



EXPLAINED:

MELTDOWN & SPECTRE





EXPLAINED:

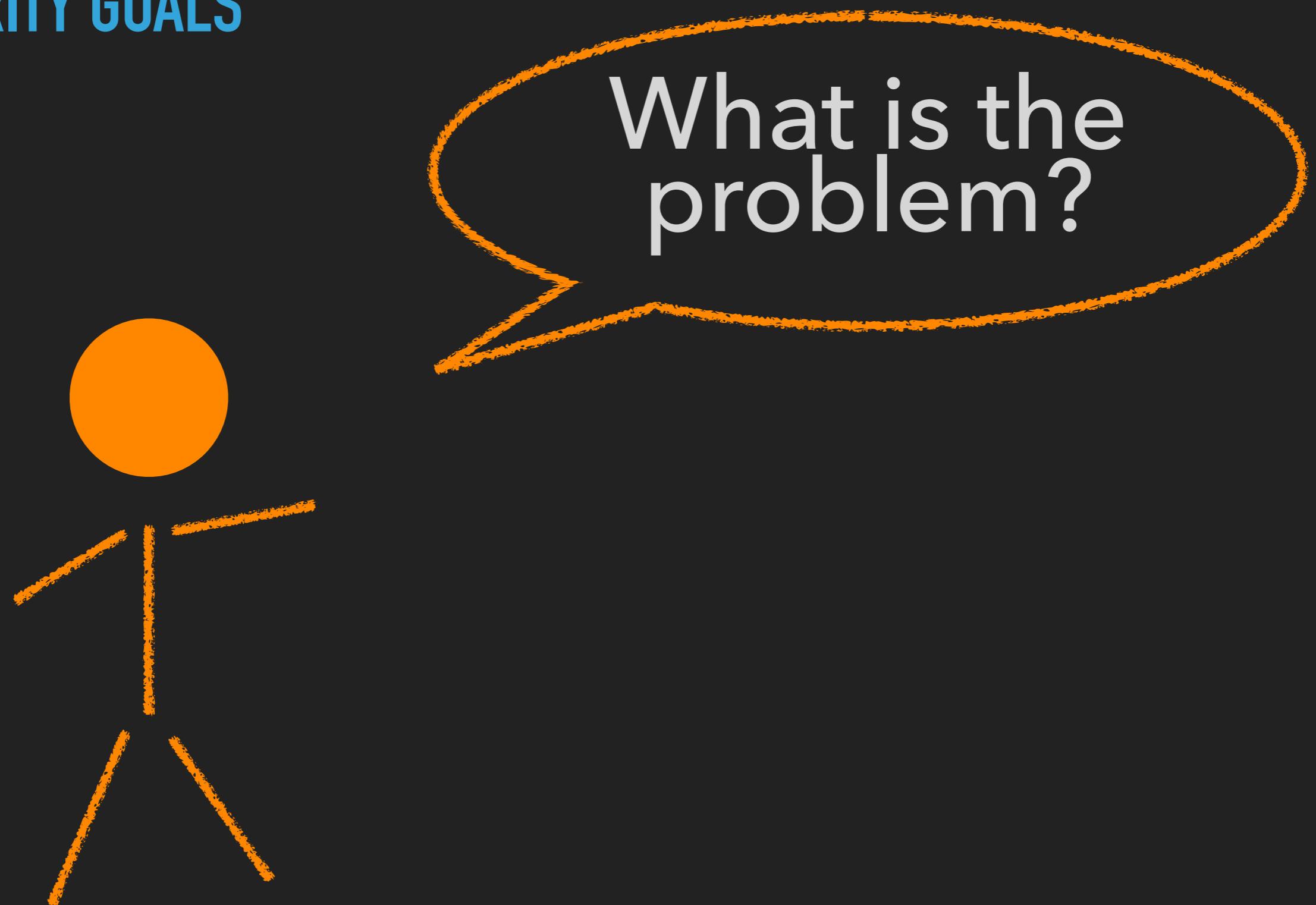
MELTDOWN & SPECTRE

for people



people
Jens Neuhalfen

SECURITY GOALS



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

You

SECURITY GOALS: APP ISOLATION

SECURITY GOALS: APP ISOLATION

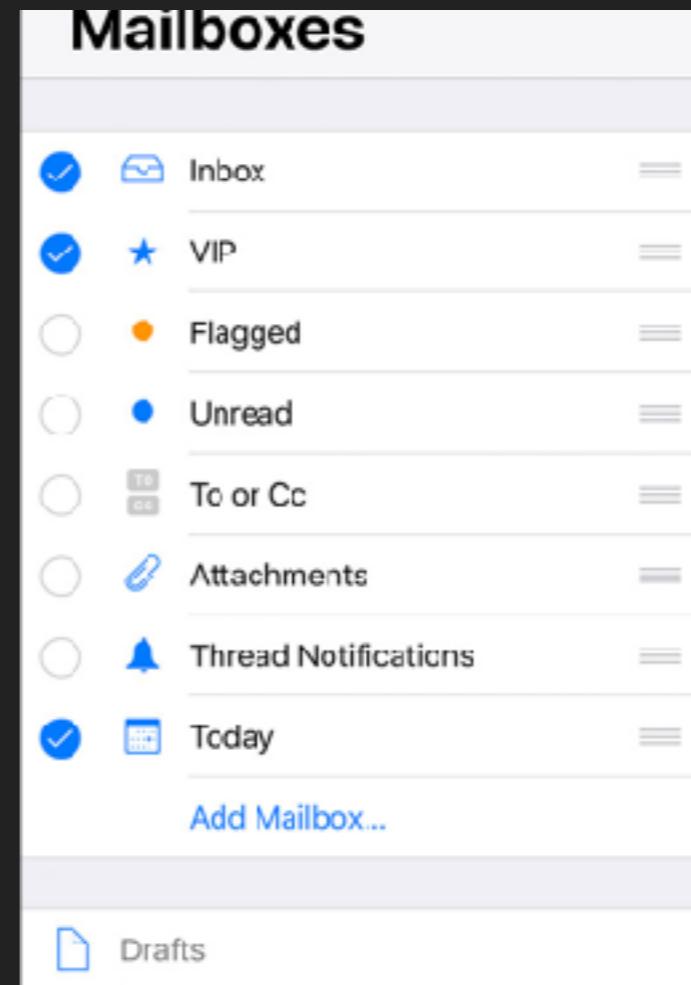


You don't want
this

SECURITY GOALS: APP ISOLATION



You don't want
this



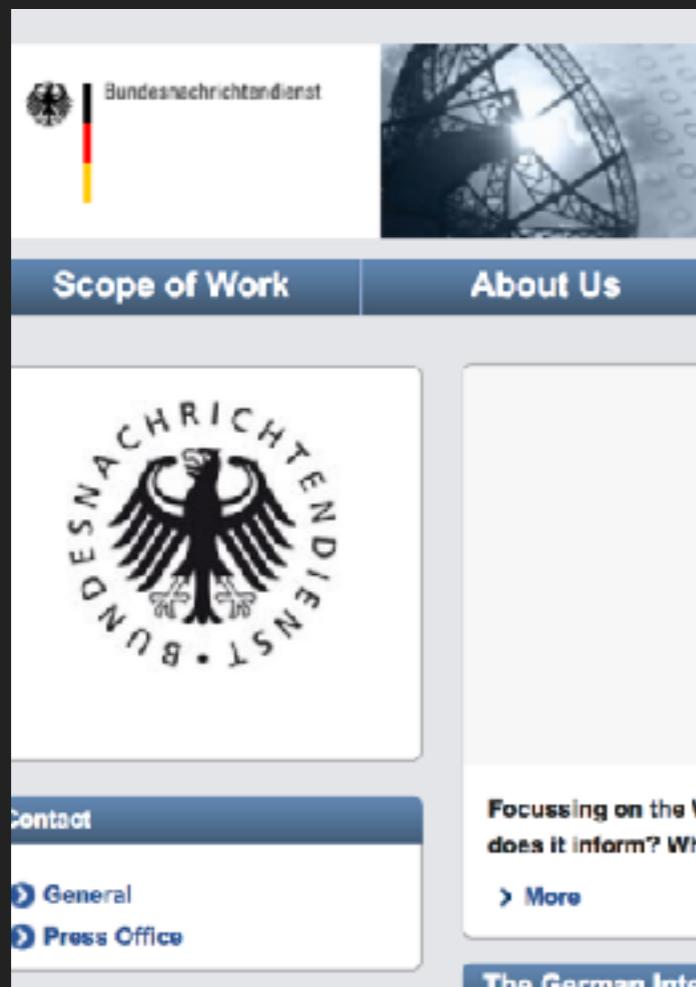
To read
that

SECURITY GOALS: BROWSER TAB ISOLATION

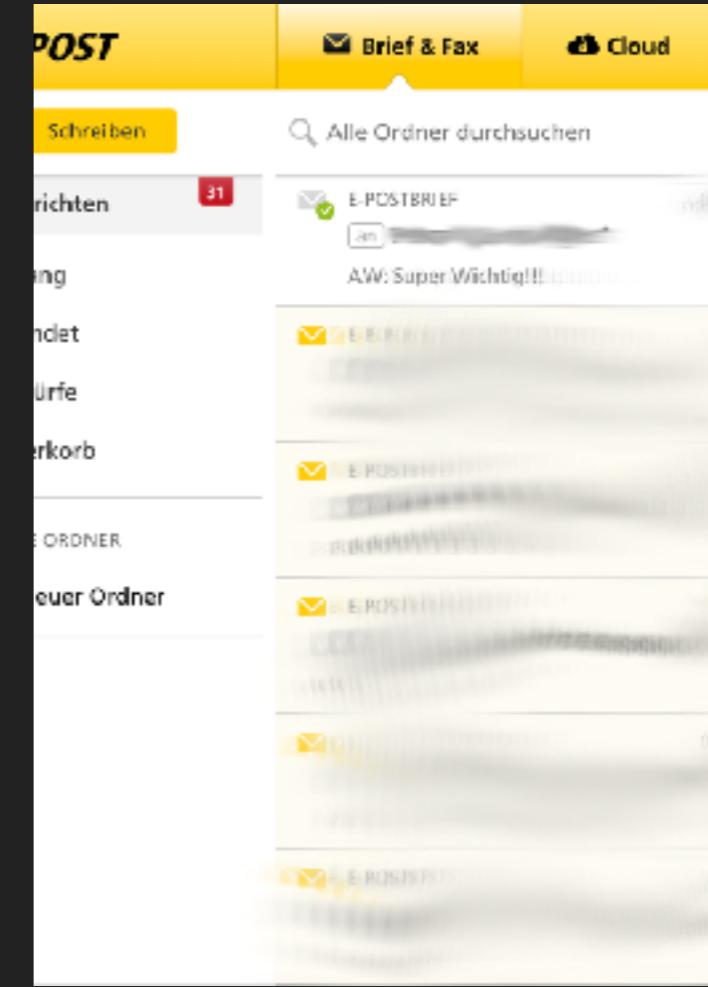


You don't want
this

SECURITY GOALS: BROWSER TAB ISOLATION



You don't want
this



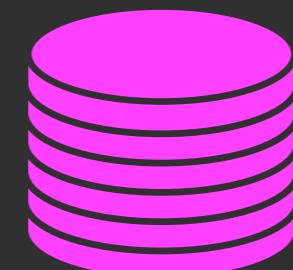
To read
that

SECURITY GOALS: CLOUD ISOLATION



You don't want
this

SECURITY GOALS: CLOUD ISOLATION



You don't want
this

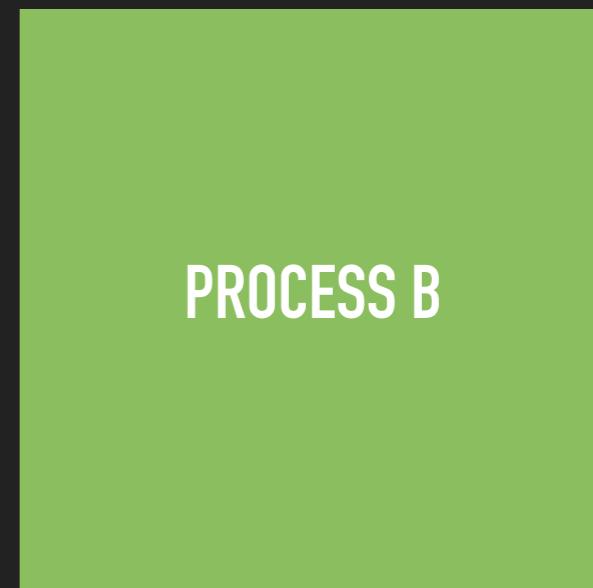
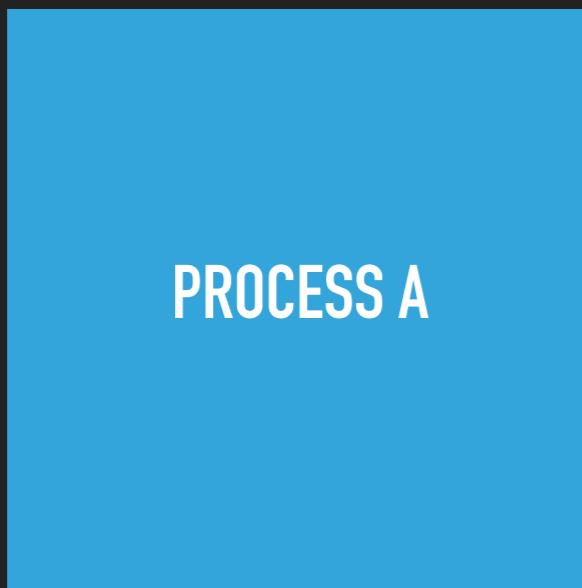
To read
that

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 diagram features a central yellow speech bubble containing the text: "The more a system relies on process isolation to achieve its security goal, the more critical Meltdown and Spectre are." To the left of the bubble is a blue rectangular box labeled "PROCESS A". To the right is a green rectangular box labeled "PROCESS B".

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

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

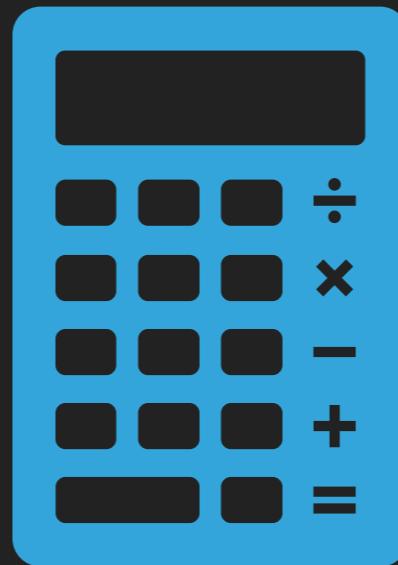
MEDIUM RISK

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



DATABASE SERVER



LOW RISK

Exploit unlikely or
running
untrusted code already
worst case

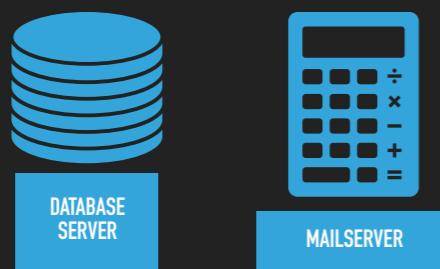
MEDIUM RISK

Exploit possible but
needs another
successful attack to run
attackers code

HIGH RISK

Exploit possible and
runs untrusted code "by
design"

THREAT-O-METER



MAILSERVER

LOW RISK

Exploit unlikely or
running
untrusted code already
worst case

MEDIUM RISK

Exploit possible
needs another
successful attack to run
attackers code

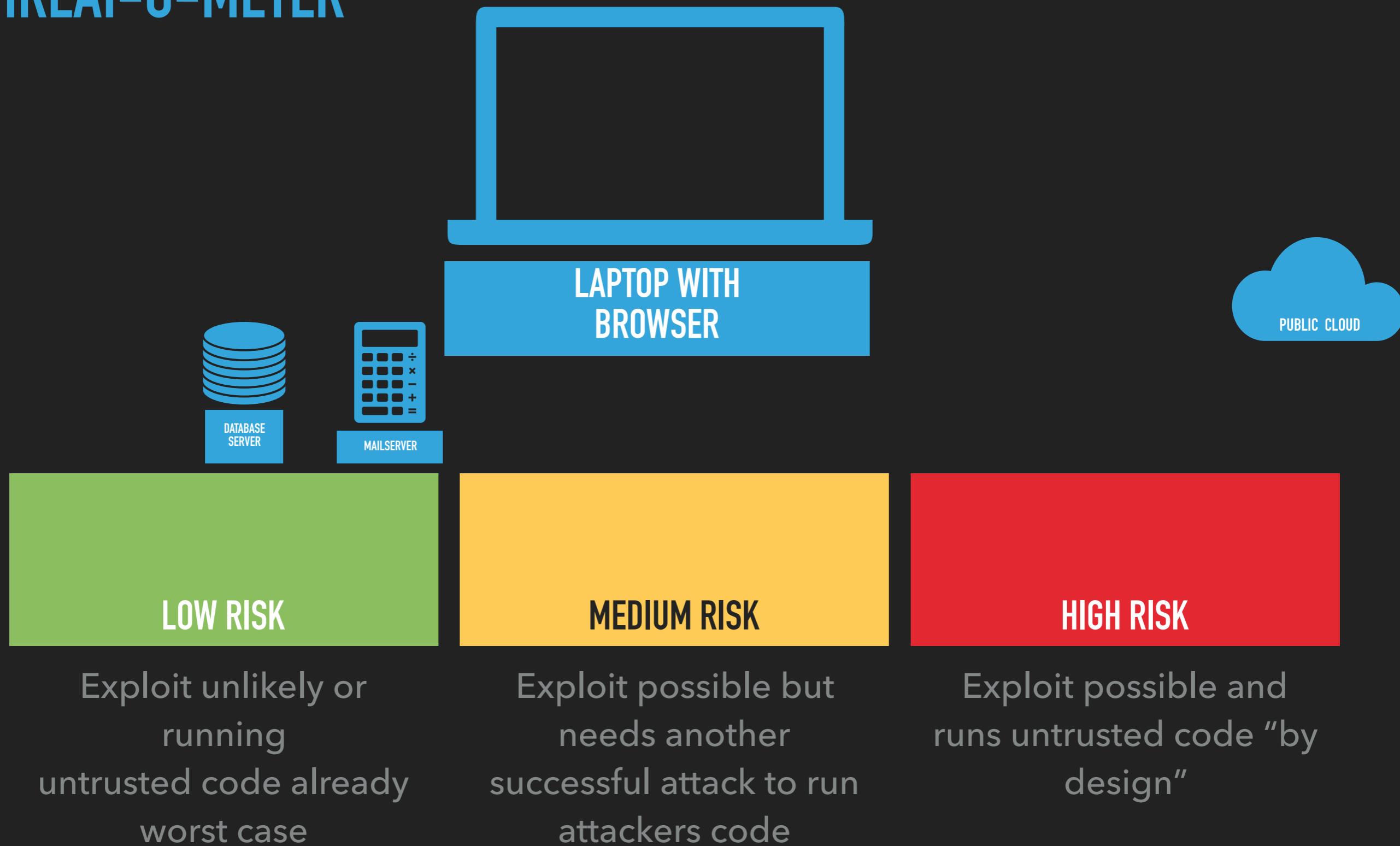
Mailserver are exposed to the internet but have been proven to be very robust to “remote code execution” attacks.

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

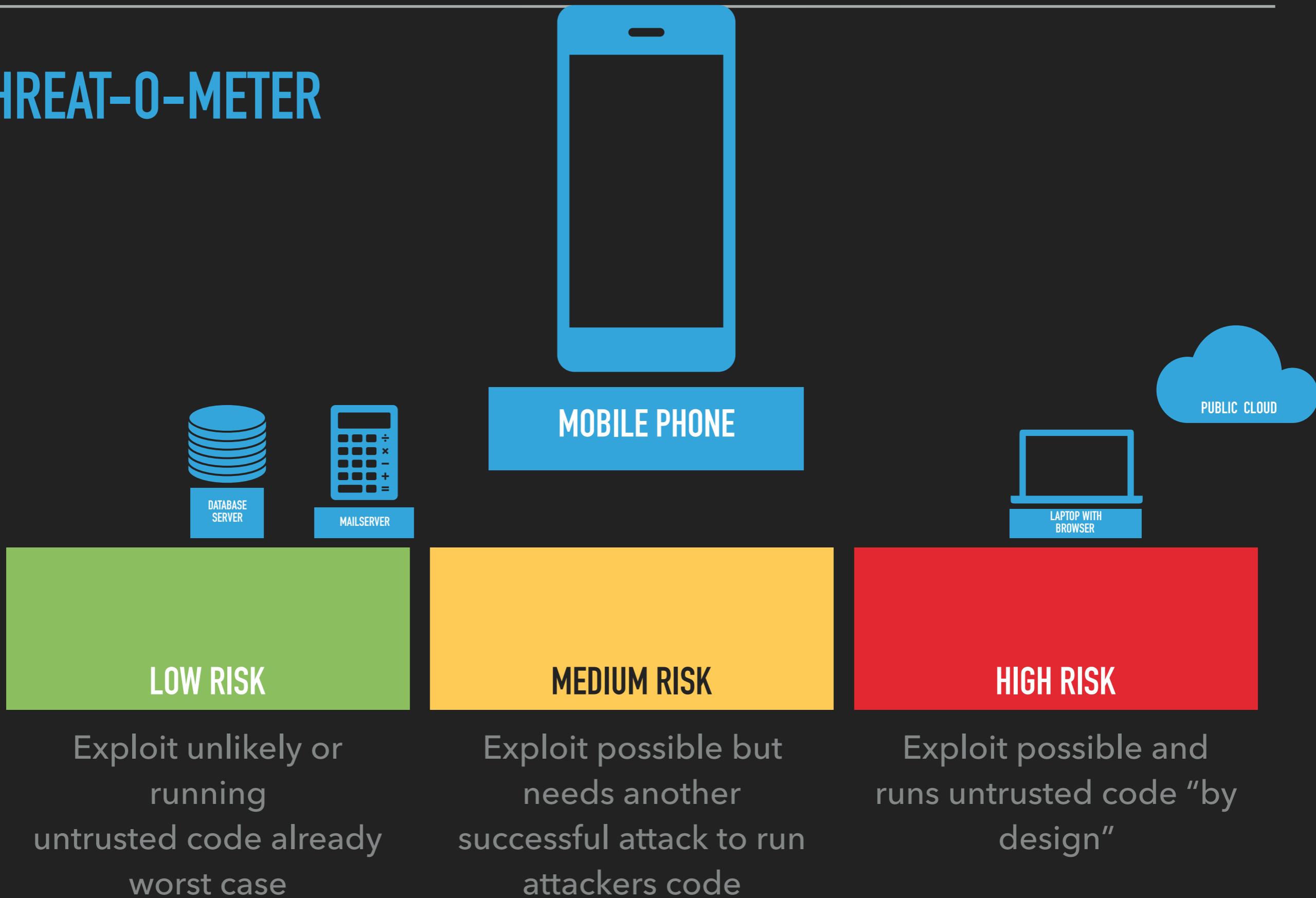
HIGH RISK

Exploit possible and
runs untrusted code "by
design"

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

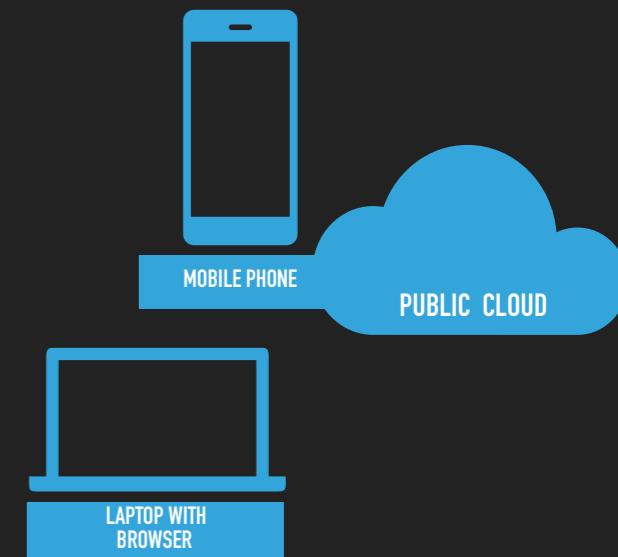


THREAT-O-METER



THREAT-O-METER

Mobile phones run apps and websites (JavaScript).



LOW RISK

Exploit unlikely or running untrusted code already worst case

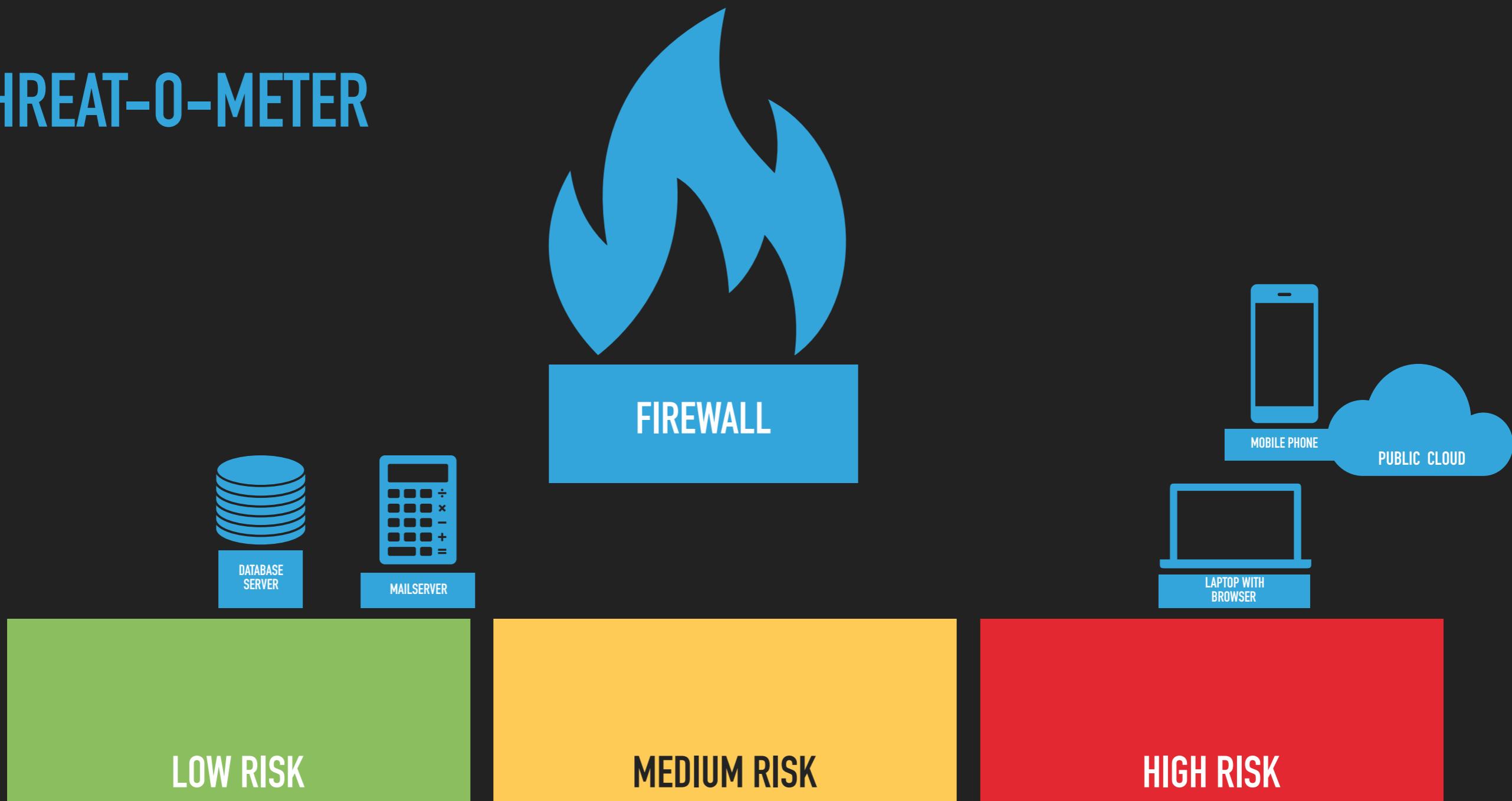
MEDIUM RISK

Exploit possible but needs another successful attack to run attackers code

HIGH RISK

Exploit possible and runs untrusted code "by design"

THREAT-O-METER

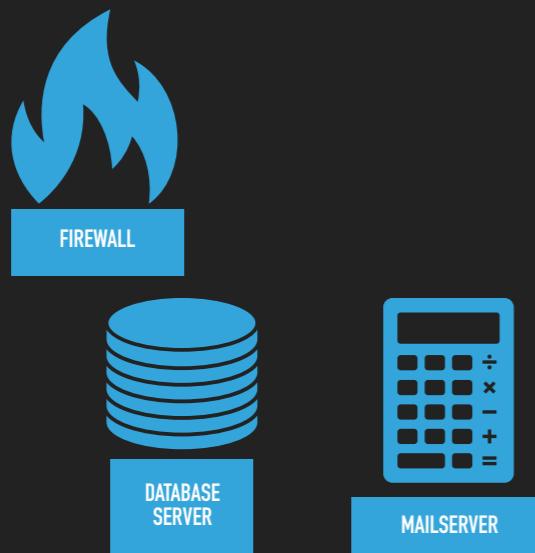


LOW RISK
Exploit unlikely or
running
untrusted code already
worst case

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

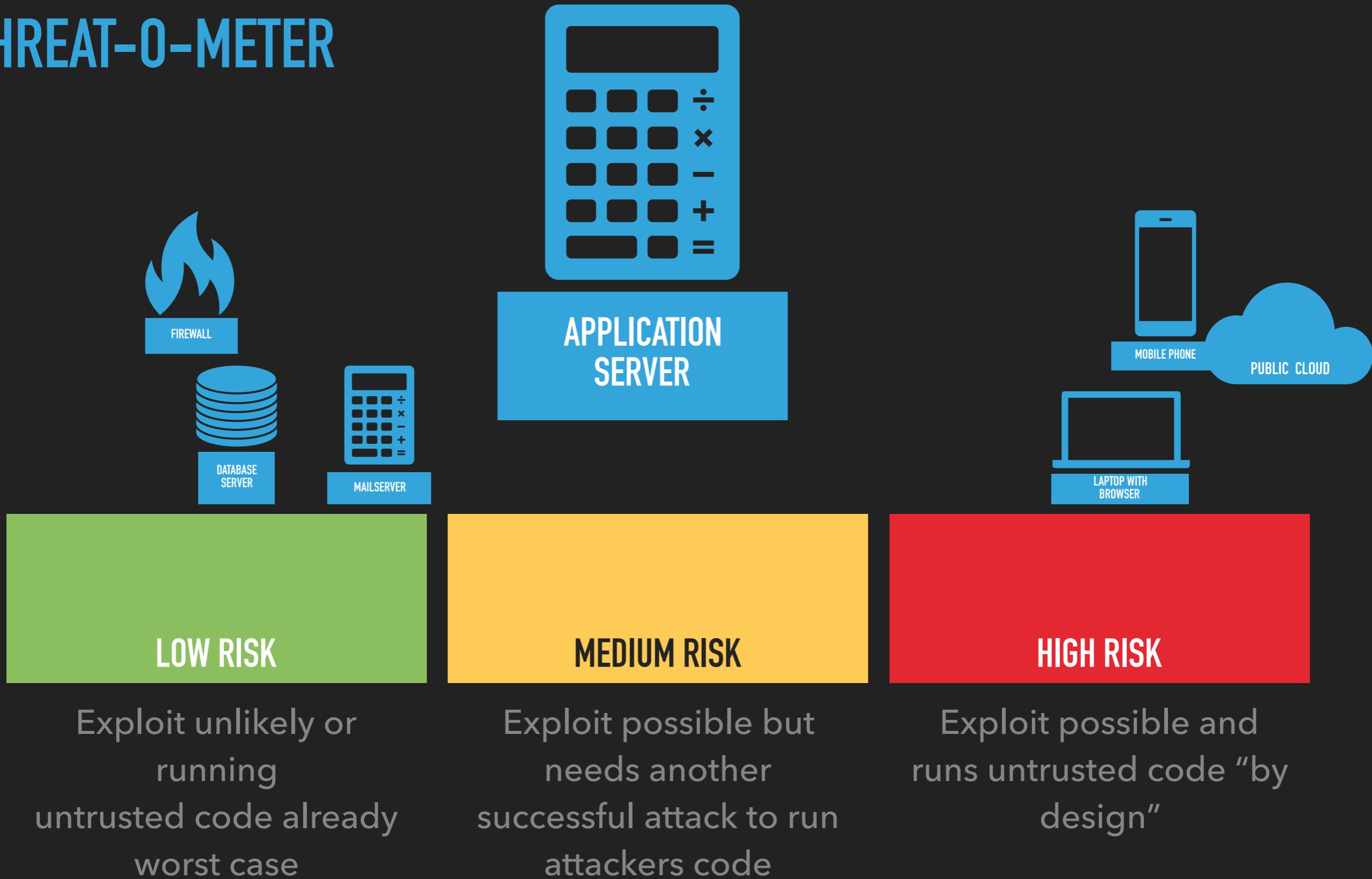
Firewalls and switches
(normally) do not expose
an attackable surface to
the external network.

This greatly reduces the
likelihood of attacks.

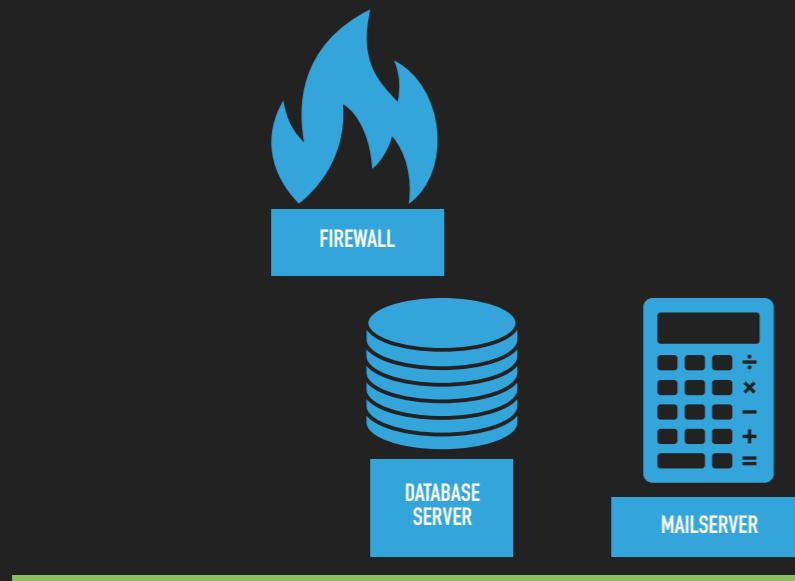
A code execution is
already the worst case.

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

THREAT-O-METER



THREAT-O-METER

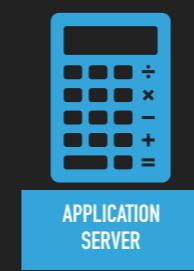


Exploit unlikely or
running
untrusted code already
worst case

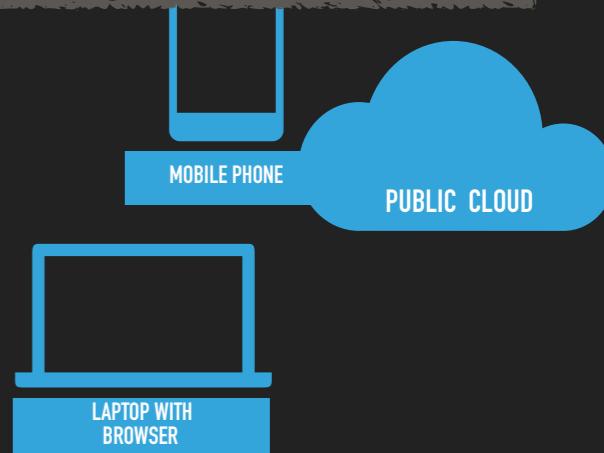


Exploit possible but
needs another
successful attack to run
attackers code

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

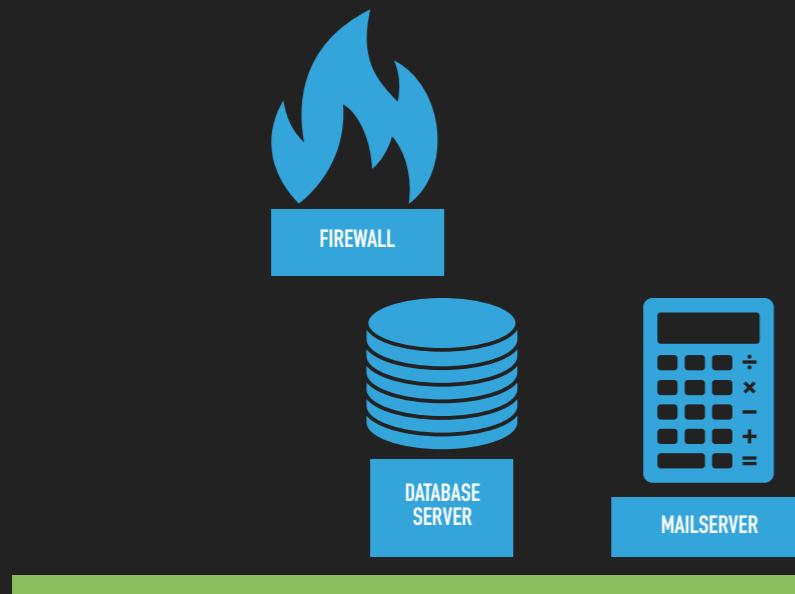
MEDIUM RISK

Exploit possible but needs another successful attack to run attackers code

HIGH RISK

Exploit possible and runs untrusted code "by design"

THREAT-O-METER



Exploit unlikely or
running
untrusted code already
worst case



Exploit possible but
needs another
successful attack to run
attackers code

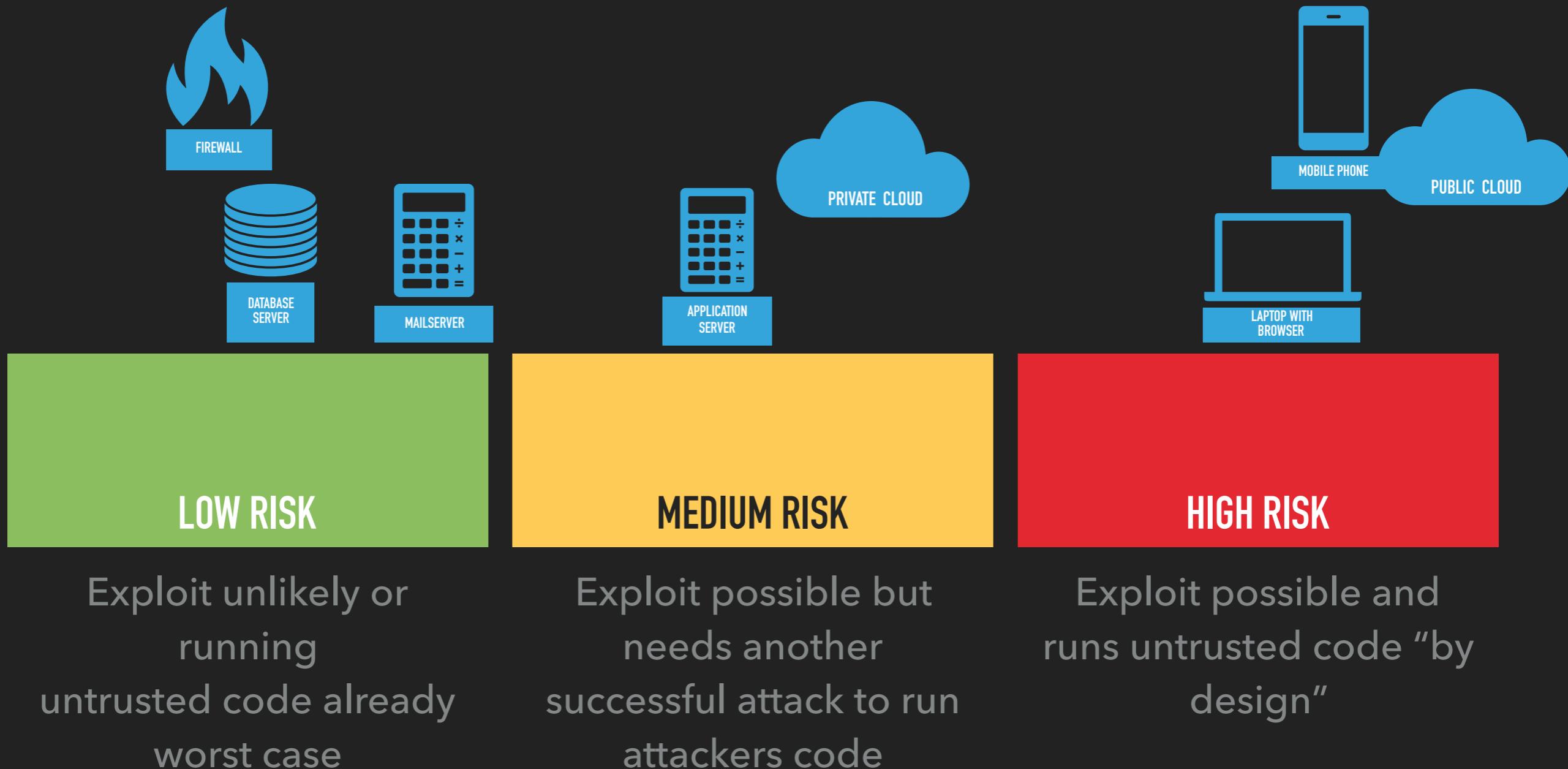


Exploit possible and
runs untrusted code "by
design"

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"

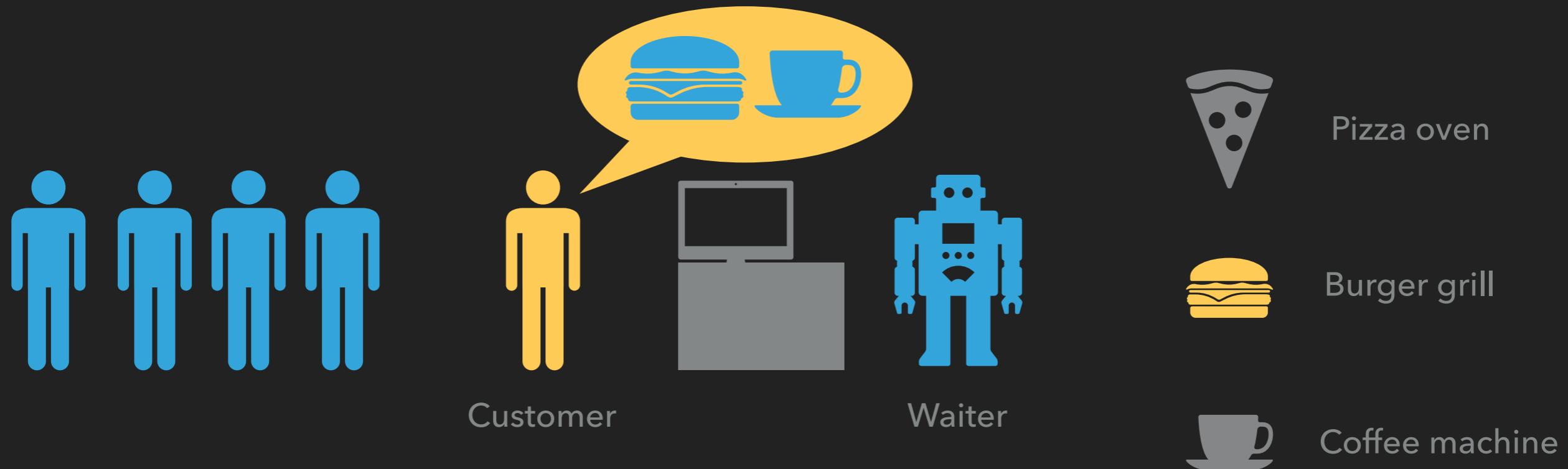


ACCIDENT, MALICE,
INCOMPETENCE?

WHY DID IT
HAPPEN?

CONFIDENTIAL BURGERS INC.

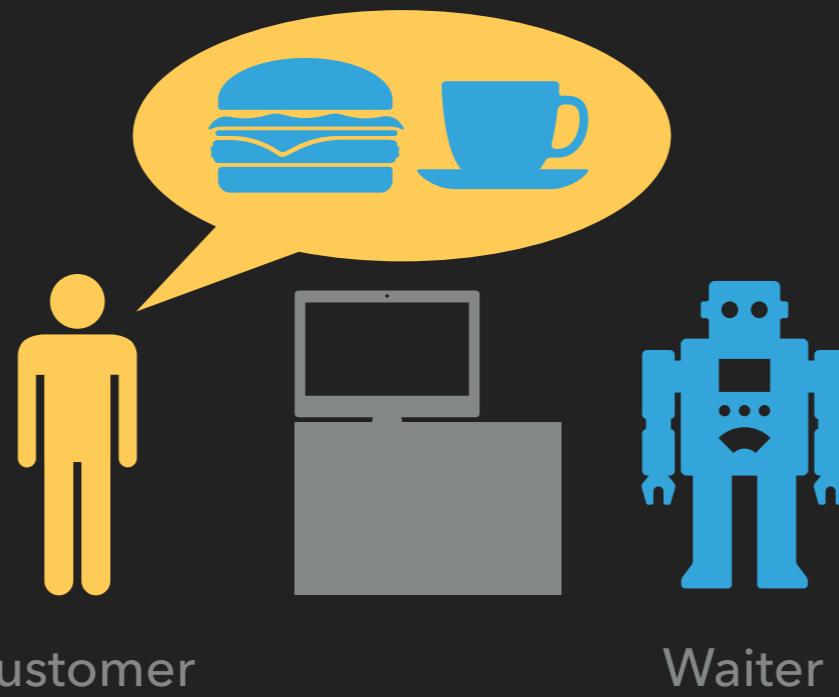
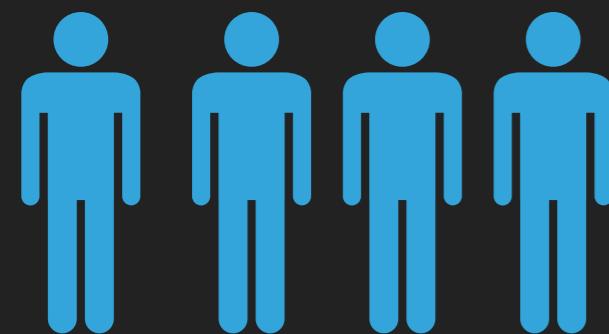
Confidential Burgers inc. sells burgers, pizza, and coffee.



Grand Opening Today

CONFIDENTIAL BURGERS INC.

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



The waiter (CPU) will

Customer

Waiter



Pizza oven



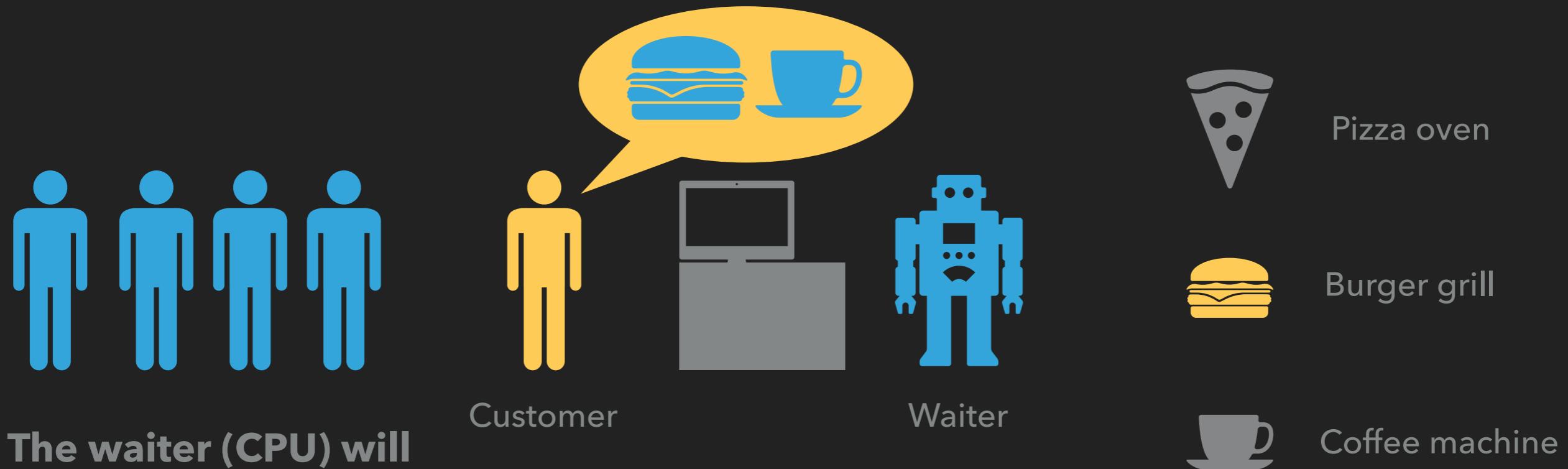
Burger grill



Coffee machine

CONFIDENTIAL BURGERS INC.

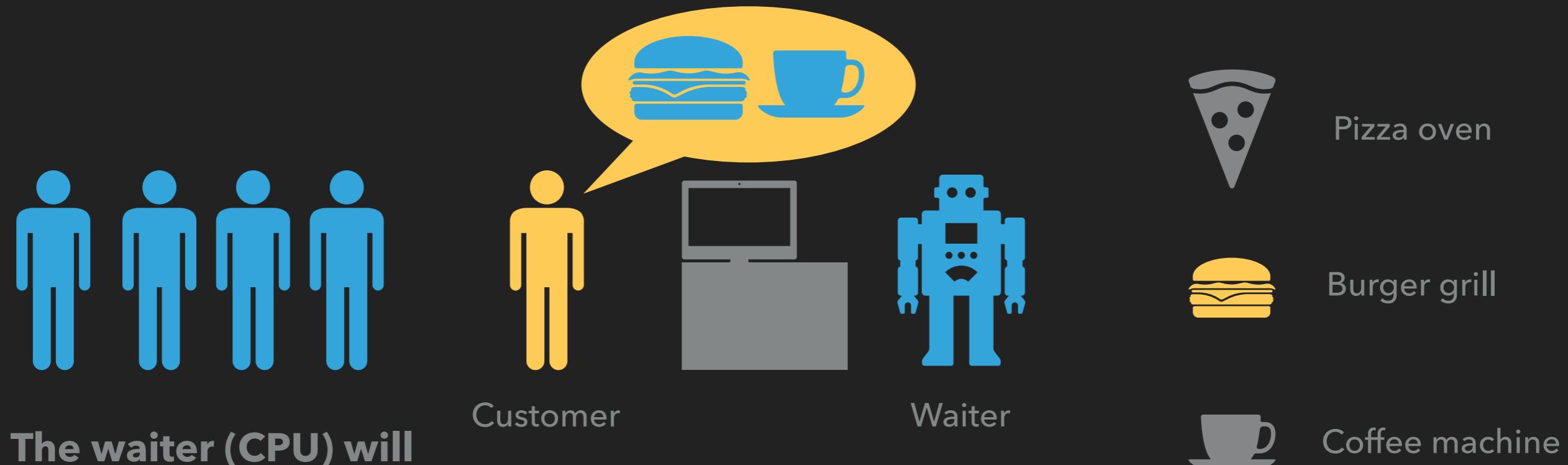
Confidential Burgers inc. sells burgers, pizza, and coffee.



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

CONFIDENTIAL BURGERS INC.

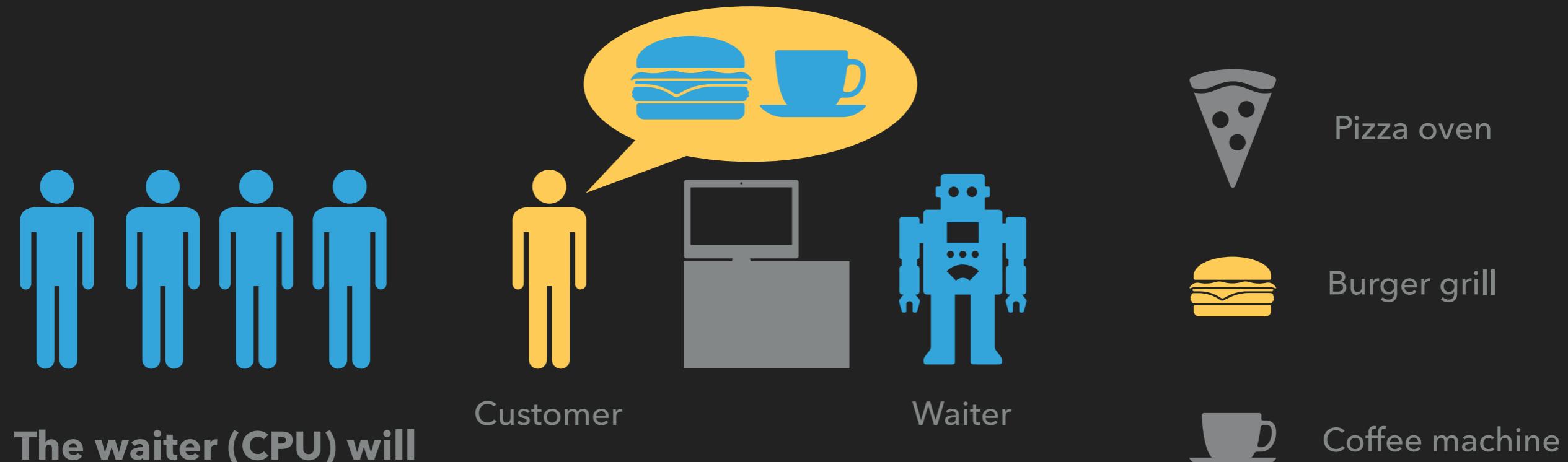
Confidential Burgers inc. sells burgers, pizza, and coffee.



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

CONFIDENTIAL BURGERS INC.

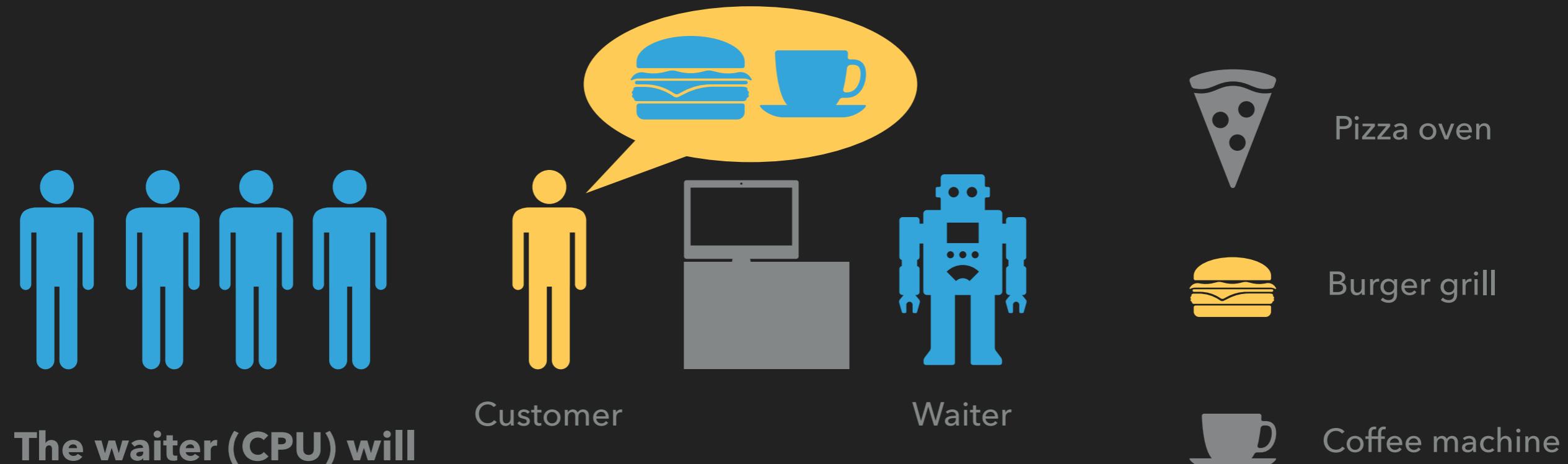
Confidential Burgers inc. sells burgers, pizza, and coffee.



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

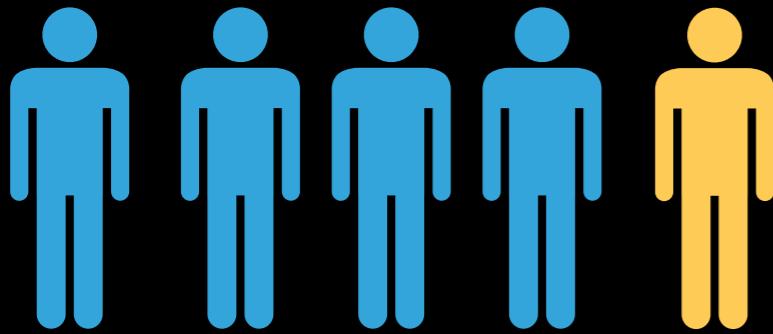
CONFIDENTIAL BURGERS INC.

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

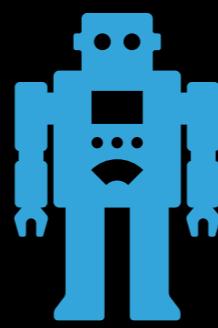


1. take an order from a customer (CPU instruction)
2. break the order (instruction) down into micro operations (μ OPs - grilling a burger, baking a pizza, ...)
3. schedule and execute the μ OPs
4. complete the order (retire the instruction)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

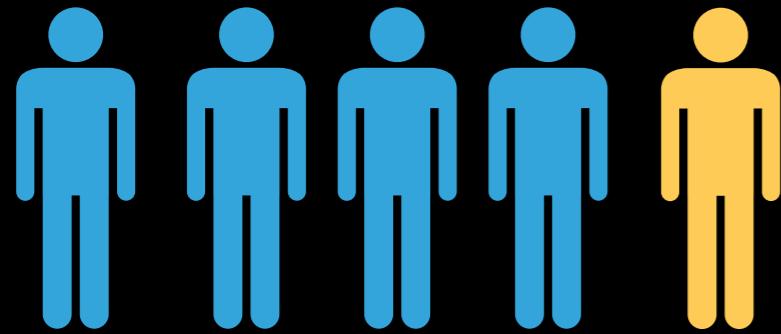


Burger grill

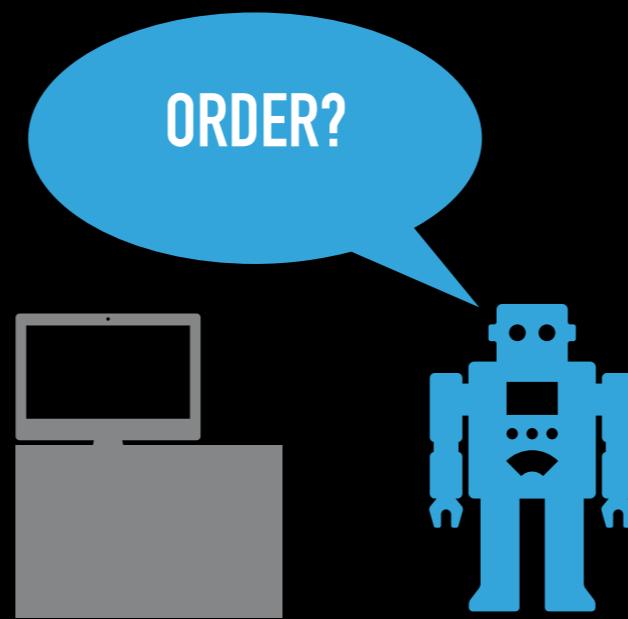


Coffee machine

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

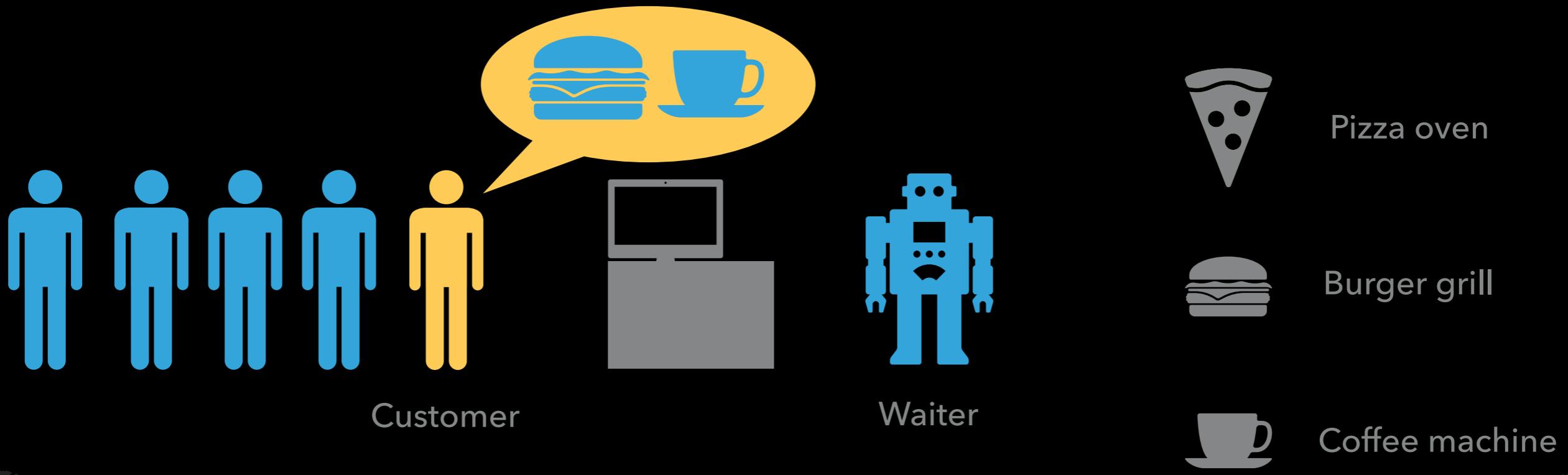


Burger grill



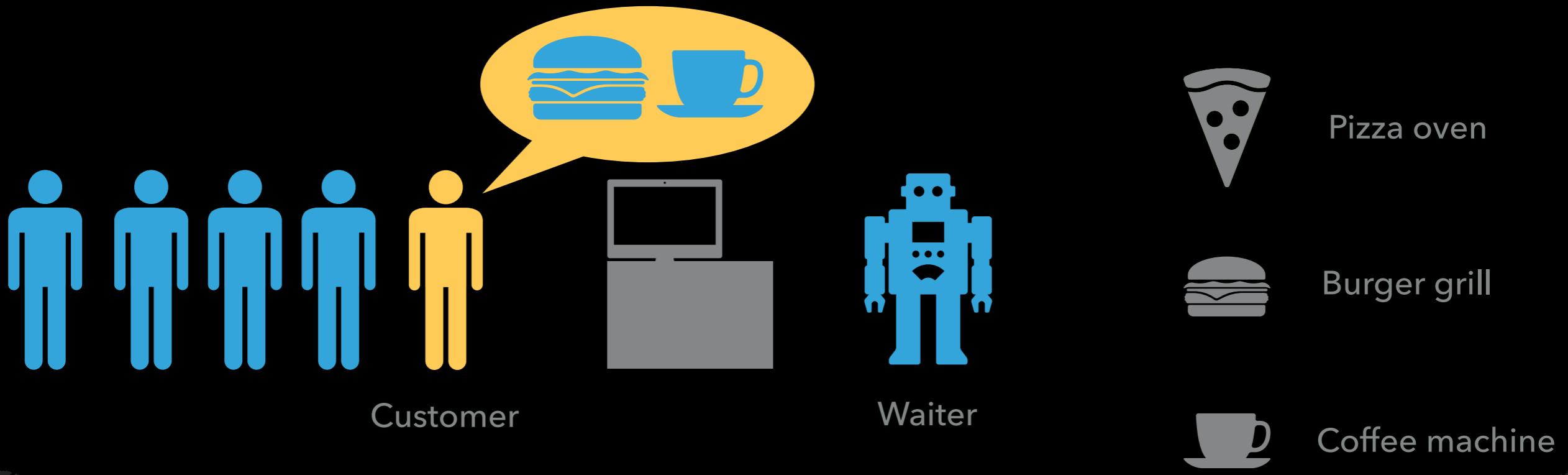
Coffee machine

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



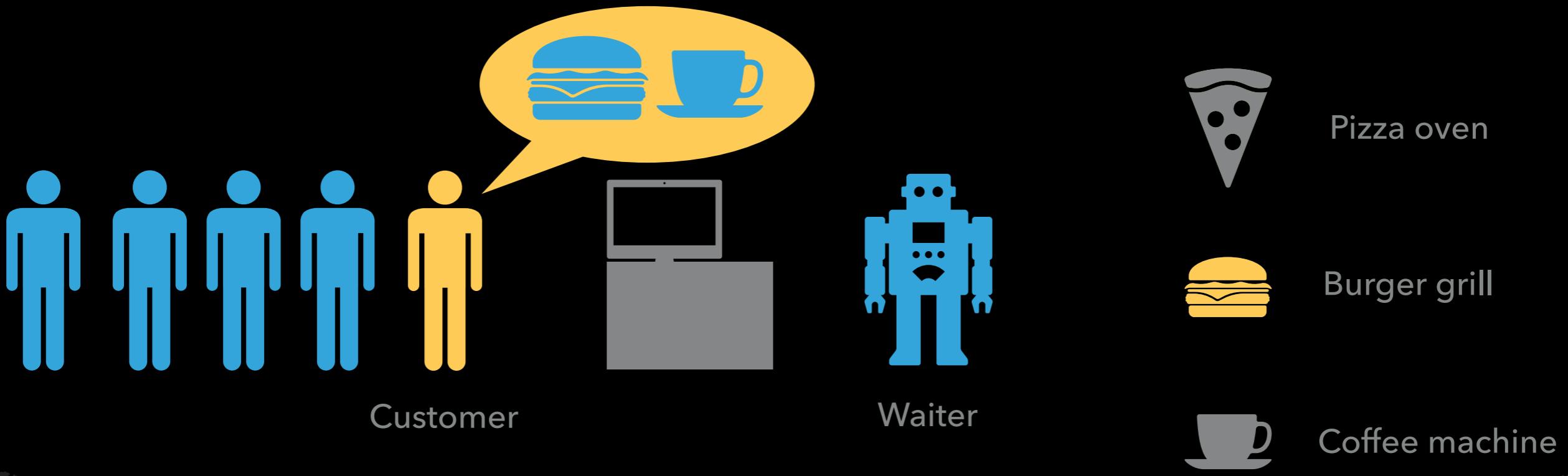
- ▶ Decode instruction into µOPs ("Burger", "Coffee")

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



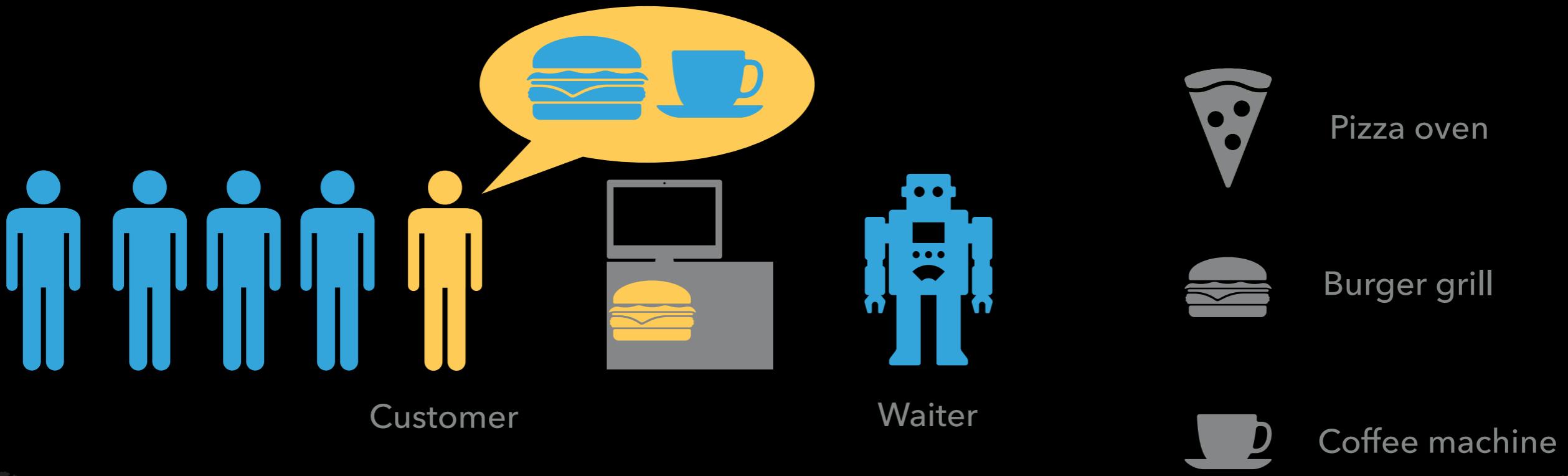
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



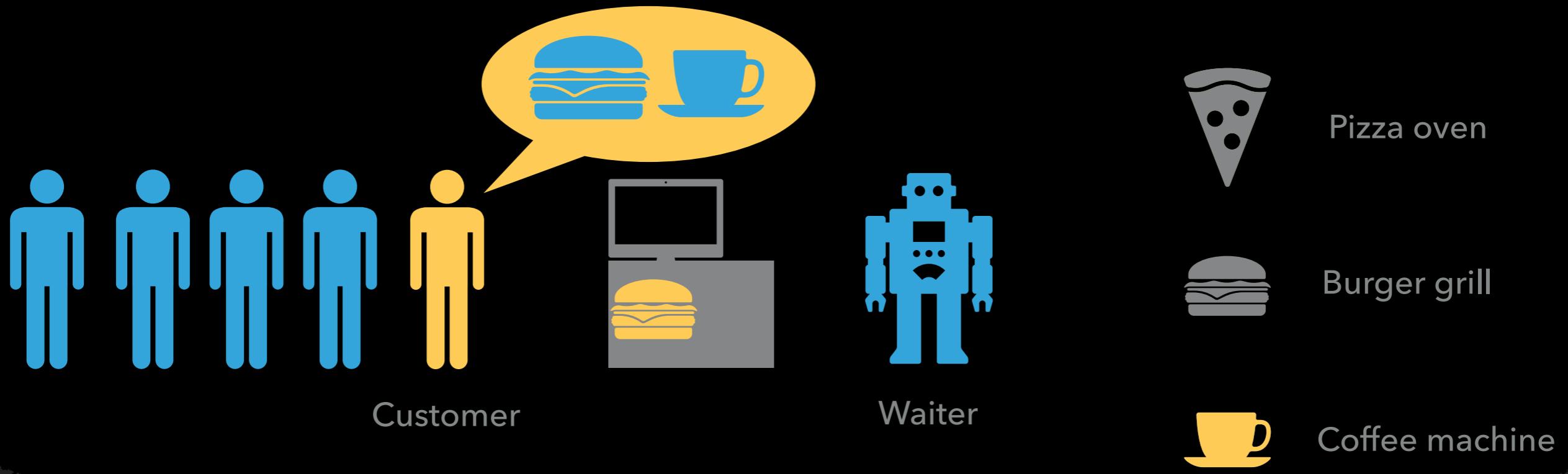
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
- ▶ run 1st µOP (grill the burger)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



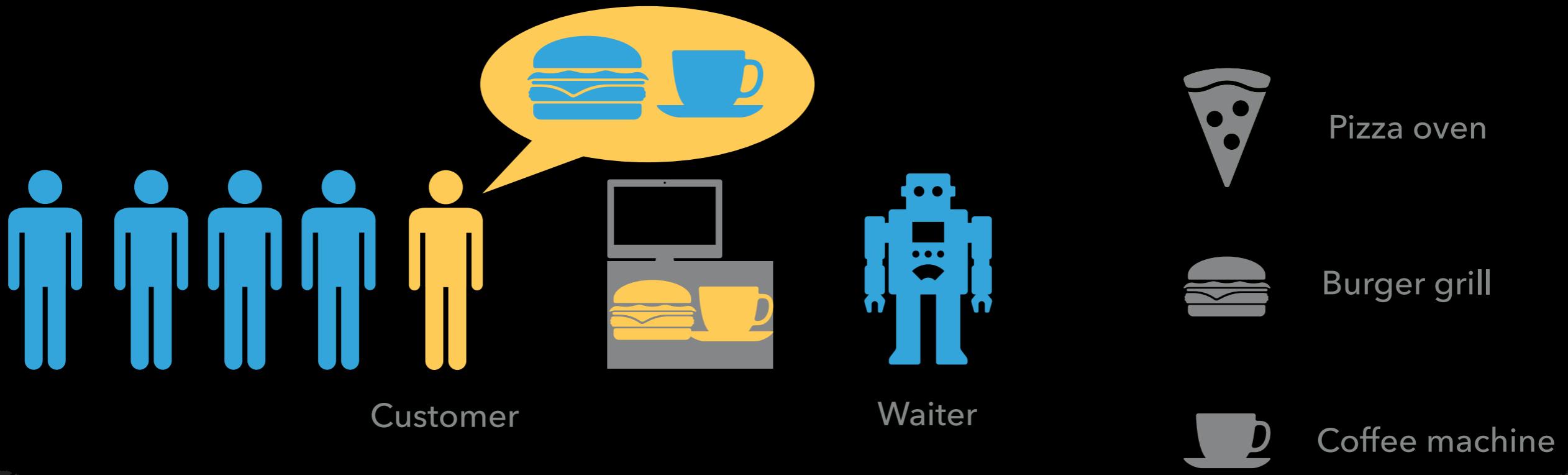
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
- ▶ run 1st µOP (grill the burger)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



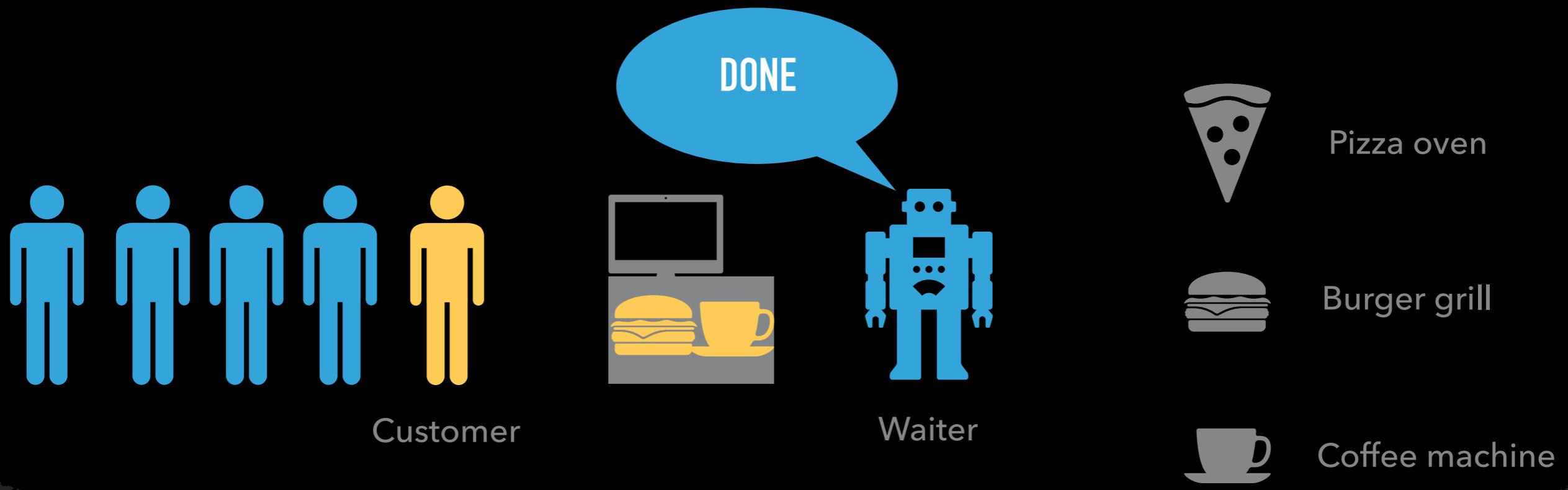
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
 - ▶ run 1st µOP (grill the burger)
 - ▶ run 2nd µOP (brew coffee, serial execution)

CONFIDENTIAL BURGERS INC. : SERIAL, IN ORDER EXECUTION



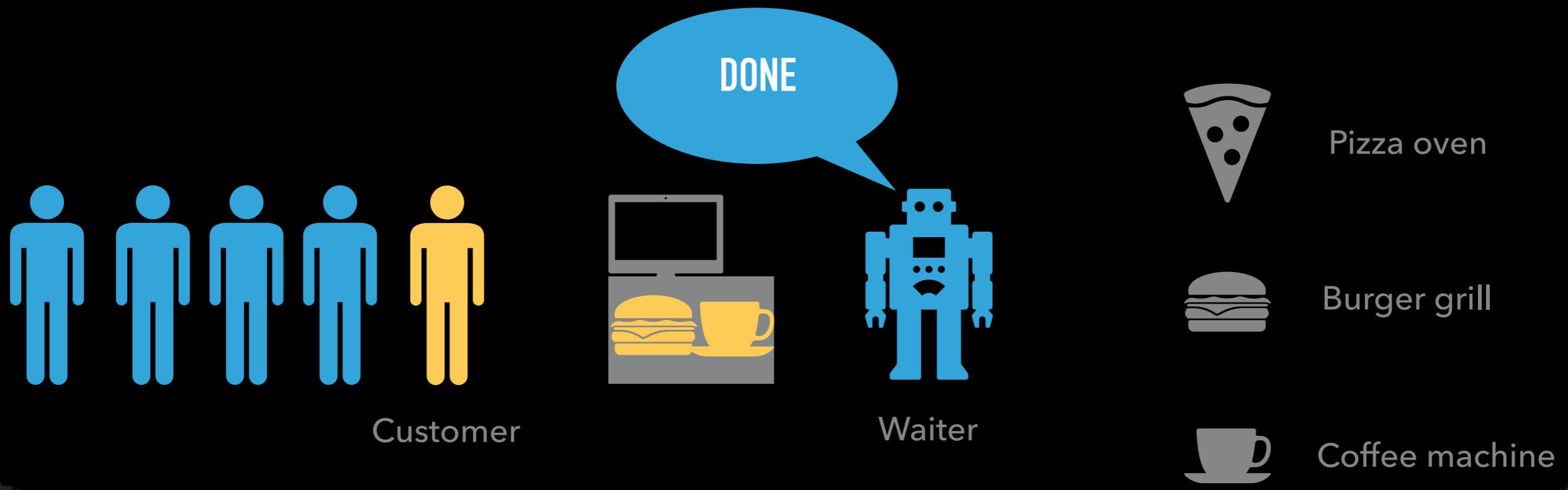
- ▶ Decode instruction into µOPs ("Burger", "Coffee")
- ▶ Schedule µOPs
 - ▶ run 1st µOP (grill the burger)
 - ▶ run 2nd µOP (brew coffee, serial execution)

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



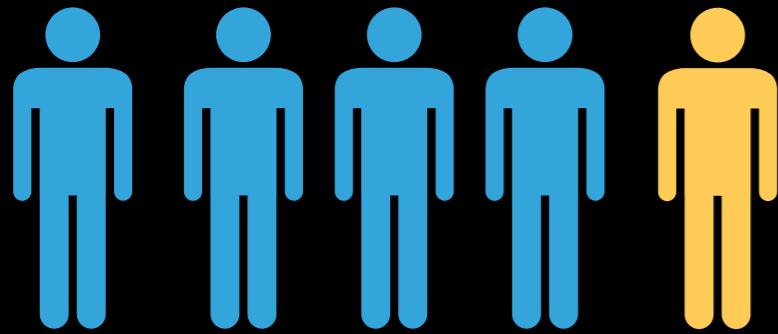
- ▶ One customer¹ after another (in order)
- ▶ Each part of the order ² executed serially

I.e. first the burger, then the coffee

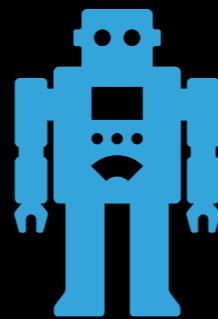
- ▶ PRO: Easy to implement and understand
- ▶ CON: Slow because resources³ not utilised fully

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

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

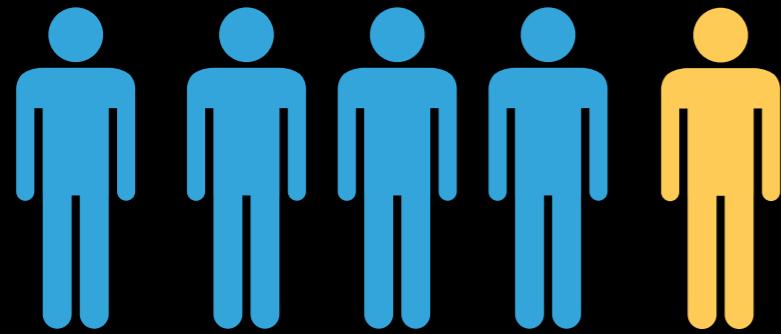


Burger grill

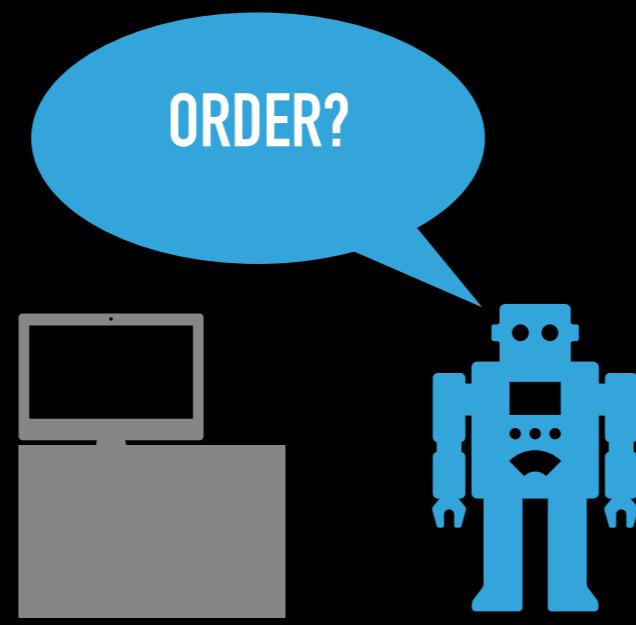


Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven

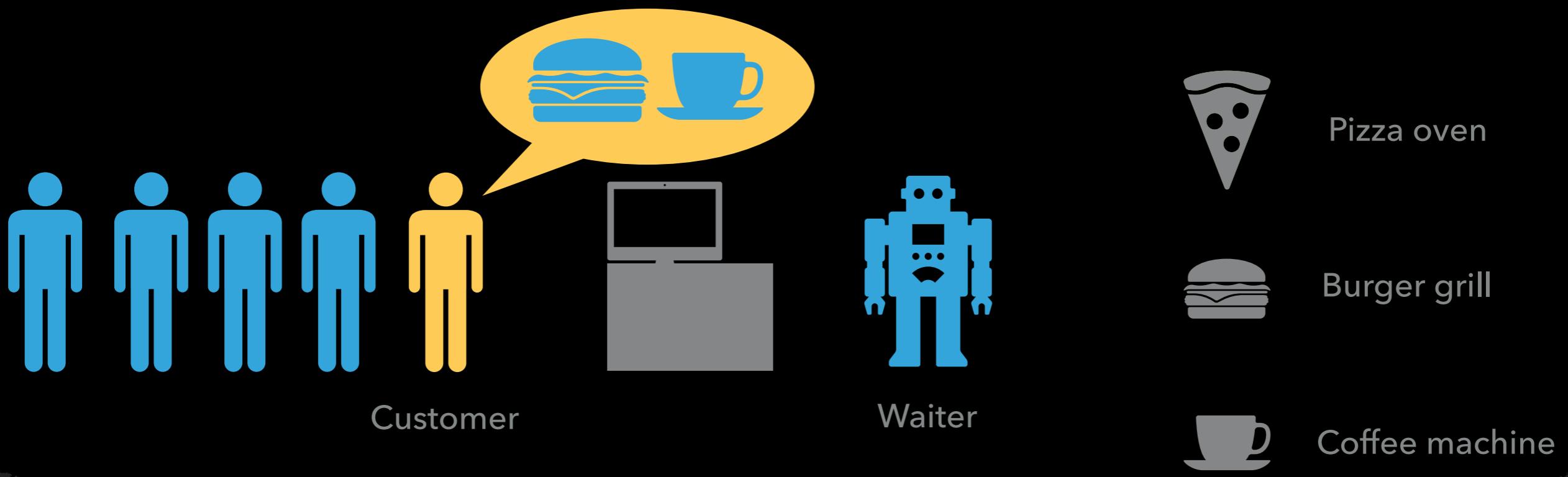


Burger grill



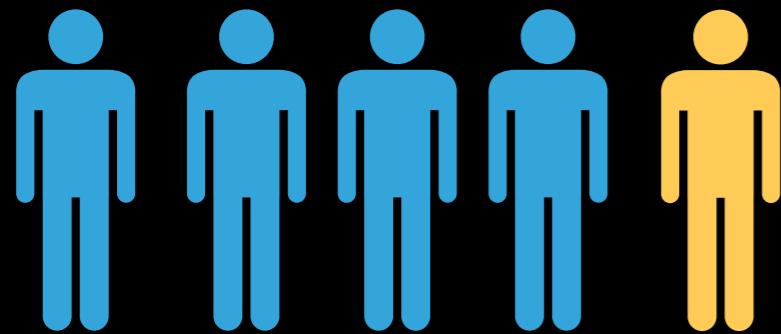
Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION

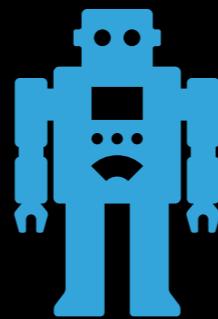


- ▶ Decode instruction into µOPs

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven



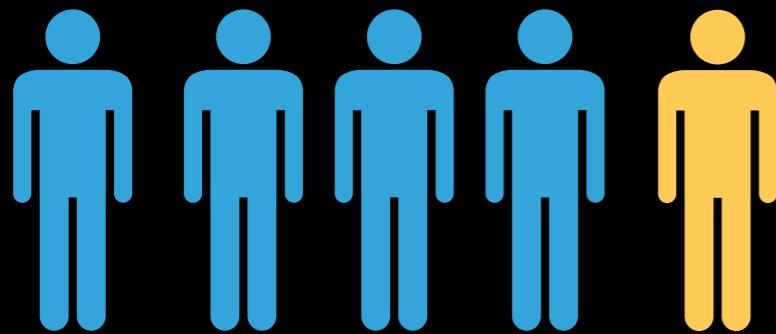
Burger grill



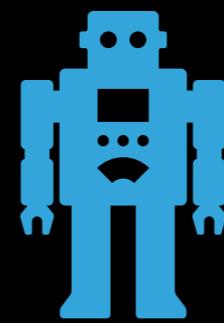
Coffee machine

- ▶ Decode instruction into µOPs
- ▶ Schedule µOPs

CONFIDENTIAL BURGERS INC. : PARALLEL, IN ORDER EXECUTION



Customer



Waiter



Pizza oven



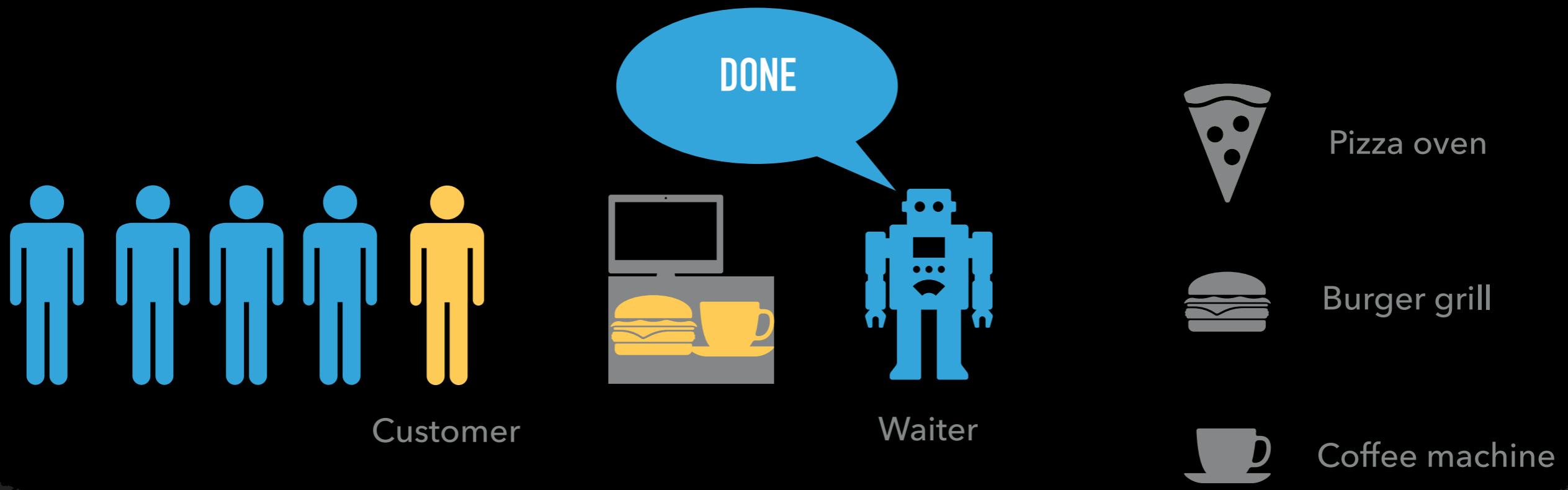
Burger grill



Coffee machine

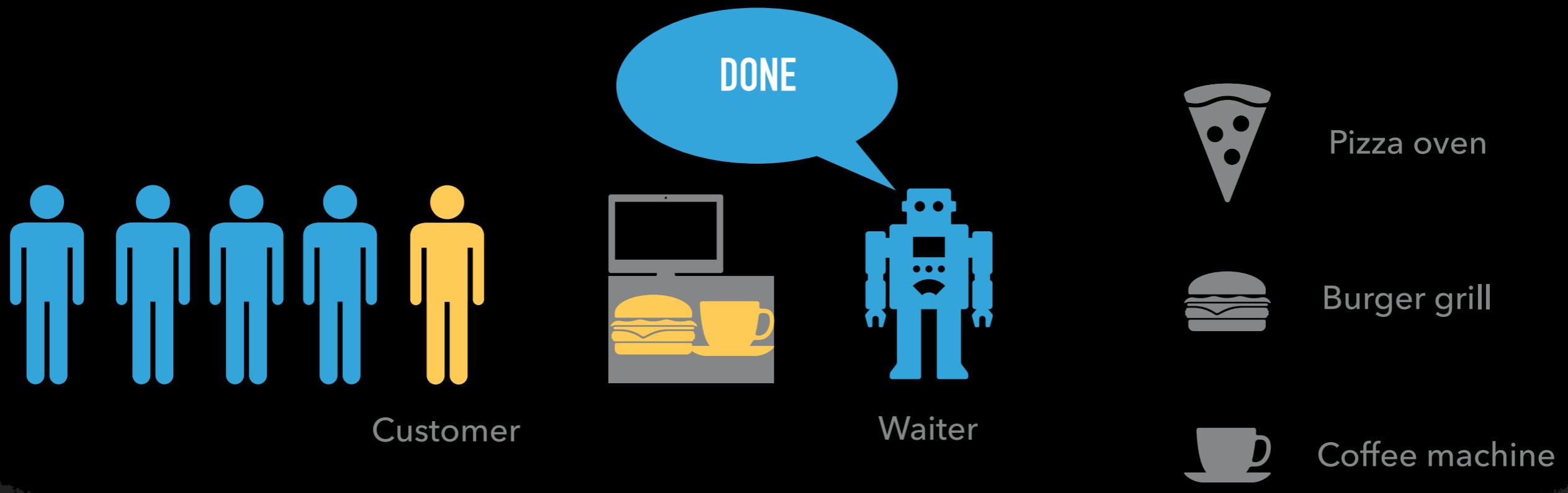
- ▶ Decode instruction into µOPs
- ▶ Schedule µOPs
- ▶ run 1st µOP and 2nd µOP (parallel execution of µOPs)

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

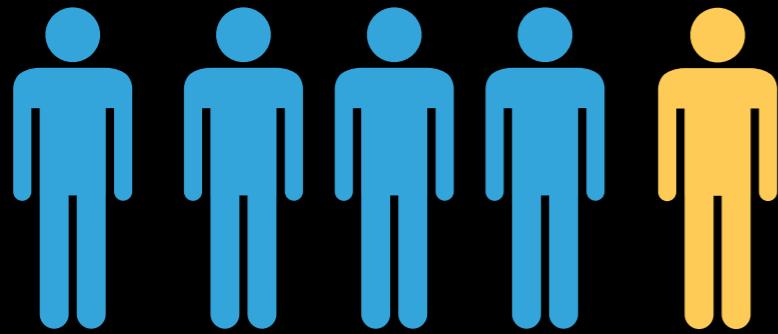


- ▶ One customer¹ after another (in order)
- ▶ Each part of the order ² executed in parallel

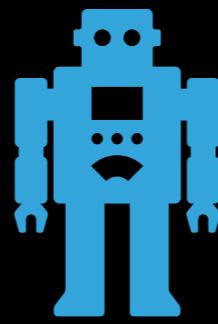
i.e. burger and coffee prepared at the same time

- ▶ 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

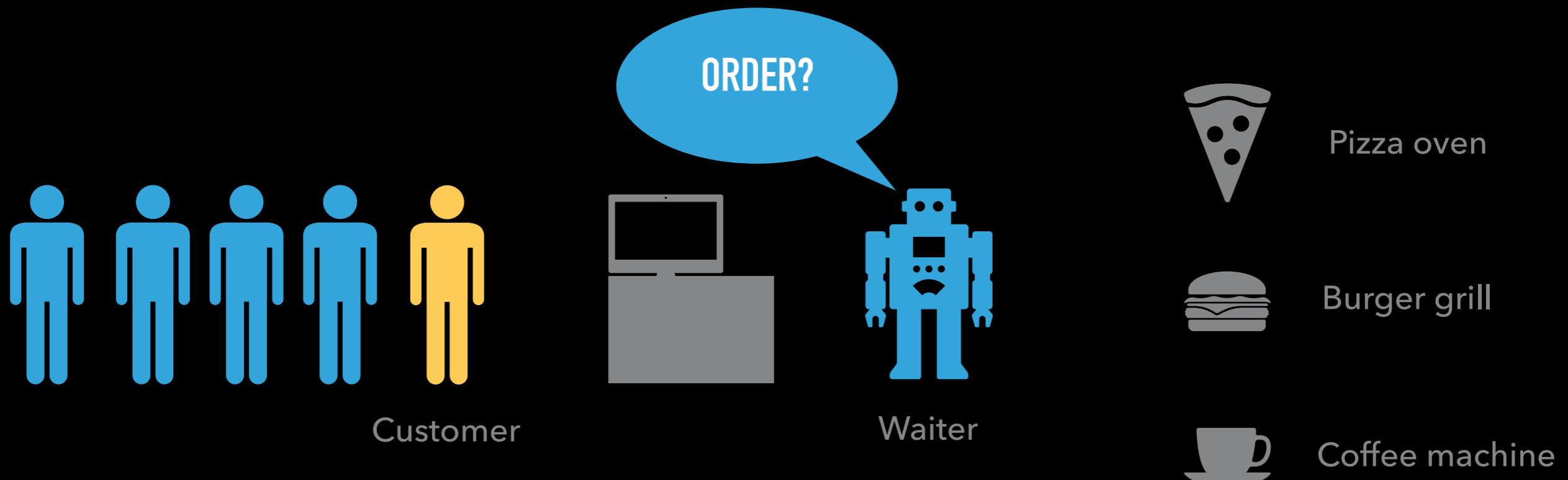


Burger grill

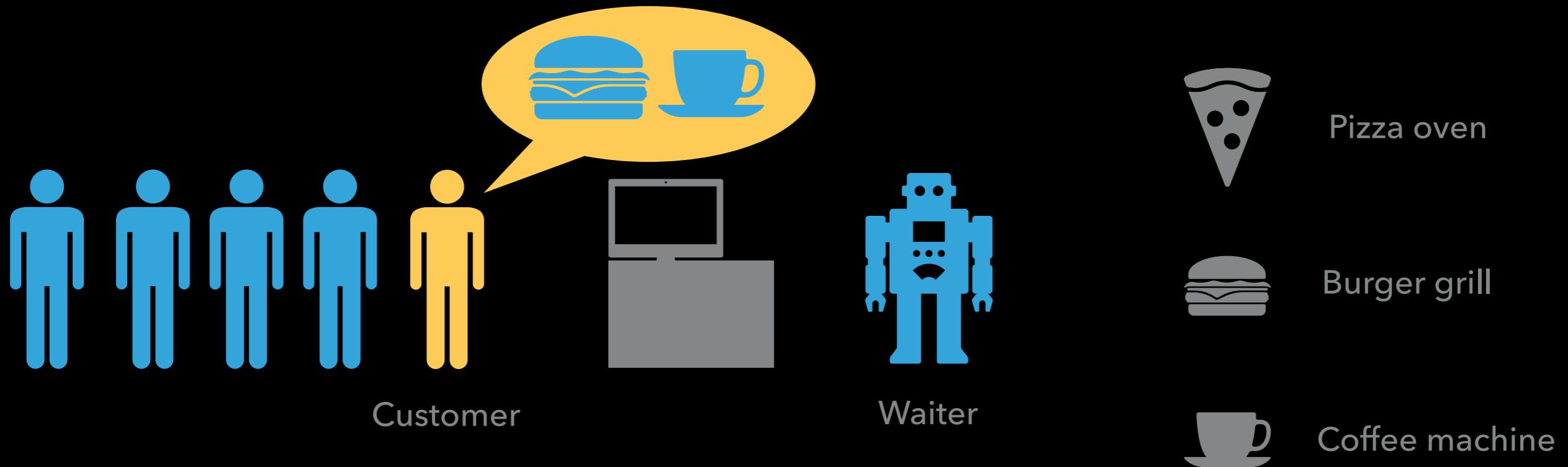


Coffee machine

CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



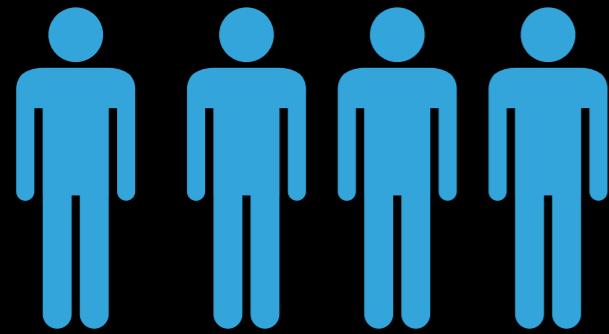
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



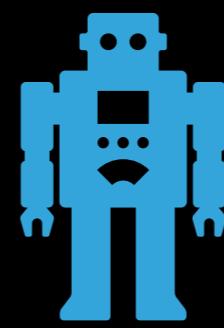
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer



Waiter



Pizza oven



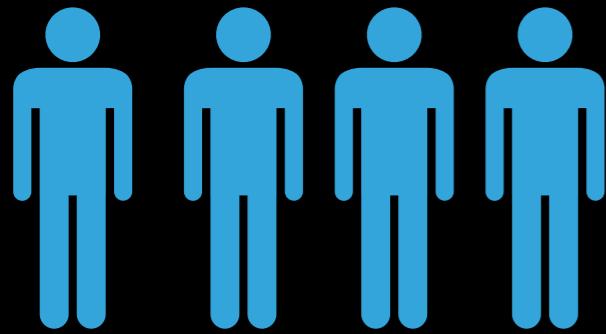
Burger grill



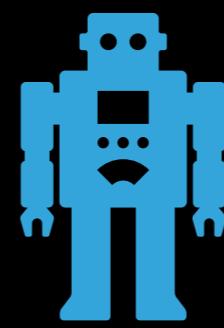
Coffee machine



CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



Customer



Waiter



Pizza oven



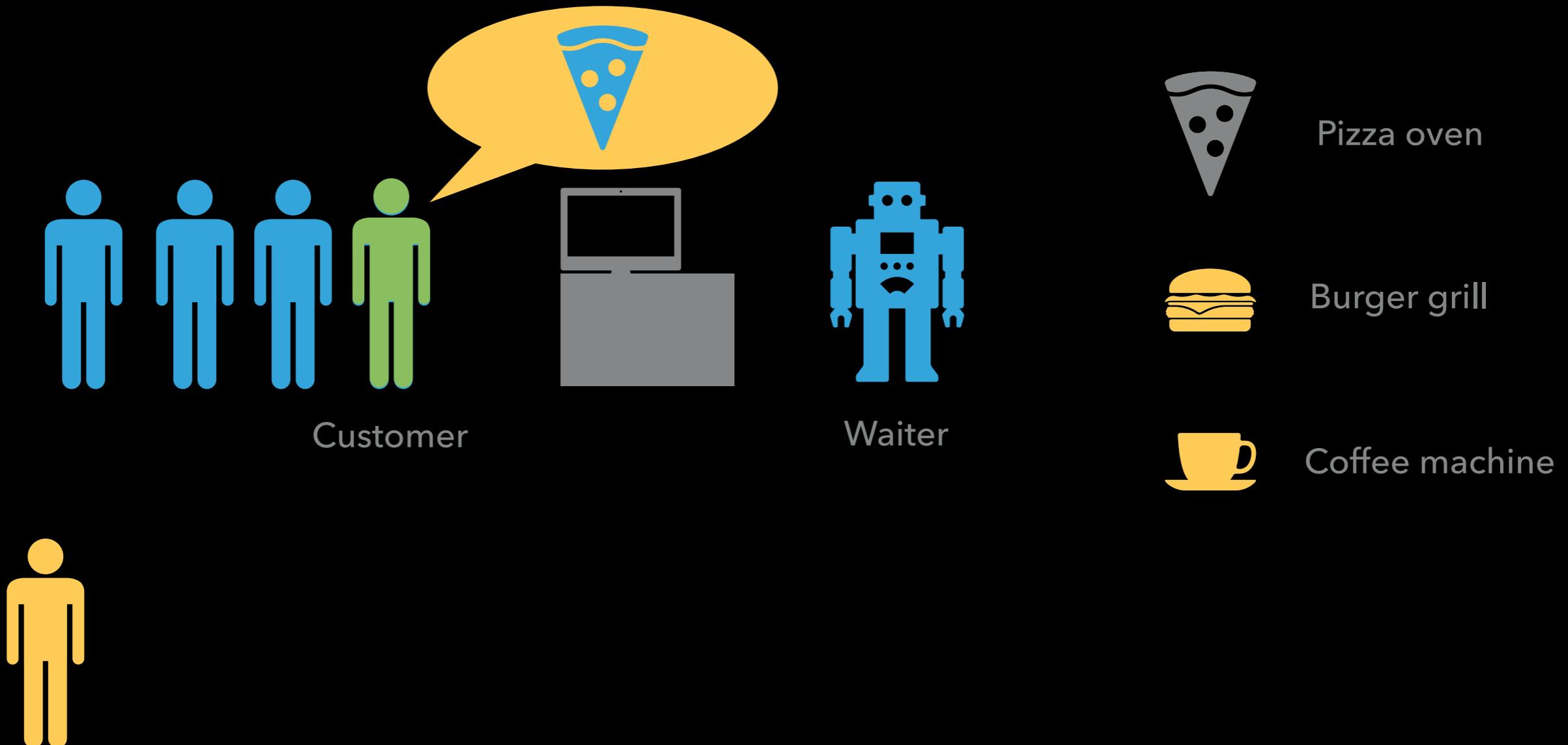
Burger grill



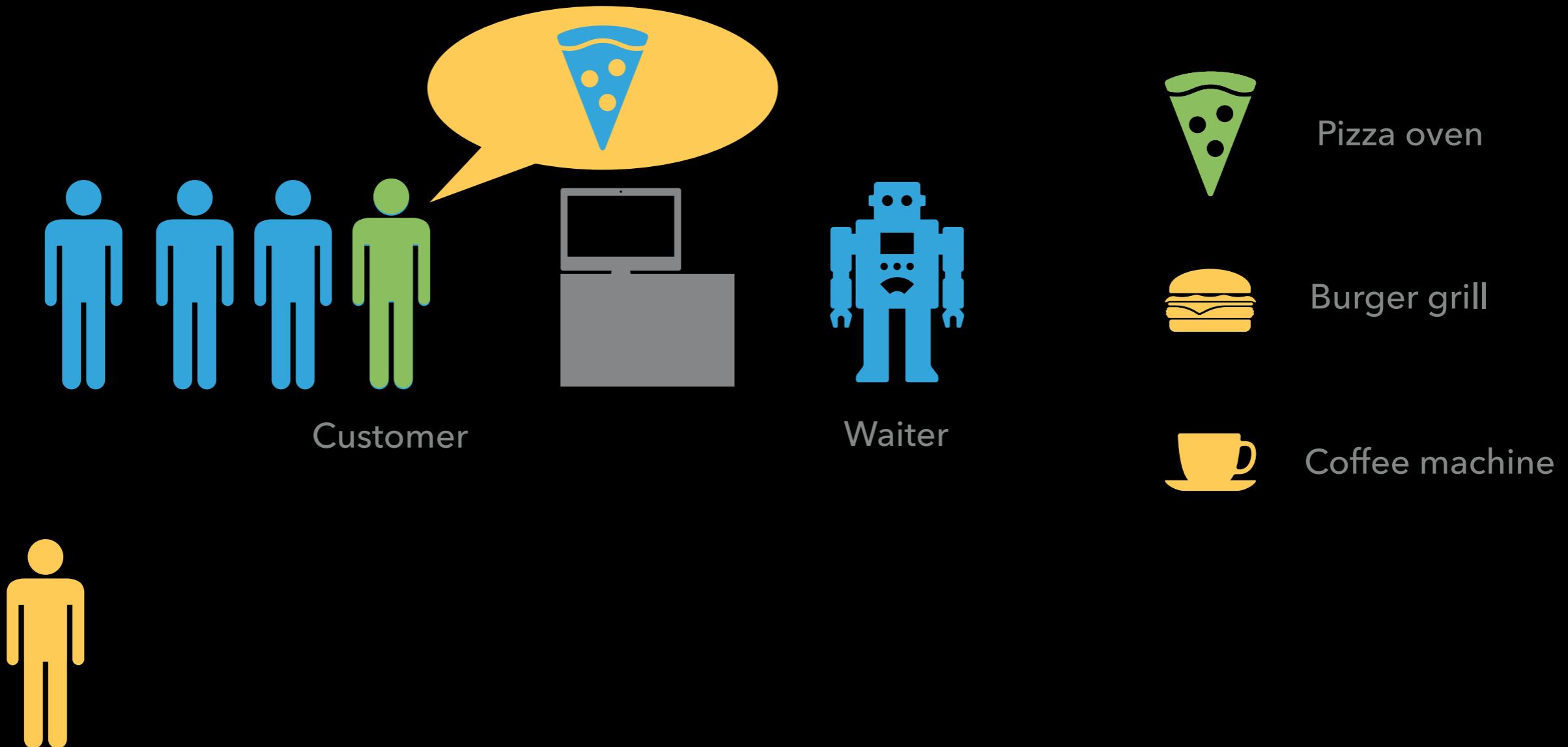
Coffee machine



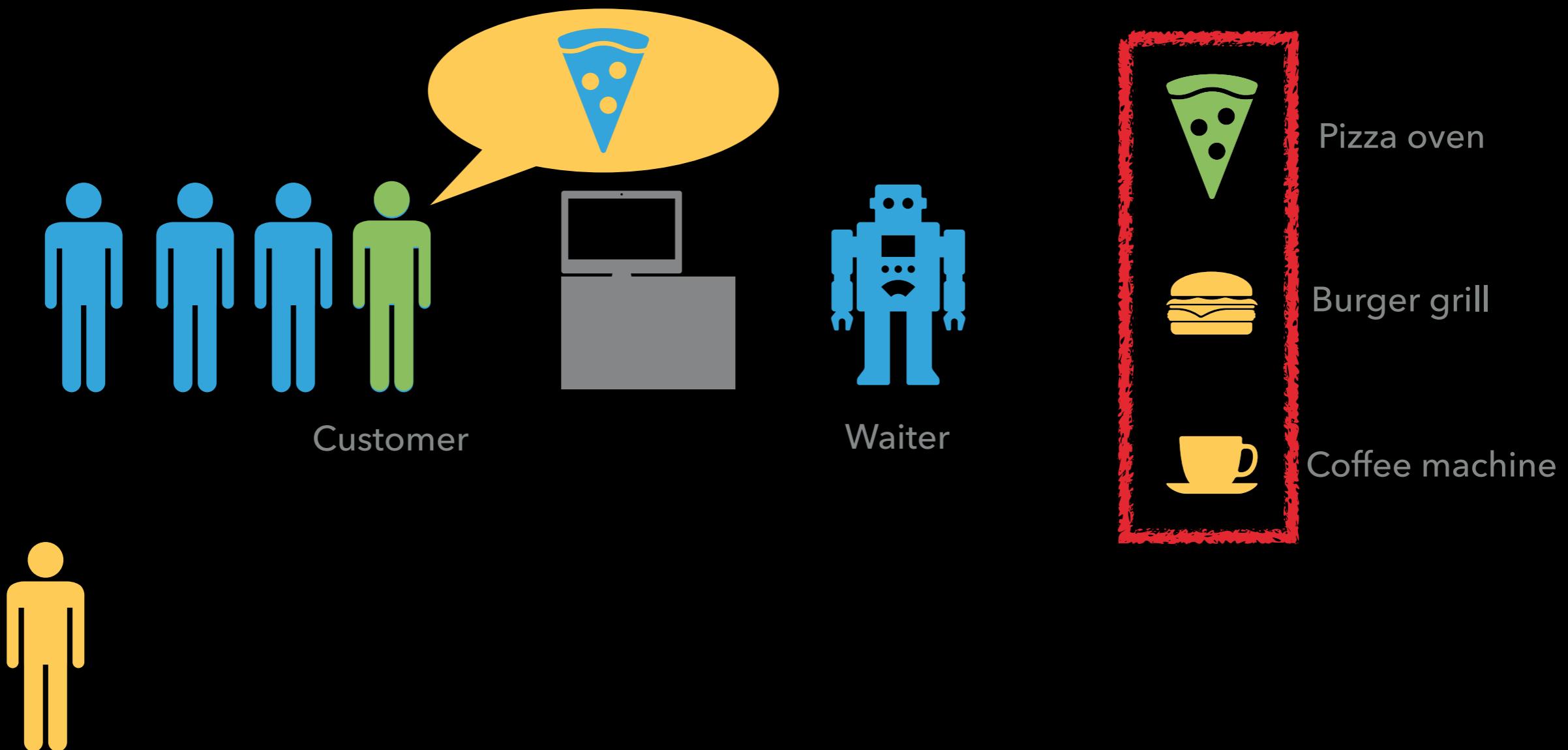
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



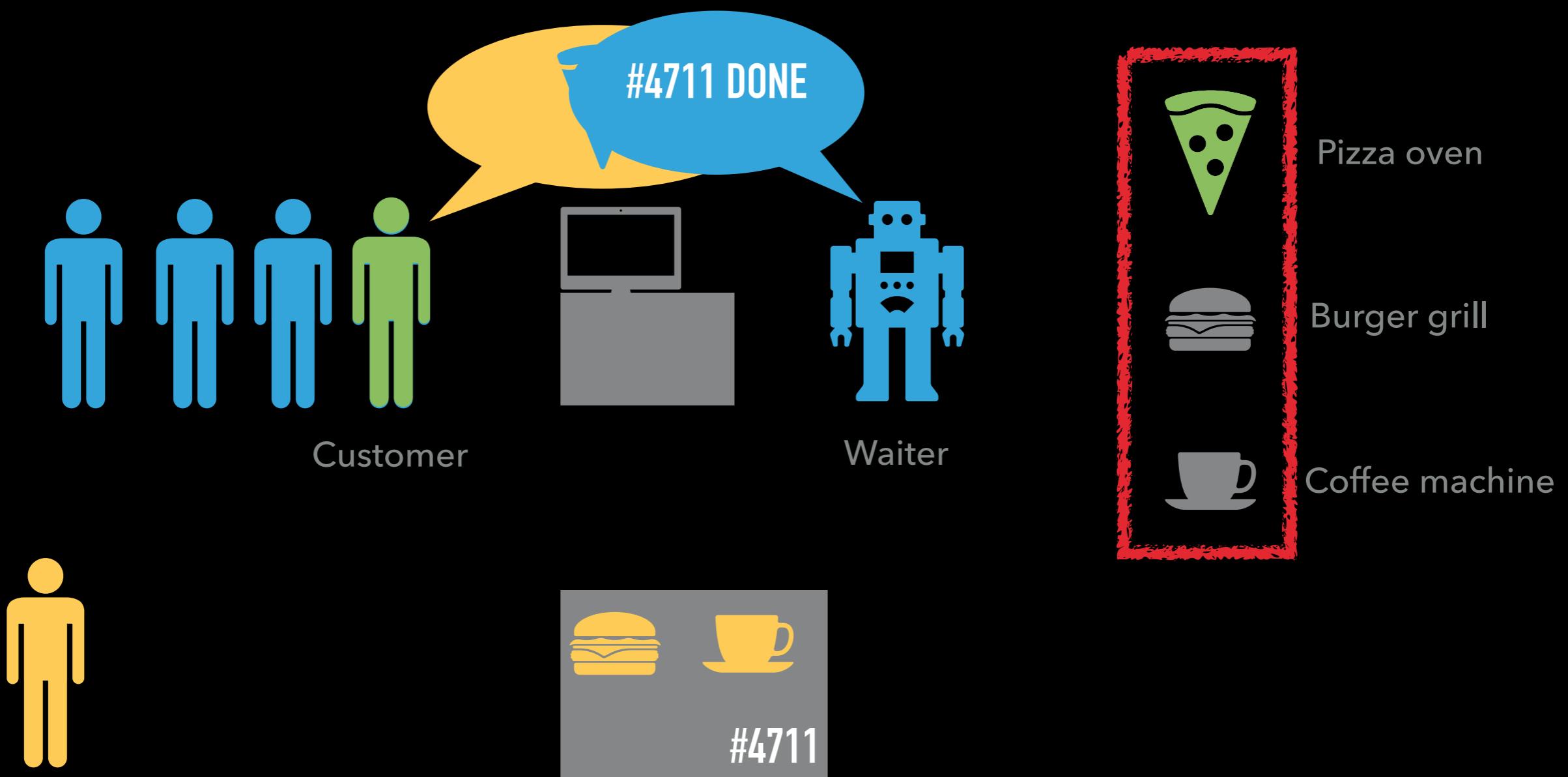
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



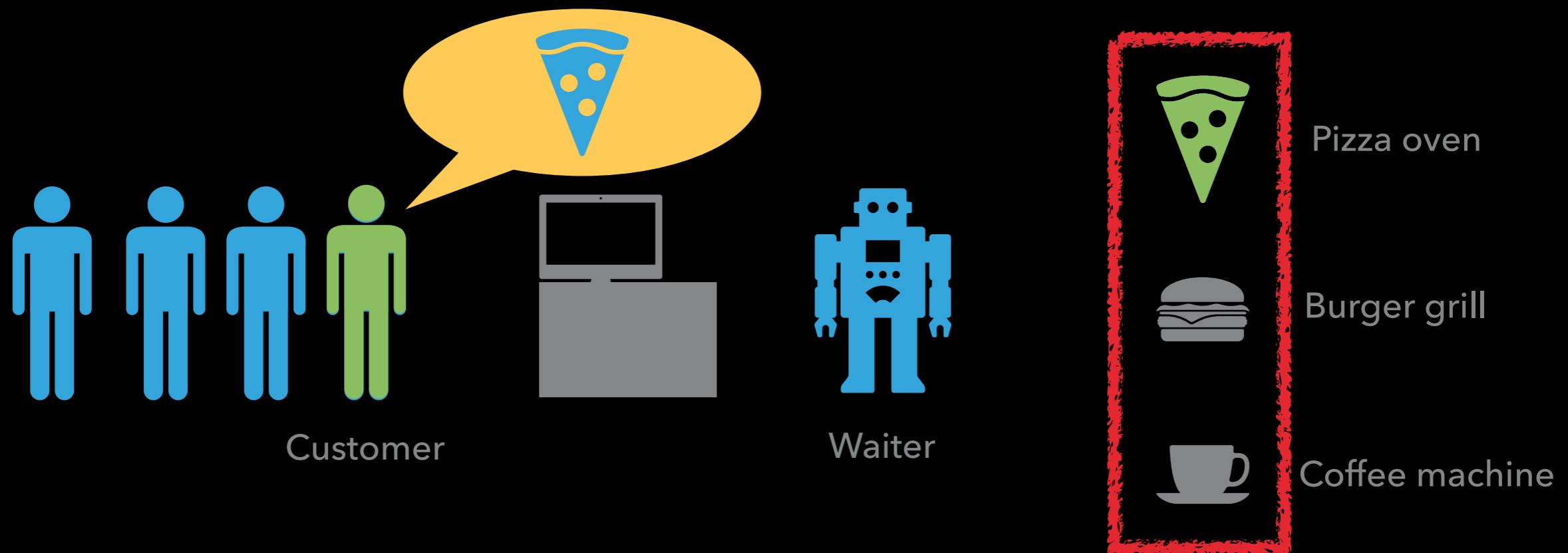
CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



CONFIDENTIAL BURGERS INC. : PARALLEL, OUT OF ORDER EXECUTION



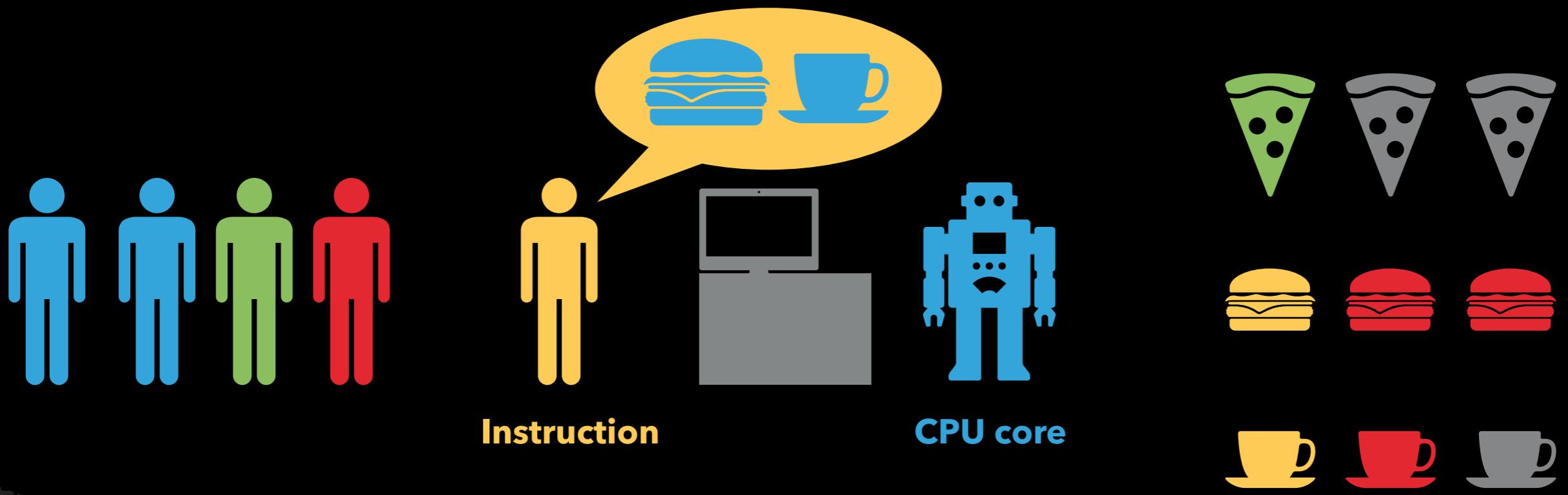
- ▶ Multiple customers' orders executed in parallel¹ and delivered (retired) **in order**

i.e. multiple orders prepared at the same time

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

¹ this is called *superscalar*

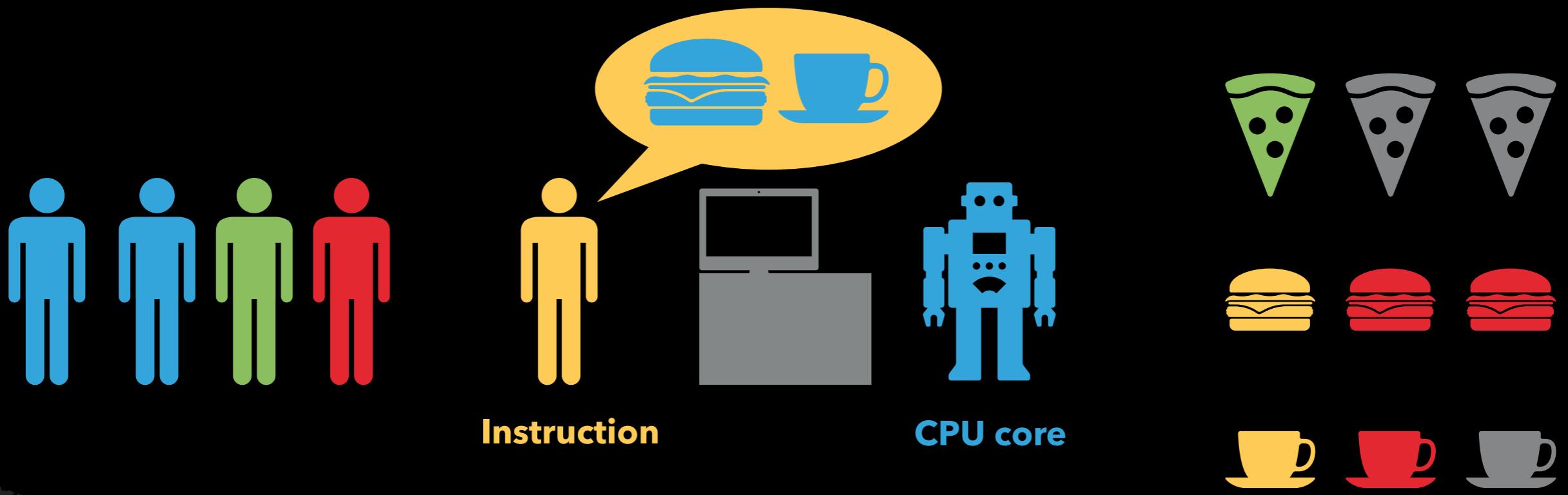
CONFIDENTIAL BURGERS INC.



Adding more resources increase parallelism & throughput.

This is all on one CPU core.

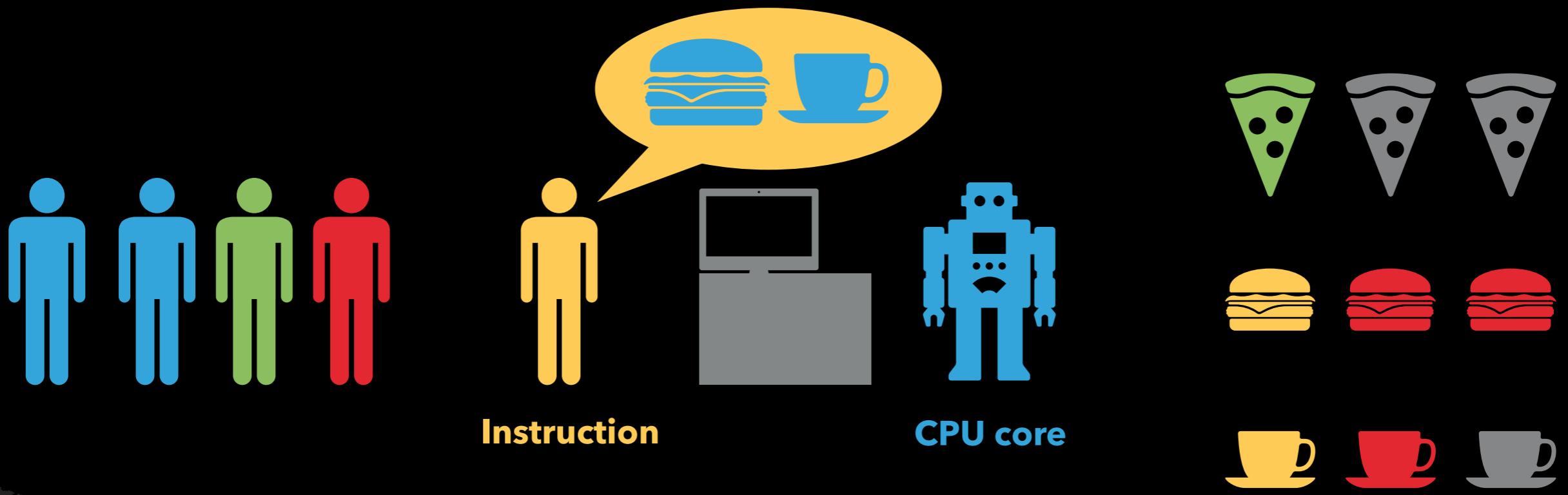
CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



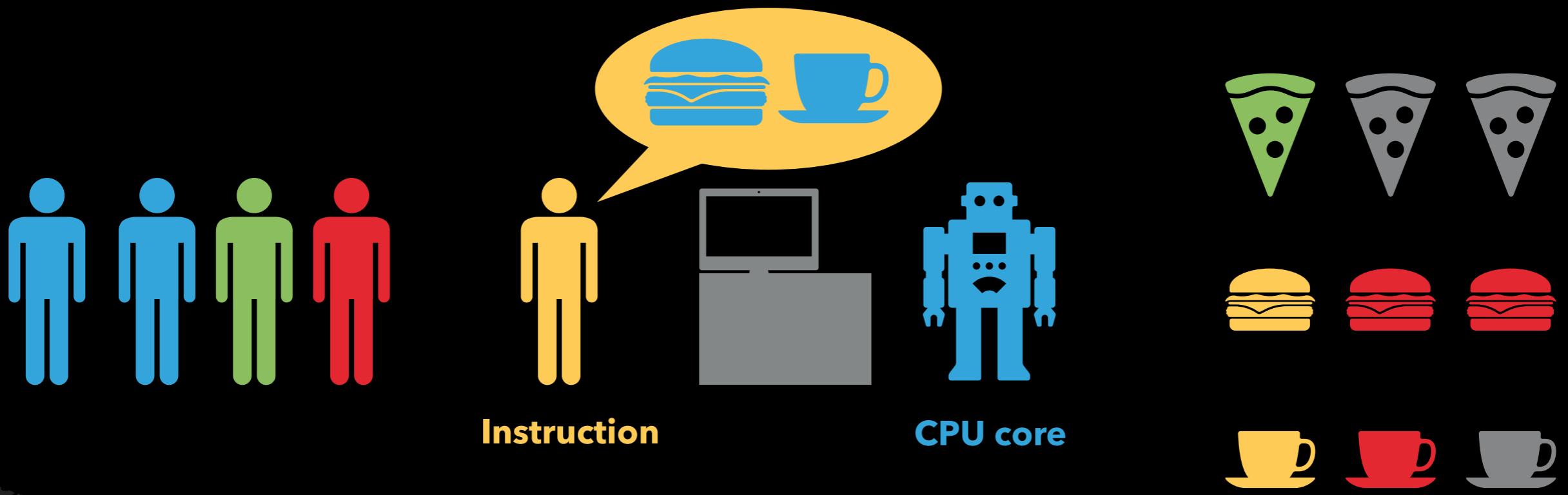
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



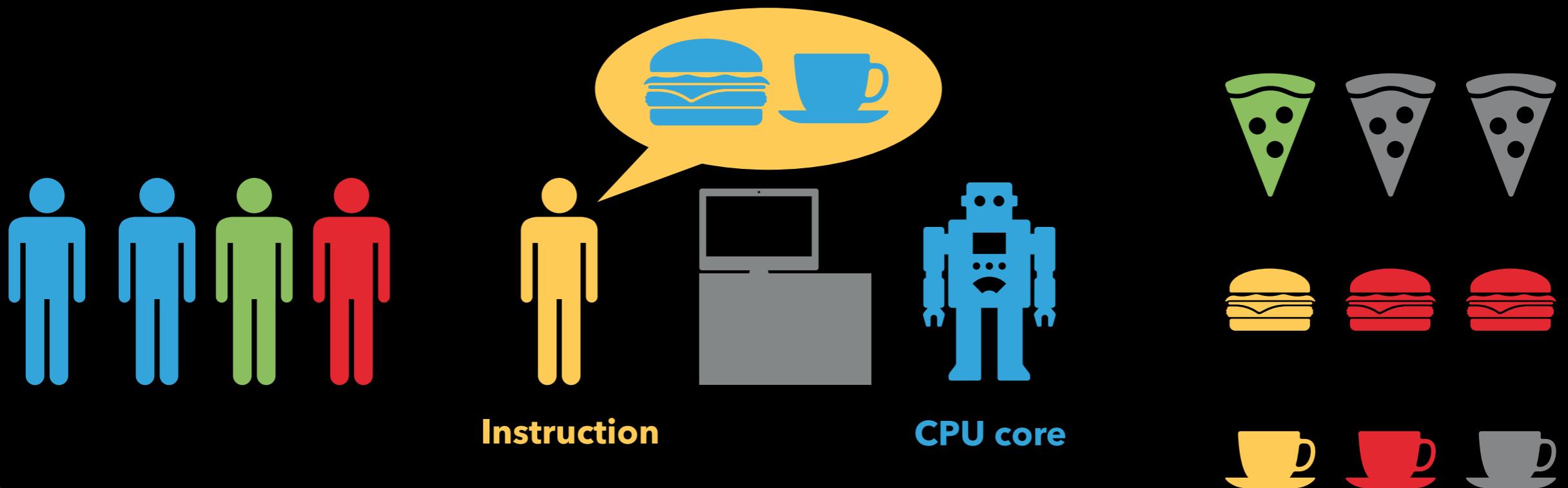
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



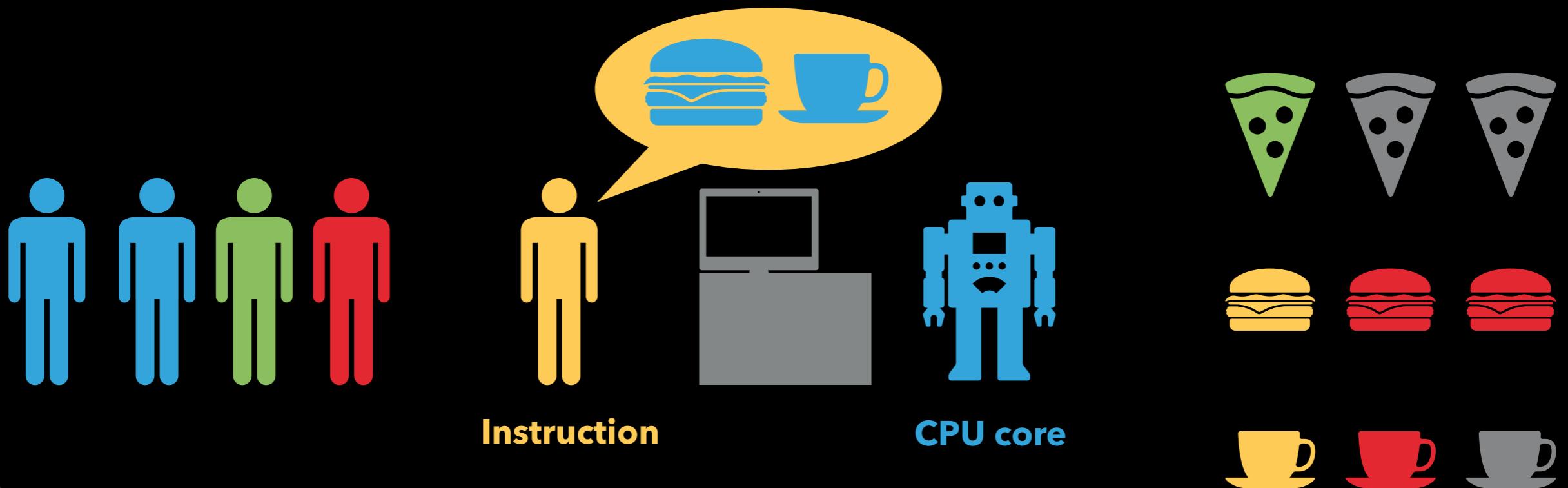
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual µOP execution order

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



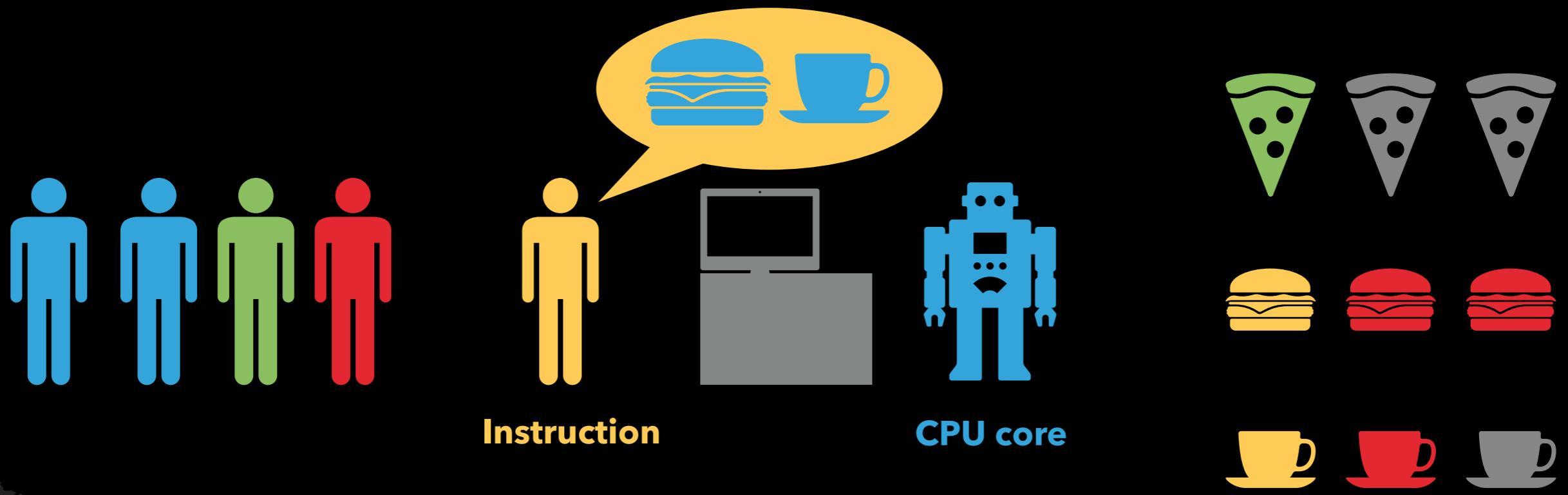
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual μOP execution order

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

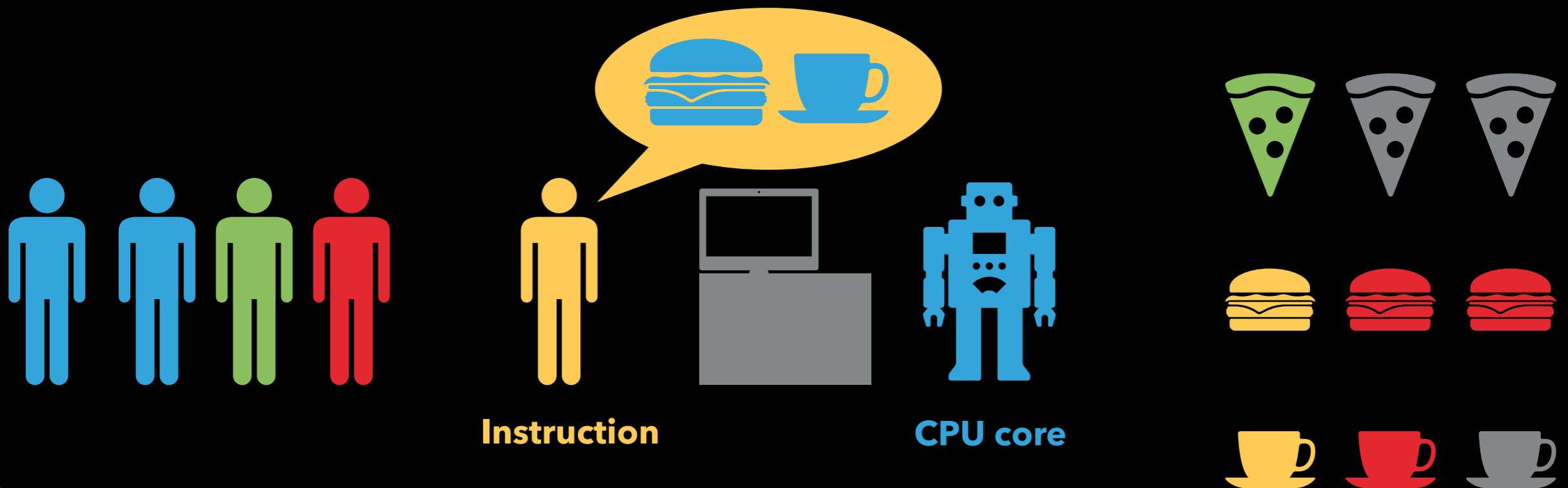


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



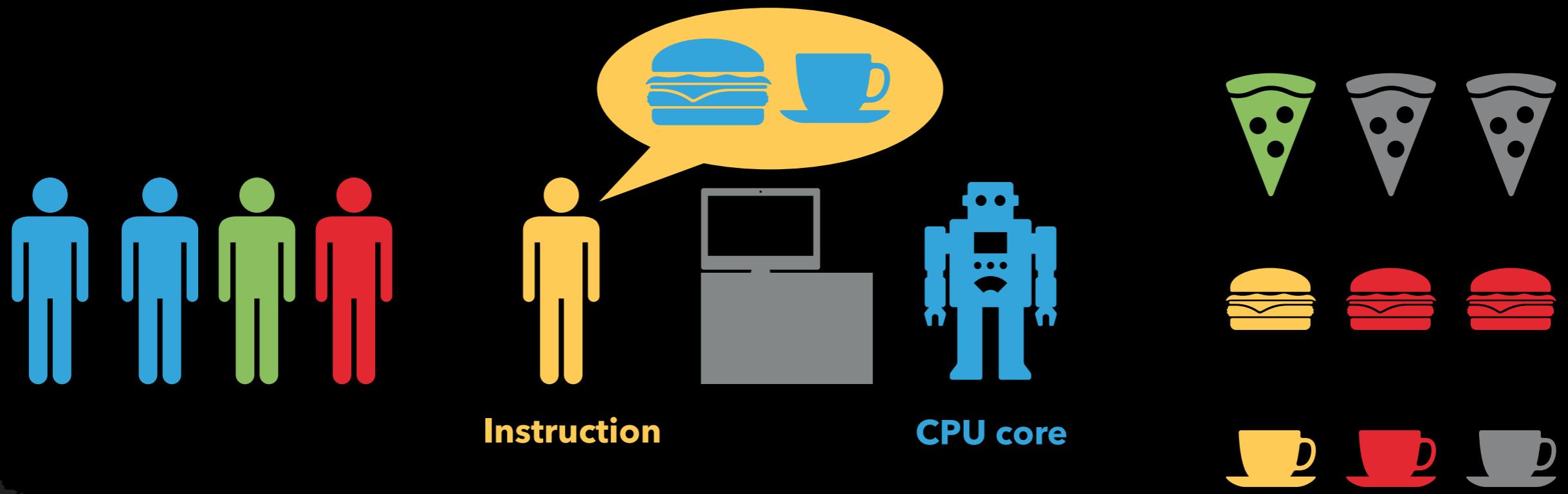
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



Actual μOP execution order

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



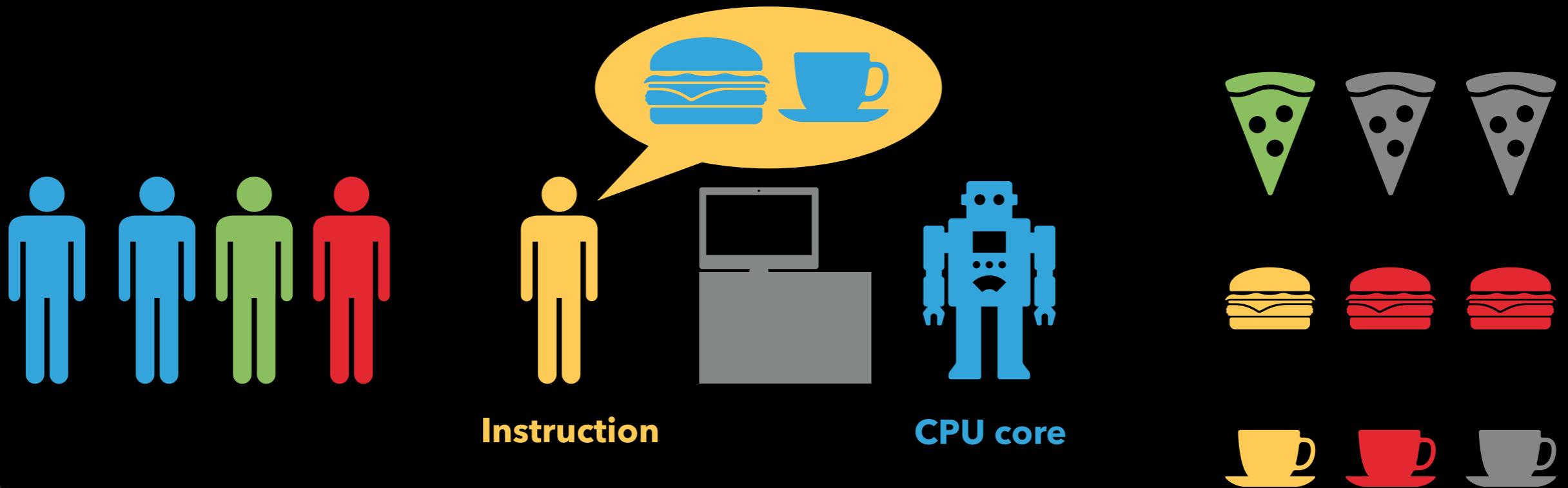
The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.



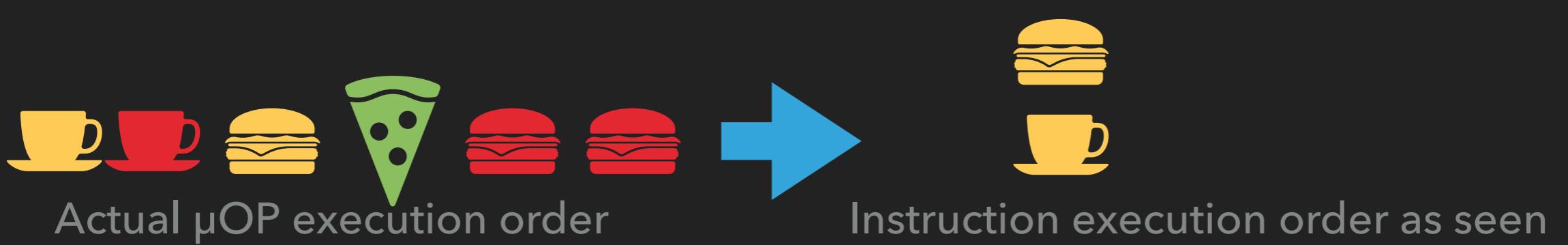
Instruction execution order as seen

CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

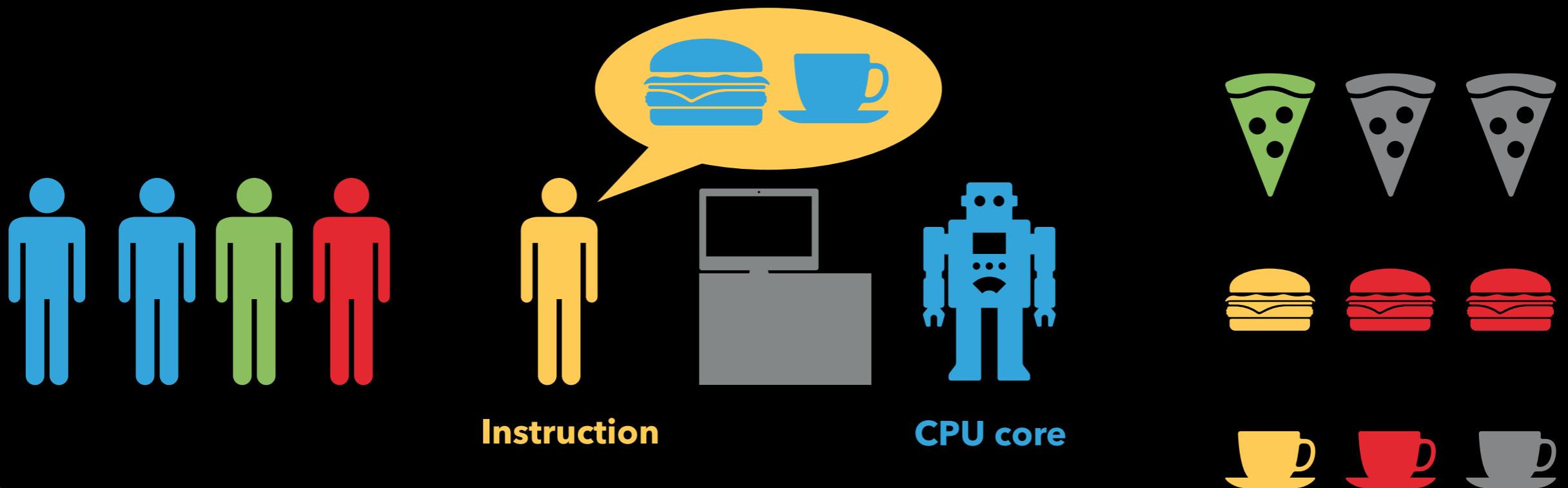


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

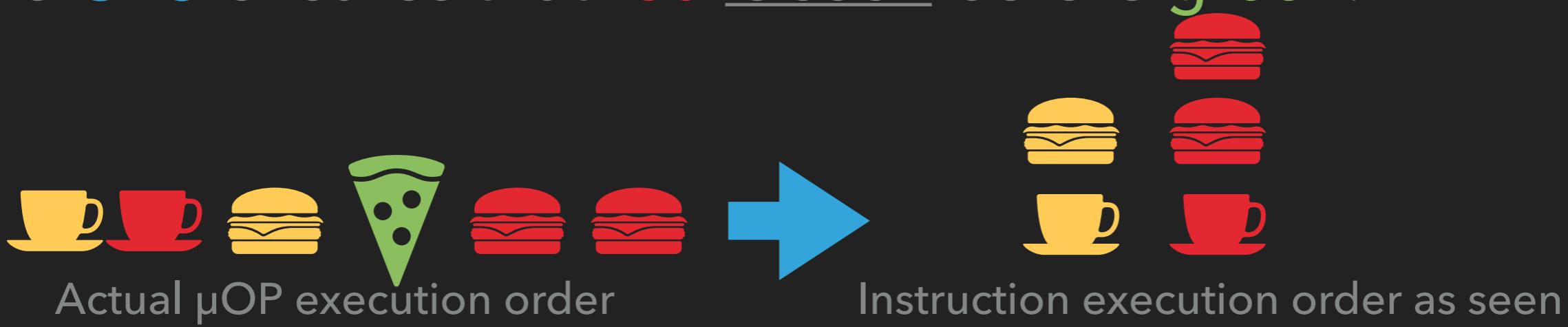


CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT

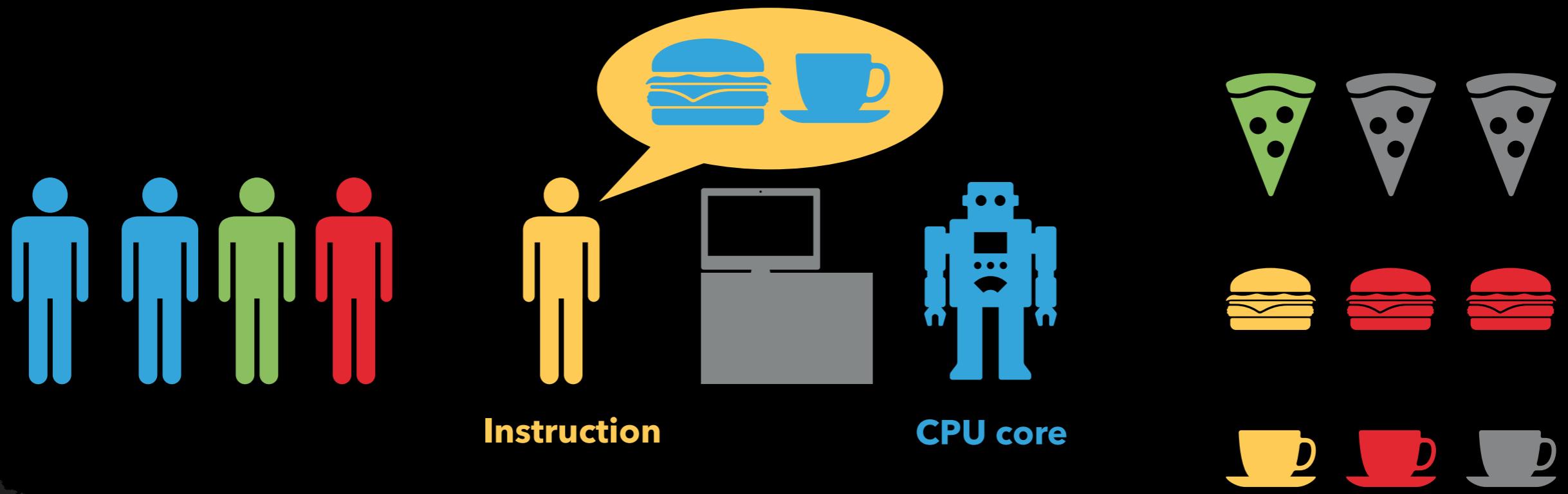


The green instruction will finish before the red instruction.

The CPU ensures that red is seen before green.

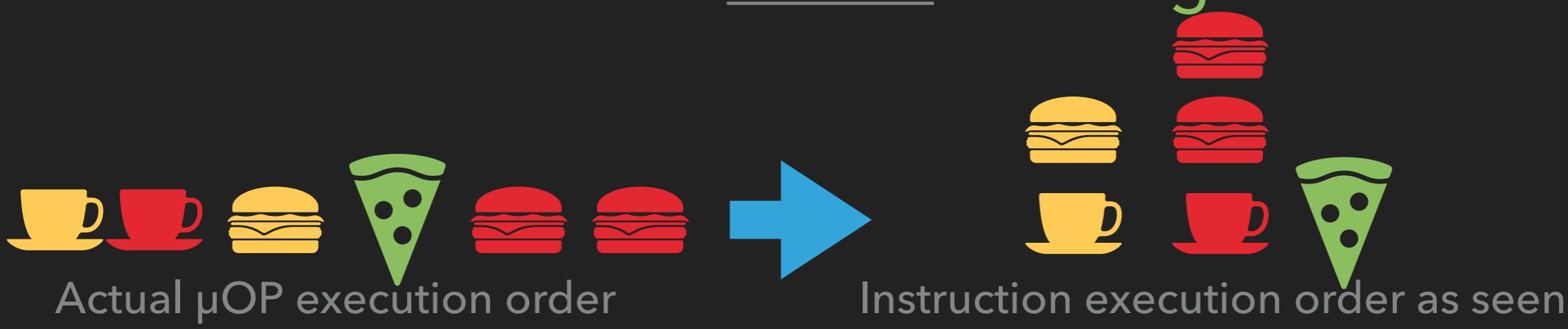


CONFIDENTIAL BURGERS INC. : ORDER IS IMPORTANT



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

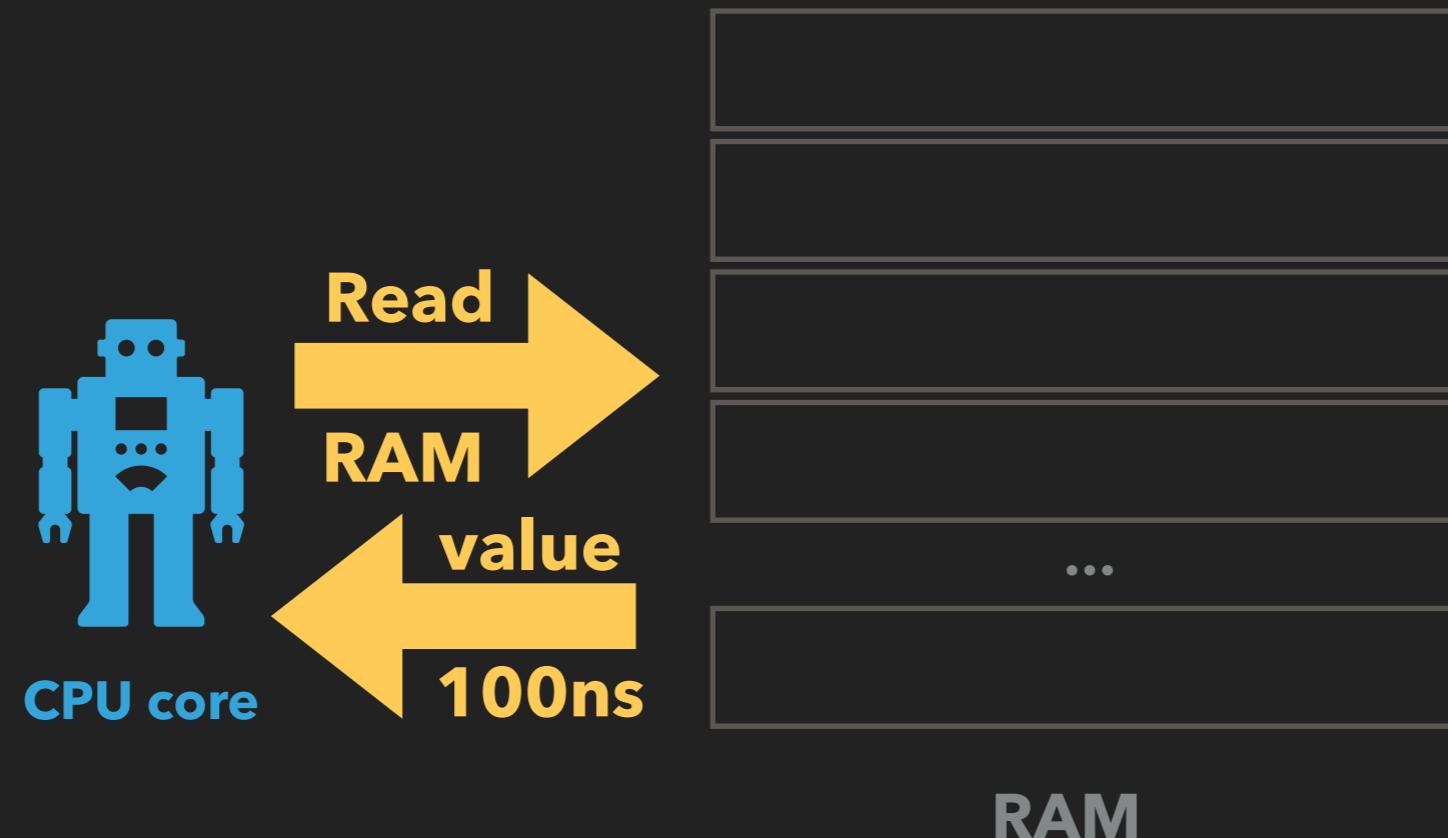
“Stashing” and “retrieving” the secret works via *side channels*.



Side channels are *observable side effects* of actions.

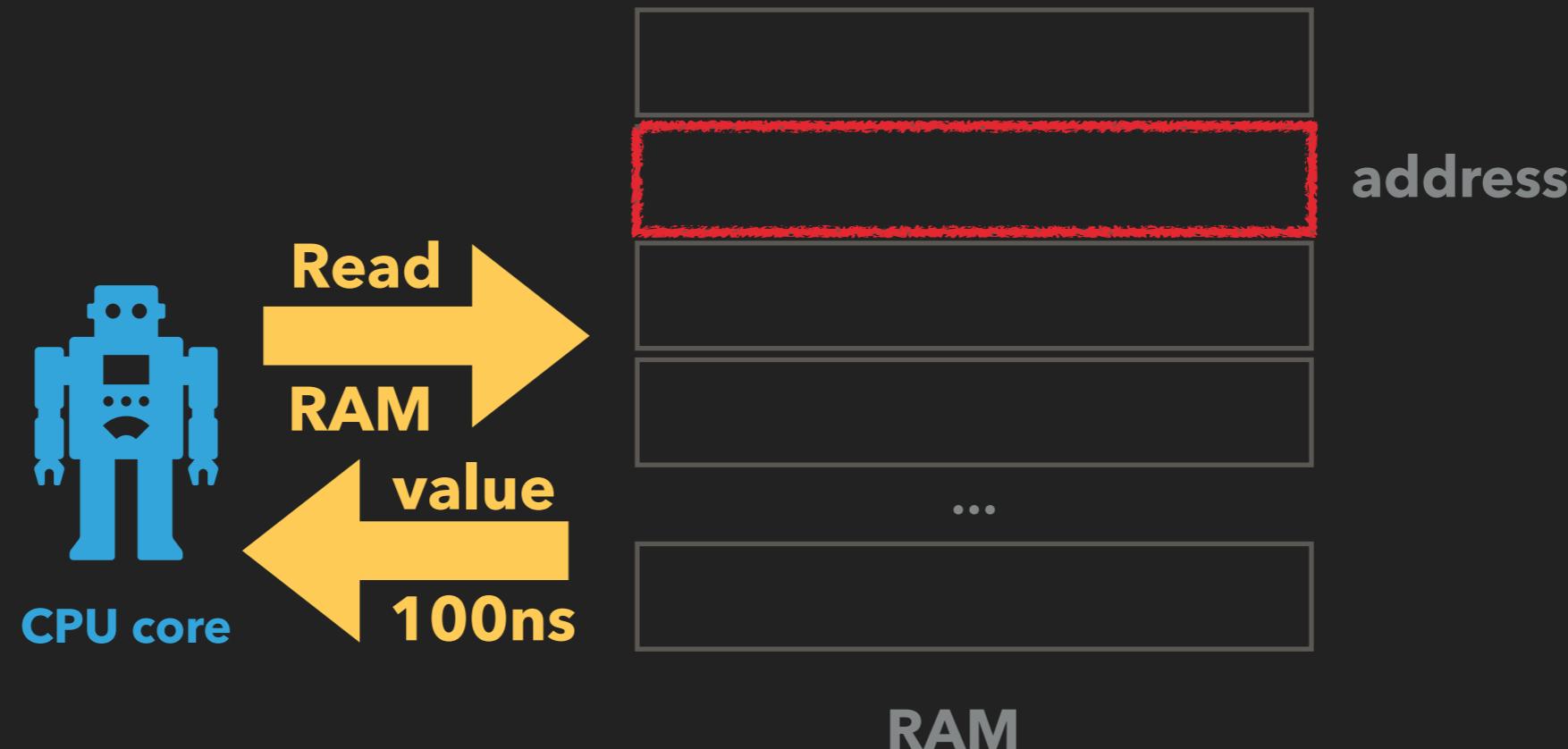
- READ secret from forbidden address
- Stash away secret by caching a memory location that depends on the secret
- Retrieve secret by finding which memory location is cached

MELTDOWN: STASHING AWAY - SIDECHANNEL



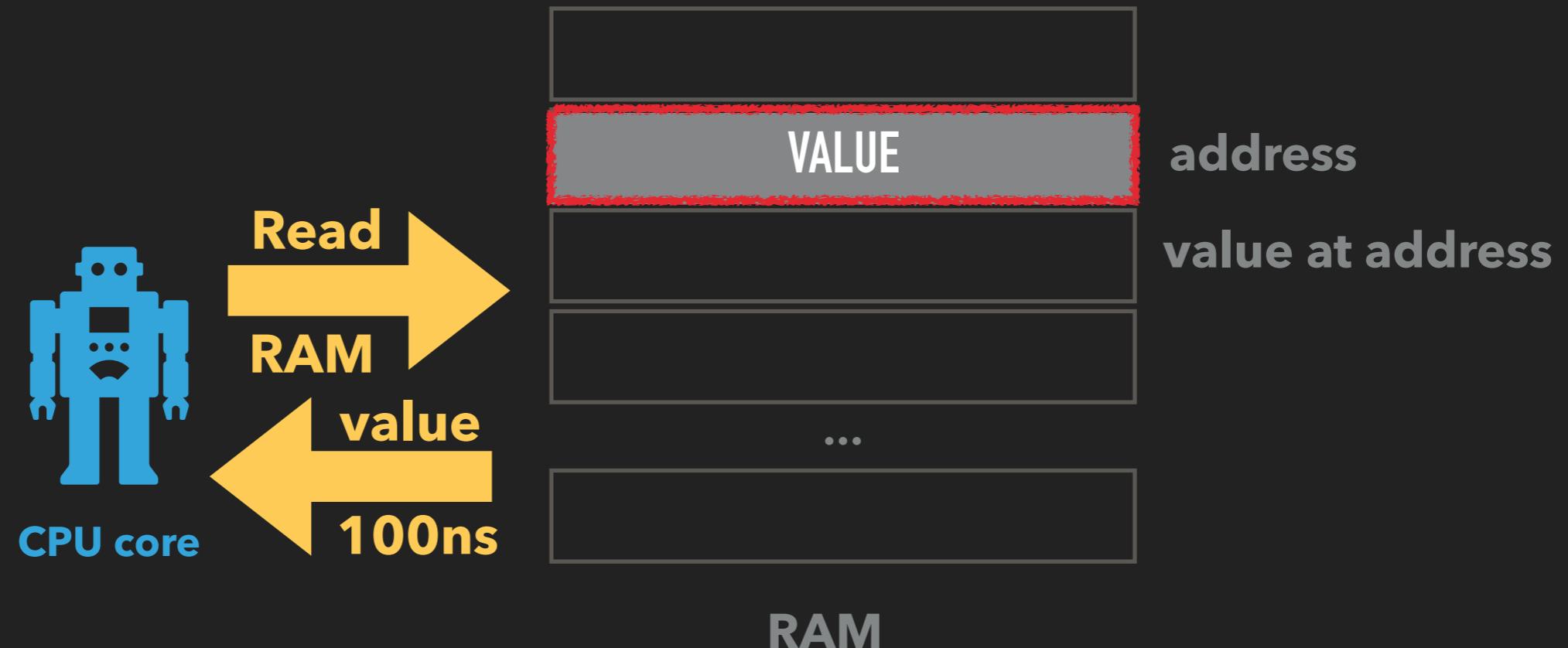
- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

MELTDOWN: STASHING AWAY - SIDECHANNEL



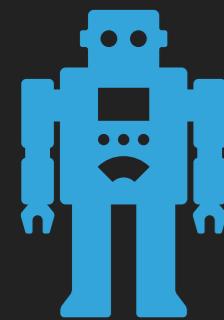
- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

MELTDOWN: STASHING AWAY - SIDECHANNEL

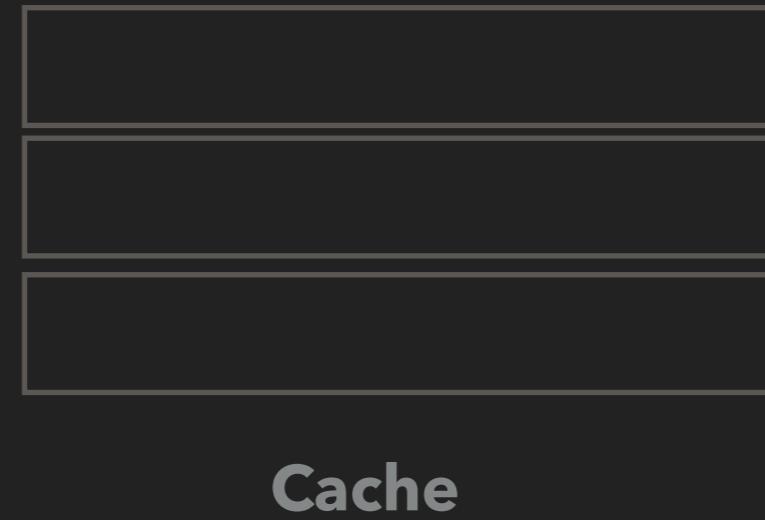


- ▶ Data is stored in RAM
- ▶ RAM is very slow
- ▶ Reading one byte stalls the CPU for hundreds of µOPs

MELTDOWN: STASHING AWAY - SIDECHANNEL



CPU core



VALUE

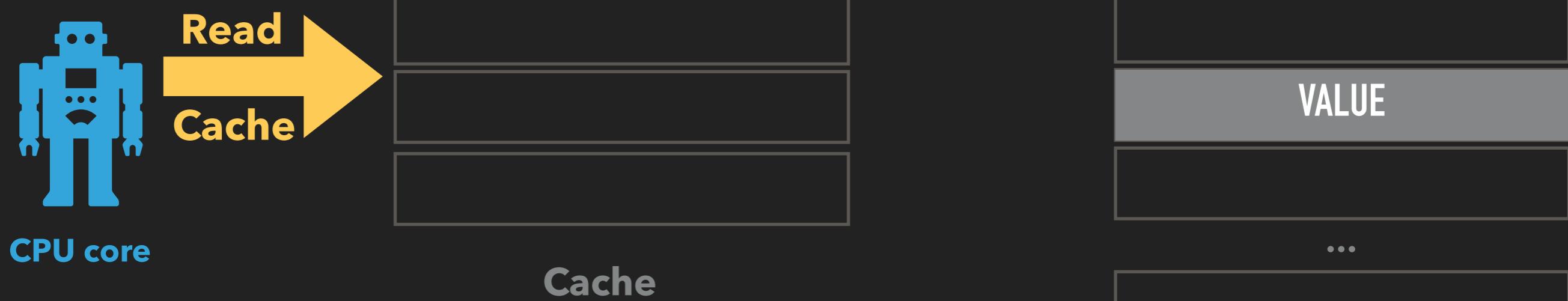
...

RAM

- ▶ Reading one byte stalls the CPU for hundreds of μ 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*”

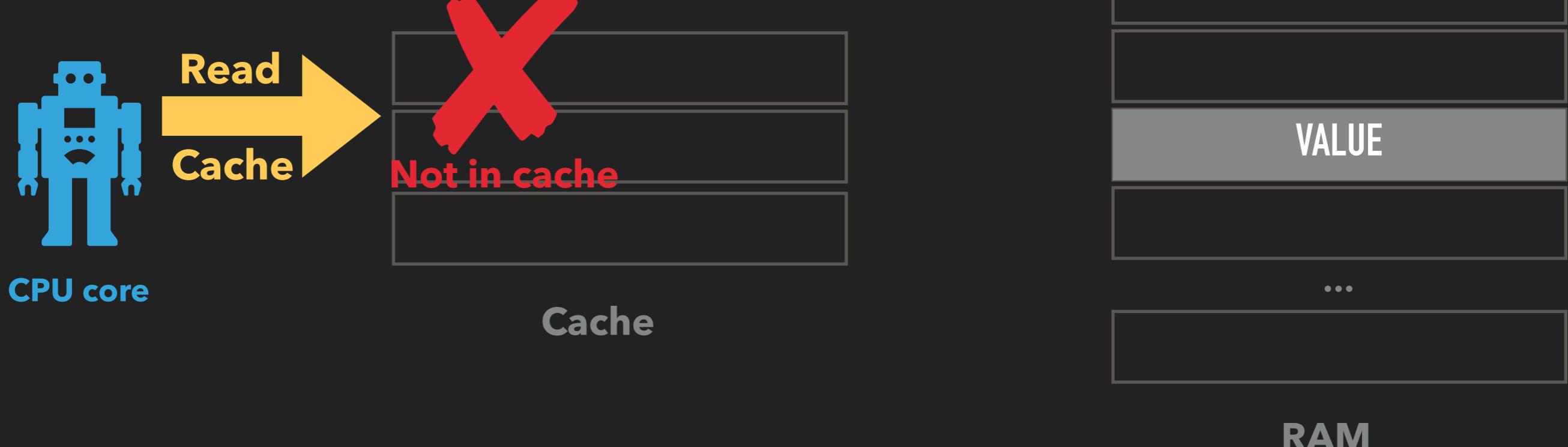
MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of μ 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 μ 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*"

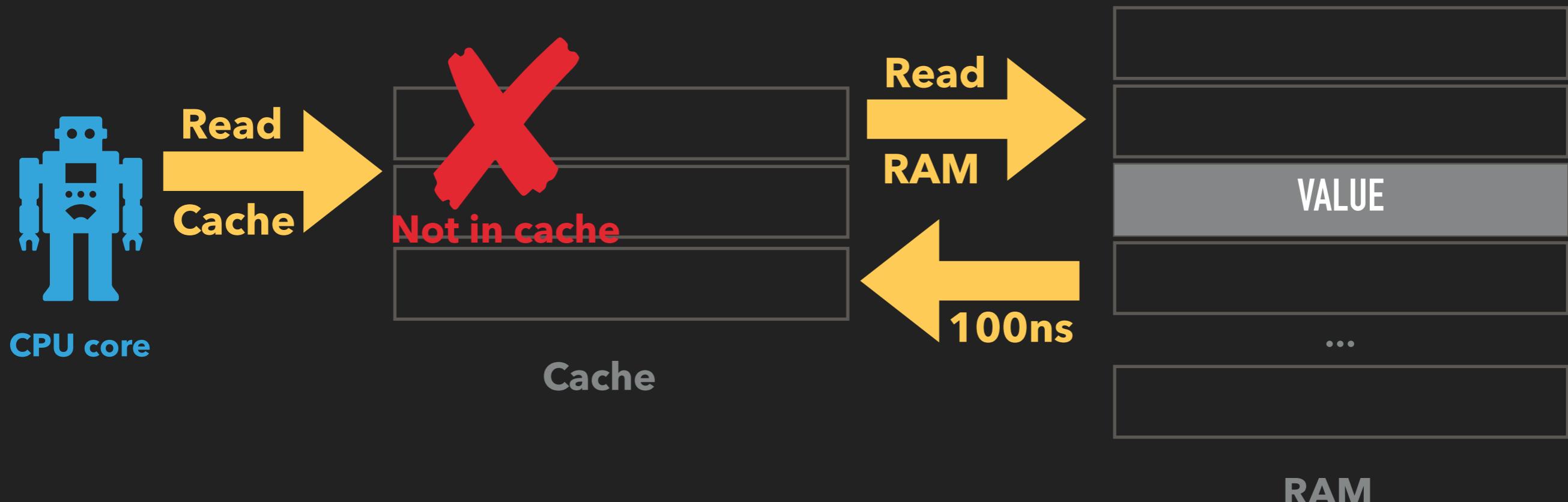
MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ 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*”

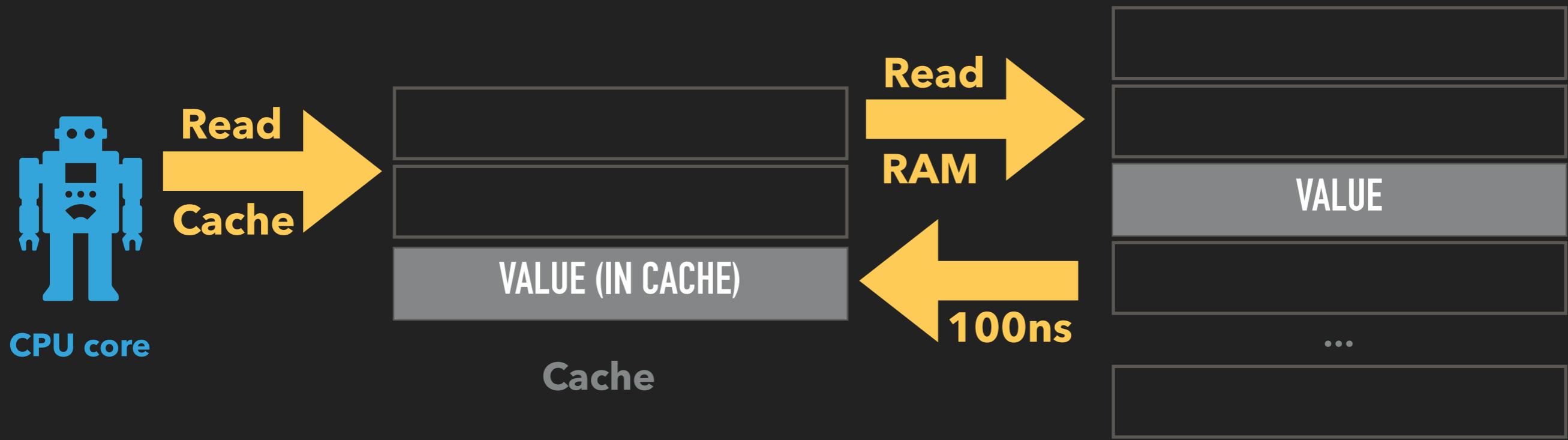
MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of μ 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"

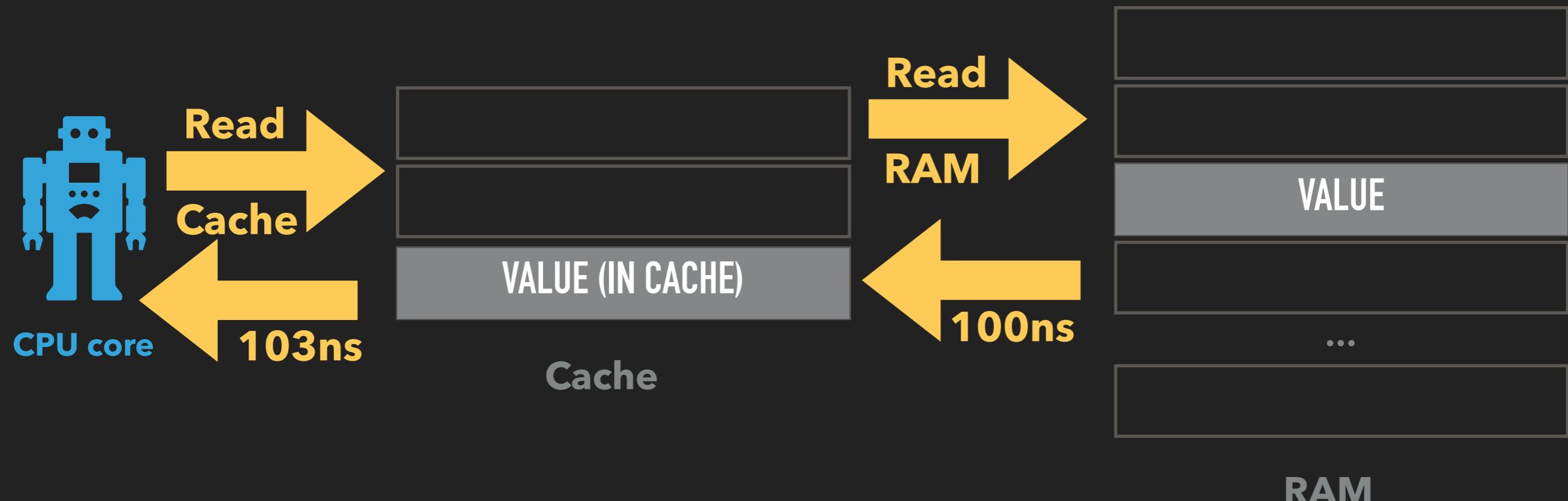
MELTDOWN: STASHING AWAY - SIDECHANNEL



- ▶ Reading one byte stalls the CPU for hundreds of μ 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*”

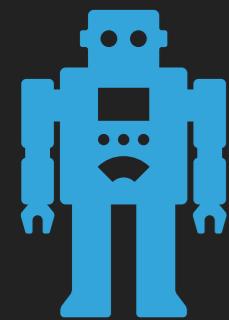
MELTDOWN: STASHING AWAY - SIDECHANNEL



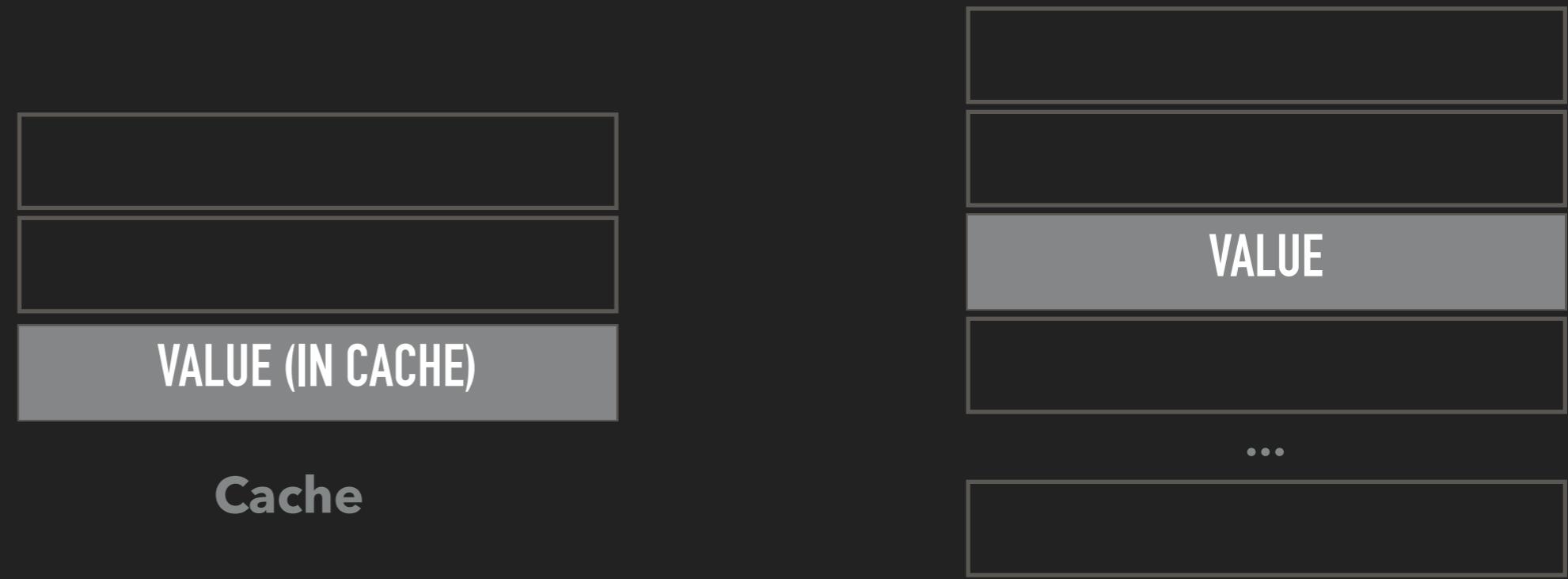
- ▶ Reading one byte stalls the CPU for hundreds of µOPs
- ▶ CPU caches considerably speed this up
- ▶ E.g. reading cached takes 3ns, reading uncached 103ns

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

MELTDOWN: STASHING AWAY - SIDECHANNEL



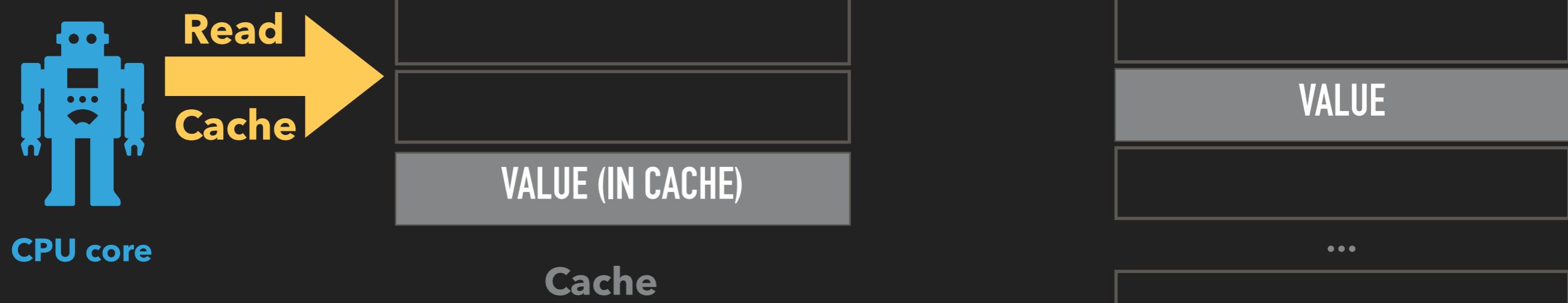
CPU core



- ▶ 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 μ 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 μ 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 μ 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 (μ OPs):

1. Check that program may read from address
2. Store the value at address in register¹

If 1 fails the program is aborted.

This can be handled by the program.

¹ Register: The CPUs scratchpad

“READ” INSTRUCTION

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

1. Check that program may read from address
2. Store the value at address in register¹

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

¹ 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*

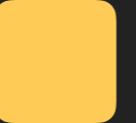
μOPs ordered by *execution*

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 a **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

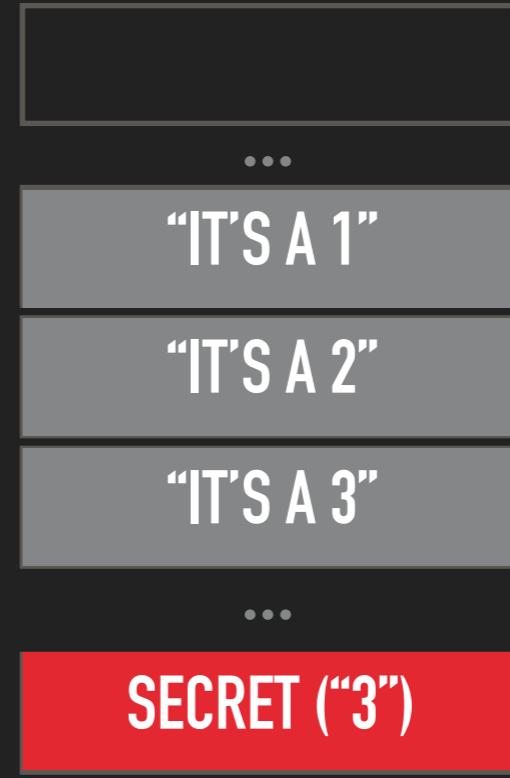
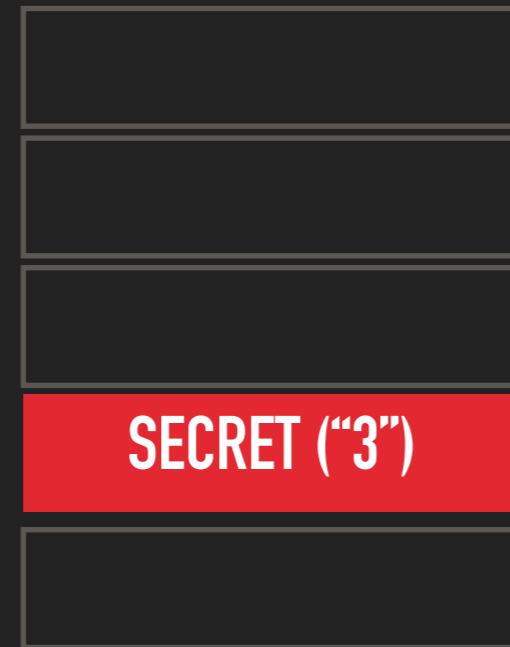
MELTDOWN: THE ATTACK

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



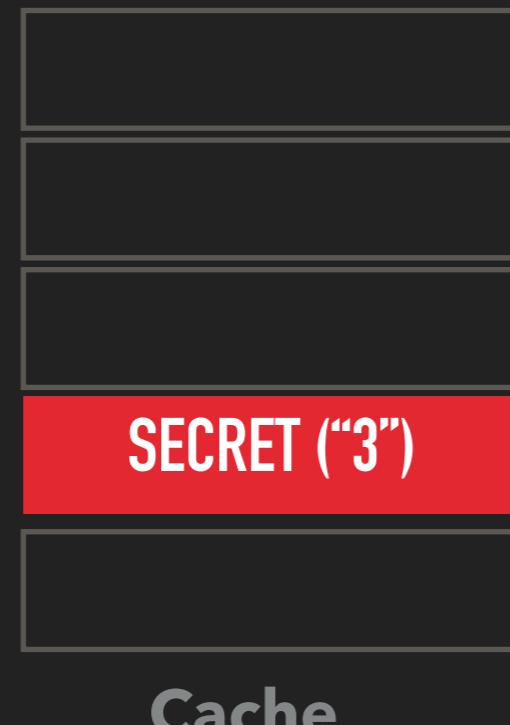
MELTDOWN: THE ATTACK

110011010
010111010
111100100
000101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010

Collector



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



- ▶ Meltdown needs some preconditions



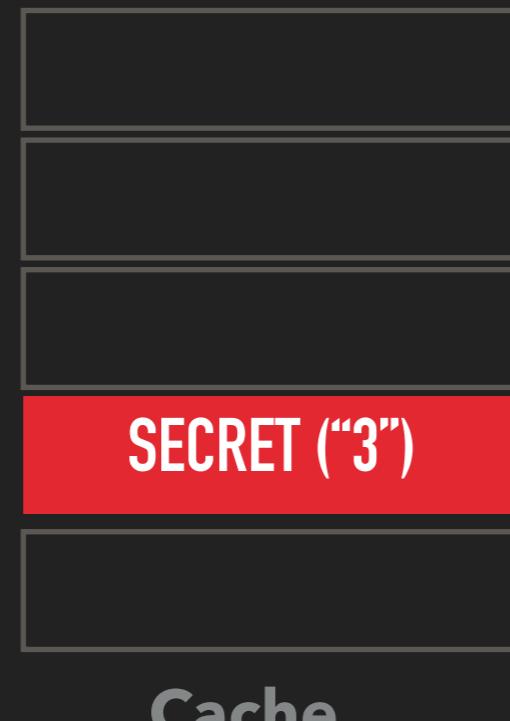
MELTDOWN: THE ATTACK

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)

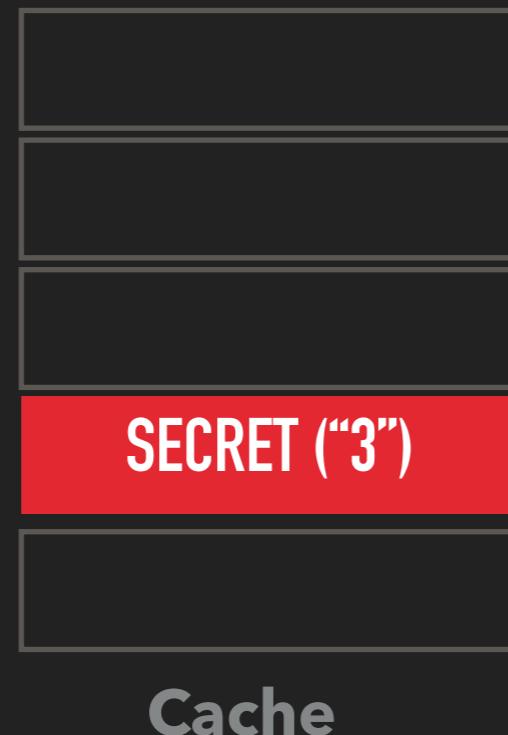
MELTDOWN: THE ATTACK

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

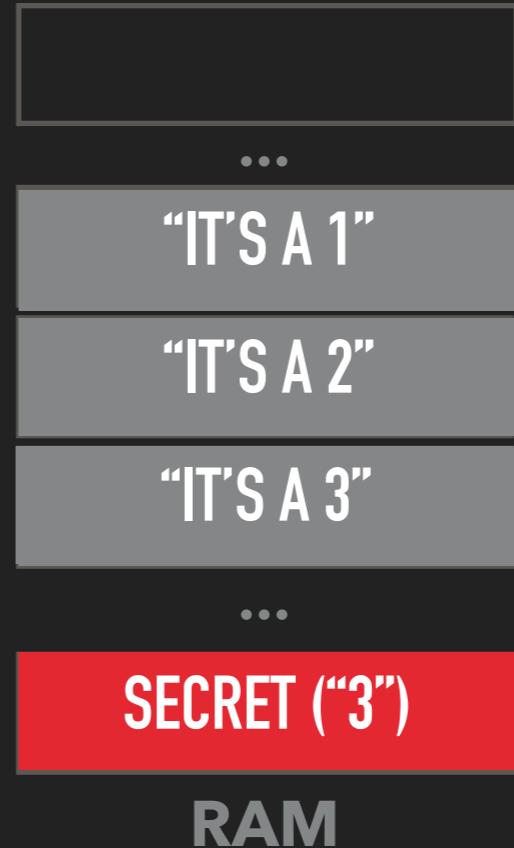
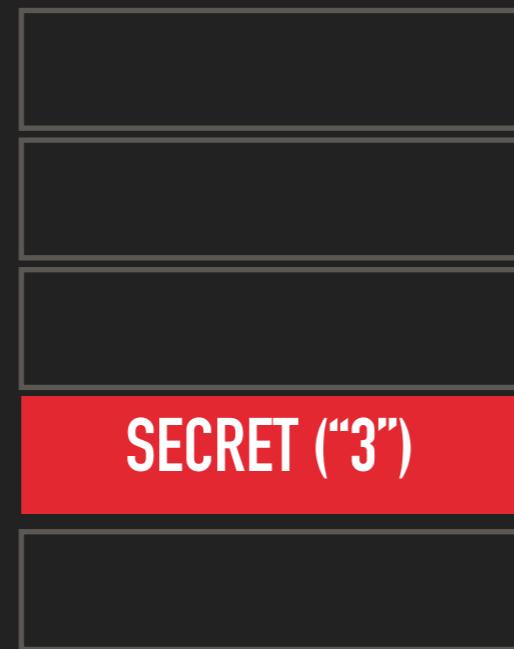
MELTDOWN: THE ATTACK

110011010
010111010
111100100
000101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010

Collector



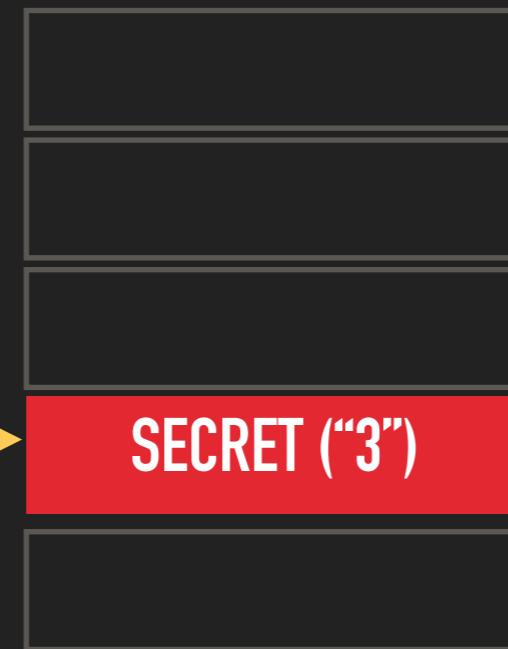
MELTDOWN: THE ATTACK

110011010
010111010
111100100
000101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010

Collector



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



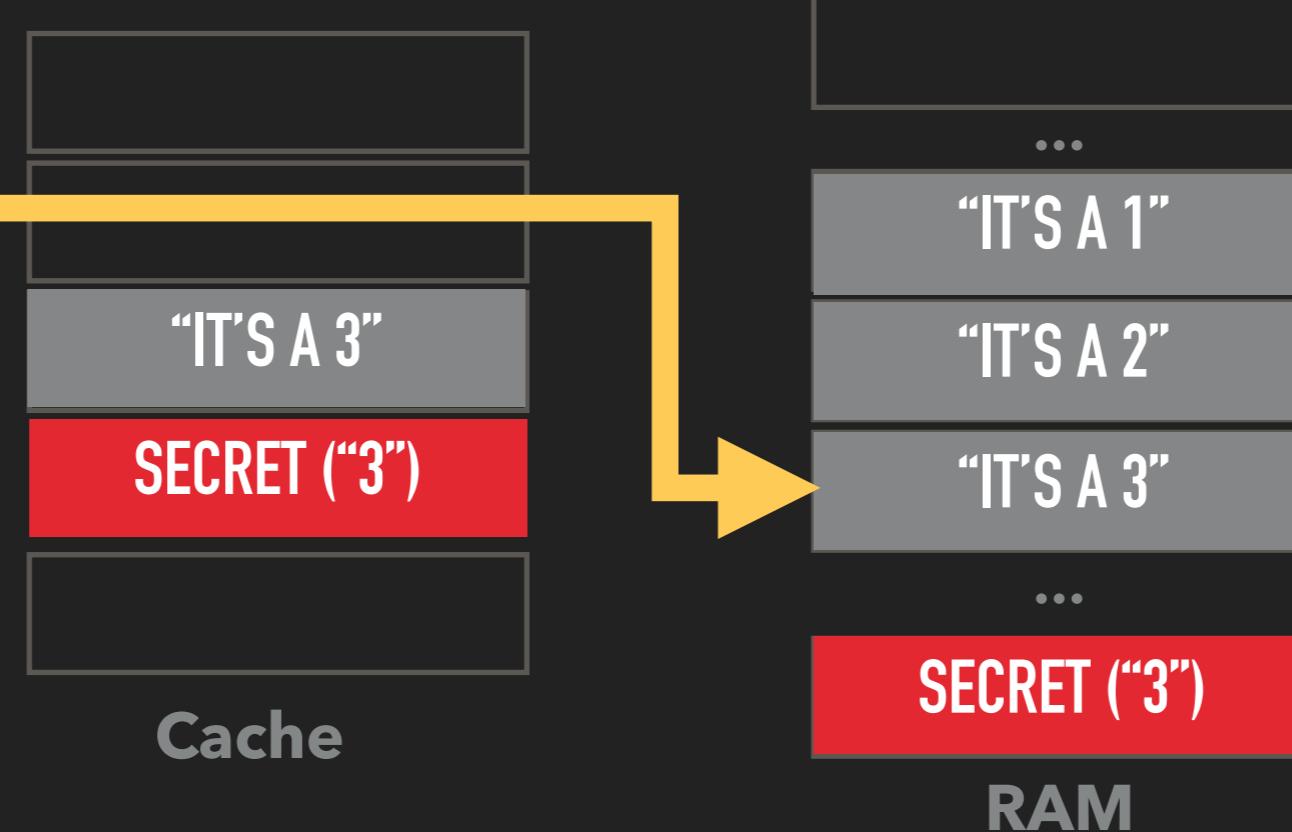
MELTDOWN: THE ATTACK



110011010
010111010
111100100
000101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010



Collector

- 2 1. Spy will read the **secret**
- 2 2. Depending on the **value**, Spy will cache a grey block¹

¹ 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 block¹
- 1 3. CPU detects **Spys** access validation and terminates **Spy**

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

MELTDOWN: THE ATTACK

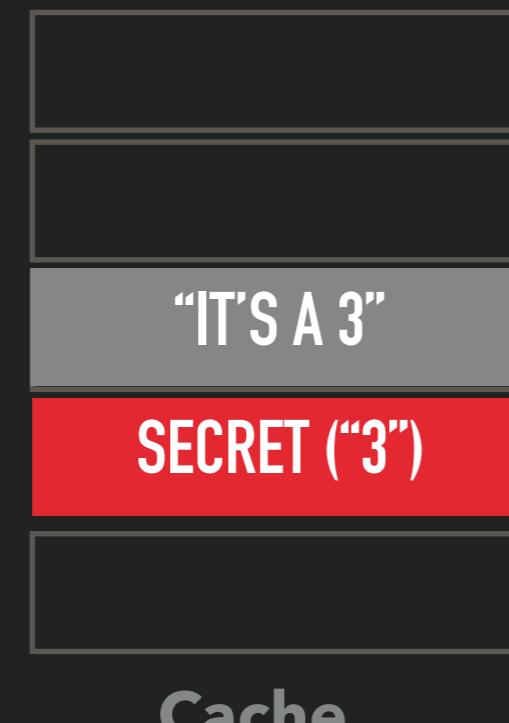


110011010
010111010
111100100
00101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010

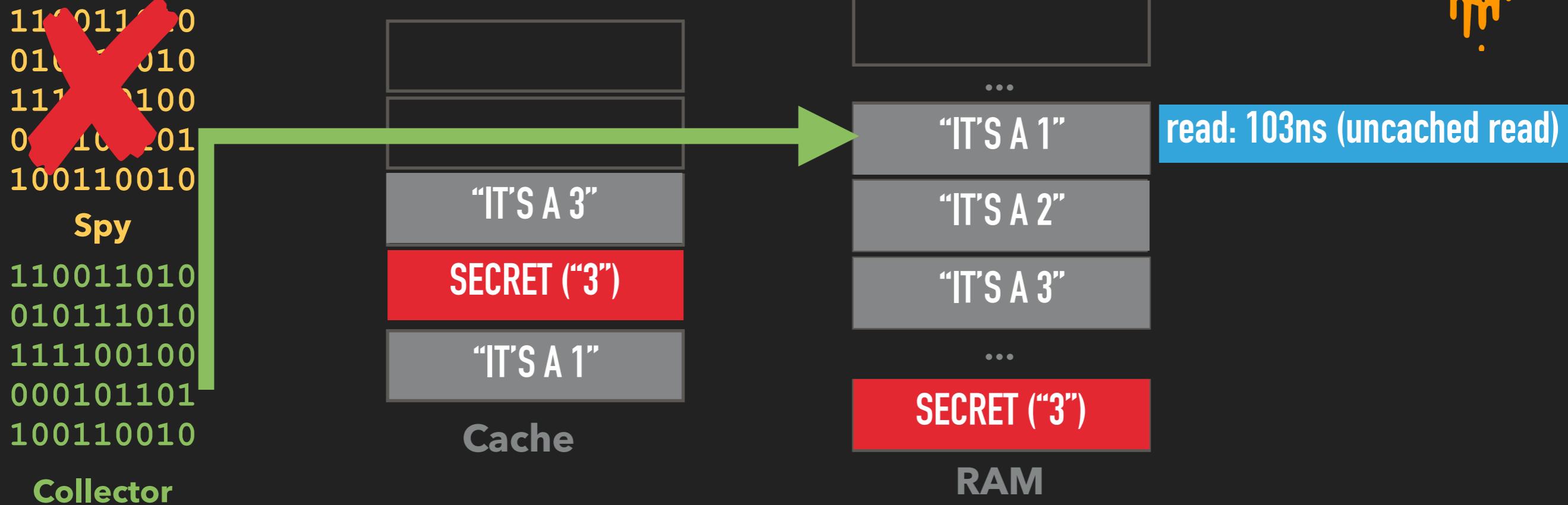
Collector



- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block¹
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

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

MELTDOWN: THE ATTACK



Collector

- 2 1. Spy will read the secret
- 2 2. Depending on the value, Spy will cache a grey block¹
- 1 3. CPU detects Spys access validation and terminates Spy
4. Collector now reads all grey blocks and stops the time

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

MELTDOWN: THE ATTACK

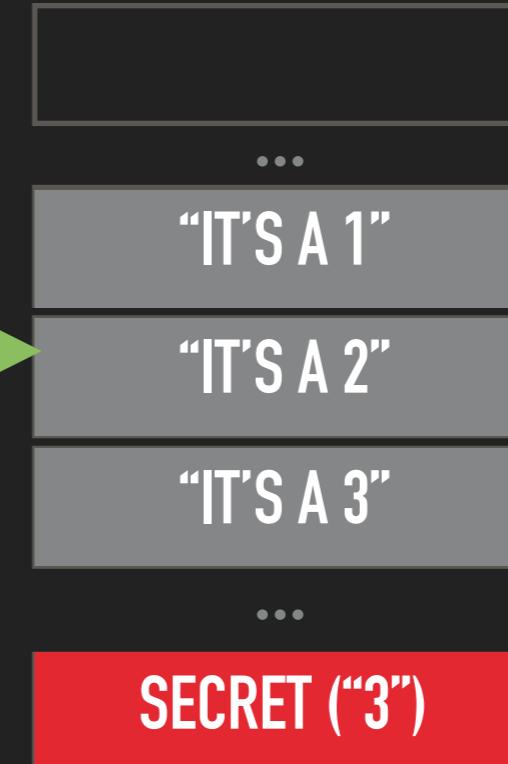
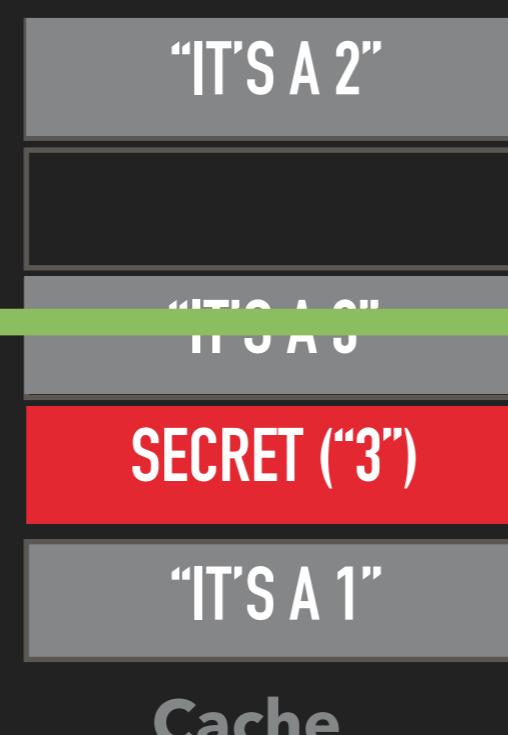


110011010
010111010
111100100
00101101
100110010

Spy

110011010
010111010
111100100
000101101
100110010

Collector



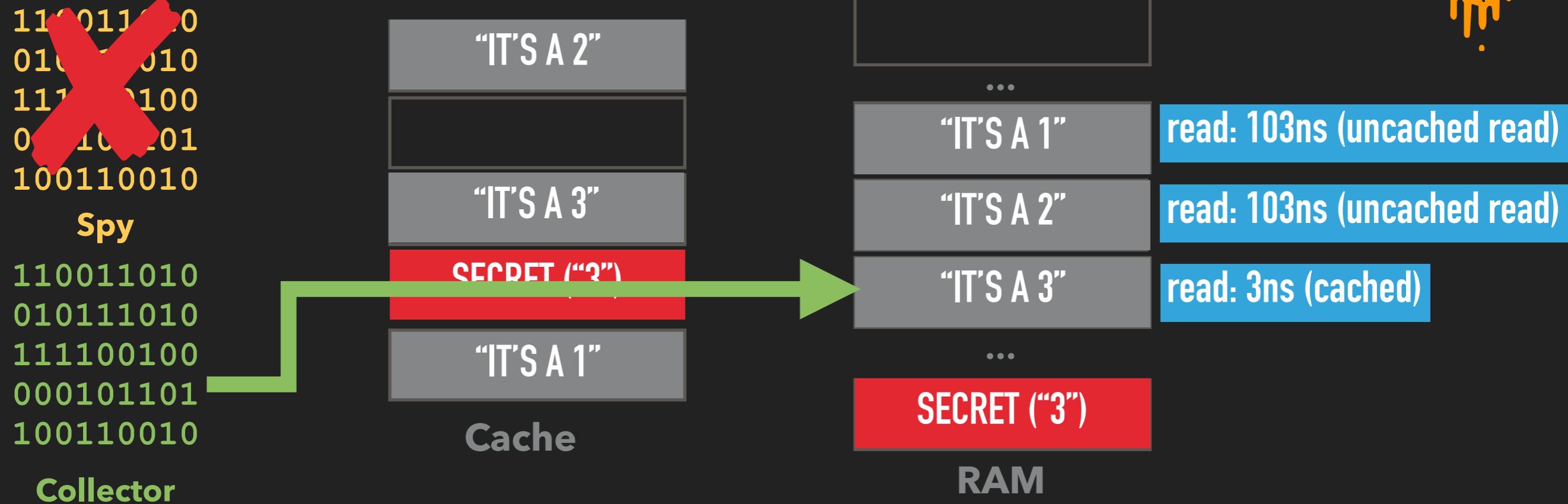
read: 103ns (uncached read)
read: 103ns (uncached read)

- 2 1. **Spy** will read the **secret**
- 2 2. Depending on the **value**, **Spy** will cache a grey block¹
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

¹ 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 block¹
- 1 3. CPU detects **Spys** access validation and terminates **Spy**
4. **Collector** now reads all grey blocks and stops the time

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

MELTDOWN: THE ATTACK

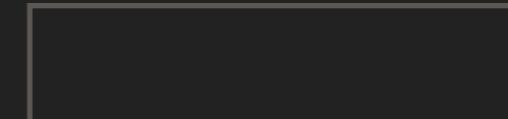
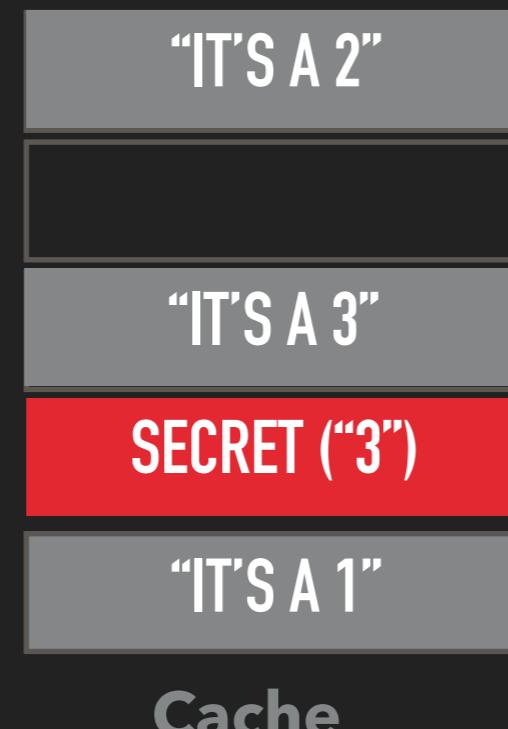


~~110011010
010111010
111100100
000101101
100110010~~

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 block¹
- 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

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

MELTDOWN: THE ATTACK

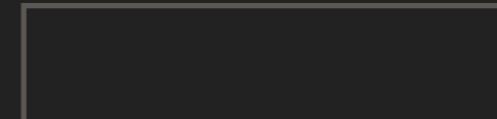


110011010
010111010
111100100
00101101
100110010

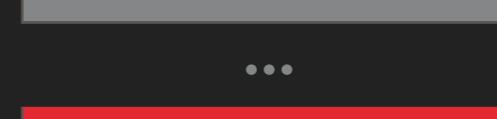
Spy

110011010
010111010
111100100
000101101
100110010

Collector



...



read: 103ns (uncached read)

read: 103ns (uncached read)

read: 3ns (cached)

RAM

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

2 Depending on the **value**, **Spy** will cache a grey block¹

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

¹ 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 μ OPs
- ▶ Timing side channels for the cache

This allows an attacker to

- ▶ Read all memory mapped¹ in a process
- ▶ This often includes all other processes memory
- ▶ This does NOT allow reading “outside of a VM²”

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



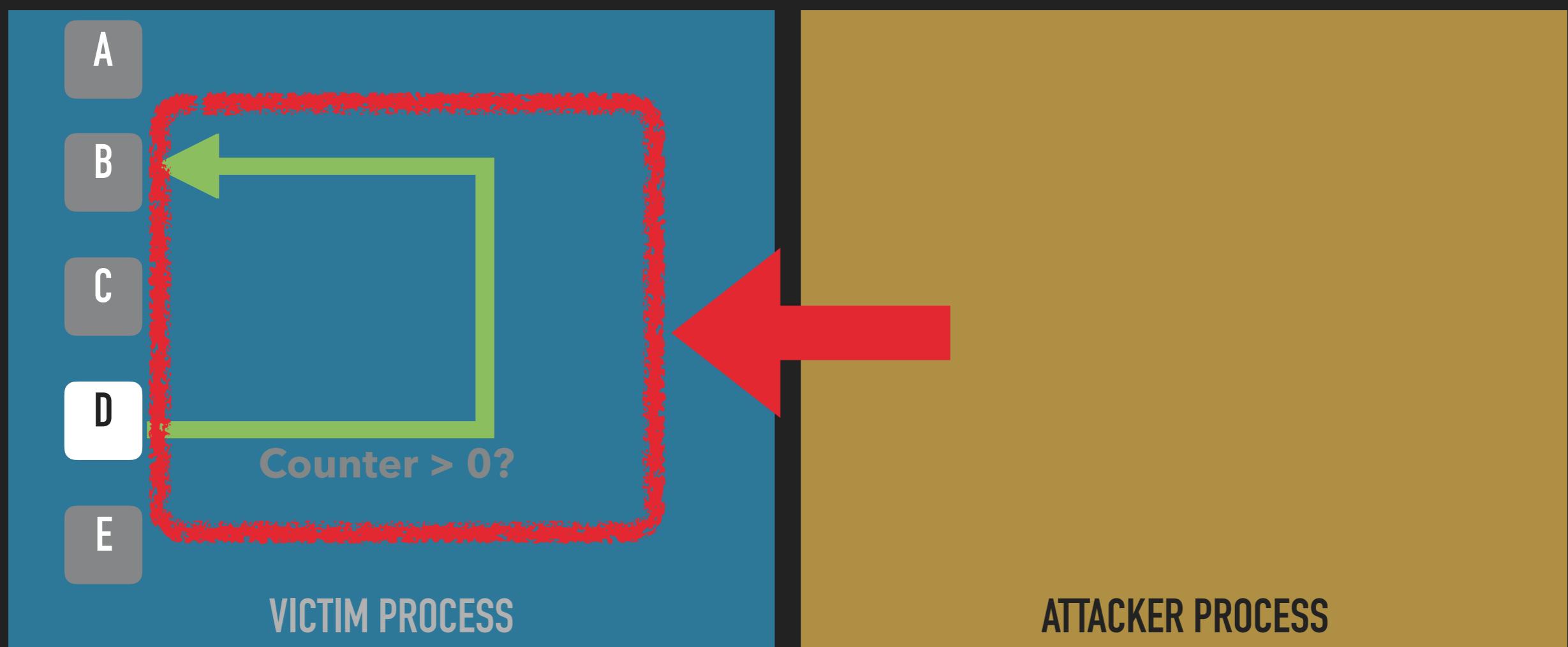
SPECULATIVE
EXECUTION

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

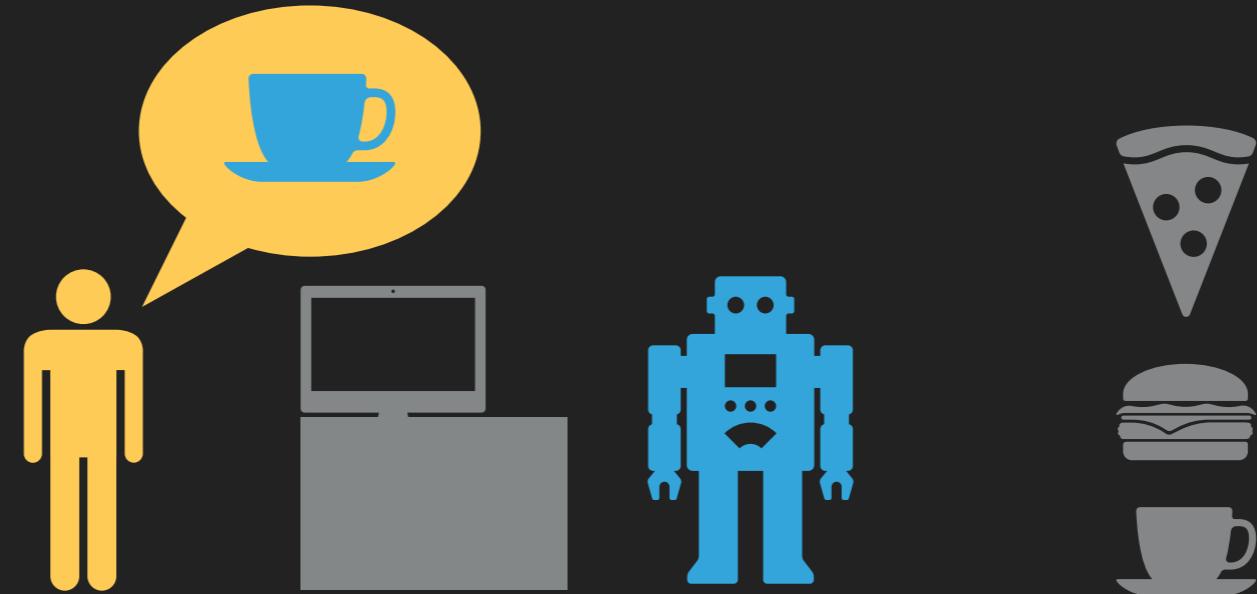
SPECTRE



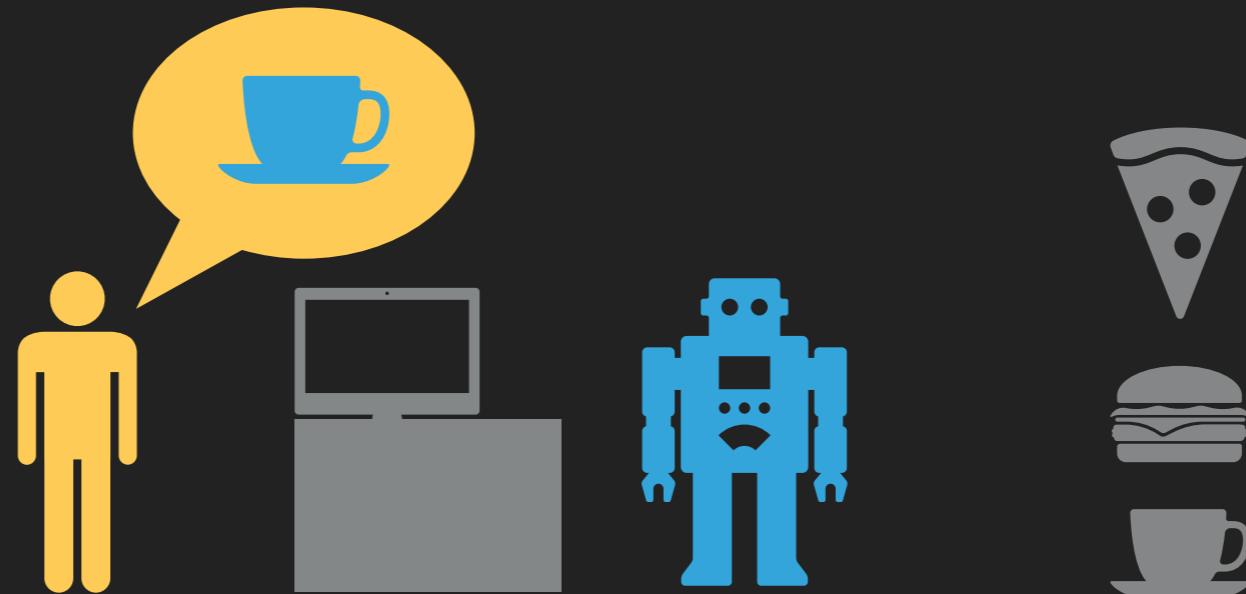
Spectre works like this:

- 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



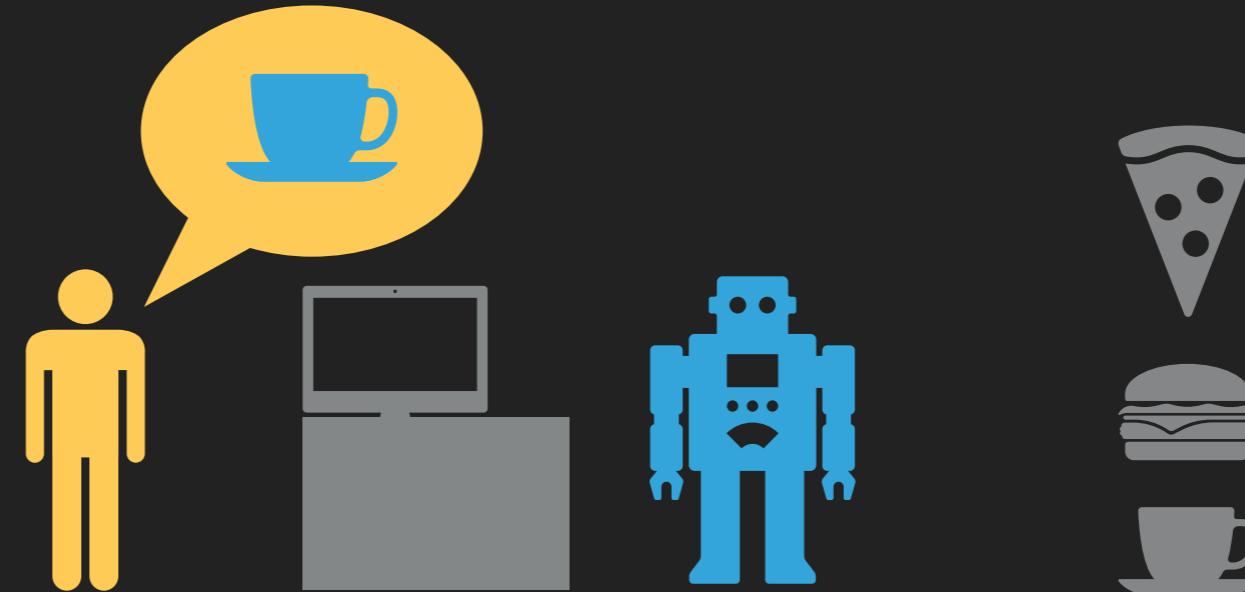
SPECTRE: BRANCH PREDICTION



Monday

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

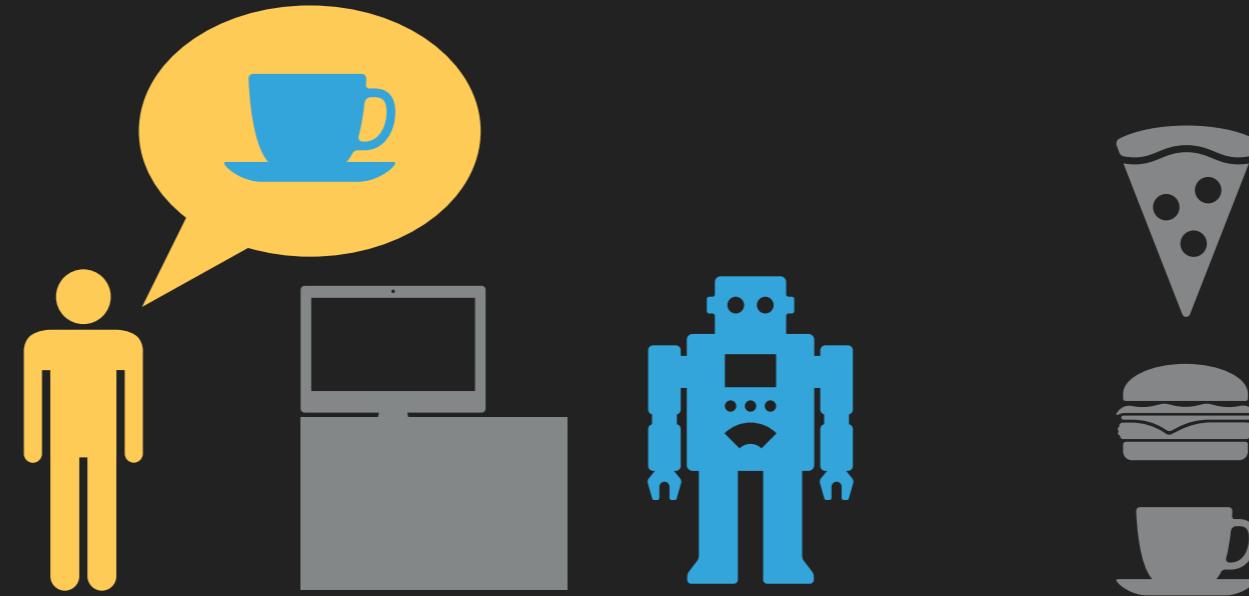
SPECTRE: BRANCH PREDICTION



 *Monday*

 *Tuesday*

SPECTRE: BRANCH PREDICTION

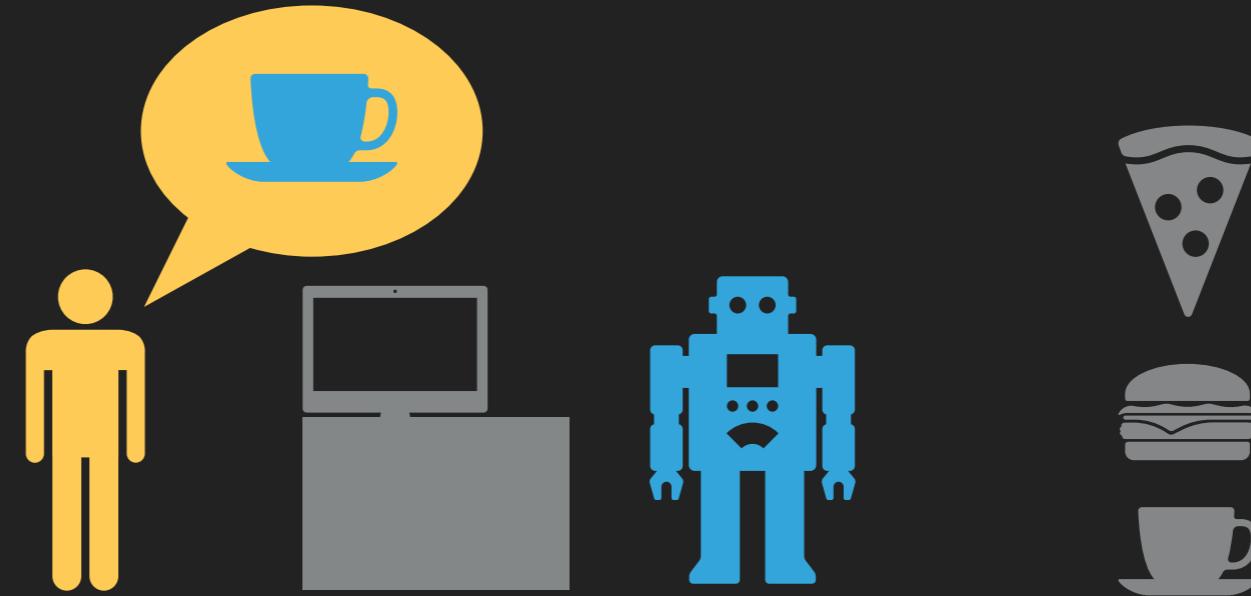


 *Monday*

 *Tuesday*

 *Wednesday*

SPECTRE: BRANCH PREDICTION



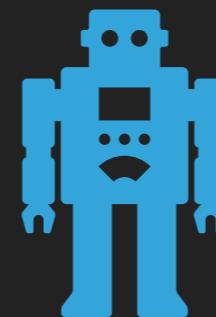
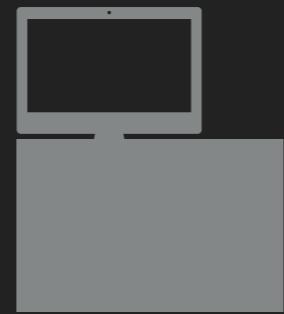
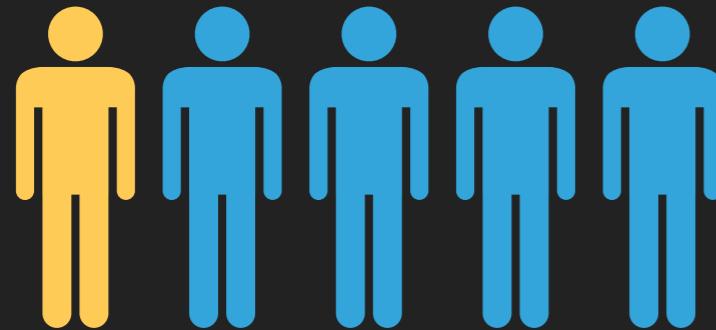
 *Monday*

 *Tuesday*

 *Wednesday*

...

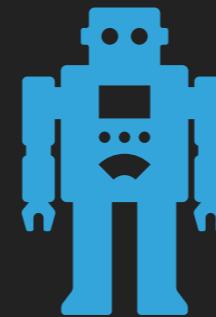
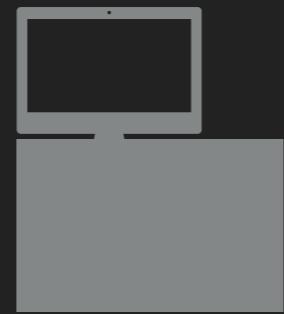
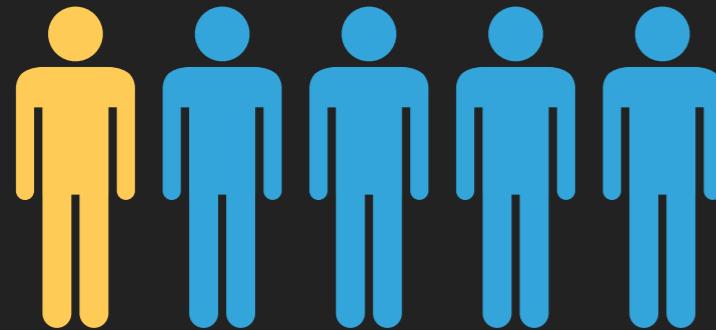
SPECTRE: SPECULATIVE EXECUTION



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

This is very similar to the effect seen in Meltdown.

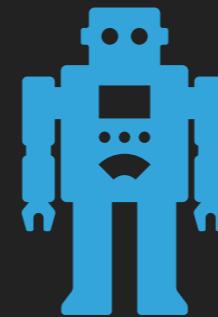
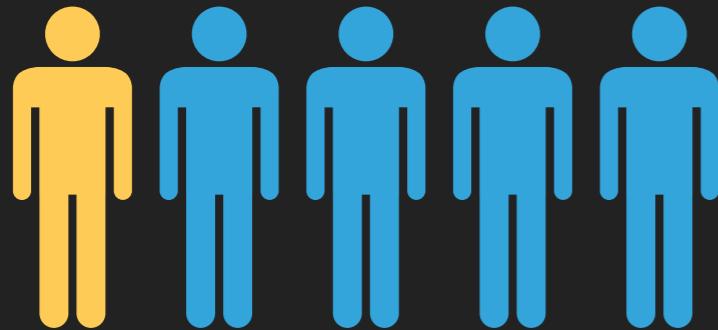
SPECTRE: SPECULATIVE EXECUTION



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

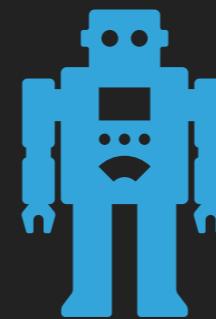
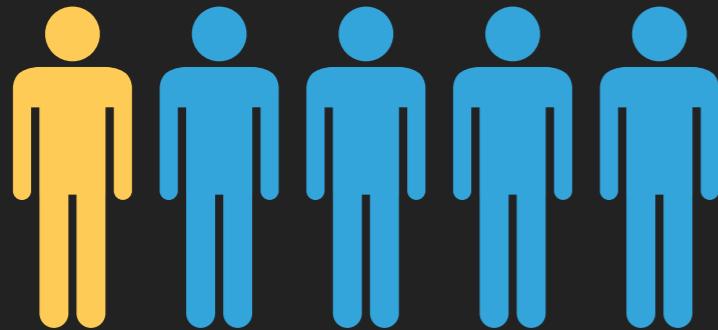


The CPU can improve the coffee machine utilisation by *speculatively* brewing the coffee for 

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

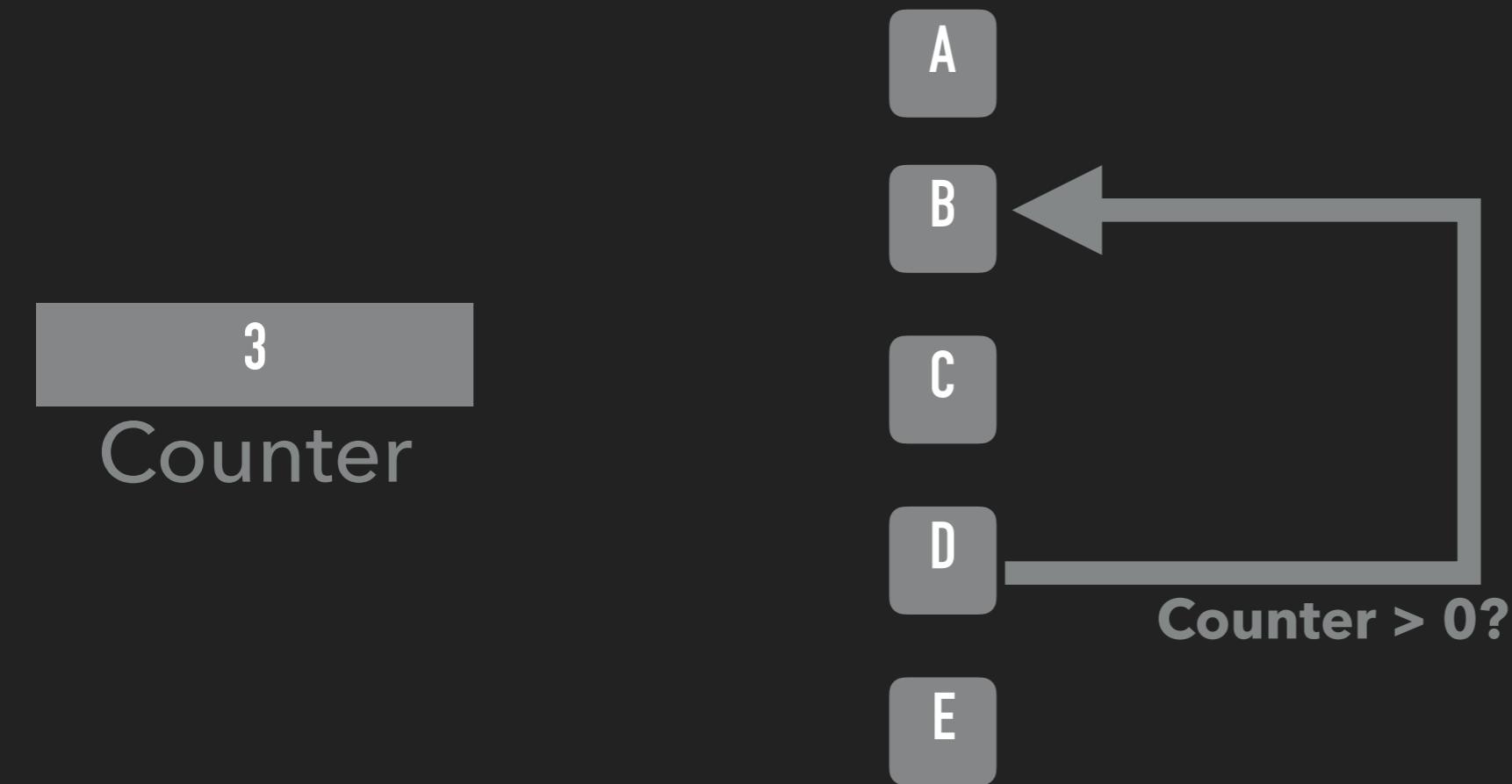


The CPU can improve the coffee machine utilisation by *speculatively* brewing the coffee for 

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

←

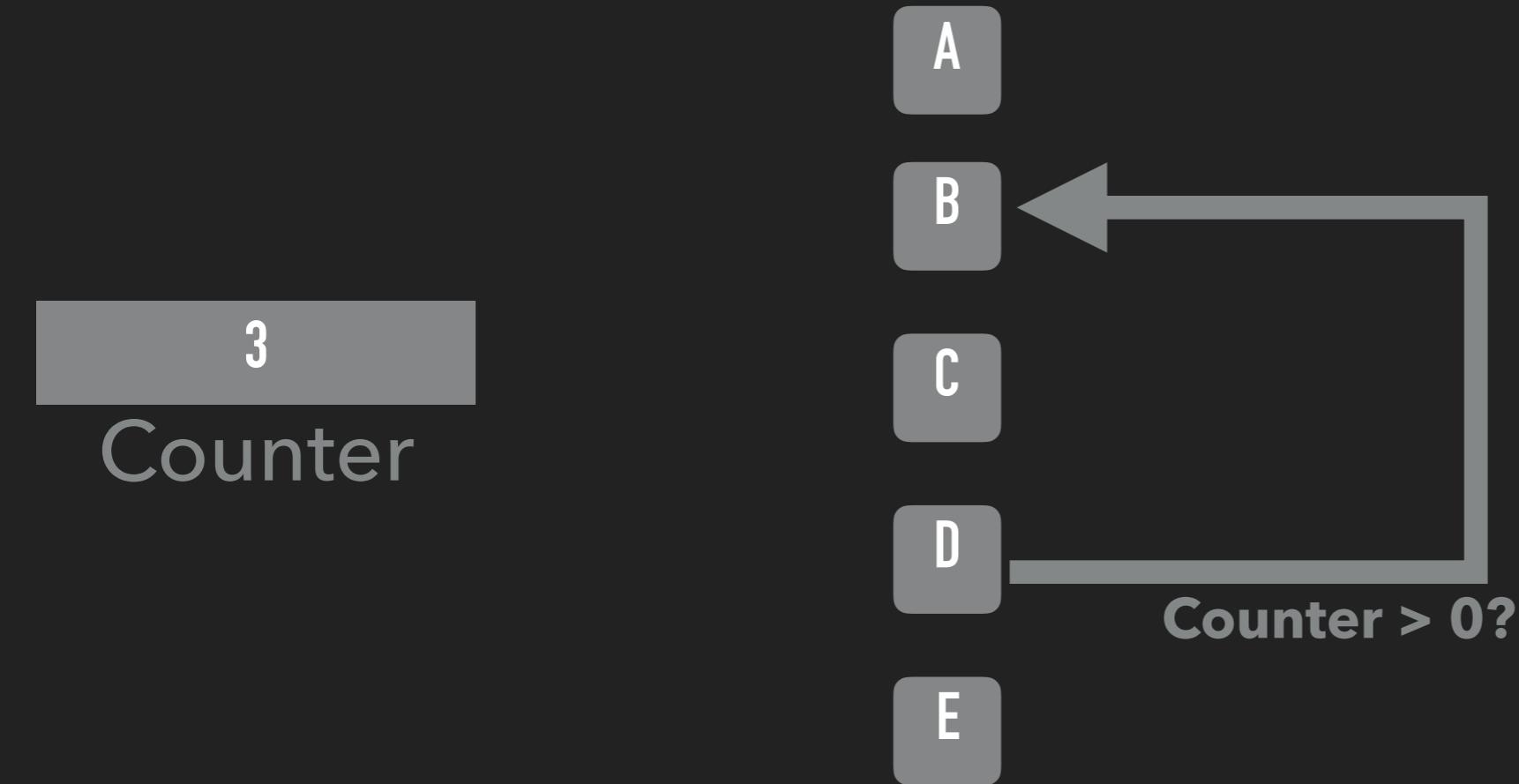
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

Some variable used in branch

A B ... E

Instructions

A

Currently active instruction



Conditional branch

SPECTRE: SPECULATIVE EXECUTION



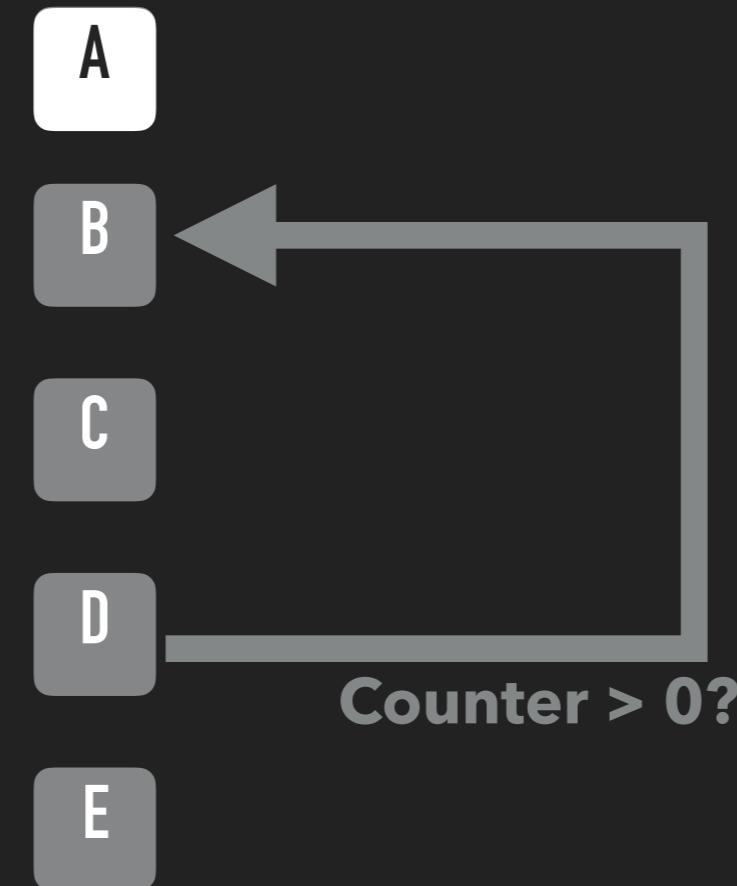
3
Counter

COUNTER

A B ... E

A

←



Some variable used in branch

Instructions

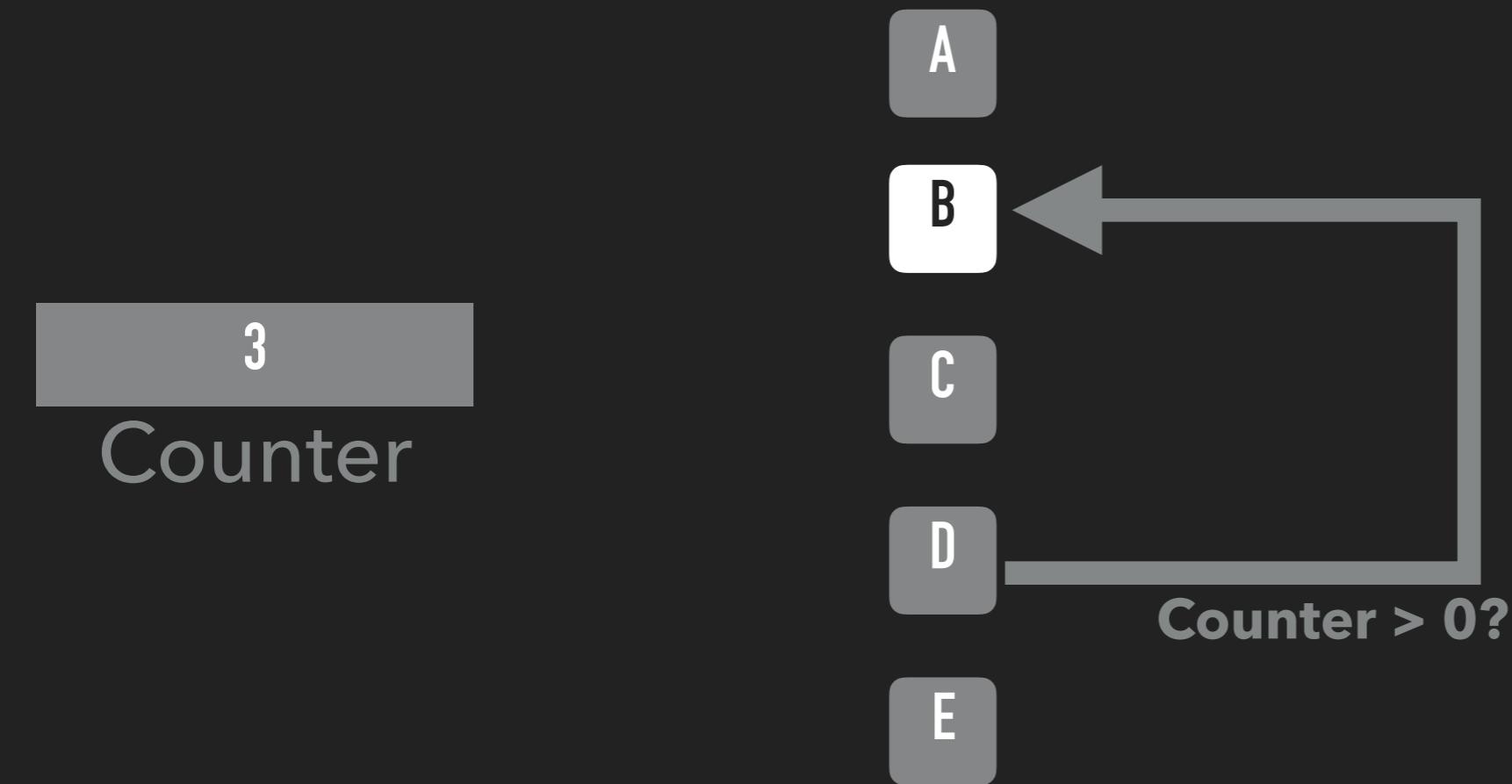
A B ... E

A

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

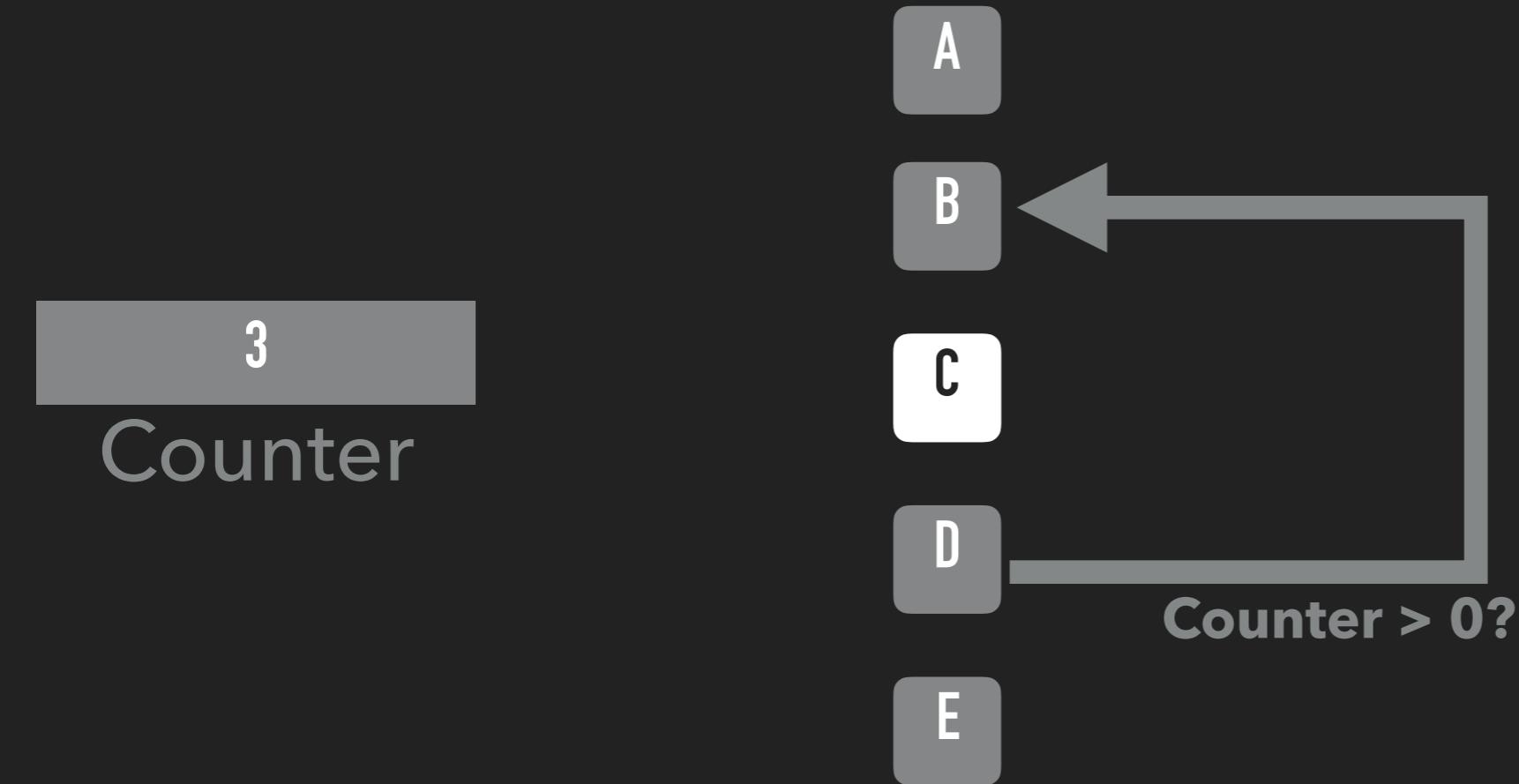
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

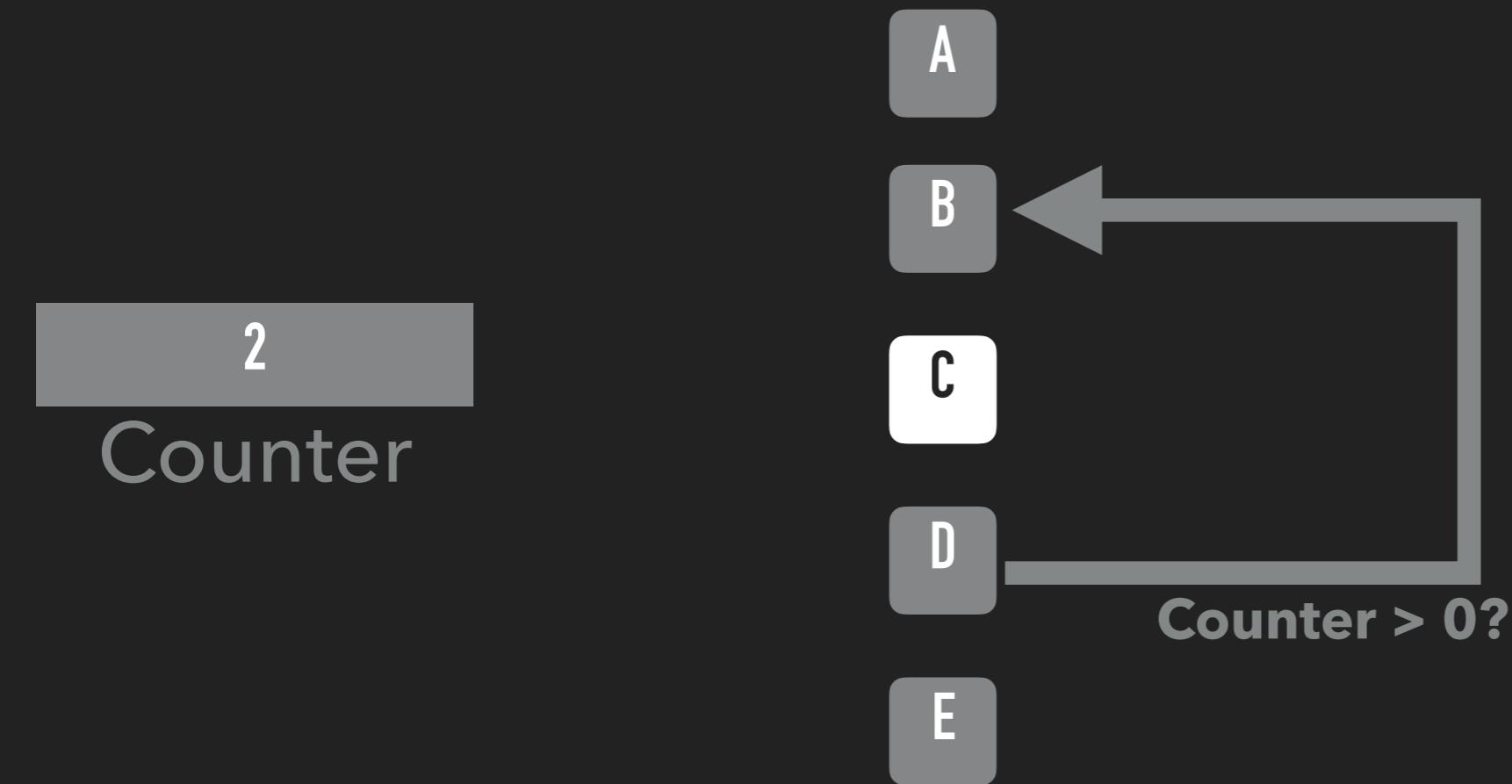
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

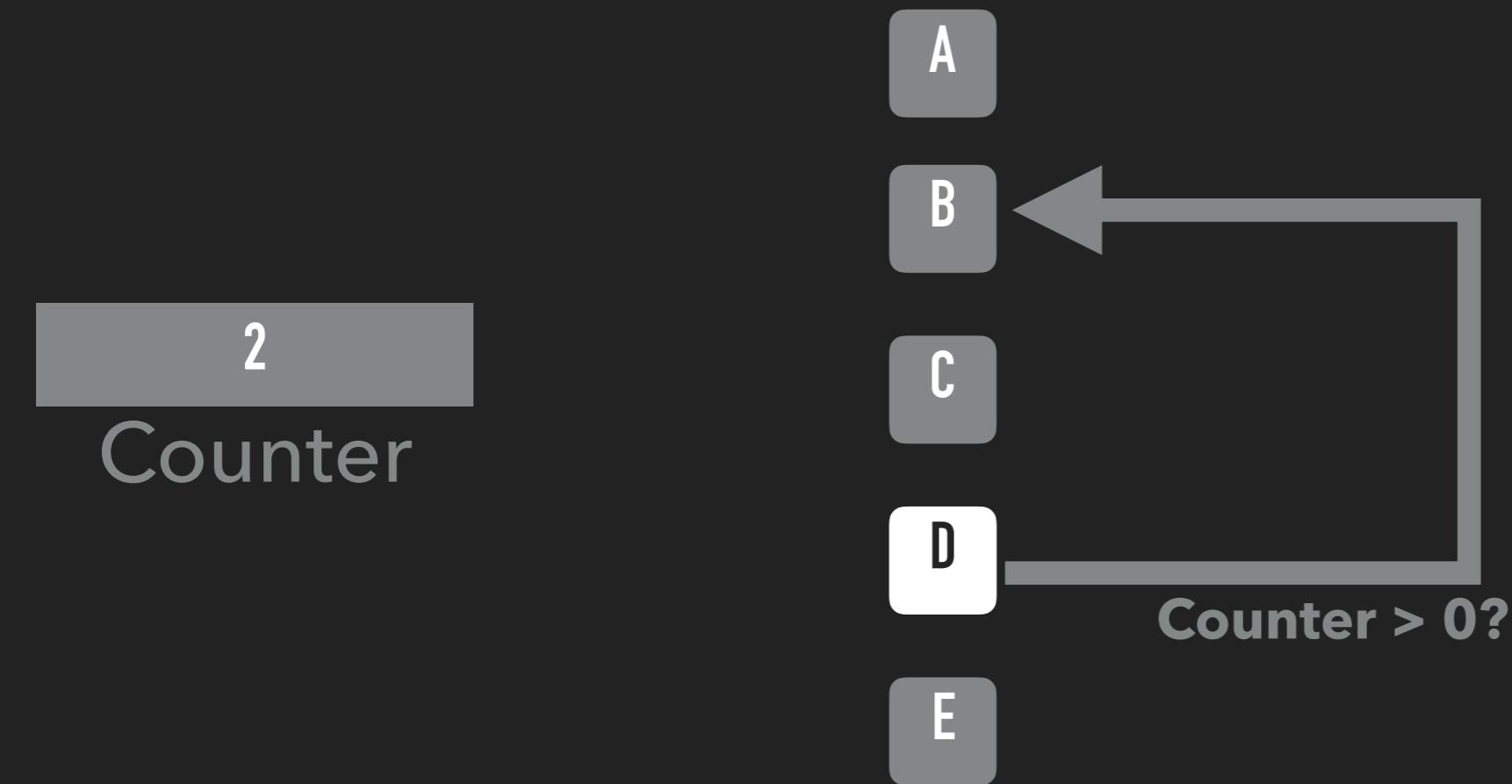
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

Some variable used in branch

A B ... E

Instructions

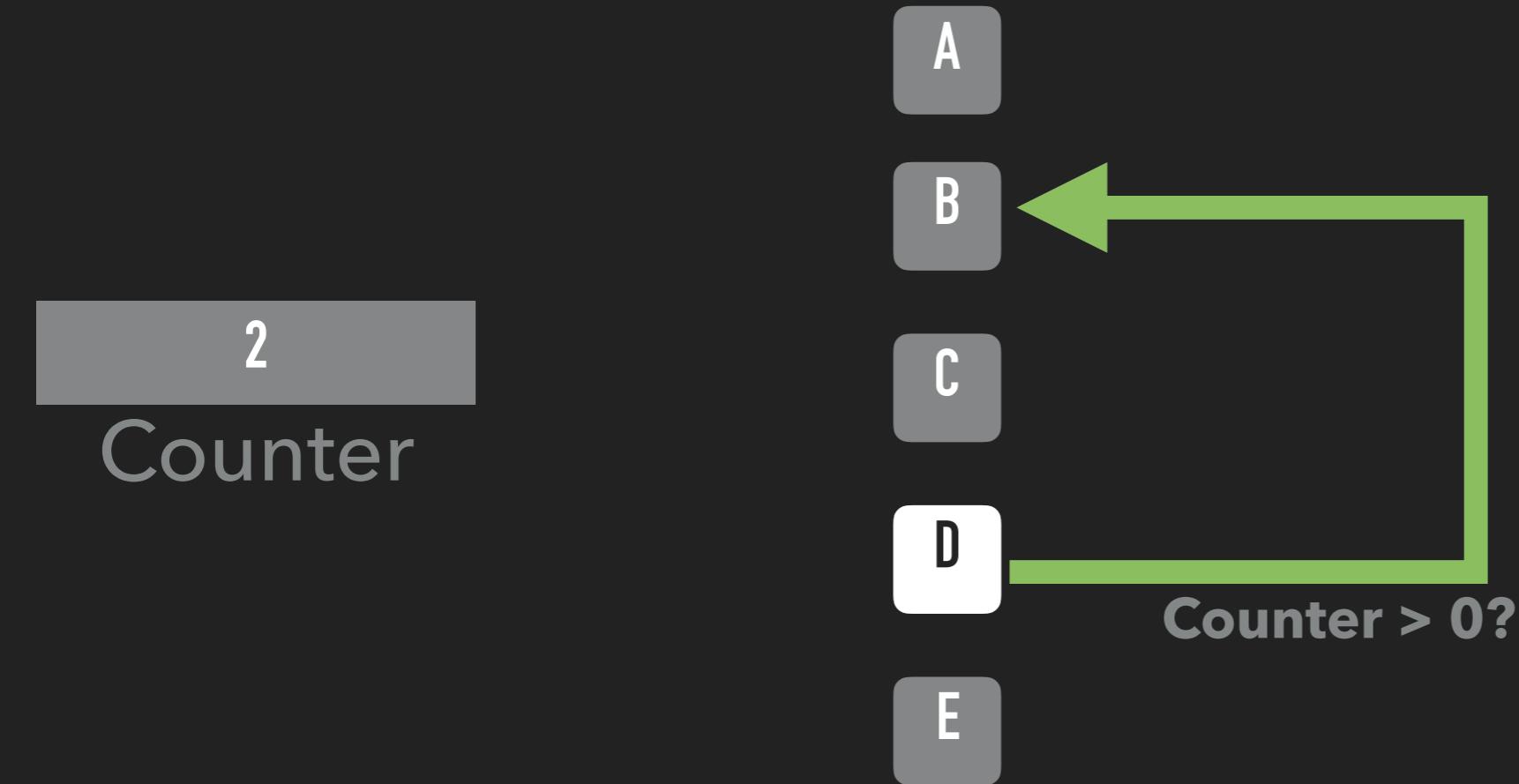
A

Currently active instruction



Conditional branch

SPECTRE: SPECULATIVE EXECUTION



Some variable used in branch

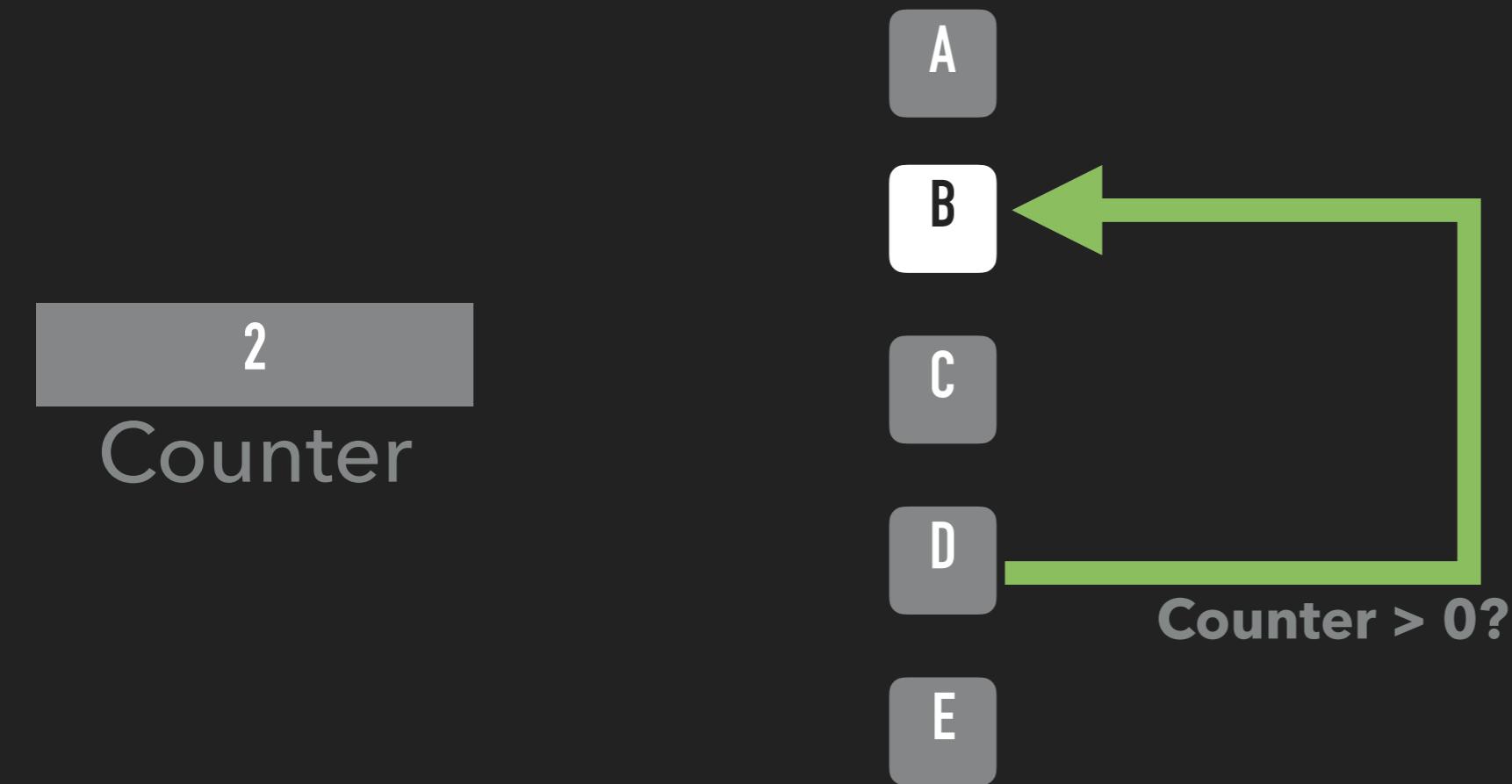
Instructions

A

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



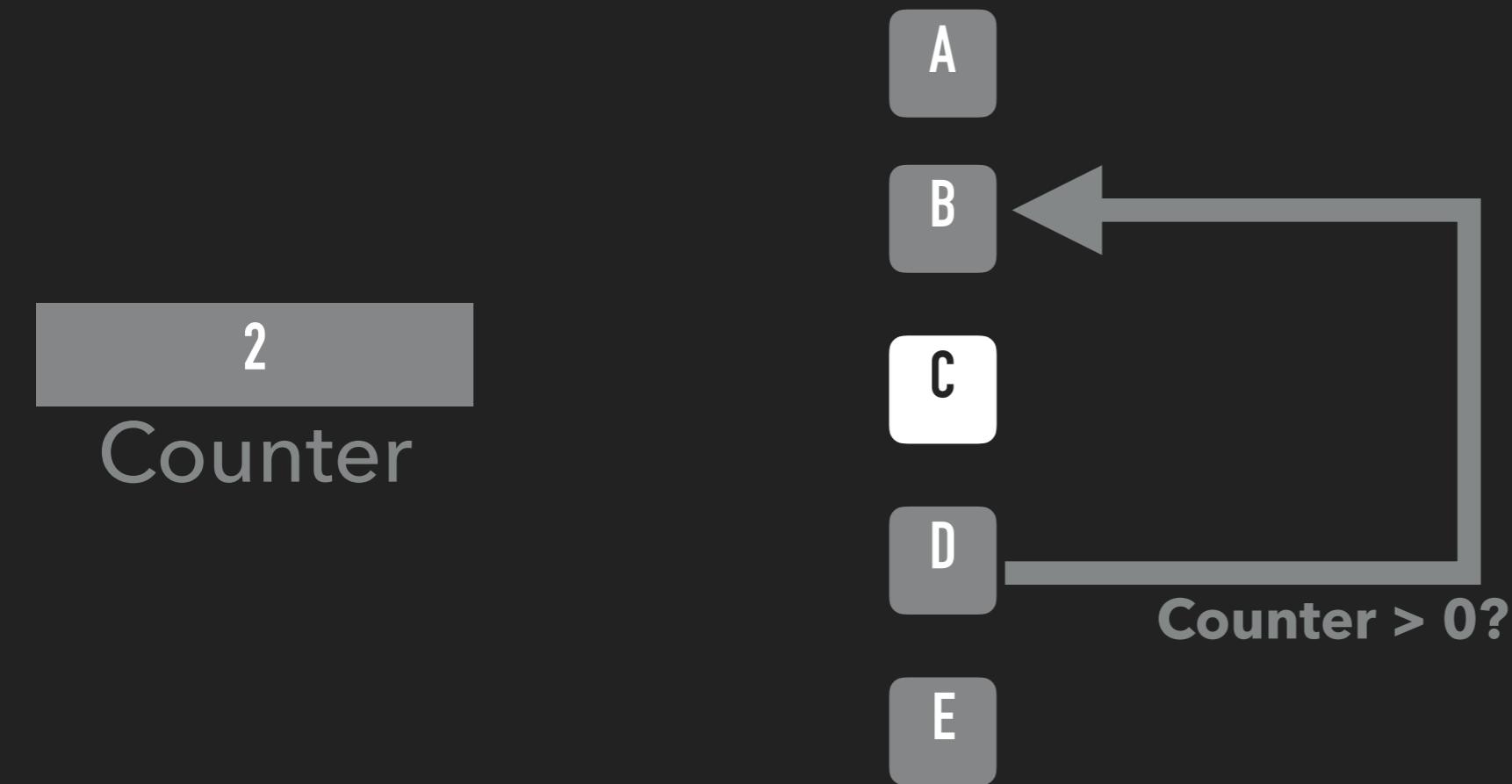
COUNTER Some variable used in branch

A, B, ... E Instructions

A Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

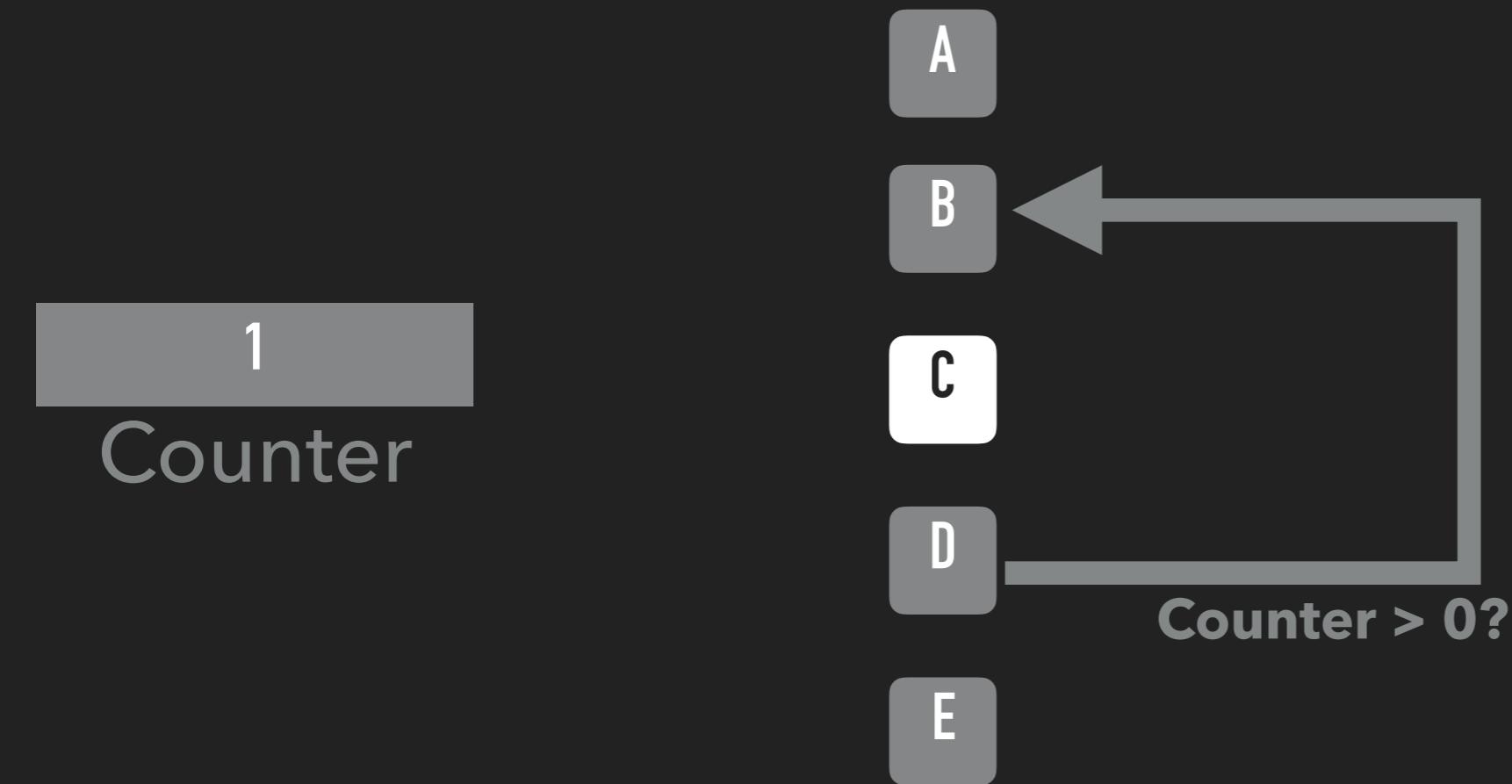
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

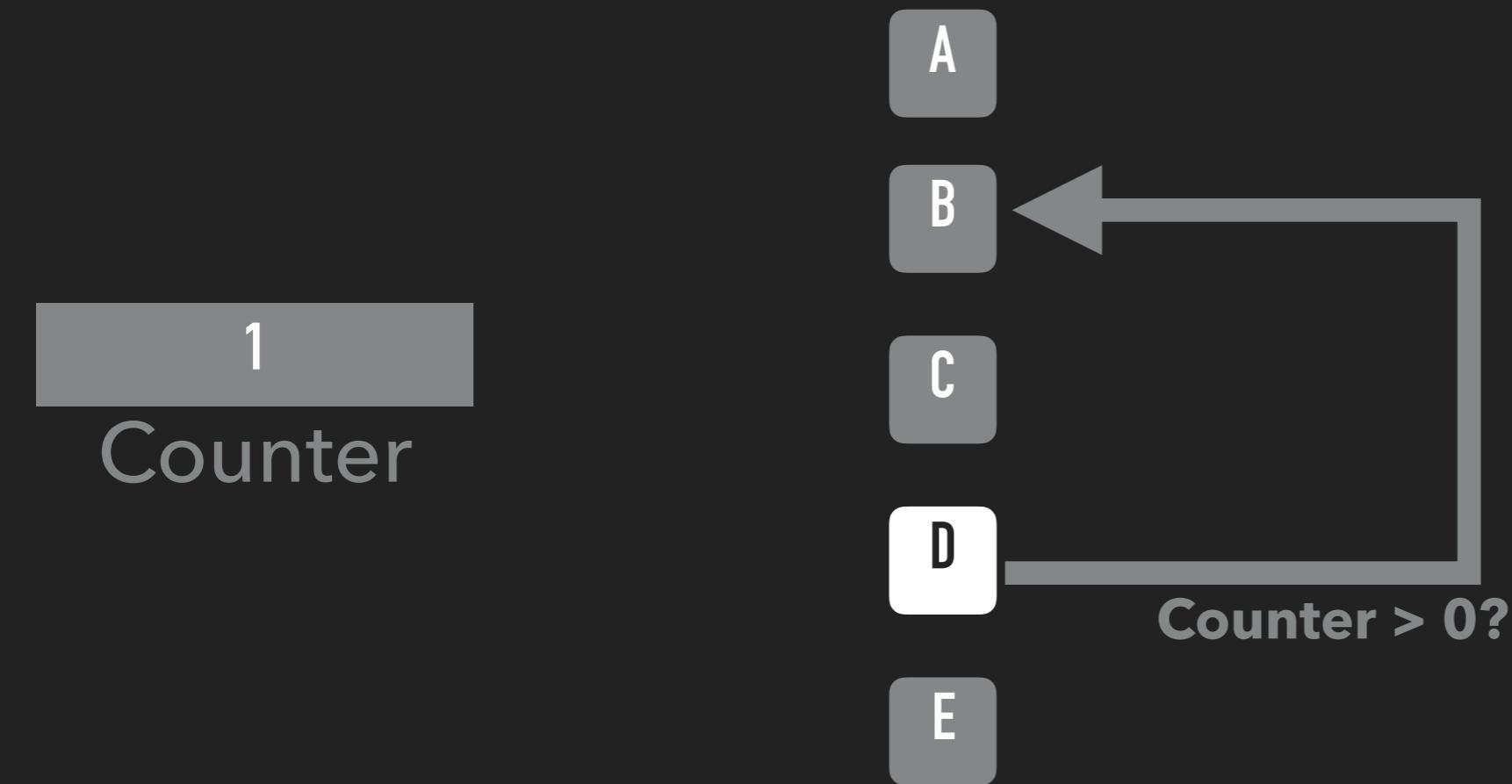
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

Some variable used in branch

Instructions

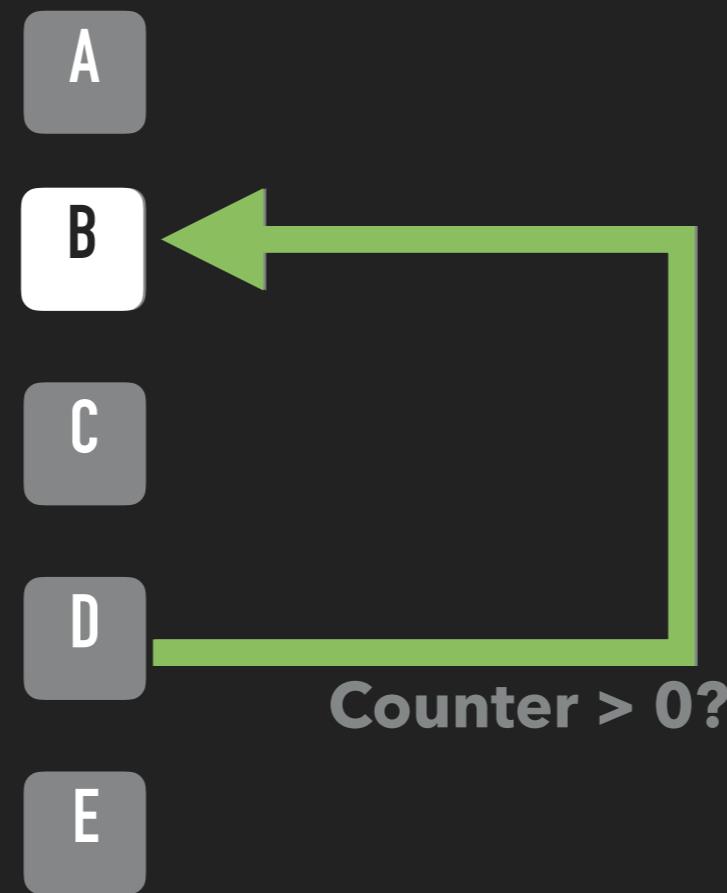
Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



1
Counter



COUNTER

A B ... E

A



Some variable used in branch

Instructions

A

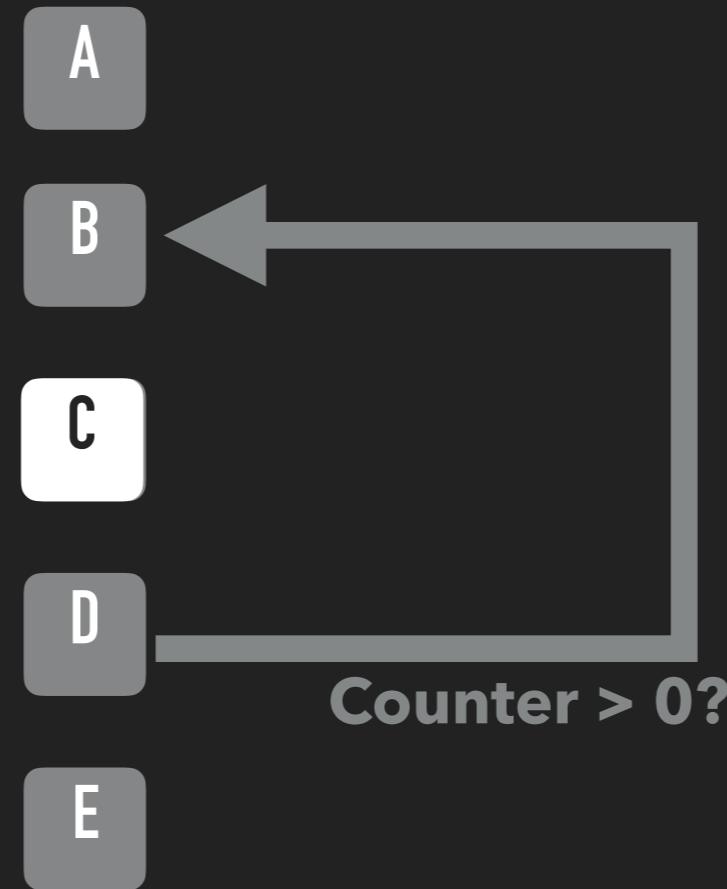
Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



1
Counter



COUNTER

A B ... E

A



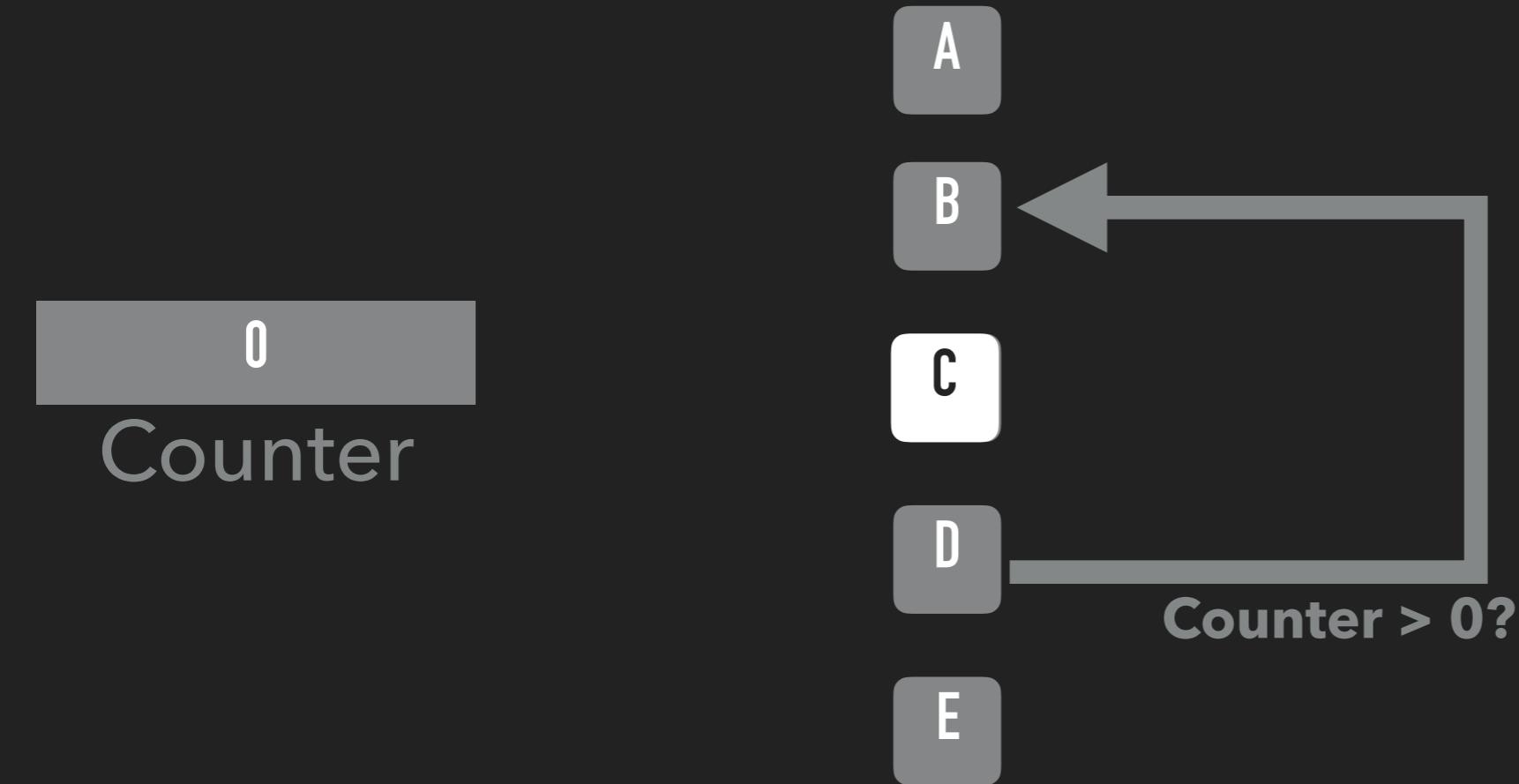
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER

A B ... E

A

←

Some variable used in branch

Instructions

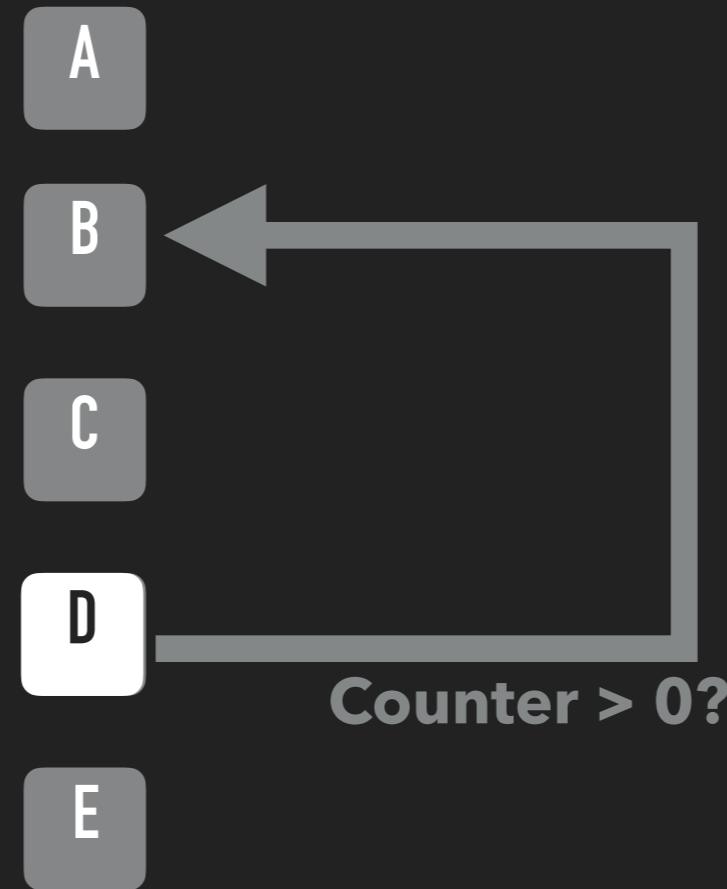
Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



0
Counter



COUNTER

A B ... E

A

←

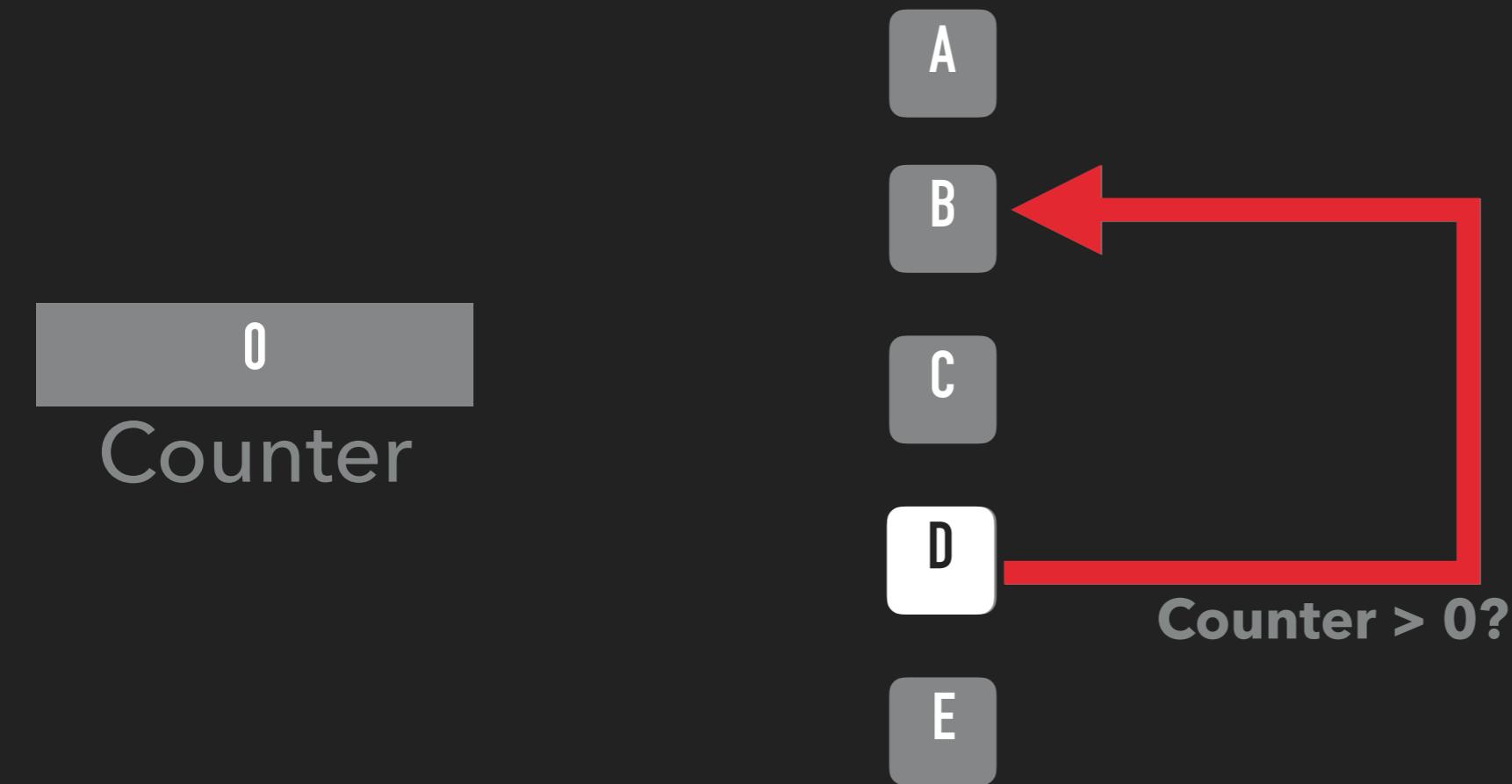
Some variable used in branch

Instructions

Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



COUNTER Some variable used in branch

Instructions

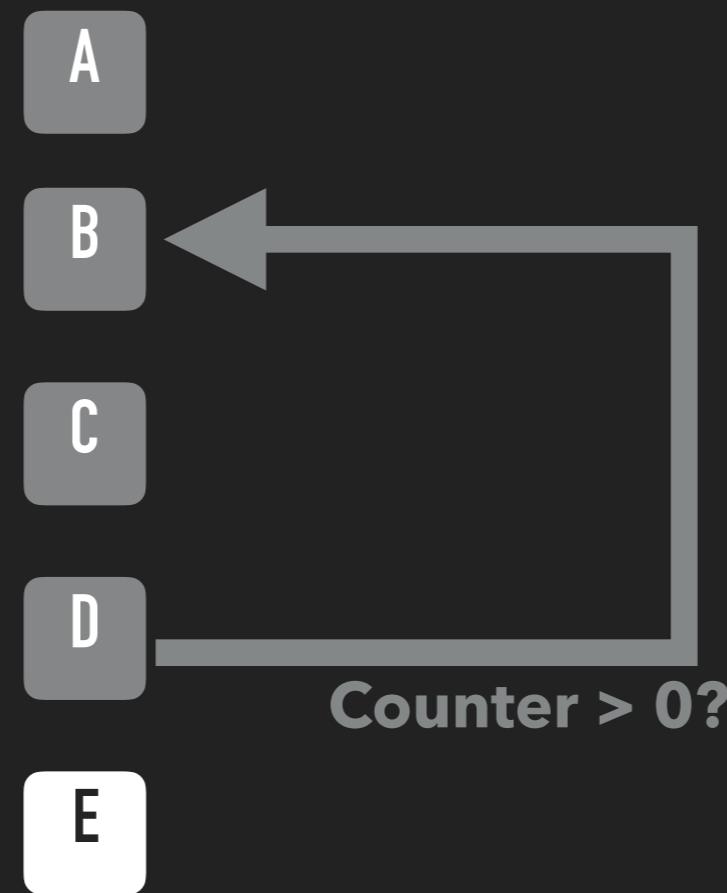
Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION



0
Counter



COUNTER

A B ... E

A

←

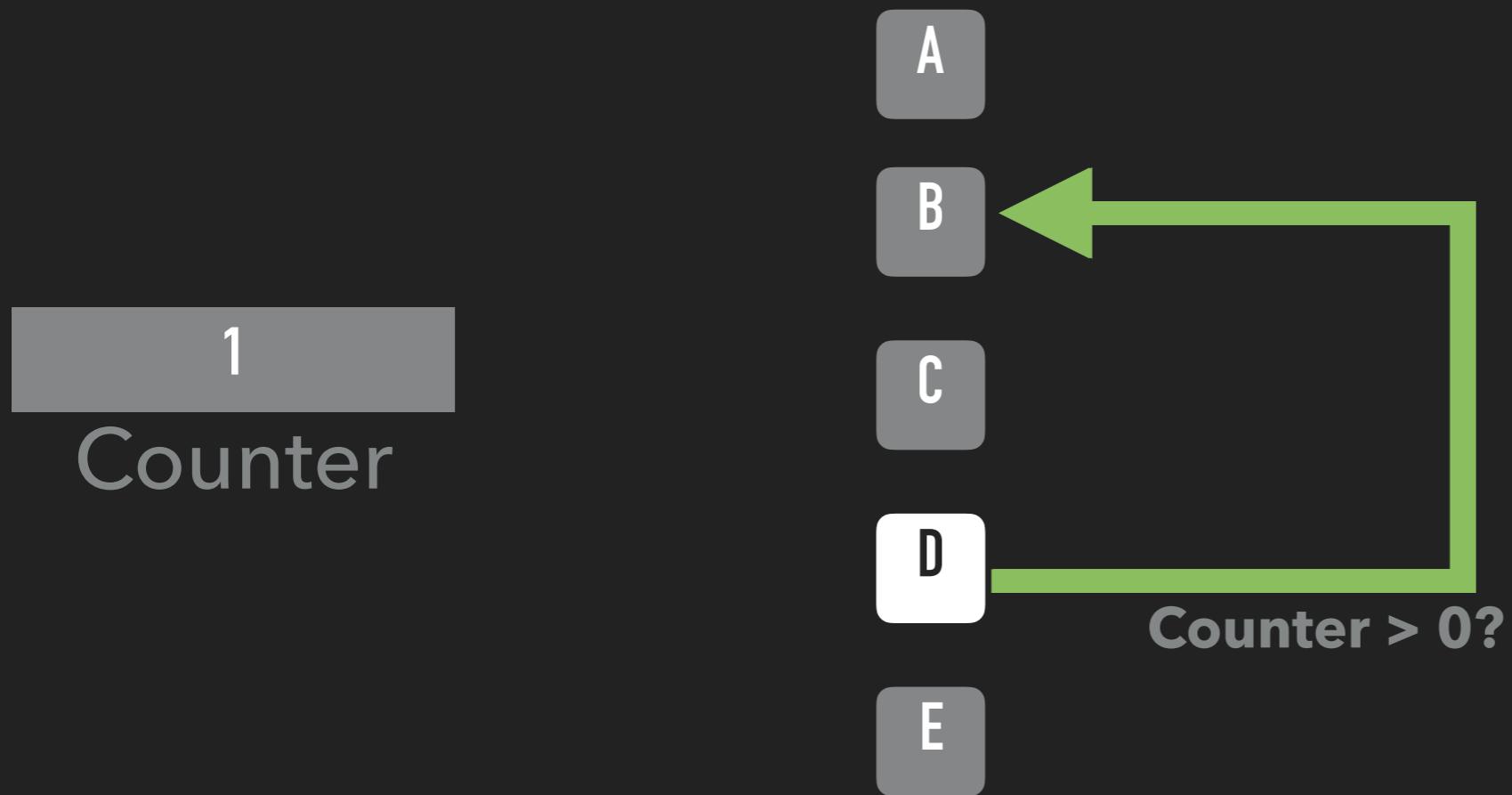
Some variable used in branch

Instructions

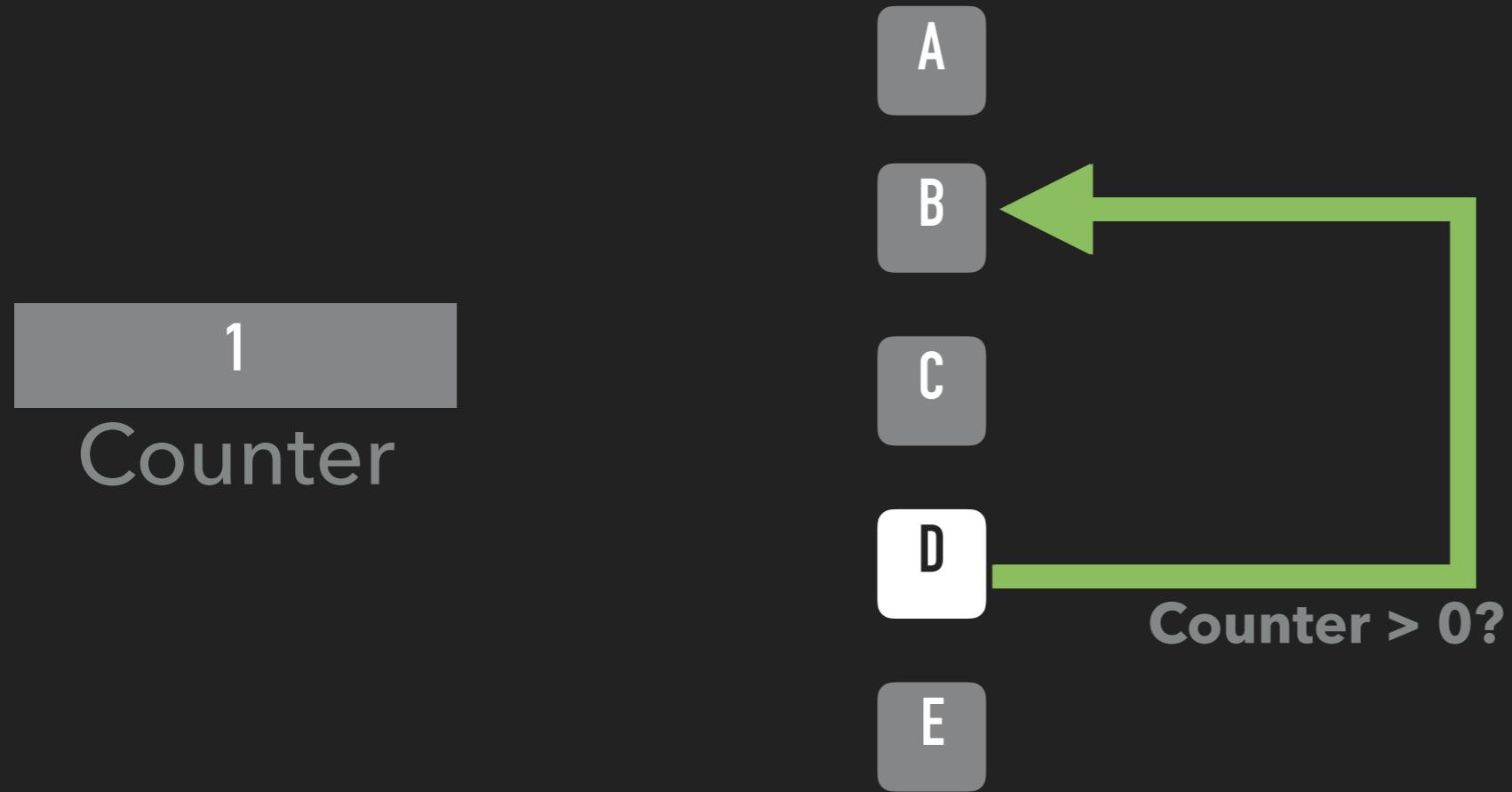
Currently active instruction

Conditional branch

SPECTRE: SPECULATIVE EXECUTION

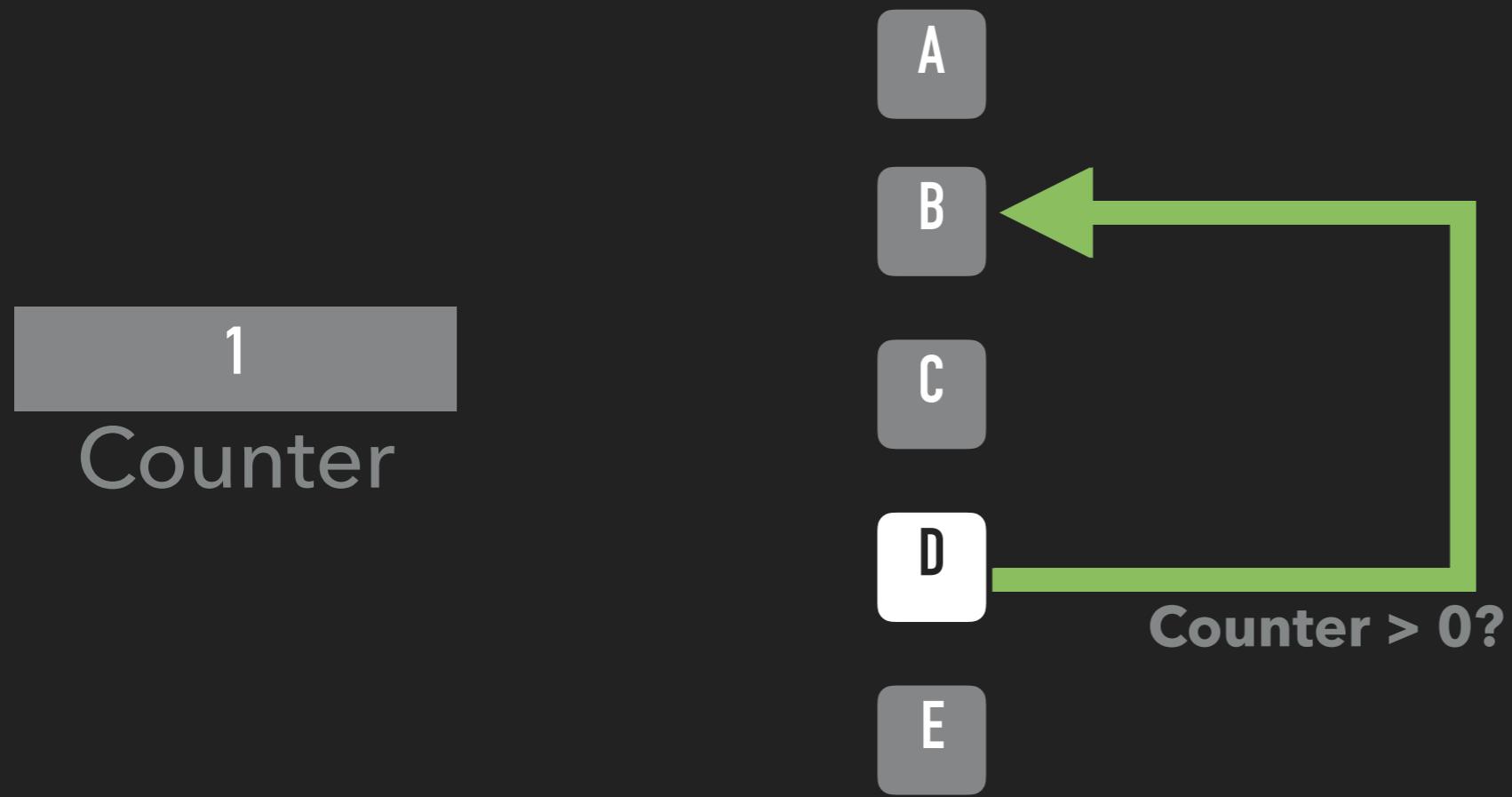


SPECTRE: SPECULATIVE EXECUTION



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

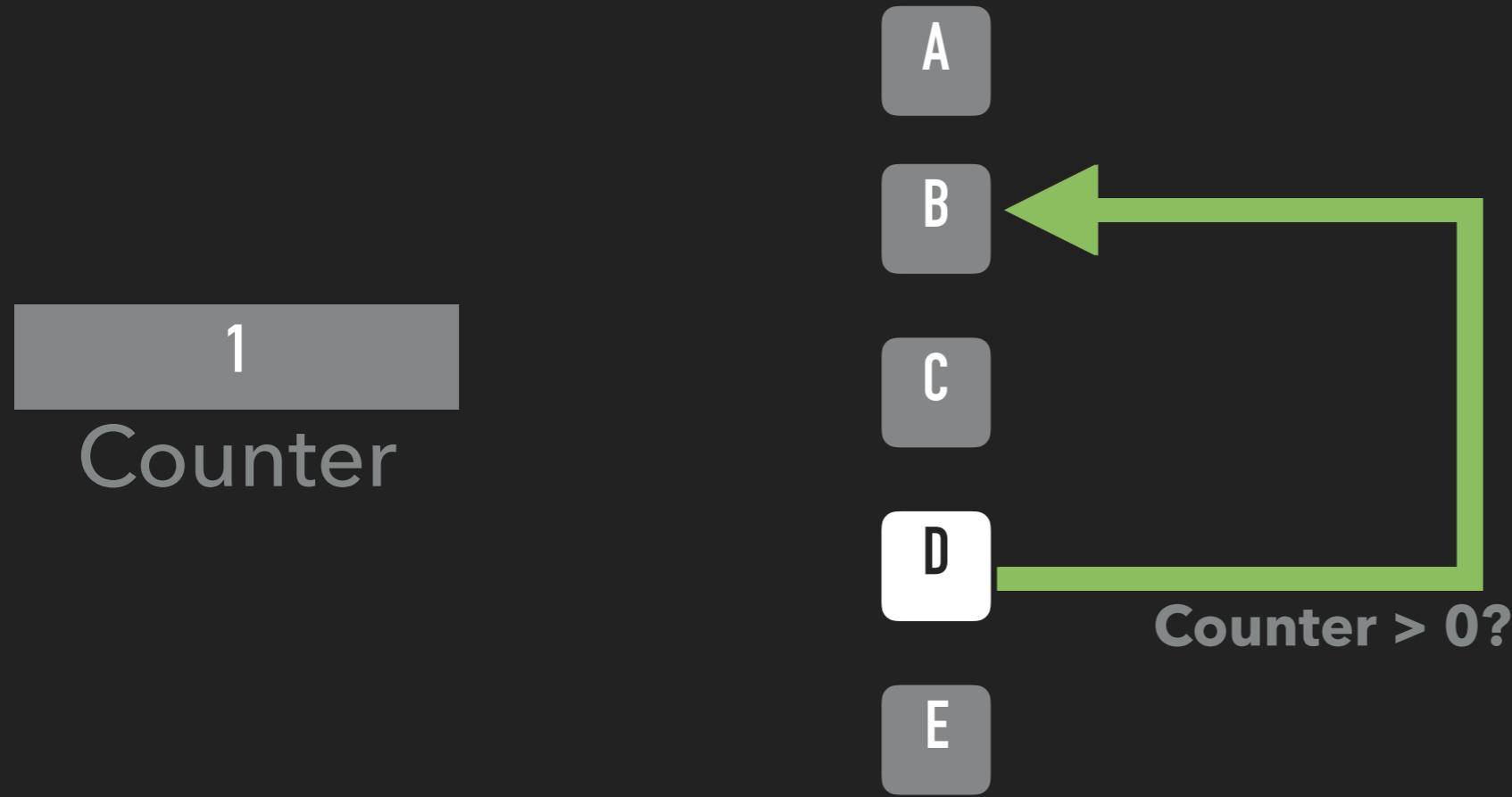
SPECTRE: SPECULATIVE EXECUTION



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

Reading Counter from memory is very slow

SPECTRE: SPECULATIVE EXECUTION

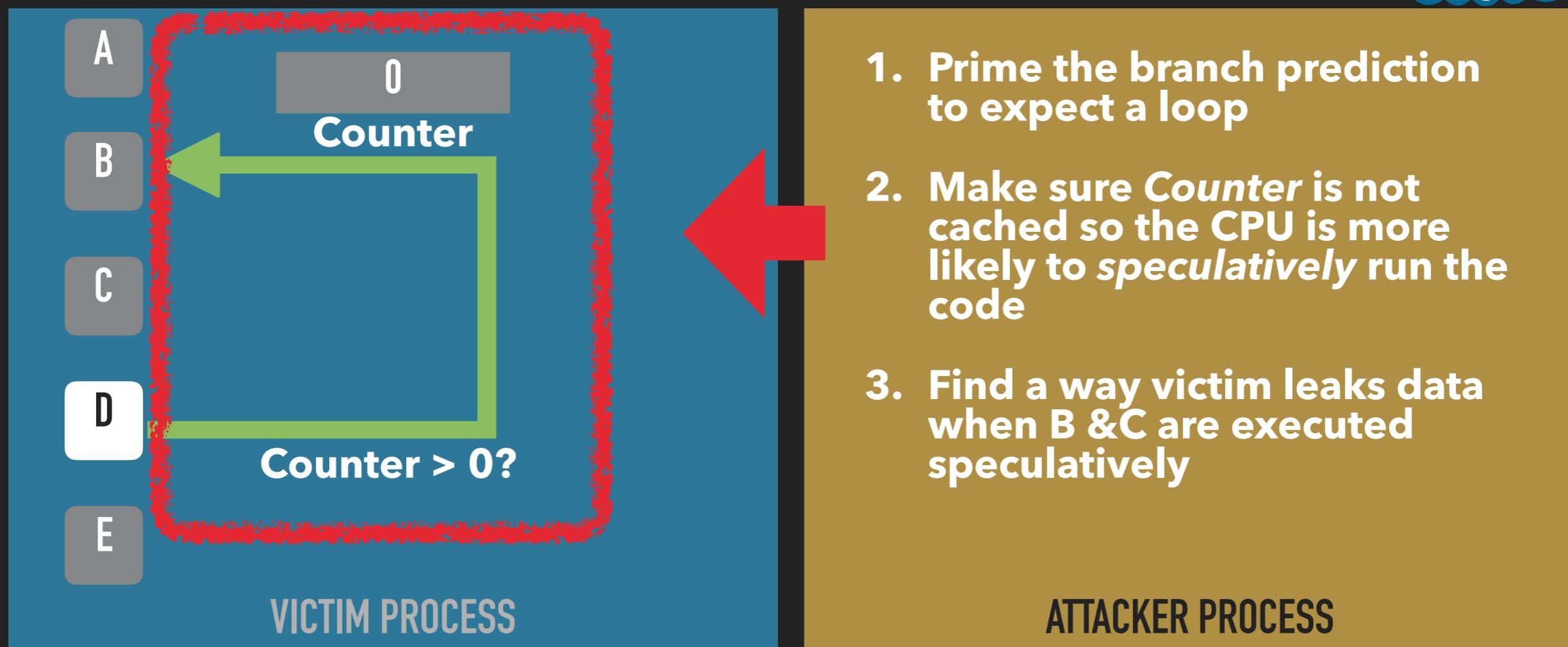


The CPU has learned that Counter *probably* is > 0

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`

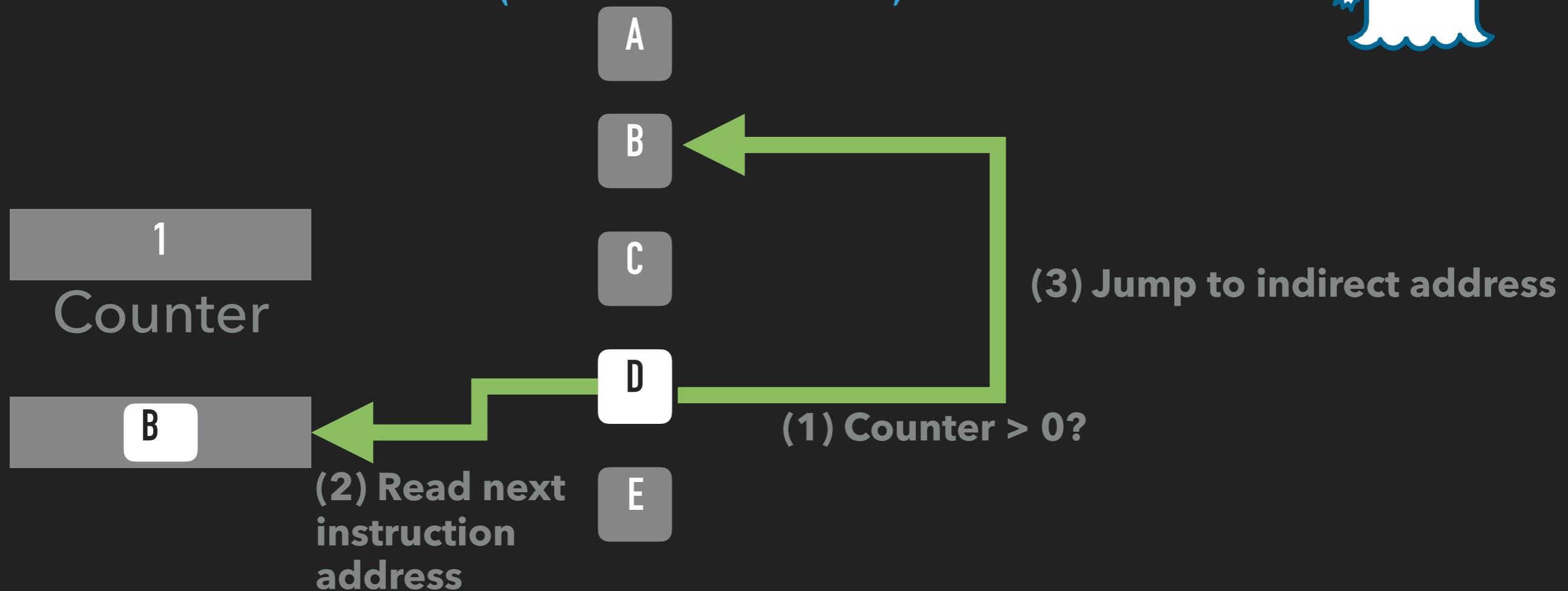
SPECTRE: VARIANT 1 (CVE-2017-5753)



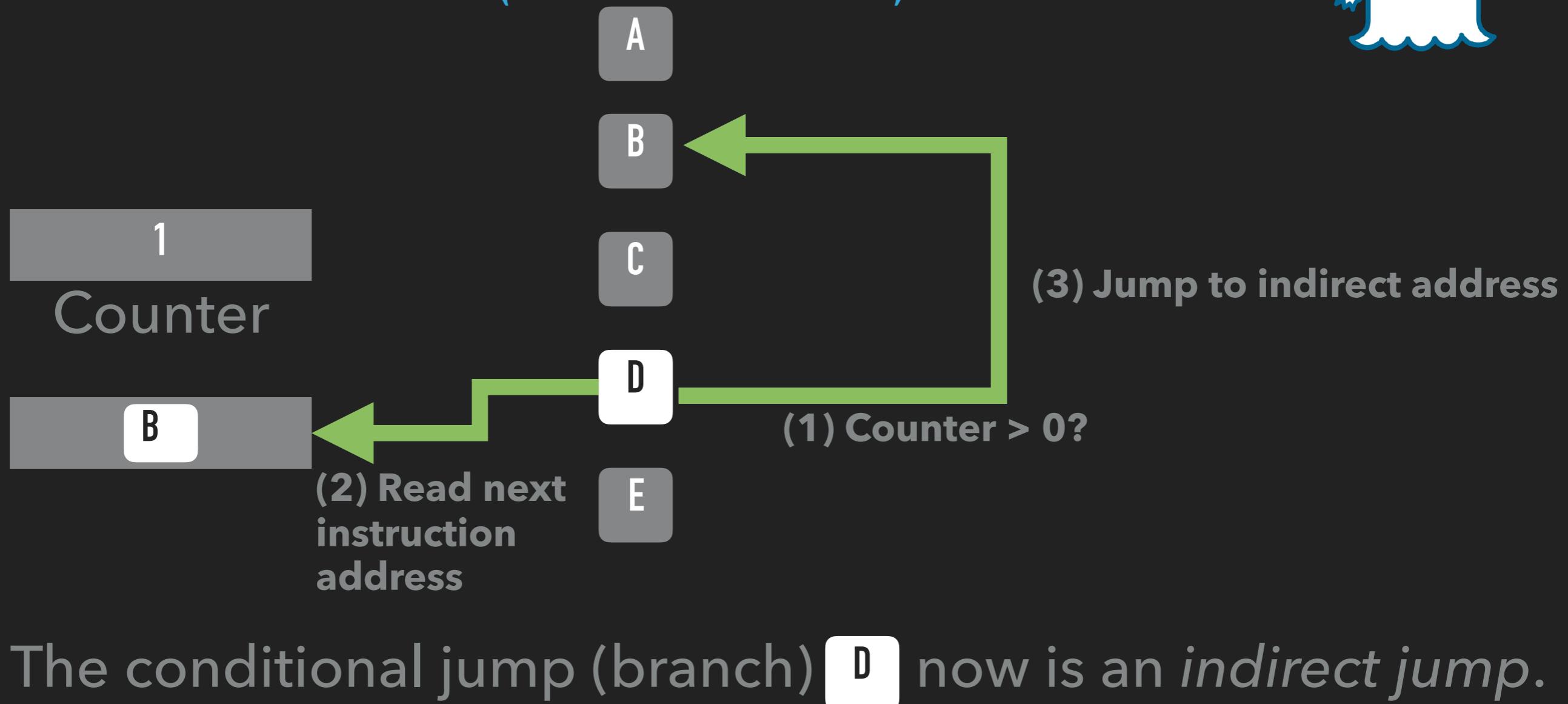
```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- ▶ 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]`

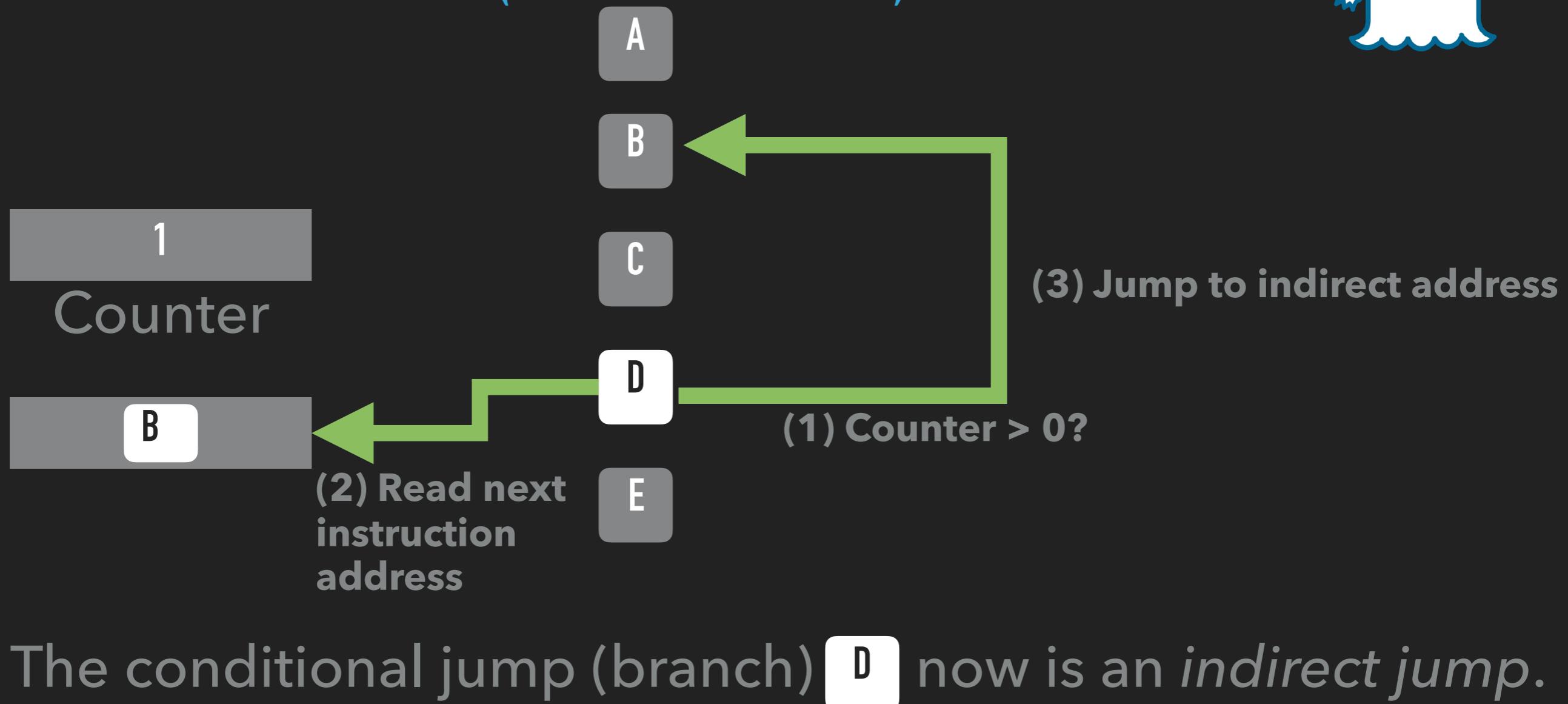
SPECTRE: VARIANT 2 (CVE-2017-5715)



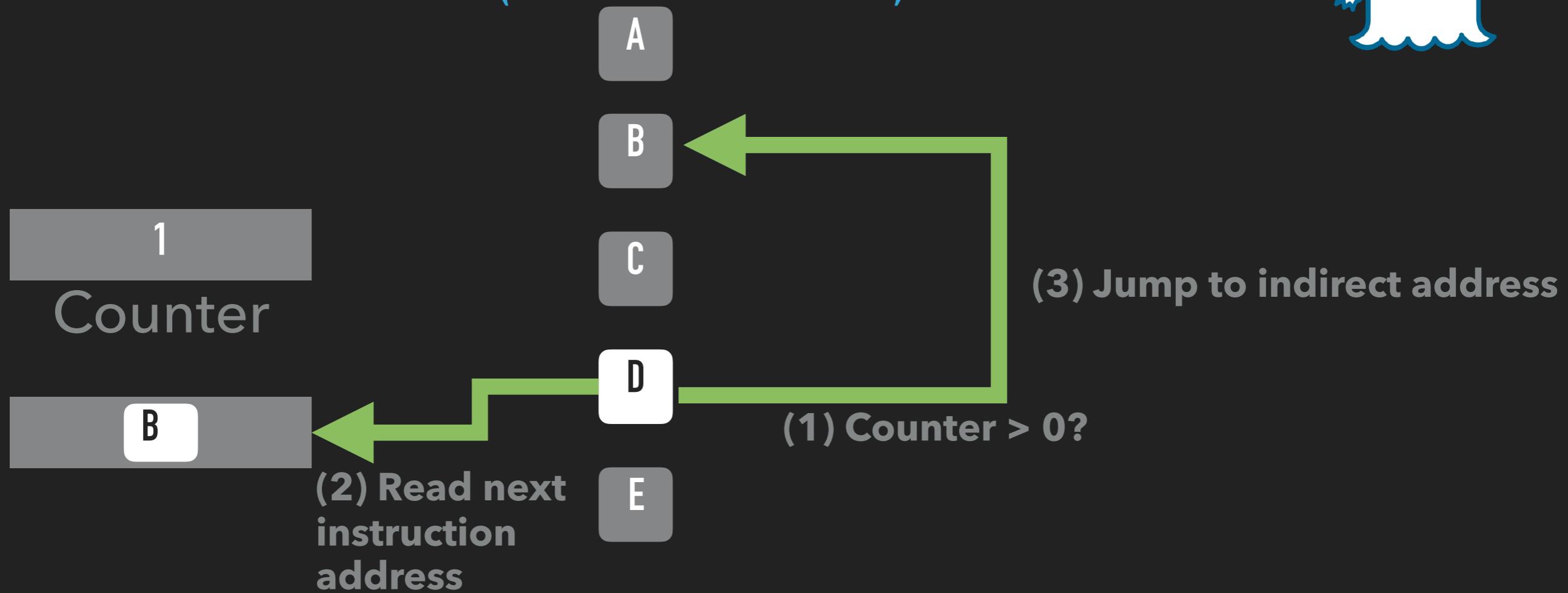
SPECTRE: VARIANT 2 (CVE-2017-5715)



SPECTRE: VARIANT 2 (CVE-2017-5715)

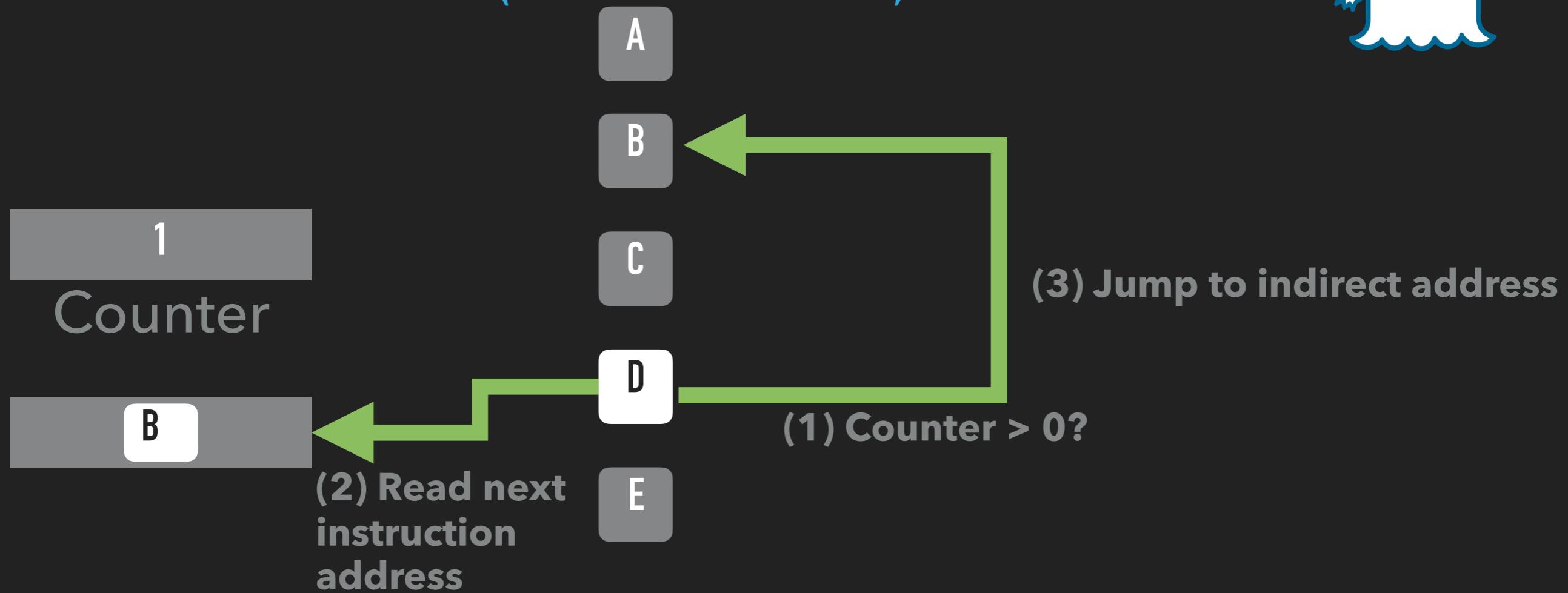


SPECTRE: VARIANT 2 (CVE-2017-5715)



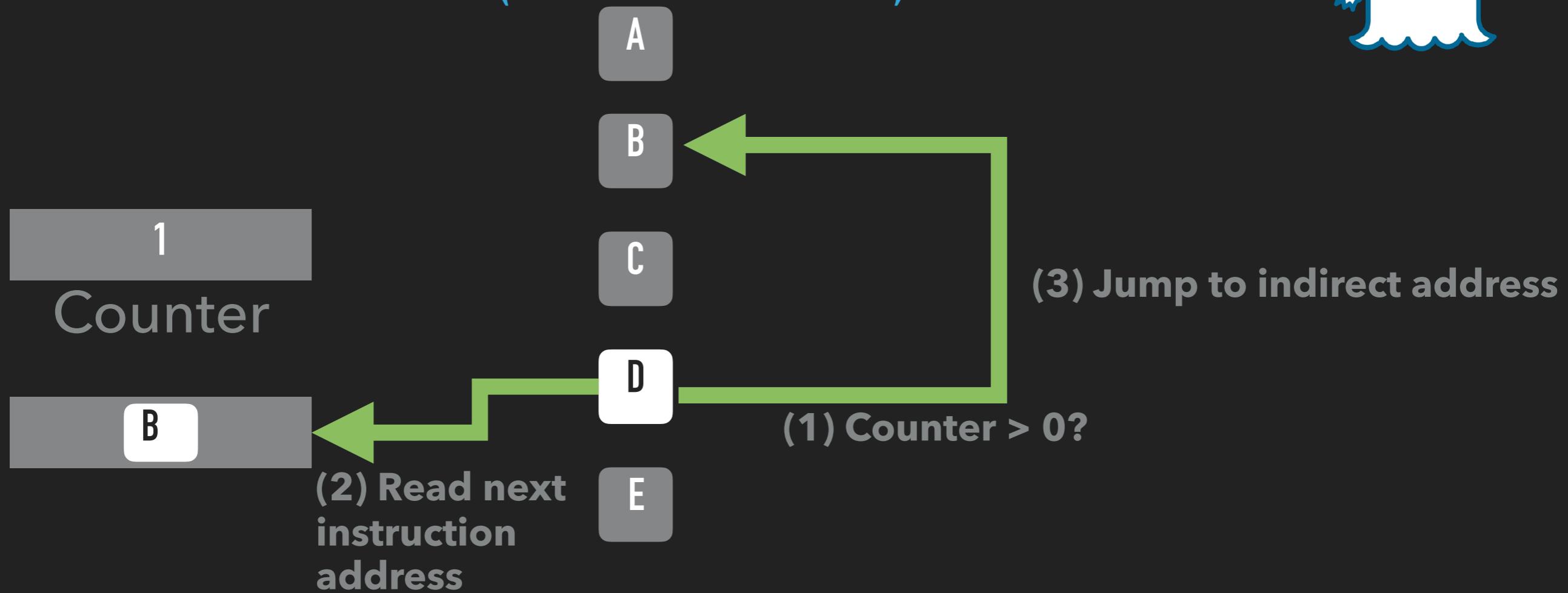
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

SPECTRE: VARIANT 2 (CVE-2017-5715)



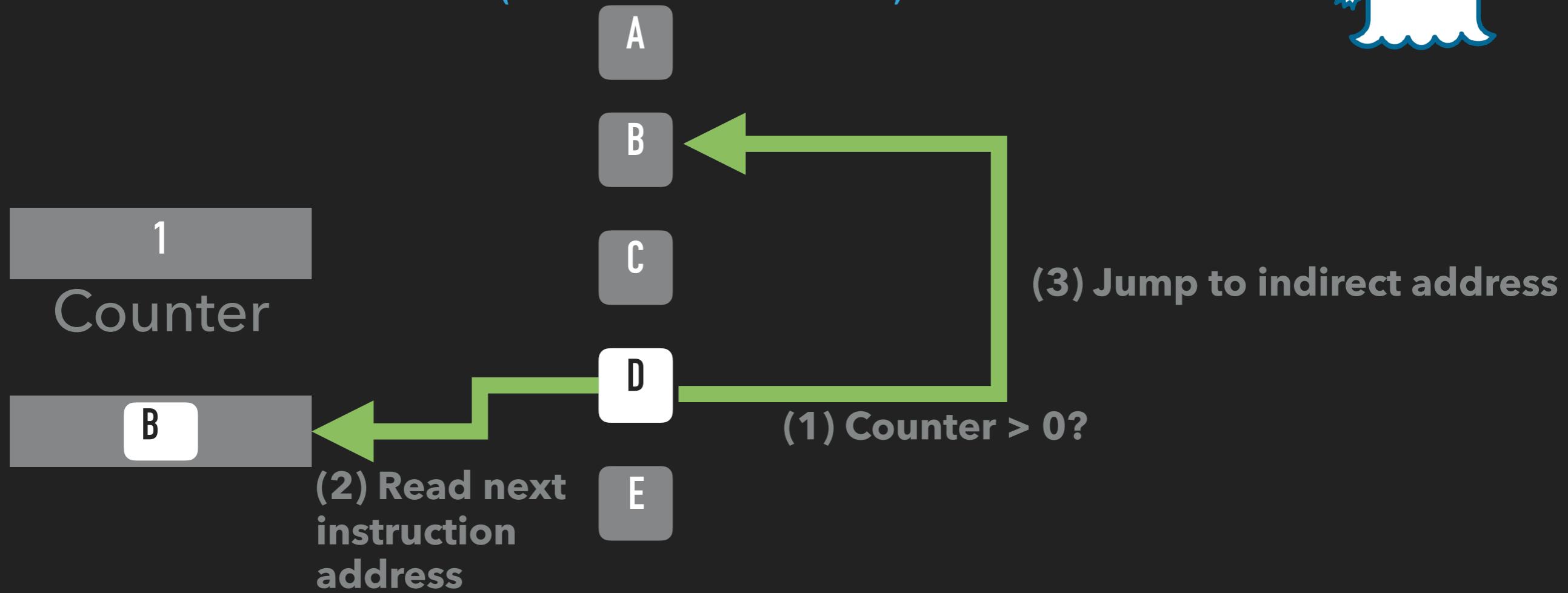
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

SPECTRE: VARIANT 2 (CVE-2017-5715)



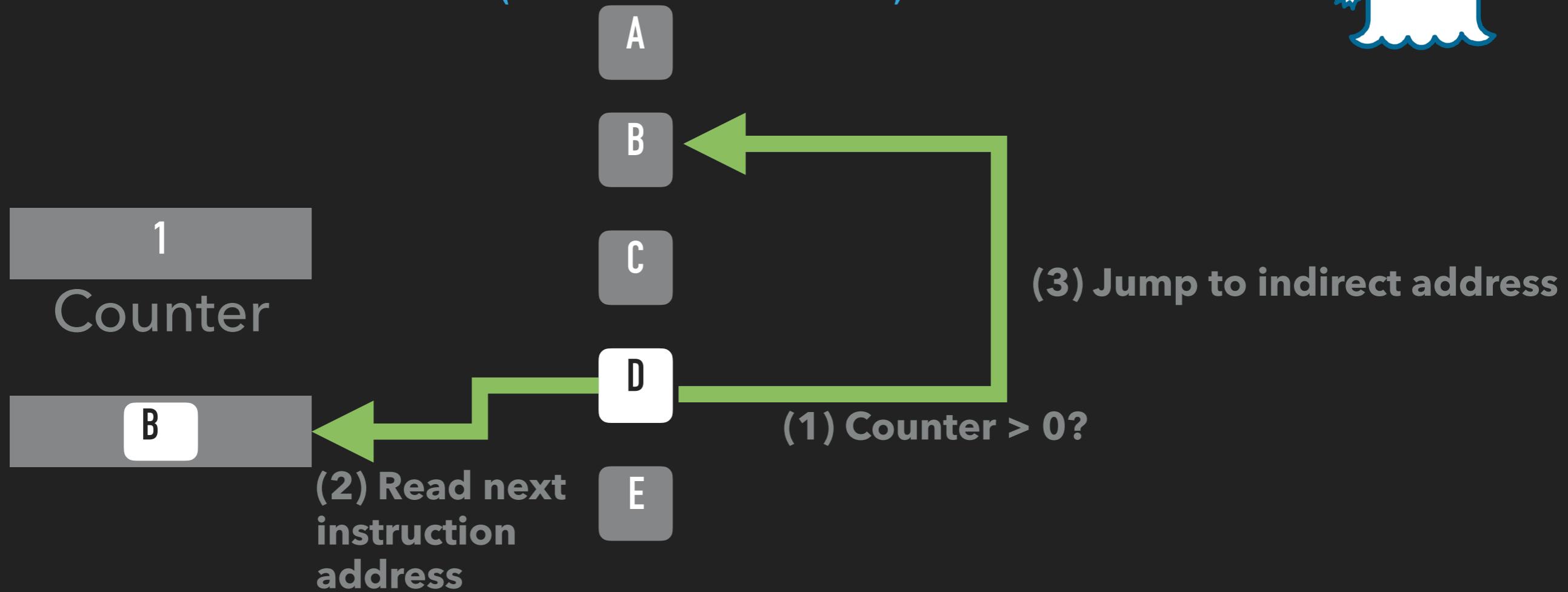
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

SPECTRE: VARIANT 2 (CVE-2017-5715)



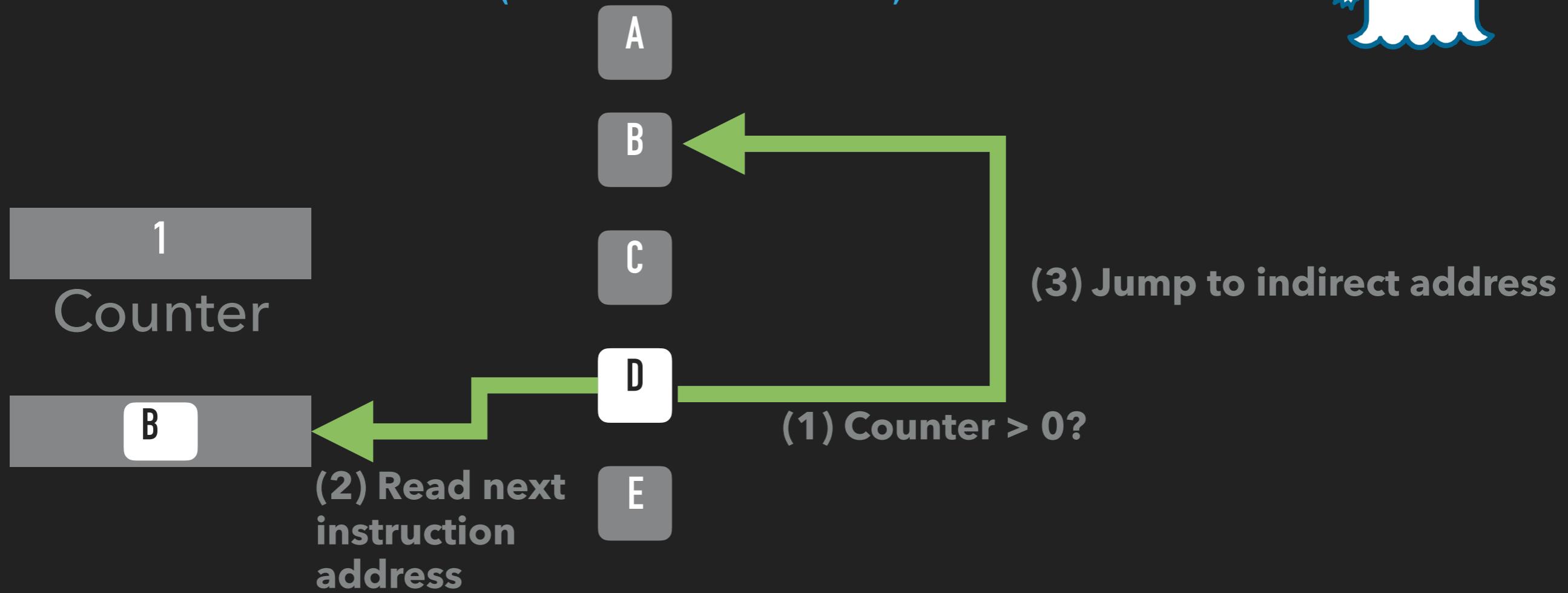
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

SPECTRE: VARIANT 2 (CVE-2017-5715)



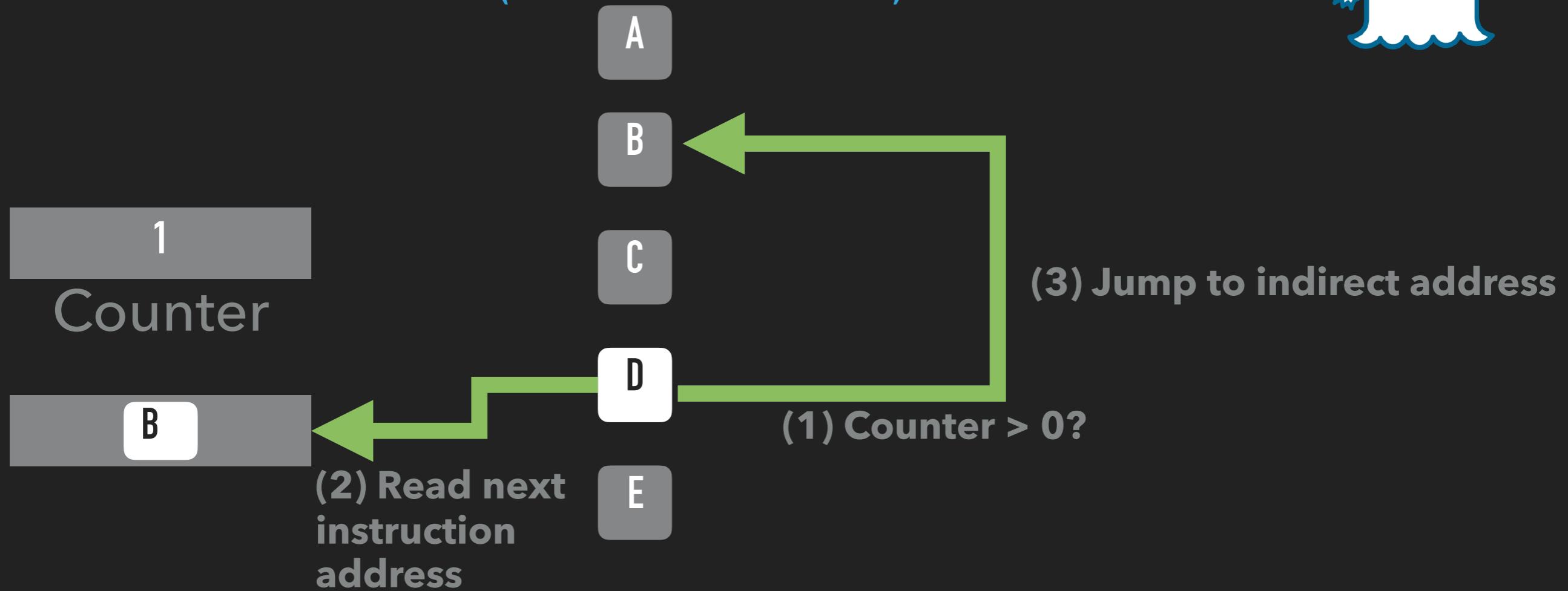
- ▶ The conditional jump (branch) **D** now is an *indirect jump*.
- ▶ Indirect jumps use addresses stored "somewhere else".

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).



MELTDOWN AND
SPECTRE

CONCLUSION

THREAT-O-METER

LOW RISK

Exploit unlikely or
running
untrusted code already
worst case

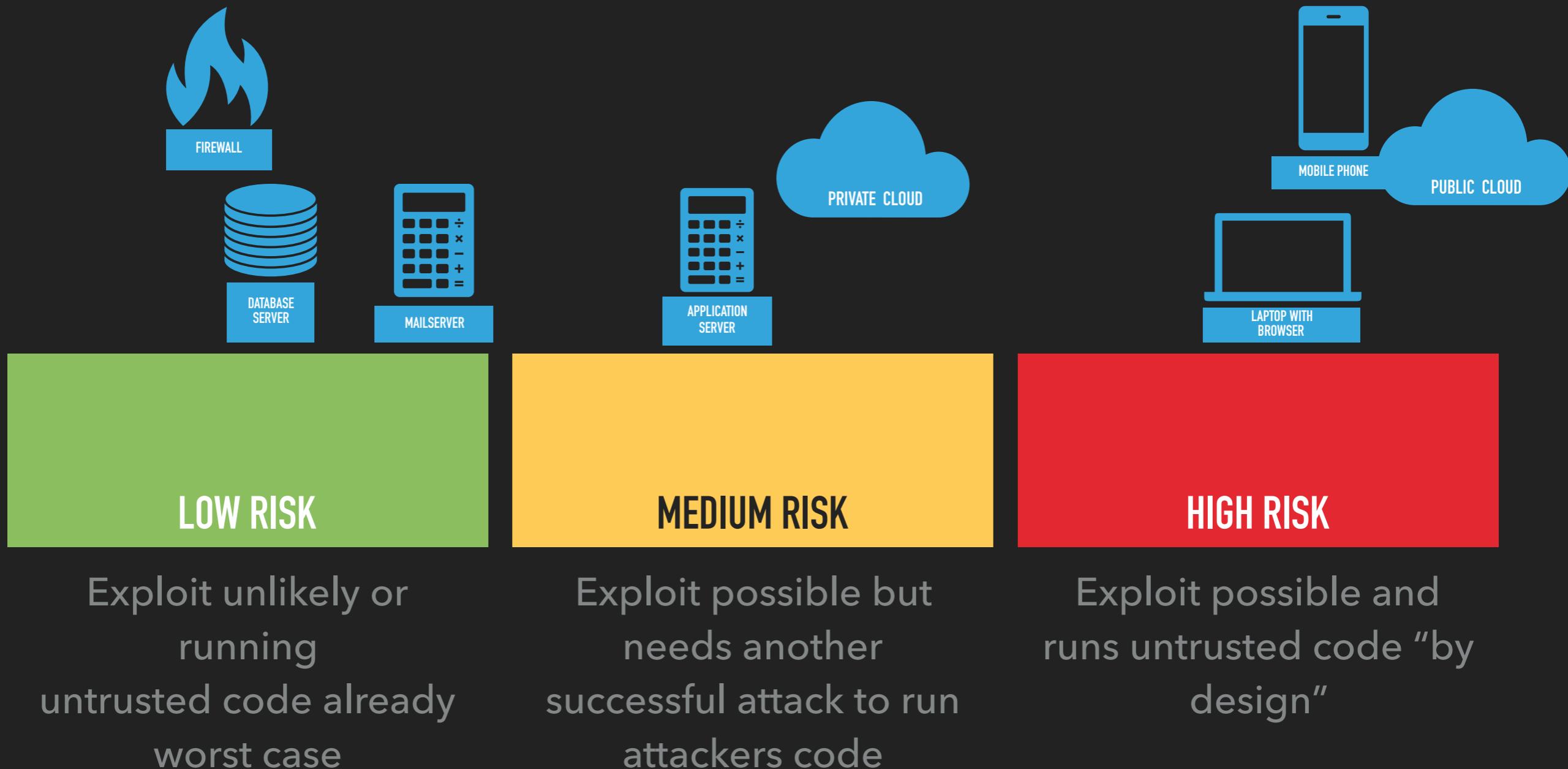
MEDIUM RISK

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

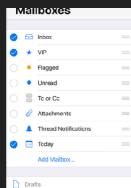




BOTH THE MELTDOWN AND SPECTRE LOGO ARE FREE TO USE, RIGHTS WAIVED
VIA [CCO](#). LOGOS ARE DESIGNED BY [NATASCHA EIBL](#).
[HTTPS://SPECTREATTACK.COM/](https://spectreattack.com/)



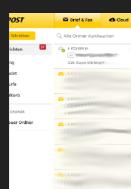
CANDY CRUSH LOGO FROM THE APP STORE © KING
[HTTPS://DISCOVER.KING.COM/ABOUT/](https://discover.king.com/about/)



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