CS 110
Computer Architecture

An Introduction to *Operating Systems*

Instructors:
Chundong Wang & Siting Liu

https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C

CA so far…

C Programs

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

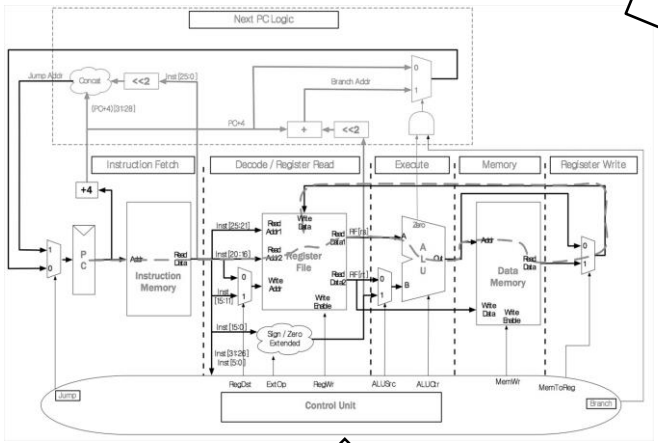
RISC-V Assembly

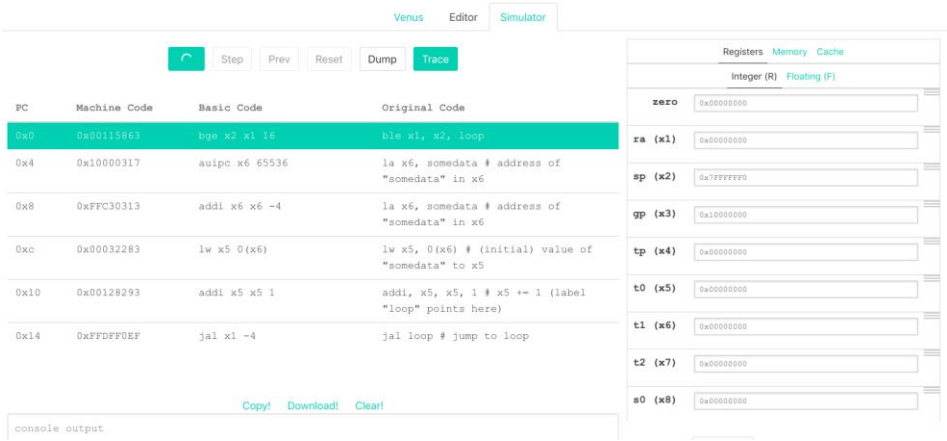
```
.foo
lw   t0, 4(s1)
addi t1, t0, 3
beq  t1, t2, foo
nop
```

Project 2

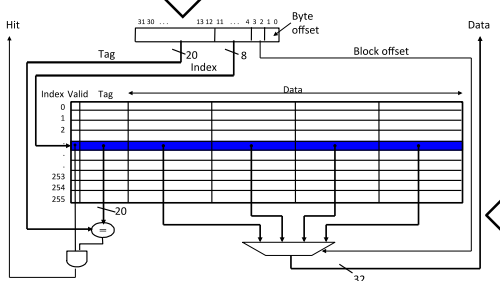
Project 1

CPU

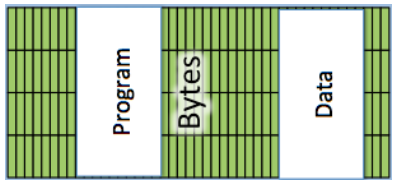




Caches



Memory



So how is this any different?

Adding I/O

C Programs

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

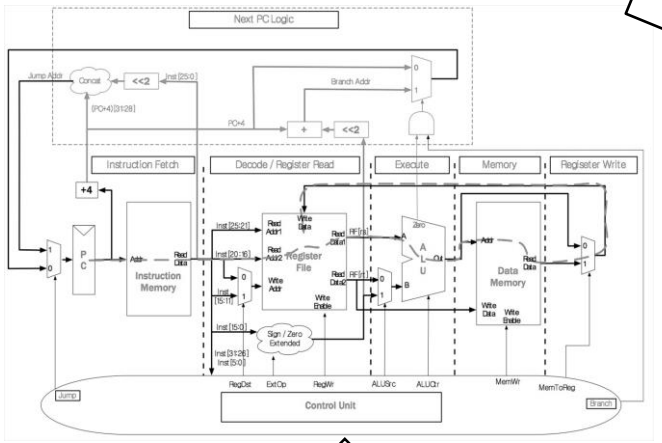
RISC-V Assembly

```
.foo
lw   t0, 4(s1)
addi t1, t0, 3
beq  t1, t2, foo
nop
```

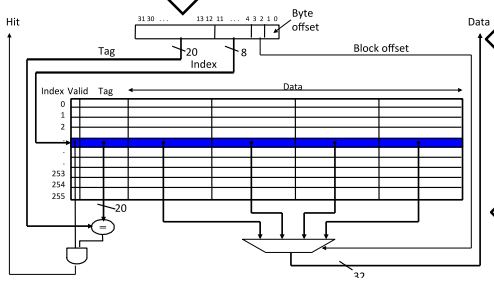
Project 1

Project 2

CPU



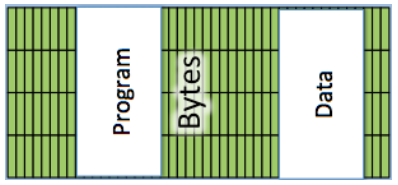
Caches



Screen | Keyboard | Storage

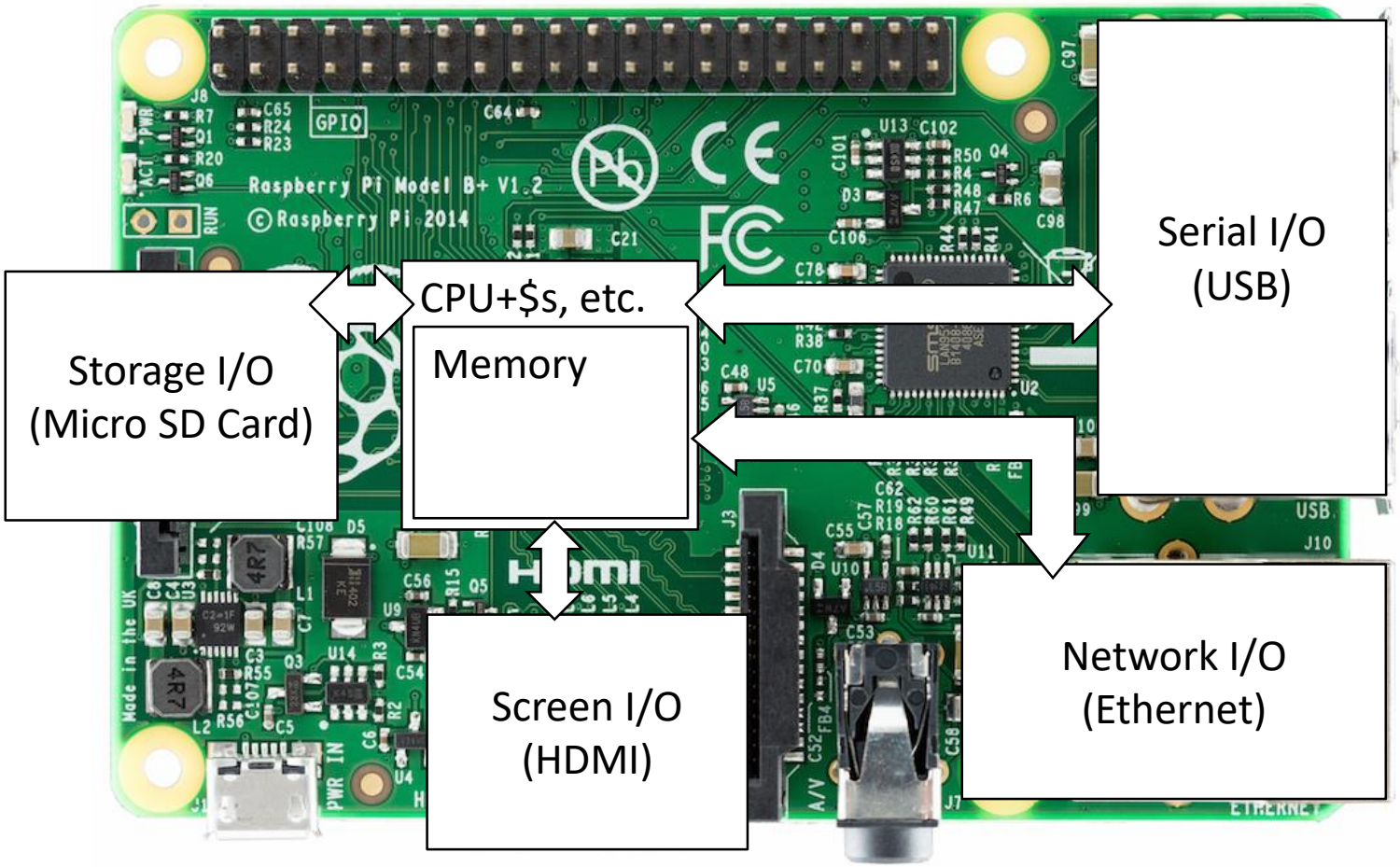
I/O (Input/Output)

Memory



Raspberry Pi

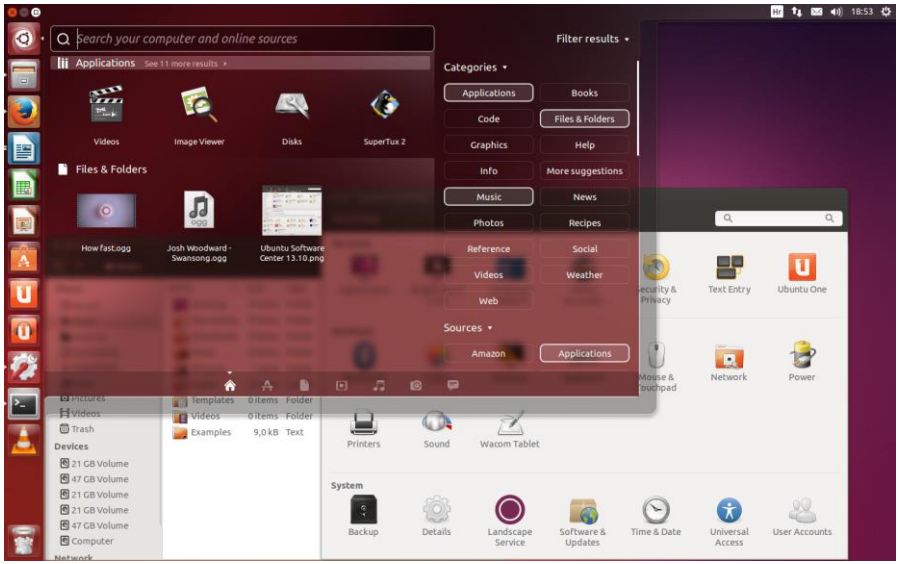
ARM架构



It’s a real computer!

But wait…

- That's not the same! Our CS 110 experience isn't like the real world. When we run VENUS, it only executes one program and then stops.
- When I switch on my computer, I get this:



Yes, but that's just software! The Operating System (OS)

最基本的基石出软件

Well, "just software"

OS: the biggest piece of software on the machine

- The biggest pi r machine?
- How many lin uesstimates:

Year		ze of zipped file
1994		1MB
1996		6MB
2001		23MB
2003		40MB
20		92MB
2015		118MB
Apr 2020		166MB
May 2021		179MB
May 2022		189MB
Apr 2023	linux-6.3.tar.gz	209MB

纯文本

6MB

unzip 1.4GB

What does the OS do?

- One of the first things that runs when your computer starts (right after firmware/ bootloader)
- Loads, runs and manages programs:
  - Multiple programs at the same time (time-sharing)
  - Isolate programs from each other (isolation)
  - Multiplex resources between applications (e.g., devices)
- Services: File System, Network stack, printer, etc.
- Finds and controls all the devices in the machine in a general way (using “device drivers”)

What does the core of OS need to do?

- Provide **interaction** with the outside world
  - Interact with “devices”
    - Disk, screen, keyboard, mouse, network, etc.
- Provide **isolation** between running programs (processes)
  - Each program runs in its own little world
    - Virtual memory

Agenda

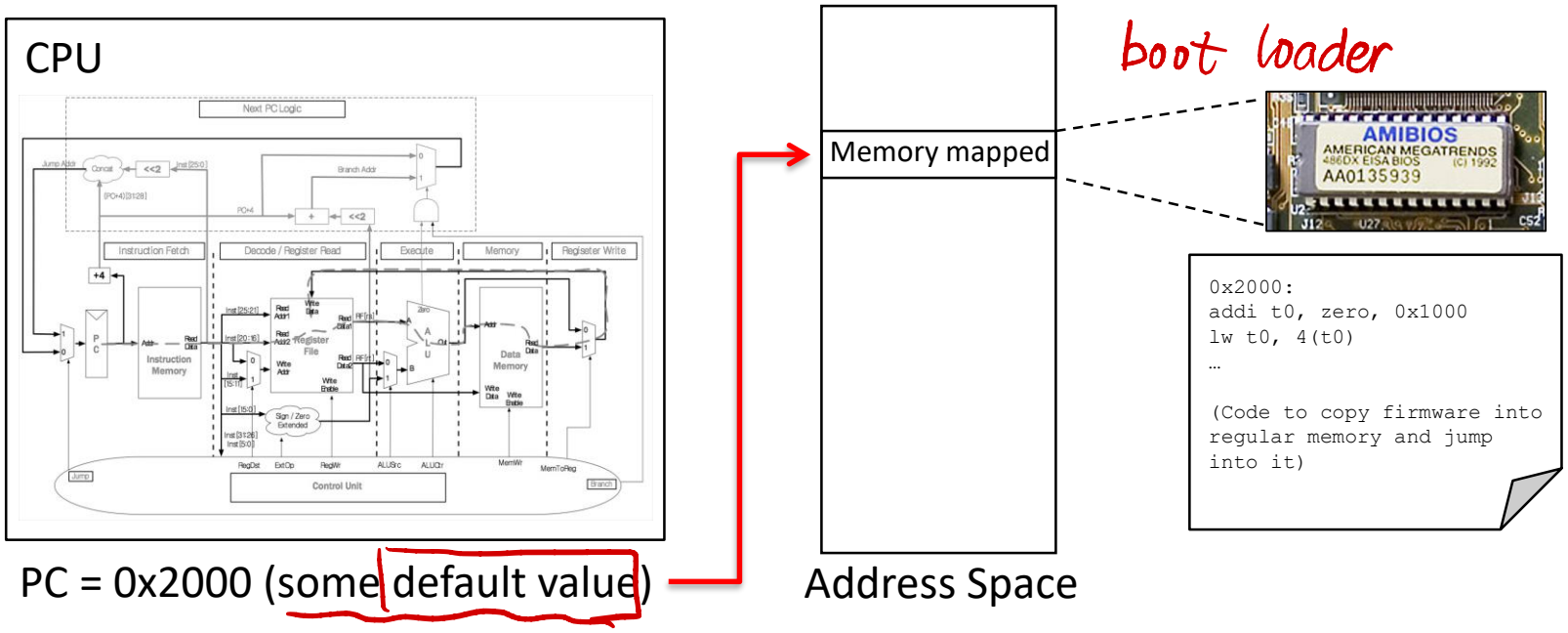
- OS Boot Sequence and Operation
- Devices and I/O, interrupt and traps
- Application, Multiprogramming/time-sharing

Agenda

- OS Boot Sequence and Operation
- Devices and I/O, interrupt and traps
- Application, Multiprogramming/time-sharing

What happens at boot?

- When the computer switches on, it does the same as Venus: the CPU executes instructions from some start address (stored in Flash ROM)



- Bootstrapping: https://en.wikipedia.org/wiki/Bootstrapping

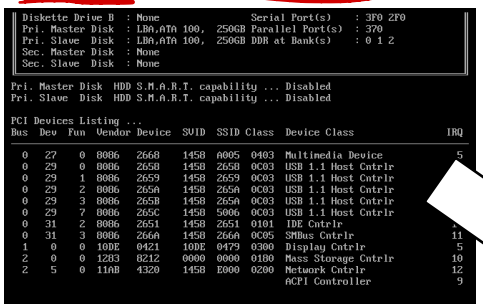
What happens at boot?

- When the computer switches on, it does the same as Venus: the CPU executes instructions from some start address (stored in Flash ROM)

BIOS 找→ bootloader 找→ kernel

1. BIOS: Find a storage device and load first sector (block of data)

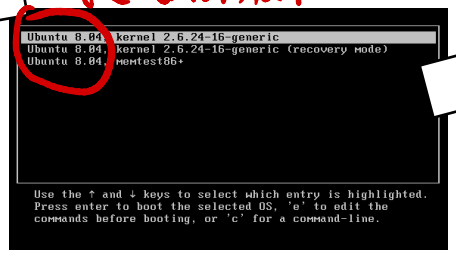
扇区



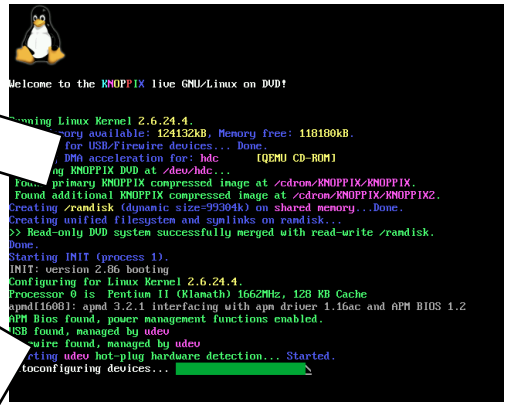
2. Bootloader (stored on, e.g., disk): Load the OS *kernel* from disk into a location in memory and jump into it.

跳转进 OS

可选 kernel 版本



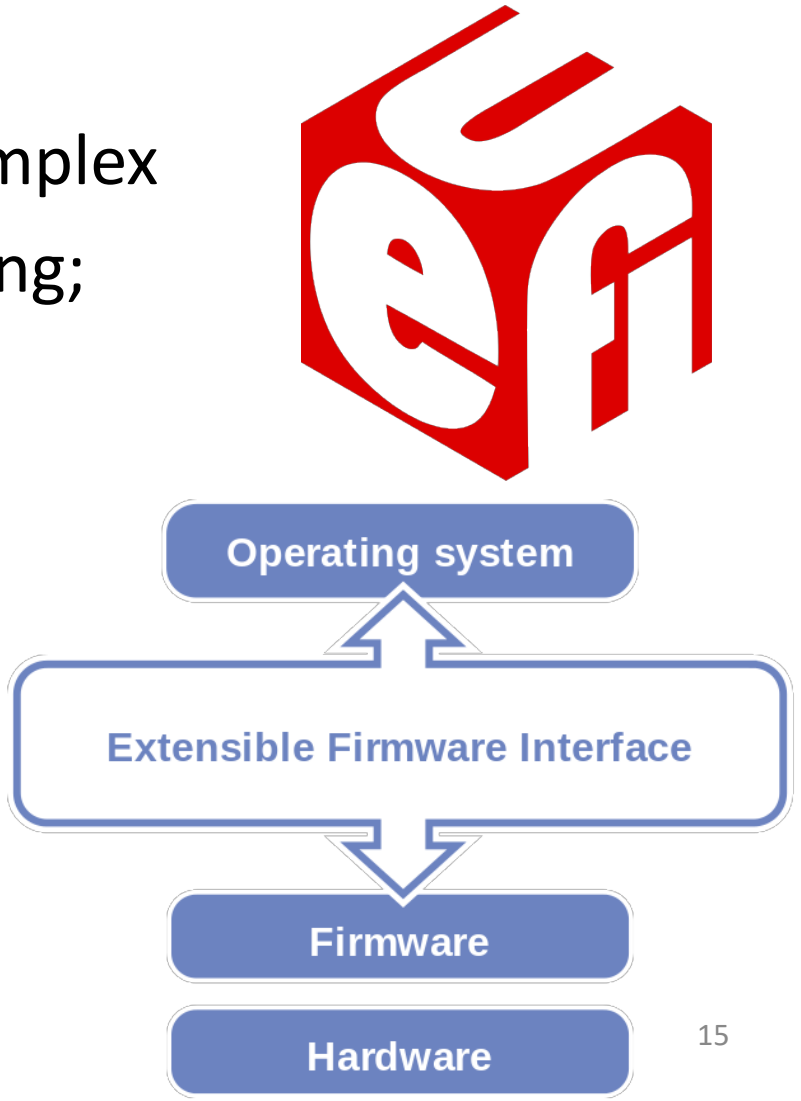
4. Init: Launch an application that waits for input in loop (e.g., Terminal/Desktop/...



3. OS Boot: Initialize services, drivers, etc.

UEFI
Unified Extensible Firmware Interface

- Successor of BIOS
- Much more powerful and complex
- E.g. graphics menu; networking; browsers
- All modern Intel & AMD based computer use UEFI

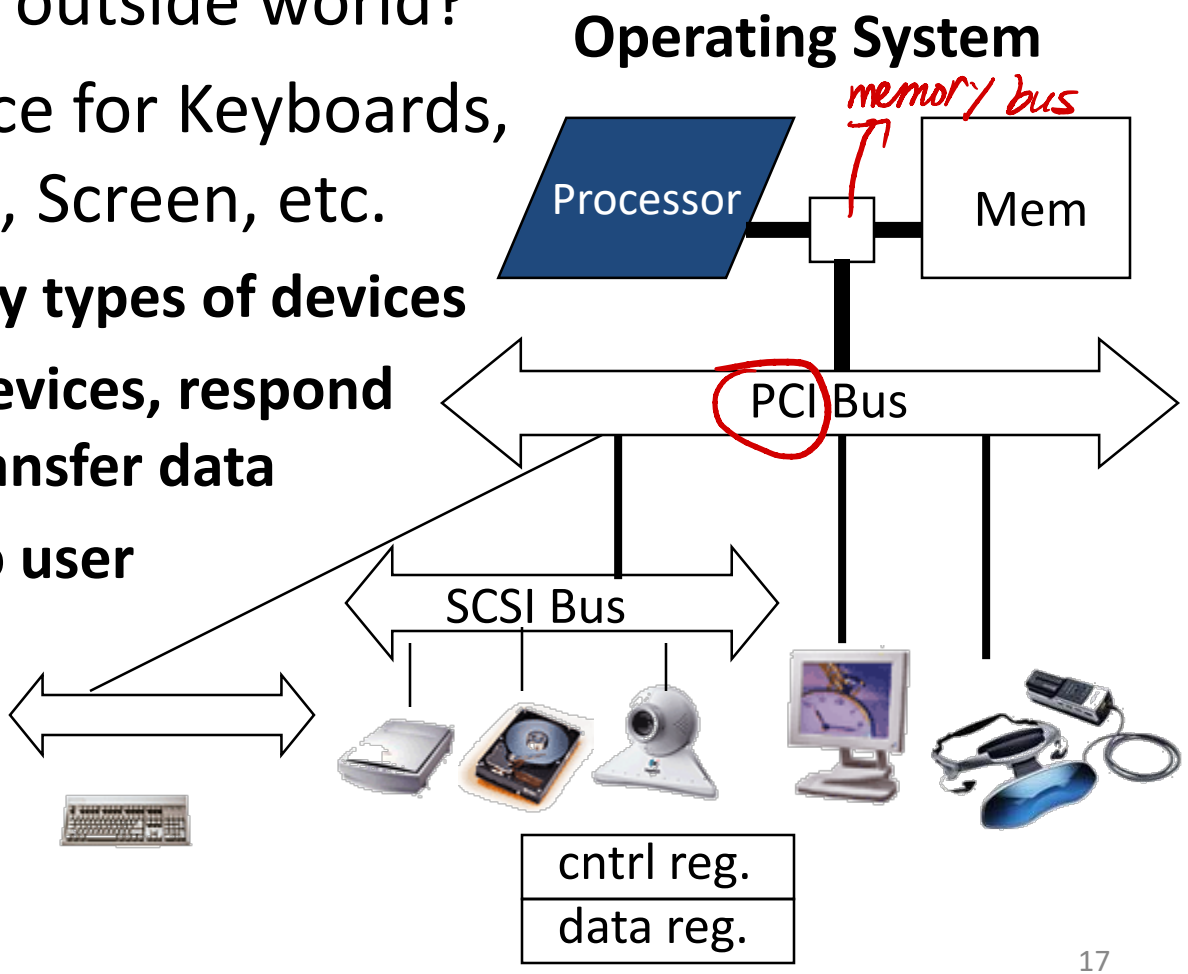


Agenda

- OS Boot Sequence and Operation
- Devices and I/O, interrupt and traps
- Application, Multiprogramming/time-sharing

How to interact with devices?

- Assume a program running on a CPU. How does it interact with the outside world?
- Need I/O interface for Keyboards, Network, Mouse, Screen, etc.
  - **Connect to many types of devices**
  - **Control these devices, respond to them, and transfer data**
  - **Present them to user programs so they are useful**



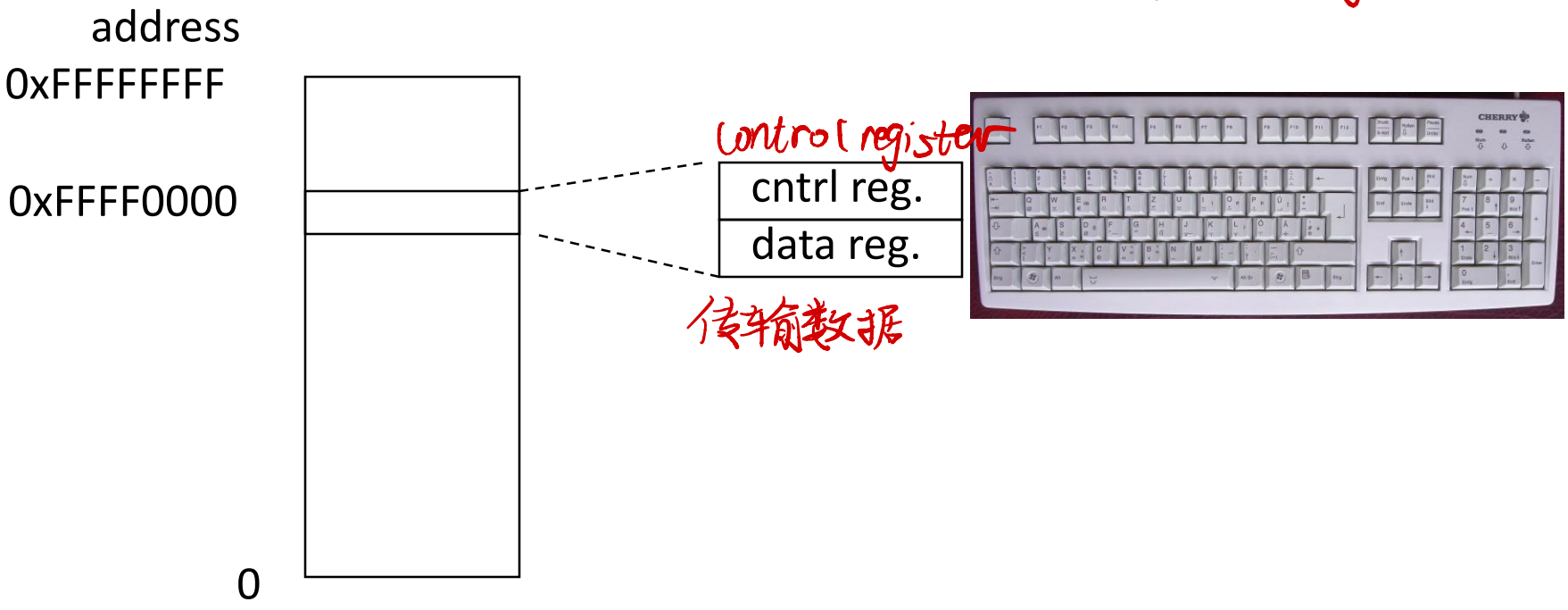
Instruction Set Architecture for I/O

- What must the processor do for I/O?
  - Input: reads a sequence of bytes
  - Output: writes a sequence of bytes
- Interface options
  - Some processors have special input/output instructions
  - Memory Mapped Input/Output (used by RISC-V):
    - Use normal load/store instructions, e.g., lw/sw, for input/output
      - In small pieces
    - A portion of the address space dedicated to IO
    - I/O device registers there (no memory there)

Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices

IO设备都有对应的register



Processor-I/O Speed Mismatch

- 1GHz microprocessor can execute 1 billion load or store instructions per second, or 4,000,000 KB/s data rate
  - I/O data rates range from 0.01 KB/s to 1,250,000 KB/s
- Input: device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act
- Output: device not be ready to accept data as fast as processor stores it
- What to do?

Processor Checks Status before Acting

- Path to a device generally has 2 registers:
  - Control Register, says it's OK to read/write (I/O ready) [think of a flagman on a road] (flag)
  - Data Register, contains data
- Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg (0 => 1) to say it's OK
- Processor then loads from (input) or writes to (output) data register
  - Load from or Store into Data Register resets Ready bit (1 => 0) of Control Register
- This is called "Polling"

input/output 以CPU的视角

I/O Example (polling)

- Input: Read from keyboard into a**0**

```
                li    t0, 0xffff0000 #ffff0000
Waitloop:       lw    t1, 0(t0)      #control
                andi  t1, t1,0x1
                beq   t1, zero, Waitloop
                lw    a0, 4(t0)      #data
```

- Output: Write to display from **a0**

```
                li    t0, 0xffff0000 #ffff0000
Waitloop:       lw    t1, 8(t0)      #control
                andi  t1, t1,0x1
                beq   t1, zero, Waitloop
                sw    a0, 12(t0)     #data
```

"Ready" bit is from processor's point of view!

Cost of Polling?

- Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling
  - Mouse: polled 30 times/sec so as not to miss user movement

% Processor time to poll

- Mouse Polling [clocks/sec]

  = 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]

- % Processor for polling:

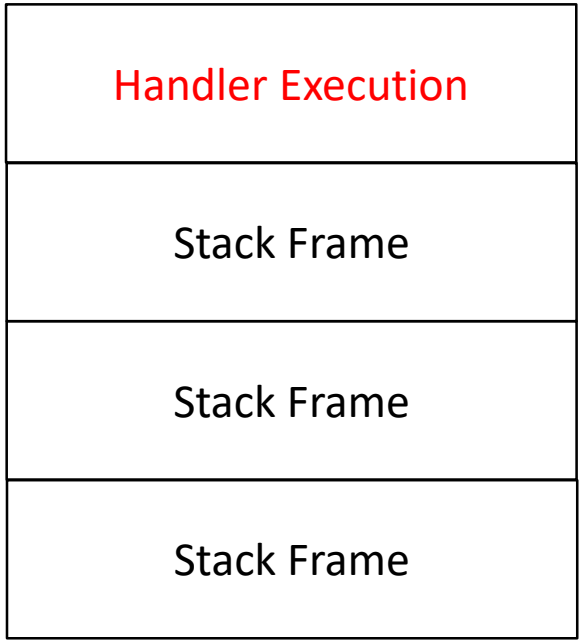
12k $12*10^3$ [clocks/s] / $1*10^9$ [clocks/s] = 0.0012%

  => Polling mouse little impact on processor

What is the alternative to polling?

- Wasteful to have processor spend most of its time "spin-waiting" for I/O to be ready
- Would like an unplanned procedure call that would be invoked only when I/O device is ready
- Solution: use exception mechanism to help I/O.
  - Interrupt program when I/O ready, return when done with data transfer
- Allow to register (post) interrupt handlers: functions that are called when an interrupt is triggered

Interrupt-driven I/O



1. Incoming interrupt suspends instruction stream
2. Looks up the vector (function address) of a handler in an interrupt vector table stored within the CPU
3. Perform a jal to the handler (needs to store any state)
4. Handler run on current stack and returns on finish (thread doesn't notice that a handler was run)

```
handler:  li   t0, 0xffff0000
          lw   t1, 0(t0)
          andi t1, t1,0x1
          lw   a0, 4(t0)
          sw   t1, 8(t0)
          ret
```

```
Label: sll  t1,s3,2
       addu t1,t1,s5
       lw   t1,0(t1)
       add  s1,s1,t1
       addu s3,s3,s4
       bne  s3,s2,abel
```

处理完中断，回到原

Interrupt(SPI0)

CPU Interrupt Table	
SPI0	handler
...	...

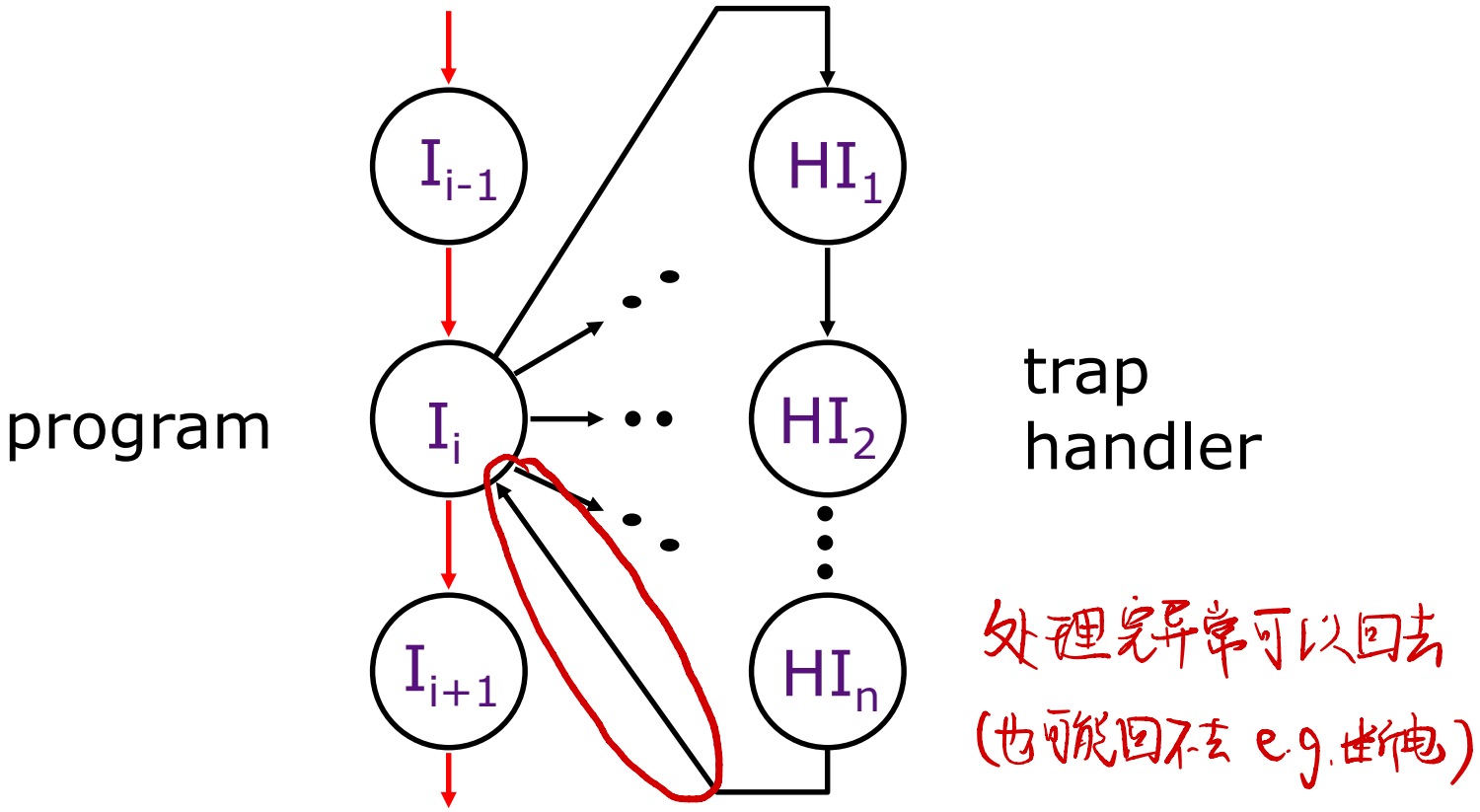
Terminology

In CA (you'll see other definitions in use elsewhere):

- Interrupt – caused by an event *external* to current running program (e.g. key press, mouse activity)
  - Asynchronous to current program, can handle interrupt on any convenient instruction
- Exception – caused by some event during execution of one instruction of current running program (e.g., page fault, bus error, illegal instruction)
  - Synchronous, must handle exception on instruction that causes exception
- Trap – action of servicing interrupt or exception by hardware jump to "trap handler" code

Traps/Interrupts/Exceptions:

altering the normal flow of control



An *external or internal event* that needs to be processed - by another program – the OS. The event is often unexpected from original program's point of view.

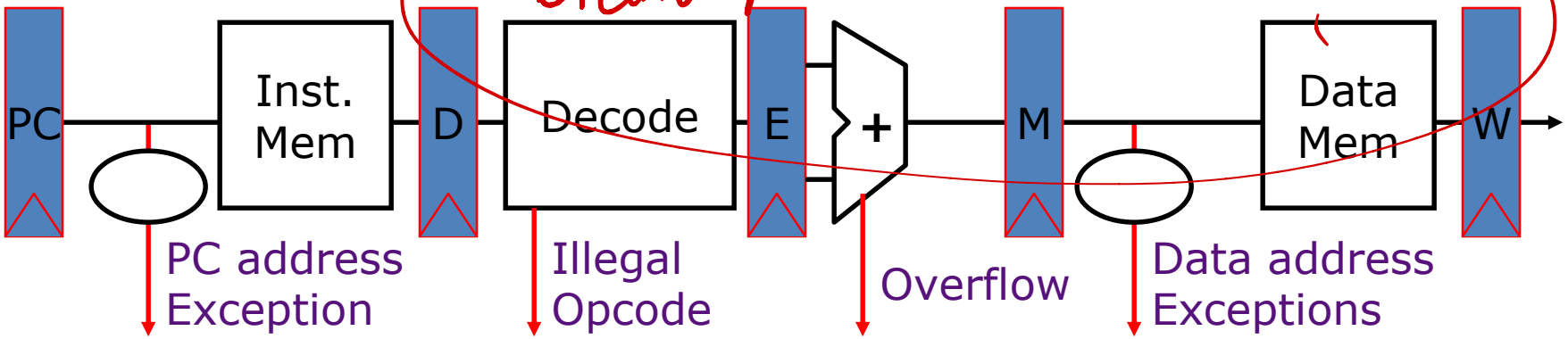
记录异常发生在哪个指令

Supervisor exception program counter

Precise Traps

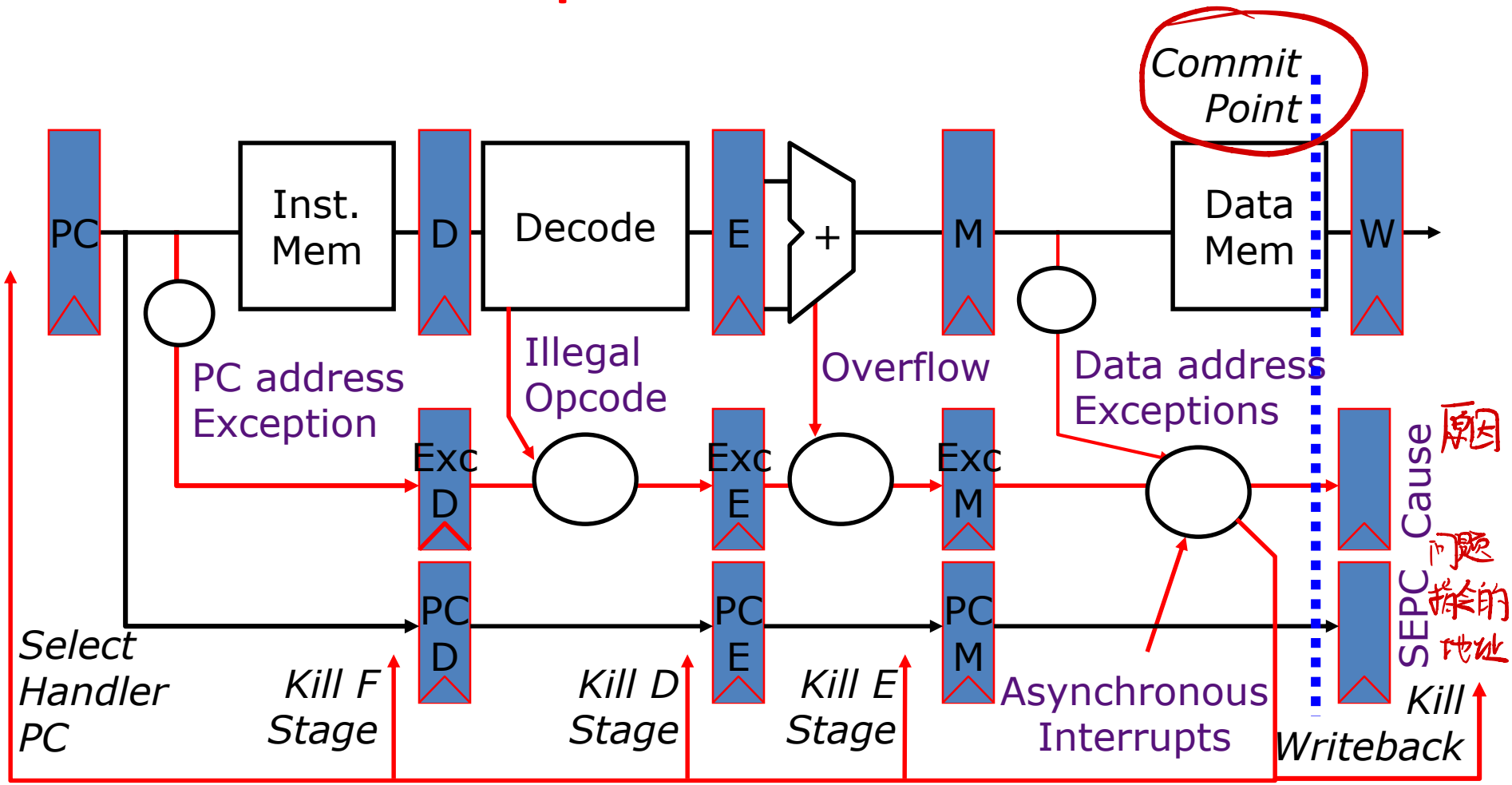
- *Trap handler's view of machine state is that every instruction prior to the trapped one has completed, and no instruction after the trap has executed.*
- Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction (SEPC register will hold the instruction address)
  - Interrupt handler software doesn't need to understand the pipeline of the machine, or what program was doing!
  - More complex to handle trap caused by an exception than interrupt
- Providing precise traps is tricky in a pipelined superscalar out-of-order processor!
  - But handling imprecise interrupts in software is even worse.

Trap Handling in 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

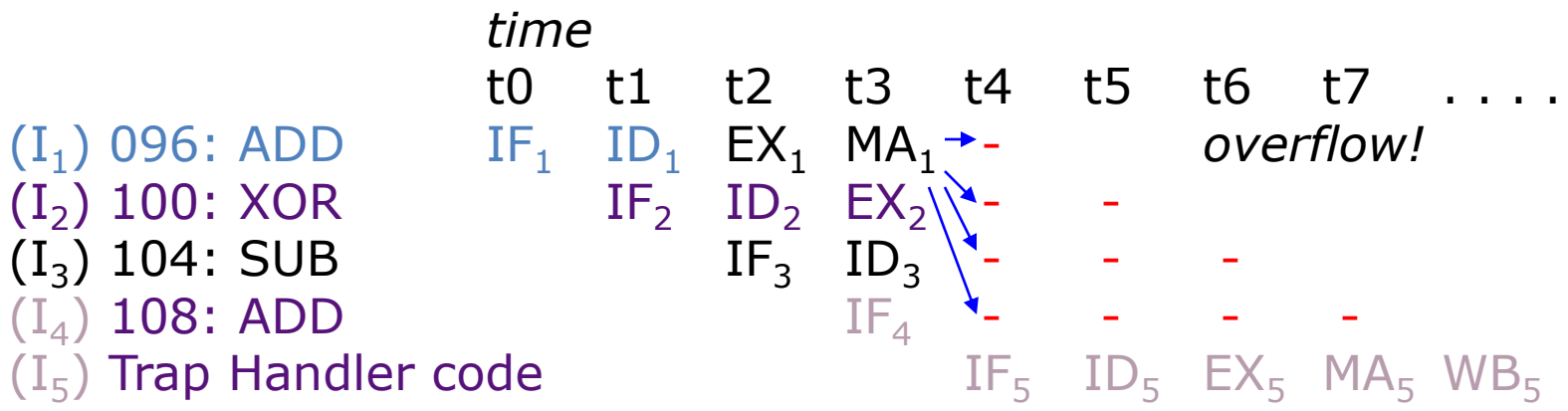
Save Exceptions Until Commit



Handling Traps in In-Order Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier instructions override exceptions in later instructions
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point
- If exception/interrupt at commit: update Cause and SEPC registers, kill all stages, inject handler PC into fetch stage

Trap Pipeline Diagram



Agenda

- OS Boot Sequence and Operation
- Devices and I/O, interrupt and trap
- Application, Multiprogramming/time-sharing

Launching Applications

- Applications are called “processes” in most OSs.
  - Process: separate memory space;
  - Thread: shared memory space.
- Created by another process calling into an OS routine (using a “syscall”, more details later).
  - Depends on OS, but Linux uses fork to create a new process, and execve to load application.
- Loads executable file from disk (using the file system service) and puts instructions & data into memory (.text, .data sections), prepare stack and heap.
- Set argc and argv, jump into the main function.

Supervisor Mode

- If something goes wrong in an application, it could crash the entire machine.
  - And what about malware, etc.?
- The OS may need to enforce resource constraints to applications (e.g., access to devices).
- To help protect the OS from the application, CPUs have a supervisor mode bit.
  - When not in supervisor mode (user mode), a process can only access a subset of instructions and (physical) memory.
  - Process can enter the supervisor mode by using an interrupt, and change out of supervisor mode using a special instruction.

Syscalls

- What if we want to call into an OS routine? (e.g., to read a file, launch a new process, send data, etc.)
  - Need to perform a syscall: set up function arguments in registers, and then raise software interrupt
  - OS will perform the operation and return to user mode
- This way, the OS can mediate access to all resources, including devices and the CPU itself.

Multiprogramming

- The OS runs multiple applications at the same time.
- But not really (unless you have a core per process)
  - Time-sharing processor
- When jumping into process, set timer interrupt.
  - When it expires, store PC, registers, etc. (process state).
  - Pick a different process to run and load its state.
  - Set timer, change to user mode, jump to the new PC.
- Switches between processes very quickly. This is called a “context switch”.
- Deciding what process to run is called scheduling.

Protection, Translation, Paging

- Supervisor mode does not fully isolate applications from each other or from the OS.
  - Application could overwrite another application's memory.
  - Also, may want to address more memory than we actually have (e.g., for sparse data structures).
- Solution: Virtual Memory. Gives each process the illusion of a full memory address space that it has completely for itself.

In Conclusion

- Once we have a basic machine, it's mostly up to the OS to use it and define application interfaces.
- Hardware helps by providing the right abstractions and features (e.g., Virtual Memory, I/O).