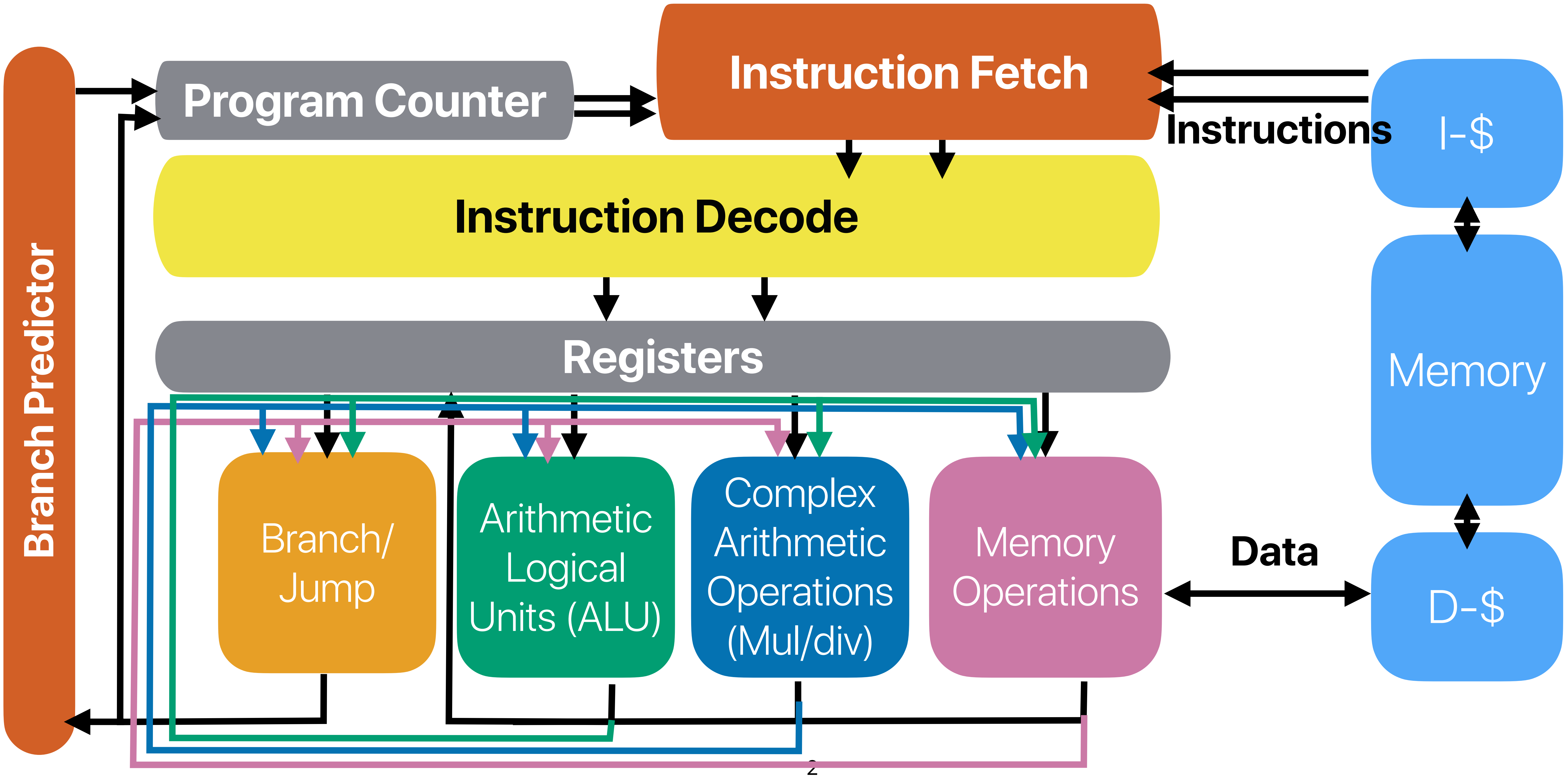
Programming on Modern Processors

Hung-Wei Tseng

Recap: Super Scalar



If we loop many times (assume perfect predictor)

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

	IF	ID	M1/ALU/BR	M2	M3	M4	WB
1	(1) (2)						
2	(3)(4)	(1) (2)					
3	(5)(6)	(3)(4)	(1)(2)				
4	(5)(6)	(3)(4)		(1)(2)			
5	(5)(6)	(3)(4)			(1)(2)		
6	(5)(6)	(3)(4)				(1)(2)	
7	(7)(8)	(5)(6)	(3)(4)				(1)(2)
8	(9)(10)	(7)(8)	(5)(6)	(3)(4)			
9	(9)(10)	(8)	(7)	(5)(6)	(3)(4)		
10	(9)(10)	(8)		(7)	(5)(6)	(3)(4)	
11	(9)(10)	(8)			(7)	(5)(6)	(3)(4)
12	(11)(12)	(9)(10)	(8)			(7)	(5)(6)
	(11)(12)	(10)	(9)	(8)			(7)
		(11)(12)	(10)	(9)	(8)		
			(11)(12)	(10)	(9)	(8)	
				(11)(12)	(10)	(9)	(8)
					(11)	(10)	(9)

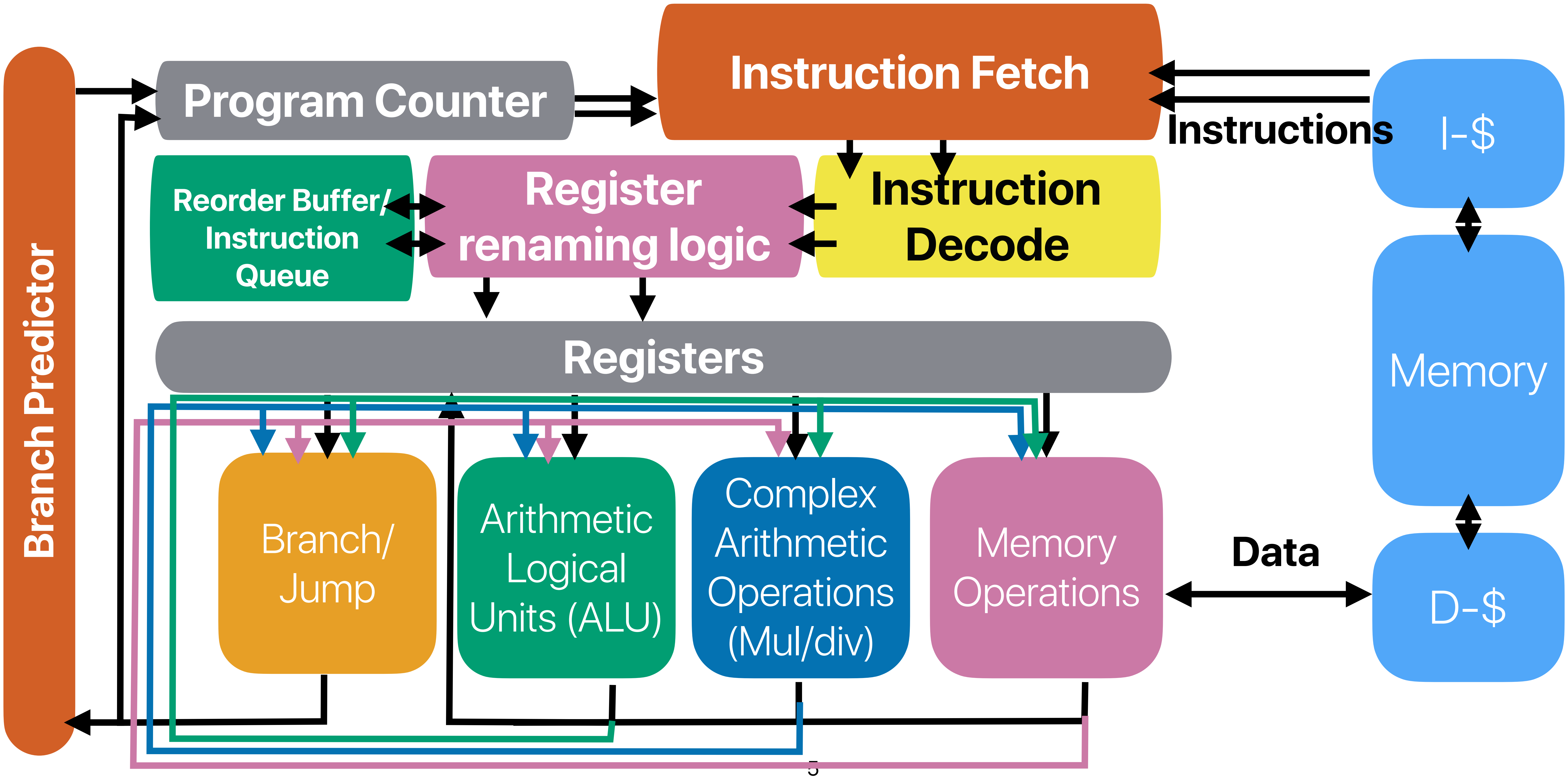
Everything we need for (4) is ready here! Why can't we execute it?

Why can't I start loading (6) & (11)?

Recap: Super-Scalar + Register Renaming + Speculative Execution

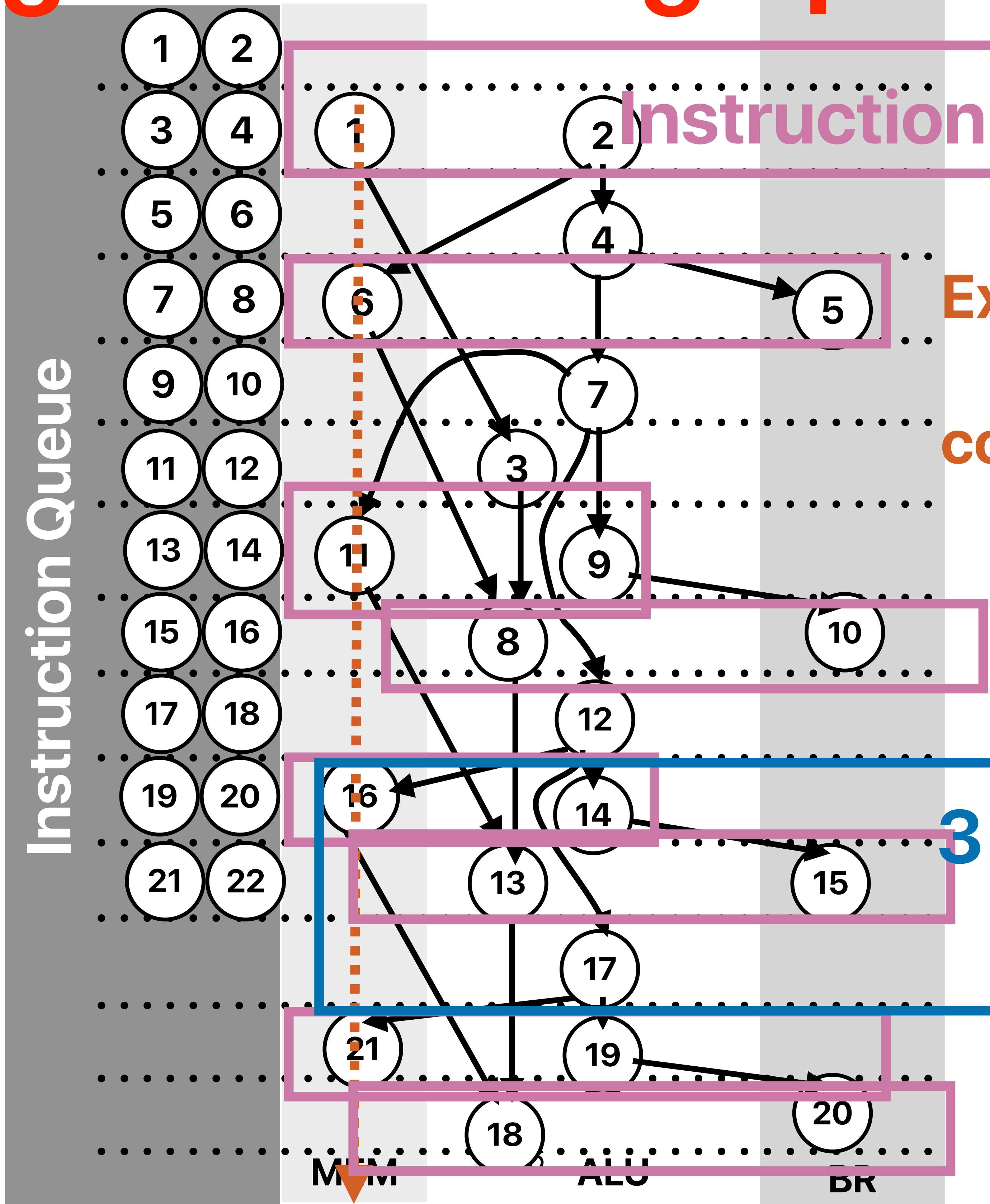
- SuperScalar: fetching & issuing multiple instructions from the same process/thread/running program at the same cycle
- Register Renaming & OoO Scheduling
  - Redirecting the output of an instruction instance to a physical register
  - Redirecting inputs of an instruction instance from architectural registers to correct physical registers
  - Executing an instruction all operands are ready (the values of depending physical registers are generated)
- Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Storing results in **reorder buffer** before the processor knows if the instruction is going to be executed or not.
  - Retiring instructions only when all earlier-order instructions are retired

Recap: Register renaming



Through data flow graph analysis

```
① movl (%rdi), %ecx
② addq $4, %rdi
③ addl %ecx, %eax
④ cmpq %rdx, %rdi
⑤ jne  .L3
⑥ movl (%rdi), %ecx
⑦ addq $4, %rdi
⑧ addl %ecx, %eax
⑨ cmpq %rdx, %rdi
⑩ jne  .L3
⑪ movl (%rdi), %ecx
⑫ addq $4, %rdi
⑬ addl %ecx, %eax
⑭ cmpq %rdx, %rdi
⑮ jne  .L3
⑯ movl (%rdi), %ecx
⑰ addq $4, %rdi
⑱ addl %ecx, %eax
⑲ cmpq %rdx, %rdi
⑳ jne  .L3
㉑ movl (%rdi), %ecx
```



Instruction-Level Parallelism (ILP)

Execution time is determined by the "critical path" composed by 1, 6, 11, …, 1+5n

3 cycles every iteration

CPI = $\frac{3}{5} = 0.6!$

What about "linked list"

Performance determined by the critical path

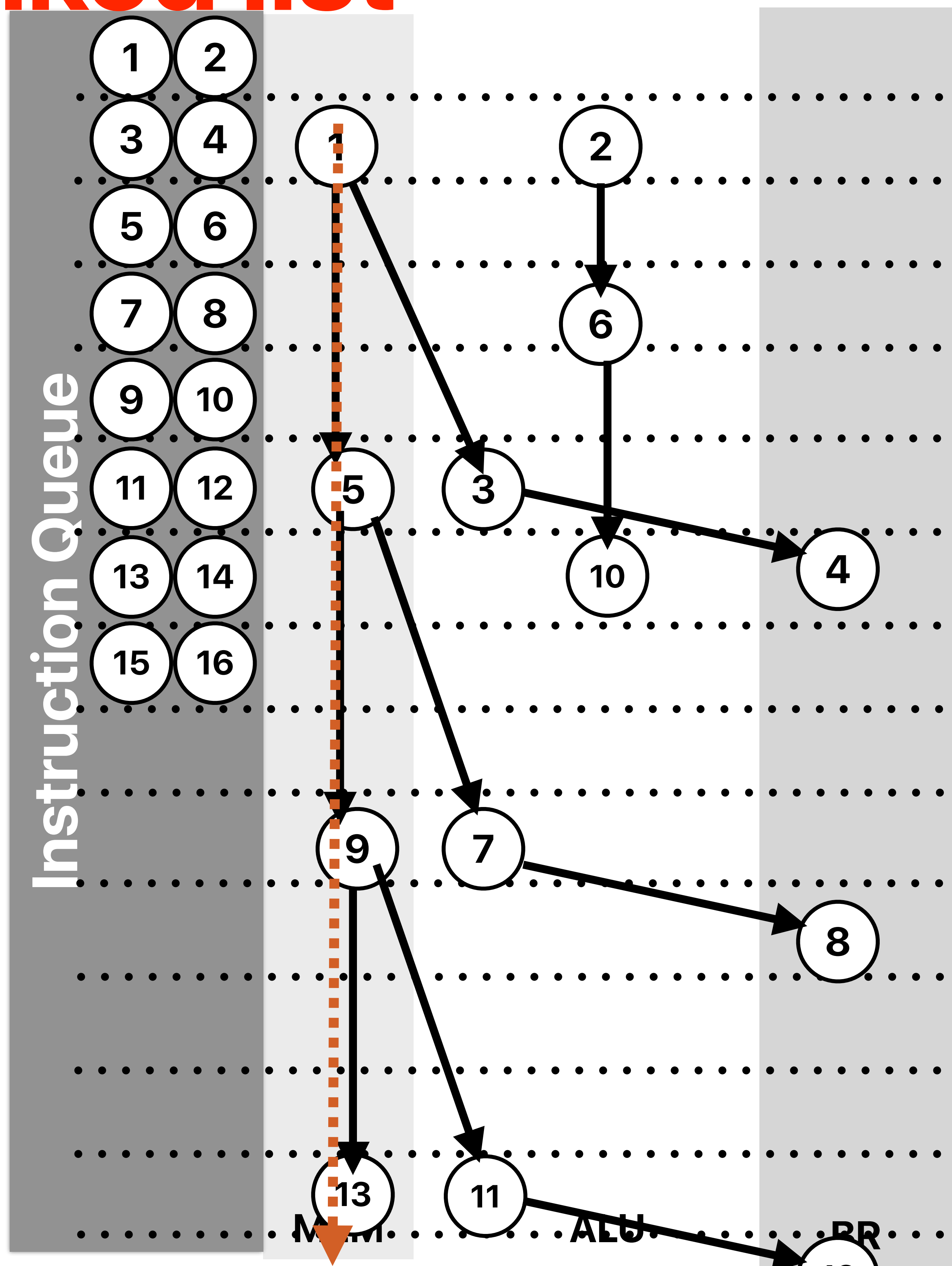
4 cycles each iteration

4 instructions per iteration

$$CPI = \frac{4}{4} = 1$$

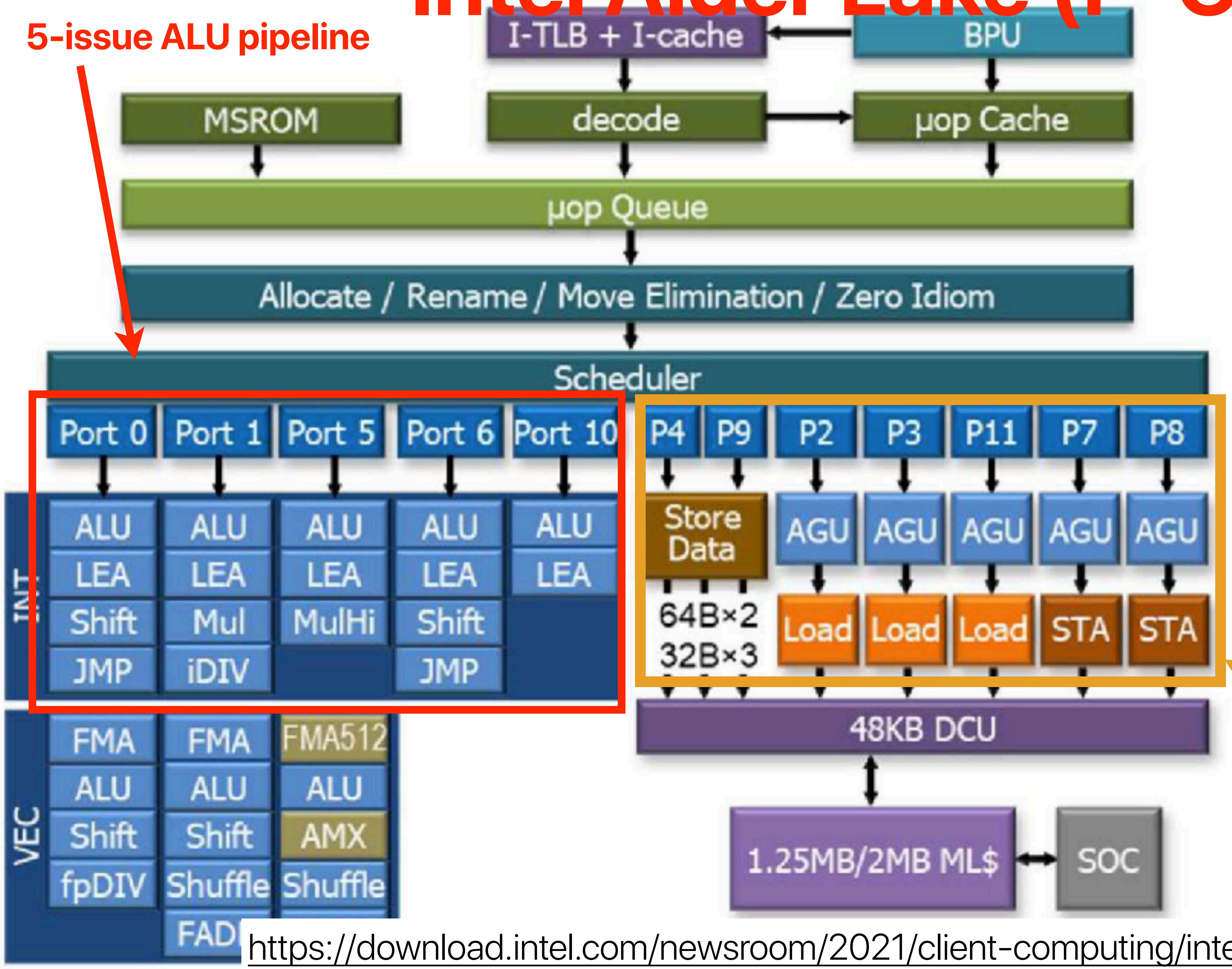
```
do {
     number_of_nodes++;
     current = current->next;
} while ( current != NULL );
```

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



Intel Alder Lake (P-Core)

5-issue ALU pipeline



7-issue memory pipeline

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

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

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

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

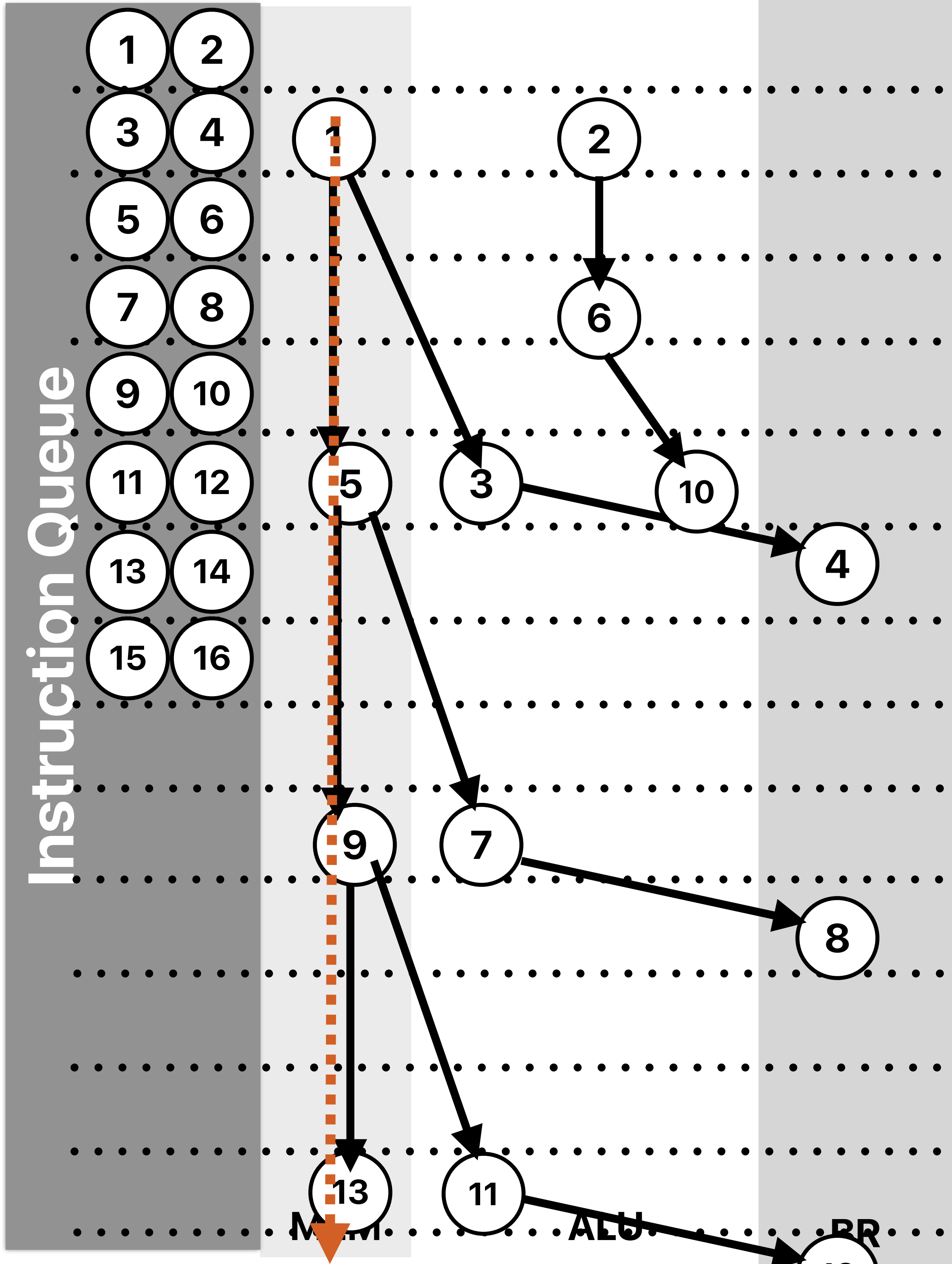
https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

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

Doesn't help that much!
— It's important that the programmer should write code that can exploit "ILP"

```
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
```



3:00

Refresh our minds! What are the characteristics of modern processors?

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
  - Variable history predictors

Outline

- Programming on modern processors — exploiting instruction-level parallelism
- Thread-level parallelism

Demo: 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.

Demo: 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;
}
```

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 can incur fewer data hazards

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?

- 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 can incur fewer data hazards

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

```c
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

```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;
}
```

```
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
```

15*(n/4) = 3.75*n instructions

Only one branch for four iterations in A

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 can incur fewer data hazards

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;
}
```

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 can incur fewer data hazards

  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?

- 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 can incur fewer data hazards — 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;
}
```

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 can incur fewer data hazards 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 can incur fewer data hazards 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;
     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 can incur fewer data hazards than C — about the same

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

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;
 }
```

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?

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

	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%

Announcements

- Assignment #4 — due this Sunday
- Lab #3 — due this Tomorrow

Computer
Science &
Engineering

142

つづく