Programming Modern Processors

Hung-Wei Tseng

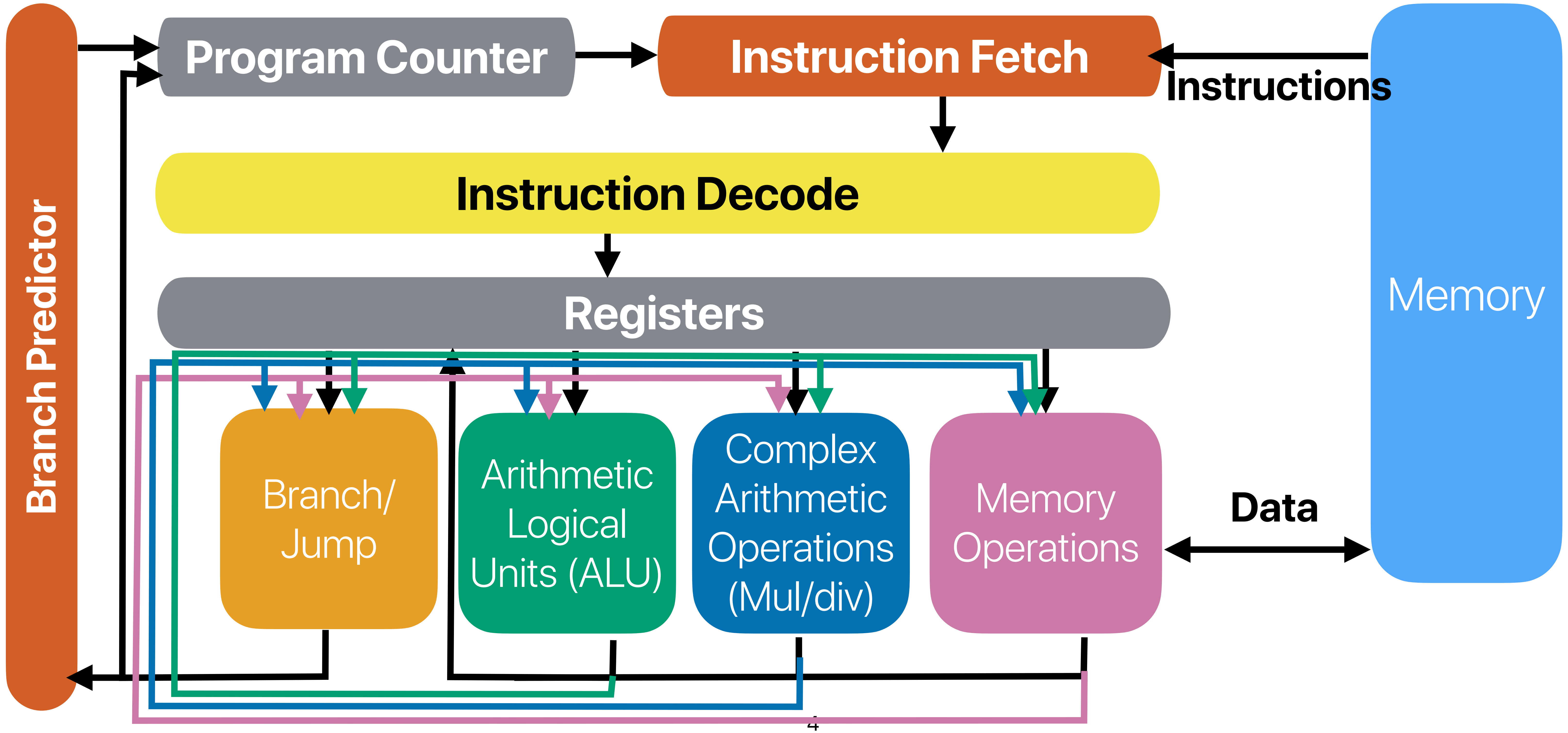
Recap: Three pipeline hazards

- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that's not yet generated or propagated when the instruction needs that

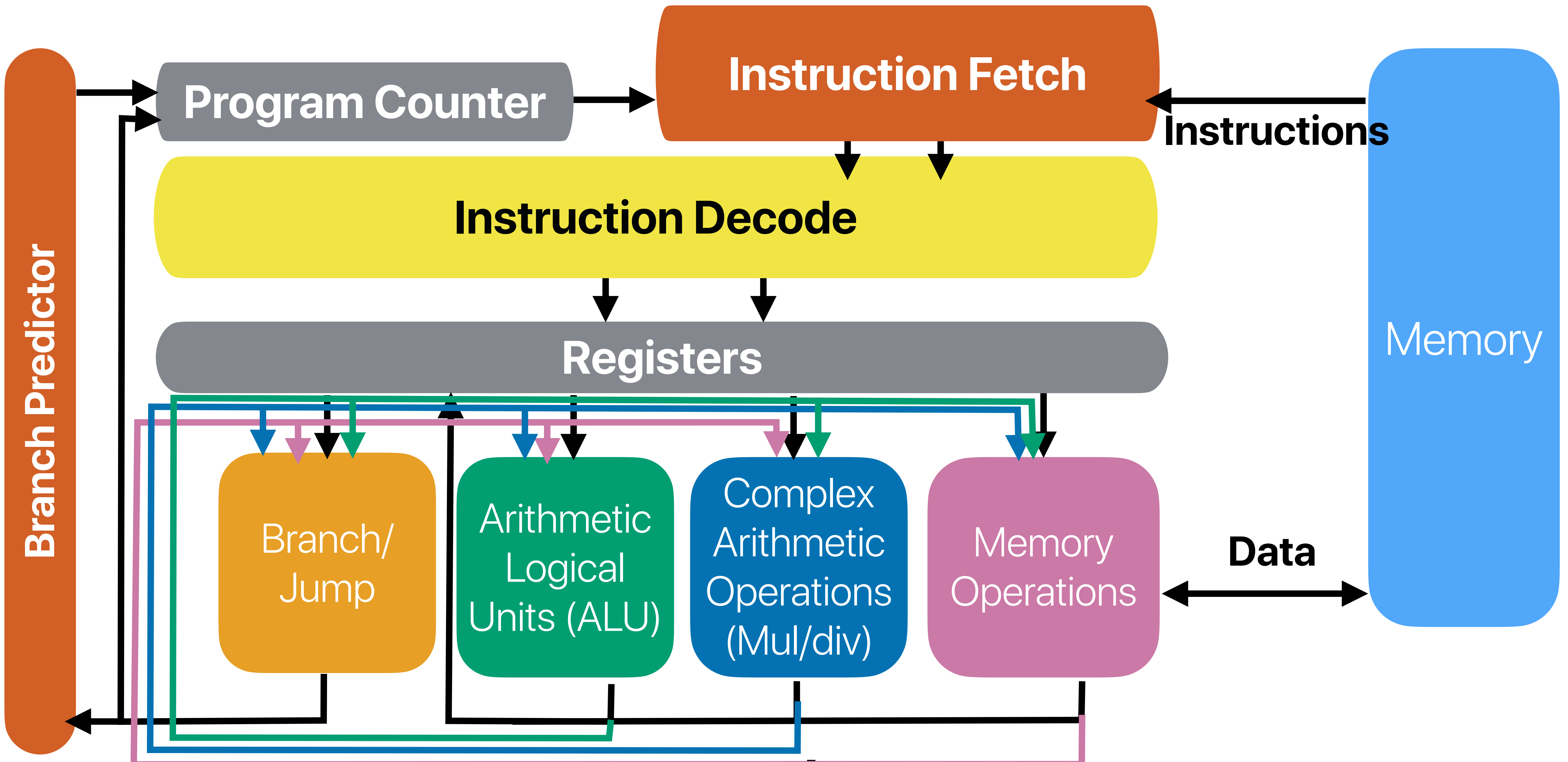
Recap: addressing hazards

- Structural hazards
  - Stall
  - Modify hardware design
- Control hazards
  - Stall
  - Static prediction
  - Dynamic prediction
- Data Hazards
  - Stall
  - Data forwarding
  - Dynamic instruction scheduling

Recap: Data "forwarding"



Super Scalar



Superscalar

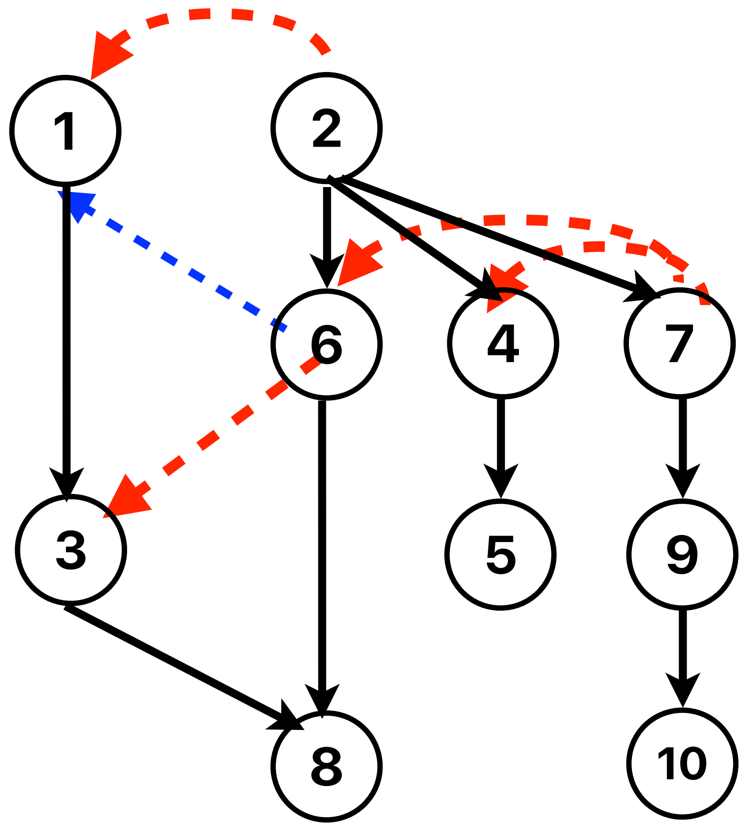
- Since we have many functional units now, we should fetch/decode more instructions each cycle so that we can have more instructions to issue!
- Super-scalar: fetch/decode/issue more than one instruction each cycle
    - **Fetch width:** how many instructions can the processor fetch/decode each cycle
    - **Issue width**: how many instructions can the processor issue each cycle
- The theoretical CPI should now be

$$\frac{1}{min(issue\ width, fetch\ width, decode\ width)}$$

False dependencies

- We are still limited by **false dependencies**
- They are not "true" dependencies because they don't have an arrow in data dependency graph
  - WAR (Write After Read): a later instruction overwrites the source of an earlier one
    - 2 and 1, 6 and 3, 7 and 4, 7 and 6
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 6 and 1

```
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
```



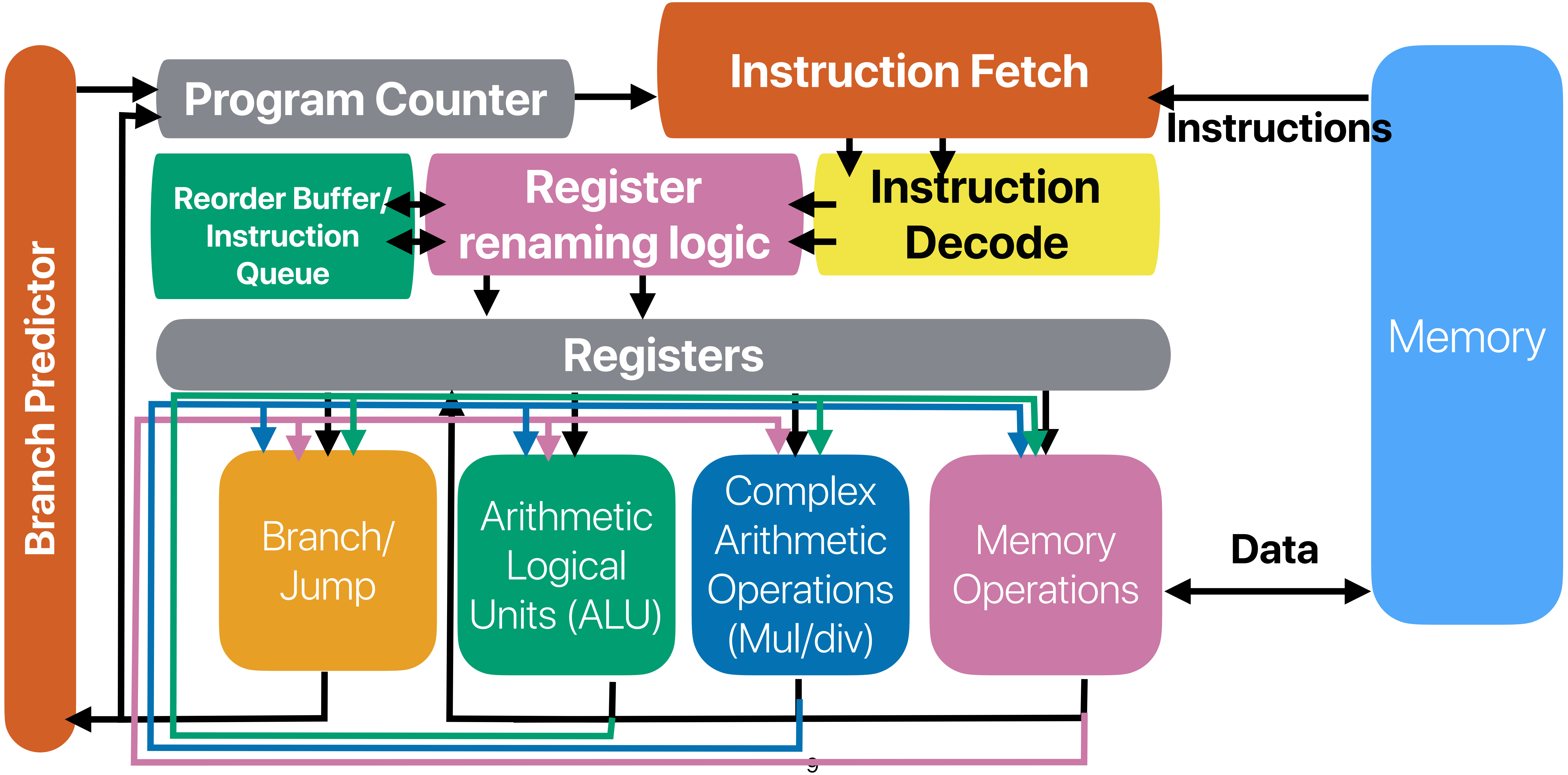
What if we can use more registers...

```
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
```

```
① movl    (%rdi), %ecx
② addq    $4, %rdi, %t0
③ addl    %ecx, %eax, %t1
④ cmpq    %rdx, %t0
⑤ jne     .L3
⑥ movl    (%t0), %t2
⑦ addq    $4, %t0, %t3
⑧ addl    %t1, %t2, %t4
⑨ cmpq    %rdx, %t3
⑩ jne     .L3
```

All false dependencies are gone!!!

Register renaming



Speculative Execution

- Exceptions (e.g. divided by 0, page fault) may occur anytime
  - A later instruction cannot write back its own result otherwise the architectural states won't be correct
- Hardware can schedule instruction across branch instructions with the help of branch prediction
  - Fetch instructions according to the branch prediction
  - However, branch predictor can never be perfect
- Execute instructions across branches
  - Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Execute an instruction all operands are ready (the values of depending physical registers are generated)
  - Store results in **reorder buffer** before the processor knows if the instruction is going to be executed or not.

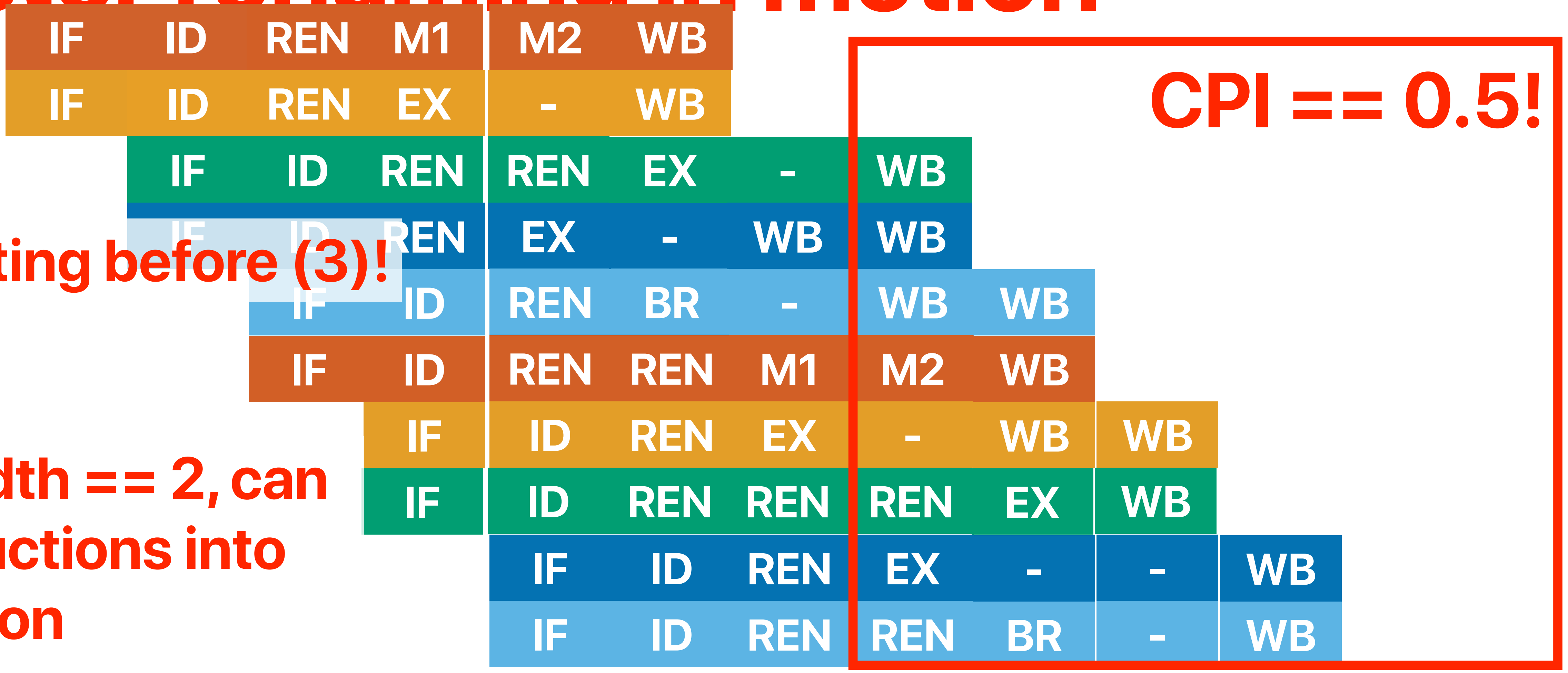
Register renaming in motion

```
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
```

(4) is now executing before (3)!

Assume issue width == 2, can only put 2 instructions into execution

CPI == 0.5!



	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	~~addl P1, %eax, P3~~
4	~~cmpq %rdx, P2~~
5	~~jne .L3~~
6	~~movl (P2), P4~~
7	~~addq $4, P2, P5~~
8	~~addl P4, P3, P6~~
9	~~cmpq %rdx, P5~~
10	~~jne .L3~~

	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	1		1	P9			
P5	1		1	P10			

Outline

- Out-of-order, Dynamic instruction scheduling
- Programming on Modern Processor

Register renaming

2-issue: Only 2 of them can have instructions

```
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
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5										
6										
7										
8										
9										
10										
11										

Register renaming

2-issue: Only 2 of them can have instructions

```
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
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5					(1)	(4)				
6										
7										
8										
9										
10										
11										

Register renaming

```
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
```

2-issue: Only 2 of them can have instructions

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6										
7										
8										
9										
10										
11										

Register renaming

```
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
```

2-issue: Only 2 of them can have instructions

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7										
8										
9										
10										
11										

Register renaming

2-issue: Only 2 of them can have instructions

```
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
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8										
9										
10										
11										

Register renaming

```
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
```

2-issue: Only 2 of them can have instructions

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8					(6)	(9)				(3)(4)
9										
10										
11										

Register renaming

2-issue: Only 2 of them can have instructions

```
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
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8					(6)	(9)				(3)(4)
9						(8)		(10)		(5)(6)
10										
11										

Register renaming

```
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
```

2-issue: Only 2 of them can have instructions

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8					(6)	(9)				(3)(4)
9						(8)		(10)		(5)(6)
10										(7)(8)
11										

Register renaming

2-issue: Only 2 of them can have instructions

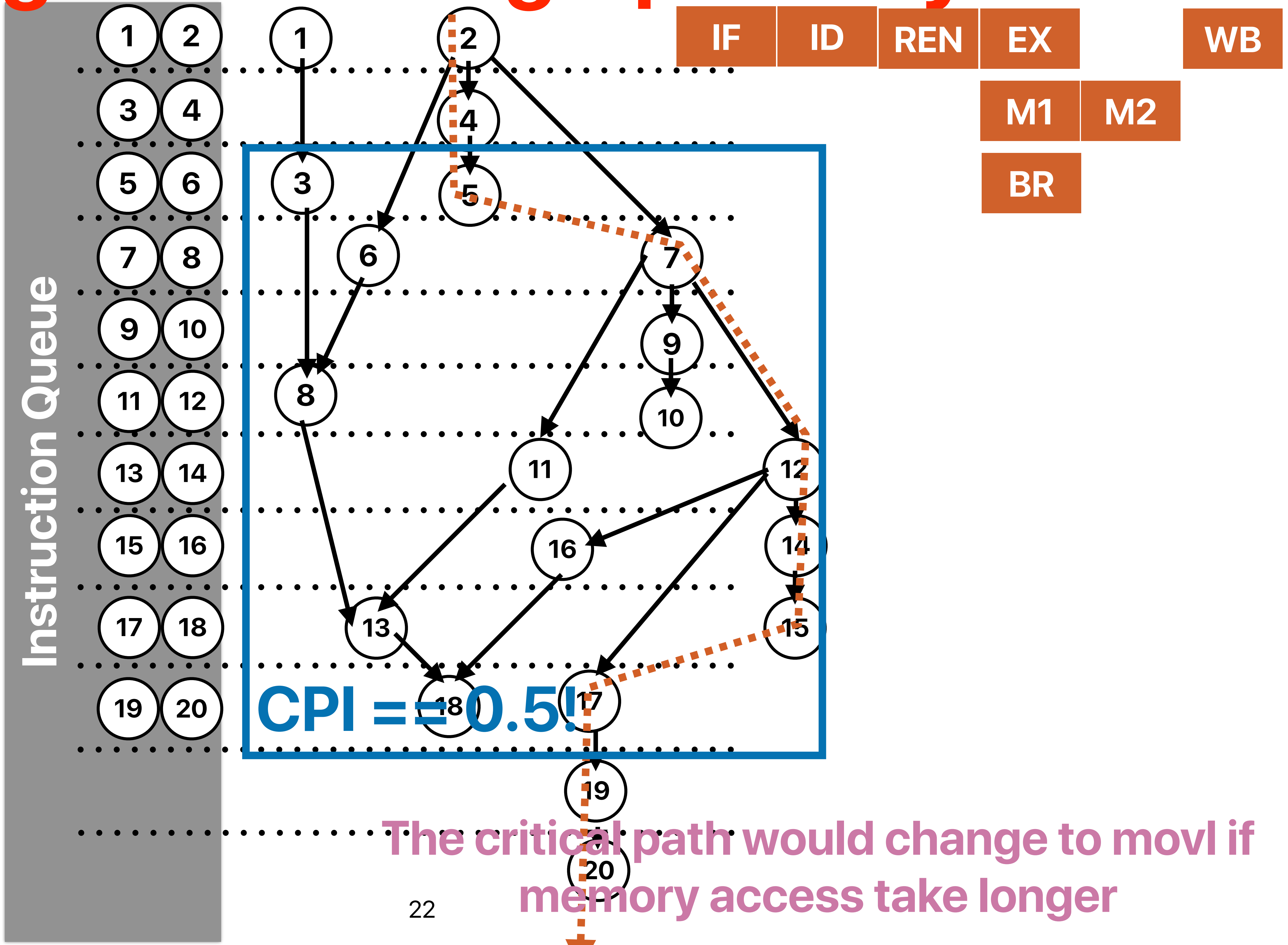
```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8					(6)	(9)				(3)(4)
9						(8)		(10)		(5)(6)
10										(7)(8)
11										(9)(10)

CPI == 0.5!

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
```



CPI == 0.5!

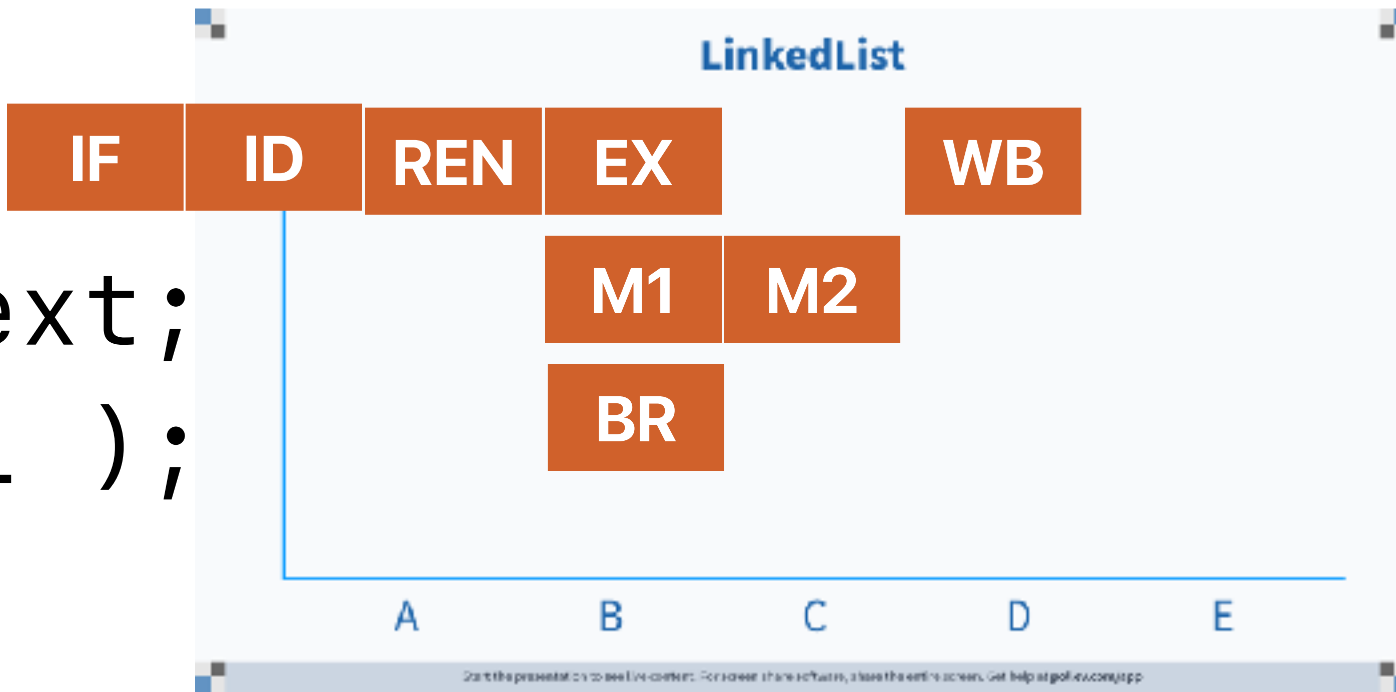
The critical path would change to movl if memory access take longer

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

What about "linked list"

- Assume the current PC is already at instruction (1) and this linked list has only three nodes. This processor can fetch and issue 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously. Which of the following C state of the code snippet determines the performance?

```
A.do {
B.     number_of_nodes++;
C.     current = current->next;
D.} while ( current != NULL );
```



Register renaming

2-issue: Only 2 of them can have instructions

```
① .L3:addq    $8, %rdi
②      movq    (%rdi), %rdi
③      addl    $1, %eax
④      testq   %rdi, %rdi
⑤      jne     .L3
⑥ .L3:addq    $8, %rdi
⑦      movq    (%rdi), %rdi
⑧      addl    $1, %eax
⑨      testq   %rdi, %rdi
⑩      jne     .L3
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)			(1)				
5	(9)(10)	(7)(8)	(5)(6)	(2)		(3)	(1)			
6		(9)(10)	(7)(8)		(2)		(3)			(1)
7			(9)(10)			(4)				(2)
8						(6)	(4)	(5)		(3)
9				(7)			(6)		(5)	(4)
10					(7)	(8)				(5)(6)
11						(9)	(8)			(7)
							(9)	(10)		(8)
									(10)	(9)(10)

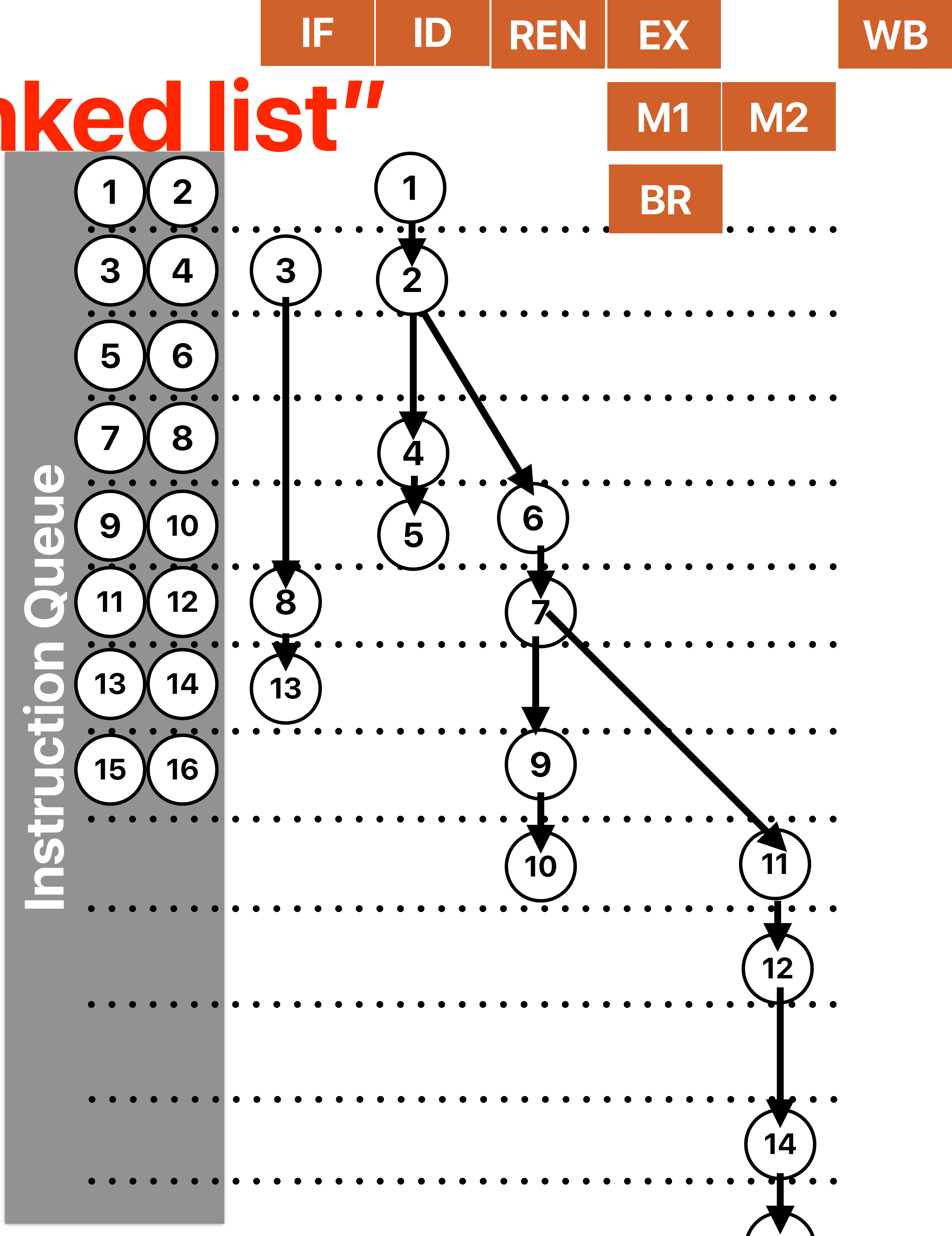
What about "linked list"

Static instructions

```
① .L3:    addq    $8, %rdi
②         movq    (%rdi), %rdi
③         addl    $1, %eax
④         testq   %rdi, %rdi
⑤         jne     .L3
```

Dynamic instructions

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



What about "linked list"

Performance determined by the critical path

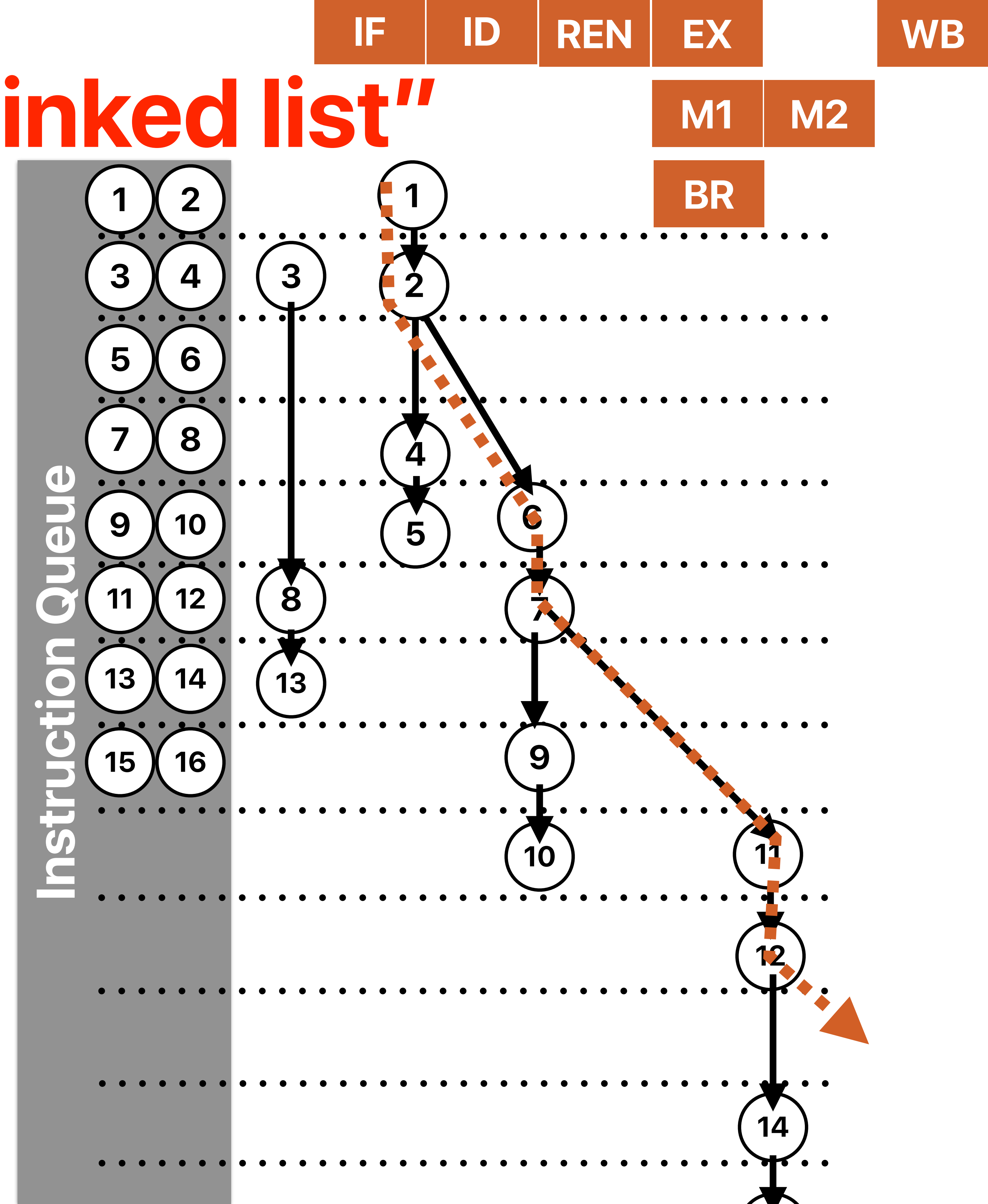
4 cycles each iteration

5 instructions per iteration

$$CPI = \frac{4}{5} = 0.8$$

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

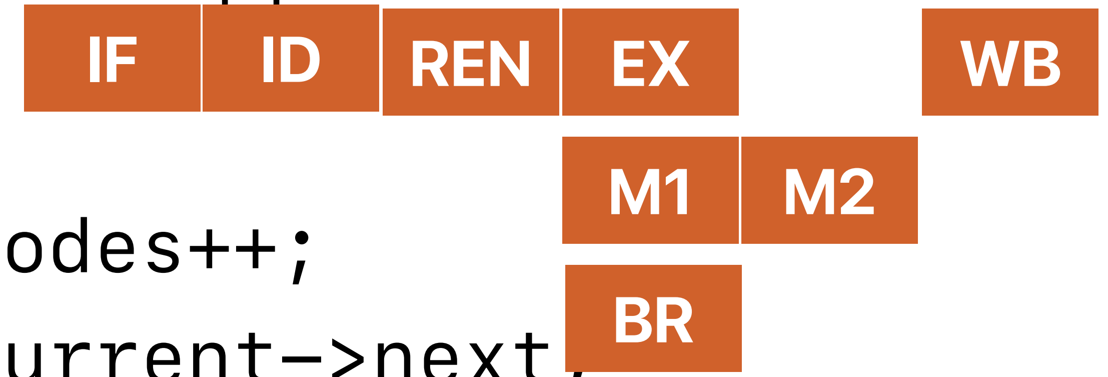
```
① .L3:     addq     $8, %rdi
②          movq     (%rdi), %rdi
③          addl     $1, %eax
④          testq    %rdi, %rdi
⑤          jne      .L3
```



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

What about "linked list"

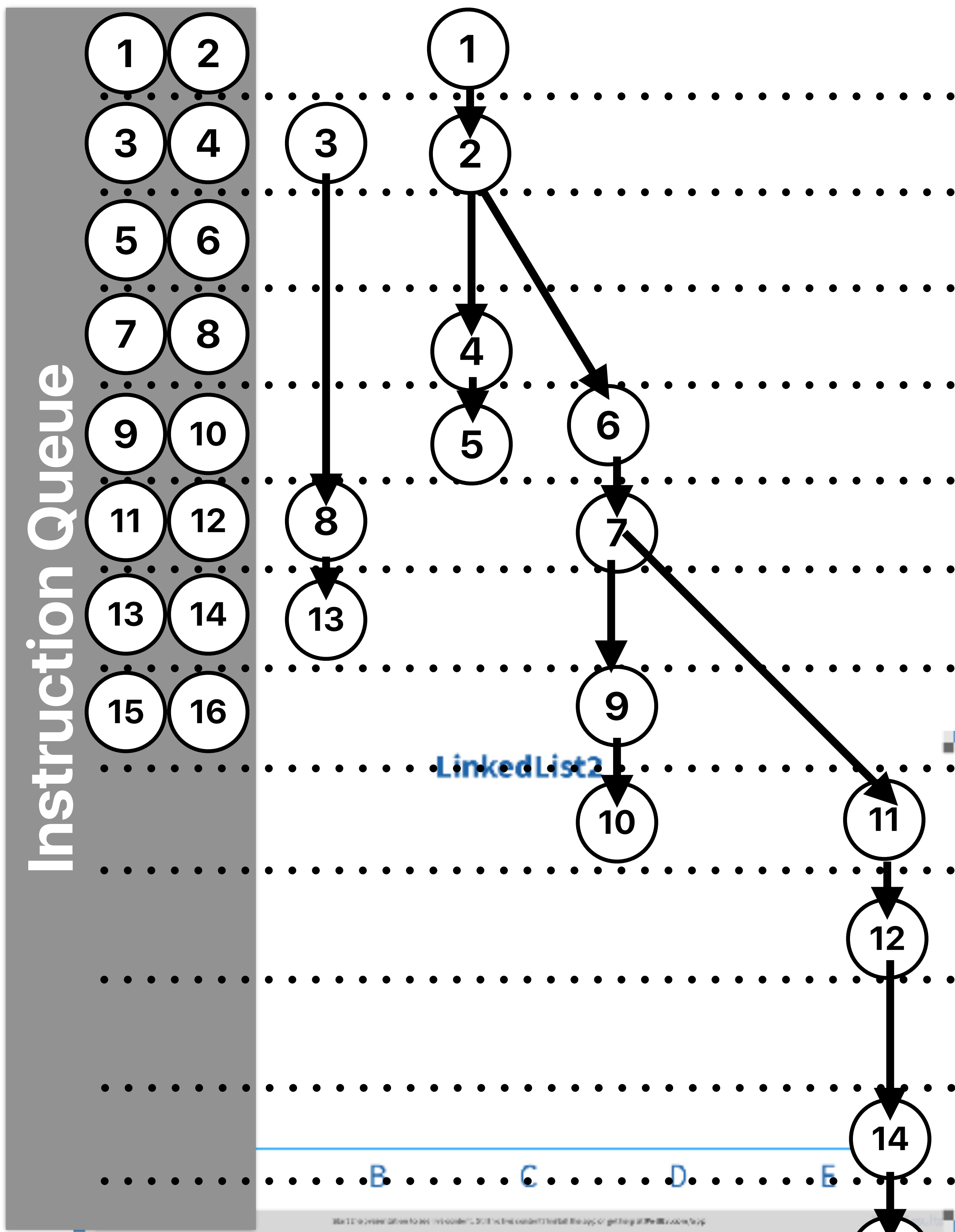
- For the following C code and it's translation in x86, **what's average CPI?** Assume the current PC is already at instruction (1) and this linked list has thousands of nodes. This processor can fetch and issue 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously.



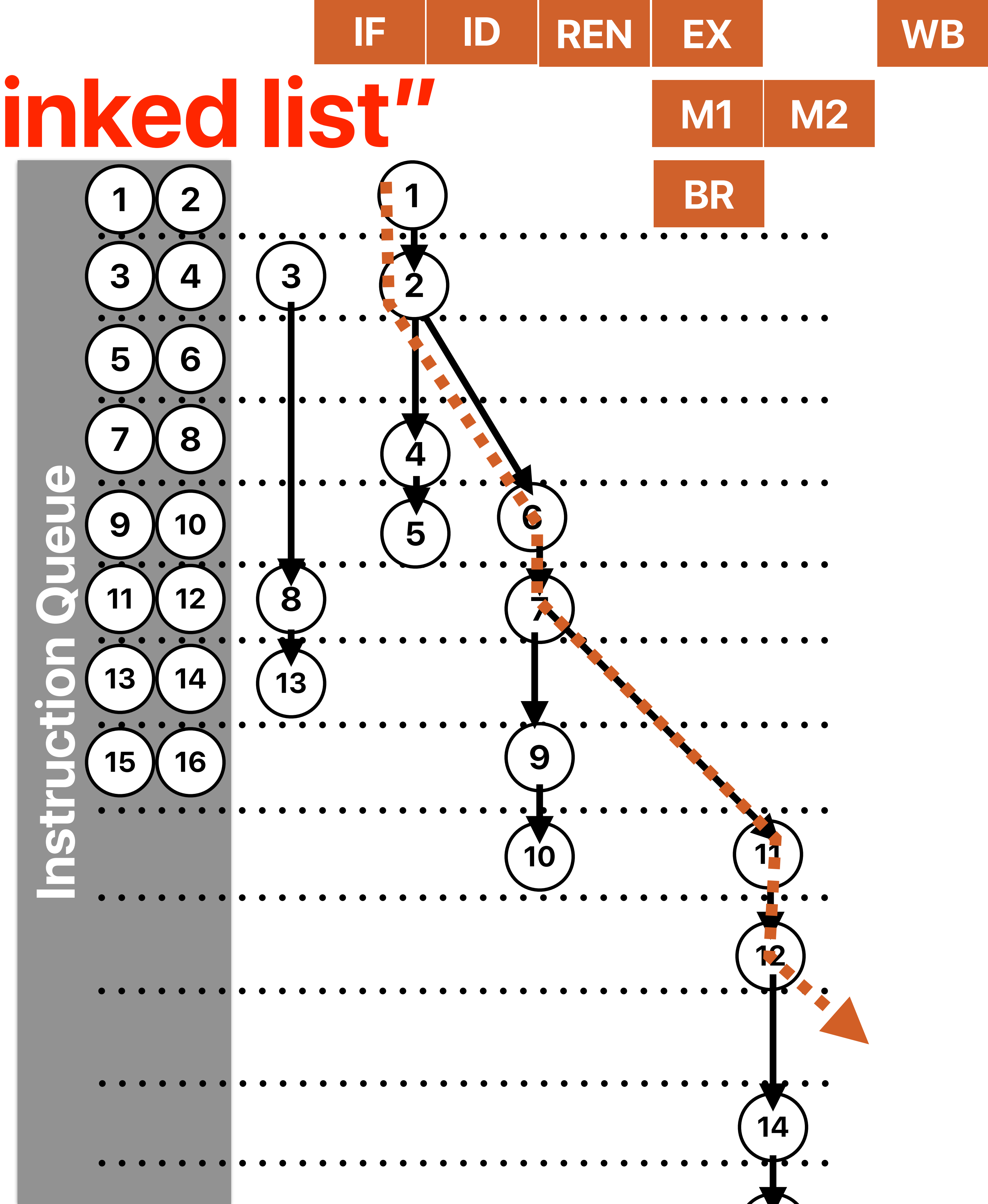
```
do {
     number_of_nodes++;
     current = current->next;
} while ( current != NULL )
```

A. 0.5
B. 0.6
C. 0.7
D. 0.8
E. 0.9

```
① .L3:    addq    $8, %rdi
②         movq    (%rdi), %rdi
③         addl    $1, %eax
④         testq   %rdi, %rdi
⑤         jne     .L3
```



What about "linked list"



Performance determined by the critical path

4 cycles each iteration

5 instructions per iteration

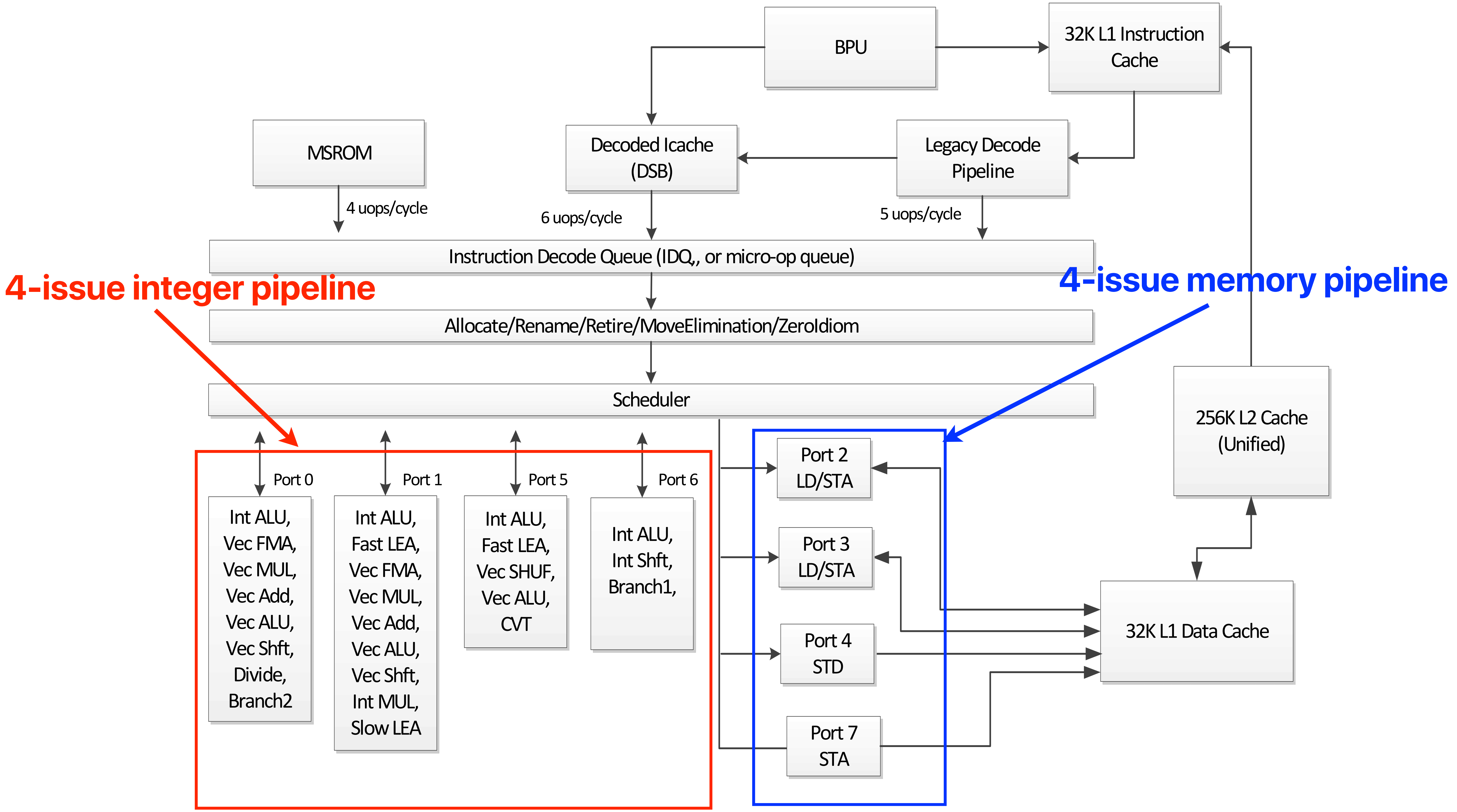
$$CPI = \frac{4}{5} = 0.8$$

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

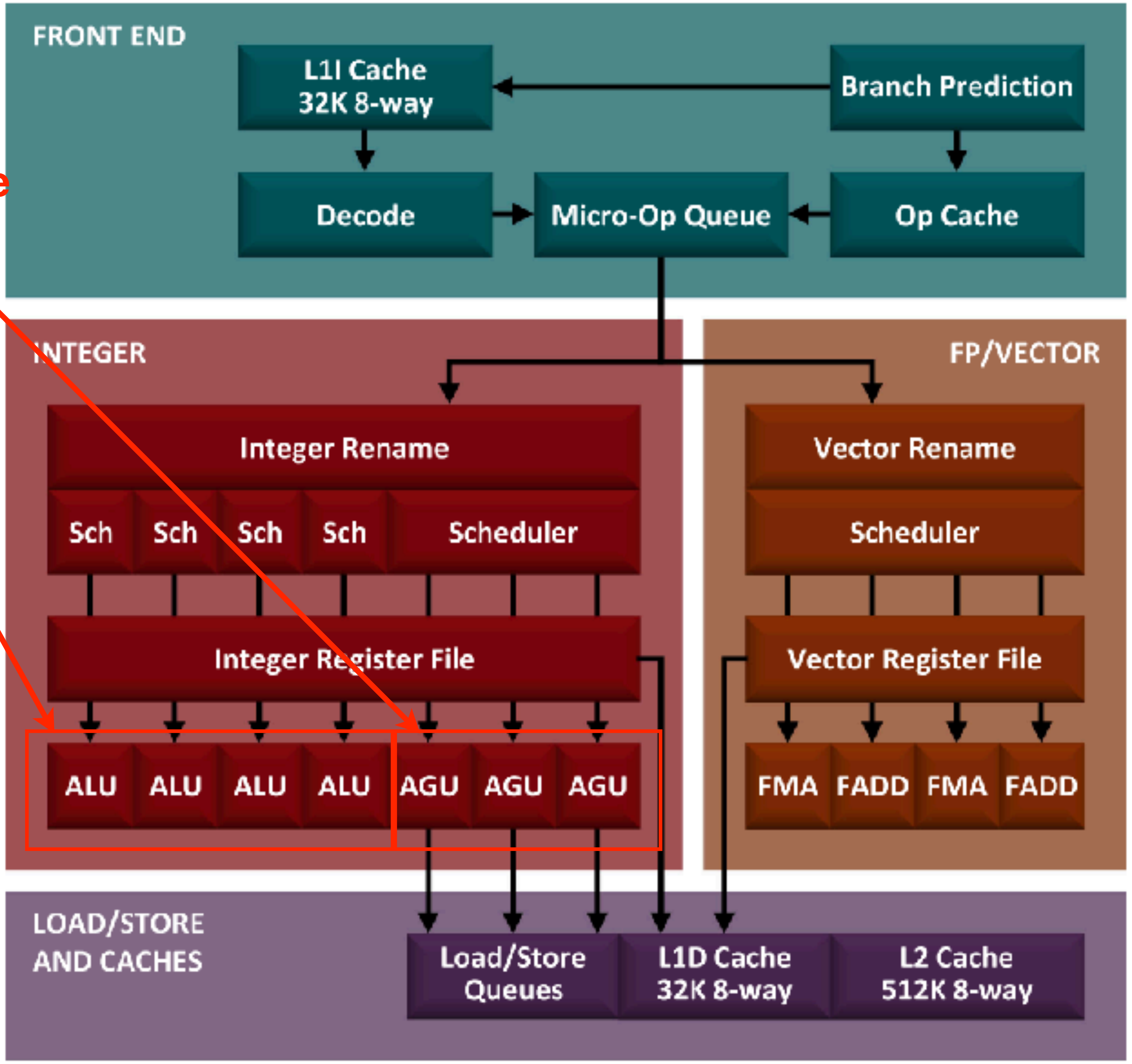
```
① .L3:    addq    $8, %rdi
②         movq    (%rdi), %rdi
③         addl    $1, %eax
④         testq   %rdi, %rdi
⑤         jne     .L3
```

The pipelines of Modern Processors

Intel Skylake


BPU
32K L1 Instruction Cache
MSROM
Decoded Icache (DSB)
Legacy Decode Pipeline
4 uops/cycle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue (IDQ,, or micro-op queue)
4-issue integer pipeline
4-issue memory pipeline
Allocate/Rename/Retire/MoveElimination/ZeroIdiom
Scheduler
256K L2 Cache (Unified)
Port 0
Port 1
Port 5
Port 6
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch1,
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
32K L1 Data Cache

AMD Zen 2 (RyZen 3000 Series)



3-issue memory pipeline

4-issue integer pipeline

$$MinCPI = \frac{1}{7}$$

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

$$MinMEMInst.CPI = \frac{1}{3}$$

Recap: Intel Skylake

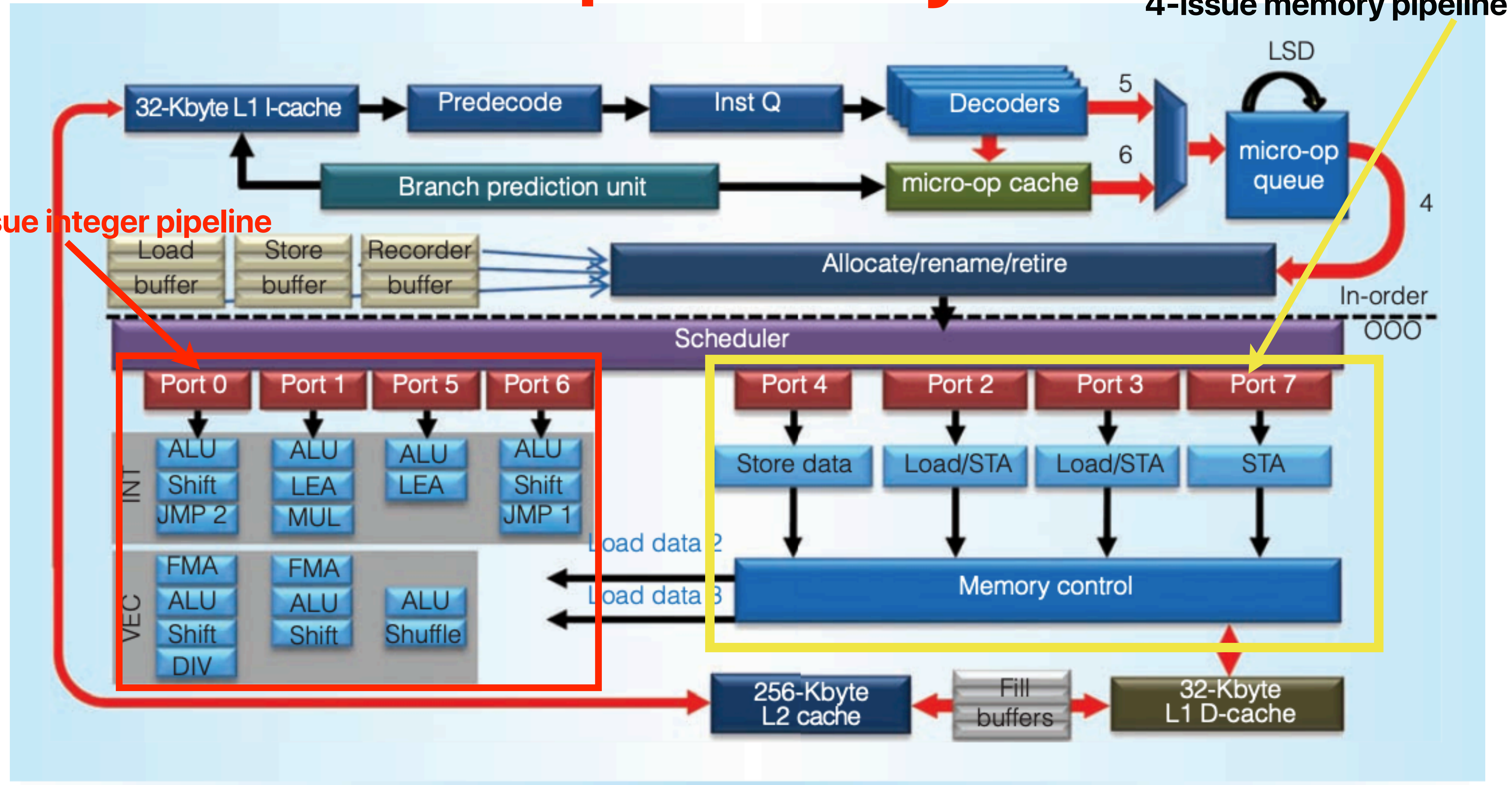
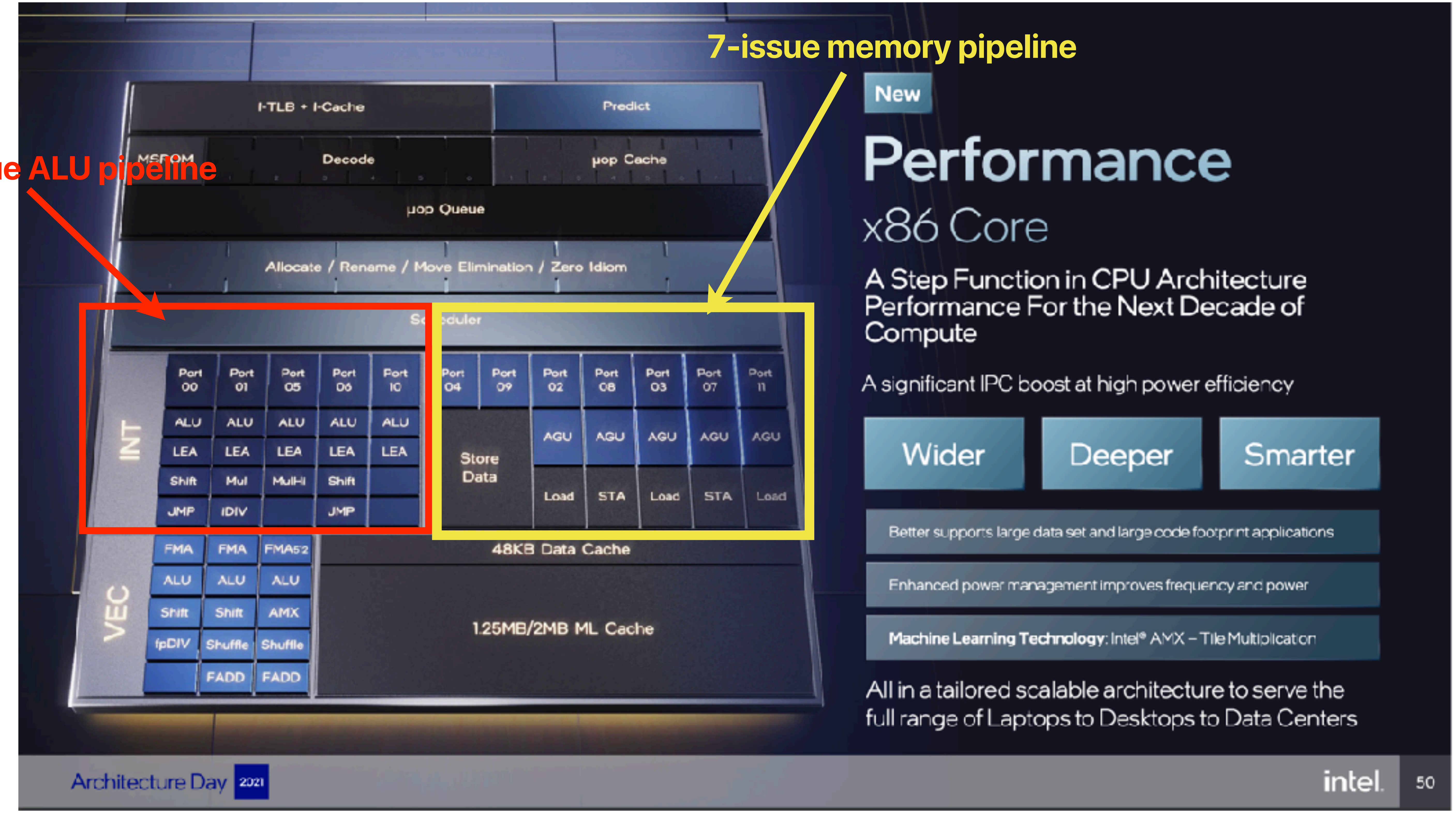


Figure 4. Skylake core block diagram.

Intel Alder Lake



New

Performance

x86 Core

A Step Function in CPU Architecture Performance For the Next Decade of Compute

A significant IPC boost at high power efficiency

Wider | Deeper | Smarter

Better supports large data set and large code footprint applications

Enhanced power management improves frequency and power

Machine Learning Technology: Intel® AMX – Tile Multiplication

All in a tailored scalable architecture to serve the full range of Laptops to Desktops to Data Centers

Architecture Day 2021

intel. 50

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

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
- Branch predictors

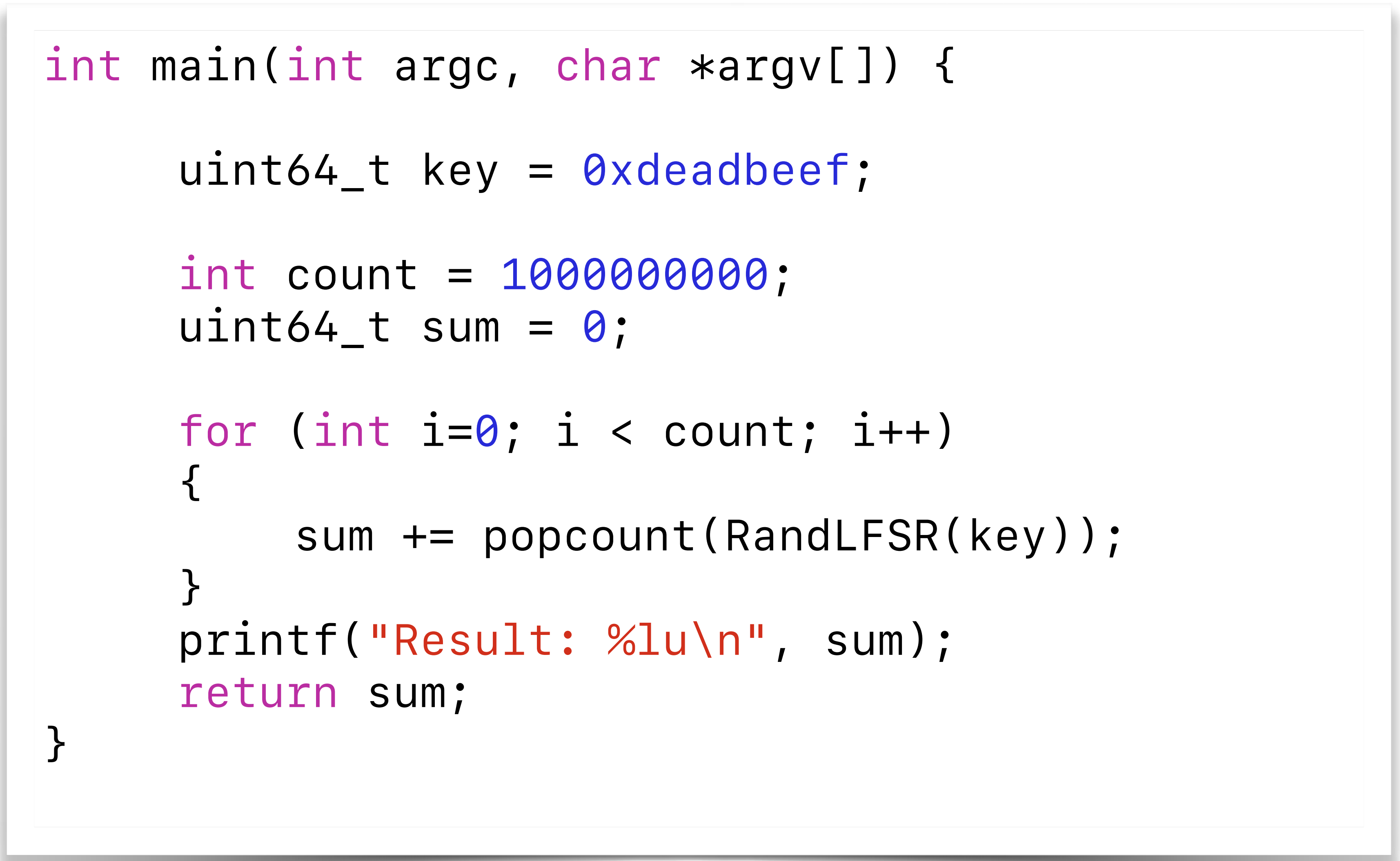
Performance Programming on Modern Processors

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: pop count

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  Input: 9487
  Output: 7
  Explanation: 9487's binary representation is 0b10010100001111



```
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;
  }
```

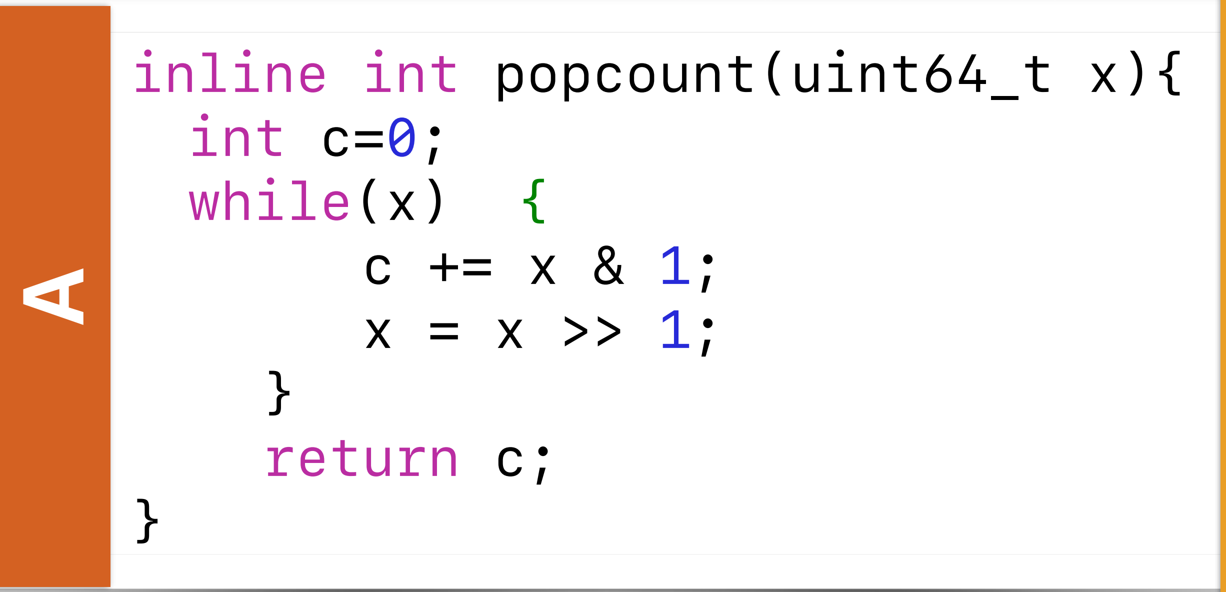
E

```
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;
  }
```

B C D

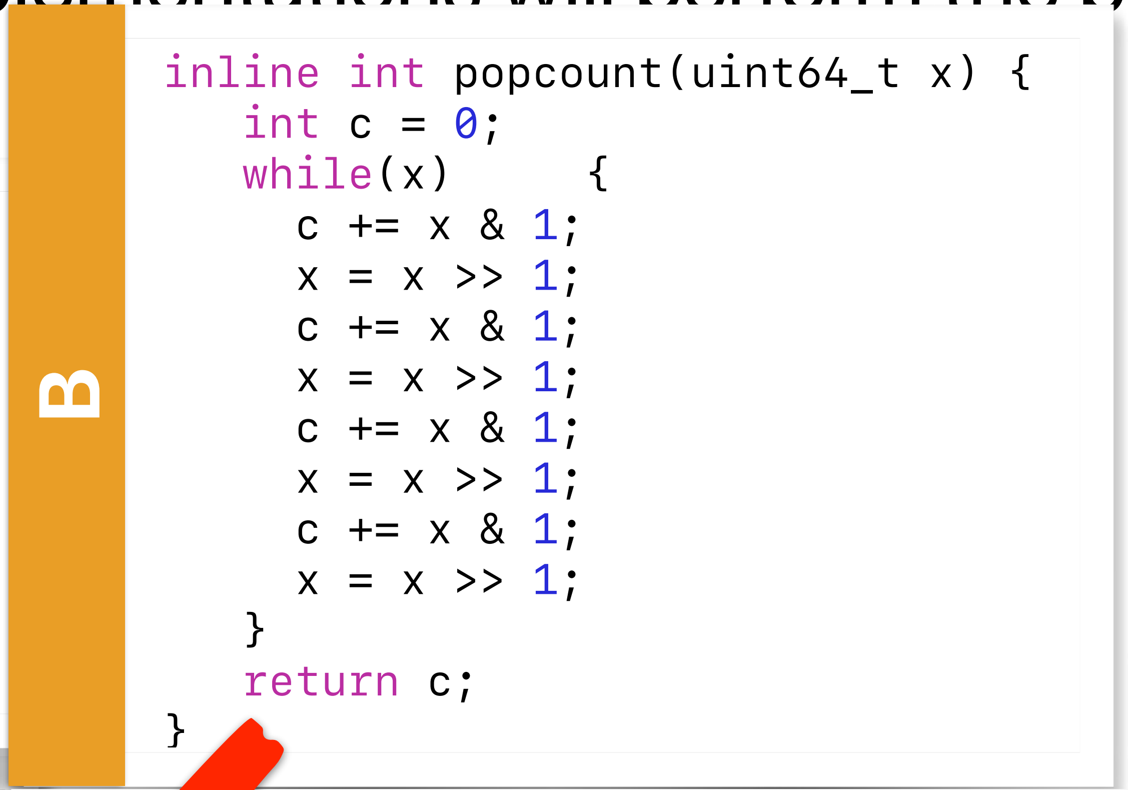
Five implementations

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



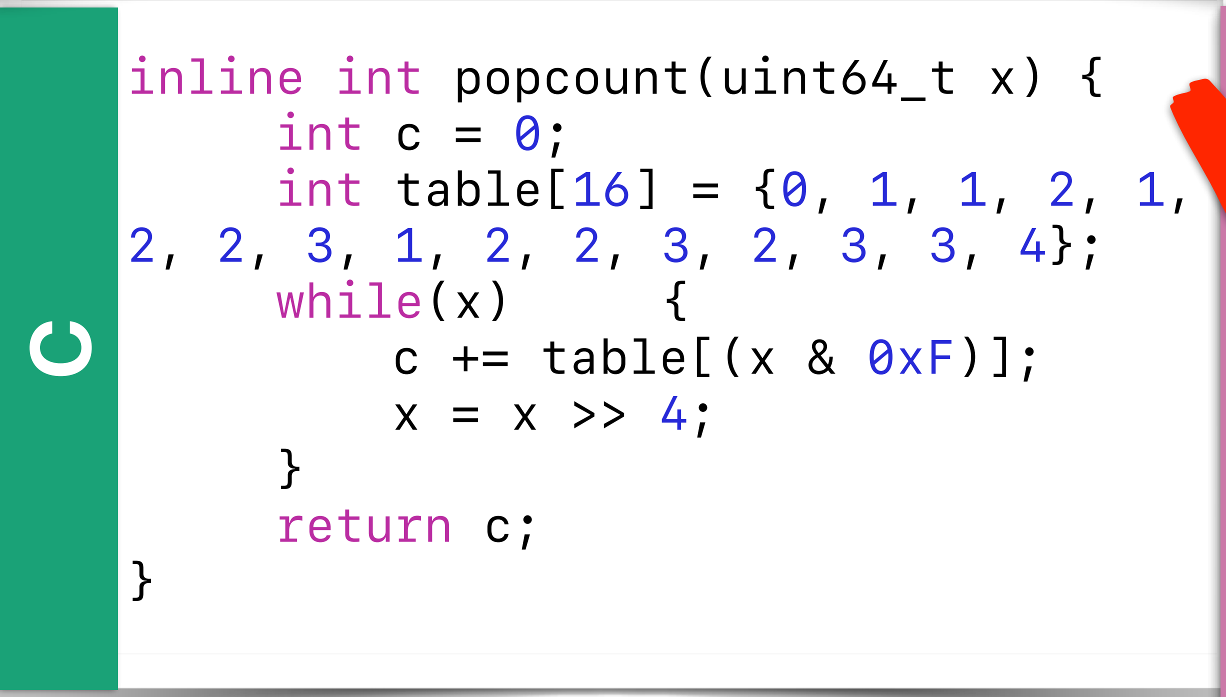
A

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



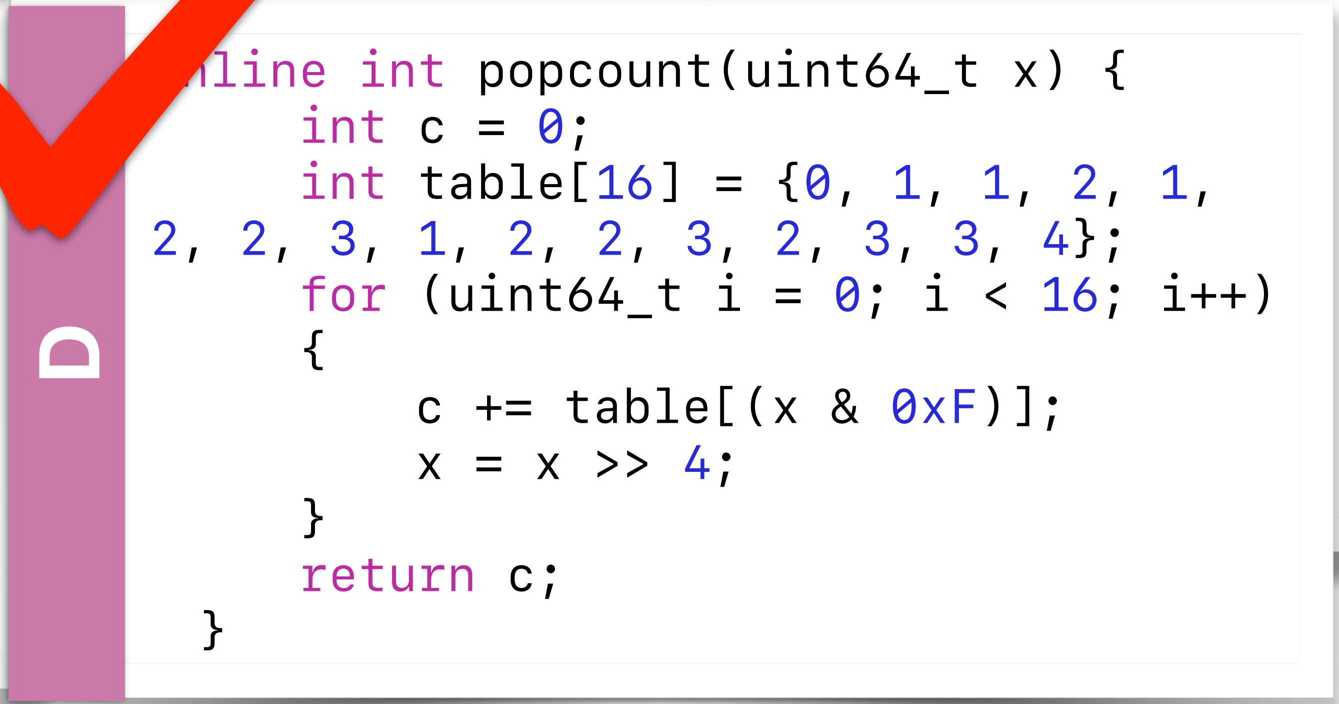
B

```
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;
}
```



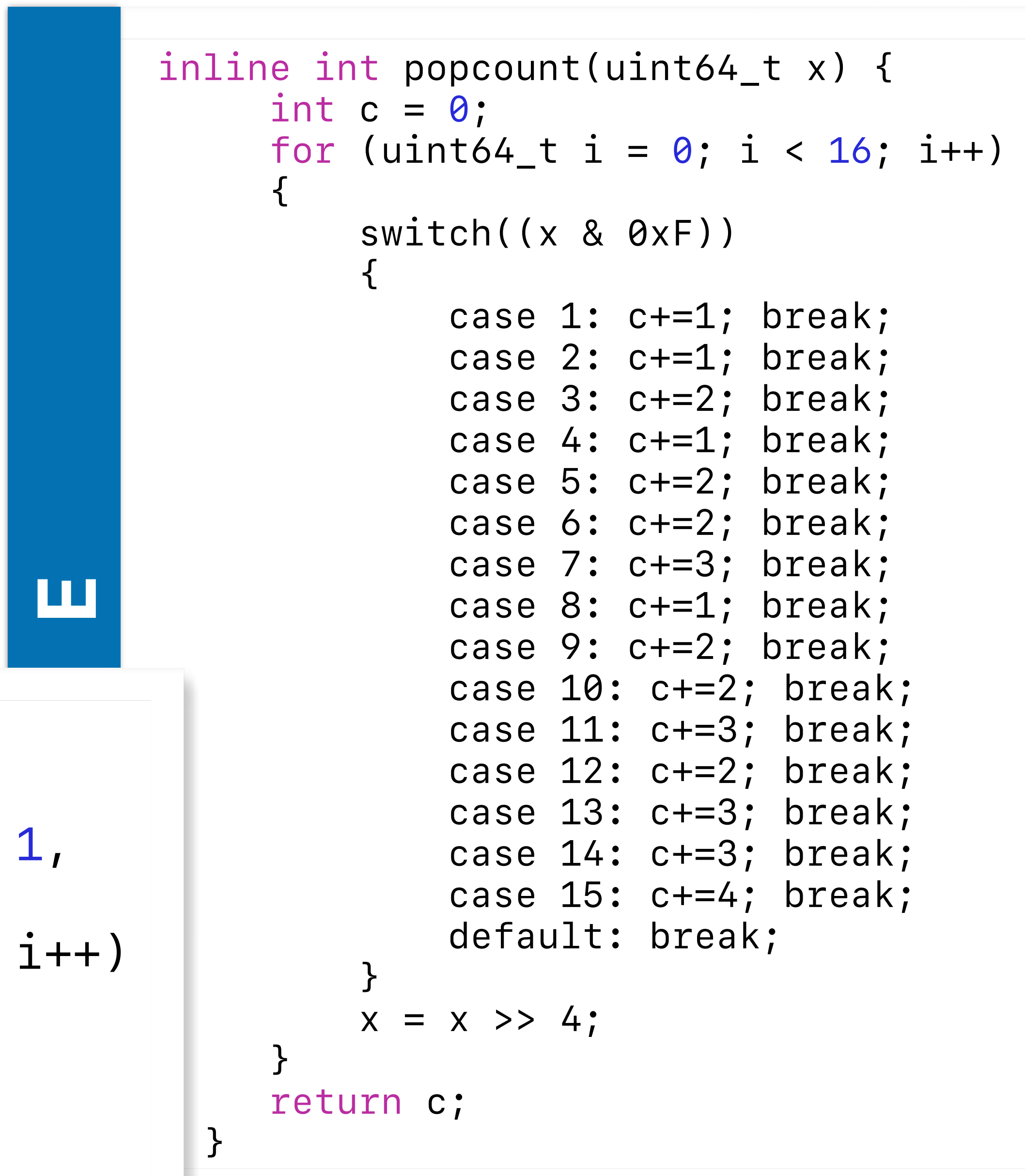
C

```
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 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

```
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;
}
```

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

5*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
```

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

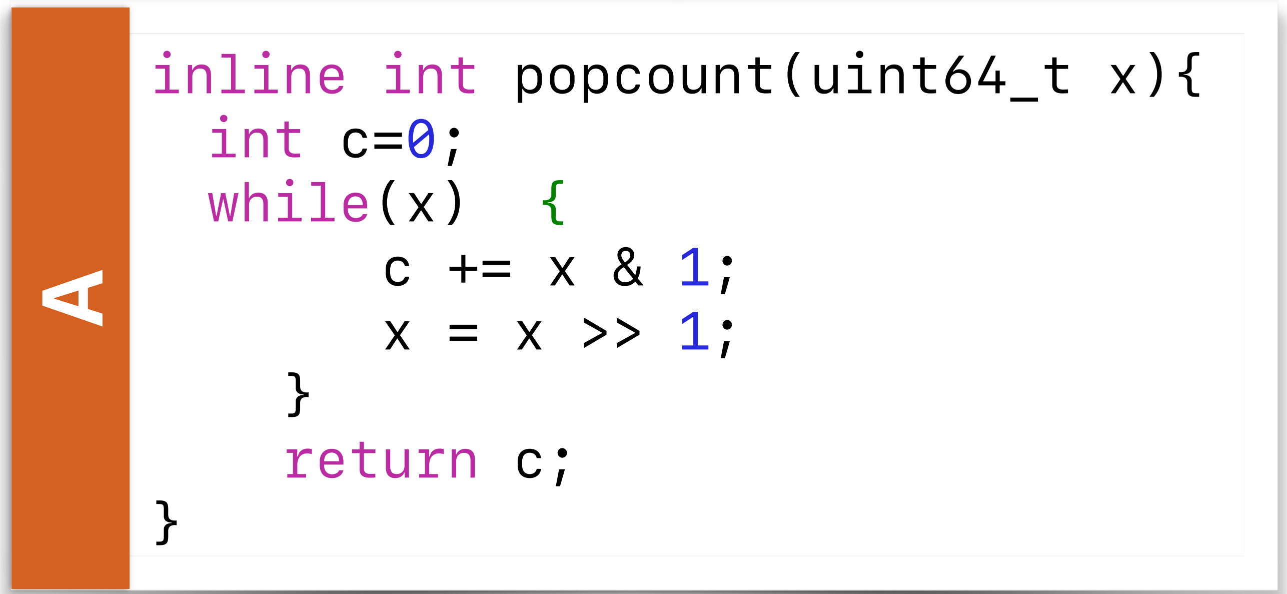
compiler reorder instructions and rename registers to mitigate hazards and exploit ILP

Only one branch for four iterations in A

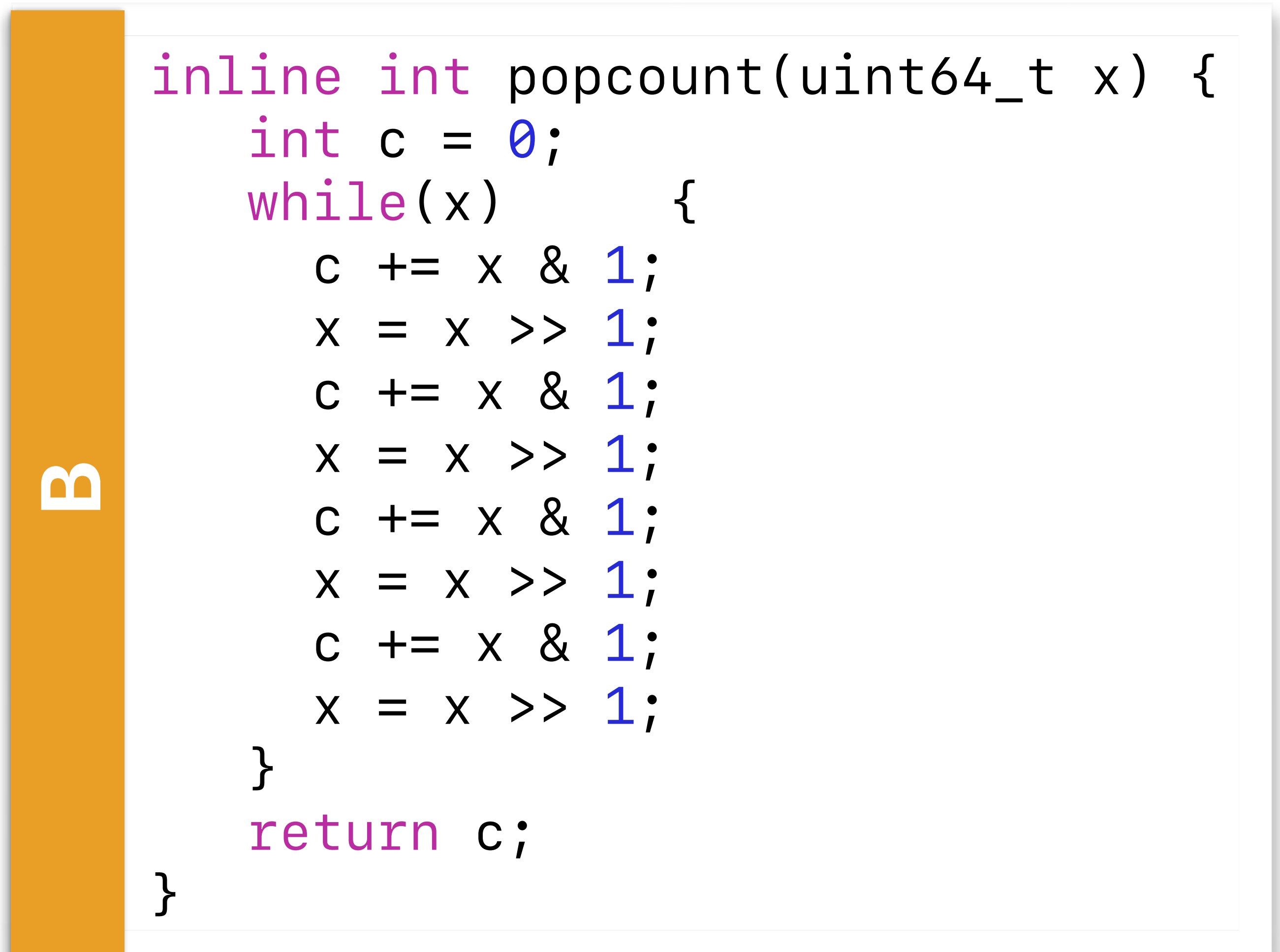
Why is B better than A?

- 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;
}
```

Announcements

- Assignment #3 due **Friday**
- Reading Quiz due next Monday

Computer
Science &
Engineering

203

つづく