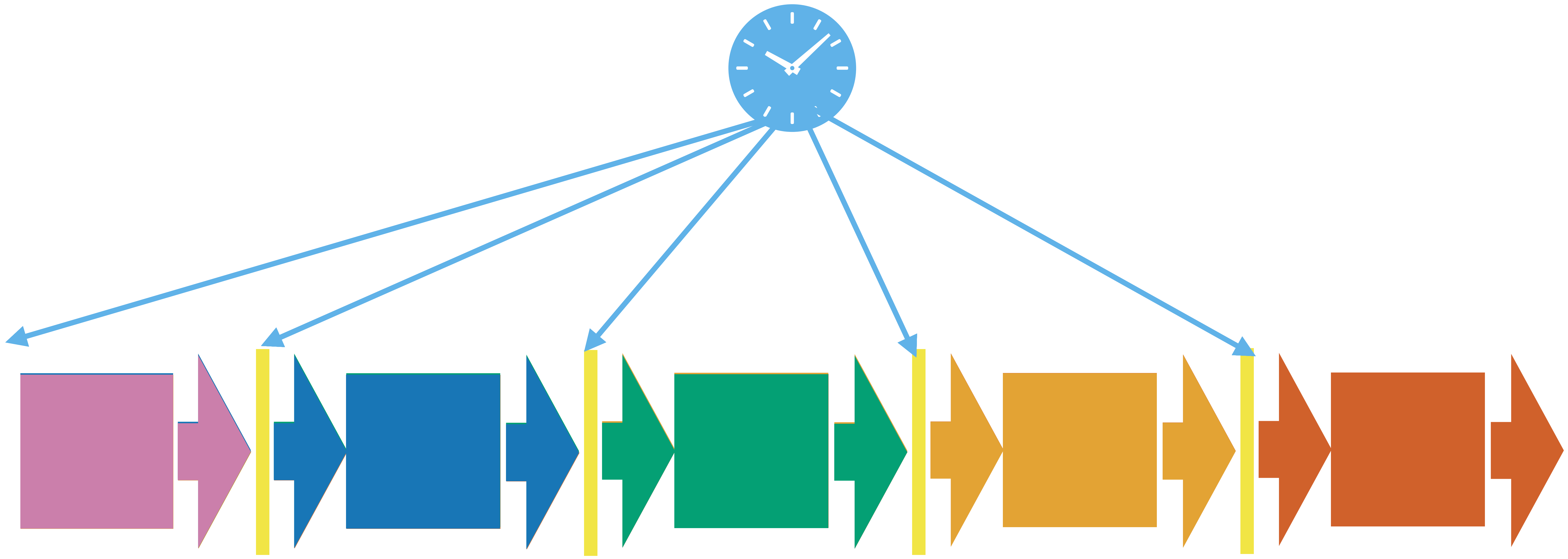
Data Hazards & Dynamic Instruction Scheduling: CPUtopia

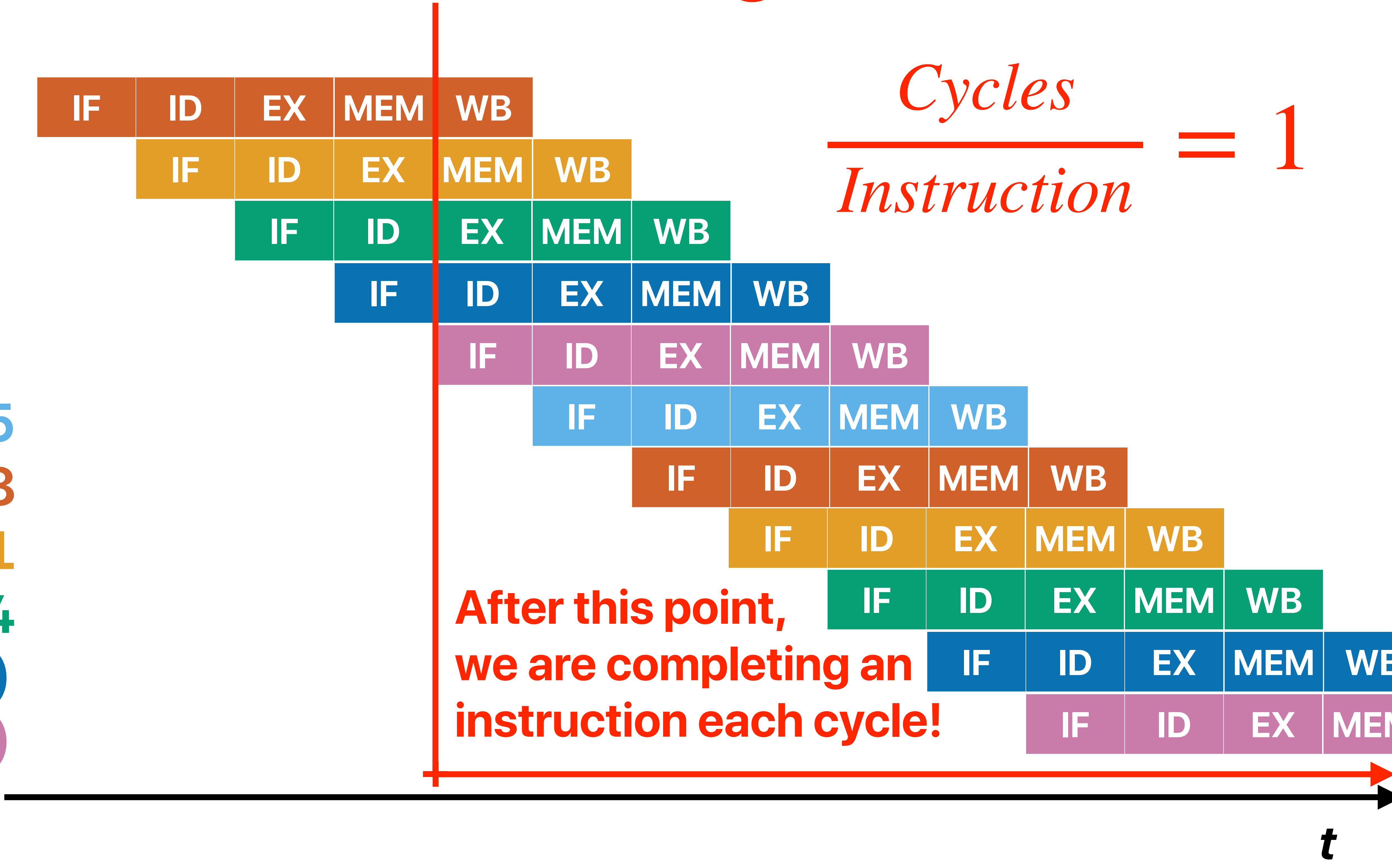
Hung-Wei Tseng

Recap: Pipelining



Recap: Pipelining

```
add x1, x2, x3
ld  x4, 0(x5)
sub x6, x7, x8
sub x9,x10,x11
sd  x1, 0(x12)
xor x13,x14,x15
and x16,x17,x18
add x19,x20,x21
sub x22,x23,x24
ld  x25, 4(x26)
sd  x27, 0(x28)
```



After this point, we are completing an instruction each cycle!

$$\frac{Cycles}{Instruction} = 1$$

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

- An instruction currently in the pipeline cannot receive the "logically" correct value for execution
- Data dependencies
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

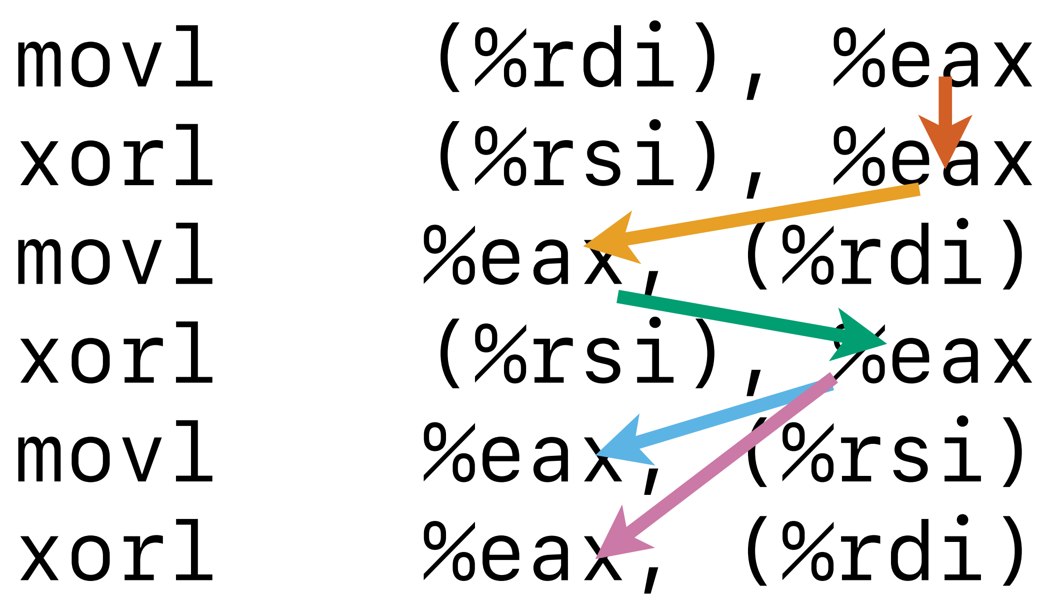
```
movl     (%rdi), %eax
movl     (%rsi), %edx
movl     %edx, (%rdi)
movl     %eax, (%rsi)
```

```
int temp = *a;
*a = *b;
*b = temp;
```

A. 1
[✓ B. 2]
C. 3
D. 4
E. 5

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

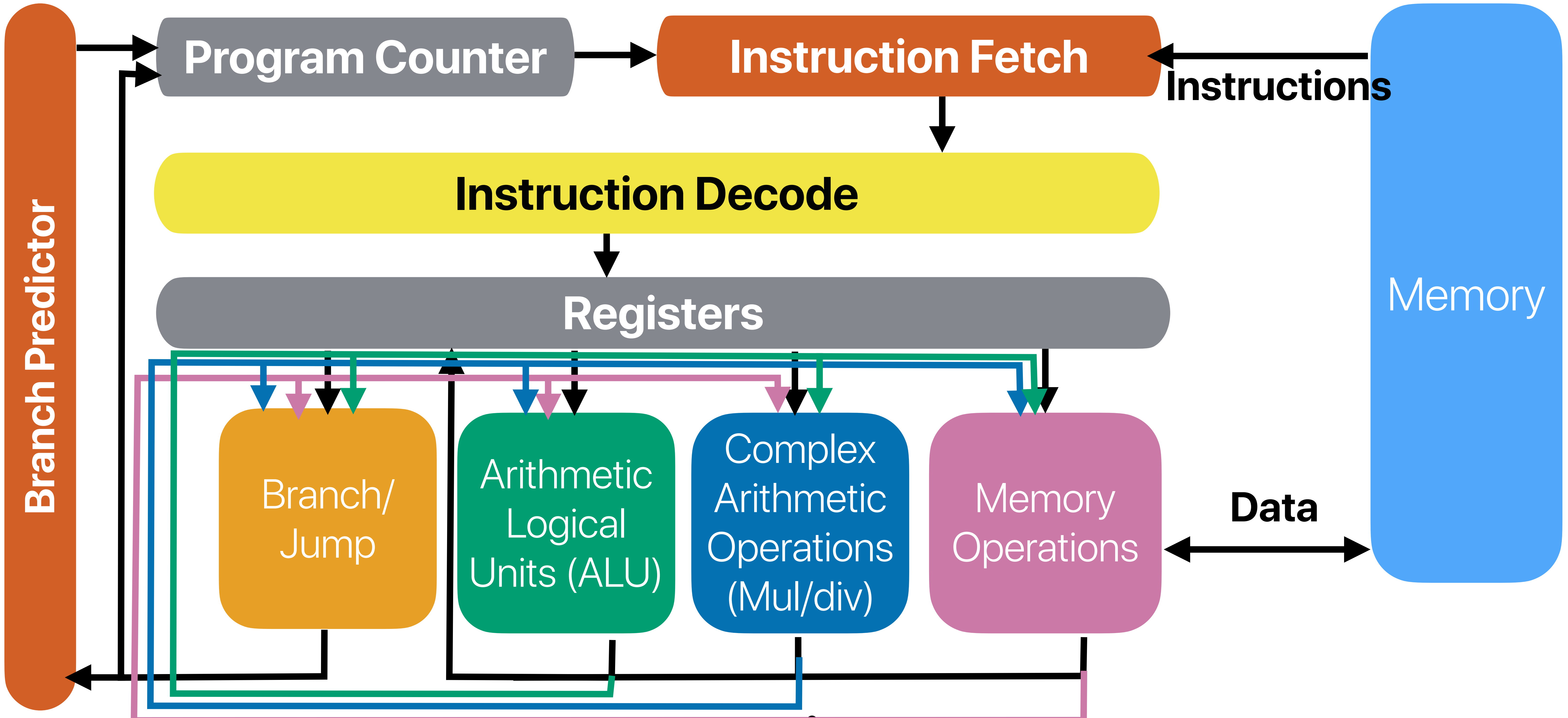


```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 1
B. 2
C. 3
D. 4
[✓ E. 5]

Data "forwarding"



3:00

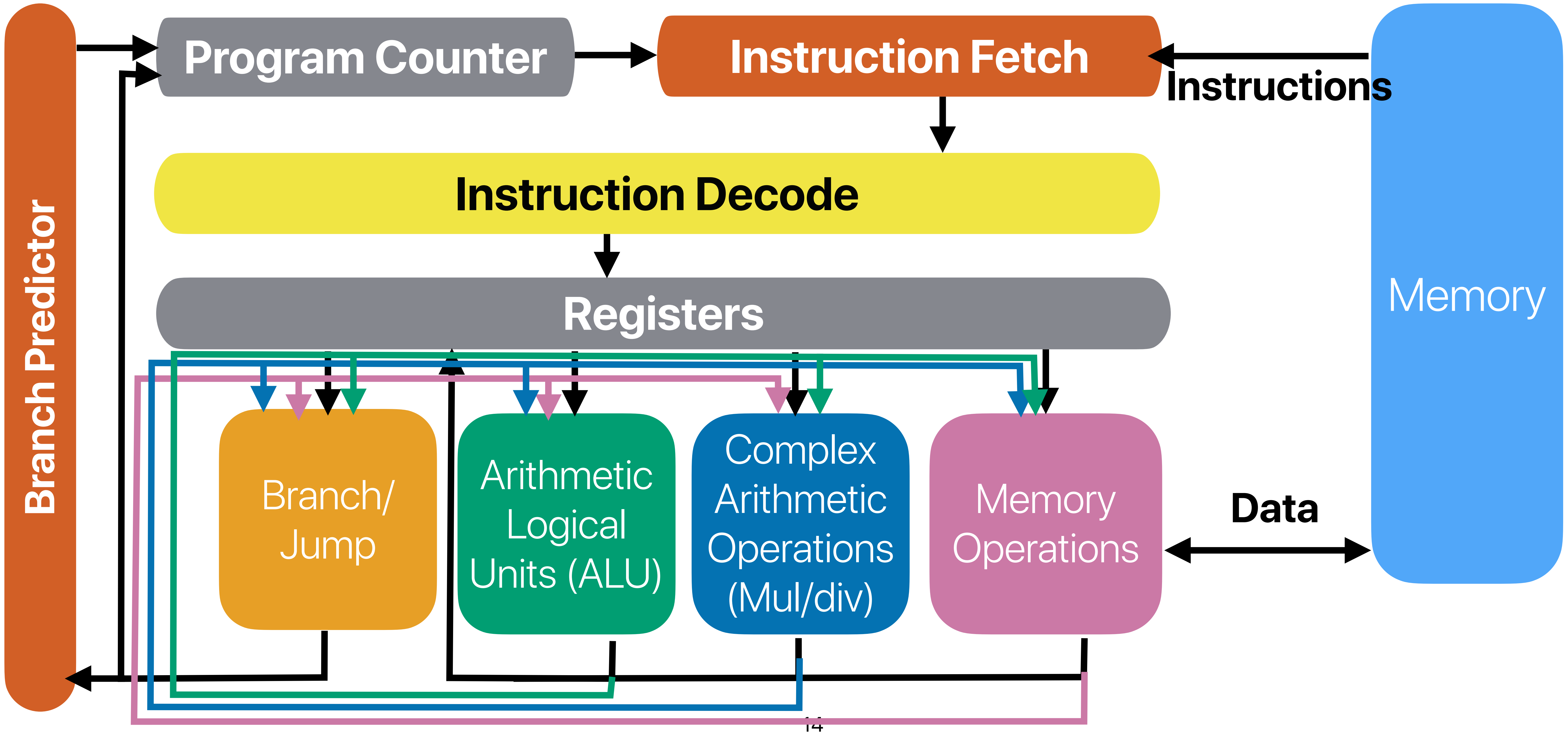
If you're the governor, what would you do to make DMV more efficient?

Ideas?

Outline

- Data hazards
  - Data forwarding
- SuperScalar
- Out-of-order, Dynamic instruction scheduling

Data “forwarding”



https://www.pollev.com/hungweitseng close in 1:00

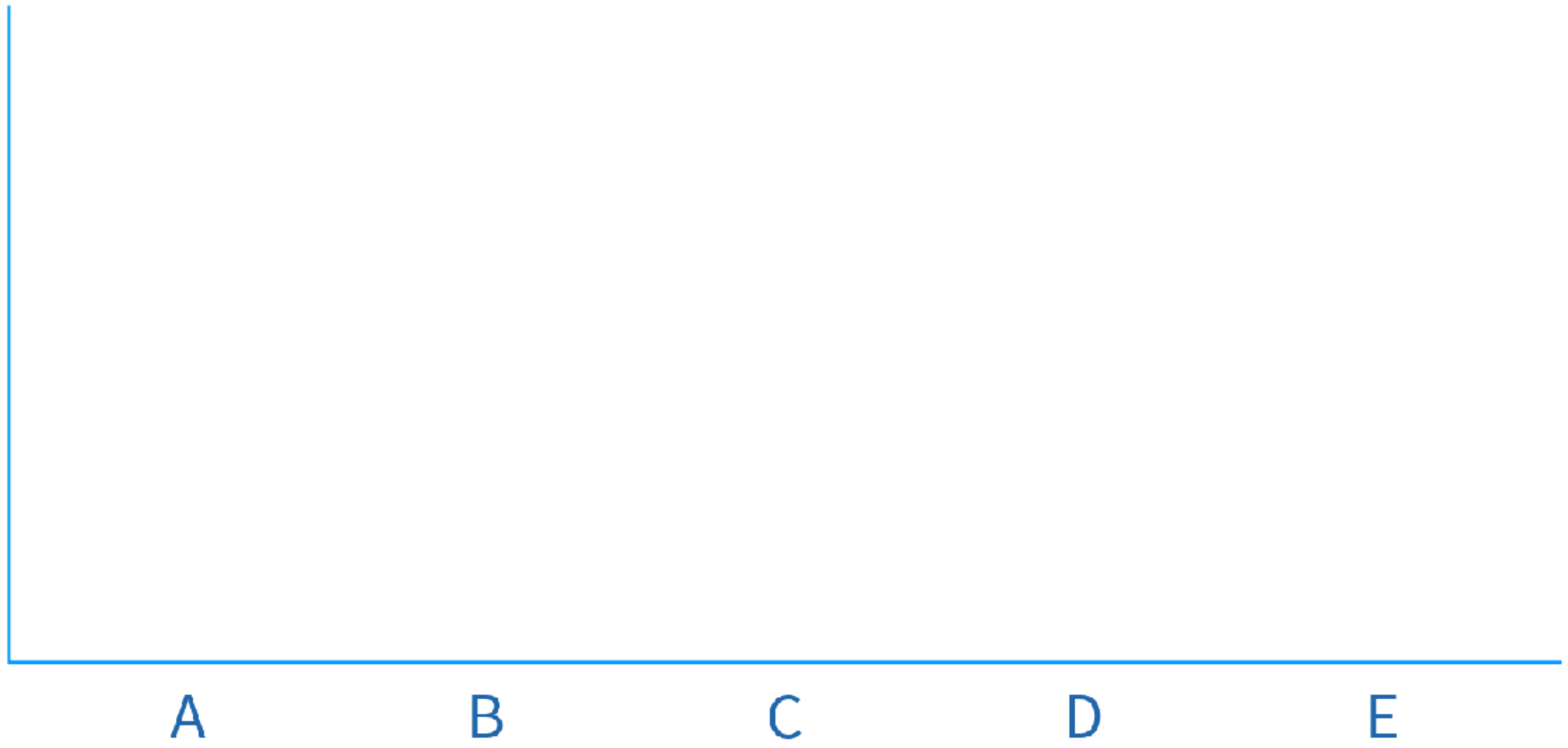
How many of them are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least)?

```
movl     (%rdi), %eax
movl     (%rsi), %edx
movl     %edx, (%rdi)
movl     %eax, (%rsi)
```

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

Stalls



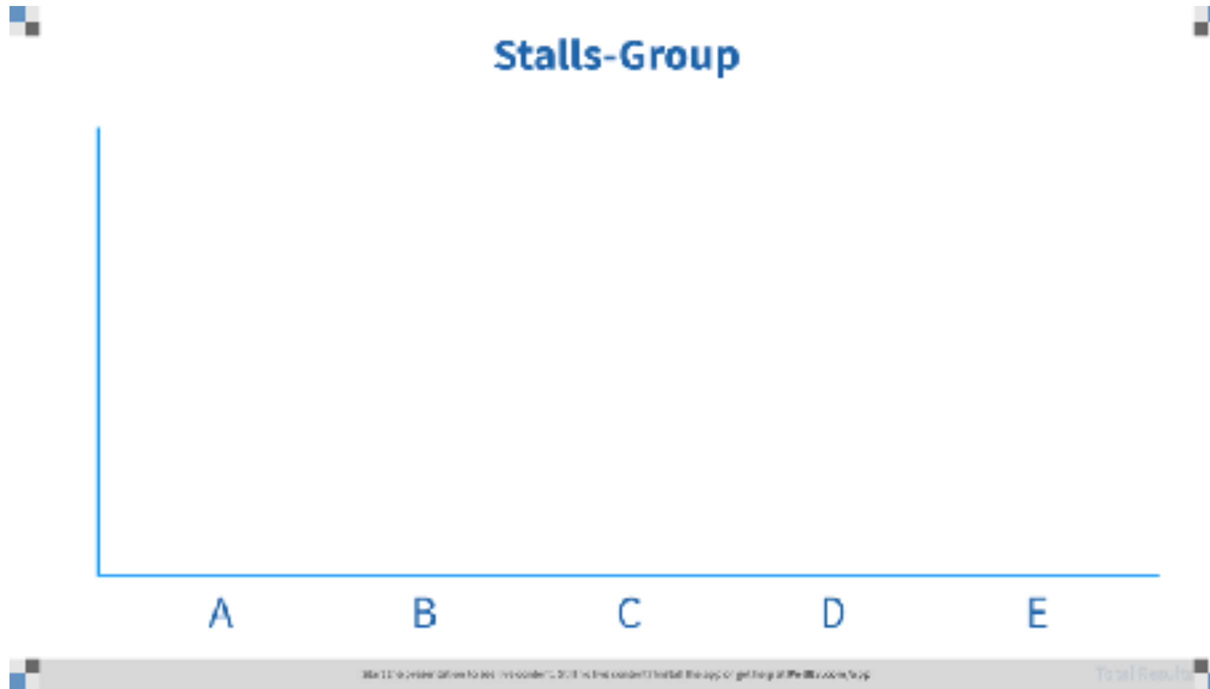
https://www.pollev.com/hungweitseng close in 1:00

How many of data hazards?

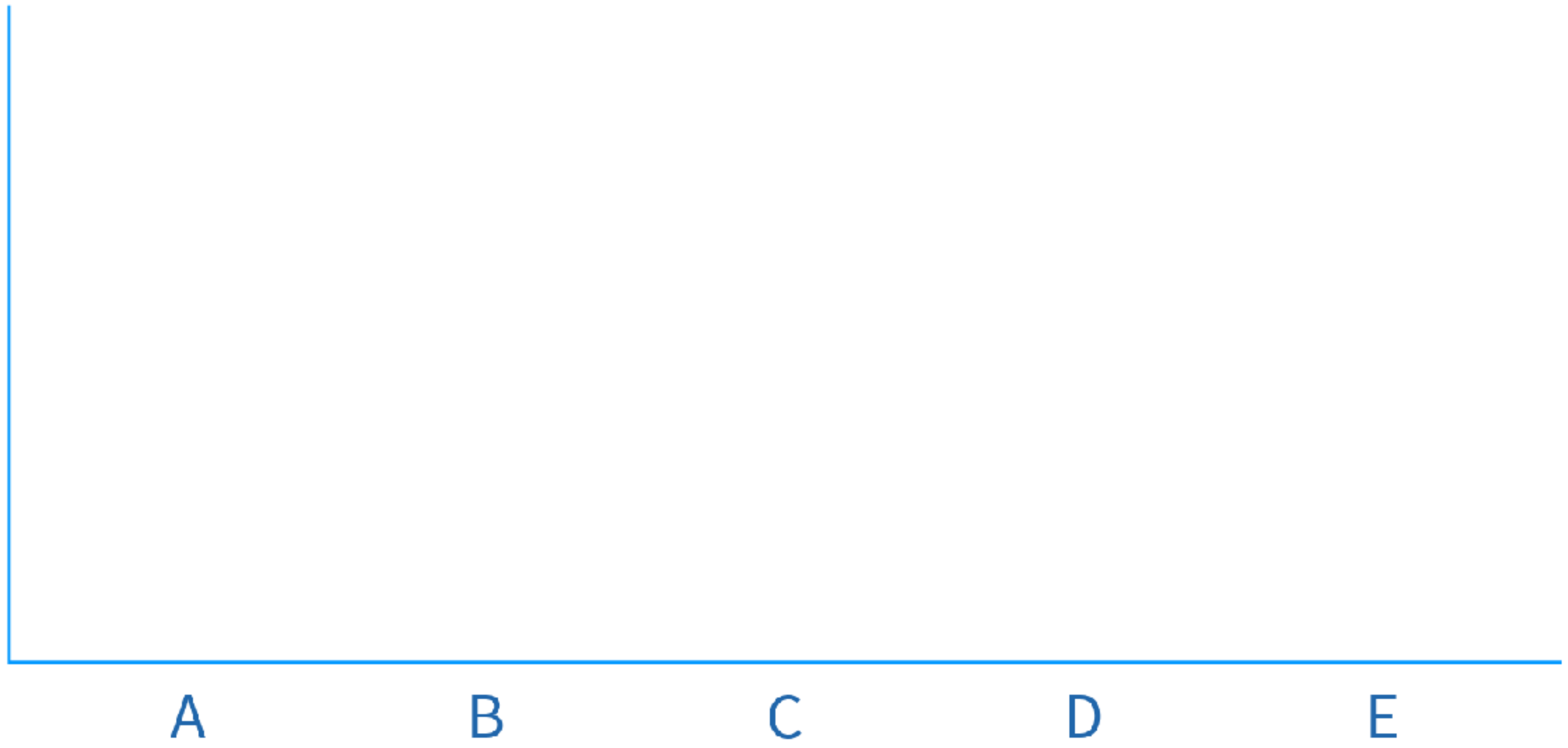
- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least)?

```
movl     (%rdi), %eax
movl     (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

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



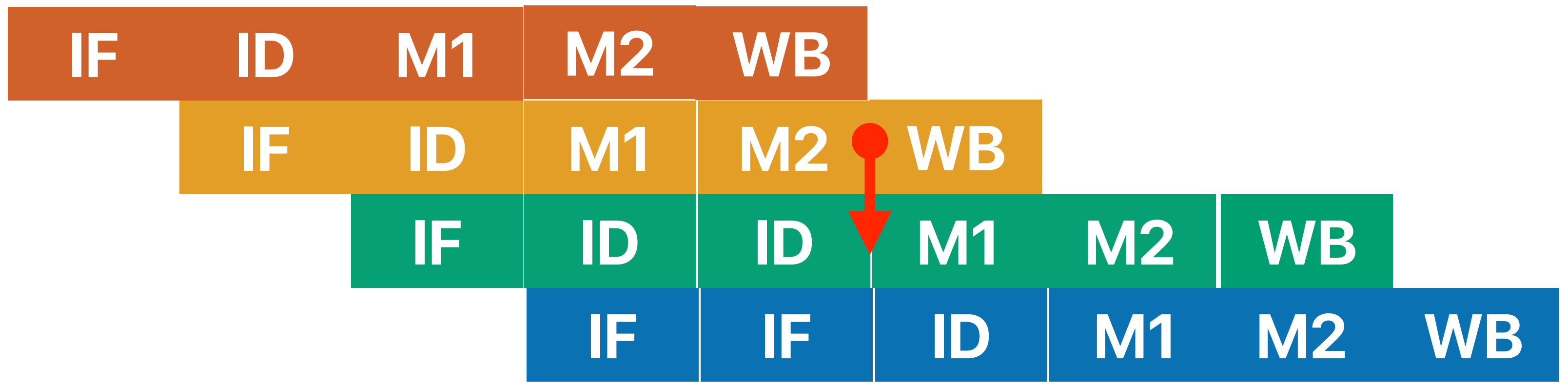
Stalls-Group



How many of data hazards?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least)?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



A. 0
[✓ B. 1]
C. 2
D. 3
E. 4

```
int temp = *a;
*a = *b;
*b = temp;
```

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

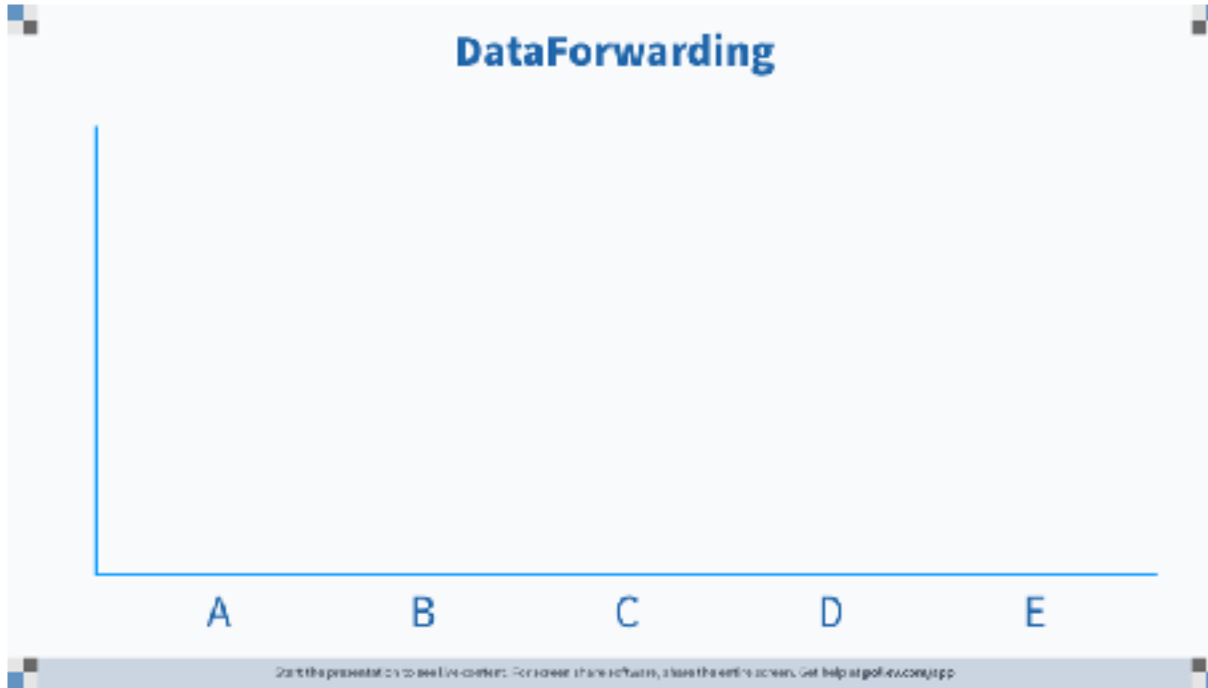
How many of them are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least) and **xorl takes 3 cycles**?

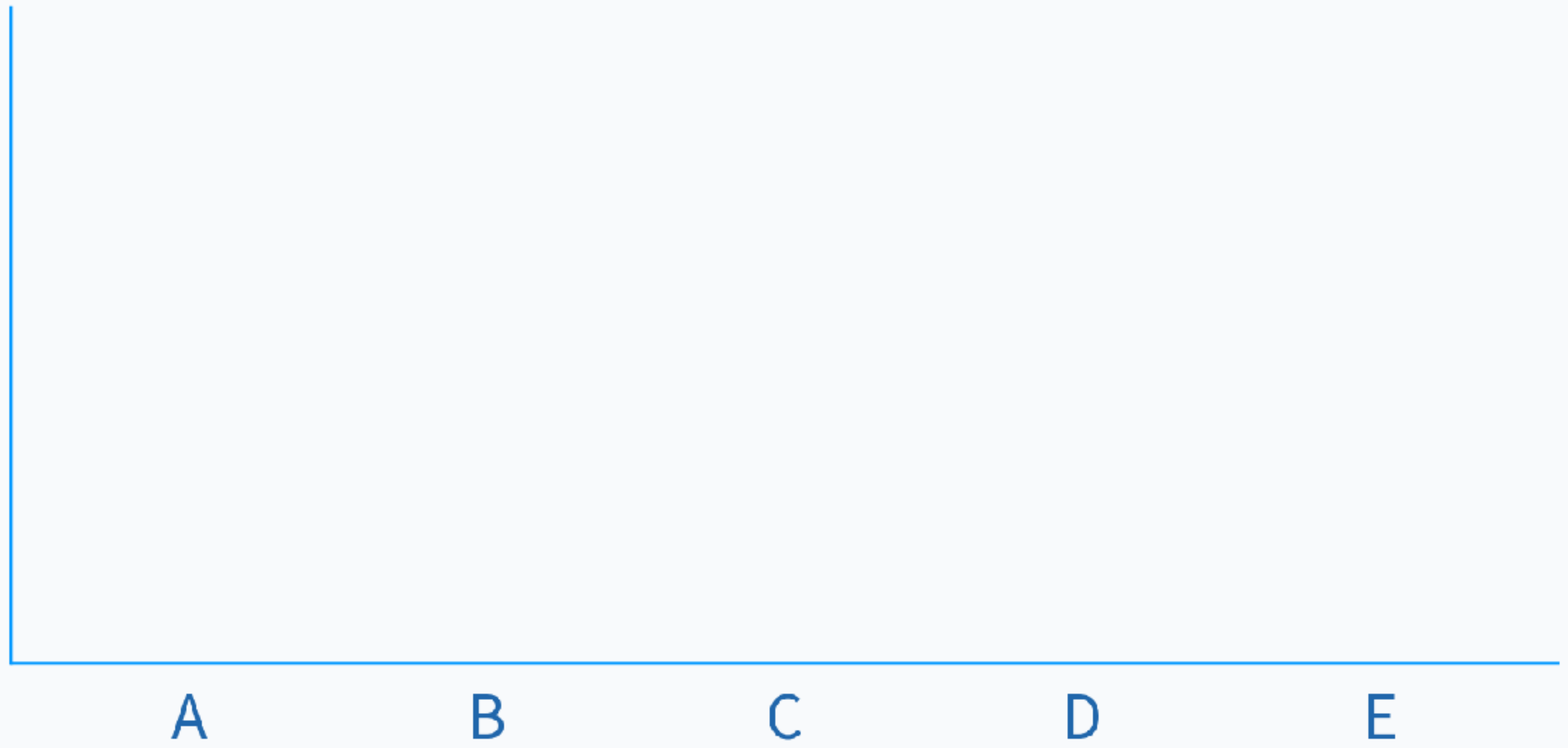
```
movl     (%rdi), %eax
xorl     (%rsi), %eax
movl     %eax, (%rdi)
xorl     (%rsi), %eax
movl     %eax, (%rsi)
xorl     %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

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



DataForwarding



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

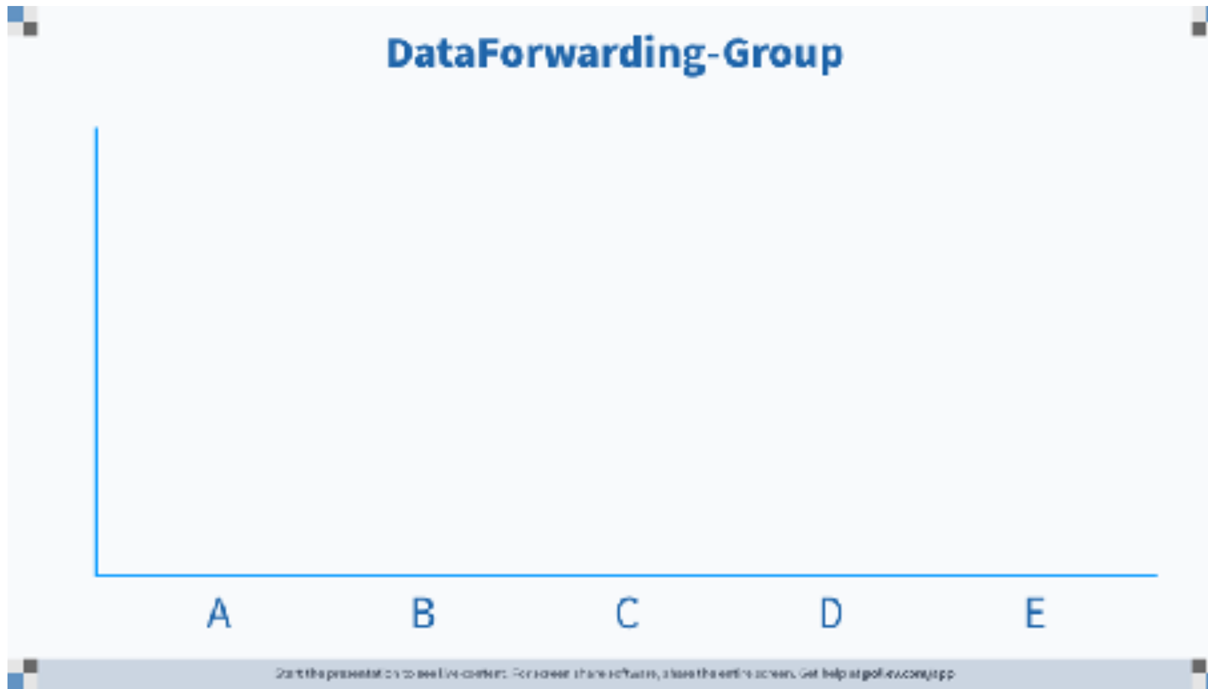
How many of data hazards w/ Data Forwarding?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least) and **xorl takes 3 cycles**?

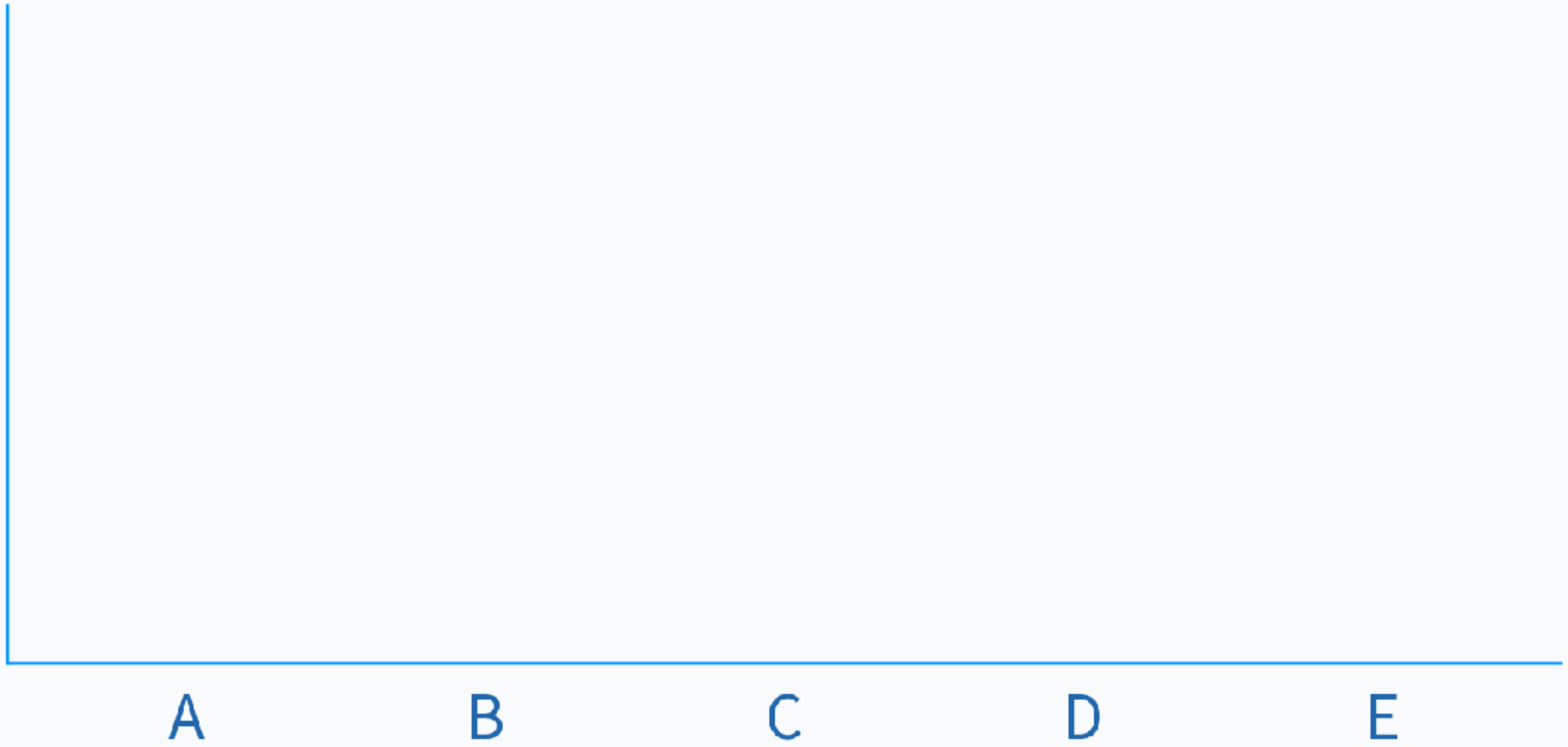
```
movl     (%rdi), %eax
xorl     (%rsi), %eax
movl     %eax, (%rdi)
xorl     (%rsi), %eax
movl     %eax, (%rsi)
xorl     %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

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



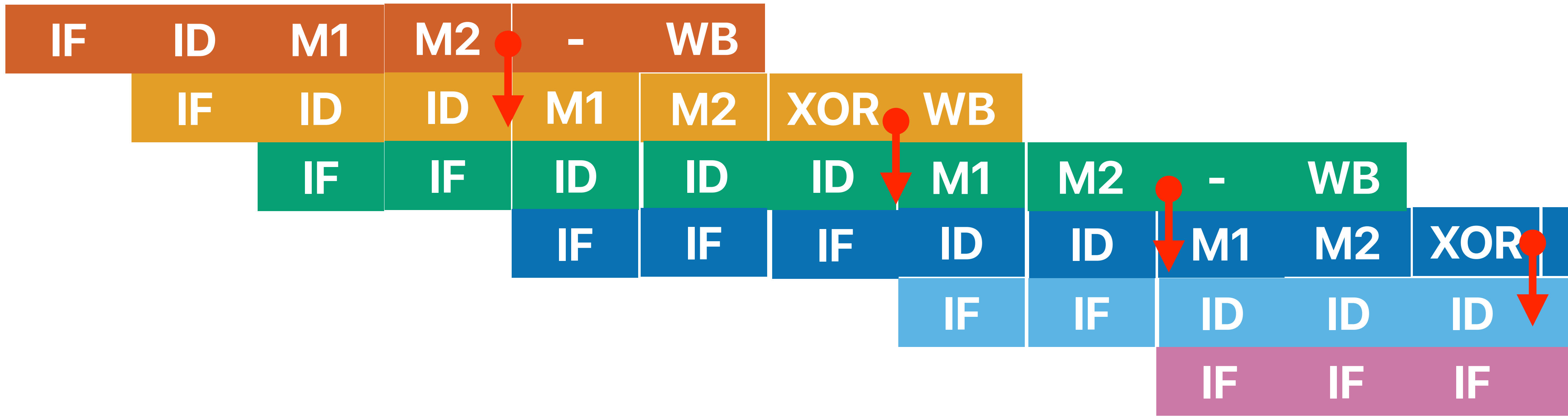
DataForwarding-Group



How many of data hazards w/ Data Forwarding?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 2 cycles (already very optimistic, most L1 cache takes 4 cycles at least) and **xorl takes 3 cycles**?

```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```



A. 1
B. 2
C. 3
[✓ D. 4]
E. 5

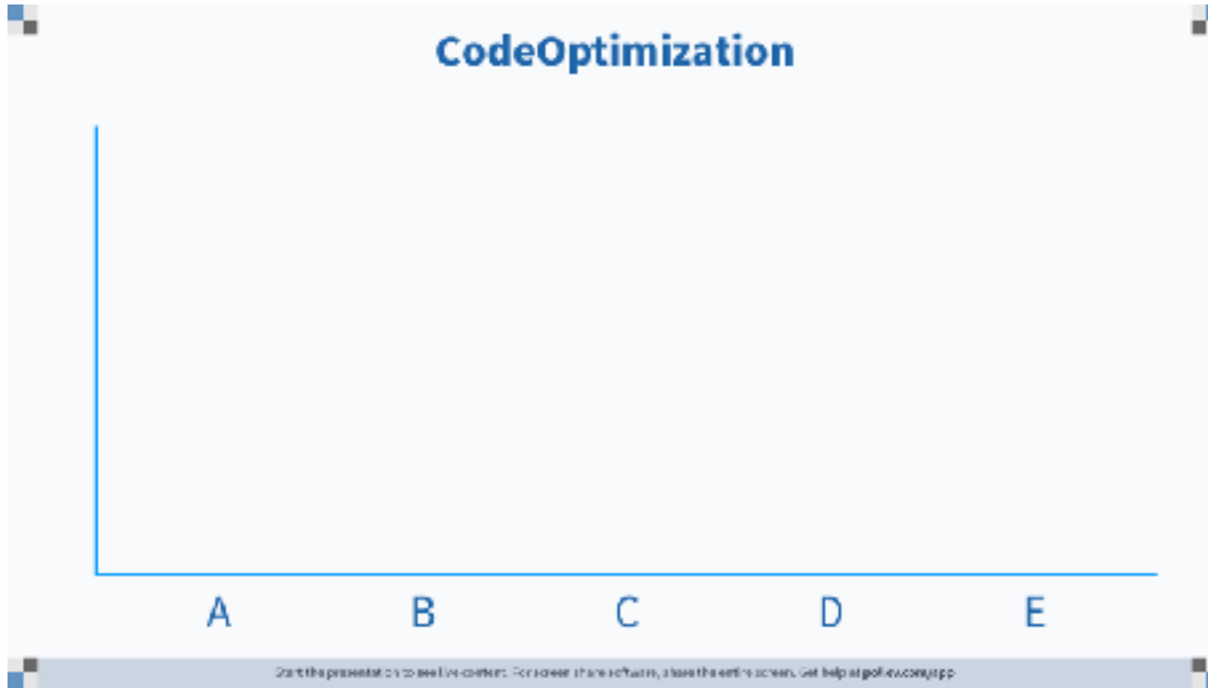
https://www.pollev.com/hungweitseng close in 1:30

The effect of code optimization

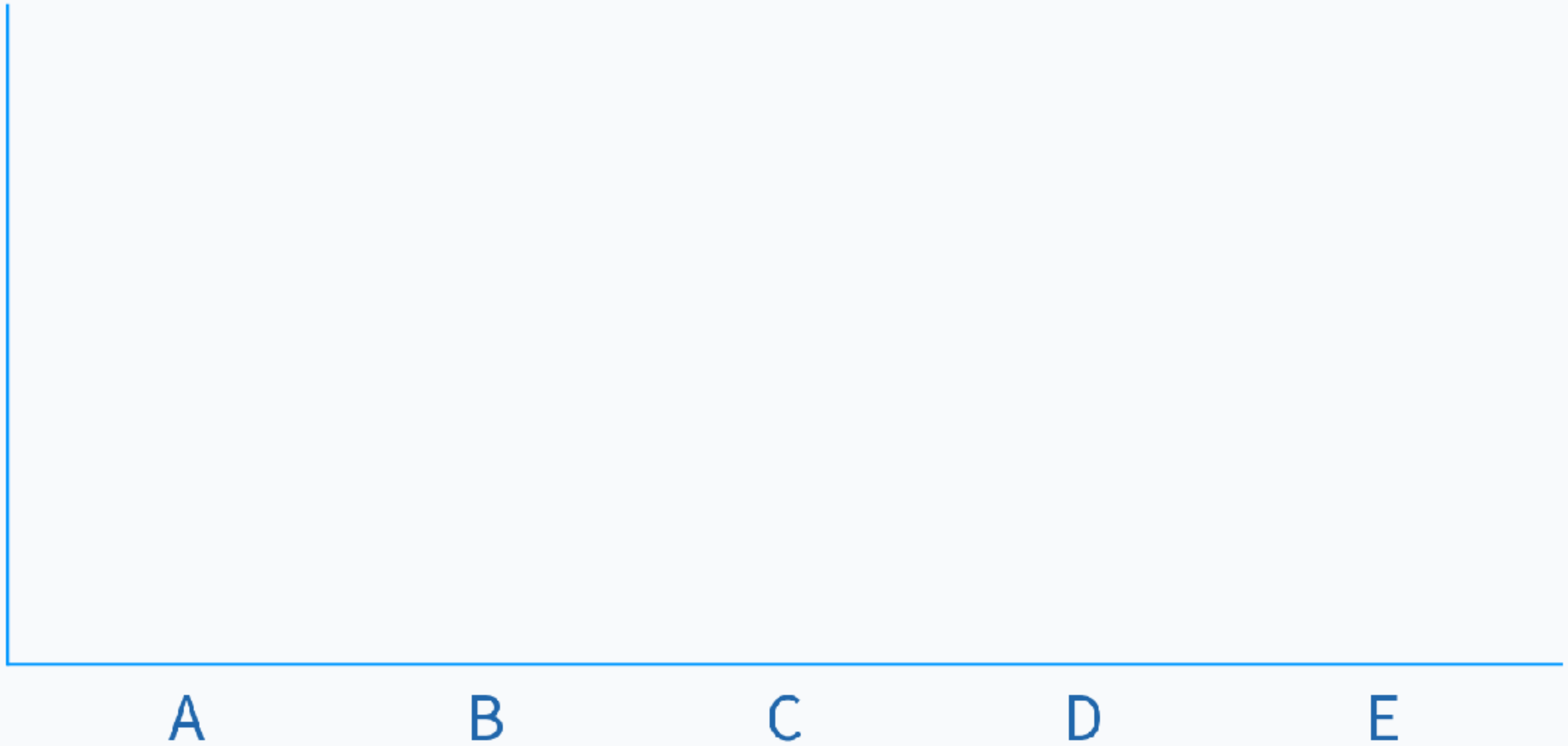
- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

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

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered



CodeOptimization



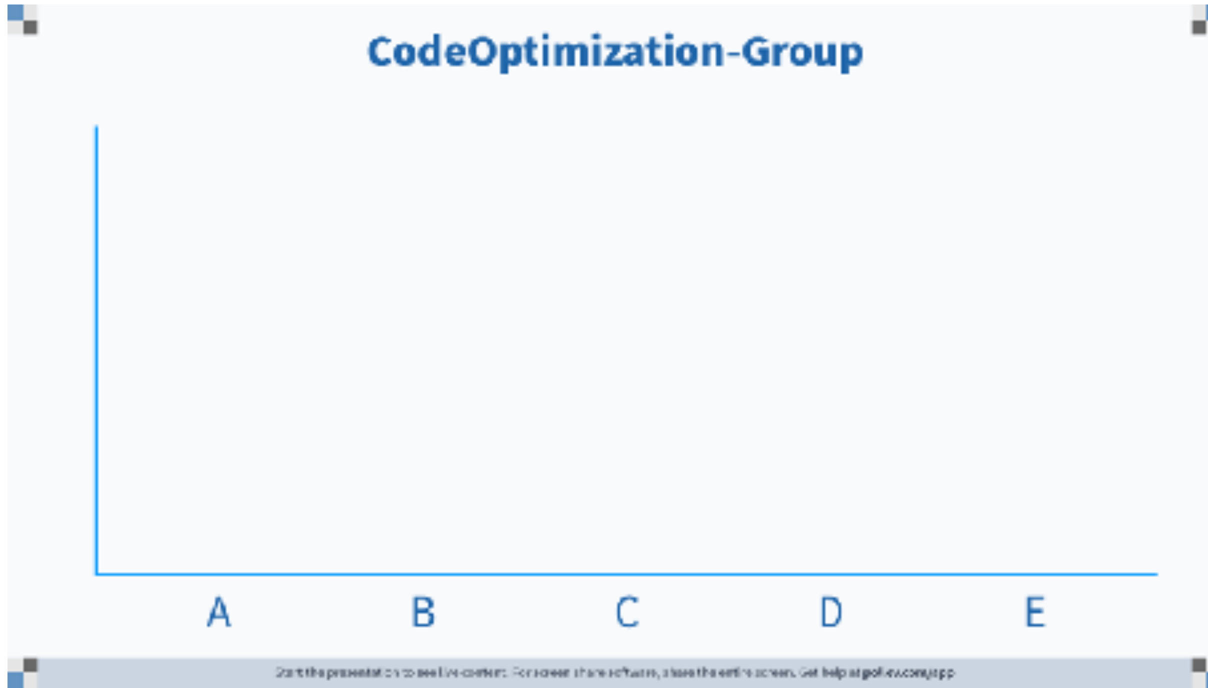
https://www.pollev.com/hungweitseng close in 1:30

The effect of code optimization

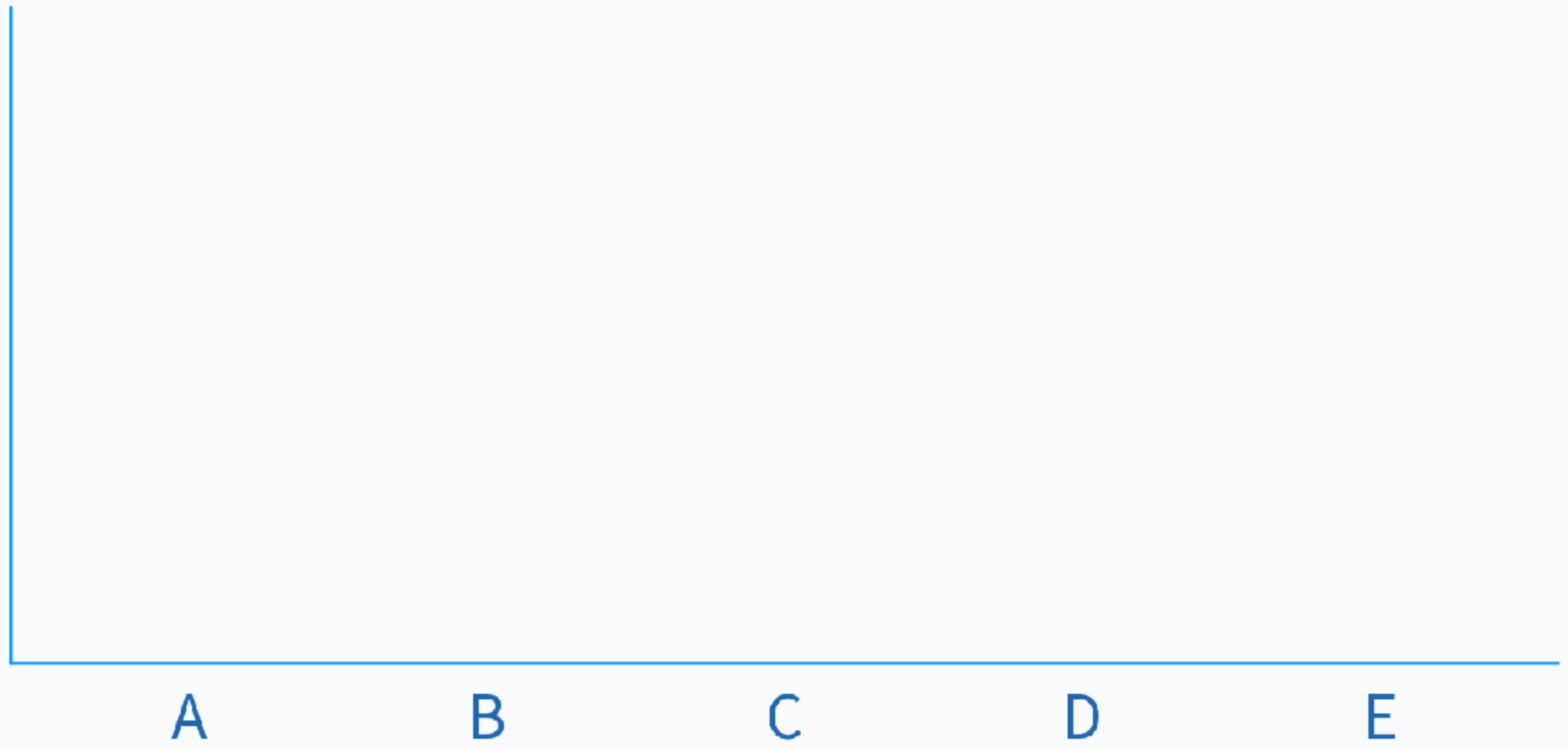
- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

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

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered



CodeOptimization-Group



The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

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

A. (1) & (2)

[✓ B. (2) & (3)]

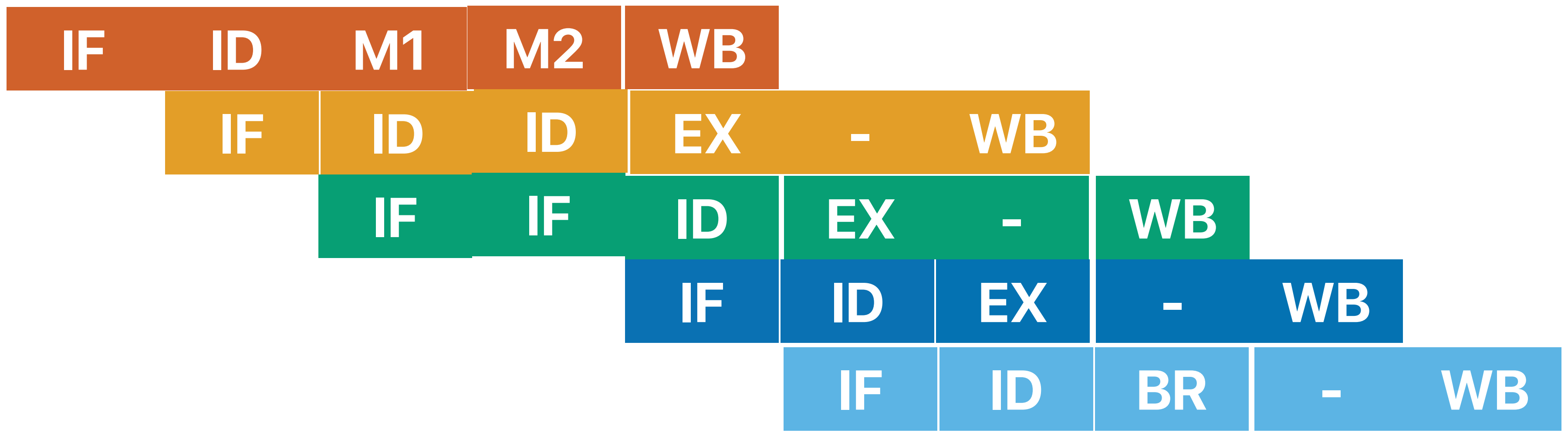
C. (3) & (4)

D. (4) & (5)

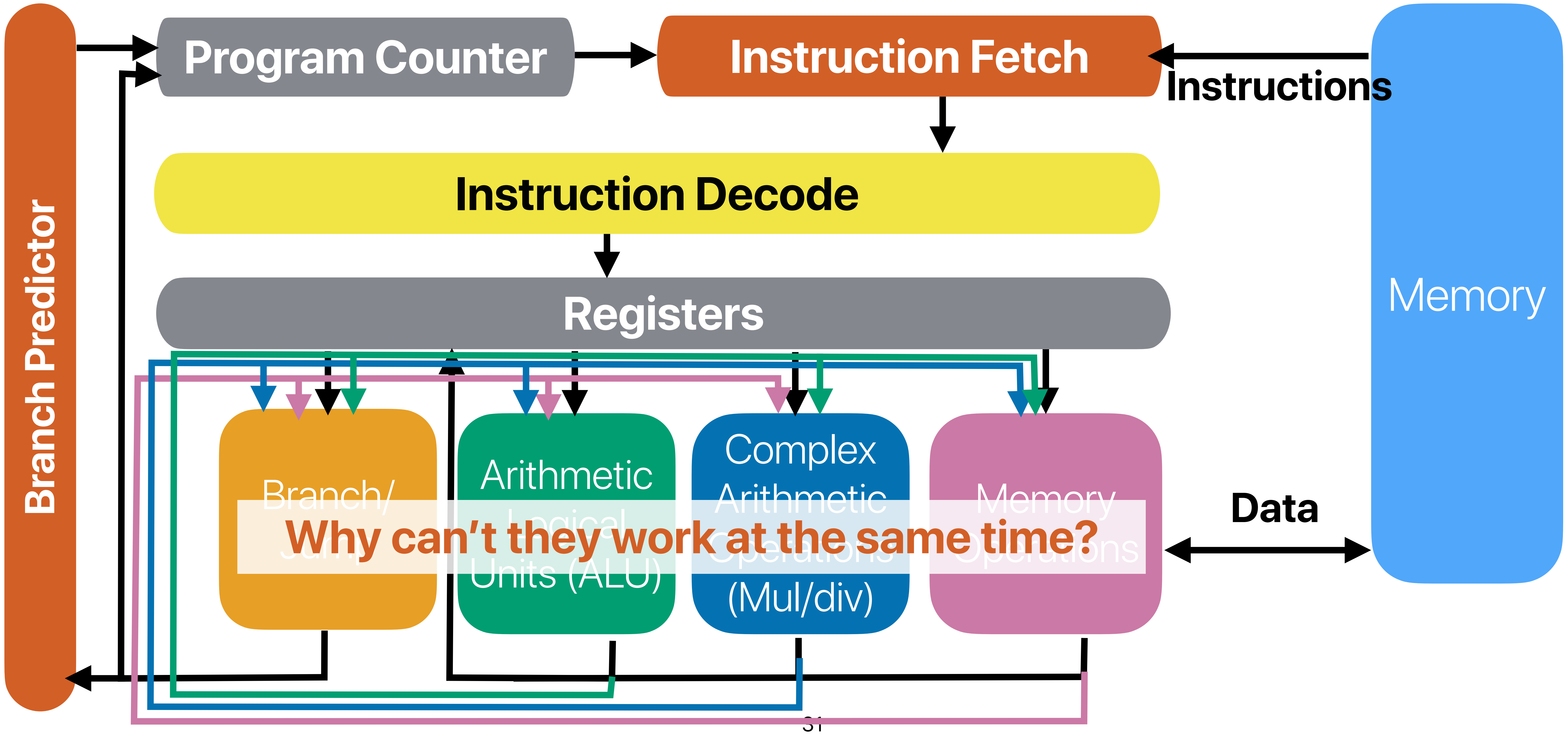
E. None of the pairs can be reordered

Single pipeline

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addl    %ecx, %eax
        addq    $4, %rdi
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```



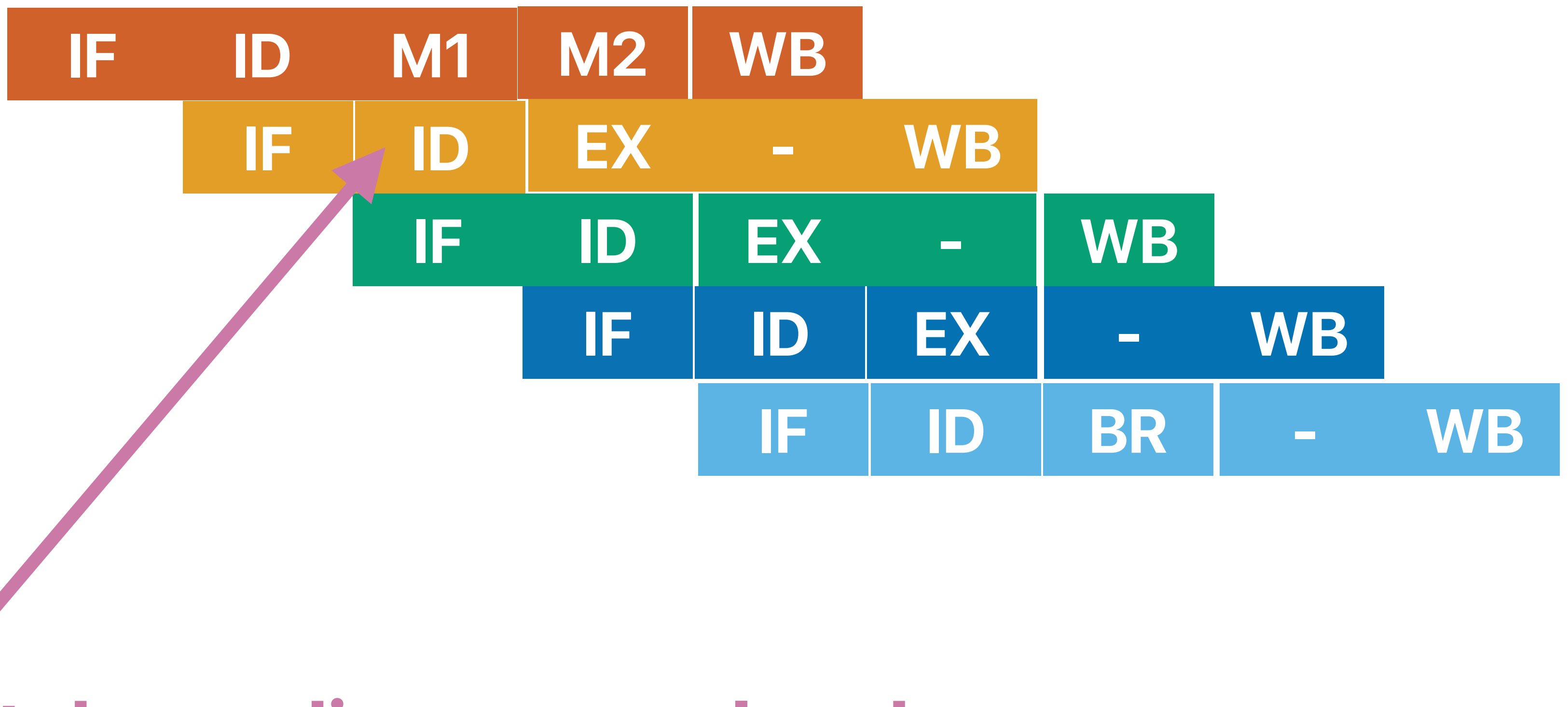
Data "forwarding"



Why can't they work at the same time?

Compiler optimization

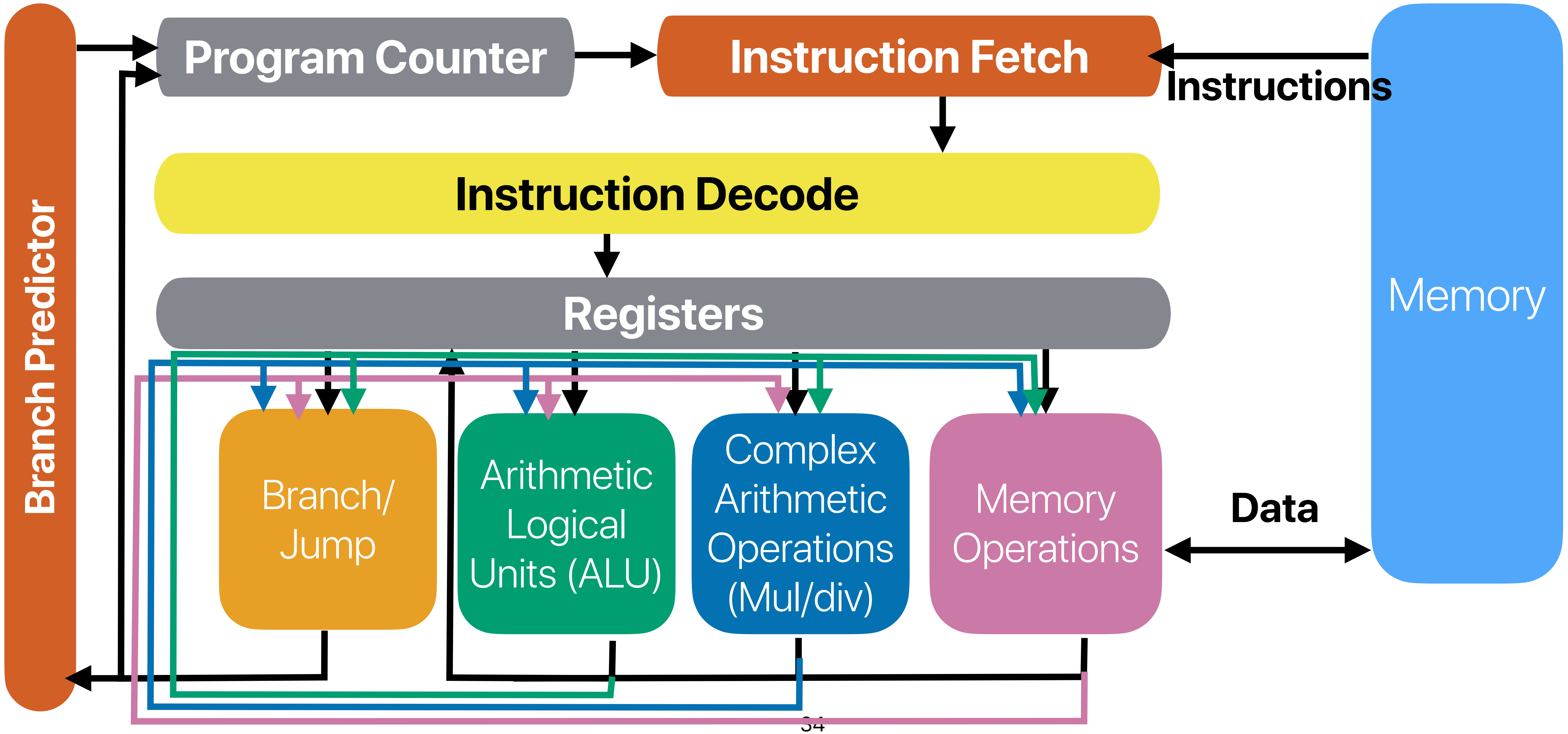
```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addq    $4, %rdi
        addl    %ecx, %eax
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```



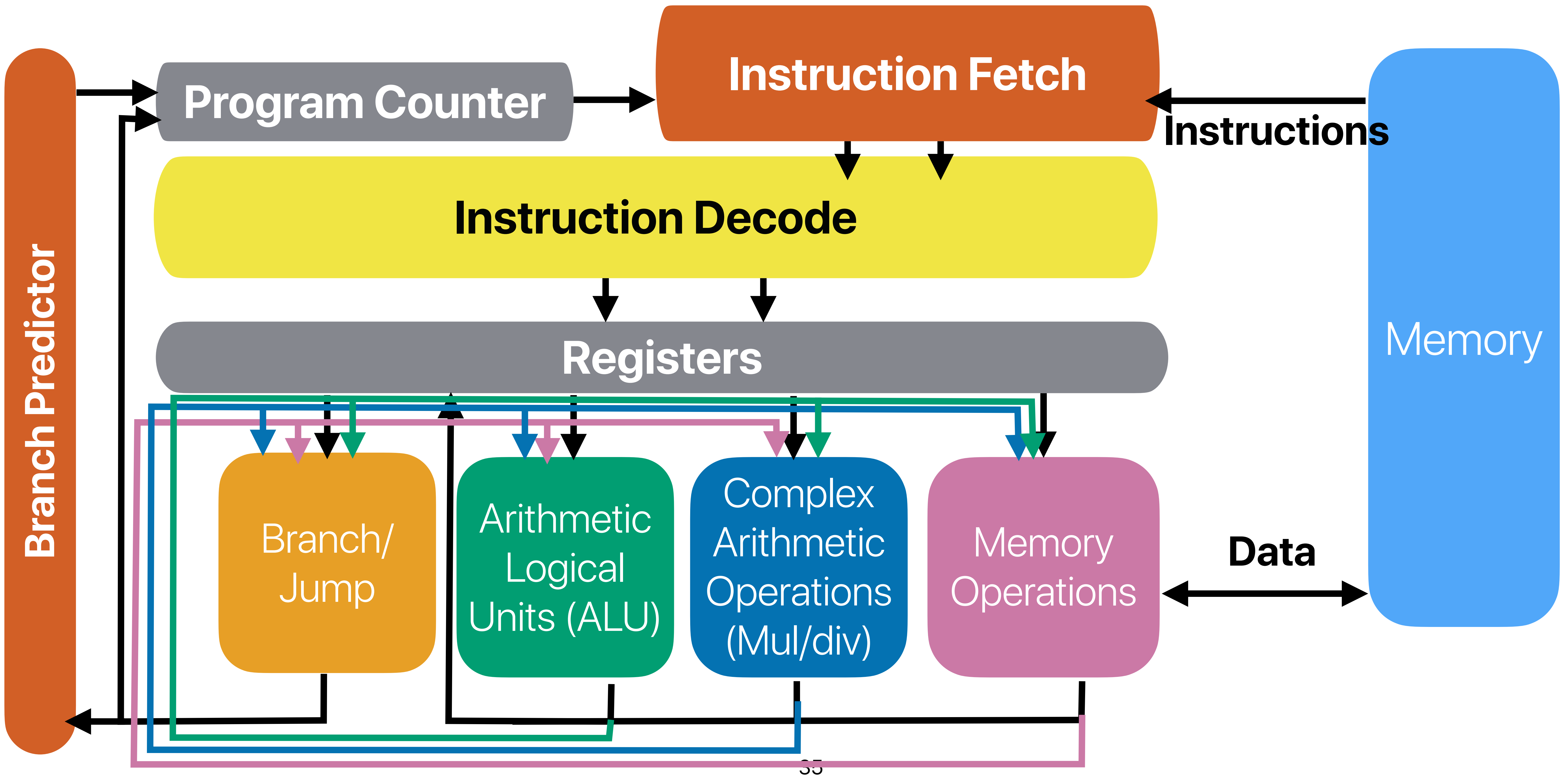
addq is not depending on movl and ALU is free! can we execute them together?

If CPI==1 the limitation?

Data "forwarding"



Super Scalar



Super Scalar

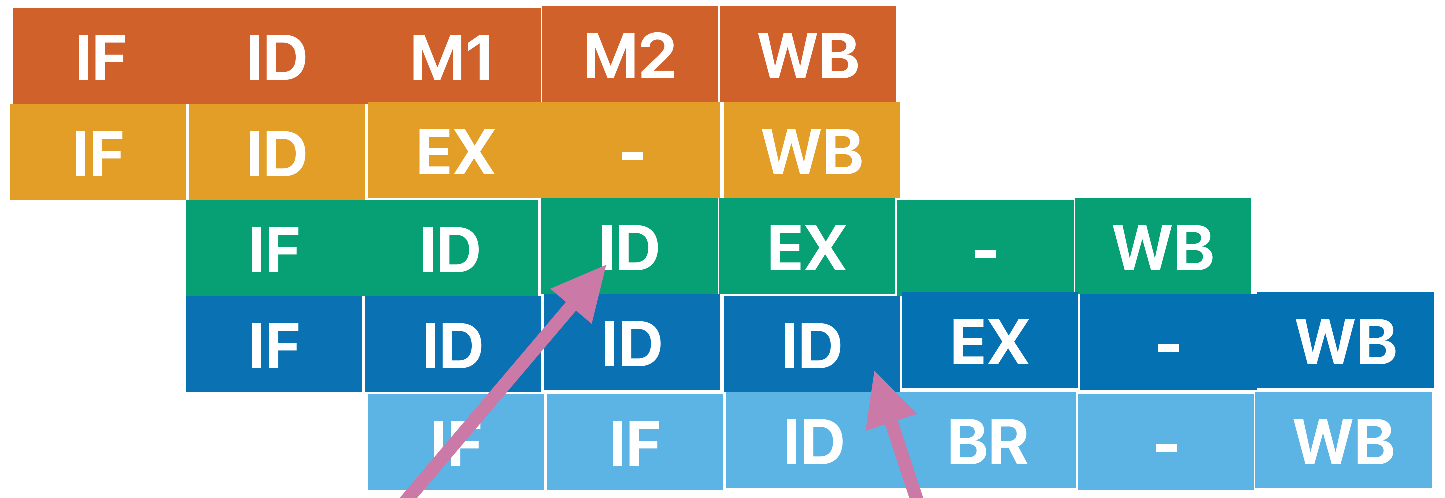
Superscalar

- 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

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

Superscalar: fetch/issue width == 2, theoretical CPI = 0.5

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addq    $4, %rdi
        addl    %ecx, %eax
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```

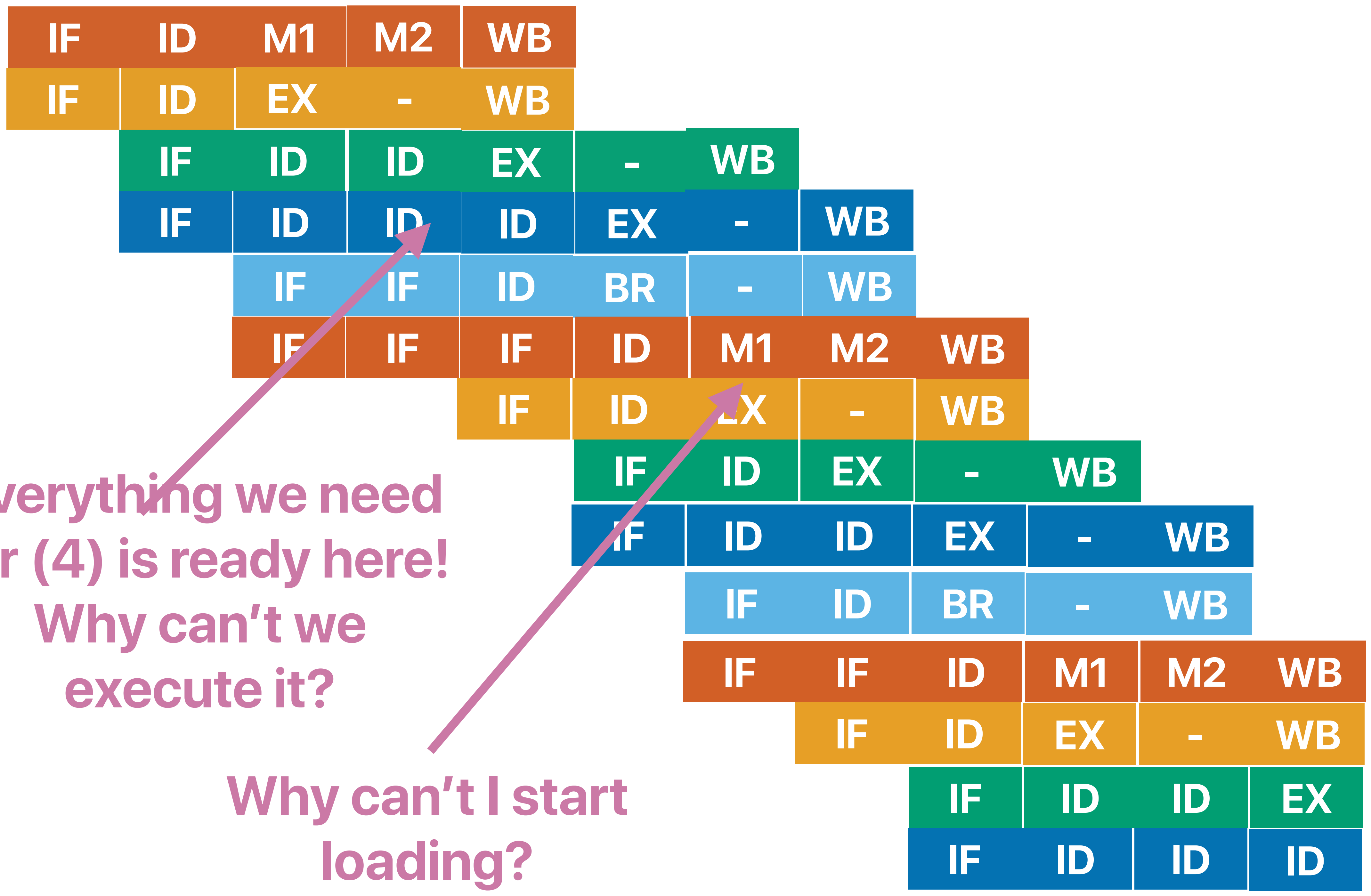


Stall because %ecx is not ready

Stall because we have only one ALU (structural hazard)

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
```



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

Why can't I start loading?

Limitations of Compiler Optimizations

- If the hardware (e.g., pipeline changes), the same compiler optimization may not be that helpful
- The compiler can only optimize on static instructions, but cannot optimize dynamic instruction
  - Compiler cannot predict branches
  - Compiler does not know if cache has the data/instructions

What do you need to execution an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — **all data dependencies are resolved**
- Whenever the target functional unit is available

Dynamic instruction scheduling/ Out-of-order (OoO) execution

What do you need to execution an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — **all data dependencies are resolved**
- Whenever the target functional unit is available

Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.

```
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
```

$$\begin{tikzcd}
1 \arrow[dd] & 2 \arrow[d] \arrow[dr] \arrow[drr] & & \\
& 6 \arrow[dd] & 4 \arrow[d] & 7 \arrow[d] \\
3 \arrow[dr] & & 5 & 9 \arrow[d] \\
& 8 & & 10
\end{tikzcd}$$

- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered

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

If we can predict the future …

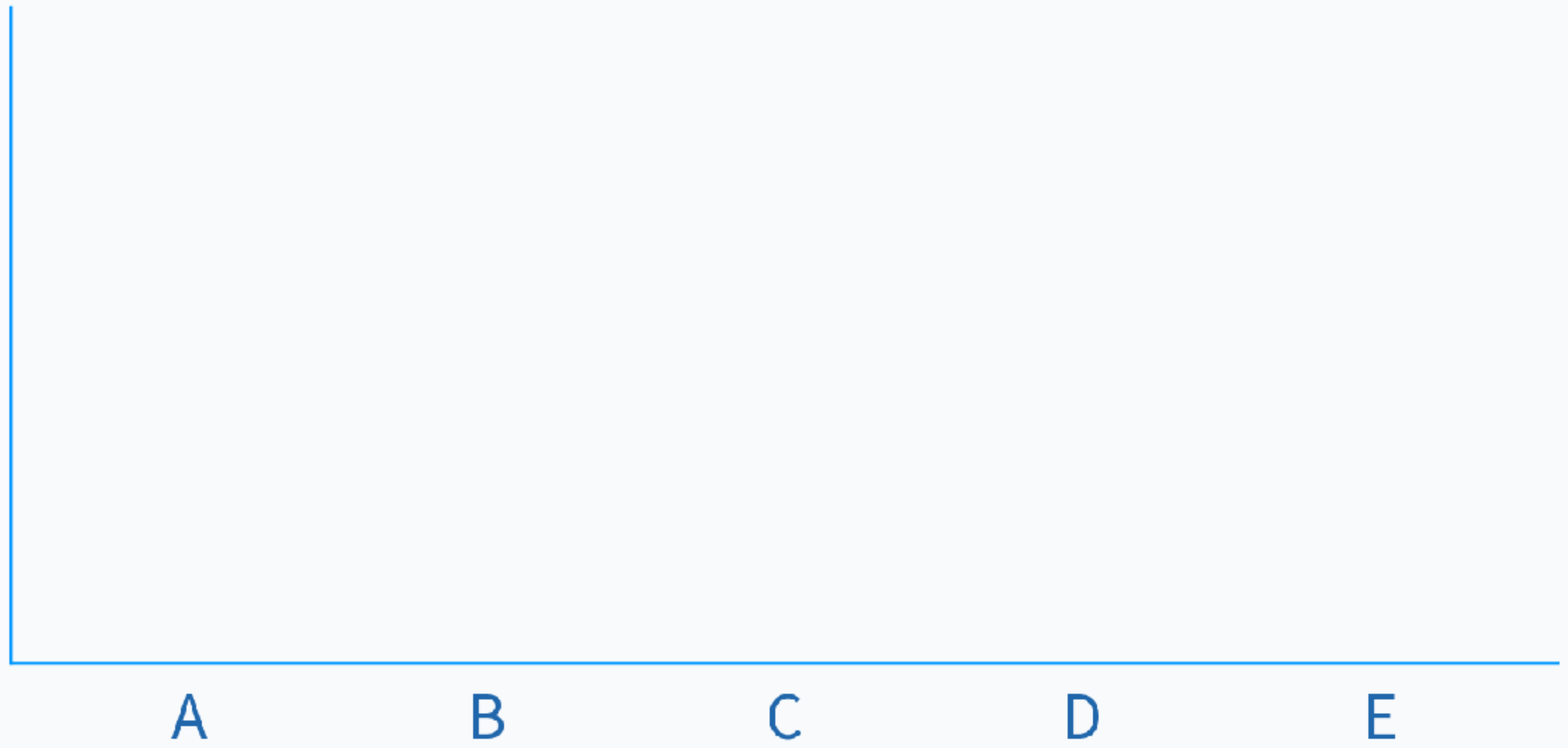
- Consider the following dynamic instructions:

```
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
```

Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (1) and (2)
B. (3) and (4)
C. (3) and (6)
D. (4) and (7)
E. (6) and (7)

CodeOptimization2



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

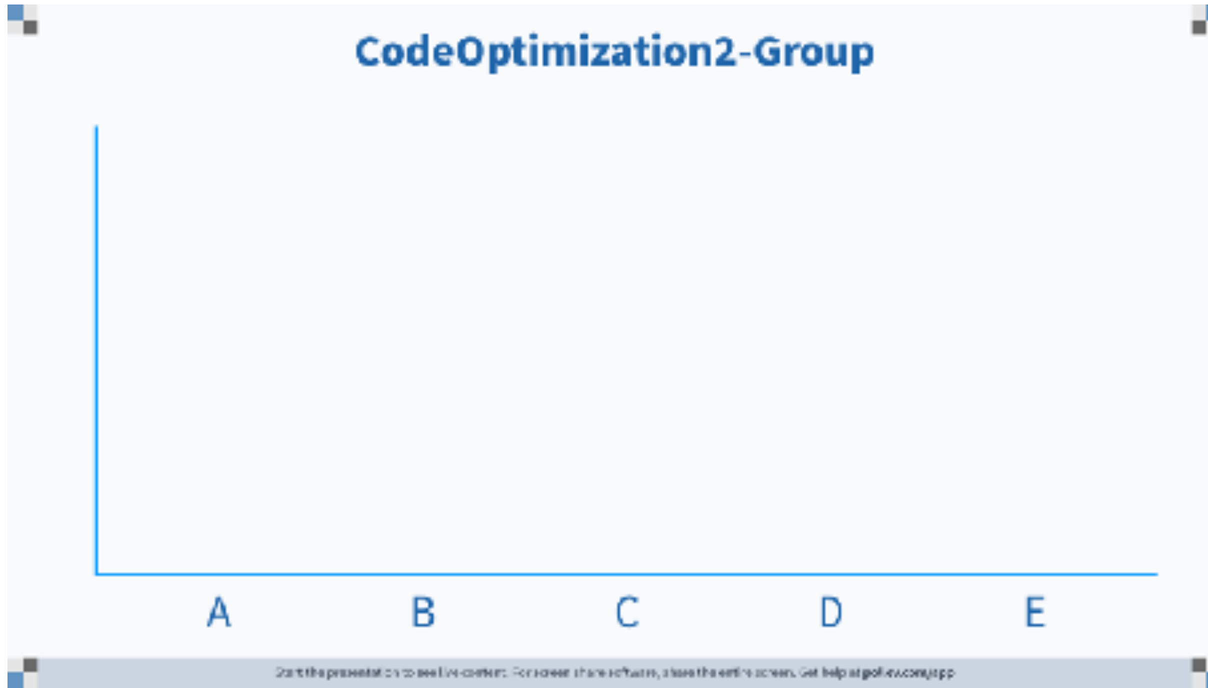
If we can predict the future …

- Consider the following dynamic instructions:

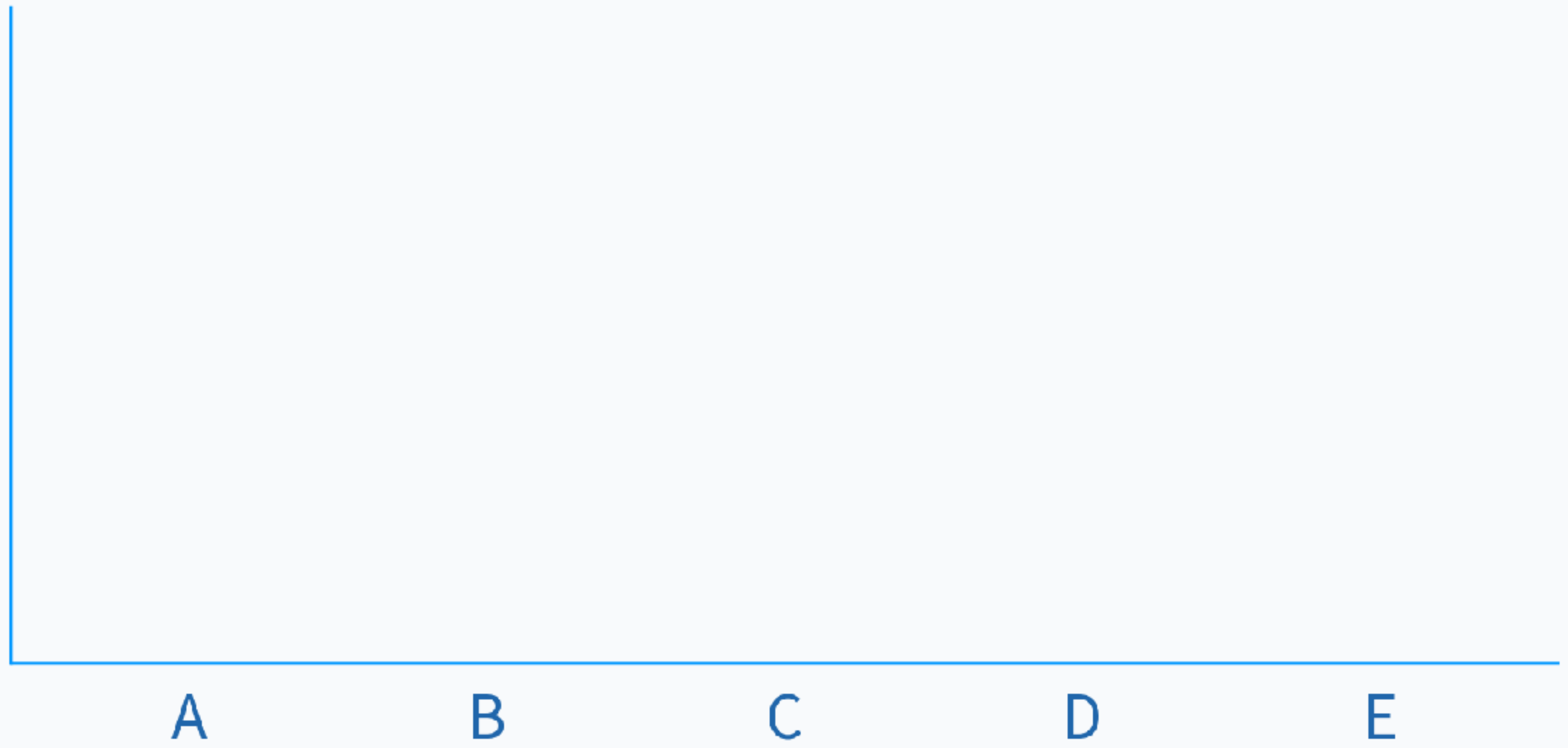
```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```

Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (1) and (2)
B. (3) and (4)
C. (3) and (6)
D. (4) and (7)
E. (6) and (7)



CodeOptimization2-Group



If we can predict the future ...

- Consider the following dynamic 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
```

Can we use "branch prediction" to predict the future and reorder instructions across the branch?

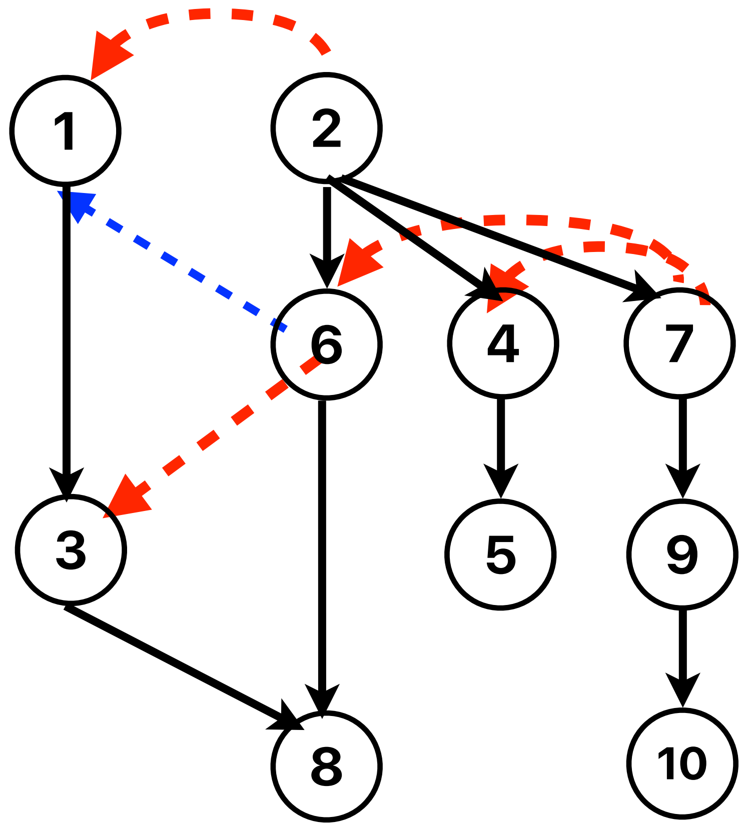
Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (1) and (2)
[✓ B. (3) and (4)]
C. (3) and (6)
D. (4) and (7)
E. (6) and (7)

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!!!

Limitations of Compiler Optimizations

- If the hardware (e.g., pipeline changes), the same compiler optimization may not be that helpful
- The compiler can only optimize on static instructions, but cannot optimize dynamic instructions
- Compilers are limited by the registers an ISA provides

Register renaming + speculative execution

- K. C. Yeager, “The Mips R10000 superscalar microprocessor,” in IEEE Micro, vol. 16, no. 2, pp. 28-41, April 1996.

Register renaming

- Provide a set of **physical registers** and a mapping table mapping **architectural registers** to physical registers
  - Architectural registers are virtual registers that software can see/use
- Allocate a physical register for a new output
- Stages
  - Dispatch/Rename (REN) — allocate a “physical register” for the output of a decoded instruction
  - Execute (EX, M1/M2, BR) — send the instruction to its corresponding pipeline if no structural hazards
  - Write Back (WB) — broadcast the result through CDB

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.

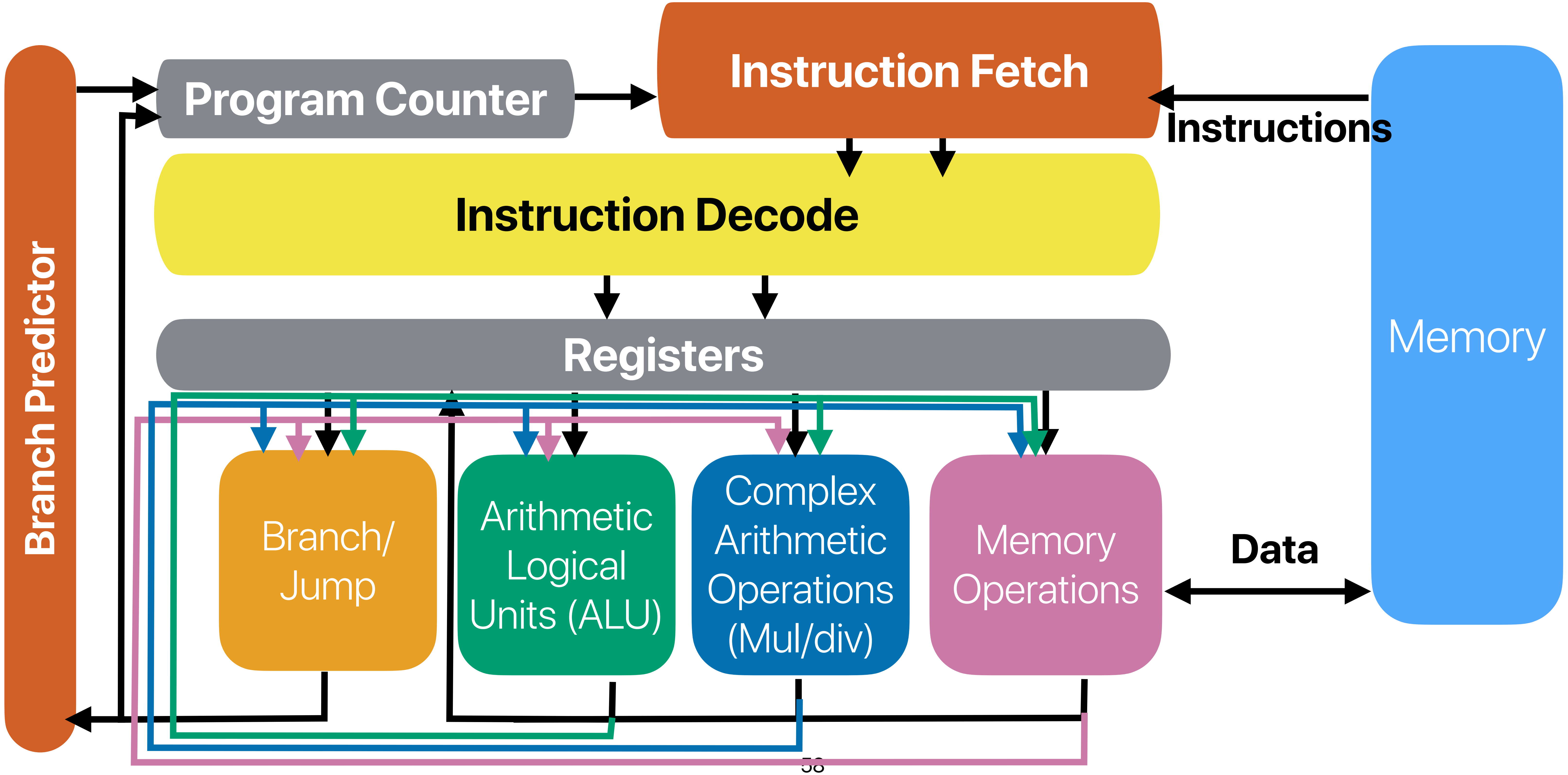
Problems with Speculative Execution

- Any execution of an instruction before a prior instruction finishes is considered as **speculative execution**
- Because it’s speculative, we need to preserve the capability to restore to the states before it’s executed
    - Branch mis-prediction
    - Exceptions
        - Page fault
        - Divided by zero and etc…

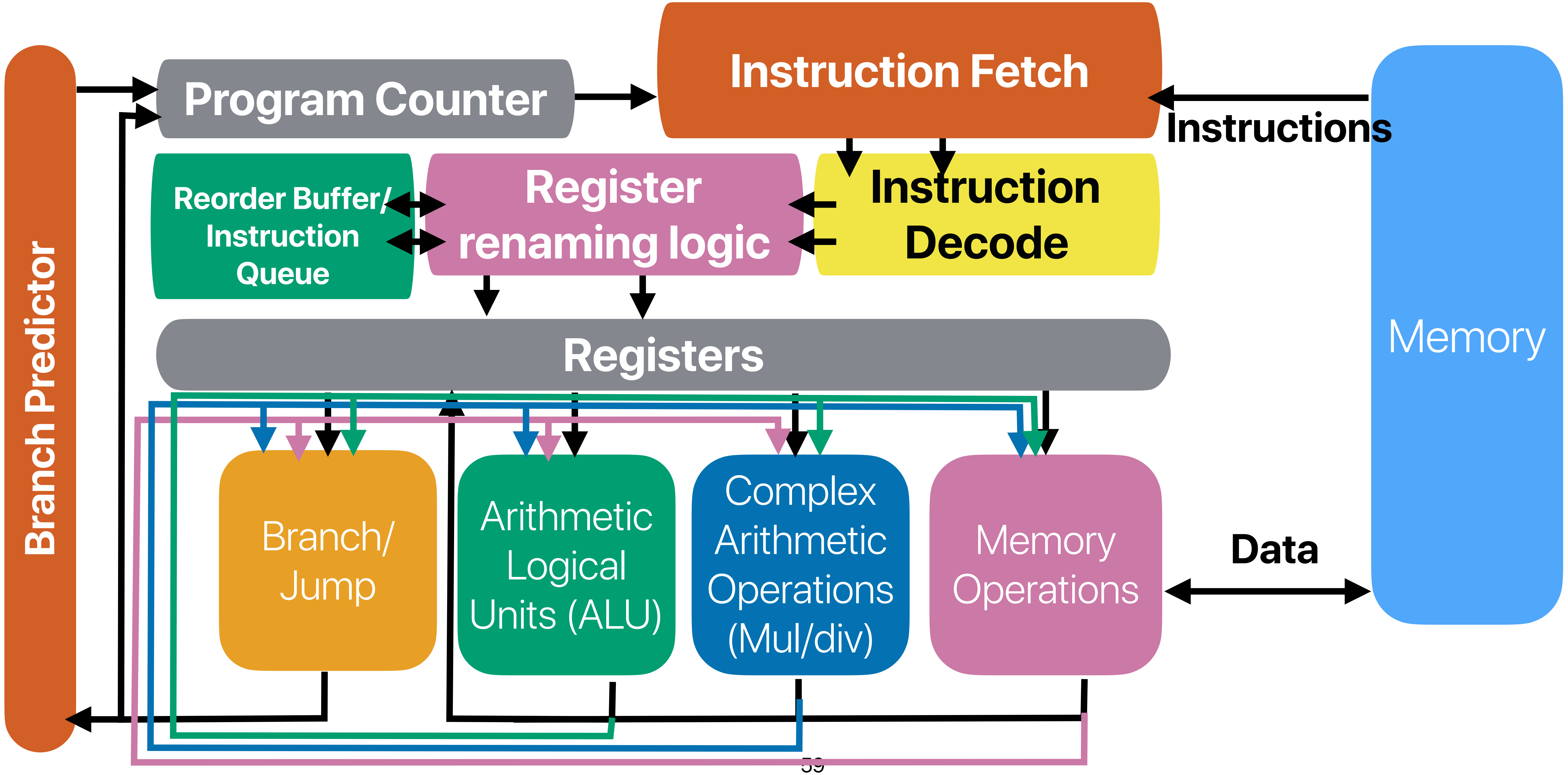
Why Reorder Buffer and In-Order Commit

- Reorder buffer — a buffer keep track of the program order of instructions
    - Can be combined with IQ or physical registers — make either as a circular queue
- Commit/WB stage — should the outcome of an instruction be realized
    - An instruction can only leave the pipeline if all it's previous are committed
    - If any prior instruction failed to commit, the instruction should yield it's ROB entry, restore all it's architectural changes

Super Scalar

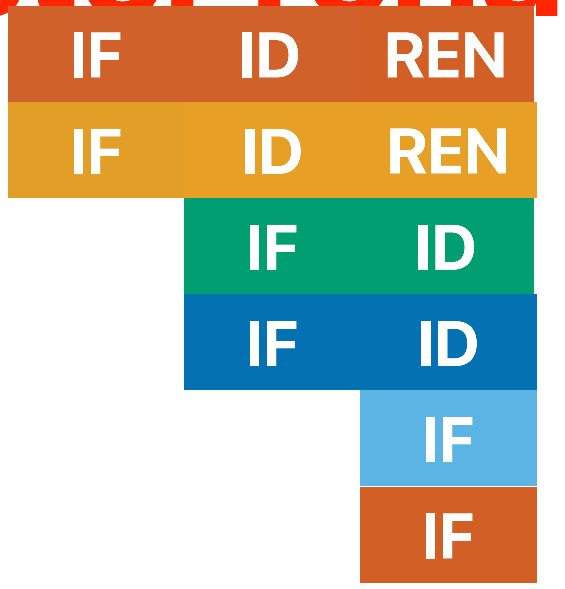


Register renaming



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
```



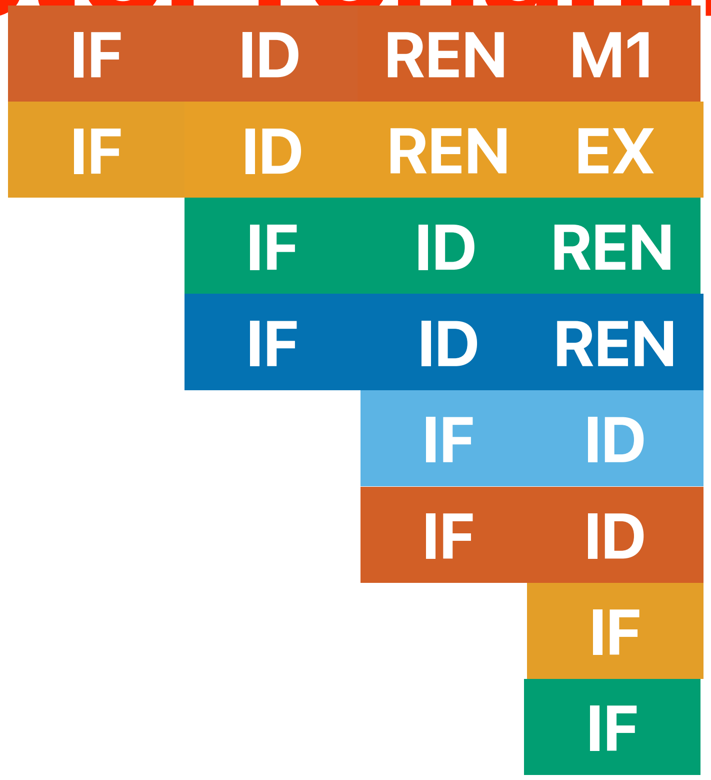
	Renamed instruction
1	`movl (%rdi), P1`
2	`addq 4, P2`
3	
4	
5	
6	
7	
8	
9	
10	

	Physical Register
eax	
ecx	P1
rdi	P2
rdx	

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

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
```



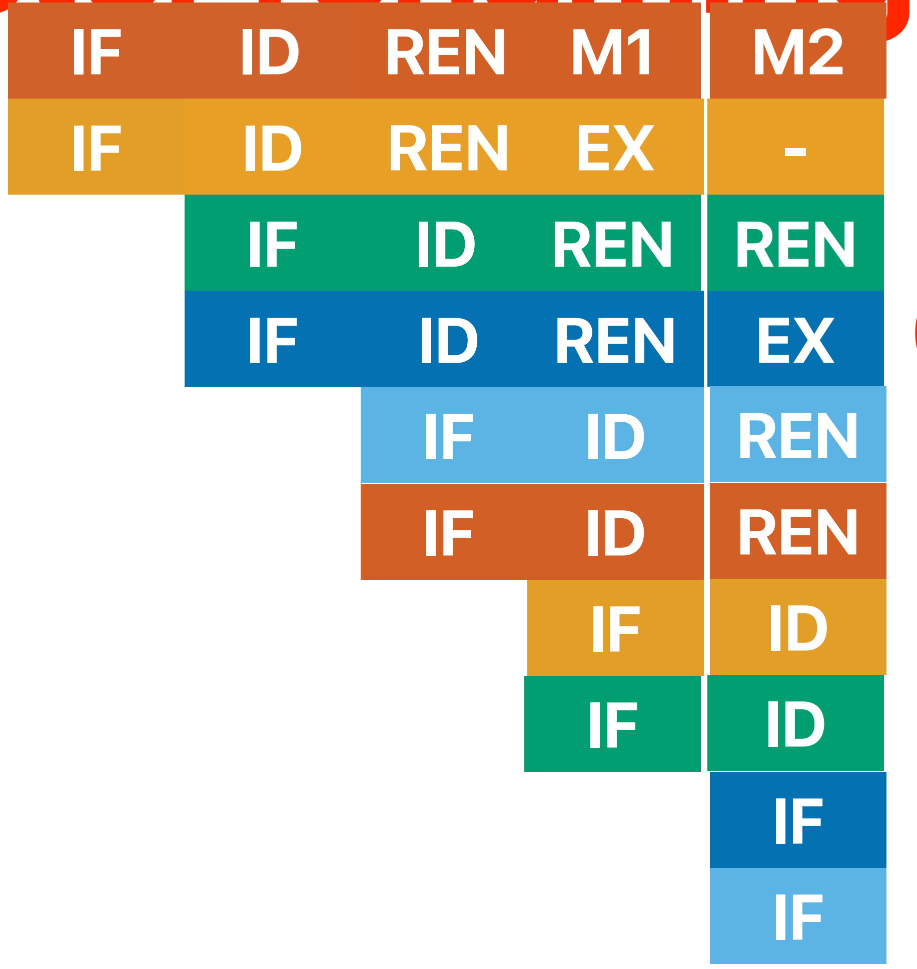
	Renamed instruction
1	`movl (%rdi), P1`
2	`addq $4,%rdi, P2`
3	`addl P1, %eax, P3`
4	`cmpq %rdx, P2`
5	
6	
7	
8	
9	
10	

	Physical Register
eax	P3
ecx	P1
rdi	P2
rdx	

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

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)!

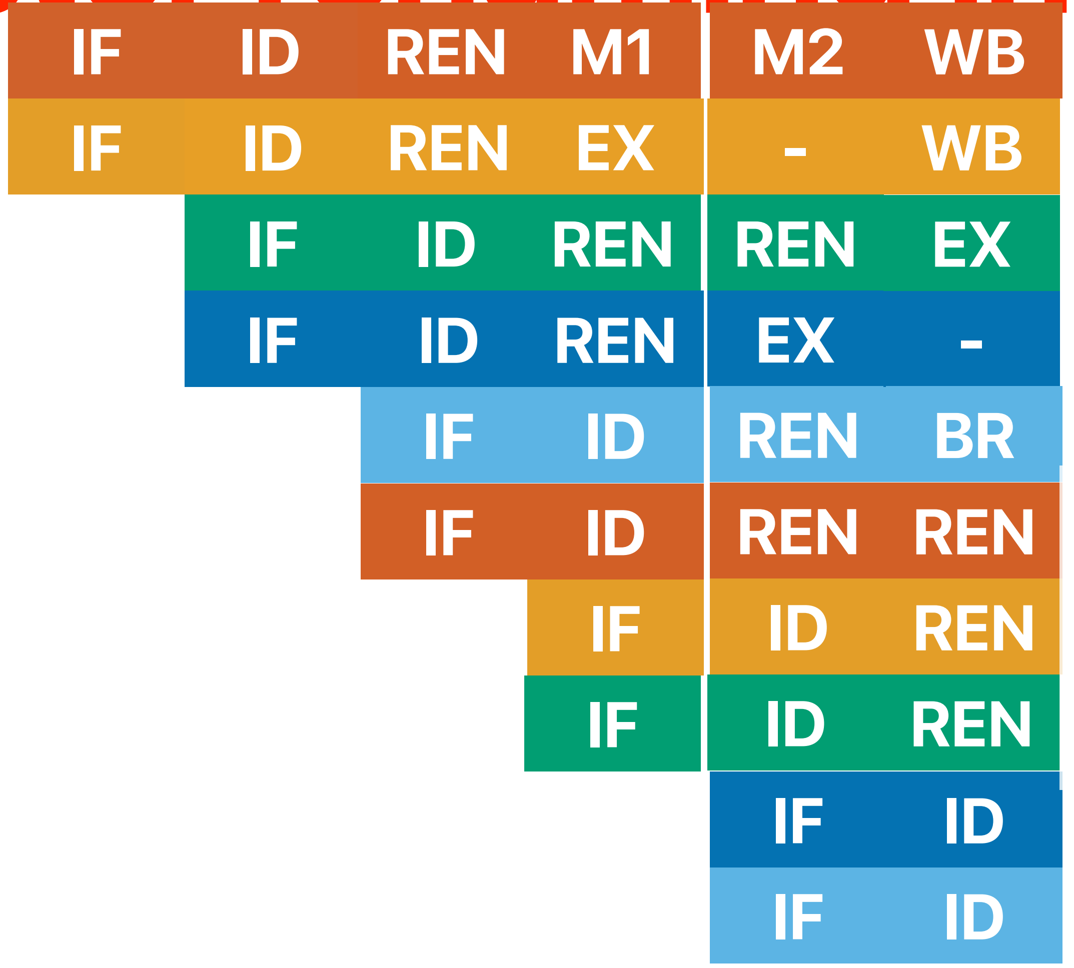
	Renamed instruction
1	movl (%rdi), P1
2	addq $4,%rdi, P2
3	addl P1, %eax, P3
4	cmpq %rdx, P2
5	jne .L3
6	movl (P2), P4
7	
8	
9	
10	

	Physical Register
eax	P3
ecx	P4
rdi	P2
rdx	

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

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
```



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

	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	
10	

	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	0		1	P9			
P5	0		1	P10			

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
```

IF	ID	REN	M1	M2	WB	
IF	ID	REN	EX	-	WB	
	IF	ID	REN	REN	EX	-
	IF	ID	REN	EX	-	-
		IF	ID	REN	BR	-
		IF	ID	REN	REN	M1
			IF	ID	REN	EX
			IF	ID	REN	REN
				IF	ID	REN
				IF	ID	REN

	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	0		1	P9			
P5	0		1	P10			

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
```

IF	ID	REN	M1	M2	WB		
IF	ID	REN	EX	-	WB		
	IF	ID	REN	REN	EX	-	WB
	IF	ID	REN	EX	-	WB	WB
		IF	ID	REN	BR	-	WB
		IF	ID	REN	REN	M1	M2
			IF	ID	REN	EX	-
			IF	ID	REN	REN	REN
				IF	ID	REN	EX
				IF	ID	REN	REN

	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	0		1	P9			
P5	1		1	P10			

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
```

IF	ID	REN	M1	M2	WB			
IF	ID	REN	EX	-	WB			
	IF	ID	REN	REN	EX	-	WB	
	IF	ID	REN	EX	-	WB	WB	
		IF	ID	REN	BR	-	WB	WB
		IF	ID	REN	REN	M1	M2	WB
			IF	ID	REN	EX	-	WB
			IF	ID	REN	REN	REN	EX
				IF	ID	REN	EX	-
				IF	ID	REN	REN	BR

	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	1		1
P2	1		1	P7			
P3	0		1	P8			
P4	1		1	P9			
P5	1		1	P10			

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
```

IF	ID	REN	M1	M2	WB				
IF	ID	REN	EX	-	WB				
	IF	ID	REN	REN	EX	-	WB		
	IF	ID	REN	EX	-	WB	WB		
		IF	ID	REN	BR	-	WB	WB	
		IF	ID	REN	REN	M1	M2	WB	
			IF	ID	REN	EX	-	WB	WB
			IF	ID	REN	REN	REN	EX	WB
				IF	ID	REN	EX	-	-
				IF	ID	REN	REN	BR	-

	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			

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
```

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

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			

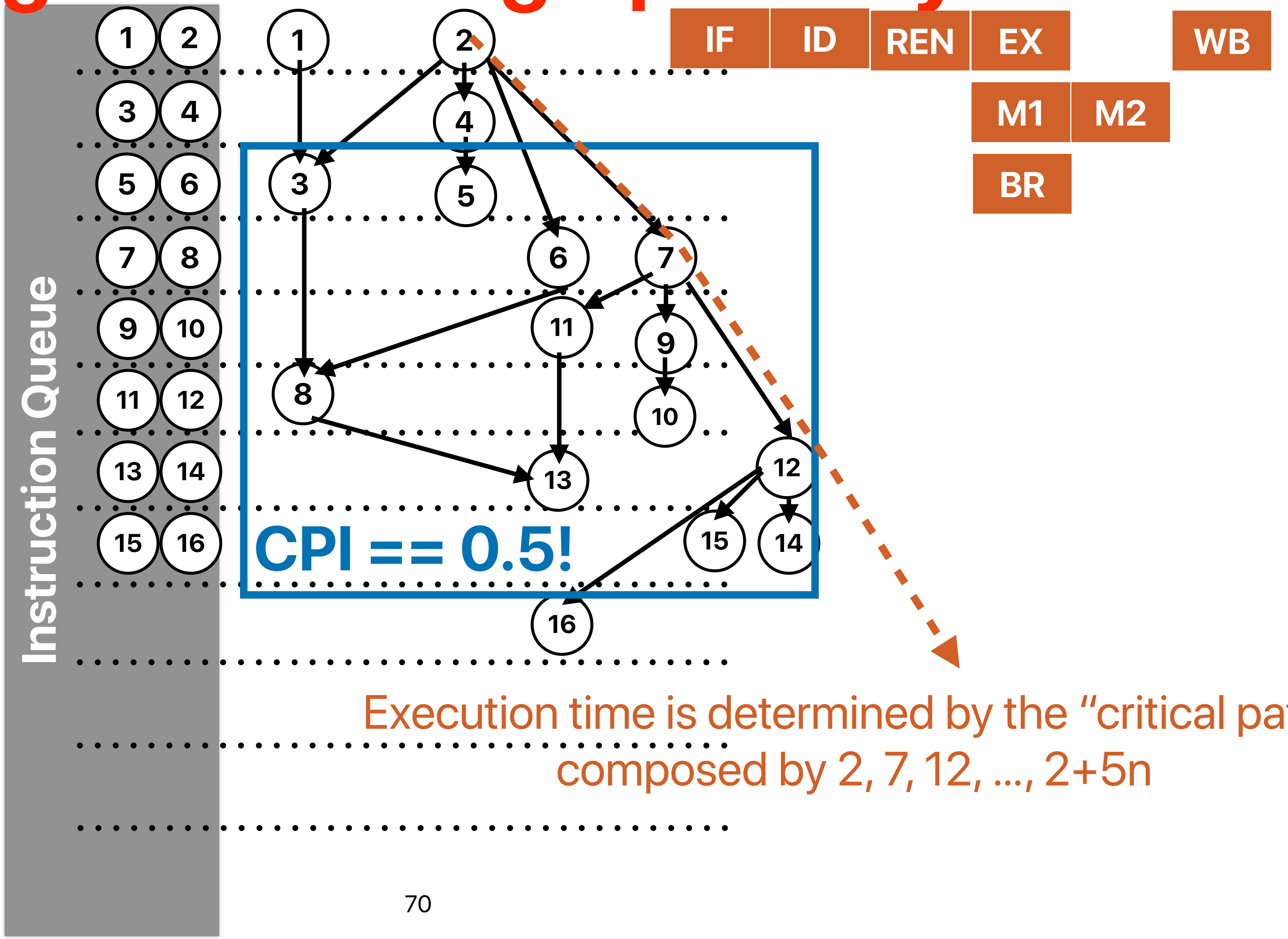
Register renaming

```
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
```

	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)

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
```



Execution time is determined by the "critical path" composed by 2, 7, 12, ..., 2+5n

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

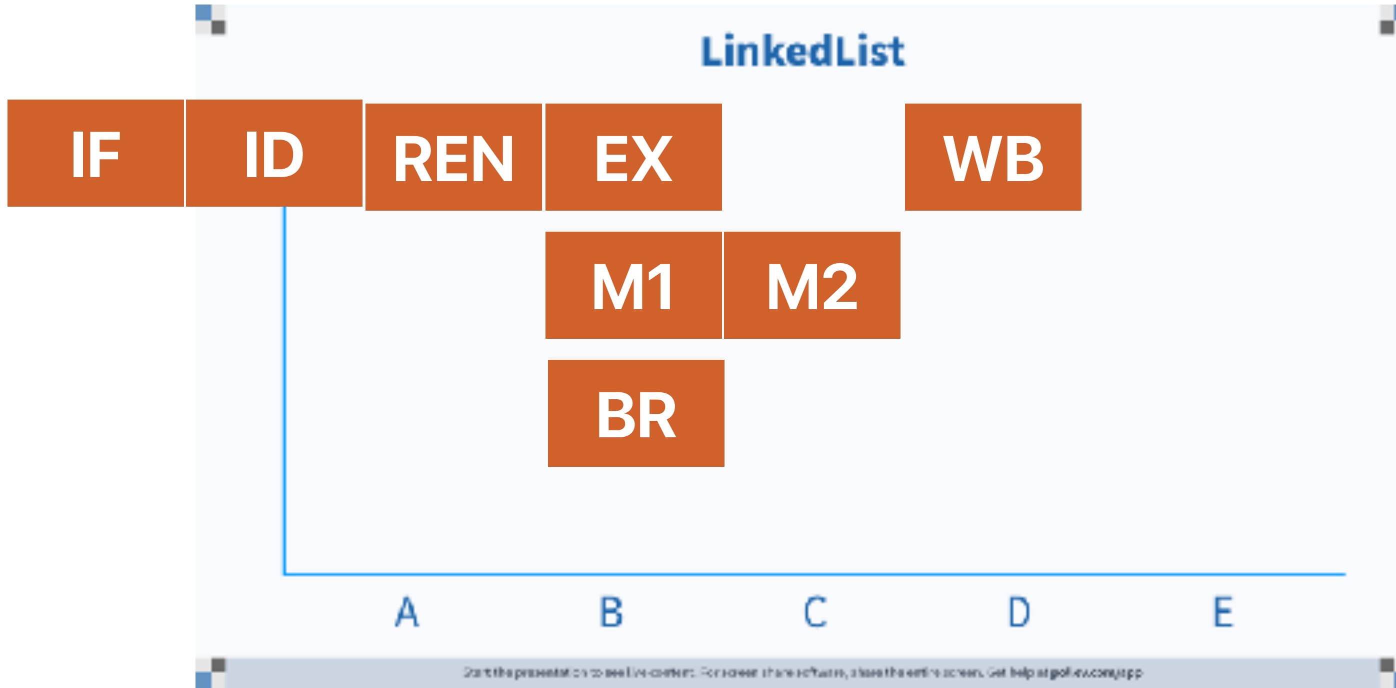
What about "linked list"

- For the following C code and it's translation in x86, how many cycles it takes the processor to issue all instructions? 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.

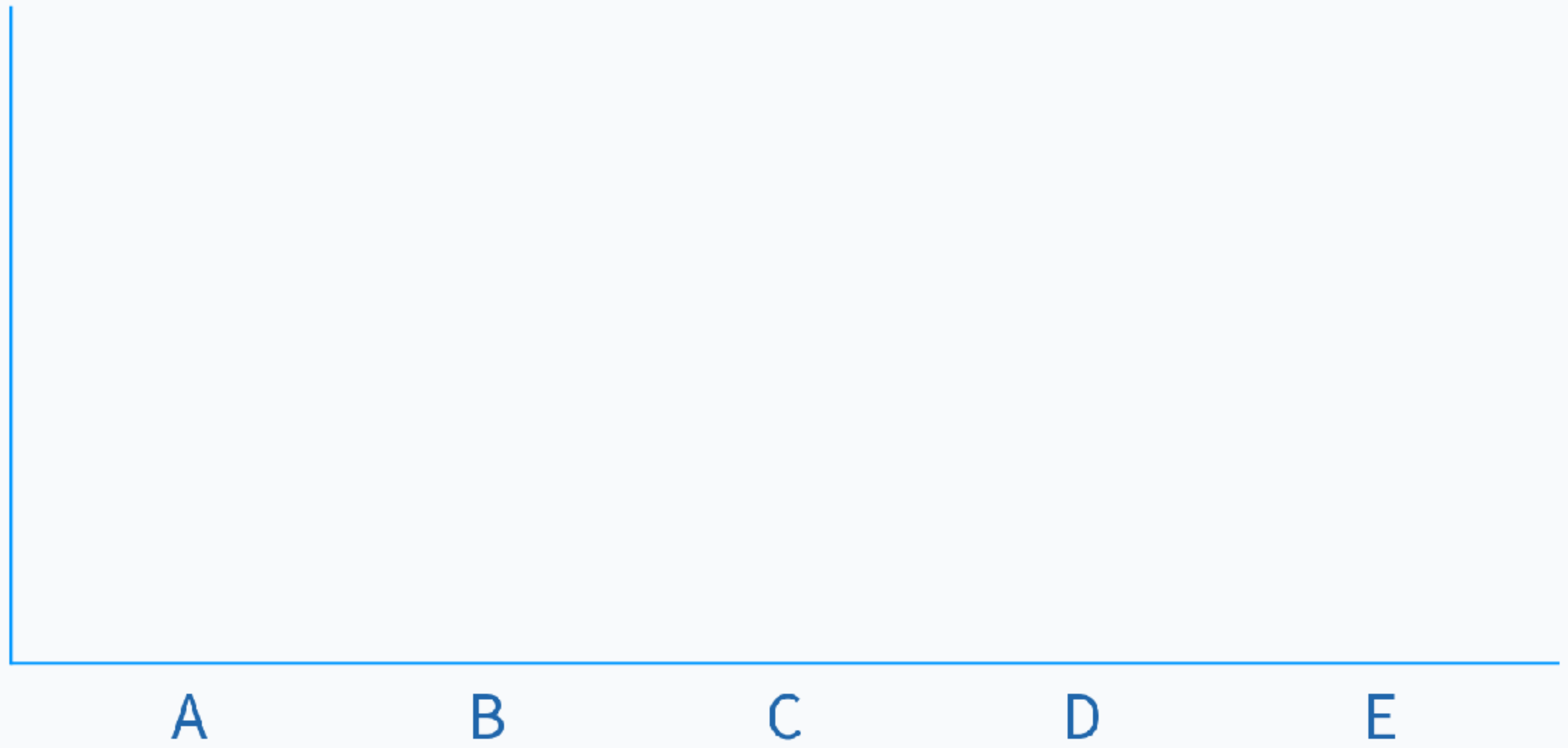
```
do {
     number_of_nodes++;
     current = current->next;
} while ( current != NULL )
```

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

A. 9
B. 10
C. 11
D. 12
E. 13



LinkedList



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

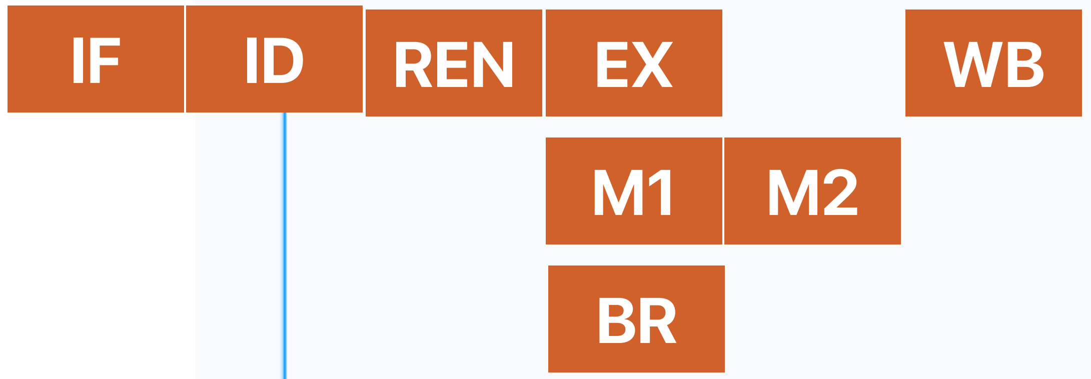
What about "linked list"

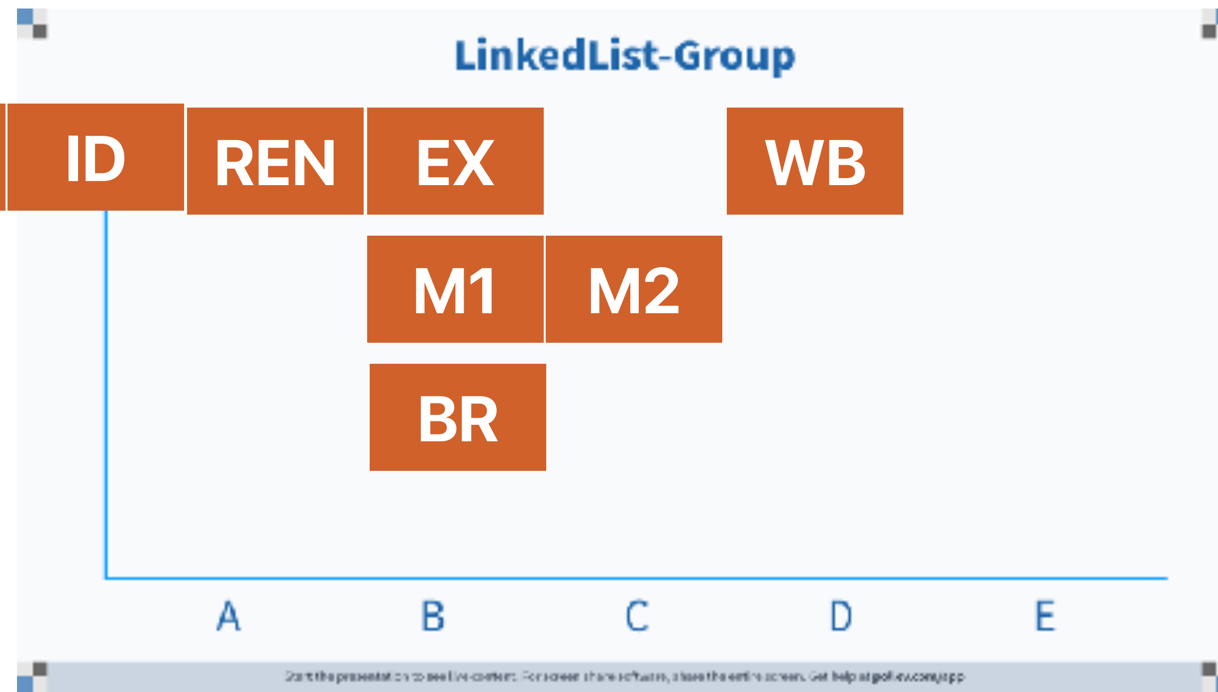
- For the following C code and it's translation in x86, how many cycles it takes the processor to issue all instructions? Assume the current PC is already at instruction (1) and this linked list has only three nodes. This processor can fetch and issue 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously.

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

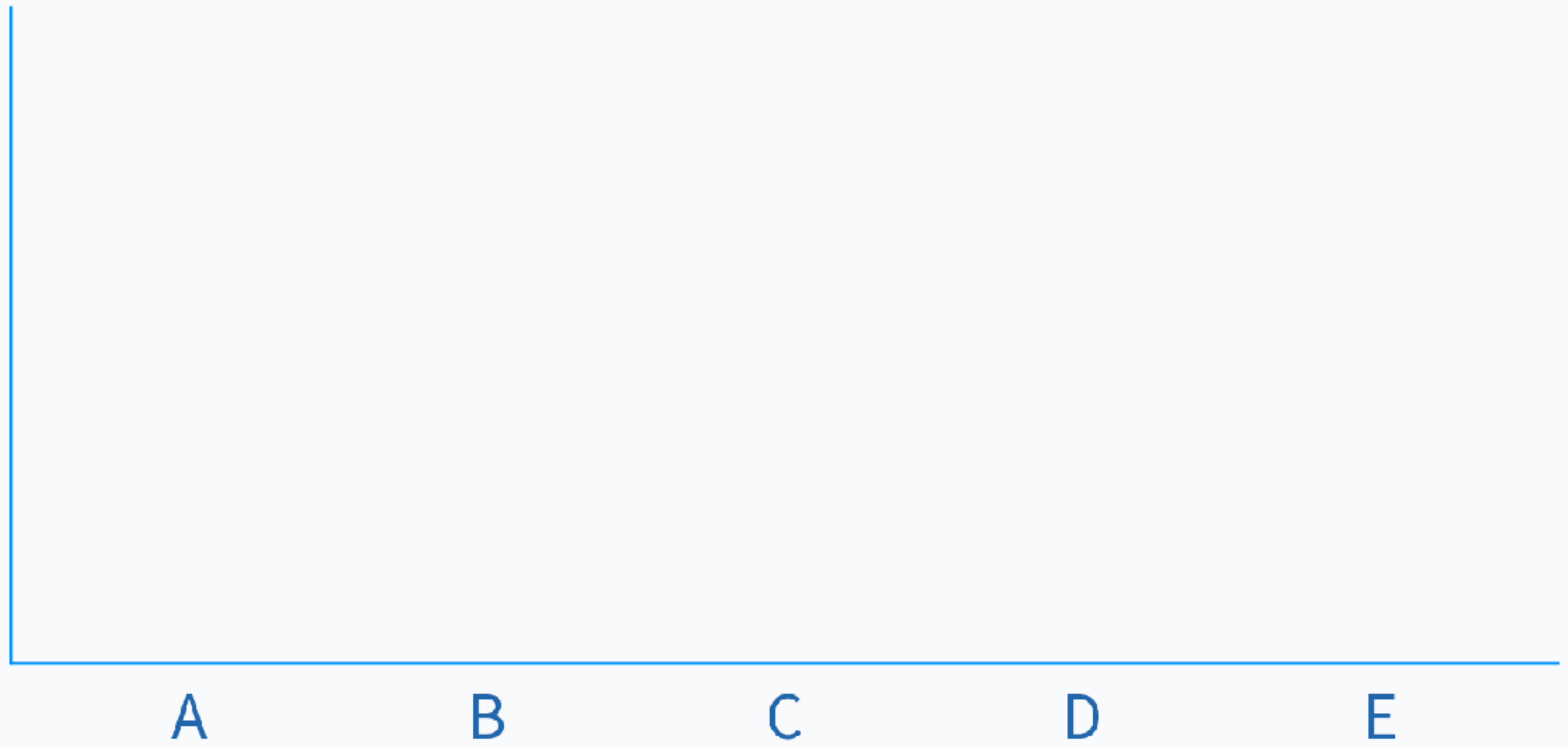
```
① .L3:    addq    $8, %rdi
②         movq    (%rdi), %rdi
③         addl    $1, %eax
④         testq   %rdi, %rdi
⑤         jne     .L3
```

A. 9
B. 10
C. 11
D. 12
E. 13





LinkedList-Group



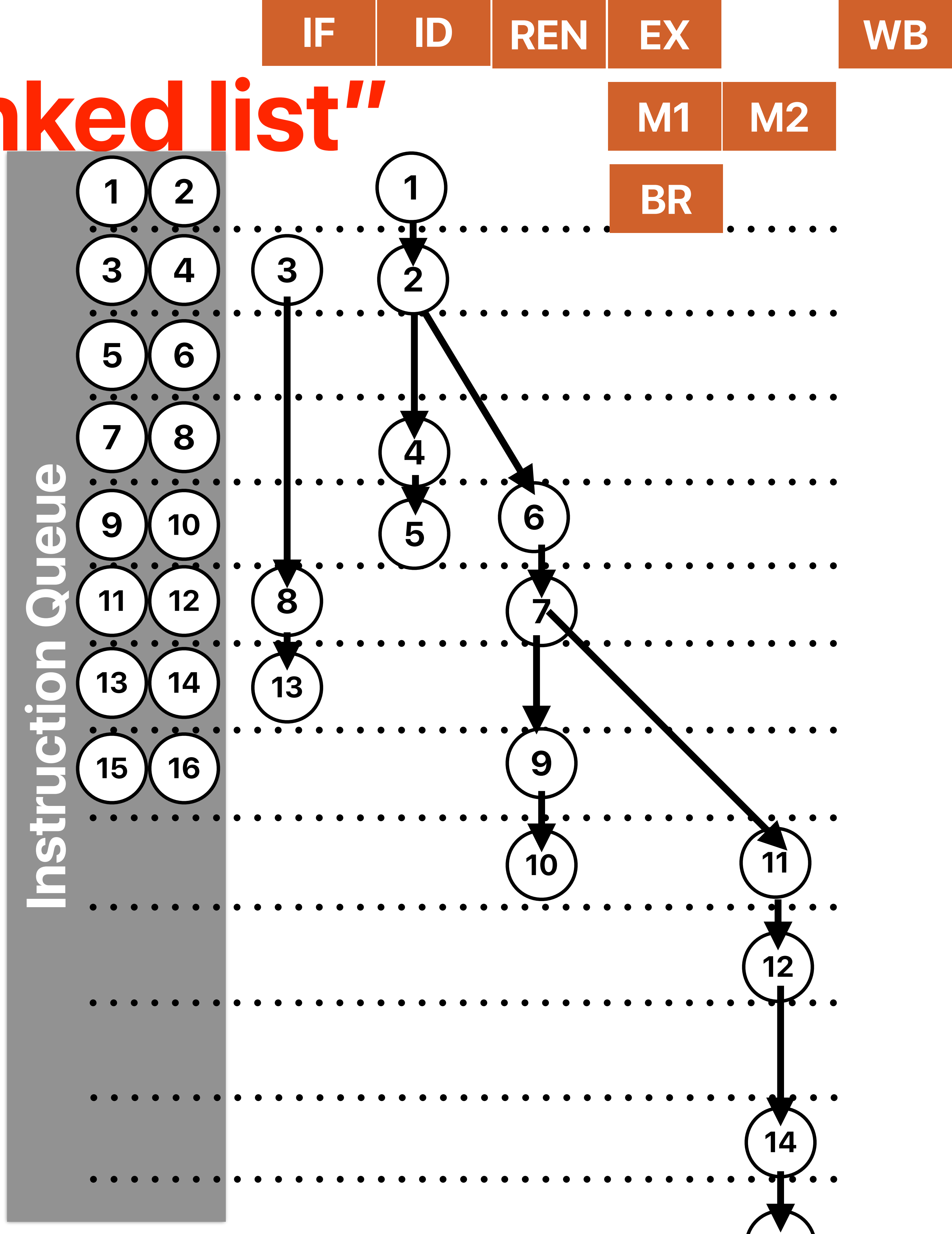
What about "linked list"

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
```



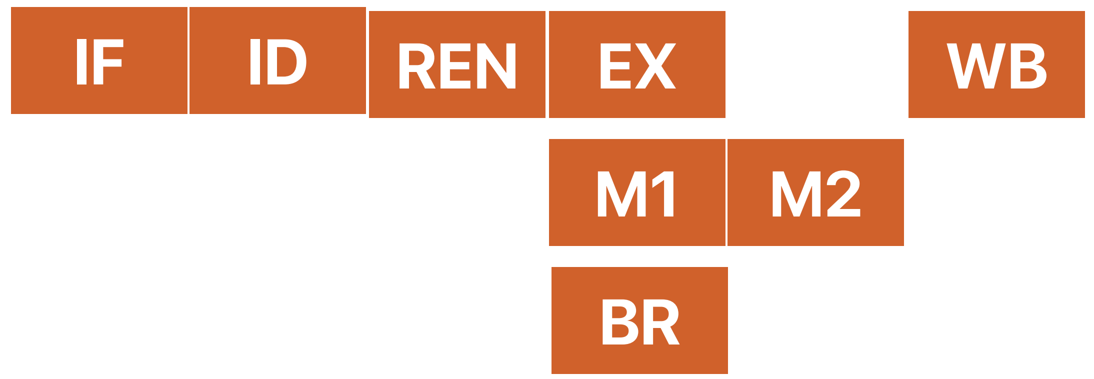
What about “linked list”

- For the following C code and it’s translation in x86, how many cycles it takes the processor to issue all instructions? 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.

```
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
```

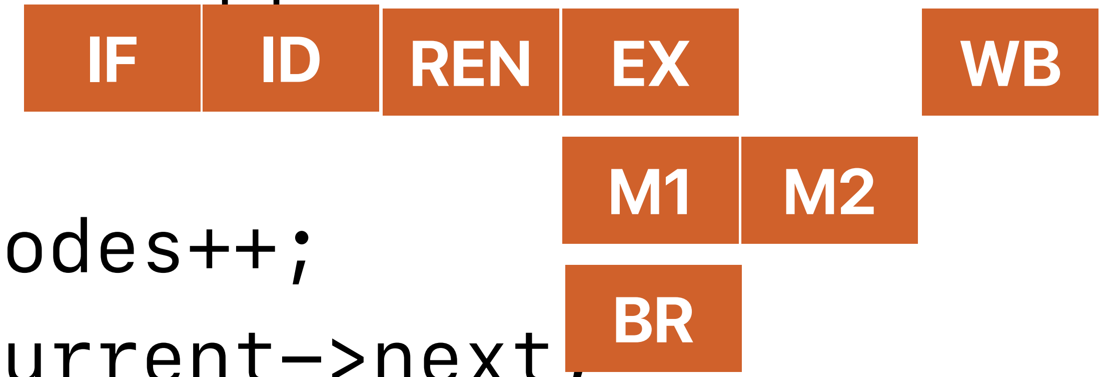
A. 9
B. 10
C. 11
D. 12
[✓ E. 13]



https://www.pollev.com/hungweitseng close in 1:00

What about "linked list"

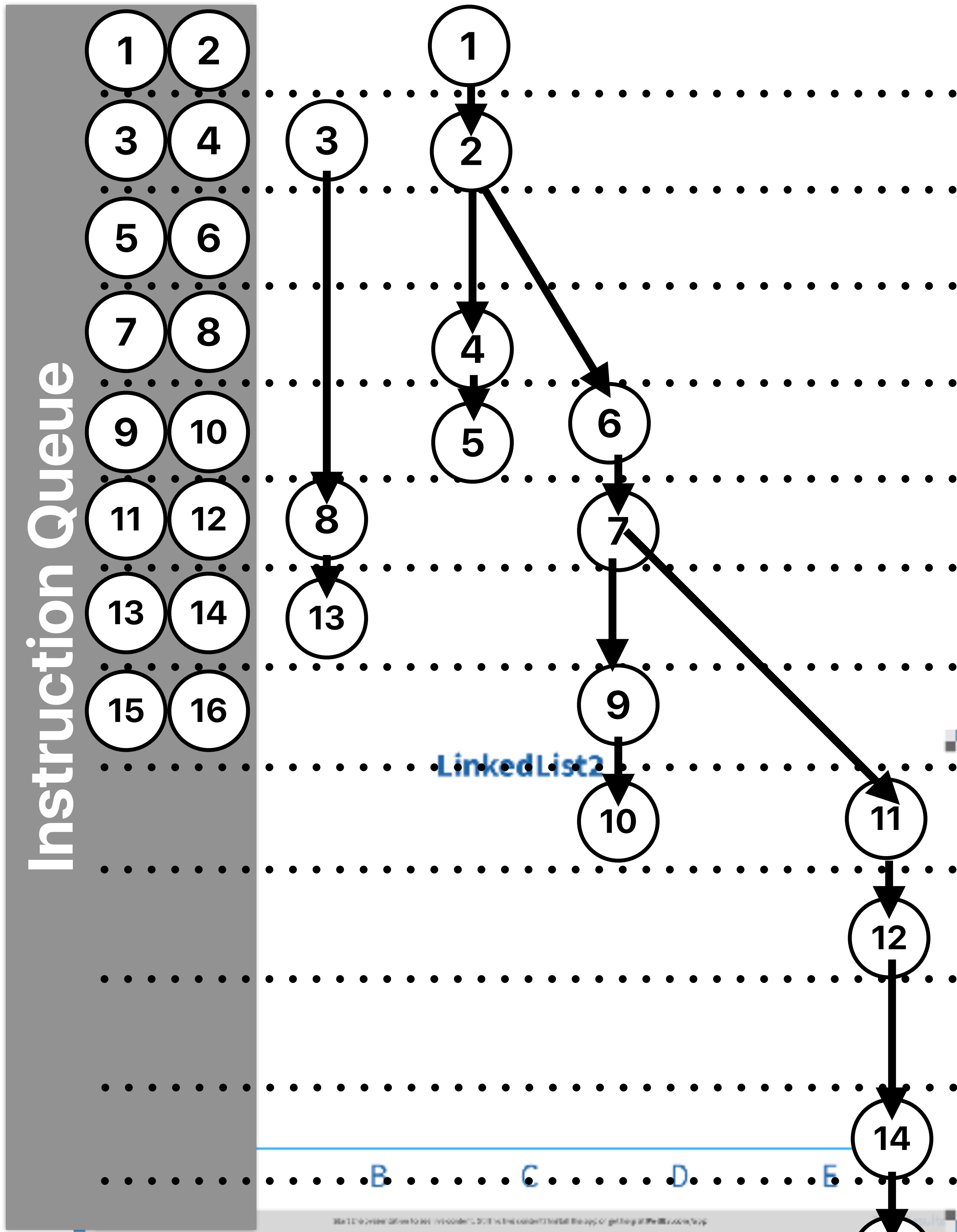
- For the following C code and it's translation in x86, **what's average CPI?** Assume the current PC is already at instruction (1) and this linked list has thousands of nodes. This processor can fetch and issue 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as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



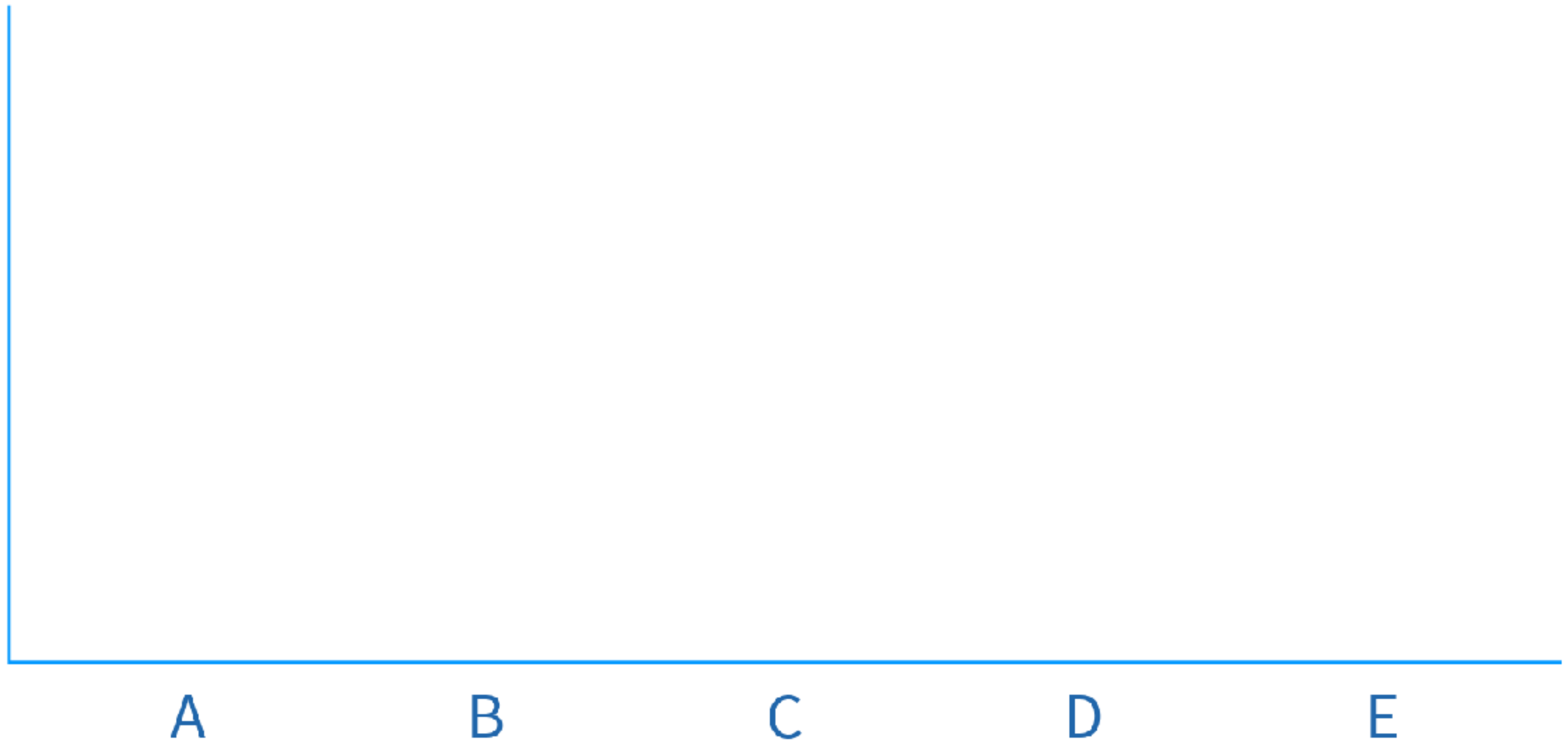
```
do {
     number_of_nodes++;
     current = current->next;
} while ( current != NULL )
```

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
```



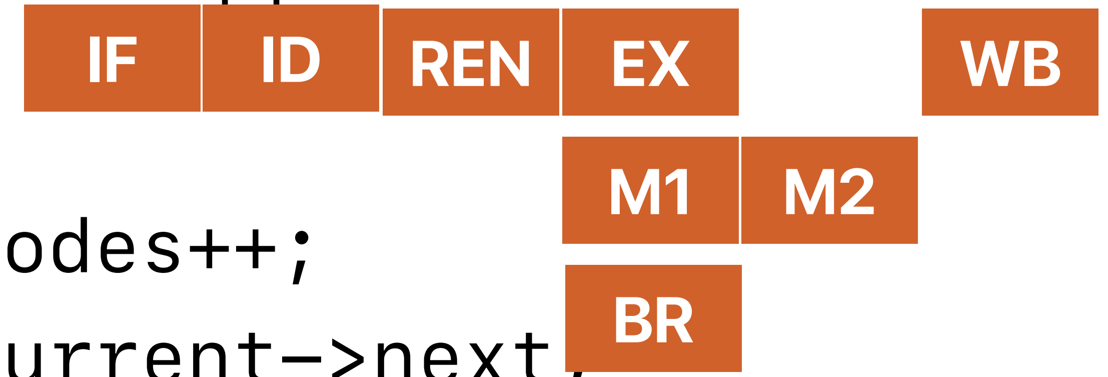
LinkedList2



https://www.pollev.com/hungweitseng close in 1:00

What about "linked list"

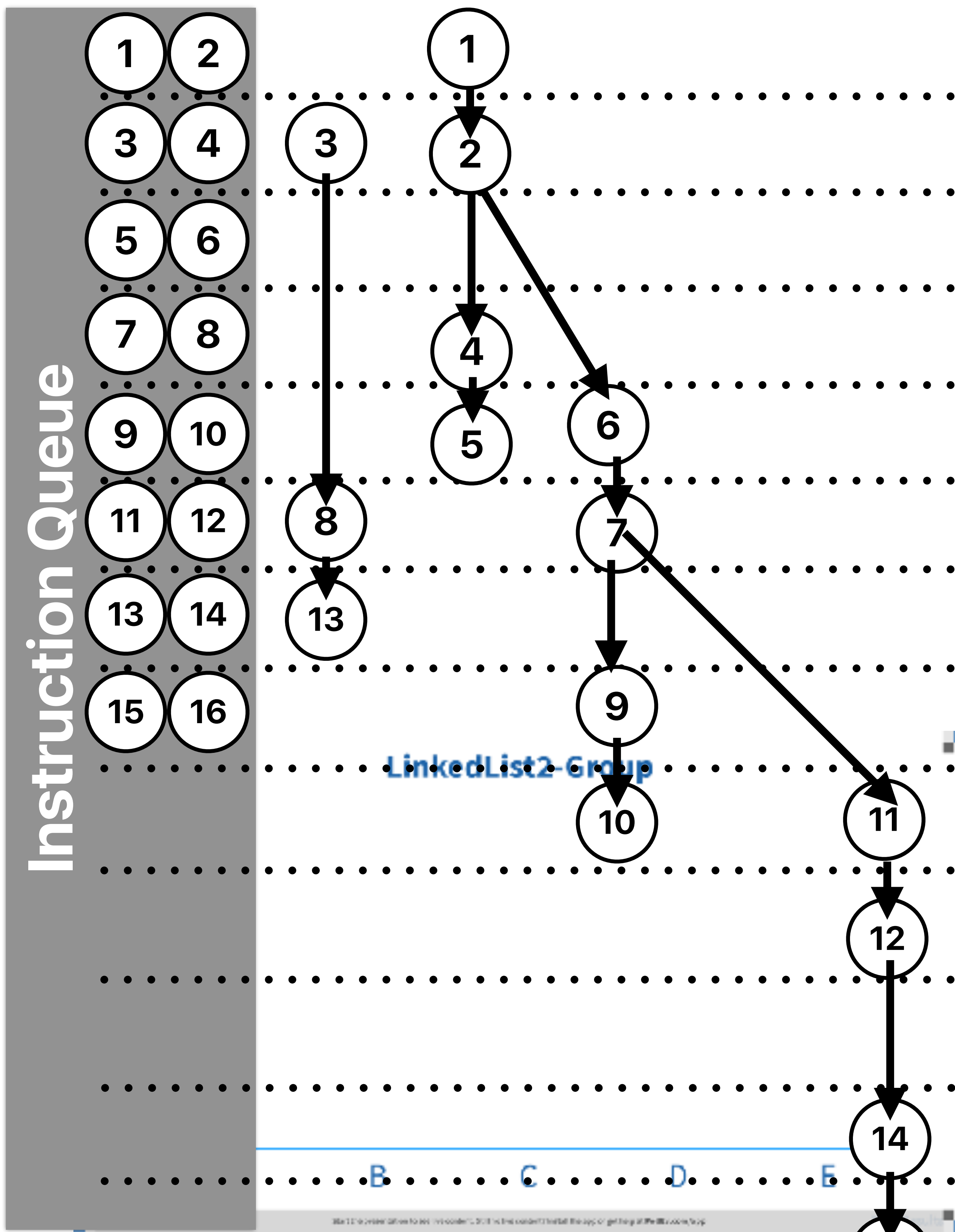
- For the following C code and it's translation in x86, **what's average CPI?** Assume the current PC is already at instruction (1) and this linked list has thousands of nodes. This processor can fetch and issue 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as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



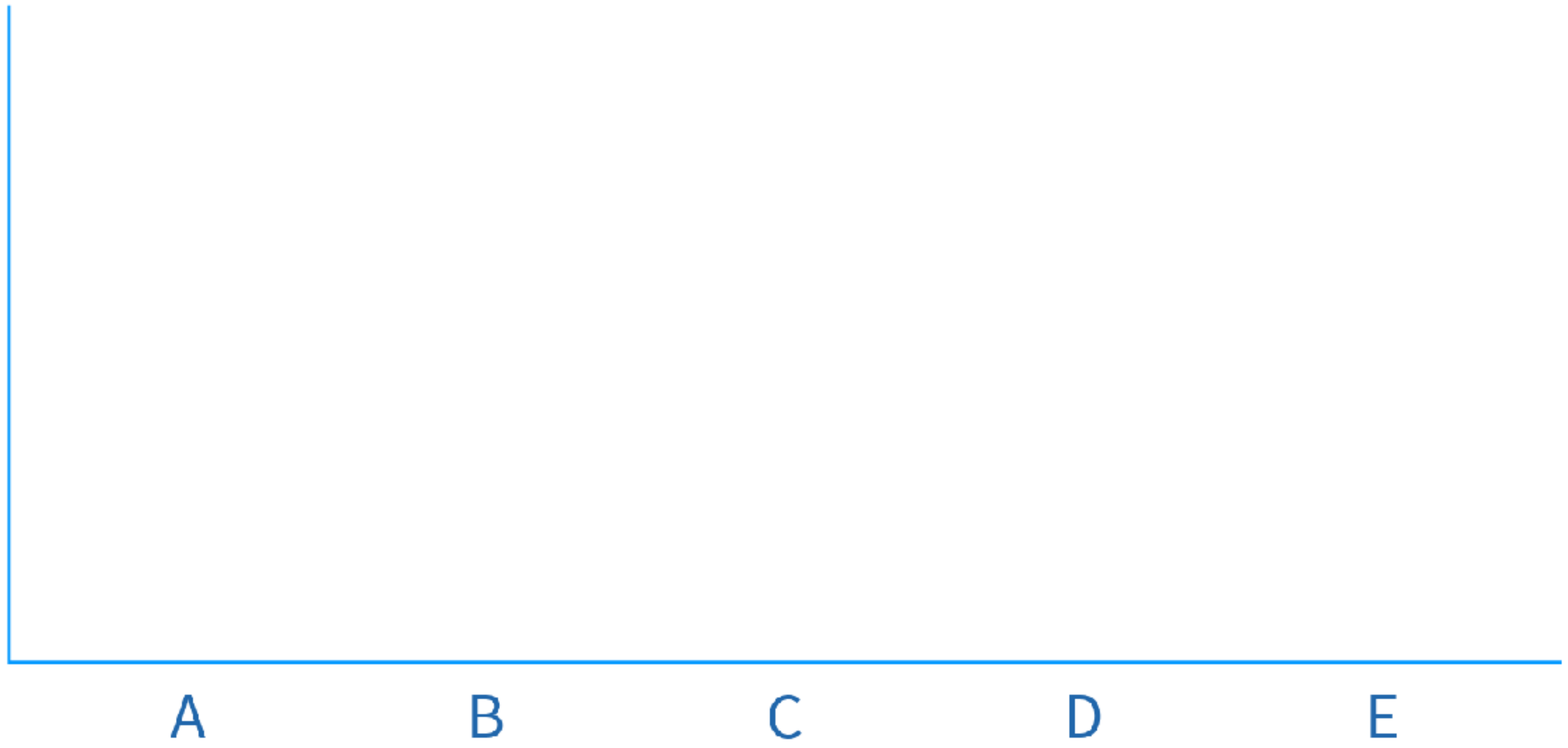
```
do {
     number_of_nodes++;
     current = current->next;
} while ( current != NULL )
```

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

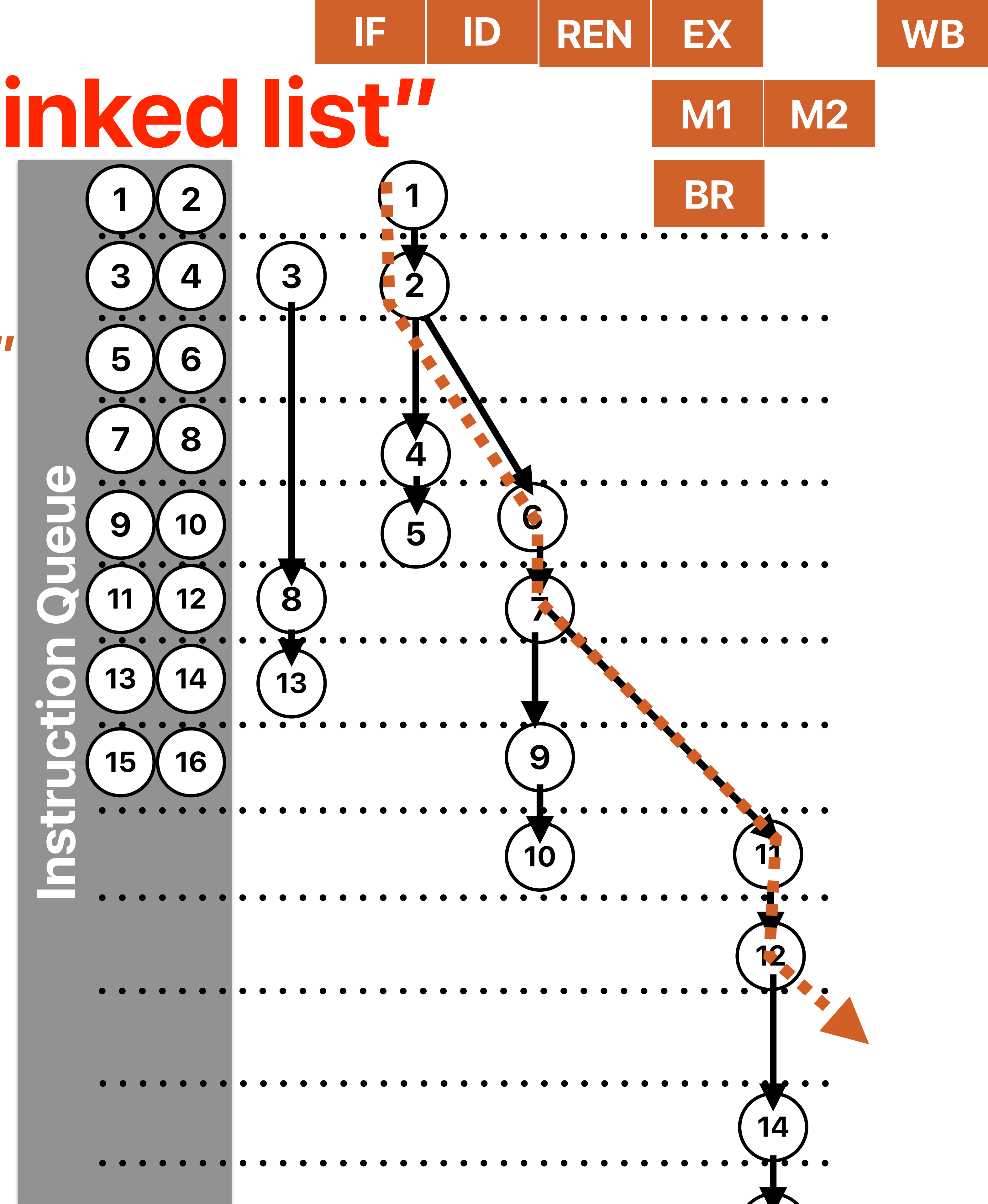
```
① .L3:     addq    $8, %rdi
②          movq    (%rdi), %rdi
③          addl    $1, %eax
④          testq   %rdi, %rdi
⑤          jne     .L3
```



LinkedList2-Group



What about "linked list"



Performance determined by the "critical path"

4 cycles each iteration

5 instructions per iteration

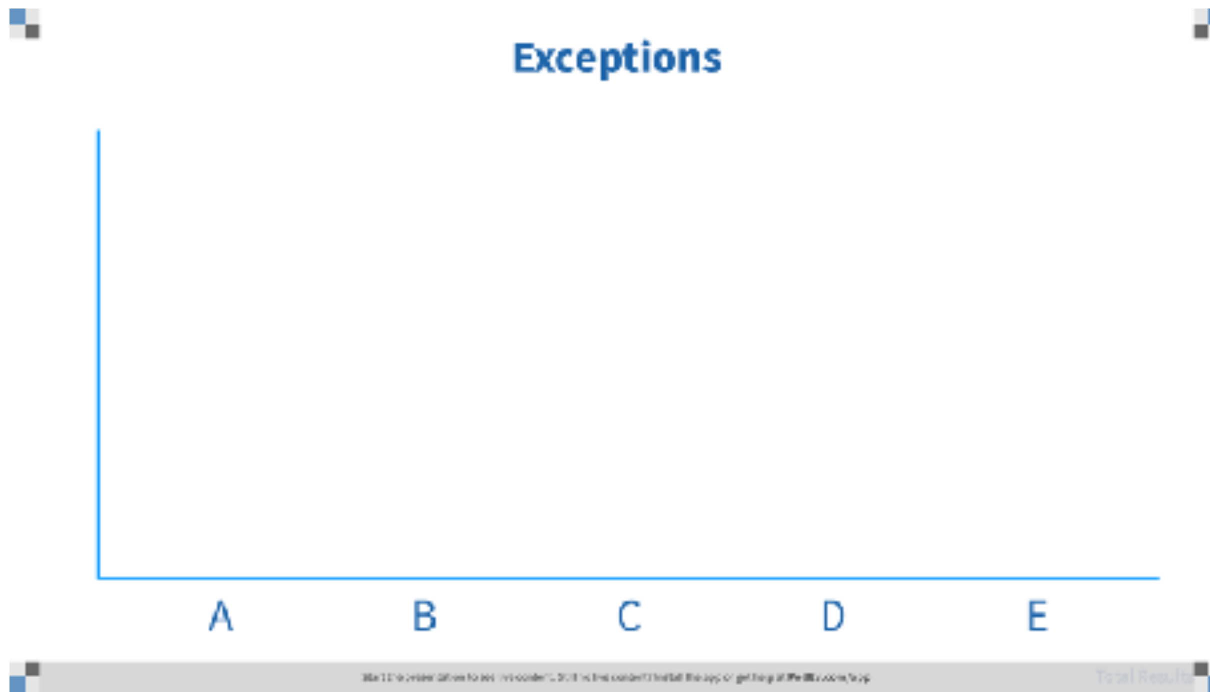
$$CPI = \frac{4}{5} = 0.8$$

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

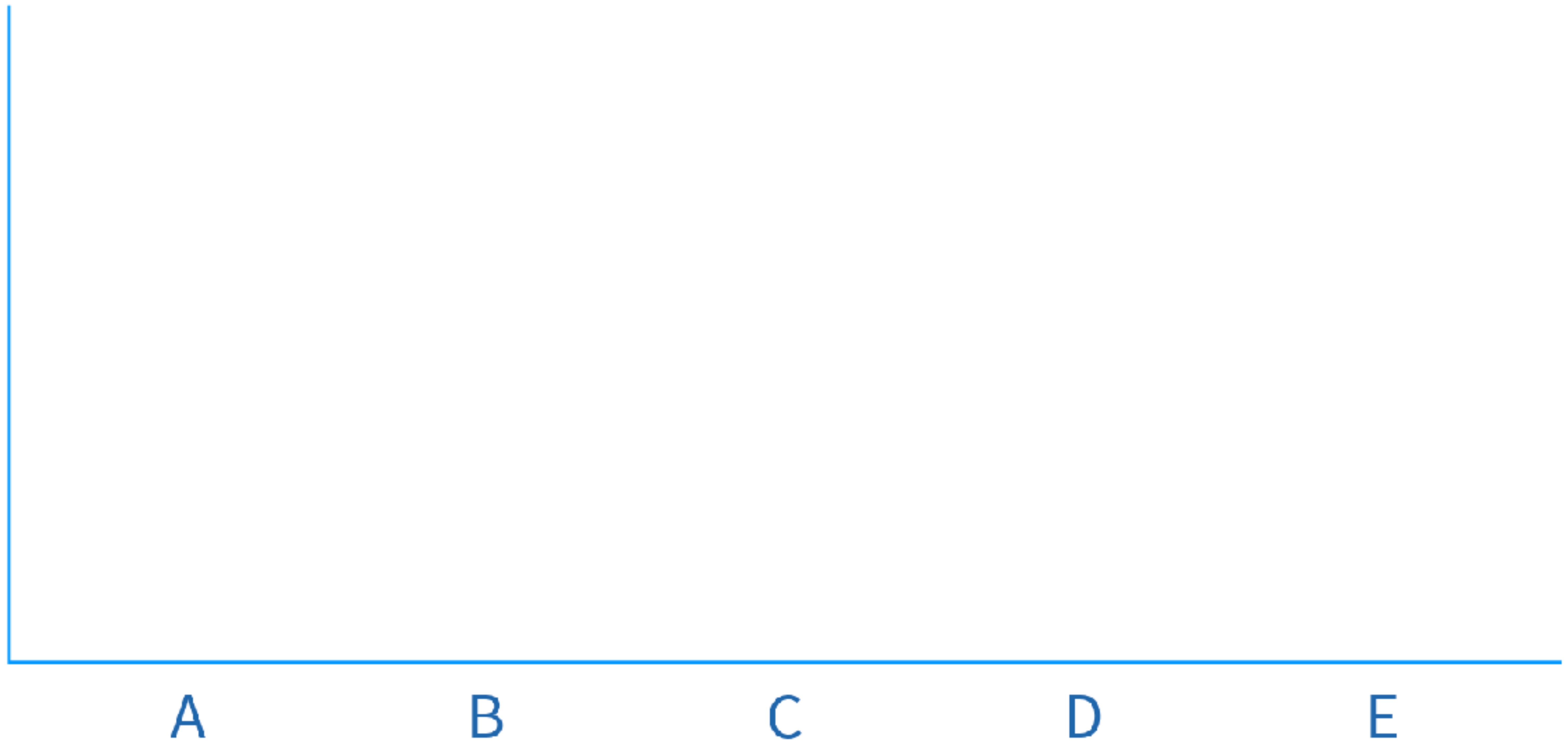
https://www.pollev.com/hungweitseng close in 1:00

In which pipeline stage can we have exceptions?

- How many of the following pipeline stages can we have exceptions?
  ① IF
  ② ID
  ③ EXE
  ④ MEM
  ⑤ WB
  A. 1
  B. 2
  C. 3
  D. 4
  E. 5



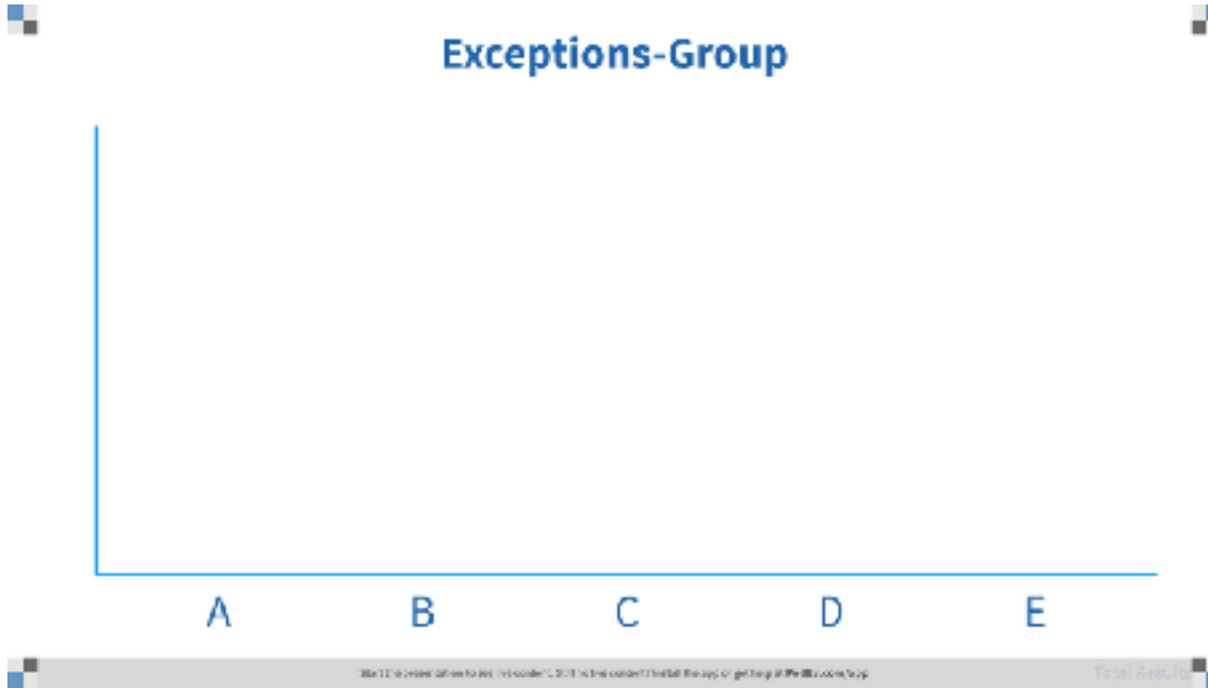
Exceptions



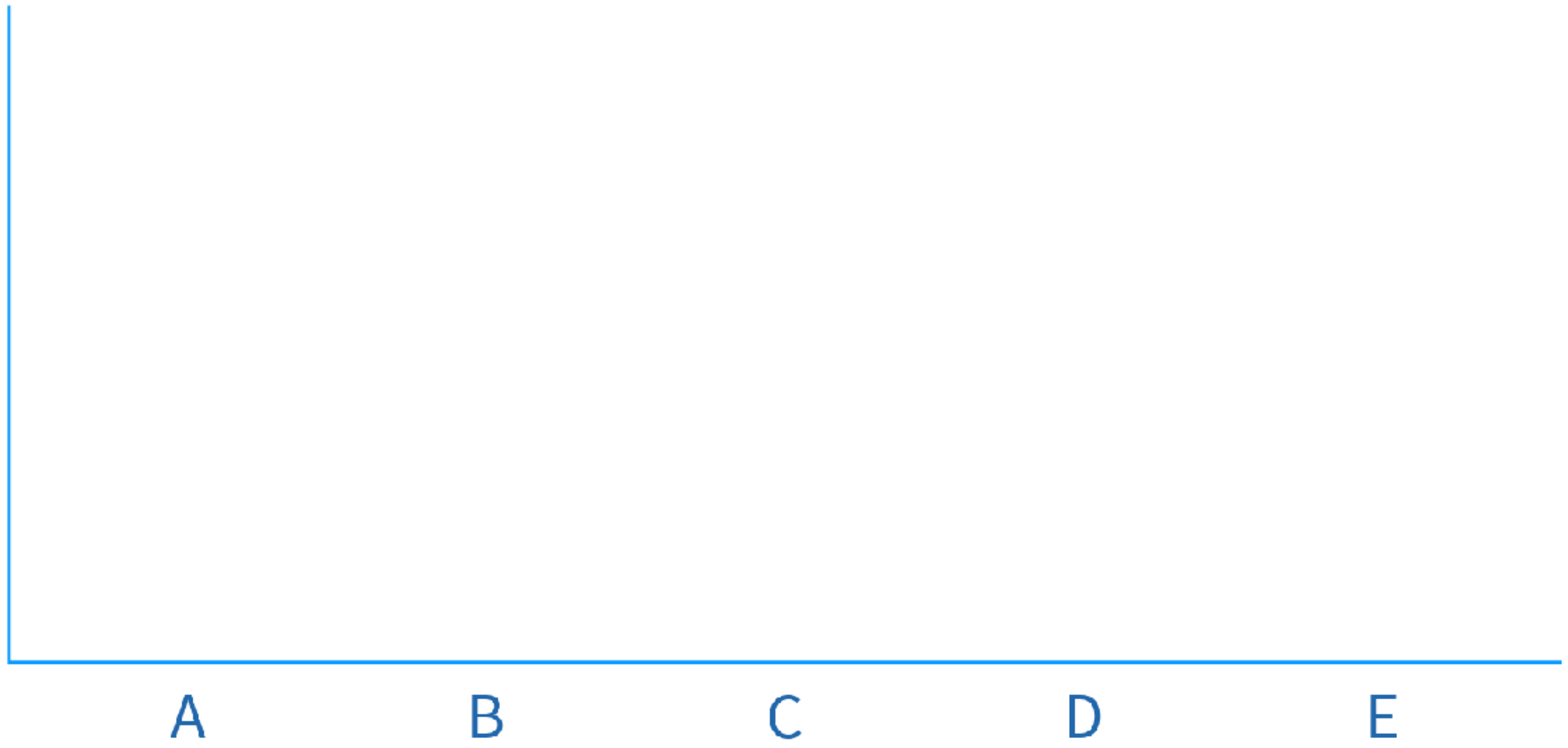
https://www.pollev.com/hungweitseng close in 1:00

In which pipeline stage can we have exceptions?

- How many of the following pipeline stages can we have exceptions?
  ① IF
  ② ID
  ③ EXE
  ④ MEM
  ⑤ WB
  A. 1
  B. 2
  C. 3
  D. 4
  E. 5



Exceptions-Group



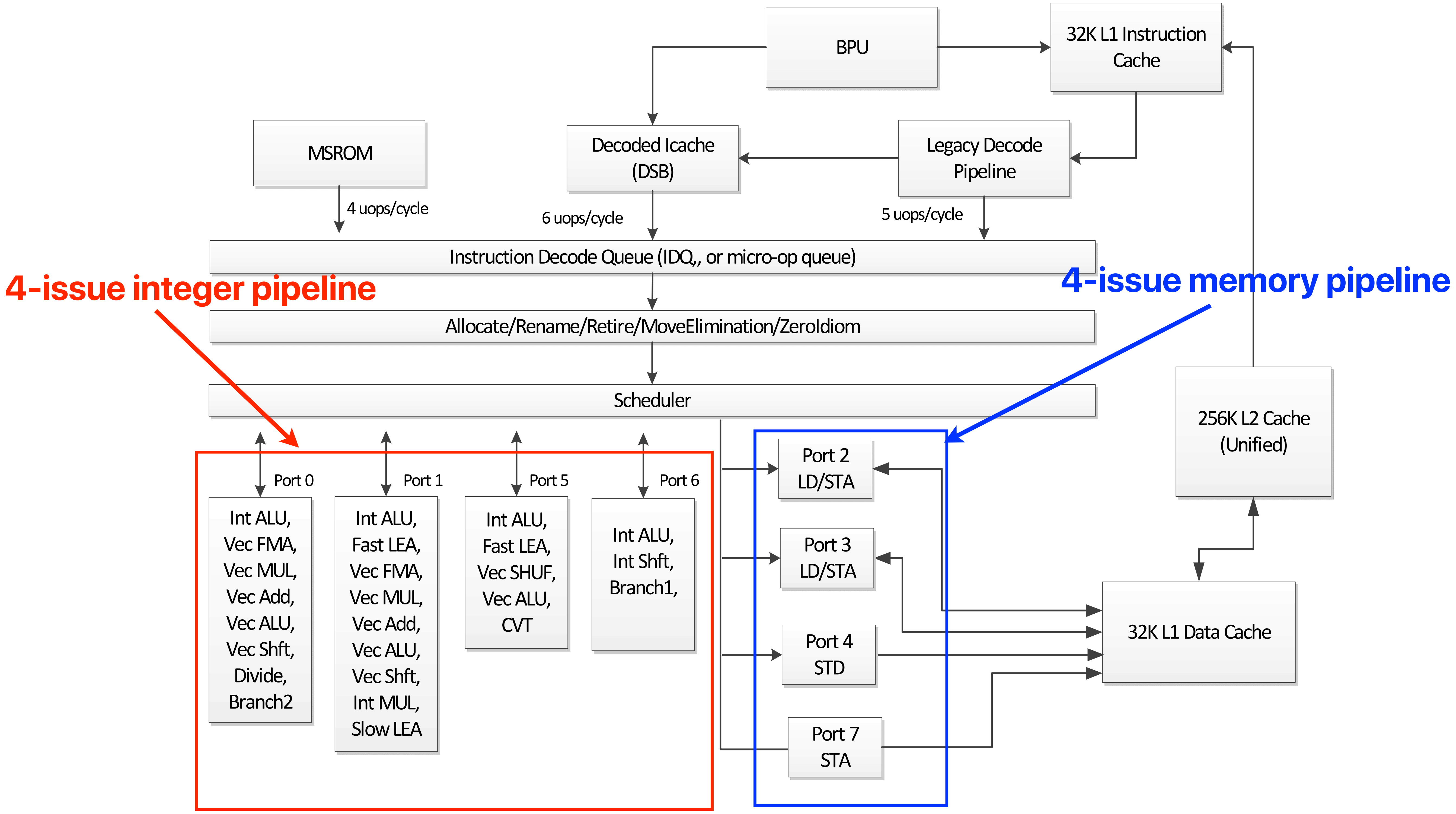
In which pipeline stage can we have exceptions?

- How many of the following pipeline stages can we have exceptions?
  ① IF — **page fault, illegal address**
  ② ID — **unknown instruction**
  ③ EXE — **divide by zero, overflow, underflow**
  ④ MEM — **page fault, illegal address**
  ⑤ WB
  A. 1
  B. 2
  C. 3
  [✓ D. 4]
  E. 5

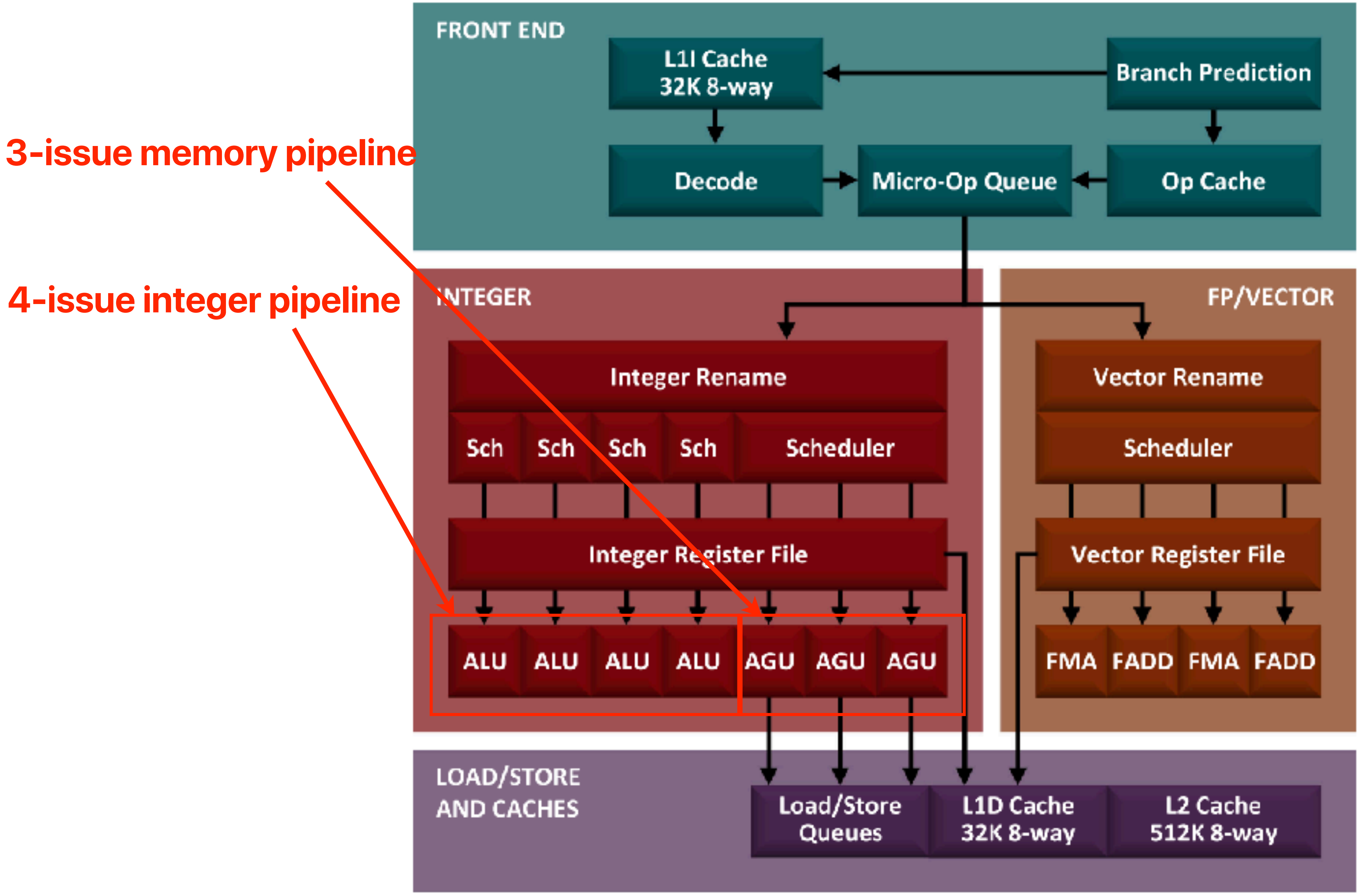
Reorder Buffer (ROB)

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)



Demo: ILP within a program

- perf is a tool that captures performance counters of your processors and can generate results like branch mis-prediction rate, cache miss rates and ILP.

Announcements

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

Computer
Science &
Engineering

つづく