Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2024

Lecture 23: Programmed IO & DMA

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Roadmap

- Wrap up parallel systems
  - Hardware support for threads
- Programmed IO and DMA
  - Chapter 10 (10.1 – 10.7)
- Networking
  - Chapter 13

Synchronization support in a uniprocessor

Synchronization support in a uniprocessor

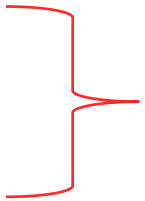
- Thread creation/termination

Synchronization support in a uniprocessor

- Thread creation/termination
- Communication among threads

Synchronization support in a uniprocessor

- Thread creation/termination
- Communication among threads

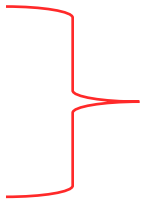


Nothing special needed…

PT, TLB, Cache all the same

Synchronization support in a uniprocessor

- Thread creation/termination
- Communication among threads
- Synchronization among threads
  - How do we implement mutex_lock?



Nothing special needed…

PT, TLB, Cache all the same

Proposed implementation of mutex

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

Unlock():

```
mem_lock = 0
```

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

  Unlock():

```
mem_lock = 0
```

- What could go wrong?

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

Oops! These instructions aren't atomic

Unlock():

```
mem_lock = 0
```

- What could go wrong?

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

Unlock():

```
mem_lock = 0
```

- What could go wrong?

Oops! These instructions aren't atomic

How did we deal with that earlier?

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

  Unlock():

```
mem_lock = 0
```

- What could go wrong?

Oops! These instructions aren't atomic

How did we deal with that earlier?

We used OS mutex calls to make instructions inseparable.

Proposed implementation of mutex

- Lock():

```
while (mem_lock != 0)
        block the thread
mem_lock = 1;
```

  Unlock():

```
mem_lock = 0
```

- What could go wrong?

Oops! These instructions aren't atomic

How did we deal with that earlier?

We used OS mutex calls to make instructions inseparable.

But now we ARE the OS!

Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
x	0	x

Lock() using machine instructions

T1

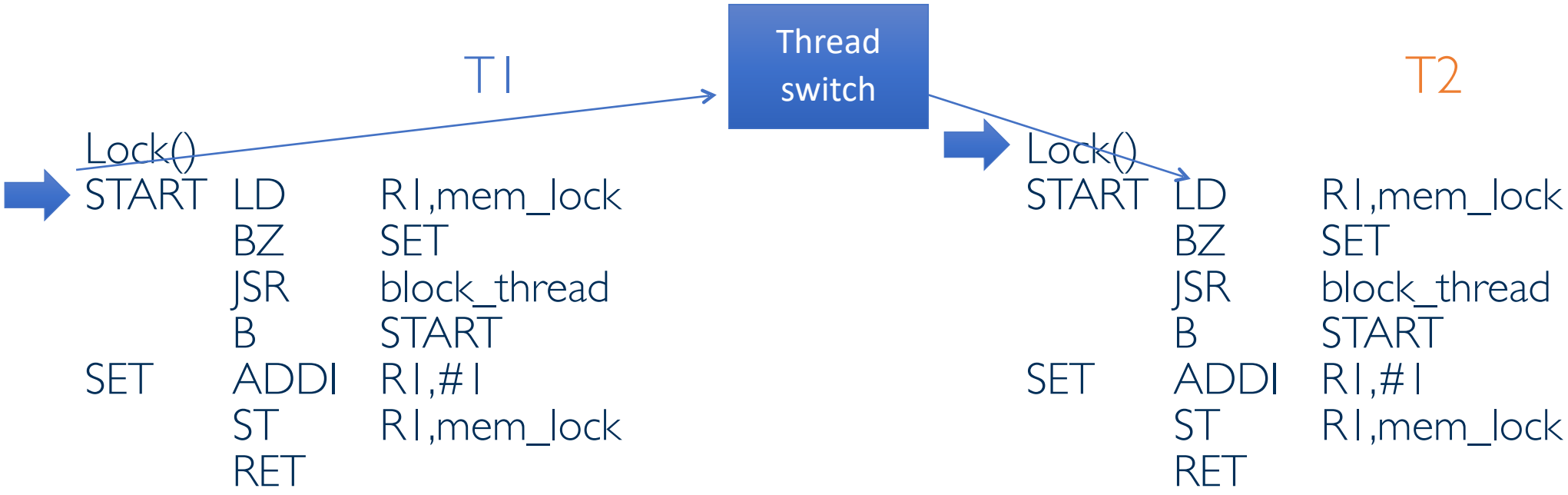
```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
x ~~0~~	0	x

Lock() using machine instructions



R1	mem_lock	R1
x ~~0~~	0	x

Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0	~~x~~ 0

Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0	~~x~~ 0

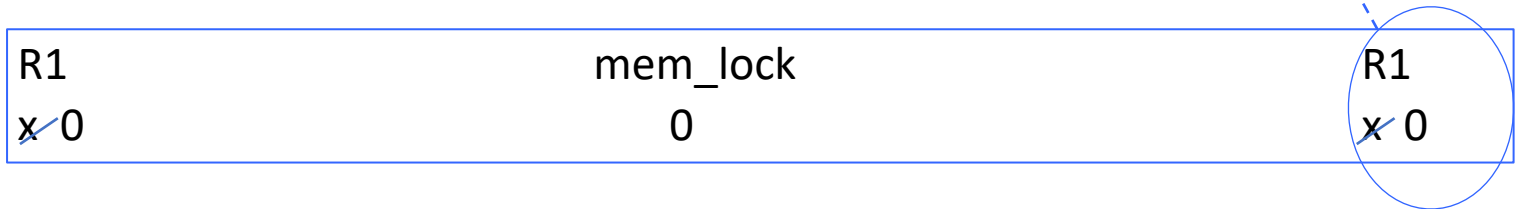
Lock() using machine instructions

T1

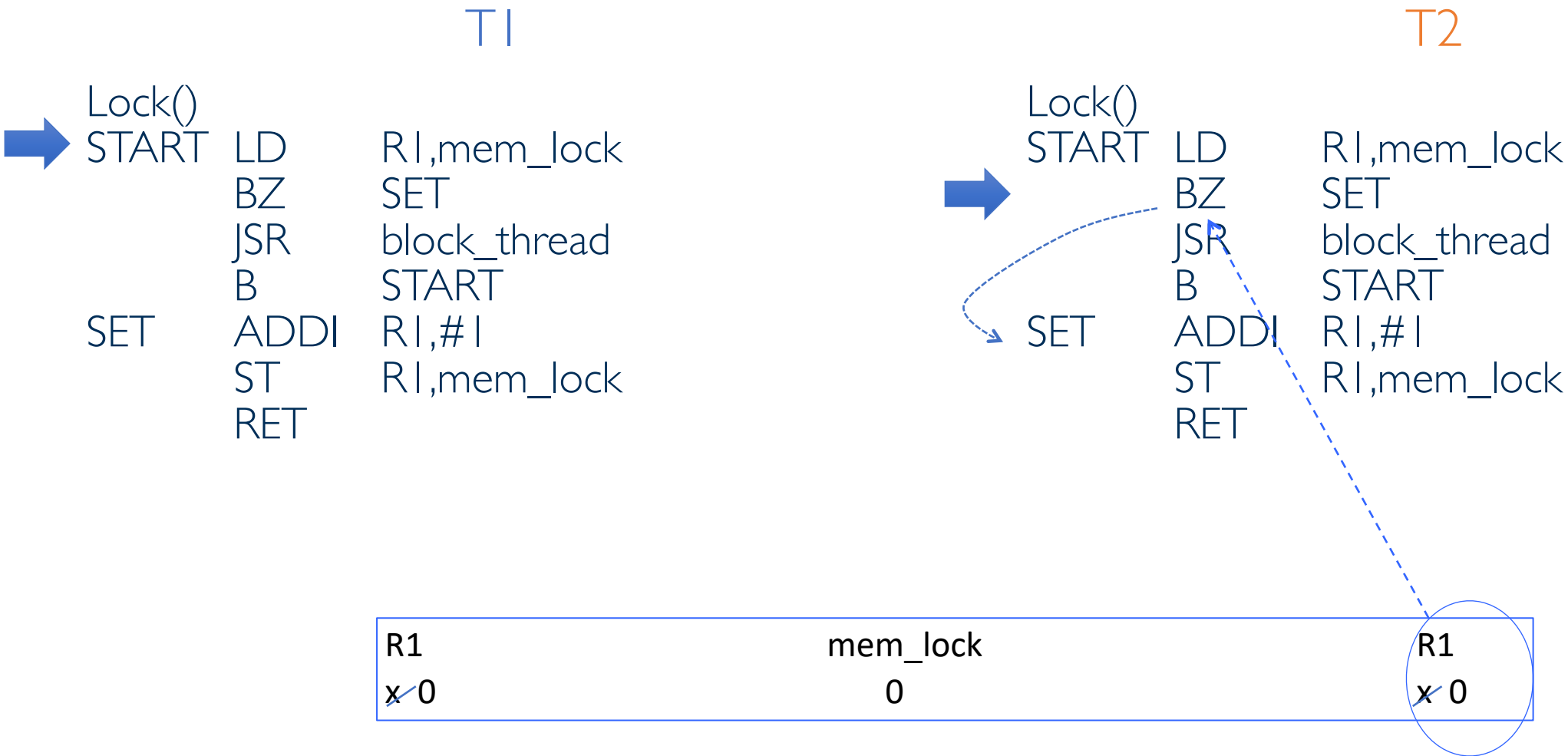
```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```



Lock() using machine instructions



Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0	~~x~~ 0 ~~1~~

Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0 ~~1~~	~~x~~ 0 ~~1~~

Lock() using machine instructions

T1

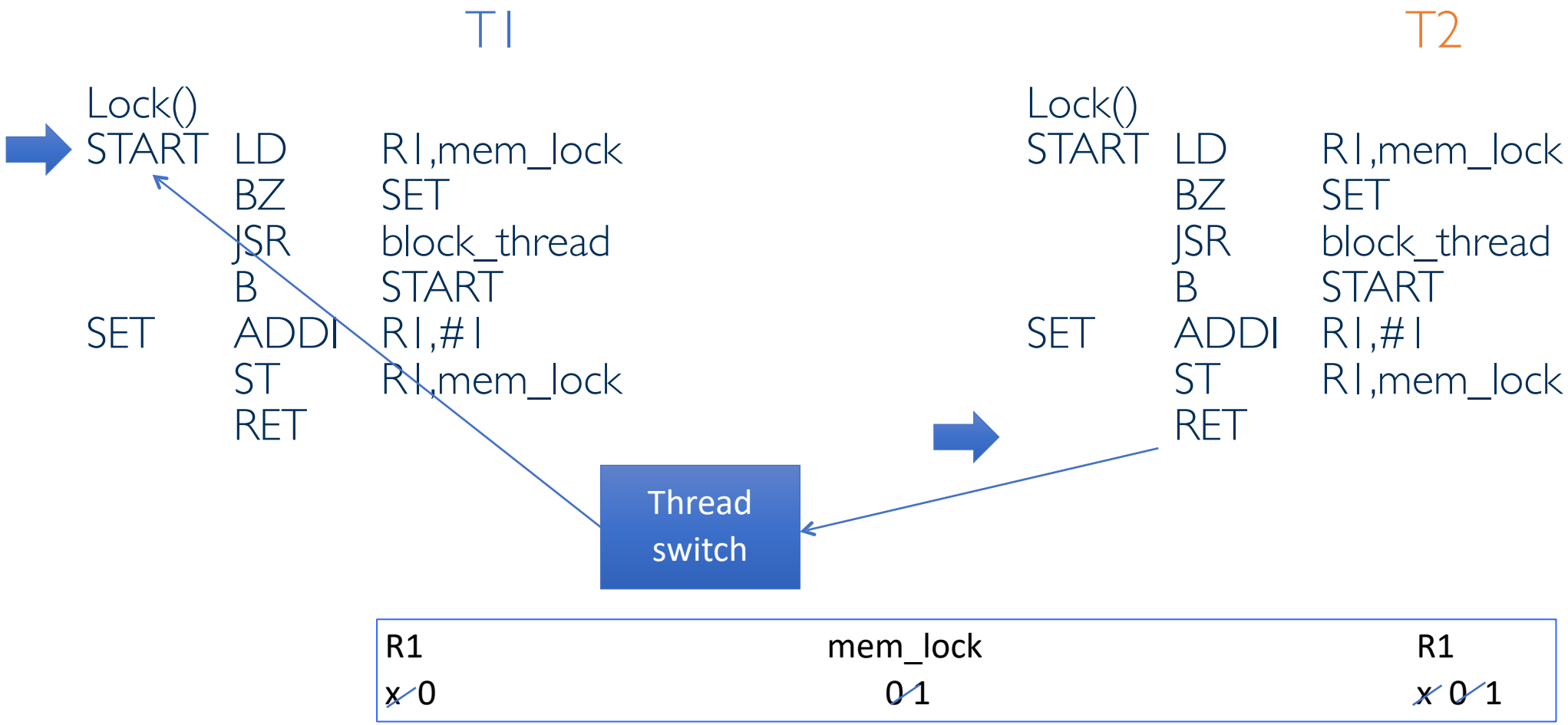
```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0 ~~1~~	~~x~~ 0 ~~1~~

Lock() using machine instructions



Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ 0	0 ~~1~~	~~x~~ 0 ~~1~~

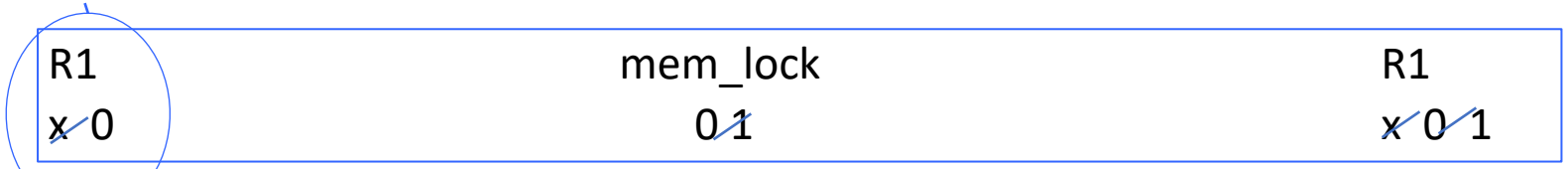
Lock() using machine instructions

T1

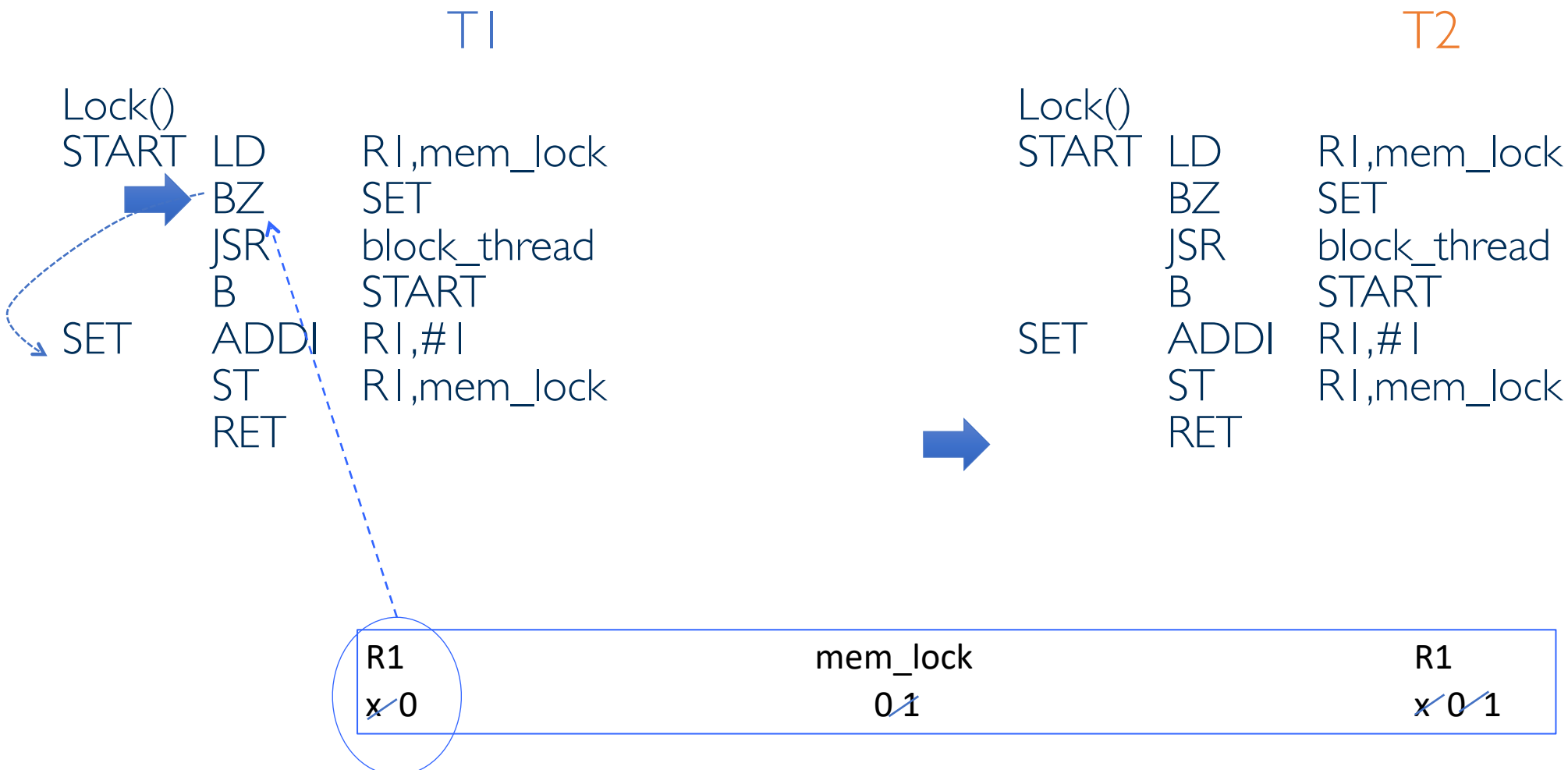
```
Lock()
START  LD     R1,mem_lock
       BZ     SET
       JSR    block_thread
       B      START
SET    ADDI   R1,#1
       ST     R1,mem_lock
       RET
```

T2

```
Lock()
START  LD     R1,mem_lock
       BZ     SET
       JSR    block_thread
       B      START
SET    ADDI   R1,#1
       ST     R1,mem_lock
       RET
```



Lock() using machine instructions



Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

R1	mem_lock	R1
~~x~~ ~~0~~ 1	~~0~~ 1	~~x~~ ~~0~~ 1

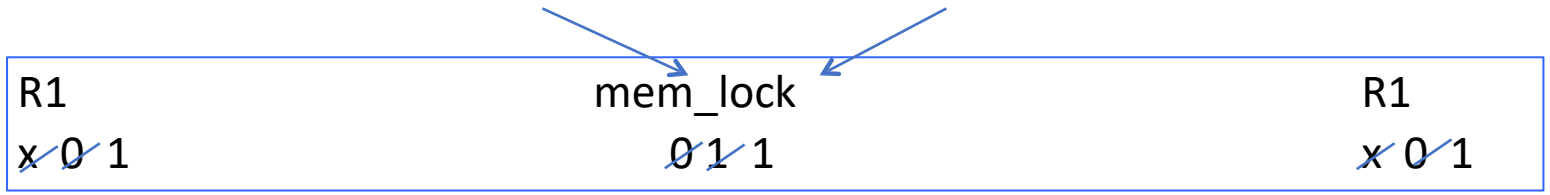
Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```



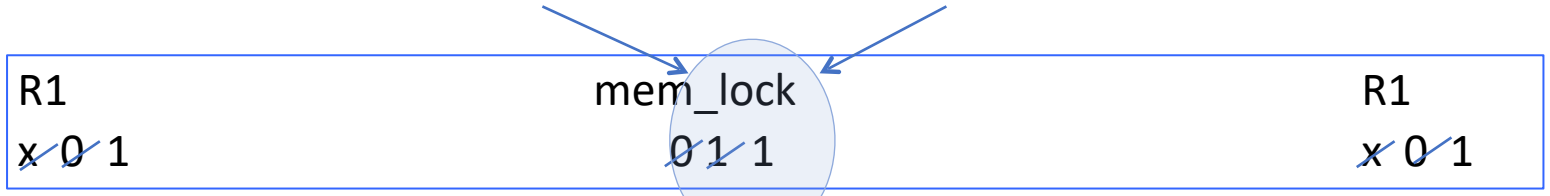
Lock() using machine instructions

T1

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```

T2

```
Lock()
START   LD      R1,mem_lock
        BZ      SET
        JSR     block_thread
        B       START
SET     ADDI    R1,#1
        ST      R1,mem_lock
        RET
```



What happened?!?

What happened?!?

- We weren't in a critical section (We're implementing the lock for a critical section – how could we be in one??)

What happened?!?

- We weren't in a critical section (We're implementing the lock for a critical section – how could we be in one??)
- Interrupts *can* happen between instructions

What happened?!?

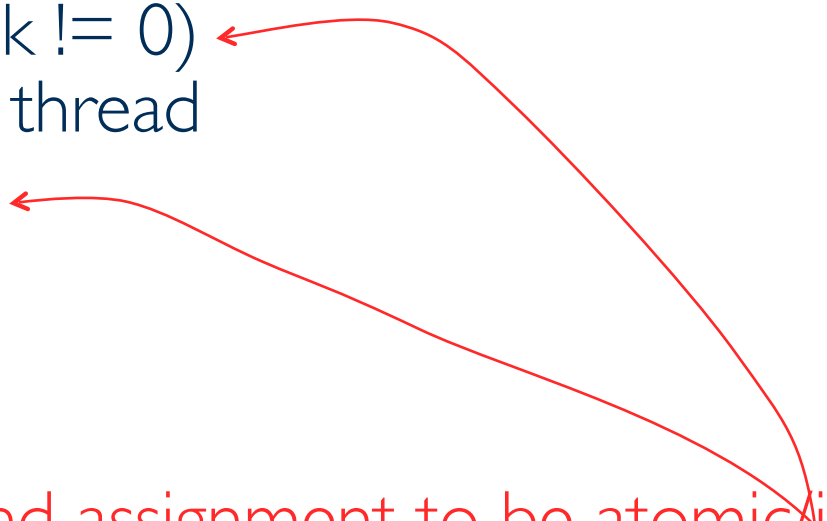
- We weren't in a critical section (We're implementing the lock for a critical section – how could we be in one??)
- Interrupts *can* happen between instructions

```
Lock():
        while (mem_lock != 0)
                block the thread
        mem_lock = 1;

Unlock():
        mem_lock = 0
```

What happened?!?

- We weren't in a critical section (We're implementing the lock for a critical section – how could we be in one??)
- Interrupts *can* happen between instructions

```
Lock():
        while (mem_lock != 0)
                block the thread
        mem_lock = 1;

Unlock():
        mem_lock = 0
```



- We need the test and assignment to be atomic/indivisible!

Hardware to the rescue!

Hardware to the rescue!

- We need an atomic Read-Modify-Write instruction

Hardware to the rescue!

- We need an atomic Read-Modify-Write instruction
- TEST-AND-SET <memory-location>
  load current value in <memory-location>
  store 1 in <memory-location>

Hardware to the rescue!

- We need an atomic Read-Modify-Write instruction
- TEST-AND-SET <memory-location>
  load current value in <memory-location>
  store 1 in <memory-location>
- Atomically:
  - Test L and set it to 1
  - If L tested originally as 0, we've claimed the lock
  - If L tested as 1, we need to try again

Hardware to the rescue!

- We need an atomic Read-Modify-Write instruction
- TEST-AND-SET <memory-location>
  load current value in <memory-location>
  store 1 in <memory-location>
- Atomically:
  - Test L and set it to 1
  - If L tested originally as 0, we've claimed the lock
  - If L tested as 1, we need to try again
- Work-alikes
  - Compare-and-swap (IBM 370)
  - Fetch-and-add (Intel x86)
  - Load-linked/store-conditional (MIPS, ARM, …)

Our new implementation of mutex

Our new implementation of mutex

- Lock():
  while (test-and-set (&mem_lock))
  block the thread

Unlock():
  mem_lock = 0

Example: Implementing Mutex Lock

```
static int shared-lock =  0; /* global variable to
                                both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int *L)
{
        int X;

        X = test-and-set(L);

        /* X = 0 for successful return */
        return(X);
}
```

Example: Implementing Mutex Lock

```
static int shared-lock =  0; /* global variable to
                                both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int *L)
{
        int X;

        X = test-and-set(L);

        /* X = 0 for successful return */
        return(X);
}
```

Two threads T1 and T2 execute the following statement simultaneously:

MyX = binary_semaphore(&shared-lock);

where MyX is a local variable in each of T1 and T2.

Example: Implementing Mutex Lock

```
static int shared-lock =  0; /* global variable to
                                both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int *L)
{
        int X;

        X = test-and-set(L);

        /* X = 0 for successful return */
        return(X);
}
```

Two threads T1 and T2 execute the following statement simultaneously:

MyX = binary_semaphore(&shared-lock);

where MyX is a local variable in each of T1 and T2.

What are the possible values returned to T1 and T2?

Example: Implementing Mutex Lock

```
static int shared-lock =  0; /* global variable to
                                both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int *L)
{
        int X;

        X = test-and-set(L);

        /* X = 0 for successful return */
        return(X);
}
```

Two threads T1 and T2 execute the following statement simultaneously:

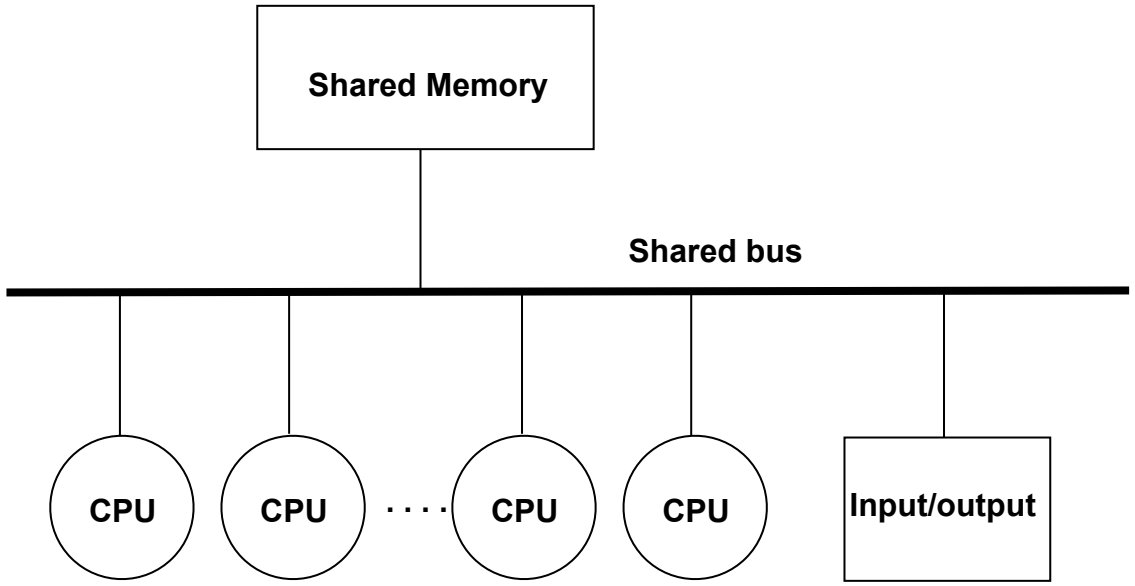
MyX = binary_semaphore(&shared-lock);

where MyX is a local variable in each of T1 and T2.

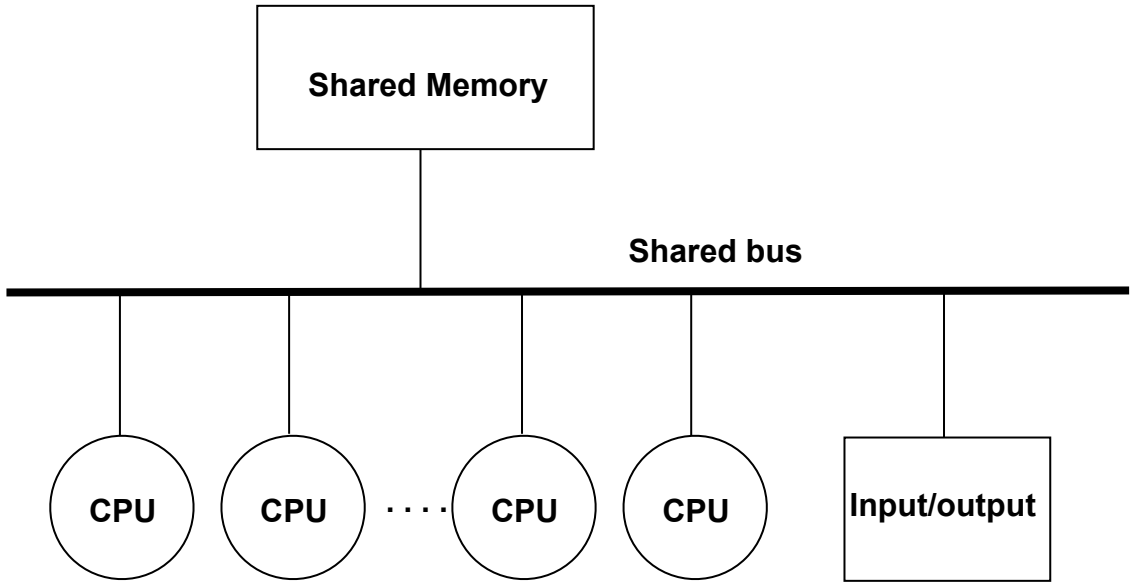
What are the possible values returned to T1 and T2?

Getting 0 0 isn't possible!

How do we implement Symmetric Multi-Processing (SMP)?

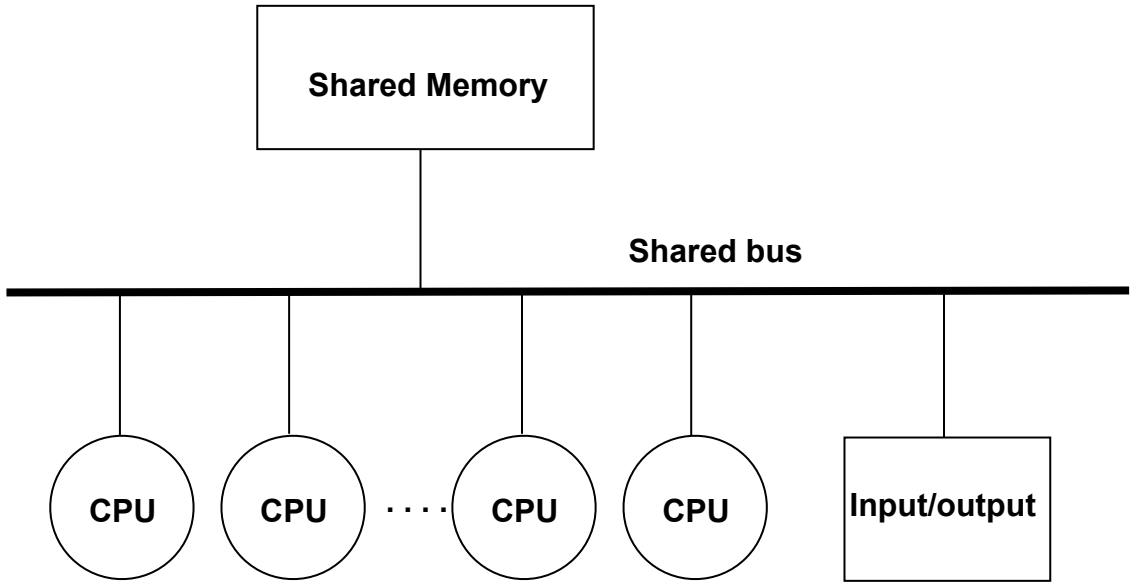


How do we implement Symmetric Multi-Processing (SMP)?



The System (hardware+OS) has to ensure 3 things:

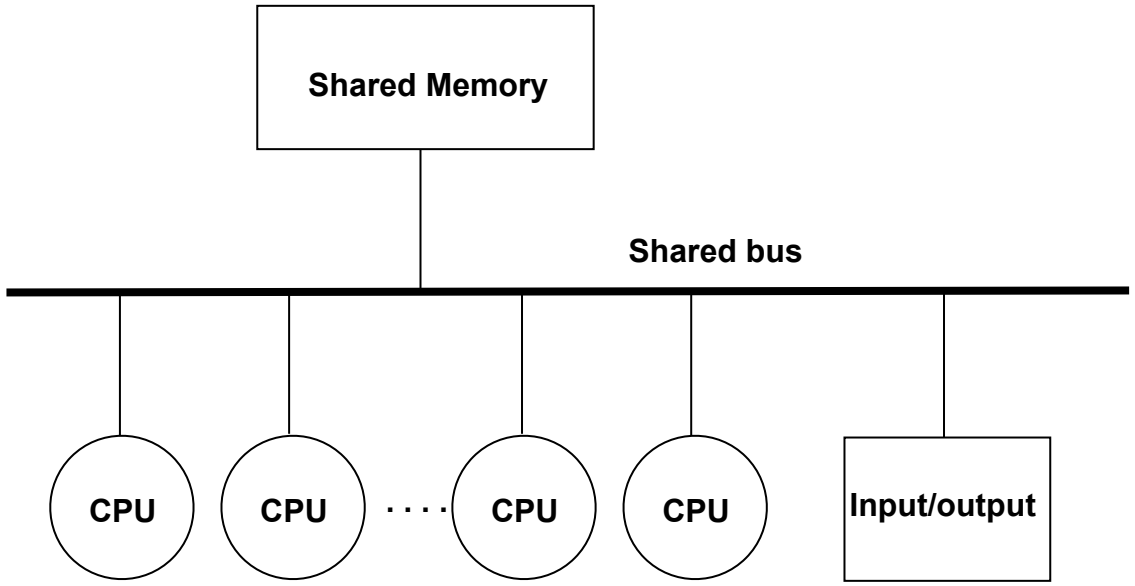
How do we implement Symmetric Multi-Processing (SMP)?



The System (hardware+OS) has to ensure 3 things:

1. Threads of the same process share the same PT

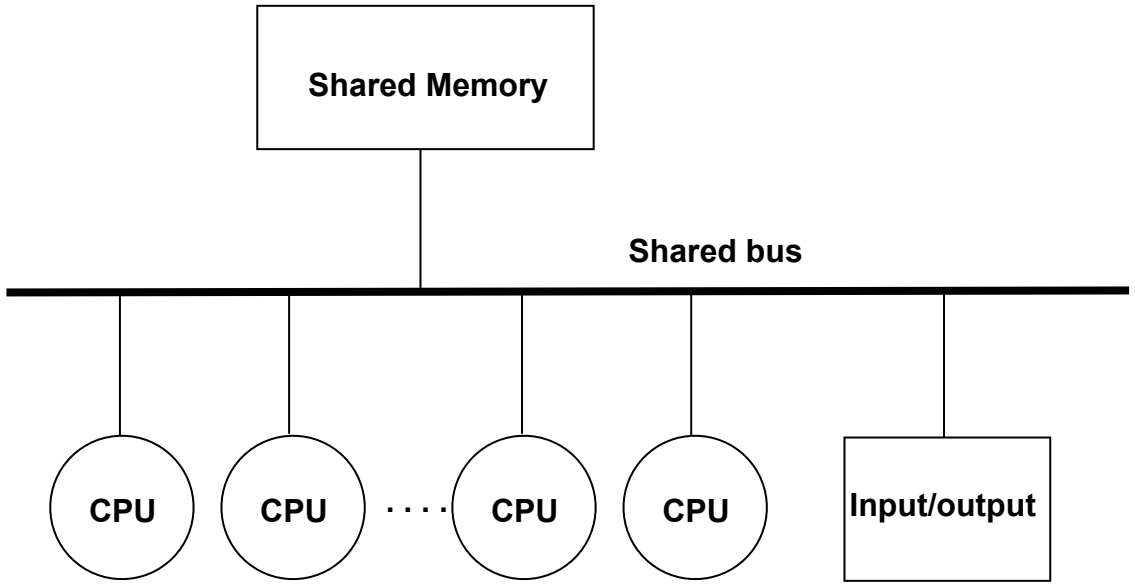
How do we implement Symmetric Multi-Processing (SMP)?



The System (hardware+OS) has to ensure 3 things:

1. Threads of the same process share the same PT
2. Threads have synchronization atomicity

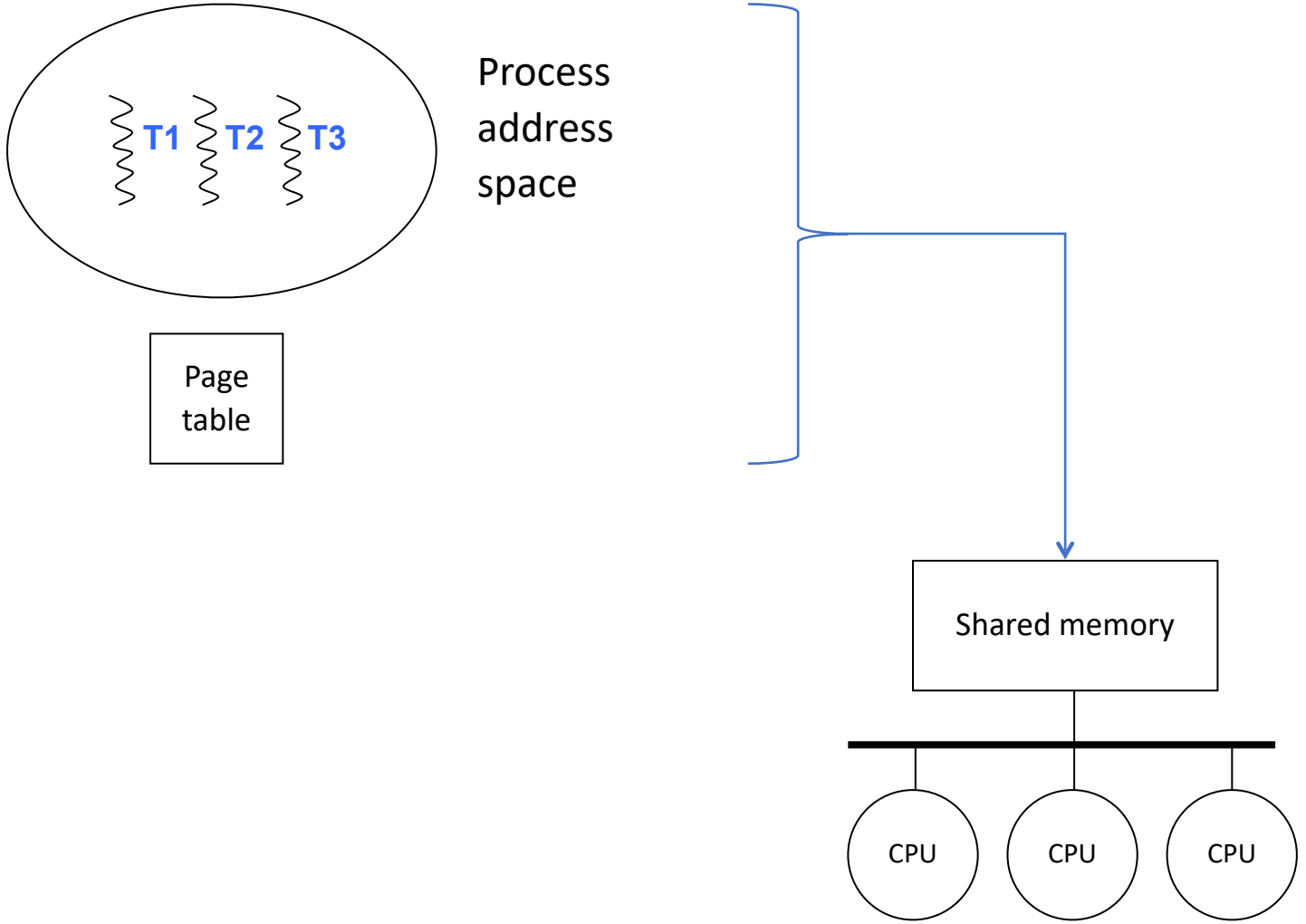
How do we implement Symmetric Multi-Processing (SMP)?



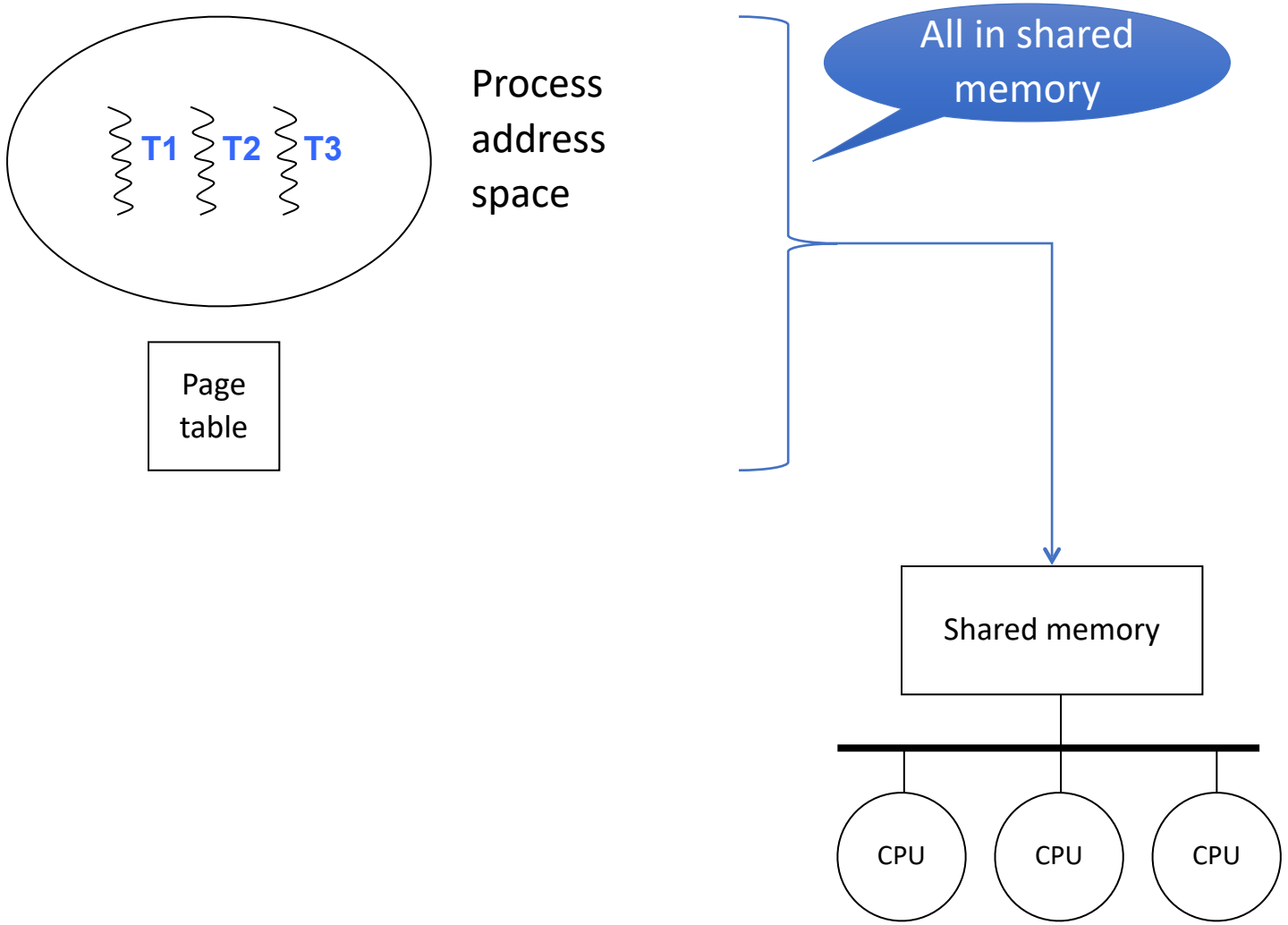
The System (hardware+OS) has to ensure 3 things:

1. Threads of the same process share the same PT
2. Threads have synchronization atomicity
3. Threads have identical views of memory

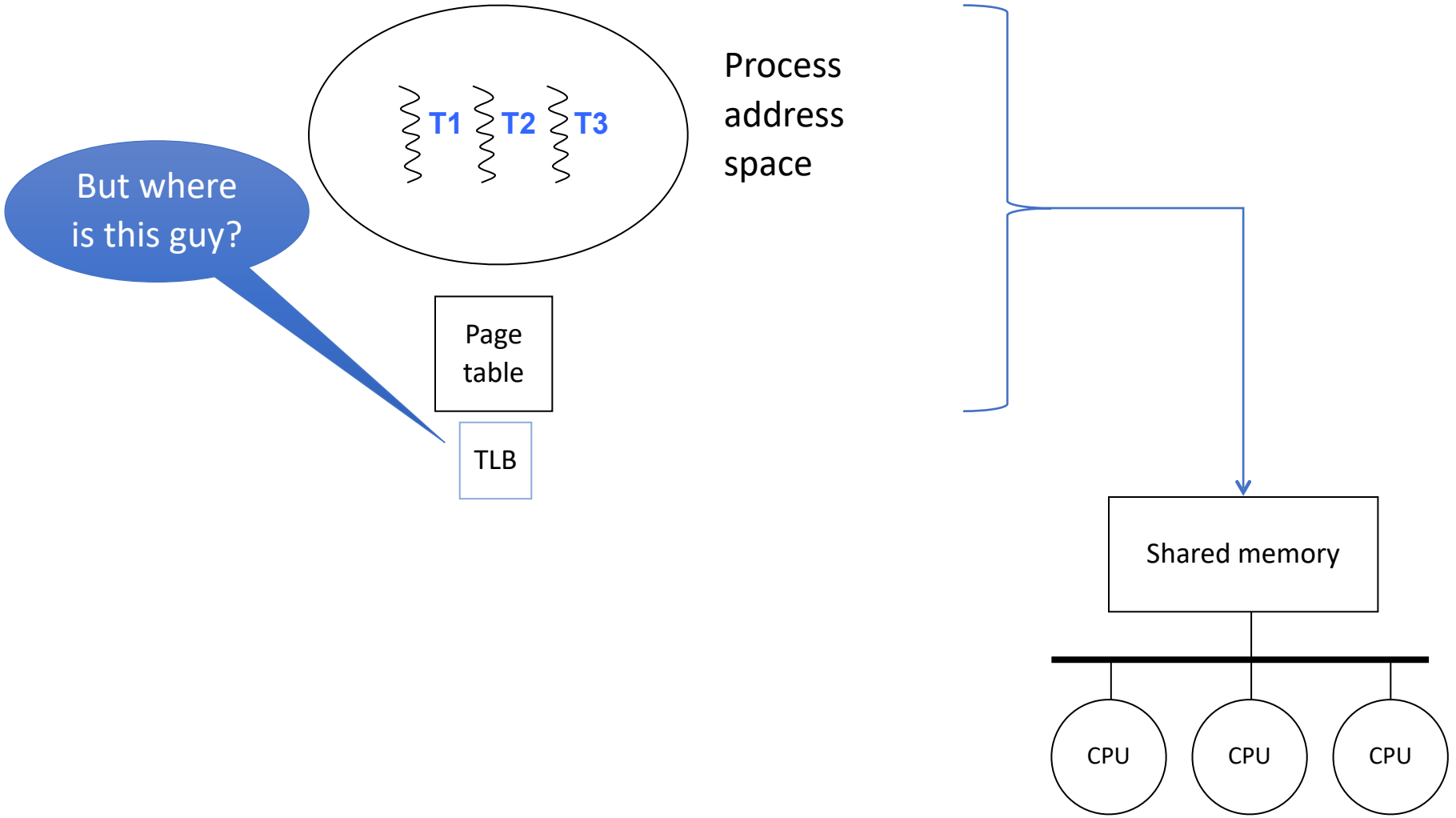
1) All threads share the same page table



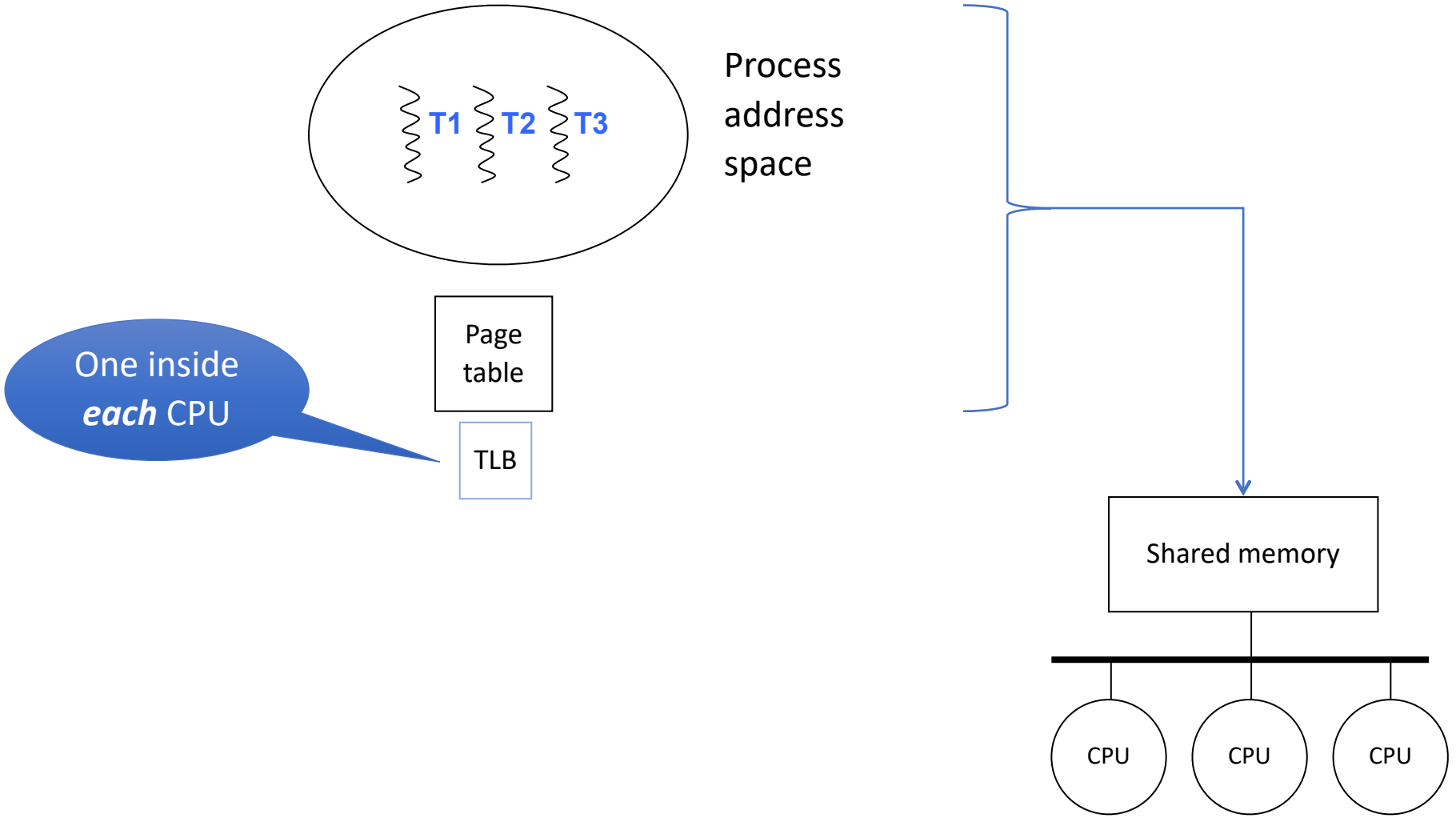
1) All threads share the same page table



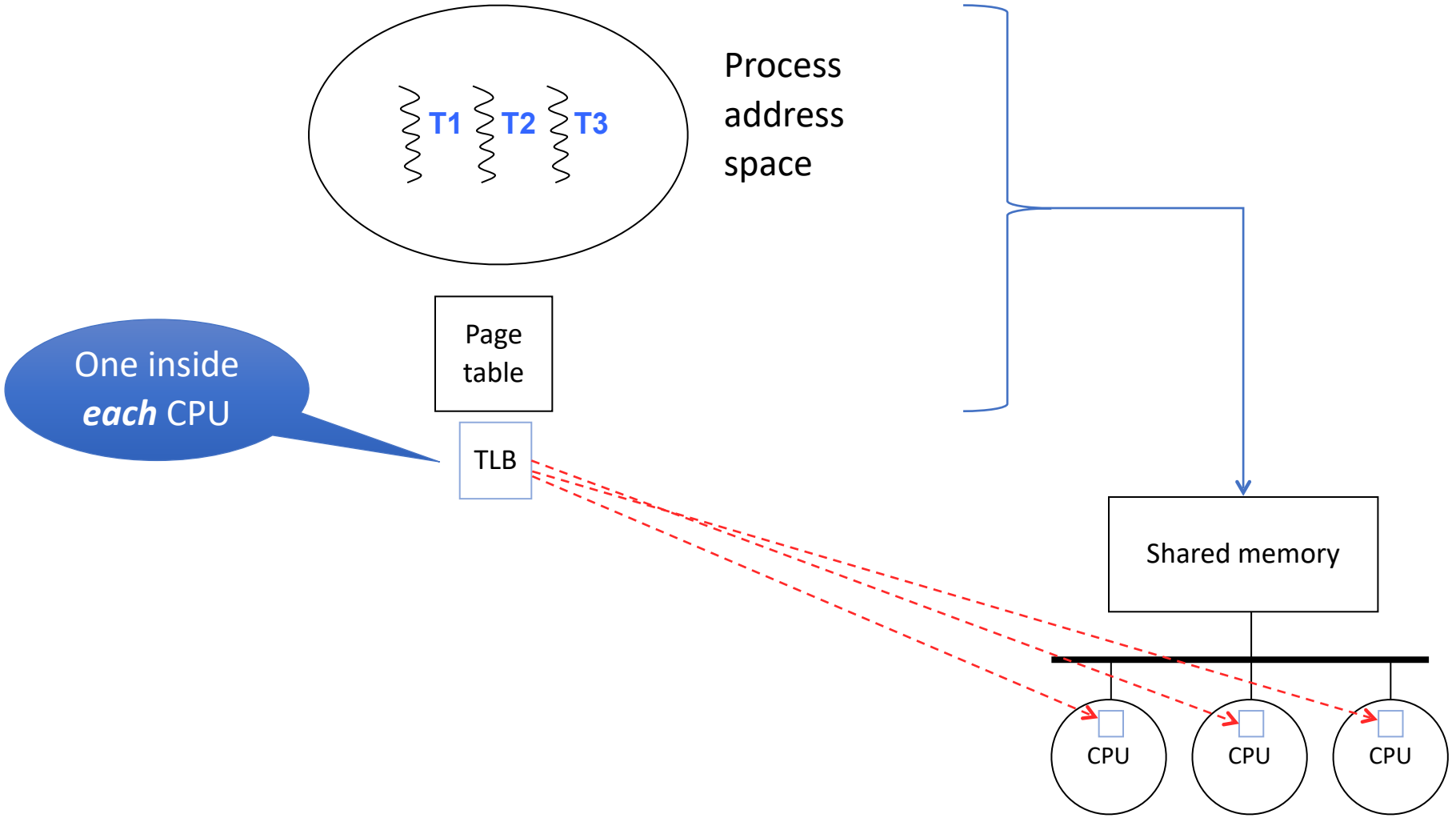
1) All threads share the same page table



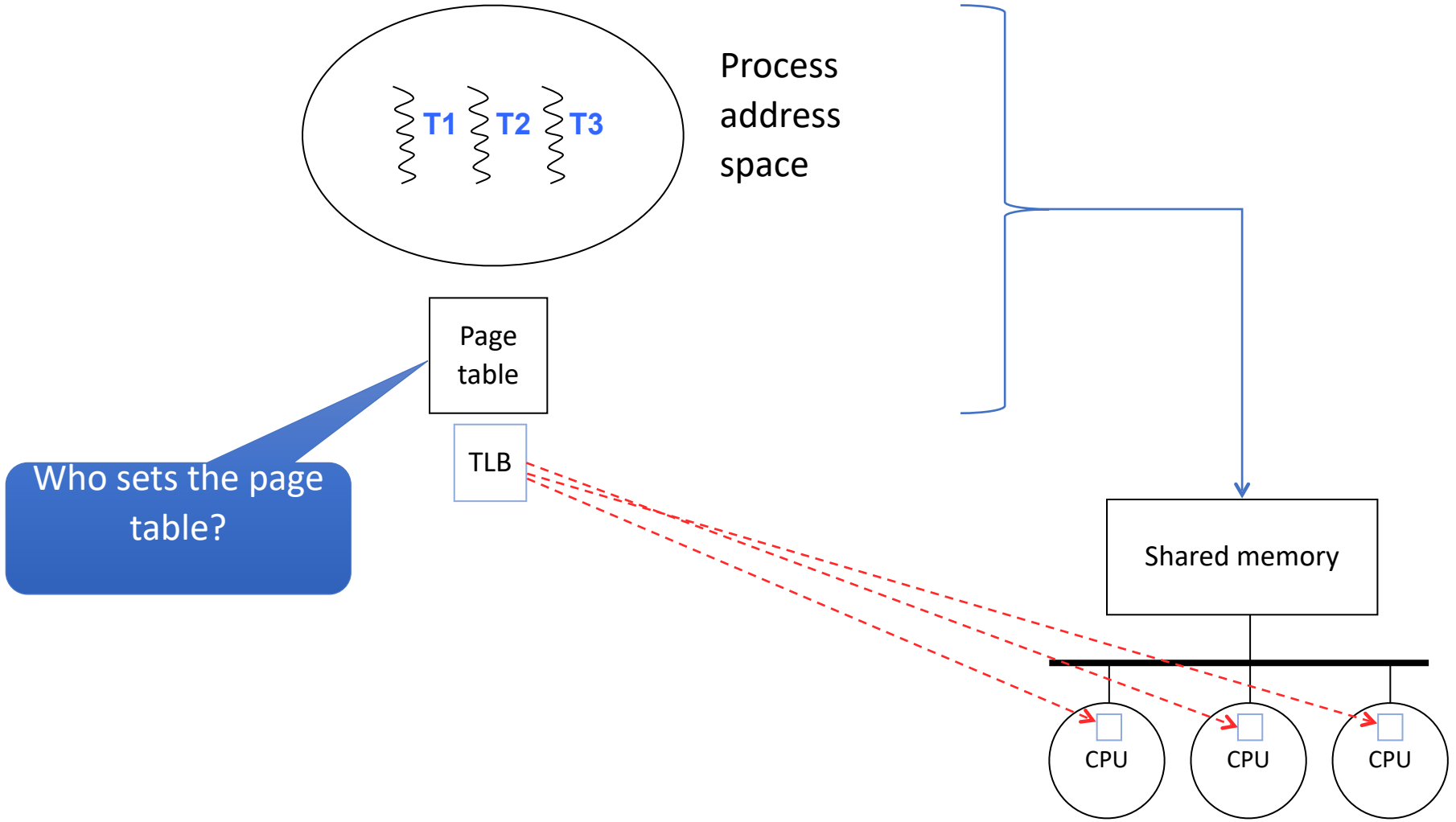
1) All threads share the same page table



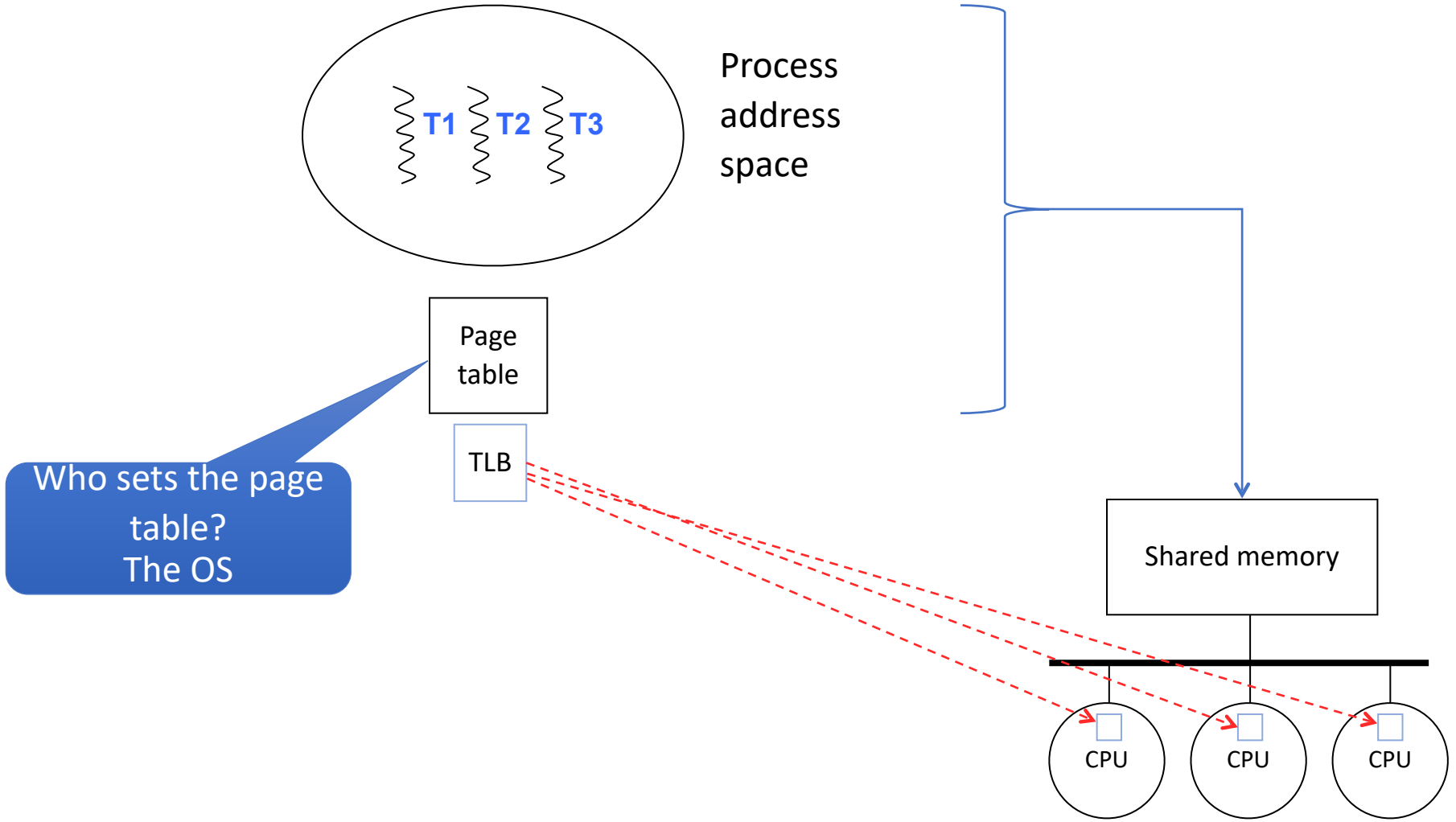
1) All threads share the same page table



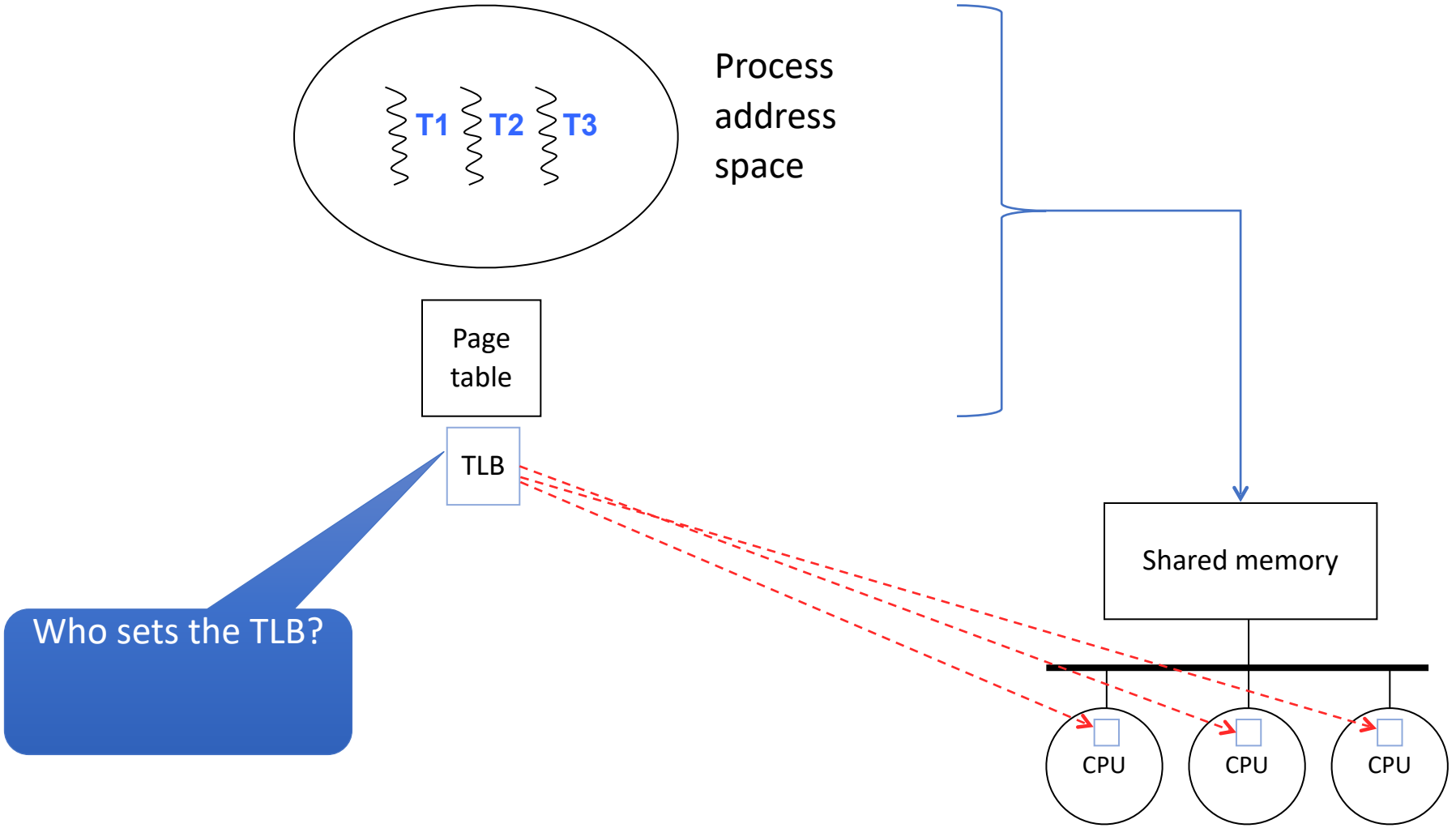
1) All threads share the same page table



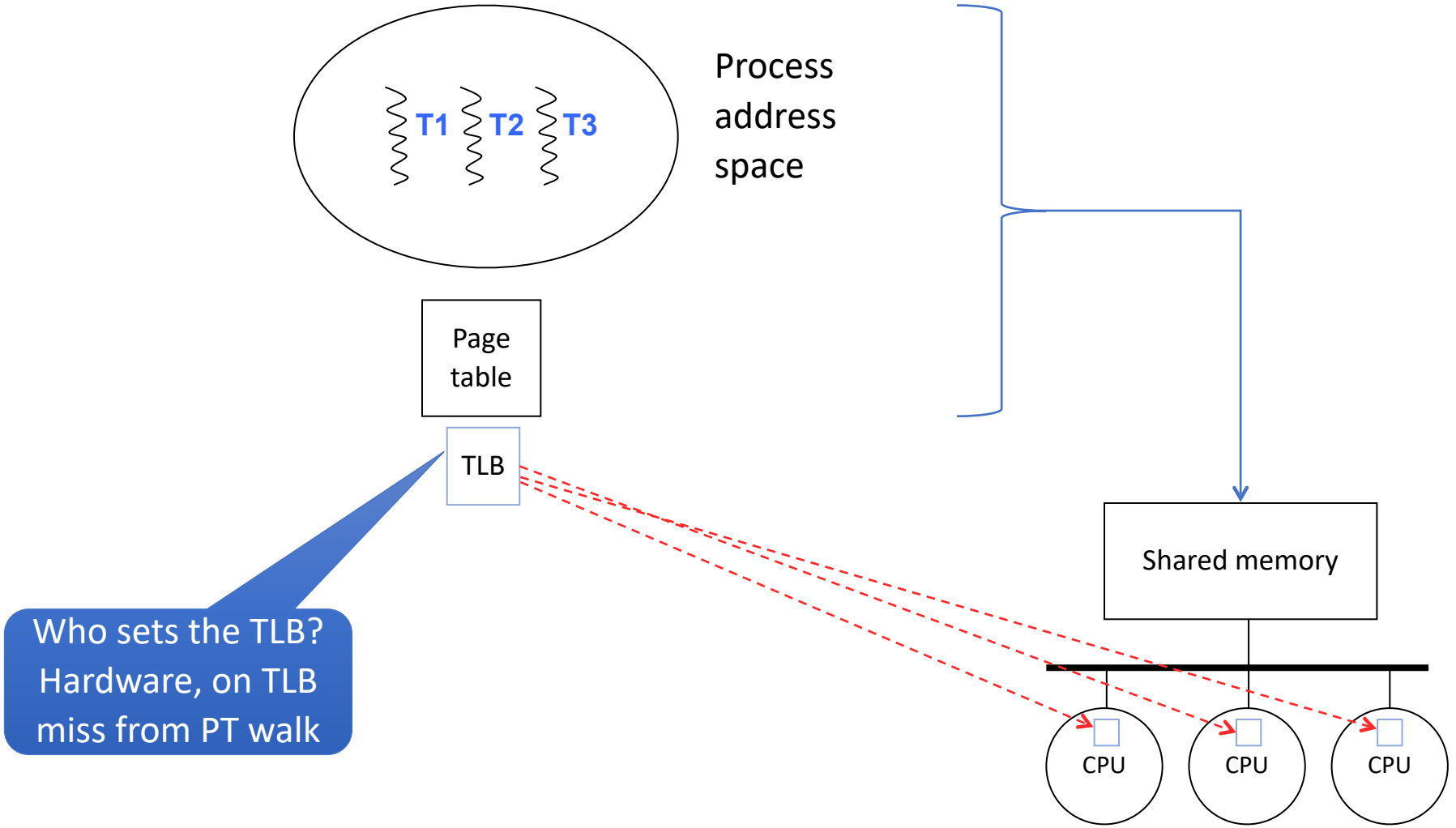
1) All threads share the same page table



1) All threads share the same page table



1) All threads share the same page table



SMP context switch handling

- As in the single-CPU case, the TLB must be flushed of user-space addresses on context switch

SMP context switch handling

- As in the single-CPU case, the TLB must be flushed of user-space addresses on context switch
- How do multiple processors complicate this?

SMP context switch handling

- As in the single-CPU case, the TLB must be flushed of user-space addresses on context switch
- How do multiple processors complicate this?
  - Basically, they don't
  - Any time a CPU is switched to a new thread, the OS flushes user entries from that CPU's TLB
  - There's no need to affect other TLBs

SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)

SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)
- OS must
  - Evict the TLB entry for that page
    (must happen even on a uniprocessor)

SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)
- OS must
  - Evict the TLB entry for that page
    (must happen even on a uniprocessor)
  - Tell the OS on other CPUs to evict the corresponding TLB entry (if present) using software interrupts

SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)
- OS must
  - Evict the TLB entry for that page
    (must happen even on a uniprocessor)
  - Tell the OS on other CPUs to evict the corresponding TLB entry (if present) using software interrupts
  - This is called TLB Shootdown and is expensive but necessary

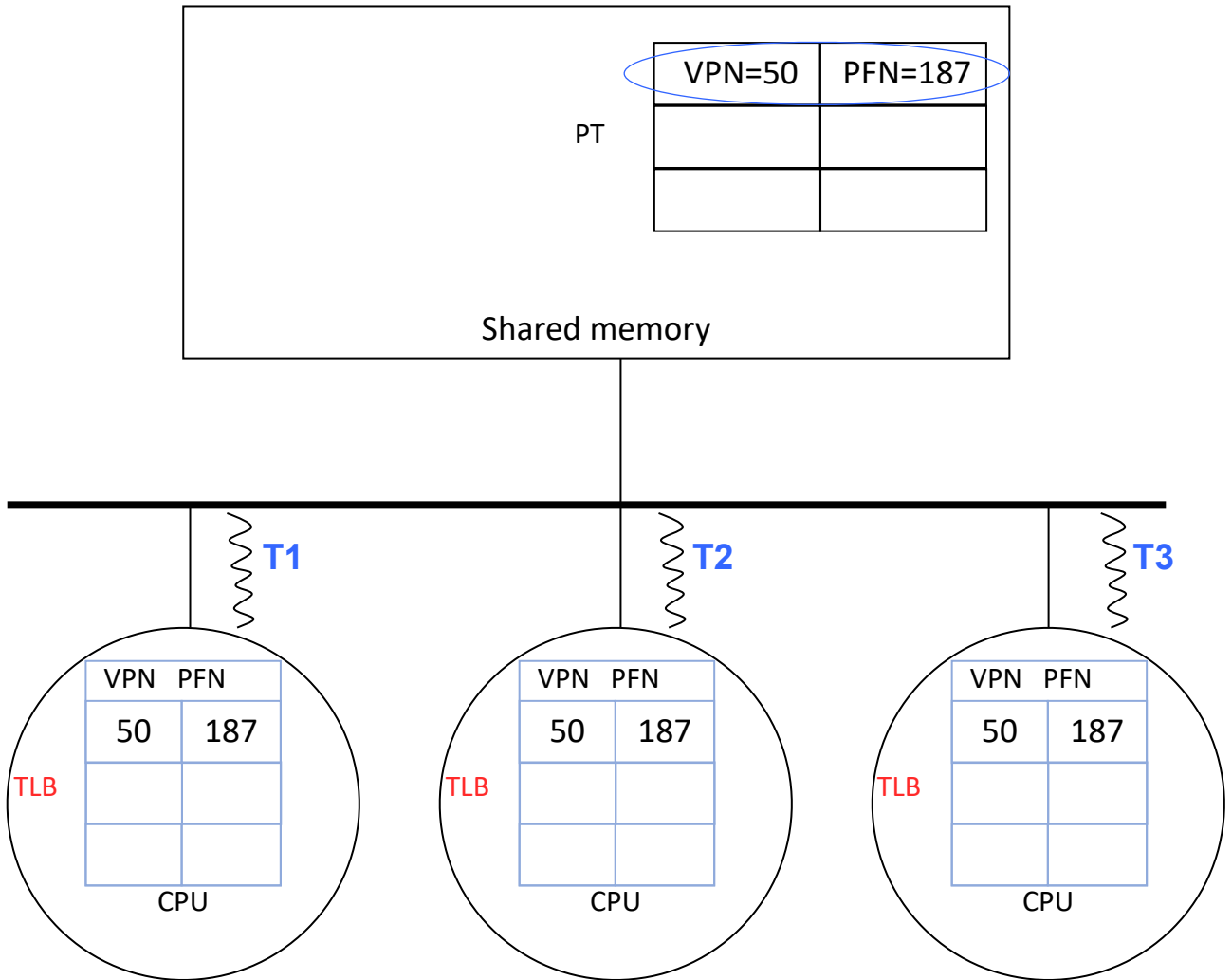
SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)
- OS must
  - Evict the TLB entry for that page
    (must happen even on a uniprocessor)
  - Tell the OS on other CPUs to evict the corresponding TLB entry (if present) using software interrupts
  - This is called TLB Shootdown and is expensive but necessary
- All of this happens in software by the OS

➔ Another partnership of hardware and software

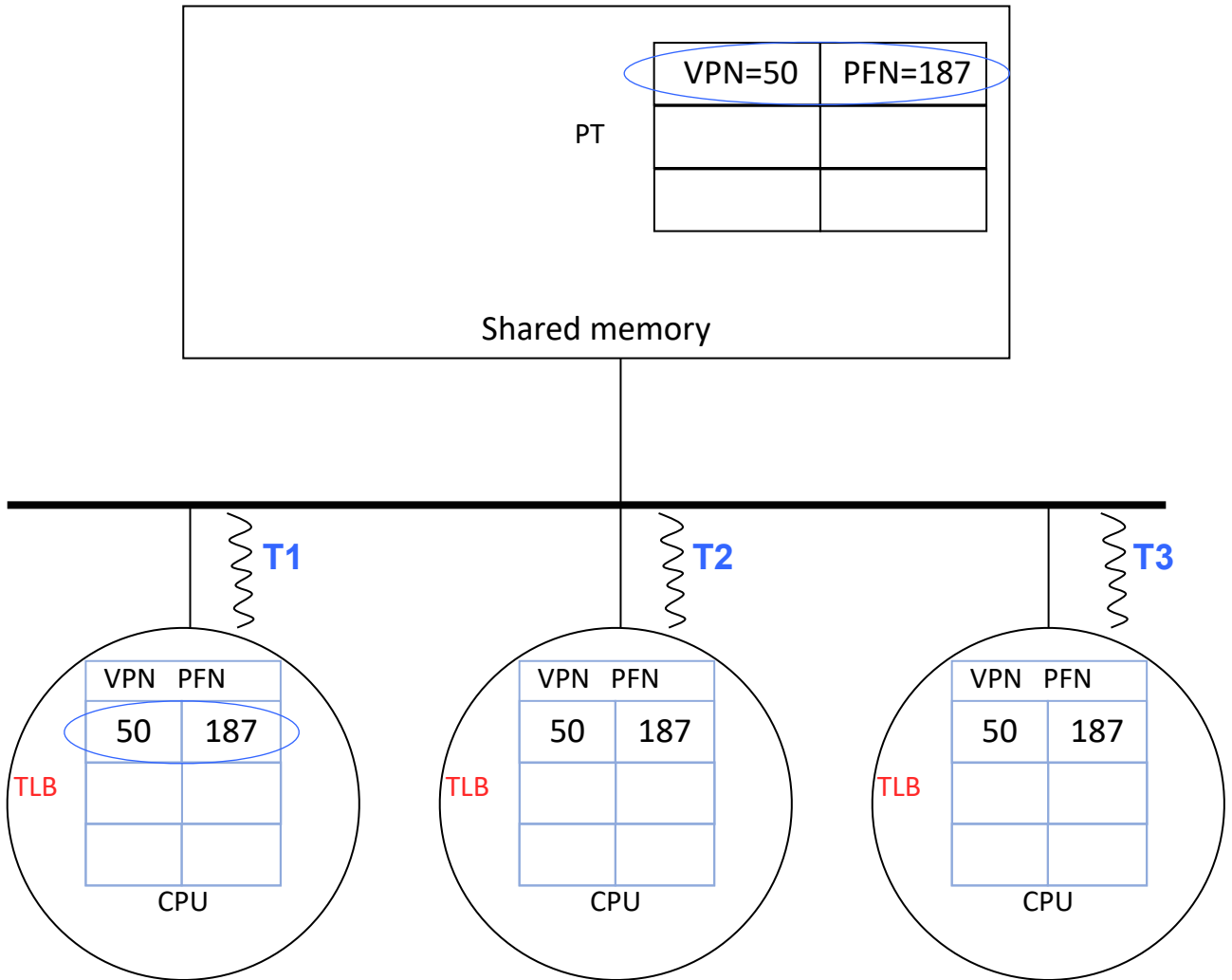
Example: Handling an SMP page fault

Note that the TLBs have each pulled in VPN=50 because each of the threads referenced that page



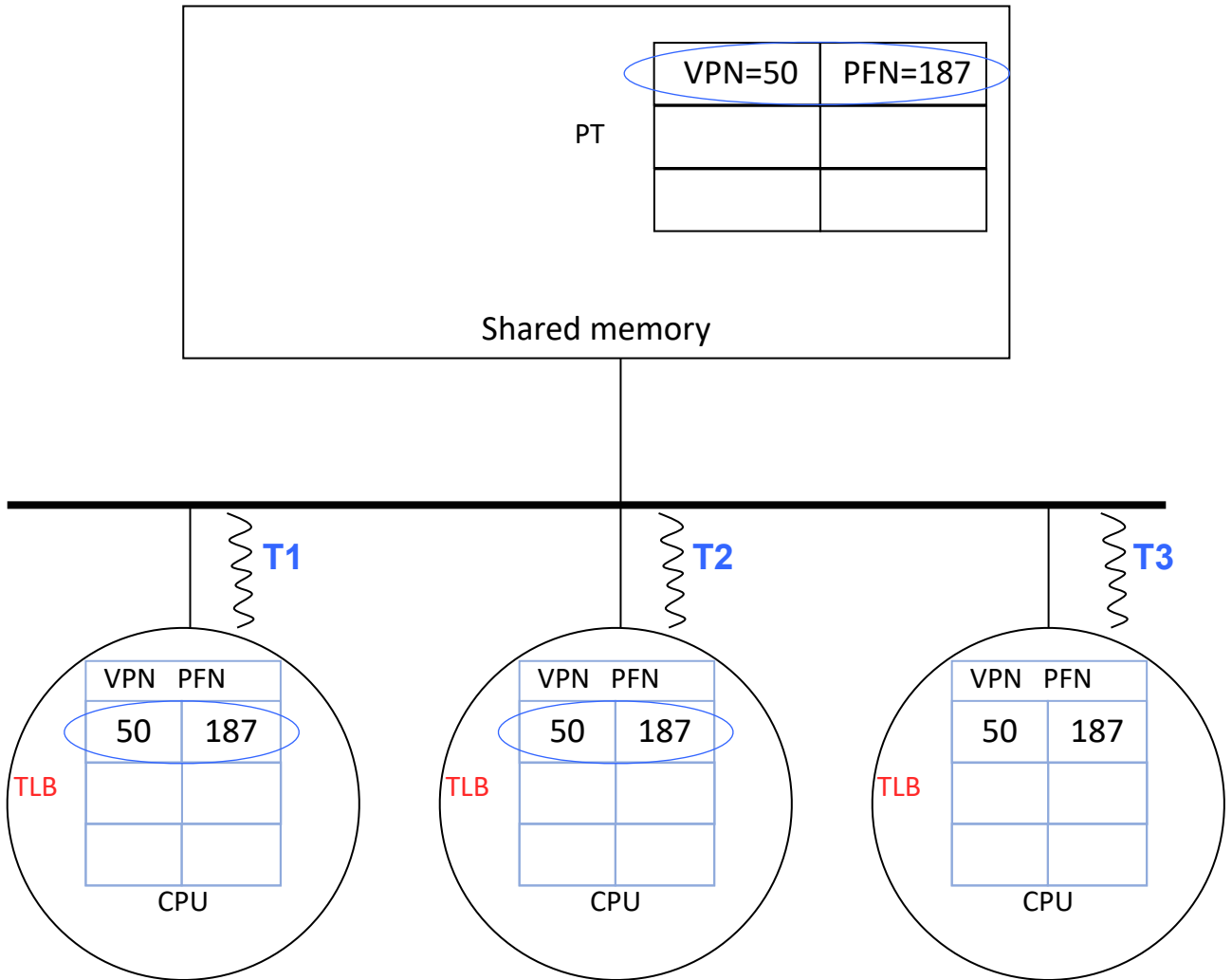
Example: Handling an SMP page fault

Note that the TLBs have each pulled in VPN=50 because each of the threads referenced that page



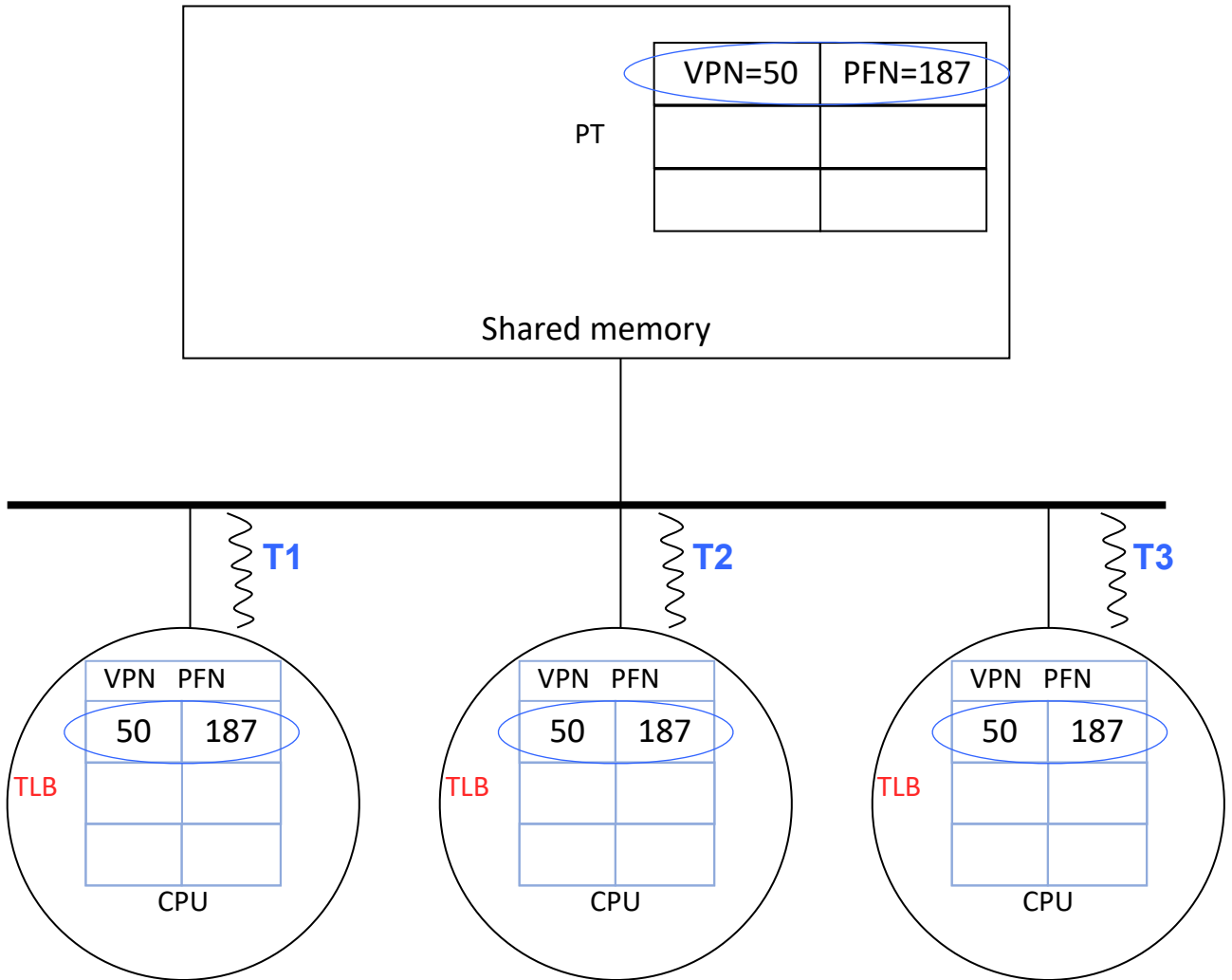
Example: Handling an SMP page fault

Note that the TLBs have each pulled in VPN=50 because each of the threads referenced that page



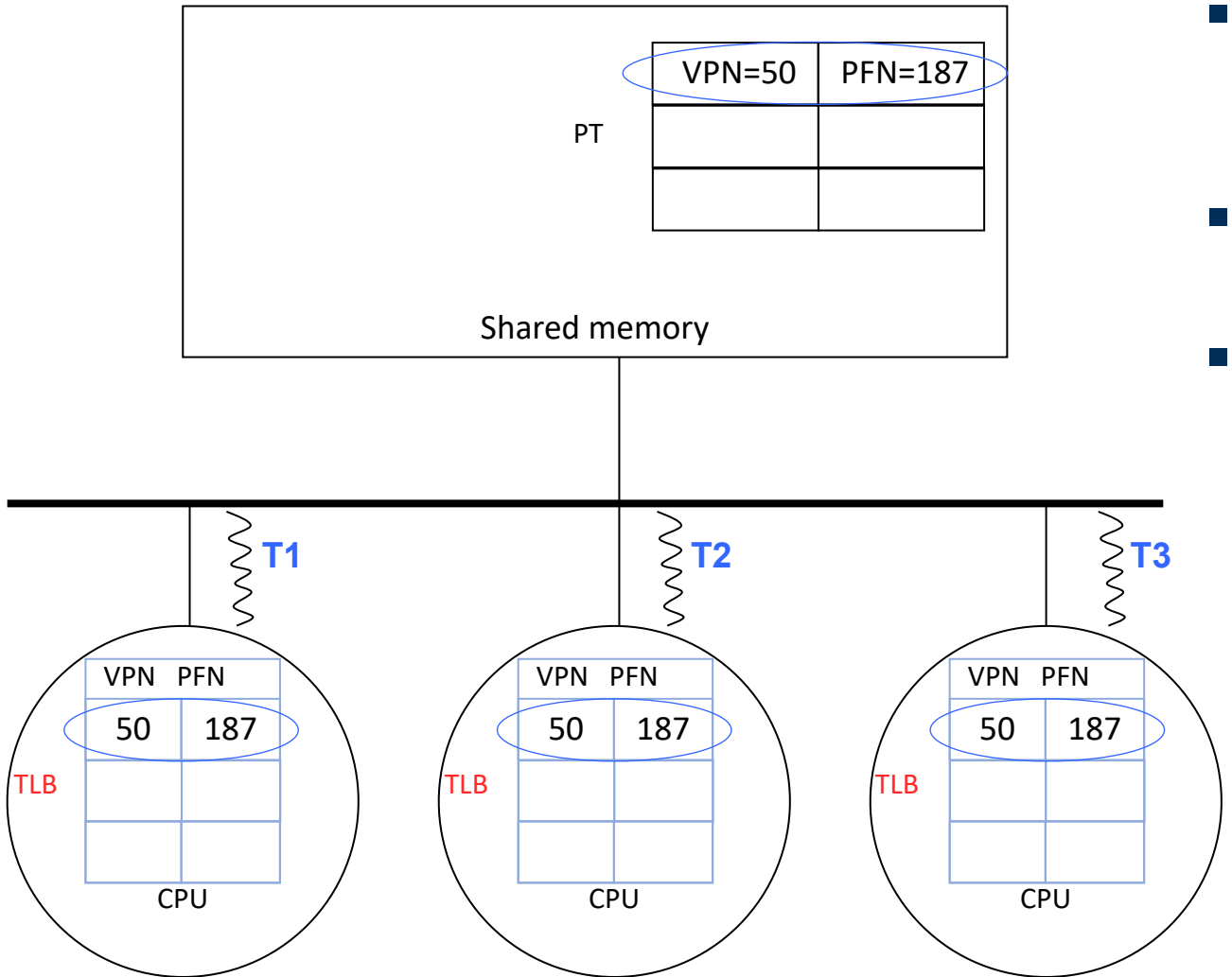
Example: Handling an SMP page fault

Note that the TLBs have each pulled in VPN=50 because each of the threads referenced that page



Example: Handling an SMP page fault

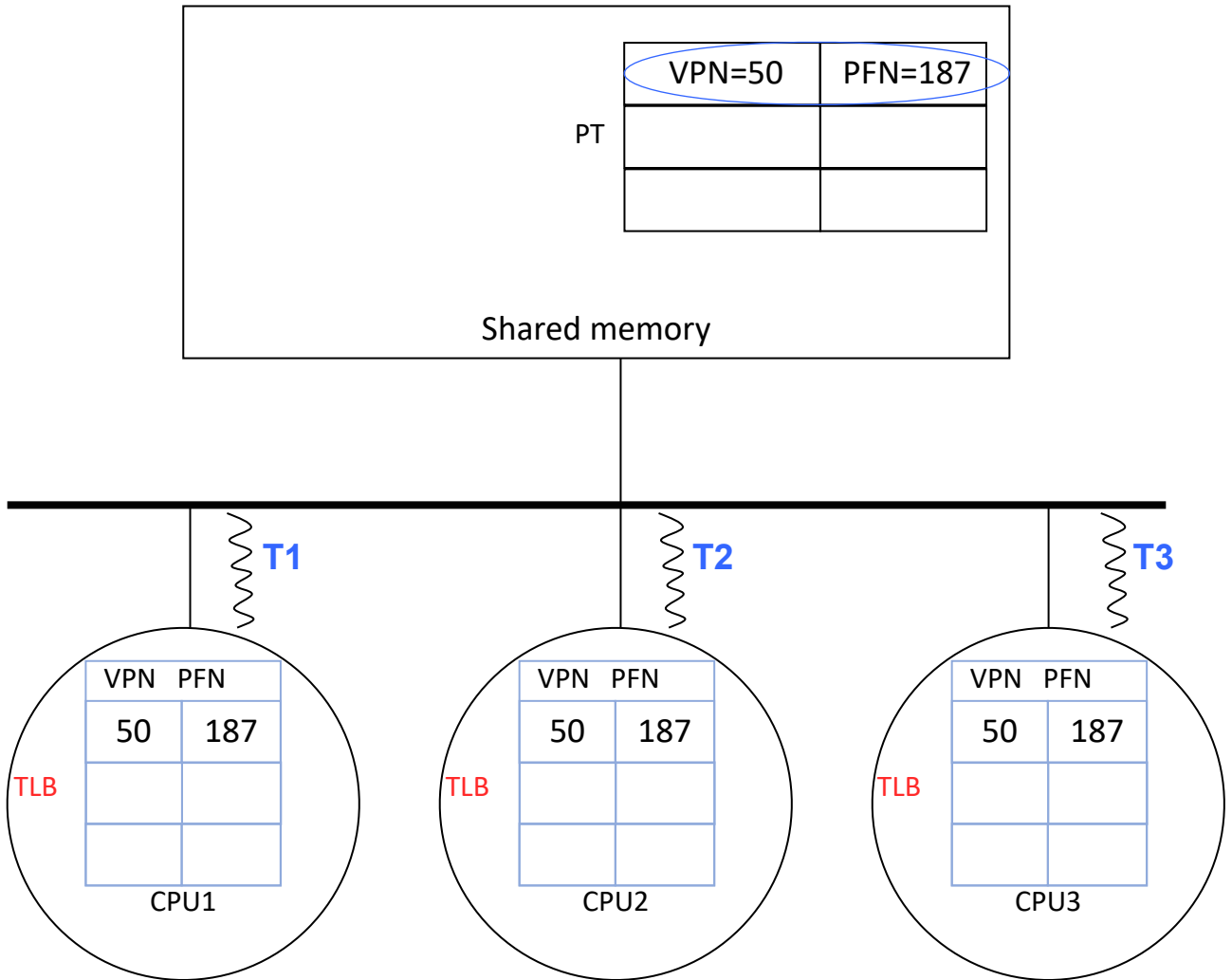
Note that the TLBs have each pulled in VPN=50 because each of the threads referenced that page



- Assume
  - T1 encounters a page fault on VPN=48
  - OS decides to evict VPN=50
  - And use PFN=187 for hosting VPN=48

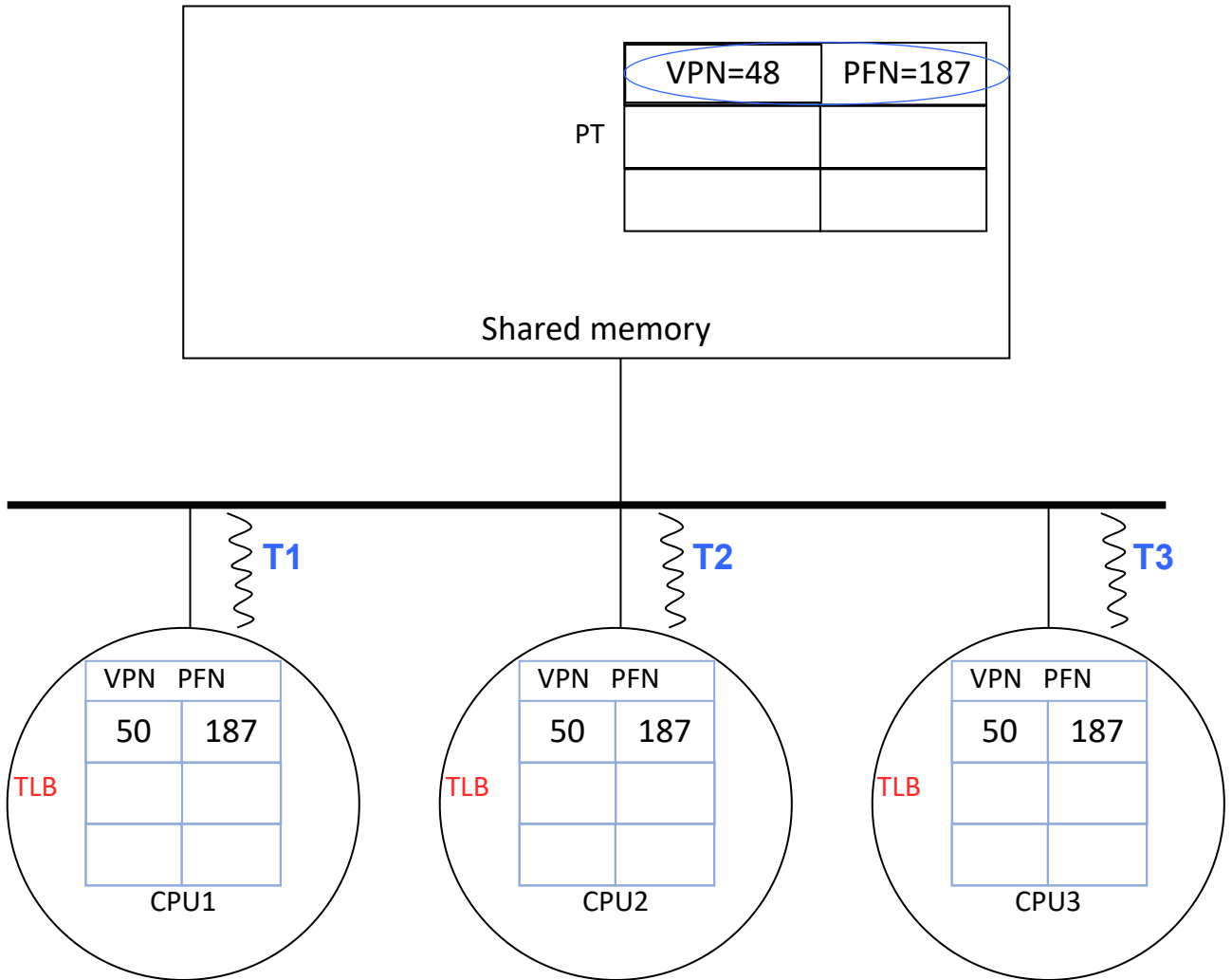
Example: Handling an SMP page fault

OS changes the page table entry for VPN=50



Example: Handling an SMP page fault

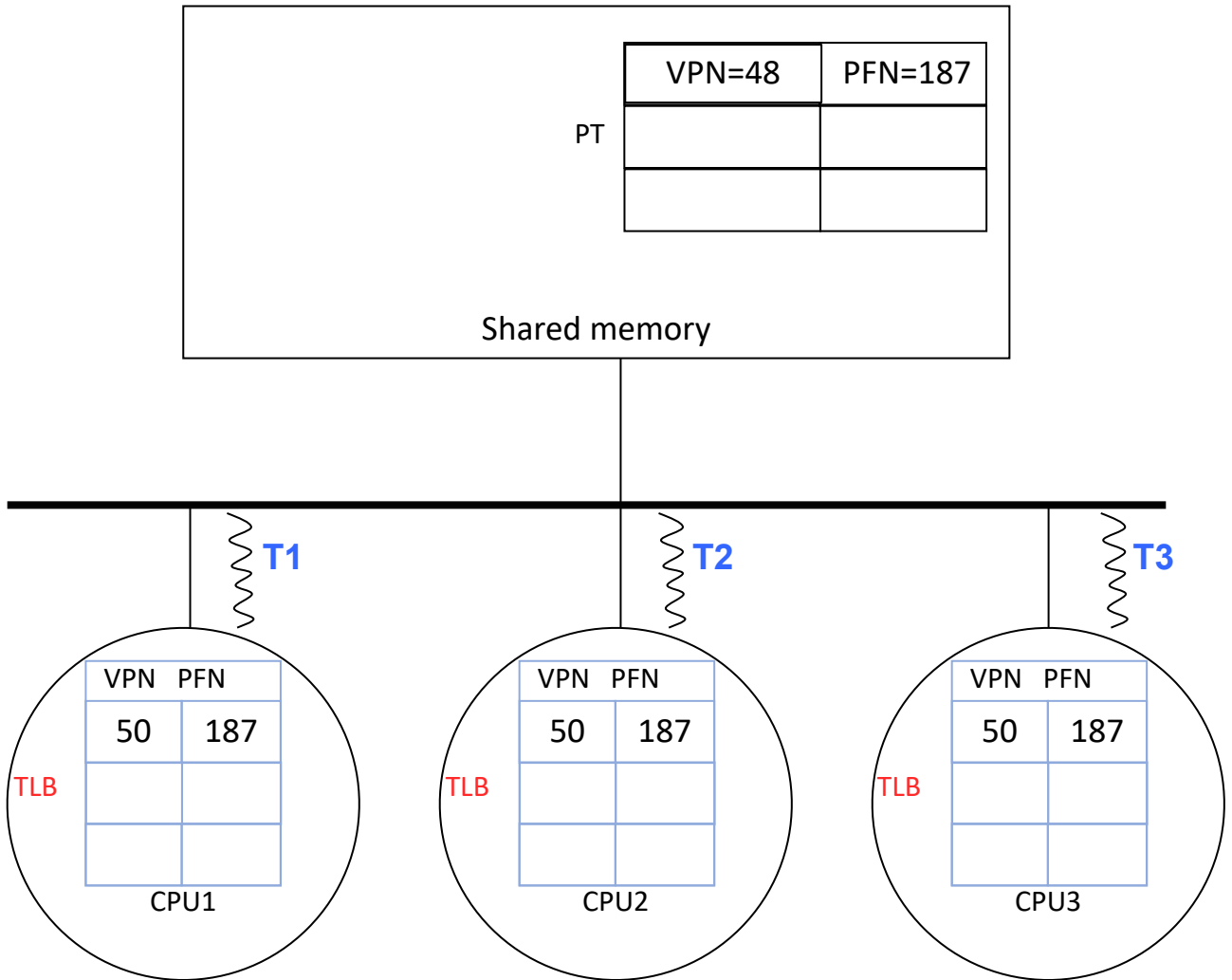
OS changes the page table entry for VPN=50



Example: Handling an SMP page fault

OS changes the page table entry for VPN=50

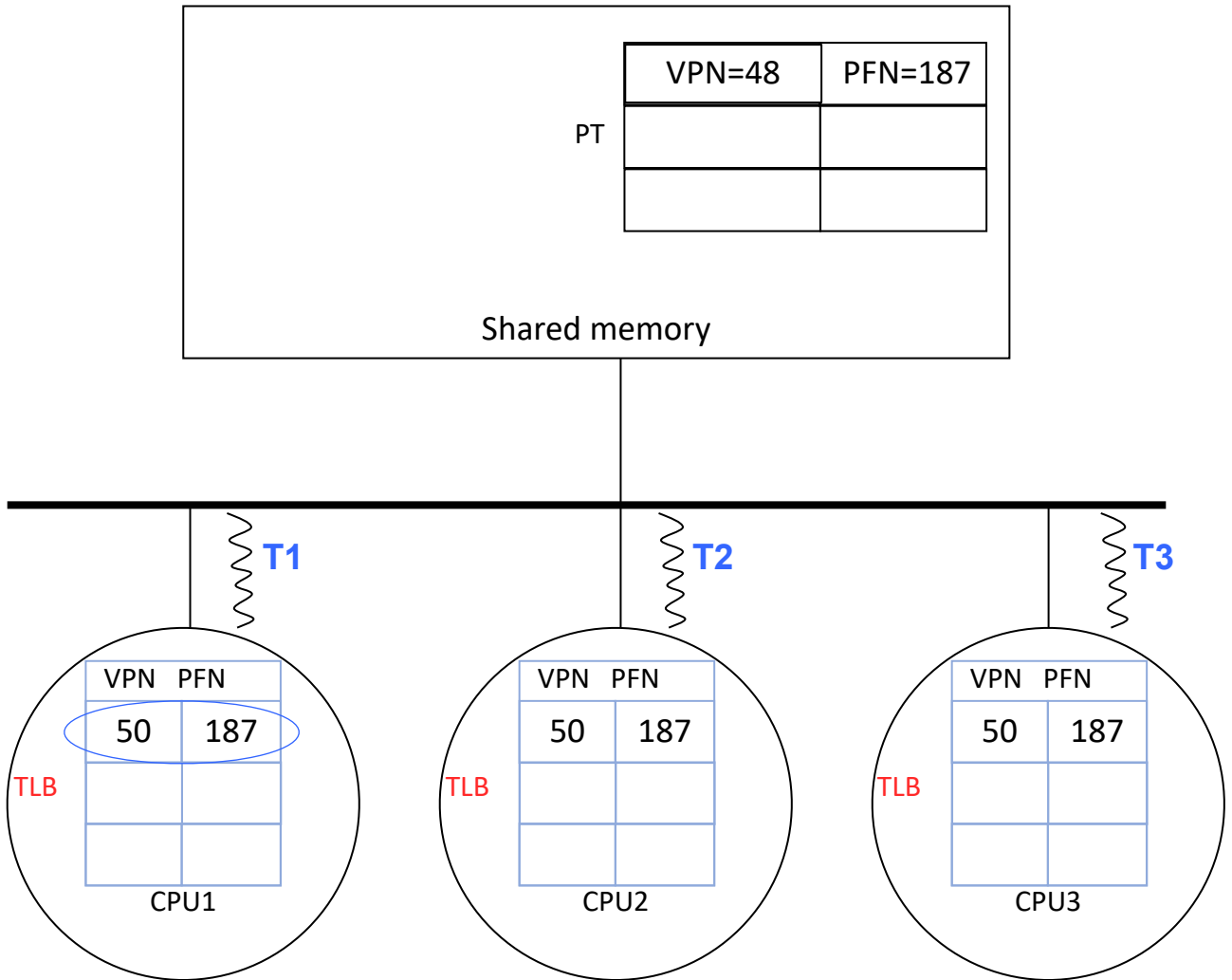
Then because it's running on CPU1, it evicts the TLB entry for VPN 50



Example: Handling an SMP page fault

OS changes the page table entry for VPN=50

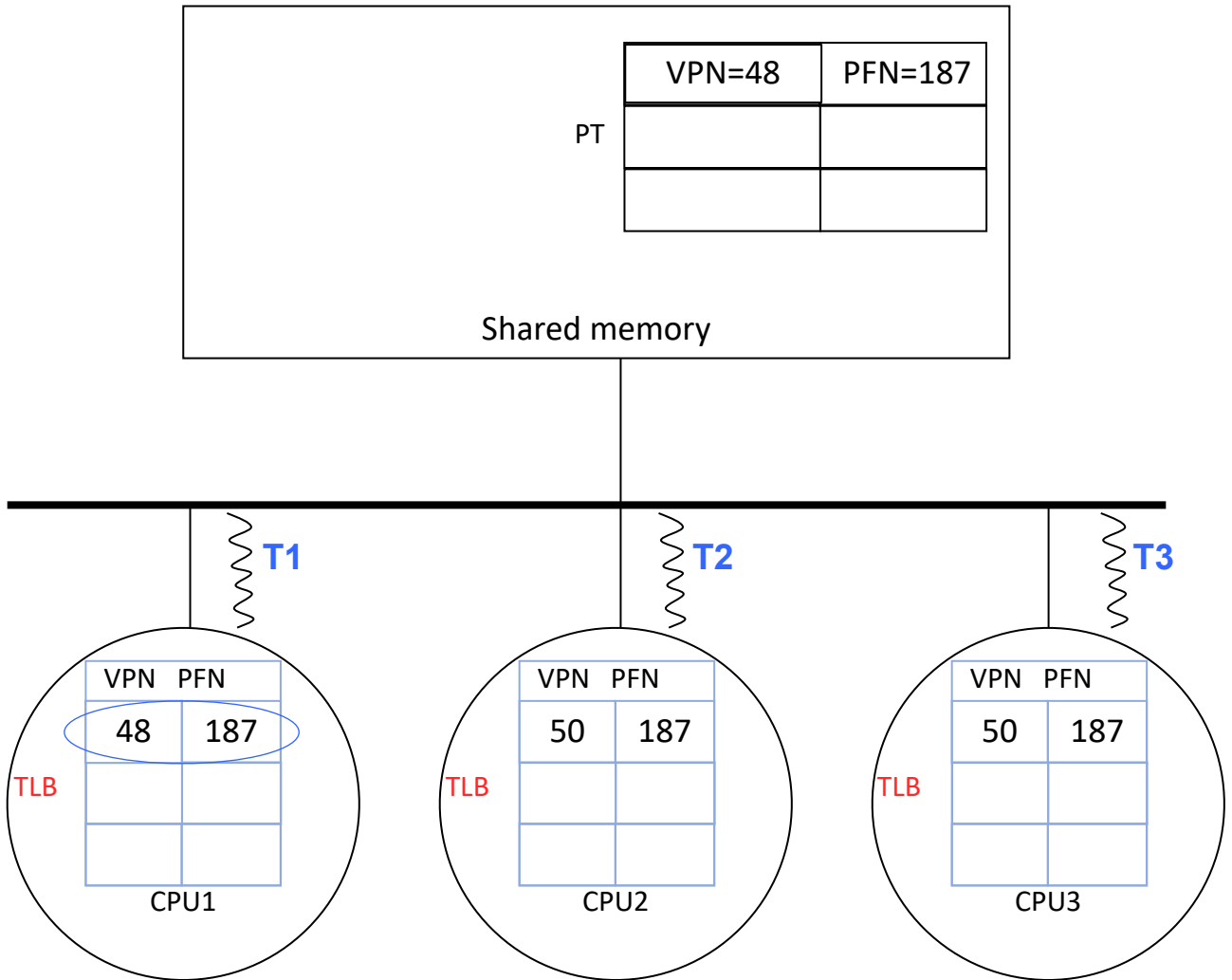
Then because it's running on CPU1, it evicts the TLB entry for VPN 50



Example: Handling an SMP page fault

OS changes the page table entry for VPN=50

Then because it's running on CPU1, it evicts the TLB entry for VPN 50

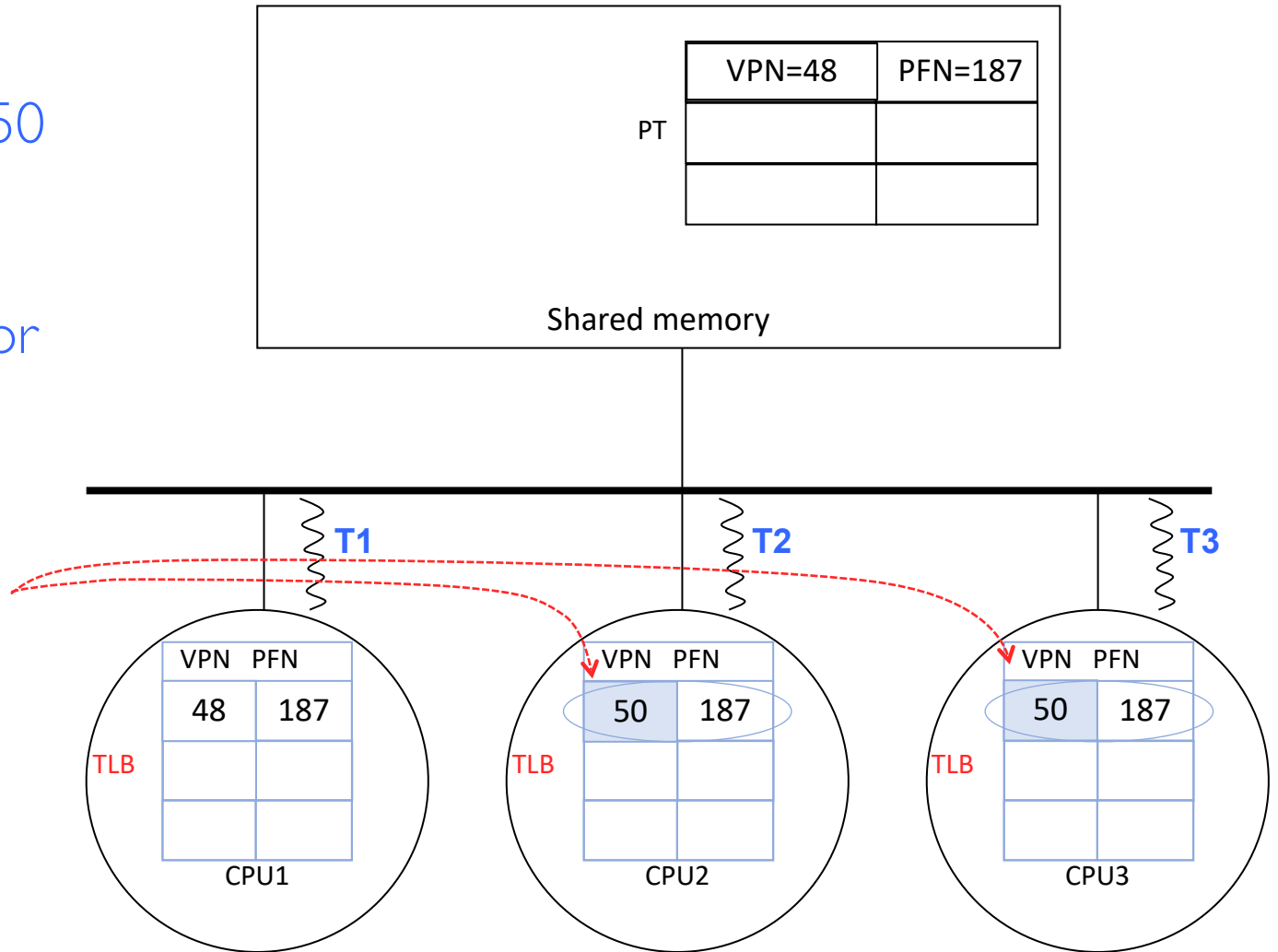


Example: Handling an SMP page fault

OS changes the page table entry for VPN=50

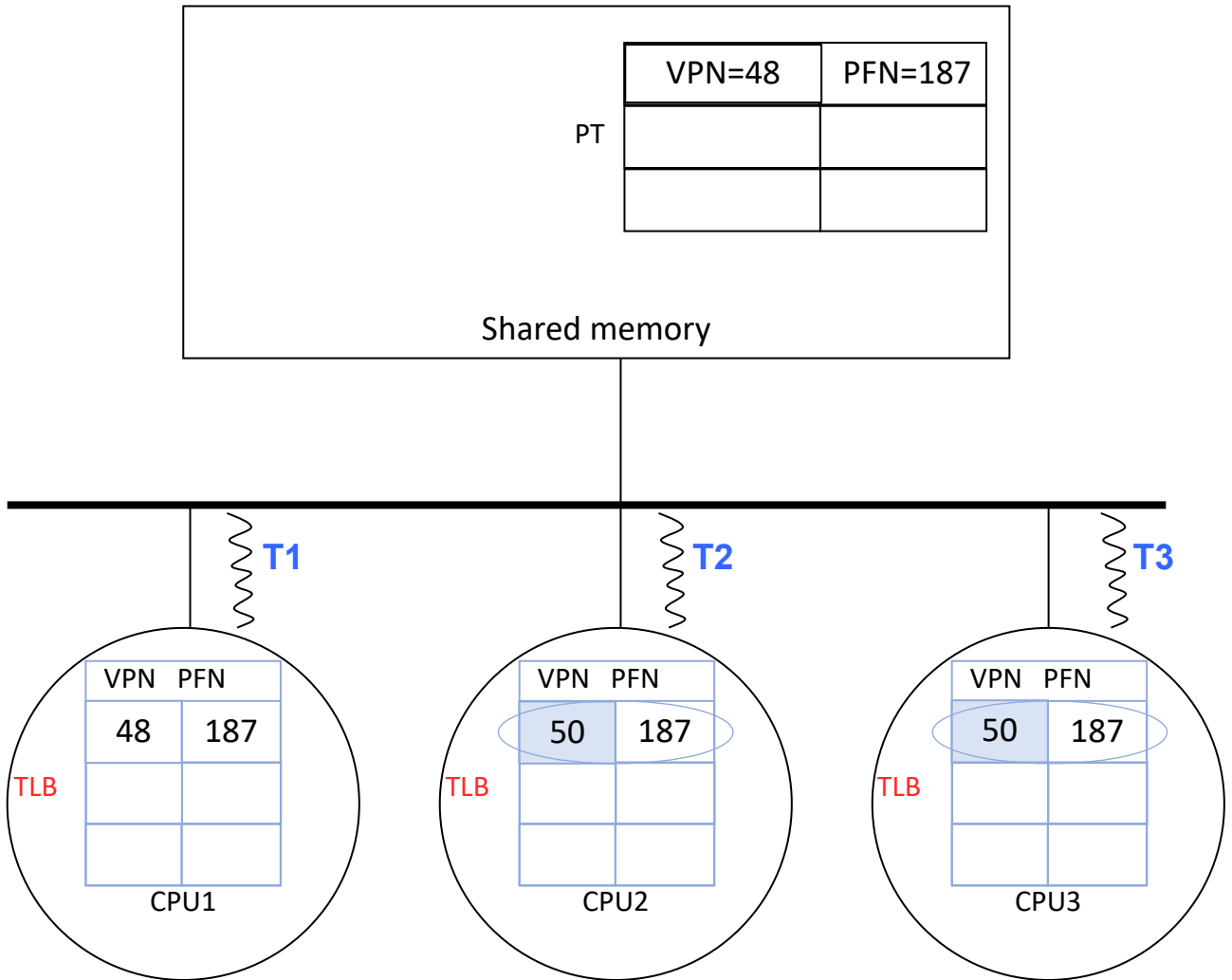
Then because it's running on CPU1, it evicts the TLB entry for VPN 50

Now we've got stale TLB entries in CPU2 and CPU3



Example: Handling an SMP page fault

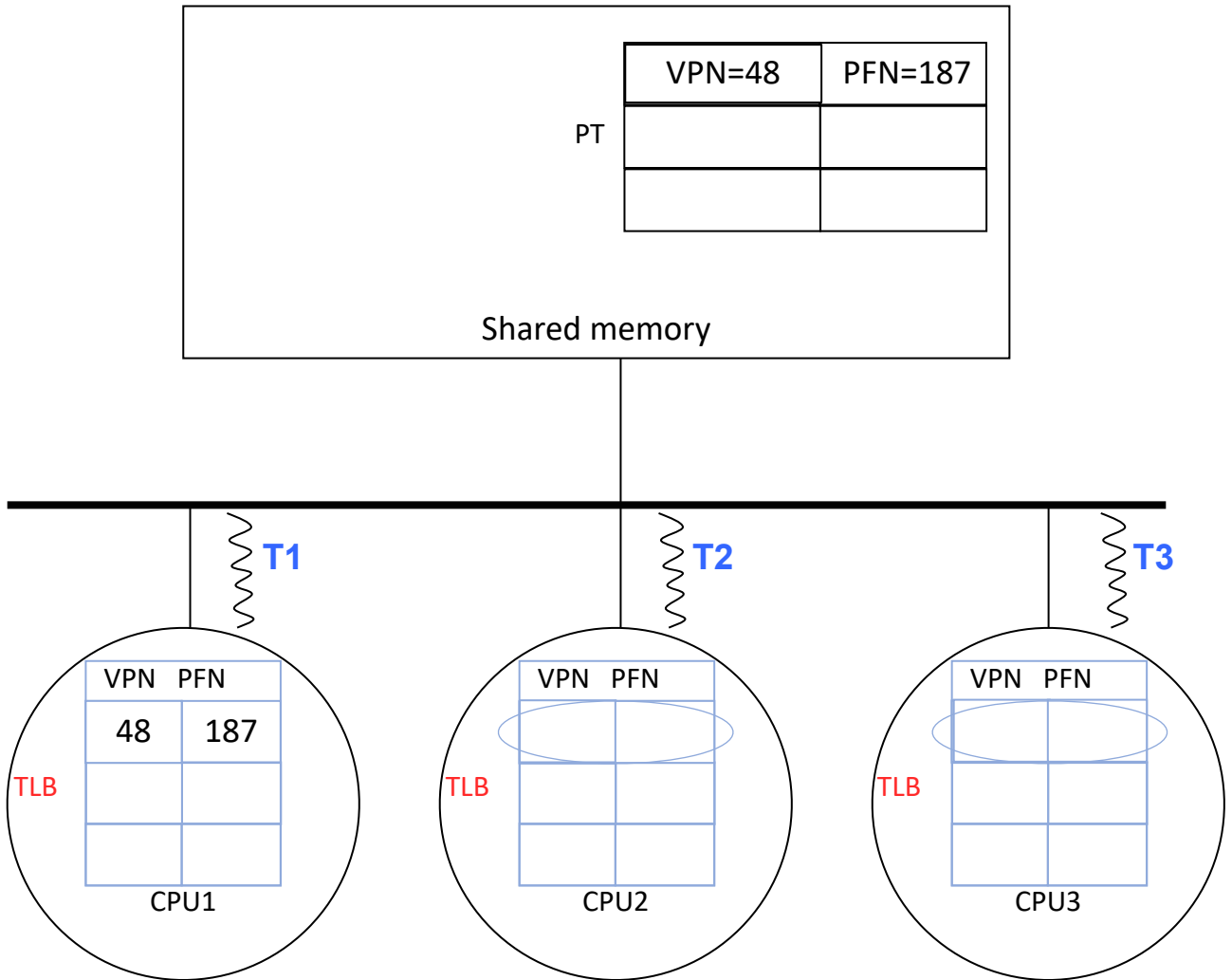
Then we have the TLB Shootdown



Example: Handling an SMP page fault

Then we have the TLB Shootdown

The OS arranges to invalidate the corresponding TLB entries on the other CPUs

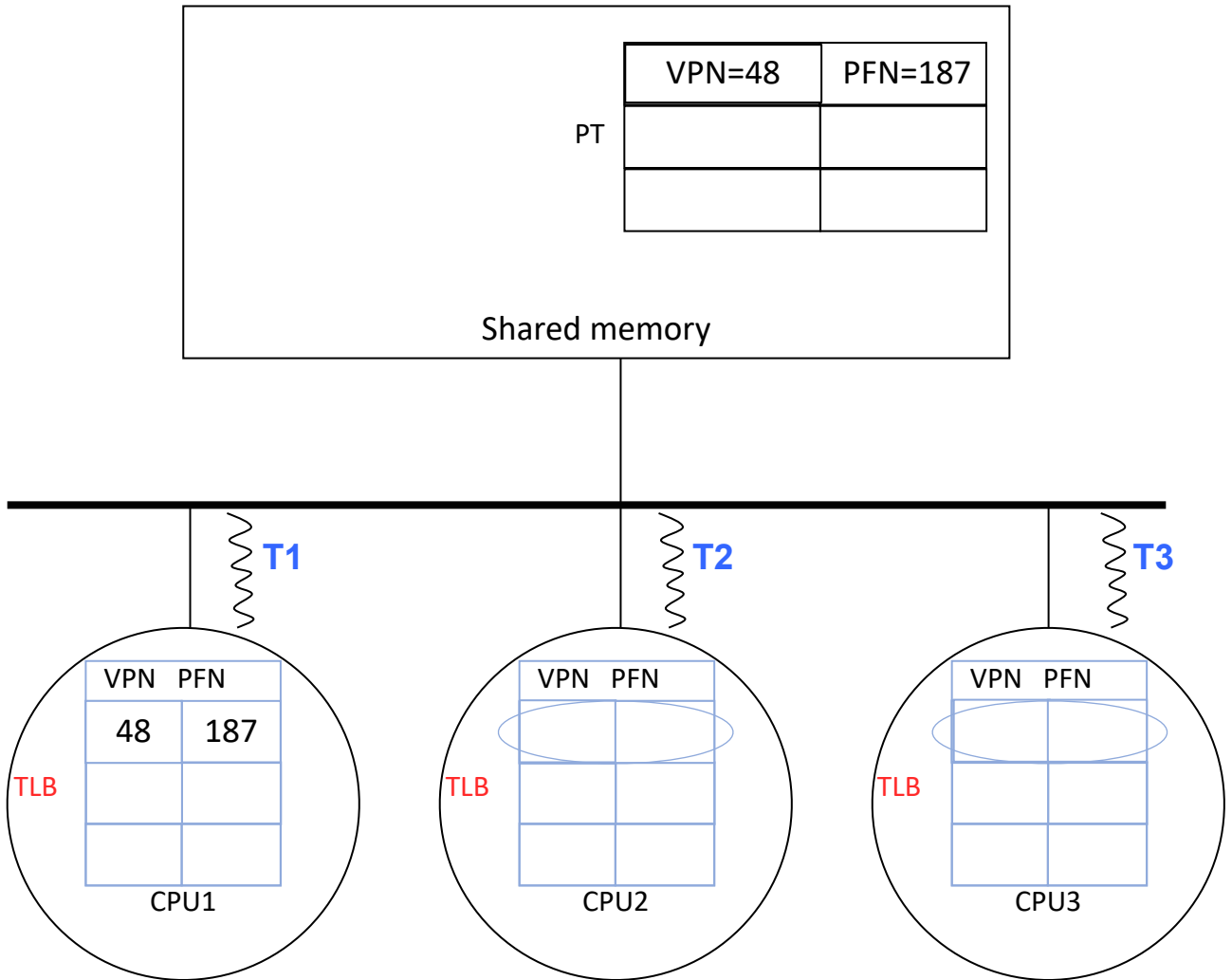


Example: Handling an SMP page fault

Then we have the TLB Shootdown

The OS arranges to invalidate the corresponding TLB entries on the other CPUs

And the CPUs can pull in the updated PTE when (and if) they next reference VPN=48



Session ID:
dag2200

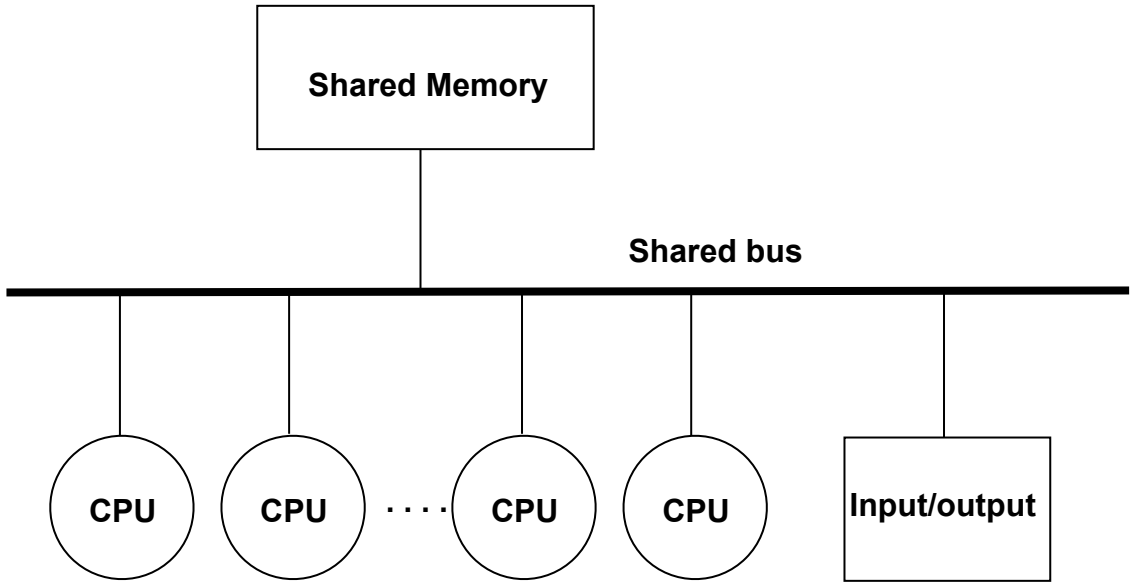
Ensuring that all threads of a process share an address space in an SMP is

A. Impossible
B. Trivially achieved since the page table resides in shared memory
C. Achieved by careful replication of the page table by the operating system for each thread
D. Achieved by special-purpose hardware that no one has told us about yet

Keeping the TLBs consistent in an SMP

A. Is the responsibility of the programmer
B. Is the responsibility of the hardware
C. Is the responsibility of the operating system
D. Is not possible

How do we implement Symmetric Multi Processing (SMP)?



The System (hardware+OS) has to ensure 3 things:

1. Threads of the same process share the same PT
2. Threads have synchronization atomicity
3. Threads have identical views of memory

2) Threads have synchronization atomicity

2) Threads have synchronization atomicity

- We already introduced the TEST-AND-SET instruction

2) Threads have synchronization atomicity

- We already introduced the TEST-AND-SET instruction
- It should be easy on a multiprocessor, right?

2) Threads have synchronization atomicity

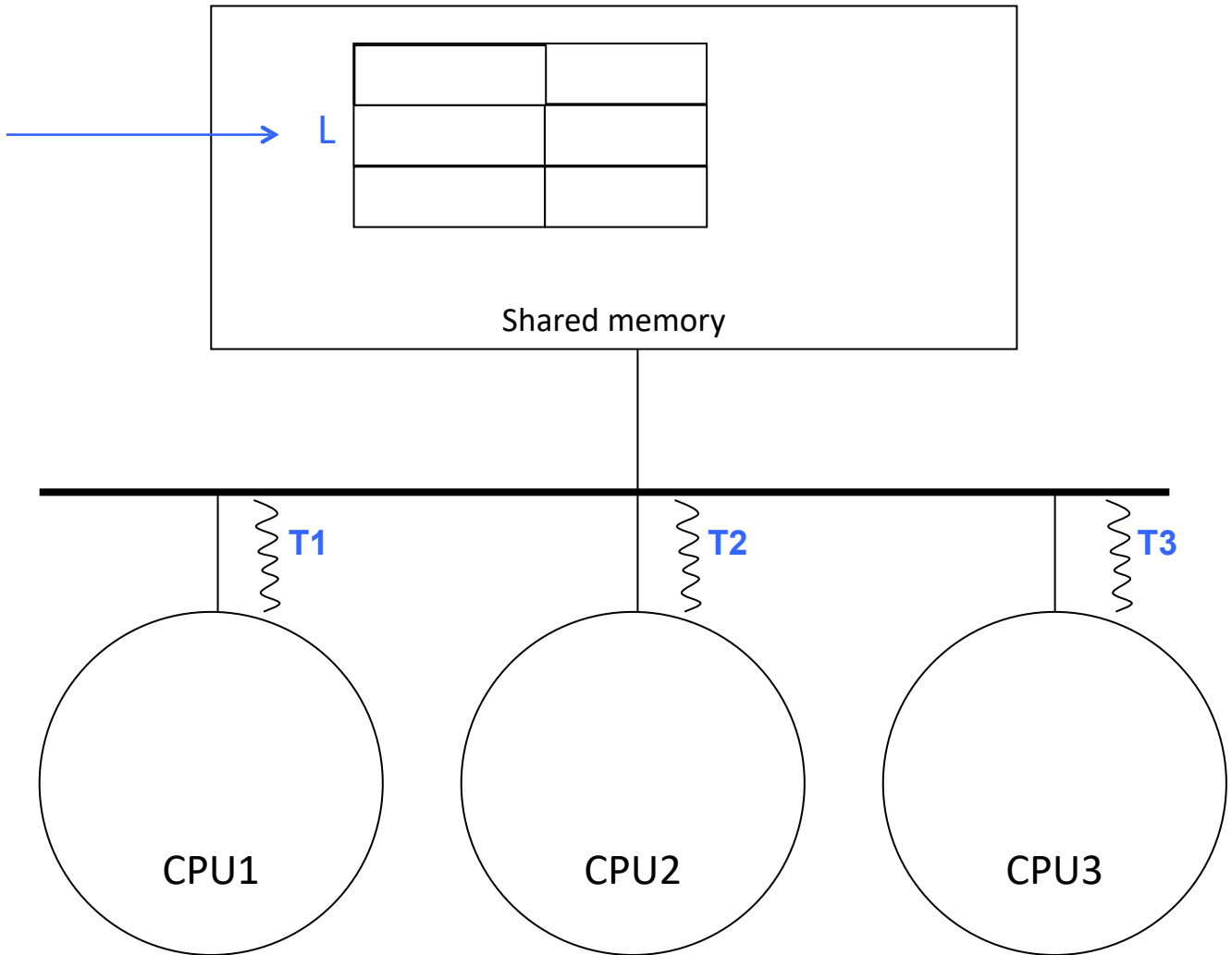
- We already introduced the TEST-AND-SET instruction
- It should be easy on a multiprocessor, right?
- The location we use for synchronization is in shared memory, so no sweat.

2) Threads have synchronization atomicity

- We already introduced the TEST-AND-SET instruction
- It should be easy on a multiprocessor, right?
- The location we use for synchronization is in shared memory, so no sweat.
- What could go wrong?

2) Threads have synchronization atomicity

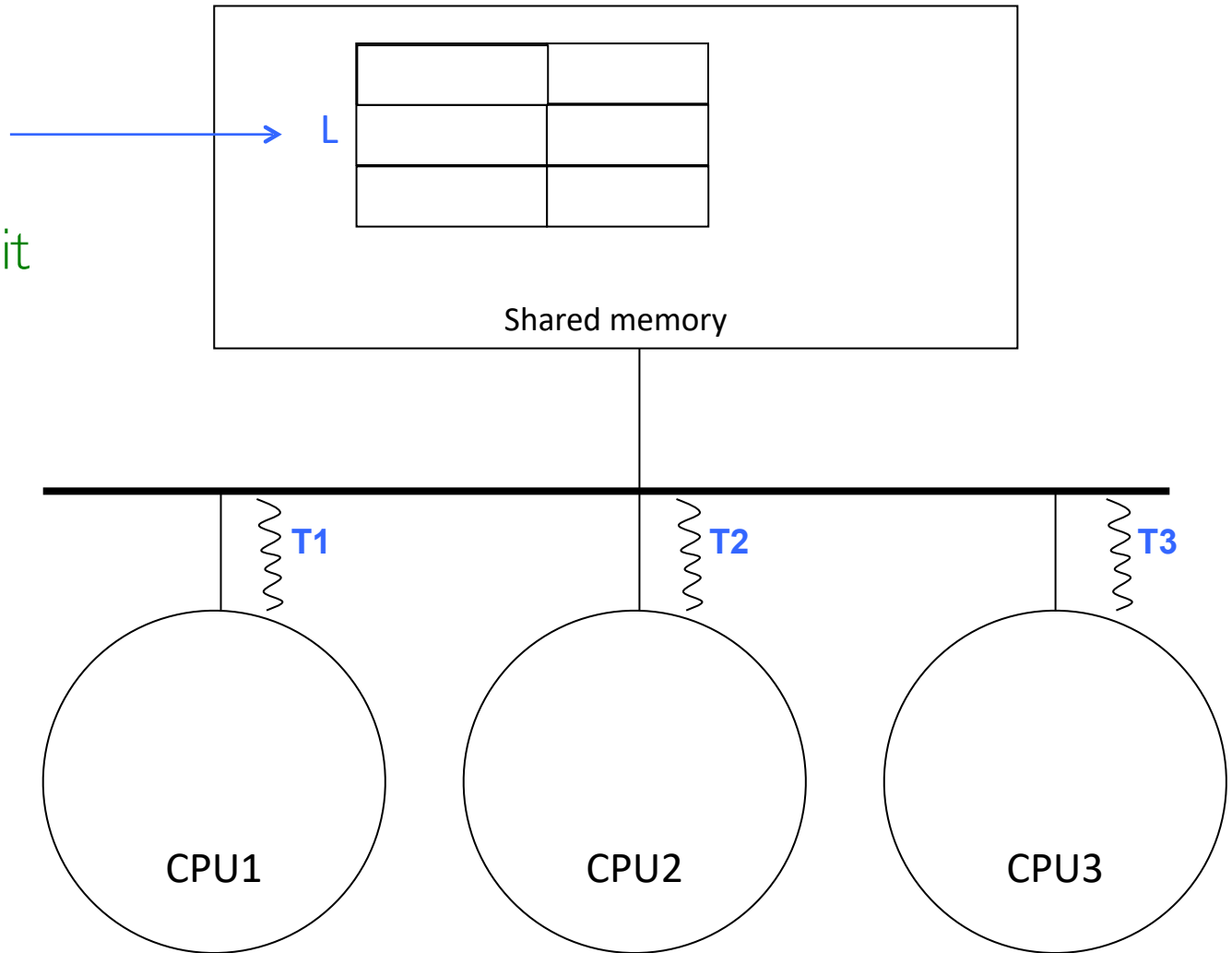
We have memory location L for TEST-AND-SET



2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

Each CPU can access it

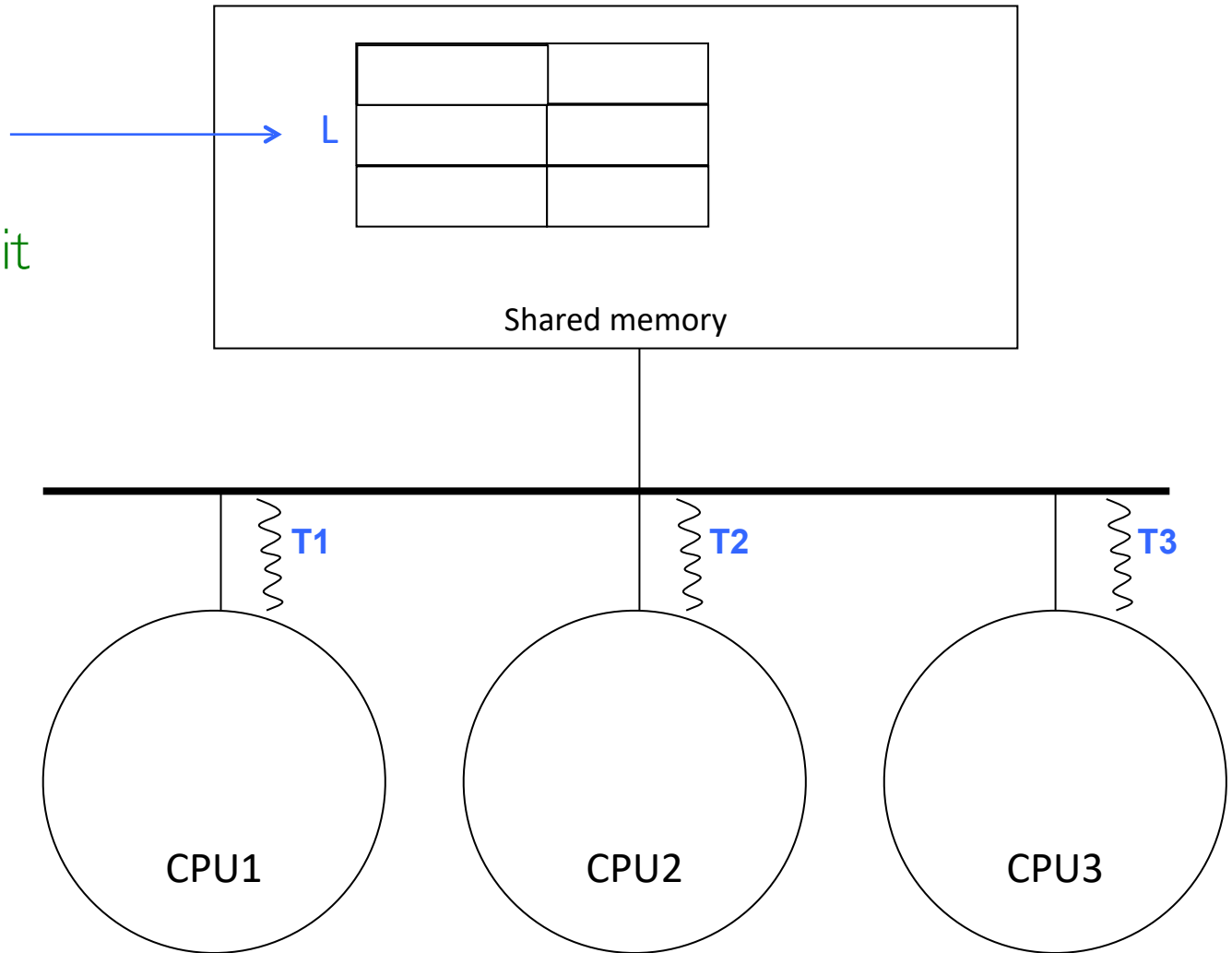


2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

Each CPU can access it

But…what did we forget?



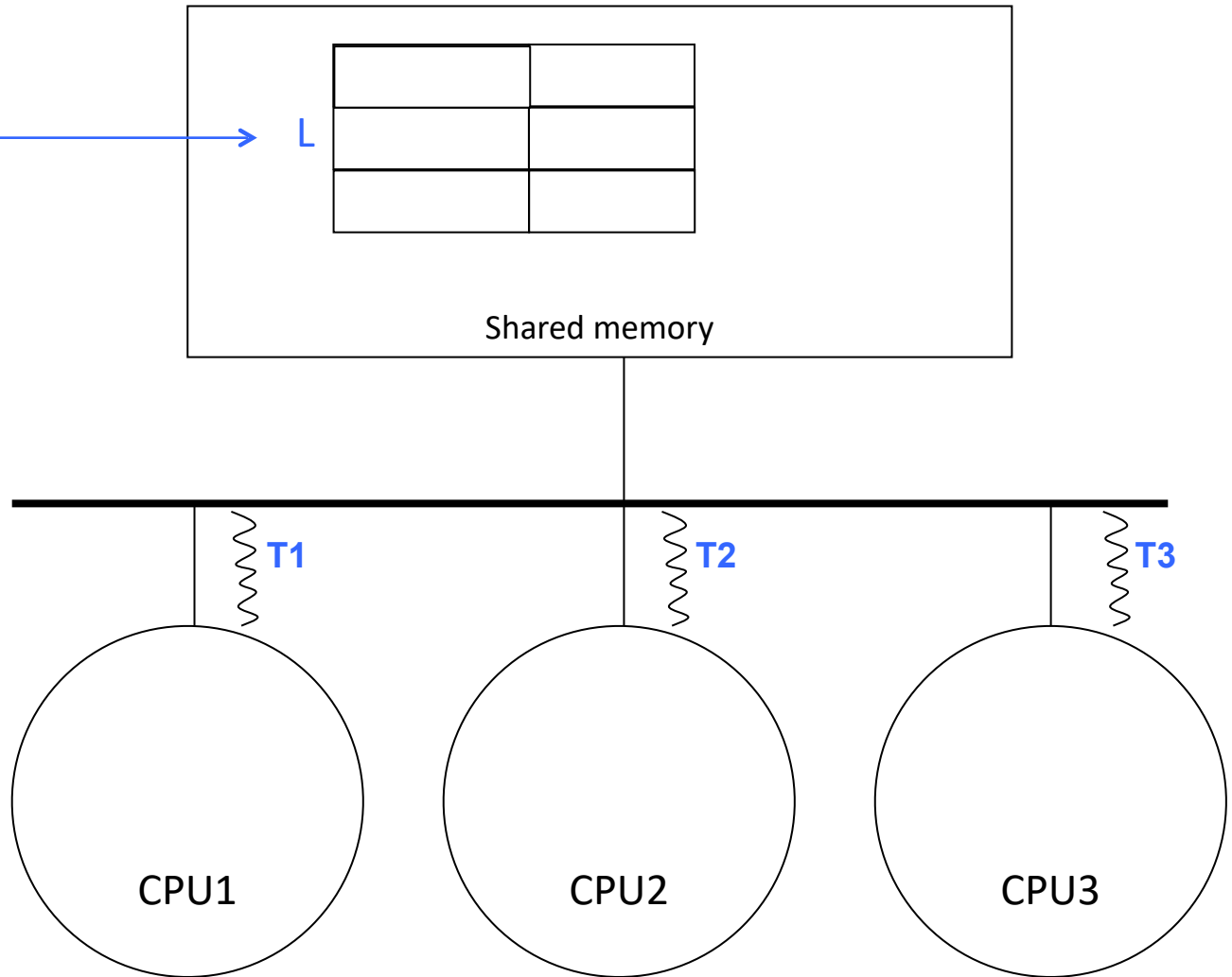
2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

Each CPU can access it

But…what did we forget?

Where are the caches?



2) Threads have synchronization atomicity

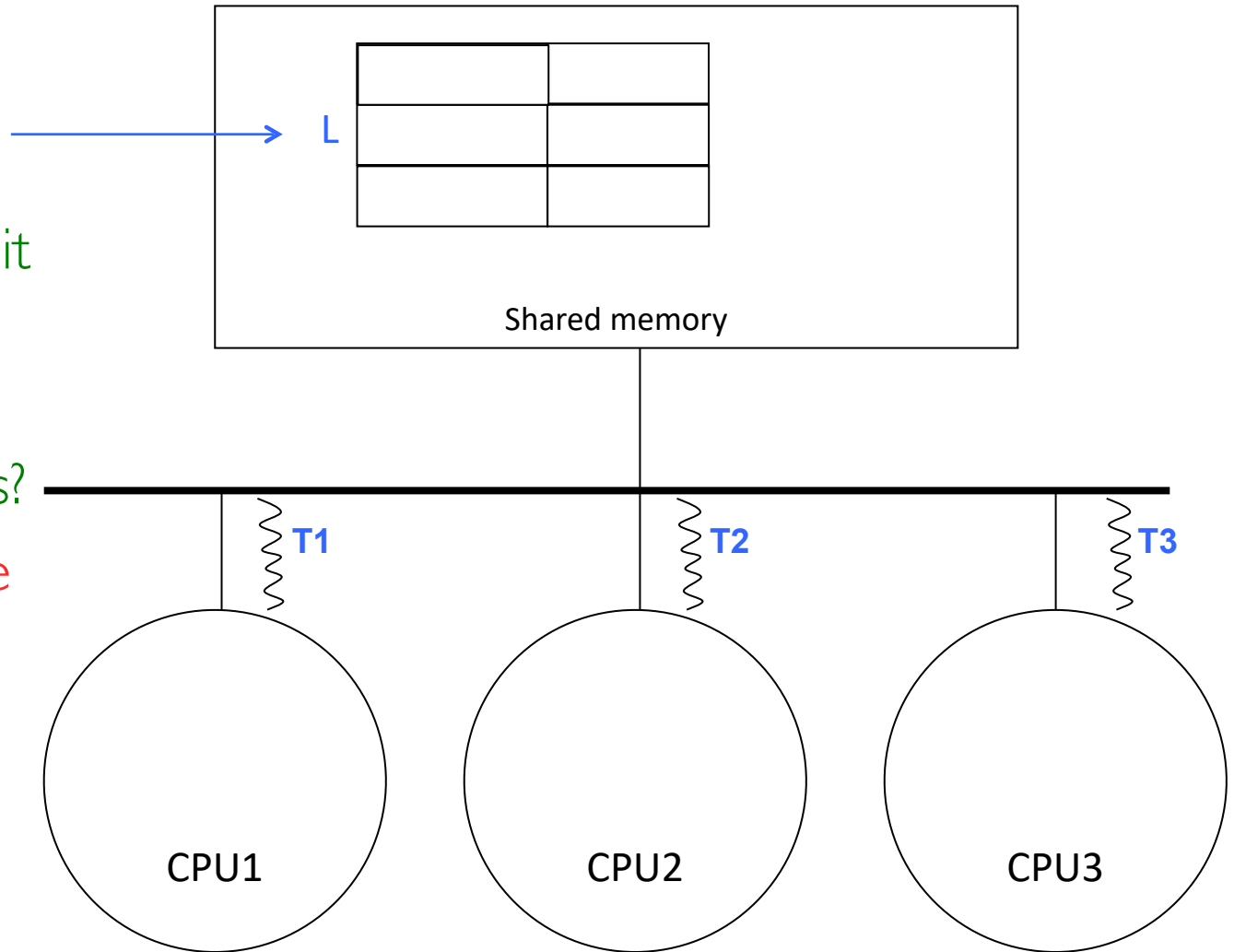
We have memory location L for TEST-AND-SET

Each CPU can access it

But…what did we forget?

Where are the caches?

In the CPUs of course



2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

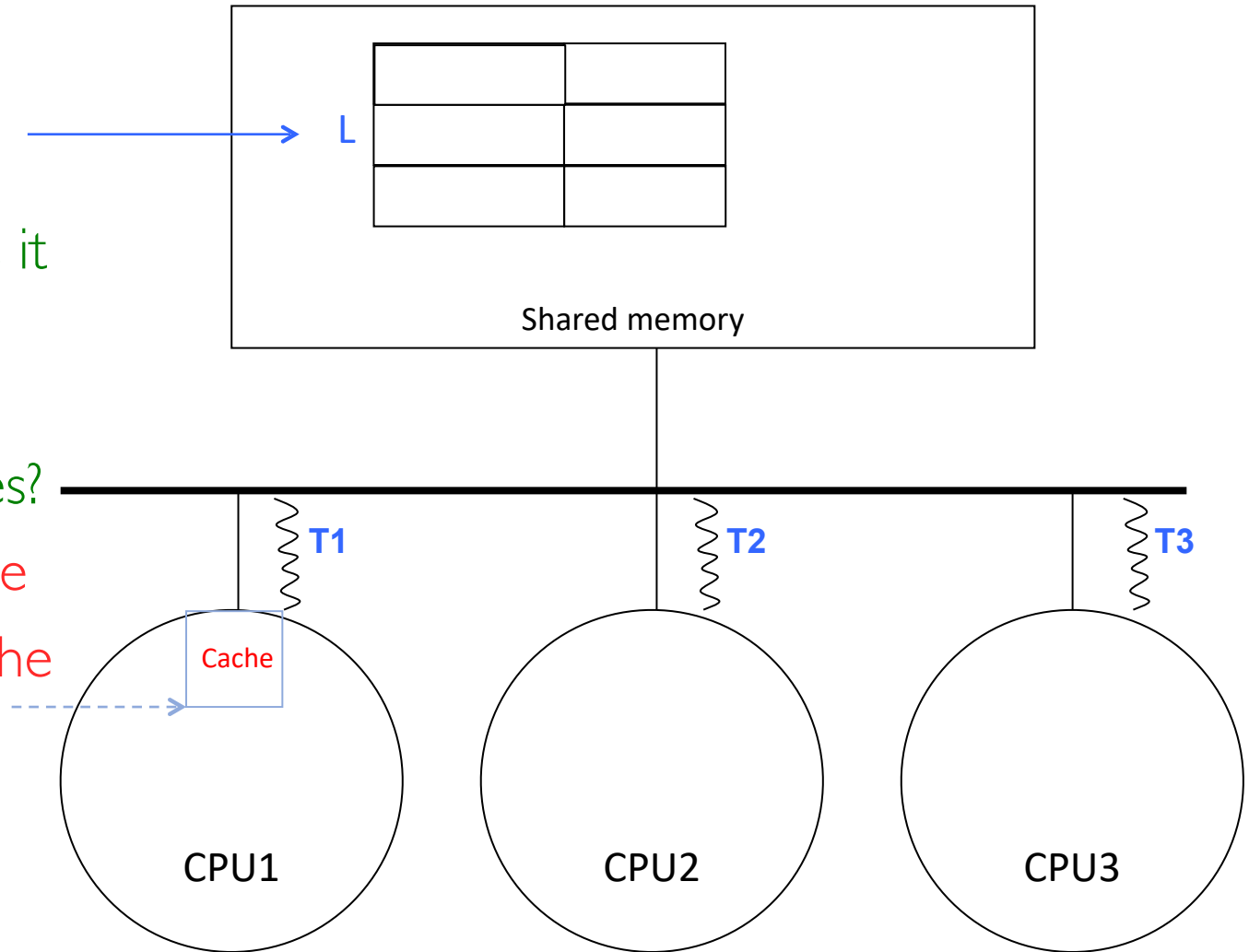
Each CPU can access it

But…what did we forget?

Where are the caches?

In the CPUs of course

If L is cached, the cache is wrong!



2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

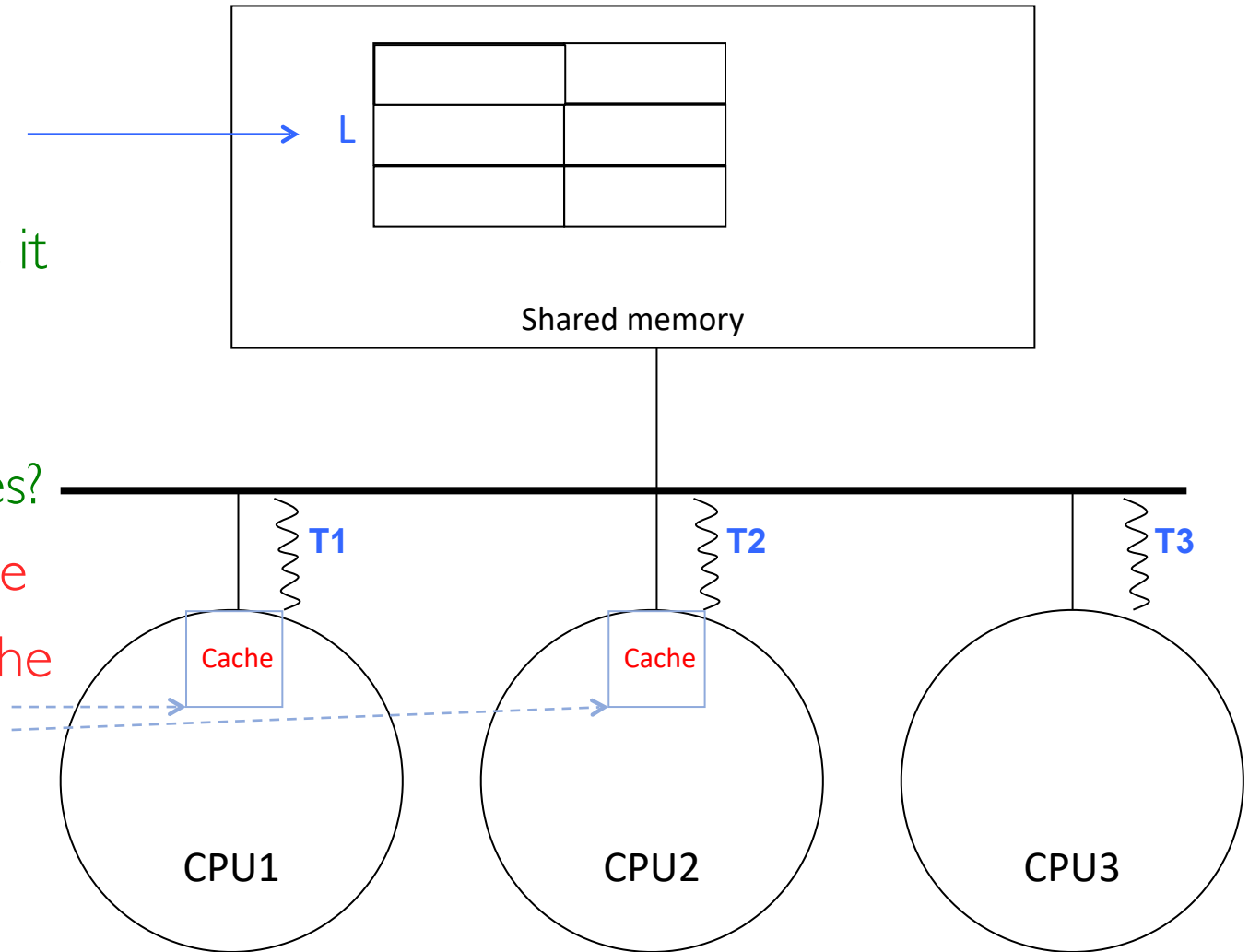
Each CPU can access it

But…what did we forget?

Where are the caches?

In the CPUs of course

If L is cached, the cache is wrong!



2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

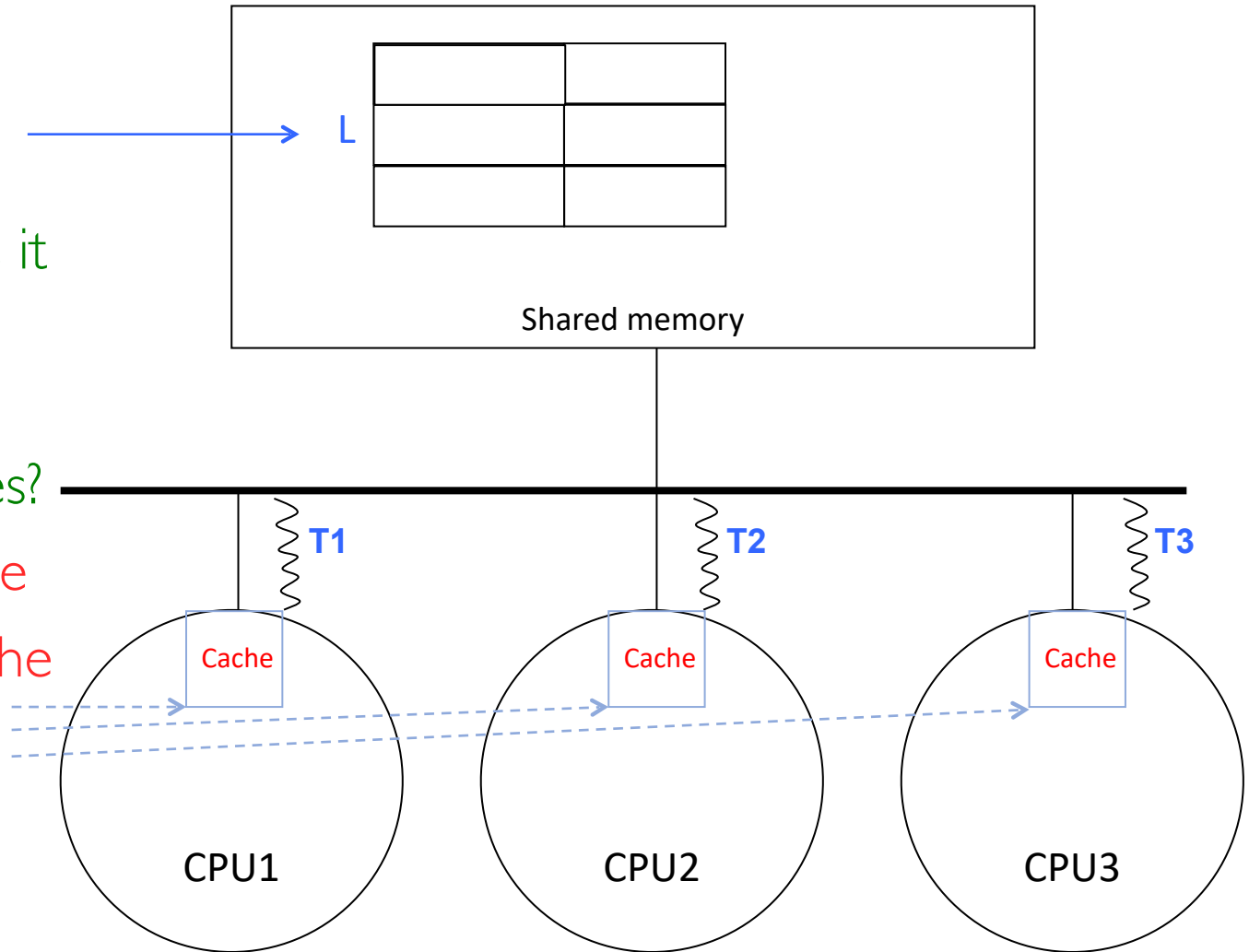
Each CPU can access it

But…what did we forget?

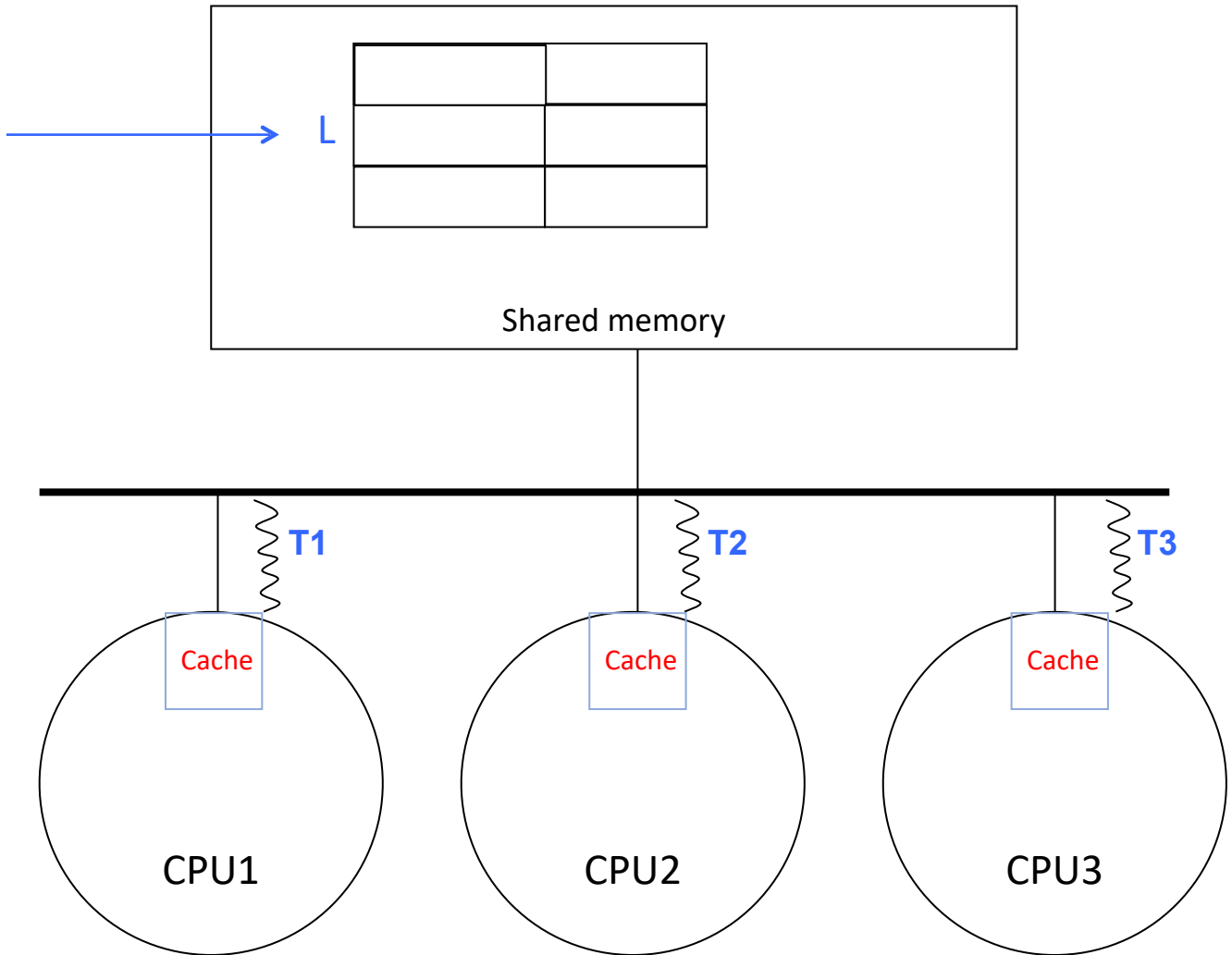
Where are the caches?

In the CPUs of course

If L is cached, the cache is wrong!



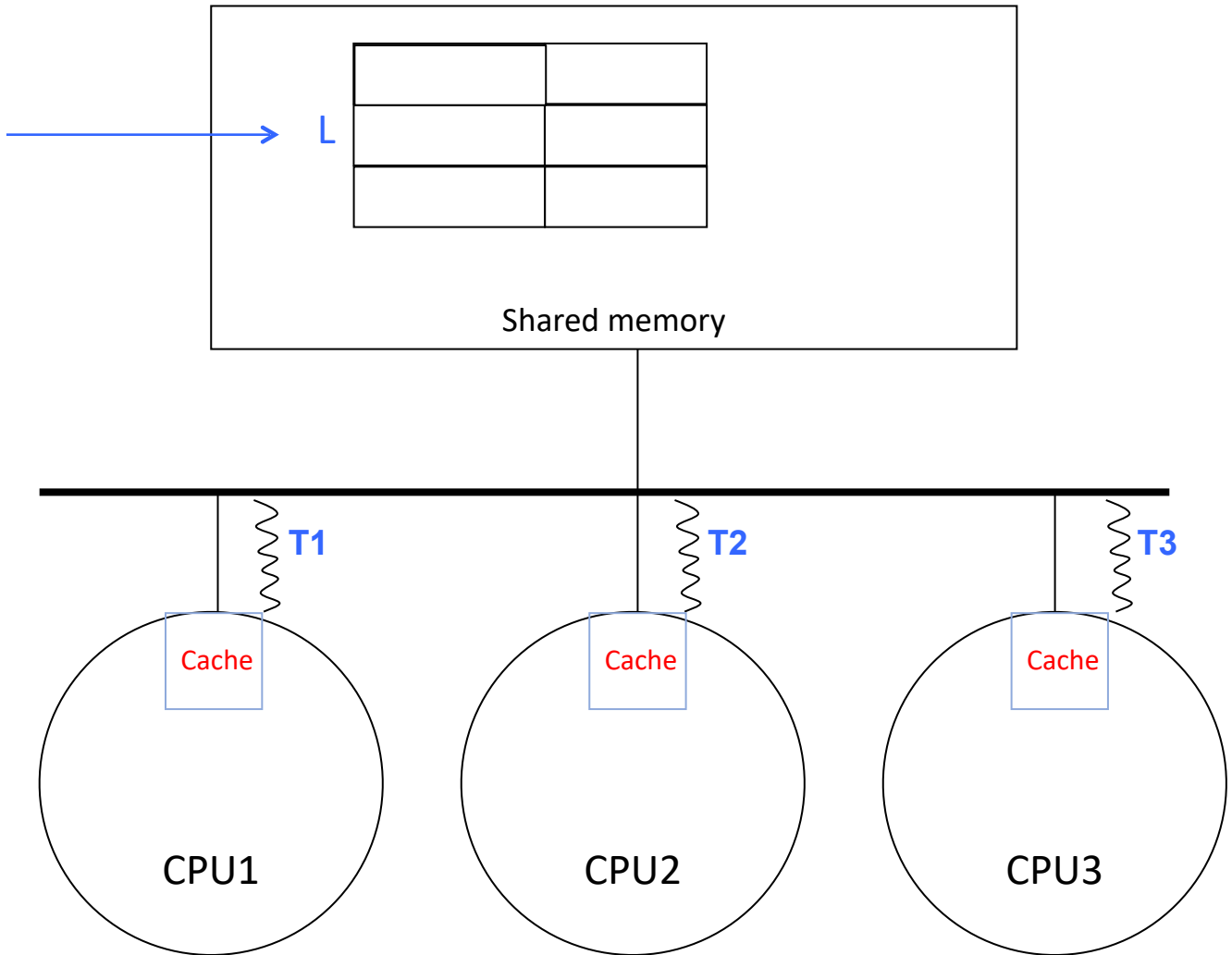
2) Threads have synchronization atomicity



2) Threads have synchronization atomicity

What shall we do?

One solution is to bypass the cache for the T&S instruction



Requirements for SMP

1. ~~Threads of the same process share the same PT~~
2. ~~Threads have synchronization atomicity~~
3. Threads have identical views of memory

Requirements for SMP

1. ~~Threads of the same process share the same PT~~
2. ~~Threads have synchronization atomicity~~
3. Threads have identical views of memory

   ➔ This implies that access to a memory location returns the same value on all CPUs

Requirements for SMP

1. ~~Threads of the same process share the same PT~~
2. ~~Threads have synchronization atomicity~~
3. Threads have identical views of memory
   - ➔ This implies that access to a memory location returns the same value on all CPUs
   - ➔ But caches make copies of data

Requirements for SMP

1. ~~Threads of the same process share the same PT~~
2. ~~Threads have synchronization atomicity~~
3. Threads have identical views of memory
   - ➔ This implies that access to a memory location returns the same value on all CPUs
   - ➔ But caches make copies of data
   - ➔ We'll refer to the method of keeping all the copies of the same data across caches as a cache coherence protocol

3) Threads have identical views of memory

- Two possible solutions, in hardware

3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus

3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'

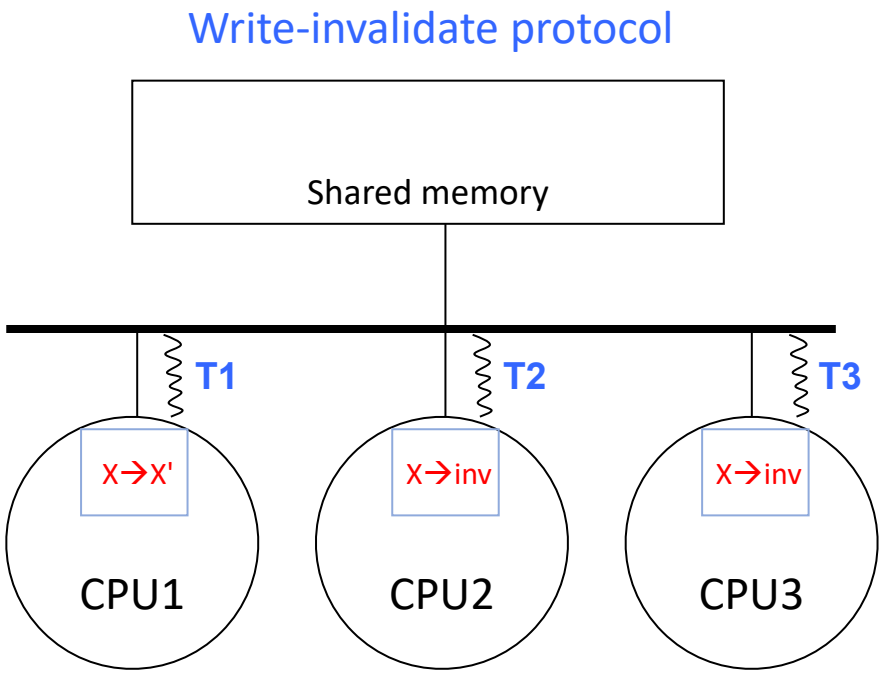
3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'

Write-invalidate protocol

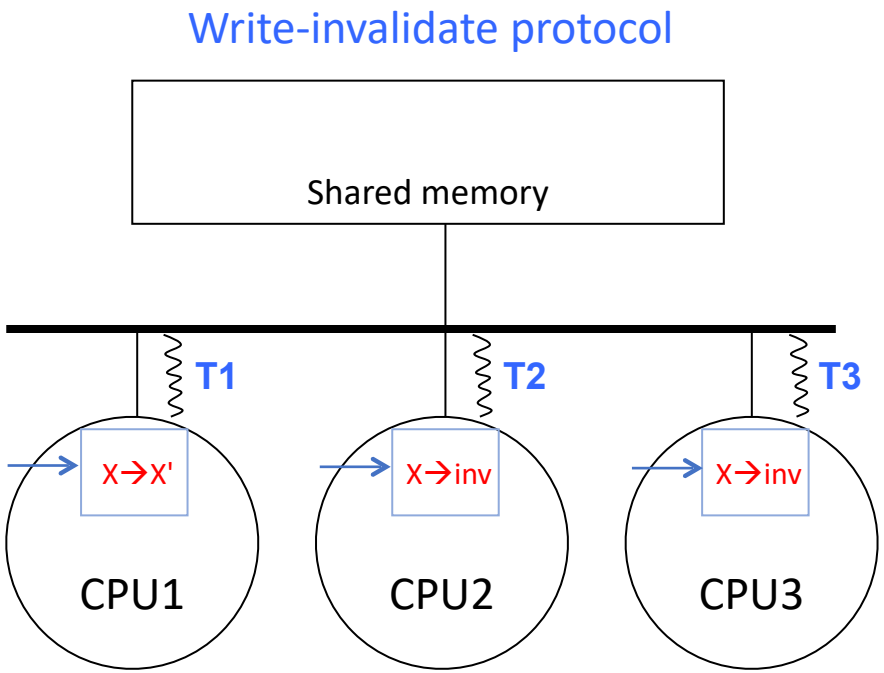
3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'



3) Threads have identical views of memory

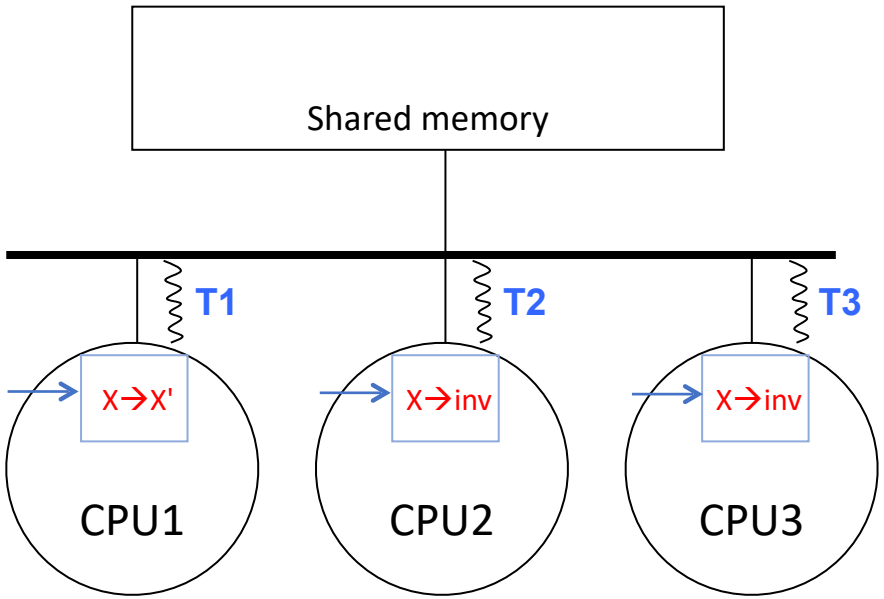
- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'



3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'

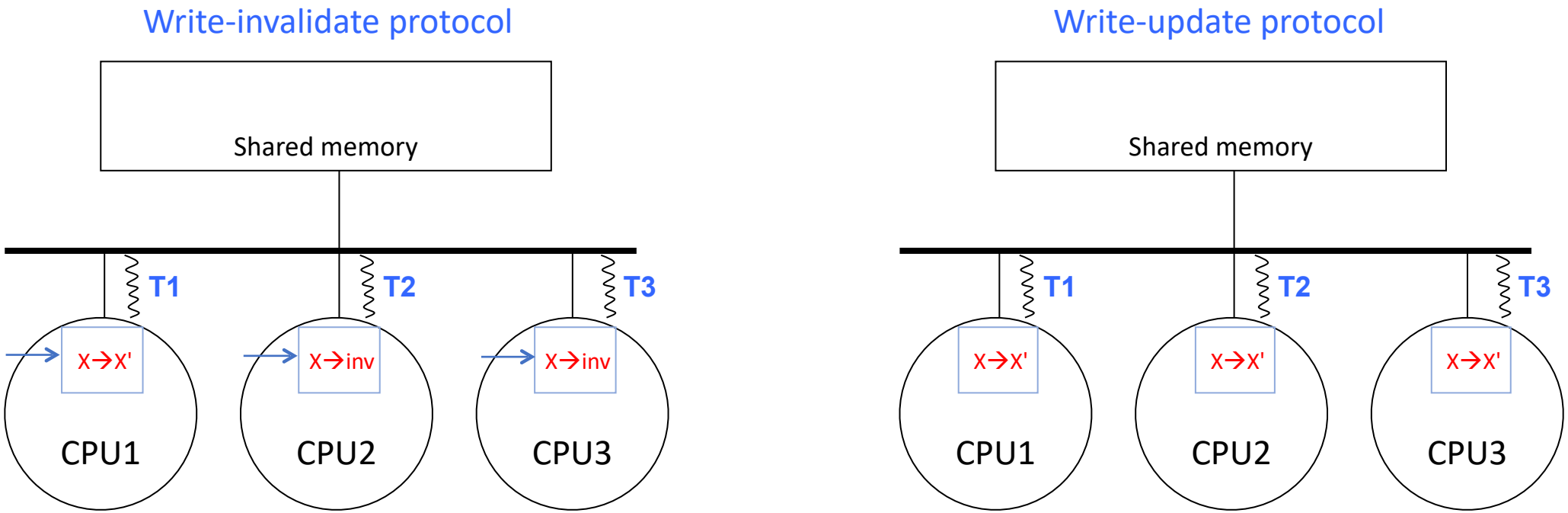
Write-invalidate protocol



Write-update protocol

3) Threads have identical views of memory

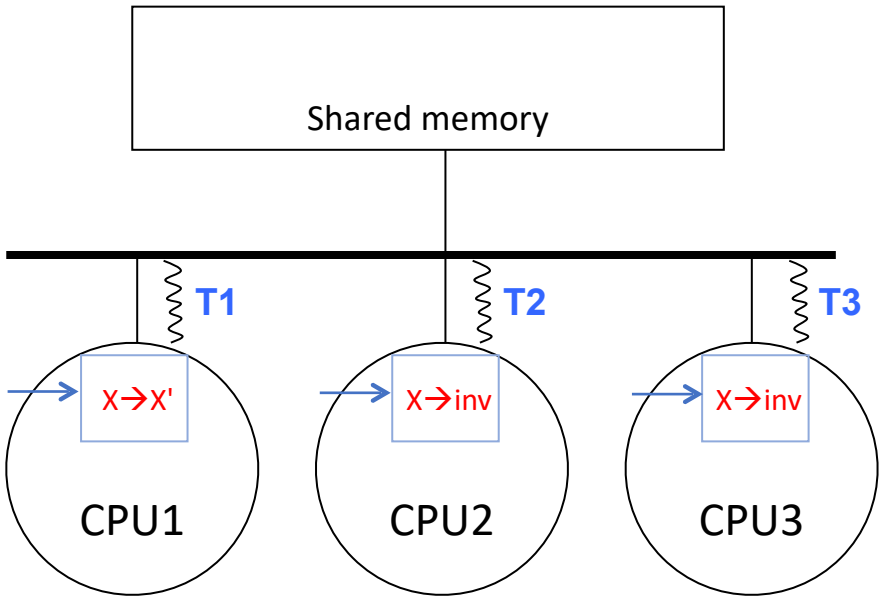
- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'



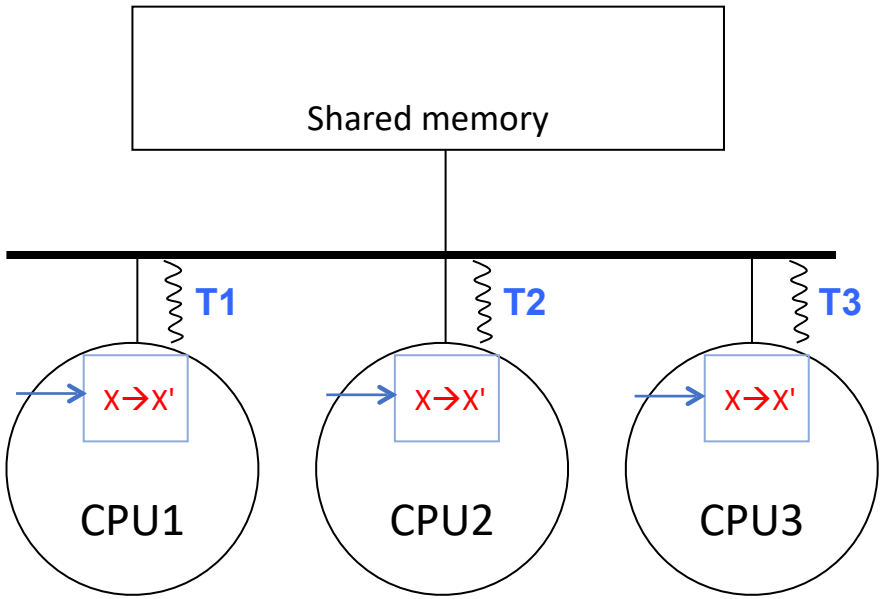
3) Threads have identical views of memory

- Two possible solutions, in hardware
- In both cases, cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from X to X'

Write-invalidate protocol



Write-update protocol



Session ID:
dag2200

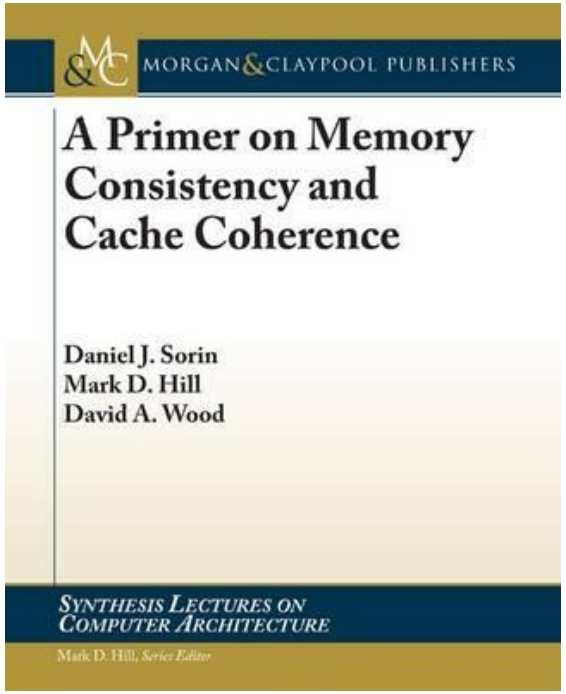
Keeping the caches coherent in an SMP

A. …is the responsibility of the user program
B. …is the responsibility of the hardware
C. …is the responsibility of the operating system
D. …is impossible
E. …is why we don't allow caches in SMP systems

Session ID: dag2200

Keeping the caches coherent in an SMP

A. …is the responsibility of the user program
B. …is the responsibility of the hardware
C. …is the responsibility of the operating system
D. …is impossible
E. …is why we don't allow caches in SMP systems



Summary

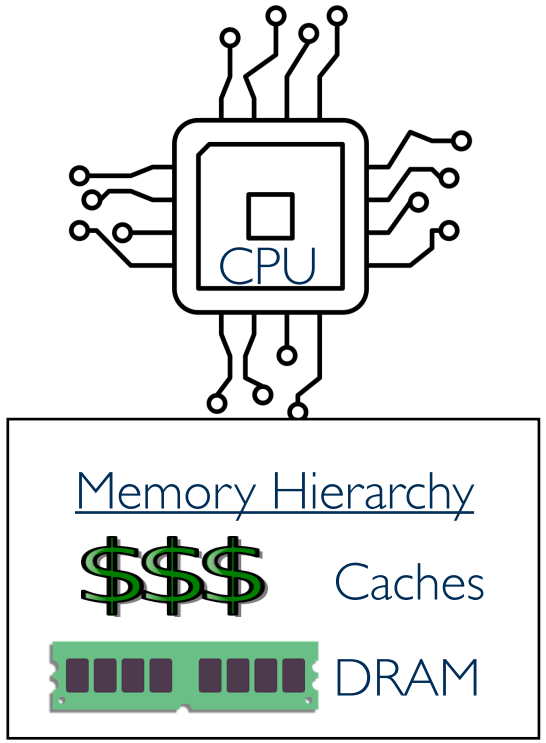
- Page tables in shared memory
  - Set up by the OS
  - Used by the hardware
- TLB consistency in software by the OS
  - Hardware brings PTE into the TLB from the PT
  - Page replacement algorithm changes the PT and does the TLB shoot-down
- Synchronized atomicity
  - Test-and-set instruction serialized by the shared bus
  - Atomic read-modify-write transaction
- Cache coherence in hardware
  - Invalidation based or update based

Roadmap

- Programmed IO and DMA
  - Chapter 10 (10.1 – 10.7)

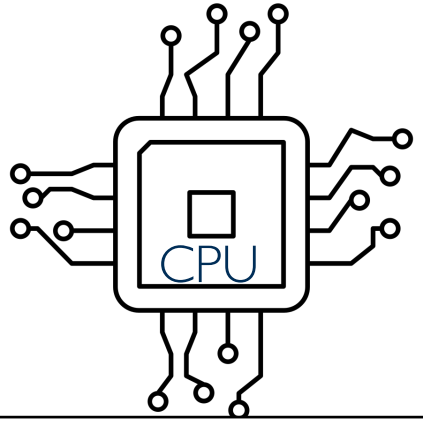
- Networking
  - Chapter 13

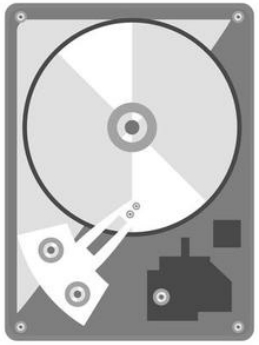
Beyond the processor and memory

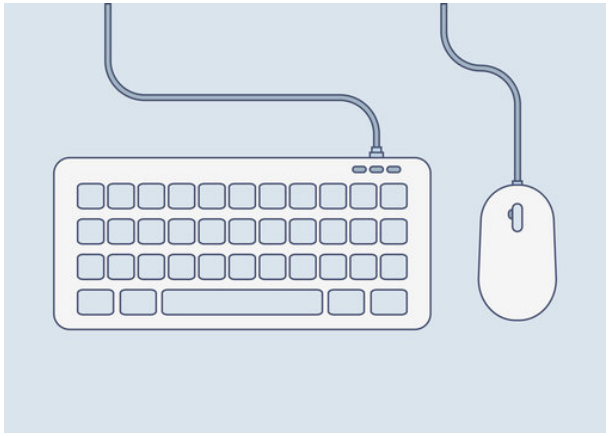


Anything else??

Beyond the processor and memory



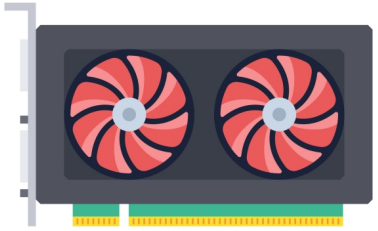


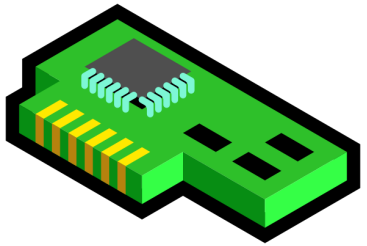


Memory Hierarchy

$$$ Caches

DRAM

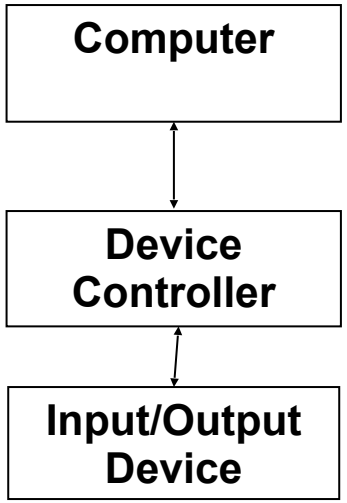




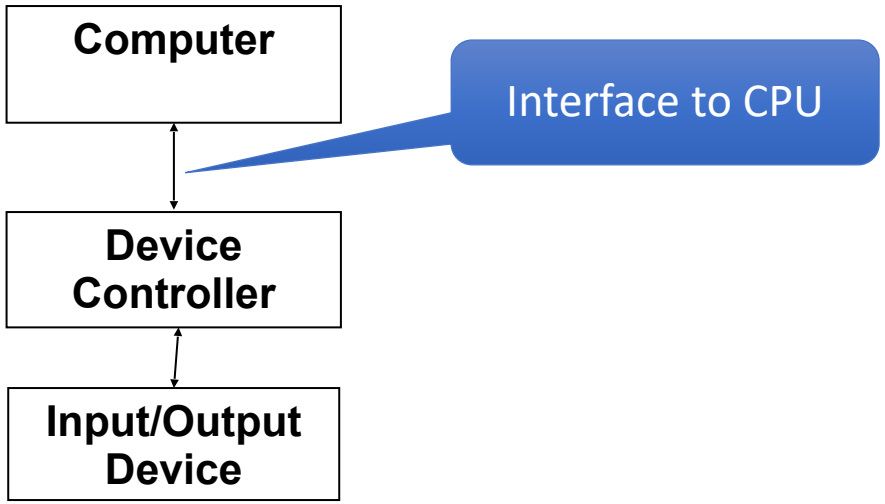
Anything else??

A lot of peripheral devices!

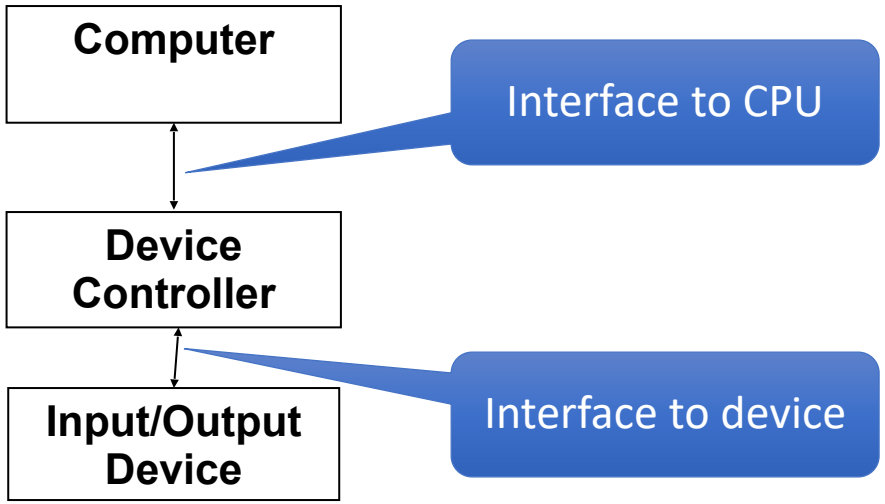
Communication: CPU and I/O Devices



Communication: CPU and I/O Devices

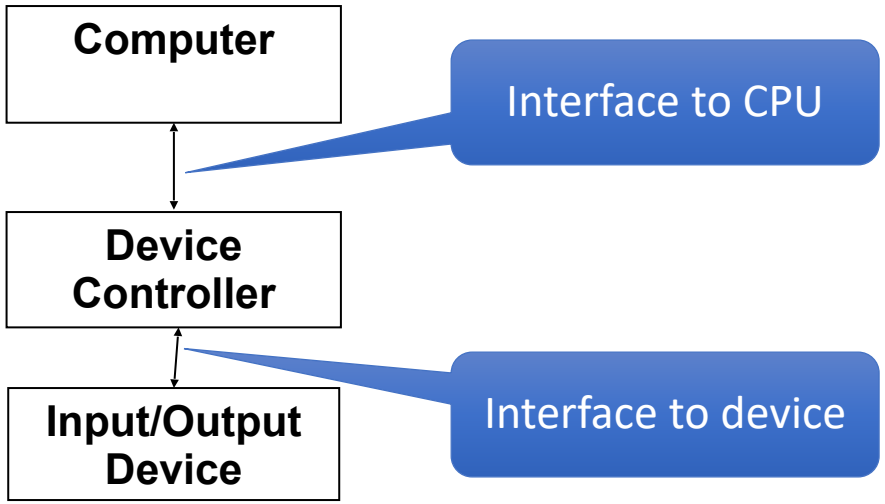


Communication: CPU and I/O Devices



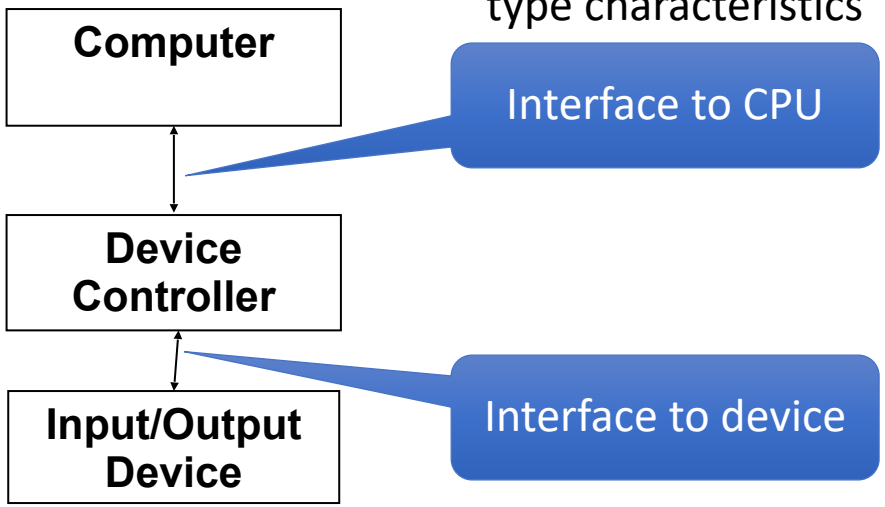
Communication: CPU and I/O Devices

- Generic part

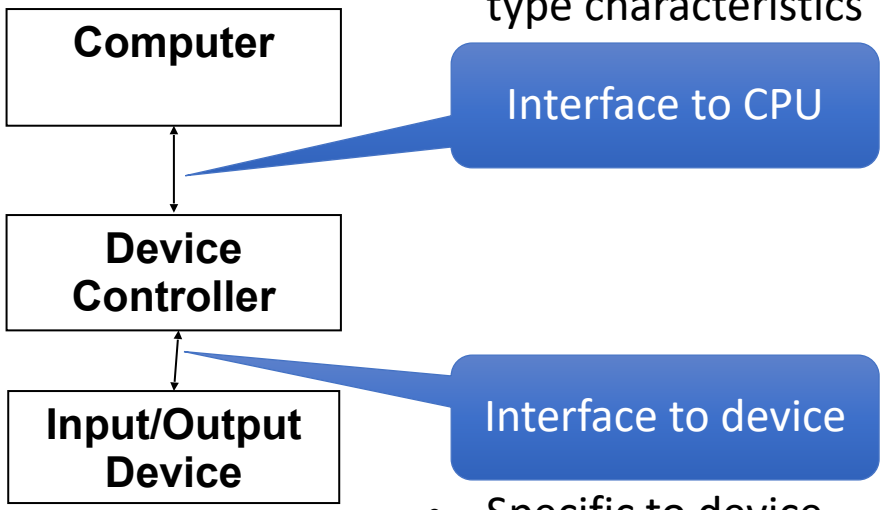


Communication: CPU and I/O Devices

- Generic part
- Specific to device-type characteristics

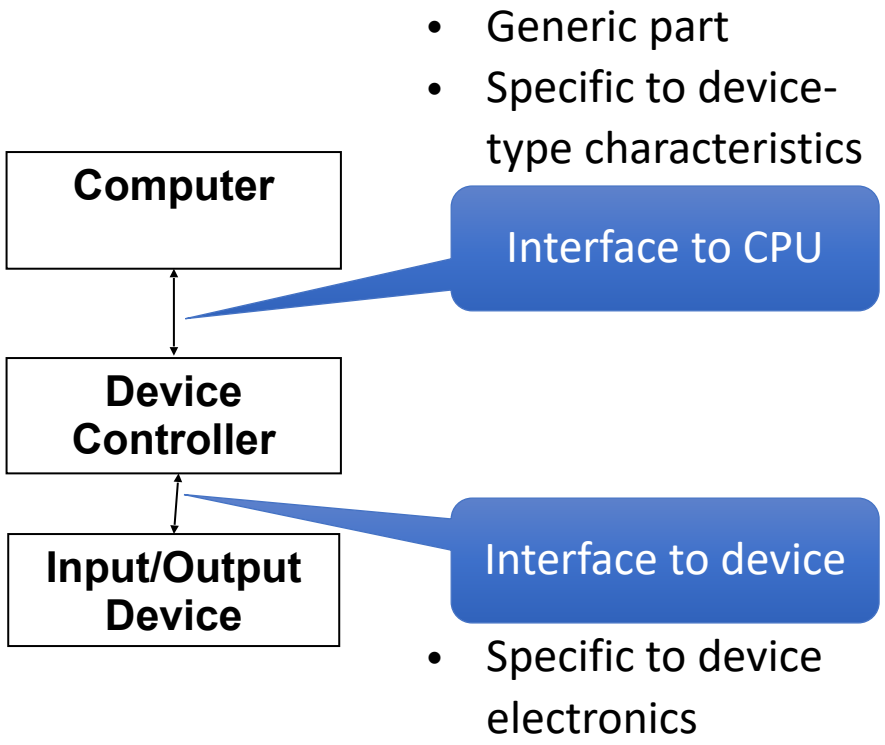


Communication: CPU and I/O Devices



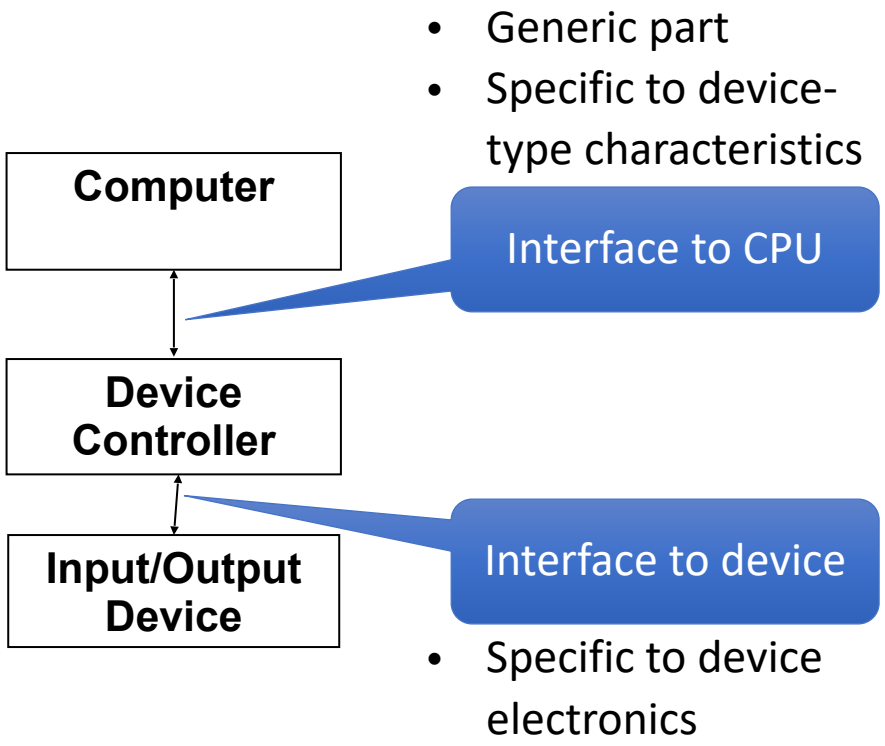
- Generic part
- Specific to device-type characteristics
- Specific to device electronics

Communication: CPU and I/O Devices



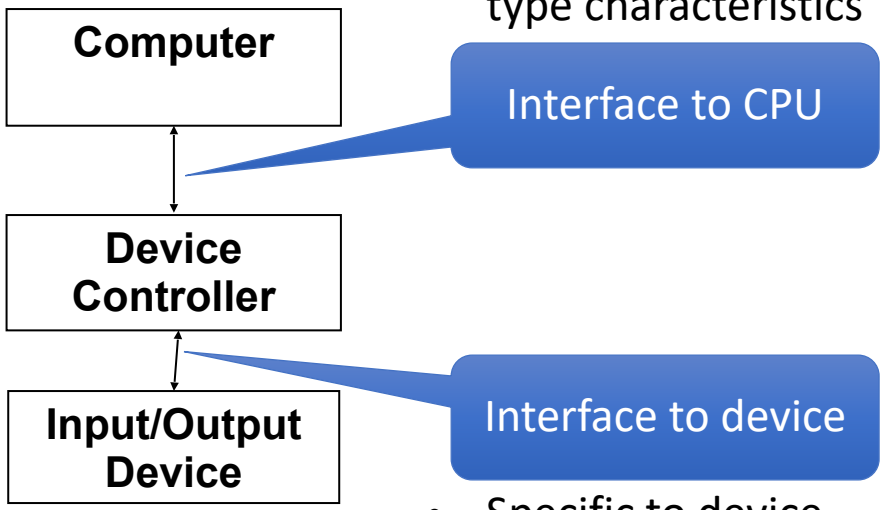
- How does the CPU "talk" to the controller
  - Memory-mapped I/O
  - Load/store instructions
  - (Alternative is special I/O instructions)

Communication: CPU and I/O Devices



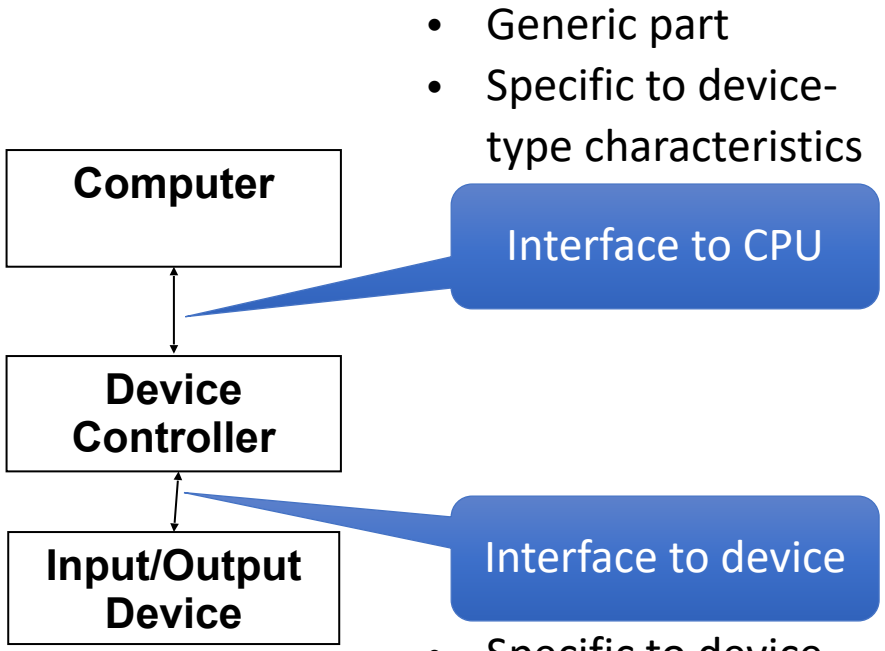
- How does the CPU "talk" to the controller
  - Memory-mapped I/O
  - Load/store instructions
  - (Alternative is special I/O instructions)
- How is data movement effected?
  - Slow speed
    - Programmed I/O
  - High speed
    - DMA & interrupts

Communication: CPU and I/O Devices



- Generic part
- Specific to device-type characteristics
- Specific to device electronics

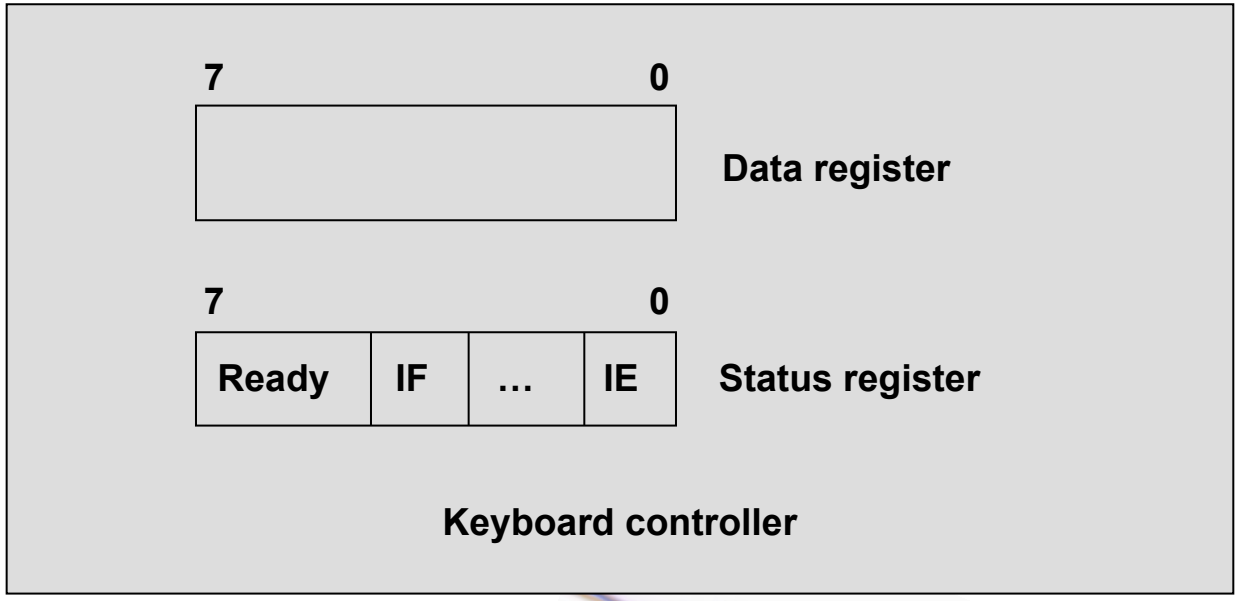
Communication: CPU and I/O Devices



- Generic part
- Specific to device-type characteristics
- Specific to device electronics

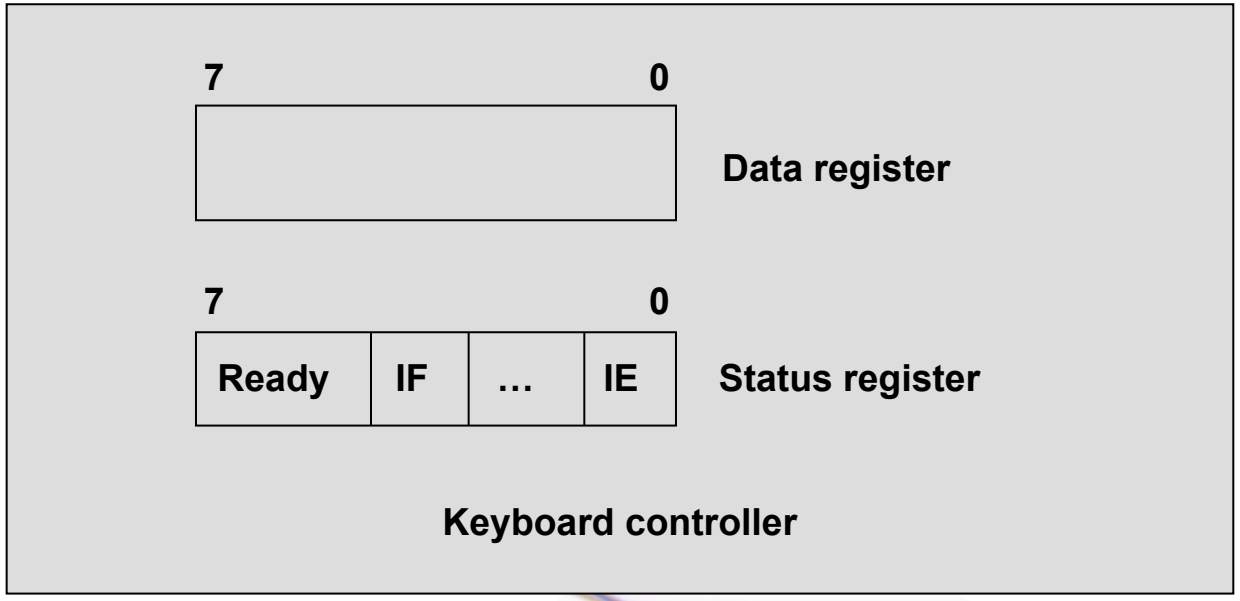
- What commands do we issue to devices?
  - Camera
  - Display
  - Audio
  - …

A keyboard device



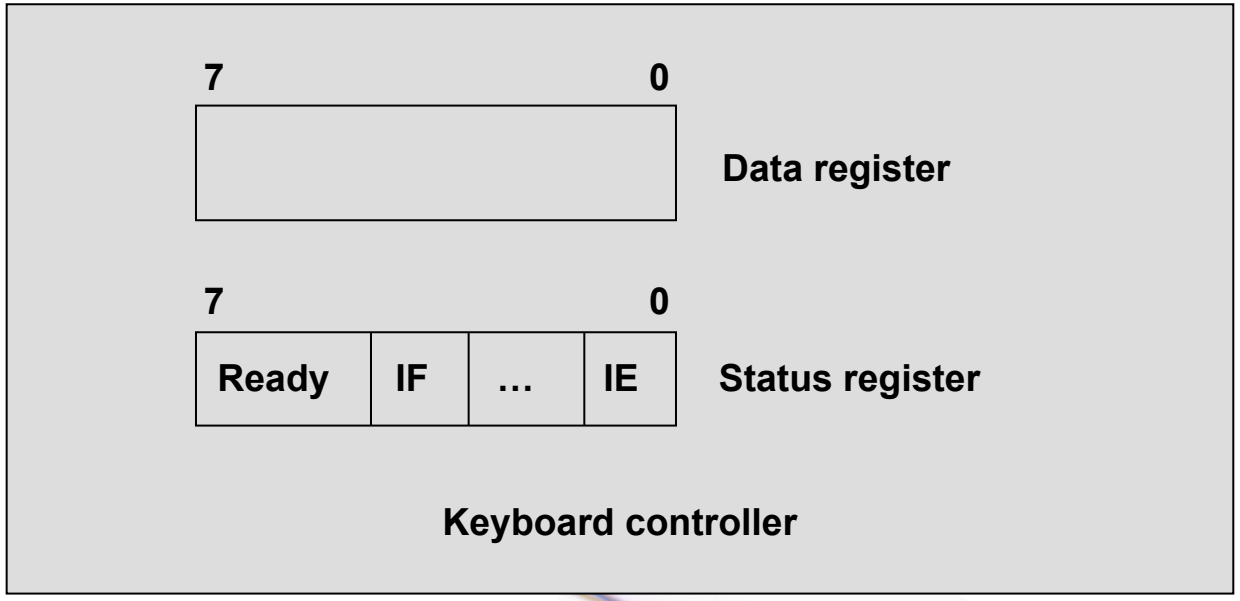
A keyboard device

Controller



A keyboard device

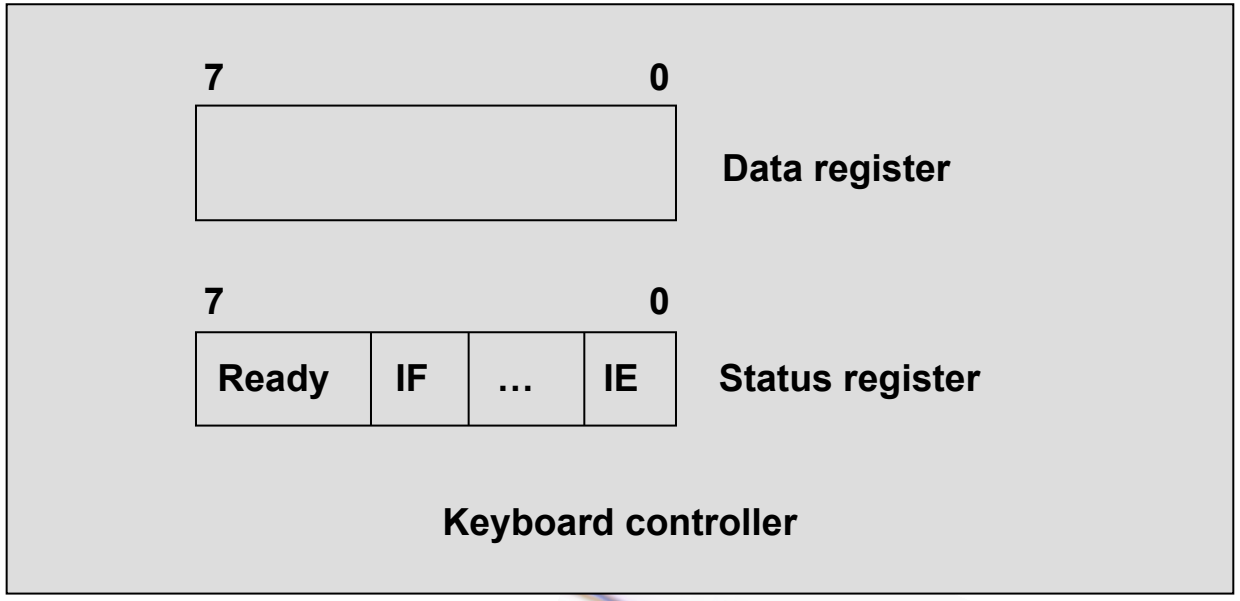
Controller



Device

A keyboard device

Controller

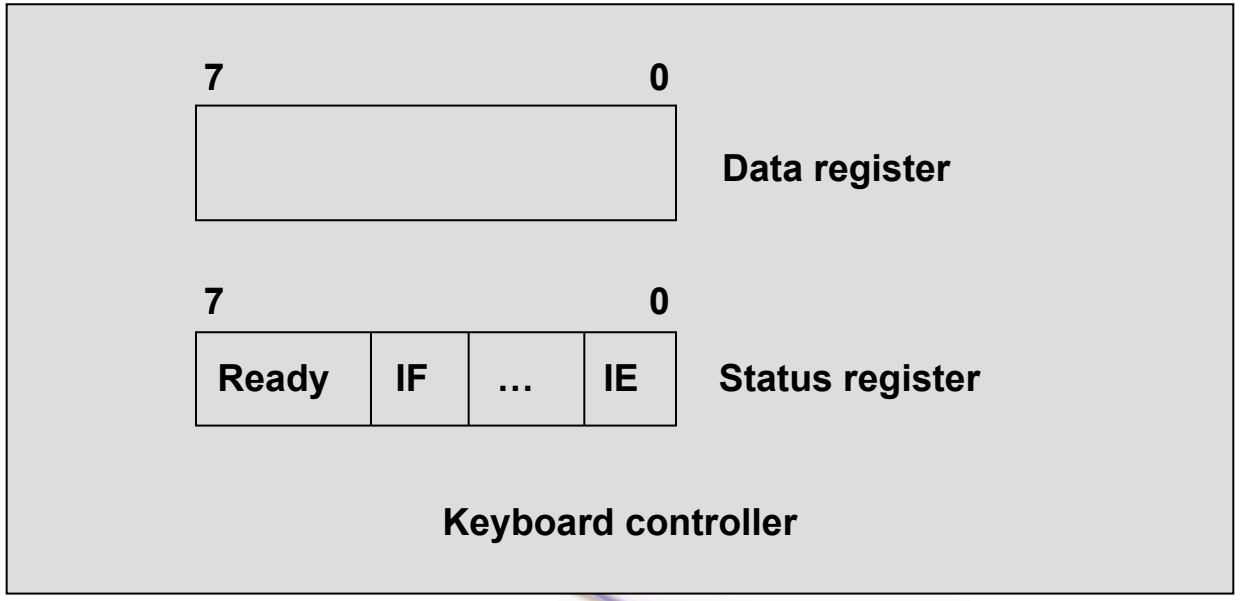


- Slow speed device

Device

A keyboard device

Controller

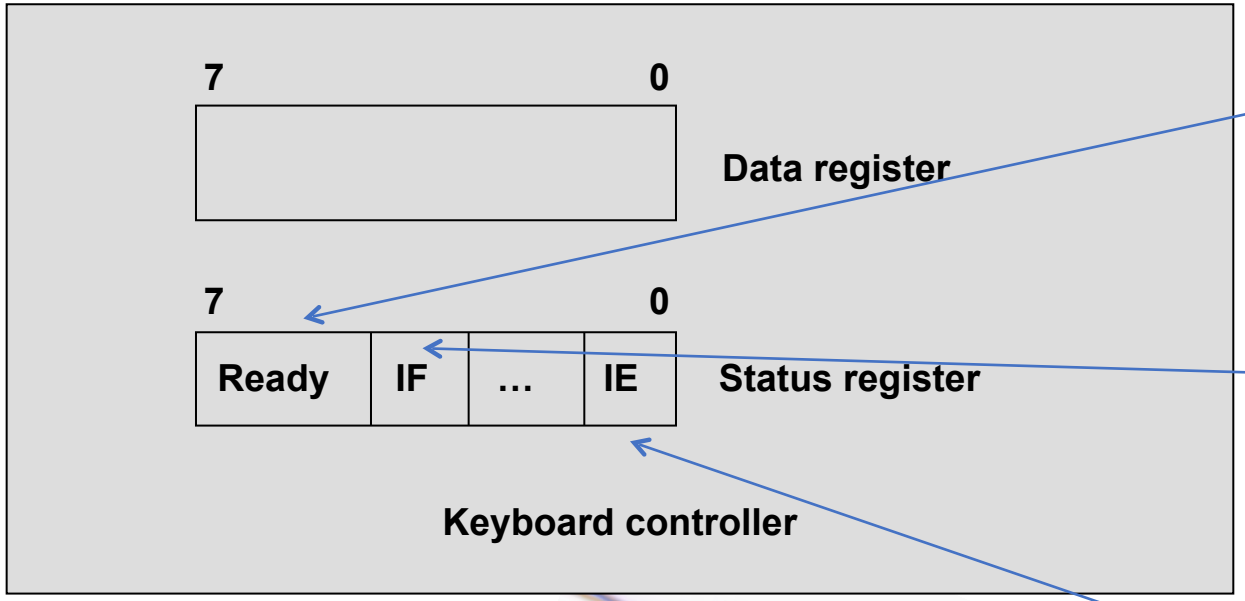


- Slow speed device
- Service it with programmed I/O (PIO)

Device

A keyboard device

Controller



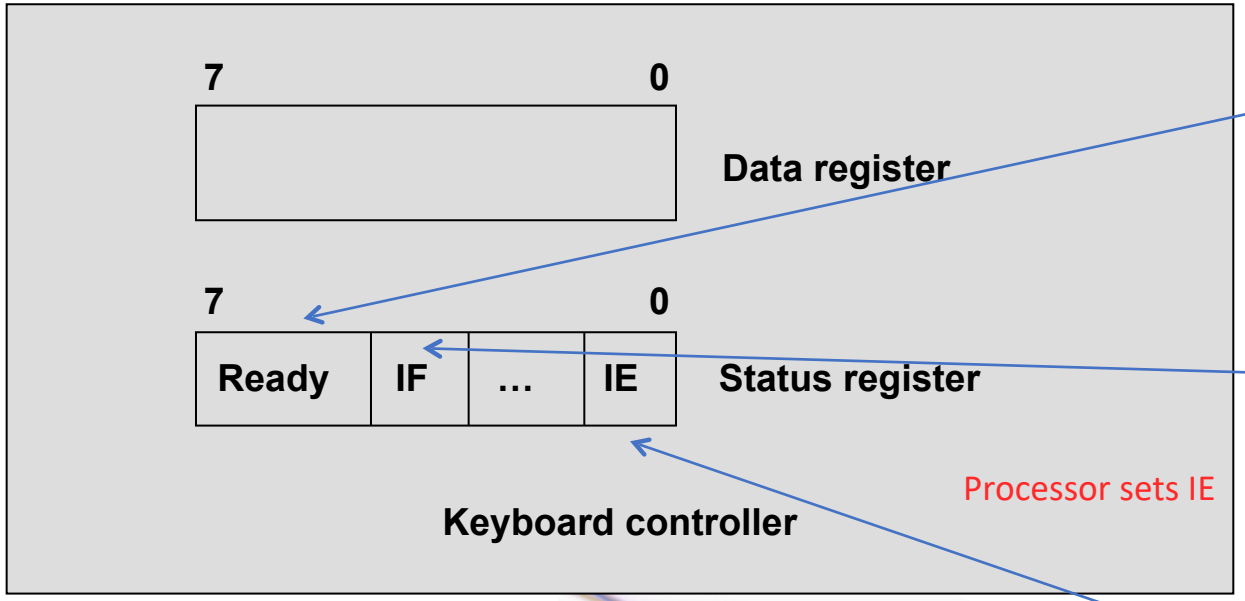
- Ready: there's a byte waiting in the data register
- IF: the device has produced data – controller ready to signal an interrupt
- IE: If 1 interrupt when Ready

- Slow speed device
- Service it with programmed I/O (PIO)

Device

A keyboard device

Controller

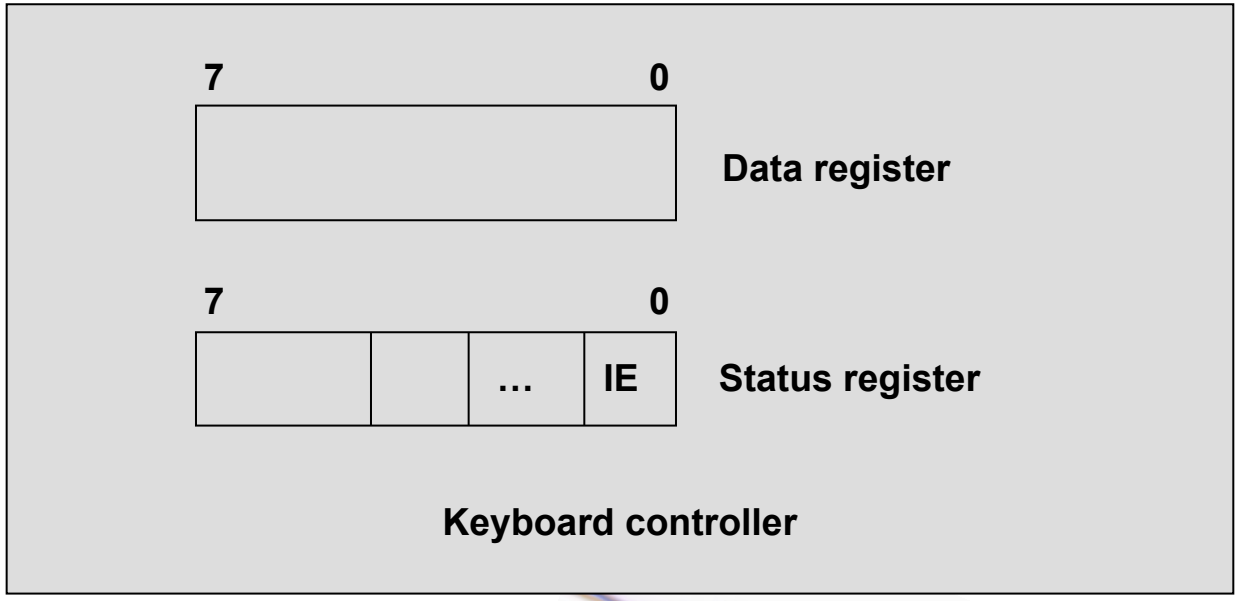


- Ready: there's a byte waiting in the data register
- IF: the device has produced data – controller ready to signal an interrupt
- IE: If 1 interrupt when Ready

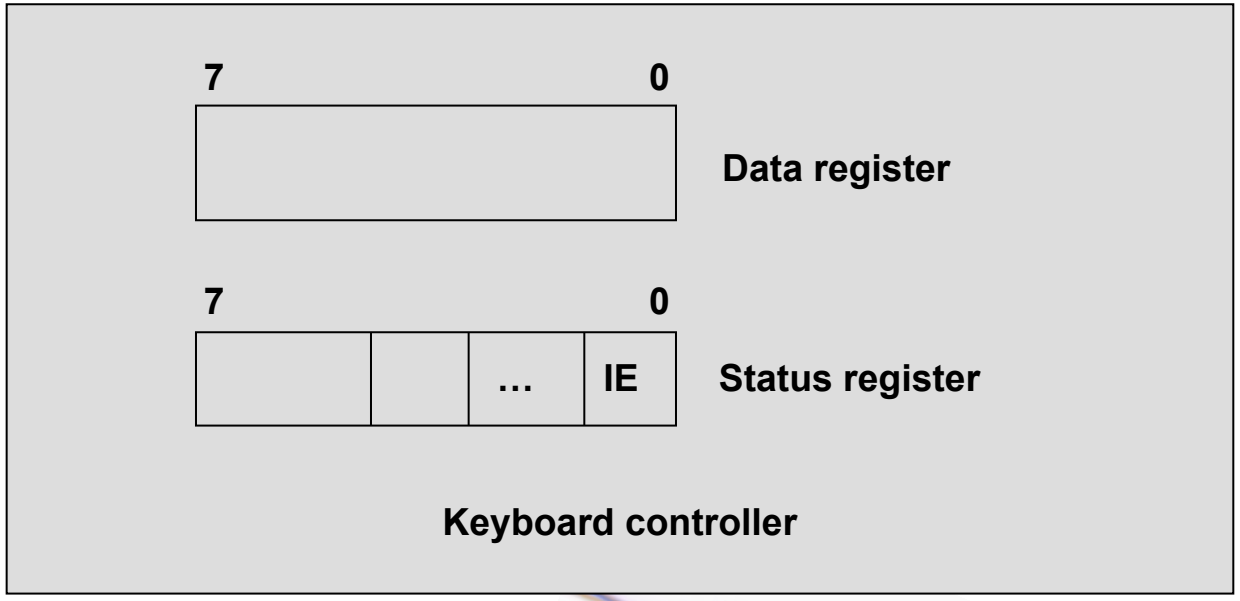
- Slow speed device
- Service it with programmed I/O (PIO)

Device

A keyboard device

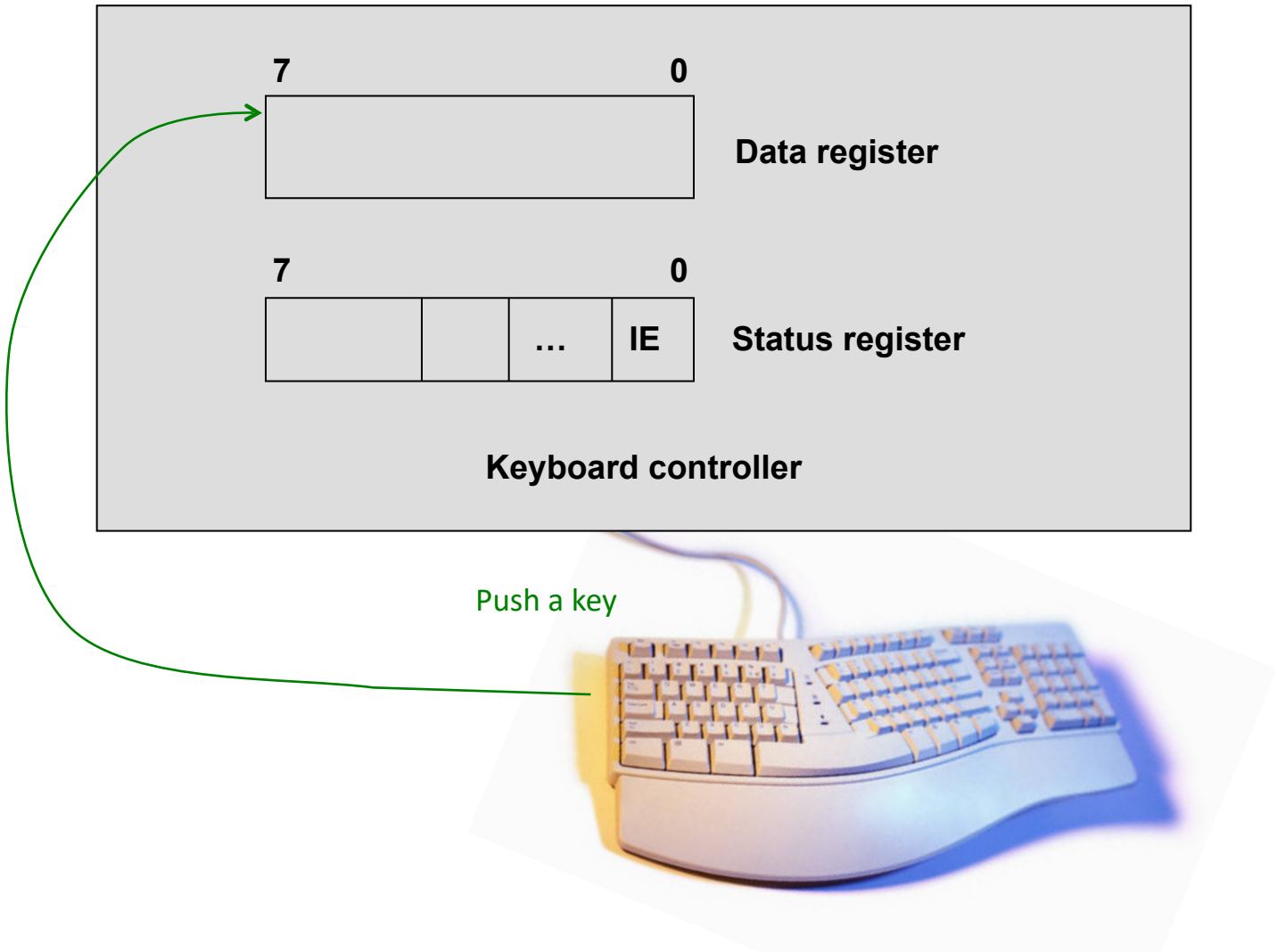


A keyboard device

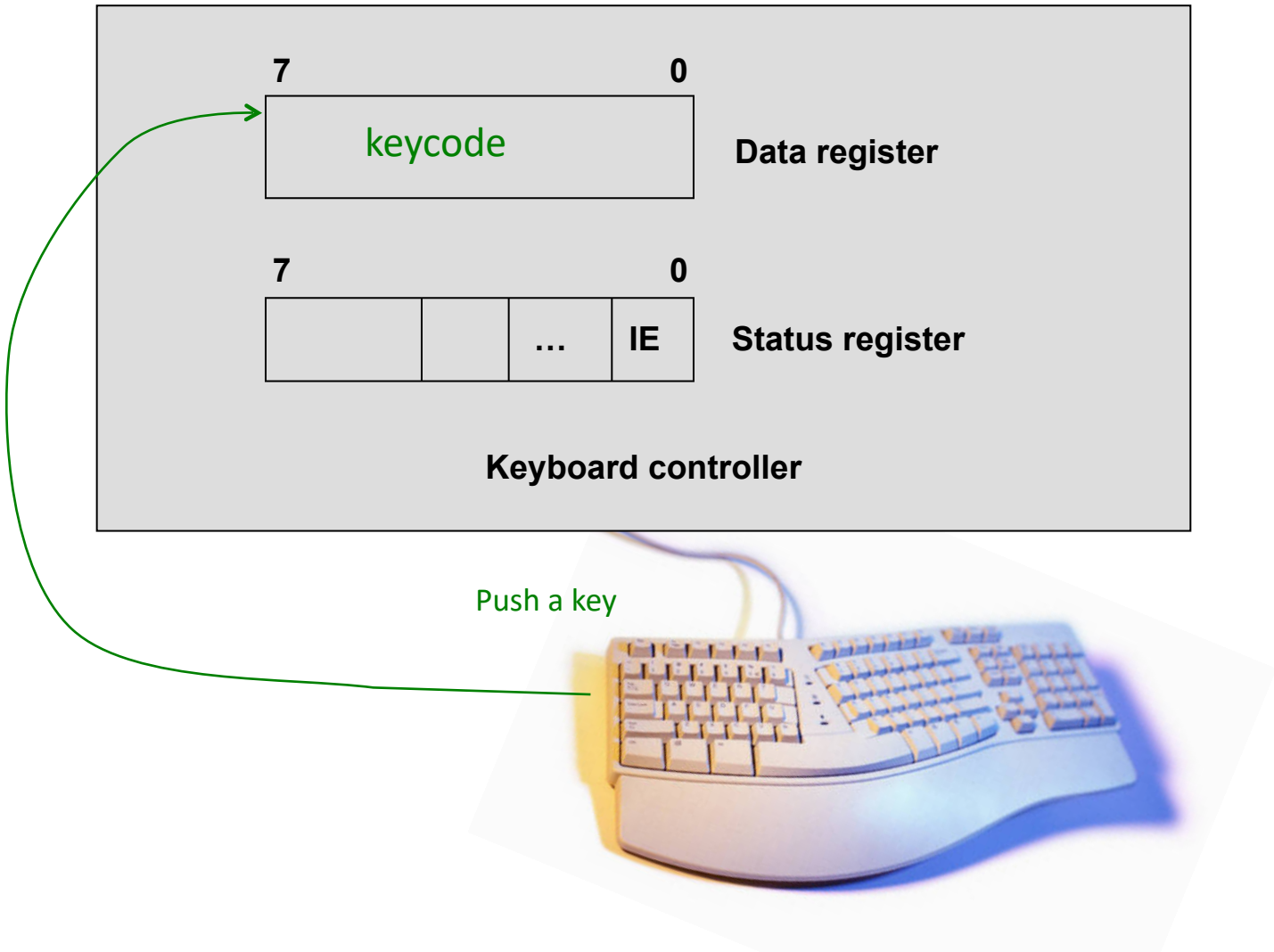


Push a key

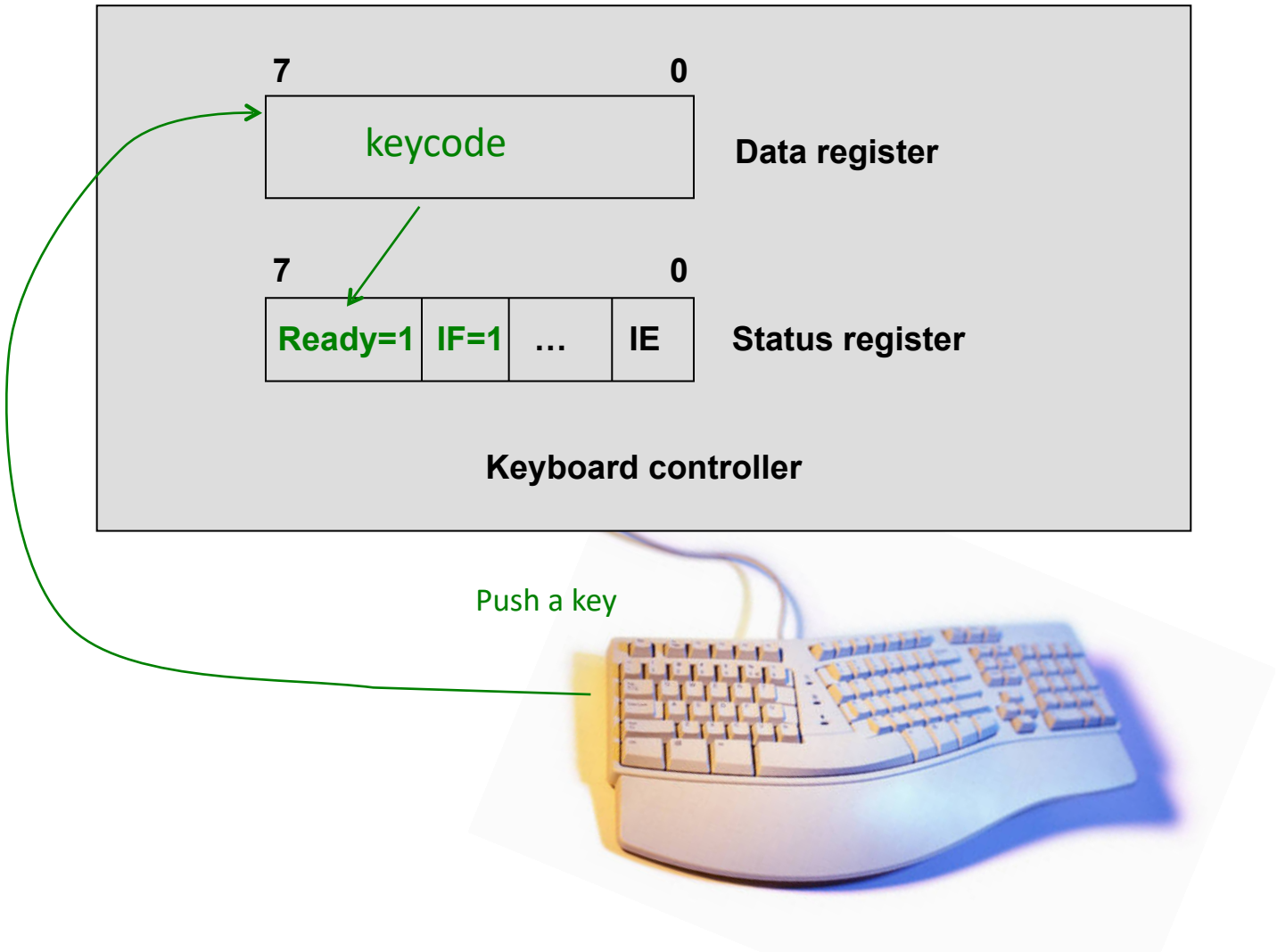
A keyboard device



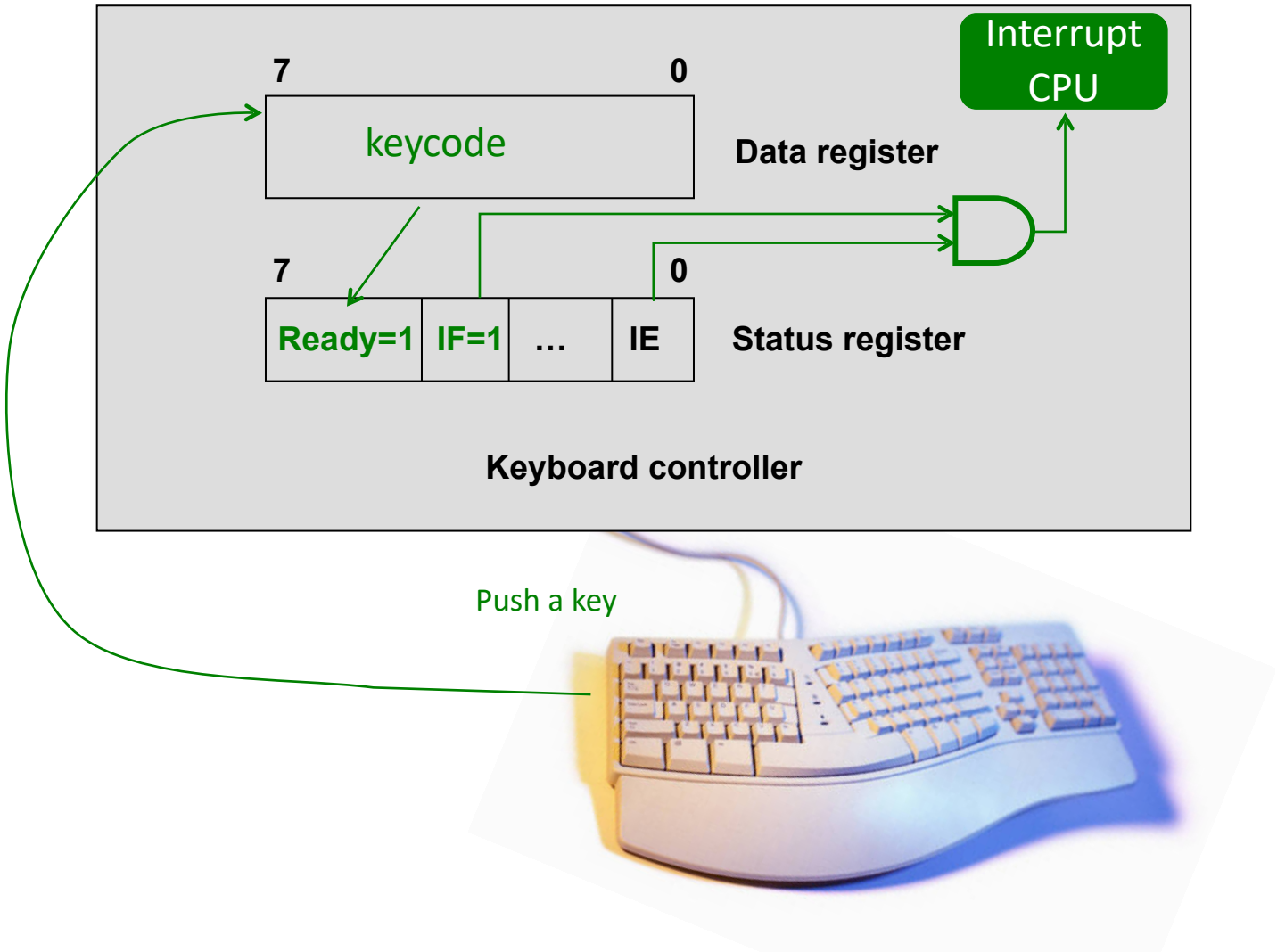
A keyboard device



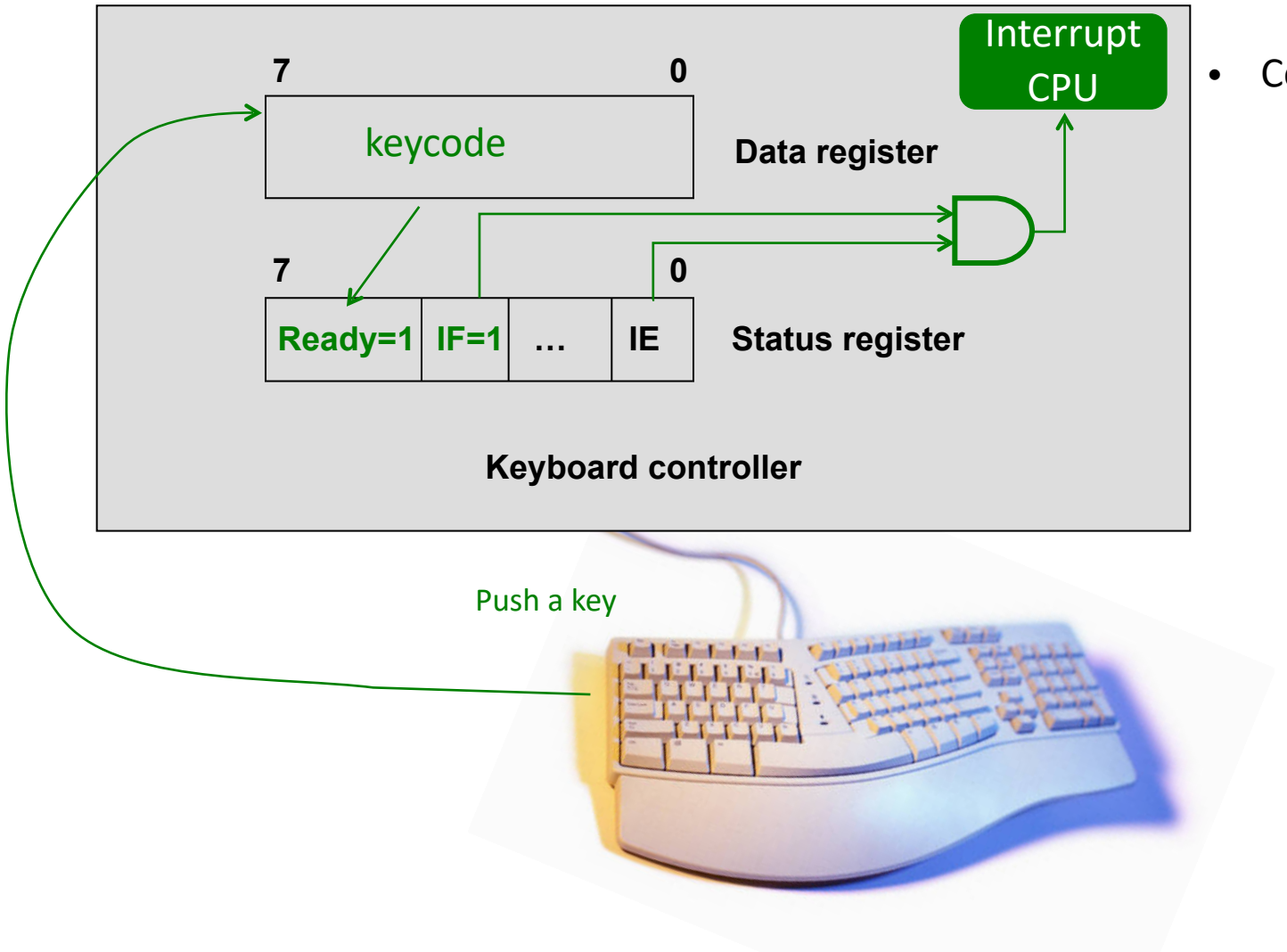
A keyboard device



A keyboard device

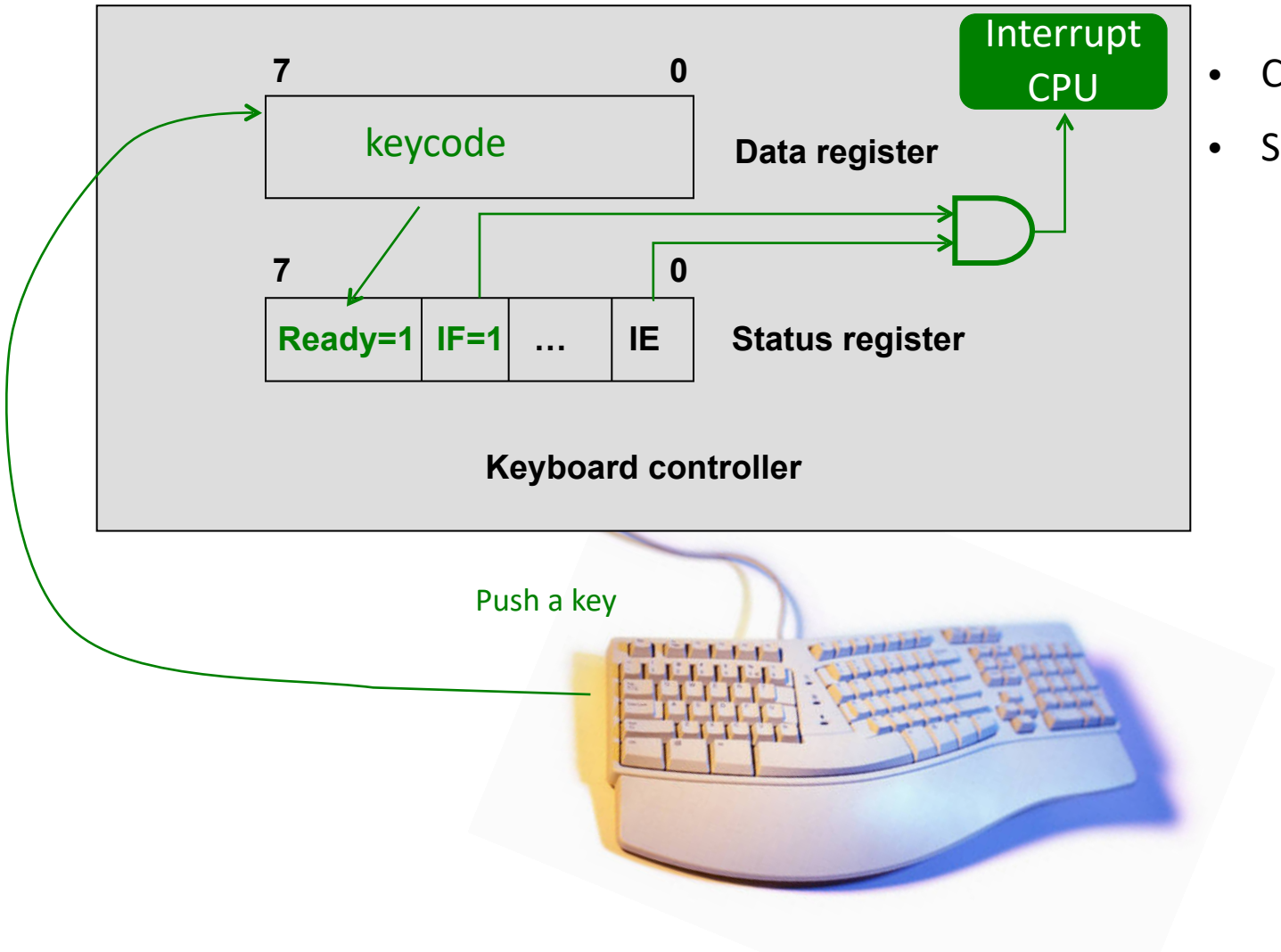


A keyboard device



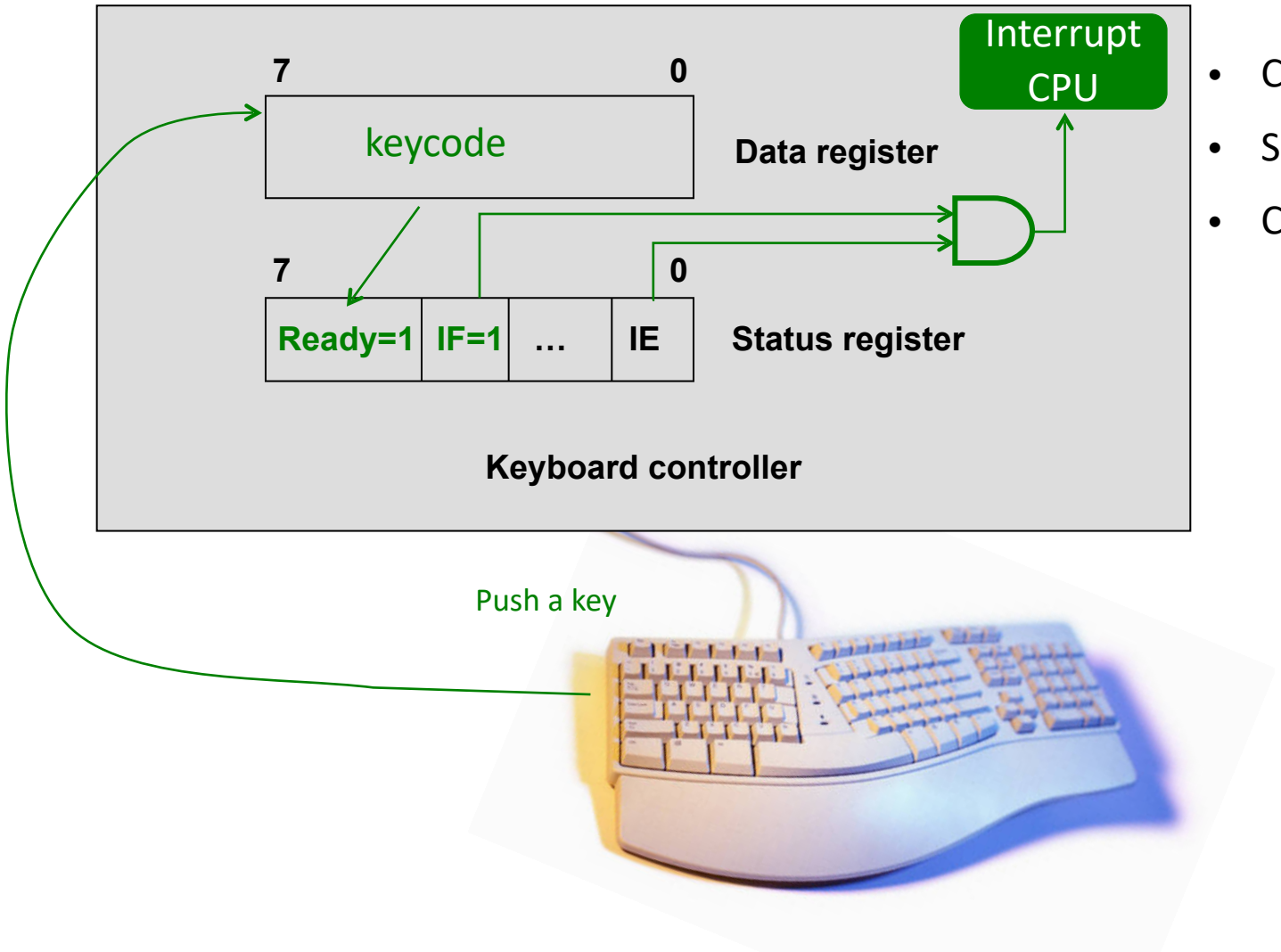
- Commands from the CPU:

A keyboard device



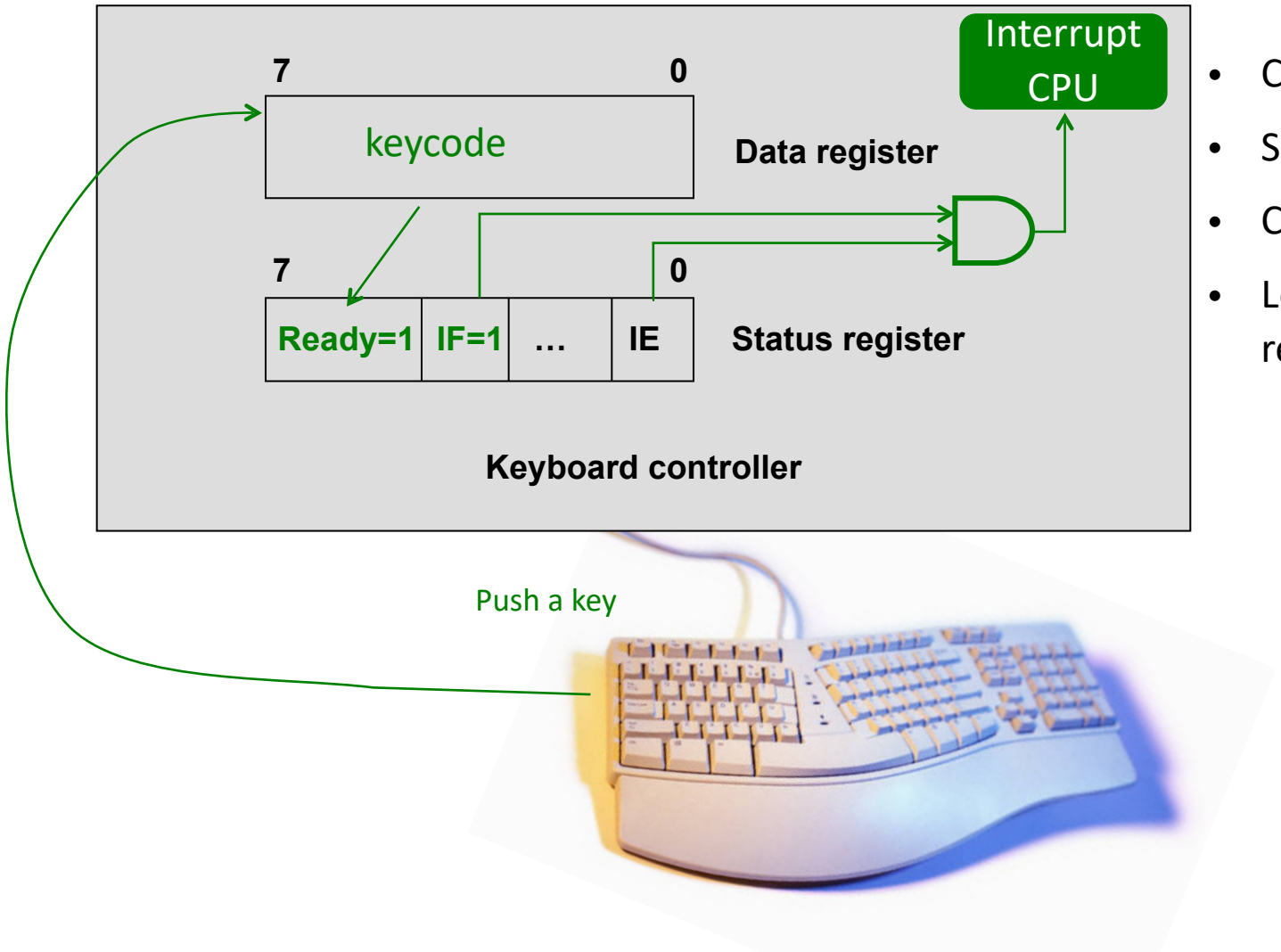
- Commands from the CPU:
- Set IE

A keyboard device



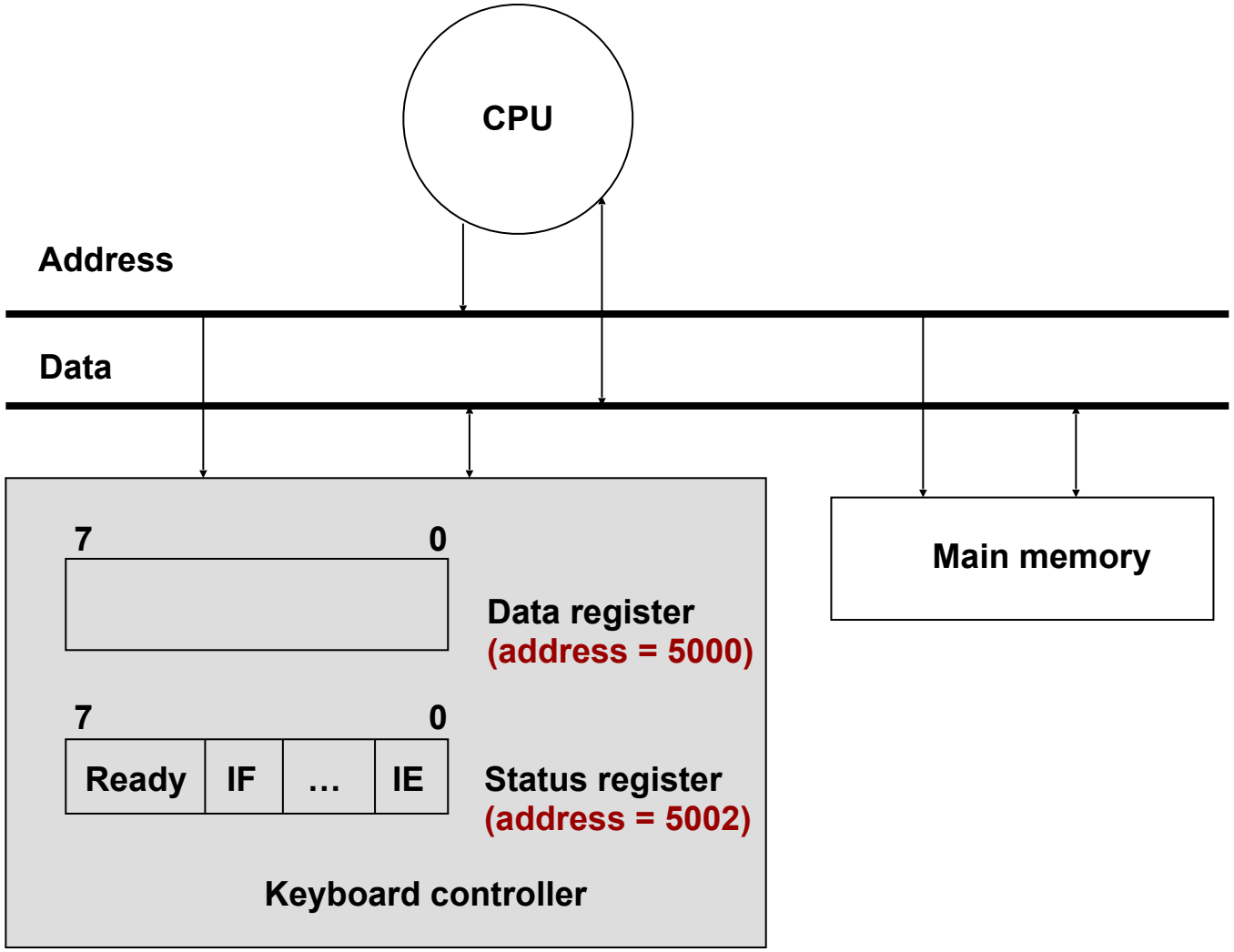
- Commands from the CPU:
- Set IE
- Check Ready equals 1

A keyboard device

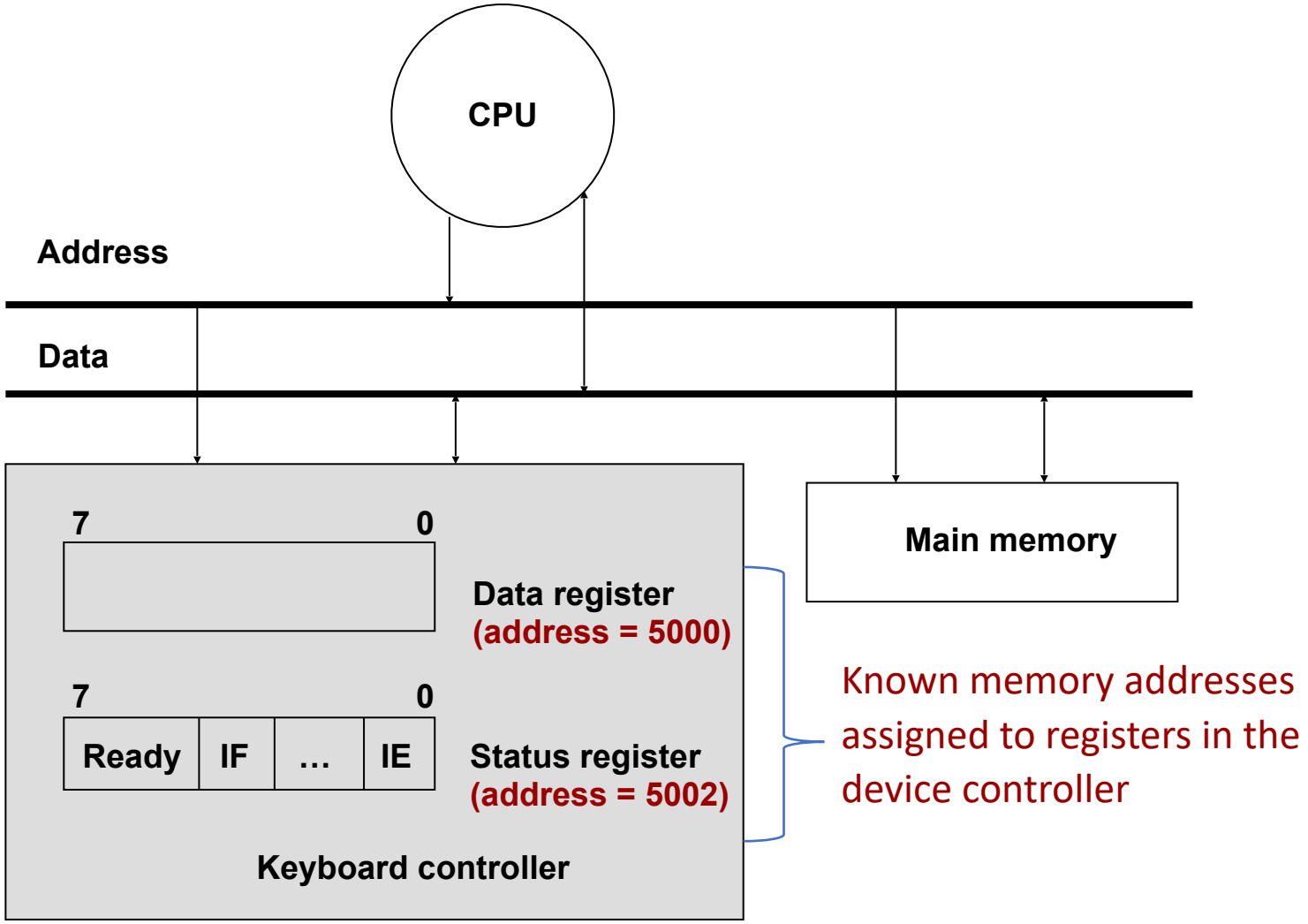


- Commands from the CPU:
- Set IE
- Check Ready equals 1
- Load keycode from data register

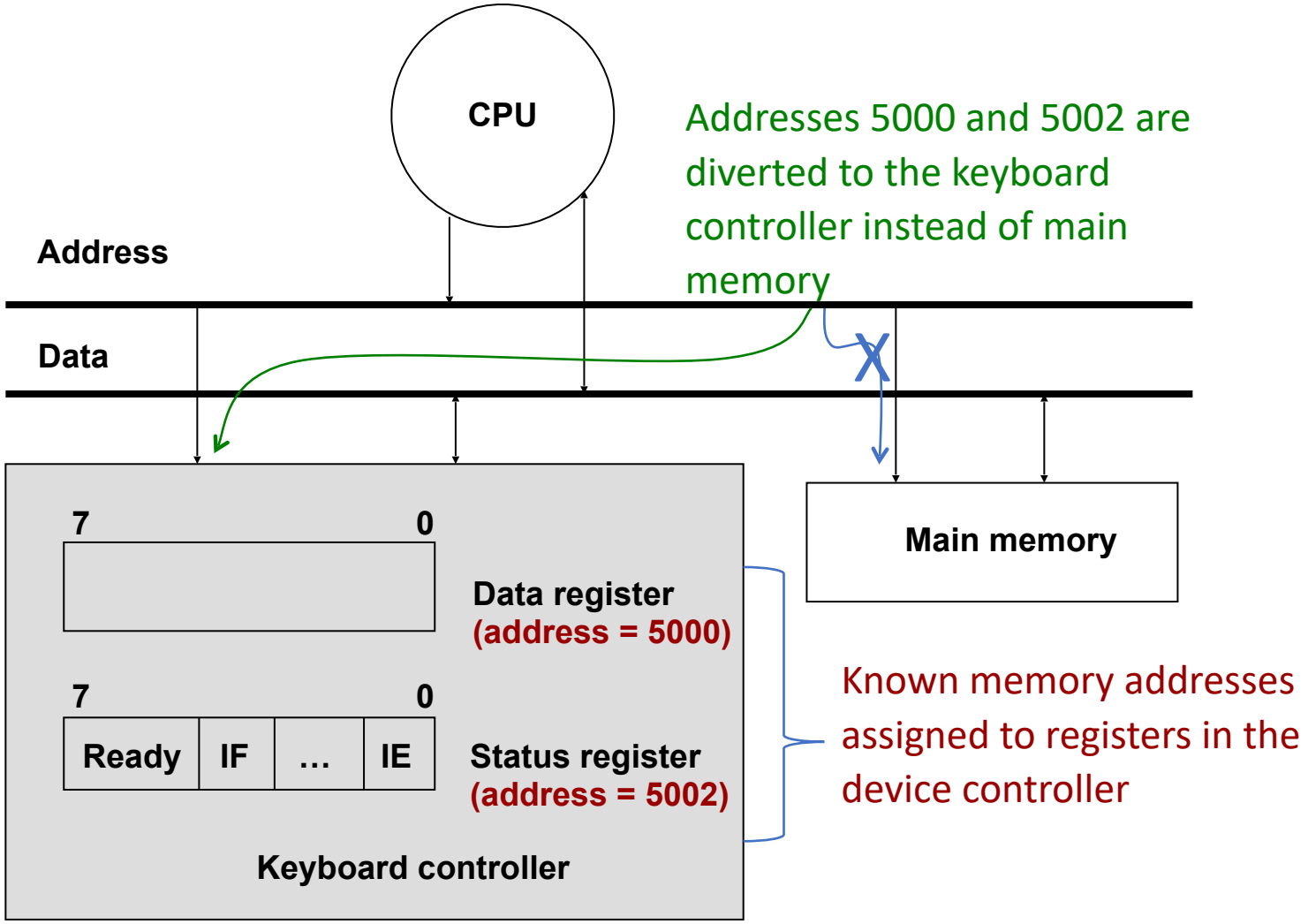
Memory mapped I/O



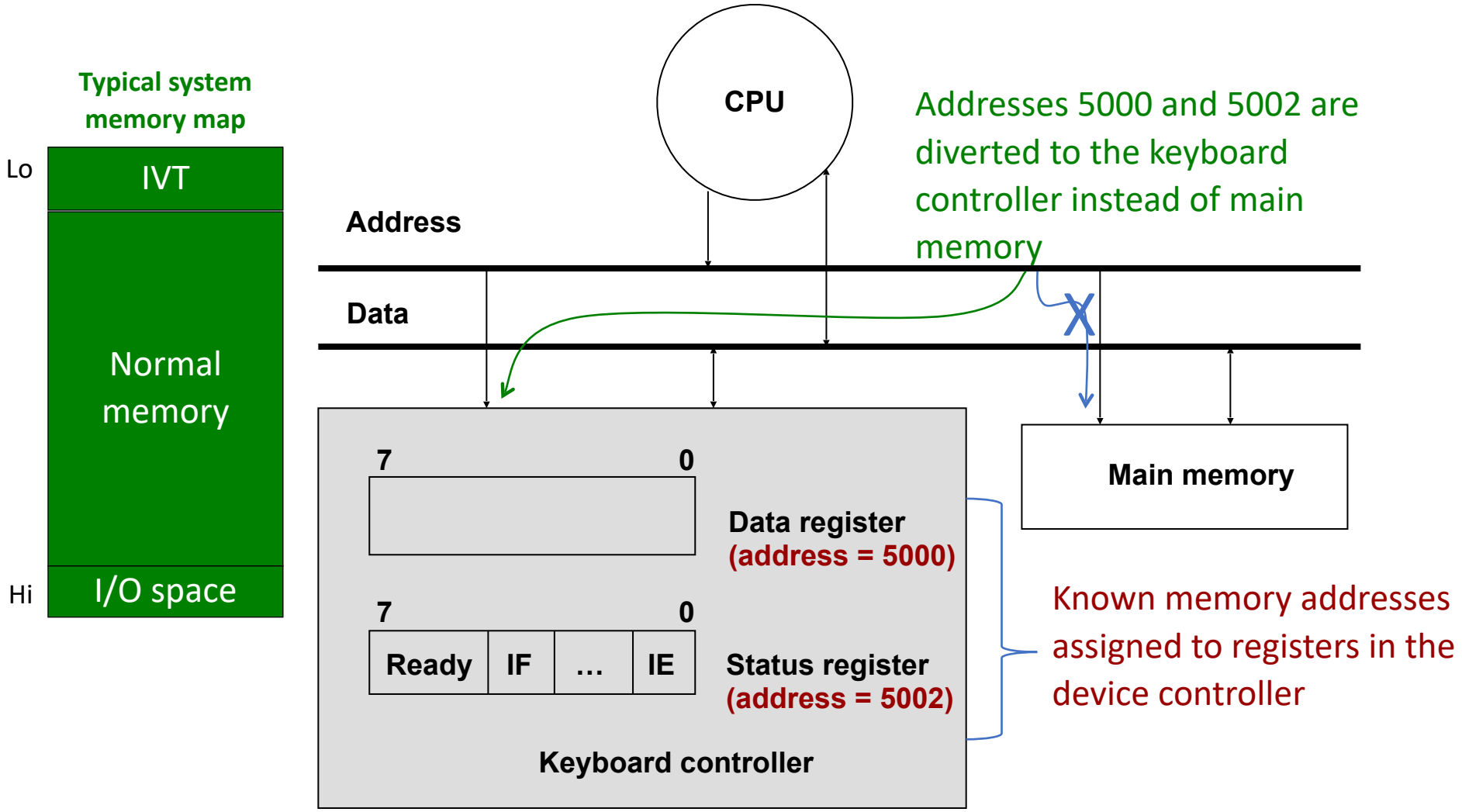
Memory mapped I/O



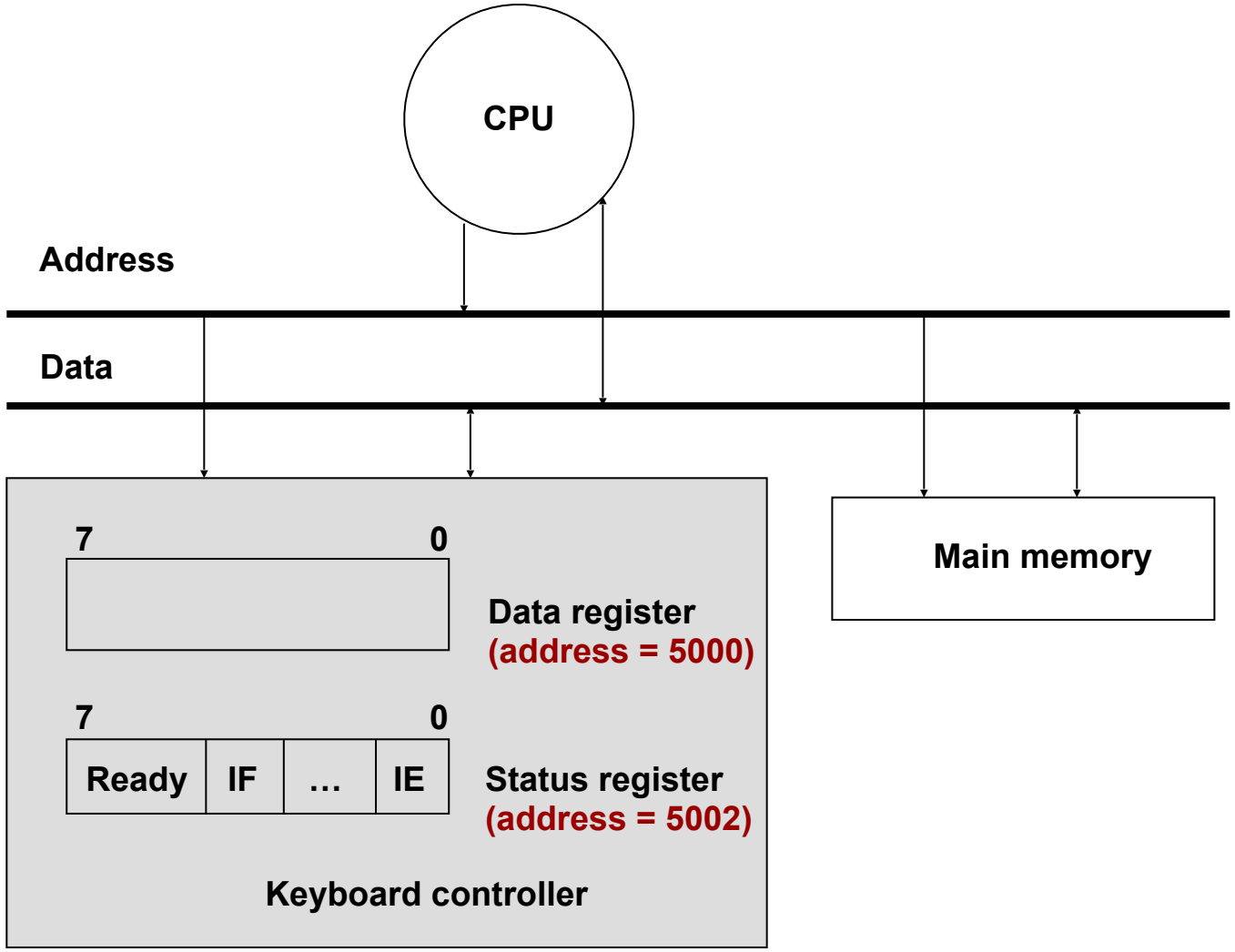
Memory mapped I/O



Memory mapped I/O

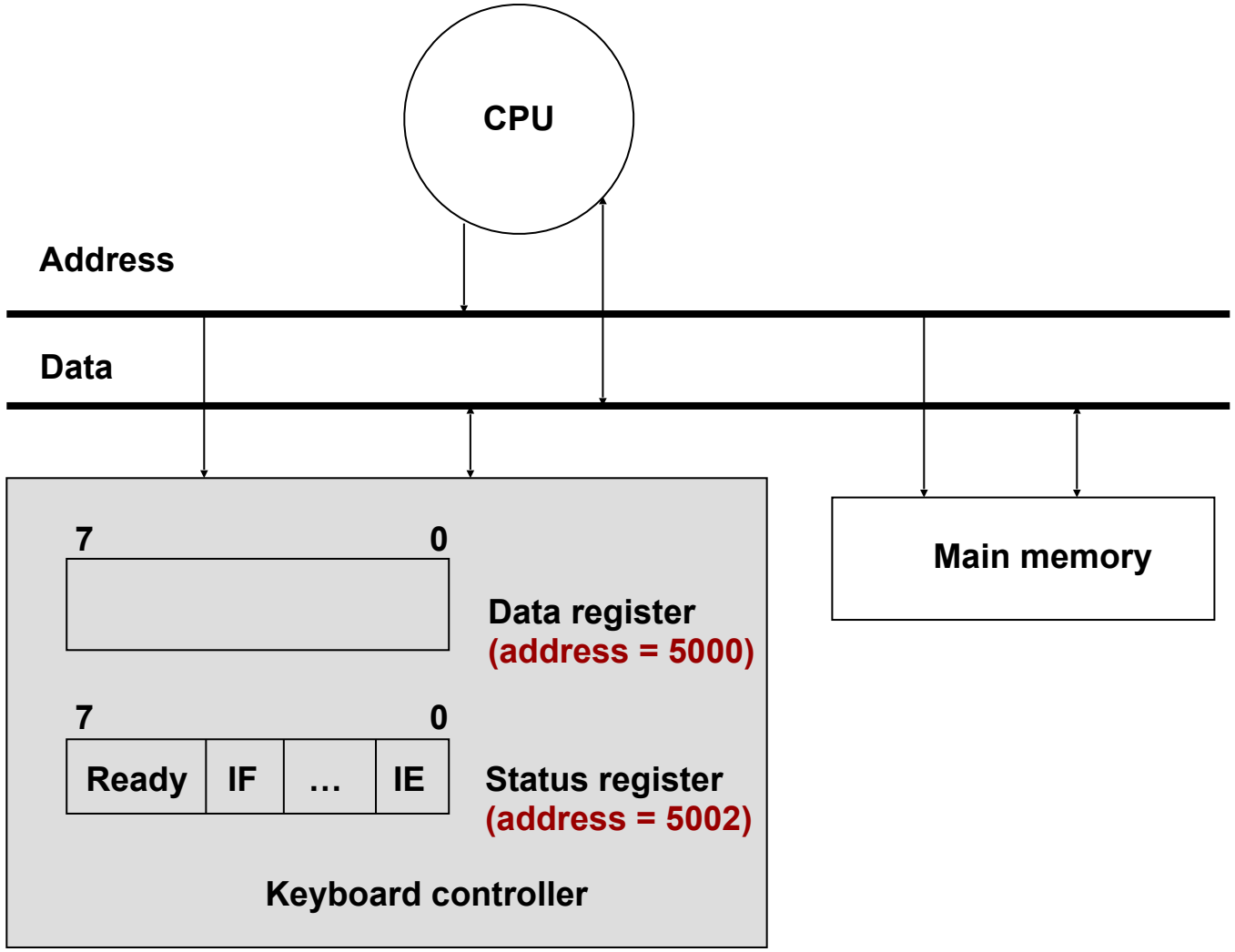


Memory mapped I/O



Memory mapped I/O

Commands?

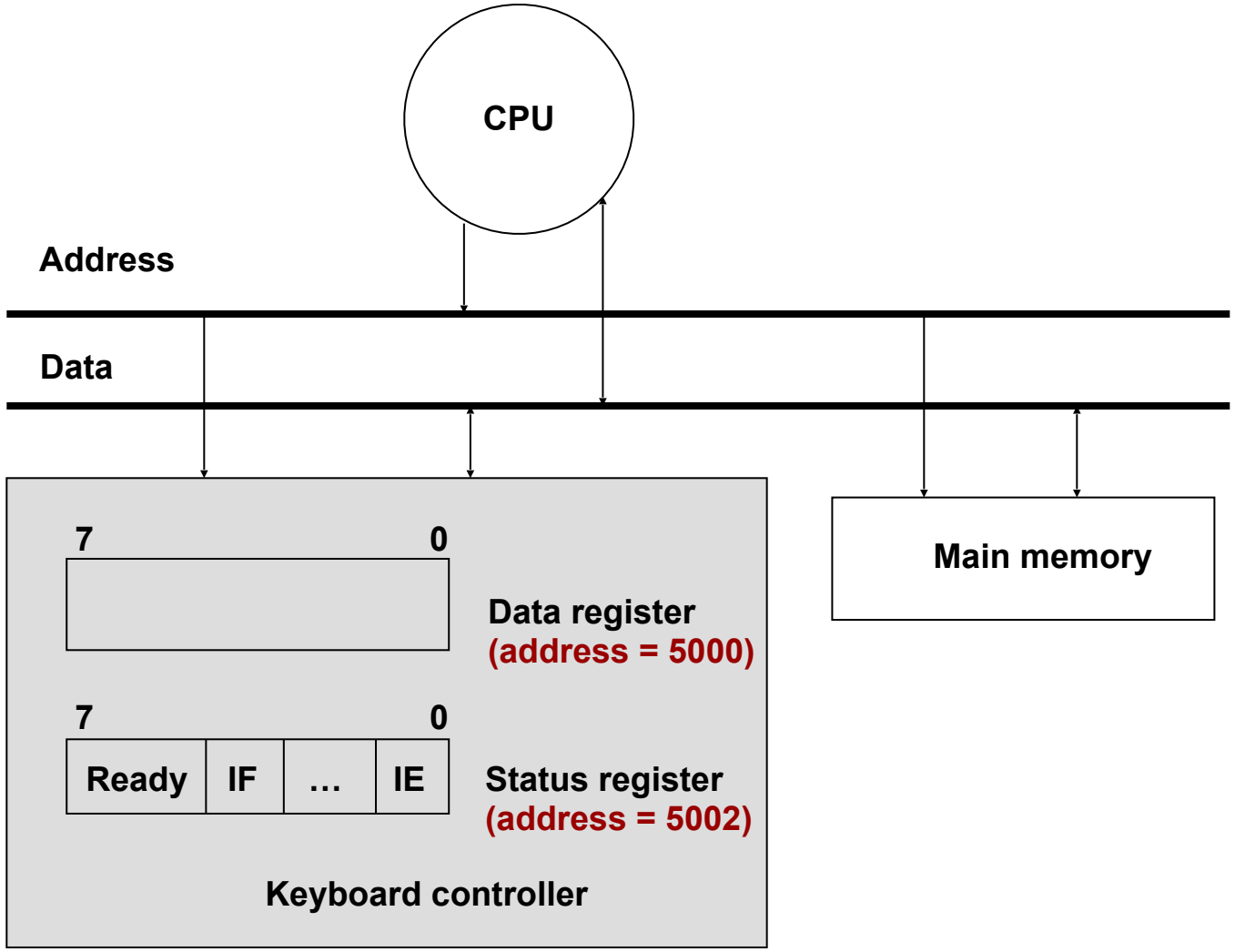


Memory mapped I/O

Commands?

Check for new data

LD R1,mem[5002]



Memory mapped I/O

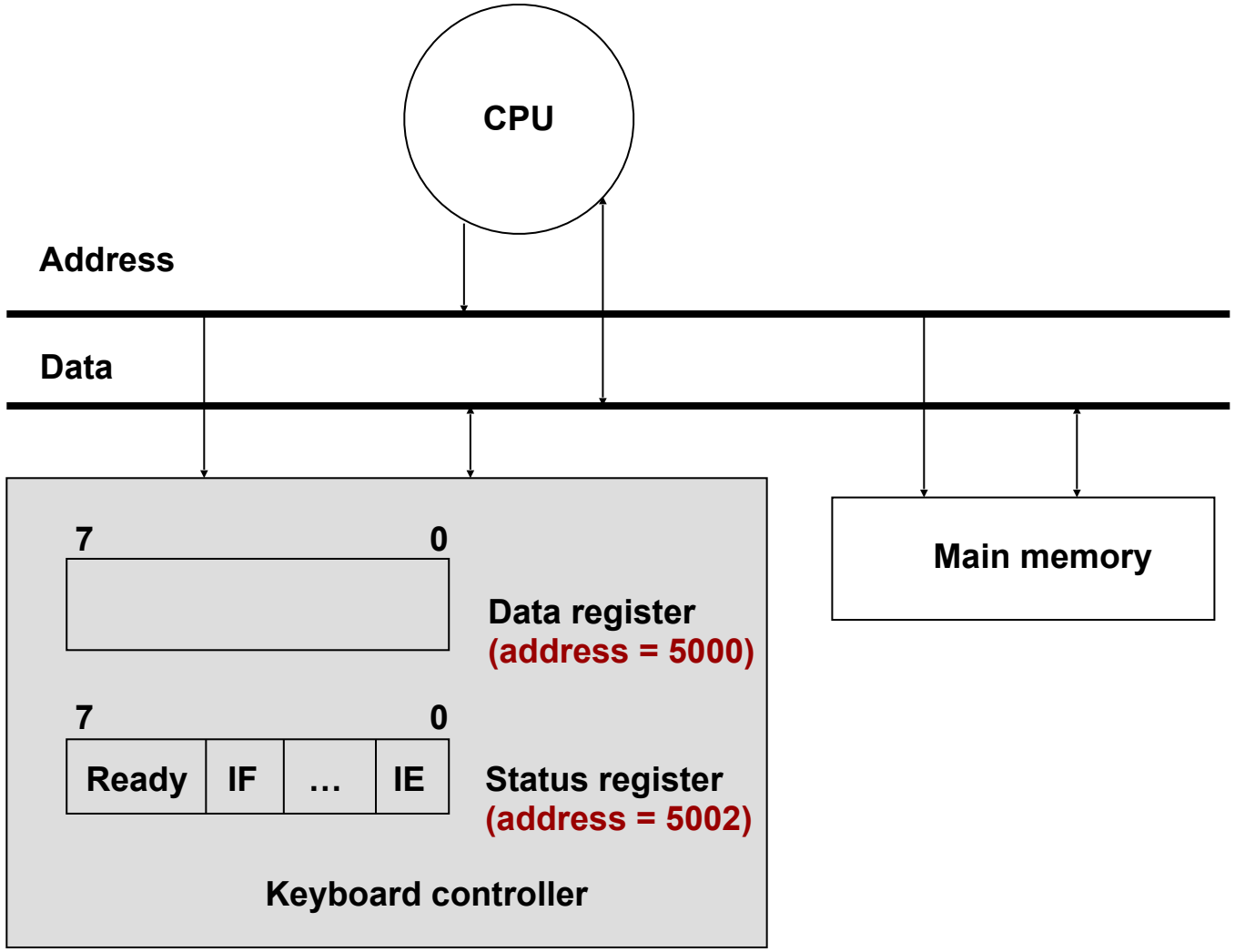
Commands?

Check for new data

LD R1,mem[5002]

Read data

LD R2,mem[5000]



Memory mapped I/O

Commands?

Check for new data

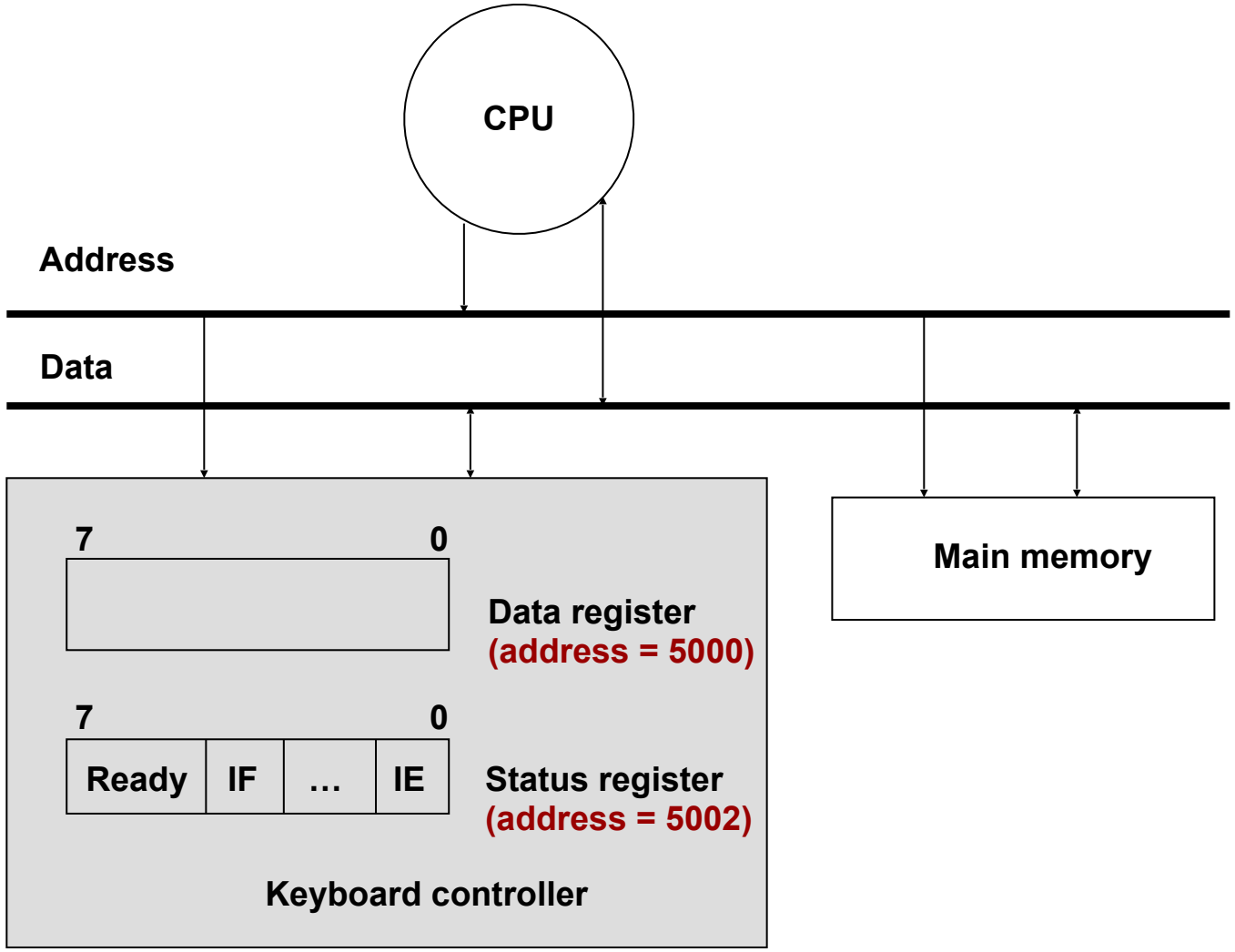
LD R1,mem[5002]

Read data

LD R2,mem[5000]

Set IE

ST #1,mem[5002]



Programmed I/O (PIO)

Programmed I/O (PIO)

- Slow speed devices (Keyboard, mouse, etc.)

Programmed I/O (PIO)

- Slow speed devices (Keyboard, mouse, etc.)
- CPU executes program to move data

Programmed I/O (PIO)

- Slow speed devices (Keyboard, mouse, etc.)
- CPU executes program to move data
- Often uses polling

```
X:  LD      R1, statusreg     ; Get the status register
    BRZP    X                 ; if high bit is 0, branch back one
    LD      R2, datareg       ; Load the data into R2
    ST      #0, statusreg     ; Clear the status register
```

LD usually clears the ready bit as a side effect of reading the data register

Programmed I/O (PIO)

- Slow speed devices (Keyboard, mouse, etc.)
- CPU executes program to move data
- Often uses polling

```
X:  LD      R1, statusreg     ; Get the status register
    BRZP    X                 ; if high bit is 0, branch back one
    LD      R2, datareg       ; Load the data into R2
    ST      #0, statusreg     ; Clear the status register
```

LD usually clears the ready bit as a side effect of reading the data register

- Busy waiting – wastes processor resources

Interrupt-driven I/O

- Slow speed devices (Keyboard, mouse, etc.)
- CPU executes program for moving data
- Can be interrupt driven

```
ST      #1, statusreg ; Set the IE bit
```

Upon interrupt, handler code is executed:

```
LD      R2, datareg     ; Load the data into R2
```

High speed devices

High speed devices

- Once a command is given by the CPU to the controller, data comes in continually
  - Streaming

High speed devices

- Once a command is given by the CPU to the controller, data comes in continually
  - Streaming
- PIO has potential for data loss if the CPU doesn't poll or respond to the interrupt quickly enough

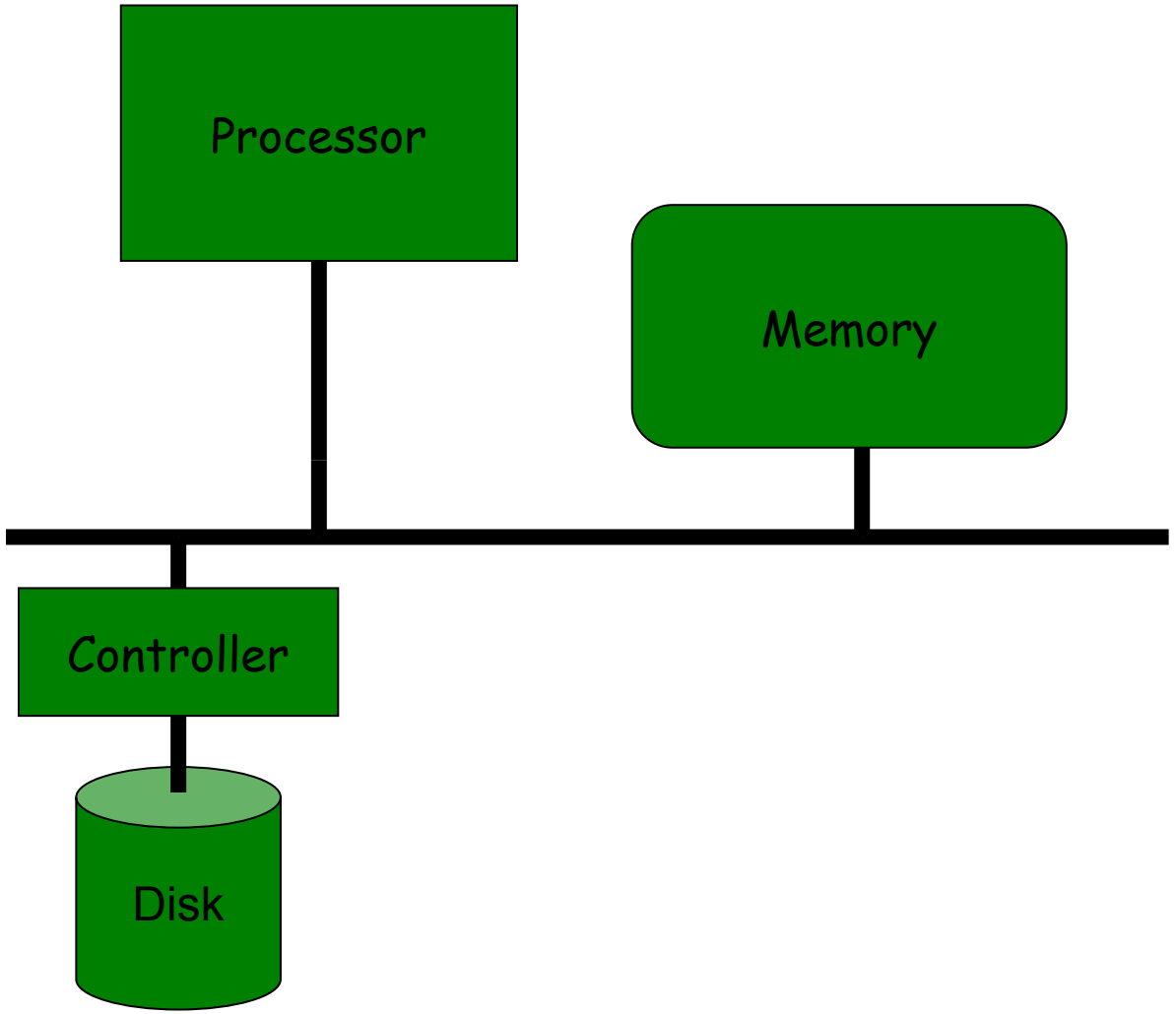
High speed devices

- Once a command is given by the CPU to the controller, data comes in continually
  - Streaming
- PIO has potential for data loss if the CPU doesn't poll or respond to the interrupt quickly enough
- Streaming devices move data to/from memory autonomously

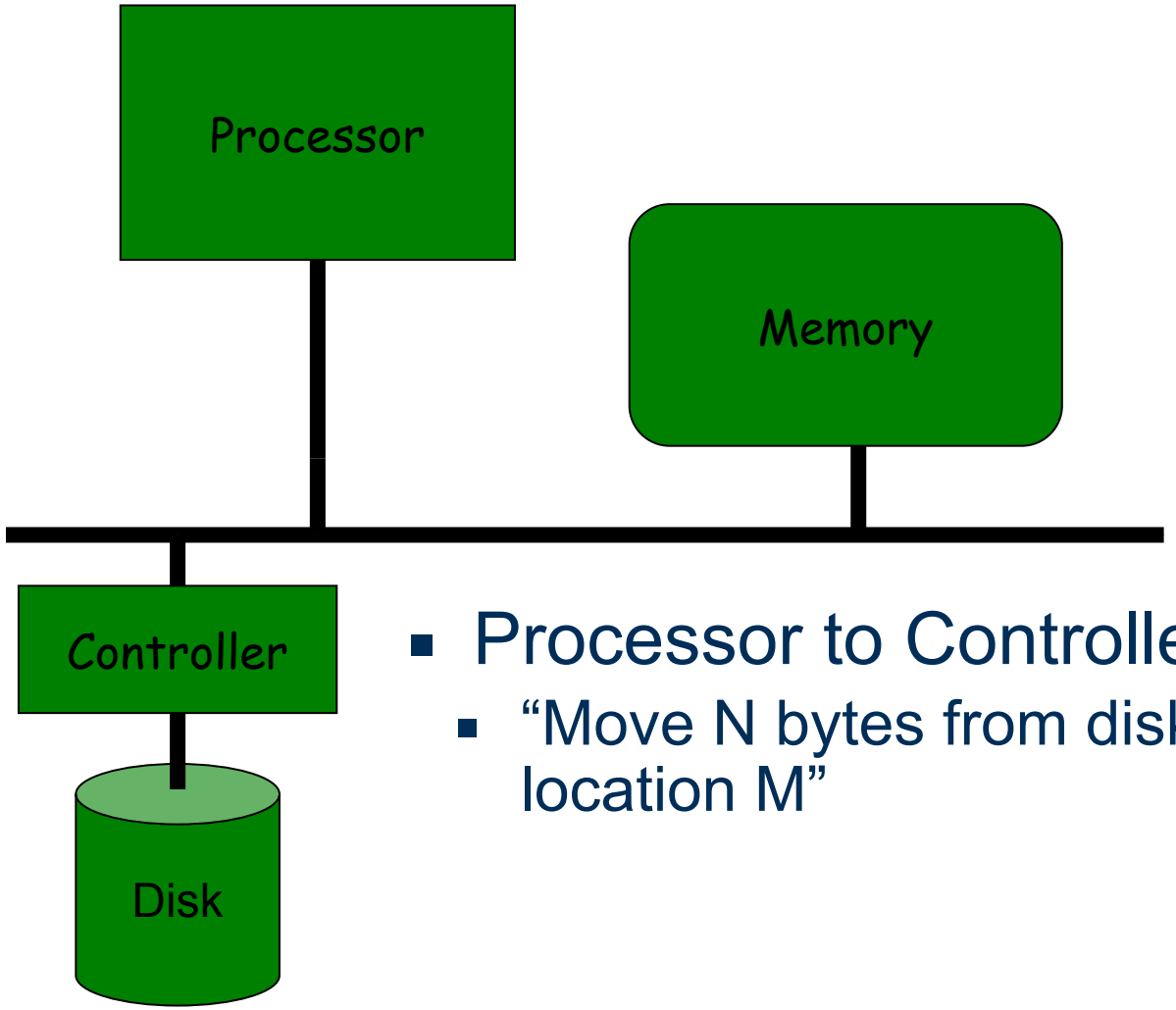
High speed devices

- Once a command is given by the CPU to the controller, data comes in continually
  - Streaming
- PIO has potential for data loss if the CPU doesn't poll or respond to the interrupt quickly enough
- Streaming devices move data to/from memory autonomously
- The CPU to controller interface?
  - Convey commands (read/write, IE, etc.)
  - Check status (error, etc.)

Direct Memory Access - DMA

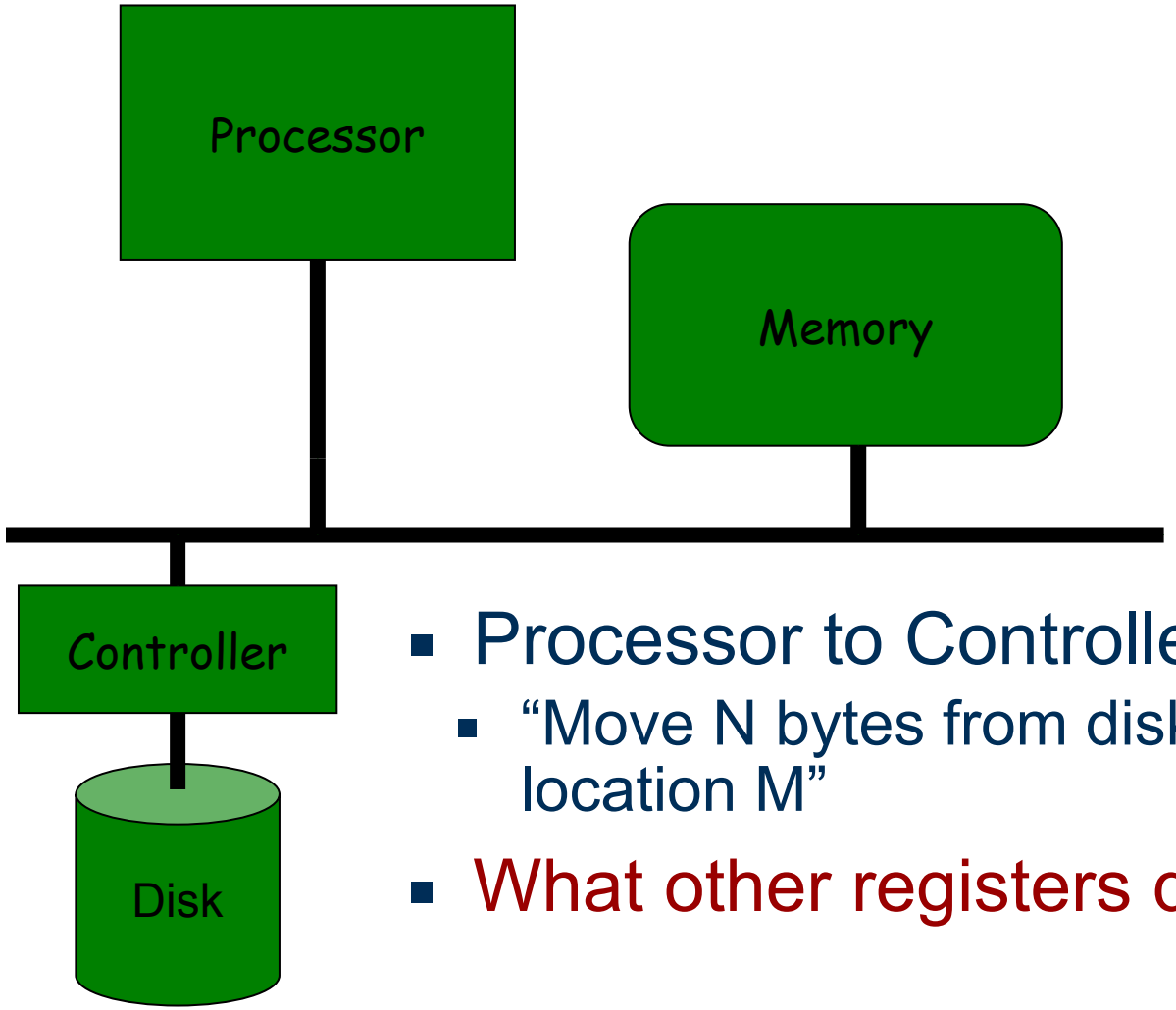


Direct Memory Access - DMA



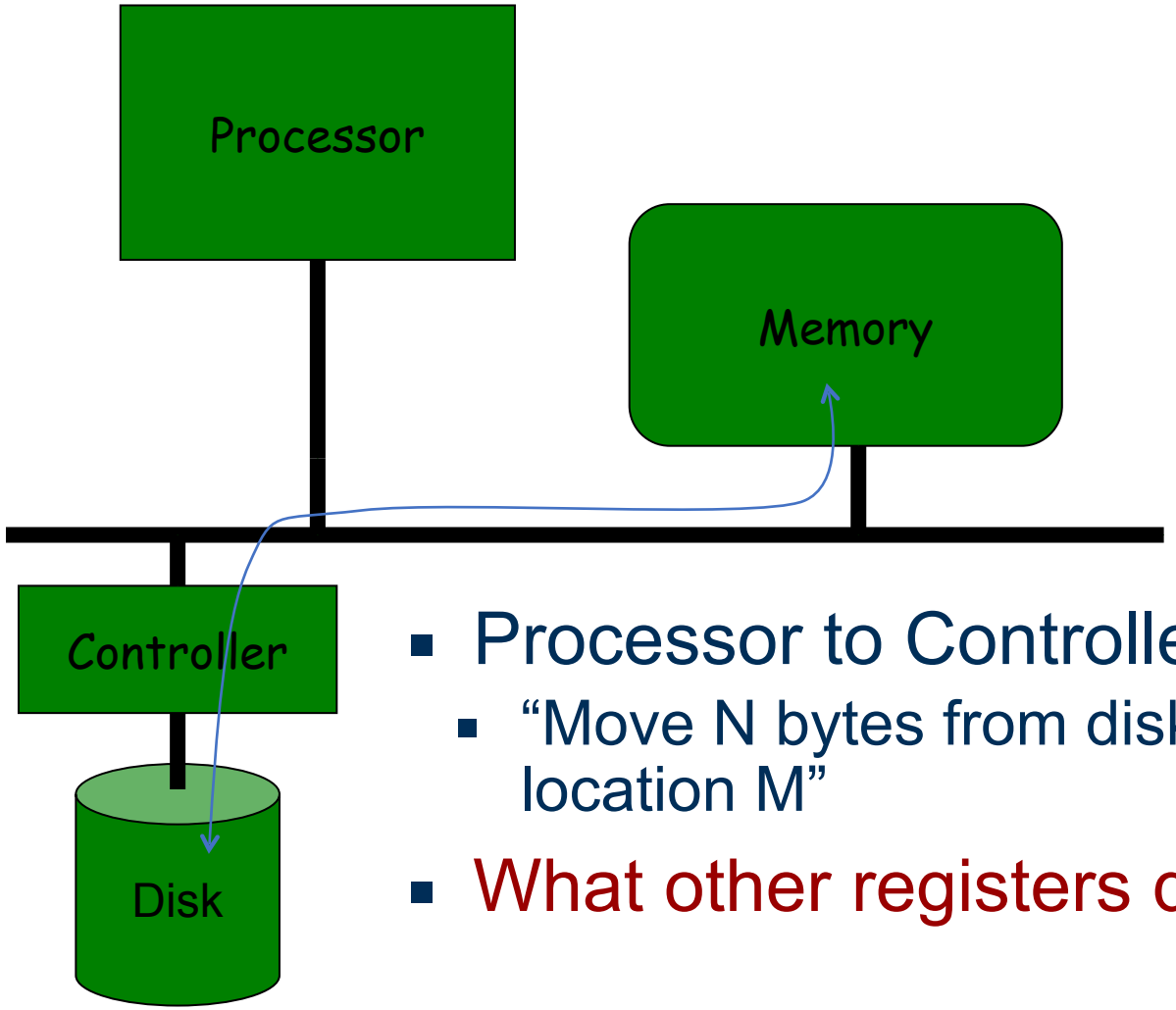
- Processor to Controller:
  - "Move N bytes from disk location X to memory location M"

Direct Memory Access - DMA



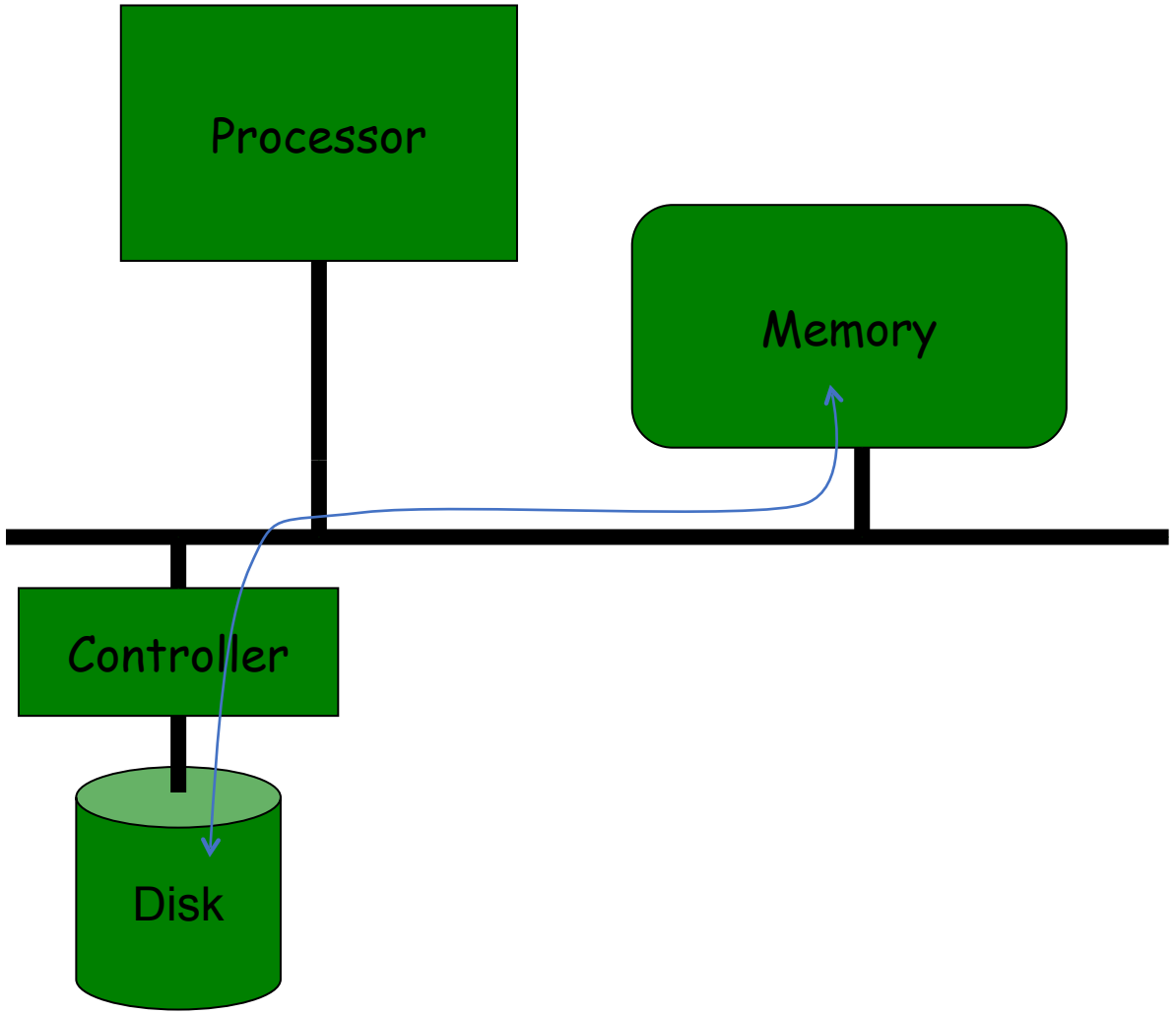
- Processor to Controller:
  - "Move N bytes from disk location X to memory location M"
- What other registers do we need?

Direct Memory Access - DMA

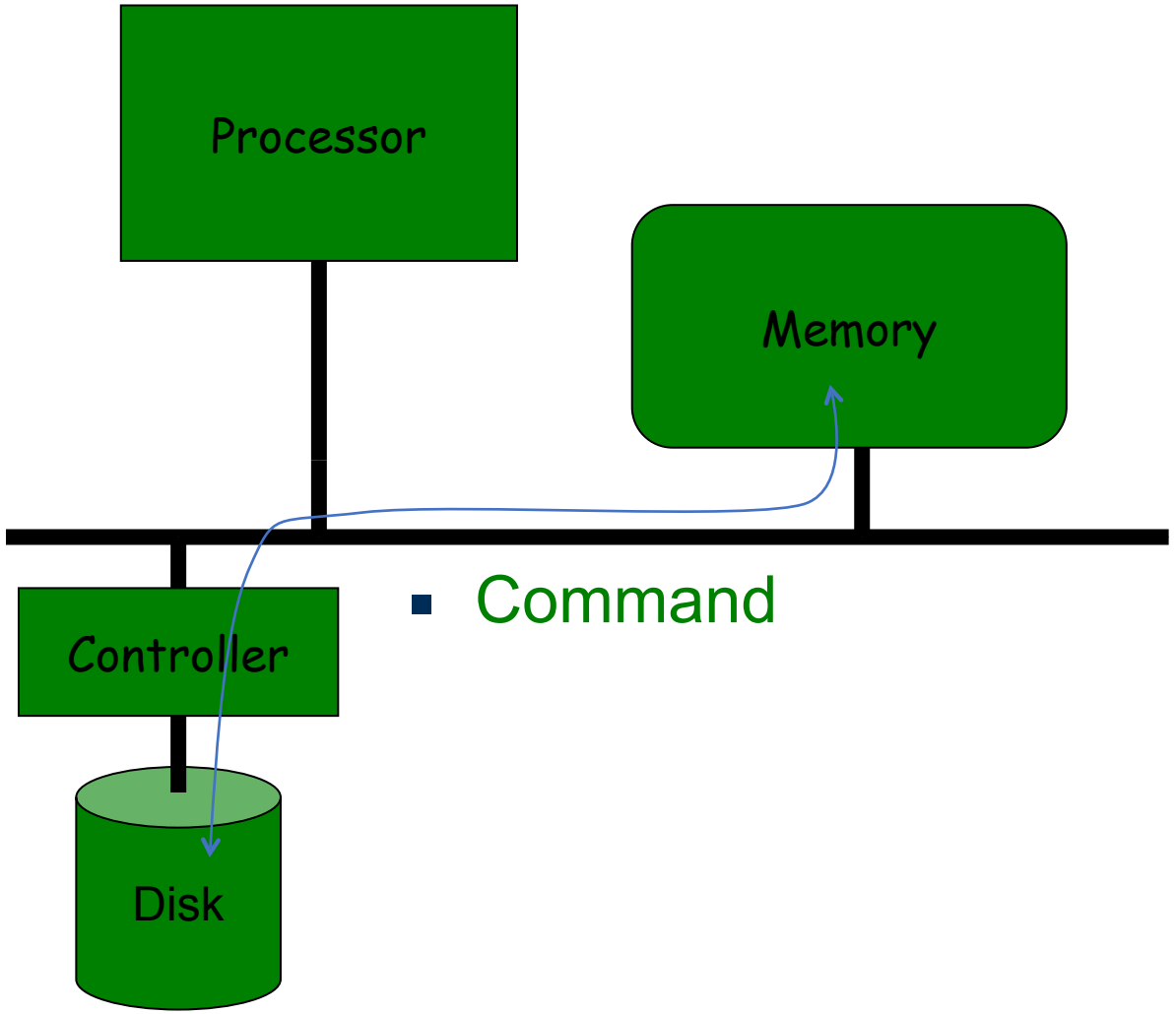


- Processor to Controller:
  - "Move N bytes from disk location X to memory location M"
- What other registers do we need?

DMA registers

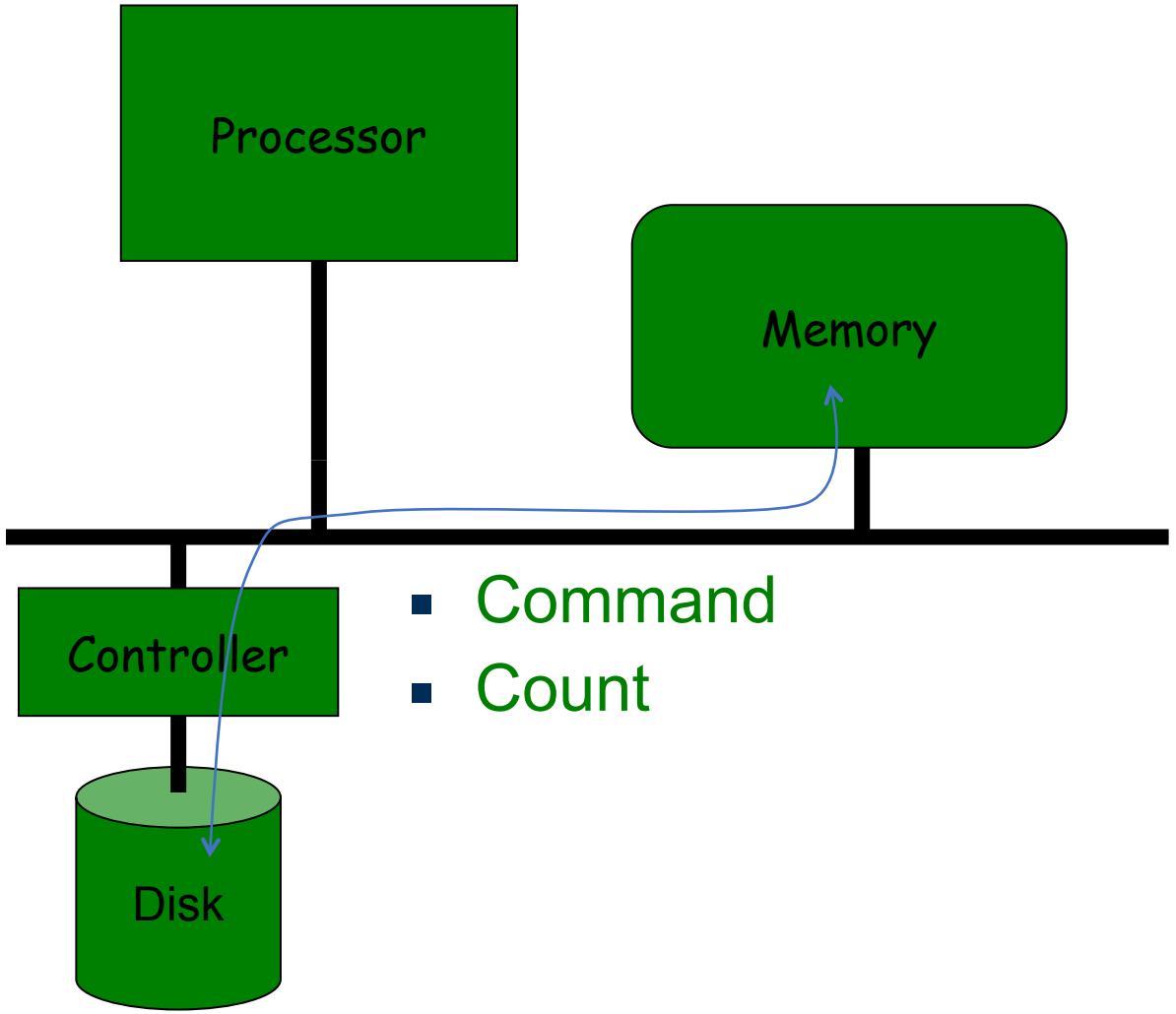


DMA registers



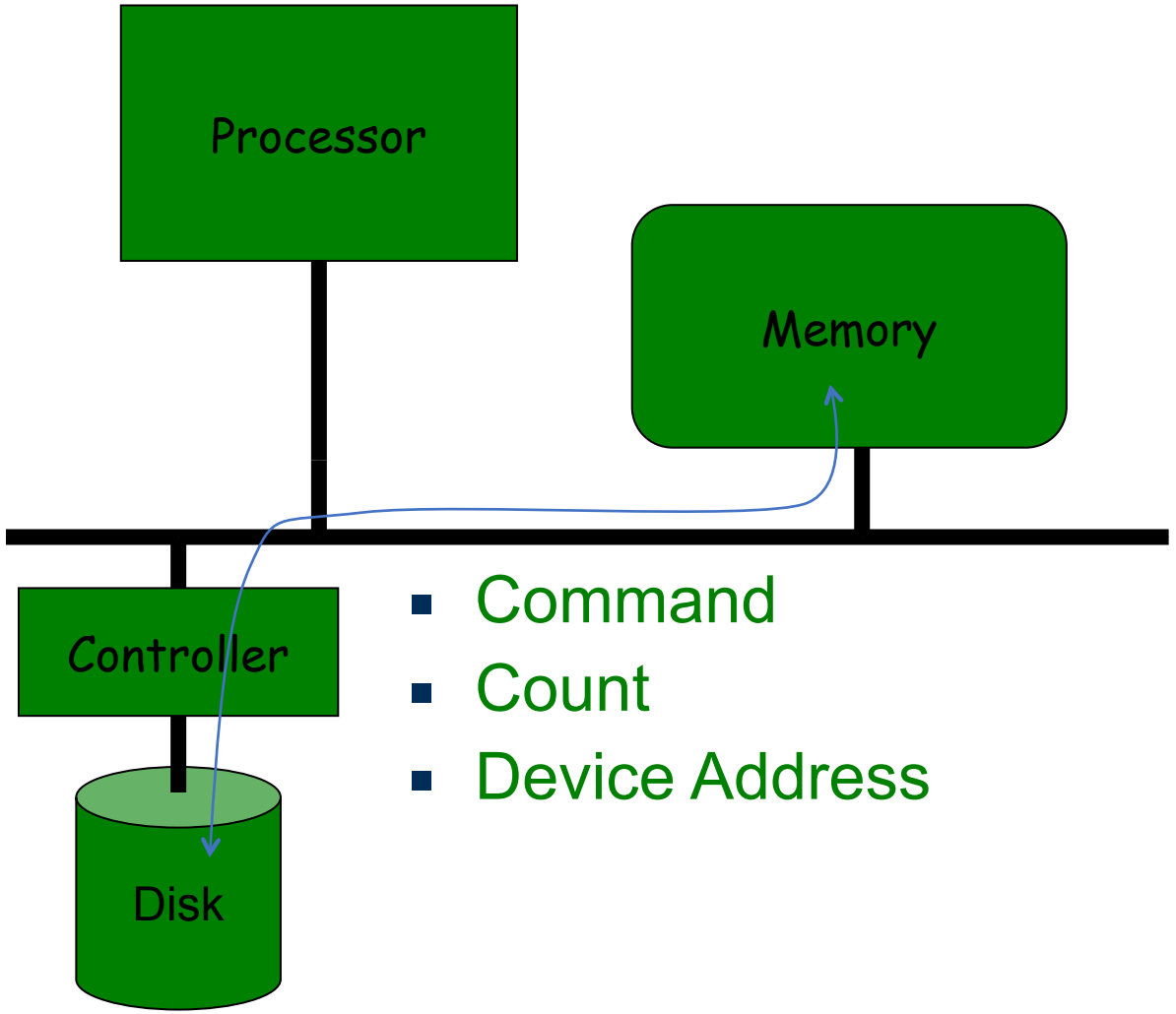
- Command

DMA registers



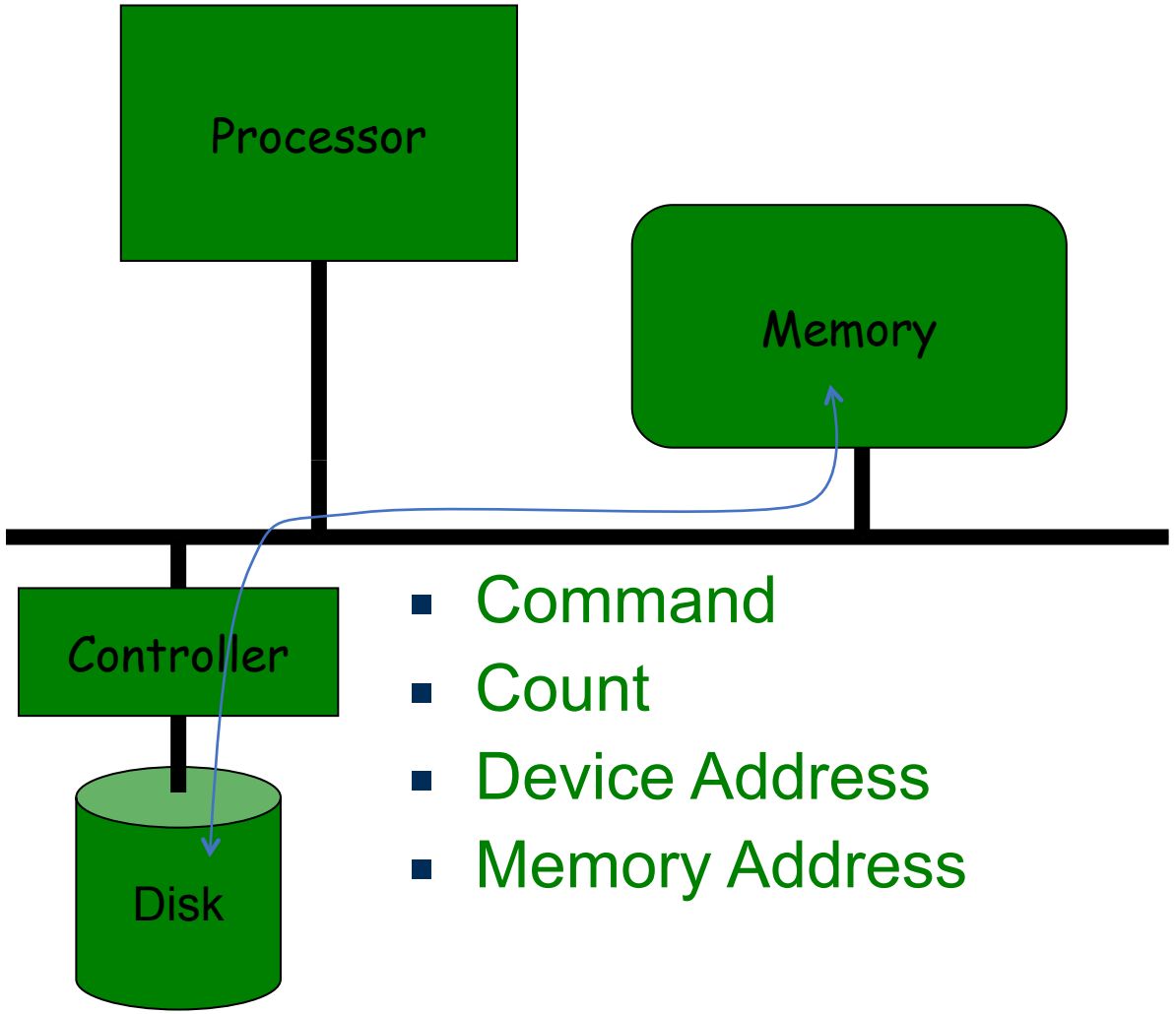
- Command
- Count

DMA registers



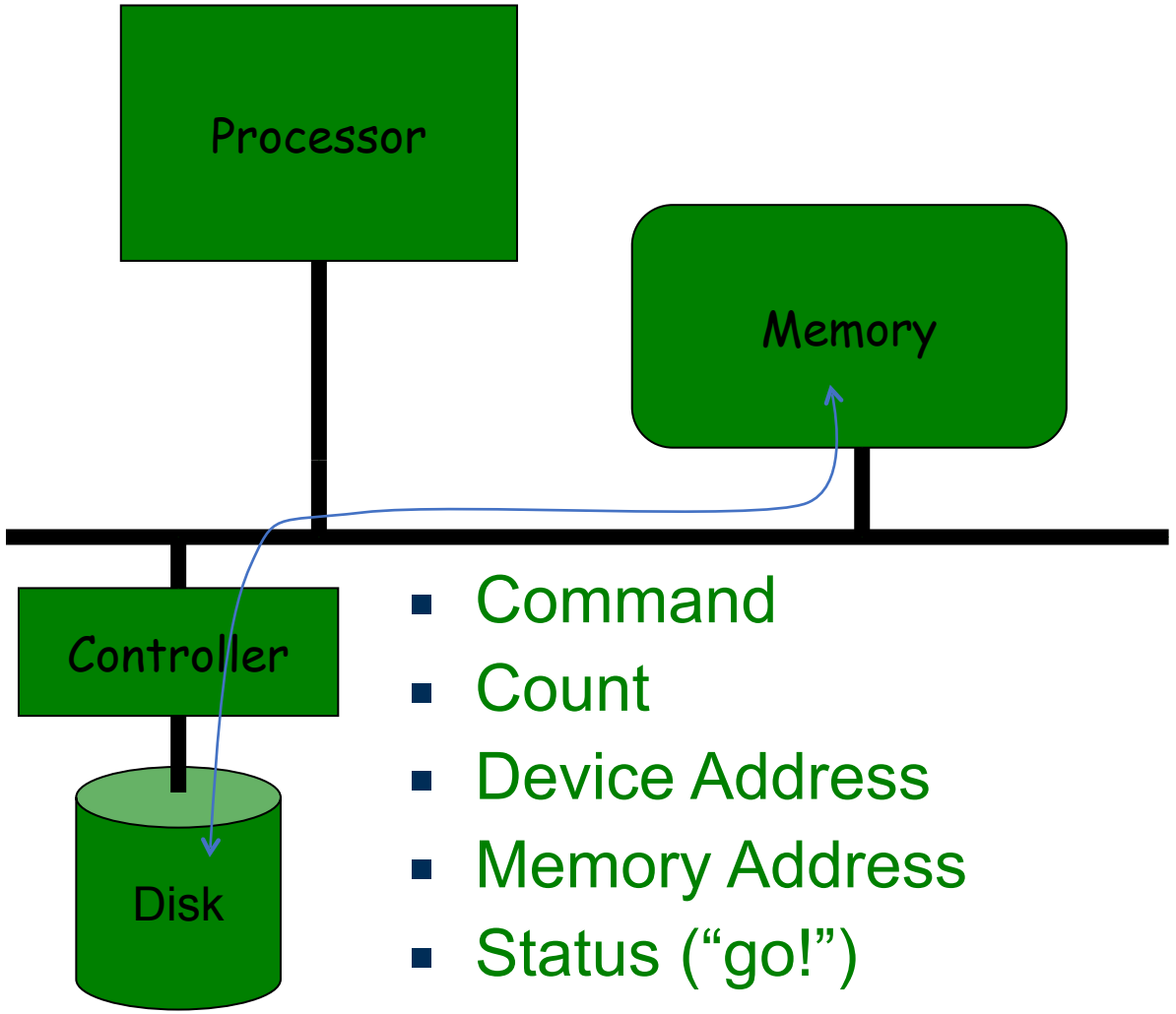
- Command
- Count
- Device Address

DMA registers



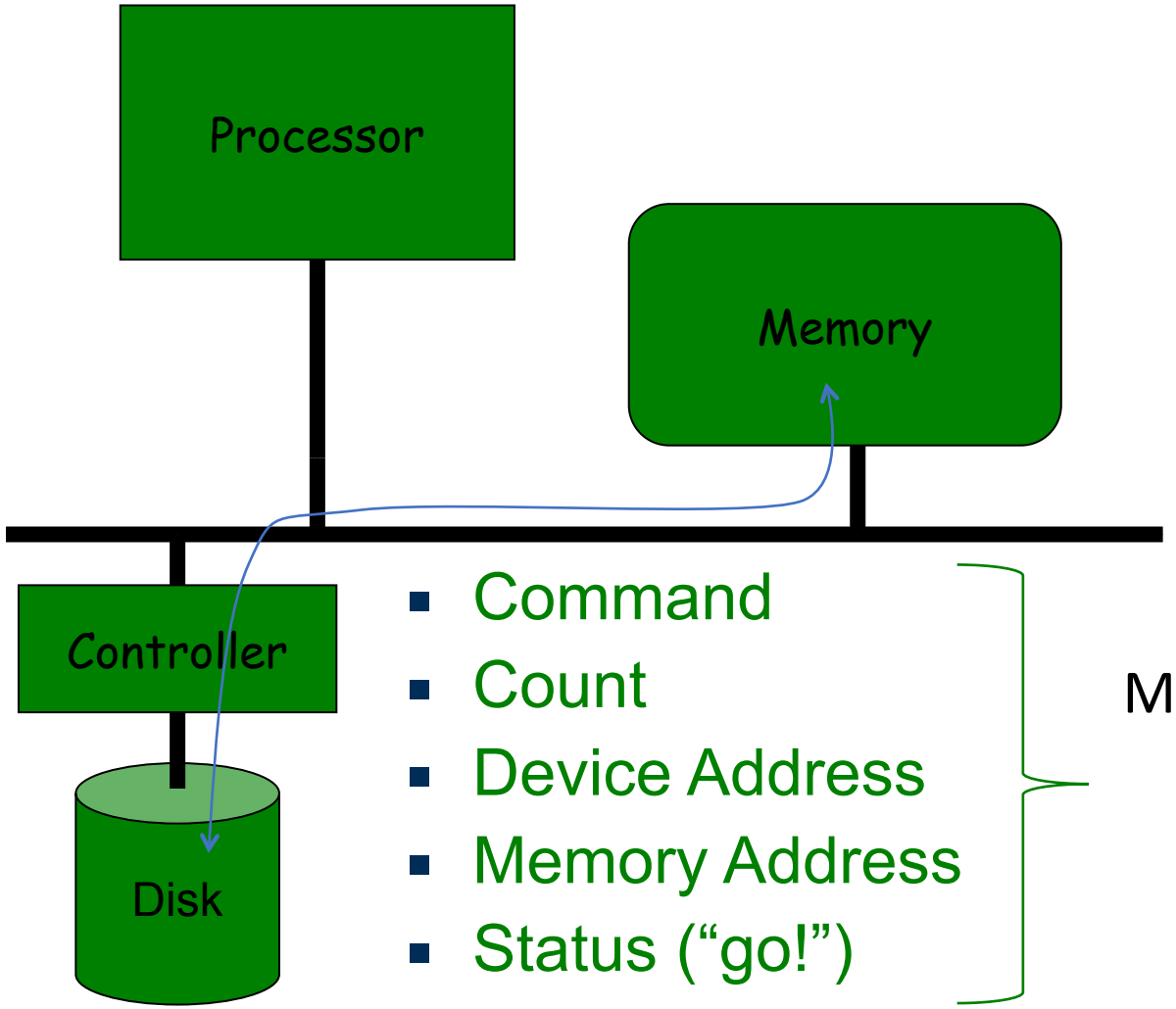
- Command
- Count
- Device Address
- Memory Address

DMA registers



- Command
- Count
- Device Address
- Memory Address
- Status (“go!”)

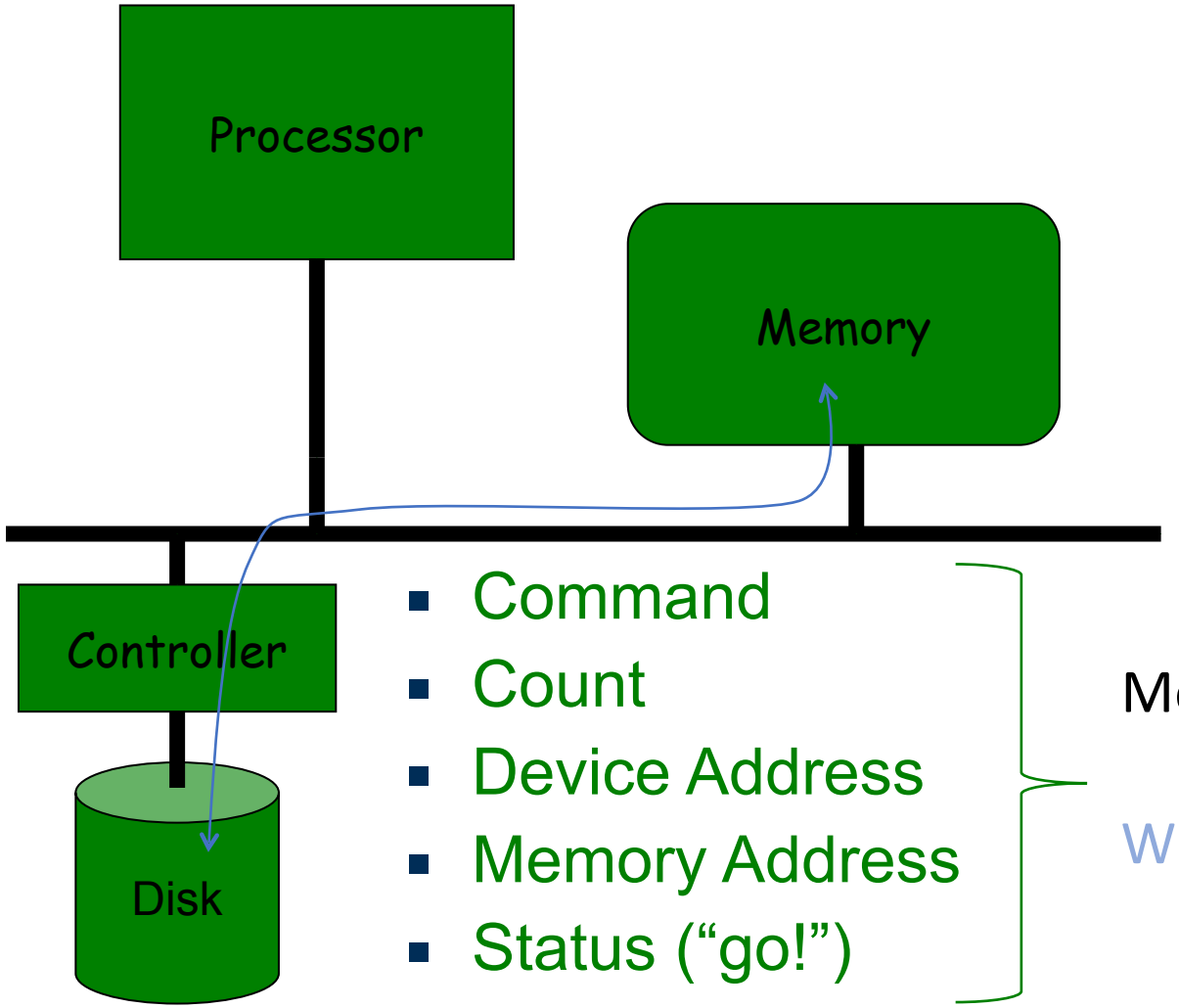
DMA registers



- Command
- Count
- Device Address
- Memory Address
- Status (“go!”)

Memory mapped registers

DMA registers



- Command
- Count
- Device Address
- Memory Address
- Status (“go!”)

Memory mapped registers

Where have you seen this before?

Program for conveying I/O commands

Program for conveying I/O commands

```
Store  N, count
Store  #block_num, device_addr
Store  #mem_buf_addr, mem_addr
Store  #write_command, command
Store  #1, status  ;  the signal to execute the command
```

Program for conveying I/O commands

```
Store  N, count
Store  #block_num, device_addr
Store  #mem_buf_addr, mem_addr
Store  #write_command, command
Store  #1, status   ;  the signal to execute the command
```

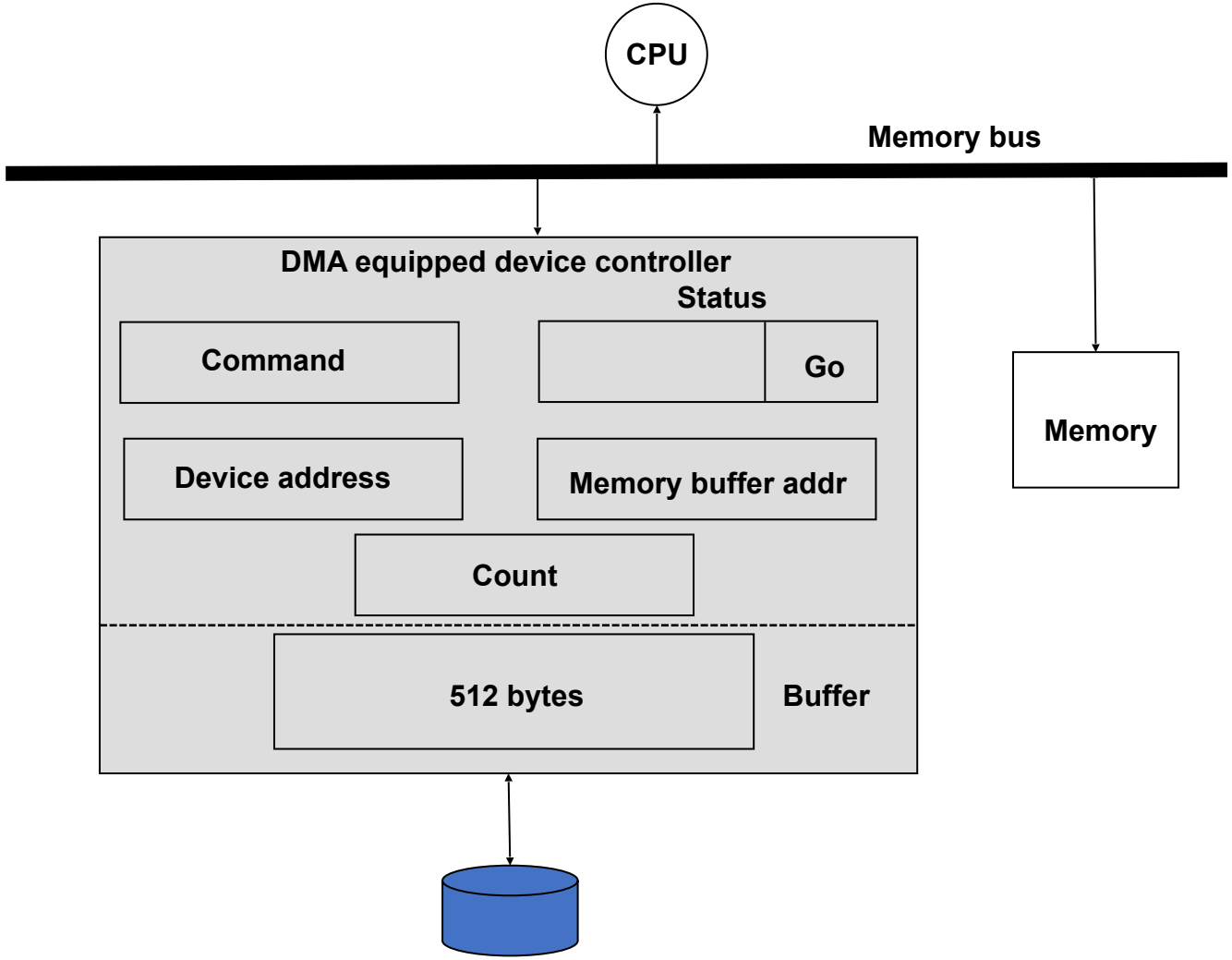
- At this point the CPU's work is done
  - The device controller takes over to do the actual data movement

Program for conveying I/O commands

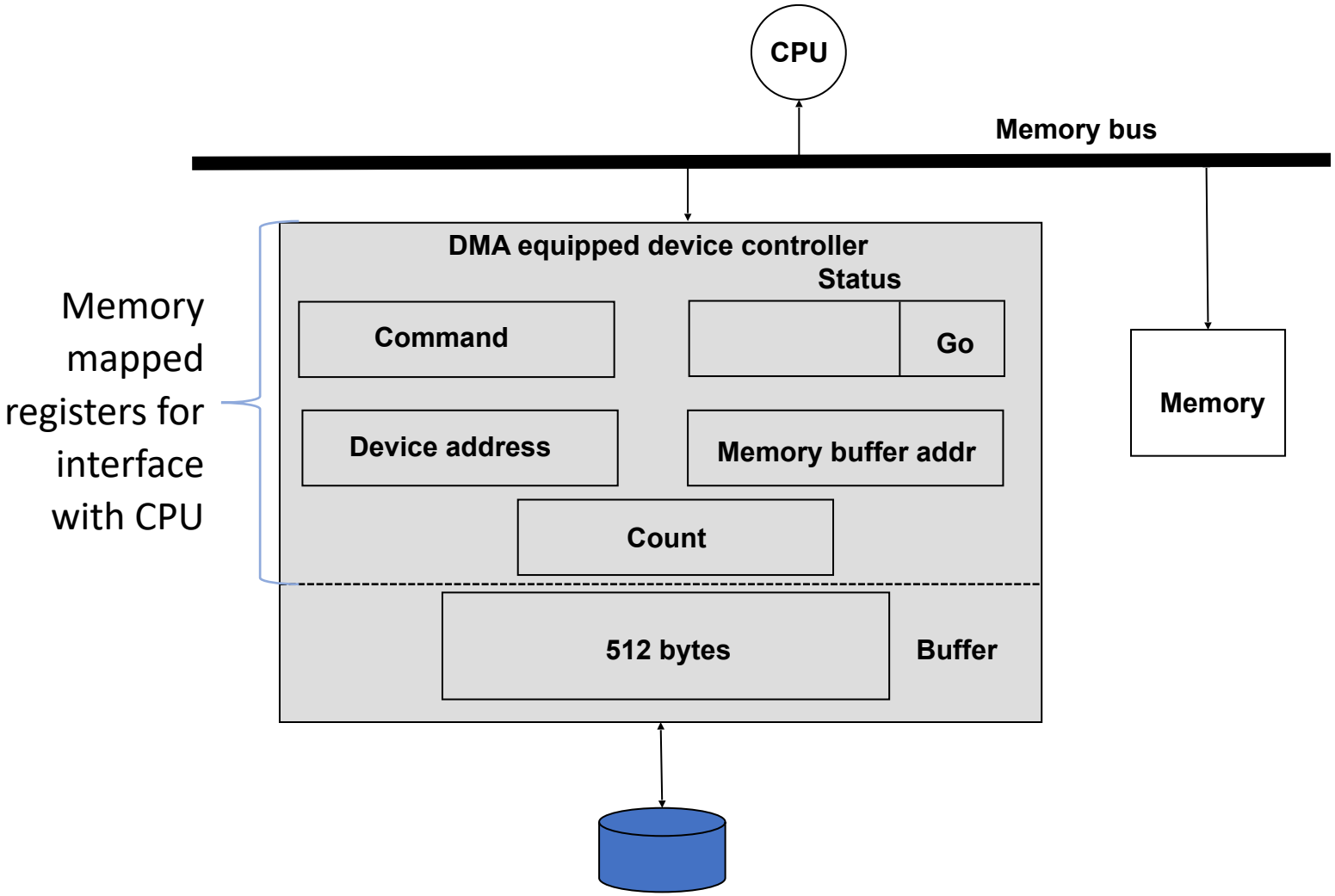
```
Store  N, count
Store  #block_num, device_addr
Store  #mem_buf_addr, mem_addr
Store  #write_command, command
Store  #1, status  ;  the signal to execute the command
```

- At this point the CPU's work is done
  - The device controller takes over to do the actual data movement
- Modern CPU chips already contain device controllers for PCI-E devices which share the on-chip memory controllers with the CPU cores

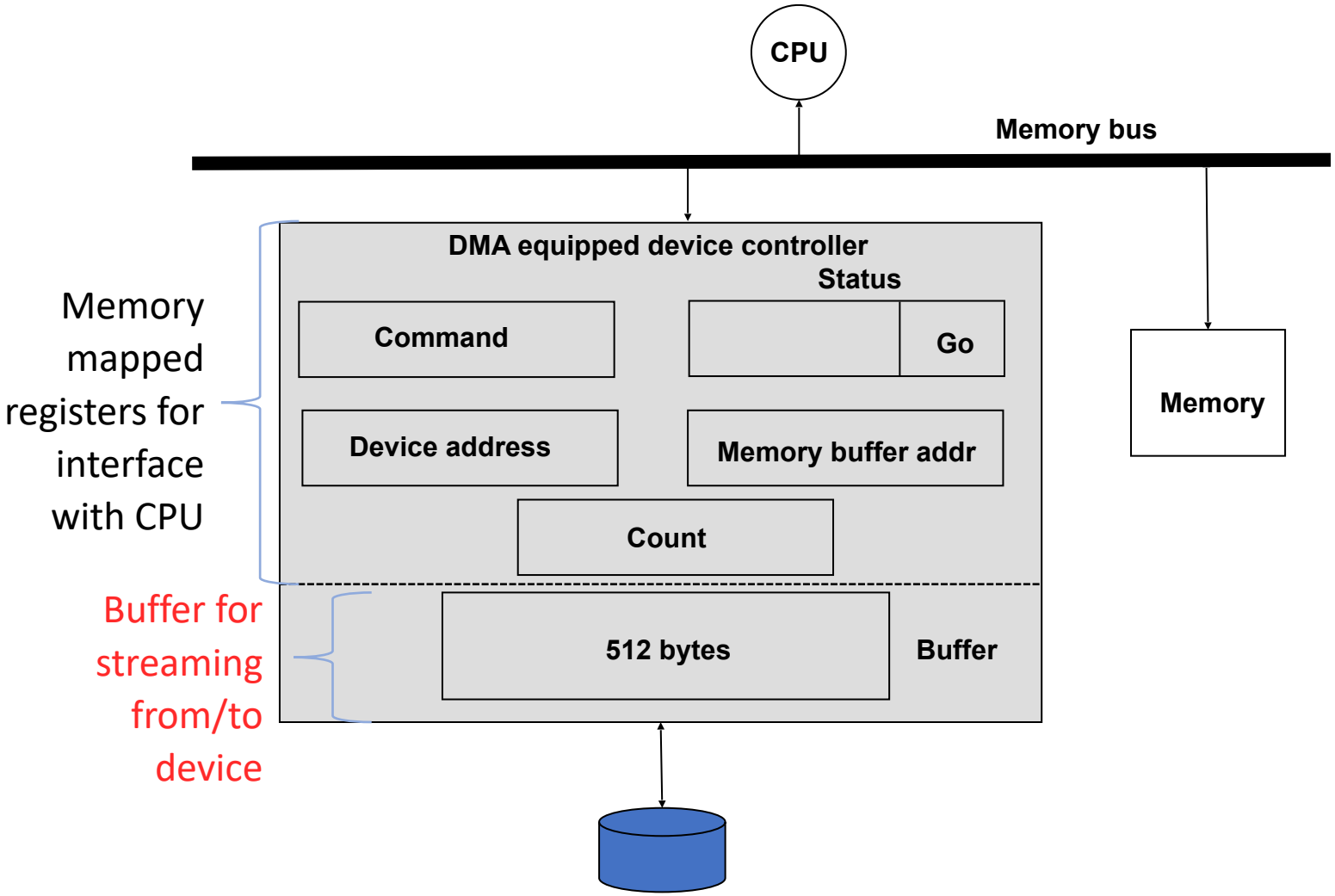
DMA


CPU
Memory bus
DMA equipped device controller
Status
Command
Go
Memory
Device address
Memory buffer addr
Count
512 bytes
Buffer

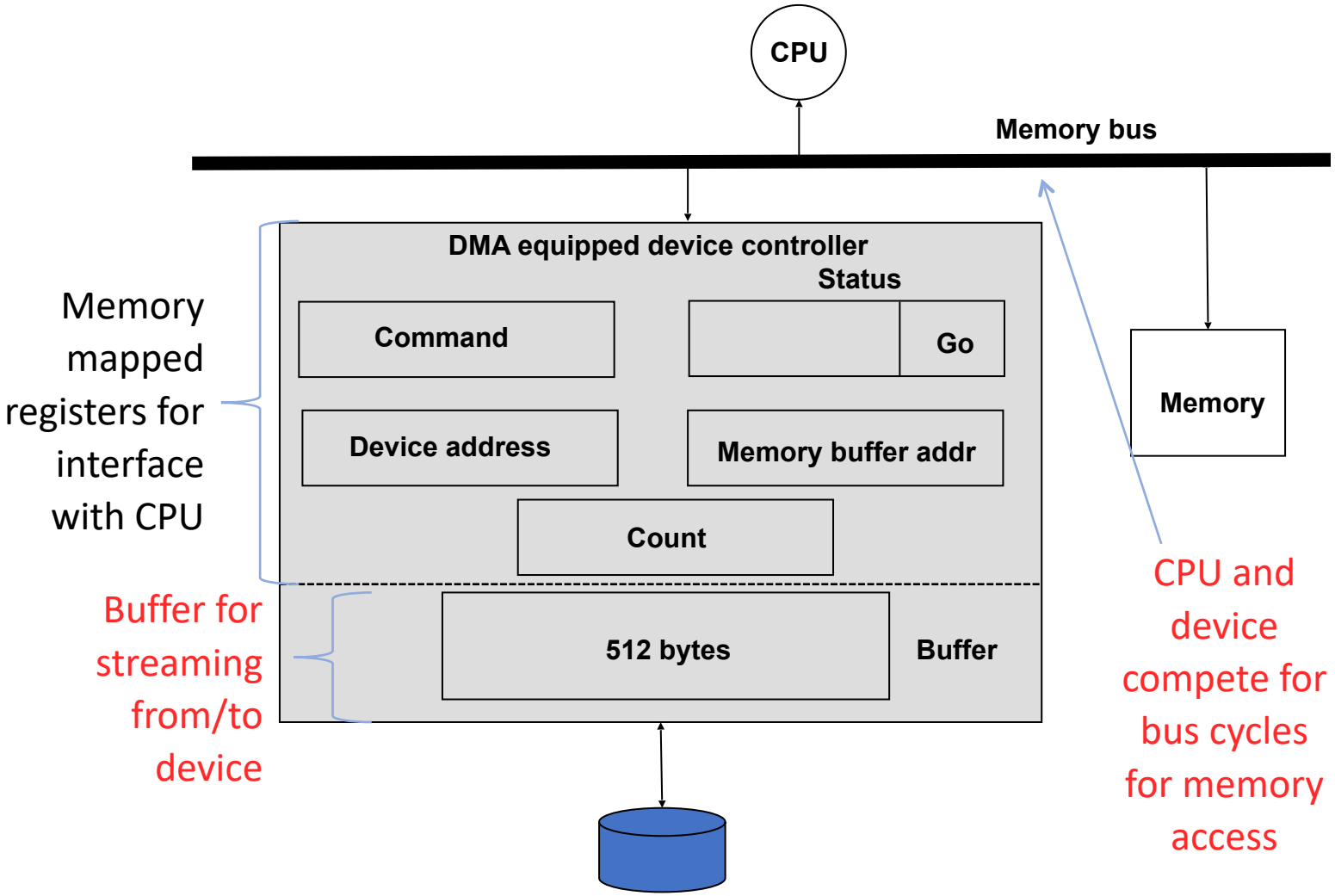
DMA


CPU
Memory bus
DMA equipped device controller
Status
Command
Go
Device address
Memory buffer addr
Count
512 bytes
Buffer
Memory
Memory mapped registers for interface with CPU

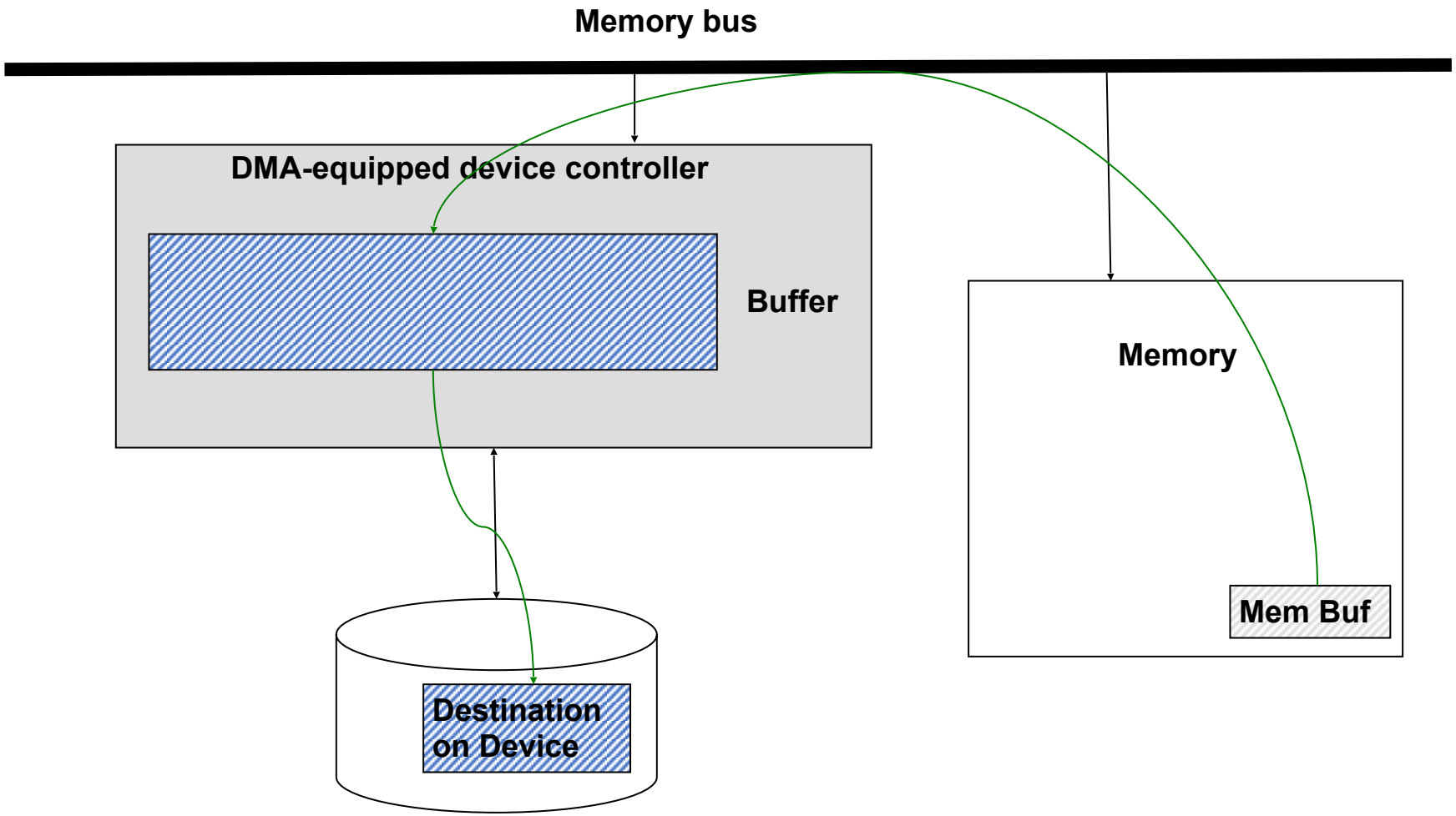
DMA


CPU
Memory bus
DMA equipped device controller
Status
Command
Go
Device address
Memory buffer addr
Count
512 bytes
Buffer
Memory
Memory mapped registers for interface with CPU
Buffer for streaming from/to device

DMA


CPU
Memory bus
DMA equipped device controller
Status
Command
Go
Device address
Memory buffer addr
Count
512 bytes
Buffer
Memory
Memory mapped registers for interface with CPU
Buffer for streaming from/to device
CPU and device compete for bus cycles for memory access

DMA Transfer example



Data transfer speeds

Device	Input/ output	Human in the loop	Data rate (circa 2008)	Data rate (circa 2020)	PIO	DMA
Keyboard	Input	Yes	5-10 bytes/sec	5-10 bytes/sec	X	
Mouse	Input	Yes	80-200 bytes/sec	80-200 bytes/sec	X	
Graphics display	Output	No	200-350 MB/sec	200-350 MB/sec		X
Disk (hard drive)	I/O	No	100-200 MB/sec	500 MB/sec (each)		X
Network (LAN)	I/O	No	1 Gbit/sec	1-400 Gbit/sec		X
Modem	I/O	No	1-8 Mbit/sec	1-8 Mbit/sec		X
Inkjet printer	Output	No	20-40 KB/sec	20-40 KB/sec	X	X
Laser printer	Output	No	200-400 KB/sec	200-400 KB/sec		X
Voice (microphone/ speaker)	I/O	Yes	10 bytes/sec	10 bytes/sec	X	
Solid State (SSD)	I/O	No	10-50 MB/sec	0.1-7GB/sec		X
CD, DVD, Blu-Ray	I/O	No	10-20 MB/sec	10-20 MB/sec		X

DMA is used instead of PIO for I/O because

33% A. PIO can lose data if the CPU doesn't poll or respond to the interrupt quickly enough

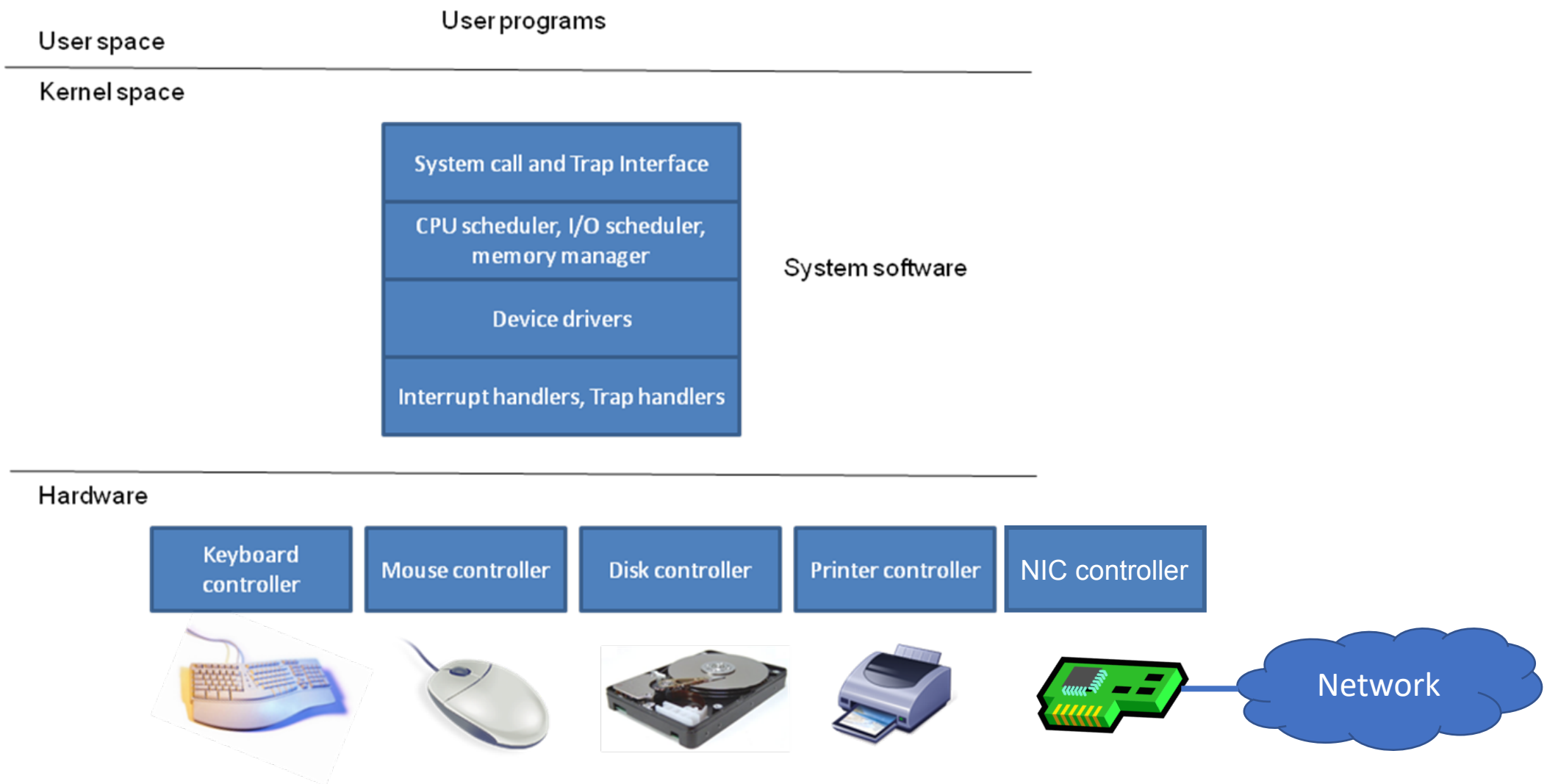
10% B. Servicing an interrupt in the CPU is expensive especially if it has to be done for every word of I/O

10% C. A DMA controller can transfer data into memory in fewer cycles than a programmed loop in the CPU

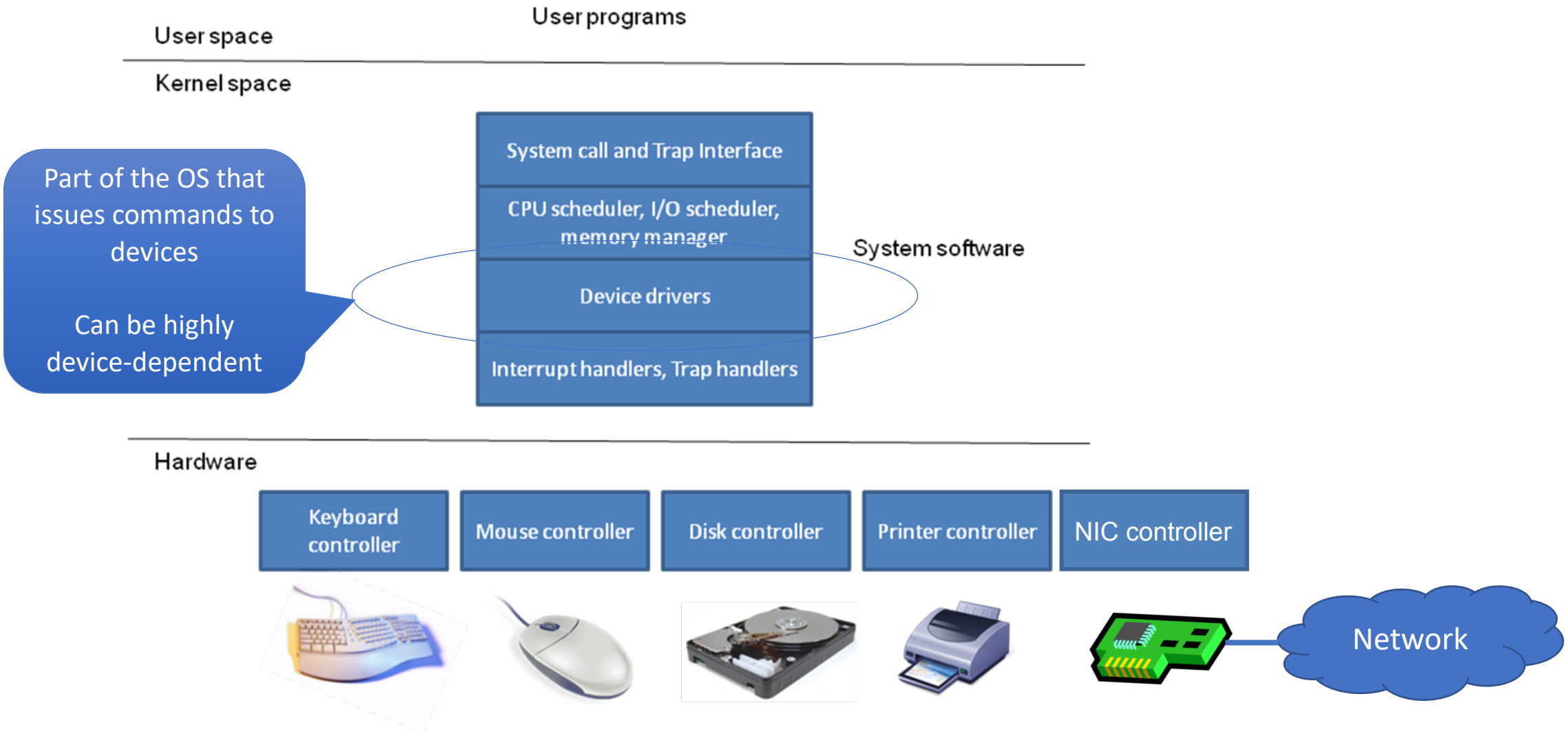
4% D. All of the above

42%

Device drivers and OS



Device drivers and OS



Device drivers and OS