Part I Introduction

Hardware and OS Review

The scientist described what is: the engineer creates what never was.

Theodor von Karman
The father of supersonic flight

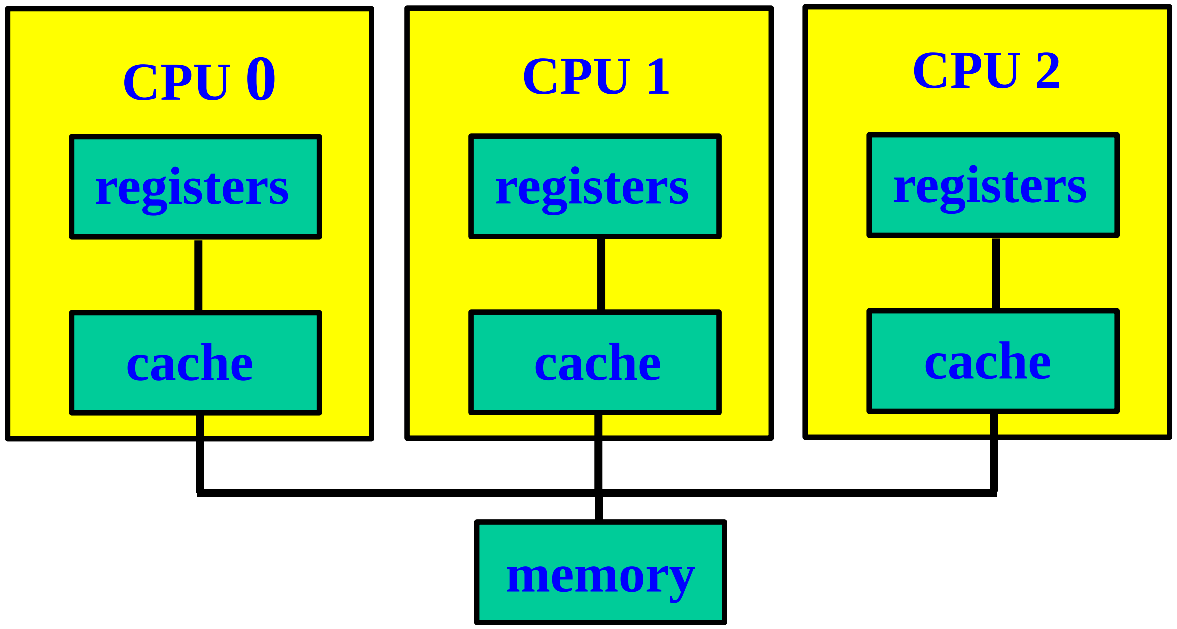
Spring 2021

Multiprocessor Systems

- **Multiprocessor systems, aka parallel systems or tightly coupled systems, have more than one CPUs.**
- **Advantages:**
  - **Increased throughput: gets more jobs done**
  - **Economy of scale: Because of resource sharing, multiprocessor systems are cheaper than multiple single processor systems.**
  - **Increased reliability: the failure of one processor will not halt the whole system.**

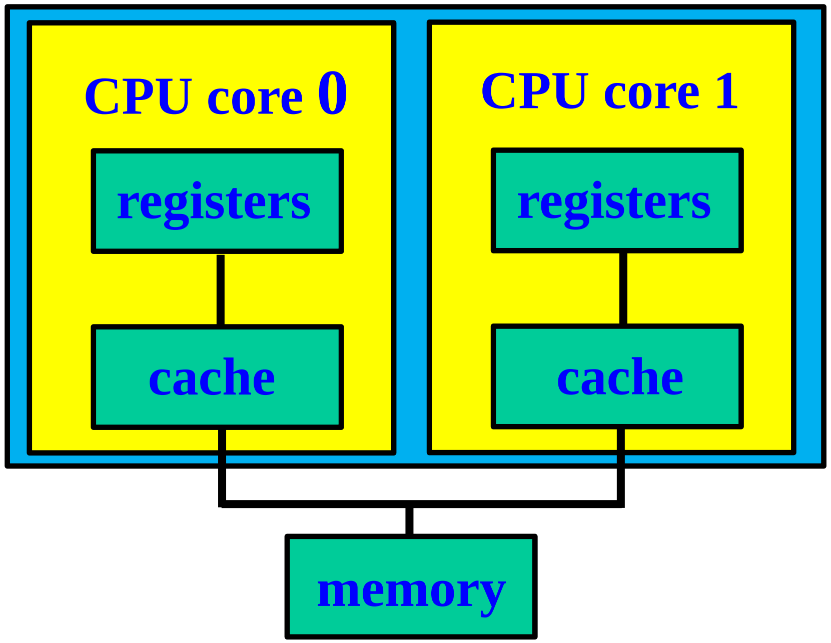
Symmetric Multiprocessing (SMP)

- **Each processor performs all tasks under the control of the same OS.**
- **All processors are peers; no special (e.g., master-slave) relationship exists.**



Multicore CPUs

- This is multiprocessor on a single chip.
- They are more efficient since communications among cores are faster than among CPUs.
- They also consume less power.



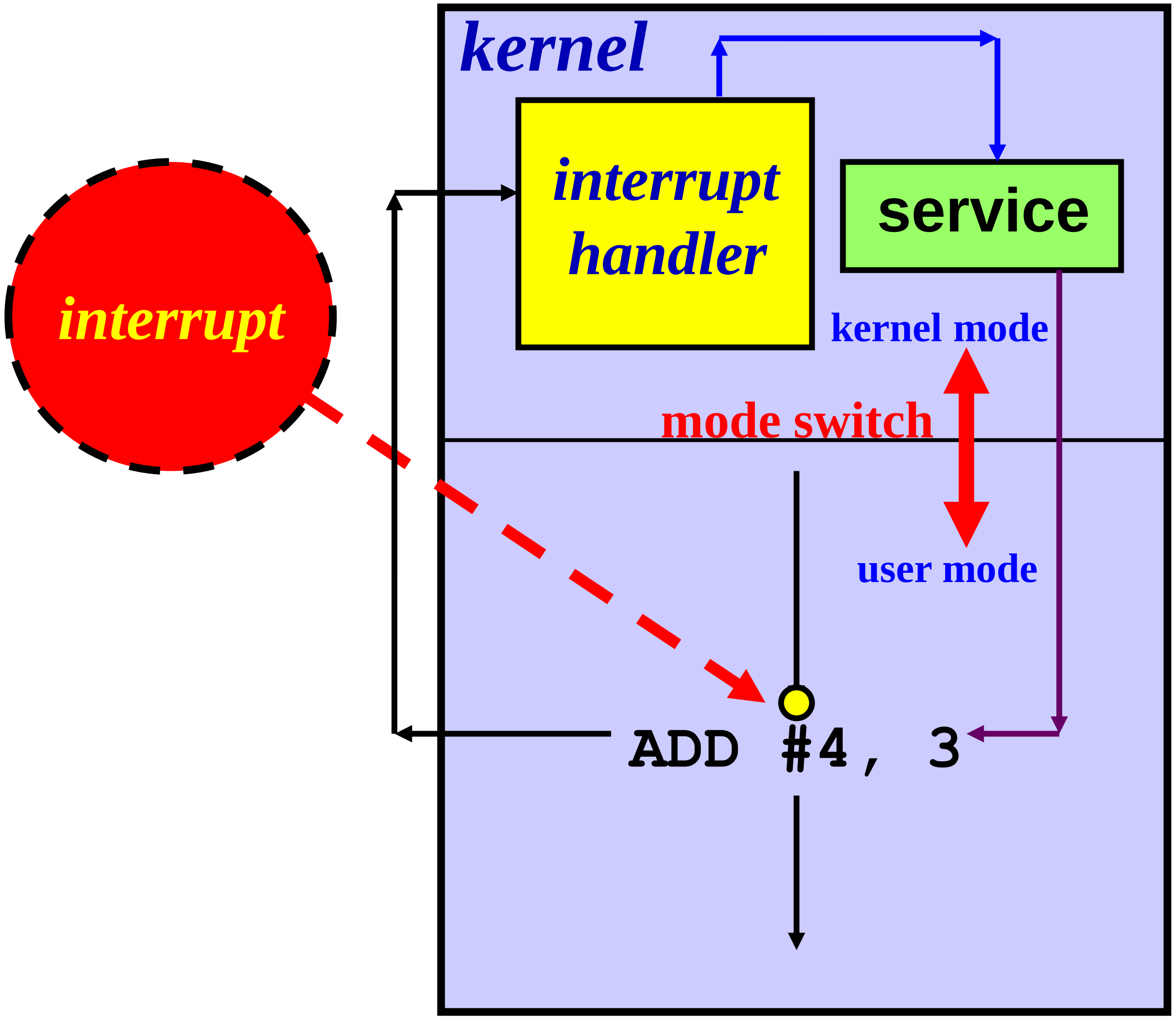
Dual-Mode Operation

- **Modern CPUs have two execution modes: the *user* mode and the *supervisor* (or system, kernel, privileged) mode, controlled by a mode bit.**
- **The OS runs in the supervisor mode and all user programs run in the user mode.**
- **Some instructions that may do harm to the OS (e.g., I/O and CPU mode change) are *privileged instructions*. Privileged instructions, for most cases, can only be used in the supervisor model.**
- **When execution switches to the OS (resp., a user program), execution mode is changed to the supervisor (resp., user) mode.**

Interrupt and Trap

- **An event that requires the attention of the OS is an interrupt. These events include the completion of an I/O, a keypress, a request for service, a division by zero and so on.**
- **Interrupts may be generated by hardware or software.**
- **An interrupt generated by software (*i.e.*, division by 0) is referred to as a *trap*.**
- **Modern operating systems are *interrupt driven*, meaning the OS is in action only if an interrupt occurs.**

What Is Interrupt-Driven?

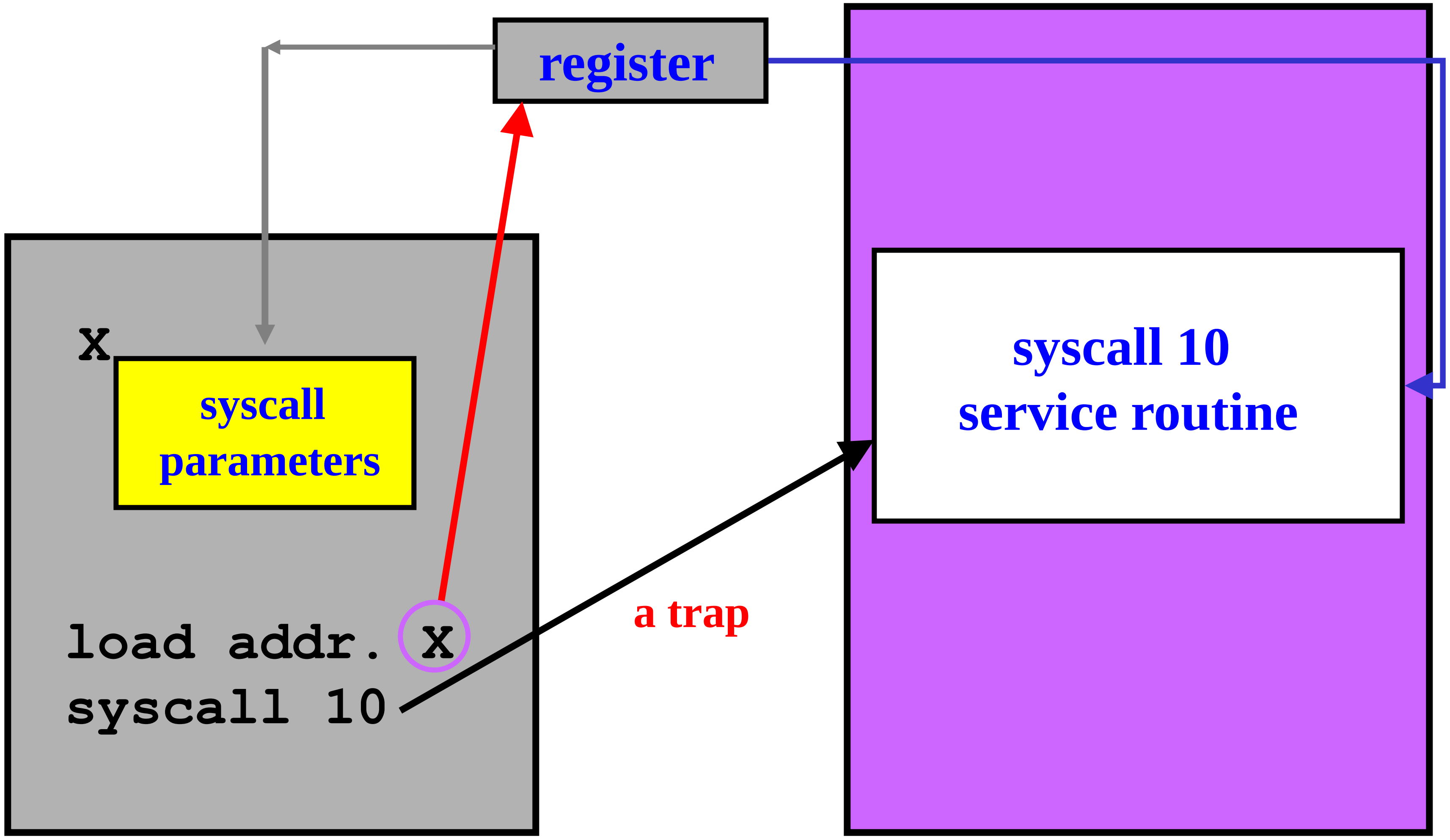


- The OS is activated by an interrupt.
- The executing program is suspended.
- Control is transferred to the OS.
- A program will be resumed when the service completes.

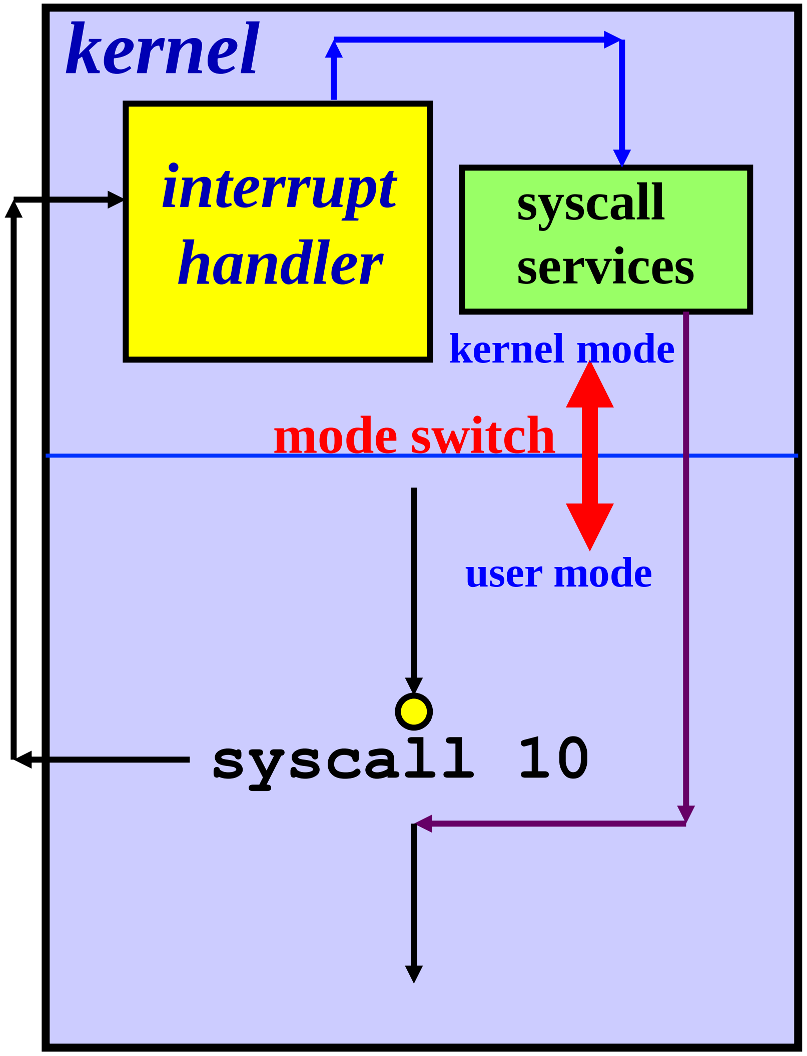
System Calls

- System calls provide an interface to the services made available by an operating system.
- A system call generates an interrupt (actually a trap), and the caller is suspended.
- Type of system calls:
  - Process control (e.g., create and destroy processes)
  - File management (e.g., open and close files)
  - Device management (e.g., read and write operations)
  - Information maintenance (e.g., get time or date)
  - Communication (e.g., send and receive messages)

System Call Mechanism: 1/2



System Call Mechanism: 2/2



- A system call generates a *trap*.
- The executing program (i.e., caller) is suspended.
- Control is transferred to the OS.
- A program will be resumed when the system call service completes.

Timer

- **Because the operating system must maintain the control over the CPU, it has to prevent a user program from getting the CPU forever without calling for system service (i.e., I/O).**
- **Use an interval timer! An interval timer is a count-down timer.**
- **Before a user program runs, the OS sets the interval timer to certain value. Once the interval timer counts down to 0, an interrupt is generated and the OS can take appropriate action.**

Fetch-Decode-Execute: 1/13

- The execution of a machine instruction in a CPU usually involves a couple of stages: **Fetch**, **Decode** and **Execute**.
- We could make this 3-stage finer as follows:
  - **Fetch**: loading the next instruction into the CPU
  - **Decode**: analyzing the fetched instruction and determining the operation code and operands.
  - **Load Operands**: loading the contents of operand from memory and registers into the CPU
  - **Execute**: executing the needed operation to obtain the result
  - **Save**: Saving the computed result to register and memory.

Fetch-Decode-Execute: 2/13

- Consider a hypothetical instruction as follows:

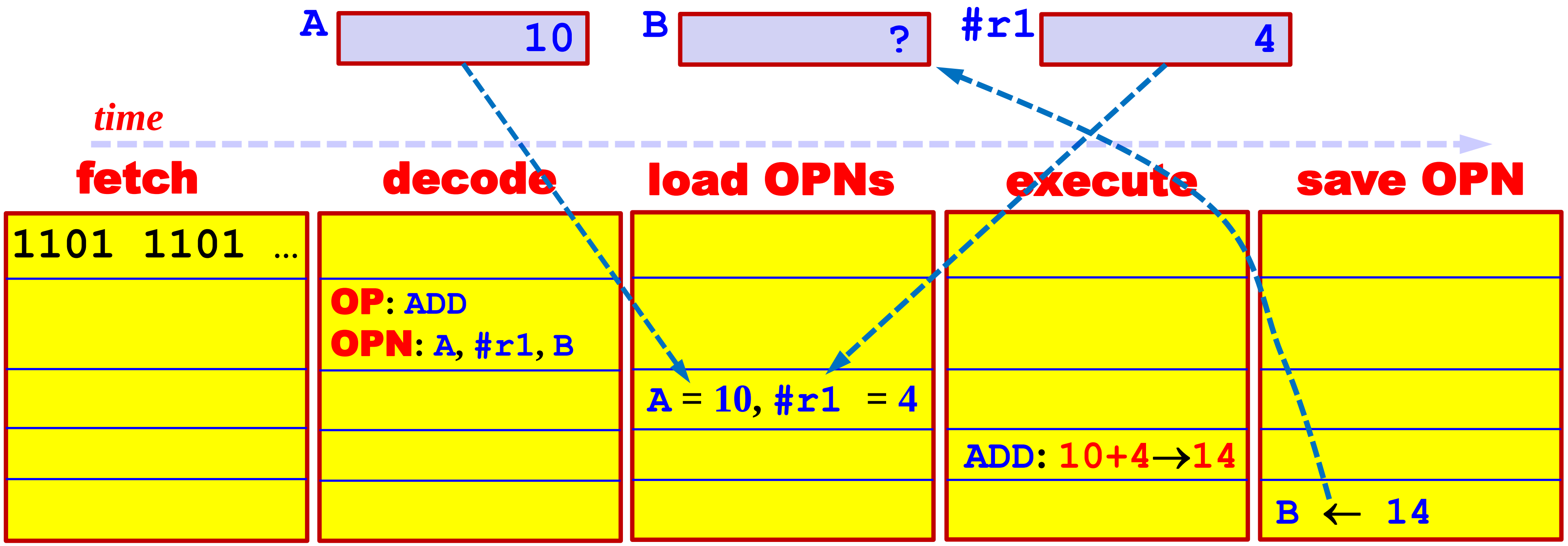
```
ADD  A, #r1, B
```

- This instruction adds the content in memory location `A` and the content in register 1 `#r1`, and saves the result to memory location `B`.
- Let this instruction in bit pattern be

```
1101 1101 … …
```

Fetch-Decode-Execute: 3/13

- **Here is a sequence of operations showing how an instruction is executed in the CPU.**



at anytime, only one stage is busy and other four stages are idle

why don’t we fetch the next instruction after this instruction moves to decode?

Fetch-Decode-Execute: 4/13

- Now consider the following five instructions:

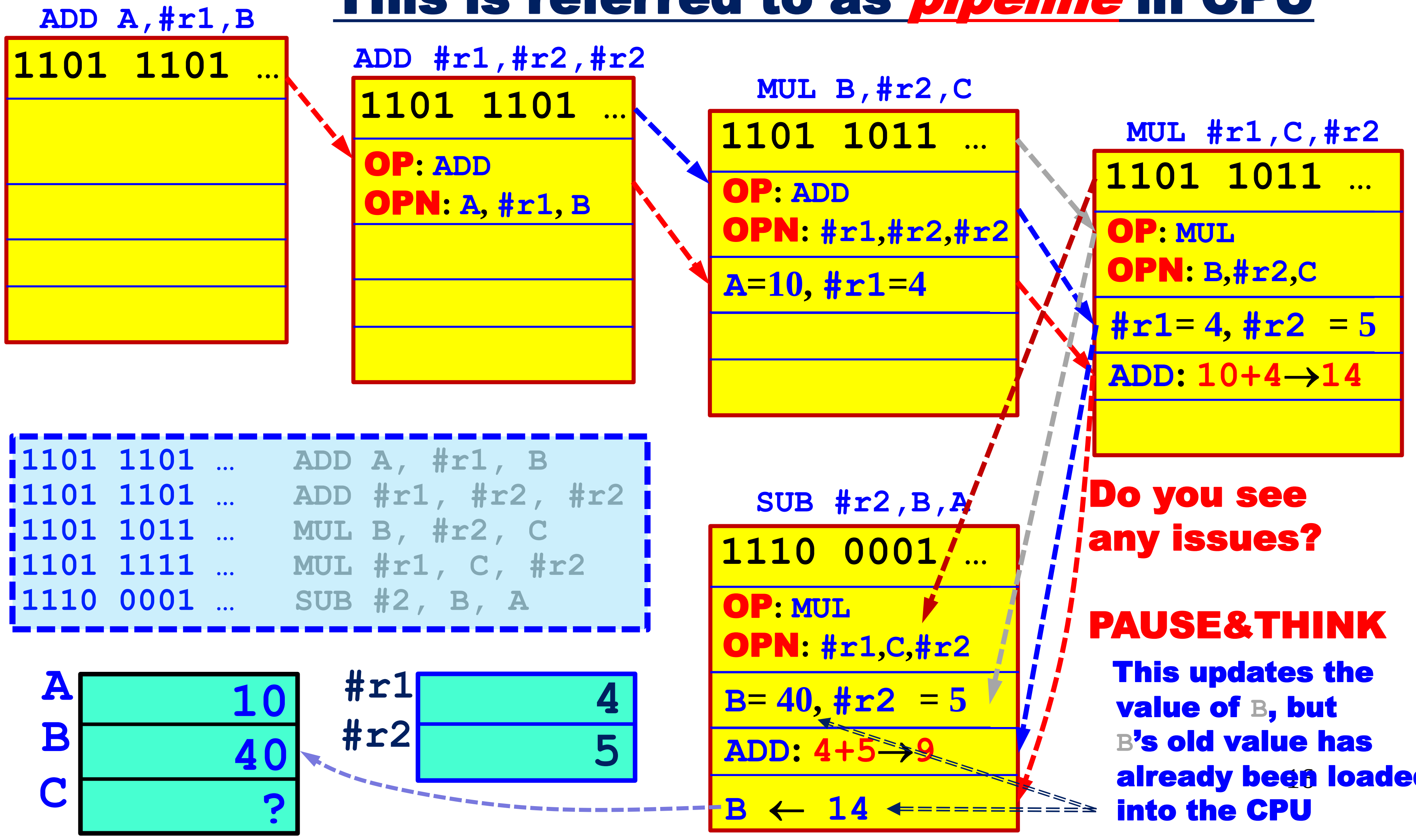
```
1101 1101 …    ADD A, #r1, B
1101 1101 …    ADD #r1, #r2, #r2
1101 1011 …    MUL B, #r2, C
1101 1111 …    MUL #r1, C, #r2
1110 0001 …    SUB #r2, B, A
```

- Each instruction uses the first two operands and saves the result to the third.

	A	B	C	#r1	#r2
initial	10	40	?	4	5
ADD_1	10	**14**	?	4	5
ADD_2	10	14	?	4	**9**
MUL_1	10	14	**126**	4	9
MUL_2	10	14	126	4	**504**
SUB	**490**	14	126	4	504

Fetch-Decode-Execute: 5/13

This is referred to as *pipeline* in CPU



```
1101 1101 …    ADD A, #r1, B
1101 1101 …    ADD #r1, #r2, #r2
1101 1011 …    MUL B, #r2, C
1101 1111 …    MUL #r1, C, #r2
1110 0001 …    SUB #2, B, A
```

Do you see any issues?

PAUSE&THINK

This updates the value of B, but B's old value has already been loaded into the CPU

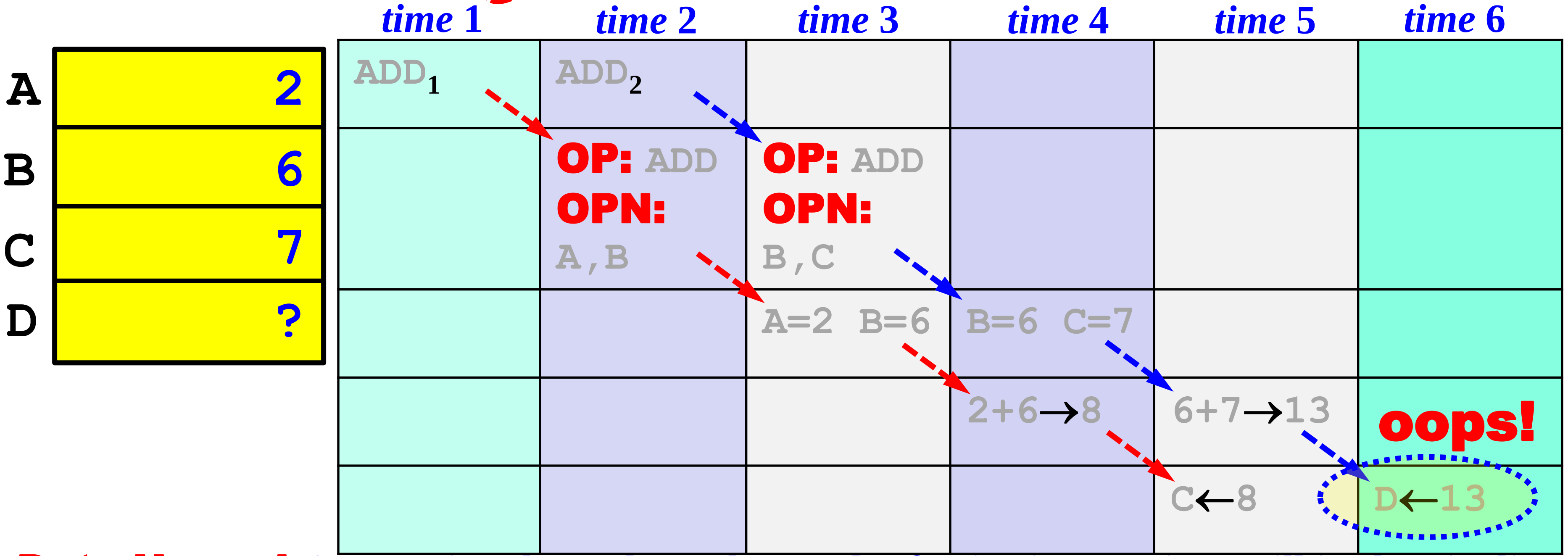
Fetch-Decode-Execute: 6/13

- Now consider the following two instructions. We expect `D = 14`.

```
ADD    A,B,C
ADD    B,C,D
```

the result of the first instruction is used by the second instruction

isn't it like concurrent sharing?



Data Hazard: instruction depends on the result of prior instructions still in the pipeline

`C` used the second instruction has already been fetched before it receives the new value

Fetch-Decode-Execute: 7/13

- Some instructions are important to concurrency.
- The `Compare-and-Swap` (`CS`) instruction is a good example.

this is a single machine instruction

```
int CS(int *p, old, new)
{
   if (*p != old)
      return FALSE;
   *p = new;
   return TRUE;
}
```

If `*p` is the same as the `old` value, updates `*p` with `new` and returns `TRUE`.
Otherwise, returns `FALSE` without update.

```
done = FALSE;
while (!done) {
   value = *p;
   done = CS(p, value, value+x);
}
```

Due to interleaved execution, this process may be suspended after executing `value = *p`.
The other process could modify `*p` and by the time `CS` is being executed `*p` and `value` may be different!
Only if `*p` and `value` are the same, `*p` is updated with `value+x`.

Fetch-Decode-Execute: 8/13

- Suppose `*p = 10`
- Consider processes P_1 and P_2.

Process P_1	Process P_2
`int x;`	`int y;`
…	…
1 `x = *p;`	**3** `y = *p;`
2 `… = CS(p,x,x+1);`	**4** `… = CS(p,y,y+2);`
…	…

- Let P_1 and P_2 be run on a single CPU without pipeline.
- Two possible interleaving executions are possible:
  - **1-2-3-4**: one process completes before the other starts
  - **1-3-2-4**: A `CS` is executing between the two statements of the other process (i.e., interleaved).

Fetch-Decode-Execute: 9/13

- A process completes before the other starts.

Process P_1	`x`	Process P_2	`y`	`*p`
				10
`x = *p`	10			10
`CS(p,x,x+1)`	10			11
		`y = *p`	11	11
		`CS(p,y,y+2)`	13	13

this is the correct result

`*p` is updated sequentially

Fetch-Decode-Execute: 10/13

- A `CS` is run between the `*p` assignment and the `CS` of the other process (i.e., interleaved).

	Process P_1	`x`	Process P_2	`y`	`*p`
1					10
2	`x = *p`	10			10
3		10	`y = *p`	10	10
4	`CS(p,x,x+1)`	10		10	11
5		10	`CS(p,y,y+2)`	10	11

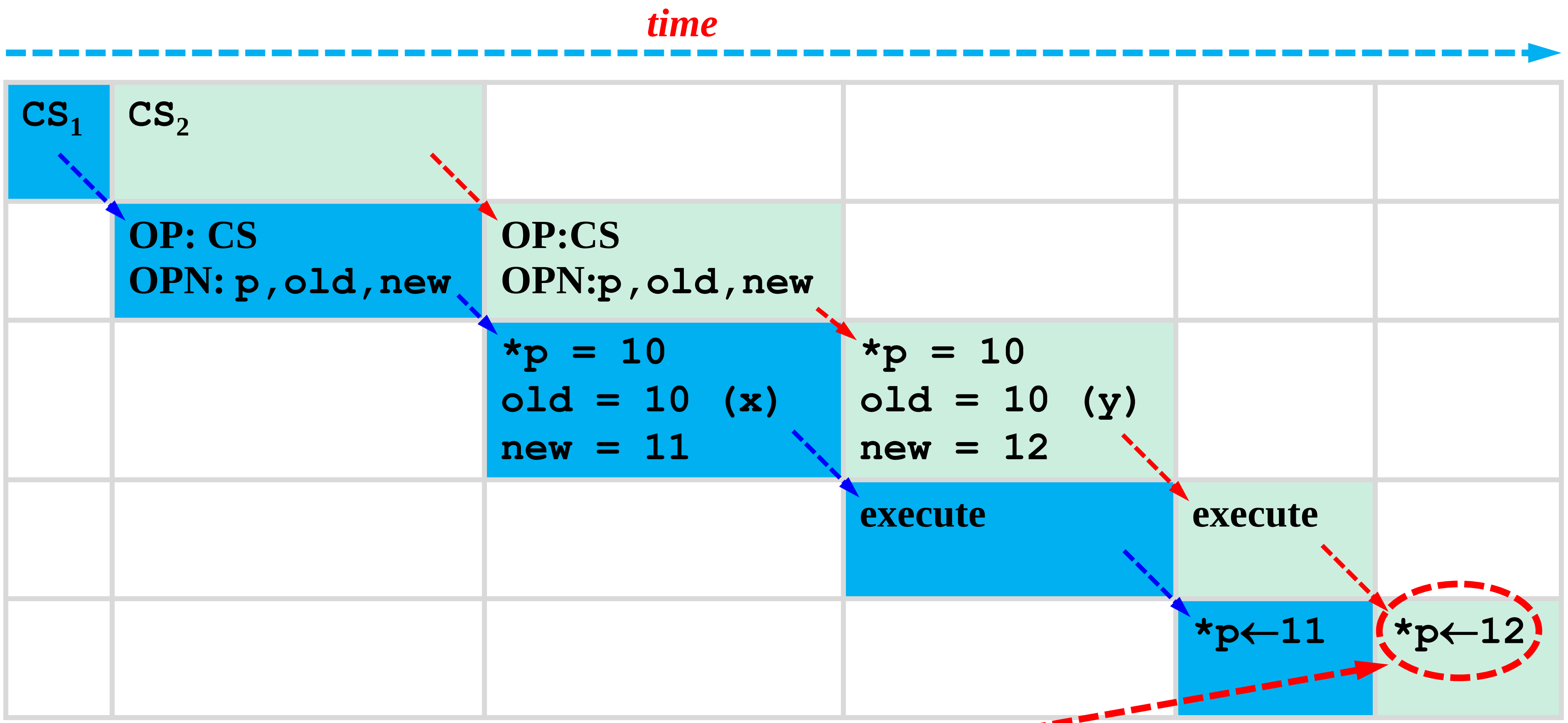
Process P_2 fails to update in this iteration, because `*p` and `y` are not the same.

updated by P_1

P_2 must loop back to try its update again.
But, if the above pattern continues, P_2 may have a less chance to update

Fetch-Decode-Execute: 11/13

- If the two processes run on a CPU **with pipeline**?



this does not seem to be correct!
because it uses the old `*p = 10` **rather than the updated** `*p = 11`

Fetch-Decode-Execute: 12/13

- What if both `CS` instructions execute in different CPU/core?

CPU 1	CPU 2
CS1	CS2
OP: CS **OPN:** `p,old,new`	**OP:** CS **OPN**: `p,old,new`

`*p = 10`
`old = 10`
`new = 11`
	`*p = 10`
	`old = 10`
	`new = 12`

execute	execute
`*p = 11`	`*p = 12`

What is the result?
11 or 12?

It depends on which CPU runs faster.
If CPU 1 is faster, the result is 12.
Otherwise, the result is 11.

Fetch-Decode-Execute: 13/13

- To overcome these problems, we need a new type of machine instructions to avoid these issues.
- These are the **atomic** instructions.
- These atomic instructions run essentially the same as a CPU without the pipeline feature.
- The "definition" of atomic instruction is on the next slide.

Special Machine Instructions: 1/3

- **Atomic Instructions** execute without interleaving and cannot be split by other instructions. When an atomic instruction is recognized by the CPU,
  - All other instructions being executed in various stages by the CPUs are suspended (and perhaps re-issued later) until this instruction finishes.
  - They cannot be interrupted
- If two atomic instructions are issued at the same time, even on different CPUs or cores, they will be executed sequentially.

Special Machine Instructions: 2/3

- **An atomic instruction means: Do this operation alone and don’t get interrupted while doing this.**
- **When the CPU recognizes an atomic instruction, it is the only instruction running in the CPU and all other instructions are stopped. Interrupts are not allowed to happen!**
- **When this atomic instruction completes, other instructions and operands may have to be re-issued and re-fetched.**
- **You will learn more in the *Computer Architectures* class. We only provide an intuitive discussion that is good enough for this course.**

Special Machine Instructions: 3/3

- **We will discuss atomic instructions (e.g., `TS`: Test-and-Set) for synchronization later. The previously mentioned Compare-and-Swap (`CS`) instruction is another commonly seen example.**
- **Without atomic instructions, race conditions could occur when instructions modify a shared memory location. Race conditions will be discussed in the next unit.**
- **Privileged Instructions: These instructions, in general, can only execute in the supervisor or kernel mode.**

The End