CS61C Final Review

•••

Starring: Jonathan & Avi

Feedback Link:

https://forms.gle/Whbx4VjA8raMiZgX8

Exam Reminders

- Any 3 hours on Thursday 12/17
- Approximate time breakdown:
  - 30 min quest material
  - 90 min midterm material
  - 60 min post-midterm material
- Per 12/1 live lecture, C/RISC-V questions are **not** free-form coding - they're fill-in-the-blank like previous years
- Do your course evaluations! https://course-evaluations.berkeley.edu/berkeley

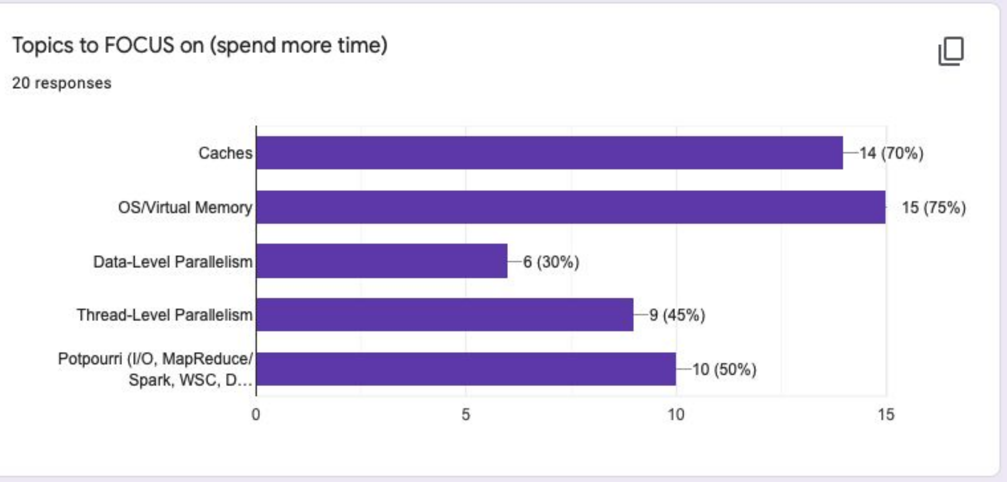
List of Other Resources

- Recorded Topical Review Sessions SP20/SU20
  - Hour long topical reviews on each major topic
- Quest Review + Midterm Review Recorded
  - Parts 1 & 2 in the trilogy
- Guerilla Section - Parallelism, I/O, Spark
  - **Friday 12/11 5-8pm PST**
- Practice Exams!

All linked on massive Finals Exam Logistics Post (@4326)!

Agenda

- Tomorrow: past exam (Fall 2019 Final~modified)
  - Live walkthrough (2.5 hours total)
- Today: topic review with questions (~2 hours total)
  - Caches ~30 min
  - Virtual Memory ~25 min
  - Parallelism ~40 min
  - Potpourri ~25 min
  - Open Q&A ~30 min



Topics to FOCUS on (spend more time)

20 responses

List of exam problems in these slides

- Printable worksheet can be found here
  - Caches: Fall 2015 Final MT2-4
  - VM: Fall 2018 Final F1, Summer 2018 Final Q11
  - TLP: Fall 2015 Final F2
  - I/O: Spring 2018 Final Q13(cd)
  - MapReduce/Spark: Summer 2018 Final Q13
  - Potpourri: Spring 2018 Final Q13(ab), Fall 2017 Final Q13(1, 2)
- More problems can be found on course website
- Tomorrow:
  - Fall 2019 Final (will replace Q3 with new question TBD since we looked at it for midterm review)

Cache me if you Can

Caches

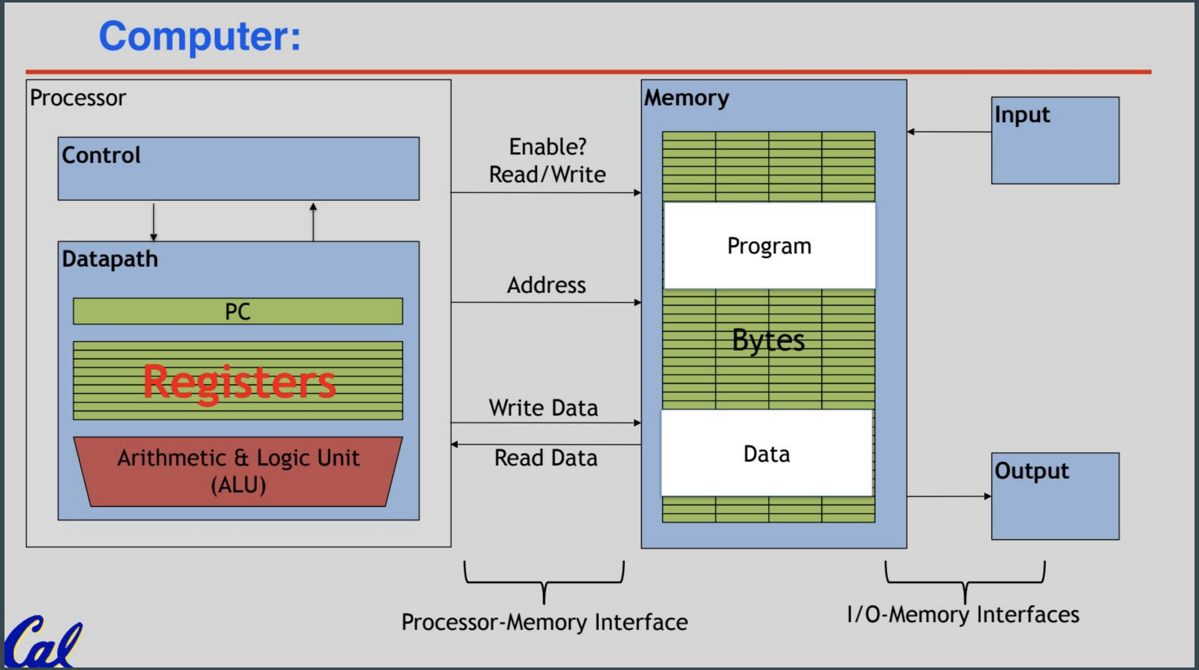
Questions you may have and are about to be answered

1. Who is a cache and why is it a thing?
2. Ok Avi, I get why I need a cache, but are there different types of caches and how do I use them?
3. Ok Avi, I know what a cache is, I know the different types and I know how to use it, but how do I analyze a given cache's performance?
4. Ok Avi, now given all this information you've put in my head how do I do a cache exam question?

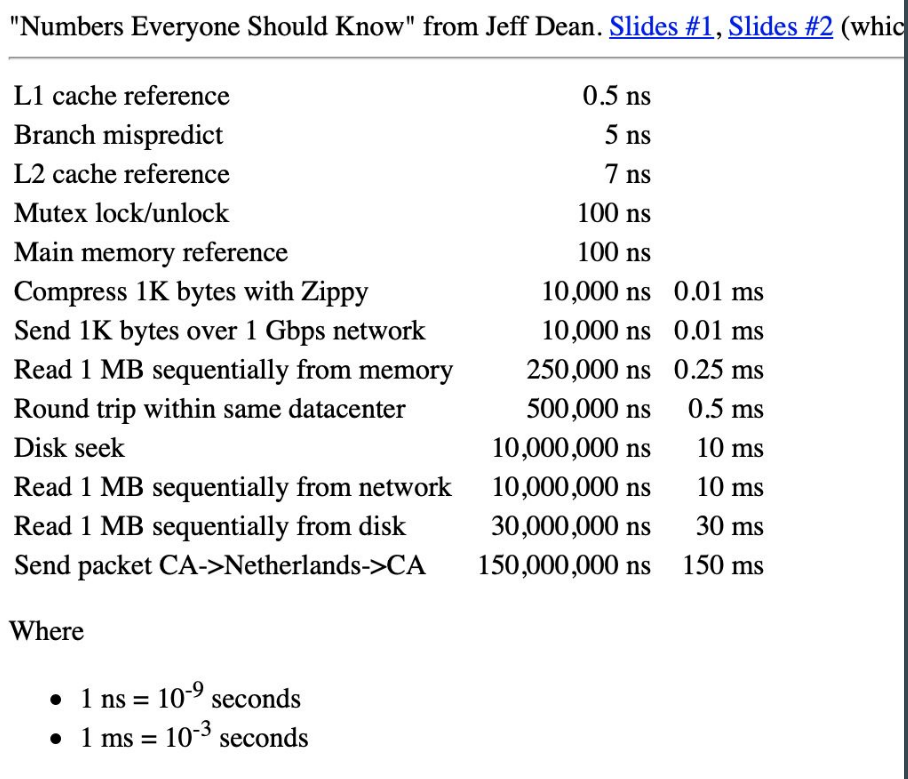
Part 1

Who is a cache and why is it a thing?

Why spend $$ on caches?



Memory Hierarchy



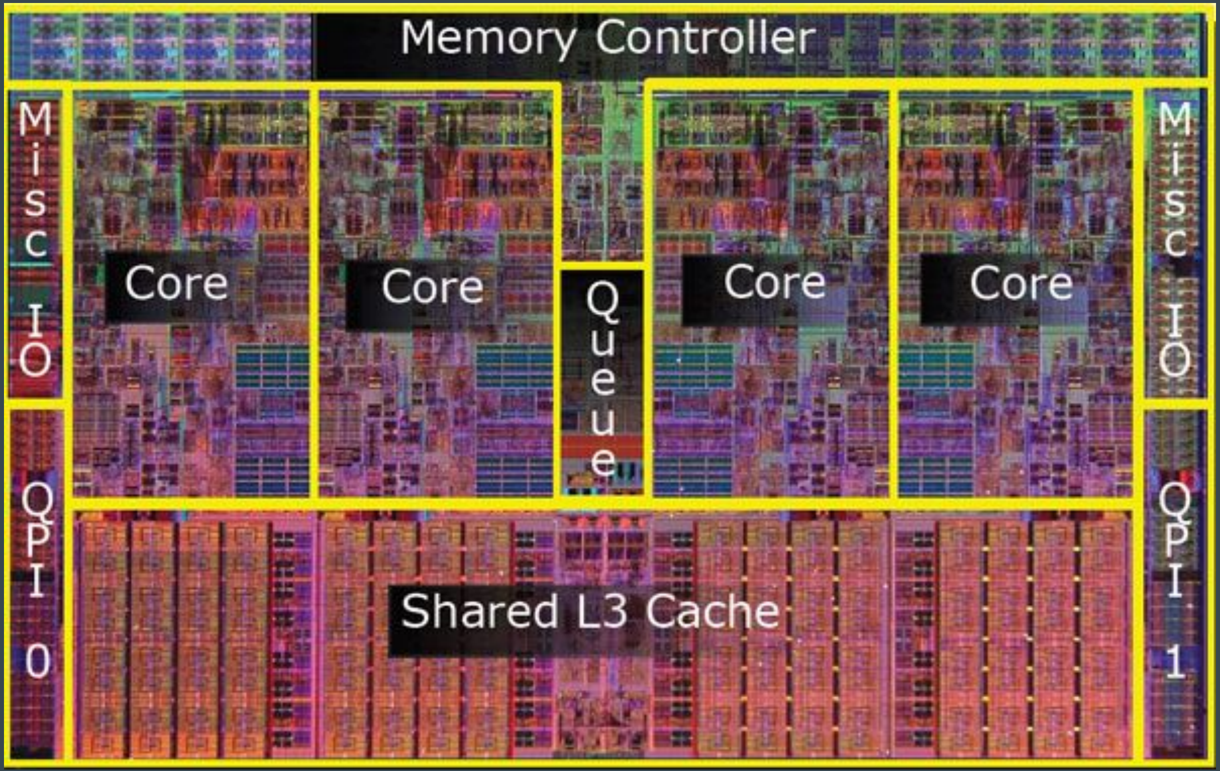
"Numbers Everyone Should Know" from Jeff Dean. Slides #1, Slides #2 (whic

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	100 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	10,000 ns	0.01 ms
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms
Read 1 MB sequentially from memory	250,000 ns	0.25 ms
Round trip within same datacenter	500,000 ns	0.5 ms
Disk seek	10,000,000 ns	10 ms
Read 1 MB sequentially from network	10,000,000 ns	10 ms
Read 1 MB sequentially from disk	30,000,000 ns	30 ms
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms

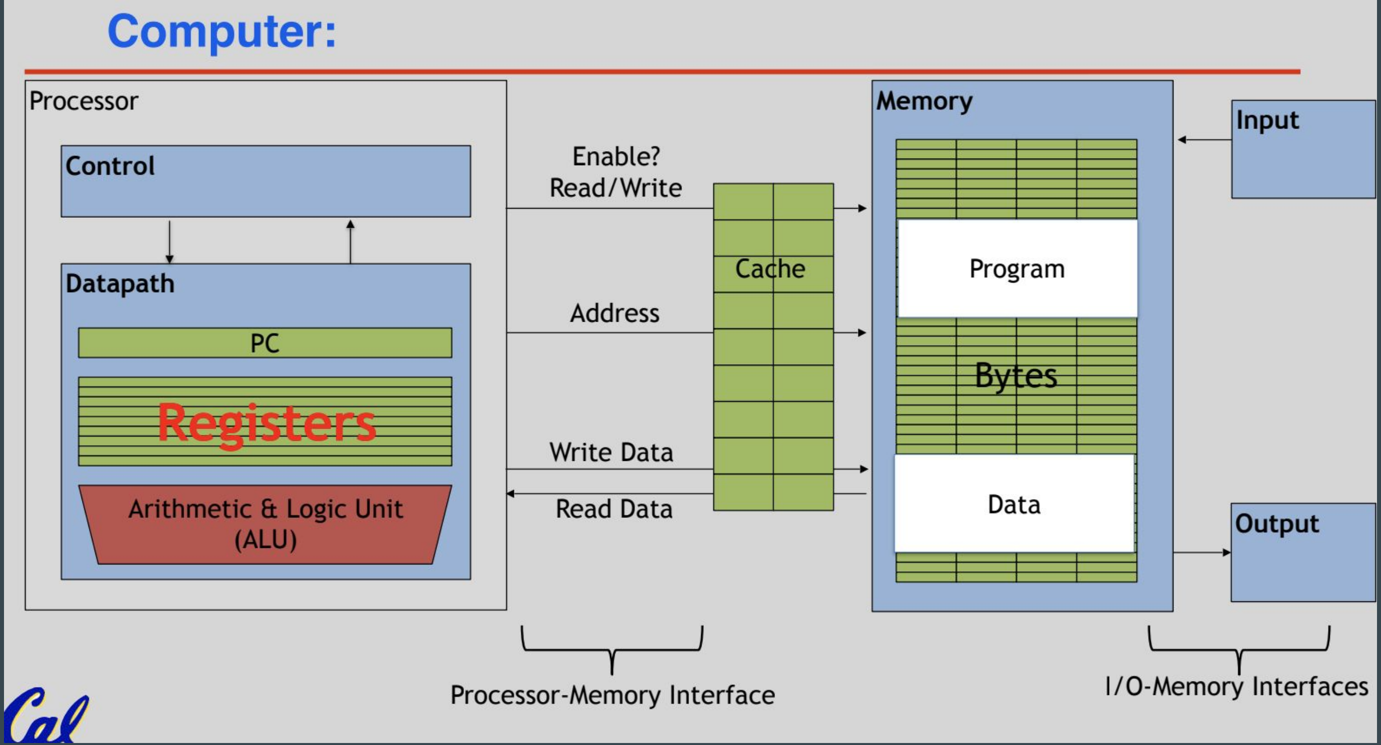
Where

- 1 ns = 10^{-9} seconds
- 1 ms = 10^{-3} seconds

Picture of an Intel i7 Processor



How do caches work?



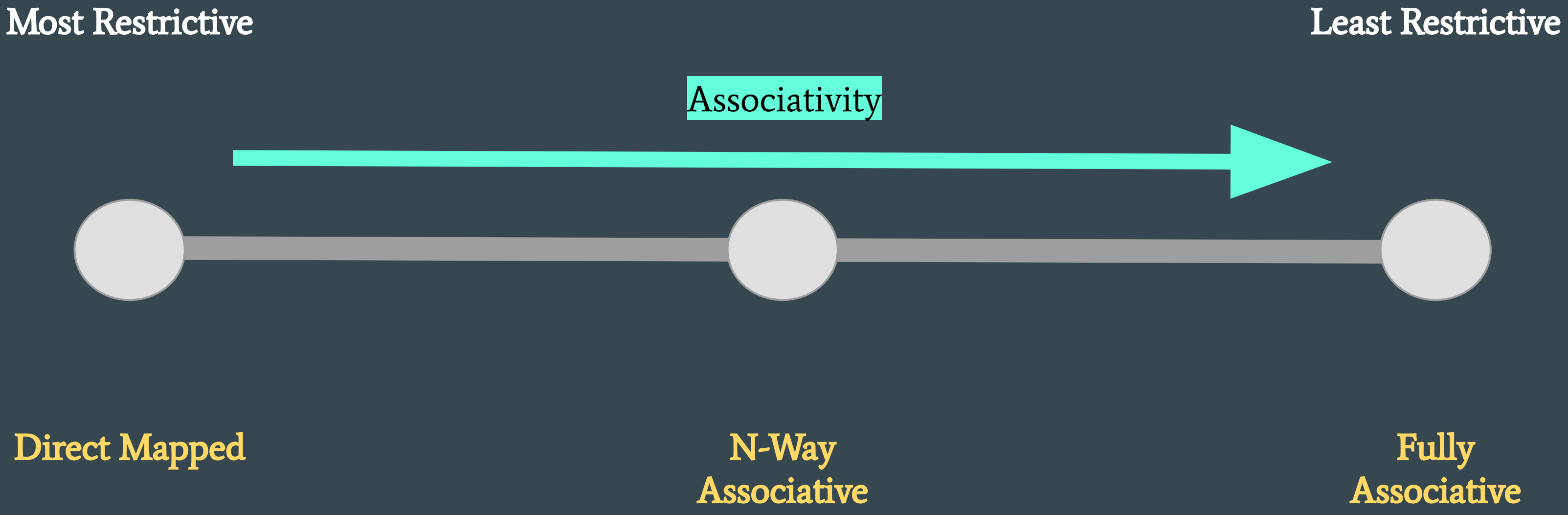
Caching + Caching Principles

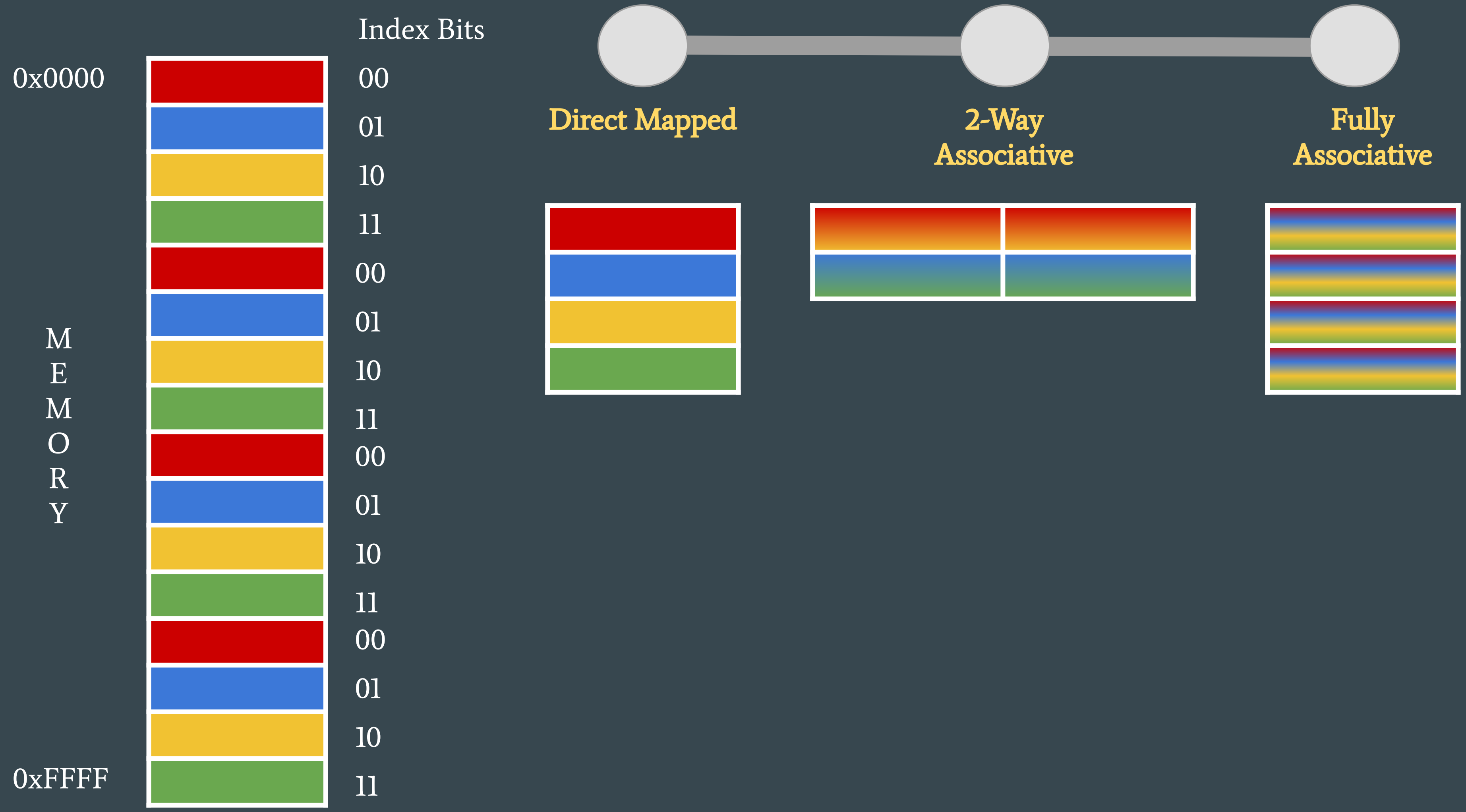
- Accessing data from memory is costly: it takes both computational resources and time!
- Caches store information closer to the processor for easy access. Caches are smaller than memory, so they can't fit everything, but they try to fit the most important stuff!
  - How do we specify what's important?
- Spatial Locality: locality with respect to location, "If we access this memory location, we might need its neighbour soon."
- Temporal Locality: locality with respect to time, "If a memory location is referenced now, it might be referenced again soon"

Part 2

Ok Avi, I get why I need a cache, but are there different types of of caches and how do I use them?

Types of Caches





Index Bits

0x0000

00
01
10
11
00
01
10
11
00
01
10
11
00
01
10
11

M
E
M
O
R
Y

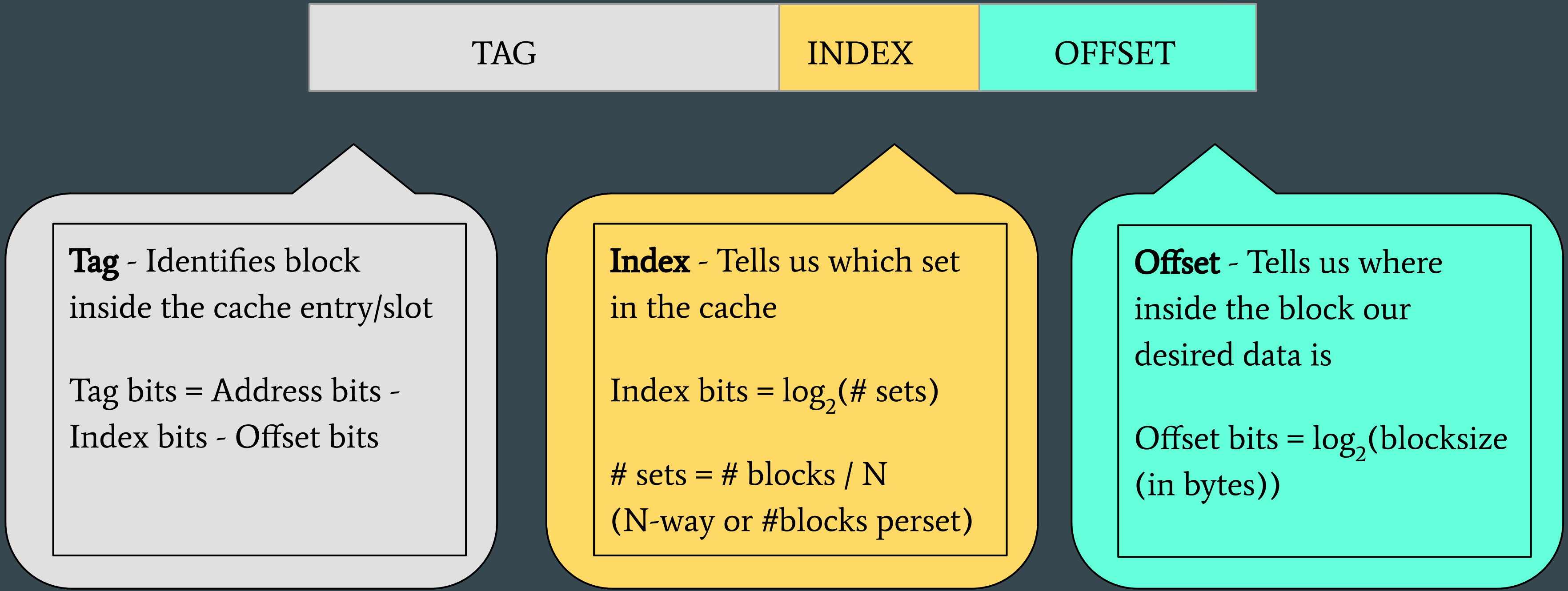
0xFFFF

Direct Mapped

2-Way Associative

Fully Associative

T:I:O Breakdown



TAG | INDEX | OFFSET

Tag - Identifies block inside the cache entry/slot

Tag bits = Address bits - Index bits - Offset bits

Index - Tells us which set in the cache

Index bits = $\log_2$(# sets)

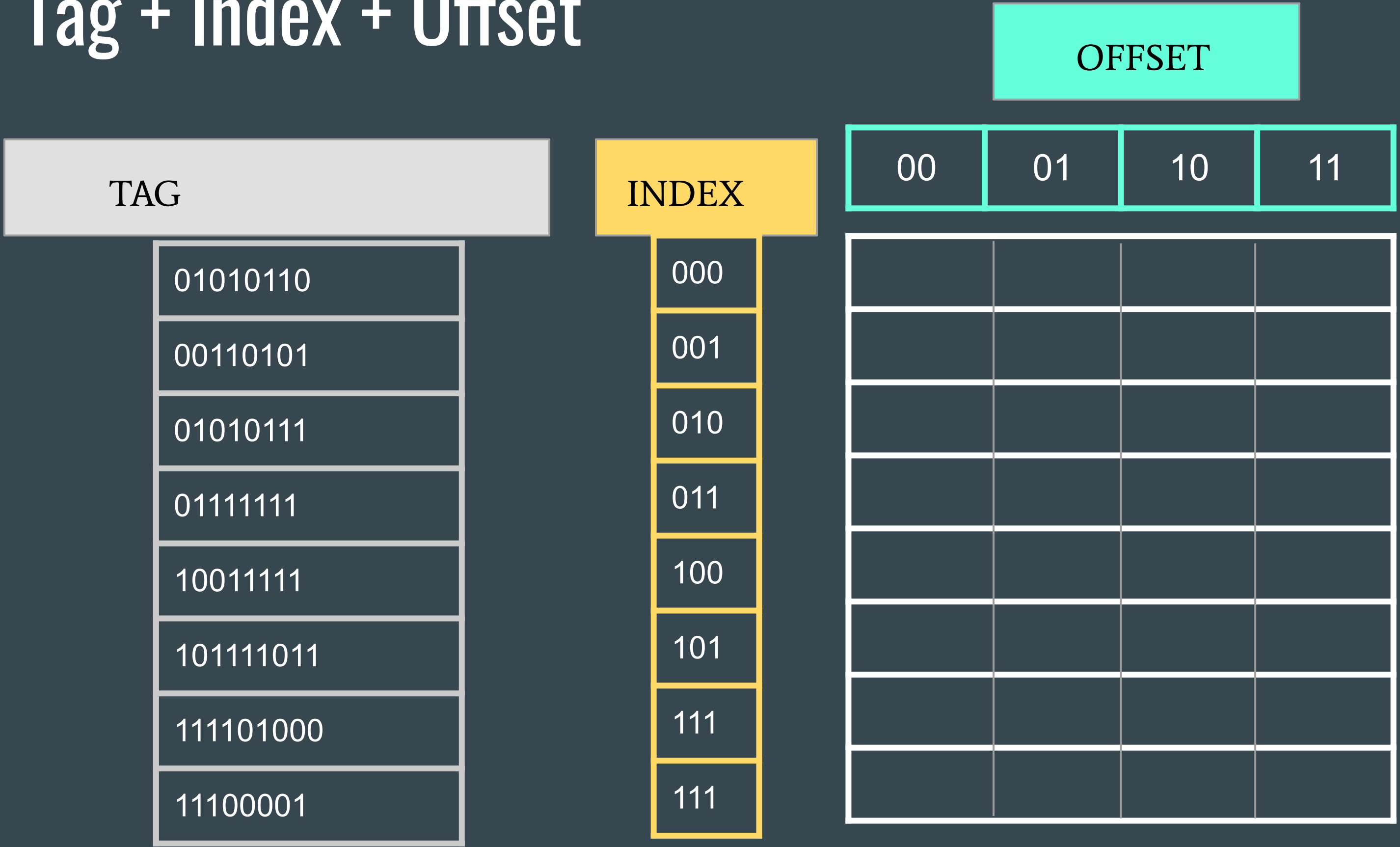
# sets = # blocks / N (N-way or #blocks perset)

Offset - Tells us where inside the block our desired data is

Offset bits = $\log_2$(blocksize (in bytes))

TAG | INDEX | OFFSET

Tag + Index + Offset

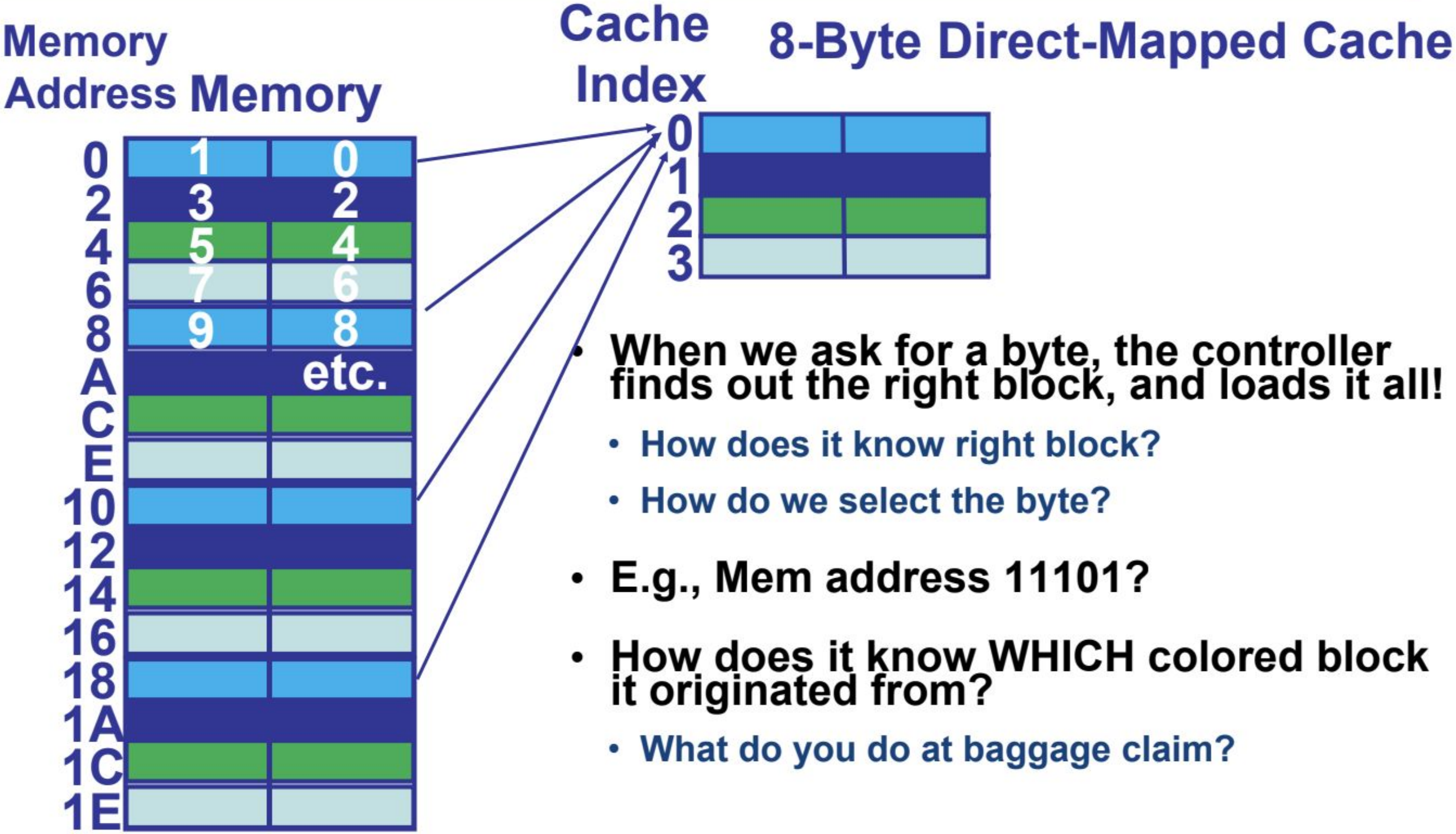


TAG	INDEX	OFFSET			
		00	01	10	11
01010110	000				
00110101	001				
01010111	010				
01111111	011				
10011111	100				
101111011	101				
111101000	111				
11100001	111				

T:I:O Breakdown

- T:I:O Breakdown (IN BITS)
    - **T**ag - Identifies block inside the cache entry/slot
        - Tag bits = Address bits - Index bits - Offset bits
    - **I**ndex - Tells us where the block is located inside the cache
        - Index bits = $\log_2$(# sets)
            - # sets = # blocks / N (N-way or # blocks per set)
            - # blocks = Cache Size (in bytes) / Block Size (in bytes)
    - **O**ffset - Tells us where inside the block our desired data is
        - Offset bits = $\log_2$(blocksize (in bytes))
- Cache size = $N * 2^{\#\text{offset bits} + \#\text{index bits}}$
- If two addresses differ by a multiple of $2^{\#\text{offset bits} + \#\text{index bits}}$, they map into the same set, since it will modify Tag.

Direct-Mapped Cache (3/4)

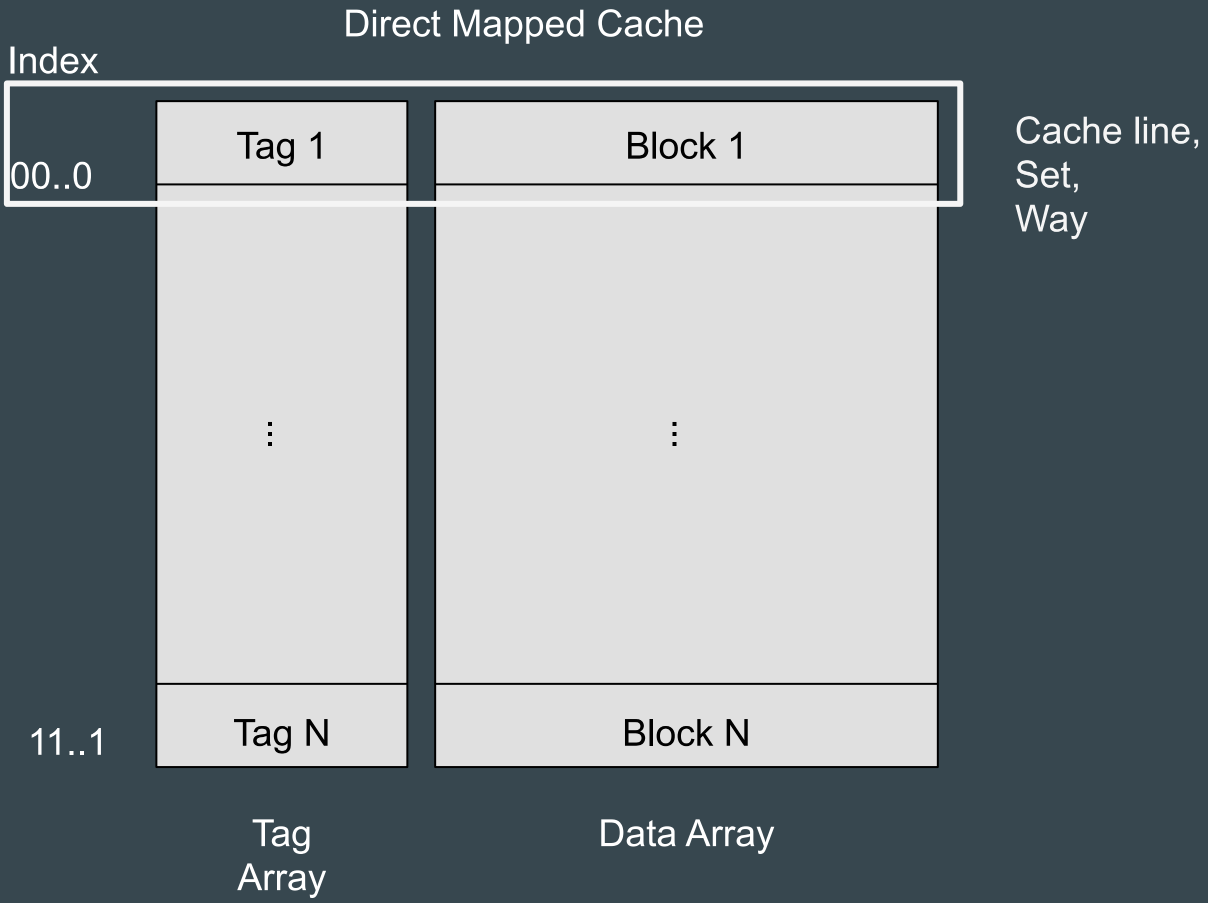


8-Byte Direct-Mapped Cache

- When we ask for a byte, the controller finds out the right block, and loads it all!
  - How does it know right block?
  - How do we select the byte?
- E.g., Mem address 11101?
- How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?

Cache organisations

Direct Mapped: A block can go in only one place, only one compare is required! Our number of sets equals the number of blocks, and there is no difference between sets and ways (1 way = 1 set)

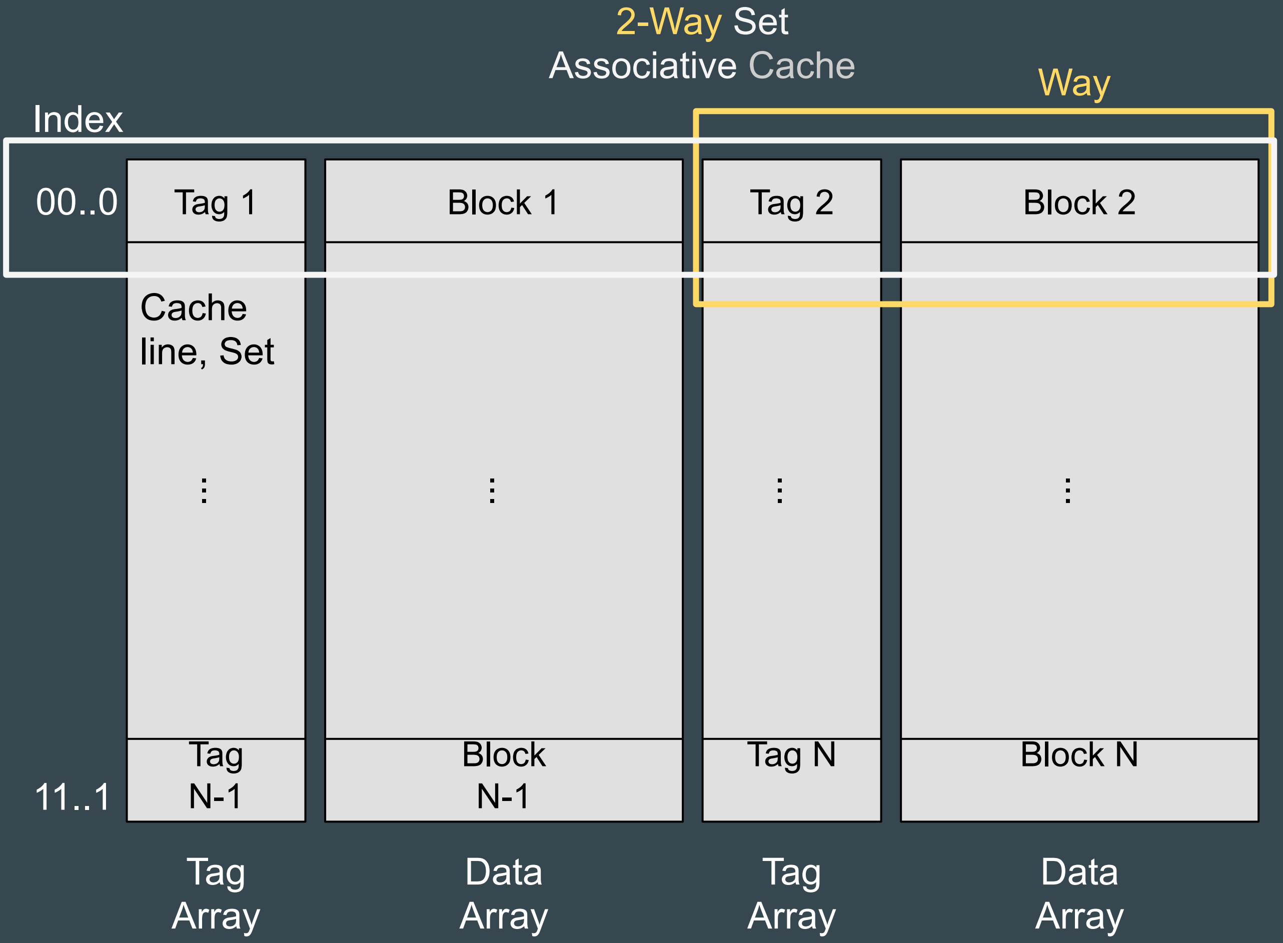


Cache organisations

Fully Associative: A block can go anywhere in this cache! There's no index (because we only have ~one set and having even 1 bit represents 2 things) and we have to do a lot of compares to find our data

N-way Set Associative: A block can go in one of N places. This requires N comparisons and gives us (# blocks/N) sets.

- Fully associative → N = # blocks
- Direct mapped → N = 1



Write Policy

- Writing Data!
- Cache **Write Hit Policy**:
    - Write through
        - Write to both cache and memory to enforce consistency
    - Write back (dirty bit)
        - Write to cache and write to memory if dirty
- Cache **Write Miss Policy**:
    - Write allocate
        - Put the block inside the cache and then perform the write-hit action
    - No write allocate
        - Write to memory and don’t put block inside the cache

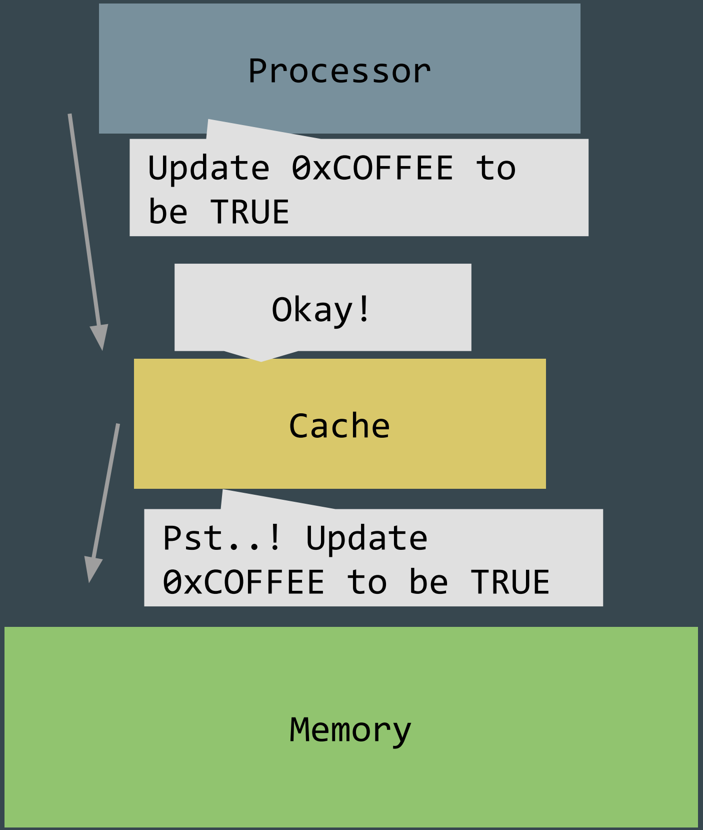
Write-through

Every time our processor tells our cache about new information it should store, our cache also passes the data back to memory

Select the word that best completes the sentence:

1. This scheme is slower than only updating our memory on eviction (write-back).

2. This scheme is better at keeping memory coherent in the face of failure.



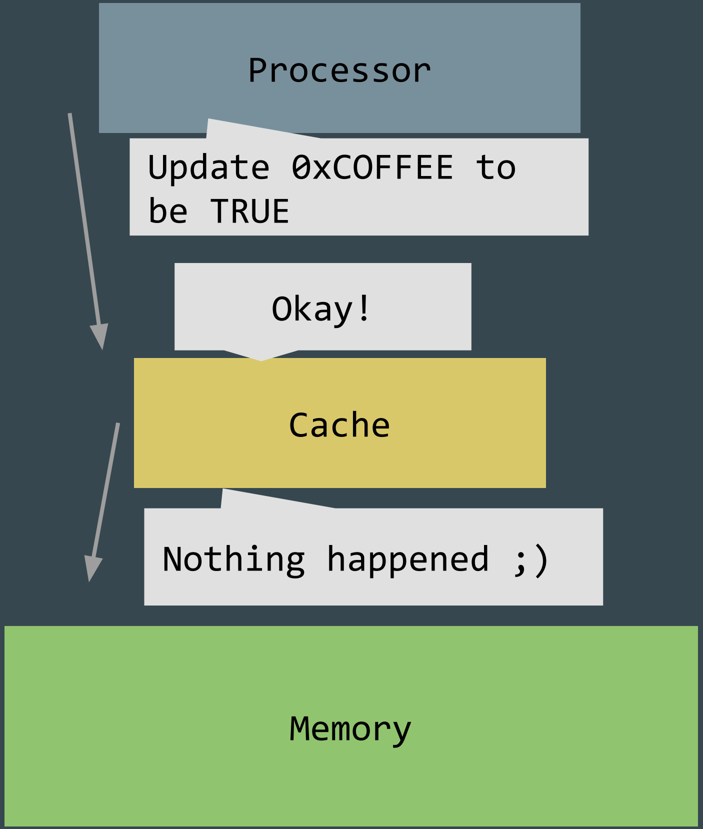
Write-back

When our processor tells our cache about new data it should store, only the cache updates. When the block containing that data is evicted from the cache, its data is written back to memory

Select the word that best completes the sentence:

1. This scheme is faster than updating memory per access (write-through).

2. This scheme is worse at keeping memory coherent in the face of failure.



Part 3

Ok Avi, I know what a cache is, I know the different types and I know how to use it, but how do I analyze a given cache’s performance?

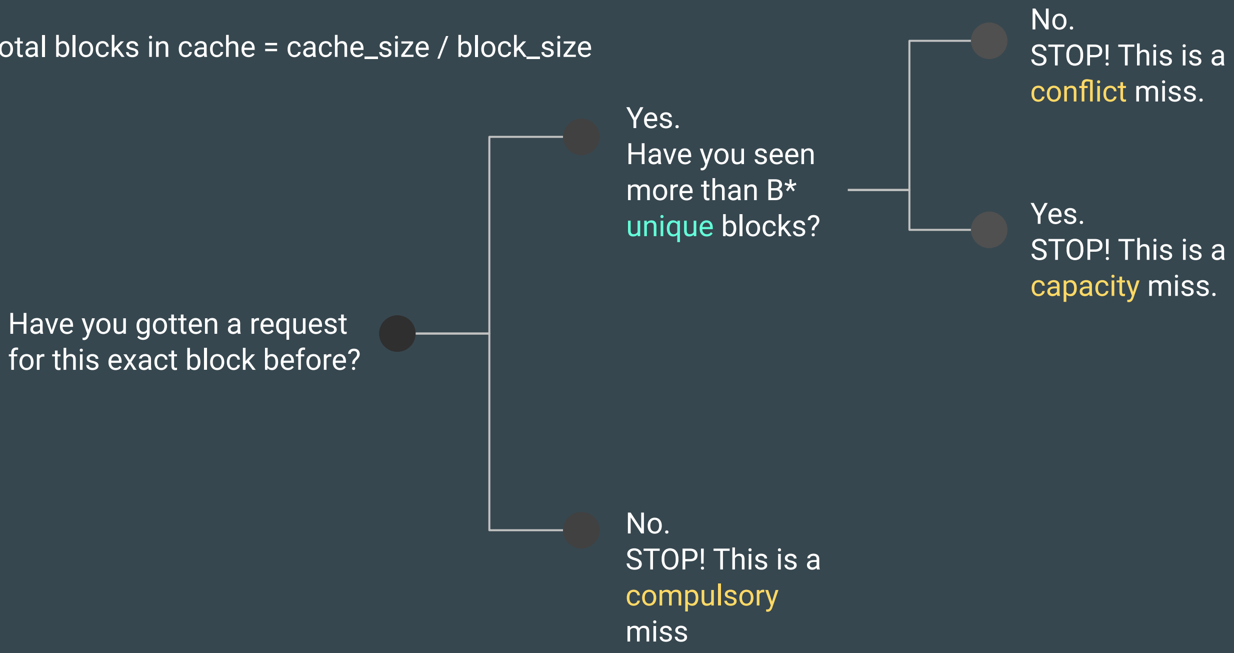
~Missing You~

1. Compulsory
    a. First time we bring in a block
2. Conflict
    a. If we increase associativity (e.g. redo all the accesses with a FA cache) would we still miss?
3. Capacity
    a. Cache was full so we had to replace a block because limited space

Classifying Misses

So, you did a tag check, and your data wasn't in the cache. Time to figure out what kind of miss you just incurred. Use the diagram below.

NOTE: B* = # total blocks in cache = cache_size / block_size



Have you gotten a request for this exact block before?

Yes.
Have you seen more than B* unique blocks?

No.
STOP! This is a conflict miss.

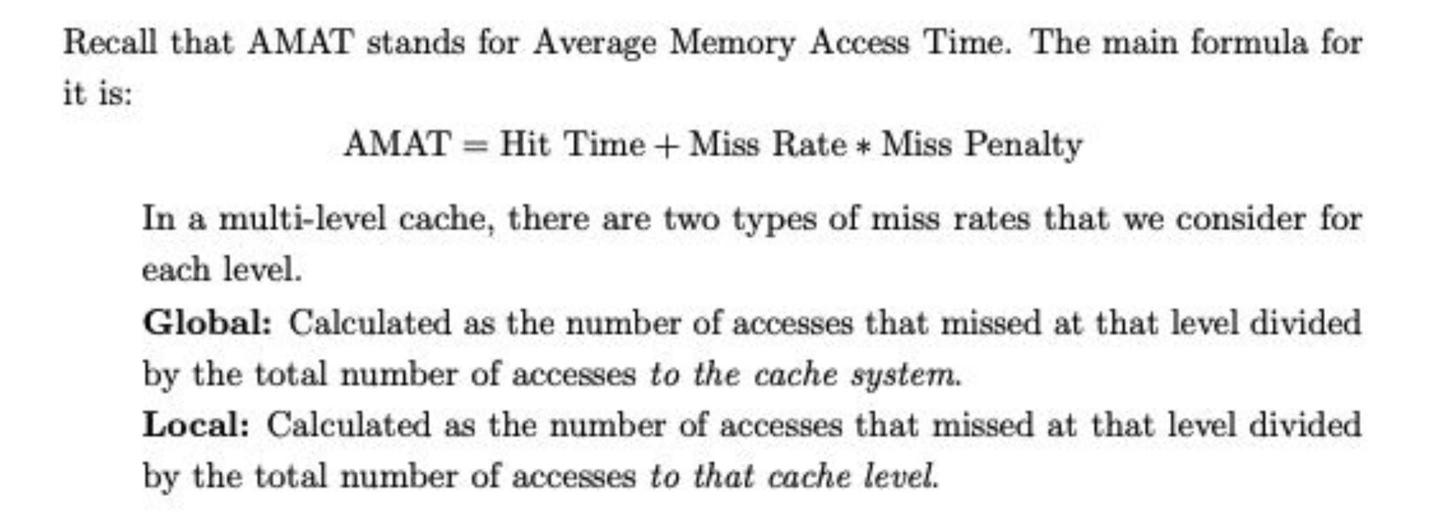
Yes.
STOP! This is a capacity miss.

No.
STOP! This is a compulsory miss

Hotel Analogy for Cache Misses

Type of Misses	Definition	Analogy for DM	Analogy for N-way
Compulsory Miss	Occurs at the first access of that memory location, so memory location has not yet been loaded into cache	When guests visit your hotel the very first time, he doesn’t have a room.	Same as DM
Conflict Miss	Two blocks are mapped to the same index(row) and there is not enough room to hold both.	A guest requests a particular floor to stay in. DM => 1 room/floor. The room/floor is already occupied and thus is unavailable.	A guest requests a particular floor to stay in. N-way => N rooms/floor. The floor has all rooms occupied and thus is unavailable.
Capacity Miss	The cache is not big enough to hold every block you want to use.	The hotel is full and thus cannot accommodate any more guests.	Same as DM

AMAT



Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache, there are two types of miss rates that we consider for each level.

Global: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to the cache system.*

Local: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to that cache level.*

Quick Check: Cache Organisations

True or False:

1. Increasing associativity always decreases conflict misses
2. It is impossible to decrease the number compulsory misses by changing our cache organisation.
3. Direct mapped caches have a larger hit time than N-way or Fully Associative caches.

Quick Check: Cache Organisations

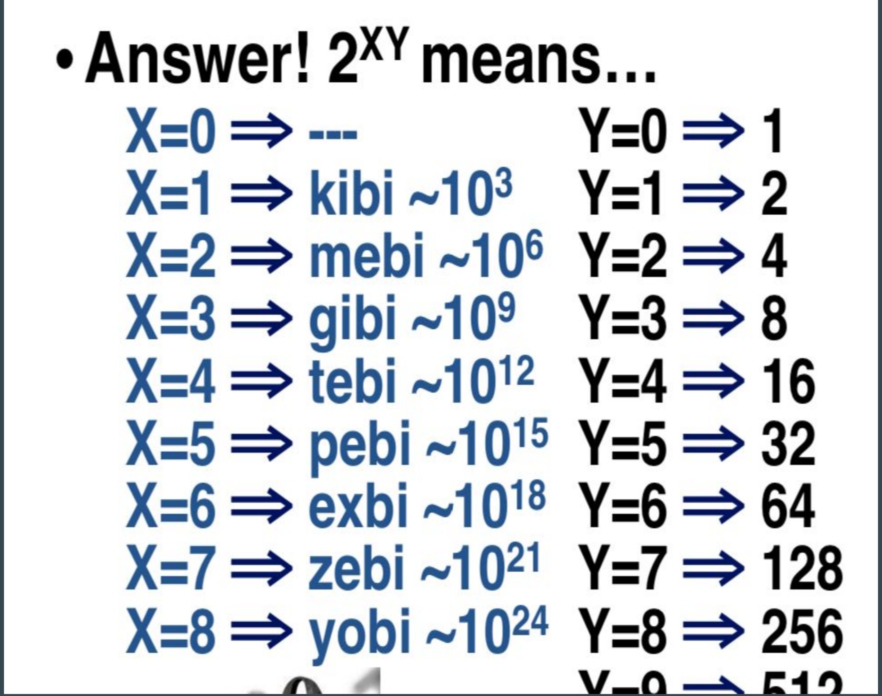
True or False:

1. Increasing associativity always decreases conflict misses
   True by definition! More associativity → more places for blocks with the same tags → less blocks kicked out due to conflicts
2. It is impossible to decrease the number compulsory misses by changing our cache organisation.
   False! Increasing block size pulls in more data per access and has the ability to decrease our overall compulsory misses (They won't go to 0, though!)
3. Direct mapped caches have a larger hit time than N-way or Fully Associative caches.
   False! Direct mapped caches have a /smaller/ hit time because they do fewer compares / have less overhead than FA/N-way SA caches.

Part 4

Ok Avi, now given all this information you’ve put in my head how do I do a cache exam question??

Conversion Cheat Sheet



- Answer! 2^{XY} means…

X=0 ⇒ --- Y=0 ⇒ 1

X=1 ⇒ kibi ~10^3 Y=1 ⇒ 2

X=2 ⇒ mebi ~10^6 Y=2 ⇒ 4

X=3 ⇒ gibi ~10^9 Y=3 ⇒ 8

X=4 ⇒ tebi ~10^{12} Y=4 ⇒ 16

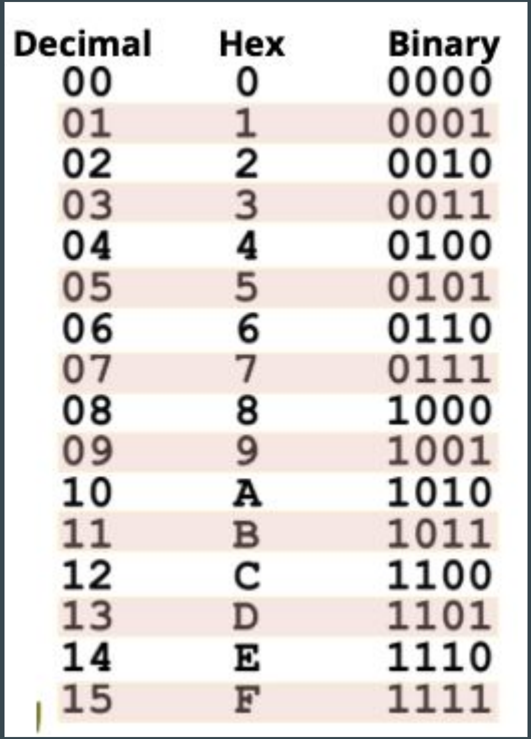
X=5 ⇒ pebi ~10^{15} Y=5 ⇒ 32

X=6 ⇒ exbi ~10^{18} Y=6 ⇒ 64

X=7 ⇒ zebi ~10^{21} Y=7 ⇒ 128

X=8 ⇒ yobi ~10^{24} Y=8 ⇒ 256

Y=9 ⇒ 512

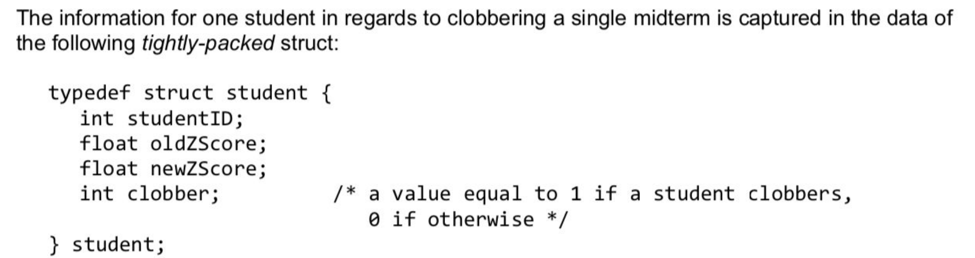


Decimal	Hex	Binary
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Things To Note To Solve Cache Questions

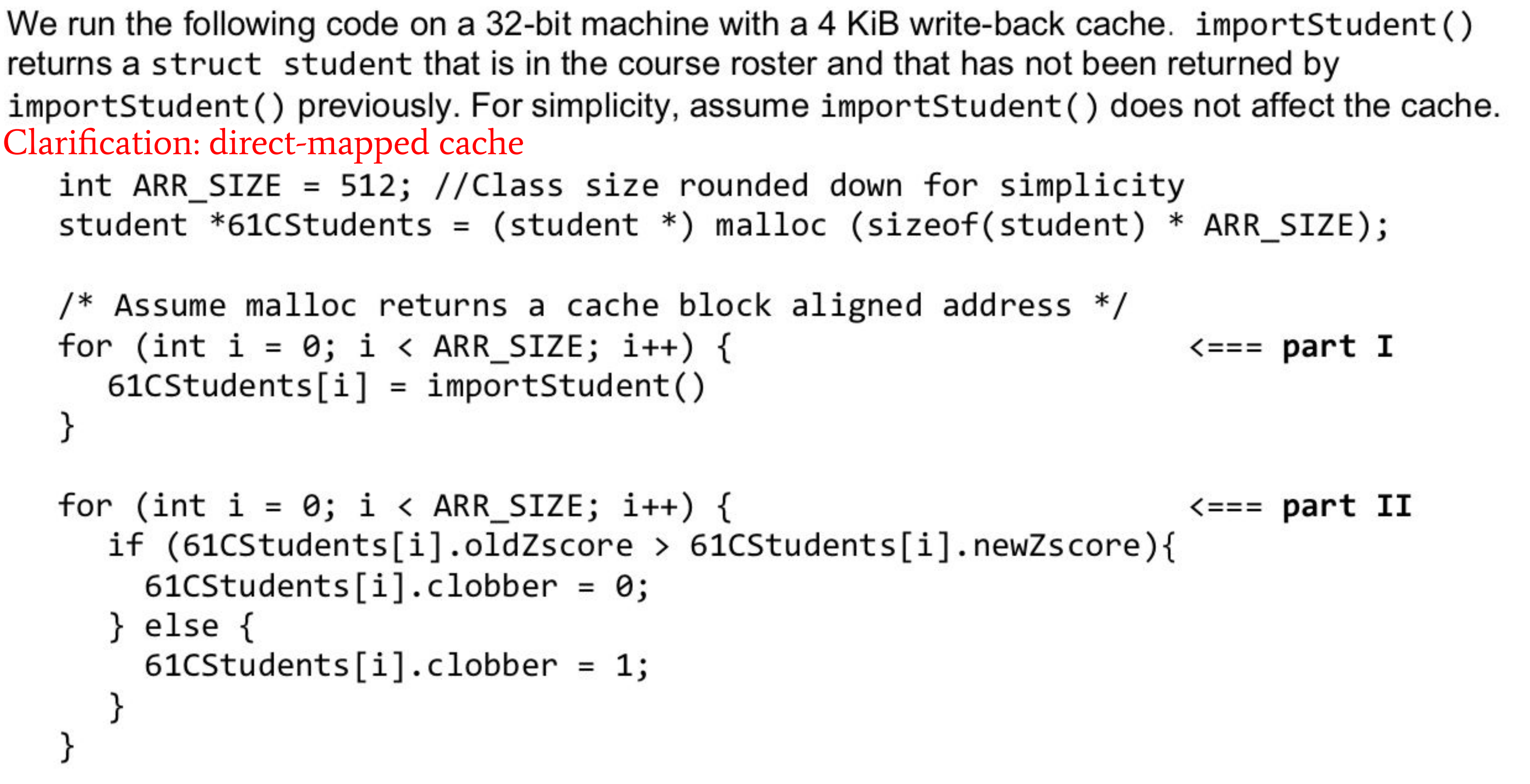
1. How many accesses per loop/step?
   a. Count # writes and reads
   b. Remember, a[i] += 5; is a read and a write
2. How many loops/steps per block?
   a. Compare step size to block size
3. How large is our cache compared to the array?
   a. Used to calculate what percentage of the array can we fit in our cache
   b. Can we reuse the cache in a later part of the function?
4. Do we reuse any blocks once we “leave” them?
   a. If we loop through again, may hit into a block again
5. How can we fix various misses

Fall 2015 Final



The information for one student in regards to clobbering a single midterm is captured in the data of the following *tightly-packed* struct:

```
typedef struct student {
   int studentID;
   float oldZScore;
   float newZScore;
   int clobber;          /* a value equal to 1 if a student clobbers,
                            0 if otherwise */
} student;
```



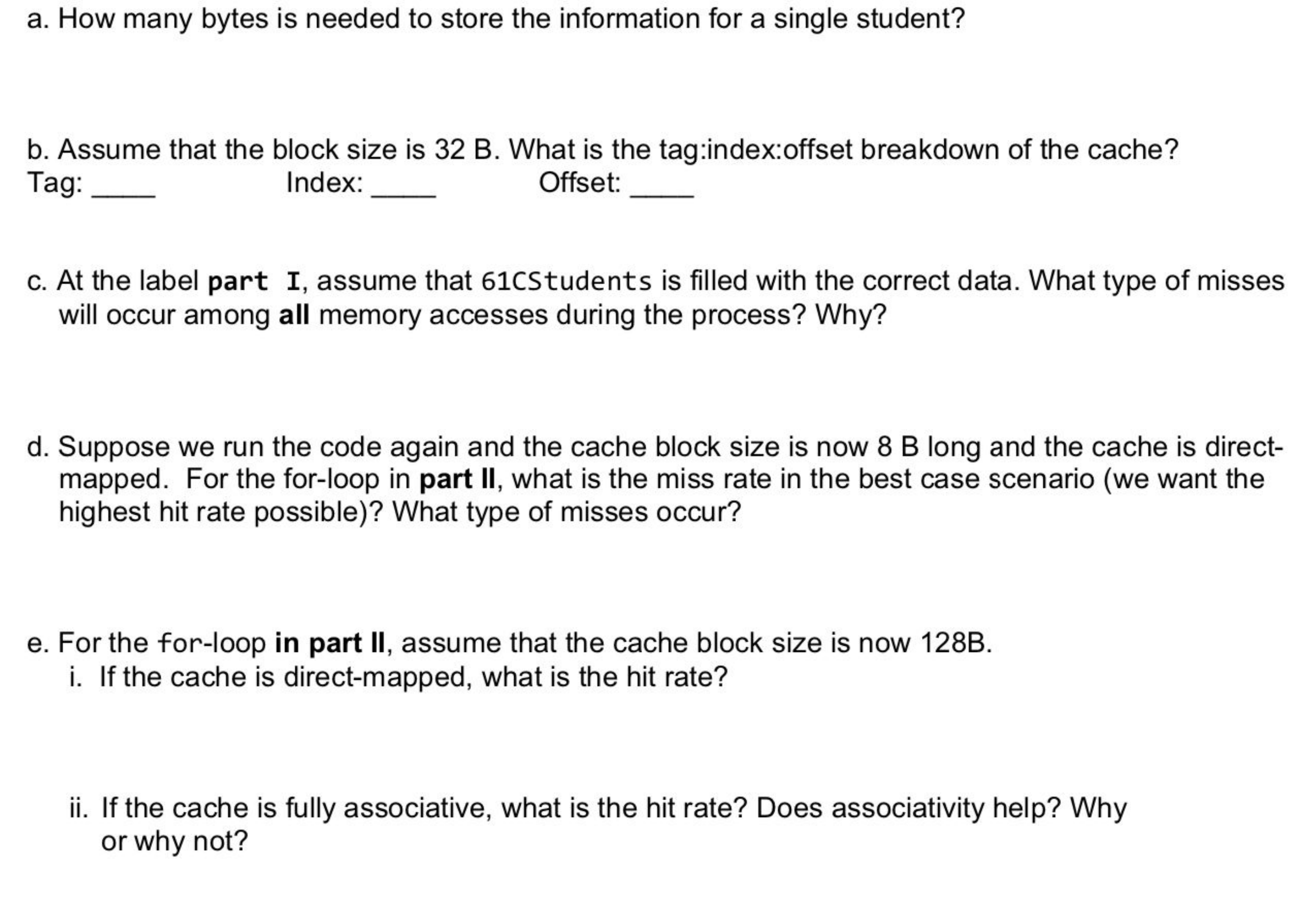
We run the following code on a 32-bit machine with a 4 KiB write-back cache. `importStudent()` returns a `struct student` that is in the course roster and that has not been returned by `importStudent()` previously. For simplicity, assume `importStudent()` does not affect the cache.

Clarification: direct-mapped cache

```
int ARR_SIZE = 512; //Class size rounded down for simplicity
student *61CStudents = (student *) malloc (sizeof(student) * ARR_SIZE);

/* Assume malloc returns a cache block aligned address */
for (int i = 0; i < ARR_SIZE; i++) {                    <=== part I
   61CStudents[i] = importStudent()
}

for (int i = 0; i < ARR_SIZE; i++) {                    <=== part II
   if (61CStudents[i].oldZscore > 61CStudents[i].newZscore){
     61CStudents[i].clobber = 0;
   } else {
     61CStudents[i].clobber = 1;
   }
}
```



a. How many bytes is needed to store the information for a single student?

b. Assume that the block size is 32 B. What is the tag:index:offset breakdown of the cache?
Tag: ____ Index: ____ Offset: ____

c. At the label **`part I`**, assume that `61CStudents` is filled with the correct data. What type of misses will occur among **all** memory accesses during the process? Why?

d. Suppose we run the code again and the cache block size is now 8 B long and the cache is direct-mapped. For the for-loop in **part II**, what is the miss rate in the best case scenario (we want the highest hit rate possible)? What type of misses occur?

e. For the `for`-loop **in part II**, assume that the cache block size is now 128B.

i. If the cache is direct-mapped, what is the hit rate?

ii. If the cache is fully associative, what is the hit rate? Does associativity help? Why or why not?

a. How many bytes is needed to store the information for a single student?

The struct has 4 fields, each of 4 bytes, so 16 bytes.

b. Assume that the block size is 32 B. What is the tag:index:offset breakdown of the cache?

Tag: 20___ Index: 7___ Offset: 5___

Thus, if the cache size is 4KiB (2^12), and the block size is 32 B (2^5), there must be 2^12/2^5 = 2^7 blocks. Thus, there are 7 index bits, 5 offset bits, and 20 tag bits. 20:7:5

c. At the label **part I**, assume that 61CStudents is filled with the correct data. What type of misses will occur among **all** memory accesses during the process? Why?

The misses that will occur from executing the for loop at label part I will be compulsory misses, because the loop will be accessing student structs that will be accessed for the first time.

d. Suppose we run the code again and the cache block size is now 8 B long and the cache is direct-mapped. For the for-loop in **part II**, what is the miss rate in the best case scenario (we want the highest hit rate possible)? What type of misses occur?

See Speaker Notes :)

e. For the for-loop **in part II**, assume that the cache block size is now 128B.

i. If the cache is direct-mapped, what is the hit rate?

8 students per block. 3 memory access per student, 1 miss and 23 hits, so 23/24

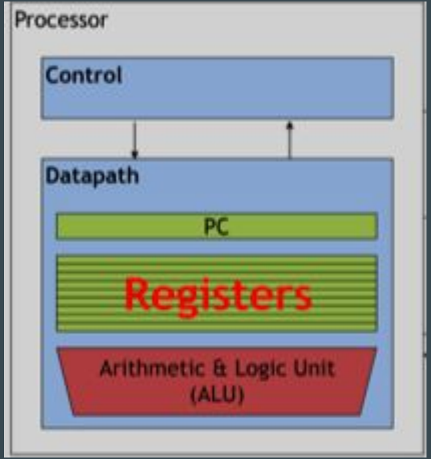
ii. If the cache is fully associative, what is the hit rate? Does associativity help? Why or why not?

It would still be 23/24, because the array is being accessed sequentially, so associativity makes no difference.

Virtual Memz

Virtual Memory

CPU



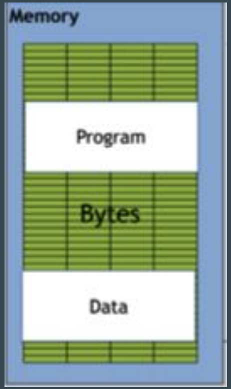
?

L1
Cache

L2
Cache

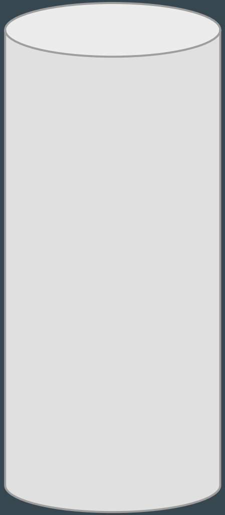
Cache
Cache

Main
Memory



?

Disk



Background

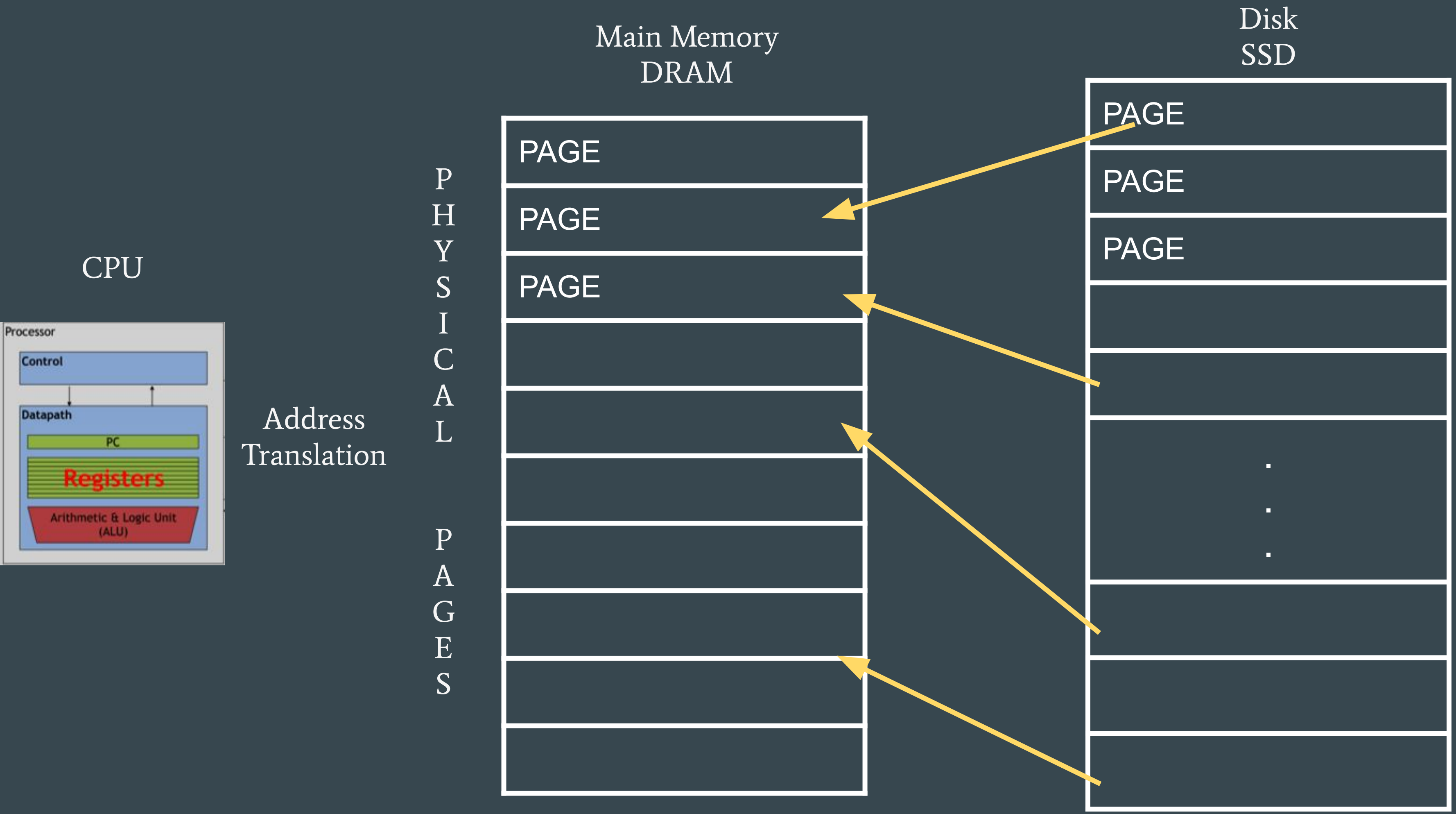
- Many processes on a CPU
  - Each process should “think” that it’s the only process on the machine!
    - Every process will assume its address space starts at 0x00000000
    - A process should not be able to access another process’s memory
  - Physical memory is limited - processes may want more
    - Should be able to use disk transparently - remember that disk is the next level of the memory hierarchy, so it’s bigger (and slower) than physical memory
    - “Illusion of infinite memory”
- Infinite Memory, Security, Protection

Virtual Memory

- Virtual memory - Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory: Working set of "pages" reside in main memory - others are on disk
  - Demand paging: Provides the ability to run programs larger than the primary memory (DRAM)
  - Hides differences between machine configurations
- Also allows OS to share memory, protect programs from each other
- Today, more important for protection than just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)

Paged Memory

- Memory is split into sections (blocks!) called pages
- Each process has a table of the blocks that belong to it called a **Page Table**
  - If process A has 6 pages that belong to it, that doesn't mean it has a range of addresses in memory. Sets of pages can be spread out all over physical memory.
- When your OS switches the active program, it changes the current page table so it knows the virtual and physical address mappings for *that* program. Before switching, the page table and associated program data are saved to disk.
  - The current page table is tracked by the "Page Table Base Register" (PTBR). This register holds the current table's address
  - Remember page tables are also stored in memory!



CPU

Address Translation

PHYSICAL PAGES

Main Memory DRAM

PAGE

PAGE

PAGE

Disk SSD

PAGE

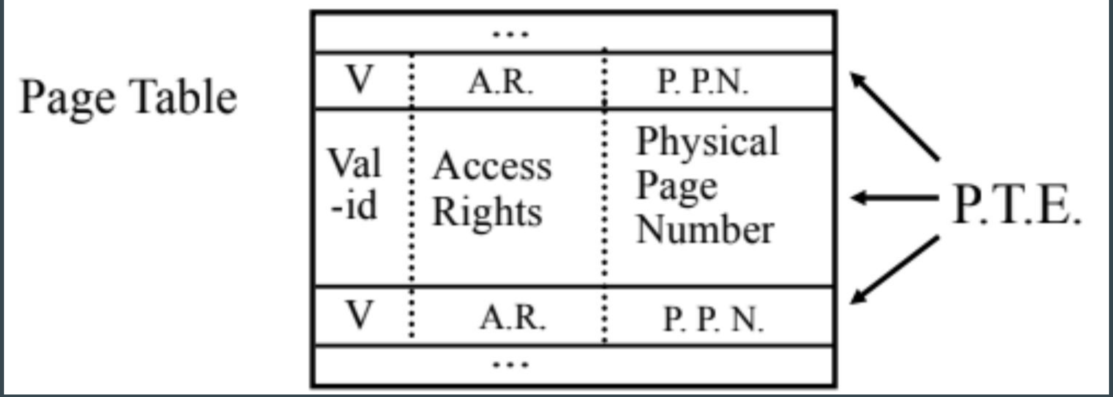
PAGE

PAGE

. . .

Page Table

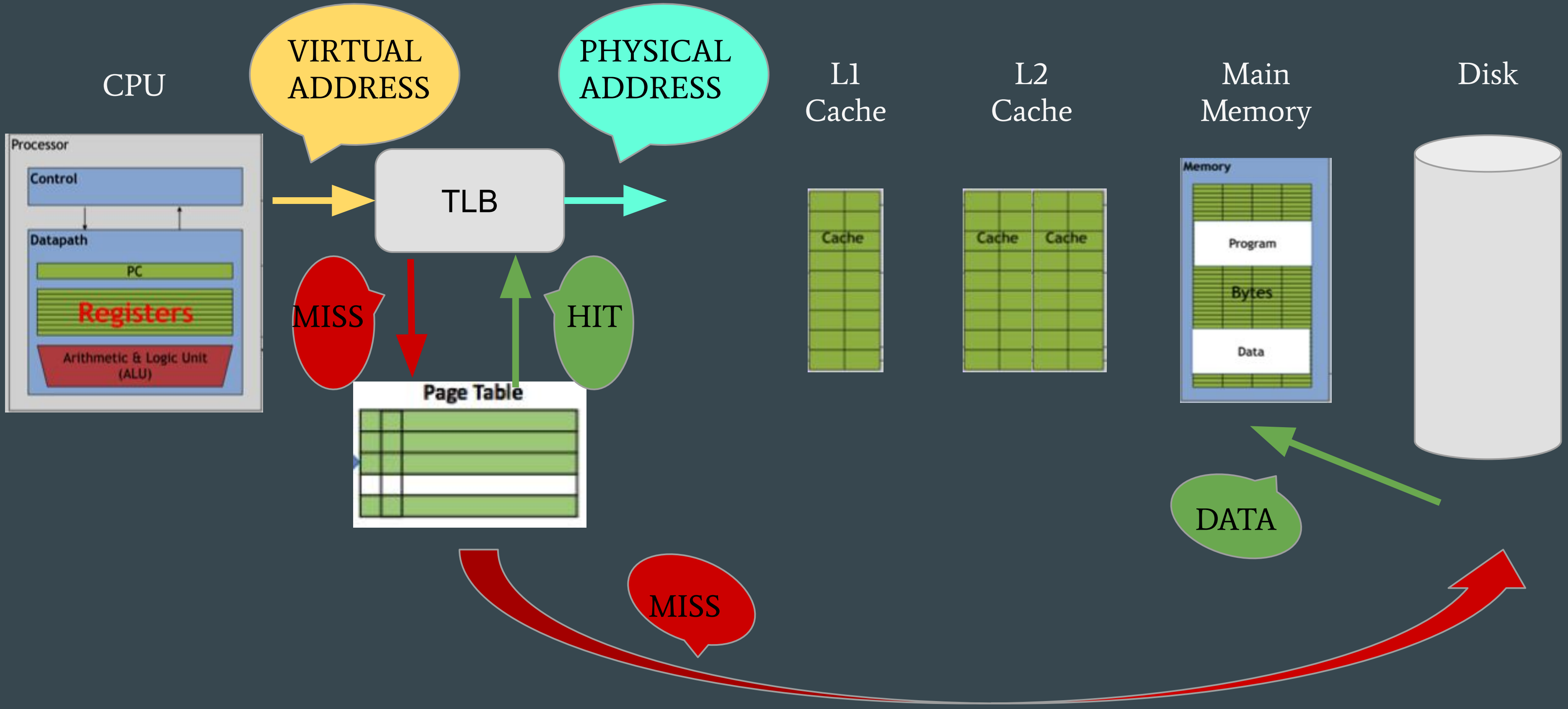
- Structure in memory that maps virtual page number (VPN) to physical page number (PPN)
- Since page tables are stored in memory, accessing them can be expensive
    - What do we do when we don't want to access memory as often?



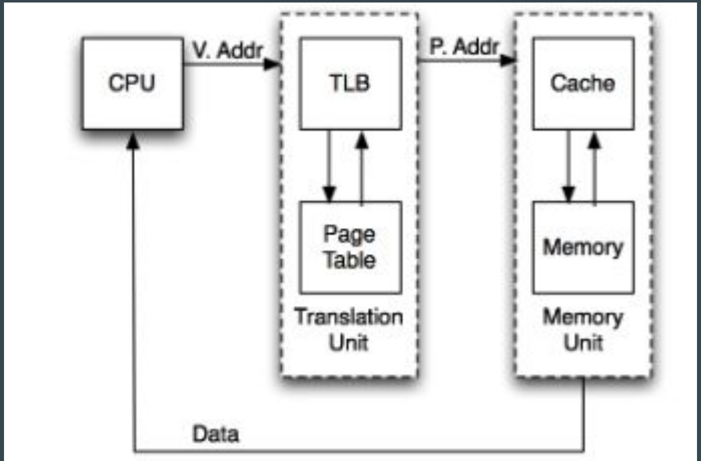
Translation Lookaside Buffer

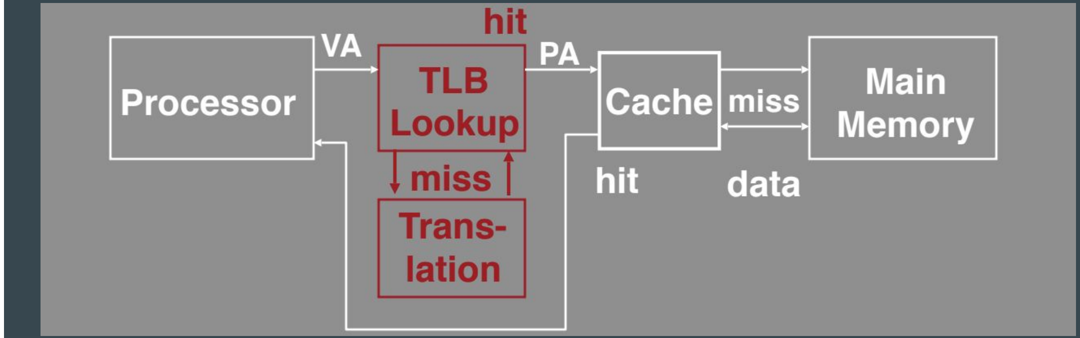
- Small, fully associative cache for address translations
- Exists in **hardware** - unlike the page table, all processes share the same TLB
    - What needs to happen in the TLB when we switch process contexts?

Big Picture


CPU
VIRTUAL ADDRESS
PHYSICAL ADDRESS
L1 Cache
L2 Cache
Main Memory
Disk
Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
TLB
MISS
HIT
Page Table
Cache
Cache
Cache
Memory
Program
Bytes
Data
DATA
MISS

Big Picture (Different Ways) Same Thing





True/False

1. If we have a TLB which contains a number of entries equal to MEMORY_SIZE / PAGE_SIZE, every TLB miss will also be a page fault.

2. If we change our TLB to direct-mapped, we're likely to see fewer TLB misses.

3. Every TLB miss is equally expensive in terms of the amount of time it takes for us to resolve our virtual address to a physical address.

4. A virtual address will always resolve to the same physical address.

(Adapted from CS 61C Fall 2018 Final)

True/False

1. If we have a TLB which contains a number of entries equal to MEMORY_SIZE / PAGE_SIZE, every TLB miss will also be a page fault.
   True - this means the TLB can hold references to every page in memory!
2. If we change our TLB to direct-mapped, we're likely to see fewer TLB misses.
   False - what kind of misses occur in direct-mapped but not fully associative?
3. Every TLB miss is equally expensive in terms of the amount of time it takes for us to resolve our virtual address to a physical address.
   False - what happens on a TLB miss?
4. A virtual address will always resolve to the same physical address.
   False

(Adapted from CS 61C Fall 2018 Final)

Page Table/TLB Simulation Practice

(Adapted from CS 61C Summer 2018 Final)

Consider a machine with 4 KiB pages, a 32-bit virtual address space with 256 MiB of DRAM for main memory. It has a single level of page table and a TLB containing 4 entries which is fully associative.

1. How many bits are there for the VPN? How many bits are there for the offset of the virtual address?

2. How many bits are there for the PPN? How many bits are there for the offset of the physical address?

Page Table/TLB Simulation Practice

(Adapted from CS 61C Summer 2018 Final)

Consider a machine with 4 KiB pages, a 32-bit virtual address space with 256 MiB of DRAM for main memory. It has a single level of page table and a TLB containing 4 entries which is fully associative.

1. How many bits are there for the VPN? How many bits are there for the offset of the virtual address?
   20, 12. 4 KiB = 2^{12} B, so there are 12 bits for offset, leaving 20 for VPN
2. How many bits are there for the PPN? How many bits are there for the offset of the physical address?
   16, 12. Physical/virtual must have the same offset size, and 256 MiB = 2^{28}, leaving 16 bits for PPN

Quick Break (probably)

Act II

Parallelism

this was a visual pun

Flynn’s Taxonomy

- SISD (Single Instruction, Single Data): one thread operating on one data source (most simple programs)
- SIMD (Single Instruction, Multiple Data): same operation being applied to multiple pieces of data at once (like Intel AVX)
- MIMD (Multiple Instruction, Multiple Data): different operations applied to different pieces of data (like running different programs on different CPUs)
- MISD (Multiple Instruction, Single Data): not covered in this course

SIMD

- Most modern CPUs have special extra-big registers that can hold multiple pieces of data at the same time
  - Intel AVX has 128-bit registers, which fit 4 32-bit integers
  - ...and also 256-bit registers, which fit 4 64-bit doubles
- General pseudocode for SIMD operations:
  - Iterate over data in groups of 4 elements
    - Load contiguous group of 4 elements into SIMD register
    - Do math on SIMD registers
    - If necessary, store 4 elements in SIMD register back to array in memory
  - If array length isn’t multiple of 4, perform naive version of code on remaining 1-3 elements

Summer 2018 Final Q9 [modified]

```
/* Computes an inner product between SM,which is a M x N matrix, and SV, which is a N x 1
 * vector. This result is stored in D, which is a M x 1 vector. */

void inner_product_simd (double *d, double *sm, double *sv, unsigned m, unsigned n) {
     __m256d matrix_part;
     __m256d vector_part;
     __m256d total;
     double arr[4];
     for (int i = 0; i < m; i += 1) {
          total = _mm256_setzero_pd ();
          for (int j = 0; j < _____________; j += _________) {
               matrix_part = _______________________________;
               vector_part =  ____________________________________;
               total = ___________________________________________;
          }
          ___________________________________________________;
          d[i] = arr[0] + arr[1] + arr[2] + arr[3];
     }
     for (int i = 0; i < m; i += 1) {
          // Tail case
          for (int j = _____________; j < _____________; j += _________) {
               _______________________________________;
          }
     }
}
```

```
__m256d _mm256_loadu_pd (double *)
     Loads the doubles at the address into a __m256d
void _mm256_storeu_pd (double *, __m256d)
     Stores the contents of the register to memory
__m256d _mm256_fmadd_pd(__m256d A, __m256d B, __m256d C)
     Calculates (A * B) + C
```

Summer 2018 Final Q9 [modified]

First, think about the naive solution:

```
void inner_product_naive (double *d, double *sm, double *sv, unsigned m, unsigned n) {
    // For each row of SM...
    for (int i = 0; i < m; i += 1) {
        // Take the dot product of the row with SV
        d[i] = 0;
        for (int j = 0; j < n; j += 1) {
            d[i] += sm[i * n + j] * sv[j];
        }
    }
}
```

We can look at the inner loop in groups of 4 elements using `fmadd`

Summer 2018 Final Q9 [modified]

```
__m256d _mm256_loadu_pd (double *)
        Loads the doubles at the address into a __m256d
void _mm256_storeu_pd (double *, __m256d)
        Stores the contents of the register to memory
__m256d _mm256_fmadd_pd(__m256d A, __m256d B, __m256d C)
        Calculates (A * B) + C
```

```
__m256d matrix_part;
__m256d vector_part;
__m256d total;
double arr[4];
for (int i = 0; i < m; i += 1) {
      total = _mm256_setzero_pd ();
      for (int j = 0; j < _____________; j += _________) {
            matrix_part = _________________________________;
            vector_part =  ______________________________;
            total = _____________________________________;
      }
      ________________________________________________;
      d[i] = arr[0] + arr[1] + arr[2] + arr[3];
}
```

```
// Naive
for (int i = 0; i < m; i += 1) {
      d[i] = 0;
      for (int j = 0; j < n; j += 1) {
            d[i] = sm[i * n + j]
                 * sv[j]
                 + d[i];
      }
}
```

Summer 2018 Final Q9 [modified]

```
__m256d _mm256_loadu_pd (double *)
      Loads the doubles at the address into a __m256d
void _mm256_storeu_pd (double *, __m256d)
      Stores the contents of the register to memory
__m256d _mm256_fmadd_pd(__m256d A, __m256d B, __m256d C)
      Calculates (A * B) + C
```

```
__m256d matrix_part;
__m256d vector_part;
__m256d total;
double arr[4];
for (int i = 0; i < m; i += 1) {
      total = _mm256_setzero_pd ();
      for (int j = 0; j < ______________; j += __________) {
            matrix_part = _mm256_loadu_pd(&sm[i * n + j]);
            vector_part = _mm256_loadu_pd(&sv[j]);
            total = _mm256_fmadd_pd(matrix_part, vector_part, total);
      }
      _mm256_storeu_pd(arr, total);
      d[i] = arr[0] + arr[1] + arr[2] + arr[3];
}
```

```
// Naive
for (int i = 0; i < m; i += 1) {
      d[i] = 0;
      for (int j = 0; j < n; j += 1) {
            d[i] = sm[i * n + j]
                 * sv[j]
                 + d[i];
      }
}
```

Summer 2018 Final Q9 [modified]

```
/* Computes an inner product between SM,which is a M x N matrix, and SV, which is a N x 1
 * vector. This result is stored in D, which is a M x 1 vector. */

void inner_product_simd (double *d, double *sm, double *sv, unsigned m, unsigned n) {
    __m256d matrix_part;
    __m256d vector_part;
    __m256d total;
    double arr[4];
    for (int i = 0; i < m; i += 1) {
        total = _mm256_setzero_pd ();
        for (int j = 0; j < n / 4 * 4; j += 4) {
            matrix_part = _mm256_loadu_pd(&sm[i * n + j]);
            vector_part = _mm256_loadu_pd(&sv[j]);
            total = _mm256_fmadd_pd(matrix_part, vector_part, total);
        }
        _mm256_storeu_pd(arr, total);
        d[i] = arr[0] + arr[1] + arr[2] + arr[3];
    }
    for (int i = 0; i < m; i += 1) {
        // Tail case
        for (int j = _____________; j < _____________; j += _________) {
            _______________________________________;
        }
    }
}
```

Summer 2018 Final Q9 [modified]

```c
/* Computes an inner product between SM,which is a M x N matrix, and SV, which is a N x 1
 * vector. This result is stored in D, which is a M x 1 vector. */

void inner_product_simd (double *d, double *sm, double *sv, unsigned m, unsigned n) {
      __m256d matrix_part;
      __m256d vector_part;
      __m256d total;
      double arr[4];
      for (int i = 0; i < m; i += 1) {
            total = _mm256_setzero_pd ();
            for (int j = 0; j < n / 4 * 4; j += 4) {
                  matrix_part = _mm256_loadu_pd(&sm[i * n + j]);
                  vector_part = _mm256_loadu_pd(&sv[j]);
                  total = _mm256_fmadd_pd(matrix_part, vector_part, total);
            }
            _mm256_storeu_pd(arr, total);
            d[i] = arr[0] + arr[1] + arr[2] + arr[3];
      }
      for (int i = 0; i < m; i += 1) {
            // Tail case
            for (int j = n / 4 * 4; j < n; j += 1) {
                  d[i] += sm[i * n + j] * sv[j];
            }
      }
}
```

```
__m256d _mm256_loadu_pd (double *)
      Loads the doubles at the address into a __m256d
void _mm256_storeu_pd (double *, __m256d)
      Stores the contents of the register to memory
__m256d _mm256_fmadd_pd(__m256d A, __m256d B, __m256d C)
      Calculates (A * B) + C
```

Thread Level Parallelism

- Thread = logical sequence of instructions, each with own execution state (PC and registers)
- With multiple physical CPUs, OS can run a different thread on each CPU (literally running in parallel)
- Can have fewer physical CPUs than threads!
  - OS will give illusion of running multiple threads at once by switching the currently running thread (copies PC, register values, and memory of next thread to run)
- OpenMP:
  - `pragma omp parallel { ... }` will spawn a new thread to execute everything between `{}`
  - `pragma omp parallel for` will spawn a few threads to divide up work on the iterations of a for loop

Race Conditions

- Threads can share memory
- Threads can be swapped at any time by the OS - what if one thread modifies memory another thread uses?
- Consider this code:

```
static int counter = 0;
// Assume there are exactly 2 threads
#pragma omp parallel for
for (int i = 0; i < 2; i++) {
    counter++;
}
```

- What is `counter` at the end?

Race Conditions (cont.)

- Translate the body of each thread to assembly:

```
la t0, counter
lw t1, 0(t0)
addi t1, t1, 1
sw t1, 0(t0)
```

```
int counter = 0;
// Assume there are exactly 2 threads
#pragma omp parallel for
for (int i = 0; i < 2; i++) {
     counter++; // counter = counter + 1
}
```

- OS can interleave 2 threads

```
thread 1: la t0, counter
thread 1: lw t1, 0(t0)      # Thread 1 reads counter = 0
thread 2: la t0, counter
thread 2: lw t1, 0(t0)      # Thread 2 reads counter = 0
thread 2: addi t1, t1, 1    # Thread 2 increments its copy of counter from 0 -> 1
thread 1: addi t1, t1, 1    # Thread 1 increments its copy of counter from 0 -> 1
thread 1: sw t1, 0(t0)      # Thread 1 stores back counter = 1
thread 2: sw t1, 0(t0)      # Thread 2 stores back counter = 1
```

- Result: wrong answer! We expect `counter` to be 2, not 1

Fixing Race Conditions

- In OMP, *variables outside* `parallel` *blocks are shared by threads*
  - Variables live in memory, so if multiple threads try to read/modify the same shared variable, you'll get the wrong result unless you're REALLY lucky
  - Variables within `parallel` blocks are thread-local
- Critical sections: identify sequence of C code where a thread must execute to completion before another thread is scheduled
  - Can be implemented in RISC-V using special "atomic" instructions, but generally not emphasized
  - See discussion 13 for details; Fall 2017 Final Q11 for example question

Fall 2015 Final F2

For each version of this code, indicate its correctness (always correct, sometimes correct, or always incorrect) and speed (faster, same, slower) compared to the serial version of the code.

Serial version:

```
#define RESULT_ARR_SIZE 8
#define ARR_SIZE 65536
result[0] = 0;
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
     int sum = 0;
     for (int j = 0; j < ARR_SIZE; j++) {
          sum += arr[j] + i;
     }
     result[i] = sum + result[i - 1];
}
```

Fall 2015 Final F2

For each version of this code, indicate its correctness (always correct, sometimes correct, or always incorrect) and speed (faster, same, slower) compared to the serial version of the code.

Version 1:

```
#define RESULT_ARR_SIZE 8
#define ARR_SIZE 65536
result[0] = 0;
#pragma omp parallel
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
     int sum = 0;
     for (int j = 0; j < ARR_SIZE; j++) {
          sum += arr[j] + i;
     }
     result[i] = sum + result[i - 1];
}
```

Always correct, but slower: “parallel” (without the “for”) will run the entire next statement/block in a new thread, so you’re just running the for loop multiple times redundantly

Fall 2015 Final F2

For each version of this code, indicate its correctness (always correct, sometimes correct, or always incorrect) and speed (faster, same, slower) compared to the serial version of the code.

Version 2:

```
#define RESULT_ARR_SIZE 8
#define ARR_SIZE 65536
result[0] = 0;
#pragma omp parallel for
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
     int sum = 0;
     for (int j = 0; j < ARR_SIZE; j++) {
          sum += arr[j] + i;
     }
     #pragma omp critical
     result[i] = sum + result[i - 1];
}
```

Sometimes correct - critical section doesn’t actually help, because there’s no guarantee thread for `result[i-1]` was run before `result[i]`

Faster, since iterations of the outer loop can be run concurrently

Fall 2015 Final F2

For each version of this code, indicate its correctness (always correct, sometimes correct, or always incorrect) and speed (faster, same, slower) compared to the serial version of the code.

Version 3:

```
#define RESULT_ARR_SIZE 8
#define ARR_SIZE 65536
result[0] = 0;
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
     int sum = 0;
     #pragma omp parallel for reduction(+:sum)
     for (int j = 0; j < ARR_SIZE; j++) {
          sum += arr[j] + i;
     }
     result[i] = sum + result[i - 1];
}
```

Always correct: no data dependencies in iterations of inner loop, and “reduction” keyword protects `sum`

Faster, since inner loop is large enough that overhead for starting threads becomes irrelevant

Cache Coherency

- Each core has its own L1/L2 cache, so how do multiple cores share memory?
- MOESI: protocol to track whether a block in cache needs updating
  - **M**odified: this cache has changed its copy of block, no other core has copy, must update memory
  - **O**wned: this cache has most updated value of block, though other cores may have copy (those others must be in **shared** state)
  - **E**xclusive: no modifications to block, no other core has copy
  - **S**hared: block up to date, but other cores may have copy
  - **I**nvalid: block not in cache
- False sharing: suppose thread 1 and thread 2 each read/write different variables `var_1` and `var_2`
  - No correctness issues since they're looking at different variables
  - However, if `var_1` and `var_2` are nearby in memory, they might be in the same cache block!
  - Result: if one cache writes to its copy of a block, it invalidates that block for other caches, meaning they have to fetch from memory or other processor, which is slow

Everything Else

I/O, OS, MapReduce/Spark, WSC, Dependability, GPUs

How to handle potpourri

- Many questions on these topics are multiple choice or true/false
- Often taken straight from lecture or discussion
    - Occasionally require a bit of additional reasoning/extrapolation, or applying formulas
    - Sometimes, problems are just glorified dimensional analysis
- We won’t be spending too much time on each topic here since there’s too many to discuss in depth
    - Important thing is for you to know key terms, and know where to look them up

I/O and OS Concepts

- OS serves three roles (see CS 162):
    - Referee: makes sure no one program hogs too many resources, schedules threads/processes
    - Glue: provides common services like access to a file system or network connections
    - Illusionist: provide easy-to-use abstractions (e.g. virtual memory, system calls)
- Interacting with I/O devices:
    - Managed by operating system - too risky to give programs direct access to hardware
    - Key concept: computers are fast (billions of instructions per second), device speeds vary
    - Polling: devices will tell us when they have data ready; user program is basically just a while loop

      ```
      while (!ready) { do_nothing(); } read_data();
      ```

        - Good if there’s lots of data (little time busy-waiting)
        - Bad if data arrives infrequently
    - Interrupts: ask OS to notify us when device has data; OS will copy data into memory for us (DMA)
        - Good if data is sparse
        - Bad if data comes too quickly (overhead for setting interrupt, switching to OS)
    - In practice, mix polling/interrupts (set interrupt for when data arrives, then switch to polling)

Spring 2018 Final Q13(d)

If the data comes in very infrequently do you want to use interrupts or polling? Why?

Interrupts if data is infrequent, since we'd waste a lot of cycles in while loop

For example, if you type at 684 characters per min., that's still only ~40k characters per sec.: most modern CPUs have frequencies in GHz, which means they execute 10^9 cycles per second

Spring 2018 Final Q13(c)

You have a processor that has a clock rate of 2GHz, a time to poll of 200 cycles for I/O, and you need to poll I/O at 100 Hz. If you use polling, what is the percentage of time you will need to spend polling?

(a) 1%
(b) 0.1%
(c) 0.01%
(d) 0.001%

Spring 2018 Final Q13(c)

You have a processor that has a clock rate of 2GHz, a time to poll of 200 cycles for I/O, and you need to poll I/O at 100 Hz. If you use polling, what is the percentage of time you will need to spend polling?

(a) 1%
(b) 0.1%
(c) 0.01%
[✓ (d) 0.001%]

100 Hz = 100 polls per sec., 200 cycles per poll → per second, we spend 100 * 200 = 20,000 cycles polling

20,000 / (2*10^9) = 0.001%

Amdahl’s Heartbreaking Law

- Parallelism has its limits
  - If you speed up a part of a program, the speedup of the whole program is bounded by the size of that part
  - speedup = $\frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$
    - s = length of non-sped-up portion
    - p = speedup of remaining (1-s) portion
    - limit is achieved as p → infinity
- Example from lecture: suppose ⅘ of a program can be parallelized for a 16x speedup; what is the overall program speedup?
  - Plug into formula: speedup = 1 / (⅕ + (⅘ / 16)) = 4x
  - Very sad :(

MapReduce/Spark

- For really large data sets, MapReduce and Spark provide simple ways to describe transformations on data that a cluster can then distribute work
- Key operations in Spark:
    - **map**: apply some function to each element in a data set
    - **reduce**: combine elements of list together by some operation
    - **reduceByKey**: combine key/value pairs that share the same key by some operation
- Try to break down problem into sequence of transformations

Summer 2018 Final Q13

In this question you will write Spark to find the mode of a list of values and how often it occurs. If there is a tie you can select any of the options. Fill in the blanks for the Python code below. Here is a sample input and output:

```
# Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
# Output: (1, 6)
def output_data(val):
      return ___

def compute_count(a, b):
      return ___

def find_max_occurrence(a, b):
      ___···___

# values = list(numbers)
modeData = sc.parallelize(values)
modeData.___(___)
        .___(___)
        .___(___)
```

Summer 2018 Final Q13

In this question you will write Spark to find the mode of a list of values and how often it occurs. If there is a tie you can select any of the options. Fill in the blanks for the Python code below. Here is a sample input and output:

```
# Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
# Output: (1, 6)
def output_data(val):
      return ___

def compute_count(a, b):
      return ___

def find_max_occurrence(a, b):
      ___...___

# values = list(numbers)
modeData = sc.parallelize(values)
modeData.___(___)
        .___(___)
        .___(___)
```

High level goals:

- Group together all identical values to produce pairs of (n, count)
  - Since we have identical values in array, this is probably with a `reduceByKey`
- Once we've produced pairs of (n, count), use `reduce` to identify the maximum

Summer 2018 Final Q13

In this question you will write Spark to find the mode of a list of values and how often it occurs. If there is a tie you can select any of the options. Fill in the blanks for the Python code below. Here is a sample input and output:

```
# Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
# Output: (1, 6)
def output_data(val):
      return (val, 1)

def compute_count(a, b):
      return ___

def find_max_occurrence(a, b):
      ____...___

# values = list(numbers)
modeData = sc.parallelize(values)
modeData.map(output_data)
        .___(___)
        .___(___)
```

High level goals:

- Group together all identical values to produce pairs of (n, count)
    - Since we have identical values in array, this is probably with a `reduceByKey`
- Once we've produced pairs of (n, count), use `reduce` to identify the maximum

Summer 2018 Final Q13

In this question you will write Spark to find the mode of a list of values and how often it occurs. If there is a tie you can select any of the options. Fill in the blanks for the Python code below. Here is a sample input and output:

```
# Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
# Output: (1, 6)
def output_data(val):
      return (val, 1)

def compute_count(a, b):
      return a + b # a, b will each subcounts

def find_max_occurrence(a, b):
      ____..___

# values = list(numbers)
modeData = sc.parallelize(values)
modeData.map(output_data)
        .reduceByKey(compute_count)
        .___(___)
```

High level goals:

- Group together all identical values to produce pairs of (n, count)
    - Since we have identical values in array, this is probably with a `reduceByKey`
- Once we've produced pairs of (n, count), use `reduce` to identify the maximum

Summer 2018 Final Q13

In this question you will write Spark to find the mode of a list of values and how often it occurs. If there is a tie you can select any of the options. Fill in the blanks for the Python code below. Here is a sample input and output:

```
# Input: [1, 2, 1, 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1, 1, 2, 2, 1]
# Output: (1, 6)
def output_data(val):
      return (val, 1)

def compute_count(a, b):
      return a + b # a, b will each subcounts

def find_max_occurrence(a, b):
      return a if a[1] > b[1] else b

# values = list(numbers)
modeData = sc.parallelize(values)
modeData.map(output_data)
        .reduceByKey(compute_count)
        .reduce(find_max_occurrence)
```

High level goals:

- Group together all identical values to produce pairs of (n, count)
    - Since we have identical values in array, this is probably with a `reduceByKey`
- Once we've produced pairs of (n, count), use `reduce` to identify the maximum

Data Centers and WSC

- In data centers, hardware and software are typically fairly homogeneous
- Performance metrics:
  - Response time/latency: time between start/finish of a task (how long it takes the autograder to run your job)
  - Throughput/bandwidth: total amount of work in a given time (how many jobs the autograder can run during one hour)
- Power Usage Effectiveness (PUE): (total building power) / (IT equipment power)
  - Accounts for power needed for cooling, network equipment, etc.
  - 1.0 = perfect (no additional overhead to run data center)

Dependability

- When you have a lot of computers, some of them start to fail
- Consequently, systems are designed to be failure-tolerant
- Availability = (mean time to failure) / (mean time to failure + mean time to repair)
  - To improve, either increase time to failure (better HW/SW) or reduce time to repair (better tools and diagnostics)
- Error correction codes (ECCs) allow systems to tolerate some amount of error in incorrectly transmitted data
  - Basic idea: add “parity” bits that are the XOR of a few known indices, act as checksums
  - Hamming ECC: choice of parity bits that can detect 2-bit errors and correct 1-bit errors
    - r parity bits protects a message of at most $2^r - r - 1$ bits
- RAID: duplicate data across multiple disks to protect against failure
  - RAID 1: just keep a copy of every disk (a “mirror”)
  - RAID 3, 4, 5: variants of using a disk as a parity check for other disks

Spring 2018 Final Q13(ab)

(a) You have a computer that, well, stinks. It goes down on average 6 times a day and it takes 1 hour to get working again. What is the current system's availability?

(b) Assume you have the computer from part (a) when the manufacturer offers you a deal. **a:** A new computer that only crashes 4 times per day or **b:** support that can reduce the time to fix to 6 minutes. Which one should you choose?

Spring 2018 Final Q13(ab)

(a) You have a computer that, well, stinks. It goes down on average 6 times a day and it takes 1 hour to get working again. What is the current system's availability?

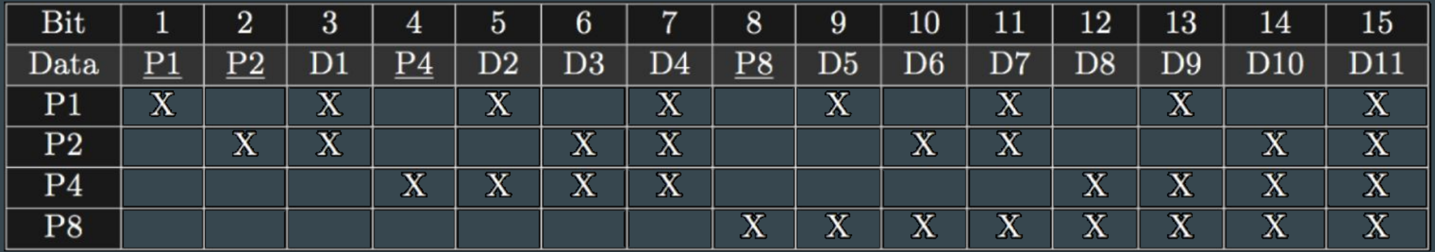
availability = MTTF / (MTTF + MTTR) = 4 / (4 + 1) = 4 / 5 = 0.8

(b) Assume you have the computer from part (a) when the manufacturer offers you a deal. **a:** A new computer that only crashes 4 times per day or **b:** support that can reduce the time to fix to 6 minutes. Which one should you choose?

option a: availability = 6 / (6 + 1) = 6 / 7
option b: availability = 4 / (4 + 0.1) = 4 / 4.1
b is better

Fall 2017 Final Q13(1, 2)

The Hamming Code table is reproduced below.



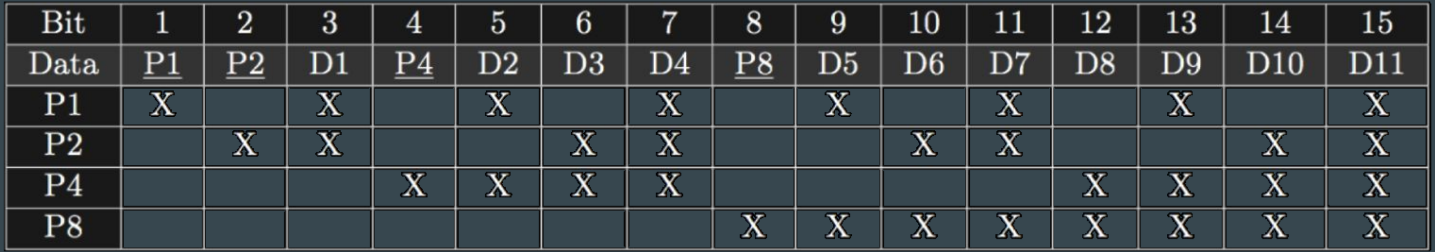
Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	P1	P2	D1	P4	D2	D3	D4	P8	D5	D6	D7	D8	D9	D10	D11
P1	X		X		X		X		X		X		X		X
P2		X	X			X	X			X	X			X	X
P4				X	X	X	X					X	X	X	X
P8								X	X	X	X	X	X	X	X

1. How many parity bits are needed to correct a single error using Hamming Codes in a message of 34 bits?

2. What data bits are encoded by the following 7-bit code word: `0b0101101`?

Fall 2017 Final Q13(1)

The Hamming Code table is reproduced below.



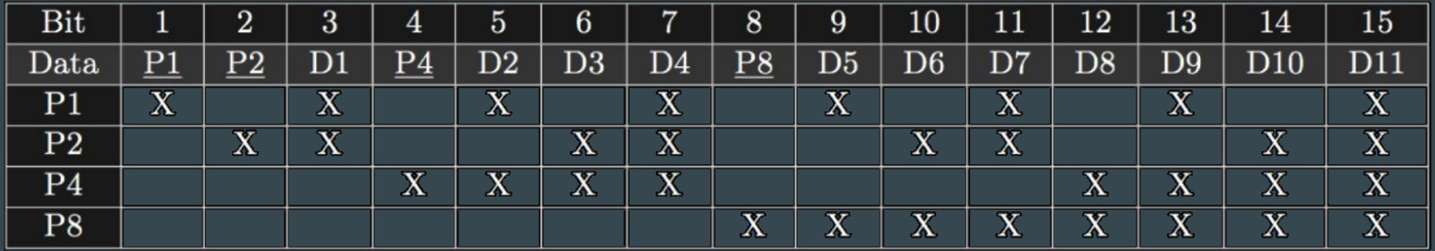
Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	P1	P2	D1	P4	D2	D3	D4	P8	D5	D6	D7	D8	D9	D10	D11
P1	X		X		X		X		X		X		X		X
P2		X	X			X	X			X	X			X	X
P4				X	X	X	X					X	X	X	X
P8								X	X	X	X	X	X	X	X

How many parity bits are needed to correct a single error using Hamming Codes in a message of 34 bits?

r parity bits corrects up to $2^r - r - 1$ message bits, so $r = 6$

Fall 2017 Final Q13(2)

The Hamming Code table is reproduced below.



Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	P1	P2	D1	P4	D2	D3	D4	P8	D5	D6	D7	D8	D9	D10	D11
P1	X		X		X		X		X		X		X		X
P2		X	X			X	X			X	X			X	X
P4				X	X	X	X					X	X	X	X
P8								X	X	X	X	X	X	X	X

What data bits are encoded by the following 7-bit code word: `0b0101101`?

From table, bits 3 and 5-7 (1-indexed) are data, so the message is

```
    0b0101101 -> 0b0101
```

Rows identify which parity bit protects which data bits

For example, P1 = D1 ^ D2 ^ D4 ^ D5 ^ D7 ^ D9 ^ D11

Taking these XORs verify that there are no errors

GPU Guest Lecture

- GPUs optimize throughput, CPUs optimize latency
- High-level pipeline:
  - GPU language (e.g. Metal, OpenGL) -> vertices -> triangles -> pixels -> pixels drawn on screen
- SIMD-like computational model (apply same transformation to many pixels)
  - One thread handles one pixel at a time
  - Analogy from lecture: a CPU is like math professors who can do complicated problems, a GPU is like an army of elementary schoolers that can do a lot of simple arithmetic

GOOD LUCK TO EVERYONE; YALL ARE GONNA DO AMAZING :) YOU MADE IT TO THE END OF THE SEMESTER ALMOST, YOU GOT THIS!!

Thanks for coming!

Give us your feedback:
https://forms.gle/Whbx4VjA8raMiZgX8