Lecture 10:
Cache Design

Announcements

- Midterm Grades Released
  - If you have any questions, email me or any of the TAs
- Project 2 bomblab due tommorow (July 28th)
  - Bomblab server will be stopped at the due date and time
- Project 3 will be released tomorrow
  - 2 Weeks, no extension
  - Due August 11th (Day before Final)
  - We'll go over the project next recitation (this Wednesday after you have time to read the project description)
- Lecture Today:
  - Cache Design
- Rest of lecture time / recitation:
  - Questions on Cache Design

Memory

- So far, we 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.
  - Different memories at different levels of the computer
  - Each vary in speed, size, and cost

Random-Access Memory (RAM)

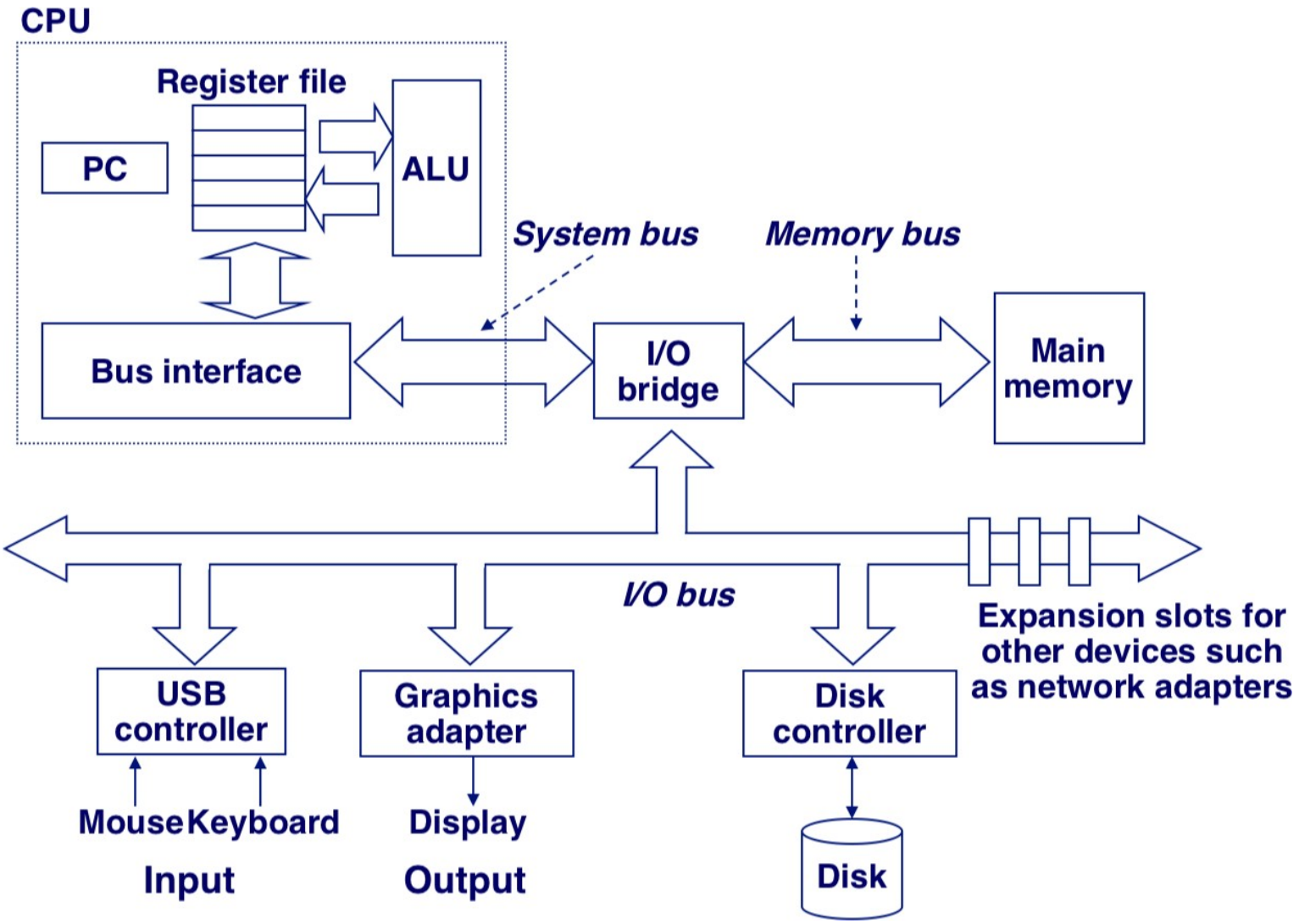
- RAM is packaged as a chip.
- Basic storage is a cell.
- Static RAM (SRAM)
  - Retains value indefinitely, as long as it is kept powered.
  - Insensitive to electrical noise disturbance.
- Dynamic RAM (DRAM)
  - Values must be refreshed every 10-100ms
  - Sensitive to disturbances

Memory speeds

- Processor Speeds: 1 GHz processor speed is 1 nsec cycle
- Memory Speed

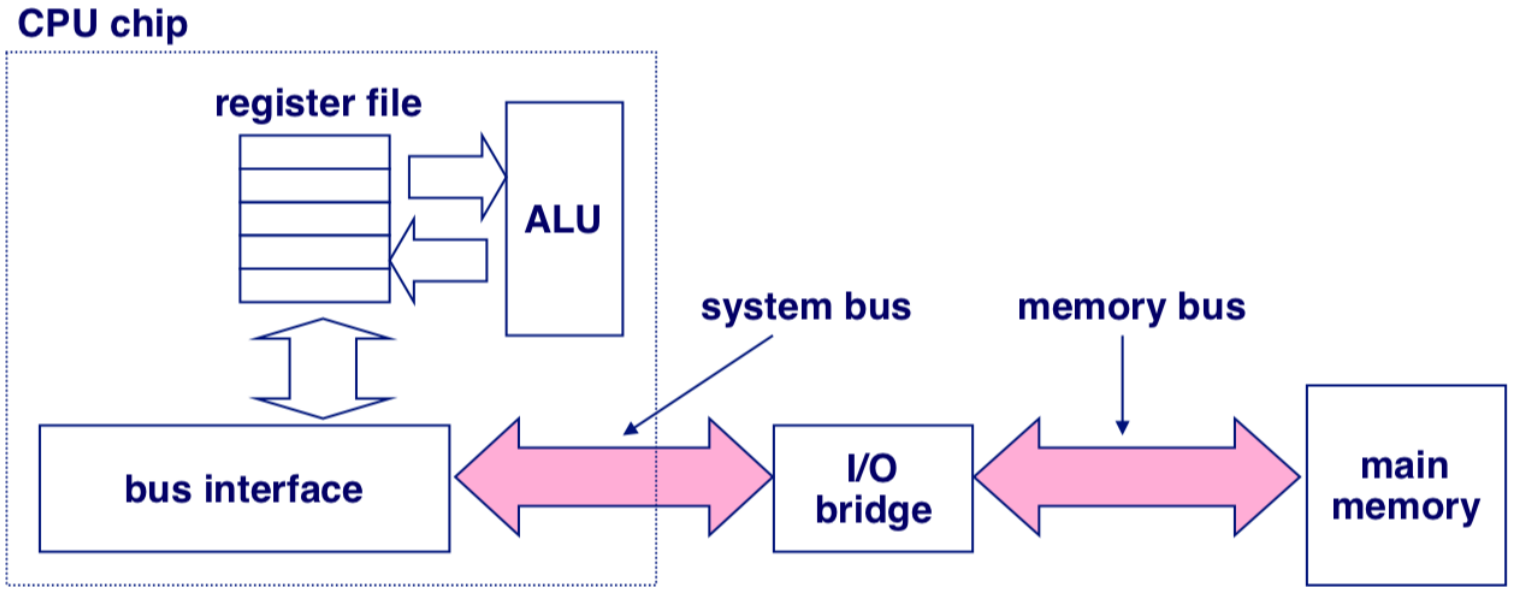
DIMM Module	Clock Speed	Bus Speed	Transfer Rate
PC1600 DDR200	100 MHz	200 MHz	1600MB/s
PC2100 DDR266	133 MHz	266 MHz	2133 MB/s
PC2400 DDR300	150 MHz	300 MHz	2400 MB/z

Machine Architecture



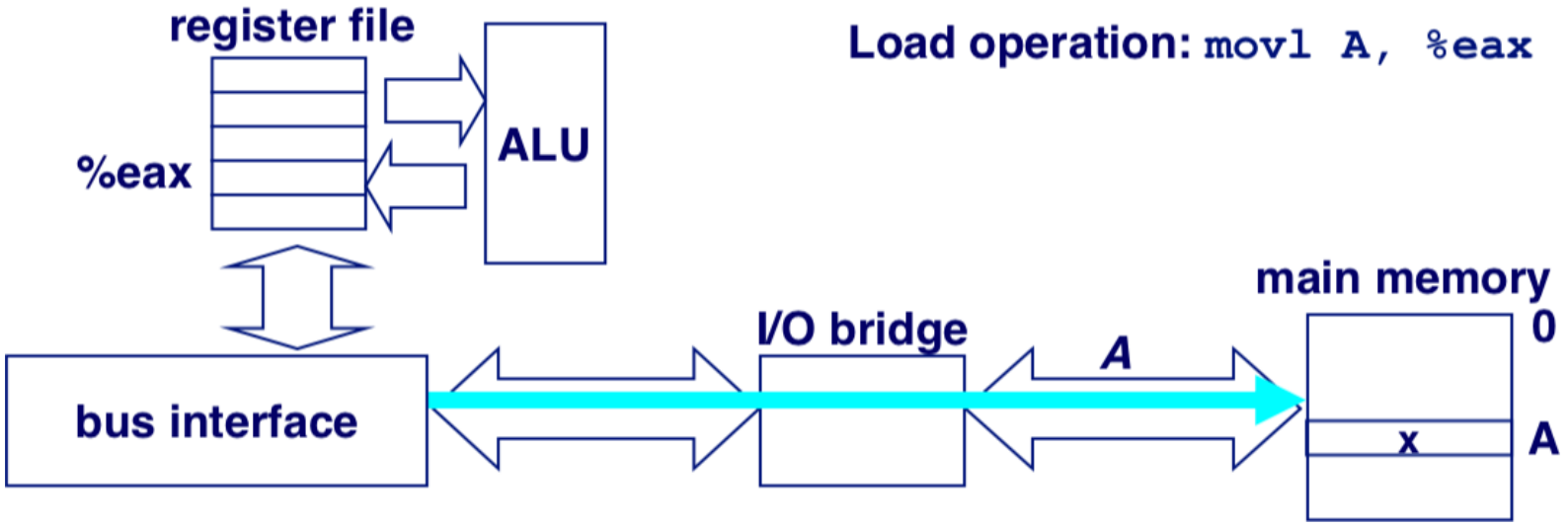
System/Memory Bus

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



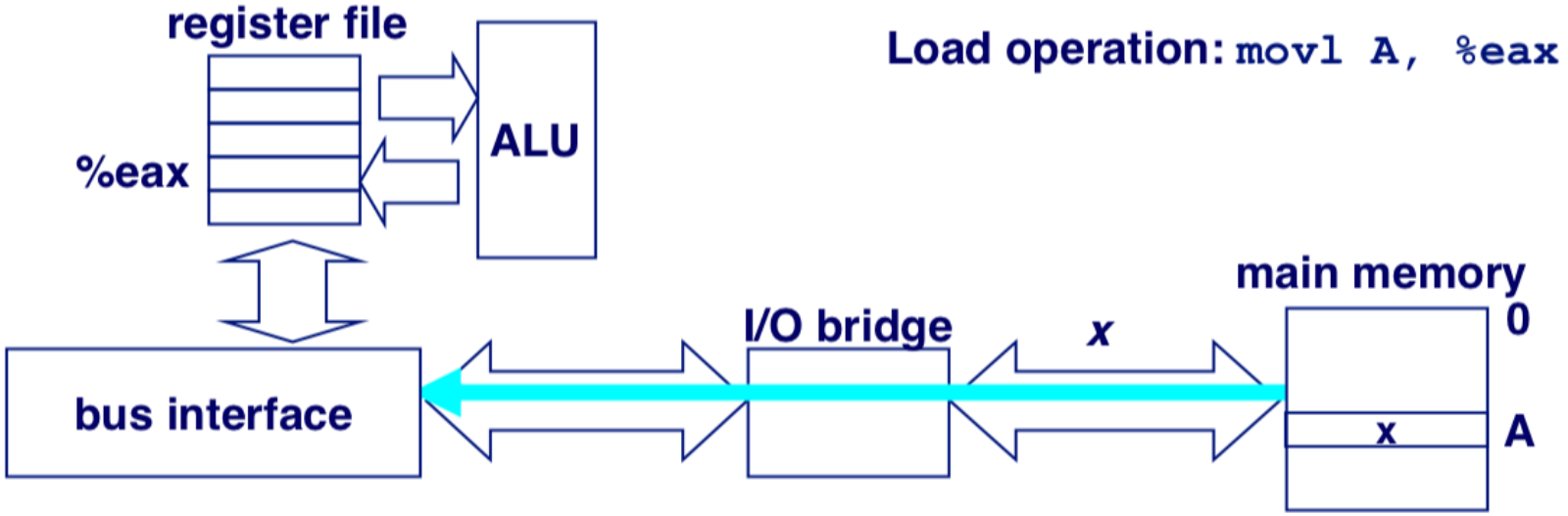
Memory Read Transaction

- CPU places address A on the memory bus



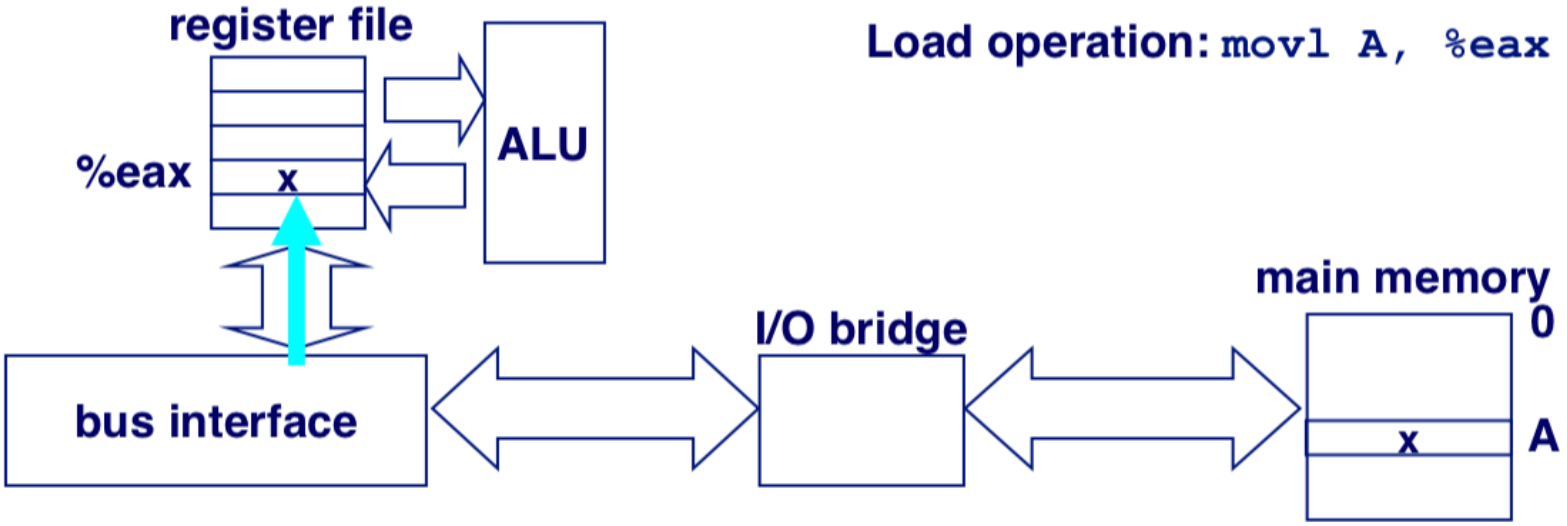
Memory Read Transaction

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



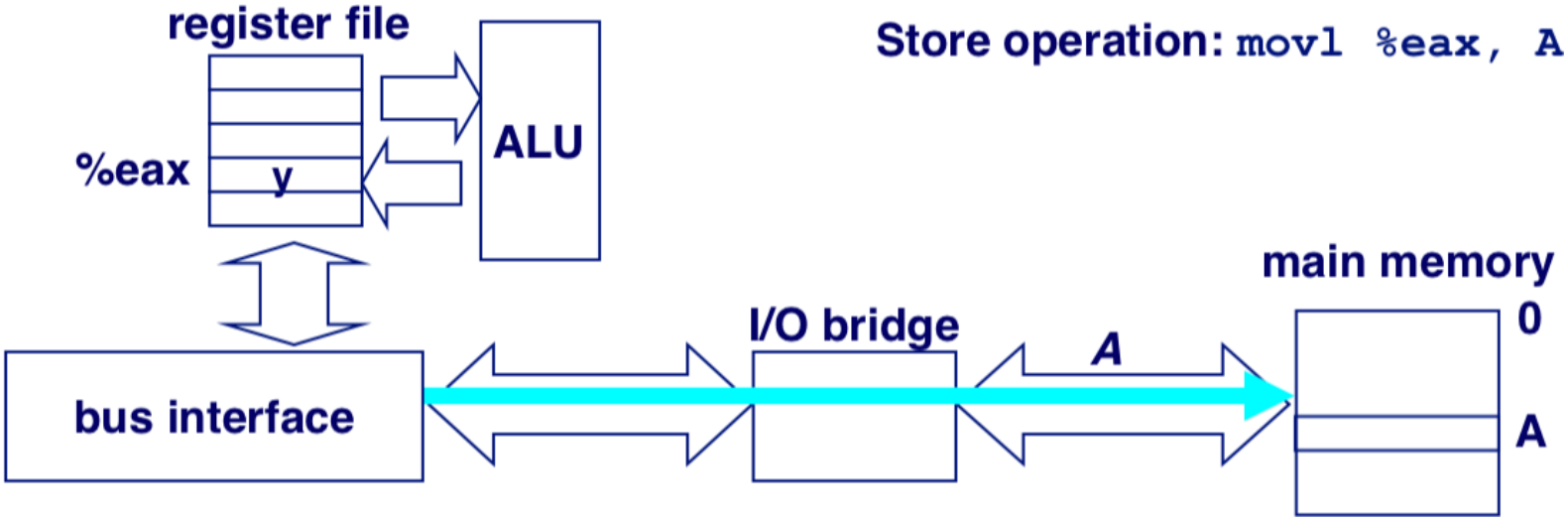
Memory Read Transaction

- CPU reads word x from the bus and copies it into %eax

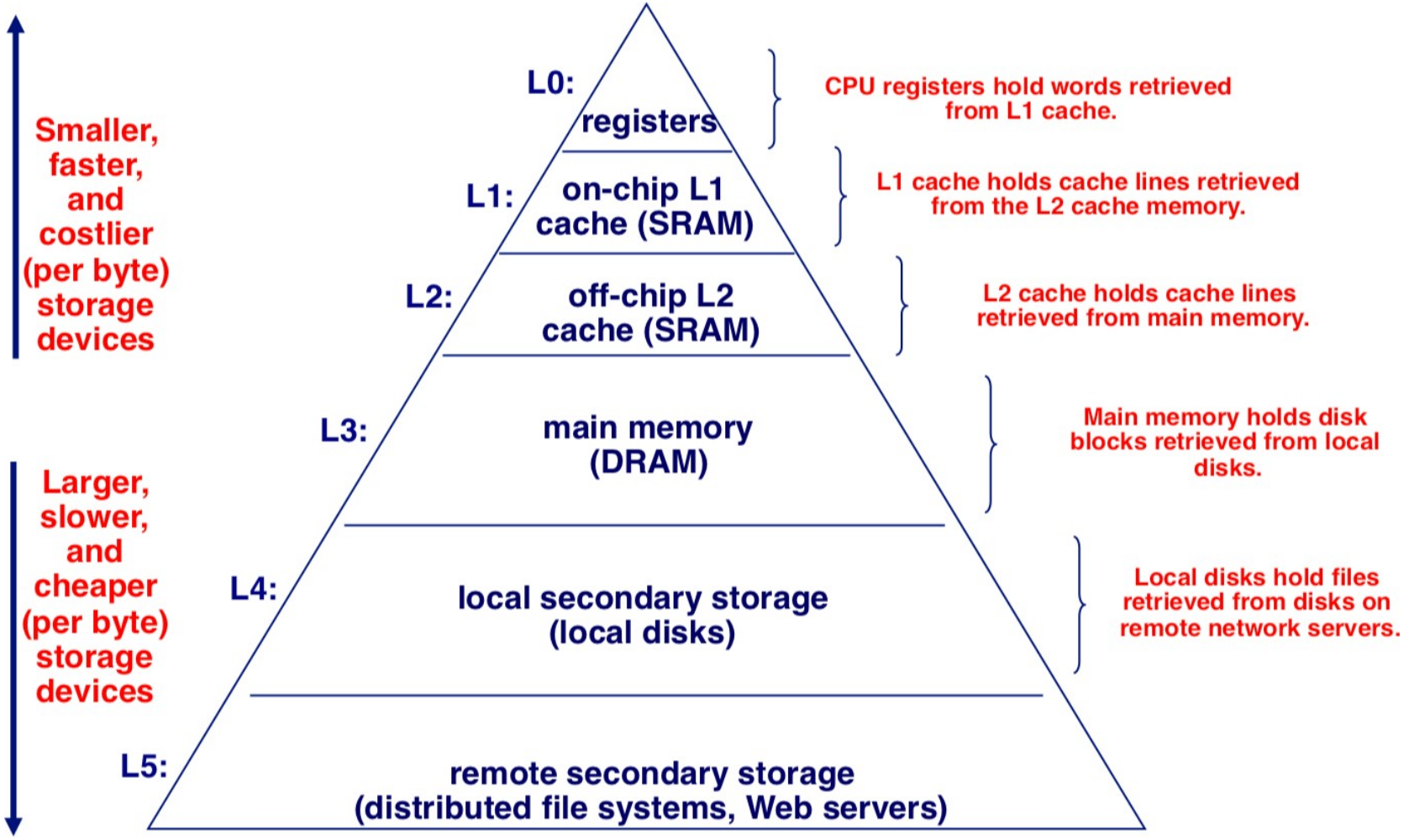


Memory Write Transaction

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

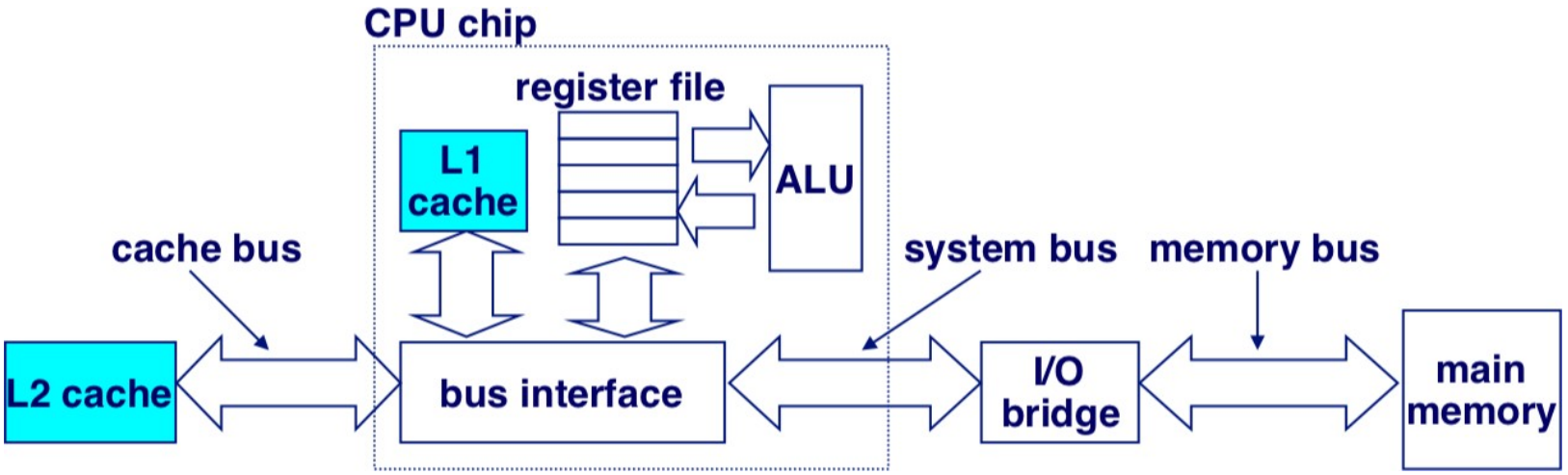


Memory Hierarchy



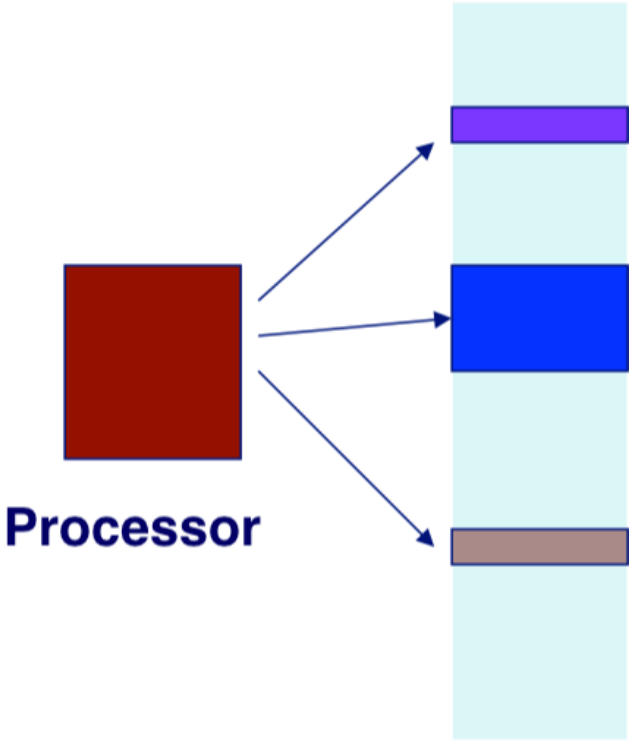
Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
- CPU looks first for data in L1, then in L2, …, then in main memory



Locality

- Memory references are bunched together
  - Small portion of address space 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 the ones recently accessed in near future

Sources of Locality

- Temporal locality (Same data will accessed again)
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality (adjacent memory data will be accessed)
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks

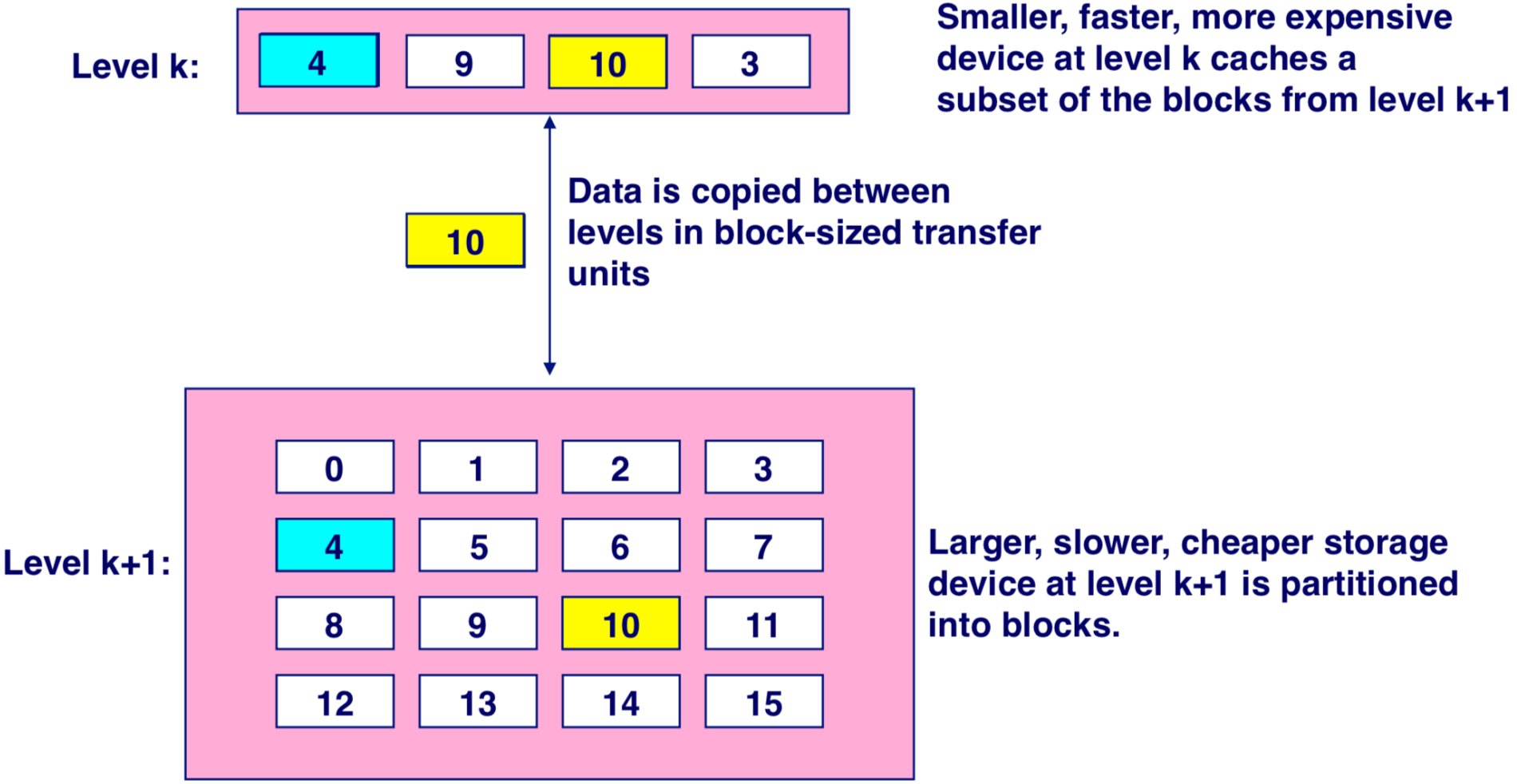
Why Is Locality Good?

- Can Address the gap between CPU and RAM speed
- Spatial and temporal locality implies a portion of overall address space can fit in high speed small memory
- CPU can access instructions and data from this high speed memory
- Small high speed memory can make computer faster and cheaper

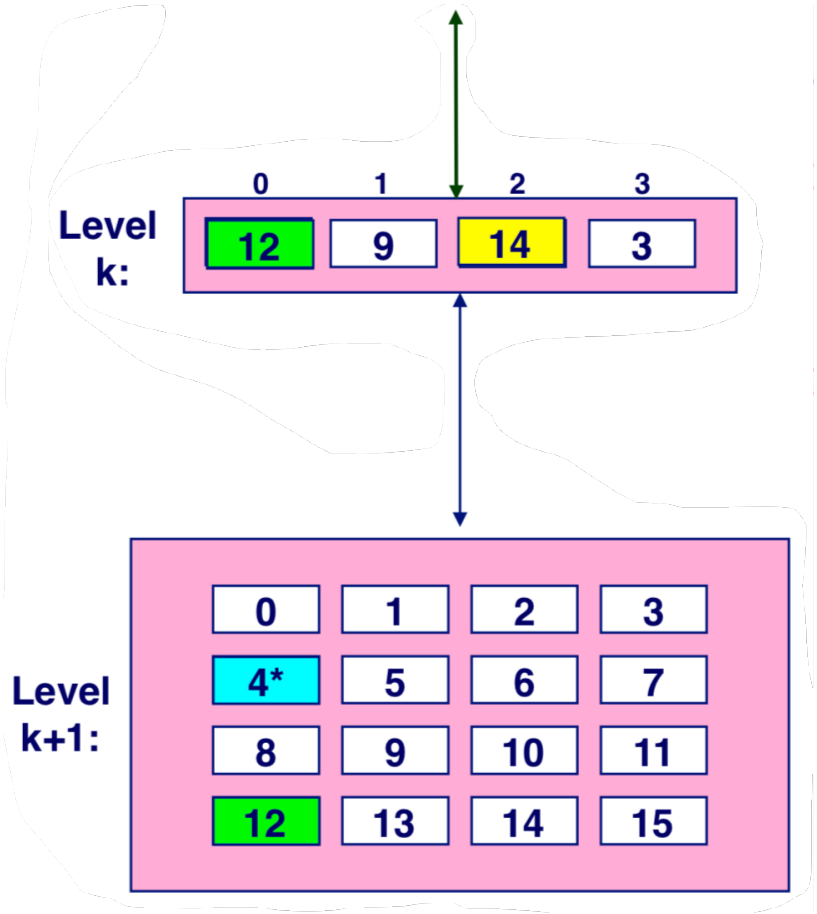
Why Is Locality Good?

- This is **caching**!

Caching in a Memory Hierarchy



General Concept

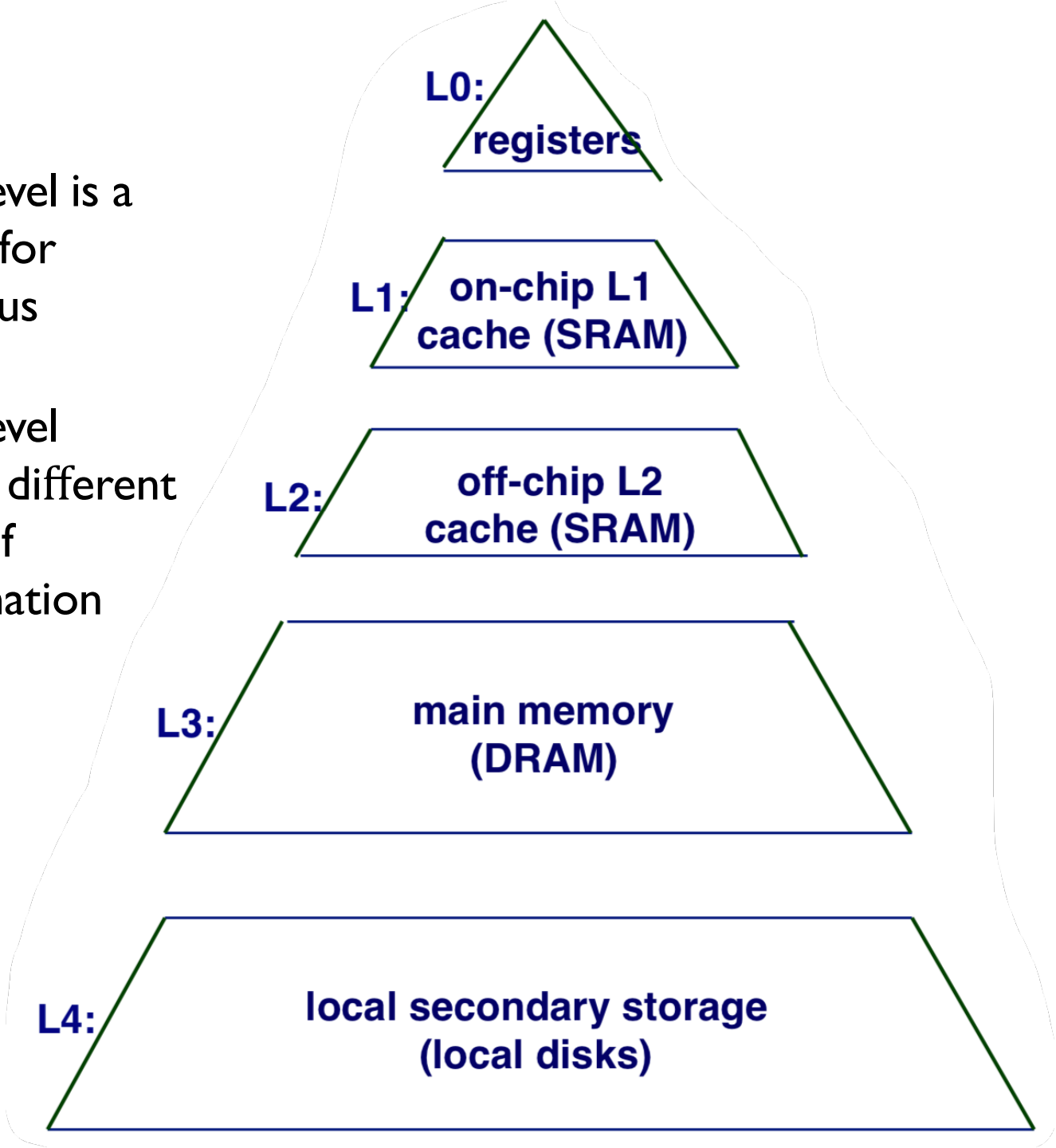


- Cache hit:
  - Program finds block b in the cache at level k (block 12, 14)
- Cache miss:
  - b is not at level k, so fetch from level k+1 (block 4)
  - If level k cache is full, we must evict a block. Which one do we evict?

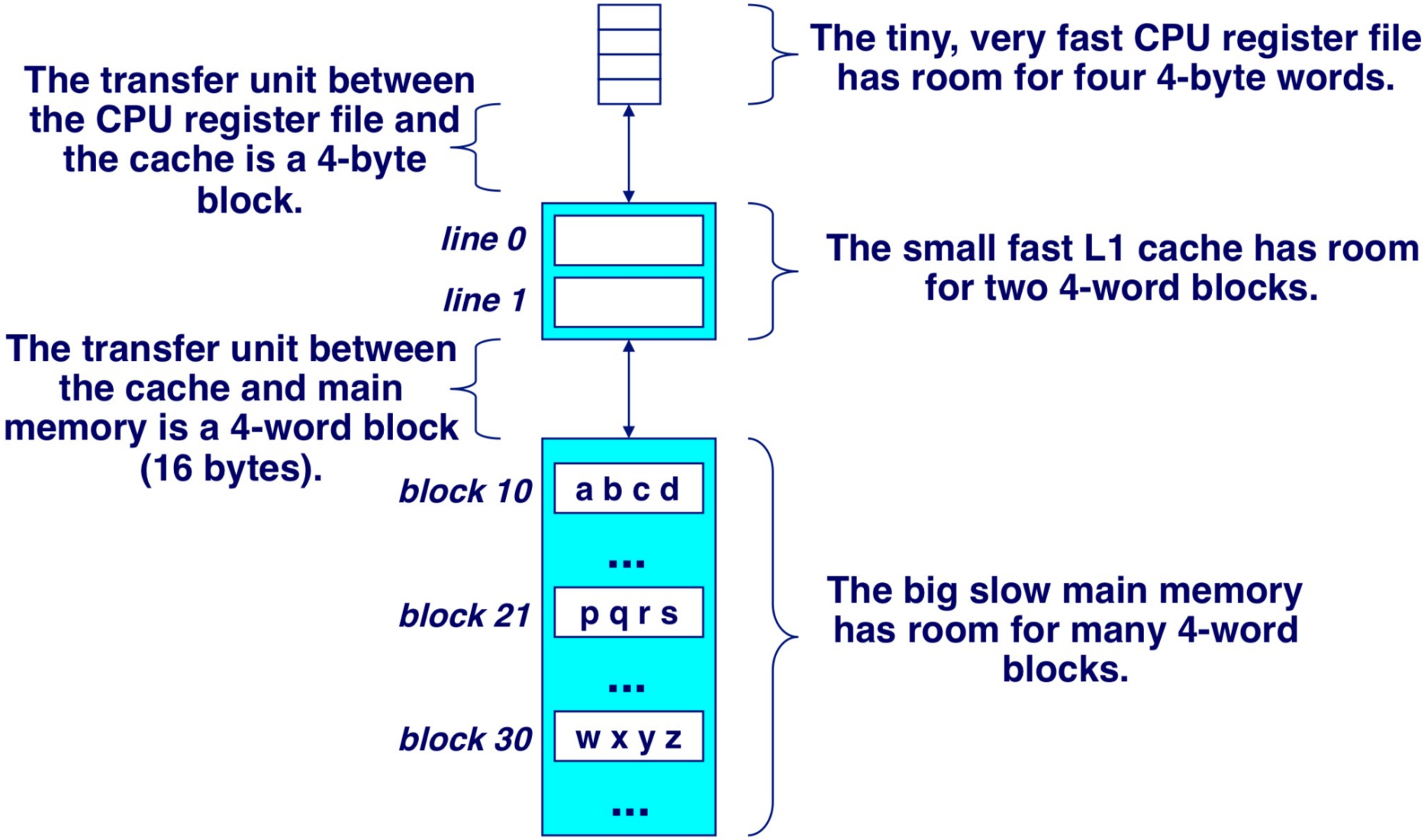
Cache Miss

- Cold miss
  - Cold miss occurs when a memory location is accessed for the 1st time
- Conflict miss
  - Conflict miss occurs when the cache is not big enough, so the previously accessed block has been replaced and we try to access the replaced block again.
- Capacity miss
  - Occurs when the working set is larger than the cache

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



L1 Cache Example

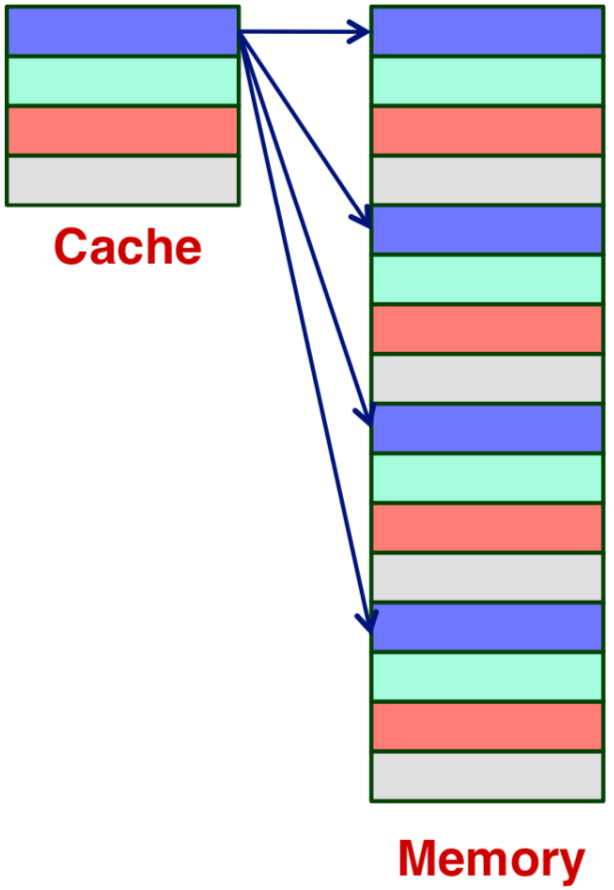


Cache Content

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data.
- You are essentially allowing a smaller region of memory to hold data from a larger region.
- What kind of information do we need to keep:
  - The actual data
  - Where the data comes from
  - If data is even considered valid

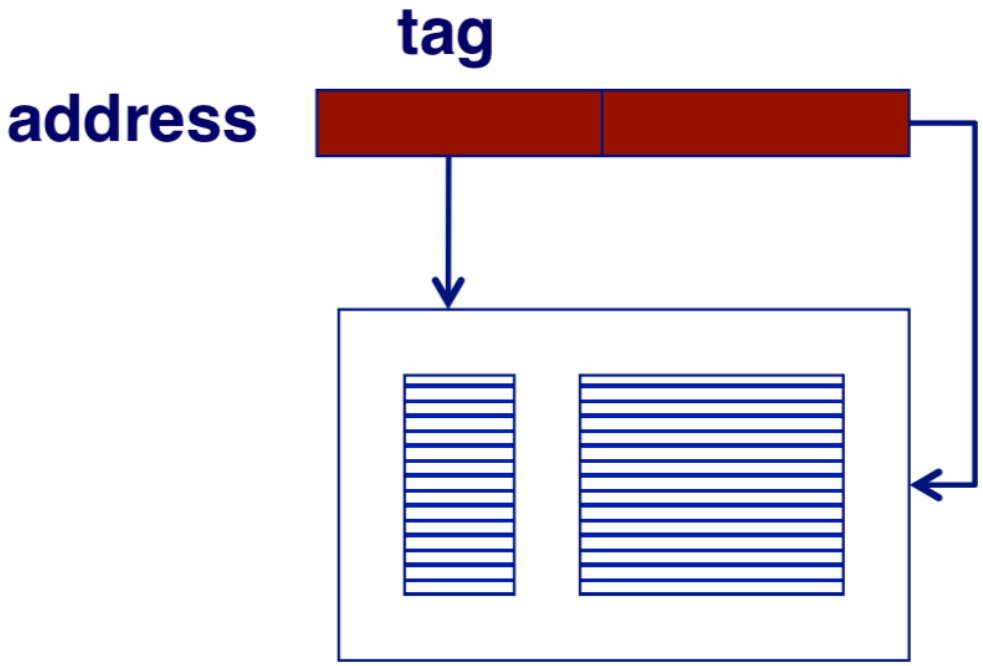
Cache Mapping

- Multiple locations in memory map to same location in cache
- In addition to content, cache must keep which entry it is caching



Finding data in cache

- Part of memory address applies to cache
- Remaining stored as "tag"
- If tag matches, hit
- If no match, miss

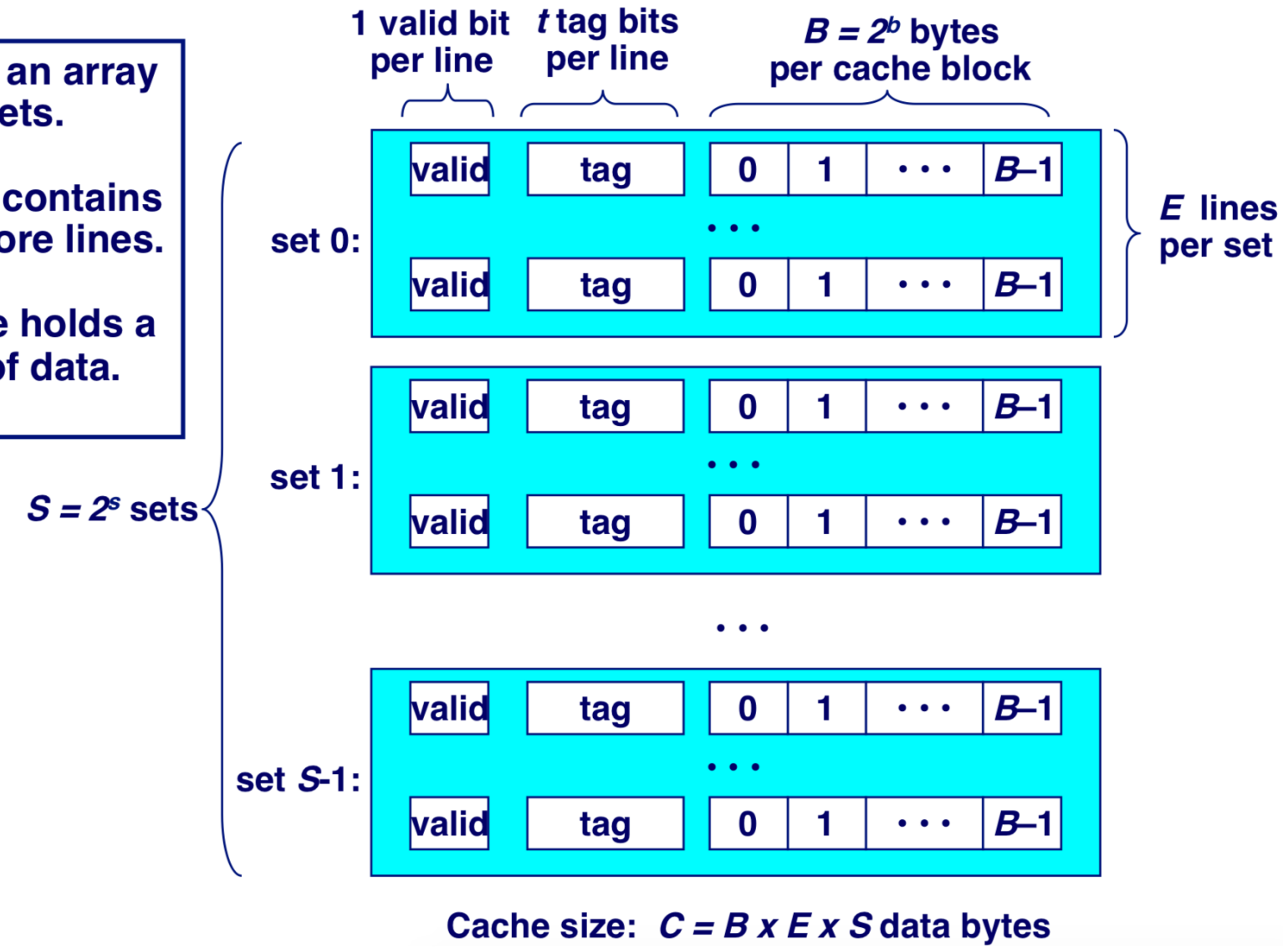


General Structure of Cache

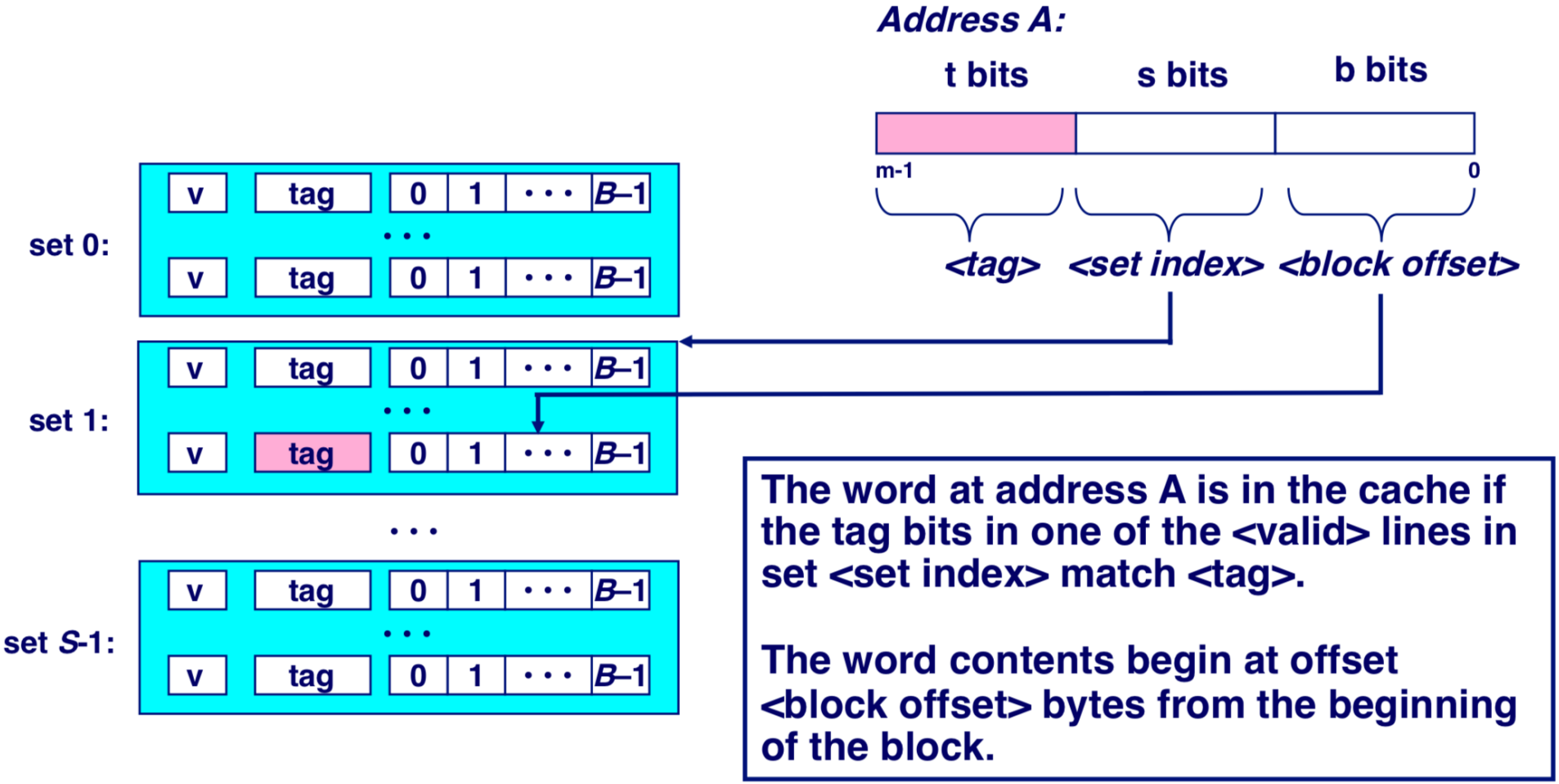
Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.



Addressing Caches

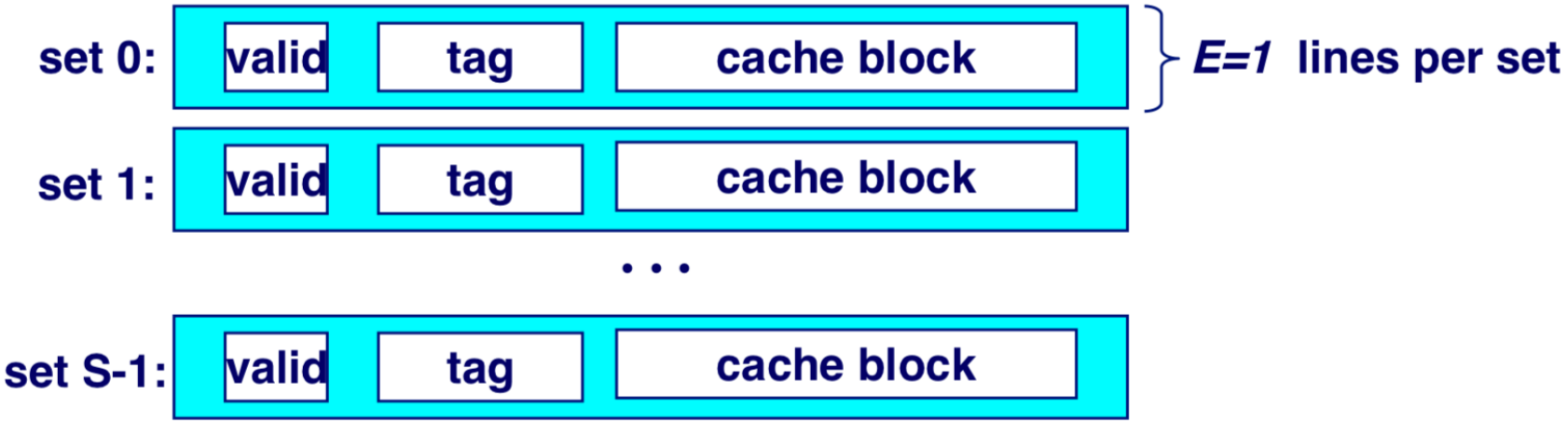


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

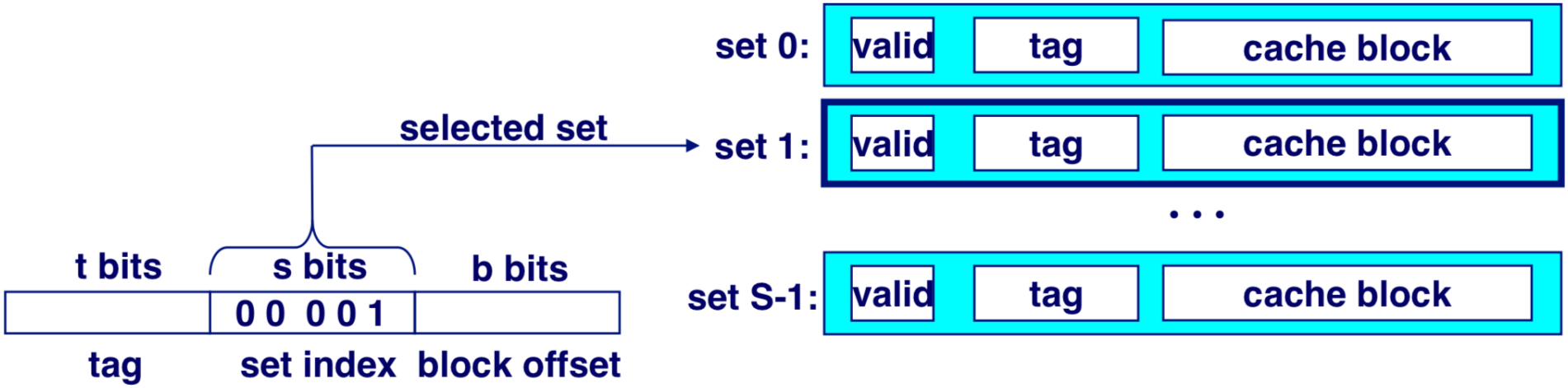
Direct-Mapped Cache

- Simplest kind of cache
- Exactly one line per set



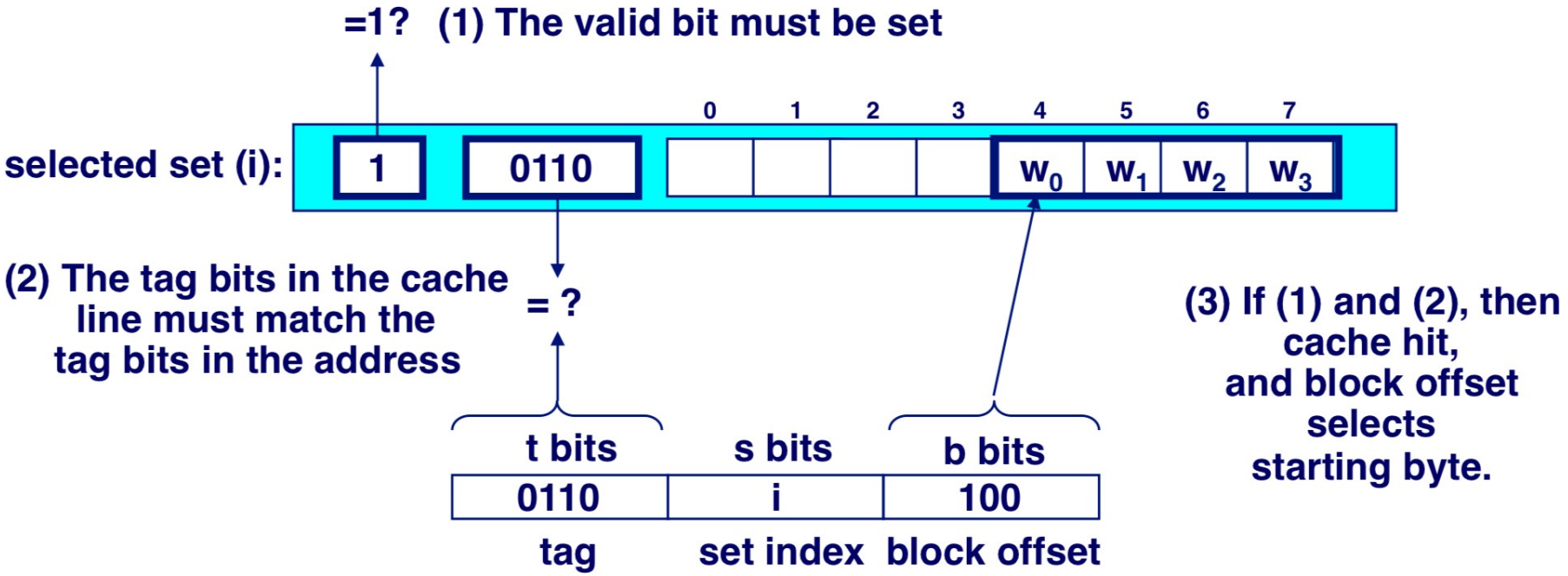
Accessing Direct-Mapped Caches

- Set selection: Use the set index bits to determine the set of interest

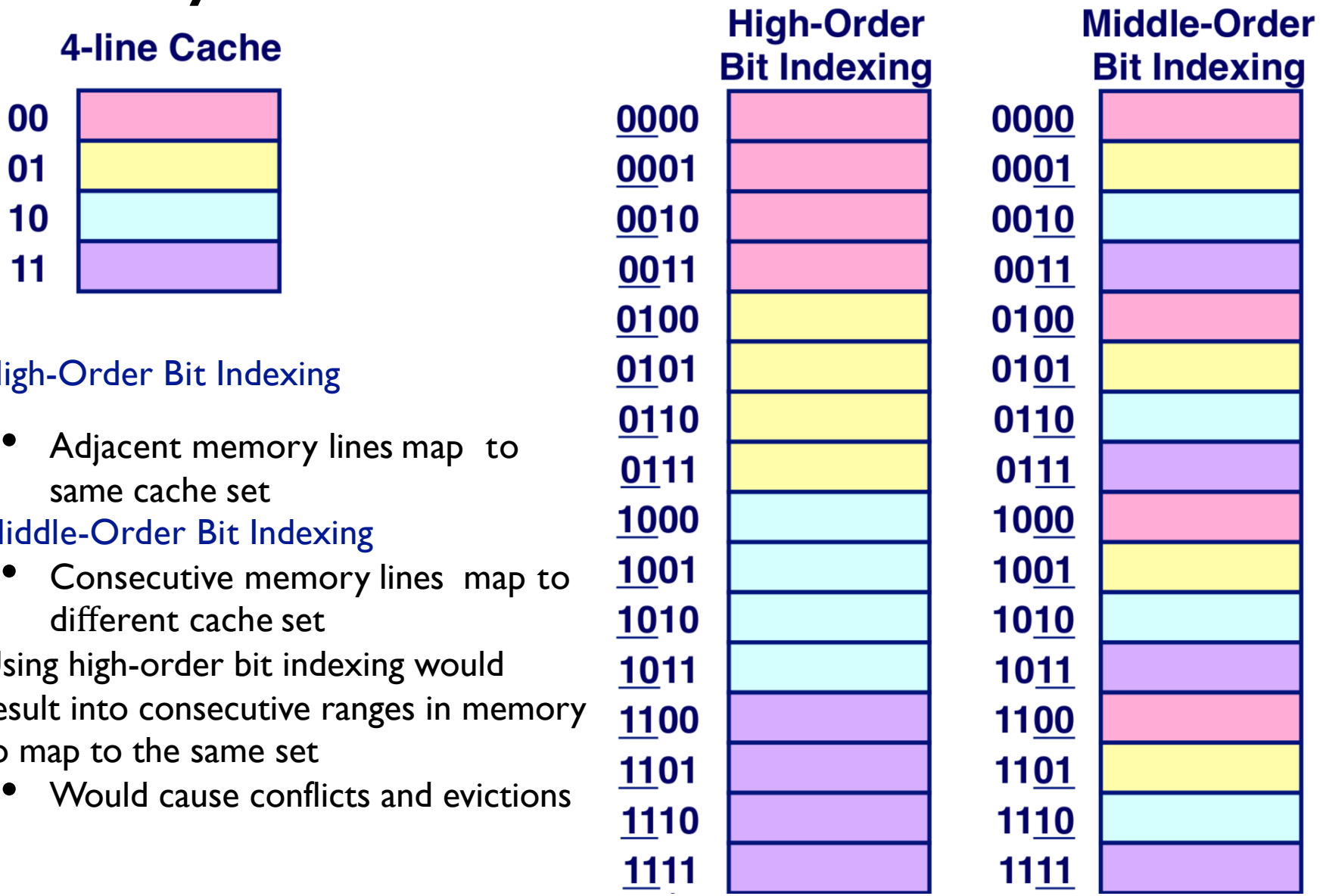


Accessing Direct-Mapped Caches

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

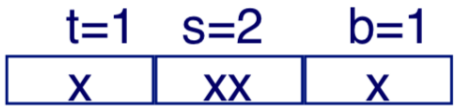


Why Use Middle Bits as Index?



- High-Order Bit Indexing
  - Adjacent memory lines map to same cache set
- Middle-Order Bit Indexing
  - Consecutive memory lines map to different cache set
- Using high-order bit indexing would result into consecutive ranges in memory to map to the same set
  - Would cause conflicts and evictions

Direct-Mapped Cache Sim.

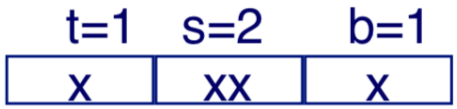


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA

Direct-Mapped Cache Sim.

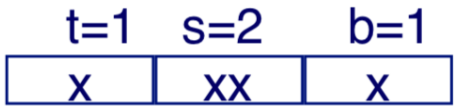


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA

Direct-Mapped Cache Sim.

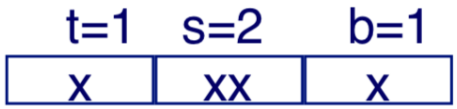


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	0	M[0-1]

Direct-Mapped Cache Sim.

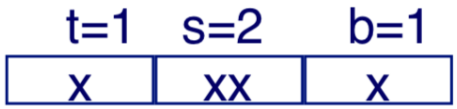


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	0	M[0-1]

Direct-Mapped Cache Sim.

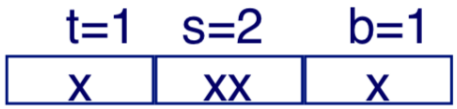


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	0	M[0-1]

Direct-Mapped Cache Sim.

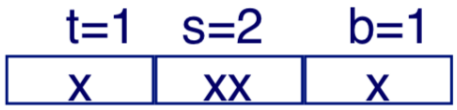


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	0	M[0-1]
1	1	M[12-13]

Direct-Mapped Cache Sim.

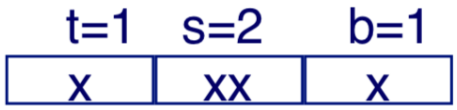


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	0	M[0-1]
1	1	M[12-13]

Direct-Mapped Cache Sim.

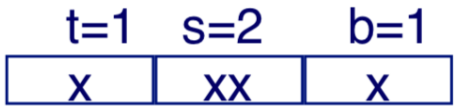


- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

V	TAG	DATA
1	1	M[8-9]
1	1	M[12-13]

Question



- M=4 bit addresses, B=2 bytes/block, S=4 sets, E=1 line/set
- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

A:

V	TAG	DATA
1	0	M[0-1]
1	1	M[12-13]

B:

V	TAG	DATA
1	1	M[8-9]
1	0	M[0-1]
1	1	M[12-13]

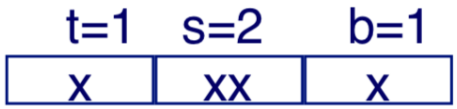
C:

V	TAG	DATA
1	1	M[8-9]
1	0	M[0-1]

D:

V	TAG	DATA
1	1	M[8-9]
1	1	M[12-13]
1	0	M[0-1]

Question



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

- Address traces (reads):
  0 [0000_2], 1 [0001_2], 13 [1101_2], 8[1000_2], 0[0000_2]

A:

V	TAG	DATA
1	0	M[0-1]
1	1	M[12-13]

B:

V	TAG	DATA
1	1	M[8-9]
1	0	M[0-1]
1	1	M[12-13]

C:

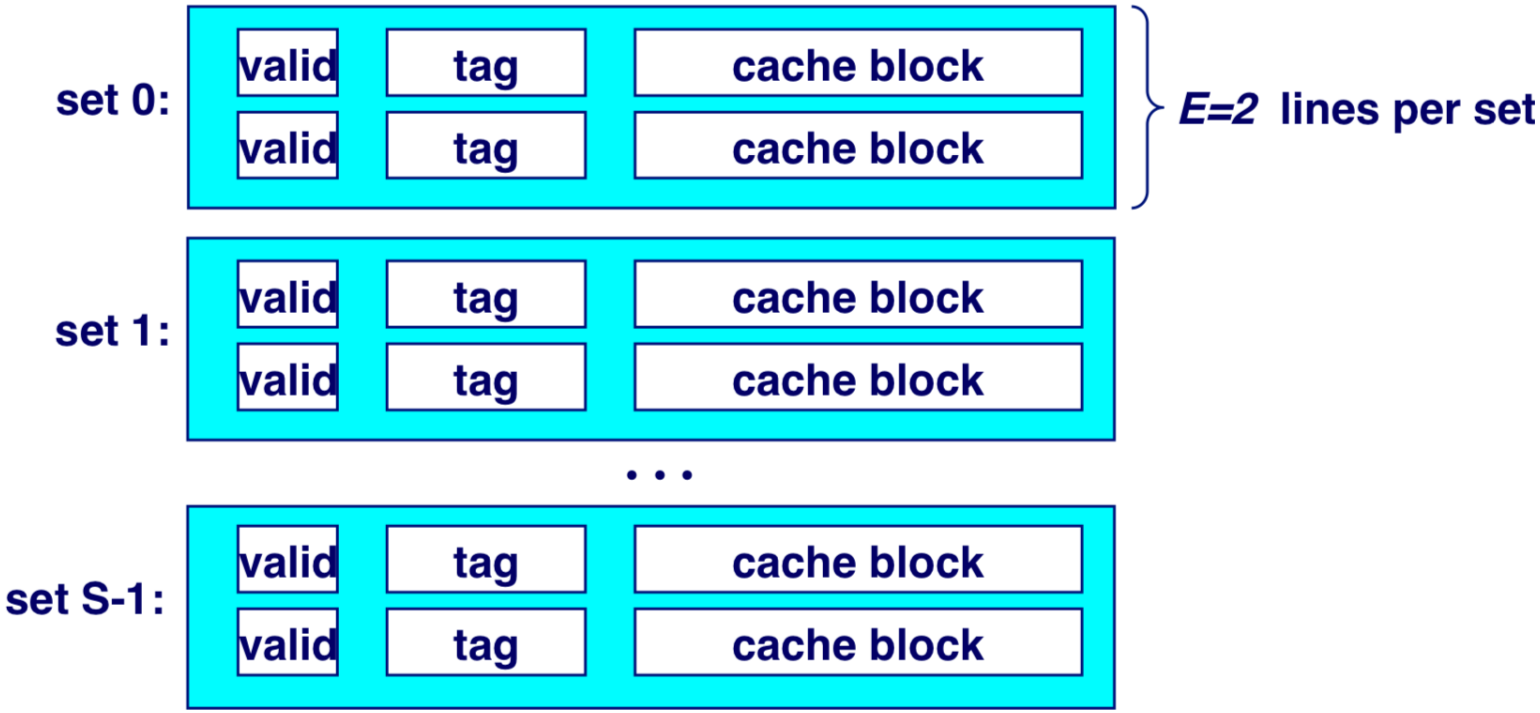
V	TAG	DATA
1	1	M[8-9]
1	0	M[0-1]

D:

V	TAG	DATA
1	1	M[8-9]
1	1	M[12-13]
1	0	M[0-1]

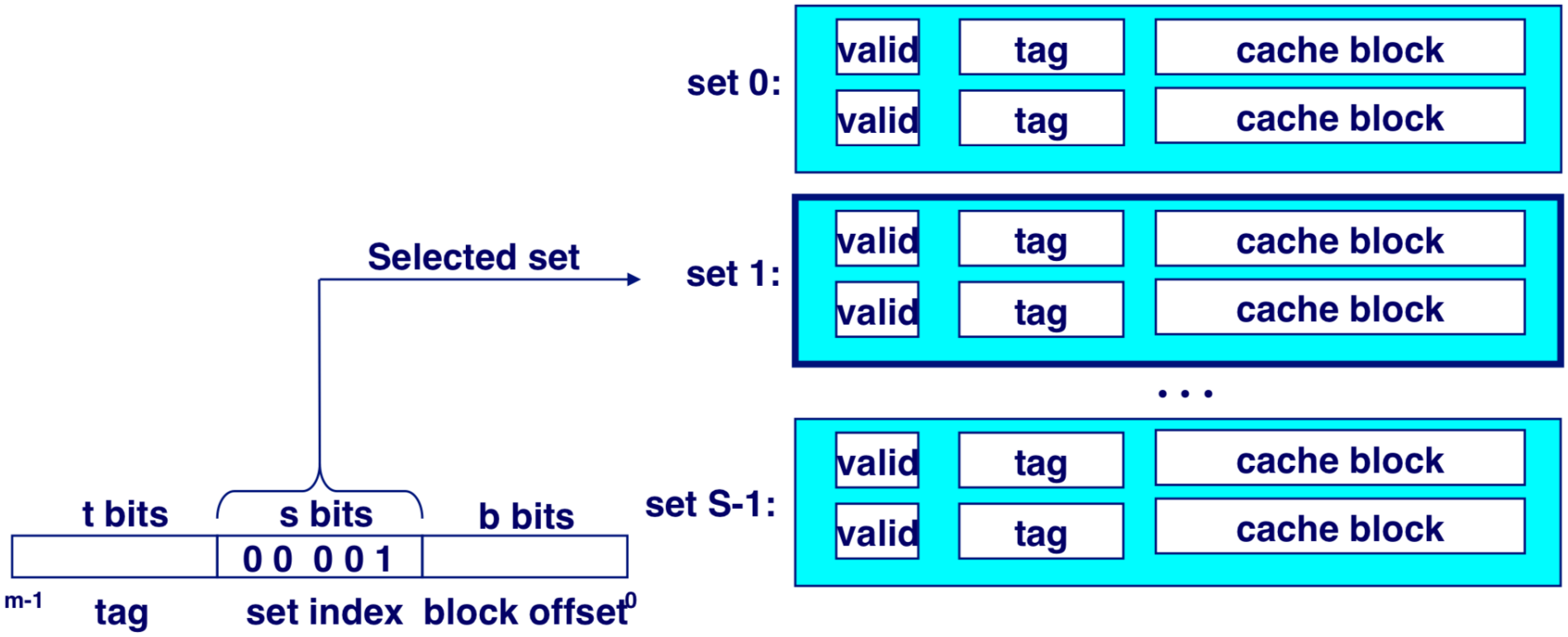
Set Associative Caches

- More than one line per set



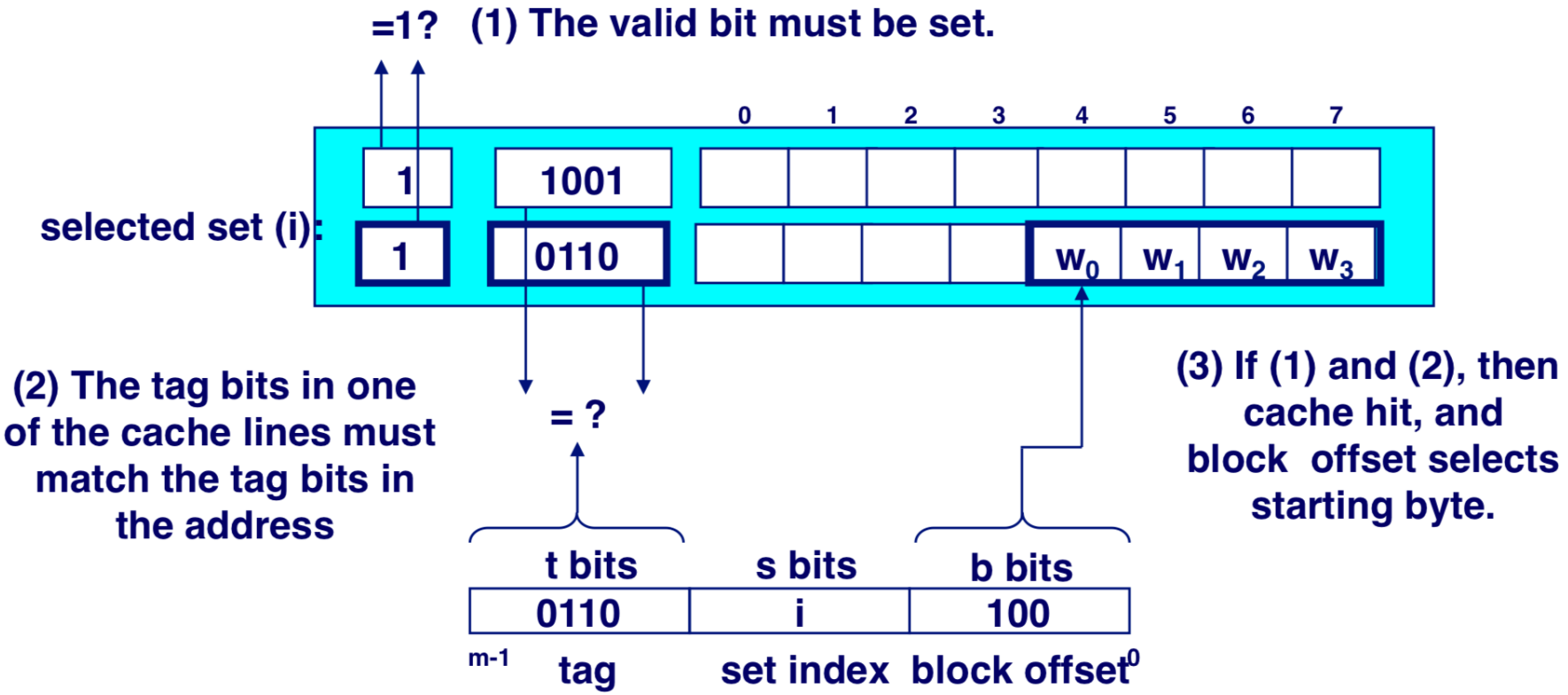
Accessing Set Associative Caches

- Set selection: identical to direct-mapped cache



Accessing Set Associative Caches

- Line matching: Must compare the tag in each valid line



Replacement

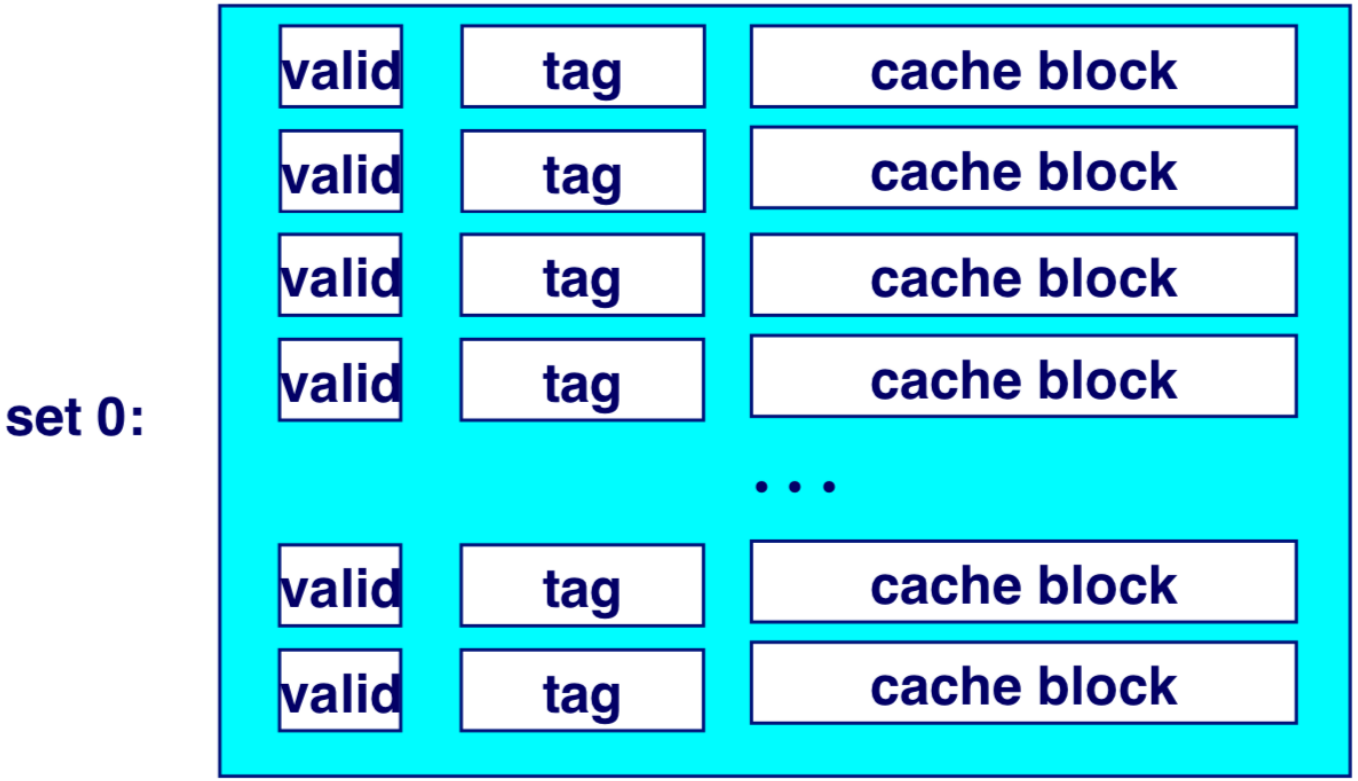
- What if we have a cache miss but all entires in the set are valid?
- Direct-mapped cache: no decision
  - Discard current content and bring in the new line
- Set Associate cache: which line to evict?
  - Need a replacement algorithm

Replacement

- If we can predict the future correctly, can you think of an optimum algorithm?
- But we can't, so we need to approximate:
  - FIFO: First-In-First-Out
  - LRU: Least Recently Used
  - Random: Select victim from set randomly

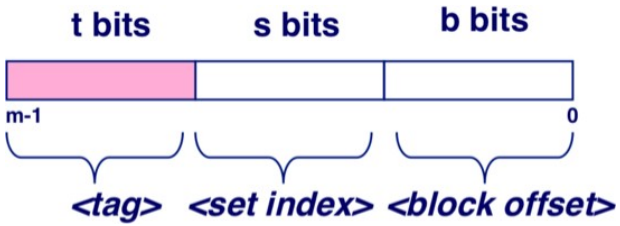
Fully Associative Caches

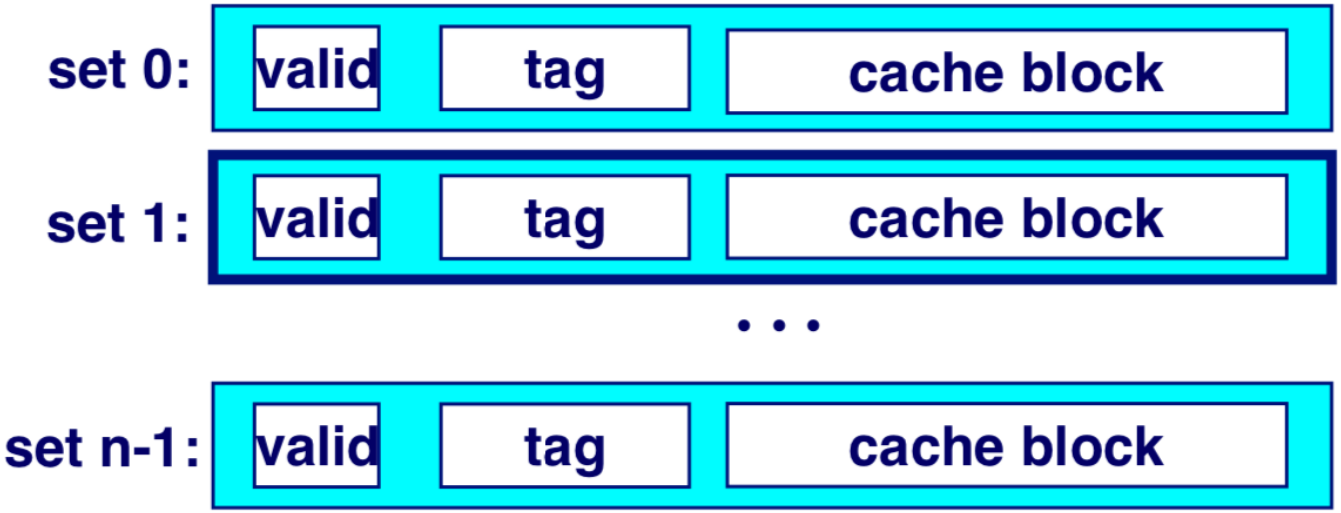
- There is only one set. Set selection is trivial



Calculating sets, tag, offset: (Direct Mapped)

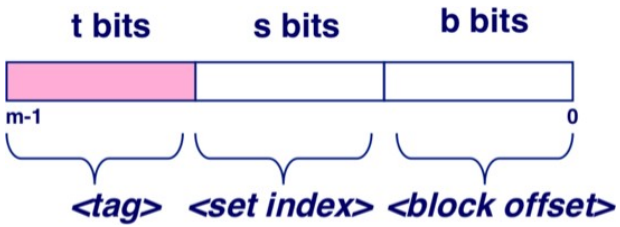
- Direct Mapped, 32 bit address, 64KB cache, 32 byte block
  - How many sets?
  - How many bits for the tag?
  - How many bits for the offset?

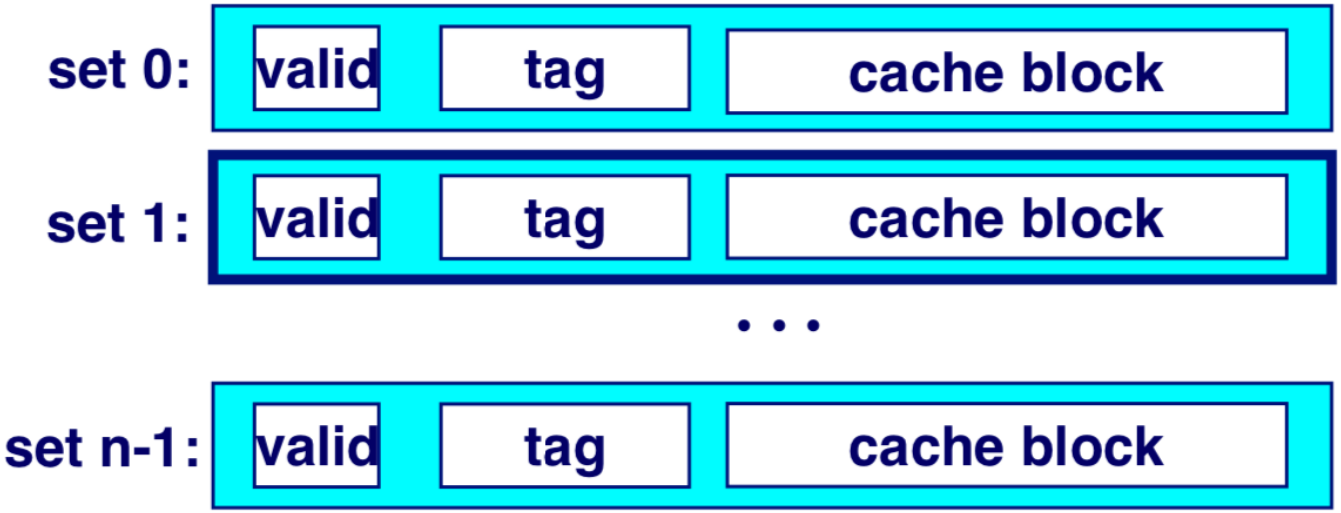




Calculating sets, tag, offset: (Direct Mapped)

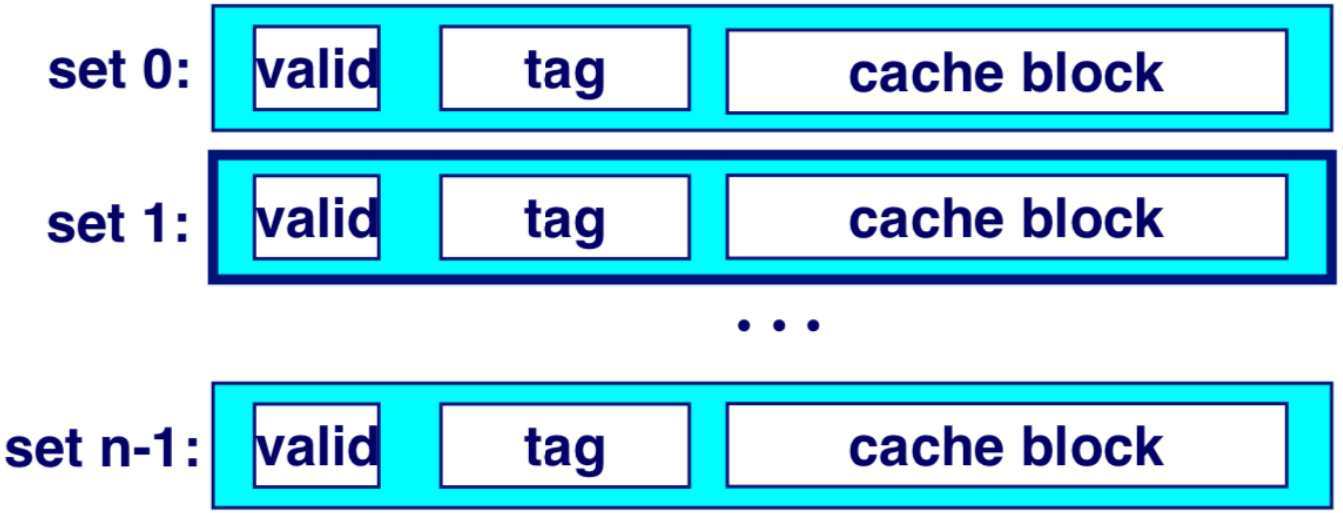
- Direct Mapped, 32 bit address, 64KB cache, 32 byte block
  - How many sets?
  - How many bits for the tag?
  - How many bits for the offset? $\log_2(32) = 5$





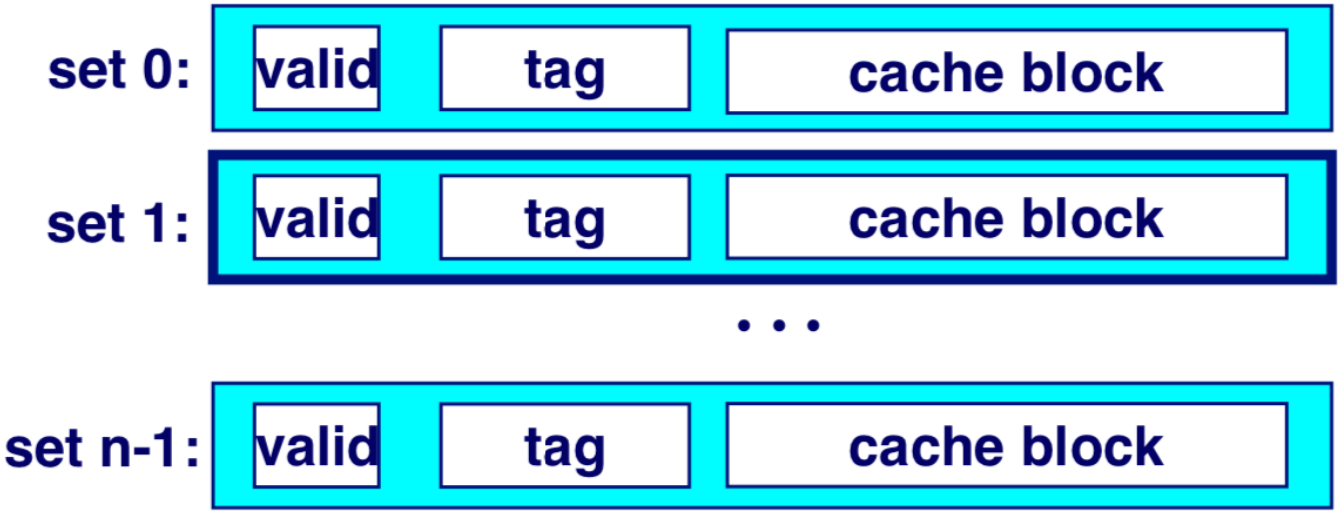
Calculating sets, tag, offset: (Direct Mapped)

- Direct Mapped, 32 bit address, 64KB cache, 32 byte block
  - How many sets? 64KB / 32 = 2048 = 2^{11}
  - How many bits for the tag?
  - How many bits for the offset? $\log_2(32) = 5$



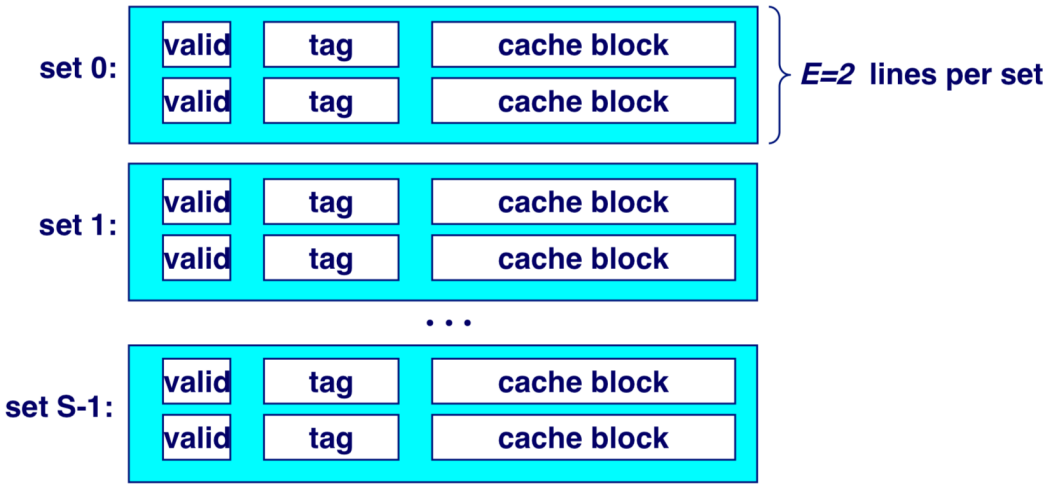
Calculating sets, tag, offset: (Direct Mapped)

- Direct Mapped, 32 bit address, 64KB cache, 32 byte block
  - How many sets? 64KB / 32 = 2048 = 2^{11}
  - How many bits for the tag? 32 - 11 - 5 = 16
  - How many bits for the offset? $\log_2(32) = 5$



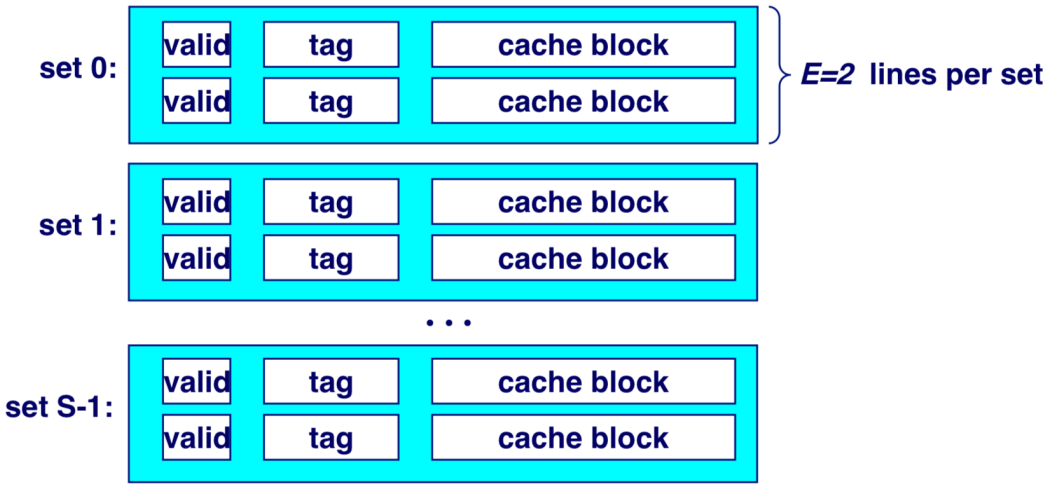
Calculating sets, tag, offset: (2-way Associative)

- 2-way Associative, 32 bit address, 64KB cache, 32 byte block
  - How many lines?
  - How many sets?
  - How many bits for the tag?
  - How many bits for the offset?



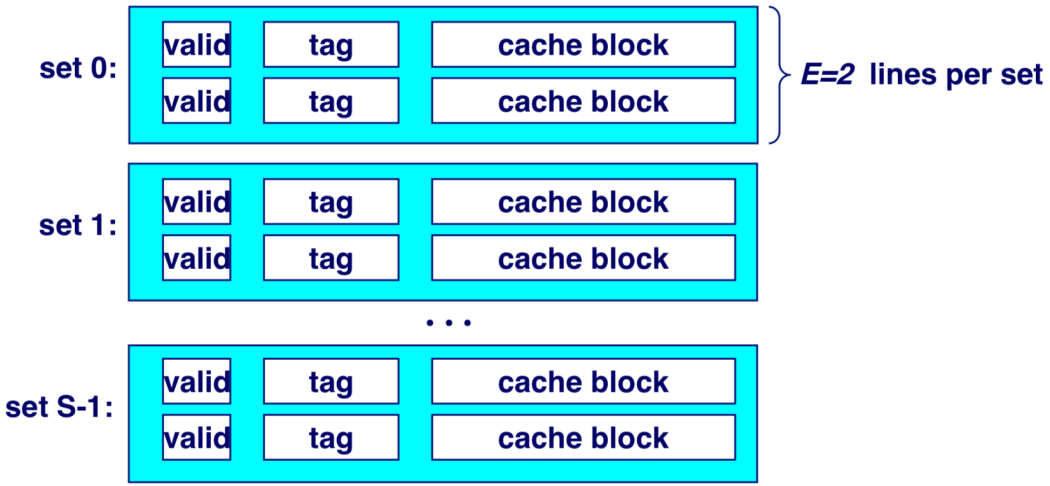
Calculating sets, tag, offset : (2-way Associative)

- 2-way Associative, 32 bit address, 64KB cache, 32 byte block
  - How many lines?
  - How many sets?
  - How many bits for the tag?
  - How many bits for the offset? $\log_2(32) = 5$



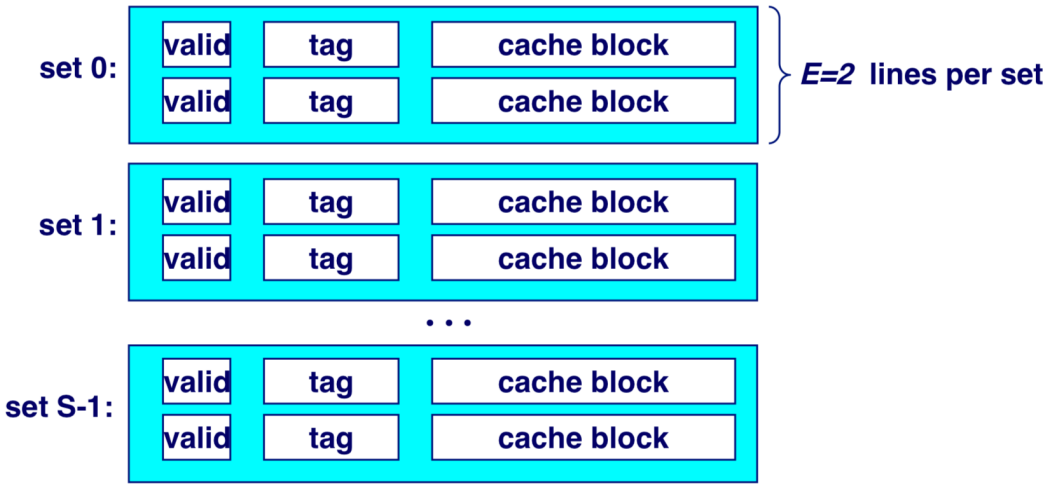
Calculating sets, tag, offset : (2-way Associative)

- 2-way Associative, 32 bit address, 64KB cache, 32 byte block
  - How many lines? 64KB / 32 = 2048 = 2^{11}
  - How many sets?
  - How many bits for the tag?
  - How many bits for the offset? $\log_2(32) = 5$



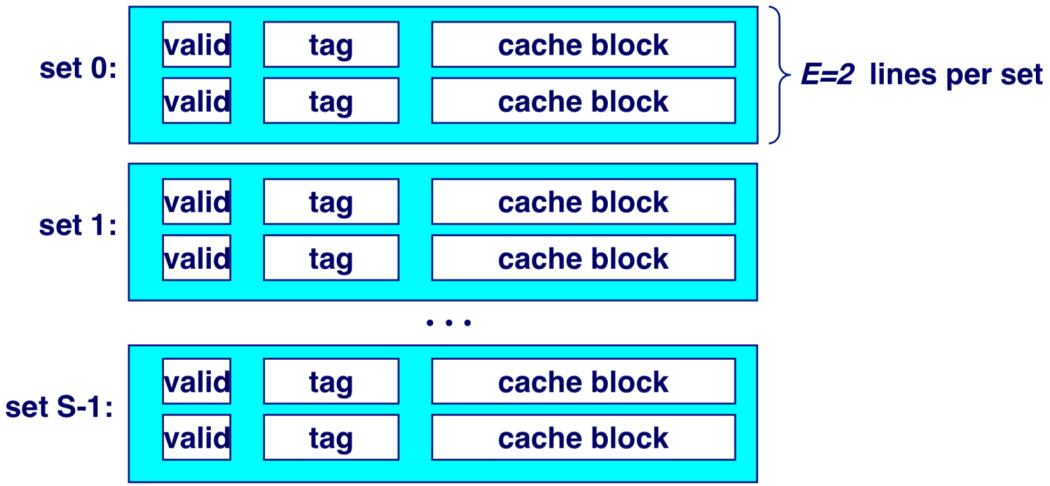
Calculating sets, tag, offset : (2-way Associative)

- 2-way Associative, 32 bit address, 64KB cache, 32 byte block
  - How many lines? 64KB / 32 = 2048 = 2^{11}
  - How many sets? 2048 / 2 = 1024 = 2^{10}
  - How many bits for the tag?
  - How many bits for the offset? $\log_2(32) = 5$



Calculating sets, tag, offset : (2-way Associative)

- 2-way Associative, 32 bit address, 64KB cache, 32 byte block
  - How many lines? 64KB / 32 = 2048 = 2^{11}
  - How many sets? 2048 / 2 = 1024 = 2^{10}
  - How many bits for the tag? 32 - 10 - 5 = 17
  - How many bits for the offset? $\log_2(32) = 5$



Question

- 4-way Associative, 32 bit address, 64KB cache, 32 byte block

	# Lines	# Sets	# Bits for Tag	# Bits for Offset
A:	2048	512	18	5
B:	2048	512	17	6
C:	2048	256	18	6
D:	1024	256	19	5
E:	2048	1024	17	5

Question

- 4-way Associative, 32 bit address, 64KB cache, 32 byte block

	# Lines	# Sets	# Bits for Tag	# Bits for Offset
A:	2048	512	18	5
B:	2048	512	17	6
C:	2048	256	18	6
D:	1024	256	19	5
E:	2048	1024	17	5

Writes and Cache

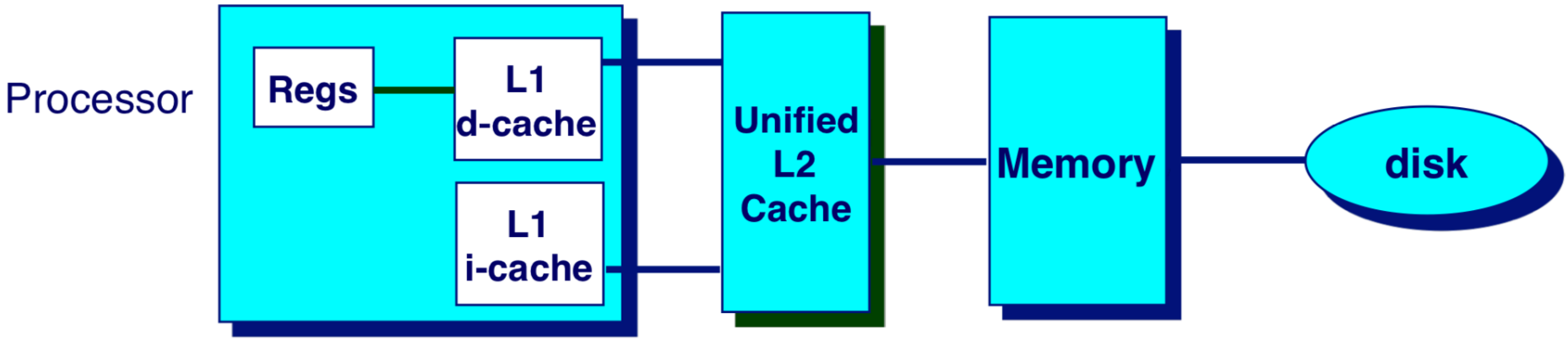
- Reading information from a cache is straight forward
- What about writing?
  - What if you're writing data that is already cached (write-hit)?
  - What if the data is not in the cache (write-miss)?

Writes and Cache

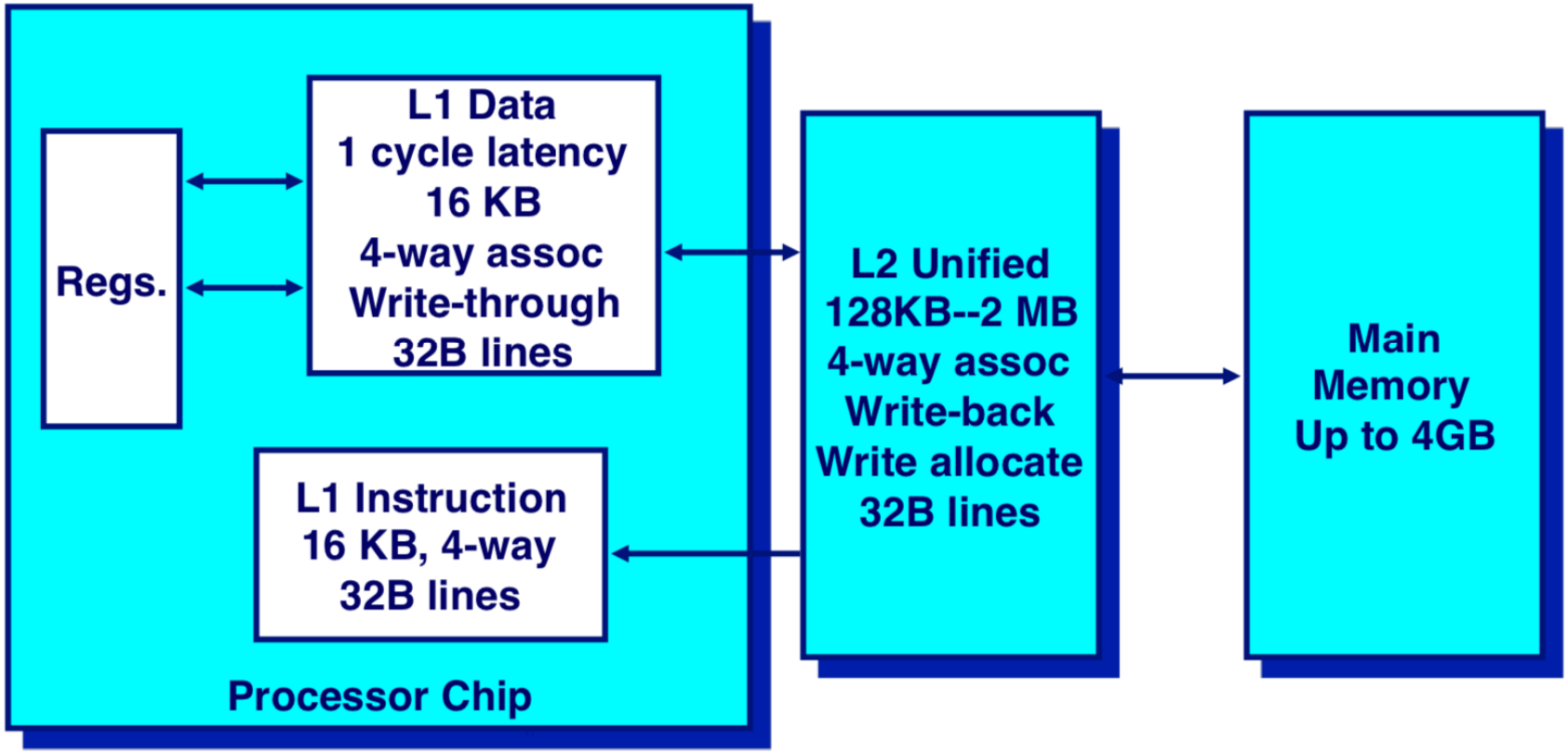
- Dealing with a write-hit
  - Write-through: immediately write data to memory
  - Write-back: defer the write to memory as long as possible
- Dealing with a write-miss
  - write-allocate: load the block into memory and update cache
  - no-write-allocate: writes directly to memory

Multi-Level Caches

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



Intel Pentium Cache Hierarchy



Cache Performance Metric

- Miss Rate
  - Fraction of memory references not found in cache
  - Typical numbers:
    - 3-10% for L1
    - < 1% for L2, depending on the size.

Cache Performance Metric

- Hit Time
  - Time to deliver a line in the cache to the processor
  - Typical numbers:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2

Cache Performance Metric

- Miss Penalty
  - Additional time required because of a miss
    - Typically 50-100 cycles from main memory

Cache Miss Types

- Cold misses
  - When a location is accessed for the 1st time
  - Can reduce by increasing block size

Cache Miss Types

- Conflict misses
  - More blocks that map into a single set is concurrently used than can be stored in the set
  - Can reduce by increasing associativity

Cache Miss Types

- Capacity miss
  - More blocks are active than can fit into the cache
  - Bigger cache?

Cache Miss Types

- 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 having a miss
  - If wrong, can worsen performance of cache

Writing Cache Friendly Code

- Repeated references to variables are good
- Stride-1 reference patterns are good (spatial locality)
- Example: cold cache, 4-byte words, 4-word cache block

```
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%

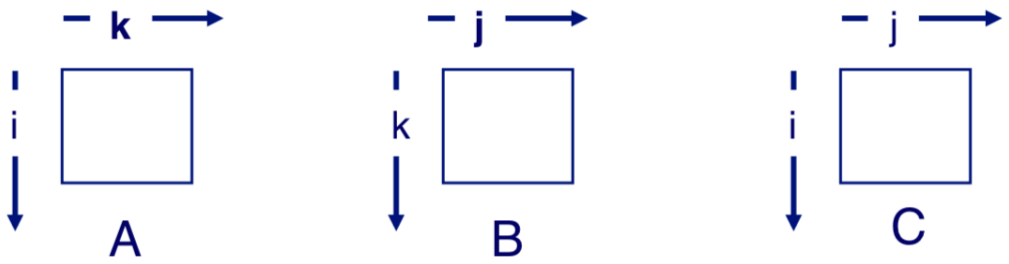
Matrix Mult. Example

- Cache effects to consider:
  - Total cache size
  - Block size
- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - N reads per source element
  - N additions per destination

```
/* ijk */                                  Variable sum
for (i=0; i<n; i++)  {                     held in register
  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;
  }
}
```

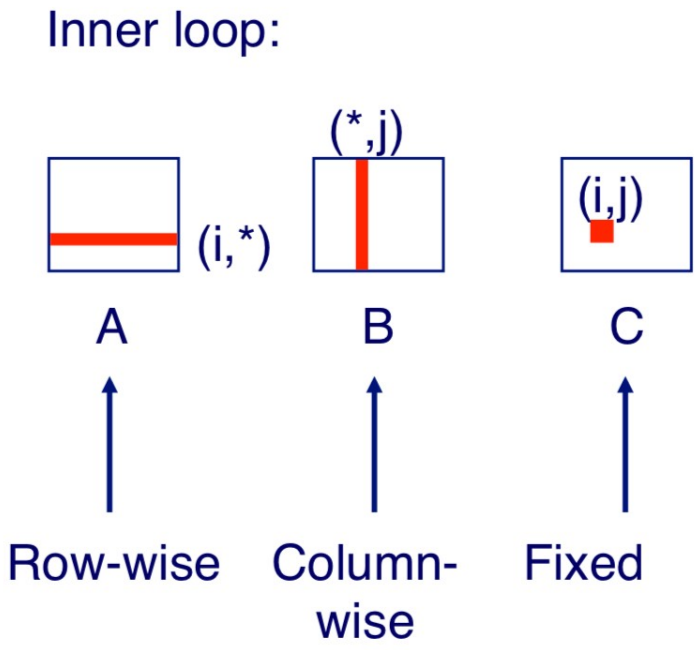
Miss Rate Analysis

- Assume:
  - Line size = 16B (big enough for 4 integers)
  - Matrix dimension (N) is very large
  - Cache is not big enough to hold a single row
  - Row major order matrix
- Analysis Method:
  - Look at access pattern of inner loop



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;
  }
}
```

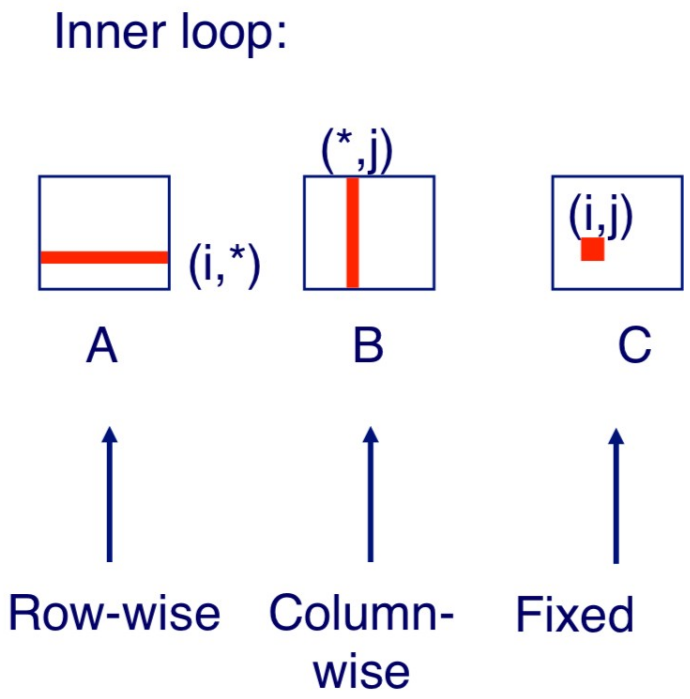


Misses per Inner Loop Iteration:

A	B	C
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
  }
}
```

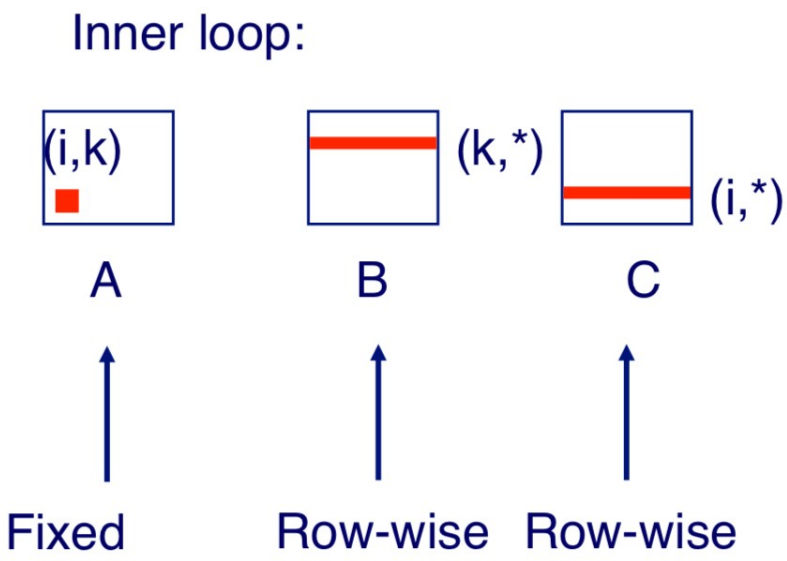


Misses per Inner Loop Iteration:

A	B	C
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];
  }
}
```

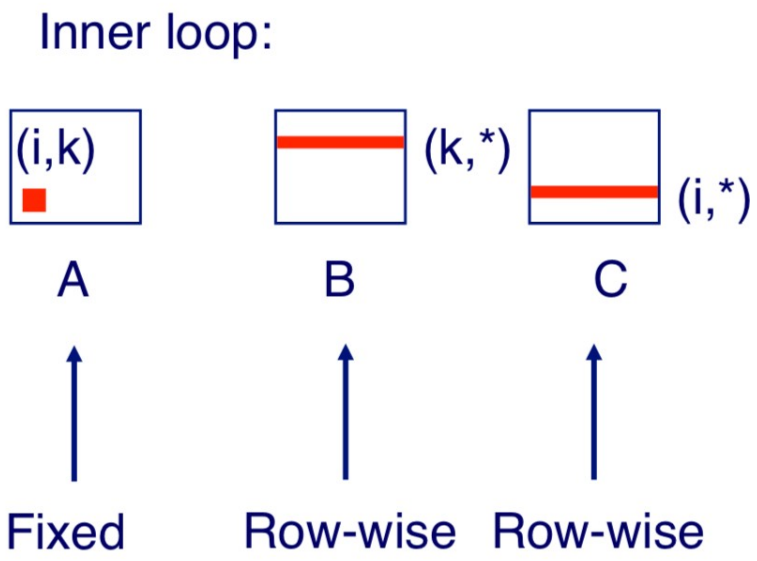


Misses per Inner Loop Iteration:

A	B	C
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];
  }
}
```

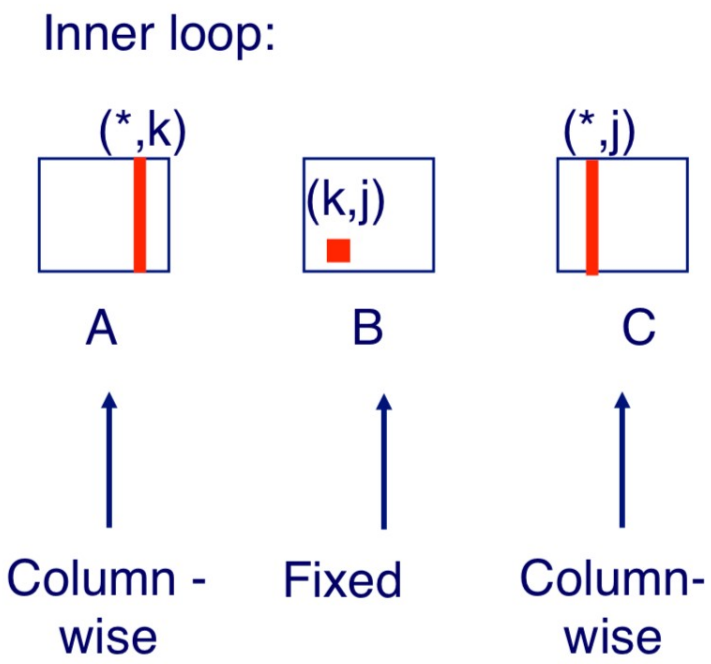


Misses per Inner Loop Iteration:

A	B	C
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;
  }
}
```

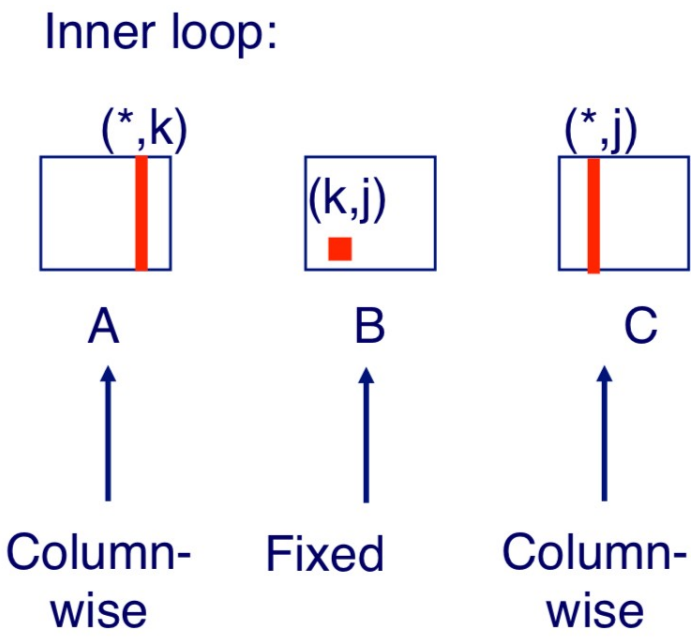


Misses per Inner Loop Iteration:

A	B	C
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:

A	B	C
1.0	0.0	1.0

Summary of Matrix Mult

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;
    }
}
```