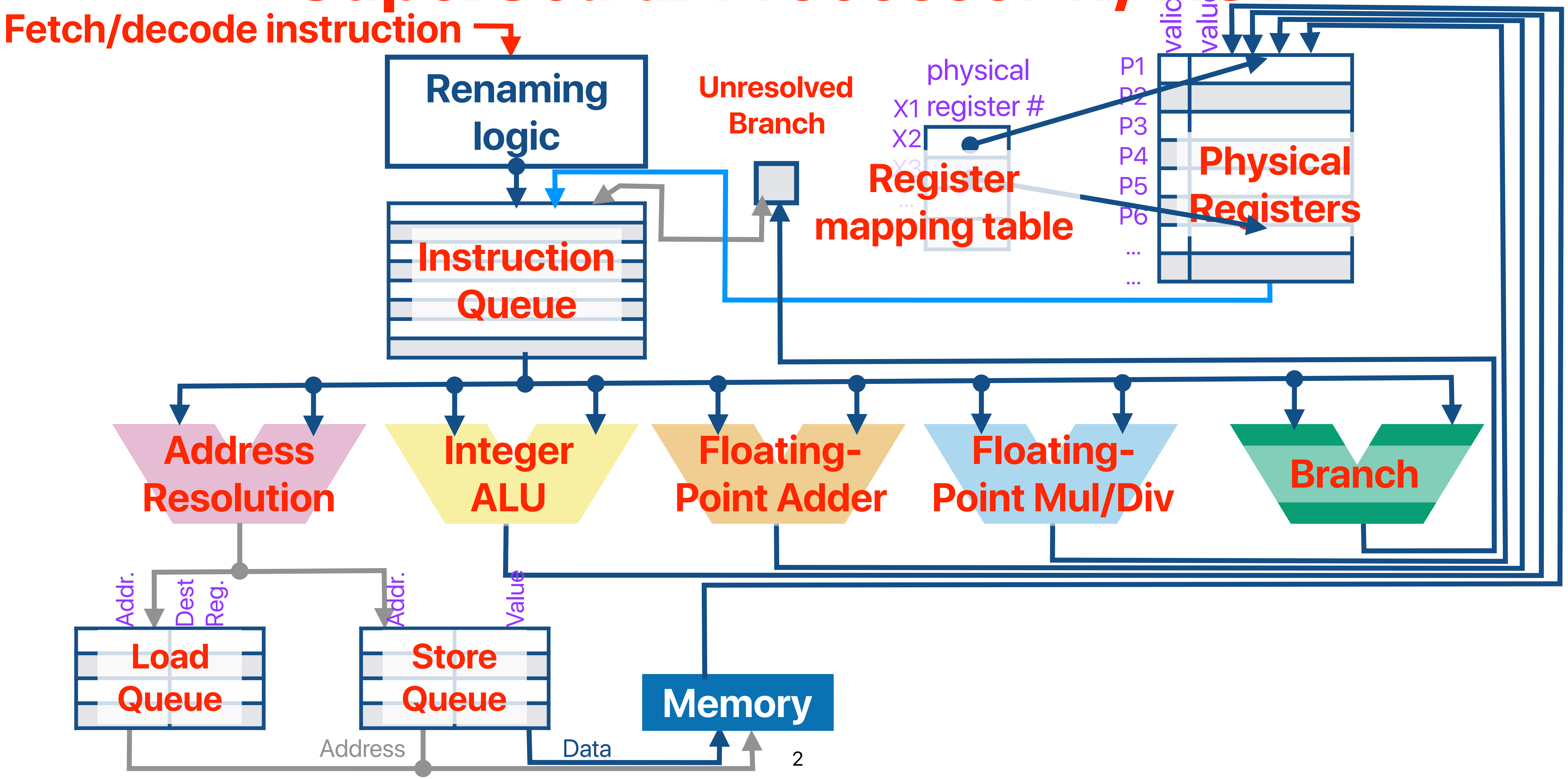
Thread-Level Parallelism — Simultaneous MultiThreading (SMT) & Chip Multi-Processors (CMP)

Hung-Wei

SuperScalar Processor w/ ROB



Recap: What about "linked list"

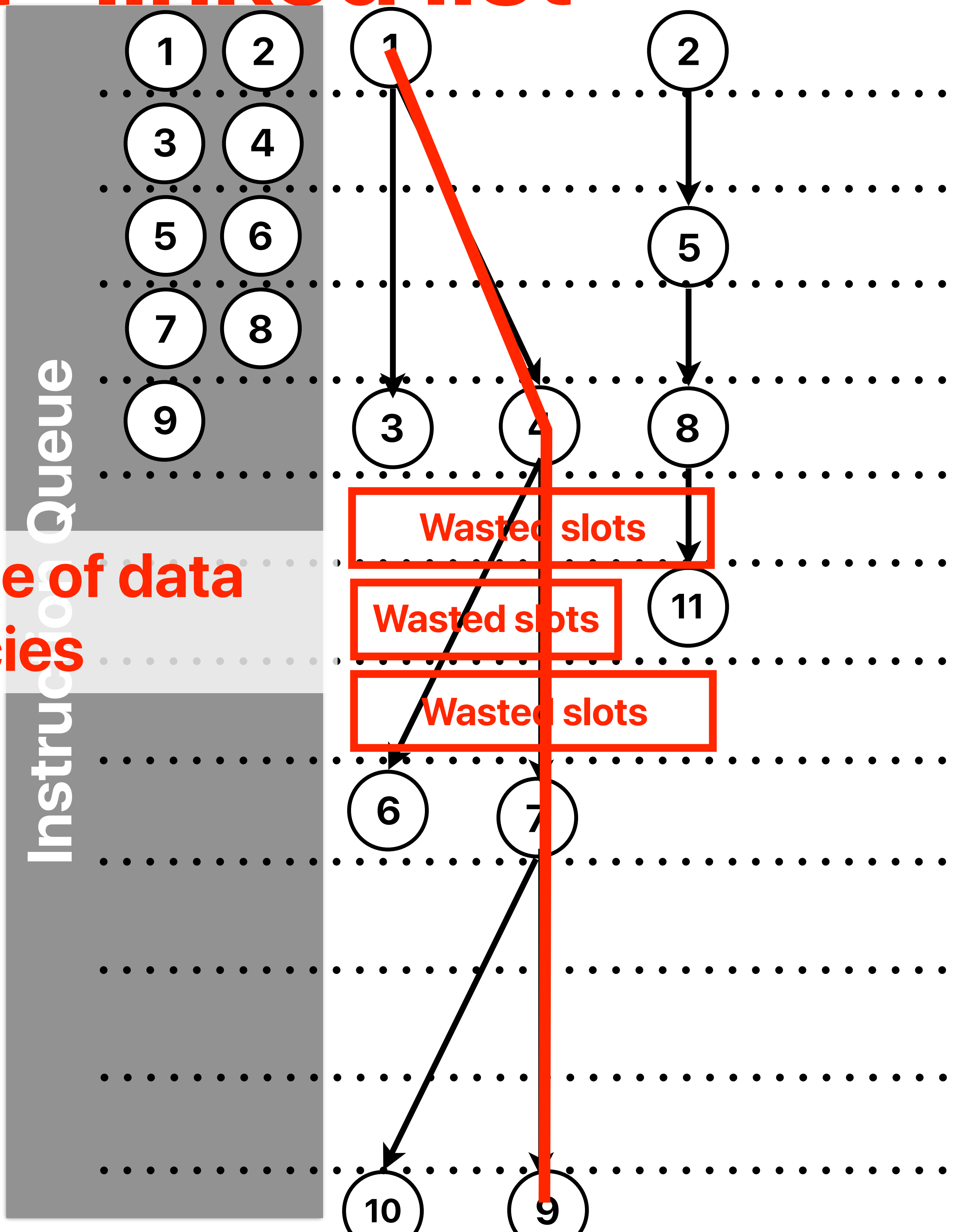
Static instructions

```
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

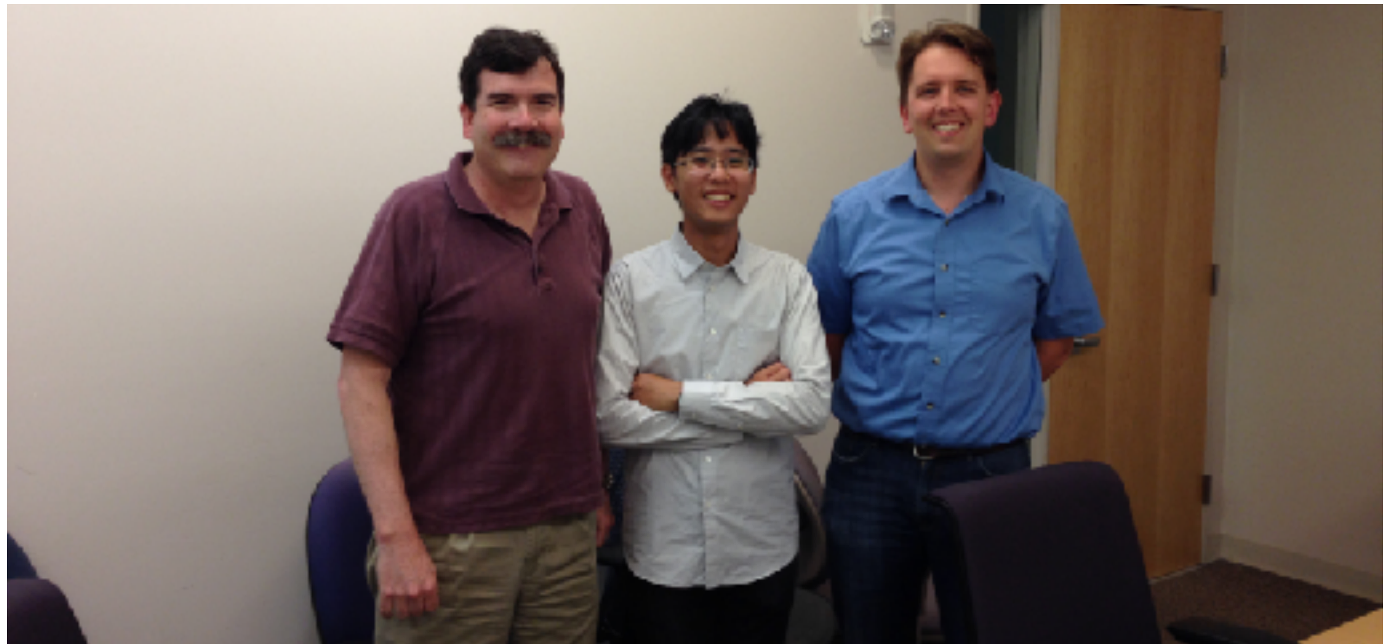


Demo: ILP within a program

- 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

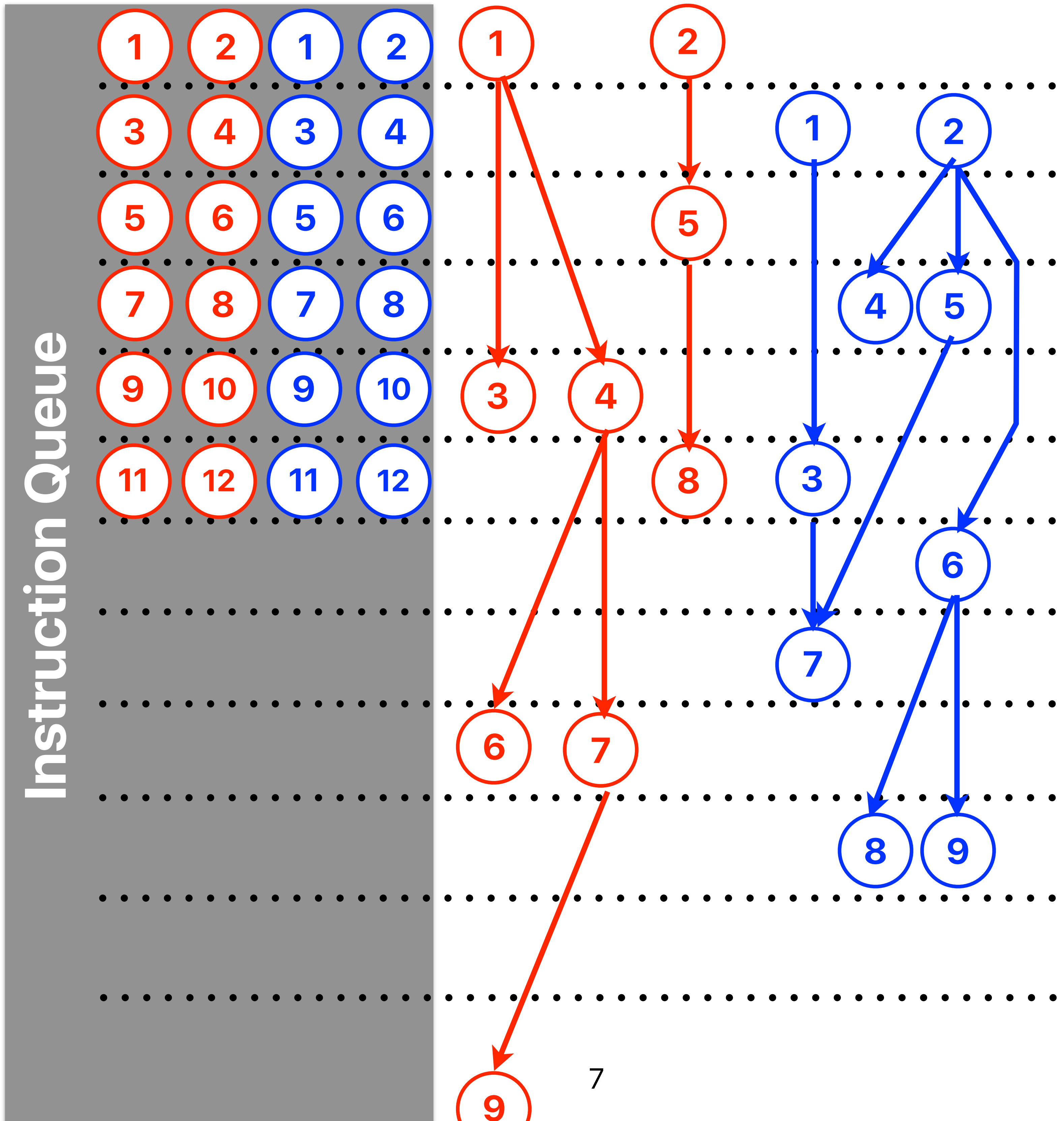


Simultaneous multithreading

- The processor can schedule instructions from different threads/processes/programs
- Fetch instructions from different threads/processes to fill the not utilized part of pipeline
  - Exploit “thread level parallelism” (TLP) to solve the problem of insufficient ILP in a single thread
  - You need to create an illusion of multiple processors for OSs

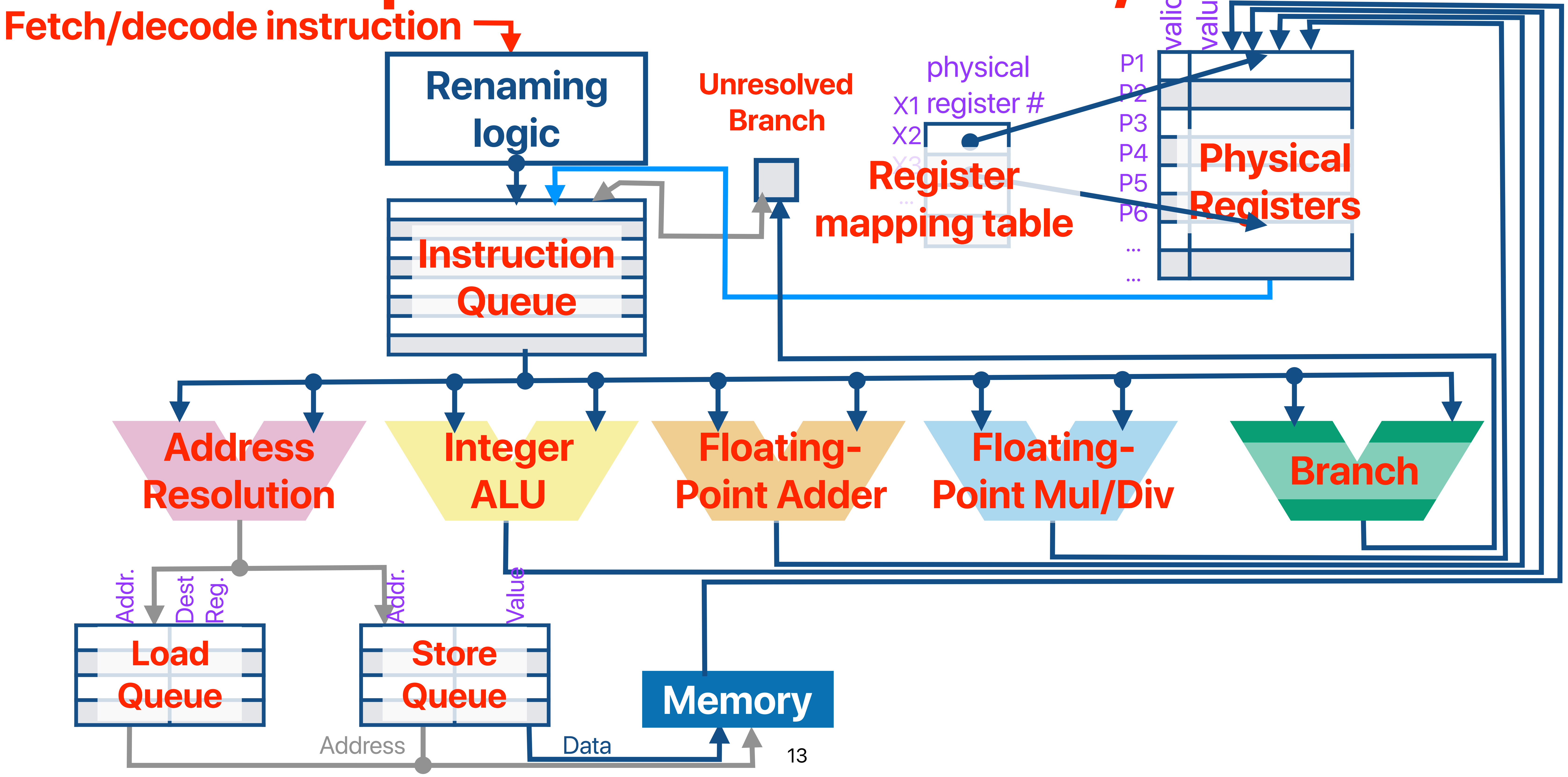
Simultaneous multithreading

```
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
```

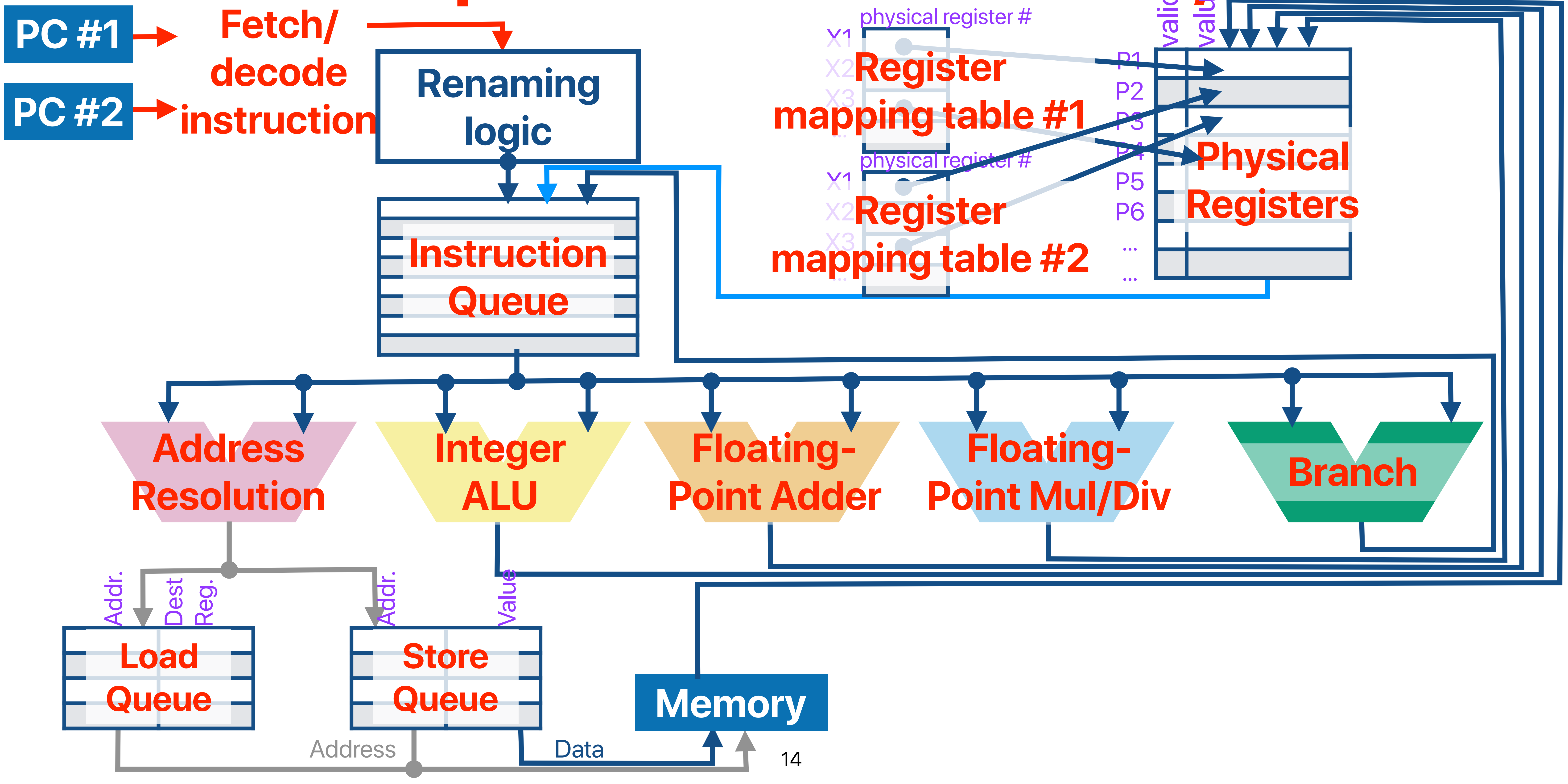



```
①  ld   X1, 0(X10)
②  addi X10, X10, 8
③  add  X20, X20, X1
④  bne  X10, X2, LOOP
⑤  ld   X1, 0(X10)
⑥  addi X10, X10, 8
⑦  add  X20, X20, X1
⑧  bne  X10, X2, LOOP
⑨  ld   X1, 0(X10)
⑩  addi X10, X10, 8
⑪  add  X20, X20, X1
⑫  bne  X10, X2, LOOP
```

SuperScalar Processor w/ ROB



SMT SuperScalar Processor w/ ROB



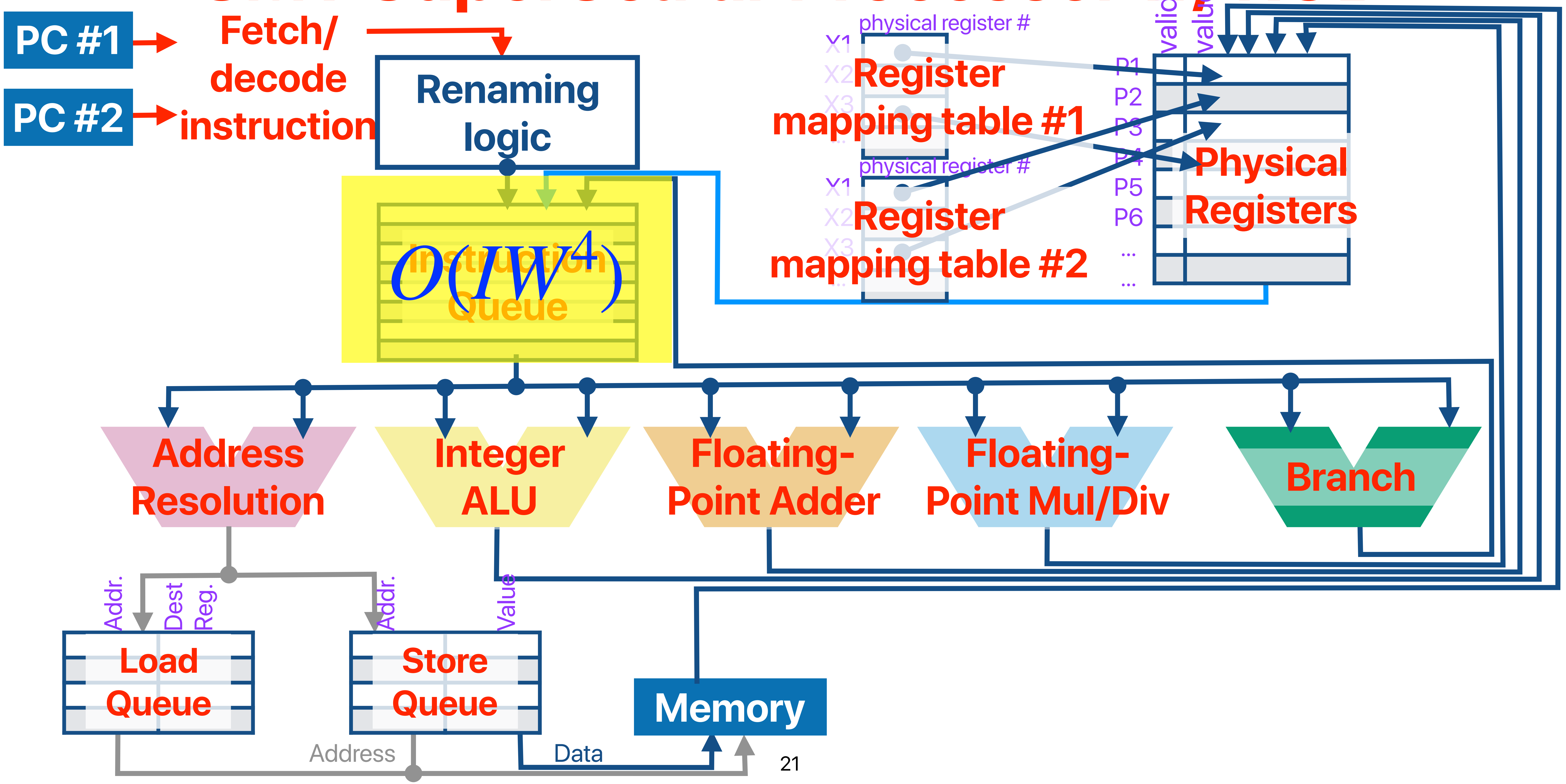
SMT

- How many of the following about SMT are correct?
  ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches **We can execute from other threads/contexts instead of the current one**
  hurt, b/c you are sharing resource with other threads.
  ② SMT can ~~improve~~ the throughput of a single-threaded application
  ③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width **We can execute from other threads/contexts instead of the current one**
  ④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.
  b/c we're sharing the cache
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

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



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.

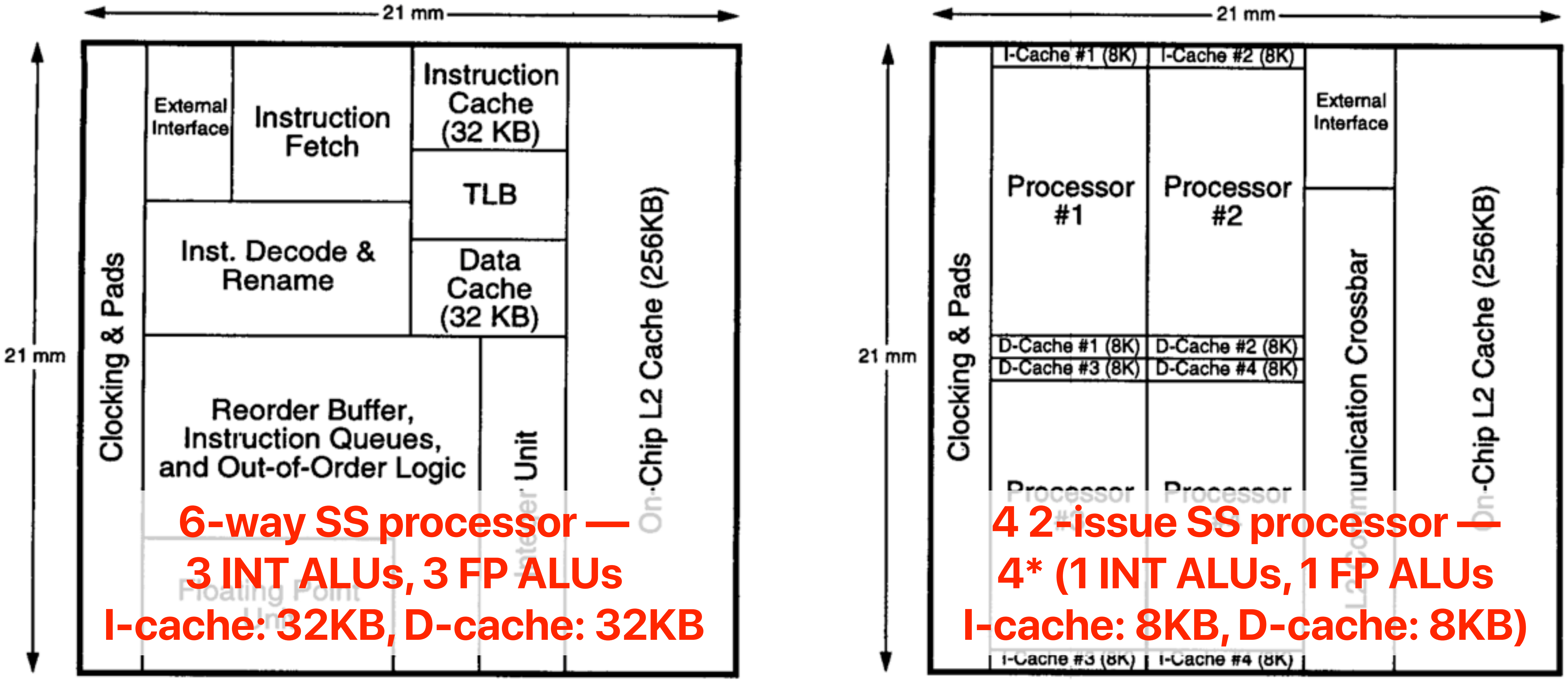
Program	IPC	BP Rate %	I cache %MPCI	D cache %MPCI	L2 cache %MPCI
compress	1.2	86.4	0.0	3.9	1.1
eqntott	1.8	80.0	0.0	1.1	1.1
m88ksim	2.3	92.6	0.1	0.0	0.0
MPsim	1.2	81.6	3.4	1.7	2.3
applu	1.7	79.7	0.0	2.8	2.8
apsi	1.2	95.6	0.2	3.1	2.6
swim	2.2	99.8	0.0	2.3	2.5
tomcatv	1.3	99.7	0.0	4.2	4.3
pmake	1.4	82.7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

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

The case for a Single-Chip Multiprocessor

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

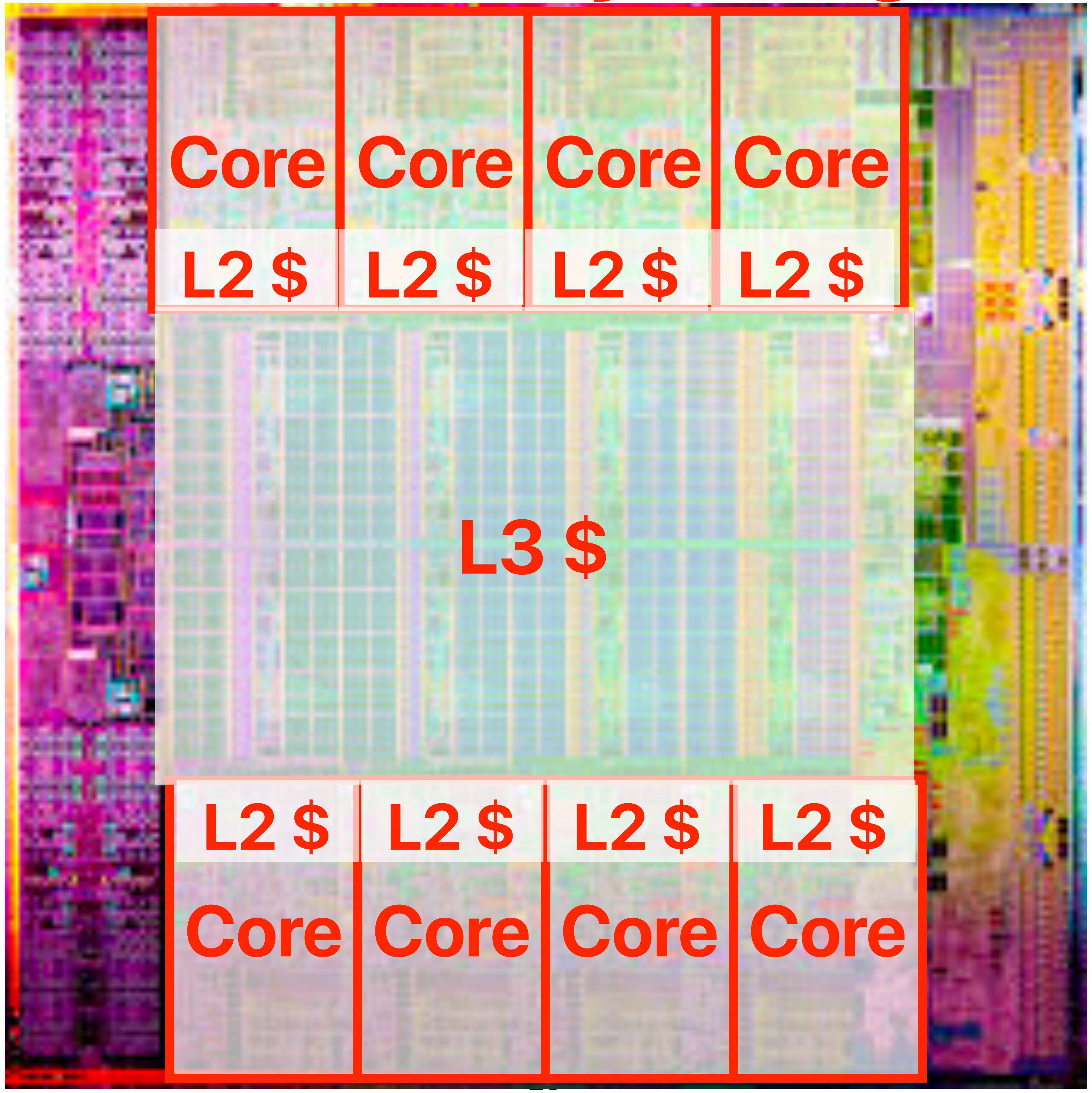
Wide-issue SS processor v.s. multiple narrower-issue SS processors

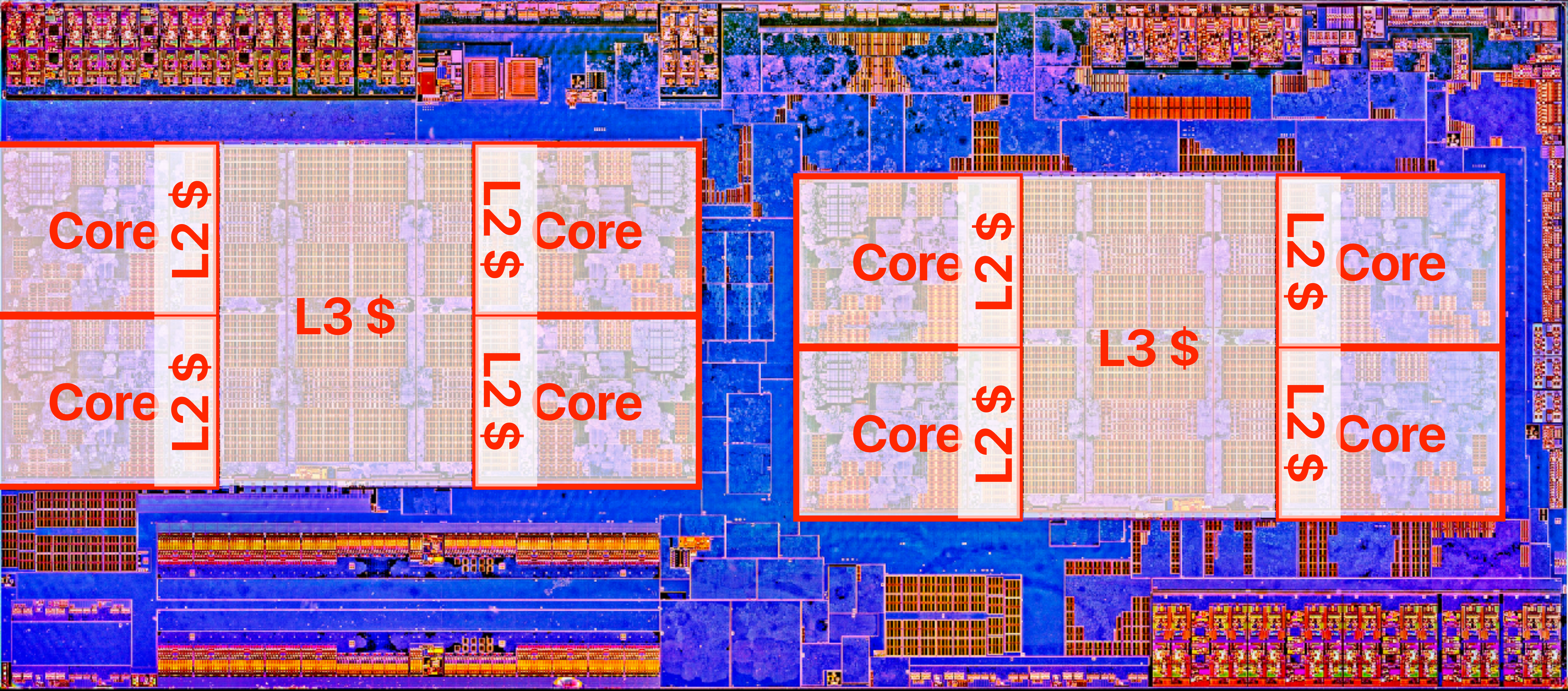


6-way SS processor —
3 INT ALUs, 3 FP ALUs
I-cache: 32KB, D-cache: 32KB

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

Intel Sandy Bridge

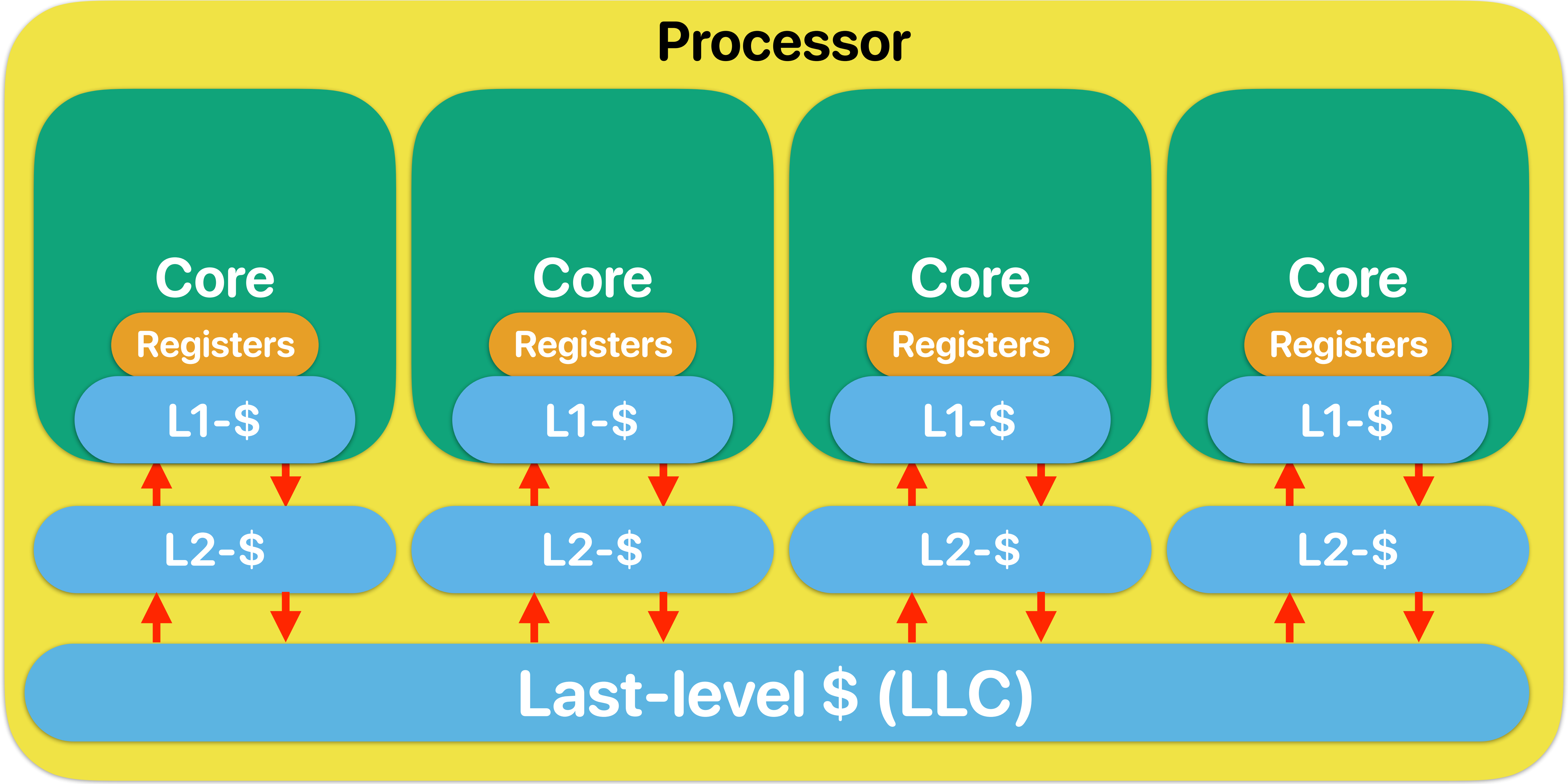



Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core

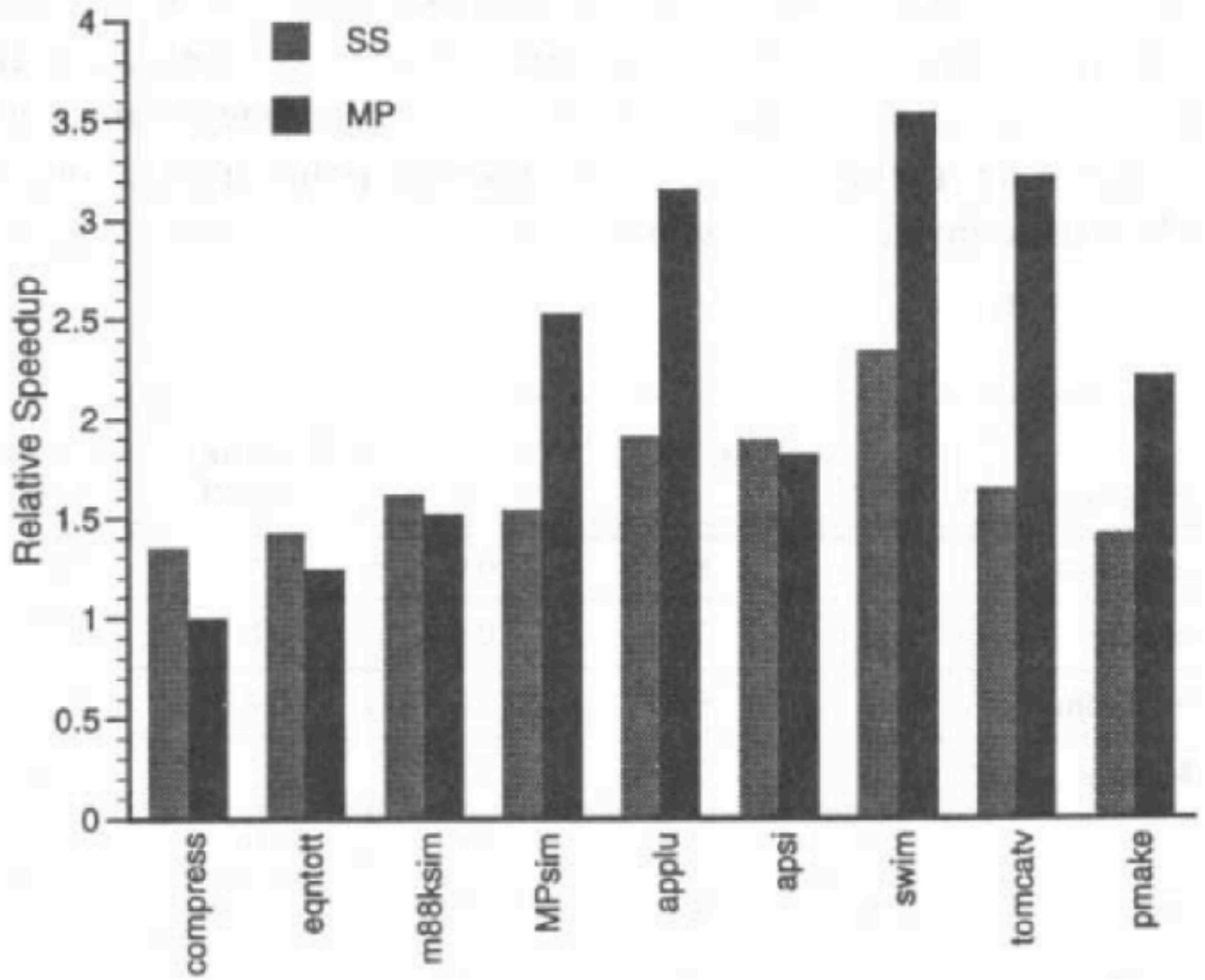
AMD

RYZEN

Concept of CMP



Performance of CMP



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?

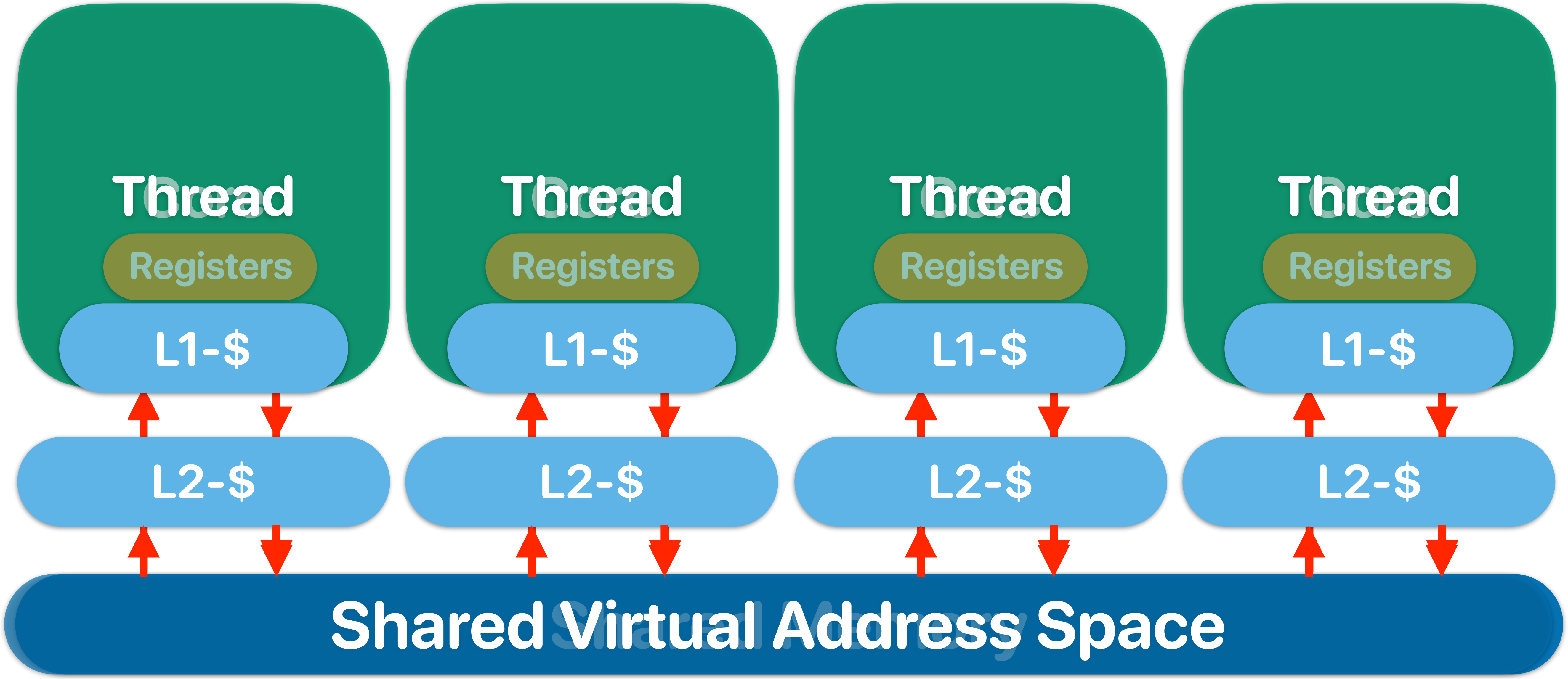
  A. P1
  B. P2

Architectural Support for Parallel Programming

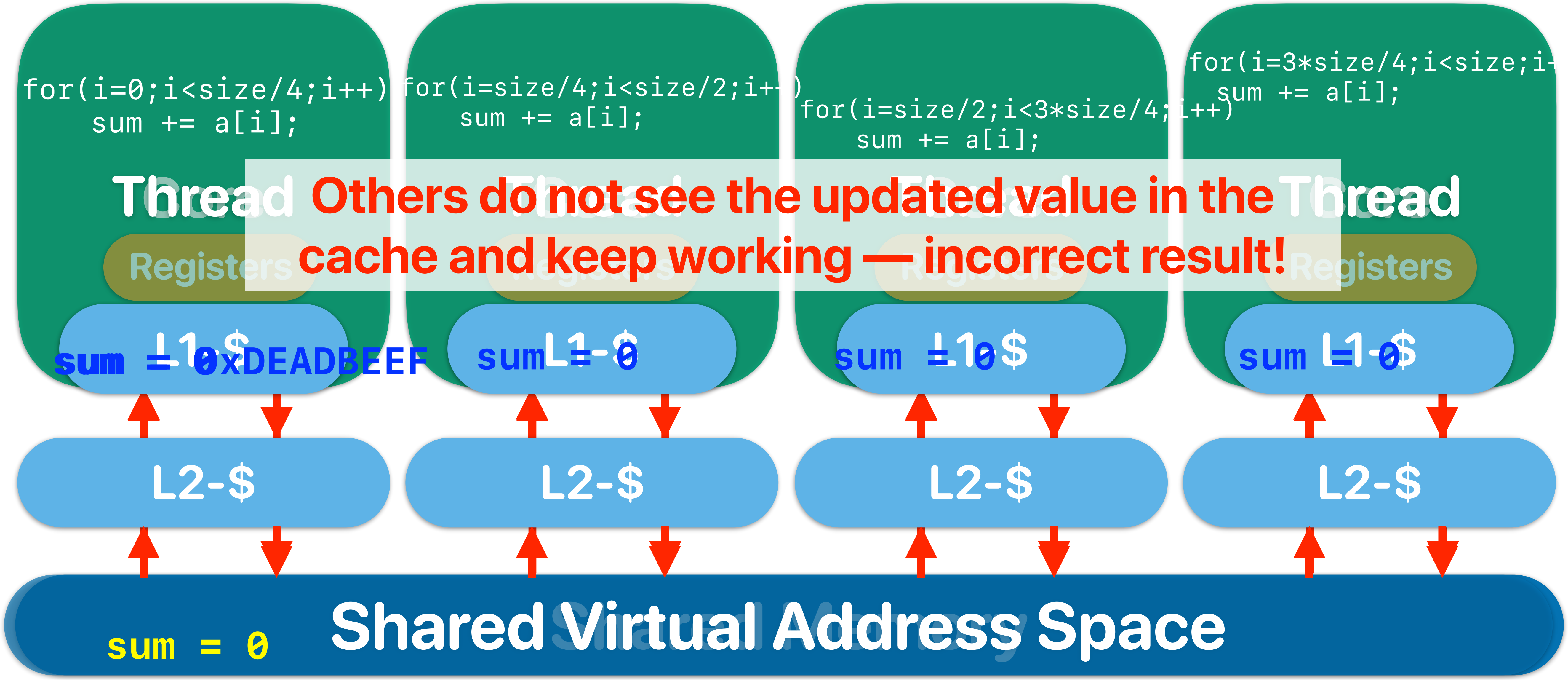
Parallel programming

- 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



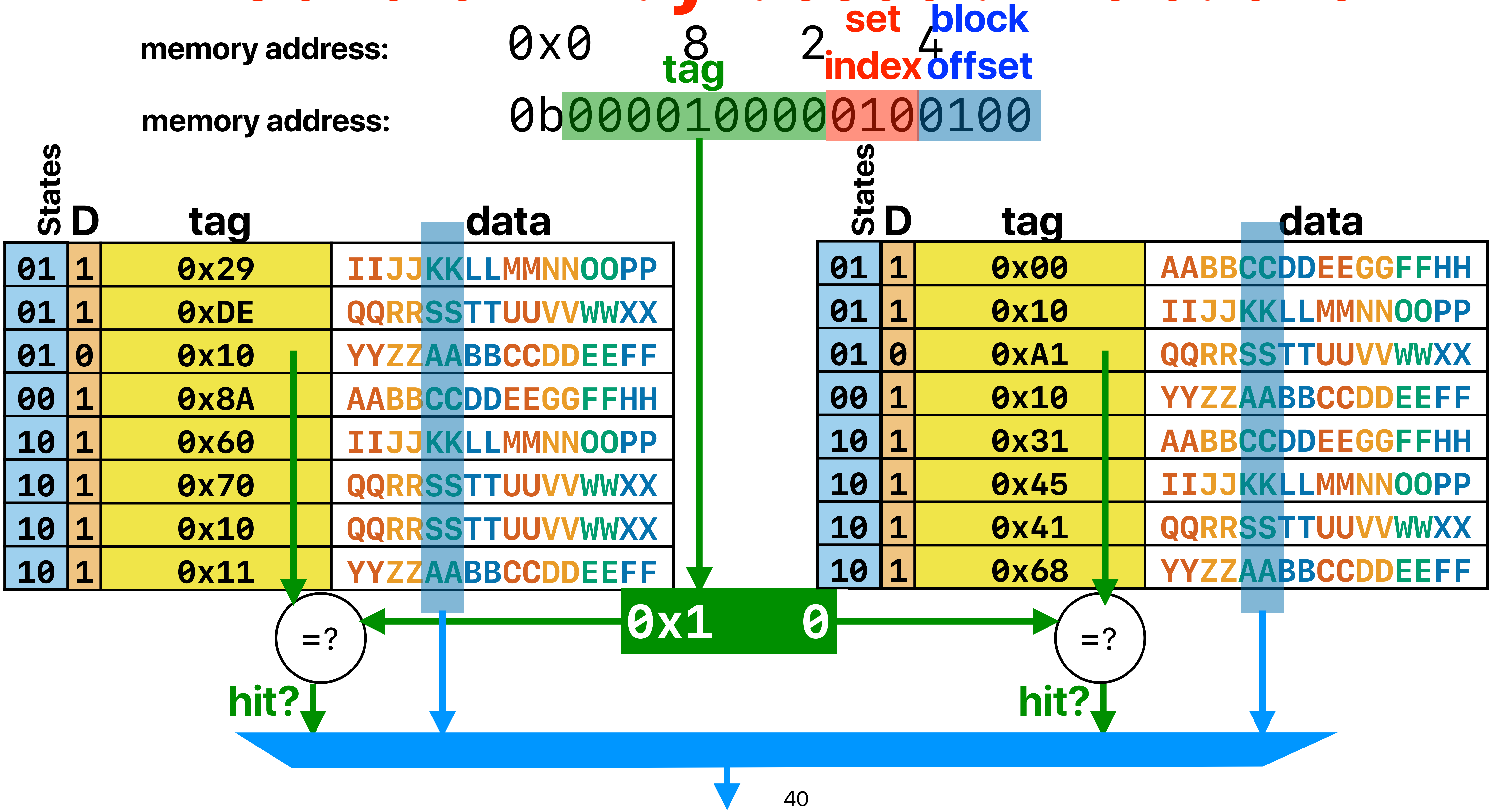
Coherency & Consistency

- 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

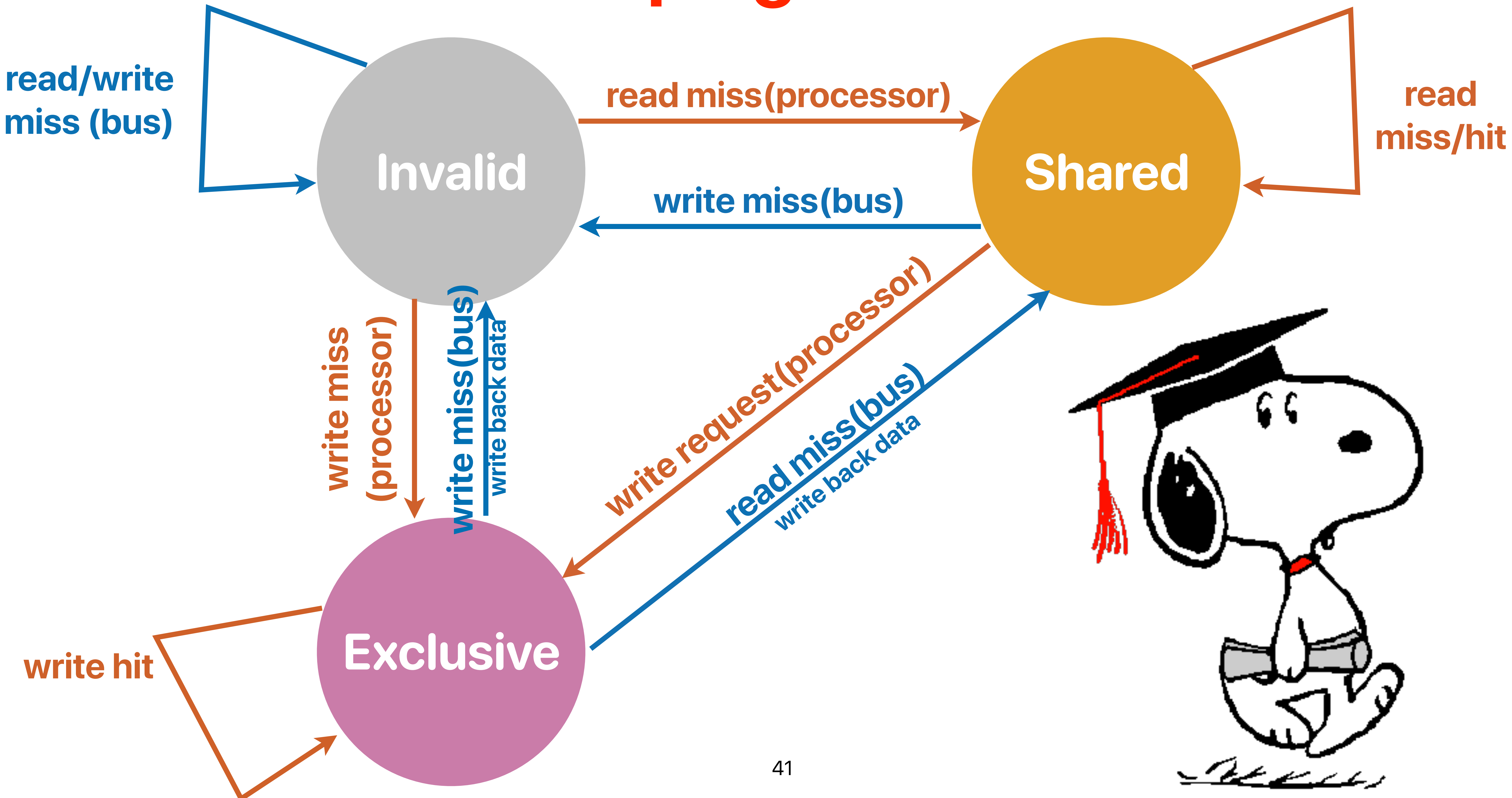
Simple cache coherency protocol

- 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

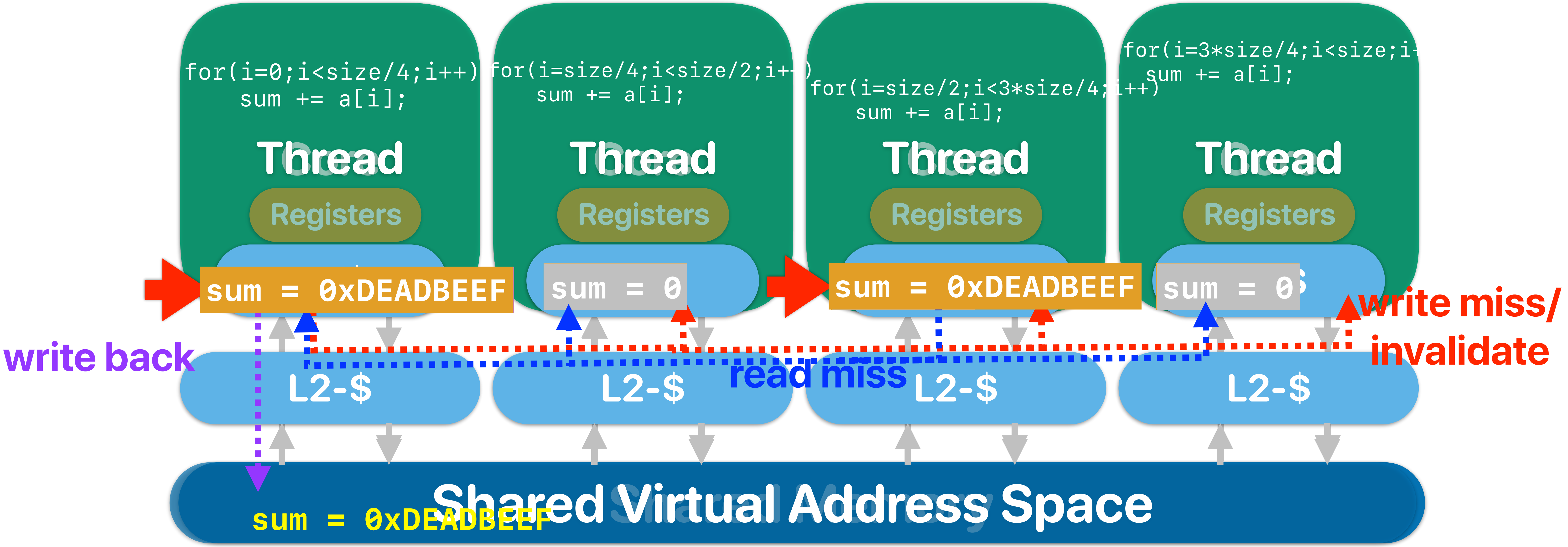
Coherent way-associative cache



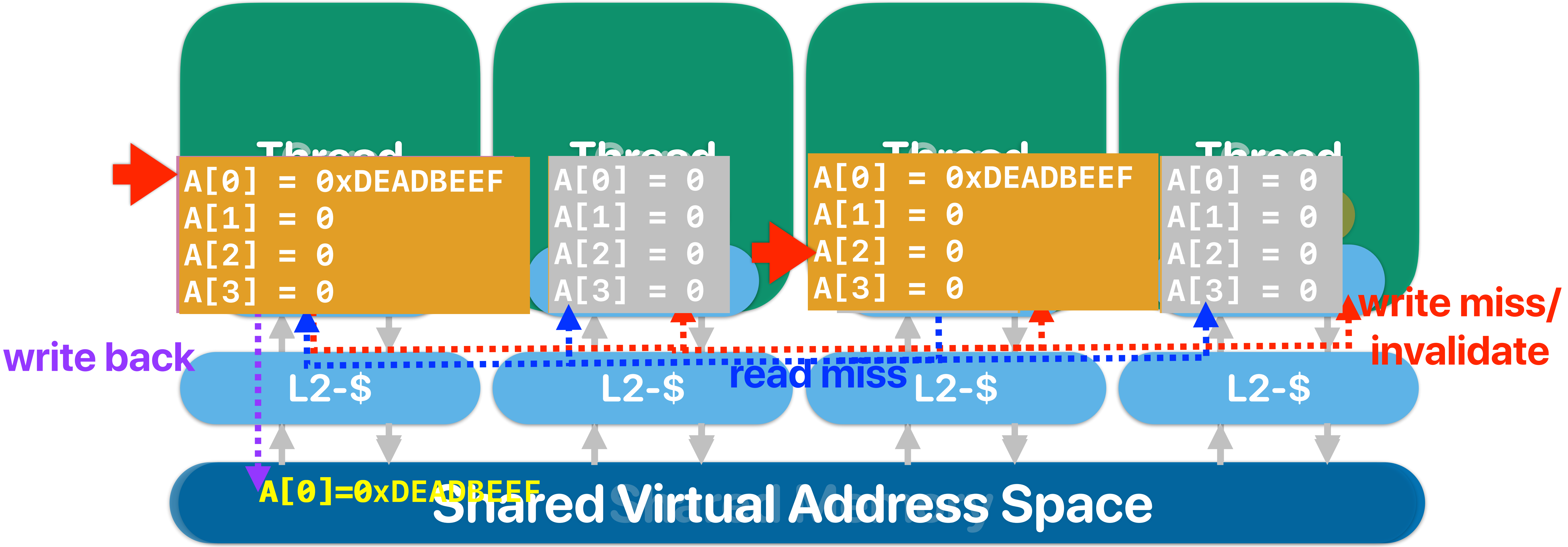
Snooping Protocol



What happens when we write in coherent caches?

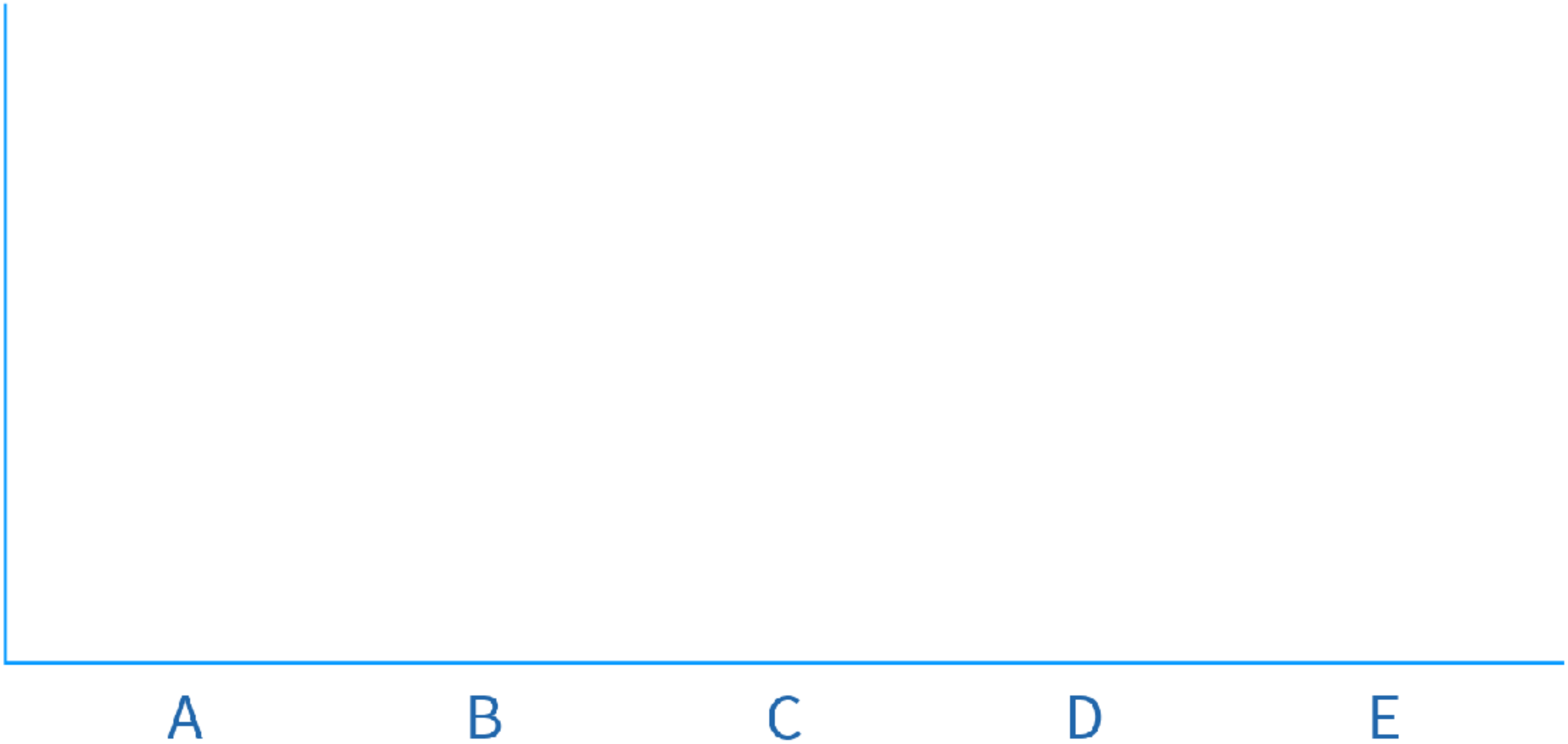


Flash



A[0] = 0xDEADBEEF
A[1] = 0
A[2] = 0
A[3] = 0
A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0
A[0] = 0xDEADBEEF
A[1] = 0
A[2] = 0
A[3] = 0
A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0
write miss/
invalidate
write back
read miss
L2-$
L2-$
L2-$
L2-$
A[0]=0xDEADBEEF
Shared Virtual Address Space

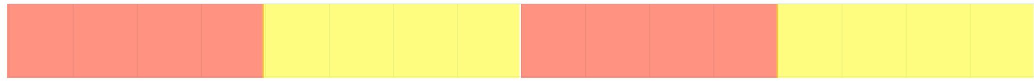
FalseSharing-Group



L v.s. R

Version L

```
void *threaded_vadd(void *thread_id)
{
  __m128 va, vb, vt;
  int tid = *(int *)thread_id;
  int i = tid * 4;
  for(i = tid * 4; i < ARRAY_SIZE; i+=4*NUM_OF_THREADS)
  {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
  }
  return NULL;
}
```

a 

Version R

```
void *threaded_vadd(void *thread_id)
{
  __m128 va, vb, vt;
  int tid = *(int *)thread_id;
  int i = tid * 4;
  for(i = tid*(ARRAY_SIZE/NUM_OF_THREADS); \
  i < (tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i+=4)
  {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
  }
  return NULL;
}
```

a 

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!

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
- 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

Announcement

- Final Review on 12/2 — 7pm-8:20pm
- Reading quiz due next Monday
- Homework #4 due 12/4
- iEval submission — attach your "confirmation" screen, you get an extra/bonus homework
- Project due on 12/2
  - You can only turn-in "helper.c"
    - `mcfutil.c:refresh_potential()` creates helper threads
    - `mcfutil.c:refresh_potential()` calls `helper_thread_sync()` function periodically
    - It's your task to think what to do in `helper_thread_sync()` and `helper_thread()` functions
    - Please DO READ papers before you ask what to do
  - Formula for grading — **min(100, speedup*100)**
  - No extension
- Office hour for Hung-Wei **next** week — MWF 1p-2p — no office hour this week

Thread-Level Parallelism — Simultaneous MultiThreading (SMT) & Chip Multi-Processors (CMP)

Hung-Wei