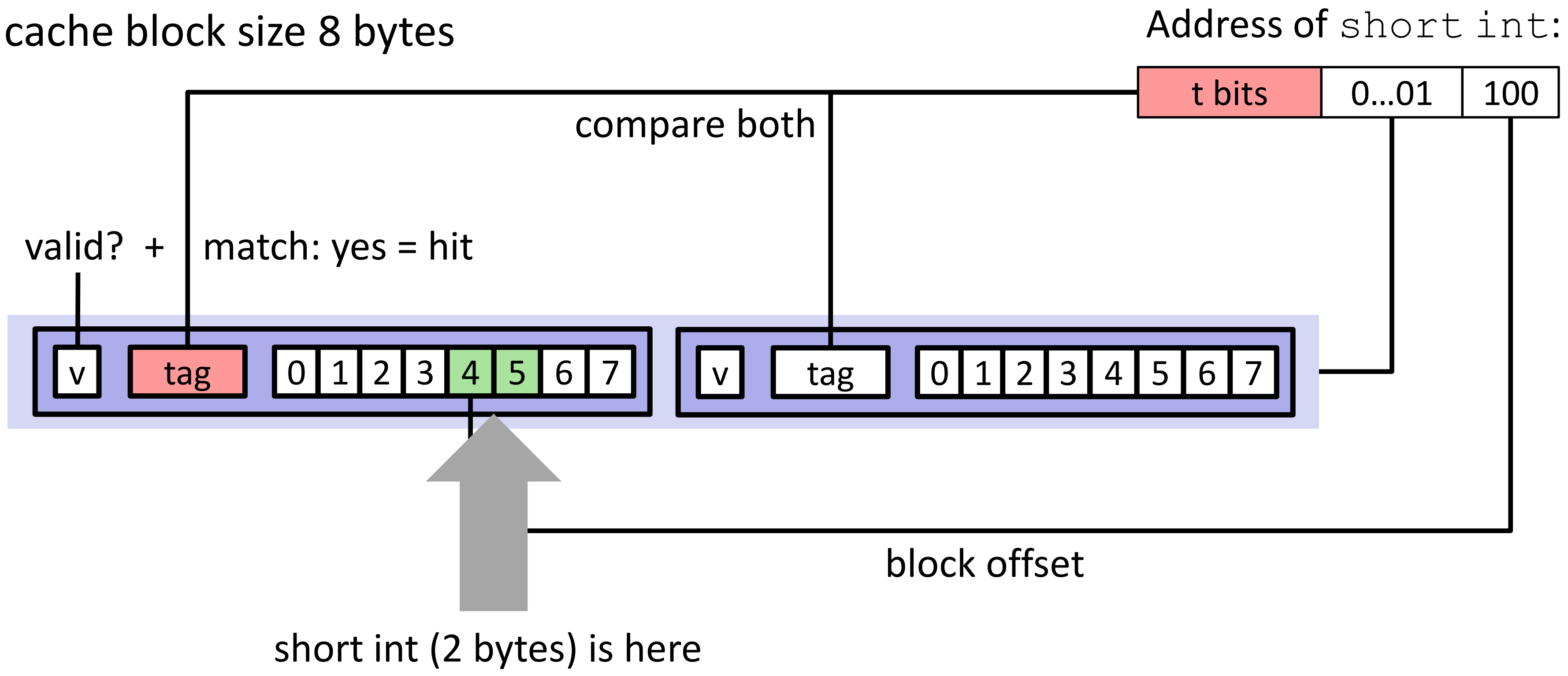
Assume: cache block size 8 bytes



2-way Set Associative Cache

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

4-bit address

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

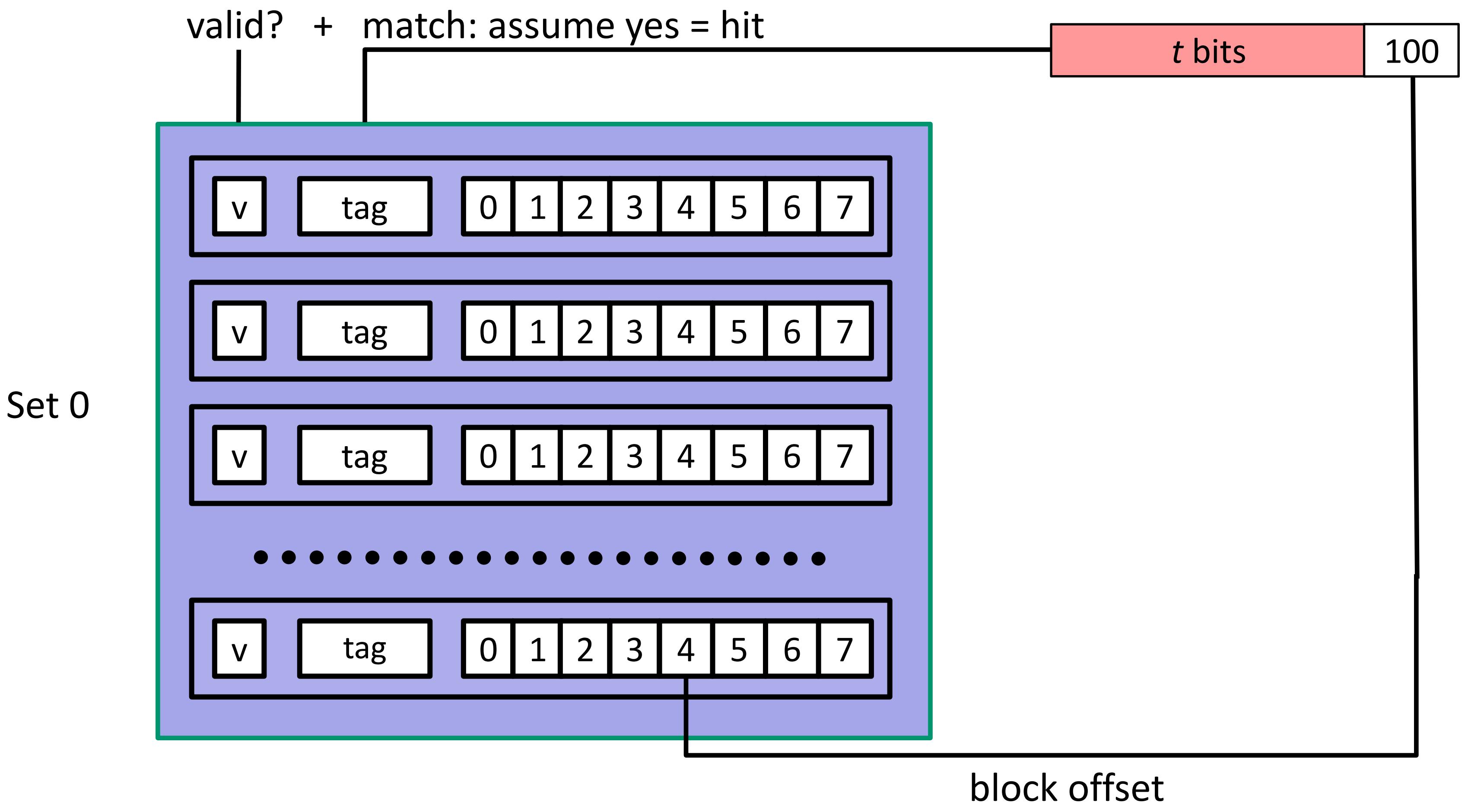
M=16 byte addresses,
B=2 bytes/block,
S=2 sets,
E=2 lines/set

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		

Fully Associative Cache (S = 1)



Assignment 7: Cache Simulator

- A C program that simulates the behavior of a cache memory
  - Parameters: *s*, *E*, *b*
  - Given a memory access *trace file* as input, simulates the hit/miss behavior of a cache memory on this trace and prints the total number of hits, misses, and evictions.

Trace Files

- Trace files generated by `valgrind`, a Linux utility program for memory debugging, memory leak detection, and profiling
  - Example trace file

```
I 0400d7d4,8          // instruction load
 M 0421c7f0,4         // modify (a load followed by a store)
 L 04f6b868,8         // load
 S 7ff0005c8,8        // store
↑       ↑        ↑
space   address  data size
```

Trace Files, cont'd

- For this assignment, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "`I`").
- To help you simplify the code, all lines with an instruction load have been already removed from trace files.

LRU Replacement Policy

- Necessary conditions for replacement
    - a cache miss occurred
    - the current set is full
- Need to determine which line from the current set will be replaced by a new line (*eviction*).
- Pick the one *least recently used* (referenced).
    - Why does it work?
    - How to implement?

Running Your Simulator

```
$ ./mycache -s 2 -E 1 -b 4 -t traces/tiny.trace
```

To verify the result:

```
$ ./refcache -s 2 -E 1 -b 4 -t traces/tiny.trace
```

Programming Notes

- For arbitrary *s*, *E*, and *b*, you will need to allocate storage for your simulator's data structures using `malloc`.
- 64-bit address field: `unsigned long int`
- Ignore the request data sizes in the traces, that is, there is no need to define blocks in lines.
- Use a timestamp to keep track of the last time each line has been referenced.
- Start with small trace files.

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- Example:

```
$ ./repeat 10 computer
```

Command-Line Arguments

- To obtain access to *command-line arguments*, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

Command-Line Arguments

- `argc` (“argument count”) is the number of command-line arguments.
- `argv` (“argument vector”) is an array of pointers to the command-line arguments (stored as strings).

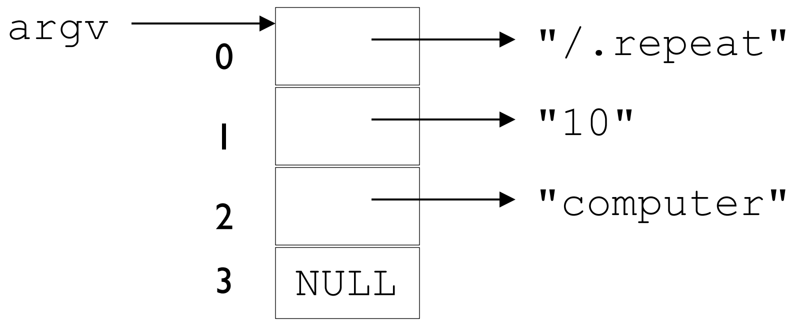
- `argv[0]` points to the name of the program,
- `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.

Command-Line Arguments

- If the user enters the command line

```
$ ./repeat 10 computer
```

then `argc` will be 3 (why?),
and `argv` will have the following appearance:



Command-Line Arguments

- Since `argv` is an array of pointers, accessing command-line arguments is easy, e.g,

```
int i;
for (i = 1; i < argc; i++)
   printf(“%s\n”, argv[i]);
```

Output:

```
/.repeat
10
computer
```

- For numeric arguments, conversion might be necessary:

```
int count = atoi(argv[1]);
```

Command-Line Options

- Command-line *options* modify the program's behavior.
- Command-line arguments and options

```
$ ls -l myfile
```

option argument

- Some command-line options take values:

```
$ tail -n 20 myfile
```

- The `getopt` function is useful in parsing command-line options.

The `getopt` Function

```
#include <unistd.h>

int getopt(int argc, char *argv[], char *optstring);
extern char *optarg;
```

- *argc* is the number of arguments.
- *argv* is an array of arguments.
- *optstring* is a string containing the option characters.
  - If such a character is followed by a colon, the option requires a value.
- *optarg* is an *external* variable string containing the option values.
- If there are no more option characters, the function returns -1.

The `getopt` Function

- Normally, `getopt` is called in a loop.
- When `getopt` returns -1, indicating no more options are present, the loop terminates.
- A `switch` statement is used to dispatch on the return value from `getopt`.

Example using `getopt`

three options: `a`, `b`, and `c`
`b` and `c` require values
`c`'s value is a number

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    int r, cval;
    while ((r = getopt(argc, argv, "ab:c:")) != -1)
       switch (r) {
          case 'a':
             printf("Do option a!\n"); break;
          case 'b':
             printf("Do option b with %s!\n", optarg); break;
          case 'c':
             cval = atoi(optarg);
             printf("Do option c with %d!\n", cval);
             break;
          default:
             printf("Error: Unknown option!\n");
       }
}
```

Opening a file

```
#include <stdlib.h>
FILE *fopen(char *filename, char *mode);
```

- *filename* is the name of the file to be opened.
- *mode* is a “mode string” that specifies what operations we intend to perform on the file.
- returns a *file pointer* (or NULL on error):

```
FILE fp = fopen(“data.in", "r");
… // do something with the file
fclose(fp);
```

Modes

Mode strings for text files:

String	**Meaning**
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Reading from a file

```
char *fgets(char *buf, int n, FILE *stream)
```

- *buf* is the pointer to an array of `char`s where the string read is stored.
- *n* is the maximum number of characters to be read, including the null character.
- *stream*, is the pointer to a FILE object
- Example:

```
    FILE fp = fopen(“data.in", "r");
    char buf[10];
    fgets(buf, 10, fp);
```

Reading from a string

```
int sscanf(char *str, char *format, ...)
```

- *str* is the pointer to an array of `char`s where the string is stored.
- *format* is a format specifier
- Example:

```
    char s[] = “November 20, 2023";
    char mon[20];
    int day, yr;
    sscanf(s, “%s %d, %d”, mon, &day, &yr);
```

→ mon = “November”, day = 20, yr = 2023