

CS 61C: Great Ideas in Computer
Architecture (Machine Structures)
Operating Systems, Interrupts

Instructors:

Nicholas Weaver & Vladimir Stojanovic

<http://inst.eecs.berkeley.edu/~cs61c/>

CS61C so far...

C Programs

```
#include <stdlib.h>

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

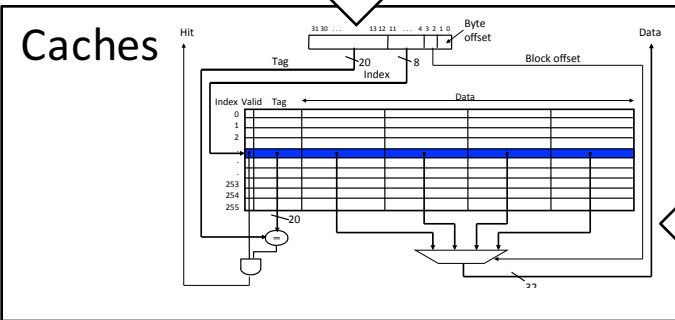
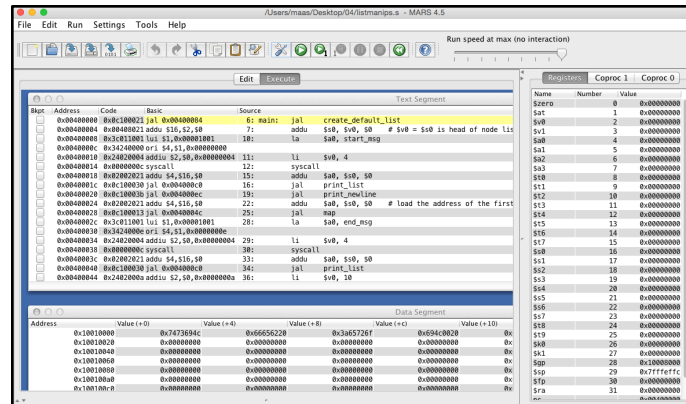
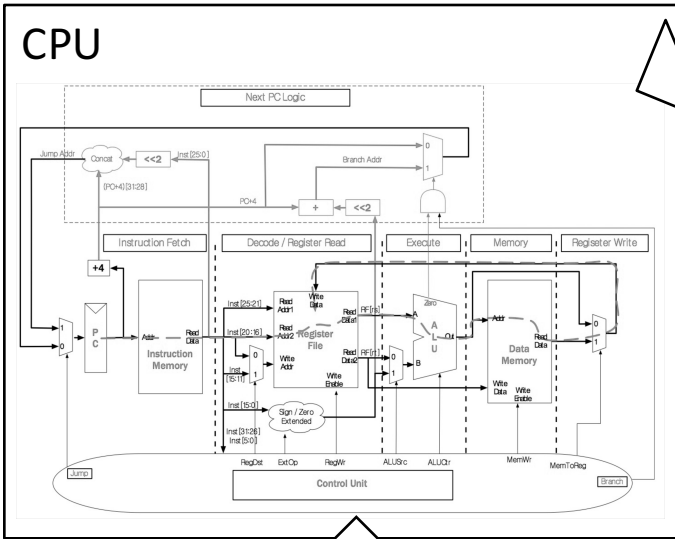
MIPS Assembly

```
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

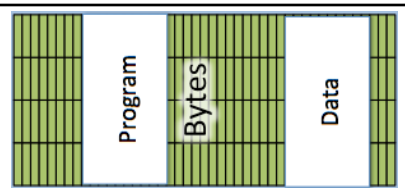
Project 2

Project 1

Labs



Memory



So how is this any different?



Adding I/O

C Programs

```
#include <stdlib.h>

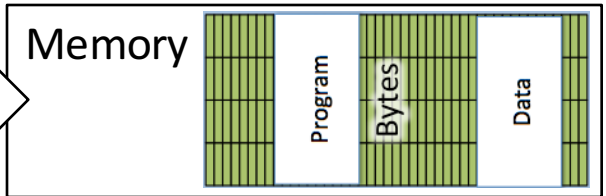
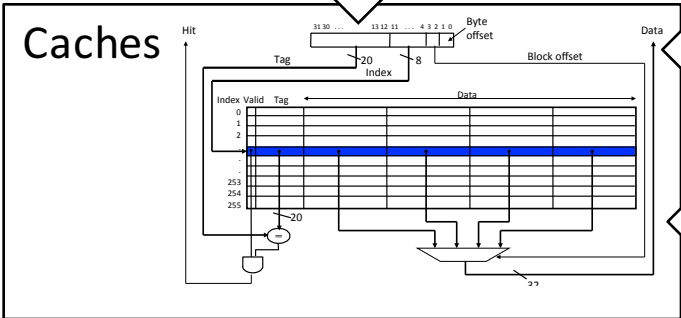
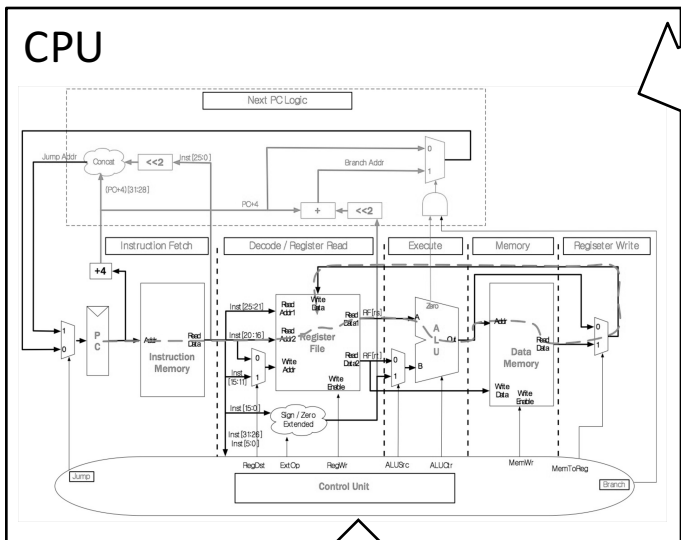
int fib(int n) {
    return
        fib(n-1) +
        fib(n-2);
}
```

MIPS Assembly

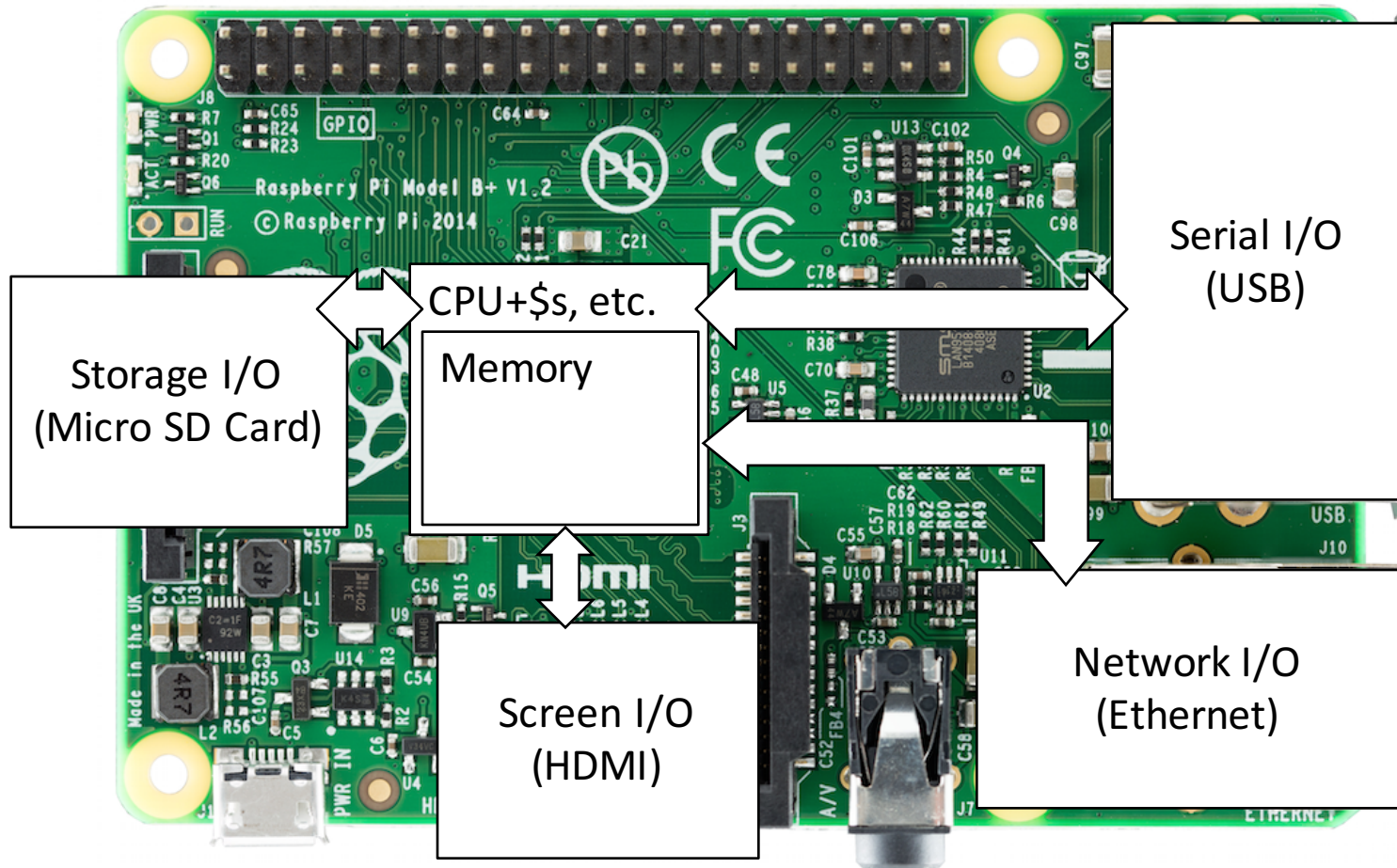
```
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

Project 2

Project 1



Raspberry Pi 3 (\$35 on Amazon)

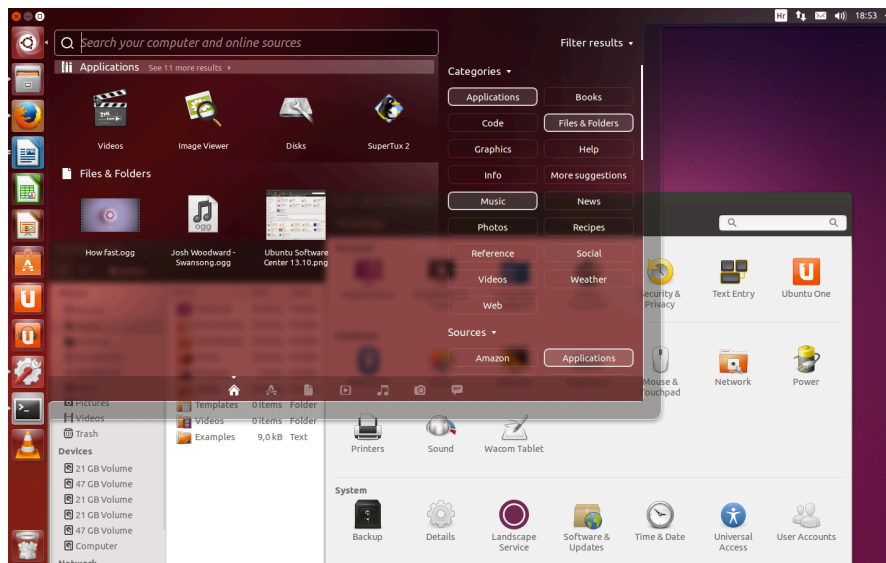


It's a real computer!



But wait...

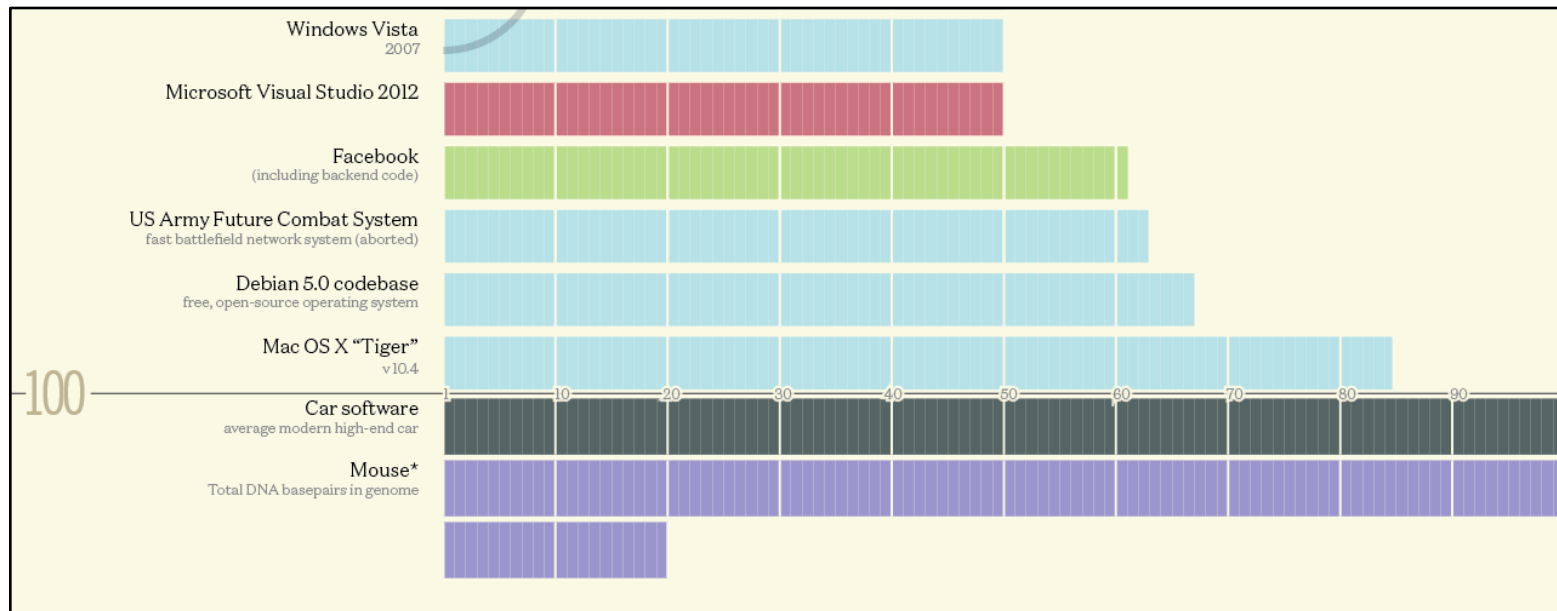
- That's not the same! When we run MARS, it only executes one program and then stops.
- When I switch on my computer, I get this:



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

Well, “just software”

- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:



Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2013
<http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

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, etc.
- Finds and controls all the devices in the machine in a general way (using “device drivers”)

Administrivia

- Project 4 delayed due date to tomorrow
 - But extra credit for turning it in today
- Project 5 will be out ASAP
 - I'm worried that we made it too easy, but eh...

Agenda

- Devices and I/O
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

Agenda

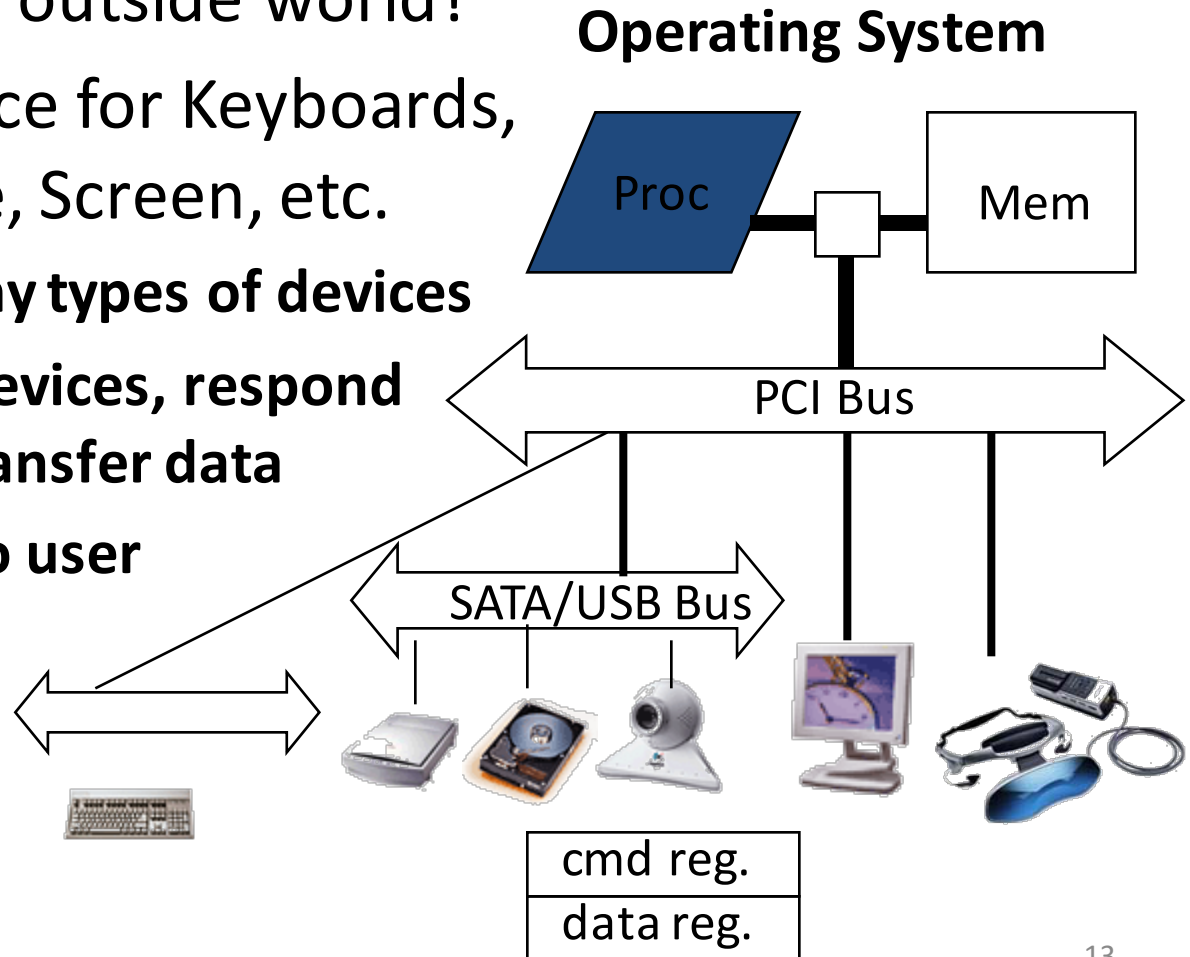
- **Devices and I/O**
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

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**

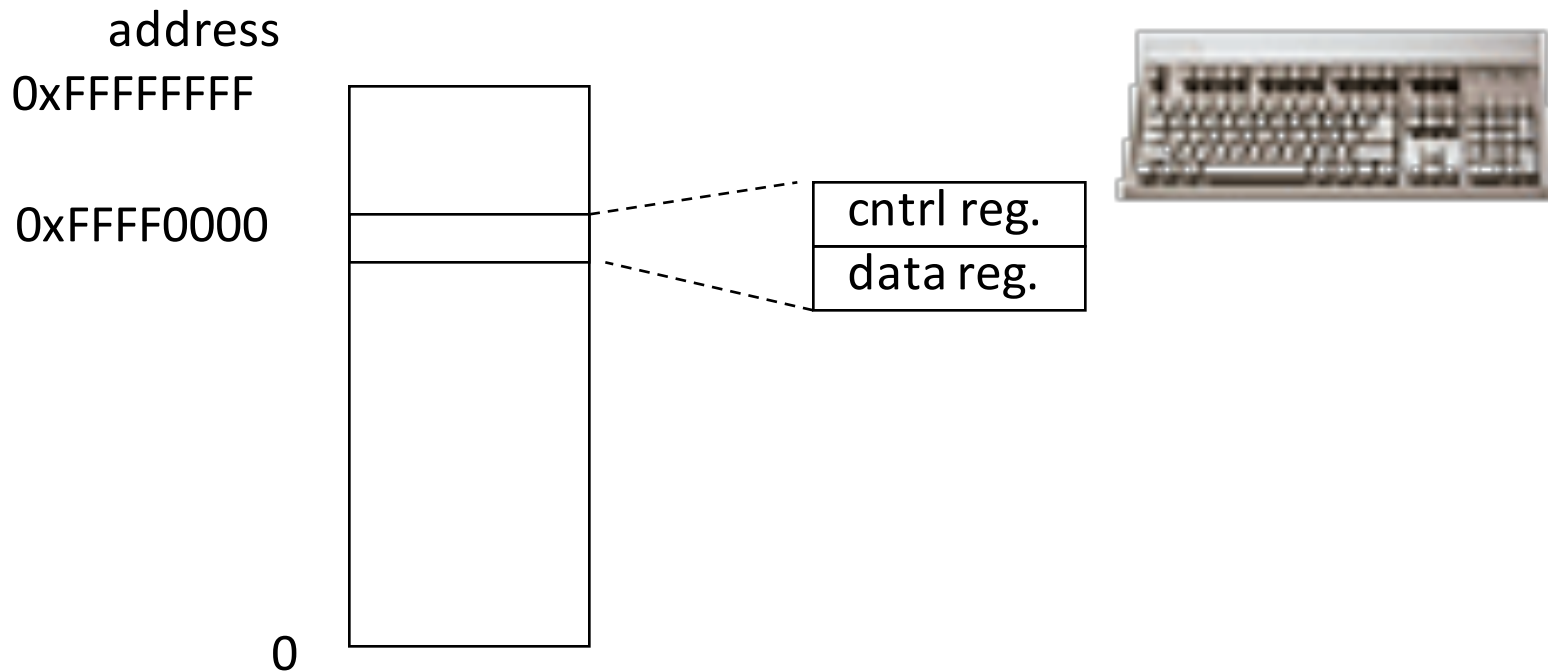


Instruction Set Architecture for I/O

- What must the processor do for I/O?
 - Input: reads a sequence of bytes
 - Output: writes a sequence of bytes
- Some processors have special input and output instructions
- Alternative model (used by MIPS):
 - Use loads for input, stores for output (in small pieces)
 - Called **Memory Mapped Input/Output**
 - A portion of the address space dedicated to communication paths to Input or Output devices (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 1B 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 \$v0

```
Waitloop:    lui    $t0, 0xffff #ffff0000
             lw     $t1, 0($t0) #control
             andi   $t1, $t1, 0x1
             beq    $t1, $zero, Waitloop
             lw     $v0, 4($t0) #data
```

- Output: Write to display from \$a0

```
Waitloop:    lui    $t0, 0xffff #ffff0000
             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
 - Hard disk: assume transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed. (we'll come up with a better way to do this)

% 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

Clicker Time

Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. No transfer can be missed. What percentage of processor time is spent in polling (assume 1GHz clock)?

- A: 2%
- B: 4%
- C: 20%
- D: 40%
- E: 80%

% Processor time to poll hard disk

- Frequency of Polling Disk
= 16 [MB/s] / 16 [B/poll] = 1M [polls/s]
- Disk Polling, Clocks/sec
= 1M [polls/s] * 400 [clocks/poll]
= 400M [clocks/s]
- % Processor for polling:
 $400 * 10^6$ [clocks/s] / $1 * 10^9$ [clocks/s] = 40%

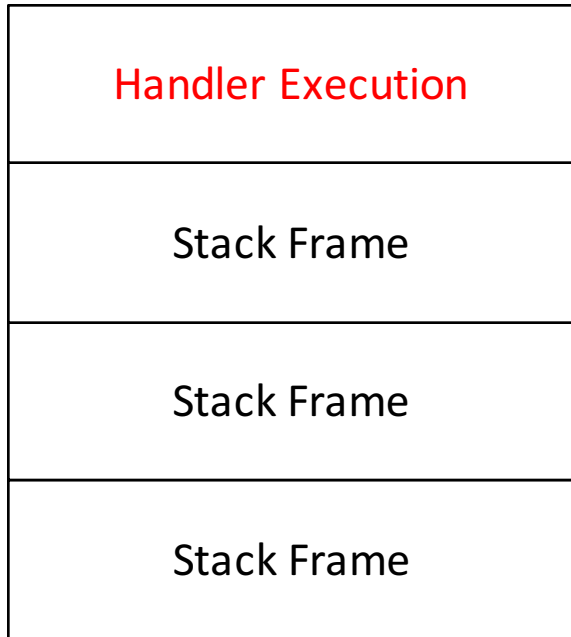
☞ Unacceptable

(Polling is only part of the problem – main problem is that accessing in small chunks is inefficient)

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



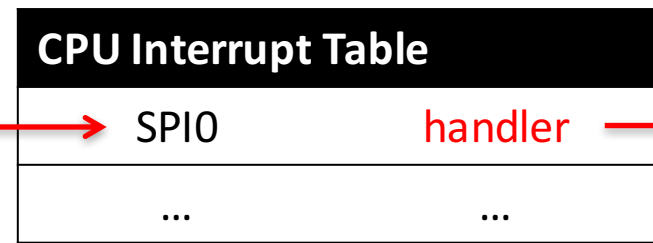
1. Incoming interrupt suspends instruction stream
2. Looks up the vector (function address) of a handler in an interrupt vector table stored within the CPU
3. Perform a jal to the handler (needs to store any state)
4. Handler run on current stack and returns on finish (thread doesn't notice that a handler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

```
handler:  lui  $t0, 0xffff  
          lw   $t1, 0($t0)  
          andi $t1,$t1,0x1  
          lw   $v0, 4($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,Label
```



Interrupt(SPIO)



Direct Memory Access

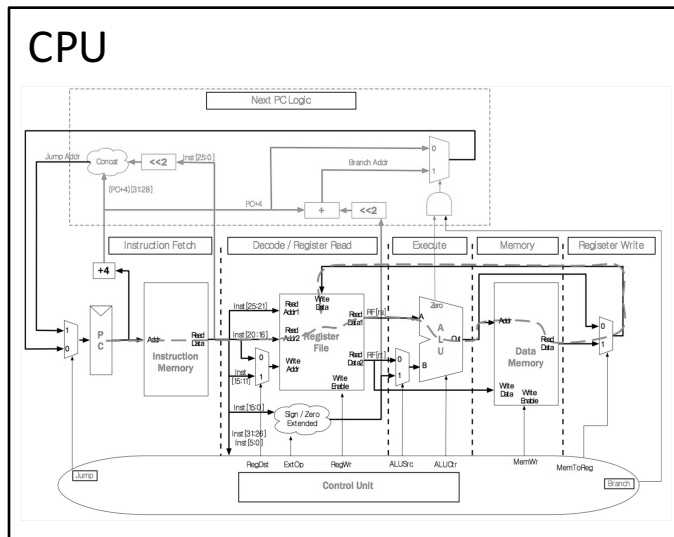
- Complements interrupts:
 - The device itself can directly read or write to a specified block of memory
- Used to buffer transfers
 - DMA write the data
 - ***Then*** trigger an interrupt
- Can even go to great extremes
 - You can buy an FPGA-based network card which will directly write into process buffers
- We will go into this in more detail later on

Agenda

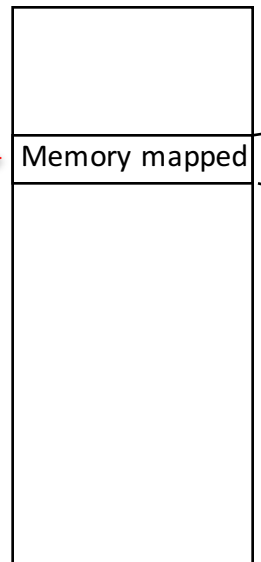
- Devices and I/O
- **OS Boot Sequence and Operation**
- Multiprogramming/time-sharing
- Introduction to Virtual Memory

What happens at boot?

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



PC = 0x2000 (some default value)



```
0x2000:  
addi $t0, $zero, 0x1000  
lw $t0, 4($r0)  
...  
  
(Code to copy firmware into  
regular memory and jump  
into it)
```

Address Space

What happens at boot?

- When the computer switches on, it does the same as MARS: 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) : 370
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 0483 Multimedia Device 5
0 29 0 8086 2658 1458 2658 0083 USB 1.1 Host Contrlr 5
0 29 1 8086 2659 1458 2659 0093 USB 1.1 Host Contrlr x
0 29 2 8086 265A 1458 265A 0083 USB 1.1 Host Contrlr
0 29 3 8086 265B 1458 265B 0083 USB 1.1 Host Contrlr
0 29 7 8086 265C 1458 5086 0093 USB 1.1 Host Contrlr
0 31 2 8086 2651 1458 2651 0101 IDE Contrlr
0 31 3 8086 266A 1458 266A 0085 SMBus Contrlr 11
1 0 0 1002 9421 100E 0479 0380 Display Contrlr 5
2 0 0 1283 8242 0900 0900 0180 Mass Storage Contrlr 10
2 5 0 11AB 4329 1458 E900 0200 Network Contrlr 12
ACPI Contrlr 9
```

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
reLu: <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_starter$ ls
answers.txt cnn cnn.c cnn.py data LICENSE Makefile test web
[23-04-09 Wed Apr 15 2015] cs61c-ti@hive22 Linux x86_64
~/src/proj3/proj3_starter$ ls src/
cnn.c main.c python.c util.c
[23-04-16 Wed Apr 15 2015] cs61c-ti@hive22
~/src/proj3/proj3_starter$ make cnn
make: 'cnn' is up to date.
[23-04-20 Wed Apr 15 2015] cs61c-ti@hive22 Linux x86_64
~/src/proj3/proj3_starter$
```

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: 118180KB.
...
DMA acceleration for: hdc [OPMU CD-ROM]
...
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...
>> 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, 428 KB Cache
symd16003: apmd 3.2.1 interfacing with apm driver 1.16ac and APM BIOS 1.2
APM Bios found, power management functions enabled.
USB found, managed by udev
...
Starting udev hot-plug hardware detection... Started.
...
Configuring devices...
```

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

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

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.
```

Validated Boot...

- The old-school BIOS (Basic Input/Output System) just started running whatever was in the boot sector
 - Allowed all sorts of shenanigans
- Modern firmware (UEFI (Universal Extensible Firmware Interface), iPhone, etc) performs validated boot
 - Cryptographically verifies that the boot code is signed by a valid cryptographic signature
- Essential to maintain a chain of trust
 - Trust the hardware and EFI to validate the boot...
- If you run Windows only, ***turn this on!***

Launching Applications

- Applications are called “processes” in most OSs.
- 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: often just 'mapping' the file into memory to be loaded on demand, which we will get to when talking about virtual memory) and puts instructions & data into memory (.text, .data sections), prepare stack and heap.
- Set argc and argv, jump into the main function.

Supervisor Mode

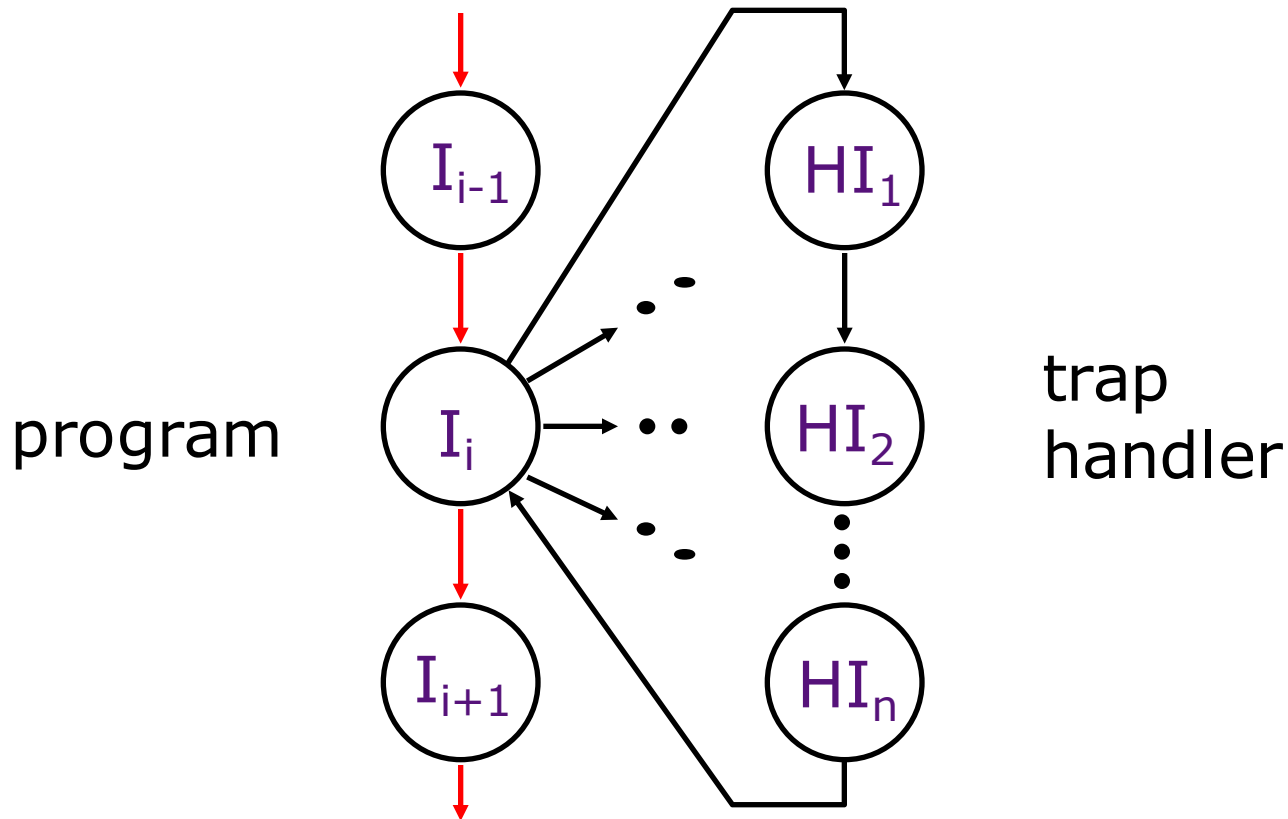
- If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
- The OS may need to enforce resource constraints to applications (e.g., access to devices).
- To help protect the OS from the application, CPUs have a **supervisor mode** bit.
 - A process can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

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
- Also, OS uses interrupts for scheduling process execution:
 - OS sets scheduler timer interrupt then drops to user mode and start executing a user task, when interrupts triggers, switch into supervisor mode, select next task to execute (& set timer) and drop back to user mode.
- This way, the OS can mediate access to all resources, including devices and the CPU itself.

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.

Terminology

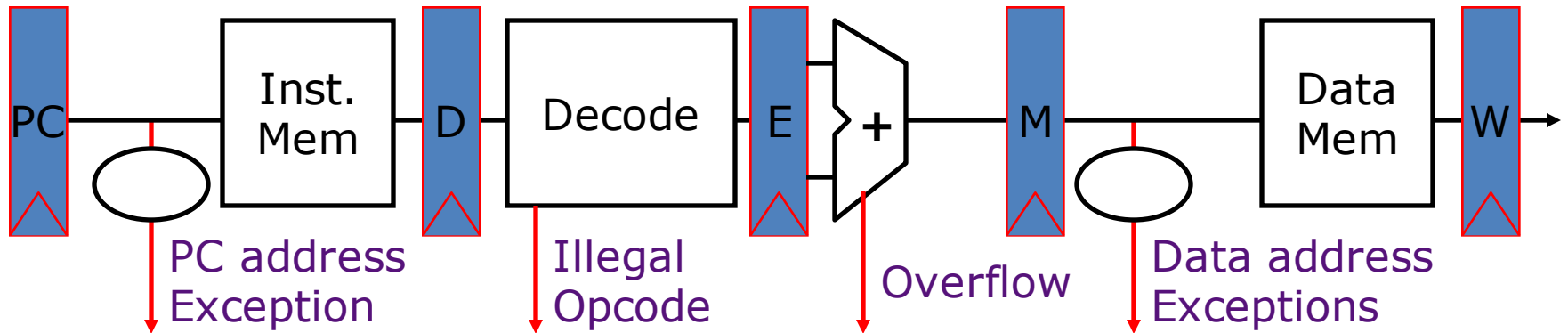
In CS61C (you'll see other definitions in use elsewhere):

- 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
 - Examples include integer overflow (add), lw/sw to invalid memory, not a valid opcode, etc...
 - Or deliberate syscall operation
 - Synchronous, must handle exception on instruction that causes exception
- Trap – action of servicing interrupt or exception by hardware jump to “trap handler” code

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 (EPC 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.

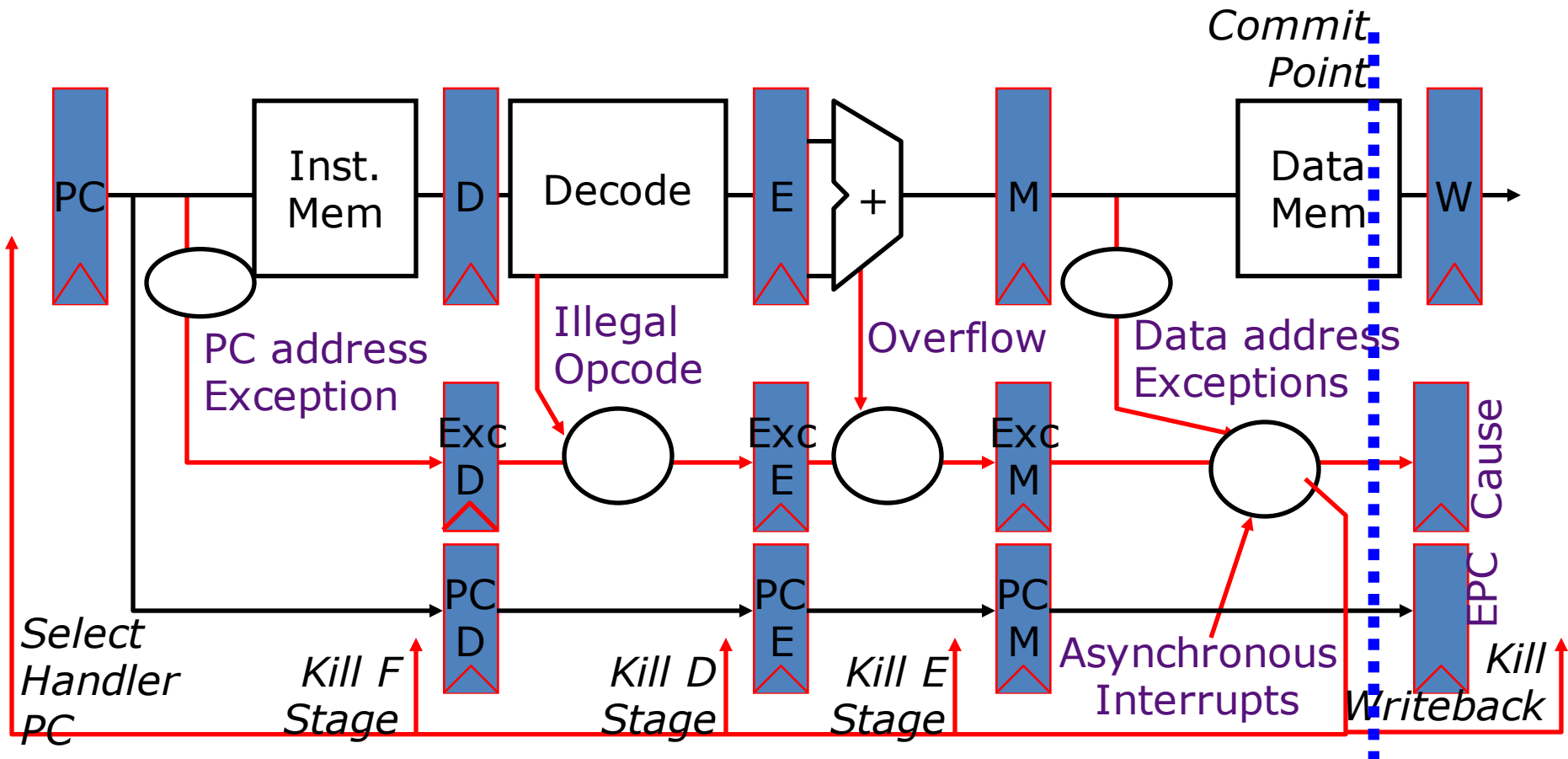
Trap Handling in 5-Stage Pipeline



→ Asynchronous Interrupts

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

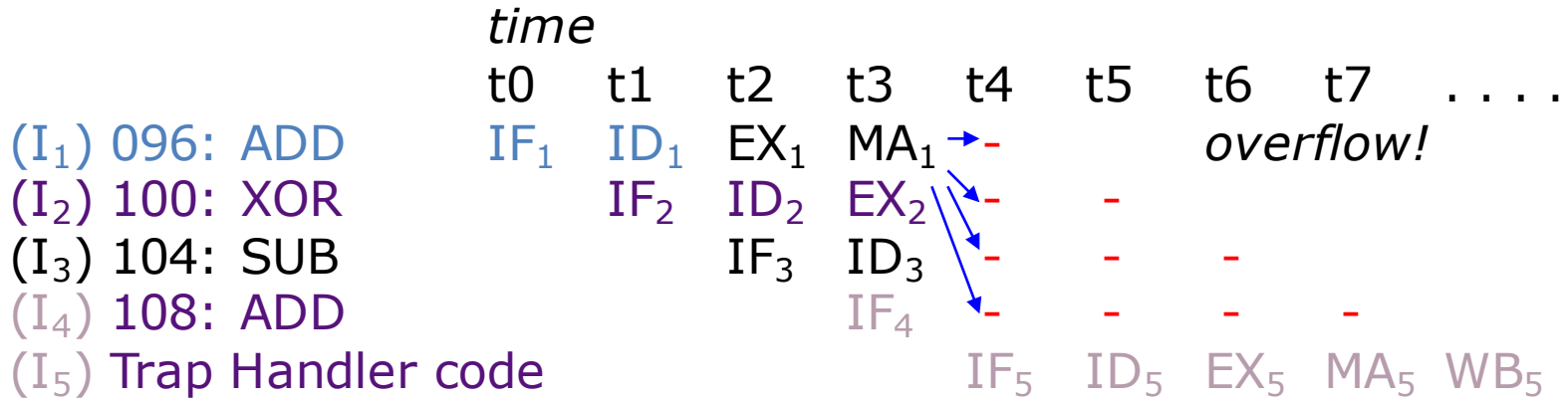
Save Exceptions Until Commit



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 (override others)
- If exception/interrupt at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Trap Pipeline Diagram



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).
- If you want to learn more about operating systems, you should take CS162!
- What's next in CS61C?
 - More details on I/O
 - More about Virtual Memory