Modern Processor Design (I): in the pipeline

Hung-Wei Tseng

3:00

What "tricky C/C++ programming questions" have you heard before?

Tricky C/C++ programming questions?

- Give a fastest way to multiply any number by 9
- How to measure the size of any variable without "sizeof" operator?.
- How to measure the size of any variable without using "sizeof" operator?
- Write code snippets to swap two variables in five different ways
- How to swap between first & 2nd byte of an integer in one line statement?
- What is the efficient way to divide a no. by 4?
- Suggest an efficient method to count the no. of 1's in a 32 bit no. Remember without using loop & testing each bit.
- Test whether a no. is power of 2 or not.
- How to check endianness of the computer.
- Write a C-program which does the addition of two integers without using '+' operator.
- Write a C-program to find the smallest of three integers without using any of the comparision operators.
- Find the maximum & minimum of two numbers in a single line without using any condition & loop.
- What "condition" expression can be used so that the following code snippet will print Hello world.
- How to print number from 1 to 100 without using conditional operators.
- WAP to print 100 times "Hello" without using loop & goto statement.
- Write the equivalent expression for x%8.

https://www.emblogic.com/blog/12/tricky-c-interview-questions/

Which swap is faster?

A

```
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```
void xorswap(int* a, int* b) {
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

- Both version A and B swaps content pointed by a and b correctly. Which version of code would have better performance?
  A. Version A
  B. Version B
  C. They are about the same (sometimes A is faster, some

Which swap is faster?

A

```
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```
void xorswap(int* a, int* b) {
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

- Both version A and B swaps content pointed by a and b correctly. Which version of code would have better performance?
  A. Version A
  B. Version B
  C. They are about the same (sometimes A is faster, sometimes B is)

Recap: Why adding a sort makes it faster

- Why the sorting the array speed up the code despite the increased instruction count?

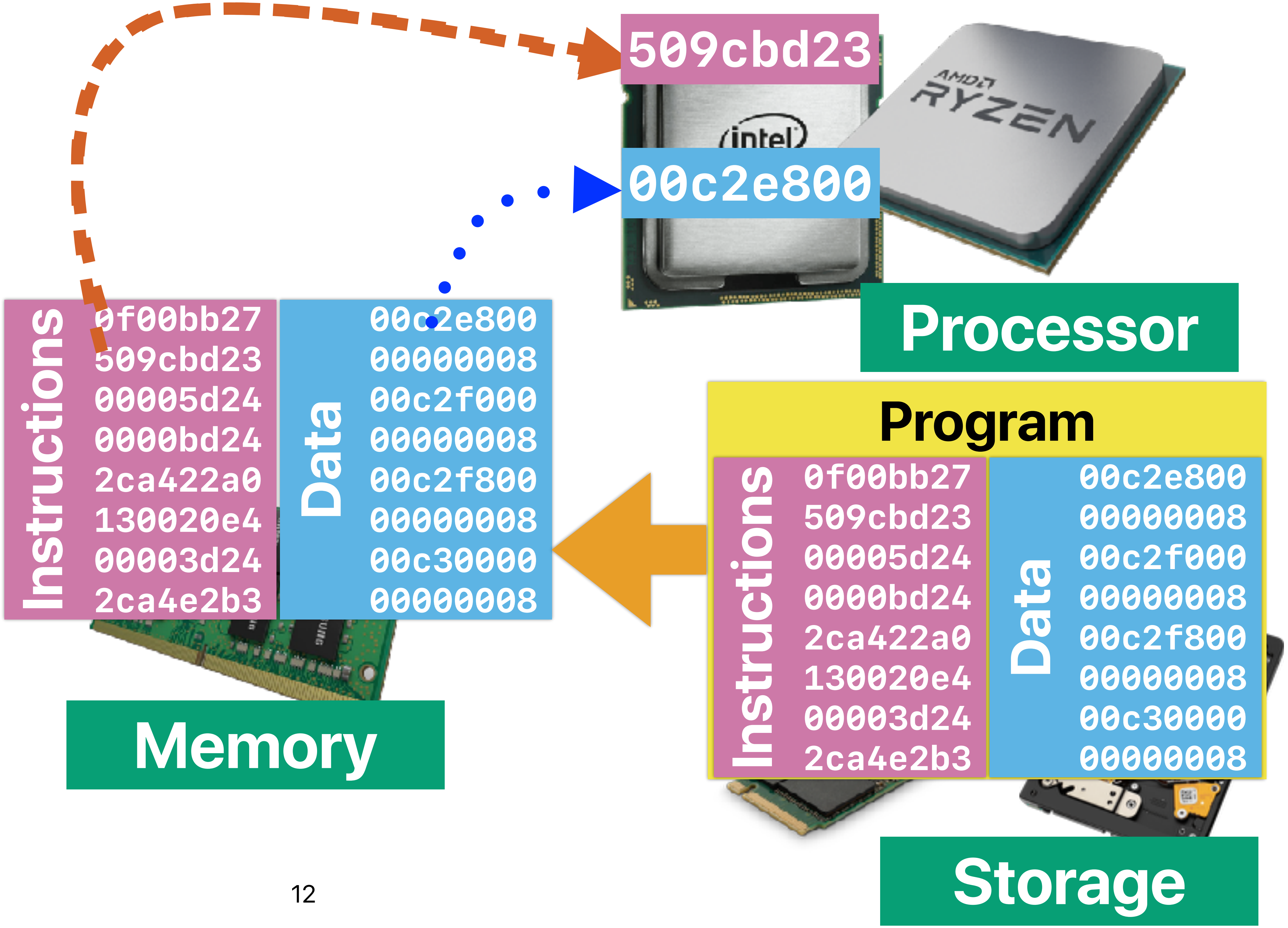
```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

Outline

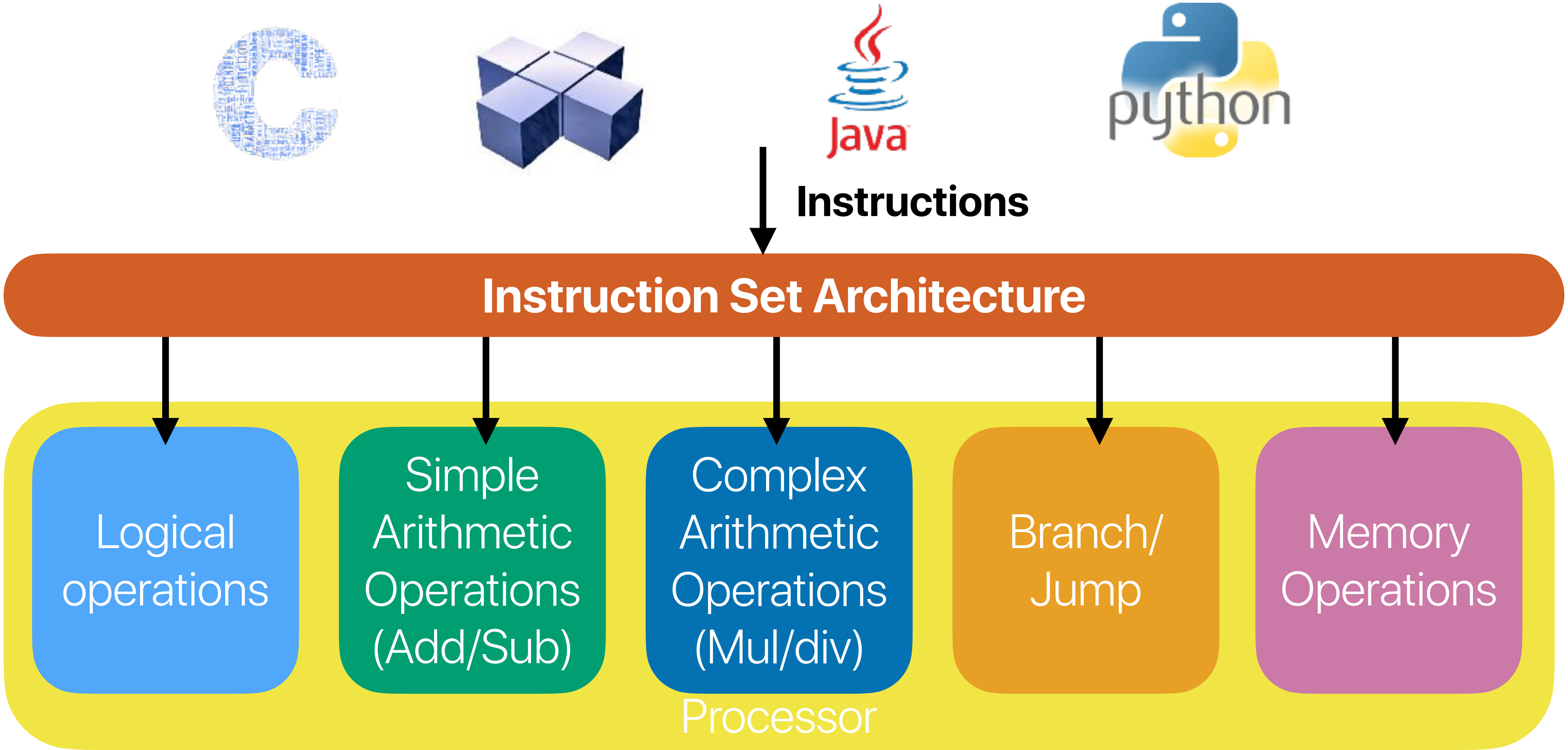
- Pipelined Processor
- Pipeline Hazards
  - Structural Hazards
  - Control Hazards
  - Data Hazards
- Dynamic Branch Predictions

Basic Processor Design

von Neumman Architecture



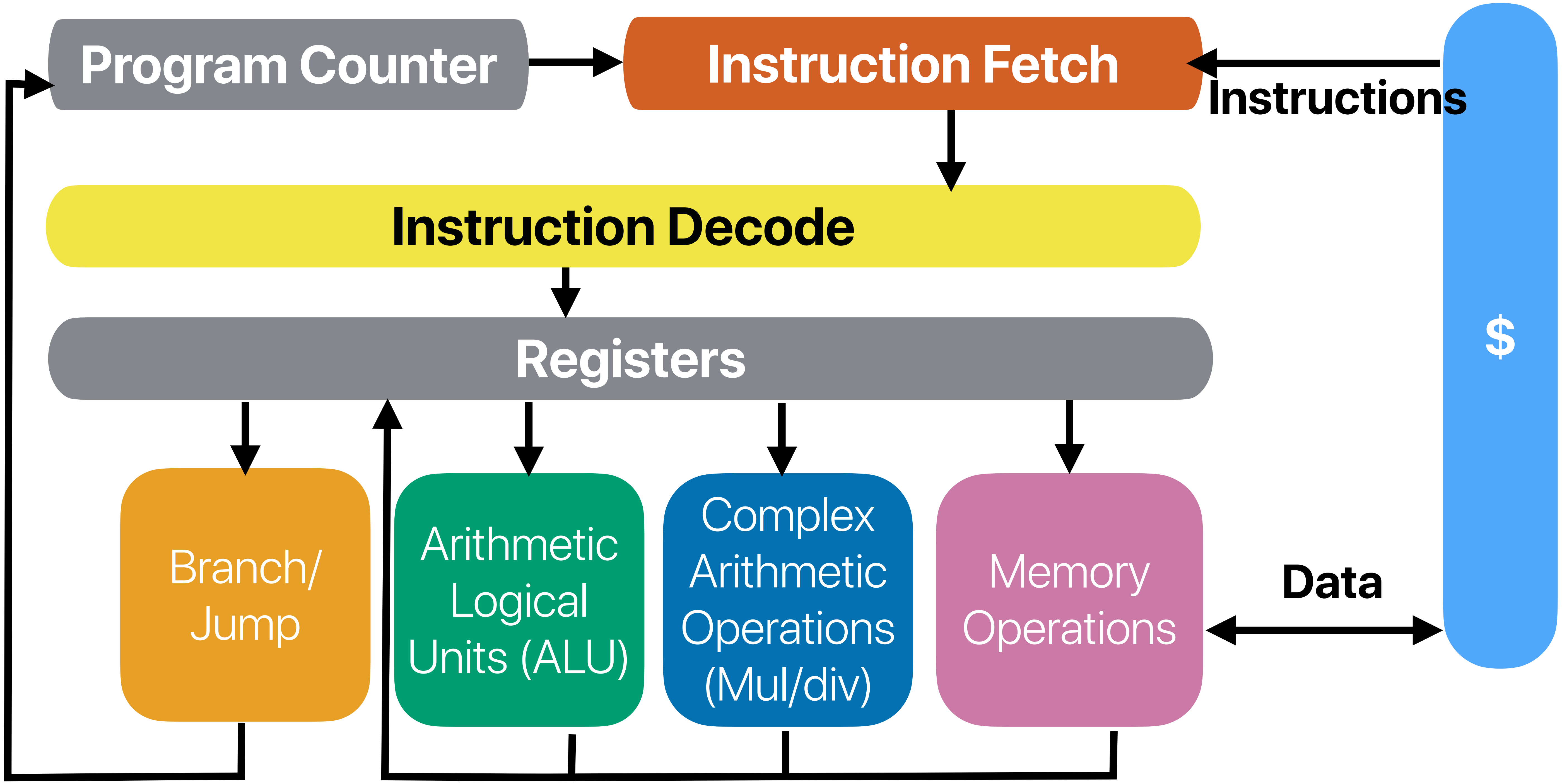
Recap: Microprocessor — a collection of functional units



The "life" of an instruction

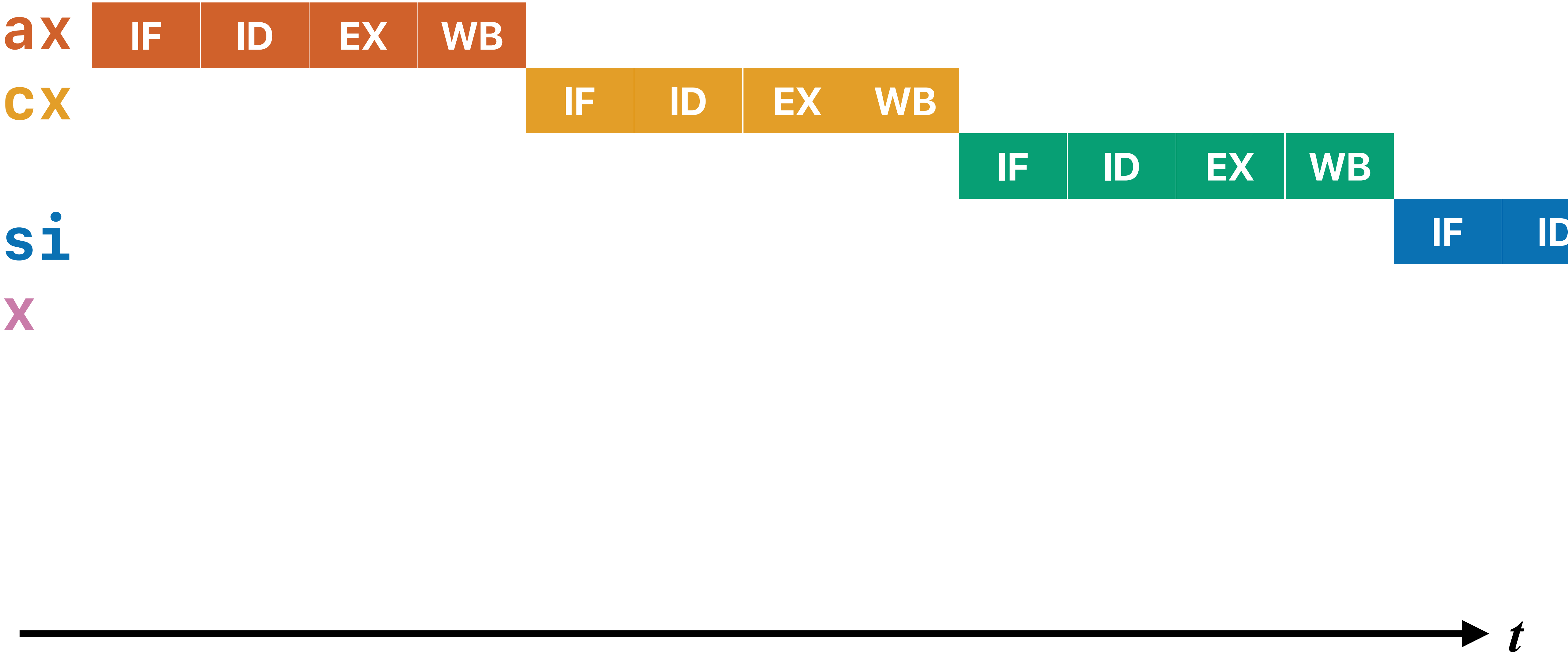
- Instruction Fetch (**IF**) — fetch the instruction from memory
- Instruction Decode (**ID**)
  - Decode the instruction for the desired operation and operands
  - Reading source register values
- Execution (**EX**)
  - ALU instructions: Perform ALU operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (**MEM**) — Read/write memory
- Write Back (**WB**) — Present ALU result/read value in the target register
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4

Functional Units of a Microprocessor



Simple implementation w/o branch

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl  %esi, %esi
movl   $10, %edx
```

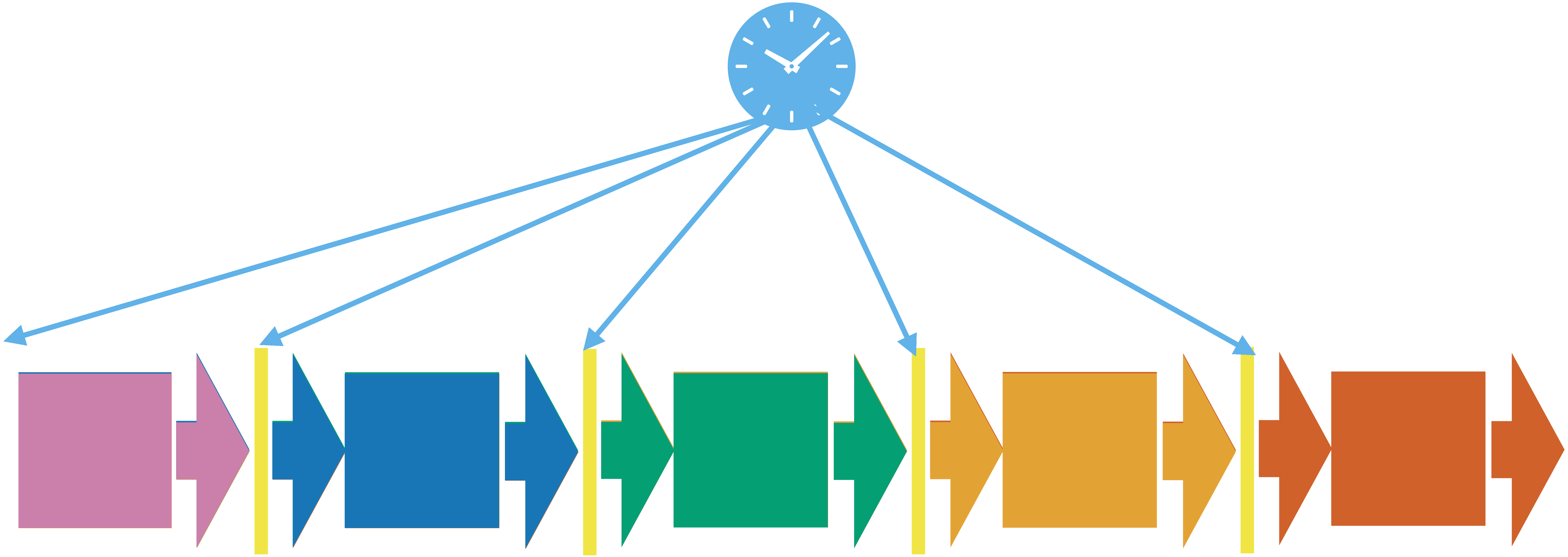


Pipelining

Pipelining

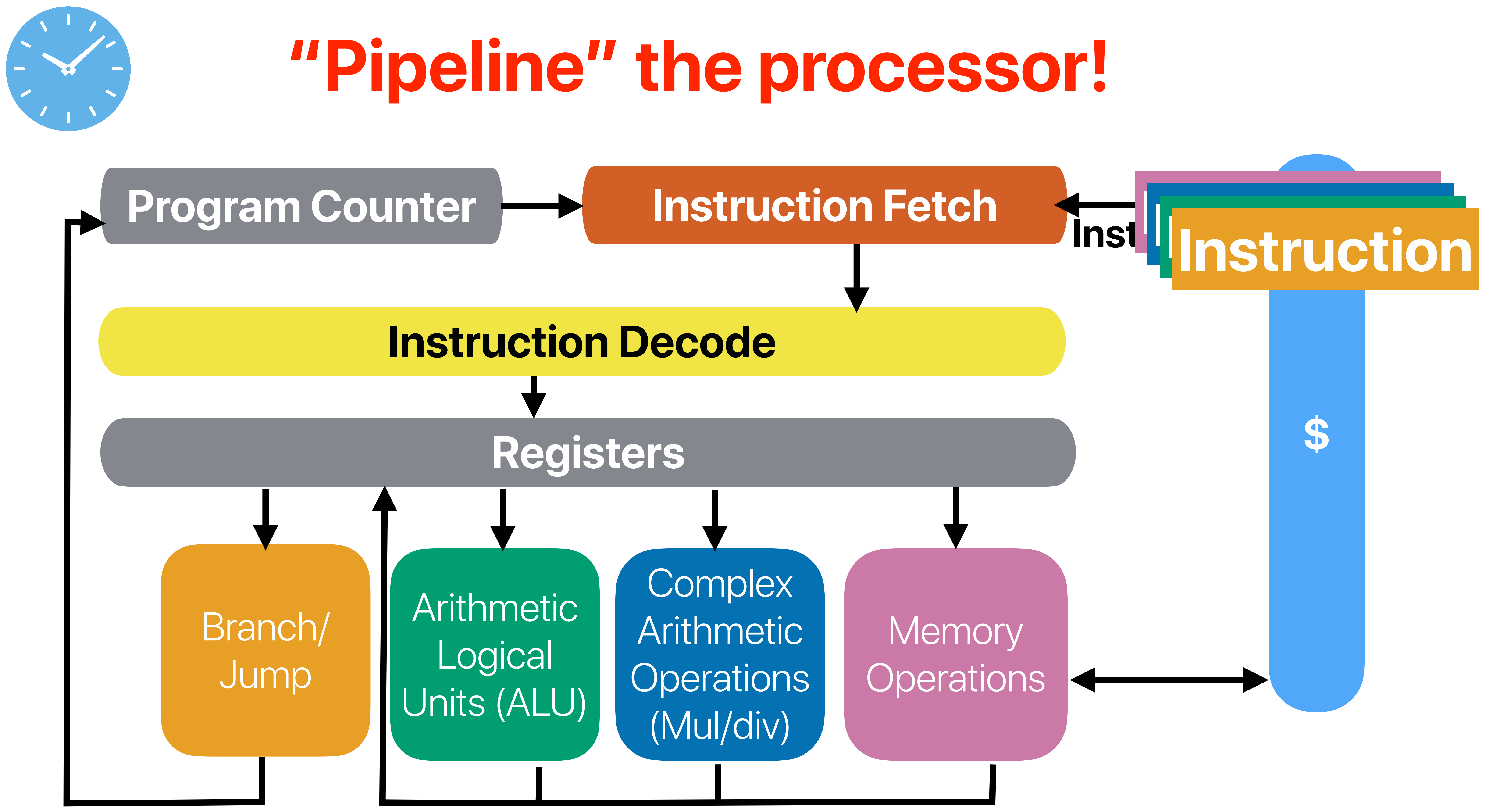
- Different parts of the processor works on different instructions simultaneously
- A processor is now working on multiple instructions from the same program (though on different stages) simultaneously.
  - ILP: **Instruction-level parallelism**
- A **clock** signal controls and synchronize the beginning and the end of each part of the work
- A **pipeline register** between different parts of the processor to keep intermediate results necessary for the upcoming work

Pipelining



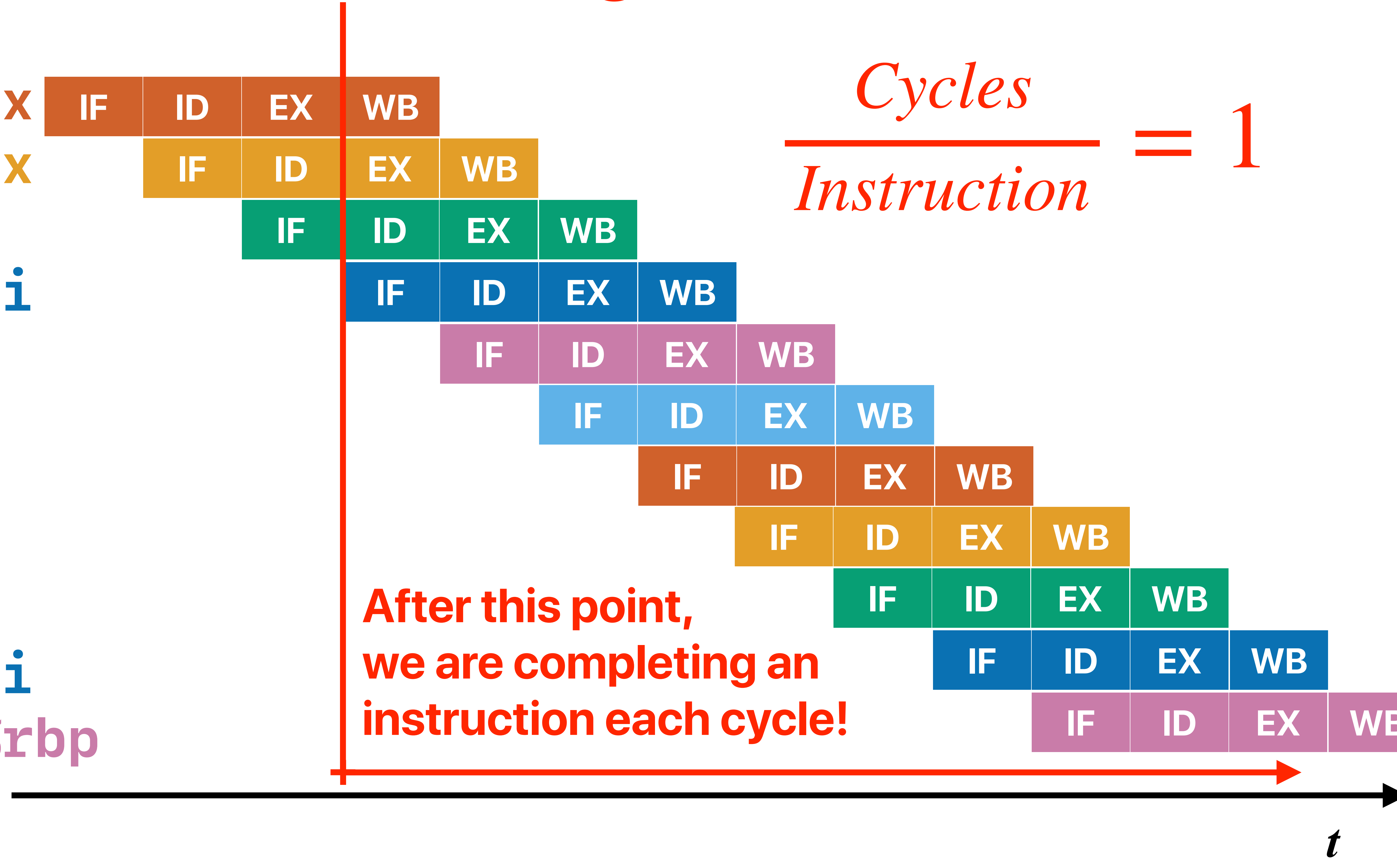
Pipelining

"Pipeline" the processor!



Pipelining

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl  %esi, %esi
movl   $10, %edx
pushq  %r12
pushq  %rbp
pushq  %rbx
subq   $8, %rsp
addl   %rsi, %rdi
movslq  %eax, %rbp
```



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

How well can we pipeline?

- With a pipelined design, the processor is supposed to deliver the outcome of an instruction each cycle. For the following code snippet, how many pairs of instructions are preventing the pipeline from generating results in back-to-back cycles?

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

```
for(i = 0; i < count; i++) {
    s += a[i];
}
return s;
```

A. 1
B. 2
C. 3
D. 4
E. 5

How well can we pipeline?

- With a pipelined design, the processor is supposed to deliver the outcome of an instruction each cycle. For the following code snippet, how many pairs of instructions are preventing the pipeline from generating results in back-to-back cycles?

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

```
for(i = 0; i < count; i++) {
    s += a[i];
}
return s;
```

A. 1
B. 2
C. 3
D. 4
E. 5

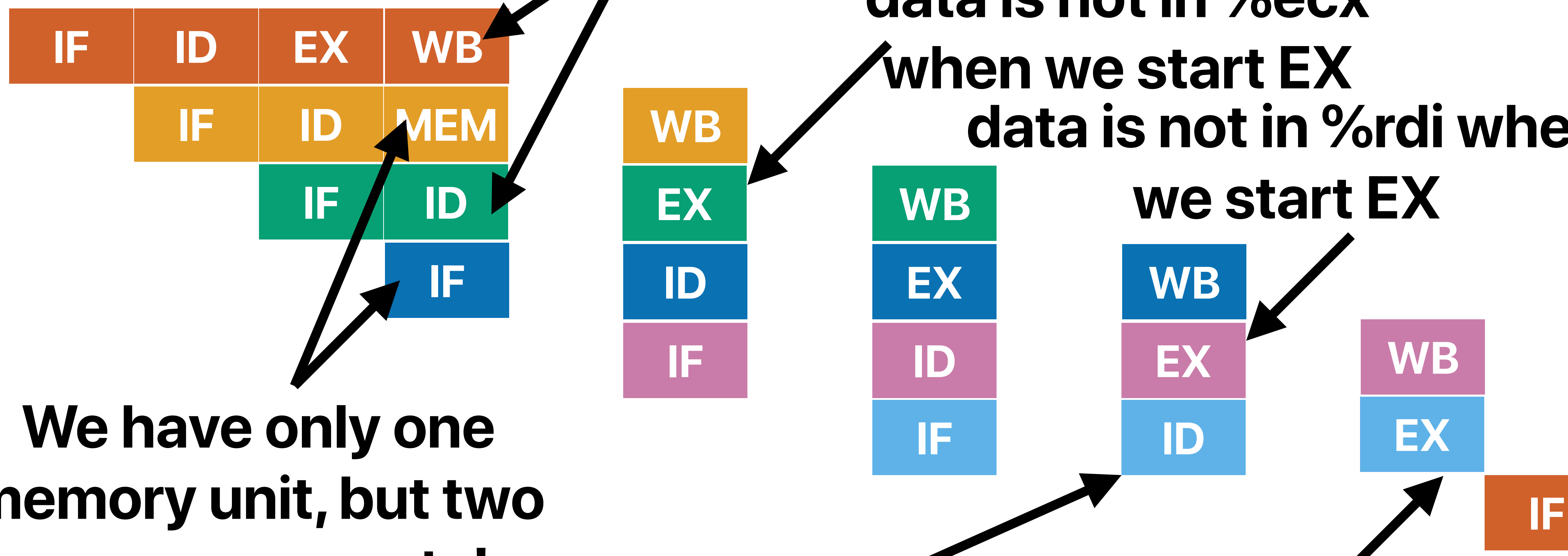
Pipelining

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

Both (1) and (3) are attempting to access %eax

data is not in %ecx when we start EX

data is not in %rdi whe we start EX



We have only one memory unit, but two access requests!

We cannot know if we should fetch (7) or (2) before the EX is done

(6) may not have the outcome from (5)

How well can we pipeline?

- With a pipelined design, the processor is supposed to deliver the outcome of an instruction each cycle. For the following code snippet, how many pairs of instructions are preventing the pipeline from generating results in back-to-back cycles?

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

```
for(i = 0; i < count; i++) {
    s += a[i];
}
return s;
```

A. 1
B. 2
C. 3
D. 4
E. 5

Pipeline hazards

Three types of 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

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

Pipelining

① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
⑤ cmpq %rdx, %rdi
⑥ jne .L3
⑦ ret

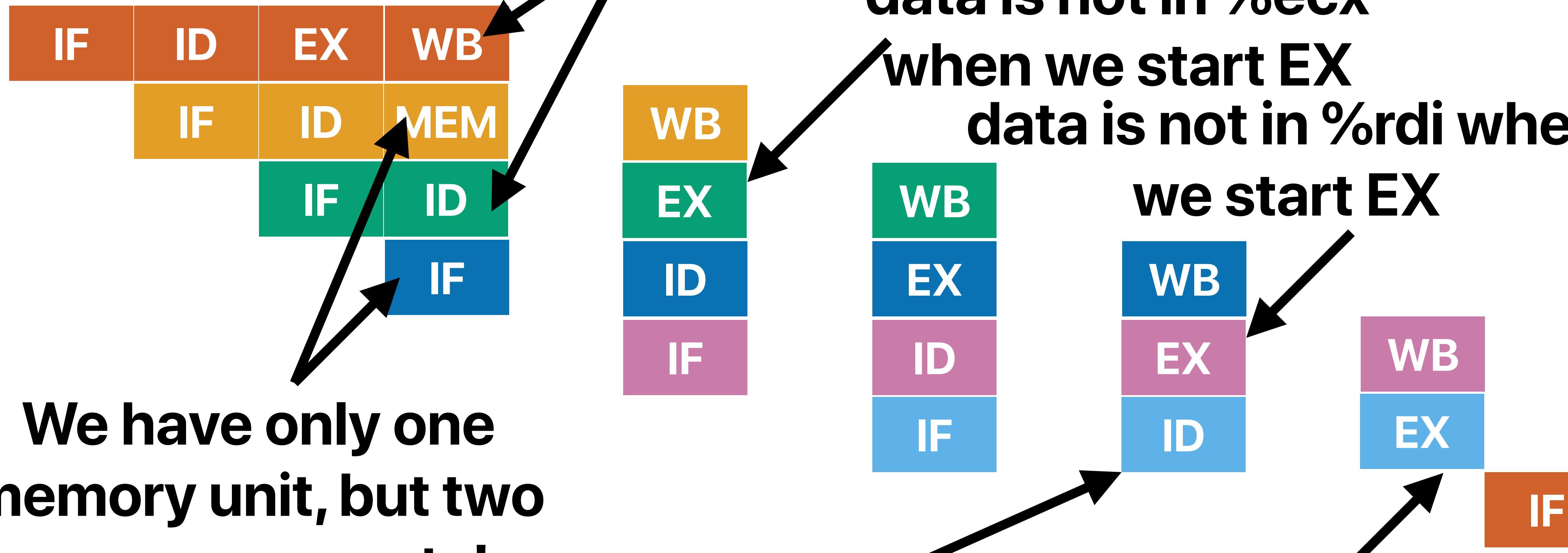
- How many of the "hazards" are data hazards?

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

Both (1) and (3) are attempting to access %eax

data is not in %ecx when we start EX

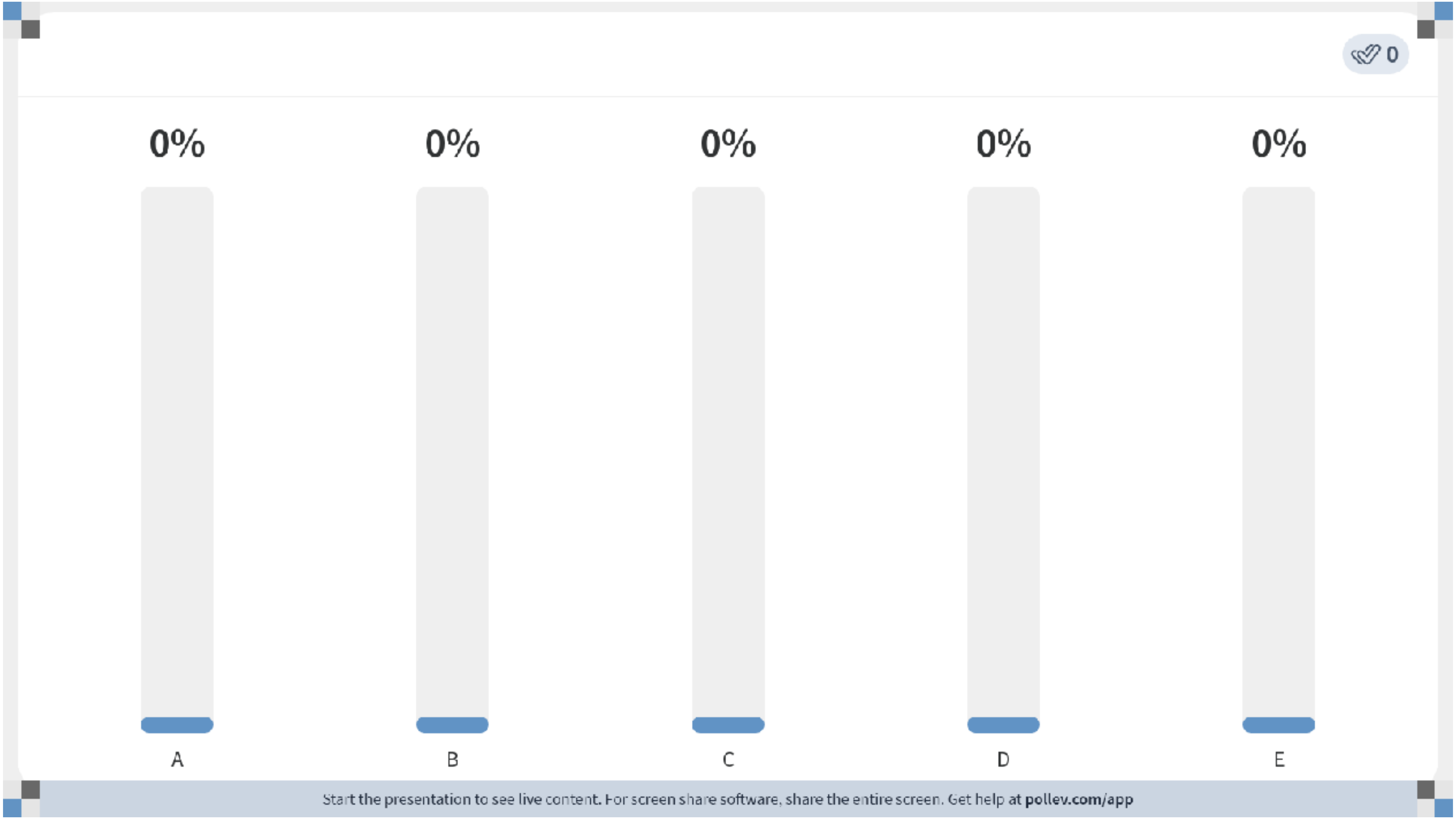
data is not in %rdi when we start EX



We have only one memory unit, but two access requests!

We cannot know if we should fetch (7) or (2) before the EX is done

(6) may not have the outcome from (5)


0
0%
0%
0%
0%
0%
A
B
C
D
E
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

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

Pipelining

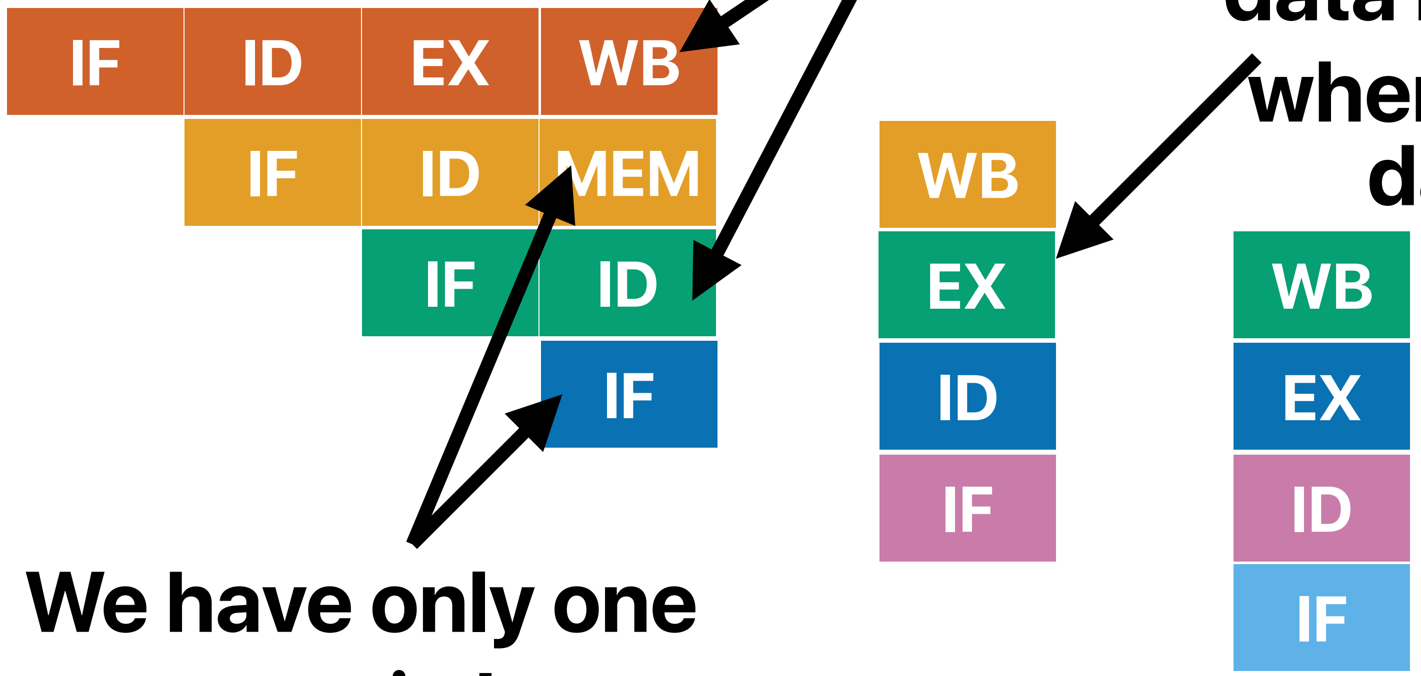
```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
⑤ cmpq %rdx, %rdi
⑥ jne  .L3
⑦ ret
```

- How many of the “hazards” are data hazards?
   A. 0
   B. 1
   C. 2
   D. 3
   E. 4

Both (1) and (3) are attempting to access %eax

data is not in %ecx when we start EX

data is not in %rdi when we start EX



We have only one memory unit, but two access requests!

We cannot know if we should fetch (7) or (2) before the EX is done

(6) may not have the outcome from (5)

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

Why is A is faster?

A

```c
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```c
void xorswap(int* a, int* b) {
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

- What's the main cause of the performance different in A and B on modern processors?
    A. Control hazards
    B. Data hazards
    C. Structural hazards

Why is A is faster?

A

```c
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```c
void xorswap(int* a, int* b) {
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

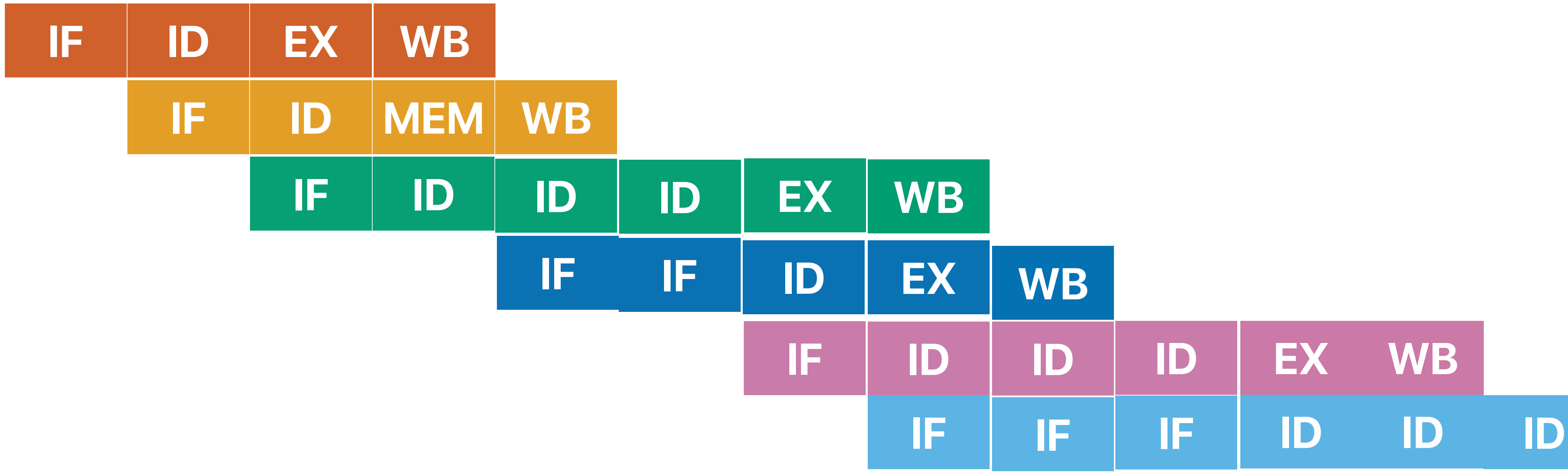
- What's the main cause of the performance different in A and B on modern processors?
    A. Control hazards
    B. Data hazards
    C. Structural hazards

Stall — the universal solution to pipeline hazards

Stall whenever we have a hazard

- Stall: the hardware allows the earlier instruction to proceed, all later instructions stay at the same stage

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



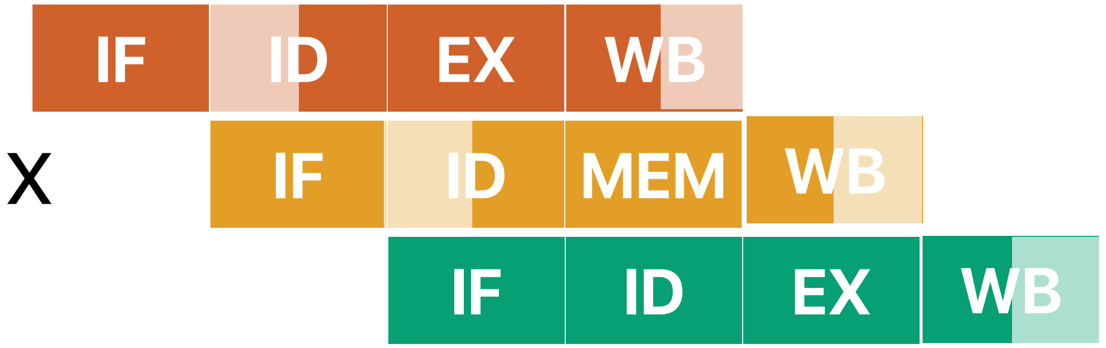
Slow! — 5 additional cycles

Structural Hazards

Dealing with the conflicts between ID/WB

- The same register cannot be read/written at the same cycle
- Better solution: write early, read late
  - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  - This leaves enough time for outputs to settle for reads
  - The revised register file is the default one from now!

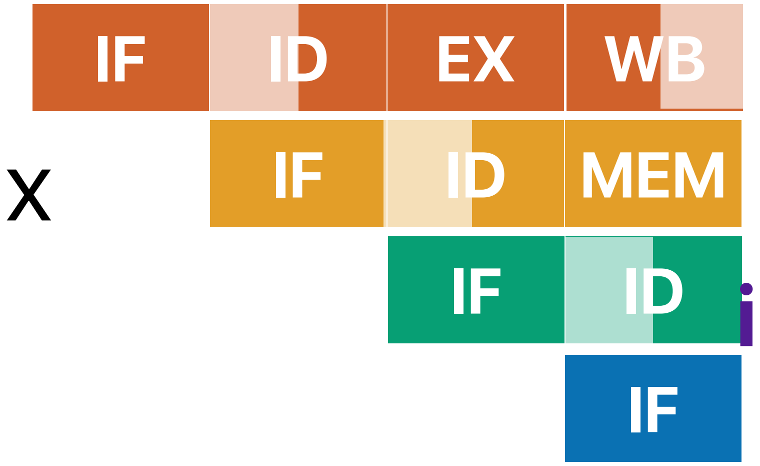
```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
```

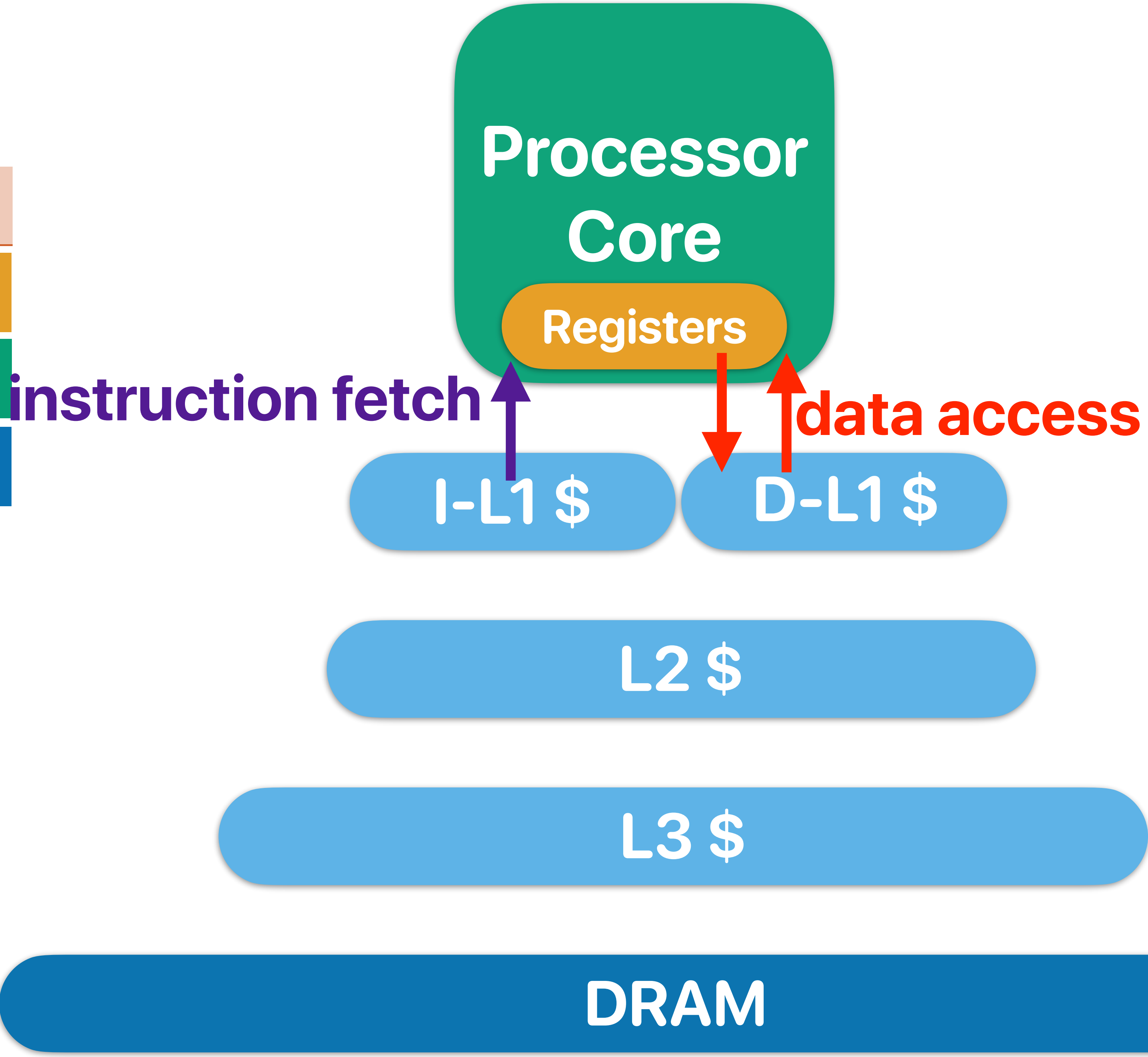


How to with the conflicts between MEM and IF?

- The memory unit can only accept/perform one request each cycle

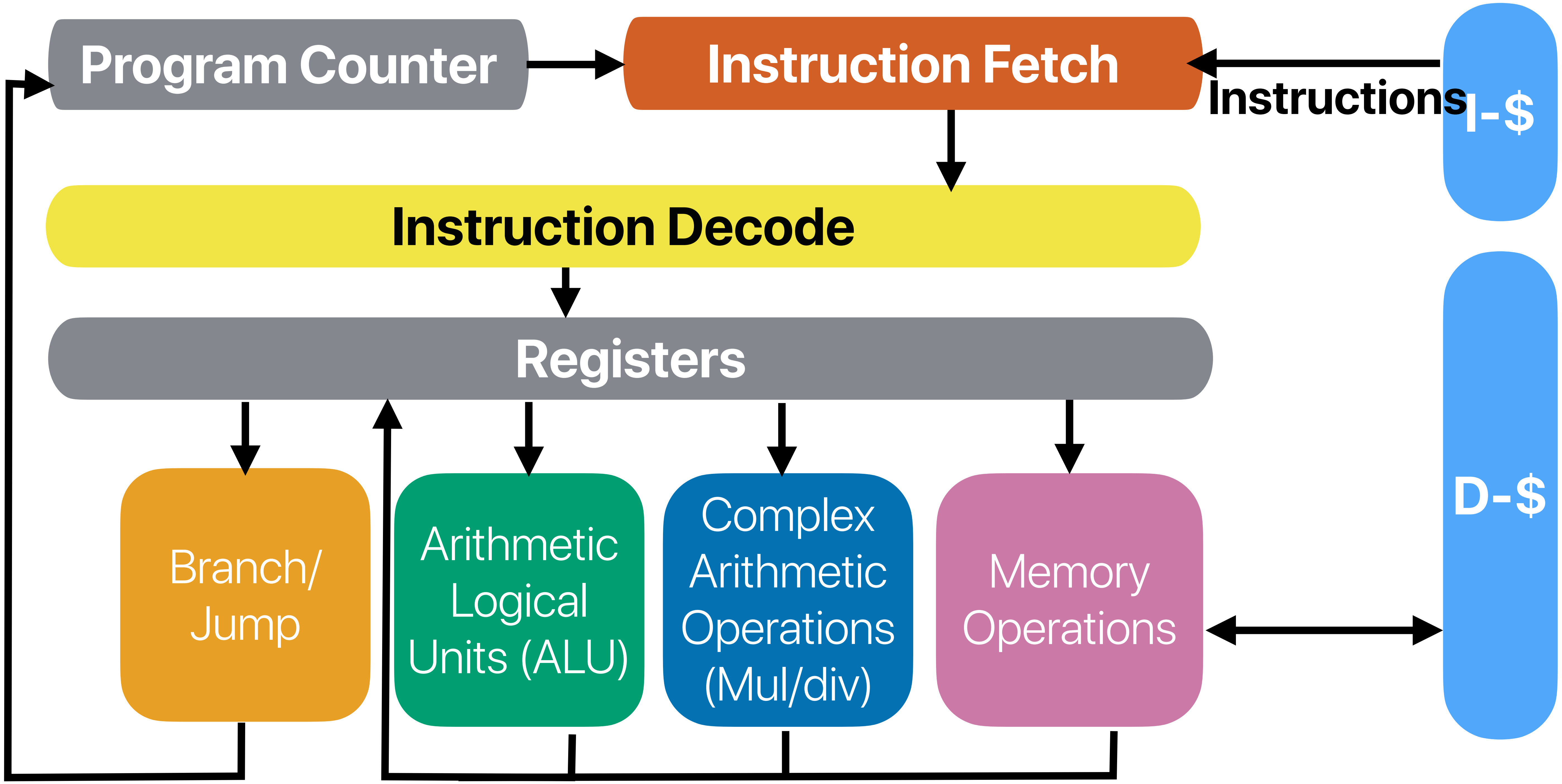
```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
```





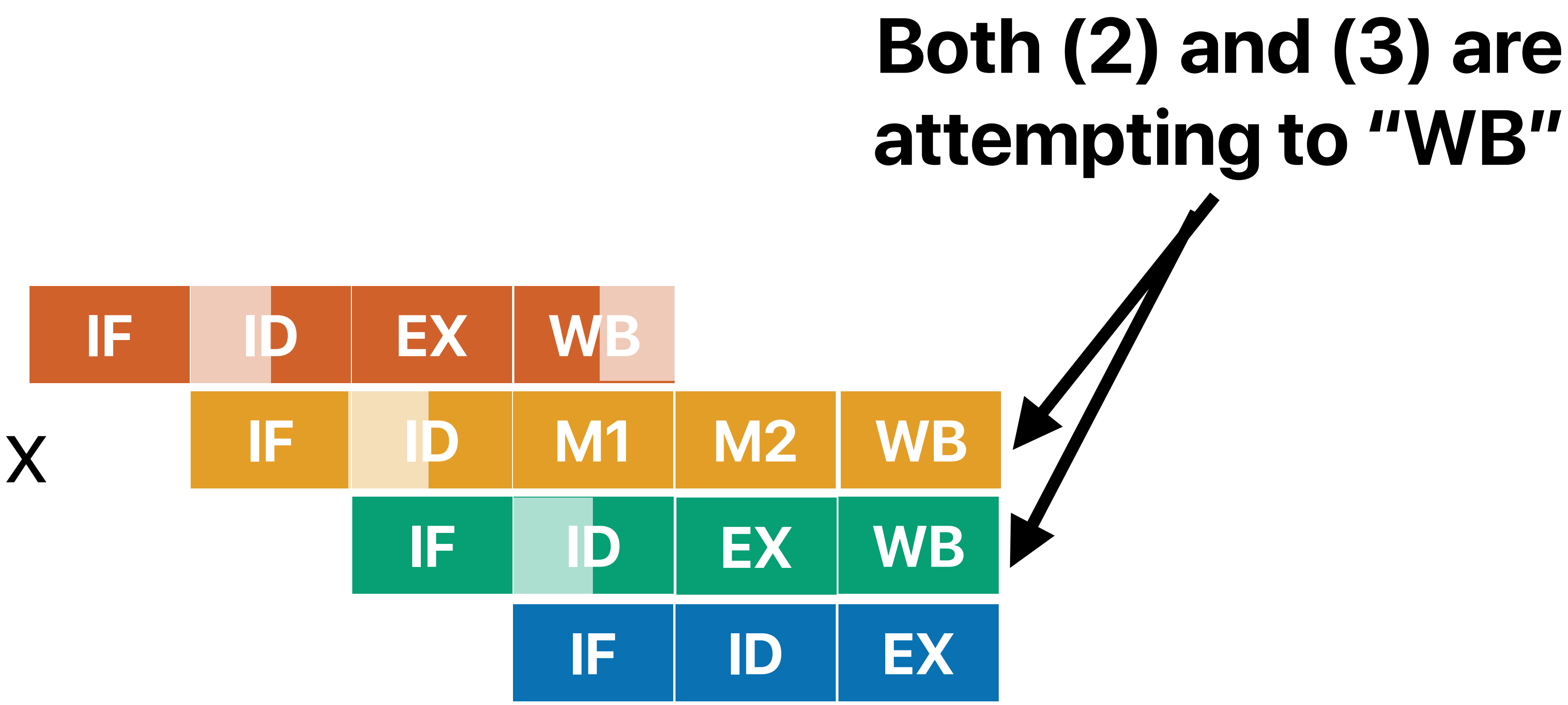
"Split L1" cache!

Split L1-$


Program Counter
Instruction Fetch
Instructions
I-$
Instruction Decode
Registers
Branch/ Jump
Arithmetic Logical Units (ALU)
Complex Arithmetic Operations (Mul/div)
Memory Operations
D-$

What if the memory instruction needs more time?

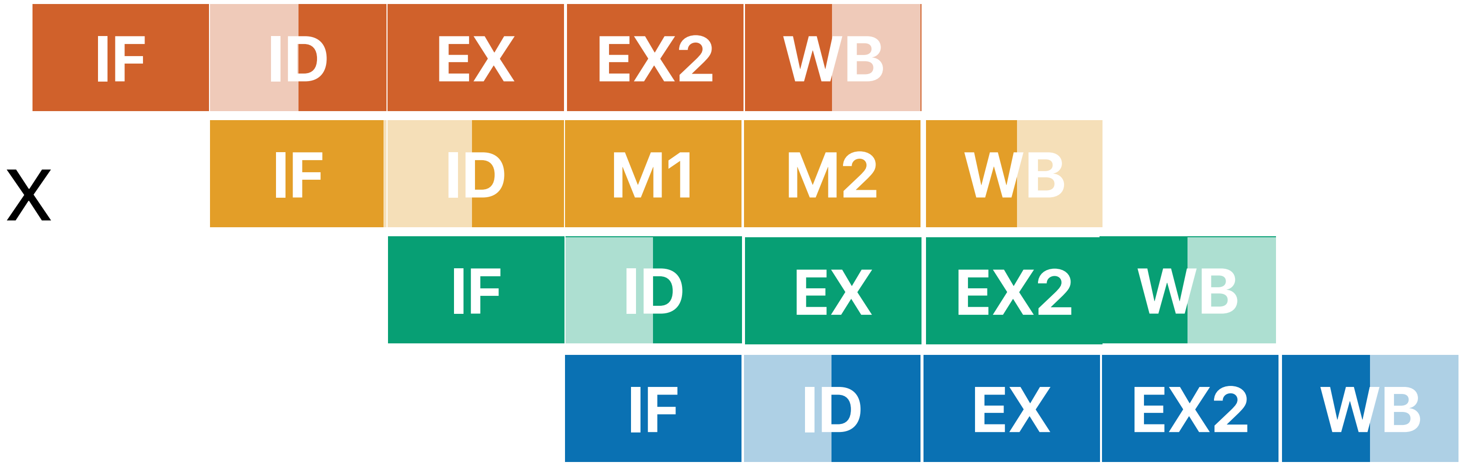
① `xorl %eax, %eax`
② `movl (%rdi), %ecx`
③ `addl %ecx, %eax`
④ `addq $4, %rdi`



What if the memory instruction needs more time?

- Every instruction needs to go through exactly the same number of stages

```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
```



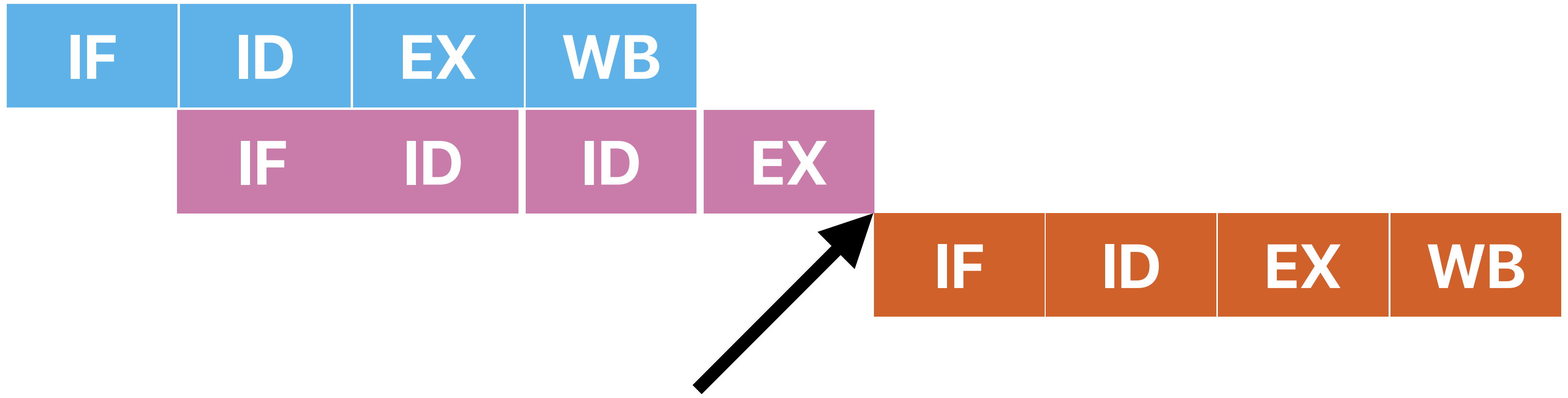
Structural Hazards

- Stall can address the issue — but slow
- Improve the pipeline unit design to allow parallel execution
  - Write-first, read later register files
  - Split L1-Cache
  - Force all instructions go through exactly the same number of stages

Control Hazards

Control Hazard

```
① cmpq %rdx, %rdi
② jne  .L3
③ ret
```



We cannot know if we should fetch (7) or (2) before the EX is done

How does the code look like?

```
for (j = 0; j < reps; ++j) {
    for (unsigned i = 0; i < size; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

We skip the following code block if data[i]< threshold

We use "backward" branches (taking if going back) to implement loops

```
loop0:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    movq  %rdi, -24(%rbp)
    movl  %esi, -28(%rbp)
    movl  %edx, -32(%rbp)
    movl  %ecx, -36(%rbp)
    movl  $0, -8(%rbp)
    movl  $0, -12(%rbp)
    jmp   .L2
.L6:
    movl  $0, -4(%rbp)
    jmp   .L3
.L5:
    movl  -4(%rbp), %eax
    leaq  0(,%rax,4), %rdx
    movq  -24(%rbp), %rax
    addq  %rdx, %rax
    movl  (%rax), %eax
    cmpl  %eax, -32(%rbp)
    jg .L4
    addl  $1, -8(%rbp)
.L4:
    addl  $1, -4(%rbp)
.L3:
    movl  -28(%rbp), %eax
    cmpl  %eax, -4(%rbp)
    jb .L5
    addl  $1, -12(%rbp)
.L2:
    movl  -12(%rbp), %eax
    cmpl  -36(%rbp), %eax
    jl .L6
    movl  -8(%rbp), %eax
    popq  %rbp
    .cfi_def_cfa 7, 8
    ret
```

Announcements

- Plan your time carefully! — Time management is a skill that could be more useful than all other things you learned from CSE142/L
- Assignment #3 — due this Sunday
- Lab #2 — due this Thursday

Computer
Science &
Engineering

つづく