Note 20: IO

Kevin Moy

Abstract

Computers are far more than the static memory and CPU we discussed earlier. In reality, they are very complex systems that people interact with on a daily basis. Allowing computers to communicate- both with people and with other computers- allow the complex systems we say today.

1 Introduction + Review

Previously, we built up the idea of the OS and Virtual memory: basically increasing the complexity of our computer. Computers need to be able to handle thousands of processes **running in parallel**. Virtual memory, along with the OS, was the means for a computer to allocate and manage resources such that these processes *could* run. The page table maps virtual addresses to physical, and the TLB serves as a cache for the page table.

Let's generalize a bit more, however, and take a look at I/O: Input and Output.

2 I/O

Computers are, of course, far more complex than the topics we've discussed so far in this course. Let's now hit upon input-output devices of the computer.

Some popular **input** devices on a computer:

- Keyboard + Mouse (USB)
- Touch Screen (before you touch it)

- Audio Input (Mic)
- Camera
- Motion
- Fingerprint sensor

Some popular **output** devices on a computer:

- Display (most obvious)
- Audio Output (Speakers)
- Printers
- LEDs

So computers take in data from some input source. It can come from program code stored in RAM. However, modern general-purpose applications take some sort of input, execute some logic with that input, then output something based on the specific input.

We can actually generalize the notion of computer inputs and outputs into what we call a **cyber-physical system**. The "physical" aspect means we are simply grabbing and outputting data from the real world. Our computers are certainly cyber-physical devices! Another great example of a cyber-physical system is a car: it contains an internal computer, is controlled by input (gas pedal, steering wheel, cameras), and has output (going forward, braking, changing direction, etc). Cyber-physical systems dominate our lives especially in the modern IoT atmosphere.

So that's the highest-of-high level idea of a computer. Imagine a laptop with no screen, no keyboard/touchpad, no WiFi/Ethernet, no USB. Would be a piece of shit useless computer. Would be like a car without wheels- still interesting and complex, sure, but practically just a piece of junk!

Let's look at some common devices we are familiar with today:

Device	Behavior	Partner	Data Rate (Kb/s)
Keyboard	Input	Human	0.2
Mouse	Input	Human	0.4
Microphone	Input	Human	700.0
Bluetooth	Input or Output	Machine	20,000.0
Hard disk drive	Storage	Machine	100,000.0
Wireless network	Input or Output	Machine	300,000.0
Solid state drive	Storage	Machine	500,000.0
Wired LAN network	Input or Output	Machine	1,000,000.0
Graphics display	Output	Human	3,000,000.0

How much data are all of these devices actually providing to our machines? Notice around *eight orders of magnitude* of data transfer speeds are covered by modern IO devices today. Devices with the slowest DT speeds are ones controlled by **human** input- mice and keyboards. But as we build up to microphones and then to Internet, more and more data transfer occurs because the I/O mechanisms and logic get more and more complex. Notice the graphics display has an incredible DT rate- 3 GB *per second*- resulting in the ultra-high-def videos you can watch today. It's also why downloading movies takes up so much memory. It's also why we have a *graphics card*- an entirely separate unit of memory *solely* for graphics display on your computer.

3 Hard Drives

Now we talk about **hard drives**- generally, disks- one of the most important components of our computer.

What's up with these hard drives and mechanical drives? Do these disks function like a memory unit (RAM) or an IO device?

We can actually treat disks as IO devices! Even though we can view disks as storage, at least for our processor, it's only real memory source is RAM. Our processor outputs onto the disk when we run out of space (replacement) and takes inputs from the disk when we need more pages. Thus disk can be treated as IO- and a backup at that! We can actually just use main memory completely for the execution of a program if it's small enough.

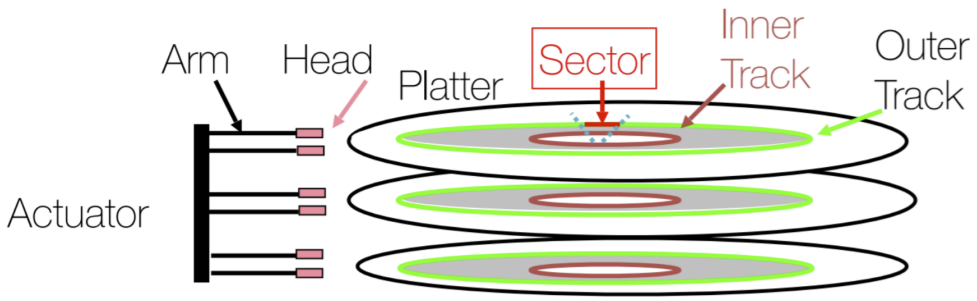
3.1 Disks

So we'll discuss disks in the context of an IO device rather than storage.

Disks, at its core, are part of the **nonvolatile storage** group - meaning it retains its data *without power*. Such storage units are also supposed to be quite large AND quite cheap (bottom of memory hierarchy)!

There are two types of disks: magnetic drives and flash drives. Magnetic drives are somewhat old fashioned and encompass mechanical drives and floppy disks- shit that really isn't used that much anymore. Flash drives, on the other hand- are more modern and useful- solid state drives (SSDs) and thumb drives.

Let's go into the science of both types of disks real quick. Magnetic disks utilize magnetism of ferrite material on rotating disks - which is somehow translated to binary. The **actuator** controls the **arm**'s movement. At the end of the arm is the **head** which is the medium for reads/writes to the disk. The **spindle** spin the disks themselves. There are a few stacked "platters" in a hard drive.



Above is an example of a 3-platter disk. Notice the actuators have one arm per disk. We split up each of these disks into two sections: **track**, which is recorded by some radius on the disk, and **sector**, recorded by some angle on a specific track. The arm decides track, and spindle specifies sector. Bits are recorded in tracks which are then subdivided into sectors. This concentric circle model is generally how memory storage works on magnetic disks.

To actually perform reads and writes on this disk, the actuator moves the head over the correct track (**seek** the track), then the spindle will rotate to give us the correct section **rotation**). Then we transfer the information.

So **disk access time = seek time + rotation time + transfer time + controller overhead**. This is the time it takes, on average, for disk to access some piece of information.

There are also averages for each of the factors that make up disk access time.

3.1.1 Average Rotation Time

The average rotation time corresponds to the average distance of a sector from the head- this would be 1/2 a rotation (symmetry argument). For example if my disk is going at 7200 revolutions/min, or 120 rev/s, then 1 revolution takes 8.33 ms, so the average time for half a rotation (and thus average rotation time) is about 4.17 ms. On a standard processor, this is equivalent to executing 16 million instructions- really bad.

3.1.2 Average Seek Time

On average, how many tracks do I need to process before I find the one I want? It's approximately the total number of tracks / 3. Thus the average seek time = (number of tracks / 3) * (time to move across one track).

3.2 Faster Hard Drives

So magnetic disks are really slow. But there are some improvements to be made. Some hard drives have *on-disk caches*, hidden from the rest of the computer. It forms the medium between RAM and disk- same as L1-L3 caches formed the medium between the CPU and RAM.

3.3 HDD Difficulties

Hopefully by now you should understand the functionality and physics of a hard drive, as well as how long it takes to work. The main allure of the hard drive is just that's it's super cheap- and can store a lot FOR cheap. However, there are some definite problems with hard disks- the main issue, being, of course, the incredible slowness (relative to today's storage units). There's also the problem with **head crashes**, where the head actually can crash into the disk itself and damage it. It's also quite difficult to fix. The arm has to be *incredibly* close to the disk.

Another issue has to do with **fragmentation**. Remember disk holds your resources and files- and there isn't any real good way of organizing these files in disk. So if files aren't stored contiguously, fragmentation may naturally occur. This can make the disk even more slow, because now we have to rotate so much more even to access a single file (since we must circle through all the fragments "before" it). Thankfully, there is a way to "defrag" our hard drive- which will make things a bit faster.

In summary, while hard drives are super cheap and can store a ton of info, they are quite slow and have a lot of overhead issues like head crashes and fragmentation. However, they are still in wide use today- they are the most cost-effective solution for storing a lot of data!

4 Solid State Drives

Solid state drives are much more modern. As more components were made faster and cheaper (Moore's law), solid state flash memory was seen as a viable large-scale storage option. Around ten years ago, microdrives and flash memory competed for the title of best large-scale storage option. Both were non-volatile (power didn't matter to retain saved info). The benefits of flash was it didn't need as much power, and minimized physical issues like crashing, since there were no moving parts, just transistors. Disk, on the other hand, had a fixed (far cheaper) cost.

So flash memory is faster and more reliable. However, it also has its issues, in particular with its **limited number of program-erase cycles**: reading and writing from flash memory causes a lot of wear and tear to the hardware over time. So after about 10K-100K writes, we can't really use flash memory anymore! A solution is **wear leveling**: moving frequently written data around, maximizing usage of all available transistors.

Pretty much all smartphones today use a ton of flash memory.

In general, solid state drives are super small and super fast. Applications that one utilizes a lot (Sublime Text, Word, NBA 2K) should be put on SSD because the speed is massively increased.

The table below compares HDD to (flash-based) SSD:

	HDD	Flash-based SSD
Durability and Noise	Loud and susceptible to shock	No moving parts!
Access Time	~ 12 ms ≈ 30M clock cycles	~ 0.1 ms ≈ 250K clock cycles
Relative Power	1	1/3
Cost	~ $0.017 / GB	~ $0.099 / GB
Capacity	500GB - 4TB	128GB - 500GB
Other Problems	Fragmentation	Limited Writes
Lifespan	5-10 years	3-5 years

Notice the advantages HDD has over SSD: cost, capacity, and lifespan. However, also notice that SSD wins in access time, power draw, durability, and most importantly SPEED!

Like always, the choice between a hard drive and SSD depends on the circumstances. If you care about storing LOTS of information, hard drive. For speed and parallel execution (and you have money to spare), SSD.

5 Memory-Mapped IO and Polling

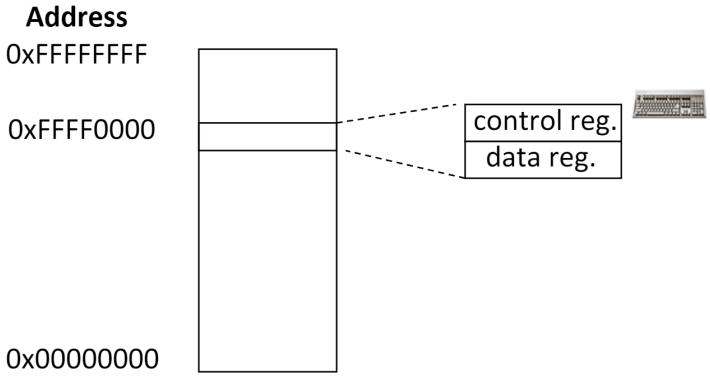
Let's move back to discussing general IO devices. We want to design a framework for a system to handle any generic IO device. There are some goals/requirements we want to set:

1. Way to connect many types of devices.
2. Way to control these devices, respond to them, transfer data to them .
3. A user-program interface.

Input to IO devices comes in the form of the device **reading bytes**, and output is logically **writing bytes** to devices. Some (C)ISC architectures have specialized IO instructions purely for this purpose. For this class, however, we will utilize RISC-V: using `lw` for loads (input) and `sw` for stores (output).

However, those require an input memory address: we can use **memory-**

mapped IO (MMIO) that basically maps IO devices to memory addresses! The below diagram shows the process:



Notice that the block of memory at `0xFFFF0000` has been allocated purely to serve as a communication channel between the processor and IO device. This cannot be used for regular memory purposes: they are ONLY to store *registers* for a single IO device. We can assume the OS will know what these memory chunks are for, so they won't interfere with virtual memory translations.

Notice the control register and data register. Control registers hold control values that indicate the IO device "ready" state(s), and the data register holds inputs/outputs corresponding to the IO device. Usually both input AND output have their own pair of control+data registers.

5.1 Processor-IO Speed Mismatch

However, a big problem with MMIO is we need to *synchronize* different IO devices because we will inherently have IO devices with different speeds-specifically, different data transfer rates.

For example, say I have a 1 GHz microprocessor, which can execute 1B load/store instructions per second (32M KB/s). Of course, not every IO device will match this data rate. This will definitely cause problems with communication between processor and IO! **The IO device might not be ready to send OR accept data as fast as the processor loads/stores it.** For example, if I want some input from an IO device but my processor is going crackhead speed, information simply won't be ready for latching from

the IO device (remember setup and hold time!).

There are a couple of solutions for this!

The first is a process called **polling**. What if the processor constantly reads the control register to see if the IO device is ready? Specifically, it would check (*poll*) if the device has control register set to 1 (called the *ready bit*). Once the device is ready, the processor knows it can load (read) or store (write) the CORRECT data from the data register. Once this read/write is one, the processor resets the control register to 0, and the loop continues.

There are a couple of ways to implement polling as well. The first is loop waiting- basically just looping through some RISC-V code until we get our ready bit 1. We use this if we know *nothing* about our IO devices. However, if we do know the IO device's **transfer frequency**, we can poll at that frequency! This will allow us to run other processes, and just stop every specified frequency interval to poll.

Let's look at an example of polling in assembly- specifically, looped waiting. We read from an input device, say a keyboard, into register `a0`, and our MMIO (IO device memory address) is at `t0`. (Note the same process for writes, except `sw` instead at the end).

```
lui t0 0xffff #ffff0000 - control reg
Waitloop:
    lw t1 0(t0) # load from CR: check if ready bit set.
    andi t1 t1 0x1
    beq t1 x0 Waitloop #If not ready, loop and poll again.
    lw a0, 4(t0) # (when ready) load from data reg.
```

Notice we waste a **ton** of cycles! How costly is it to poll? There are set "poll rates" for IO devices. For example, given a 1 GHz processor that takes 400 cycles to poll once, the standards states the ideal mouse poll rate should be 30 polls per second. The hard disk transfers data in halfword chunks at 16 MB/s. We DON'T want to miss any of the hard disk data transfers!

With waiting, we simply poll too many times. For small chunks of data transferred at a high rate, polling might not be the best option! Another issue is that upon no input (to say, a mouse)- polli

1.

ng will go on forever, and really be just wasting cycles.

Perhaps there's a better way to know when to interact with IO device data.

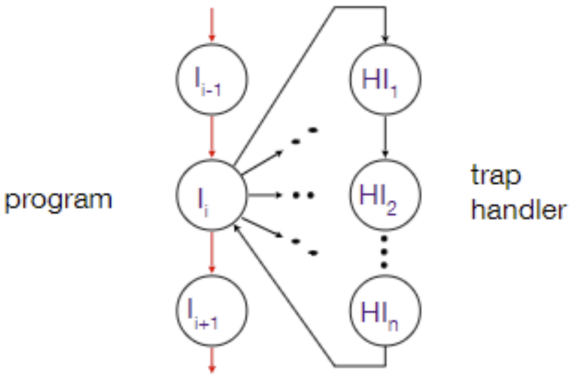
6 Interrupts

Instead of wasting time constantly checking if the IO device is ready, what if we instead **set notifications**? We call this unplanned procedure an **interrupt**- we interrupt the CPU's current process to handle the IO data. We handle interrupts very similarly to how we handle exceptions in C.

An **exception** is an unexpected event that requires a change in code flow. It arises within the CPU- for example, a segfault, illegal instruction, overflow, etc. We have to deal with it *immediately*- specifically, we synchronize our system so we handle our exception precisely on the instruction that raised it.

On the other hand, an **interrupt** that arises asynchronous to the current program- i.e. can arise whenever. We don't need to handle this interrupt immediately- we have a leeway of a few instructions.

The mechanisms used to deal with interrupts and exceptions are called **traps**. Whenever an interrupt/exception is raised, the trap handler jumps hardware to handler code. Note that dealing with interrupts and exceptions without sacrificing performance is difficult!



Above, we see that at instruction i, an interrupt/exception is raised. So we need to transfer control to the *trap handler*, execute all trap handler code,

then, depending on whether it's appropriate or not, return control back to where we left off at. The OS handles this control transfer.

How does the trap handler know to execute s.t. it doesn't affect the original program? It makes a few assumptions: every instruction up to the instruction raising the instruction has completed, AND no instructions AFTER trap have executed. The *state* of the original program needs to be saved. The trap handler should not need to understand any of the original process (code).

When handling an exception/interrupt, there are 5 main steps that occur:

1. Unexpected event
2. Save PC + registers
3. Change PC -¿ trap handler
4. Execute handler code
5. Restore registers and PC

How do we save "program state"? We have special hardware registers called **Control and Status Registers** (CSR), for the sole purpose of dealing with hardware devices. We place the PC in the "sepc" CSR, and the reason for interrupt into the "scause" CSR. When we move control to the trap handler, it looks in the "scause" SCR for the reason, since that affects what code executes. The address of the trap handler is contained in the "stvec" CSR. The "sscratch" CSR is another register that points to memory addresses to save registers and PC.

At a very low level, `csrrw rd, rs, csr` is an **atomic instruction** does all its instructions (a read and write) in a single instruction. It reads the old value of `csr` and places it into `rd`. If `rs != 0`, we place it into `csr`.

Let's see how we handle exceptions in our 5-stage pipeline. Let's say I have `fadd.s x1 x2 x1`, and it's executing (EX stage) and it hits an overflow error. We can't really assume instructions before the interrupt have completed-in fact, FOUR of them have not! Thus we must complete previous instructions completely. Then we *flush* `fadd` and subsequent instructions to ensure they DON'T get executed. This is very similar to control hazards with a mispredicted branch and flushing (undoing) all instructions we want to NOT execute.

For IO interrupts, a lot more information needs to be conveyed asynchronously. Unlike exceptions, IO interrupts are NOT associated with any instructions- it can happen at any point. Thus interrupts should be like a subroutine- we always return flow of control BACK to the caller.

Context switching, basically switching between processes, is a core concept of the trap handler, for both exceptions AND interrupts. The OS handles does this with **timed interrupts**, where it saves state and switches processes at timed intervals.

So an **interrupt-driven model** really solves the issues with polling's wasting cycles: *interrupting* our CPU ONLY when the data we need from our IO device is actually there. However, this usually results in a larger overhead.

7 Direct Memory Access

When should we use polling vs. interrupts for communication with IO devices?

For IO devices with *low* data transfer rates (doesn't give data that often), use interrupts since polling uses a lot of wasted cycles waiting. The overhead that comes with interrupts is offset by the saved time by NOT wasting cycles polling.

For IO devices with *high* data transfer rates, like Internet or disk, we should *begin* with interrupts. This way, we can *focus* the CPU on the IO device. Then, once we have all the CPU's attention directed to disk, then we switch to polling. Now, since data is coming in at a high rate, polling will now no longer waste cycles! It'll always be grabbing data each time.

However, there's still a big problem: we want to make to bring in more data PER interrupt. We might solve this with **Direct Memory Access** (DMA). We have a ton of IO devices working, and the CPU simply doesn't have time to babysit all of them. So a separate device, called the **DMA Engine**, that handles all the IO device data transfer to/from memory- and will let the CPU know when it's done. So now the CPU doesn't have to waste time on this stuff!

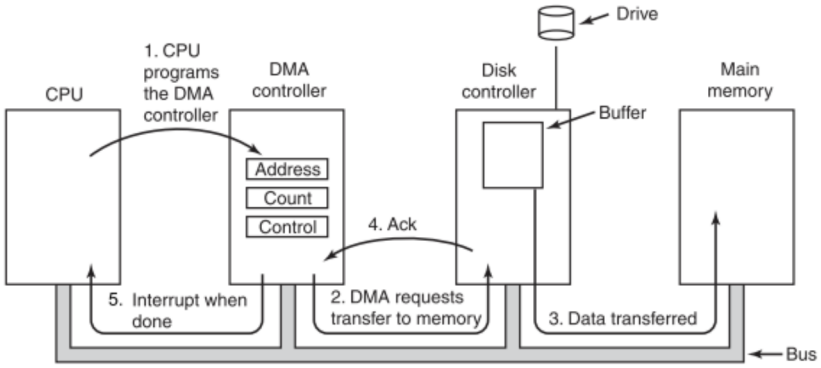
So what we've had before is **Programmed IO** (PIO)- CPU serves as a medium between disk and RAM. Now, with DMA- we just bypass the CPU

and let it access RAM directly.

An analogy is going to a restaurant and constantly pesking employees if your food is ready (polling) vs. ordering online, get a notification that food is ready, then picking it up (interrupts). But we can extend this even further: what if we didn't even have to travel to the restaurant? What if we just set up a delivery option? This is what DMA is. It's a process with a lot shorter interrupt.

So DMA allows IO devices to *directly* read/write to main memory- without going through the CPU. The DMA engine contains these hardware registers that contain parameters for a memory access by the DMA engine: memory addresses, number of bytes, IO device (number), and the unit of transfer (data transfer rate to CPU).

The below diagram shows the operation of a DMA Transfer:



The CPU first gives the DMA controller appropriate parameters (address, count, control). The DMA controller then communicates with the IO device, requesting a transfer to main memory. The device than transfers data to main memory, and then sends an ACK back to the DMA controller, saying that it's transferred the data. Only *after* all this will the DMA controller interrupt the CPU, and the CPU can grab IO device data from memory.

What are the steps involved with **incoming** data?

1. Device interrupts CPU (when it has info ready)
2. CPU begins transfer, by instructing DMA engine to place data @ address.

3. DMA Engine handles transfer, CPU can resume doing other shit
4. DMA engine interrupts CPU when transfer done.

What are the steps involved with **outgoing** data?

1. CPU confirms IO device is ready to take its load
2. CPU begins transfer by instructing DMA engine that data is to be STORED at a certain address (IO device)
3. DMA Engine stores data into address. CPU can resume doing other shit.
4. Device or DMA engine can **interrupt** the CPU when transfer done.

Remember that pretty much anything that isn't RAM on your computer is an IO device- it's certainly worth having a DMA engine to handle IO for us, saving CPU clock cycle time!

DMA engines are certainly not the perfect solution. Since DMA can actually access main memory, where does it belong on the memory hierarchy? Usually we place it between the last-level cache and main memory as a sort of separate cache. This is good- we don't mess with the caches. However, there's also a big problem: if cache and main memory has TWO different versions of data. For example, if the cache is write-back and not write-through, then memory might not have the most up-to-date information- and if DMA writes to memory, the cache might not be up-to-date either. This is called a **coherency issue**, and we solve this with cache coherency- treating DMA as basically another processor.

There's thus a sort of structural hazard between DMA and the CPU for main memory! How can we manage this? Three options:

- Burst mode: DMA engine "bursts" and hogs all of main memory- which CPU cannot access.
- Cycle stealing mode: DMA Engine and CPU take turns transferring bytes to main memory.
- Transparent Mode: DMA Engine should only work when CPU is NOT busy (not using the memory buses).

8 General-Purpose IO

A general purpose IO machine (GPIO) is a set of pins on a computer chip that can be set to HIGH (1) or LOW (0). It basically tells us if a button/switch is pressed/flipped or not.