CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 19:
Caches II: Direct-Mapped and Set Associative, Types of Misses

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

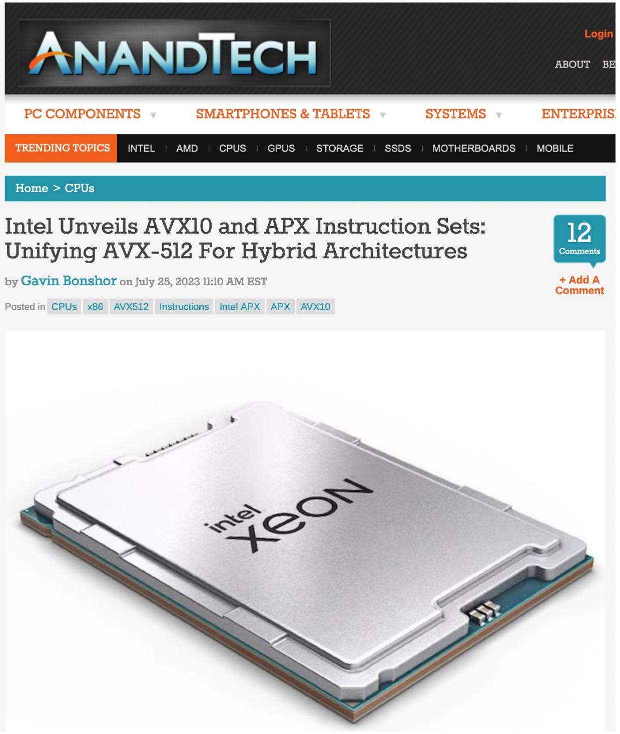
61C in the News: New Intel ISA Extensions

7/25/2023:

“Intel has announced two new x86-64 instruction sets designed to bolster and offer more performance in AVX-based workloads with their hybrid architecture of performance (P) and efficiency (E) cores [...]

Intel has also published a technical paper detailing their new AVX10, **enabling both Intel's performance (P) and efficiency (E) cores to support the converged AVX10/256-bit instruction set going forward**. This means that Intel's future generation of hybrid desktop, server, and workstation chips will be able to support multiple AVX vectors, including 128, 256, and 512-bit vector sizes throughout the entirety of the cores holistically [...]

The idea behind APX is to **allow access to more registers** [...] with Intel claiming 10% fewer loads and 20% fewer stores when the code is compiled for APX versus the same code for x86-64 using Intel 64 [...] Another essential feature of Intel's APX is its support for **three-operand instruction formats**.”


ANANDTECH

Login

ABOUT BE

PC COMPONENTS ▾ SMARTPHONES & TABLETS ▾ SYSTEMS ▾ ENTERPRIS

TRENDING TOPICS | INTEL | AMD | CPUS | GPUS | STORAGE | SSDS | MOTHERBOARDS | MOBILE

Home > CPUs

Intel Unveils AVX10 and APX Instruction Sets: Unifying AVX-512 For Hybrid Architectures

12 Comments

by Gavin Bonshor on July 25, 2023 11:10 AM EST

+ Add A Comment

Posted in CPUs x86 AVX512 Instructions Intel APX APX AVX10

https://www.anandtech.com/show/18975/intel-unveils-avx10-and-apx-isas-unifying-avx512-for-hybrid-architectures-

Announcements

- P/NP deadline is Friday, July 28
    - Student support meetings available with course staff (see Ed)
    - As always, instructor OH open for support, course feedback, or anything else!
- Exam prep sections continue
    - Parallelism and some caches: Thursday 1-3pm in Soda 380, Friday 1-3pm in Cory 540AB
- HW5 due tonight!
- Labs 7 and 8 due tomorrow, Thursday 7/27

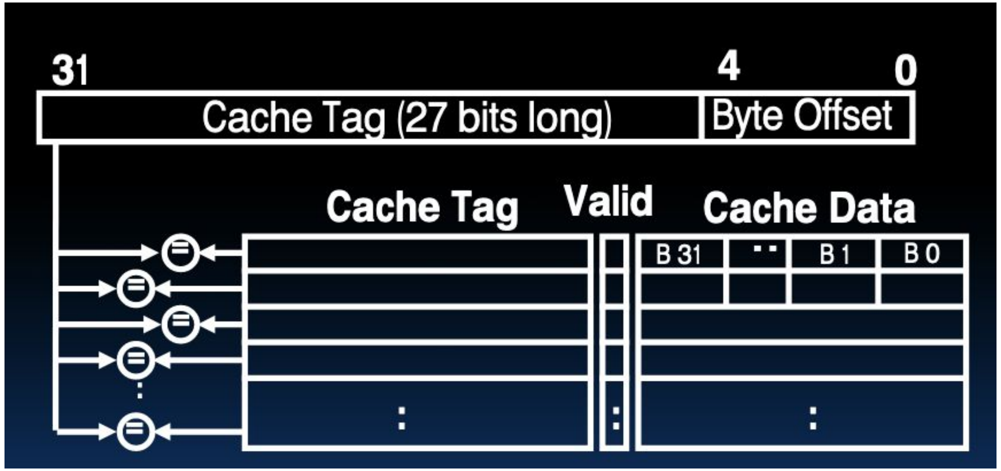
Agenda

- Direct-Mapped Cache
- Set Associate Cache
- Types of Cache Misses

Direct-Mapped Cache

Problems with Fully-Associative Caches

- While nice, an actual implementation of a fully associative cache has certain problems:
  - Hard to find a good and efficient replacement policy
  - Requires a lot of comparators/circuitry.



Problems with Fully-Associative Caches

- While nice, an actual implementation of a fully associative cache has certain problems:
  - Hard to find a good and efficient replacement policy
  - Requires a lot of comparators/circuitry
- Reason: Any slot in our cache could store our data, so we need to check all spots
- Library analogy: We don't have any organization scheme in our bookshelf, so it takes a long time to check if our data is in our bookshelf
- Cache tag checking circuit may be in CPU critical path
- Solutions?

Problems with Fully-Associative Caches

- Idea #1: Sort the data in the cache, so we can check in log time.
    - Works for a bookshelf; doesn't work in caches
    - Takes more time to move data between blocks than to check tag bits
    - Would still need a lot of comparators to do this
- Idea #2: Separate books into groups and only assign books to a cache block matching their group
    - Library analogy: Have the top shelf for books that start with "A", the second shelf for books that start with "B", etc. That way, we only need to check 1/26 of our bookshelf.

Idea 2: Picking a grouping mechanism

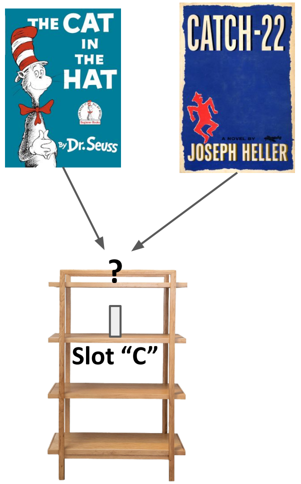
- Books can be grouped based on genre, name, etc. How can we group cache blocks?
- Hash the address?
  - Used in 61B to distribute data into even groups, but we don’t want the extra hardware
  - Plus, we can do better than random
- Top N bits of the address?
  - Doesn't distribute blocks well
  - In practice, we'll be using consecutive blocks of memory, and chances are that consecutive blocks will have the same top N bits of memory address
- **Bottom N bits of the tag**: Works well with consecutive blocks of memory
  - Also scales to arbitrarily many groups without much additional circuitry
  - Note: Number of groups must be a power of 2.

Index Bits

- Bottom N bits of the tag: Works well with consecutive blocks of memory
  - Also scales to arbitrarily many cache slots without much additional circuitry
- A memory address can be divided into a tag, **an index**, and an offset
- Ex. Let's say we have a block size of 4096 and 16 distinct groups
  - Ex. **`0xDEADBEEF`**
  - This address is in Block `0xDEADB`
    - This block has tag `0xDEAD` and index `0xB`
    - We want the `0xEEF`th byte in that block
- Determining the number of bits in each component of a cache address is called the "**TIO** breakdown", for "Tag, Index, Offset".

Direct-Mapped Cache

- Each slot in a direct-mapped cache is assigned a unique index; if we have 32 slots, one will be for index 0, one for index 1, … , and one for index 31
- When a memory access occurs:
  - Step 1: Check **the unique slot that could contain our data.** If the tag matches and the valid bit is on, cache hit.
  - Step 2: Otherwise, cache miss
- A direct-mapped cache is parameterized on two aspects: The block size, and the number of blocks that can be stored in the cache
  - Same as fully-associative
- Library analogy: You buy a bookshelf with N slots, each holding 1 book. Each book can only go in 1 specific slot
  - For a given book, only need to check one slot of your bookshelf
  - However, you might have to return books while slots are still empty.



Direct-Mapped Cache: Example Memory Access (1/2)

- Say we have a direct-mapped cache with 4 byte blocks and 4 blocks storage
- Addresses are 10 bits
- To access address 0x3CD:
  - Split into TIO:
    - 4 bytes blocks -> O = 2 bits
    - 4 indexes -> I = 2 bits
    - `0x3CD == 0b11 1100 11 01`
  - For index 3, tag matches, and valid bit is on, so hit
  - We get the 1st byte of the block, which is `0xDE`.

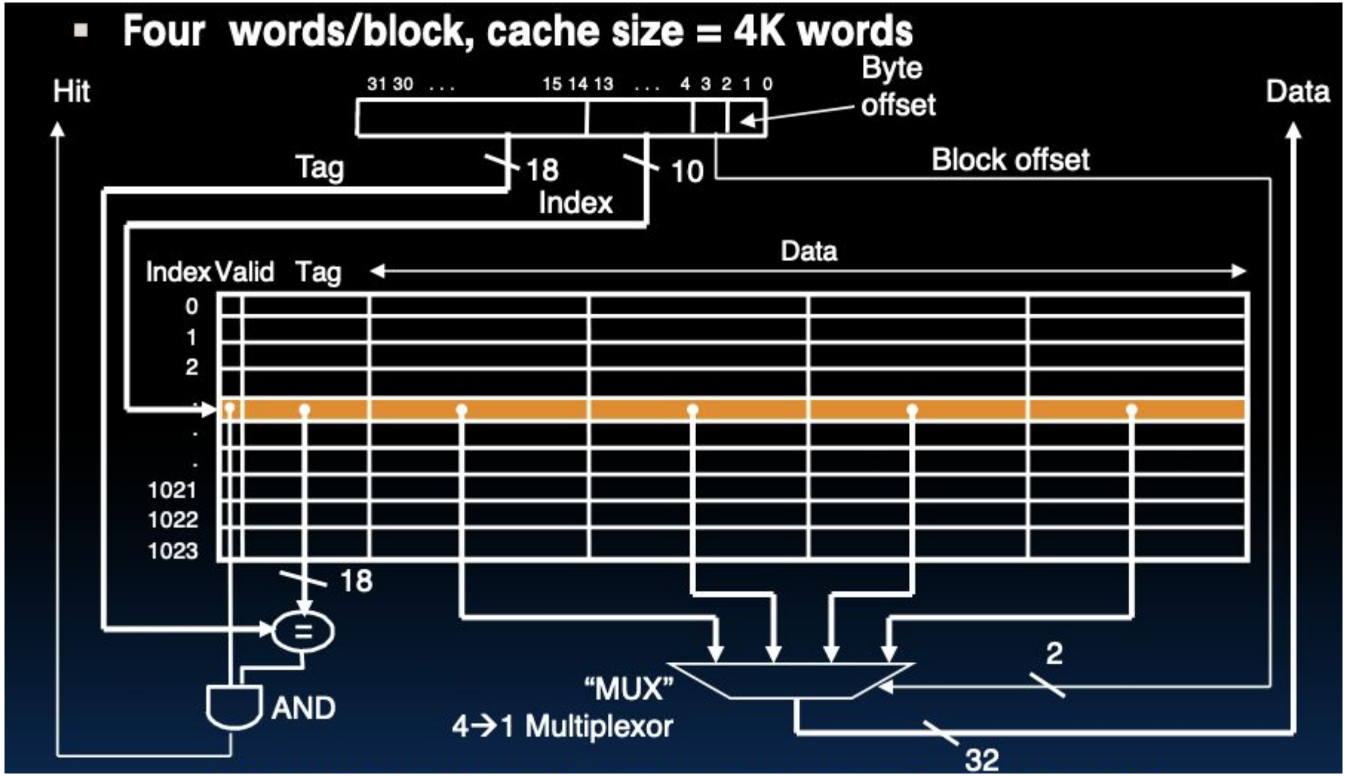
Index	Tag	V	Data
0	0b10 0110	0	0xDE 0xAD 0xBE 0xEF
1	0b10 1100	0	0x01 0x23 0x45 0x67
2	0b01 1010	1	0xAB 0xAD 0xCA 0xFE
3	0b11 1100	1	0xAC 0xDE 0xFF 0x61

Direct-Mapped Cache: Example Memory Access (2/2)

- To access address 0x3C1:
  - Split into TIO:
    - 4 bytes blocks -> O = 2 bits
    - 4 indexes -> I = 2 bits
    - `0x3CD == 0b11 1100 00 01`
  - For index 0, tag doesn't match, so miss, even though the correct tag is in another index
    - Note: same tag + different index is a different memory address.

Index	Tag	V	Data
0	`0b10 0110`	1	`0xDE 0xAD 0xBE 0xEF`
1	`0b10 1100`	0	`0x01 0x23 0x45 0x67`
2	`0b01 1010`	1	`0xAB 0xAD 0xCA 0xFE`
3	`0b11 1100`	1	`0xAC 0xDE 0xFF 0x61`

Direct-Mapped Cache Circuit



Direct-Mapped Cache Pros and Cons

- Pros:
  - Simpler circuit
    - Compare one tag rather than all tags
    - Faster accesses
    - Scales better than fully associative caches
  - No replacement policy
    - Only one block can be evicted, so no need to store LRU
- Cons:
  - If an index isn't used, we end up with free space in our cache that we can't use
  - Let's see how this behaves in Matrix Multiply with 4 blocks.

Caching Example: Write-Back Direct-Mapped

- 16-bit address, 16-byte blocks
- TIO?
    - 10, 2, 4

Index	Tag	V	Dirty	Data
0	`0x100`	0	0	`0xBF 0x16 0x88 0x2B`
1	`0x133`	0	0	`0x3B 0x18 0xF1 0xB3`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

🟪			

Caching Example: Write-Back Direct-Mapped

- Start: Cache starts cold
  - Note: Index field, and no LRU field
    - Index isn't actually stored, but we'll show it anyway
  - Note: Matrix B has been transposed into Bt to optimize cache efficiency
- 0 misses, 0 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Index	Tag	V	Dirty	Data
0	`0x100`	0	0	`0xBF 0x16 0x88 0x2B`
1	`0x133`	0	0	`0x3B 0x18 0xF1 0xB3`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

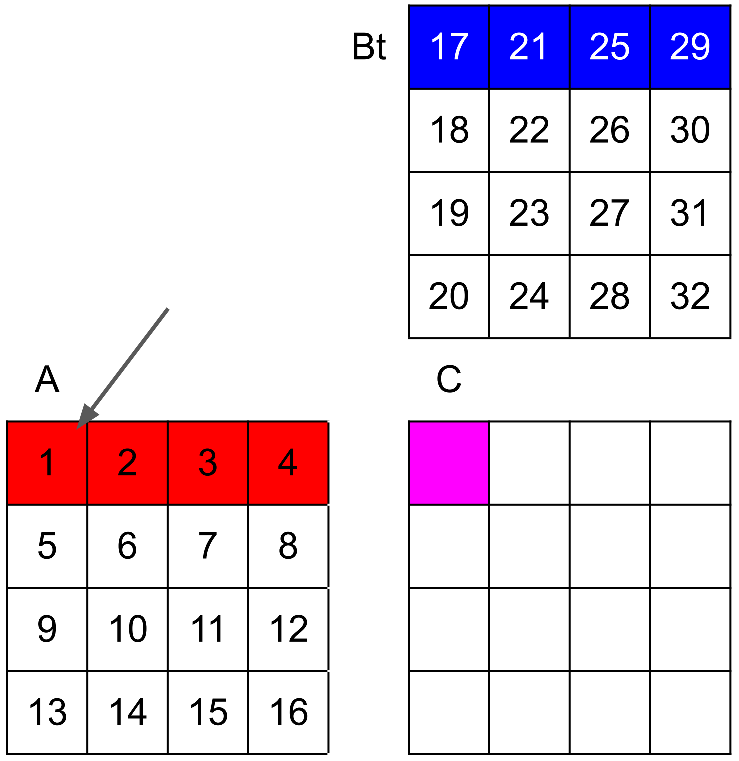
C

🟪			

Caching Example: Write-Back Direct-Mapped

- Access A[0]: 0x1000
  - Index 0b00 = 0, Tag 0x040 (top 10 bits)
  - Miss, so replace the block in index 0

Index	Tag	V	Dirty	Data
0	`0x100`	0	0	`0xBF 0x16 0x88 0x2B`
1	`0x133`	0	0	`0x3B 0x18 0xF1 0xB3`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`



Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

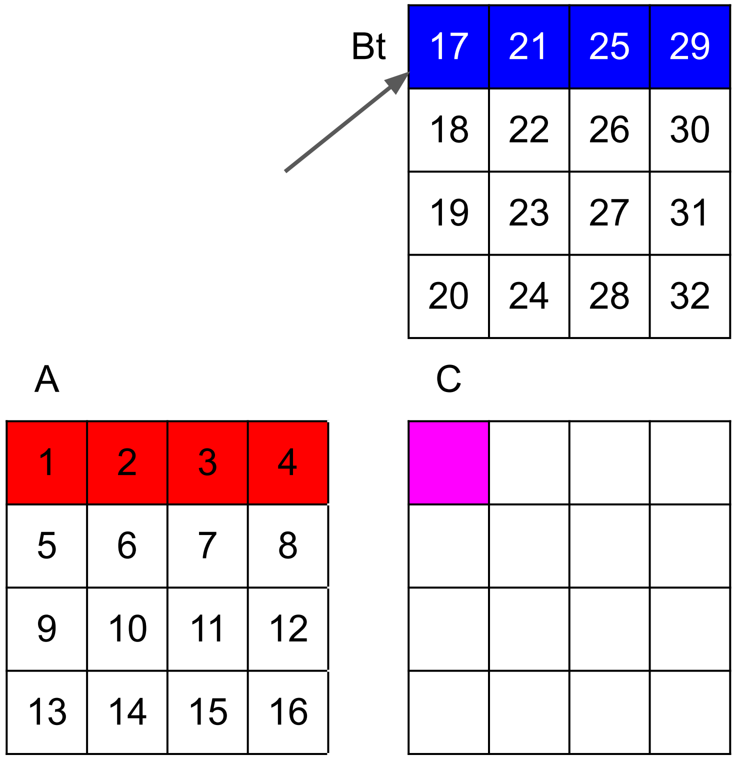
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Direct-Mapped

- Access B[0]: 0x2000
  - Index 0, Tag 0x060
  - Miss, so replace the block in index 0
    - Uh oh

Index	Tag	V	Dirty	Data
0	`0x040`	1	0	`0x01 0x02 0x03 0x04`
1	`0x133`	0	0	`0x3B 0x18 0xF1 0xB3`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`



Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

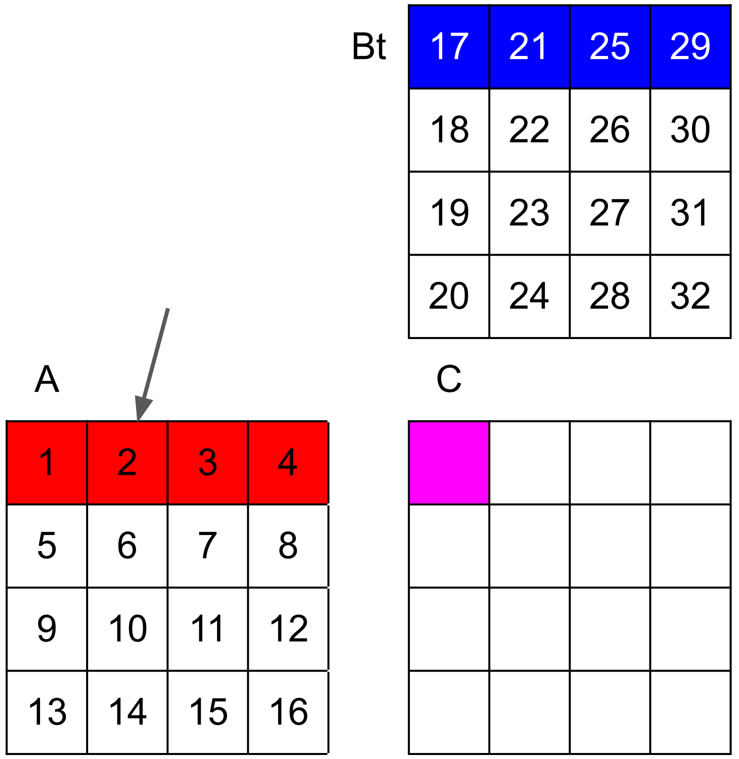
C

🟪			

Caching Example: Write-Back Direct-Mapped

- Access A[1]: 0x1001
    - Index 0, Tag 0x040
    - Miss, so replace the block in index 0
        - Oh no

Index	Tag	V	Dirty	Data
0	`0x060`	`1`	`0`	`0x11 0x15 0x19 0x1D`
1	`0x133`	`0`	`0`	`0x3B 0x18 0xF1 0xB3`
2	`0x156`	`0`	`0`	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	`0`	`0`	`0xB5 0x81 0x67 0x3F`



Caching Example: Write-Back Direct-Mapped

- B[1]: Miss, A[2]: Miss, B[2]: Miss, … , C[0]: Miss
- 9 misses, 0 hits…
- A[0]: Miss, B[4]: Miss, but now address is 0x2010, which is index 1

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Index	Tag	V	Dirty	Data
0	0x040	1	0	0x01 0x02 0x03 0x04
1	0x133	0	0	0x3B 0x18 0xF1 0xB3
2	0x156	0	0	0xE6 0x57 0x49 0xEE
3	0x0E9	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

🟪			

Caching Example: Write-Back Direct-Mapped

- A[1]: Hit, B[5]: Hit, … , C[1]: Miss, evicting Block 0x1000
- While calculating C[1]: 3 misses, 6 hits
    - Better!

Index	Tag	V	Dirty	Data
0	`0x040`	1	0	`0x01 0x02 0x03 0x04`
1	`0x060`	1	0	`0x12 0x16 0x1A 0x1E`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250			

Caching Example: Write-Back Direct-Mapped

- In general: if the element of C is on the diagonal, 9 misses, 0 hits
- Otherwise, 3 misses, 6 hits
- Total: 72 misses, 72 hits, or 50% hit rate
    - Back to pre-transpose hit rate!

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Index	Tag	V	Dirty	Data
0	`0x080`	1	1	`0xFA 0x04 0x?? 0x??`
1	`0x060`	1	0	`0x12 0x16 0x1A 0x1E`
2	`0x156`	0	0	`0xE6 0x57 0x49 0xEE`
3	`0x0E9`	0	0	`0xB5 0x81 0x67 0x3F`

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250			

Problem with Direct-Mapped Caches

- If we access two distinct segments of memory that access the same index, we end up continuously evicting blocks
    - This is known as **thrashing**, and causes a significant increase in misses
- Library analogy: may have unnecessary evictions if we don't have an even distribution of books
    - Not many books will start with "X", for example, so that space will likely go unused.
- Problem: How do we get the benefit of direct-mapped caches without the downsides?

Set Associative Cache

Meeting in the Middle: The Set-Associative Cache

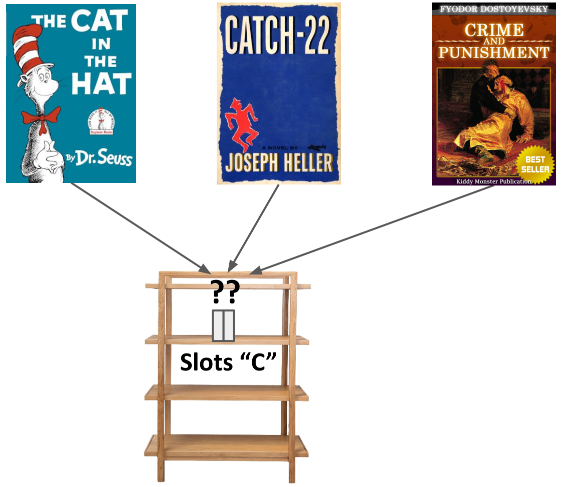
- Direct-mapped caches thrash a lot, but fully associative caches use a lot of comparators
- Solution: Instead of just one slot per index, assign a small number of slots per index
- The **associativity** of a cache is the number of slots assigned to each index.

N-way Set Associative Cache (1/3)

- An N-way set associative cache is parameterized on **three** aspects: The block size, the number of blocks that can be stored in the cache, and the associativity N.
    - The other two types were parameterized on block size and number of blocks only.
- Blocks are split into groups of N, and each group of blocks is assigned a unique index
- When a memory access occurs:
    - Step 1: Check **the slots that could contain our data**. If the tag matches and the valid bit is on for any of these slots, cache hit
    - Step 2: Otherwise, cache miss.

N-way Set Associative Cache (2/3)

- Library analogy: You buy a bookshelf. You set it up so that each shelf can only hold a certain type of book, and promise never to put books in the wrong shelf, even if it wastes space. That way, you only need to check a few blocks on a given shelf.



N-way Set Associative Cache: Example

- Say we have a 2-way set-associative cache with 4 byte blocks and 4 blocks storage
- Addresses are 10 bits
- To access address 0x3CD:
  - Split into TIO:
    - 4 bytes blocks -> O = 2 bits
    - 4/2 = 2 indexes -> I = 1 bits
    - `0x3CD == 0b111 1001 1 01`
  - For index 1, the tag matches one of our blocks, and the valid bit is on, so we get the data
  - We get the 1st byte of the block, which is `0xDE`.

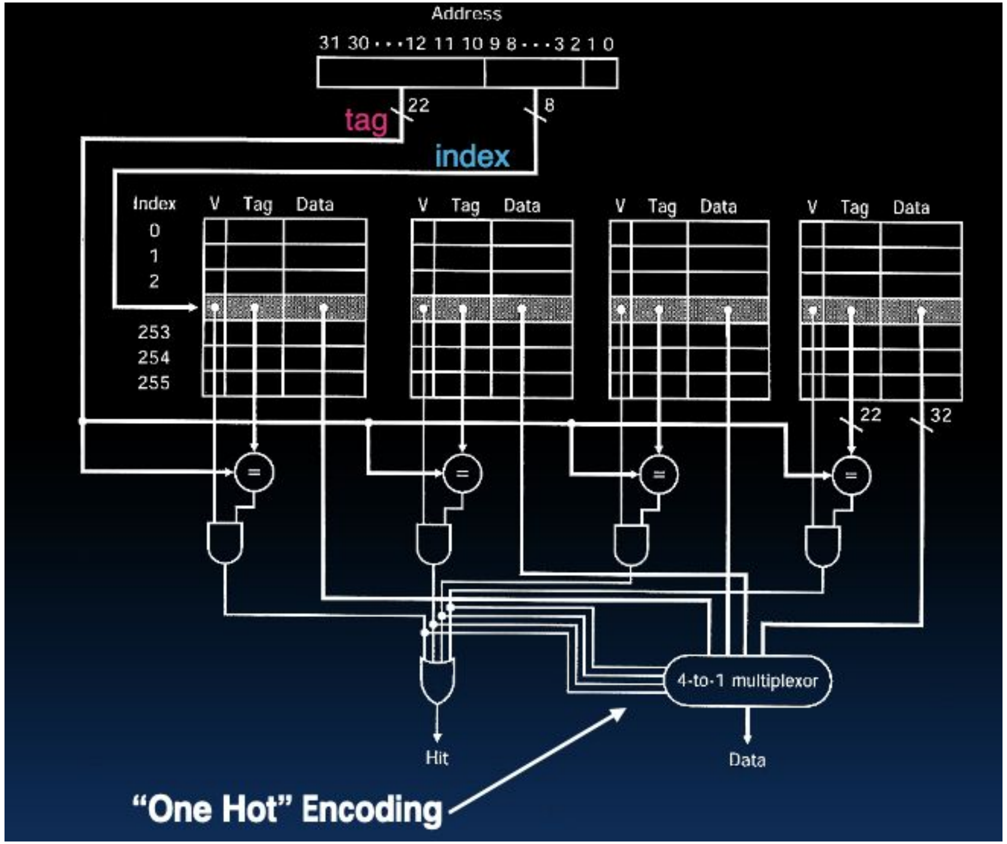
I	Tag	V	Data
0	0b100 0110	1	0xDE 0xAD 0xBE 0xEF
0	0b101 1100	0	0x01 0x23 0x45 0x67
1	0b011 1010	1	0xAB 0xAD 0xCA 0xFE
1	0b111 1001	1	0xAC 0xDE 0xFF 0x61

Calculating TIO Breakdown

- # of Offset bits = $\log_2$(block size in bytes)
- # of Index bits = $\log_2$(# of sets)
    - Direct-mapped: # of sets = # of blocks
    - Fully associative: # of sets = 1
- # of Tag bits = Address length - # of Index bits - # of Offset bits

4-way Set Associative Cache Circuit

- 1024 blocks
- 1024 / 4 = 256 sets
- 4B blocks
- 32-bit address
- TIO?
    - 22, 8, 2.



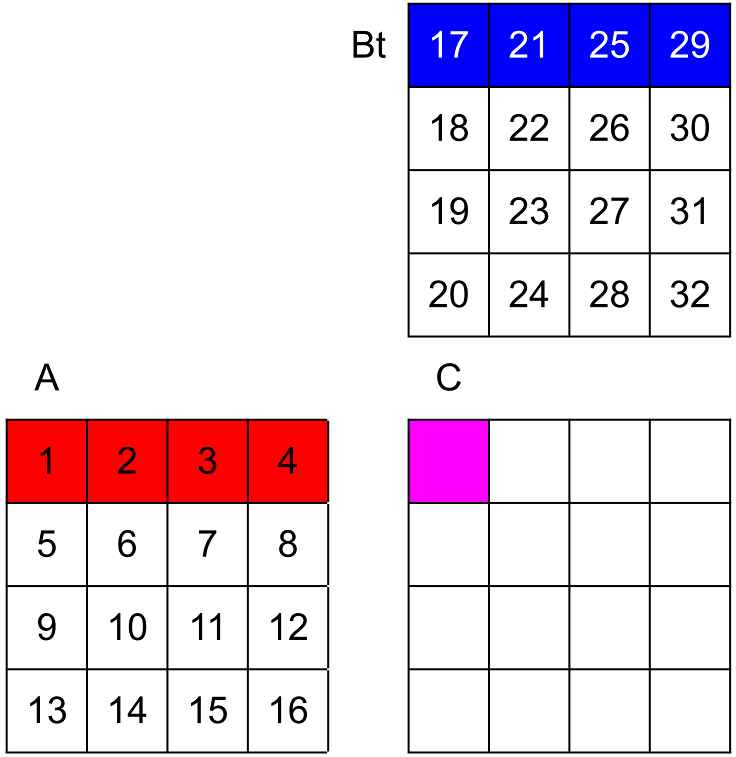
N-way Set Associative Cache (3/3)

- Direct-mapped and Fully Associative caches can be considered special cases of the N-way set associative cache:
  - Direct-mapped == 1-way set associative
  - Fully associative == N = block count-way set associative
- Generally ends up with most of the performance of a fully associative cache and most of the circuit simplicity of a direct-mapped cache
- Let's see it in action with a 2-way set associative cache with 4 blocks (2 sets).

Caching Example: Write-Back LRU 2-way Set Associative

- Start: Cache starts cold
- A[0]: Miss, B[0]: Miss (but this time no eviction)
- 2 misses, 0 hits

I	Tag	LRU	V	D	Data
0	`0x100`	0	0	0	`0xBF 0x16 0x88 0x2B`
0	`0x333`	0	0	0	`0x3B 0x18 0xF1 0xB3`
1	`0x156`	0	0	0	`0xE6 0x57 0x49 0xEE`
1	`0x0E9`	0	0	0	`0xB5 0x81 0x67 0x3F`



Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

🟪			

Caching Example: Write-Back LRU 2-way Set Associative

- A[1] Hit, B[1] Hit, … , B[3] Hit, C[0] Miss with eviction
  - Still have a bit of thrashing, but it's much less now
- Total: 6 Hits, 3 Misses
  - Same as Fully Associative cache!

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	0x080	2	1	0	0x01 0x02 0x03 0x04
0	0x100	1	1	0	0x11 0x15 0x19 0x1D
1	0x156	0	0	0	0xE6 0x57 0x49 0xEE
1	0x0E9	0	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

🟪			

Caching Example: Write-Back LRU 2-way Set Associative

- A[0] miss (evict block 0x200 with tag 0x100 and index 0), B[4] miss (index 1)
- A[1] - B[7] hit, C[1] hit
- Total for this element: 2 misses, 7 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	`0x180`	1	1	1	`0xFA 0x?? 0x?? 0x??`
0	`0x100`	2	1	0	`0x11 0x15 0x19 0x1D`
1	`0x156`	0	0	0	`0xE6 0x57 0x49 0xEE`
1	`0x0E9`	0	0	0	`0xB5 0x81 0x67 0x3F`

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250			

Caching Example: Write-Back LRU 2-way Set Associative

- Computing C[2]: A[0] hits, B[8] misses, evicting block 0x300 … , C[2] misses
- Total for this element: 2 misses, 7 hits
- C[3]: Same as C[1], so 2 misses, 7 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	`0x180`	1	1	1	`0xFA 0x04 0x?? 0x??`
0	`0x080`	2	1	0	`0x01 0x02 0x03 0x04`
1	`0x100`	1	1	0	`0x12 0x16 0x1A 0x1E`
1	`0x0E9`	0	0	0	`0xB5 0x81 0x67 0x3F`

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	🟪	

Caching Example: Write-Back LRU 2-way Set Associative

- C[4]: Similar to C[0], except this time, the A and C blocks are in index 1
  - 3 misses, 6 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	0x080	2	1	0	0x01 0x02 0x03 0x04
0	0x180	1	1	1	0xFA 0x04 0x0E 0x18
1	0x100	2	1	0	0x12 0x16 0x1A 0x1E
1	0x101	1	1	0	0x14 0x18 0x1C 0x20

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
🟪			

Caching Example: Write-Back LRU 2-way Set Associative

- C[5]: Acts like C[2] (B block on same index as A and C block)
  - 2 misses, 7 hits
- C[6],C[7]: 2 misses, 7 hits each

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	0x100	1	1	0	0x11 0x15 0x19 0x1D
0	0x180	2	1	1	0xFA 0x04 0x0E 0x18
1	0x100	2	1	0	0x12 0x16 0x1A 0x1E
1	0x181	1	1	1	0x6A 0x?? 0x?? 0x??

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618			

Caching Example: Write-Back LRU 2-way Set Associative

- C[8]-C[11]: Same as first row
- C[12]-C[15]: Same as second row
- Total: 36 misses, 108 hits
- 75% hit rate; not as good as FA, but much better than DM!

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

I	Tag	LRU	V	D	Data
0	0x100	2	1	0	0x11 0x15 0x19 0x1D
0	0x101	1	1	0	0x13 0x17 0x1B 0x1F
1	0x081	2	1	0	0x05 0x06 0x07 0x08
1	0x181	1	1	1	0x6A 0x84 0x9E 0xB8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696
986	1028	1070	1112
1354	1412	1470	1528

Types of Cache Misses

Miss Classifications: Definitions in this class

- **Compulsory Misses**: Misses that would occur even if the cache was a fully associative cache with infinite space
- **Capacity Misses**: Misses that are neither conflict nor compulsory
- **Conflict Misses**: Misses that would not have happened if the cache was fully associative under at least one consistent replacement policy
- **Consistent replacement policy**: A replacement policy that keeps all the data the lower-associativity cache would
  - Any replacement policy such that the data in the Fully Associative cache is a superset of the data in our actual cache.
  - This definition guarantees that we don't have "conflict hits", but also tends to classify misses more as capacity misses than as conflict misses.

Miss Classifications: Simplified Equivalent Definitions

- If a miss occurs, look at the most recent time that that block was evicted
- If that block was never evicted before, it's a **compulsory miss**
- If the block was evicted when the cache was full (no empty spaces), it's a **capacity miss**
- If the block was evicted when the cache was not full (at least one empty space), it's a **conflict miss**
- Let's look again at the 2-way set associative cache example.

Caching Example: Hit Classifications

- Start: Cache starts cold
  - We'll restrict the cache to only the parts that are important (no data/dirty/etc, block instead of tag).
  - Also will write the status of each block on the right
  - Blocks will be ordered by LRU within a set (unlike an actual cache)
- First miss: A[0]
  - Compulsory miss

Block	Status	Block	Status
A[0]	NA	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	NA
A[12]	NA	C[4]	NA
B[0]	NA	C[8]	NA
B[4]	NA	C[12]	NA

Index	Block	Index	Block
0	–	1	–
0	–	1	–

Caching Example: Hit Classifications

- Next miss is B[0]
  - Compulsory Miss
- Next miss is C[0]
  - Compulsory Miss, evicts A[0]
  - A[0] was evicted even though there were empty slots, so the next access of A[0] will be a conflict miss

Block	Status	Block	Status
A[0]	In Cache	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	NA
A[12]	NA	C[4]	NA
B[0]	NA	C[8]	NA
B[4]	NA	C[12]	NA

Index	Block	Index	Block
0	A[0]	1	-
0	-	1	-

Caching Example: Hit Classifications

- A[0]
  - Conflict miss
  - Evicts B[0] (will cause conflict miss)
- B[4]
  - Compulsory miss
- B[8]
  - Compulsory miss
  - Evicts C[0] (conflict)

Index	Block	Index	Block
0	`B[0]`	1	–
0	`C[0]`	1	–

Block	Status	Block	Status
A[0]	Conflict	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	In Cache
A[12]	NA	C[4]	NA
B[0]	In Cache	C[8]	NA
B[4]	NA	C[12]	NA

Caching Example: Hit Classifications

- C[2]
  - Conflict miss
  - Evicts A[0] (conflict)
- A[0]
  - Conflict miss
  - Evicts B[8] (conflict)
- B[12]
  - Compulsory miss
  - At this point, all blocks are used, so any further evictions will cause capacity misses

Block	Status	Block	Status
A[0]	In Cache	B[8]	In Cache
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	Conflict
A[12]	NA	C[4]	NA
B[0]	Conflict	C[8]	NA
B[4]	In Cache	C[12]	NA

Index	Block	Index	Block
0	A[0]	1	B[4]
0	B[8]	1	-

Caching Example: Hit Classifications

- A[4]
  - Compulsory miss
  - Evicts B[4] (will cause a capacity miss next)
- B[0]
  - Conflict miss
  - Evicts C[0] (capacity)
- C[4]
  - Compulsory miss
  - Evicts B[12] (capacity)

Block	Status	Block	Status
A[0]	In Cache	B[8]	Conflict
A[4]	NA	B[12]	In Cache
A[8]	NA	C[0]	In Cache
A[12]	NA	C[4]	NA
B[0]	Conflict	C[8]	NA
B[4]	In Cache	C[12]	NA

Index	Block	Index	Block
0	`C[0]`	1	`B[4]`
0	`A[0]`	1	`B[12]`

Caching Example: Hit Classifications

- B[4]
  - Capacity miss
  - Evicts A[4] (capacity)
- A[5]
  - Capacity miss
  - Evicts C[4] (capacity)
- C[5]
  - Capacity miss
  - Evicts B[4] (capacity)
  - This is the downside of this definition (this feels like it should be a conflict miss)

Block	Status	Block	Status
A[0]	In Cache	B[8]	Conflict
A[4]	In Cache	B[12]	Capacity
A[8]	NA	C[0]	Capacity
A[12]	NA	C[4]	In Cache
B[0]	In Cache	C[8]	NA
B[4]	Capacity	C[12]	NA

Index	Block	Index	Block
0	A[0]	1	A[4]
0	B[0]	1	C[4]

Summary

- **Direct-mapped caches** allow for a simpler implementation of caches, but may decrease cache hit rate
- **Set associative caches** are a middle ground between fully associative and direct-mapped caches
- We classify cache misses into **compulsory, capacity, and conflict misses** to analyze performance more easily (tomorrow)
- Tomorrow: Analyzing cache performance, and advanced cache techniques!