Memory and Caches

13-1

See also cache study guide.

Contents

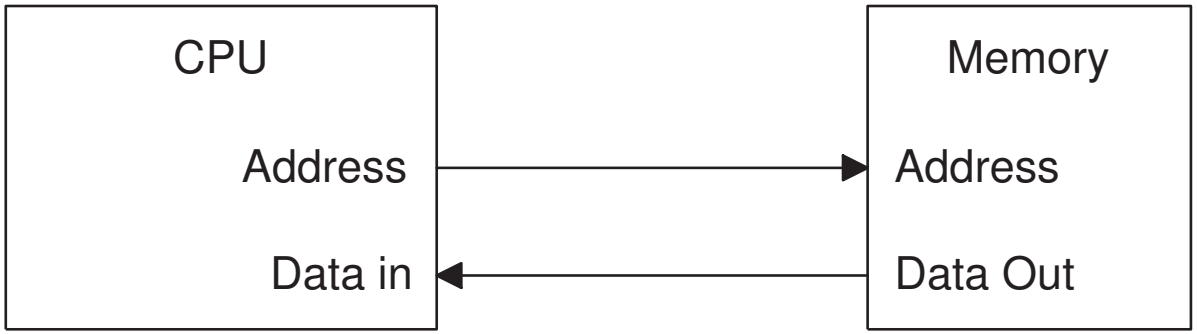
Supplement to material in section 5.2.

Includes notation presented in class.

Memory Basics and Terminology

13-2

In essence a memory system is quite simple:



Retrieve or store data at specified effective address.

In practice its complexity can rival that of the processor.

EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

13-2

Terminology

13-3

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.

Typical address space sizes:

DLX, MIPS32, Sun SPARC V8, IA-32, and other older ISAs, $a = 32$,

Itanium, MIPS64, DEC Alpha, Sun SPARC V9, $a = 64$.

Typical Address Spaces

13-4

Real Address Space: a.k.a. *Physical Address Space*
The set of addresses used by main memory.

Real Address:
An address in a real address space.

Given a real address, can open up computer's case and point to data.

Virtual Address Space:
The set of addresses usually used by programs.

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.

13-5

Usage of Virtual Address

Instructions use virtual addresses.

Memory devices use physical addresses.

Hardware between CPU and memory translates from virtual to physical address.

Depending on design, cache uses virtual or physical addresses.

Each process (running program) gets its own virtual address space.

Processor can switch between virtual and real addresses.

Only real addresses used in this set of notes.

Terminology

13-6

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.

13-7

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.

Example

13-8

```
! 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
```

13-9

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

13-10

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 $w = 128\,\mathrm{b} = 16\,\mathrm{B}$:

13-11

```
! r1 = 0x1003
! MEM[0x1000..0x100F] = 0x80 0x81 0x82 ... 0x8f
 lb r2, 0(r1)   ! Bus contents: 0x80 0x81...0x8f, r2<-0xffffff83
 lu 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 `lu` 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.

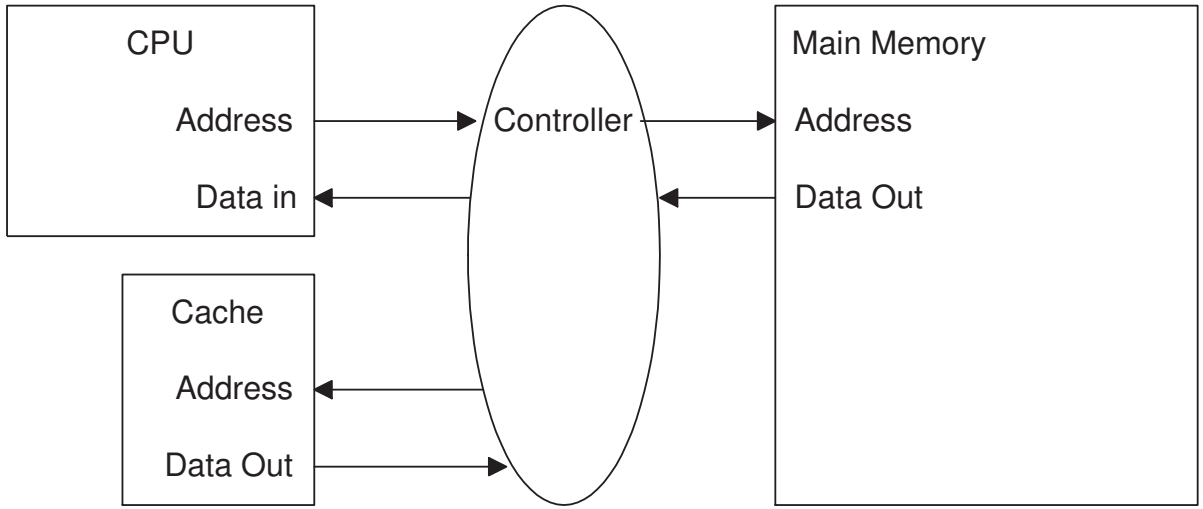
13-12

Cache Motivation

Loads and stores are performed frequently.

Fast memory devices are expensive and size limited and ...
... cheap memory devices are slow.

By organizing these devices into a *cache* performance will be ...
... almost as fast as expensive devices ...
... almost as cheap as slow devices.



13-12

13-13

Cache 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 in expensive devices, data returned quickly.

Called a *hit*.

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

Called a *miss*.

13-14

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.

Hit:

Data corresponding to the effective address found in the cache.

Miss:

Data corresponding to the effective address not found in the cache and had to be read from memory (or a higher cache level).

13-15

Organization of Simple Cache

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.

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, $l = \log_2 L$.

13-16

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.

Memory (Effective) Address Relationship to Cache Structure

13-17

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

For

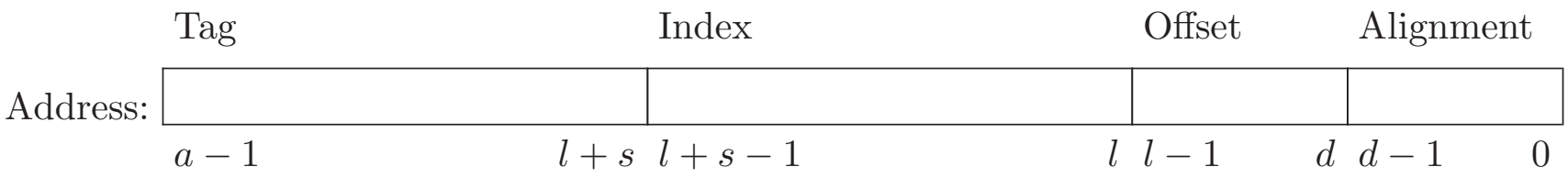
a-bit address space,

2^s-set cache,

2^l-character lines,

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

the fields would be as follows:



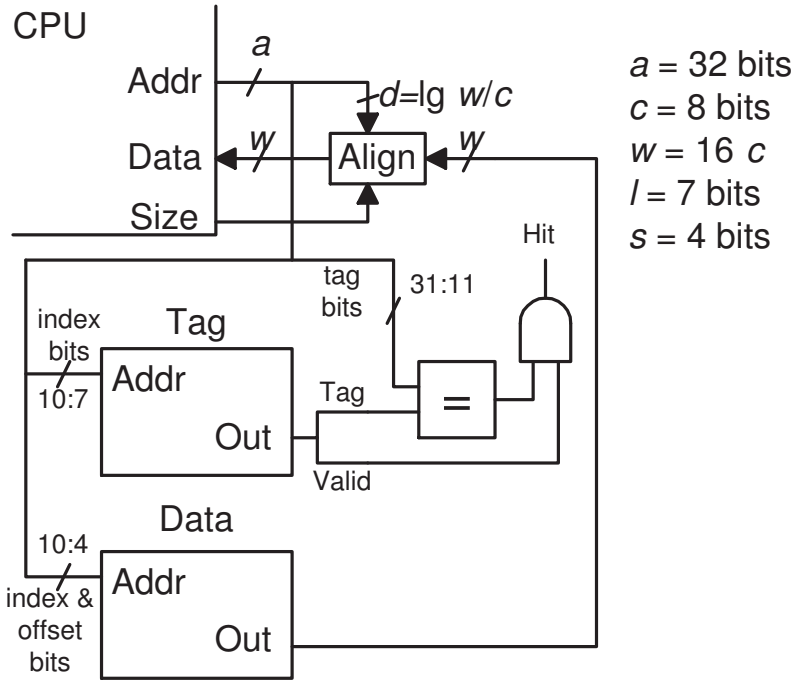
EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

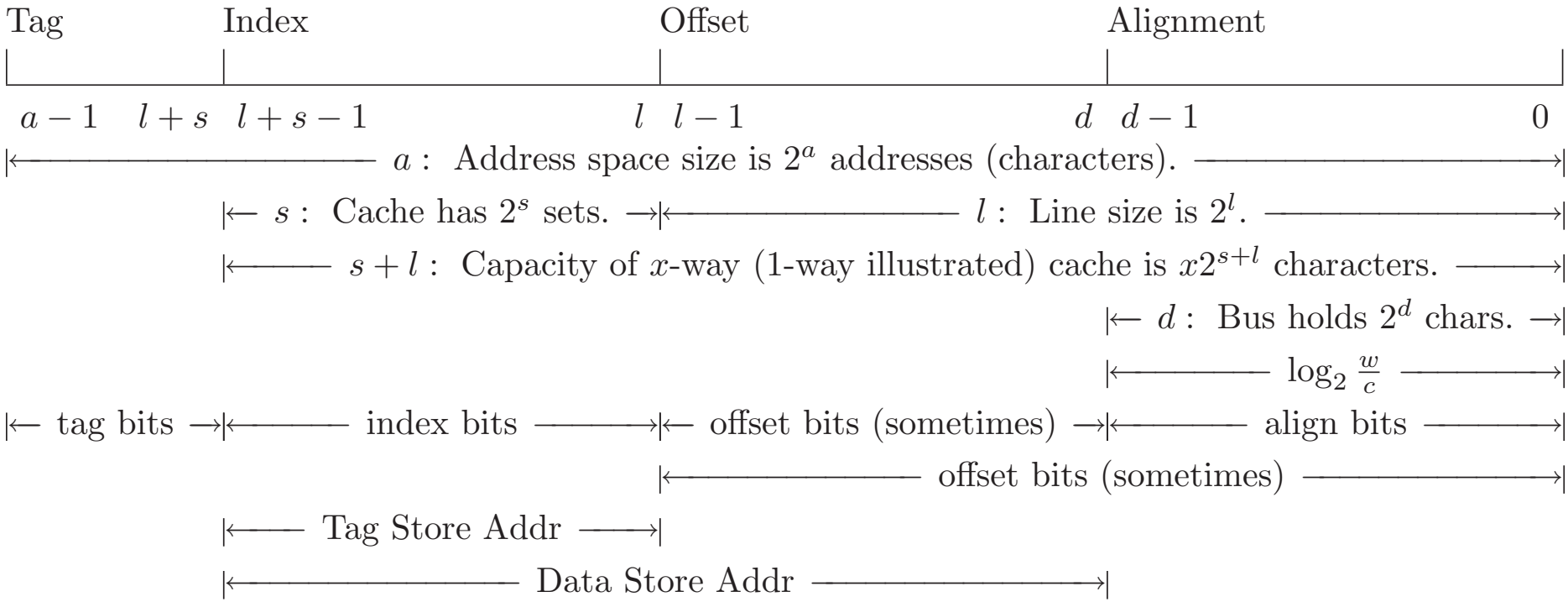
13-17

13-18

More Cache Structure and Address Bits

Details on later slides.





EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

13-18

13-19

Alignment Bits:

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

Bits: $0 : (d - 1)$

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

Assuming aligned fetches, the d least significant bits of an address will either be zero or ignored.

This part of the address is ignored by the cache.

These bits are labeled "offset" in figure 5.8. Three of those five offset bits are also alignment bits.

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.

13-20

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.

13-21

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

13-22

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).

13-23

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.

Cache Miss Rate v. Block Size

13-24

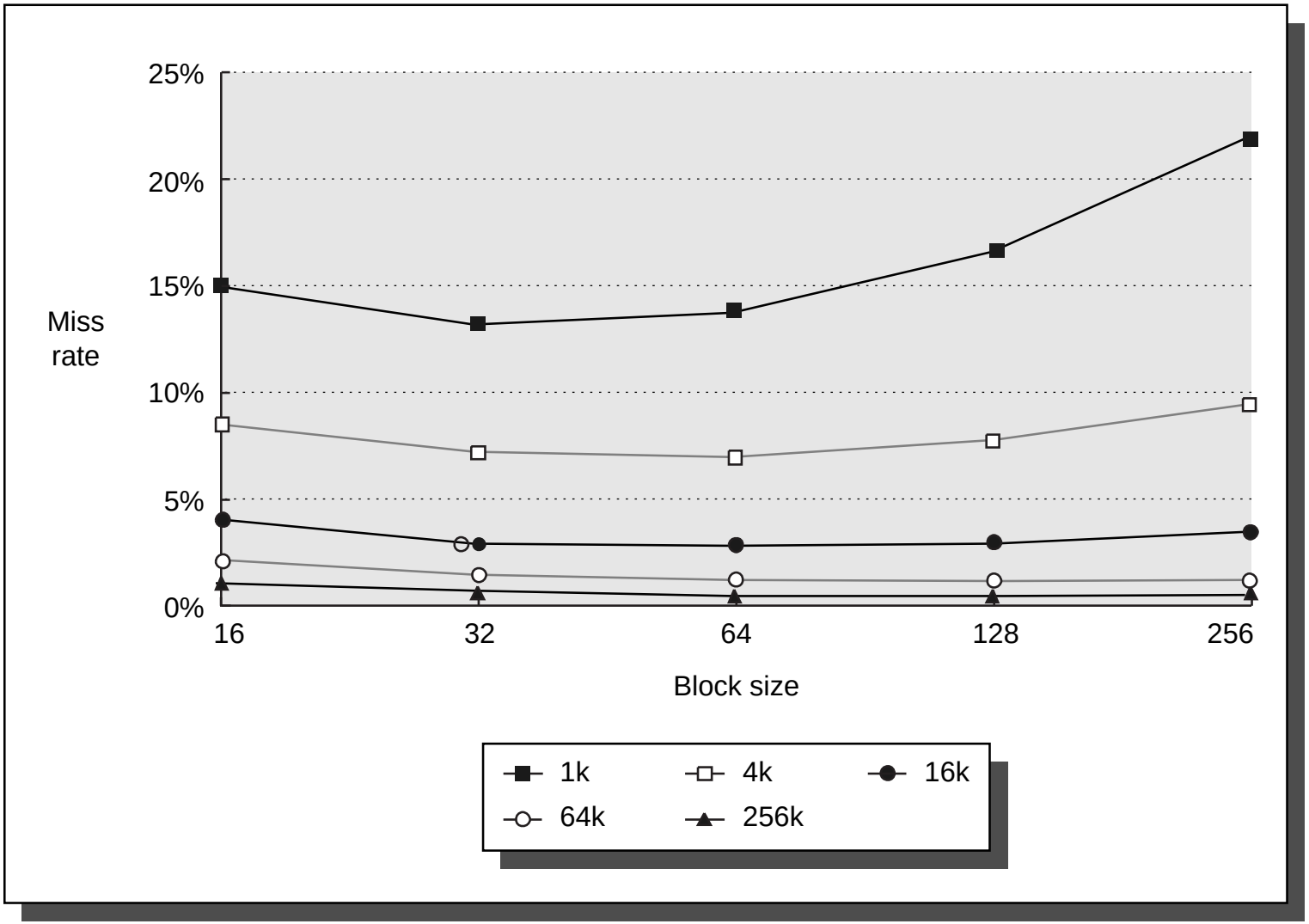


FIGURE 5.11 Miss rate versus block size for five different-sized caches.

EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

13-24

Set-Associative Cache

13-25

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(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.

Set-Associative Cache

13-26

Idea

Duplicate a direct mapped cache x times ...
... each duplicate sometimes called a *way*.

In typical set-associative caches associativity from 2 to 8.

Simultaneously look up line in each duplicate.

Can hold x lines with same index.

Set:
Storage for lines with a particular index.

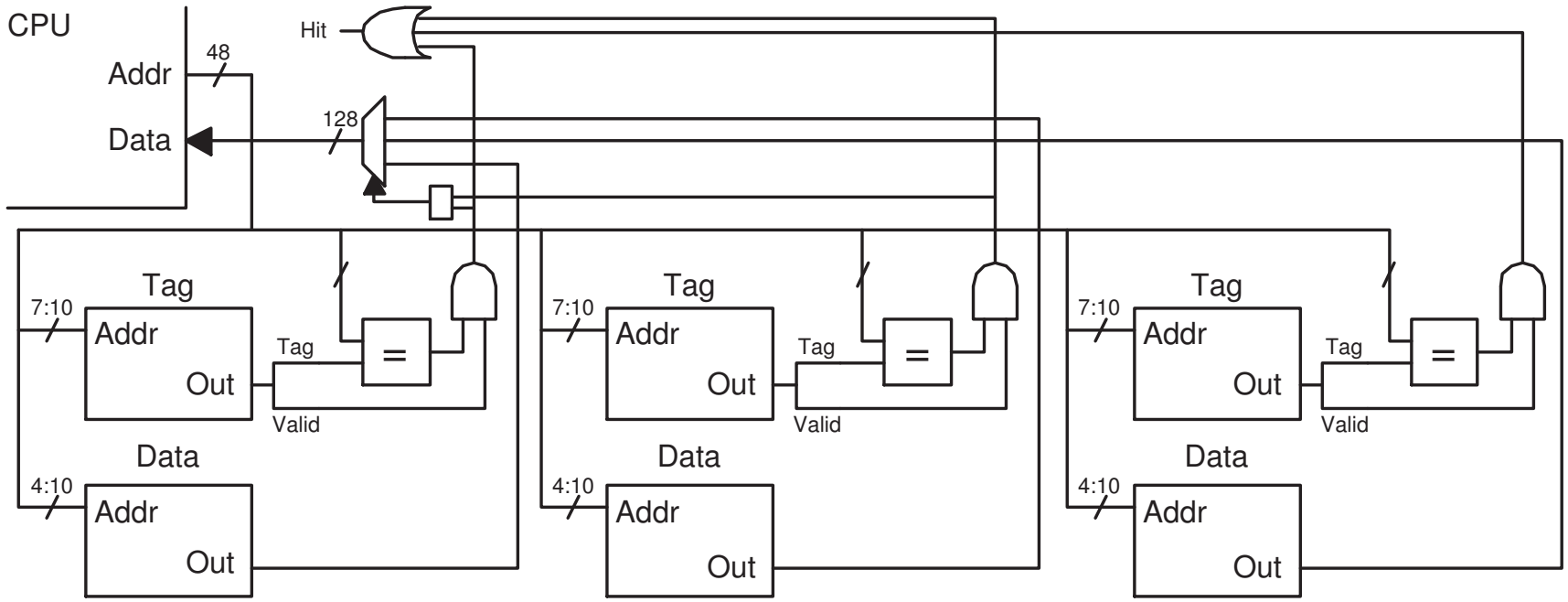
Associativity:
Number of lines that a set can hold.

x-Way Set-Associative Cache:
A cache with associativity x.

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

Set-Associative Cache Example

13-27



Memory System: $a = 48$, $c = 8$ bits, and $w = 128$ bits.

Cache: Three-way associativity (set holds 3 lines), 128-byte lines (each line holds 8 bus-widths of data), 16 sets.

Capacity: $3 \times 16\,\mathrm{B} \times 2^{10-4+1} = 6144\,\mathrm{B}$.

EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

13-27

Set-Associative Cache Replacement Strategies

13-28

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.

EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.

13-28

Miss Types

13-29

Miss Categories

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

13-30

Direct Mapped Cache:

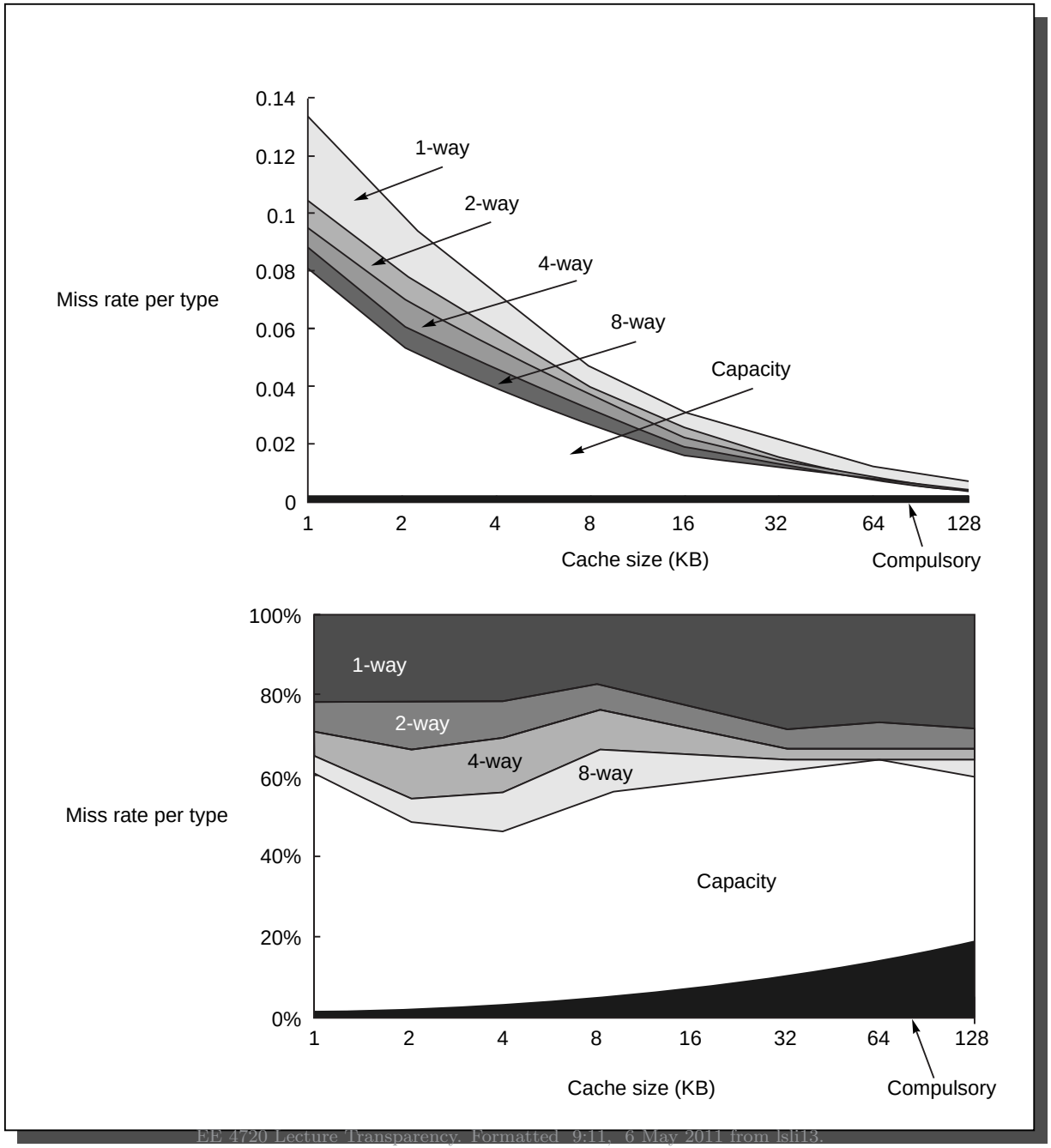
A 1-way set-associative cache (not really set associative).

Fully Associative Cache:

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

Cache Miss Rate v. Size and Associativity

13-31



13-31

FIGURE 5.10 Total miss rate (top) and distribution of miss rate (bottom) for each

Set-Associative Cache Sample Program

13-32

What Program Does

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

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

```
#include <stdio.h>

#define LG_LINE_SIZE 4
#define LG_SET_COUNT 10
#define ASSOCIATIVITY 4

#define SET_COUNT (1<<LG_SET_COUNT)
#define CACHE_SIZE ( ASSOCIATIVITY << ( LG_LINE_SIZE + LG_SET_COUNT ) )

char a[ 2 * CACHE_SIZE ];

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

13-33

Fill up cache with minimum number of accesses.

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

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

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


EE 4720 Lecture Transparency. Formatted 9:11, 6 May 2011 from lsli13.
13-33

13-34

Generate lots of misses while accessing only five (ASSOCIATIVITY + 1) array elements.

```
{
  int i, w;
  int deadly_stride = 1 << ( LG_LINE_SIZE + LG_SET_COUNT );

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

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

Organization of Caches

13-35

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 multiple *levels*.

Split cache into independent parts (all at same level).

Split Caches

13-36

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

13-37

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.

Lowest Level Criteria

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

For speed may be direct mapped or low associativity.

Almost always on same chip as CPU.

13-38

Second Level Criteria

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

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

Sometimes on same chip as CPU ...
... sometimes tags on CPU chip while data off-chip ...
... sometimes entire second-level cache off-chip.

Third Level Criteria

If present, large enough so that a direct mapped organization would have few conflict misses.

Defining Hit Ratios in Multiple-Level Caches

13-39

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.

Typical Cache Organization

13-40

Cache split into three levels.

Separate level-one instruction and data caches ...
... both on same chip as CPU.

First-level cache typically write through (to second level, which is nearby).

Unified second-level cache.

Second-level cache tags on CPU but data off chip ...
... allowing a level-two hit to be detected early.

Second level cache typically write back, especially if there is no third level cache.

Unified third-level cache.

Third level cache usually write back.

Miscellaneous Cache Features

13-41

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.