



# CS3281 / CS5281 I/O Devices

CS3281 / CS5281  
Fall 2025

*\*Some lecture slides borrowed and adapted from  
Andrea Arpaci-Dusseau*



Tel (615) 343-7472 | Fax (615) 343-7440  
1025 16th Avenue South Nashville, TN 37212  
[www.isis.vanderbilt.edu](http://www.isis.vanderbilt.edu)



# Motivation

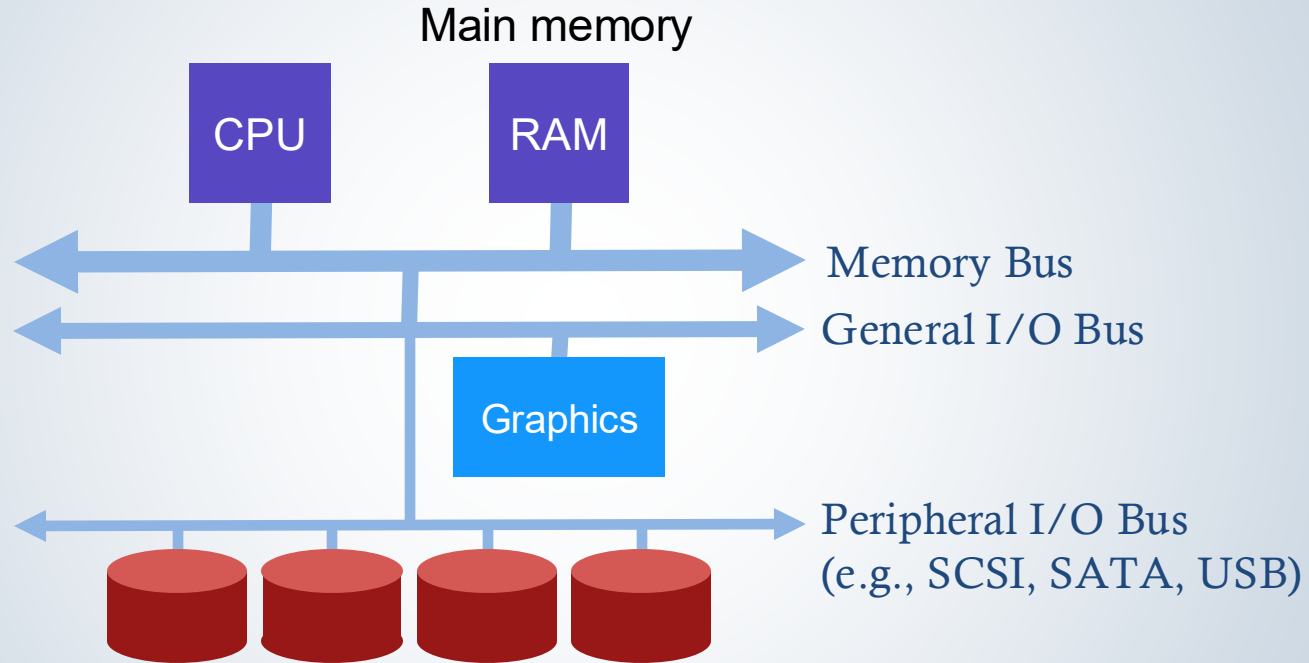
What good is a computer without any I/O devices?

- e.g., keyboard, display, disks

We want:

- **H/W** that will let us plug in different devices
- **OS** that can interact with different combinations
- I/O also allows for *persistence*
  - RAM is *volatile*, i.e., contents are lost when the machine restarts

# Hardware support for I/O

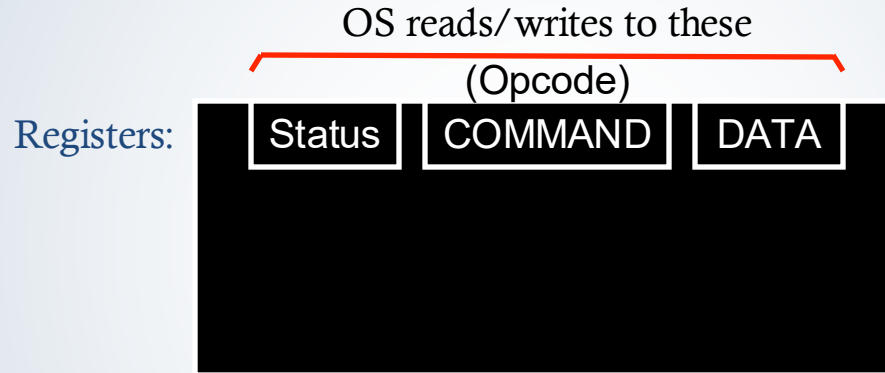


Why use hierarchical buses?

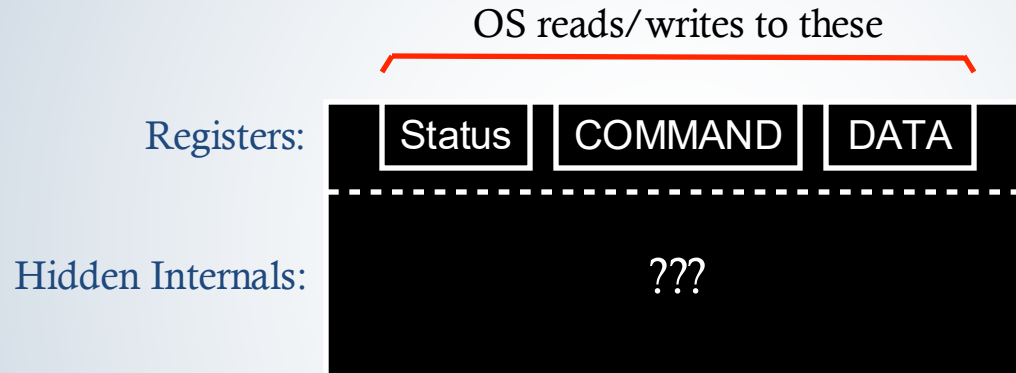
# Motivation

- Device controller is a circuit that operates (controls) the specific I/O devices
- To do the operation, the controller has registers and flags that are set and examined by the device *drivers*
- A device driver is a program that translates the user application *generic* I/O operations to interact with the device controller. The translated operations are used to control the specific device

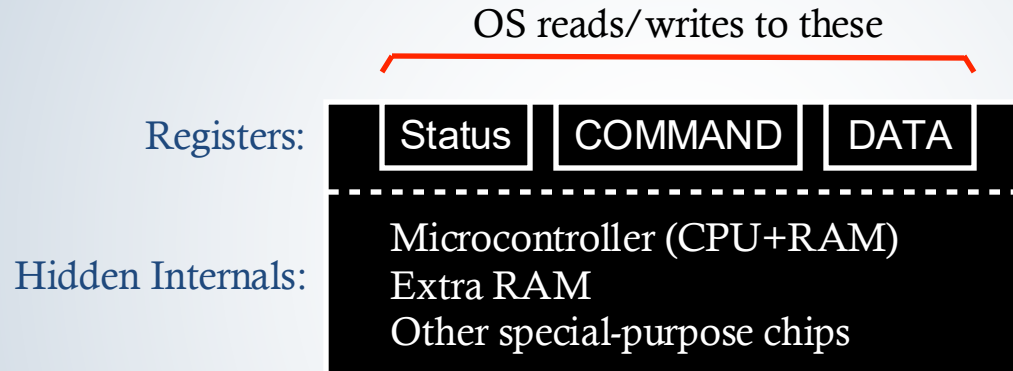
# Canonical Device Controller



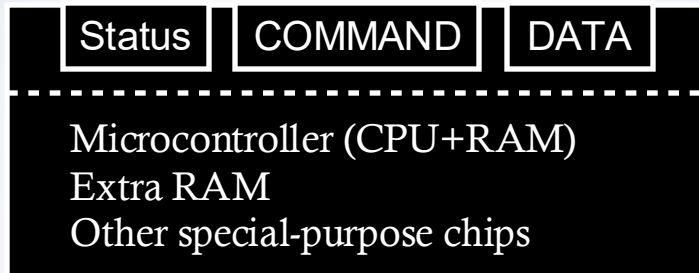
# Canonical Device Controller



# Canonical Device Controller



# Example Write Protocol



```
while (STATUS == BUSY)
    ; // spin
```

Write data to DATA register

Write command to COMMAND register

```
while (STATUS == BUSY)
    ; // spin
```

Wait for device to be free

Wait for write to finish

This is called polling when the processor keeps “asking” what the hardware is doing by reading *status* flag.

CPU:

Disk:

```
while (STATUS == BUSY)      // 1
```

```
;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```

CPU:  A

Disk:  C

```
while (STATUS == BUSY)      // 1
```

```
;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```

A wants to do I/O



CPU: A

Disk: C

```
while (STATUS == BUSY)      // 1
```

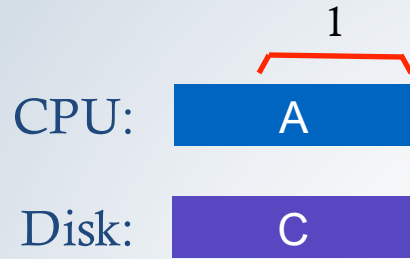
```
;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```



```
while (STATUS == BUSY)    // 1
```

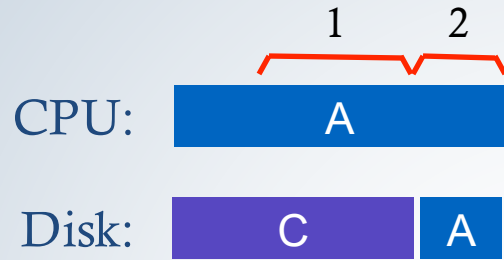
```
;
```

```
Write data to DATA register    // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)    // 4
```

```
;
```



```
while (STATUS == BUSY)      // 1
```

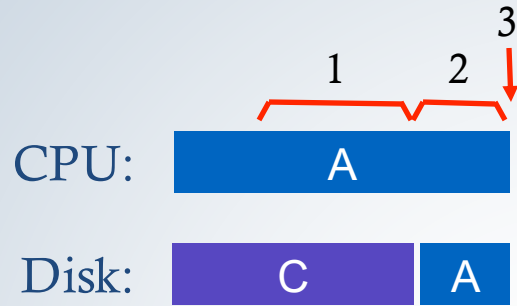
```
;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```



```
while (STATUS == BUSY)    // 1
```

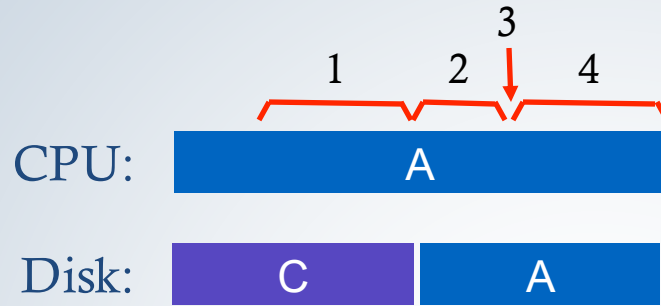
```
;
```

```
Write data to DATA register    // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)    // 4
```

```
;
```



```
while (STATUS == BUSY)      // 1
```

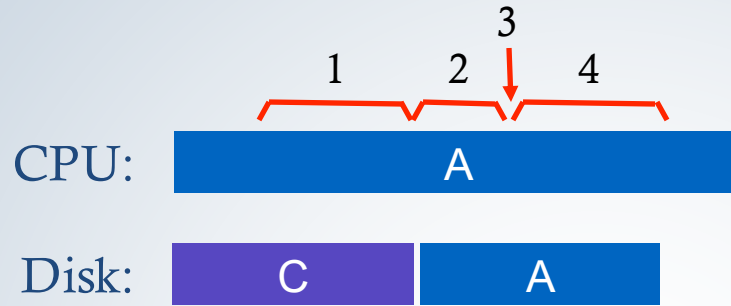
```
;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```



```
while (STATUS == BUSY)      // 1
```

```
;
```

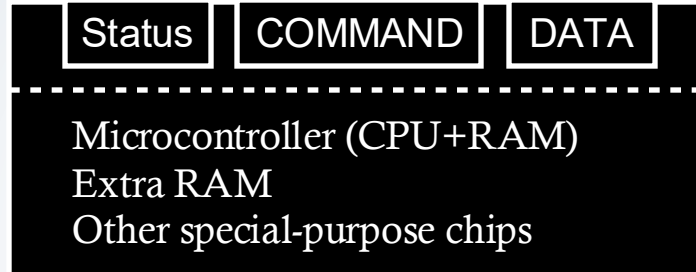
```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
;
```

# Example Read Protocol



```
while (STATUS == BUSY)
```

```
    ; // spin
```

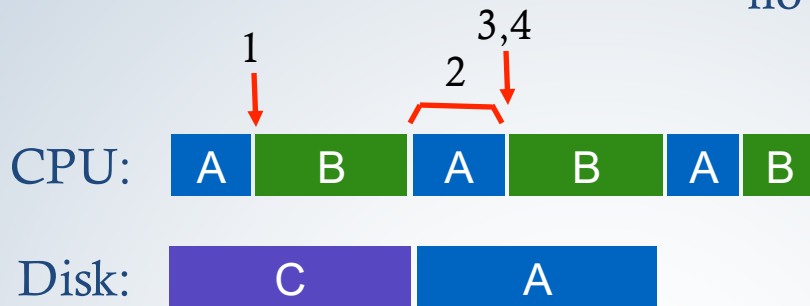
```
Write command to COMMAND register
```

```
while (STATUS == BUSY)
```

```
    ; // spin
```

```
Read data from DATA register
```

how to avoid spinning?  
interrupts!



```
while (STATUS == BUSY)      // 1
```

```
    wait for interrupt;
```

```
    Write data to DATA register    // 2
```

```
    Write command to COMMAND register // 3
```

```
    while (STATUS == BUSY)      // 4
```

```
        wait for interrupt;
```

A process blocks if a device is busy.  
When the device is not busy,  
controller issues an interrupt to CPU  
and unblocks the process

# Interrupts vs. Polling

Are interrupts ever worse than polling?

- Interrupts represent *predictable* overhead for each I/O operation. CPU time is not wasted on spinning when a device is slow
- Fast device or a single process running: Better to poll (spin) than take interrupt and context-switching overhead
- Device time unknown? Hybrid approach (spin then use interrupts)

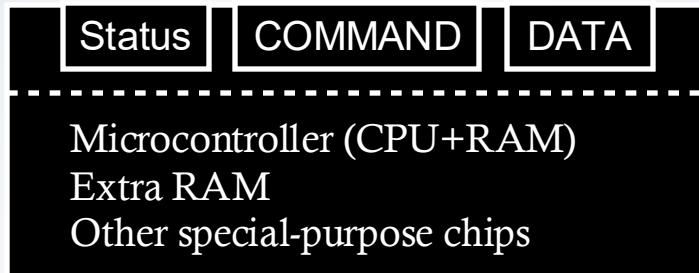
Flood of interrupts arrive for packet arrivals

- Can lead to livelock (always handling interrupts)
- Use polling for high load of arrivals and interrupts for low load of arrivals (hybrid)

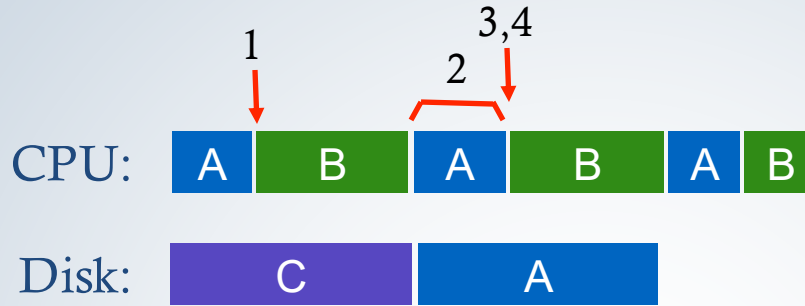
Other improvement

- Interrupt coalescing (hardware batches together several interrupts)

# Protocol Variants



- **Status checks:** polling vs. interrupts
- **Data:** programmed I/O (PIO) vs. direct memory access (DMA)
- **Control:** special instructions vs. memory-mapped I/O



```
while (STATUS == BUSY)      // 1
```

```
    wait for interrupt;
```

```
    Write data to DATA register    // 2
```

```
    Write command to COMMAND register // 3
```

```
    while (STATUS == BUSY)      // 4
```

```
        wait for interrupt;
```

what else can we optimize?

data transfer!

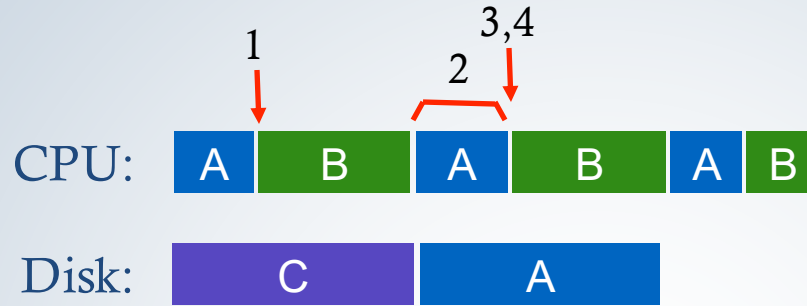
# Programmed I/O vs. Direct Memory Access

## **PIO** (Programmed I/O):

- CPU directly copies data between the device controller and main memory
- Efficient for a few bytes/words, but *scales terribly*

## **DMA** (Direct Memory Access):

- CPU leaves data in memory and just initiates an I/O operation
- Device reads/writes data directly from/to memory using the DMA device without involving CPU
- CPU is free to serve the other processes
- Efficient for large data transfers



```
while (STATUS == BUSY)      // 1
```

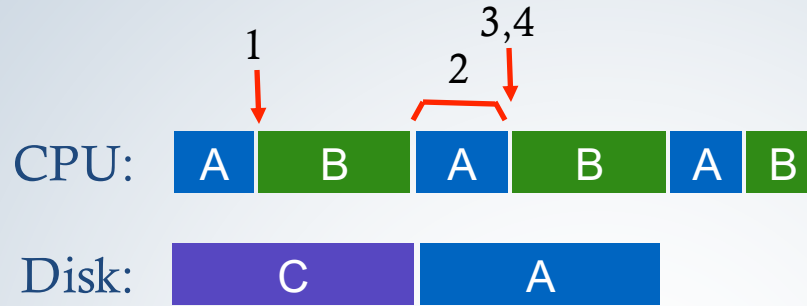
```
    wait for interrupt;
```

```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
    wait for interrupt;
```



```
while (STATUS == BUSY)      // 1
```

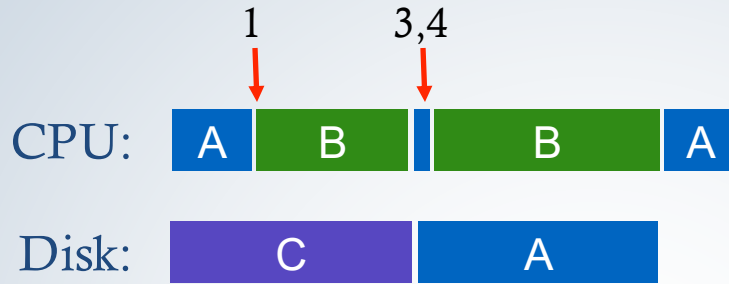
```
    wait for interrupt;
```

```
    Write data to DATA register // 2
```

```
    Write command to COMMAND register // 3
```

```
    while (STATUS == BUSY)    // 4
```

```
        wait for interrupt;
```



```
while (STATUS == BUSY)      // 1
```

```
    wait for interrupt;
```

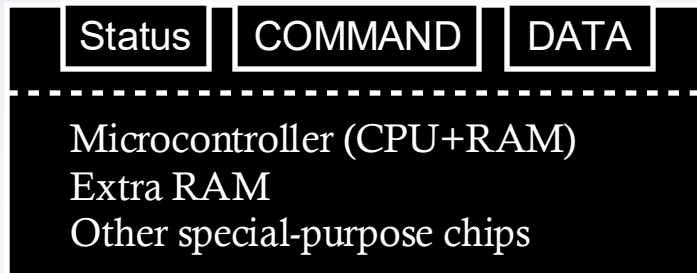
```
Write data to DATA register // 2
```

```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
    wait for interrupt; Device controller issues an interrupt to CPU to unblock A
```

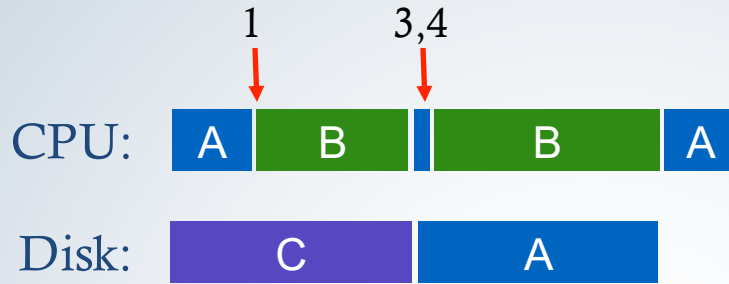
# Protocol Variants



**Status checks:** polling vs. interrupts

**Data:** PIO vs. DMA

**Control:** special instructions vs. memory-mapped I/O



```
while (STATUS == BUSY)      // 1
```

```
    wait for interrupt;
```

```
Write data to DATA register // 2
```

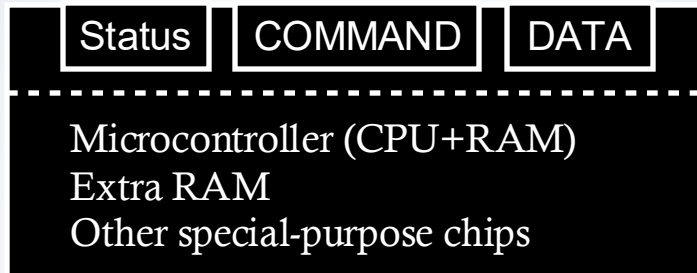
```
Write command to COMMAND register // 3
```

```
while (STATUS == BUSY)      // 4
```

```
    wait for interrupt;
```

how does OS read and write registers?

# Protocol Variants



**Status checks:** polling vs. interrupts

**Data:** PIO vs. DMA

**Control:** special instructions vs. memory-mapped I/O

# Special Instructions vs. Memory-Mapped I/O

## Special instructions

- each device has a port
- in/out instructions (x86) communicate with device

## Memory-Mapped I/O

- H/W maps registers into the virtual address space
- loads/stores sent to device

Doesn't matter much (both are used)