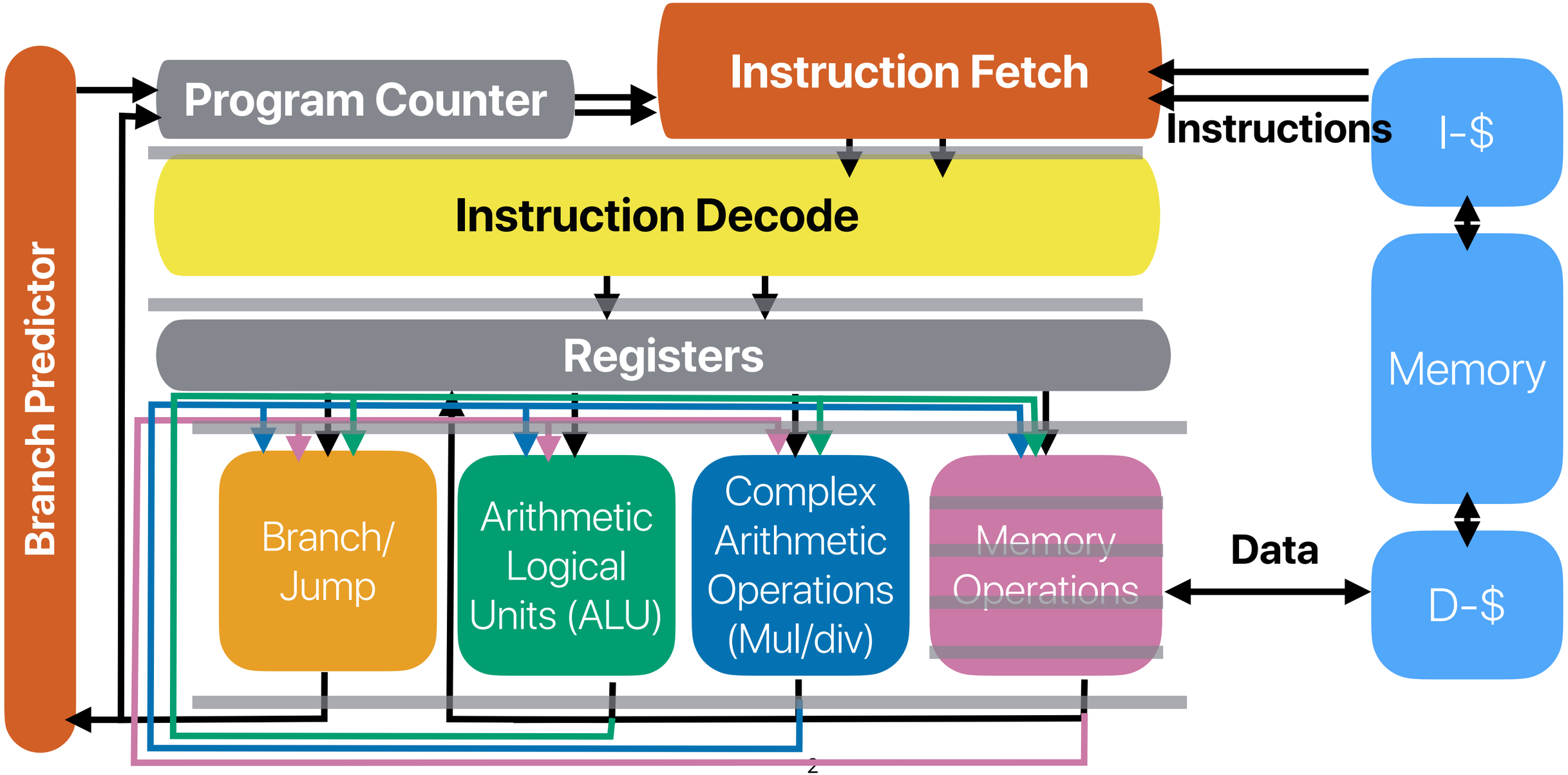
Programming on Modern Processors: The Single Thread Version

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)

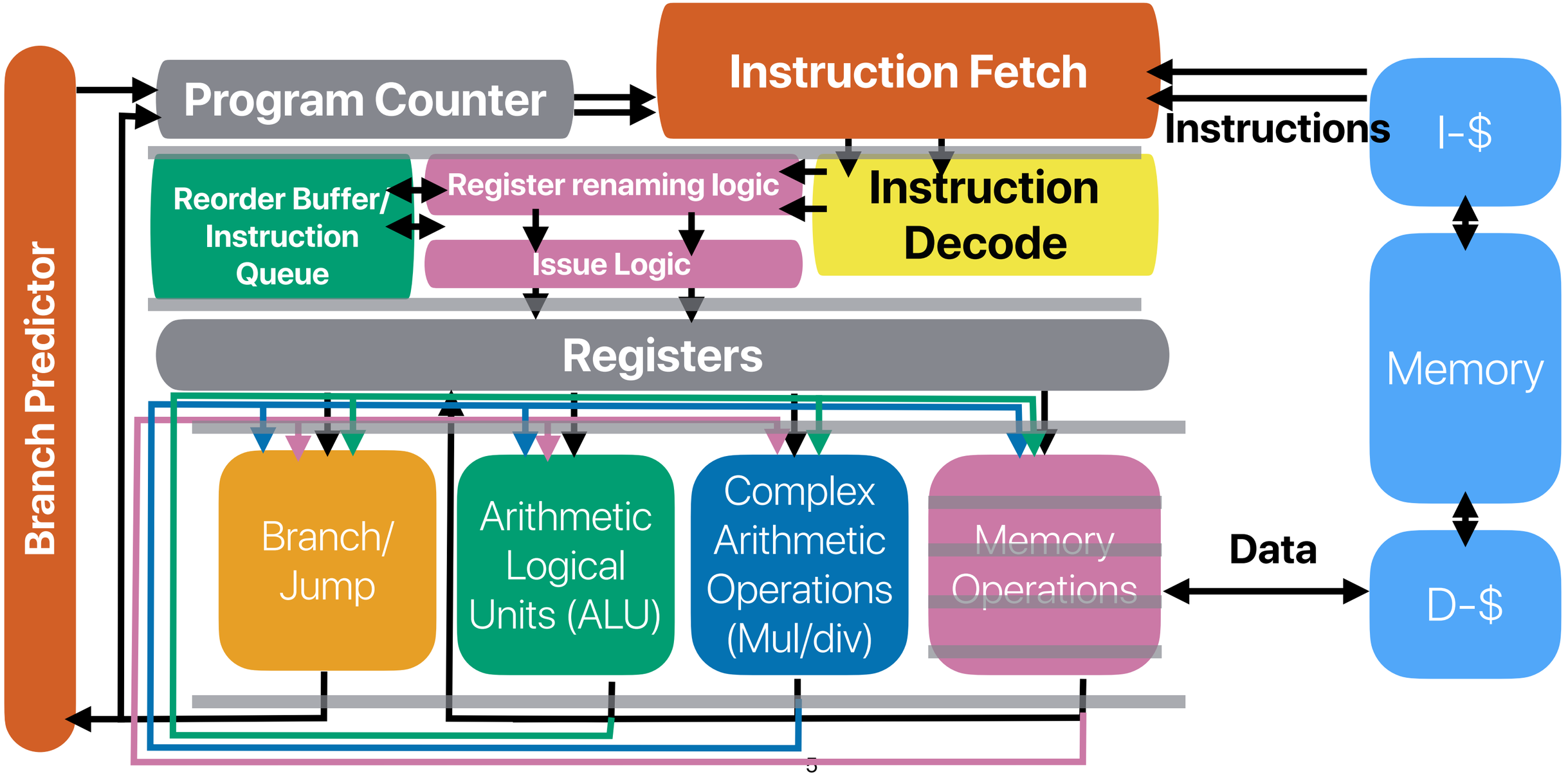
Everything we need for (4) is ready here! Why can't we execute it?

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

Recap: Super-Scalar + Register Renaming + Speculative Execution

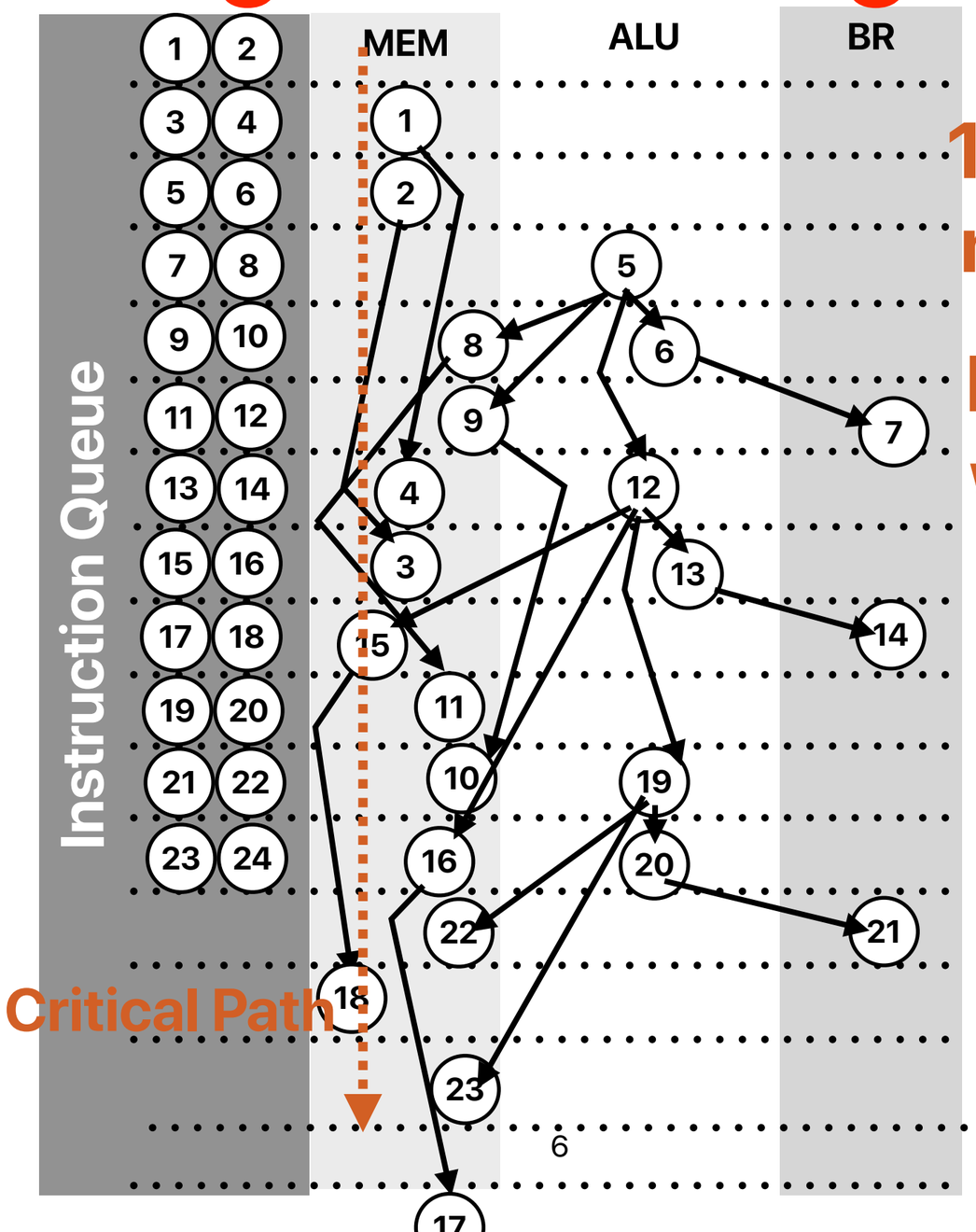
- SuperScalar: fetching & issuing multiple instructions from the same process/thread/running program at the same cycle
- Register Renaming & OoO Scheduling
  - Redirecting the output of an instruction instance to a physical register
  - Redirecting inputs of an instruction instance from architectural 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



Recap: Through data flow graph analysis

```
①  movq (%rdi,%rax), %rsi
②  movq (%rcx,%rax), %r8
③  movq %r8, (%rdi,%rax)
④  movq %rsi, (%rcx,%rax)
⑤  addq $8, %rax
⑥  cmpq %r9, %rax
⑦  jne  .L9
⑧  movq (%rdi,%rax), %rsi
⑨  movq (%rcx,%rax), %r8
⑩  movq %r8, (%rdi,%rax)
⑪  movq %rsi, (%rcx,%rax)
⑫  addq $8, %rax
⑬  cmpq %r9, %rax
⑭  jne  .L9
⑮  movq (%rdi,%rax), %rsi
⑯  movq (%rcx,%rax), %r8
⑰  movq %r8, (%rdi,%rax)
⑱  movq %rsi, (%rcx,%rax)
⑲  addq $8, %rax
⑳  cmpq %r9, %rax
㉑  jne  .L9
㉒  movq (%rdi,%rax), %rsi
㉓  movq (%rcx,%rax), %r8
㉔  movq %r8, (%rdi,%rax)
㉕  movq %rsi, (%rcx,%rax)
㉖  addq $8, %rax
㉗  cmpq %r9, %rax
```



12 cycles for every 11 memory instructions

If we have 11 loops, it will have 44 memory instructions, 77 instructions in total and take 48 cycles

CPI:

$$\frac{48}{77} = 0.62$$

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

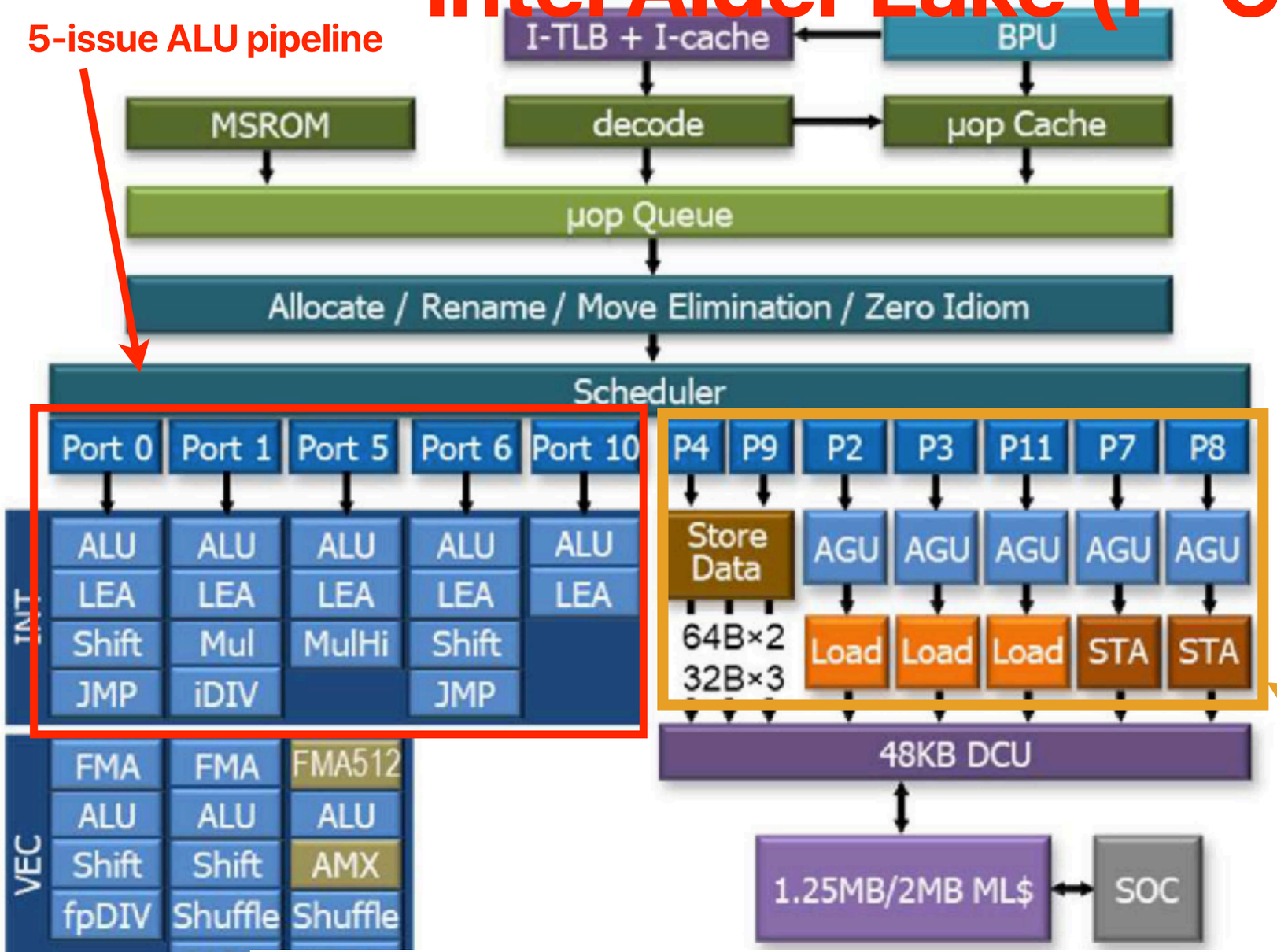
Intel Alder Lake (P-Core)

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

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

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

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



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

Linked list v.s. arrays

- We can use either a linked list or an array to store a list of data, compare the performance of the array (version A) and linked list (version B) implementations that can achieve the same outcome as below. Assume we have a processor with a reasonably good branch predictor and unlimited fetch/issue width, please identify the correct statements.
  - ① If the dataset is large, the A will outperform the B
  - ② If the dataset is small, there is very little performance difference between A and B
  - ③ If the dataset is small, B will outperform A as A has more branch instructions
  - ④ If the dataset is small, A will outperform B as A has fewer data dependencies

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

A

```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

```
while(node)
{
    node = node->next;
    number_of_nodes++;
}
```

Linked list v.s. arrays

- We can use either a linked list or an array to store a list of data, compare the performance of the array (version A) and linked list (version B) implementations that can achieve the same outcome as below. Assume we have a processor with a reasonably good branch predictor and unlimited fetch/issue width, please identify the correct statements.
  - ① If the dataset is large, the A will outperform the B
  - ② If the dataset is small, there is very little performance difference between A and B
  - ③ If the dataset is small, B will outperform A as A has more branch instructions
  - ④ If the dataset is small, A will outperform B as A has fewer data dependencies

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

A

```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

```
while(node)
{
    node = node->next;
    number_of_nodes++;
}
```

Linked list v.s. arrays

- We can use either a linked list or an array to store a list of data, compare the performance of the array (version A) and linked list (version B) implementations that can achieve the same outcome as below. Assume we have a processor with a reasonably good branch predictor and unlimited fetch/issue width, please identify the correct statements.
  - ① If the dataset is large, the A will outperform the B
  - ② If the dataset is small, there is very little performance difference between A and B
  - ③ If the dataset is small, B will outperform A as A has more branch instructions
  - ④ If the dataset is small, A will outperform B as A has fewer data dependencies

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

A

```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

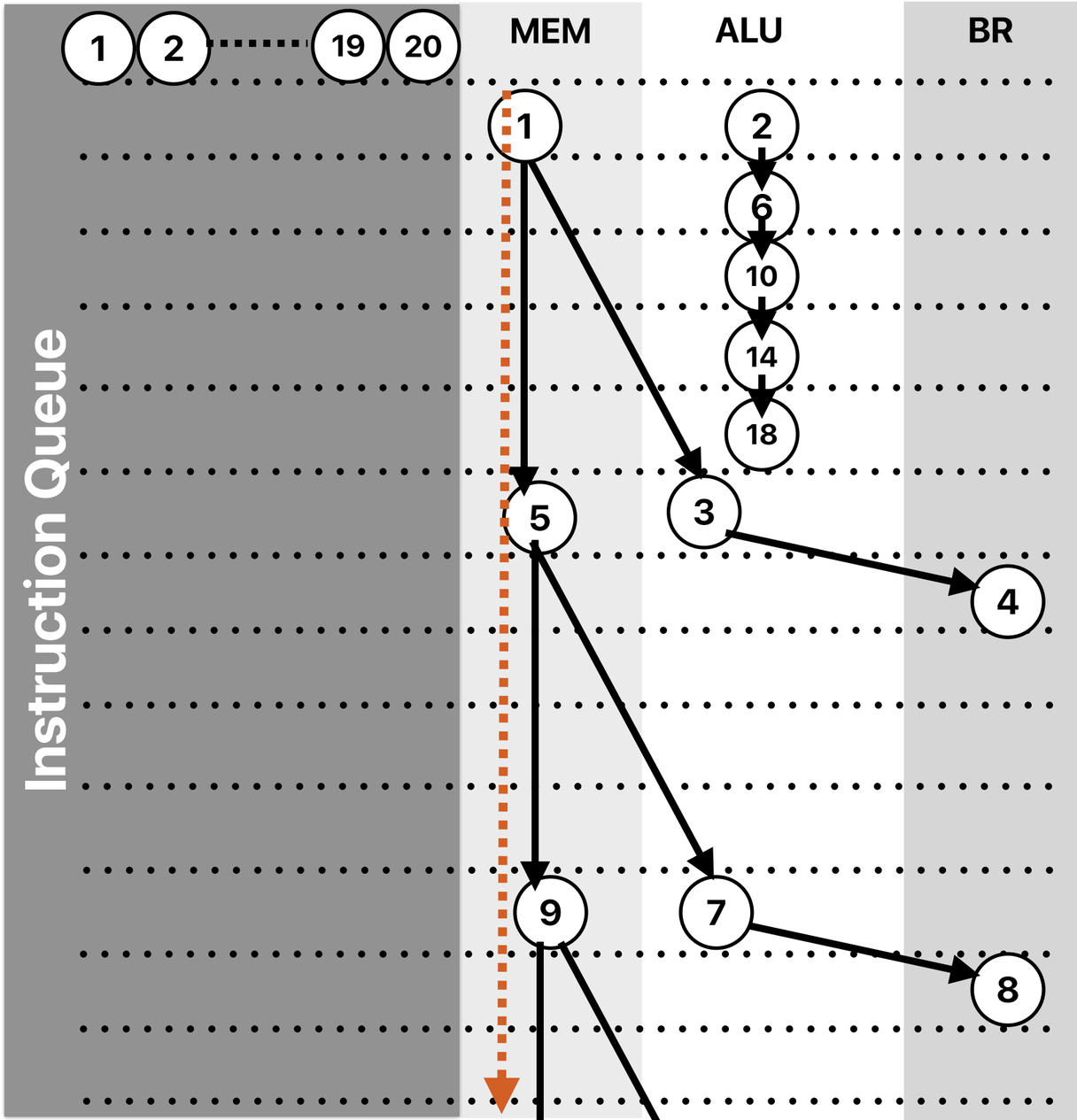
```
while(node)
{
    node = node->next;
    number_of_nodes++;
}
```

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
```

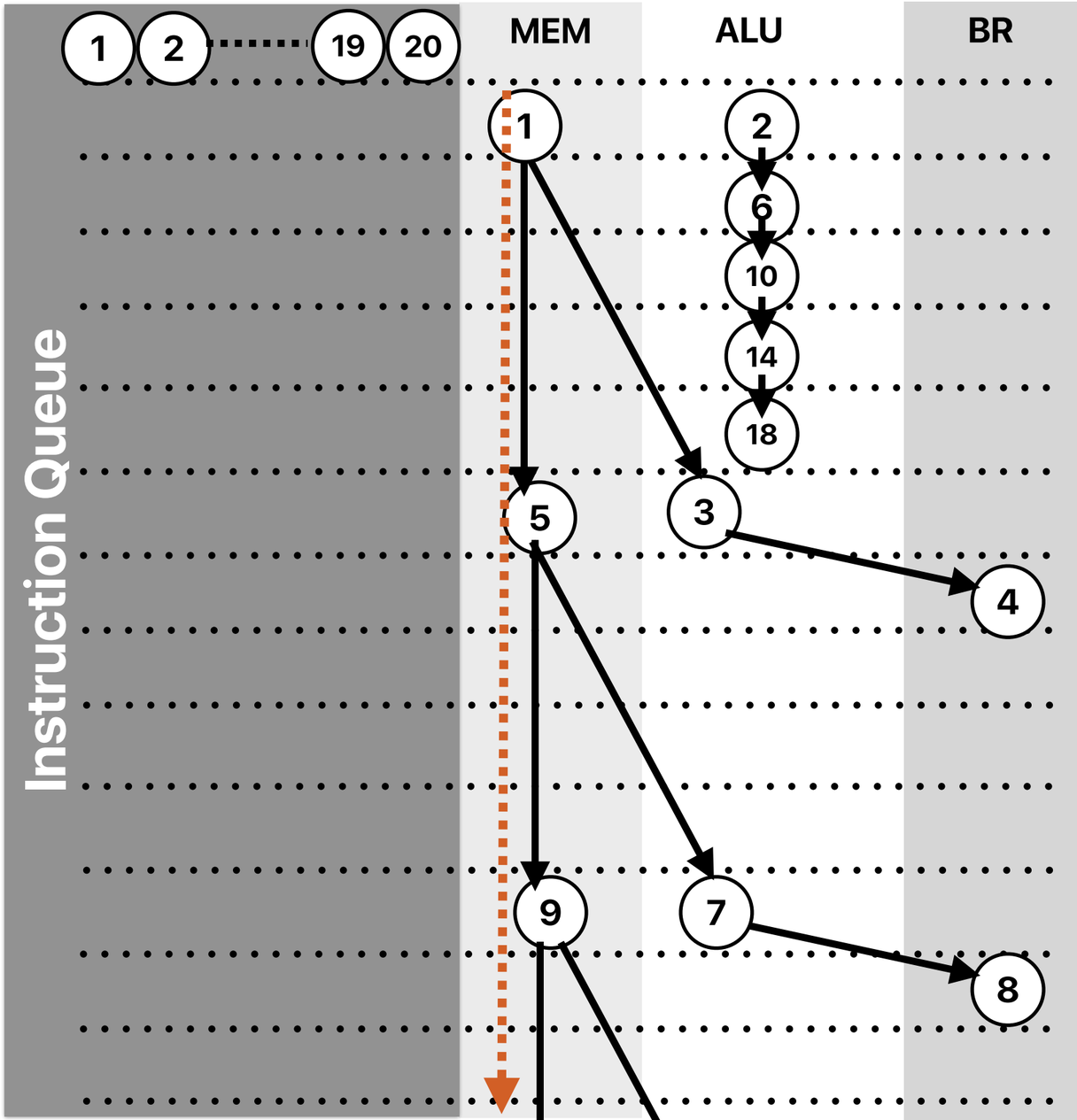


What if we have "unlimited" fetch/issue width — "linked list"

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
```

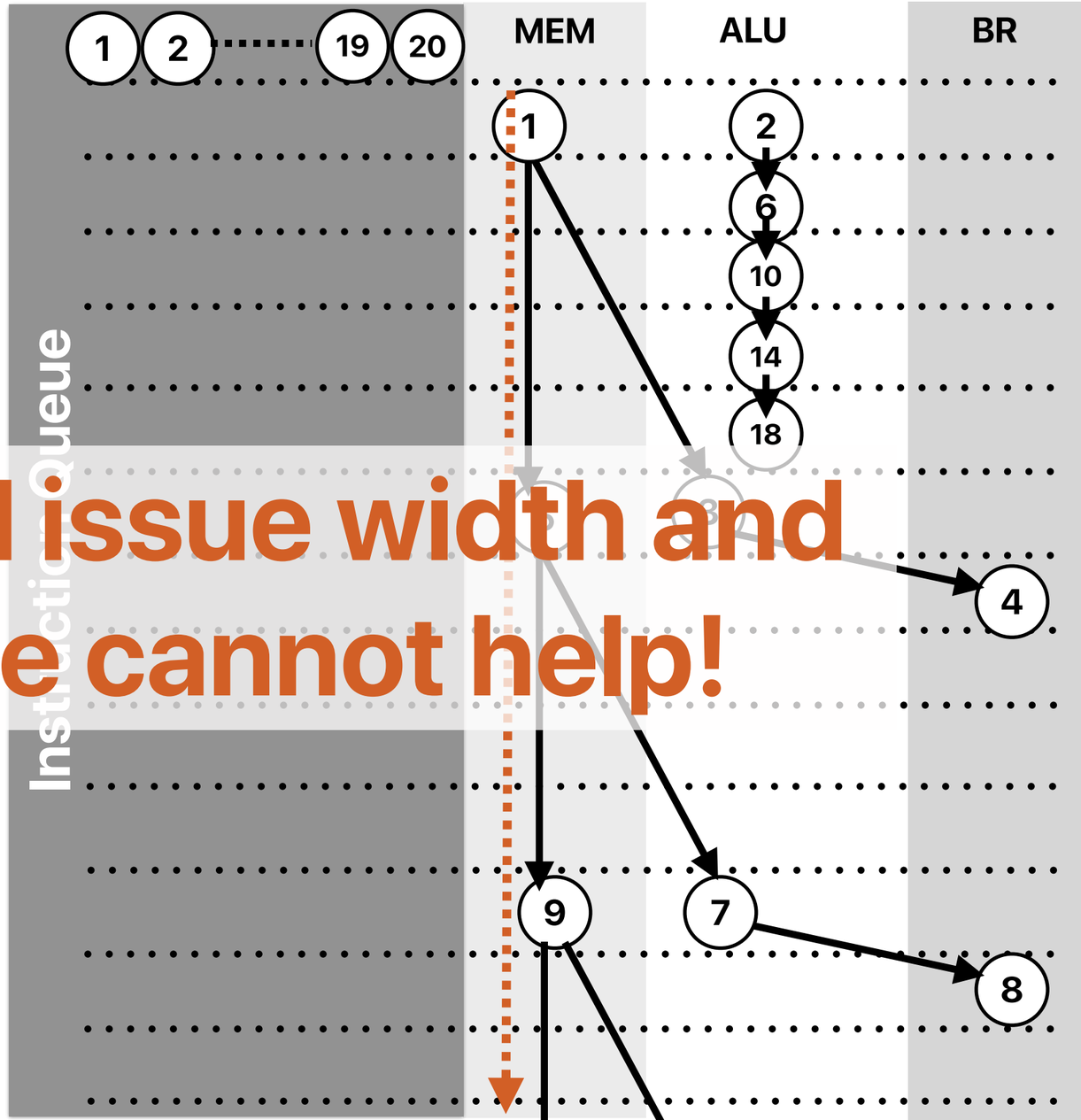


What if we have "unlimited" fetch/issue width — "linked list"

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!

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;
}
```

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;
}
```

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?

- 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;
}
```

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

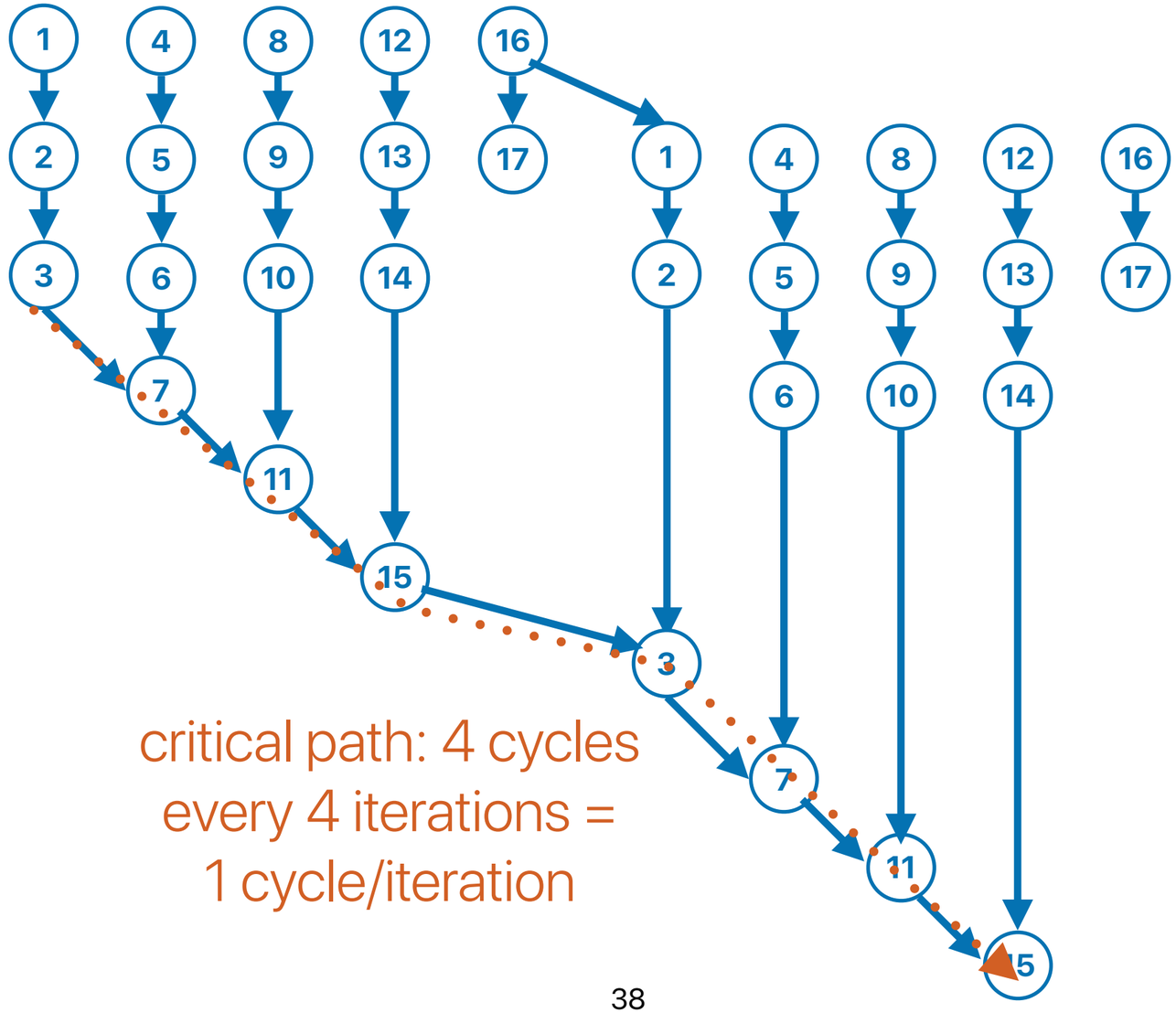
```
movl    %ecx, %eax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    %rdx
andl    $1, %edx
addl    %eax, %edx
movq    %rcx, %rax
shrq    $2, %rax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    $3, %rdx
andl    $1, %edx
addl    %eax, %edx
shrq    $4, %rcx
jne     .L6
```

Only one branch for four iterations in A

Why is B better than A?

B

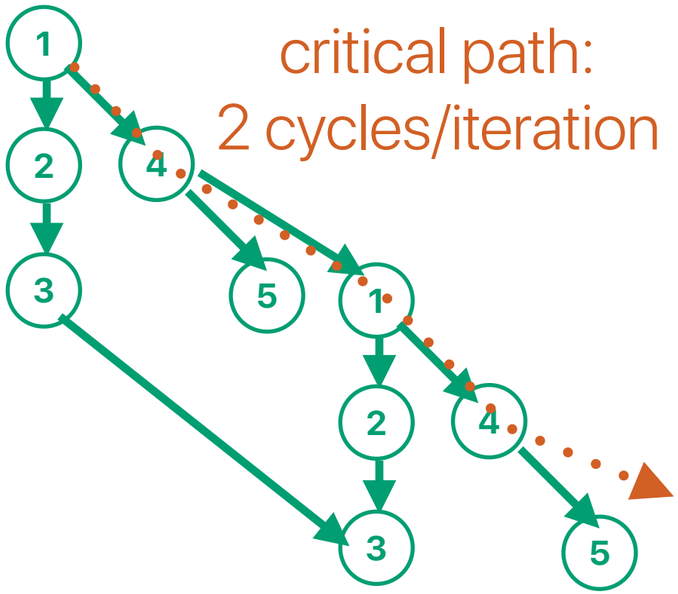
```
① movl    %ecx, %eax
② andl    $1, %eax
③ addl    %edx, %eax
④ movq    %rcx, %rdx
⑤ shrq    %rdx
⑥ andl    $1, %edx
⑦ addl    %eax, %edx
⑧ movq    %rcx, %rax
⑨ shrq    $2, %rax
⑩ andl    $1, %eax
⑪ addl    %edx, %eax
⑫ movq    %rcx, %rdx
⑬ shrq    $3, %rdx
⑭ andl    $1, %edx
⑮ addl    %eax, %edx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

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?

- 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

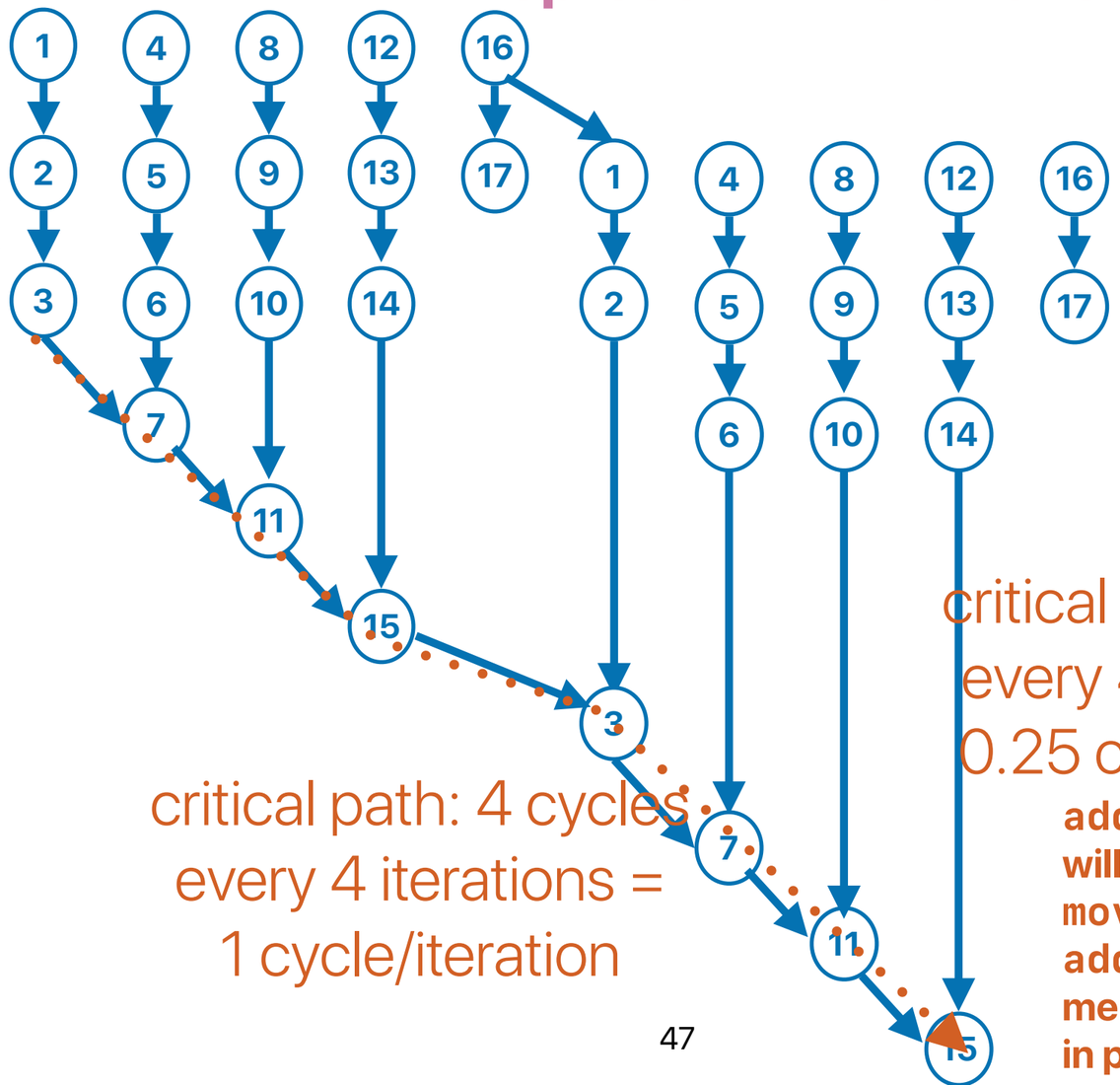
```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)      {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

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
```

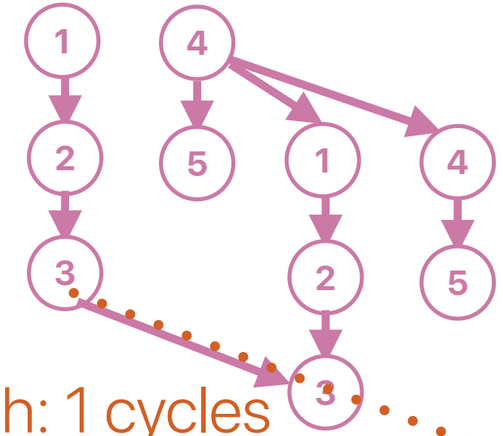
These 5 instructions represent 4 iterations!!!



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

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

— C only needs one load, one add, one shift, the same amount of iterations

— the same number being predicted.

— the same amount of branches

— 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

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;
}
```

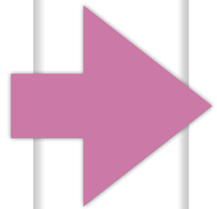
D

```
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

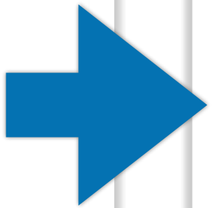
- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
- Loop unrolling is effective as control overhead is still significant despite we have branch predictors and OoO.
- With caches, 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

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;
}
```

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!

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

Announcements

- Assignment #4 — due this Saturday
- Last reading quiz — due next Tuesday. Will be up later today.
- Assignment #5 — due next Thursday. Will be up later tomorrow.
- Datahub service will be non-reachable between **6p-10p tomorrow** due to an emergency maintenance of UCR's network

Computer
Science &
Engineering

142

つづく