

211: Computer Architecture

Instructor: Prof. David Menendez

Topics:

- Caches

Memory

Thus far, we have used a very simple model of memory

- Main Memory is a linear array of bytes that can be accessed given a memory address
- Also used registers to store values

Reality is more complex. There is an entire memory system.

- Different memories exist at different levels of the computer
- Each vary in their speed, size, and cost

Random-Access Memory (RAM)

Key features

- **RAM** is packaged as a chip.
- Basic storage unit is a **cell** (one bit per cell).
- Multiple RAM chips form a memory.

Static RAM (**SRAM**)

- Each cell stores bit with a six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to disturbances such as electrical noise.
- 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.
- Sensitive to disturbances.
- Slower and cheaper than SRAM.

Memory speeds

Processor Speeds : 1 GHz processor speed is 1 nsec cycle time.

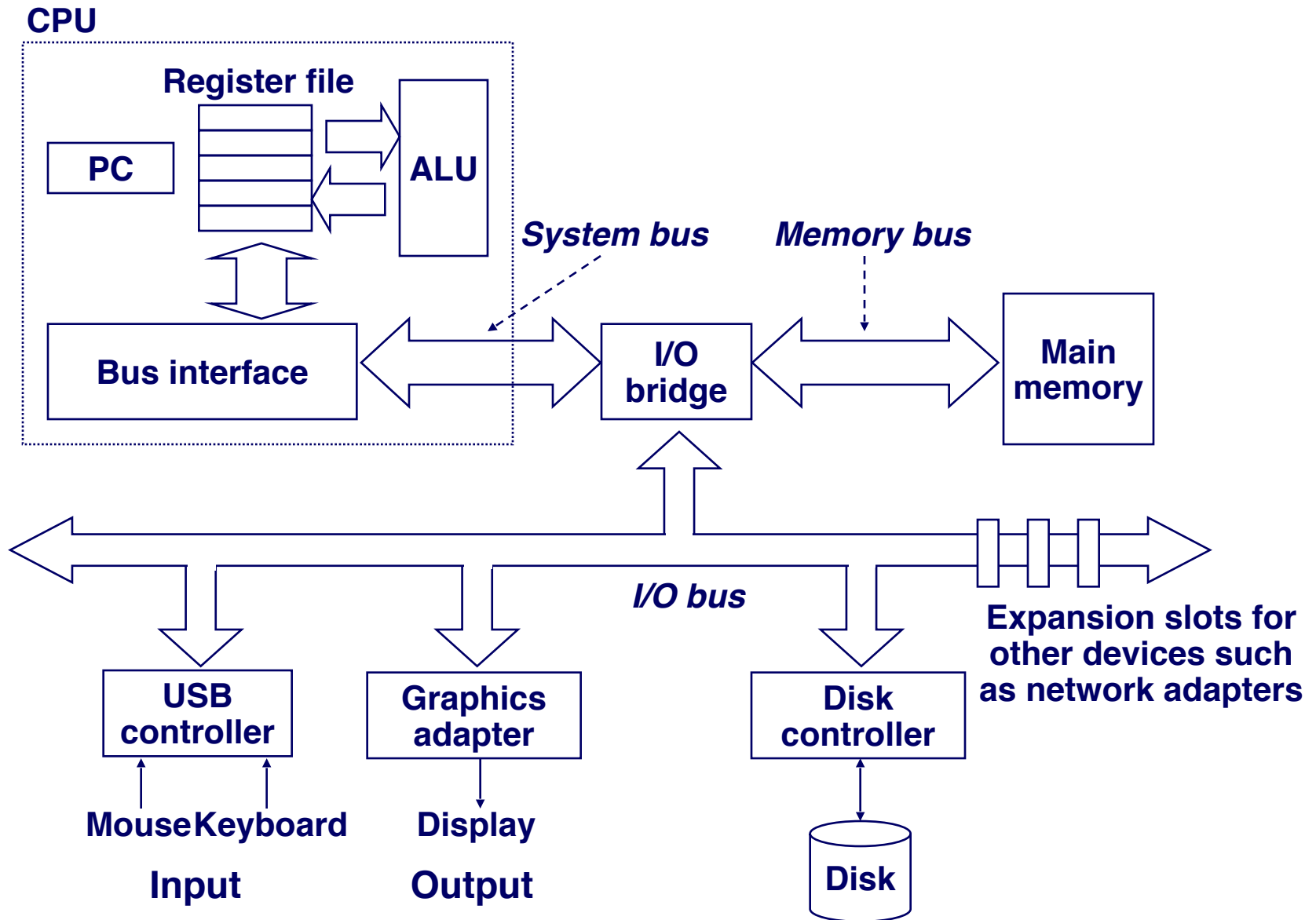
Memory Speeds (50 nsec)

DIMM Module	Clock Speed[MHz]	Bus Speed[MHz]	Transfer Rate [MB/s]
PC1600 DDR200	100	200	1,600
PC2100 DDR266	133	266	2,133
PC2400 DDR300	150	300	2,400

Access Speed gap

- Instructions that store or load from memory

Machine Architecture

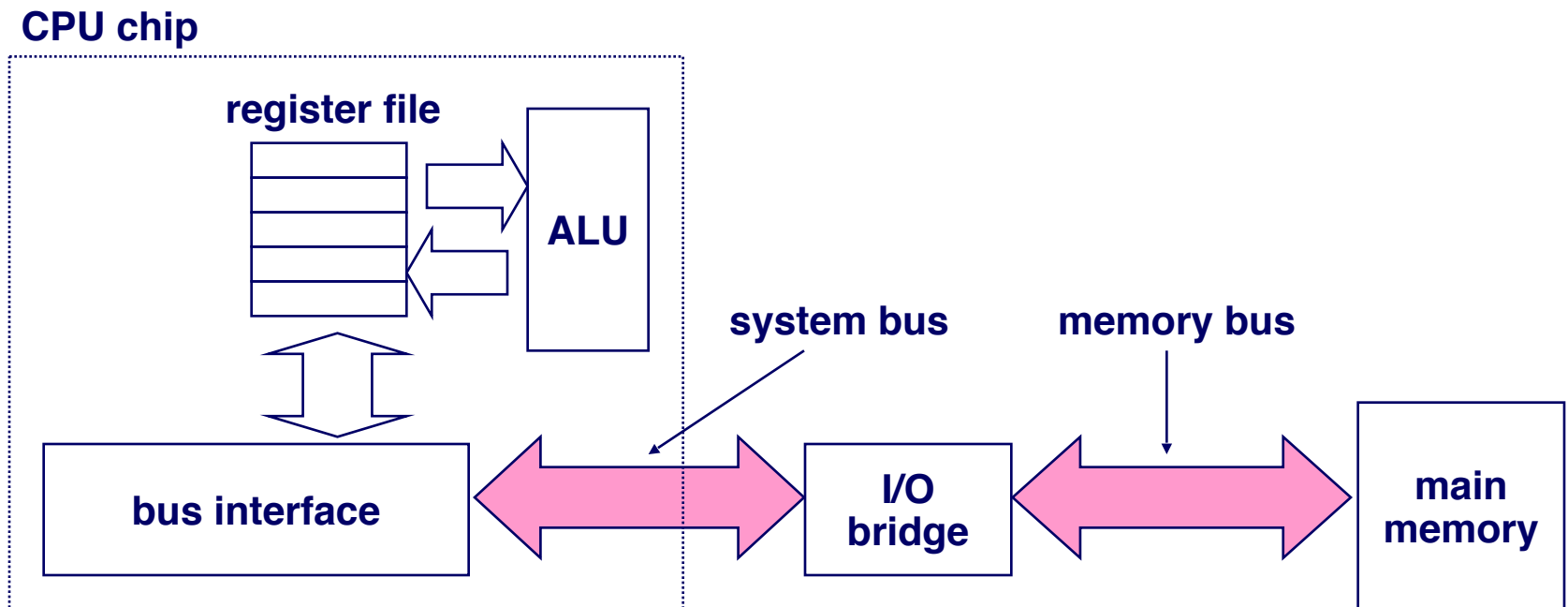


System/Memory Bus

A bus is a collection of parallel wires that carry address, data, and control signals

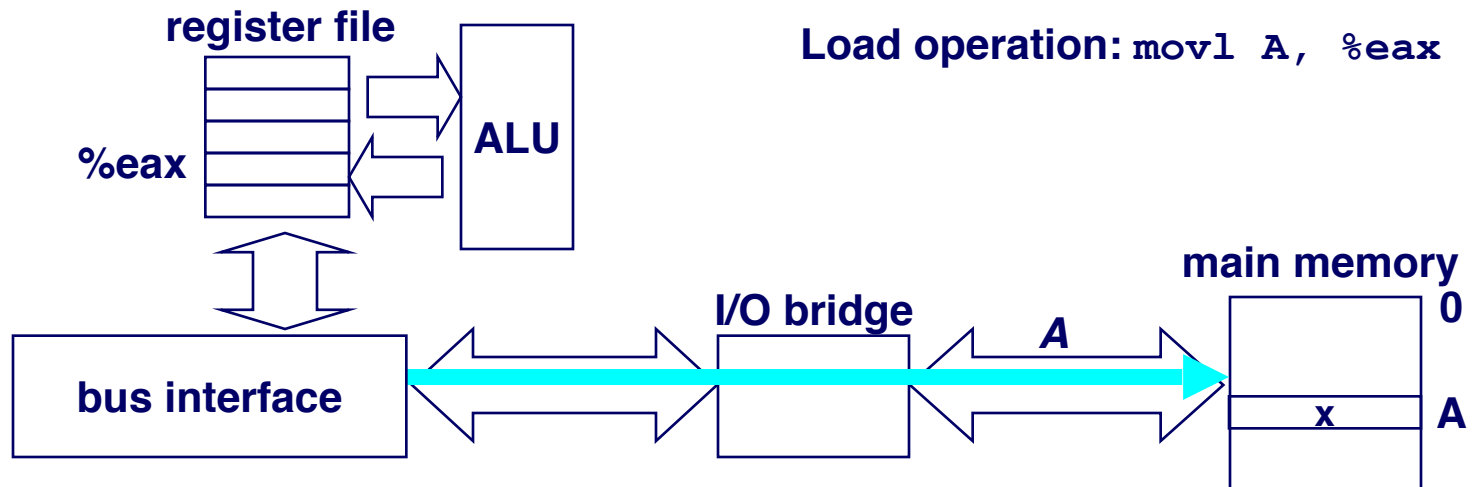
Buses are typically shared by multiple devices

Information passed through transactions



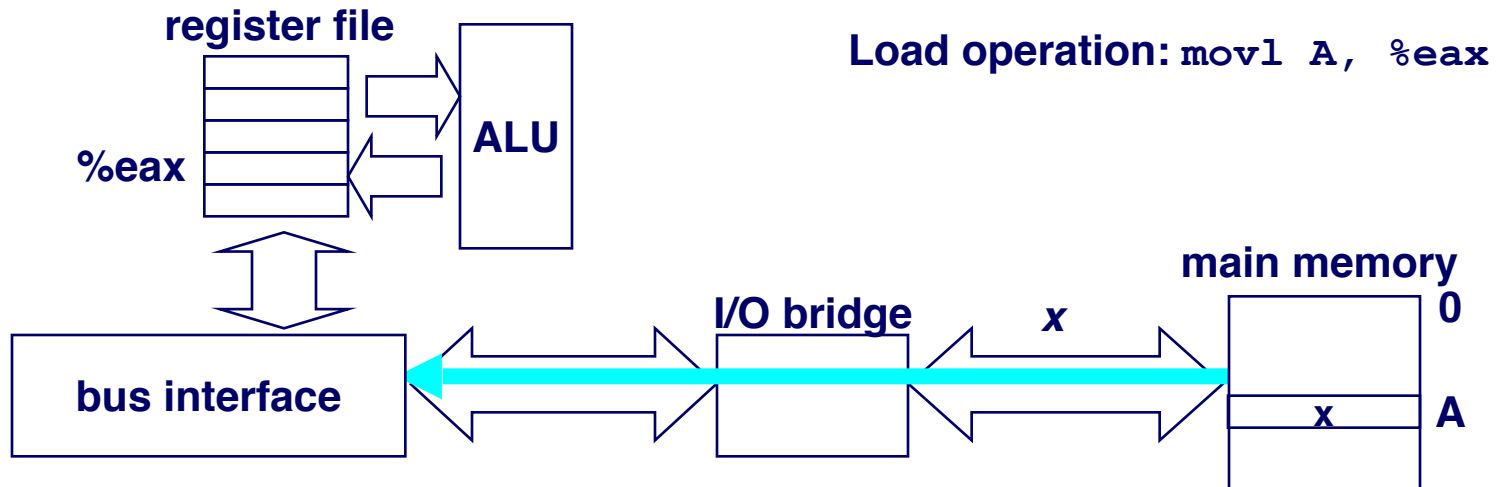
Memory Read Transaction (1)

CPU places address A on the memory bus.



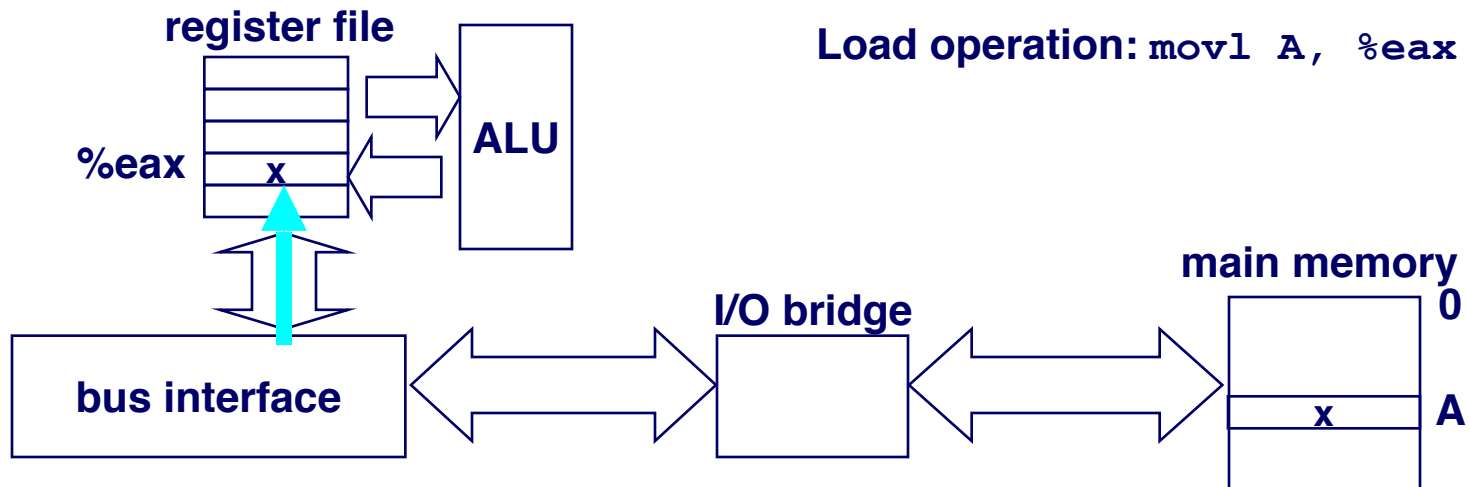
Memory Read Transaction (2)

Main memory reads *A* from the memory bus, retrieves word *x*, and places it on the bus.



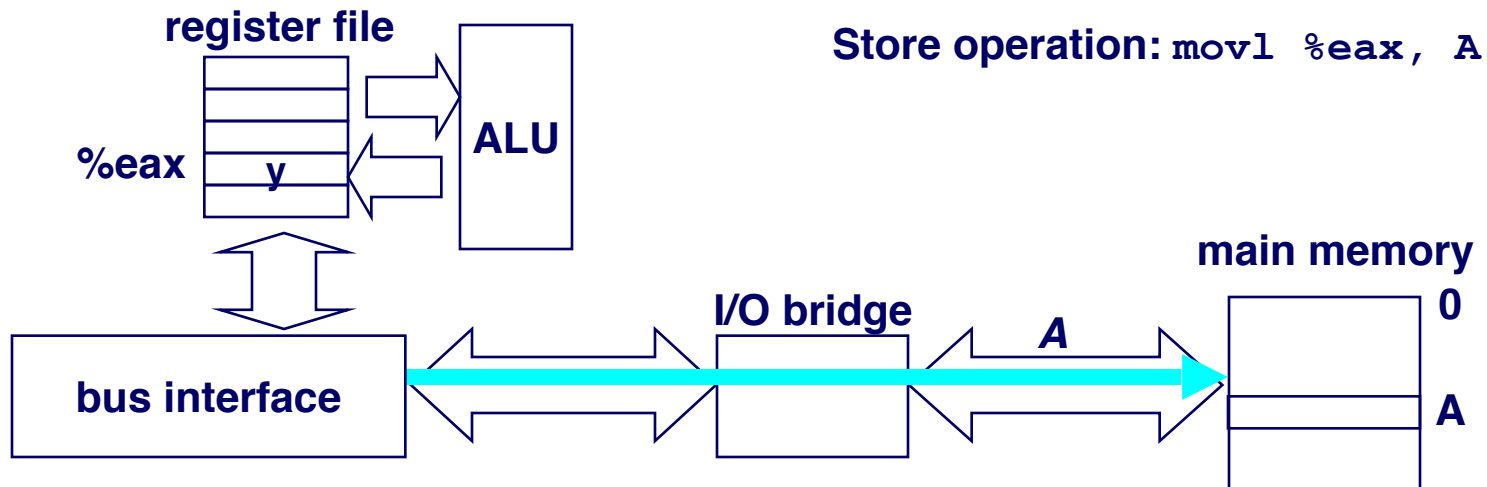
Memory Read Transaction (3)

CPU read word x from the bus and copies it into register %eax.



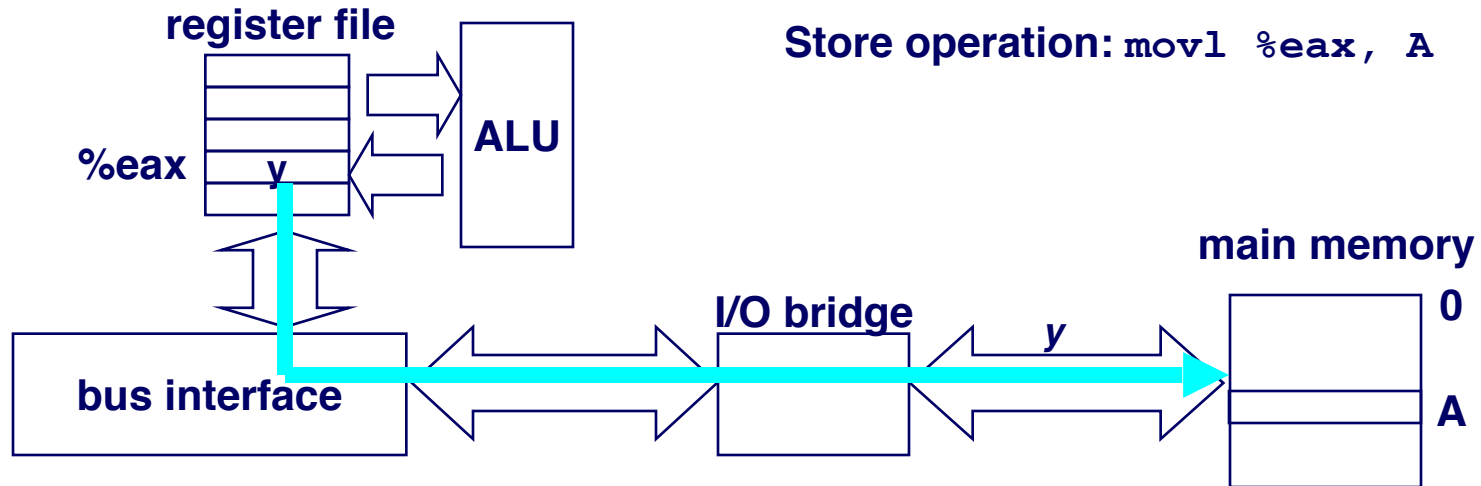
Memory Write Transaction (1)

CPU places address *A* on bus. Main memory reads it and waits for the corresponding data word to arrive.



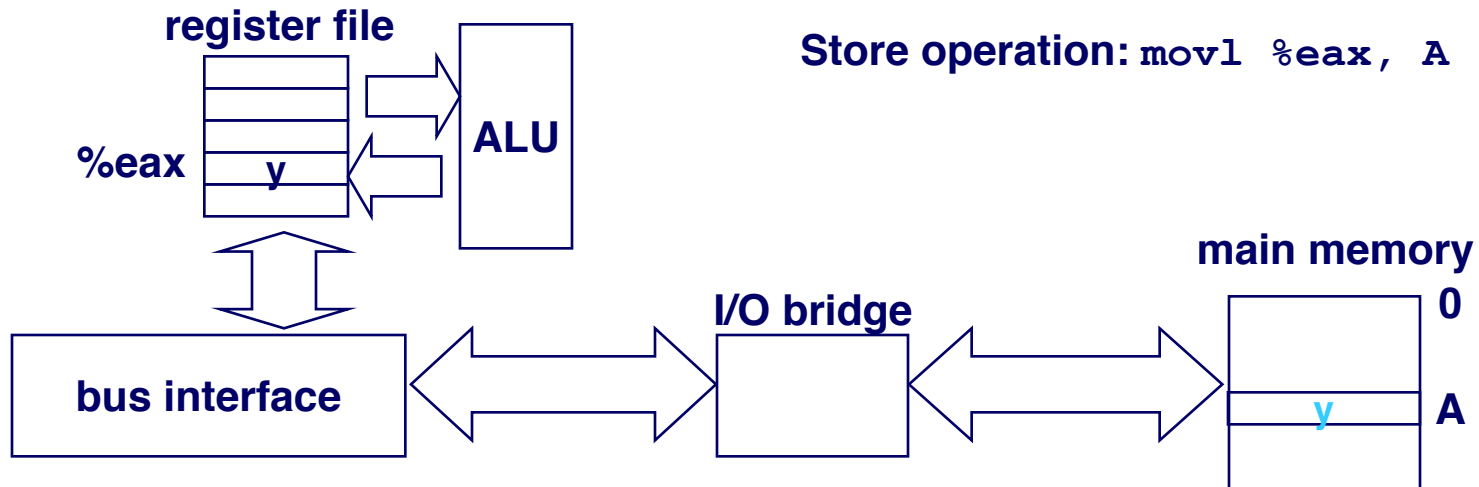
Memory Write Transaction (2)

CPU places data word y on the bus.

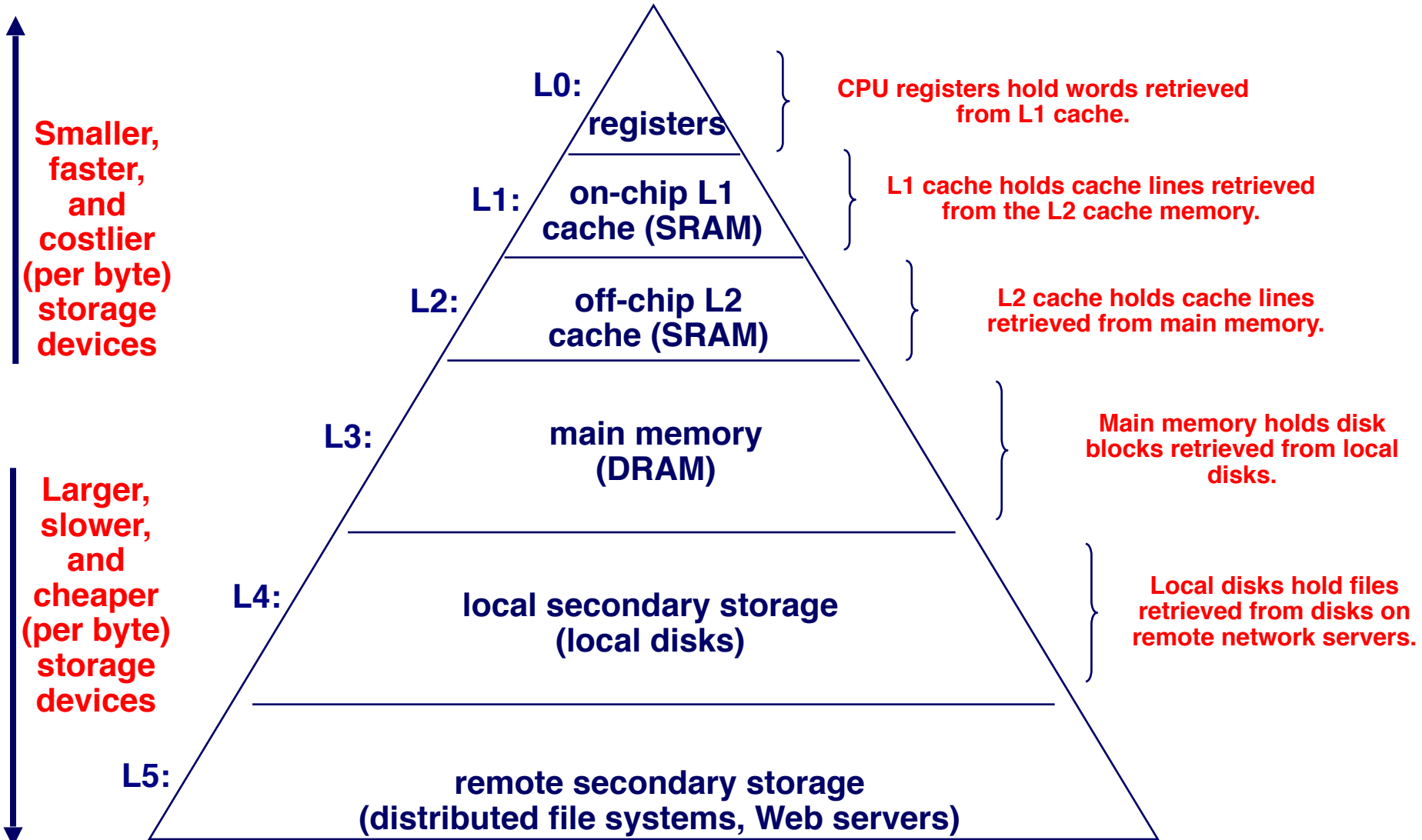


Memory Write Transaction (3)

Main memory read data word *y* from the bus and stores it at address *A*.



Memory Hierarchy (Review)

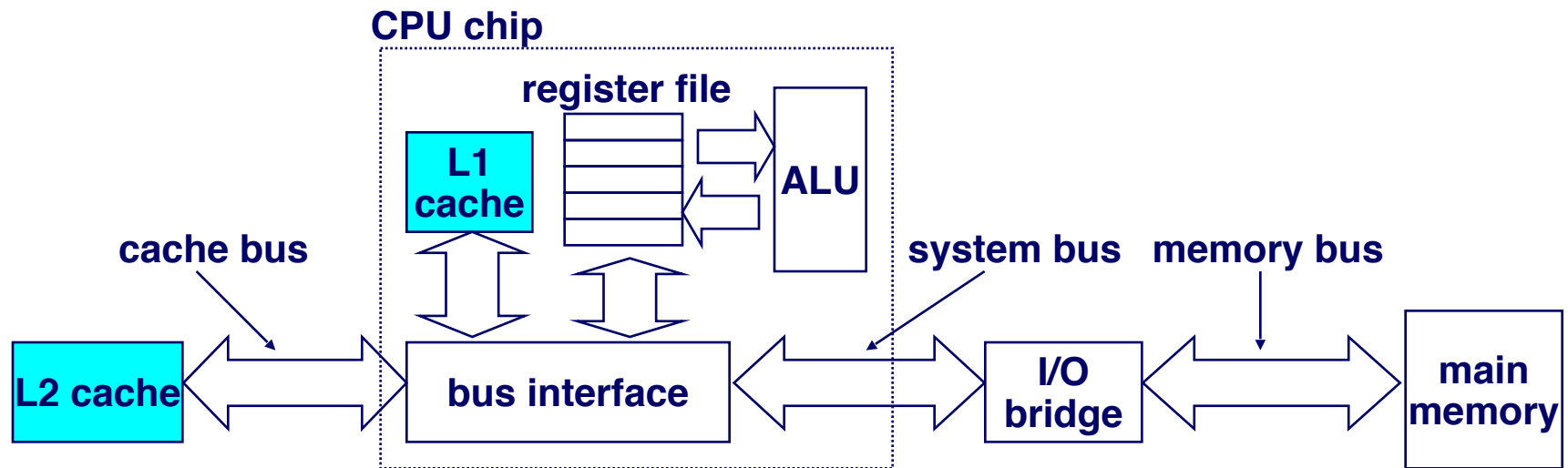


Cache Memories

Cache memories are small, fast 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

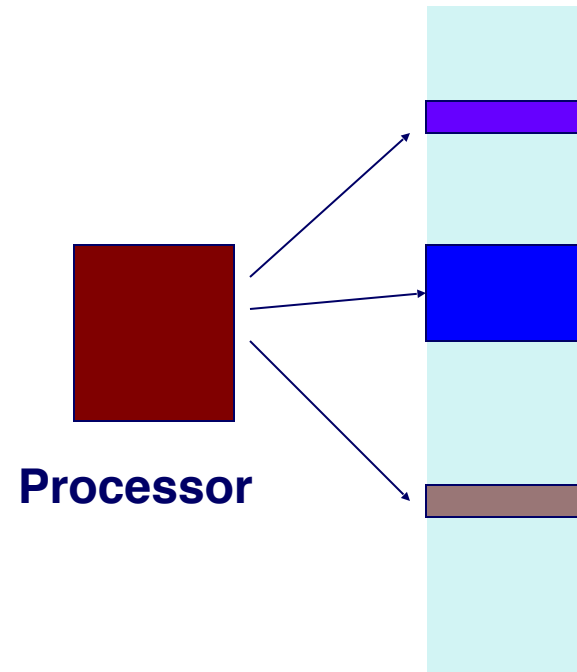


Locality

Memory references are bunched together

- A small portion of address space is accessed at any given time

Cache this portion of the address space in faster memories



Types of Locality

Temporal locality

- Recently accessed locations will likely be accessed again in near future



Spatial locality

- Will likely access locations close to ones recently accessed in near future



Sources of Locality

Temporal locality

- Code within a loop
- Same instructions fetched repeatedly

Spatial locality

- Data arrays
- Local variables in stack
- Data allocated in chunks (contiguous bytes)

Why Is Locality Good?

Address the gap between CPU speed and RAM speed

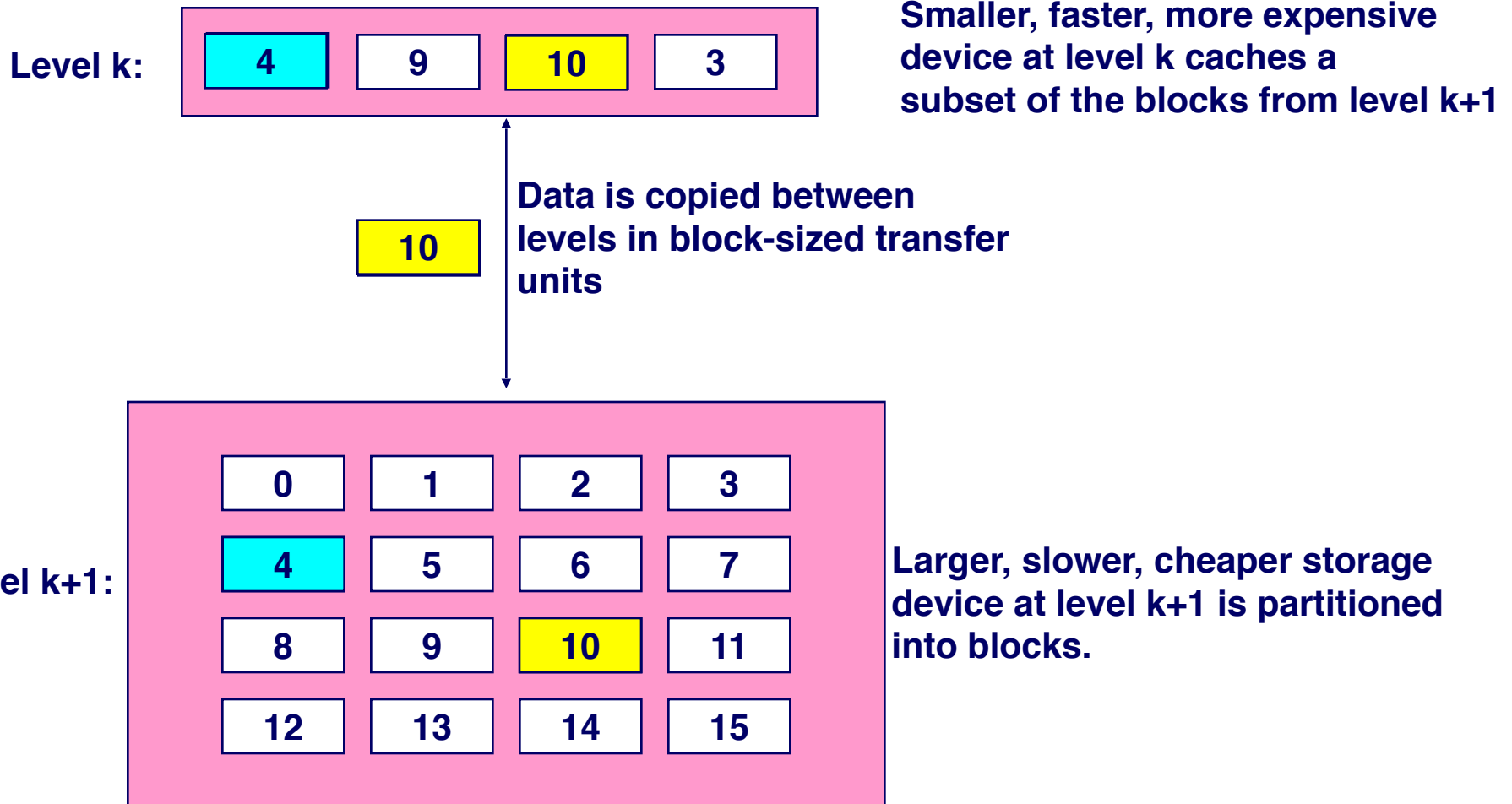
Spatial and temporal locality implies a portion of overall address space can fit in high speed memory

CPU can access instructions and data from this high speed memory

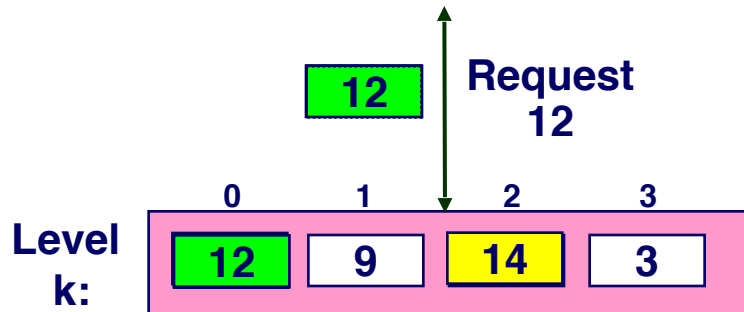
Small high speed memory can make computer faster and cheaper

This is **caching**

Caching in a Memory Hierarchy



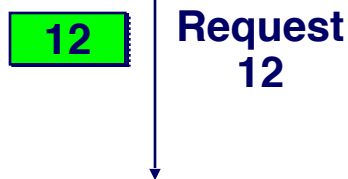
General Caching Concepts



Program needs object d, which is stored in some block b.

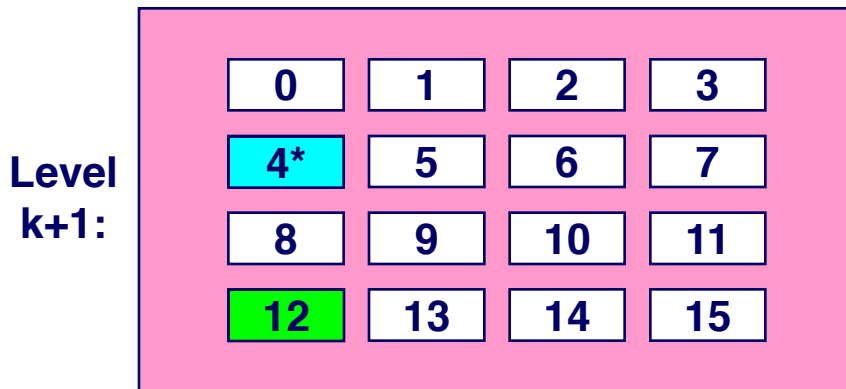
Cache hit

- Program finds b in the cache at level k. E.g., block 14.



Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the “victim”?
 - Placement policy:** where can the new block go? E.g., $b \bmod 4$
 - Replacement policy:** which block should be evicted? E.g., LRU



Cache Miss

Cold (compulsary) miss

- Cold miss occurs when a memory location is accessed for the 1st time

Conflict miss

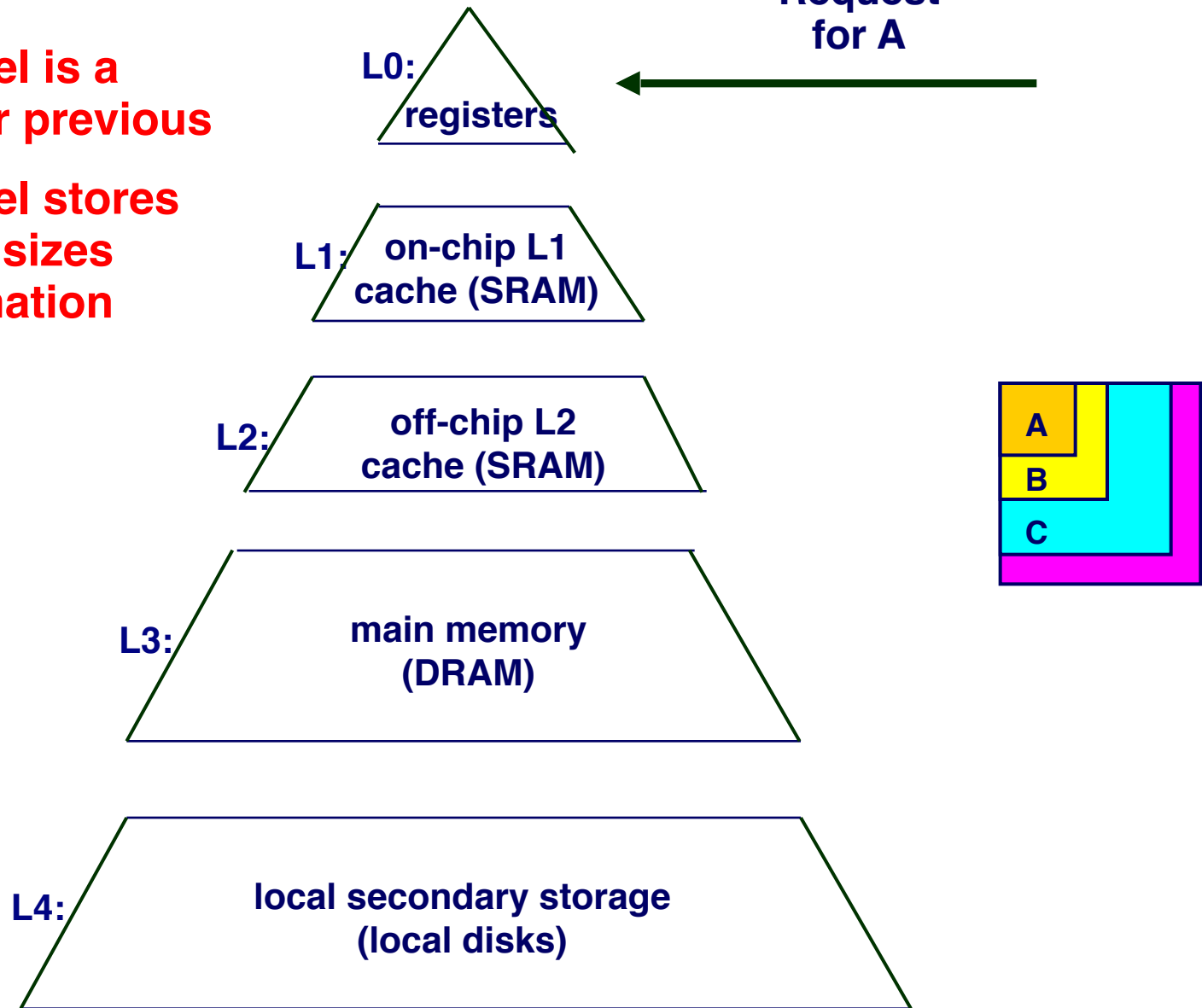
- Most caches limit blocks at level $k+1$ to a small subset of the block positions at level k .
 - E.g., Block i at level $k+1$ must be placed in block $(i \bmod 4)$ at level $k+1$
- Conflict misses occur when the level k cache is large enough, but multiple data items all map to the same level k block.
 - E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

Capacity miss

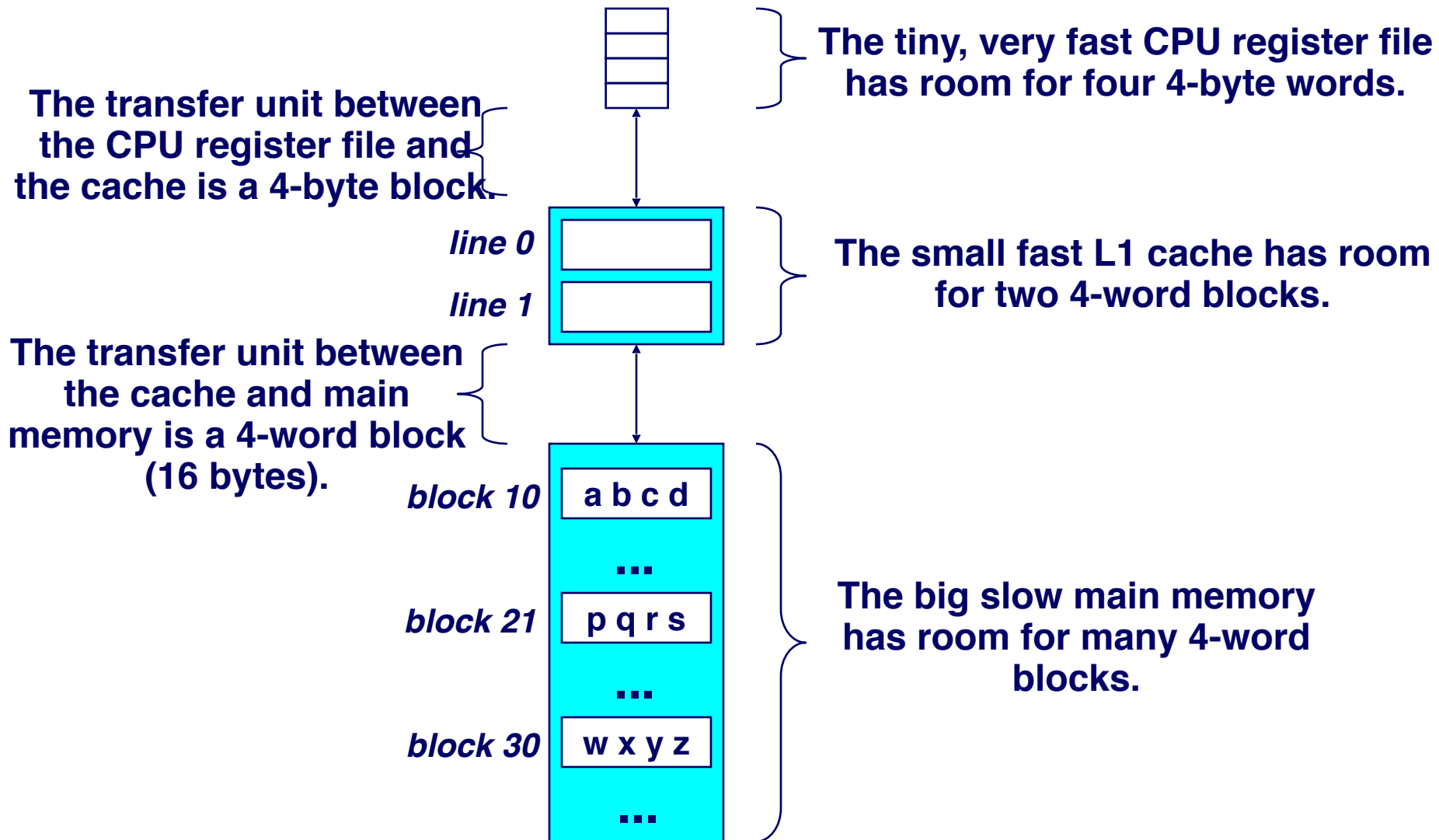
- Occurs when the set of active data blocks (working set) is larger than the cache

Remember:

- Each level is a cache for previous
- Each level stores different sizes of information



L1 Cache



Cache Content

Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

You are essentially allowing a smaller region of memory to hold data from a larger region. Not a 1-1 mapping.

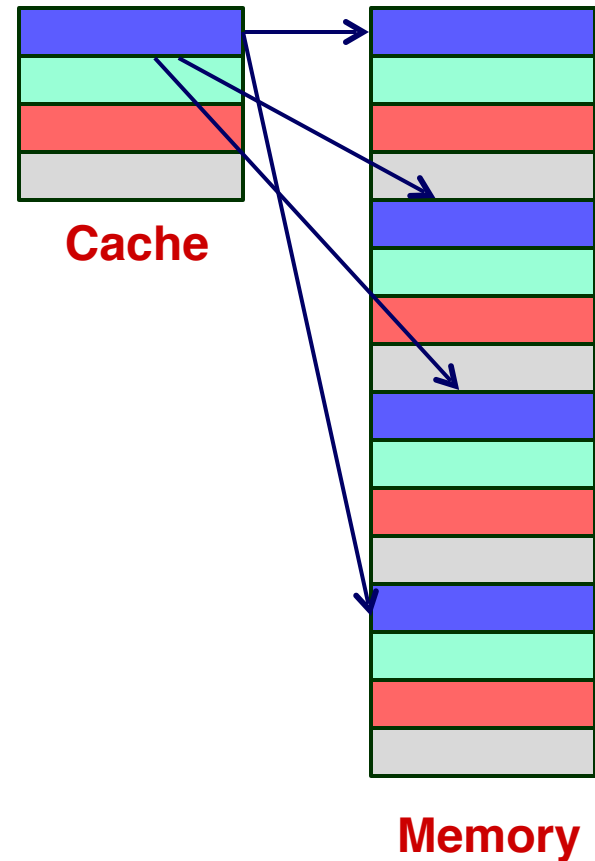
What kind of information do we need to keep:

- The actual data
- Where the data actually comes from
- If data is even considered valid

Cache Mapping

Multiple locations in memory map to same location in cache

In addition to content, cache must keep which entry it is actually caching



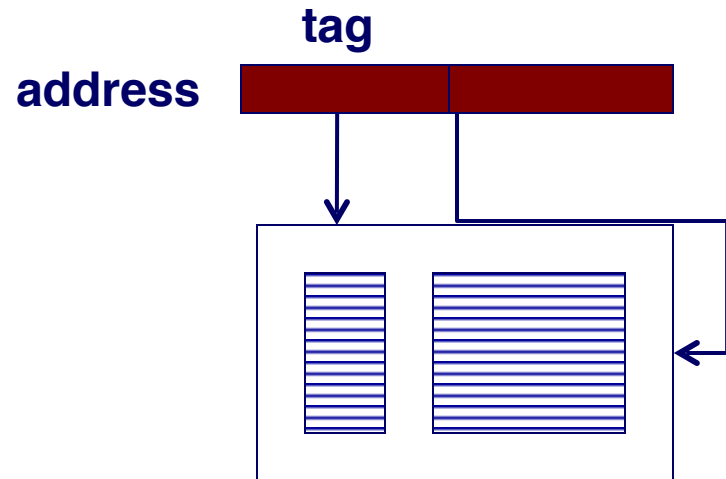
Finding data in cache

Part of memory address applied to cache

Remaining is stored as tag in cache

If tag matches, hit, use data

No match, miss, fetch data from memory



General Org of a Cache Memory

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.

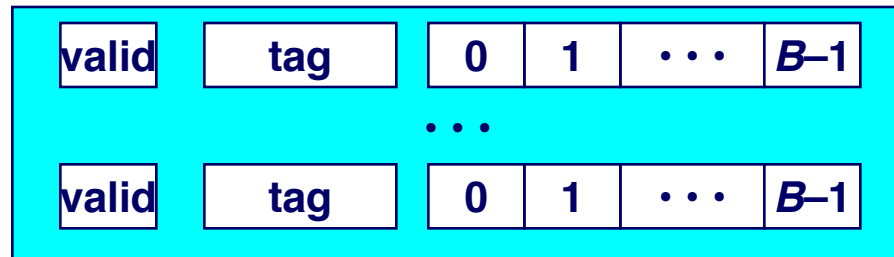
$S = 2^s$ sets

1 valid bit
per line

t tag bits
per line

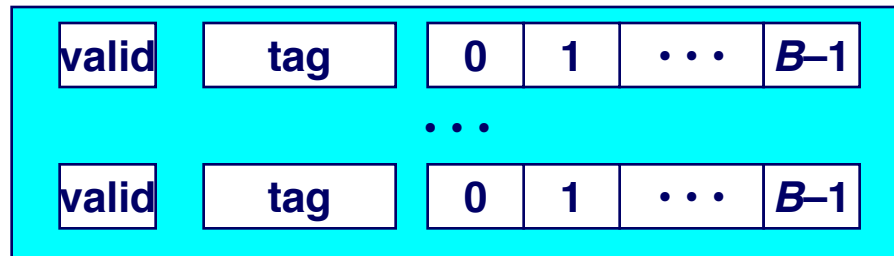
$B = 2^b$ bytes
per cache block

set 0:



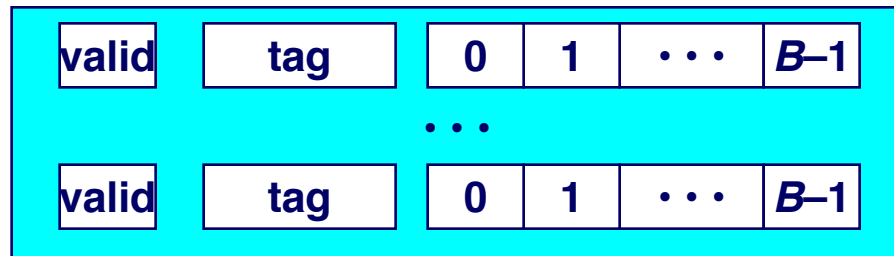
E lines
per set

set 1:



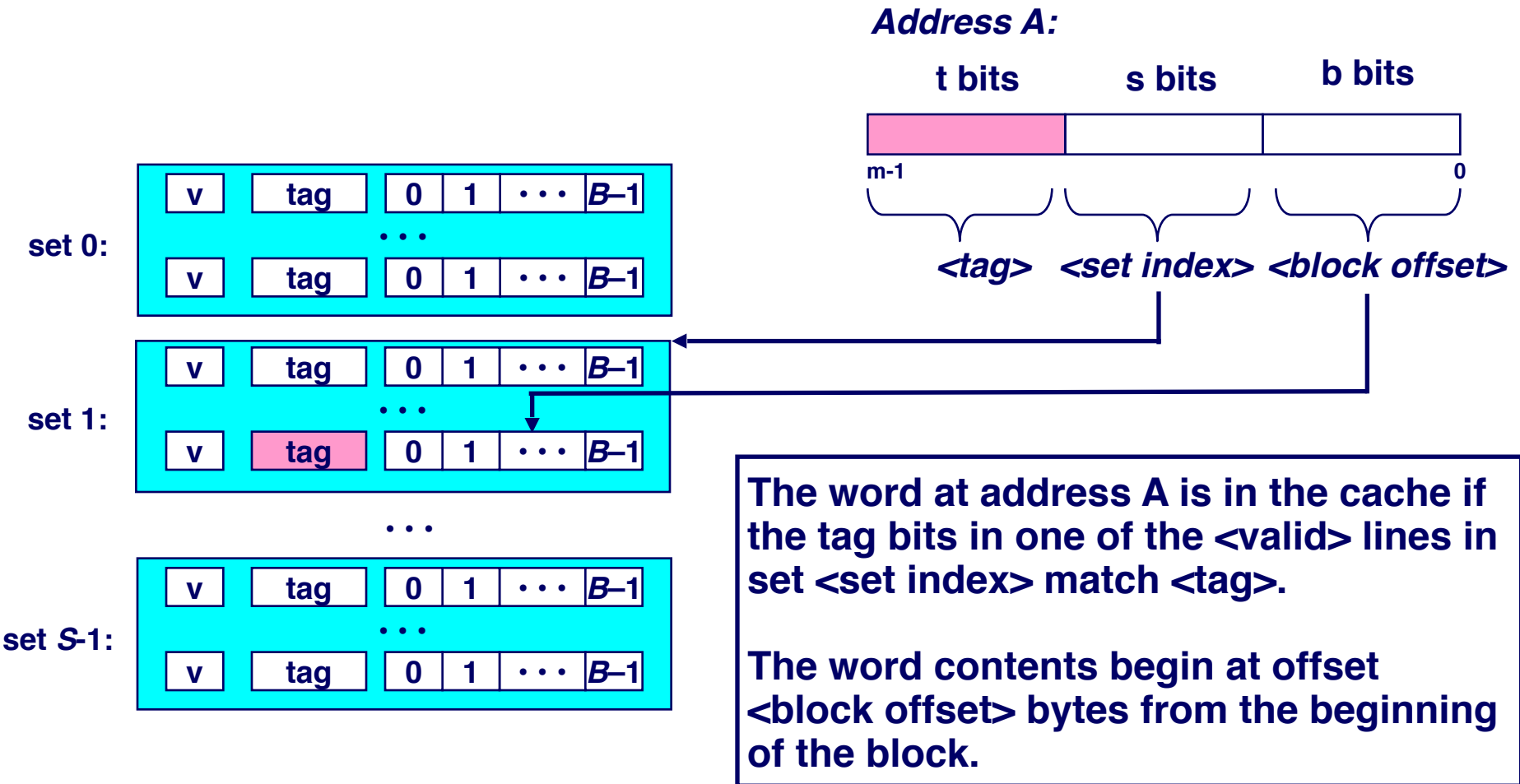
...

set $S-1$:



Cache size: $C = B \times E \times S$ data bytes

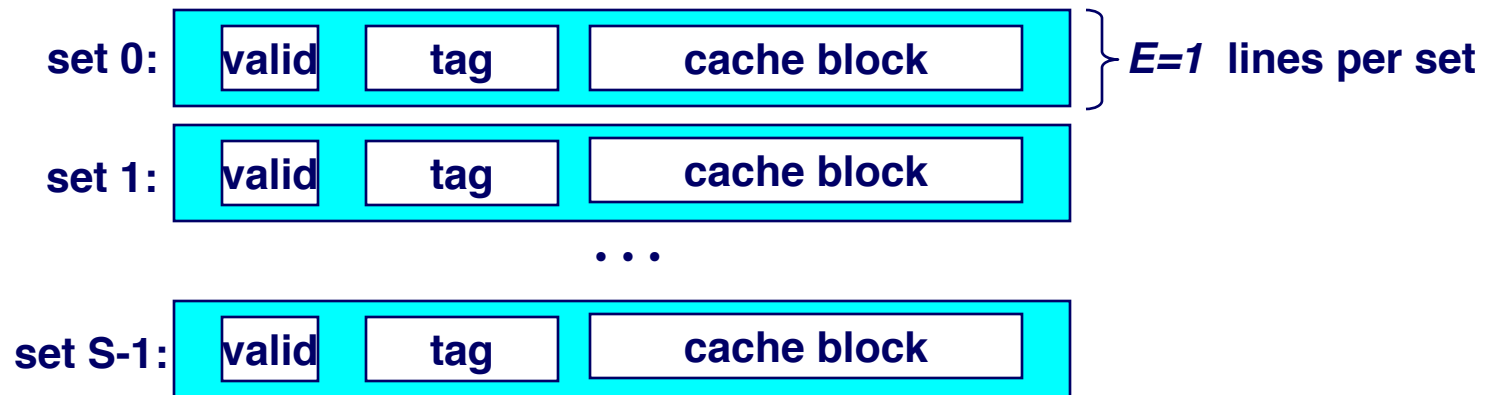
Addressing Caches



Direct-Mapped Cache

Simplest kind of cache

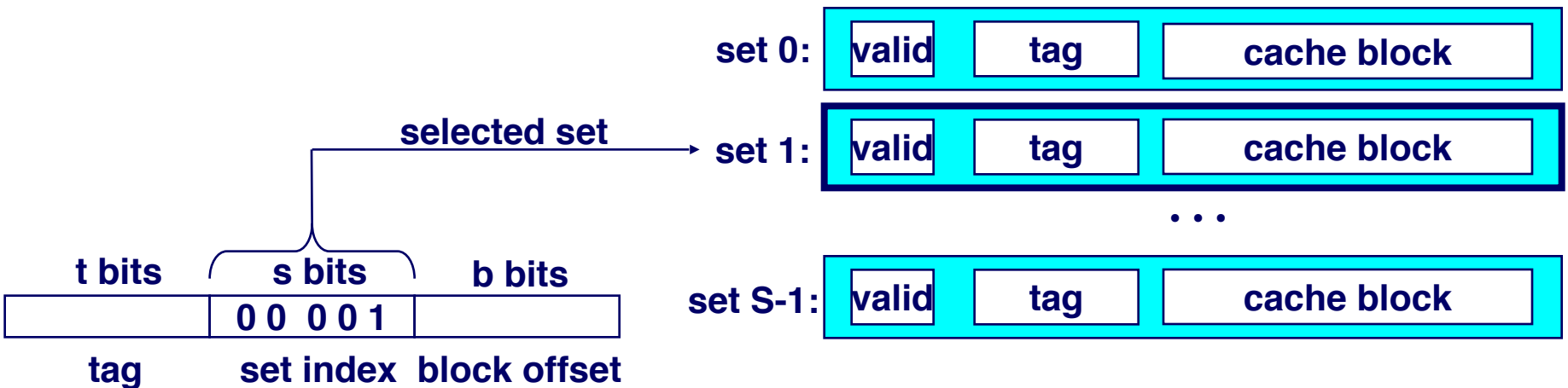
Characterized by exactly one line per set.



Accessing Direct-Mapped Caches

Set selection

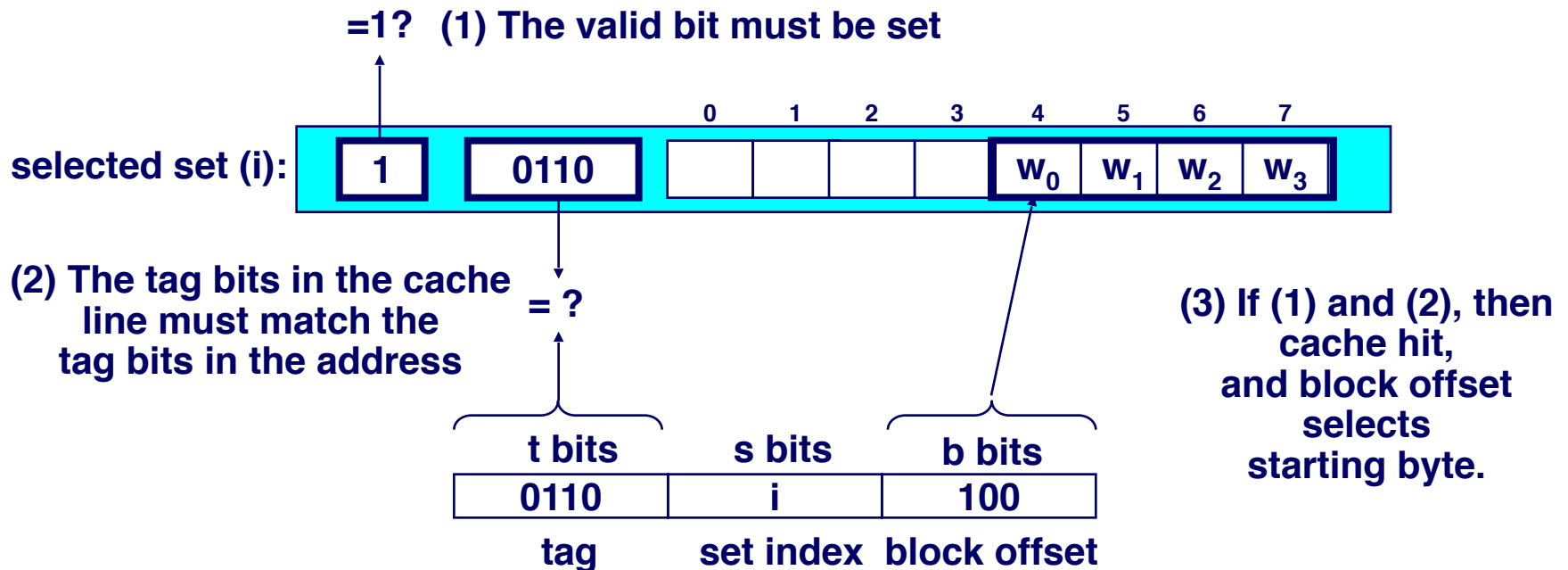
- Use the set index bits to determine the set of interest.



Accessing Direct-Mapped Caches

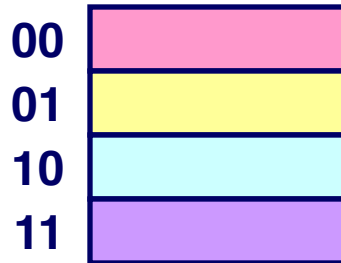
Line matching and word selection

- **Line matching:** Find a valid line in the selected set with a matching tag
- **Word selection:** Then extract the word



Why Use Middle Bits as Index?

4-line Cache



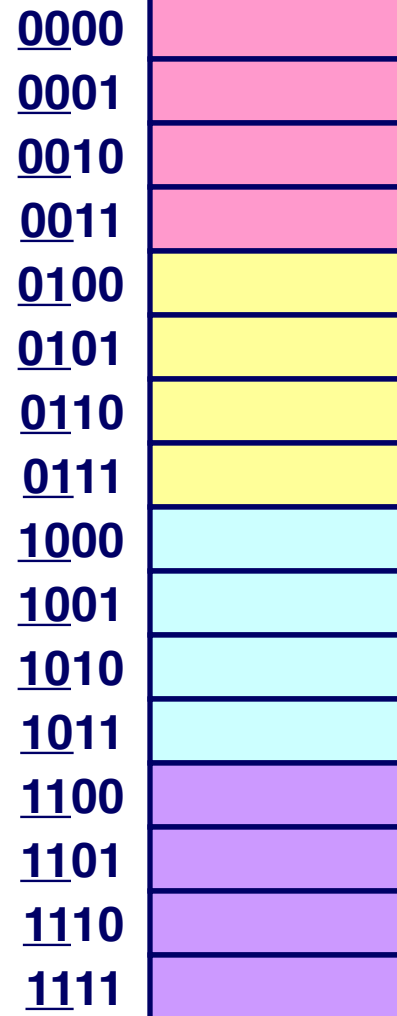
High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

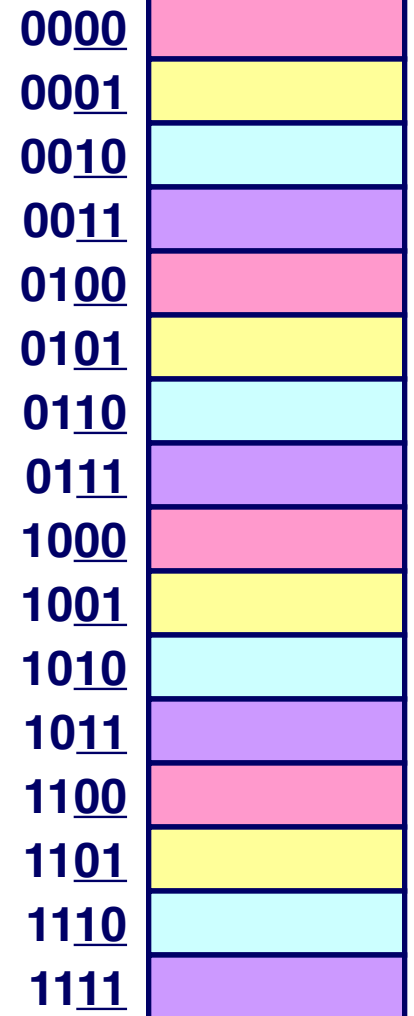
Middle-Order Bit Indexing

- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time

High-Order
Bit Indexing



Middle-Order
Bit Indexing



Direct-Mapped Cache Simulation

M=16 bit addresses, B=2 bytes/block,
S=4 sets, E=1 entry/set

t=1	s=2	b=1
X	XX	X

Address trace (reads):

0 [0000₂], 1 [0001₂], 13 [1101₂], 8 [1000₂], 0 [0000₂]

0 [0000₂] (*miss*)

(1)

v	tag	data
1	0	M[0-1]

13 [1101₂] (*miss*)

(3)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

8 [1000₂] (*miss*)

(4)

v	tag	data
1	1	M[8-9]
1	1	M[12-13]

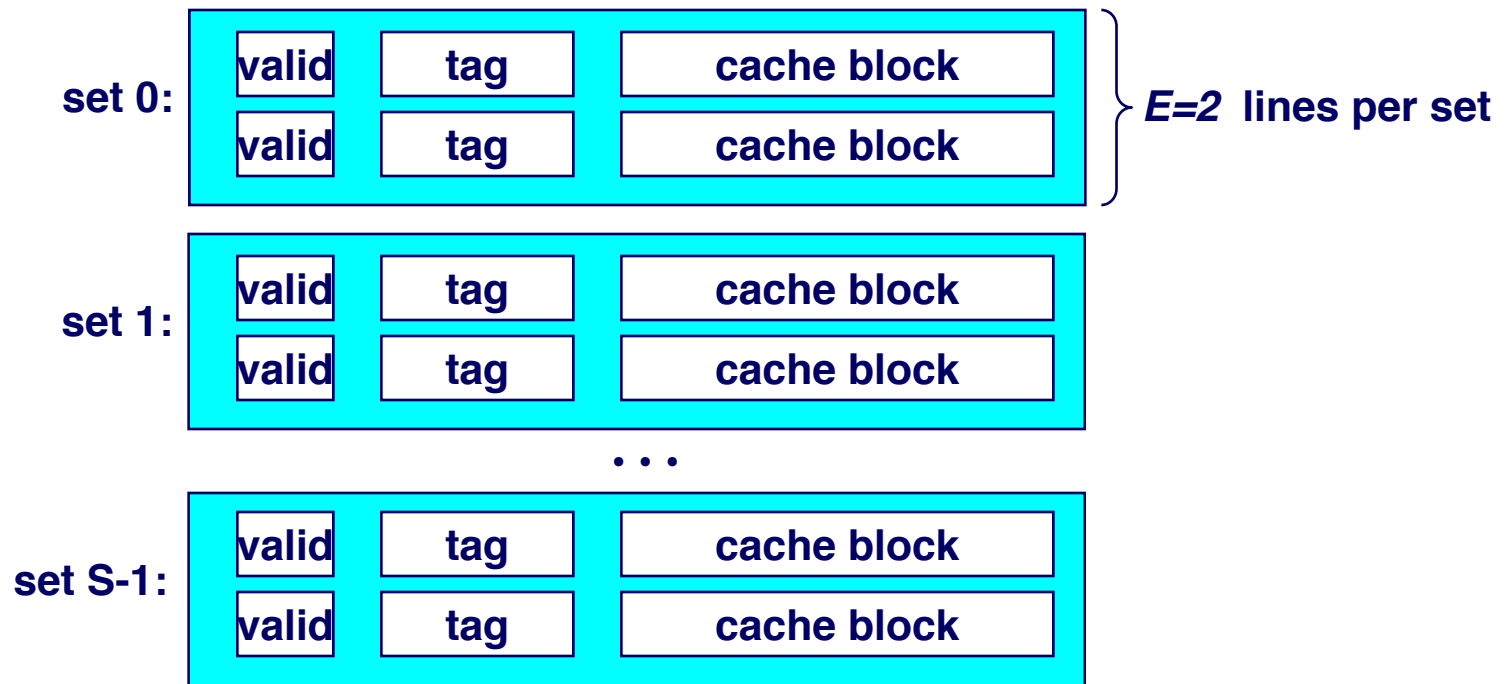
0 [0000₂] (*miss*)

(5)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

Set Associative Caches

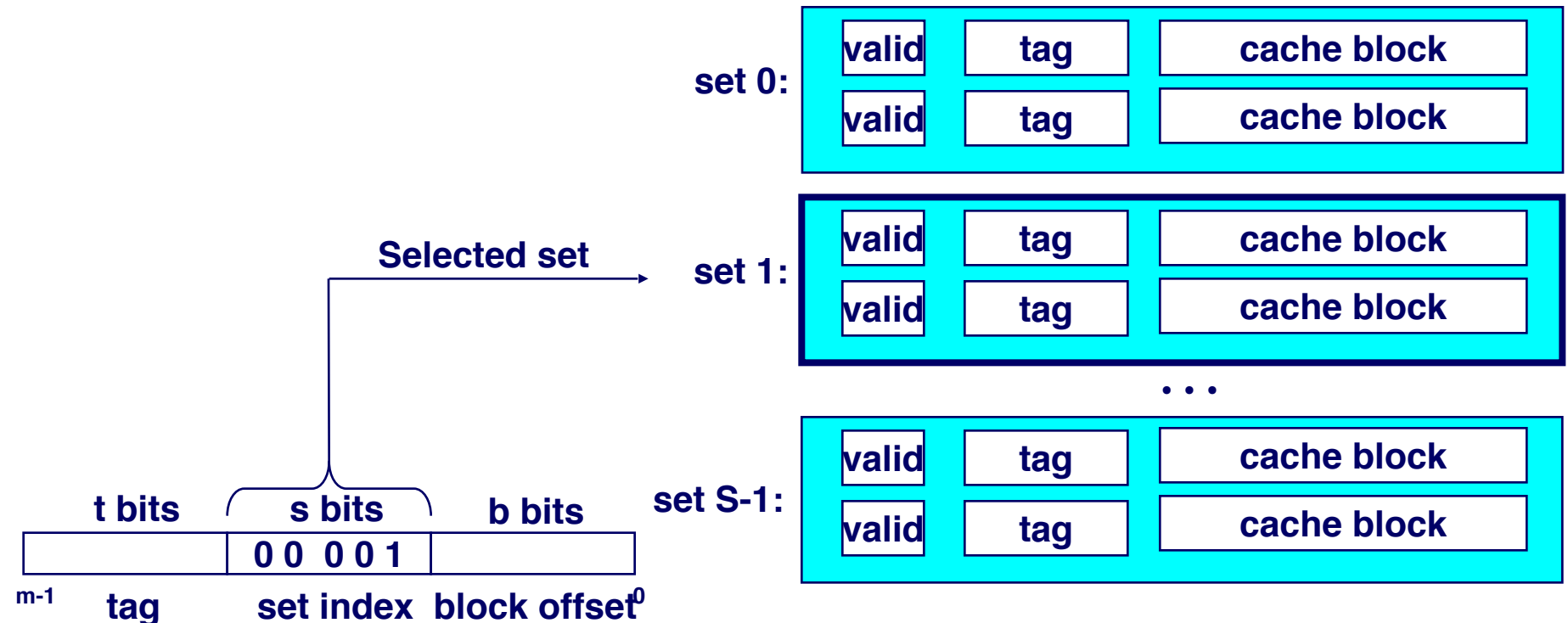
Characterized by more than one line per set



Accessing Set Associative Caches

Set selection

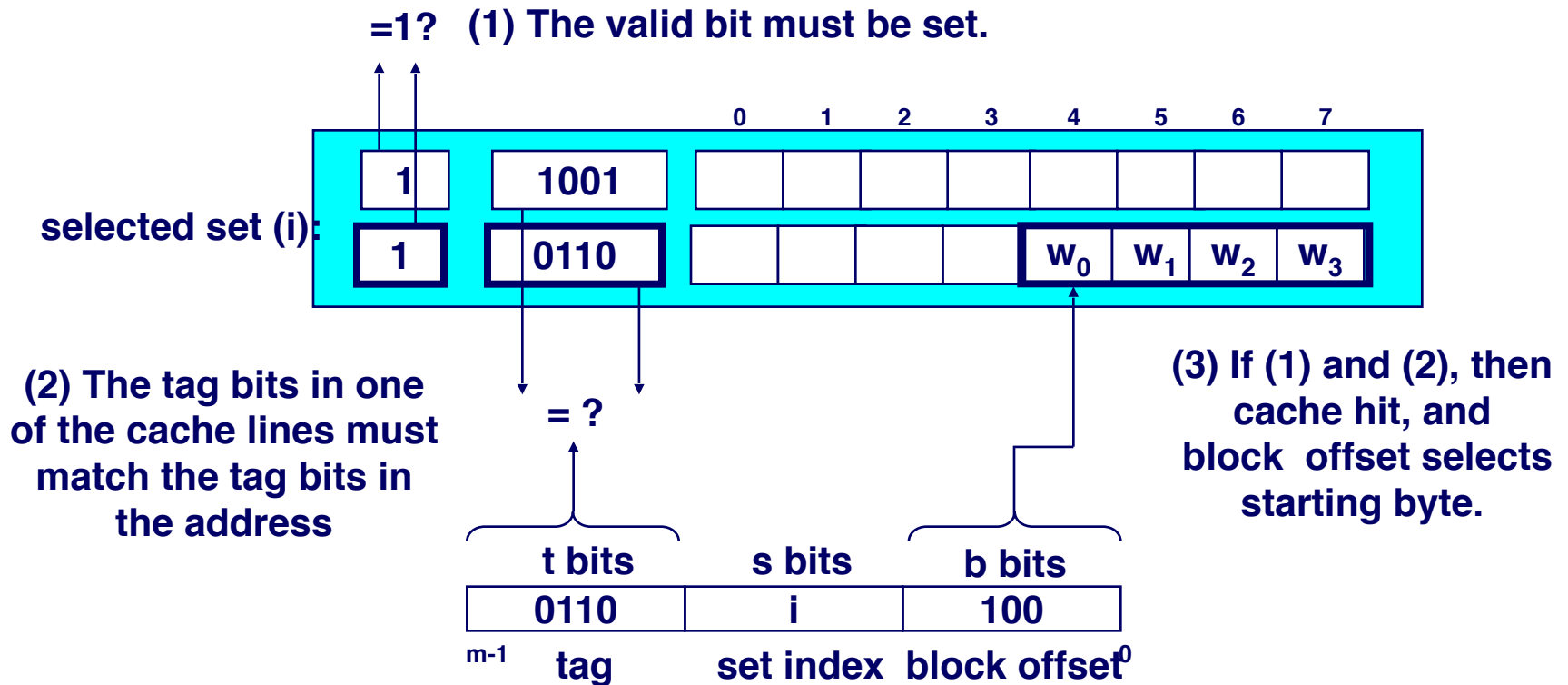
- identical to direct-mapped cache



Accessing Set Associative Caches

Line matching and word selection

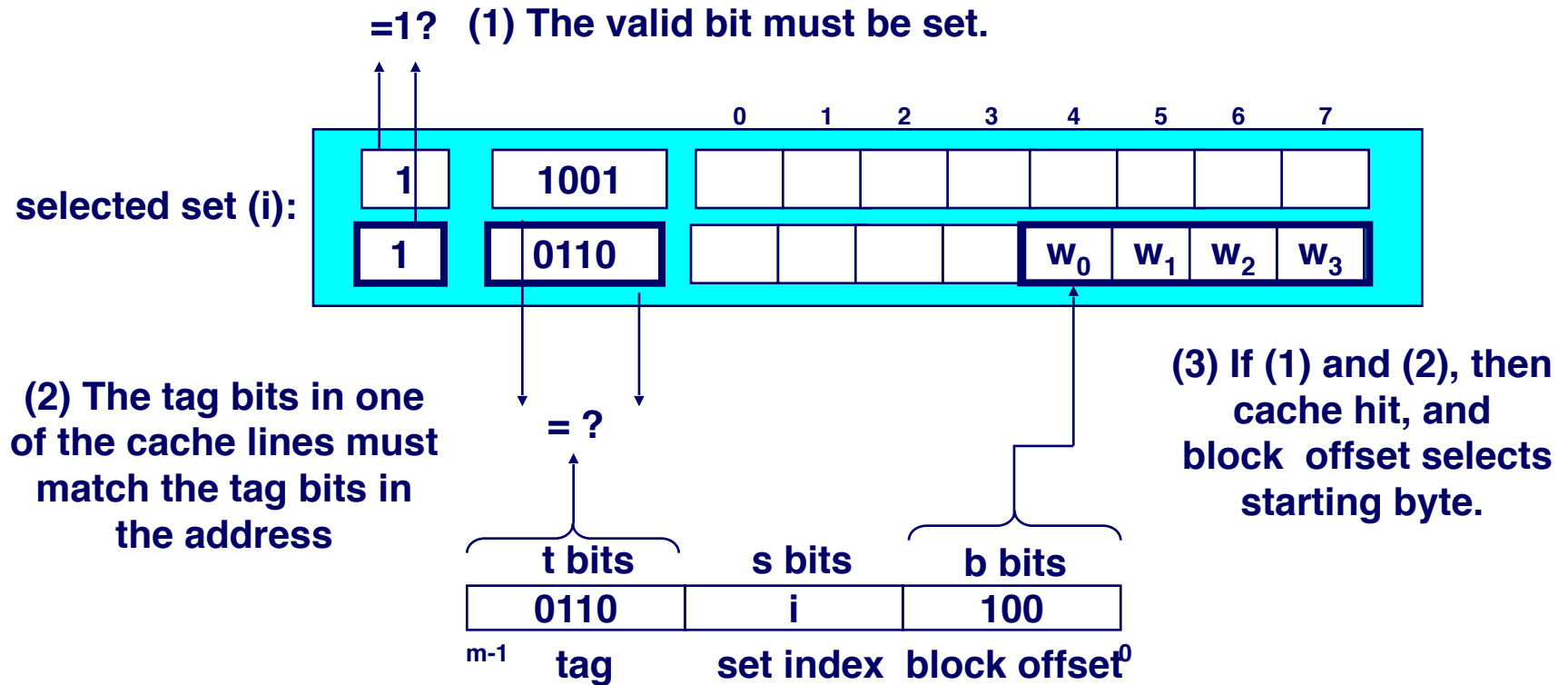
- must compare the tag in each valid line in the selected set.



Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.



Why all of this extra work? Any other issues?

Replacement

What if has a cache miss but all entries in the set are valid?

Direct-mapped cache: no decision

- Discard current content and bring needed content in

Set Associate cache: which line to evict?

- Need a replacement algorithm

If we knew the future, can you think of an optimal replacement algorithm?

Since we are not oracles, we need to approximate

- FIFO: First-In-First-Out
- LRU: Least Recently Used
- Random: Select victim from set randomly

Replacement

How would you implement FIFO replacement in software?

How would you implement LRU replacement in software?

How would you implement these algorithms in hardware?

Fully Associative Caches

Set selection is trivial

Accessing a line is the same as a set associative cache

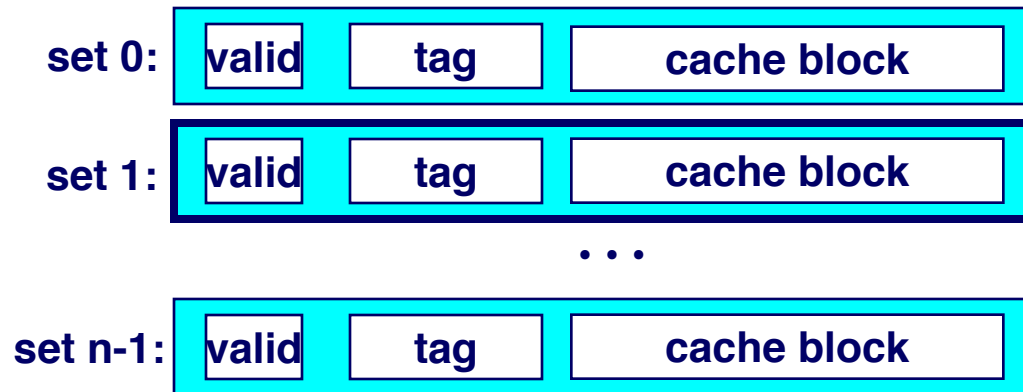
set 0:

valid	tag	cache block
valid	tag	cache block
valid	tag	cache block
valid	tag	cache block
...		
valid	tag	cache block
valid	tag	cache block

Example: Direct mapped cache

32 bit address, 64KB cache, 32 byte block

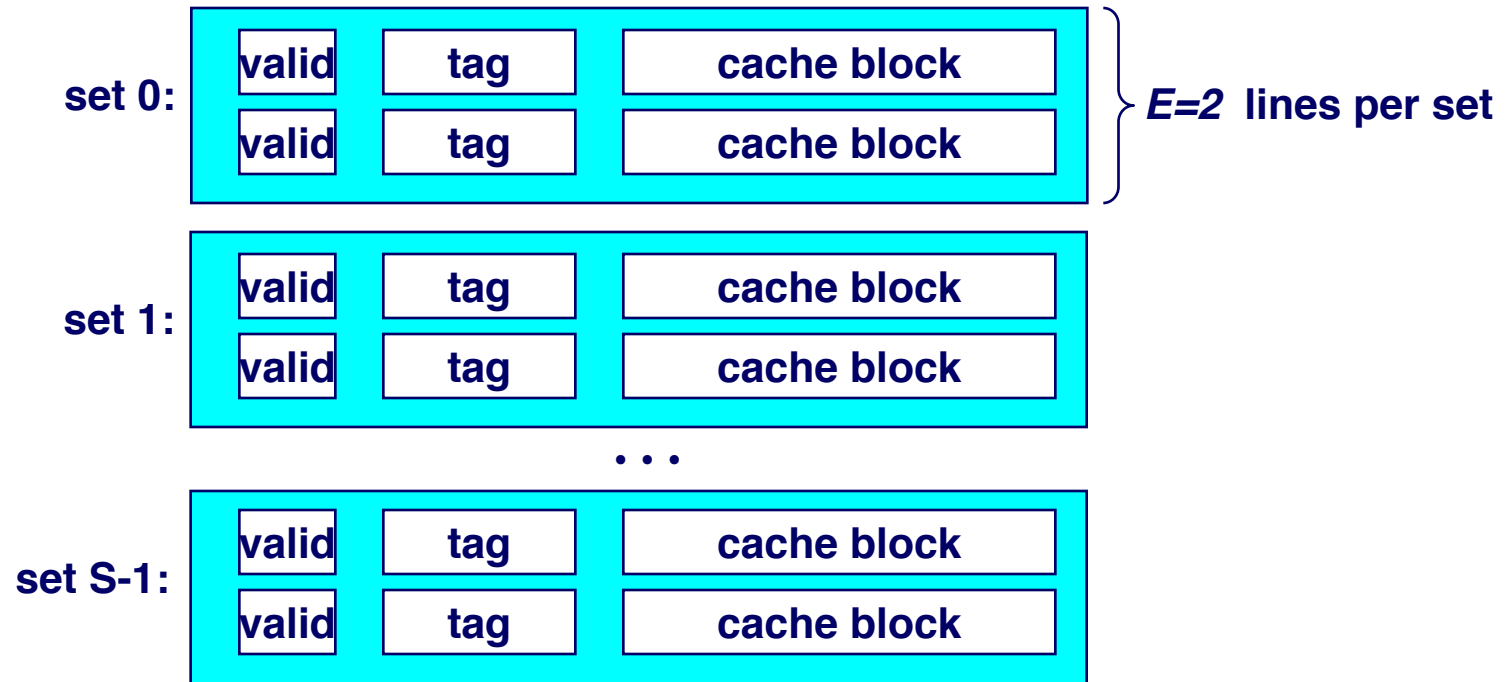
How many sets, how many bits for the tag, how many bits for the offset?



Example: 2-way associative cache

32 bit address, 64KB cache, 32 byte block

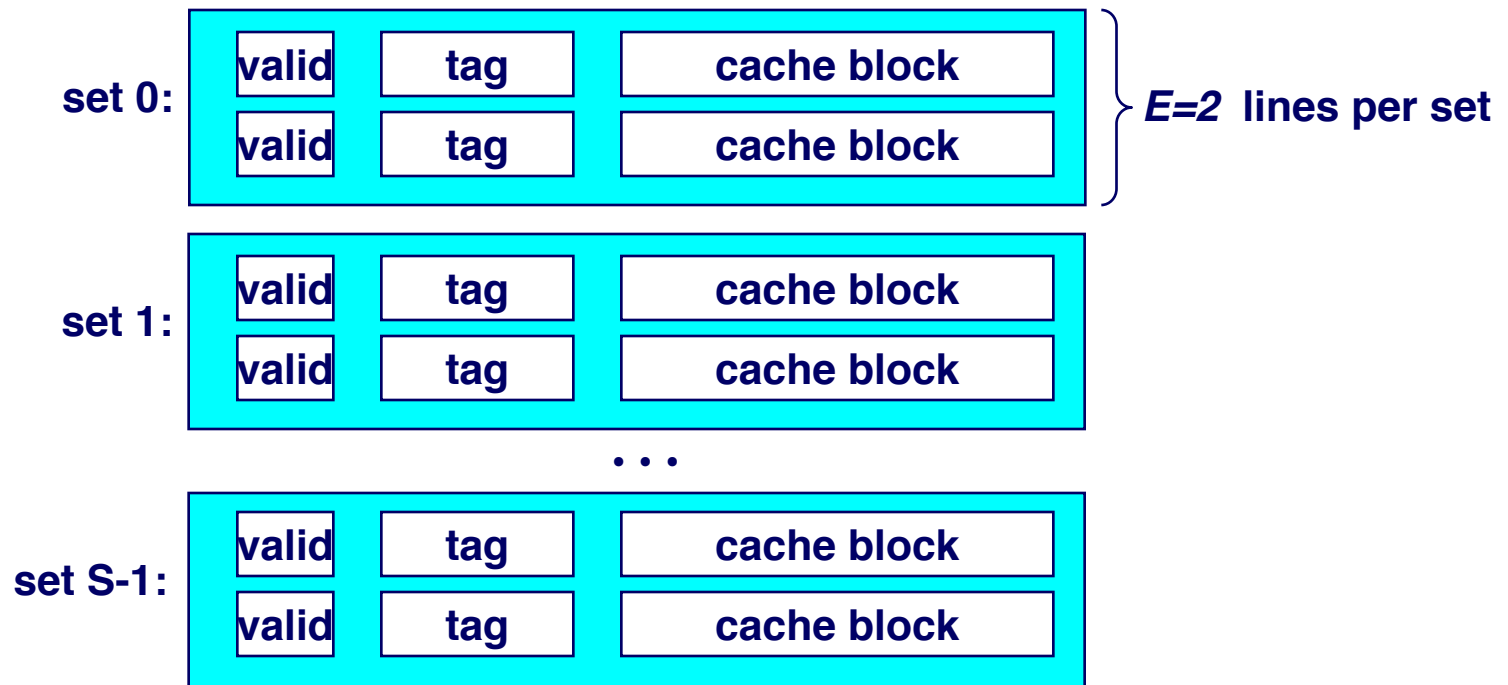
How many sets (lines), how many bits for the tag, how many bits for the offset?



Example: 2-way associative cache

32 bit address, 32KB cache, 16 byte block

How many sets, how many bits for the tag, how many bits for the offset?



Writes and Cache

Reading information from a cache is straight forward

What about writing?

- What if you're writing data that is already cached (**write-hit**)?
- What if the data is not in the cache (**write-miss**)?

Dealing with a write-hit

- **Write-through** - immediately write data back to memory
- **Write-back** - defer the write to memory for as long as possible

Dealing with a write-miss

- **write-allocate** - load the block into the cache and update
- **no-write-allocate** - writes directly to memory

Write-through caches are typically **no-write-allocate**

Write-back caches are typically **write-allocate**

Write-Back Caches

What happens if we need to evict a block that has been written into?

Need to write cached copy to lower-level

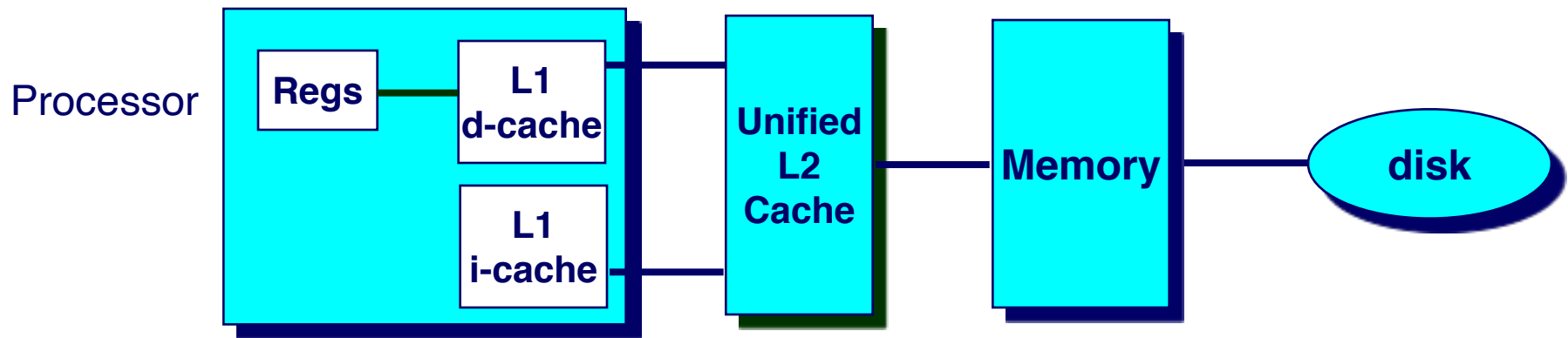
- Hence the name write-back

How do we know whether a block has been written to?

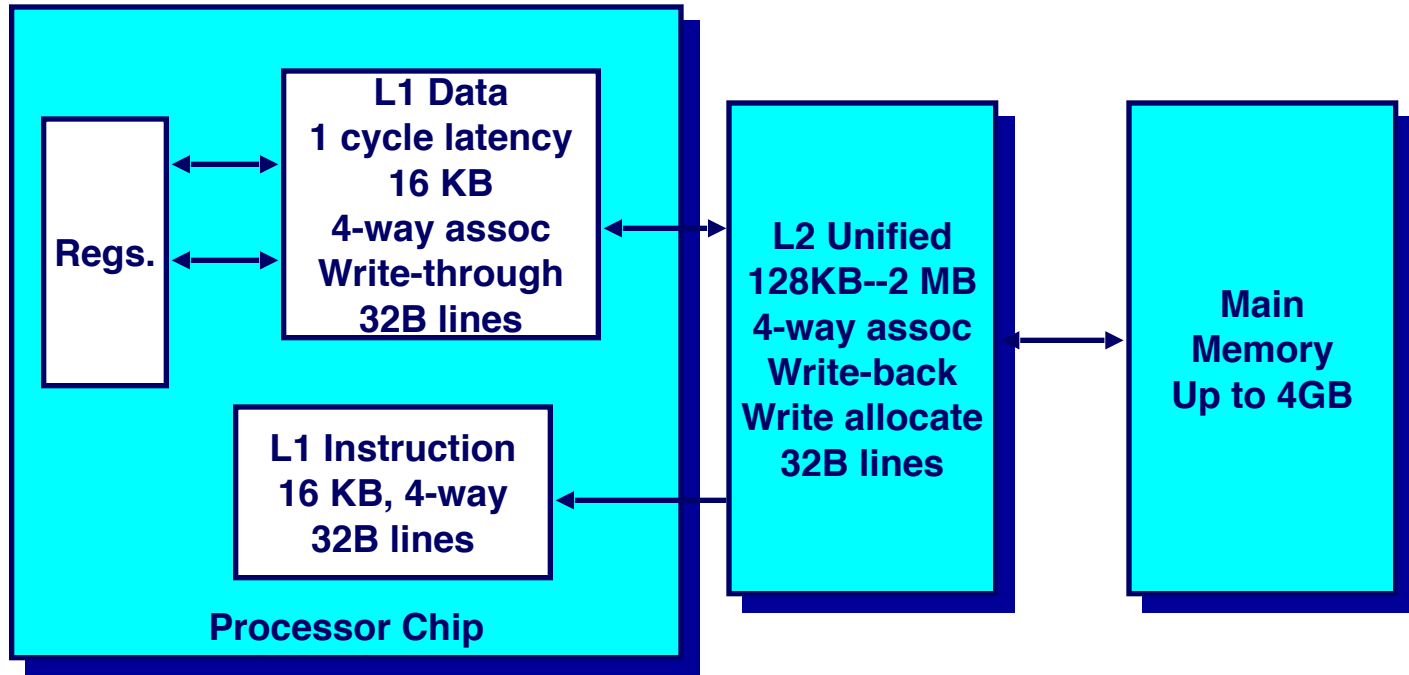
- Add a bit – called the dirty bit
- Set dirty bit when writing to a block
- When evicting a block, need to write to lower-level if dirty bit is set

Multi-Level Caches

Options: separate **data** and **instruction** caches, or a **unified** cache



Intel Pentium Cache Hierarchy



Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
 - 3-10% for L1
 - can be quite small (e.g., $< 1\%$) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
 - 1 clock cycle for L1
 - 3-8 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
 - Typically 25-100 cycles for main memory

Miss Types

Cold misses

- When a location is accessed for the 1st time
- Can reduce by increasing block size (leveraging spatial locality)

Conflict misses

- More blocks that map into a single set is concurrently active than can be stored in the set
- Can reduce by increase associativity

Capacity misses

- More blocks are active than can fit into the cache
- Working set

Pre-fetching is a technique that could be used to alleviate all three types of misses

- Predict what will be used next and pre-fetch before actually have a miss
 - Large block sizes is a form of prefetching
- If wrong, can worsen performance of cache

Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

- cold cache, 4-byte words, 4-word cache blocks

```
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;
}
```

Miss rate = $1/4 = 25\%$

```
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;
}
```

Miss rate = 100%

Matrix Multiplication Example

Major Cache Effects to Consider


- Total cache size
 - Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
 - Exploit spatial locality

Description:

- Multiply $N \times N$ matrices
- $O(N^3)$ total operations
- Accesses
 - N reads per source element
 - N values summed per destination
 - » but may be able to hold in register

```
/* ijk */  
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

*Variable **sum**
held in register*



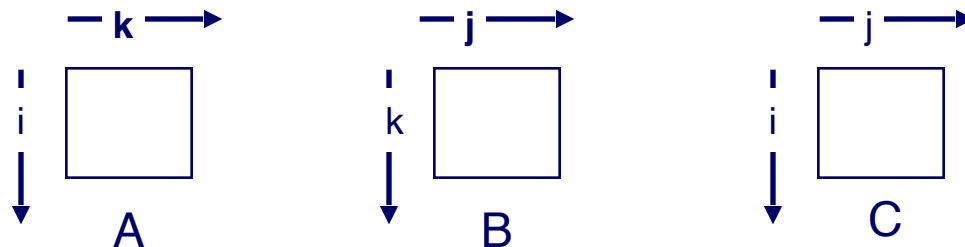
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = $32B$ (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

- for ($i = 0$; $i < N$; $i++$)
 - $\text{sum} += a[0][i];$
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality
 - compulsory miss rate = $4 \text{ bytes} / B$

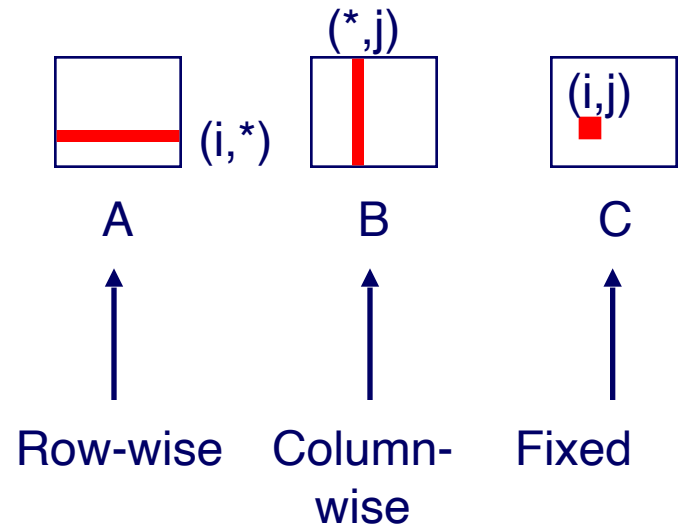
Stepping through rows in one column:

- for ($i = 0$; $i < n$; $i++$)
 - $\text{sum} += a[i][0];$
- accesses distant elements
- no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



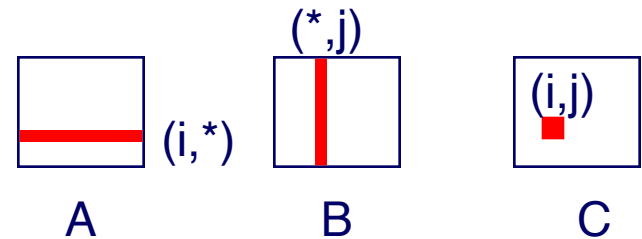
Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

Inner loop:



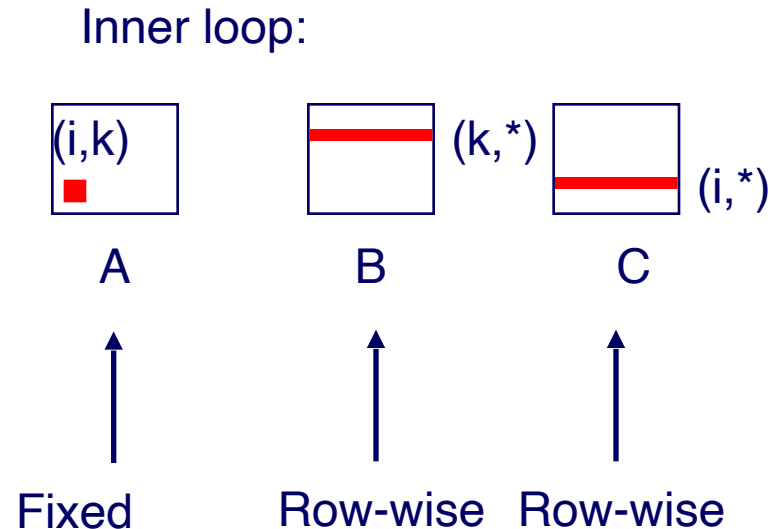
Row-wise Column-wise Fixed

Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



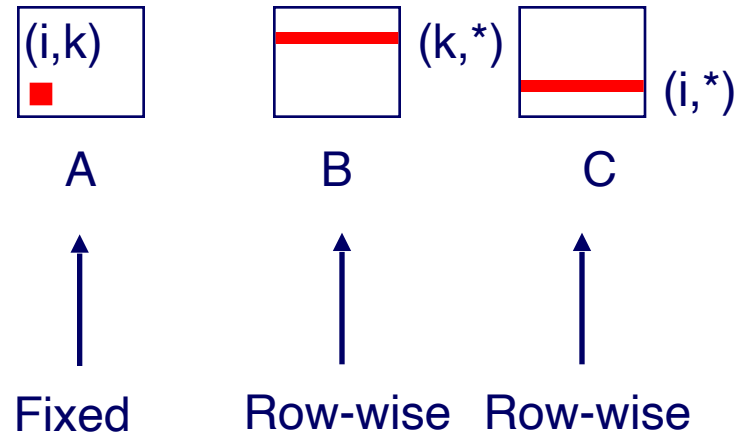
Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



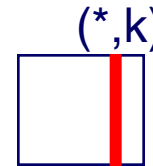
Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Matrix Multiplication (jki)

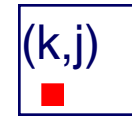
```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



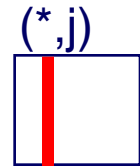
A

Column -
wise



B

Fixed



C

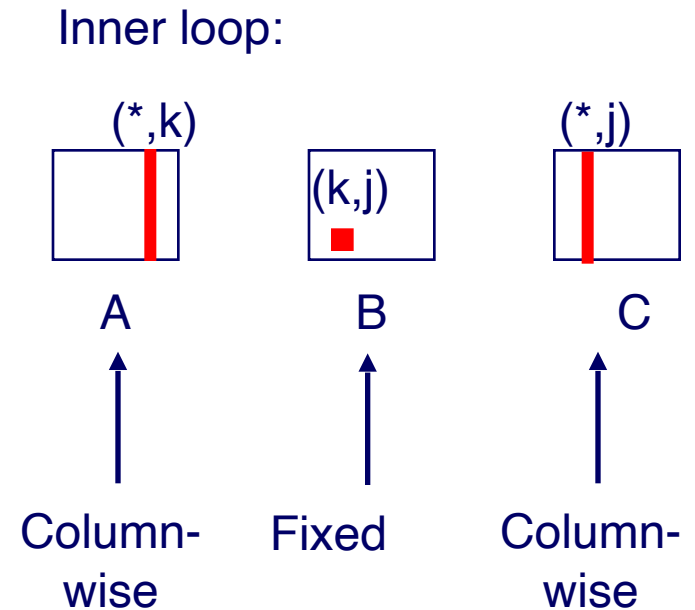
Column-
wise

Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] *  
b[k][j];  
        c[i][j] = sum;  
    }  
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++) {  
    for (i=0; i<n; i++)  
    {  
        r = a[i][k];  
        for (j=0; j<n;  
j++)  
            c[i][j] += r  
* b[k][j];  
    }  
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {  
    for (k=0; k<n; k++) {  
        r = b[k][j];  
        for (i=0; i<n; i+  
+)  
            c[i][j] +=  
a[i][k] * r;  
    }  
}
```

Concluding Observations

Programmer can optimize for cache performance

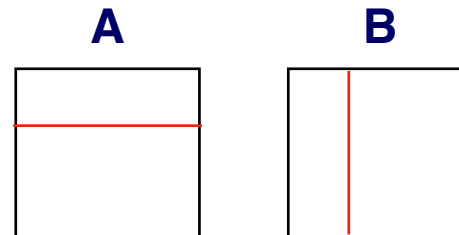
- How data structures are organized
- How data are accessed
 - Nested loop structure
 - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
 - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)

Matrix Transpose

```
for (r = 0; r < M; ++r) {  
    for (c = 0; c < N; ++c) {  
        B[c][r] = A[r][c];  
    }  
}
```



Problem: if M (number of rows/column height) is too large,
then every access of B is a cache miss

Solution: break A and B into sub-matrices which can fit
an entire column into the cache

But what if we don't know the cache size?