

Multithreaded Architectures and Programming on Modern Processors: the Multithreaded Version

Hung-Wei Tseng

Outline

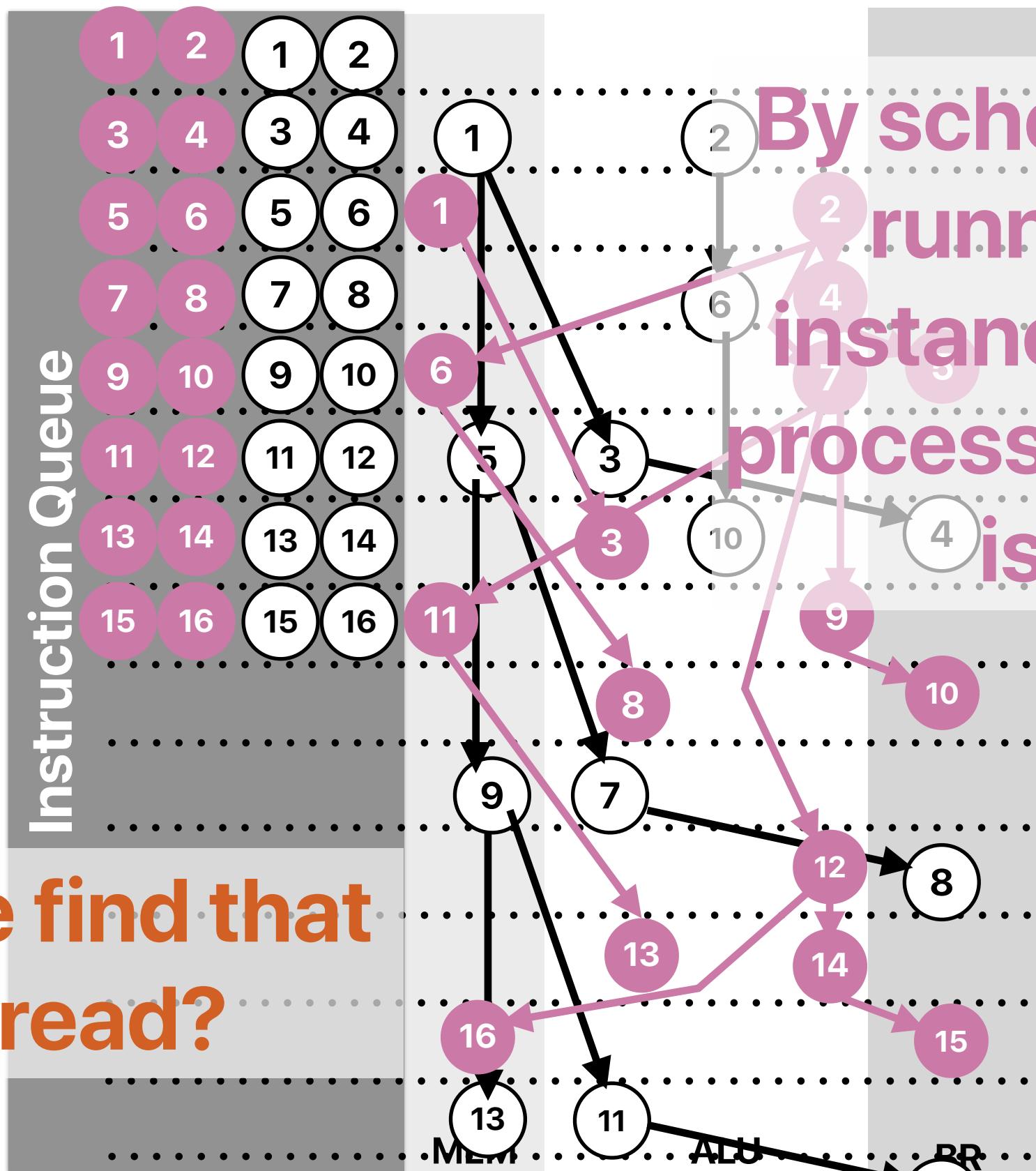
- Parallel architectures
 - Simultaneous multithreading (SMT)
 - Chip multiprocessors (CMP)
- Parallel programming

Parallel architectures

Simultaneous multithreading

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



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

Where can we find that "other" thread?

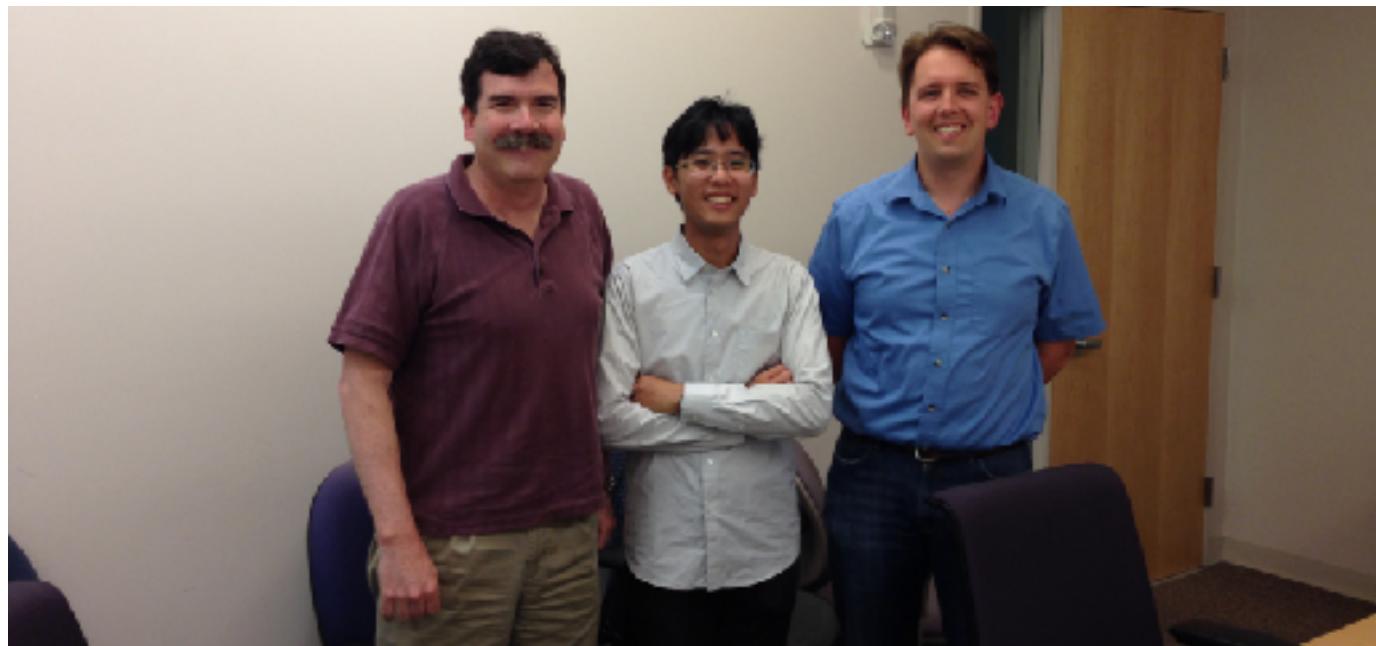
- ① movl (%rdi), %ecx
- ② addl %rdi, %rdi
- ③ addl %rdi, %rdi
- ④ addl %rdi, %rdi
- ⑤ addl %rdi, %rdi
- ⑥ addl %rdi, %rdi
- ⑦ addl %rdi, %rdi
- ⑧ addl %rdi, %rdi
- ⑨ addl %rdi, %rdi
- ⑩ addl %rdi, %rdi
- ⑪ addl %rdi, %rdi
- ⑫ addl %rdi, %rdi
- ⑬ addl %rdi, %rdi
- ⑭ addl %rdi, %rdi
- ⑮ addl %rdi, %rdi
- ⑯ addl %rdi, %rdi

Where to find another “thread”

- Another process/running program forked from a completely different program
- Another process forked/cloned from the current process
 - The forked program cloned the memory content at the forked time
 - The forked program cannot view the memory space of the original process
 - The forked program can perform a subset of tasks form the original process
 - The forked and the original process can exchange data through files or inter-process communication APIs
- A software thread spawned from the current process
 - The spawned thread executes a function instance from the main process to offload computation from the main process
 - The spawned thread shares the memory space with the main process

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
- Invented by Dean Tullsen (Now a professor at **UCSD CSE**)



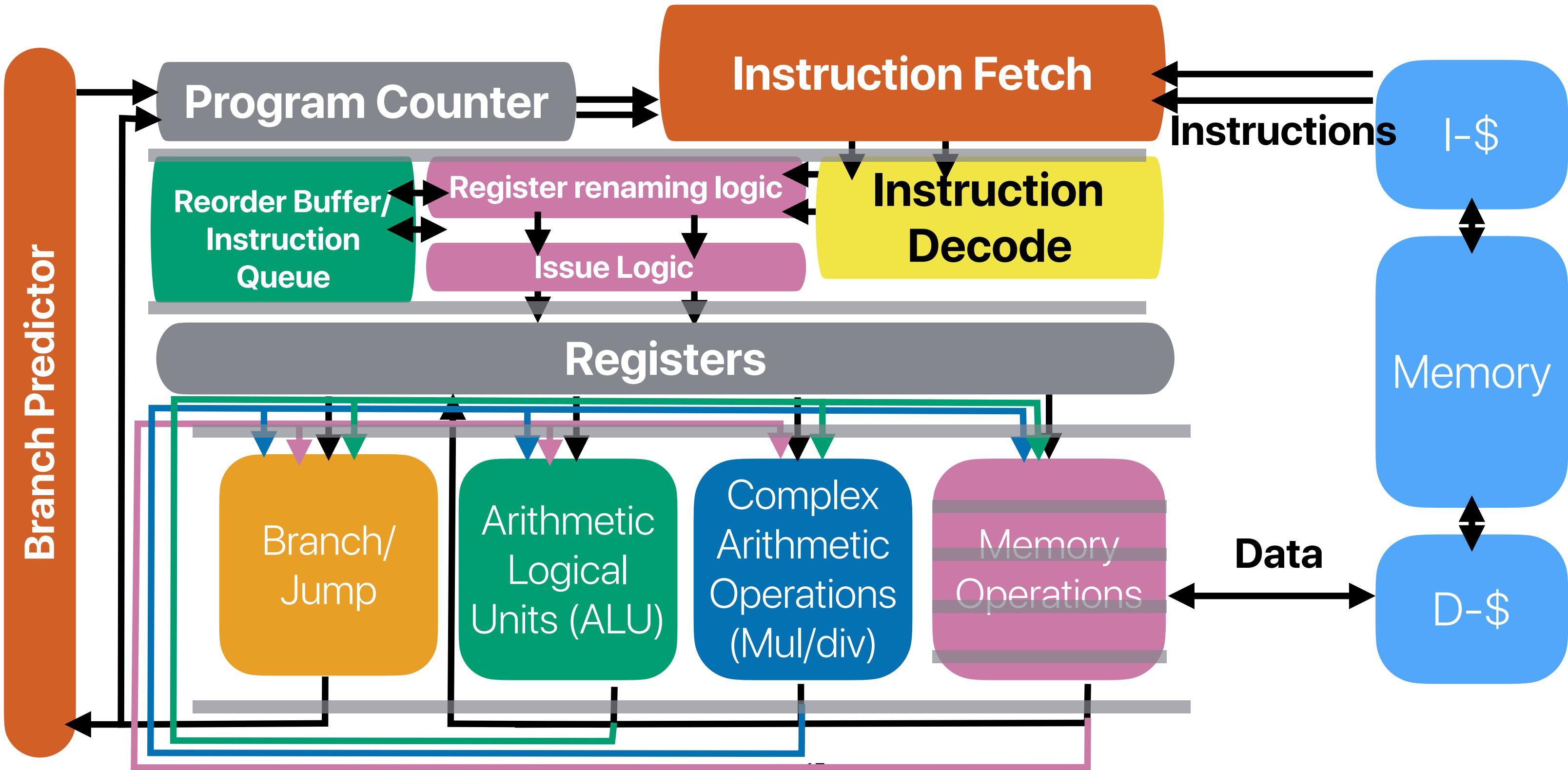
SMT from the user/OS' perspective



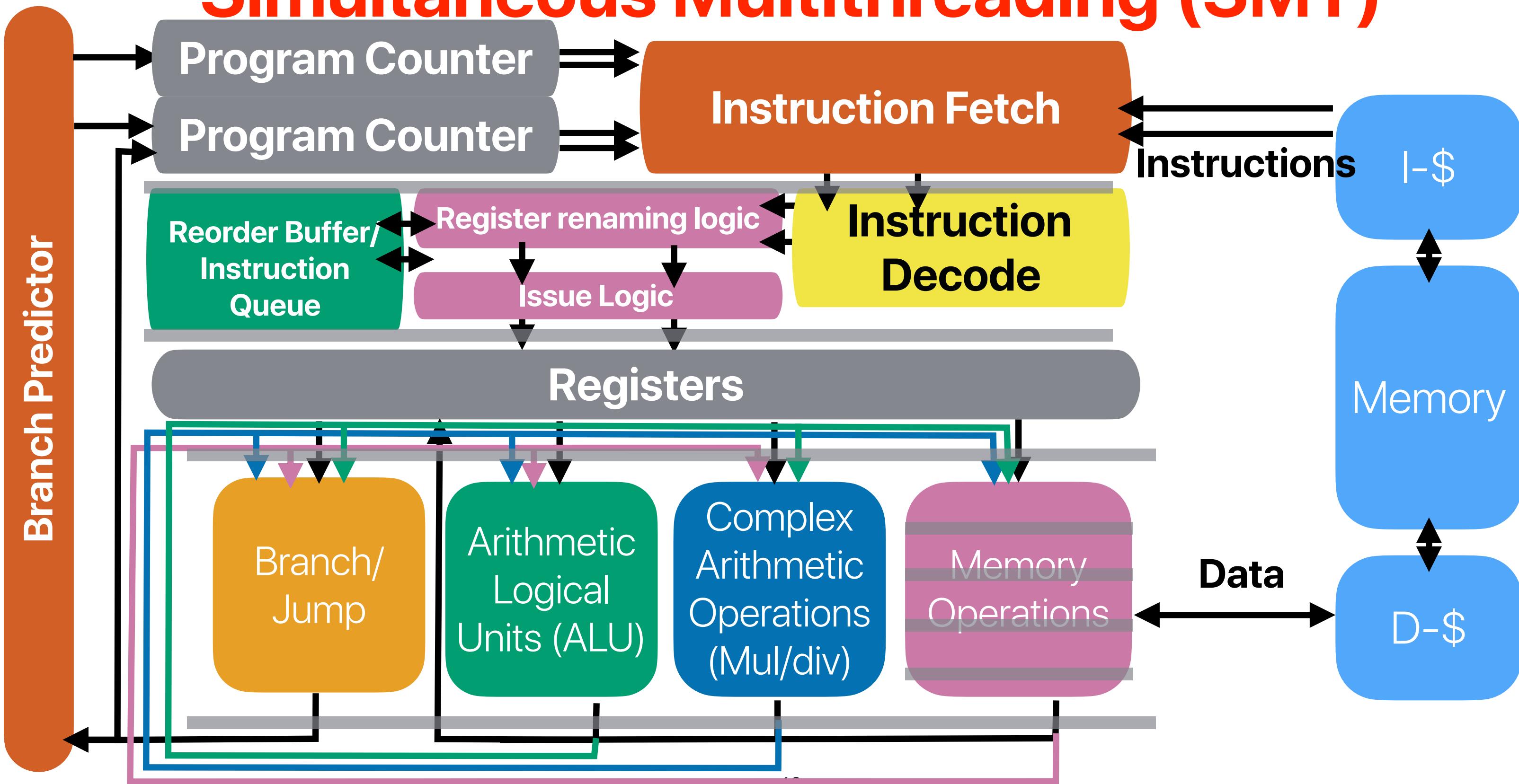
How do we support two running programs in one pipeline?

- We need two program counters
- We need two sets of architectural to physical register mappings
- We do not need
 - Duplicated cache — virtually indexed, physically tagged cache already addressed that
 - Duplicated pipeline functional units — isn't sharing the whole purpose?
 - Duplicated reorder buffer — you simply need to tag which process the instruction belongs to

Recap: Register renaming



Simultaneous Multithreading (SMT)

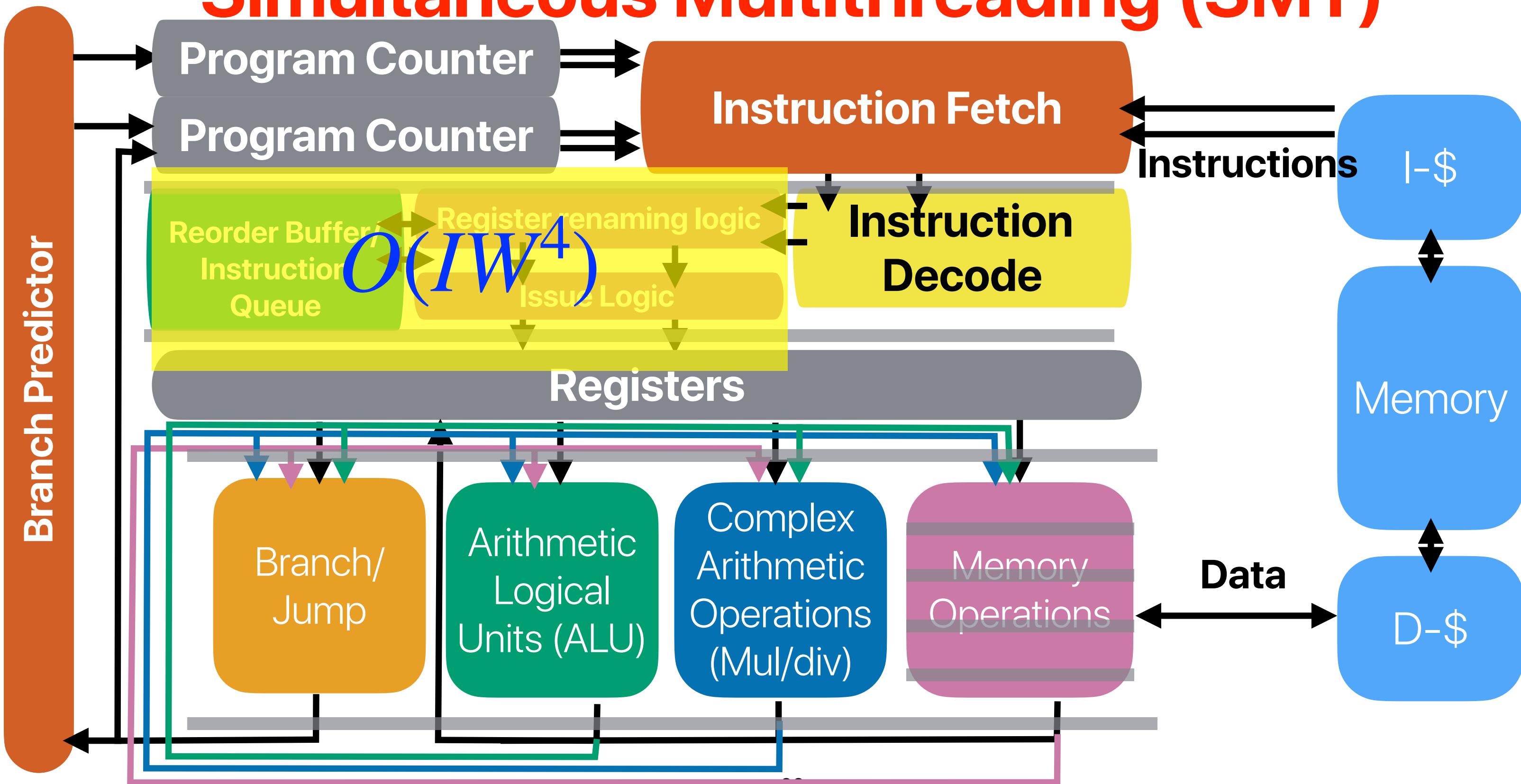


Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	39 bits physical, 48 bits virtual
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	151
Model name:	12th Gen Intel(R) Core(TM) i3-12100F
Stepping:	5
CPU MHz:	3300.000
CPU max MHz:	5500.0000
CPU min MHz:	800.0000
BogoMIPS:	6604.80
Virtualization:	VT-x
L1d cache:	192 KiB
L1i cache:	128 KiB

SMT in real practice

- Intel HyperThreading (supports up to two threads per core)
 - Intel Pentium 4, Intel Atom, All Intel Core i7s and Core i9s
- AMD RyZen (Zen microarchitecture)
- If you see a processor with “threads” more than “cores”, that must because of SMT!

Simultaneous Multithreading (SMT)



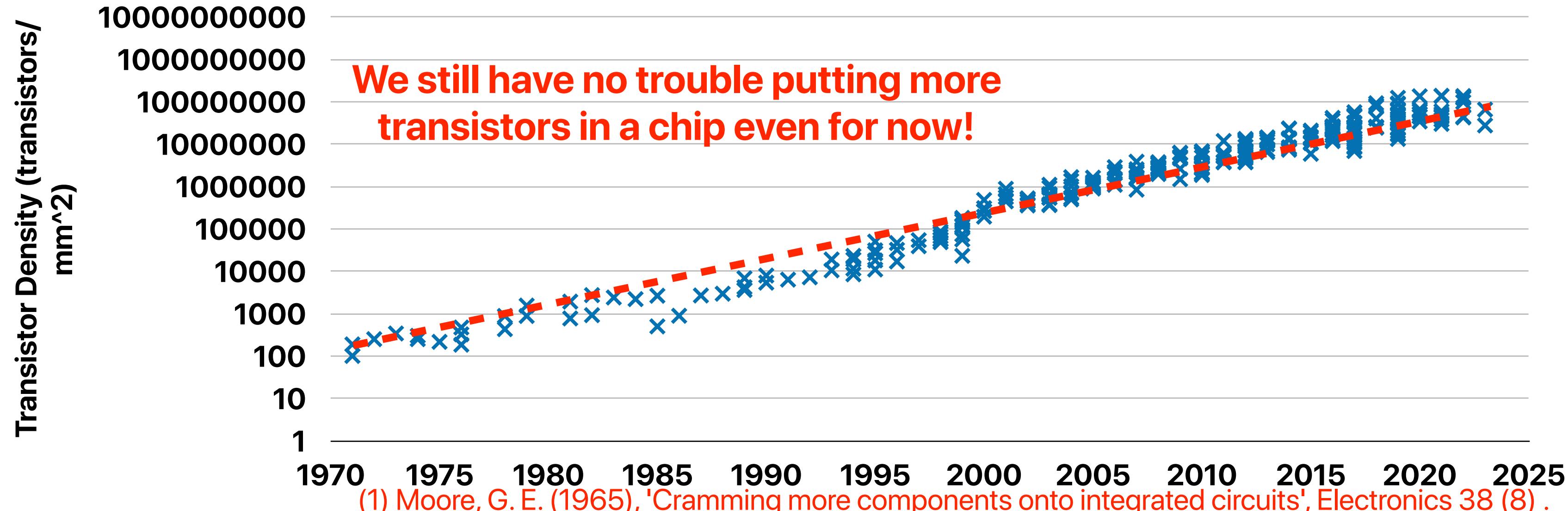
Takeaways: parallel architectures

- SMT processors can better utilize the pipeline resources by allowing simultaneous execution of multiple threads
 - Improved execution throughput
 - May hurt the latency of each thread since we share functional units & cache

Chip-Multiprocessors (CMP) or Multi-core processors

Recap: Moore's Law⁽¹⁾

- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains



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 process. Coffee Lake processors introduced i5 and i7 CPUs featuring six cores (along with hyper-threading in the case of the latter) and no hyperthreading.



The transistor count per core on Coffee Lake is lower than that of some other modern processors, such as the Ryzen 5 5600X, which has 390 million transistors per core. However, Coffee Lake still offers good performance, thanks to its high clock speeds and efficient design.

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
Core i7 12700K	475.8 million

Nehalem Alder Lake
6-issue 12-issue

Recap: do we really need very wide issue processors?

	ET	IC	IPC/ILP	# of branches	Branch mis-prediction rate
A	22.21	332 Trillions	2.88	65 Trillions	1.13%
B	12.29	287 Trillions	4.52	17 Trillions	0.04%
C	5.01	102 Trillions	3.95	17 Trillions	0.04%
D	3.73	80 Trillions	4.13	1 Trillions	~0%
E	54.4	173 Trillions	0.61	44 Trillions	18.6%
SSE4.2	1.57	22 Trillions	2.7	1 Trillions	~0%

One powerful core or two “good enough” cores?

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 process. Lake processors introduced i5 and i7 CPUs featuring six cores (along with hyper-threading in the case of the latter) and no hyperthreading.



The transistor count per core on Coffee Lake is lower than that of some other modern processors, such as the Ryzen 5 5600X with 390 million transistors per core. However, Coffee Lake still offers good performance, thanks to its high clock speeds and efficient power delivery.

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
Core i7 4790K	475.8 million

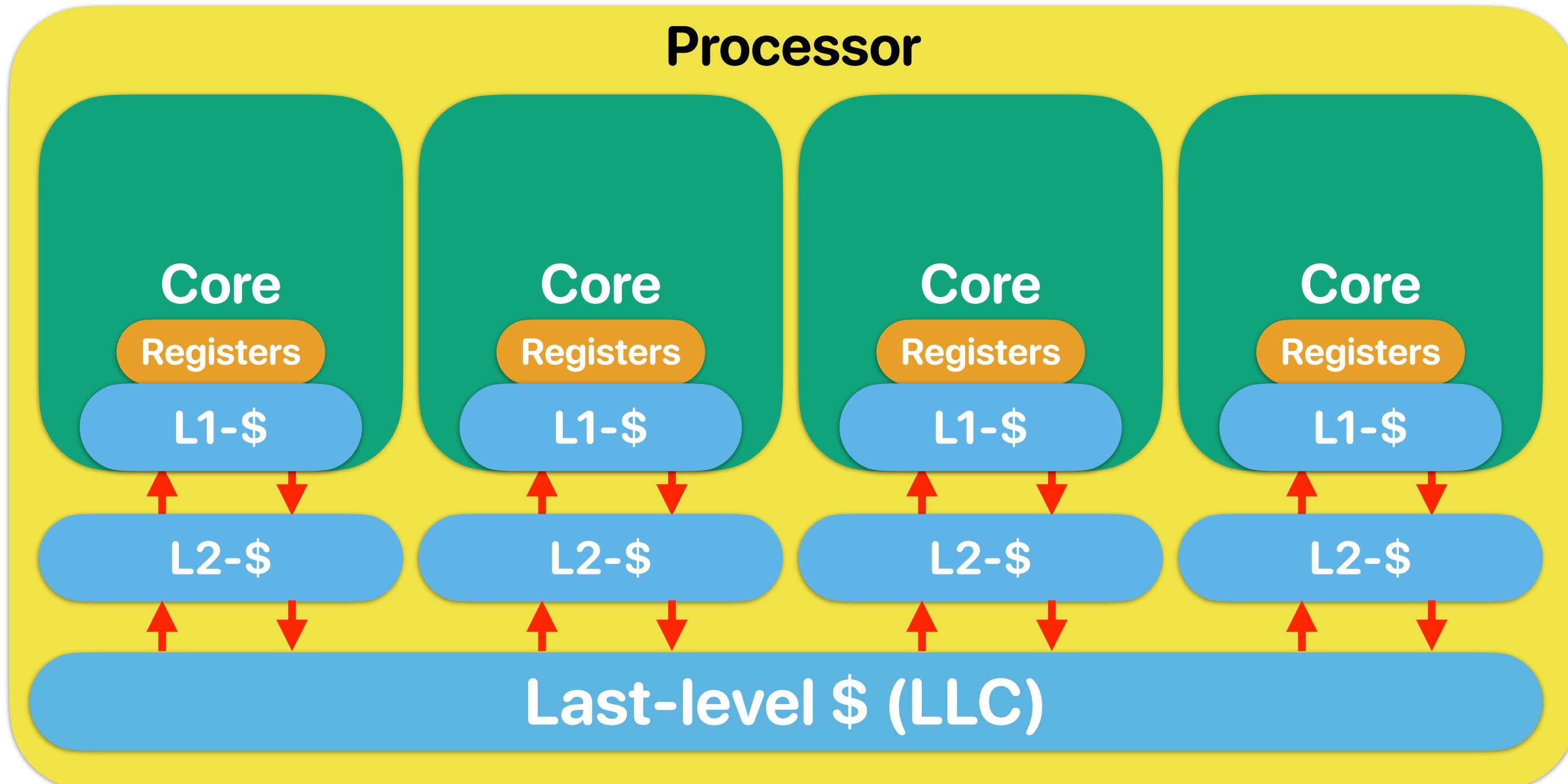
2x 3-issue ALUs Nehalem

Nehalem Alder Lake Nehalem
6-issue 12-issue 6-issue

1x 5-issue ALUs Alder Lake

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

Concept of CMP



CMP from the user/OS' perspective

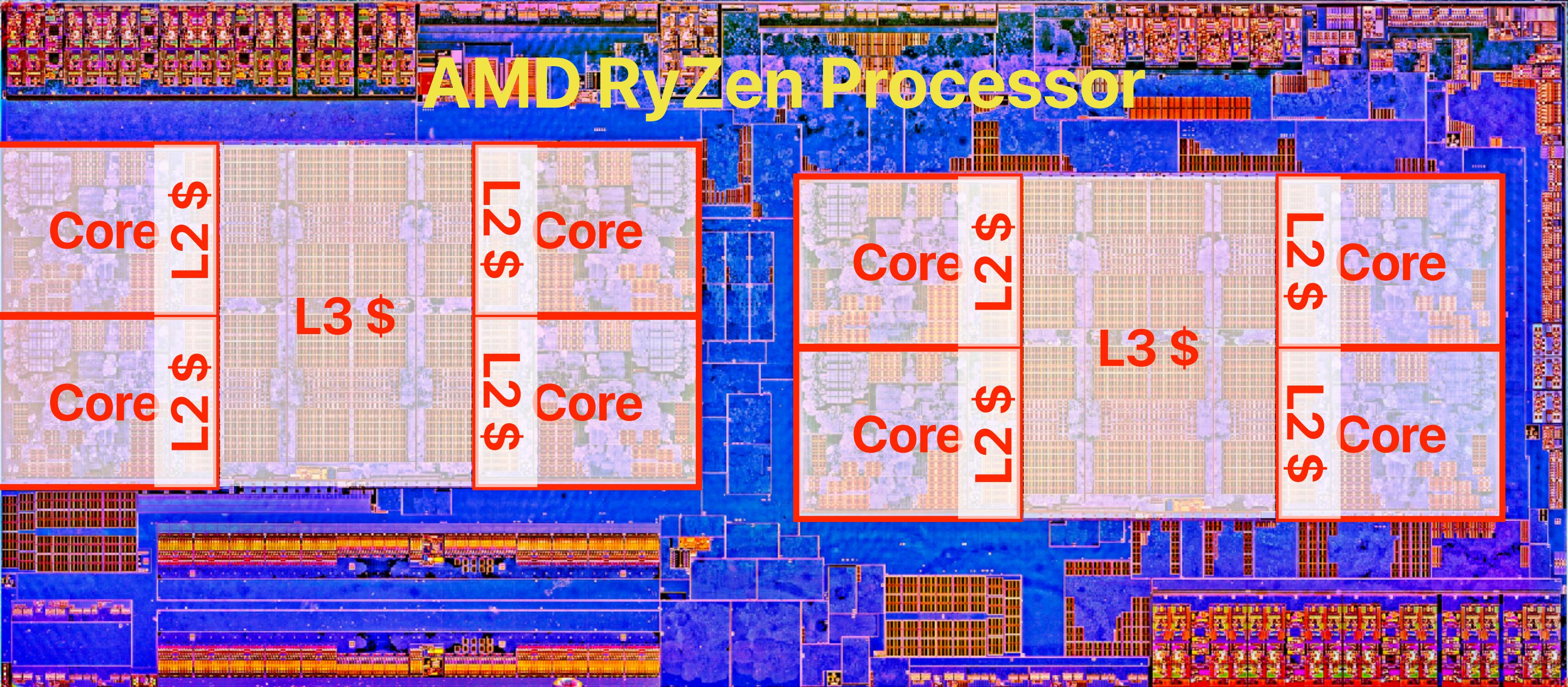


Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	39 bits physical, 48 bits virtual
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	151
Model name:	12th Gen Intel(R) Core(TM) i3-12100F
Stepping:	5
CPU MHz:	3300.000
CPU max MHz:	5500.0000
CPU min MHz:	800.0000
BogoMIPS:	6604.80
Virtualization:	VT-x
L1d cache:	192 KiB
L1i cache:	128 KiB

Modern processors have both CMP/SMT



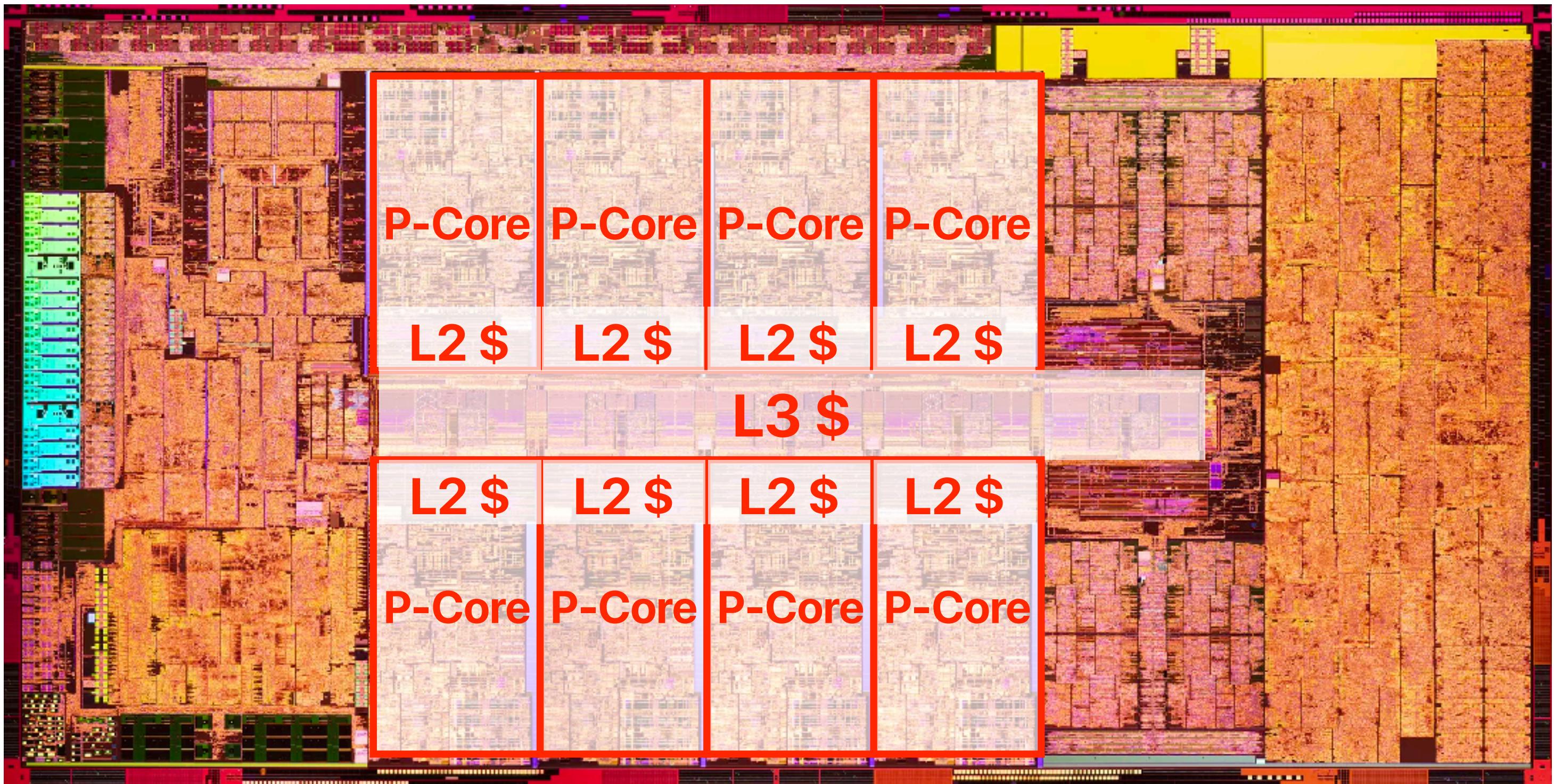
AMD Ryzen Processor



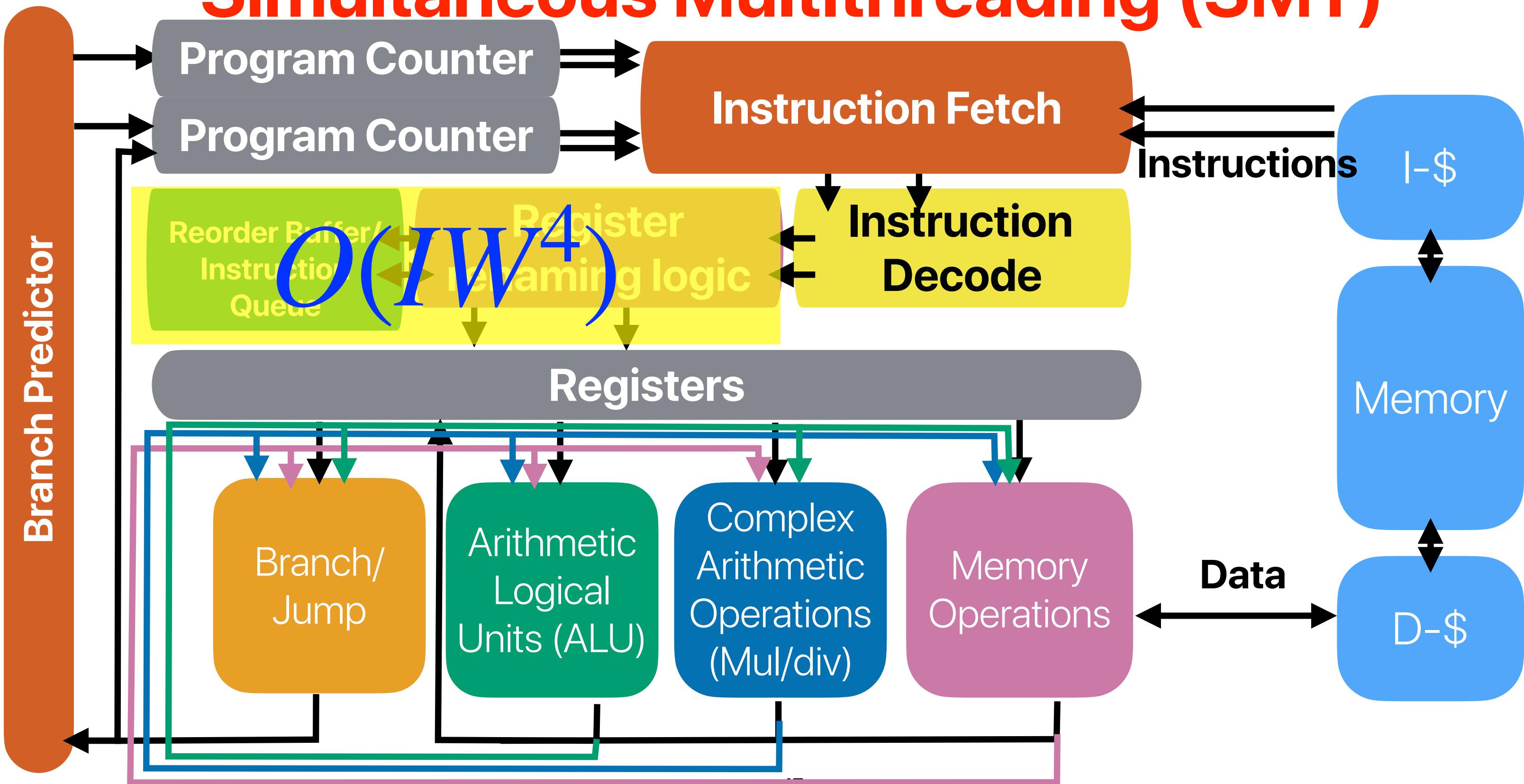
AMD

RYZEN

Intel Alder Lake



Simultaneous Multithreading (SMT)



Takeaways: parallel architectures

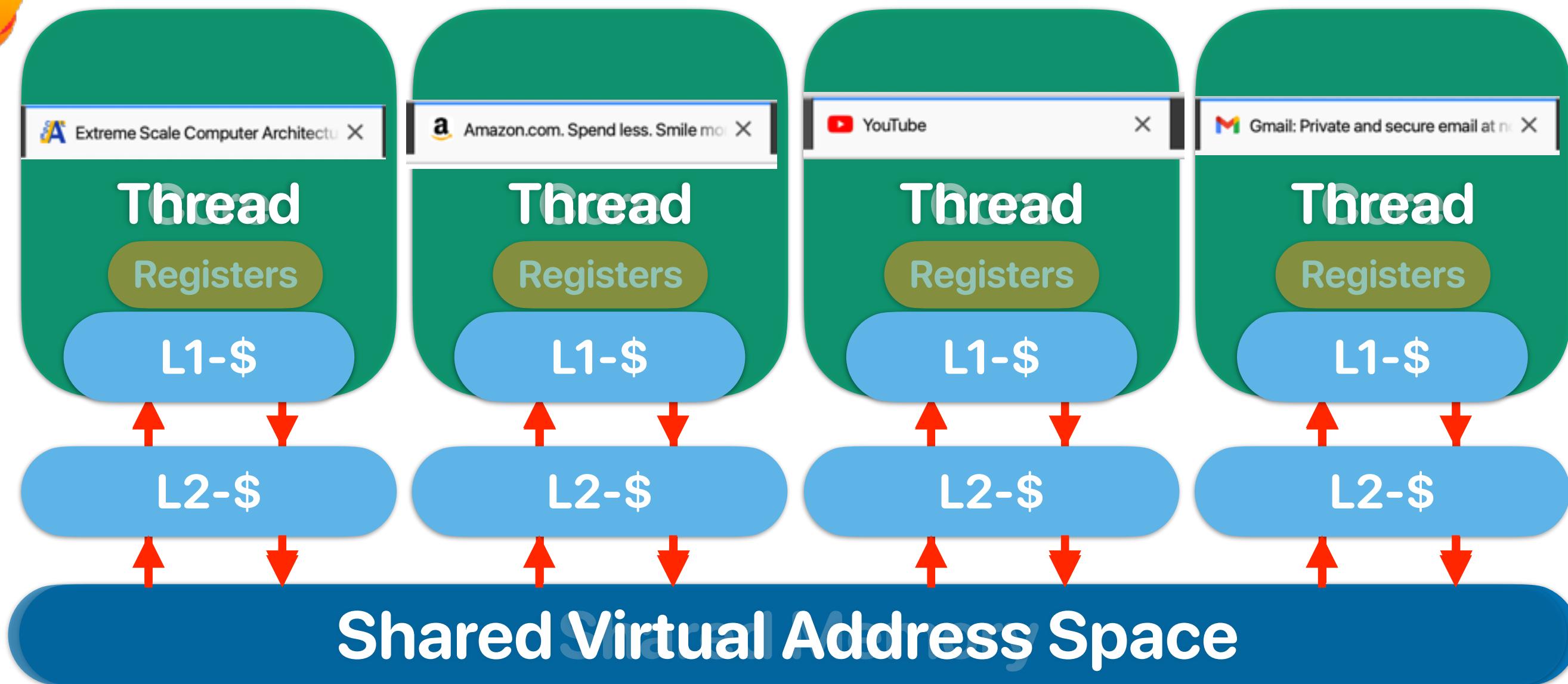
- SMT processors can better utilize the pipeline resources by allowing simultaneous execution of multiple threads
 - Improved execution throughput
 - May hurt the latency of each thread since we share functional units & cache
- CMP provides more processor cores for parallel threads or multiprogrammed environments
 - More isolated, protected pipeline
 - No flexibility if the running program needs more resource

Parallel Programming & Architectural Supports for Parallel Programming

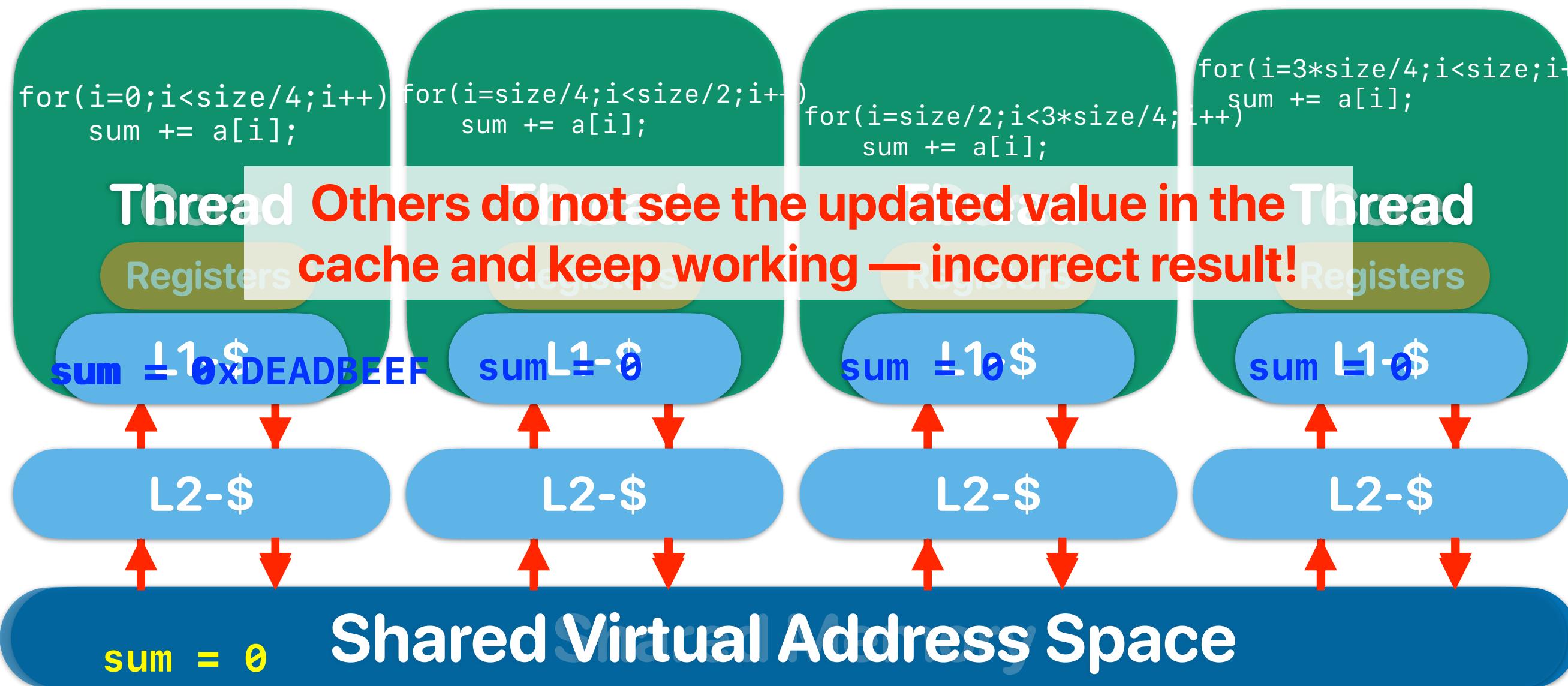
Parallel programming

- 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
 - Programming model: MPI, Spark
- Threads (in OS/software systems)
 - Independent portions of your program that can run in parallel
 - All threads share the same virtual memory space
 - Programming model: pthread or openmp
- 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

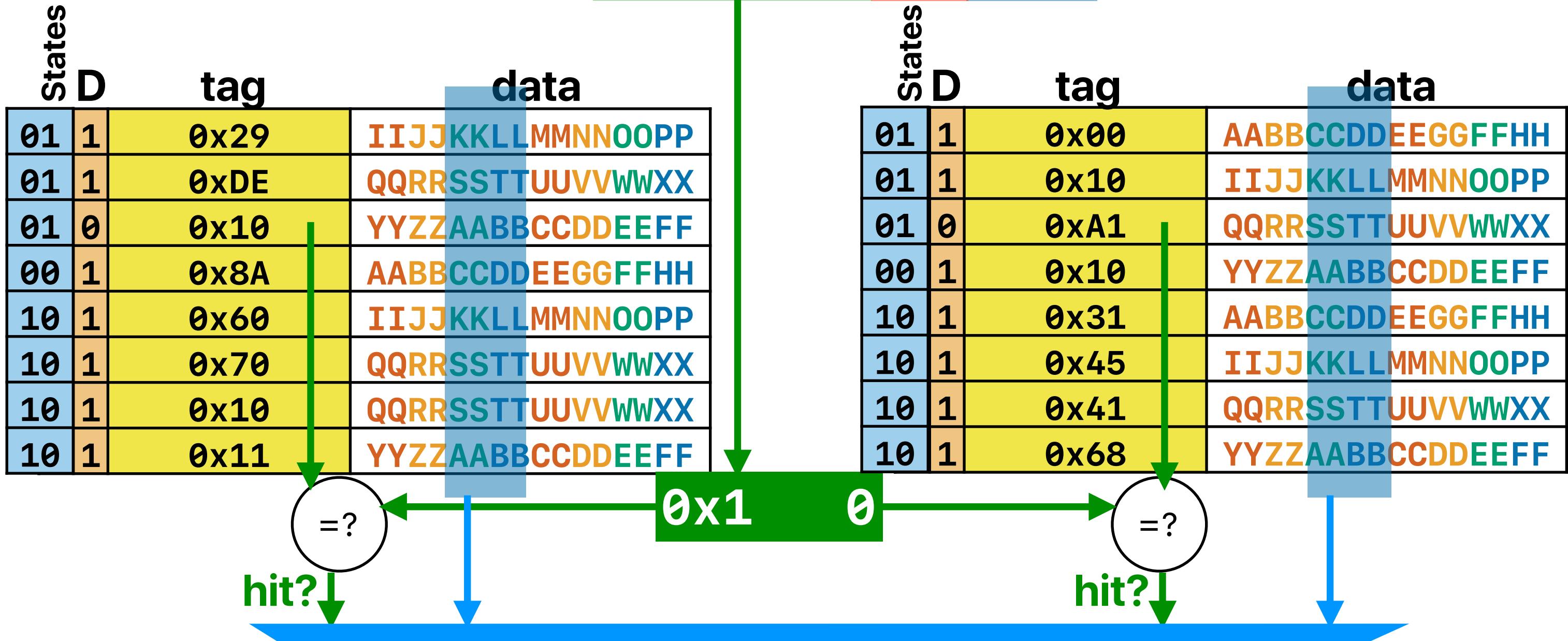
Coherent way-associative cache

memory address:

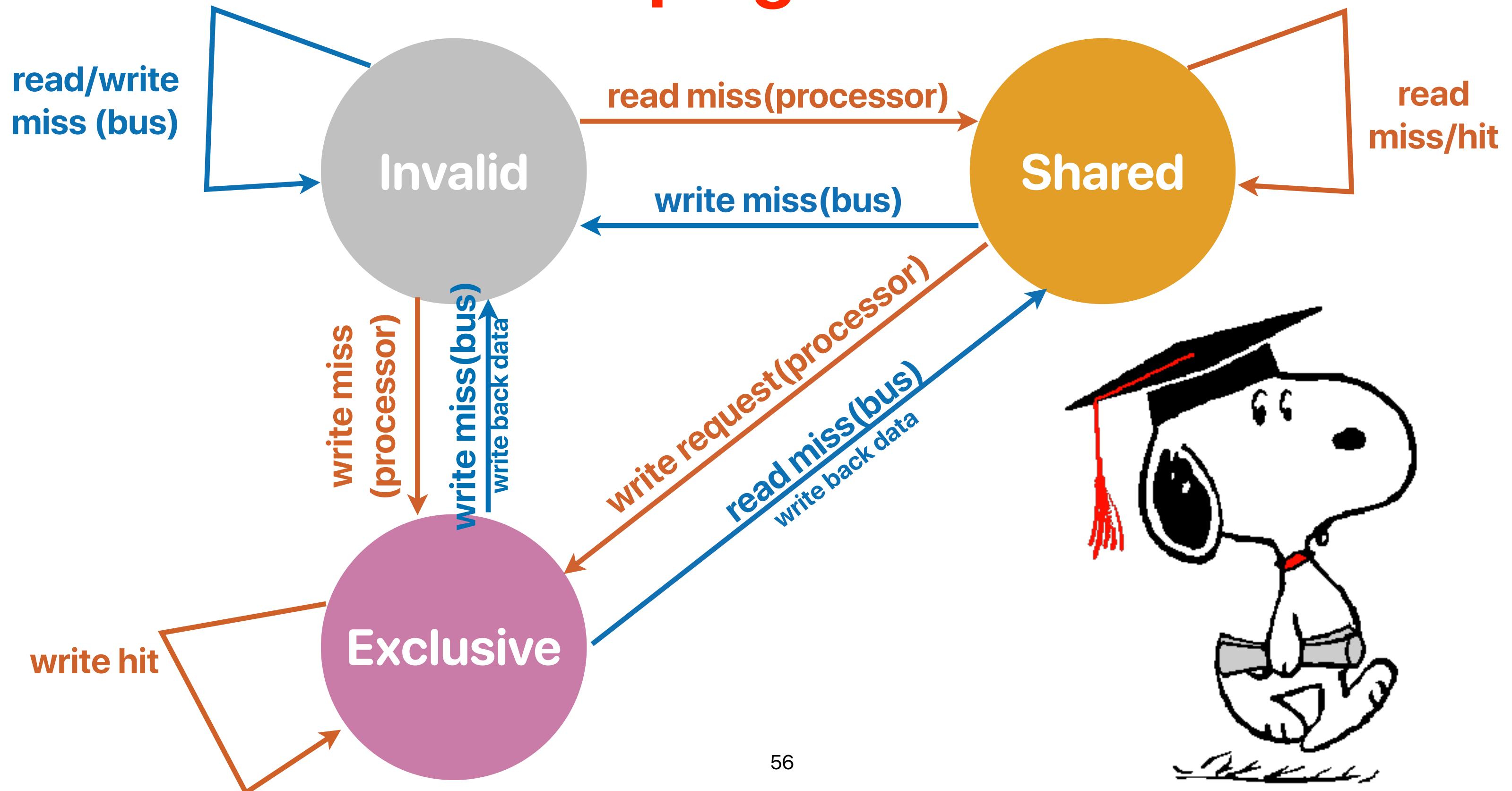
$0x0$ 8 tag 2 set 4 block
index offset

memory address:

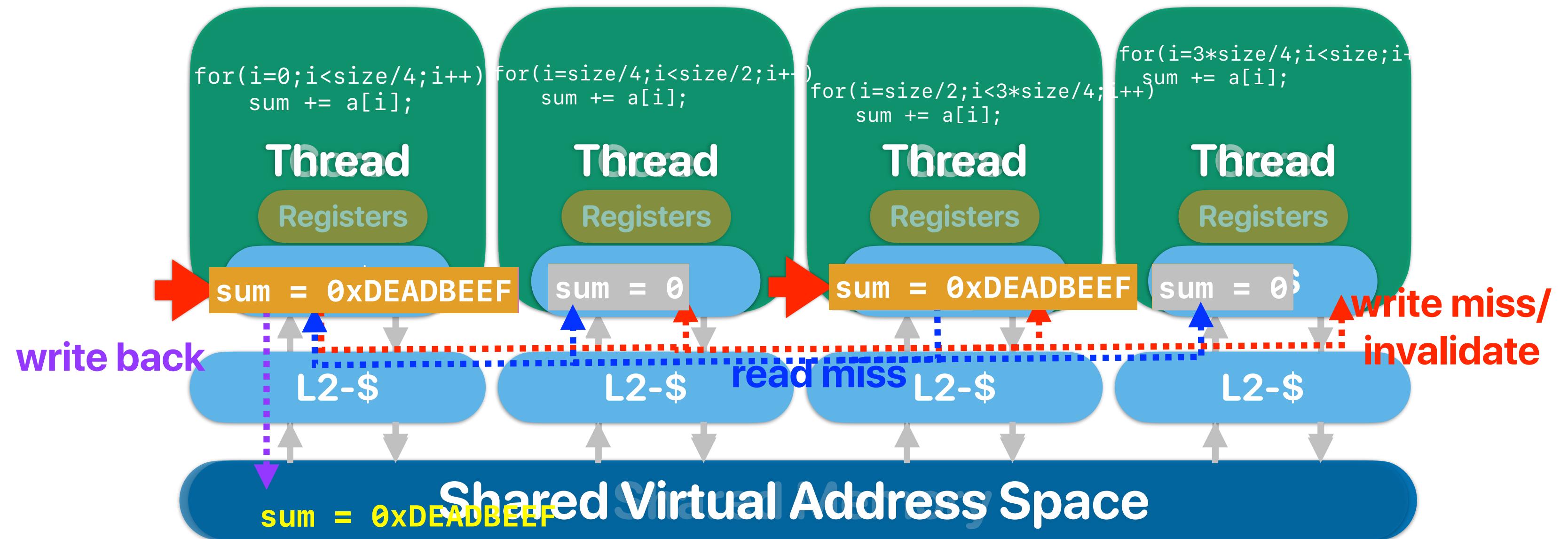
0b0000100000100100



Snooping Protocol



What happens when we write in coherent caches?



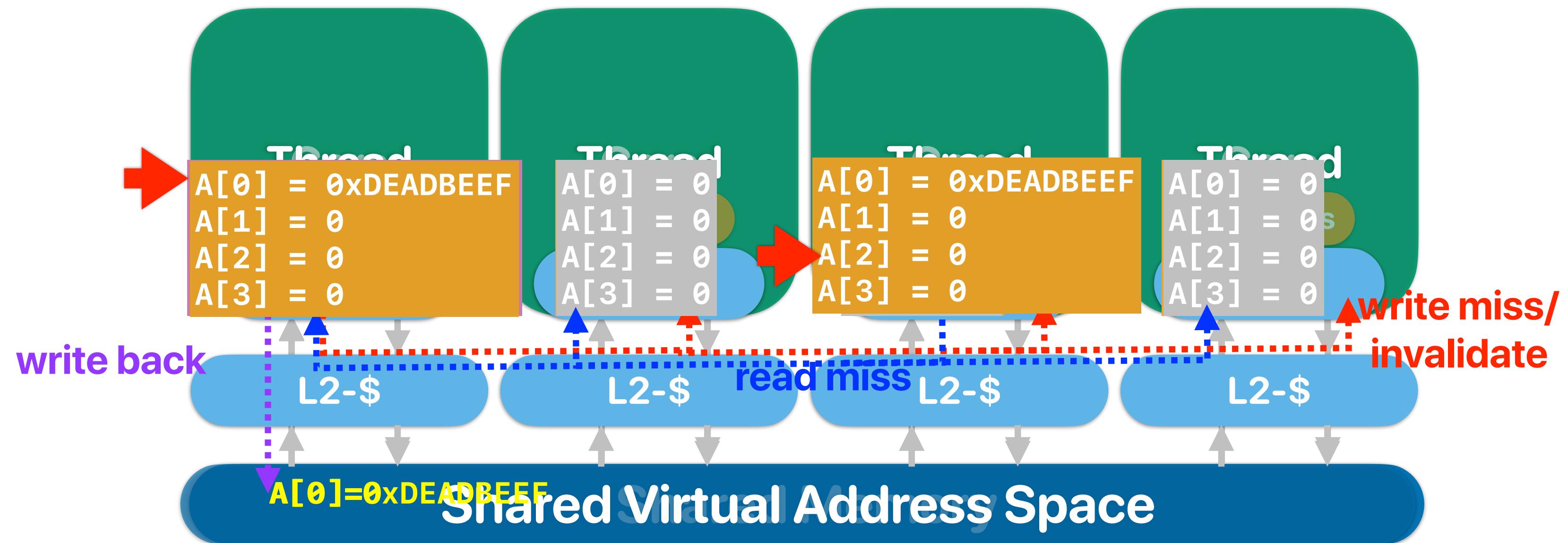
Observer

thread 1	thread 2
<pre>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",</pre>	<pre>void* modifyloop(void *x) { sleep(1); printf("Please input a number:\n"); scanf("%d", &loop); return NULL; }</pre>

Takeaways: parallel architectures

- SMT processors can better utilize the pipeline resources by allowing simultaneous execution of multiple threads
 - Improved execution throughput
 - May hurt the latency of each thread since we share functional units & cache
- CMP provides more processor cores for parallel threads or multiprogrammed environments
 - More isolated, protected pipeline
 - No flexibility if the running program needs more resource
 - Cache coherence can guarantee that everyone would eventually have a coherent view of data, but not when

Cache coherency



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);
```

Take-aways: parallel programming

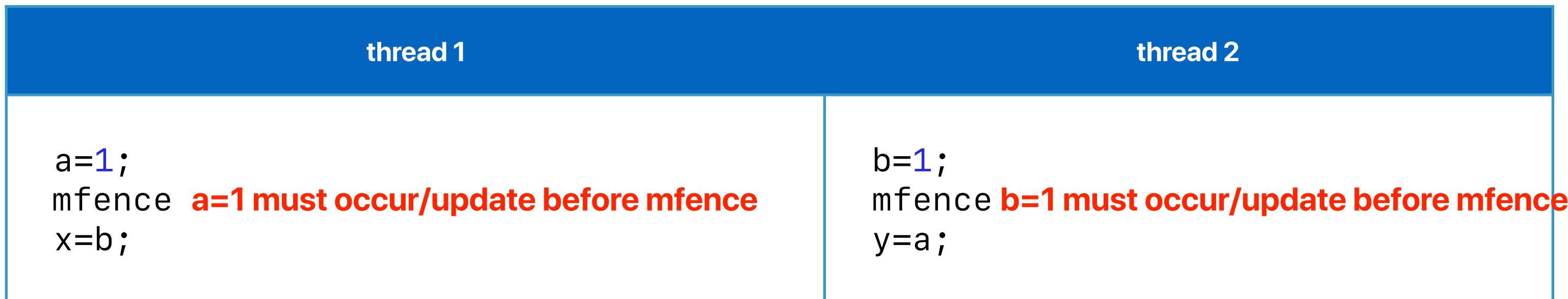
- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/misses if you do it wrong

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/misses if you do it wrong
- Processor behaviors are non-deterministic
 - You cannot predict which processor is going faster
 - You cannot predict when OS is going to schedule your thread
 - You cannot predict when the processor is going to schedule an instruction

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



Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache coherency may create unexpected cache invalidations/misses if you do it wrong
- Processor behaviors are non-deterministic
 - You cannot predict which processor is going faster
 - You cannot predict when OS is going to schedule your thread
 - You cannot predict when the processor is going to schedule an instruction
- Cache consistency is hard to support

Beyond “scalar”

Gemini was just updated. See update.

Hello, Hung-Wei

How can I help you today?

Explain what the keto diet is in simple terms

Teach me to make homemade ice cream

Come up with a product name for a new app

Write a descrij type o



Humans review some saved chats to improve Google AI. To stop this for future chats, turn off Gemini Apps Activity. If this setting is on, don't enter info you wouldn't want reviewed or used. [How it works](#)

[Manage Activity](#) [Dismiss](#)

G



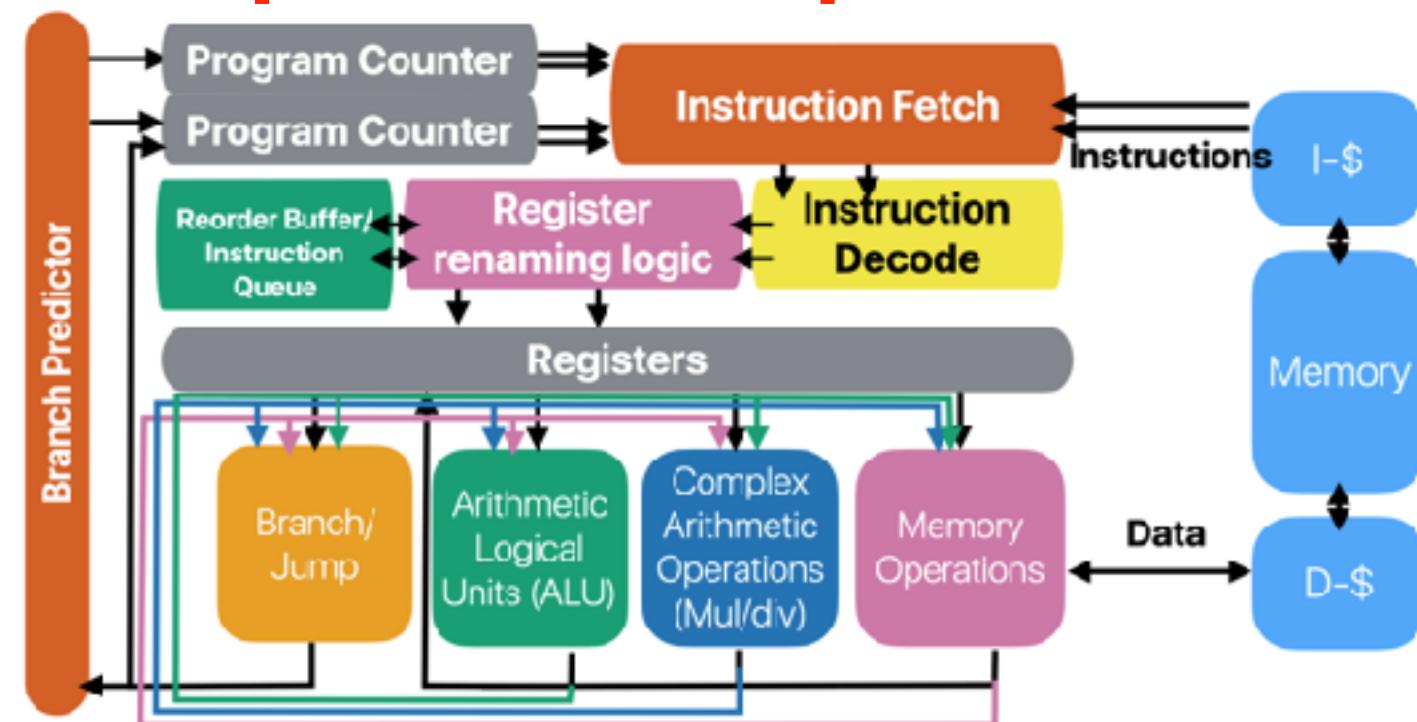
- But how do we process them using “superscalar” processors?

```
for(uint64_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint64_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

How well does vector processing map to super “scalar” processors

```
for(uint64_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint64_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

Assume we have a 5-issue INT, 3-load, 4-store pipeline like Alder Lake



Performance is very limited by both the memory bandwidth and available functional units

load vector[0]	load matrix[0][1]	load vector[3]	load matrix[0][4]
load matrix[0][0]	load vector[2]	load matrix[0][3]	load vector[5]
load vector[1]	load matrix[0][2]	load vector[4]	load matrix[0][5]
	mul matrix[0][0], vector[0]	mul matrix[0][1], vector[1]	mul matrix[0][3], vector[3]
		mul matrix[0][2], vector[2]	

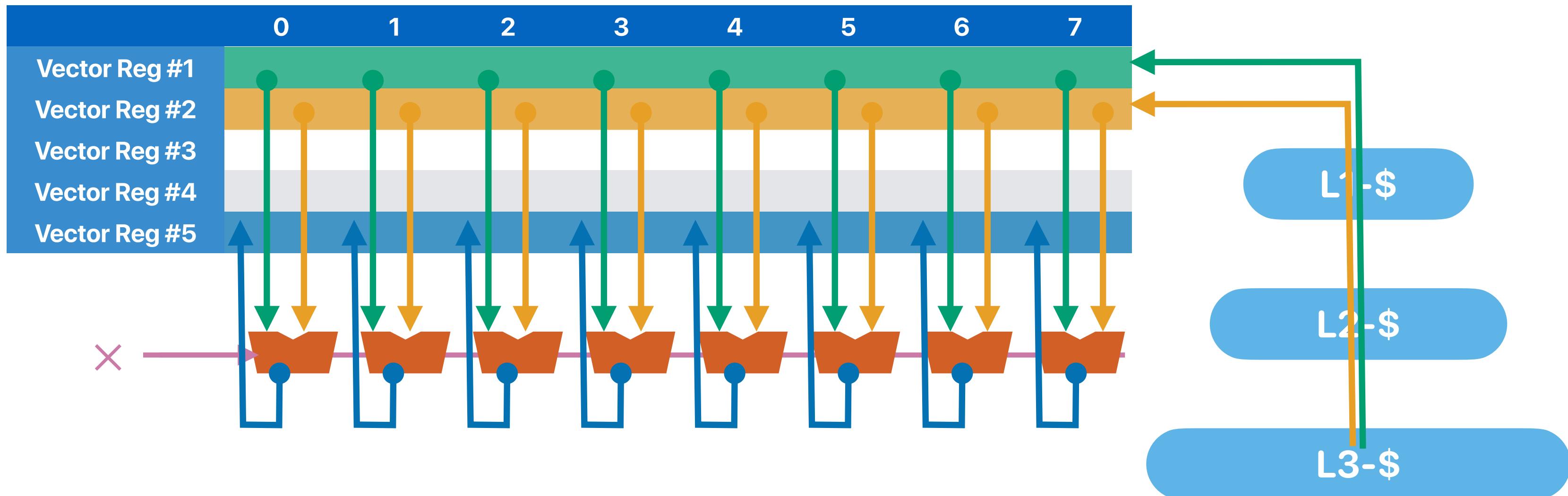
Characteristics of vector processing

- Their operations are uniform across all pairs of elements
 - **Can we have just one instruction to control all pair-wise operations?**
 - There are very limited vector operations with mathematical meanings
 - **Can we simplify the processing elements design and make space for more PEs?**
 - They have very good spatial locality
 - **Can we have fetch them once & put in wide registers?**

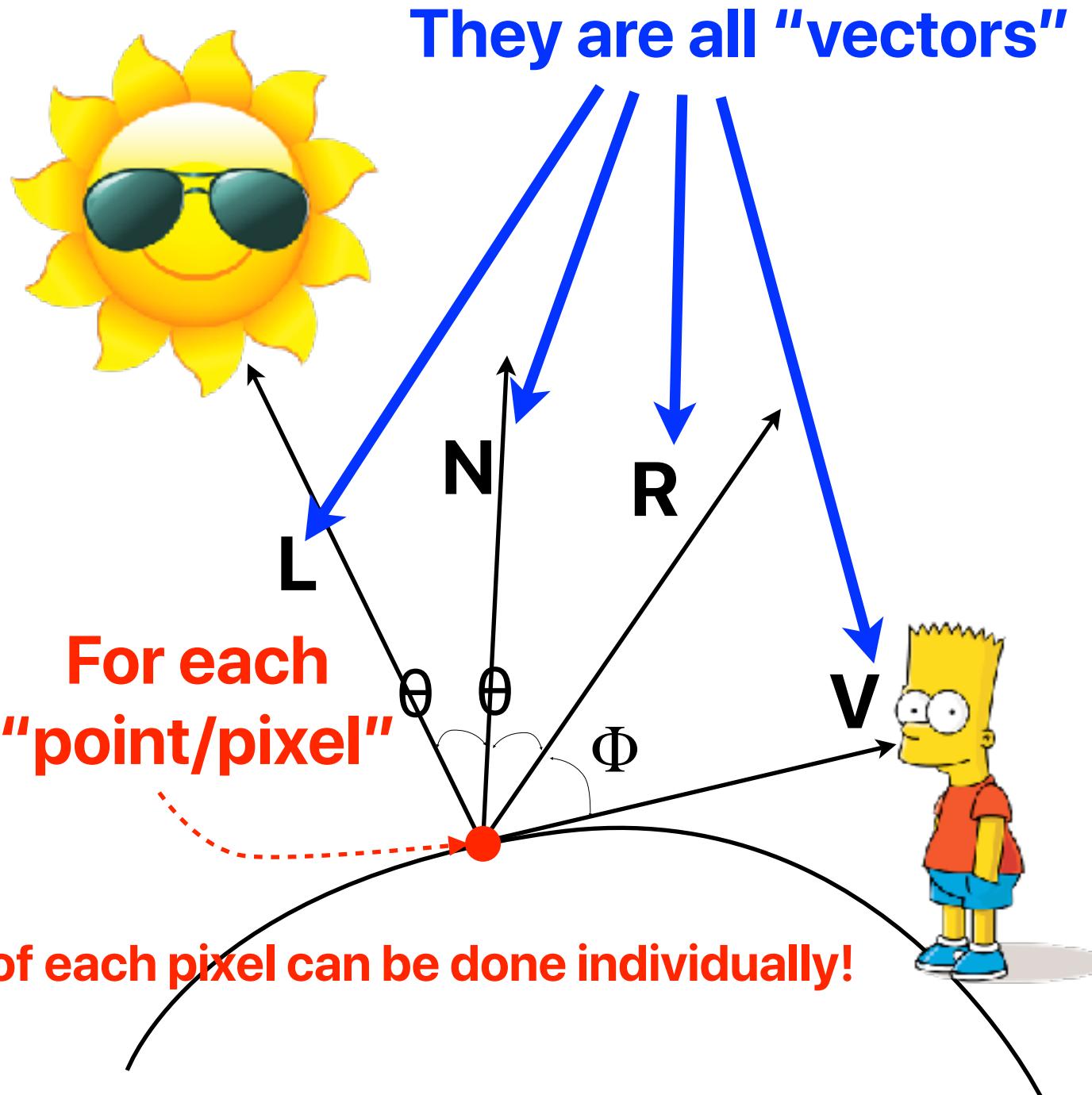
```
for(uint64_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint64_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```

load vector[0]	load matrix[0][1]	load vector[3]	load matrix[0][4]
load matrix[0][0]	load vector[2]	load matrix[0][3]	load vector[5]
load vector[1]	load matrix[0][2]	load vector[4]	load matrix[0][5]
	mul matrix[0][0], vector[0]	mul matrix[0][1], vector[1]	mul matrix[0][3], vector[3]
		mul matrix[0][2], vector[2]	

Vector processing architecture



Basic concept of shading



$$I_{amb} = K_{amb} \cdot M_{amb}$$

$$I_{diff} = K_{diff} \cdot M_{diff} \cdot (N \cdot L)$$

$$I_{spec} = K_{spec} \cdot M_{spec} \cdot (R \cdot V)^n$$

$$I_{total} = I_{amb} + I_{diff} + I_{spec}$$

```
void main(void)
{
    // normalize vectors after interpolation
    vec3 L = normalize(o_toLight);
    vec3 V = normalize(o_toCamera);
    vec3 N = normalize(o_normal);

    // get Blinn-Phong reflectance components
    float Iamb = ambientLighting();
    float Idif = diffuseLighting(N, L);
    float Ispe = specularLighting(N, L, V);

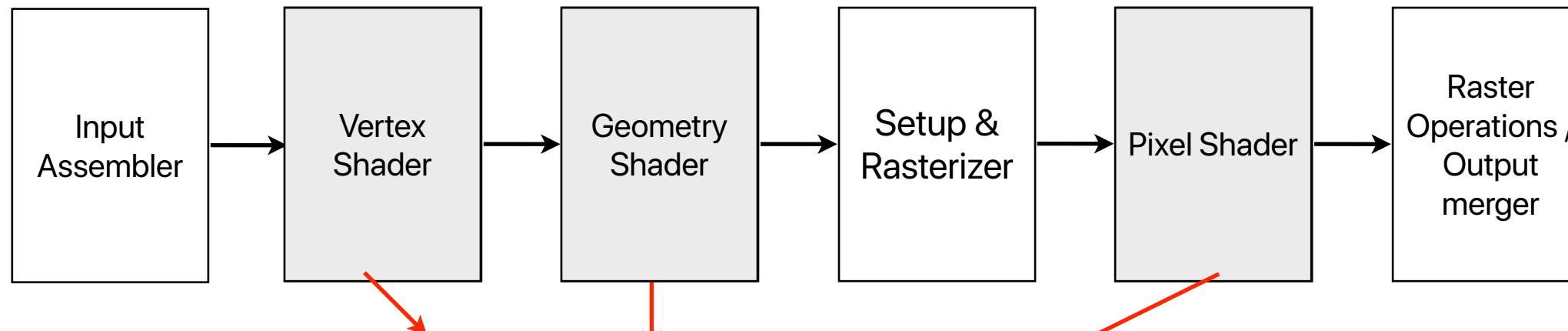
    // diffuse color of the object from texture
    vec3 diffuseColor = texture(u_diffuseTexture, o_texcoords);

    // combination of all components and diffuse color of the
    resultingColor.xyz = diffuseColor * (Iamb + Idif + Ispe);
    resultingColor.a = 1;
```

GPU (Graphics Processing Unit)

- Originally for displaying images
- HD video: 1920×1080 pixels * 60 frames per second
 - Therefore, GPU is not latency-oriented by design!
 - Even for 120 frames, you still have 8ms latency to get everything done!
- Graphics processing pipeline

1 GHz can give you 8000000 cycles!!!



These shaders need to be “programmable” to apply different rendering effects/algorithms
(Phong shading, Gouraud shading, and etc...)

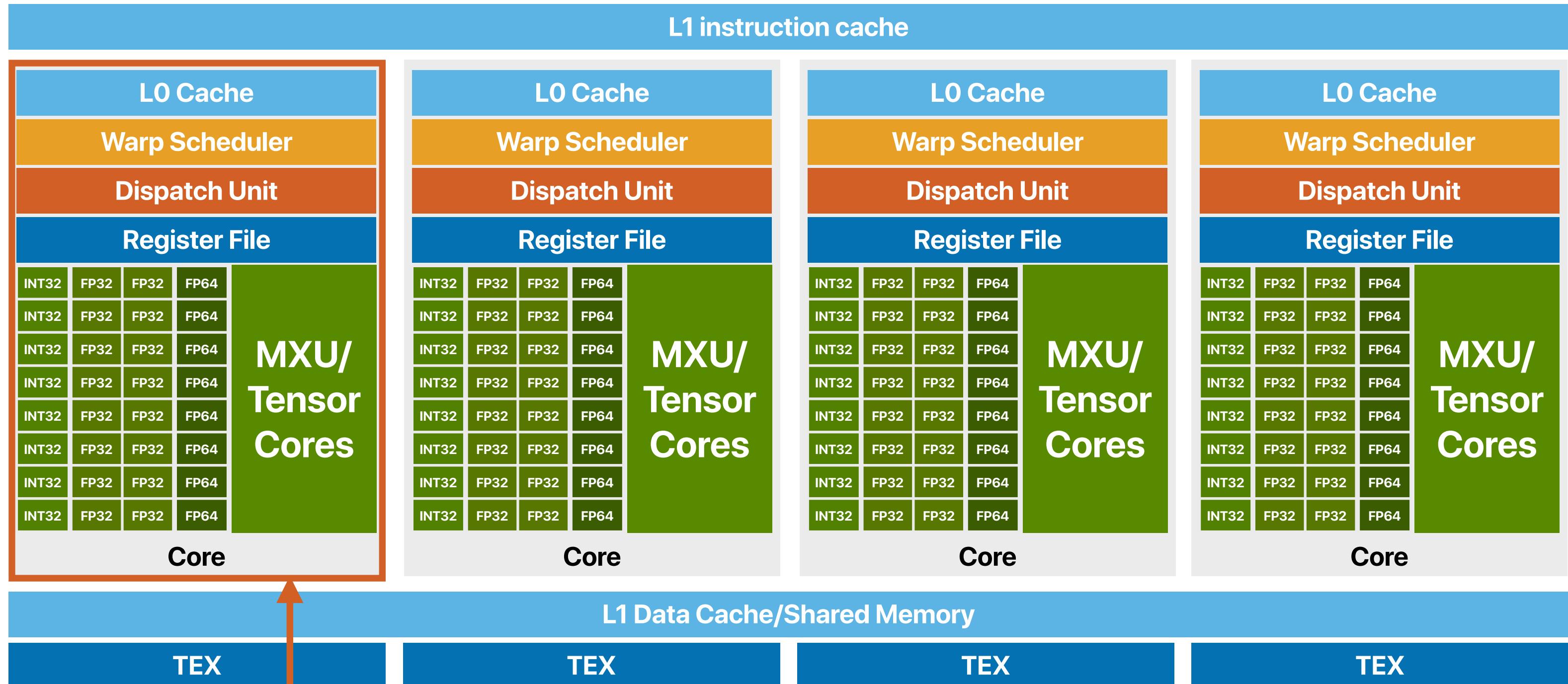
What GPU architectures should look like?

- Based on the original target of GPU design, what do you think would be reasonable design decisions that GPU architectures make?
 - ① The architecture should render pixels as fast as possible
 - ② The architecture does not need a branch unit
 - ③ The architecture should contain an array of powerful ALUs and functional units that can perform a rich set of operations
 - ④ The architecture should offer very large bandwidth of memory accesses
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

What's the “appropriate” GPU architecture

- Lots of ALUs to process pixels in parallel — 2M pixels in HD resolution, very regular workloads
 - Vector processing model
- Simple operations
 - The ALUs only supports very few instructions
 - Almost no branches
- Deadline driven and throughput-oriented rather than latency oriented
 - High-bandwidth but also “higher-latency” memory
 - ALUs can be slower

GPU Architecture



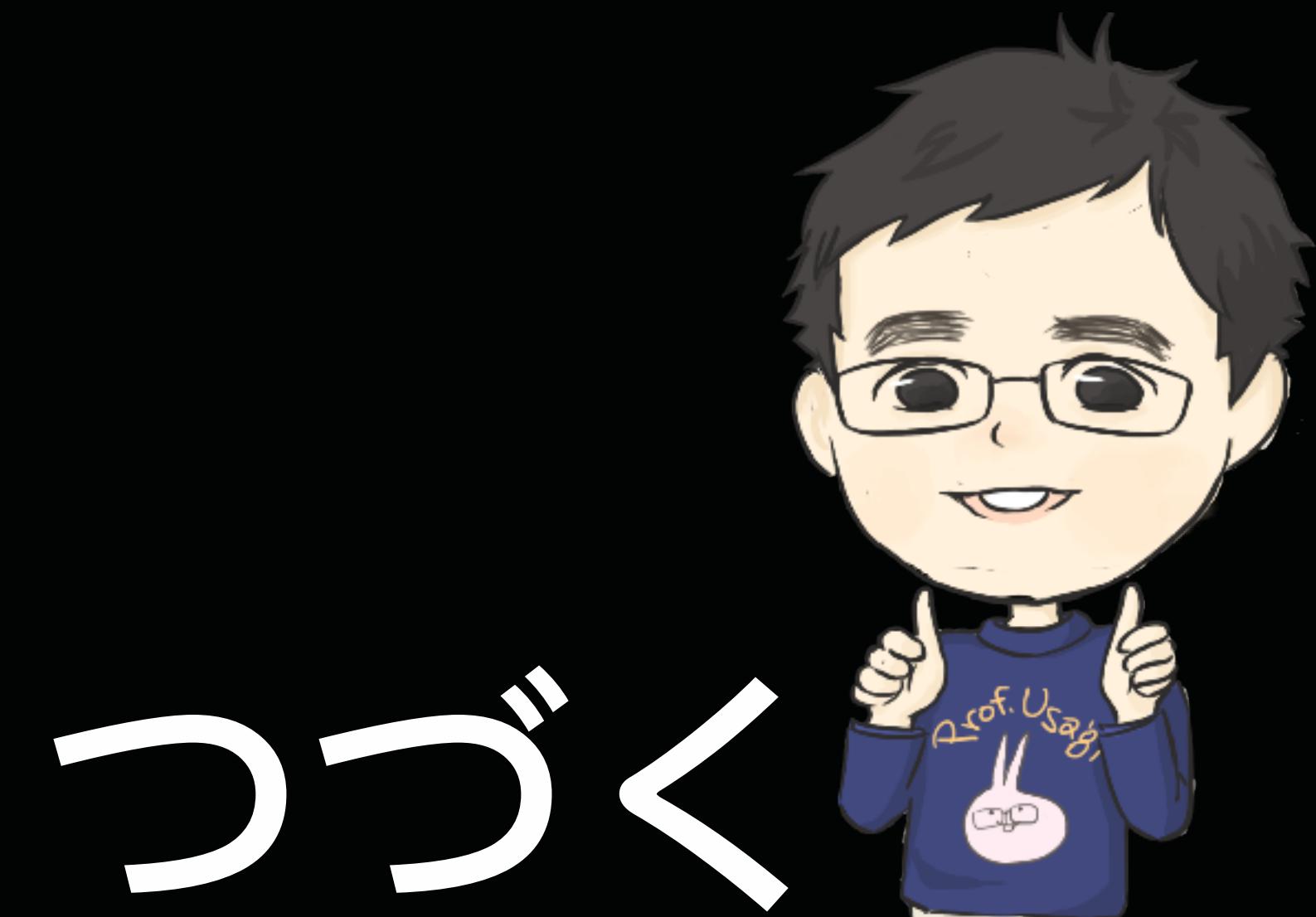
Each core is a "vector processing" unit

Announcements

- **Reading Quiz 8 (last reading quiz)** due **next Tuesday** before the lecture
- **Assignment 4 is released** due this Saturday
- **Assignment 5 is released** due 9/5/2024

Computer Science & Engineering

142



つづく