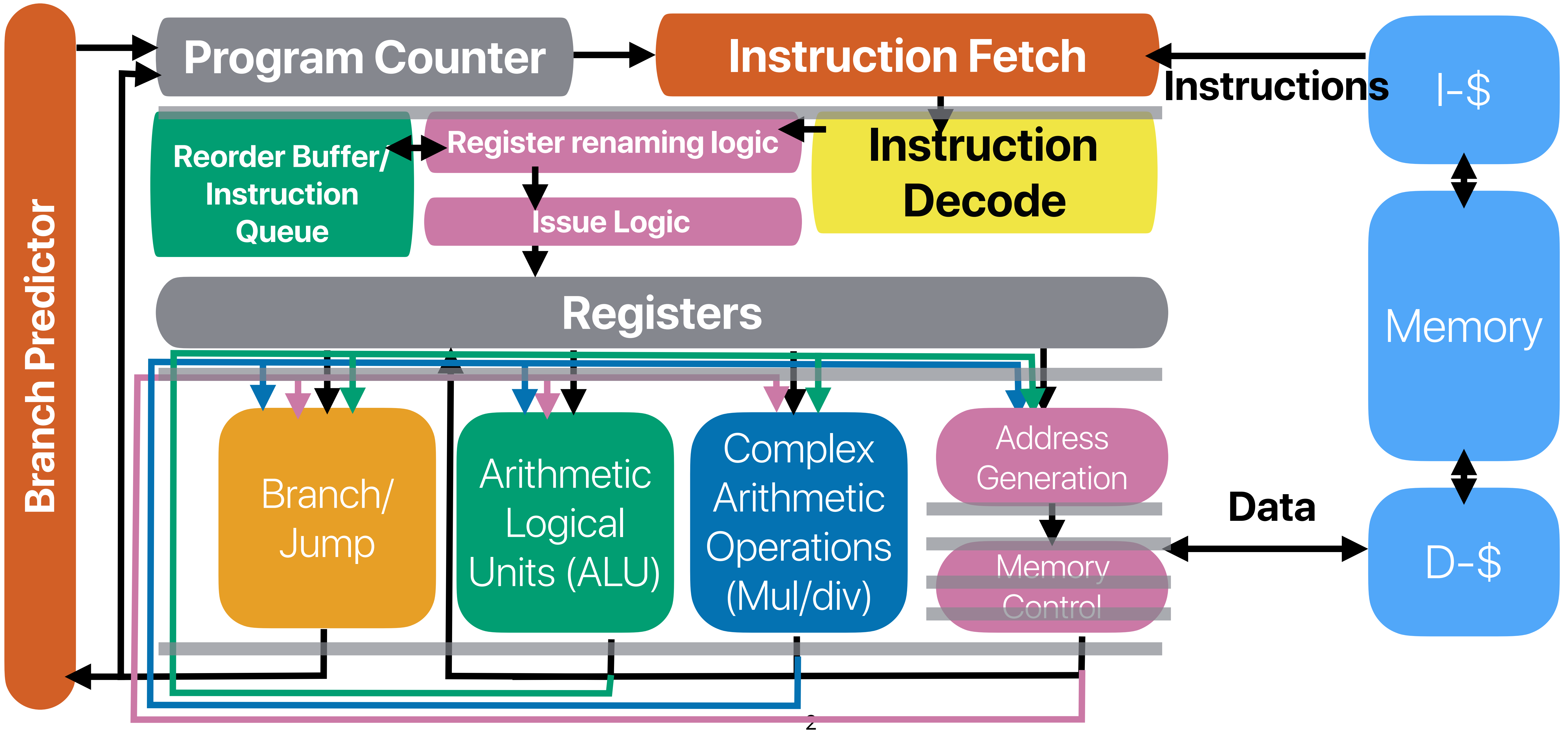
Programming on Modern Processors: The Single Thread Version

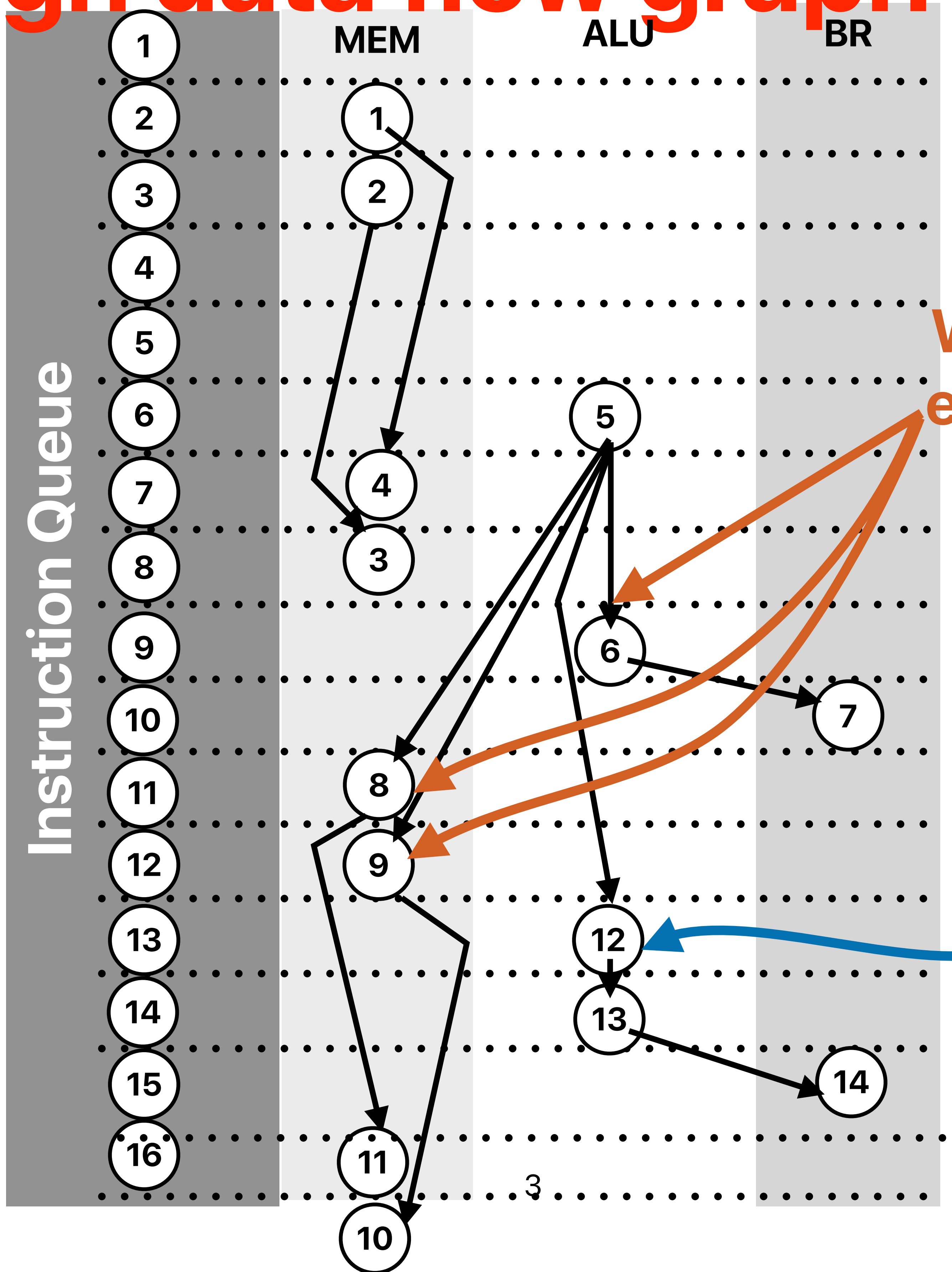
Hung-Wei Tseng

Recap: Register renaming + OoO + RoB



Through data flow graph analysis

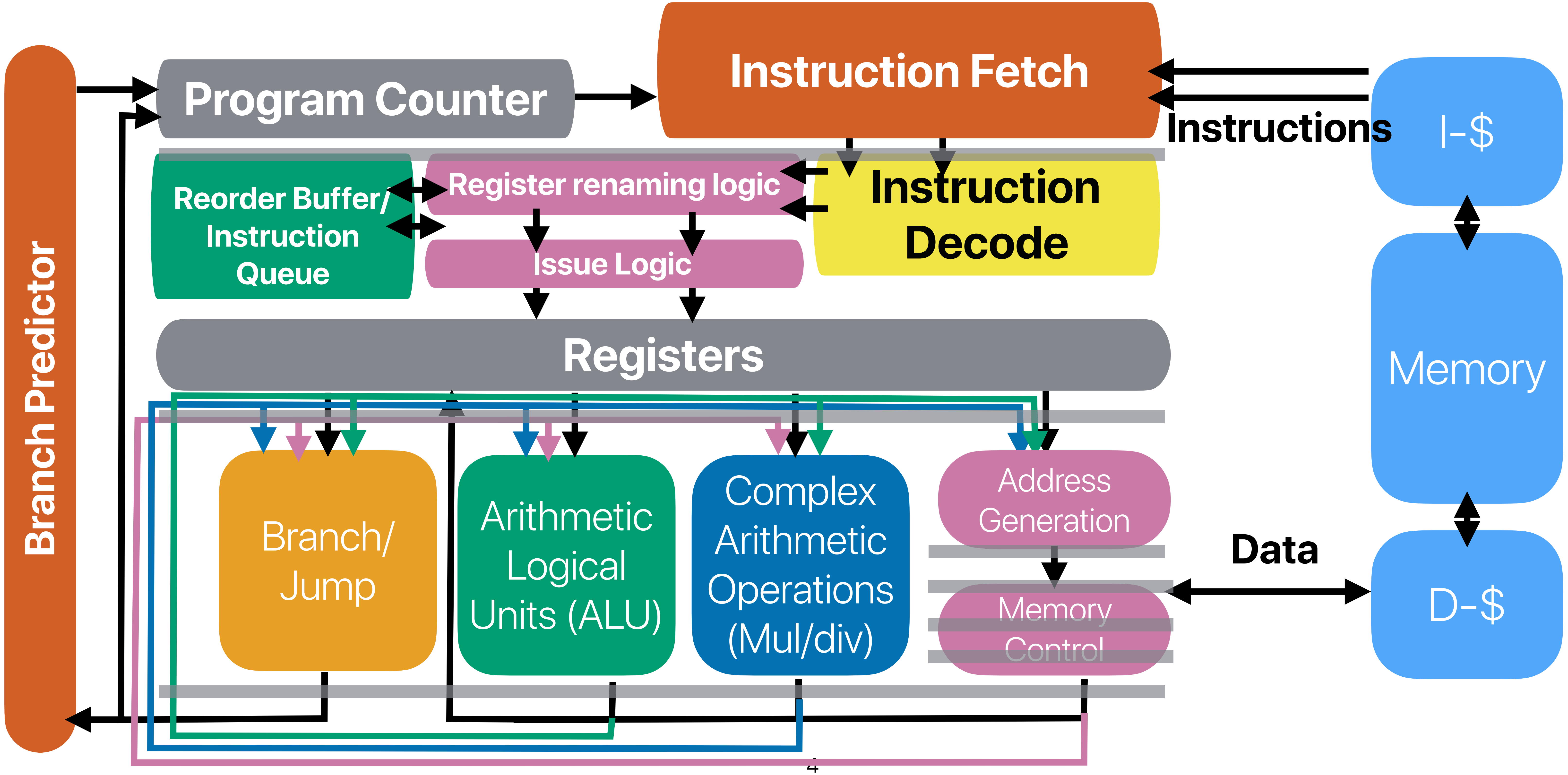
```
①   movq (%rdi,%rax), %rsi
②   movq (%rcx,%rax), %r8
③   movq %r8, (%rdi,%rax)
④   movq %rsi, (%rcx,%rax)
⑤   addq $8, %rax
⑥   cmpq %r9, %rax
⑦   jne  .L9
⑧   movq (%rdi,%rax), %rsi
⑨   movq (%rcx,%rax), %r8
⑩   movq %r8, (%rdi,%rax)
⑪   movq %rsi, (%rcx,%rax)
⑫   addq $8, %rax
⑬   cmpq %r9, %rax
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
```



We cannot issue them earlier simply because structural hazards!

We could have this executed earlier if it's in the queue earlier

Recap: Register renaming + OoO + ROB + SuperScalar

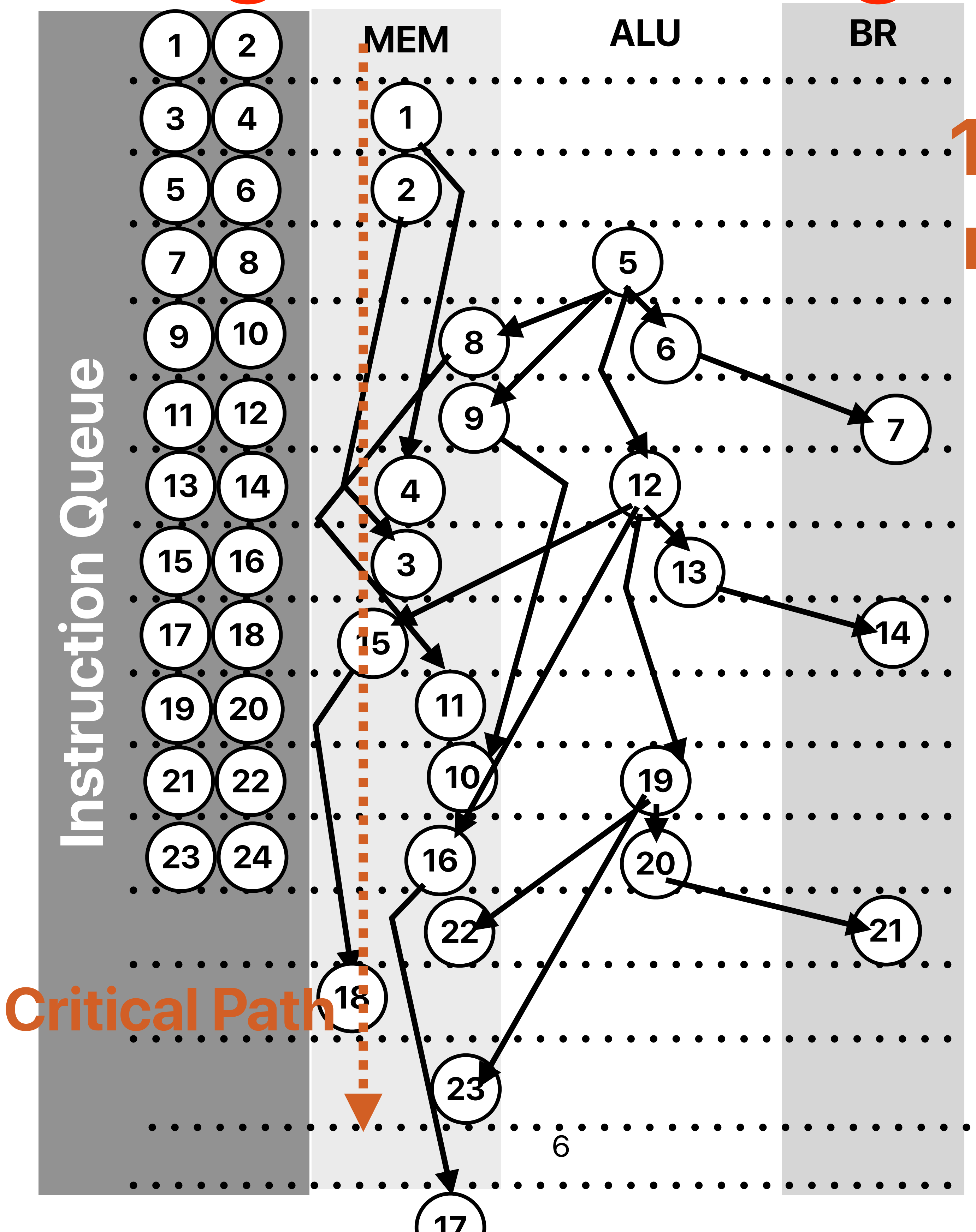


Recap: Super-Scalar + Register Renaming + Speculative Execution

- 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: 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 n loops, it will have $4n$ memory instructions, $7n$ instructions in total and take $\frac{4n \times 11}{12} = 4.37n$ cycles

CPI: $\frac{4.37n}{7n} = 0.62$

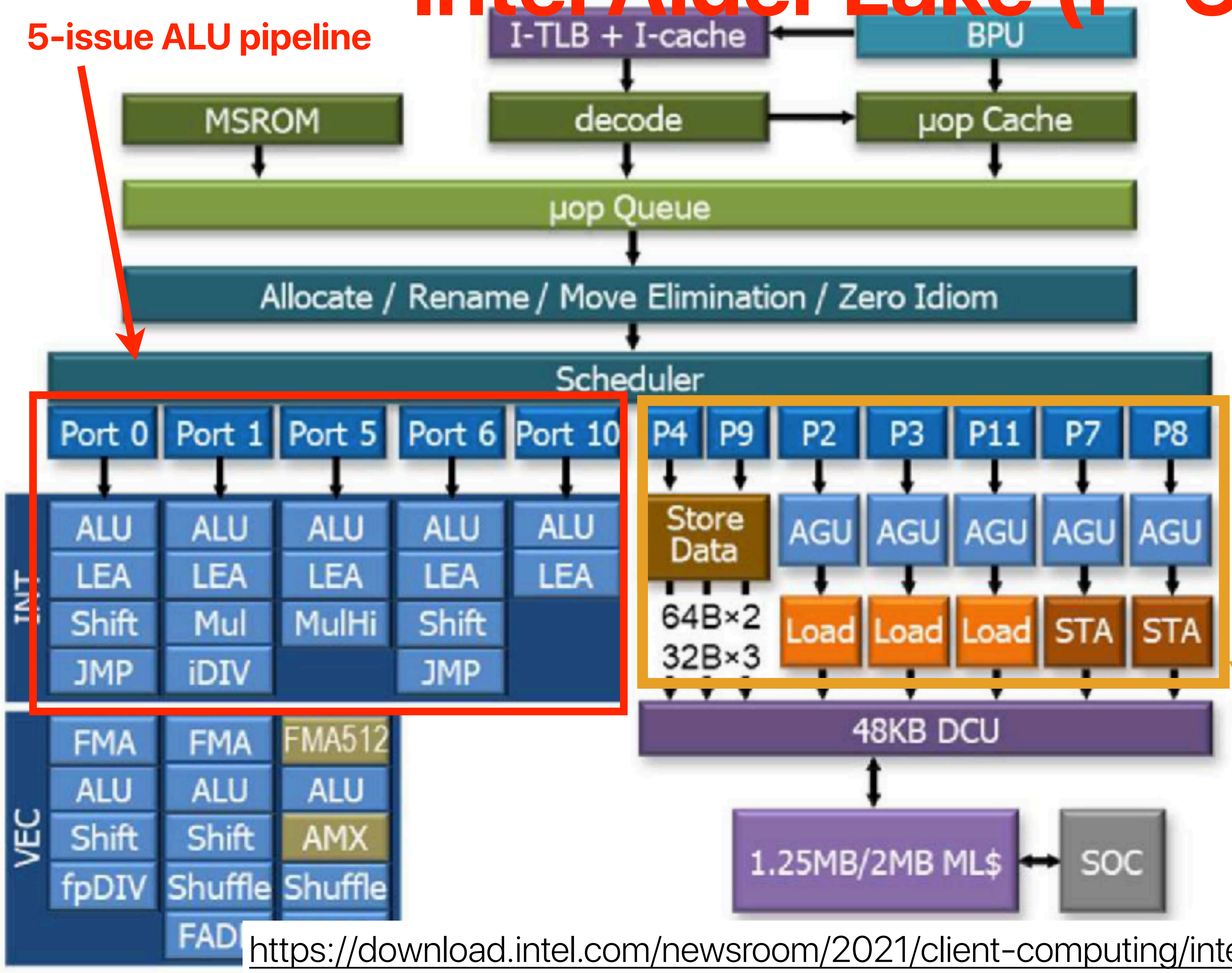
Refresh our minds! What are the characteristics of modern processors?

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
  - Tournament predictors

Intel Alder Lake (P-Core)

5-issue ALU pipeline



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

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

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

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

7-issue memory pipeline

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

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

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 and the dataset size is small enough to **fit inside the L1** cache, please identify the correct statements.
  ① There is very little performance difference between A and B
  ② B will outperform A as A has more branch instructions
  ③ A will outperform B as A has fewer cache misses
  ④ A will outperform B as A has fewer data dependance related stalls
  ⑤ A will outperform B as A has fewer dynamic instructions

  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 and the dataset size is small enough to **fit inside the L1** cache, please identify the correct statements.

① There is very little performance difference between A and B
② B will outperform A as A has more branch instructions
③ A will outperform B as A has fewer cache misses **about the same**
④ A will outperform B as A has fewer data dependance related stalls
⑤ A will outperform B as A has fewer dynamic instructions

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++;
}
```

Take a look of their instructions

A

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

```
① .L9:   cmpq  $1, 8(%rax)
②        sbbl  $-1, %edx
③        addq  $16, %rax
④        cmpq  %rdi, %rax
⑤        jne   .L9
```

If we have n iterations, we will execute 5n instructions.

B

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

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

If we have n iterations, we will execute 4n instructions.

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 and the dataset size is small enough to **fit inside the L1** cache, please identify the correct statements.

  ① There is very little performance difference between A and B

  ② B will outperform A as A has more branch instructions

  ③ A will outperform B as A has fewer cache misses **about the same**

  ④ A will outperform B as A has fewer data dependance related stalls

  ⑤ A will outperform B as A has fewer dynamic instructions **more instructions**

  A. 0

  B. 1

  C. 2

  D. 3

  E. 4

A

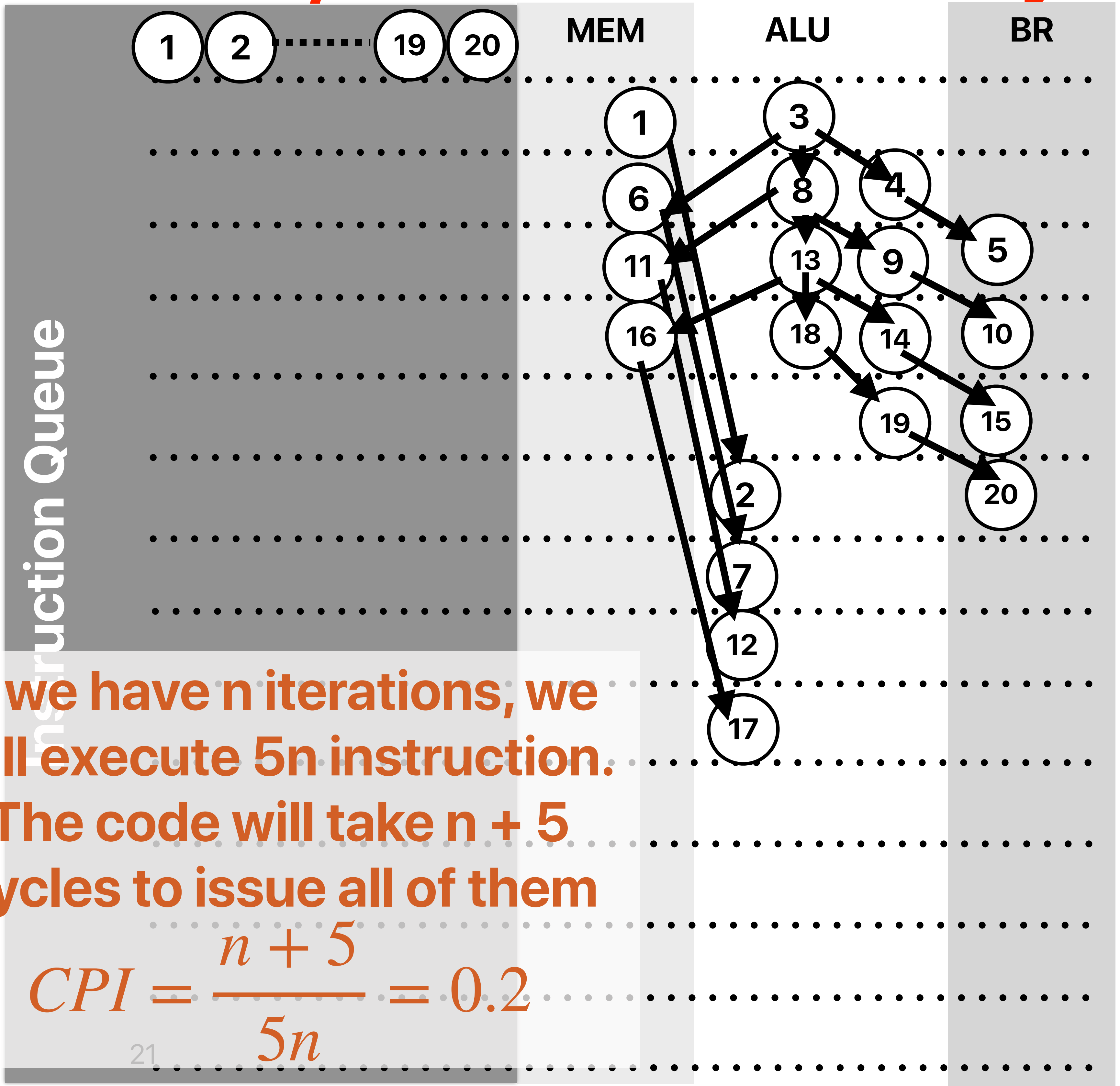
```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

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

What if we have "unlimited" fetch/issue width — "array"

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



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

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

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

What about "linked list"

- For the following C code and it's translation in x86, what's **average CPI**? Assume the current PC is already at instruction (1) and this linked list has thousands of nodes. This processor can fetch and issue **unlimited** number of instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously (**5-cycle** memory access latencies, 100% hit rate).

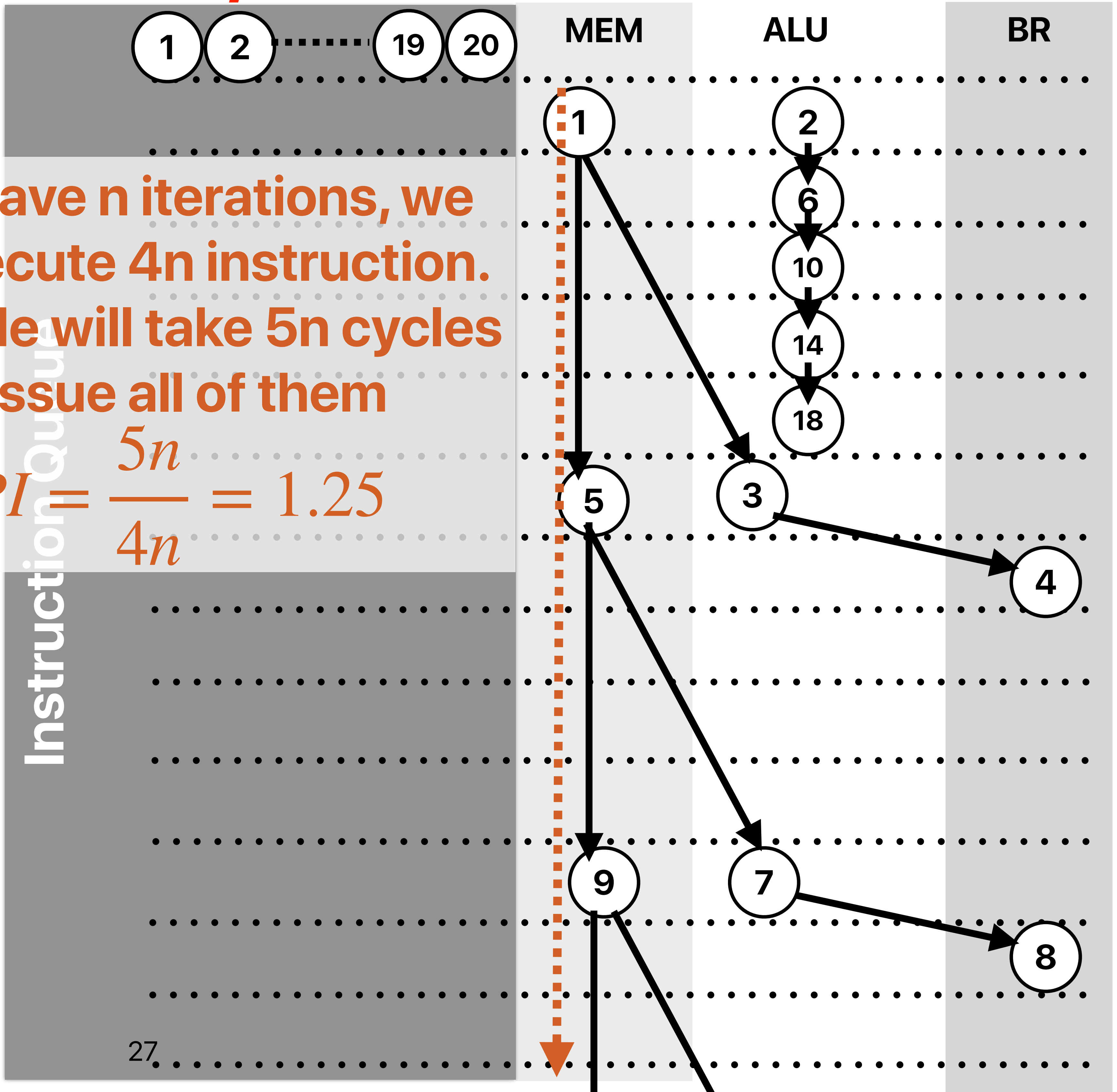
```
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL )
```

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

A. 0.5
B. 0.75
C. 1.0
D. 1.25
E. 1.5

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

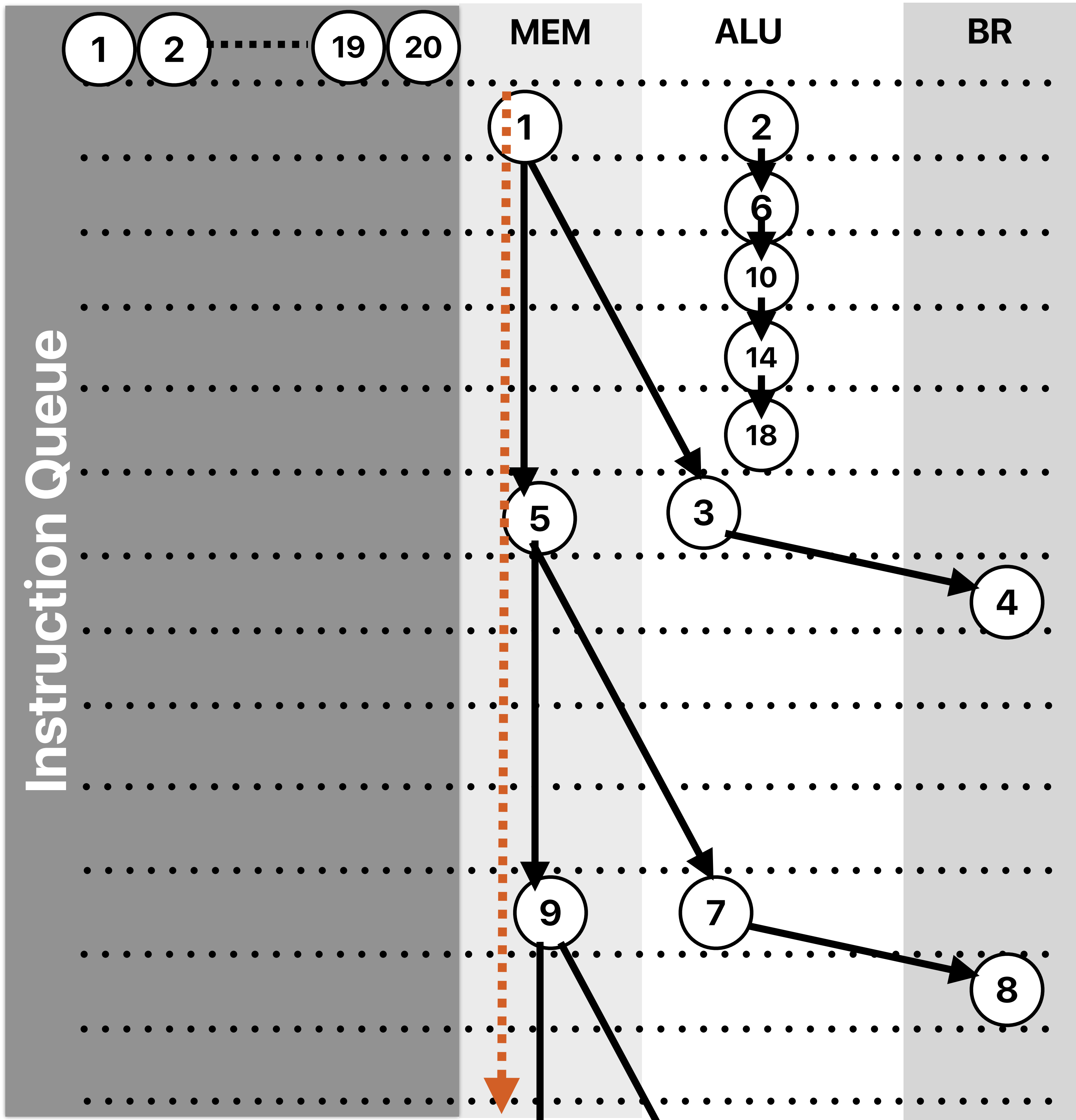
$$CPI = \frac{5n}{4n} = 1.25$$

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
```

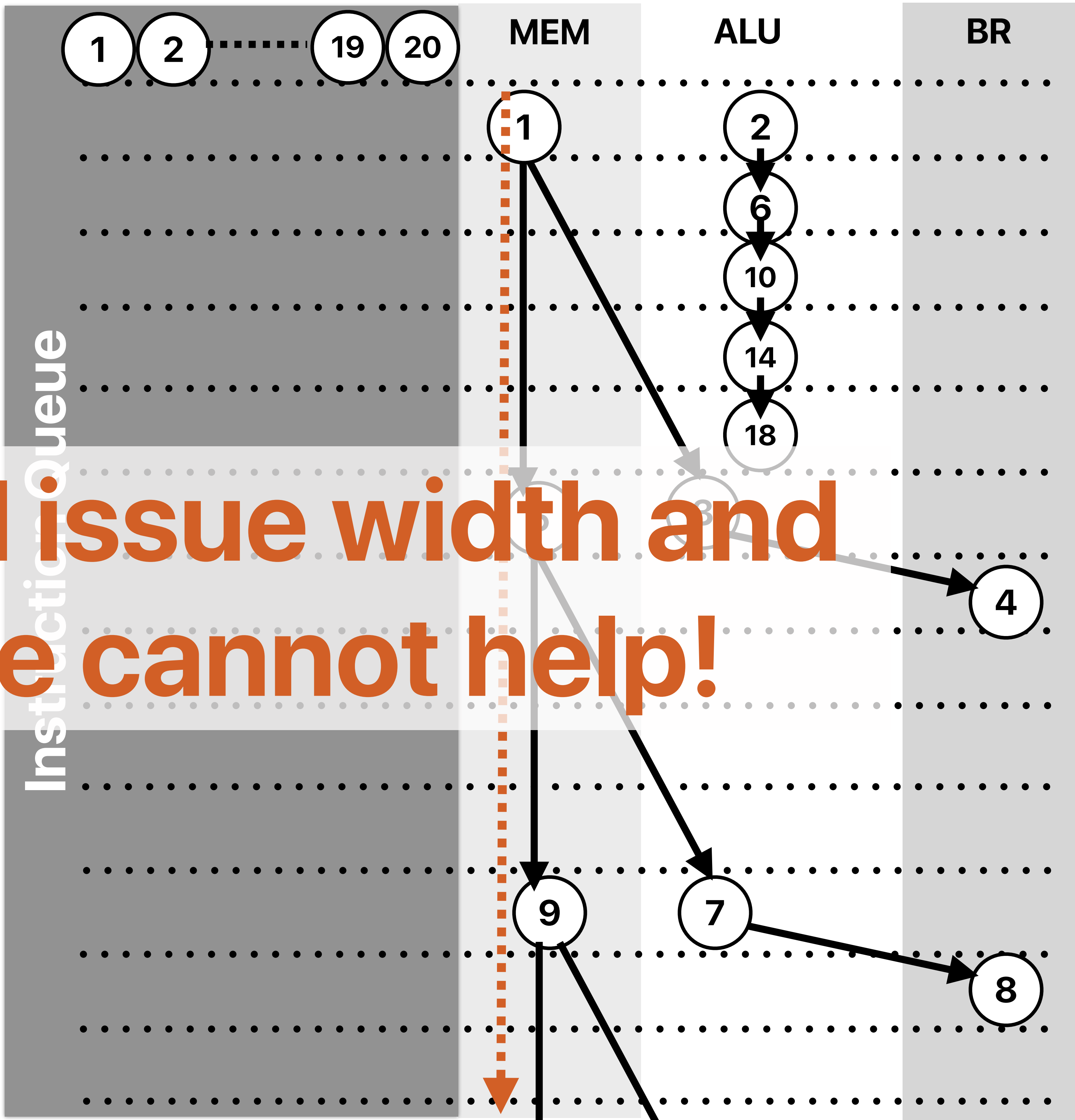


What if we have "unlimited" fetch/issue width — "linked list"

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 about "linked list"

- For the following C code and it's translation in x86, what's **average CPI**? Assume the current PC is already at instruction (1) and this linked list has thousands of nodes. This processor can fetch and issue **unlimited** number of instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously (**5-cycle** memory access latencies, 100% hit rate).

```
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
```

A. 0.5
B. 0.75
C. 1.0
[✓ D. 1.25]
E. 1.5

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 and the dataset size is small enough to **fit inside the L1** cache, please identify the correct statements.
  ① There is very little performance difference between A and B
  ② B will outperform A as A has more branch instructions
  ③ A will outperform B as A has fewer cache misses
  [✓ ④] A will outperform B as A has fewer data dependance related stalls
  ⑤ A will outperform B as A has fewer dynamic instructions

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++;
}
```

Perfectly matches our analysis!!!

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

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

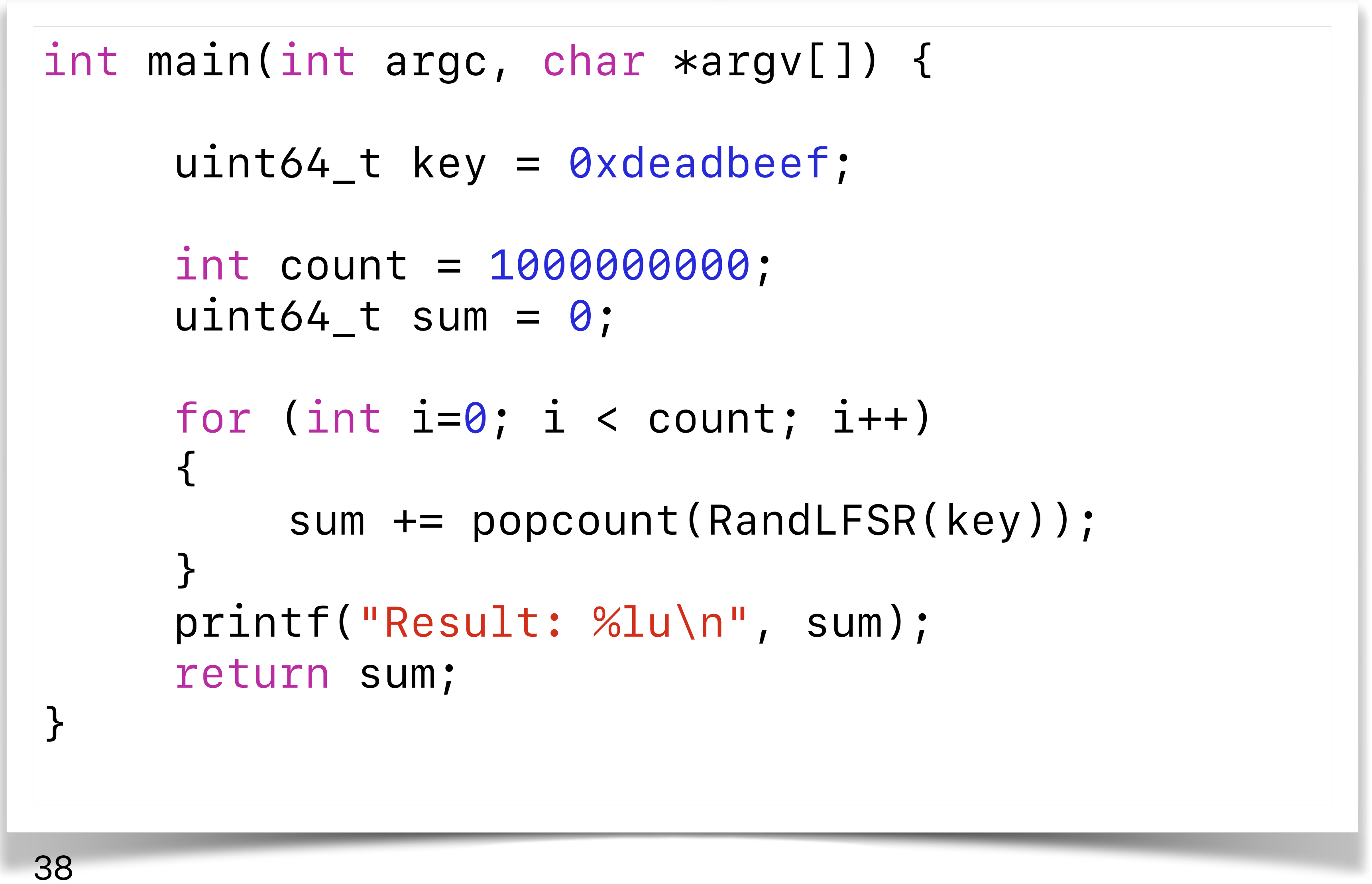
- The key to efficient code is exploiting as much instruction-level parallelism (ILP) or say higher instructions per cycle (IPC) or lower cycles per instruction (CPI) as possible
  - Avoiding data dependent operations (e.g., pointer chasing)
  - Avoiding inter-iteration dependencies (e.g., linked list, trees)

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
  `0b1011001010100001111`



```
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 ✓

```
nline 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;
}
```

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
```

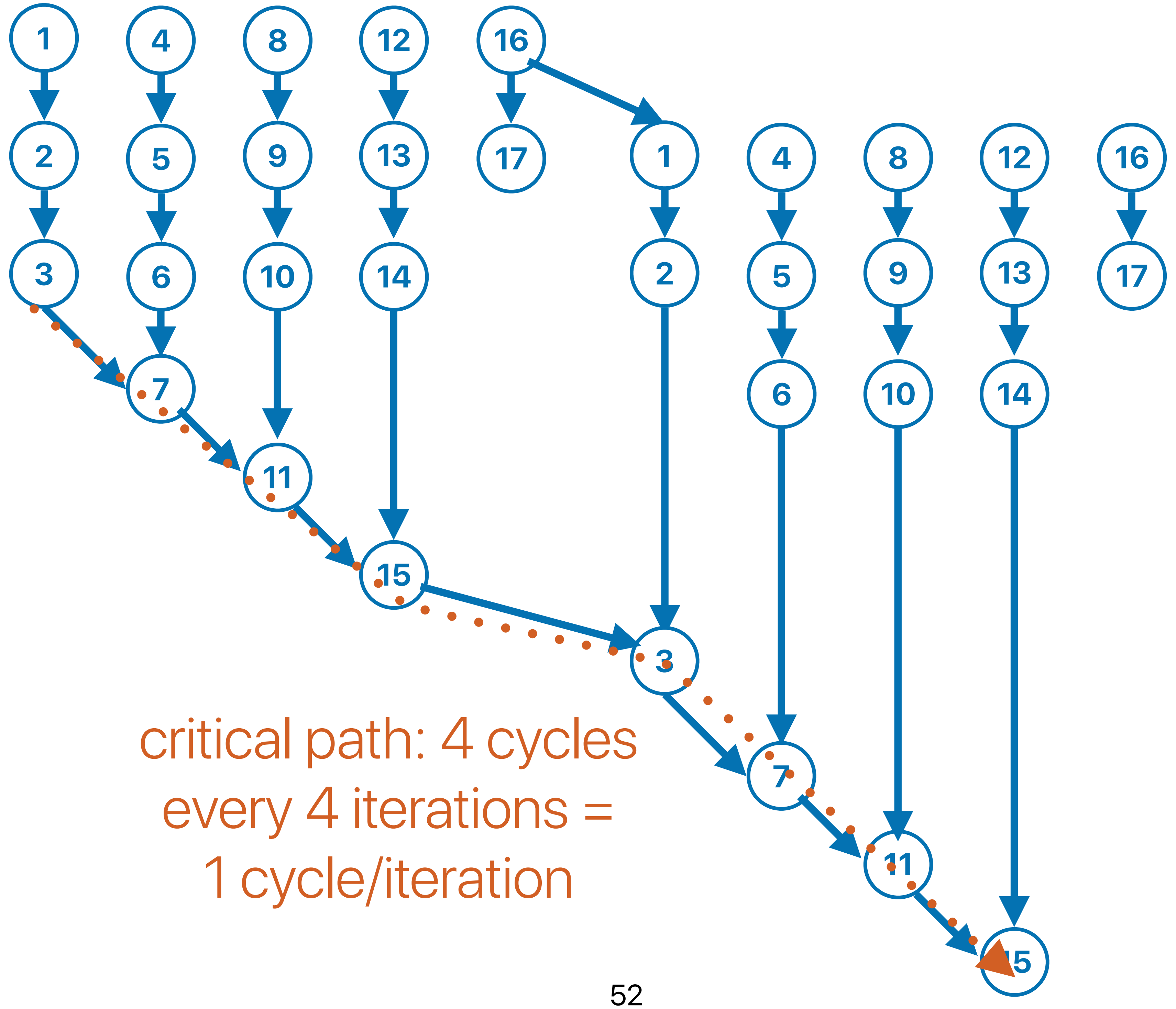
17*(n/4) = 4.25*n instructions

Only one branch for four iterations in A

Why is B better than A?

B

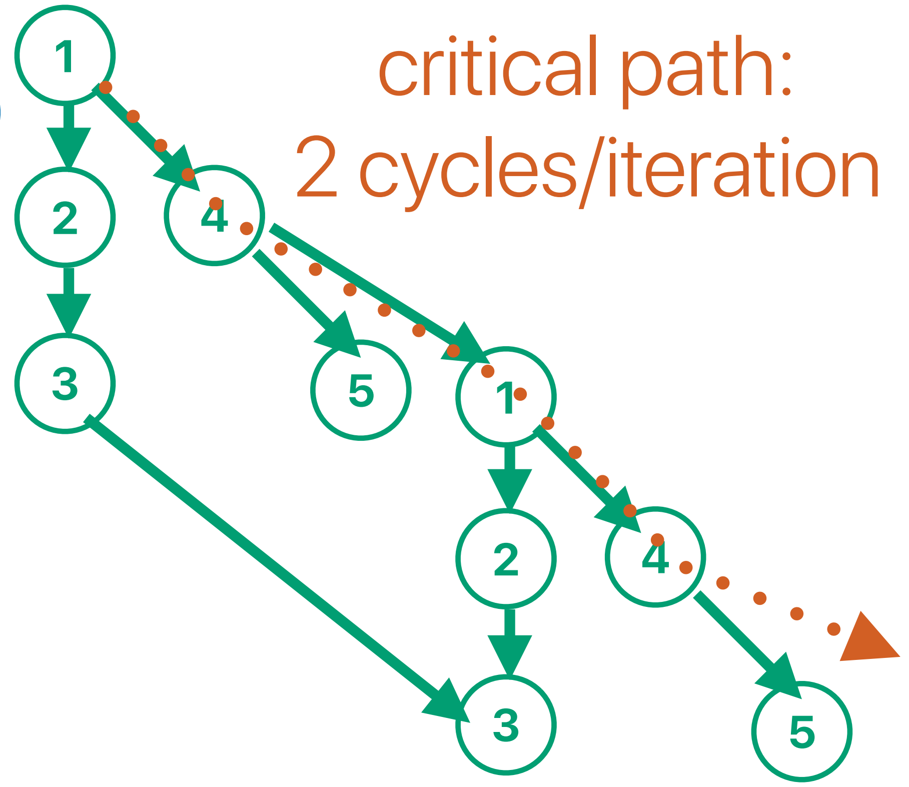
```
① movl     %ecx, %eax
② andl     $1, %eax
③ addl     %edx, %eax
④ movq     %rcx, %rdx
⑤ shrq     %rdx
⑥ andl     $1, %edx
⑦ addl     %eax, %edx
⑧ movq     %rcx, %rax
⑨ shrq     $2, %rax
⑩ andl     $1, %eax
⑪ addl     %edx, %eax
⑫ movq     %rcx, %rdx
⑬ shrq     $3, %rdx
⑭ andl     $1, %edx
⑮ addl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

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?

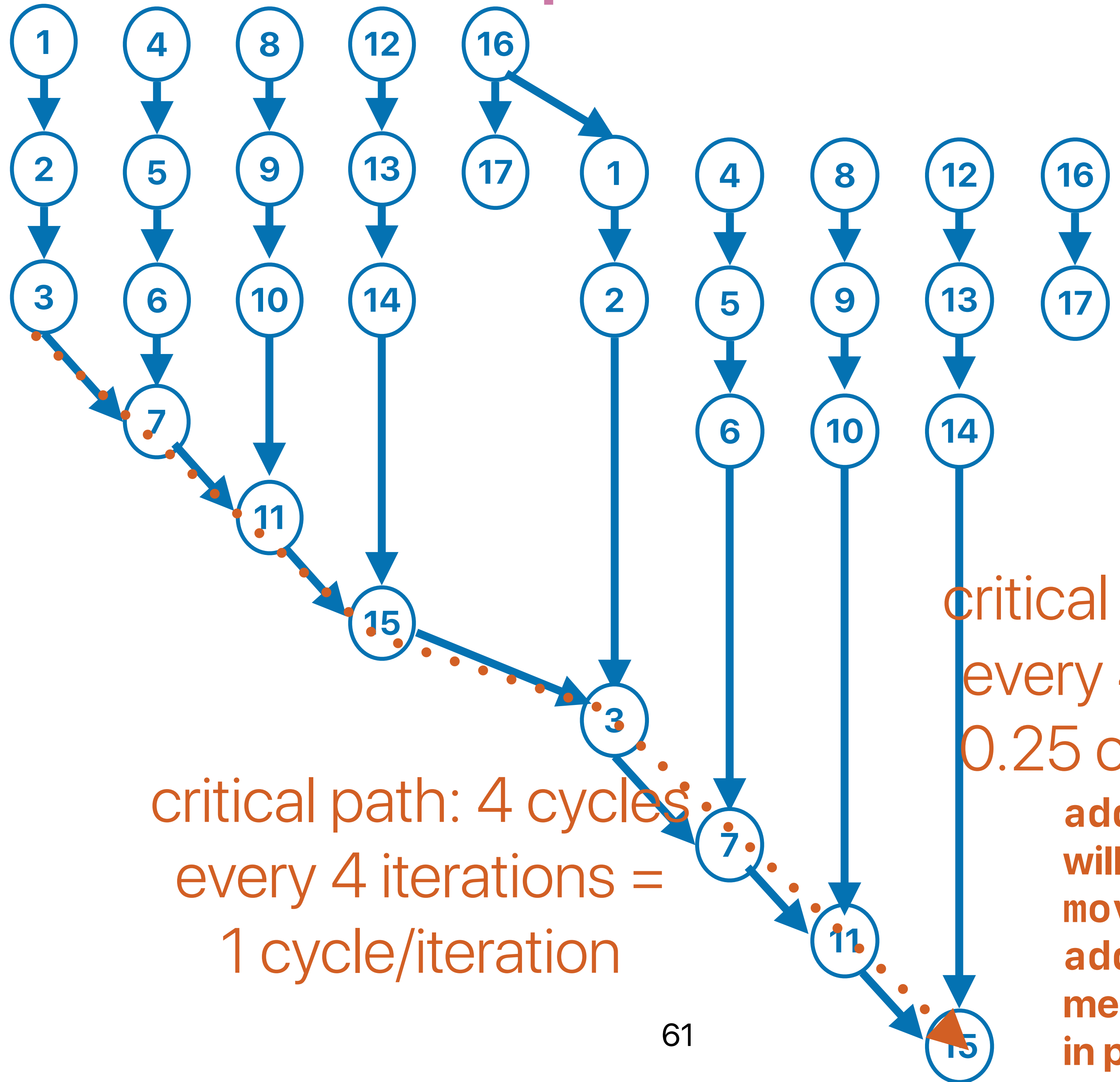
B

```
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
```

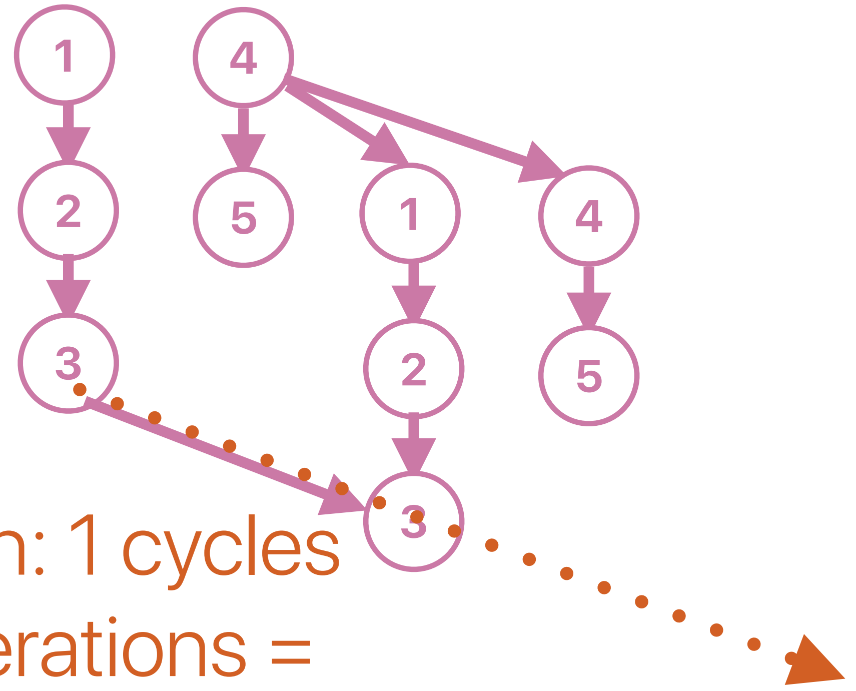
These 5 instructions represent 4 iterations!!!

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 iterat
  - 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

```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

```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;
}
```

Takeaways: programming modern processors

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

Computer
Science &
Engineering

203

つづく