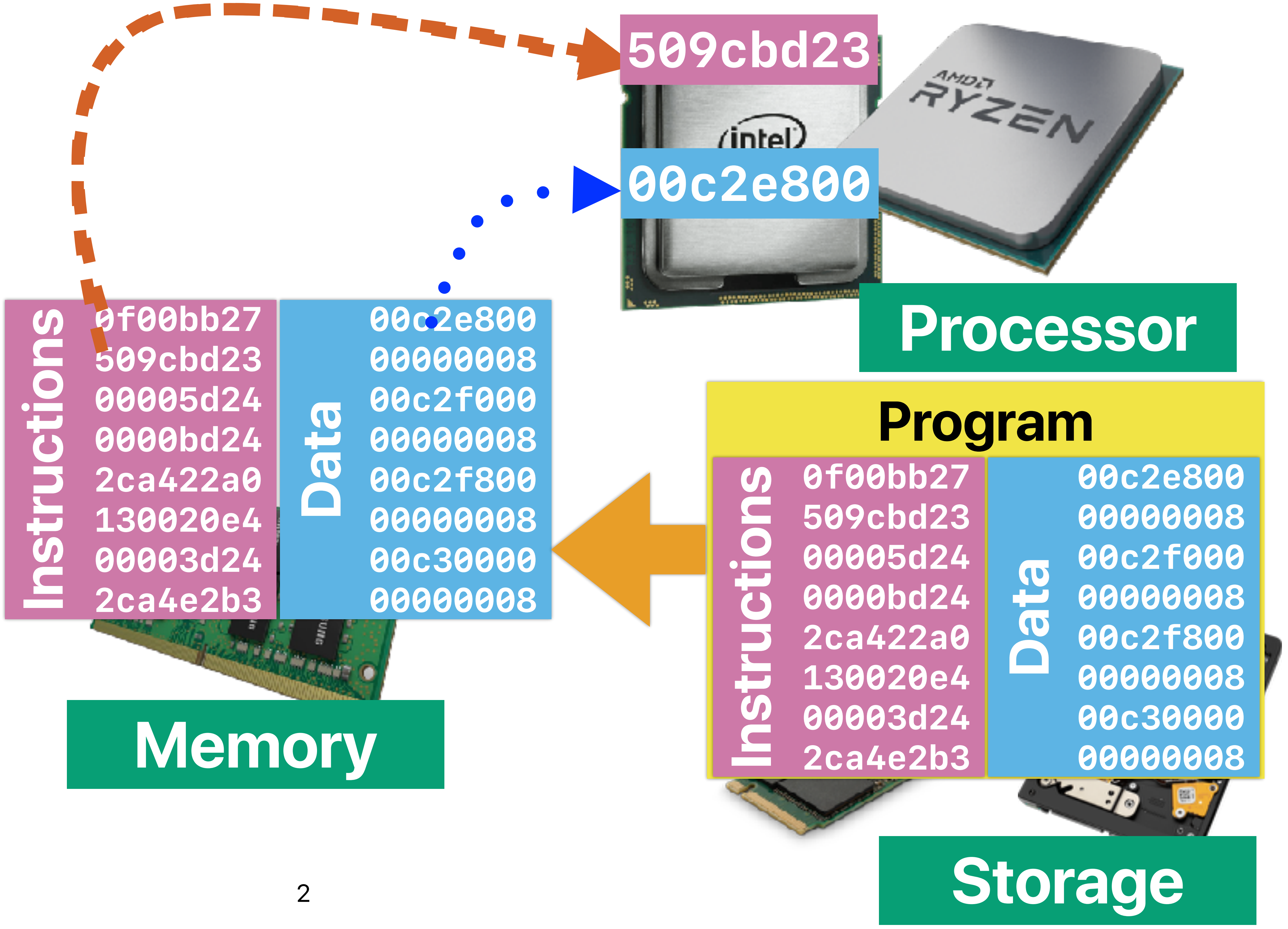
# Basic Pipelined Processor

Hung-Wei Tseng

# Recap: von Neumman Architecture



## Outline

- Basic Pipeline Processor Design
- Pipeline Hazards
    - Structural Hazards
    - Control Hazards
    - Data Hazards
- Dynamic Branch Predictions

# 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 ++;
    }
}
```

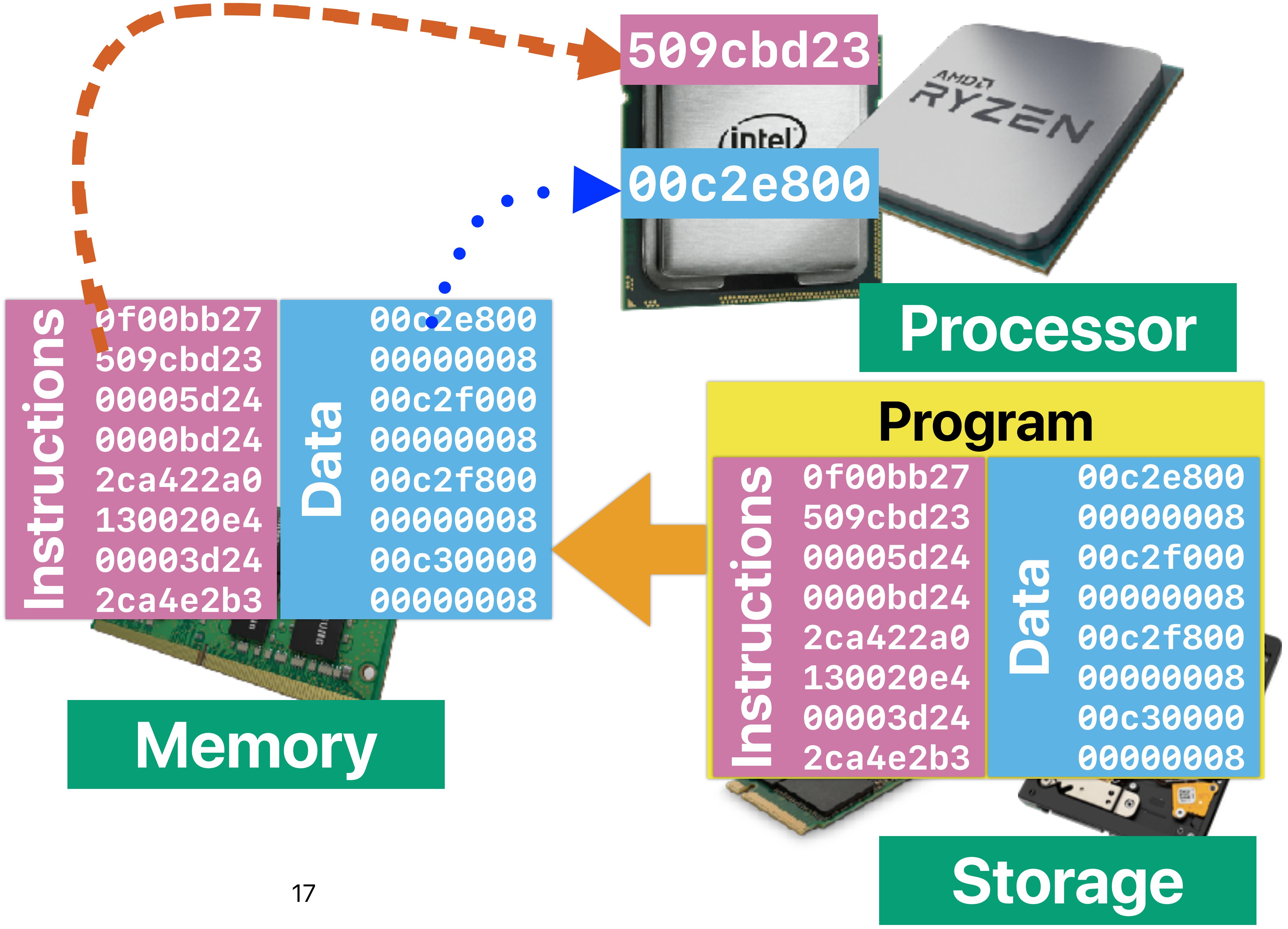
## Recap: Adding a sort...

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

for (unsigned c = 0; c < arraySize*1000; ++c) {
        if (data[c%arraySize] >= INT_MAX/2)
            sum ++;
    }
}
```

## Basic Processor Design

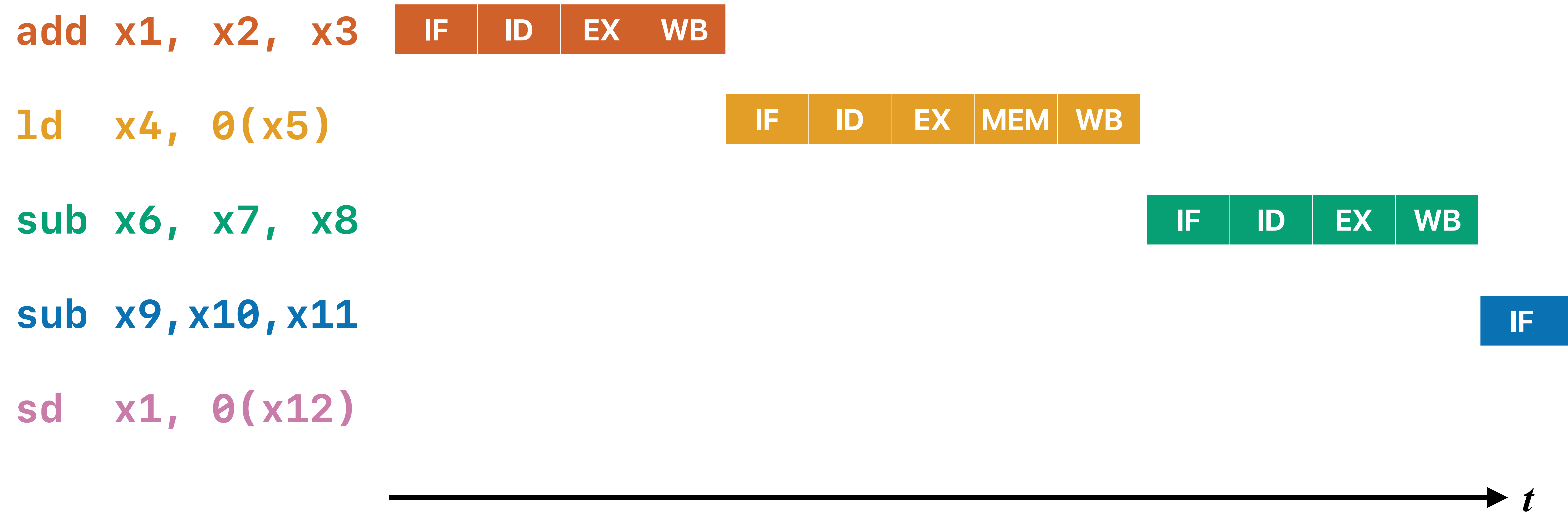
# von Neumman Architecture



# Tasks in RISC-V ISA

- 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

# Simple implementation w/o branch



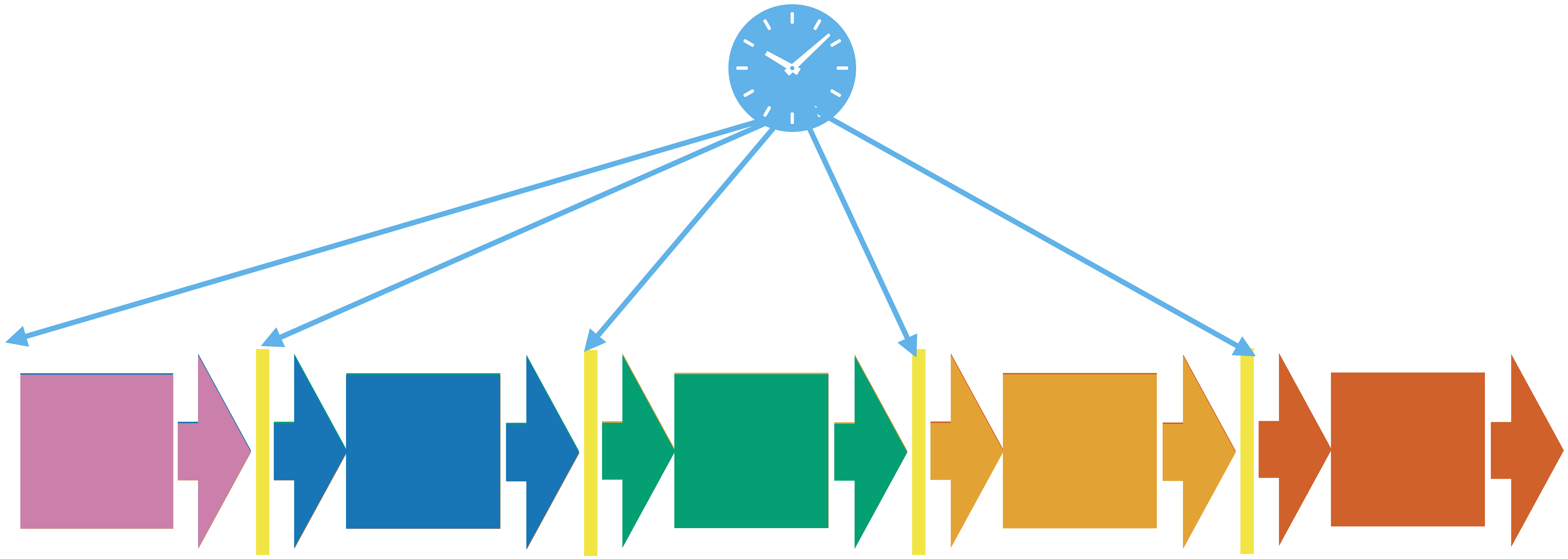
## Pipelining

# Pipelining

## Pipelining

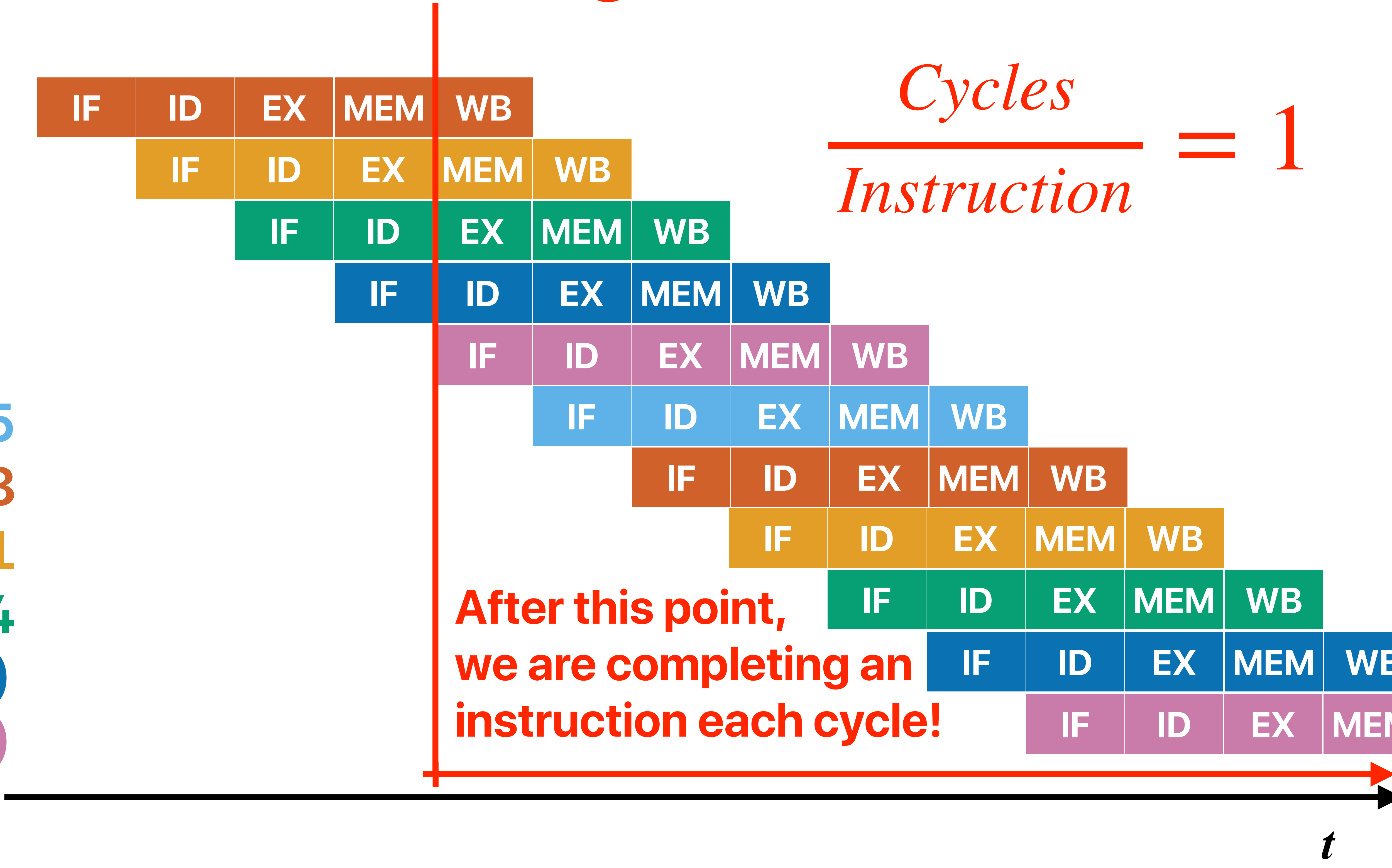
- Different parts of the processor works on different instructions simultaneously
- A clock signal controls and synchronize the beginning and the end of each part of the work
- A pipeline register between different parts of the processor to keep intermediate results necessary for the upcoming work

## Pipelining



# Pipelining

```
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)
```


IF
ID
EX
MEM
WB
After this point, we are completing an instruction each cycle!
t

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

## Pipeline hazards

## 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

# Can we get them right?

- Given a simple pipelined RISC-V processor that we discussed so far, how many of the following code snippets can be executed with expected outcome?

|   | I | II | III | IV |
|---|---|---|---|---|
| a | add x1, x2, x3 | add x1, x2, x3 | add x1, x2, x3 | add x1, x2, x3 |
| b | ld x4, 0(**x1**) | ld x4, 0(x5) | ld x4, 0(x5) | ld x4, 0(x5) |
| c | sub x6, x7, x8 | sub x6, x7, x8 | **bne x0, x7, L** | sub x6, x7, x8 |
| d | sub x9,x10,x11 | sub x9, **x1**, x10 | sub x9,x10,x11 | sub x9,x10,x11 |
| e | sd x1, 0(x12) | sd x11, 0(x12) | sd x1, 0(x12) | sd x1, 0(x12) |

A. 0
[✓ B. 1]
C. 2
D. 3
E. 4

**b cannot get x1 produced by a before WB**

**Data Hazard**

**both a and d are accessing x1 at the 5th cycle**

**Structural Hazard**

**We don't know if d & e will be executed or not until c finishes**

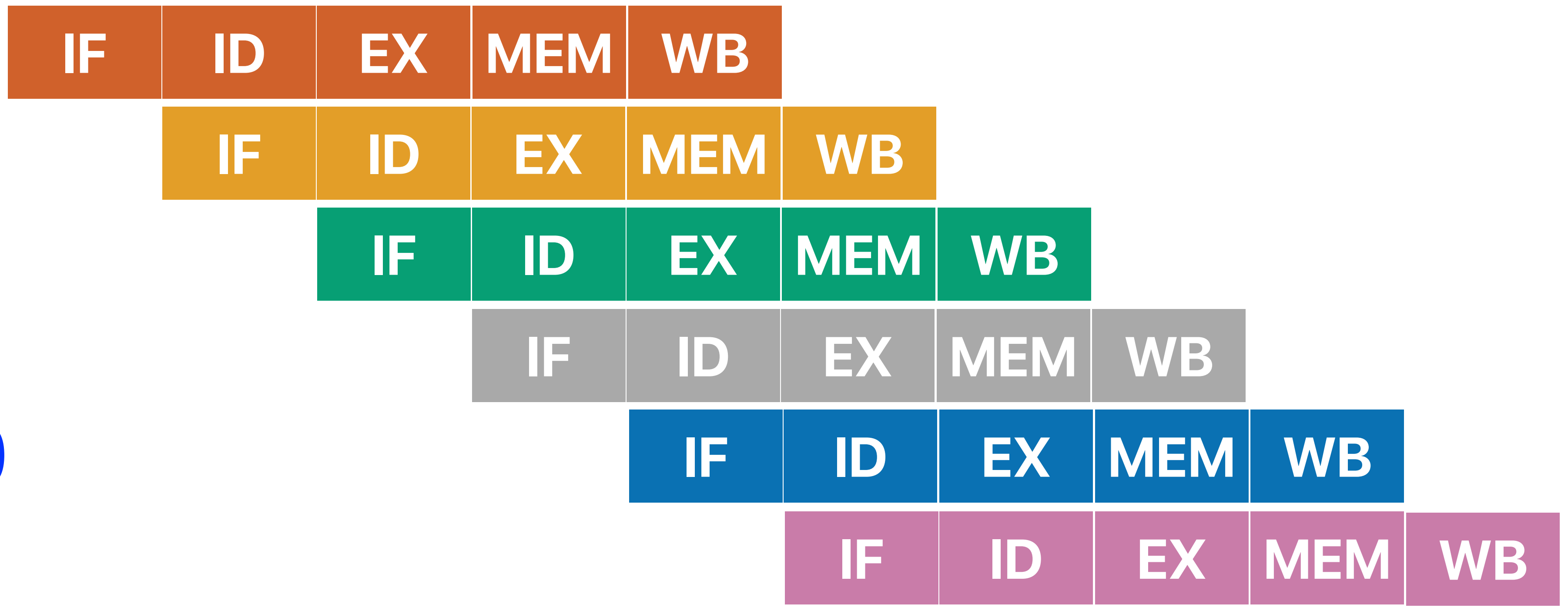
**Control Hazard**

## Structural Hazards

## Dealing with the conflicts between ID/WB

- The same register cannot be read/written at the same cycle
- Solution: insert no-ops (e.g, `add x0,x0,x0`) between them
- Drawback
  - If the number of pipeline stages changes, the code won't work
  - Slow

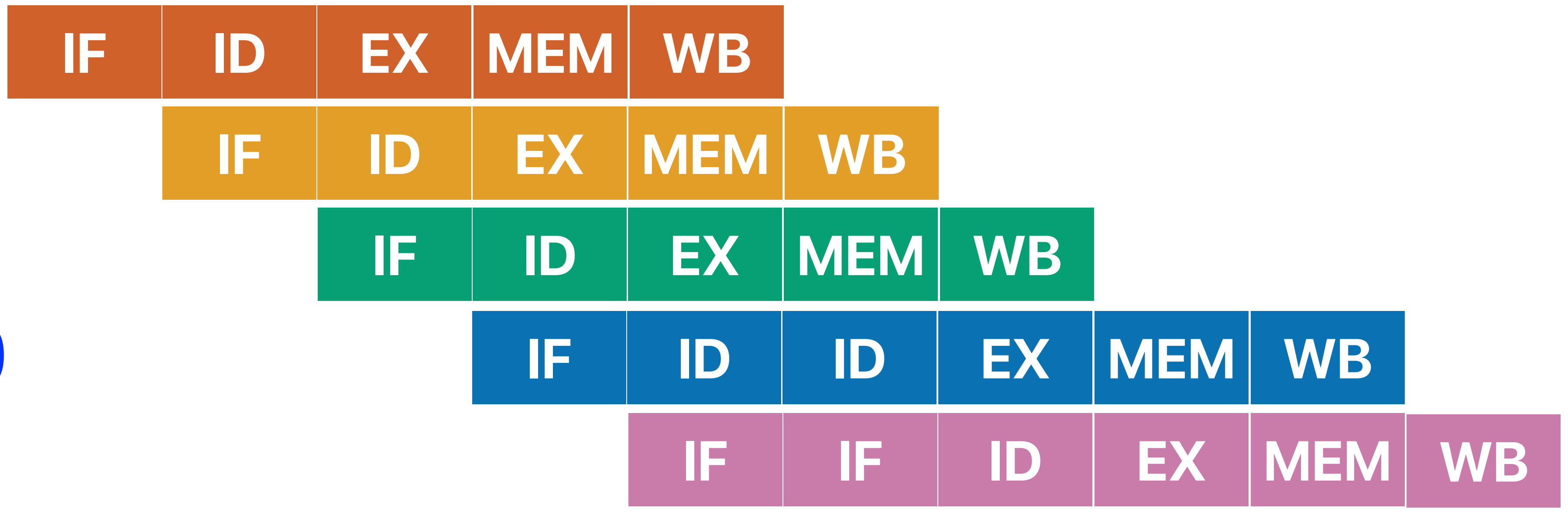
```
add x1, x2, x3
ld  x4, 0(x5)
sub x6, x7, x8
add x0, x0, x0
sub x9, x1, x10
sd  x11, 0(x12)
```



# Dealing with the conflicts between ID/WB

- The same register cannot be read/written at the same cycle
- Solution: stall the later instruction, allowing the write to present the change in the register and the later can get the desired value
- Drawback: slow

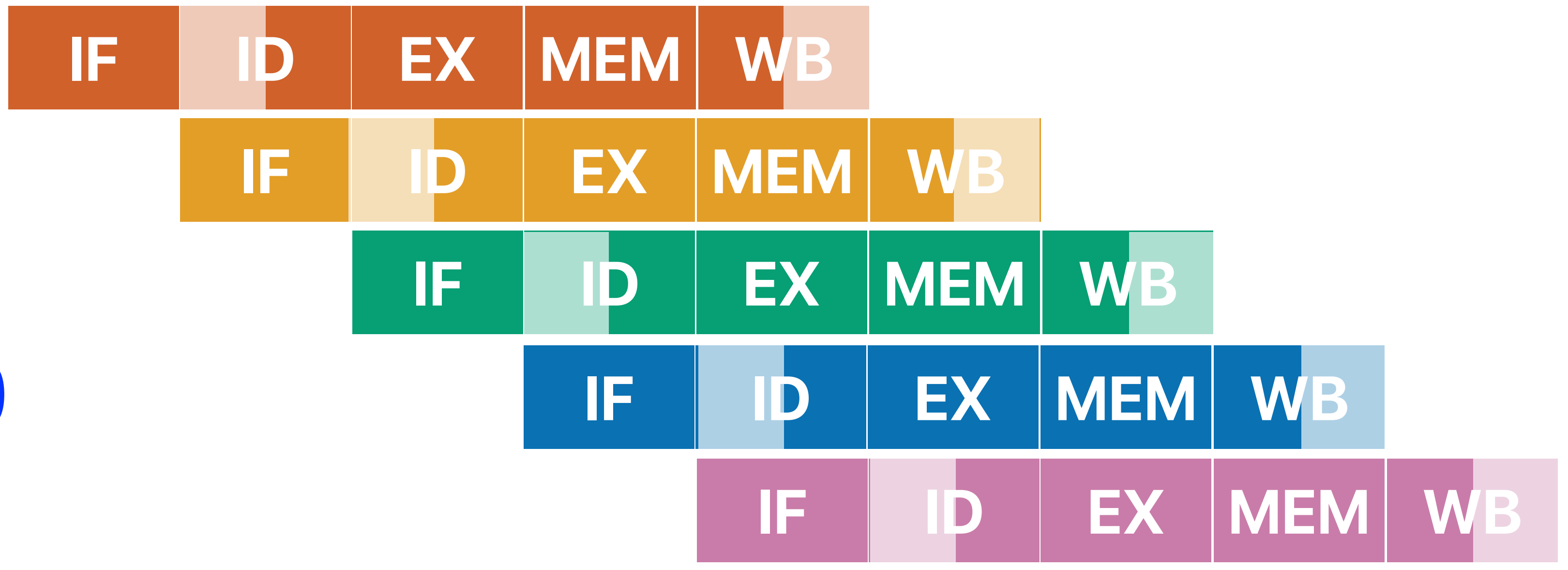
```
add x1, x2, x3
ld  x4, 0(x5)
sub x6, x7, x8
sub x9, x1, x10
sd  x11, 0(x12)
```



# 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!

```
add x1, x2, x3
ld  x4, 0(x5)
sub x6, x7, x8
sub x9, x1, x10
sd  x11, 0(x12)
```



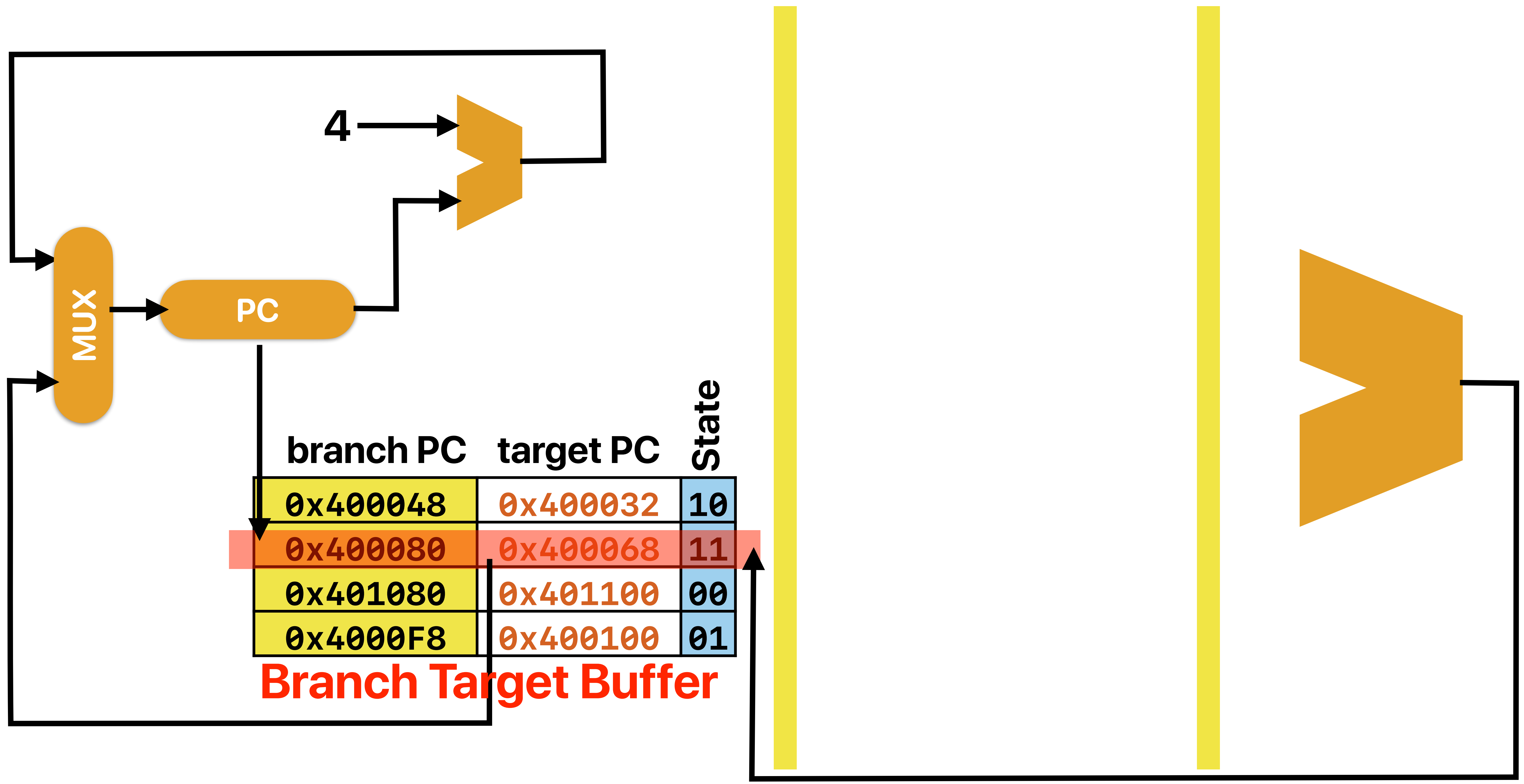
## Structural Hazards

- 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
  - All instructions need to go through 5 stages

# Control Hazards

## Dynamic Branch Prediction

# A basic dynamic branch predictor

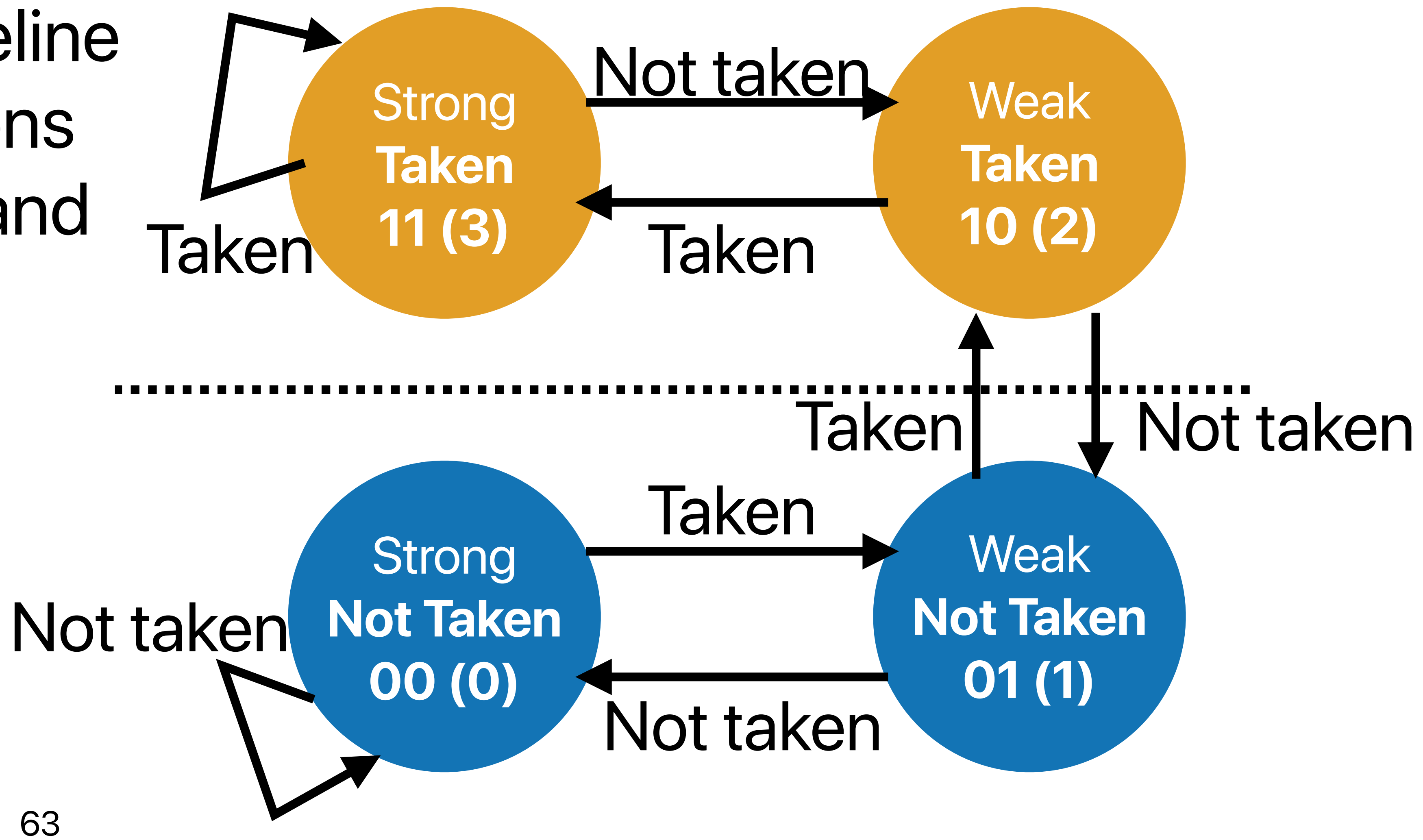


| branch PC | target PC | State |
|---|---|---|
| 0x400048 | 0x400032 | 10 |
| 0x400080 | 0x400068 | 11 |
| 0x401080 | 0x401100 | 00 |
| 0x4000F8 | 0x400100 | 01 |

# 2-bit/Bimodal local predictor

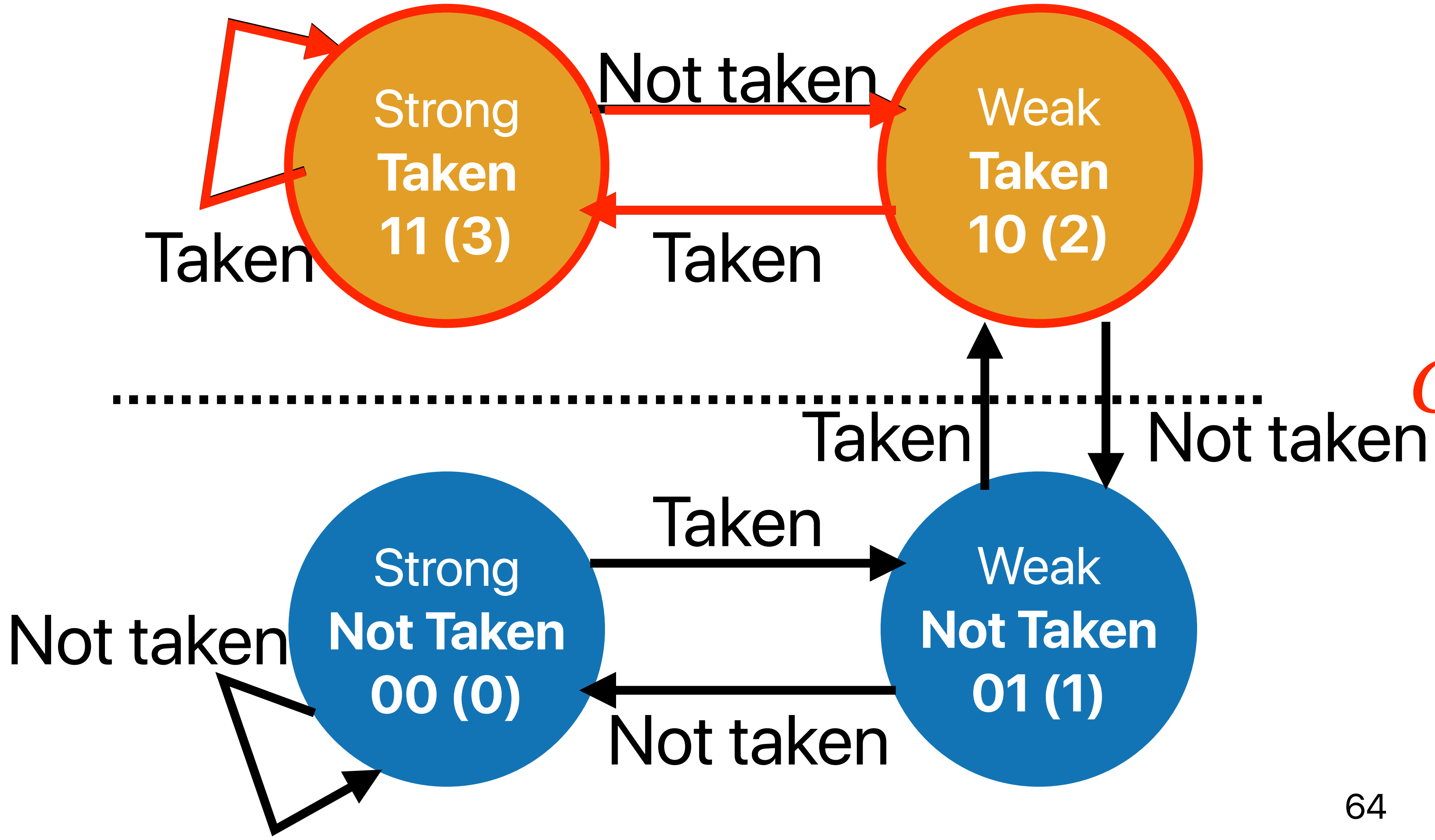
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

| | branch PC | target PC | State |
|---|---|---|---|
| | 0x400048 | 0x400032 | 10 |
| **Predict Taken** | 0x400080 | 0x400068 | 11 |
| | 0x401080 | 0x401100 | 00 |
| | 0x4000F8 | 0x400100 | 01 |



# 2-bit local predictor

```
i = 0;
do {
      sum += a[i];
} while(++i < 10);
```



| i | state | predict | actual |
|---|---|---|---|
| 1 | 10 | T | T |
| 2 | 11 | T | T |
| 3 | 11 | T | T |
| 4-9 | 11 | T | T |
| 10 | 11 | T | NT |

**90% accuracy!**

$$CPI_{average} = 1 + 20\% \times 10\% \times 2 = 1.04$$