Announcement

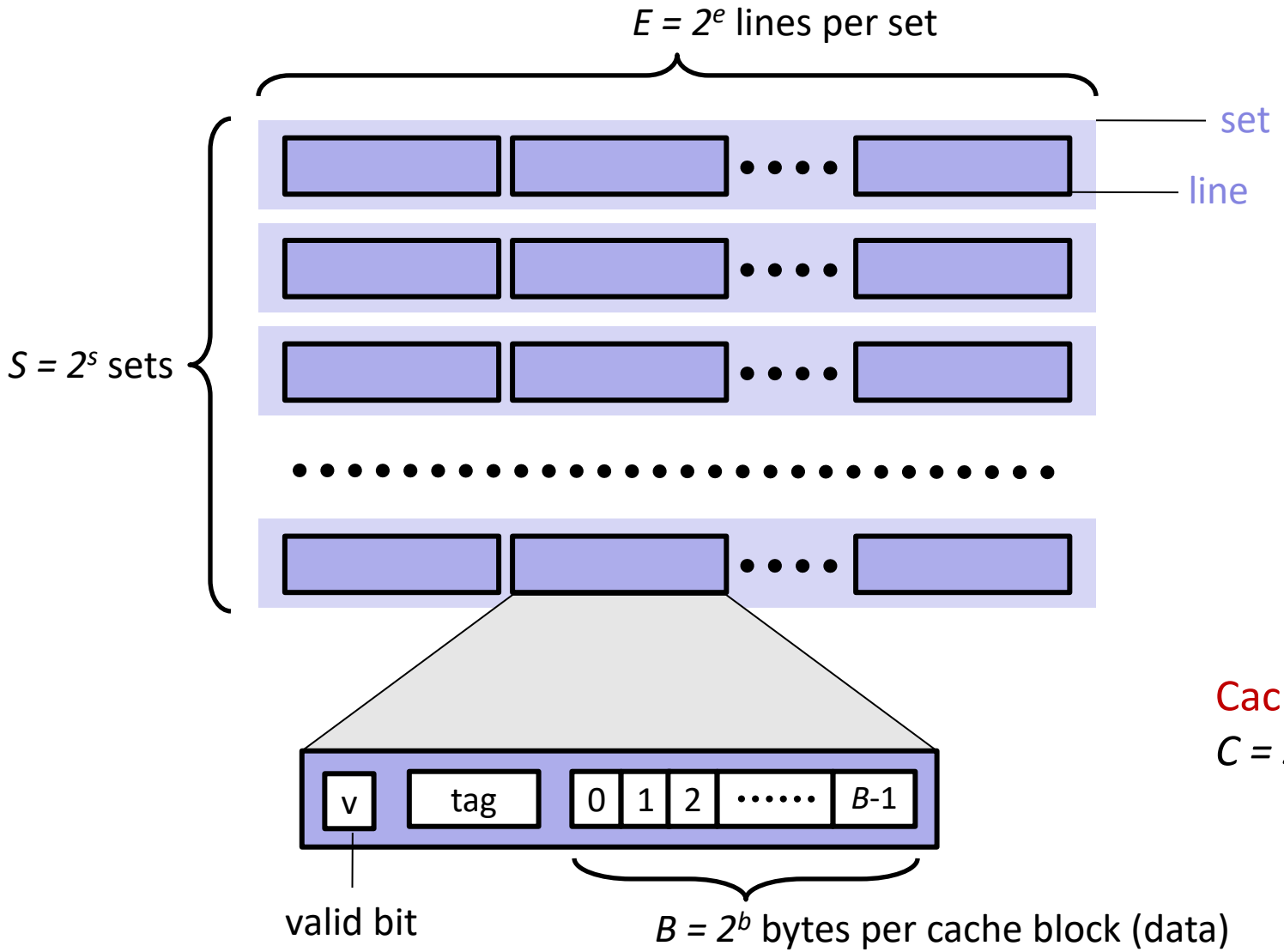
- Assignment 7
  - Due November 18
  - Cache simulator

Lecture 27

More on Cache

CPSC 275
Introduction to Computer Systems

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

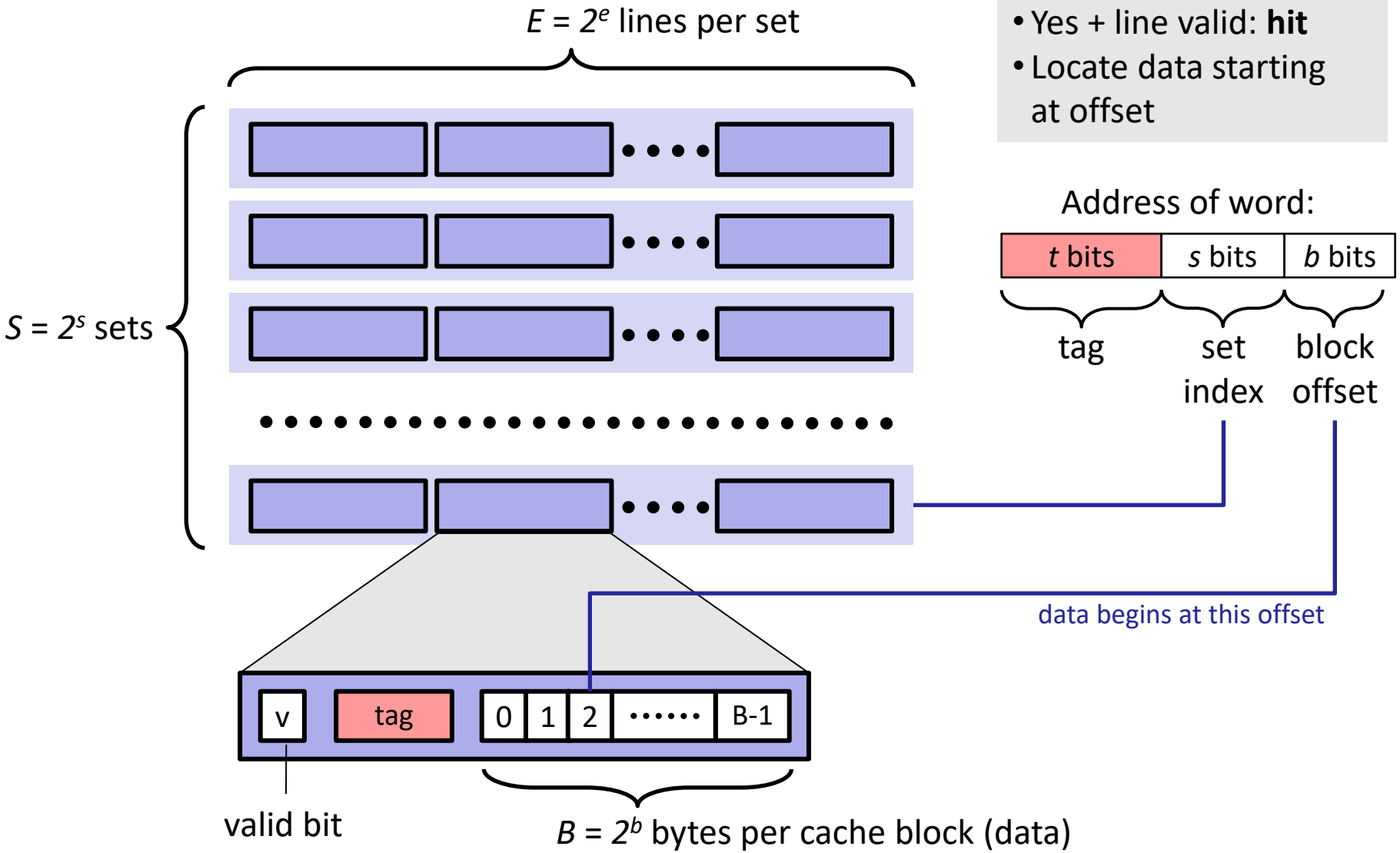


Cache size?

C = S x E x B bytes

Cache Read

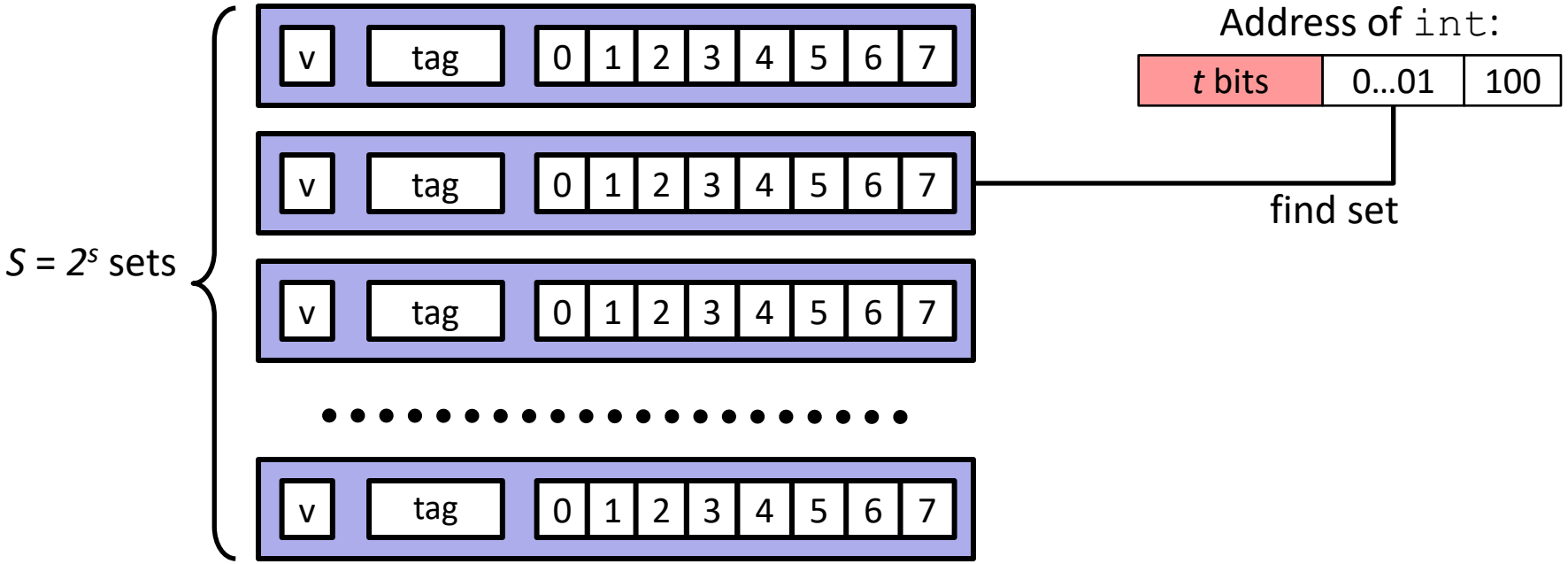
- Locate set
- Check if any line in set has matching tag
- Yes + line valid: **hit**
- Locate data starting at offset



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

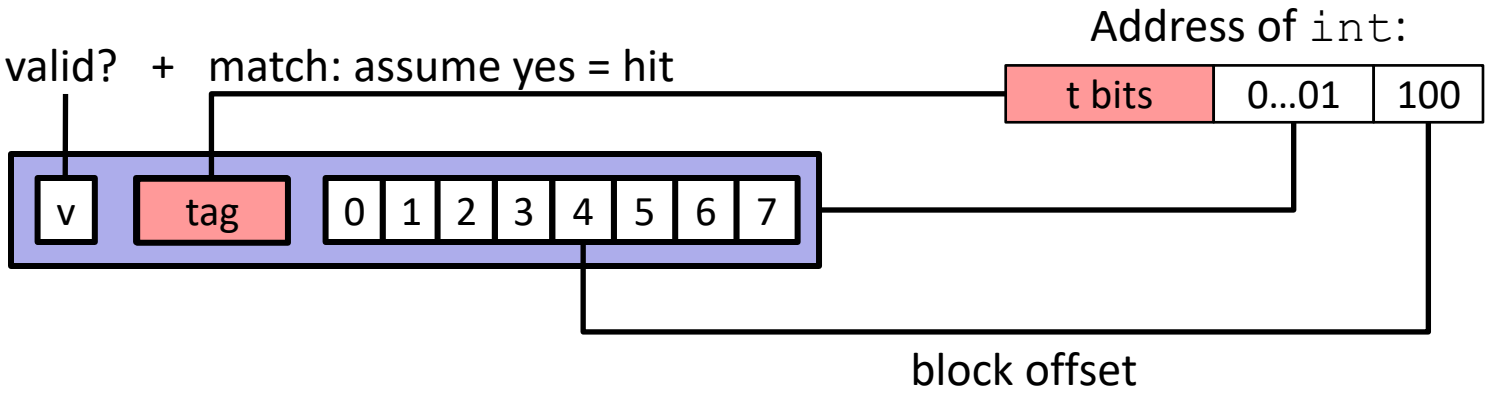
Assume: cache block size 8 bytes (*b* = ?)



Example: Direct Mapped Cache (E = 1)

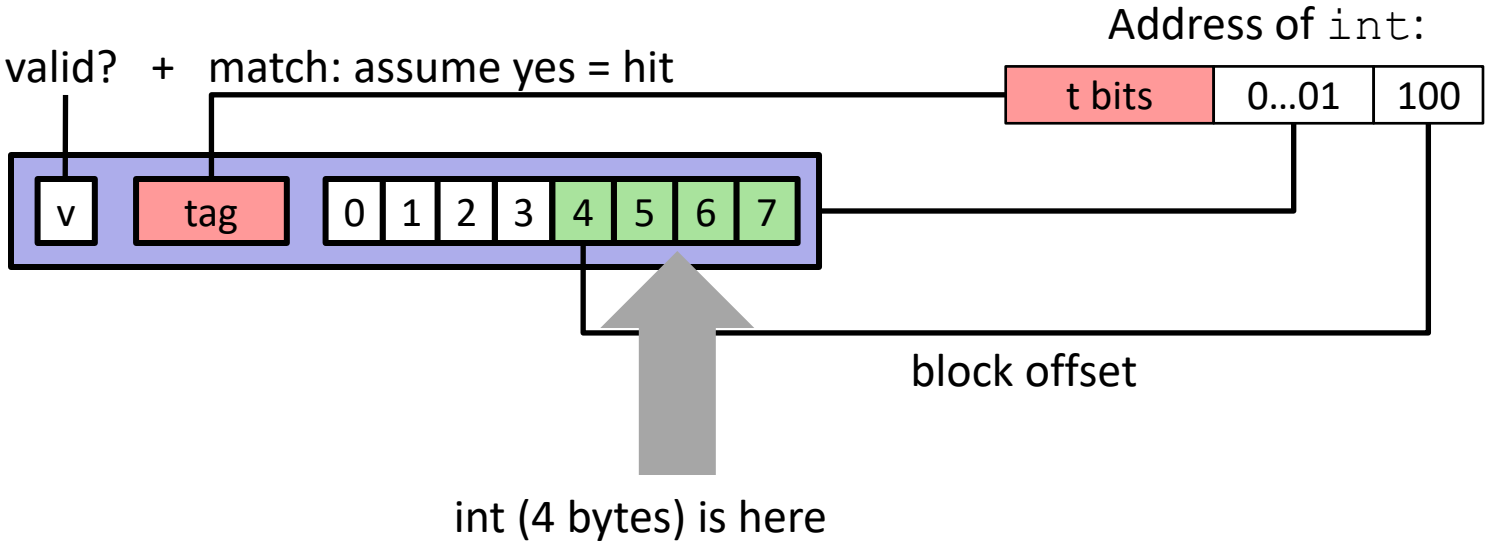
Direct mapped: One line per set

Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

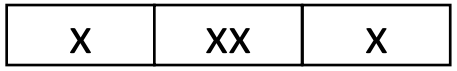
Direct mapped: One line per set
Assume: cache block size 8 bytes



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

Direct-Mapped Cache Simulation

4-bit address



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

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:

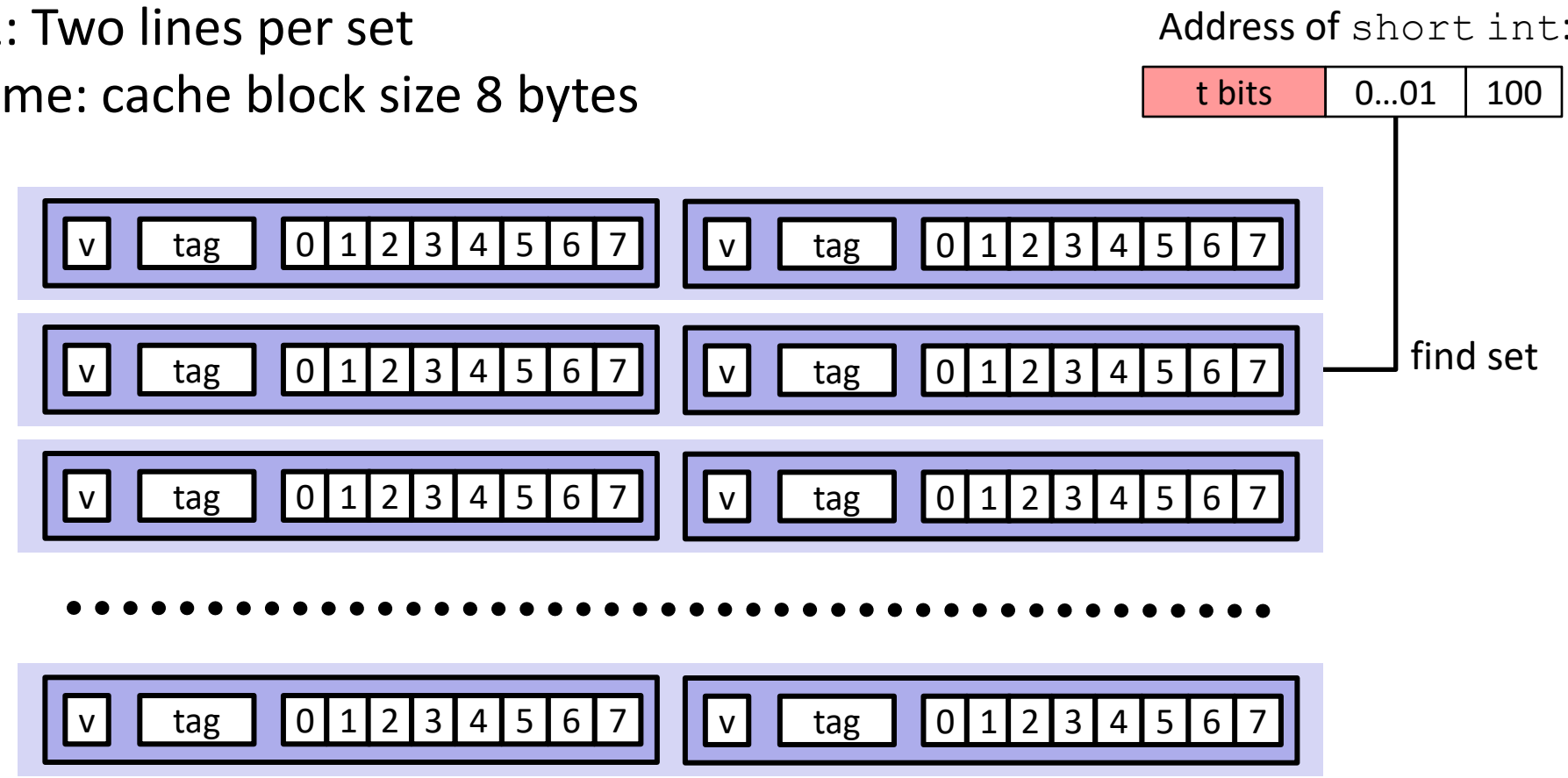
0	$[0\underline{00}0_2]$	miss
1	$[0\underline{00}1_2]$	hit
7	$[0\underline{11}1_2]$	miss
8	$[1\underline{00}0_2]$	miss
0	$[0\underline{00}0_2]$	miss

	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

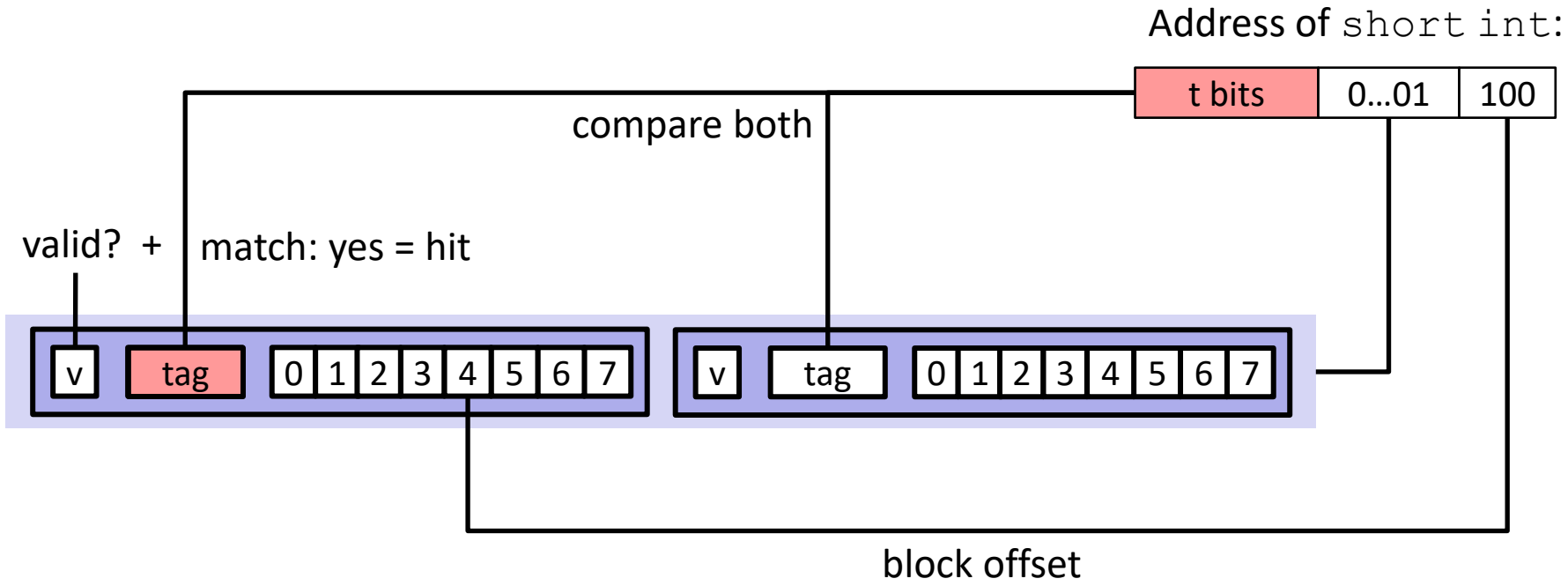
Assume: cache block size 8 bytes



2-way Set Associative Cache

E = 2: Two lines per set

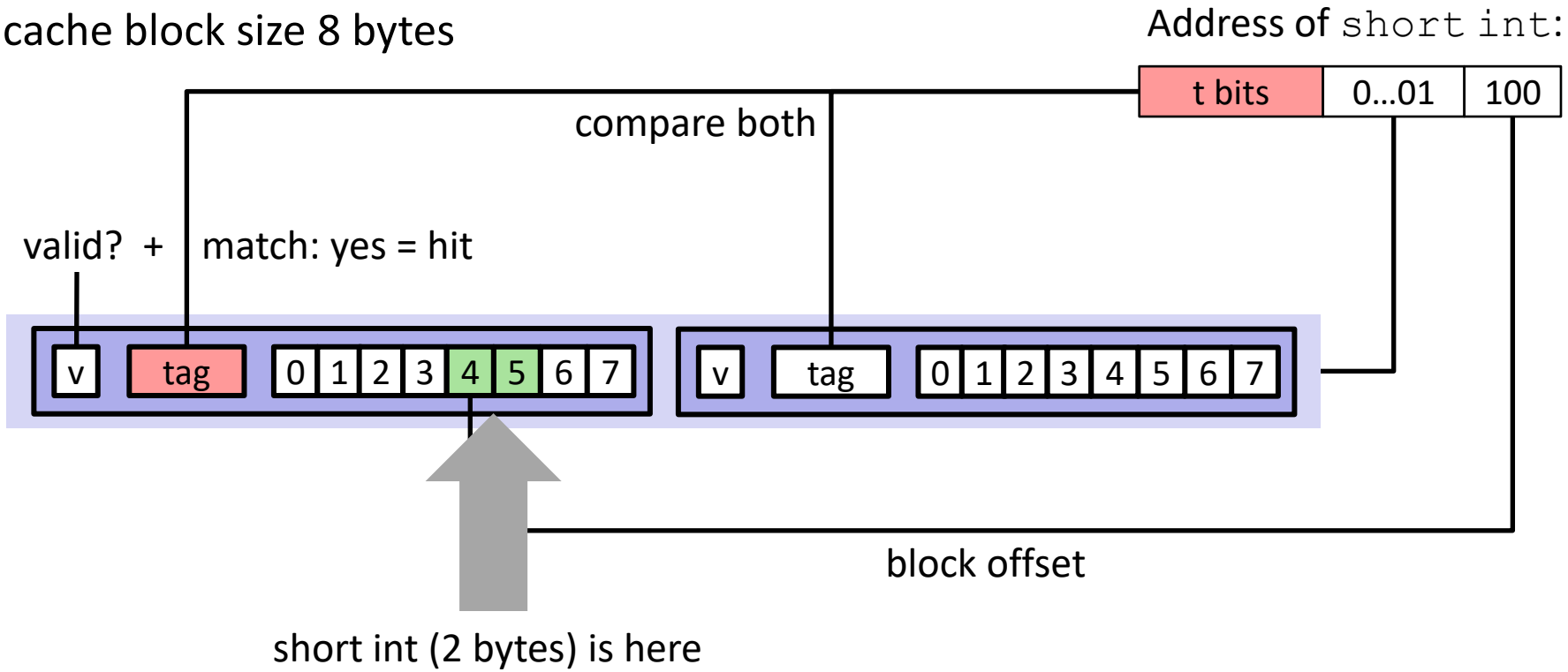
Assume: cache block size 8 bytes



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

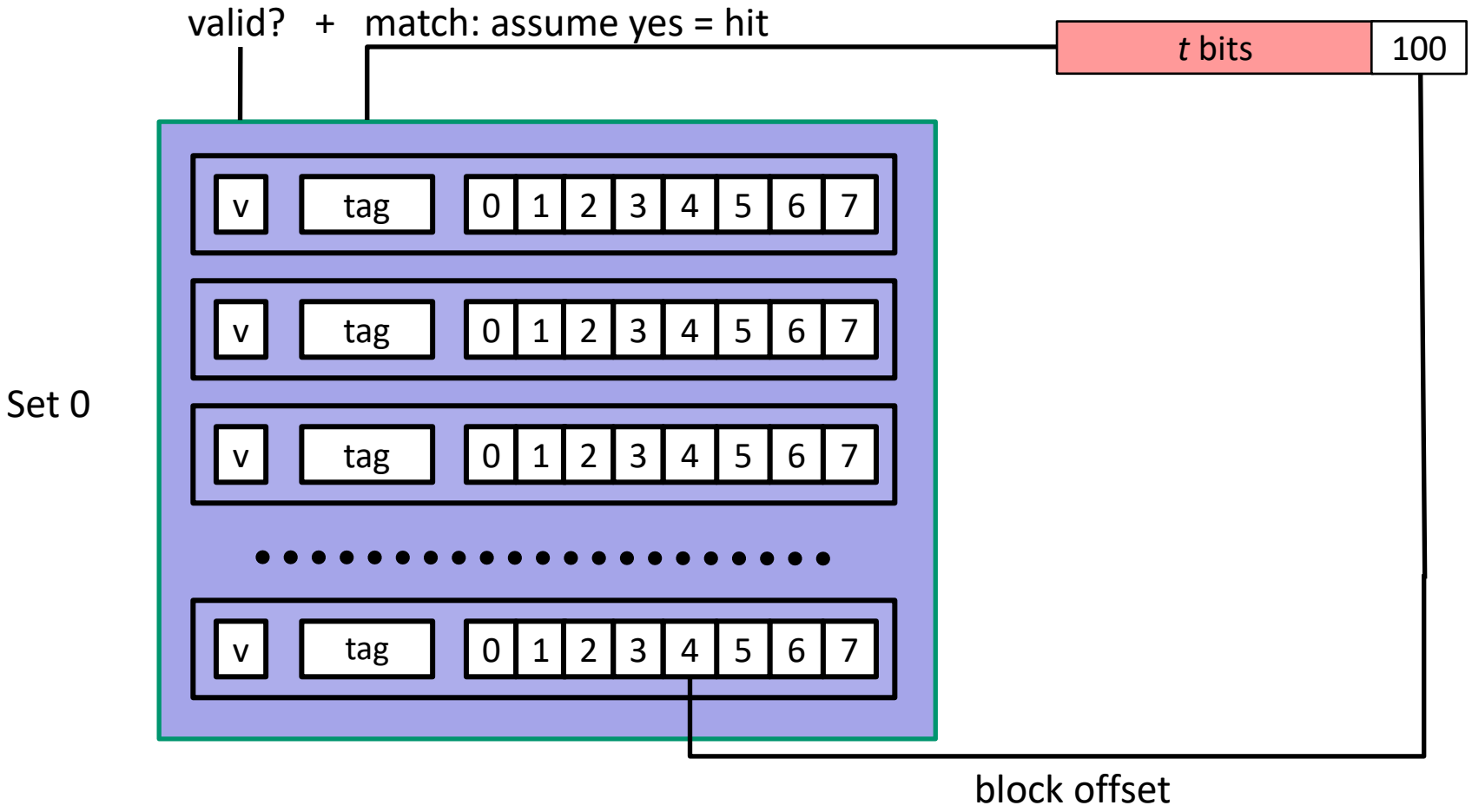
M=16 byte addresses,
B=2 bytes/block,
S=2 sets,
E=2 lines/set

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).

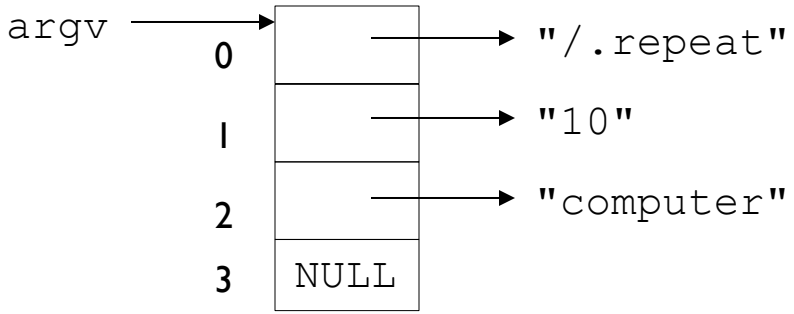
- `argv[0]` points to the name of the program,
- `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.

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:

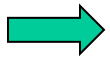
```
FILE fp = fopen("data.in", "r");
char buf[10];
fgets(buf, 10, fp);
```

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