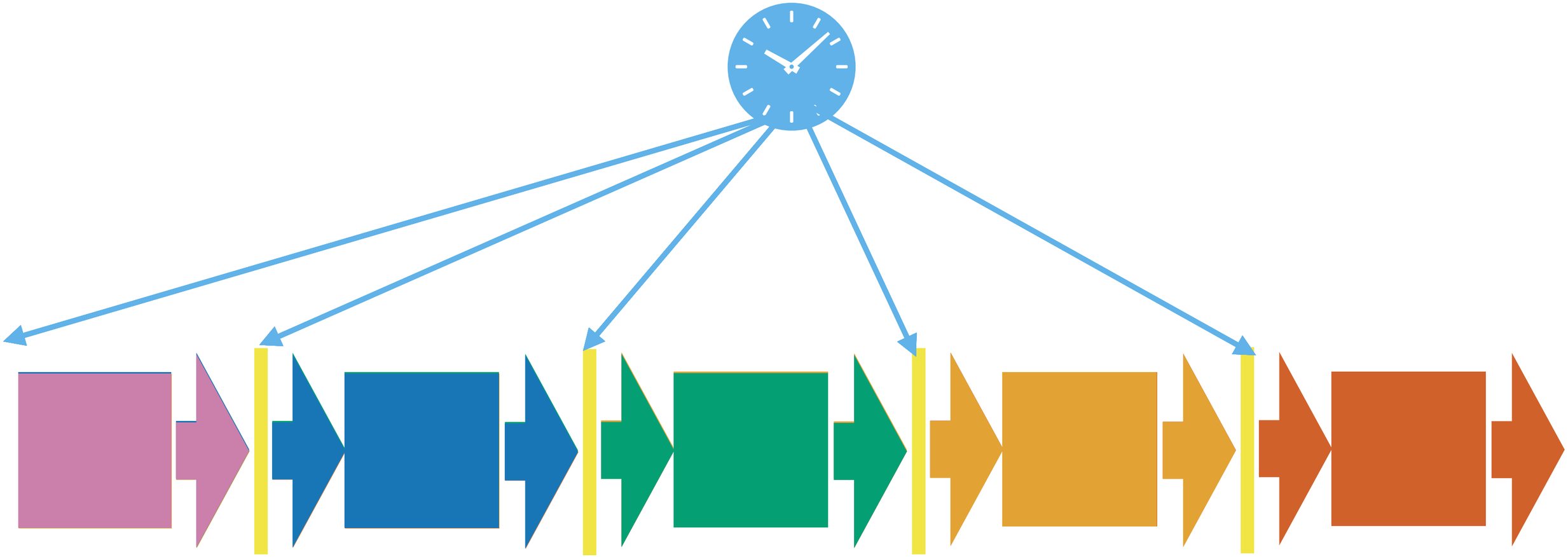
Data Hazards & Dynamic Instruction Scheduling: CPUtopia

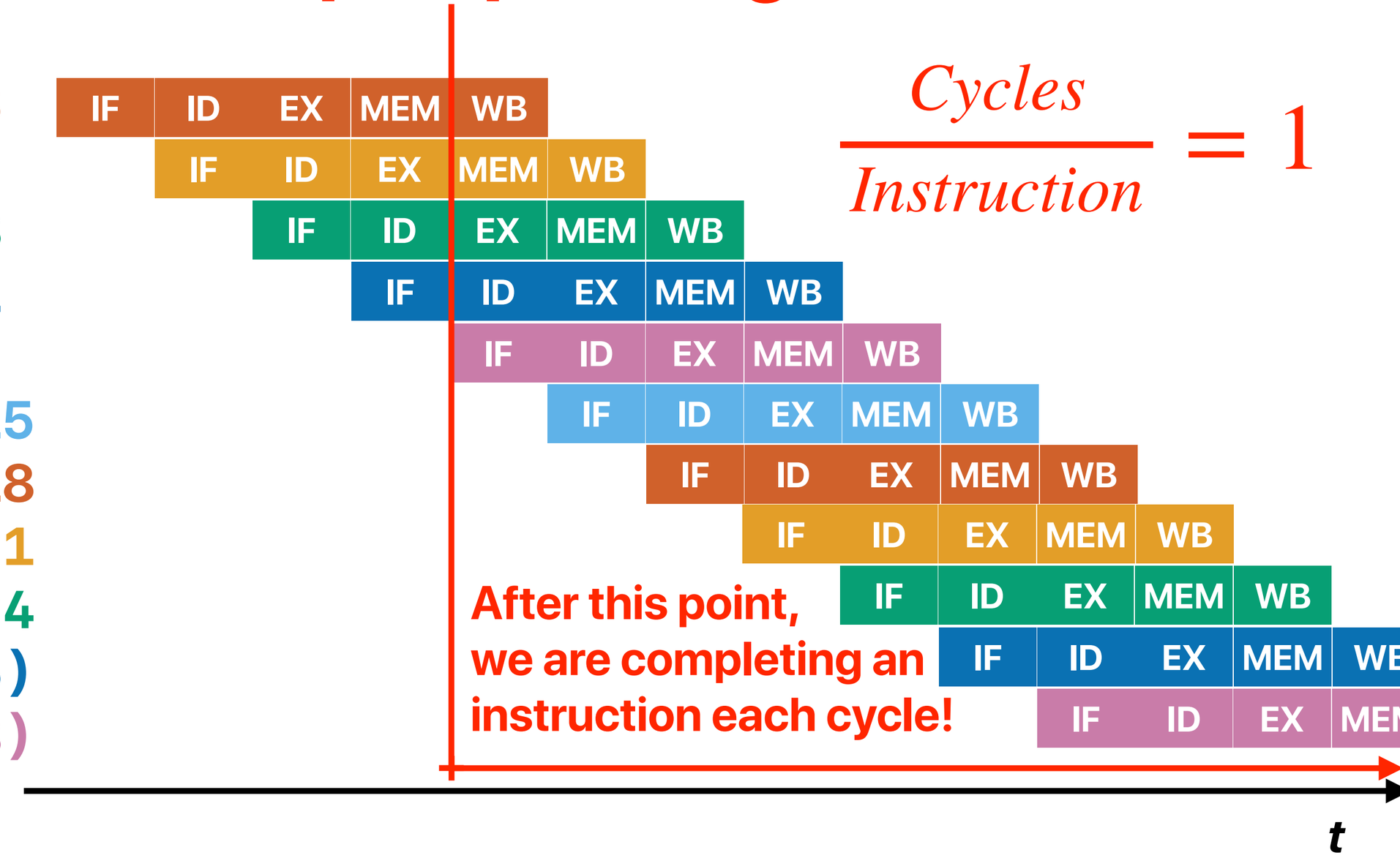
Hung-Wei Tseng

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)
```



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

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

Recap: 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 (but not necessary)** 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)
```

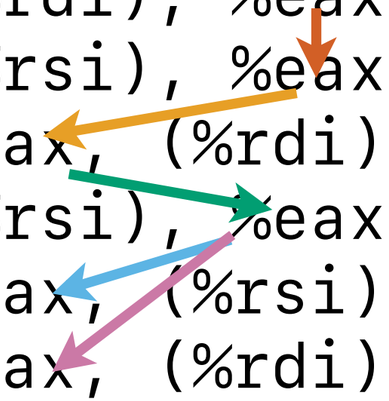
```
int temp = *a;
*a = *b;
*b = temp;
```

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

3:00

What will you do if you're waiting for someone else's response or outcome?

Ideas?

Outline

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

Solution 1: Let's try "stall" again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

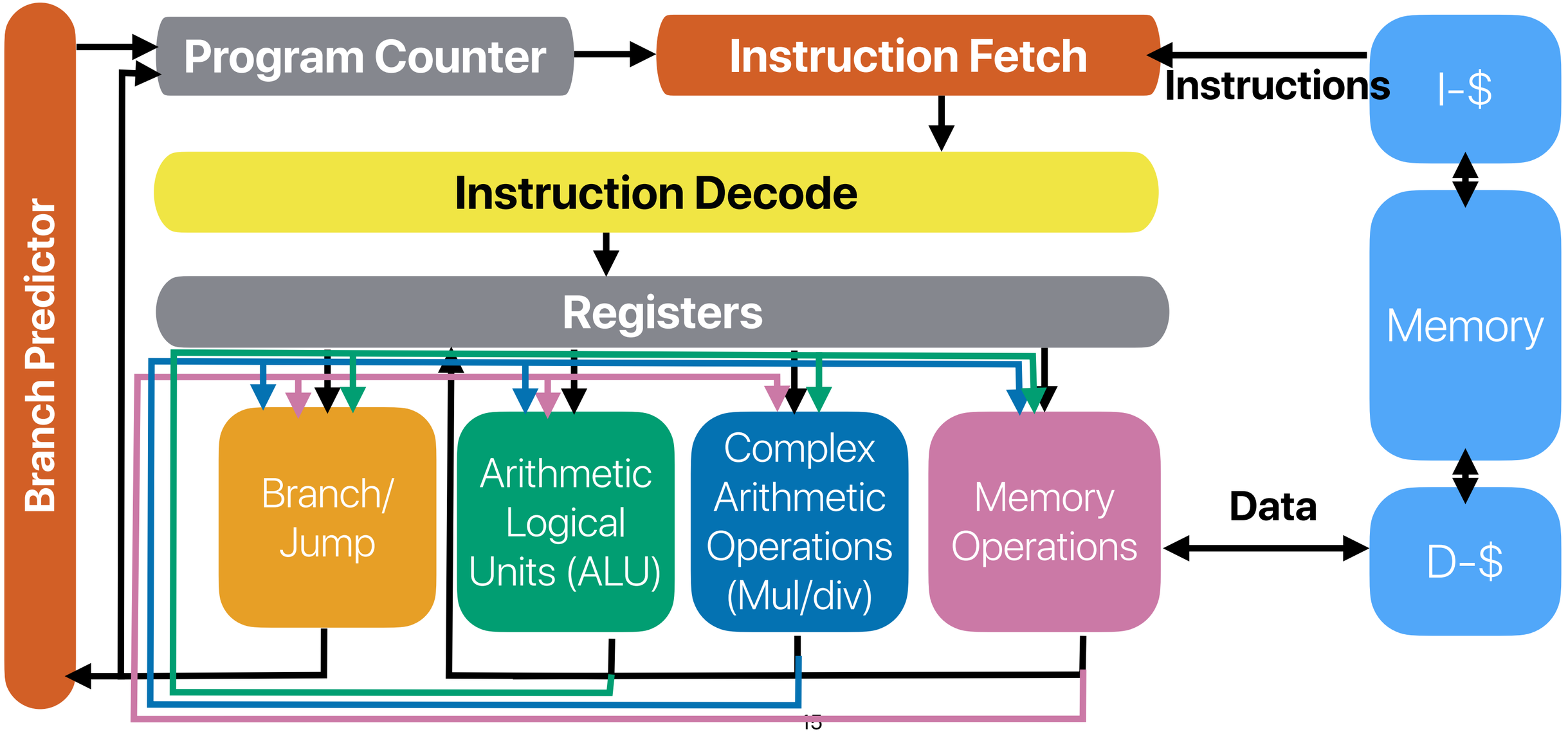
we have the value for %edx already!
Why another cycle?

	IF	ID	ALU/BR/M1	M2	M3	M4	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(4)	(3)				(2)	(1)
8	(4)	(3)					(2)
9		(4)	(3)				
10			(4)	(3)			
11				(4)	(3)		
12					(4)	(3)	
13						(4)	(3)
14							(4)

Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions

Data "forwarding"



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 4 cycles?

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

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

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 4 cycles?

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

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

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

	IF	ID	ALU/BR/M1	M2	M3	M4	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7		(4)				(2)	(1)
8			(3)				(2)
9			(4)	(3)			
10				(4)	(3)		
11					(4)	(3)	
12						(4)	(3)
13							(4)
14							

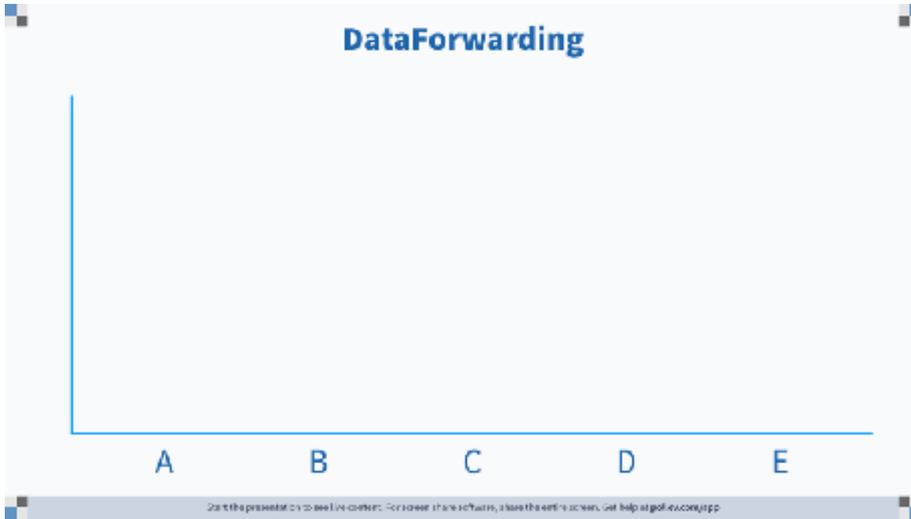
How many of them are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation takes 4 cycles and **xorl also takes 4 cycles**?

```
① movl    (%rdi), %eax
② xorl    (%rsi), %eax
③ movl    %eax, (%rdi)
④ xorl    (%rsi), %eax
⑤ movl    %eax, (%rsi)
⑥ xorl    %eax, (%rdi)
```

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

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



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 4 cycles and **xorl also takes 4 cycles**?

```
① movl    (%rdi), %eax
② xorl    (%rsi), %eax
③ movl    %eax, (%rdi)
④ xorl    (%rsi), %eax
⑤ movl    %eax, (%rsi)
⑥ xorl    %eax, (%rdi)
```

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

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(3)	(2)		(1)			
5	(3)	(2)			(1)		
6	(3)	(2)				(1)	
7	(4)	(3)	(2)				(1)
8	(4)	(3)		(2)			
9	(4)	(3)			(2)		
10	(4)	(3)				(2)	
11	(5)	(4)	(3)				(2)
12	(6)	(5)	(4)	(3)			
13	(6)	(5)		(4)	(3)		
14	(6)	(5)			(4)	(3)	
15	(6)	(5)				(4)	(3)
16		(6)	(5)				(4)
17			(6)	(5)			
18				(6)	(5)		
19					(6)	(5)	
20						(6)	(5)
21							(6)
22							

Single pipeline

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
①       movl    (%rdi), %ecx
②       addl    %ecx, %eax
③       addq    $4, %rdi
④       cmpq    %rdx, %rdi
⑤       jne     .L3
⑥       ret
```

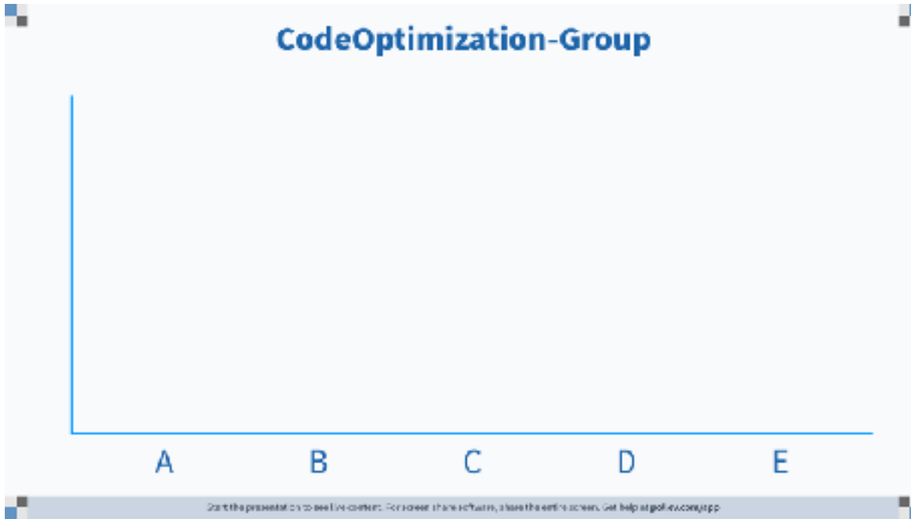
	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(3)	(2)		(1)			
5	(3)	(2)			(1)		
6	(3)	(2)				(1)	
7	(4)	(3)	(2)				(1)
8	(5)	(4)	(3)	(2)			
9		(5)	(4)	(3)	(2)		
10			(5)	(4)	(3)	(2)	
11				(5)	(4)	(3)	(2)
12					(5)	(4)	(3)
13						(5)	(4)

The effect of code optimization

- By reordering which pair of the following instruction stream can we not affect 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



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`

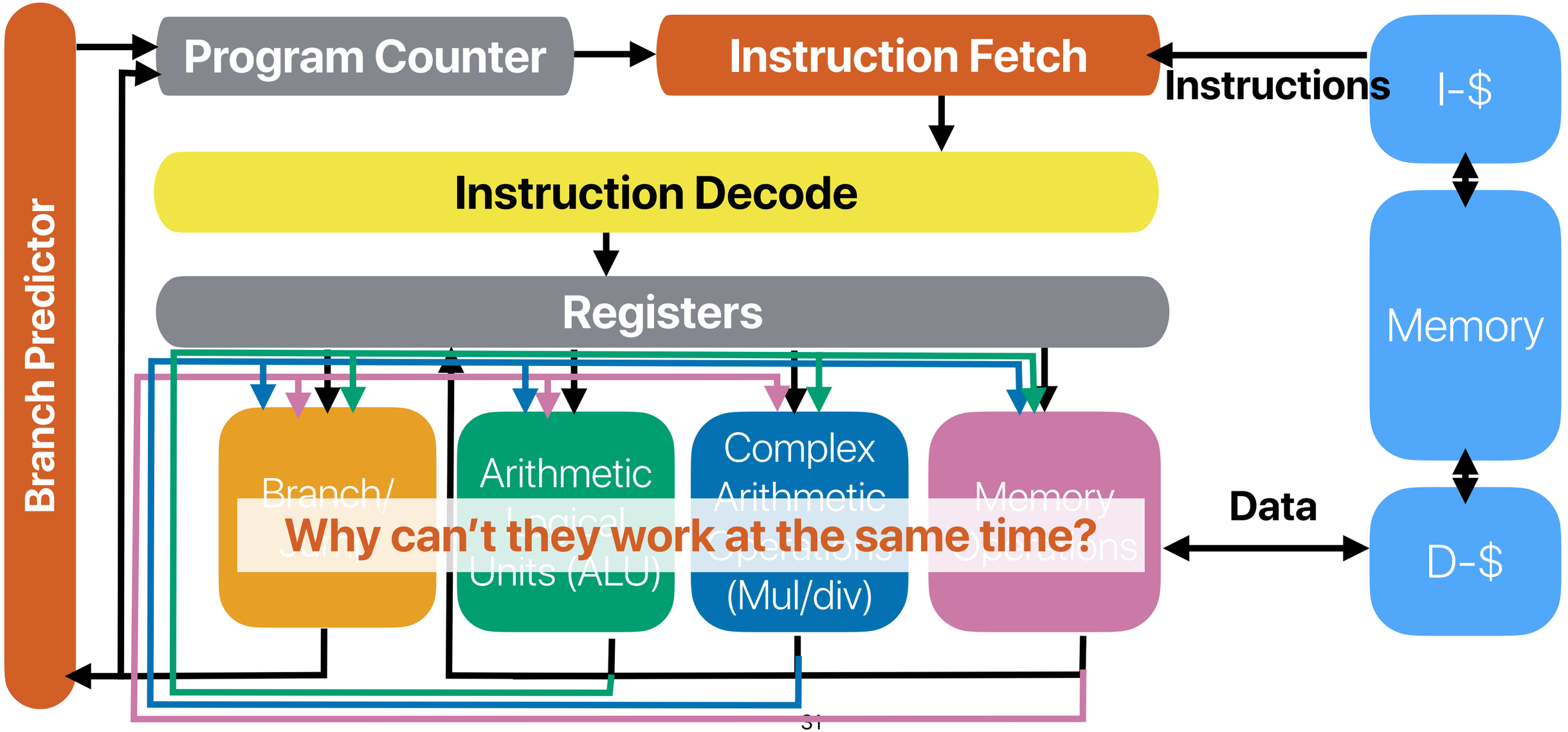
A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered

Compiler optimization

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
①       movl    (%rdi), %ecx
②       addq    $4, %rdi
③       addl    %ecx, %eax
④       cmpq    %rdx, %rdi
⑤       jne     .L3
⑥       ret
```

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(3)	(2)	(2)	(1)			
5	(3)	(2)		(2)	(1)		
6	(4)	(2)			(2)	(1)	
7	(5)	(4)	(3)			(2)	(1)
8		(5)	(4)	(3)			(2)
9			(5)	(4)	(3)		
10				(5)	(4)	(3)	
11					(5)	(4)	(3)
12						(5)	(4)
13							(5)

Data "forwarding"



Compiler optimization

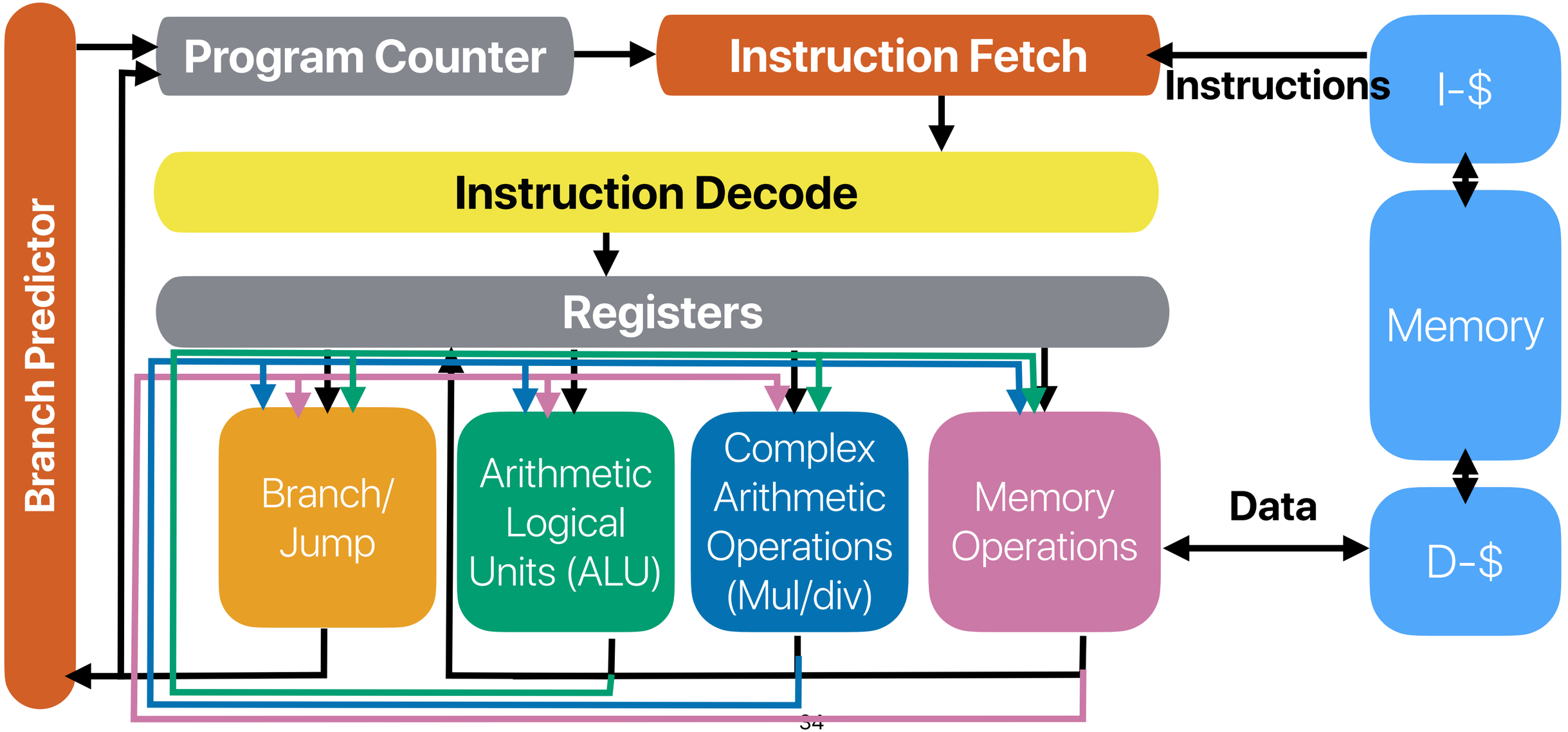
```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
①       movl    (%rdi), %ecx
②       addq    $4, %rdi
③       addl    %ecx, %eax
④       cmpq    %rdx, %rdi
⑤       jne     .L3
⑥       ret
```

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

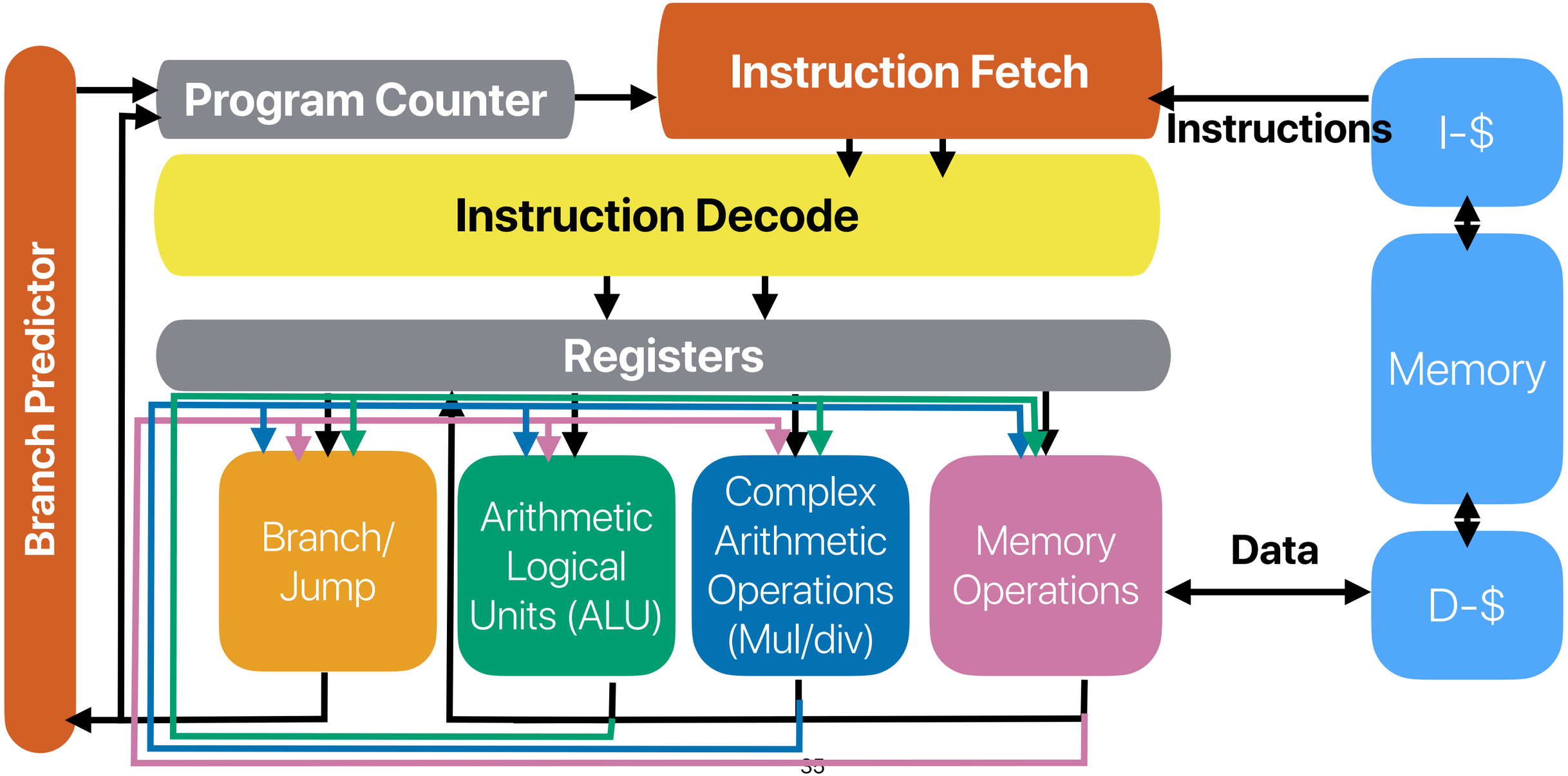
	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(3)	(2)	(2)	(1)			
5	(3)	(2)		(2)	(1)		
6	(4)	(2)			(2)	(1)	
7	(5)	(4)	(3)			(2)	(1)
8		(5)	(4)	(3)			(2)
9			(5)	(4)	(3)		
10				(5)	(4)	(3)	
11					(5)	(4)	(3)
12						(5)	(4)
13							(5)

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

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

If we loop many times (assume perfect predictor)

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

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

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

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

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

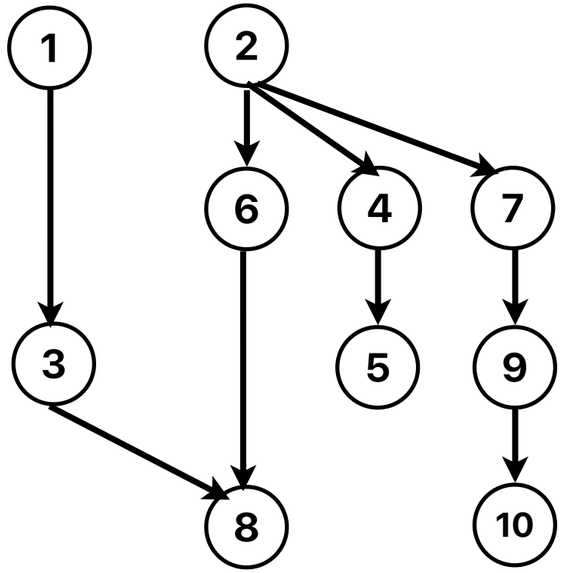
What do you need to execution an instruction?

- 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
```



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

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)

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?

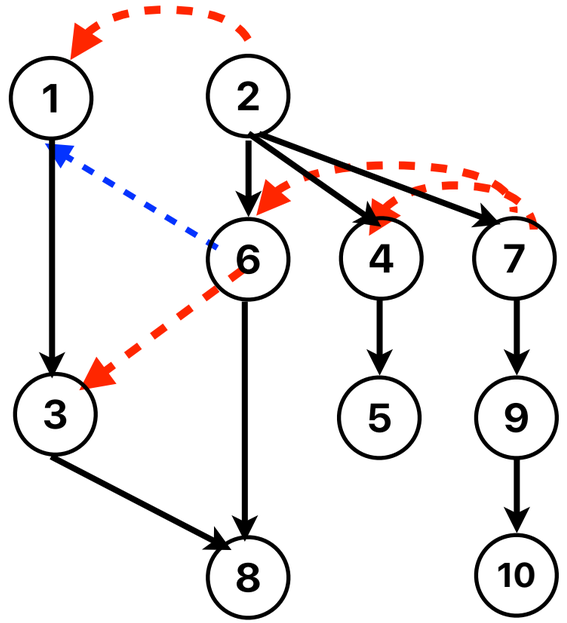
Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

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

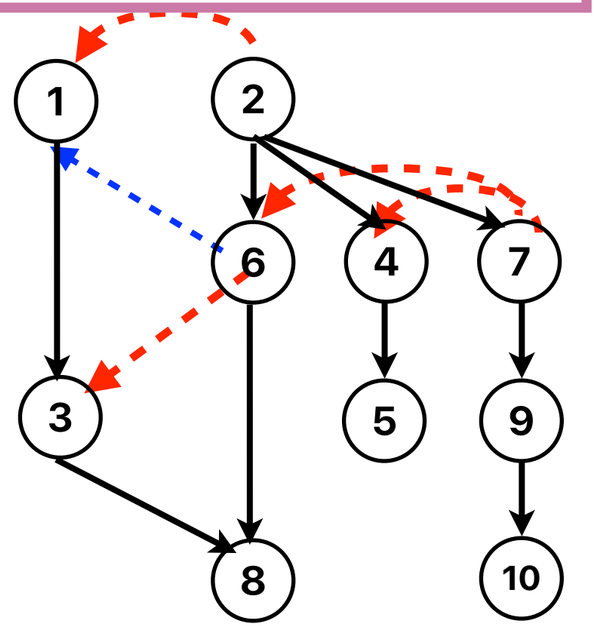
```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



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
```



Announcements

- Assignment #3 due **this Thursday**
- Reading Quiz due next Tuesday

Computer
Science &
Engineering

203

つづく