

CS 110

Computer Architecture

An Introduction to *Operating Systems*

Instructors:

Sören Schwertfeger and Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/22s>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C

Review: OpenMP and Multi-threading

- Architectural support for synchronization
 - Load reserved/store conditional
 - Atomic memory operation (AMO)
- OpenMP
 - An approach for programming with multi-threads
 - Shared memory
 - Language extension
 - Pragma, directives, reduction, etc. for multi-threading programs

CA so far...

C Programs

```
#include <stdlib.h>

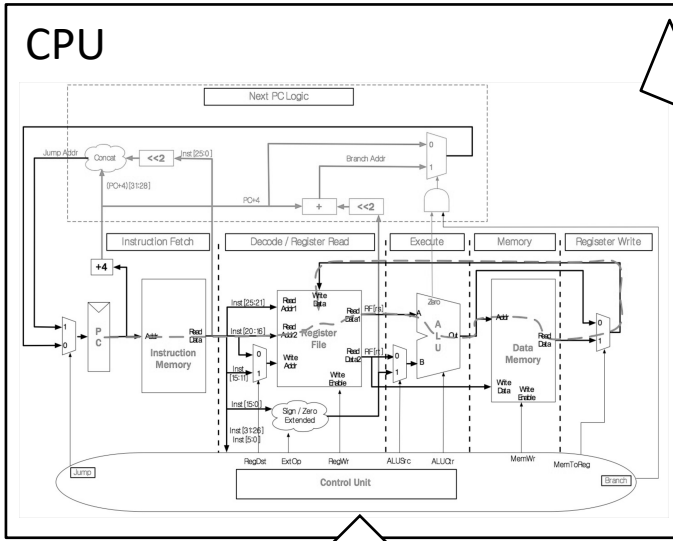
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

Project 1

RISC-V Assembly

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

Project 2



Venus Editor Simulator

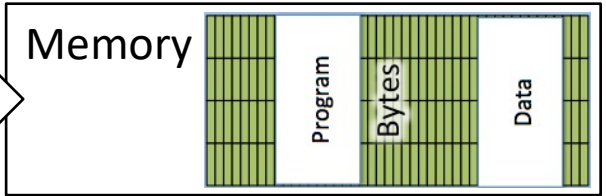
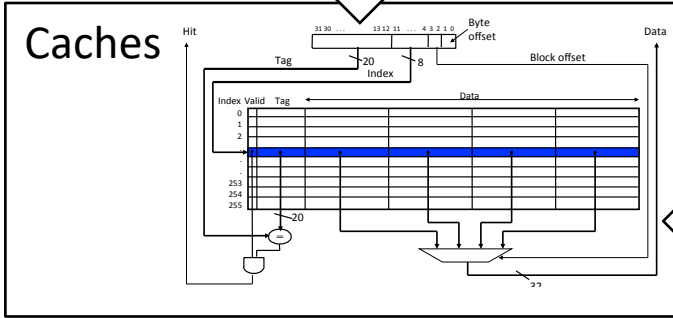
Step Prev Reset Dump Trace

PC	Machine Code	Basic Code	Original Code
0x0	0x00119863	bge x2 x1 16	ble x1, x2, loop
0x4	0x10000317	auipc x6 65536	la x6, somedata # address of "somedata" in x6
0x8	0xFPC30313	addi x6 x6 -4	la x6, somedata # address of "somedata" in x6
0xc	0x00032283	lw x5 0(x6)	lw x5, 0(x6) # (initial) value of "somedata" to x5
0x10	0x00128293	addi x5 x5 1	addi, x5, x5, 1 # x5 ++ 1 (label "loop" points here)
0x14	0xFFDF00EF	jal x1 -4	jal loop # jump to loop

Registers Memory Cache

Register	Value
zero	0x00000000
ra (x1)	0x00000000
sp (x2)	0x7FFFFFF0
gp (x3)	0x10000000
tp (x4)	0x00000000
t0 (x5)	0x00000000
t1 (x6)	0x00000000
t2 (x7)	0x00000000
s0 (x8)	0x00000000

console output



So how is this any different?



Adding I/O

C Programs

```
#include <stdlib.h>

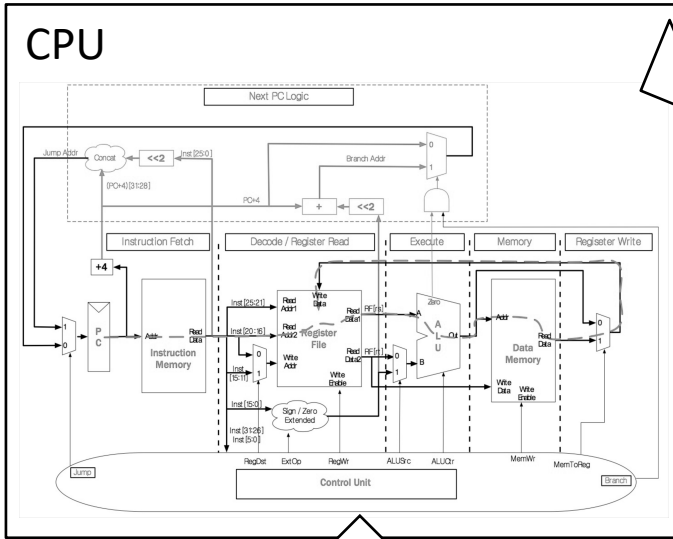
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

Project 1

RISC-V Assembly

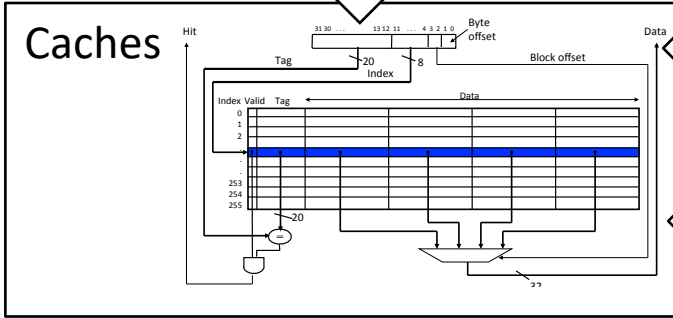
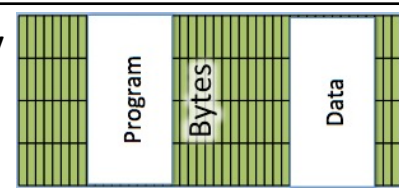
```
.foo
lw    t0, 4(s1)
addi  t1, t0, 3
beq   t1, t2, foo
nop
```

Project 2

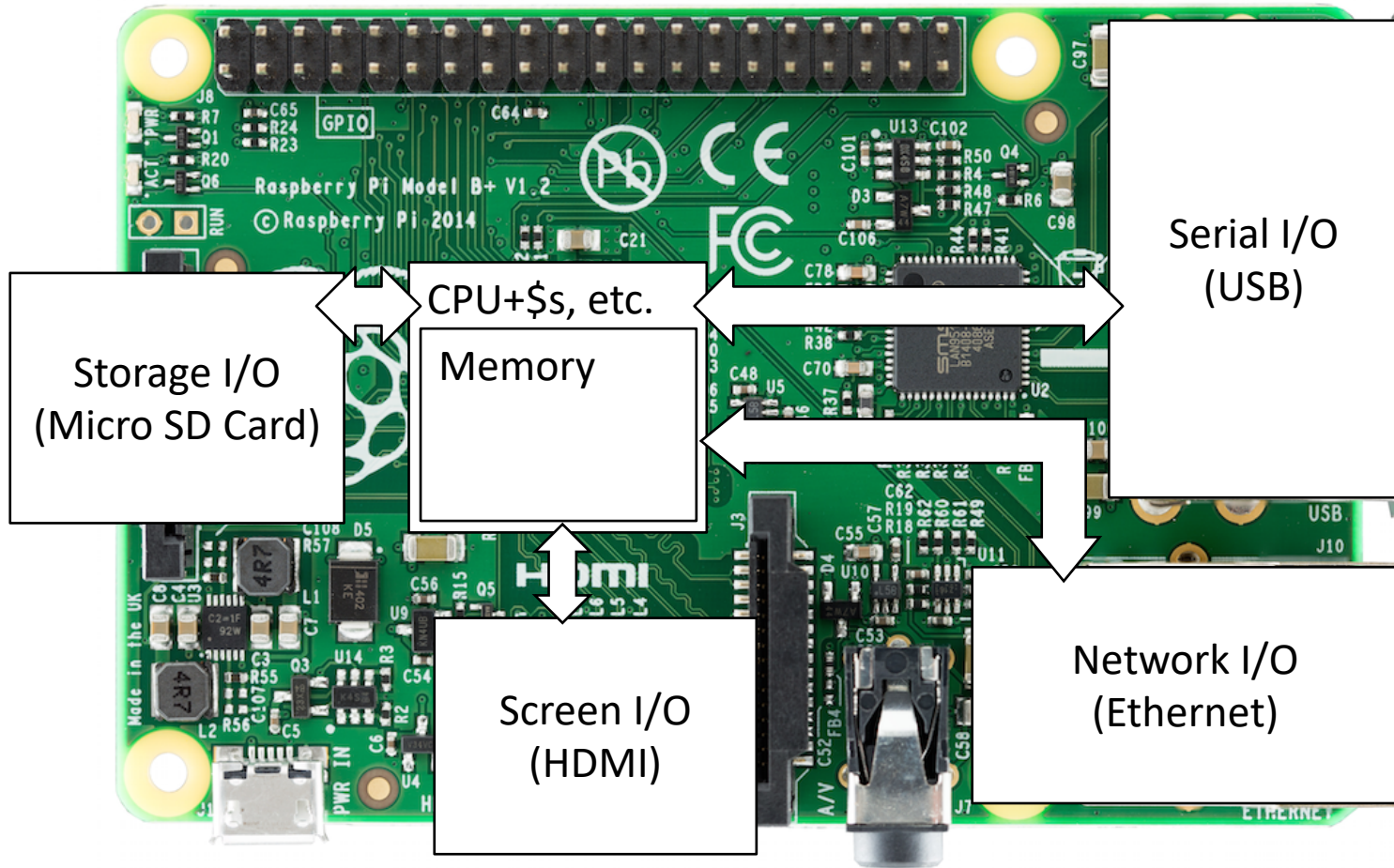


I/O (Input/Output)

Memory



Raspberry Pi

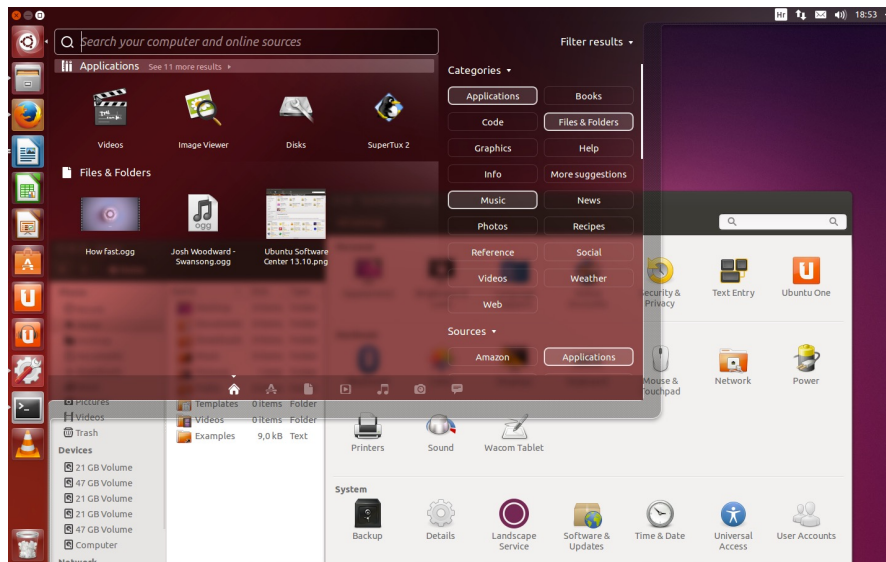


It's a real computer!



But wait...

- That's not the same! Our CS 110 experience isn't like the real world.
- When I switch on my computer, I get this:



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

Well, “just software”

- The biggest piece of software for a personal computer?
- How many lines of code does it have? (guesstimates:)

Year	Size of zipped file
1994	1MB
1996	6MB
2001	23MB
2003	40MB
2007	92MB
2015	118MB
2019	155MB
Apr 2020	166MB
May 2021	179MB
May 2022	189MB



All 7 fictions in txt format
zipped to be **2.5MB**

Say No to Pirated Products
(拒绝盗版)

5MB

linux-5.17.5.tar.gz

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?

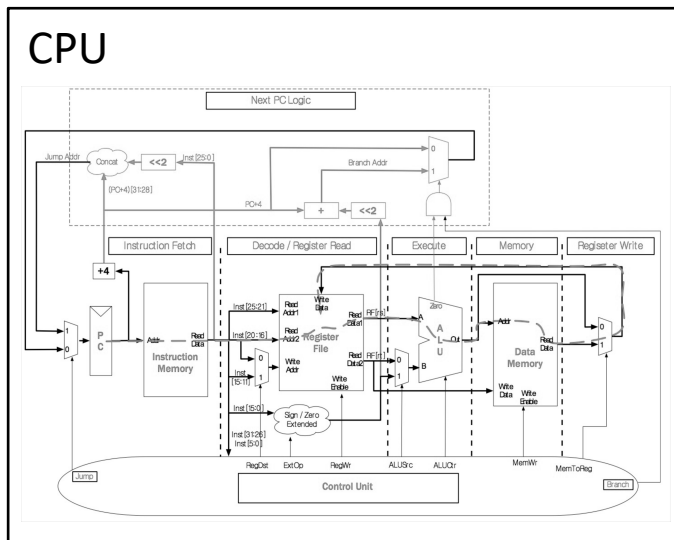
- Provide **interaction** with the outside world
 - Interact with “devices”
 - Disk, screen, keyboard, mouse, network, etc.
- Provide **isolation** between running programs (processes)
 - Each program runs in its own little world
 - Virtual memory

Agenda

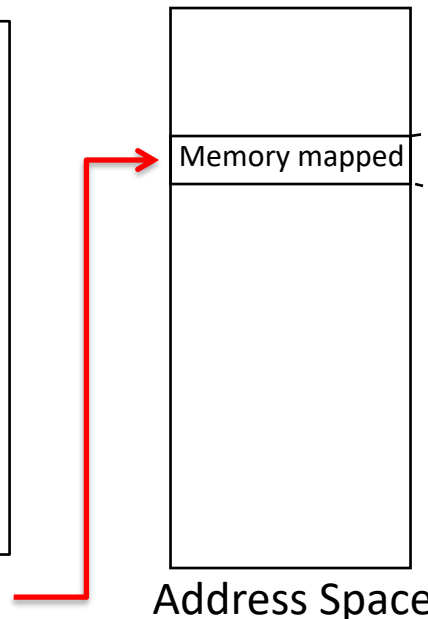
- OS Boot Sequence and Operation
- Devices and I/O, interrupt and traps
- Application, Multiprogramming/time-sharing

What happens at boot?

- When the computer switches on, the CPU executes instructions from some start address (stored in Flash ROM)



PC = 0x2000 (some default value)



```
0x2000:  
addi t0, zero, 0x1000  
lw t0, 4(t0)  
...  
  
(Code to copy firmware into  
regular memory and jump  
into it)
```

- Bootstrapping:

<https://en.wikipedia.org/wiki/Bootstrapping>

What happens at boot?

- When the computer switches on, the CPU executes instructions from some start address (stored in Flash ROM)

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

```
Diskette Drive B : None          Serial Port(s) : 3F0 2F0
Pri. Master Disk : LBA,ATA 100, 250GB Parallel Port(s) : 3F0
Pri. Slave Disk : LBA,ATA 100, 250GB DDR at Bank(s) : 0 1 2
Sec. Master Disk : None
Sec. Slave Disk : None

Pri. Master Disk HDD S.M.A.R.T. capability ... Disabled
Pri. Slave Disk HDD S.M.A.R.T. capability ... Disabled

PCI Devices Listing ...
Bus Dev Fun Vendor Device SUID SSID Class Device Class IRQ
-----
0 27 0 8086 2668 1458 A005 0403 Multimedia Device 5
0 29 0 8086 2658 1458 2658 0C03 USB 1.1 Host Contrlr 5
0 29 1 8086 2659 1458 2659 0C03 USB 1.1 Host Contrlr 5
0 29 2 8086 265A 1458 265A 0C03 USB 1.1 Host Contrlr 5
0 29 3 8086 265B 1458 265B 0C03 USB 1.1 Host Contrlr 5
0 29 7 8086 265C 1458 5906 0C03 USB 1.1 Host Contrlr 5
0 31 2 8086 2651 1458 2651 0101 IDE Contrlr 11
0 31 3 8086 266A 1458 266A 0C05 SMBus Contrlr 11
1 0 0 1002 0421 1002 0479 0300 Display Contrlr 5
2 0 0 1203 8212 0000 0000 0100 Mass Storage Contrlr 10
2 5 0 11AB 4320 1458 E000 0200 Network Contrlr 12
2 5 0 11AB 4320 1458 E000 0200 Network Contrlr 9
ACPI Controller
```

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

```
QUESTION 3:
conv: <speedup> x
re1u: <speedup> x
pool: <speedup> x
fc: <speedup> x
softmax: <speedup> x
Which layer should we opt
<which layer>
(23:04:03 Wed Apr 15 2015) cs61c-ti@hive22 Linux x86_64
~/src/proj3/proj3_start
answers.txt cnn cnn1 cnn.py data LICENSE Makefile test web
(23:04:09 Wed Apr 15 2015) cs61c-ti@hive22 Linux x86_64
~/src/proj3/proj3_start
~/src/proj3/proj3_start $ ls src/
cnn.c main.c python.c util.c
(23:04:16 Wed Apr 15 2015) cs61c-ti@hive22
~/src/proj3/proj3_start $ make cnn
make: 'cnn' is up to date.
(23:04:20 Wed Apr 15 2015) cs61c-ti@hive22 Linux x86_64
~/src/proj3/proj3_start $
```

```
Ubuntu 8.04, kernel 2.6.24-16-generic
Ubuntu 8.04, kernel 2.6.24-16-generic (recovery mode)
Ubuntu 8.04, hextest185+

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the
commands before booting, or 'c' for a command-line.
```

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

```
Welcome to the KNOPPIX live GNU/Linux on DVD!

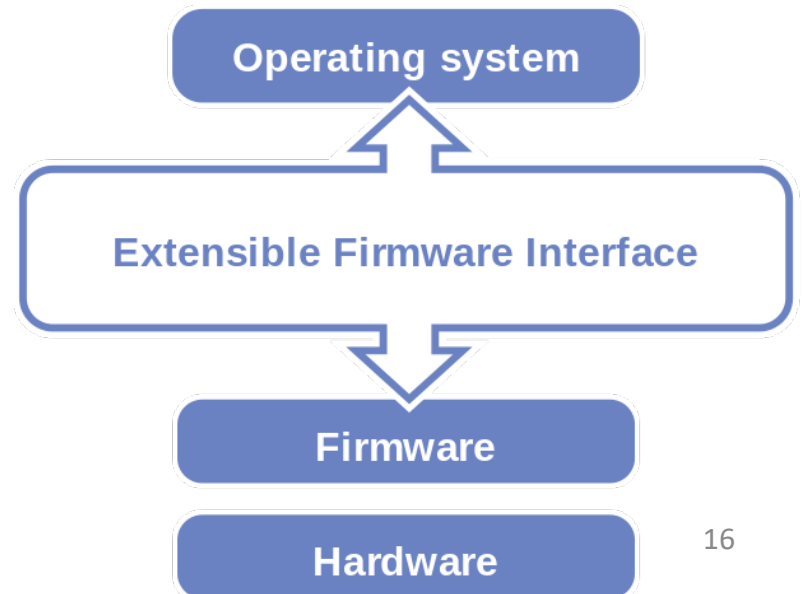
Loading Linux Kernel 2.6.24.4.
Memory available: 124132kB. Memory free: 110180kB
for USB/firewire devices... Done.
DMA acceleration for: hdc [QPMU CD-ROM]
Loading KNOPPIX DVD at /dev/hdc...
Found primary KNOPPIX compressed image at /cdrom/KNOPPIX/KNOPPIX.
Found additional KNOPPIX compressed image at /cdrom/KNOPPIX/KNOPPIX2.
Creating /randisk (dynamic size=99304k) on shared memory...Done.
Creating unified filesystem and symlinks on /randisk...
Done.
>> Read-only DVD system successfully merged with read-write /randisk.
Done.
Starting INIT (process 1).
INIT: version 2.86 booting
Configuring for Linux Kernel 2.6.24.4.
Processor 0 is Pentium III (Klamath) 1662MHz, 128 KB Cache
apmd(1668): apmd 2.7.1 interfacing with apm driver 1.16ac and APM BIOS 1.2
APM Bios found, power management functions enabled.
USB found, managed by udev
firewire found, managed by udev
Loading udev hot-plug hardware detection... Started.
Detecting and configuring devices...
```

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

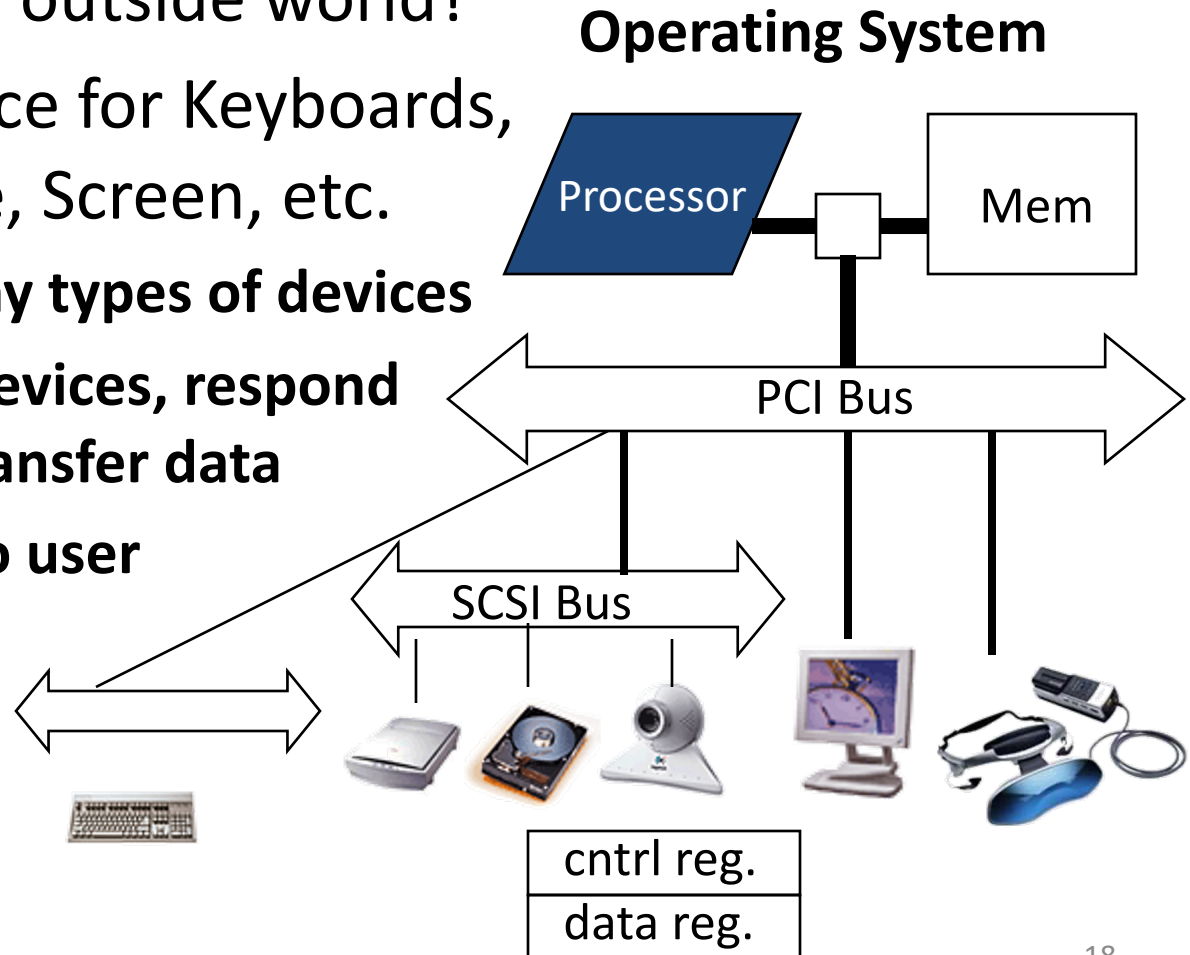


Agenda

- 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**

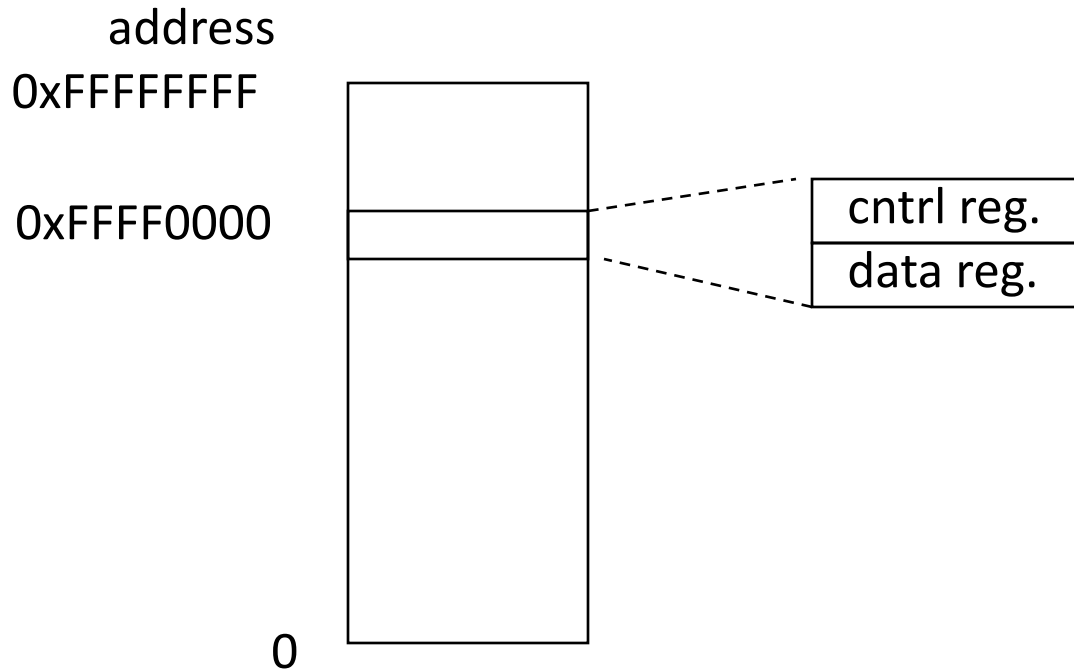


Instruction Set Architecture for I/O

- 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



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]
 - **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**"

I/O Example (polling)

- Input: Read from keyboard into a0

```
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

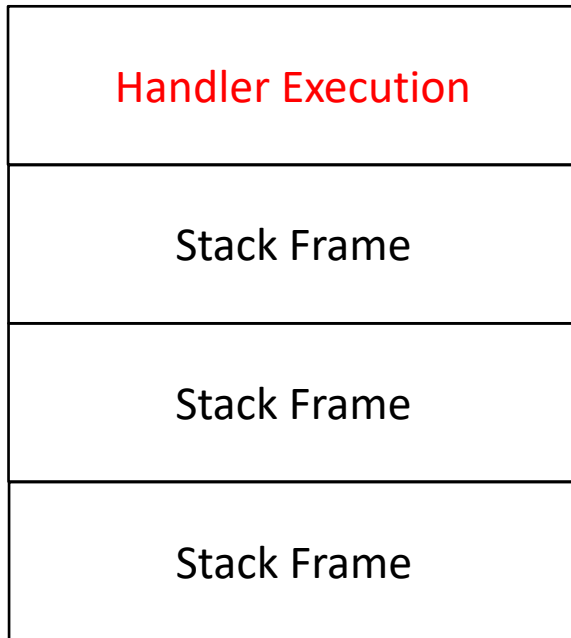
% Processor time to poll

- Mouse Polling [clocks/sec]
= 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]
- % Processor for polling:
 $12 * 10^3$ [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
```



Interrupt(SPI0)

CPU Interrupt Table	
SPI0	handler
...	...

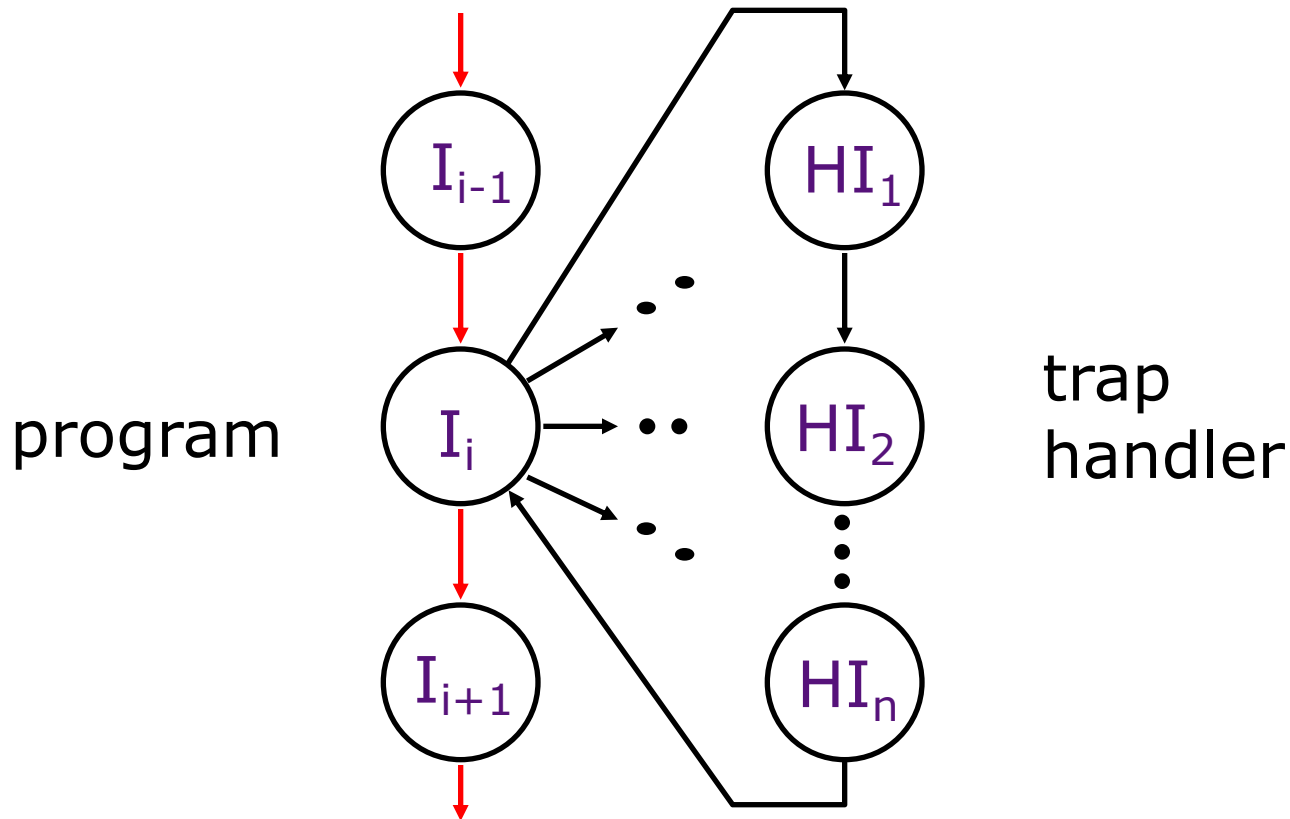
Terminology

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



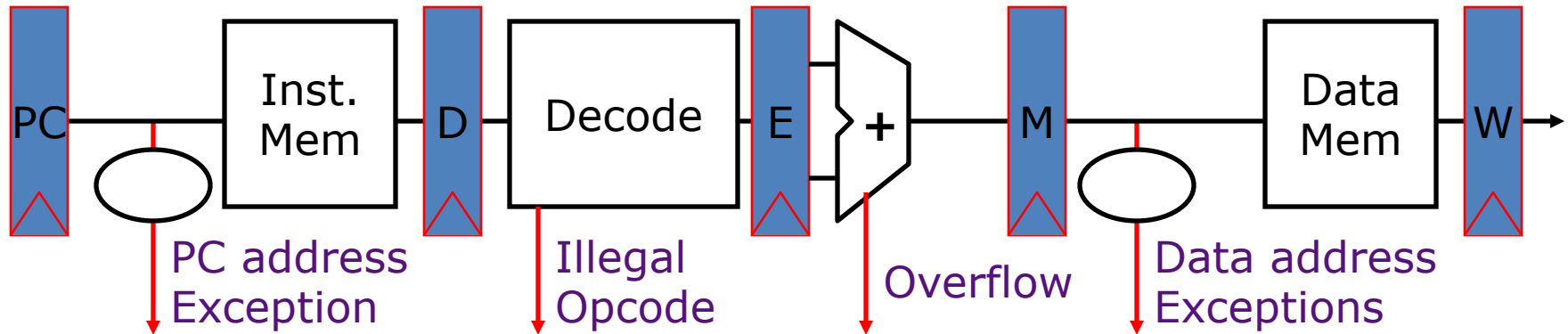
An *external or internal event* that needs to be processed - by another program - the OS. The event is often unexpected from original program's point of view.

Precise Traps

Supervisor
exception
program counter

- *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



→ Asynchronous Interrupts

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

In Conclusion

- Once we have a basic machine, it's mostly up to the OS to use it and define application interfaces.
- I/O
 - Polling
 - Interrupt
- Exception, interrupt, trap
 - Precise trap