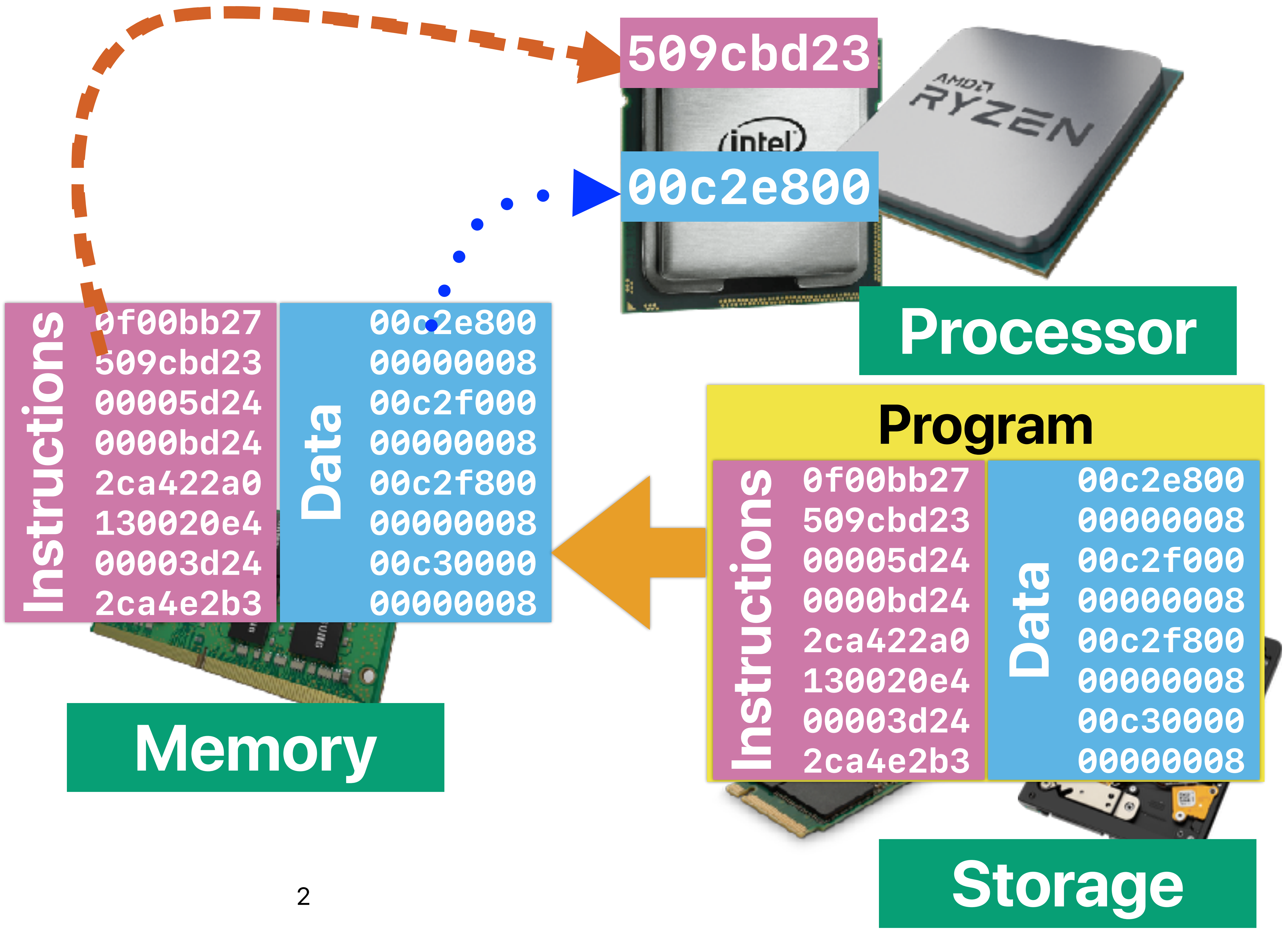
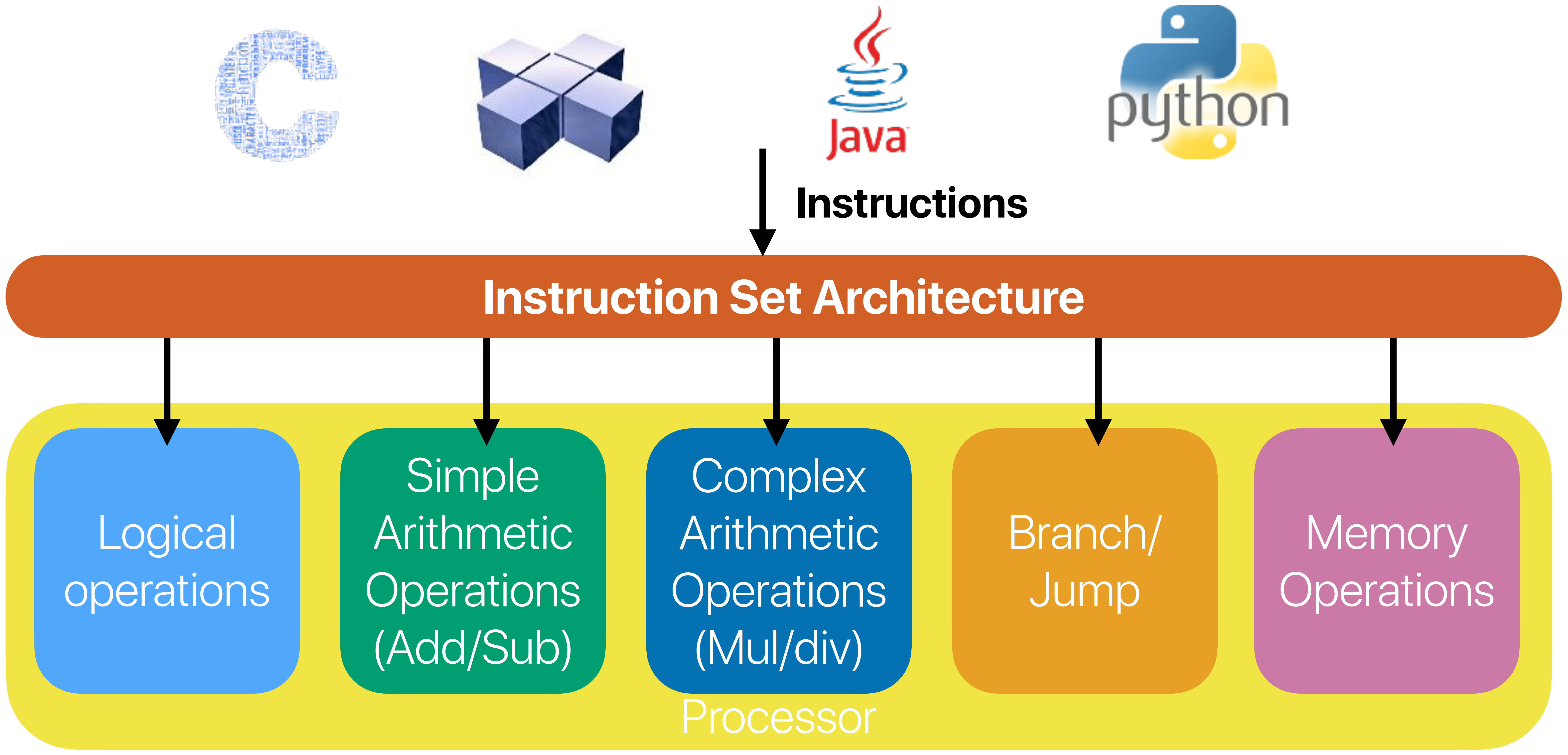
Modern Processor Design (I): in the pipeline

Hung-Wei Tseng

von Neumman Architecture



Recap: Microprocessor — a collection of functional units

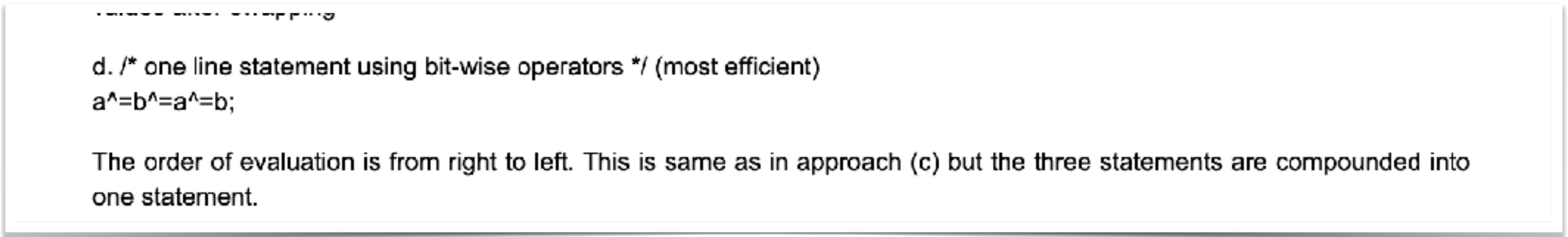

Java
python
Instructions
Instruction Set Architecture
Logical operations
Simple Arithmetic Operations (Add/Sub)
Complex Arithmetic Operations (Mul/div)
Branch/ Jump
Memory Operations
Processor

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/

Recap: Demo (3) — Bitwise operations?



d. /* one line statement using bit-wise operators */ (most efficient)
a^=b^=a^=b;

The order of evaluation is from right to left. This is same as in approach (c) but the three statements are compounded into one statement.

A

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

B

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

Recap: Leveraging more "bit-wise" operations in C code will make the program significantly faster

✗

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

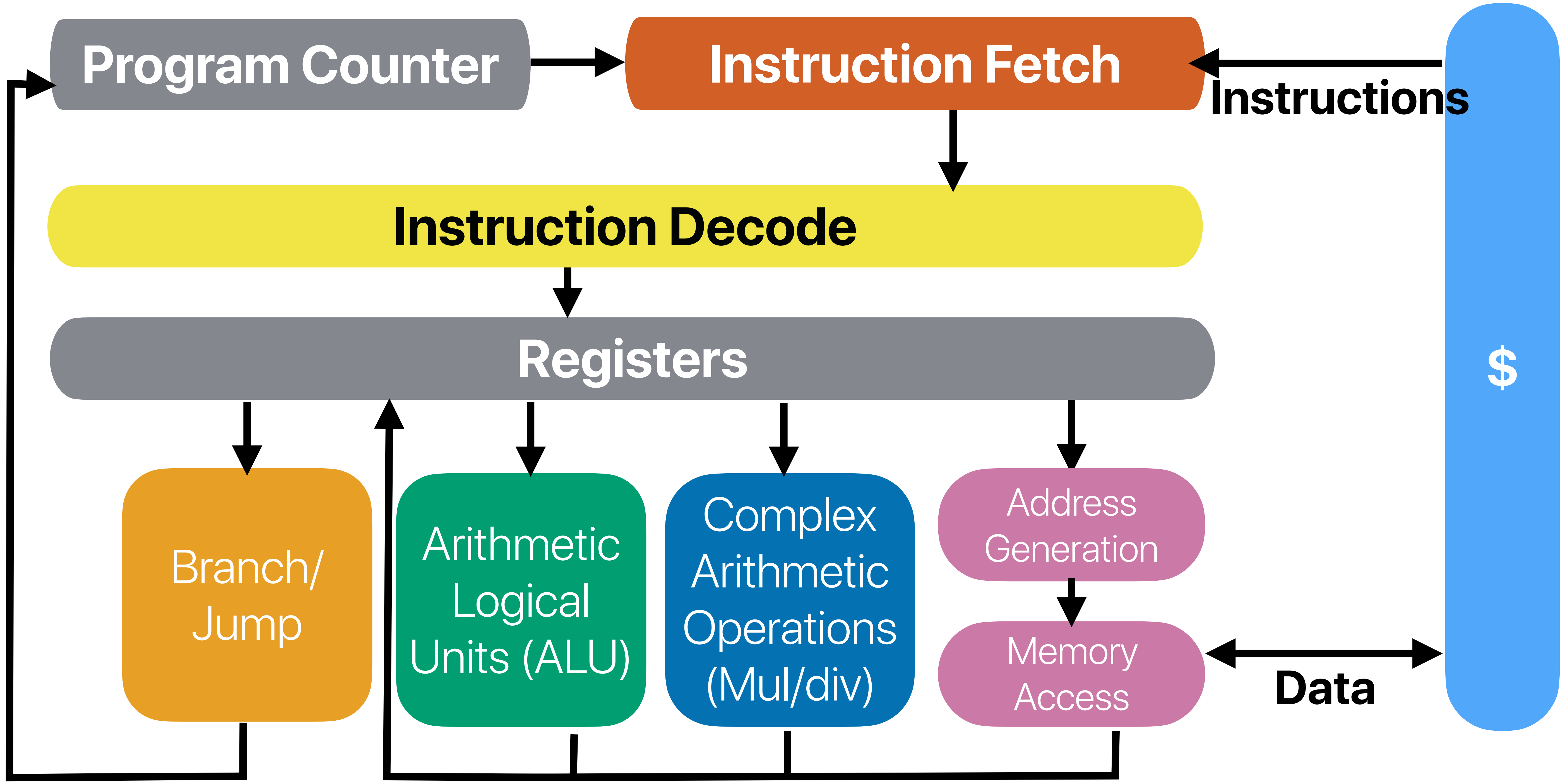
- Recap: the concept of a processor
- Pipelined Processor
- Pipeline Hazards
  - Structural Hazards
  - Control Hazards
  - Data Hazards

Basic Processor Design

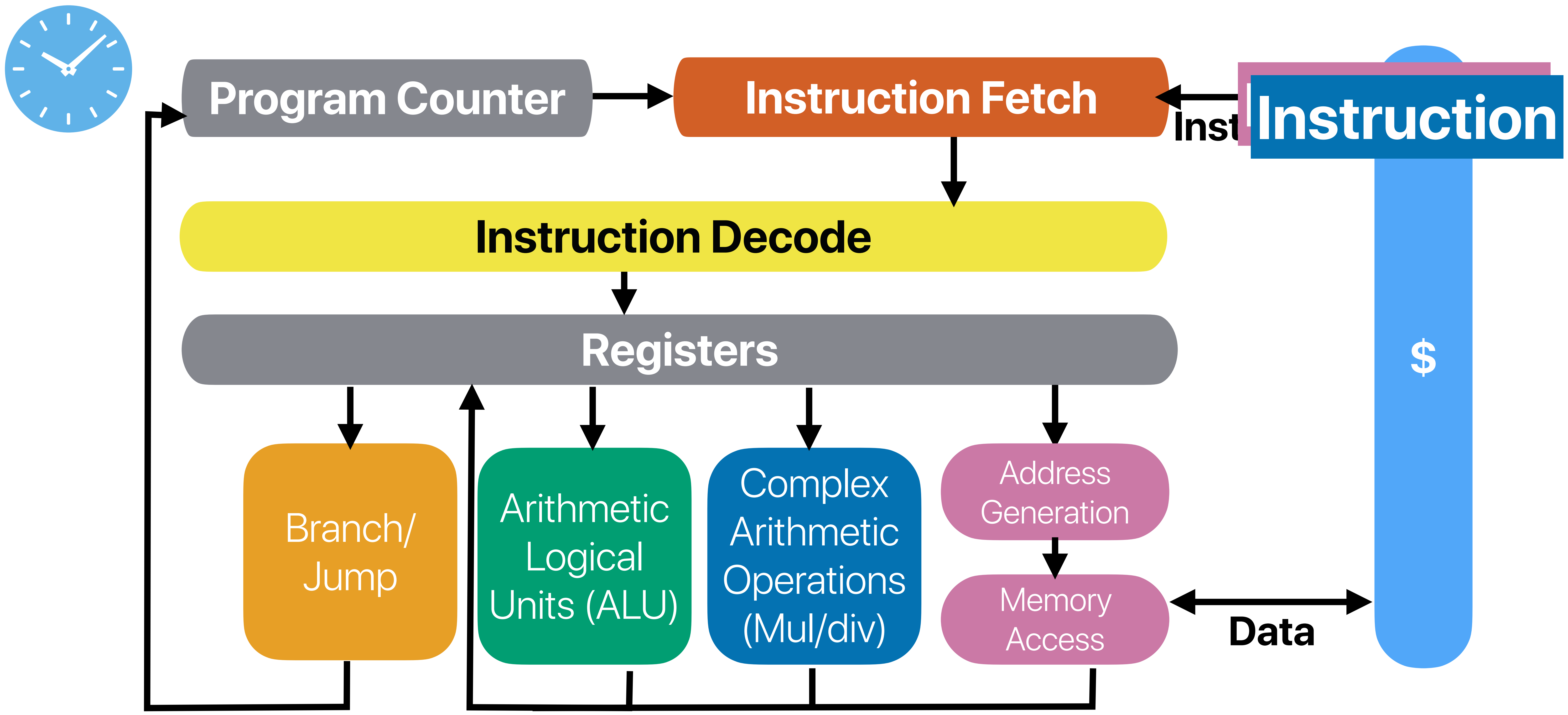
The “life” of an instruction

- 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

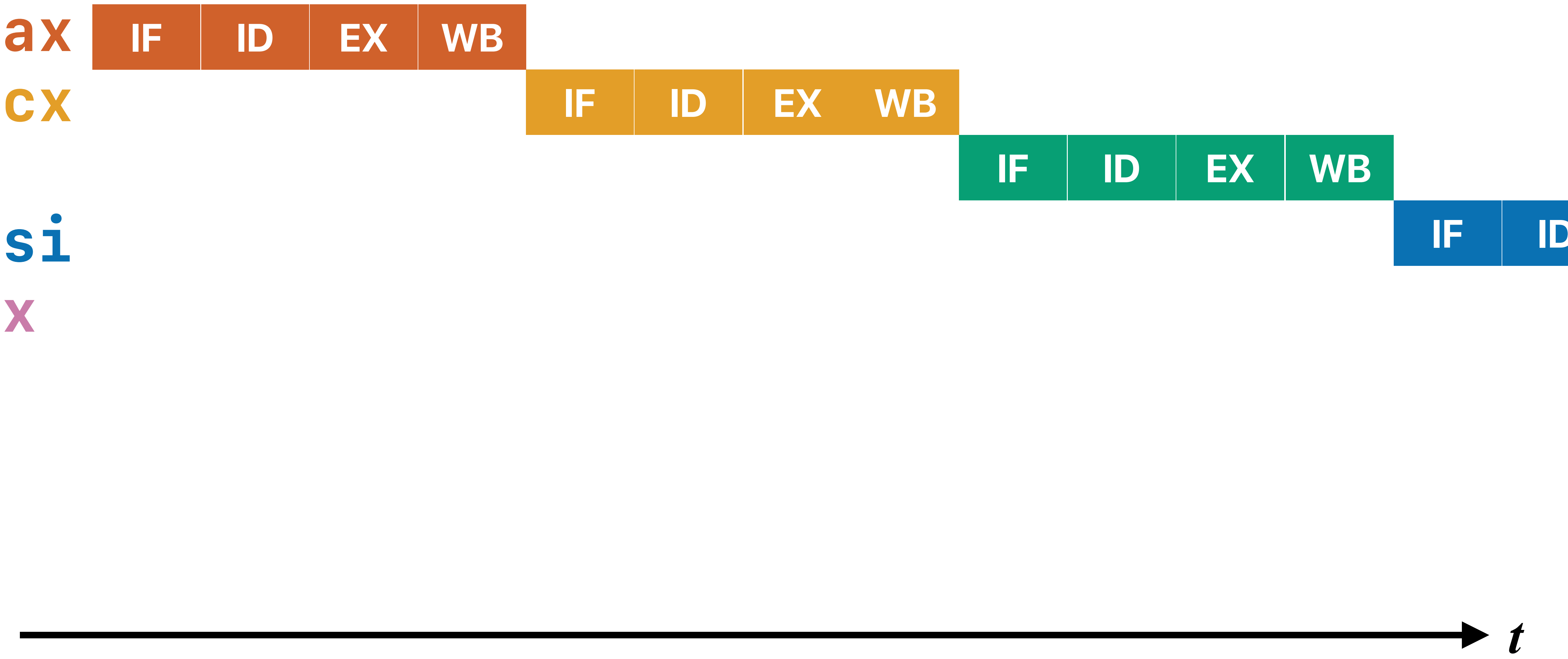


If we want to perform one instruction each cycle...



Simple implementation w/o branch

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

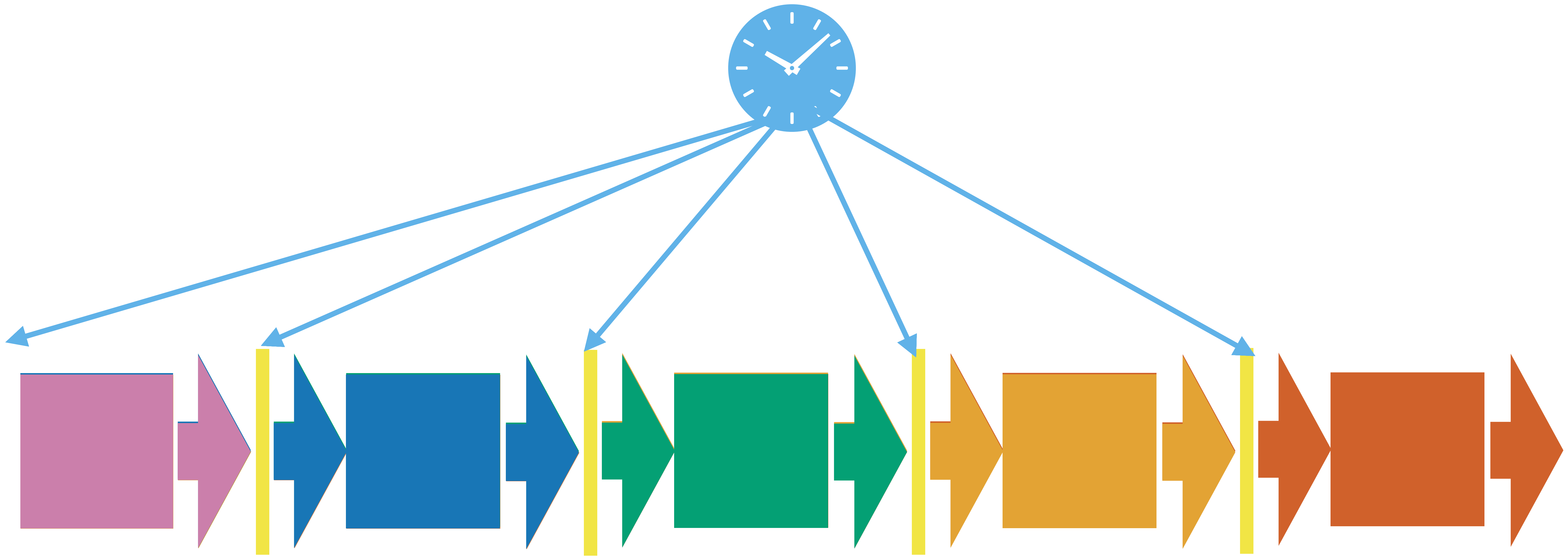


Pipelining

Pipelining

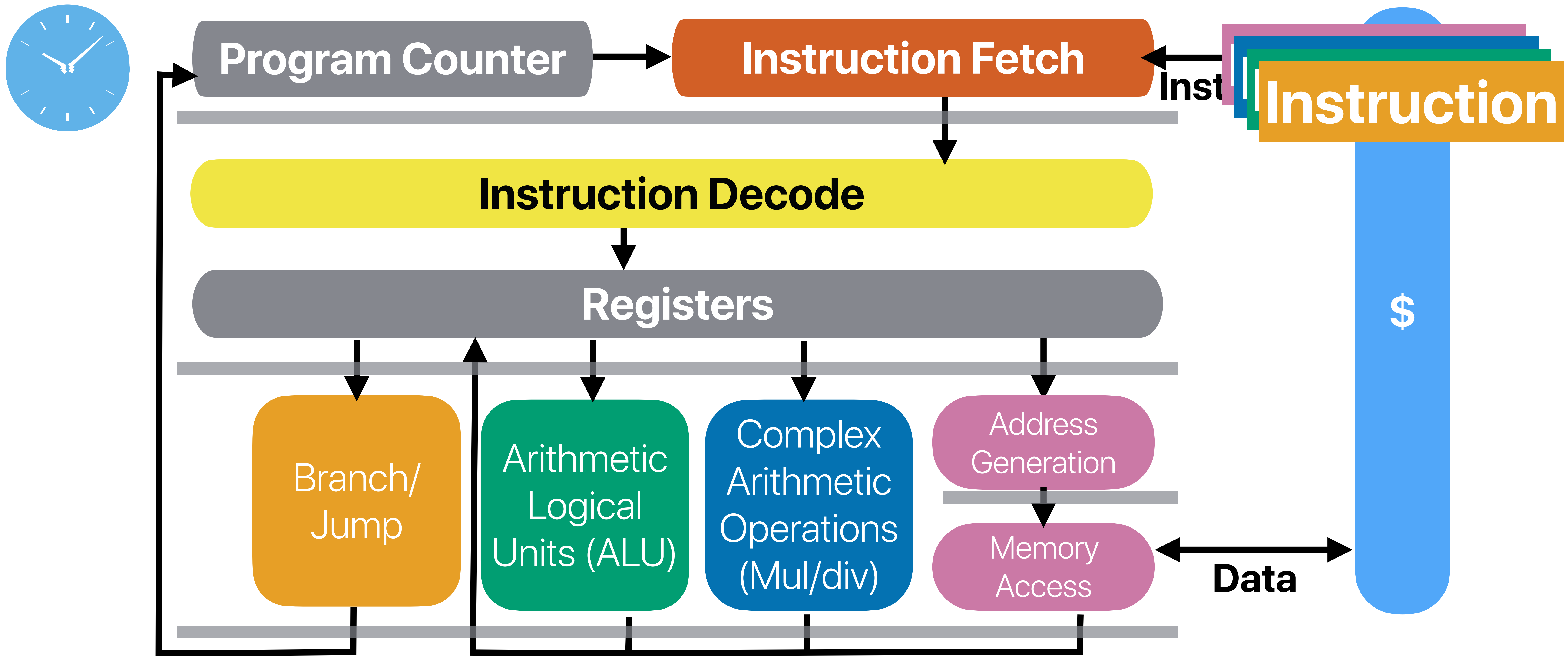
- Different parts of the processor works on different instructions simultaneously
- A processor is now working on multiple instructions from the same program (though on different stages) simultaneously.
  - ILP: **Instruction-level parallelism**
- A **clock** signal controls and synchronize the beginning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



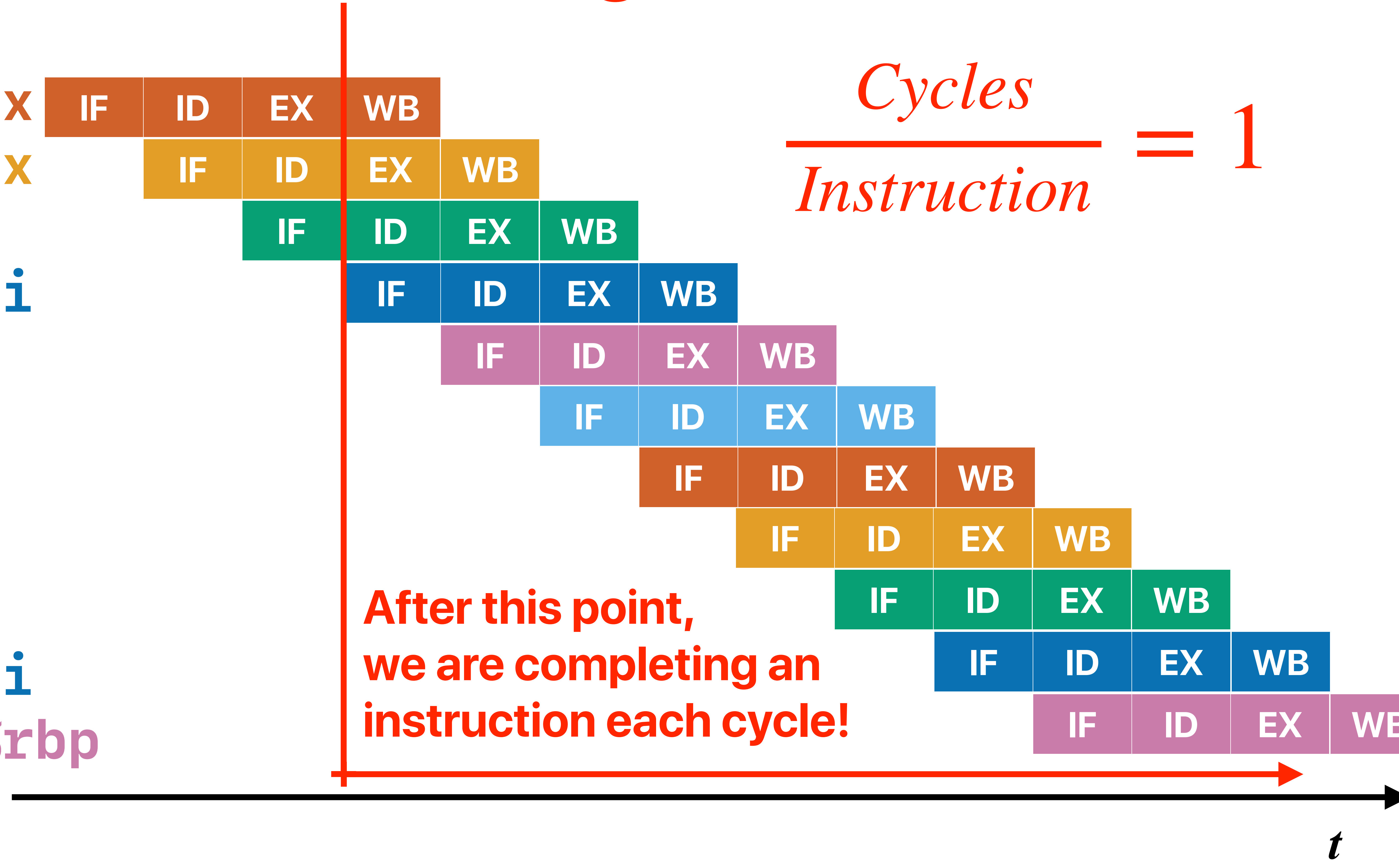
Pipelining

Pipelined execution



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
```

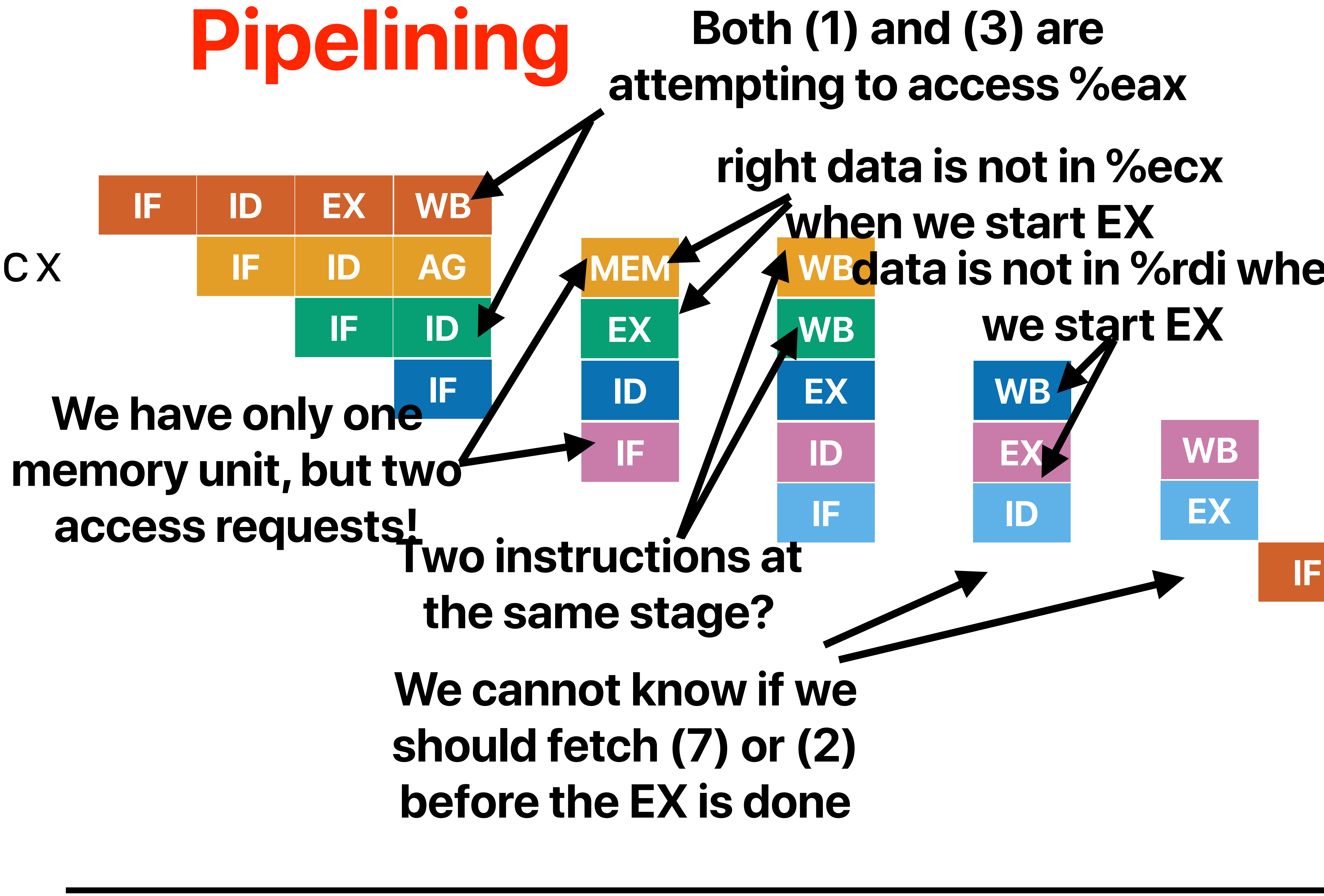


$$\frac{Cycles}{Instruction} = 1$$

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

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
right 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!
Two instructions at the same stage?
We cannot know if we should fetch (7) or (2) before the EX is done
IF
ID
EX
WB
IF
ID
AG
IF
ID
IF
MEM
EX
ID
IF
WB
WB
EX
ID
IF
WB
EX
ID
WB
EX
IF

Takeaways: pipeline processors

- Pipelining helps to improve the throughput of processors
  - Allowing shorter cycle time as each cycle only make progress for part of an instruction
  - Different pipeline stages work on different instructions concurrently
  - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time

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

Takeaways: pipeline processors

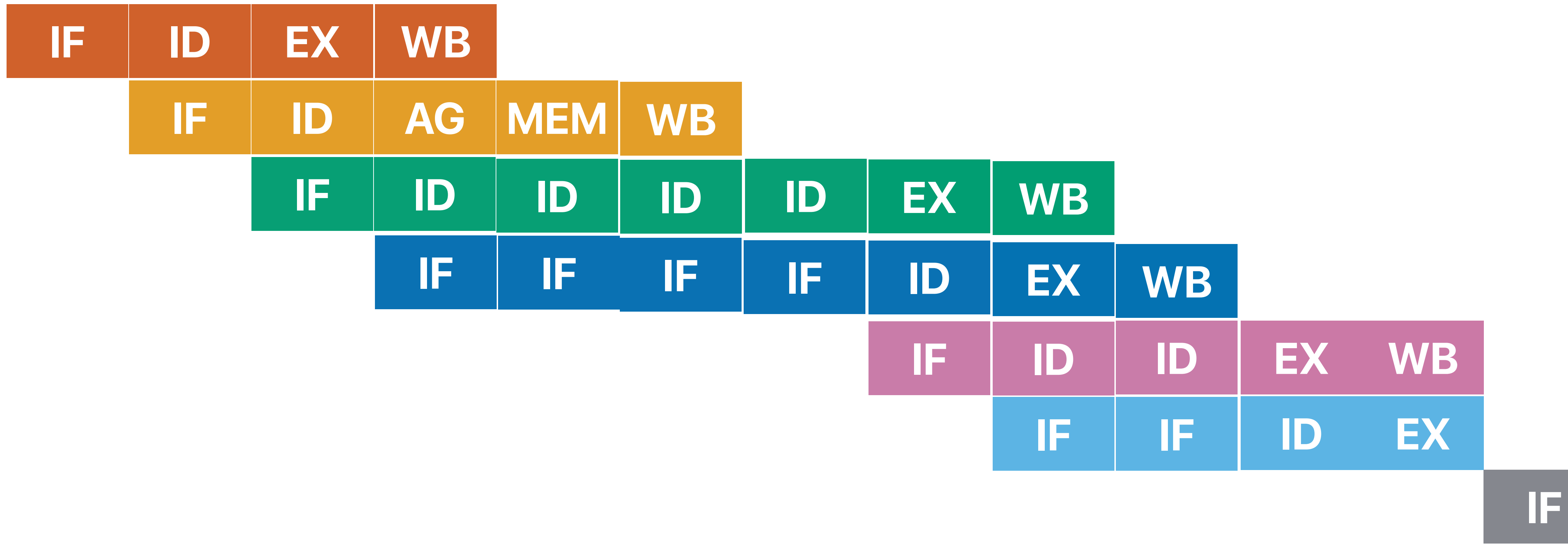
- Pipelining helps to improve the throughput of processors
  - Allowing shorter cycle time as each cycle only make progress for part of an instruction
  - Different pipeline stages work on different instructions concurrently
  - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time
- Pipeline hazards prevent us from reaching the theoretical CPI
  - Structural hazards
  - Control hazards
  - Data 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
- Disable the pipeline register update for later instructions
- The stalled instructions still have the same input from the pipeline registers

```
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
```



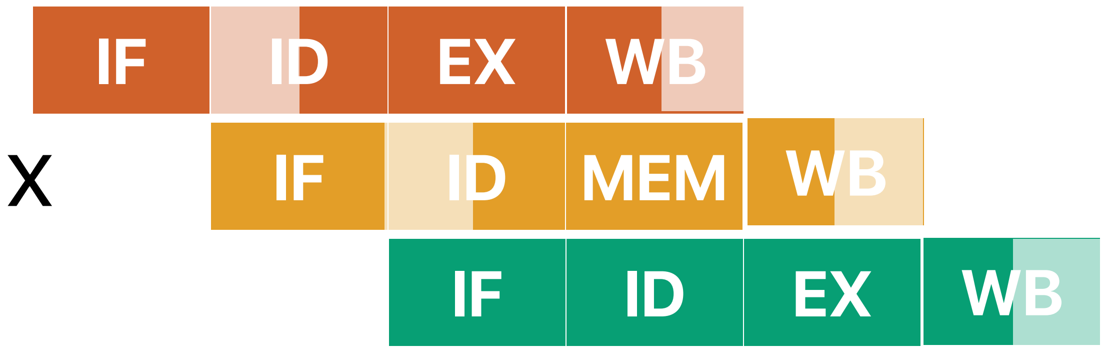
Slow! — 4 additional cycles

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!

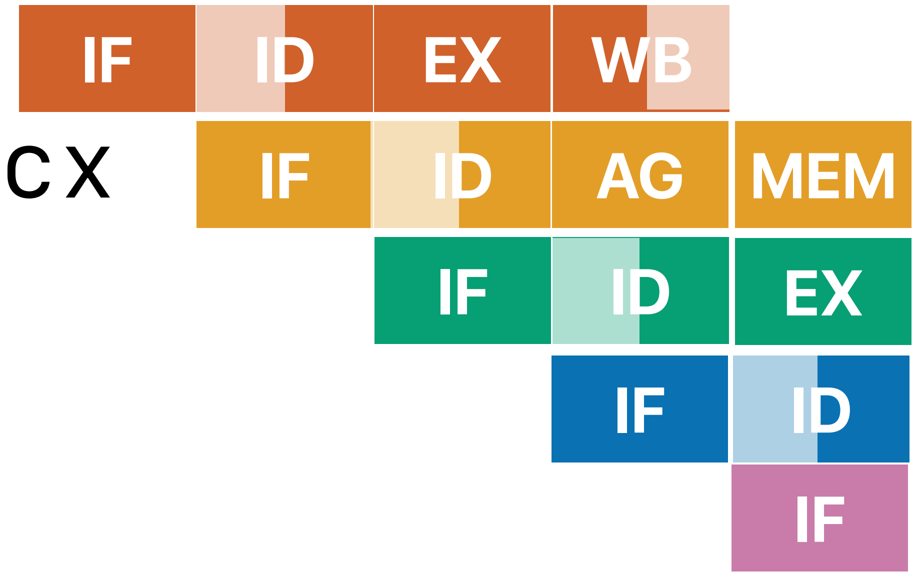
```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
```

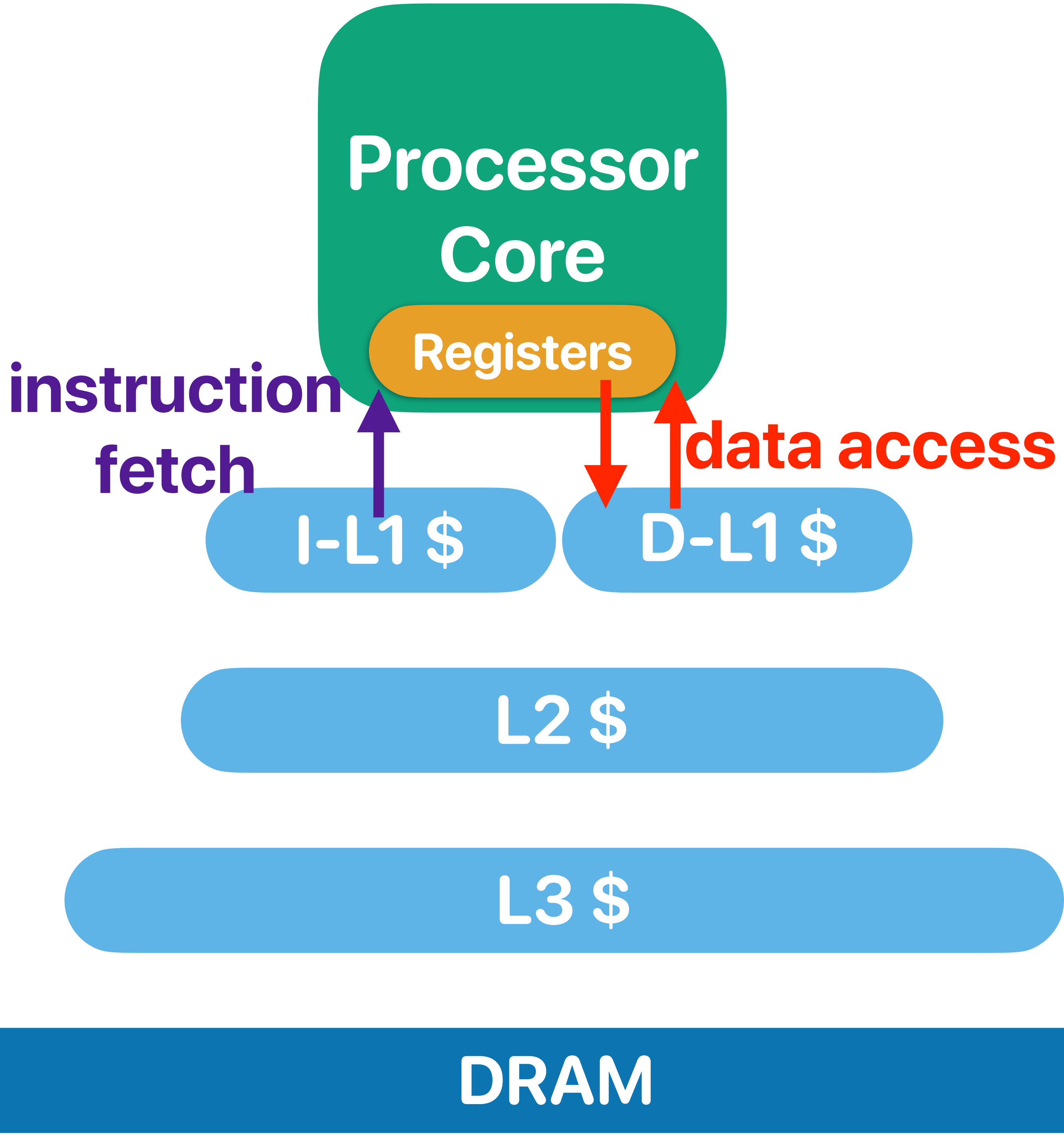


How to with the conflicts between MEM and IF?

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

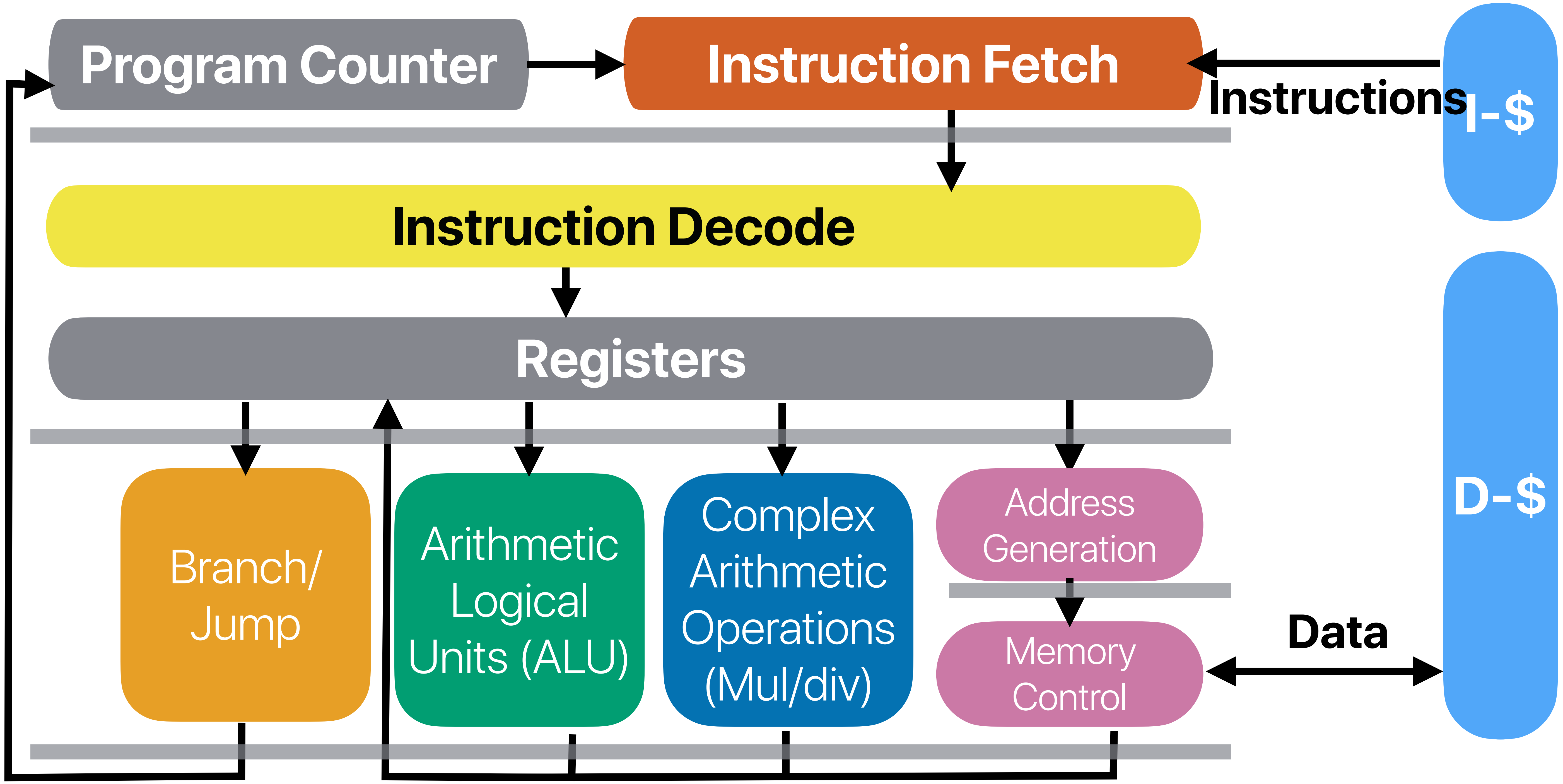
```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
⑤ cmpq %rdx, %rdi
```





"Split L1" cache!

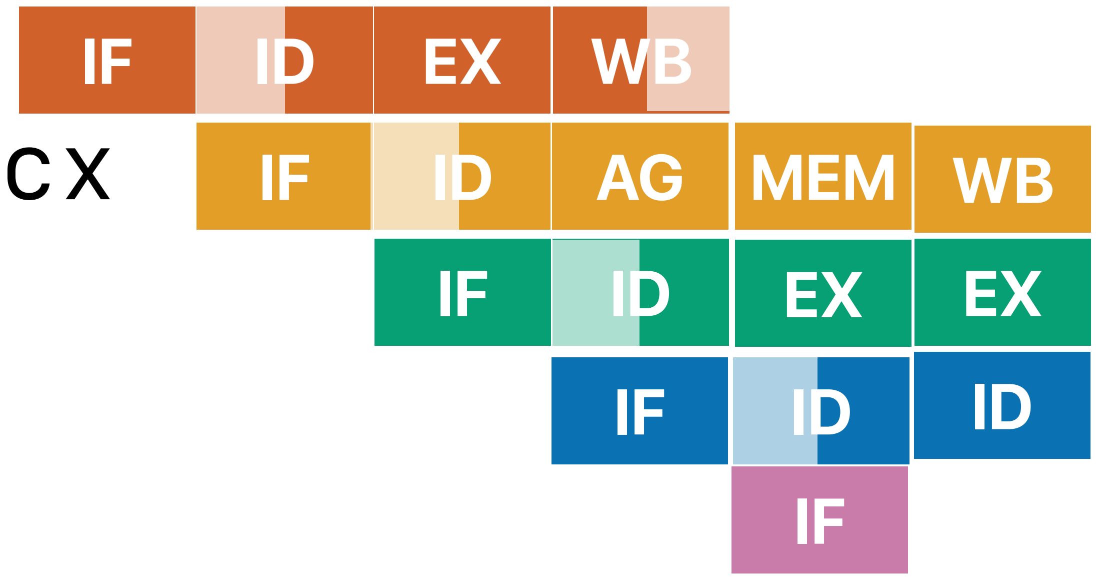
Split L1-$


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

Both (2) and (3) want to "WB"

- 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
⑤ cmpq %rdx, %rdi
```

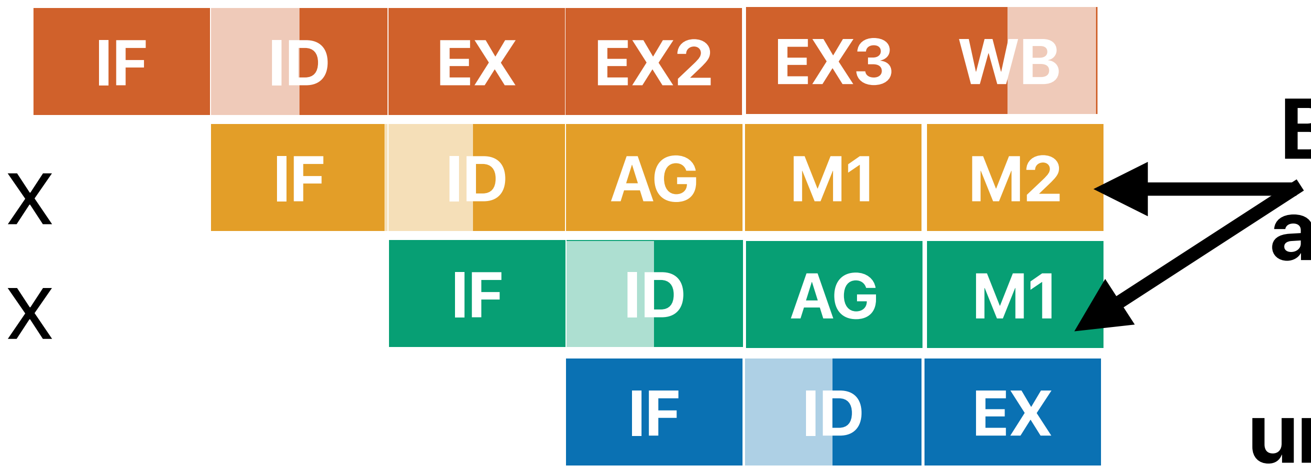


(3) has to stall

What if we've back-to-back memory instructions?

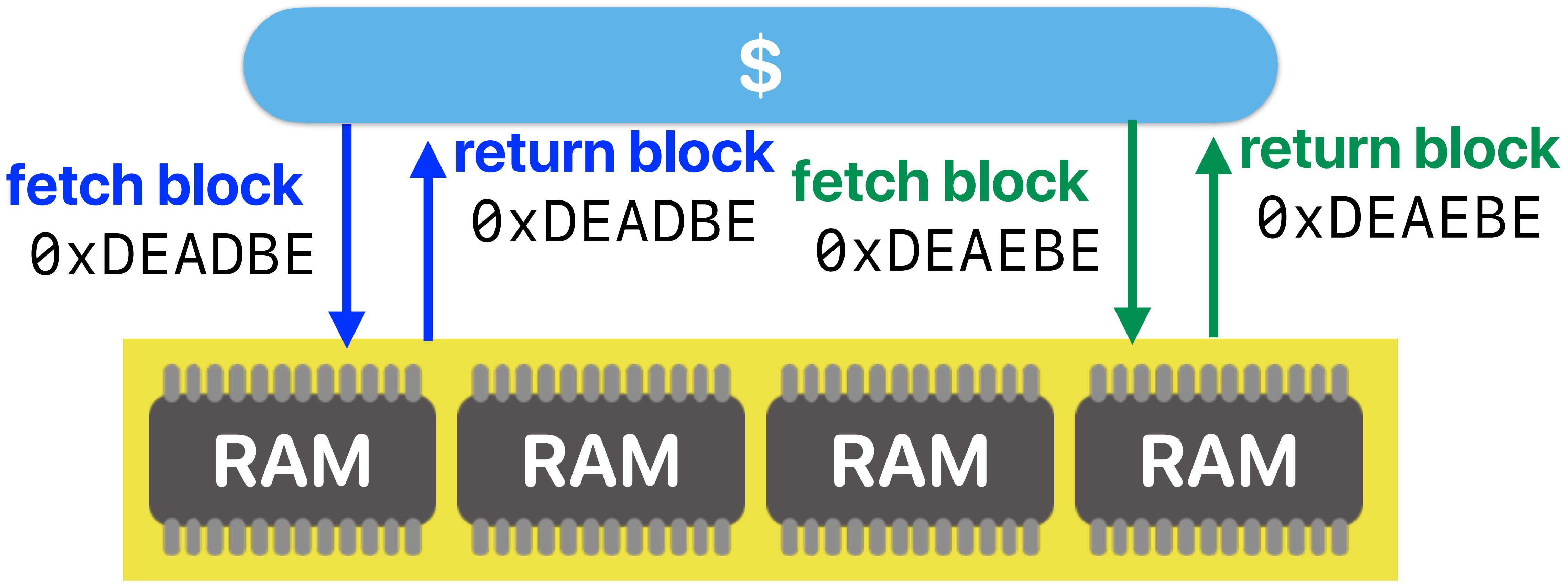
- Every instruction needs to go through exactly the same number of stages

```
① xorl %eax, %eax
② movl (%rdi), %rcx
③ movl (%rcx), %rax
④ addq $8, %rdi
```

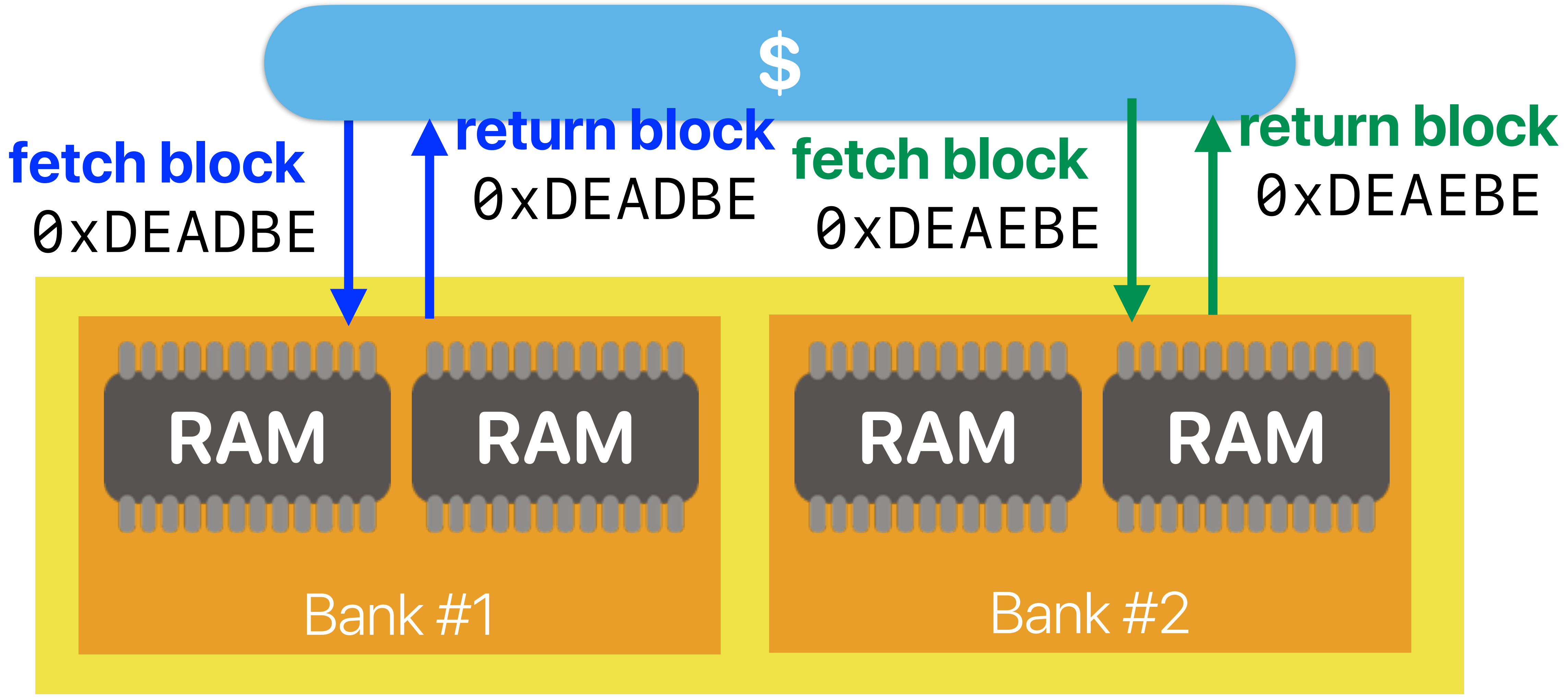


Both (2) and (3) are attempting use data memory unit and it's occupied by (2)

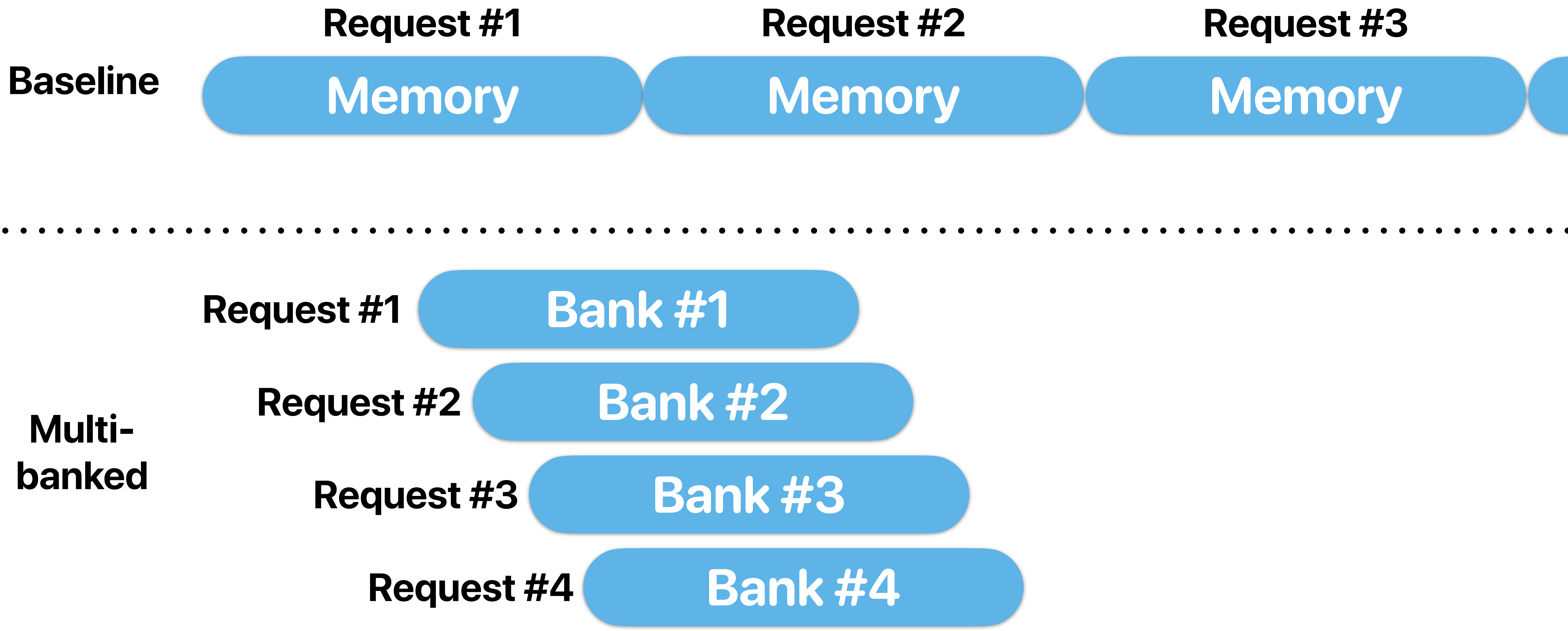
Blocking cache



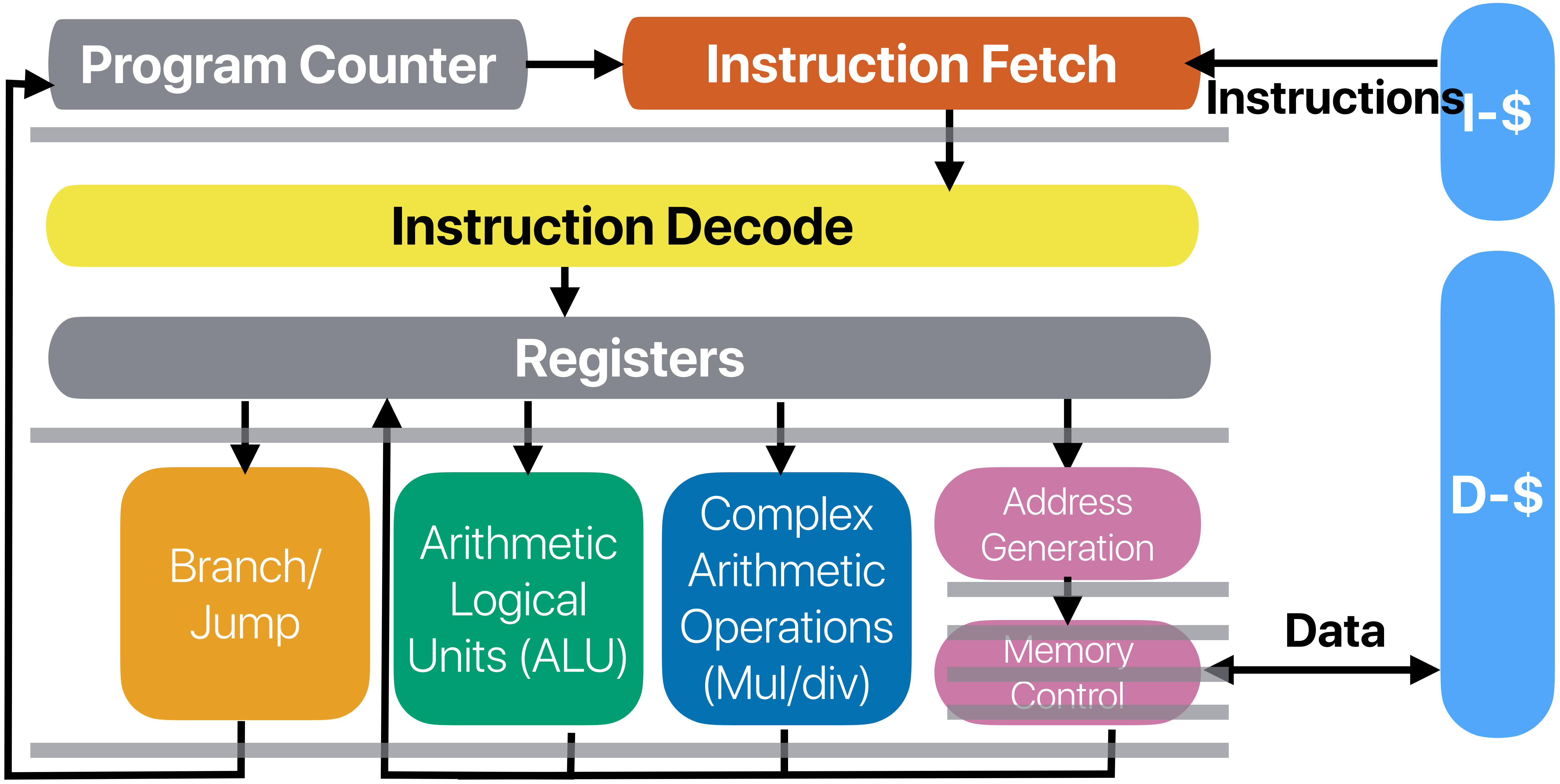
Multibanks & non-blocking caches



Pipelined access and multi-banked caches



Split L1-$ + Multibanked, non-blocking cache



Takeaways: pipeline processors

- Pipelining helps to improve the throughput of processors
  - Allowing shorter cycle time as each cycle only make progress for part of an instruction
  - Different pipeline stages work on different instructions concurrently
  - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time
- Pipeline hazards prevent us from reaching the theoretical CPI
  - Structural hazards
  - Control hazards
  - Data hazards
- The most efficient approach to address structural hazards is to make the hardware available to support concurrent execution
  - Register file
  - Split caches
  - Multibanked, non-blocking cache