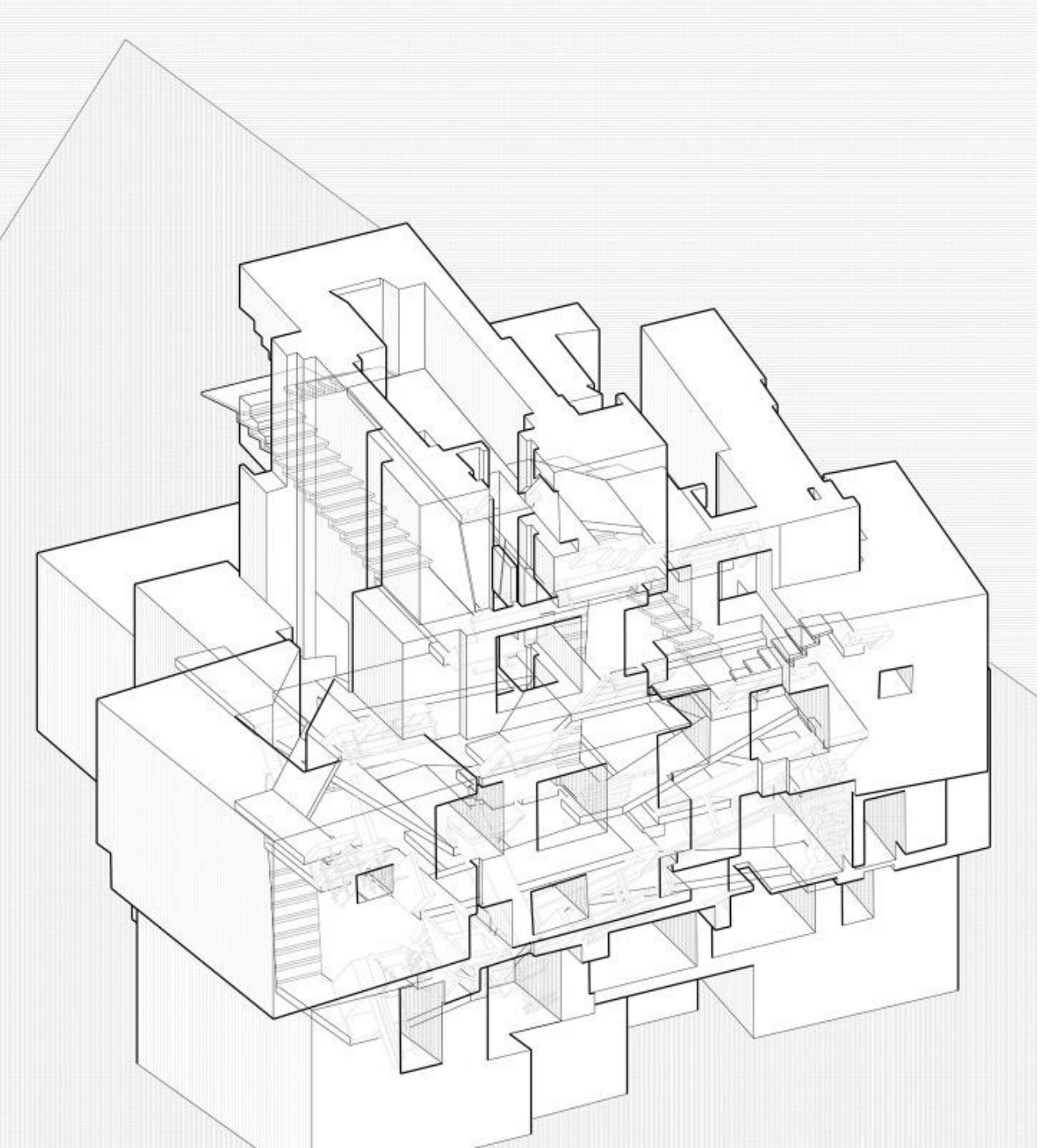


Overview of the MIPS Architecture: Part II

CS 161: Lecture 1

1/26/17

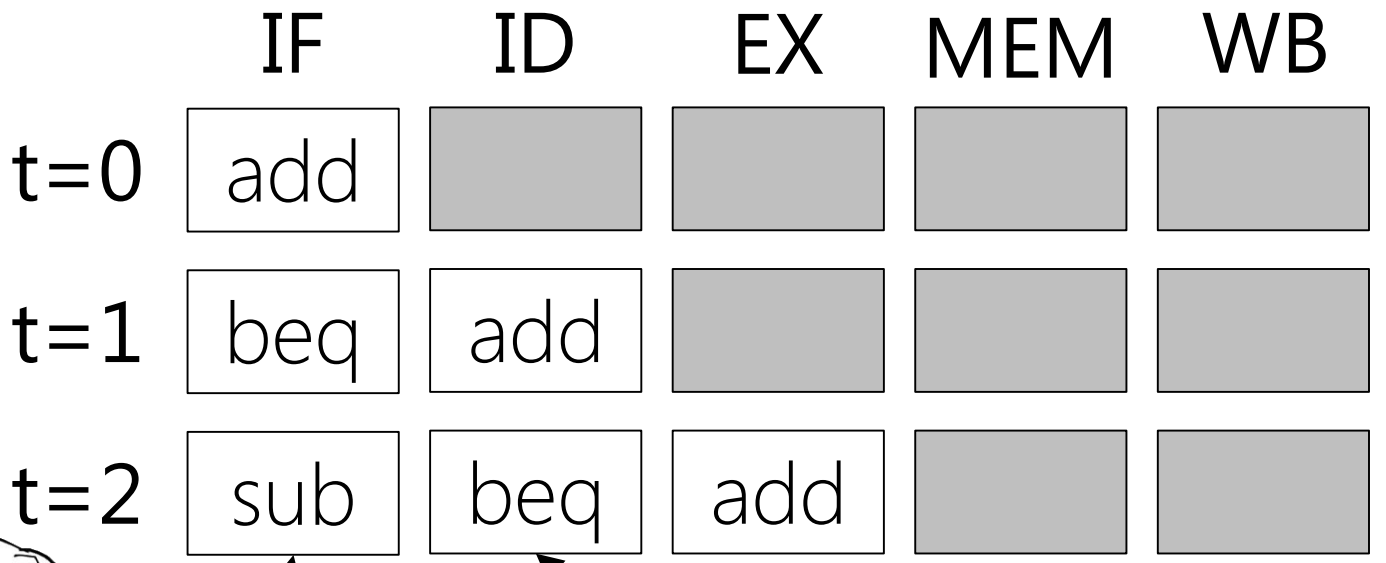
Outline

- Pipelining and branches
- Traps
- Synchronization

The Problem with Branches

- We don't know if a branch is taken until the end of the ID stage . . .
- . . . which means that the IF stage may have fetched the wrong instruction!

```
    add t0, t1, t2
    beq t3, zero, lbl
    sub a0, a1, a2
    . . .
lbl: lw t4, 16(t5)
```



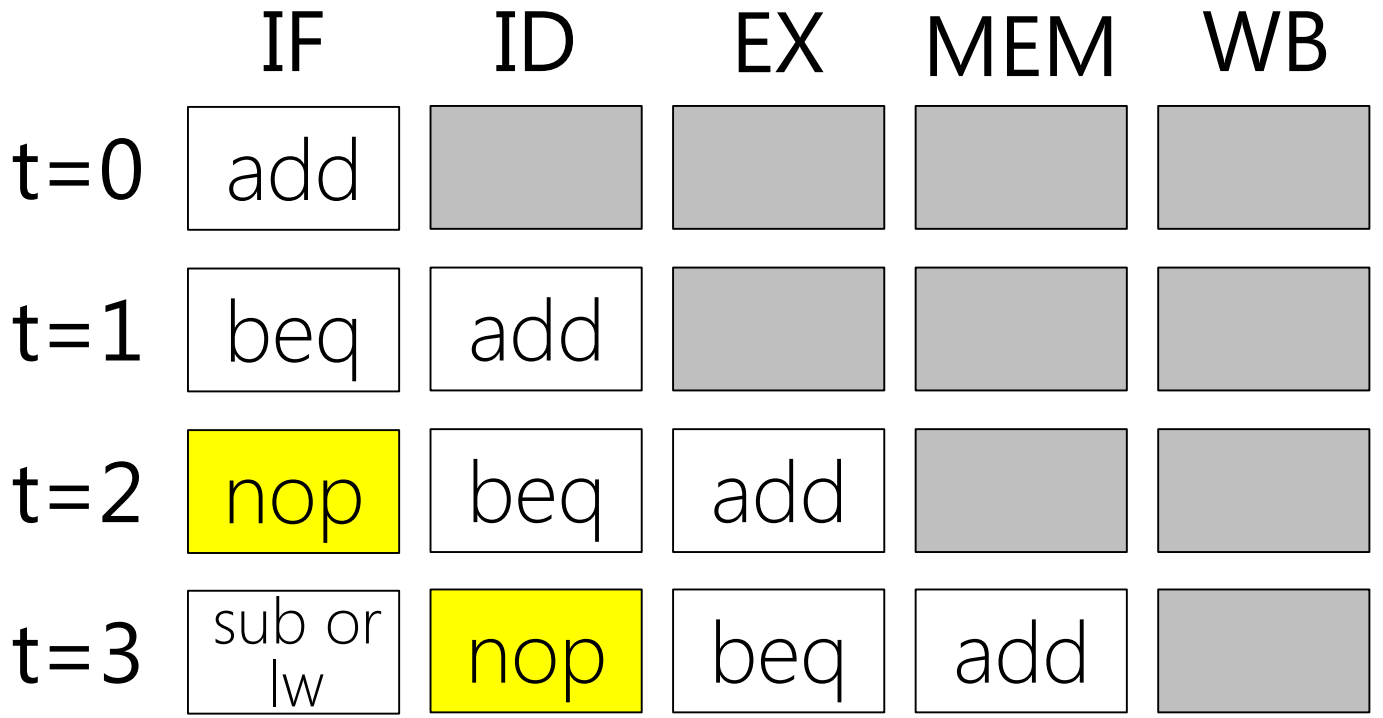
We don't know whether we should branch until the end of t=2 . . .

. . . so we don't know whether lw should have been fetched until end of t=2!

The Problem with Branches

- One solution: Processor automatically inserts a nop after each branch
  - A nop ("no operation") does not change the processor's state
  - So, executing a nop never affects correctness (although it does slow down the program due to a wasted processor cycle)

```
      add t0, t1, t2
      beq t3, zero, lbl
      sub a0, a1, a2
      . . .
lbl:  lw t4, 16(t5)
```



At beginning of t=3, IF examines output of ID from t=2 and fetches the appropriate instruction

The Problem with Branches

- Different solution: Have compiler insert a “branch delay” instruction after a branch
  - This instruction must be one that a program should ALWAYS execute, regardless of whether branch is taken or not!
  - If the program has no such instruction, compiler inserts a nop

```
        add t0, t1, t2
        beq t3, zero, lbl
        sub a0, a1, a2
        . . .
lbl:    lw t4, 16(t5)
```

If compiler emits this code, then the program should always execute the **sub**, regardless of whether the branch is taken

The Problem with Branches

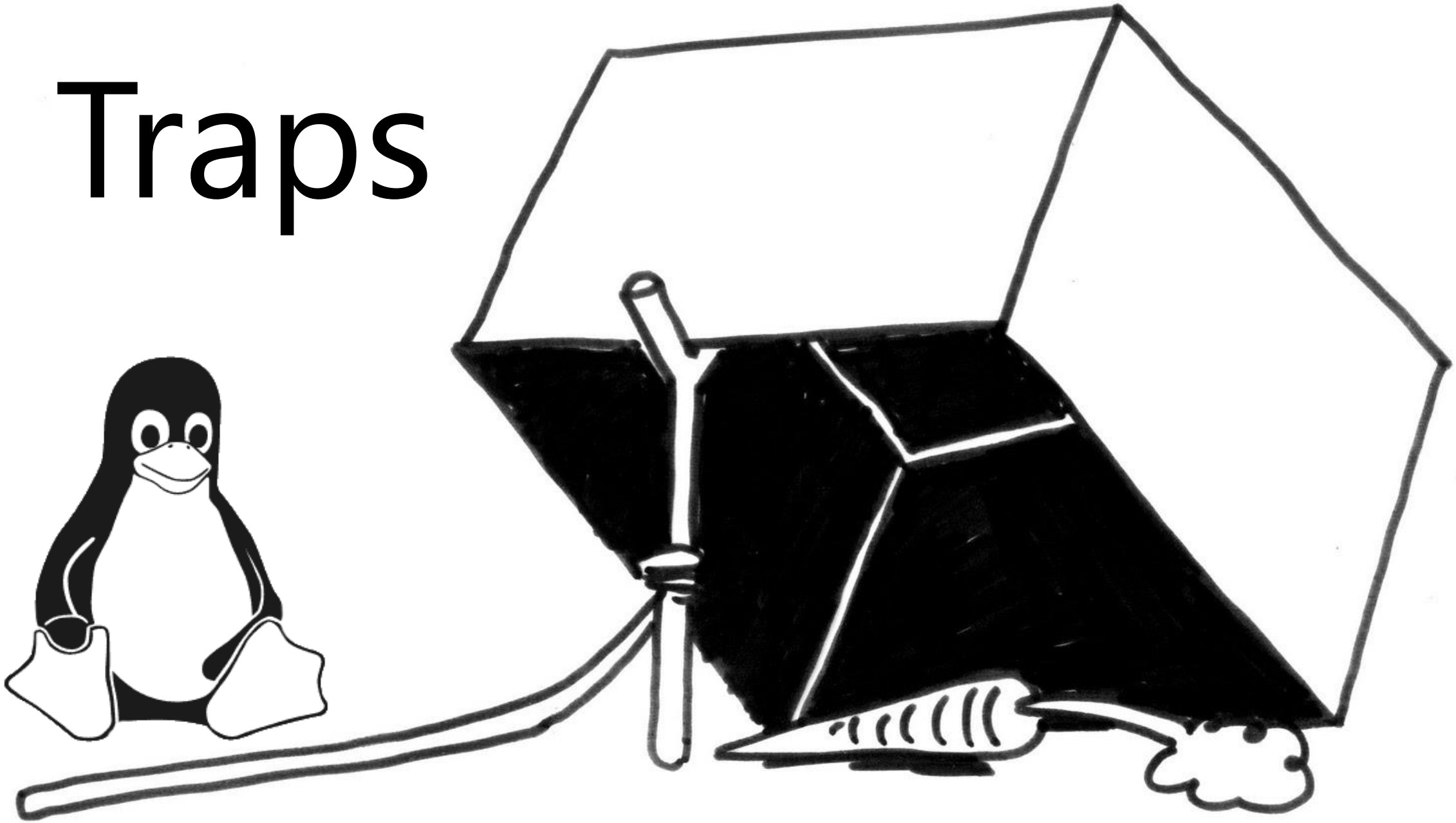
- Different solution: Have compiler insert a "branch delay" instruction after a branch
  - This instruction must be one that a program should ALWAYS execute, regardless of whether branch is taken or not!
  - If the program has no such instruction, compiler inserts a nop

```
        add t0, t1, t2
        beq t3, zero, lbl
        nop
        sub a0, a1, a2
        . . .
lbl:    lw t4, 16(t5)
```

If compiler emits this code, then the program should only execute the **sub** if the branch is NOT taken

- MIPS R3000 uses the branch delay approach

Traps



Invoking the OS

User-level app

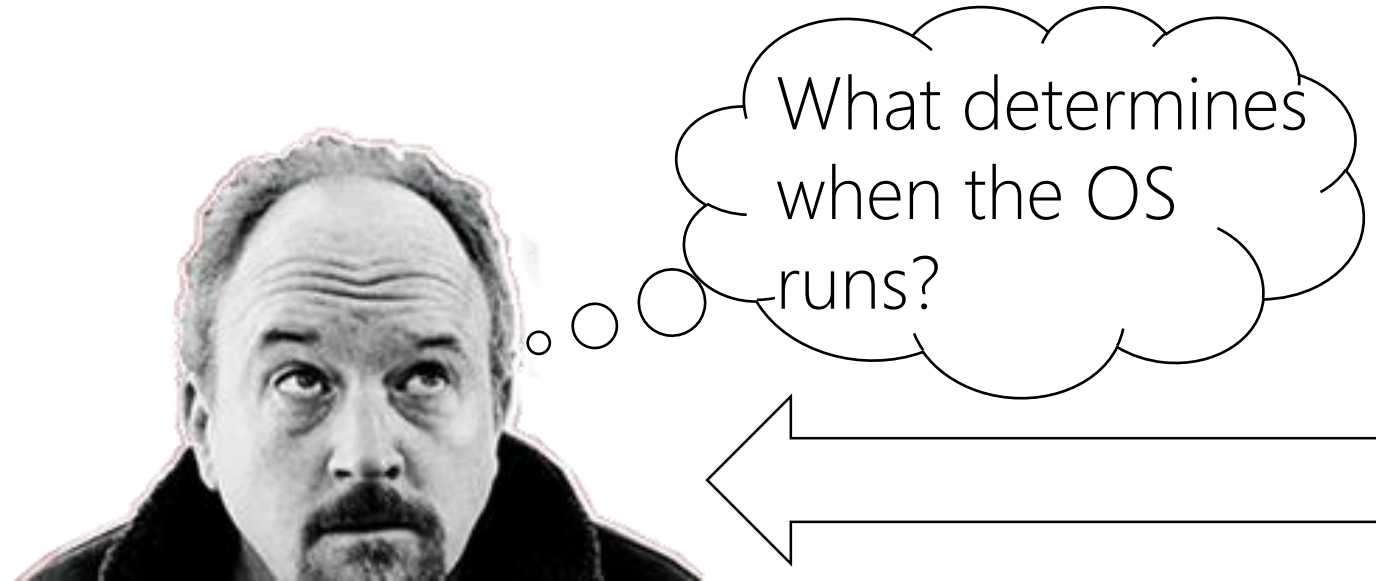
Operating system

Hardware

The OS contains executable instructions, just like a user-level application!

```
app_instr0
app_instr1
app_instr2
...
```

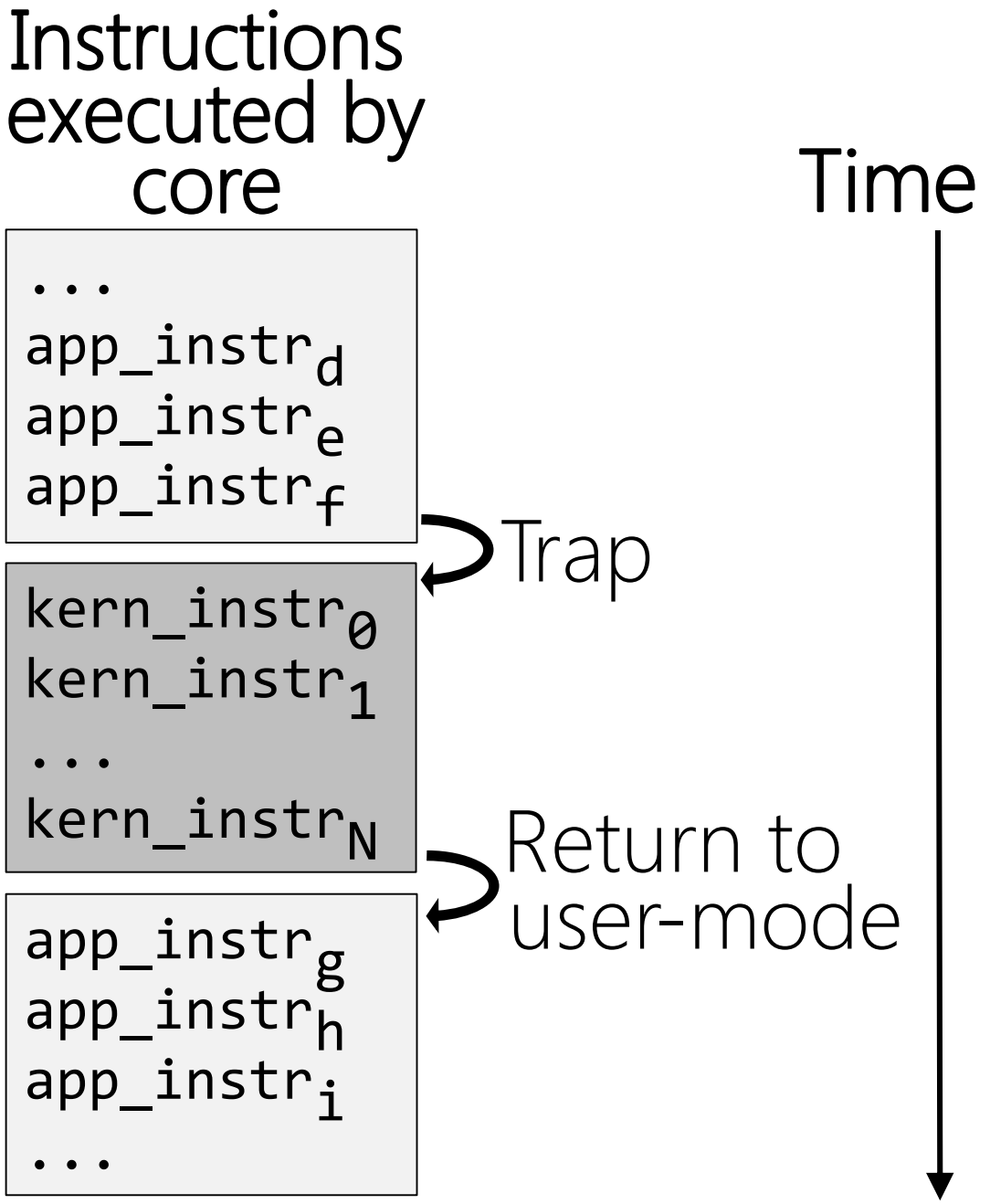
```
kern_instr0
kern_instr1
kern_instr2
...
```



Hardware

Traps: Invoking the OS

- OS code only runs in response to stimuli known as traps
  - A trap forces the processor to stop running user-level code, and start running kernel-level code
  - During a trap, the register state of the user-level application must be saved; later, when the kernel is finished, the register state of the user-level application must be restored
- Imagine that we have a single-core (i.e., single-pipeline) machine . . .



LET’S SET A TRAP

LET'S SET A TRAP

Synchronous Exceptions

Directly and immediately caused by something that a user-level program did,

- Divide-by-zero
- Null pointer dereference
- System calls (`int` instruction on x86, `syscall` instruction on MIPS)

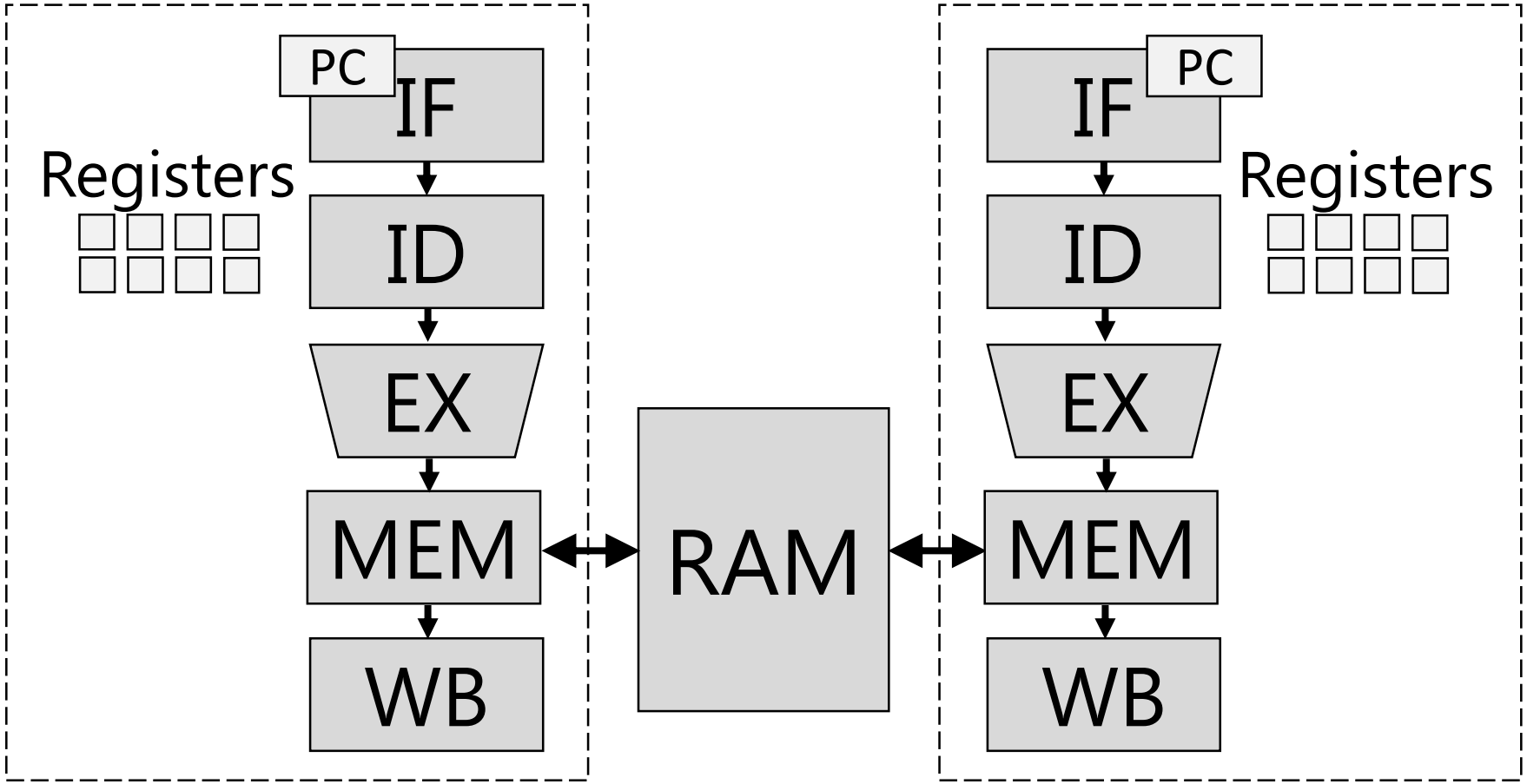
Asynchronous Interrupts

Caused by the reception of an "external" event

- Hardware timer expires
- Network packet arrives
- User generates mouse or keyboard input

Multi-core Machines

- A multi-core machine has multiple pipelines which execute instructions simultaneously
  - Each core has a separate, private set of registers
  - However, cores share the same physical RAM with the other cores
- A core can send an interrupt to another core (synchronous w.r.t. sender, but asynchronous w.r.t. receiver)
- Each core can independently disable interrupts and later reenable them

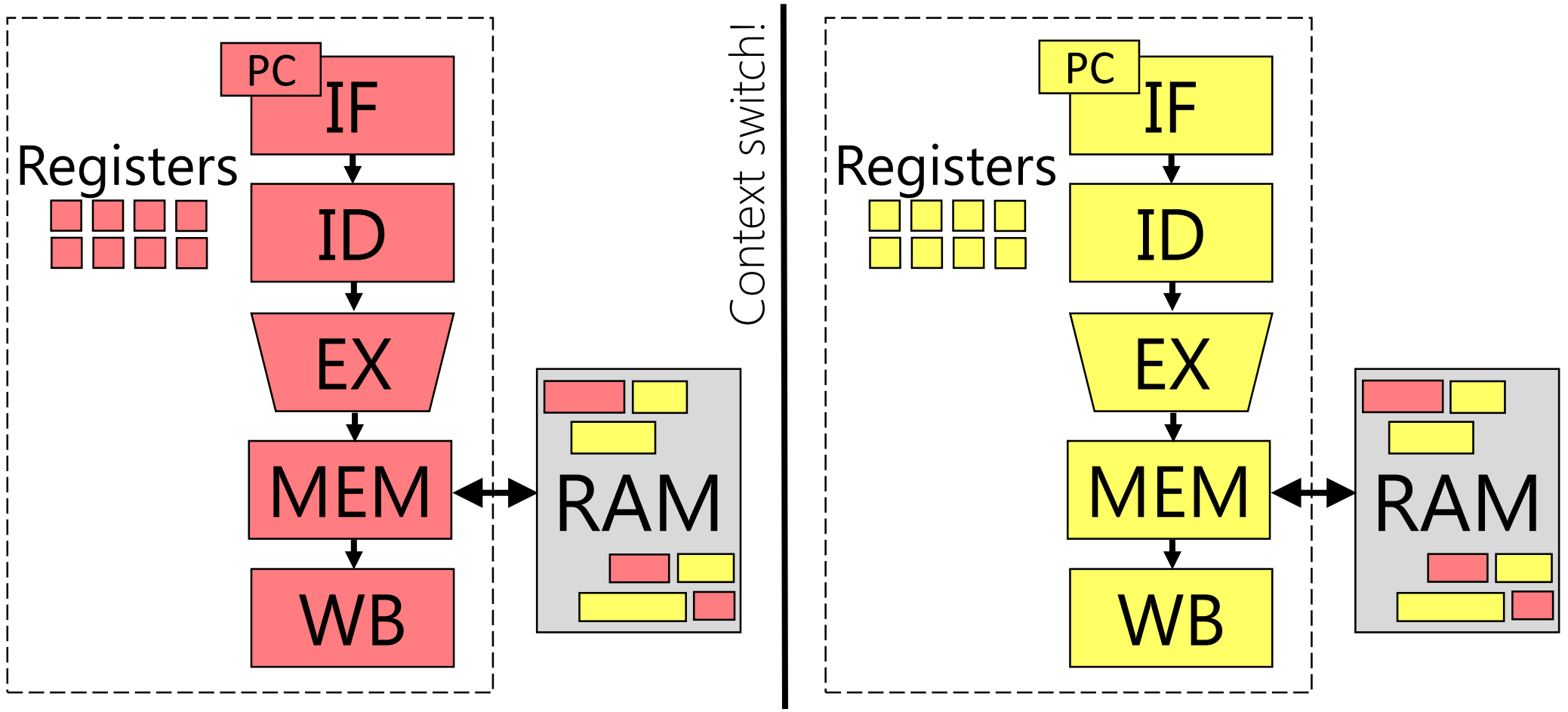


Concurrency and Synchronization

Concurrency: Doing Multiple Things At The Same Time

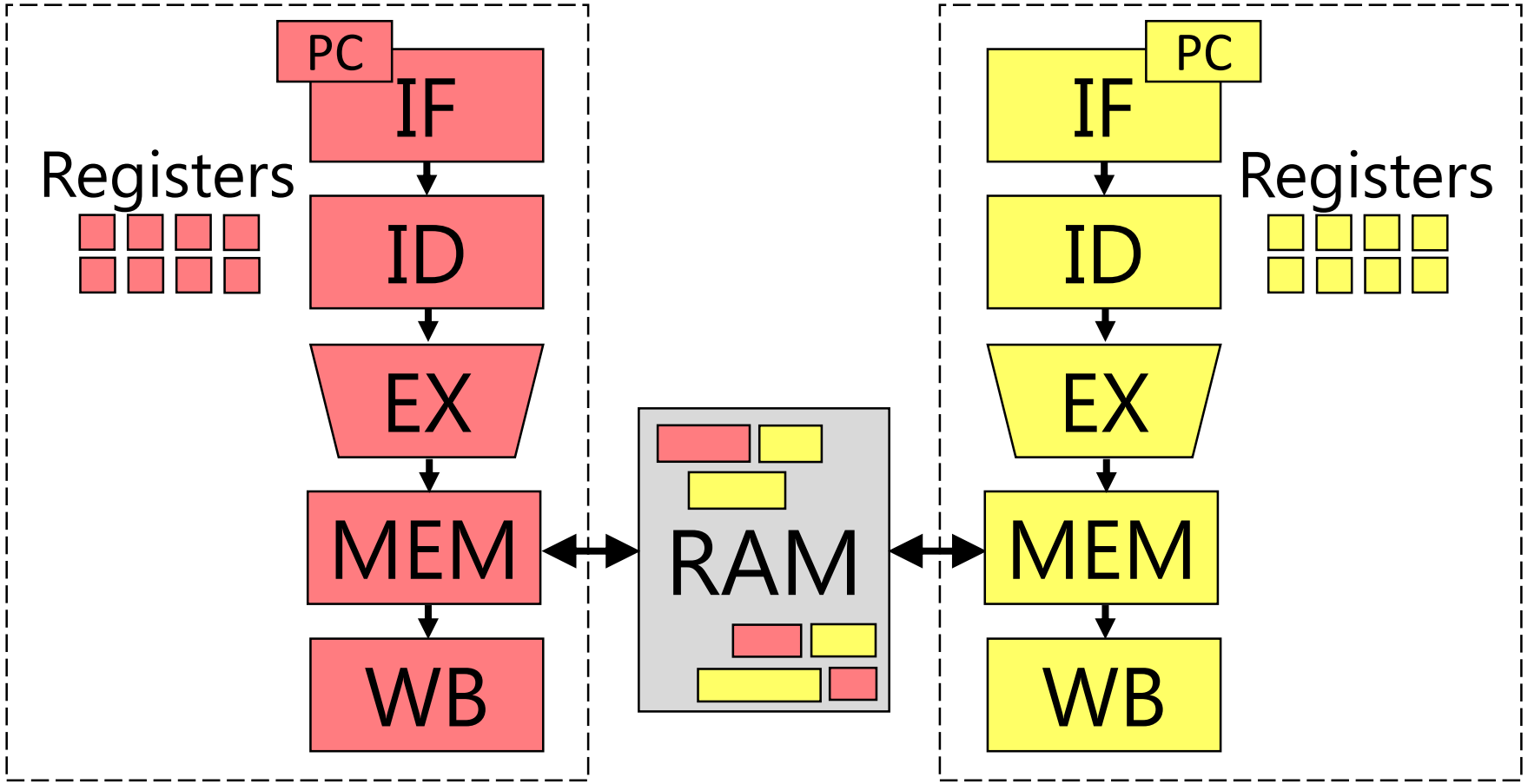
- On a single-core machine, (quasi-)concurrency arises because the OS forces different applications to share the single pipeline
  - First one application runs for a while, then another, then another . . .
  - Context switching and scheduling are tricky—we'll return to these topics later!

Suppose that there are two processes (red and yellow) . . .



Concurrency: Doing Multiple Things At The Same Time

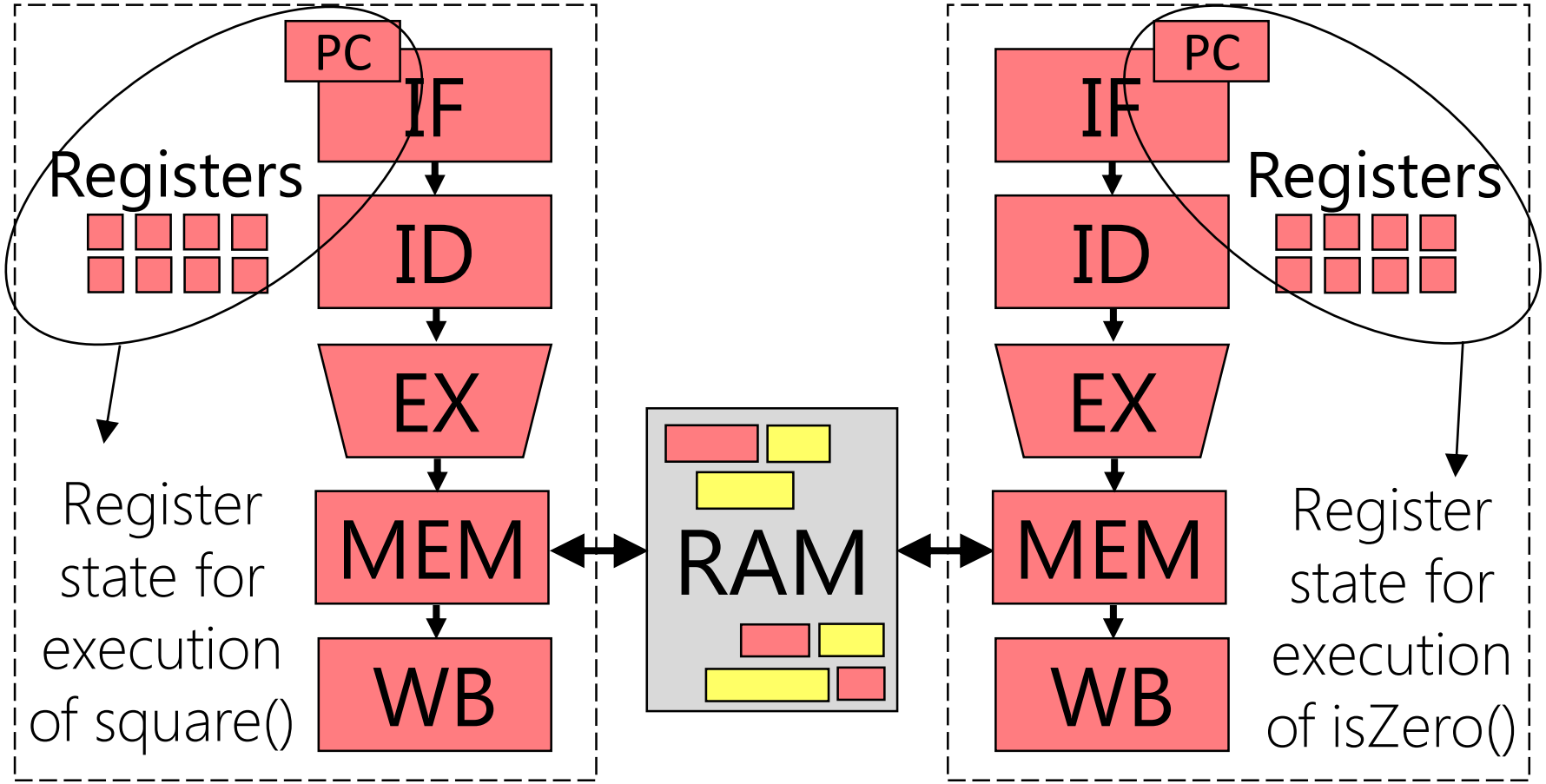
- On a multi-core machine, there is true concurrency: different pipelines are simultaneously executing independent instruction streams
- Each pipeline might be executing a stream from a different application . . .



Concurrency: Doing Multiple Things At The Same Time

- On a multi-core machine, there is true concurrency: different pipelines are simultaneously executing independent instruction streams
- . . . or some pipelines may be executing instructions from the same application, but with a different execution context (i.e., values of PC and other registers)

```
//Application code
int square(int x){
    return x*x;
}
int isZero(int x){
    return x==0;
}
```



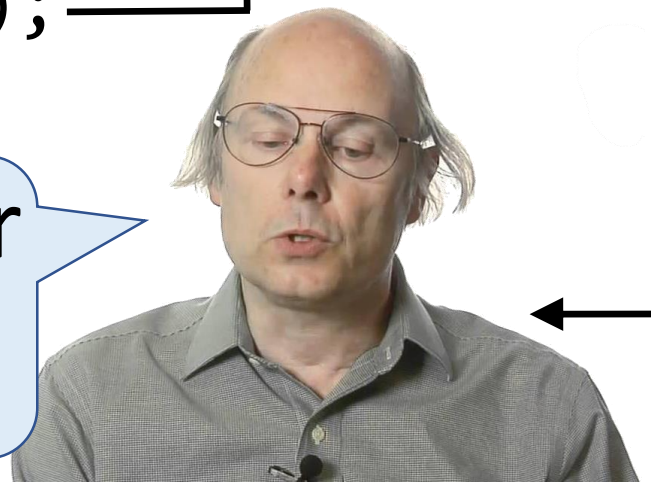
Critical Sections

- Critical section: A piece of code that accesses a resource which is shared between concurrent threads of execution
  - A critical section must be executed atomically, i.e., at any given moment, at most one thread can be manipulating the shared resource
  - If critical sections are not executed atomically, subtle bugs will occur
- Synchronization: Ensuring that critical sections are actually atomic!
  - Synchronization is important even on a uniprocessor: a thread might be taken off the processor in the middle of its critical section!
  - On a multi-core processor, you must worry about synchronization between threads on the same core, and between threads on different cores

```
std::list<int> results;

//Runs in thread one.
void square(int x){
    int s = x*x;
    results.push_back(x);
}
//Runs in thread two.
void isZero(int x){
    int iz = (x==0);
    results.push_back(iz);
}
```

STL containers are not thread-safe!



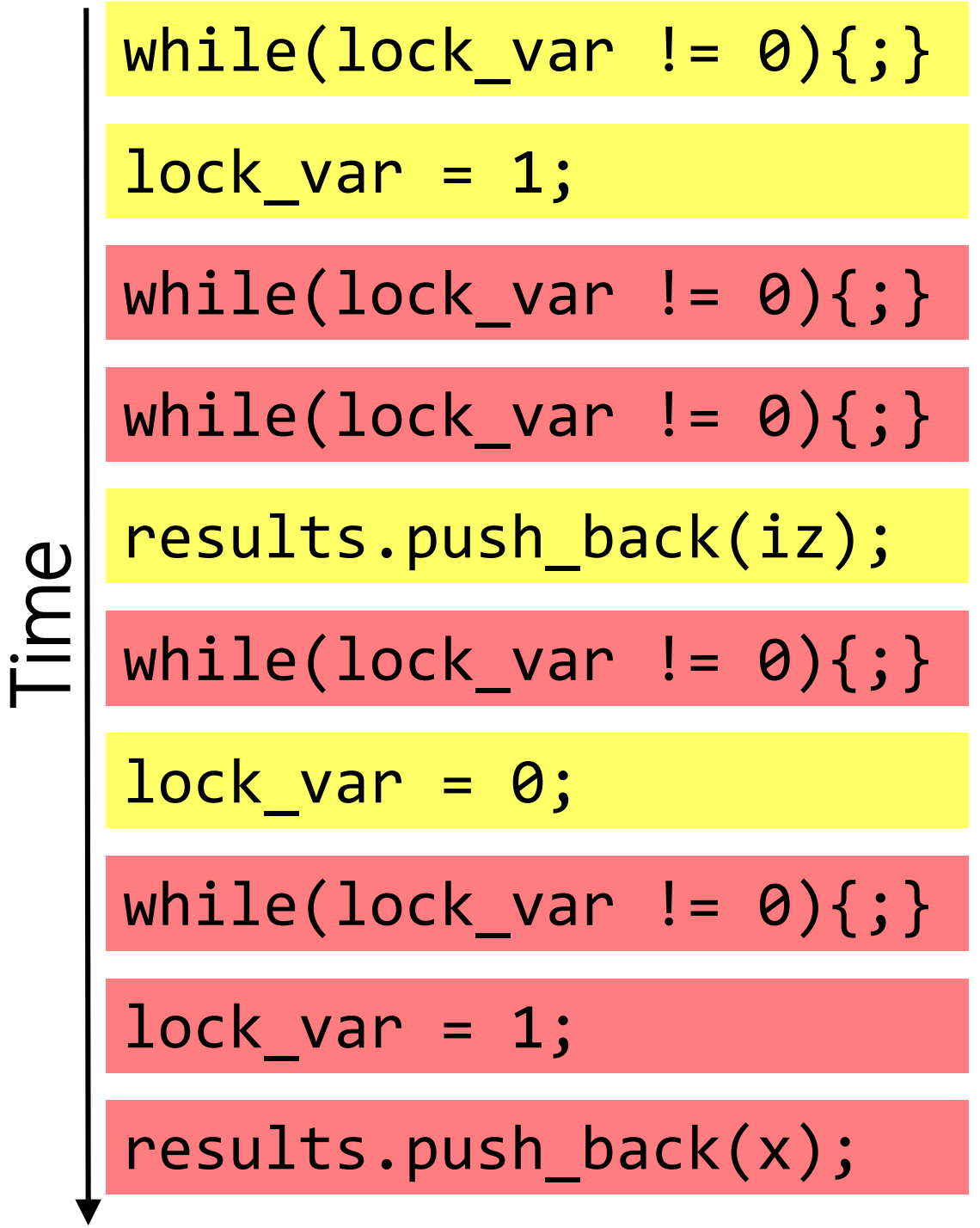
STL is optimized for speed in the non-concurrent case!

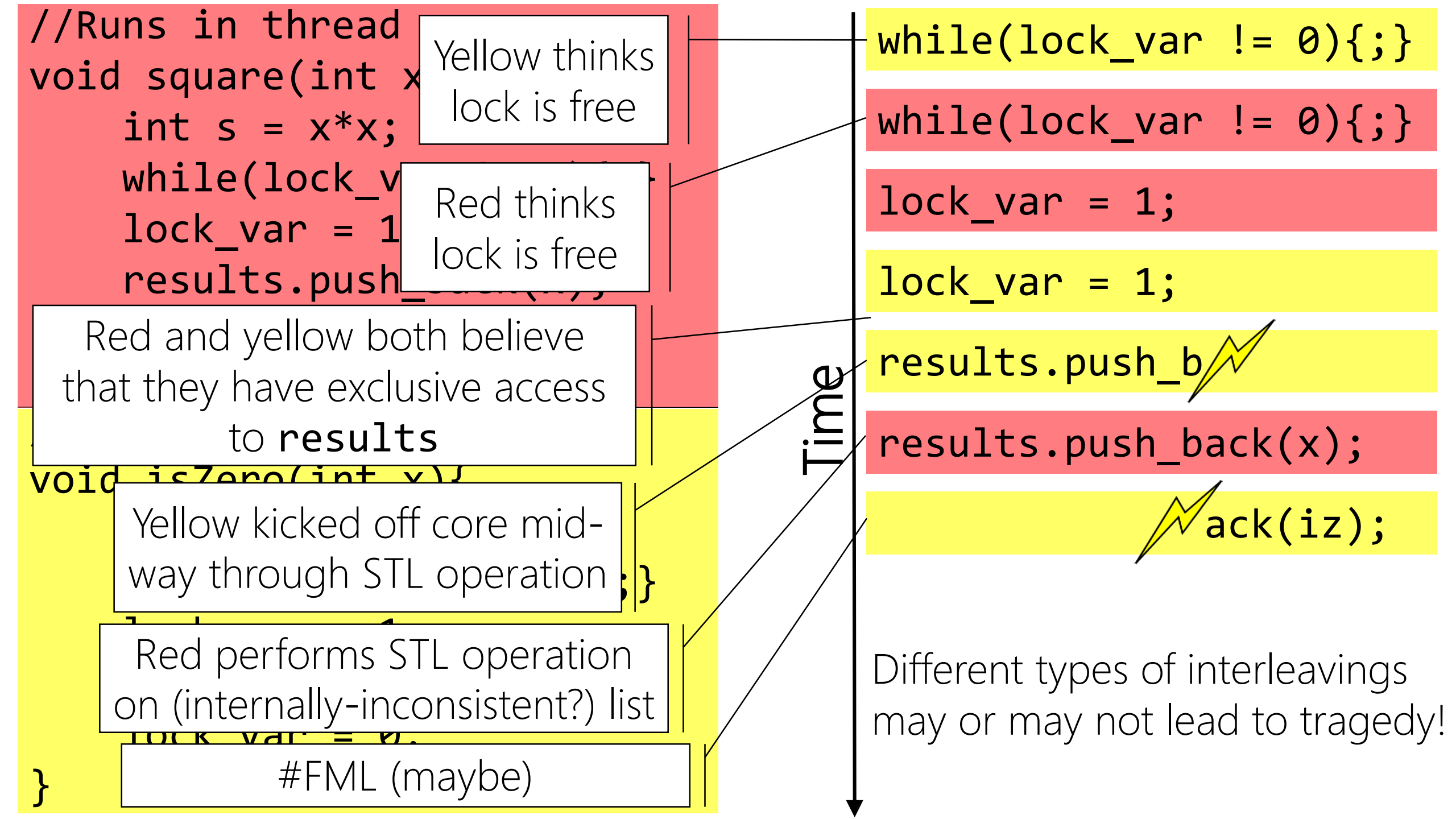
Spinlocks: A Mechanism For Protecting Critical Sections

- Spinlock: a memory location that can be in one of two states
  - Zero when spinlock is unlocked (i.e., not held by a thread)
  - One when the spinlock is locked (i.e., held by a thread)
- Here's a possible implementation:
  - Assume that a read or write to an integer is atomic (this is true on all reasonable ISAs)
  - Initialize the spinlock: `int lock_var = 0; //Unlocked`
  - Acquire the spinlock:
    ```
    while(lock_var != 0){;}
    lock_var = 1;
    ```
  - Release the spinlock: `lock_var = 0;`

```
//Runs in thread one.
void square(int x){
    int s = x*x;
    while(lock_var != 0){;}
    lock_var = 1;
    results.push_back(x);
    lock_var = 0;
}
```

```
//Runs in thread two.
void isZero(int x){
    int iz = (x==0);
    while(lock_var != 0){;}
    lock_var = 1;
    results.push_back(iz);
    lock_var = 0;
}
```



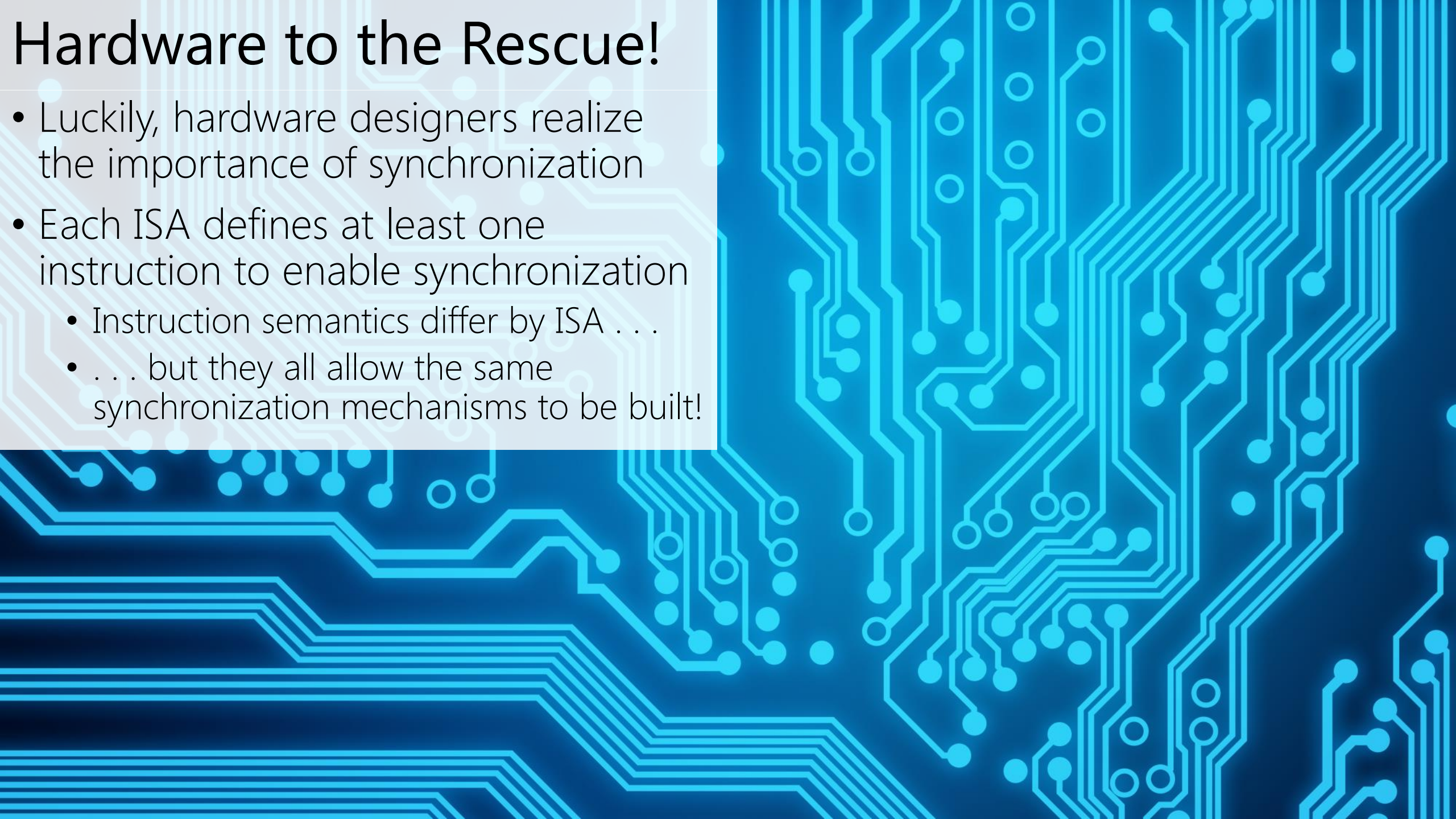

//Runs in thread
void square(int x
int s = x*x;
while(lock_v
lock_var = 1
results.push_
void isZero(int x){
lock_var = 0;
}
while(lock_var != 0){;}
while(lock_var != 0){;}
lock_var = 1;
lock_var = 1;
results.push_b
results.push_back(x);
ack(iz);
Time
Yellow thinks lock is free
Red thinks lock is free
Red and yellow both believe that they have exclusive access to results
Yellow kicked off core mid-way through STL operation
Red performs STL operation on (internally-inconsistent?) list
#FML (maybe)
Different types of interleavings may or may not lead to tragedy!

RACE CONDITIONS

YOU WILL BE DESTROYED AT A TIME AND PLACE OF CTHULU'S CHOOSING

Hardware to the Rescue!

- Luckily, hardware designers realize the importance of synchronization
- Each ISA defines at least one instruction to enable synchronization
  - Instruction semantics differ by ISA . . .
  - . . . but they all allow the same synchronization mechanisms to be built!



Hardware Primitive: Test-and-set (TAS)

- Given a memory location, TAS atomically:
  - retrieves the value of a memory location, and then
  - sets the value at that memory location to 1
- TAS is useful for building spinlocks
  - Initiatilize: `int lock_var = 0;`
  - Lock: `while(TAS(lock_var) != 0){;}`
  - Unlock: `lock_var = 0;`
- Interrupts should be disabled before the lock()->critical section-->unlock sequence, and then reenabled (why?)

Hardware Primitive: Load Link/Store Conditional (LL/SC)

- This synchronization primitive consists of two paired instructions
  - **`ll rt, offset(rs):`** Loads a value from memory into **`rt`**
  - **`sc rt, offset(rs):`** Stores value in **`rt`** back to the memory location ONLY if the location has not changed since the associated **`ll`** instruction executed; **`rt`** is set to 1 if the store succeeded, 0 otherwise
- When used as a pair, the instructions are used to build an atomic read-write that either succeeds or fails
- MIPS supports LL/SC; to see an example, look at OS 161's **`kern/arch/mips/include/spinlock.h`**

```
spinlock_data_t
spinlock_data_testandset(volatile spinlock_data_t *sd)
{
    spinlock_data_t x;
    spinlock_data_t y;

    /*
     * Test-and-set using LL/SC.
     *
     * Load the existing value into X, and use Y to store 1.
     * After the SC, Y contains 1 if the store succeeded,
     * 0 if it failed.
     *
     * On failure, return 1 to pretend that the spinlock
     * was already held.
     */
    y = 1;
    __asm volatile(
        ".set push;"        /* save assembler mode */
        ".set mips32;"      /* allow MIPS32 instructions */
        ".set volatile;"    /* avoid unwanted optimization */
        "ll %0, 0(%2);"     /*   x = *sd */
        "sc %1, 0(%2);"     /*   *sd = y; y = success? */
        ".set pop"      /* restore assembler mode */
        : "=&r" (x), "+r" (y) : "r" (sd));
    if (y == 0) {
        return 1;
    }
    return x;
}
```