ECE/CS 250
Computer Architecture

Summer 2023

Caches and Memory Hierarchies

Tyler Bletsch

Duke University

Slides are derived from work by
Daniel J. Sorin (Duke), Amir Roth (Penn), and Alvin Lebeck (Duke)

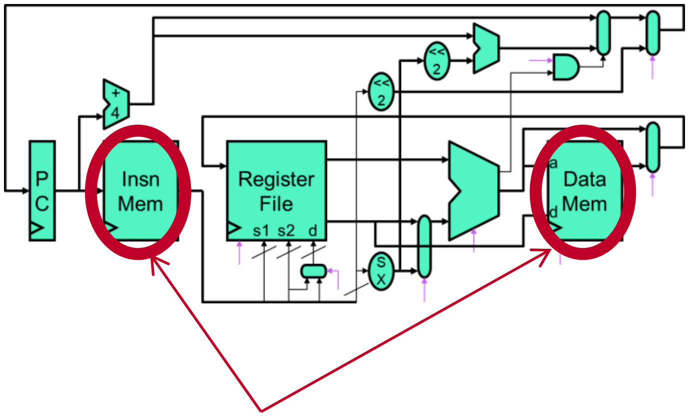
Where We Are in This Course Right Now

- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We have assumed that memory storage (for instructions and data) is a magic black box
- Now:
  - We learn why memory storage systems are hierarchical
  - We learn about caches and SRAM technology for caches
- Next:
  - We learn how to implement main memory

Readings

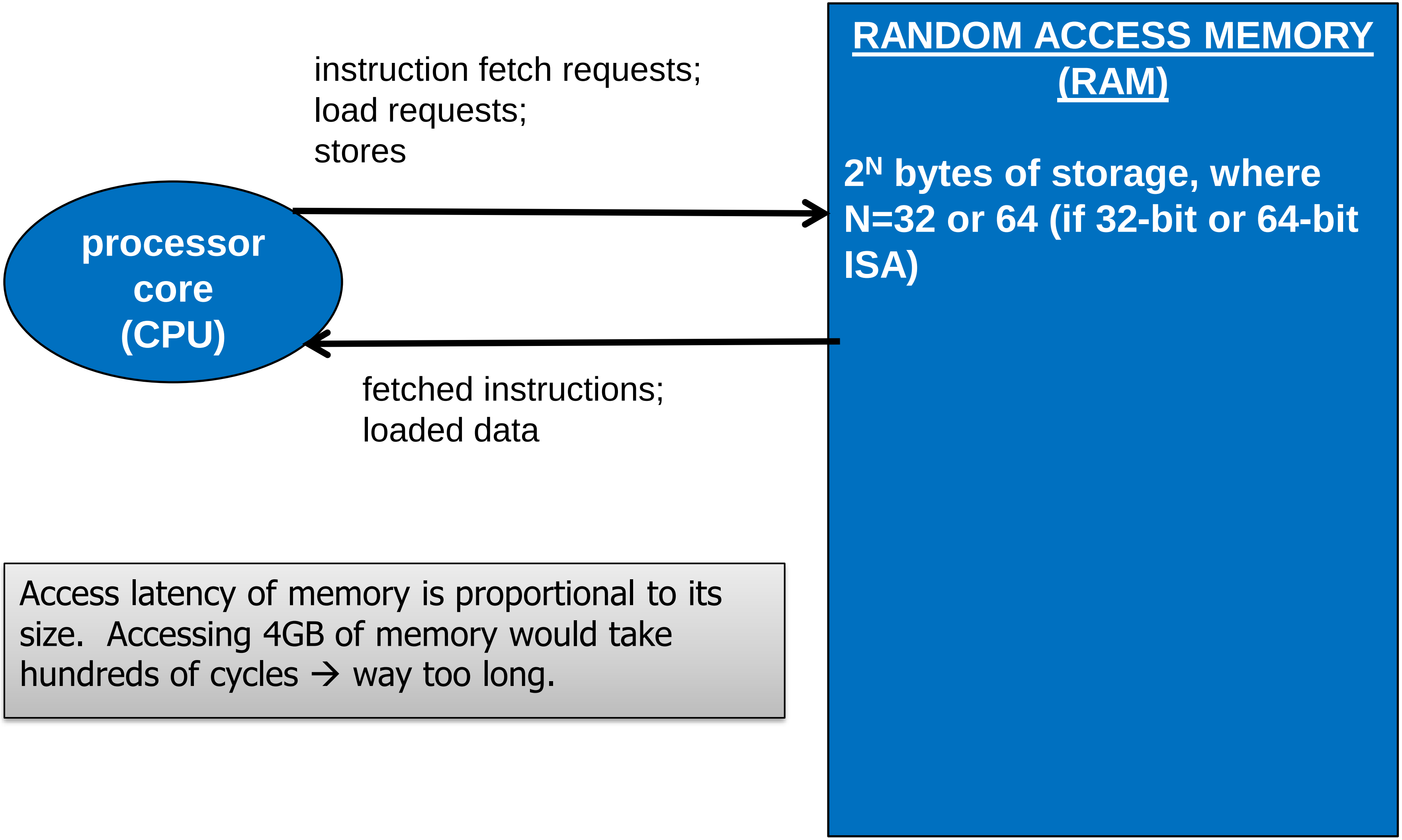
- Patterson and Hennessy
  - Chapter 5

What is memory made of?



???

Computer layout (as far as you know so far)



instruction fetch requests;
load requests;
stores

processor
core
(CPU)

fetched instructions;
loaded data

RANDOM ACCESS MEMORY (RAM)

2^N bytes of storage, where N=32 or 64 (if 32-bit or 64-bit ISA)

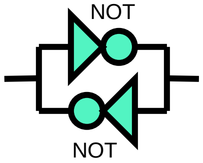
Access latency of memory is proportional to its size. Accessing 4GB of memory would take hundreds of cycles → way too long.

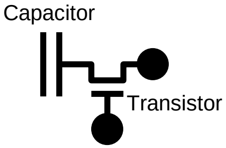
What is RAM made of?

- We *could* implement RAM as a vast number of D flip-flops
  - Too big! Our goal is density (bits/area)
  - D Flip-flop is ~32 transistors!

Let's use analog circuit properties to make more efficient RAM…

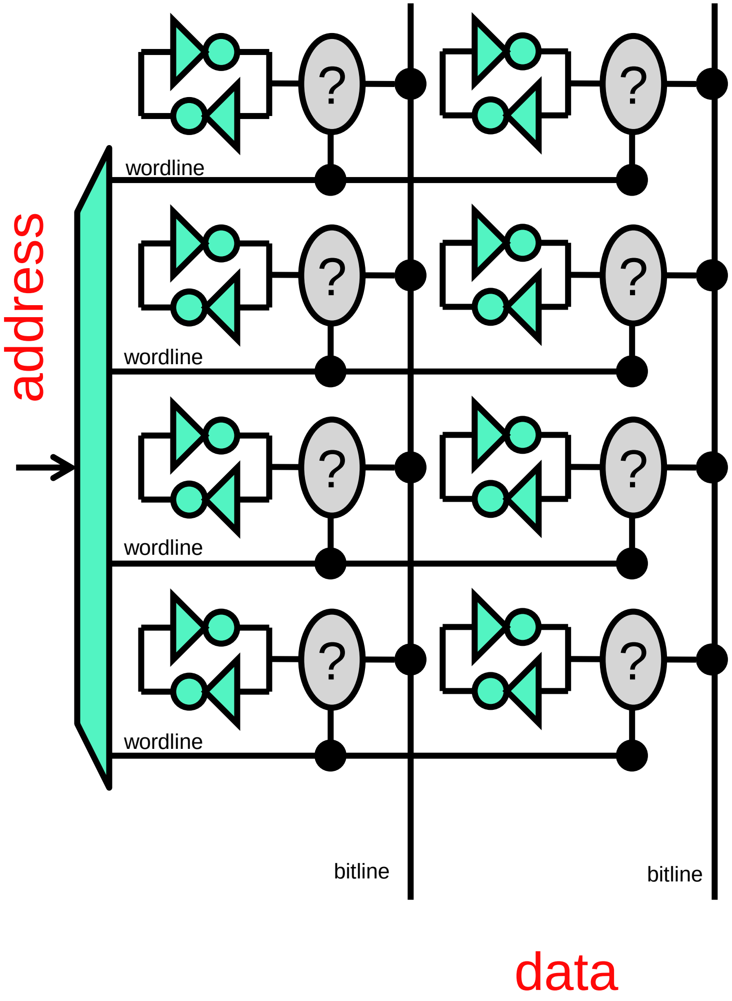
- Two main types of RAM:
  - **Static RAM (SRAM)**
    - Expensive, fast, usually fairly small
    - Bits stored in two NOT gates (4 transistors)
  - **Dynamic RAM (DRAM)**
    - Cheap, slower, can be very large
    - Bits stored in capacitors (with 1 transistor)
    - Capacitors slowly drain and need to be refilled:
      Need to <u>refresh</u> data in DRAM periodically (makes it slower)





Static Random Access Memory (SRAM)

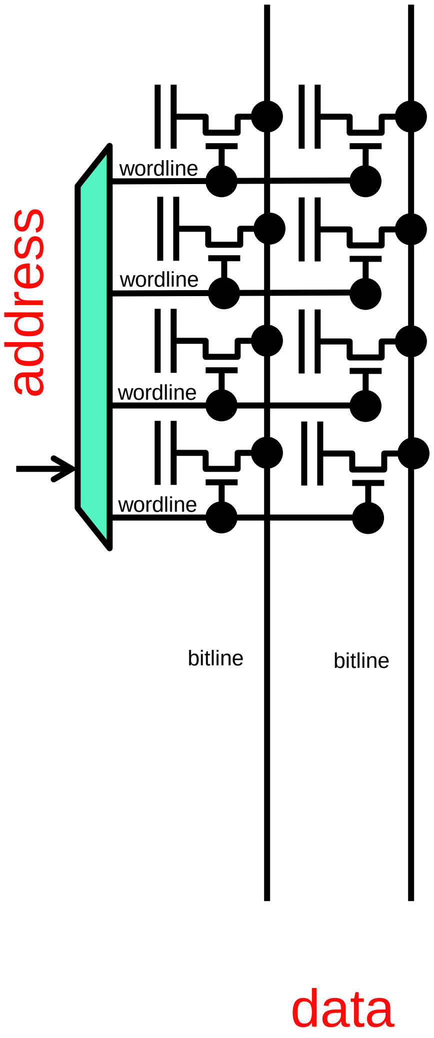
- Implemented as a big 2D array:
  - One dimension is which word do you want ("wordlines")
  - The other dimension are the bits of that word ("bitlines")
  - Slides at end of deck go deeper
  - Why "static"?
    - A written bit maintains its value (doesn't leak out)
    - But still volatile → bit loses value if chip loses power
- Designed for speed



SRAM Executive Summary

- Large storage arrays cannot be implemented “digitally”
  - Muxing and wire routing become impractical
- SRAM implementation exploits analog transistor properties
  - Inverter pair bits much smaller than flip-flop bits
  - Wordline/bitline arrangement makes for simple “grid-like” routing
  - Basic understanding of reading and writing
    - Wordlines select words (which address)
    - To write, we *overwhelm* the inverter-pair
  - Access latency proportional to √#bits * #ports

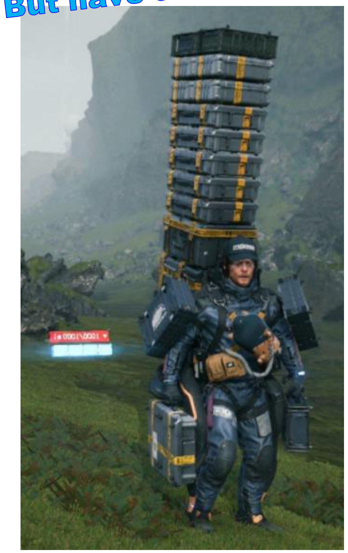
Dynamic RAM (DRAM)



- **DRAM**: dynamic RAM
  - Bits as capacitors (if charge, bit=1)
  - "Pass transistors" as ports
  - One transistor per bit/port

- **"Dynamic"** means
  - Capacitors not connected to power/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed

- Designed for density


Gotta go...kinda fast?
But have a lot of stuff.

Moore's Law (DRAM chip capacity)

Year	Capacity	$/MB	Access time
1980	64Kb	$1500	250ns
1988	4Mb	$50	120ns
1996	64Mb	$10	60ns
2004	1Gb	$0.5	35ns
2008	2Gb	~$0.15	20ns
2013	8Gb	~$0	<10ns

- Commodity DRAM parameters
    - 16X increase in capacity every 8 years = 2X every 2 years
        - Not quite 2X every 18 months (Moore's Law) but still close

Access Time and Cycle Time

- DRAM access much slower than SRAM
  - More bits → longer wires
  - SRAM access latency: 2–3ns
  - DRAM access latency: 20-35ns

- DRAM cycle time also longer than access time
  - **Cycle time**: time between start of consecutive accesses
  - SRAM: cycle time = access time
    - Begin second access as soon as first access finishes
  - DRAM: cycle time = 2 * access time
    - Why? Can't begin new access while DRAM is refreshing row

How do we use SRAM and DRAM?

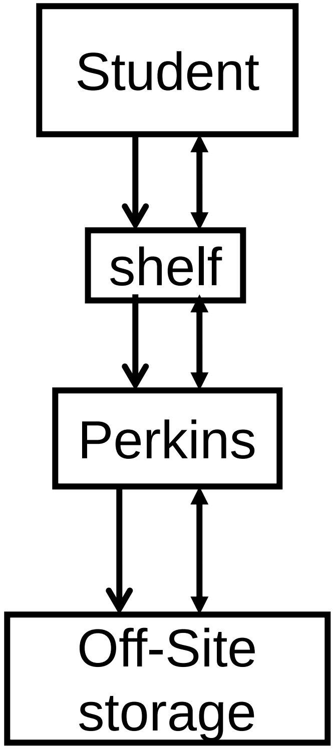
- Making a little embedded chip with 8kB RAM?
  - Use SRAM. It’s fine.
- Making a laptop and need 8**G**B RAM?
  - Can’t use SRAM – not practical ☹
  - Must use DRAM!
    - But it’s big, so it’s slow!
    - And it’s DRAM, so it’s even slower!
  - CPU might otherwise be able to do 100+ instructions in time it takes for ONE read from DRAM!

Introducing caching

Motivation

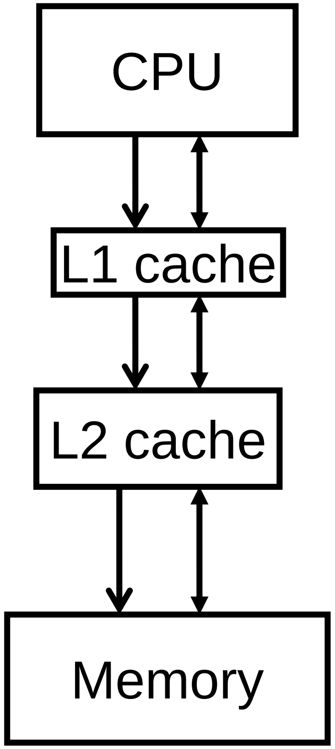
- Problem: Large memory must be made of DRAM;
  DRAM is too dang slow
- Thing we have access to: some SRAM, which is fast, but small
- **Can we duct tape the SRAM on top of the DRAM and get the best of both worlds??????**
  - Answer: yes, but it’s a little complicated.
- We’ll start with an analogy…

An Analogy: Duke's Library System



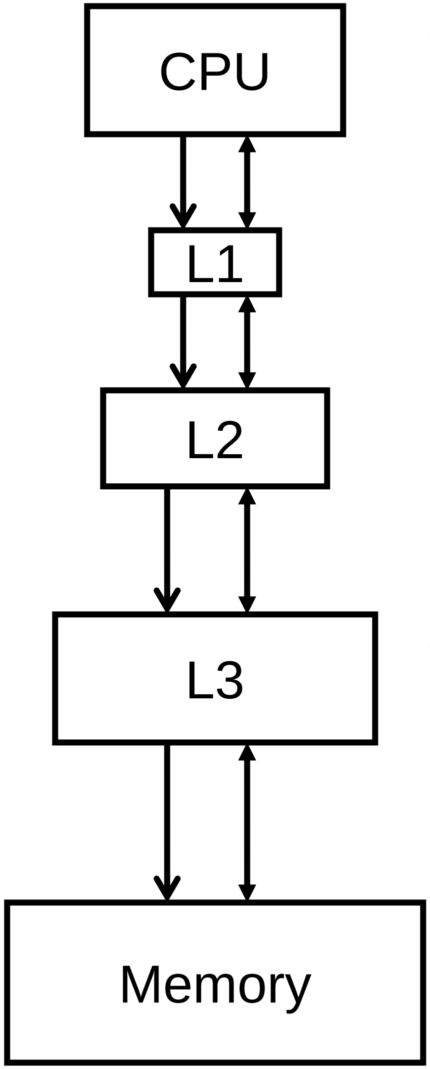
- Student keeps small subset of Duke library books on bookshelf at home
  - Books she's actively reading/using
  - Small subset of all books owned by Duke
  - Fast access time
- If book not on her shelf, she goes to Perkins
  - Much larger subset of all books owned by Duke
  - Takes longer to get books from Perkins
- If book not at Perkins, must get from off-site storage
  - Guaranteed (in my analogy) to get book at this point
  - Takes much longer to get books from here

An Analogy: Duke's Library System



- CPU keeps small subset of memory in its level-1 (L1) cache
  - Data it's actively reading/using
  - Small subset of all data in memory
  - Fast access time
- If data not in CPU's cache, CPU goes to level-2 (L2) cache
  - Much larger subset of all data in memory
  - Takes longer to get data from L2 cache
- If data not in L2 cache, must get from main memory
  - Guaranteed to get data at this point
  - Takes much longer to get data from here

Big Concept: Memory Hierarchy



- Use hierarchy of memory components
  - Upper components (closer to CPU)
    - Fast ↔ Small ↔ Expensive
  - Lower components (further from CPU)
    - Slow ↔ Big ↔ Cheap
  - Bottom component (for now!) = what we have been calling "memory" until now

- Make average access time close to L1's
  - How?
  - Most frequently accessed data in L1
  - L1 + next most frequently accessed in L2, etc.
  - **Automatically** move data up & down hierarchy

Some Terminology

- If we access a level of memory and find what we want → called a hit
- If we access a level of memory and do NOT find what we want → called a miss

Some Goals

- Key 1: High “hit rate” → high probability of finding what we want at a given level
- Key 2: Low access latency

- Misses are expensive (take a long time)
  - Try to avoid them
  - But, if they happen, amortize their costs → bring in more than just the specific word you want → bring in a whole block of data (multiple words)

Blocks

- Block = a group of spatially contiguous and aligned bytes
  - Typical sizes are 32B, 64B, 128B
- Spatially contiguous and aligned
  - Example: 32B blocks
  - Blocks = [address 0- address 31], [32-63], [64-95], etc.
  - NOT:
    - [13-44] = unaligned
    - [0-22, 26-34] = not contiguous
    - [0-20] = wrong size (not 32B)

Why Hierarchy Works For Duke Books

- **Temporal locality**
  - Recently accessed book likely to be accessed again soon

- **Spatial locality**
  - Books near recently accessed book likely to be accessed soon (assuming spatially nearby books are on same topic)

Why Hierarchy Works for Memory

- **Temporal locality**
  - Recently executed instructions likely to be executed again soon
    - Loops
  - Recently referenced data likely to be referenced again soon
    - Data in loops, hot global data

- **Spatial locality**
  - Insns near recently executed insns likely to be executed soon
    - Sequential execution
  - Data near recently referenced data likely to be referenced soon
    - Elements in array, fields in struct, variables in stack frame

- **Locality** is one of the most important concepts in computer architecture → don't forget it!

Hierarchy Leverages Non-Uniform Patterns

- **10/90 rule (of thumb)**
  - For Instruction Memory:
    - 10% of static insns account for 90% of executed insns
    - Inner loops
  - For Data Memory:
    - 10% of variables account for 90% of accesses
    - Frequently used globals, inner loop stack variables

- What if processor accessed every block with equal likelihood? Small caches wouldn’t help much.

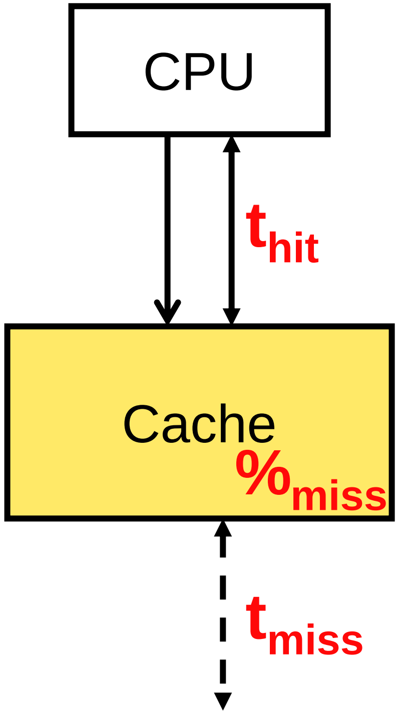
Memory Hierarchy: All About Performance

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

- t_{avg} = average time to satisfy request at given level of hierarchy
- t_{hit} = time to hit (or discover miss) at given level
- t_{miss} = time to satisfy miss at given level
- Problem: hard to get low t_{hit} and $\%_{miss}$ in one structure
  - Large structures have low $\%_{miss}$ but high t_{hit}
  - Small structures have low t_{hit} but high $\%_{miss}$
- Solution: use a **hierarchy** of memory structures

> "Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."
>
> Burks, Goldstine, and Von Neumann, 1946

Memory Performance Equation



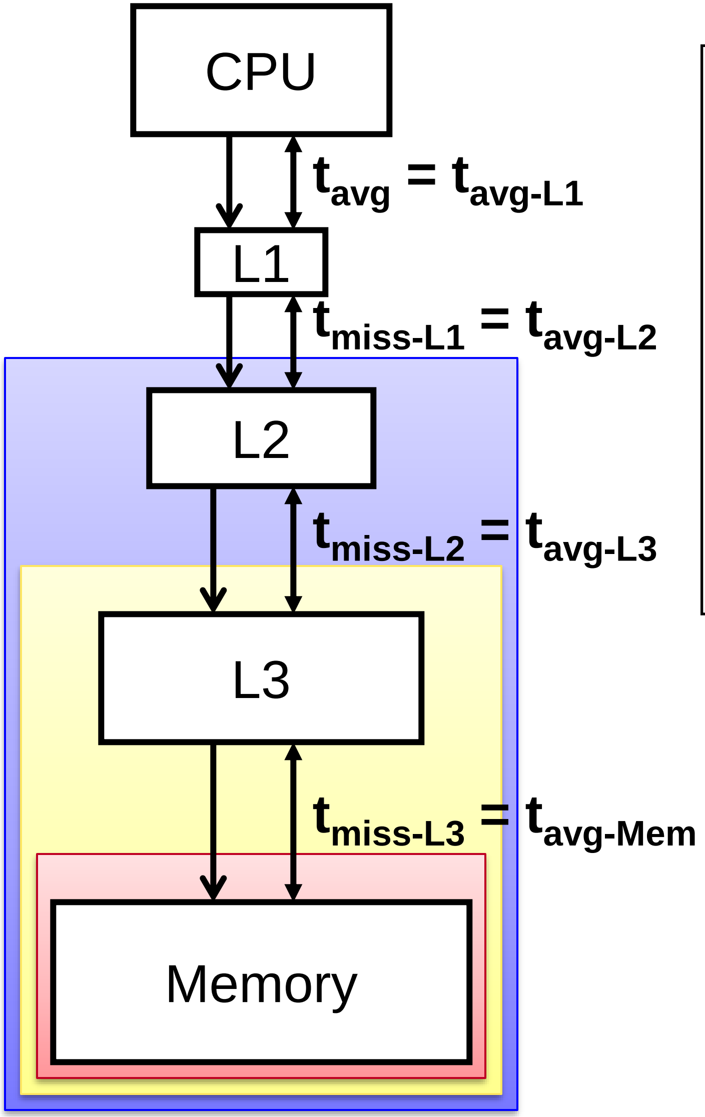
- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) M
  - t_{miss}: time to read data into M from lower level

- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Abstract Hierarchy Performance


CPU
$t_{avg} = t_{avg\text{-}L1}$
L1
$t_{miss\text{-}L1} = t_{avg\text{-}L2}$
L2
$t_{miss\text{-}L2} = t_{avg\text{-}L3}$
L3
$t_{miss\text{-}L3} = t_{avg\text{-}Mem}$
Memory

How do we compute t_{avg} ?

$=t_{avg\text{-}L1}$

$=t_{hit\text{-}L1} + (\%_{miss\text{-}L1} * t_{miss\text{-}L1})$

$=t_{hit\text{-}L1} + (\%_{miss\text{-}L1} * t_{avg\text{-}L2})$

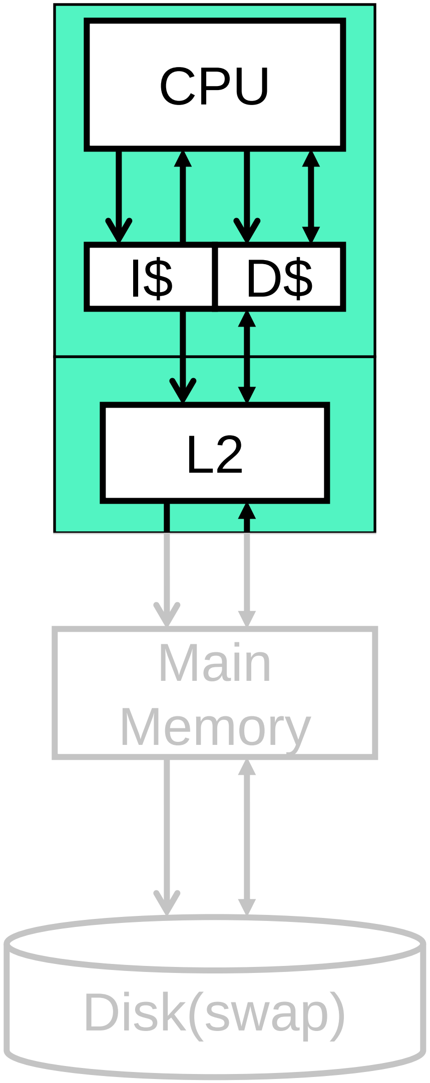
$=t_{hit\text{-}L1} + (\%_{miss\text{-}L1} * (t_{hit\text{-}L2} + (\%_{miss\text{-}L2} * t_{miss\text{-}L2})))$

$=t_{hit\text{-}L1} + (\%_{miss\text{-}L1} * (t_{hit\text{-}L2} + (\%_{miss\text{-}L2} * t_{avg\text{-}L3})))$

= …

Note: Miss at level n = access at level n+1

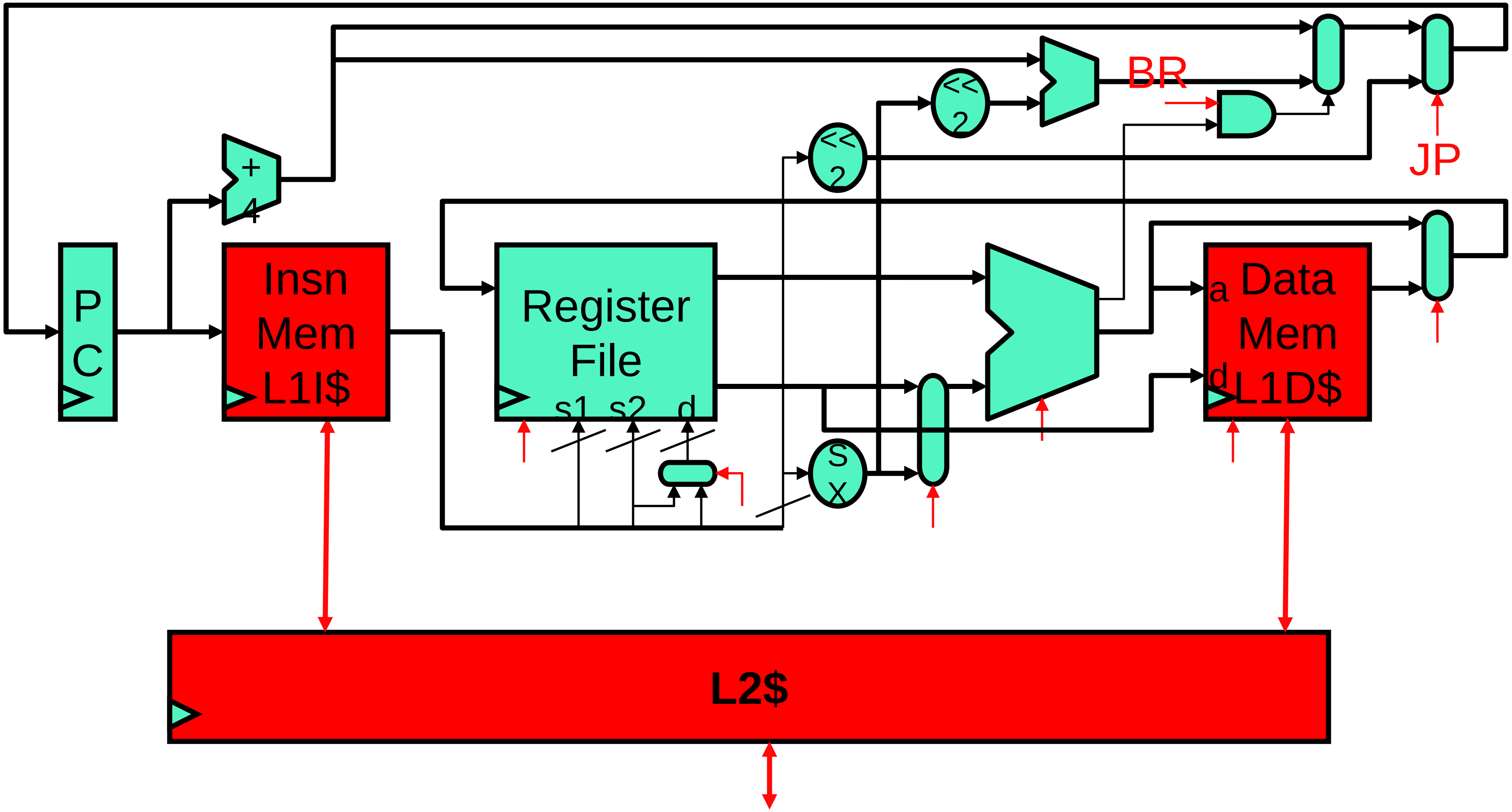
Typical Memory Hierarchy



- 1st level: **L1 I$**, **L1 D$** (L1 insn/data caches)
- 2nd level: **L2 cache (L2$)**
  - Also on same chip with CPU
  - Made of SRAM (same circuit type as CPU)
  - Managed in hardware
  - This unit of ECE/CS 250
- 3rd level: **main memory**
  - Made of DRAM
  - Managed in software
  - Next unit of ECE/CS 250
- 4th level: **disk (swap space)**
  - Made of magnetic iron oxide discs
  - Managed in software
  - Course unit after main memory
- Could be other levels (e.g., Flash, PCM, tape, etc.)

Note: many processors have L3$ between L2$ and memory

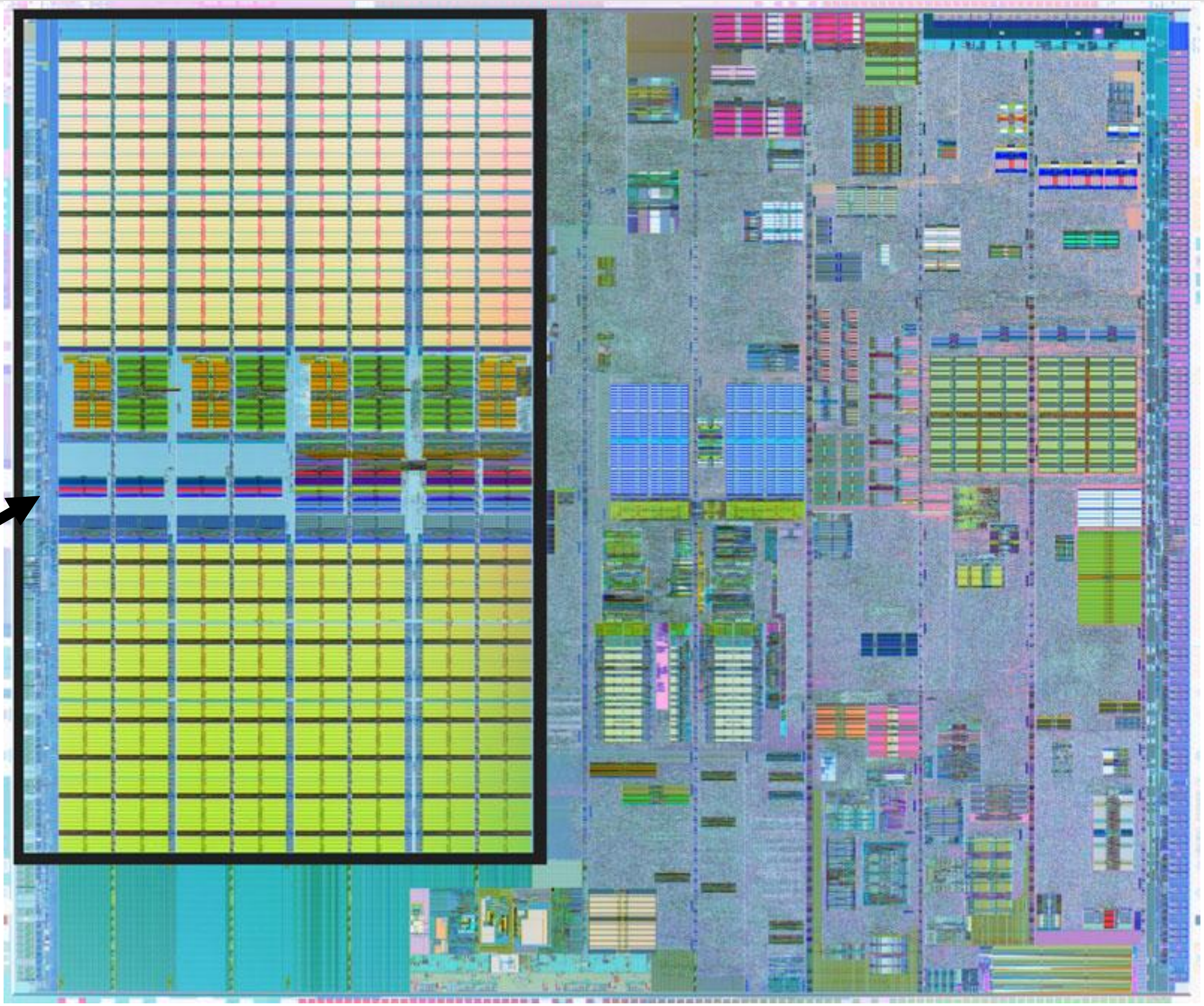
Concrete Memory Hierarchy



- Much of today’s chips used for caches → important!

A Typical Die Photo

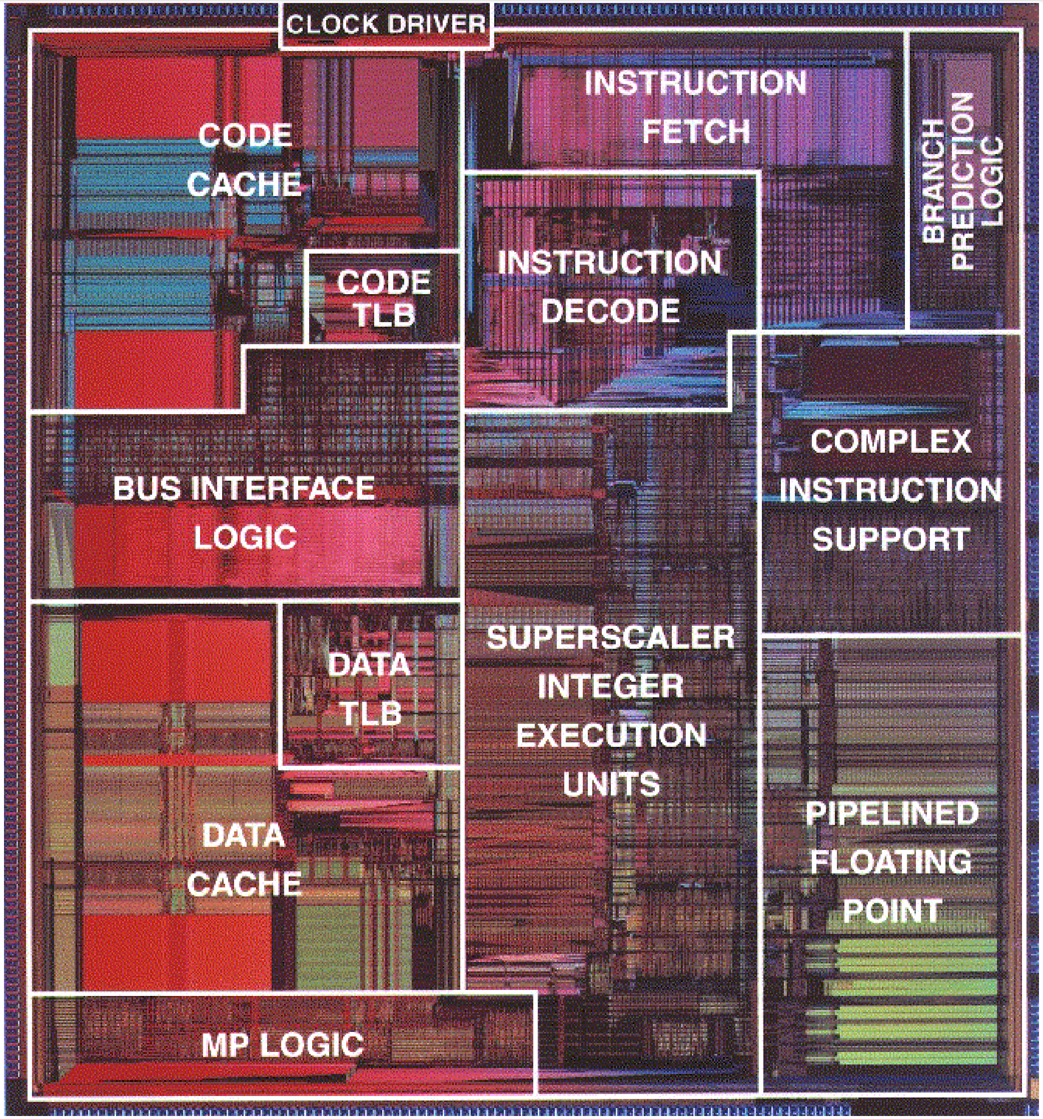
Intel Pentium4 Prescott chip with 2MB L2$



L2 Cache

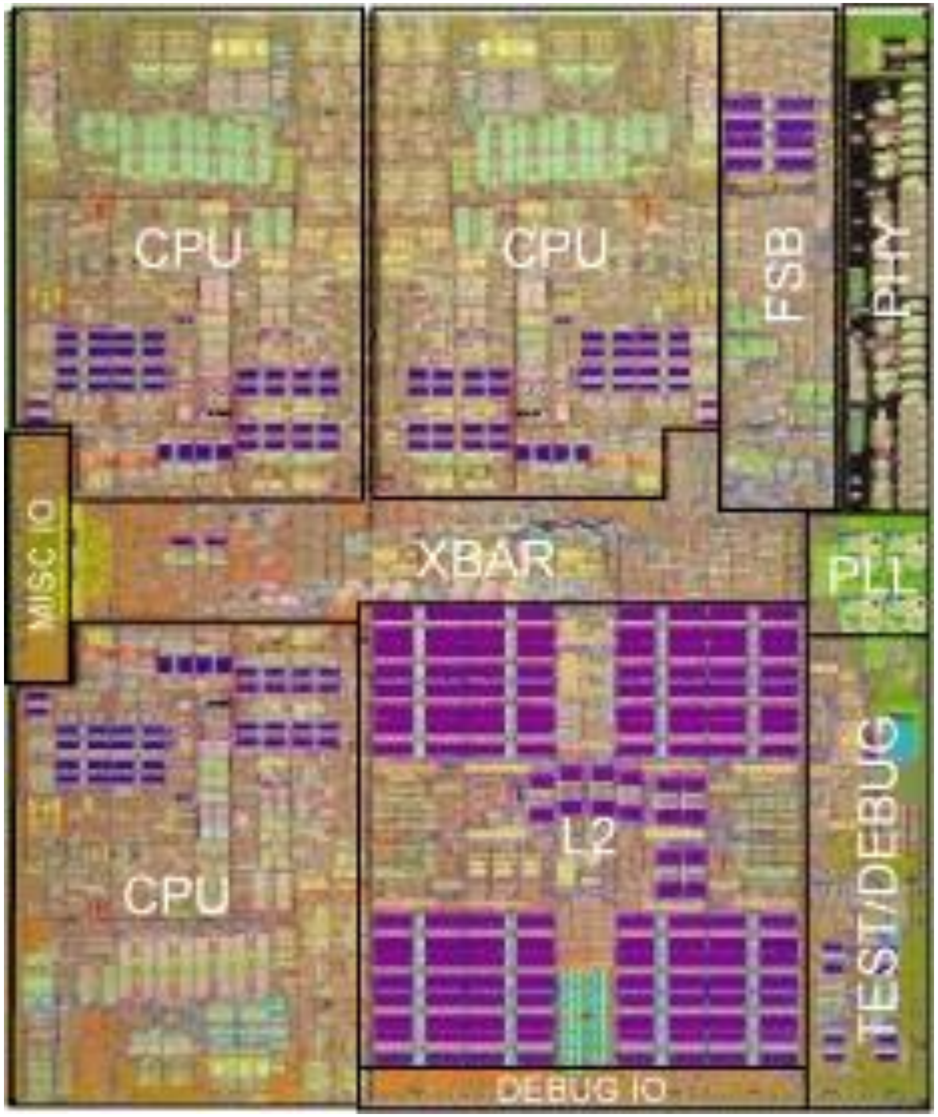
A Closer Look at that Die Photo

Intel Pentium chip with 2x16kB split L1$

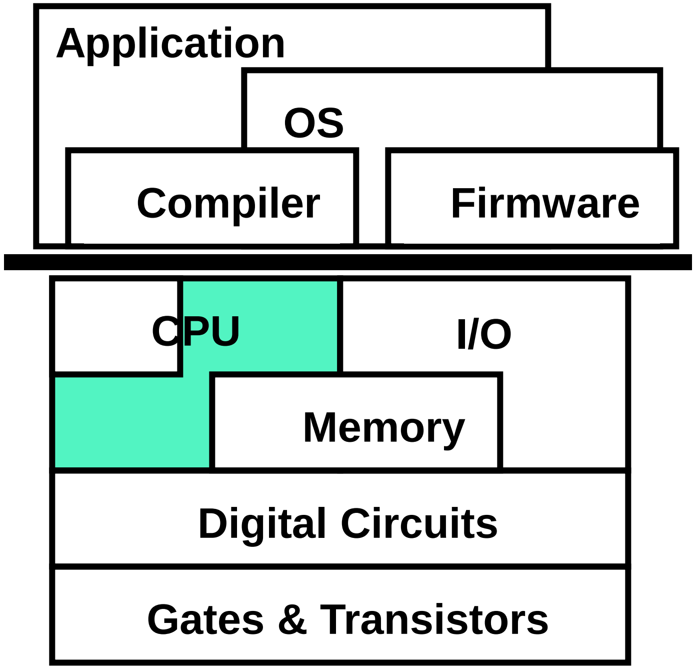


A Multicore Die Photo from IBM

IBM’s Xenon chip with 3 PowerPC cores



This Unit: Caches and Memory Hierarchies



- Memory hierarchy
- Cache organization
- Cache implementation

Back to Our Library Analogy

- This is a base-10 (not base-2) analogy
- Assumptions
  - 1,000,000 books (blocks) in library (memory)
  - Each book has 10 chapters (bytes)
  - Every chapter of every book has its own unique number (address)
    - E.g., chapter 3 of book 2 has number 23
    - E.g., chapter 8 of book 110 has number 1108
  - My bookshelf (cache) has room for 10 books
    - Call each place for a book a "frame"
    - The number of frames is the "capacity" of the shelf
  - I make requests (loads, fetches) for 1 or more chapters at a time
    - But everything else is done at book granularity (not chapter)

Organizing My Bookshelf (cache!)

- Two extreme organizations of flexibility (associativity)
    - Most flexible: any book can go anywhere (i.e., in any frame)
    - Least flexible: a given book can only go in one frame
- In between the extremes
    - A given book can only go in a subset of frames (e.g., 1 or 10)
- If not most flexible, how to map book to frame?

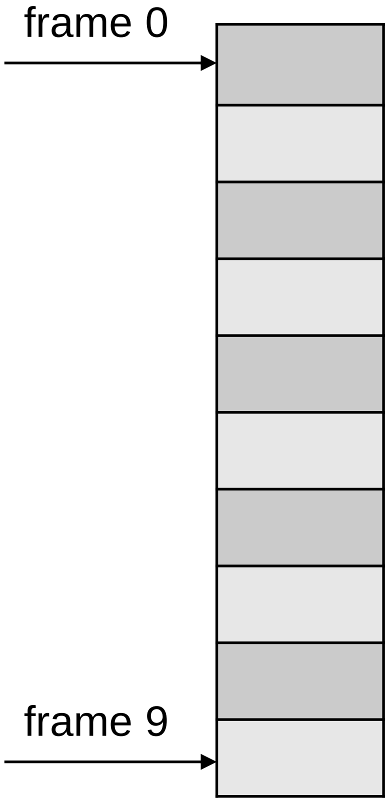
Least Flexible Organization: Direct-mapped

- Least flexible (direct-mapped)
- Book X maps to frame X mod 10
  - Book 0 in frame 0
  - Book 1 in frame 1
  - Book 9 in frame 9
  - Book 10 in frame 0
  - Etc.
- What happens if you want to keep book 3 and book 23 on shelf at same time? You can't! Have to replace (evict) one to make room for other.

This spot reserved for a book ending in '0' (0, 10, 20, etc.)

This spot reserved for a book ending in '1' (1, 11, 21, etc.)

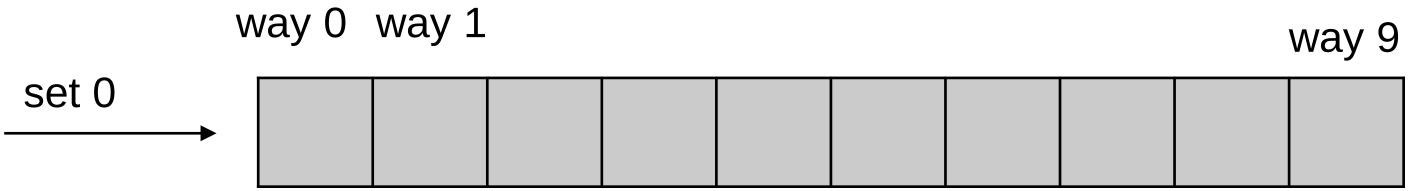
This spot reserved for a book ending in '9' (9, 19, 29, etc.)



Most Flexible Organization: Fully Associative

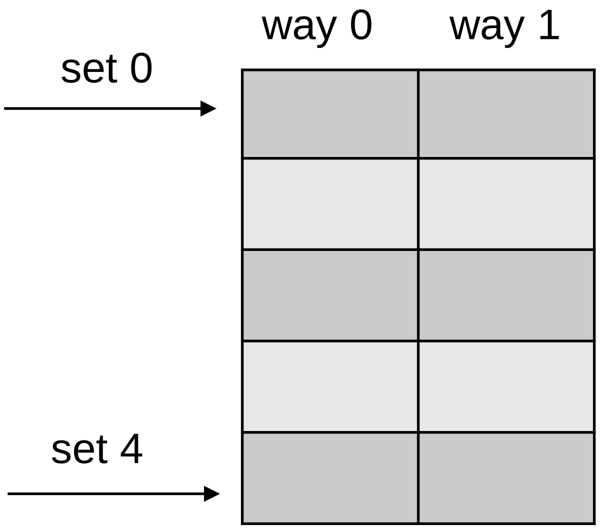
- Keep same shelf capacity (10 frames)
- Allow a book to be in any frame
  - fully-associative
- Whole shelf is one set
  - Ten ways in this set
  - Book could be in any way of set
- All books map to set 0 (only 1 set!)

You can put any book in any of these ten spots. Go nuts.



In-between Flexibility (Associativity)

- Keep same shelf capacity (10 frames)
- Now allow a book to map to multiple frames
- Frames now grouped into sets
  - If 2 frames/set, 2-way set-associative
- 1-to-1 mapping of book to set
  - 1-to-many mapping of book to frame
- If 5 sets, book X maps to set X mod 5
  - Book 0 in set 0
  - Book 1 in set 1
  - Book 4 in set 4
  - Book 5 in set 0
  - Etc.



These two spots reserved for books ending in ‘0’ or ‘5’ (0, 5, 10, 15, etc.)

These two spots reserved for books ending in ‘1’ or ‘6’ (1, 6, 11, 16, etc.)

These two spots reserved for books ending in ‘4’ or ‘9’ (4, 9, 14, 19, etc.)

Reminder about book/chapter numbers

- Remember how we're numbering our books and chapters:

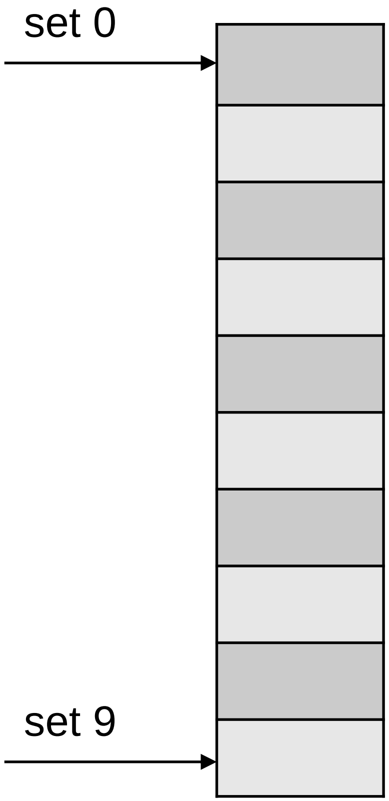
$$\underbrace{1362}_{\text{Book number}}\underbrace{5}_{\text{Chapter number}}$$

- If we're talking about a whole book (block), discard the chapter number:

1362 <- Book number

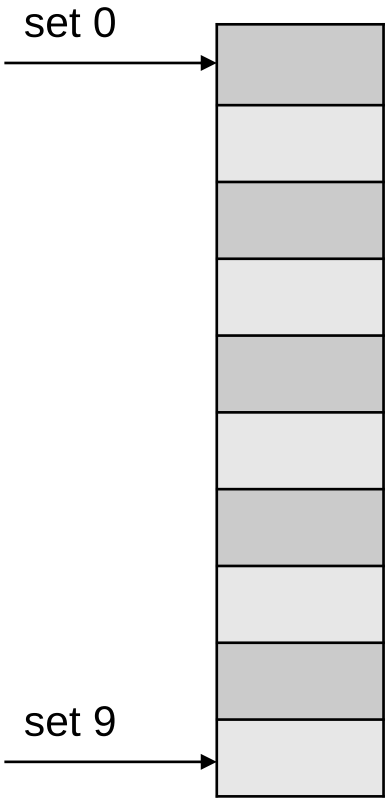
Tagging Books on Shelf

- Let’s go back to direct-mapped organization (w/10 sets)
- How do we find if book is on shelf?
- Consider book 1362
  - At library, just go to location 1362 and it’s there
  - But shelf doesn’t have 1362 locations
- OK, so go to set 1362%10=2
- If book is on shelf, it’s there
- But same is true for other books!
  - Books 2, 12, 22, 32, etc.
- How do we know which one is there?
- Must tag each book to distinguish it



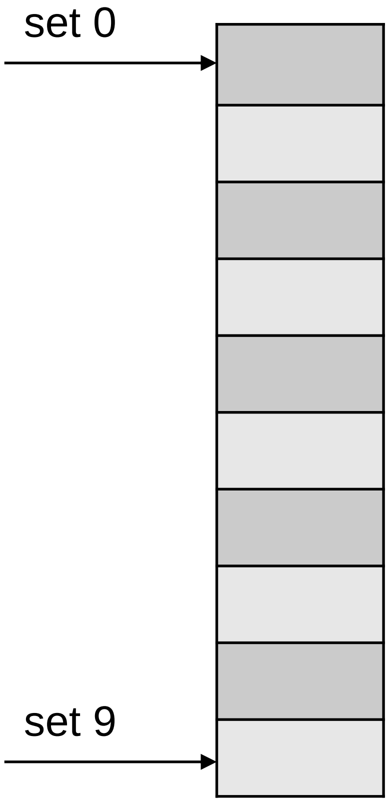
How to Tag Books on Shelf

- Still assuming direct-mapped shelf
- How to tag book 1362?
  - Must distinguish it from other books that could be in same set
- Other books that map to same set (2)?
  - 2, 12, 22, 32, … 112, 122, … 2002, etc.
- Could tag with entire book number
  - But that’s overkill – we already know last digit
- Tag for 1362 = 136



How to Find Book on Shelf

- Consider direct-mapped shelf
- How to find if book 1362 is on shelf?
- Step 1: go to right set (set 2)
- Step 2: check every frame in set
  - If tag of book in frame matches tag of requested book, then it's a match (hit)
  - Else, it's a miss



Revisiting book/chapter numbers

- Remember how we're numbering our books and chapters:

$$\underbrace{1362}_{\text{Book number}}\underbrace{5}_{\text{Chapter number}}$$

- If we're talking about a whole book (block), discard the chapter number:

  1362 <- Book number

- Now we're shaving off a digit to distinguish:
  - **Index** (which set?)

    vs.
  - **Tag** (of all the books that could go in this set, which is this one?)
- Putting it together:

$$\underbrace{136}_{\text{Tag}}\underbrace{2}_{\text{Index}}\underbrace{5}_{\text{Chapter number}}$$

From Library/Book Analogy to Computer

- If you understand this library/book analogy, then you’re ready for computer caches
- Everything is similar in computer caches, but remember that computers use base-2 (not base-10)

Cache Structure

- A cache (shelf) consists of frames, and each frame is the storage to hold one block of data (book)
  - Also holds a “valid” bit and a “tag” to label the block in that frame
- Valid: if 1, frame holds valid data; if 0, data is invalid
  - Useful? Yes. Example: when you turn on computer, cache is full of invalid “data” (better examples later in course)
- Tag: specifies which block is living in this frame
  - Useful? Yes. Far fewer frames than blocks of memory!

valid	“tag”	block data
1	[64-95]	32 bytes of valid data
0	[0-31]	32 bytes of junk
1	[0-31]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data

Cache Structure

I write “tag” in quotes because I’m not using a proper tag, as we’ll see later. I’m using “tag” now to label the block. For example, a “tag” of [64-95] denotes that the block in this frame is the block that goes from address 64 to address 95. This “tag” uniquely identifies the block, which is its purpose, but it’s overkill as we’ll see later.

valid	“tag”	block data
1	[64-95]	32 bytes of valid data
0	[0-31]	32 bytes of junk
1	[0-31]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data

Cache Example (very simplified for now)

valid	"tag"	block data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- When computer turned on, no valid data in cache (everything is zero, including valid bits)

Cache Example (very simplified for now)

valid	"tag"	block data
1	[32-63]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks for word (book chapters) at byte addresses [32-35]
  - Either due to a load or an instruction fetch
- Word [32-35] is part of block [32-63]
- Miss! No blocks in cache yet
- Fill cache (from lower level) with block [32-63]
  - don't forget to set valid bit and write tag

Cache Example (very simplified for now)

valid	“tag”	block data
1	[32-63]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks for word [1028-1031]
  - Either due to a load or an instruction fetch
- Word [1028-1031] is part of block [1024-1055]
- Miss!
- Fill cache (from lower level) with block [1024-1055]

Cache Example (very simplified for now)

valid	"tag"	block data
1	[32-63]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks (again!) for word [1028-1031]
  - Hit! Hooray for temporal locality
- Assume CPU asks for word [1032-1035]
  - Hit! Hooray for spatial locality
- Assume CPU asks for word [0-3]
  - Miss! Don't forget those valid bits.

Where to Put Blocks in Cache

- How to decide which frame holds which block?
  - And then how to find block we're looking for?
- Some more cache structure:
  - Divide cache into sets
    - A block can only go in its set
  - Each set holds some number of frames = set associativity
    - E.g., 4 frames per set = 4-way set-associative
- The two extremes of set-associativity
  - Whole cache has just one set = fully associative
    - Most flexible (longest access latency)
  - Each set has 1 frame = 1-way set-associative = "direct mapped"
    - Least flexible (shortest access latency)

Interlude: How does time work?

Except we use powers of 2!

- To help understand how we're going to break down addresses, how do we break down time?
- It is 40,000 seconds since midnight. What time is it?
  - Divide and mod by 60 (because 60 min/sec):
    - 40000 / 60 = 666 minutes since midnight plus…
    - 40000 % 60 = 40 seconds
    - Can write as "666 minutes : 40 seconds"
  - Wait, what about hours? Divide and mod by 60 (because 60 hour/min)
    - 666 / 60 = 11 hours since midnight plus…
    - 666 % 60 = 6 minutes
    - Can write as "11 hours : 06 minutes"
    - Include seconds from before: "11:06:40" is the time!
  - Can go back to seconds too: 11*60*60 + 6*60 + 40 = 40000
  - "11:06:40" and "40000" are two ways of writing the same number!

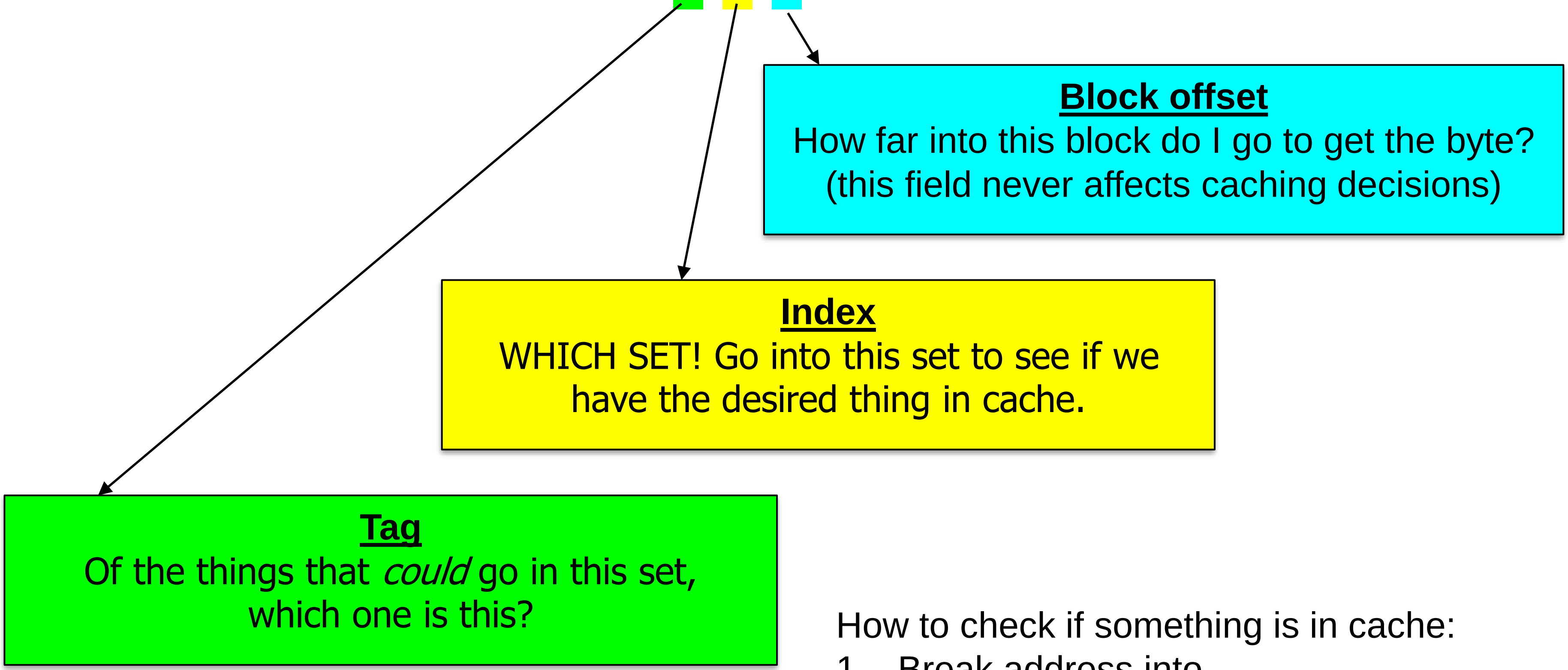
Breaking down an address in the same way

- My cache has 8-byte blocks;
  my cache has 4 sets.

- It is 51 bytes into memory (address = 51). Break it down.
  - Divide and mod by 8 (because 8 bytes/block):
    - 51 / 8 = 6 is the block number, plus…
    - 51 % 8 = 3 block offset
    - Can write as "6 blocknum : 3 block offset"
  - Wait, what about tag and index? Divide and mod by 4 (because 4 sets)
    - 6 / 4 = 1 is the tag, and the index is…
    - 6 % 4 = 2 is the index (i.e., we look in set 2)
    - Can write as "1 tag : 2 index"
    - Include block offset from before: address broken into "1:2:3"!
  - Can go back to address too: 1*4*8 + 2*8 + 3 = 51
  - "1:2:3" and "51" are two ways of writing the same number!

What do the fields mean?

- Address 51 is broken into “1:2:3”



Block offset
How far into this block do I go to get the byte?
(this field never affects caching decisions)

Index
WHICH SET! Go into this set to see if we have the desired thing in cache.

Tag
Of the things that *could* go in this set, which one is this?

How to check if something is in cache:

1. Break address into
   tag : ***index*** : ***offset***
2. Go to set number ***index***
3. See if it has ***tag***

Mod vs the bits

Divide and modulo by powers of two is like splitting up bit fields!

Example:
11101 (29 decimal)

Want to split last 3 bits?

```
11101
 ^  ^
 3  5
```

By div and mod:
$29 / 2^3 = 29/8 = 3$
$29 \% 2^3 = 29\%8 = 5$

While learning, we'll show div and mod. Then we'll switch to bits!

Base 10		
num	num/8	num%8
0	0	0
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5
6	0	6
7	0	7
8	1	0
9	1	1
10	1	2
11	1	3
12	1	4
13	1	5
14	1	6
15	1	7
16	2	0
17	2	1
18	2	2
19	2	3
20	2	4
21	2	5
22	2	6
23	2	7
24	3	0
25	3	1
26	3	2
27	3	3
28	3	4
29	3	5
30	3	6
31	3	7
32	4	0
33	4	1
34	4	2
35	4	3

Base 2				
num	num/8	num>>3	num&7	num%8
0	0	0	0	0
1	0	0	1	1
10	0	0	10	10
11	0	0	11	11
100	0	0	100	100
101	0	0	101	101
110	0	0	110	110
111	0	0	111	111
1000	1	1	0	0
1001	1	1	1	1
1010	1	1	10	10
1011	1	1	11	11
1100	1	1	100	100
1101	1	1	101	101
1110	1	1	110	110
1111	1	1	111	111
10000	10	10	0	0
10001	10	10	1	1
10010	10	10	10	10
10011	10	10	11	11
10100	10	10	100	100
10101	10	10	101	101
10110	10	10	110	110
10111	10	10	111	111
11000	11	11	0	0
11001	11	11	1	1
11010	11	11	10	10
11011	11	11	11	11
11100	11	11	100	100
11101	11	11	101	101
11110	11	11	110	110
11111	11	11	111	111
100000	100	100	0	0
100001	100	100	1	1
100010	100	100	10	10
100011	100	100	11	11

Direct-Mapped (1-way) Cache

- Assume 8B blocks
- 4 sets, 1 way/set → 4 frames
- Each block can only be put into 1 set (1 option)
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 2
  - Block [24-31] → set 3
  - Block [32-39] → set 0
  - Block [40-47] → set 1
  - Block [48-55] → set 2
  - Block [56-63] → set 3
  - …
- Block [X-(X+7)] → set $\underbrace{\overbrace{(X/8)}^{\text{“Block number”}}\%4}_{\text{“Index”}}$

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			

Direct-Mapped (1-way) Cache

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 11, 5, 50, 67, 51, 3
- Which hit? Which miss?
- First find out where they live:

Blk#=(Addr/Blocksize)

Set=Blk# % NumSets

Address	Blk#	Index
2		
11		
5		
50		
67		
51		
3		
512		
1024		
2		

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			

Problem with Direct Mapped Caches

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 35, 2, 35, 2, 35, 2, 35, …
- Which hit? Which miss?
- Did we make good use of all of our cache capacity?

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			

Address	Index
2	
35	

2-Way Set-Associativity

	way 0			way 1		
	valid	tag	data	valid	tag	data
set 0						
set 1						

- 2 sets, 2 ways/set → 4 frames (just like our 1-way cache)
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 0
  - Block [24-31] → set 1
  - Block [32-39] → set 0
  - Etc.

2-Way Set-Associativity

	way 0			way 1		
	valid	tag	data	valid	tag	data
set 0						
set 1						

- Assume the same pathological stream of CPU requests:
  - Byte addresses 2, 35, 2, 35, 2, 35, etc.
  - Which hit? Which miss?
- Now how about this: 2, 35, 65, 2, 35, 67, etc.
- How much more associativity can we have?

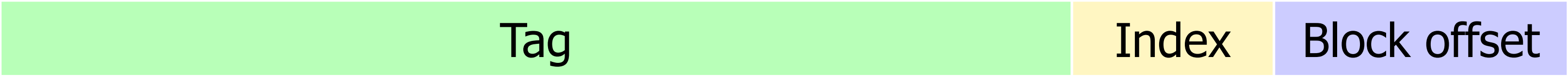
Full Associativity

	way 0			way 1			way 2			way 3		
	v	t	d	v	t	d	v	t	d	v	t	d
set 0												

- 1 set, 4 ways/set → 4 frames (just like previous examples)
  - Block [0-7] → set 0
  - Block [8-15] → set 0
  - Block [16-23] → set 0
  - Etc.

Mapping Addresses to Sets

- MIPS has 32-bit addresses
  - Let’s break down address into three components
- If blocks are 8B, then $\log_2 8=3$ bits required to identify a byte within a block. These bits are called block offset.
  - Given block, offset (book chapter) tells you which byte within block
- If there are S sets, then $\log_2 S$ bits required to identify the set. These bits are called set index or just index.
- Rest of the bits ($32 - 3 - \log_2 S$) specify the tag



Tag

Index

Block offset

Mapping Addresses to Sets

- How many blocks map to the same set?
- Let’s assume 8-byte blocks
  - $8=2^3$ → 3 bits to specify block offset
- Let’s assume we have direct-mapped cache with 256 sets
  - 256 sets $=2^8$ sets → 8 bits to specify set index
- 2^{32} bytes of memory/(8 bytes/block) = 2^{29} blocks
- 2^{29} blocks / 256 sets = 2^{21} blocks / set
- So that means we need 2^{21} tags to distinguish between all possible blocks in the set → 21 tag bits
  - Note: 21=32-3-8 ☺

Tag (21 bits)	Index (8 bits)	Block offset (3 bits)

Mapping Addresses to Sets

Tag (21 bits)	Index (8 bits)	Block offset (3 bits)

- Assume cache from previous slide (8B blocks, 256 sets)
- Example: What do we do with the address 58?

  0000 0000 0000 0000 0000 0000 0011 1010

  - offset = 2 (2nd byte in block)
  - index=7 (set 7)
  - tag = 0
- This matches what we did before – recall:
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 2
  - etc.

Cache Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - **Random**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier-to-implement approximation of LRU
    - NMRU=LRU for 2-way set-associative caches

ABCs of Cache Design

- Architects control three primary aspects of cache design
  - And can choose for each cache independently
- A = Associativity
- B = Block size
- C = Capacity of cache

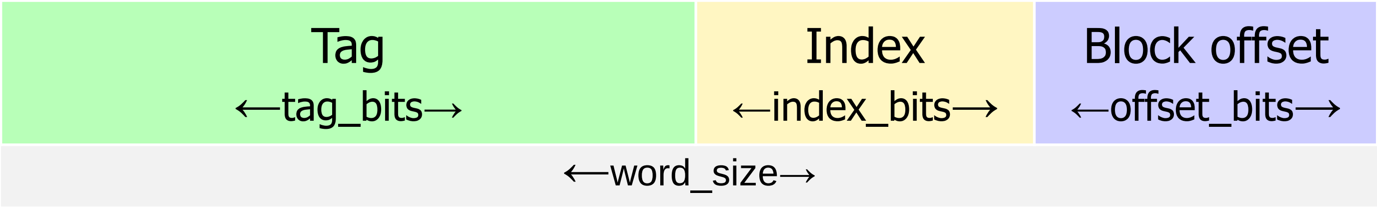
- Secondary aspects of cache design
  - Replacement algorithm
  - Some other more subtle issues we'll discuss later

Cache structure math: cache design

- Given **a**ssociativity (ways), **b**lock_size, **c**apacity, and word_size.

- Cache parameters:
  - num_frames = capacity / block_size
  - sets = num_frames / ways = capacity / block_size / ways

- Address bit fields:
  - offset_bits = $\log_2$(block_size)
  - index_bits = $\log_2$(sets)
  - tag_bits = word_size - index_bits - offset_bits



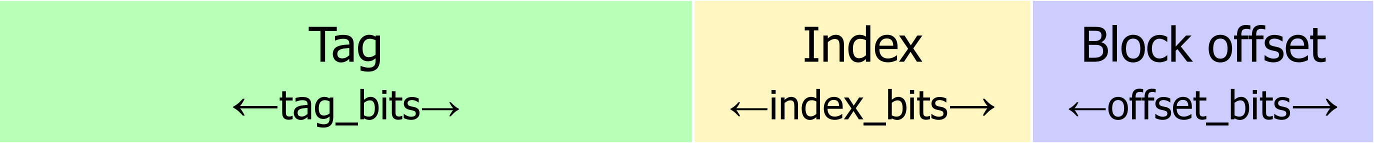
Tag
←tag_bits→

Index
←index_bits→

Block offset
←offset_bits→

←word_size→

Cache structure math: address decomposition



Tag	Index	Block offset
←tag_bits→	←index_bits→	←offset_bits→

- Way to get offset/index/tag from address (bitwise & numeric):
  - block_offset = addr & ones(offset_bits)
    = addr % block_size
  - index = (addr >> offset_bits) & ones(index_bits)
    = (addr / block_size) % sets
  - tag = addr >> (offset_bits+index_bits)
    = addr / (sets*block_size)

ones(n) = a string of *n* ones = ((1<<n)-1)

Cache structure math: example

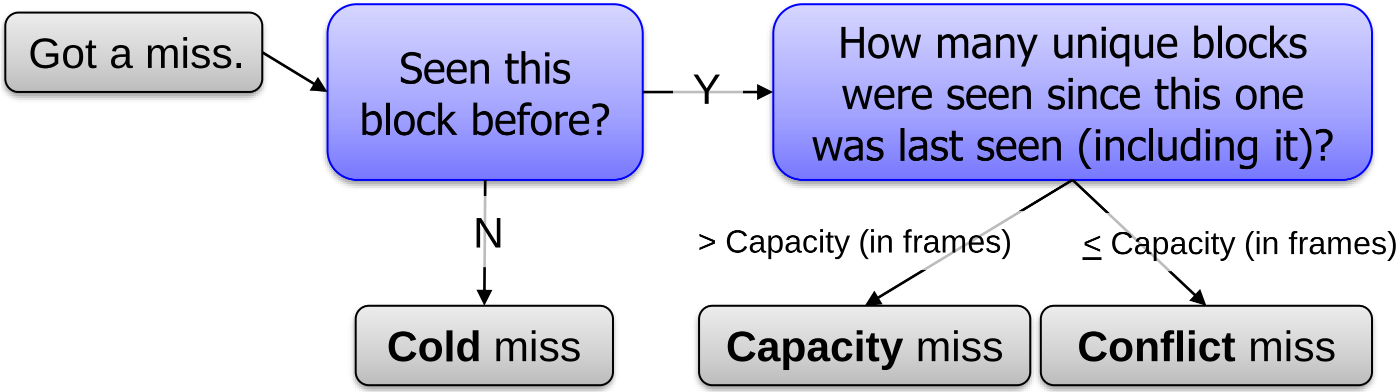
- Example: a 16-bit computer with a 1kB 4-way cache, block size 16

	value	units	eqn
wordsize	16	bits	*given*
associativity (ways)	4	ways	*given*
block size	16	bytes	*given*
capacity	1024	bytes	*given*
num frames	64	frames	capacity / block_size
sets	16	sets	num_frames / ways
offest bits	4	bits	lg(block_size)
index bits	4	bits	lg(sets)
tag bits	8	bits	wordsize-index_bits-offset_bits

Decimal				0	**Hex**			
	*addr/(sets*block_size)*	*(addr/block_size)%sets*	*addr%block_size*			*addr[15:8]*	*addr[7:4]*	*addr[3:0]*
addr	**tag**	**index**	**block_offset**		**addr**	**tag**	**index**	**block_offset**
0	0	0	0		0000	0	0	0
1	0	0	1		0001	0	0	1
2	0	0	2		0002	0	0	2
16	0	1	0		0010	0	1	0
32	0	2	0		0020	0	2	0
48	0	3	0		0030	0	3	0
256	1	0	0		0100	1	0	0
512	2	0	0		0200	2	0	0
768	3	0	0		0300	3	0	0

Analyzing Cache Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - Easy to identify
  - **Capacity**: miss caused because cache is too small – would've been miss even if cache had been fully associative
    - Consecutive accesses to block separated by accesses to at least N other distinct blocks where N is number of frames in cache
  - **Conflict**: miss caused because cache associativity is too low – would've been hit if cache had been fully associative
    - All other misses



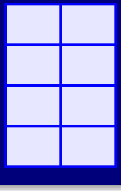
3C Example

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 11, 5, 50, 67, 128, 256, 512, 1024, 2
  - Is the last access a capacity miss or a conflict miss?

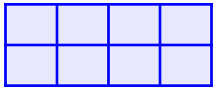
Location	Set
2	0
11	1
5	0
50	6
67	0
128	0
256	0
512	0
1024	0
2	0

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			
set 4			
set 5			
set 6			
set 7			

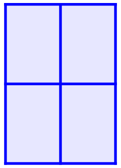
ABCs of Cache Design and 3C Model



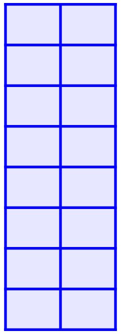
- **Associativity** (increase, all else equal)
  - + Decreases conflict misses
  - – Increases t_{hit}
- **Block size** (increase, all else equal)
  - – Increases conflict misses
  - + Decreases compulsory misses
  - ± Increases or decreases capacity misses
  - • Negligible effect on t_{hit}
- **Capacity** (increase, all else equal)
  - + Decreases capacity misses
  - – Increases t_{hit}



more columns (ways), fewer rows (sets), same area



fewer rows (sets), bigger blocks, same area



more area via more rows (sets)

Inclusion/Exclusion

- If L2 holds superset of every block in L1, then L2 is inclusive with respect to L1
- If L2 holds no block that is in L1, then L2 and L1 are exclusive
- L2 could be neither inclusive nor exclusive
    - Has some blocks in L1 but not all
- This issue matters a lot for multicores, but not a major issue in this class

- Same issue for L3/L2

Stores: Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately (as soon as store writes to this level)
    - + Conceptually simpler
    - + Uniform latency on misses
    - – Requires additional bandwidth to next level
  - **Write-back**: later, when block is replaced from this level
    - Requires additional "dirty" bit per block → why?
    - + Minimal bandwidth to next level
      - Only write back dirty blocks
    - – Non-uniform miss latency
      - Miss that evicts clean block: just a fill from lower level
      - Miss that evicts dirty block: writeback dirty block and then fill from lower level

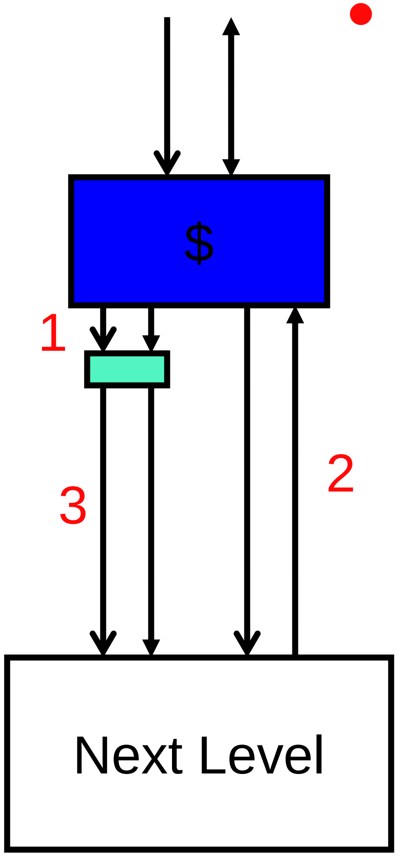
Stores: Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-allocate**: read block from lower level, write value into it
    - + Decreases read misses
    - – Requires additional bandwidth
    - Use with write-back
  - **Write-non-allocate**: just write to next level
    - – Potentially more read misses
    - + Uses less bandwidth
    - Use with write-through

Cache behavior summary

				Your cache is:	
				Write-through, write-no-allocate	**Write-back, write-allocate**
We try to	*load*	**And it's...**	*in the cache*	LOAD HIT: Take the block from cache and find the word the CPU wanted and provide it.	< Same as that
We try to	*load*	**And it's...**	*not in the cache*	LOAD MISS: Go to the next lower level and fetch the whole block, storing it in cache. We may have to evict something to make room. Finally, give the CPU the word it wanted.	< Same as that, *but* if the block we decide to evict is dirty, we have to write the changes out to the next lower level before evicting.
We try to	*store*	**And it's...**	*in the cache*	STORE HIT: Commit the change to the copy in cache *and* to the next lower level.	STORE HIT: Commit the change to the copy in cache and *don't* change it in the next lower level. Now this cache has the most up to date copy, and the level under us is out of date. Mark this block "dirty" so we remember to flush these changes during eviction later.
We try to	*store*	**And it's...**	*not in the cache*	STORE MISS: Commit the change to the next lower level. Do *not* put the block into this cache (that's "write-no-allocate").	STORE MISS: Bring the whole block into cache, evicting something else if needed (and flushing it to the lower level if it was dirty). Now that it's in cache, update it in cache and mark it dirty, as above.

Optimization: Write Buffer



- **Write buffer**: between cache and memory
  - Write-through cache? Helps with store misses
    - + Write to buffer to avoid waiting for next level
      - Store misses become store hits
  - Write-back cache? Helps with dirty misses
    - + Allows you to do read (important part) first
      1. Write dirty block to buffer
      2. Read new block from next level to cache
      3. Write buffer contents to next level

Typical Processor Cache Hierarchy

- First level caches: optimized for t_{hit} and parallel access
  - Insns and data in separate caches (**I$**, **D$**) → why?
  - Capacity: 8–64KB, block size: 16–64B, associativity: 1–4
  - Other: write-through or write-back
  - t_{hit}: 1–4 cycles
- Second level cache (**L2**): optimized for $\%_{miss}$
  - Insns and data in one cache for better utilization
  - Capacity: 128KB–1MB, block size: 64–256B, associativity: 4–16
  - Other: write-back
  - t_{hit}: 10–20 cycles
- Third level caches (**L3**): also optimized for $\%_{miss}$
  - Capacity: 2–16MB
  - t_{hit}: ~30 cycles

Performance Calculation Example

- Parameters
  - Reference stream: 20% stores, 80% loads
  - L1 D$: t_{hit} = 1ns, $\%_{miss}$ = 5%, write-through + write-buffer
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%, write-back, 50% dirty blocks
  - Main memory: t_{hit} = 50ns, $\%_{miss}$ = 0%
- What is $t_{avgL1D\$}$ without an L2?
  - Write-through+write-buffer means all stores effectively hit
  - $t_{missL1D\$} = t_{hitM}$
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads} * \%_{missL1D\$} * t_{hitM}$ = 1ns+(0.8*0.05*50ns) = 3ns
- What is $t_{avgD\$}$ with an L2?
  - $t_{missL1D\$} = t_{avgL2}$
  - Write-back (no buffer) means dirty misses cost double
  - $t_{avgL2} = t_{hitL2} + (1 + \%_{dirty}) * \%_{missL2} * t_{hitM}$ = 10ns+(1.5*0.2*50ns) =25ns
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads} * \%_{missL1D\$} * t_{avgL2}$ = 1ns+(0.8*0.05*25ns) =2ns

Cost of Tags

- “4KB cache” means cache holds 4KB of data
  - Called **capacity**
  - Tag storage is considered overhead (not included in capacity)
- Calculate tag overhead of 4KB cache with 1024 4B frames
  - Not including valid bits
  - 4B frames → 2-bit offset
  - 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag * 1024 frames = 20Kb tags = 2.5KB tags
  - 63% overhead → much higher than usual because blocks are so small (and cache is small)

Cache structure math summary

- Given capacity, block_size, ways (associativity), and word_size.
- Cache parameters:
  - num_frames = capacity / block_size
  - sets = num_frames / ways = capacity / block_size / ways
- Address bit fields:

Tag	Index	Block offset

  - offset_bits = $\log_2$(block_size)
  - index_bits = $\log_2$(sets)
  - tag_bits = word_size - index_bits - offset_bits
- Way to get offset/index/tag from address (bitwise & numeric):
  - block_offset = addr & ones(offset_bits) = addr % block_size
  - index = (addr >> offset_bits) & ones(index_bits)
    = (addr / block_size) % sets
  - tag = addr >> (offset_bits+index_bits) = addr / (sets*block_size)

ones(n) = a string of *n* ones = ((1<<n)-1)

What this means to the programmer

- If you’re writing code, you want good performance.
- The cache is **crucial** to getting good performance.
- The effect of the cache is influenced by the **order of memory accesses**.

CONCLUSION:

The programmer can change the order of memory accesses to improve performance!

Cache performance matters!

- A **HUGE** component of software performance is how it interacts with cache
- Example:

Assume that x[i][j] is stored next to x[i][j+1] in memory ("row major order").

Which will have fewer cache misses?

```
for (k = 0; k < 100; k++)
  for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
            x[i][j] = 2 * x[i][j];
```

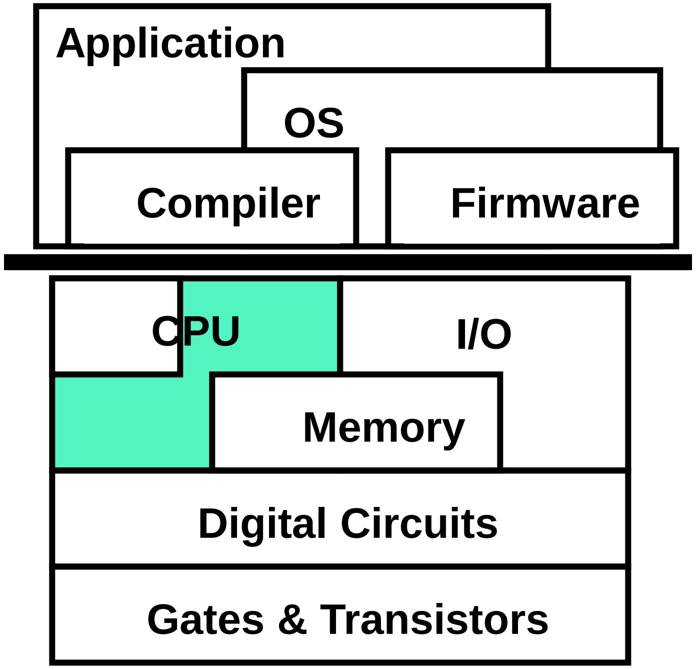
A

```
for (k = 0; k < 100; k++)
  for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
            x[i][j] = 2 * x[i][j];
```

B ←

Adapted from Lebeck and Porter (creative commons)

This Unit: Caches and Memory Hierarchies



- Memory hierarchy
- Cache organization
- Cache implementation

What do we make cache out of?

- We said that main memory had to be DRAM

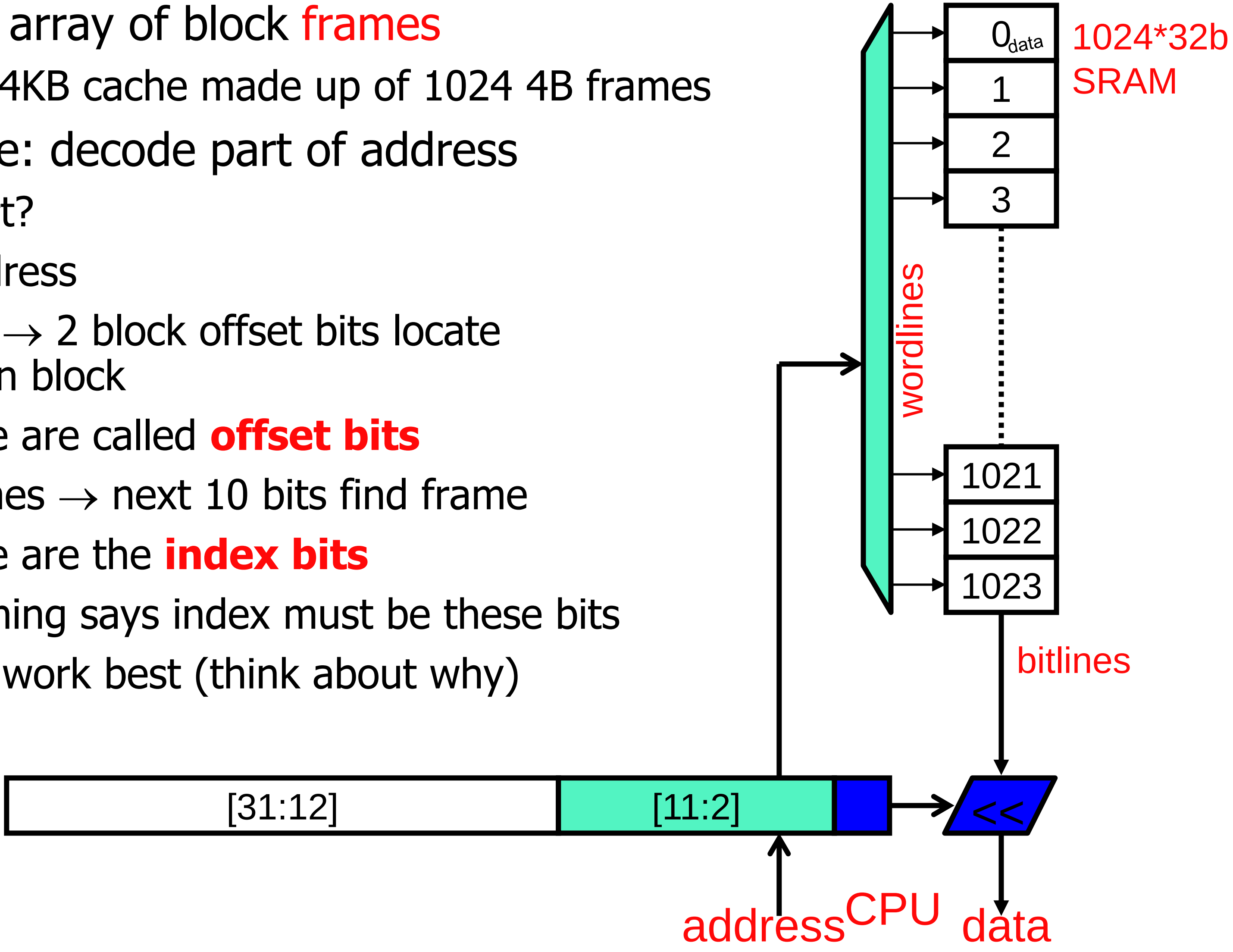
- But DRAM is slow, so we invented caches

- But what are the caches made of?
  - Do we know of a kind of RAM that is small but fast?
    - SRAM!!!!

- Caches are made of SRAM. Let’s see how.

Basic Cache Structure

- Basic cache: array of block frames
  - Example: 4KB cache made up of 1024 4B frames
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 4B blocks → 2 block offset bits locate byte within block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
  - But these work best (think about why)

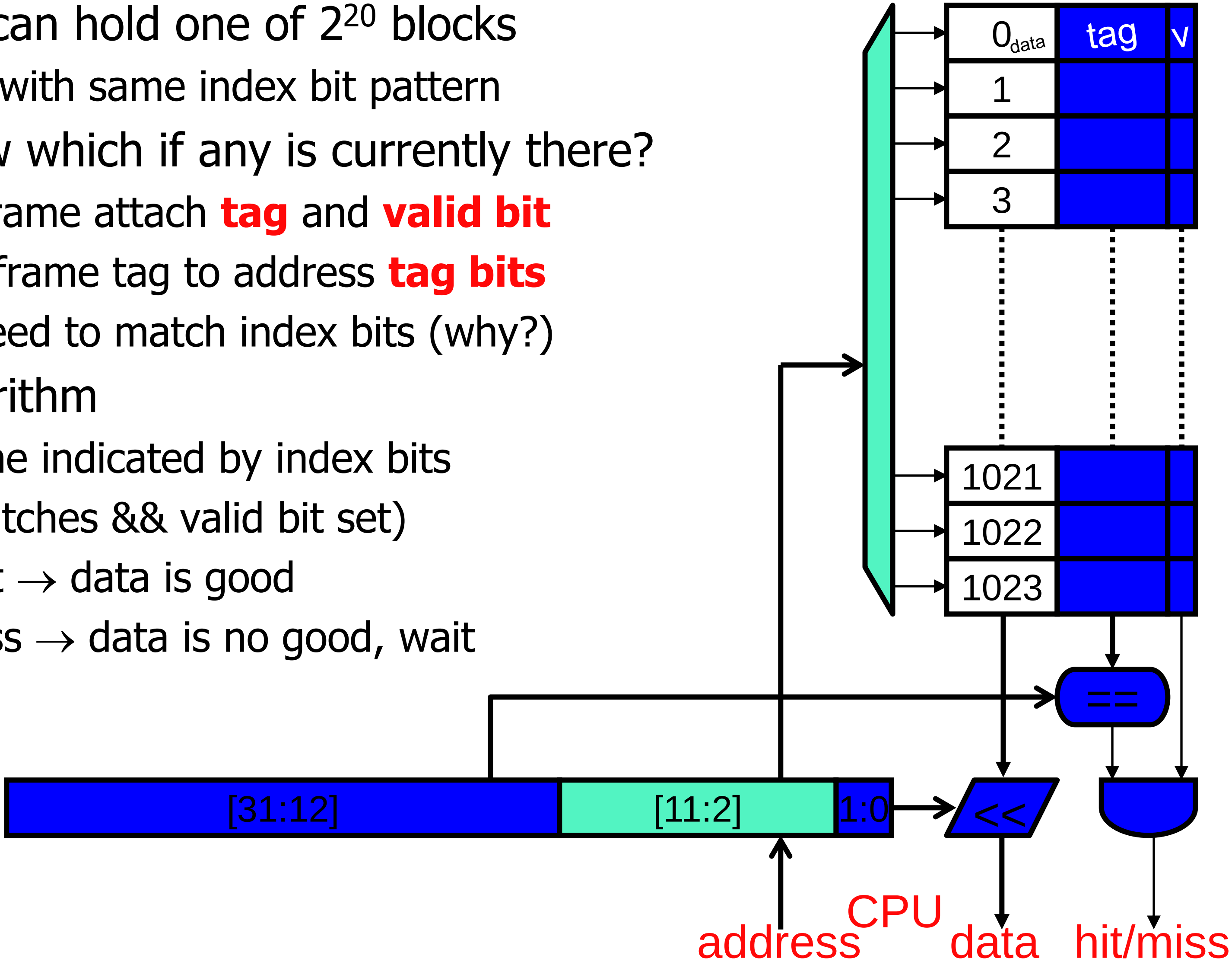


Basic Cache Structure

- Each frame can hold one of 2^{20} blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - If (tag matches && valid bit set)

    then Hit → data is good

    Else Miss → data is no good, wait

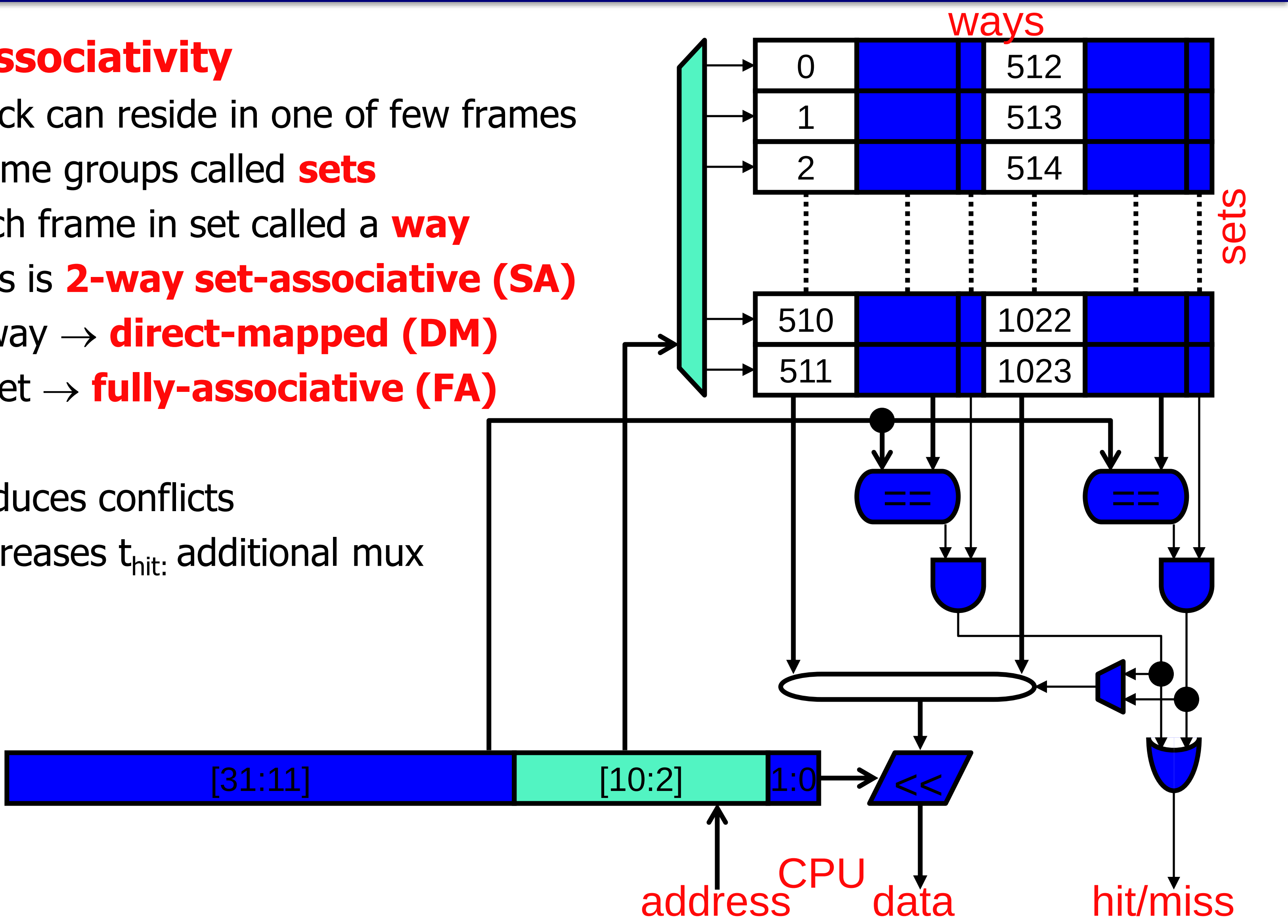


Set-Associativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called **sets**
  - Each frame in set called a **way**
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

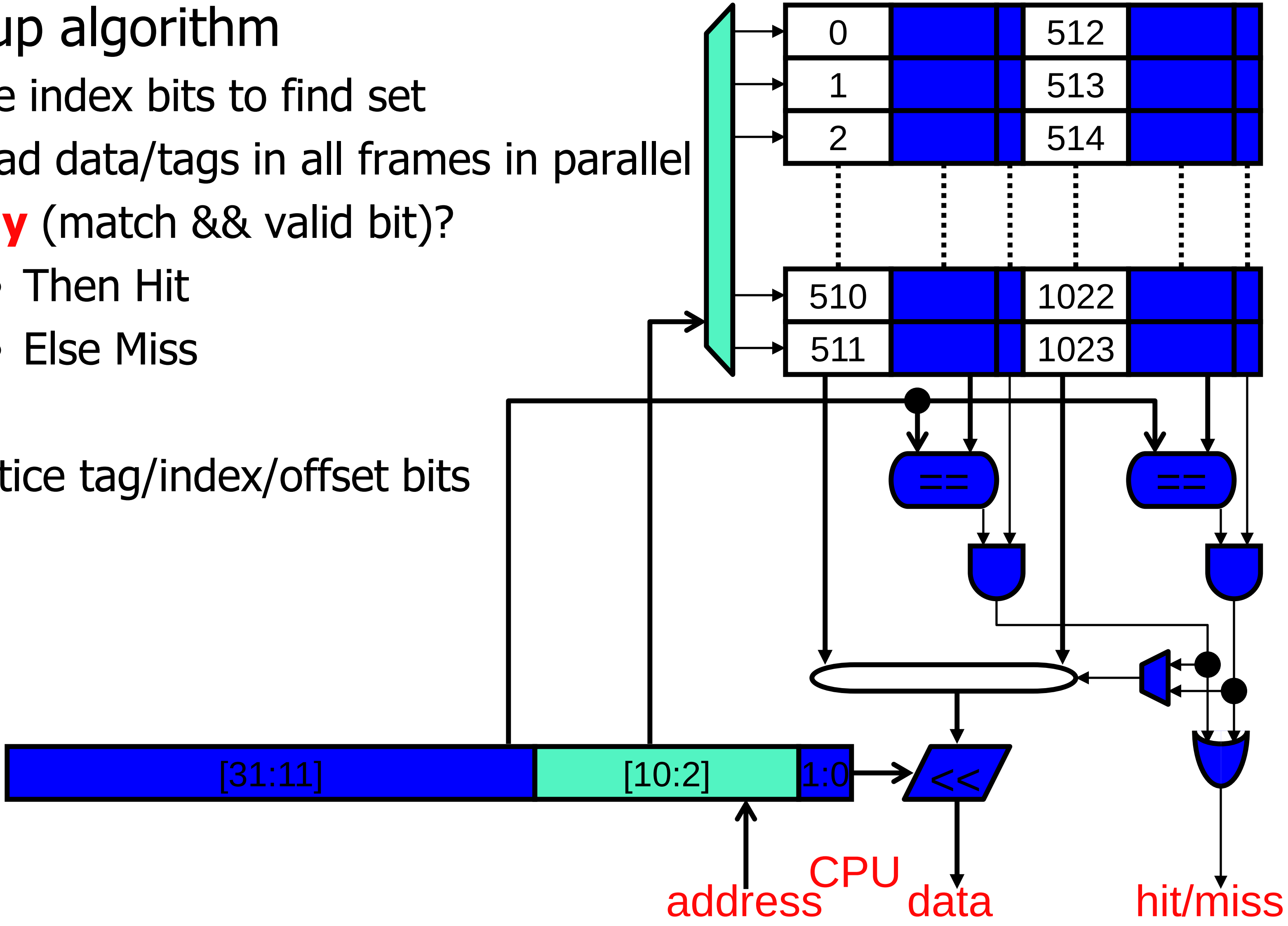
+ Reduces conflicts

– Increases $t_{hit:}$ additional mux



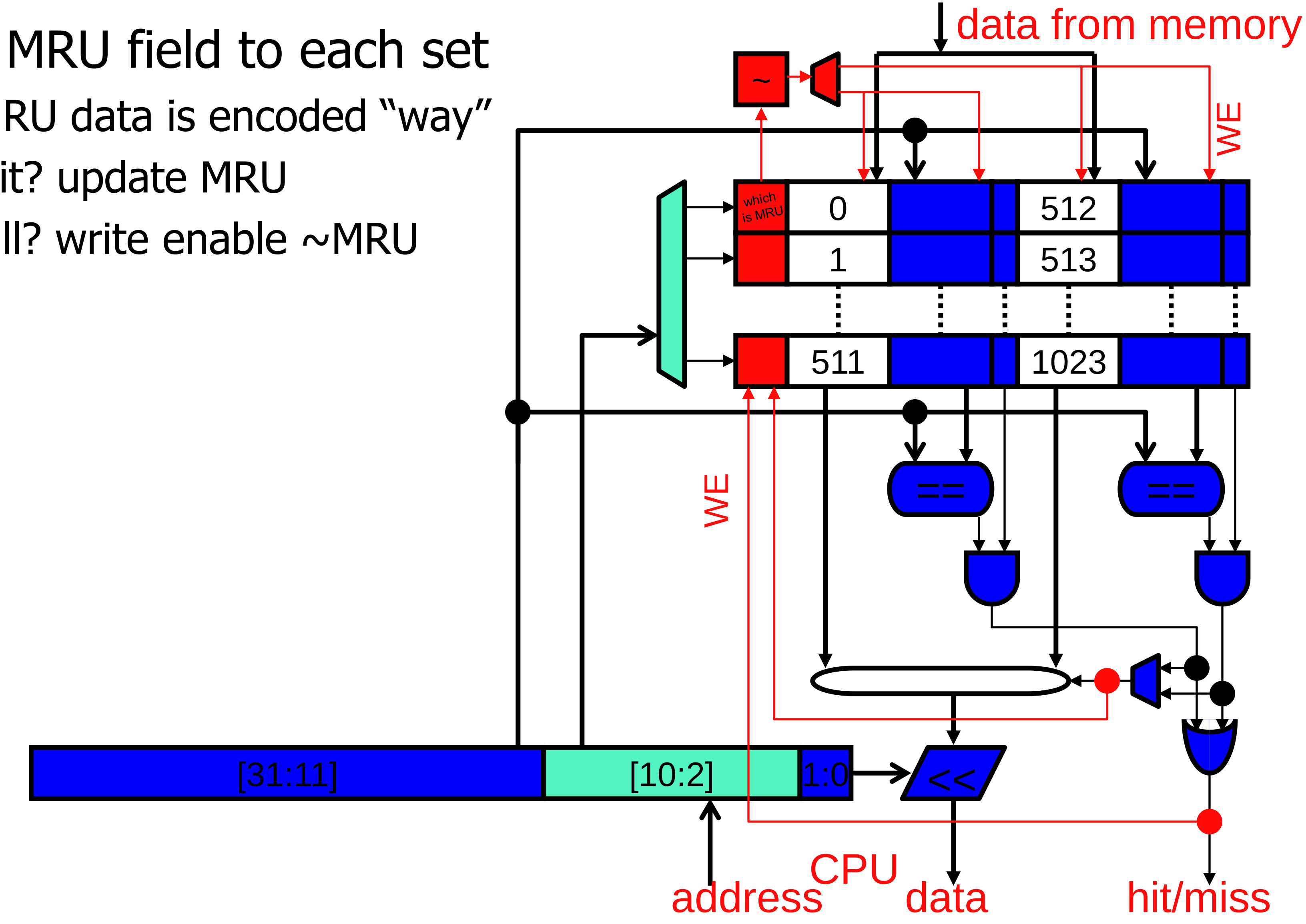
Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - **Any** (match && valid bit)?
    - Then Hit
    - Else Miss
  - Notice tag/index/offset bits

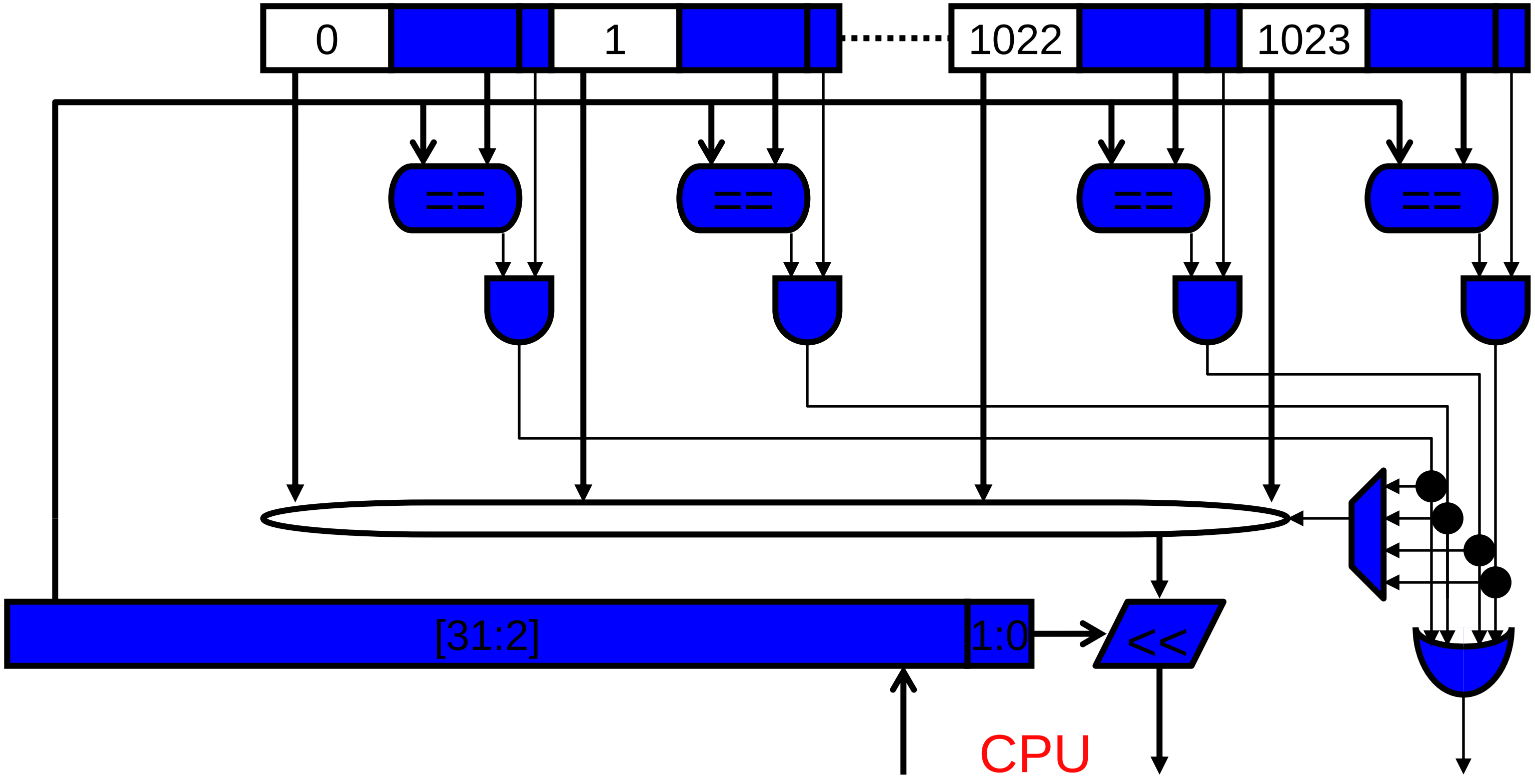


NMRU and Miss Handling

- Add MRU field to each set
  - MRU data is encoded "way"
  - Hit? update MRU
  - Fill? write enable ~MRU



Full-Associativity



- How to implement full (or at least high) associativity?
  - Doing it this way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?

Normal RAM vs Content Addressable Memory

RAM

- Cell number 5, what are you storing?

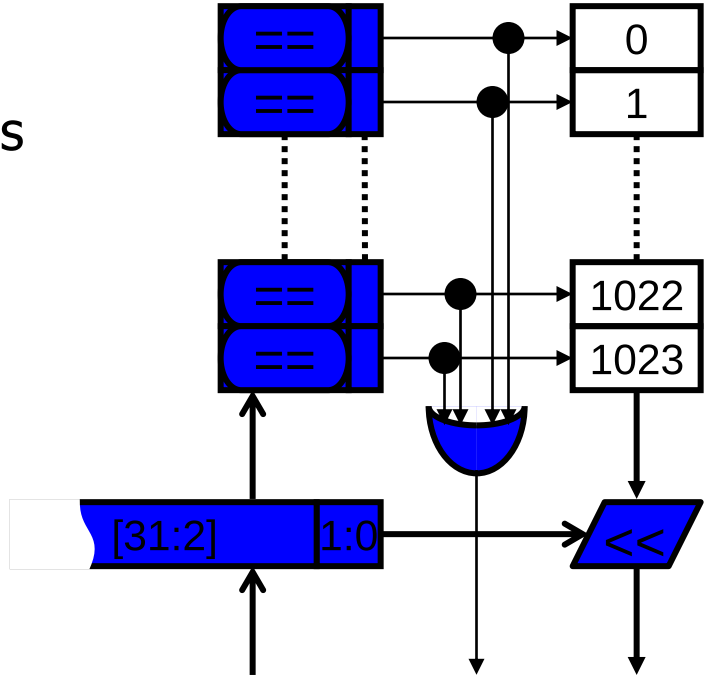
CAM

- Attention all cells, will the owner of data “12” please stand up?

i cant really think of a video game person who is a CAM, so how about, like, isabelle?

Full-Associativity with CAMs

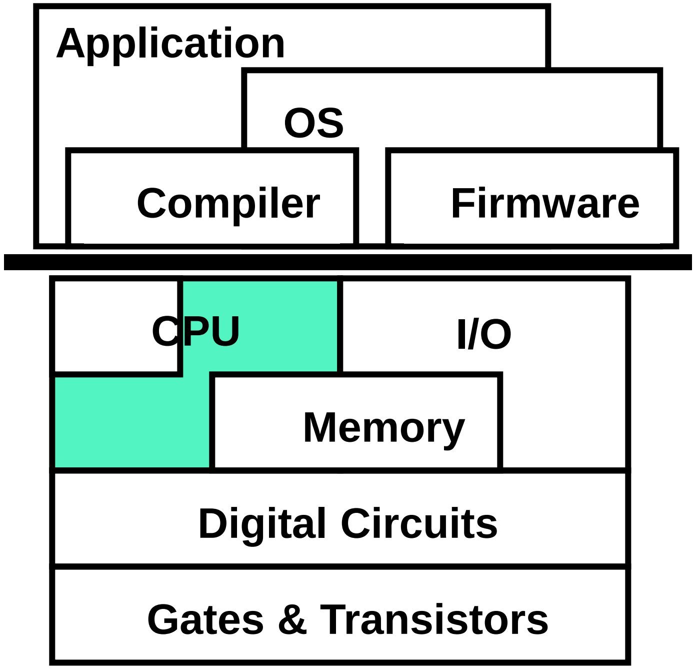
- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is "one-hot" encoding of match
- FA cache?
  - Tags as CAM
  - Data as RAM



CAM Upshot

- CAMs are effective but expensive
  - – Matchlines are very expensive (for nasty circuit-level reasons)
  - CAMs are used but only for 16 or 32 way (max) associativity
  - Not for 1024-way associativity
    - – No good way of doing something like that
    - + No real need for it either

This Unit: Caches and Memory Hierarchies



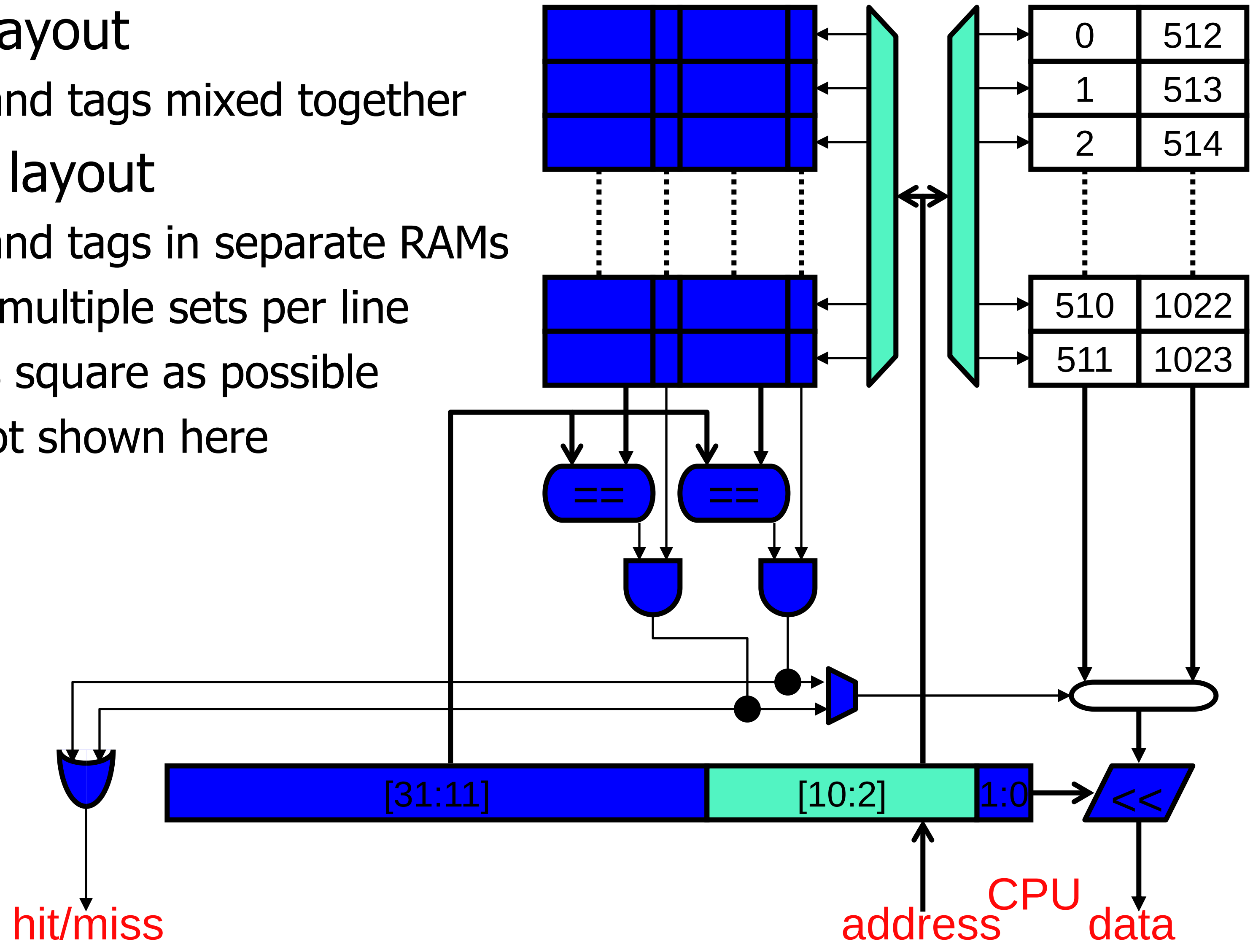
- Memory hierarchy
- Cache organization
- Cache implementation

Extra material

Cache structure and some optimizations

Physical Cache Layout

- Logical layout
  - Data and tags mixed together
- Physical layout
  - Data and tags in separate RAMs
  - Often multiple sets per line
    - As square as possible
    - Not shown here



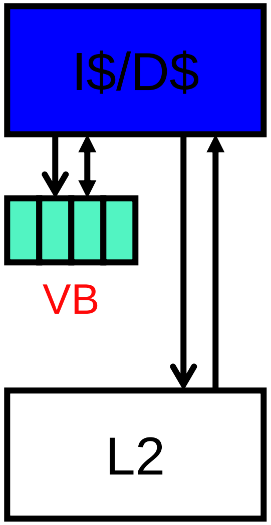
Two (of many possible) Optimizations

- **Victim buffer**: for conflict misses
- **Prefetching**: for capacity/compulsory misses

Victim Buffer

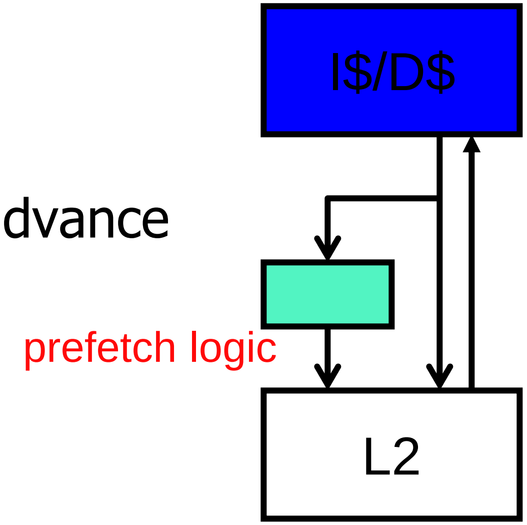
- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (ABC)*

- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Sits on I$/D$ fill path
  - VB is small → very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit ? Place block back in I$/D$
  - 4 extra ways, shared among all sets
    - + Only a few sets will need it at any given time
  - + Very effective in practice



Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - Key: anticipate upcoming miss addresses accurately
    - Can do in software or hardware
  - Simple example: **next block prefetching**
    - Miss on address **X** → anticipate miss on **X+block-size**
    - Works for insns: sequential execution
    - Works for data: arrays
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don’t evict useful data



Other techniques in cache-efficient coding

Blocking (Tiling) Example

```
/* Before */
 for(i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      for (k = 0; k < SIZE; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Two Inner Loops:
  - Read all NxN elements of z[ ] (N = SIZE)
  - Read N elements of 1 row of y[ ] repeatedly
  - Write N elements of 1 row of x[ ]
- Capacity Misses a function of N & Cache Size:
  - 3 NxN => no capacity misses; otherwise ...
- Idea: compute on BxB submatrix that fits

Adapted from Lebeck and Porter (creative commons)

Blocking (Tiling) Example

```
/* After */
 for(ii = 0; ii < SIZE; ii += B)
    for (jj = 0; jj < SIZE; jj += B)
      for (kk = 0; kk < SIZE; kk +=B)
        for(i = ii; i < MIN(ii+B-1,SIZE); i++)
          for (j = jj; j < MIN(jj+B-1,SIZE); j++)
            for (k = kk; k < MIN(kk+B-1,SIZE); k++)
               c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

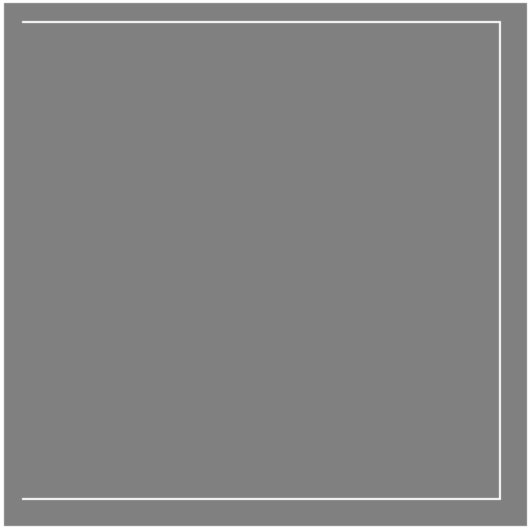
- Capacity Misses decrease

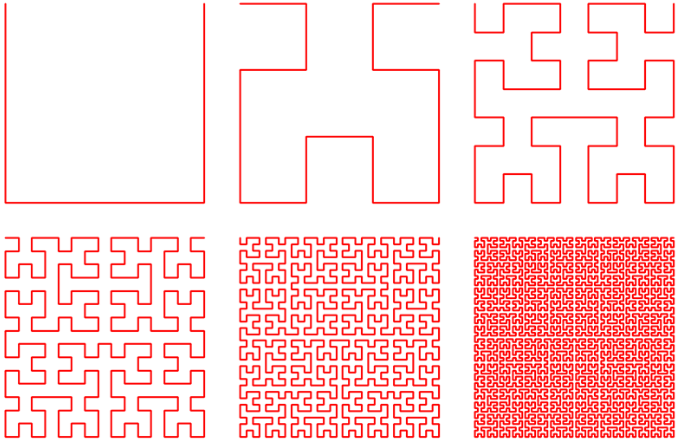
  $2N^3 + N^2$ to $2N^3/B + N^2$
- B called *Blocking Factor (Also called Tile Size)*

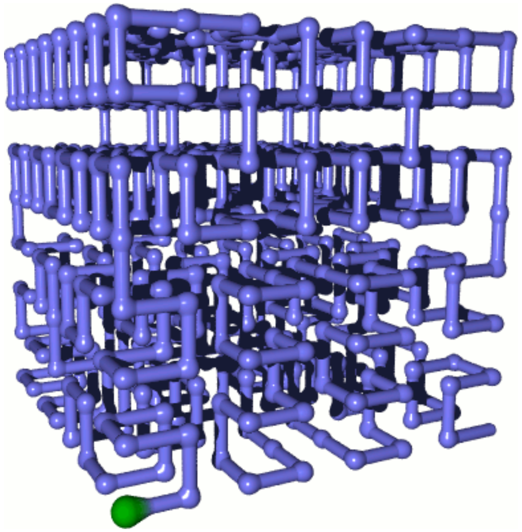
Adapted from Lebeck and Porter (creative commons)

Hilbert curves: A fancy trick for matrix locality

- Turn a 1D value into an n-dimensional “walk” of a cube space (like a 2D or 3D matrix) in a manner that maximizes locality
- Extra overhead to compute curve path, but computation takes no memory, and cache misses are very expensive, so it may be worth it
- (Actual algorithm for these curves is simple and easy to find)



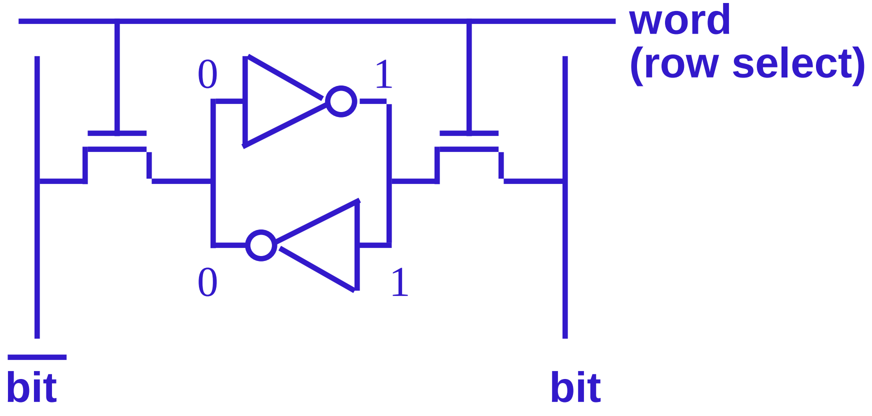


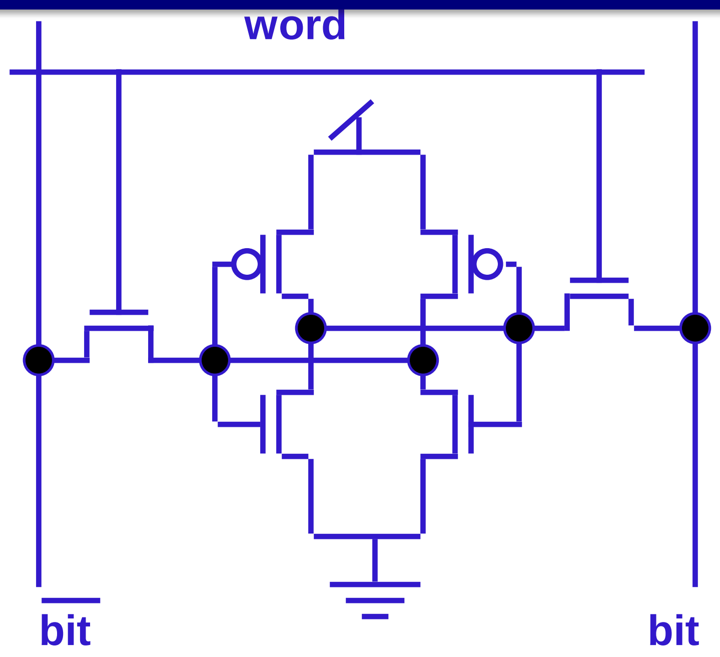


SRAM internals

One Static RAM Cell

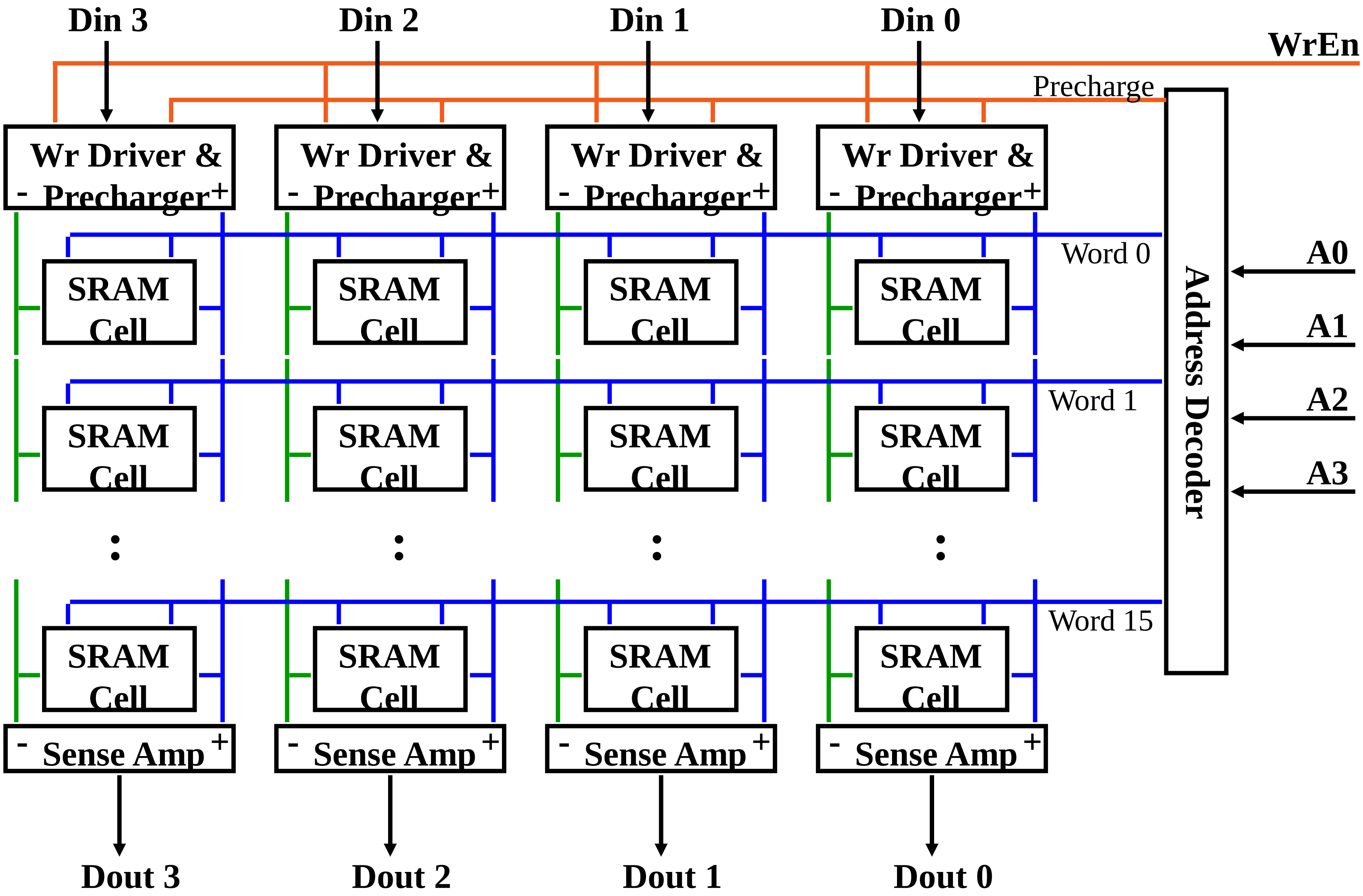
6-Transistor SRAM Cell



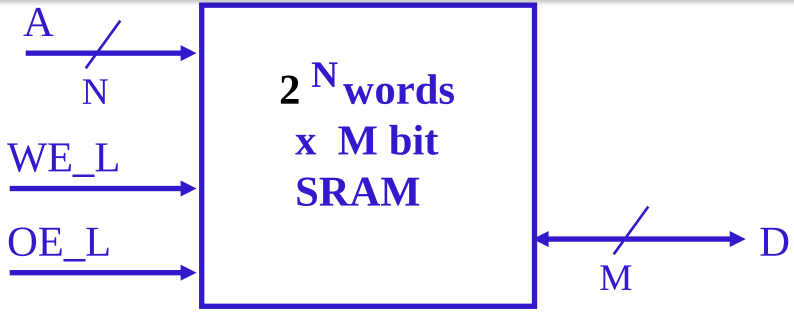


- To write (a 1):
  1. Drive bit lines (bit=1, $\overline{\text{bit}}$=0)
  2. Select row
- To read:
  1. Pre-charge bit and $\overline{\text{bit}}$ to Vdd (set to 1)
  2. Select row
  3. Cell pulls one line lower (pulls towards 0)
  4. Sense amp on column detects difference between bit and bit

Typical SRAM Organization: 16-word x 4-bit



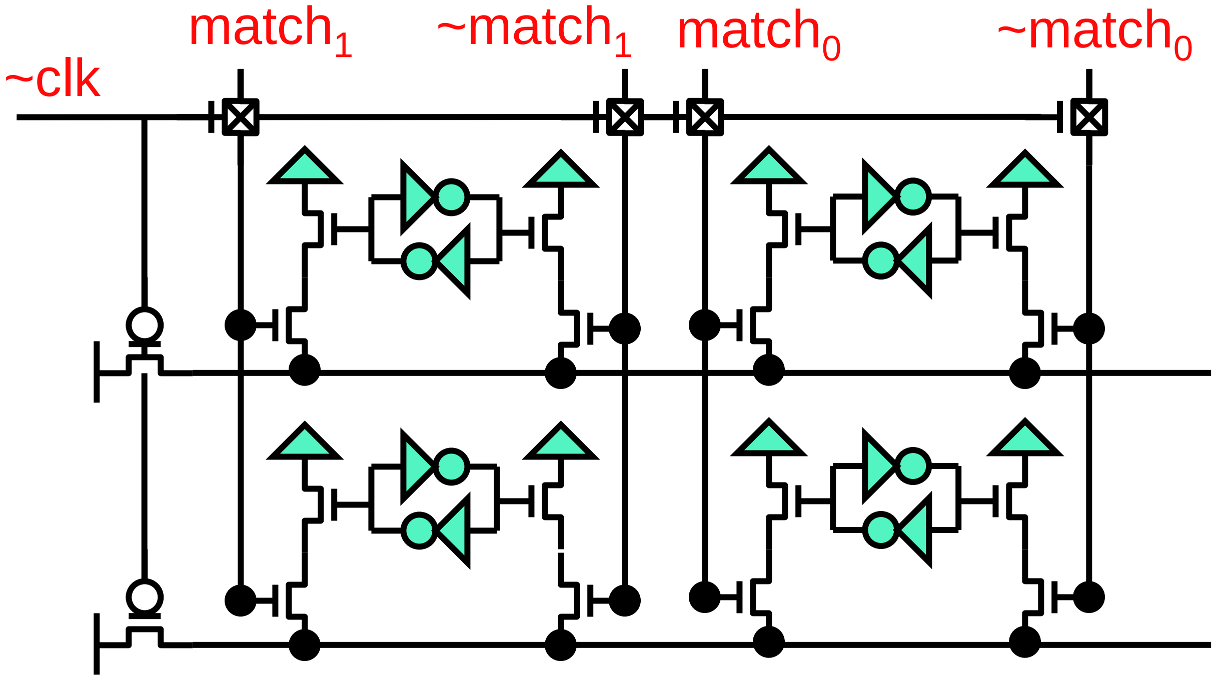
Logic Diagram of a Typical SRAM



- Write Enable is usually active low (WE_L)
- Din and Dout are combined (D) to save pins:
    - A new control signal, output enable (OE_L) is needed
    - WE_L is asserted (Low), OE_L is de-asserted (High)
        - D serves as the data input pin
    - WE_L is de-asserted (High), OE_L is asserted (Low)
        - D is now the data output pin
    - Both WE_L and OE_L are asserted:
        - Result is unknown. Don't do that!!!

CAM internals

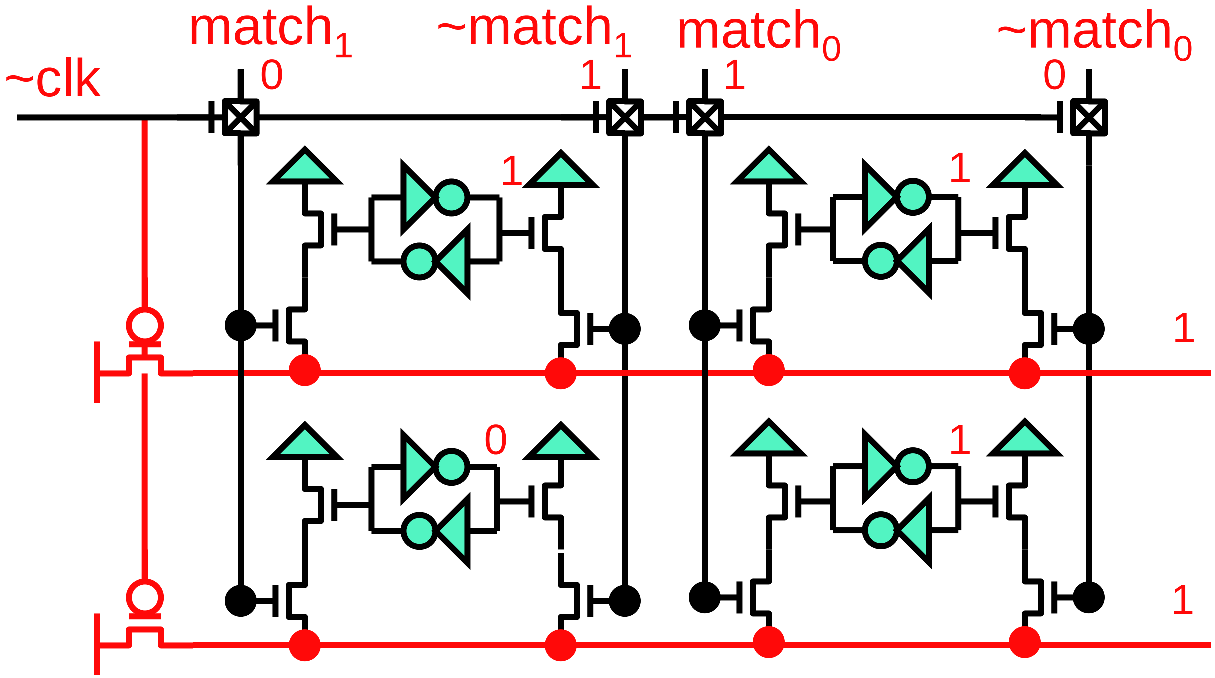
CAM Circuit



- **Matchlines** (correspond to bitlines in SRAM): inputs
- **Wordlines**: outputs

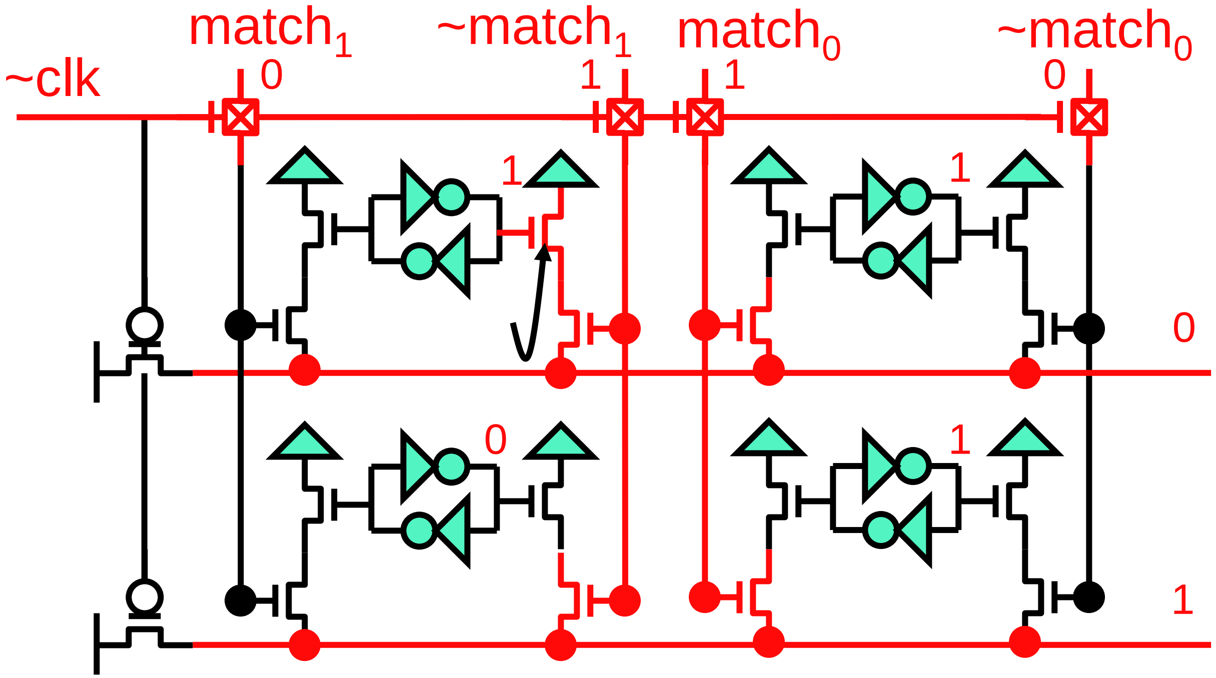
- Two phase match
  - Phase I: clk=1, pre-charge wordlines to 1
  - Phase II: clk=0, enable matchlines, non-matched bits dis-charge wordlines

CAM Circuit In Action



- Phase I: clk=1
  - Pre-charge wordlines to 1

CAM Circuit In Action



Looking for match with 01

- Phase I: clk=0
  - Enable matchlines (notice, match bits are flipped)
  - Any non-matching bit discharges entire wordline
    - Implicitly ANDs all bit matches (NORs all bit non-matches)
  - Similar technique for doing a fast OR for hit detection