

Modern Processor Design (I): in the pipeline

Hung-Wei Tseng

What “tips” of have you heard before regarding writing a faster program?

Tips (or myth) of performance programming?

- Bitwise operators?
- Shorter code?
- Avoid multiplications?

Outline

- Pipelined Processor
- Pipeline Hazards
 - Structural Hazards
 - Control Hazards
 - Data Hazards

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? (you may or may not consider optimizations)
Performance xor
- A. Version A
- B. Version B
- C. They are about the same (sometimes A is faster, sometimes B is)

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? (you may or may not consider optimizations)
 - Version A
 - Version B
 - They are about the same (sometimes A is faster, sometimes B is)

Recap: Why adding a sort makes it faster

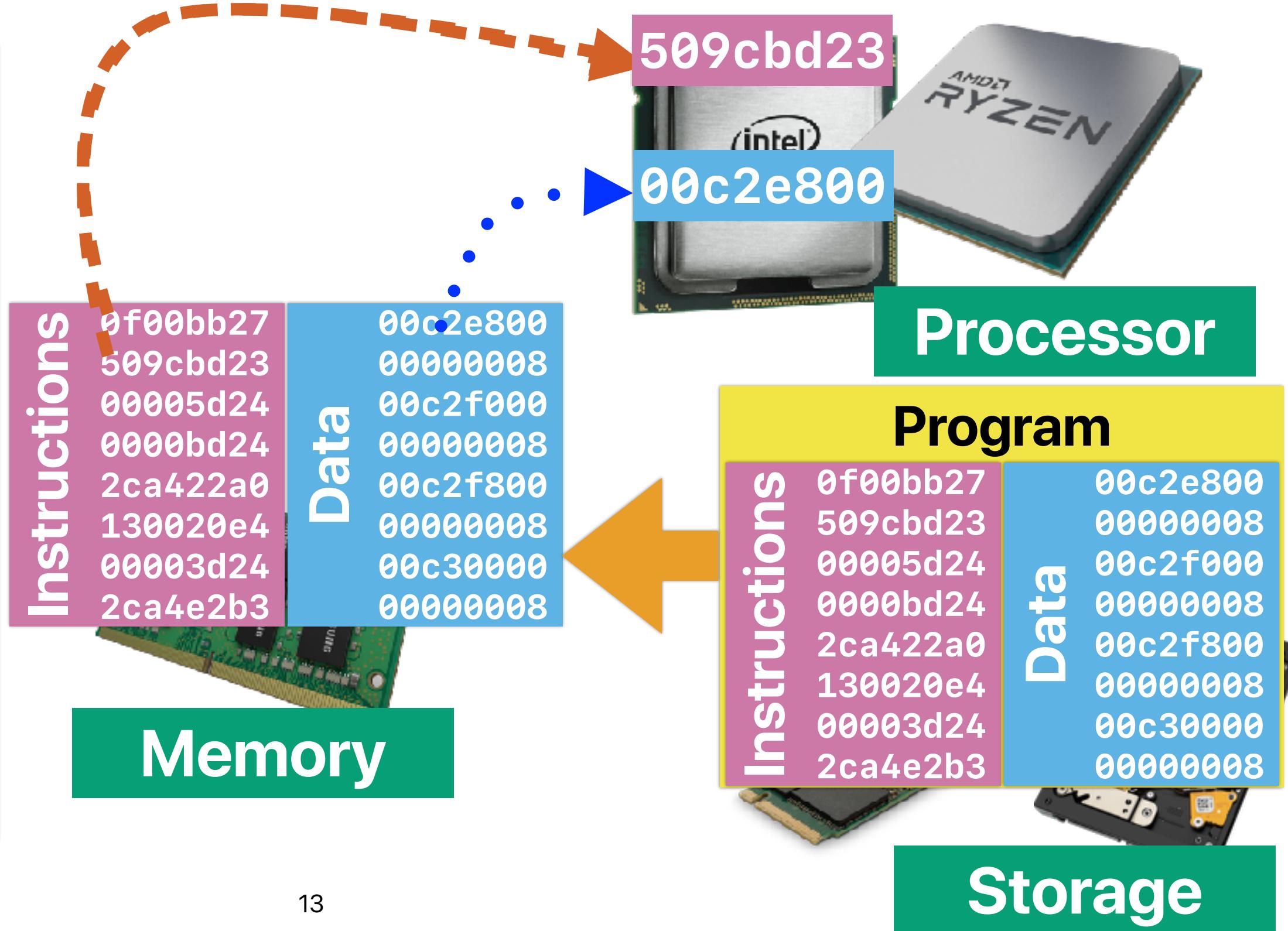
- Why the sorting the array speed up the code despite the increased instruction count?

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

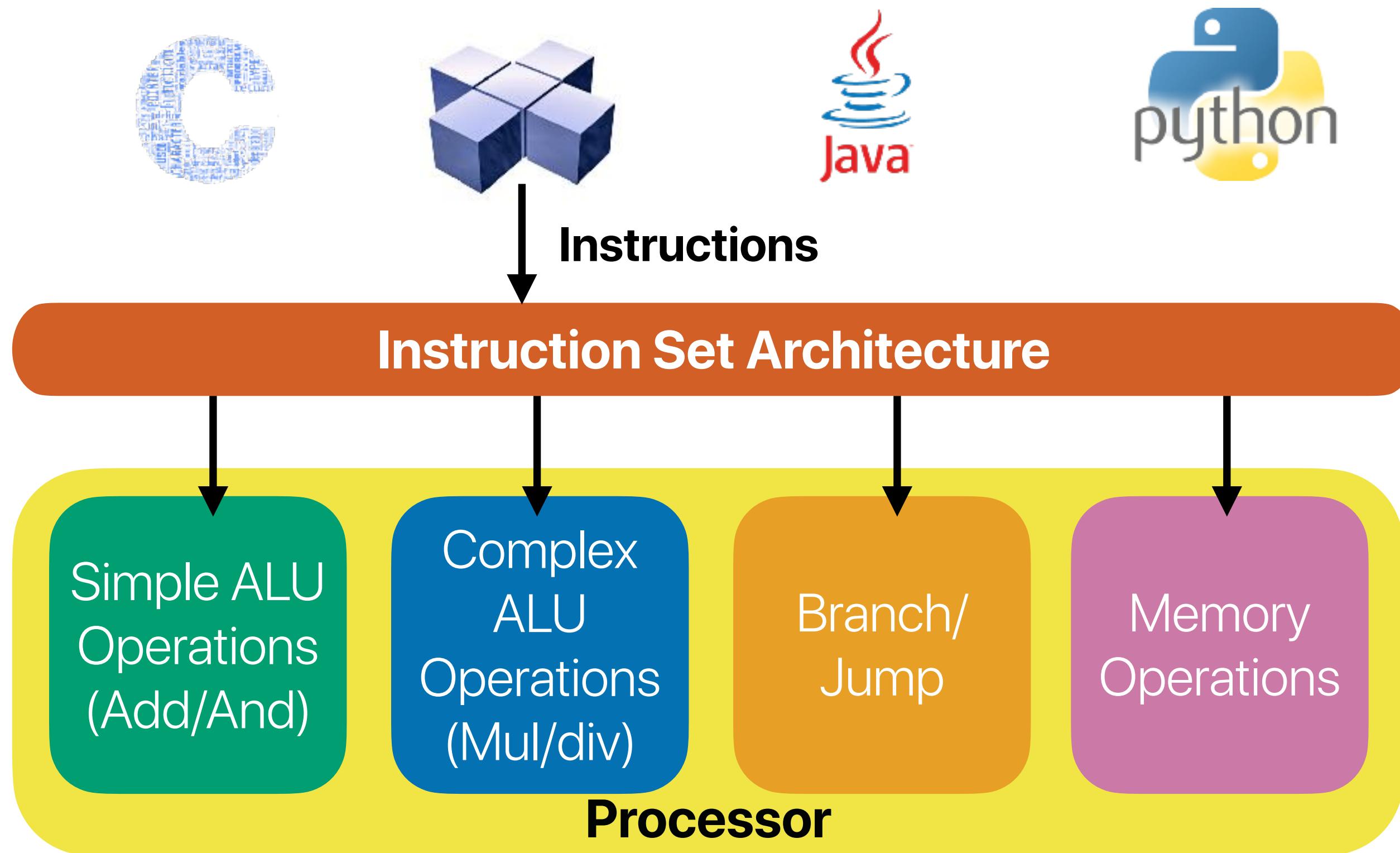
for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum++;
    }
}
```

Basic Processor Design

von Neumann Architecture



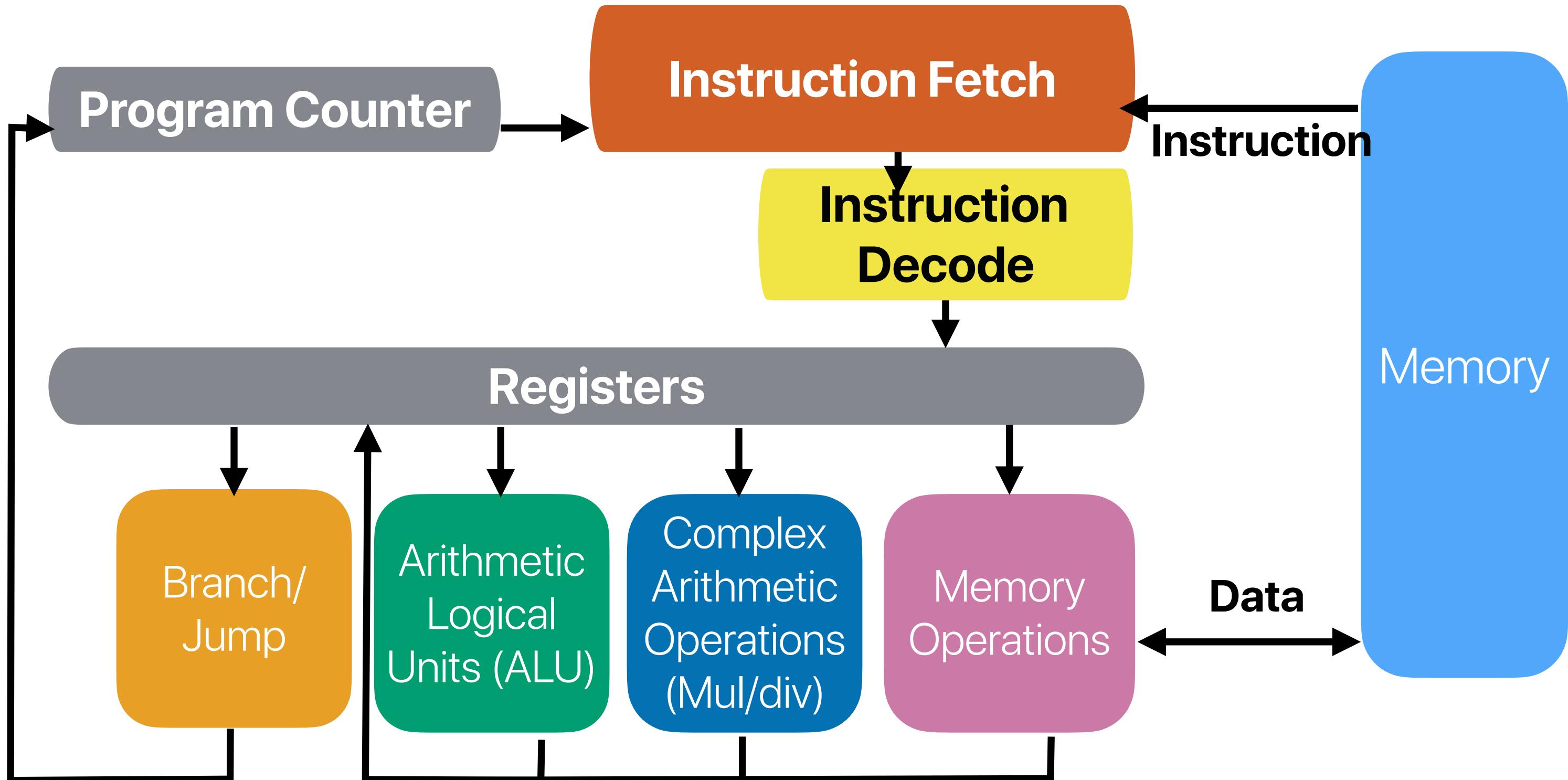
Recap: Microprocessor — a collection of functional units

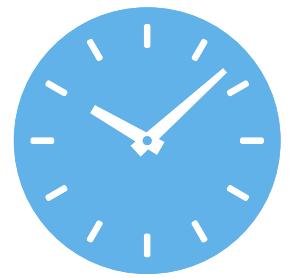


The “life” of an instruction

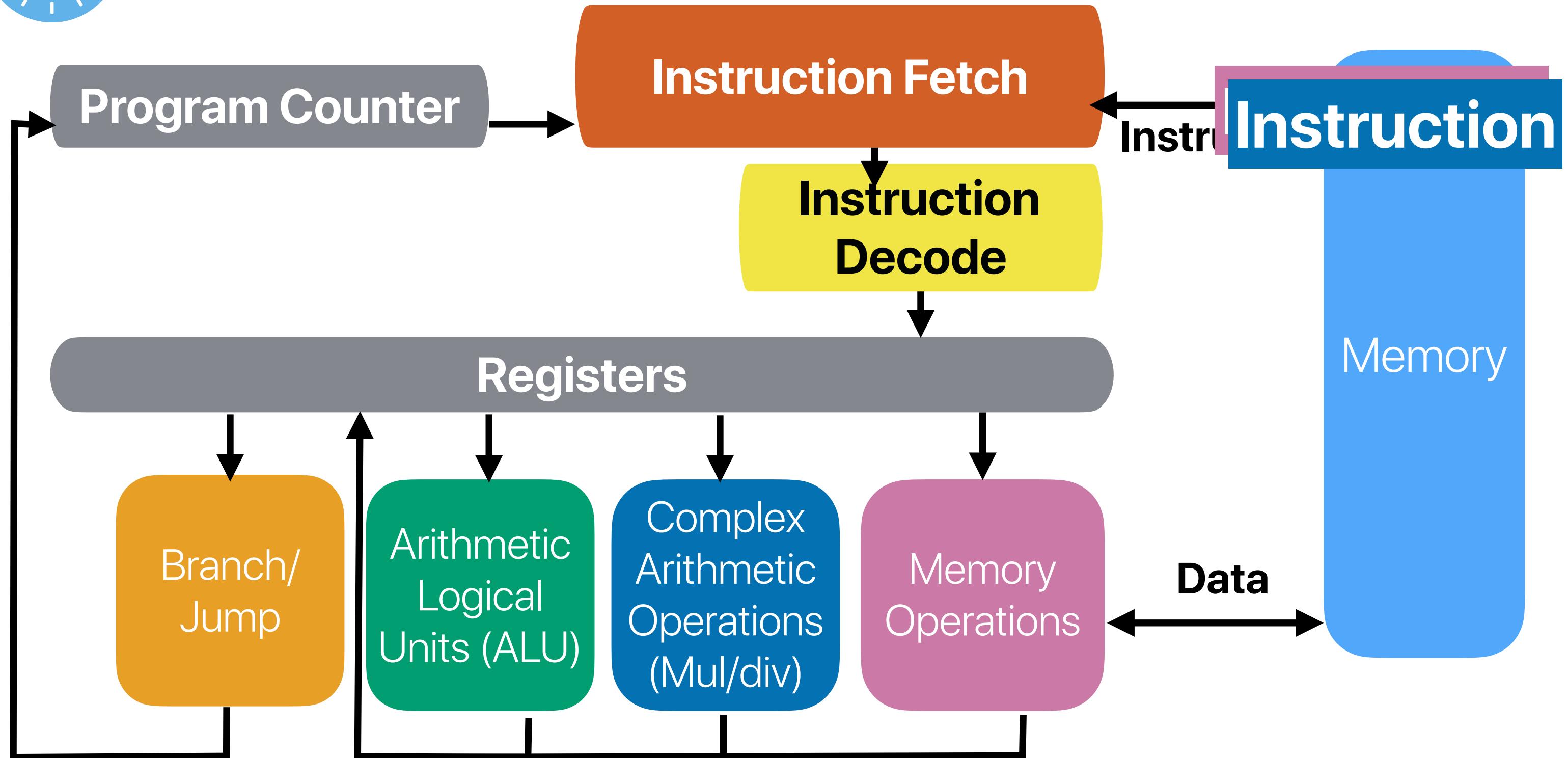
- Instruction Fetch (**IF**) — fetch the instruction from memory
- Instruction Decode (**ID**)
 - Decode the instruction for the desired operation and operands
 - Reading source register values
- Execution (**EX**)
 - ALU instructions: Perform ALU operations
 - Conditional Branch: Determine the branch 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

“Basic” idea of the processor

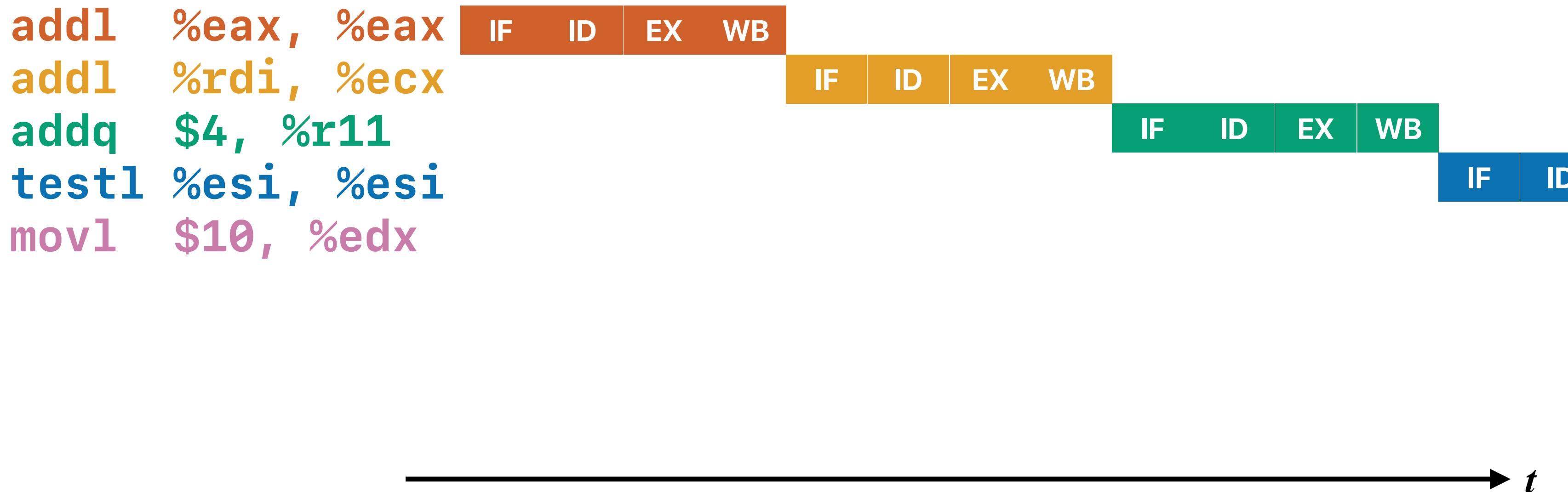




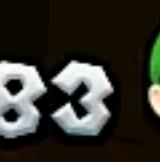
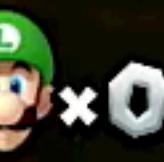
Within a cycle...



Simple implementation w/o branch



Pipelining

 ×83  ×05  ×05  ×04





 41

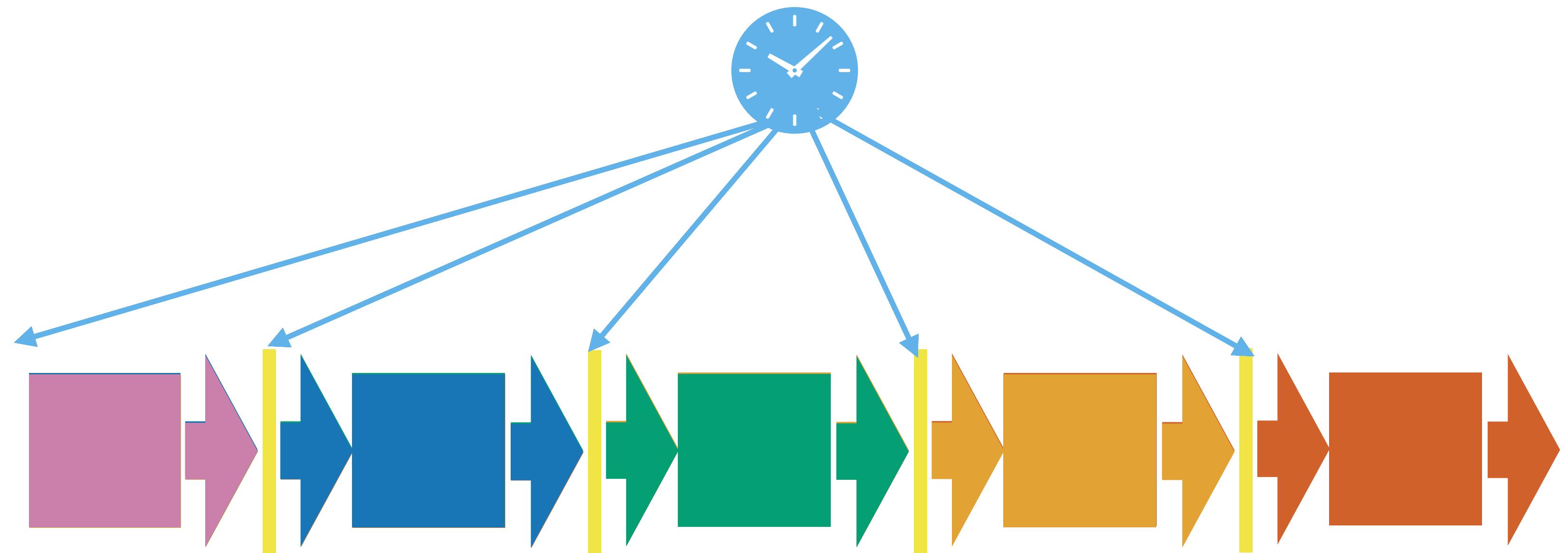
004861790  163



Pipelining

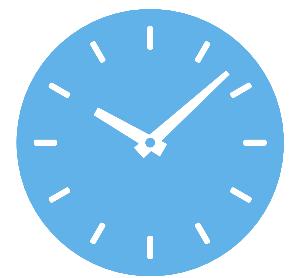
- Different parts of the processor works on different instructions simultaneously
- A processor is now working on multiple instructions from the same program (though on different stages) simultaneously.
 - ILP: **Instruction-level parallelism**
- A **clock** signal controls and synchronize the beginning 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

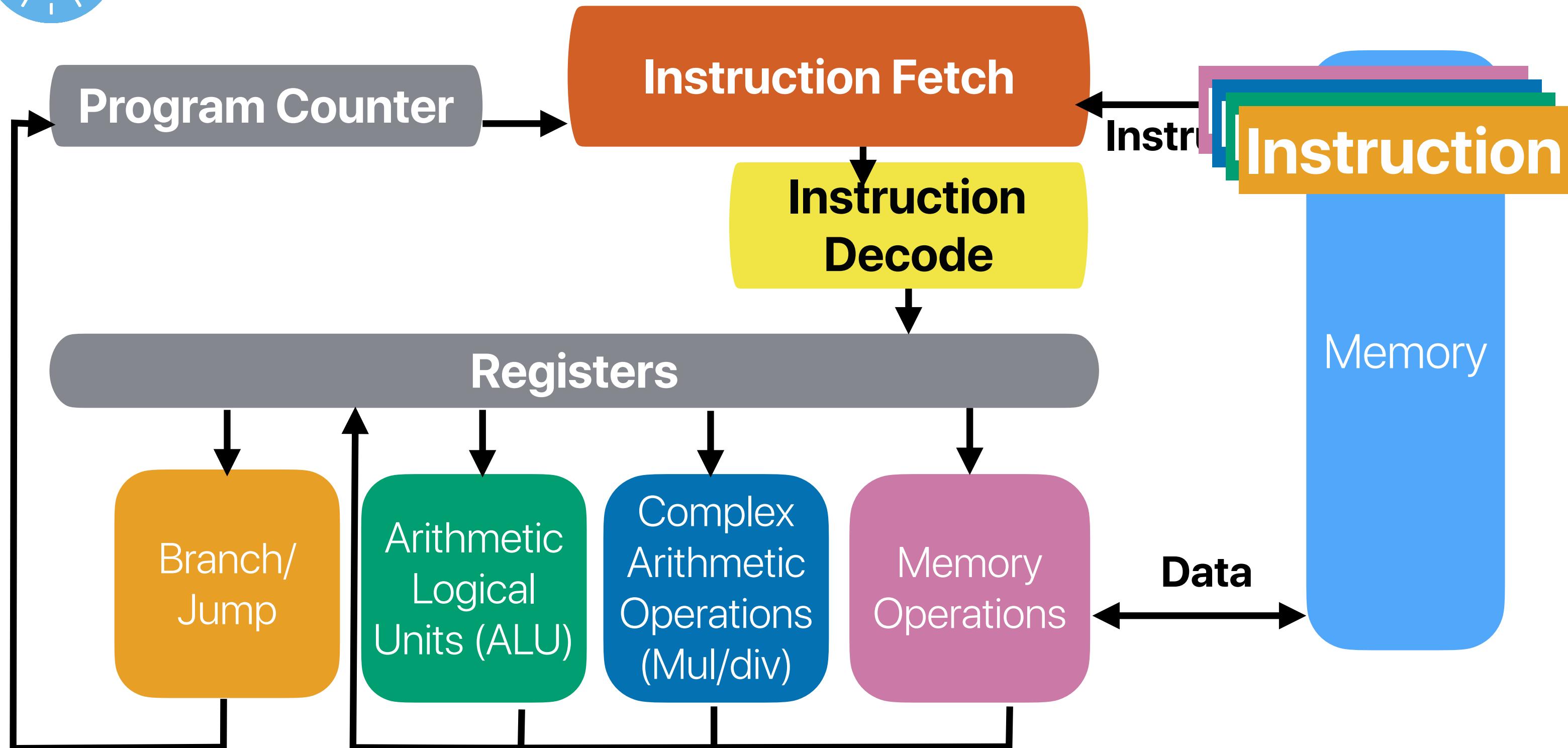


Pipelining



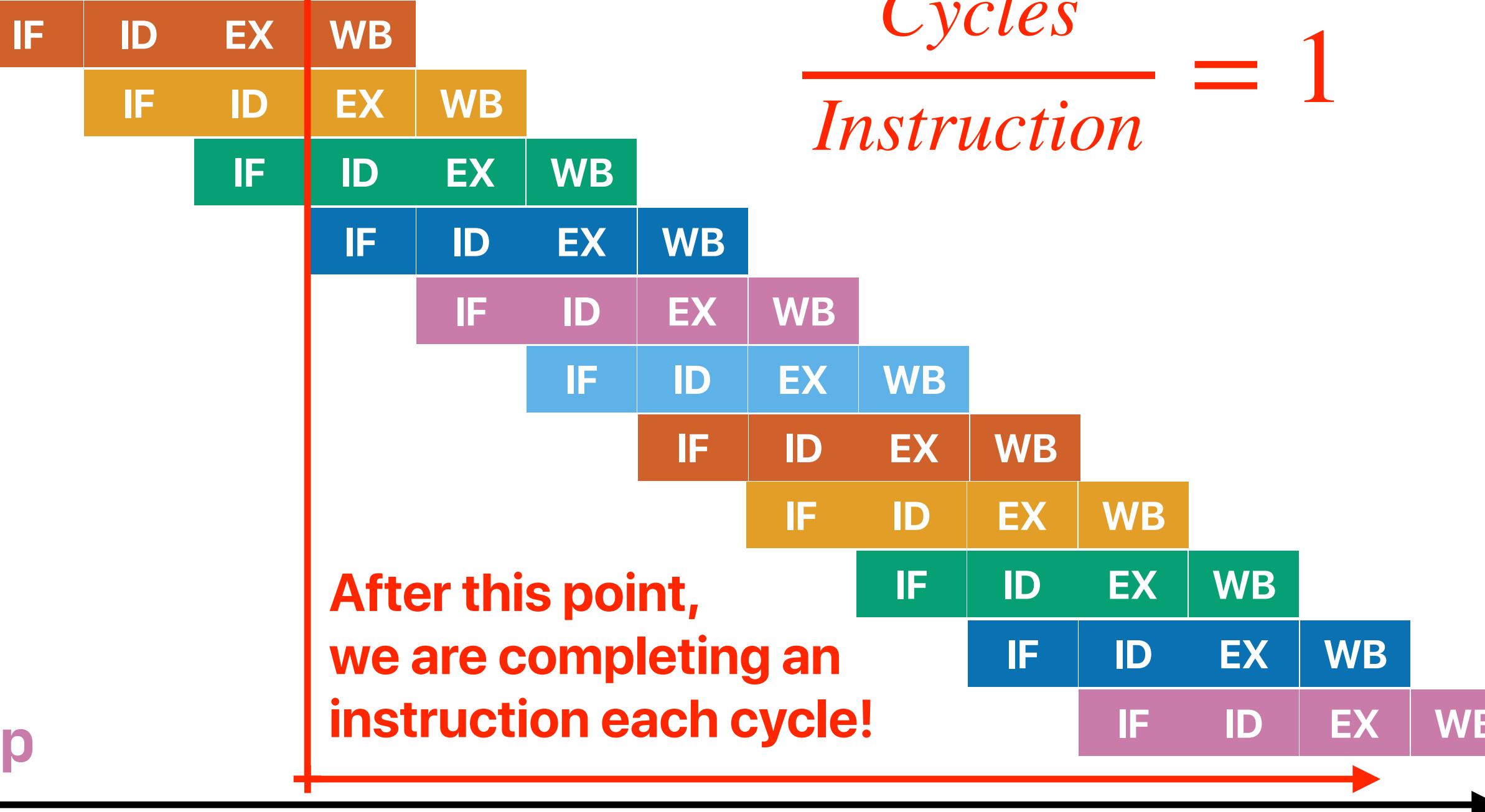


"Pipeline" the processor!



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



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

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

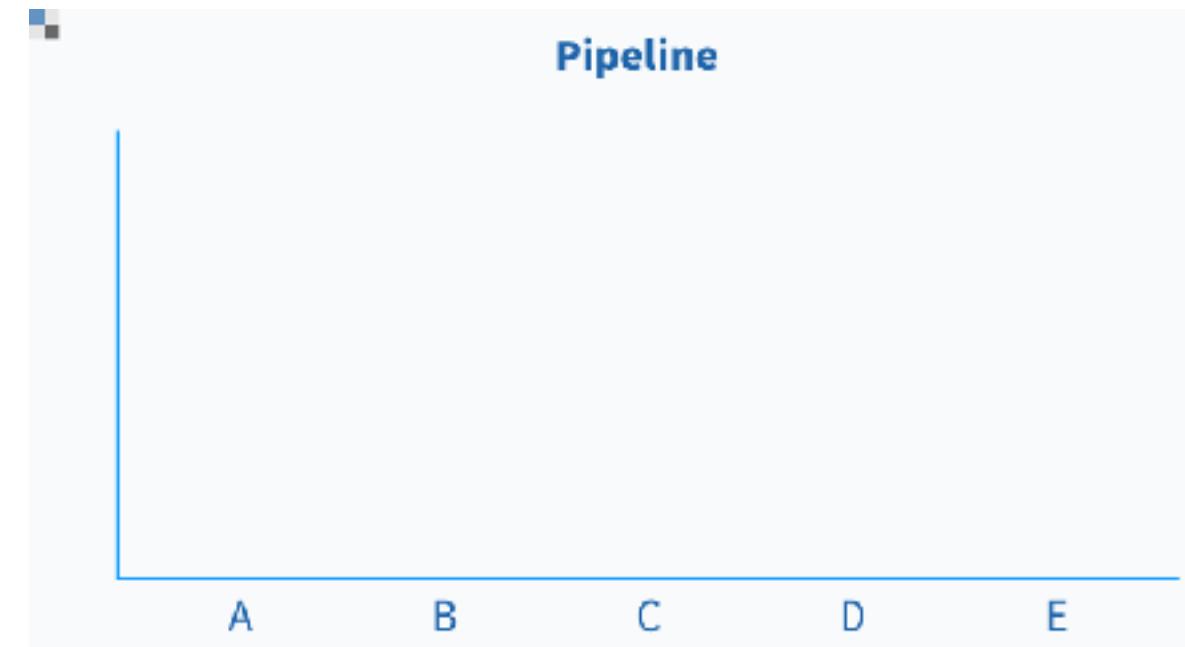
How well can we pipeline?

- 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
```

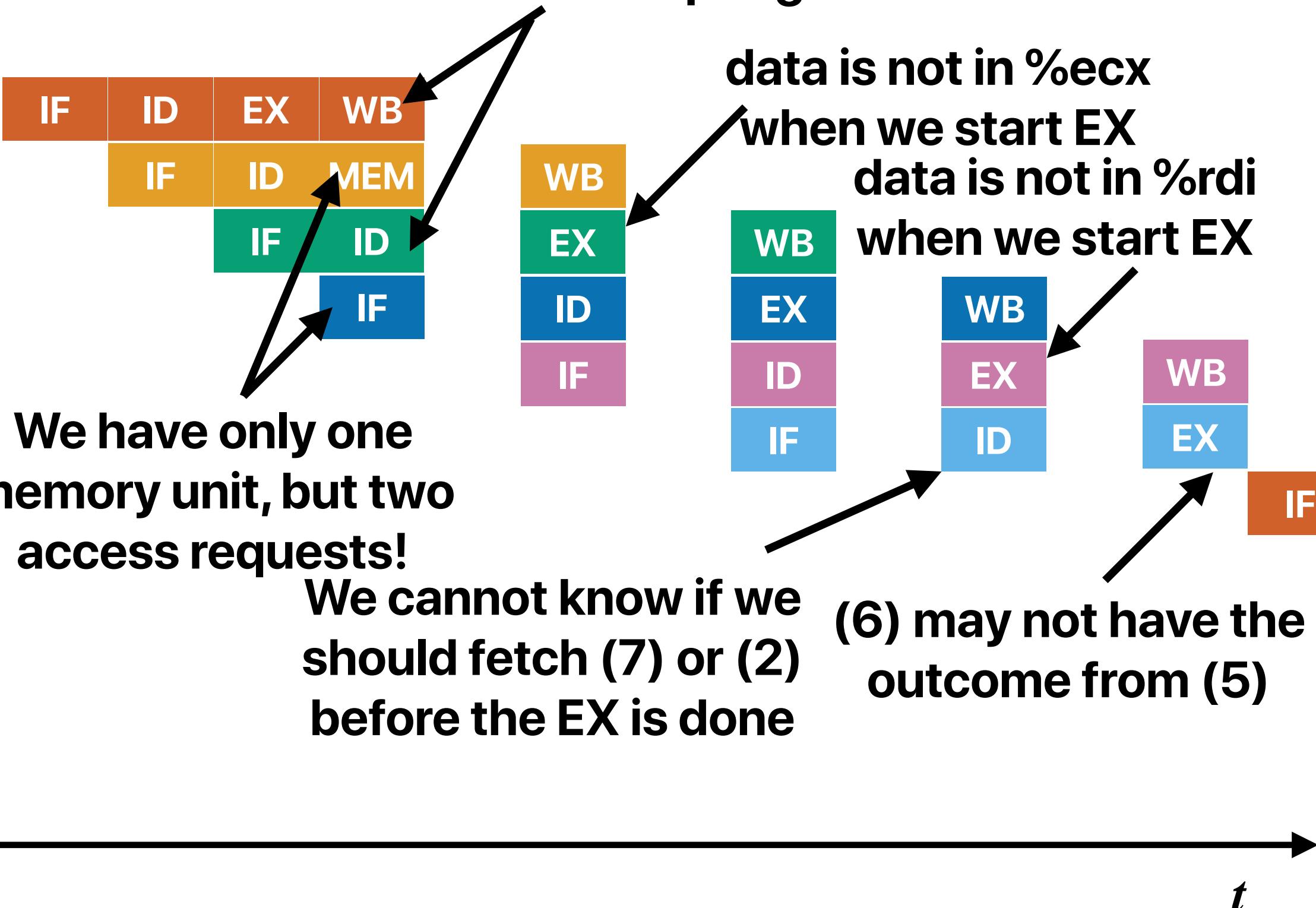
- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

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



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 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

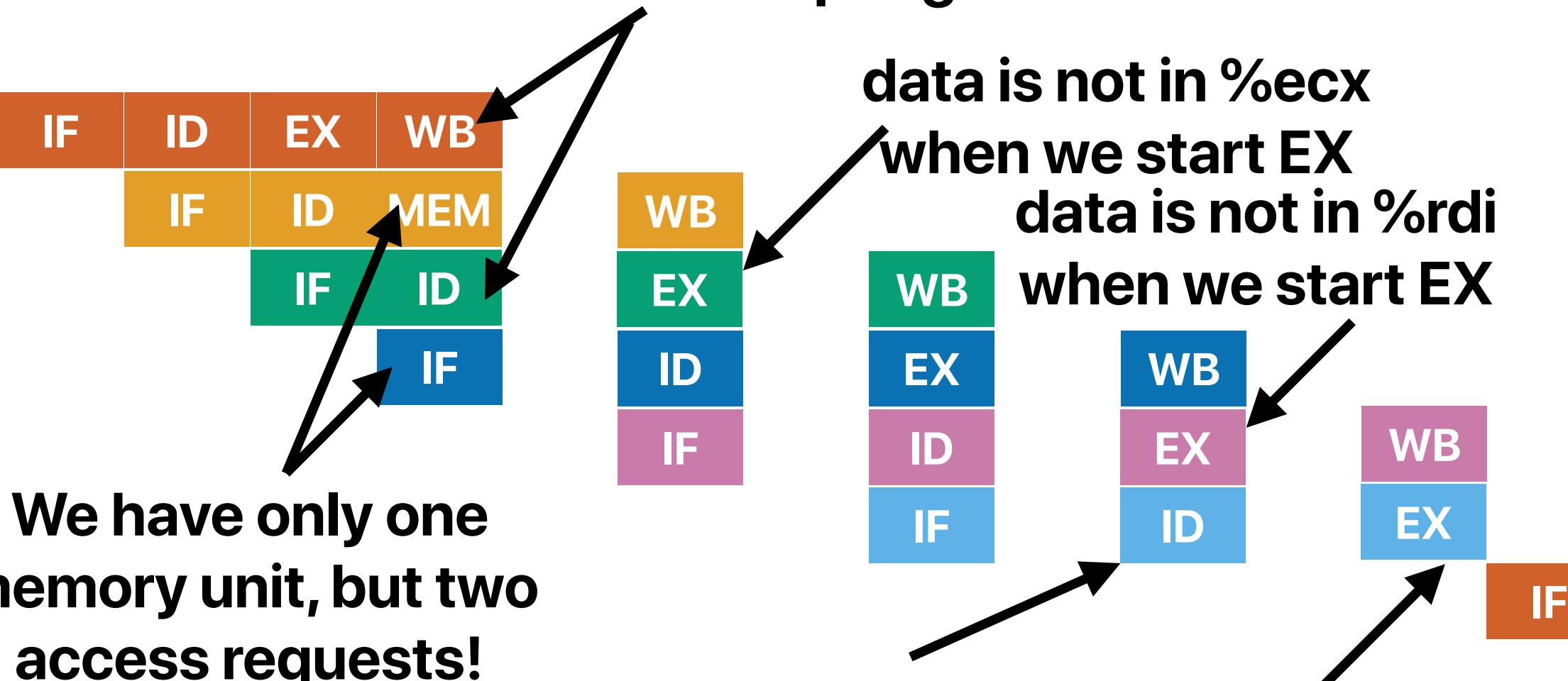
Pipeline hazards

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

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

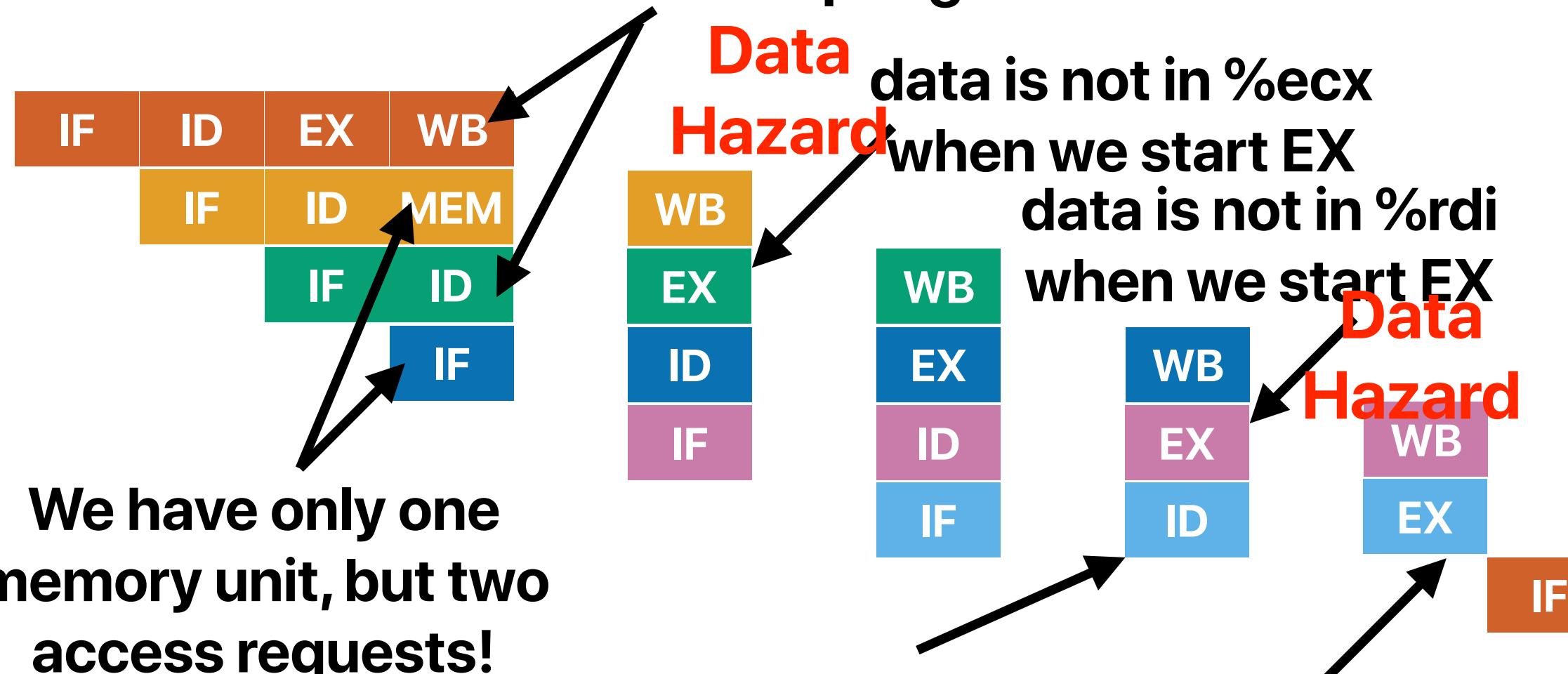
We cannot know if we should fetch (7) or (2) before the EX is done



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

**Structural
Hazard**



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

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

**Control
Hazard**

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

Data Hazard

Why is A 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;  
}
```

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

Slow! — 5 additional cycles

Structural Hazards

Dealing with the conflicts between ID/WB

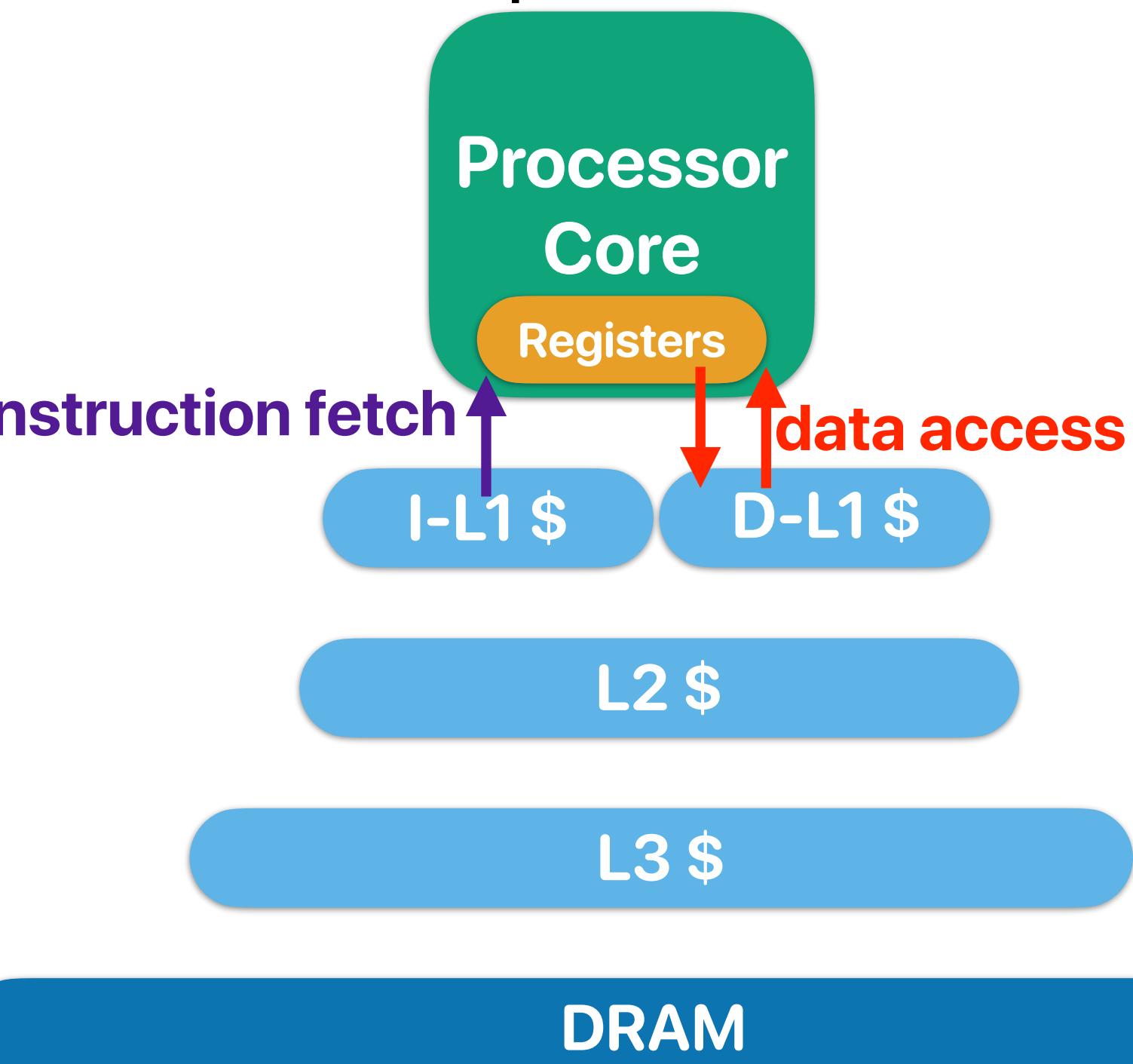
- The same register cannot be read/written at the same cycle
- Better solution: write early, read late
 - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
 - This leaves enough time for outputs to settle for reads
 - The revised register file is the default one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

①	xorl %eax, %eax	IF	ID	EX	WB
②	movl (%rdi), %ecx	IF	ID	MEM	WB
③	addl %ecx, %eax	IF	ID	EX	WB

How to handle the conflicts between MEM and IF?

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

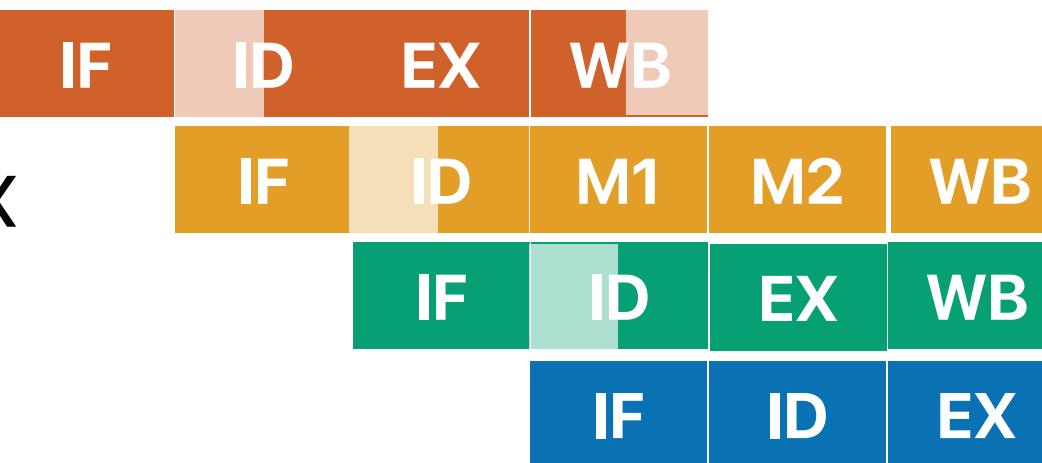
① xorl %eax, %eax	IF	ID	EX	WB
② movl (%rdi), %ecx	IF	ID	MEM	
③ addl %ecx, %eax	IF	ID		
④ addq \$4, %rdi			IF	



"Split L1" cache!

What if the memory instruction needs more time?

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

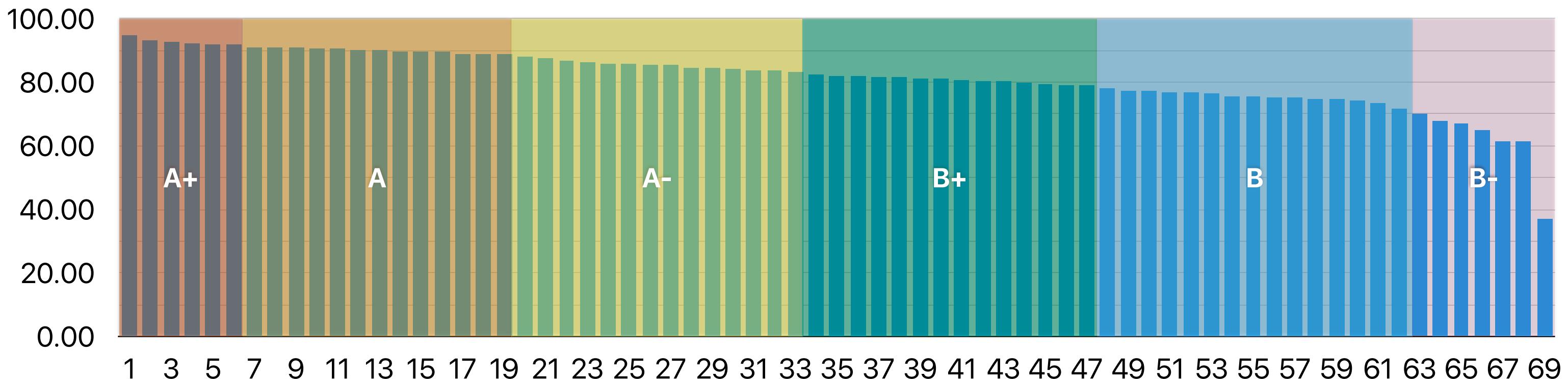


Both (2) and (3) are attempting to "WB"

Announcements

- Reading quiz due next **Monday!!!**
 - Your overall grade decides your final letter grade, not just the midterm. Midterm is only 20%
 - You should take assignments seriously — you learn more by doing it yourself.
 - Midterm and how are you doing so far
 - Midterm average is 65
 - Your overall grade decides your final letter grade, not just the midterm

Current “Total” in Canvas and “Projected” Letter Grades



Computer Science & Engineering

203

つづく

