Multithreaded Architectures and Programming on Modern Processors: the Multithreaded Version

Hung-Wei Tseng

Recap: Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Tips of programming on modern processors

- Minimize the critical path operations
  - Don’t forget about optimizing cache/memory locality first!
    - Memory latencies are still way longer than any arithmetic instruction
    - Can we use arrays/hash tables instead of lists?
  - Branch can be expensive as pipeline get deeper
    - Sorting
    - Loop unrolling
  - Still need to carefully avoid long latency operations (e.g., mod)
- Since processors have multiple functional units — code must be able to exploit instruction-level parallelism
  - Hide as many instructions as possible under the “critical path”
  - Try to use as many different functional units simultaneously as possible
- Modern processors also have accelerated instructions
- Compiler can do fairly go optimizations, but with limitations

3:00

If you have 2 millions, are you going to buy (1) one house in San Diego or (2) two apartments in San Diego?

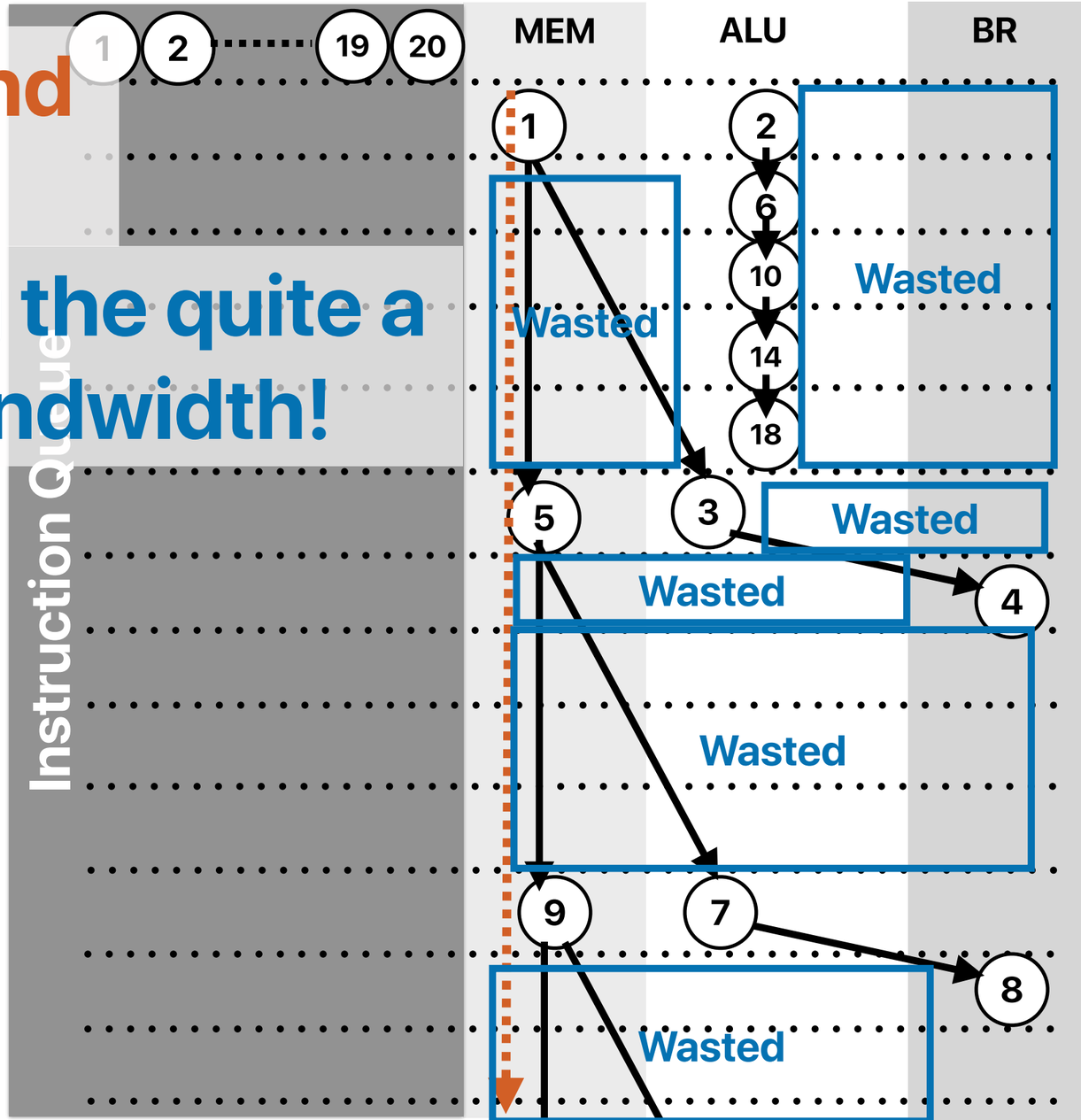
What if we have “unlimited” fetch/issue width — “linked list”

Even unlimited issue width and a perfect cache cannot help!

We’re wasting the quite a lot issue bandwidth!

```
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL );
```

```
① .L3:    movq    8(%rdi), %rdi
②         addl    $1, %eax
③         testq   %rdi, %rdi
④         jne     .L3
```



Summary of popcounts

	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%

Best at 4.5

Best performing one at 2.7

Outline

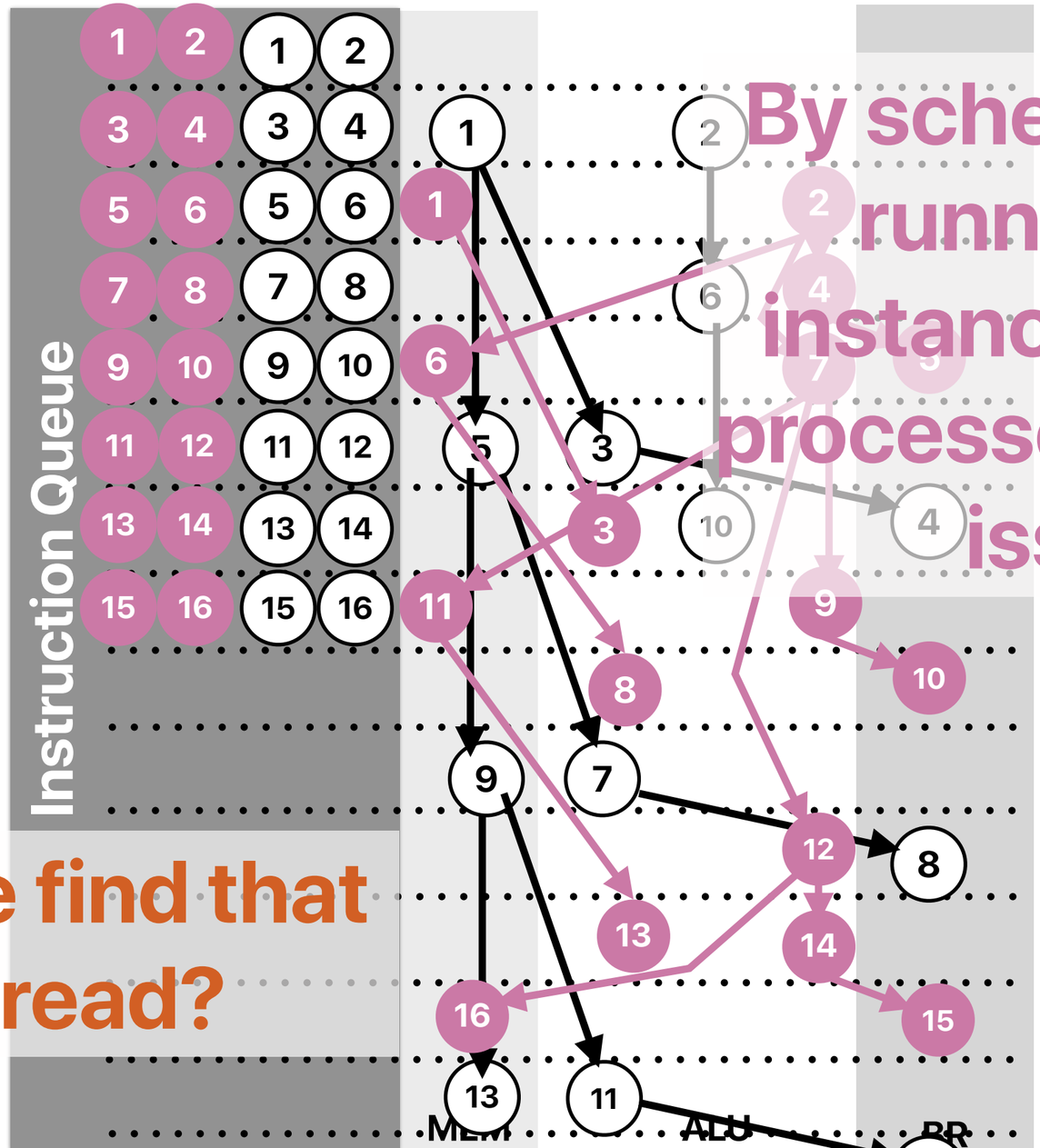
- Parallel architectures
  - Simultaneous multithreading (SMT)
  - Chip multiprocessors (CMP)
- Parallel programming

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
```




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

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

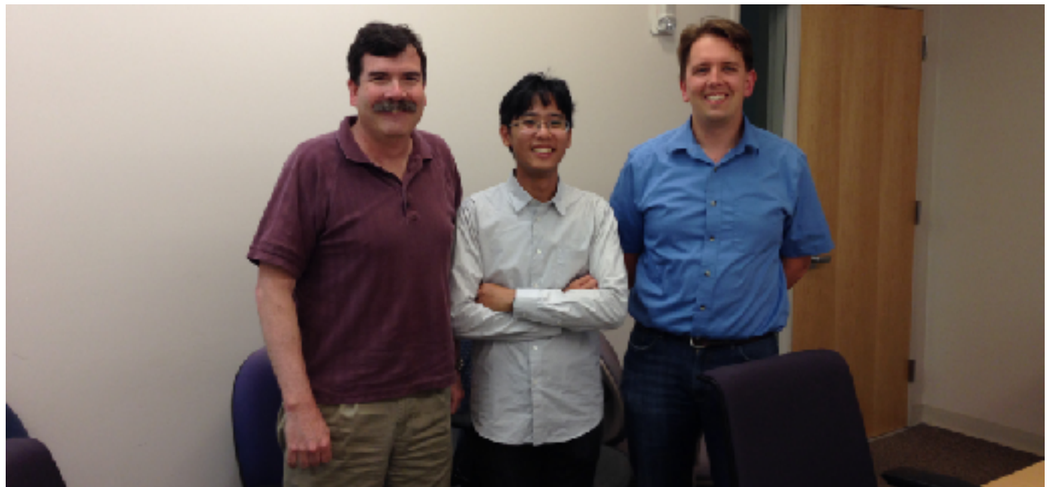
Where can we find that "other" thread?

Where to find another “thread”

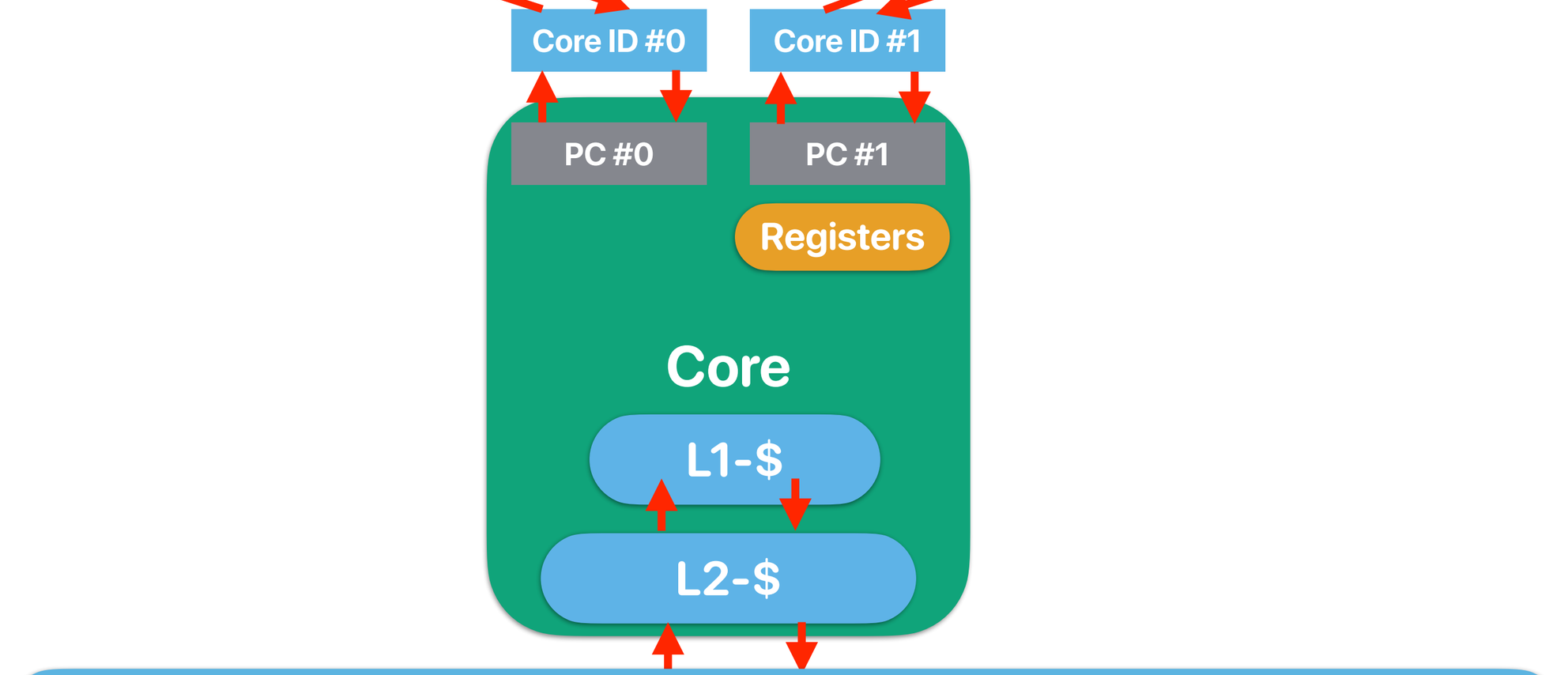
- Another process/running program forked from a completely different program
- Another process forked/cloned from the current process
  - The forked program cloned the memory content at the forked time
  - The forked program cannot view the memory space of the original process
  - The forked program 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**)



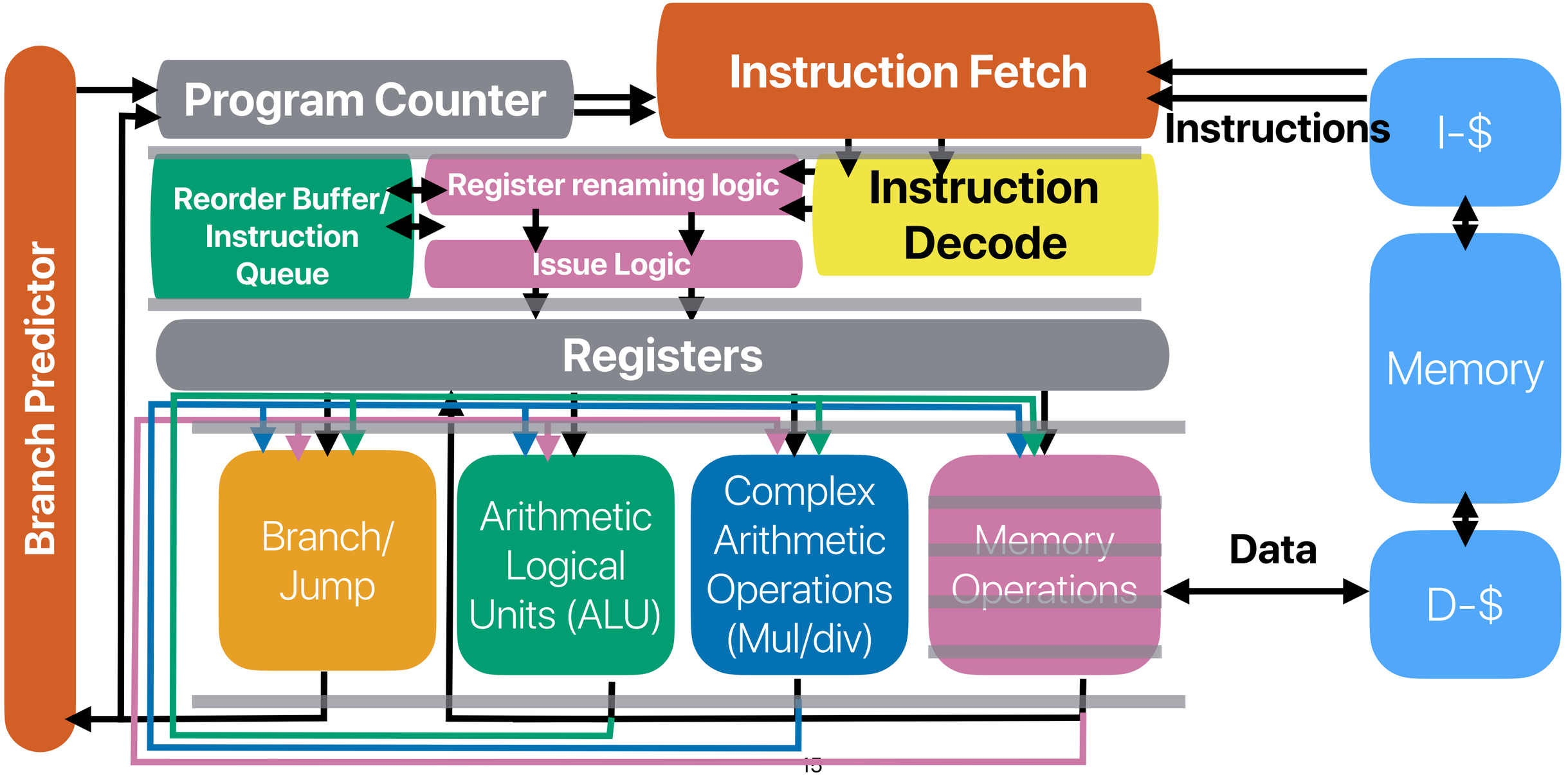
SMT from the user/OS' perspective



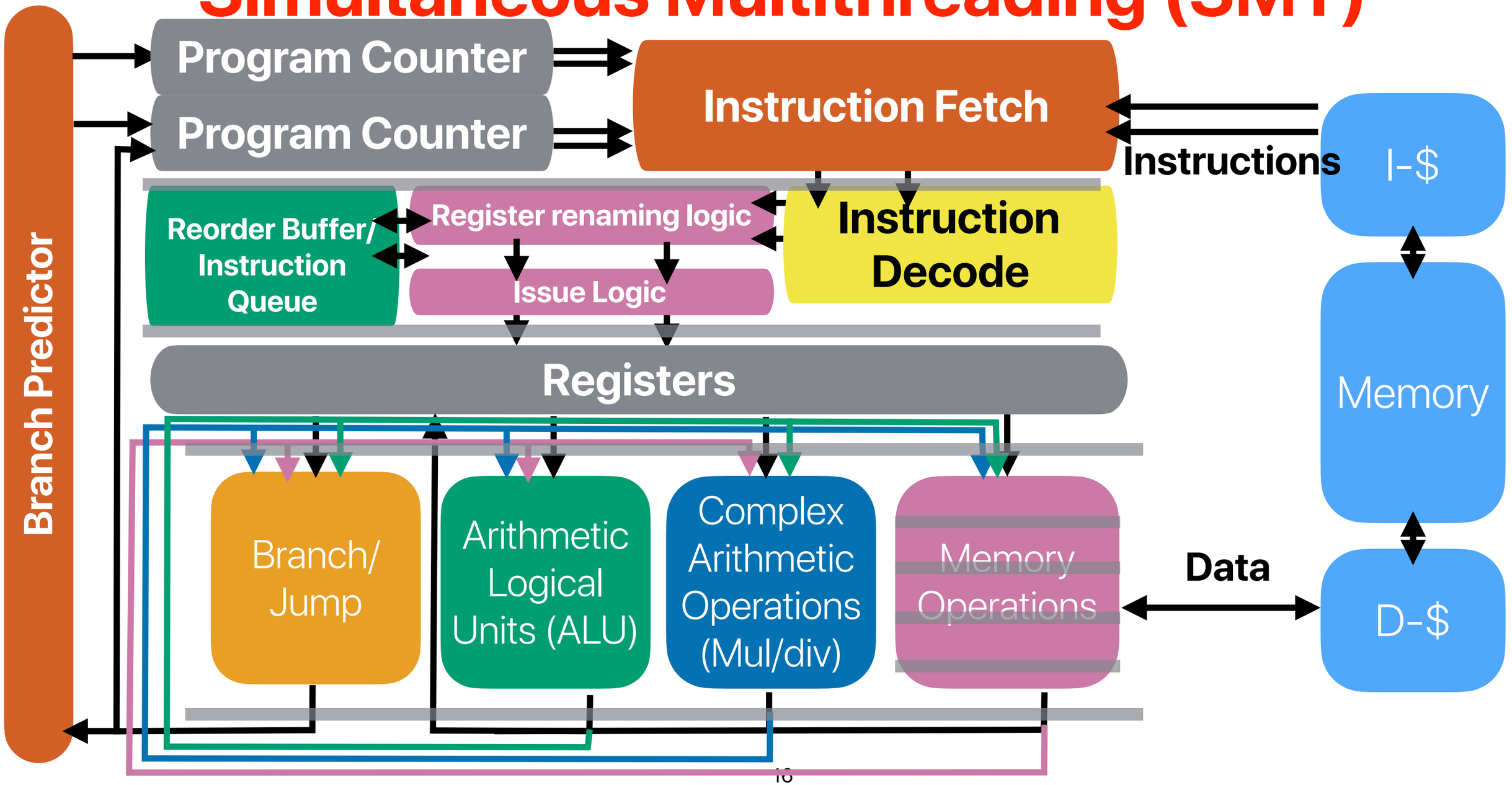
How do we support two running programs in one pipeline?

- 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)

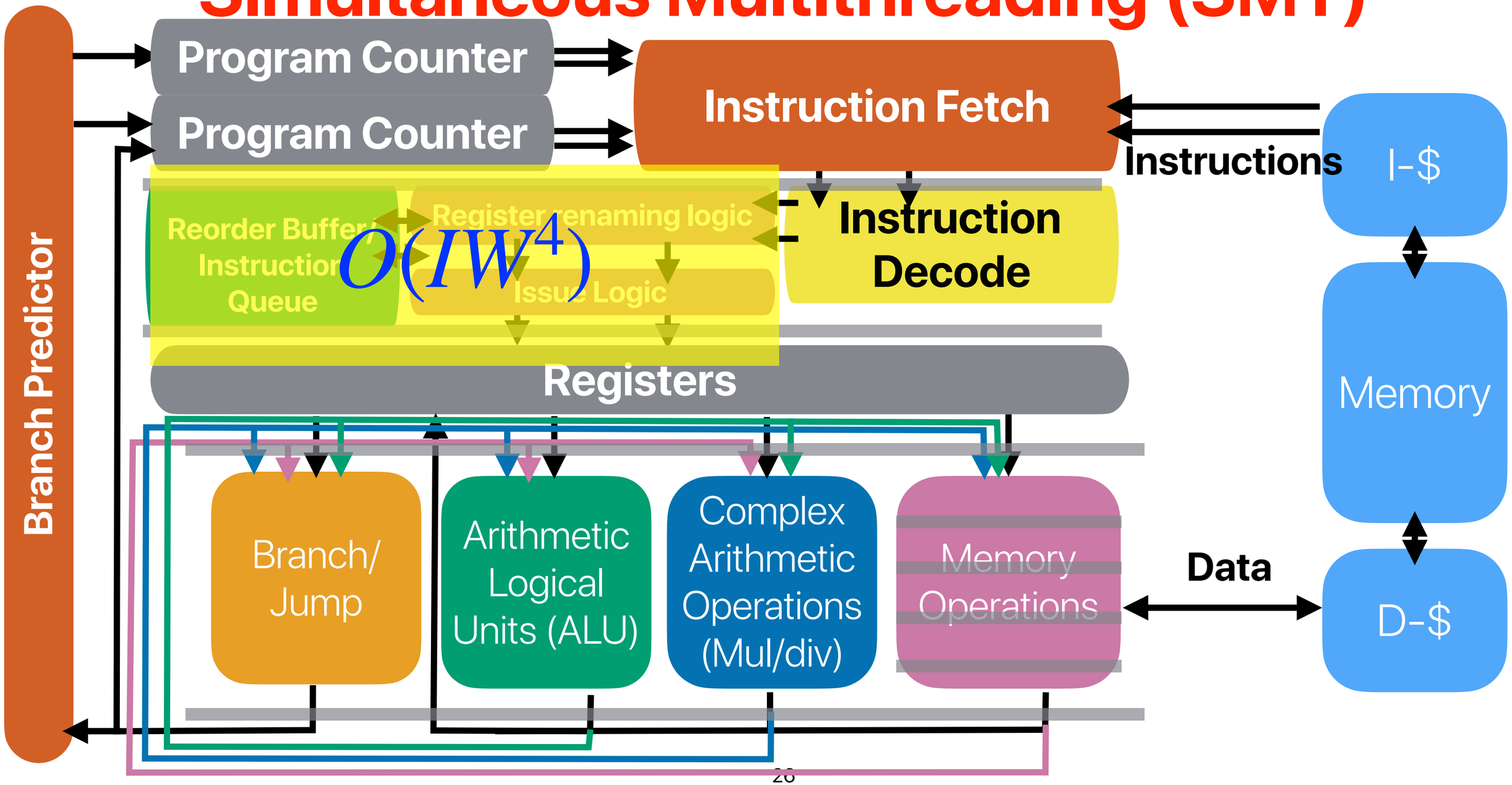


```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              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)



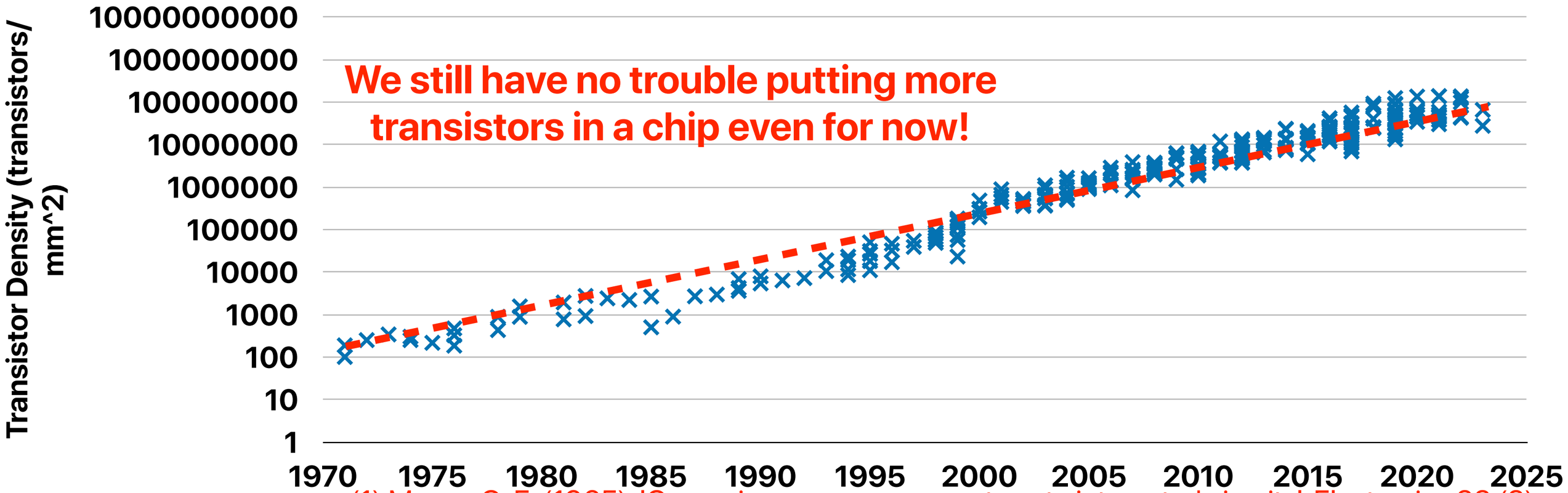
Takeaways: parallel architectures

- 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

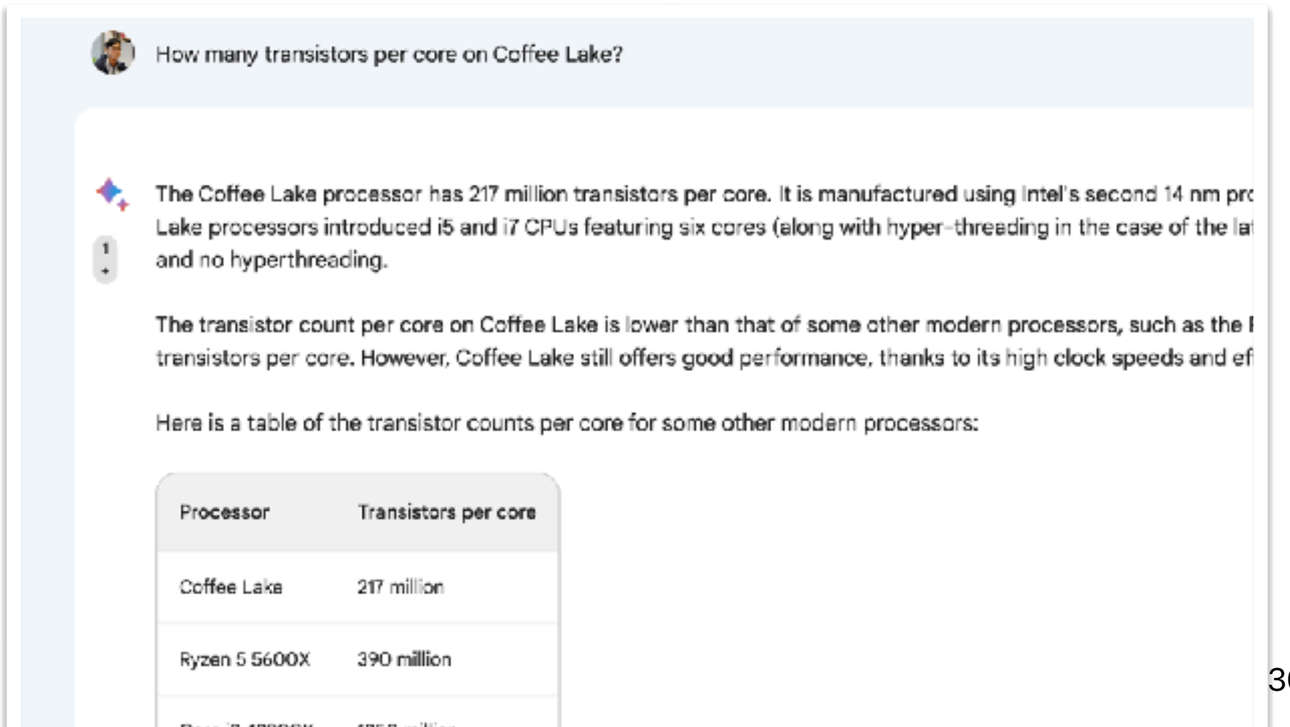


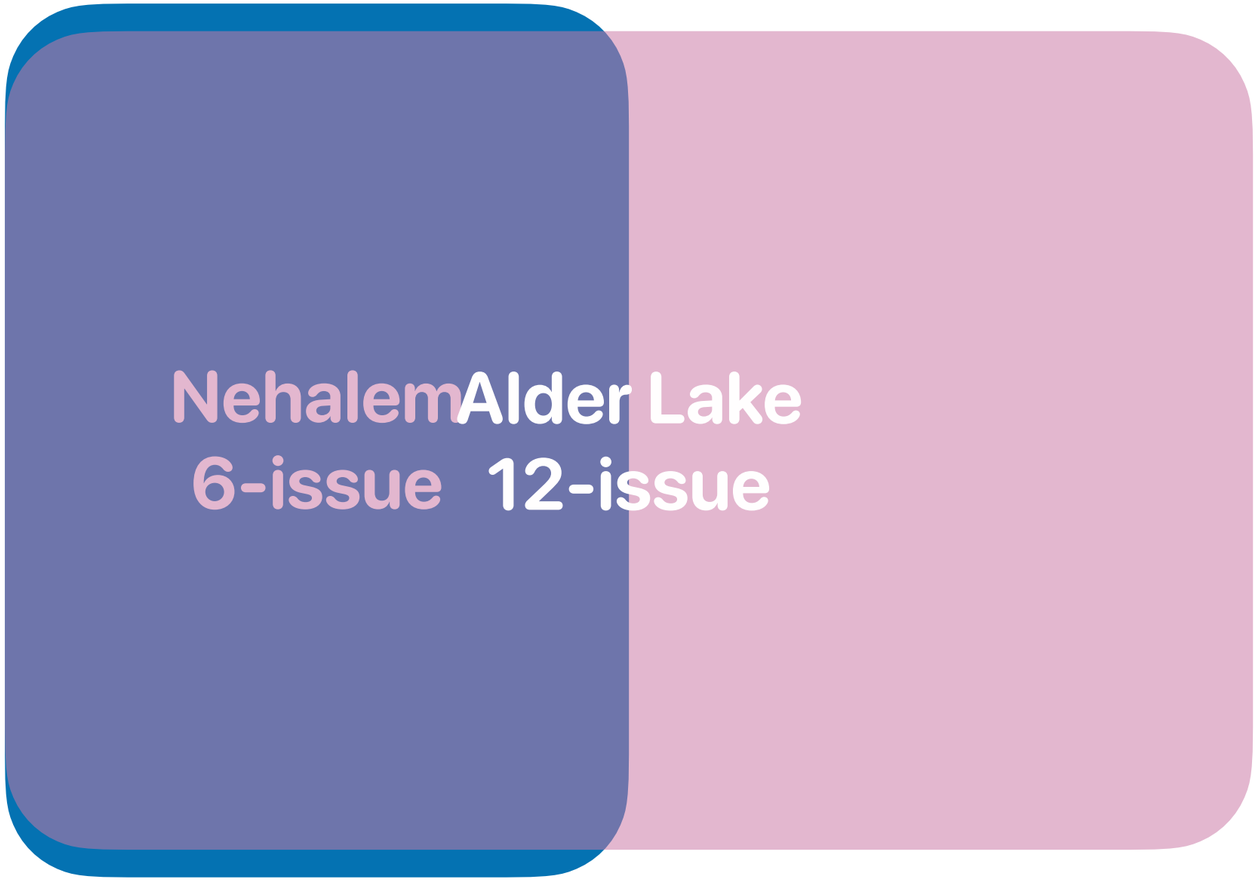
(1) Moore, G. E. (1965), 'Cramming more components onto integrated circuits', Electronics 38 (8) .

Plot based on https://en.wikipedia.org/wiki/Transistor_count by Hung-Wei Tseng

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





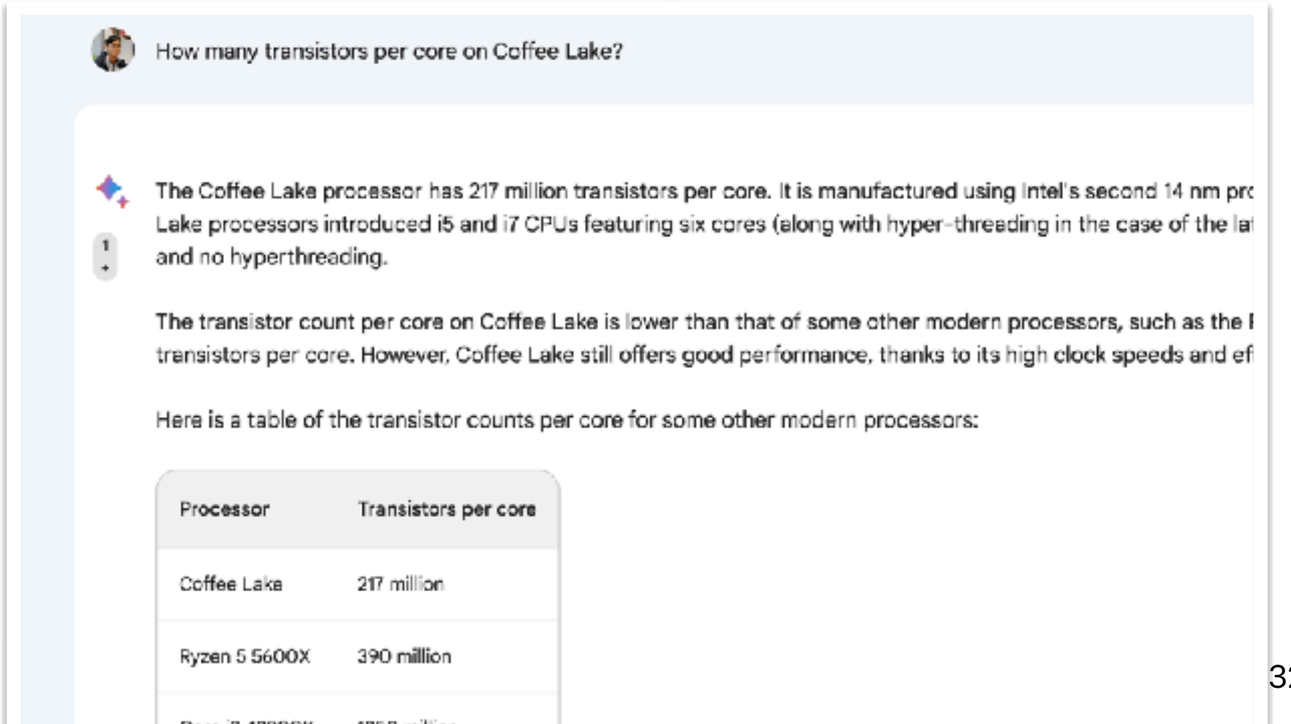
Based on https://en.wikipedia.org/wiki/Transistor_count

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



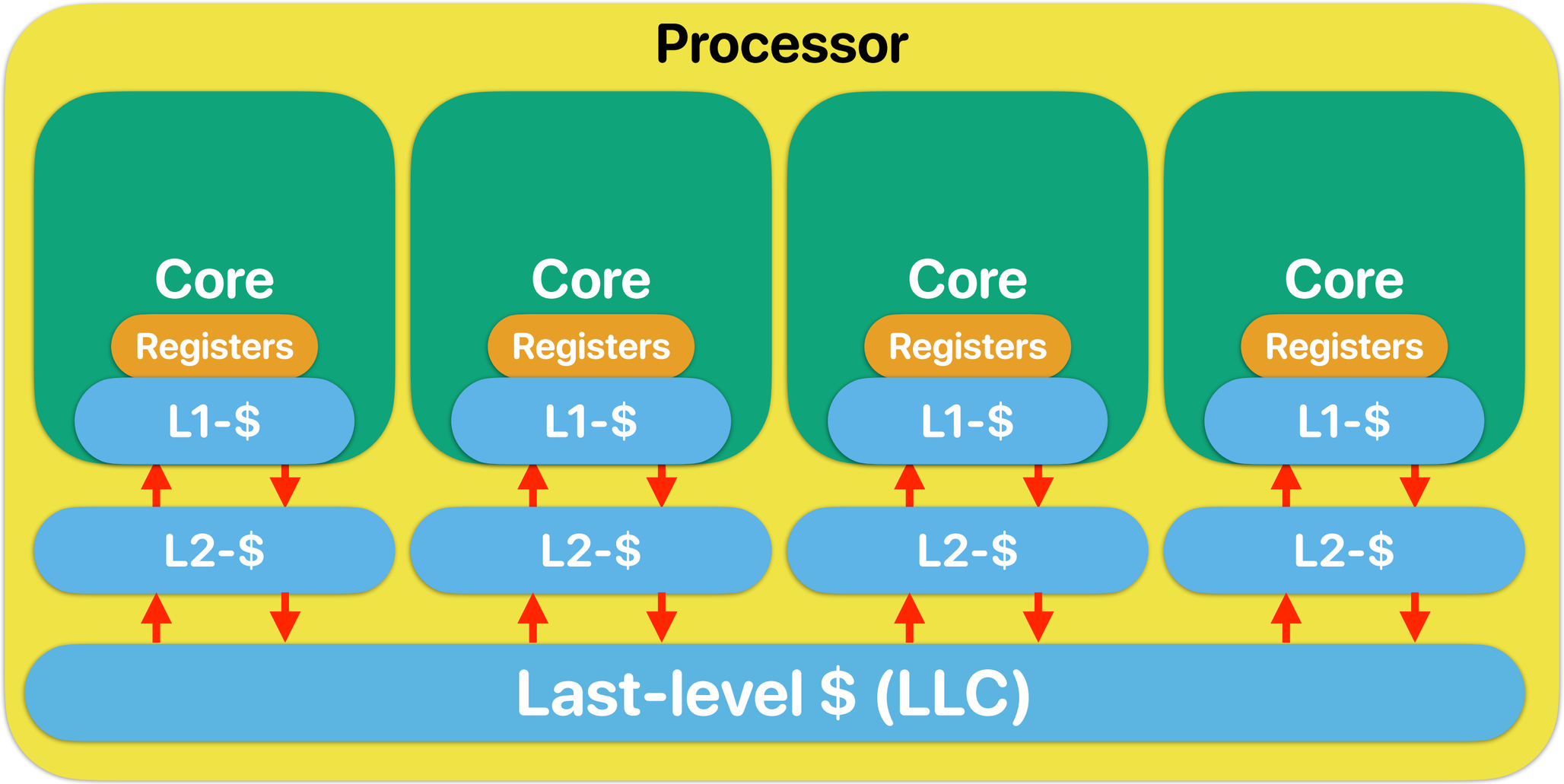
2x 3-issue ALUs Nehalem

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

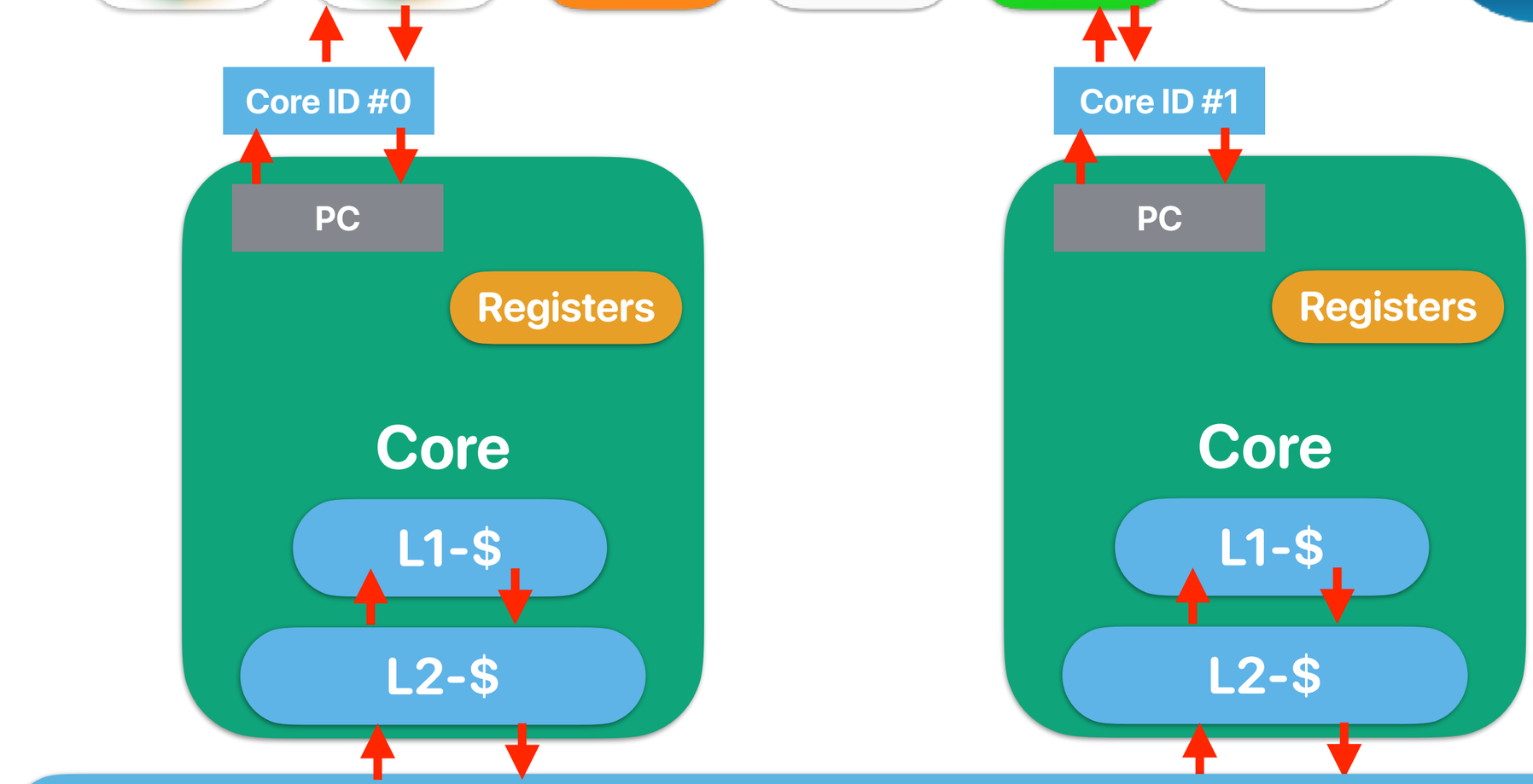
1x 5-issue ALUs Alder Lake

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

Concept of CMP

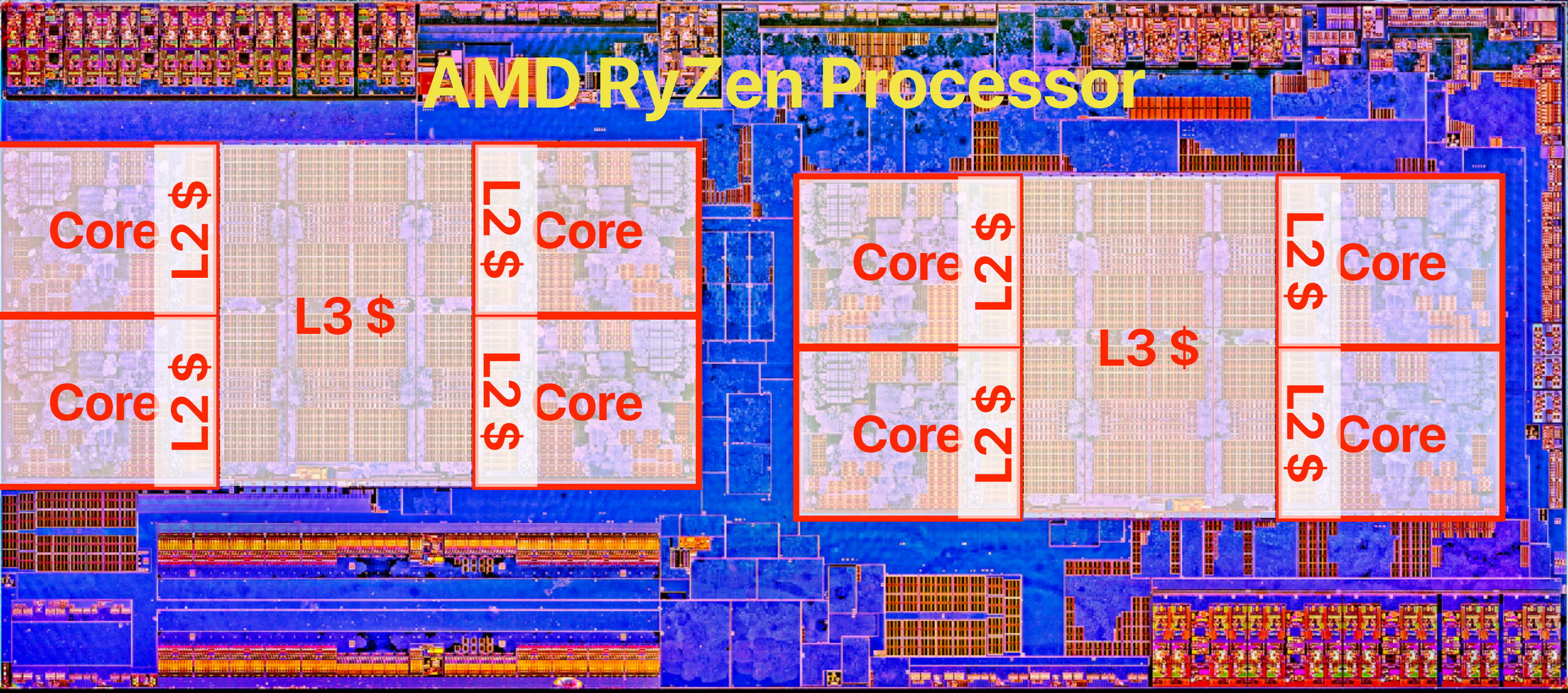


CMP from the user/OS' perspective

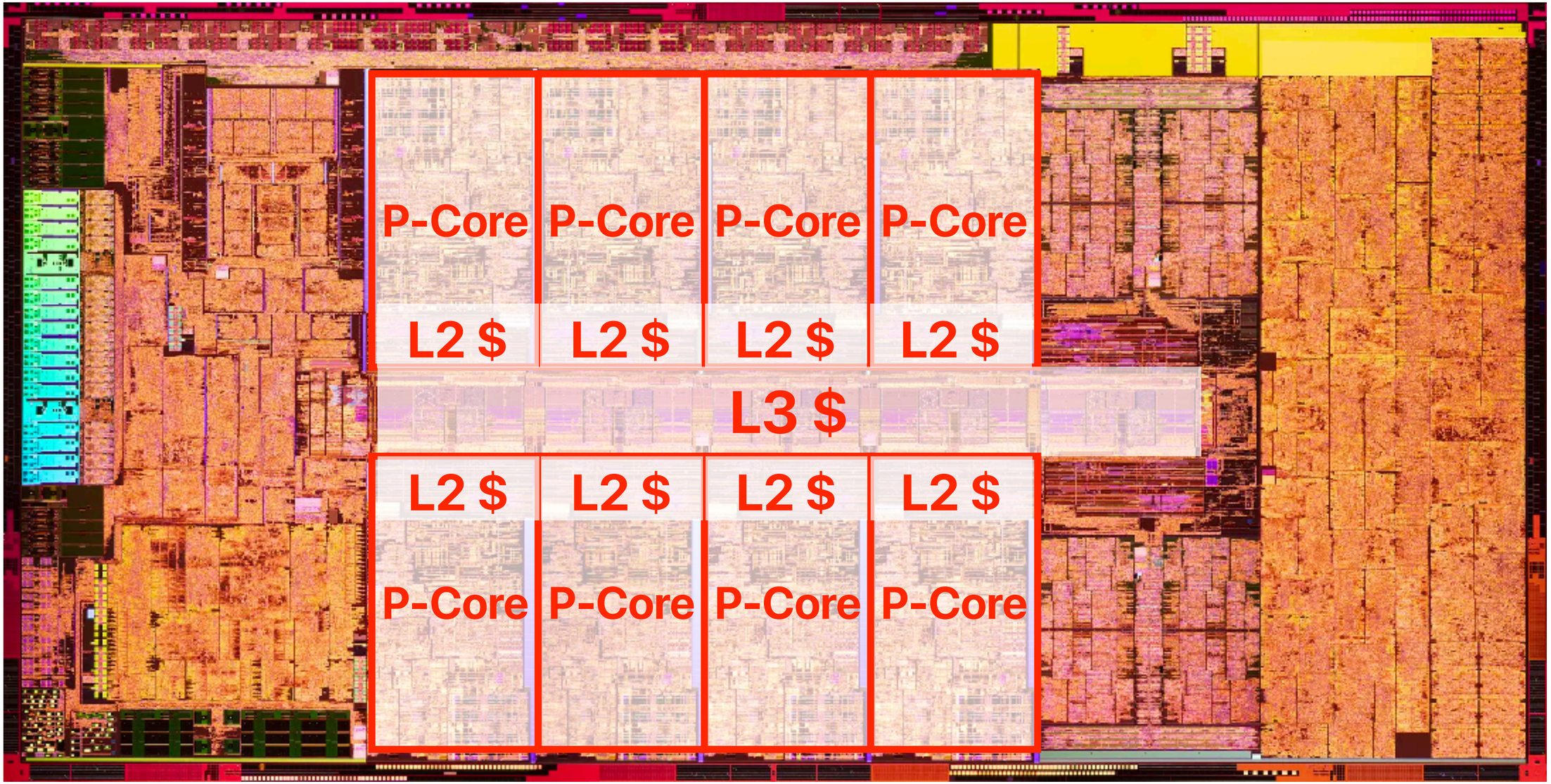


```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              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
```

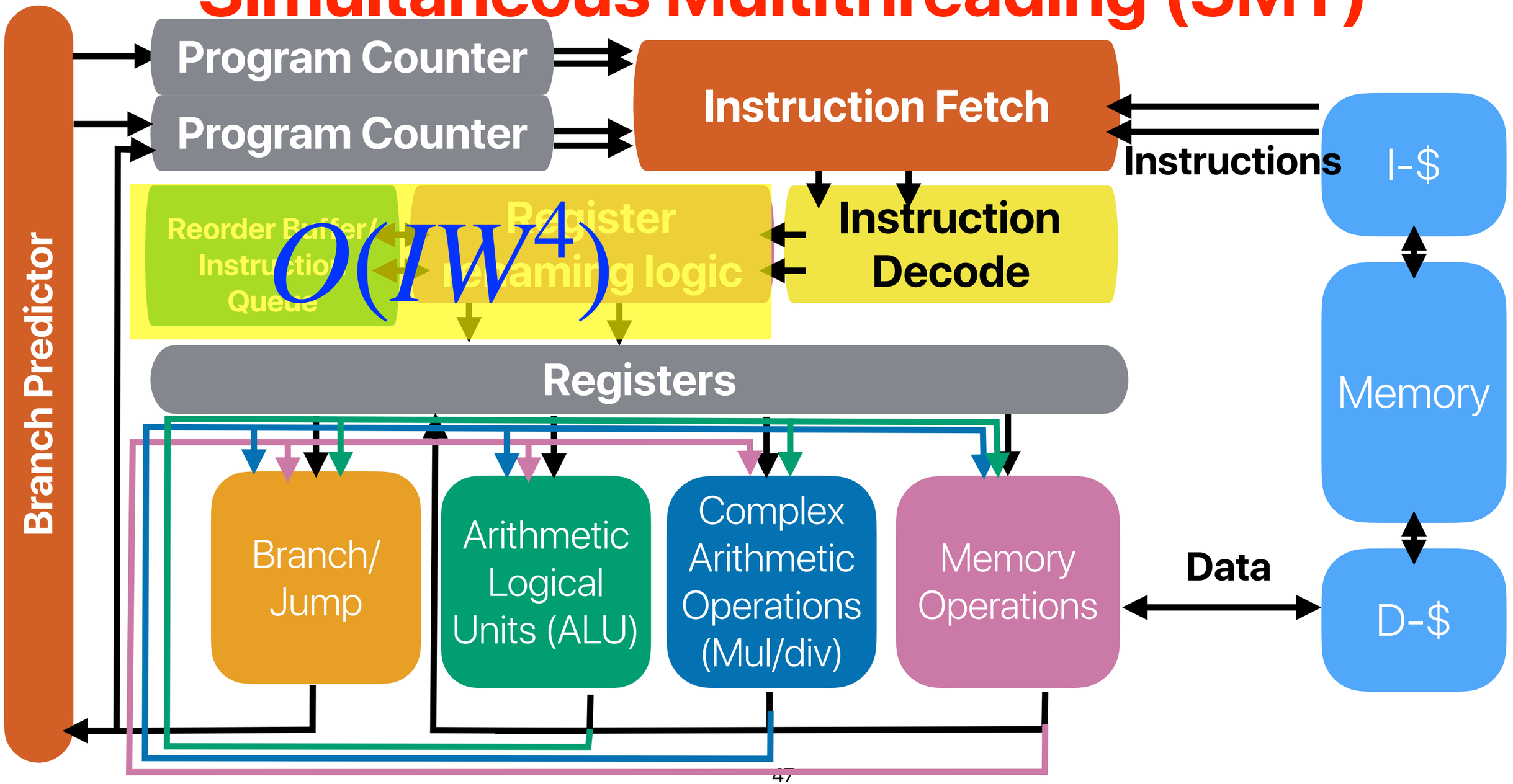
Modern processors have both CMP/SMT


AMD RyZen Processor
Core
L2 $
L3 $
Core
L2 $
Core
L2 $
Core
L2 $
Core
L2 $
L3 $
Core
L2 $
Core
L2 $
AMD
RYZEN

Intel Alder Lake



Simultaneous Multithreading (SMT)



Takeaways: parallel architectures

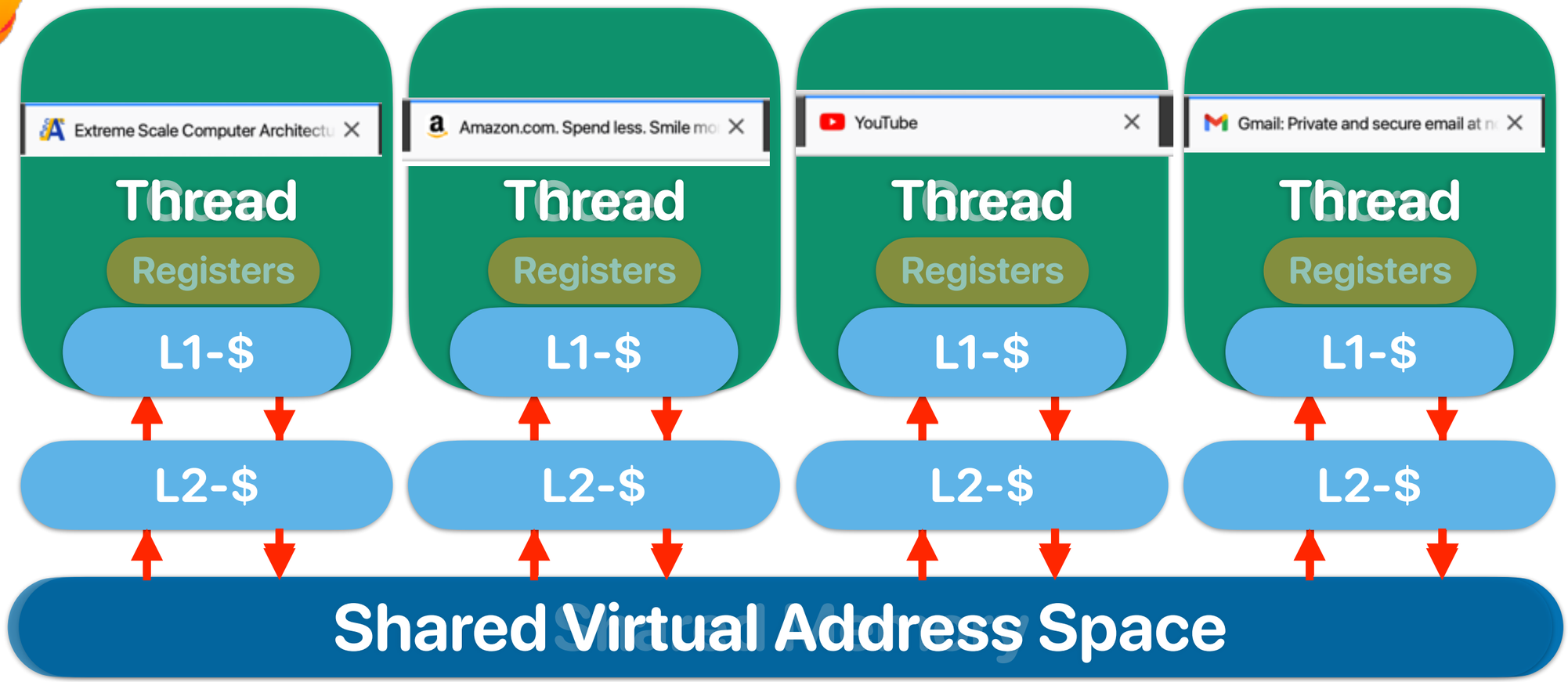
- SMT processors can better utilize the pipeline resources by allowing simultaneous execution of multiple threads
  - Improved execution throughput
  - May hurt the latency of each thread since we share functional units & cache
- CMP provides more processor cores for parallel threads or multiprogrammed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

Parallel Programming & Architectural Supports for Parallel Programming

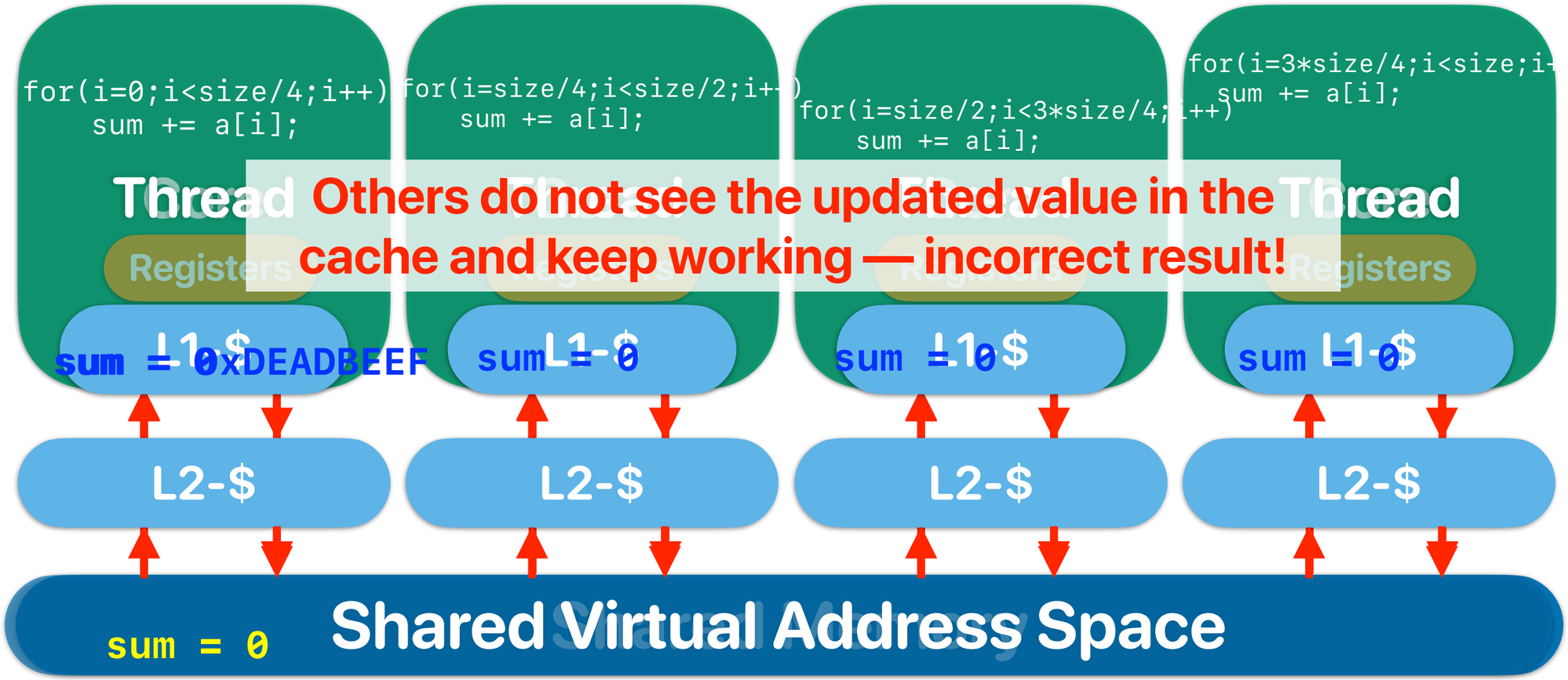
Parallel programming

- 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



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

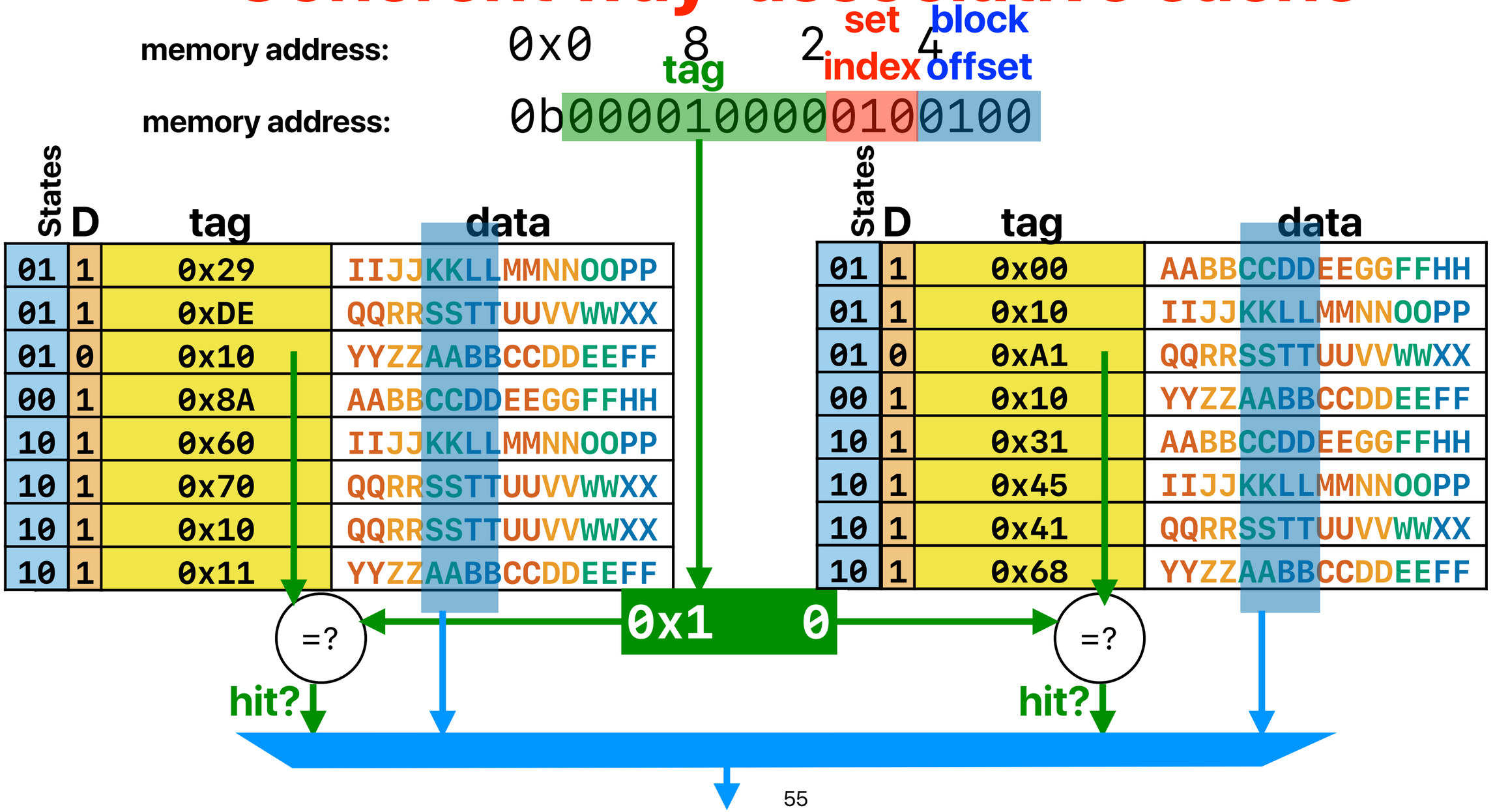
Coherency & Consistency

- 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

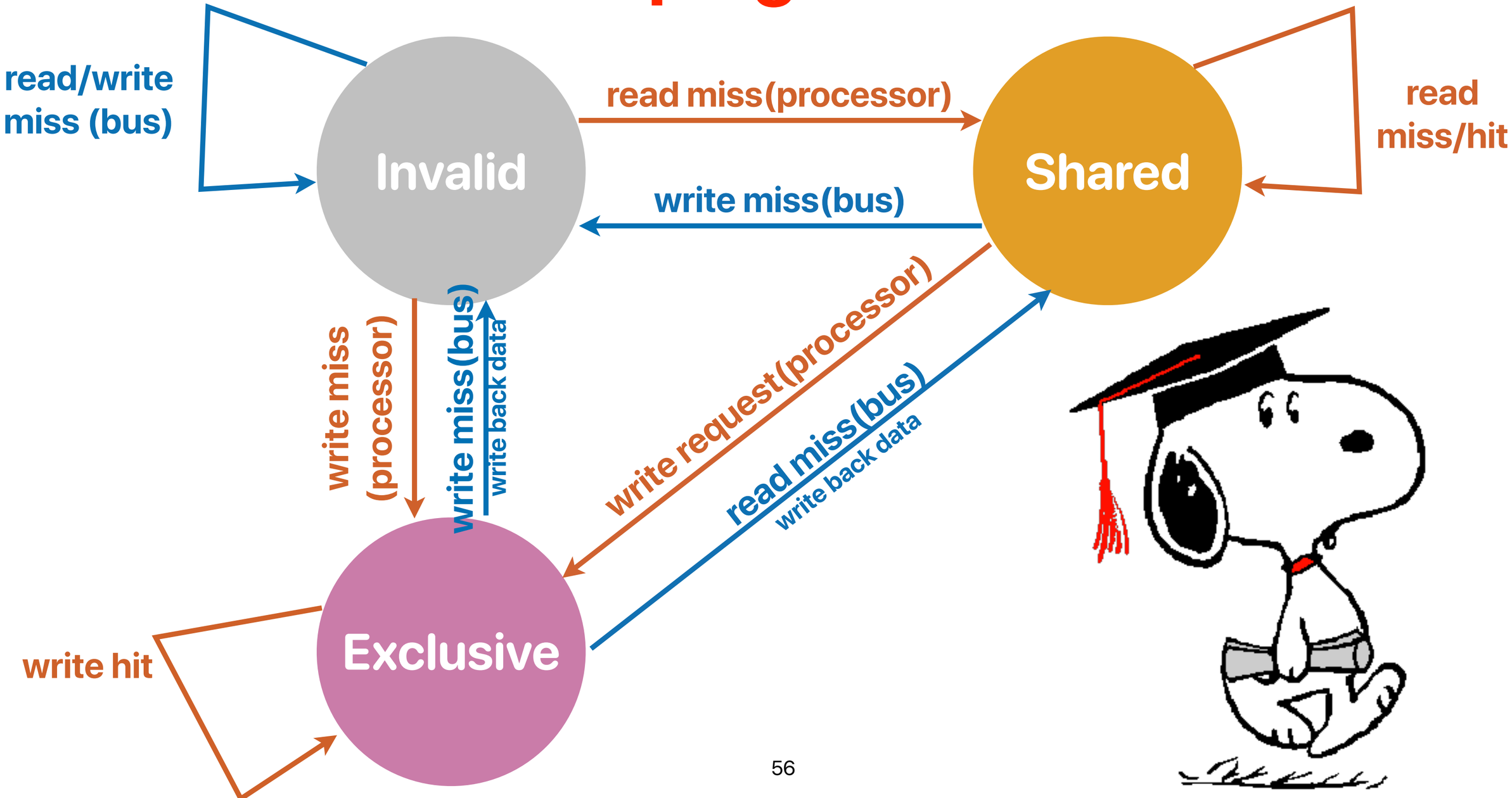
Simple cache coherency protocol

- 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

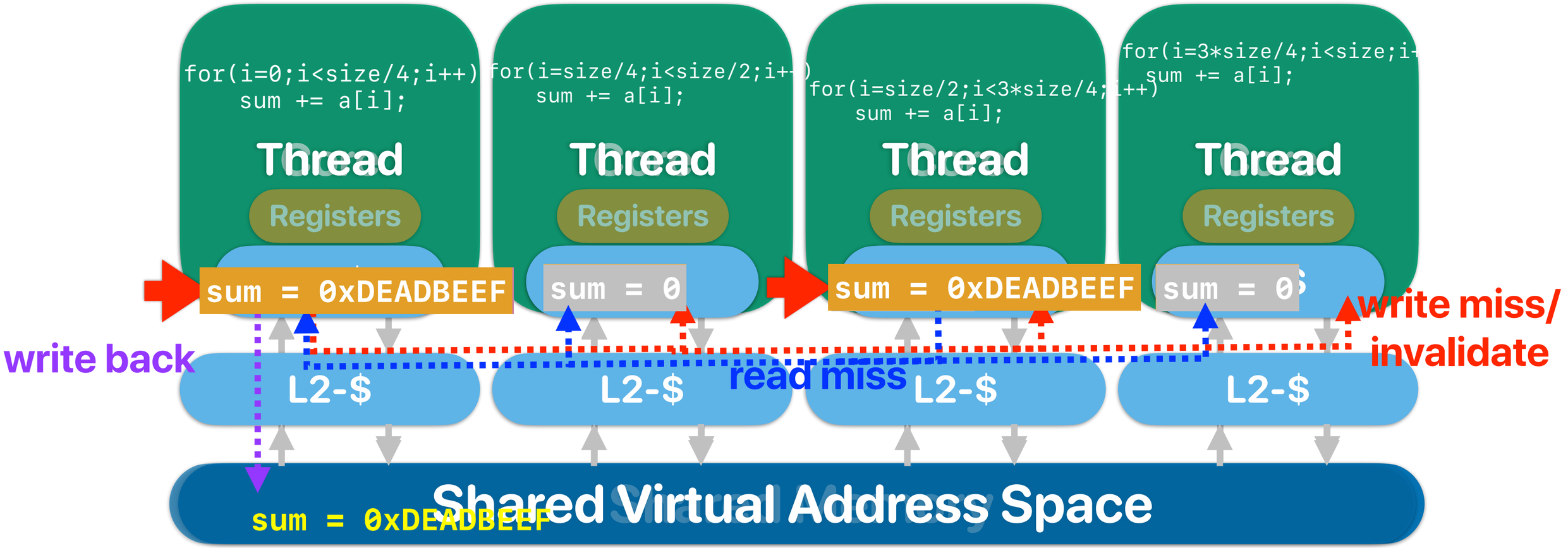
Coherent way-associative cache



Snooping Protocol



What happens when we write in coherent caches?



Observer

thread 1

```
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",
```

thread 2

```
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

```
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",
```

thread 2

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

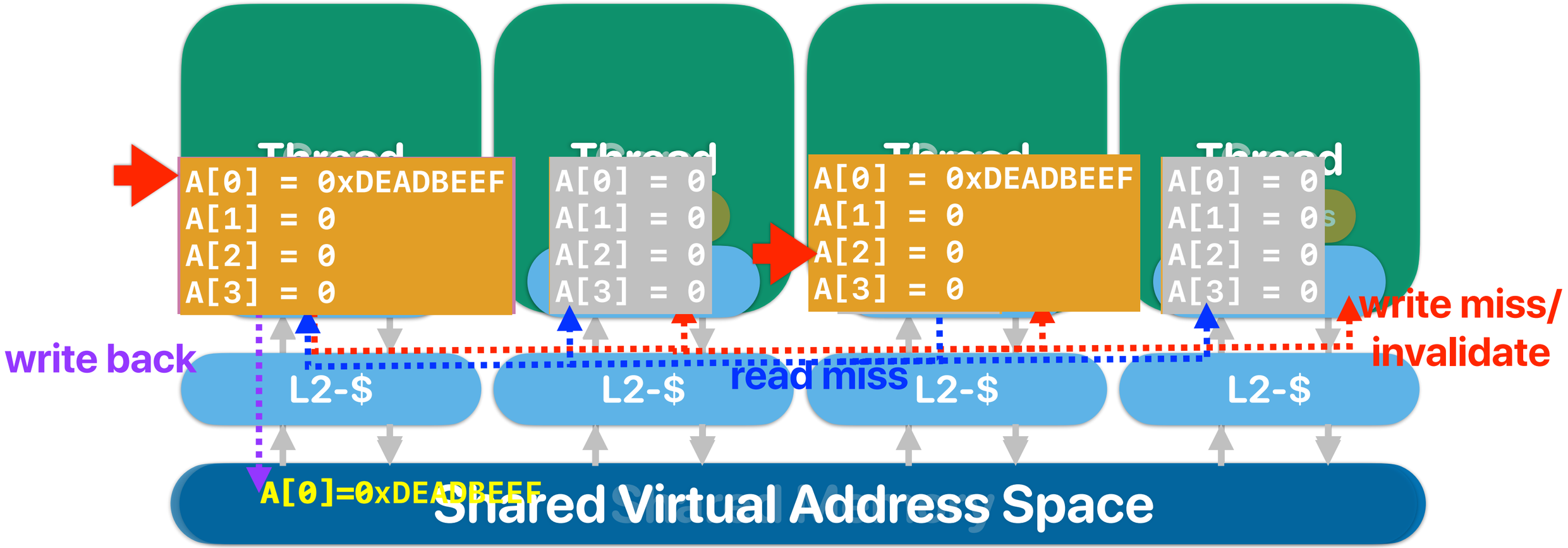
Takeaways: parallel architectures

- 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

Take-aways: parallel programming

- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when

Cache coherency



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;
}
```

C 

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;
}
```

C 

4Cs of cache misses

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

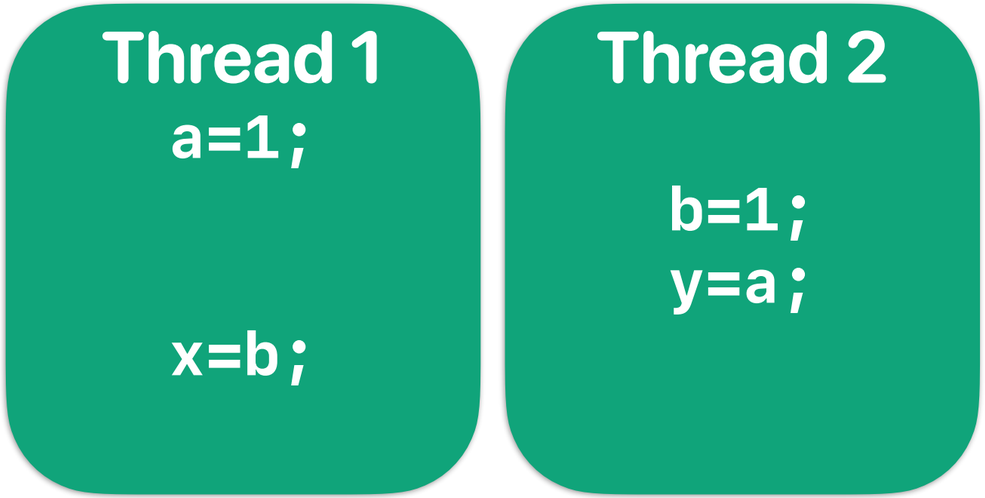
False sharing

- 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!

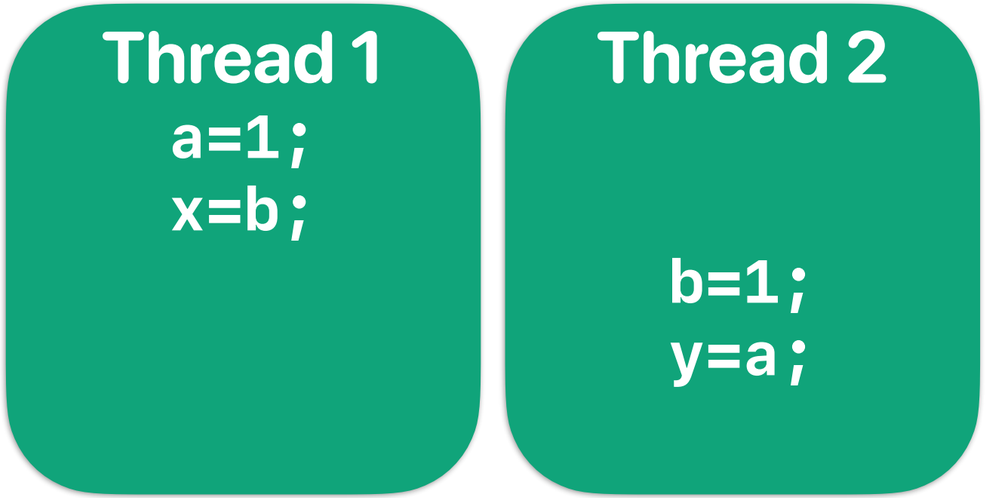
Take-aways: parallel programming

- 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

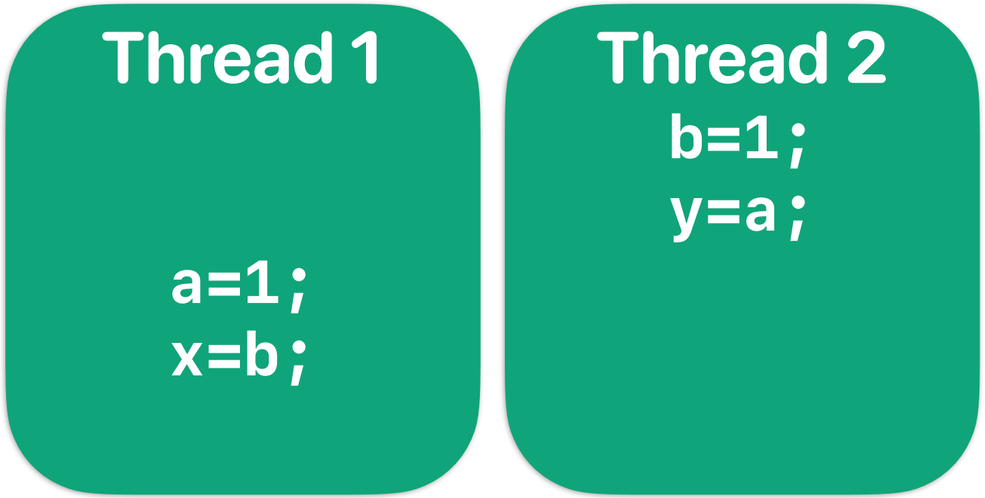
Possible scenarios



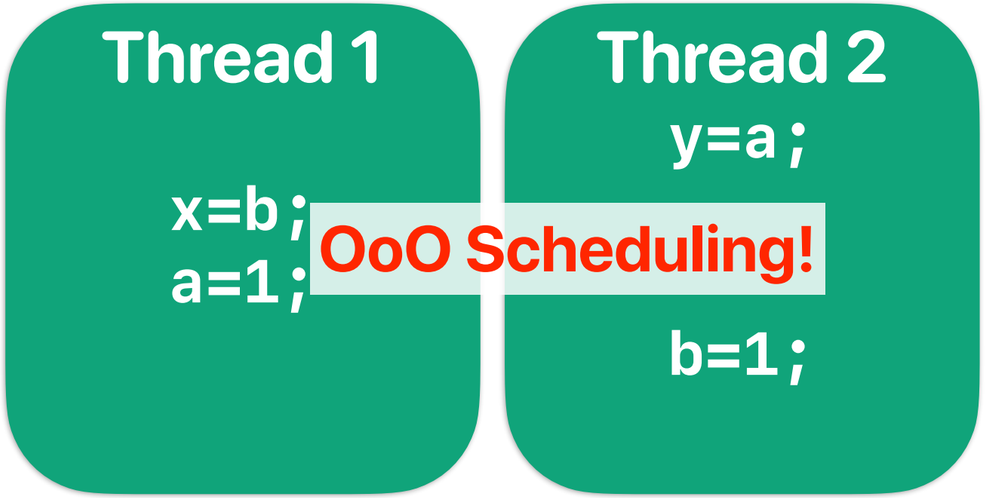
(1,1)



(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

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

thread 1	thread 2

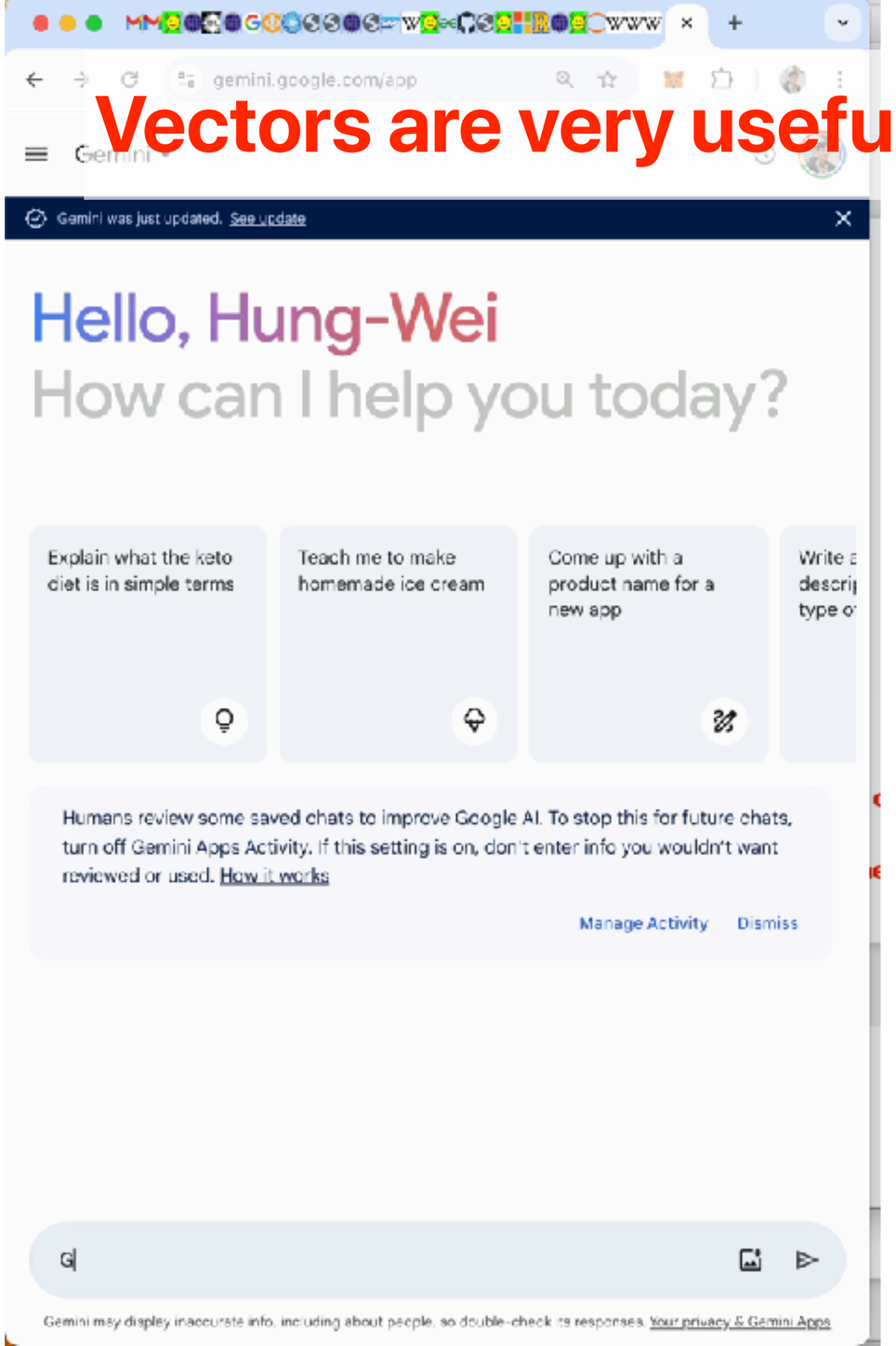
`a=1;`
`mfence` **a=1 must occur/update before mfence**
`x=b;`
	`b=1;`
	`mfence` **b=1 must occur/update before mfence**
	`y=a;`

Take-aways: 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”

Vectors are very useful data/mathematical representations

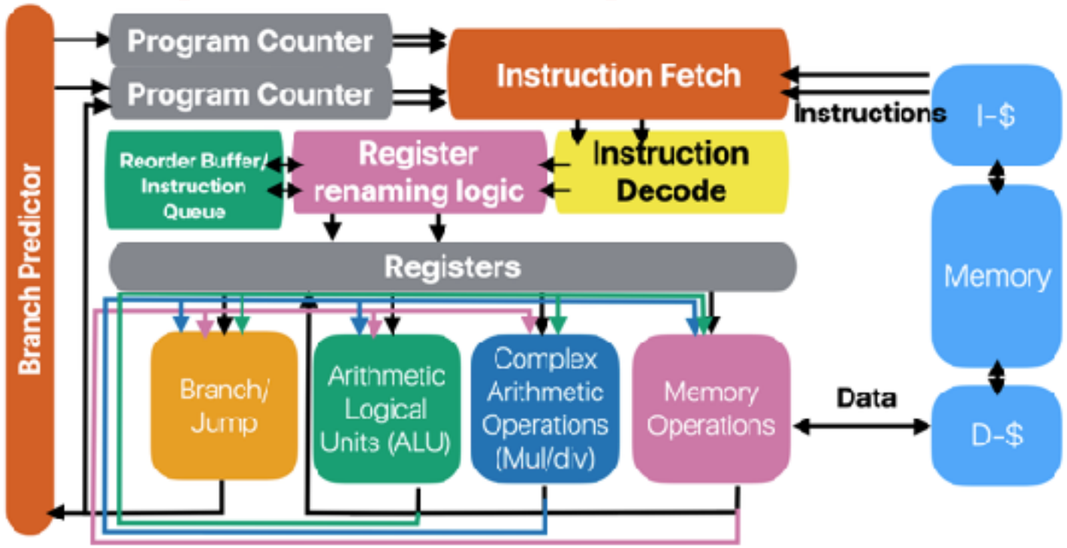


- But how do we process them using "superscalar" processors?

```
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][0]
load vector[1]
```

```
load matrix[0][1]
load vector[2]
load matrix[0][2]

mul matrix[0][0], vector[0]
```

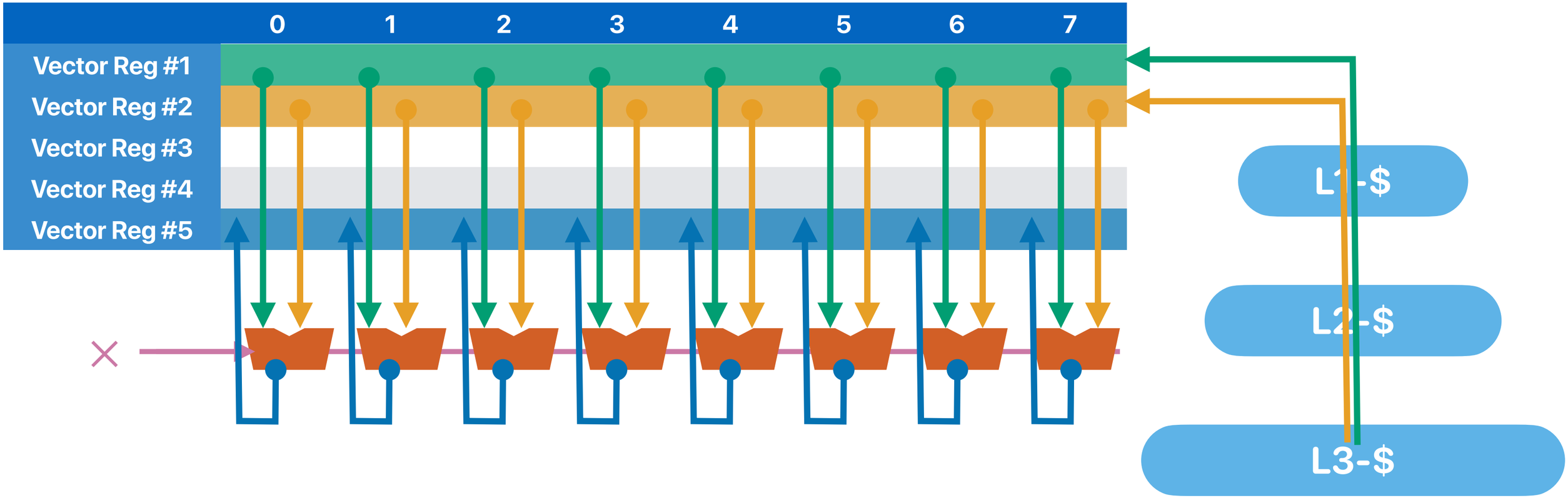
```
load vector[3]
load matrix[0][3]
load vector[4]

mul matrix[0][1], vector[1]
mul matrix[0][2], vector[2]
```

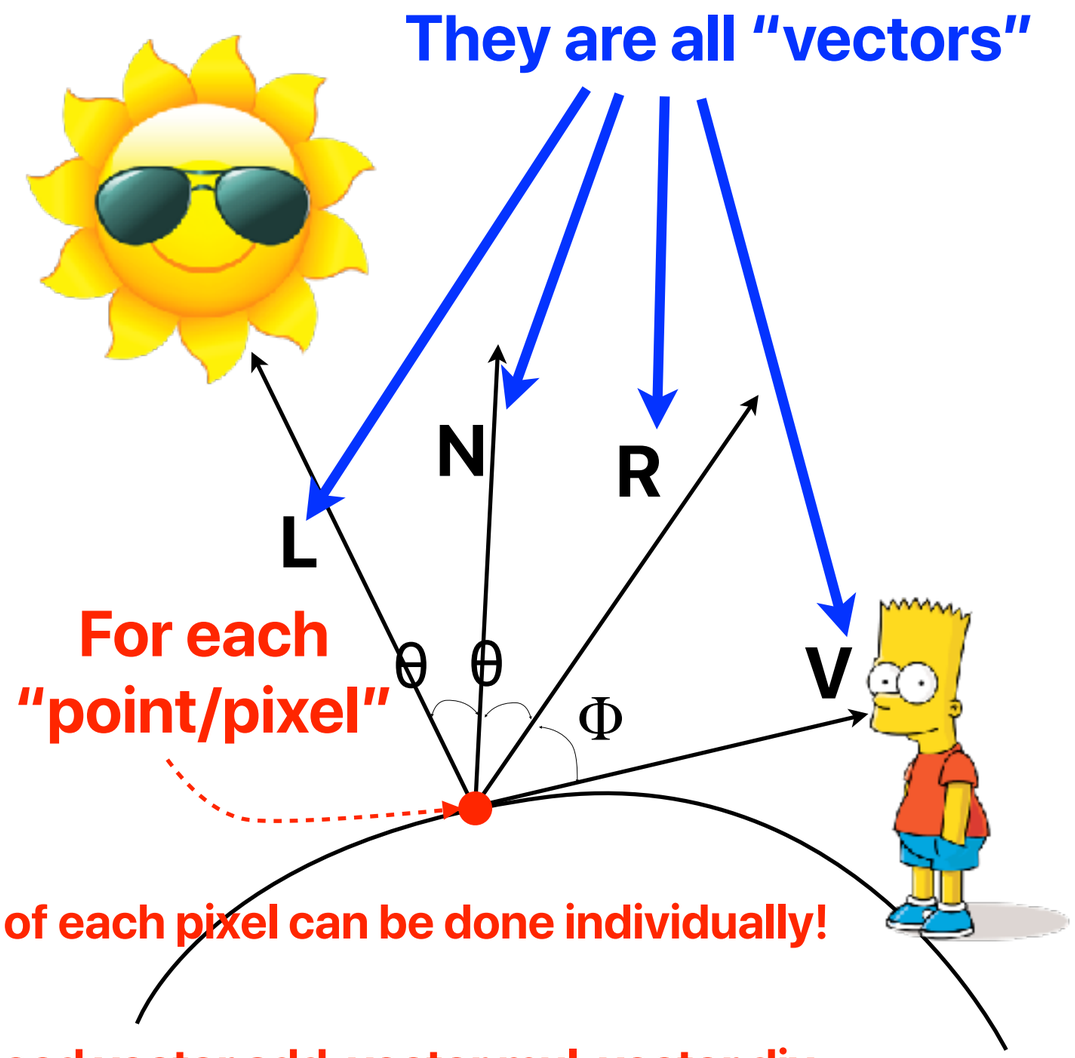
```
load matrix[0][4]
load vector[5]
load matrix[0][5]

mul matrix[0][3], vector[3]
```

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;
}
```

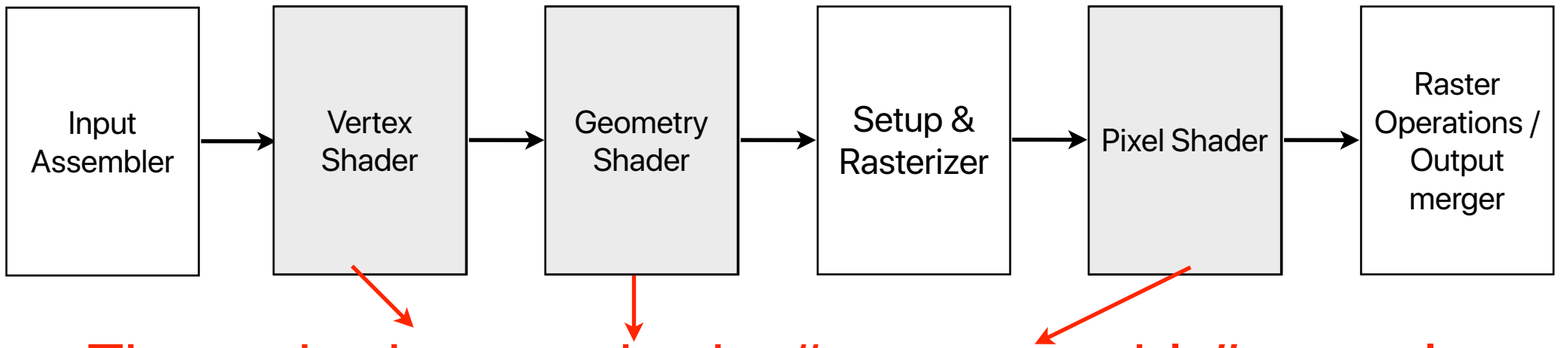
The work of each pixel can be done individually!

We only need vector add, vector mul, vector div.

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!!!

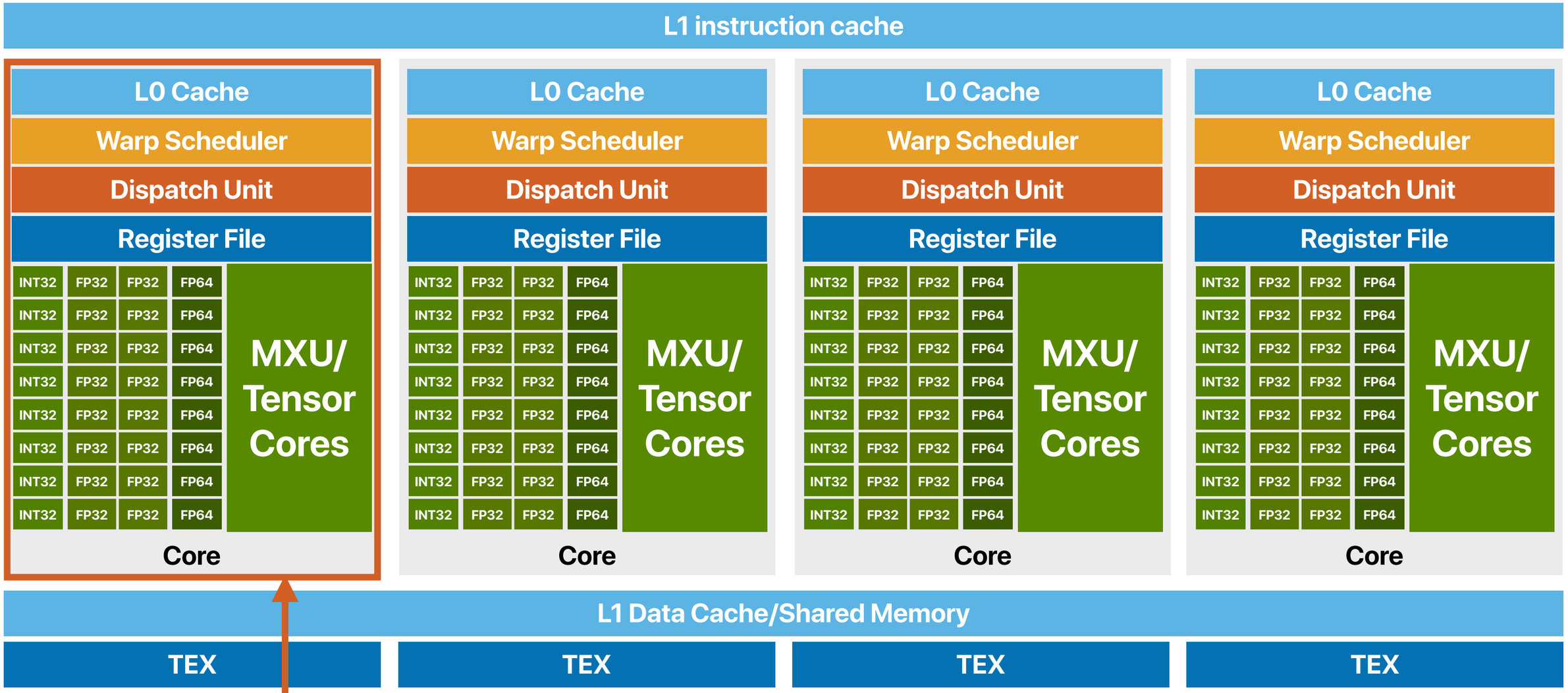


These shaders need to be "programmable" to apply different rendering effects/algorithms (Phong shading, Gouraud shading, and etc...)

What'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

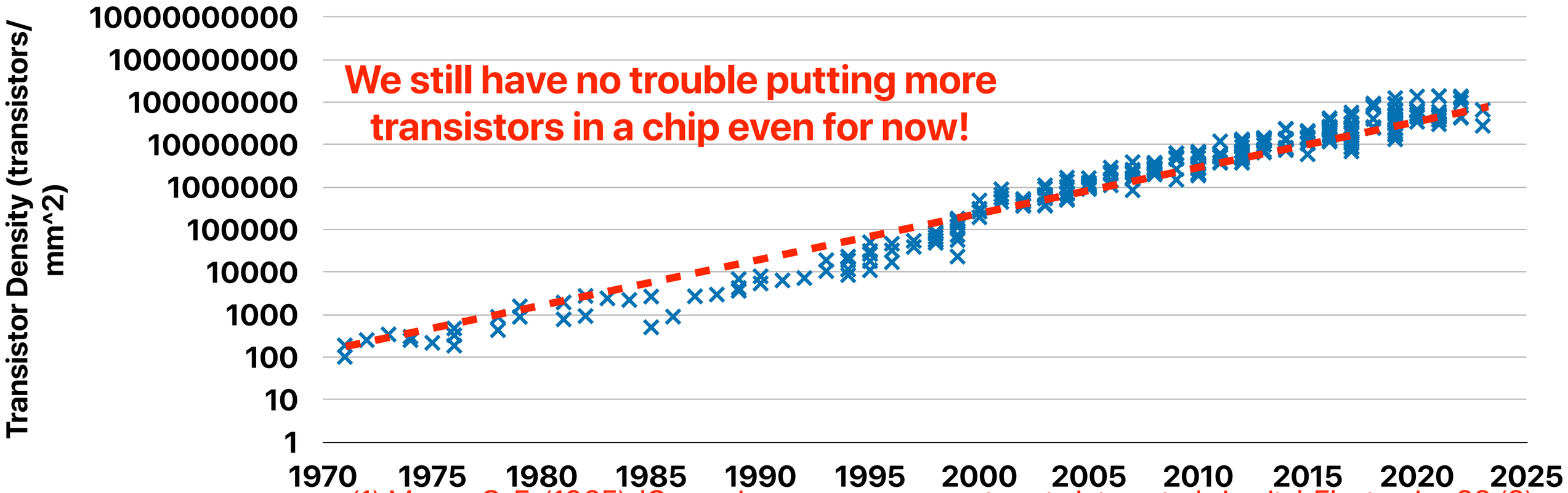
GPU Architecture



The limiting factor of modern processor performance

Moore's Law[1]

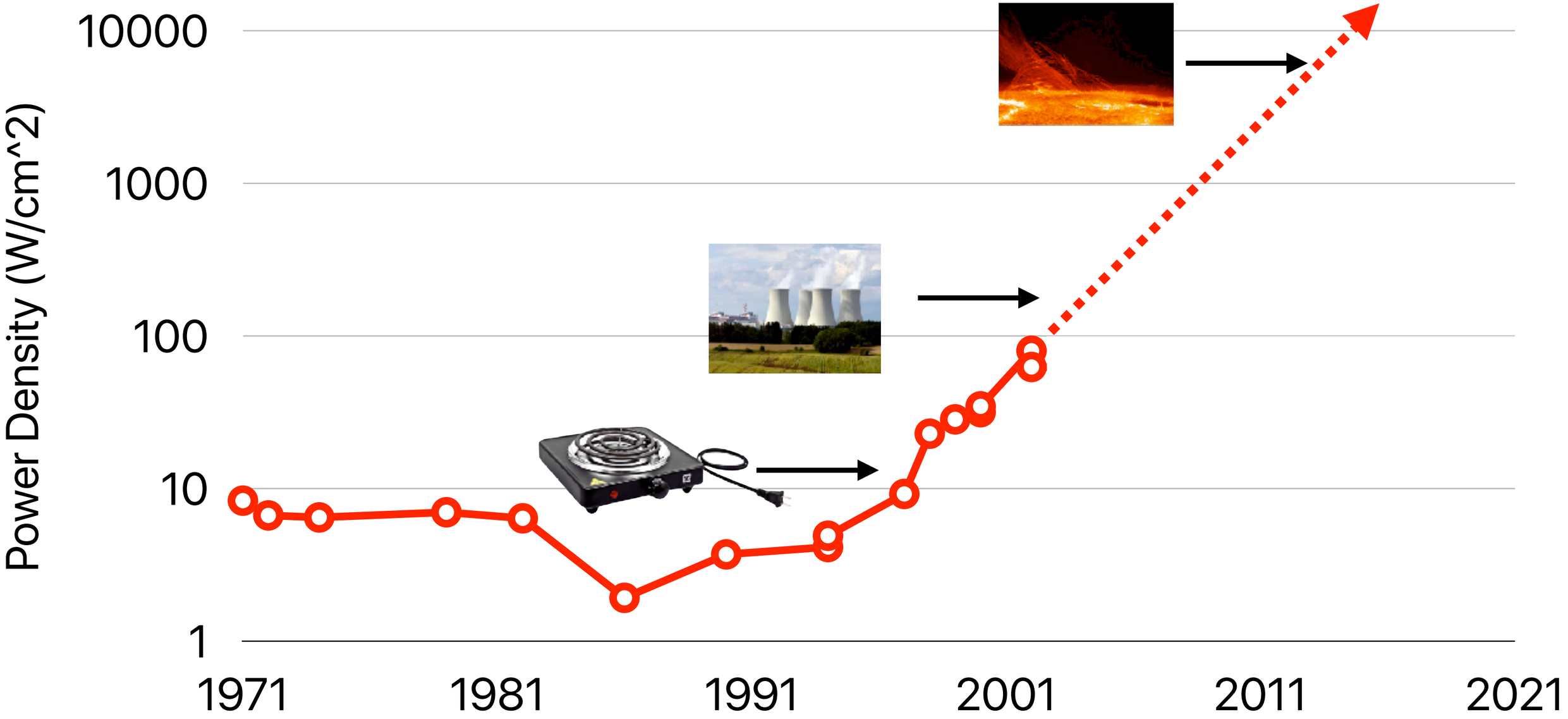
- The number of transistors we can build in a fixed area of silicon doubles every 12 ~ 24 months.
- Moore's Law "was" the most important driver for historic CPU performance gains



(1) Moore, G. E. (1965), 'Cramming more components onto integrated circuits', Electronics 38 (8) .

Plot based on https://en.wikipedia.org/wiki/Transistor_count by Hung-Wei Tseng

Power Density of Processors



Power consumption

Power v.s. Energy

- Power is the direct contributor of “heat”
  - Packaging of the chip
  - Heat dissipation cost
- Energy = P * ET
  - The electricity bill and battery life is related to energy!
  - Lower power does not necessary means better battery life if the processor slow down the application too much

Dynamic/Active Power

- The power consumption due to the switching of transistor states
- Dynamic power per transistor

  $P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$

  - α: average switches per cycle
  - C: capacitance
  - V: voltage
  - f: frequency, usually linear with V
  - N: the number of transistors

Static/Leakage Power

- The power consumption due to leakage — transistors do not turn all the way off during no operation
- Becomes the **dominant** factor in the most advanced process technologies.

$$P_{leakage} \sim N \times V \times e^{-V_t}$$

  - N: number of transistors
  - V: voltage
  - V_t: threshold voltage where transistor conducts (begins to switch)

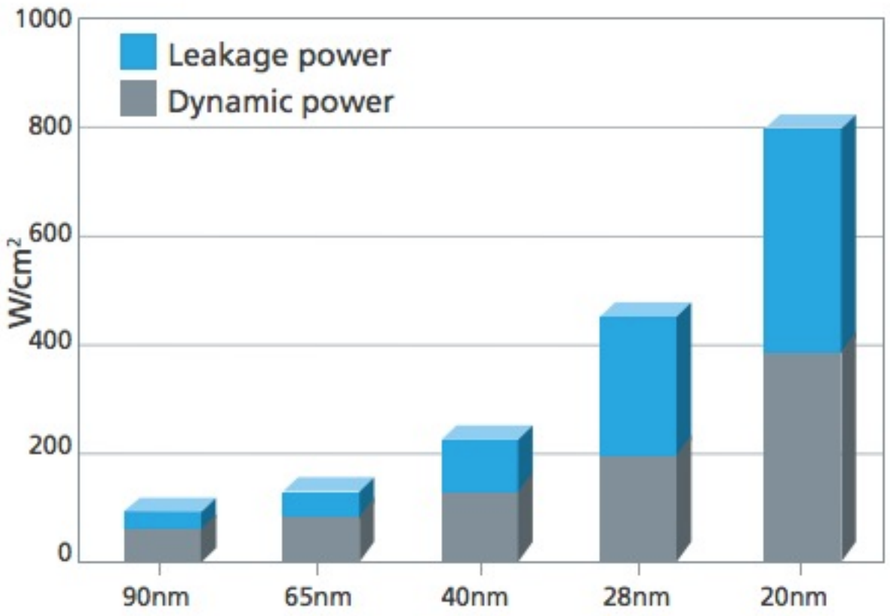


Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).

Dynamic/Active Power

- The power consumption due to the switching of transistor states
- Dynamic power per transistor

  $P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$

  - α: average switches per cycle
  - C: capacitance
  - V: voltage
  - f: frequency, usually linear with V
  - N: the number of transistors

Demo — changing the max frequency and performance

- Change the maximum frequency of the intel processor — you learned how to do this when we discuss programmer's impact on performance
- LIKWID a profiling tool providing power/energy information
  - likwid-perfctr -g ENERGY [command_line]
  - Let's try blockmm and popcount and see what's happening!

matmul

```
-------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz
CPU type:       Intel Coffeelake processor
CPU clock:      3.70 GHz
-------------------------------------------------------------
```

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	56.7402	9.4567	9.4567	9.4567
Runtime unhalted [s] STAT	38089.9565	2.764721e-05	38089.9547	6348.3261
Clock [MHz] STAT	1.504735e+07	2530.3125	15027380	2.507892e+06
CPI STAT	29.6923	0.7684	11.1499	4.9487
Temperature [C] STAT	275	42	57	45.8333
Energy [J] STAT	247.1262	0	247.1262	41.1877
Power [W] STAT	26.1325	0	26.1325	4.3554
Energy PP0 [J] STAT	240.7318	0	240.7318	40.1220
Power PP0 [W] STAT	25.4563	0	25.4563	4.2427
Energy PP1 [J] STAT	0	0	0	0
Power PP1 [W] STAT	0	0	0	0
Energy DRAM [J] STAT	17.2582	0	17.2582	2.8764
Power DRAM [W] STAT	1.8250	0	1.8250	0.3042

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	202.3806	33.7301	33.7301	33.7301
Runtime unhalted [s] STAT	38089.3146	4.738974e-05	38089.3131	6348.2191
Clock [MHz] STAT	4.202013e+06	1135.3997	4196110	700335.5505
CPI STAT	14.9386	0.9997	6.2225	2.4898
Temperature [C] STAT	223	37	38	37.1667
Energy [J] STAT	88.4080	0	88.4080	14.7347
Power [W] STAT	2.6210	0	2.6210	0.4368
Energy PP0 [J] STAT	67.5248	0	67.5248	11.2541
Power PP0 [W] STAT	2.0019	0	2.0019	0.3337
Energy PP1 [J] STAT	0	0	0	0
Power PP1 [W] STAT	0	0	0	0
Energy DRAM [J] STAT	44.9120	0	44.9120	7.4853
Power DRAM [W] STAT	1.3315	0	1.3315	0.2219

pop count

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	11.8086	1.9681	1.9681	1.9681
Runtime unhalted [s] STAT	2.3622	0	2.3620	0.3937
Clock [MHz] STAT	17807.5257	2106.5416	4465.2964	2967.9209
CPI STAT	36.5243	0.3968	24.1941	6.0874
Temperature [C] STAT	249	39	49	41.5000
Energy [J] STAT	40.4858	0	40.4858	6.7476
Power [W] STAT	20.5715	0	20.5715	3.4286
Energy PP0 [J] STAT	39.1853	0	39.1853	6.5309
Power PP0 [W] STAT	19.9106	0	19.9106	3.3184
Energy PP1 [J] STAT	0	0	0	0
Power PP1 [W] STAT	0	0	0	0
Energy DRAM [J] STAT	1.9736	0	1.9736	0.3289
Power DRAM [W] STAT	1.0028	0	1.0028	0.1671

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	43.8276	7.3046	7.3046	7.3046
Runtime unhalted [s] STAT	2.3623	3.218655e-06	2.3620	0.3937
Clock [MHz] STAT	6822.7683	903.2673	1197.0151	1137.1281
CPI STAT	20.0735	0.3968	12.4175	3.3456
Temperature [C] STAT	218	36	37	36.3333
Energy [J] STAT	16.6005	0	16.6005	2.7668
Power [W] STAT	2.2726	0	2.2726	0.3788
Energy PP0 [J] STAT	11.7680	0	11.7680	1.9613
Power PP0 [W] STAT	1.6110	0	1.6110	0.2685
Energy PP1 [J] STAT	0	0	0	0
Power PP1 [W] STAT	0	0	0	0
Energy DRAM [J] STAT	7.3173	0	7.3173	1.2196
Power DRAM [W] STAT	1.0017	0	1.0017	0.1670

Power consumption to light on all transistors

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Dennardian Scaling

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W

Dennardian Broken

Chip

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

On ~ 50W

Off ~ 0W

Dark!

=100W!

The "power/energy cost" of doubling the clocking rate

$$Power_{new} = Power_{old} \times (\frac{f_{new}}{f_{old}})^3$$

$$Power_{new} = Power_{old} \times (2)^3 = Power_{old} \times 8$$

$$Speedup = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}} = \frac{5}{4} = 1.25$$

$$Energy_{new} = Power_{new} \times ET_{new}$$

$$= Power_{new} \times \frac{ET_{old}}{Speedup}$$

$$= Power_{old} \times 8 \times \frac{ET_{old}}{1.25} = 6.4 \times Power_{old} \times ET_{old}$$

$$= 6.4 \times Energy_{old}$$

Power consumption & power density

- The power consumption due to the switching of transistor states
- Dynamic power per transistor:

$$P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$$

  - α: average switches per cycle
  - C: capacitance
  - V: voltage
  - f: frequency, usually linear with V
  - N: the number of transistors

We cannot make chips always operating at very high frequencies

- Power density:

$$P_{density} = \frac{P}{area}$$

Moore's Law allows higher frequencies as transistors are smaller

Moore's Law makes this smaller

The "power" of doubling the clocking rate v.s. doubling the number of cores

$$P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$$

Doubling the clocking rate:

$$Power_{new} = Power_{old} \times (\frac{f_{new}}{f_{old}})^3$$

$$Power_{new} = Power_{old} \times (2)^3 = Power_{old} \times 8$$

Doubling the number of cores:

$$Power_{new} = Power_{old} \times number\_of\_cores = Power_{old} \times 2$$

Recap: Amdahl's Law on Multicore Architectures

- Symmetric multicore processor with n cores (if we assume the processor performance scales perfectly)

$$Speedup_{parallel}(f_{parallelizable}, n) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{n}}$$

What if we double the number of cores?

$$Power_{new} = Power_{old} \times number\_of\_cores = Power_{old} \times 2$$

Assume 40% of execution time can be parallelized

$$Speedup_{parallel}(f_{parallelizable}, n) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{n}}$$

$$= \frac{1}{(1 - 40\%) + \frac{40\,\%}{2}} = 1.25$$

$$Energy_{new} = Power_{new} \times ET_{new}$$

$$= Power_{new} \times \frac{ET_{old}}{Speedup}$$

$$= Power_{old} \times 2 \times \frac{ET_{old}}{1.25} = 1.6 \times Power_{old} \times ET_{old}$$

$$= 1.6 \times Energy_{old}$$

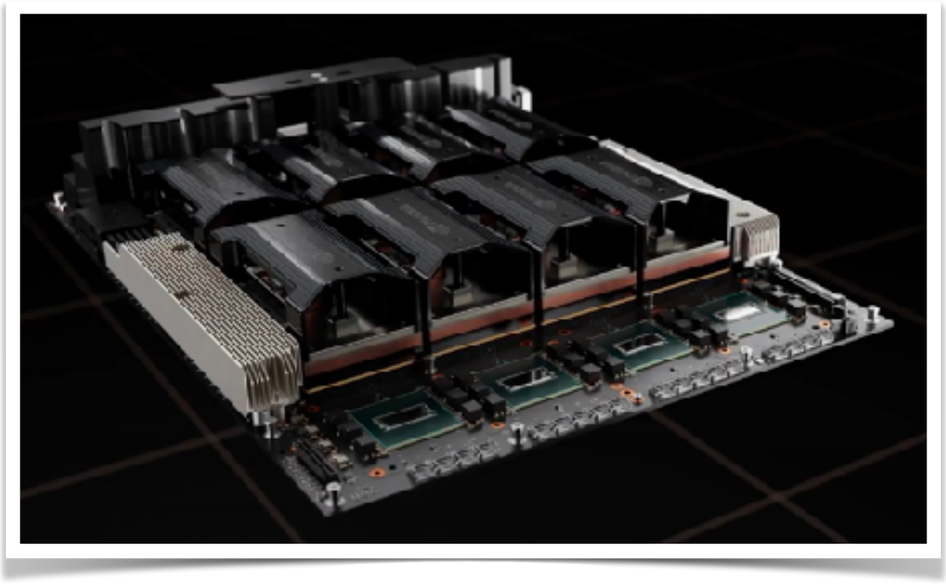
Same performance gain!

A better deal in terms of energy!

If you can add power budget…

NVIDIA Accelerator Specification Comparison			
	H100	A100 (80GB)	V100
FP32 CUDA Cores	16896	6912	5120
Tensor Cores	528	432	640
Boost Clock	~1.78GHz (Not Finalized)	1.41GHz	1.53GHz
Memory Clock	4.8Gbps HBM3	3.2Gbps HBM2e	1.75Gbps HBM2
Memory Bus Width	5120-bit	5120-bit	4096-bit
Memory Bandwidth	3TB/sec	2TB/sec	900GB/sec
VRAM	80GB	80GB	16GB/32GB
FP32 Vector	60 TFLOPS	19.5 TFLOPS	15.7 TFLOPS
FP64 Vector	30 TFLOPS	9.7 TFLOPS (1/2 FP32 rate)	7.8 TFLOPS (1/2 FP32 rate)
INT8 Tensor	2000 TOPS	624 TOPS	N/A
FP16 Tensor	1000 TFLOPS	312 TFLOPS	125 TFLOPS
TF32 Tensor	500 TFLOPS	156 TFLOPS	N/A
FP64 Tensor	60 TFLOPS	19.5 TFLOPS	N/A
Interconnect	NVLink 4 18 Links (900GB/sec)	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)
GPU	GH100 (814mm2)	GA100 (826mm2)	GV100 (815mm2)
Transistor Count	80B	54.2B	21.1B
TDP	700W	400W	300W/350W
Manufacturing Process	TSMC 4N	TSMC 7N	TSMC 12nm FFN
Interface	SXM5	SXM4	SXM2/SXM3
Architecture	Hopper	Ampere	Volta

https://www.workstationspecialist.com/product/nvidia-tesla-a100/



https://www.servethehome.com/wp-content/uploads/2022/03/NVIDIA-GTC-2022-H100-in-HGX-H100.jpg

Computer
Science &
Engineering

203

つづく