



EXPLAINED:

---

# MELTDOWN & SPECTRE





EXPLAINED:

# MELTDOWN & SPECTRE

*for people*

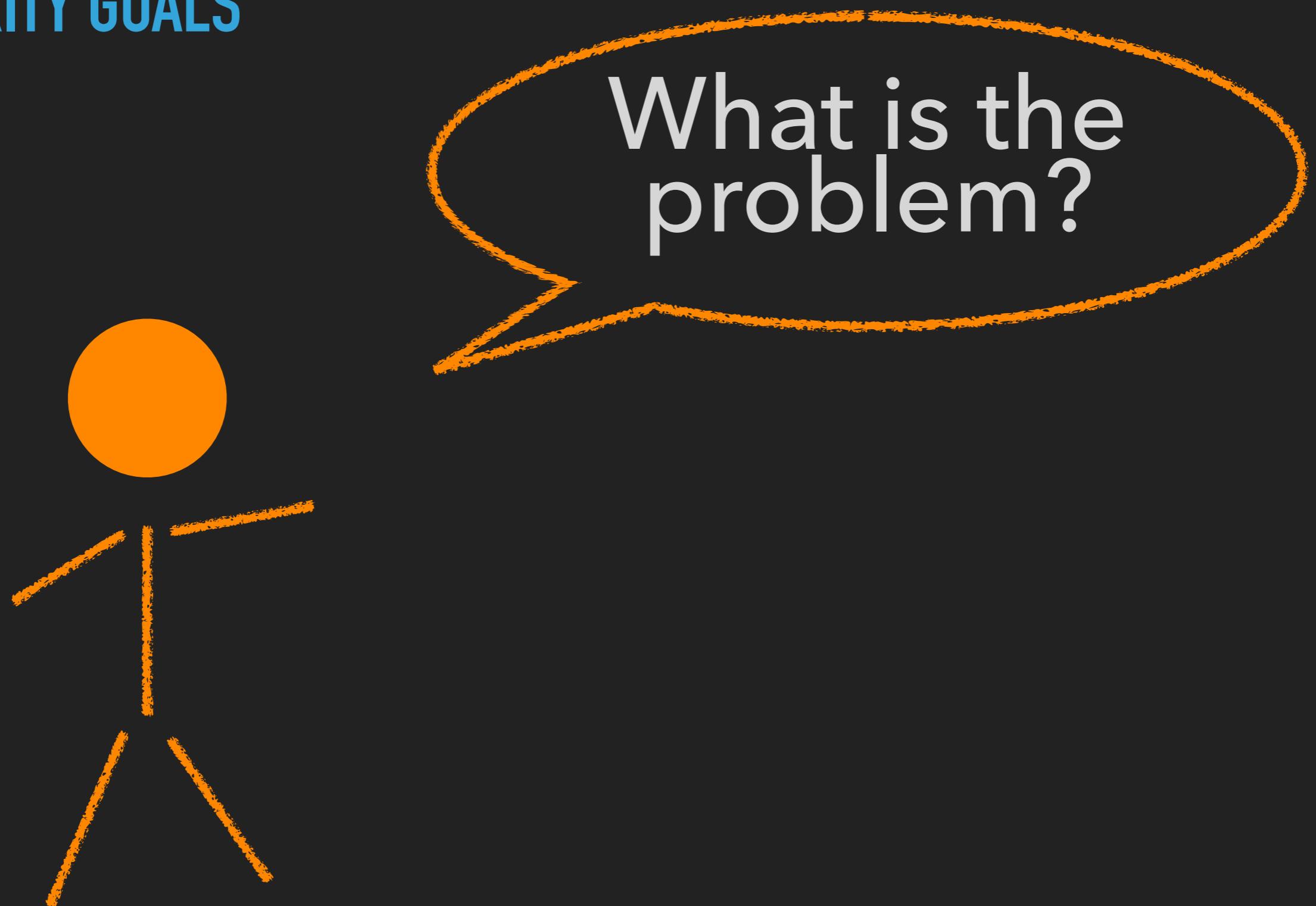


# WHO AM I?

- ▶ Jens Neuhalfen
- ▶ Age: Forty something
- ▶ IT since: ever
- ▶ Skills: Bridge between IT and business, IT-Security Management, writing software
- ▶ <https://github.com/neuhalje>



## SECURITY GOALS



A PROCESS MUST BE  
ISOLATED (PROTECTED) FROM  
OTHER PROCESSES!

You

# SECURITY GOALS: APP ISOLATION

## SECURITY GOALS: APP ISOLATION

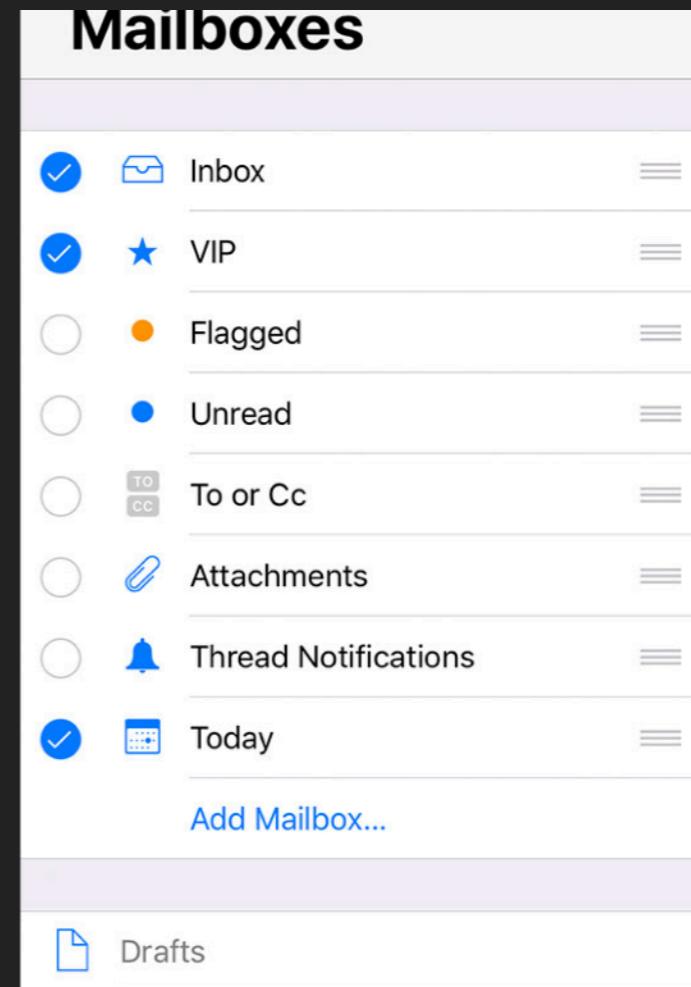


You don't want  
*this*

## SECURITY GOALS: APP ISOLATION

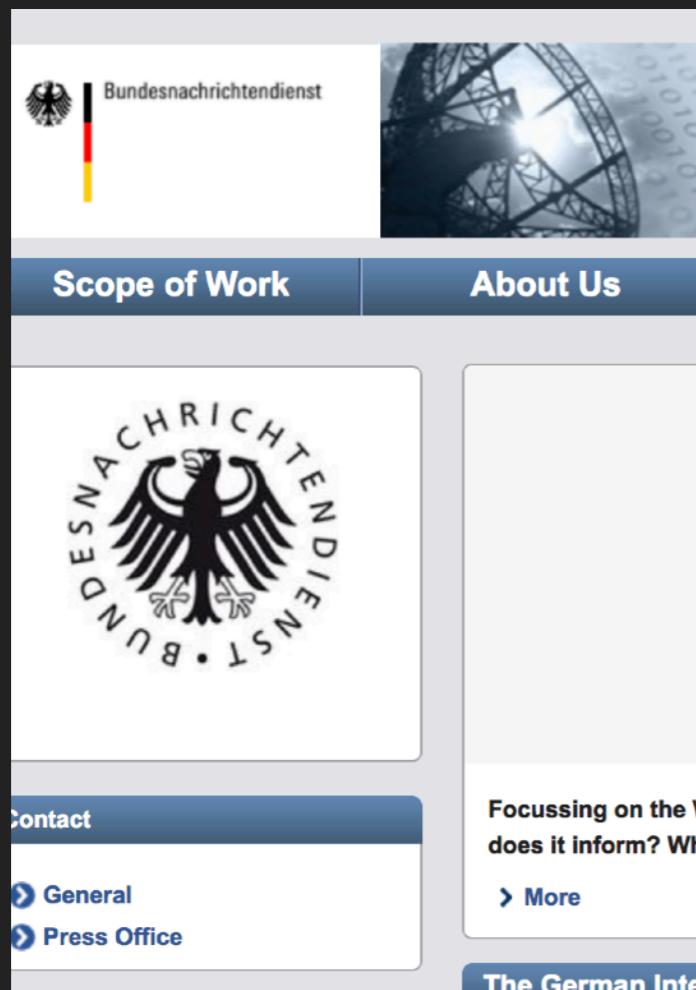


You don't want  
*this*



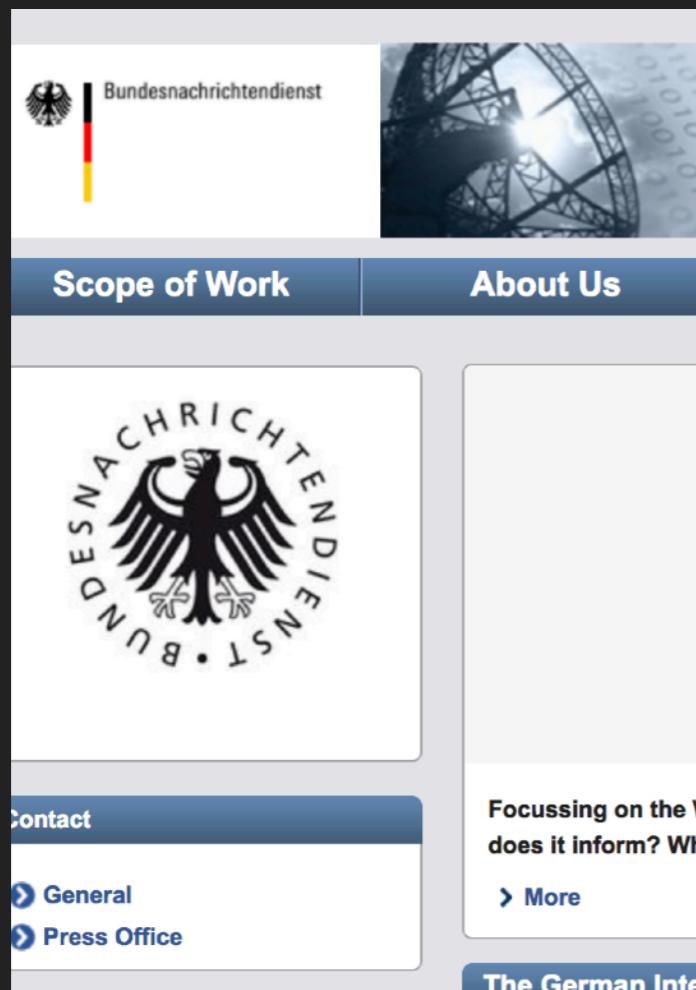
To read  
*that*

## SECURITY GOALS: BROWSER TAB ISOLATION

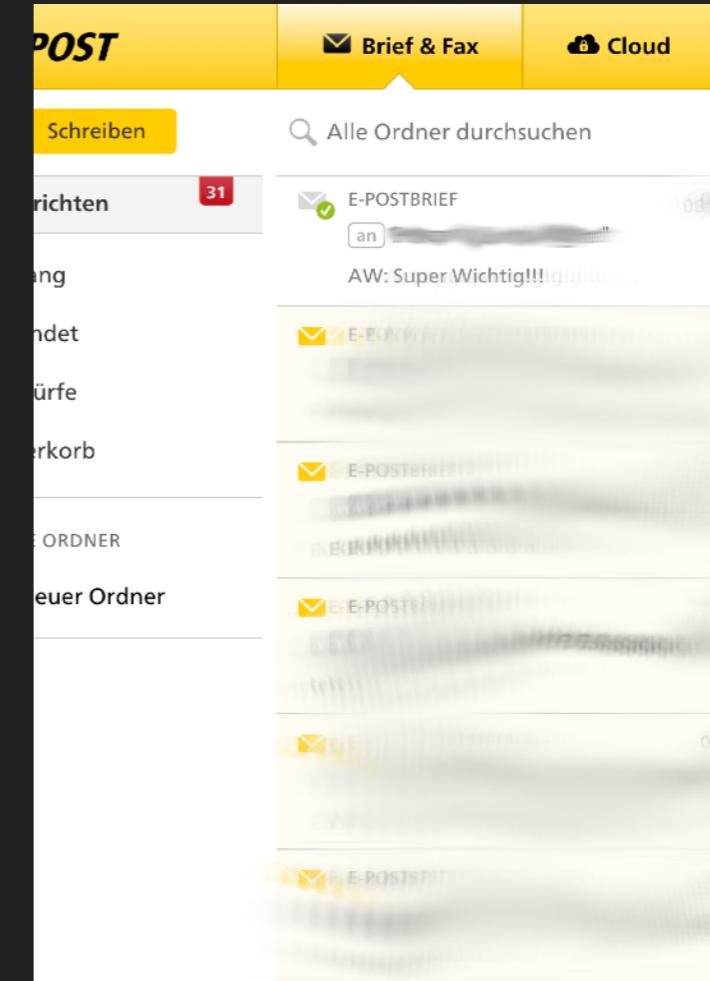


You don't want  
*this*

# SECURITY GOALS: BROWSER TAB ISOLATION



You don't want  
*this*



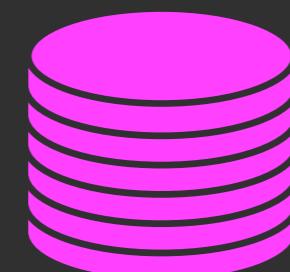
# To read *that*

## SECURITY GOALS: CLOUD ISOLATION



You don't want  
*this*

## SECURITY GOALS: CLOUD ISOLATION



You don't want  
*this*

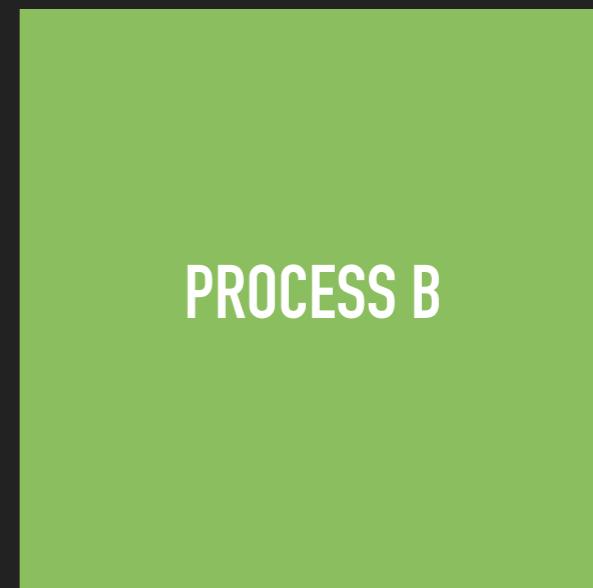
To read  
*that*

## SECURITY GOALS: MEMORY ISOLATION



You don't want  
*this*

## SECURITY GOALS: MEMORY ISOLATION



You don't want  
*this*

To read  
*that*

## SECURITY GOALS: MEMORY ISOLATION

The more a system relies on process isolation to achieve its security goal, the more critical Meltdown and Spectre are.

PROCESS A

PROCESS B

You don't want  
*this*

To read  
*that*

## SECURITY GOALS: MEMORY ISOLATION

The more a system relies on process isolation to achieve its security goal, the more critical Meltdown and Spectre are.

PROCESS A

PROCESS B

Fortunately an attacker must be able to execute his code on a system to exploit the Meltdown and Spectre attacks.

You don't  
this  
read  
hat



MELTDOWN AND  
SPECTRE

---

# MANAGEMENT SUMMARY

## MELTDOWN



- ▶ **Result:** Programs can read memory it should not
- ▶ **Affects:** All modern CPU/OS
- ▶ **Vector:** Uses *out of order execution* to read forbidden memory and *cache timing* as side channel to exfiltrate data
- ▶ **How bad:** Bad
- ▶ **Fixes:** Needs changes in CPU and/or OS patches. Modest (X%) to severe (XX%) performance impact, higher on older CPU. Performance impact varies and depends on CPU and workload type.

## SPECTRE



- ▶ **Result:** Programs can read all memory
- ▶ **Affects:** All modern CPU/OS
- ▶ **Vector:** Uses *speculative execution* to read forbidden memory and *cache timing* to exfiltrate data
- ▶ **How bad:** Very bad
- ▶ **Fixes:** Needs changes in CPU and/or changes in programs. Performance impact varies and depends on CPU and workload type.

## THREAT-O-METER

*Threat - O - Meter*

**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER

**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

# MELTDOWN & SPECTRE FOR NORMAL PEOPLE

## THREAT-O-METER

Public clouds run code of many untrusted parties which makes them very vulnerable.



### LOW RISK

Exploit unlikely or running untrusted code already worst case

### MEDIUM RISK

Exploit possible but needs another successful attack to run attackers code

### HIGH RISK

Exploit possible and runs untrusted code "by design"

## THREAT-O-METER



**DATABASE  
SERVER**



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



### LOW RISK

Exploit unlikely or running untrusted code already worst case

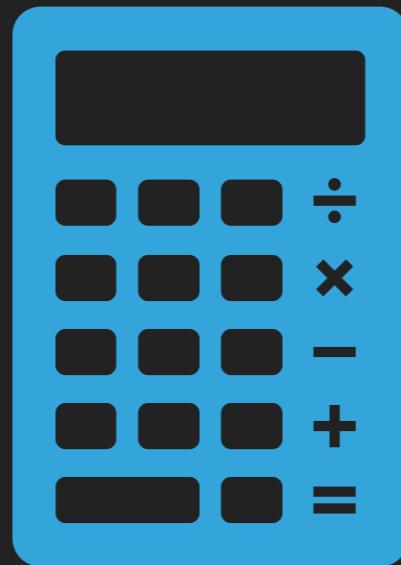
### MEDIUM RISK

Exploit possible needs another successful attack  
attackers co

Databases are often protected from the internet and are accessed only by application servers.

Running untrusted code on a database is often already the worst case scenario. Patching against Meltdown/Spectre would only marginally increase security.

## THREAT-O-METER



MAILSERVER



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

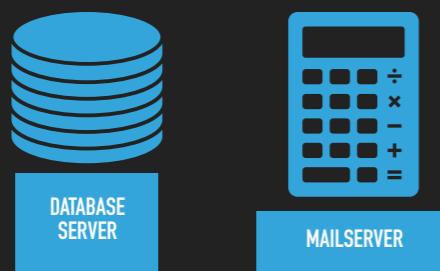
**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



MAILSERVER

**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible  
needs another  
successful attack to run  
attackers code

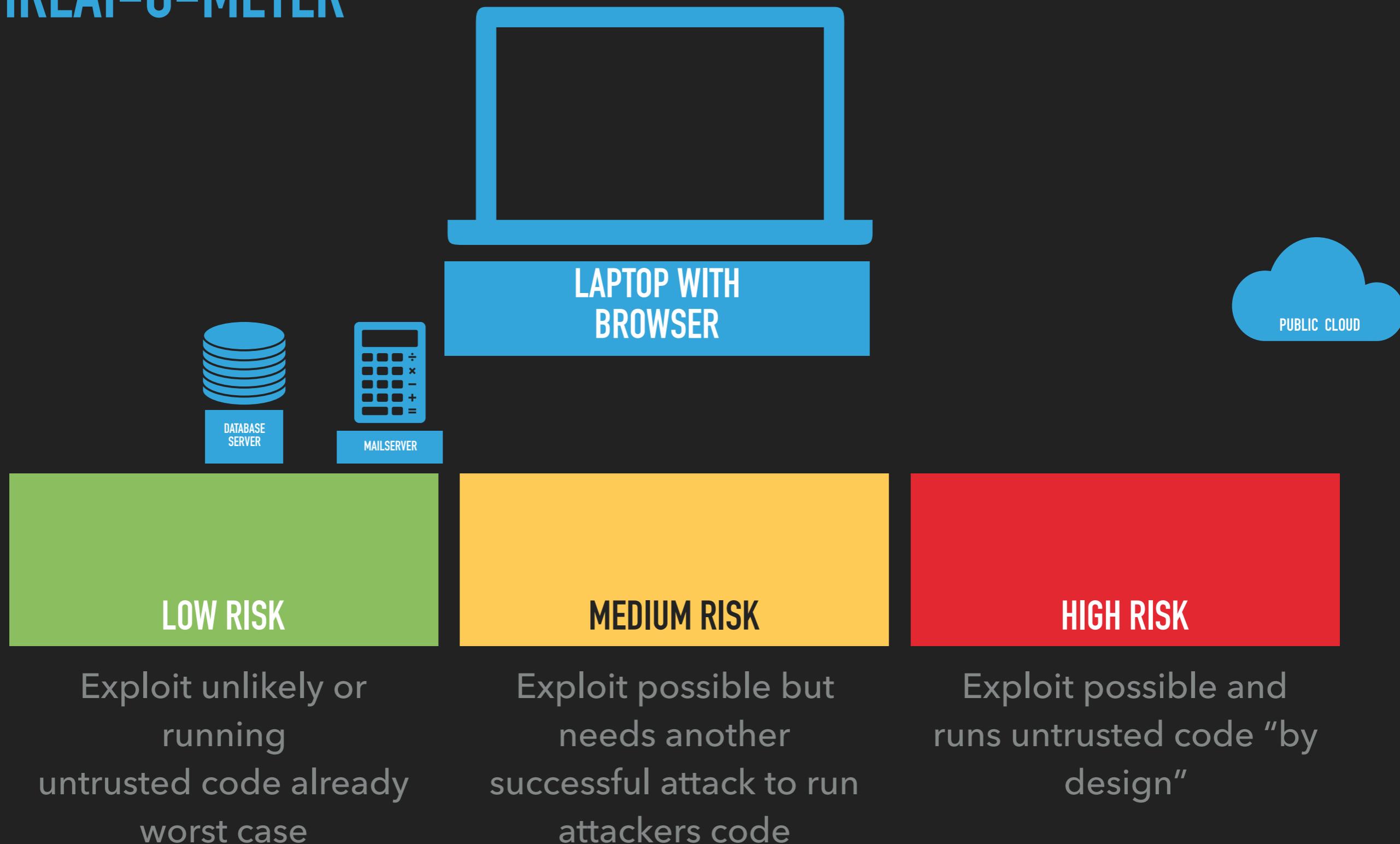
Mailserver are exposed to the internet but have been proven to be very robust to “remote code execution” attacks.

Also a code execution is already the worst case.

Arguably mail servers can be placed in “medium” due to their exposure to the internet.

“design”

## THREAT-O-METER



## THREAT-O-METER



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Laptops/desktop systems with browsers are very vulnerable because they execute untrusted code in the form of JavaScript from websites.



Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



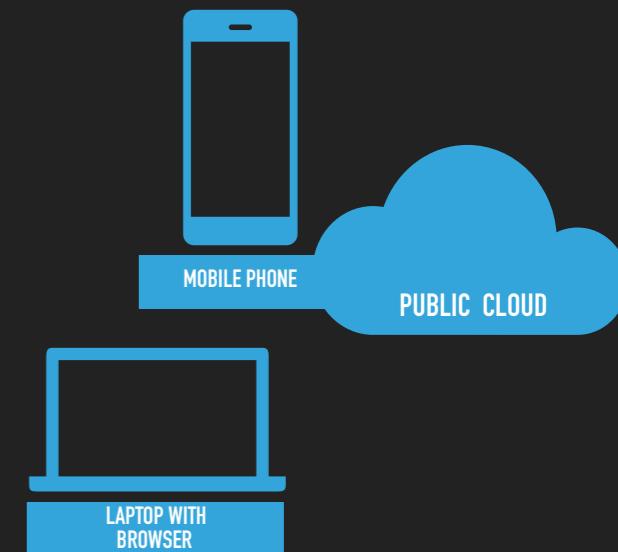
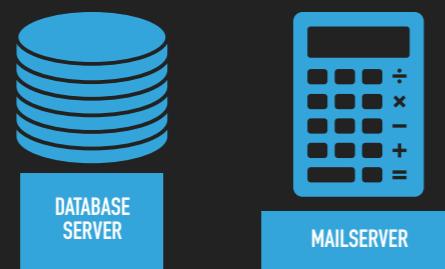
Exploit unlikely or  
running  
untrusted code already  
worst case

Exploit possible but  
needs another  
successful attack to run  
attackers code

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER

Mobile phones run apps and websites (JavaScript).



### LOW RISK

Exploit unlikely or running untrusted code already worst case

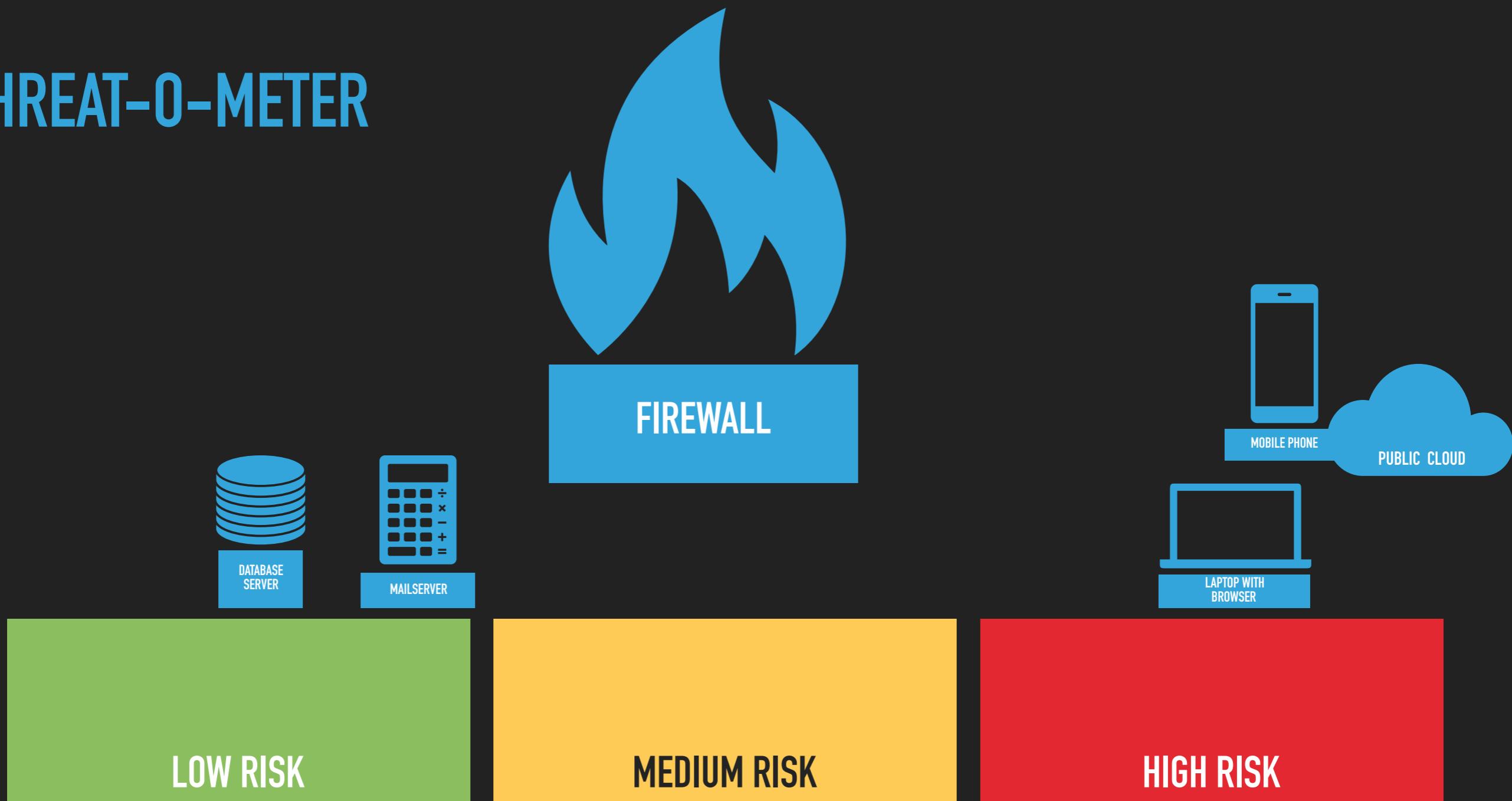
### MEDIUM RISK

Exploit possible but needs another successful attack to run attackers code

### HIGH RISK

Exploit possible and runs untrusted code "by design"

## THREAT-O-METER

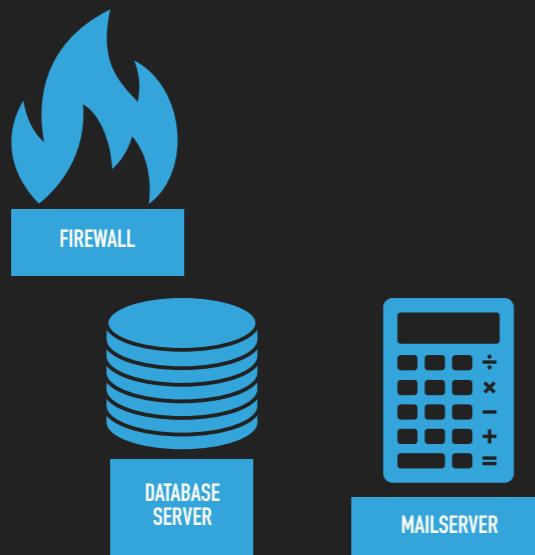


**LOW RISK**  
Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**  
Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**  
Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possibl  
needs another  
successful attack  
attackers co

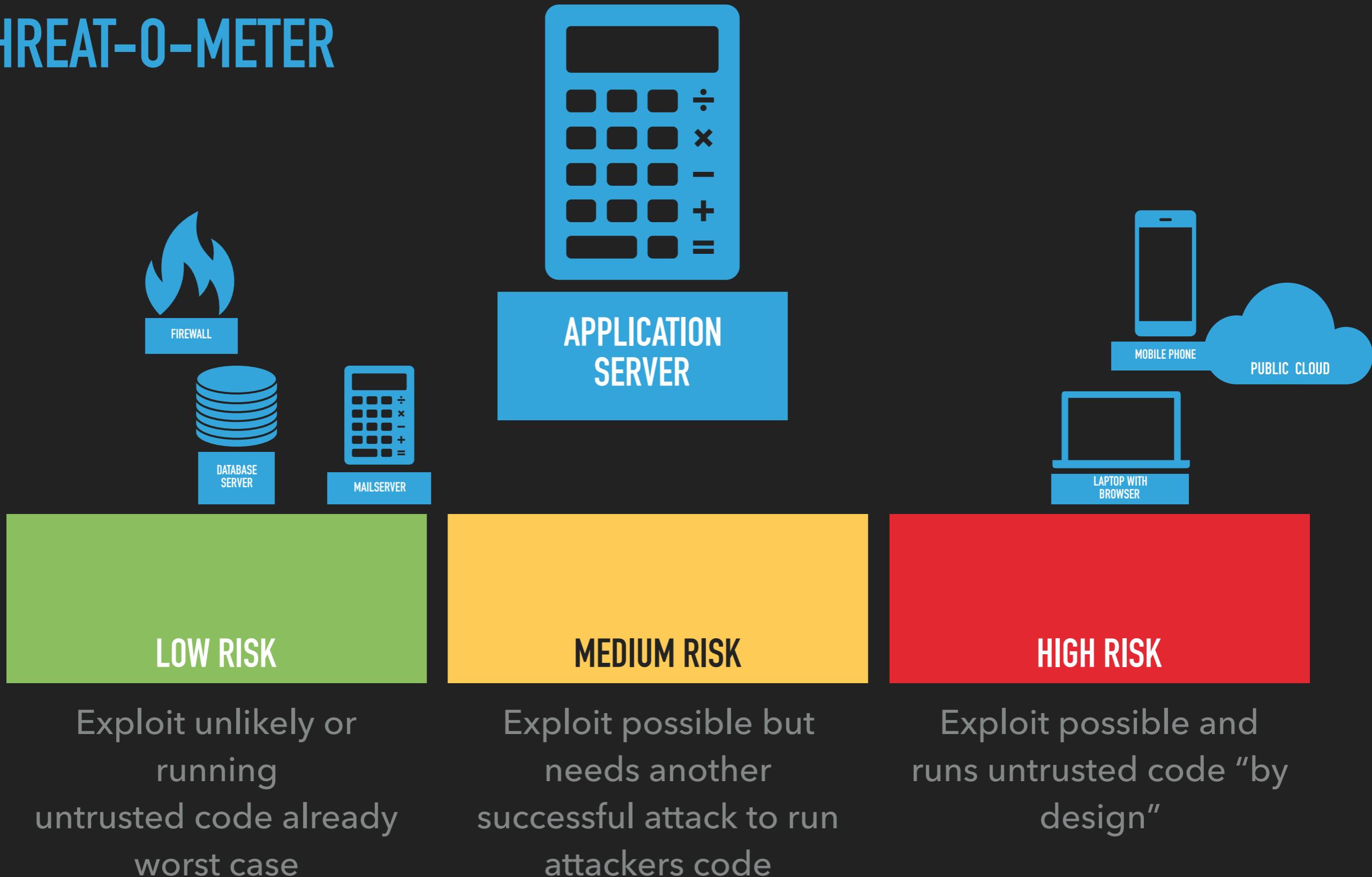
Firewalls and switches  
(normally) do not expose  
an attackable surface to  
the external network.

This greatly reduces the  
likelihood of attacks.

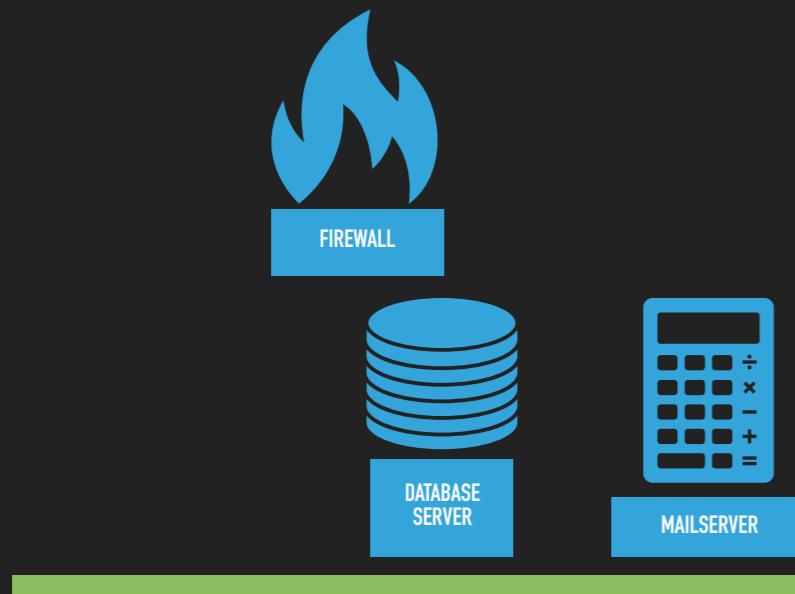
A code execution is  
already the worst case.

VPN gateways expose a  
complex interface and  
are more likely to be  
attacked.

## THREAT-O-METER



## THREAT-O-METER



Exploit unlikely or  
running  
untrusted code already  
worst case

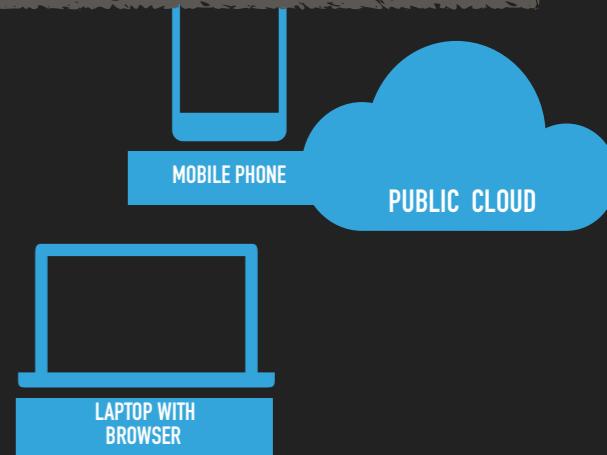


Exploit possible but  
needs another  
successful attack to run  
attackers code

Application servers only  
run trusted code but  
attacks can lead to code  
execution.



Exploit possible and  
runs untrusted code "by  
design"



## THREAT-O-METER

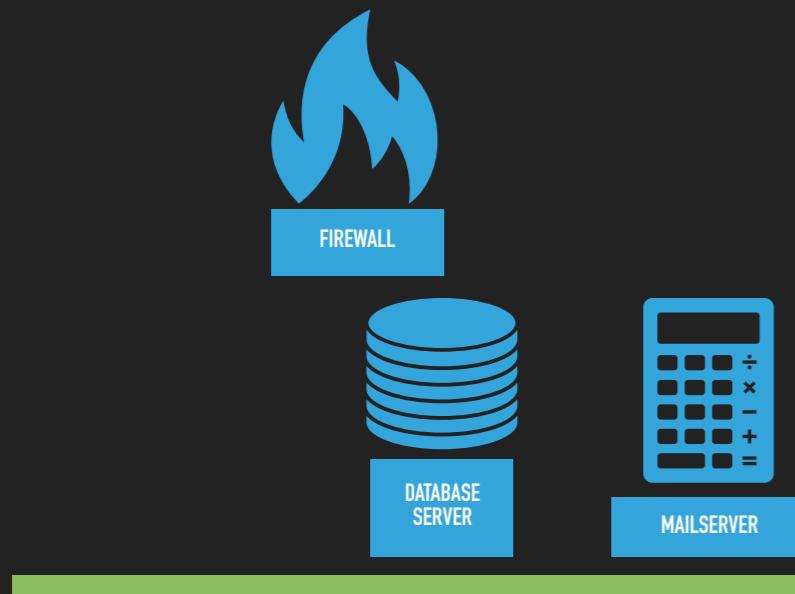


LOW RISK  
Exploit unlikely or  
running  
untrusted code already  
worst case

MEDIUM RISK  
Exploit possible but  
needs another  
successful attack to run  
attackers code

HIGH RISK  
Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

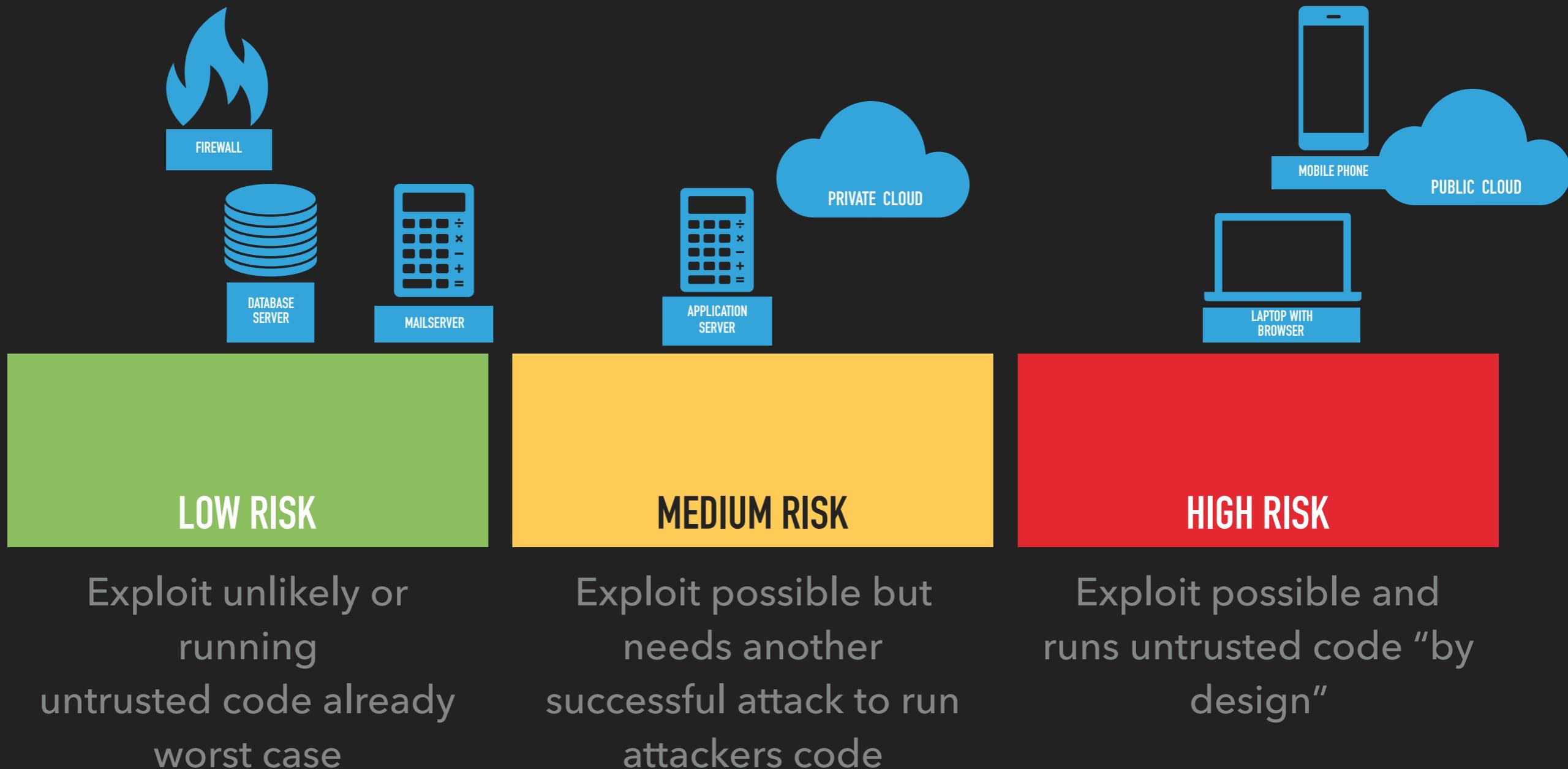
**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

Private clouds run many different workloads but they are all trusted.

An attacker only needs to hack one application running in the cloud to run a Spectre attack.

## THREAT-O-METER



## THREAT-O-METER

Given the patches are risky w. regards to performance and availability.

**What would be your patching strategy for each risk class?**



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"



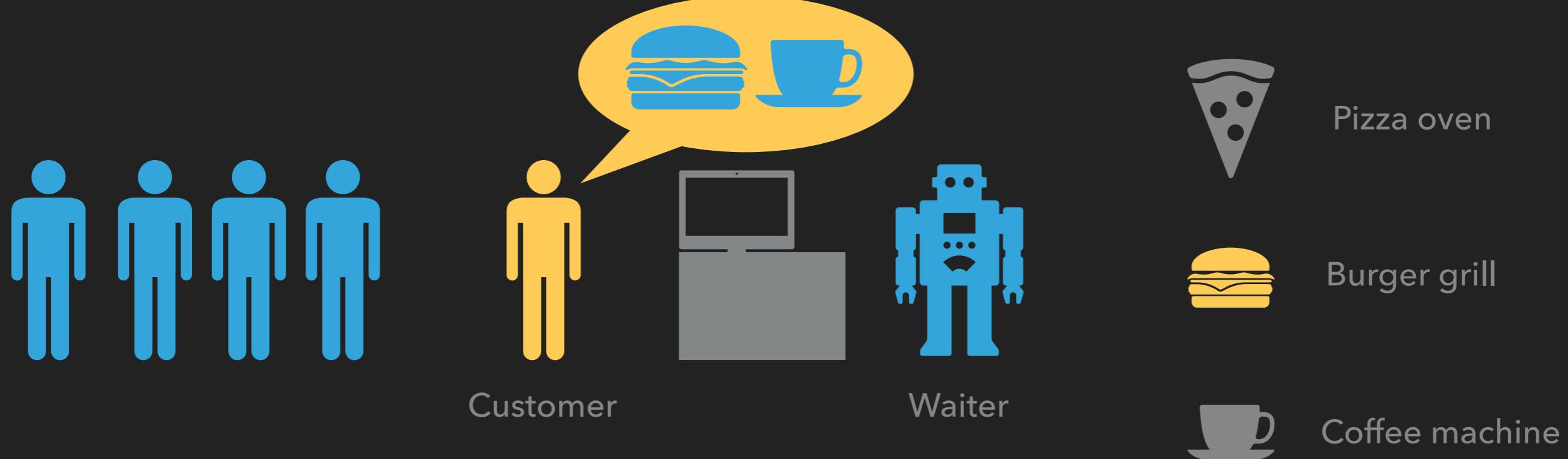
ACCIDENT, MALICE,  
INCOMPETENCE?

---

WHY DID IT  
HAPPEN?

## CONFIDENTIAL BURGERS INC.

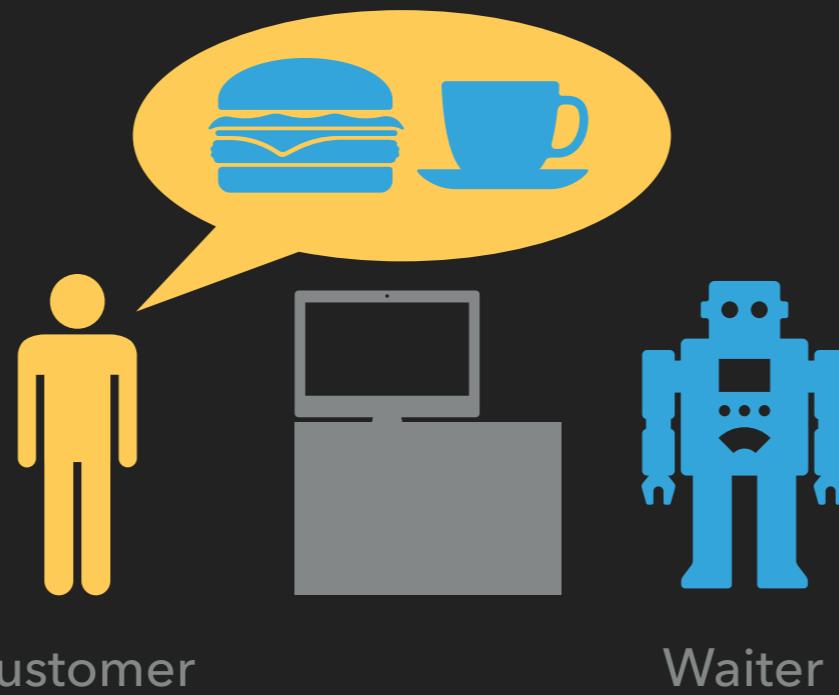
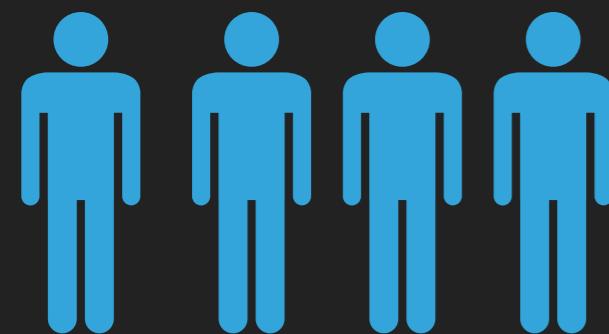
**Confidential Burgers inc.** sells burgers, pizza, and coffee.



*Grand Opening Today*

## CONFIDENTIAL BURGERS INC.

**Confidential Burgers inc.** sells burgers, pizza, and coffee.



The waiter (CPU) will

Customer

Waiter



Pizza oven



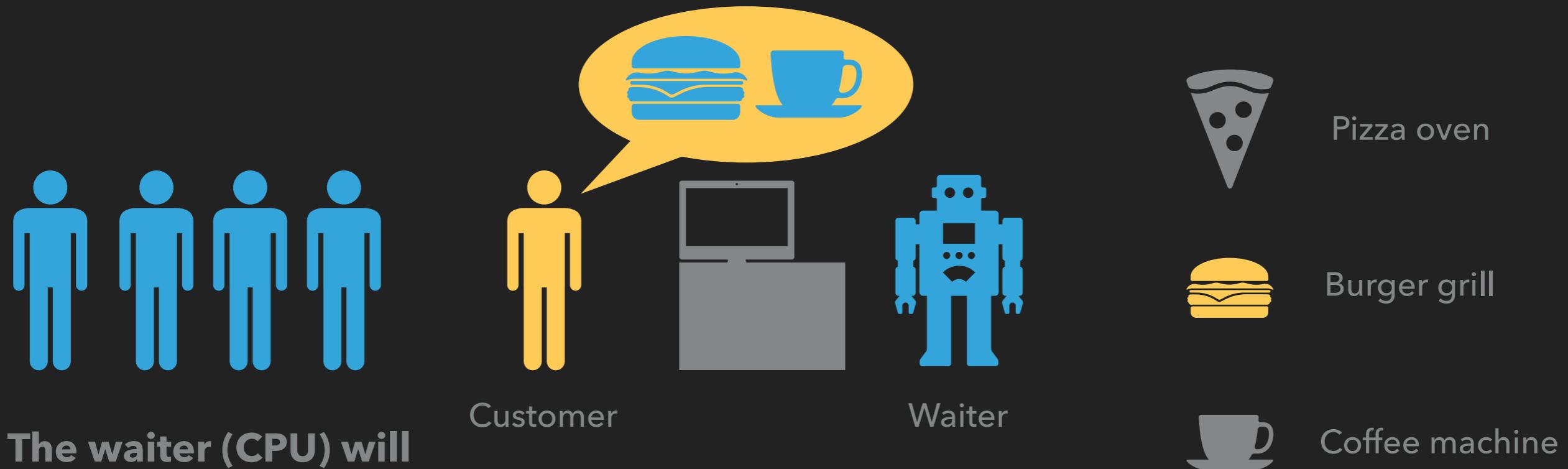
Burger grill



Coffee machine

## CONFIDENTIAL BURGERS INC.

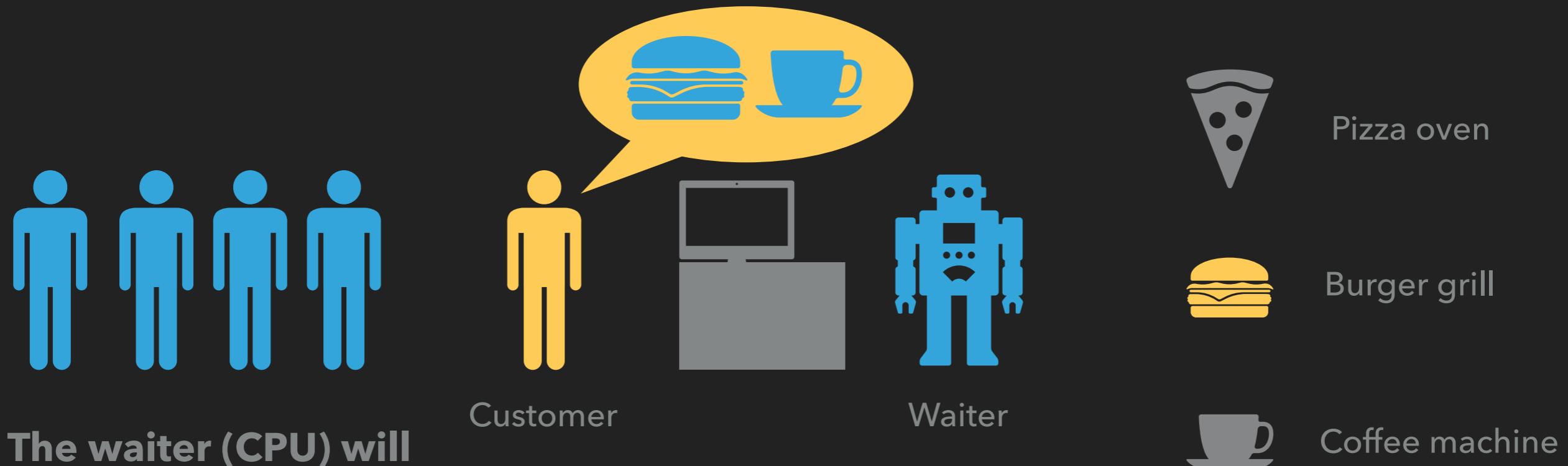
**Confidential Burgers inc. sells burgers, pizza, and coffee.**



1. take an order from a customer (CPU instruction)

## CONFIDENTIAL BURGERS INC.

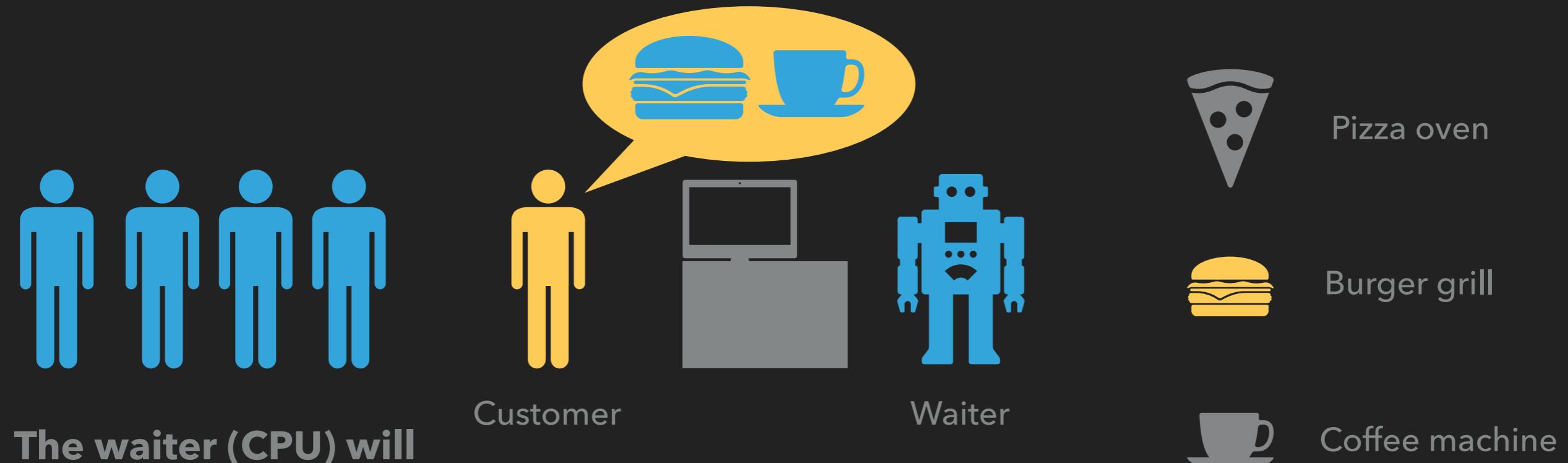
**Confidential Burgers inc.** sells burgers, pizza, and coffee.



1. take an order from a customer (CPU instruction)
2. break the order (instruction) down into micro operations ( $\mu$ OPs - grilling a burger, baking a pizza, ...)

## CONFIDENTIAL BURGERS INC.

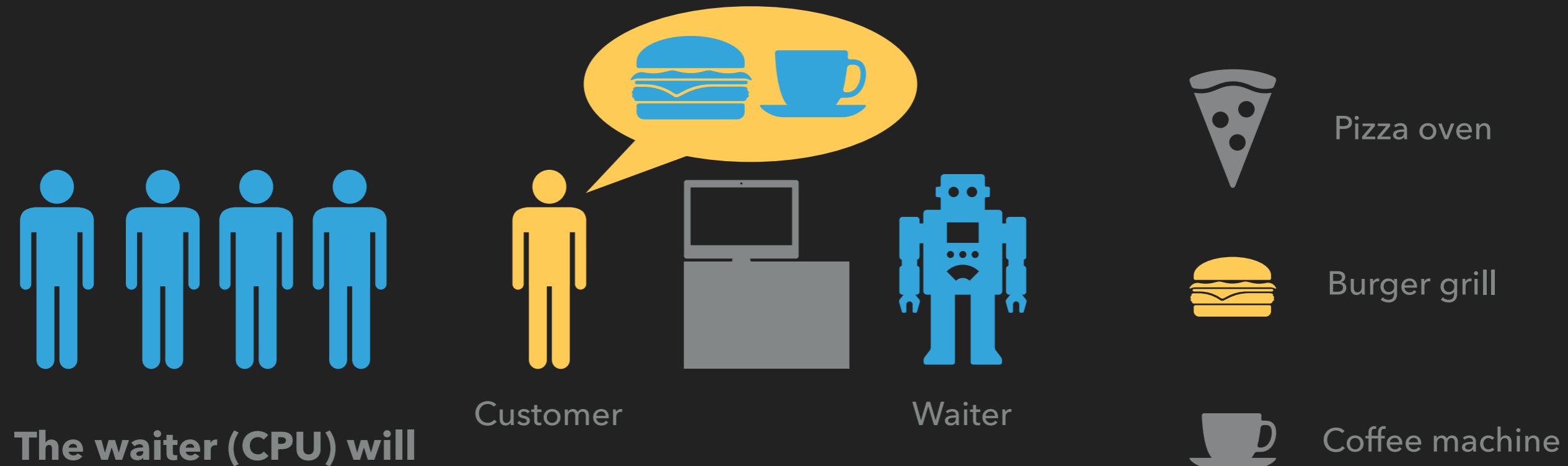
**Confidential Burgers inc.** sells burgers, pizza, and coffee.



1. take an order from a customer (CPU instruction)
2. break the order (instruction) down into micro operations ( $\mu$ OPs - grilling a burger, baking a pizza, ...)
3. schedule and execute the  $\mu$ OPs

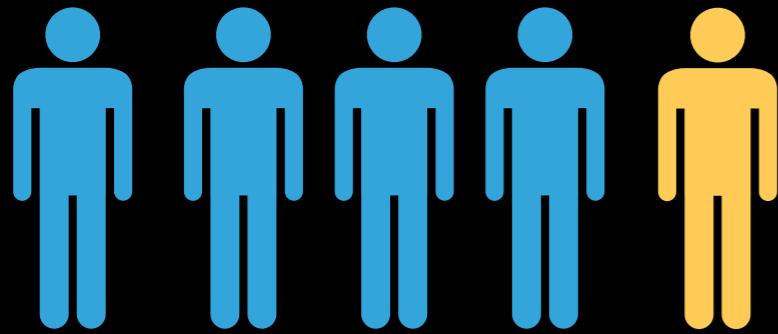
## CONFIDENTIAL BURGERS INC.

**Confidential Burgers inc.** sells burgers, pizza, and coffee.

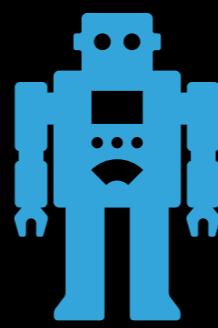


1. take an order from a customer (CPU instruction)
2. break the order (instruction) down into micro operations ( $\mu$ OPs - grilling a burger, baking a pizza, ...)
3. schedule and execute the  $\mu$ OPs
4. complete the order (retire the instruction)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

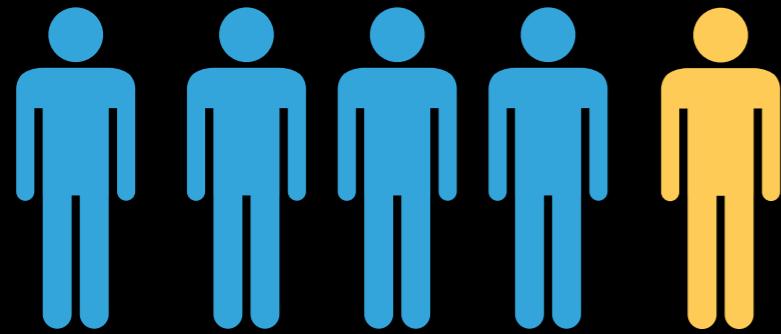


Burger grill

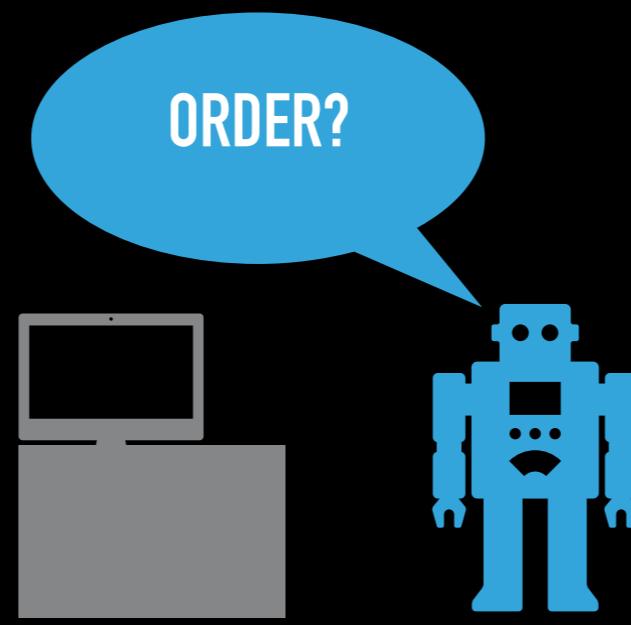


Coffee machine

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

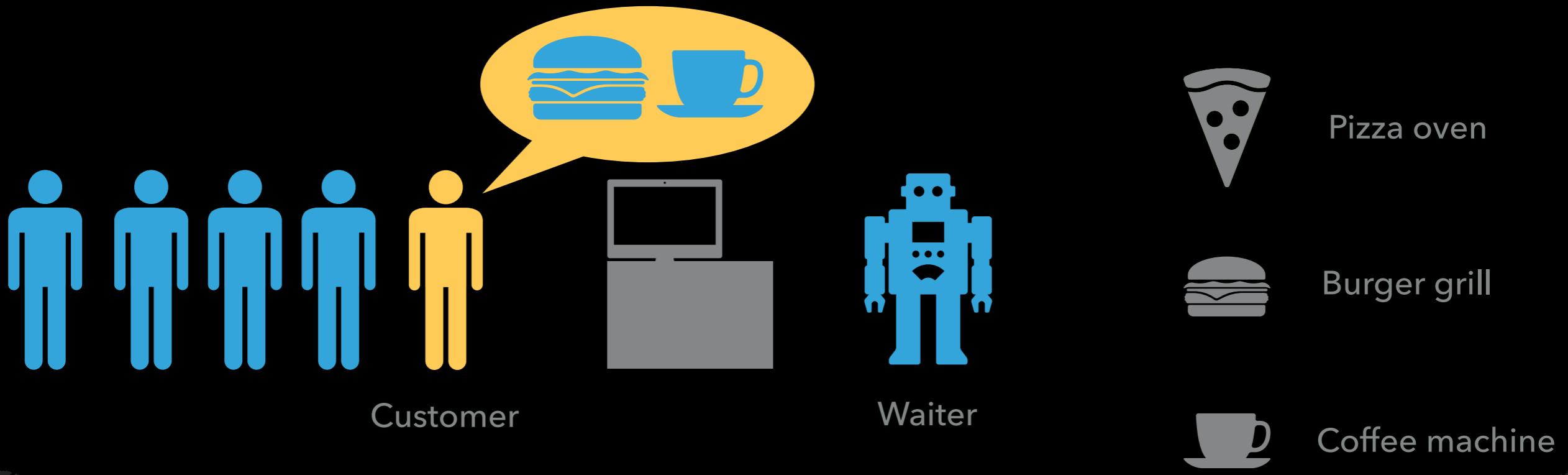


Burger grill



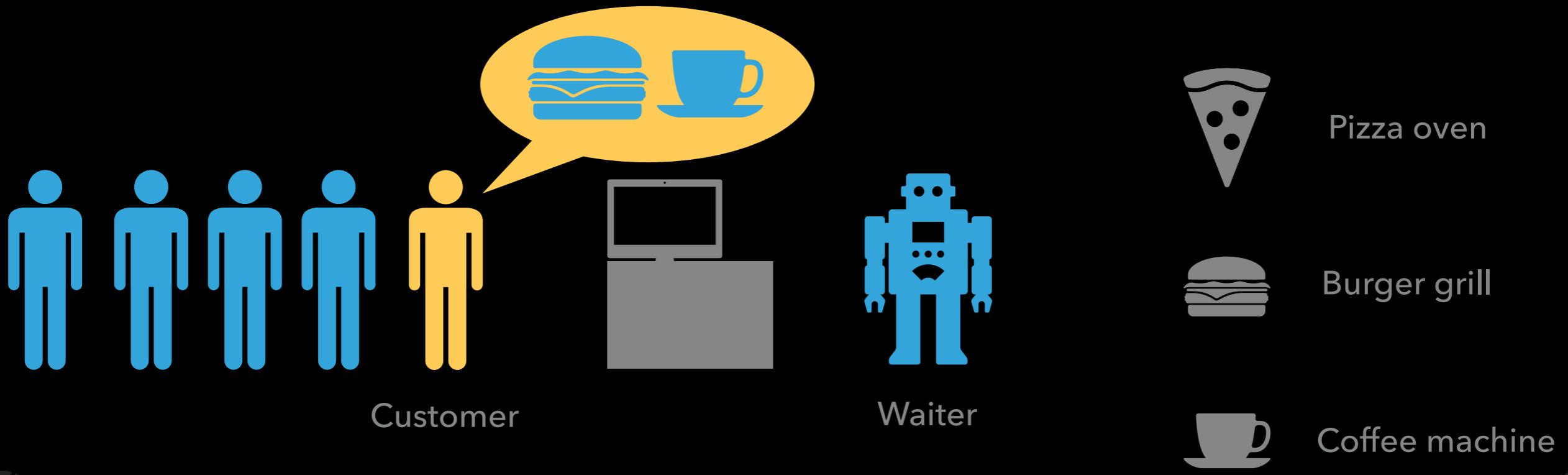
Coffee machine

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



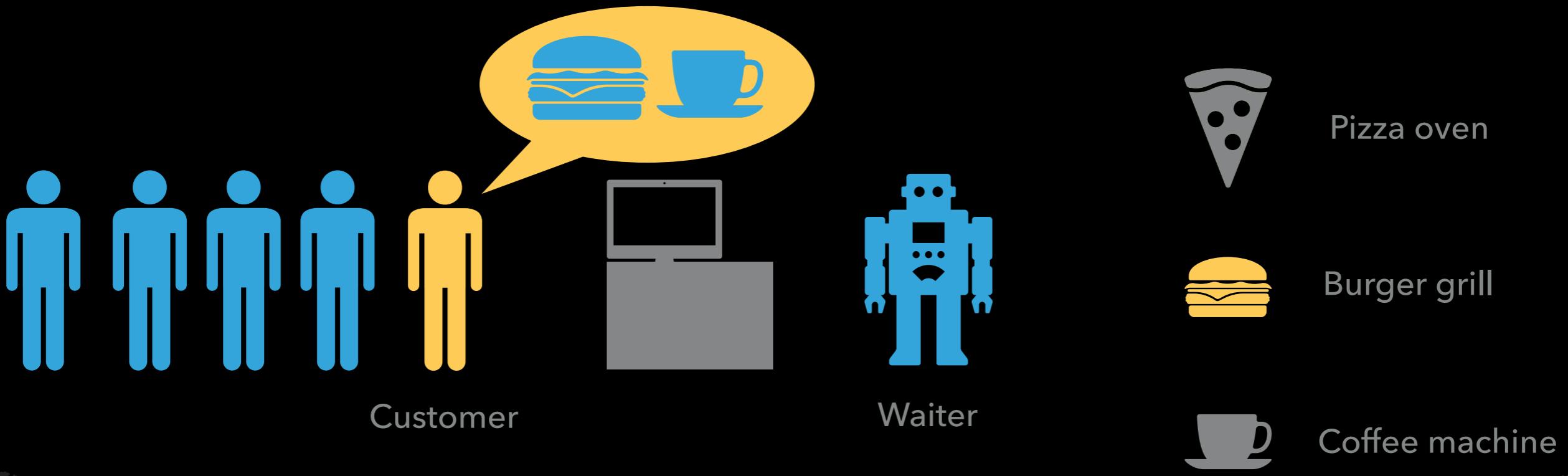
- ▶ Decode instruction into µOPs ("Burger", "Coffee")

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



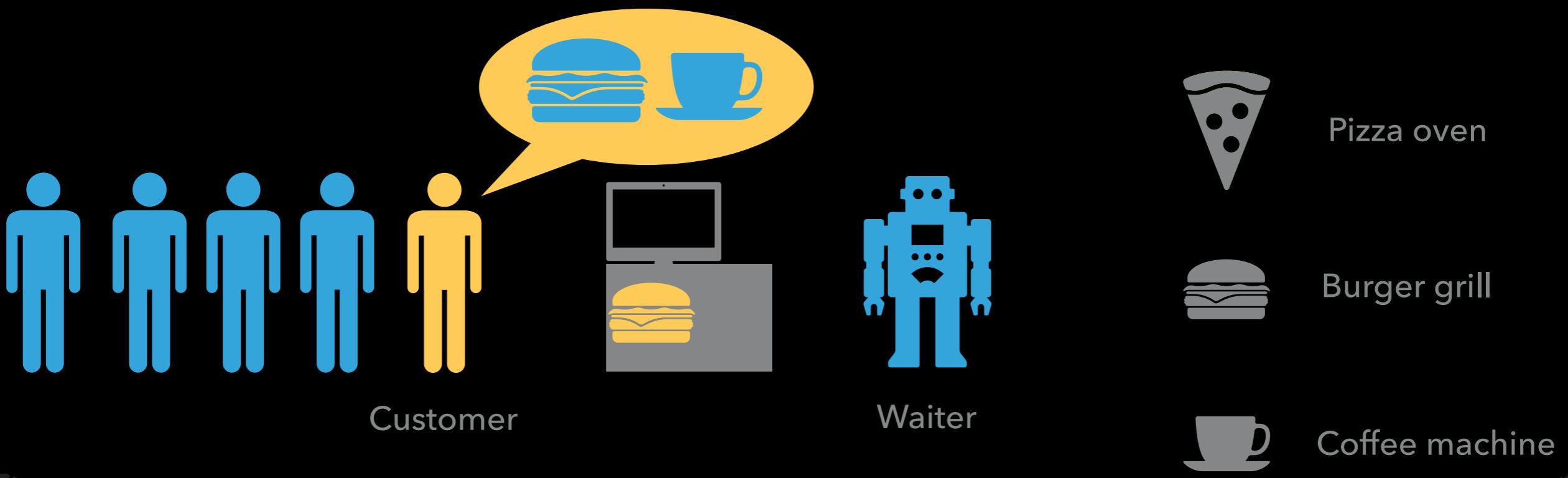
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



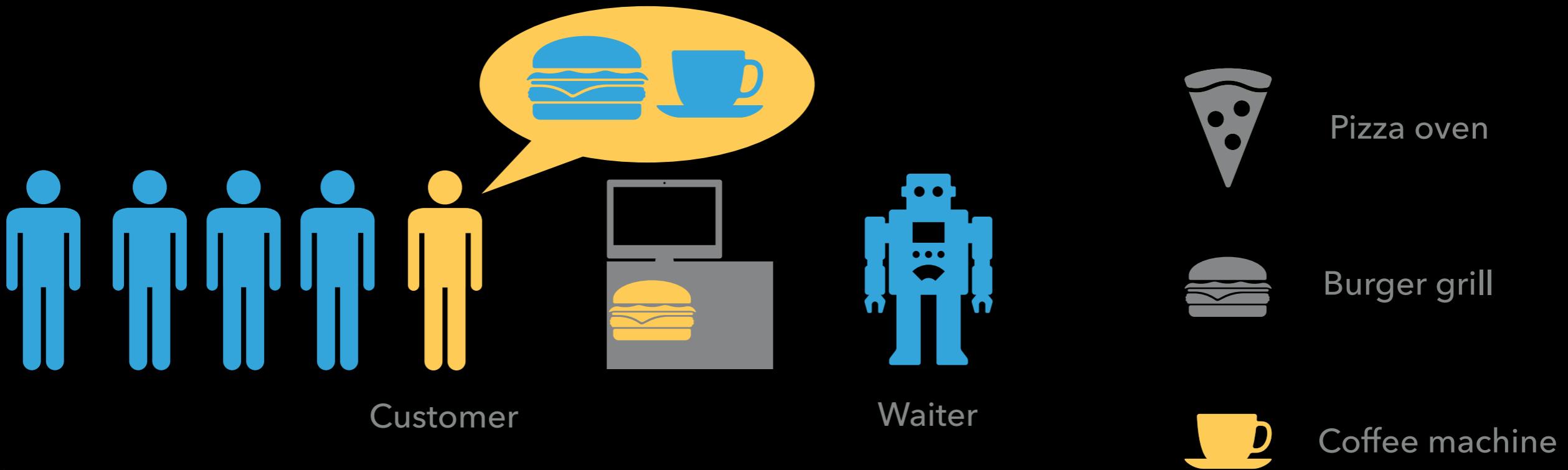
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
- ▶ run 1st µOP (grill the burger)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



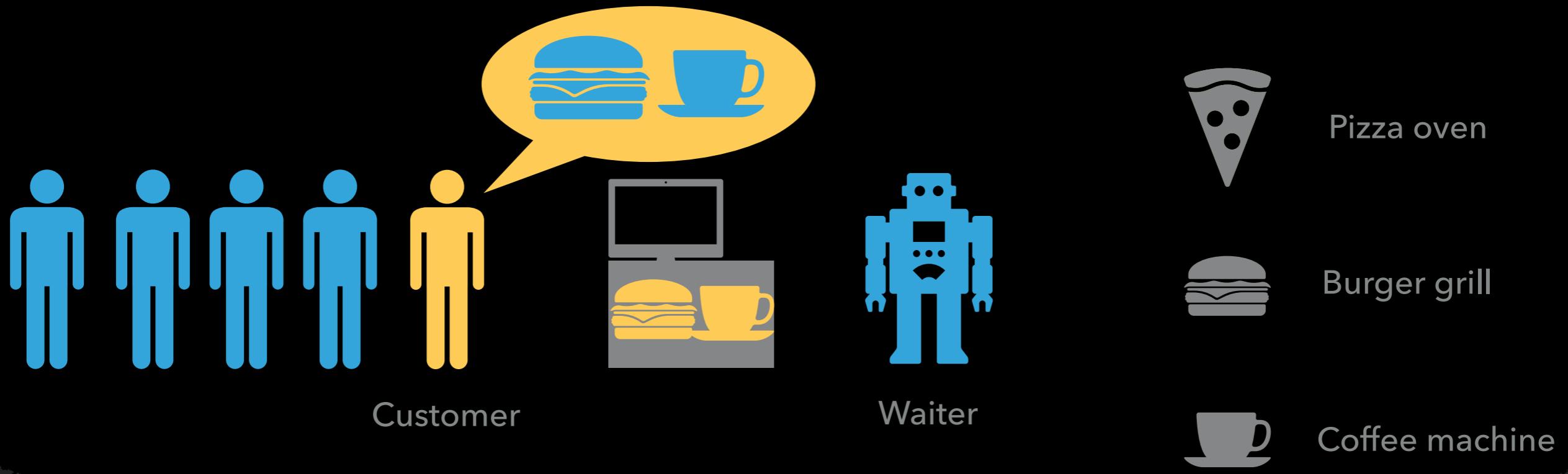
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
- ▶ run 1st µOP (grill the burger)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



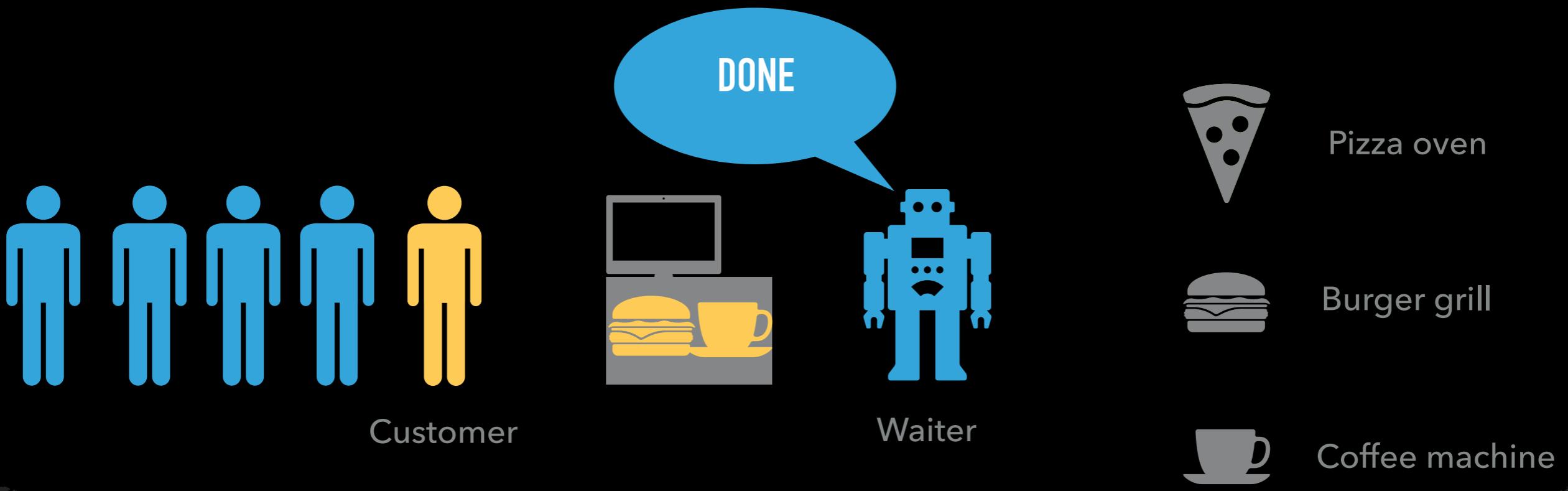
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
  - ▶ run 1st µOP (grill the burger)
  - ▶ run 2nd µOP (brew coffee, serial execution)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



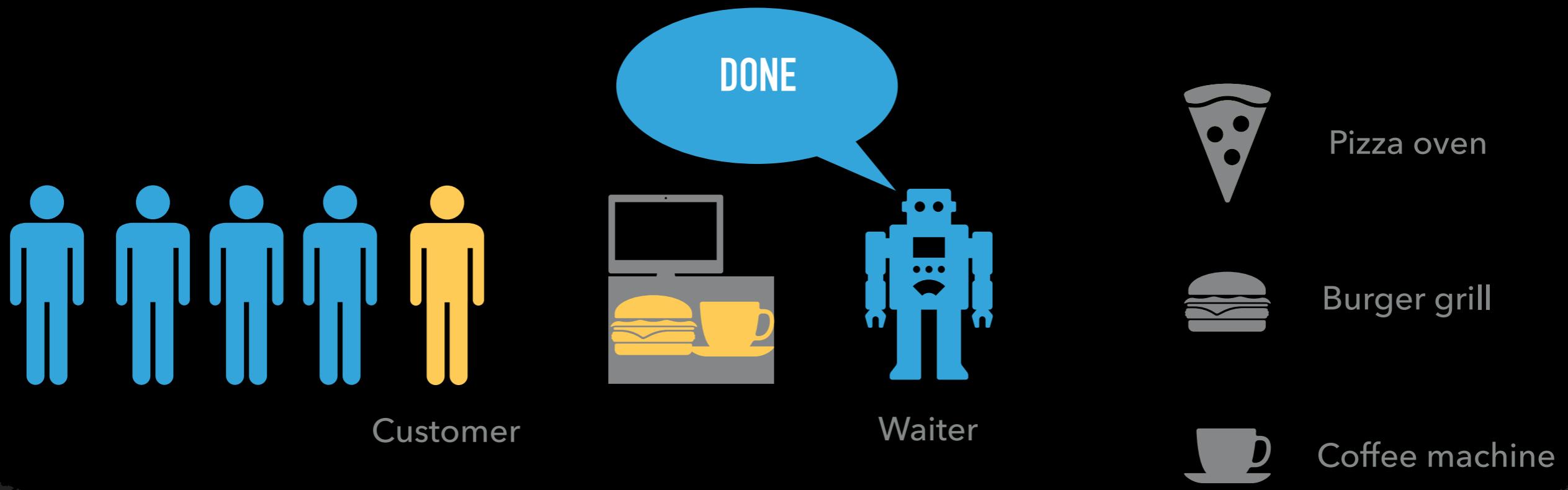
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
  - ▶ run 1st µOP (grill the burger)
  - ▶ run 2nd µOP (brew coffee, serial execution)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
  - ▶ run 1st µOP (grill the burger)
  - ▶ run 2nd µOP (brew coffee, serial execution)
- ▶ Retire instruction (customer)

# CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



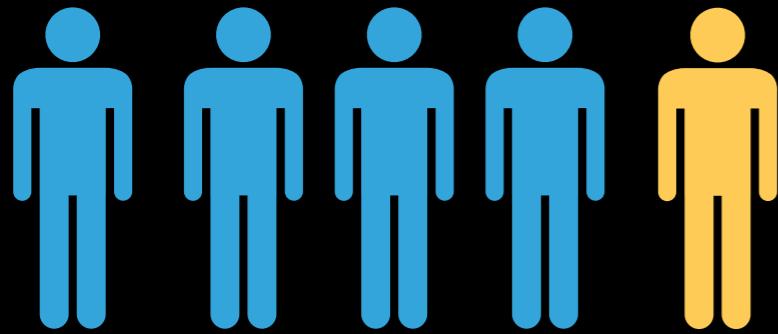
- ▶ One customer<sup>1</sup> after another (in order)
- ▶ Each part of the order <sup>2</sup> executed serially

i.e. first the burger, then the coffee

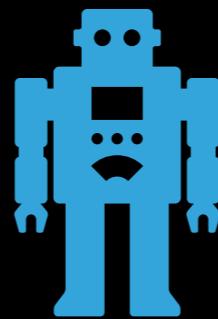
- ▶ PRO: Easy to implement and understand
- ▶ CON: Slow because resources<sup>3</sup> not utilised fully

<sup>1</sup> customer == CPU instruction    <sup>2</sup> part == µOP - micro operation    <sup>3</sup> oven, grill, coffee machine

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

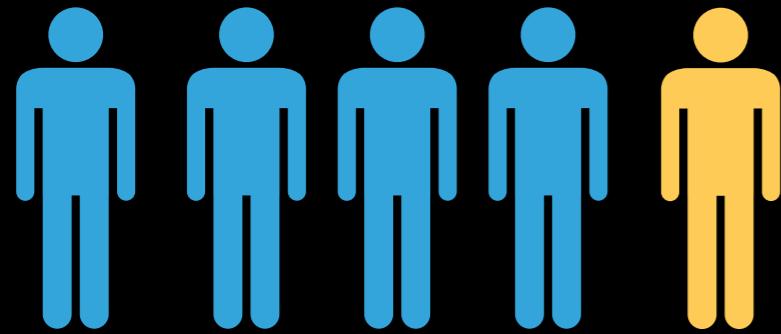


Burger grill

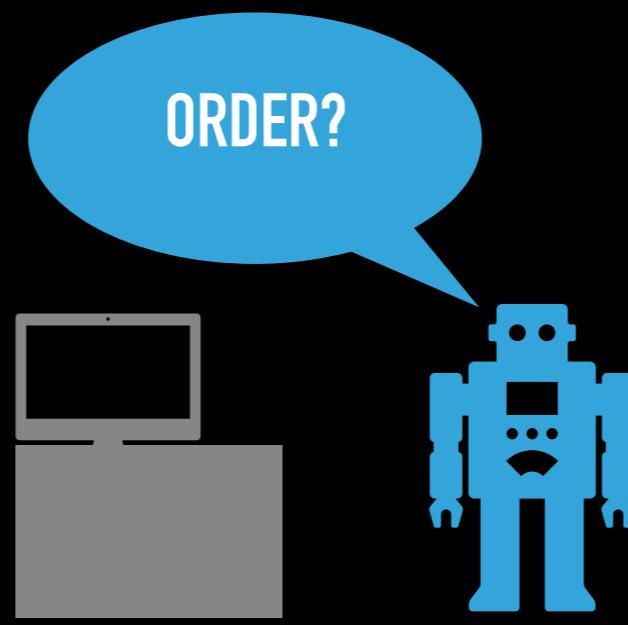


Coffee machine

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

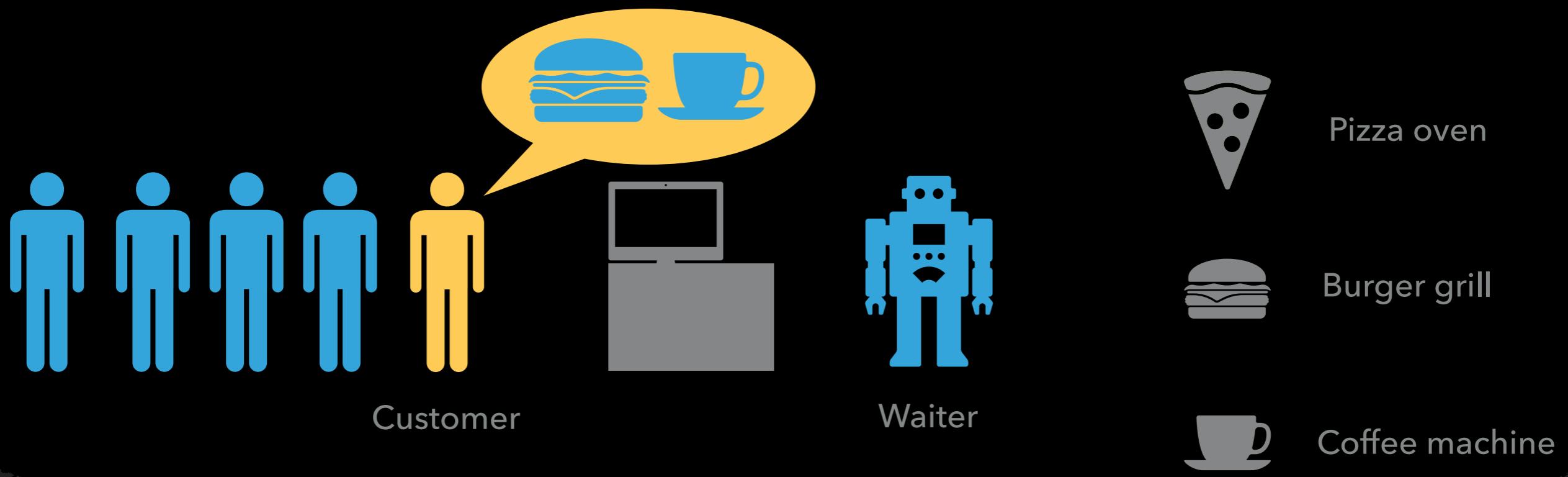


Burger grill



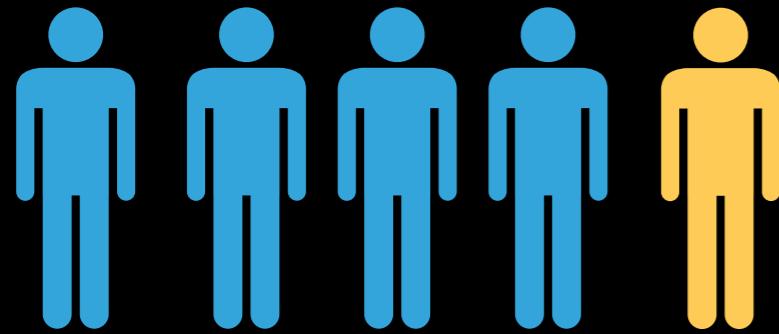
Coffee machine

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

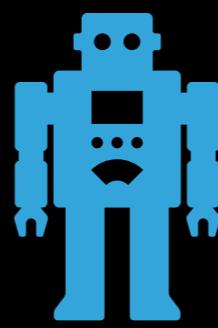


- ▶ Decode instruction into  $\mu$ OPs

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven



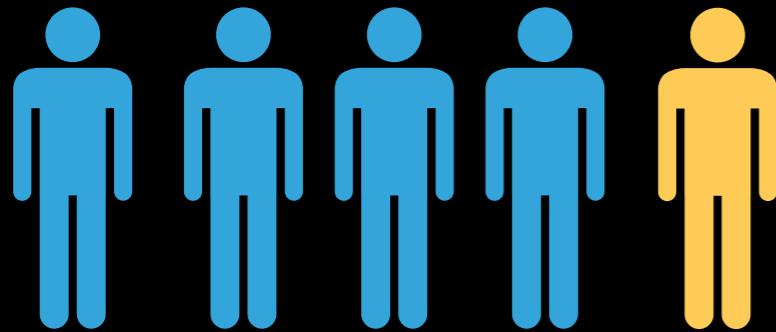
Burger grill



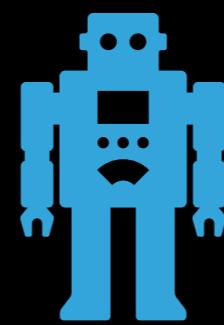
Coffee machine

- ▶ Decode instruction into µOPs
- ▶ Schedule µOPs

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven



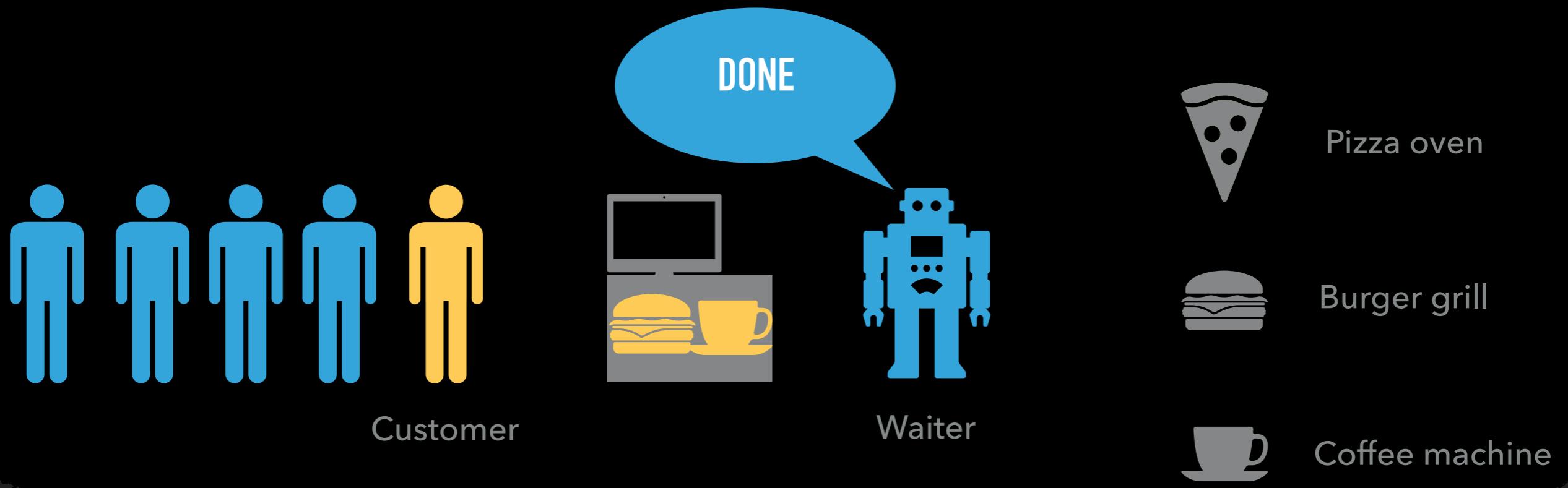
Burger grill



Coffee machine

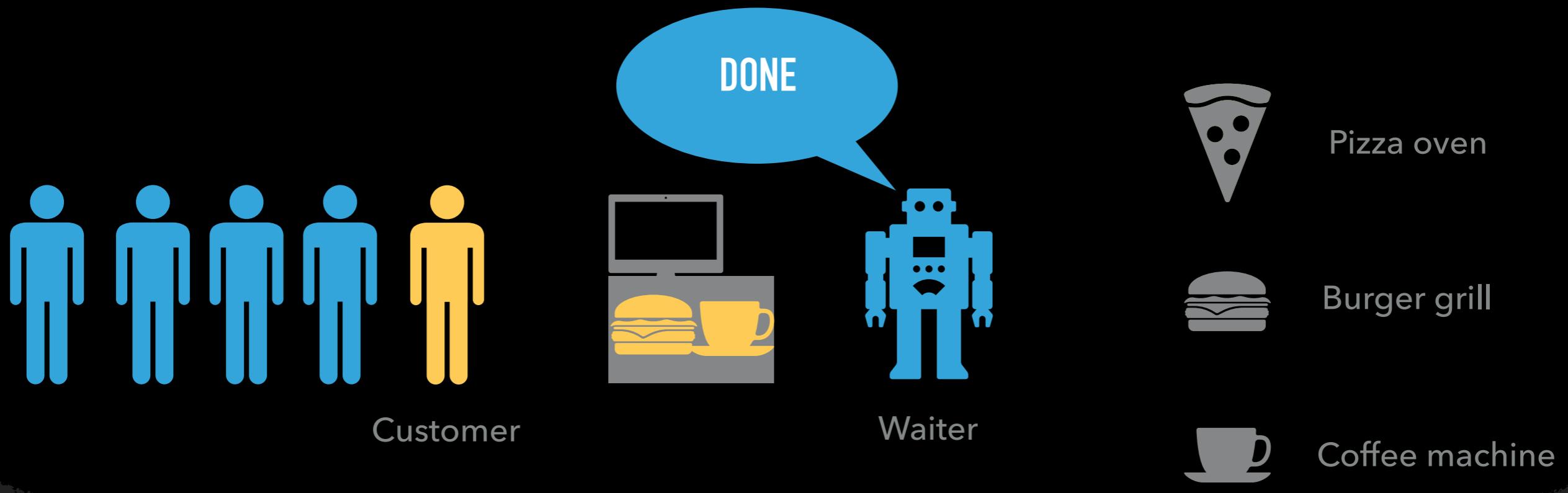
- ▶ Decode instruction into µOPs
- ▶ Schedule µOPs
- ▶ run 1st µOP and 2nd µOP (parallel execution of µOPs)

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



- ▶ Decode instruction into µOPs
- ▶ Schedule µOPs
  - ▶ run 1st µOP and 2nd µOP (parallel execution of µOPs)
- ▶ retire instruction (customer)

# CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

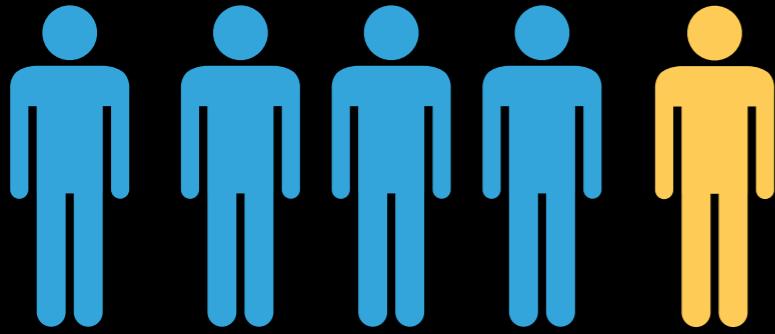


- ▶ One customer<sup>1</sup> after another (in order)
- ▶ Each part of the order <sup>2</sup> executed in parallel

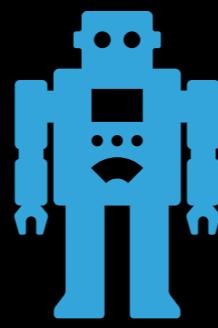
i.e. burger and coffee prepared at the same time

- ▶ PRO: **Faster bc. of better resource utilisation.**
- ▶ CON: Still not perfect, more complex

## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer



Waiter



Pizza oven

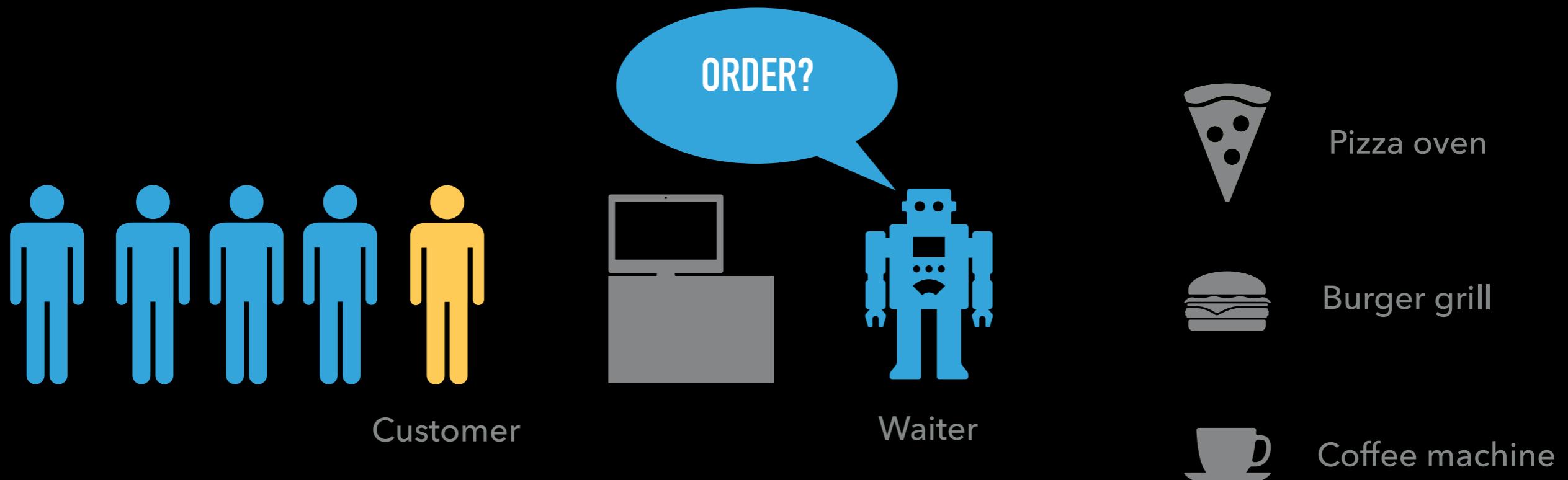


Burger grill

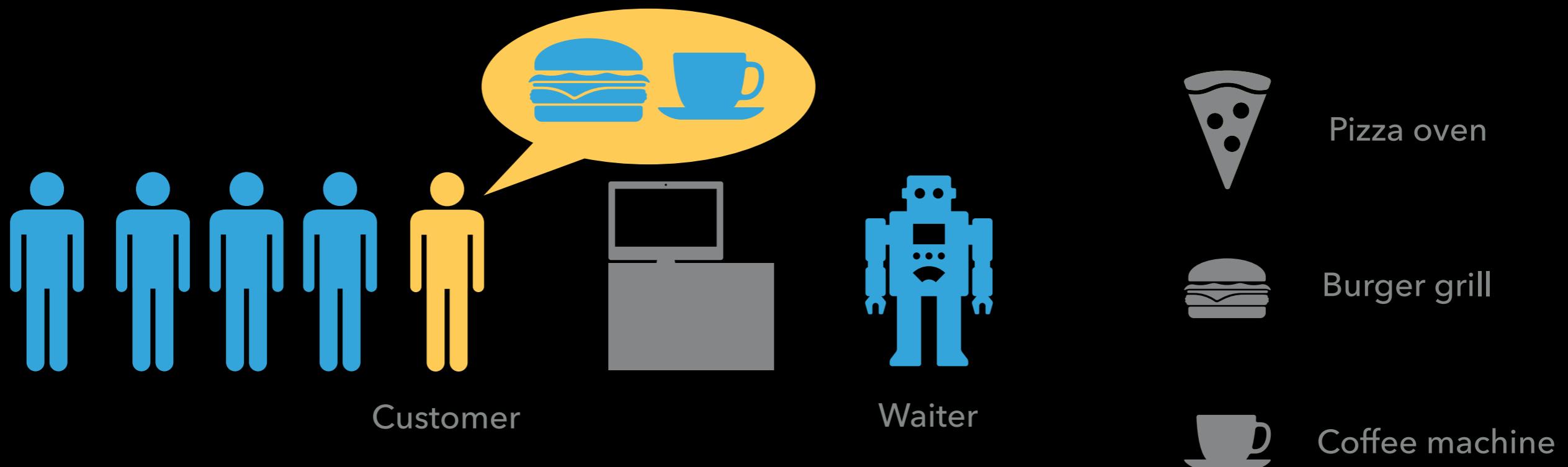


Coffee machine

## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



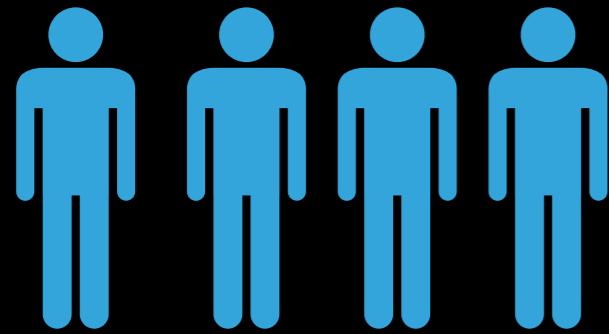
## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



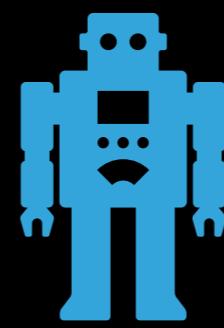
## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



# CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer



Waiter



Pizza oven



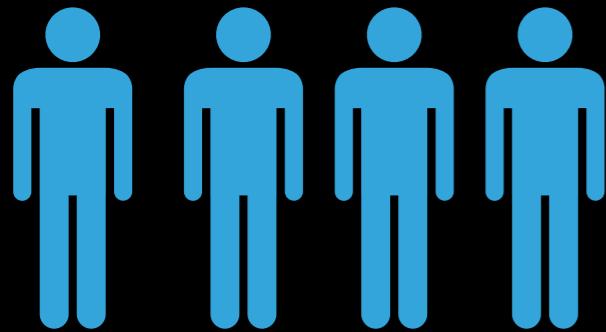
Burger grill



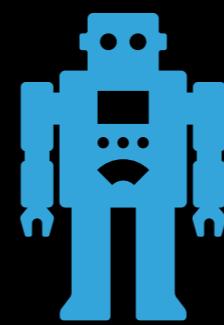
Coffee machine



## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer



Waiter



Pizza oven



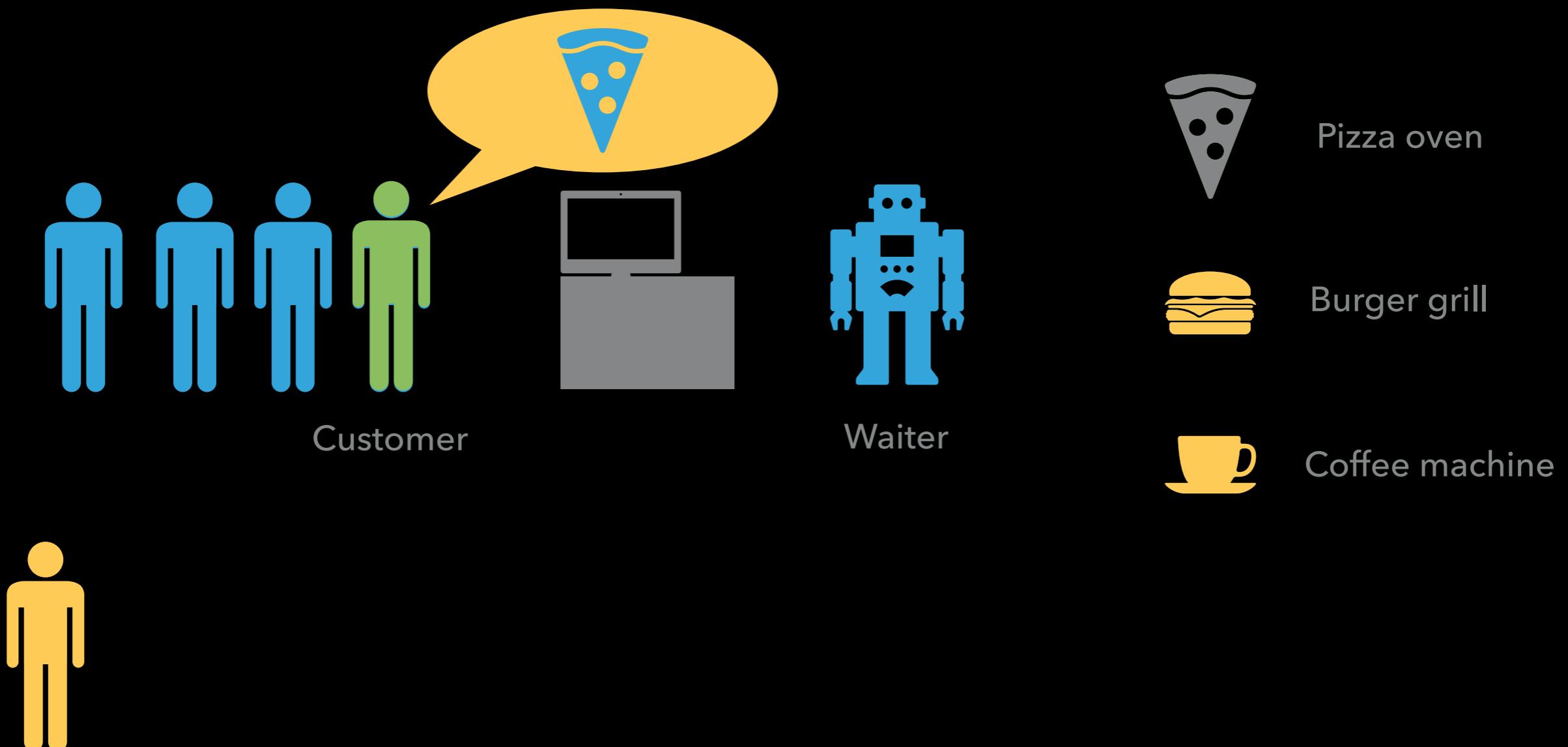
Burger grill



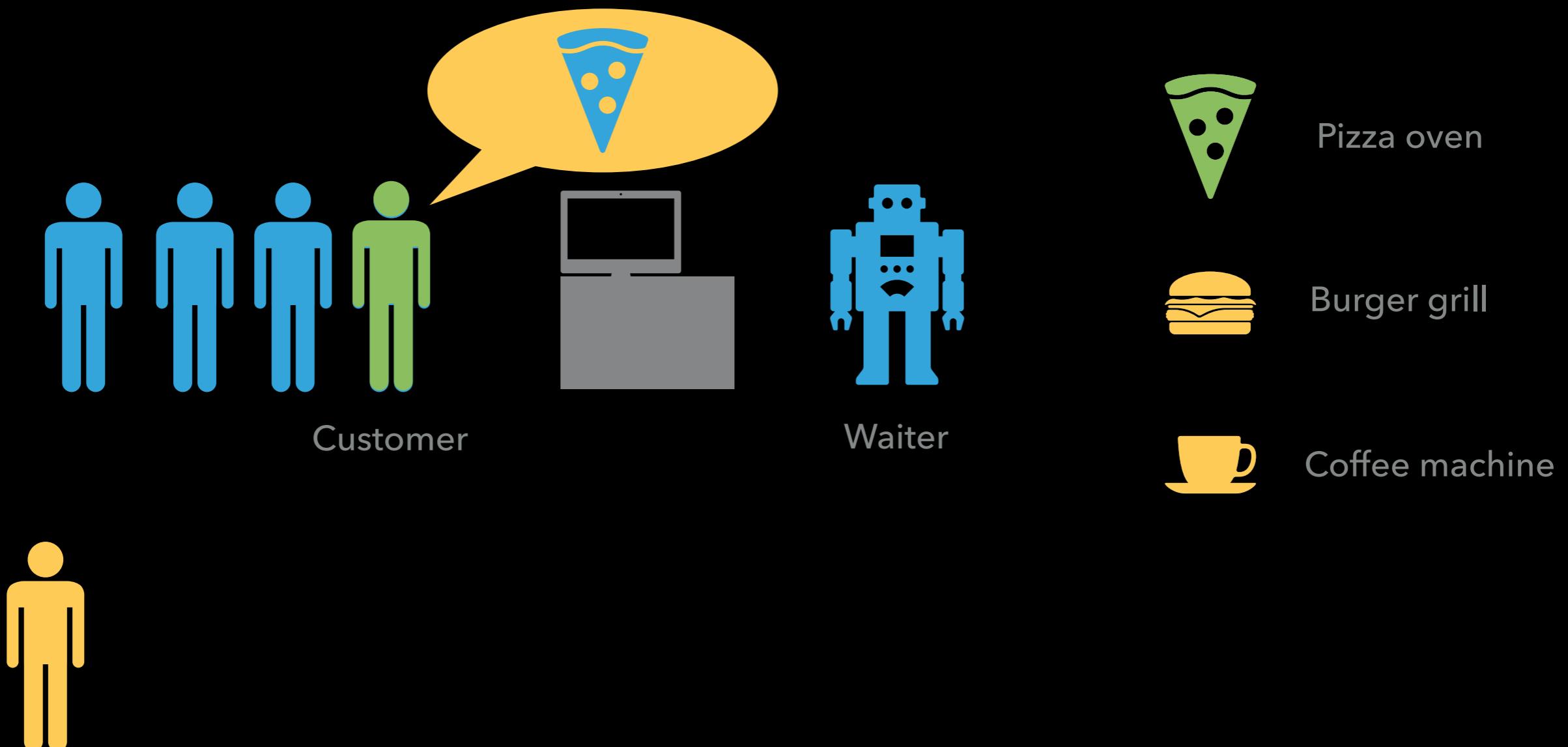
Coffee machine



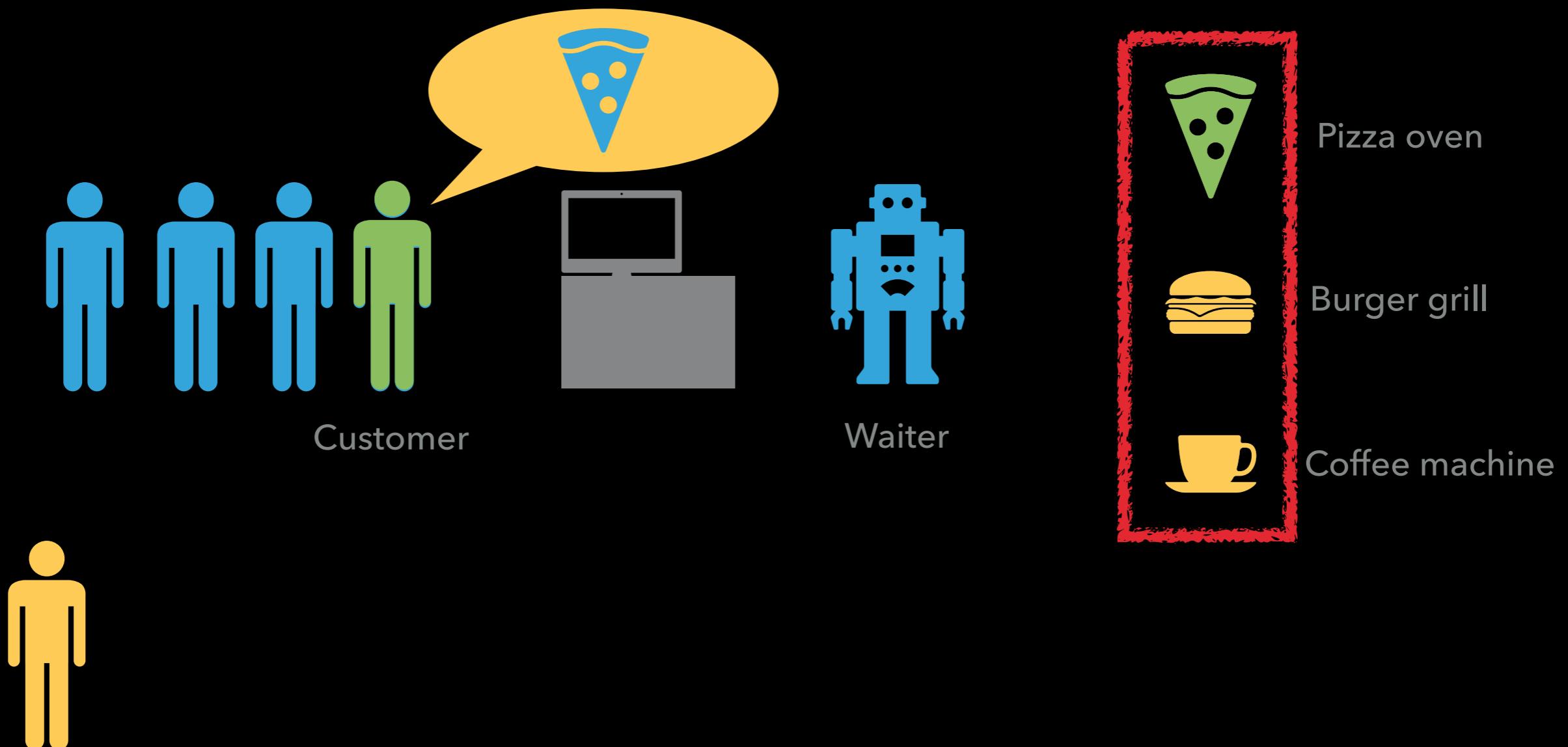
## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



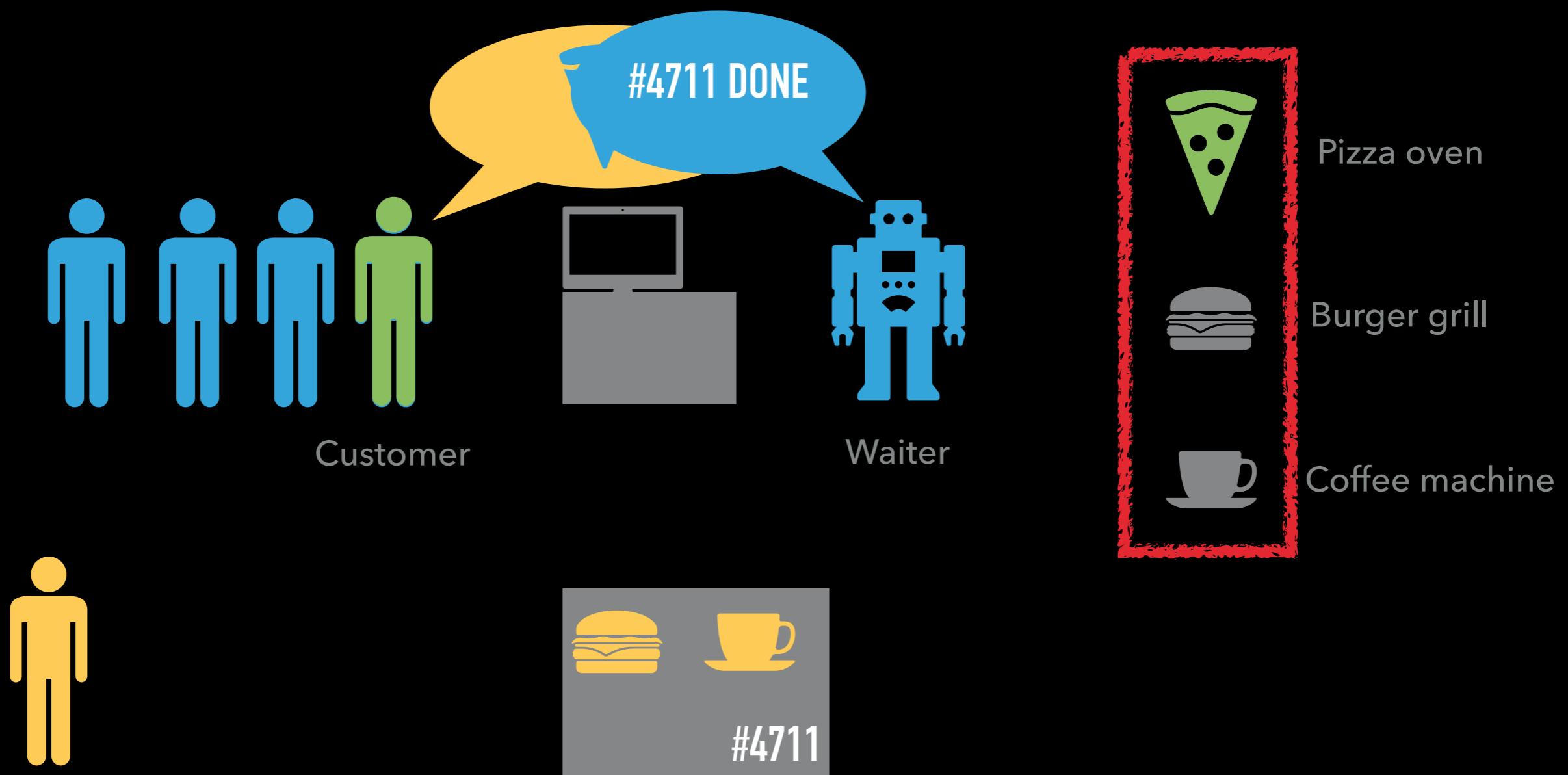
# CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



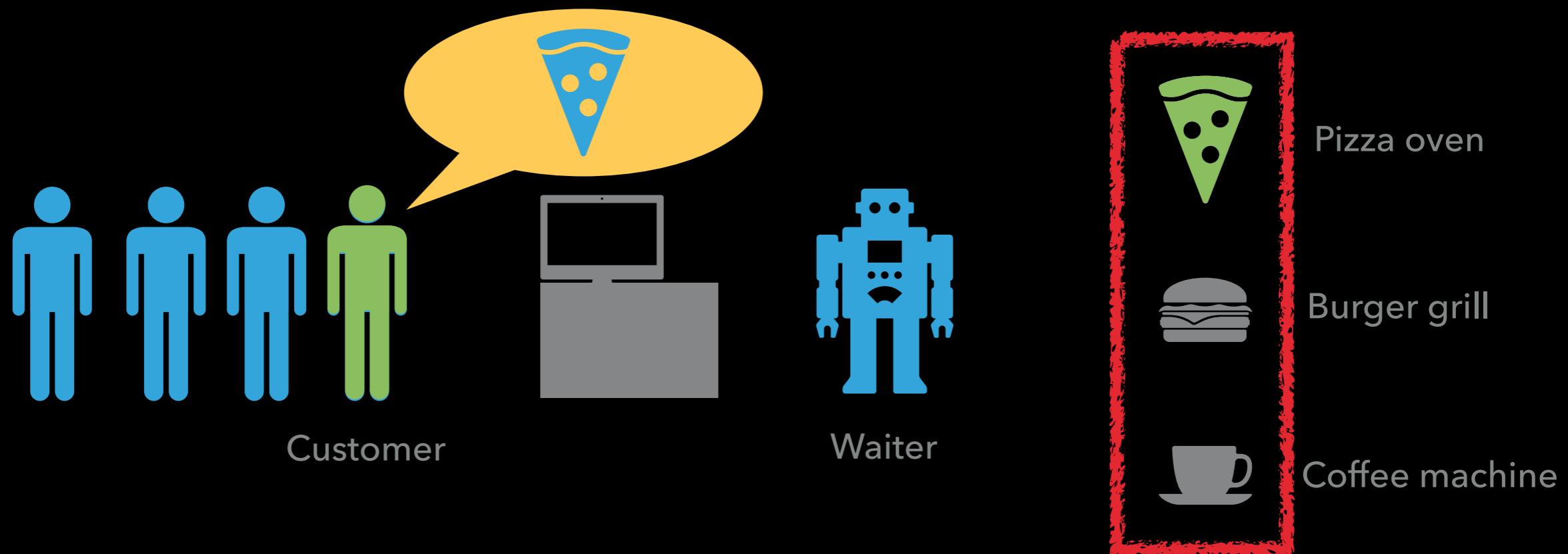
## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



# CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



## CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



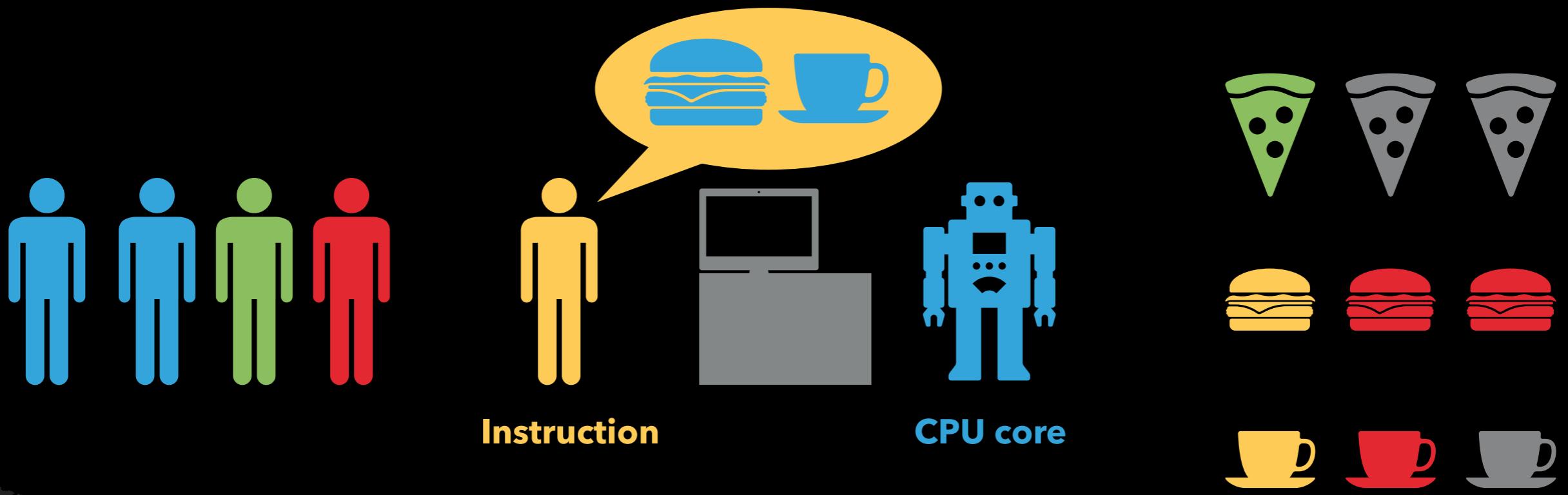
- ▶ Multiple customers' orders executed in parallel<sup>1</sup> and delivered (retired) **in order**

i.e. multiple orders prepared at the same time

- ▶ PRO: **Faster because resources are utilised even better**
- ▶ CON: More difficult to implement

<sup>1</sup> this is called *superscalar*

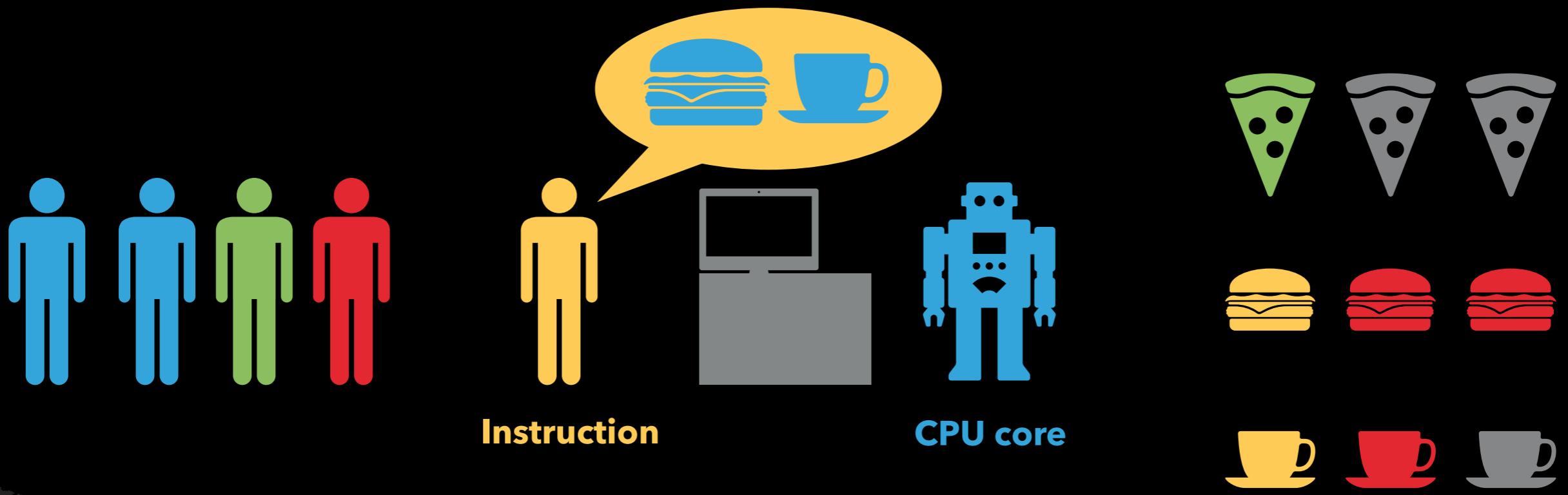
# CONFIDENTIAL BURGERS INC.



Adding more resources increase parallelism & throughput.

This is all on one CPU core.

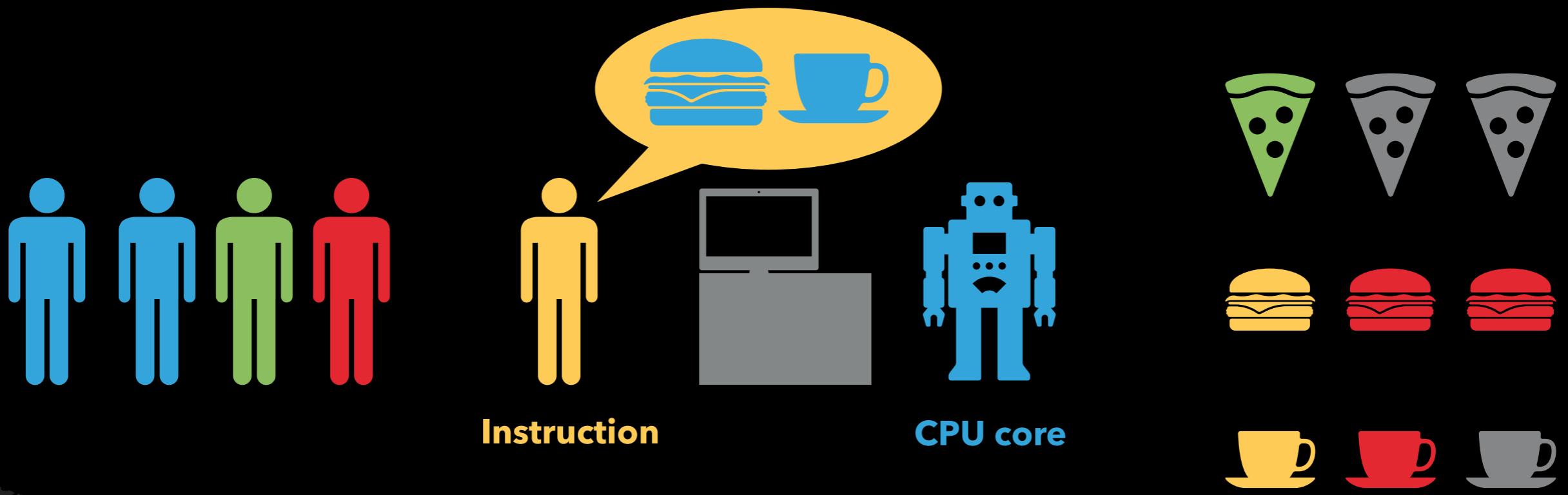
## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



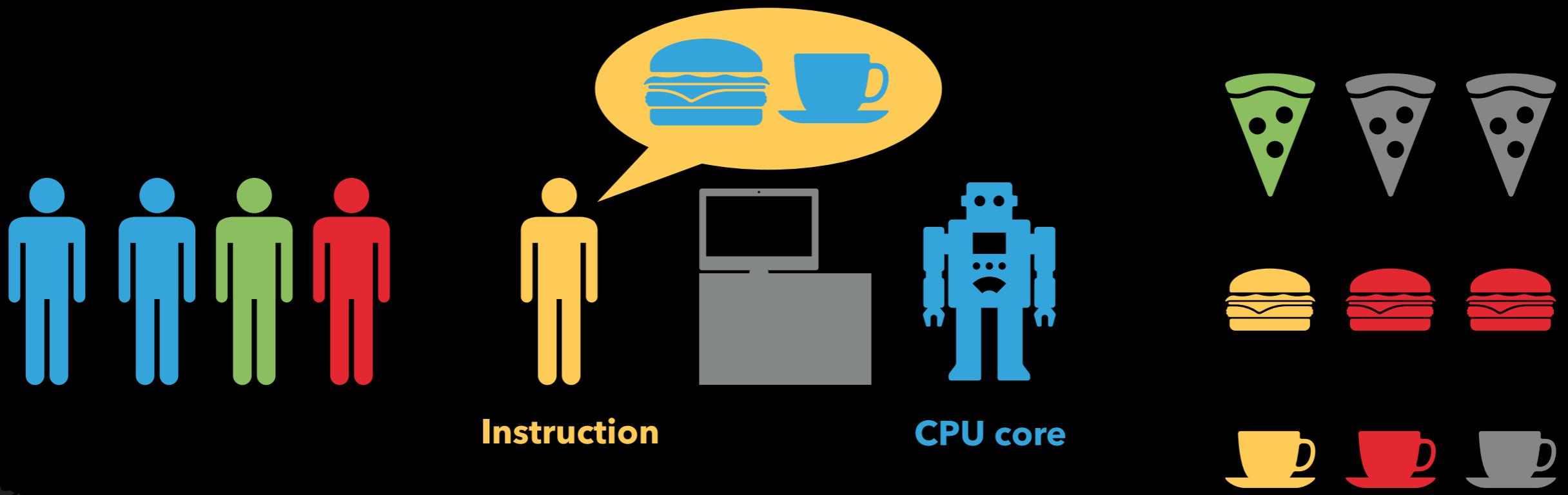
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



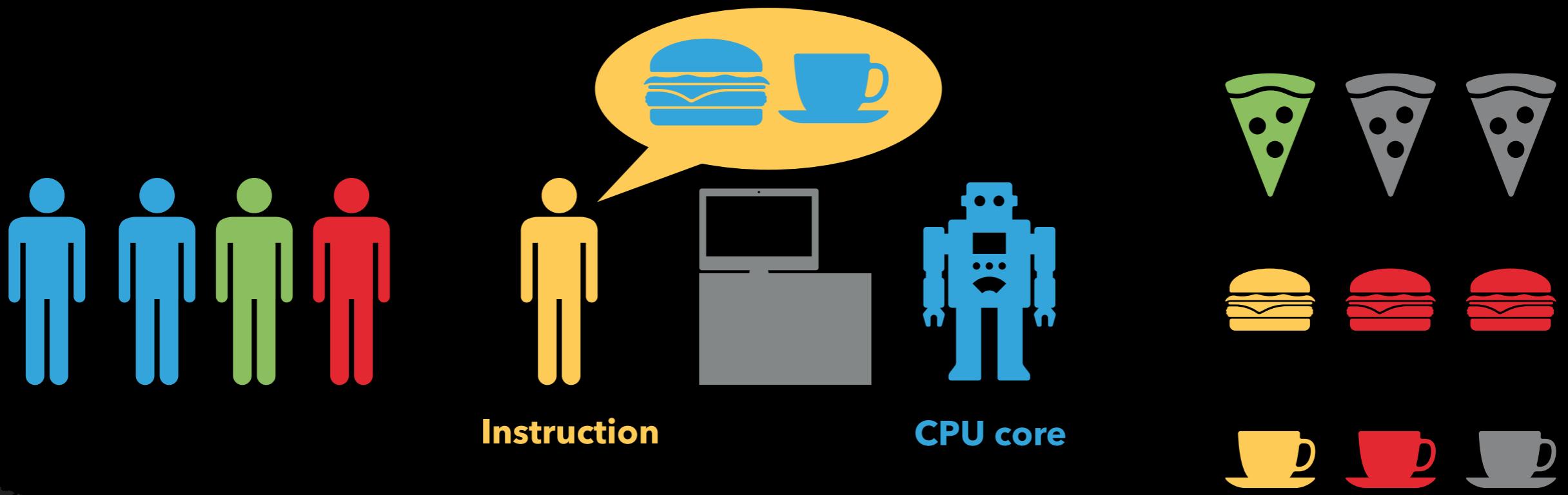
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



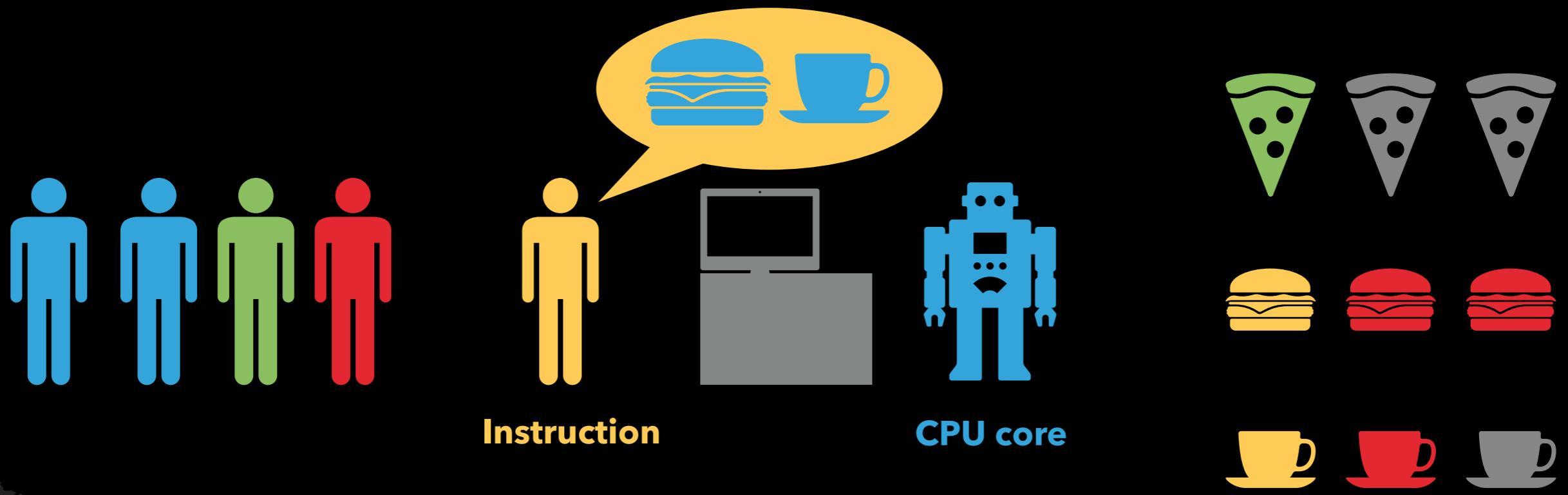
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



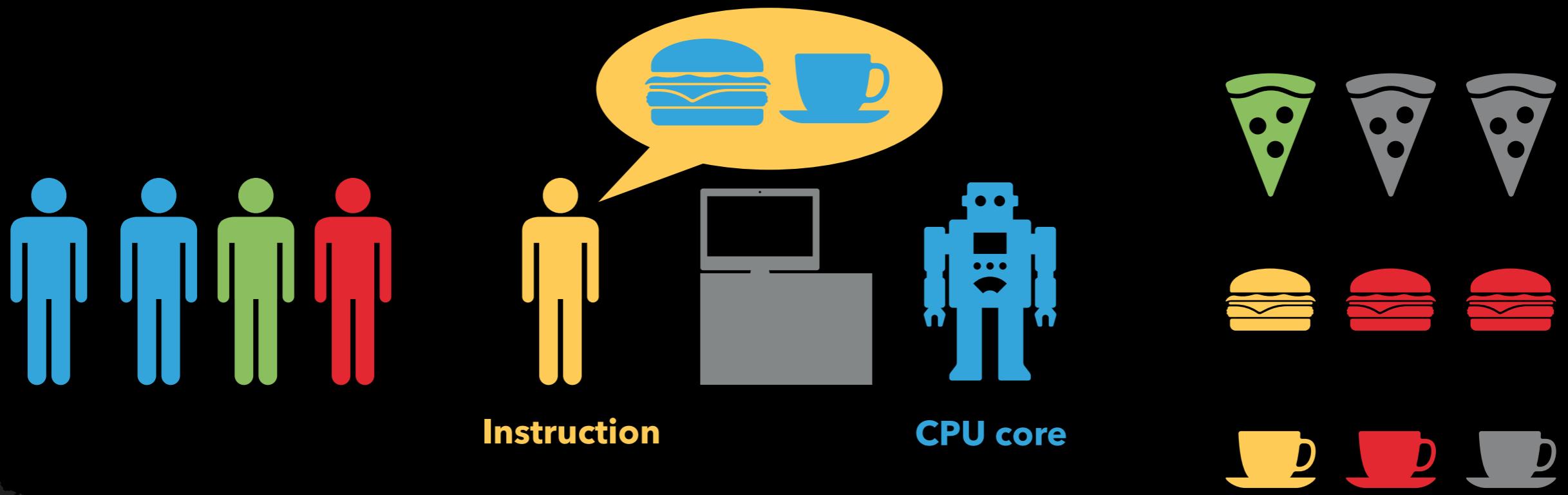
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

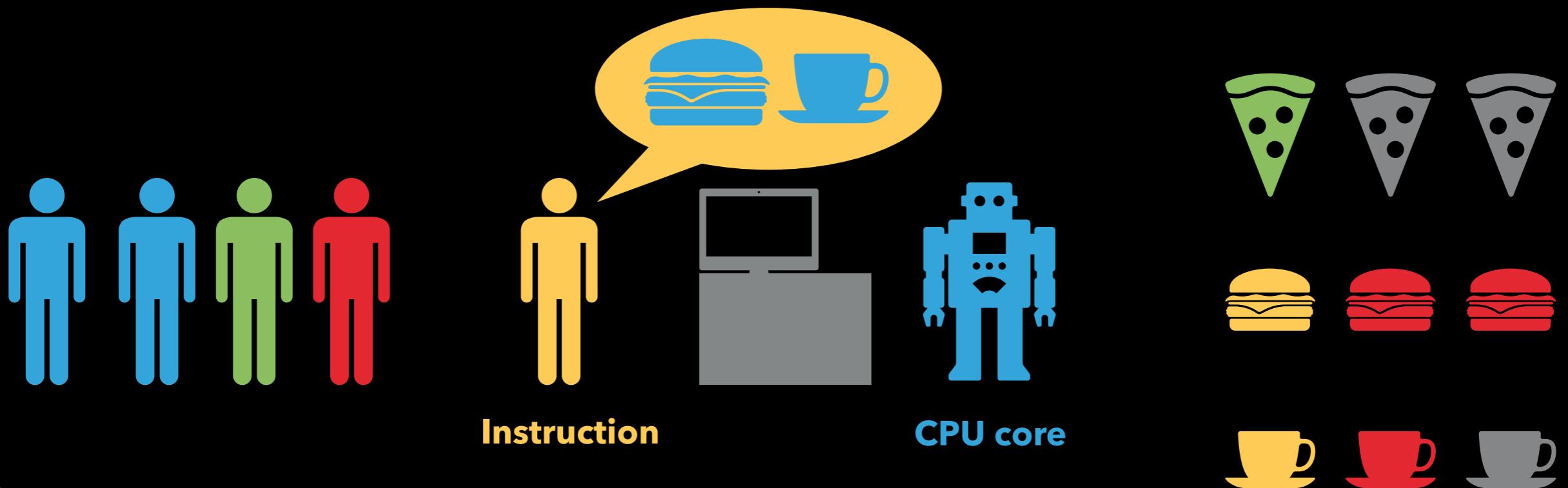


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



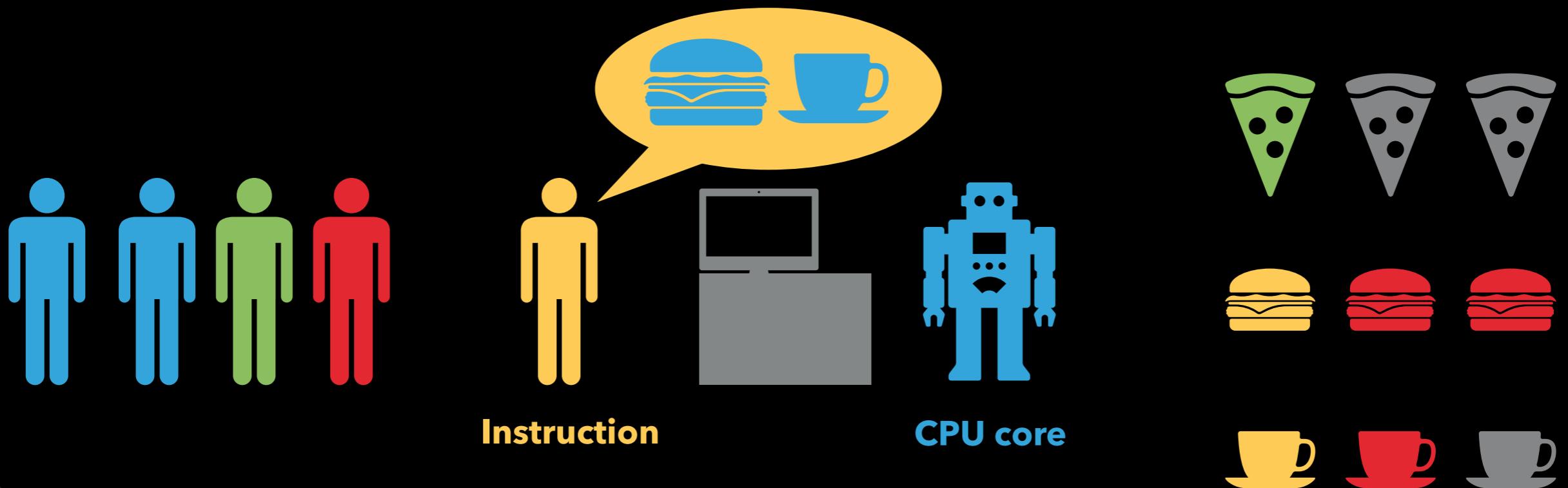
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



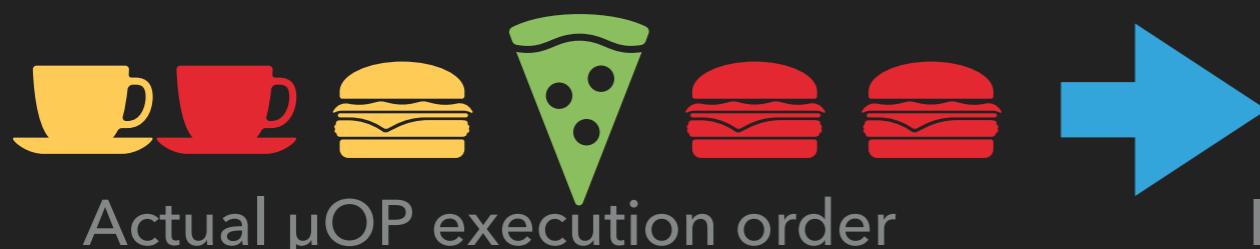
Actual μOP execution order

## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

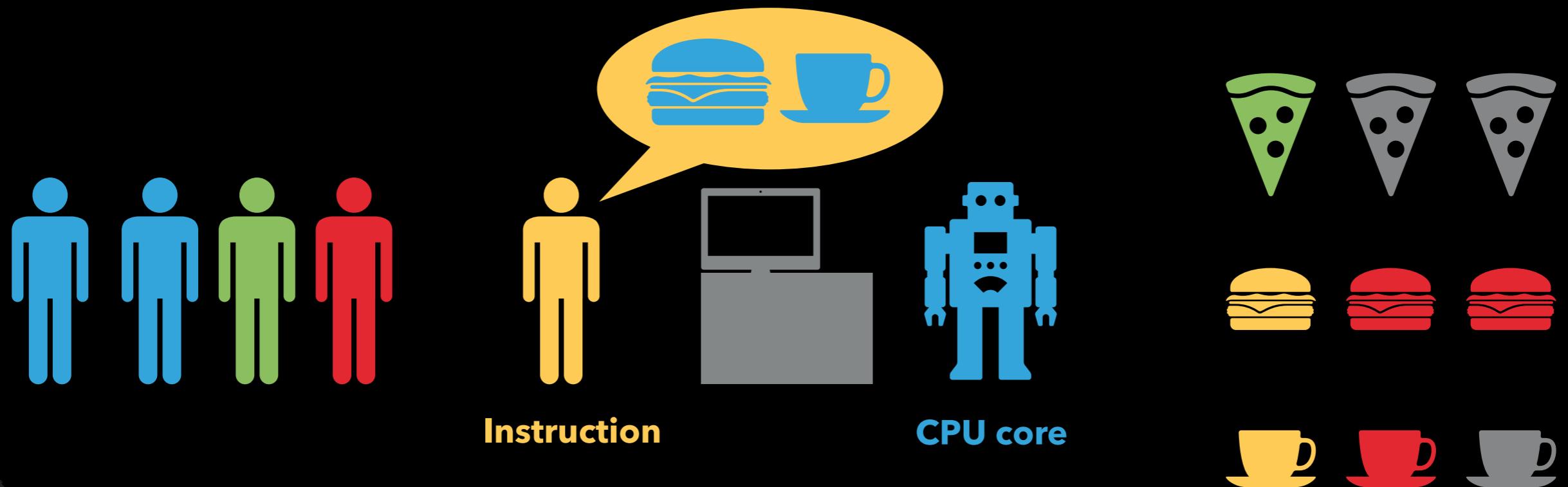


The **green** instruction will finish before the **red** instruction.

The **CPU** ensures that **red** is seen before **green**.

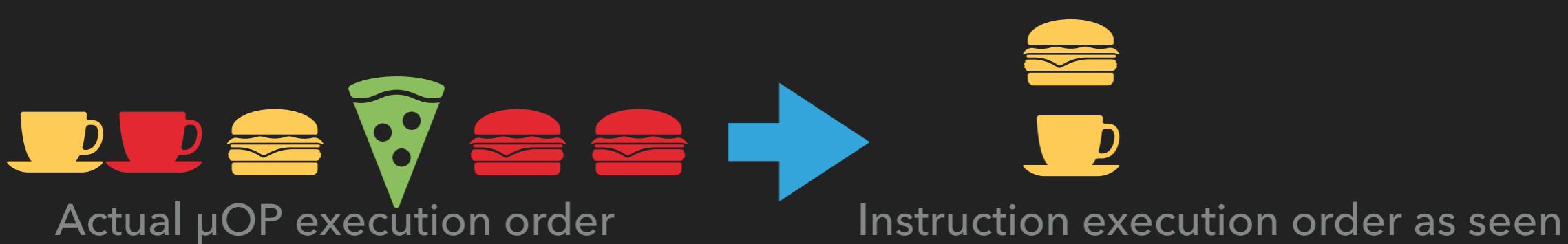


## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

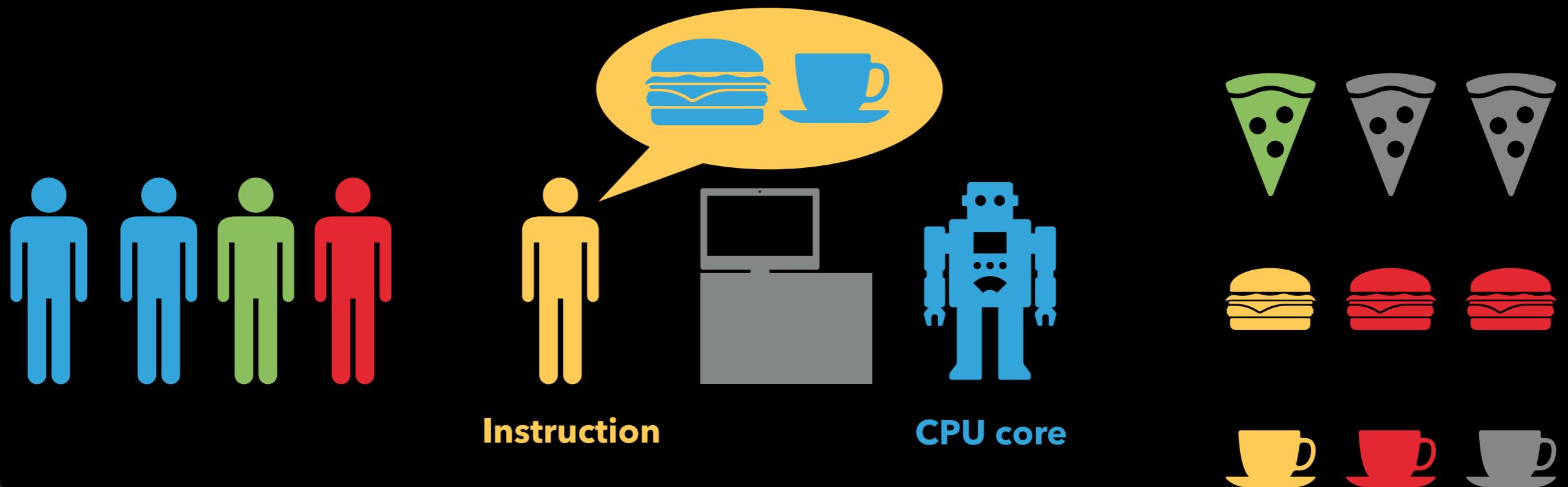


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

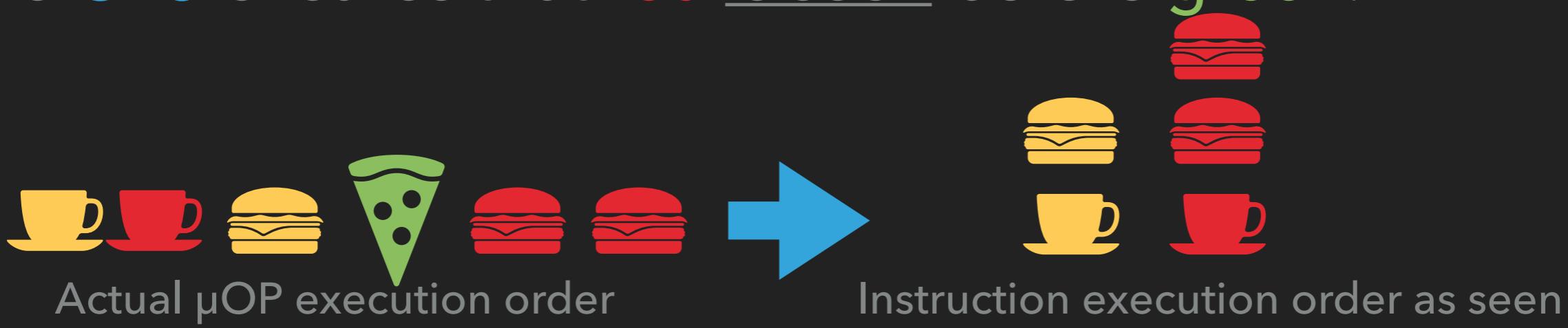


## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

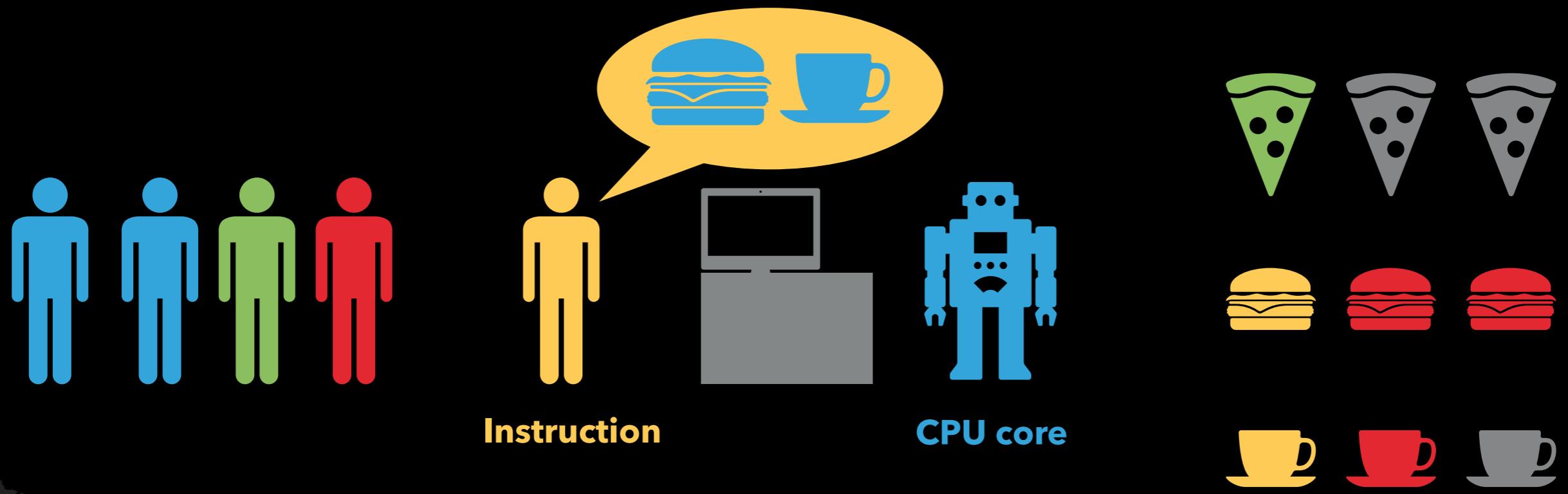


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

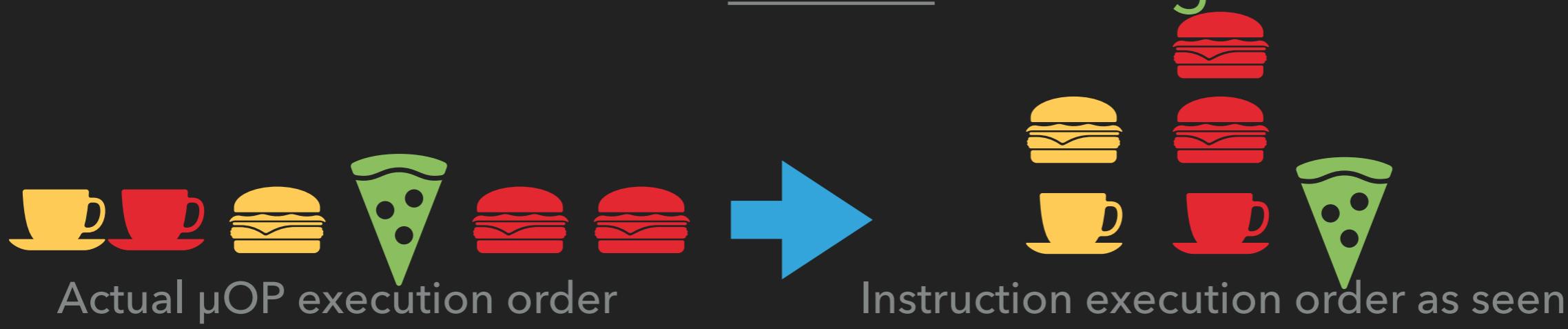


## CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



# WHY DID MELTDOWN & SPECTRE HAPPEN?

# WHY DID MELTDOWN & SPECTRE HAPPEN?

- ▶ Accident, malice, incompetence?

# WHY DID MELTDOWN & SPECTRE HAPPEN?

- ▶ Accident, malice, incompetence?
- ▶ Or just a very advanced and complex system designed to be as fast as possible?

# WHY DID MELTDOWN & SPECTRE HAPPEN?

- ▶ Accident, malice, incompetence?
- ▶ Or just a very advanced and complex system designed to be as fast as possible?
- ▶ Lessons learned: Complex systems have complex side effects!

# WHY DID MELTDOWN & SPECTRE HAPPEN?

- ▶ Accident, malice, incompetence?
- ▶ Or just a very advanced and complex system designed to be as fast as possible?
- ▶ Lessons learned: Complex systems have complex side effects!

...  
MISUNDERSTANDINGS AND NEGLECT  
CREATE MORE CONFUSION IN THIS WORLD  
THAN TRICKERY AND MALICE. AT ANY RATE, THE  
LAST TWO ARE CERTAINLY MUCH LESS  
FREQUENT.

Goethe's The Sorrows of Young Werther



OUT OF ORDER  
EXECUTION

---

MELTDOWN

# MELTDOWN



Meltdown basically works like this:

- READ secret from forbidden address
- Stash away secret before CPU detects wrongdoing
- Retrieve secret

## MELTDOWN

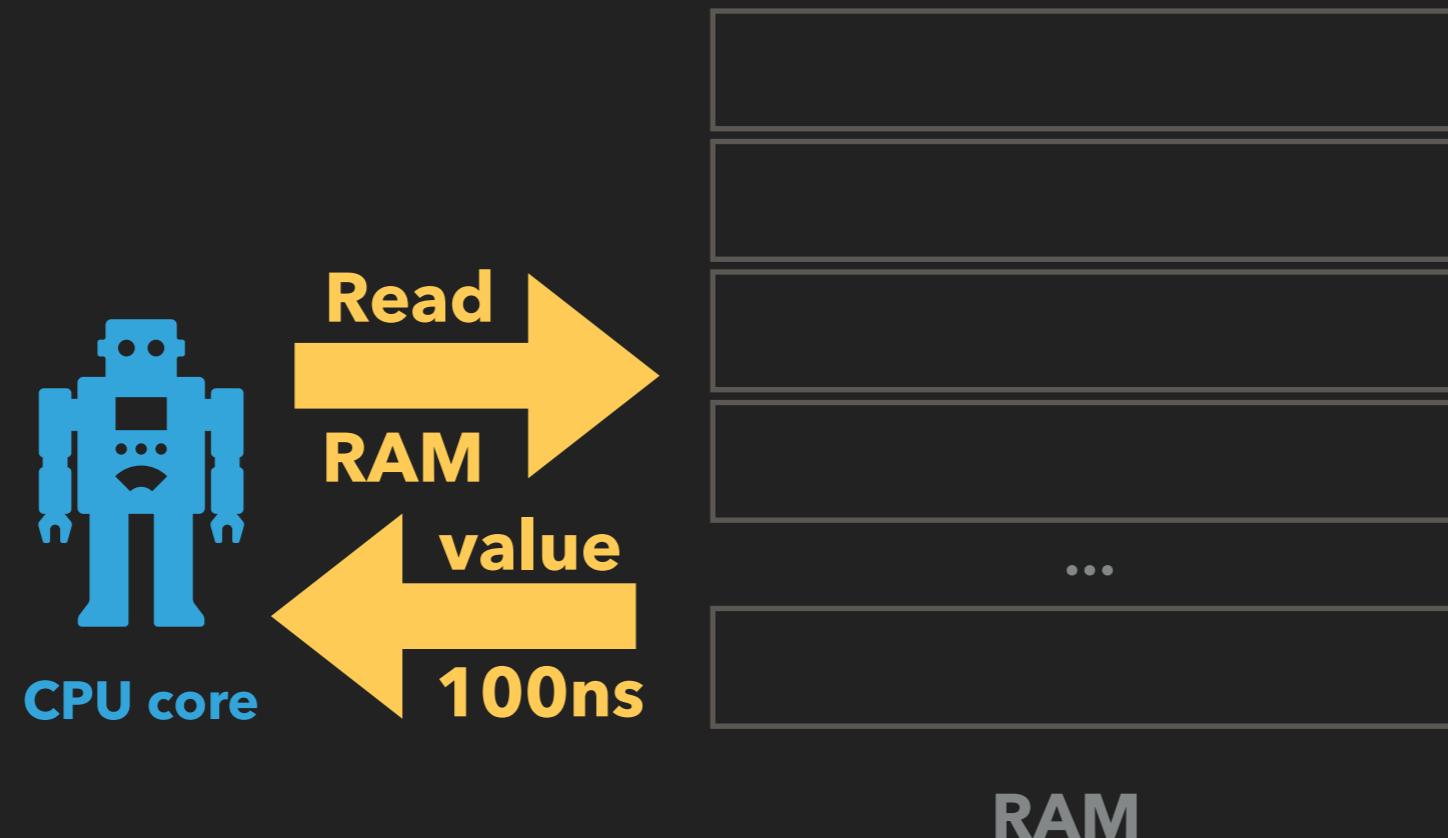
“Stashing” and “retrieving” the secret works via *side channels*.



Side channels are *observable side effects* of actions.

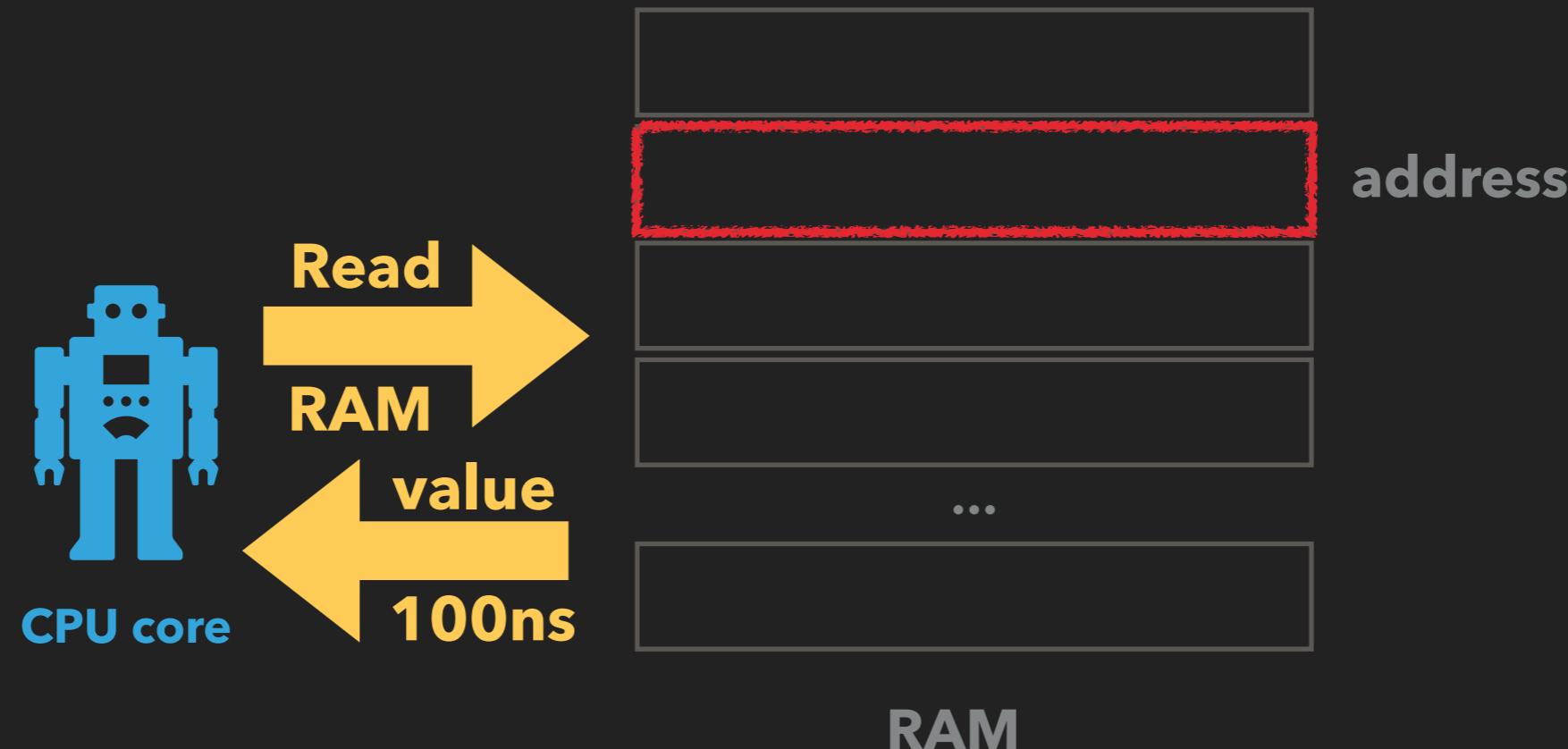
- READ secret from forbidden address
- Stash away secret by caching a memory location that depends on the secret
- Retrieve secret by finding which memory location is cached

## MELTDOWN: STASHING AWAY - SIDECHANNEL



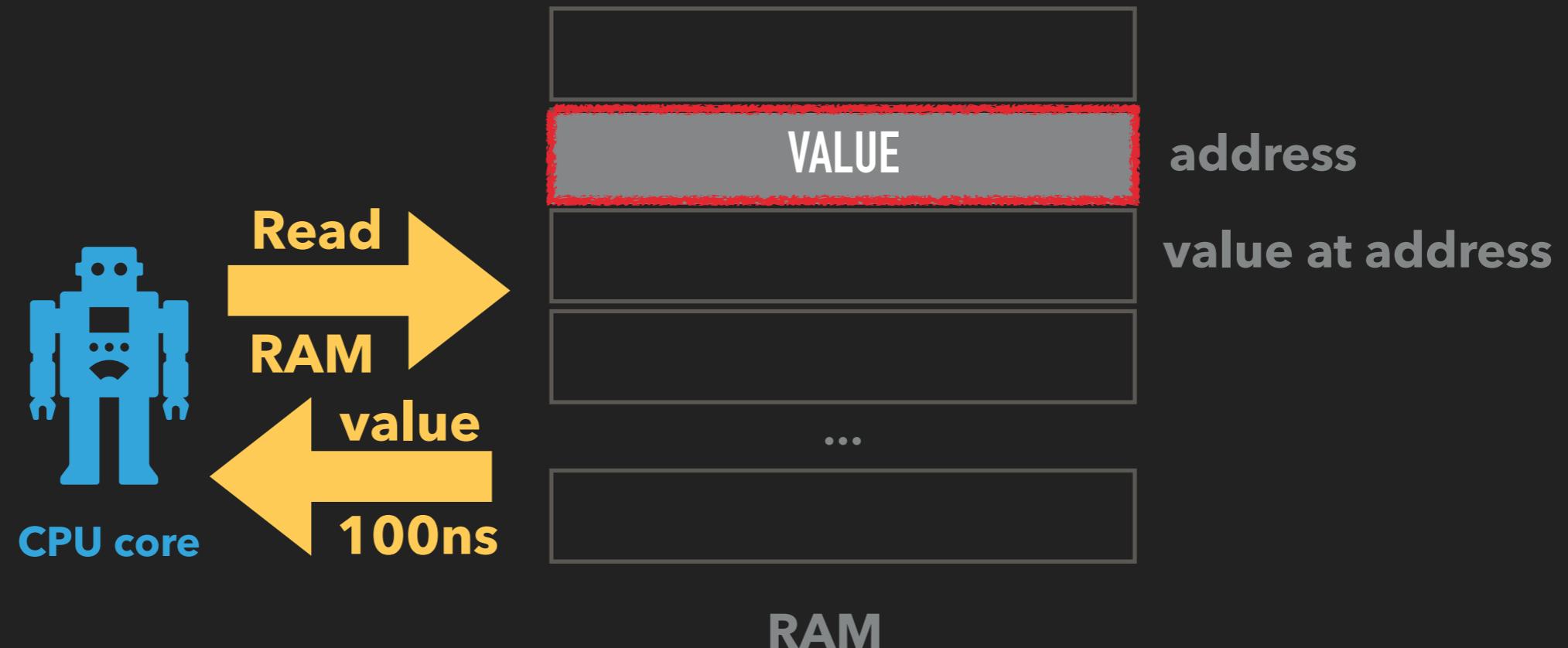
- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

## MELTDOWN: STASHING AWAY - SIDECHANNEL



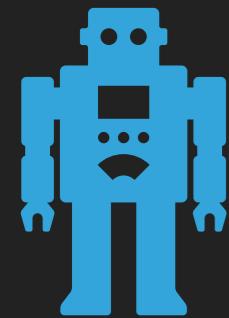
- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

# MELTDOWN: STASHING AWAY - SIDECHANNEL

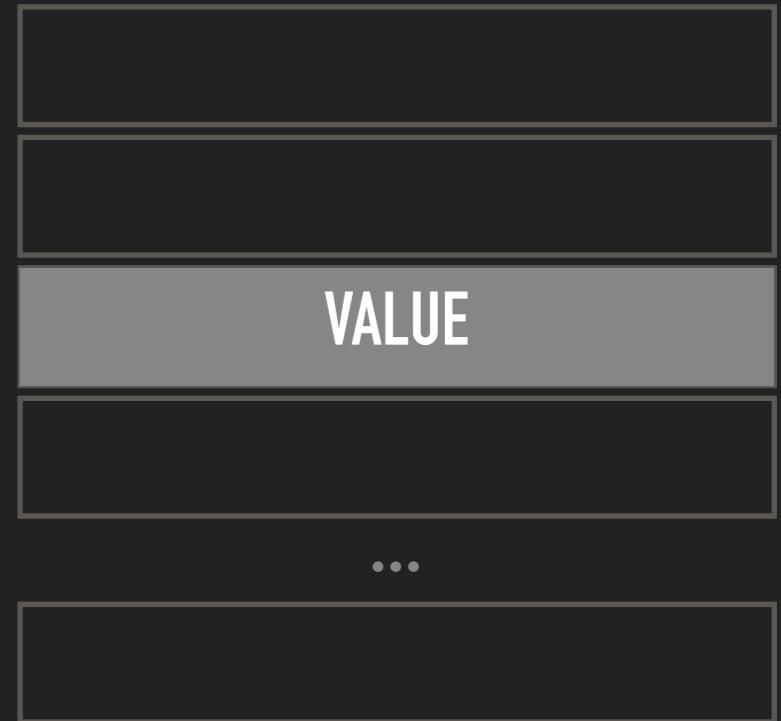
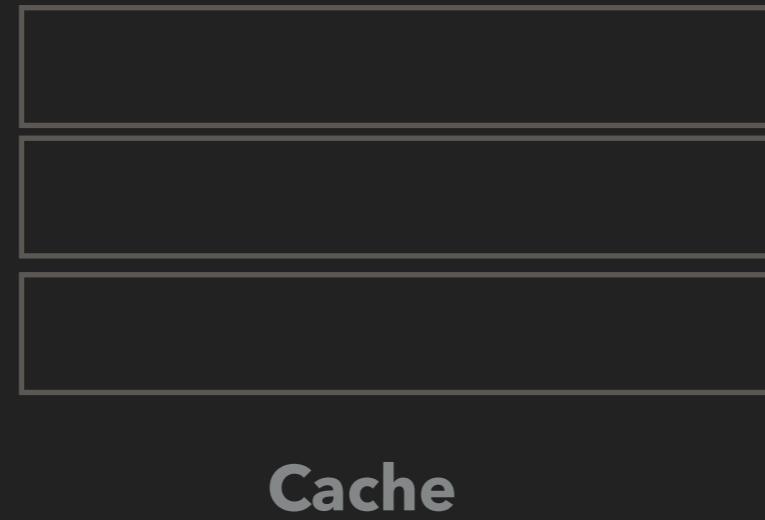


- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

# MELTDOWN: STASHING AWAY - SIDECHANNEL



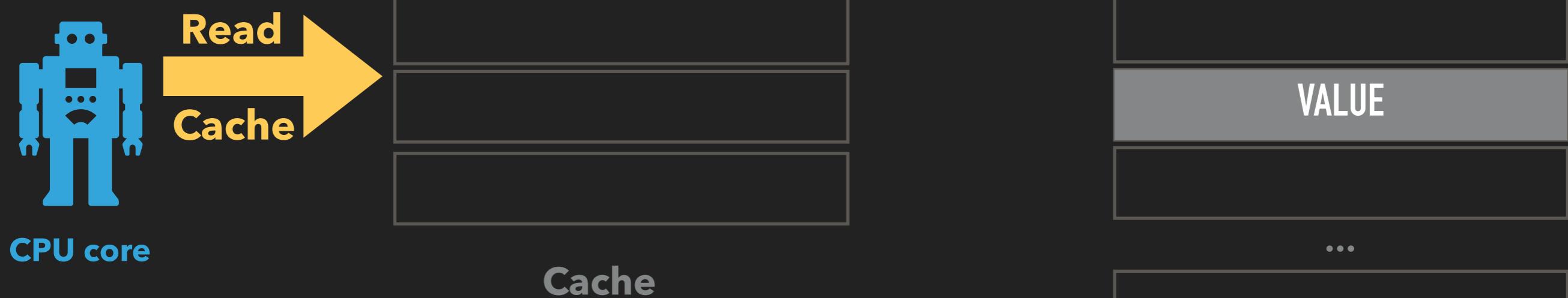
CPU core



- ▶ Reading one byte stalls the CPU for hundreds of µOPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up “*what is the value at address X?*”. This is called “*(address) X is cached*”

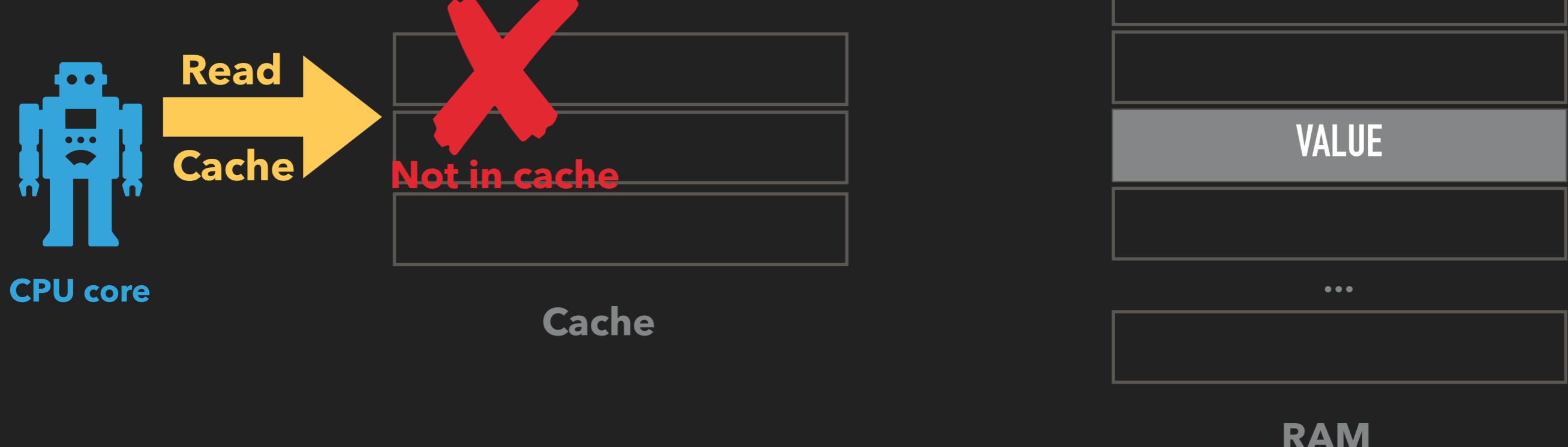
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "(address) X is cached"

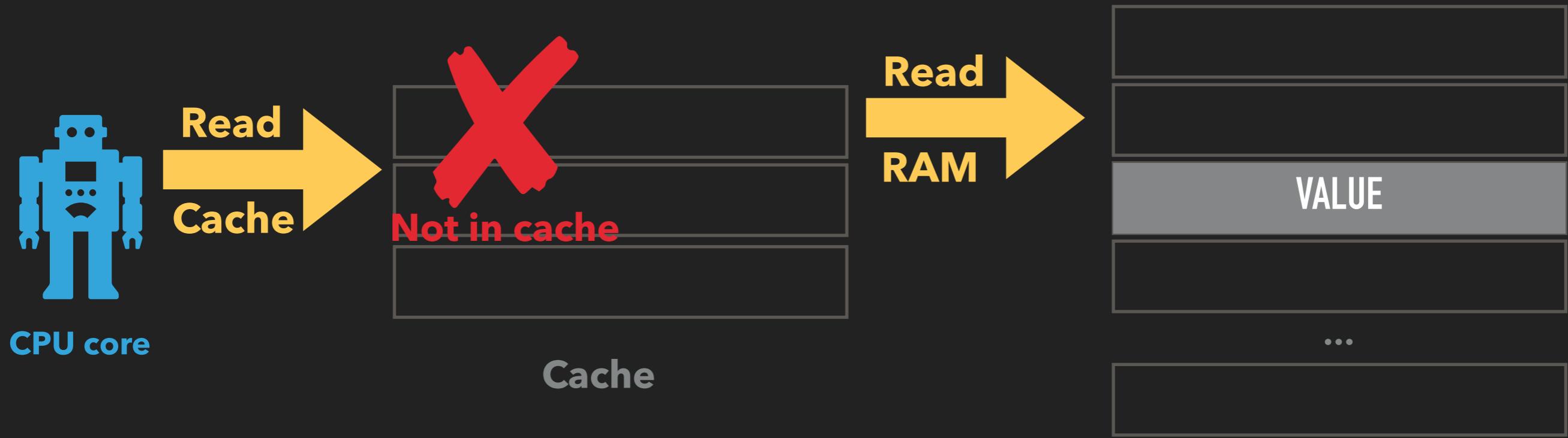
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

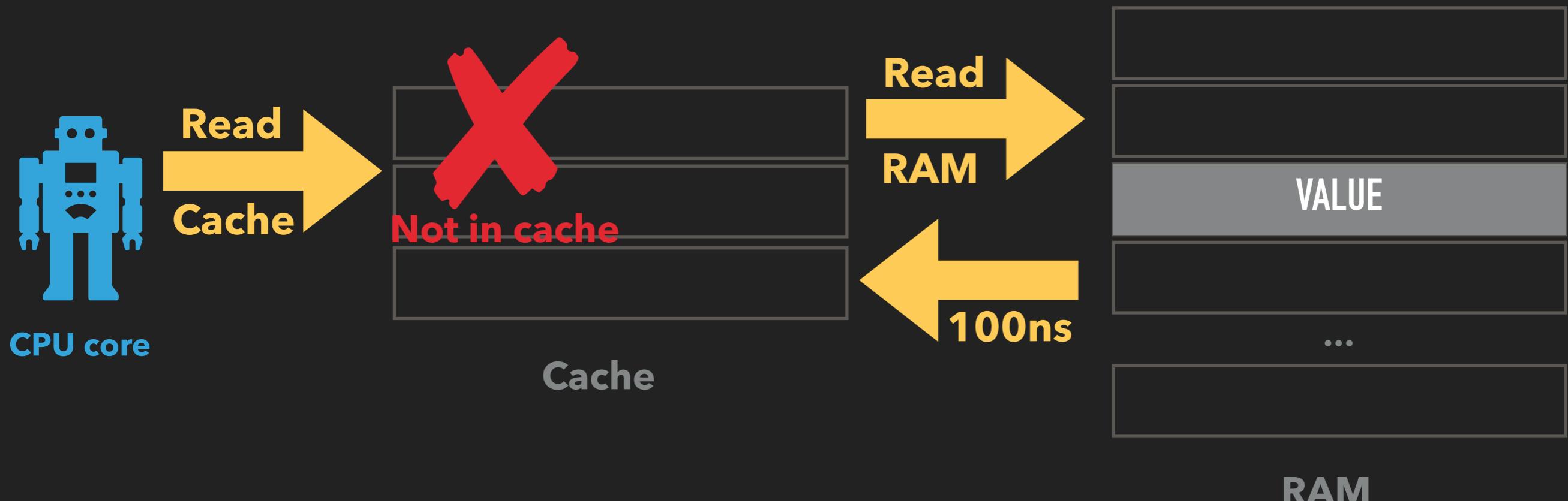
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of µOPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up “*what is the value at address X?*”. This is called “*(address) X is cached*”

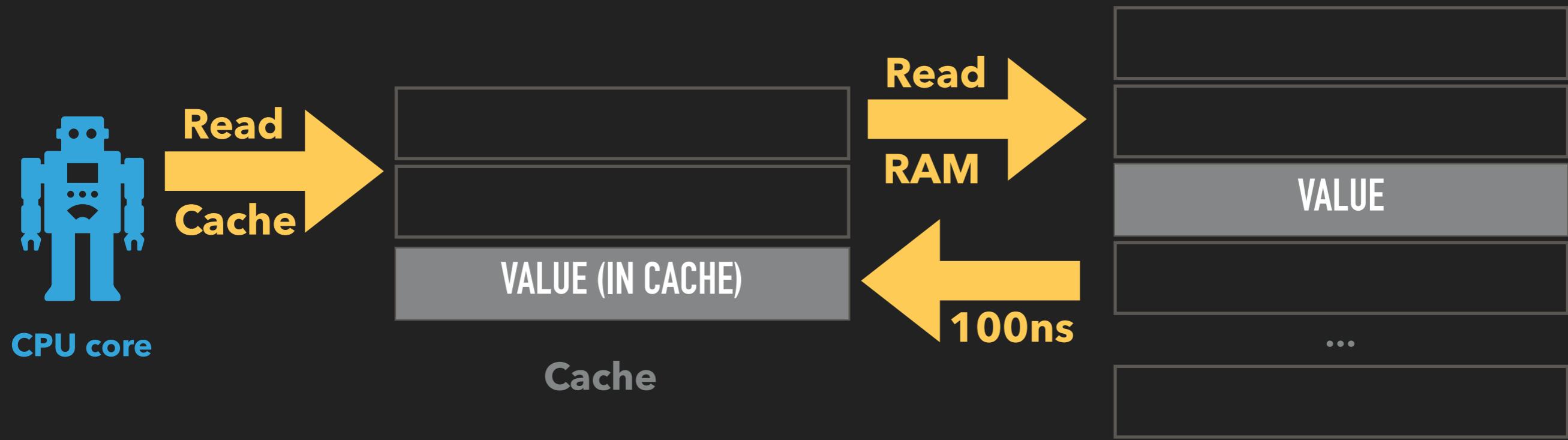
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "what is the value at address X?". This is called "(address) X is cached"

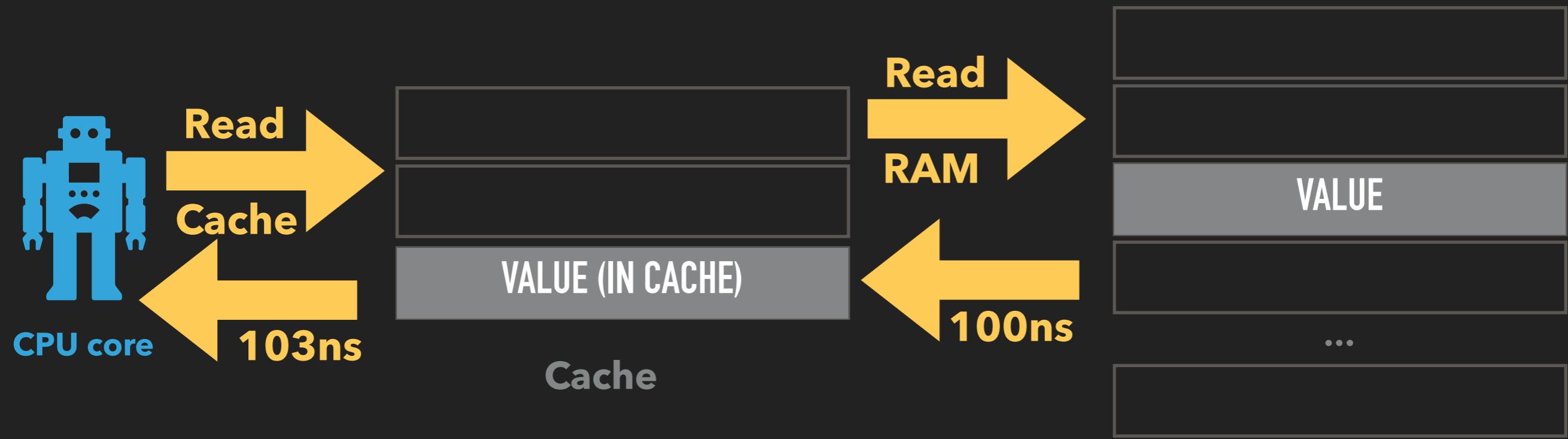
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up “*what is the value at address X?*”. This is called “*(address) X is cached*”

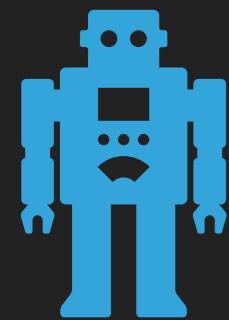
## MELTDOWN: STASHING AWAY - SIDECHANNEL



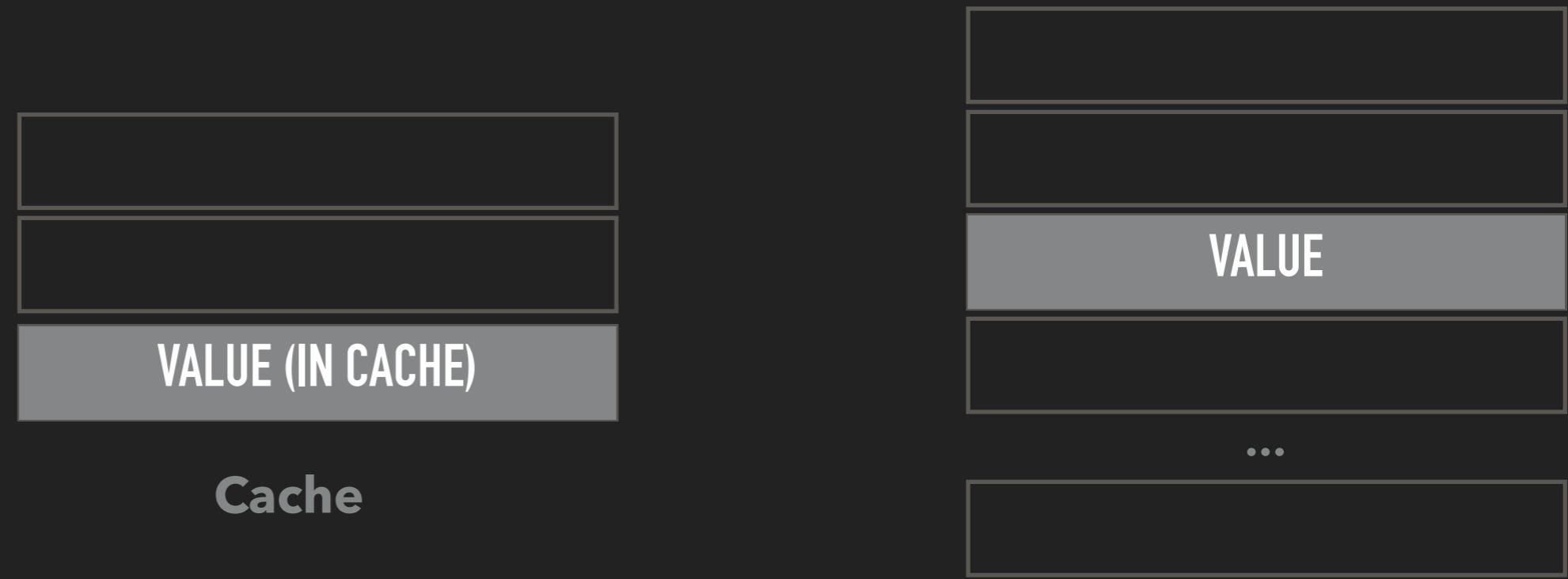
- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

# MELTDOWN: STASHING AWAY - SIDECHANNEL



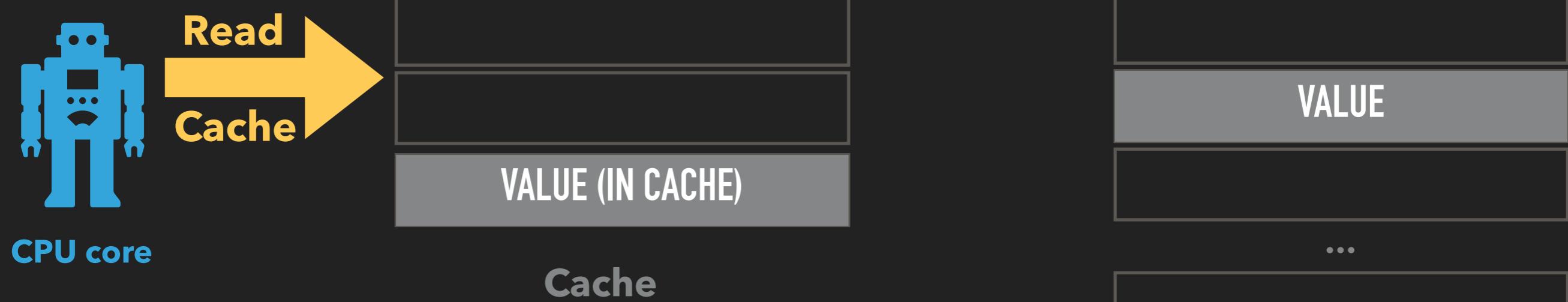
CPU core



- ▶ Reading one byte stalls the CPU for hundreds of µOPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

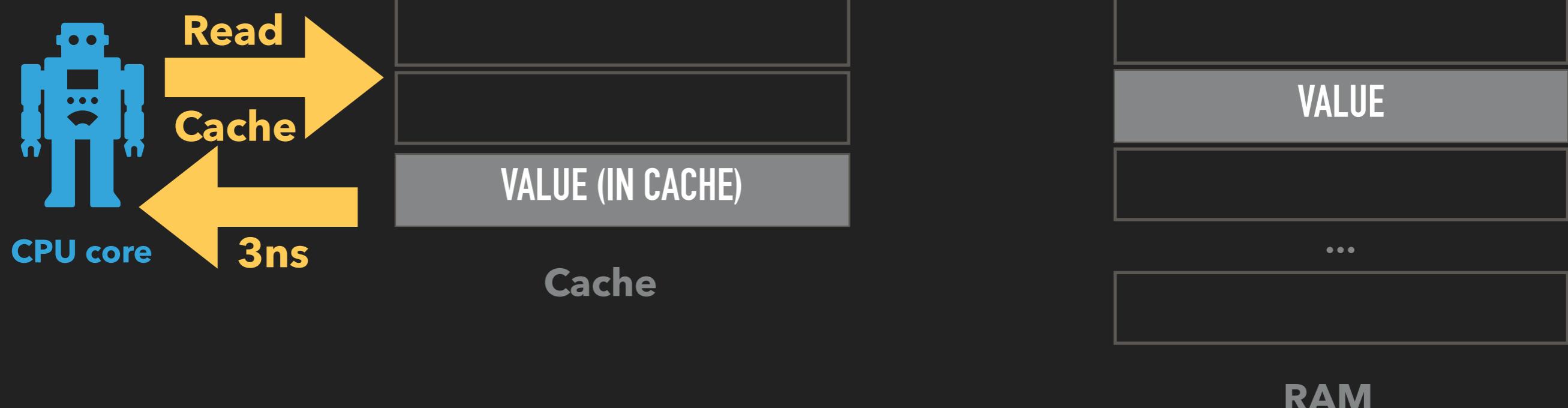
## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of µOPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

## MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of  $\mu$ OPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

The cache speeds up "*what is the value at address X?*". This is called "*(address) X is cached*"

## “READ” INSTRUCTION

For a CPU the “READ value from memory at 4711” instruction looks like this ( $\mu$ OPs):

1. Check that program may read from address
2. Store the value at address in register<sup>1</sup>

If 1 fails the program is aborted.

This can be handled by the program.

<sup>1</sup> Register: The CPUs scratchpad

## “READ” INSTRUCTION

For a CPU the “READ value from memory at 4711” instruction looks like this ( $\mu$ OPs):

1. Check that program may read from address
2. Store the value at address in register<sup>1</sup>

If 1 fails the program is aborted.

This can be handled by the program.

In our burger example:

1. Customer orders a burger
2. Customer has not enough money
3. Customer does not get his burger

<sup>1</sup> Register: The CPUs scratchpad

## MELTDOWN: READING FORBIDDEN DATA



Meltdown basically works like this:

- READ secret from forbidden address
- 1. Check that program may read from address
- 2. Store the read value in register
- Stash away secret
- 1 *Magic*
- Retrieve secret (*later*)

µOPs: 1 2 1

## MELTDOWN: READING FORBIDDEN DATA



μOPs ordered by *instruction*

1 Check access

2 Read into register

1 *Magic*

μOPs ordered by *execution*

2 Read into register

1 *Magic*

1 Check access

The re-ordering on the right happens, when the “forbidden data” is already cached (because cache access is so fast).

## MELTDOWN: READING FORBIDDEN DATA



μOPs ordered by *instruction*

1 Check access

2 Read into register

1 *Magic*

μOPs ordered by *execution*

2 Read into register

1 *Magic*

1 Check access

The re-ordering on the right happens, when the “forbidden data” is already cached (because cache access is so fast).

Reordering is not a problem because the CPU will ensure that  is only seen *iff*  succeeds.

## MELTDOWN: READING FORBIDDEN DATA



μOPs ordered by *instruction*

1 Check access

2 Read into register

1 *Magic*

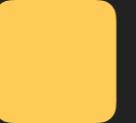
μOPs ordered by *execution*

2 Read into register

1 *Magic*

1 Check access

The re-ordering on the right happens, when the “forbidden data” is already cached (because cache access is so fast).

Reordering is not a problem because the CPU will ensure that  is only seen *iff*  succeeds.

Unless  is able to hide the secret in such a way that the attacker can find it later.

## MELTDOWN: READING FORBIDDEN DATA



μOPs ordered by *instruction*

1 Check access

2 Read into register

1 Magic

μOPs ordered by *execution*

2 Read into register

1 Magic

1 Check access

The re-ordering on the right happens, when the “forbidden data” is already cached (because cache access is so fast).

Reordering is not a problem because that   is only seen *iff*   succeeds.

Unless   is able to hide the secret, the attacker can find it later.

In our burger example:

1. Customer orders a burger
2. Customer gets his burger
3. Customer has not enough money
4. Customer runs away with burger

# MELTDOWN



For Meltdown two actors are needed

The **spy** and a **collector**.

110011010      The **spy** will “steal” the secret and stash it away.  
010111010  
111100100      The CPU will kill him for accessing the secret  
000101101  
100110010      information.

Spy

110011010  
010111010  
111100100  
000101101  
100110010

The **collector** will find the stashed away secret.

Collector

## MELTDOWN: THE SIDECHANNEL (IDEA)

110011010

010111010

111100100

000101101

100110010

**Spy**

110011010

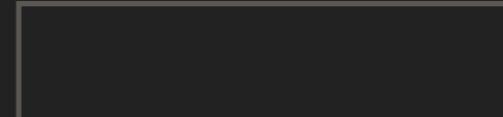
010111010

111100100

000101101

100110010

**Collector**



...

"IT'S A 1"

"IT'S A 2"

"IT'S A 3"

...

SECRET ("3")

**Places**



## MELTDOWN: THE SIDECHANNEL (IDEA)

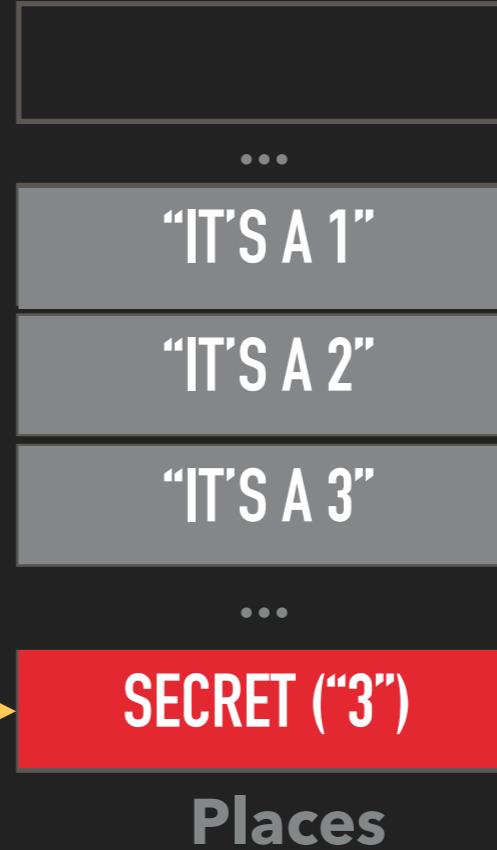
110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**

■ **Spy** will read the **secret**



## MELTDOWN: THE SIDECHANNEL (IDEA)

110011010  
010111010  
111100100  
000101101  
100110010

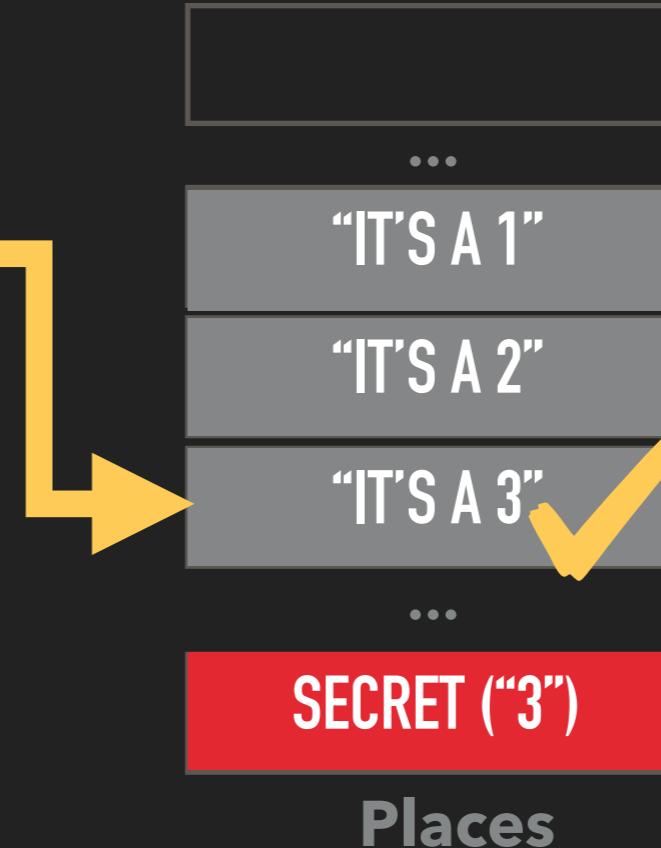
**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**

█ **Spy** will read the **secret**

█ Depending on the **value**, **Spy** will mark a grey block



## MELTDOWN: THE SIDECHANNEL (IDEA)

110011010  
010111010  
111100100  
00101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**

■ **Spy** will read the **secret**

■ Depending on the **value**, **Spy** will mark a grey block

■ CPU detects **Spys** access validation and terminates **Spy**



**Places**



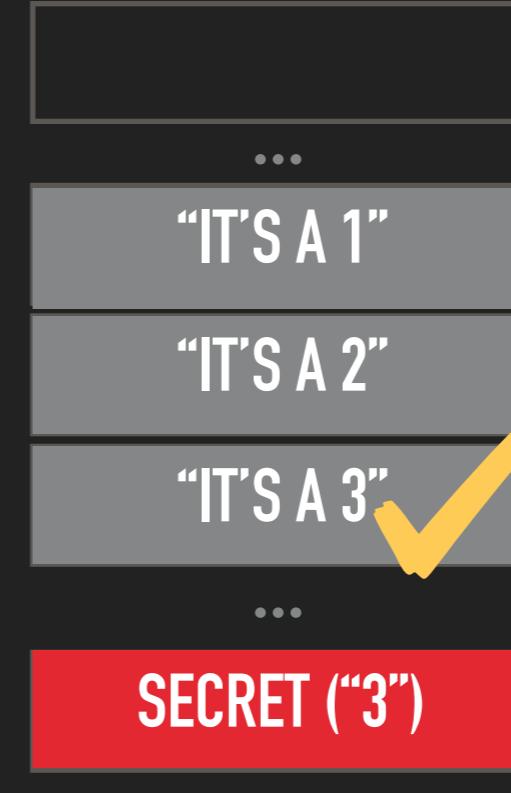
## MELTDOWN: THE SIDECHANNEL (IDEA)

110011010  
010111010  
111100100  
00101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



**Places**

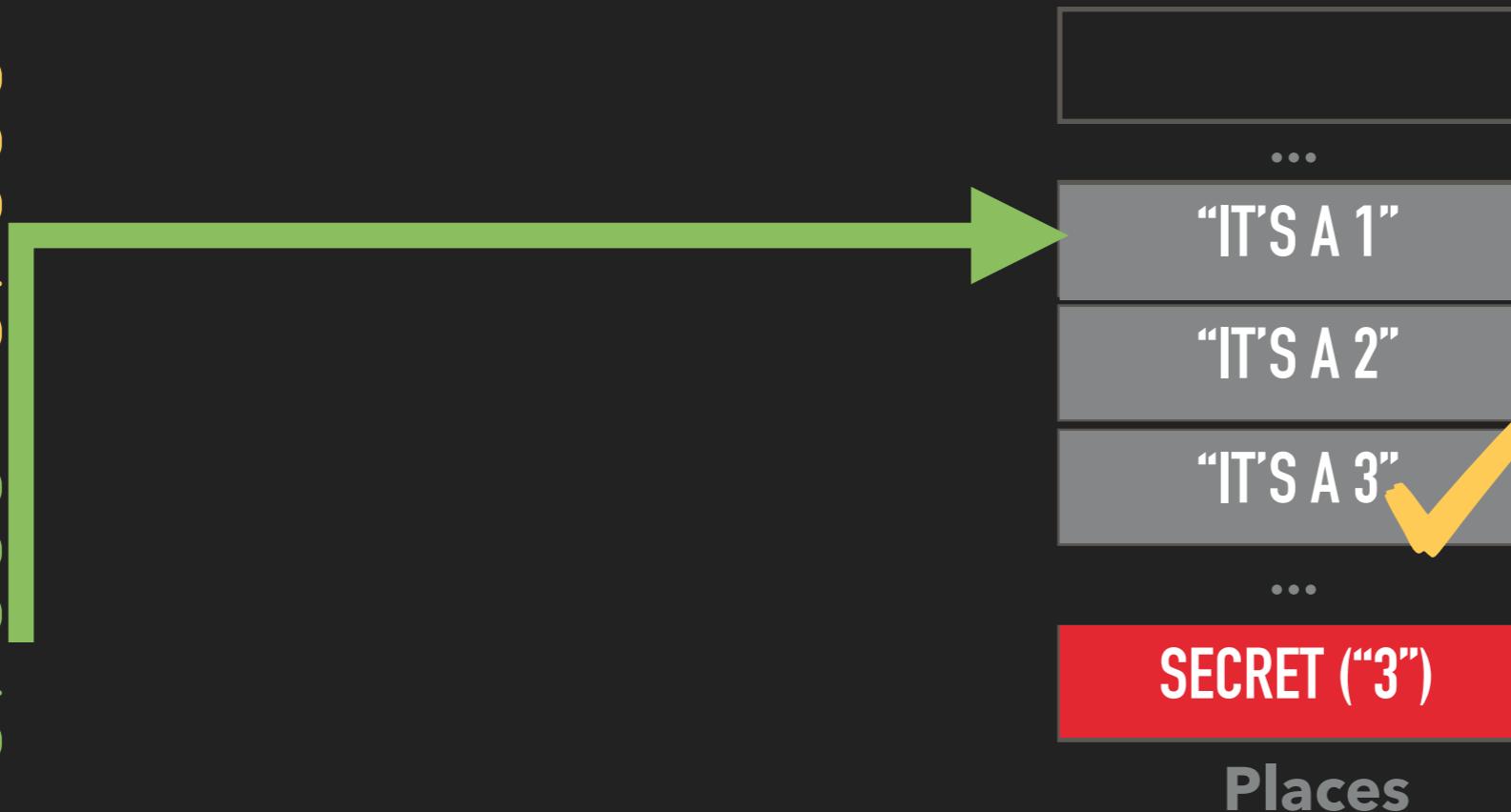
- **Spy** will read the **secret**
- Depending on the **value**, **Spy** will mark a grey block
- CPU detects **Spys** access validation and terminates **Spy**
- **Collector** now looks for **Spys** mark in all grey blocks



## MELTDOWN: THE SIDECHANNEL (IDEA)



110011010  
010111010  
111100100  
00101101  
100110010  
**Spy**  
110011010  
010111010  
111100100  
000101101  
100110010  
**Collector**

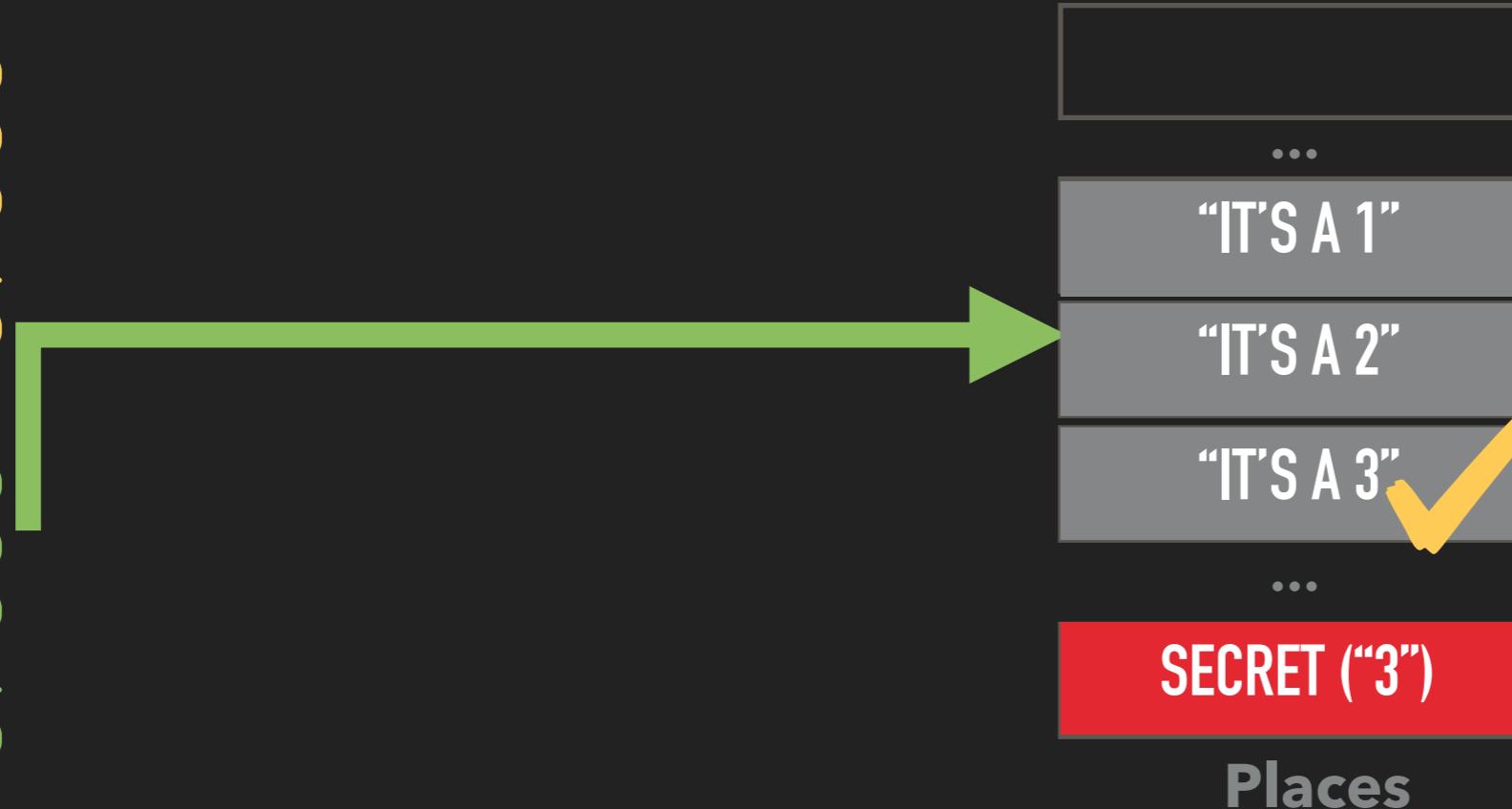


- **Spy** will read the **secret**
- Depending on the **value**, **Spy** will mark a grey block
- CPU detects **Spys** access validation and terminates **Spy**
- **Collector** now looks for **Spys** mark in all grey blocks

## MELTDOWN: THE SIDECHANNEL (IDEA)



110011010  
010111010  
111100100  
00101101  
100110010  
**Spy**  
110011010  
010111010  
111100100  
000101101  
100110010  
**Collector**



- **Spy** will read the **secret**
- Depending on the **value**, **Spy** will mark a grey block
- CPU detects **Spys** access validation and terminates **Spy**
- **Collector** now looks for **Spys** mark in all grey blocks

## MELTDOWN: THE SIDECHANNEL (IDEA)

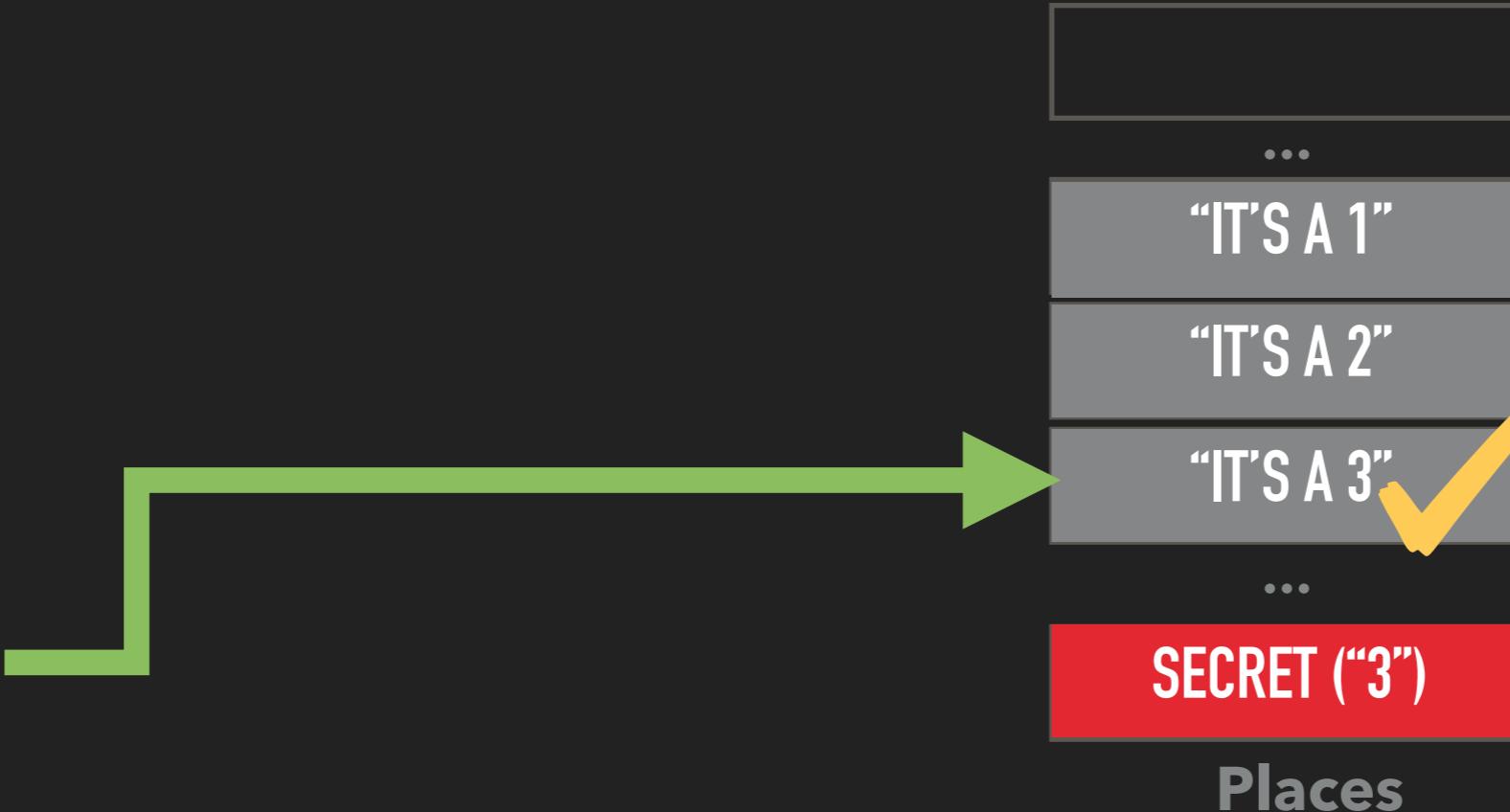


110011010  
010111010  
111100100  
001010101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



- **Spy** will read the **secret**
- Depending on the **value**, **Spy** will mark a grey block
- CPU detects **Spys** access validation and terminates **Spy**
- **Collector** now looks for **Spys** mark in all grey blocks

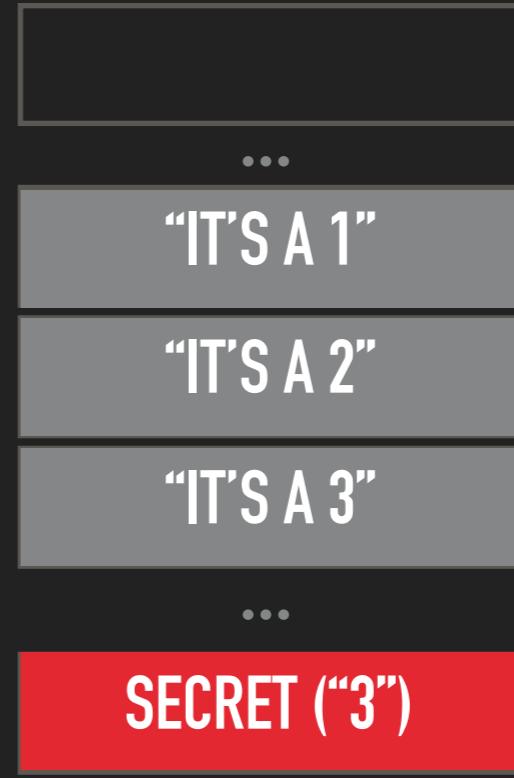
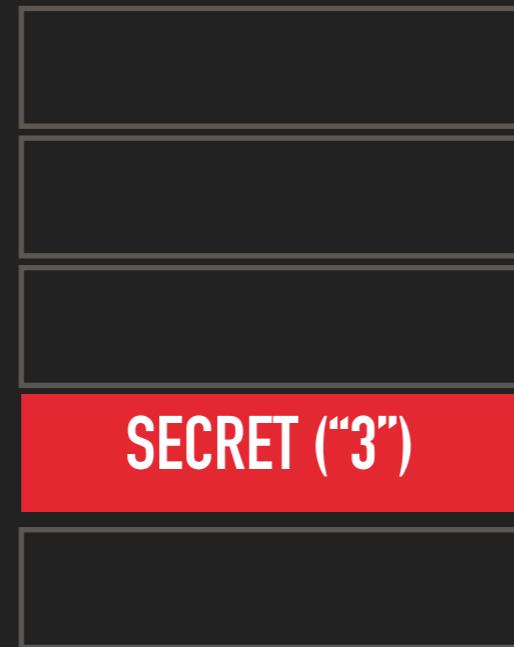
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



- ▶ Meltdown needs some preconditions



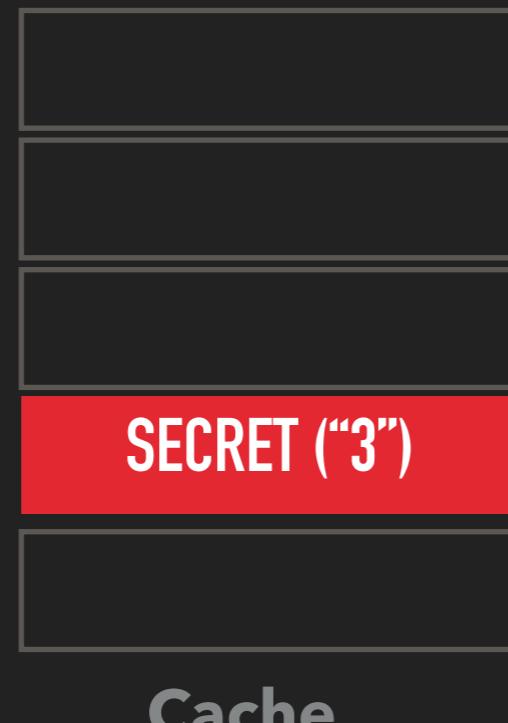
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



grey box:  
memory block  
tested by **Collector**



**RAM**

- ▶ Meltdown needs some preconditions



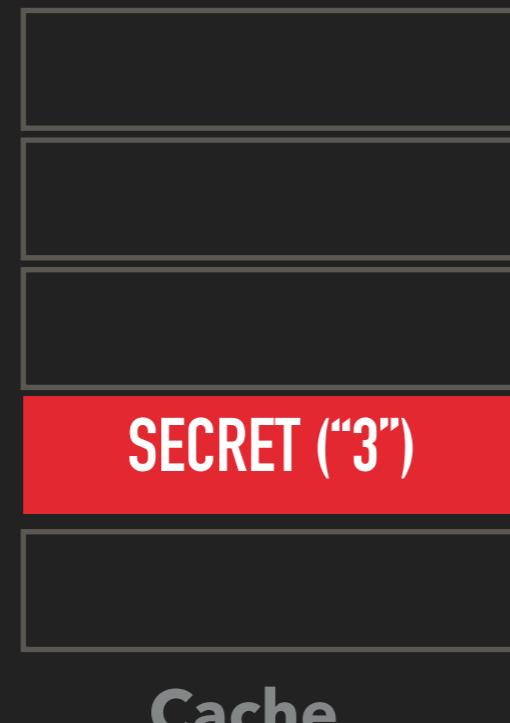
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

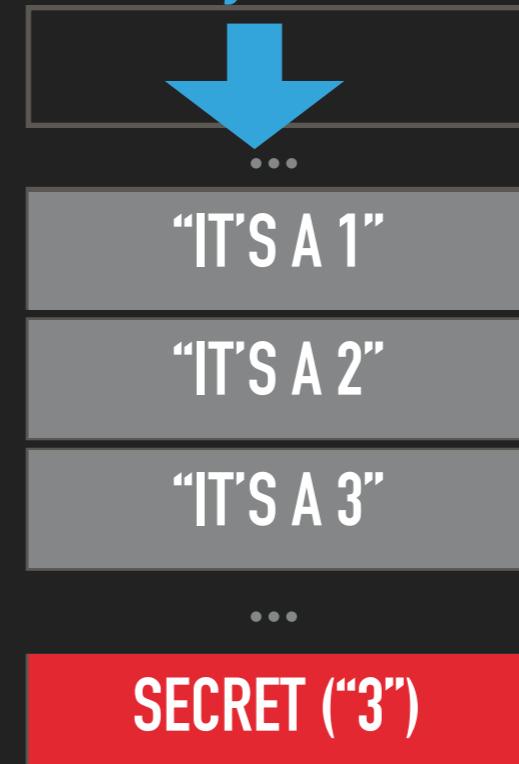
Spy

110011010  
010111010  
111100100  
000101101  
100110010

Collector



grey box:  
memory block  
tested by **Collector**



- ▶ Meltdown needs some preconditions
- ▶ The **secret** is in the cache (value: 3)



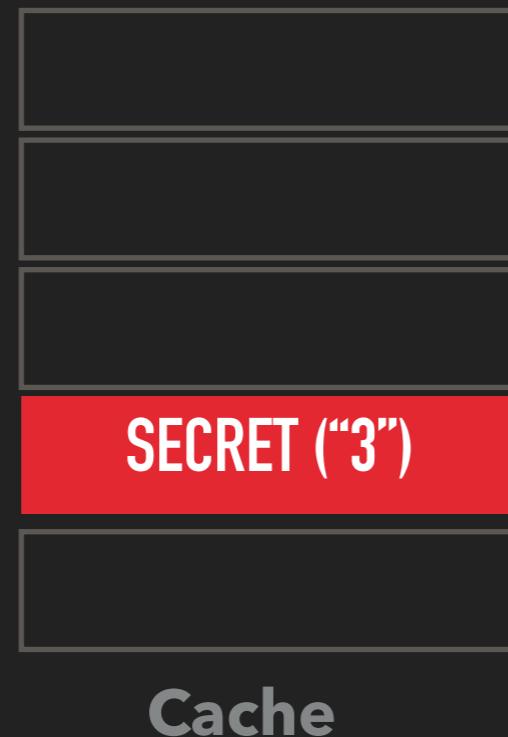
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



grey box:  
memory block  
tested by **Collector**



allowed to  
read?



**RAM**

- ▶ Meltdown needs some preconditions
- ▶ The **secret** is in the cache (value: 3)
- ▶ Both **Spy** and **Collector** can read grey memory blocks

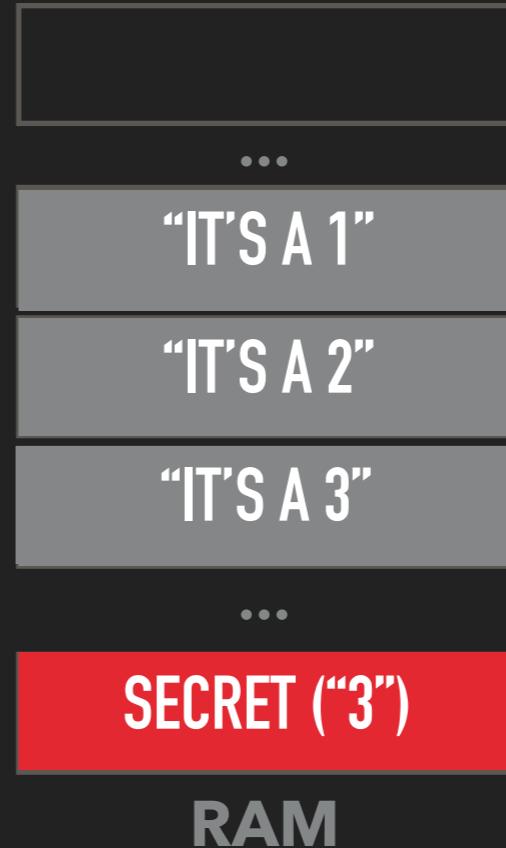
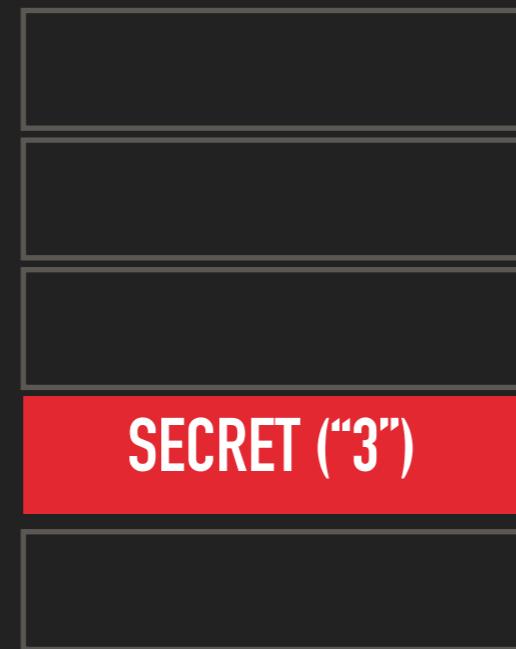
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**





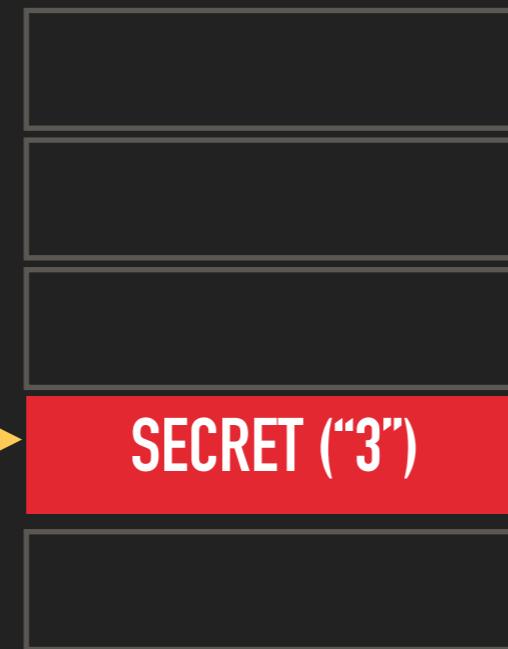
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
000101101  
100110010

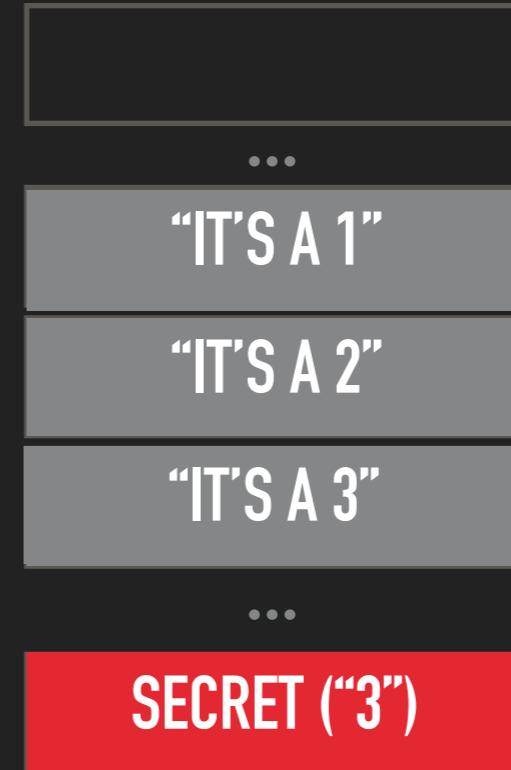
Spy

110011010  
010111010  
111100100  
000101101  
100110010

Collector



Cache



RAM

- 2 1. Spy will read the **secret**

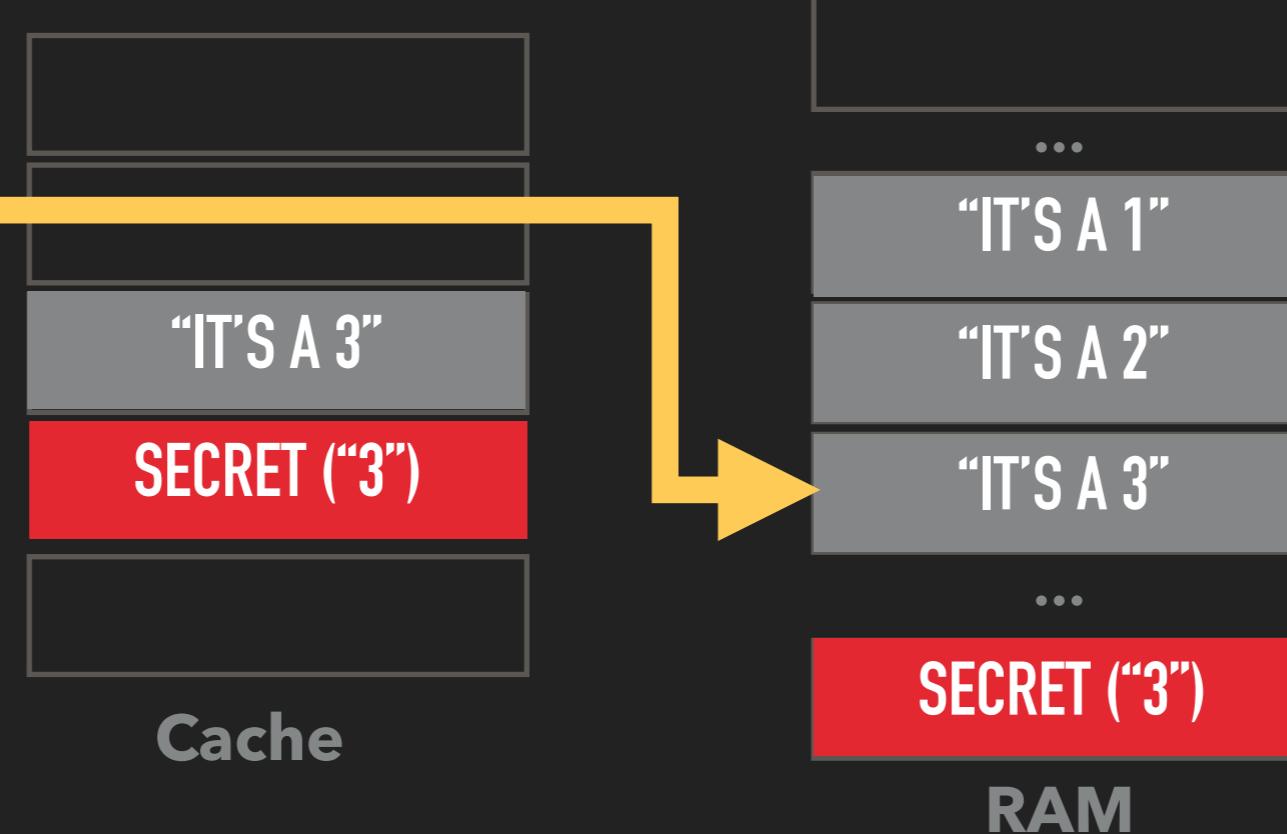
## MELTDOWN: THE ATTACK



110011010  
010111010  
111100100  
000101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010



**Collector**

- 2 1. **Spy** will read the **secret**
2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses

## MELTDOWN: THE ATTACK

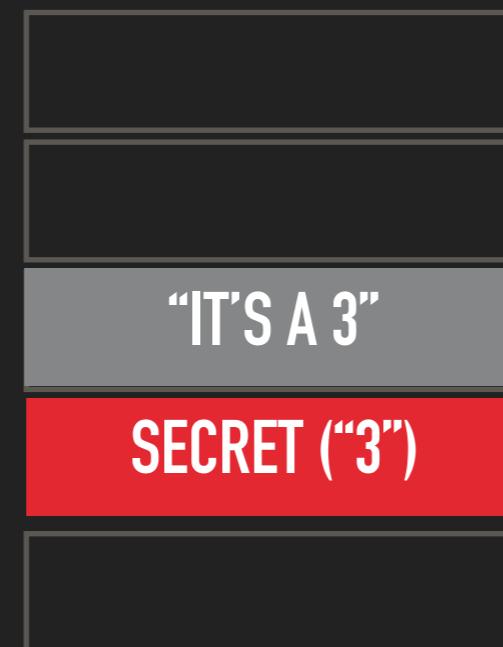


110011010  
010111010  
111100100  
00101101  
100110010

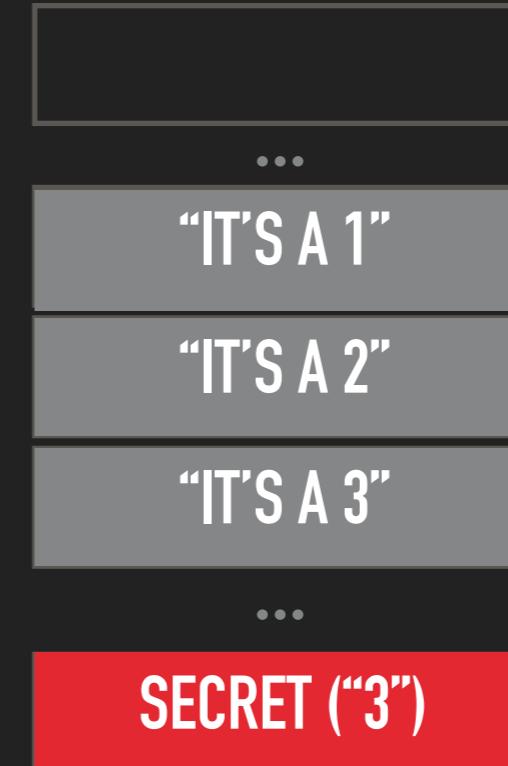
**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



**Cache**



**RAM**

- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses

## MELTDOWN: THE ATTACK

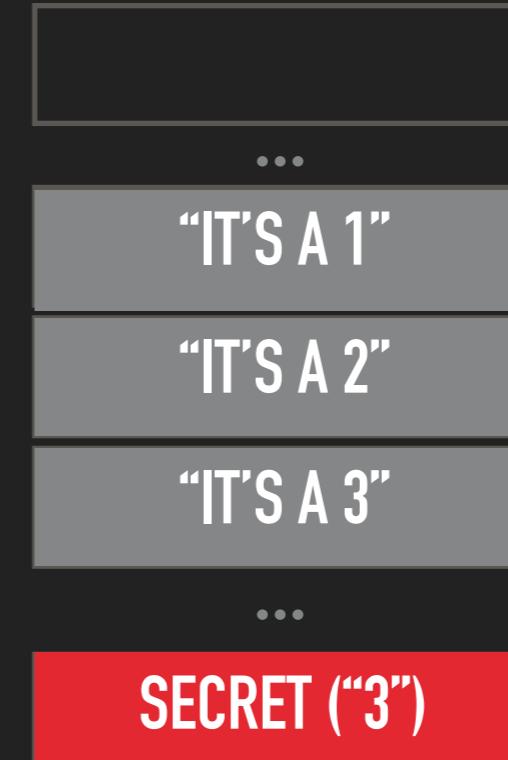
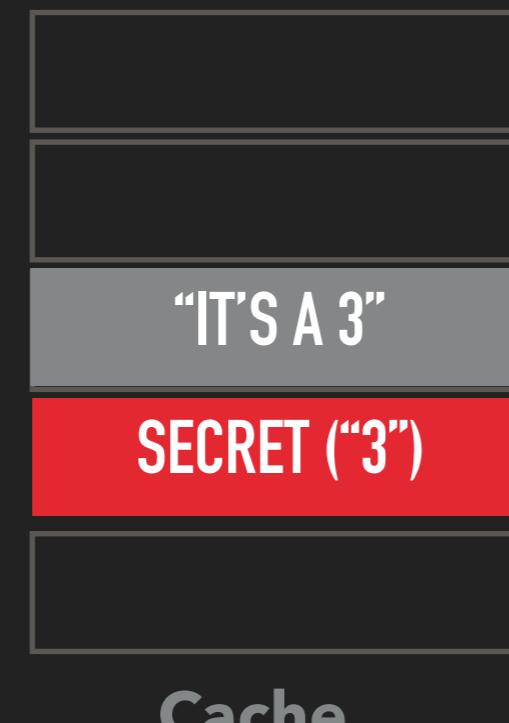


110011010  
010111010  
111100100  
00101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**

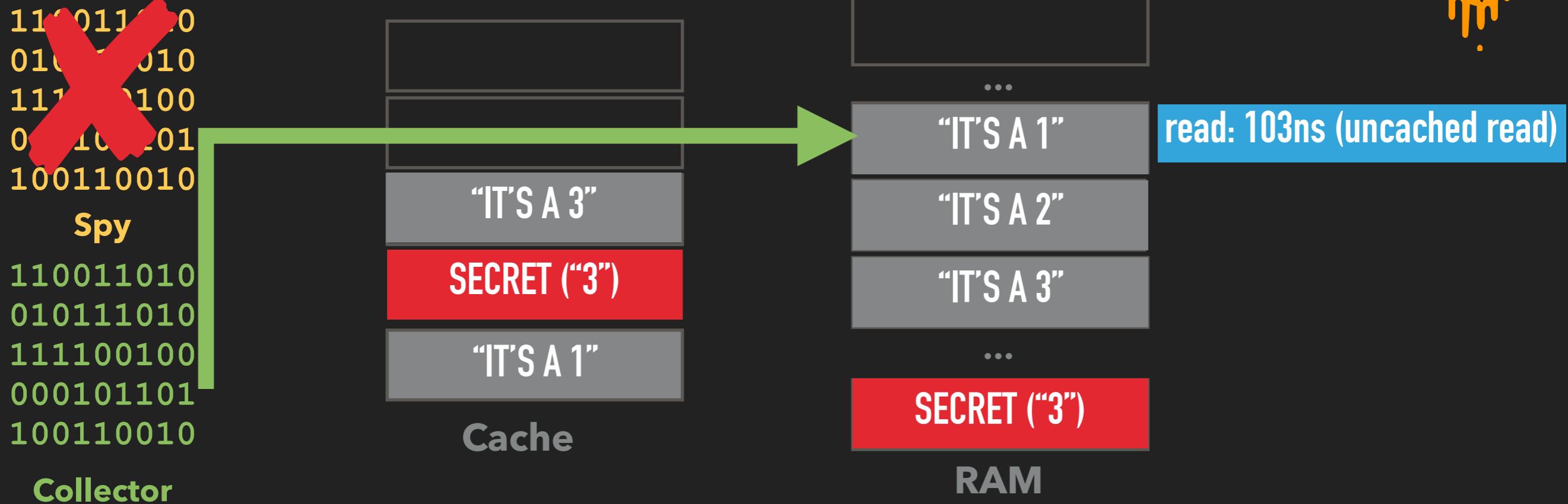


**RAM**

- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses

## MELTDOWN: THE ATTACK



- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses



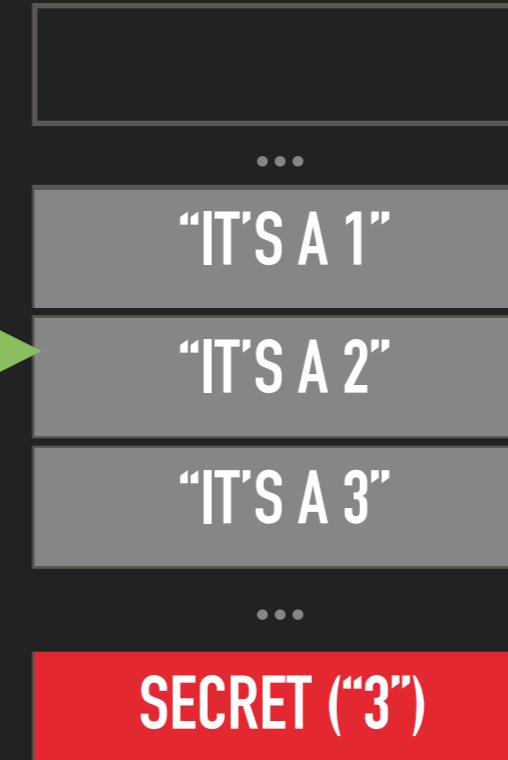
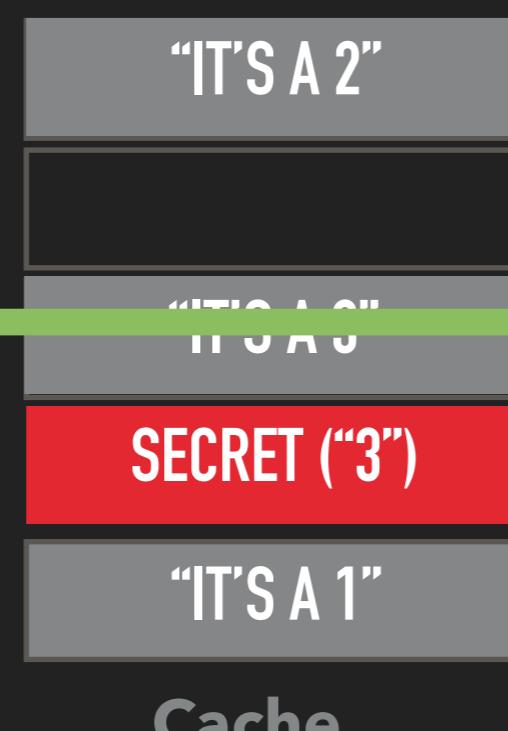
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
00101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



read: 103ns (uncached read)  
read: 103ns (uncached read)

- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses



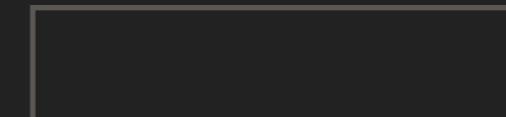
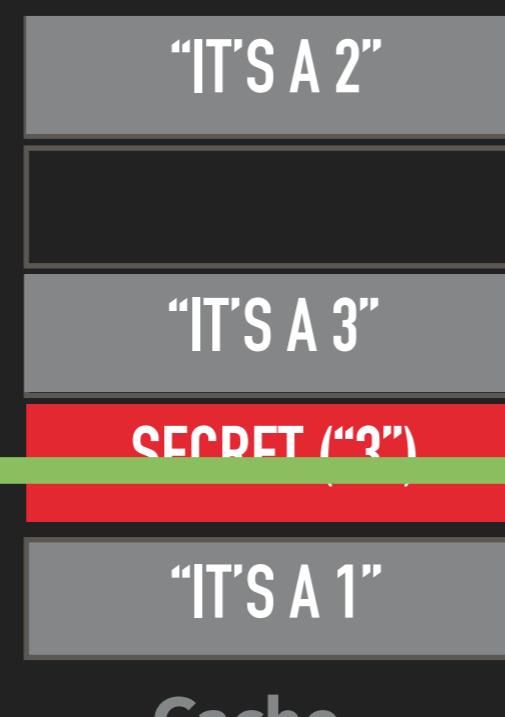
## MELTDOWN: THE ATTACK

110011010  
010111010  
111100100  
00101101  
100110010

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses



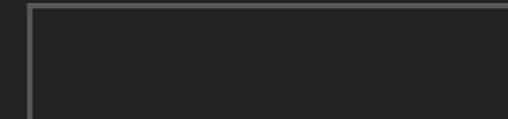
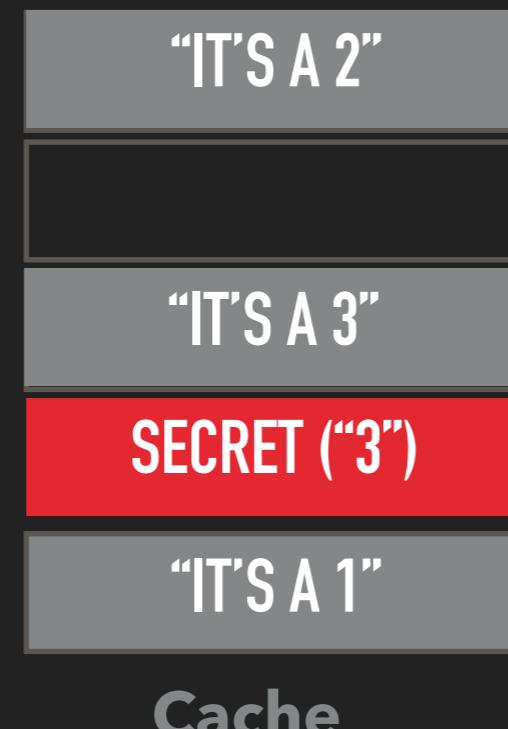
## MELTDOWN: THE ATTACK

~~110011010  
010111010  
111100100  
000101101  
100110010~~

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



...

"IT'S A 1"

"IT'S A 2"

"IT'S A 3"

...

SECRET ("3")

read: 103ns (uncached read)

read: 103ns (uncached read)

read: 3ns (cached)

RAM

- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time
1. Block "It's a 3" will be the block read the fastest

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses



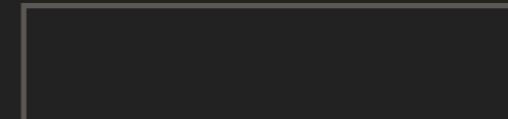
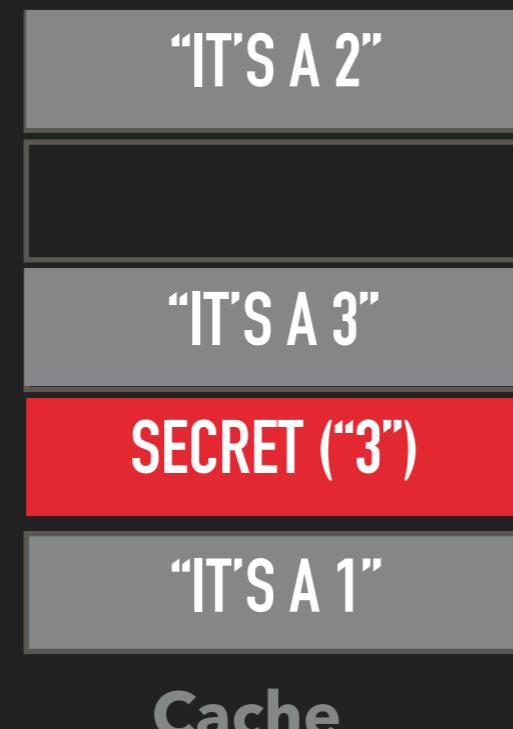
## MELTDOWN: THE ATTACK

~~110011010  
010111010  
111100100  
000101101  
100110010~~

**Spy**

110011010  
010111010  
111100100  
000101101  
100110010

**Collector**



...



read: 103ns (uncached read)

read: 103ns (uncached read)

read: 3ns (cached)

2 1. **Spy** will read the **secret**

2 Depending on the **value**, **Spy** will cache a grey block<sup>1</sup>

1 3. CPU detects **Spys** access validation and terminates **Spy**

4. **Collector** now reads all grey blocks and stops the time

1. Block "It's a 3" will be the block read the fastest

<sup>1</sup> Actually Spy will cache the address of block #3 and Collector will read the blocks addresses

## MELTDOWN

Meltdown exploits two properties of modern CPUs

- ▶ *Out of order execution* of OPs and μOPs
- ▶ Timing side channels for the cache

This allows an attacker to

- ▶ Read all memory mapped<sup>1</sup> in a process
- ▶ This often includes all other processes memory
- ▶ This does NOT allow reading “outside of a VM<sup>2</sup>”

<sup>1</sup> Virtual vs. physical memory is a subject for another time    <sup>2</sup> For fully virtualised VMs





SPECULATIVE  
EXECUTION

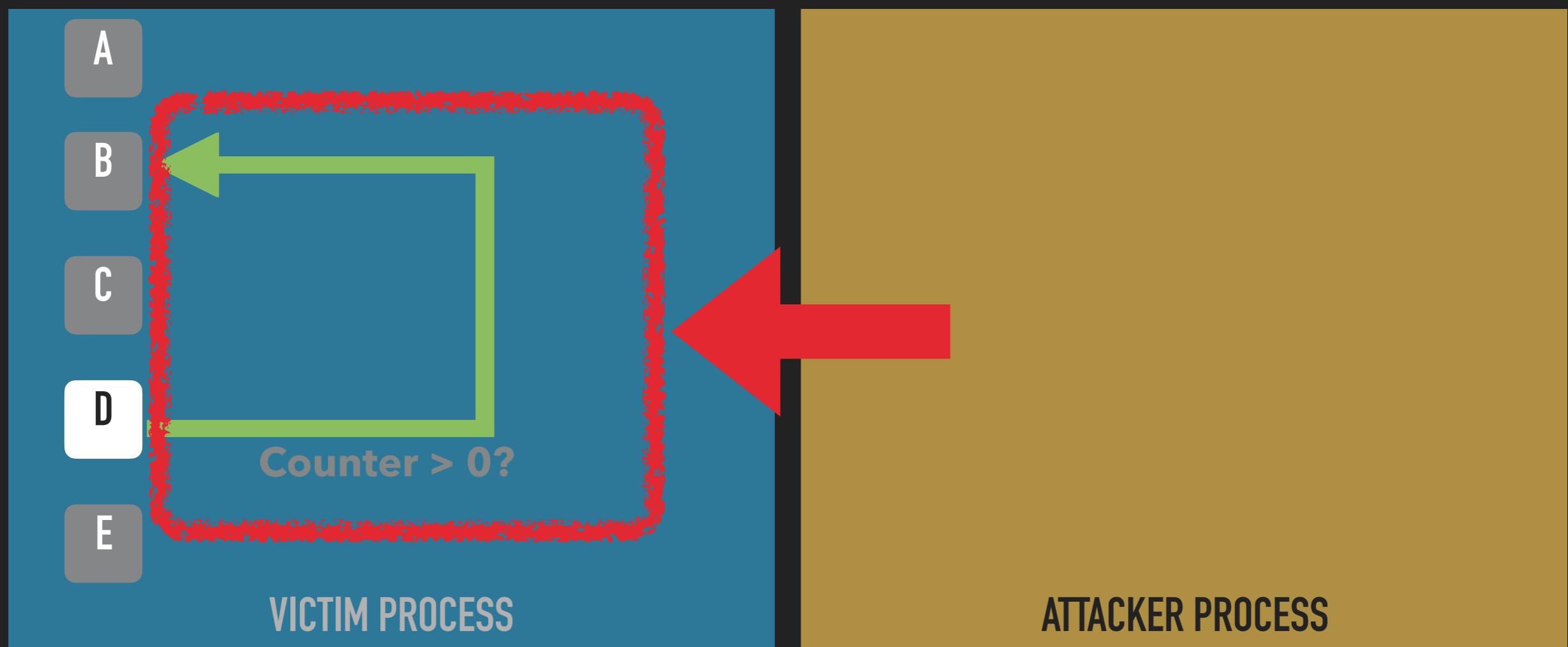
---

**SPECTRE**

# SPECTRE



Spectre attacks other processes by forcing them to *speculatively* run other code paths



## SPECTRE



Spectre works like this:

- force victim to leak secret
- stash away secret
- retrieve secret

## SPECTRE



Spectre works like this:

- force victim to leak secret
- stash away secret
- retrieve secret
  
- and ■ *basically work like in Meltdown*

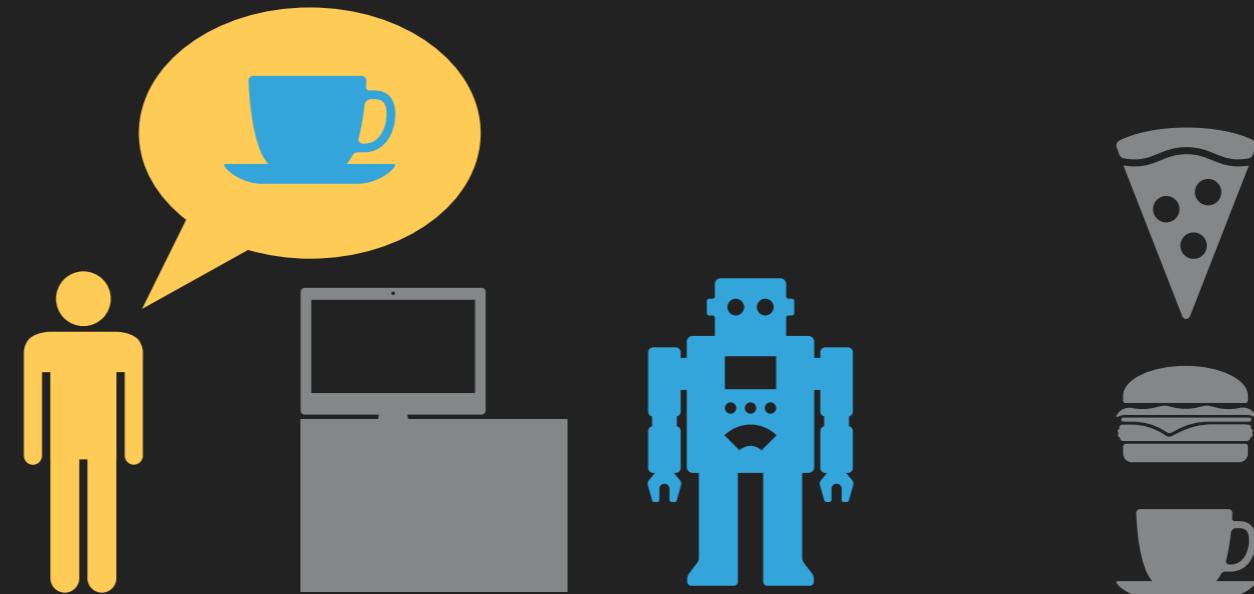
## SPECTRE



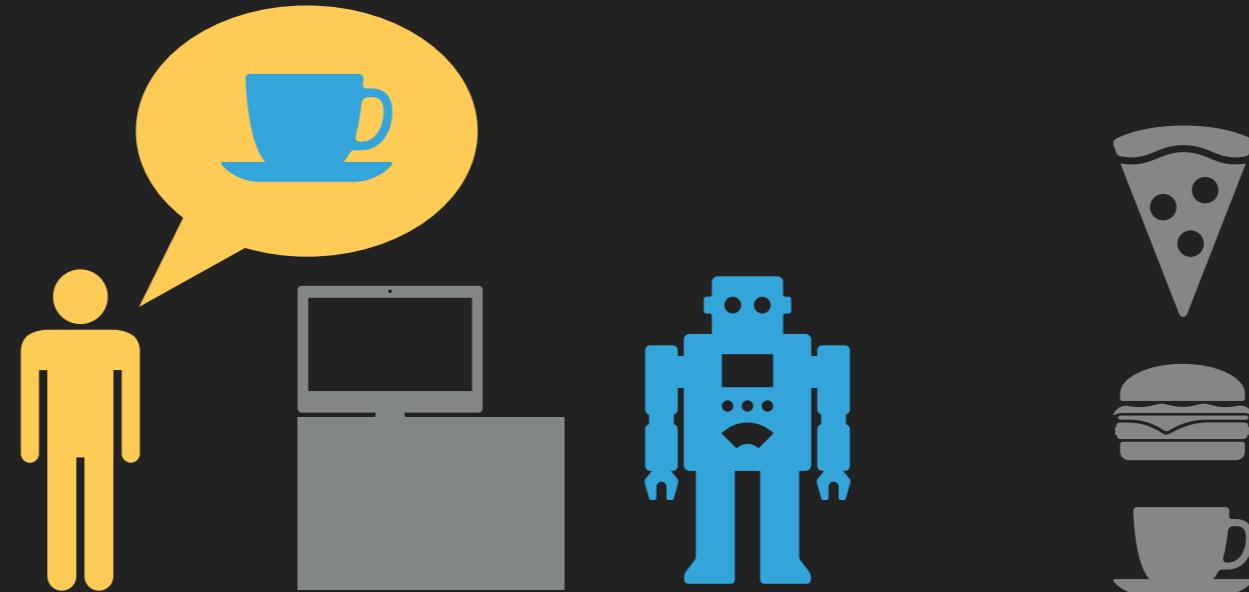
Spectre works like this:

- force victim to leak secret
- stash away secret
- retrieve secret
- and ■ *basically work like in Meltdown*
- works by manipulating the *branch prediction* of the CPU

## SPECTRE: BRANCH PREDICTION



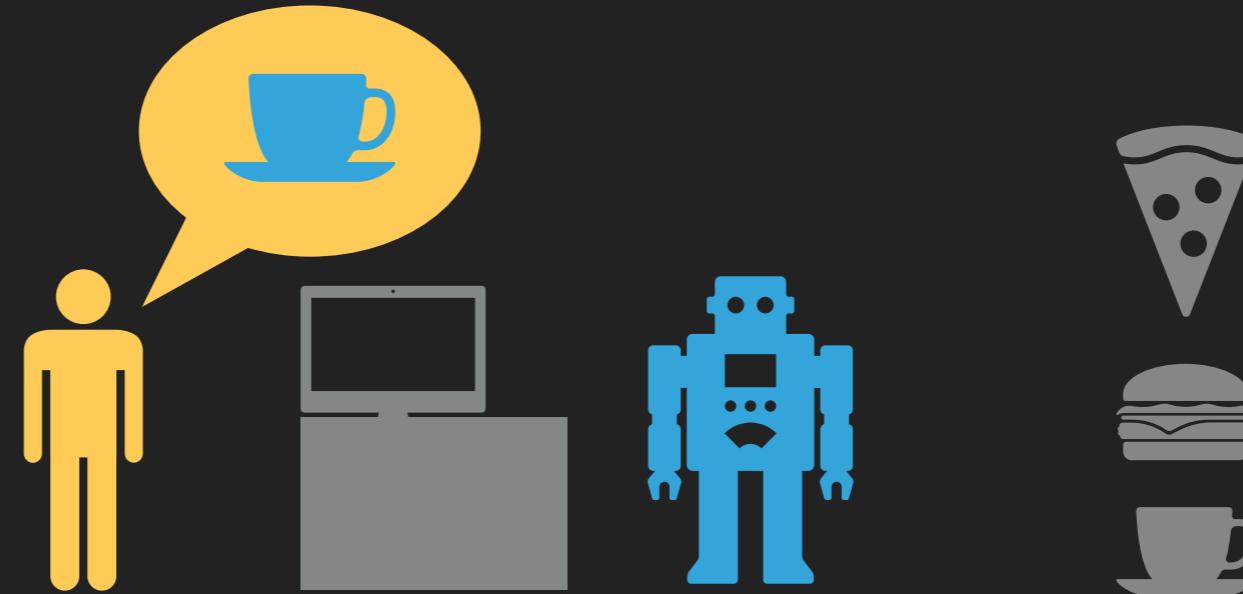
## SPECTRE: BRANCH PREDICTION



*Monday*

A yellow speech bubble containing a blue coffee cup icon, followed by the word "Monday" in a stylized script font.

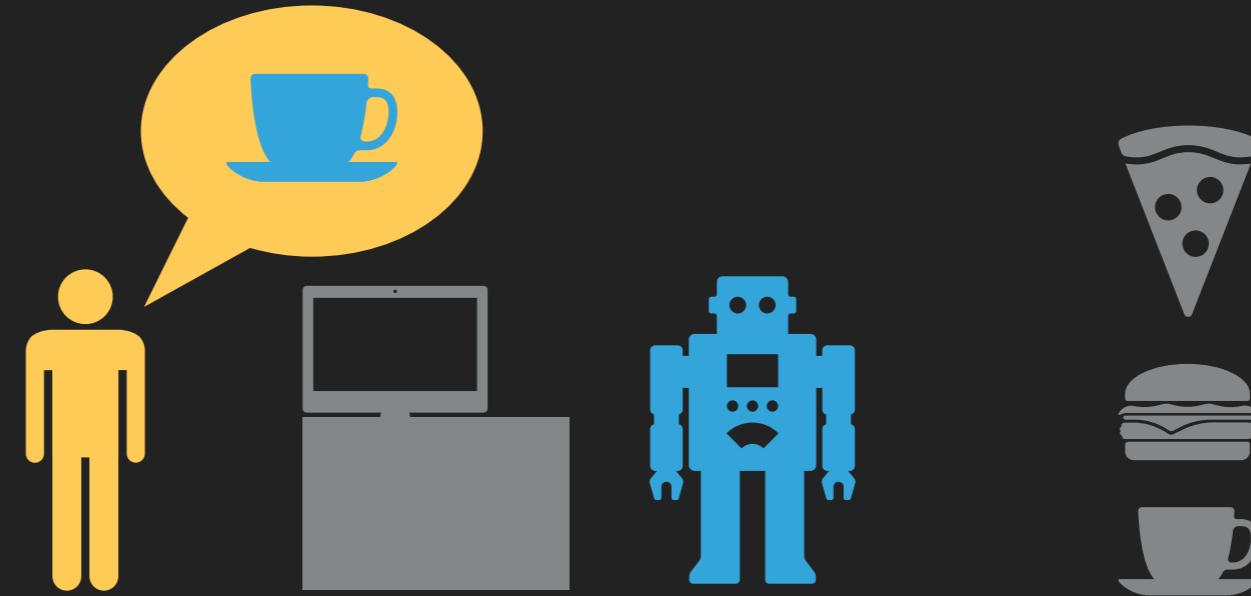
## SPECTRE: BRANCH PREDICTION



 *Monday*

 *Tuesday*

## SPECTRE: BRANCH PREDICTION



*Monday*

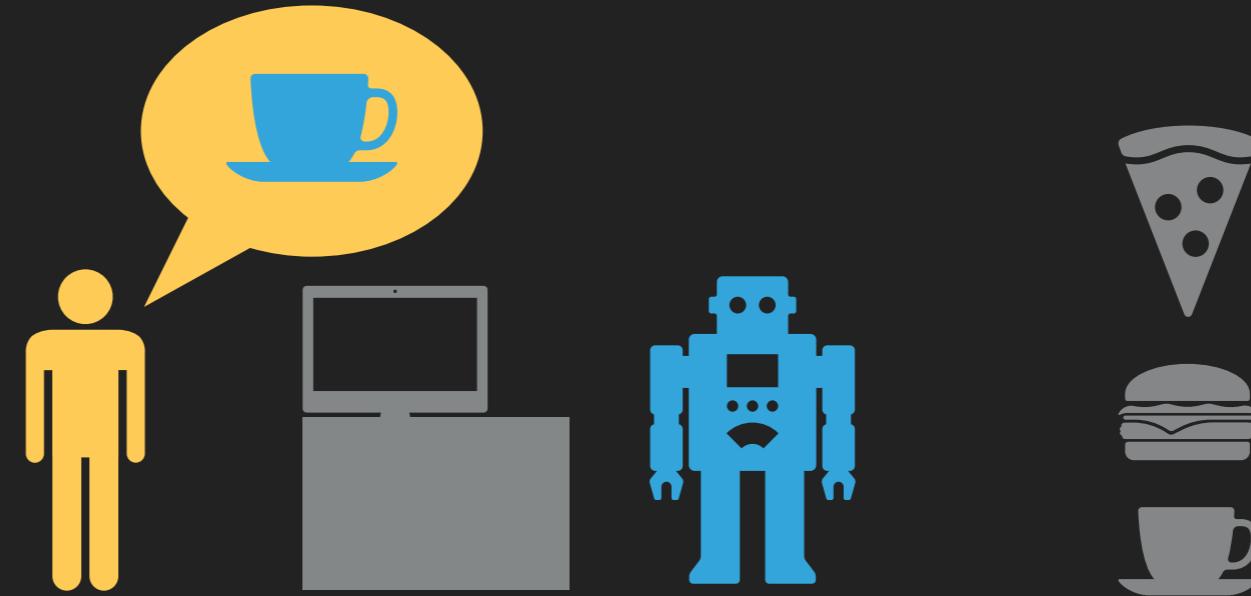


*Wednesday*



*Tuesday*

## SPECTRE: BRANCH PREDICTION



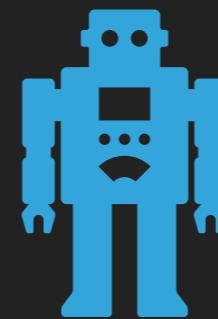
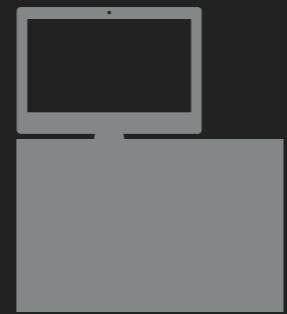
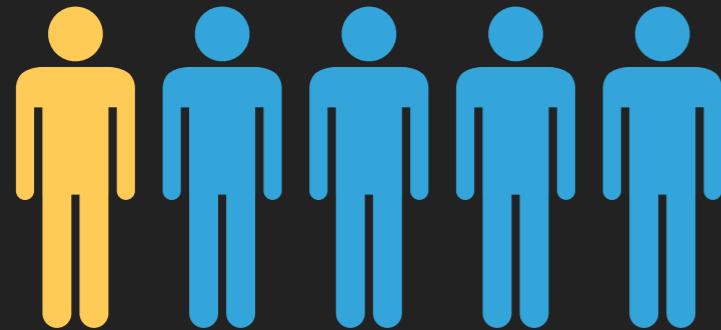
 *Monday*

 *Wednesday*

 *Tuesday*

...

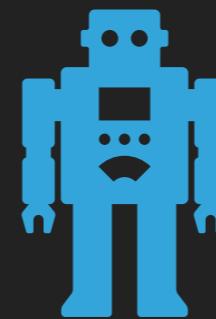
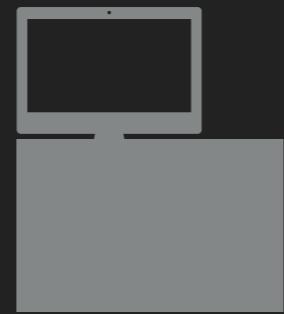
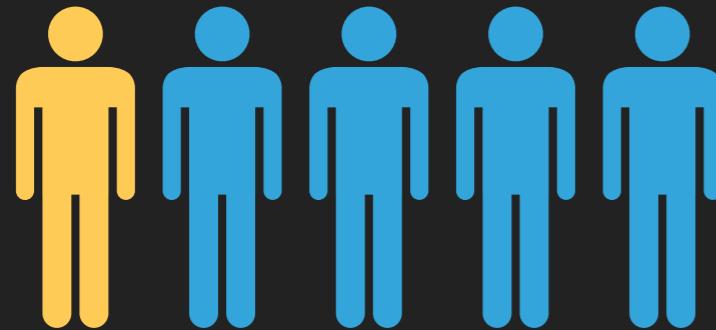
## SPECTRE: SPECULATIVE EXECUTION



The CPU can improve the coffee machine utilisation by  
*speculatively* brewing the coffee for 

This is very similar to the effect seen in Meltdown.

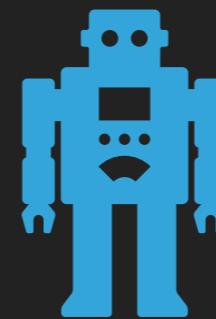
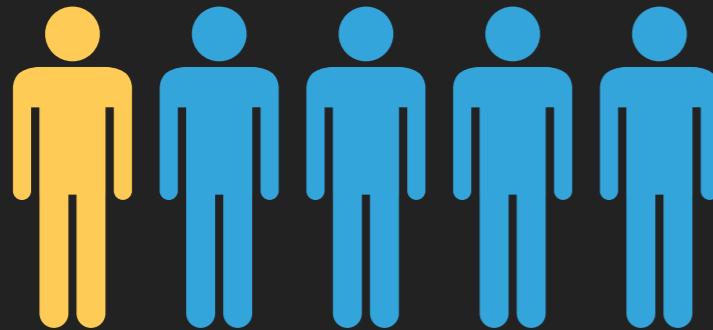
## SPECTRE: SPECULATIVE EXECUTION



The CPU can improve the coffee machine utilisation by  
*speculatively* brewing the coffee for 

This is very similar to the effect seen in Meltdown.

## SPECTRE: SPECULATIVE EXECUTION

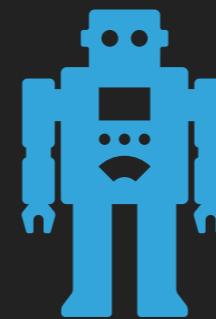
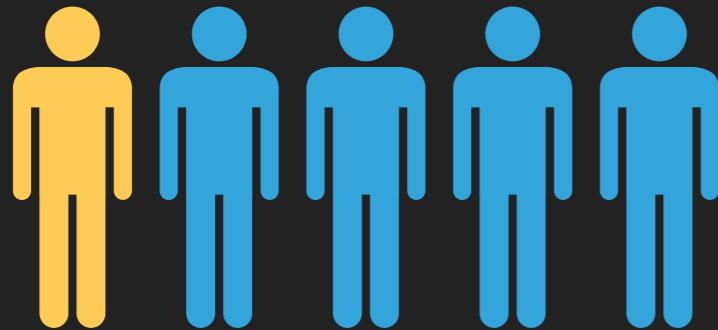


The CPU can improve the coffee machine utilisation by *speculatively* brewing the coffee for 

This is very similar to the effect seen in Meltdown.

- ▶ In the **Meltdown** attack the CPU knows the next instruction (order) and asynchronously checks the permissions

## SPECTRE: SPECULATIVE EXECUTION

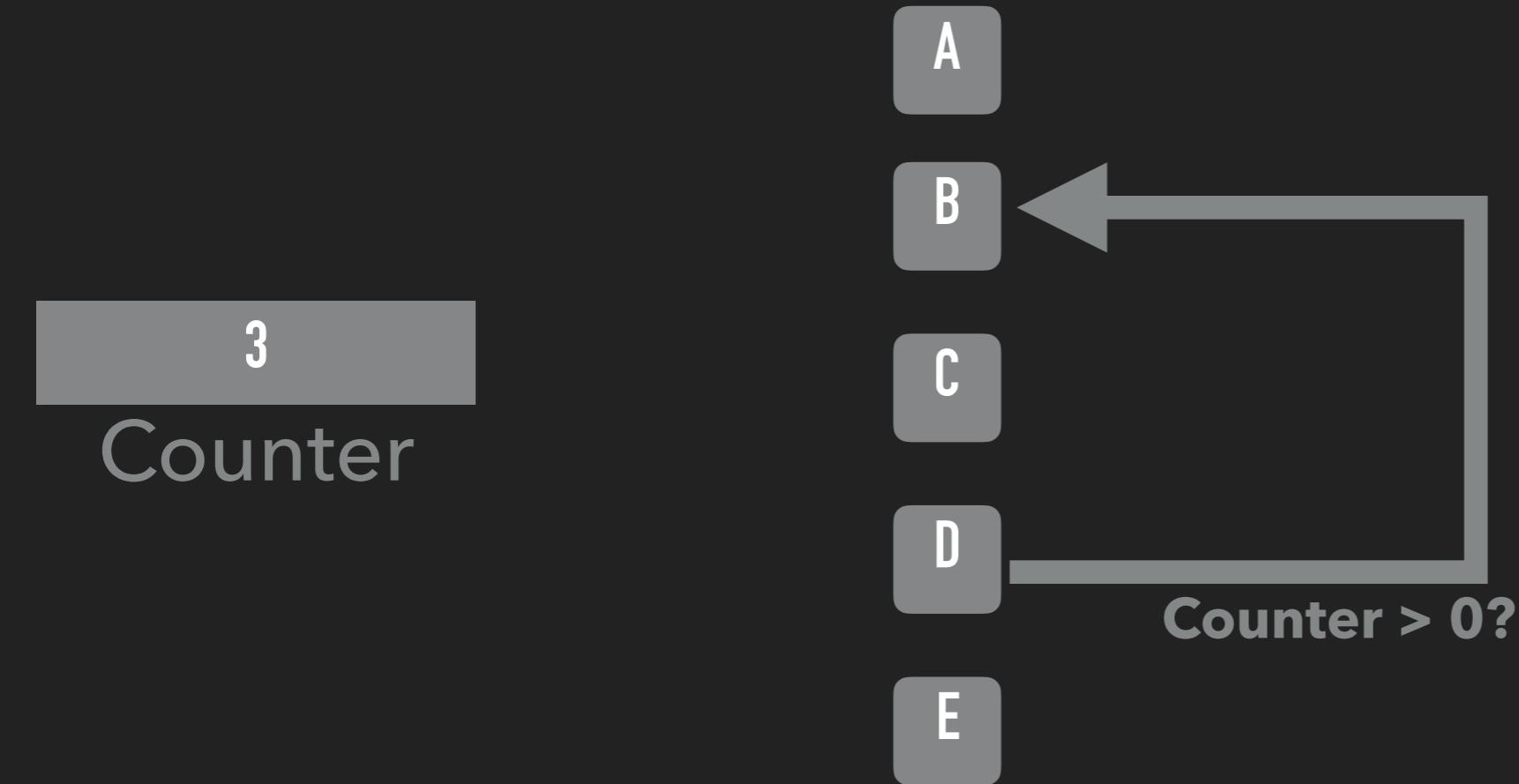


The CPU can improve the coffee machine utilisation by *speculatively* brewing the coffee for 

This is very similar to the effect seen in Meltdown.

- ▶ In the **Meltdown** attack the CPU knows the next instruction (order) and asynchronously checks the permissions
- ▶ In **Spectre** the CPU guesses the next instructions based on heuristics (brew coffee without knowing the order)

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

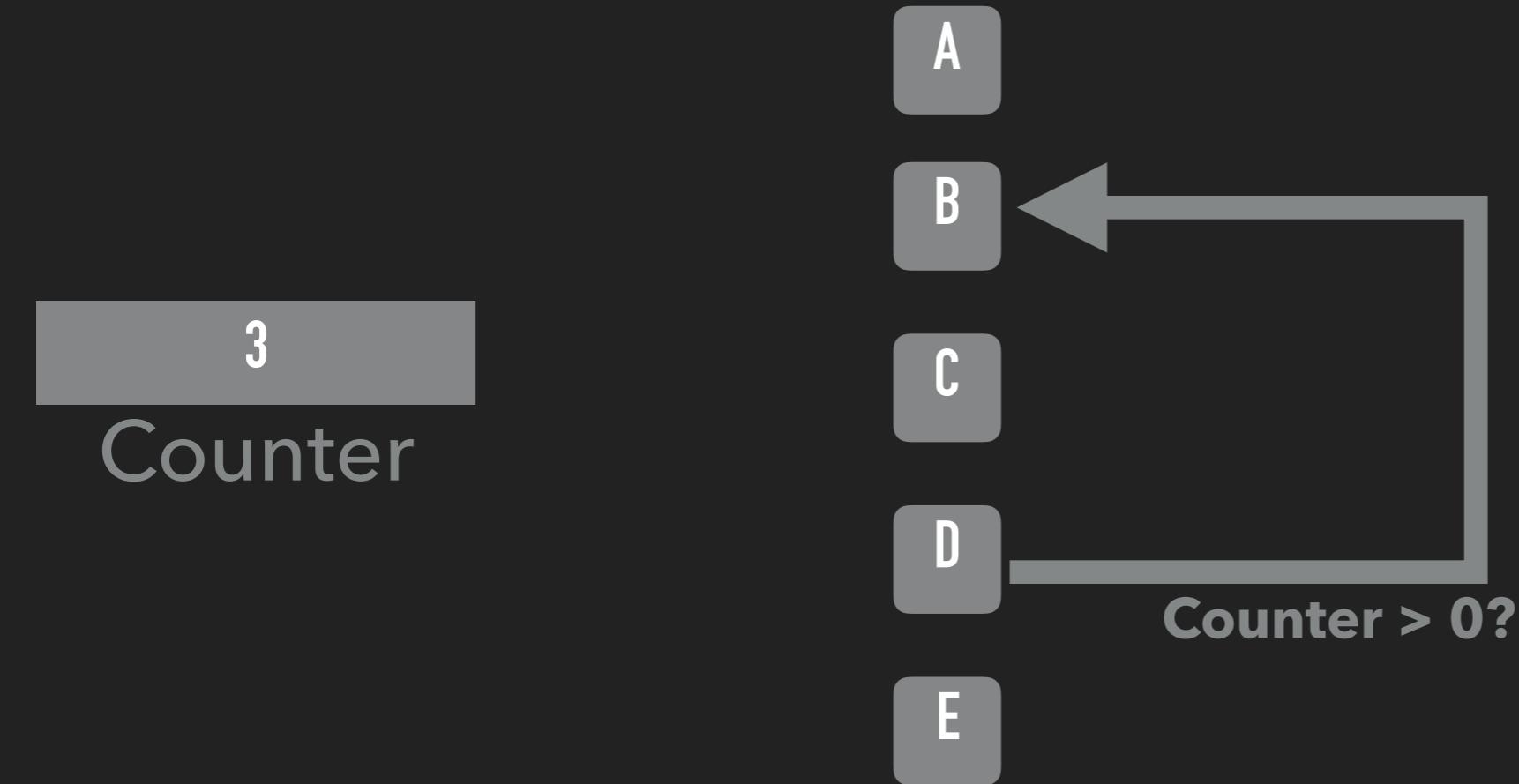
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

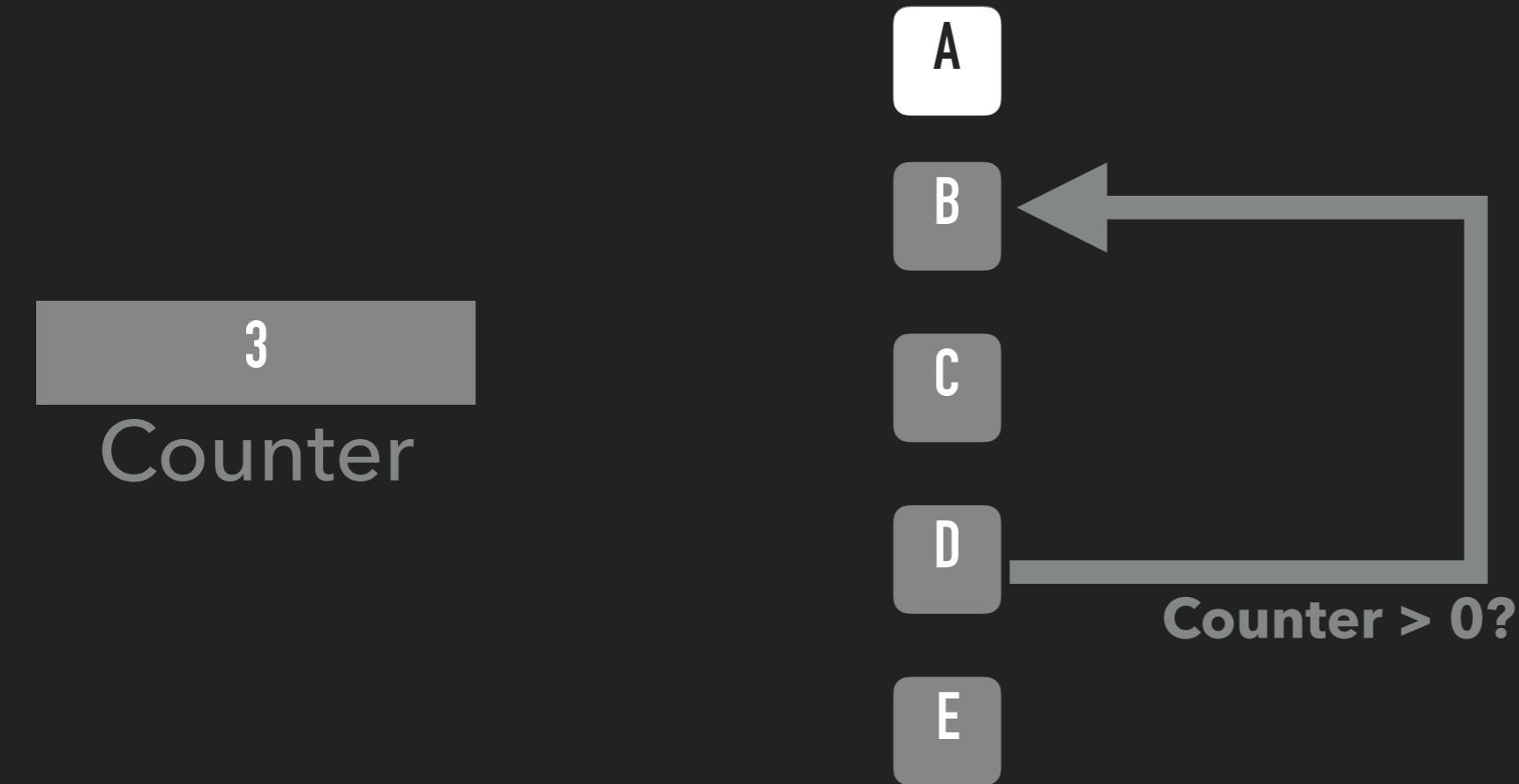
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A    B    ...    E

A

←

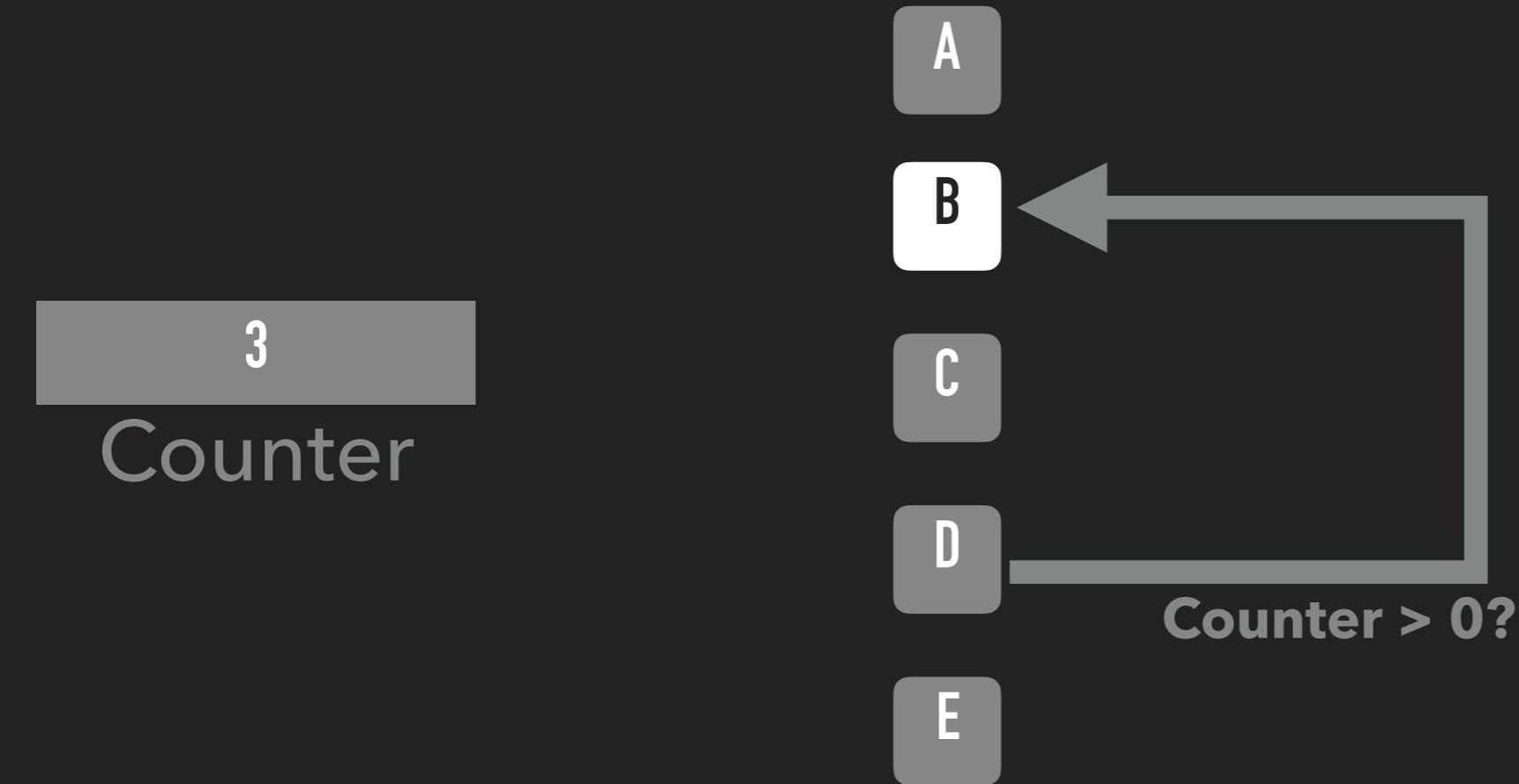
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A    B    ...    E

A

←

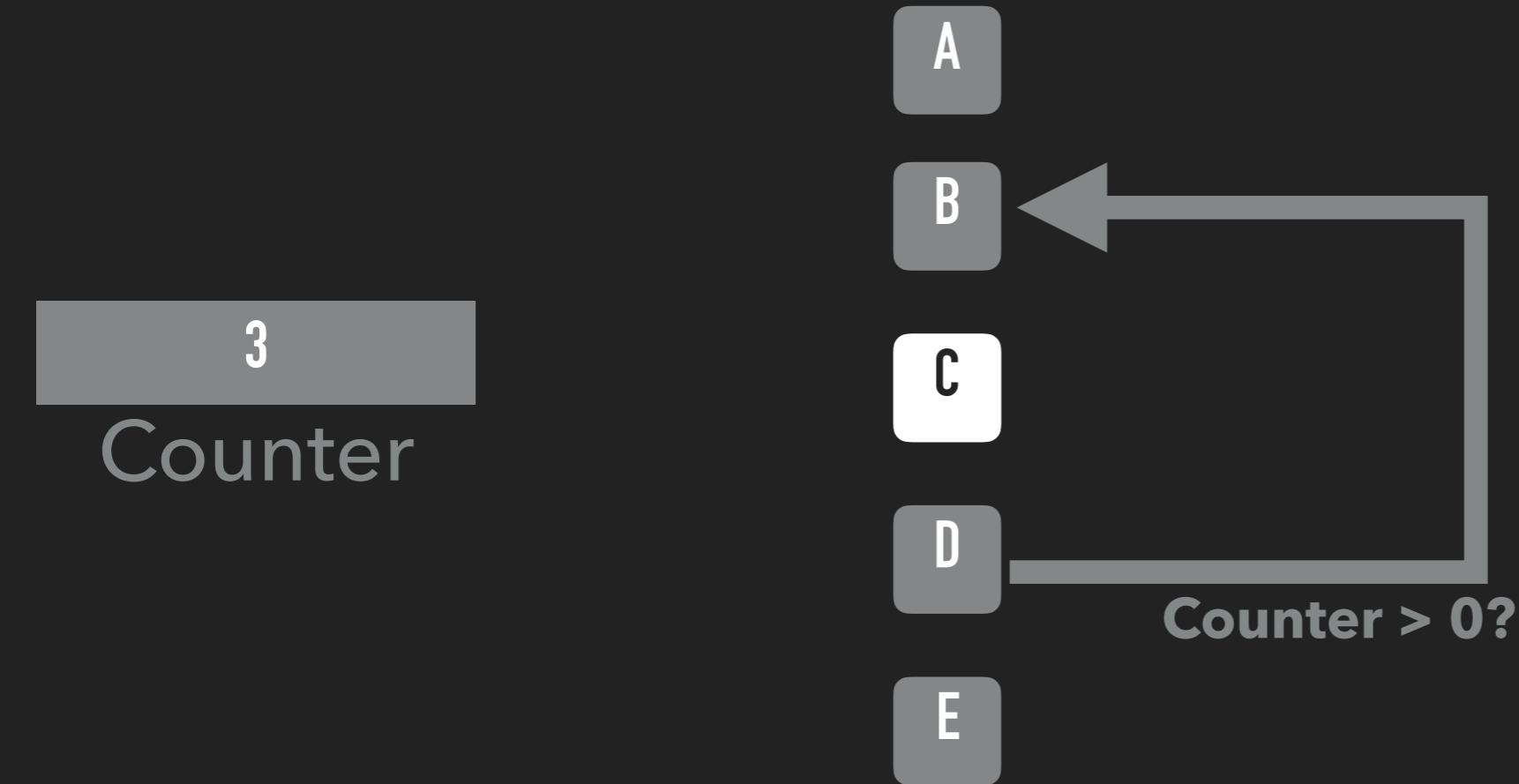
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A    B    ...    E

A

←

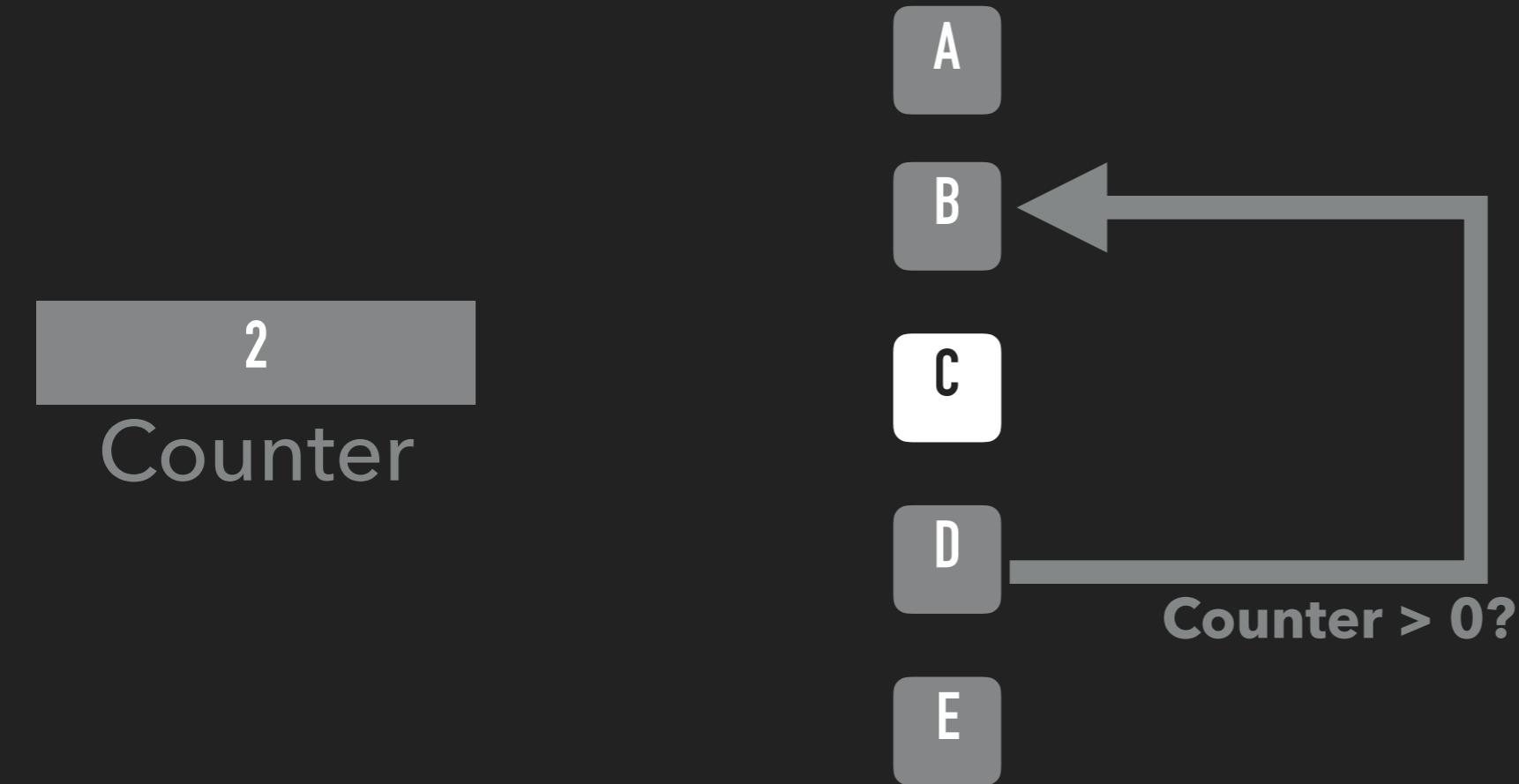
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

Some variable used in branch

A    B    ...    E

Instructions

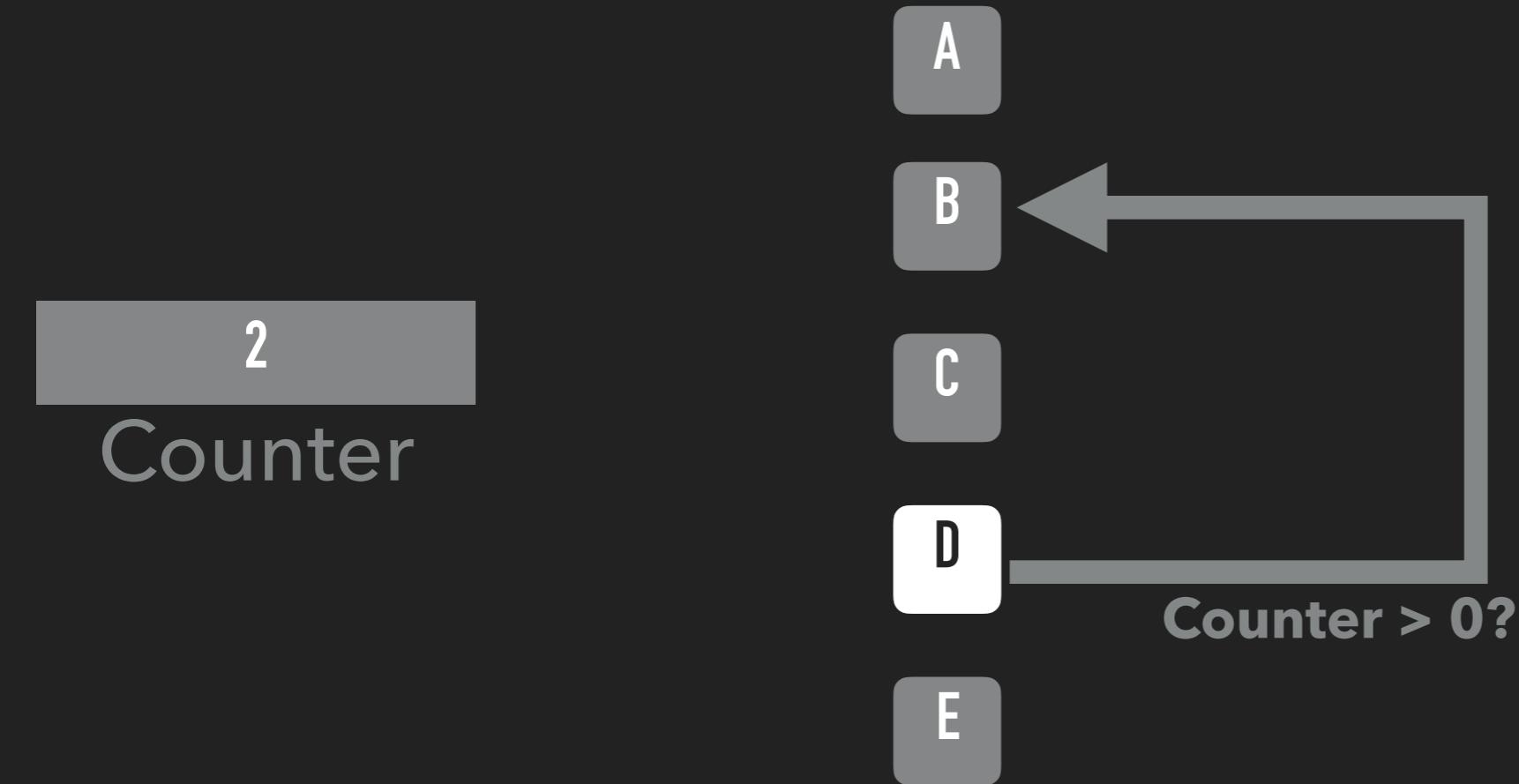
A

Currently active instruction



Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

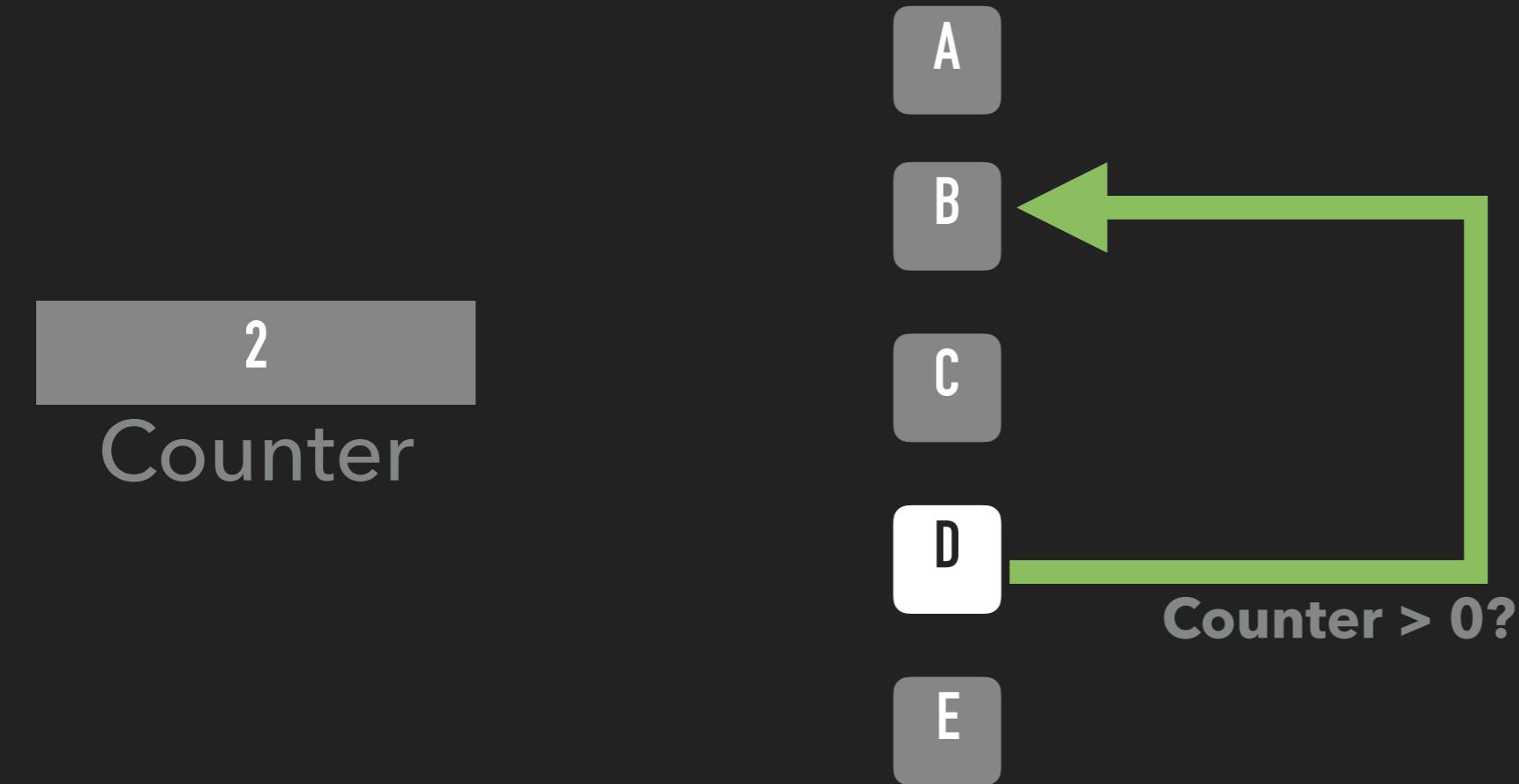
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



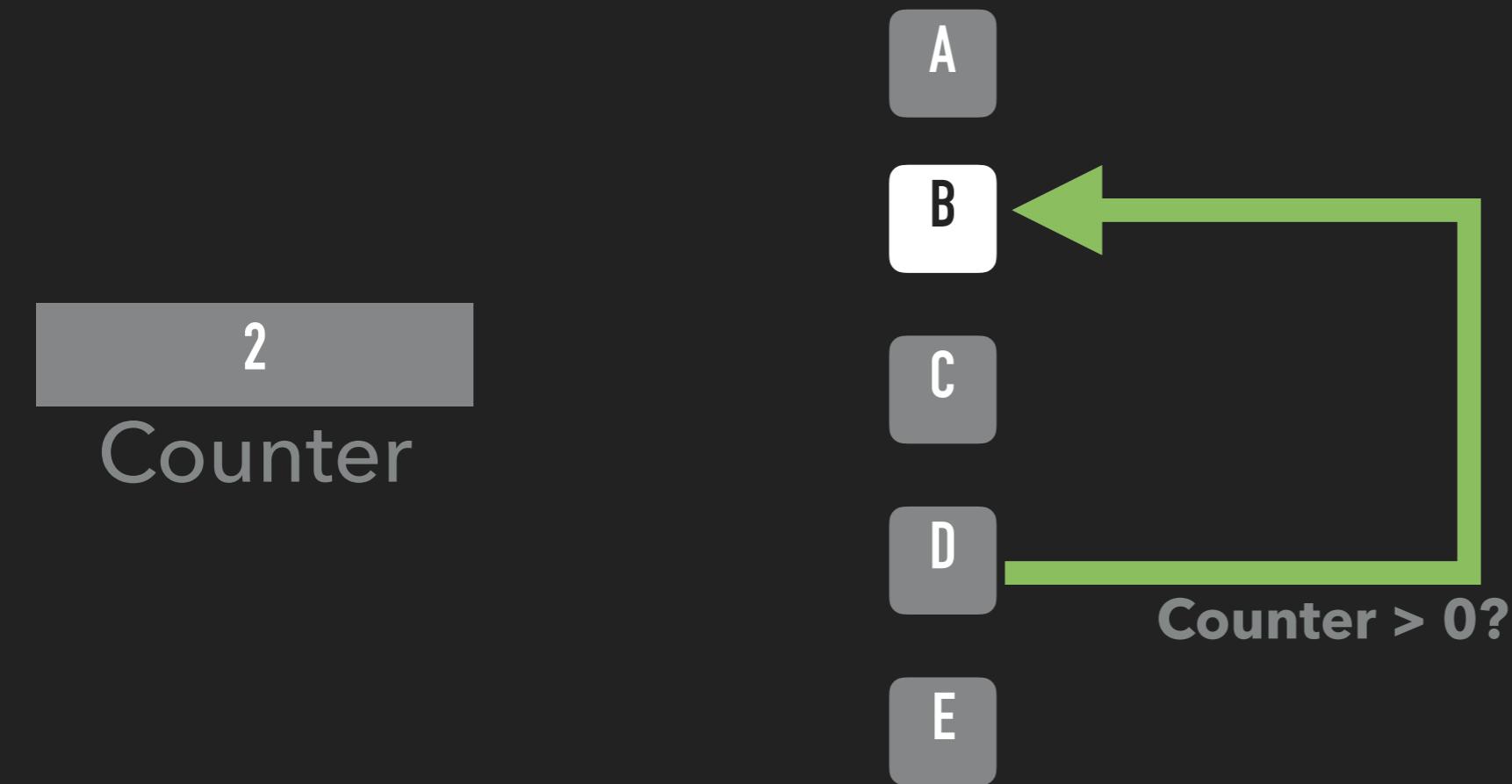
COUNTER

A B ... E

A

←

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A   B   ...   E

A

←

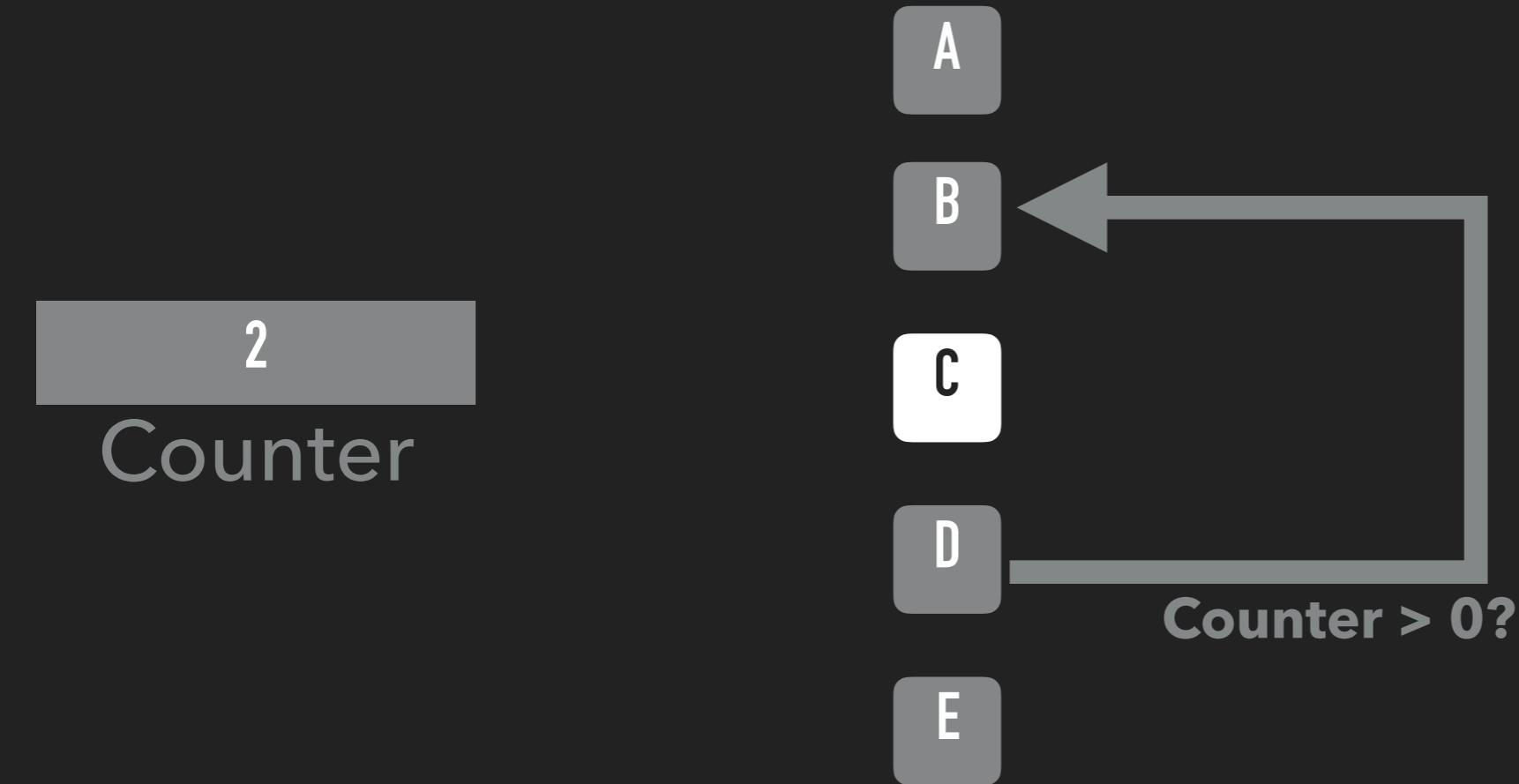
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

Some variable used in branch

A    B    ...    E

Instructions

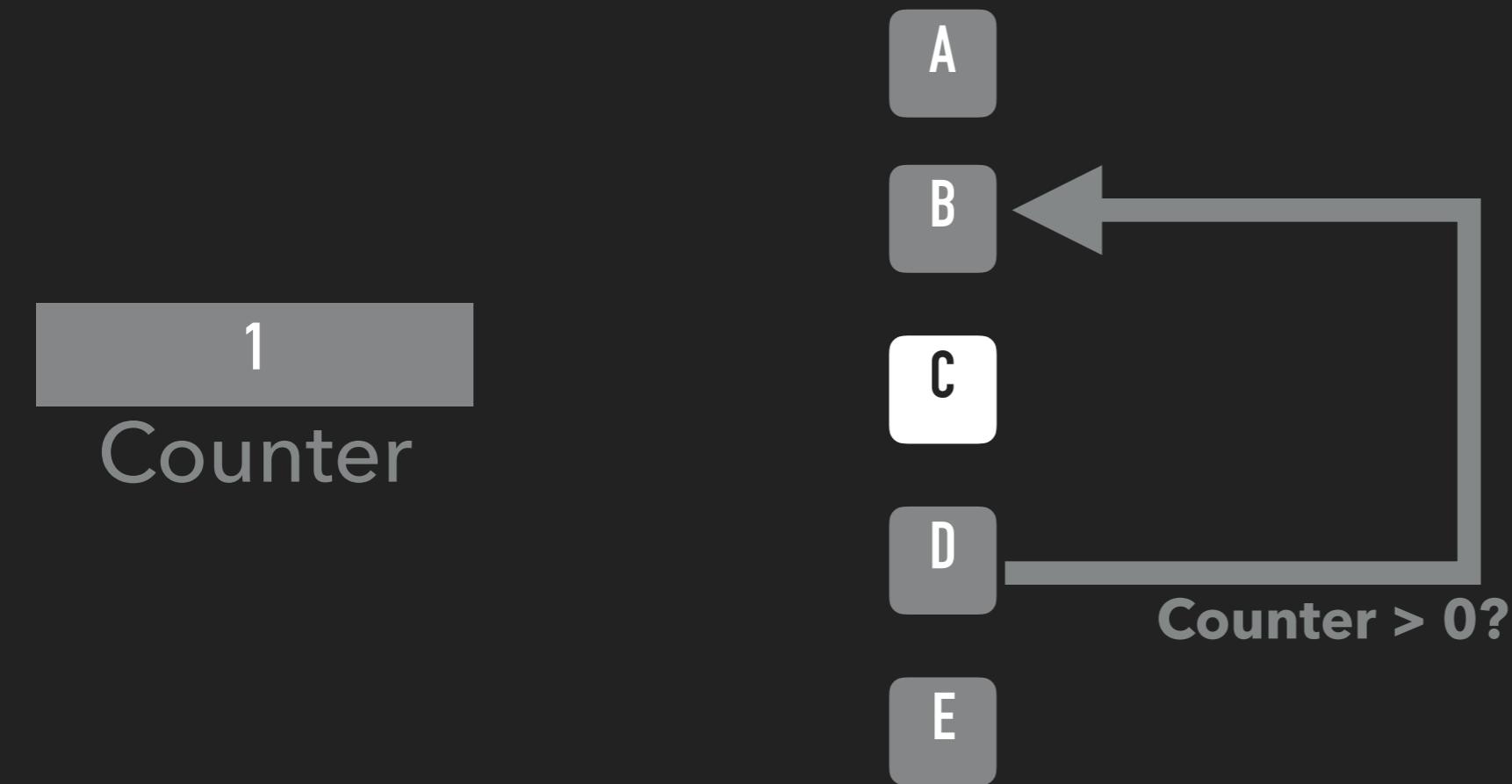
A

Currently active instruction



Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

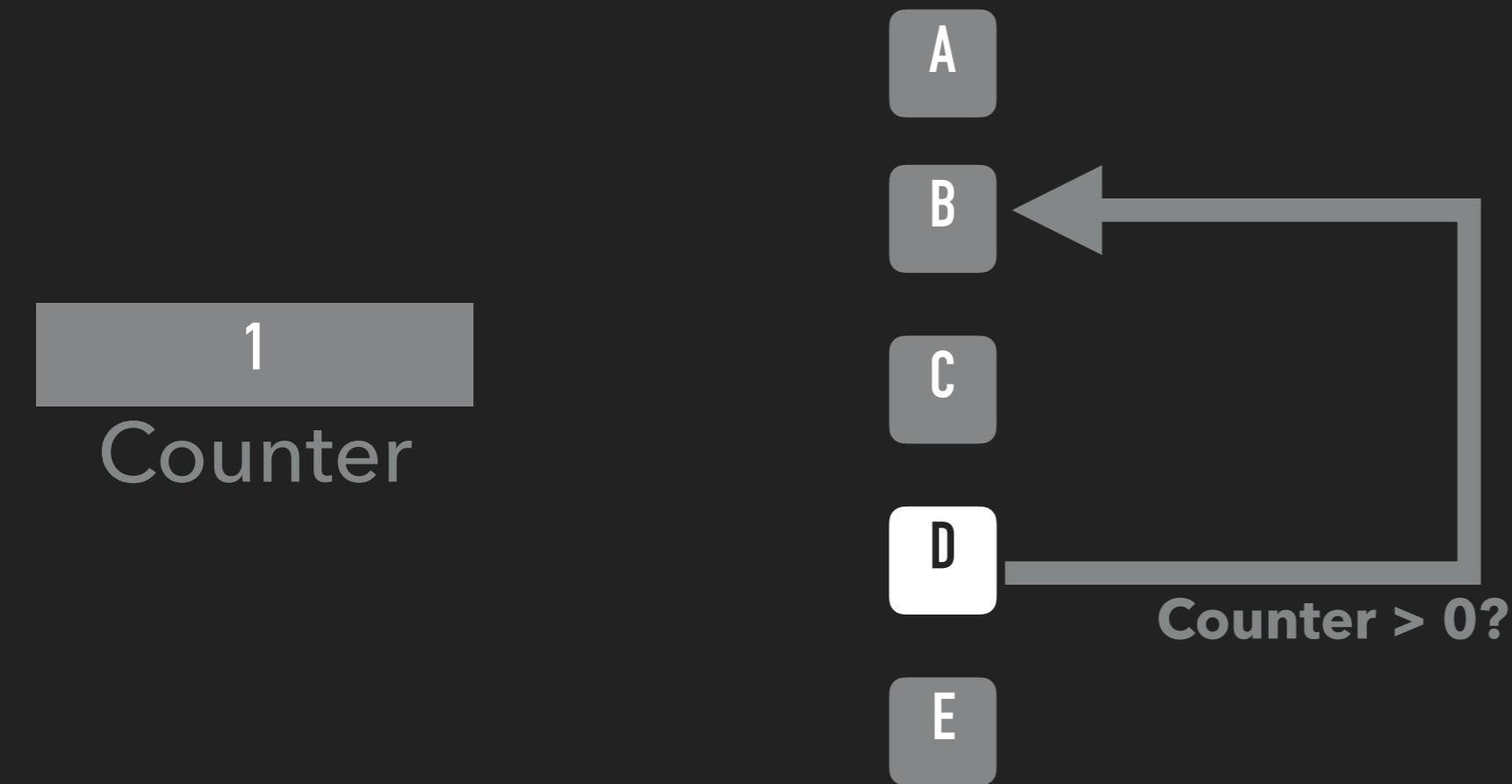
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A   B   ...   E

A

←

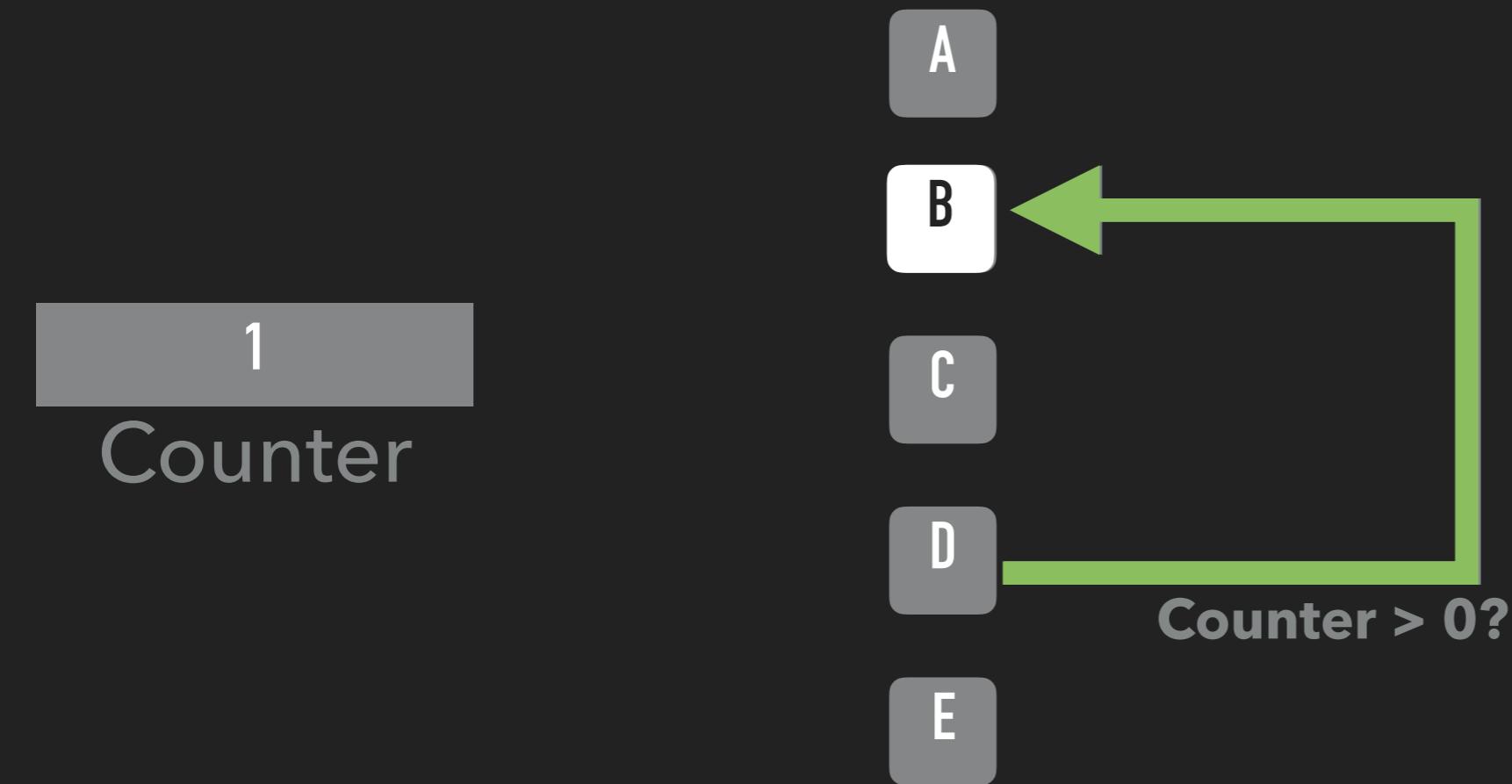
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



Some variable used in branch

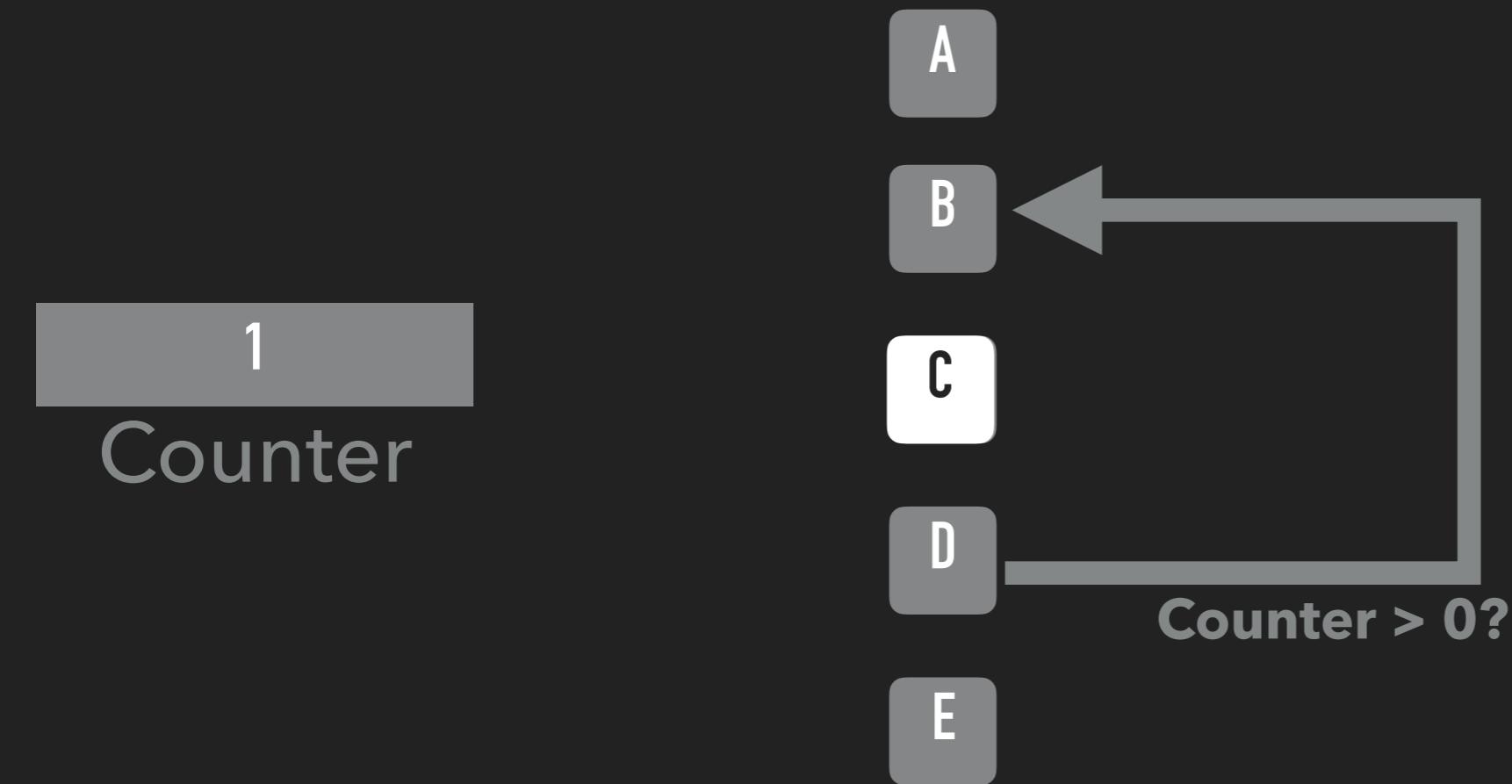
Instructions

A

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A   B   ...   E

A

←

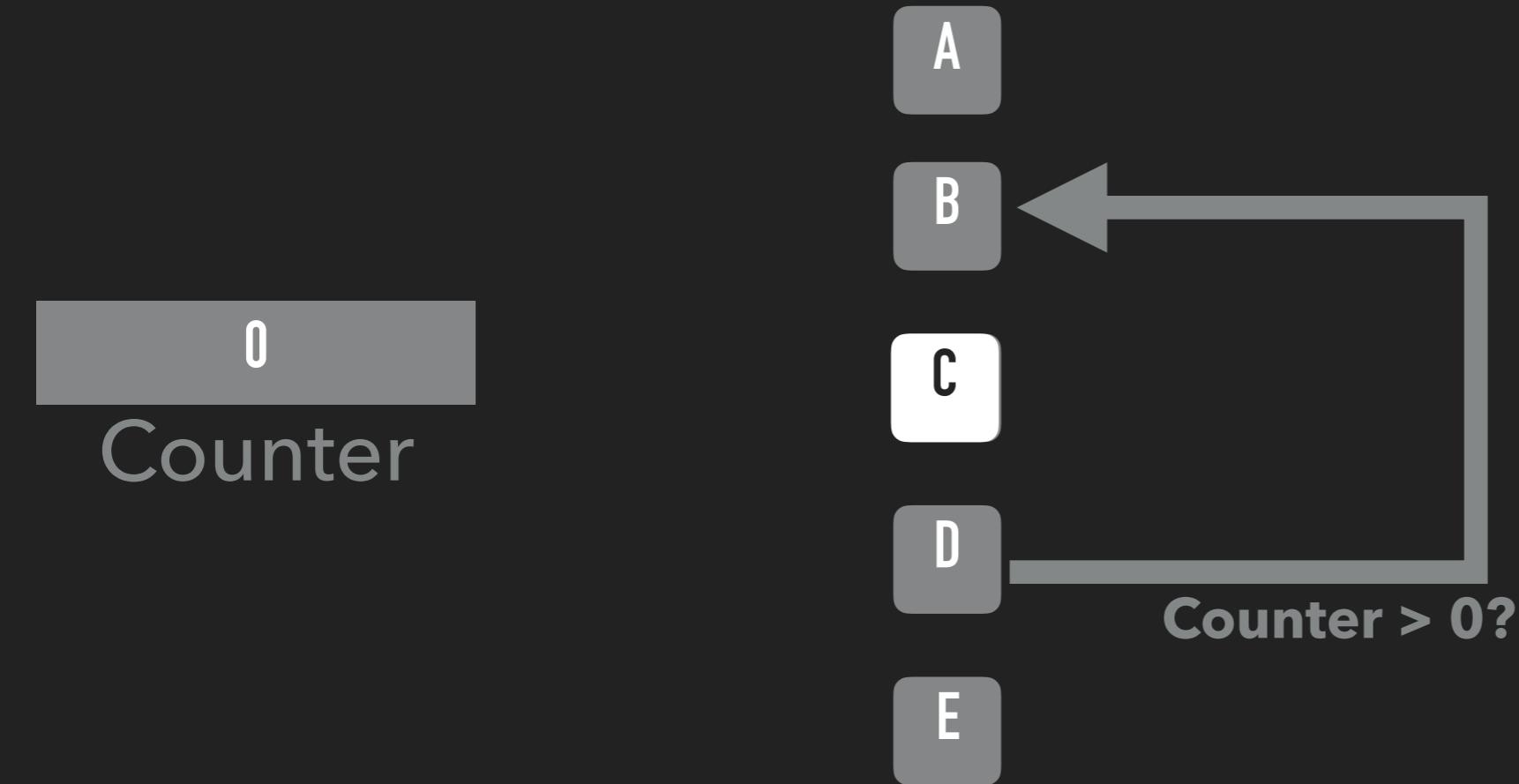
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

Some variable used in branch

A   B   ...   E

Instructions

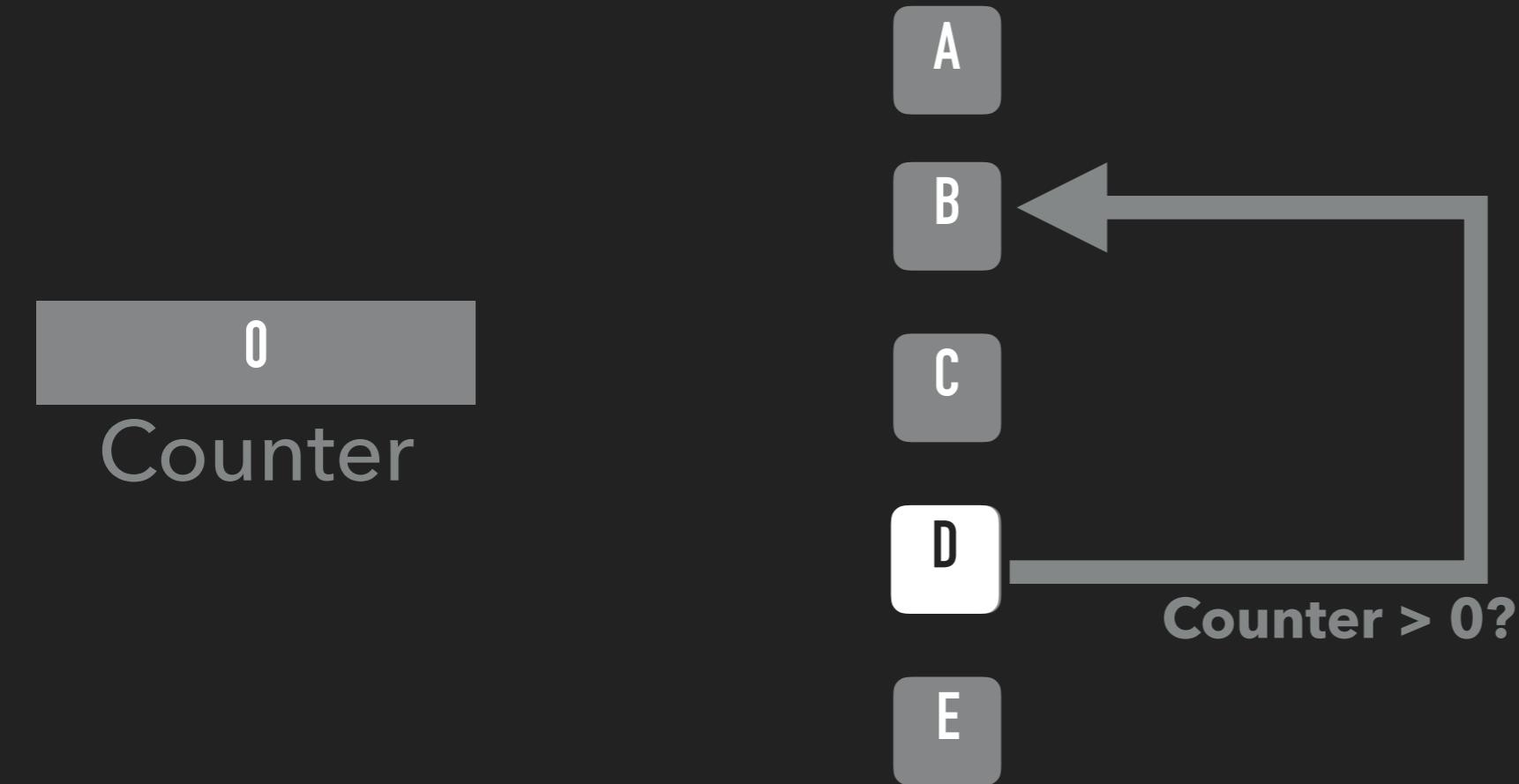
A

Currently active instruction



Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A   B   ...   E

A

←

Some variable used in branch

Instructions

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



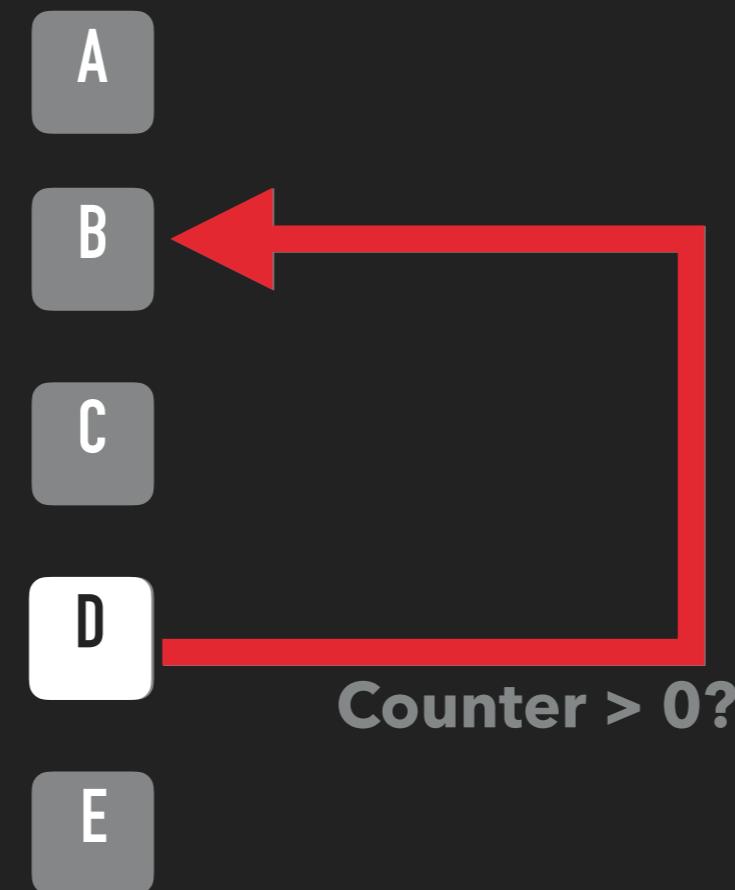
0  
Counter

COUNTER

A B ... E

A

←



Some variable used in branch

Instructions

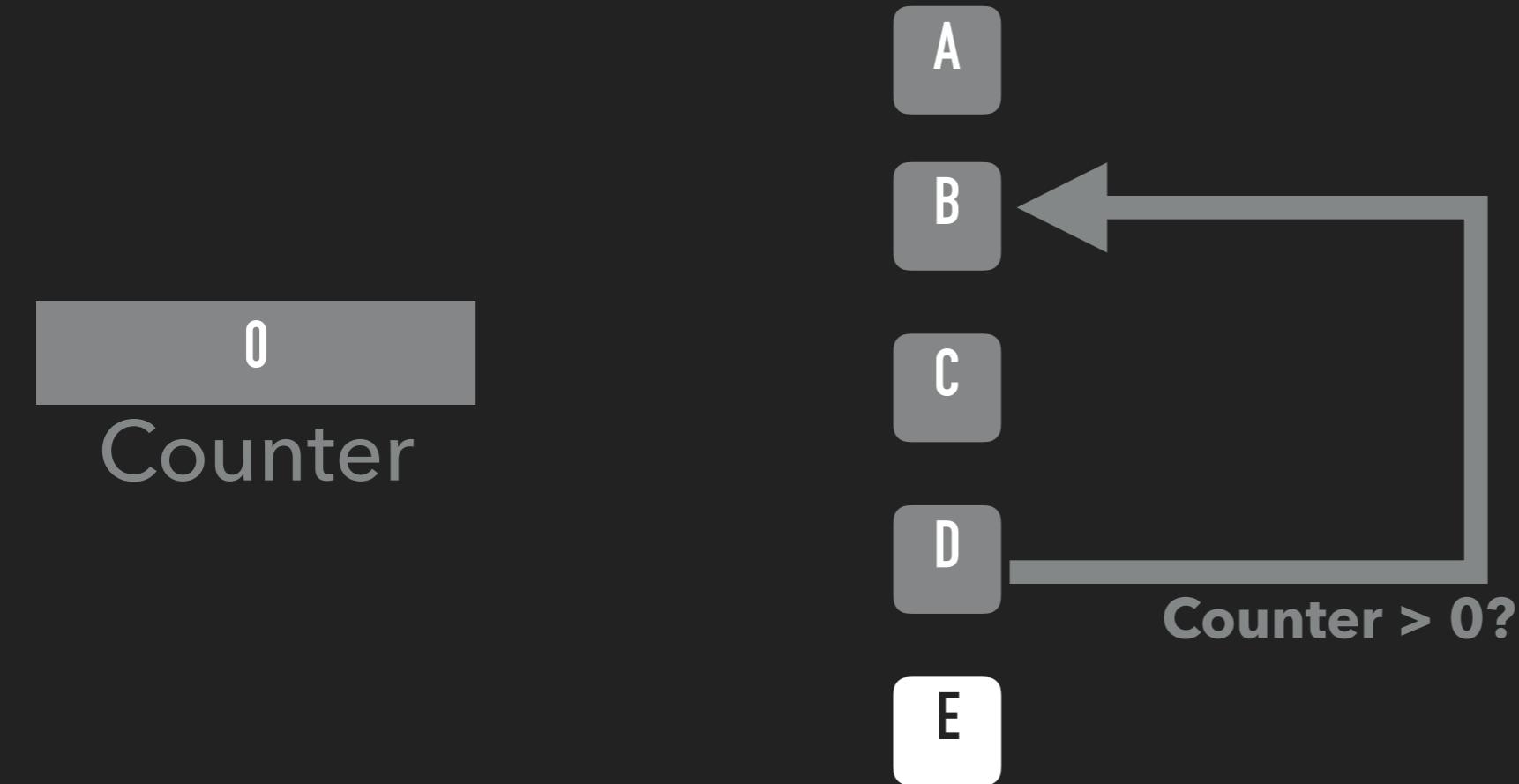
A B ... E

A

Currently active instruction

Conditional branch

# SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

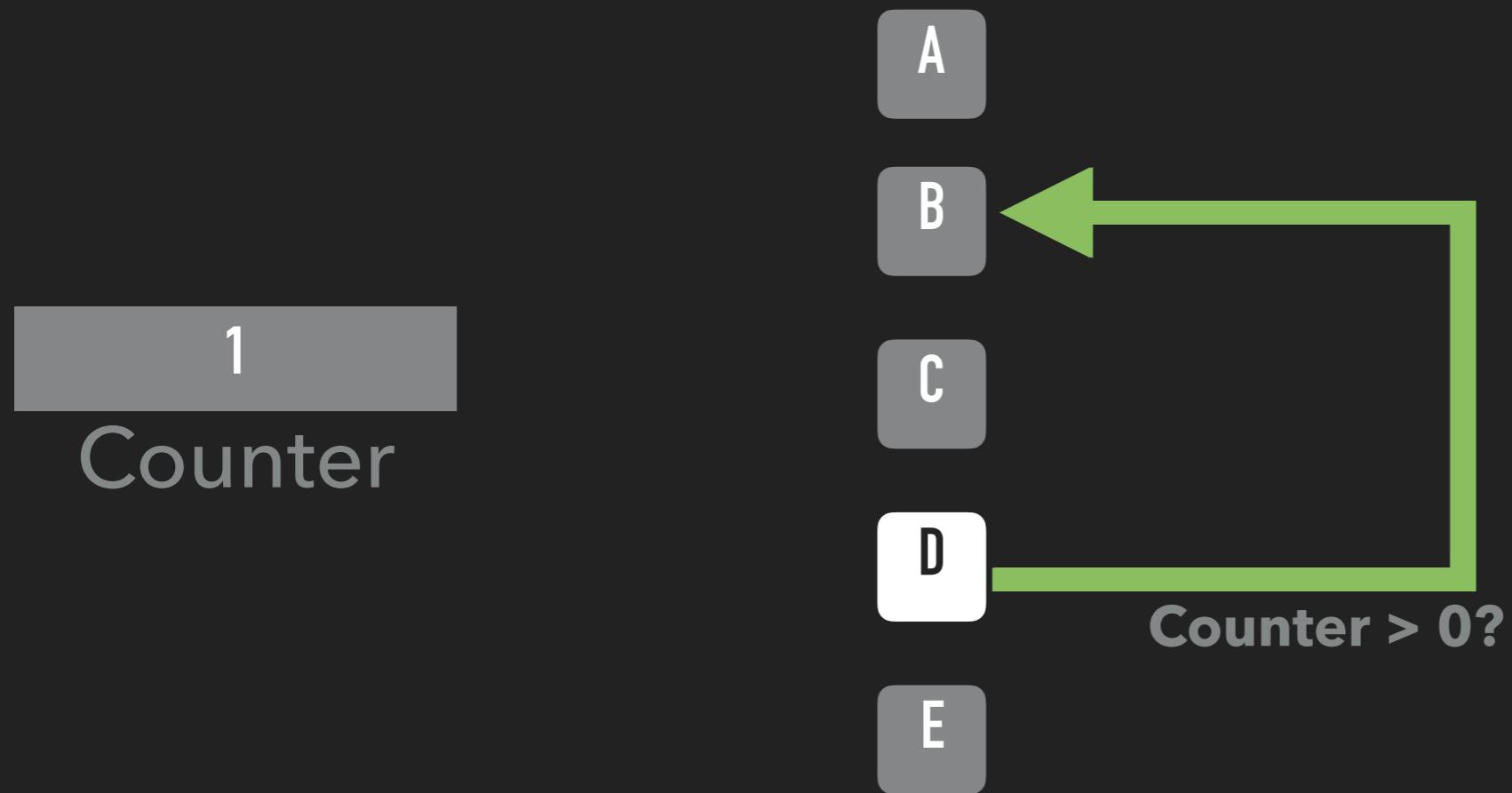
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

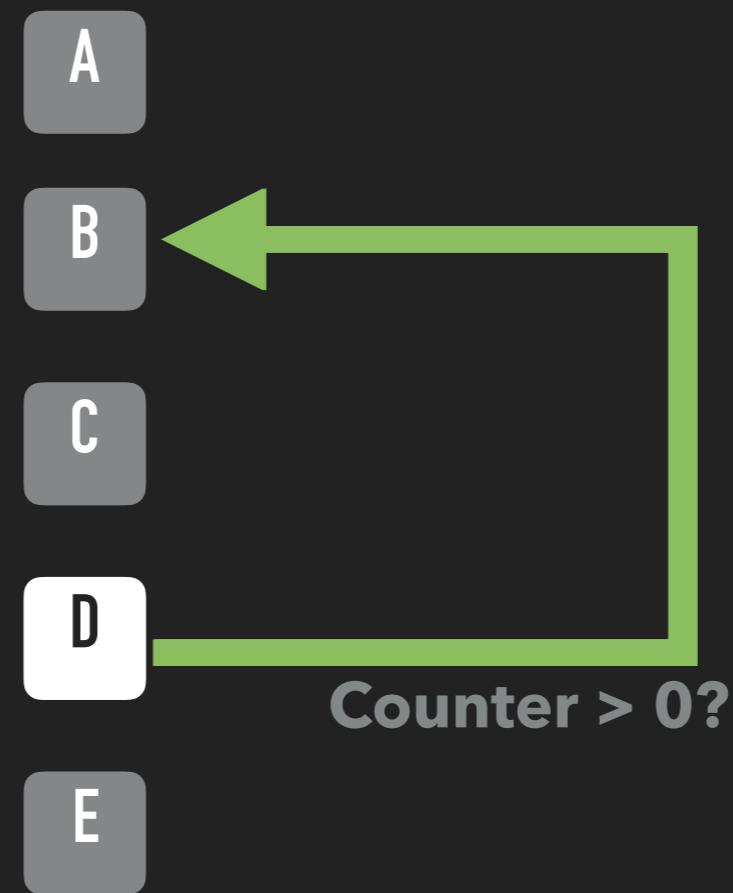
# SPECTRE: SPECULATIVE EXECUTION



## SPECTRE: SPECULATIVE EXECUTION

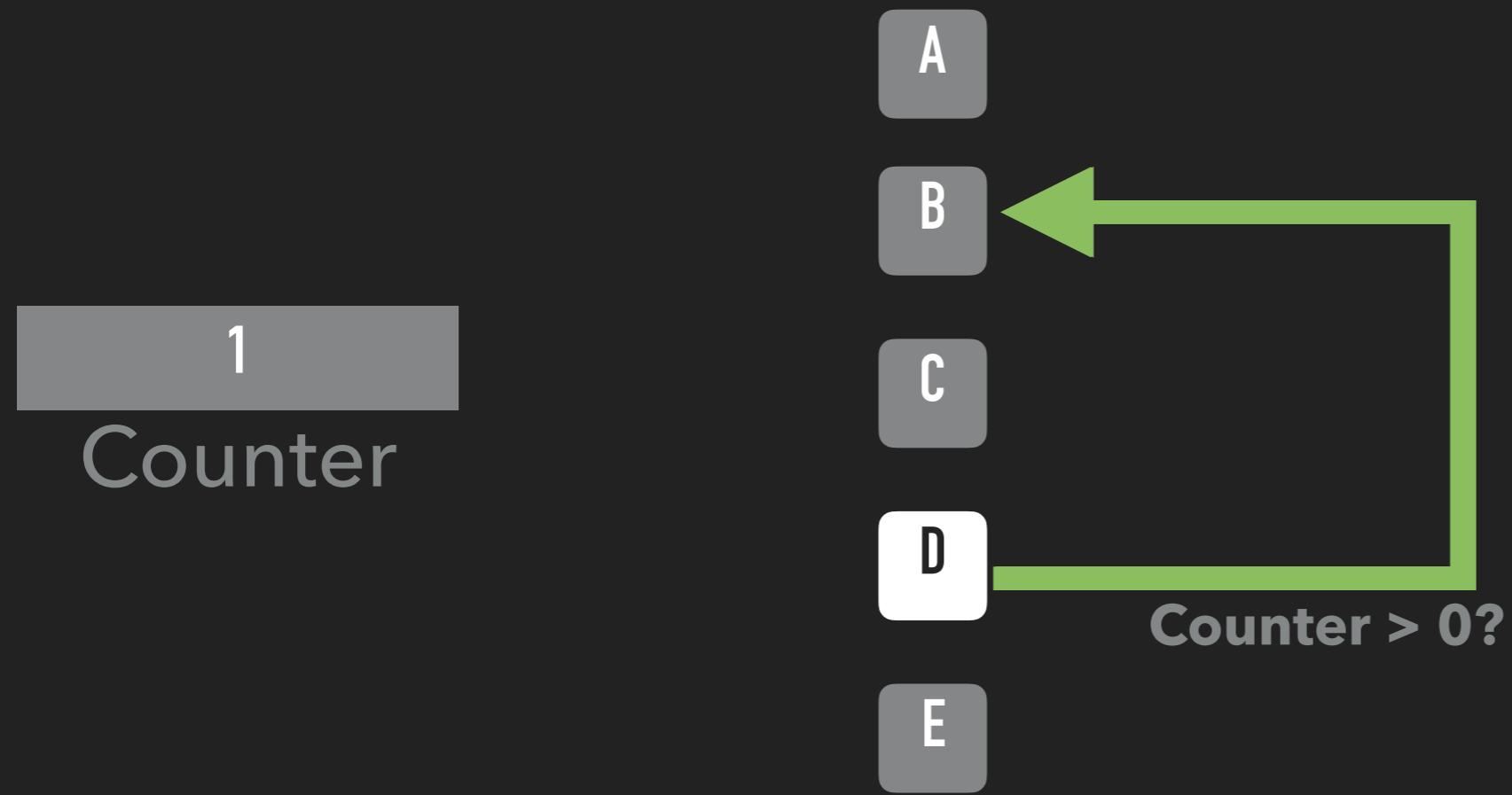


1  
Counter



The CPU has learned that Counter *probably* is  $> 0$

## SPECTRE: SPECULATIVE EXECUTION



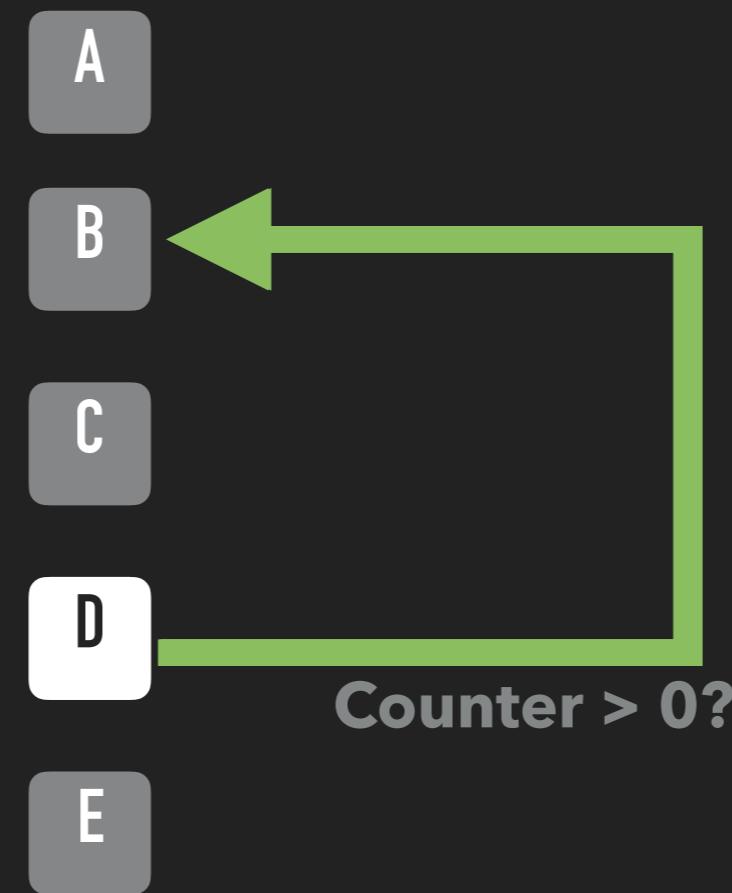
The CPU has learned that Counter *probably* is  $> 0$

Reading Counter from memory is very slow

## SPECTRE: SPECULATIVE EXECUTION



1  
Counter

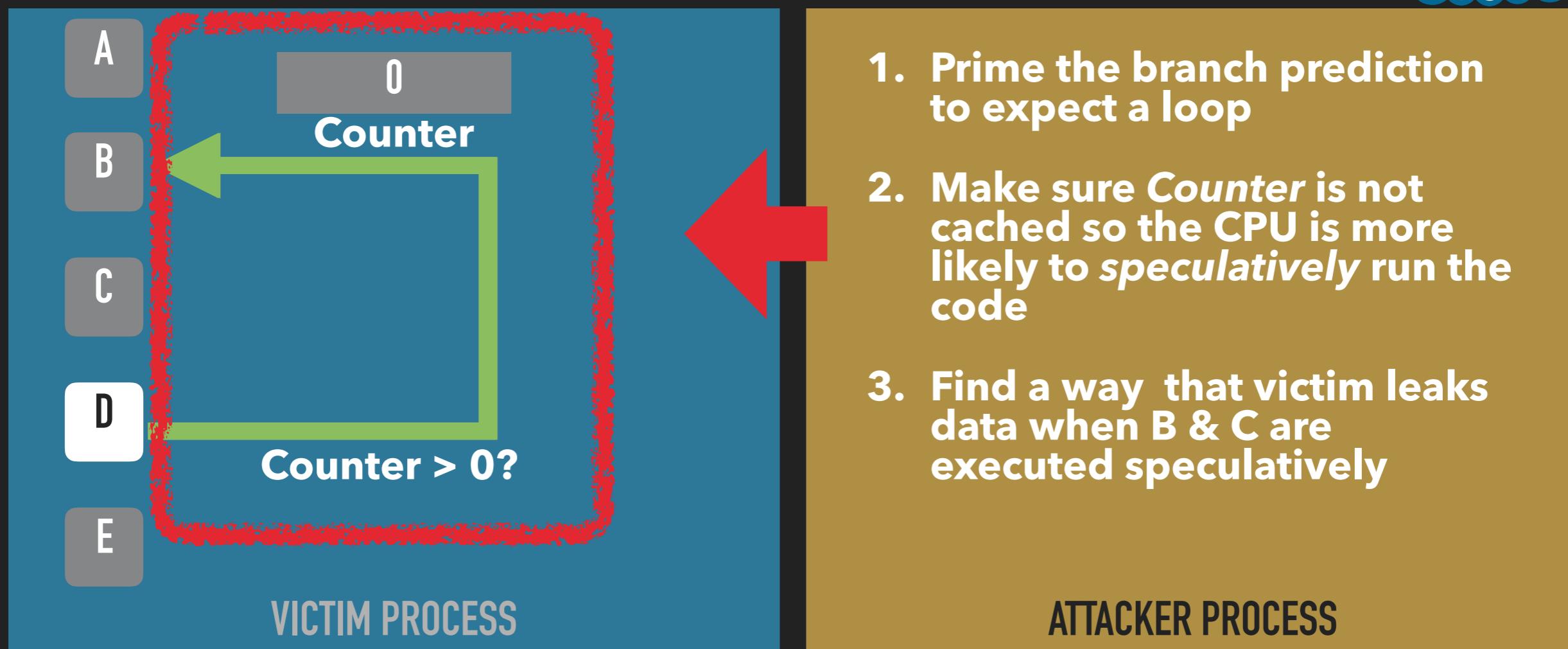


The CPU has learned that Counter *probably* is  $> 0$

Reading Counter from memory is very slow

The CPU *speculatively* executes B C to improve performance

## SPECTRE: SPECULATIVE EXECUTION



Attacker can influence the CPUs branch prediction of victim.

Making the victim *speculatively* execute “wrong” code.

E.g. loop even when Counter is == 0.

## SPECTRE: VARIANT 1 (CVE-2017-5753)



```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- ▶ This is code of the victim
- ▶ `x` is controlled by the attacker
- ▶ attacker wants to read `array1[x]` out of bounds
- ▶ `array2` is used to leak the value of `y` (like in Meltdown)

## SPECTRE: VARIANT 1 (CVE-2017-5753)



```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- ▶ This is code of the victim
  - ▶ **x** is controlled by the attacker
  - ▶ attacker wants to read **array1[x]** out of bounds
  - ▶ **array2** is used to leak the value of y (like in Meltdown)
1. Attacker manipulates branch prediction

## SPECTRE: VARIANT 1 (CVE-2017-5753)



```
if (x < array1_size)
```

```
    y = array2[array1[x] * 256];
```

- ▶ This is code of the victim
  - ▶ `x` is controlled by the attacker
  - ▶ attacker wants to read `array1[x]` out of bounds
  - ▶ `array2` is used to leak the value of `y` (like in Meltdown)
1. Attacker manipulates branch prediction
  2. Speculatively runs even when `x > array1_size`

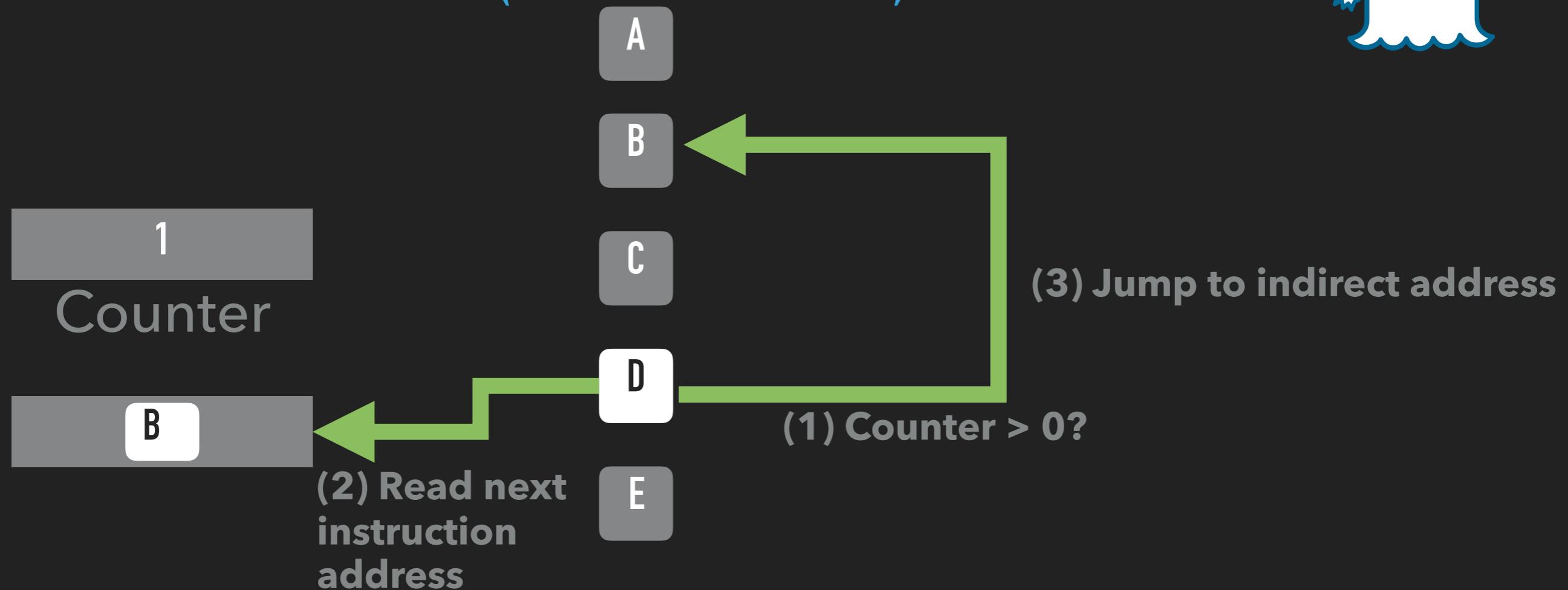
## SPECTRE: VARIANT 1 (CVE-2017-5753)



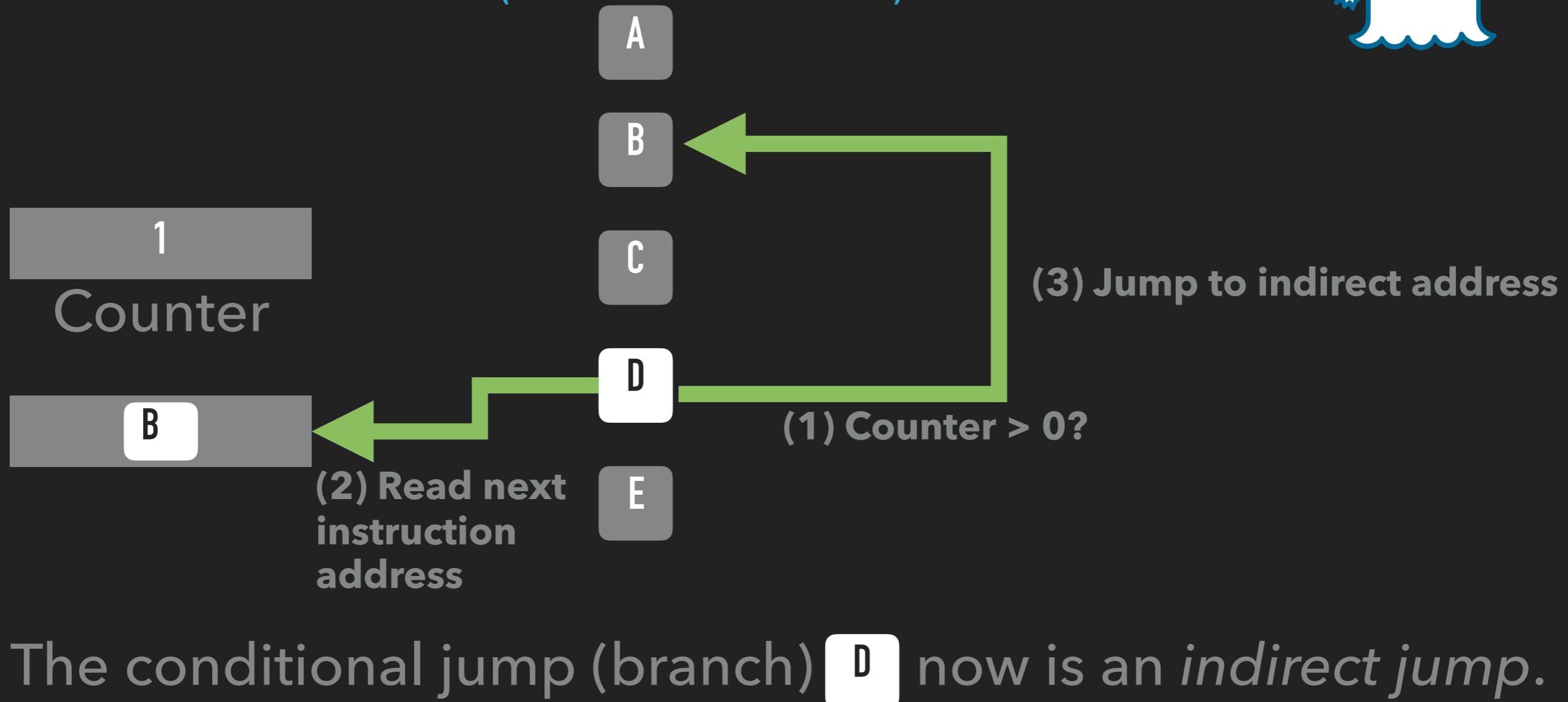
```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- ▶ This is code of the victim
  - ▶ `x` is controlled by the attacker
  - ▶ attacker wants to read `array1[x]` out of bounds
  - ▶ `array2` is used to leak the value of `y` (like in Meltdown)
1. Attacker manipulates branch prediction
  2. Speculatively runs even when `x > array1_size`
  3. The cache timing of `array2[..]` leaks the value `array1[x]`

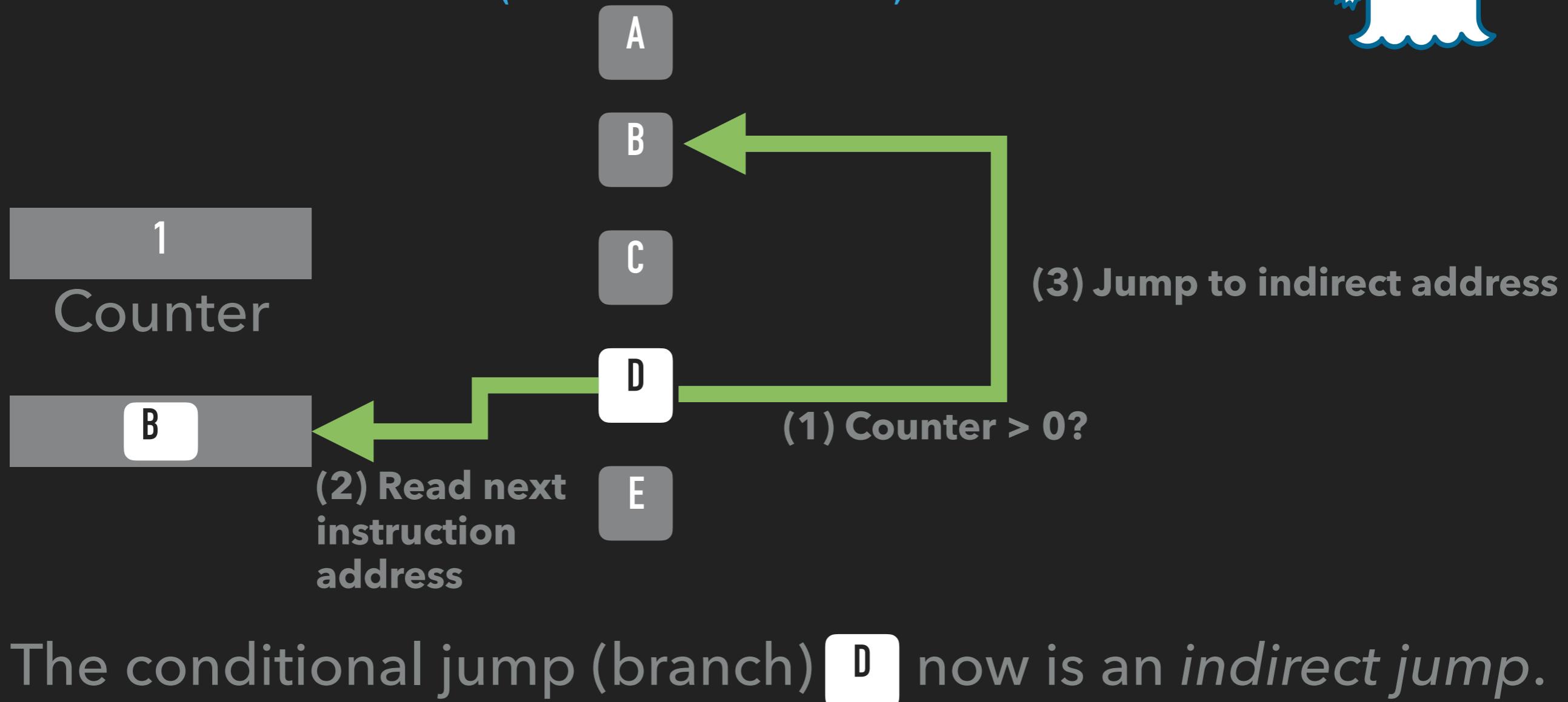
## SPECTRE: VARIANT 2 (CVE-2017-5715)



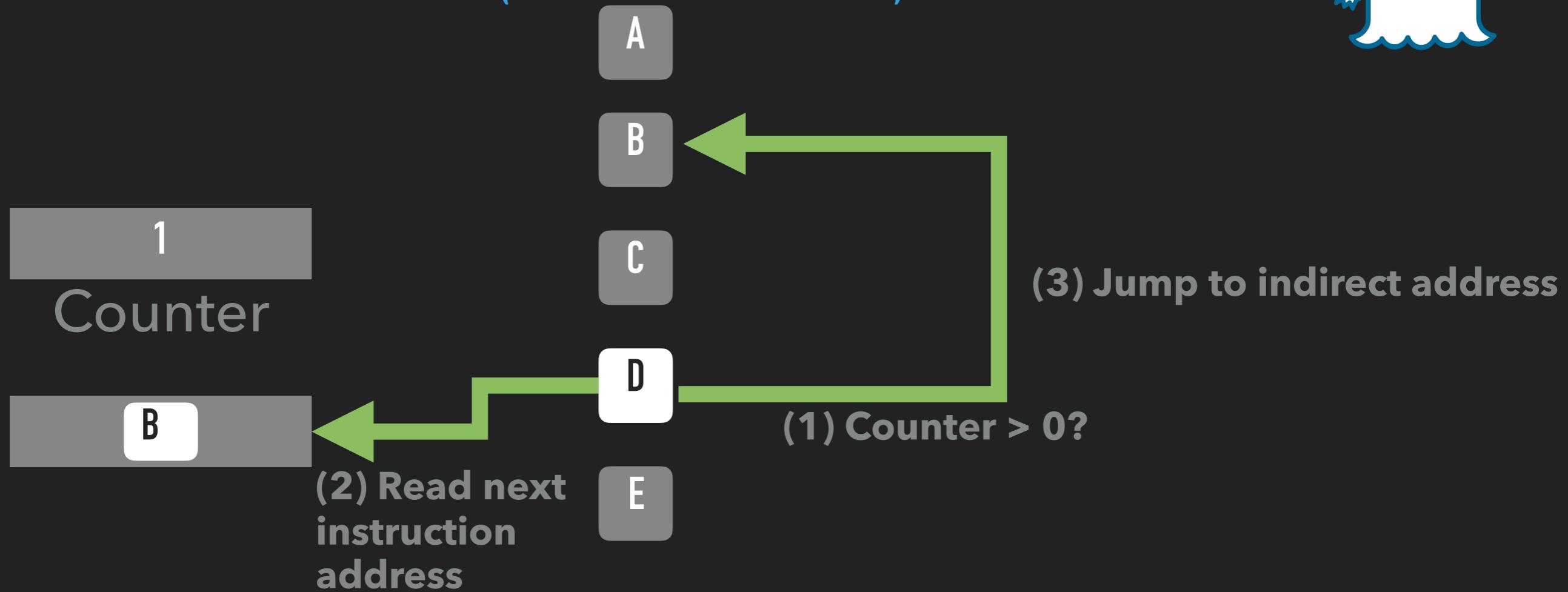
## SPECTRE: VARIANT 2 (CVE-2017-5715)



## SPECTRE: VARIANT 2 (CVE-2017-5715)

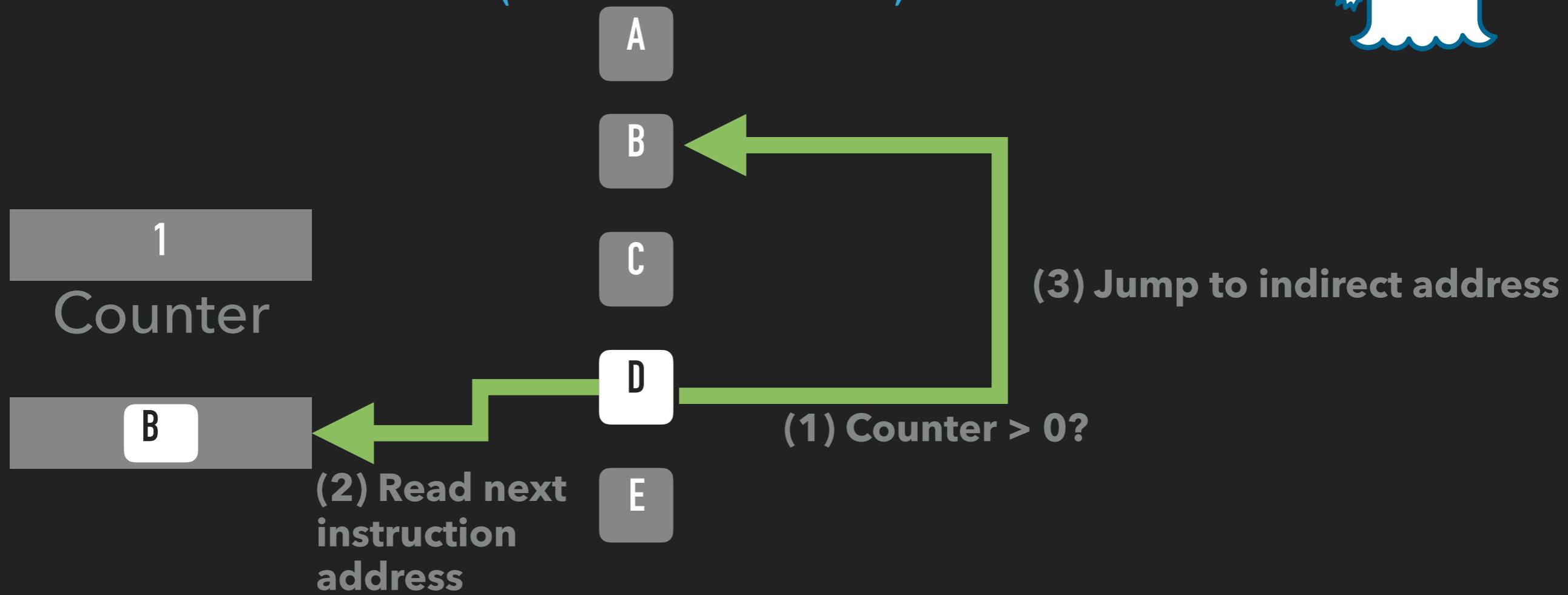


## SPECTRE: VARIANT 2 (CVE-2017-5715)



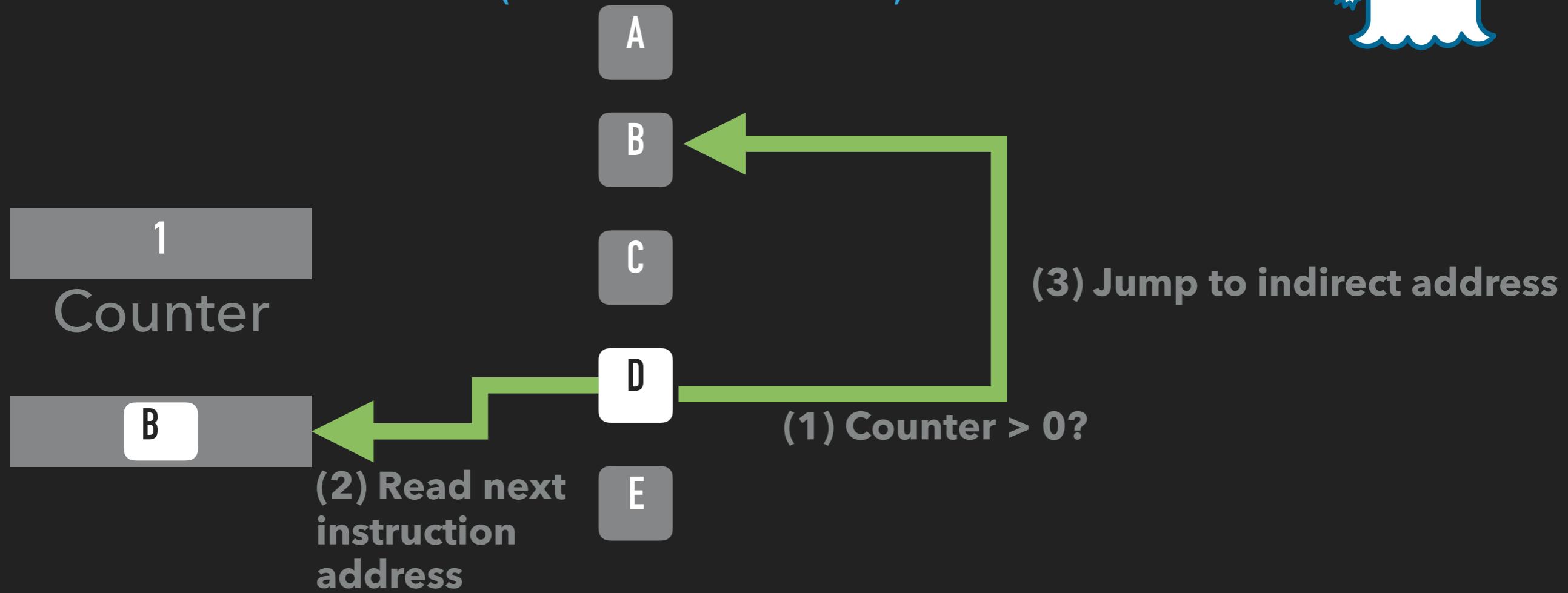
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

## SPECTRE: VARIANT 2 (CVE-2017-5715)



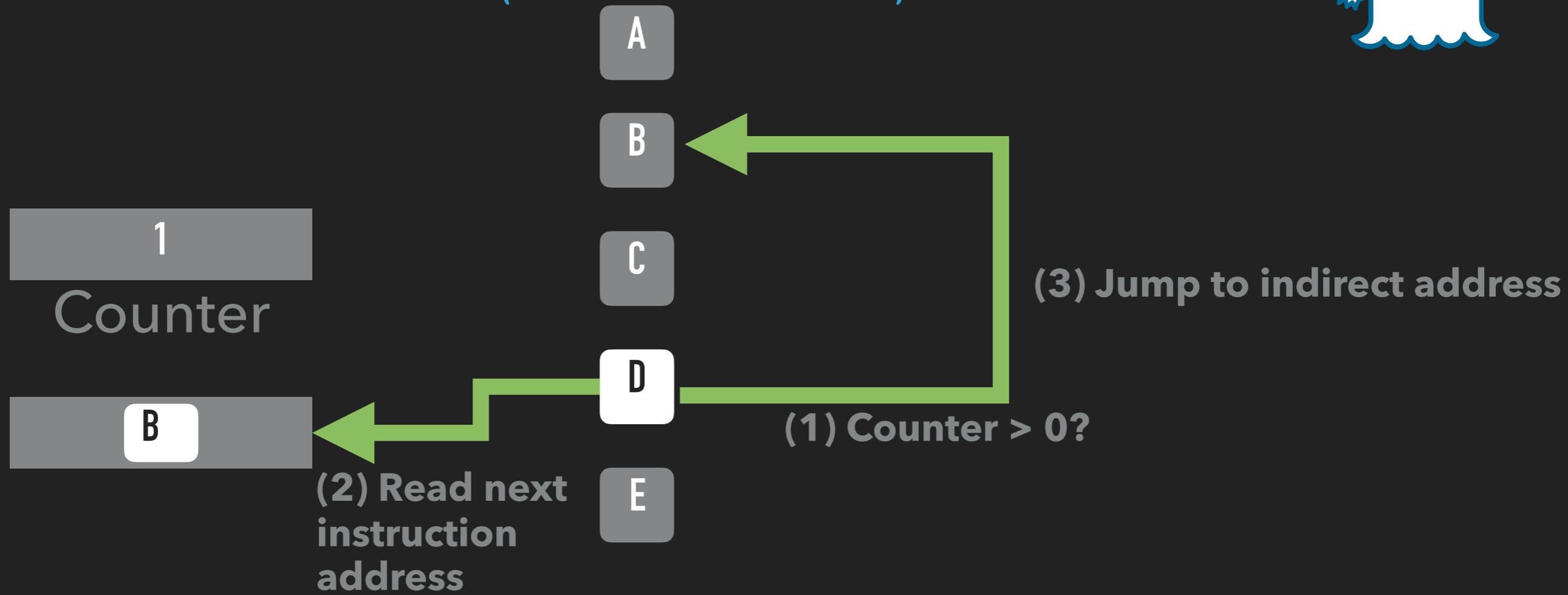
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

## SPECTRE: VARIANT 2 (CVE-2017-5715)



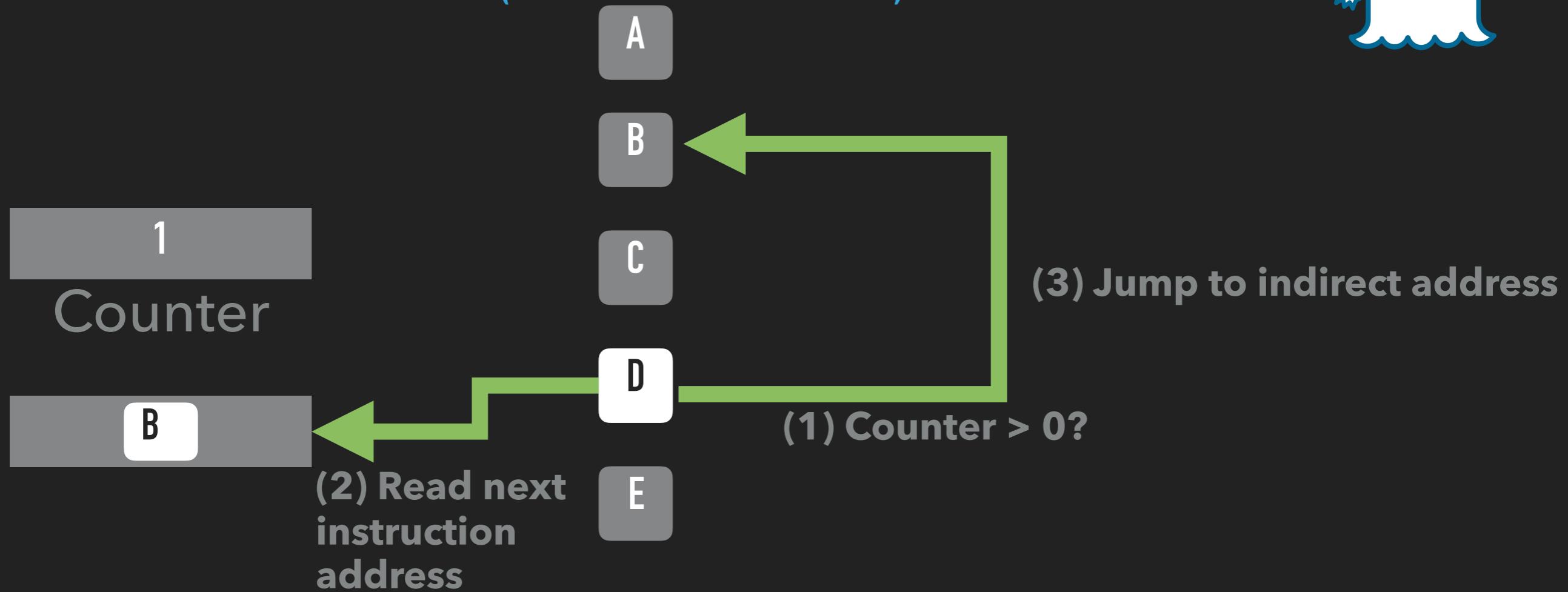
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

## SPECTRE: VARIANT 2 (CVE-2017-5715)



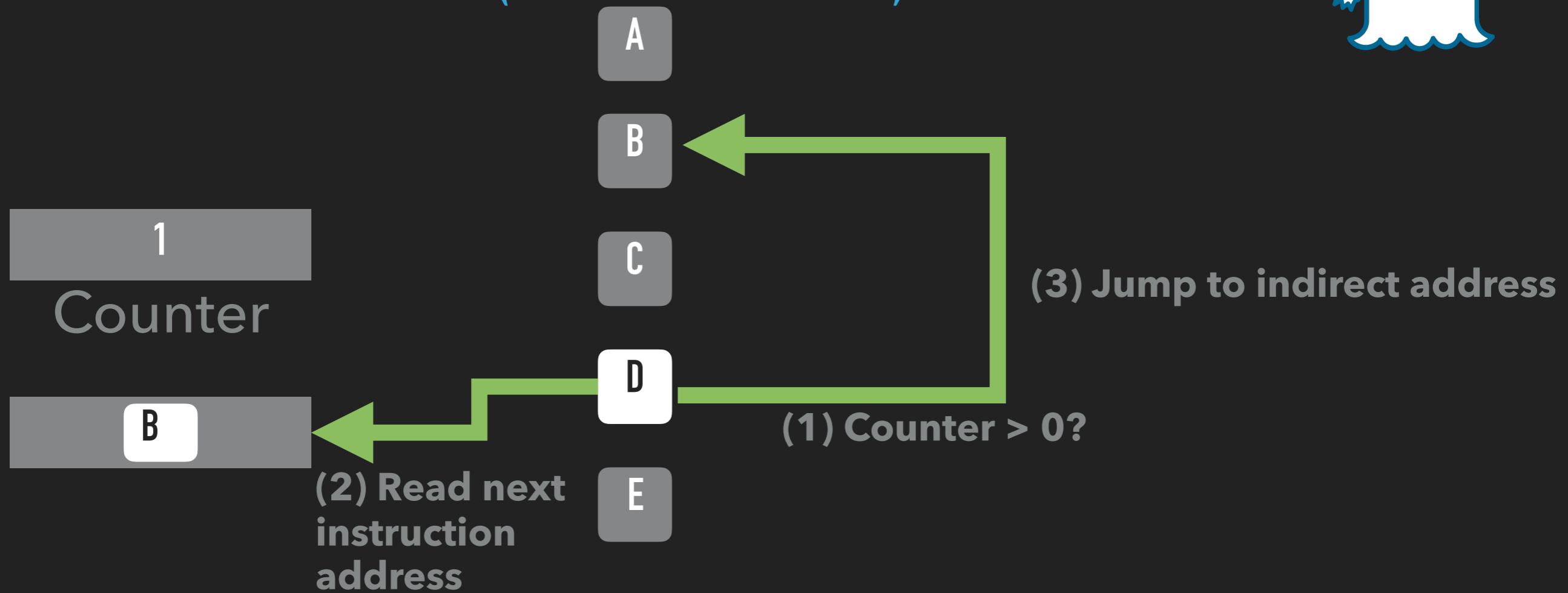
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

## SPECTRE: VARIANT 2 (CVE-2017-5715)



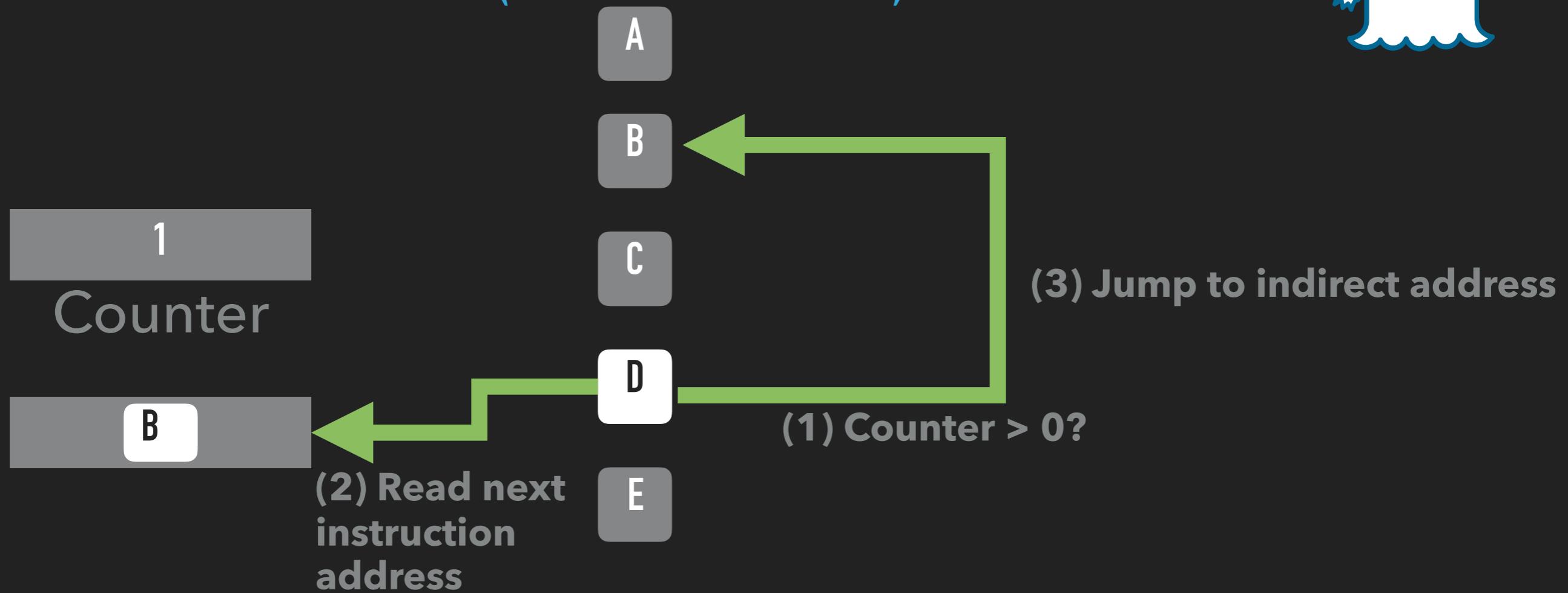
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

## SPECTRE: VARIANT 2 (CVE-2017-5715)



- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".
- ▶ This can also be used to *speculatively* execute any code found in the target process (kernel).

## SPECTRE: VARIANT 2 (CVE-2017-5715)



- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".
- ▶ This can also be used to *speculatively* execute any code found in the target process (kernel).

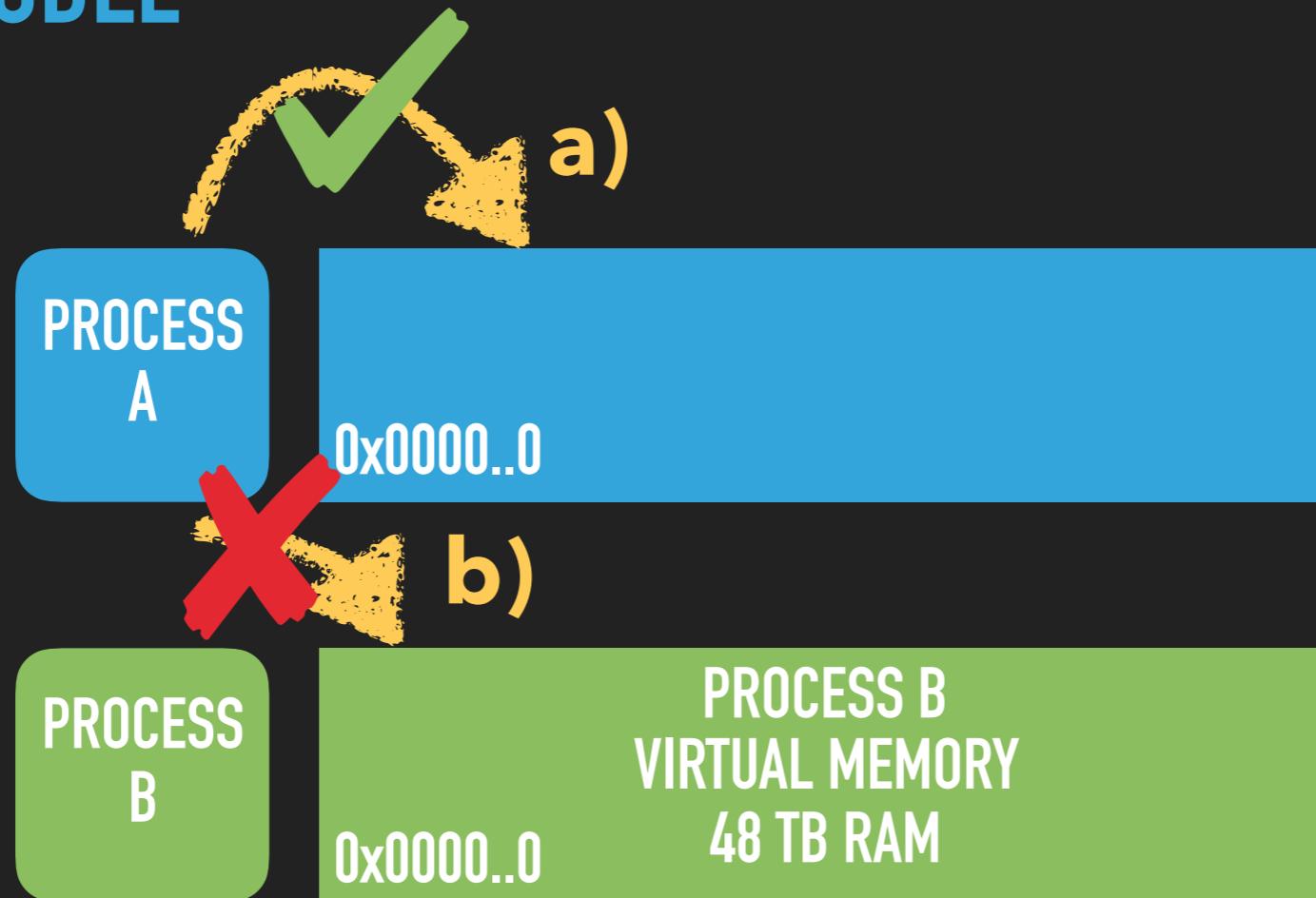


♩ ♪ INTERLUDE ♪ ♪

---

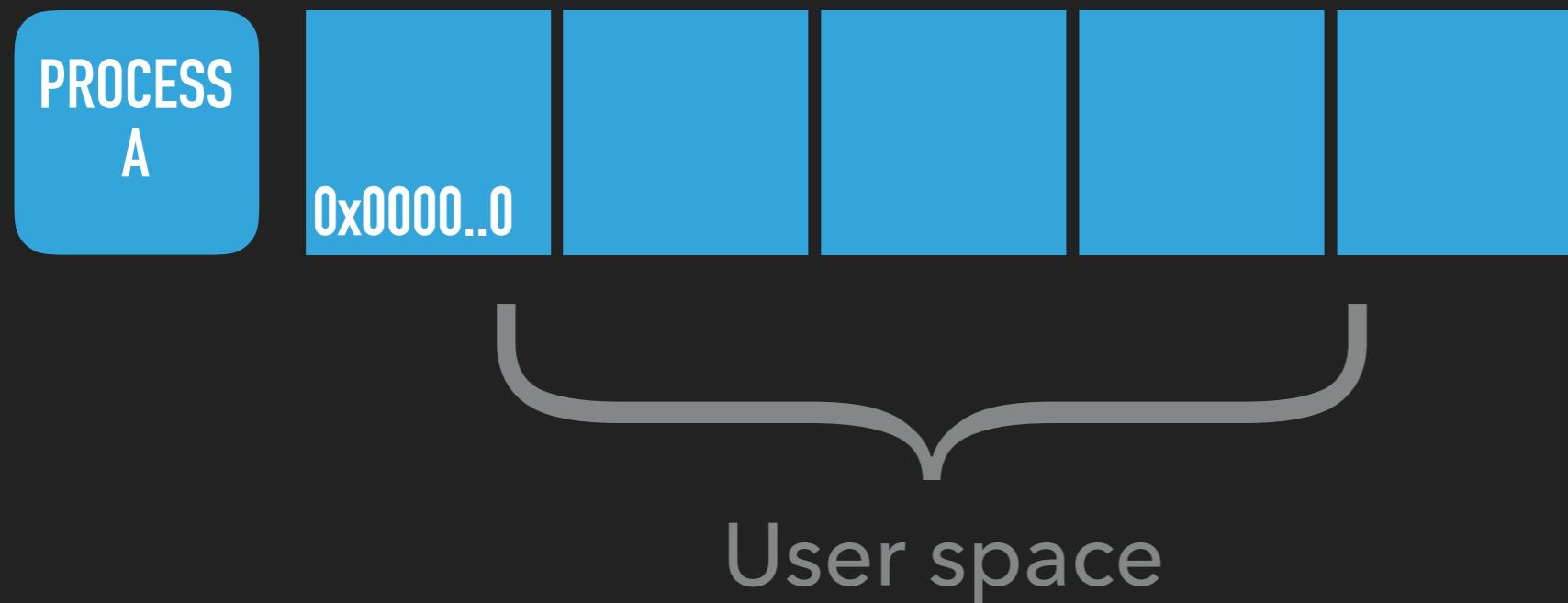
# MEMORY MODEL

## MEMORY MODEL

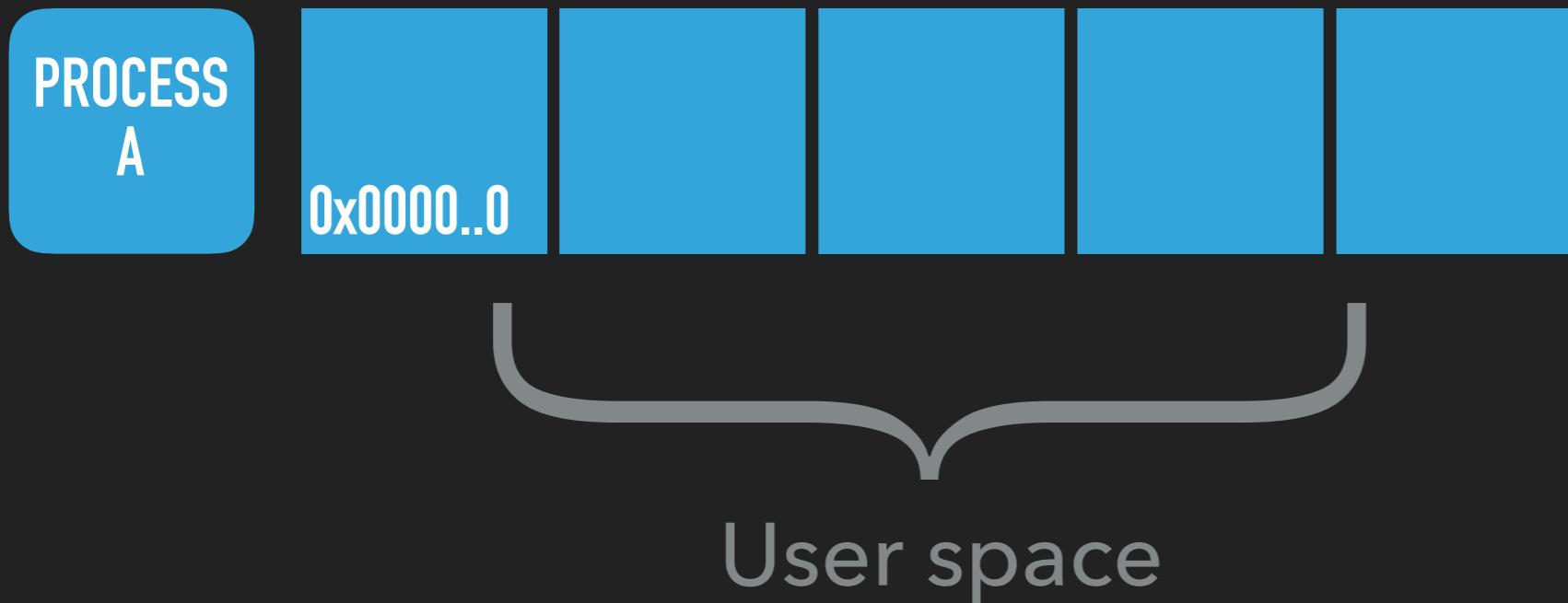


- ▶ CPU and OS isolate processes memory from each other
- ▶ *Virtual Memory* gives each process its own *address space*
- ▶ Each address space starts at “virtual address **0x000..0**”

## MEMORY MODEL

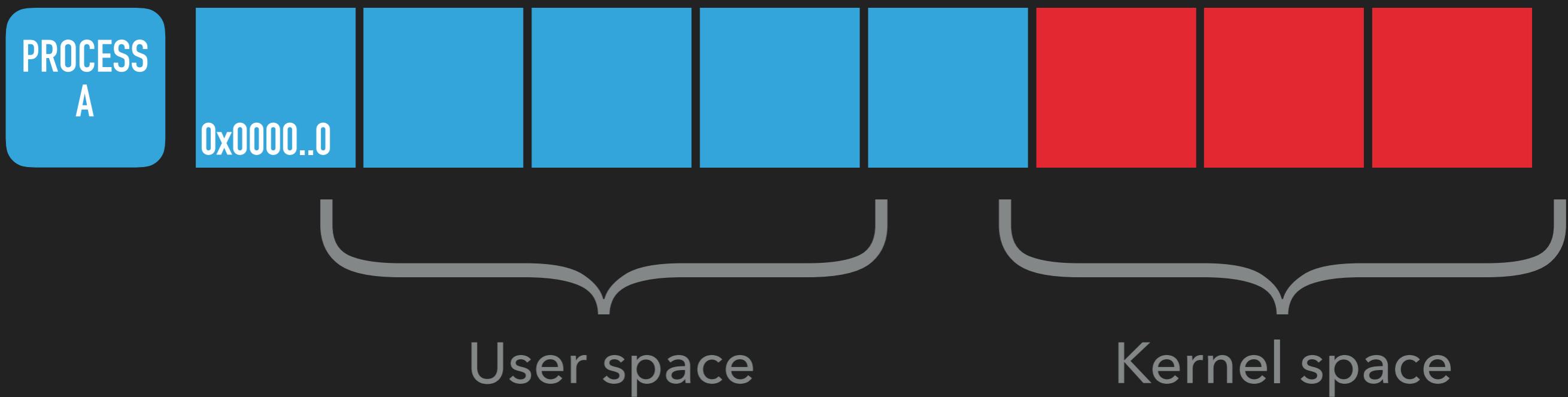


## MEMORY MODEL



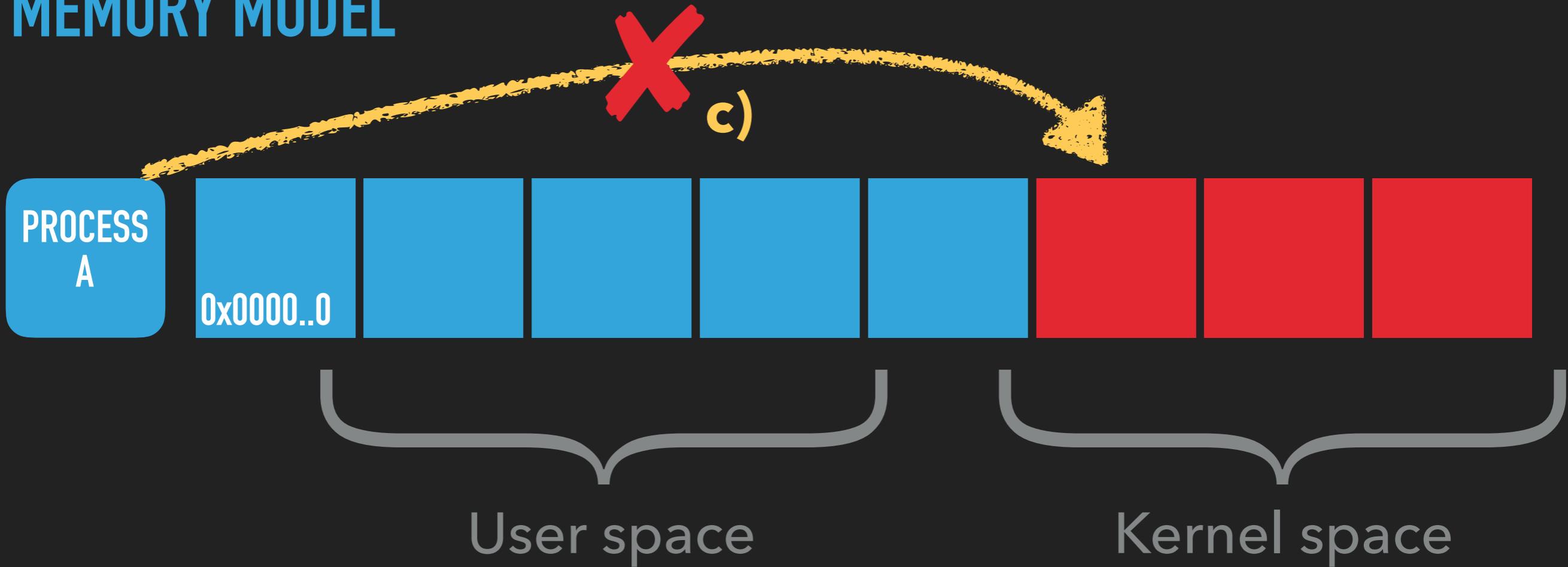
- ▶ Memory is split into *pages* (each 4KiB on x86)

## MEMORY MODEL



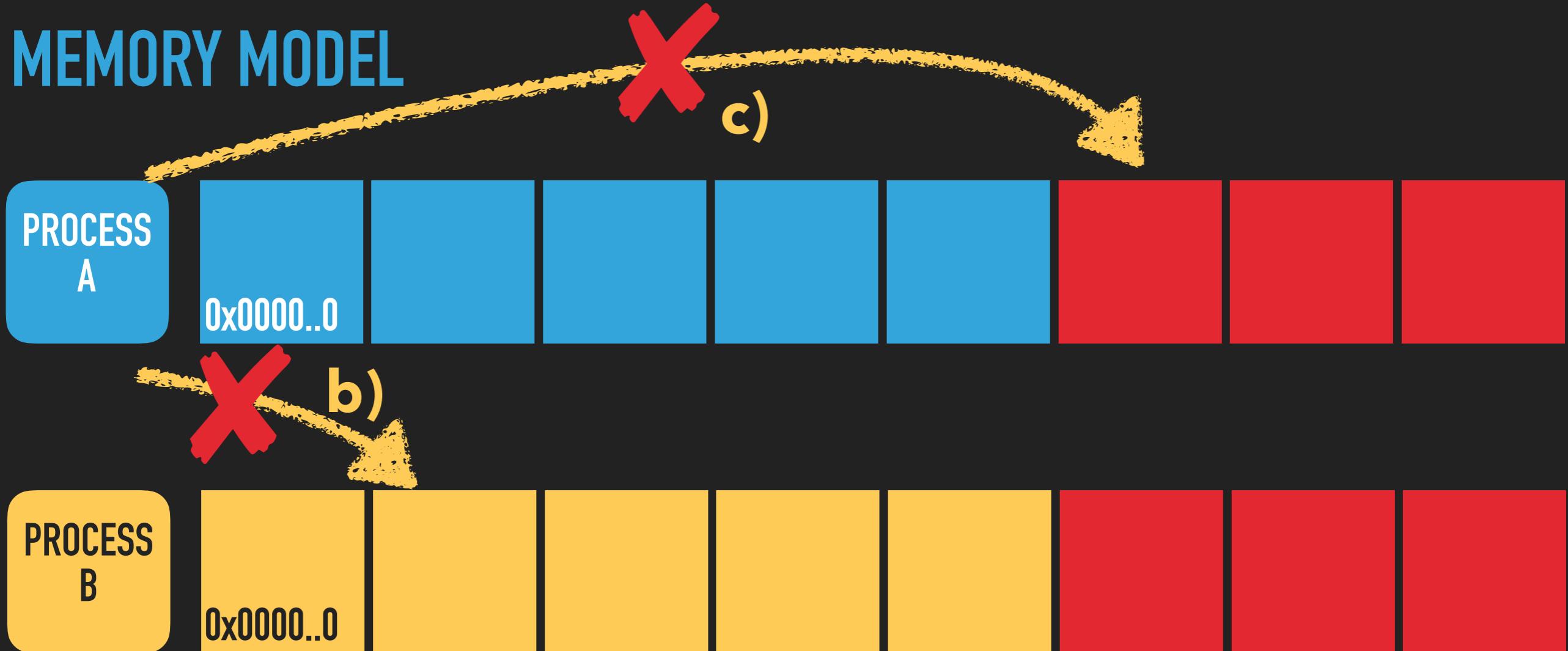
- ▶ Memory is split into *pages* (each 4KiB on x86)
- ▶ The kernel *maps* its own memory into each process

## MEMORY MODEL



- ▶ Memory is split into *pages* (each 4KiB on x86)
- ▶ The kernel *maps* its own memory into each process
- ▶ This “kernel” memory is only accessible by the kernel

## MEMORY MODEL

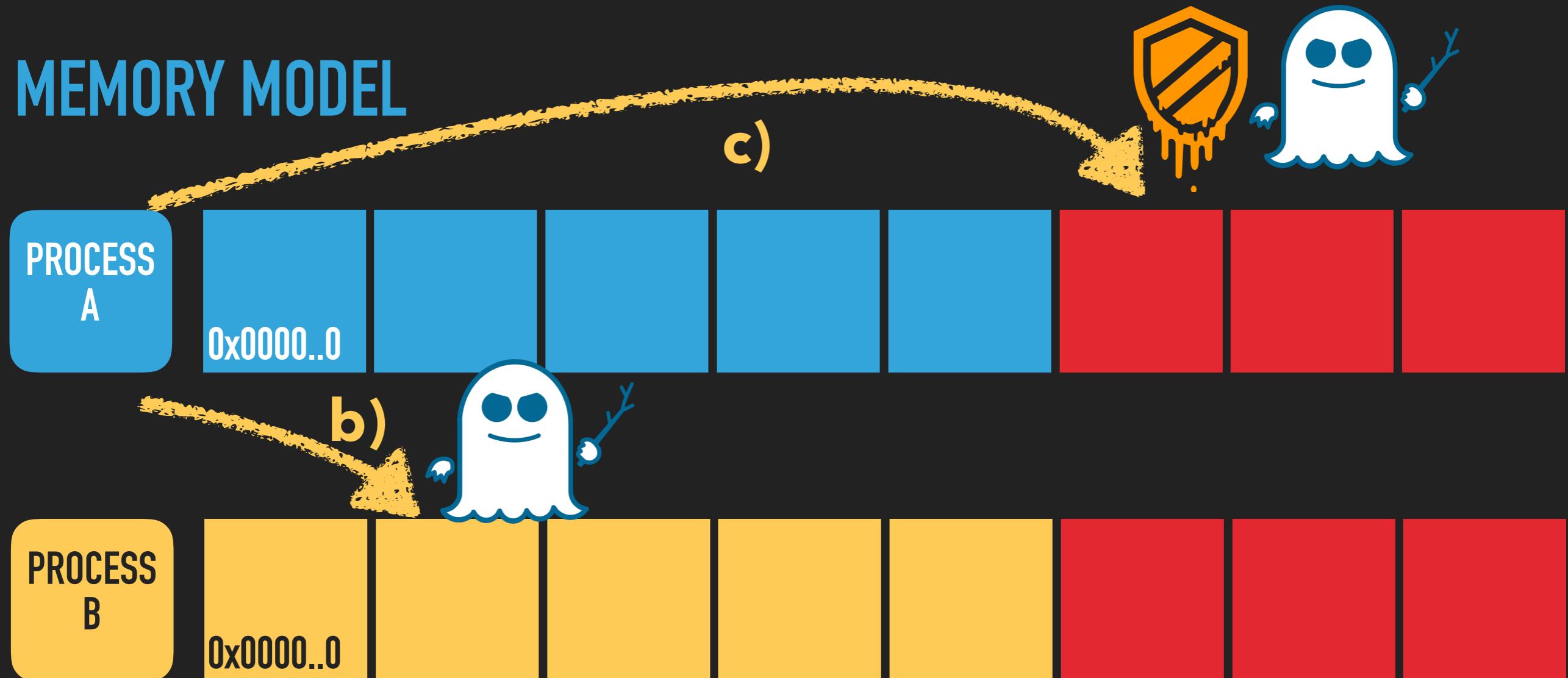


**b) and c) are completely different error scenarios**

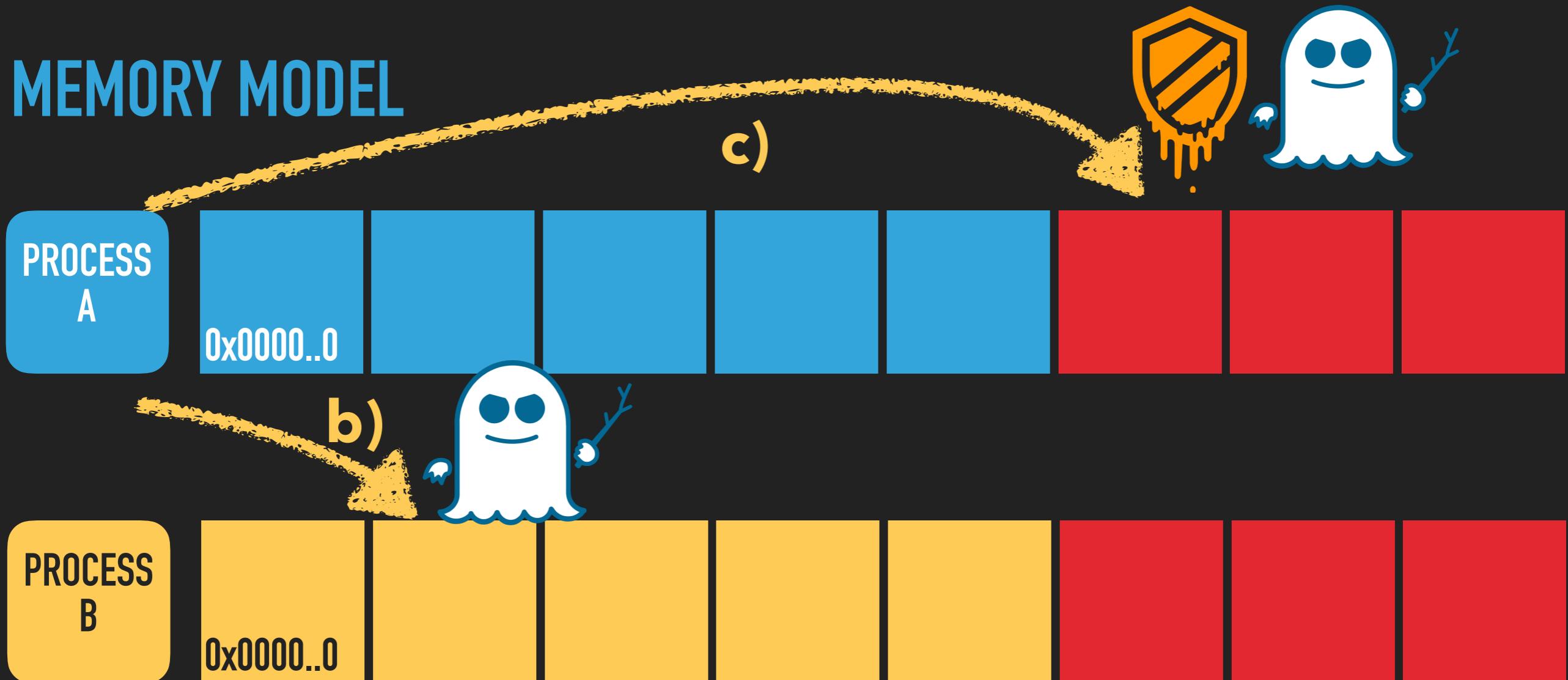
- ▶ **c)** Kernel memory pages are marked “kernel only” but the process could try to access the pages via a pointer
- ▶ **b)** Process **B** has no possibility to even *describe* the address

# MELTDOWN & SPECTRE FOR NORMAL PEOPLE

## MEMORY MODEL

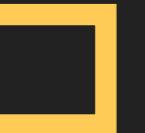


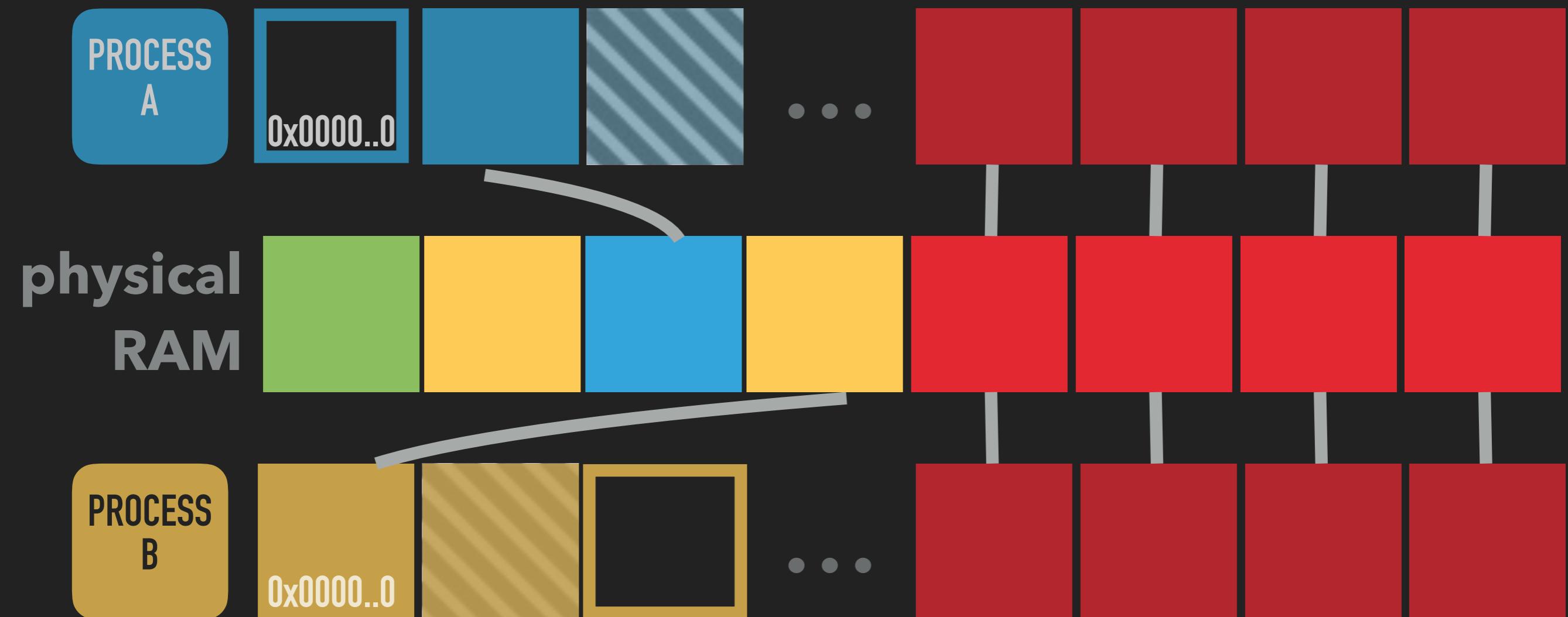
## MEMORY MODEL



- ▶ **c)** is vulnerable to Meltdown and Spectre
- ▶ **b)** is vulnerable to Spectre

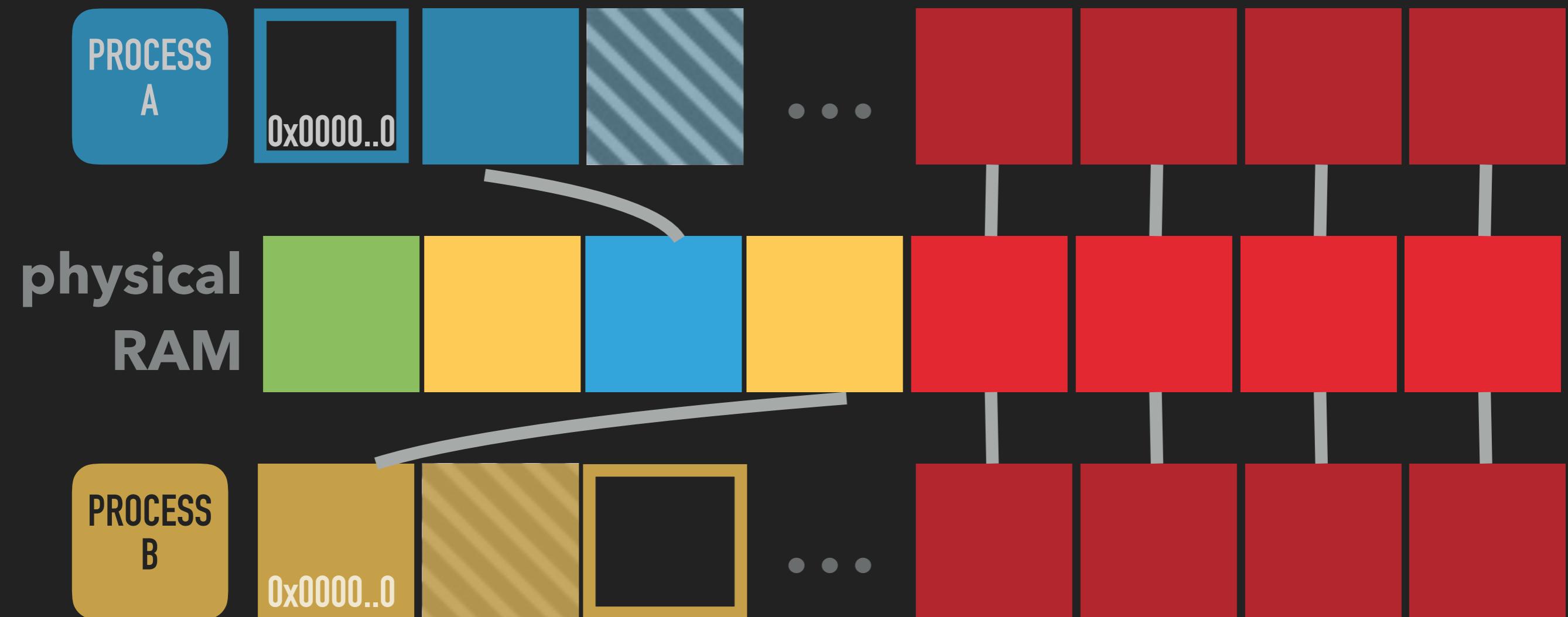
## MEMORY MODEL

- ▶ Virtual memory is backed by physical RAM
- ▶ Virtual memory is much, much larger than physical RAM
- ▶ Not all virtual memory is backed by RAM, e.g.  



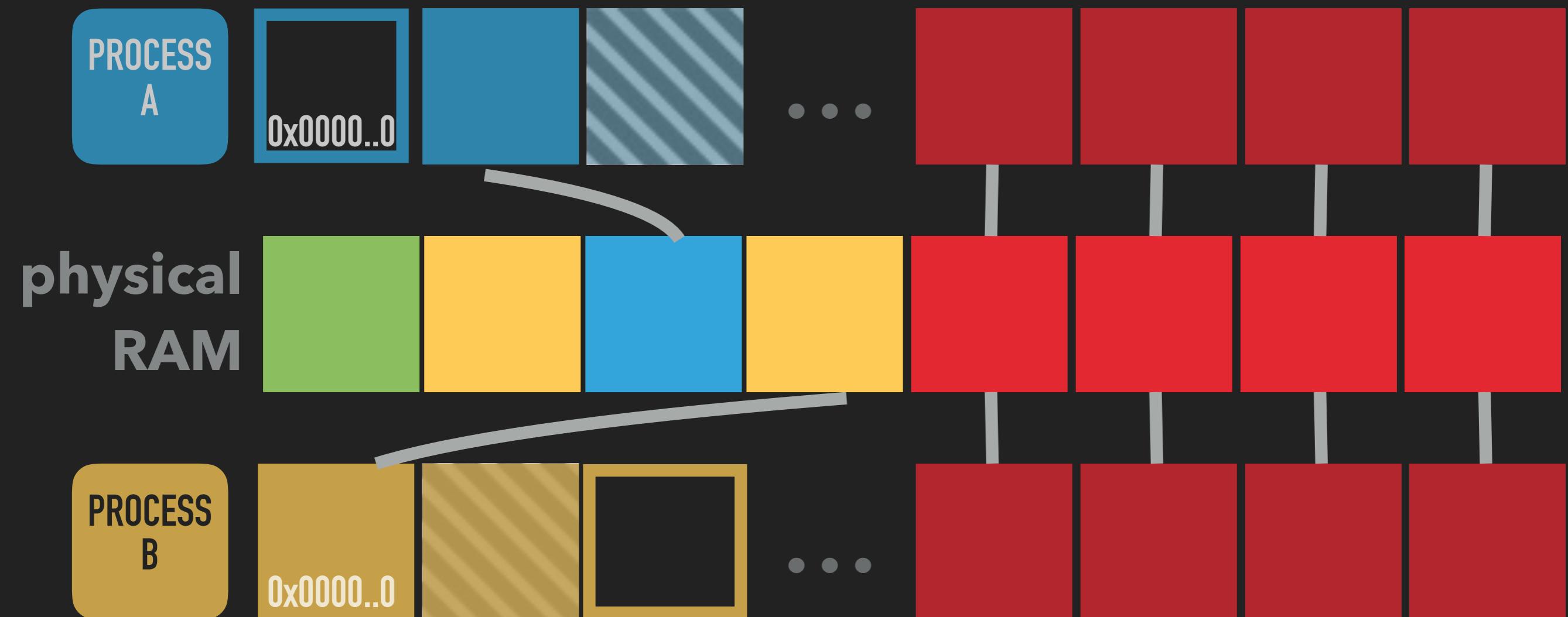
## MEMORY MODEL

- ▶ Virtual memory is backed by physical RAM
- ▶ Virtual memory is much, much larger than physical RAM
- ▶ Not all virtual memory is backed by RAM, e.g.  



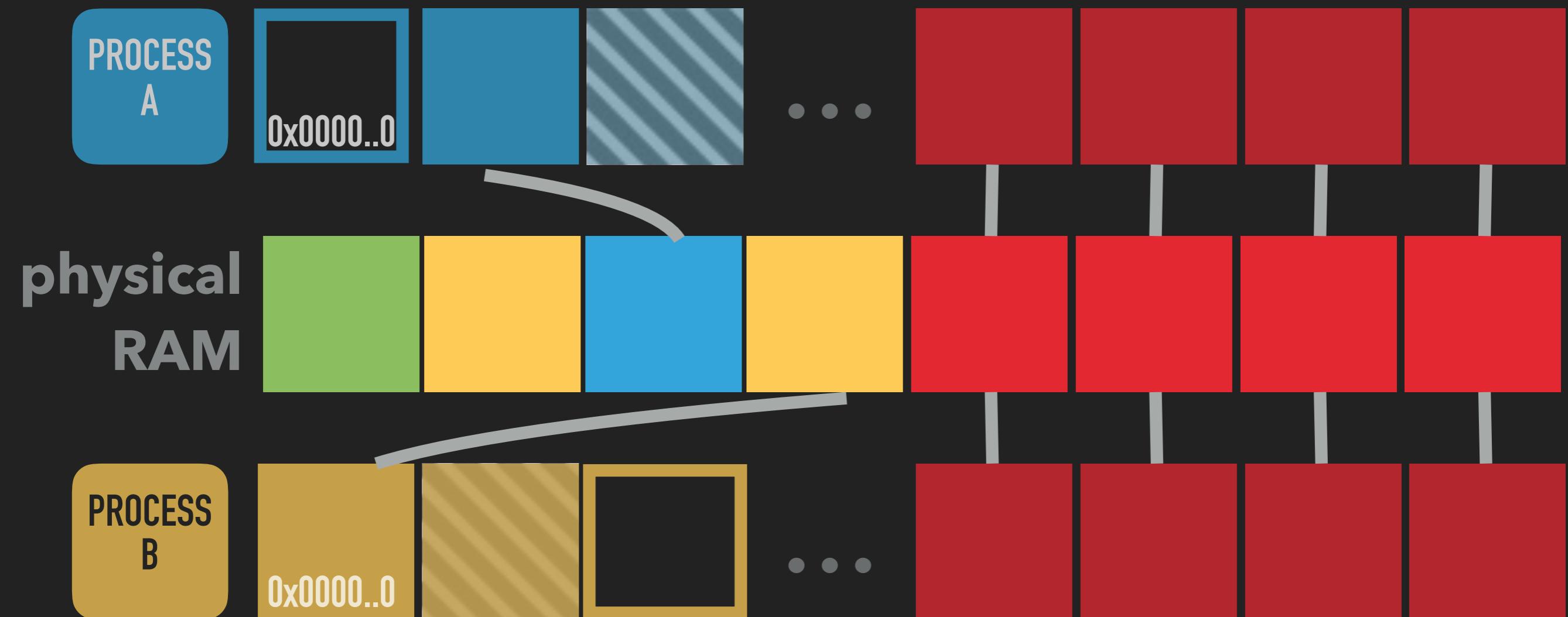
## MEMORY MODEL

- ▶ Virtual memory is backed by physical RAM
- ▶ Virtual memory is much, much larger than physical RAM
- ▶ Not all virtual memory is backed by RAM, e.g.  

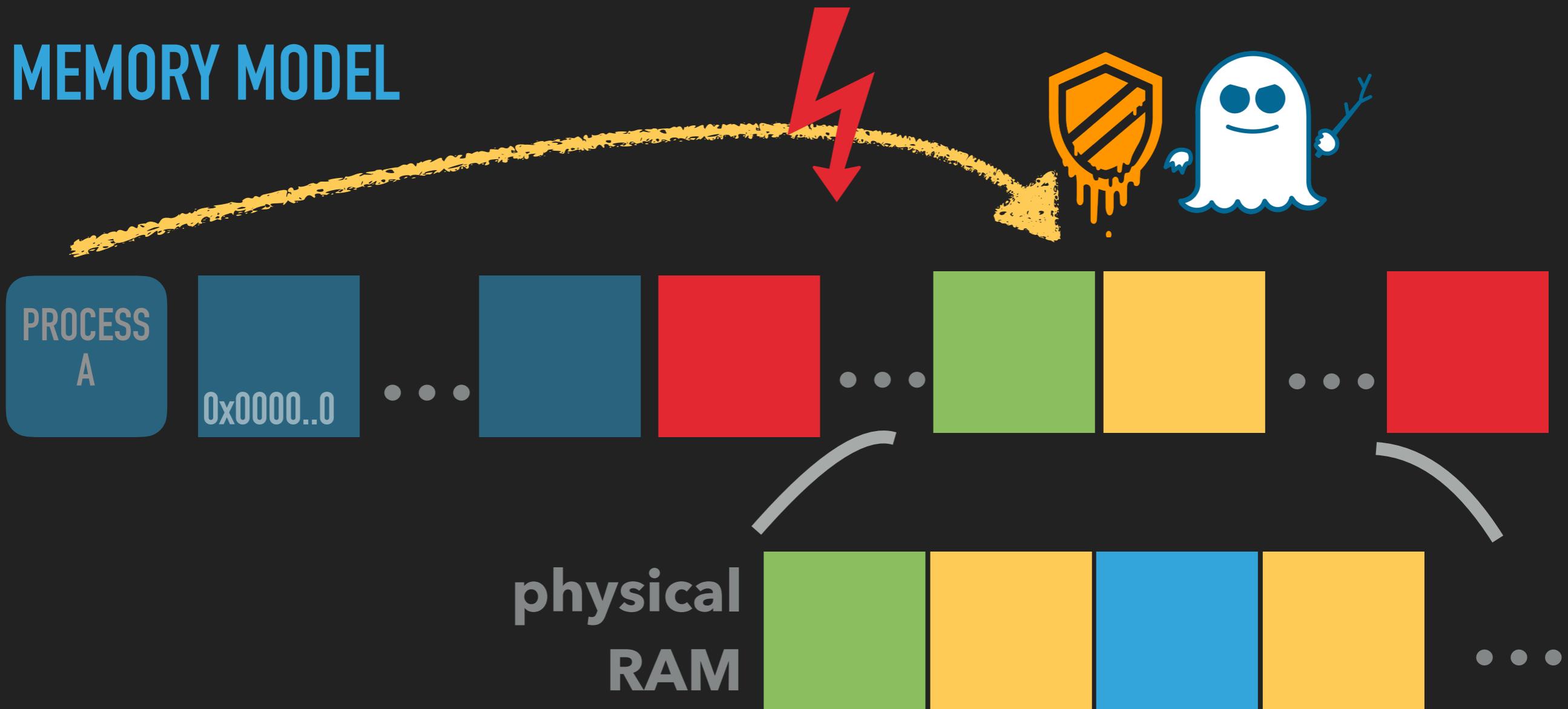


## MEMORY MODEL

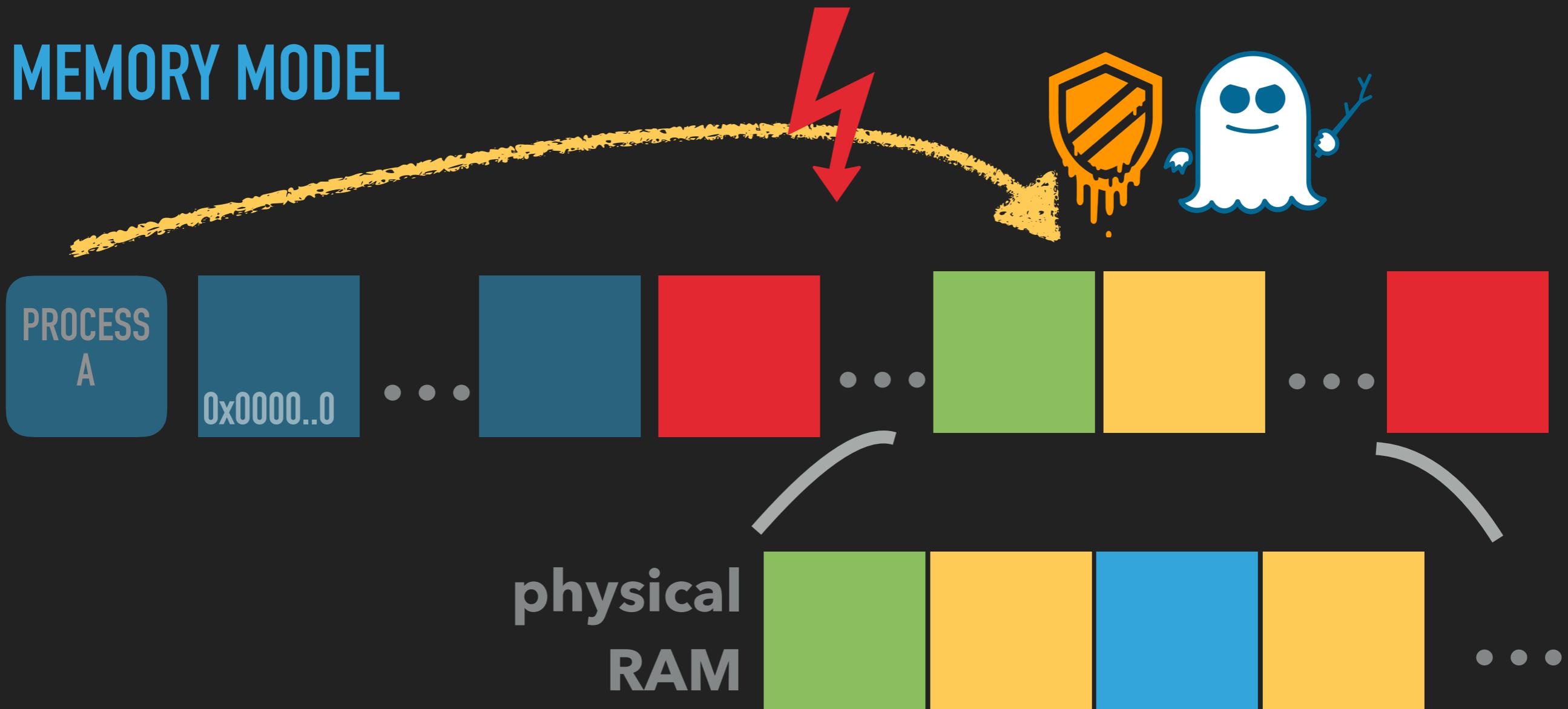
- ▶ Virtual memory is backed by physical RAM
- ▶ Virtual memory is much, much larger than physical RAM
- ▶ Not all virtual memory is backed by RAM, e.g.  



## MEMORY MODEL



## MEMORY MODEL



- ▶ Like a matryoshka doll the kernel *maps all physical* memory into its address space
- ▶ Reading kernel memory allows reading of all (mapped) memory of all processes

## MEMORY MODEL

Virtual memory map with 4 level page tables:

```
0000000000000000 - 00007ffffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffffff (=64 TB) direct mapping of all phys. memory
fffffc80000000000 - ffffc8ffffffffffff (=40 bits) hole
fffffc90000000000 - ffffe8ffffffffffff (=45 bits) vmalloc/ioremap space
fffffe90000000000 - ffffe9ffffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaafffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...

vaddr_end for KASLR
fffffe0000000000 - fffffe7ffffffffffff (=39 bits) cpu_entry_area mapping
fffffe8000000000 - fffffefffffffffff (=39 bits) LDT remap for PTI
ffffff0000000000 - ffffff7ffffffffffff (=39 bits) %esp fixup stacks
... unused hole ...
fffffef00000000 - ffffffefffffffff (=64 GB) EFI region mapping space
... unused hole ...
fffffff80000000 - ffffff9ffffffffffff (=512 MB) kernel text mapping, from phys 0
fffffff80000000 - ffffffeffffefffffff (1520 MB) module mapping space
[fixmap start] - ffffff5fffff5fffff kernel-internal fixmap range
fffffff600000 - ffffff600ffff600ffff (=4 kB) legacy vsyscall ABI
fffffff800000 - ffffffefffffefffff (=2 MB) unused hole
```



MELTDOWN AND  
SPECTRE

---

CONCLUSION

## THREAT-O-METER

**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

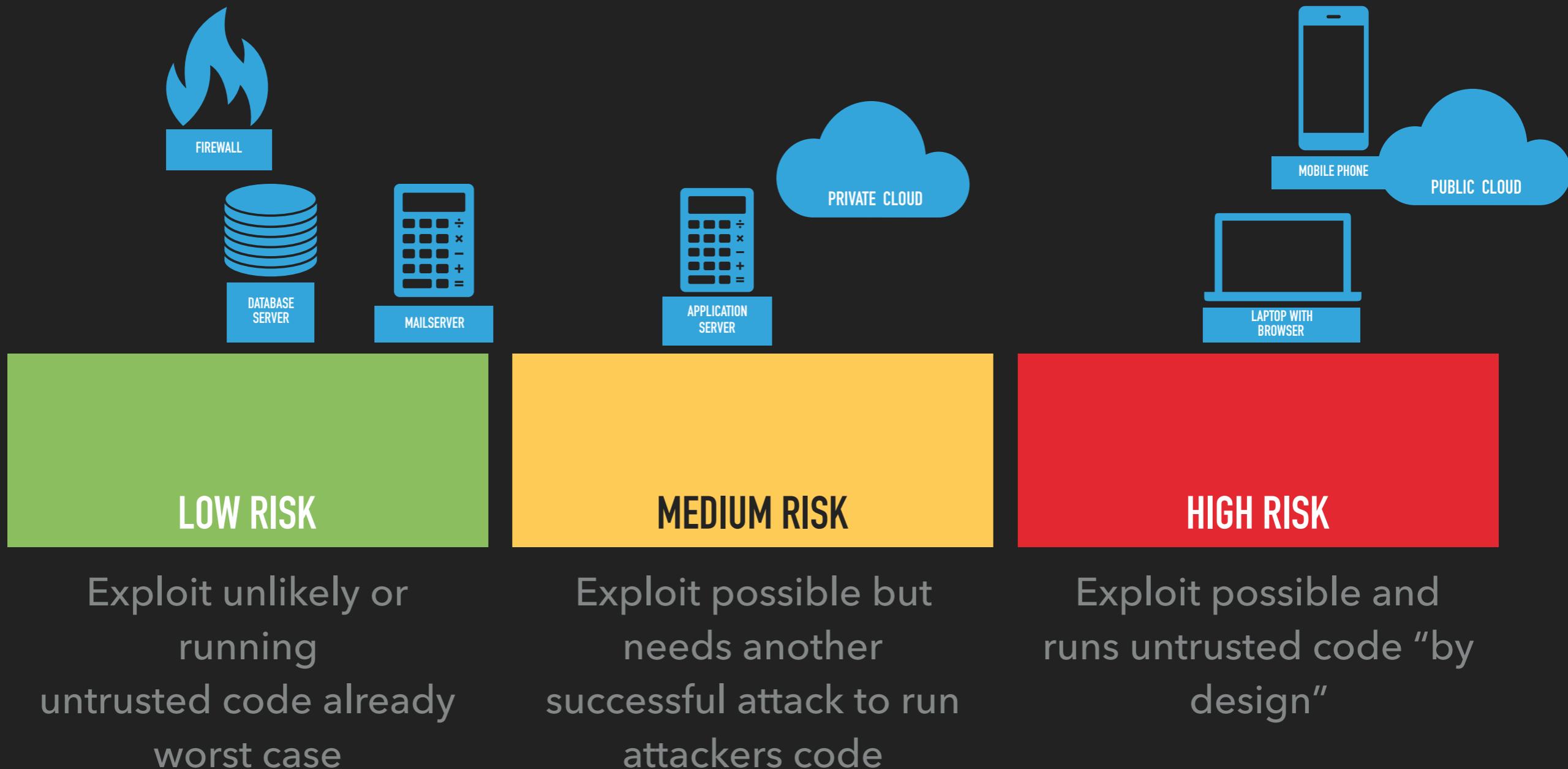
**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER



## THREAT-O-METER

Meltdown and Spectre exploit side effects of modern CPU architectures.



**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code "by  
design"

## THREAT-O-METER

Meltdown and Spectre exploit side effects of modern CPU architectures.

The *exploitability* very much depends on the field of application of the system.



**LOW RISK**

Exploit unlikely or running untrusted code already worst case

**MEDIUM RISK**

Exploit possible but needs another successful attack to run attackers code

**HIGH RISK**

Exploit possible and runs untrusted code "by design"

## THREAT-O-METER

Meltdown and Spectre exploit side effects of modern CPU architectures.

The *exploitability* very much depends on the field of application of the system.

Expect new “bugs” of this type!

**LOW RISK**

Exploit unlikely or  
running  
untrusted code already  
worst case

**MEDIUM RISK**

Exploit possible but  
needs another  
successful attack to run  
attackers code

**HIGH RISK**

Exploit possible and  
runs untrusted code “by  
design”



# Knock knock

## Branch prediction

### Who's there?

# Q & A

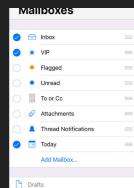




BOTH THE MELTDOWN AND SPECTRE LOGO ARE FREE TO USE, RIGHTS WAIVED  
VIA [CCO](#). LOGOS ARE DESIGNED BY [NATASCHA EIBL](#).  
[HTTPS://SPECTREATTACK.COM/](https://spectreattack.com/)



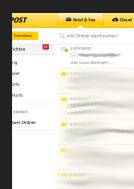
CANDY CRUSH LOGO FROM THE APP STORE © KING  
[HTTPS://DISCOVER.KING.COM/ABOUT/](https://discover.king.com/about/)



APPLE MAIL SCREENSHOT © APPLE  
[HTTPS://SUPPORT.APPLE.COM/EN-GB/HT207213](https://support.apple.com/en-gb/ht207213)



SCREENSHOT © BUNDESNACHRICHTENDIENST  
[HTTPS://WWW.BND.BUND.DE/EN/\\_HOME/HOME\\_NODE.HTML](https://www.bnd.bund.de/en/_home/home_node.html)



SCREENSHOT E-POST  
[HTTPS://PORTAL.E-POST.DE](https://portal.e-post.de)

# ASSETS

# ABANDONED SLIDES