

## Memory and Caches

See also cache study guide.

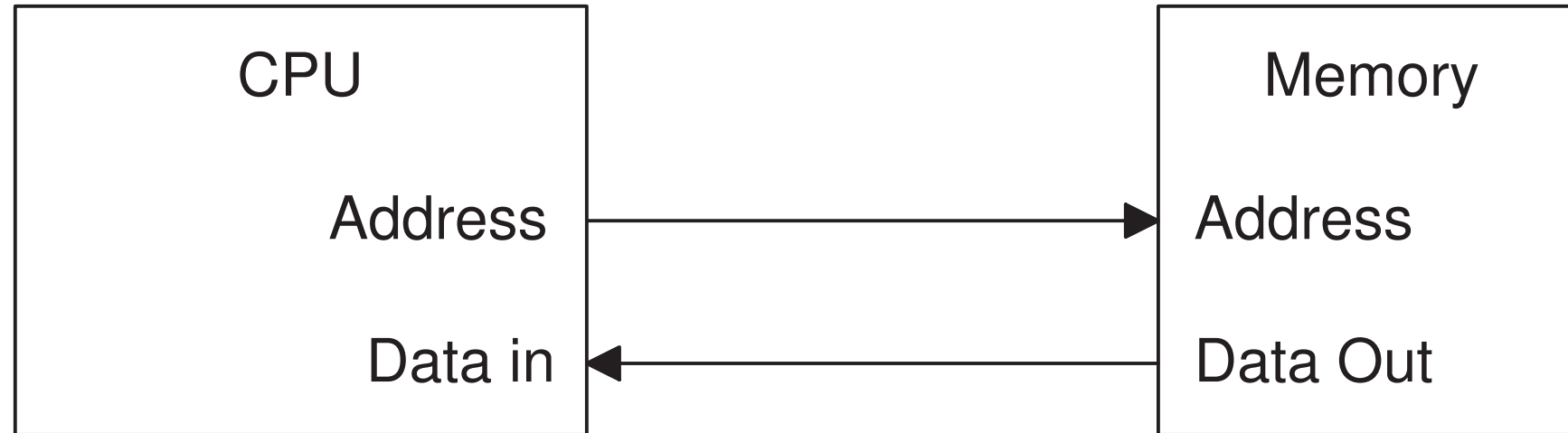
## Contents

Supplement to material in section 5.2.

Includes notation presented in class.

## Memory Basics and Terminology

In essence a memory system is quite simple:



Retrieve or store data at specified effective address.

But in practice its complexity can rival that of the processor.

## Address Spaces

### *Address Space:*

The set of addresses defined in the ISA.

### *Address Space Size:*

The (maximum) number of bits needed to represent an address.

Symbol used in class:  $a$ .

*The size of the address space defined by the MIPS32 ISA is 32 bits and so a memory address is 32 bits and there are  $2^{32}$  memory locations.*

Typical address space sizes:

DLX, MIPS32, Sun SPARC V8, IA-32, ARM T16, ARM A32, RISC-V RV32, and other older ISAs,  $a = 32$ ,

Intel 64, Itanium, MIPS64, DEC Alpha, Sun SPARC V9, ARM A64, RISC-V RV64,  $a = 64$ .

## Real v. Virtual Address Spaces

*Real Address Space:* a.k.a. *Physical Address Space*

The set of addresses used by the memory hardware (typically DRAM chips).

*Real Address:*

An address in a real address space.

Given a real address, can open up computer's case, find the right DRAM chip, saw off its top, ...

... and with a very steady hand and a very fine needle...

... point to the data at that address.

*Virtual Address Space:*

The set of addresses usually used by programs.

The  $2+r3$  address computed by `lw r1, 2(r3)` is typically a virtual address.

Physical location used for a virtual address can change as a program runs.

Given a virtual address, cannot point to a chip and say data is there...

... since location of data is not pre-determined and location can change.

## Usage of Virtual Address

### Virtual Address “Users”

Instructions in most programs ...  
... including our code samples from class.

Usually the L1 cache, sometimes the L2 cache (covered later).

### Real Address “Users”

The first instructions that a computer runs after it resets.

Low-level OS code written by those getting A+'s in this class.

Memory devices use physical addresses.

## Translation between Virtual and Real Addresses

Translation from Virtual to Real Addresses ...

... performed in part by hardware somewhere between the CPU and memory.

The hardware called a *translation lookaside buffer (TLB)* ...

... which is itself a kind of a cache.

The TLB is typically updated by OS software ...

... in response to a *TLB Miss* exception.

Depending on design, cache uses virtual or physical addresses.

Processor can switch between virtual and real addresses.

## Data Bus and Alignment

### *Character:*

What a memory address refers to; defined by the ISA.

Number of bits in a character denoted  $c$

In most systems,  $c = 8$  bits and called a byte.

In MIPS `lb r1, 0(r2) ...`

... loads the one character at address `0(r2)` into register `r1`.

*Bus Width:*

The number of bits brought into the processor in a single memory access.

Symbol used in class  $w$ .

The number of bits accessed by an instruction may be less, the other bits are ignored.

This is an implementation feature.

In any reasonable system  $w$  is a multiple of  $c$ .

Typical values,  $w = 64$  bits.



## Bus/Alignment Example

---

```
# Data in memory:
# 0x1000: 5, 0x1001: 6, 0x1002: 7, 0x1003: 8.
# 0x1004 to 0x1007: 0x11111111
addi r1, r0, 0x1000
lb r10, 0(r1) # Eff. addr = 0x1000, r10 <- 0x00000005
lb r11, 1(r1) # Eff. addr = 0x1001, r11 <- 0x00000006
lb r12, 2(r1) # Eff. addr = 0x1002, r12 <- 0x00000007
lb r13, 3(r1) # Eff. addr = 0x1003, r13 <- 0x00000008
lh r15, 0(r1) # Eff. addr = 0x1000, r15 <- 0x00000506
lh r16, 2(r1) # Eff. addr = 0x1002, r16 <- 0x00000708
lw r20, 0(r1) # Eff. addr = 0x1000, r20 <- 0x05060708
```

---

## Relationship Between Address and Data on Bus

If  $w = c$  then data on bus is for a single address.

If  $w/c = 2$  then data on bus is for two consecutive addresses.

If  $w/c = D$  then data on bus is for  $D$  consecutive addresses.

When load instructions refer to less than  $w/c$  addresses ...  
... the processor hardware extracts bits needed ...  
... using an *alignment network*.

## Alignment Network

### Alignment Network:

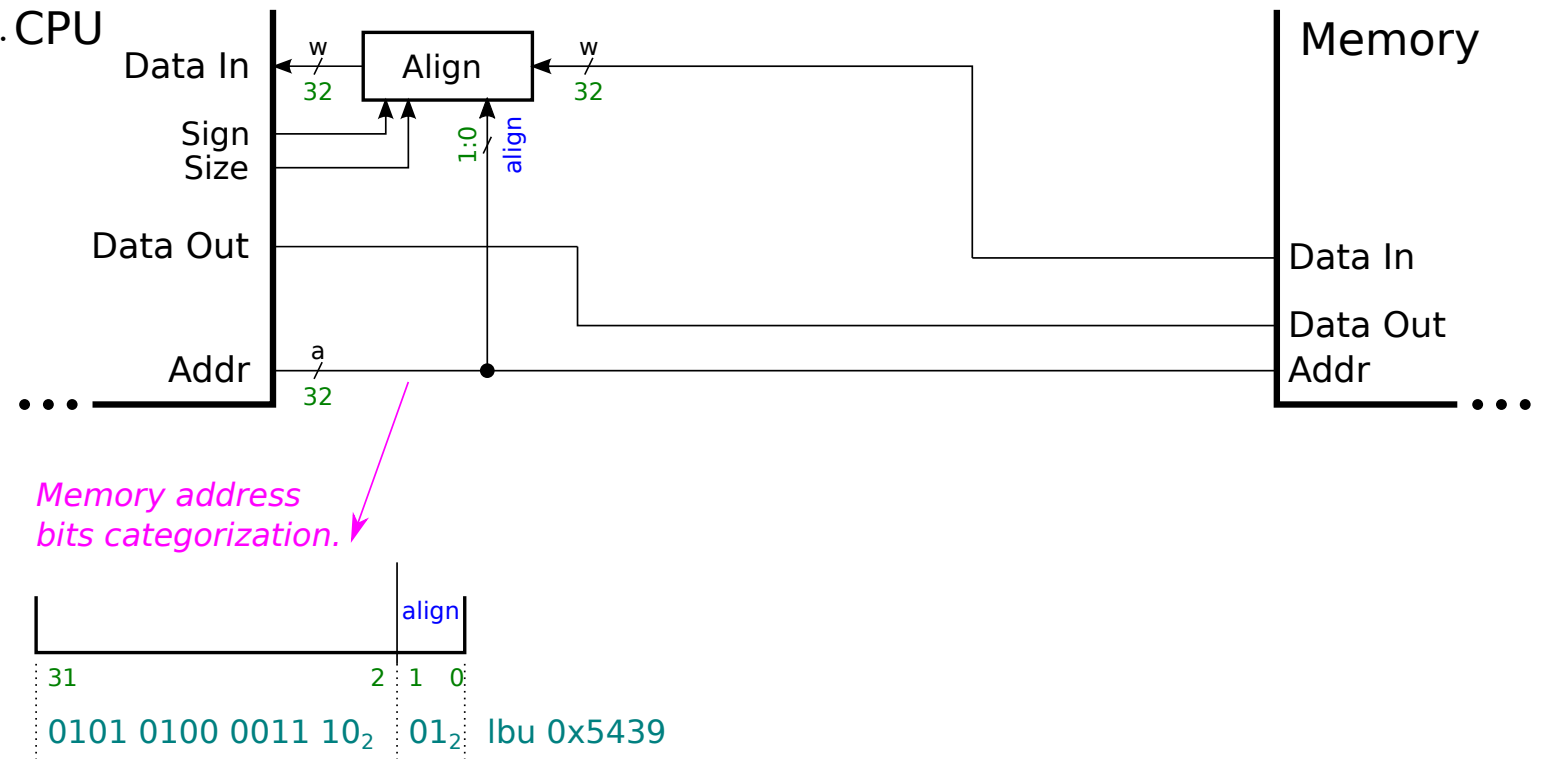
The hardware that shifts and pads or sign-extends data on the bus so that it can be used by the processor.

Bus width fixed at  $w$  bits ...

... but instructions may expect fewer bits.

Alignment network extracts bits that instruction needs ...

... and pads unused high-order bits with zeros or sign.



Example: Suppose  $w = 128 \text{ b} = 16 \text{ B}$ :

---

```
# r1 = 0x1003
# MEM[0x1000..0x100F] = 0x80 0x81 0x82 ... 0x8f
lb r2, 0(r1)    # Bus contents: 0x80 0x81...0x8f, r2<-0xffffffff83
lbu r3, 0(r1)   # Bus contents: 0x80 0x81...0x8f, r3<-0x83
lw  r4, 1(r1)   # Bus contents: 0x80 0x81...0x8f, r4<-0x84858687
```

---

For `lb` alignment network extracts bits 24 : 31 from bus ...  
 ... padding high-order bits with 1 (MSB of 0x83).

For `lbu` alignment network extracts bits 24 : 31 from bus ...  
 ... padding high-order bits with 0 (unsigned load).

For `lw` alignment network extracts bits 32 : 63 from bus ...  
 ... no padding since register and data are both 32 bits.

## Cache Motivation

Loads and stores are performed frequently.

Fast memory devices (SRAM) are expensive and size-limited and ...

... cheap memory devices (DRAM) are slow.

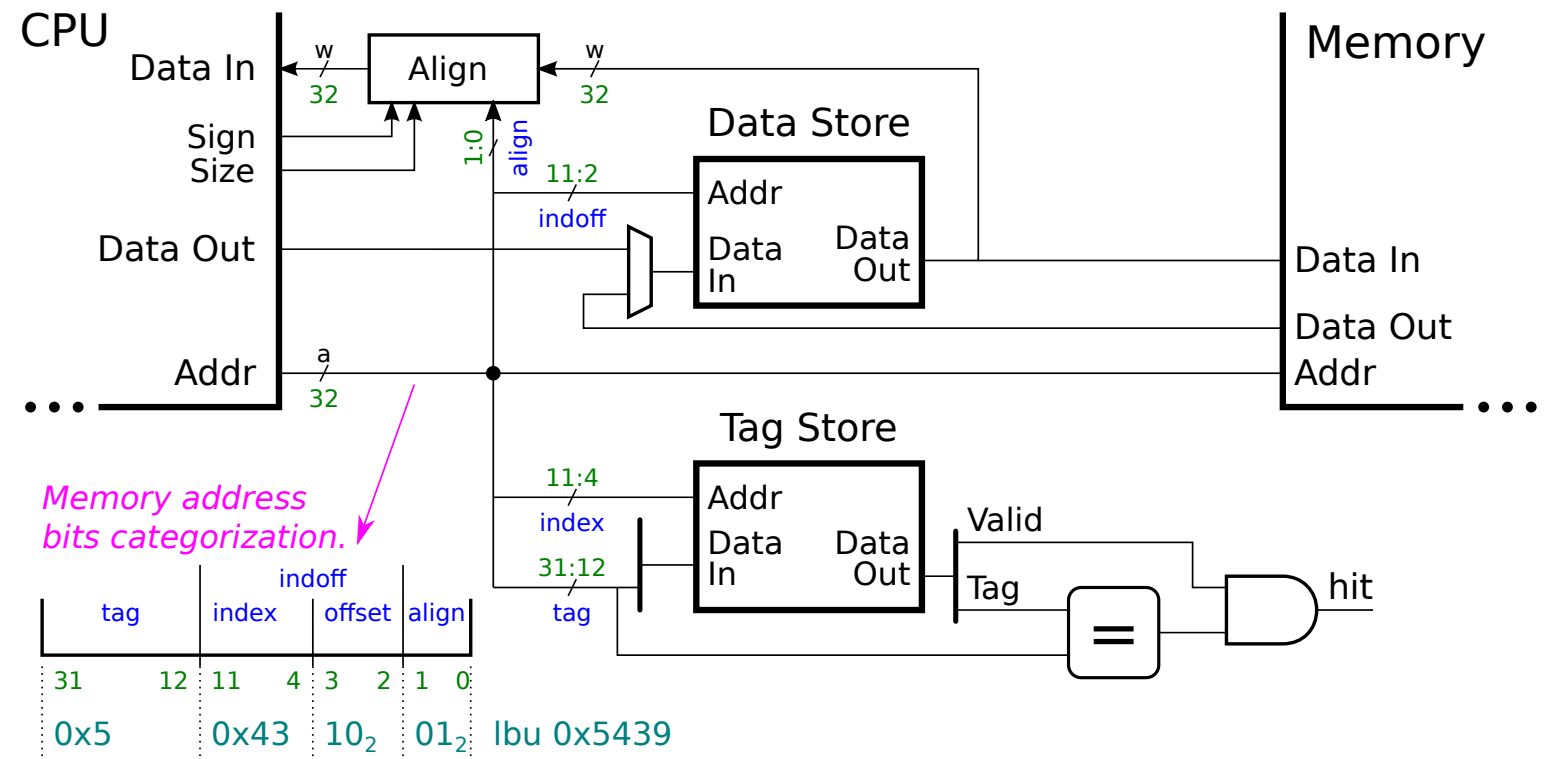
By organizing these devices into a cache performance will be ...

... almost as fast as expensive devices ...

... almost as cheap as slow devices.

Note: It's pronounced cash ...

... and occasionally abbreviated \$.



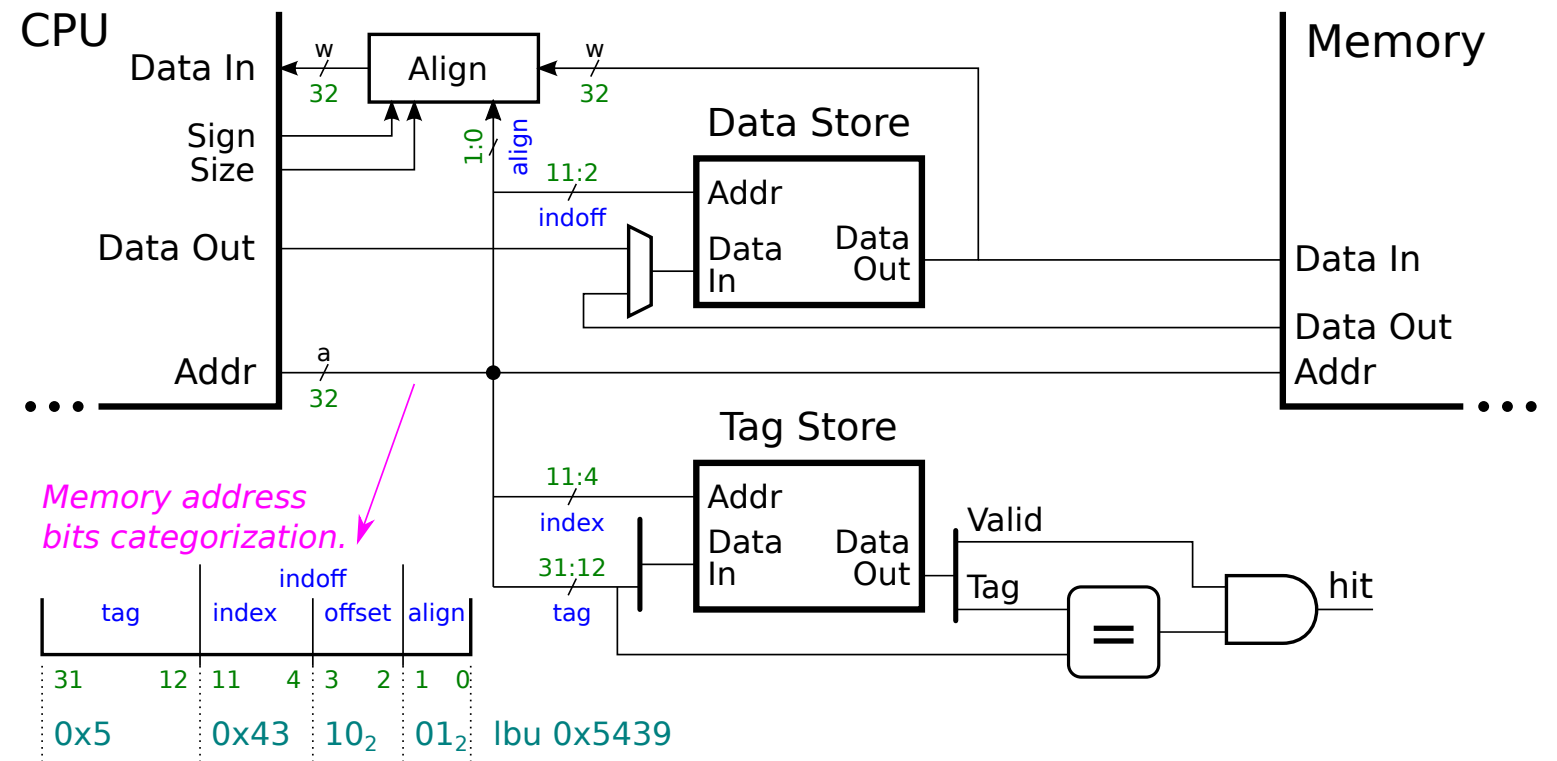
## Cache Key Idea

Use cheap devices to implement (real part of) address space.

Use expensive devices to duplicate parts of address space.

Definition above can apply to many types of caches, *e.g.*, disk caches.

This set concerned with main-memory caches.



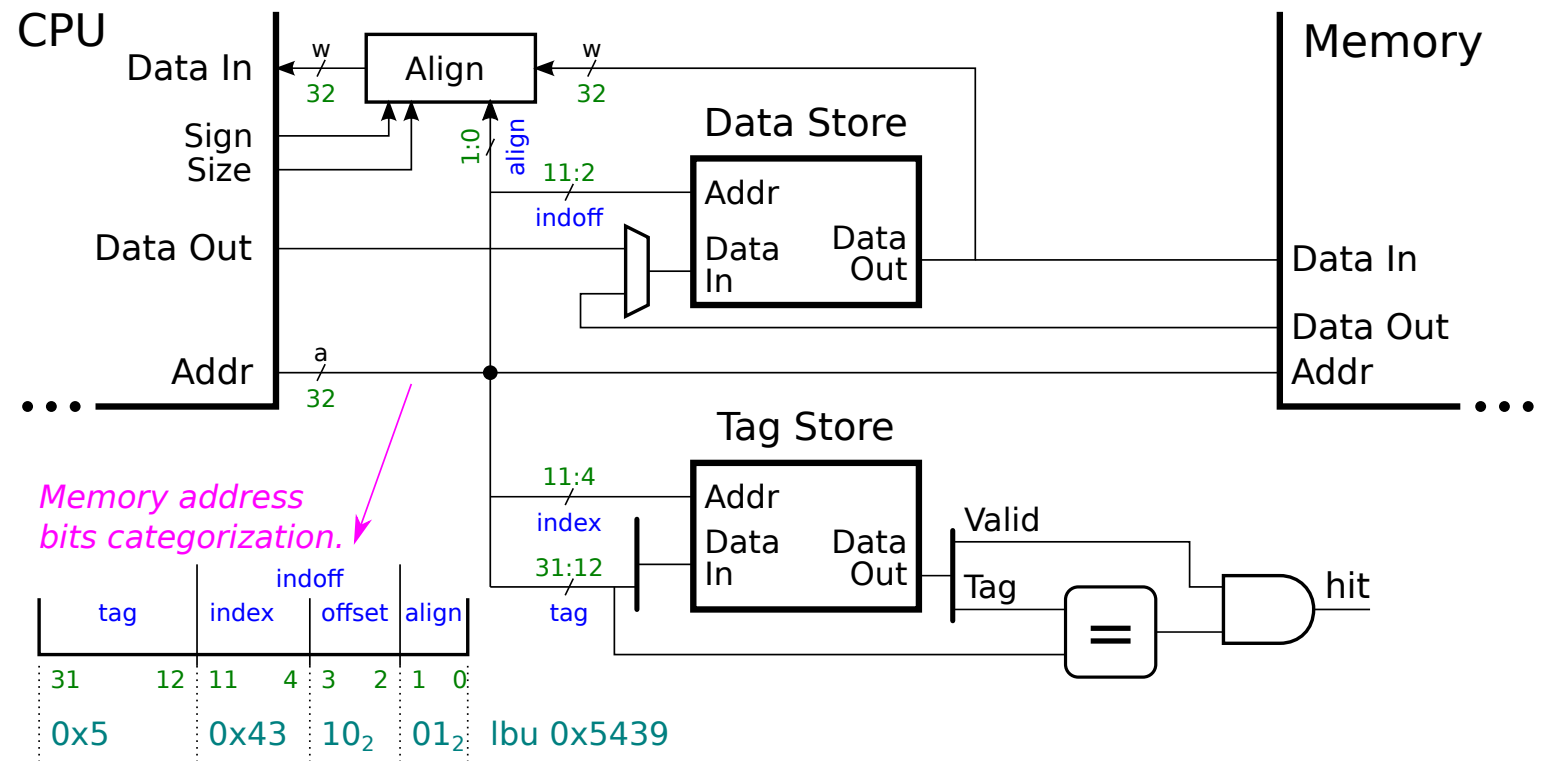
## Cache Use

If accessed data found in expensive devices (labeled Data and Tag Store in diagram), ...  
 ... data will be returned quickly.

Called a *hit*.

If accessed data not in expensive devices ...  
 ... data copied from cheap (labeled Memory in diagram) ...  
 ... to expensive devices ...  
 ... and passed to processor...  
 ... taking much more time than a hit.

Called a *miss*.



## Direct-Mapped Cache

### Definitions

#### *Cache:*

High speed storage that holds data which is normally kept in larger, slower storage. The larger storage is ordinarily accessed only after not finding data in the cache.

#### *Direct-Mapped Cache:*

A simple cache design in which the low bits of an address determine where the data will be stored. Two addresses that are identical in their low bits but differ in the other bits cannot reside in a direct mapped cache at the same time.

#### *Hit:*

A successful attempt to find data in a cache. The data will be available quickly.

#### *Miss:*

An unsuccessful attempt to find data in a cache. Typically, the data will be retrieved from memory (or the next cache layer) placed into the cache, and sent to the CPU (or the previous cache layer).



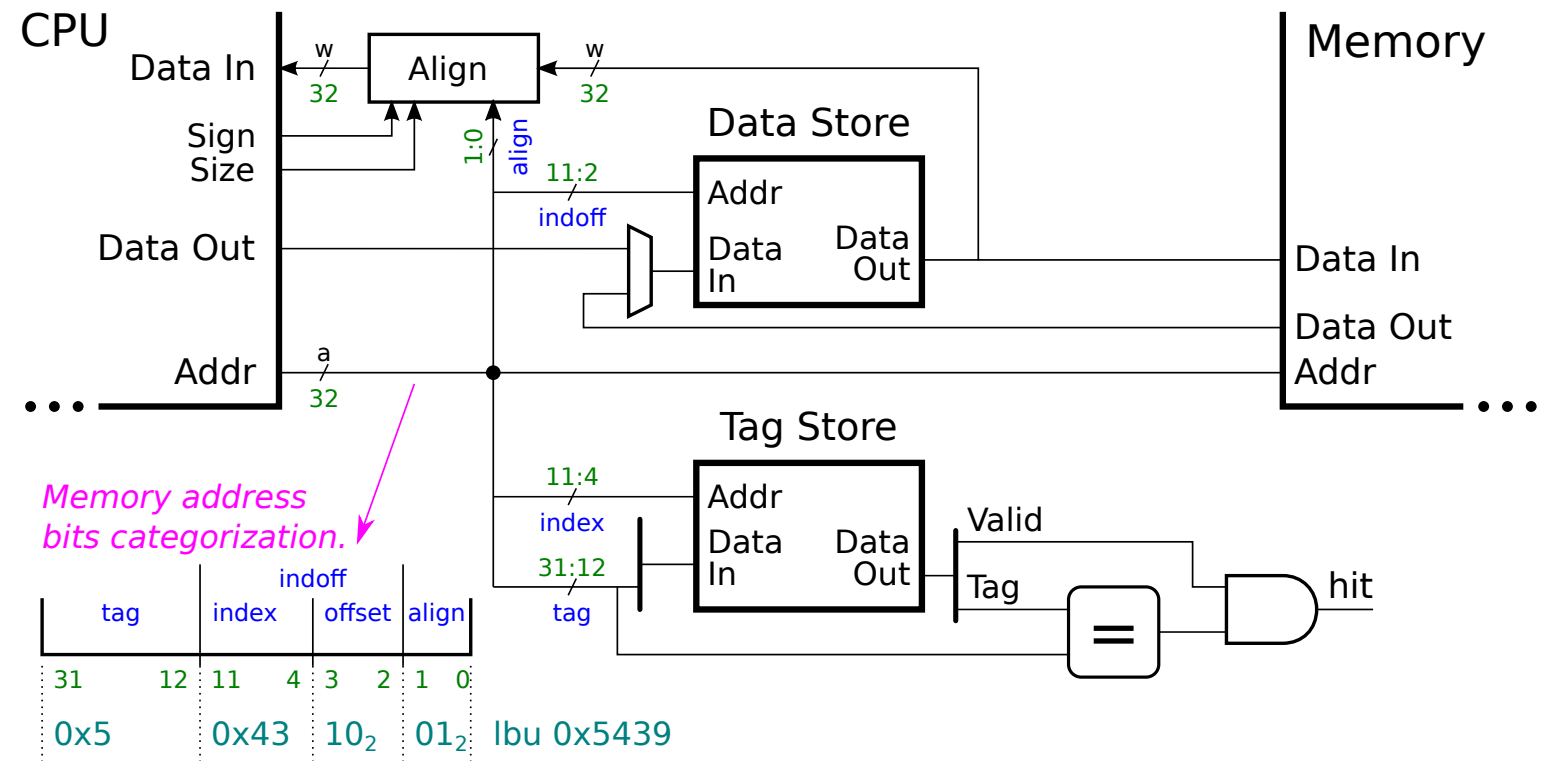
## Direct Map Cache Organization

Uses two memories, called *data store* and *tag store*. (Both expensive.)

### *Data Store:*

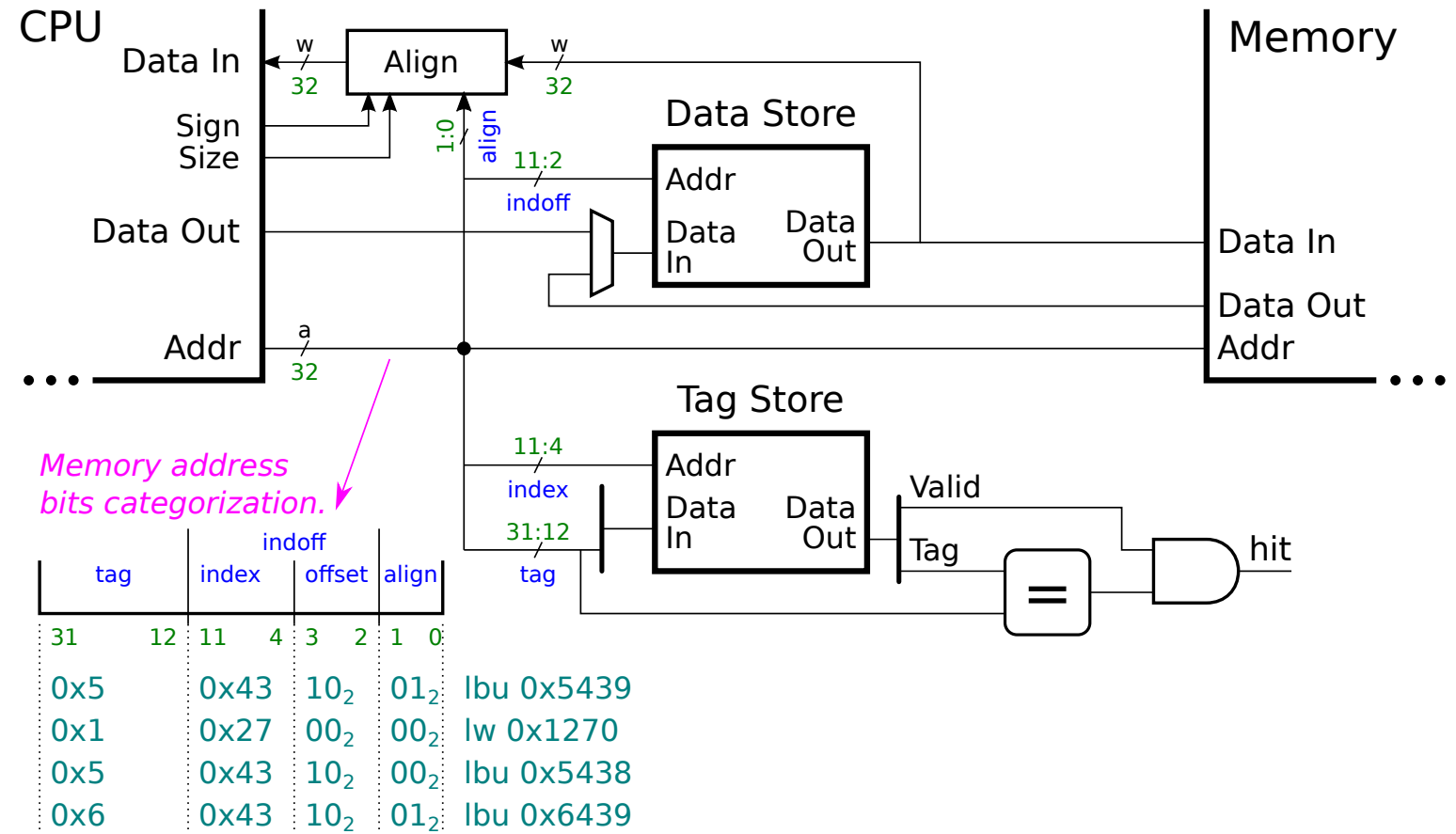
The cache memory used to hold cached data.

Data memory holds copies of data held by cheap devices.



**Tag Store:**

Cache memory that holds information about cached data, including part of the cached data's addresses, called a *tag*, and status bits.



## Cache Lines

**Line:** also called a block

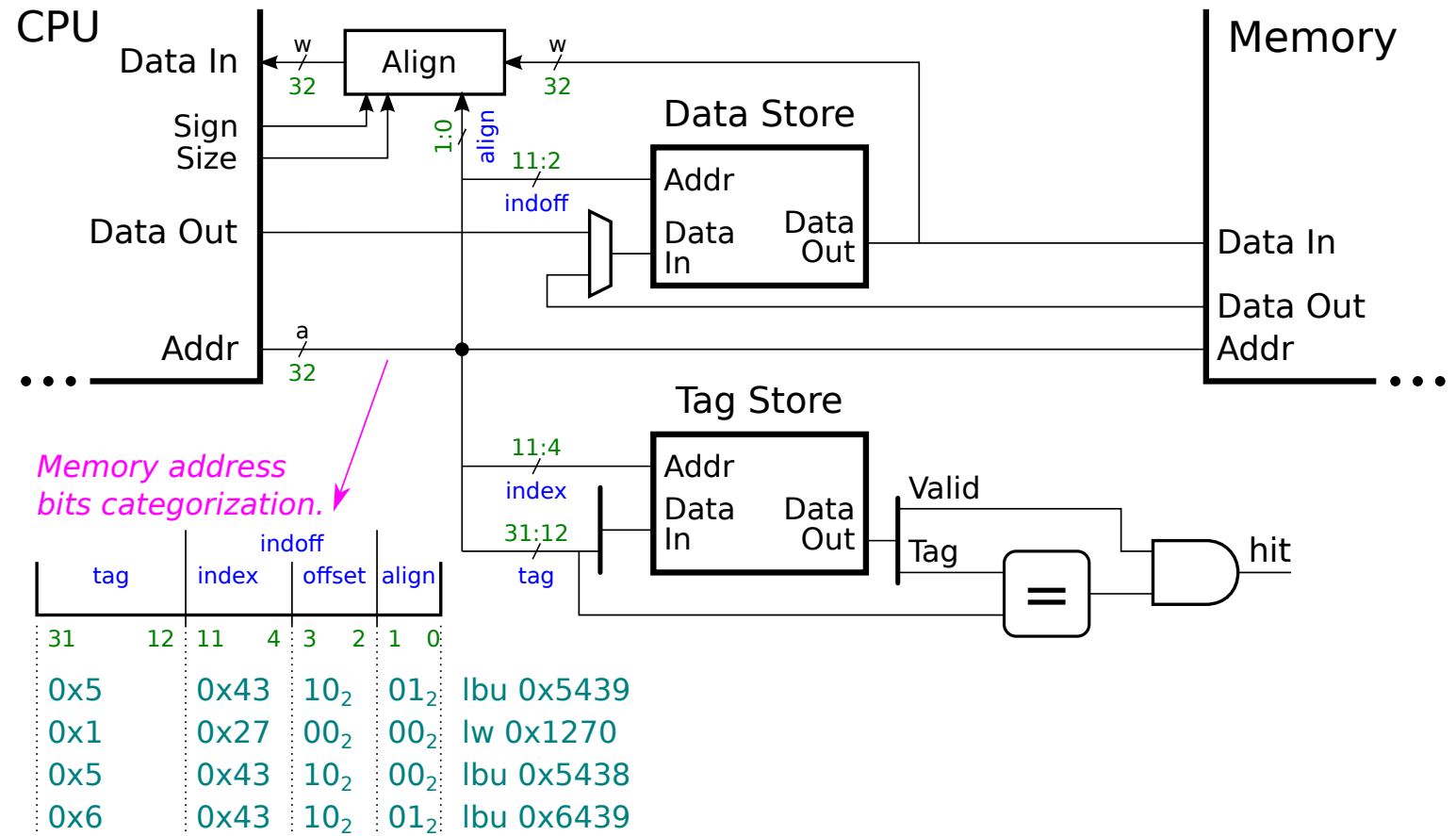
The unit of storage in a cache.

Consists of one or more bus-width's worth of data.

On a miss an entire line's worth of data copied from main memory.

The size of a line usually given in terms of characters.

Symbols used in class:  $L$  line size in characters,  $l = \log_2 L$ .



## Cache Sets

### Set:

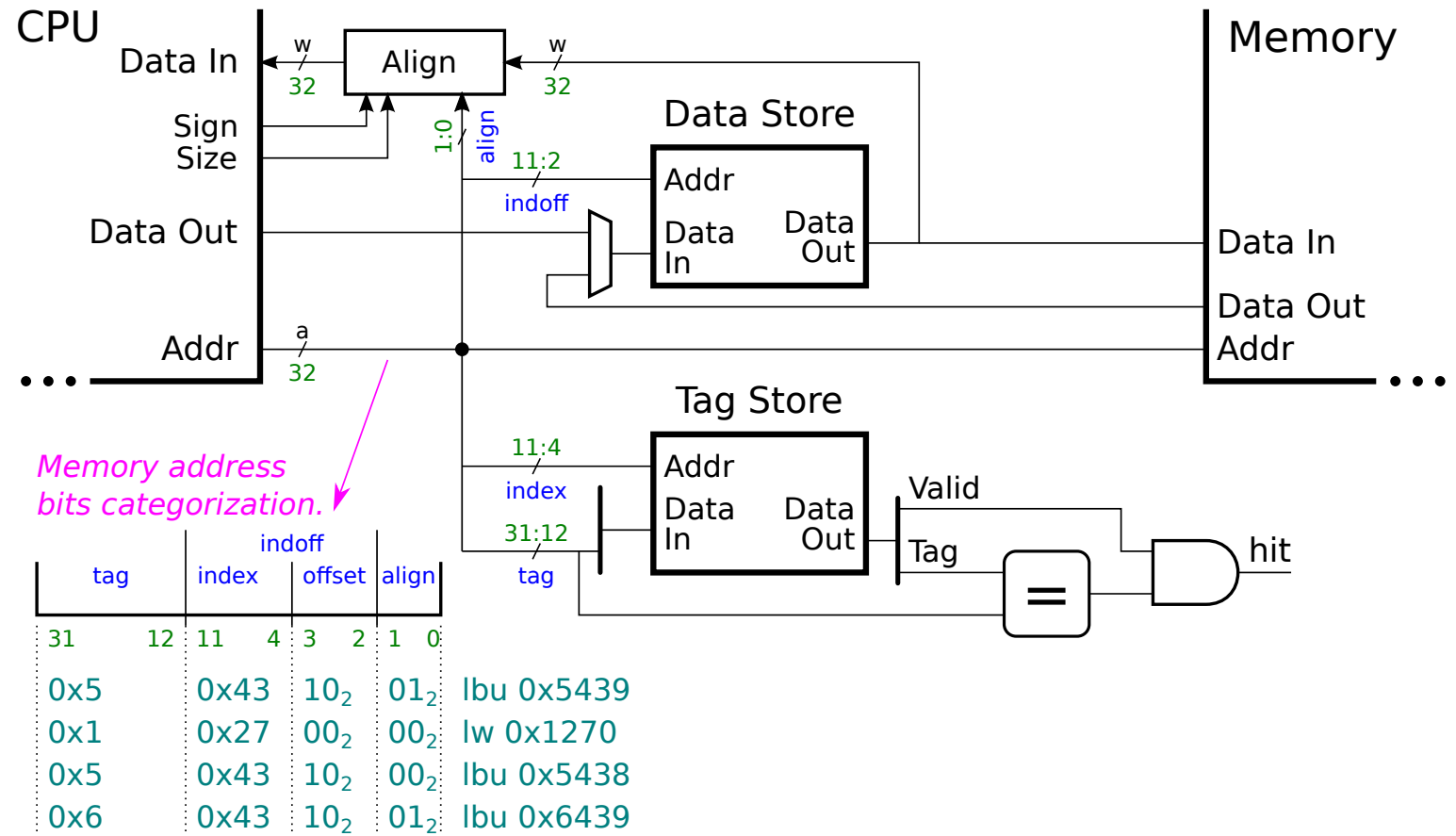
Lines that share a common *index*.

Index explained further below.

One measure of a cache is the number of sets it can hold.

The cache capacity is the product of number of sets, lines per set, and line size.

Symbol:  $s$ , denotes  $\log_2$  of the number of sets.



## Address and Cache Structure

Address bit positions, *fields*, can be used to locate data in cache.

For

$a$ -bit address space,

$2^s$ -set cache,

$2^l$ -character lines,

and  $d = \log_2(w/c)$  character bus width

the fields would be as follows:

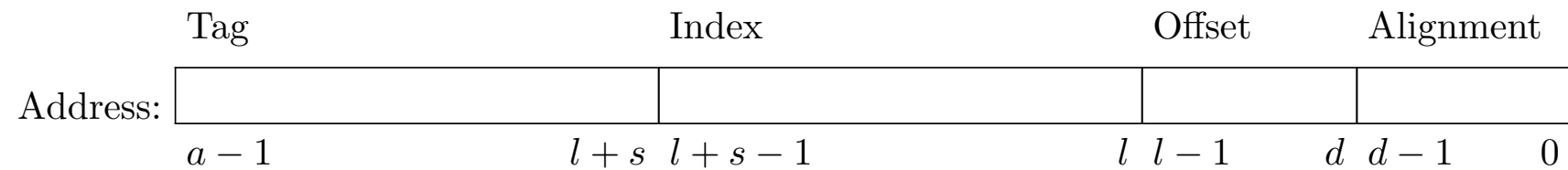
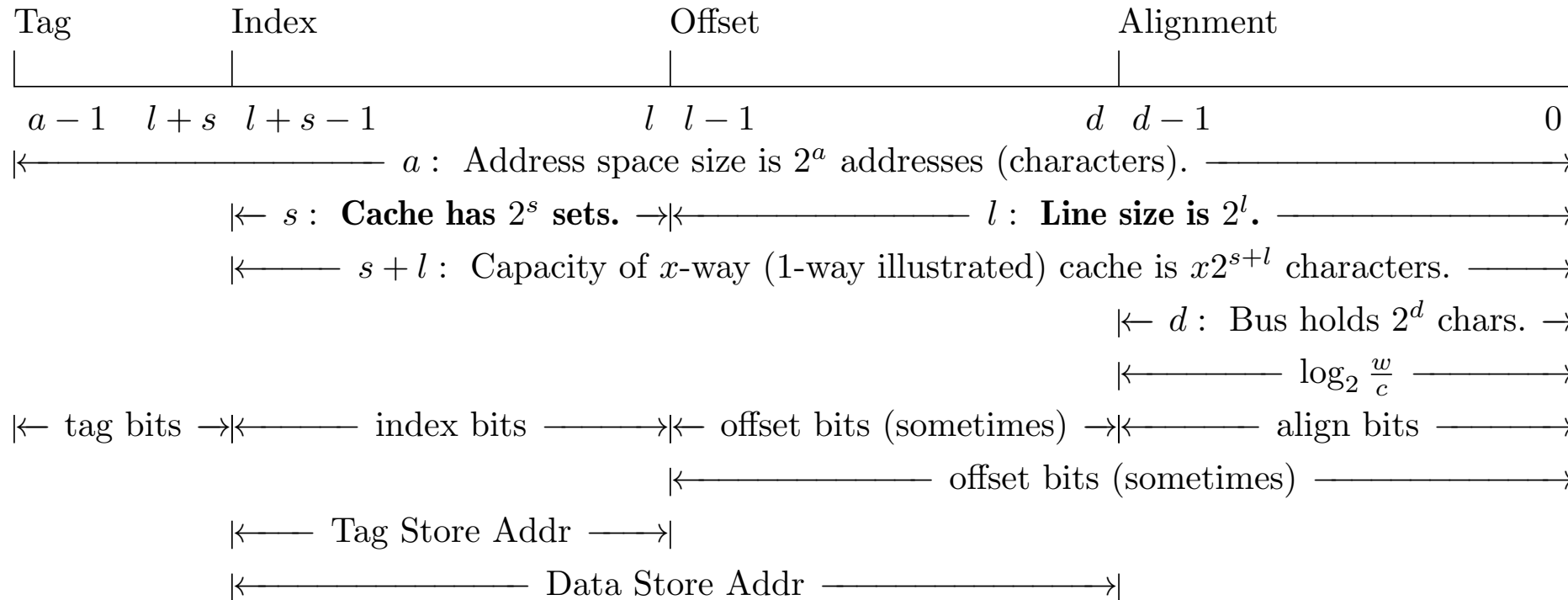
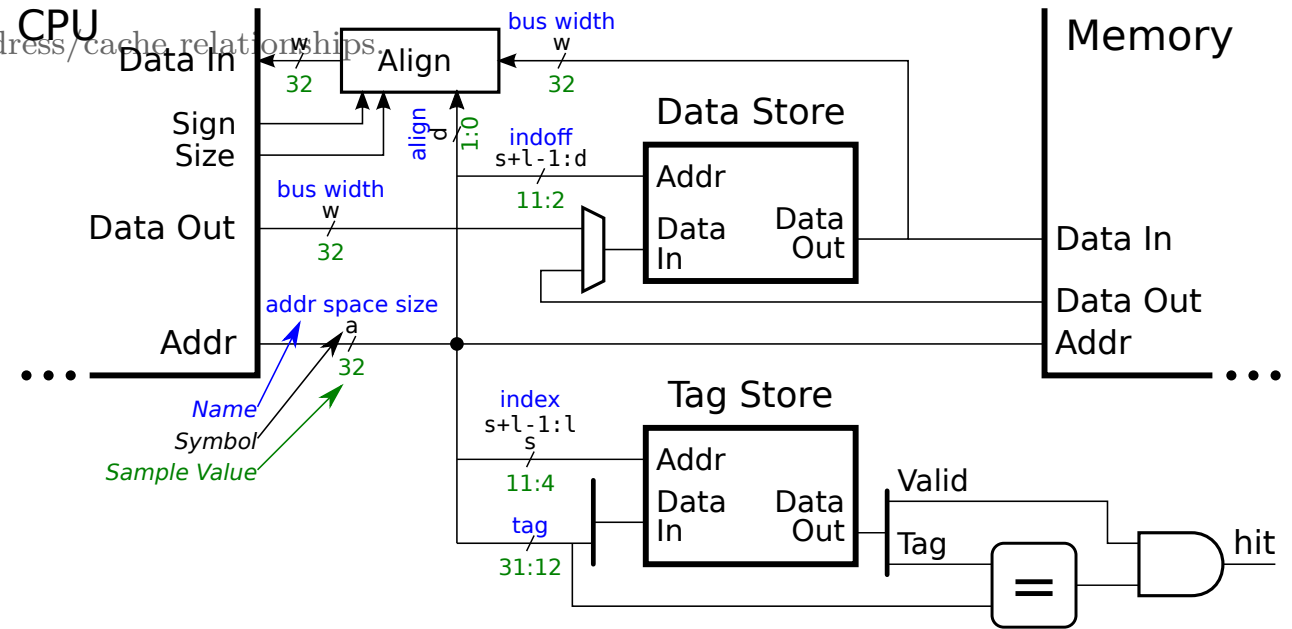


Diagram showing address/cache relationships.

Details on later slides.



*Alignment Bits:*

The low-order address bits that refer to a character position on the bus.

Bits:  $0 : (d - 1) \dots$

$\dots$  where  $d = \log_2(w/c)$  (characters per bus width).

Assuming aligned fetches  $\dots$

$\dots$  the  $d$  least significant bits of an address will either be zero or ignored.

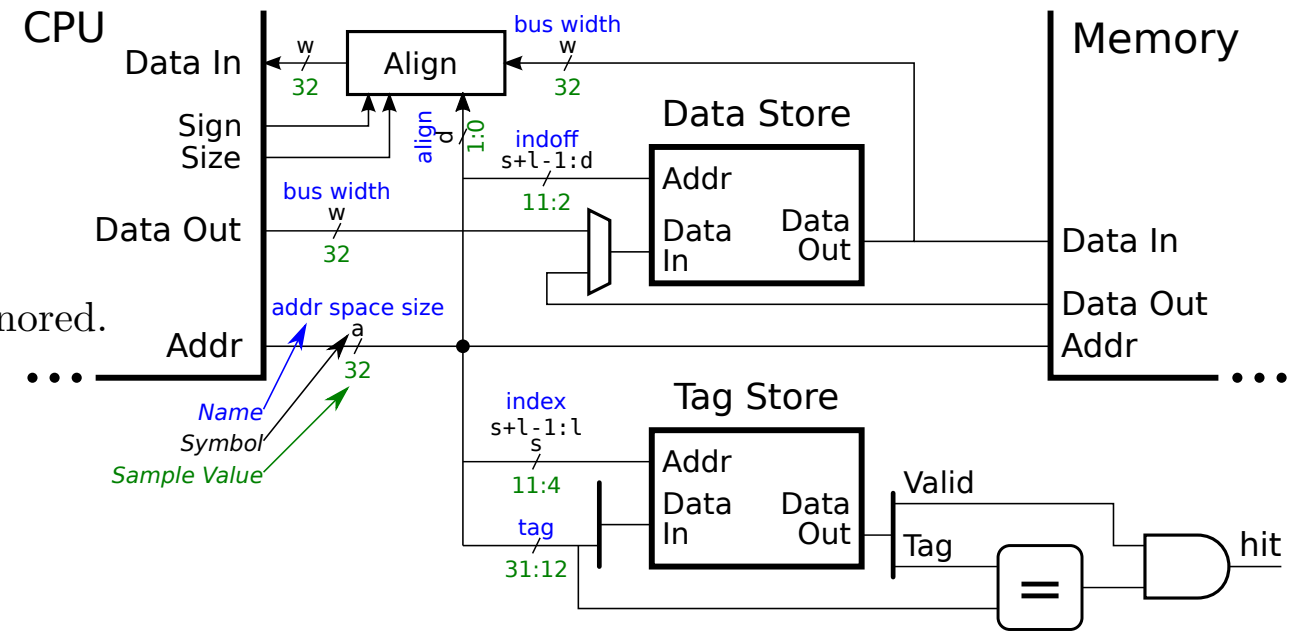
This part of the address is ignored by the cache.

**Example:** Let  $w = 32$  and  $c = 8$ .

Then the bus can hold 4 characters and so  $d = \log_2 \frac{32}{8} = 2$ .

If the 2 least significant address bits are 0 then the address refers to the first character position, bits 0:7 (assuming little-endian numbering).

If the 2 least significant address bits are 01 then the address refers to the second character position, bits 8:15, etc.



*Offset:*

The low-order address bits that refer to a character position within a line.

Bits:  $d : (l - 1)$ ,

(A line can hold more than one bus-width worth of data.)

This and the index are used to look up the data in each of the data memories.

These bits are labeled “offset” in figure 5.8. Two of those five bits are offset bits as defined here and are shown connected (along with index) to the address inputs of the data memories.

*Index:*

The address bits, adjacent to the offset, that specify the cache set to be accessed.

Bits:  $l : (l + s - 1)$

Used to look up a tag in each of the tag memories.

Along with the offset, used to look up the data in each of the data memories.

These are labeled “index” in figure 5.8 and are shown connecting to the data and tag memories.



*Tag:*

The address bits that are neither index nor offset bits.

Bits:  $(l + s) : (a - 1)$

An address bit is either an offset bit, an index bit or a tag bit.

Tag bits are stored in tag memories.

They are later are retrieved and compared to the tag of the address being accessed.

There is a hit if a tag matches and the corresponding valid bit is 1.

Labeled “tag” in figure 5.8. The figure omits the data-in port to the tag memories, which is used for writing a new tag and valid bit on cache replacement.

## Tag Store Information

### Tag Store

Memory with one entry for each line.

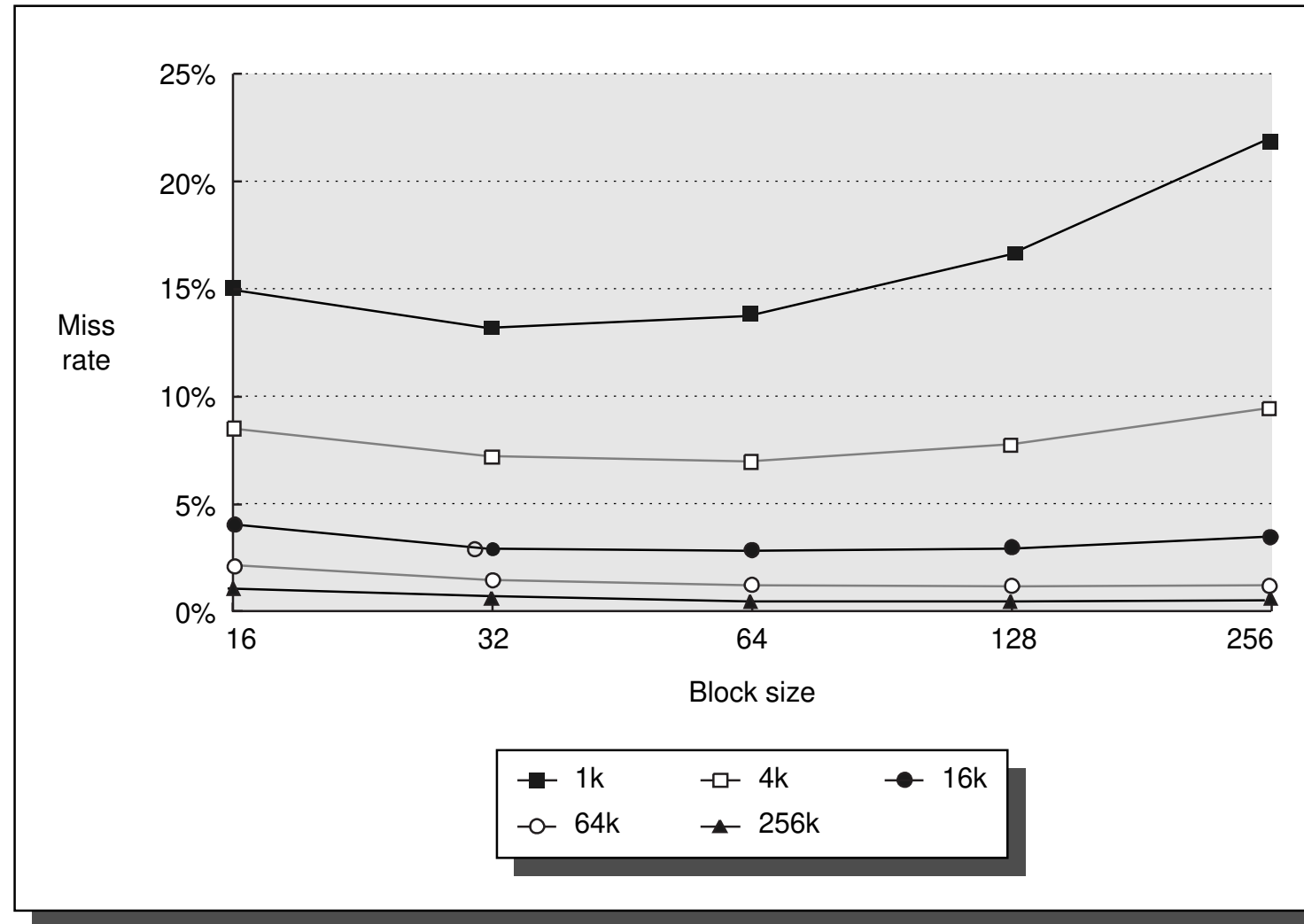
Each entry holds ...

... tag (high-order part of line address) ...

... valid bit (indicating that line is in use) ...

... dirty bit (in write-back caches, indicating higher levels not updated).

### Cache Miss Rate v. Block Size



**FIGURE 5.11 Miss rate versus block size for five different-sized caches.**

## Program Analysis

### Goals:

Determine hit ratio for a given piece of code.

Determine minimum cache size needed to achieve some given hit ratio.

Tune code so that it performs well (or poorly :->) on a given cache.

### Typical Procedure

Determine sequence of memory addresses that code will access.

Split addresses into tag, index, and offset parts.

Use tag, index, and offset of an address to determine if outcome is a hit.

## Simple Sequential Access

The Program:

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
for ( int i=0; i < 1048576; i++ ) sum += a[ i ];
```

First, lets review the address of  $a[i]$ :

Given in program the address of  $a[0]$  is  $0x2000000$ .

Following C syntax, let  $\&a[0]$  denote the address of  $a[0]$ .

Since each element of  $a$  is 4 bytes (see the `sizeof` comment) ...

... the  $\&a[1]$  is  $0x2000004$ ,  $\&a[2]$  is  $0x2000008$ , ...

... generalizing,  $\&a[i]$  is  $0x2000000 + 4 * i$ .

The Program:

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
for ( int i=0; i < 1048576; i++ ) sum += a[ i ];
```

Behavior on a Cold (empty) Direct Mapped Cache with Line Size  $L$

Let  $L$  denote the line size.

First access will miss, bringing in one line of data, ...

... consisting of  $L$  bytes ( $L/4$  elements of  $a$ ).

If  $\&a[0]$  is a multiple of  $L$ , then  $\&a[0]$  is the first element of the line.

The next  $L/4 - 1$  access (to  $a[1]$  through  $a[L/4-1]$ ) will hit.

This pattern will continue, for a hit ratio of  $\frac{L/4-1}{L/4}$ .

## Cache Write Options

### *Write Through:*

A cache feature in which memory (or a higher-level cache) is updated on every store, even if the write address is cached.

### *Write Back:*

A cache feature in which memory is not accessed on a write hit. Data is written into the cache and is only written into memory when the line is replaced.

A cache is either write through or write back.

### *Write Allocate:*

A cache feature in which data is written into a cache (perhaps replacing a line) after a write miss. (See write around.)

### *Write Around:* or No-write allocate

A cache feature in which on a cache miss data is written to memory and not into the cache.

## Set-Associative Cache

### Motivation

A direct-mapped cache can store no more than one line with a particular index.

Poor performance when program frequently accesses two lines with same index.

Example, 4 kiB cache, line size  $L = 2^l = 2^8$ , number of sets  $2^s = 2^4 = 16$ .

---

```
extern char *a; // Big array, allocated elsewhere.
for ( int i=0; i<1024; i++ ) {
    int b = a[ i ];           // At most one hit.
    int c = a[ i + 0x1000 ]; // Always misses.
    d += b + c;
}
```

---

Note, hit rate much better if cache size doubled ...

... or using a *set-associative* cache of same or smaller size.



## Key Ideas

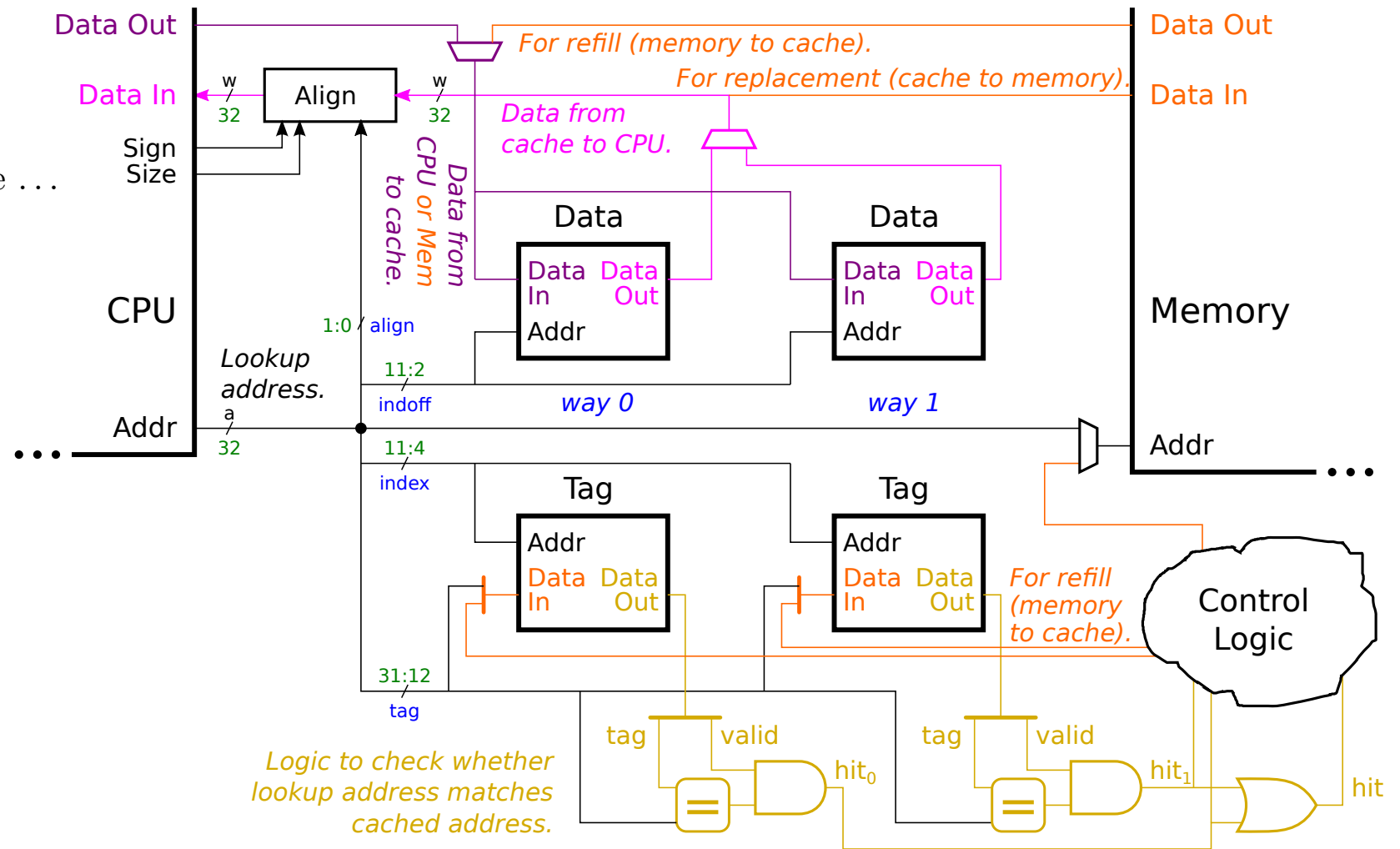
Start with  $x$  direct-mapped caches ...  
 ... each direct-mapped cache called a *way* ...  
 ... and the cache is called  $x$ -way set-associative ...  
 ... the cache has an associativity of  $x$ .

Illustration shows two ways.

In typical set-associative caches ...  
 ... associativity from 2 to 16.

Simultaneously look up line in each way.

Can hold  $x$  lines with same index.



## Definitions

**Set:**

Storage for lines with a particular index.

**Associativity:**

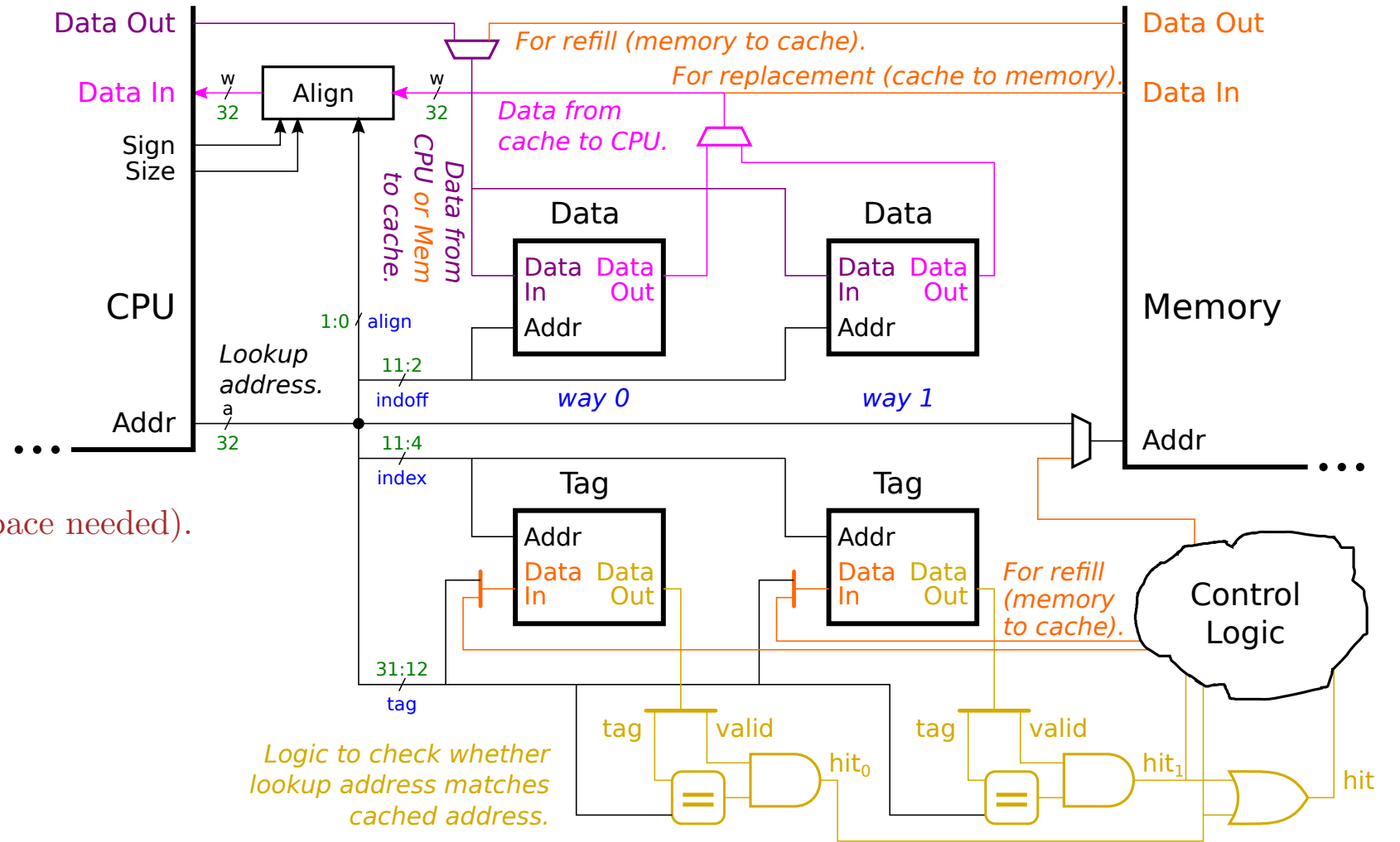
Number of lines that a set can hold.

***x*-Way Set-Associative Cache:**

A cache with associativity *x*.

**Replacement Strategy:**

Method used to choose line to replace (when space needed).



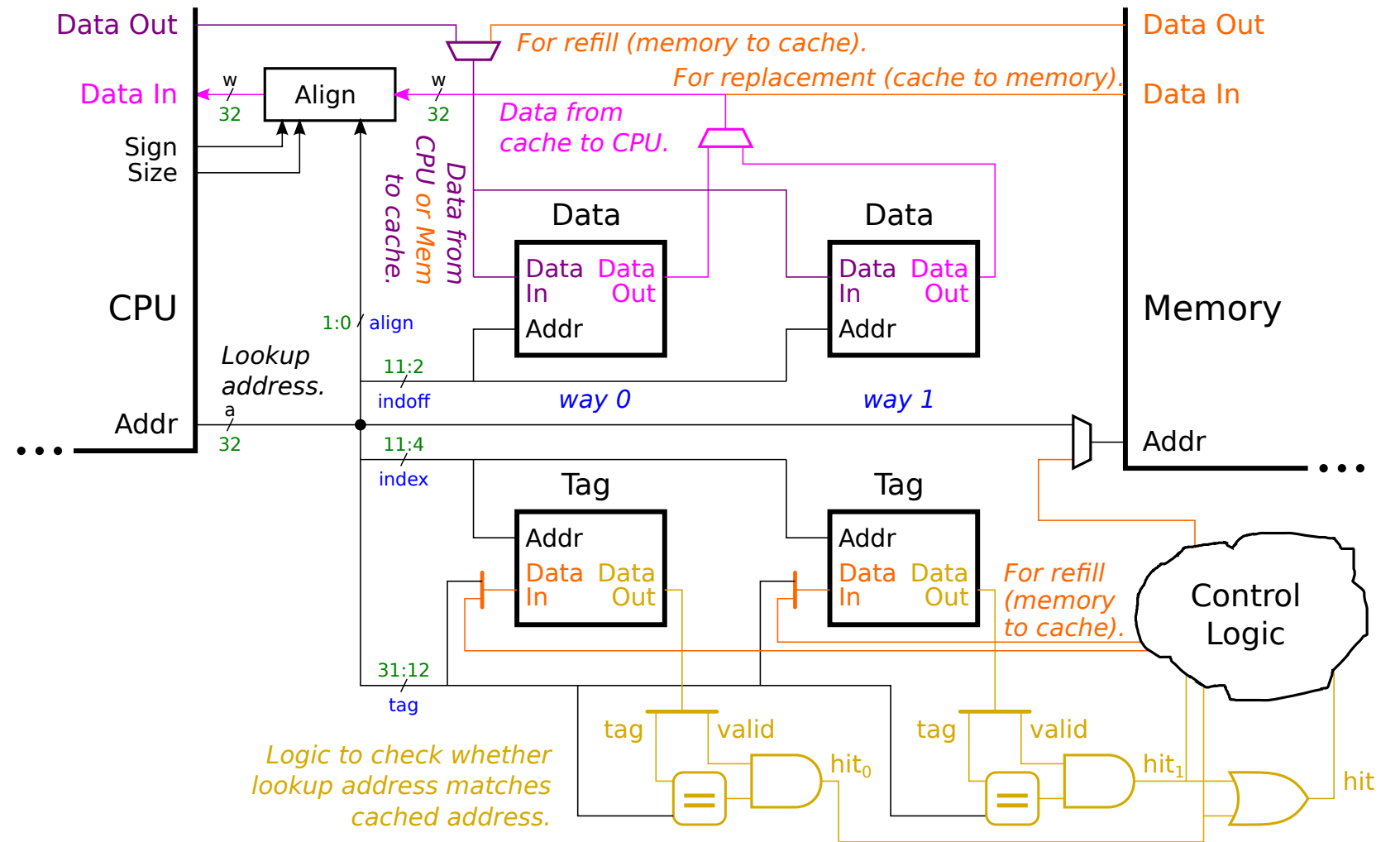
### Cache Example

Memory System:...

...  $a = 32$ ,  $c = 8$  bits, and  $w = 32$  bits.

Two-way associativity, 16-byte lines.

Capacity:  $2 \times 2^{12} = 8192$  B.



## Set-Associative Cache Replacement Strategies

Problem not faced in direct-mapped caches: which line to evict.

### *LRU:*

A replacement strategy in which the least-recently used (accessed) line is considered.

Is effective because address references usually exhibit temporal locality.

Easy to implement if associativity small.

Too time consuming if associativity is large.

If associativity is large ( $> 16$ ) LRU can be approximated.

### *Random*

Usually approximated ...

... for example, using least significant bits of a cycle counter.

Less effective than LRU ...

... but effective enough and easy to implement.

## Miss Classification

*Compulsory:* or *Cold* Miss

A miss to a line that was never cached.

*Conflict Miss:*

A miss on a system that would not occur on one using a fully-associative cache of the same size.

*Capacity Miss:*

A miss on a system that would also occur on one using a fully-associative cache of the same size.

## Set-Associative Cache Special Cases

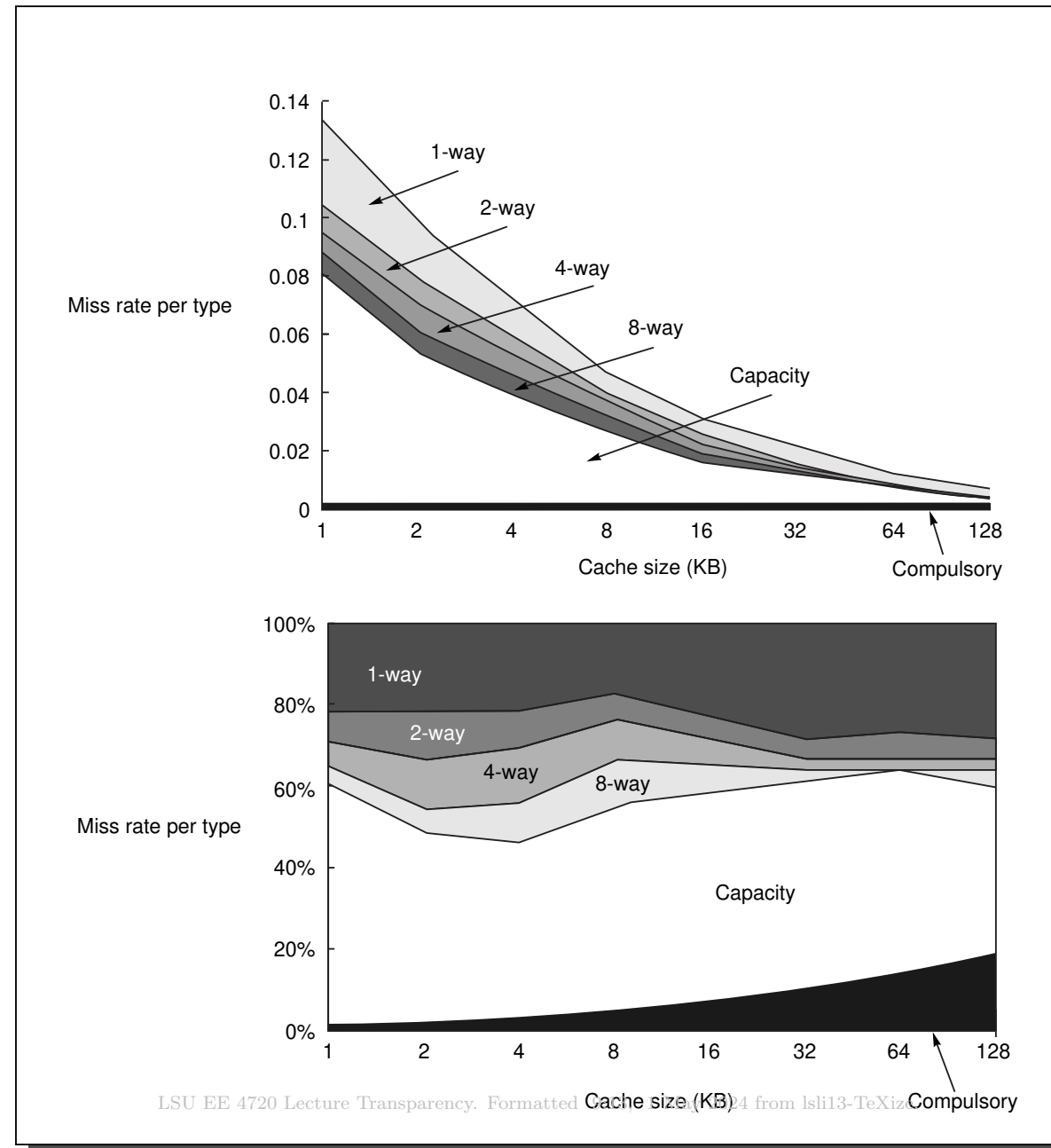
### *Direct Mapped Cache:*

A 1-way set-associative cache (not really set associative).

### *Fully Associative Cache:*

A set-associative cache with one set ( $s = 0$ ).

### Cache Miss Rate v. Size and Associativity



## Set-Associative Cache Sample Program

### What Program Does

First, program exactly fills up cache using minimum number of accesses.

Then, program generates hundreds of misses accessing only five items.

---

```
constexpr int LG_LINE_SIZE = 4;
constexpr int LG_SET_COUNT = 10;
constexpr int ASSOCIATIVITY = 4;

constexpr int SET_COUNT = 1 << LG_SET_COUNT;
constexpr int CACHE_SIZE = ASSOCIATIVITY << ( LG_LINE_SIZE + LG_SET_COUNT );

char a[ 2 * CACHE_SIZE ];

int main(int argv, char **argc)
{
    int elt_sum = 0;
```

---



Fill up cache with minimum number of accesses.

---

```
{
  char *addr = &a[0];
  constexpr int linesize = 1 << LG_LINE_SIZE;

  for ( int w=0; w<ASSOCIATIVITY; w++ )

    for ( int s=0; s<SET_COUNT; s++ )
      {
        elt_sum += *addr;
        /* Below, increment index part of address. */
        addr += linesize;
      }
}
```

---

Generate lots of misses while accessing only five ( $\text{ASSOCIATIVITY} + 1$ ) array elements.

---

```
{
  constexpr int deadly_stride = 1 << ( LG_LINE_SIZE + LG_SET_COUNT );

  for ( int i=0; i<100; i++ )
  {
    char *addr = &a[0];

    for ( int w=0; w<=ASSOCIATIVITY; w++ )
    {
      elt_sum += *addr;
      /* Below: Increment tag part of address. */
      addr += deadly_stride;
    }
  }
}
```

---

## Organization of Caches

Problem with systems using only one cache:

Need multiple ports, one for IF and one or more for load/stores ...

... adding ports increases latency.

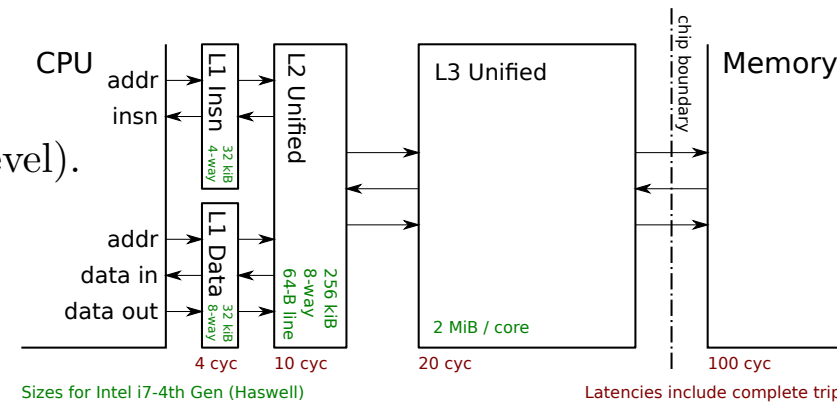
Larger caches have higher latencies ...

... must balance miss ratio with hit latency.

## Solutions

Split cache into separate *instruction* and *data* caches (all at same level).

Split cache into multiple *levels*, abbreviated *L1*, *L2*, etc.



## Split Caches

Idea:

Split caches into independent parts.

Each part used by a different part of CPU.

Common Split: Instruction/Data

### *Instruction Cache:*

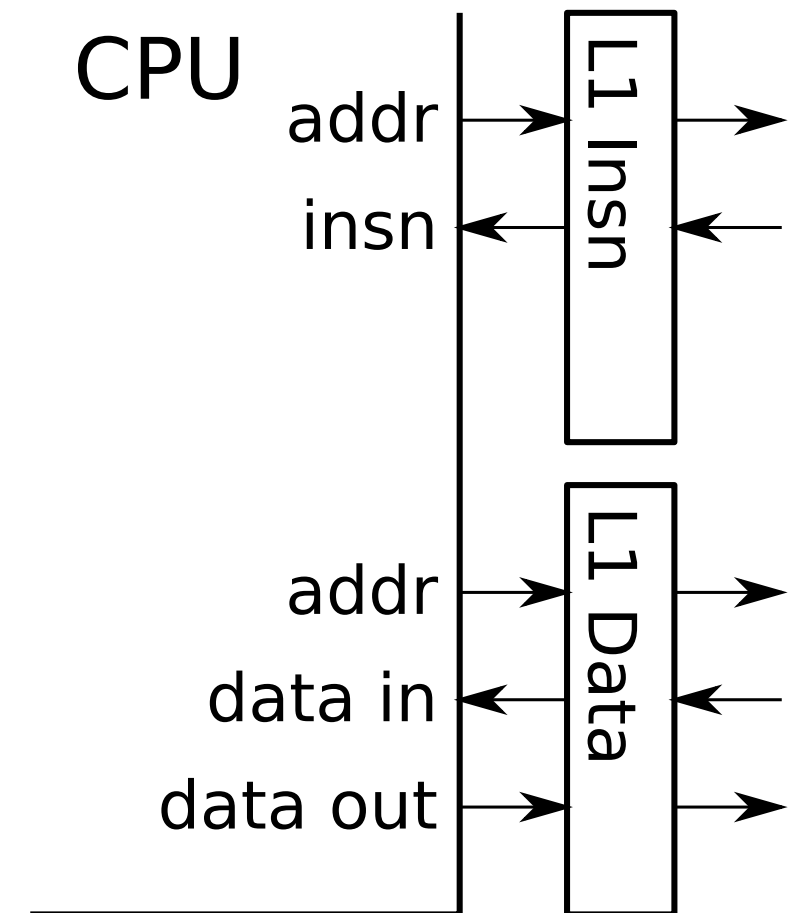
A cache only used for instruction fetches.

### *Data Cache:*

A cache only used for data reads and writes.

### *Unified Cache:*

A cache that can be used for instructions and data.



## Multiple-Level Caches

Due to fabrication technology limitations and addressing logic needed...  
... **cache latency increases** as **cache size increases**.

Latency and hit ratio balanced using multiple cache levels.

Cache level indicates “distance” from CPU.

Lowest-level (level one) cache checked first ...  
... if data not present second level checked ...  
... until data found or there are no more levels.

## Choice of Size for Cache Levels

### Level 1 Criteria

Designed to meet a target hit latency. *E.g.*, one cycle.

To achieve speed may be direct mapped or of low associativity.

On same chip as CPU.

### Level 2 Criteria

Hit latency can be much longer, *e.g.*, 10 cycles.

Latency can be met with a much larger size and associativity.

Often on same chip as CPU.

Size either chosen ...

... to fill the rest of the chip (when L3, if any, is off chip) ...

... or to meet a latency goal (when L3 is on chip).

## Level 3 Criteria

If on chip, fill chip area.

## Defining Hit Ratios in Multiple-Level Caches

### *Global Hit Ratio:*

Number of hits to any level divided by number of accesses to cache.

### *Local Hit Ratio of a Level:*

Number of hits to a level divided by number of access that reach that level.



## Example Cache Organization

### Cache Organization in Intel i7 4th Gen (Haswell).

Note: (Same as Nehalem, Sandy Bridge).

Chip has from 2 to 10 cores, sizes below are per core.

#### Level-One Caches

Separate instruction and data caches.

Each is 32 kiB, data cache is 8-way, insn cache is 4-way.

#### Level-Two Cache

Size is 256 kiB, 8-way.

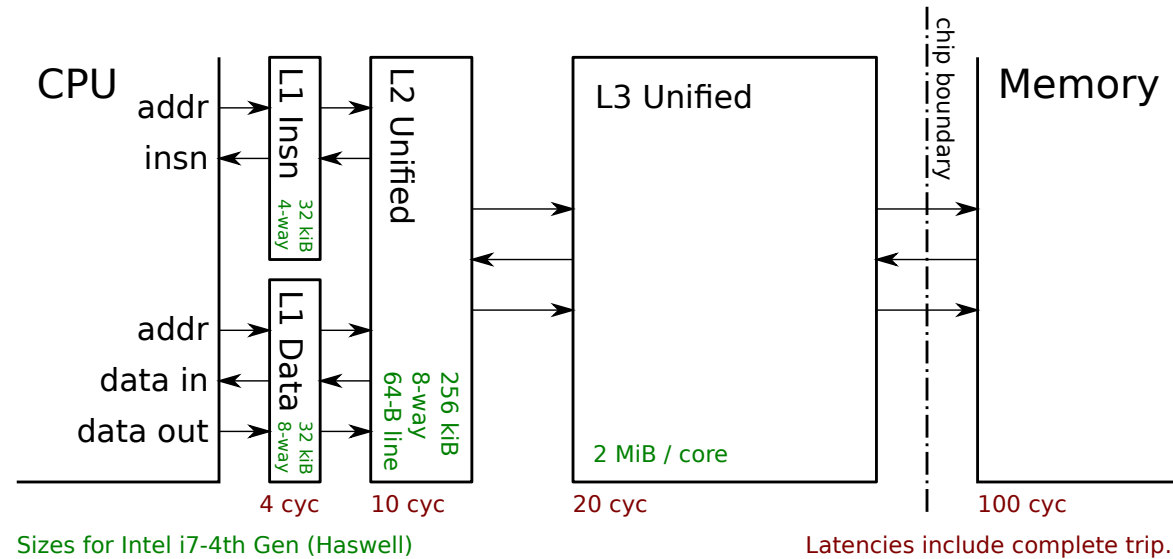
## Level-Three Cache

Size varies with model, about 2 MiB per core.

L3 cache is shared by all cores.

## Example Cache Organization

Cache Organization in Intel i7 4th Gen (Haswell).



Note: Connections to L3 simplified, in particular connection between cores and to memory controller are omitted.

## Miscellaneous Cache Features

### *Lock-Free:* or *Non-Blocking* Cache

A cache that can continue to handle accesses while processing a miss.

### *Multilevel Inclusion:*

Describes a multiple level cache in which any data found in a lower level (closer to CPU) will also be found in the higher level.