Lecture 3 - Addressing Modes, Memory Mapped I/O, and DMA

Logistics:

- Project 1 prelim due Thurs
- office hours changed to Thurs @ 11AM (will be same for labor day week)

How do we solve the addressing issue?

- Register based offsets + a load immediate (but how does this work??)
- Relative-addressing
    - PC-relative (address computation is done relative to current PC (what does this mean for addressability in the instruction?). What type of integer should the relative address be?
    - Page-relative offsets (upper part of the address is fixed)
    - Segment-relative offsets (upper part of address lives in a register)
    - If we don't have absolute addressing, isn't that a pain for programming? Translate abs addresses (given by programmer) into PC-relative addresses (that the machine understands). This is the job of the assembler traditionally

Memory-Mapped I/O Review

- How it works: memory controller sits between CPU and peripherals (DRAM, Chipset, Disk, etc.) It translates addresses on the address bus into device requests.
- A device thus "owns" a region of the physical address space. A corollary of this is that not every physical address corresponds to RAM! If your'e an operating system (or an embedded program) don't just expect to be able to do loads and stores to random addresses!

DMA

- Why? Even with asynchronous interrupts (which notify us when a device completes or generally does something), we (the OS) still have to copy data from device buffers into RAM.
- Direct Memory Access (DMA) eliminates this step. Devices instead read/write directly from/to RAM. The CPU is then not involved at all in data transfer.
- Devices must be designed to have support for this. Device standards (like PCIe) will provide guidelines on just *how* the DMA is implemented, but the idea is that DMA will work with the same interface, independent of device. If you're curious to learn more, do a search for "PCIe bus mastering." From the OS's point of view, if it finds a PCI device that supports bus mastering, it means the device is doing DMA

- Practically, this means that device read/write requests go to the memory controller, which routes them directly to RAM, bypassing the CPU entirely (although the CPU *will* be interrupted at the end when the DMA transaction finishes)
- Some devices will have something like “write coalescing,” where the OS isn’t interrupted until a certain *number* of DMA transactions have completed. This number is usually programmable, and this feature is important for high-bandwidth devices, e.g. 10GbE NICs (Imagine a large (say 2MB) packet of data being transferred, I could raise interrupts every 1K or just wait for the one interrupt when the entire thing has been transferred.
- DMA becomes more complicated when we consider systems with virtual memory. If we give a physical address to a device, that physical address has to be valid when the transaction is actually complete. However, with typical virtual memory systems, the OS is free to remap pages at will (e.g. to swap them). The OS must ensure that this *does not happen* for pages which it uses for DMA!
- That doesn’t really matter though for chips like the 6502 which (1) don’t have virtual memory support and (2) don’t necessarily need to run with an OS.