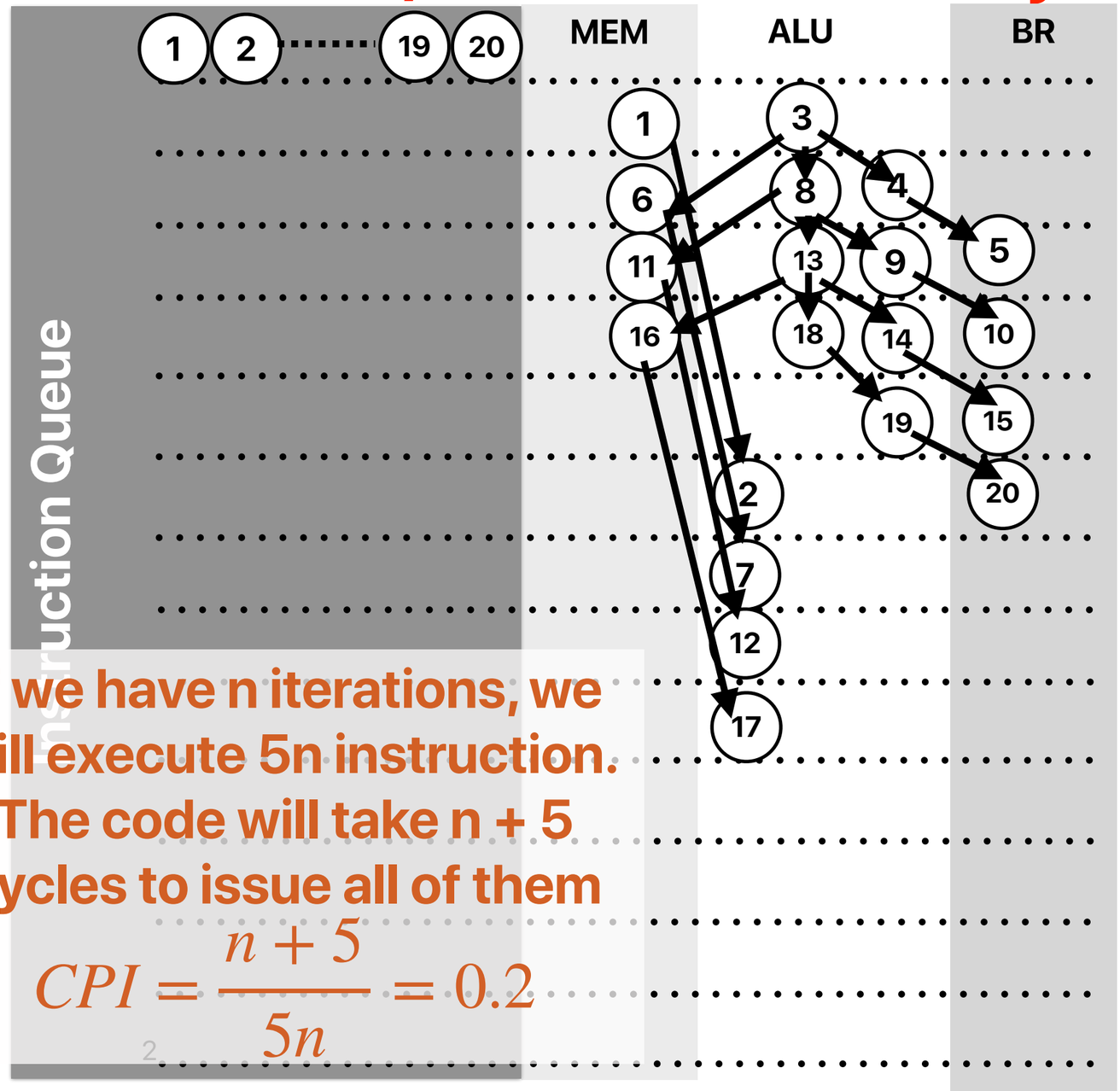
Programming on Modern Processors: the Single Thread Version (cont.) & Multithreaded Architectures

Hung-Wei Tseng

Recap: What if we have “unlimited” fetch/issue width — “array”

```
①  .L9:    cmpq  $1, 8(%rax)
②          sbbl  $-1, %edx
③          addq  $16, %rax
④          cmpq  %rdi, %rax
⑤          jne   .L9
⑥  .L9:    cmpq  $1, 8(%rax)
⑦          sbbl  $-1, %edx
⑧          addq  $16, %rax
⑨          cmpq  %rdi, %rax
⑩          jne   .L9
⑪  .L9:    cmpq  $1, 8(%rax)
⑫          sbbl  $-1, %edx
⑬          addq  $16, %rax
⑭          cmpq  %rdi, %rax
⑮          jne   .L9
⑯  .L9:    cmpq  $1, 8(%rax)
⑰          sbbl  $-1, %edx
⑱          addq  $16, %rax
⑲          cmpq  %rdi, %rax
⑳          jne   .L9
```



If we have n iterations, we will execute 5n instruction. The code will take n + 5 cycles to issue all of them

$$CPI = \frac{n+5}{5n} = 0.2$$

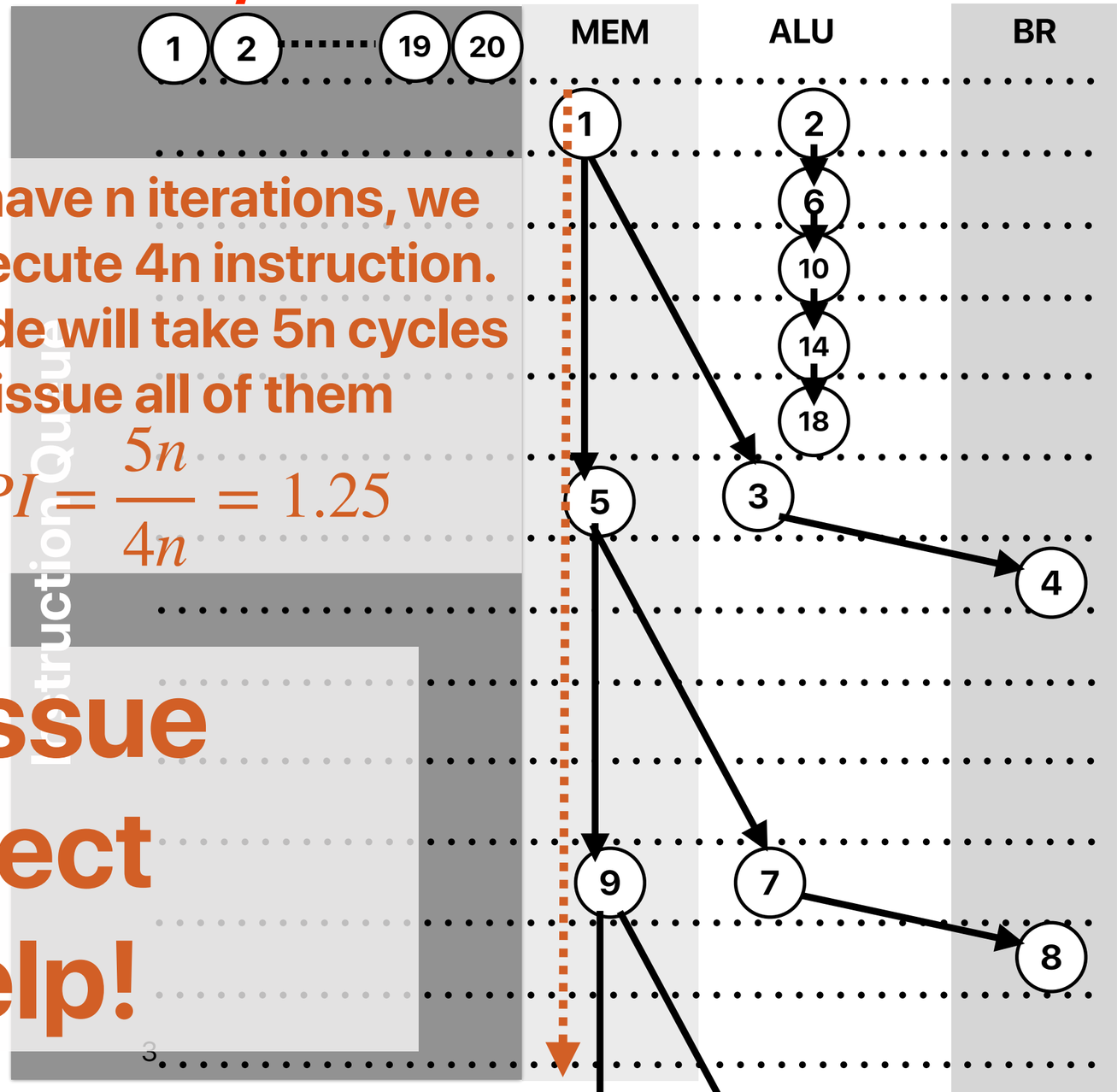
What if we have "unlimited" fetch/issue width — "linked list"

```
①  .L3:     movq    8(%rdi), %rdi
②           addl    $1, %eax
③           testq   %rdi, %rdi
④           jne     .L3
⑤  .L3:     movq    8(%rdi), %rdi
⑥           addl    $1, %eax
⑦           testq   %rdi, %rdi
⑧           jne     .L3
⑨  .L3:     movq    8(%rdi), %rdi
⑩           addl    $1, %eax
⑪           testq   %rdi, %rdi
⑫           jne     .L3
⑬  .L3:     movq    8(%rdi), %rdi
⑭           addl    $1, %eax
⑮           testq   %rdi, %rdi
⑯           jne     .L3
⑰  .L3:     movq    8(%rdi), %rdi
⑱           addl    $1, %eax
⑲           testq   %rdi, %rdi
⑳           jne     .L3
```

If we have n iterations, we will execute 4n instruction. The code will take 5n cycles to issue all of them

$$CPI = \frac{5n}{4n} = 1.25$$

Even unlimited issue width and a perfect cache cannot help!



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;
 }
```

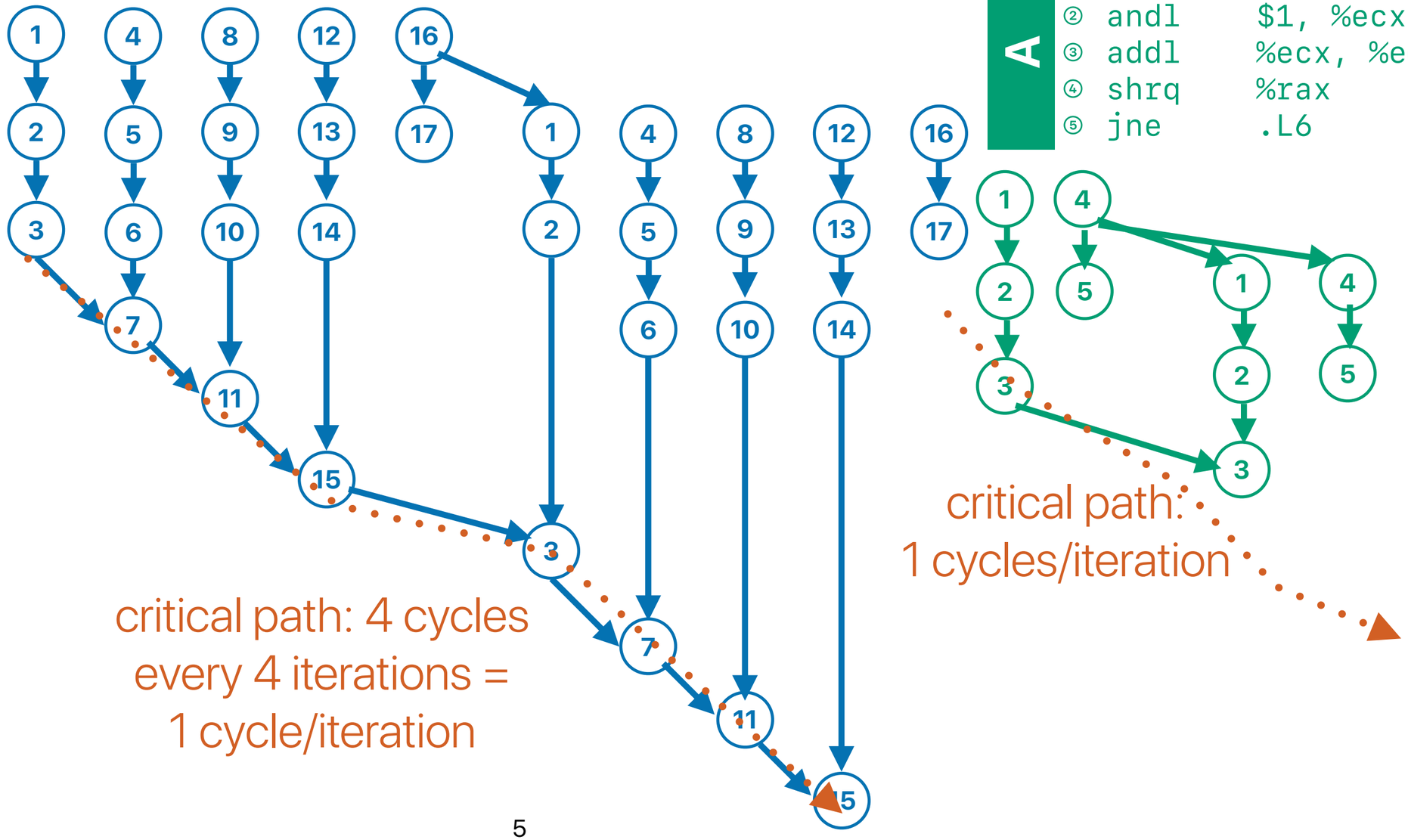
E

```
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;
}
```

Why is B better than A?

B

```
① movl   %ecx, %eax
② andl   $1, %eax
③ addl   %edx, %eax
④ movq   %rcx, %rdx
⑤ shrq   %rdx
⑥ andl   $1, %edx
⑦ addl   %eax, %edx
⑧ movq   %rcx, %rax
⑨ shrq   $2, %rax
⑩ andl   $1, %eax
⑪ addl   %edx, %eax
⑫ movq   %rcx, %rdx
⑬ shrq   $3, %rdx
⑭ andl   $1, %edx
⑮ addl   %eax, %edx
⑯ shrq   $4, %rcx
⑰ jne    .L6
```



critical path: 4 cycles
every 4 iterations =
1 cycle/iteration

A

```
① movl   %eax, %ecx
② andl   $1, %ecx
③ addl   %ecx, %edx
④ shrq   %rax
⑤ jne    .L6
```

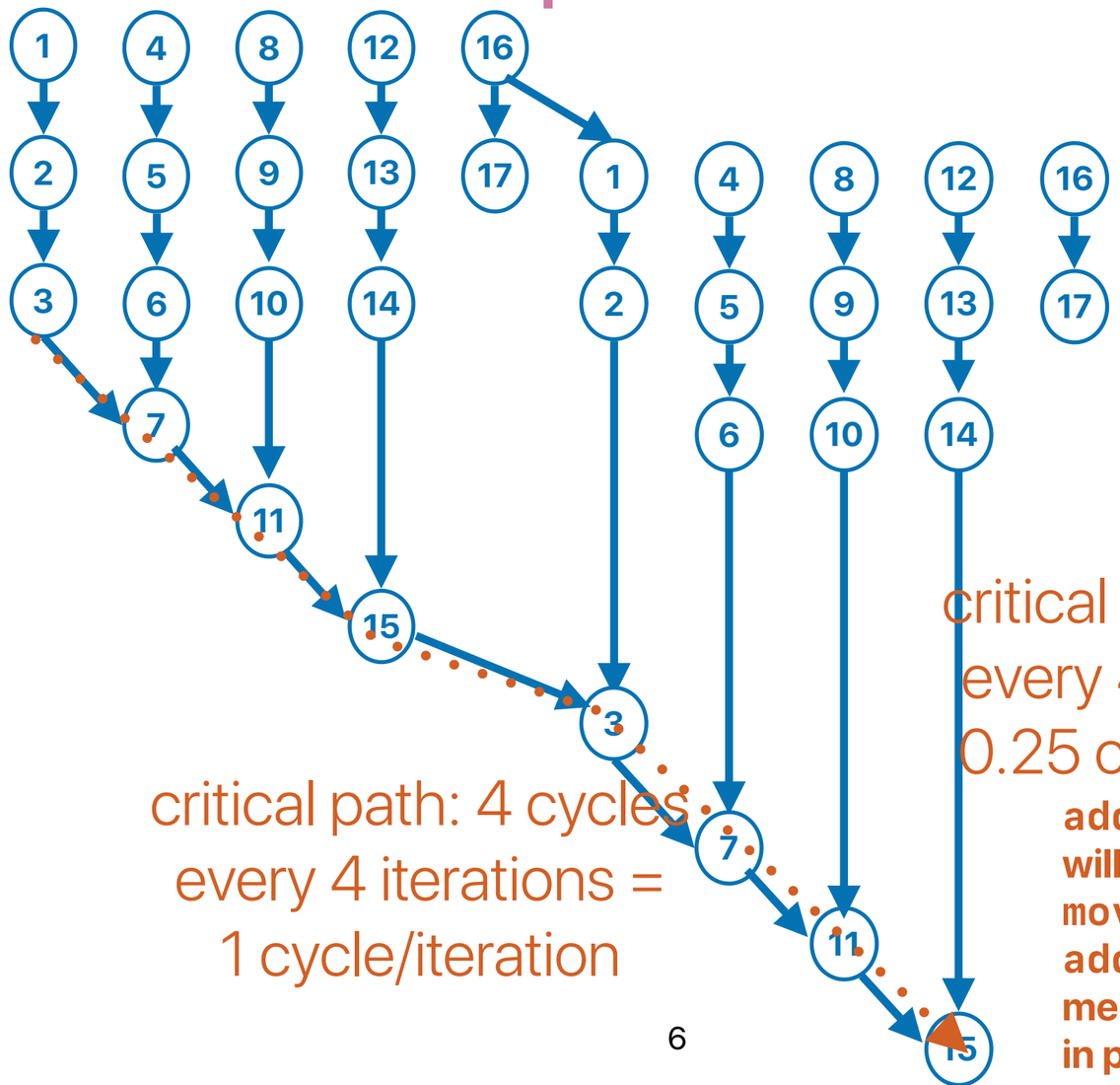

critical path:
1 cycles/iteration

Why is C better than B?

B

```
① movl    %ecx, %eax
② andl    $1, %eax
③ addl    %edx, %eax
④ movq    %rcx, %rdx
⑤ shrq    %rdx
⑥ andl    $1, %edx
⑦ addl    %eax, %edx
⑧ movq    %rcx, %rax
⑨ shrq    $2, %rax
⑩ andl    $1, %eax
⑪ addl    %edx, %eax
⑫ movq    %rcx, %rdx
⑬ shrq    $3, %rdx
⑭ andl    $1, %edx
⑮ addl    %eax, %edx
⑯ shrq    $4, %rcx
⑰ jne     .L6
```

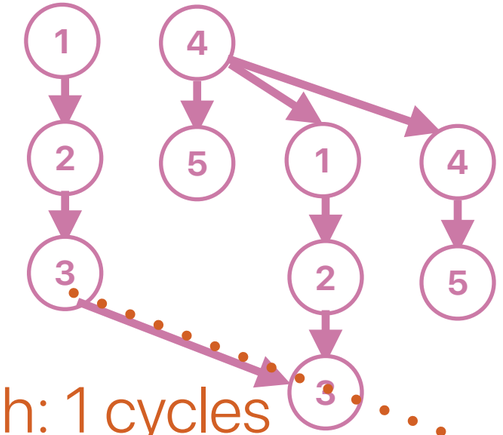
These 5 instructions represent 4 iterations!!!



critical path: 4 cycles
every 4 iterations =
1 cycle/iteration

C

```
.L6:
① movq  %rcx, %rdi
② andl  $15, %edi
③ addl  (%rsp,%rdi,4), %eax
④ shrq  $4, %rcx
⑤ jne   .L6
```



critical path: 1 cycles
every 4 iterations =
0.25 cycle/iteration

`addl (%rsp,%rdi,4), %eax`
will be translated into uOPs of
`mov (%rsp,%rdi,4), something` and
`addl something, %eax` –
memory operations can be executed in parallel

Takeaways: programming modern processors

- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
  - Avoiding data dependent operations (e.g., pointer chasing)
  - Avoiding inter-iteration dependencies (e.g., linked list, trees)
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.
- With caches, we can potentially use small lookup tables to replace more expensive data dependent operations

3:00

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

Outline

- Programming on Modern, Single-threaded Processors (cont.)
- Parallel architectures
  - Simultaneous multithreading (SMT)
  - Chip multiprocessors (CMP)

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C
  - ② D has significantly lower branch mis-prediction rate than C
  - ③ D has significantly fewer branch instructions than C
  - ④ D has better CPI than C

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

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;
 }
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C
  - ② D has significantly lower branch mis-prediction rate than C
  - ③ D has significantly fewer branch instructions than C
  - ④ D has better CPI than C

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

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;
}
```

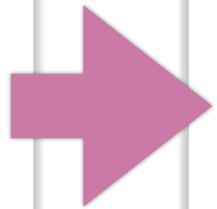
D

```
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;
 }
```

Loop unrolling eliminates all branches!

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;
 }
```



```
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};
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
     return c;
}
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C — Compiler can do loop unrolling — no branches
  - ② D has significantly lower branch mis-prediction rate than C — Could be
  - ③ D has significantly fewer branch instructions than C — maybe eliminated through loop unrolling…
  - ④ D has better CPI than C — about the same

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

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;
 }
```

Takeaways: programming modern processors

- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
  - Avoiding data dependent operations (e.g., pointer chasing)
  - Avoiding inter-iteration dependencies (e.g., linked list, trees)
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.
- With caches, we can potentially use small lookup tables to replace more expensive data dependent operations
- Making your code more predictable is the key!
  - Compilers can confidently perform aggressive optimizations
  - Branch predictors can be more accurate
  - Cache miss rate can be really low

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

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

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

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;
}
```

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

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

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

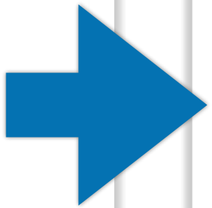
```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
          switch((x & 0xF))
          {
               case 1: c+=1; break;
               case 2: c+=1; break;
               case 3: c+=2; break;
               case 4: c+=1; break;
               case 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;
 }
```

Why is E the slowest?

It's not predicting "taken" or "not taken", it's about which address to jump — hard for BPUs

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;
}
```



```
.L11:
        movq    %r9, %rcx
        andl    $15, %ecx
        movslq  (%r8,%rcx,4), %rcx
        addq    %r8, %rcx
        notrack jmp     *%rcx
.L7:
        .long   .L5-.L7
        .long   .L10-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L8-.L7
        .long   .L6-.L7
.L8:
        addl    $3, %eax
.L5:
        shrq    $4, %r9
        subq    $1, %rsi
        jne     .L11
        cltq
        addq    %rax, %rbx
        subl    $1, %edi
        jne     .L12
```

```
.L9:
        .cfi_restore_state
        addl    $2, %eax
        jmp     .L5
        .p2align 4,,10
        .p2align 3
.L10:
        addl    $1, %eax
        jmp     .L5
        .p2align 4,,10
        .p2align 3
.L6:
        addl    $4, %eax
        jmp     .L5
```

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

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

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;
 }
```

Hardware acceleration

- Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a
- In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number
  - You need to compile the program with -m64 -msse4.2 flags to enable these new features

```
#include <smmintrin.h>
inline int popcount(uint64_t x) {
      int c = _mm_popcnt_u64(x);
      return c;
}
```

Summary of popcounts

	Cycles	IC	IPC/ILP	ET	# of branches	Branch mis-prediction rate
A	334270949858	118508036769	2.821	23.237	65366730559	0.012
B	290179950840	68341242029	4.246	13.419	18319495534	0.004
C	102882218758	27580097715	3.730	5.380	18129328282	0.004
D	80043714833	19072124536	4.197	3.754	1037120144	0.000
E	253238690876	376641071475	0.672	73.977	73967642413	0.184
SSE4.2	22089754243	8444815558	2.616	1.654	1014543318	0.000

Takeaways: programming modern processors

- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
  - Avoiding data dependent operations (e.g., pointer chasing)
  - Avoiding inter-iteration dependencies (e.g., linked list, trees)
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.
- With caches, we can potentially use small lookup tables to replace more expensive data dependent operations
- Making your code more predictable is the key!
  - Compilers can confidently perform aggressive optimizations
  - Branch predictors can be more accurate
  - Cache miss rate can be really low
- If there is a hardware feature supporting the desire computation — we should try it!

Summary of popcounts

	Cycles	IC	IPC/ILP	ET	# of branches	Branch mis-prediction rate
A	334270949858	118508036769	2.821	23.237	65366730559	0.012
B	290179950840	68341242029	4.246	13.419	18319495534	0.004
C	102882218758	27580097715	3.730	5.380	18129328282	0.004
D	80043714833	19072124536	4.197	3.754	1037120144	0.000
E	253238690876	376641071475	0.672	73.977	73967642413	0.184
SSE4.2	22089754243	8444815558	2.616	1.654	1014543318	0.000

Most of time, we can't fully utilize the function units

Top at 4.3

Best performing one at 2.7

Performance code may not even fully utilize FUs, either.

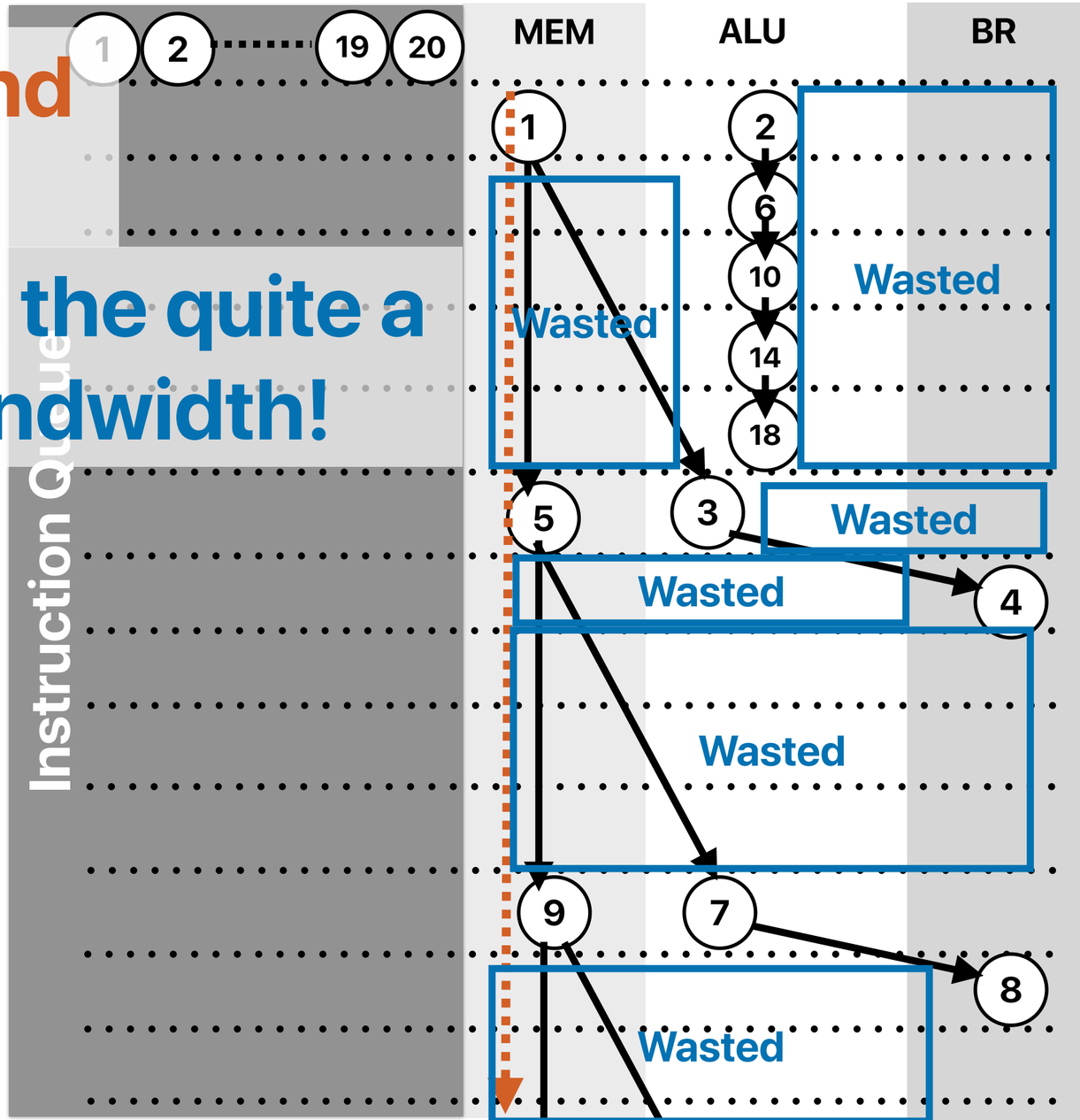
What if we have "unlimited" fetch/issue width — "linked list"

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
```



Tips of programming on modern processors

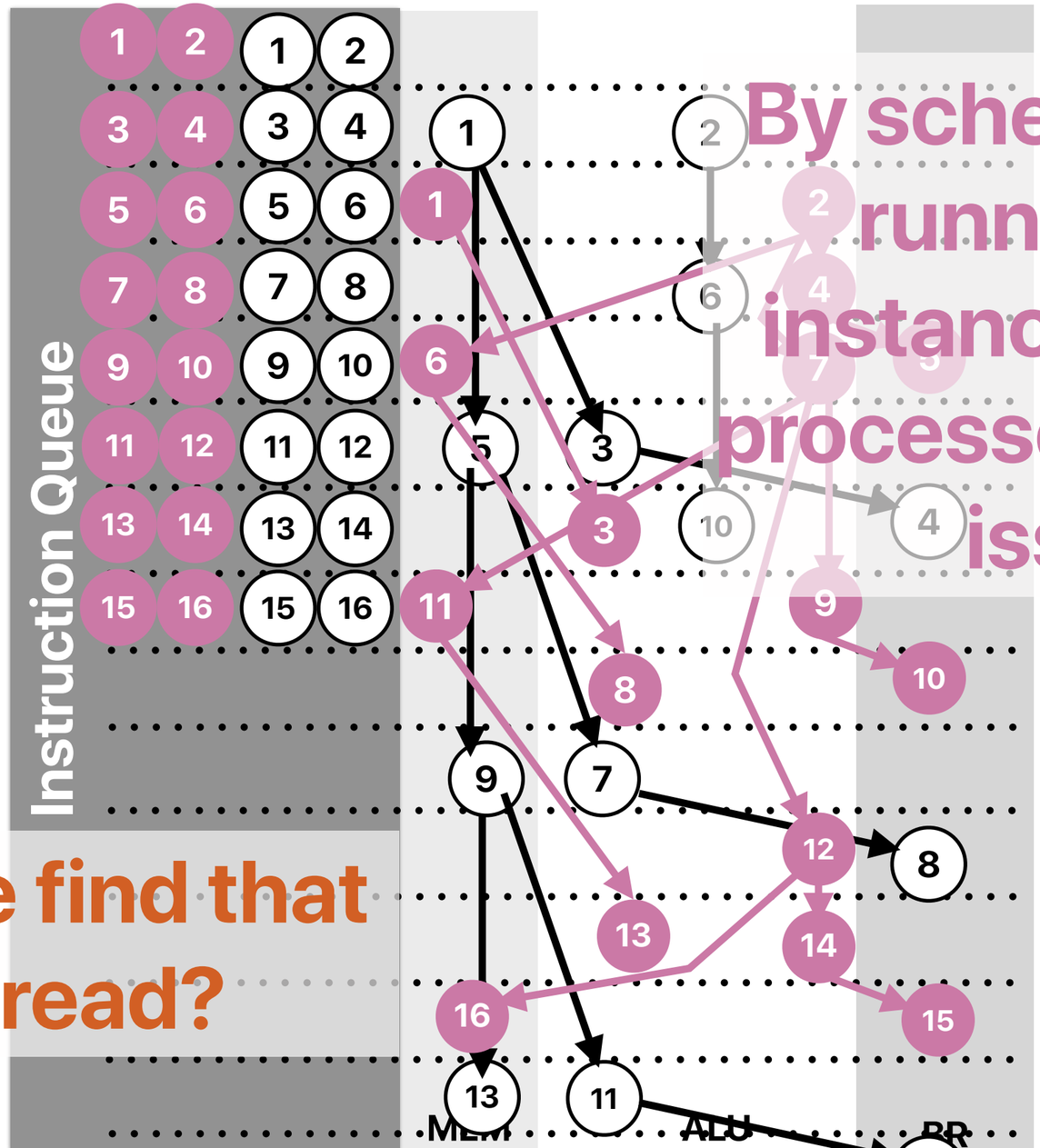
- Minimize the critical path operations
  - Don’t forget about optimizing cache/memory locality first!
    - Memory latencies are still way longer than any arithmetic instruction
    - Can we use arrays/hash tables instead of lists?
  - Branch can be expensive as pipeline get deeper
    - Sorting
    - 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

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!

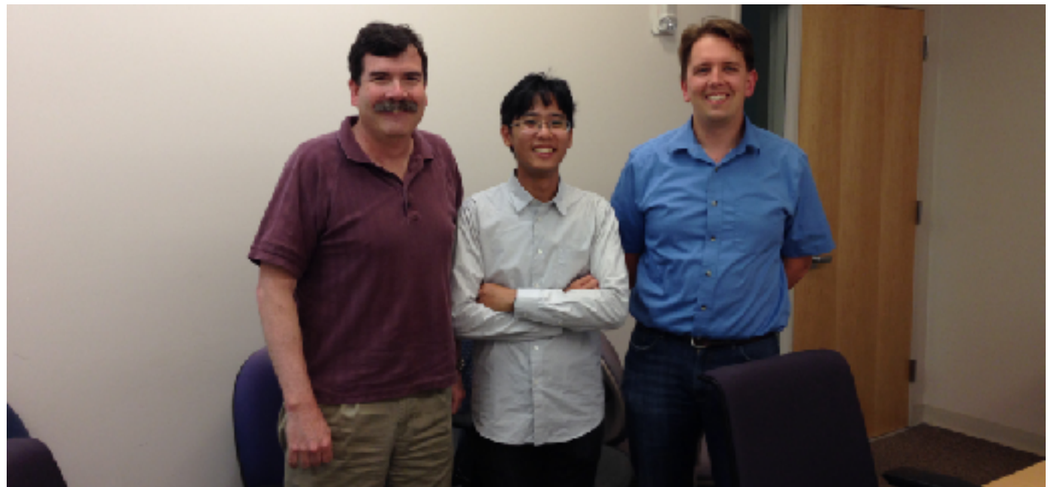
Where can we find that "other" thread?

Where to find another “thread”

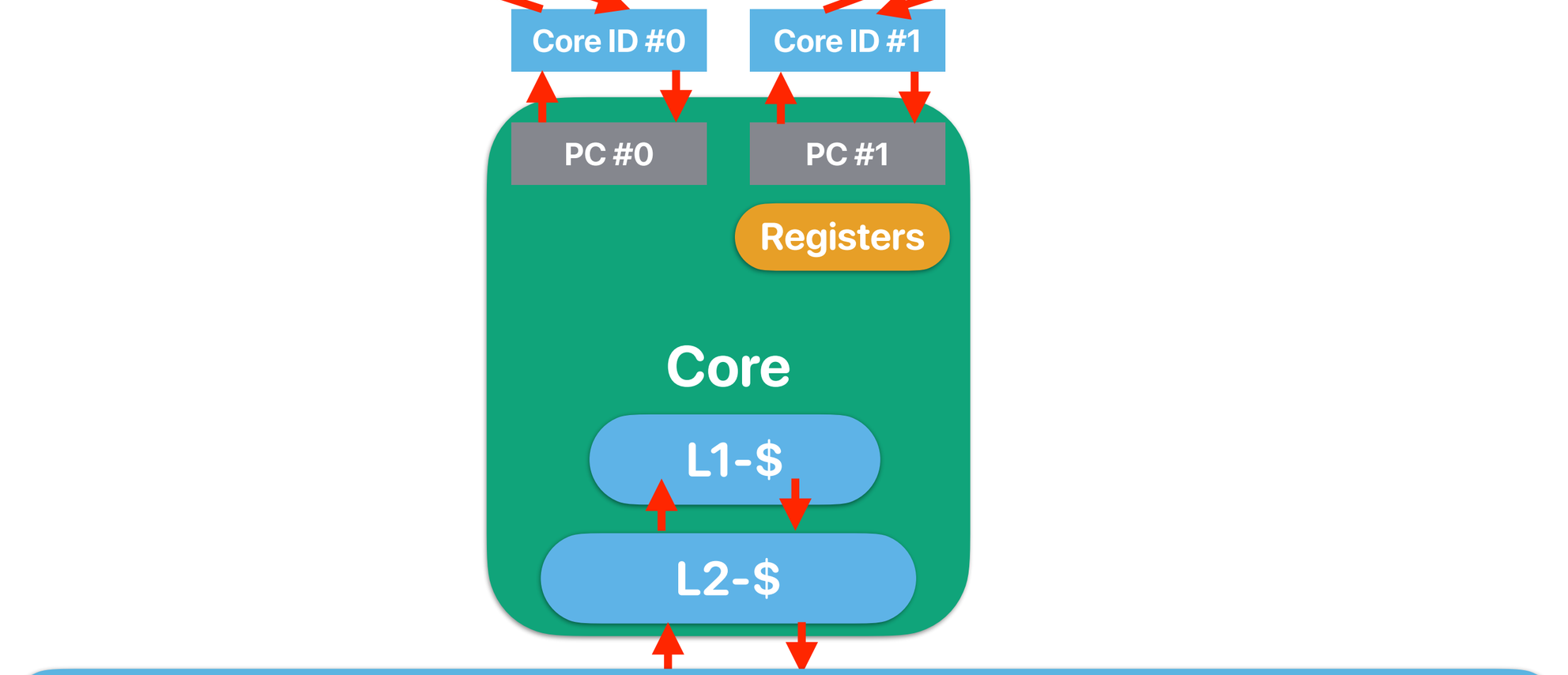
- Another process/running program forked from a completely different program
- Another process forked/cloned from the current process
  - The forked program cloned the memory content at the forked time
  - The forked program cannot view the memory space of the original process
  - The forked program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



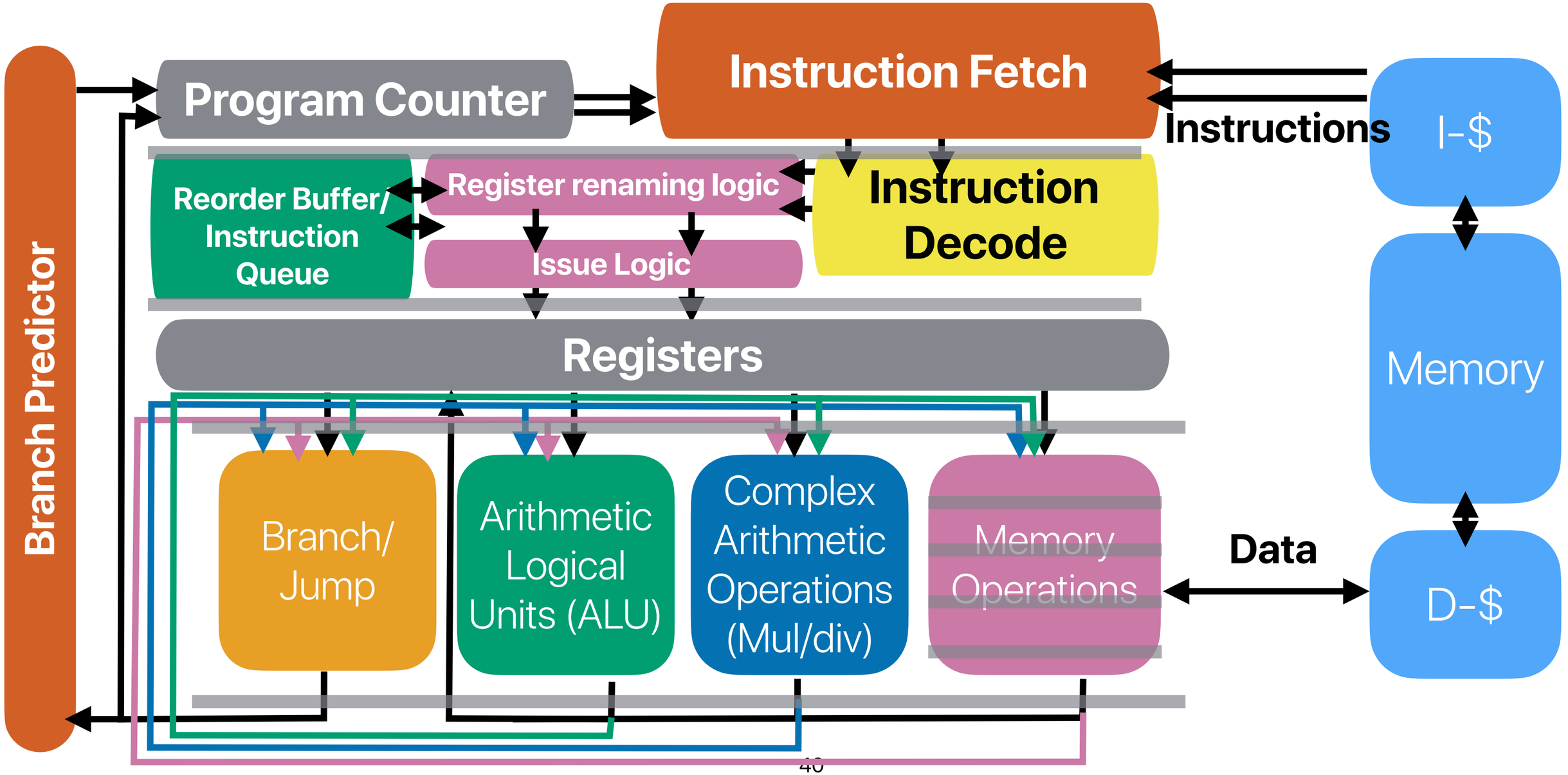
SMT from the user/OS' perspective



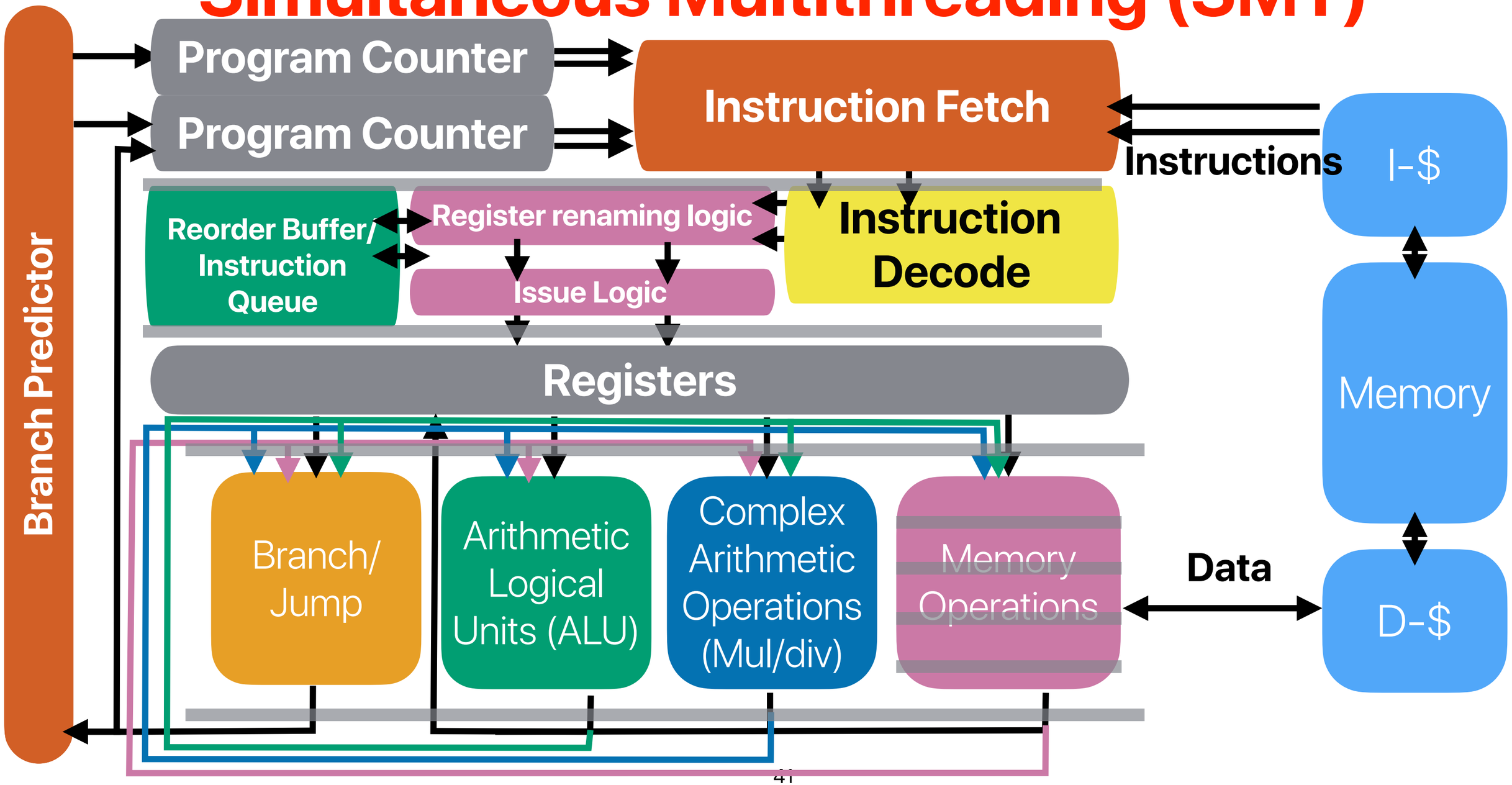
How do we support two running programs in one pipeline?

- 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)



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

Pros/Cons of SMT

- How many of the following statements about SMT is/are correct?
  - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  - 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
  - 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.

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

Pros/Cons of SMT

- How many of the following statements about SMT is/are correct?
  - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  - 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
  - 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.
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Pros/Cons of SMT

- How many of the following statements about SMT is/are correct?
  - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  - 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
  - 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.

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

We can execute from other threads/contexts instead of the current one

hurt, b/c you are sharing resource with other threads.

We can execute from other threads/ contexts instead of the current one

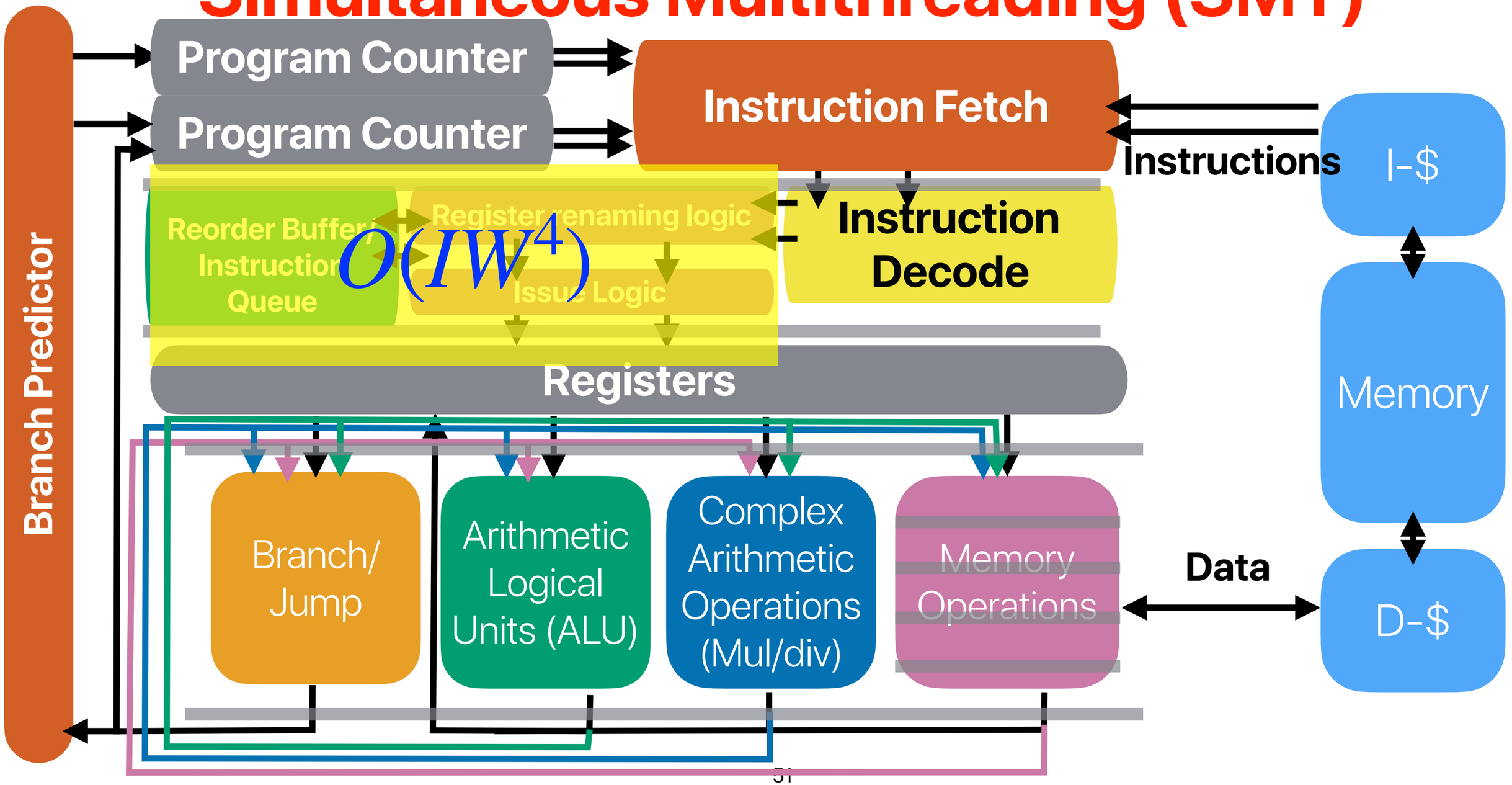
b/c we're sharing the cache

```
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)



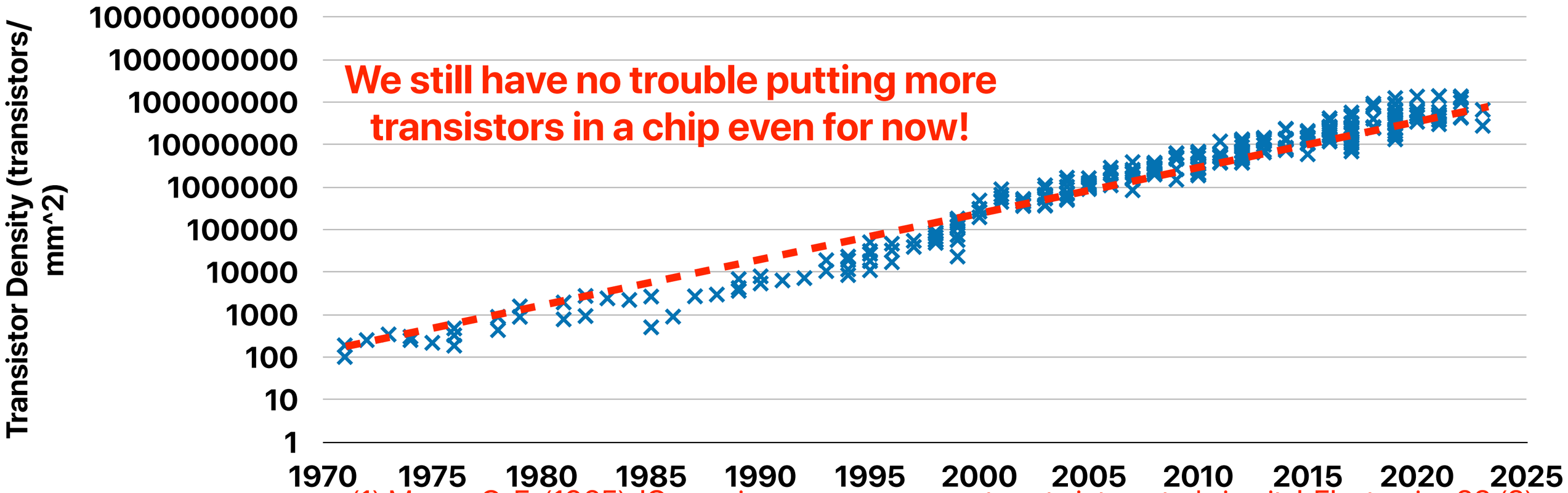
Takeaways: parallel architectures

- 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

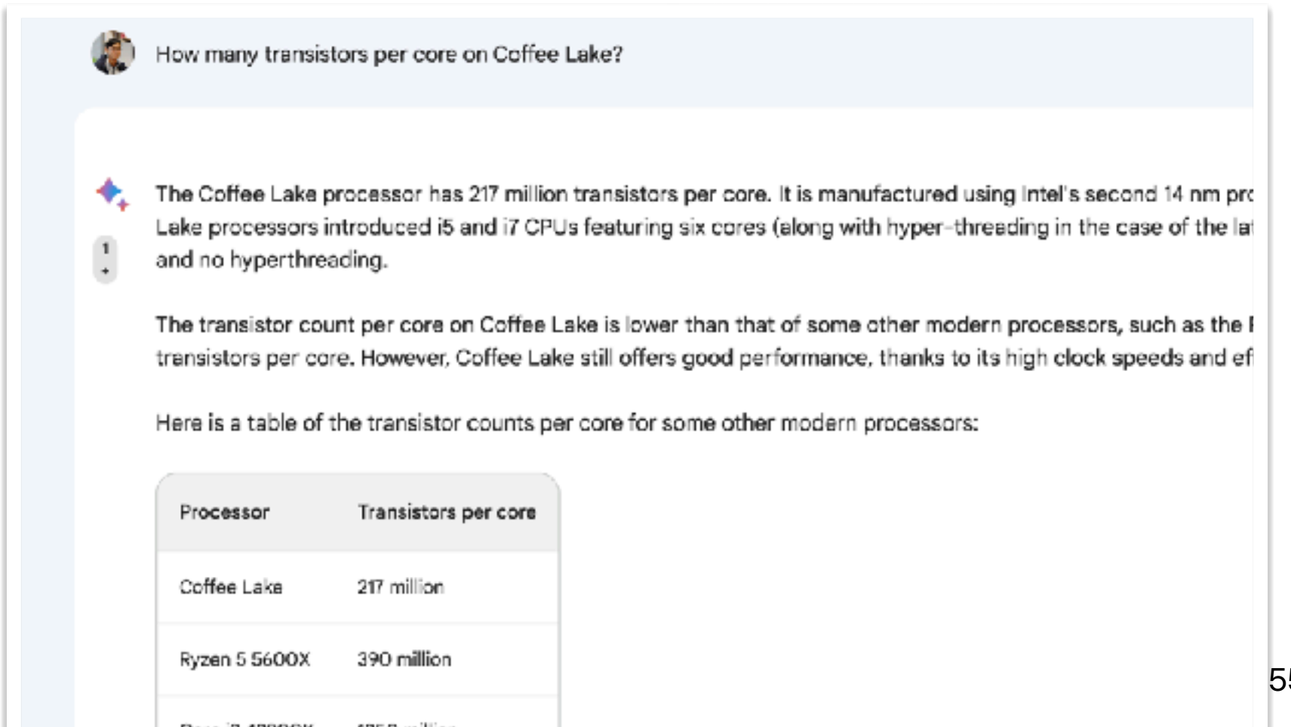


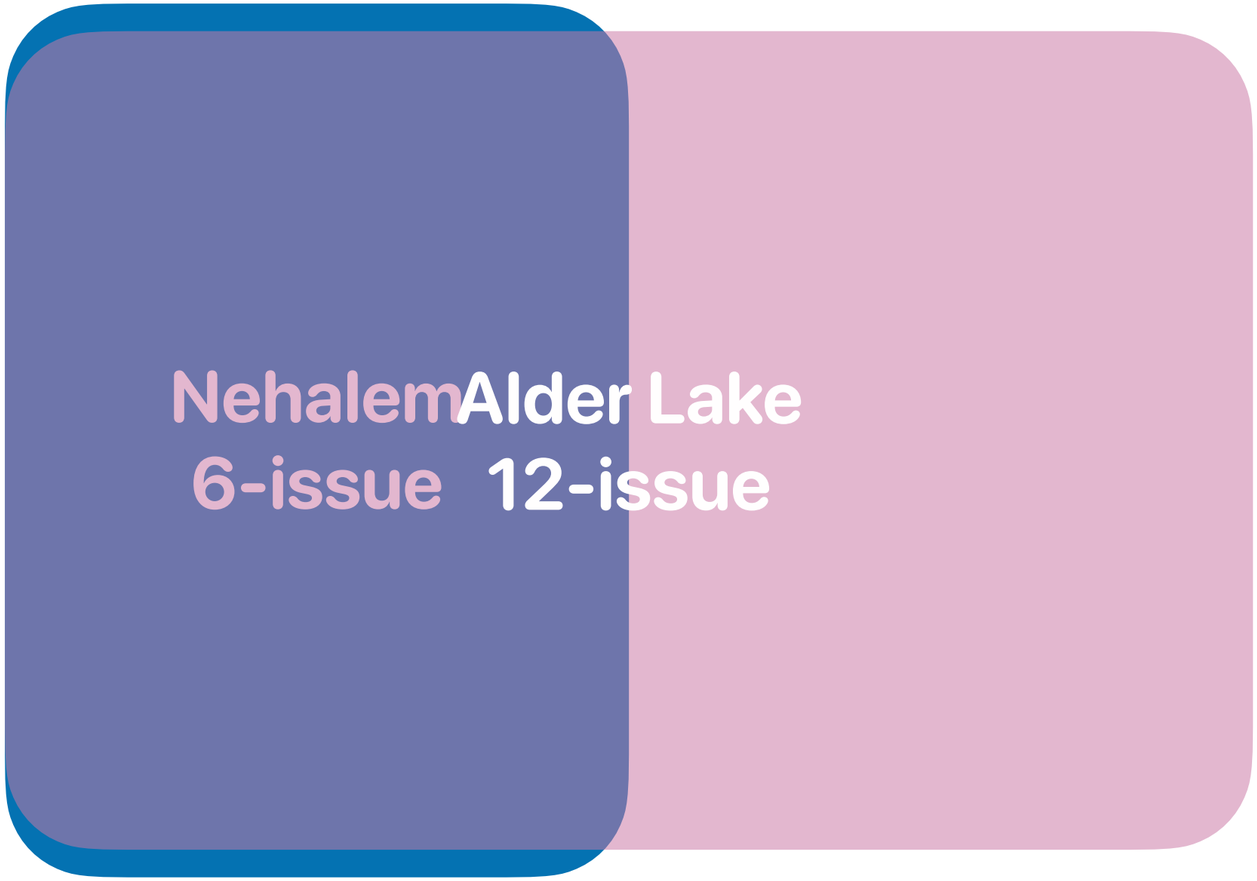
(1) Moore, G. E. (1965), 'Cramming more components onto integrated circuits', Electronics 38 (8) .

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





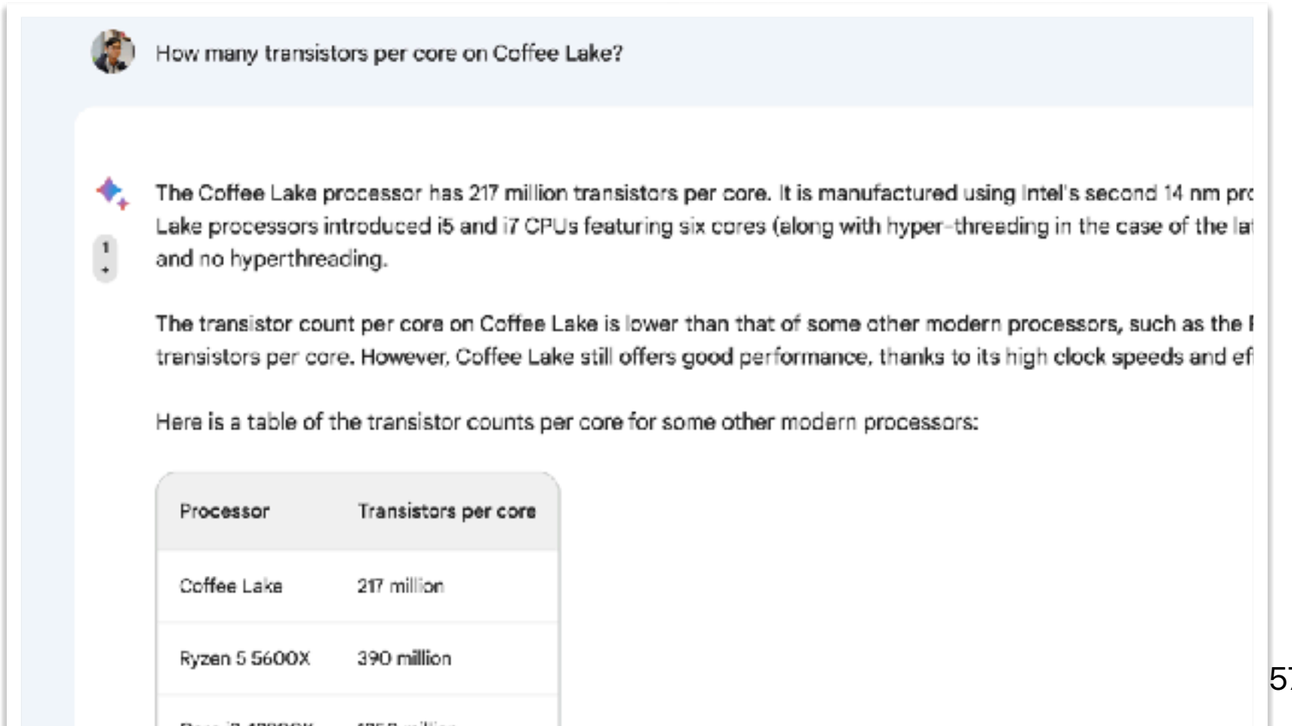
Based on https://en.wikipedia.org/wiki/Transistor_count

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



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

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

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

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

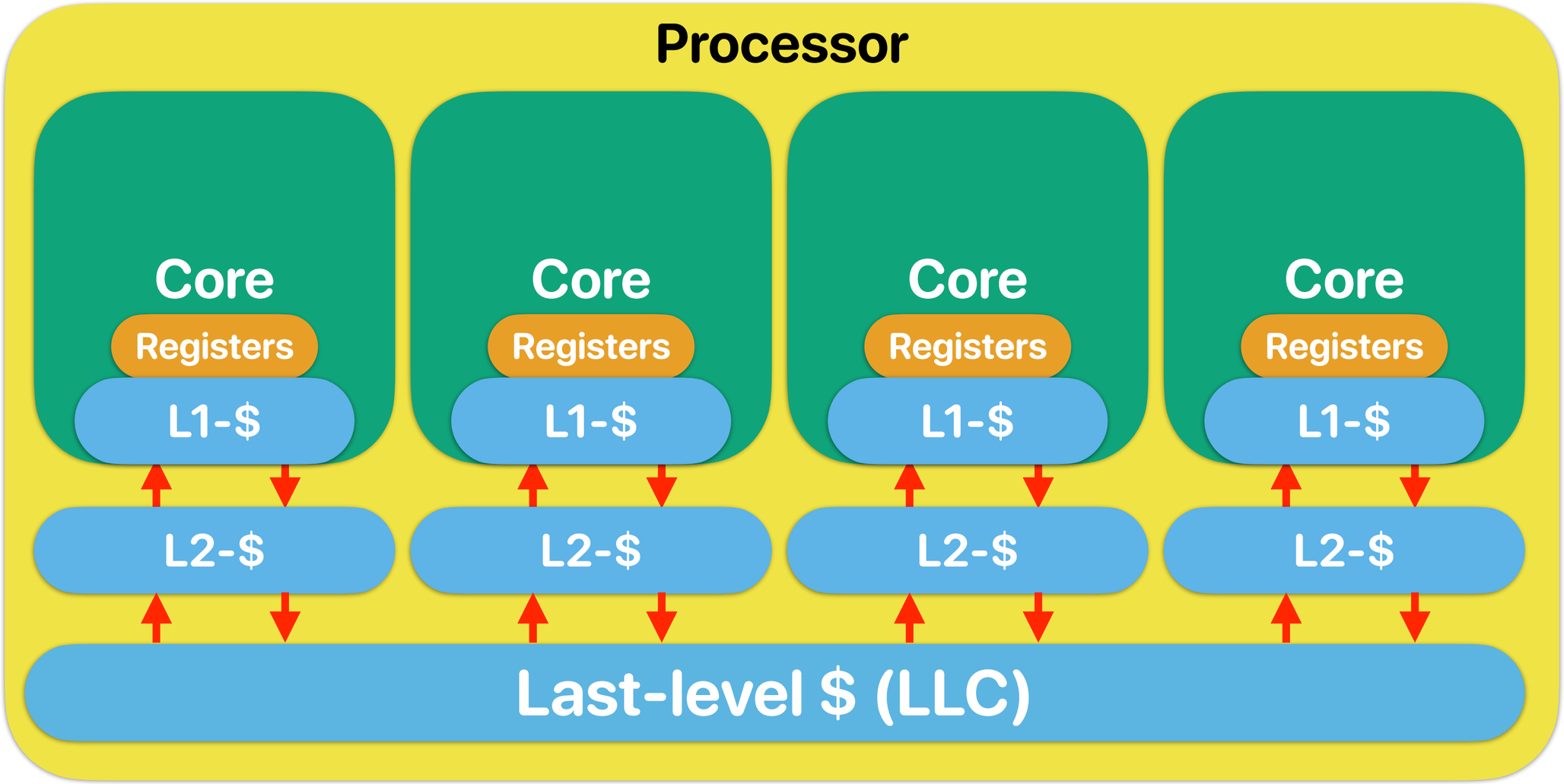
2x 3-issue ALUs Nehalem

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

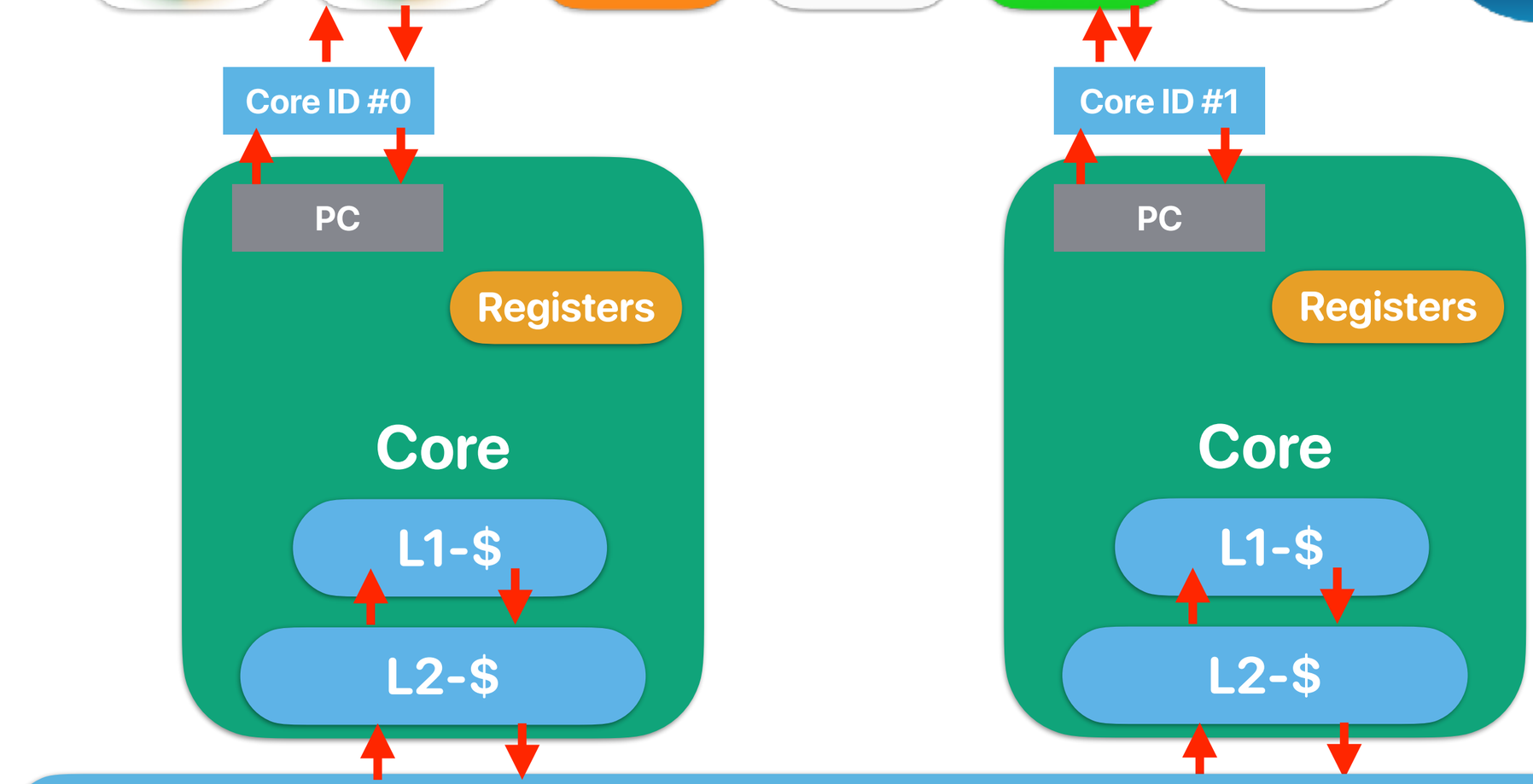
1x 5-issue ALUs Alder Lake

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

Concept of CMP



CMP from the user/OS' perspective



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

SMT v.s. CMP

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

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

SMT v.s. CMP

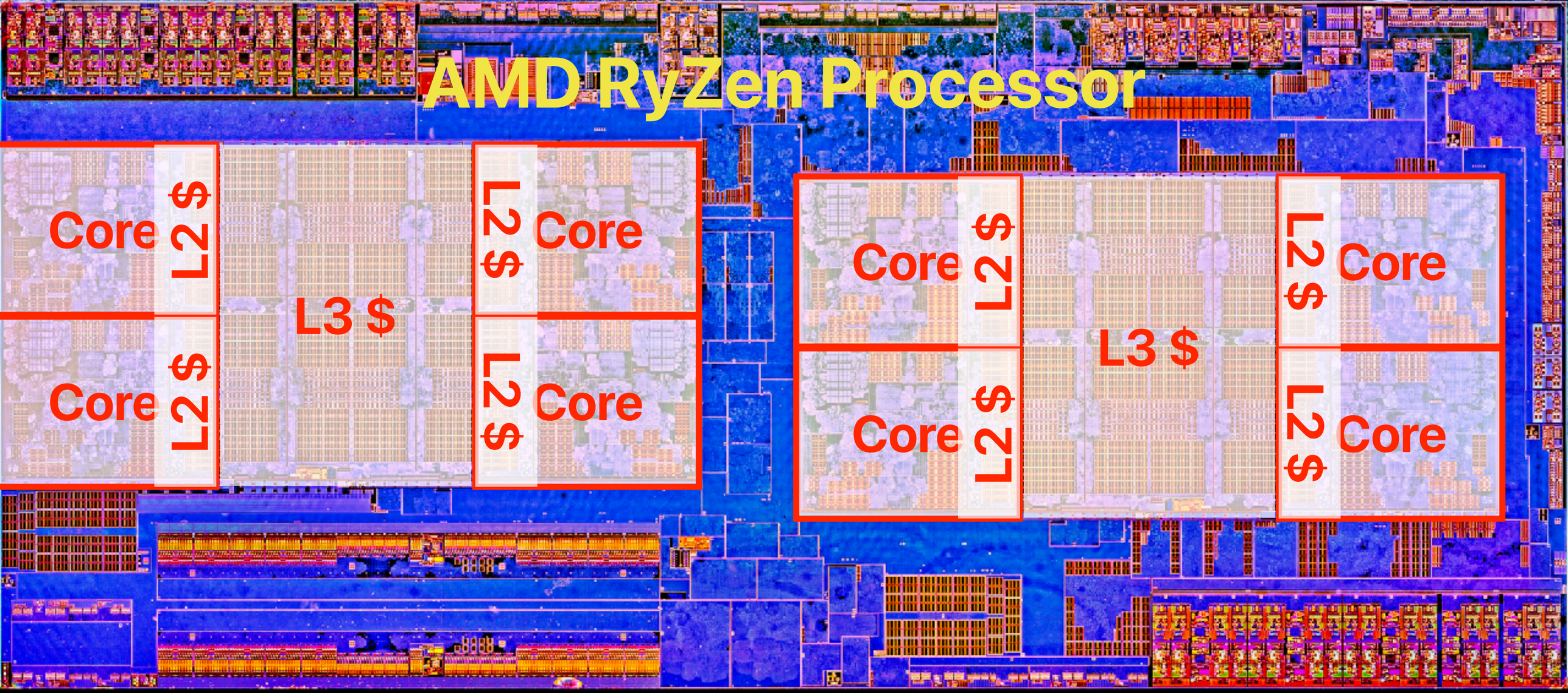
- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many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

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

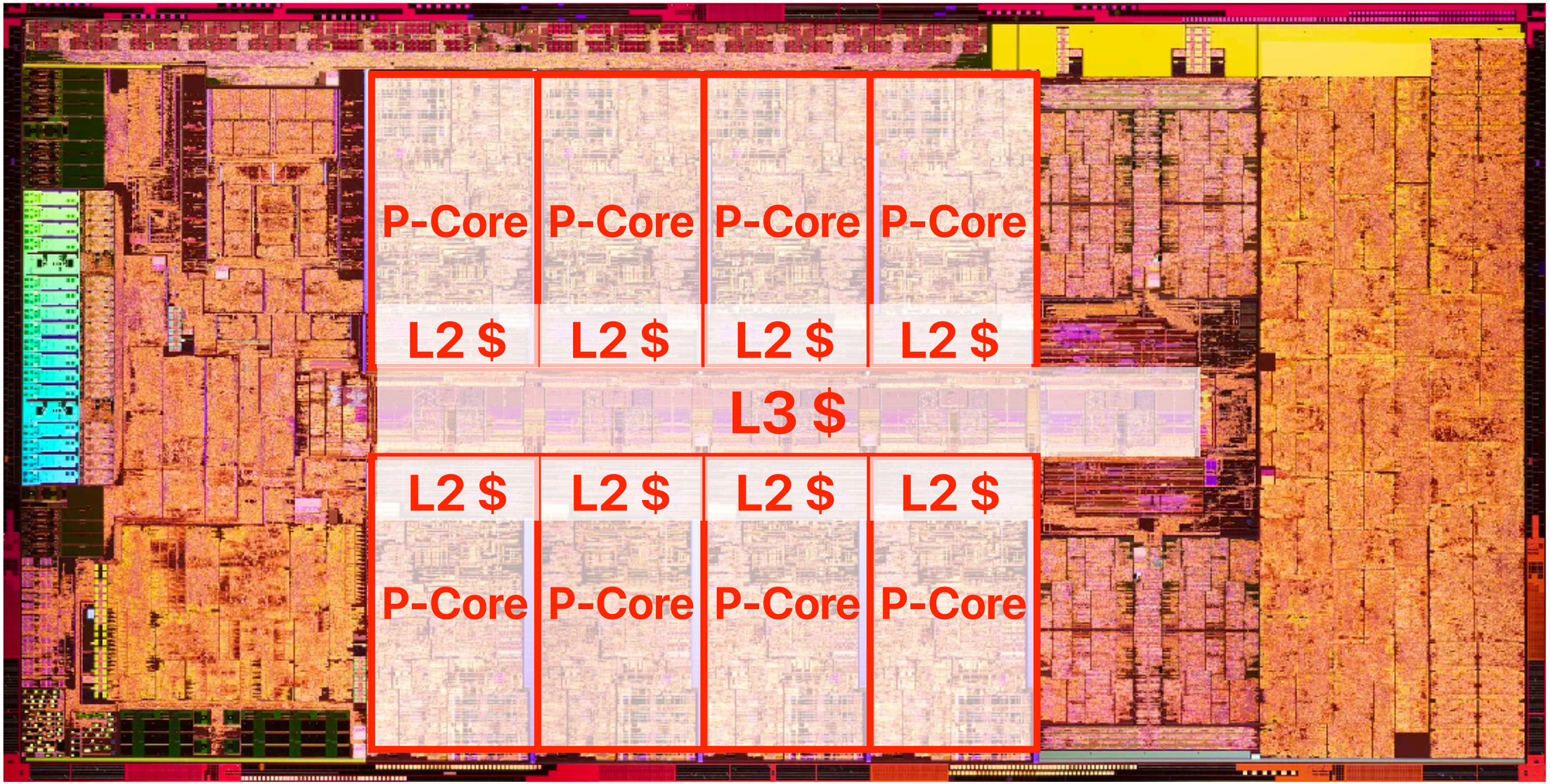
There is no clear win on each — why not having both?

```
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



SMT v.s. CMP

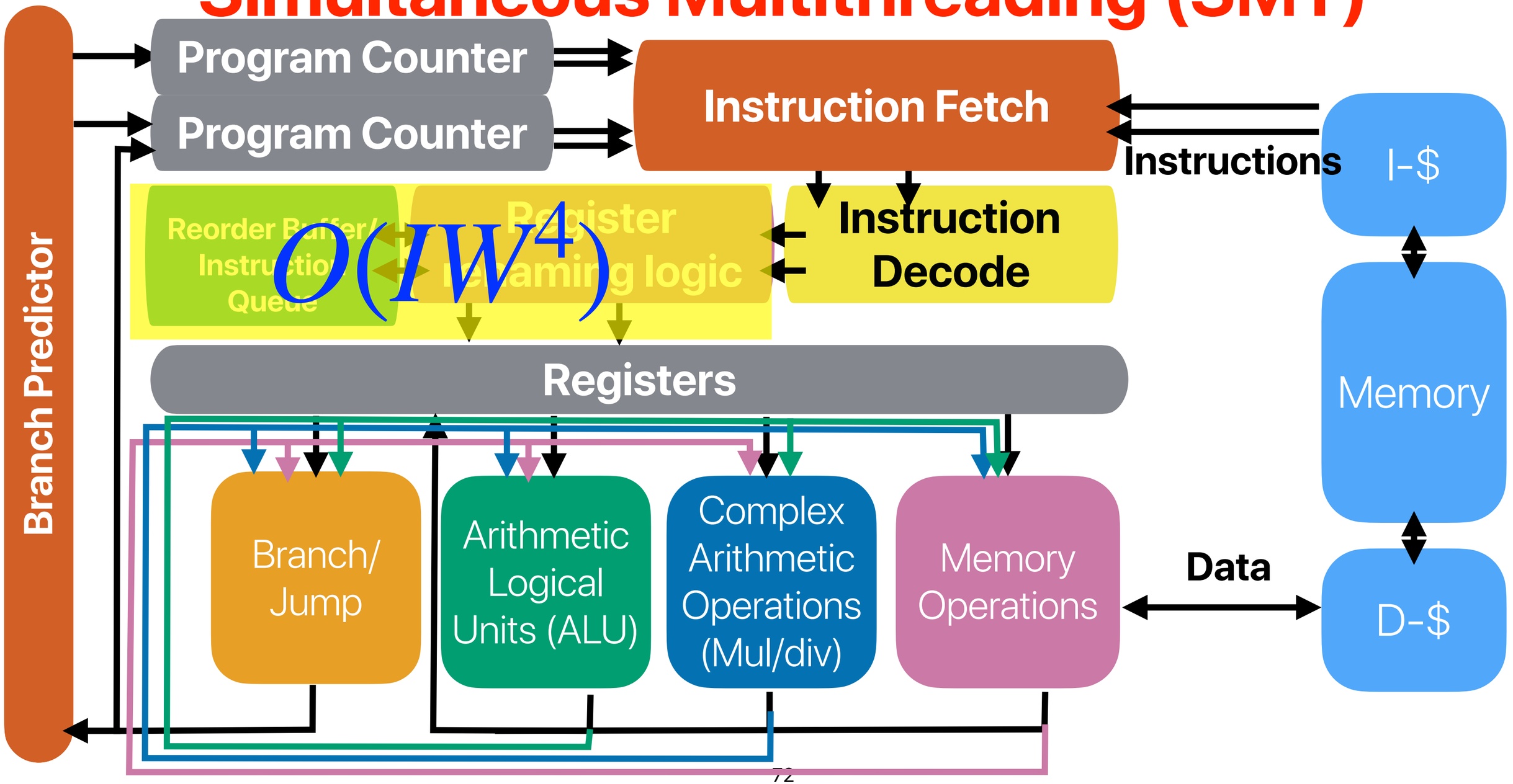
- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many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

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

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

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

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

Announcements

- **Assignment 4** is **due tonight**
- **Assignment 5 will be released** this Saturday

Computer
Science &
Engineering

203

つづく