Programming on Modern Processors: The Single Thread Version

Hung-Wei Tseng

Summary: Characteristics of modern processor architectures

- Multiple-issue pipelines with multiple functional units available
  - Multiple ALUs
  - Multiple Load/store units
  - Dynamic OoO scheduling to reorder instructions whenever possible
- Cache — very high hit rate if your code has good locality
  - Very matured data/instruction prefetcher
- Branch predictors — very high accuracy if your code is predictable
  - Perceptron
  - Tournament predictors

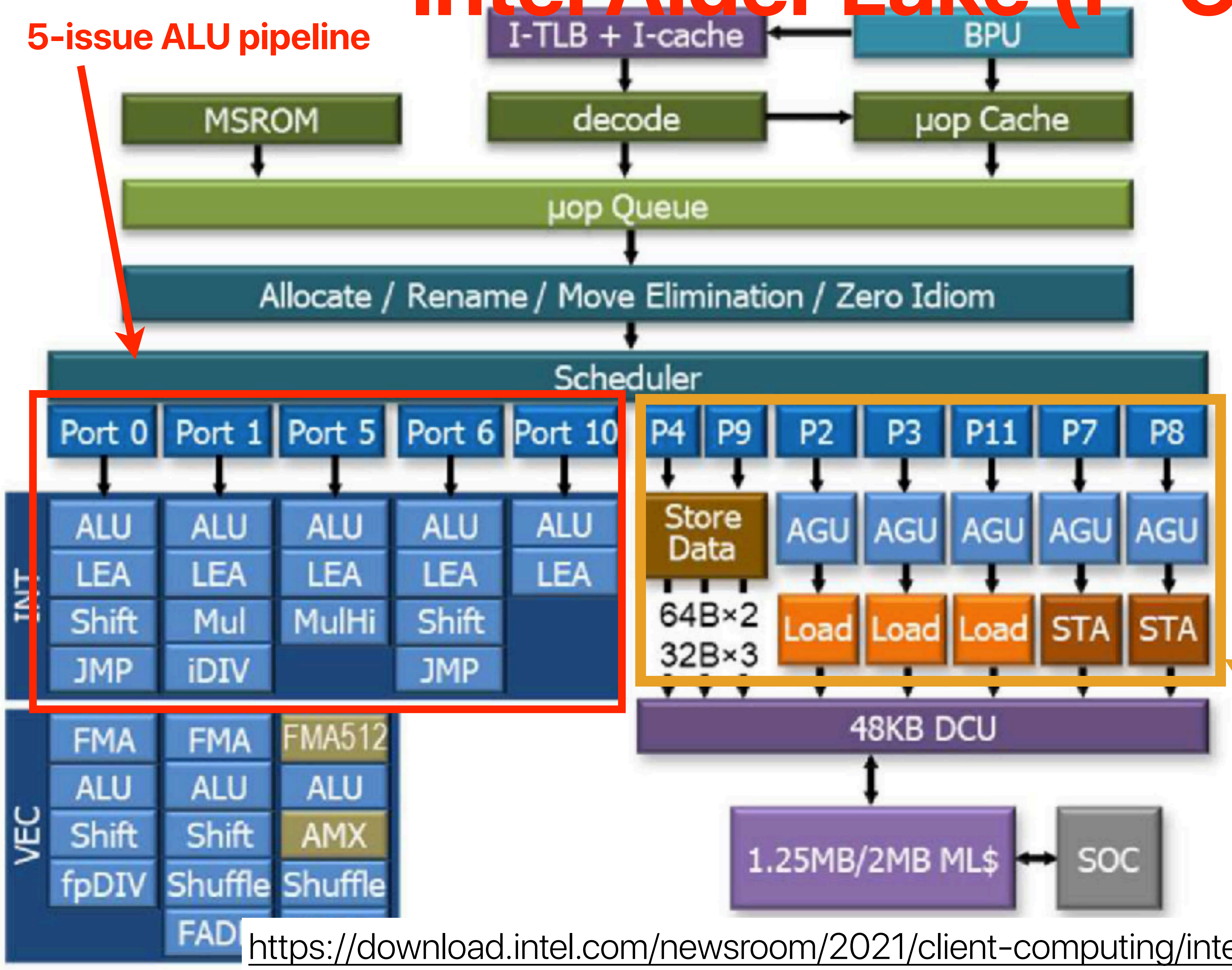
Intel Alder Lake (P-Core)

$$MinCPI = \frac{1}{12}$$

$$MinINTInst.CPI = \frac{1}{5}$$

$$MinMEMInst.CPI = \frac{1}{7}$$

$$MinBRInst.CPI = \frac{1}{2}$$



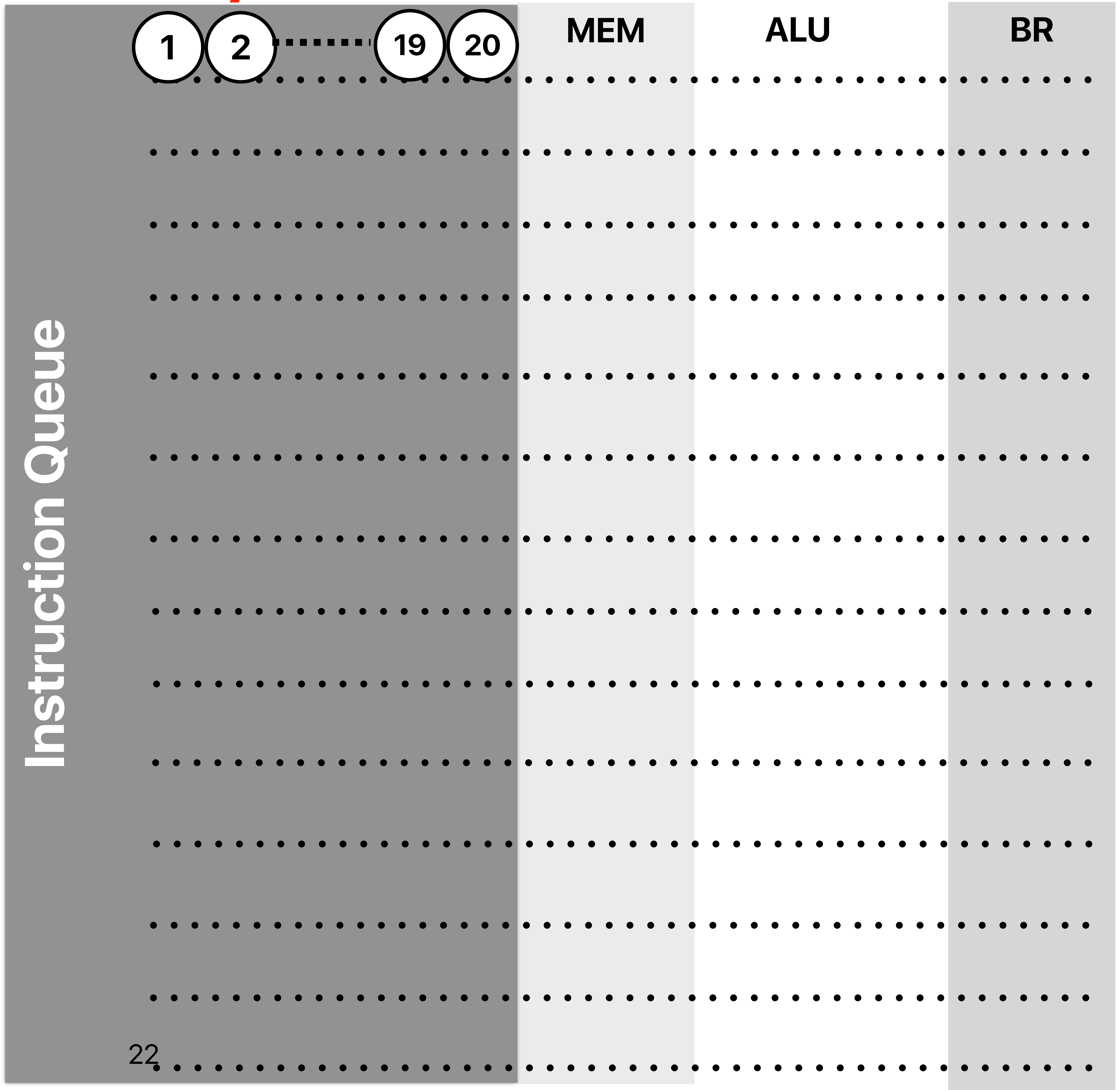
https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

Outline

- Programming on modern processors — exploiting instruction-level parallelism
- Simultaneous multithreading

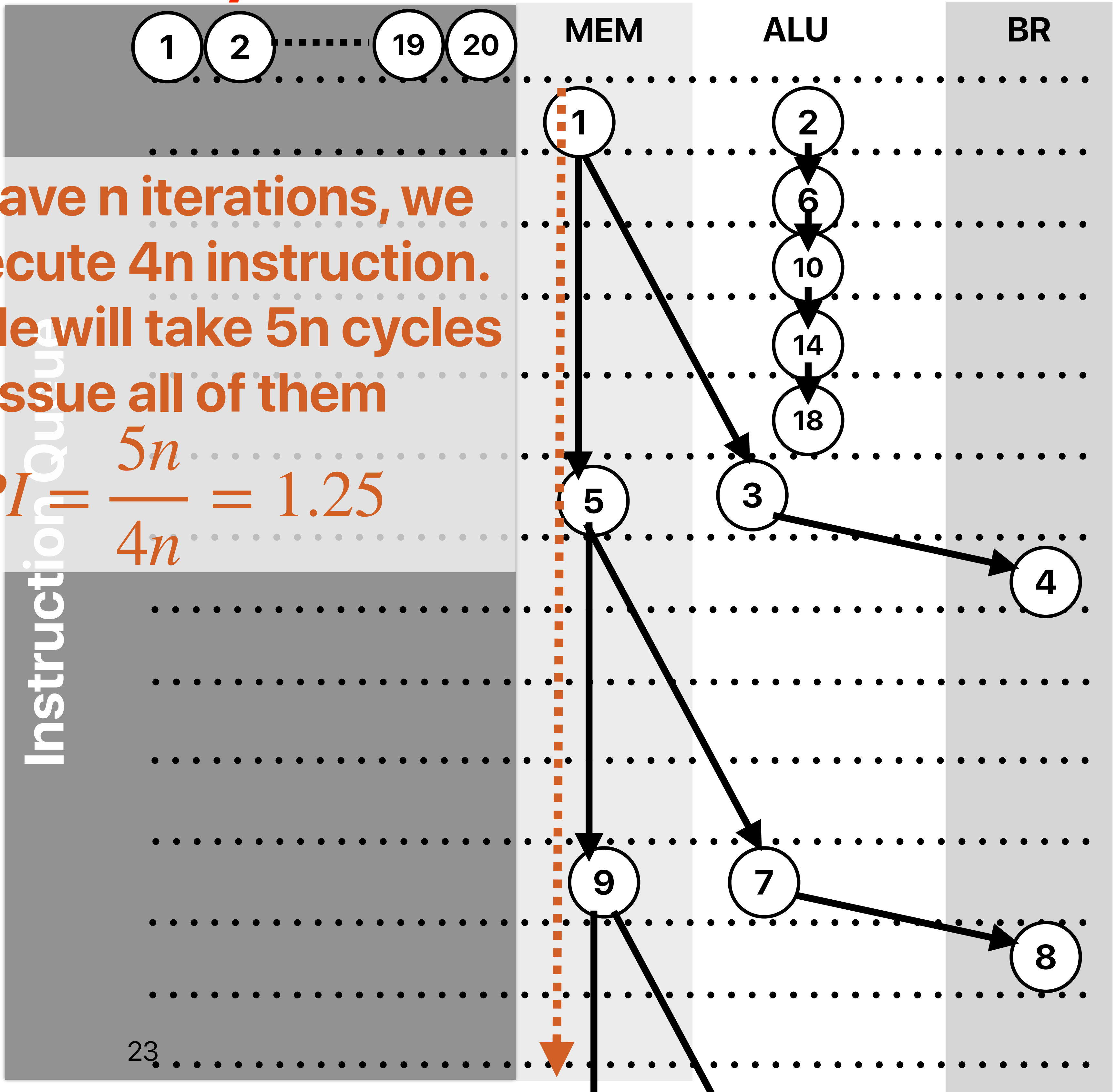
What if we have “unlimited” fetch/issue width — “linked list”

```
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
```



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

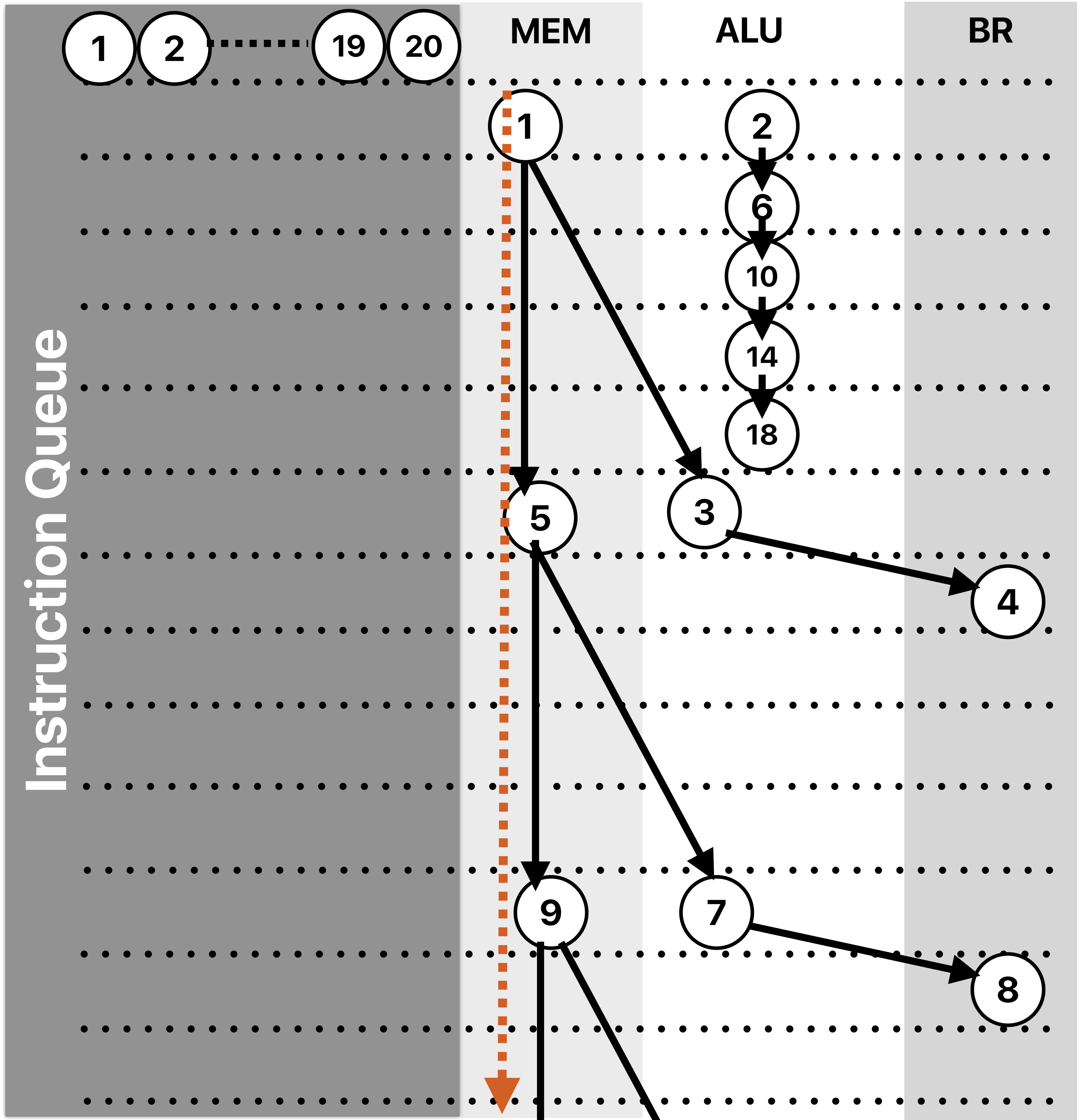
$$CPI = \frac{5n}{4n} = 1.25$$

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

If we cannot improve the performance of executing
`movq 8(%rdi), %rdi`
we cannot improve the execution time.
That's the "critical path"!

```
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
```

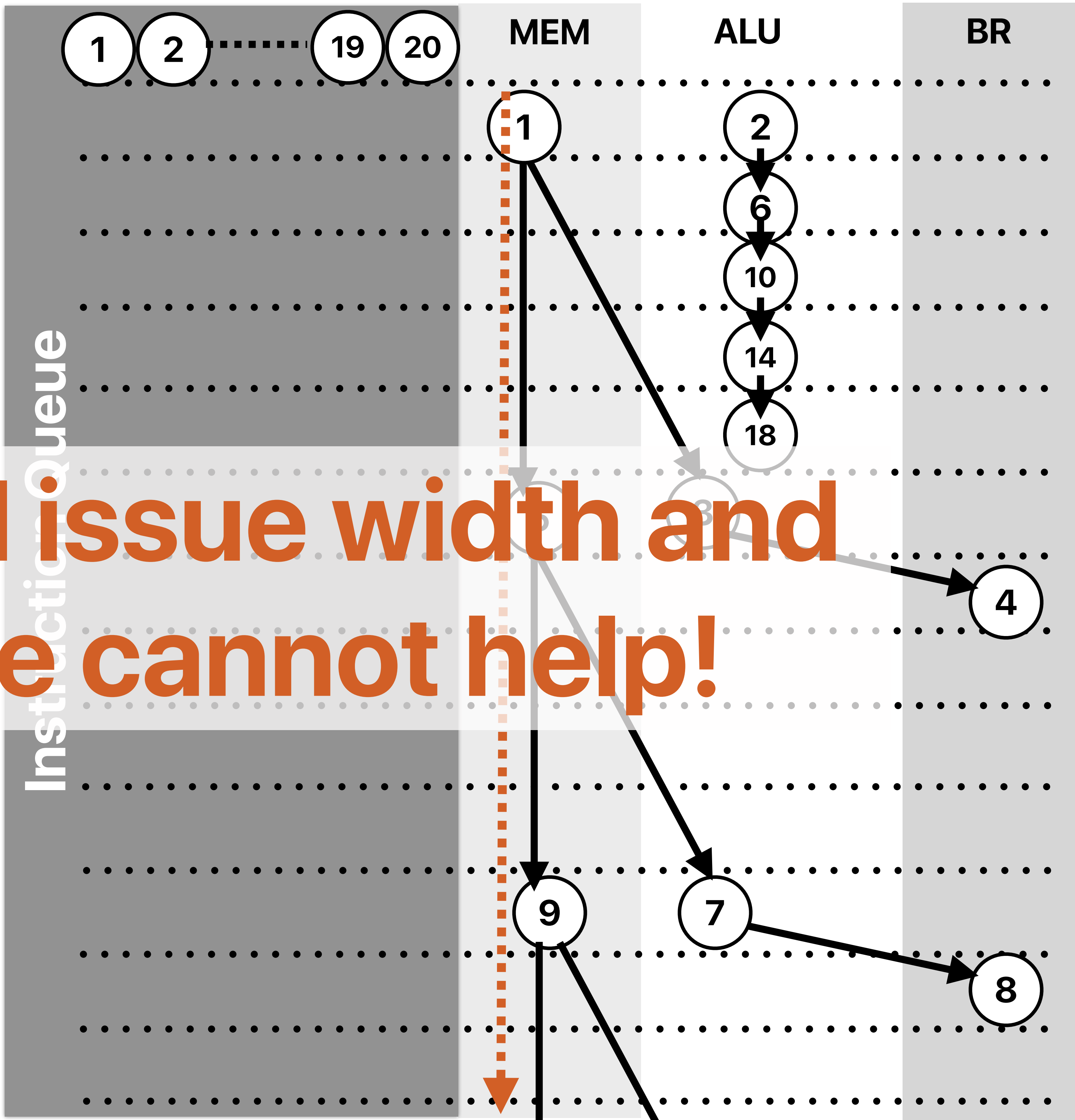


What if we have "unlimited" fetch/issue width — "linked list"

If we cannot improve the performance of executing
`movq 8(%rdi), %rdi`
we cannot improve the execution time.
That's the "critical path"!

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

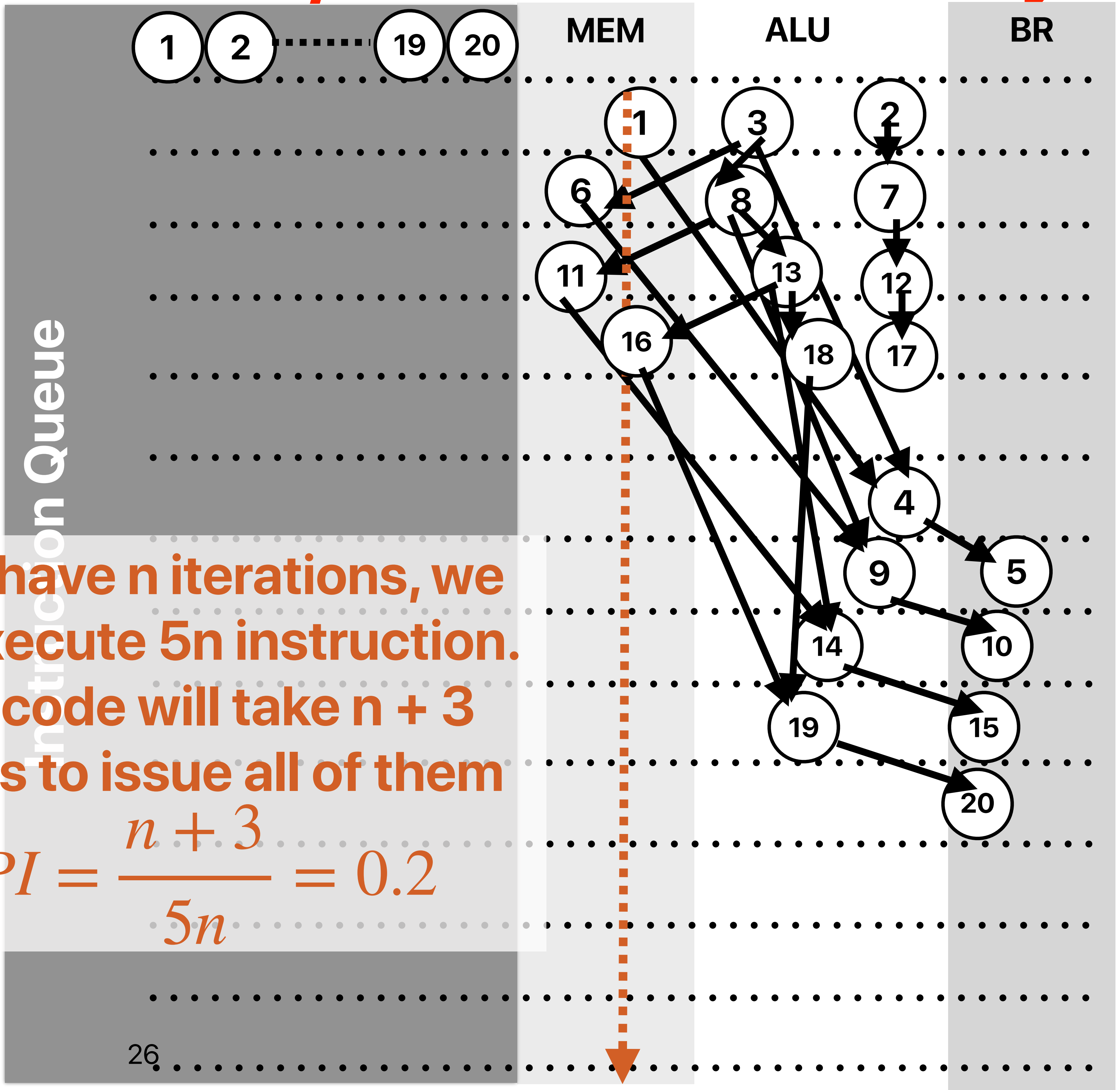
```
① .L3:    movq    8(%rdi), %rdi
②         addl    $1, %eax
③         testq   %rdi, %rdi
④         jne     .L3
```



Even unlimited issue width and a perfect cache cannot help!

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 + 3 cycles to issue all of them

$$CPI = \frac{n+3}{5n} = 0.2$$

Linked-list is never an ideal option

CPI is a lot higher even with lower IC, perfect cache and branch predictors

size	list	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate
1024	array	514259657	106295528	0.206696	0.197243	0.020966	0.000014
1024	list	411795196	515812542	1.252595	0.196544	0.101380	0.000071

size	list	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate
4096	array	205080322	41547848	0.202593	0.196833	0.008178	0.250965
4096	list	164474202	356755154	2.169065	0.196561	0.070124	0.334309
8192	array	409931842	82621771	0.201550	0.196462	0.016232	0.250467
8192	list	329389168	1048885357	3.184335	0.196532	0.206140	0.701870

$ miss rate dominates the performance despite lower ICs than using arrays

Takeaways: programming modern processors

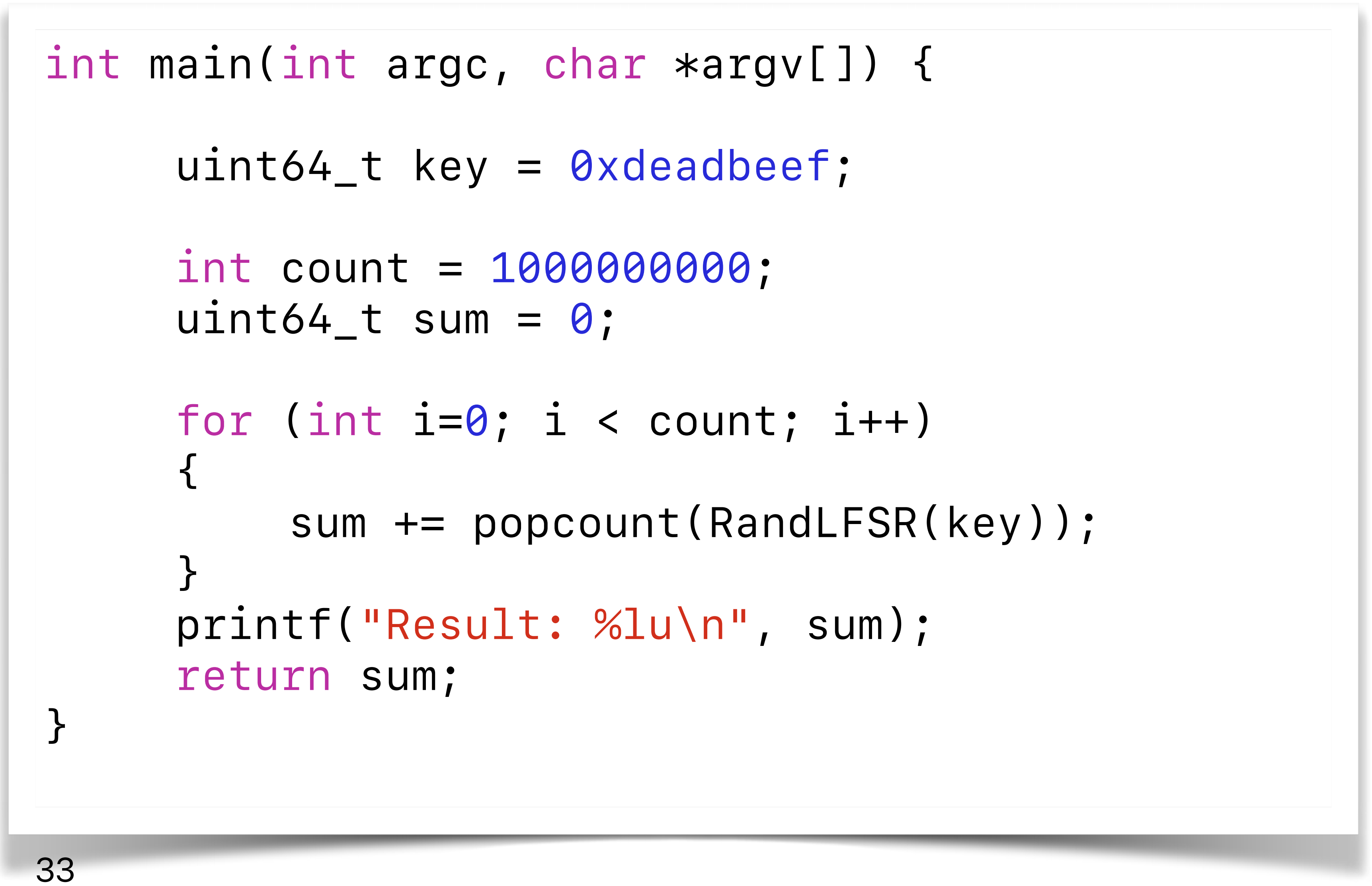
- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible

Problem: Popcount

- The population count (or popcount) of a specific value is the number of set bits (i.e., bits in 1s) in that value.
- Applications
  - Parity bits in error correction/detection code
  - Cryptography
  - Sparse matrix
  - Molecular Fingerprinting
  - Implementation of some succinct data structures like bit vectors and wavelet trees.

Problem: Popcount

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  `Input:   59487`
  `Output:  9`
  Explanation: 59487's binary representation is
  `0b1011001010001111`



```
int main(int argc, char *argv[]) {

    uint64_t key = 0xdeadbeef;

    int count = 1000000000;
    uint64_t sum = 0;

    for (int i=0; i < count; i++)
    {
        sum += popcount(RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```

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

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;
}
```

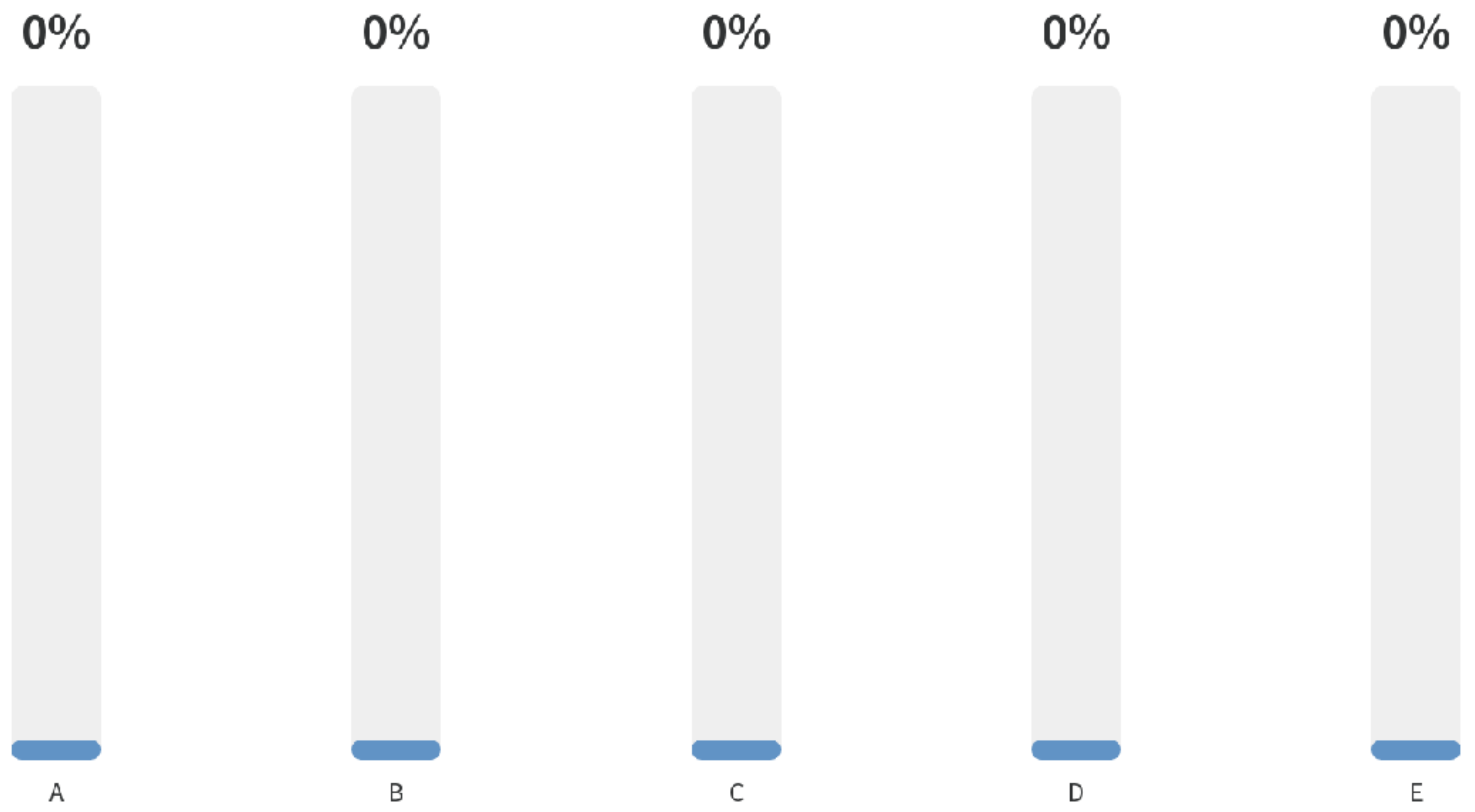
D

```
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;
}
```

0



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

Five implementations

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

A

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

B

```c
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

```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;
}
```

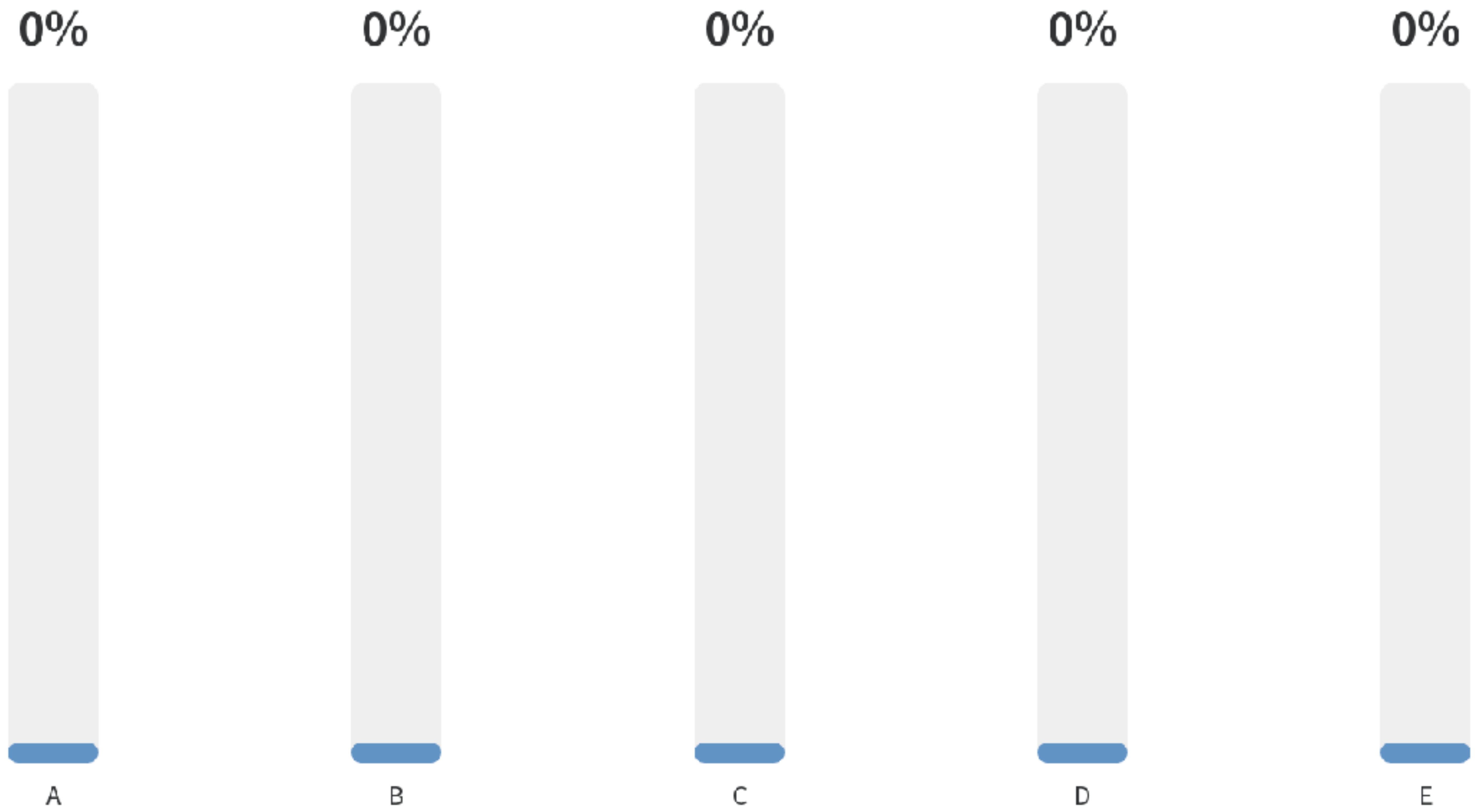
D

```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};
     for (uint64_t i = 0; i < 16; i++)
     {
          c += table[(x & 0xF)];
          x = x >> 4;
     }
     return c;
  }
```

E

```c
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;
}
```

0



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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;
}
```

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

Why is B better than A?

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

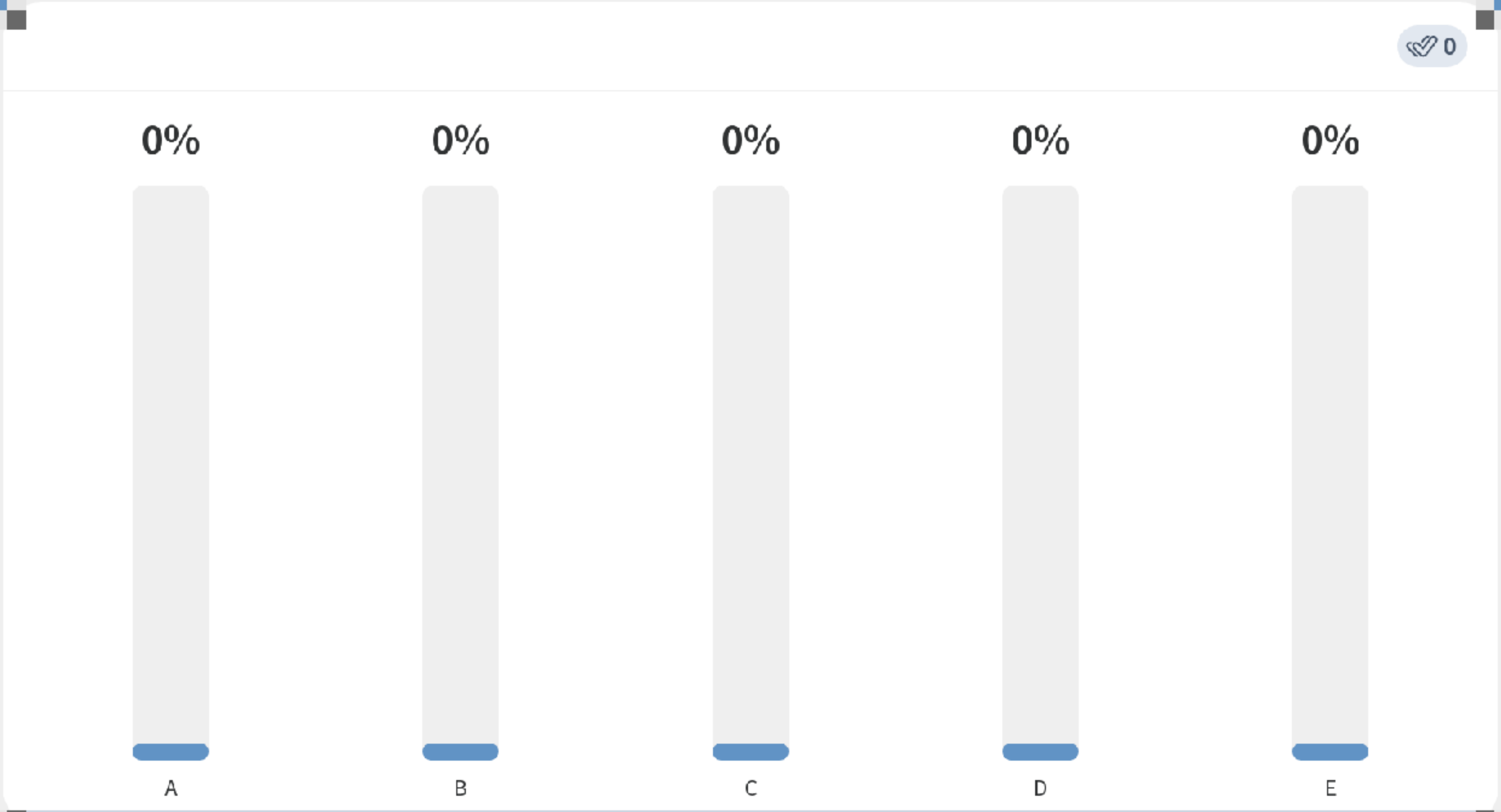
A. 0
B. 1
C. 2
D. 3
E. 4

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;
}
```

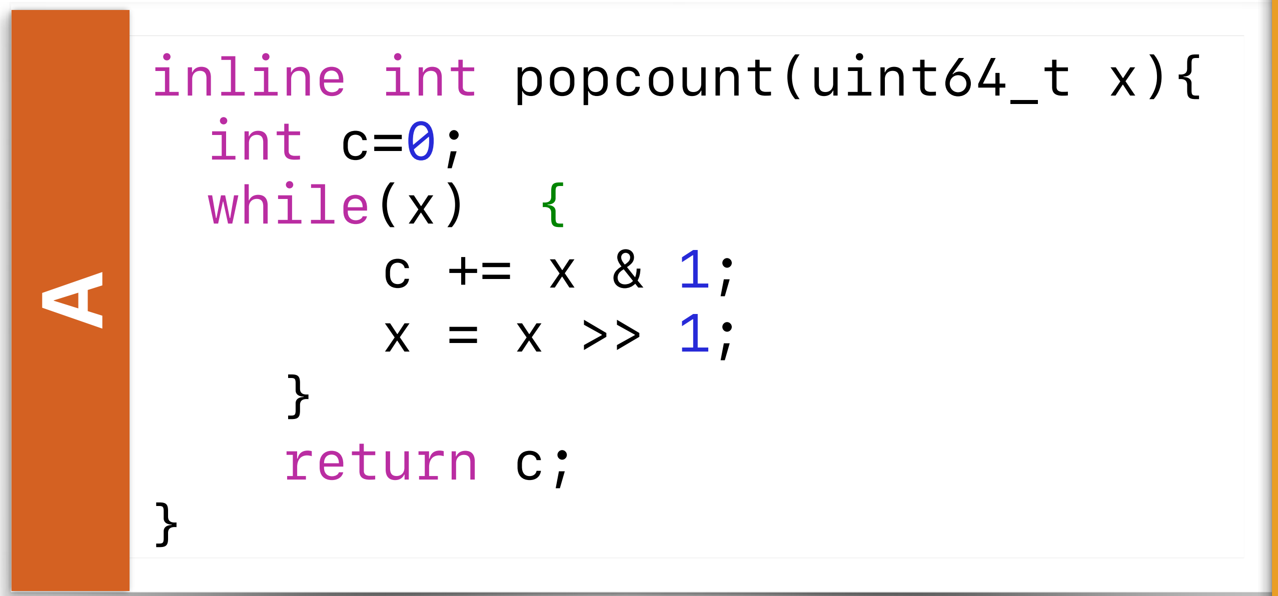

0
0%
0%
0%
0%
0%
A
B
C
D
E

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

Why is B better than A?

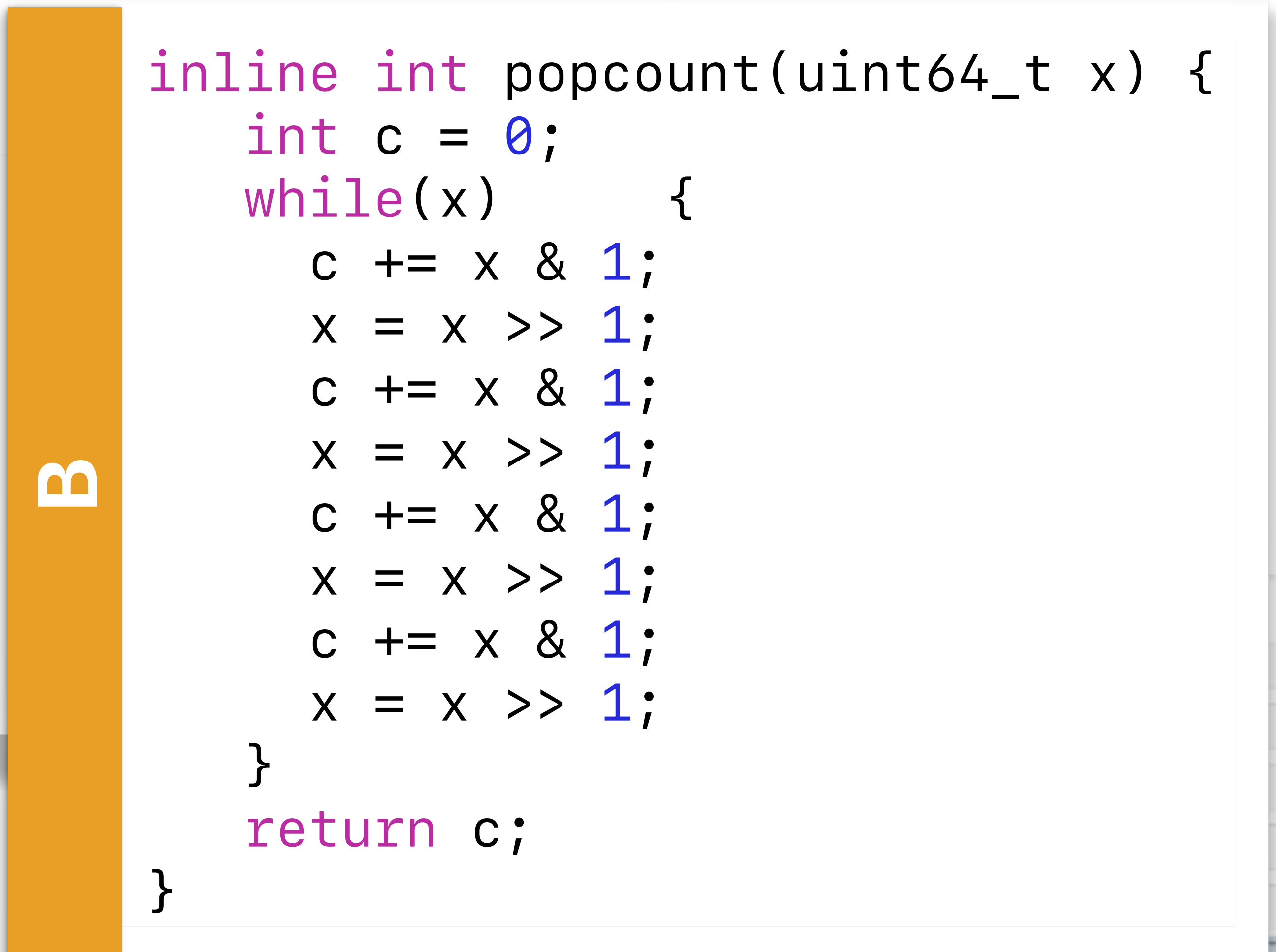
- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  1. B has lower dynamic instruction count than A
  2. B has significantly lower branch mis-prediction rate than A
  3. B has significantly fewer branch instructions than A
  4. B has better CPI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

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



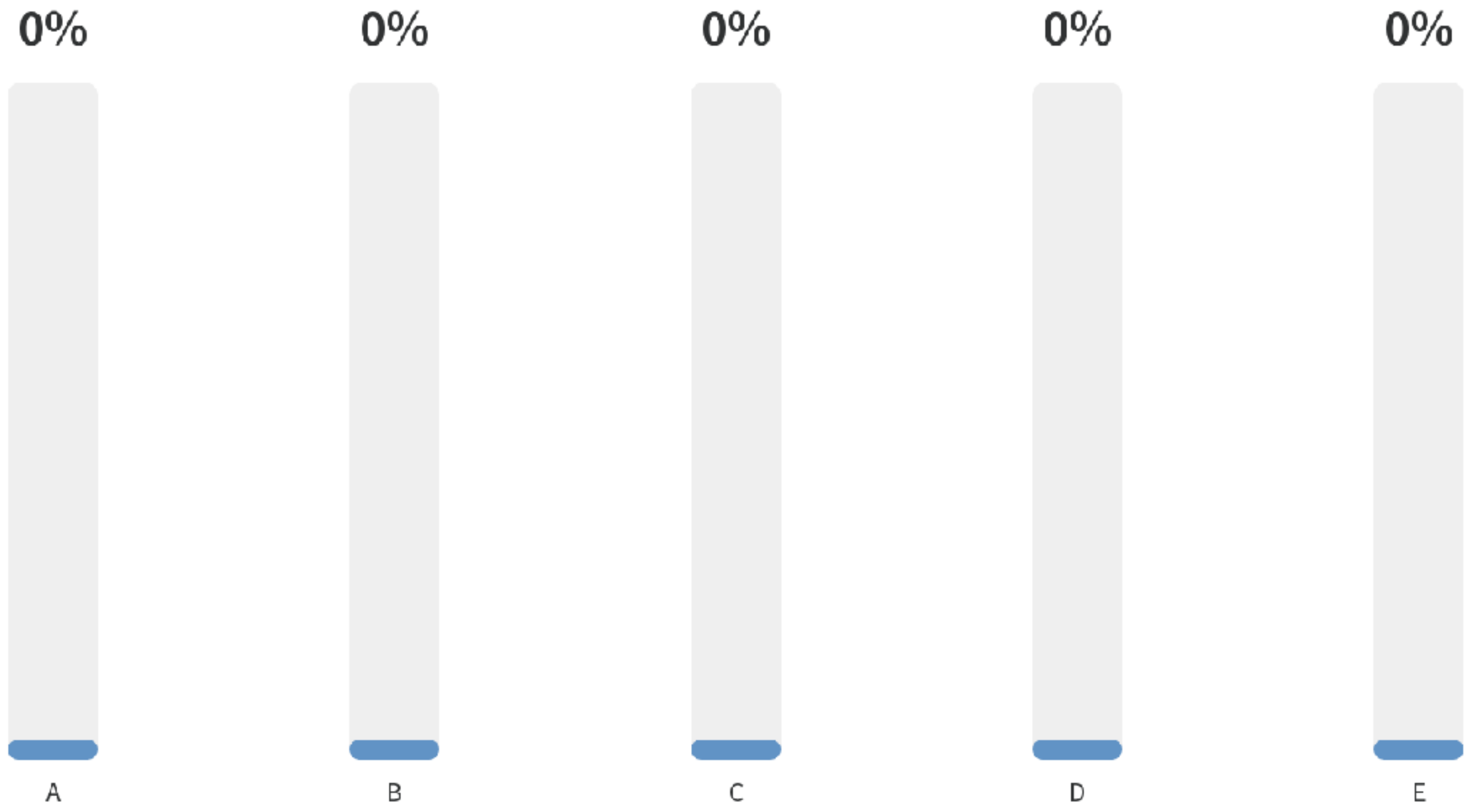
A

```
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;
}
```



0

Why is B better than A?

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

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

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;
}
```

Why is B better than A?

A

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

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

5*n instructions

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;
}
```

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

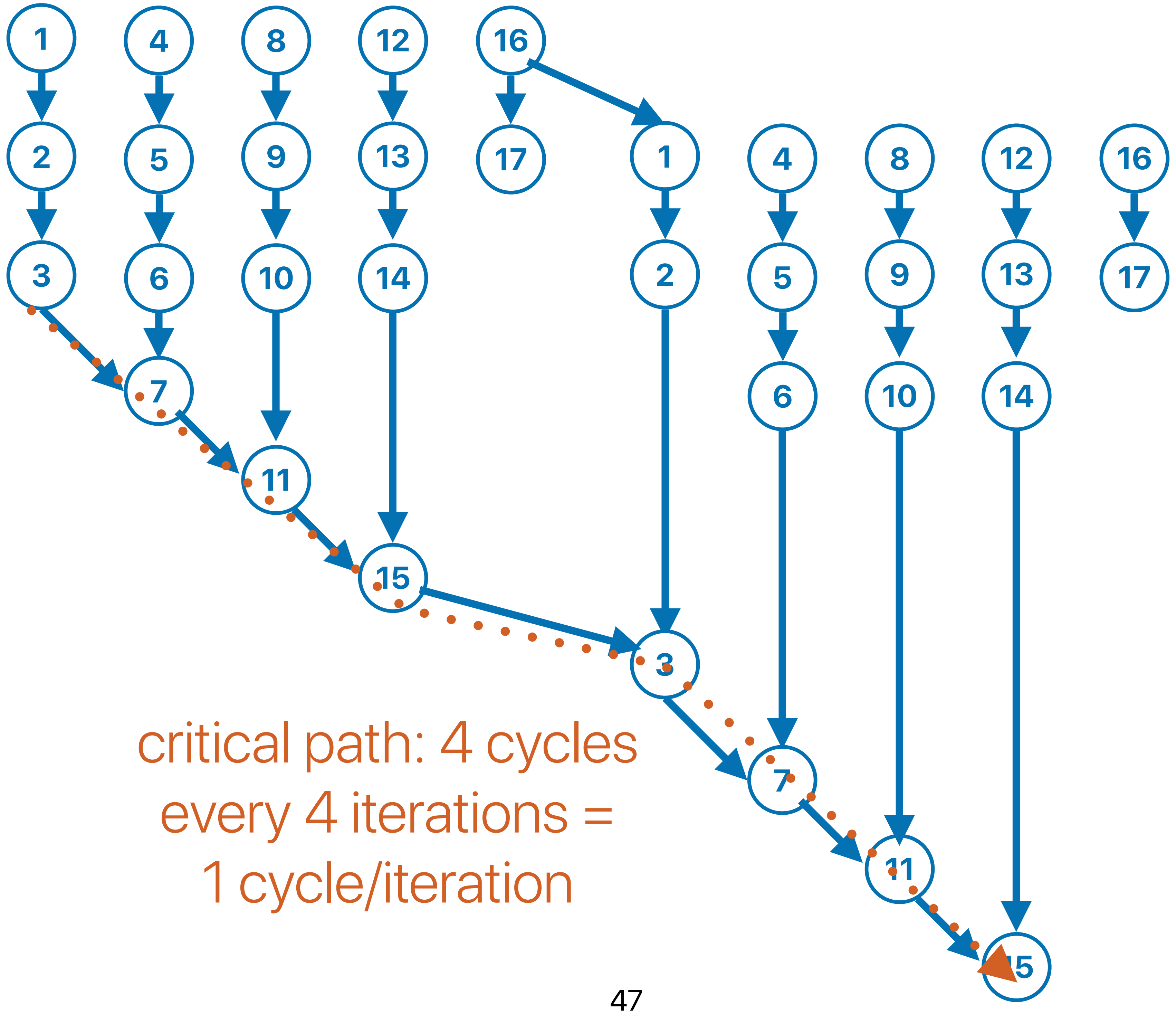
17*(n/4) = 4.25*n instructions

Only one branch for four iterations in A

Why is B better than A?

B

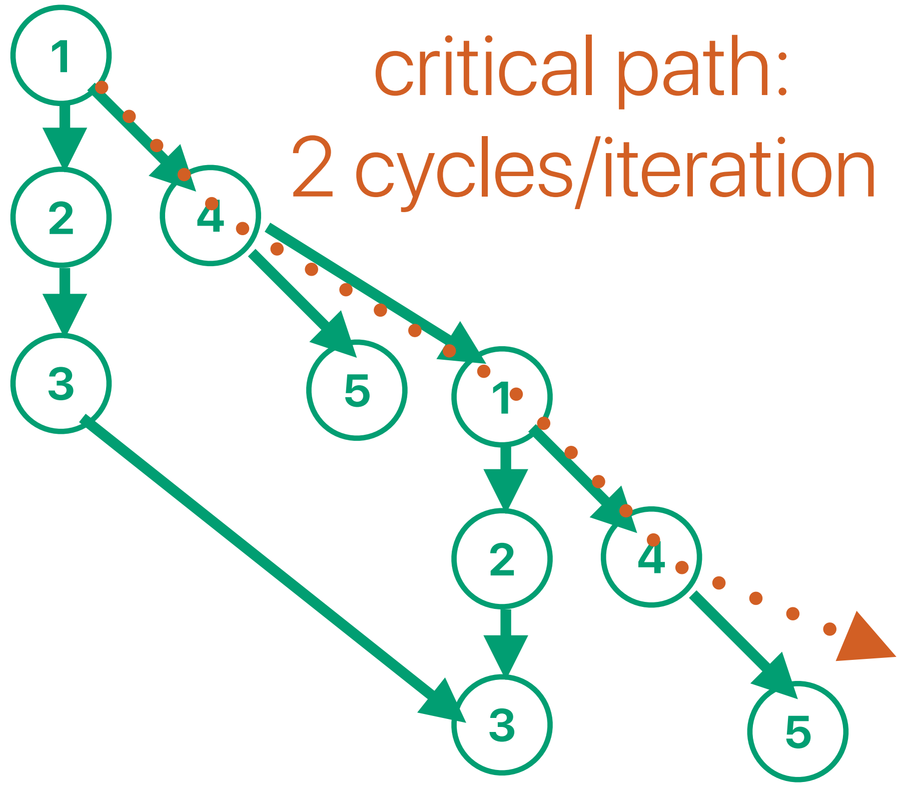
```
① movl     %ecx, %eax
② andl     $1, %eax
③ addl     %edx, %eax
④ movq     %rcx, %rdx
⑤ shrq     %rdx
⑥ andl     $1, %edx
⑦ addl     %eax, %edx
⑧ movq     %rcx, %rax
⑨ shrq     $2, %rax
⑩ andl     $1, %eax
⑪ addl     %edx, %eax
⑫ movq     %rcx, %rdx
⑬ shrq     $3, %rdx
⑭ andl     $1, %edx
⑮ addl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

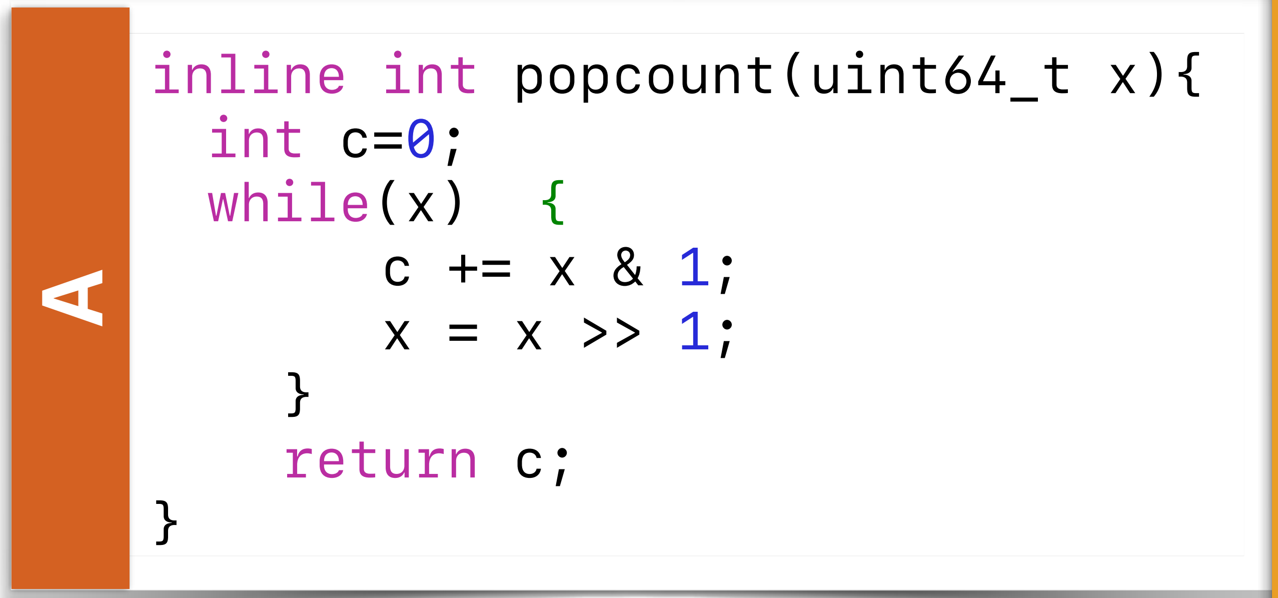


critical path: 2 cycles/iteration

Why is B better than A?

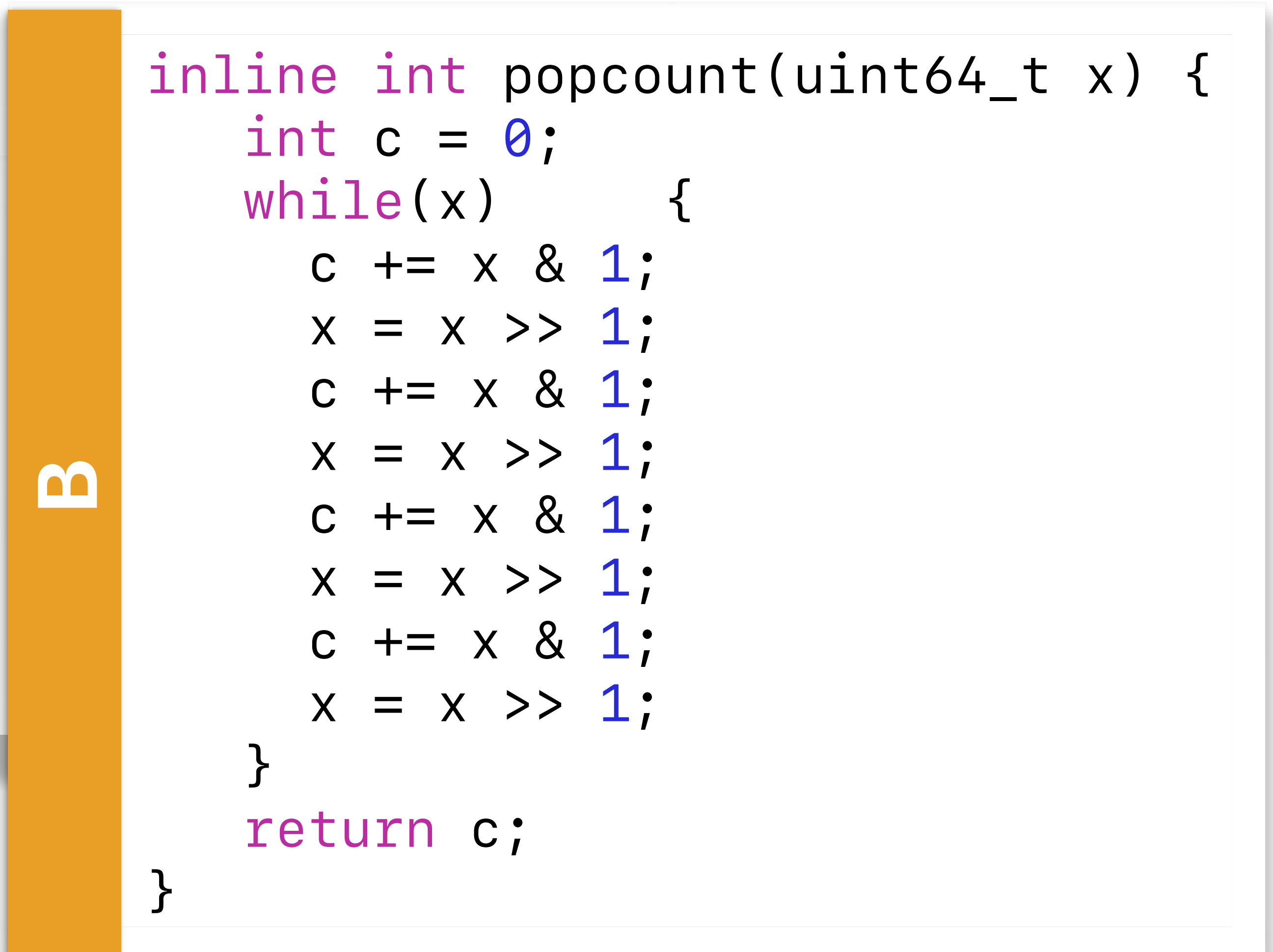
- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  - [✓ ①] B has lower dynamic instruction count than A
  - ② B has significantly lower branch mis-prediction rate than A
  - [✓ ③] B has significantly fewer branch instructions than A
  - [✓ ④]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

A. 0
B. 1
C. 2
[✓ D. 3]
E. 4



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;
}
```

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
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.

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

Why is C better than B?

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

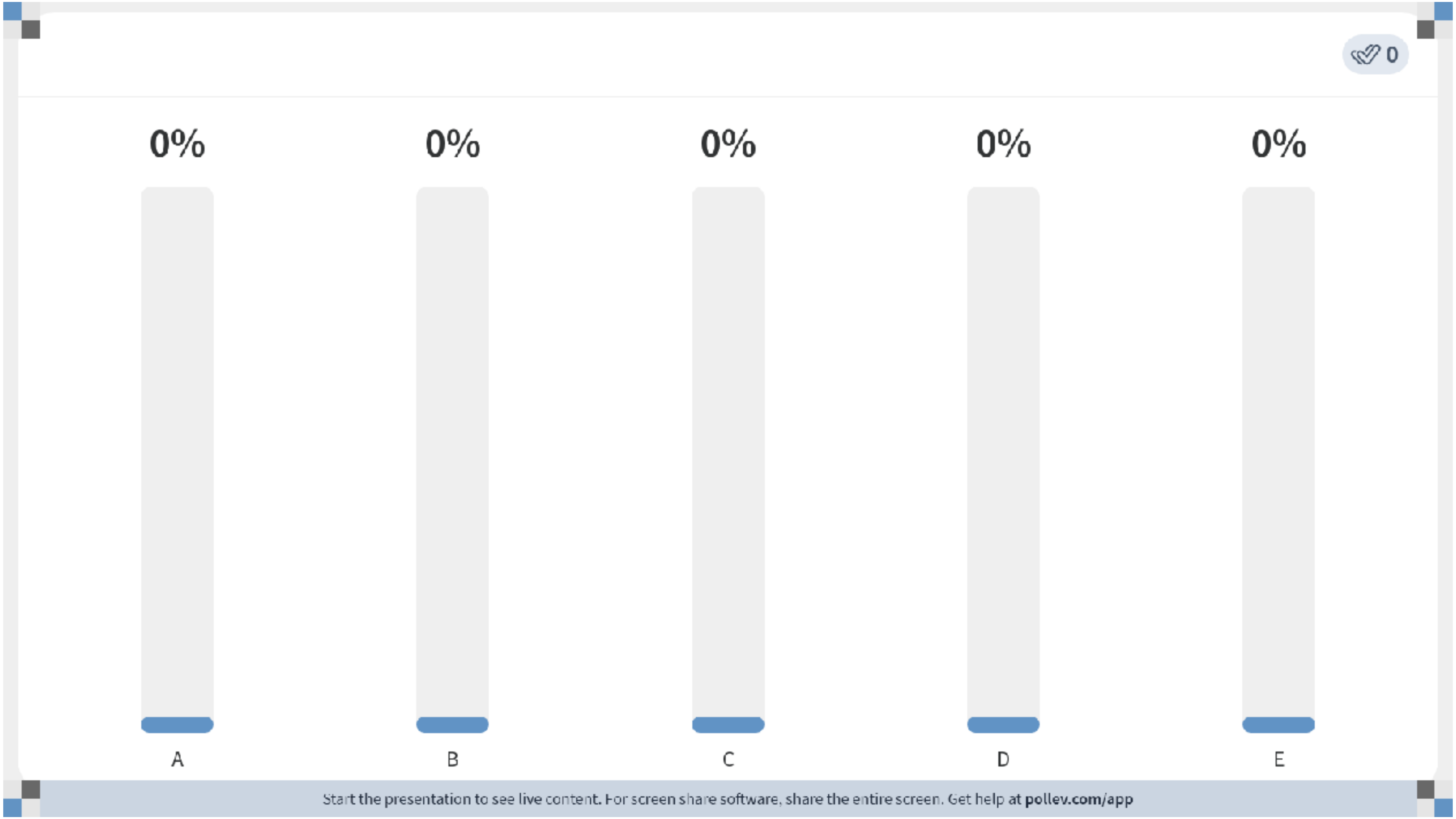
A. 0
B. 1
C. 2
D. 3
E. 4

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;
}
```

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;
}
```



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

Why is C better than B?

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

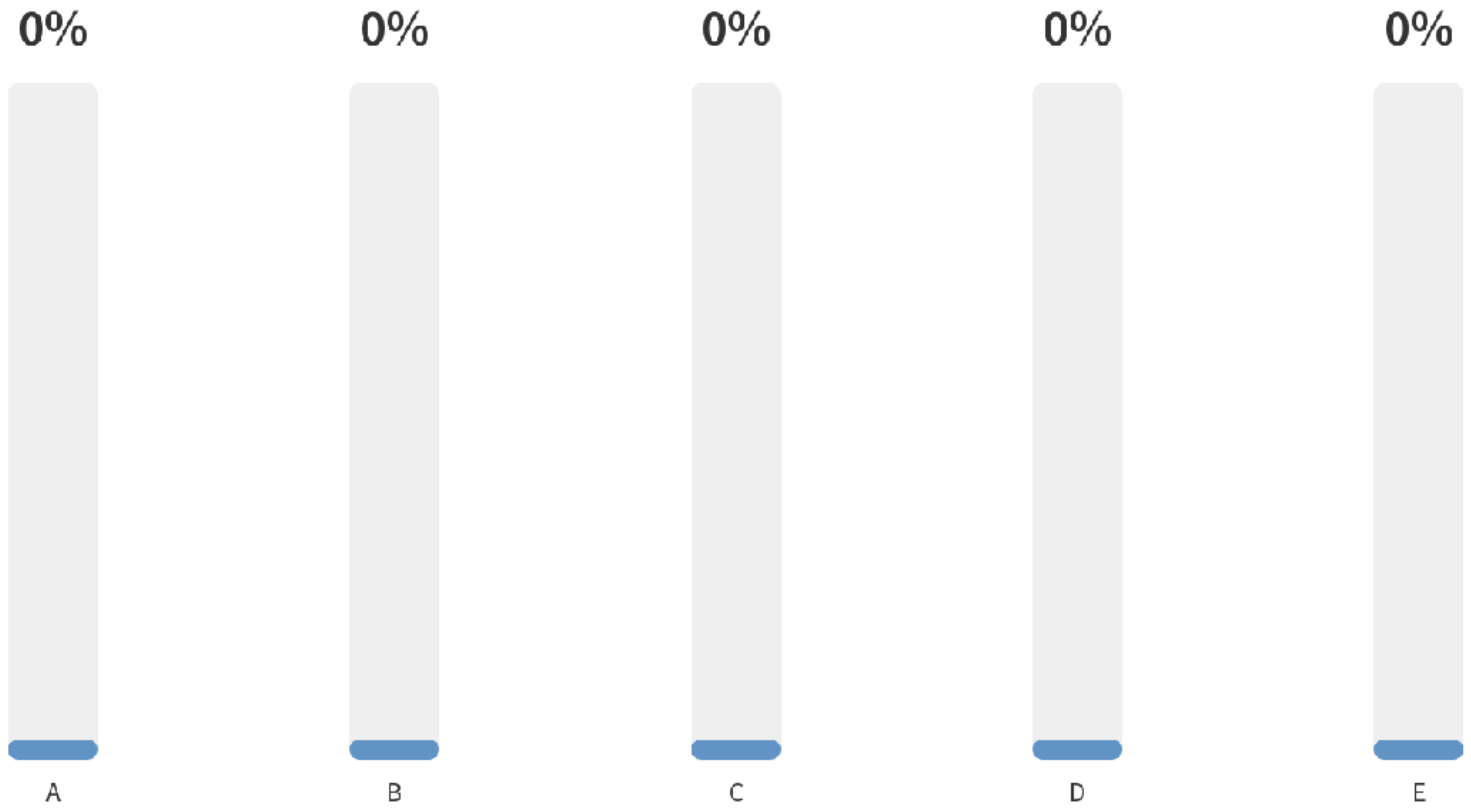
A. 0
B. 1
C. 2
D. 3
E. 4

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;
}
```

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;
}
```



0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Why is C better than B?

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

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;
}
```

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;
}
```

Why is C better than B?

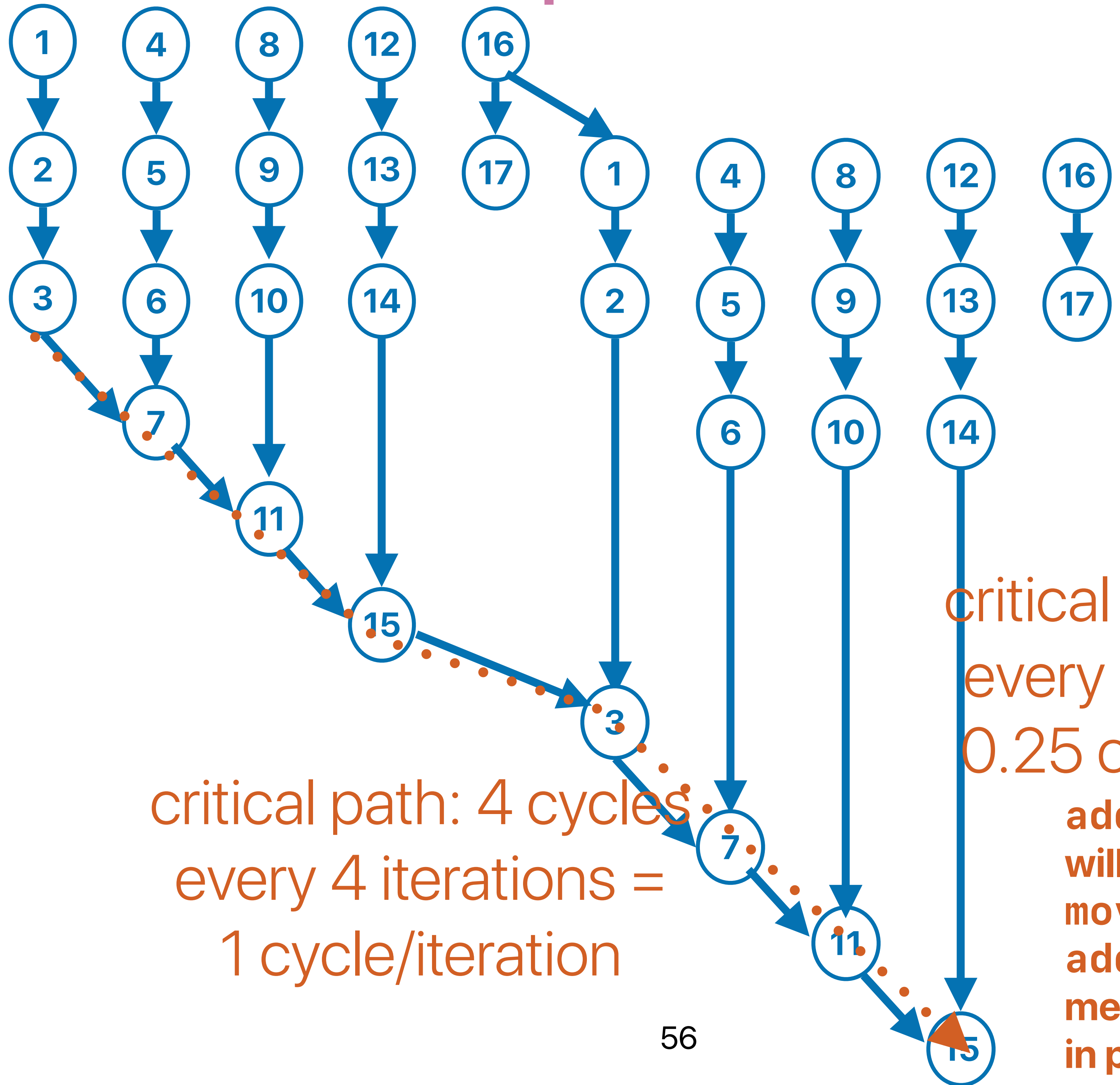
B

```
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
```

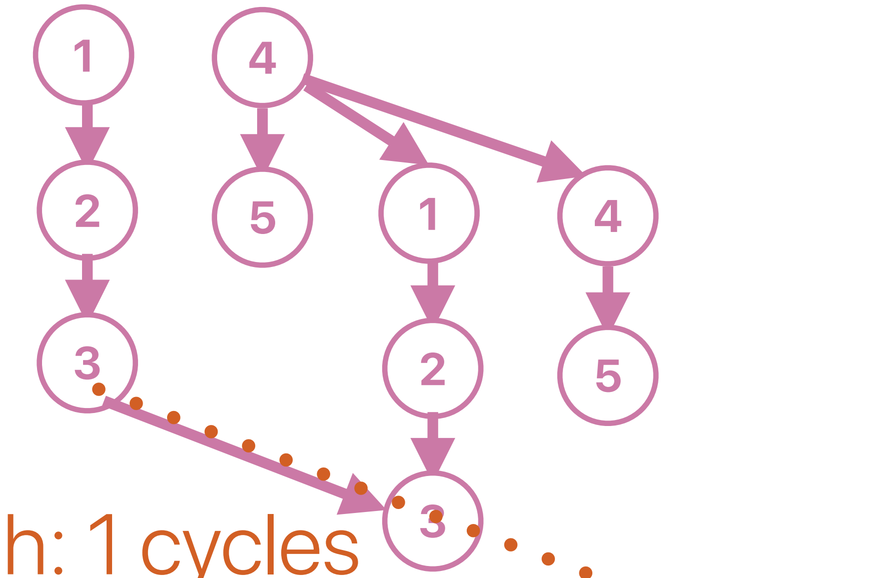
These 5 instructions represent 4 iterations!!!

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: 4 cycles every 4 iterations = 1 cycle/iteration



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

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - [✓ ①] C has lower dynamic instruction count than B — C only needs one load, one add, one shift, the same amount of iterations
  - ② C has significantly lower branch mis-prediction rate than B — the same number being predicted.
  - ③ C has significantly fewer branch instructions than B — the same amount of branches
  - ④ C has better CPI than B — Probably not. In fact, the load may have negative effect without architectural supports

  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;
}
```

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;
}
```

Takeaways: programming modern processors

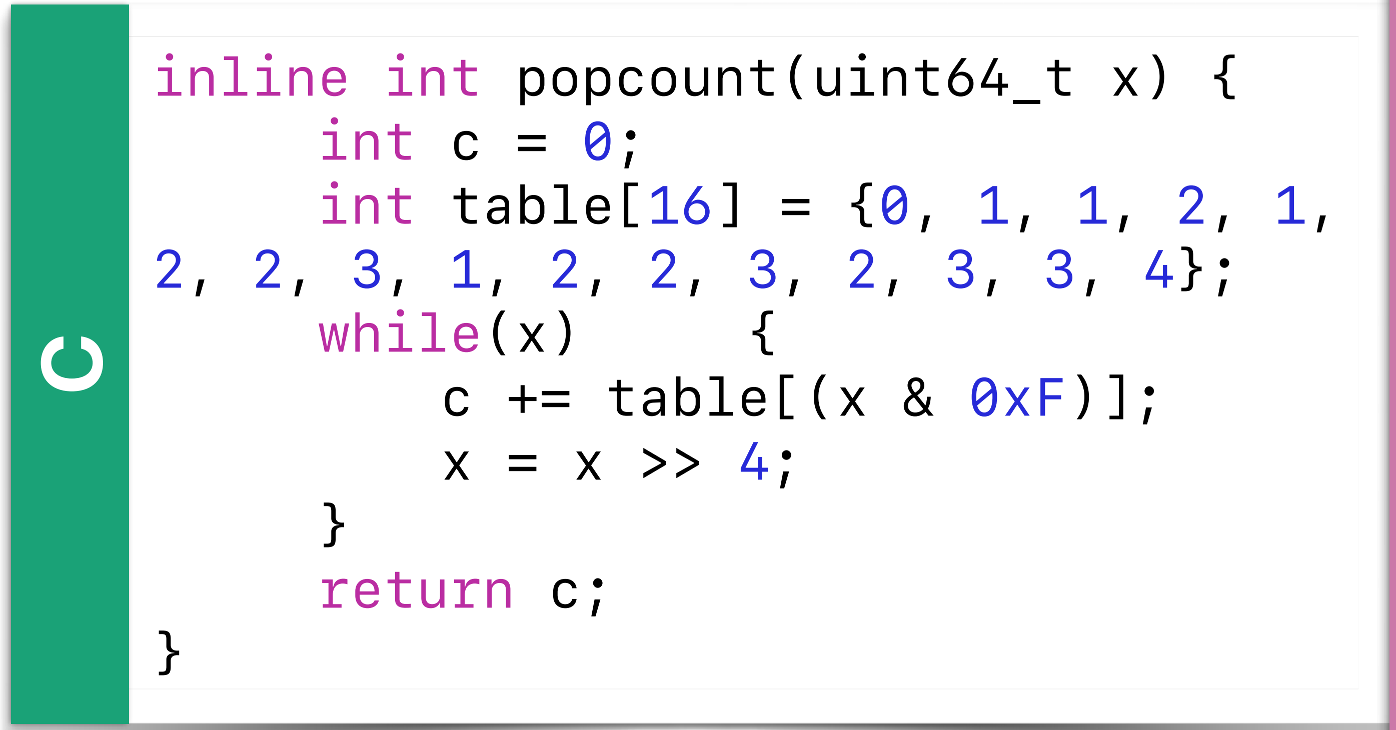
- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.
- With caches,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

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

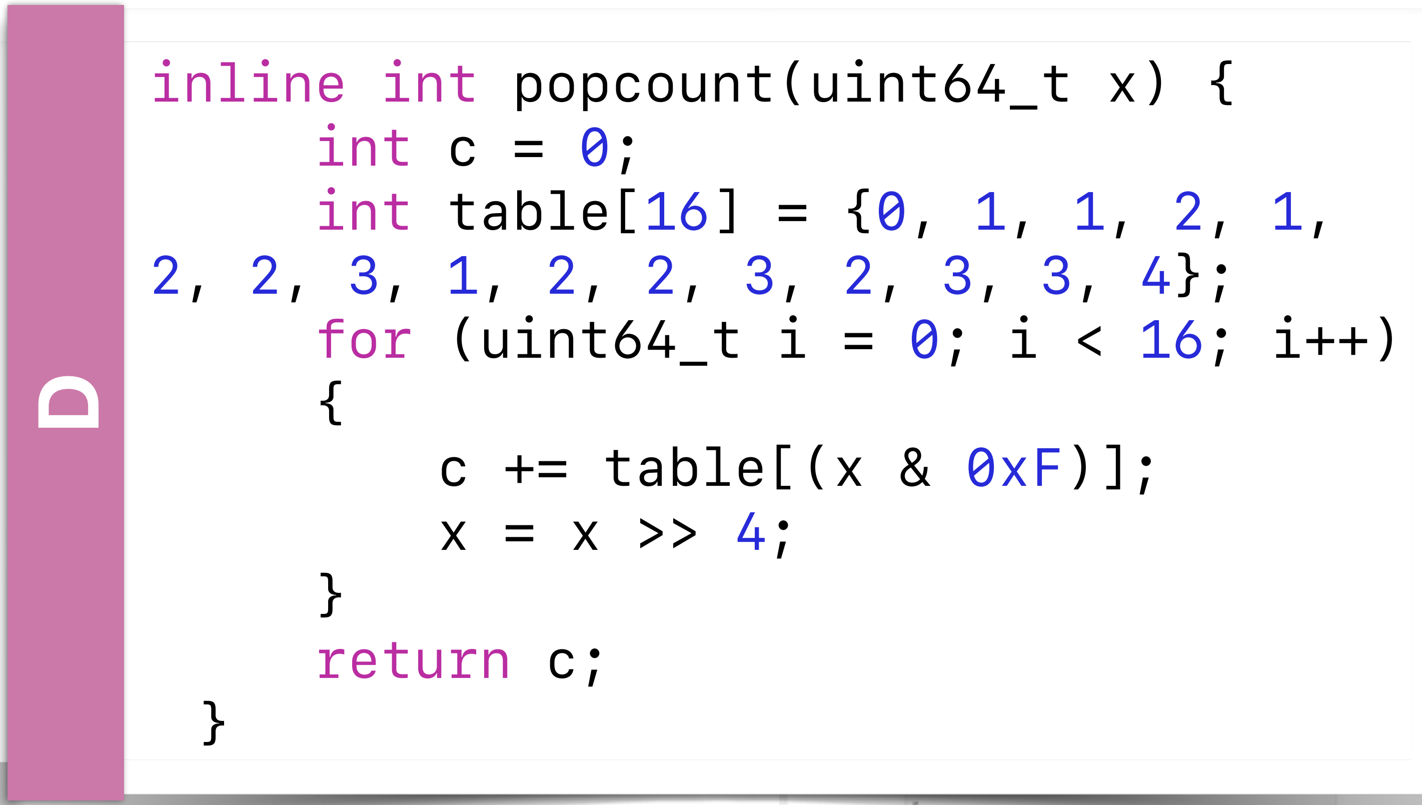
Why is D better than C?

- 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

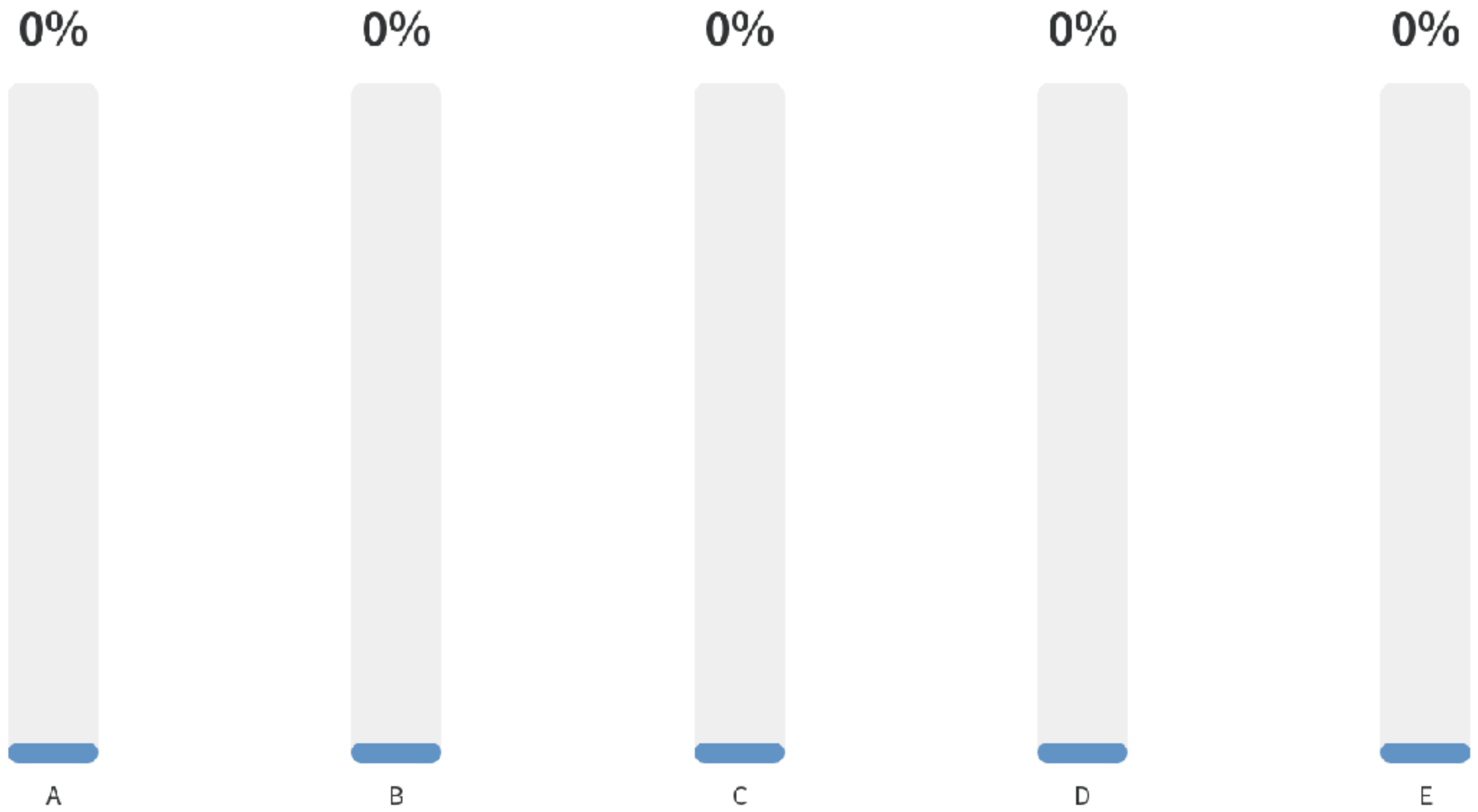

C

```
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;
 }
```

0



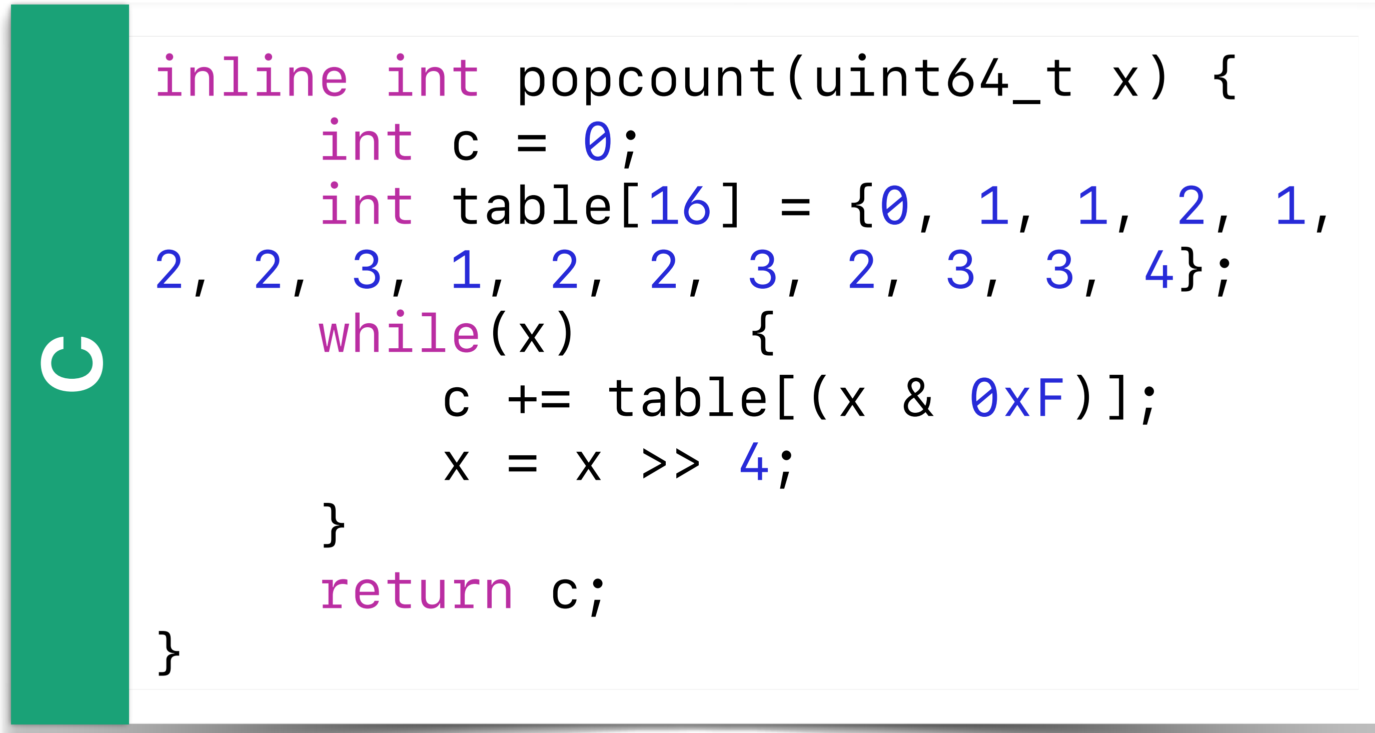
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

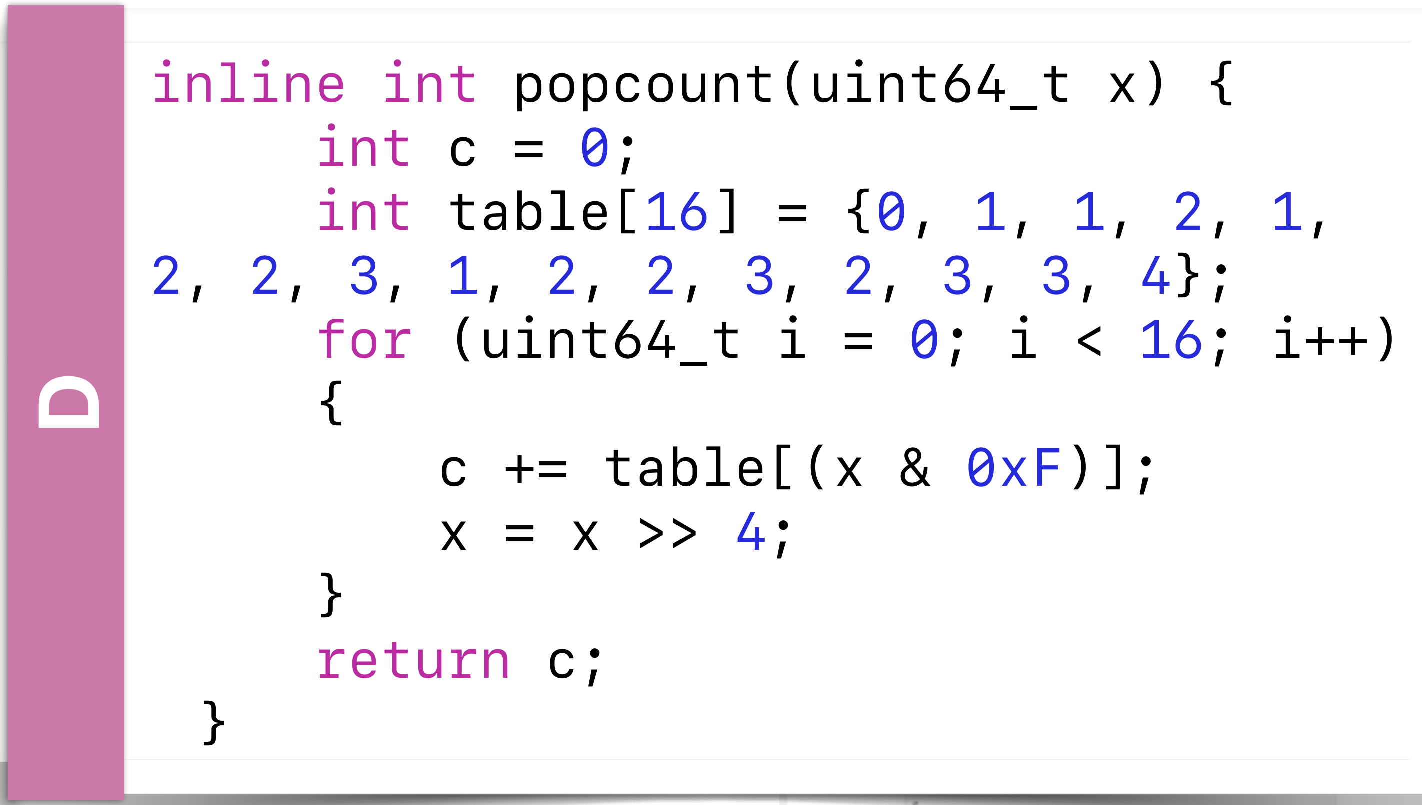
Why is D better than C?

- 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

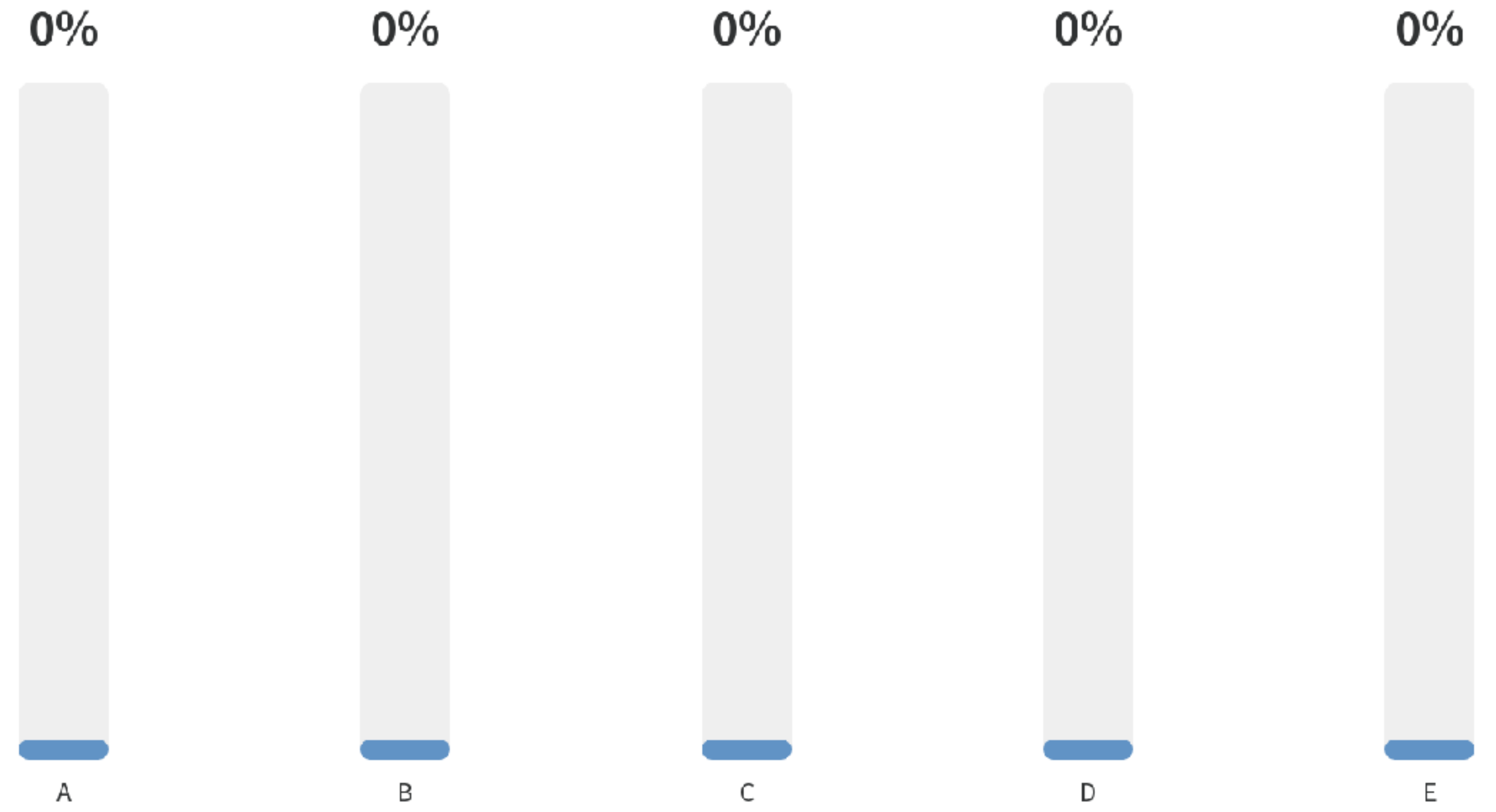
A. 0
B. 1
C. 2
D. 3
E. 4


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;
}
```


0
0%
0%
0%
0%
0%
A
B
C
D
E

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;
 }
```

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

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;
     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
- 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

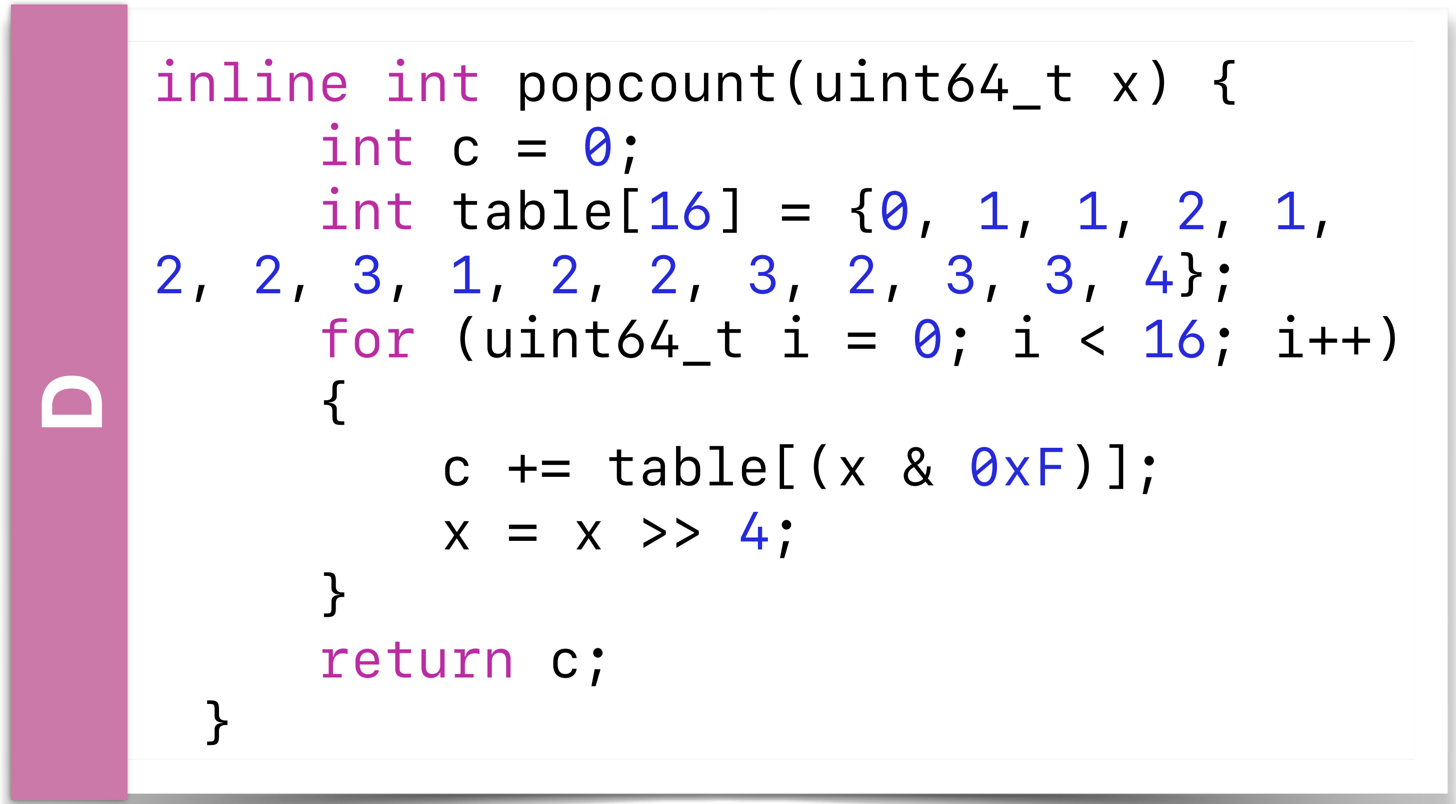
https://www.pollev.com/hungweitseng close in 1:30

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

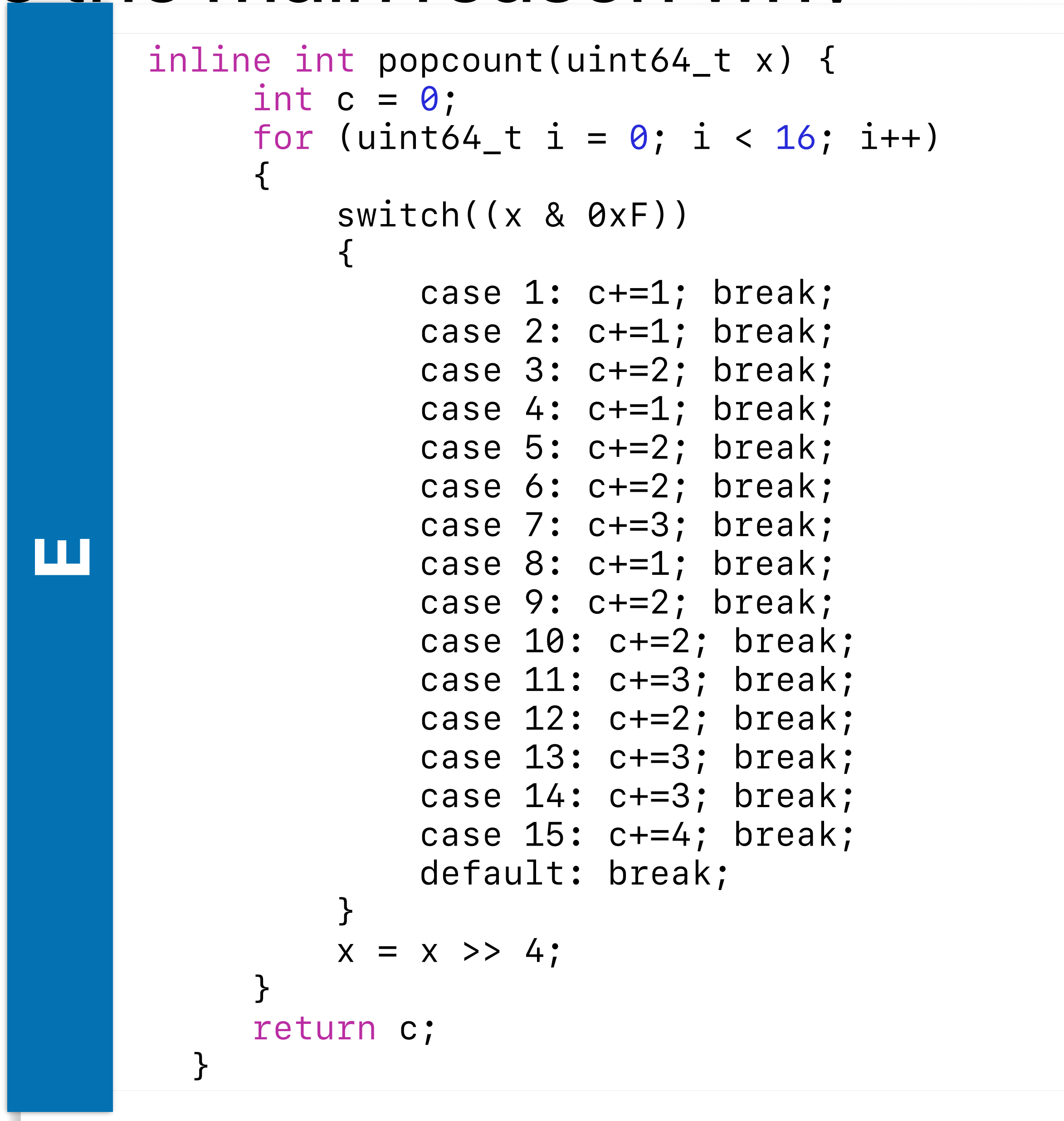
A. 0
B. 1
C. 2
D. 3
E. 4

D



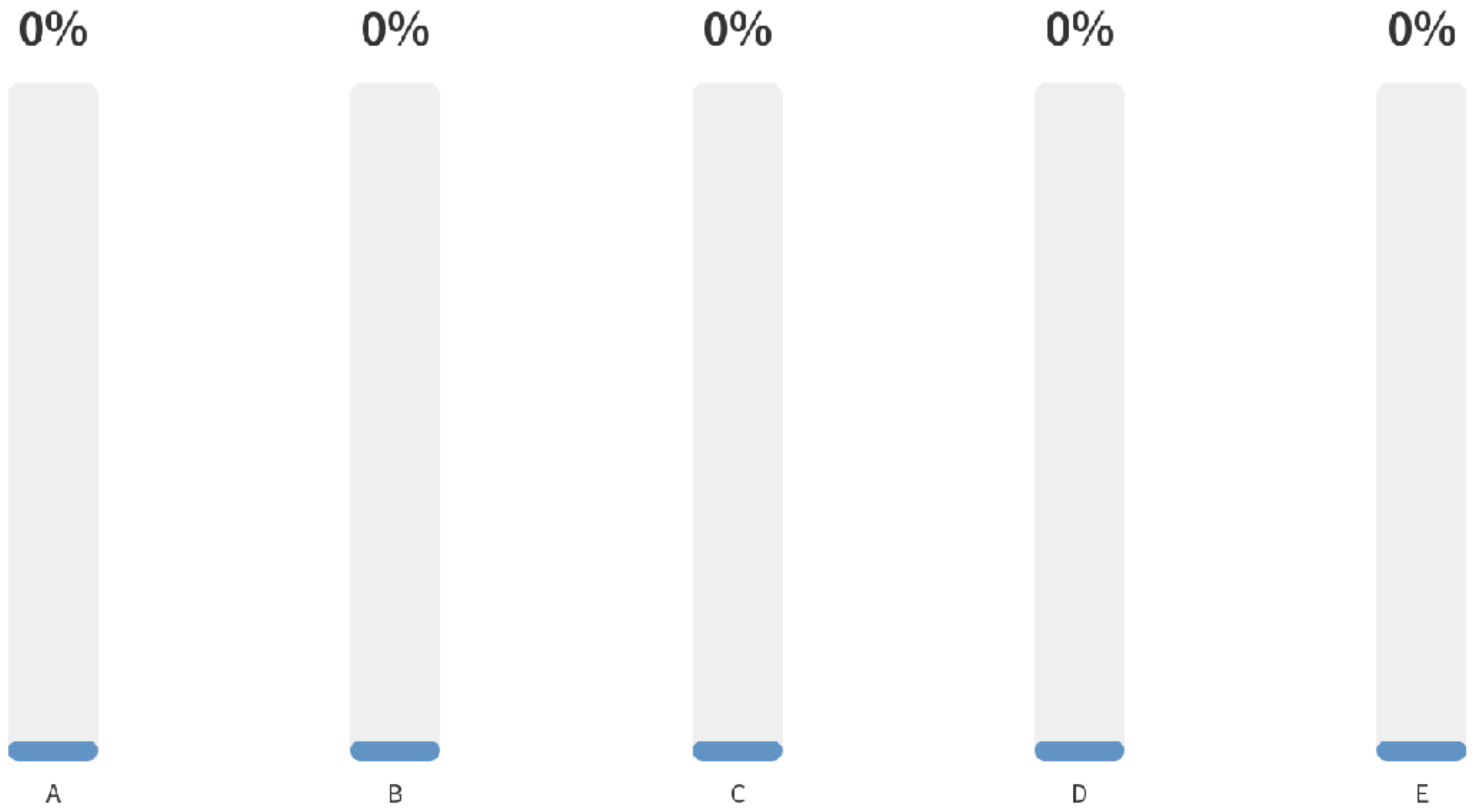
```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
  }
```

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;
 }
```

0



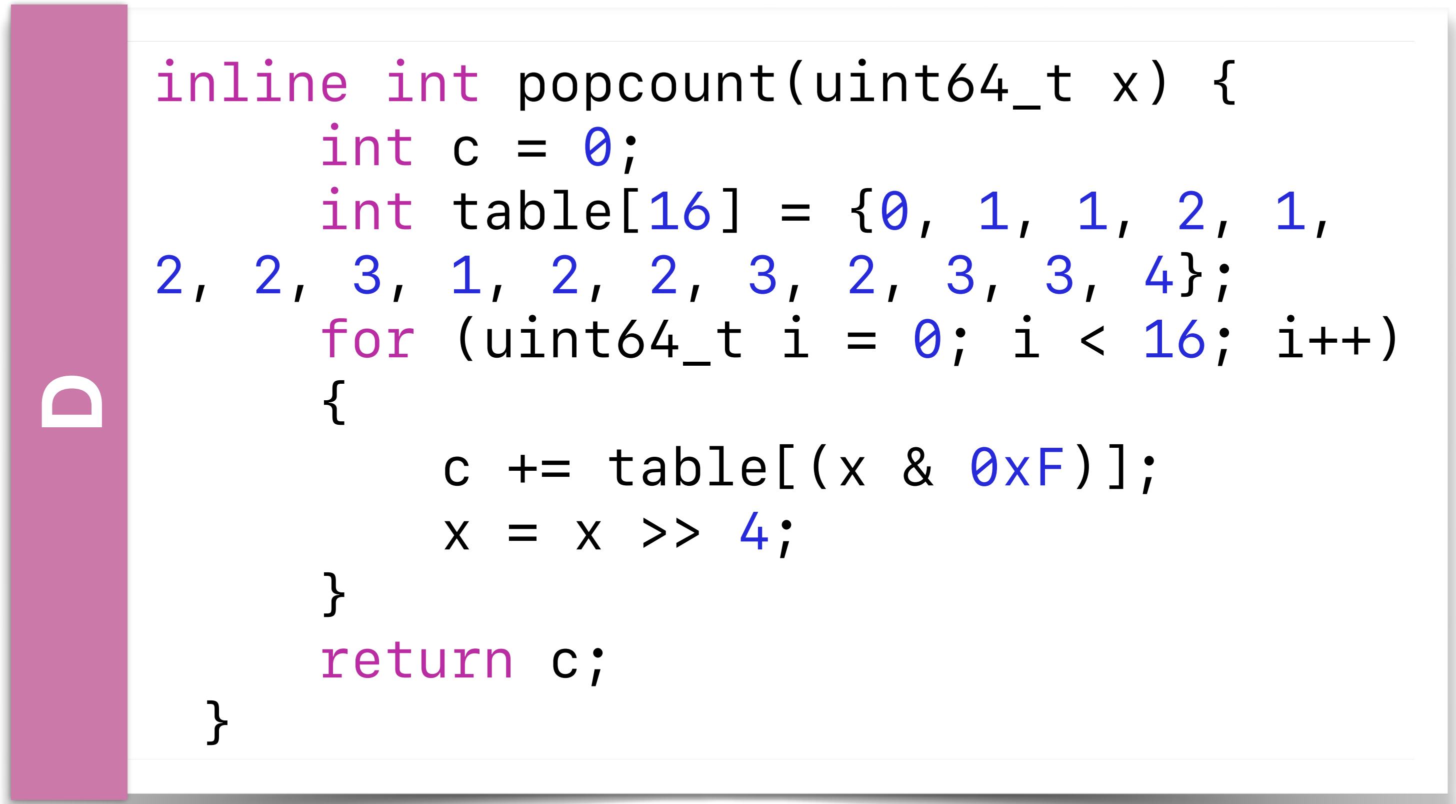
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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

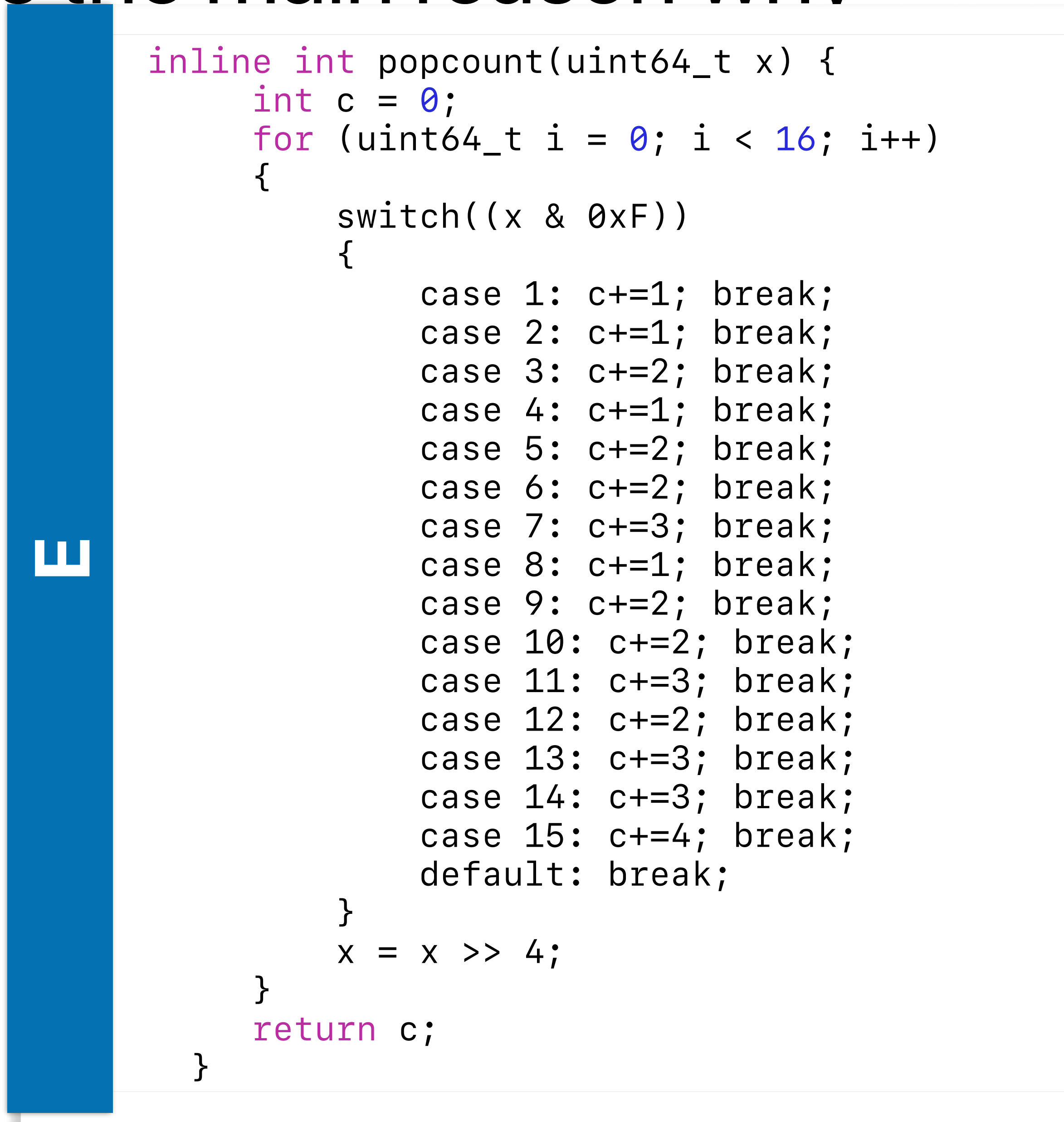
Why is E the slowest?

- 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

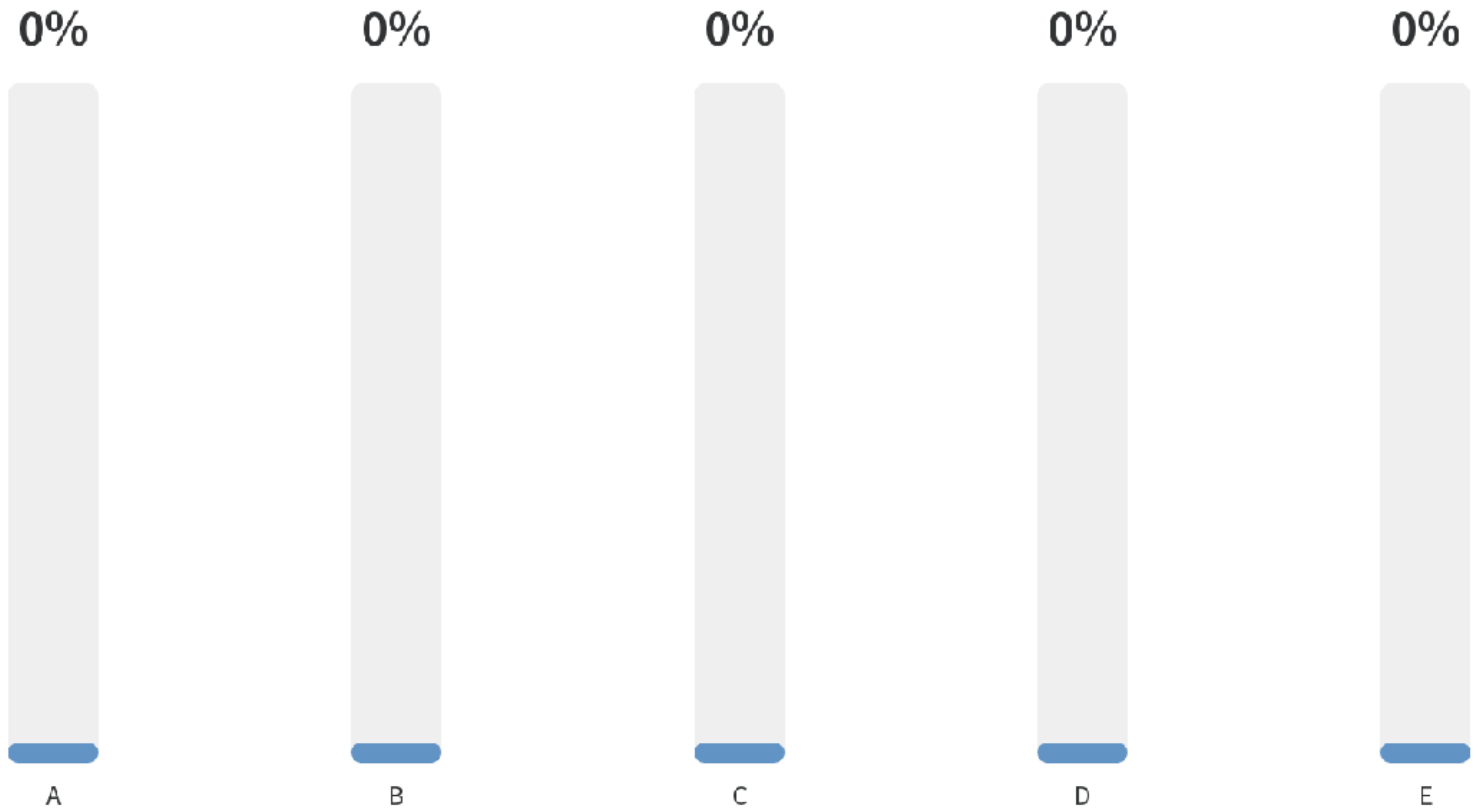

D

```
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;
  }
```

0



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

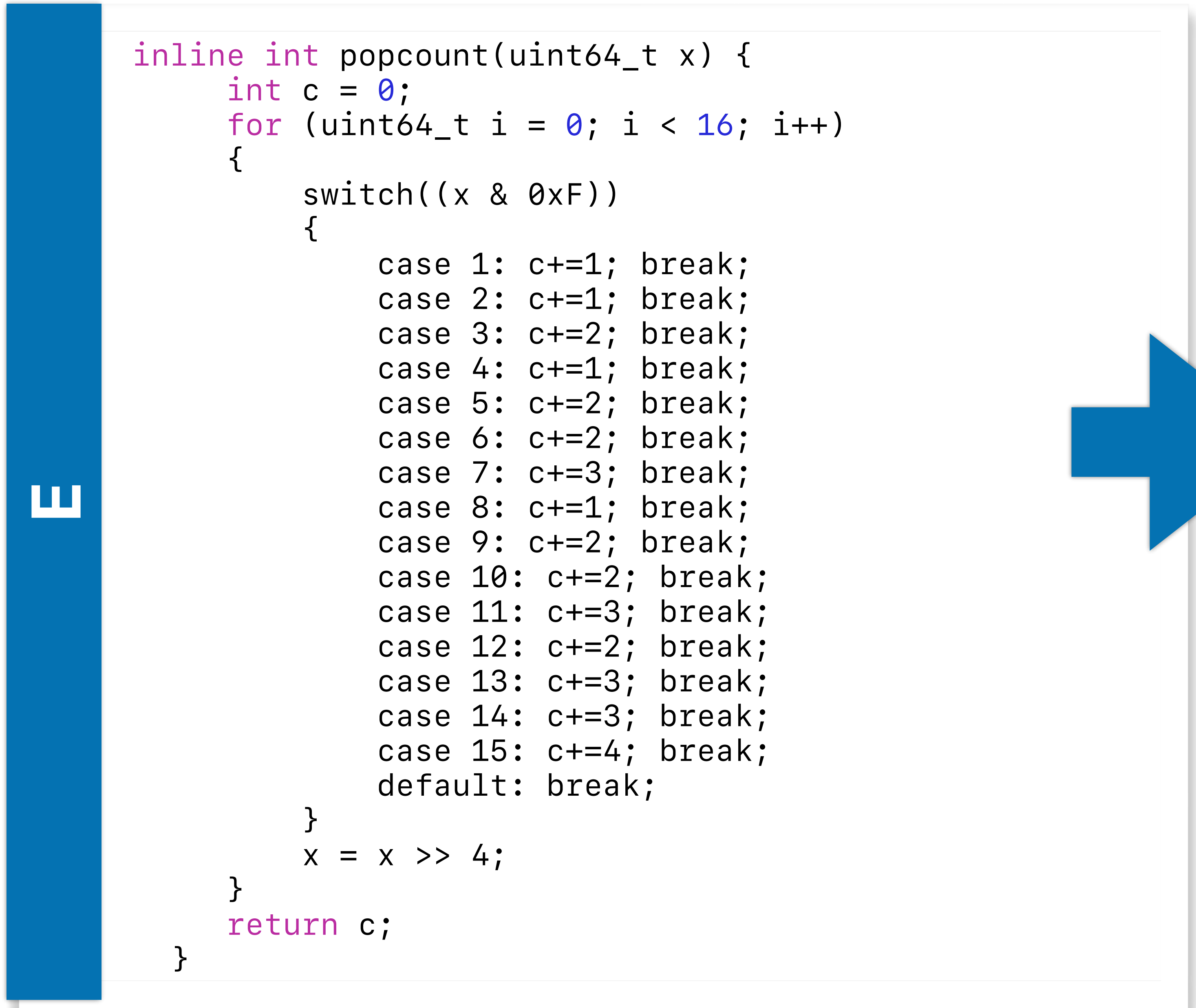
```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

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;
 }
```

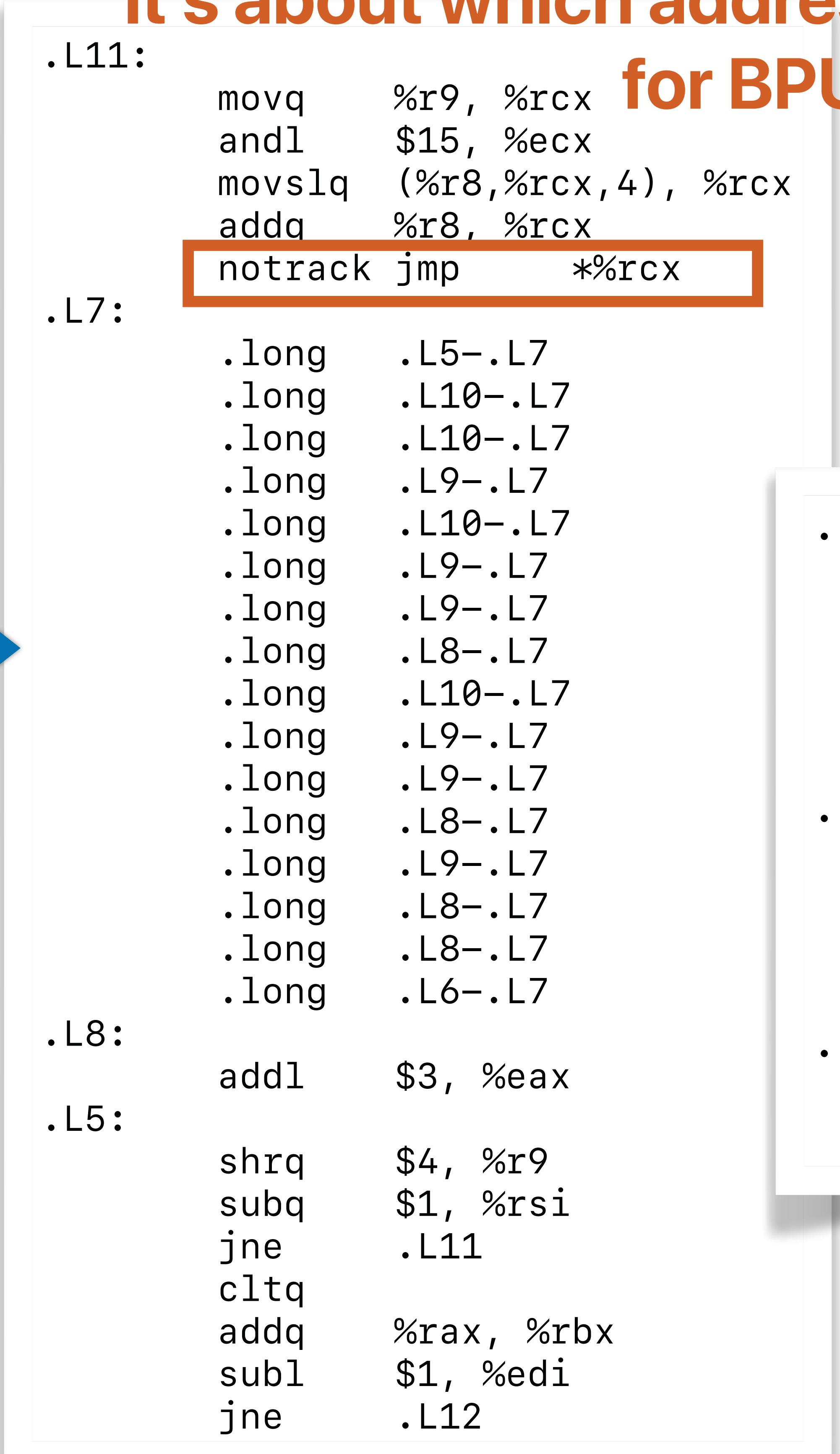
Why is E the slowest?

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

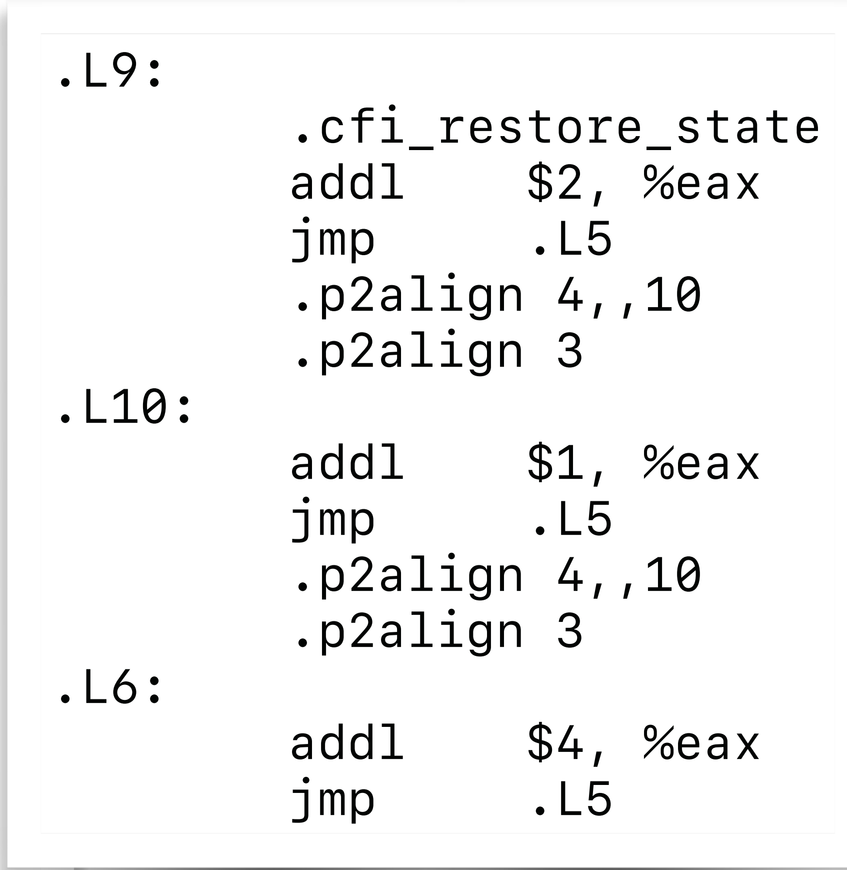


E

```c
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
```

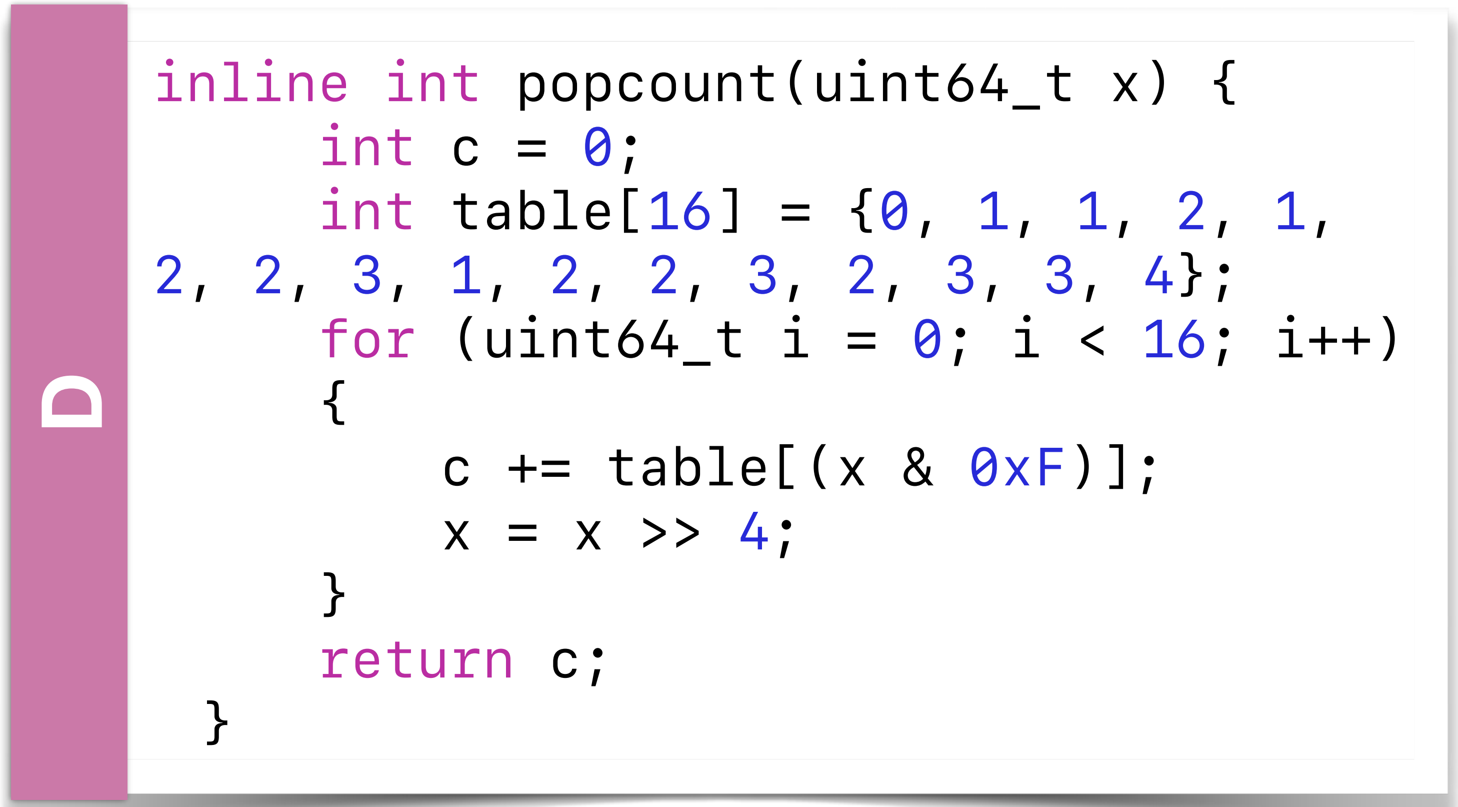


```
.L9:
        .cfi_restore_state
        addl     $2, %eax
        jmp      .L5
        .p2align 4,,10
        .p2align 3
.L10:
        addl     $1, %eax
        jmp      .L5
        .p2align 4,,10
        .p2align 3
.L6:
        addl     $4, %eax
        jmp      .L5
```

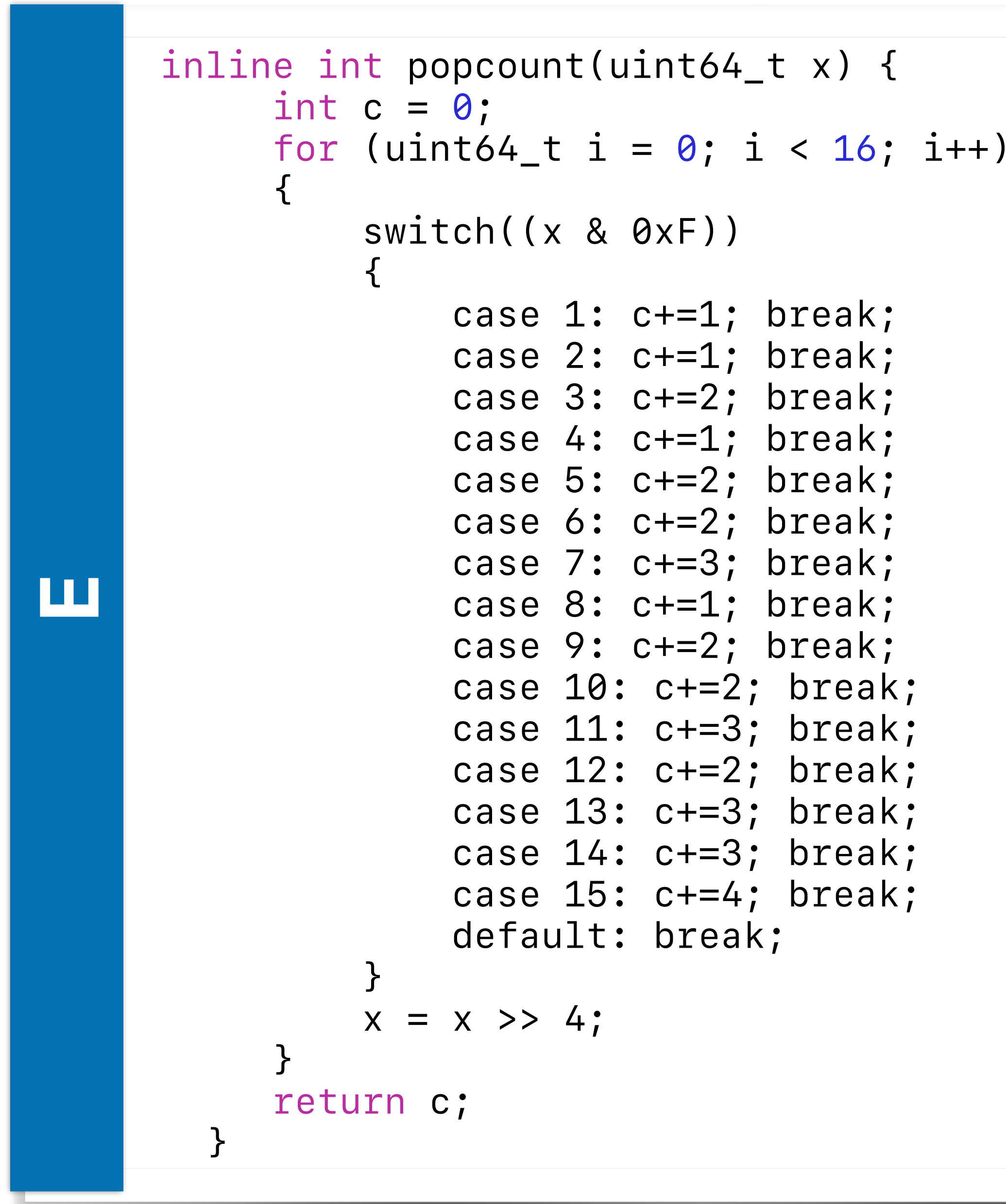
Why is E the slowest?

- 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

	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%

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
- 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!

Announcements

- Assignment #4 — due this Thursday
- Assignment #5 will release this Thursday

Computer
Science &
Engineering

203

つづく