Caches, Memory, and Memories

Chapter 6

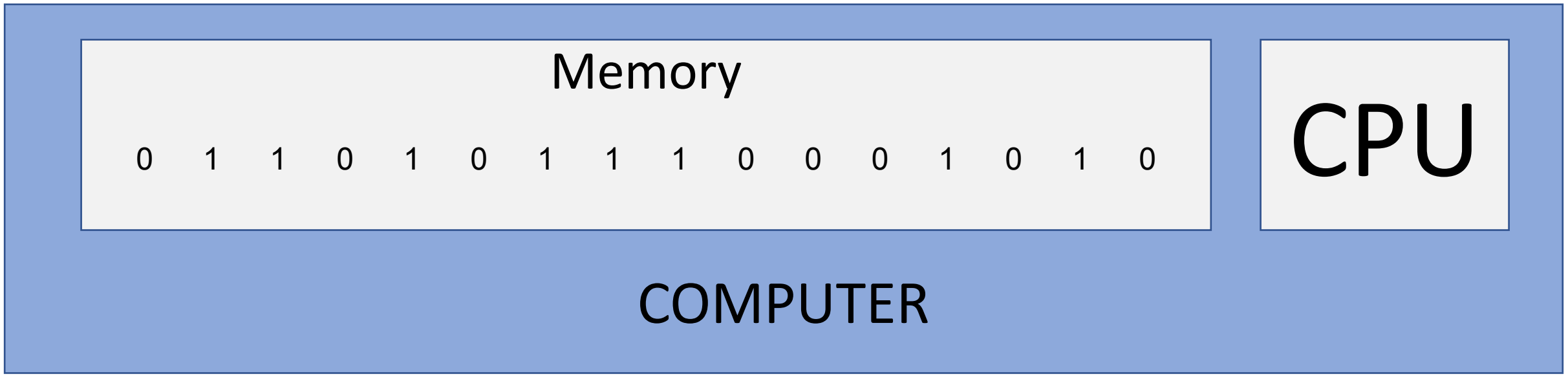
Which program is better?

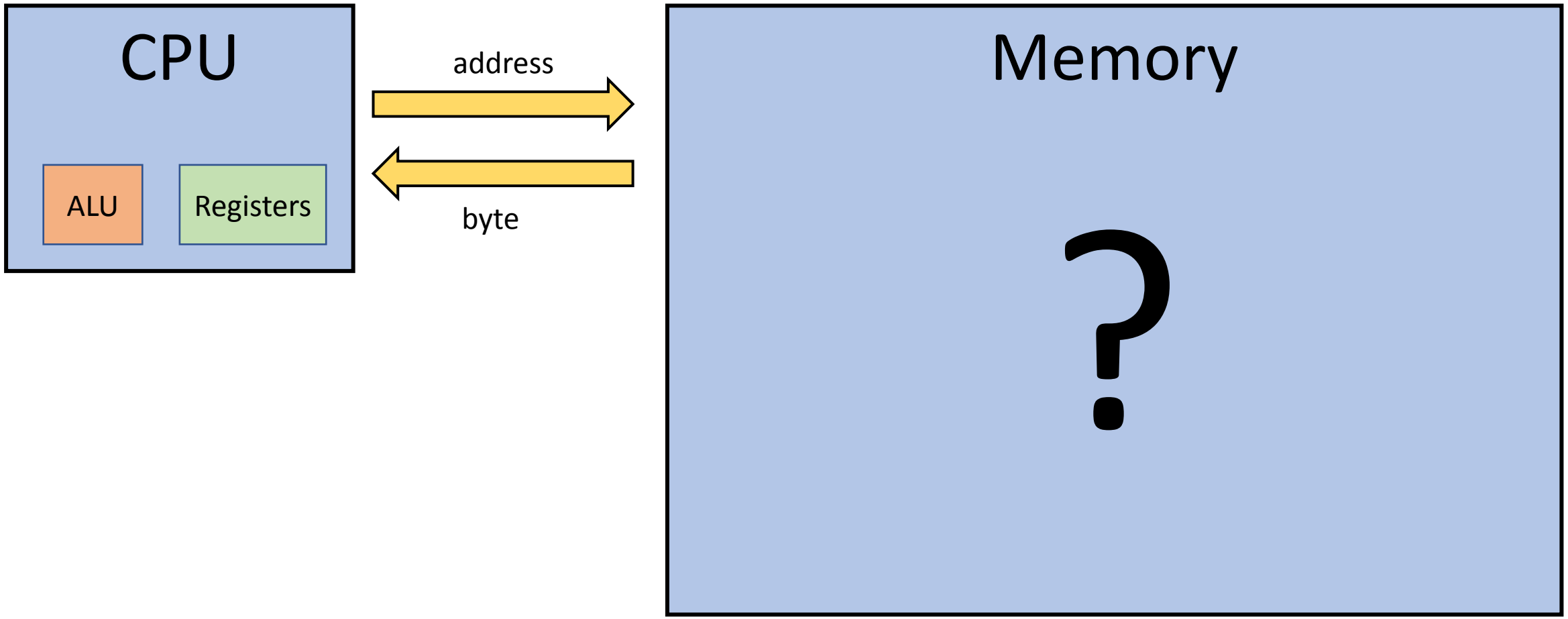
```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

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

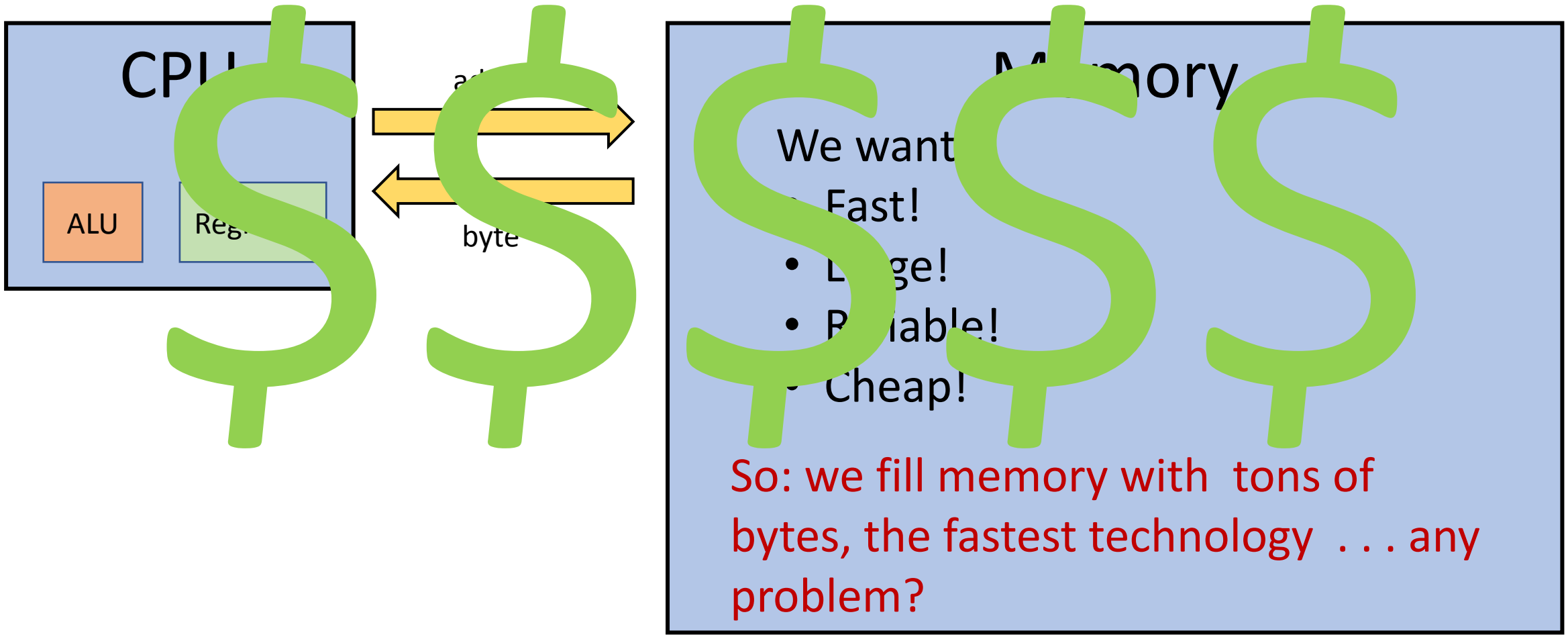
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

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

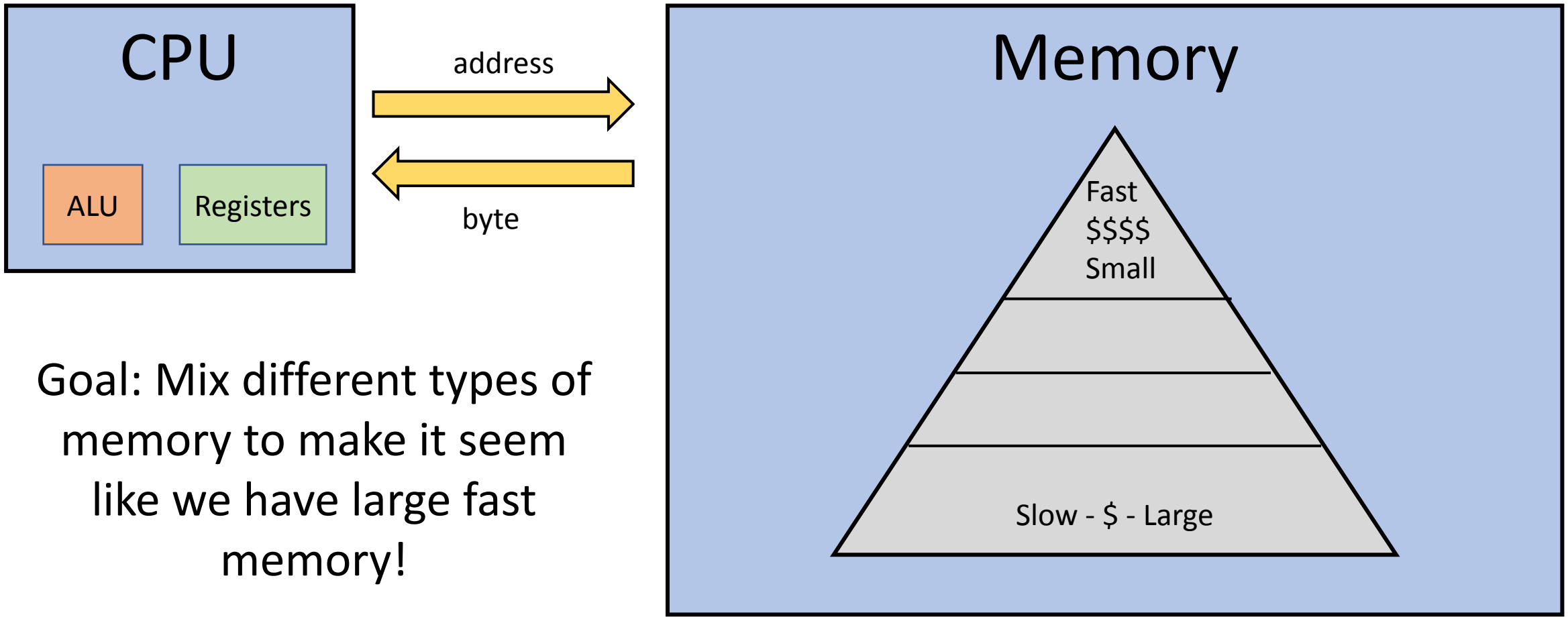
Would you believe one can be an order of magnitude faster?
Why?

Our abstraction




CPU
ALU
Registers
address
byte
Memory
?


CPU
ALU
byte
We want
Fast!
Cheap!
So: we fill memory with tons of bytes, the fastest technology . . . any problem?



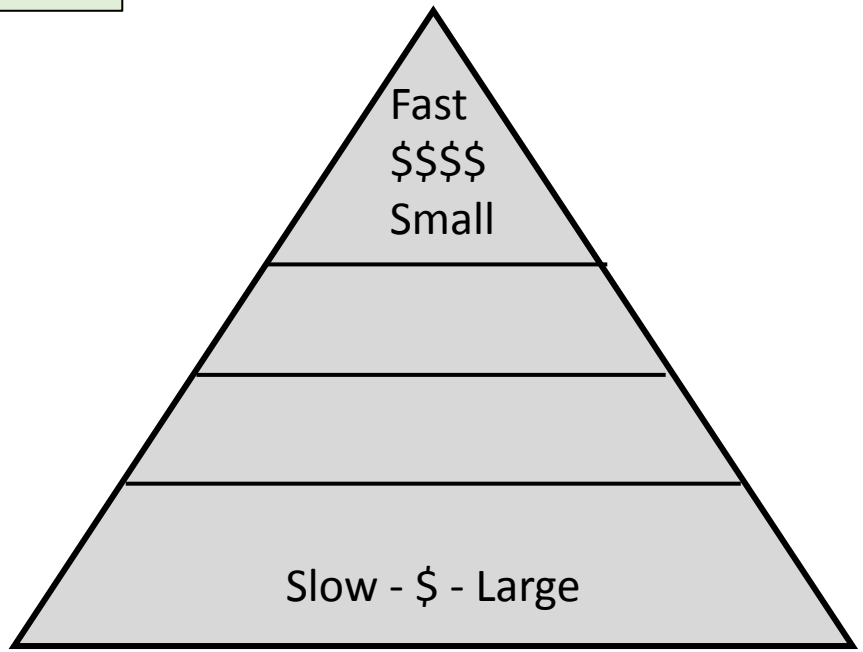
Goal: Mix different types of memory to make it seem like we have large fast memory!

Cache (pronounced “Cash”) (like the valley)

Cache = “a place for hiding, storing, or preserving treasure or supplies” – Merriam-Webster

Caching basics

- Each level serves as a cache for the next level down
- We store some of what that level has (but not all)
- The bottom level has “everything”



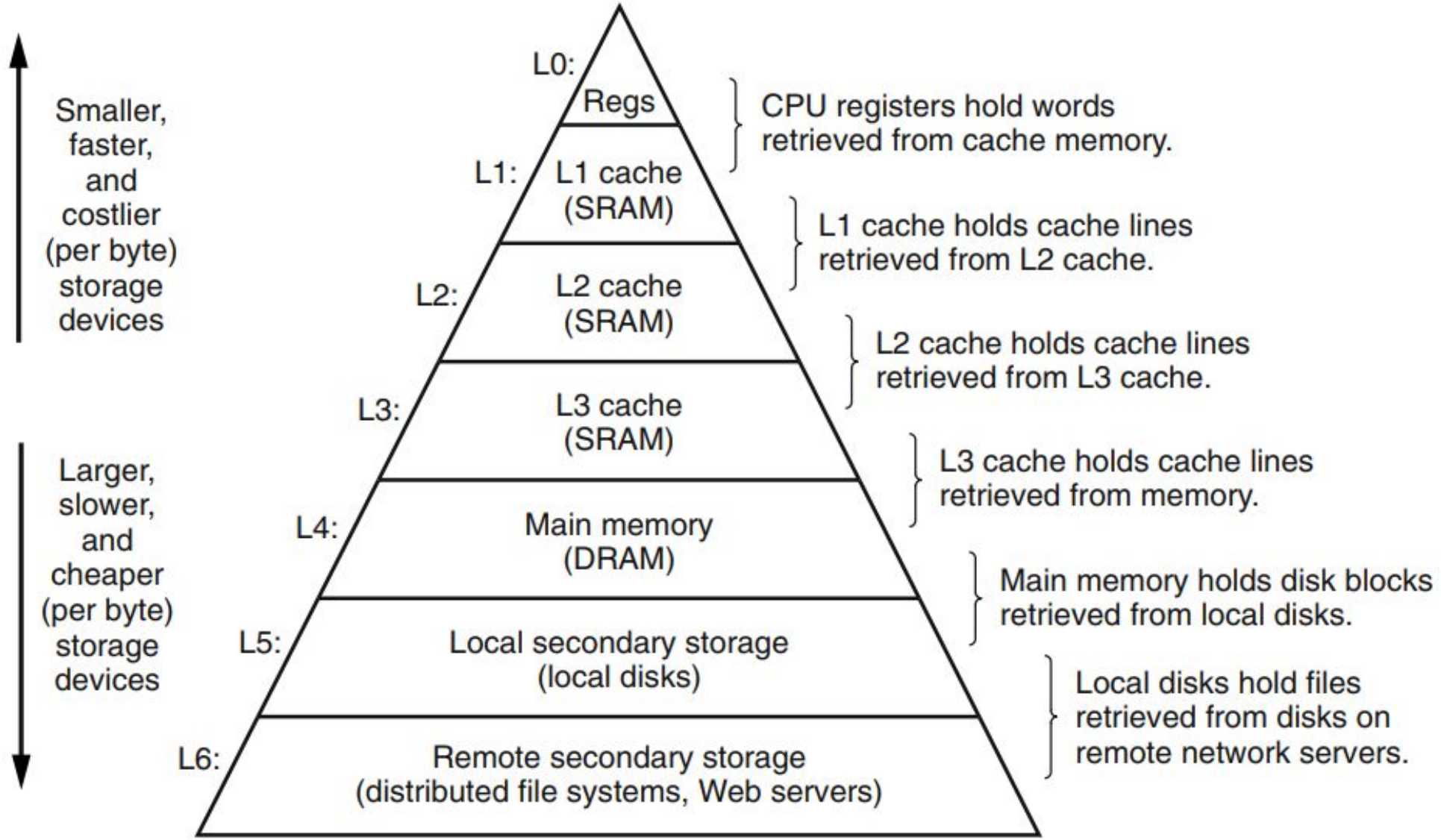


Figure 6.21 **The memory hierarchy.**

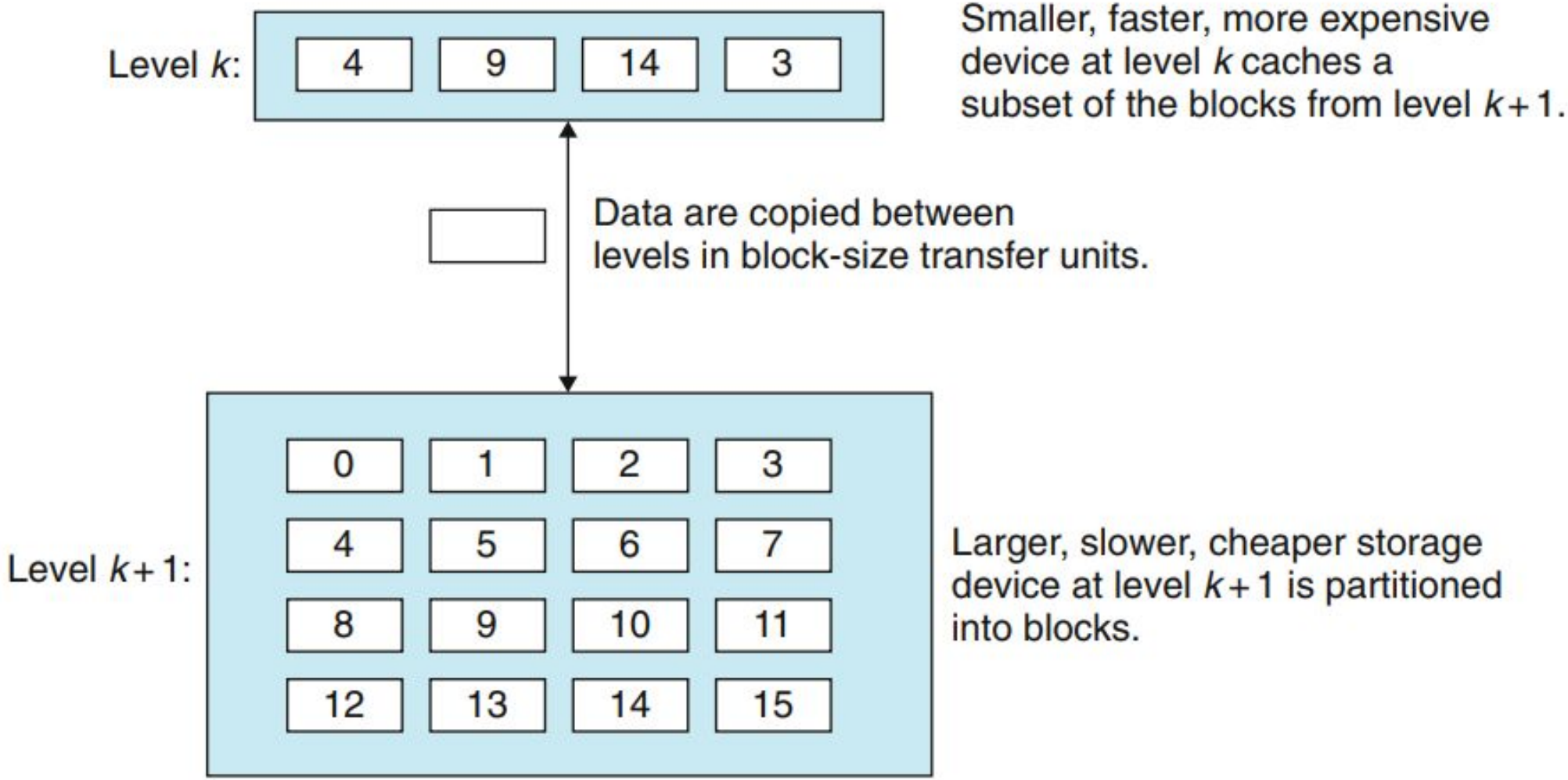


Figure 6.22 **The basic principle of caching in a memory hierarchy.**

Terminology

- Working set – all data blocks used by a program (or phase of it)
- Cache Hit – requested data is in the cache
- Cache Miss – requested data is *not* in the cache
  - Cold Cache – nothing is in the cache yet, so everything misses
  - Conflict Miss – requested data *could* fit in cache, but policy prevents it
  - Capacity Miss – all of the working set cannot fit in cache
- Cache replacement policy – which data block is removed to make room for new block?

Why does caching work? Locality

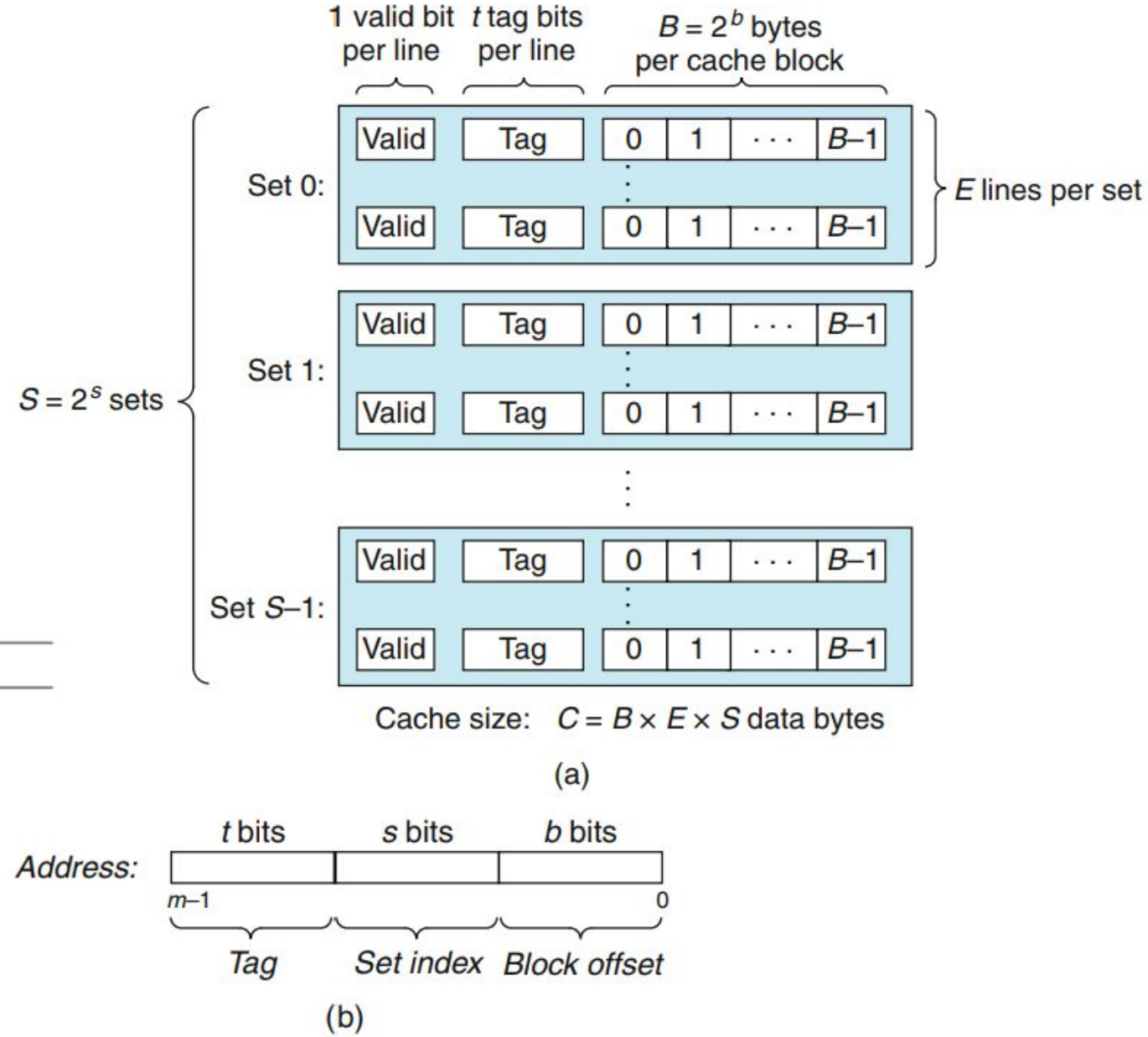
- Temporal locality – same data objects are likely to be reused multiple times
- Spatial locality – data blocks contain multiple data objects. When some data is referenced, likely to refer to near-by data as well.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	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

Figure 6.23 The ubiquity of caching in modern computer systems. Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.

Cache Organization

Figure 6.25
General organization of cache (S, E, B, m).
(a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Practice Problem 6.9 (solution page 663)

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

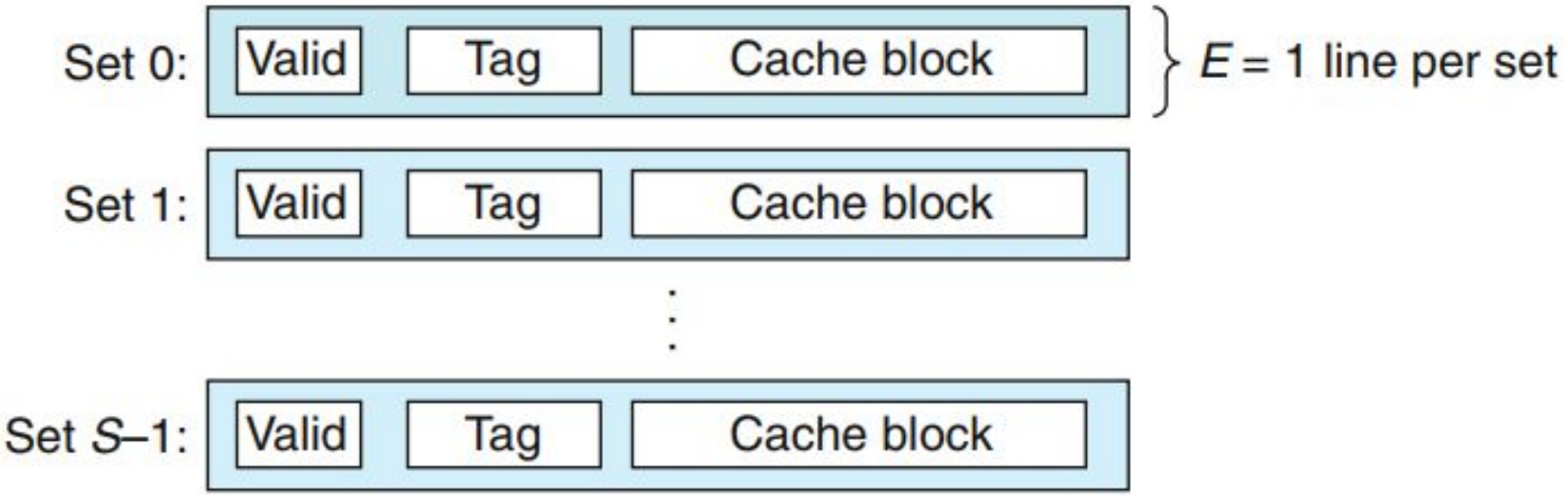
Cache	m	C	B	E	S	t	s	b
1.	32	1,024	4	1				
2.	32	1,024	8	4				
3.	32	1,024	32	32				

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Figure 6.26 Summary of cache parameters.

Direct-Mapped Caches (E=1)

Figure 6.27
Direct-mapped cache ($E = 1$). There is exactly one line per set.



Processing a request (Direct-mapped Cache)

1. Set selection – Which set in the cache would this block be in?
2. Line matching – only one line per set
    1. Does tag match?
    2. Is valid bit set?
    - Yes to both? Cache hit!
    - Otherwise – cache miss – get block from next level – replace current line
4. Word selection – use block offset to get the desired word

Example: Direct-Mapped Cache in Action

- DM-Cache
  - (S, E, B, m) = (4, 1, 2, 4)
    - 4 sets
    - 1 line per set
    - 2 bytes per block
    - 4 bit addresses
  - Assume word is single byte

- Tag + Index uniquely identifies block
- 8 blocks of memory, only 4 cache sets, 2 map to each
- Blocks in same set can be differentiated by the tag

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

1. Read word at address 0

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. Read word at address 1

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

3. Read word at address 13

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. Read word at address 8

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. Read word at address 0

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

THE CACHE

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

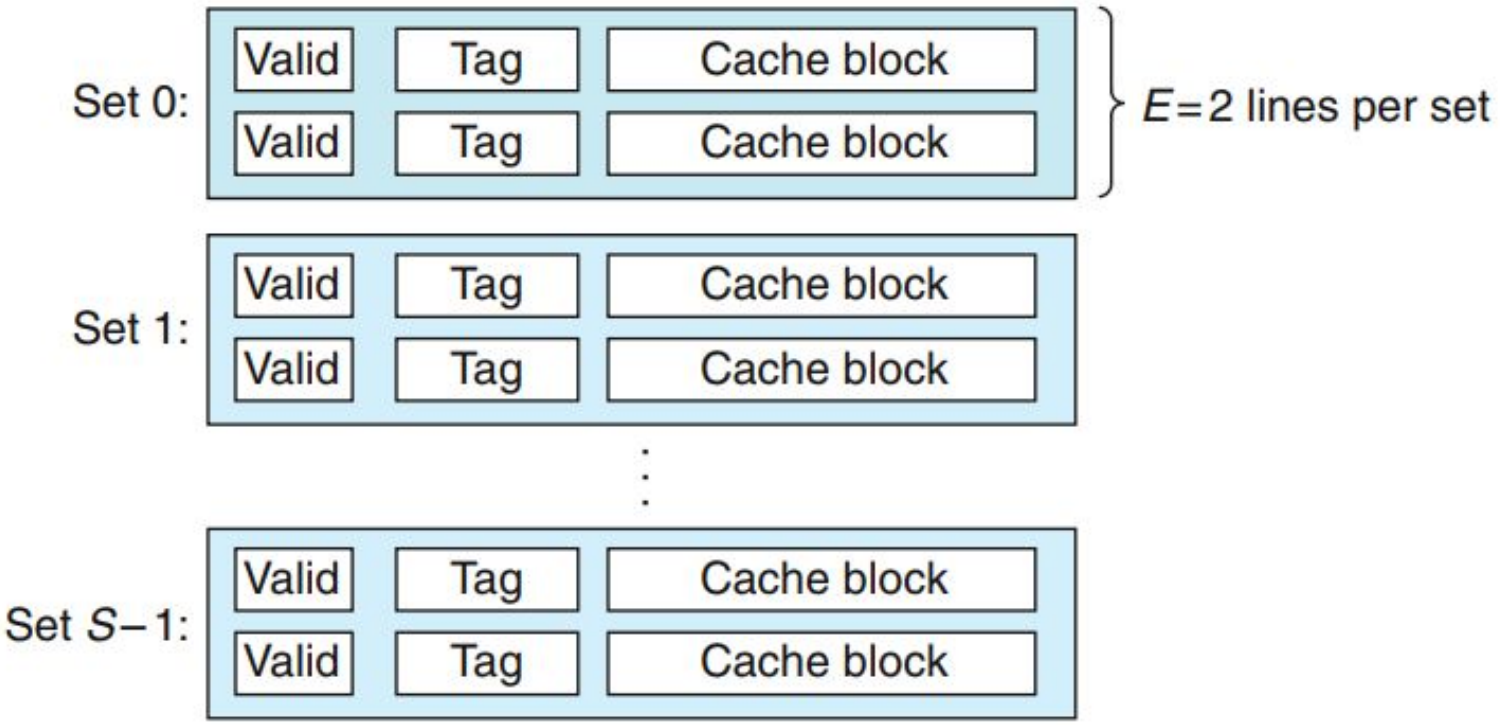
Final State of Cache

	Address bits			
Address (decimal)	Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

Set Associative Caches (1 < E < C/B)

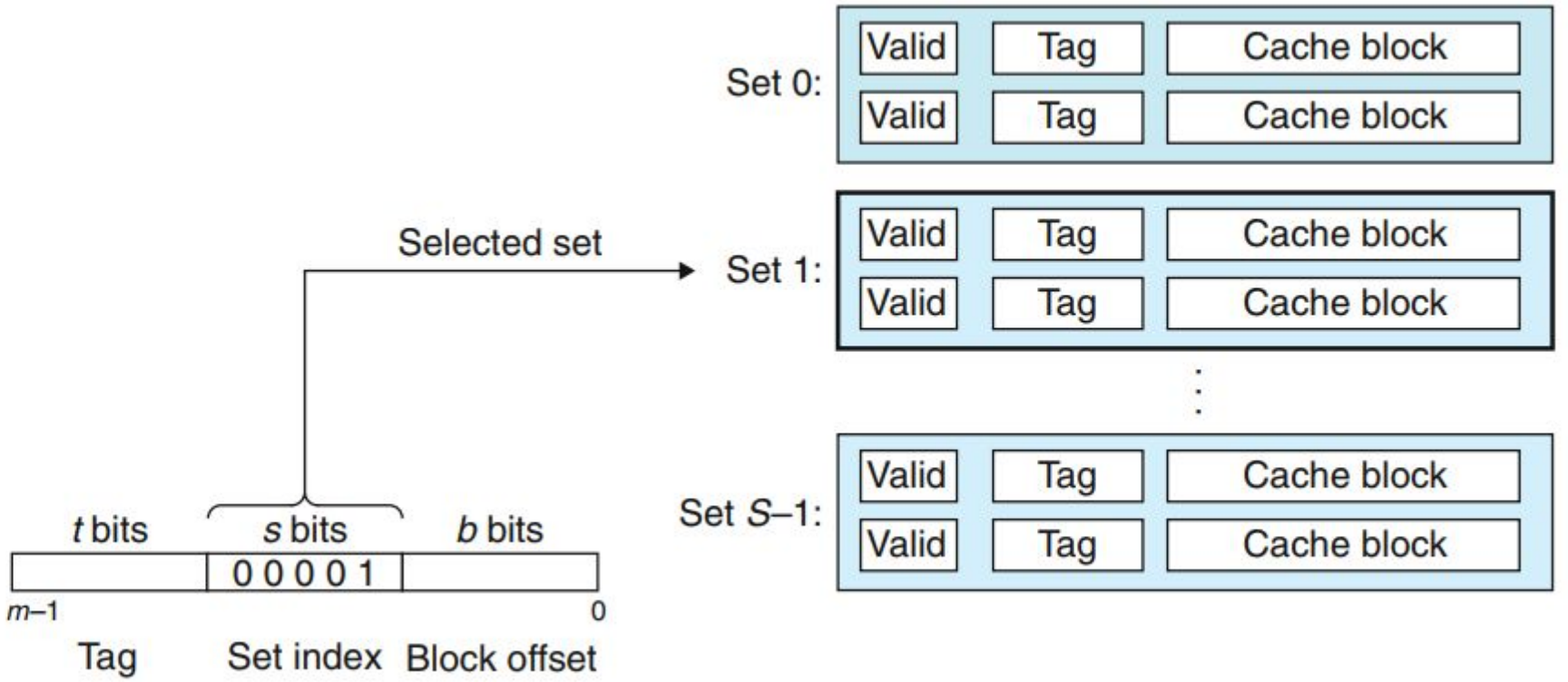
Figure 6.32
Set associative cache ($1 < E < C/B$). In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.



1. Set selection in Set Associative Cache

- Same as in Direct-Mapped Cache

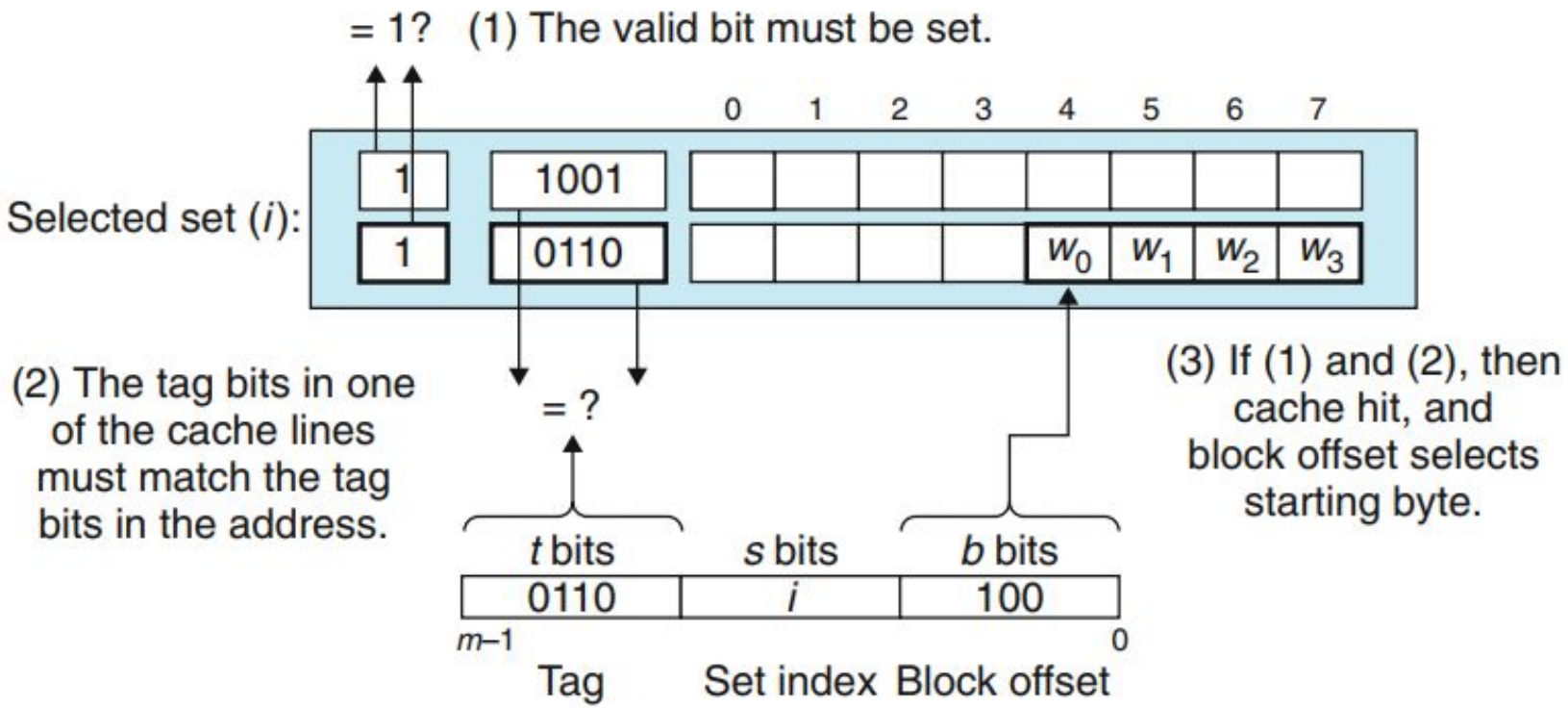
Figure 6.33
Set selection in a set associative cache.



2. Line Matching in Set Associative Caches

- Uses associative memory

Figure 6.34
Line matching and word selection in a set associative cache.



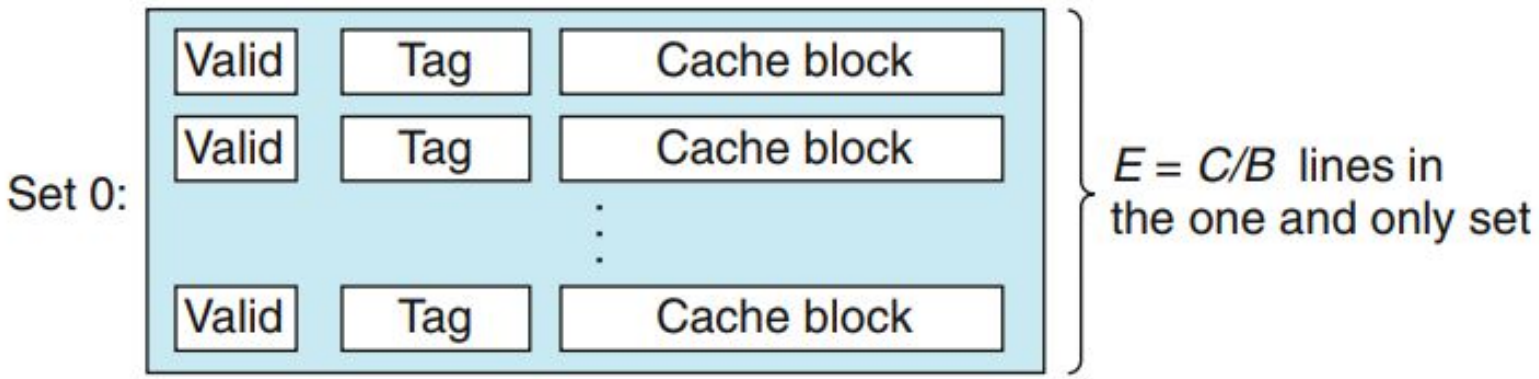
3. Replacement policy in Set Associative Cache

- More than one line in set, which do we evict?
  - Random line
  - Least Frequently Used (LFU) over recent time window
  - Least Recently Used (LRU) – accessed the farthest in the past

- Complicated policies require extra time and hardware, but are more worth it the further down the memory hierarchy we go, since a miss costs so much then.

Fully Associate Caches (S = 1 or E = C/B)

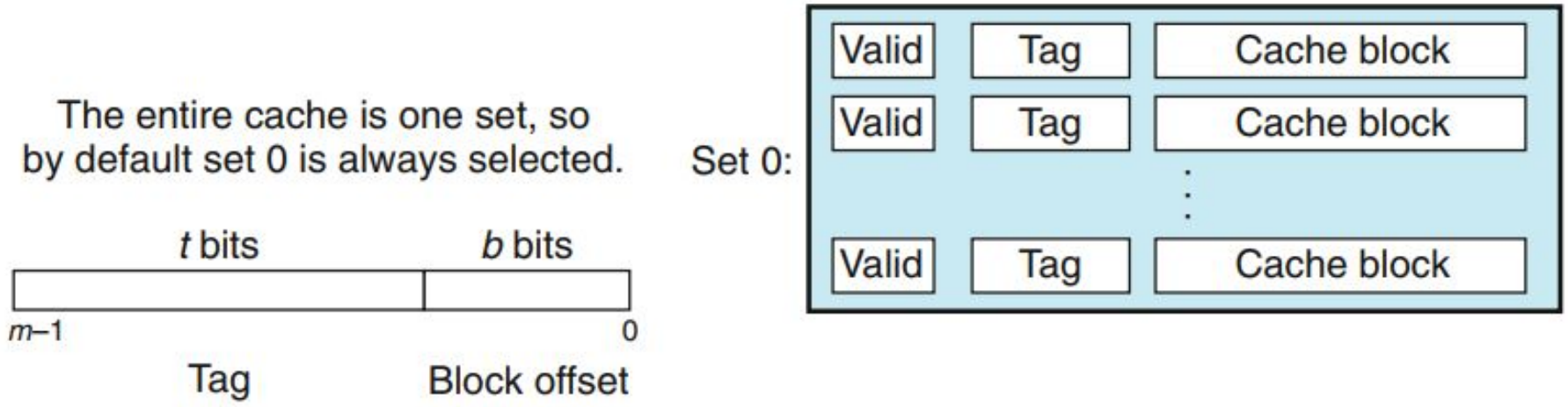
Figure 6.35
Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.



1. Set Selection in Fully Associative Cache

- Easy, only one set! No set index bits are used

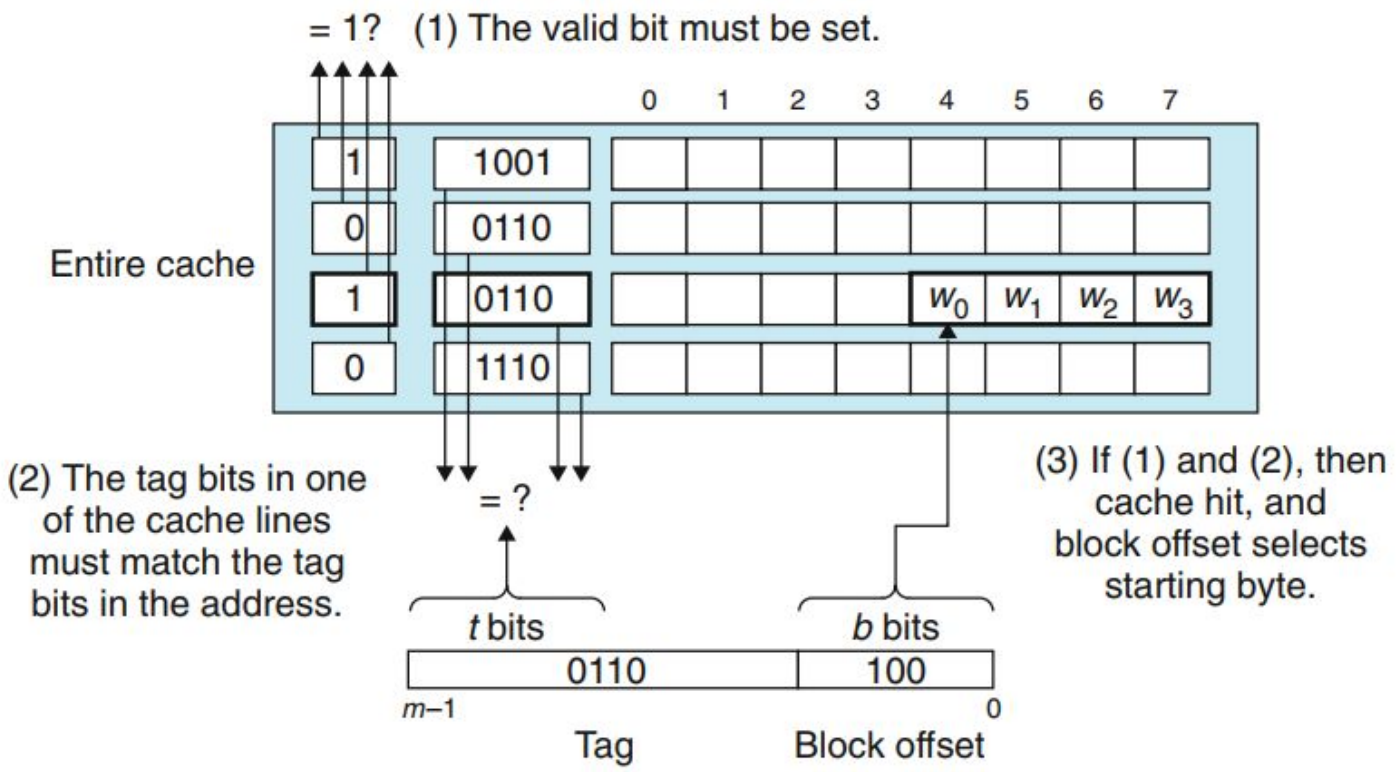
Figure 6.36
Set selection in a fully associative cache. Notice that there are no set index bits.



2. Line matching in Fully Associative Cache

- Same as Set Associative. Difference in scale

Figure 6.37
Line matching and word selection in a fully associative cache.



Practice Problems

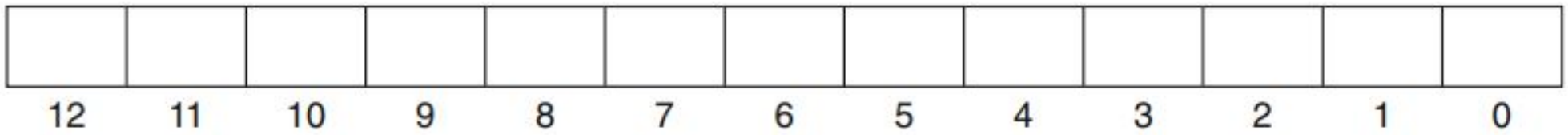
Assume: memory is byte addressable. Memory accesses are to 1-byte words. Addresses are 13 bits wide.
Cache: two-way set associative (E=2) with 4-byte block size (B = 4) and eight sets (S = 8)

The following figure shows the format of an address (1 bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO. The cache block offset

CI. The cache set index

CT. The cache tag



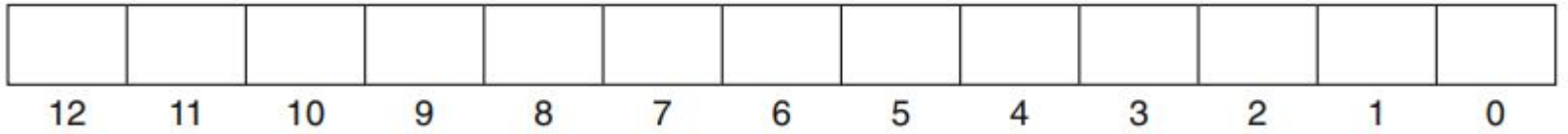
Assume: memory is byte addressable. Memory accesses are to 1-byte words. Addresses are 13 bits wide.
Cache: two-way set associative (E=2) with 4-byte block size (B = 4) and eight sets (S = 8)

2-way set associative cache

	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
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	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

Address: 0x0E34

- Cache block offset?
- Cache set index?
- Cache tag?
- Cache hit/miss?
- Cache byte returned



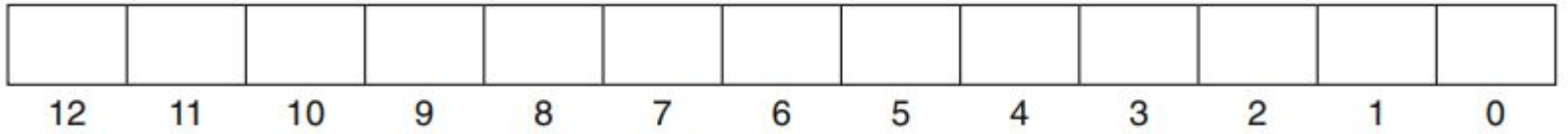
Assume: memory is byte addressable. Memory accesses are to 1-byte words. Addresses are 13 bits wide.
Cache: two-way set associative (E=2) with 4-byte block size (B = 4) and eight sets (S = 8)

2-way set associative cache

	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
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	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

Address: 0x0DD5

- Cache block offset?
- Cache set index?
- Cache tag?
- Cache hit/miss?
- Cache byte returned



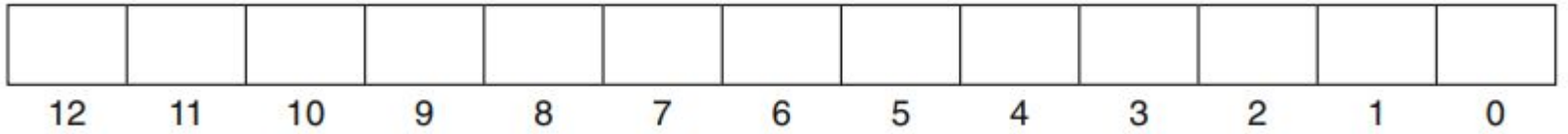
Assume: memory is byte addressable. Memory accesses are to 1-byte words. Addresses are 13 bits wide.
Cache: two-way set associative (E=2) with 4-byte block size (B = 4) and eight sets (S = 8)

2-way set associative cache

	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
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	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

Address: 0x1FE4

- Cache block offset?
- Cache set index?
- Cache tag?
- Cache hit/miss?
- Cache byte returned



Working with Caches

Writes? Write *w* to memory

- Write hit? When do we update the copy in lower levels of the hierarchy?
  1. Write through – immediately write *w*'s cache block to next lower level
     - Advantage? Simple
     - Disadvantage? Lots of traffic on the bus, move block for each write
  2. Write back – write *w*'s cache block out only when it is evicted by replacement policy
     - Advantage? Reduced bus traffic
     - Disadvantage? Increased complexity. Extra *dirty bit* required for each block
- Write Miss? Should we load the block into memory?
  1. Write-allocate – load block, then update
  2. No-write-allocate – don't load, just write to next level
- Typically:
  - Write through caches are no-write-allocate
  - Write back caches are write-allocate
- Book suggests assuming write-back, write-allocate for your mental model

Writing Cache-Friendly Code

1. *Make common case fast*
   - Most time spent in a few core functions, focus on inner loops, ignore the rest
2. *Minimize number of cache misses in each inner loop*
   - General rule: better miss rate = faster code

Practice Problem 6.18 (solution page 666)

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1,024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```
1   struct algae_position {
2       int x;
3       int y;
4   };
5
6   struct algae_position grid[16][16];
7   int total_x = 0, total_y = 0;
8   int i, j;
```

You should also assume the following:

- `sizeof(int)` $= 4$.
- `grid` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

1,024 DM Cache: B = 16, S = 64 | int = 4 Bytes, ap = 2 ints = 8 bytes | grid = 16x16 =256 aps = 512 ints = 2048 bytes

Determine the cache performance for the following code:

```
1       for (i = 0; i < 16; i++) {
2           for (j = 0; j < 16; j++) {
3               total_x += grid[i][j].x;
4           }
5       }
6
7       for (i = 0; i < 16; i++) {
8           for (j = 0; j < 16; j++) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

1,024 DM Cache: B = 16, S = 64 | int = 4 Bytes, ap = 2 ints = 8 bytes | grid = 16x16 =256 aps = 512 ints = 2048 bytes

Practice Problem 6.19 (solution page 666)

Given the assumptions of Practice Problem 6.18, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[j][i].x;
4               total_y += grid[j][i].y;
5           }
6       }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

D. What would the miss rate be if the cache were twice as big?

6.36 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
1       int x[2][128];
2       int i;
3       int sum = 0;
4
5       for (i = 0; i < 128; i++) {
6           sum += x[0][i] * x[1][i];
7       }
```

Assume we execute this under the following conditions:

- `sizeof(int)` = 4.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

A. Case 1: Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate?

6.36 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
1       int x[2][128];
2       int i;
3       int sum = 0;
4
5       for (i = 0; i < 128; i++) {
6           sum += x[0][i] * x[1][i];
7       }
```

Assume we execute this under the following conditions:

- `sizeof(int)` = 4.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

B. Case 2: What is the miss rate if we double the cache size to 1,024 bytes?

6.36 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
1       int x[2][128];
2       int i;
3       int sum = 0;
4
5       for (i = 0; i < 128; i++) {
6           sum += x[0][i] * x[1][i];
7       }
```

Assume we execute this under the following conditions:

- `sizeof(int)` = 4.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?

6.36 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
1       int x[2][128];
2       int i;
3       int sum = 0;
4
5       for (i = 0; i < 128; i++) {
6           sum += x[0][i] * x[1][i];
7       }
```

Assume we execute this under the following conditions:

- `sizeof(int)` = 4.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?

D. For case 3, will a larger cache size help to reduce the miss rate? Why or why not?

E. For case 3, will a larger block size help to reduce the miss rate? Why or why not?

Conclusions

Congratulations! You have learned a lot!

- Bits/Bytes/Memory
- chars/ints/longs
- strings/arrays
- Linux, C, Y86-64, X86-86, gdb
- Debugging, security, caches

- Your understand helps you have a better picture of what is “under the hood”
- This will make you a better programmer
- I also hope you have had fun!