

---

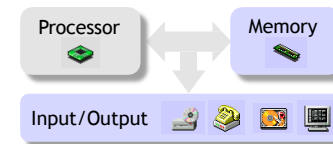
## CS 305 Computer Architecture

### Cache Introduction

---

## Welcome to the Memory System

- We've already seen how to make a processor. How can we supply the CPU with enough data to keep it busy?
- This part of CS250 focuses on **memory** issues, which are frequently bottlenecks that limit the performance of a system.
- We'll start off by looking at memory systems for the next 2 weeks
  - How caches can dramatically improve the speed of memory accesses.
  - How virtual memory provides security and ease of programming
  - How processors, memory and peripheral devices can be connected



## Large and fast

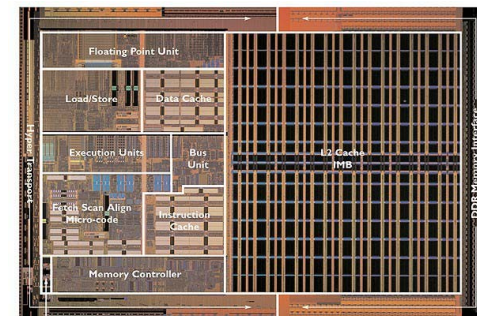
- Today's computers depend upon large and fast storage systems.
  - Large storage capacities are needed for many database applications, scientific computations with large data sets, video and music, and so forth.
  - Speed is important to keep up with our pipelined CPUs, which may access both an instruction and data in the same clock cycle. Things get become even worse if we move to a superscalar CPU design.
- So far we've assumed our memories can keep up and our CPU can access memory twice in one cycle, but as we'll see that's a simplification.



3

## Today: Cache introduction

- Today we'll answer the following questions.
  - What are the challenges of building big, fast memory systems?
  - What is a cache?
  - Why caches work?
  - How are caches organized?
    - Where do we put things -and- how do we find them?



4

### Small or slow

- Unfortunately there is a tradeoff between speed, cost and capacity.

Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

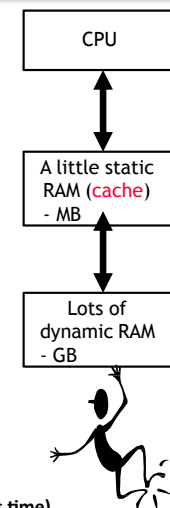
- Fast memory is too expensive for most people to buy a lot of.
- But dynamic memory has a much longer delay than other functional units in a datapath. If every lw or sw accessed dynamic memory, we'd have to either increase the cycle time or stall frequently.

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~\$5	128KB-2MB
Dynamic RAM	100-200 cycles	~\$0.10	128MB-4GB
Hard disks	10,000,000 cycles	~\$0.0005	20GB-400GB

5

### Solving the Tradeoff

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- We do this by introducing a **cache**, which is a small amount of fast, expensive memory.
  - The cache goes between the processor and the slower, dynamic main memory.
  - It keeps a copy of the most frequently used data from the main memory.
- Memory access speed increases overall, because we've made the common case faster.
  - Reads and writes to the most frequently used addresses will be serviced by the cache.
  - We only need to access the slower main memory for less frequently used data.



$$AMAT = (\text{hit ratio} * \text{cache access time}) + ((1 - \text{hit ratio}) * \text{memory access time})$$

*when will such a system work well?*

6

## The principle of locality

- It's usually difficult or impossible to figure out what data will be “*most frequently accessed*” before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of *temporal locality* says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of *spatial locality* says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

Cache introduction

7

## Temporal locality in code

- The principle of *temporal locality* says that if a program accesses one memory address, there is a good chance that it will access the same address again.
- *Loops* are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- For example:

```

Loop: lw    $t0, 0($s1)
      add   $t0, $t0, $s2
      sw    $t0, 0($s1)
      addi  $s1, $s1, -4
      bne   $s1, $0, Loop
  
```

- Each instruction of the loop body will be fetched over and over again, once on every loop iteration.

Cache introduction

8

### Temporal locality in data

- Programs often access the same variables over and over, especially within loops. Below, `sum` and `i` are repeatedly read and written.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

- Commonly-accessed variables can sometimes be kept in registers, but this is not always possible. Why?
  - There are a limited number of registers.
  - You can't take the address of a register (i.e., create a pointer to it)
  - There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.

Cache introduction

9

### Spatial locality in code

- The principle of **spatial locality** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

```
sub $sp, $sp, 16
sw $ra, 0($sp)
sw $s0, 4($sp)
sw $a0, 8($sp)
sw $a1, 12($sp)
```

- Every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location  $i$ , then we will probably also execute the next instruction, at memory location  $i+1$ .
- Code fragments such as loops exhibit both temporal and spatial locality.*

Cache introduction

10

### Spatial locality in data

- Programs often access data that is stored contiguously.
  - Arrays, like `a` in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like `employee` are also kept contiguously in memory.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

Cache introduction

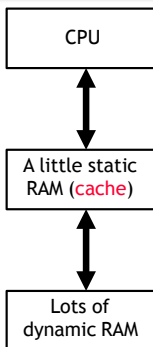
11

### Recap

- We want large, fast, cheap memory systems - conflicting goals because fast memory is expensive
- Compromise - use a small amount of fast memory to CACHE data/instructions used frequently and load this from the slower, larger, cheaper memory.
- Hope that programs reuse instructions and data. Luckily, it turns out that most programs exhibit LOCALITY.
- Temporal Locality
- Spatial Locality
- Each applies to both code and data.

### How caches take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower dynamic memory.
  - So *the first read is a little slower than subsequent reads* since it goes through both main memory and the cache, but subsequent reads are much faster.
- This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.

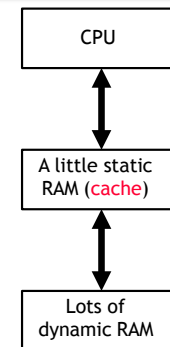


Cache introduction

13

### How caches take advantage of spatial locality

- When the CPU reads location  $i$  from main memory, a copy of that data is placed in the cache.
- But *instead of just copying the contents of location  $i$ , we can copy several values into the cache at once, such as the four bytes from locations  $i$  through  $i + 3$ .*
  - If the CPU later does need to read from locations  $i + 1$ ,  $i + 2$  or  $i + 3$ , it can access that data from the cache and not the slower main memory.
  - For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.
- Again, *the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data.*



Cache introduction

14

### Definitions: Hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.
- A **cache miss** occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.
- There are two basic measurements of cache performance.
  - The **hit rate** is the percentage of memory accesses that are handled by the cache.
  - The **miss rate** (1 – hit rate) is the percentage of accesses that must be handled by the slower main RAM.
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.

$$\text{AMAT} = (\text{hit ratio} * \text{cache access time}) + ((1 - \text{hit ratio}) * \text{memory access time})$$

### A simple cache design

- **Caches are divided into blocks**, which may be of various sizes.
  - The number of blocks in a cache is usually a power of 2.
  - For now we'll say that each block contains one byte. This won't take advantage of spatial locality, but we'll do that next time.
- Here is an example cache with eight blocks, each holding one byte.

Block index	8-bit data
000	
001	
010	
011	
100	
101	
110	
111	

Cache introduction

16



### Four important questions



1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

Cache introduction

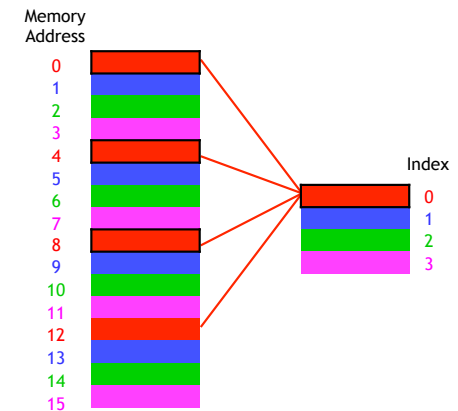
17

### Where should we put data in the cache?

- A **direct-mapped** cache is the simplest approach: each main memory address maps to exactly one specific cache block.

- For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks).
- Memory locations **0, 4, 8** and **12** all map to cache block **0**.
- Addresses **1, 5, 9** and **13** map to cache block **1**, etc.

- **How can we compute this mapping?**



18

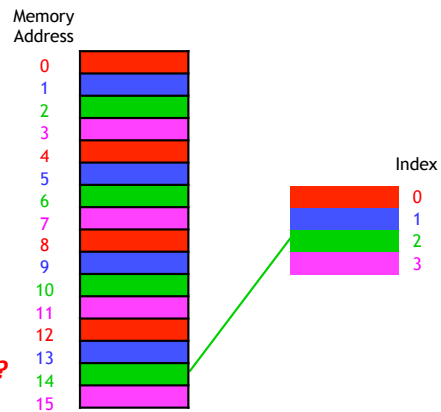
### It's all divisions...

- One way to figure out which cache block a particular memory address should go to is to use the **mod (remainder)** operator.
- If the cache contains  $2^k$  blocks, then the data at memory address  $i$  would go to cache block index

$$i \bmod 2^k$$

- For instance, with the four-block cache here, address 14 would map to cache block 2.

*But.... Isn't division expensive??*

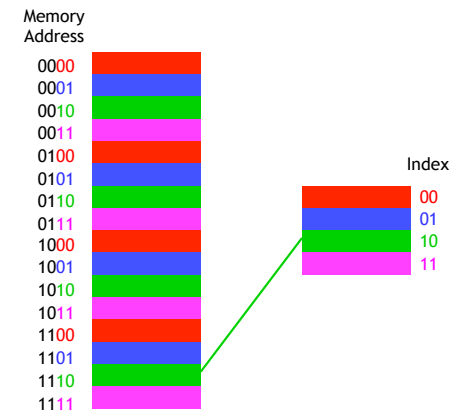


Cache introduction

19

### ...or least-significant bits

- An equivalent way to find the placement of a memory address in the cache is to look at the least significant  $k$  bits of the address.
- With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- Taking the least  $k$  bits of a binary value is the same as computing that value mod  $2^k$ .

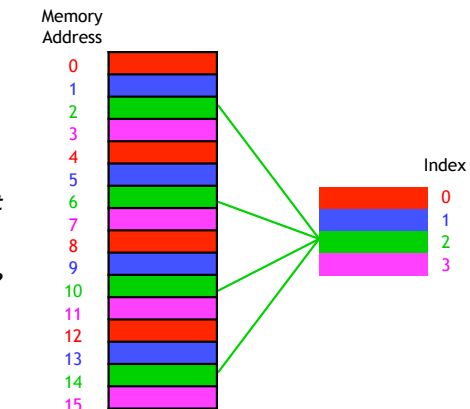


Cache introduction

20

### How can we find data in the cache?

- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address  $i$ , we can use the same mod trick to determine which cache block would contain  $i$ .
- *But other addresses might also map to the same cache block. How can we distinguish between them?*
- For instance, cache block 2 could contain data from addresses 2, 6, 10 or 14.

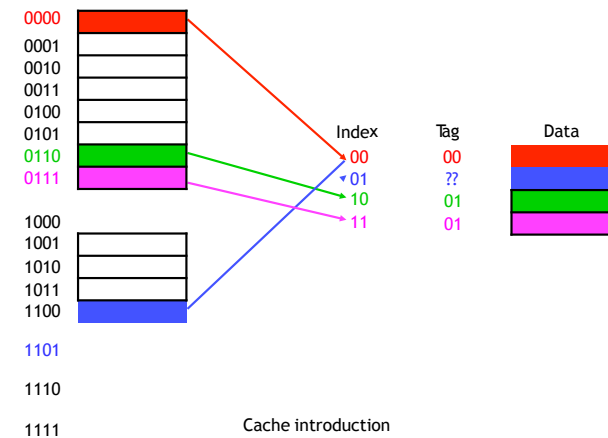


Cache introduction

21

### Adding tags

- We need to add **tags** to the cache, which supply the rest of the address bits to let us *distinguish between different memory locations that map to the same cache block*.







Cache introduction

22

### Figuring out what's in the cache

- Now we can tell exactly which addresses of main memory are stored in the cache, *by concatenating the cache block tags with the block indices.*

Index	Tag	Data	Main memory address in cache block
00	00		$00 + 00 = 0000$
01	11		$11 + 01 = 1101$
10	01		$01 + 10 = 0110$
11	01		$01 + 11 = 0111$







Cache introduction

24

### One more detail: How do we recognize if the cache block is actually populated?

- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.

Index	Valid Bit	Tag	Data	Main memory address in cache block
00	1	00		$00 + 00 = 0000$
01	0	11		Invalid
10	0	01		???
11	1	01		???

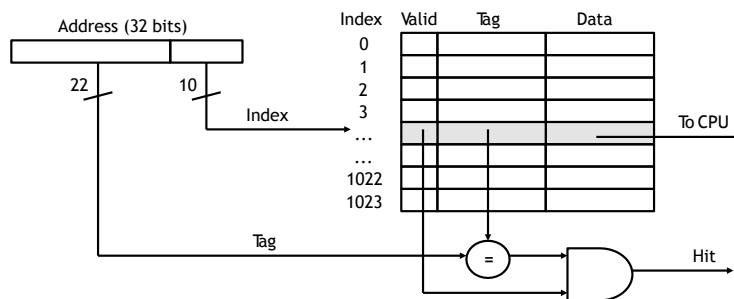
- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity
- It has overhead - a 1 MB L2 cache does not mean 1MB is available for data storage.

Cache introduction

25

### What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a **cache controller**.
    - The lowest  $k$  bits of the address will index a block in the cache.
    - If the block is valid **AND** the tag matches the upper  $(m - k)$  bits of the  $m$ -bit address, then that data will be sent to the CPU.
- What **hardware** components would you need for this control?



Cache introduction

27

### What happens on a cache miss

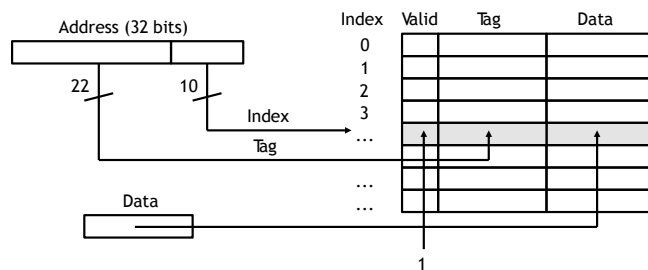
- The delays that we've been assuming for memories (e.g., 2ns) are really assuming cache hits.
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger.
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time.
- However, a much slower main memory access is needed on a cache miss. The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache).

Cache introduction

28

### Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest  $k$  bits of the address specify a cache block.
  - The upper  $(m - k)$  address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



Cache introduction

29

### What if the cache fills up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address.
- We answered this question implicitly on the last page!
  - A miss causes a new block to be loaded into the cache, automatically **overwriting** any previously stored data.
  - This is a **least recently used** replacement policy, which assumes that older data is less likely to be requested than newer data.

Cache introduction

30

### Question: How big is the cache?

For a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:

It is **direct-mapped** (as discussed last time)

Each block holds **one byte**

The cache/block index is the **four** least significant **bits**

**Two questions:**

How many blocks does the cache hold?

$$2^4 = 16$$

How many bits of storage are required to build the cache (e.g., for the data array, tags, etc.)?

**Data = 16 bytes = 128 bits**

**Valid bits = 16**

**Index bits =  $16 \times 4 = 64$  bits**

**Tags = 12 bits each => total =  $12 \times 16$  bits = 192 bits.**

### More cache organizations



Now, we'll explore some cache organizations to improve hit rate

How can we take advantage of spatial locality too?

How can we reduce the number of potential conflicts?

October 21, 2016

30

## Spatial locality

One-byte cache blocks don't take advantage of **spatial locality**, which predicts that an access to one address will be followed by an access to a nearby address.

What can we do?

October 21, 2016

More cache organizations

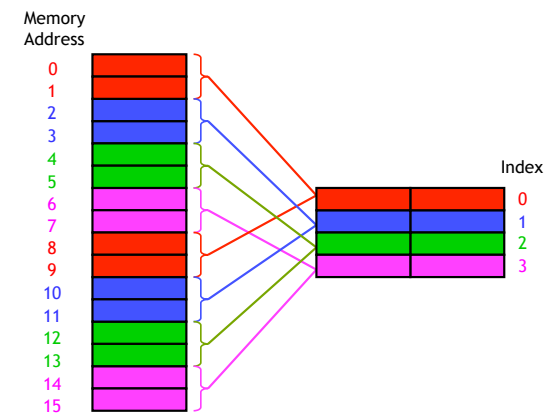
31

## Supporting Spatial locality

**Make the cache block size *larger than one byte*.**

Here we use two-byte blocks, so we can load the cache with two bytes at a time.

If we read from address 12, the data in addresses 12 and 13 would both be copied to cache block 2.



October 21, 2016

More cache organizations

32



## Block addresses

Now, how can we figure out where data should be placed in the cache?

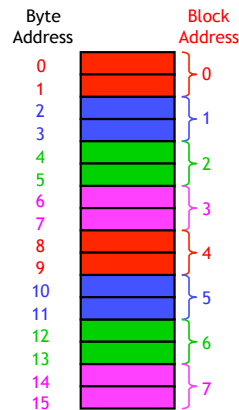
It's time for **block addresses**! If the cache block size is  $2^n$  bytes, we can conceptually split the main memory into  $2^n$ -byte chunks too.

To determine the block address of a byte address  $i$ , you can do the integer division

$$i / 2^n$$

Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an "8-block" main memory instead.

For instance, memory addresses 12 and 13 both correspond to block address 6, since  $12 / 2 = 6$  and  $13 / 2 = 6$  (integer division).

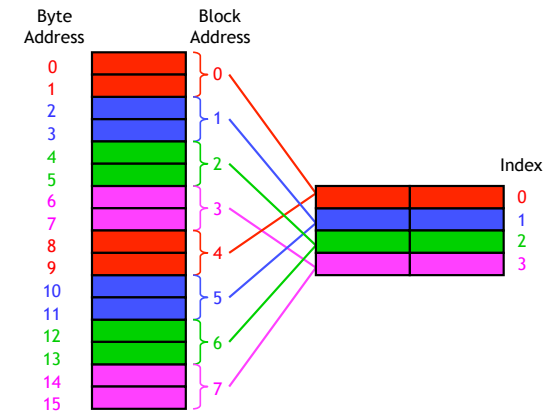


## Cache mapping

Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks.

In our example, memory block 6 belongs in cache block 2, since  $6 \bmod 4 = 2$ .

This corresponds to placing data from memory byte addresses 12 and 13 into cache block 2.



October 21, 2016

More cache organizations

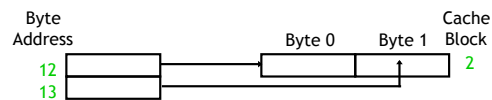
34

### Data placement within a block

When we access one byte of data in memory, we'll copy its entire *block* into the cache, to hopefully take advantage of spatial locality.

In our example, if a program reads from byte address 12 we'll load all of memory block 6 (both addresses 12 and 13) into cache block 2. Note byte address 13 corresponds to the *same* memory block address! So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2.

To make things simpler, byte  $i$  of a memory block is always stored in byte  $i$  of the corresponding cache block.



October 21, 2016

More cache organizations

35

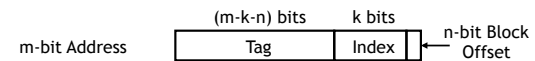
### Locating data in the cache

Let's say we have a cache with  $2^k$  blocks, each containing  $2^n$  bytes.

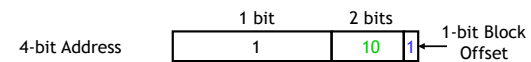
We can determine where a byte of data belongs in this cache by looking at its address in main memory.

$k$  bits of the address will select one of the  $2^k$  cache blocks.

The lowest  $n$  bits are now a **block offset** that decides which of the  $2^n$  bytes in the cache block will store the data.



Our example used a  $2^2$ -block cache with  $2^1$  bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2.

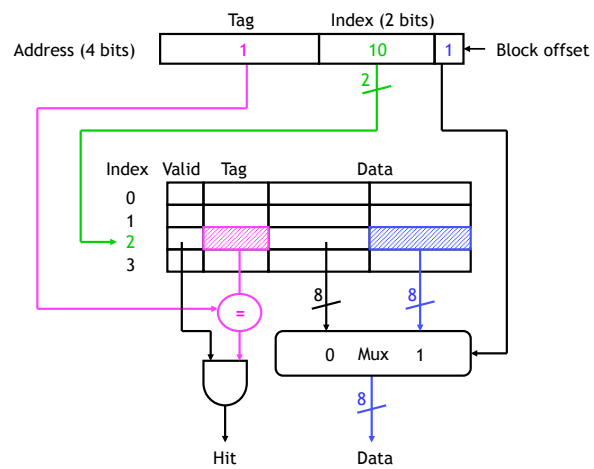


October 21, 2016

More cache organizations

36

## The new look cache!

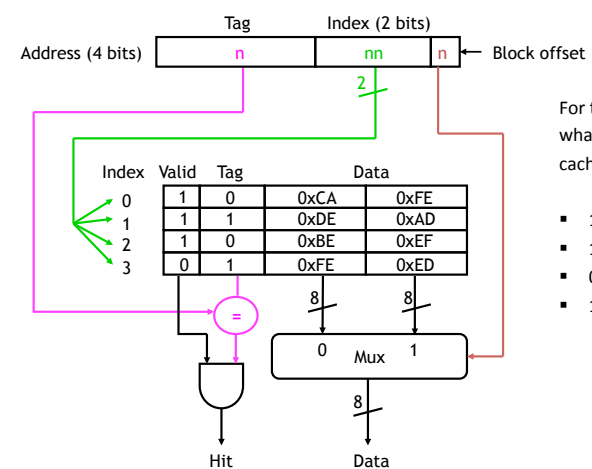


October 21, 2016

More cache organizations

37

## An exercise



For the addresses below,  
what byte is read from the  
cache (or is there a miss)?

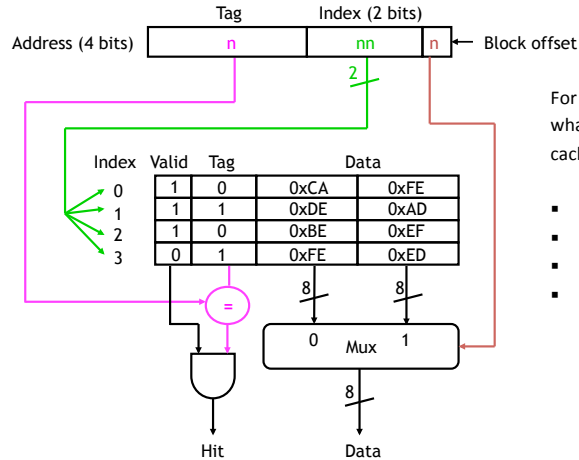
- 1010
- 1110
- 0001
- 1101

October 21, 2016

More cache organizations

38

## An exercise



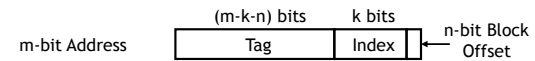
October 21, 2016

More cache organizations

39

## Using arithmetic

An equivalent way to find the right location within the cache is to use arithmetic again.



We can find the **index** in two steps, as outlined earlier.

Do integer division of the address by  $2^n$  to find the block address.

Then mod the block address with  $2^k$  to find the index.

The **block offset** is just the memory address mod  $2^n$ .

For example, we can find address 13 in a 4-block, 2-byte per block cache.

The block address is  $13 / 2 = 6$ , so the index is then  $6 \bmod 4 = 2$ .

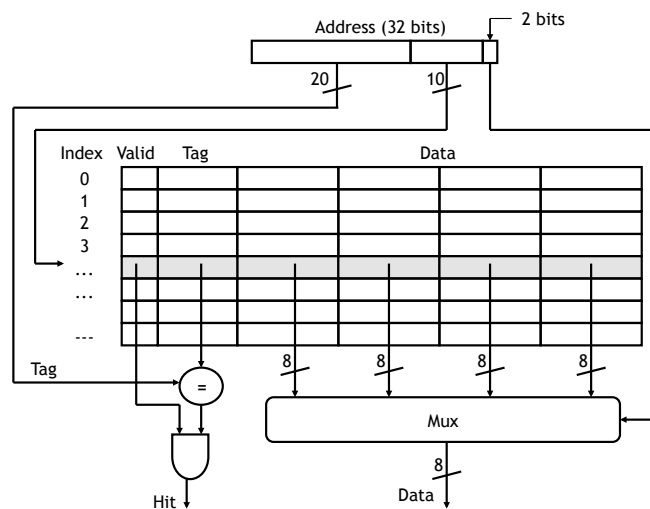
The block offset would be  $13 \bmod 2 = 1$ .

October 21, 2016

More cache organizations

40

### How large is this cache??



### A larger example cache mapping

Where would the byte from memory address 6146 be stored in this direct-mapped  $2^{10}$ -block cache with  $2^2$ -byte blocks?

We can determine this with the binary force.

6146 in binary is 00...01 1000 0000 00 10.

The lowest 2 bits, 10, mean this is the second byte in its block.

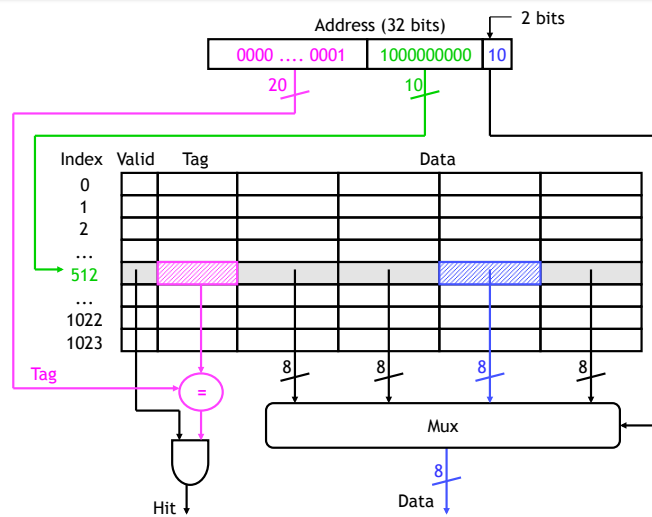
The next 10 bits, 1000000000, are the block address itself (512).

Equivalently, you could use your arithmetic mojo instead.

The block address is  $6146 / 4 = 1536$ , so the index is  $1536 \bmod 1024$ , or 512.

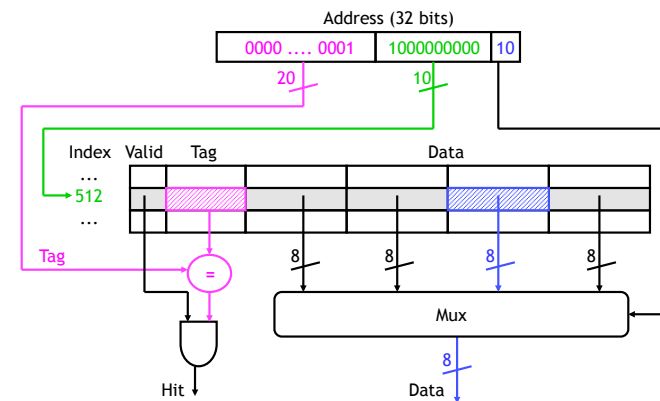
The block offset is  $6146 \bmod 4$ , which equals 2.

### The solution - pictorially speaking



### What goes in the rest of that cache block?

The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (00...01).



October 21, 2016

More cache organizations

44

## The rest of that cache block

Again, byte  $i$  of a memory block is stored into byte  $i$  of the corresponding cache block.

In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively.

You can also look at the lowest 2 bits of the memory address to find the block offsets.

Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147

Index	Valid	Tag	Data			
...						
512						
...						

October 21, 2016

More cache organizations

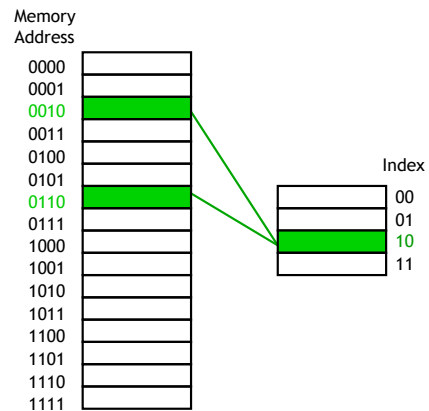
45

## RECAP

### Disadvantage of direct mapping

The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.

But, what happens if a program uses addresses 2, 6, 2, 6, 2, ...?



October 21, 2016

More cache organizations

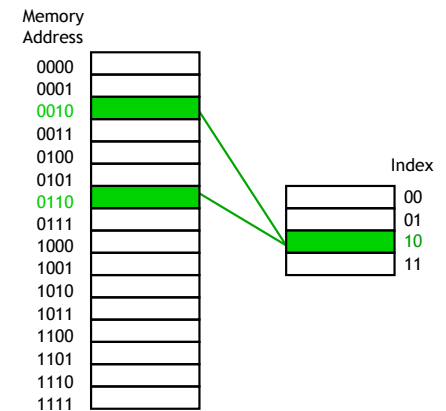
47

### Disadvantage of direct mapping

The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.

However, this isn't really flexible. If a program uses addresses 2, 6, 2, 6, 2, ..., then each access will result in a cache miss and a load into cache block 2.

This cache has four blocks, but direct mapping might not let us use all of them. This will result in more misses than we might like.



October 21, 2016

More cache organizations

48



### A possible solution: A fully-associative cache

A fully-associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block. When data is fetched from memory, it can be placed in *any* unused block of the cache.

*This way we'll never have a conflict between two or more memory addresses that map to a single cache block.*

In the previous example, we might put memory address 2 in cache block 2, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 would all be hits instead of misses.

If all the blocks are already in use, it's usually best to replace the **least recently used** one, assuming that if it hasn't used it in a while, it won't be needed again anytime soon.

October 21, 2016

More cache organizations

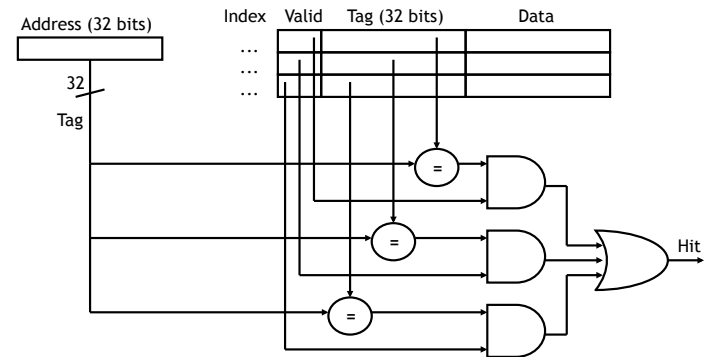
49

### But at what price?

However, a fully-associative cache is expensive to implement.

Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.

Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!



## A compromise: Set associativity

An intermediate possibility is a **set-associative cache**.

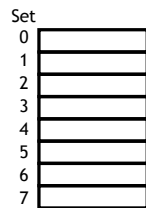
The cache is divided into *groups* of blocks, called **sets**.

Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.

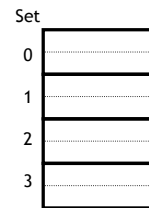
If each set has  $2^k$  blocks, the cache is a  **$2^k$ -way associative cache**.

Here are several possible organizations of an eight-block cache.

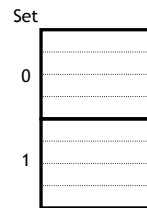
direct mapped  
8 "sets", 1 block each



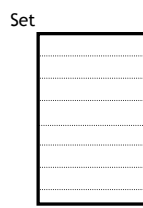
2-way associativity  
4 sets, 2 blocks each



4-way associativity  
2 sets, 4 blocks each



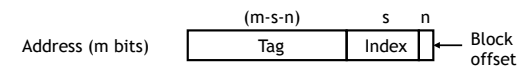
8-way associativity  
1 set, 8 blocks



## Locating a set associative block

We can determine where a memory address belongs in an associative cache in a similar way as before.

If a cache has  $2^s$  sets and each block has  $2^n$  bytes, the memory address can be partitioned as follows.



Our arithmetic computations now compute a **set index**, to select a *set* within the cache instead of an individual block.

$$\text{Block Offset} = \text{Memory Address} \bmod 2^n$$

$$\text{Block Address} = \text{Memory Address} / 2^n$$

$$\text{Set Index} = \text{Block Address} \bmod 2^s$$

### Where would this address go?

Where would data from memory byte address 6195 be placed, assuming the different eight-block cache designs, with 16 bytes per block?

6195 in binary is 00...0110000 011 0011.

53

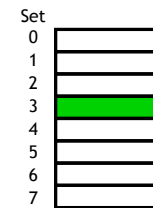
### Block replacement

Any empty block in the correct set may be used for storing data.

If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.

For highly associative caches, it's expensive to keep track of what's really the least recently used block, so some approximations are used. We won't get into the details.

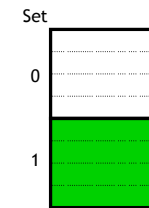
1-way associativity  
8 sets, 1 block each



2-way associativity  
4 sets, 2 blocks each



4-way associativity  
2 sets, 4 blocks each



October 21, 2016

More cache organizations

54

## LRU example

Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.

assume distinct addresses go to distinct blocks

addresses	0	Tags	1	LRU
A	--	--		0
B				
A				
C				
B				
A				
B				

April 21, 2003

More cache organizations

55

## LRU example

Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.

assume distinct addresses go to distinct blocks

On a miss, we  
replace the LRU.

On a hit, we just  
update the LRU.

addresses	0	Tags	1	LRU
miss A	--	--		0
miss B	A	--		1
A	A	B		0
miss C	A	B		1
miss B	A	C		0
A	B	C		1
miss A	B	A		0
B	B	A		1

April 21, 2003

More cache organizations

56

### Another Strategy - MRU

Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.

assume distinct addresses go to distinct blocks

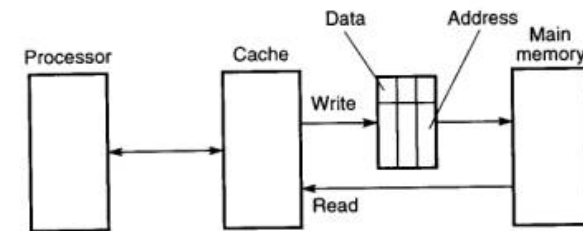
On a miss, we replace the MRU.

On a hit, we just update the MRU.

addresses	0	Tags	1	MRU
miss A	--	--	--	X
miss B	A	--	--	0
A	A	B	--	1
miss C	A	B	--	0
miss A	C	B	--	0
miss B	A	B	--	0
B	A	B	--	1
miss D	A	D	--	1

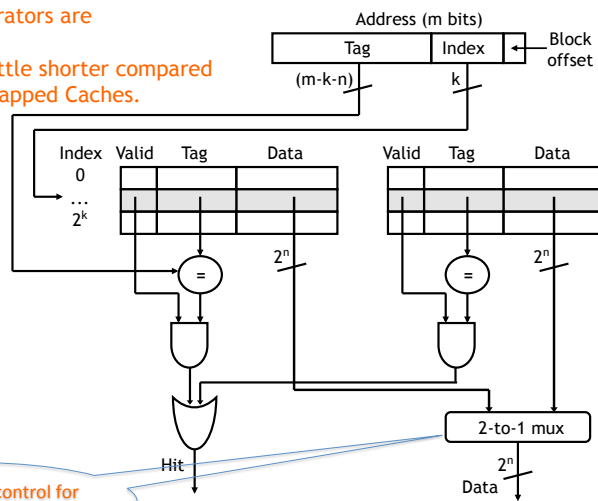
### What happens when you write to a cache block?

- **Writethrough** - When the processor does a write to a cached location, then it is **also written into the memory location** at the same time.
- **Writeback** - Only **when the cache block** is replaced is the change written back to memory if the cache block is dirty.
- Pros and Cons for Reading and Writing?



## 2-way set associative cache implementation

Only two comparators are needed.  
The tags are a little shorter compared with Direct Mapped Caches.



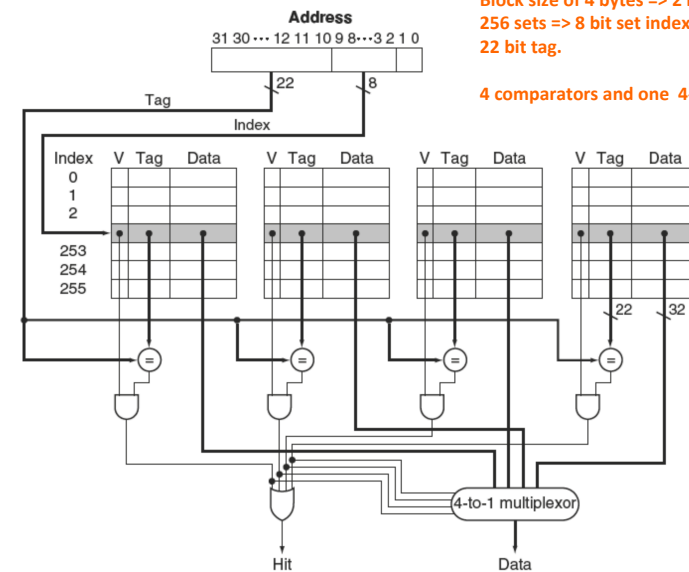
Where does the control for this MUX come from?

59

## Four Way Set Associative Cache

Block size of 4 bytes => 2 bit block offset  
256 sets => 8 bit set index  
22 bit tag.

4 comparators and one 4-1 Mux



## Summary

---

Larger **block** sizes can take advantage of **spatial locality** by loading data from not just one address, but also nearby addresses, into the cache.

**Associative caches** assign each memory address to a particular set within the cache, but not to any specific block within that set.

Set sizes range from 1 (**direct-mapped**) to  $2^k$  (**fully associative**).

Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.

In practice, 2-way through 16-way **set-associative caches strike a good balance between lower miss rates and higher costs.**