Programming on Multithreaded Architectures

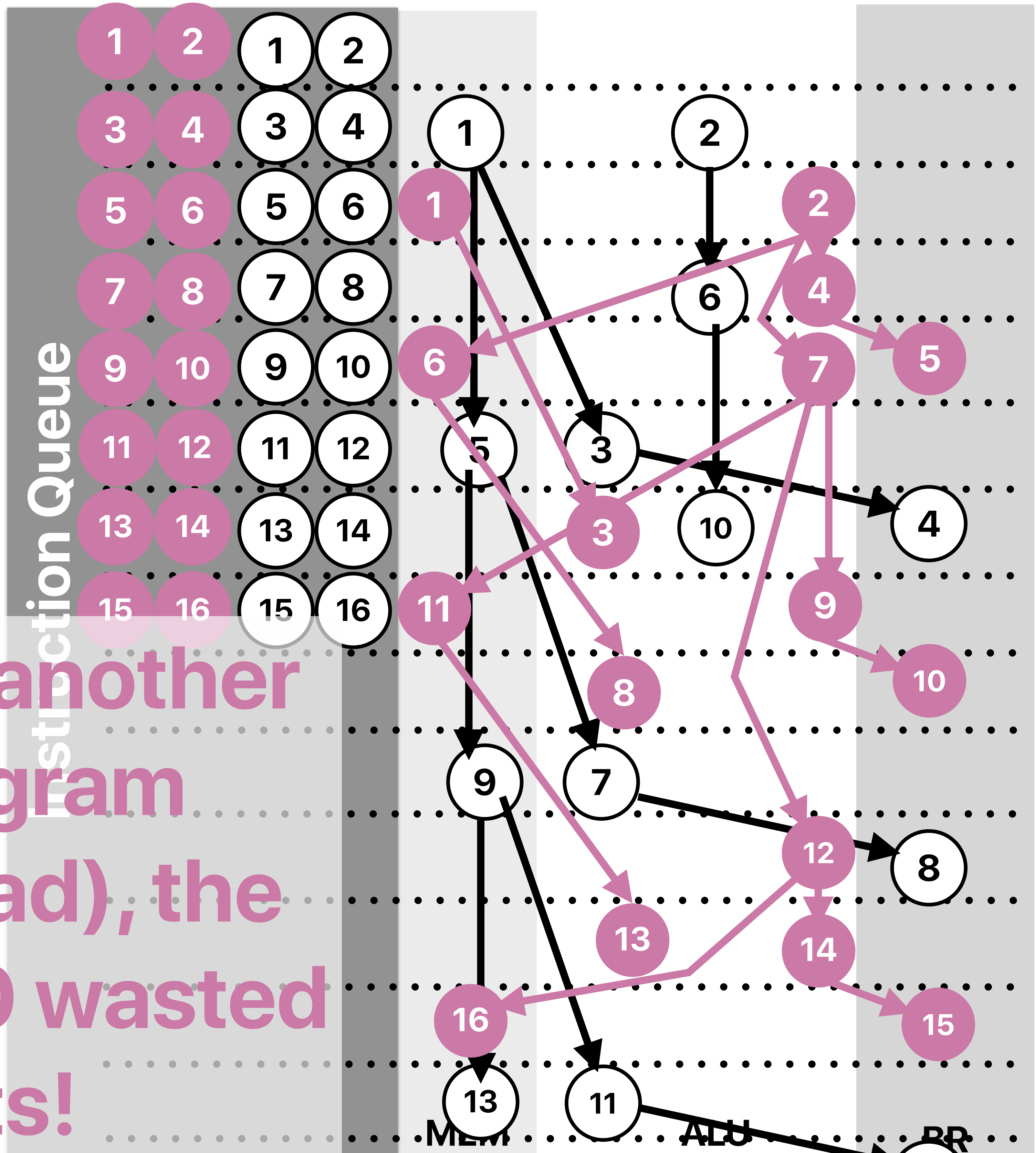
Hung-Wei Tseng

Recap: Parallelism in Modern Computers

- Instruction-level parallelism — concurrent execution of instructions from the same running program instance (process)
  - Superscalar
- Thread-level parallelism — concurrent execution of instructions from different running program instances
  - Simultaneous multithreading
  - Chip multiprocessor
- Data-level parallelism — concurrent execution of data streams from the same running program instance
  - Vector instructions (e.g., SSE, AVX)
  - GPU

Concept: Simultaneous Multithreading (SMT)

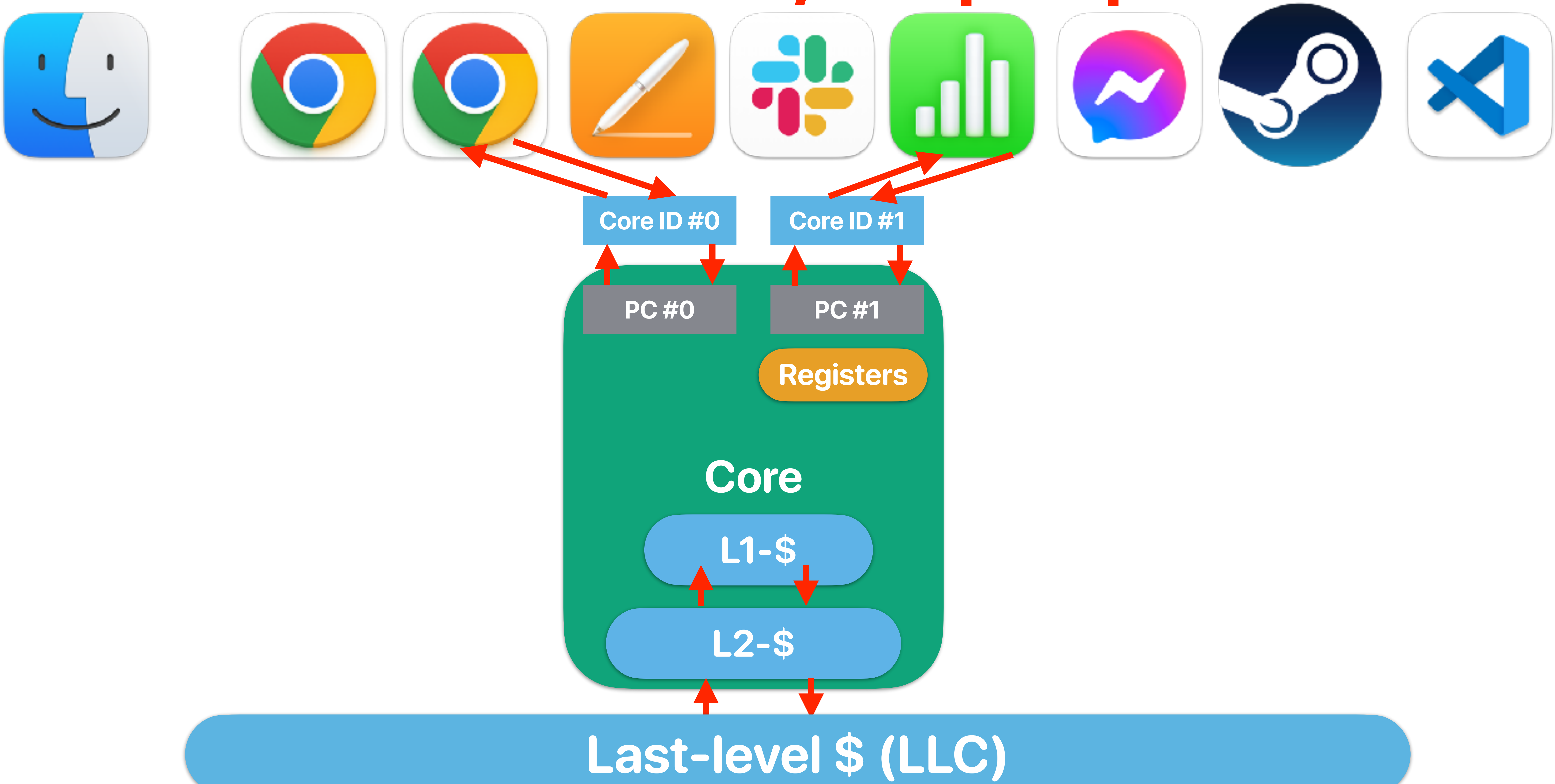
① movq 8(%rdi), %rdi
② addl $1, %eax
③ testq %rdi, %rdi
④ jne .L3
⑤ movq 8(%rdi), %rdi
⑥ addl $1, %eax
⑦ testq %rdi, %rdi
⑧ jne .L3
⑨ movq 8(%rdi), %rdi
⑩ addl $1, %eax
⑪ testq %rdi, %rdi
⑫ jne .L3



① movl (%rdi), %ecx
② addq $4, %rdi
③ addl %ecx, %eax
④ cmpq %rdx, %rdi
⑤ jne .L3
⑥ movl (%rdi), %ecx
⑦ addq $4, %rdi
⑧ addl %ecx, %eax
⑨ cmpq %rdx, %rdi
⑩ jne .L3
⑪ movl (%rdi), %ecx
⑫ addq $4, %rdi
⑬ addl %ecx, %eax
⑭ cmpq %rdx, %rdi
⑮ jne .L3
⑯ movl (%rdi), %ecx

By scheduling another running program instance (thread), the processor has 0 wasted issue slots!

SMT from the user/OS' perspective



Recap: Transistor counts

Microarchitecture	Transistor Count	Issue-width	Year
Alder Lake	325 M	5x ALU, 7x Memory	2021
Coffee Lake	217 M	4x ALU, 4x Memory	2017
Sandy Bridge	290 M	3x ALU, 3x Memory	2011
Nehalem	182.75 M	3x ALU, 3x Memory	2008

How many transistors per core on Coffee Lake?

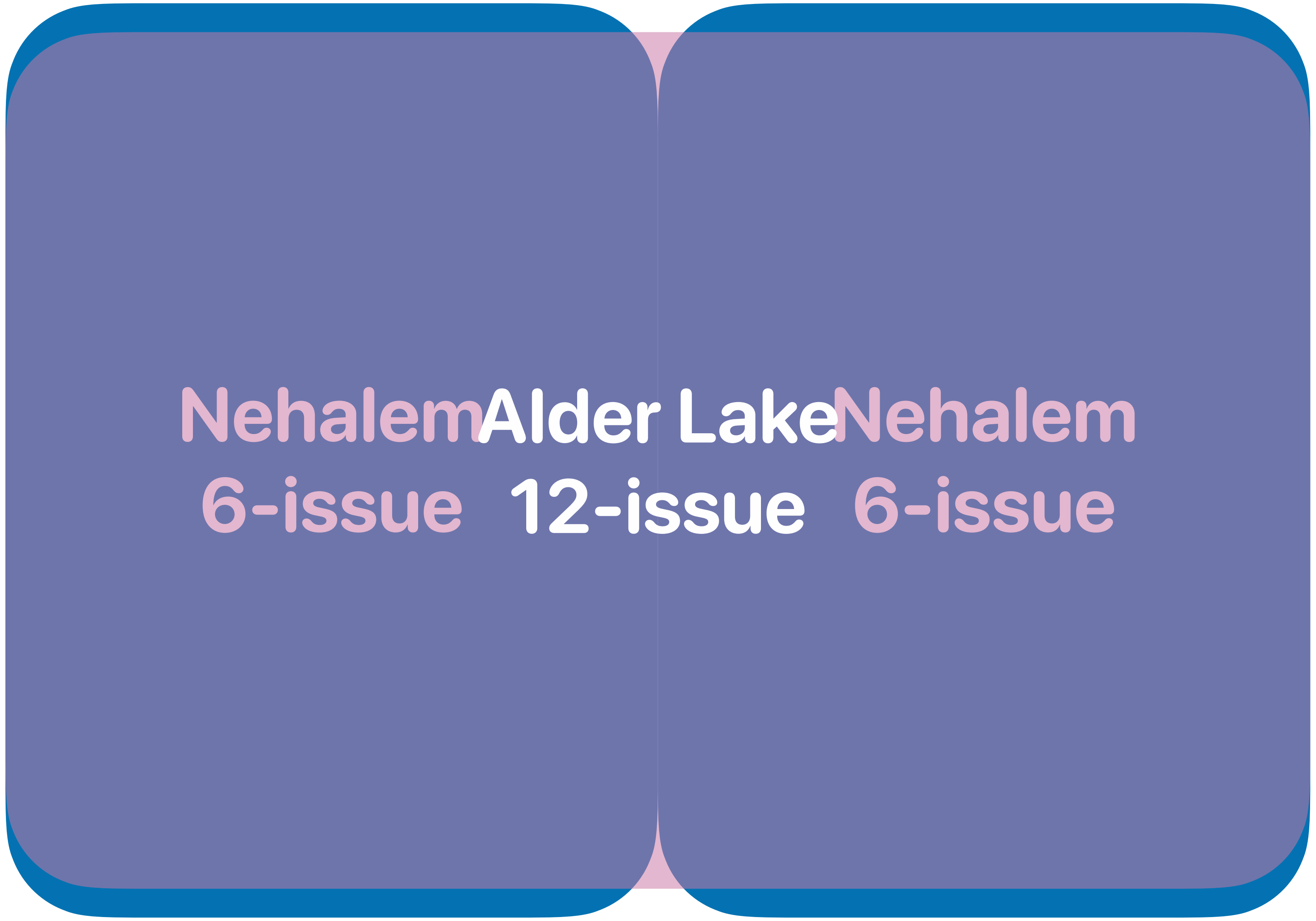
The Coffee Lake processor has 217 million transistors per core. It is manufactured using Intel's second 14 nm pr
Lake processors introduced i5 and i7 CPUs featuring six cores (along with hyper-threading in the case of the la
and no hyperthreading.

The transistor count per core on Coffee Lake is lower than that of some other modern processors, such as the
transistors per core. However, Coffee Lake still offers good performance, thanks to its high clock speeds and ef

Here is a table of the transistor counts per core for some other modern processors:

Processor	Transistors per core
Coffee Lake	217 million
Ryzen 5 5600X	390 million

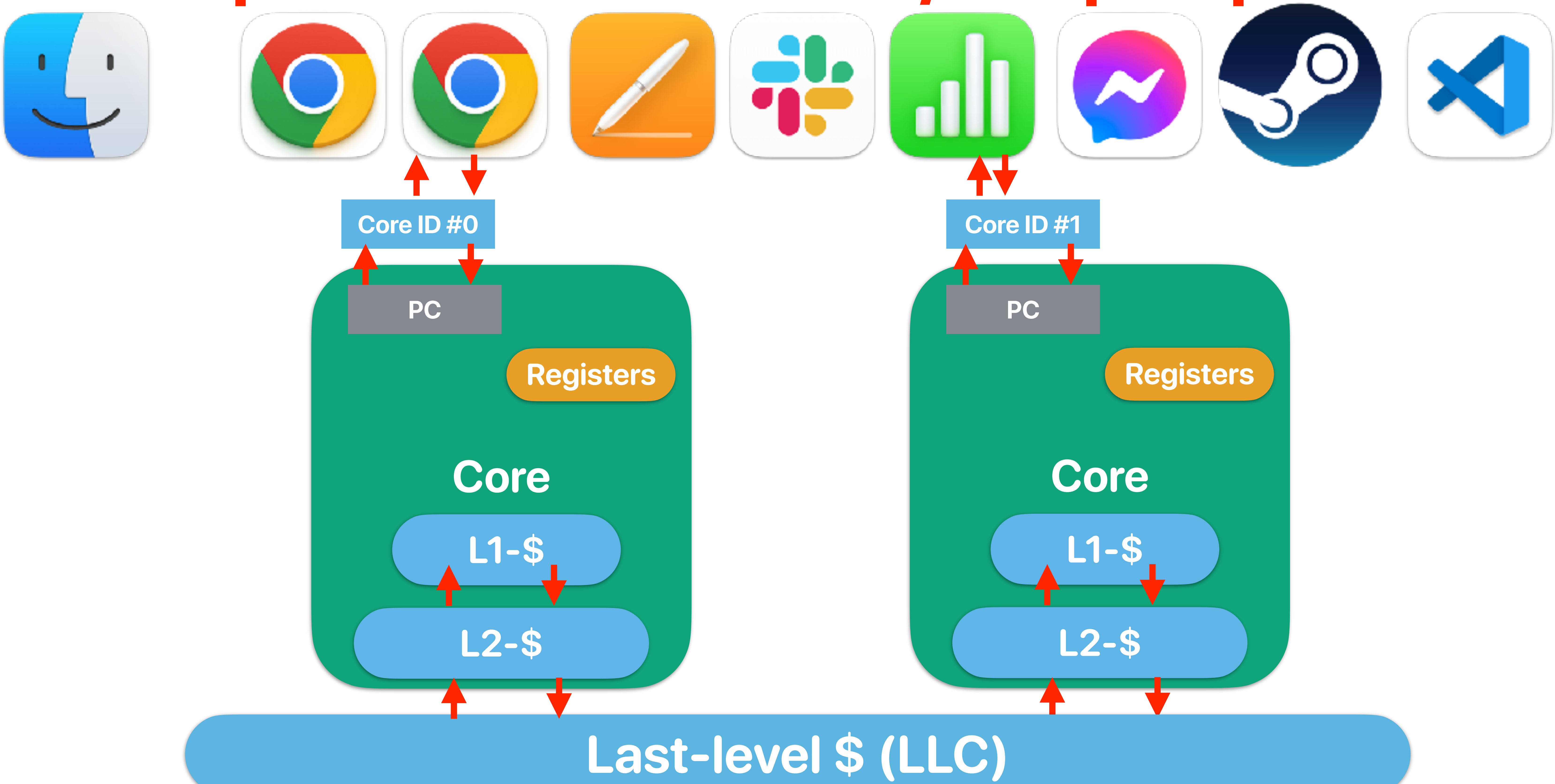
2x 3-issue ALUs Nehalem



1x 5-issue ALUs Alder Lake

Based on https://en.wikipedia.org/wiki/Transistor_count

Recap: CMP from the user/OS' perspective



SMT v.s. CMP

- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many of the following statements are/is correct when running programs on these processors.
  ① If we are just running one program in the system, the program will perform better on an SMT processor **— you have more resources for the program**
  ② If we are running 4 applications simultaneously, the cache miss rates will be higher in the SMT processor **— it depends!**
  ③ If we are running 4 applications simultaneously, the branch mis-prediction will be higher in the SMT processor **— it depends!**
  ④ If we are running one program with 4 parallel threads, the cache miss rates will be higher in the SMT processor **— it depends!**
  ⑤ If we are running one program with 4 parallel threads simultaneously, the branch mis-prediction will be longer in the SMT processor **— it depends!**

  A. 1
  B. 2
  C. 3
  D. 4
  E. 5

There is no clear win on each — why not having both?

The only thing we know for sure — if we don't parallel the program, it won't get any faster on CMP

Modern processors have both CMP/SMT

Outline

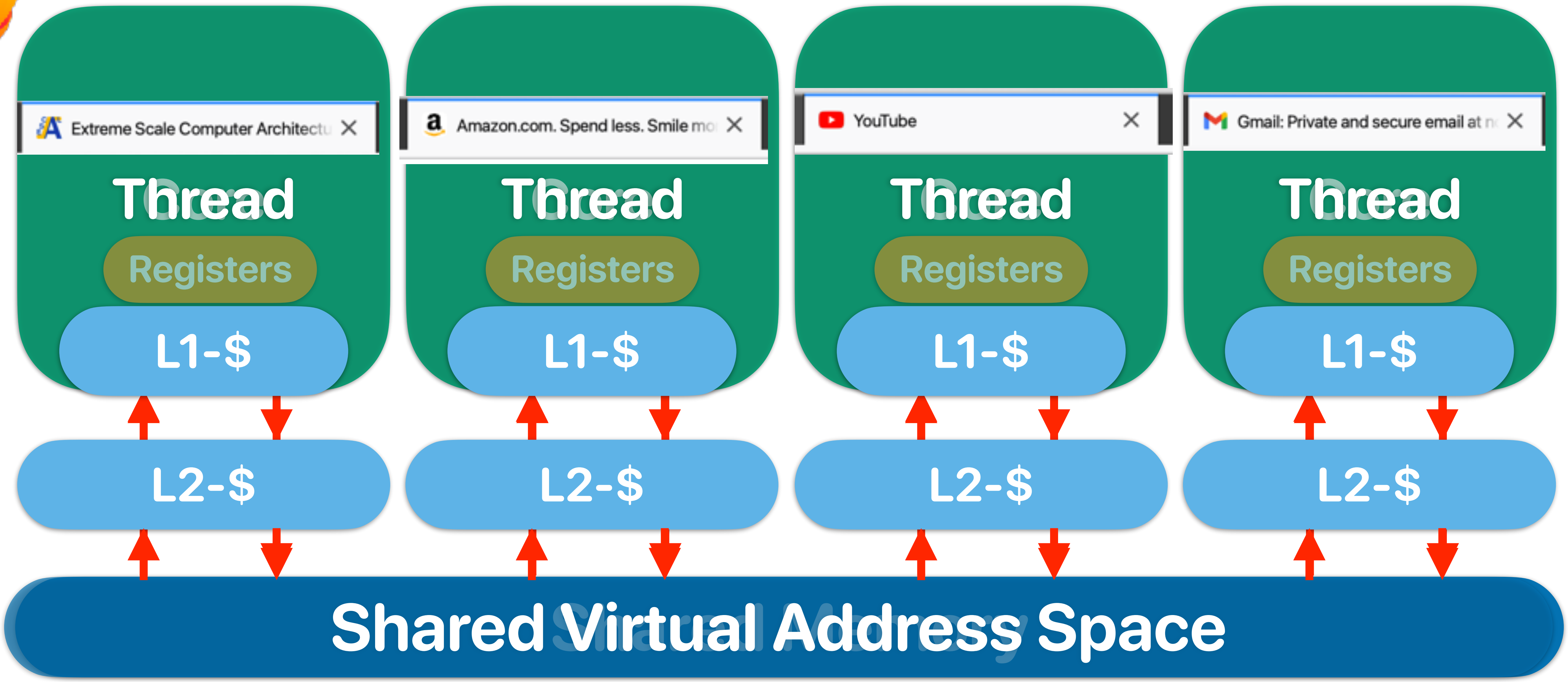
- Programming multithreaded processors & necessary architectural support (cont.)

Parallel Programming & Architectural Supports for Parallel Programming

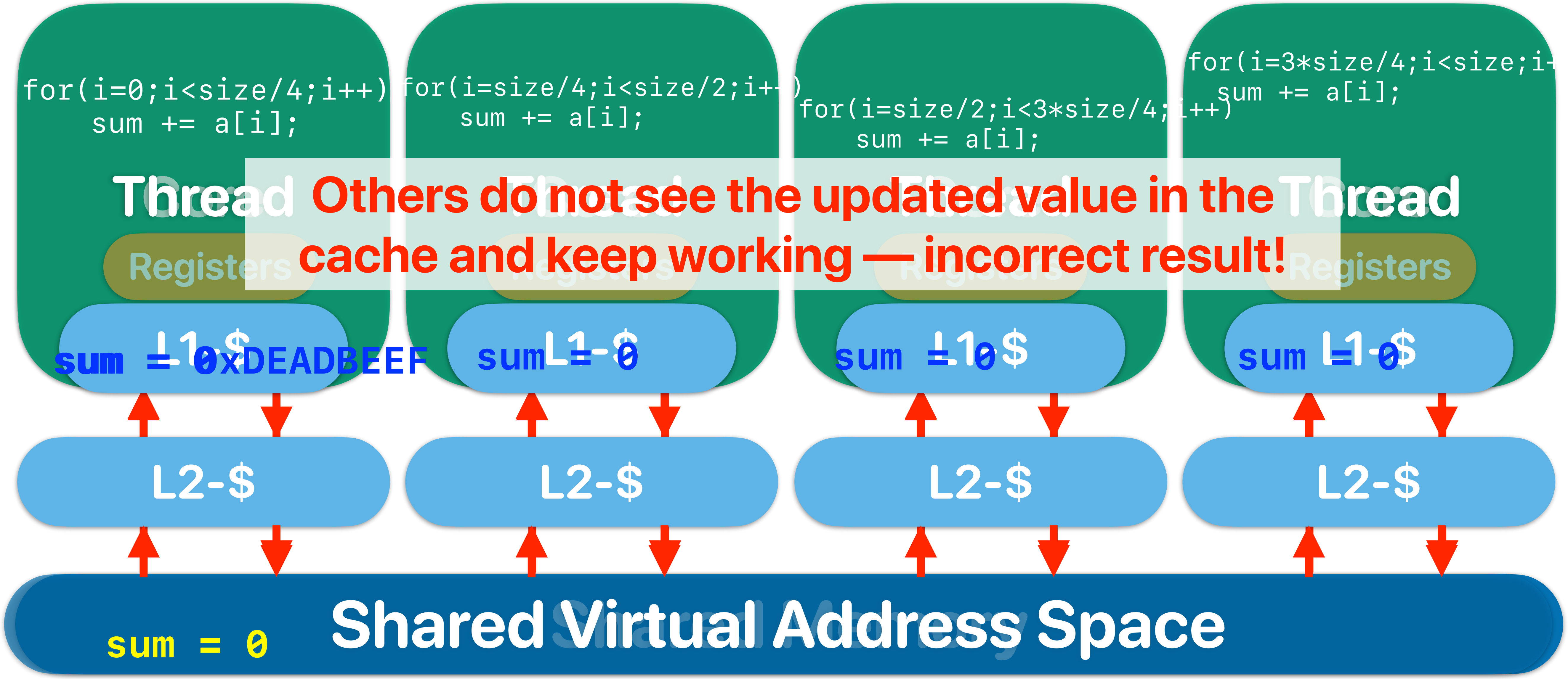
Parallel programming

- To exploit parallelism you need to break your computation into multiple “processes” or multiple “threads”
- Processes (in OS/software systems)
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs
- Threads (in OS/software systems)
  - Independent portions of your program that can run in parallel
  - All threads share the same virtual memory space
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)

What software thinks about "multiprogramming" hardware



What software thinks about "multiprogramming" hardware



Coherency & Consistency

- Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
  - What value should be seen
- Consistency — All threads see the change of data in the same order
  - When the memory operation should be done

Simple cache coherency protocol

- Snooping protocol
  - Each processor broadcasts / listens to cache misses
- State associate with each block (cacheline)
  - Invalid
    - The data in the current block is invalid
  - Shared
    - The processor can read the data
    - The data may also exist on other processors
  - Exclusive
    - The processor has full permission on the data
    - The processor is the only one that has up-to-date data

Coherent way-associative cache

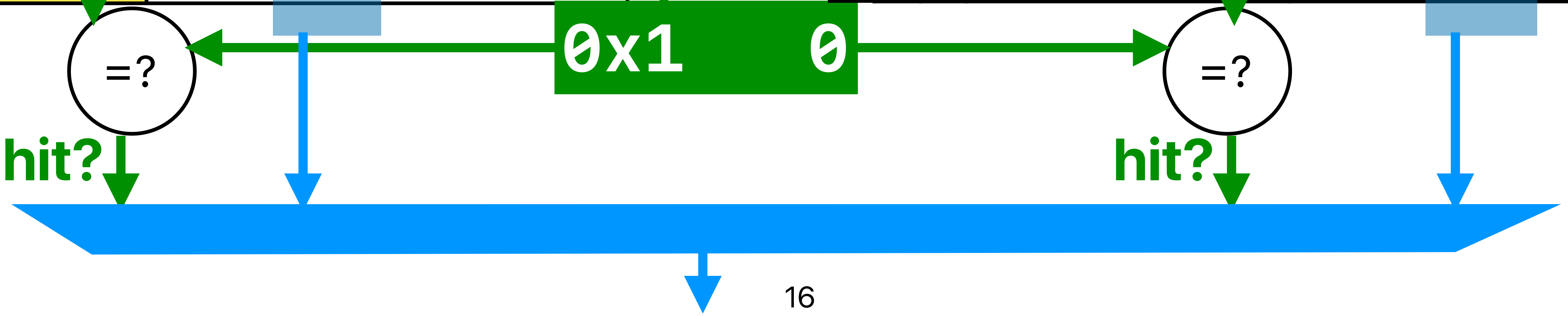
memory address: 0x0 8 2 4

tag, set index, block offset

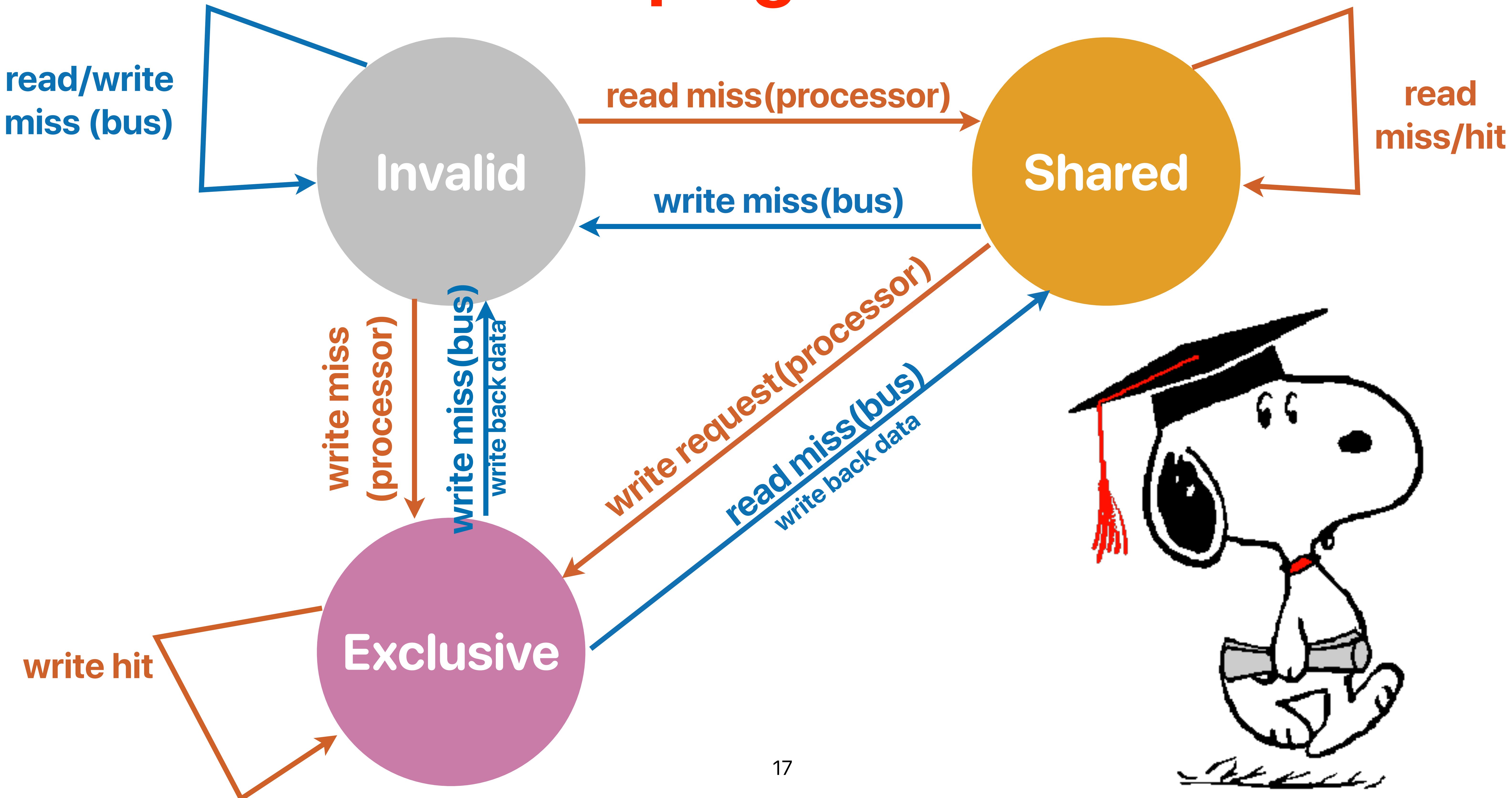
memory address: 0b0000100000100100

States	D	tag	data
01	1	0x29	IIJJKKLLMMNNOOPP
01	1	0xDE	QQRRSSTTUUVVWWXX
01	0	0x10	YYZZAABBCCDDEEFF
00	1	0x8A	AABBCCDDEEGGFFHH
10	1	0x60	IIJJKKLLMMNNOOPP
10	1	0x70	QQRRSSTTUUVVWWXX
10	1	0x10	QQRRSSTTUUVVWWXX
10	1	0x11	YYZZAABBCCDDEEFF

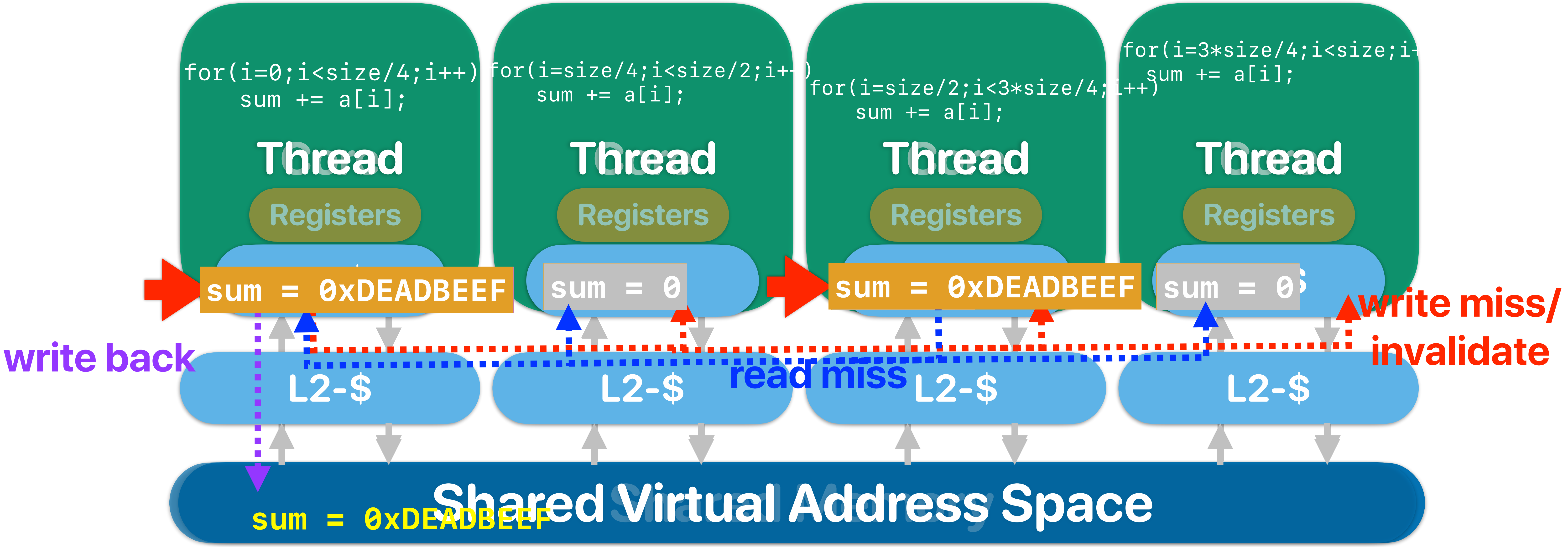
States	D	tag	data
01	1	0x00	AABBCCDDEEGGFFHH
01	1	0x10	IIJJKKLLMMNNOOPP
01	0	0xA1	QQRRSSTTUUVVWWXX
00	1	0x10	YYZZAABBCCDDEEFF
10	1	0x31	AABBCCDDEEGGFFHH
10	1	0x45	IIJJKKLLMMNNOOPP
10	1	0x41	QQRRSSTTUUVVWWXX
10	1	0x68	YYZZAABBCCDDEEFF



Snooping Protocol



What happens when we write in coherent caches?



Observer

thread 1	thread 2

thread 1

```c
int loop;

int main()
{
  pthread_t thread;
  loop = 1;

  pthread_create(&thread, NULL, modifyloop, 
NULL);
  while(loop == 1)
  {
    continue;
  }
  pthread_join(thread, NULL);
  fprintf(stderr,"User input: %d\n", loop);
  return 0;
}
```

thread 2

```c
void* modifyloop(void *x)
{
  sleep(1);
  printf("Please input a number:\n");
  scanf("%d",&loop);
  return NULL;
}
```

Observer

prevents the compiler from putting the variable “loop" in the “register”

thread 1	thread 2

```
volatile int loop;

int main()
{
  pthread_t thread;
  loop = 1;

  pthread_create(&thread, NULL, modifyloop,
NULL);
  while(loop == 1)
  {
    continue;
  }
  pthread_join(thread, NULL);
  fprintf(stderr,"User input: %d\n", loop);
  return 0;
}
```

```
void* modifyloop(void *x)
{
  sleep(1);
  printf("Please input a number:\n");
  scanf("%d",&loop);
  return NULL;
}
```

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

Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
`while(1)` `    printf("%d ",a);`	`while(1)` `    a++;`

① 0 1 2 3 4 5 6 7 8 9

② 1 2 5 9 3 6 8 10 12 13

③ 1 1 1 1 1 1 1 64 100

④ 1 1 1 1 1 1 1 1 1 100

A. 0

B. 1

C. 2

D. 3

E. 4

Cache coherency

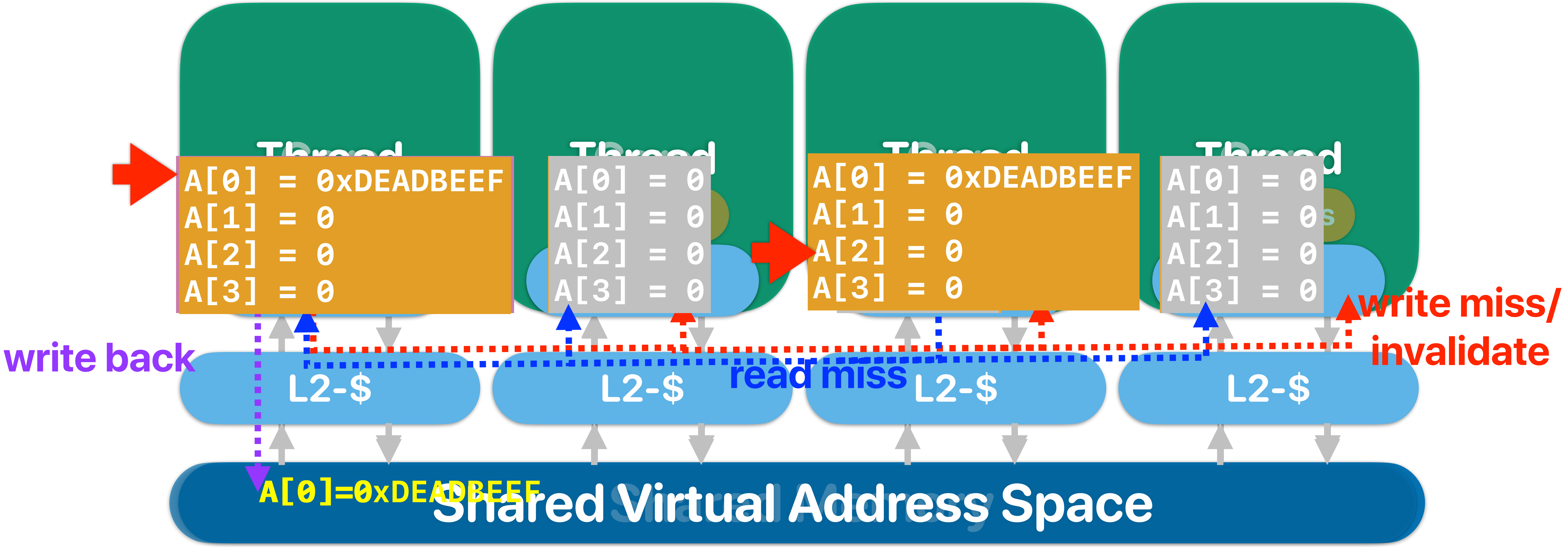
- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
`while(1)` `    printf("%d ",a);`	`while(1)` `    a++;`

① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100

A. 0
B. 1
C. 2
D. 3
E. 4

Cache coherency



Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
  tids[i] = i;
  pthread_create(&thread[i], NULL, threaded_vadd, &tids
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
  pthread_join(thread[i], NULL);
```

L v.s. R

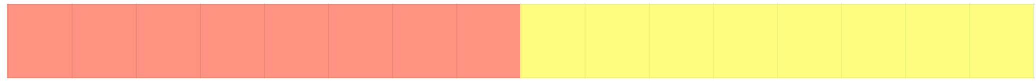
Version L

```c
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

C 

Version R

```c
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

C 

4Cs of cache misses

- 3Cs:
  - Compulsory, Conflict, Capacity
- Coherency miss:
  - A “block” invalidated because of the sharing among processors.

False sharing

- True sharing
  - Processor A modifies X, processor B also want to access X.
- False sharing
  - Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
  tids[i] = i;
  pthread_create(&thread[i], NULL, threaded_vadd, &tids
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
  pthread_join(thread[i], NULL);
```

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

Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
  a=1;
  x=b;
  return NULL;
}
void* modifyb(void *z) {
  b=1;
  y=a;
  return NULL;
}
```

```c
int main() {
  int i;
  pthread_t thread[2];
  pthread_create(&thread[0], NULL, modifya, NULL);
  pthread_create(&thread[1], NULL, modifyb, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  fprintf(stderr,"(%d, %d)\n",x,y);
  return 0;
}
```

Possible scenarios

Thread 1	Thread 2
a=1;	
	b=1;
	y=a;
x=b;	

(1,1)

Thread 1	Thread 2
a=1;	
x=b;	
	b=1;
	y=a;

(0,1)

Thread 1	Thread 2
	b=1;
	y=a;
a=1;	
x=b;	

(1,0)

Thread 1	Thread 2
	y=a;
x=b;	
a=1;	
	b=1;

OoO Scheduling!

(0,0)

Why (0,0)?

- Processor/compiler may reorder your memory operations/instructions
  - Coherence protocol can only guarantee the update of the same memory address
  - Processor can serve memory requests without cache miss first
  - Compiler may store values in registers and perform memory operations later
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it's spawned

Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
  a=1;
  x=b;
  return NULL;
}
void* modifyb(void *z) {
  b=1;
  y=a;
  return NULL;
}
```

```
int main() {
  int i;
  pthread_t thread[2];
  pthread_create(&thread[0], NULL, modifya, NULL);
  pthread_create(&thread[1], NULL, modifyb, NULL);
  pthread_join(thread[0], NULL);
  pthread_join(thread[1], NULL);
  fprintf(stderr,"(%d, %d)\n",x,y);
  return 0;
}
```

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
  - All updates prior to mfence must finish before the instruction can proceed
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

thread 1	thread 2

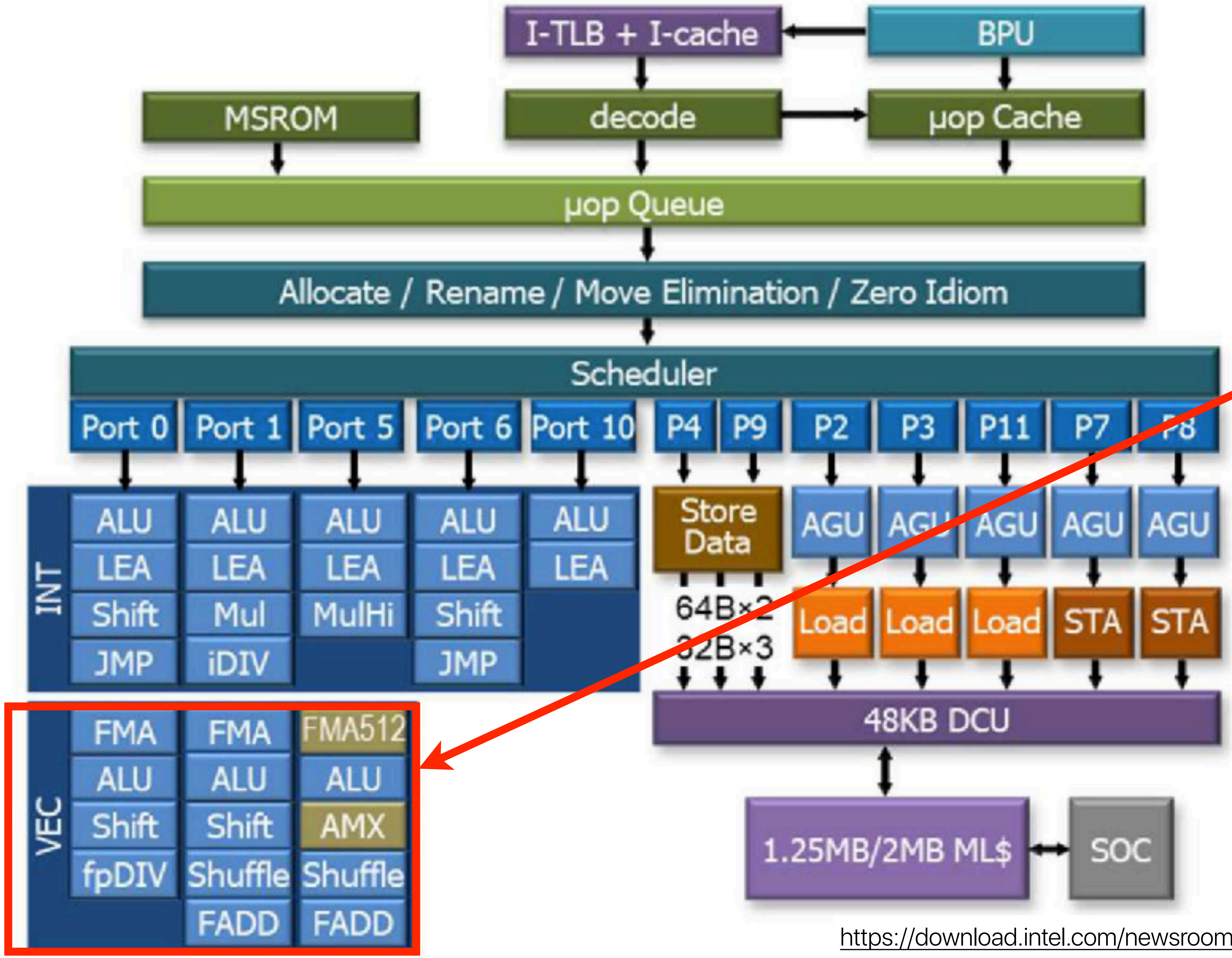
`a=1;`
`mfence` a=1 must occur/update before mfence
`x=b;`
	`b=1;`
	`mfence` b=1 must occur/update before mfence
	`y=a;`

Take-aways of parallel programming

- Processor behaviors are non-deterministic
  - You cannot predict which processor is going faster
  - You cannot predict when OS is going to schedule your thread
- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache consistency is hard to support

Data-Level Parallelism

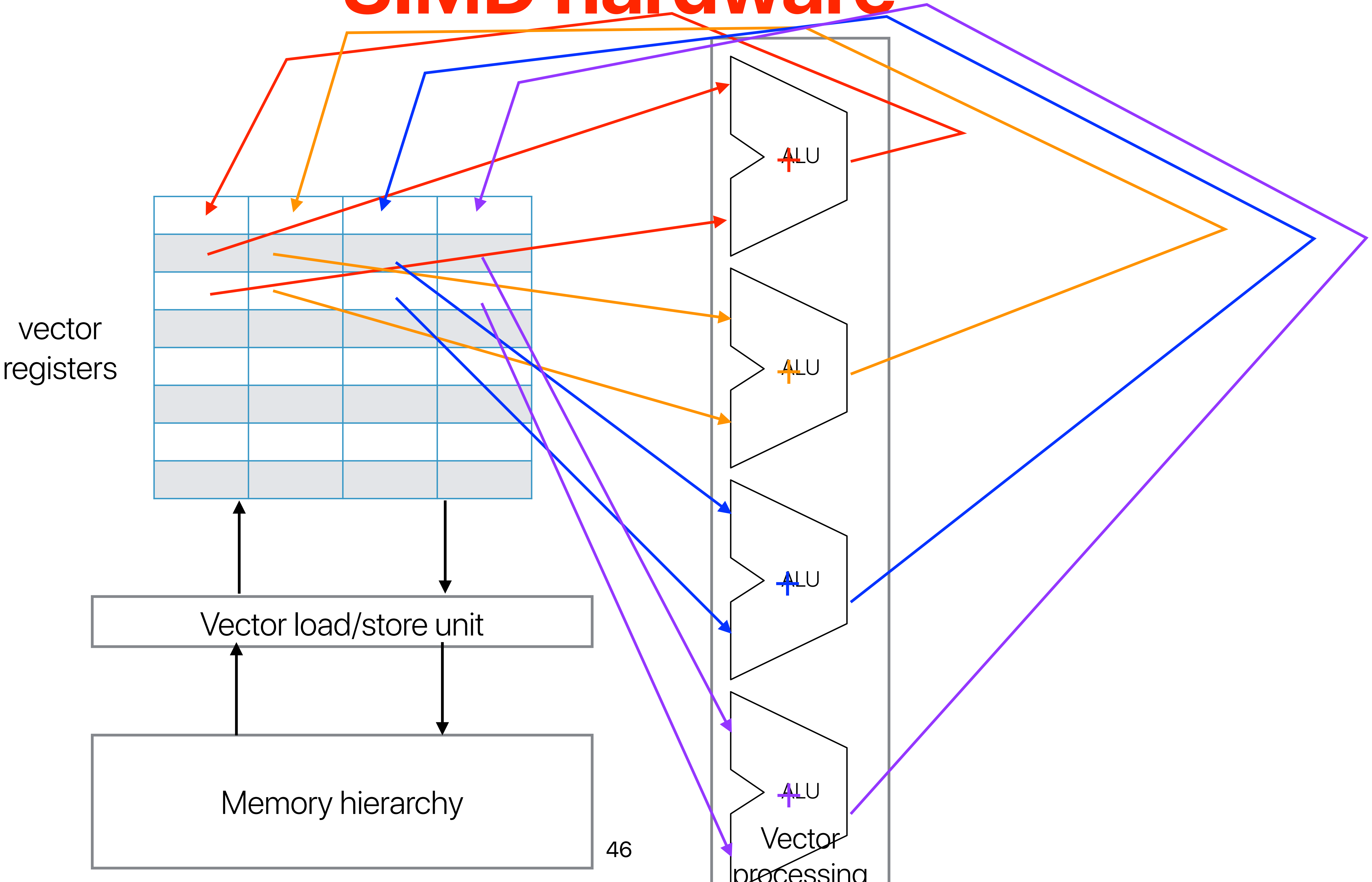
Intel Alder Lake (P-Core)



Data-Level Parallelism (DLP)

https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

SIMD hardware



x86's Streaming SIMD Extensions (SSE)

- SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers
  - Added 8 128-bit registers — XMM0-XMM7
  - You may use each register to store
    - 2 double-precision floating point numbers
    - 4 single-precision floating point numbers
  - Extended since introduced — SSE2, SSE3, SSE4, SSE4.1, SSE4.2, SSE4a
- They are processor-dependent instructions
  - AMD RyZen supports SSE4a, SSE4.1, SSE4.2
  - intel Core i7 doesn't support SSE4a
  - VIA Nano only support SSE4.1

Matrix multiplication with SSE4

```
void vector_blockmm(double **a, double **b, double **c)
{
  int i,j,k, ii, jj, kk, x;
  __m256d va, vb, vc; // compiler would allocate a register as long as these variables can fit
  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+=VECTOR_WIDTH) {
                      vc = _mm256_load_pd(&c[ii][jj]);           // load values into a vector register

                      for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                      {
                          va = _mm256_broadcast_sd(&a[ii][kk]); // load one value & fill the vector register
                          vb = _mm256_load_pd(&b[kk][jj]);      // load values into a vector register
                          vc = _mm256_add_pd(vc,_mm256_mul_pd(va,vb));// vector multiplication
                      }
                      _mm256_store_pd(&c[ii][jj],vc);      // store values into a vector register
                  }
              }
          }
      }
    }
}
```

Announcements

- Office hours — please check Google calendar
  - https://calendar.google.com/calendar/u/2?cid=Y18zNzNlYTdiYTFhZGlyNWRjYjQ0YzNhM2QxY2I2MmFmOTM0Zjc2MDE5NTUzODFjZGM4OTExNmQ5MTU5NmJhNGFmQGdyb3VwLmNhbGVuZGFyLmdvb2dsZS5jb20
  - You need to login @ucsd.edu account to read it
- Assignment #5 — due this Thursday 11:59pm
- Final exam this Friday @ 3pm—6pm
  - In-person — about 1.5x to 2x questions of the in-person midterm
  - Online session — about 1.5x to 2x questions of the online midterm
  - You should expect more open-ended questions like the SMT/CMP discussions
    - You need to first summarize your opinion
    - You need to provide evidence/citation to support your opinion

Computer
Science &
Engineering

つづく