What Is Synchronization?

Two schemes for interacting with I/O devices

Way have seen so far is **polling**

- “Are we there yet? Are we there yet? Are we there yet?”
- CPU keeps checking status register in a loop
- Very inefficient, CPU has better things to do

Alternative scheme is called **interrupts**

- “Wake me when we get there.”
- Device sends special signal to CPU when status changes
- CPU stops current program, saves its state
- CPU “handles interrupt”: checks status, moves data
- CPU resumes stopped program, as if nothing happened

So What Is Synchronization?

Interrupt driven I/O is **asynchronous**

- Highly overloaded word, here means “takes unbounded time”

Asynchronous programming is nasty

- Suspend, find something else to do, resume on signal

Better leave this to the OS

- Suspends you transparently
- Runs another program while you wait (it knows about these)
- Resumes you transparently on signal

To you, I/O looks **synchronous**

- Similarly overloaded, here means “happens immediately”
Mechanics: Ready & Interrupt Bit

A way for I/O device to signal CPU that it's wants service
- We already have a Ready bit indicating that

A way to know that device has a right to request service
- Mainly to support/handle nested interrupts
- When ready bit is set and IE bit is set, interrupt is signaled
  I/O device will get service

Mechanics: Service Routine

Each device as a unique identifier

Interrupt vector: INTV
- Identifier is used as an index Interrupt Vector Table, which gives
  starting address of Interrupt Service Routine (ISR)
- Just like Trap Vector Table and Trap Service Routine

Mechanics: Processor State

You were in the middle of adding 2 numbers in your program, but there was interrupt
- Did I complete my operation?
- After I resume, is my data in the registers that I was storing the same as I left of?

Before FETCH stage in the instruction cycle:
- Check for Interrupt
  - If interrupt, then save to memory the state of the machine, i.e.
    Registers and PC and CCs
    - This way I can come back start executing where I left
    - But where???
    - Usually the OS will handle this detail (OS keeps a data structure)
- Load the PC with starting address of the program that is to carry out the requirements of the I/O

Nested Interrupt

Dealing with interrupts while servicing another:
1. Disable Interrupts while servicing another
   - Not good if the interrupt is of critical nature

2. Assign priorities to devices
   - If current interrupt has low priority, suspend and service new interrupt
   - Remember to save the state of the previous interrupt
Nested Interrupts

Mechanics: Priority
Whether I/O device has higher priority among multiple I/O requests AND with the current program in execution
- Every instruction executes at a stated level of urgency
  - E.g. in LC-3
    - 8 priority levels (PL0-PL7) with 7 being higher priority
    - If we were to implement this LC3/LC4 we would use PSR

LC4 Simulator Does Not Support Interrupts
Why not?
- Suppose program wants to read from the keyboard
- Interrupts make sense if CPU could do something else meantime
- Program itself can't do anything, it's waiting for the key ...
  - ... so another program has to run while it's waiting
- And LC4 doesn't support multi-programming
- So there is no real point

Discussion Question I/O
Do you think polling is a good approach for other devices, such as a disk or a network interface?