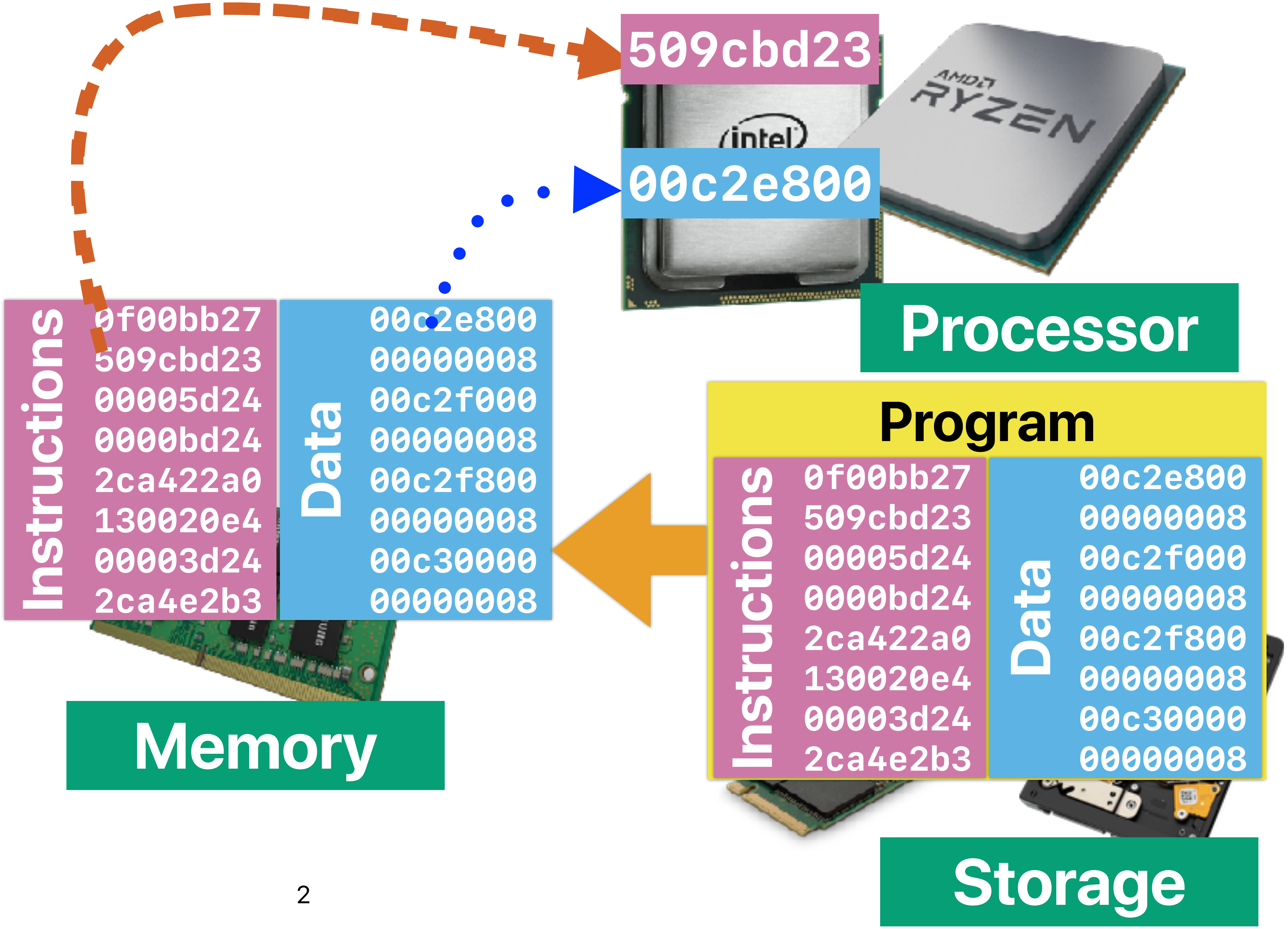
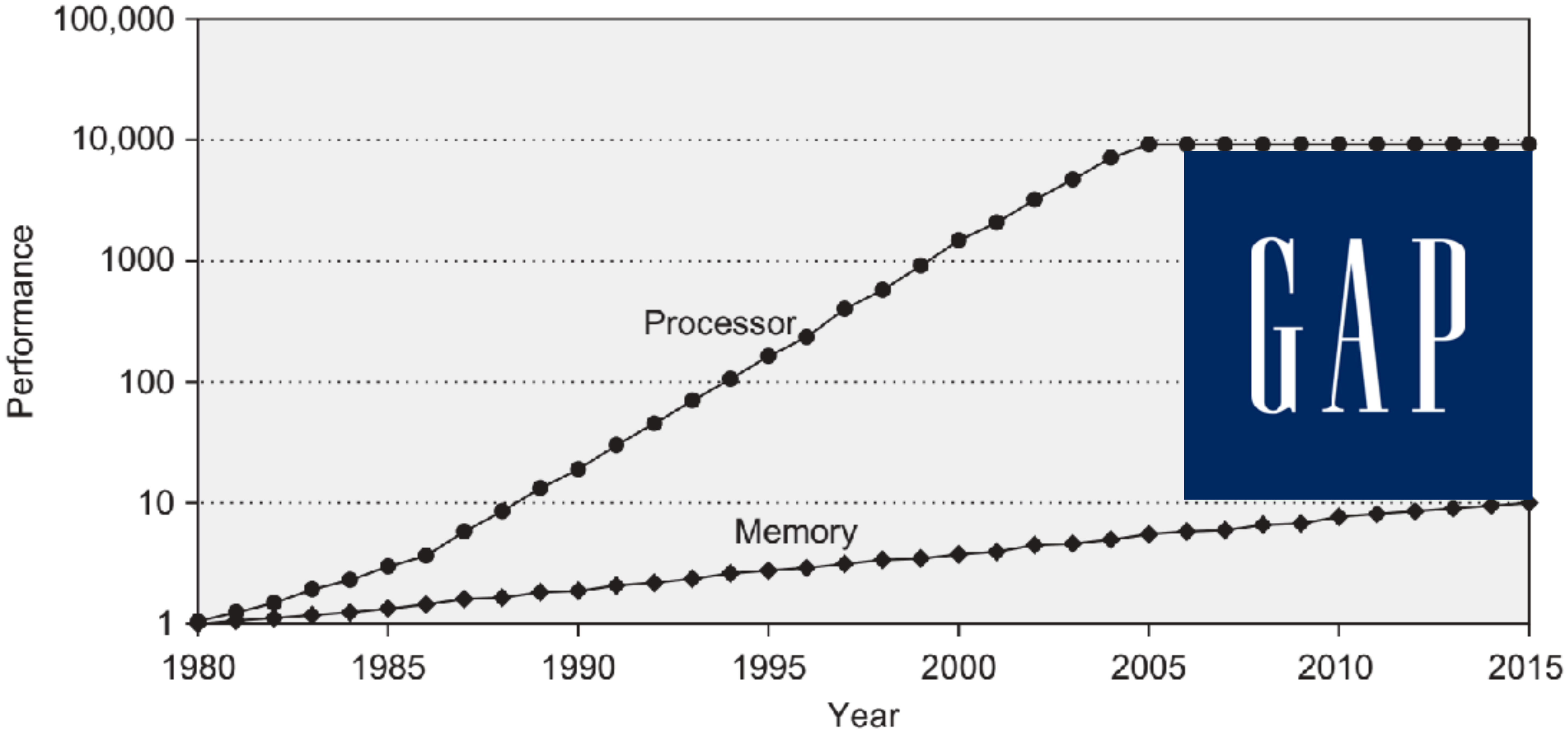
# Memory Hierarchy (3): Cache Misses and How to Address Them

Hung-Wei Tseng

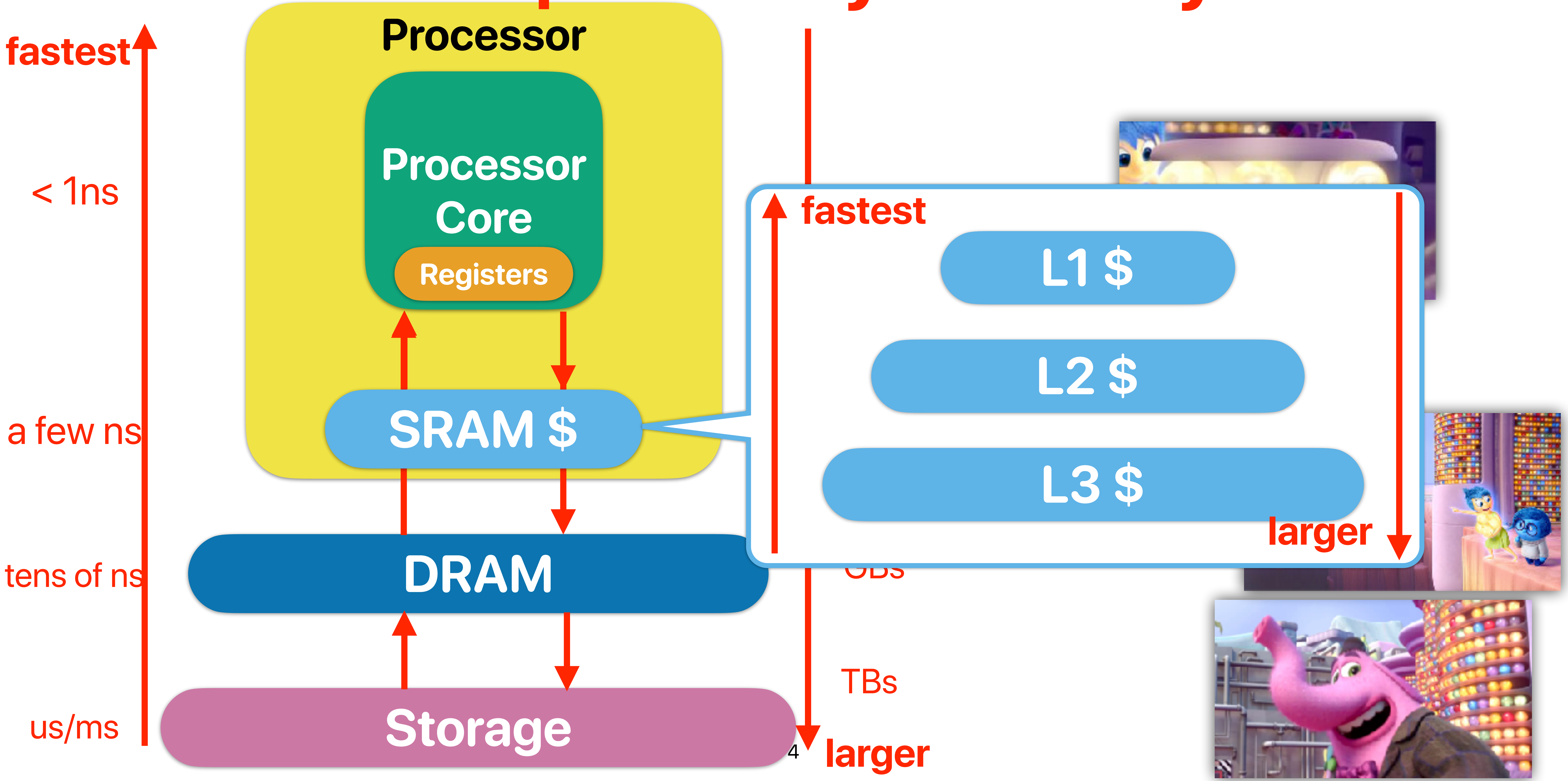
# von Neumman Architecture



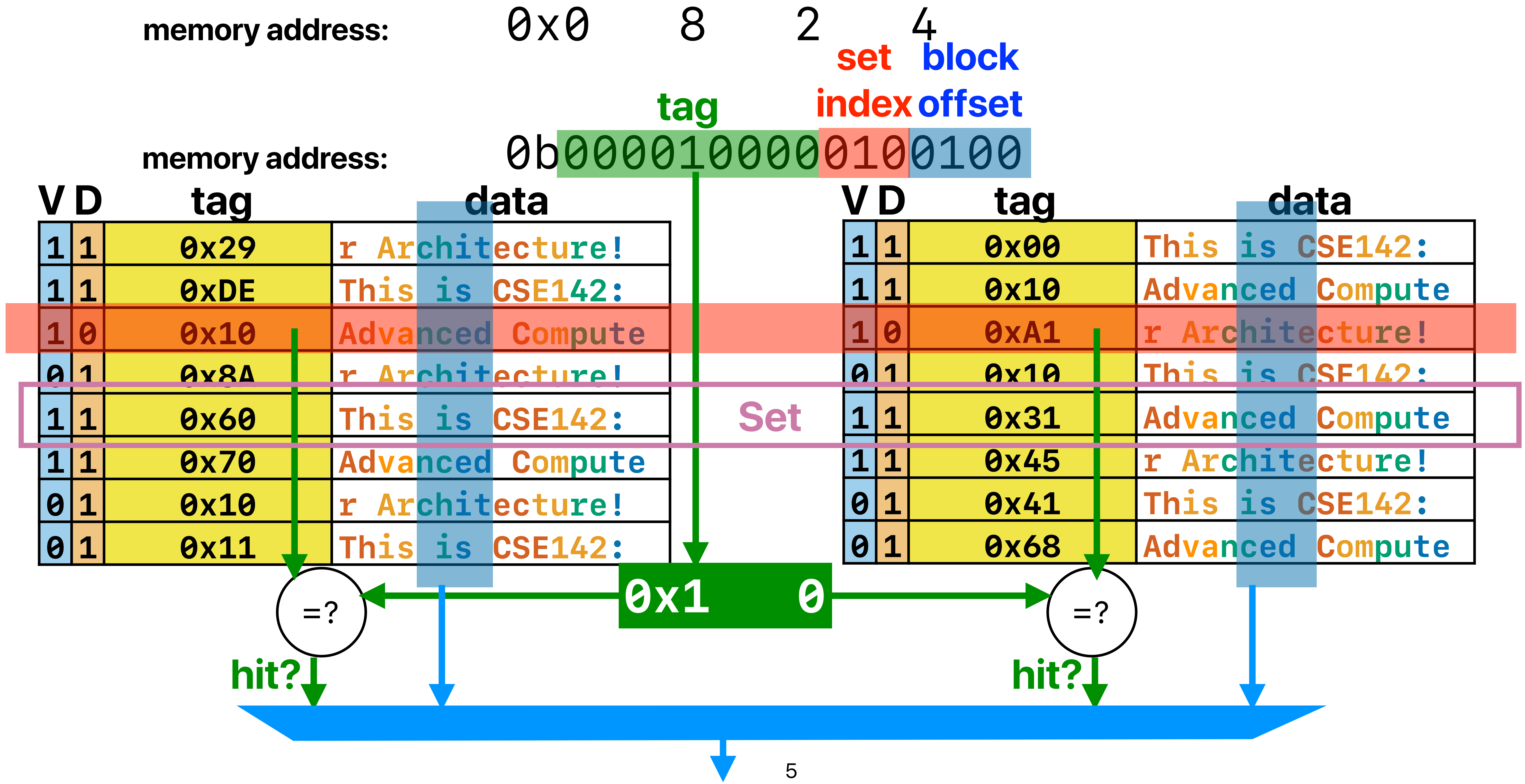
# Recap: Performance gap between Processor/Memory



# Recap: Memory Hierarchy



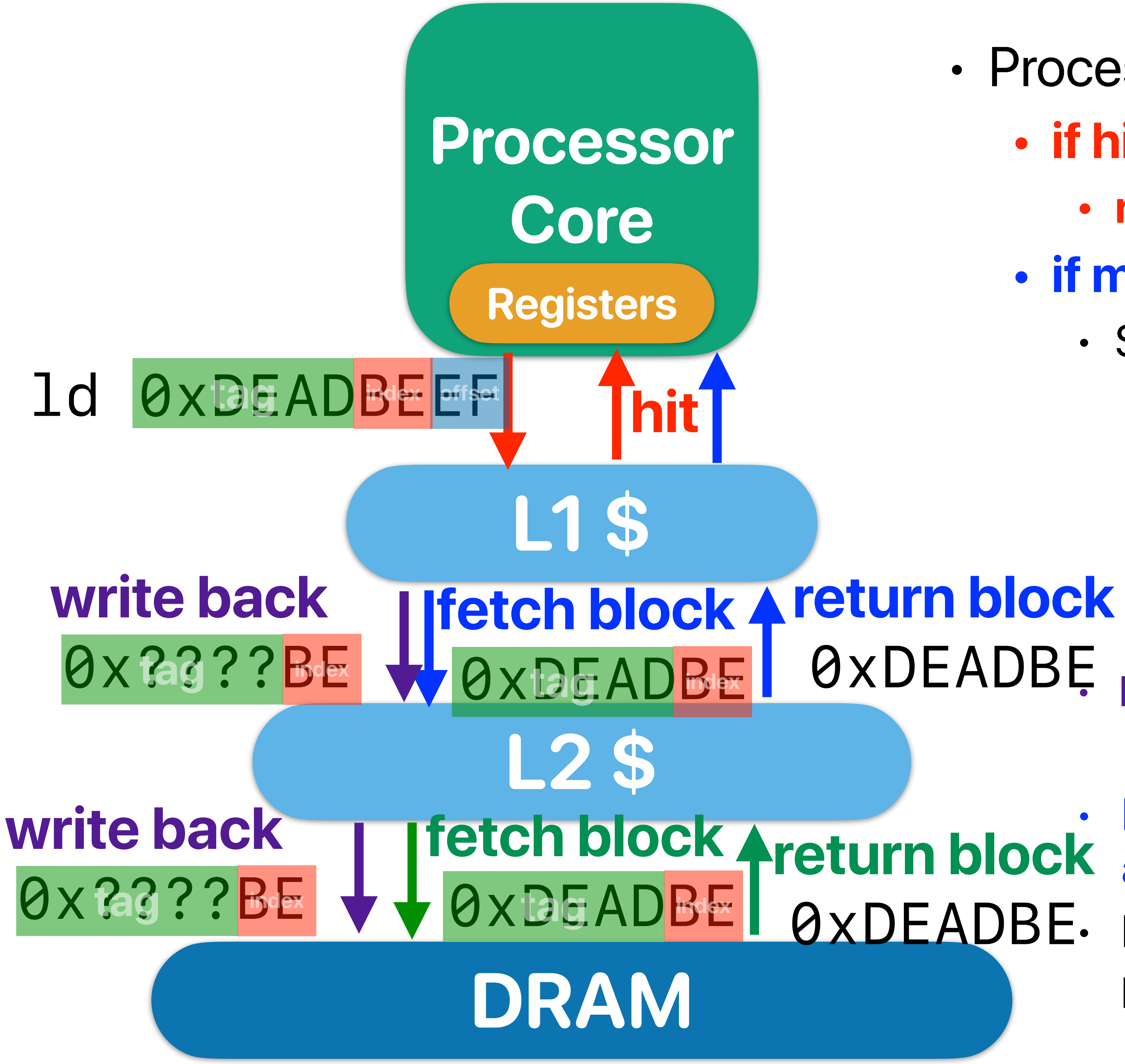
# Way-associative cache



# Recap: C = ABS

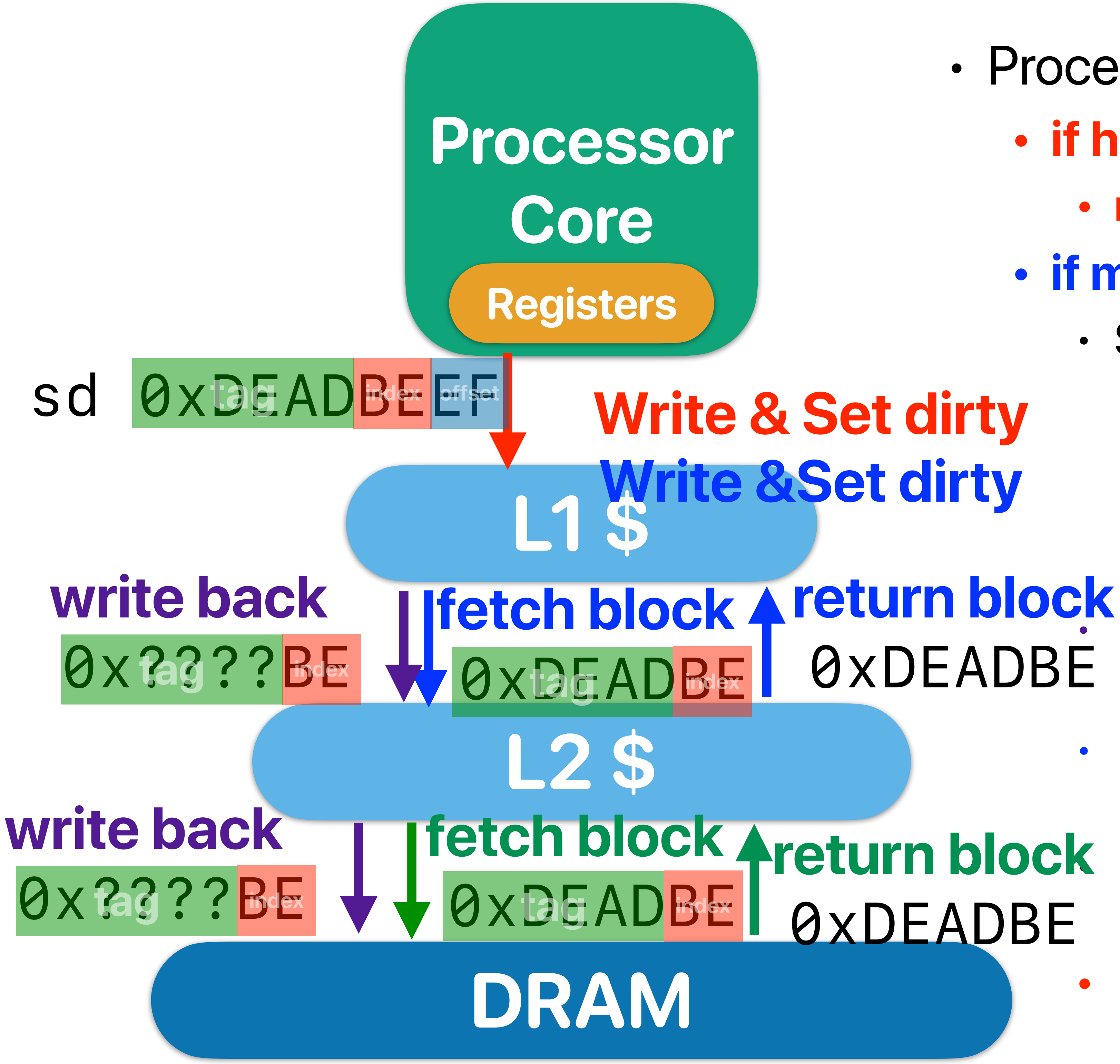
- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Cacheline)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

# What happens when we read data




- Processor sends load request to L1-$
  - if hit
    - **return data**
  - if miss
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process

# What happens when we write data




- Processor sends load request to L1-$
  - **if hit**
    - **return data — set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
    - **Present the write "ONLY" in L1 and set DIRTY**

# NVIDIA Tegra X1

**100% miss rate!**

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS

32KB = 4 * 64 * S

S = 128

offset = lg(64) = 6 bits

index = lg(128) = 7 bits

tag = the rest bits

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? |
|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b0001000000000000000000 | 0x8 | 0x0 | Miss | |
| b[0] | 0x20000 | 0b0010000000000000000000 | 0x10 | 0x0 | Miss | |
| c[0] | 0x30000 | 0b0011000000000000000000 | 0x18 | 0x0 | Miss | |
| d[0] | 0x40000 | 0b0100000000000000000000 | 0x20 | 0x0 | Miss | |
| e[0] | 0x50000 | 0b0101000000000000000000 | 0x28 | 0x0 | Miss | a[0-7] |
| a[1] | 0x10008 | 0b0001000000000000001000 | 0x8 | 0x0 | Miss | b[0-7] |
| b[1] | 0x20008 | 0b0010000000000000001000 | 0x10 | 0x0 | Miss | c[0-7] |
| c[1] | 0x30008 | 0b0011000000000000001000 | 0x18 | 0x0 | Miss | d[0-7] |
| d[1] | 0x40008 | 0b0100000000000000001000 | 0x20 | 0x0 | Miss | e[0-7] |
| e[1] | 0x50008 | 0b0101000000000000001000 | 0x28 | 0x0 | Miss | a[0-7] |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

# Outline

- Taxonomy/reasons of cache misses
- How to address cache misses: the software perspective

# Taxonomy/reasons of cache misses

# 3Cs of misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash

https://www.pollev.com/hungweitseng close in 1:30

# NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **conflict** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

E. 100%

# NVIDIA Tegra X1

**100% miss rate!**

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS

32KB = 4 * 64 * S

S = 128

**offset = lg(64) = 6 bits**

**index = lg(128) = 7 bits**

**tag = the rest bits**

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? |
|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b00010000000000000000 | 0x8 | 0x0 | Compulsory Miss | |
| b[0] | 0x20000 | 0b00100000000000000000 | 0x10 | 0x0 | Compulsory Miss | |
| c[0] | 0x30000 | 0b00110000000000000000 | 0x18 | 0x0 | Compulsory Miss | |
| d[0] | 0x40000 | 0b01000000000000000000 | 0x20 | 0x0 | Compulsory Miss | |
| e[0] | 0x50000 | 0b01010000000000000000 | 0x28 | 0x0 | Compulsory Miss | a[0-7] |
| a[1] | 0x10008 | 0b00010000000000001000 | 0x8 | 0x0 | **Conflict Miss** | b[0-7] |
| b[1] | 0x20008 | 0b00100000000000001000 | 0x10 | 0x0 | **Conflict Miss** | c[0-7] |
| c[1] | 0x30008 | 0b00110000000000001000 | 0x18 | 0x0 | **Conflict Miss** | d[0-7] |
| d[1] | 0x40008 | 0b01000000000000001000 | 0x20 | 0x0 | **Conflict Miss** | e[0-7] |
| e[1] | 0x50008 | 0b01010000000000001000 | 0x28 | 0x0 | **Conflict Miss** | a[0-7] |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

https://www.pollev.com/hungweitseng close in 1:30

# intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

E. 100%

# intel Core i7

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 8 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? |
|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b000100000000000000000 | 0x10 | 0x0 | Compulsory Miss | |
| b[0] | 0x20000 | 0b001000000000000000000 | 0x20 | 0x0 | Compulsory Miss | |
| c[0] | 0x30000 | 0b001100000000000000000 | 0x30 | 0x0 | Compulsory Miss | |
| d[0] | 0x40000 | 0b010000000000000000000 | 0x40 | 0x0 | Compulsory Miss | |
| e[0] | 0x50000 | 0b010100000000000000000 | 0x50 | 0x0 | Compulsory Miss | |
| a[1] | 0x10008 | 0b000100000000000001000 | 0x10 | 0x0 | Hit | |
| b[1] | 0x20008 | 0b001000000000000001000 | 0x20 | 0x0 | Hit | |
| c[1] | 0x30008 | 0b001100000000000001000 | 0x30 | 0x0 | Hit | |
| d[1] | 0x40008 | 0b010000000000000001000 | 0x40 | 0x0 | Hit | |
| e[1] | 0x50008 | 0b010100000000000001000 | 0x50 | 0x0 | Hit | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

# intel Core i7 (cont.)

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 8 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? |
|---|---|---|---|---|---|---|
| a[7] | 0x10038 | 0b00010000000000111000 | 0x10 | 0x0 | Hit | |
| b[7] | 0x20038 | 0b00100000000000111000 | 0x20 | 0x0 | Hit | |
| c[7] | 0x30038 | 0b00110000000000111000 | 0x30 | 0x0 | Hit | |
| d[7] | 0x40038 | 0b01000000000000111000 | 0x40 | 0x0 | Hit | |
| e[7] | 0x50038 | 0b01010000000000111000 | 0x50 | 0x0 | Hit | |
| a[8] | 0x10040 | 0b00010000000001000000 | 0x10 | 0x1 | Compulsory Miss | |
| b[8] | 0x20040 | 0b00100000000001000000 | 0x20 | 0x1 | Compulsory Miss | |
| c[8] | 0x30040 | 0b00110000000001000000 | 0x30 | 0x1 | Compulsory Miss | |
| d[8] | 0x40040 | 0b01000000000001000000 | 0x40 | 0x1 | Compulsory Miss | |
| e[8] | 0x50040 | 0b01010000000001000000 | 0x50 | 0x1 | Compulsory Miss | |
| a[9] | 0x10048 | 0b00010000000001001000 | 0x10 | 0x1 | Hit | |
| b[9] | 0x20048 | 0b00100000000001001000 | 0x20 | 0x1 | Hit | |
| c[9] | 0x30048 | 0b00110000000001001000 | 0x30 | 0x1 | Hit | |
| d[9] | 0x40048 | 0b01000000000001001000 | 0x40 | 0x1 | Hit | |

# intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%

B. 66.67%

C. 68.75%

D. 87.5%

E. 100%

# How can programmers improve memory performance?

# Loop interchange/fission/fusion

# Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

| A | | B |
|---|---|---|
| $O(n^2)$ | Complexity | $O(n^2)$ |
| Same | Instruction Count? | Same |
| Same | Clock Rate | Same |
| Better | CPI | Worse |

# NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

A. 12.5%

B. 56.25%

C. 66.67%

D. 68.75%

E. 100%

https://www.pollev.com/hungweitseng close in 1:30

# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

A. ~10%

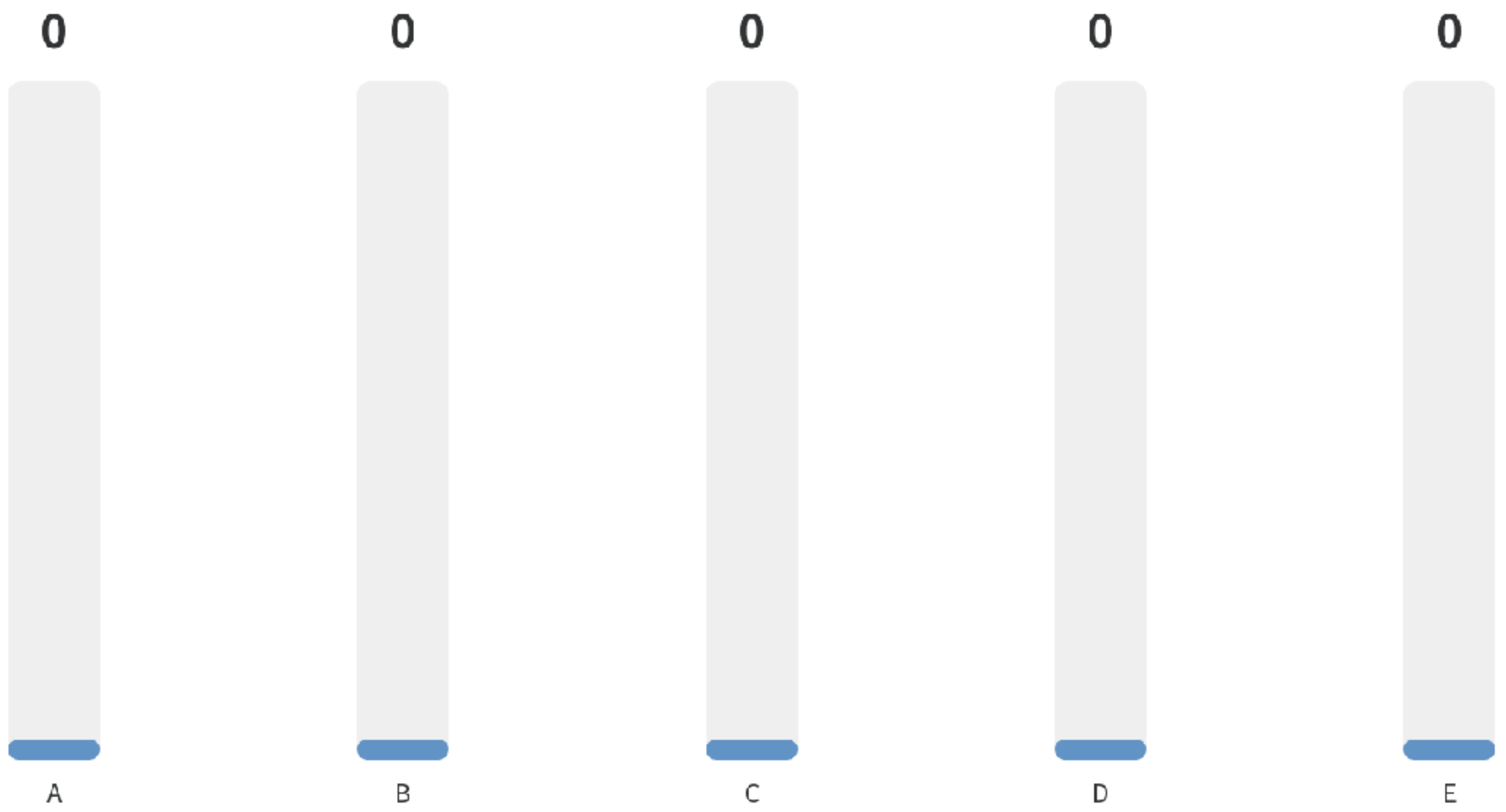
B. ~20%

C. ~40%

D. ~80%

E. 100%

0



https://www.pollev.com/hungweitseng close in 1:30

# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

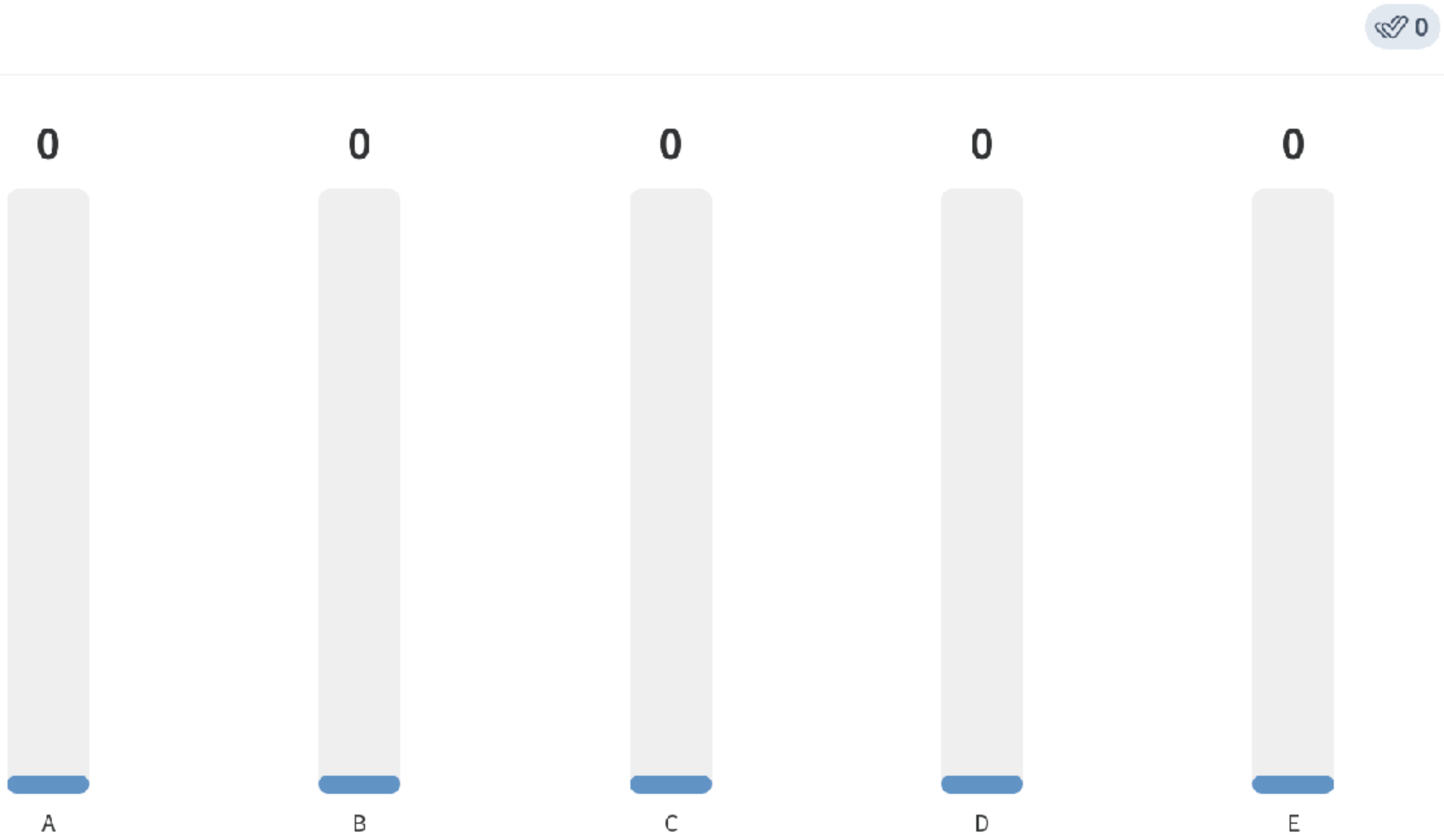
A. ~10%

B. ~20%

C. ~40%

D. ~80%

E. 100%


0
0
0
0
0
0
A
B
C
D
E

# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

A. ~10%

B. ~20%

C. ~40%

D. ~80%

E. 100%

# Loop fission

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

## Loop fission

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

# What if we change the processor?

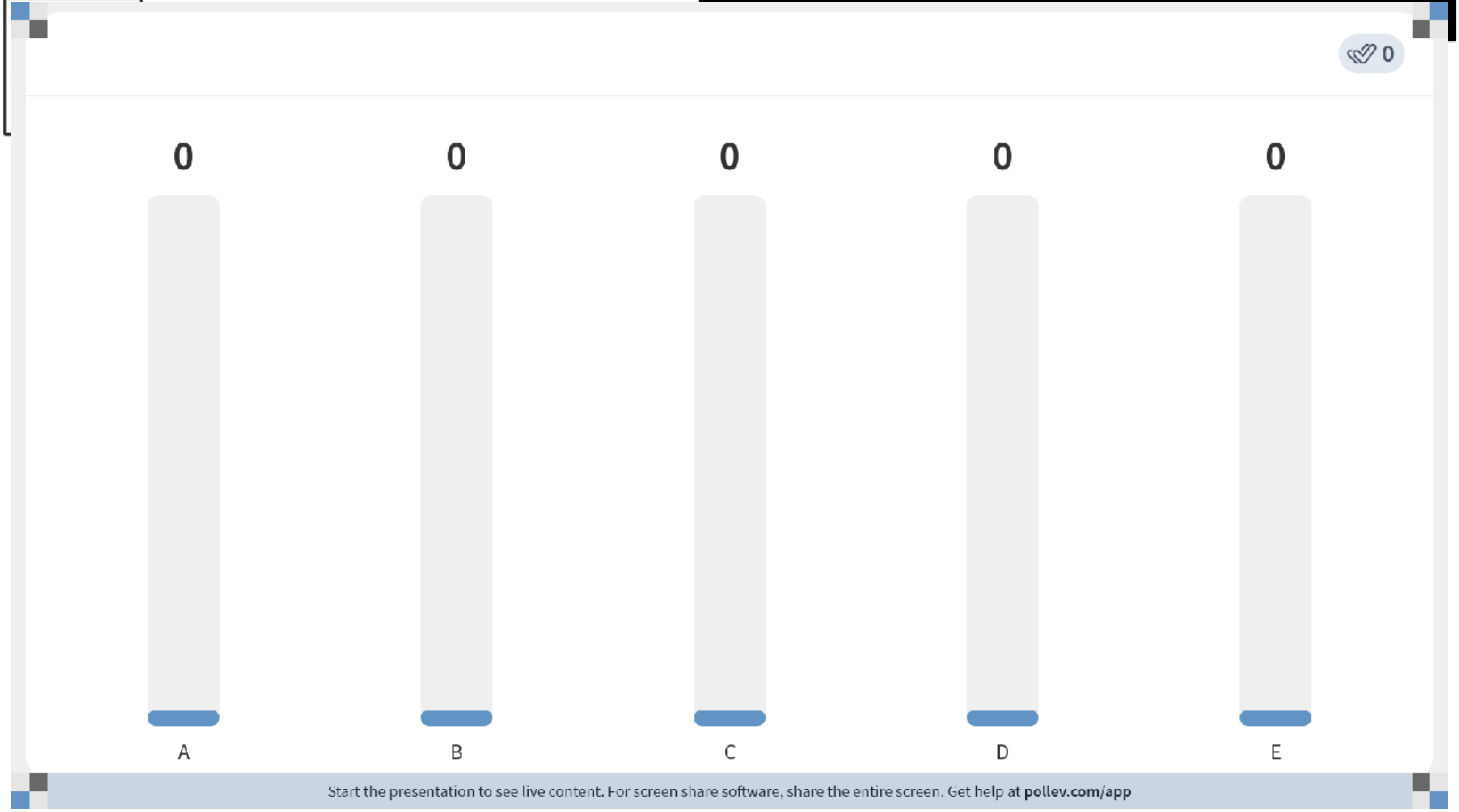
- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```


0
0
0
0
0
0
A
B
C
D
E
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

https://www.pollev.com/hungweitseng close in 1:30

# What if we change the processor?

- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

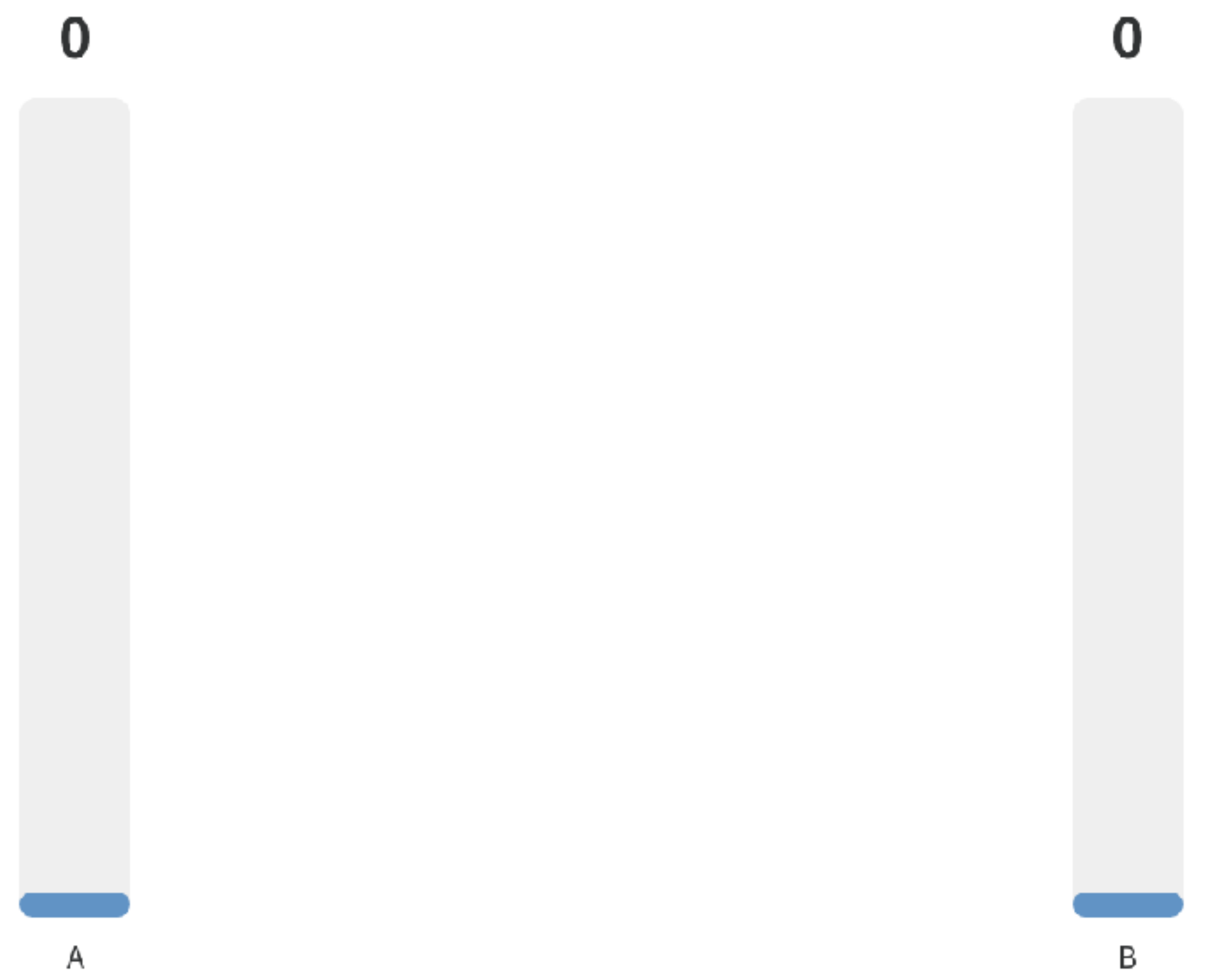
A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

0




Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

# What if we change the processor?

- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

# Loop optimizations

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

## Loop fission

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

## Loop fusion

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

# Blocking

# Case study: Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

**Algorithm class tells you it's $O(n^3)$**

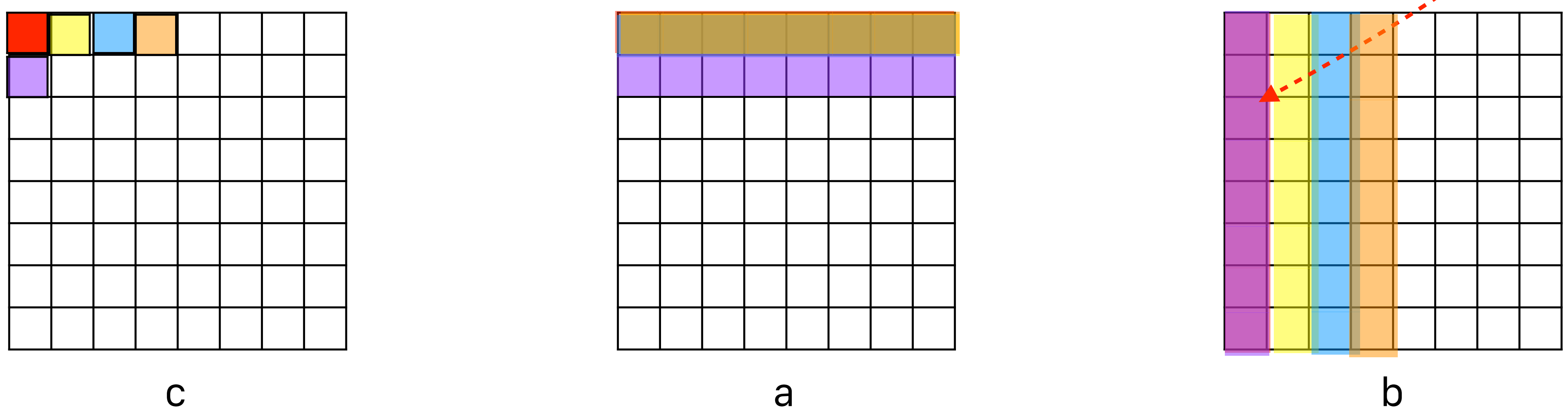
**If n=1024, it takes about 1 sec**

**How long is it take when n=2048?**

# Matrix Multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Very likely a miss if array is large

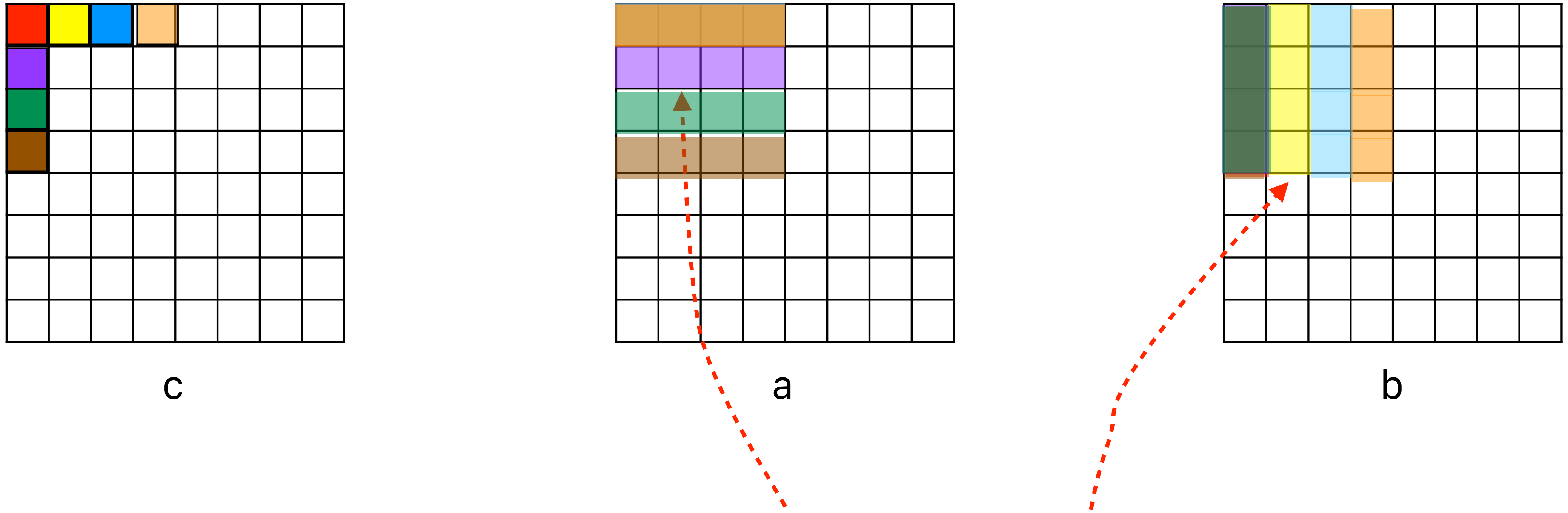



- If each dimension of your matrix is 2048
  - Each row takes 2048*8 bytes = 16KB
  - The L1 $ of intel Core i7 is 48KB, 12-way, 64-byte blocked
  - You can only hold at most 3 rows/columns of each matrix!
  - You need the same row when j increase!

# Block algorithm for matrix multiplication

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```




**You only need to hold these sub-matrices in your cache**

https://www.pollev.com/hungweitseng close in 1:30

# What kind(s) of misses can block algorithm remove?

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

**Naive**

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

**Block**

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

# What kind(s) of misses can block algorithm remove?

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

**Naive**

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

**Block**

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. $\boxed{\text{Capacity \& conflict miss}}$
E. Compulsory & conflict miss

# Announcement

- Assignment #2 due the upcoming Sunday
  - Assignments SHOULD BE done/submitted individually — if discussed with others, make sure their names on your submission
  - We will drop your least performing assignment as well
  - Check the website tonight for the template and questions
- Midterm next Monday — will release a midterm review online only lecture by Friday
  - It will be held during the lecture time — both in-person and online sessions — 8/21 2p-3:20p
  - Both formats contain multiple choices, short answers, free answers
  - Online examine will have 1.5x questions compared to in-person since typing is faster and online is open-book.

Computer
Science &
Engineering



つづく