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

Lecture 31: Caches Part 3

Instructors: Dan Garcia, Justin Yokota

Computing in the News

Computer Made of DNA Works Out Prime Factors of 6 and 15

Filling a test tube with molecules made from folded DNA can work as a simple computer. The approach has been used to split two numbers into prime factors.

Conventional computers work by passing electricity through tiny on-off switches to perform simple calculations. However, the new computer relies on the way that differently shaped DNA molecules combine.

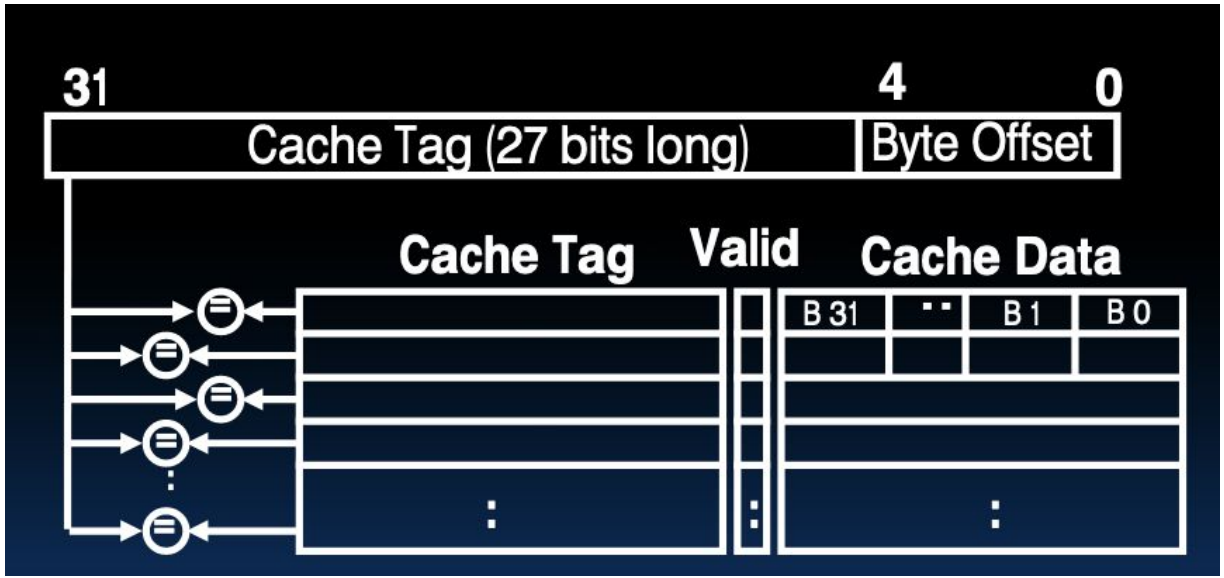
Based on typical DNA functional motifs, DNA computational devices can perform diverse powerful computational functions, such as simple Boolean logics and sophisticated neural network algorithms. Thus, DNA computer is widely regarded as one of the most excellent next-generation molecular computers performing Boolean logic. Benefiting from DNAs' inherent properties of biocompatibility, low-cost, ease of synthesis, and sequence programmability, DNA computational devices have shown great potential in various biosensing applications.

Agenda

- Direct-Mapped Caches
- N-way Set Associative Caches
-

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 eviction 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 eviction policy
  - Requires a lot of comparators/circuitry
- Ultimately, this stems from the fact that 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
- In practice, this means we're limited in our maximum cache size by hit time
- Solutions?

Problems with Fully-Associative Caches

- Attempt #1: Sort the data in the cache, so we can check in log time.
  - Works if you have a bookshelf at home.
  - Doesn't work in caches.
  - Takes time to move data between blocks, so we don't want to have to do that (it's the same reason why we can't sort by LRU)
  - All the comparators still need to exist, and they run in constant time anyway (due to hardware parallelism)
- Attempt #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.
  - May cause 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

How should we "group" our cache blocks into (for example) 8 groups?

- Hash the tag with some hashing algorithm
- Take the top three bits of the tag
- Take the bottom three bits of the tag
- Hash the data in the block with some hashing algorithm
- Take the first three bits of the data in the block
- Take the last three bits of the data in the block

How should we "group" our cache blocks into (for example) 8 groups?

- Hash the tag with some hashing algorithm
- Take the top three bits of the tag
- Take the bottom three bits of the tag
- Hash the data in the block with some hashing algorithm
- Take the first three bits of the data in the block
- Take the last three bits of the data in the block

How should we "group" our cache blocks into (for example) 8 groups?

Hash the tag with some hashing algorithm

Take the top three bits of the tag

Take the bottom three bits of the tag

Hash the data in the block with some hashing algorithm

Take the first three bits of the data in the block

Take the last three bits of the data in the block

Attempt 2: Picking a grouping mechanism

- No option that uses the data in the block will work
  - We need to get the block first to get the data, so we need to be able to figure out where the block is from the memory address alone
- Top three bits of the tag: 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 three bits of memory address
- Hashing is a nice "all-purpose" approach to distributing data into even groups, but it's not as good here
  - Adds extra computation time
  - Distributes blocks randomly, whereas we can get exactly even distribution
- Bottom three bits of the tag: Works well with consecutive blocks of memory
  - Also scales to arbitrarily many cache slots without much additional circuitry

Attempt 2: Picking a grouping mechanism

- No option that uses the data in the block will work
  - We need to get the block first to get the data, so we need to be able to figure out where the block is from the memory address alone
- Top three bits of the tag: 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 three bits of memory address
- Hashing is a nice "all-purpose" approach to distributing data into even groups, but it's not as good here
  - Adds extra computation time
  - Distributes blocks randomly, whereas we can get exactly even distribution
- Bottom three bits of the tag: Works well with consecutive blocks of memory
  - Also scales to arbitrarily many cache slots without much additional circuitry
  - Note: Assumes that we'll always have our number of groups be a power of 2.

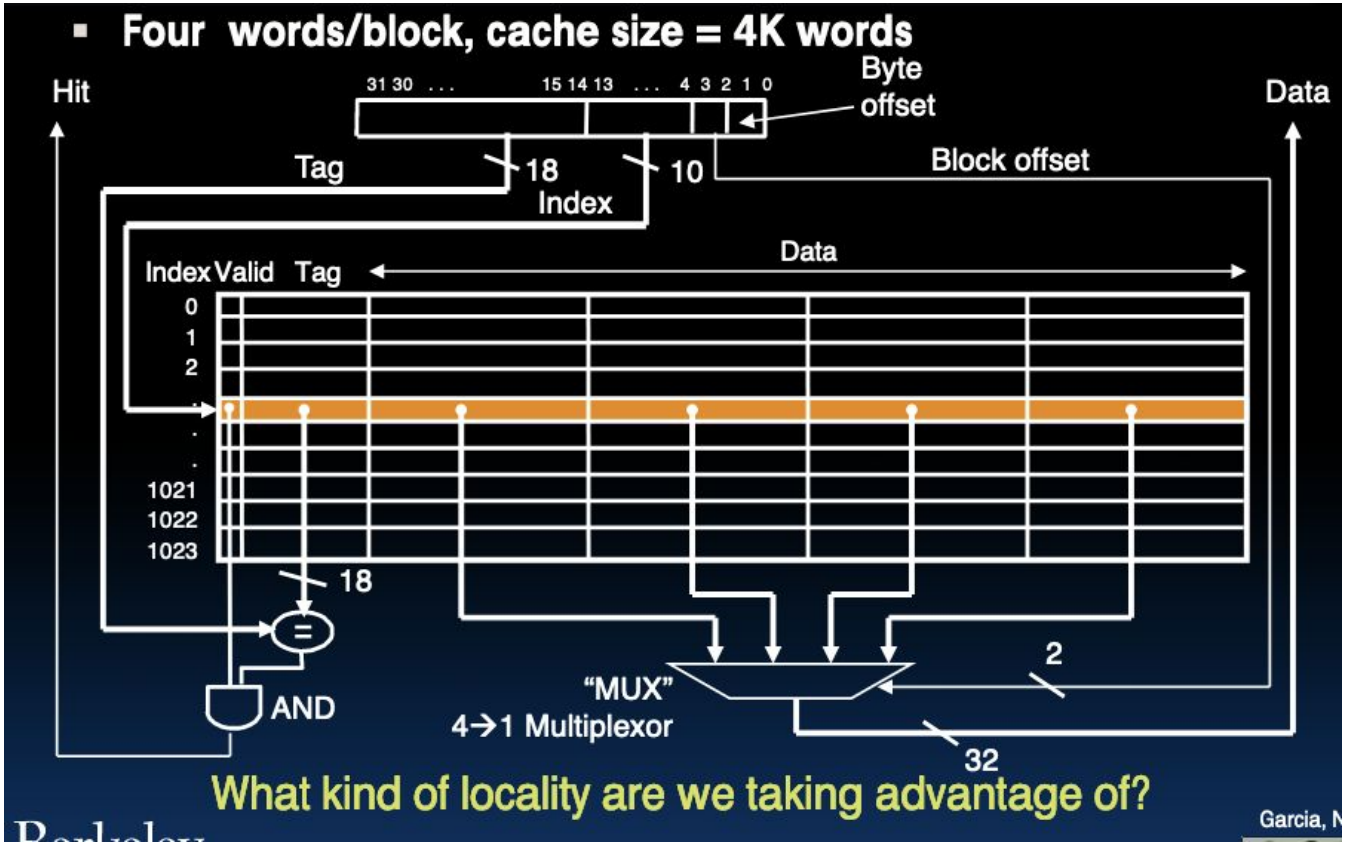
New Blocks

- Bottom three 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
- The bottom bits of our address is our offset, the next few bits the index, and the top bits will be the tag
- Ex. Let's say we have a block size of 4096 and 16 distinct groups
  - Ex. **`0xDEADBEEF`**
  - This address is in Block `0xDEADB`, which is the unique block with tag `0xDEAD` and index `0xB`, and is 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 (1/4)

- A direct-mapped cache is parametrized on two aspects: The block size, and the number of blocks that can be stored in the 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
- Library analogy: You buy a bookshelf that can hold N books. You set it up so that each slot can only hold a certain type of book, and promise never to put books in the wrong spot, even if it wastes space. That way, you only need to check one slot of your bookshelf when cache hitting.

Direct-Mapped Cache (2/4)



Direct-Mapped Cache (3/4): Example Memory Access

- Let's say we have a direct-mapped cache with 4 byte blocks and 4 blocks storage, and let's assume that we're working with a system that uses 10-bit addresses
- 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, the tag matches, and the valid bit is on, so we get the data
  - 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 (3/4): Example Memory Access

- Let's say we have a direct-mapped cache with 4 byte blocks and 4 blocks storage, and let's assume that we're working with a system that uses 10-bit addresses
- 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, the tag doesn't match, so it's a miss, even though the correct tag is in another index (same tag + different index is still 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 (4/4)

- Pros:
- Much simpler circuit
  - Instead of comparing everything, we mux all the indexes and use one comparator. Comparators use a lot of circuitry, so this is much smaller
  - Therefore leads to faster accesses
  - Also scales better than fully associative caches
- No need for an eviction 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

- 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

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 A[0]: 0x1000
  - Index 0, Tag 0x040 (taking the bottom two 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

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

- 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

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

- A[1]: Hit, B[5]: Hit, … , C[1]: Miss, evicting Block 0x1000
- Overall for this element of C: 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

A

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

Bt

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

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 are aligned, we end up continuously evicting blocks
  - This is known as **thrashing**, and causes a significant increase in misses
- Even without this, we have a number of unused slots in our cache until we start computing C[3], so until then, it's as if we had a smaller cache; we aren't using the cache to its full capacity
- Problem: How do we get the benefit of direct-mapped caches without the downsides?

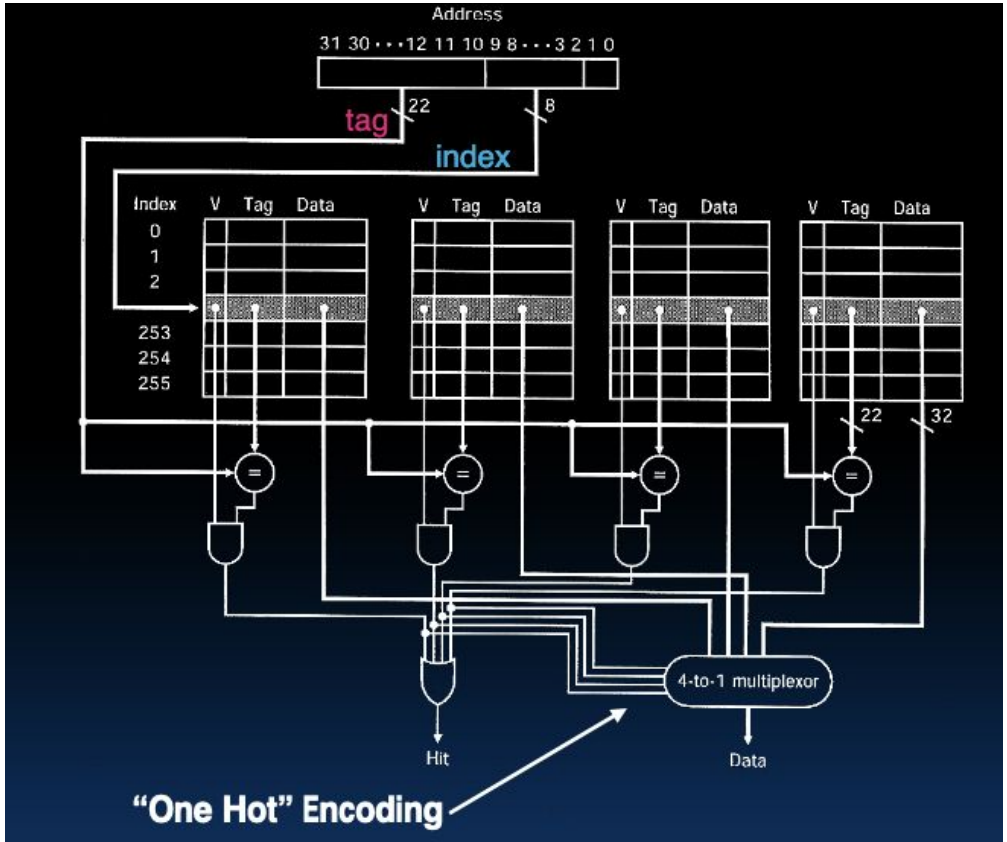
Meeting in the Middle: The Set-Associative Cache

- 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 (1/4)

- An N-way set associative cache is parametrized on three aspects: The block size, the number of blocks that can be stored in the cache, and the associativity N.
- 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
- 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.

4-way Set Associative Cache (2/4)



N-way Set Associative Cache (3/4): Example

- Let's say we have a 2-way set-associative cache with 4 byte blocks and 4 blocks storage, and let's assume that we're working with a system that uses 10-bit addresses
- 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

N-way Set Associative Cache (4/4)

- 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 2 blocks

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!

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

Bt

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

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

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

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

- 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

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

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

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	260		

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

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

Conclusion

- Increasing associativity has pros and cons
- Pro:
  - Reduces thrashing (effect diminishes after a small associativity increase)
  - Increases cache usage
- Con:
  - Increases circuitry required
  - Forces a smaller cache, so the increased cache usage eventually gets countered by this anyway
- Goal is to get a good middle-ground between cache size and cache associativity
- How do we measure the effectiveness of our cache (relative to other cache types)?
  - Monday's lecture: Classifying cache misses according to how we could have avoided them