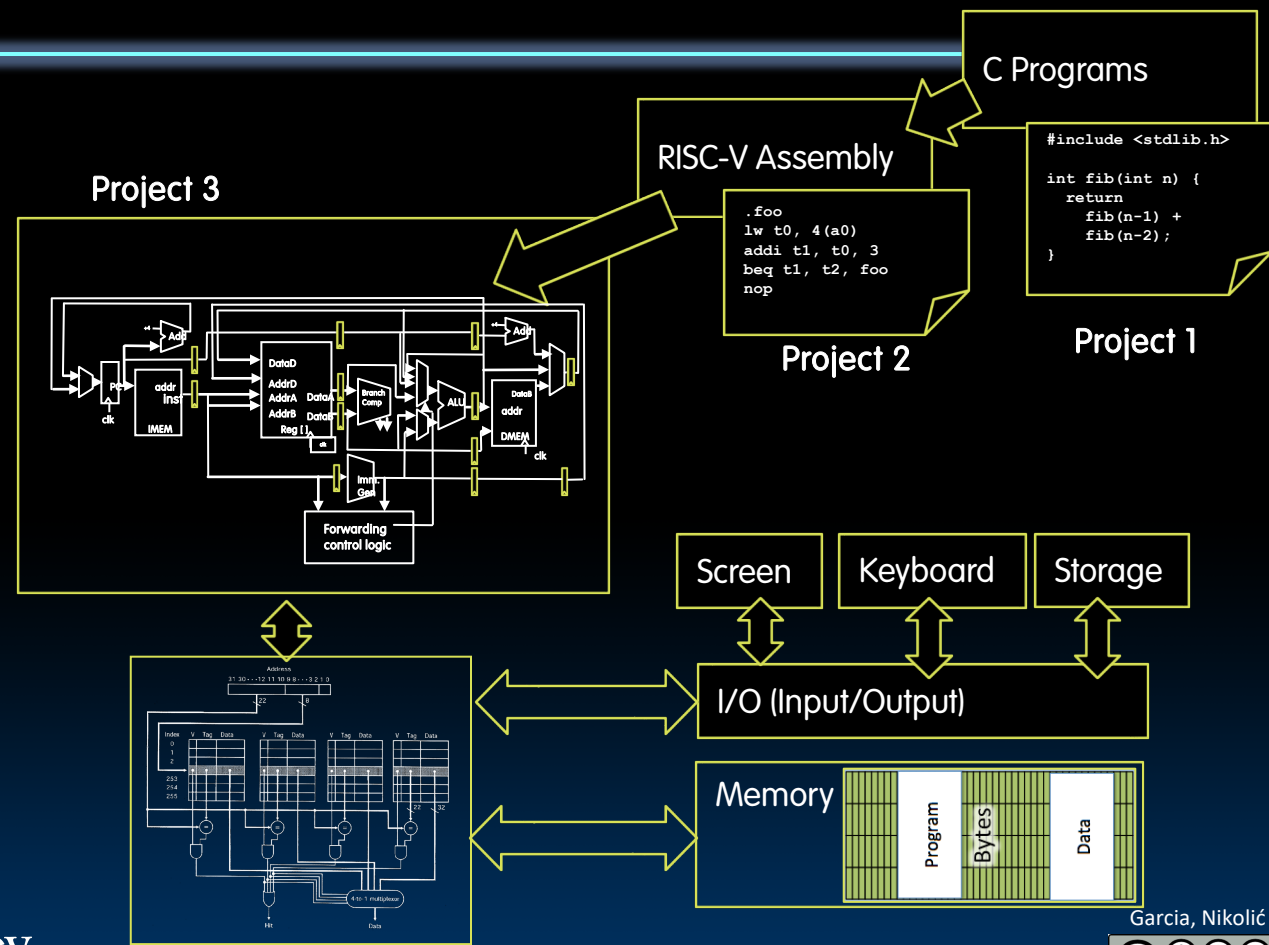


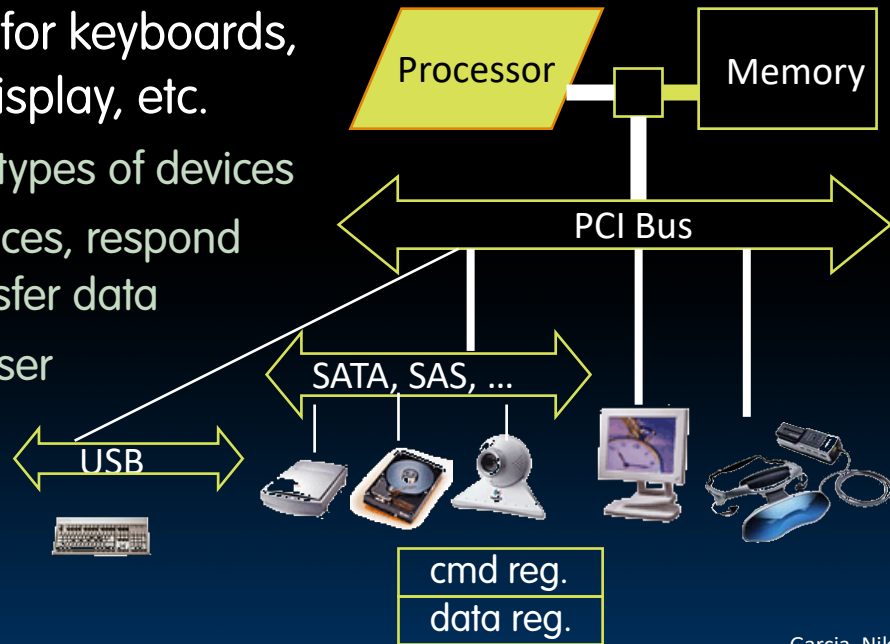
I/O Devices

Adding I/O



How to Interact with Devices?

- Assume a program running on a CPU. How does it interact with the outside world? **Operating System**
- Need I/O interface for keyboards, network, mouse, display, 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

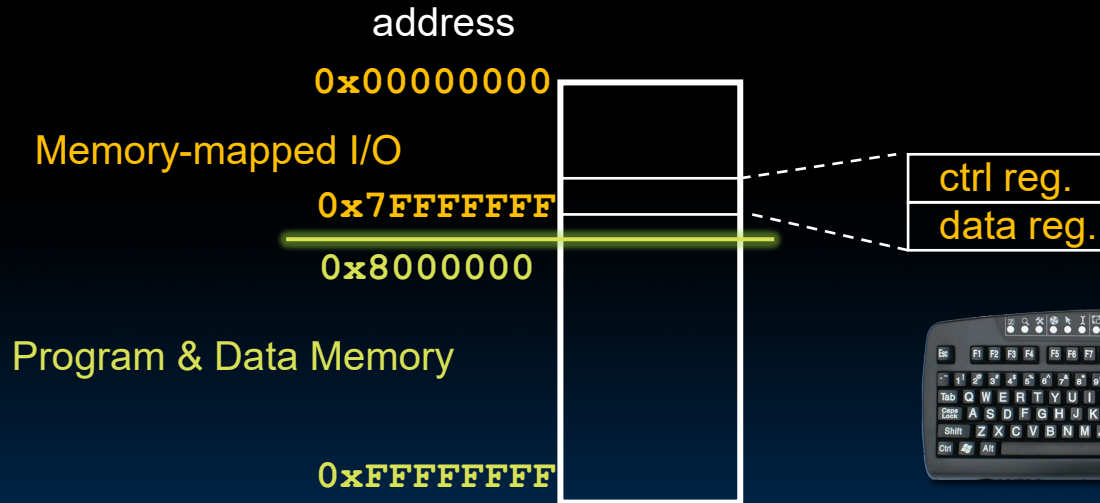


Instruction Set Architecture for I/O

- What must the processor do for I/O?
 - Input: Read a sequence of bytes
 - Output: Write a sequence of bytes
- Interface options
 - a) Special input/output instructions & hardware
 - b) Memory mapped I/O
 - Portion of address space dedicated to I/O
 - I/O device registers there (no memory)
 - Use normal load/store instructions, e.g. **lw/sw**
 - Very common, used by RISC-V

Memory Mapped I/O

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



Processor-I/O Speed Mismatch

- 1 GHz microprocessor I/O throughput:
 - 4 GiB/s (**1w/sw**)
- Typical I/O data rates:
 - 10 B/s (keyboard)
 - 3 MiB/s (Bluetooth 3.0)
 - 0.06-1.25 GiB/s (USB 2/3.1)
 - 7-250 MiB/s (Wifi, depends on standard)
 - 125 MiB/s (G-bit Ethernet)
 - 480MiB/s (SATA3 HDD)
 - 560 MiB/s (cutting edge SSD)
 - 5GiB/s (Thunderbolt 3)
 - 32 GiB/s (High-end DDR4 DRAM)
 - 64 GiB/s (HBM2 DRAM)
 - These are peak rates – actual throughput is lower
- Common I/O devices neither deliver nor accept data matching processor speed

I/O Polling

Polling: Processor Checks Status, Then Acts

- Device registers generally serve two functions:
 - **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 ($0 \rightarrow 1$)
 - Indicates "data available" or "ready to accept data"
- Processor then loads from (input) or writes to (output) data register
 - I/O device resets control register bit ($1 \rightarrow 0$)
- Procedure called "**Polling**"

I/O Example (Polling)

- Input: Read from keyboard into **a0**

```
lui t0 0x7fffff #7ffff000 (io addr)
```

Waitloop: `lw t1 0(t0) #read control` Memory map

```
andi t1 t1 0x1 #ready bit 7ffff000
```

```
beq t1 zero Waitloop 7ffff004
```

```
lw a0 4(t0) #data 7ffff008
```

- Output: Write to display from **a1**

```
lui t0 0x7fffff #7ffff000 7ffff00c
```

Waitloop: `lw t1 8(t0) #write control`

```
andi t1 t1 0x1 #ready bit
```

```
beq t1 zero Waitloop
```

```
sw a1 12(t0) #data
```

input ctrl reg
input data reg
output ctrl reg
output data reg

“Ready” bit is from processor’s point of view!

Cost of Polling?

- Assume for a processor with
 - 1 GHz clock rate
 - Taking 400 clock cycles for a polling operation
 - Call polling routine
 - Check device (e.g., keyboard or WiFi input available)
 - Return
 - What's the percentage of processor time spent polling?
- Example:
 - Mouse
 - Poll 30 times per second
 - Set by requirement not to miss any mouse motion (which would lead to choppy motion of the cursor on the screen)

% Processor Time to Poll Mouse

- Mouse Polling [clocks/sec]
 $= 30 \text{ [polls/s]} * 400 \text{ [clocks/poll]} = 12\text{K [clocks/s]}$
- % Processor for polling:
 $12 * 10^3 \text{ [clocks/s]} / 1 * 10^9 \text{ [clocks/s]} = 0.0012\%$
 \Rightarrow Polling mouse little impact on processor...

(Except that we need to know we should be polling...)

% Processor Time to Poll Hard Disk

- Frequency of Polling Disk (rate at which chunks come could off disk) = $16 \text{ [MB/s]} / 16 \text{ [B/poll]} = 1\text{M [polls/s]}$
- Disk Polling, Clocks/sec =
 $1\text{M [polls/s]} * 400 \text{ [clocks/poll]} = 400\text{M [clocks/s]}$
- % Processor for polling:
 $400 * 10^6 \text{ [clocks/s]} / 1 * 10^9 \text{ [clocks/s]} = 40\%$
=> Unacceptable
(Polling is only part of the problem –
accessing in small chunks is inefficient, too)

I/O Interrupts

Alternatives to Polling: Interrupts

- Polling wastes processor resources
 - Akin to waiting at the door for guests to show up
 - What about a bell?
- Computer lingo for bell:
 - Interrupt
 - Occurs when I/O is ready or needs attention
 - Interrupt current program
 - Transfer control to the trap handler in the operating system
- Interrupts:
 - No I/O activity: Nothing to do
 - Lots of I/O: Expensive – thrashing caches, VM, saving/restoring state



Polling, Interrupts and DMA

- Low data rate (e.g. mouse, keyboard)
 - Use interrupts. Could poll with the timer interrupt, but why?
 - Overhead of interrupts ends up being low
- High data rate (e.g. network, disk)
 - Start with interrupts...
 - If there is no data, you don't do anything!
 - Once data starts coming... Switch to Direct Memory Access (DMA)

Aside: Programmed I/O

- “Programmed I/O”:
 - Standard for ATA hard-disk drives
 - CPU execs lw/sw instructions for all data movement to/from devices
 - CPU spends time doing two things:
 1. Getting data from device to main memory
 2. Using data to compute
- Not ideal because ...
 1. CPU has to execute all transfers, could be doing other work
 2. Device speeds don’t align well with CPU speeds
 3. Energy cost of using beefy general-purpose CPU where simpler hardware would suffice
- Until now CPU has sole control of main memory
- 5% of CPU cycles on Google Servers spent in memcpy() and memmove() library routines!*

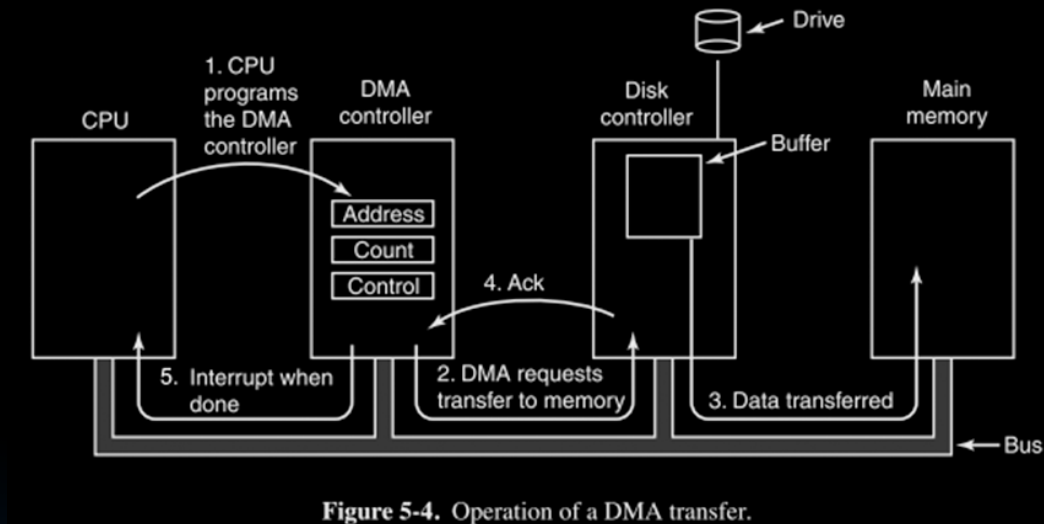
*Kanev et al., “Profiling a warehouse-scale computer,” ICSA 2015, (June 2015), Portland, OR.

DMA

Direct Memory Access (DMA)

- Allows I/O devices to directly read/write main memory
- New hardware: The DMA Engine
- DMA engine contains registers written by CPU:
 - Memory address to place data
 - # of bytes
 - I/O device #, direction of transfer
 - unit of transfer, amount to transfer per burst

DMA Illustration



From Section 5.1.4 Direct Memory Access in *Modern Operating Systems* by Andrew S. Tanenbaum, Herbert Bos, 2014

DMA: Incoming Data

1. Receive interrupt from device
2. CPU takes interrupt, initiates transfer
 - Instructs DMA engine/device to place data @ certain address
3. Device/DMA engine handle the transfer
 - CPU is free to execute other things
4. Upon completion, Device/DMA engine interrupt the CPU again

DMA: Outgoing Data

1. CPU decides to initiate transfer, confirms that external device is ready
2. CPU begins transfer
 - Instructs DMA engine/device that data is available @ certain address
3. Device/DMA engine handle the transfer
 - CPU is free to execute other things
4. Device/DMA engine interrupt the CPU again to signal completion

DMA: Some New Problems

- Where in the memory hierarchy do we plug in the DMA engine? Two extremes:
 - Between L1\$ and CPU:
 - Pro: Free coherency
 - Con: Trash the CPU's working set with transferred data
 - Between Last-level cache and main memory:
 - Pro: Don't mess with caches
 - Con: Need to explicitly manage coherency

Networking

Networks: Talking to the Outside World

- Originally sharing I/O devices between computers
 - E.g., printers
- Then communicating between computers
 - E.g., file transfer protocol
- Then communicating between people
 - E.g., e-mail
- Then communicating between networks of computers
 - E.g., file sharing, www, ...

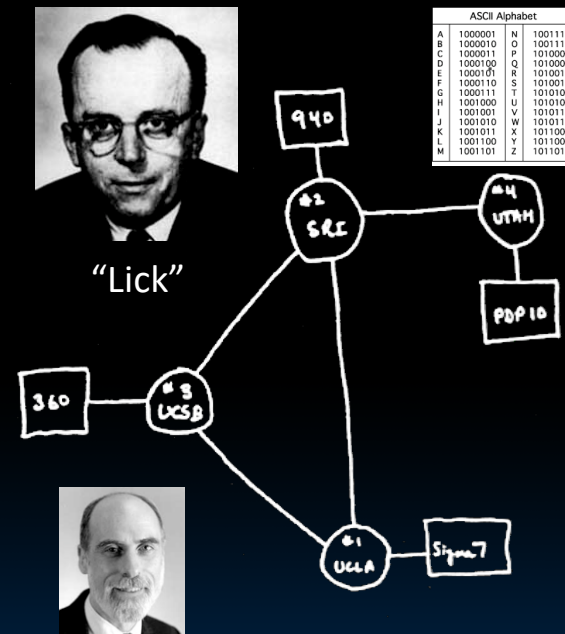
The Internet (1962)

www.computerhistory.org/internet_history

- History
 - 1963: JCR Licklider, while at DoD's ARPA, writes a memo describing desire to connect the computers at various research universities: Stanford, Berkeley, UCLA, ...
 - 1969 : ARPA deploys 4 "nodes" @ UCLA, SRI, Utah, & UCSB
 - 1973 Robert Kahn & Vint Cerf invent TCP, now part of the Internet Protocol Suite
- Internet growth rates
 - Exponential since start!



"Lick"



Vint Cerf

www.greatachievements.org/?id=3736

en.wikipedia.org/wiki/Internet_Protocol_Suite

The World Wide Web (1989)

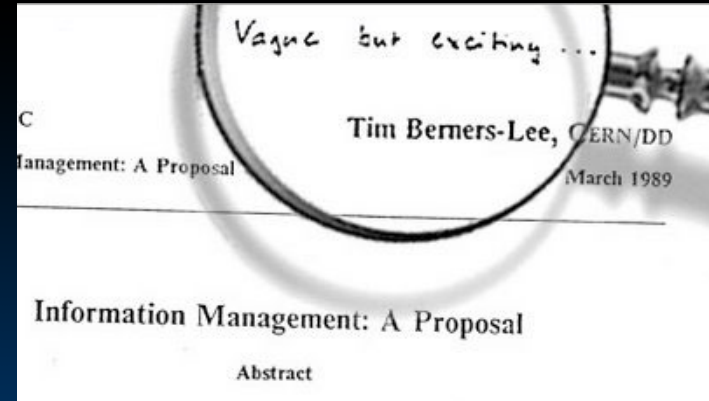
- “System of interlinked hypertext documents on the Internet”
- History
 - 1945: Vannevar Bush describes hypertext system called “memex” in article
 - 1989: Sir Tim Berners-Lee proposed and implemented the first successful communication between a Hypertext Transfer Protocol (HTTP) client and server using the internet.
 - ~2000 Dot-com entrepreneurs rushed in, 2001 bubble burst
- Today : Access anywhere!



Tim Berners-Lee



World's First web server in 1990



Software Protocol to Send and Receive

- SW Send steps
 - 1: Application copies data to OS buffer
 - 2: OS calculates checksum, starts timer
 - 3: OS sends data to network interface HW and says start
- SW Receive steps
 - 3: OS copies data from network interface HW to OS buffer
 - 2: OS calculates checksum, if OK, send ACK; if not, delete message (sender resends when timer expires)
 - 1: If OK, OS copies data to user address space, & signals application to continue



What do we need?

- Traditionally, a Network Interface Card (NIC)



- Wired or wireless
- Transfers data by using programmed I/O (old) or DMA (new)

“And In conclusion...”

- We have figured out how computers work!
 - And figured out how the OS works and how to interact with it
- We have built a virtual memory system
 - And have developed understanding of physical memory, storage devices
- And we can attach peripherals for I/O!