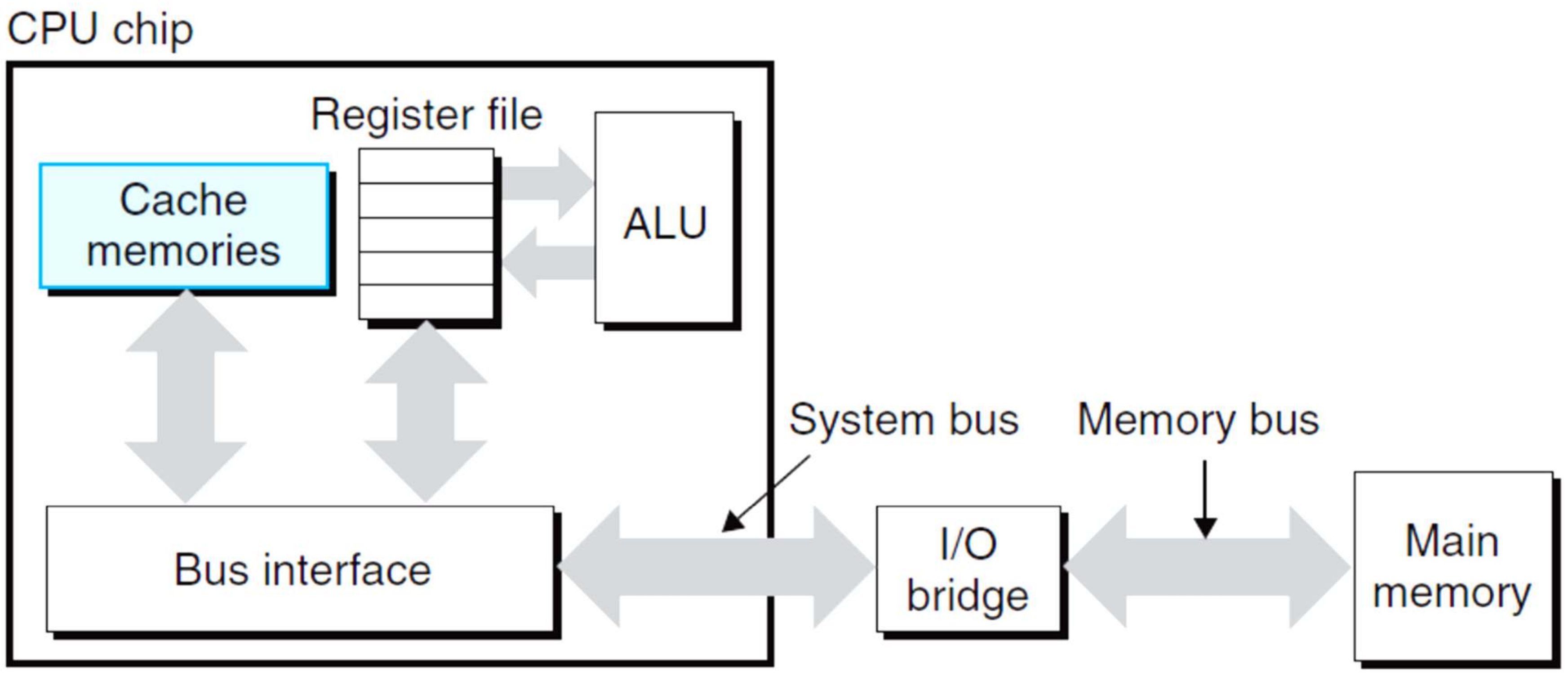
Typical bus structure for cache memories.

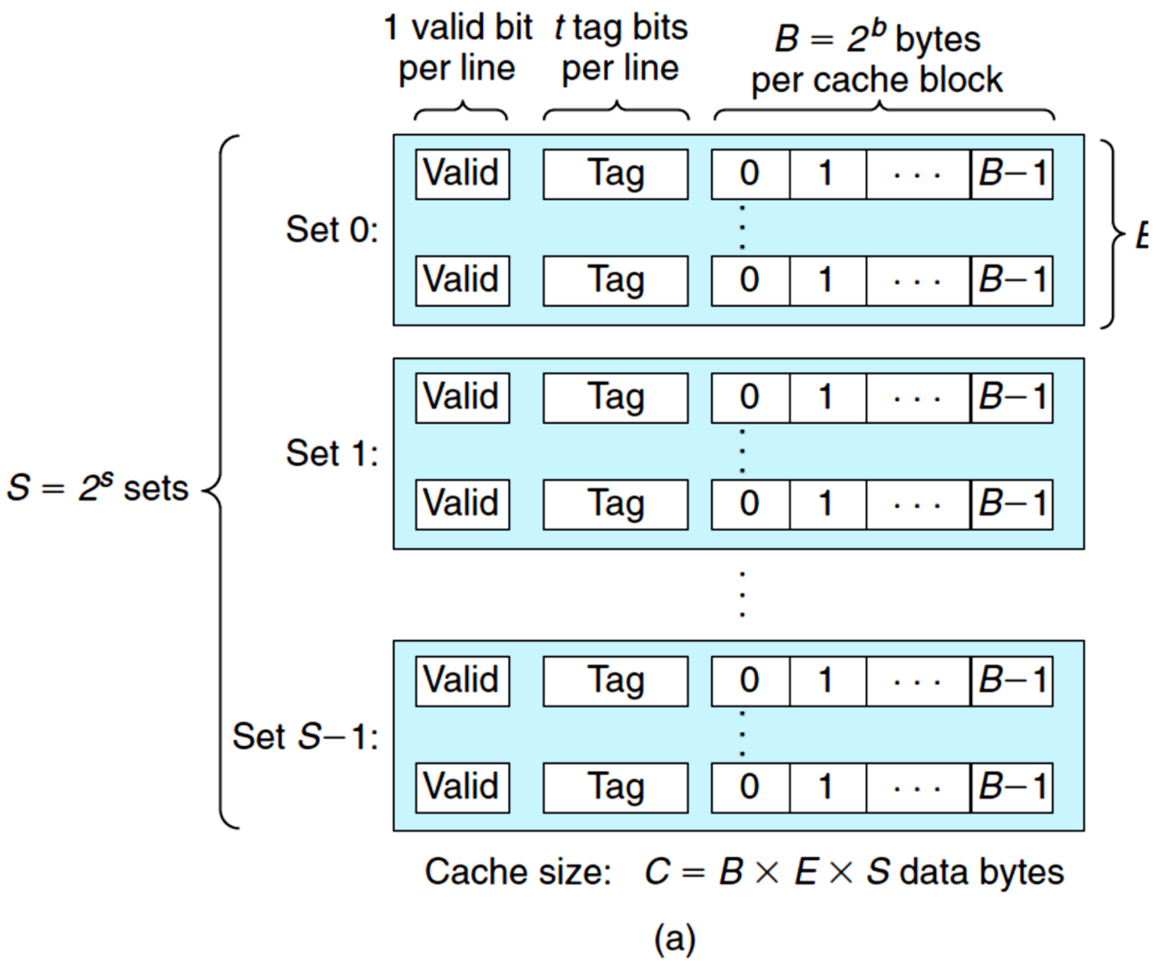


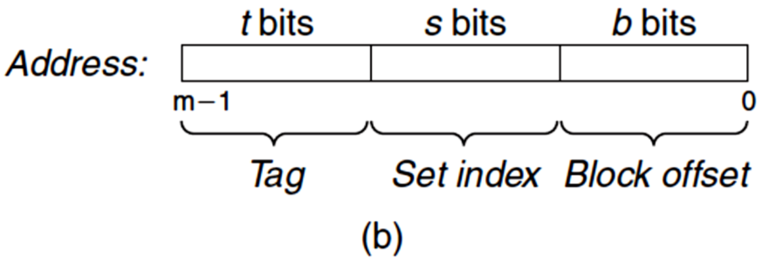
- The memory hierarchies of early computer systems consisted of only three levels:
  - CPU registers
  - Main DRAM memory
  - Disk storage
- However, because of the increasing gap between CPU and main memory, system designers were compelled to insert a small SRAM*cache memory*, called an *L1 cache* (Level 1 cache) between the CPU register file and main memory

General organization of cache

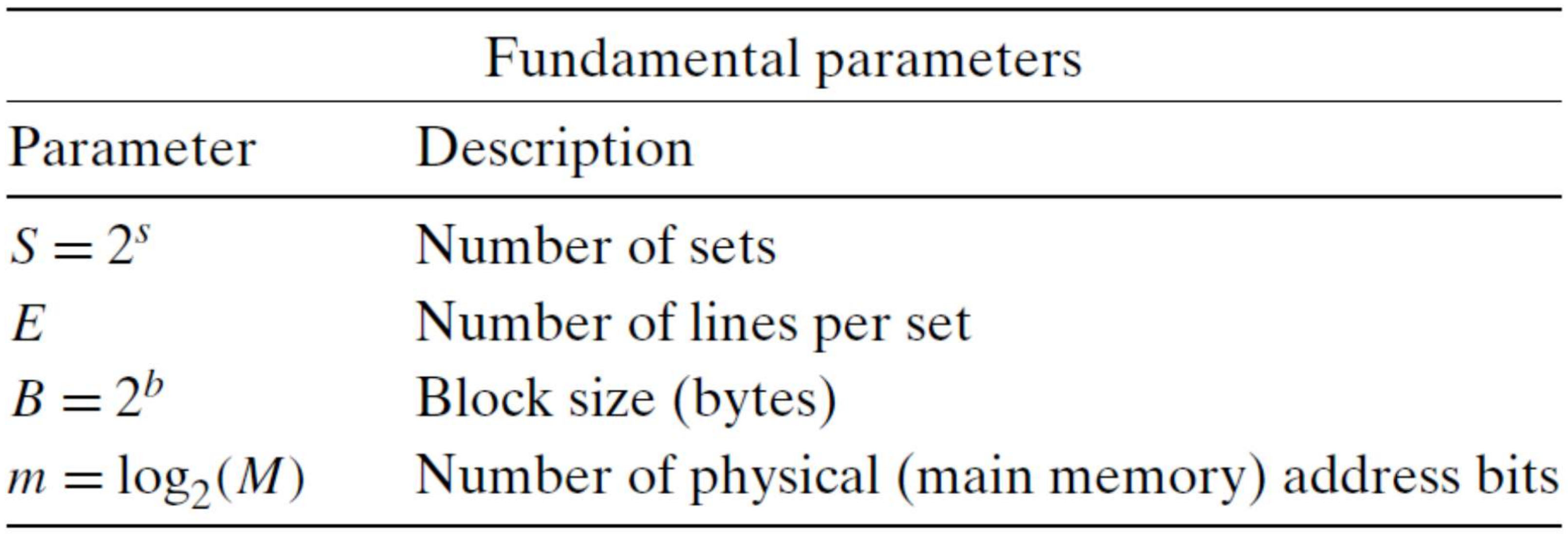
(a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data.

(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.





The cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function. Here is how it works:
The parameters *S* and *B* induce a partitioning of the *m* address bits into the three

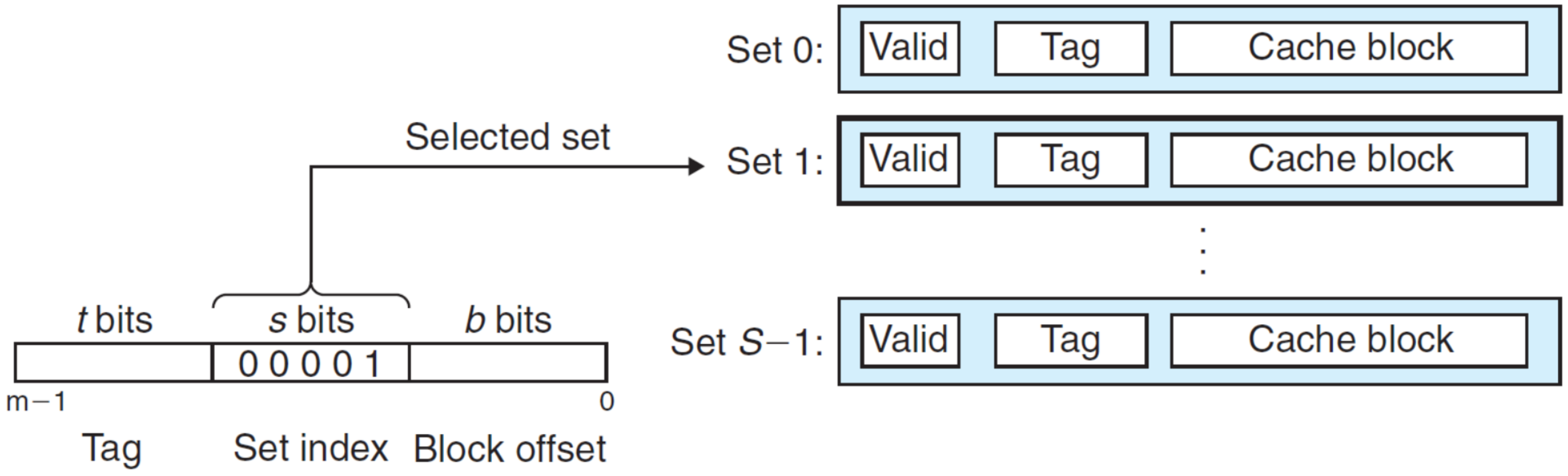


Fundamental parameters

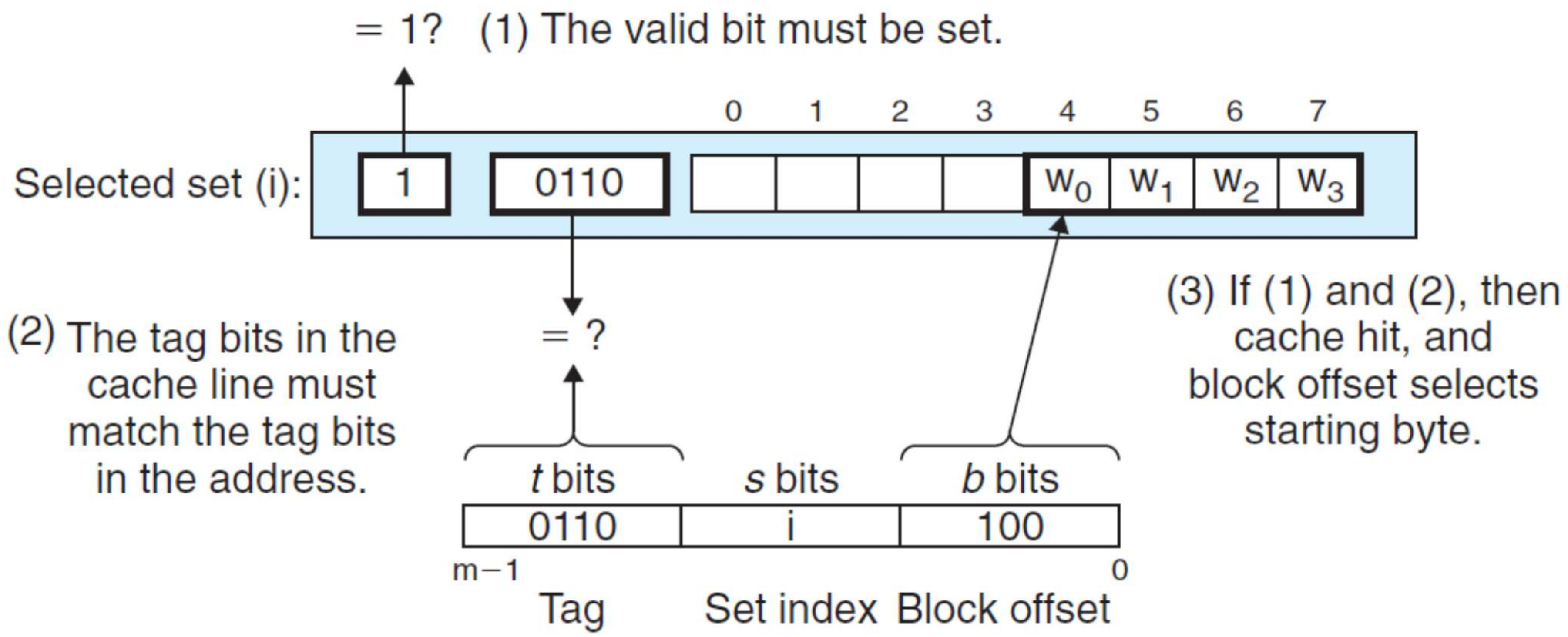
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

- When the CPU is instructed by a load instruction to read a word from address *A* of main memory, it sends the address *A* to the cache. If the cache is holding a copy of the word at address *A*, it sends the word immediately back to the CPU.
- how does the cache know whether it contains a copy of the word at address *A*?

- **Set selection in a direct-mapped cache.**



- **Line matching and word selection in a directmapped cache**


= 1? (1) The valid bit must be set.
0 1 2 3 4 5 6 7
Selected set (i):
1
0110
w_0 w_1 w_2 w_3
= ?
(2) The tag bits in the cache line must match the tag bits in the address.
(3) If (1) and (2), then cache hit, and block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m−1
0
Tag
Set index
Block offset

A concrete example will help clarify the process

- Suppose we have a direct-mapped cache described by:

  $(S, E, B, m) = (4, 1, 2, 4)$

- Initially, the cache is empty (i.e., each valid bit is zero):

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

- **1. Read word at address 0. (**cache miss)

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

- **Read word at address 1.** This is a cache hit
- **Read word at address 13**

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			

- **Read word at address 8.**

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			

- **Read word at address 0.**

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			

Conflict Misses in Direct-Mapped Caches

- Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2
  - For example, consider a function that computes the dot product of two vectors:

```
float dotprod(float x[8], float y[8])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i] * y[i];
    return sum;
}
```

- This function has good spatial locality with respect to x and y, and so we might expect it to enjoy a good number of cache hits. Unfortunately, this is not always true.

- Suppose that x is loaded into the 32 bytes of contiguous memory starting at address 0, and that y starts immediately after x at address 32
- For simplicity, suppose that a block is 16 bytes and that the cache consists of two sets, for a total cache size of 32 bytes.
- We will assume that the variable sum is actually stored in a CPU register and thus does not require a memory reference

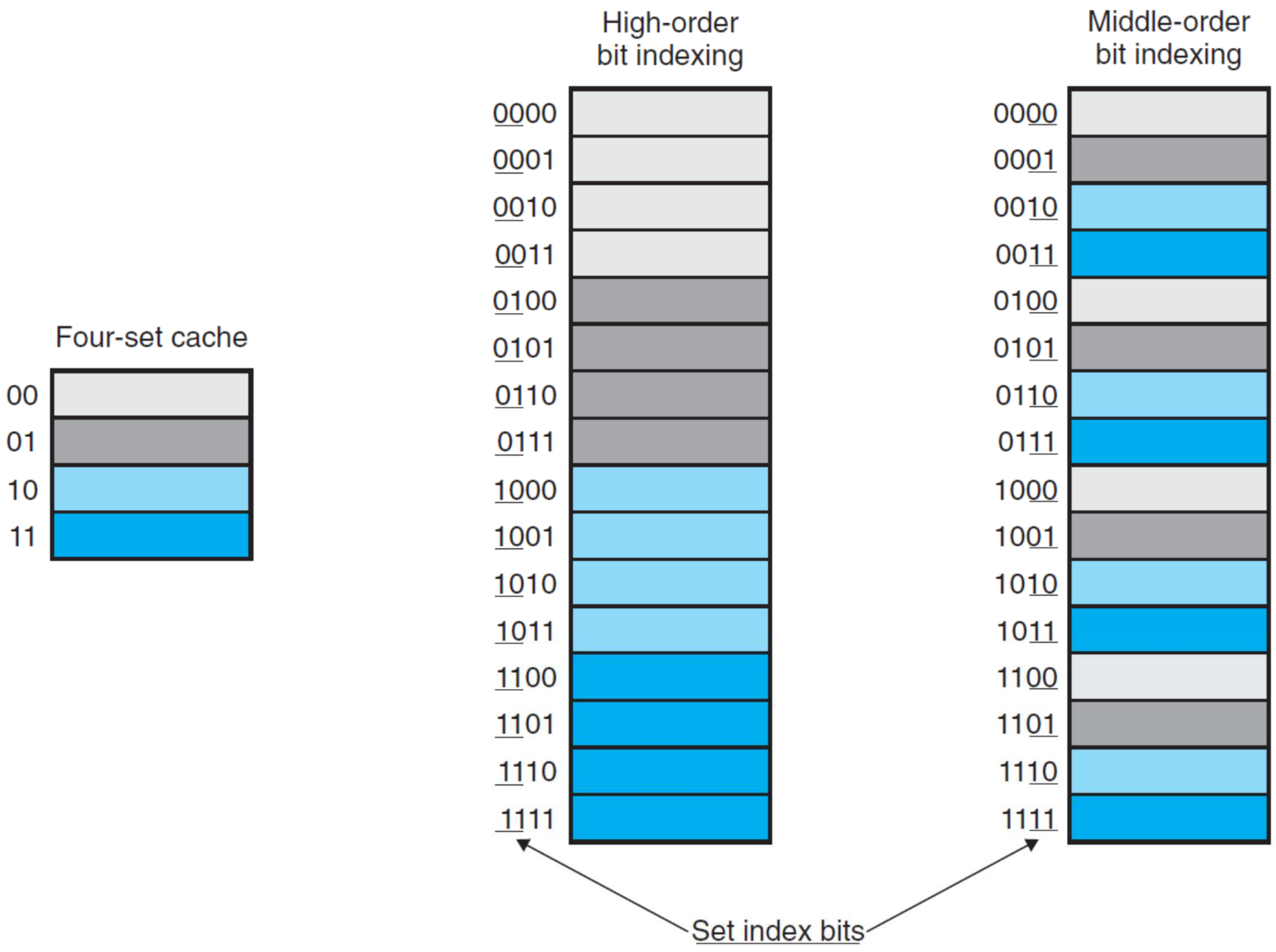
- Given these assumptions, each x[i] and y[i] Will map to the identical cache set:

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

- One easy solution is to put *B* bytes of padding at the end of each array.
- For example, instead of defining x to be float x[8], we define it to be float x[12]. Assuming y starts immediately after x in memory, we have the following mapping of array elements to sets:

Element	Address	Set index	Element	Address	Set index
`x[0]`	0	0	`y[0]`	48	1
`x[1]`	4	0	`y[1]`	52	1
`x[2]`	8	0	`y[2]`	56	1
`x[3]`	12	0	`y[3]`	60	1
`x[4]`	16	1	`y[4]`	64	0
`x[5]`	20	1	`y[5]`	68	0
`x[6]`	24	1	`y[6]`	72	0
`x[7]`	28	1	`y[7]`	76	0

- what fraction of the total references to x and y will be hits once we have padded array x?

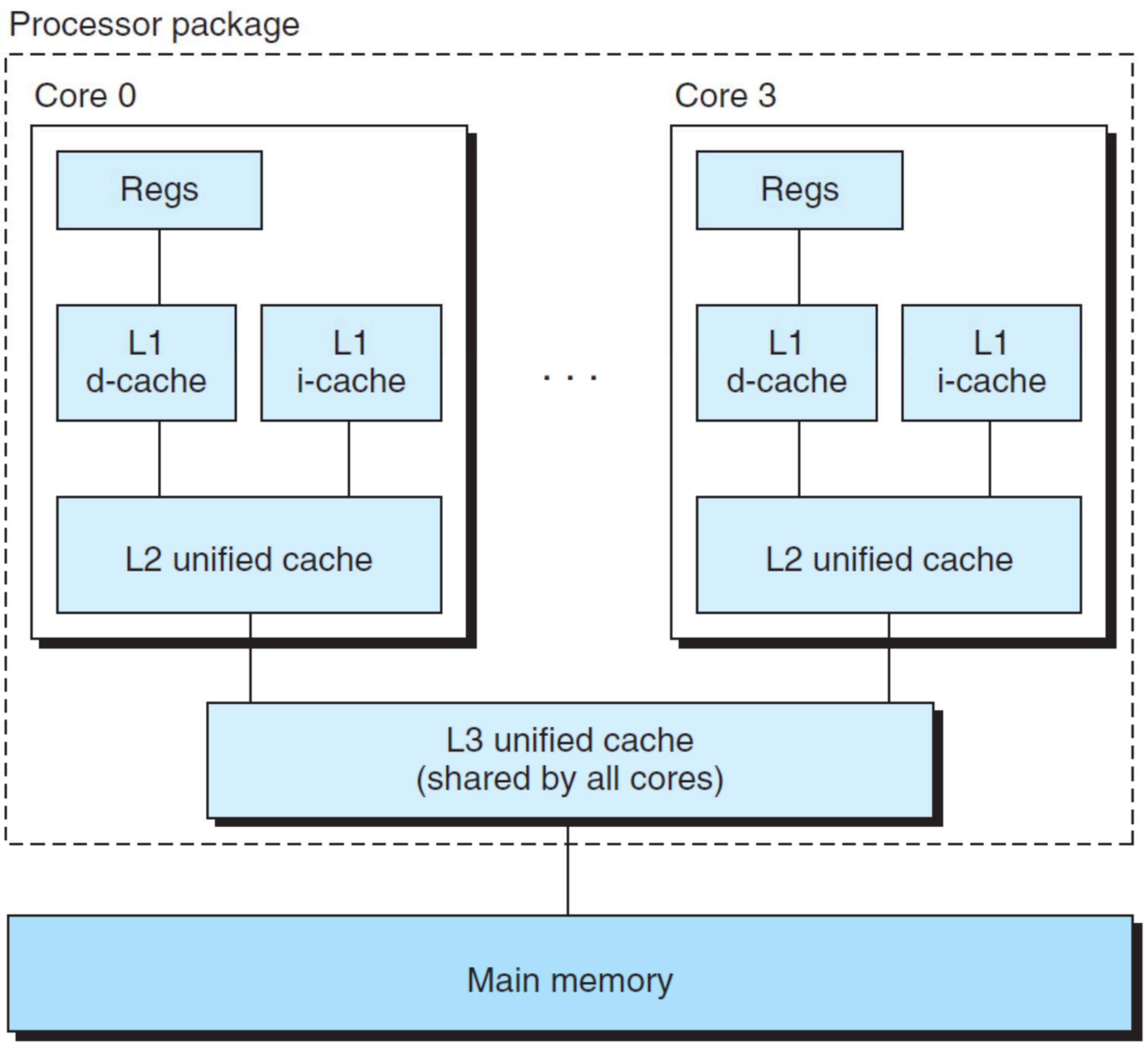

Four-set cache
00
01
10
11
High-order bit indexing
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Middle-order bit indexing
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Set index bits

“cache-friendly” code?

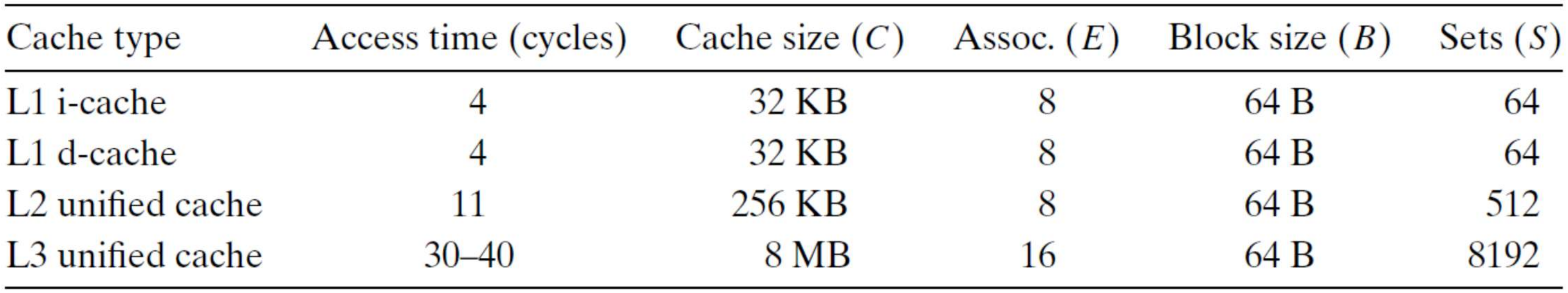
```
for(unsigned int y=0; y<height; ++y)
        { for(unsigned int x=0; x<width; ++x) { ...
        image[y][x] ... } }

for(unsigned int x=0; x<width; ++y)
        { for(unsigned int y=0; y<height; ++x) { ...
                        image[y][x] ... } }
```

Anatomy of a Real Cache Hierarchy

Intel Core i7 cache hierarchy.



Characteristics of the Intel Core i7 cache hierarchy



Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	11	256 KB	8	64 B	512
L3 unified cache	30–40	8 MB	16	64 B	8192

Cache-friendly Code

```
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;
}
```

a[i][j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

```
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;
}
```

a[i][j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

- **Solution to Problem 6.20 (page 620)**
- The key to this problem is noticing that the cache can only hold 1/2 of the array.
- So the column-wise scan of the second half of the array evicts the lines that
- were loaded during the scan of the first half. For example, reading the first element of grid[8][0] evicts the line that was loaded when we read elements from
- grid[0][0]. This line also contained grid[0][1]. So when we begin scanning the
- next column, the reference to the first element of grid[0][1] misses.
- A. What is the total number of read accesses? 512 reads.
- B. What is the total number of read accesses that miss in the cache? 256 misses.
- C. What is the miss rate? 256/512 = 50%.
- D. What would the miss rate be if the cache were twice as big? If the cache were
- twice as big, it could hold the entire grid array. The only misses would be
- the initial cold misses, and the miss rate would be 1/4 = 25%.

- 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.

- determine the cache performance of the following code:

```
    for (i = 0; i < 16; i++){
            for (j = 0; j < 16; j++) {
                    total_x += grid[j][i].x;
                    total_y += grid[j][i].y;
             }
}
```

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?

Considere el problema de la multiplicación de un par de matrices de nxn : C=A.B. Por ejemplo si *n* = 2, entonces

- Donde:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$c_{11} = a_{11}b_{11} + a_{12}b_{21}$

$c_{12} = a_{11}b_{12} + a_{12}b_{22}$

$c_{21} = a_{21}b_{11} + a_{22}b_{21}$

$c_{22} = a_{21}b_{12} + a_{22}b_{22}$