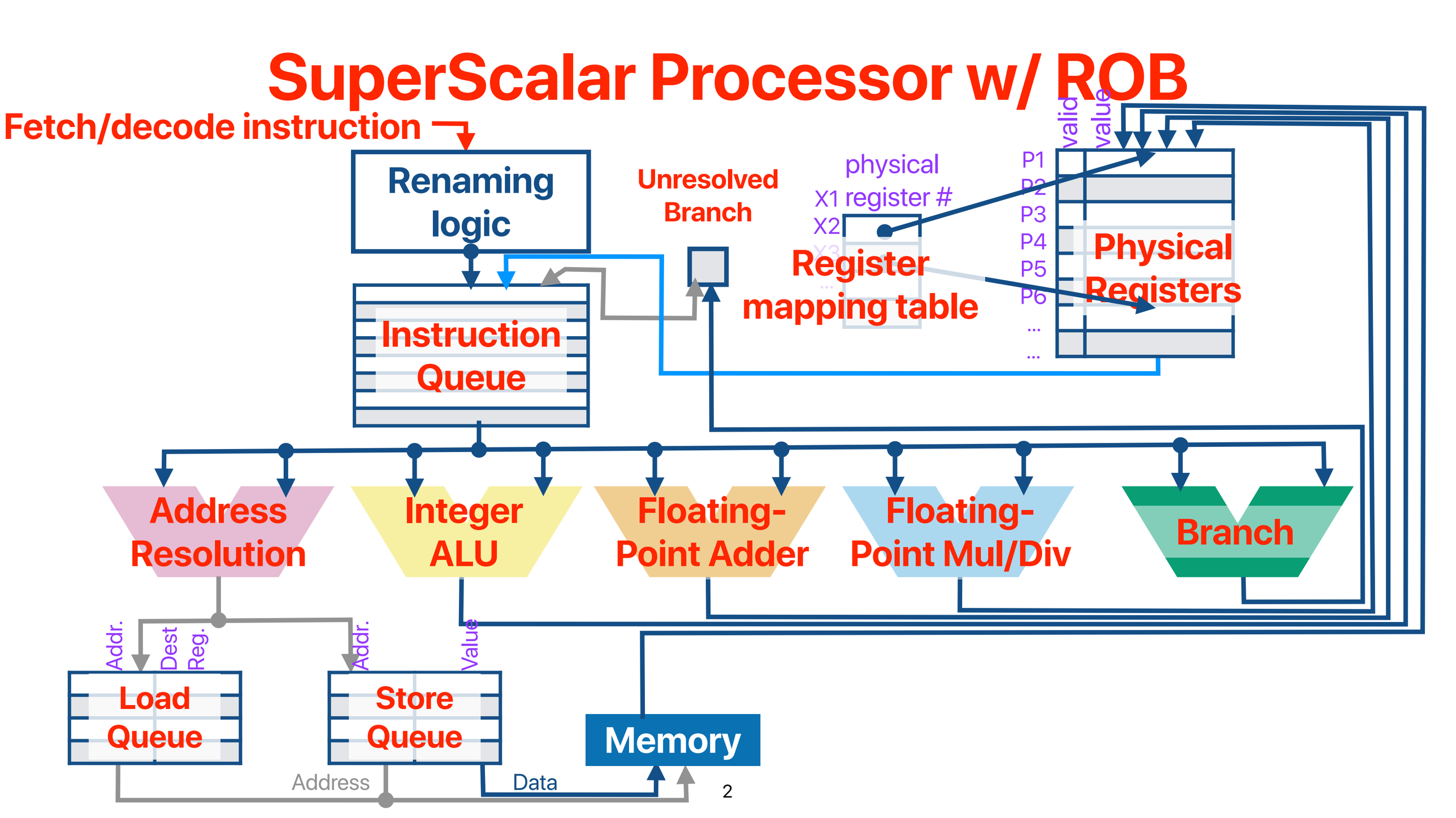
Multithreaded Architectures and Programming on Multithreaded Architectures

Hung-Wei

SuperScalar Processor w/ ROB



Recap: What about “linked list”

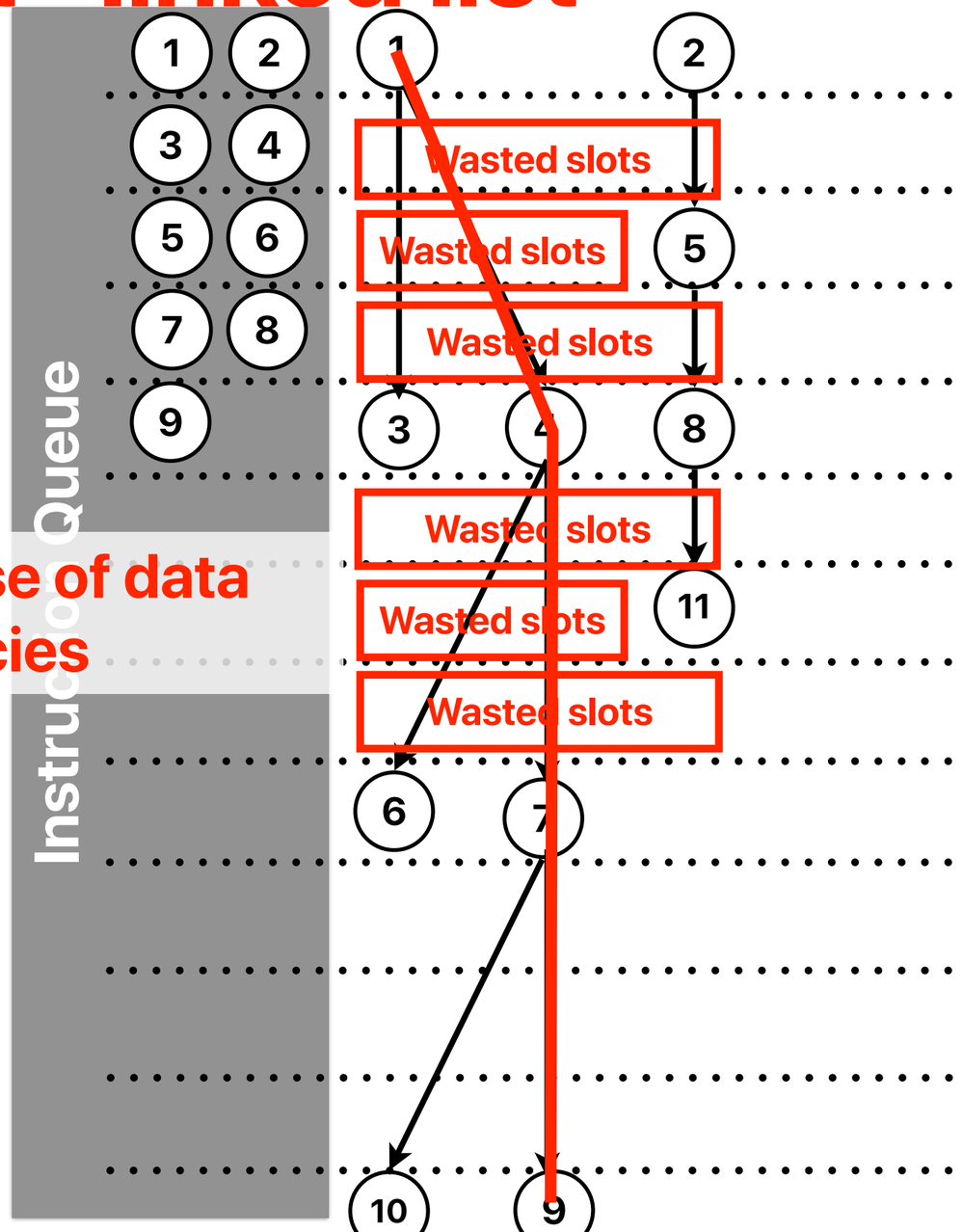
Static instructions

```
LOOP: ld   X10, 8(X10)
      addi  X7, X7, 1
      bne  X10, X0, LOOP
```

Dynamic instructions

```
① ld   X10, 8(X10)
② addi  X7, X7, 1
③ bne  X10, X0, LOOP
④ ld   X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne  X10, X0, LOOP
⑦ ld   X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne  X10, X0, LOOP
```

ILP is low because of data dependencies

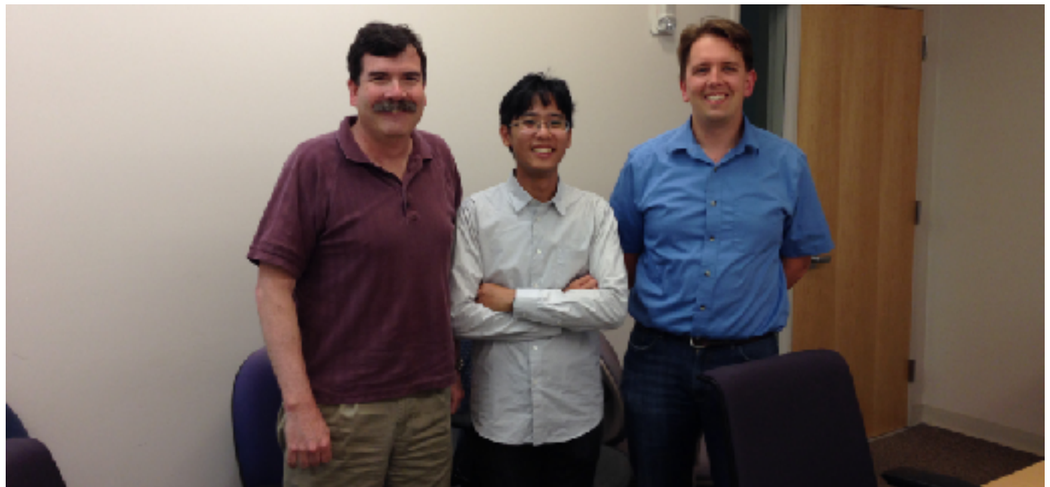


Demo: ILP within a program

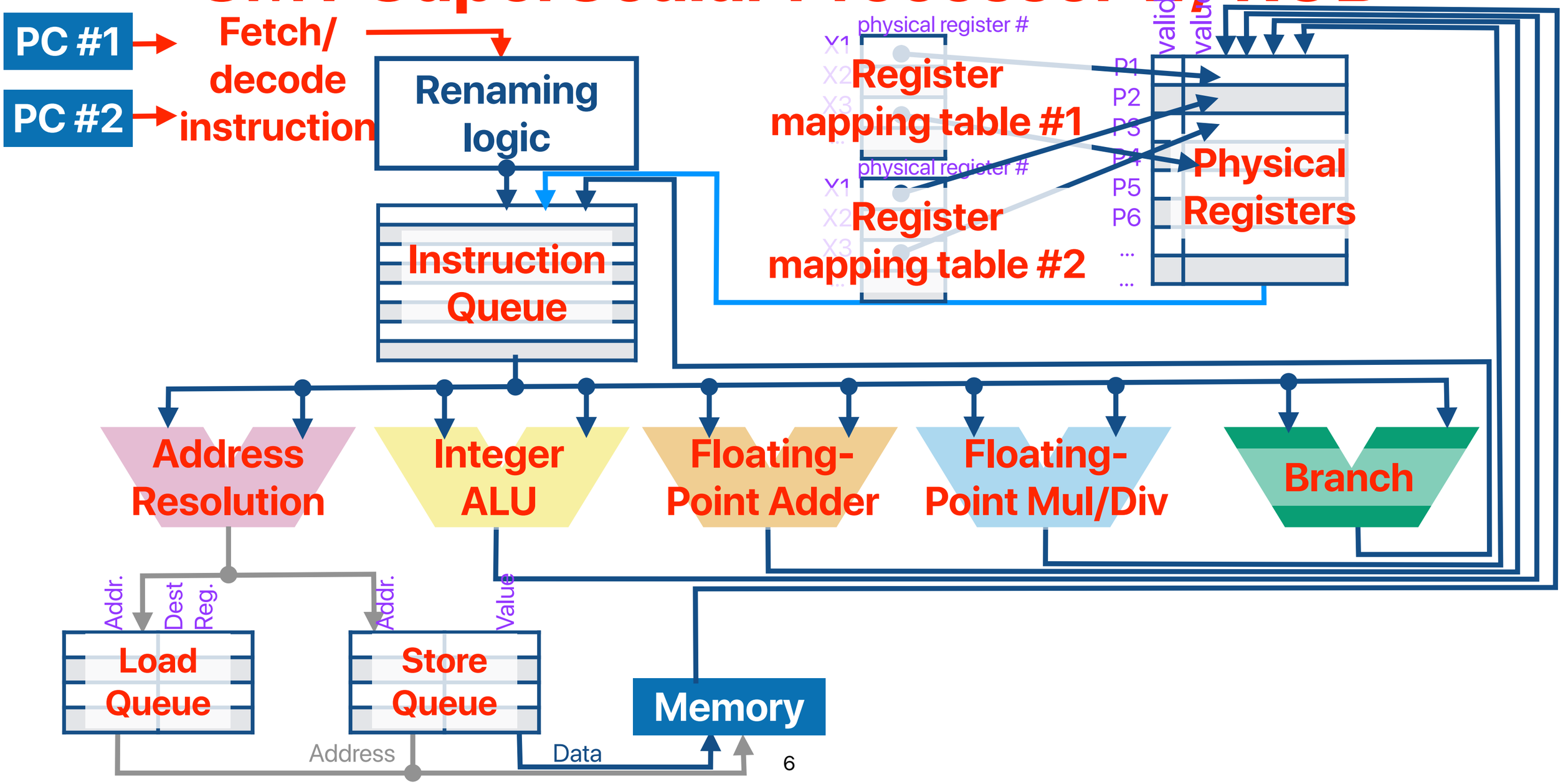
- perf is a tool that captures performance counters of your processors and can generate results like branch mis-prediction rate, cache miss rates and ILP.

Simultaneous multithreading: maximizing on-chip parallelism

Dean M. Tullsen, Susan J. Eggers, Henry M. Levy
Department of Computer Science and Engineering, University of Washington



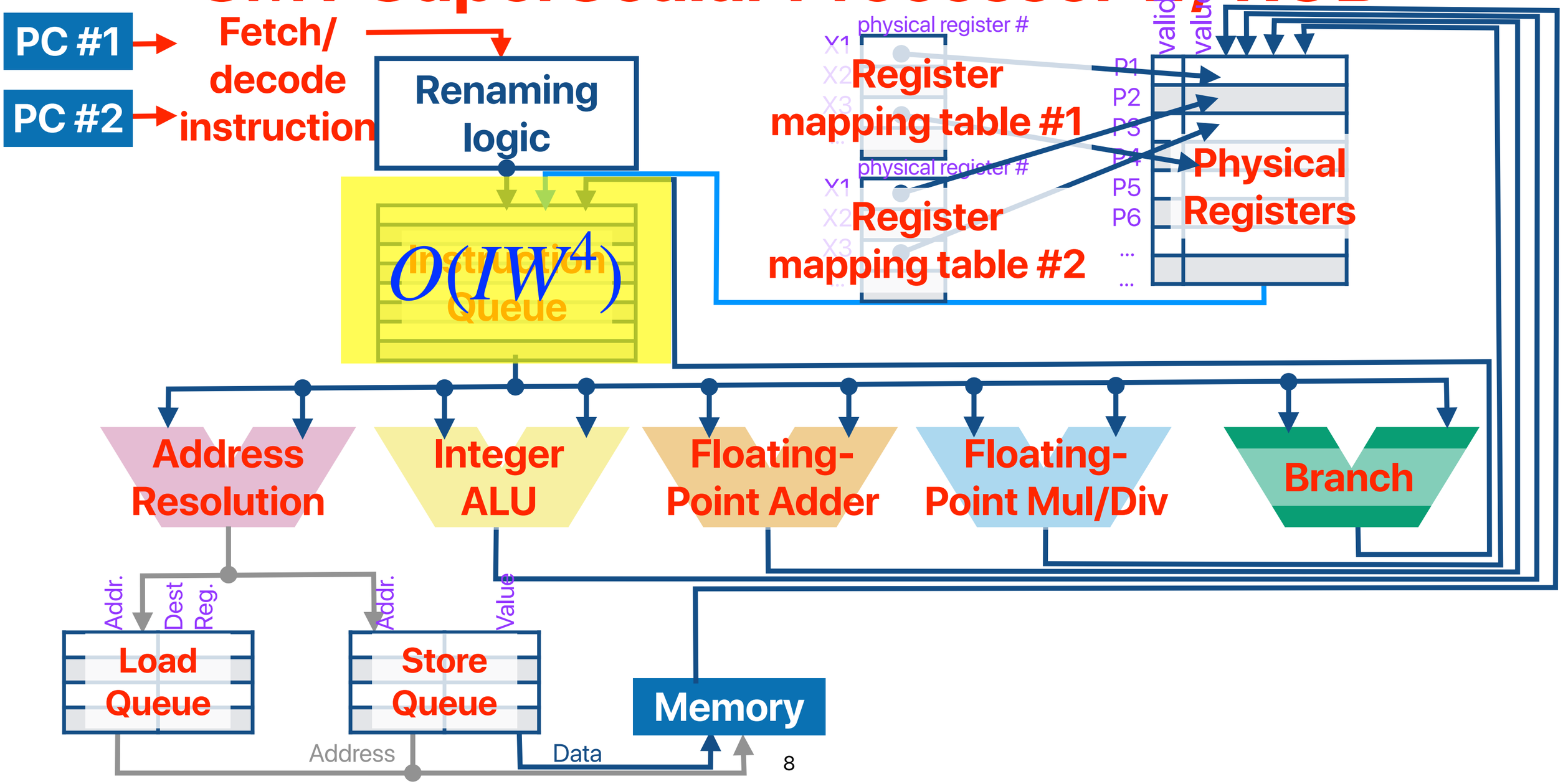
SMT SuperScalar Processor w/ ROB



SMT

- Improve the throughput of execution
  - May increase the latency of a single thread
- Less branch penalty per thread
- Increase hardware utilization
- Simple hardware design: Only need to duplicate PC/Register Files
- Real Case:
  - Intel HyperThreading (supports up to two threads per core)
    - Intel Pentium 4, Intel Atom, Intel Core i7
  - AMD RyZen

SMT SuperScalar Processor w/ ROB



Outline

- Chip Multiprocessor
- Programming in multithreaded architectures

Wider-issue processors won't give you much more

Program	IPC	BP Rate %	I cache %MPCI	D cache %MPCI	L2 cache %MPCI
compress	0.9	85.9	0.0	3.5	1.0
eqntott	1.3	79.8	0.0	0.8	0.7
m88ksim	1.4	91.7	2.2	0.4	0.0
MPsim	0.8	78.7	5.1	2.3	2.3
applu	0.9	79.2	0.0	2.0	1.7
apsi	0.6	95.1	1.0	4.1	2.1
swim	0.9	99.7	0.0	1.2	1.2
tomcatv	0.8	99.6	0.0	7.7	2.2
pmake	1.0	86.2	2.3	2.1	0.4

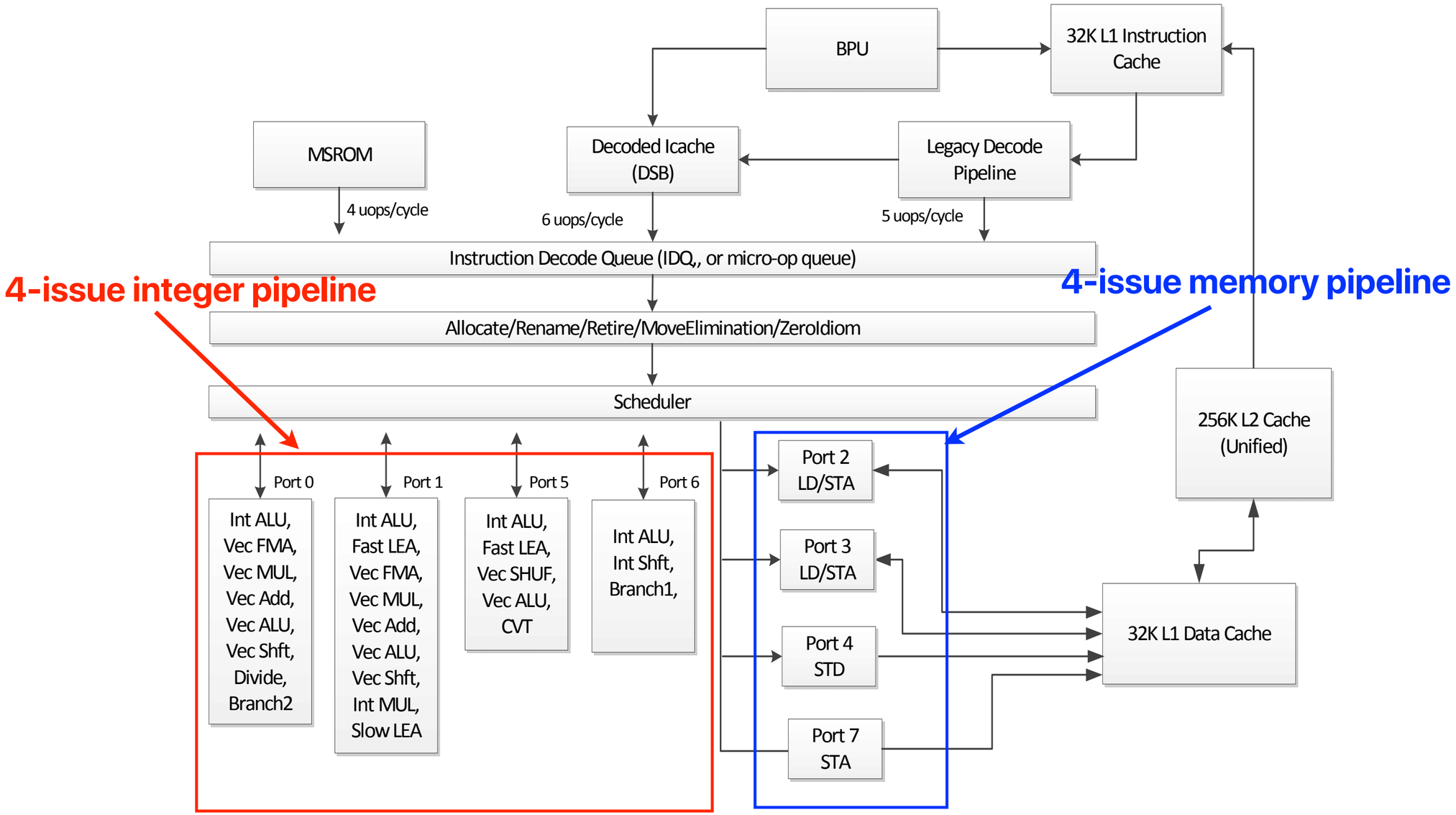
Table 5. Performance of a single 2-issue superscalar processor.

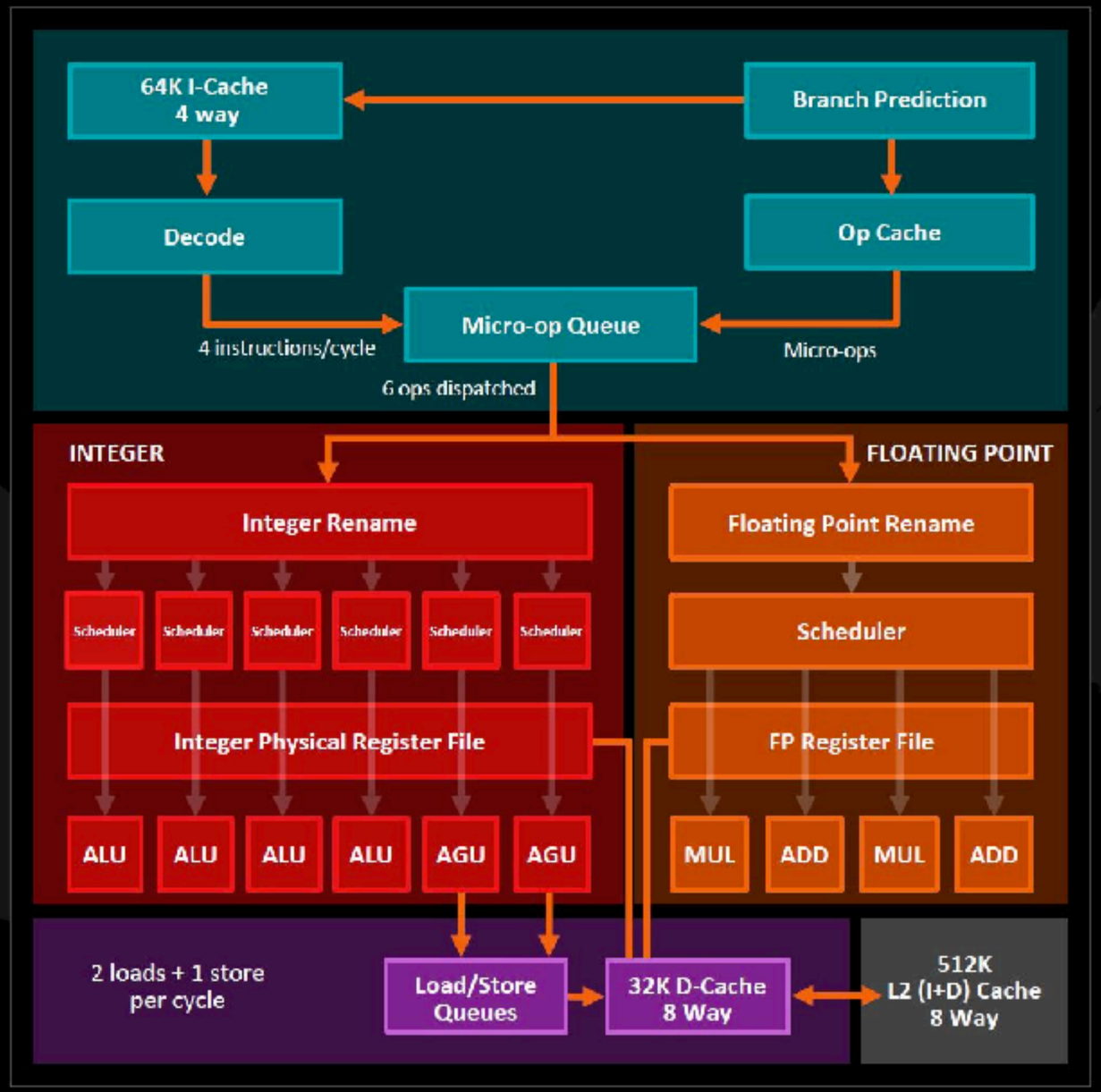
Program	IPC	BP Rate %	I cache %MPCI	D cache %MPCI	L2 cache %MPCI
compress	1.2	86.4	0.0	3.9	1.1
eqntott	1.8	80.0	0.0	1.1	1.1
m88ksim	2.3	92.6	0.1	0.0	0.0
MPsim	1.2	81.6	3.4	1.7	2.3
applu	1.7	79.7	0.0	2.8	2.8
apsi	1.2	95.6	0.2	3.1	2.6
swim	2.2	99.8	0.0	2.3	2.5
tomcatv	1.3	99.7	0.0	4.2	4.3
pmake	1.4	82.7	0.7	1.0	0.6

Table 6. Performance of the 6-issue superscalar processor.

The pipelines of Modern Processors

Intel Skylake

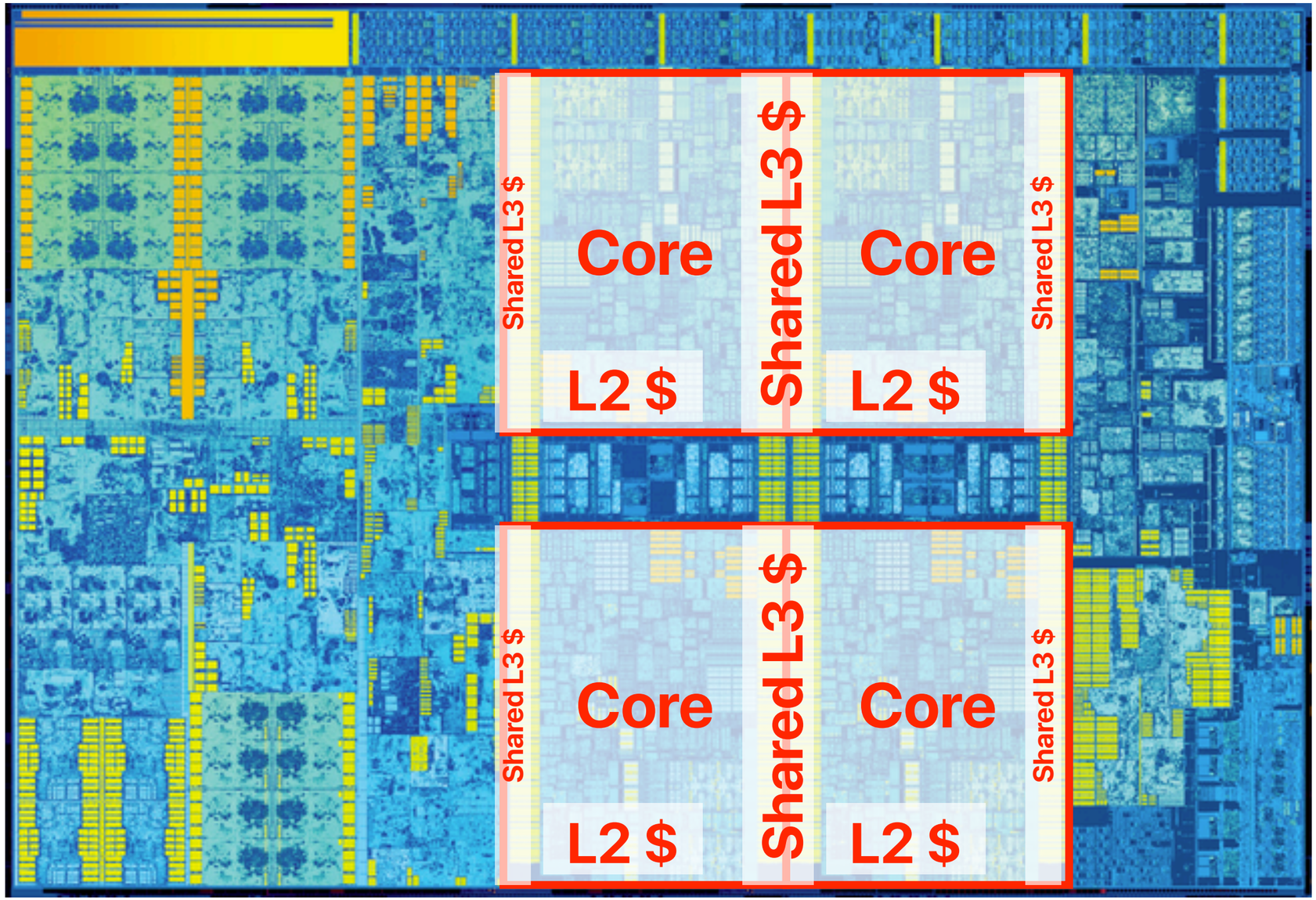


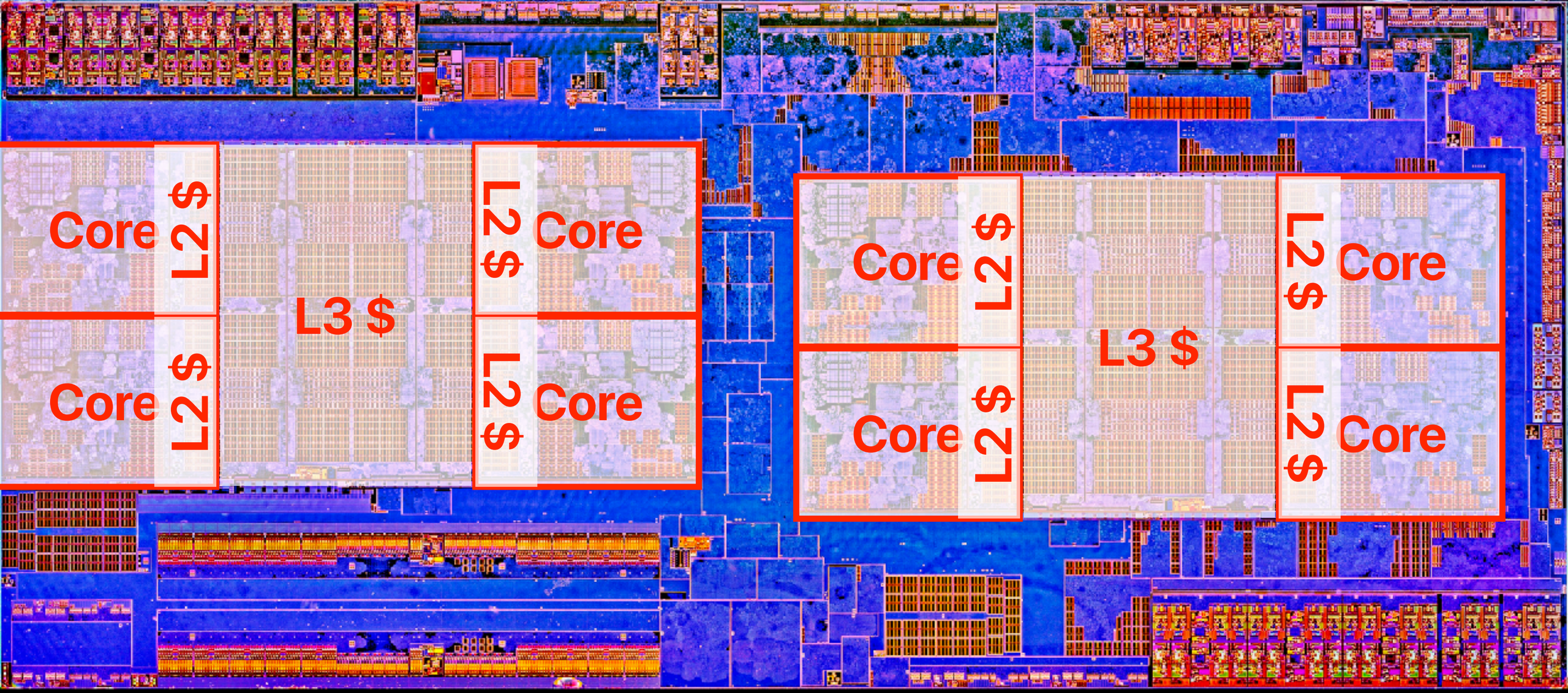


ZEN MICROARCHITECTURE

- Fetch Four x86 instructions
- Op Cache instructions
- 4 Integer units
  - Large rename space – 168 Registers
  - 192 instructions in flight/8 wide retire
- 2 Load/Store units
  - 72 Out-of-Order Loads supported
- 2 Floating Point units x 128 FMACs
  - built as 4 pipes, 2 Fadd, 2 Fmul
- I-Cache 64K, 4-way
- D-Cache 32K, 8-way
- L2 Cache 512K, 8-way
- Large shared L3 cache
- 2 threads per core

Intel SkyLake




Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core

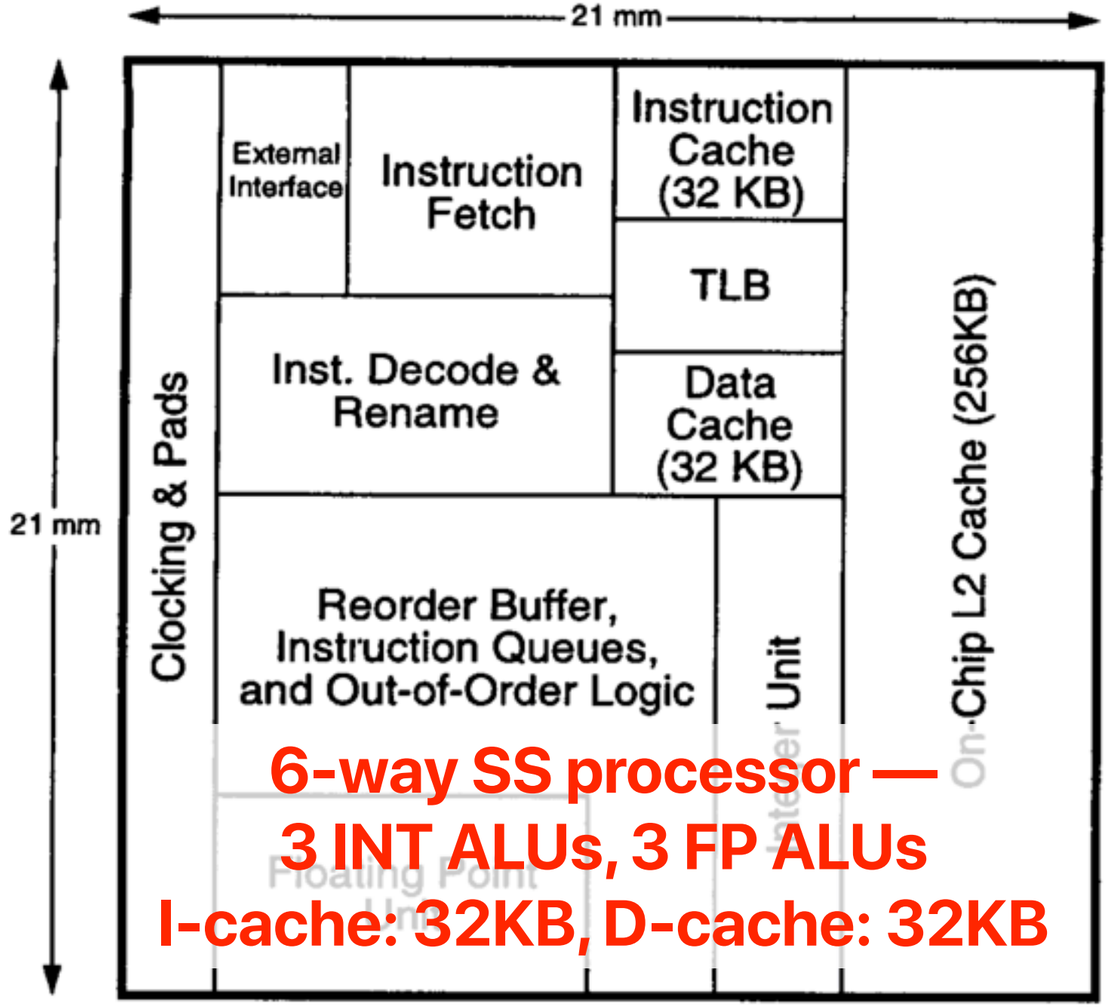
AMD

RYZEN

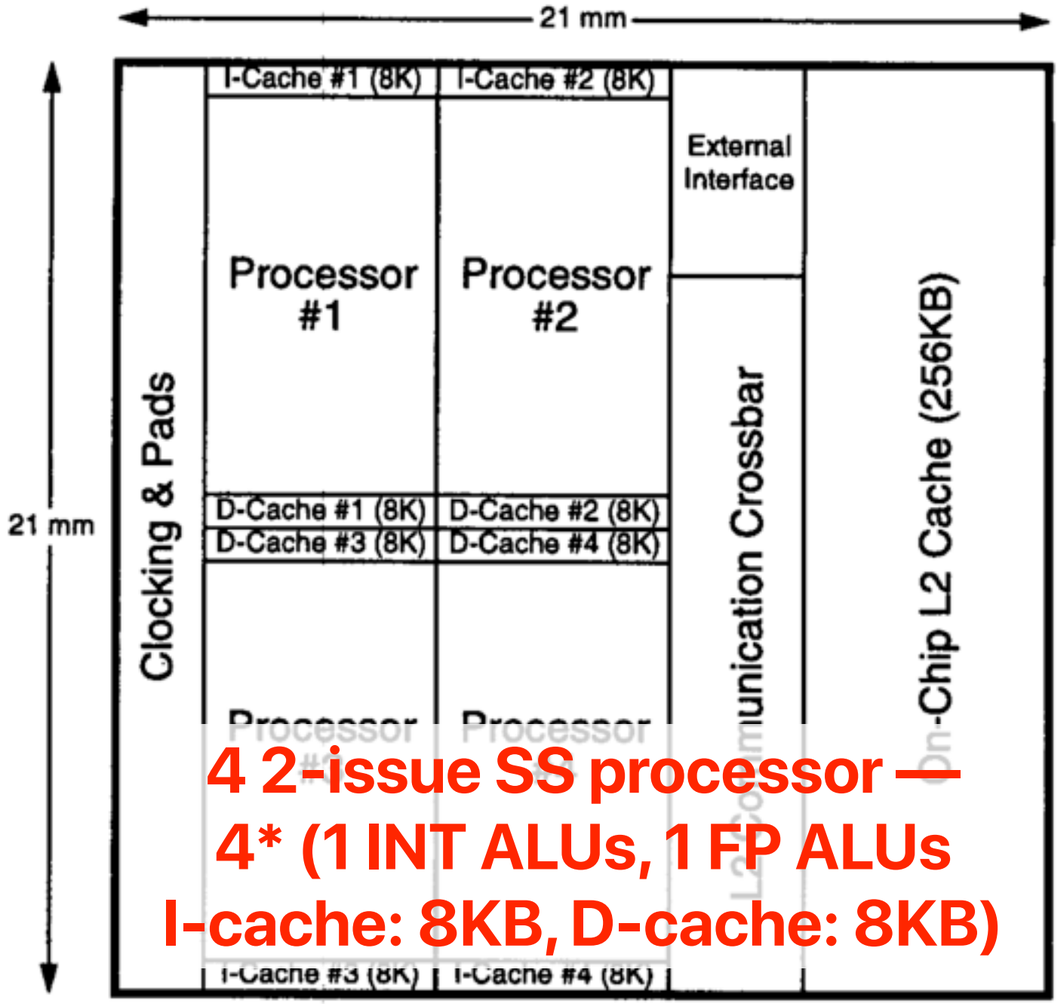
The case for a Single-Chip Multiprocessor

Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang
Stanford University

Wide-issue SS processor v.s. multiple narrower-issue SS processors

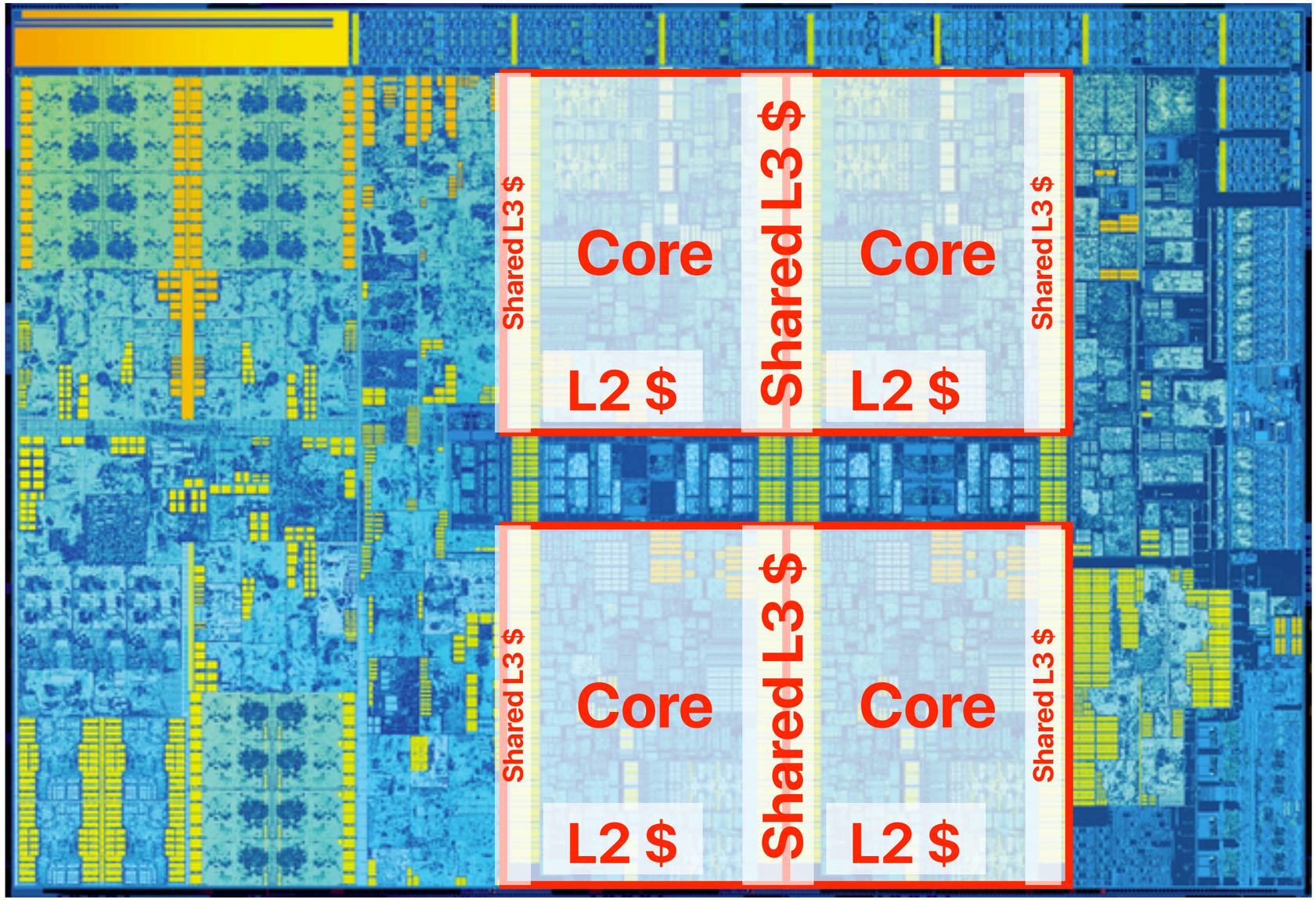


6-way SS processor —
3 INT ALUs, 3 FP ALUs
I-cache: 32KB, D-cache: 32KB

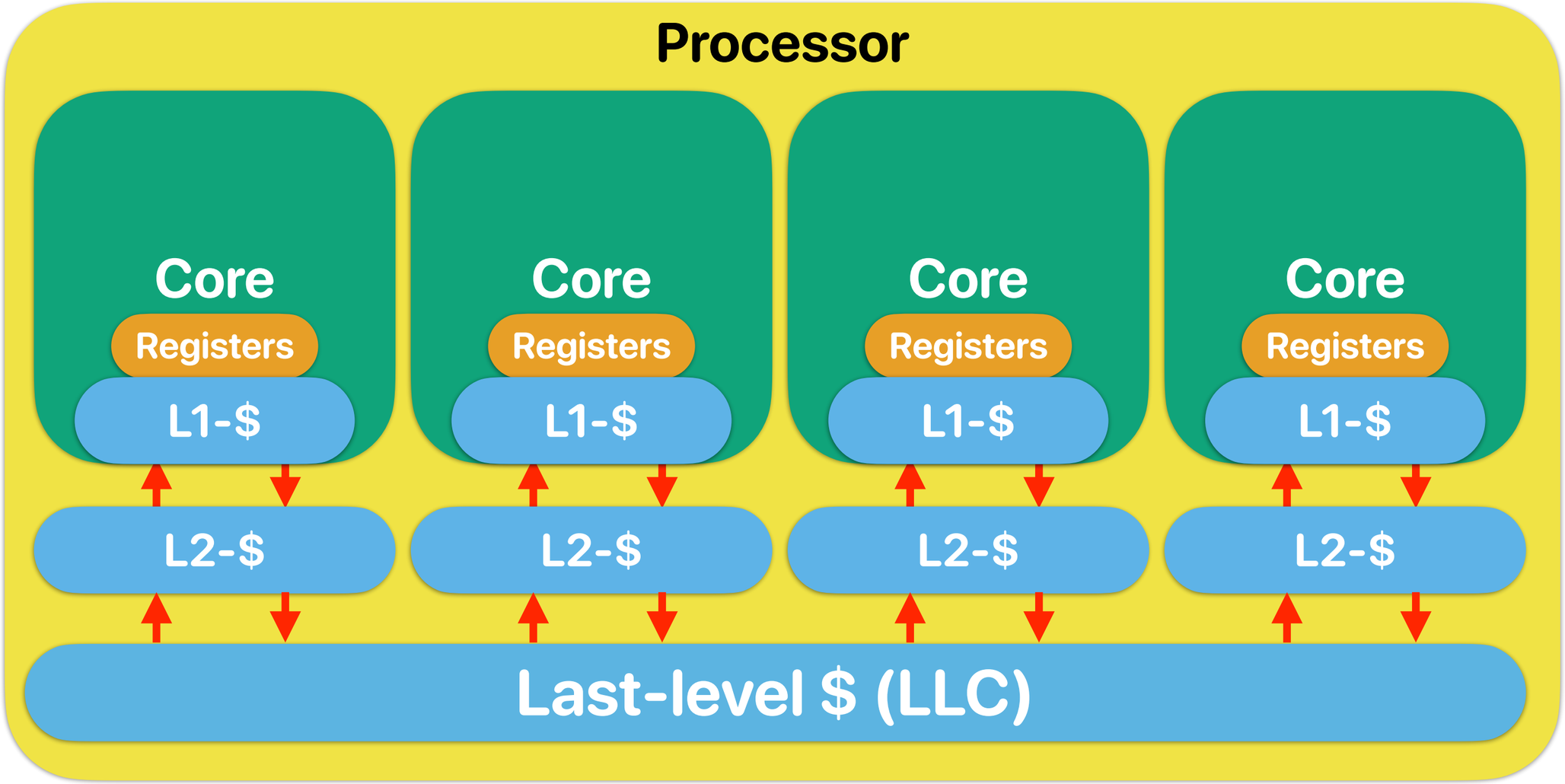


4 2-issue SS processor —
4* (1 INT ALUs, 1 FP ALUs
I-cache: 8KB, D-cache: 8KB)

Intel SkyLake



Concept of CMP



Poll close in 1:30

SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

  P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

  P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

  Which one do you think is better?

  A. P1
  B. P2

Poll close in 1:30

SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

  P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

  P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

  Which one do you think is better?

  A. P1
  B. P2

SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

  P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

  P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

  Which one do you think is better?

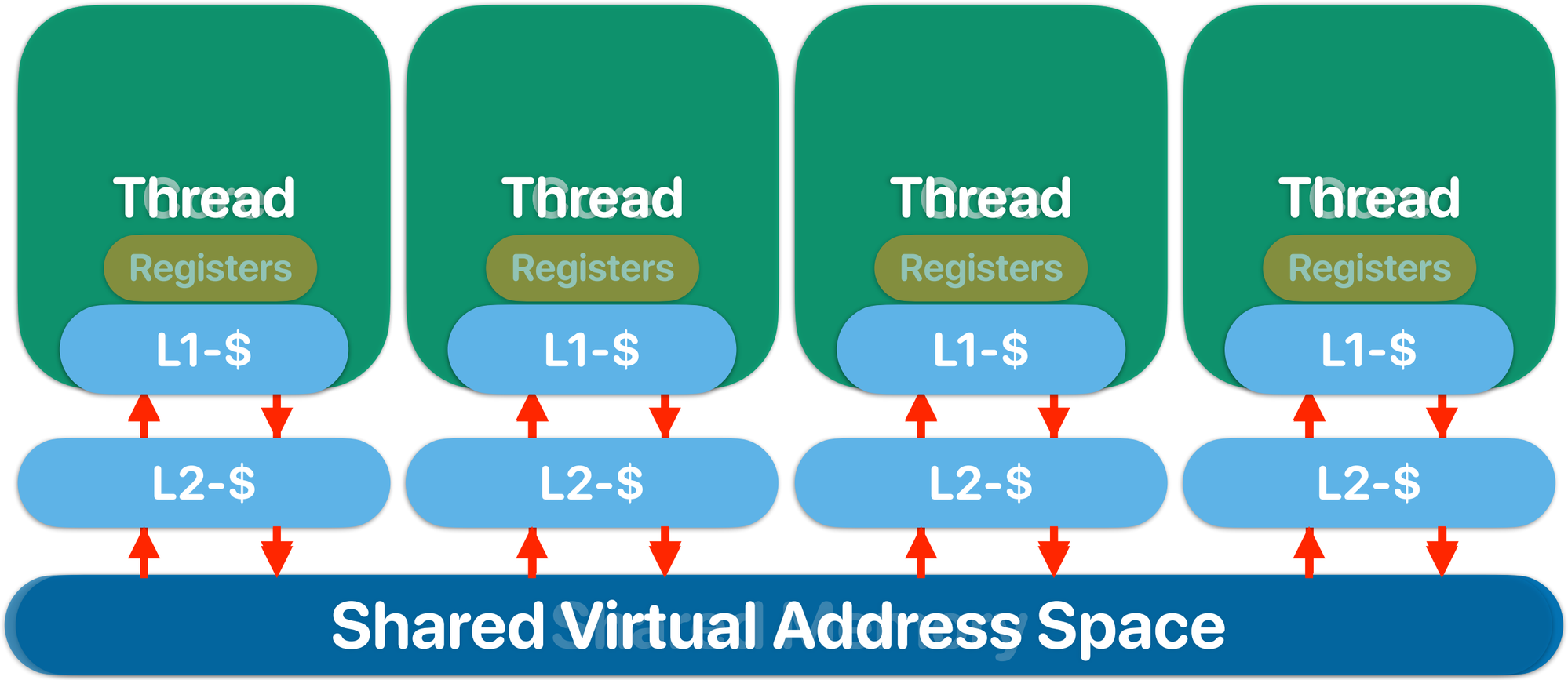
  A. P1
  B. P2

Architectural Support for Parallel Programming

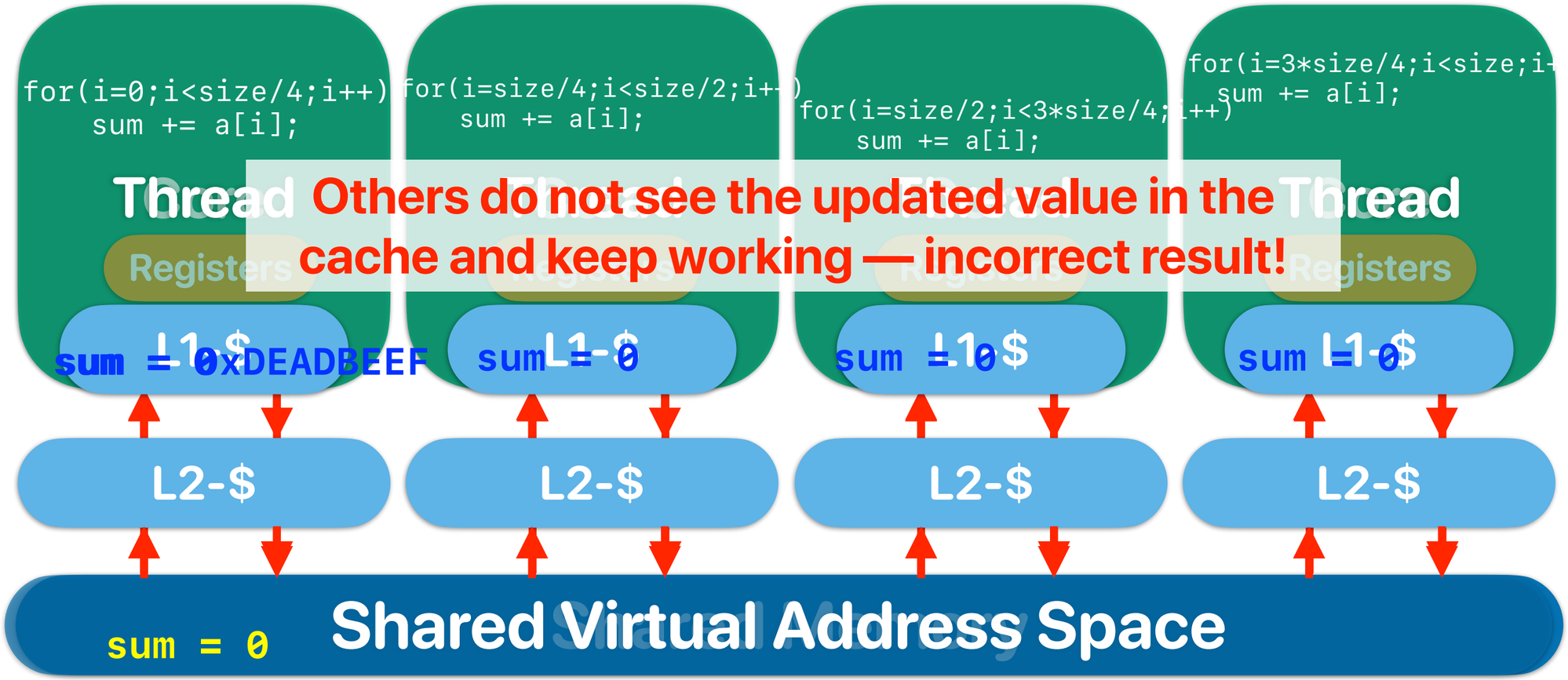
Parallel programming

- 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



Others do not see the updated value in the cache and keep working — incorrect result!

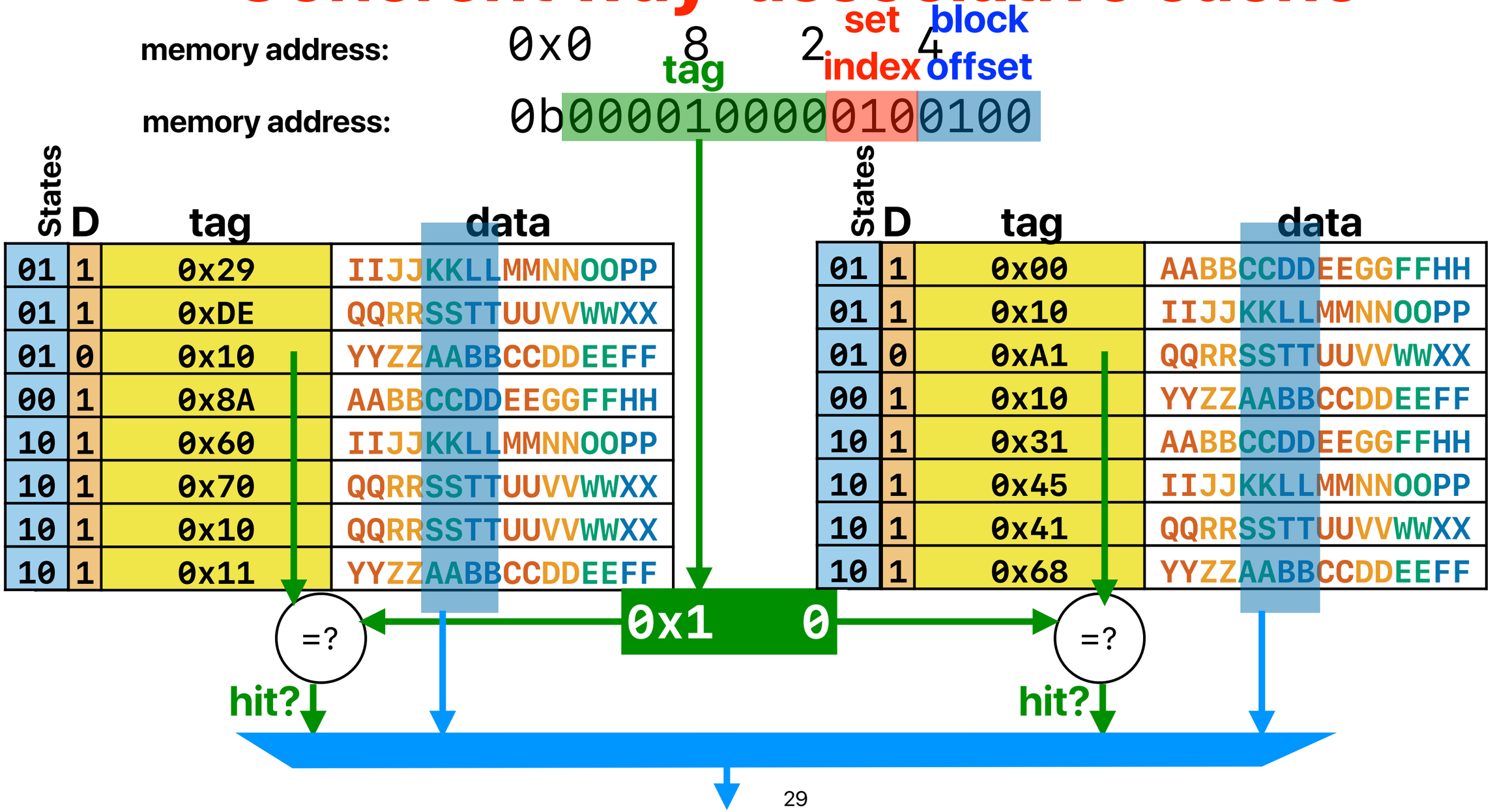
Coherency & Consistency

- 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

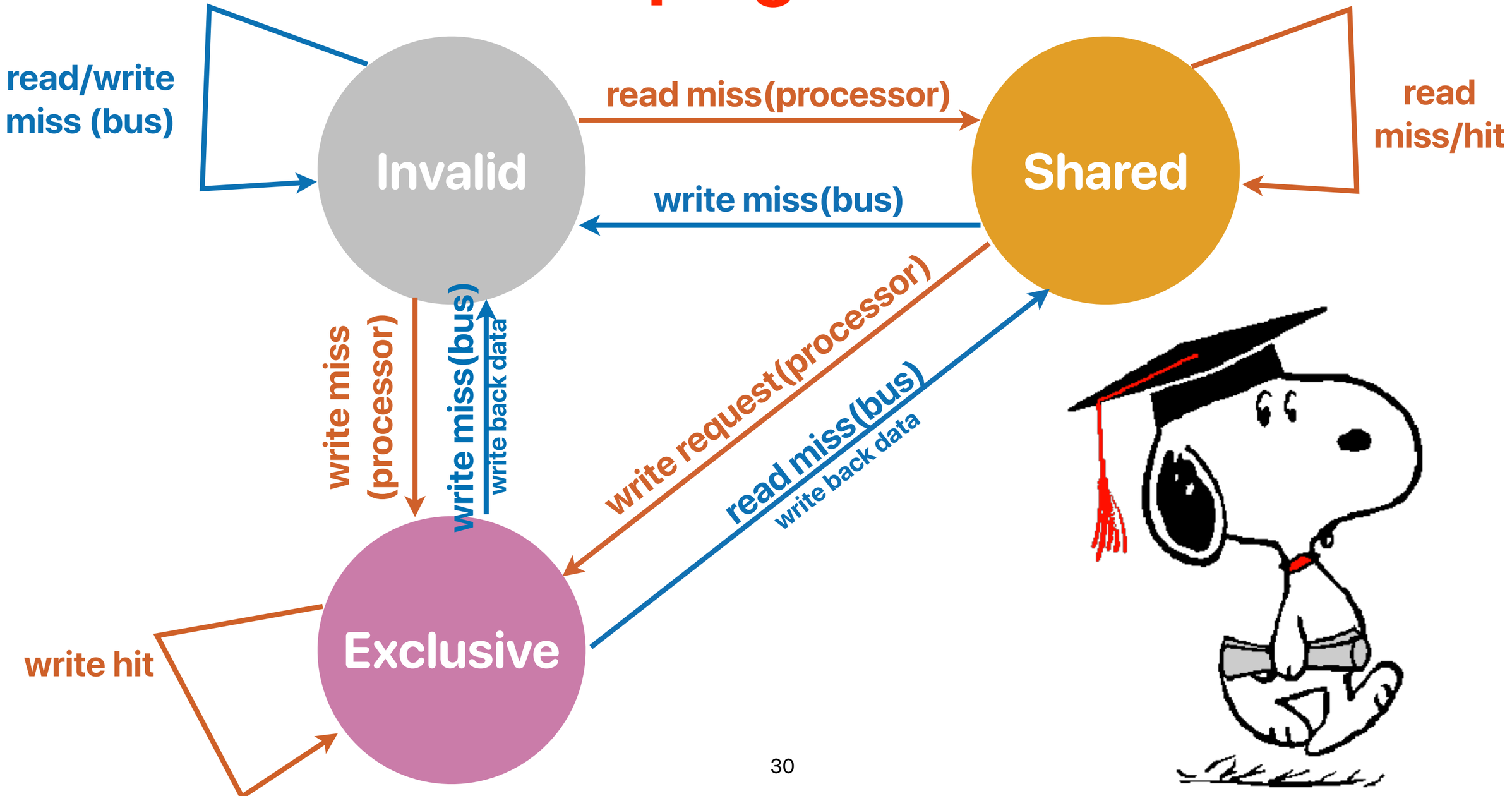
Simple cache coherency protocol

- 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

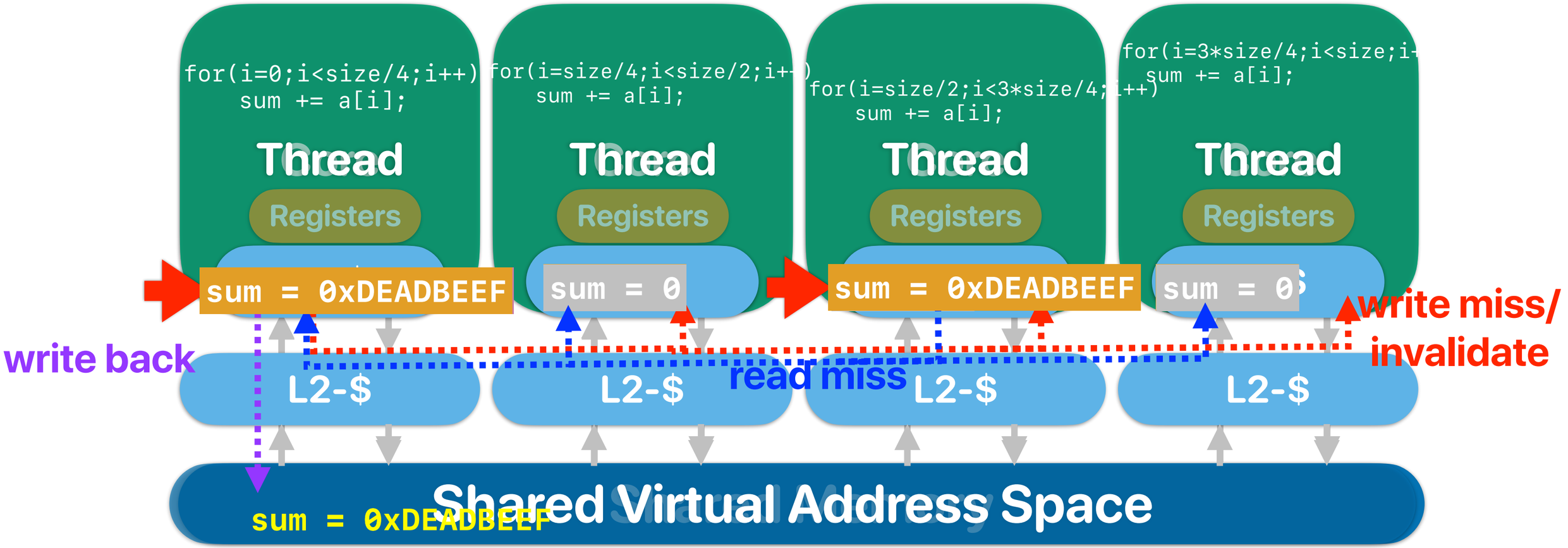
Coherent way-associative cache



Snooping Protocol



What happens when we write in coherent caches?



Poll 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

Poll 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 64 100

④ 1 1 1 1 1 1 1 1 1 100

A. 0

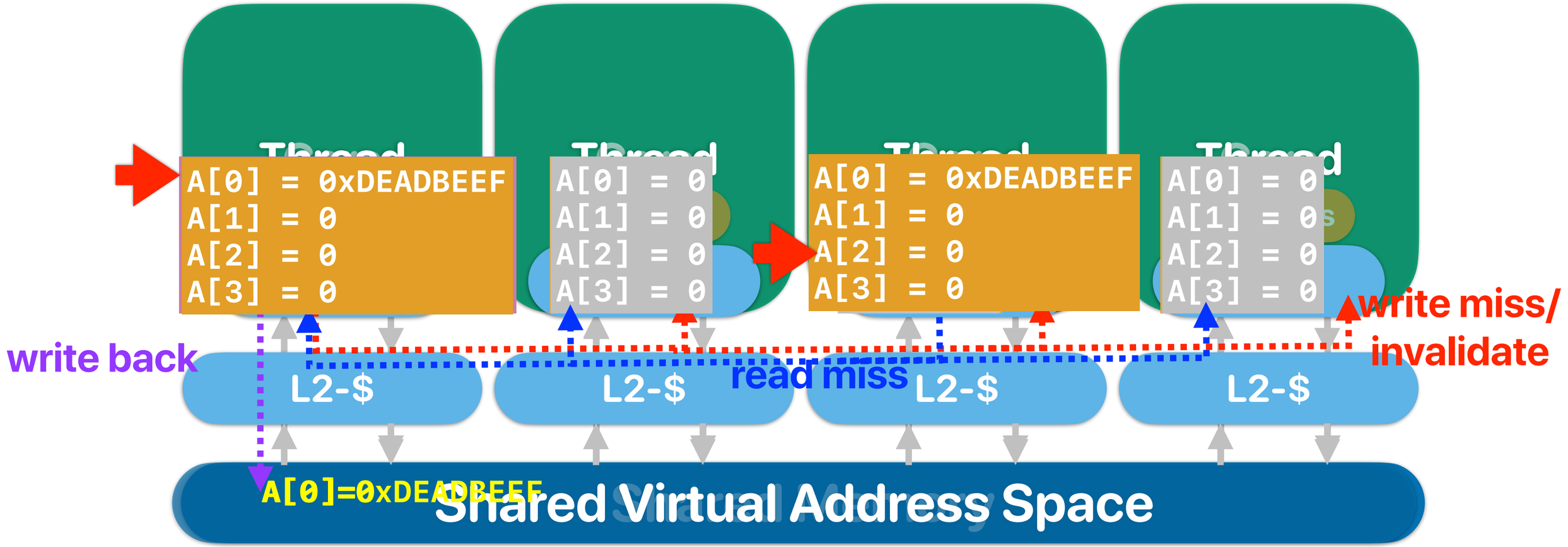
B. 1

C. 2

D. 3

E. 4

Cache coherency



Poll close in 1:30

Performance comparison

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

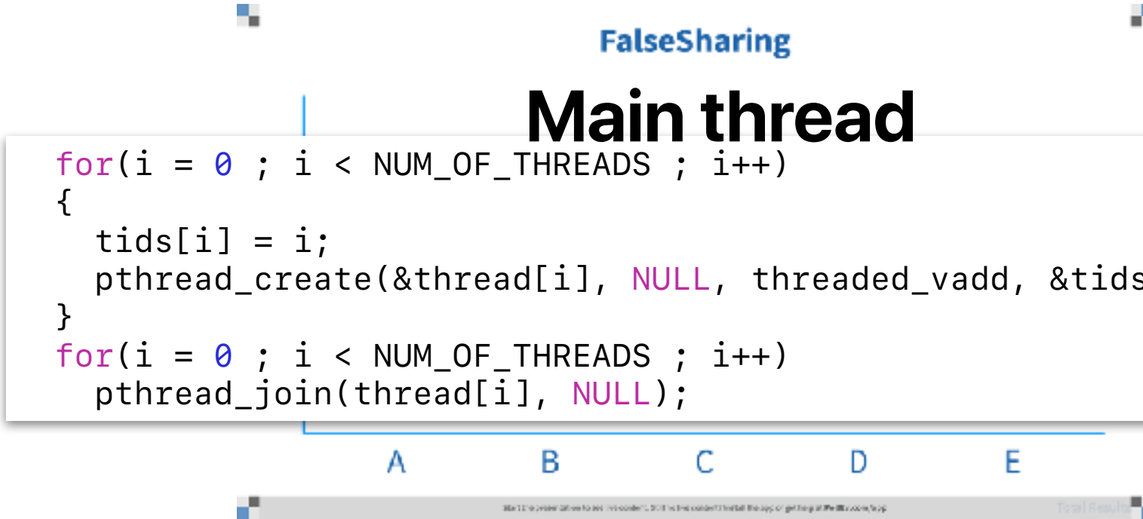
Version L

```
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



Poll close in 1:30

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

FalseSharing

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);
```

A B C D E

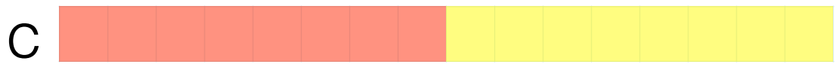
L v.s. R

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;
}
```



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);
```

Poll 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;
}
```

Poll close in 1:30

Again — how many values are possib

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

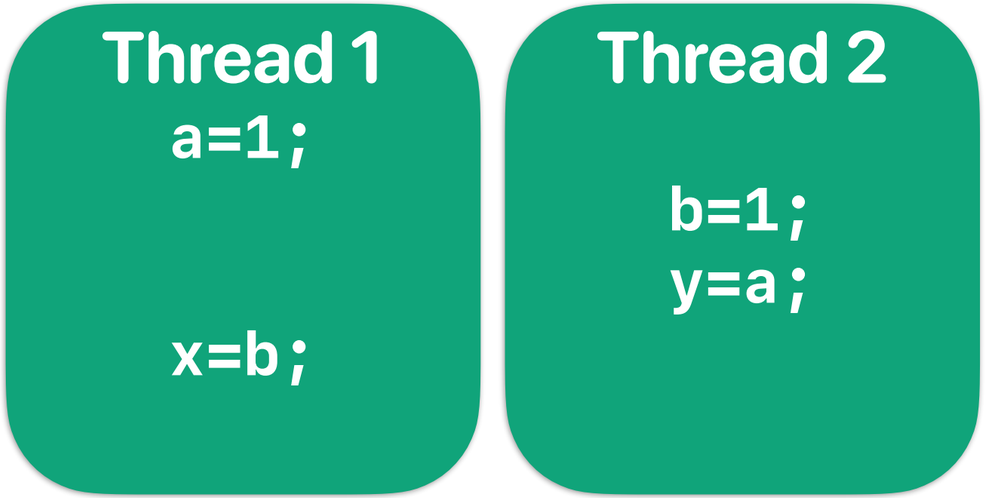
① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

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

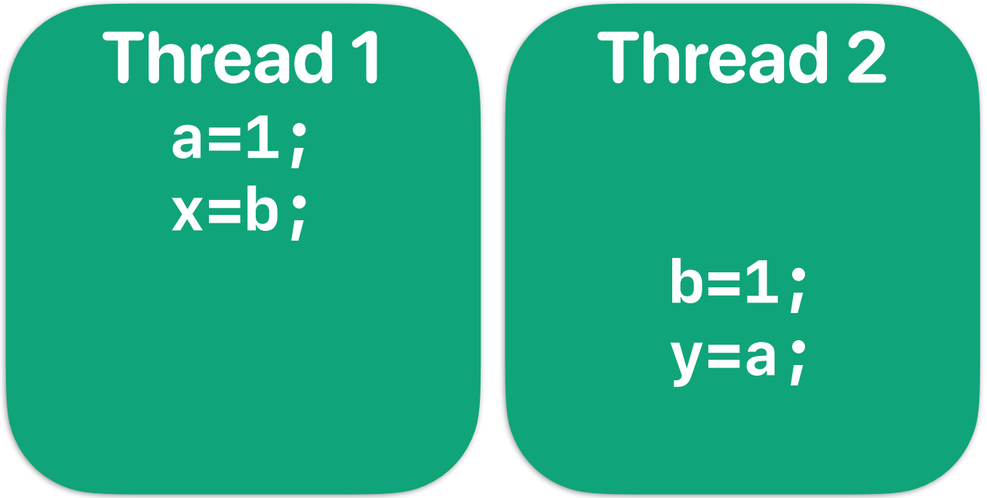
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

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;
}
```

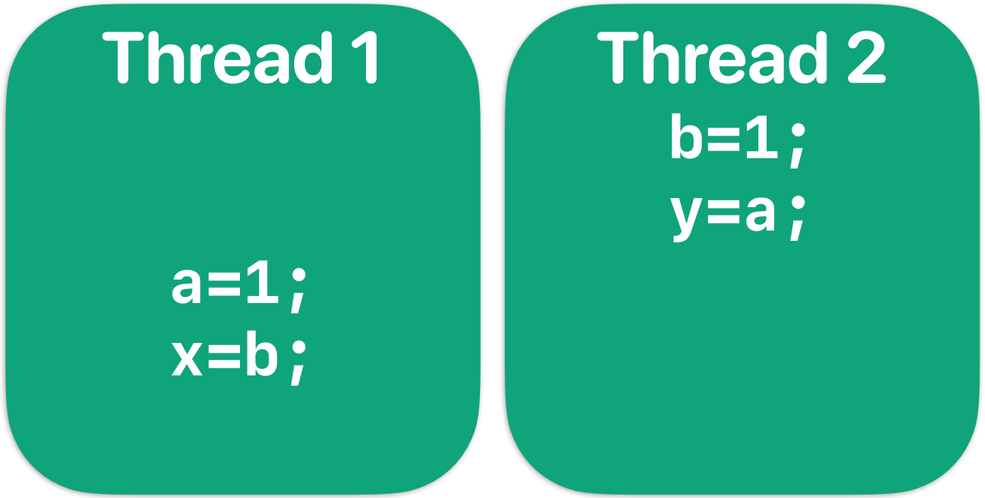
Possible scenarios



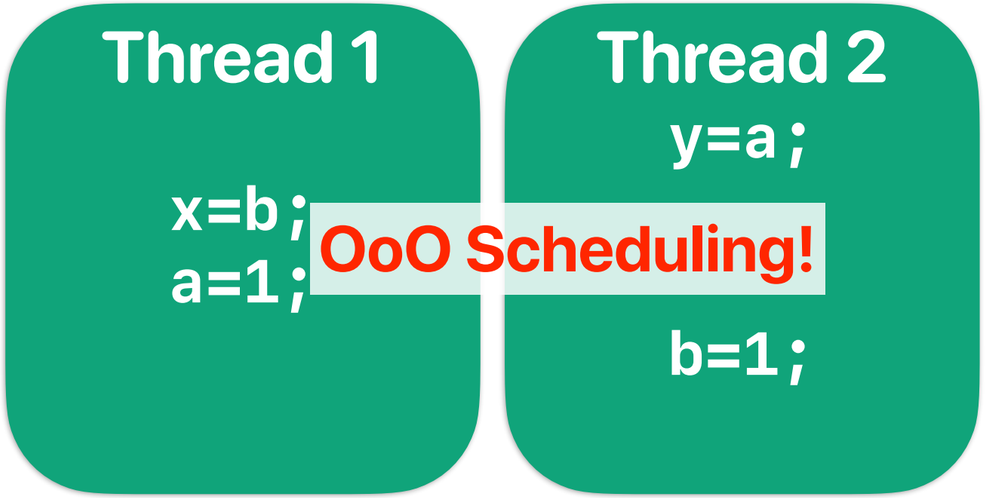
(1,1)



(0,1)



(1,0)



(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

```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;
}
```

Announcement

- Project due next Monday
- Last reading quiz due next Monday
- Assignment #5 due next Wednesday
- iEVAL, starting tomorrow until 12/11
  - Please fill the survey to let us know your opinion!
  - Don't forget to take a screenshot of your submission and submit through iLearn — it counts as a **full credit assignment**
  - **We will drop your lowest 2 assignment grades**
- Talk by Reetu Das next Monday — attend and submit a screenshot, count as a full credit reading quiz
- Final Exam
  - Starting from 12/10 to 12/15 11:59pm (we won't provide any technical support after 12pm 12/15), any consecutive 180 minutes you pick
  - Similar to the midterm, but more time and about 1.5x longer
  - Two of the questions will be comprehensive exam questions
  - Will release a sample final at the end of the last lecture
- Office Hours on Zoom (the office hour link, not the lecture one)
  - Hung-Wei/Prof. Usagi: M 8p-9p, W 2p-3p
  - Quan Fan: F 1p-3p

Computer
Science &
Engineering

203

つづく