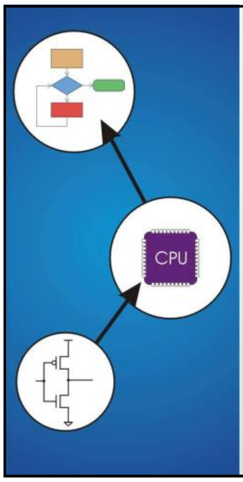
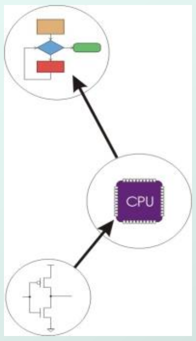


Chapter 8 I/O

Original slides from Gregory Byrd, North Carolina State University

Modified slides by Chris Wilcox, Colorado State University

Computing Layers



- Problems
- Algorithms
- Language
- Instruction Set Architecture ←
- Microarchitecture
- Circuits
- Devices

I/O: Connecting to Outside World

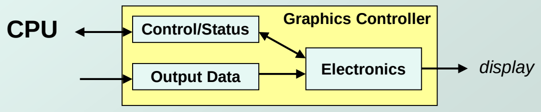
- So far, we've learned how to:
  - compute with values in registers
  - load data from memory to registers
  - store data from registers to memory
- But where does data in memory come from?
- And how does data get out of the system so that humans can use it?

I/O: Connecting to the Outside World

- Types of I/O devices characterized by:
  - behavior: input, output, storage
    - input: keyboard, motion detector, network interface
    - output: monitor, printer, network interface
    - storage: disk, CD-ROM
  - data rate: how fast can data be transferred?
    - keyboard: < 100 bytes/sec
    - disk: 30 MB/s
    - network: 1 Mb/s - 1 Gb/s

I/O Controller

- **Control/Status Registers**
  - CPU tells device what to do -- write to control register
  - CPU checks if task is done -- read status register
- **Data Registers**
  - CPU transfers data to/from device



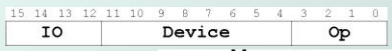
- **Device electronics**
  - performs actual operation
    - pixels to screen, bits to disk, chars from keyboard

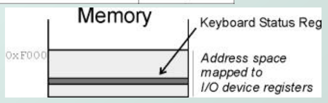
Programming Interface

- How are device registers identified?
  - **Memory-mapped** vs. **special instructions**
- How is timing of transfer managed?
  - **Asynchronous** vs. **synchronous**
- Who controls transfer?
  - CPU (**polling**) vs. device (**interrupts**)

Memory-Mapped vs. I/O Instructions

- **Instructions**
  - designate opcode(s) for I/O
  - register and operation encoded in instruction





- **Memory-mapped**
  - assign a memory address to each device register
  - use data movement instructions (load/store) for control and data transfer

Transfer Timing

- I/O events generally happen much slower than CPU cycles.
- **Synchronous**
  - data supplied at a fixed, predictable rate
  - CPU reads/writes every X cycles
- **Asynchronous**
  - data rate less predictable
  - CPU must *synchronize* with device, so that it doesn't miss data or write too quickly

Transfer Control

- Who determines when the next data transfer occurs?
- Polling
  - CPU keeps checking status register until *new data* arrives OR *device ready* for next data
  - "Are we there yet? Are we there yet? Are we …
- Interrupts
  - Device sends a special signal to CPU when *new data* arrives OR *device ready* for next data
  - CPU can be performing other tasks instead of polling device.
  - "Wake me when we get there."

LC-3

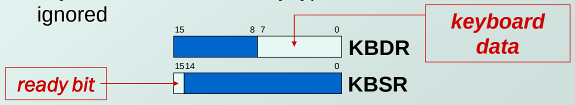
- Memory-mapped I/O (Table A.3)

Location	*I/O Register*	*Function*
`xFE00`	**Keyboard Status (KBSR)**	**Bit [15] is one when keyboard has received a new character.**
`xFE02`	**Keyboard Data (KBDR)**	**Bits [7:0] contain the last character typed on keyboard.**
`xFE04`	**Display Status (DSR)**	**Bit [15] is one when device ready to display char on screen.**
`xFE06`	**Display Data (DDR)**	**Character written to bits [7:0] will be displayed on screen.**

- Asynchronous devices
  - synchronized through status registers
- Polling and Interrupts
  - Interrupt details will be discussed in Chapter 10

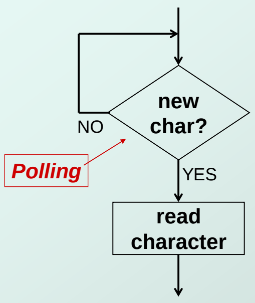
Input from Keyboard

- When a character is typed:
  - its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
  - the "ready bit" (KBSR[15]) is set to one
  - keyboard is disabled -- any typed characters will be ignored



- When KBDR is read:
  - KBSR[15] is set to zero
  - keyboard is enabled

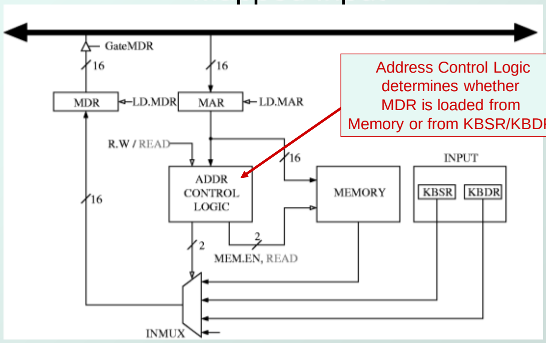
Basic Input Routine



```
POLL    LDI  R0, KBSRPtr
        BRzp POLL
        LDI  R0, KBDRPtr
        ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

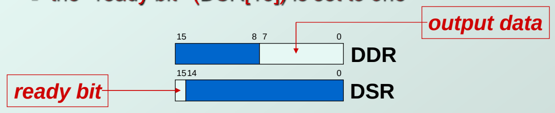
Simple Implementation: Memory-Mapped Input



Address Control Logic determines whether MDR is loaded from Memory or from KBSR/KBDR.

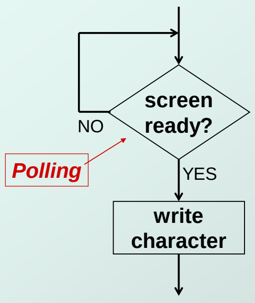
Output to Monitor

- When Monitor is ready to display another character:
  - the "ready bit" (DSR[15]) is set to one



- When data is written to Display Data Register:
  - DSR[15] is set to zero
  - character in DDR[7:0] is displayed
  - any other character data written to DDR is ignored

Basic Output Routine

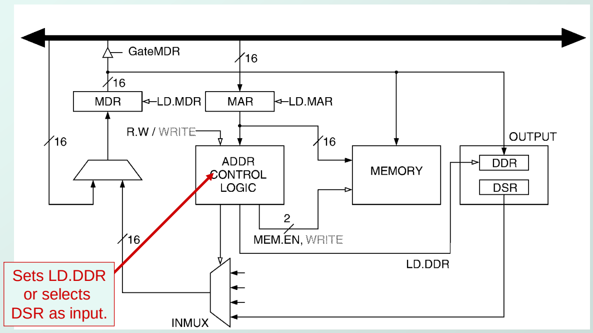


```
POLL    LDI   R1, DSRPtr
        BRzp  POLL
        STI   R0, DDRPtr

        ...

DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```

Simple Implementation: Memory-Mapped Output



Sets LD.DDR or selects DSR as input.

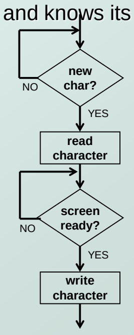
Keyboard Echo Routine

- Usually, input character is also printed to screen.
  - User gets feedback on character typed and knows its ok to type the next character.

```
POLL1    LDI   R0, KBSRPtr
         BRzp  POLL1
         LDI   R0, KBDRPtr
POLL2    LDI   R1, DSRPtr
         BRzp  POLL2
         STI   R0, DDRPtr

         ...

KBSRPtr  .FILL xFE00
KBDRPtr  .FILL xFE02
DSRPtr   .FILL xFE04
DDRPtr   .FILL xFE06
```



Interrupt-Driven I/O

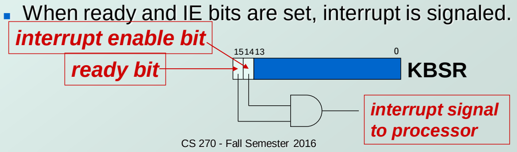
- External device can:

(1) Force currently executing program to stop.
(2) Have the processor satisfy the device needs.
(3) Resume the program as if nothing happened.

- Why?
  - Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
  - I/O device is faster than the CPU.
  - Example: Process previous input while collecting current input. (See Example 8.1 in text.)

Interrupt-Driven I/O

- To implement an interrupt mechanism, we need:
  - A way for the I/O device to signal the CPU that an interesting event has occurred.
  - A way for the CPU to test if the interrupt signal is set and if its priority is higher than current program.
- Generating Signal
  - Software sets "interrupt enable" bit in device register.
  - When ready and IE bits are set, interrupt is signaled.

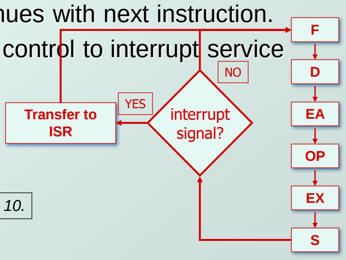


Priority

- Every instruction executes at a stated level of urgency.
- LC-3: 8 priority levels (PL0-PL7)
  - Example:
    - Payroll program runs at PL0.
    - Nuclear power correction program runs at PL6.
  - It's OK for PL6 device to interrupt PL0 program, but not the other way around.
- Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.

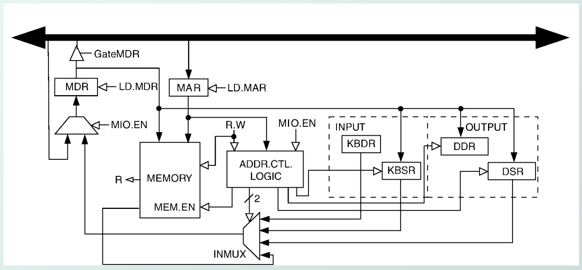
Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases.
- If not set, continues with next instruction.
- If set, transfers control to interrupt service routine.



More details in Chapter 10.

Full Implementation of LC-3 Memory-Mapped I/O



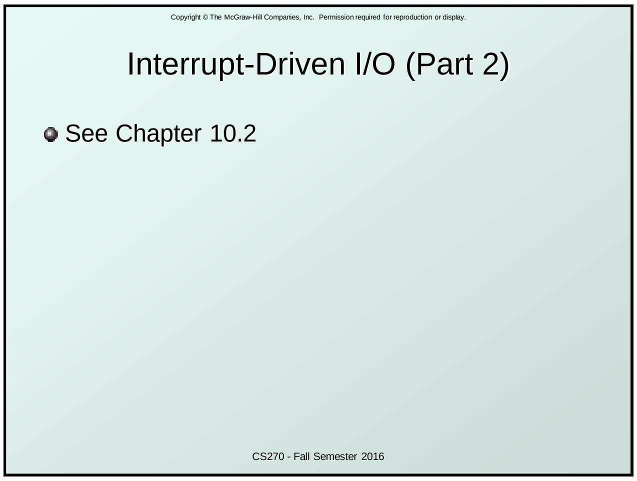
Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.

Review Questions

- What is the danger of not testing the DSR before writing data to the screen?
- What is the danger of not testing the KBSR before reading data from the keyboard?
- What if the Monitor were a synchronous device, e.g., we know that it will be ready 1 microsecond after character is written.
  - Can we avoid polling? How?
  - What are advantages and disadvantages?

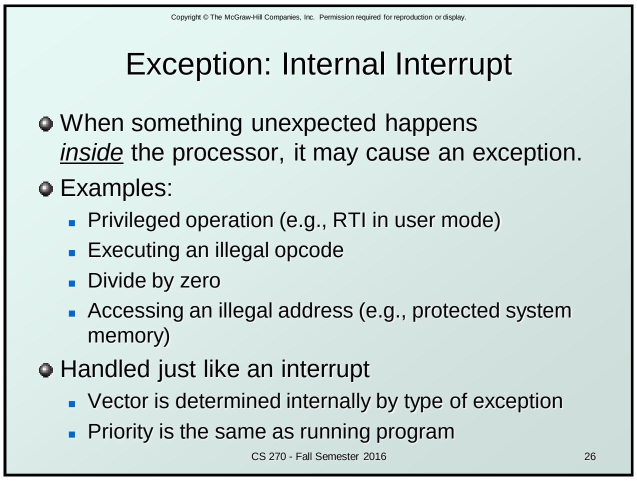
Review Questions

- Do you think polling is a good approach for other devices, such as a disk or a network interface?
- What is the advantage of using LDI/STI for accessing device registers?



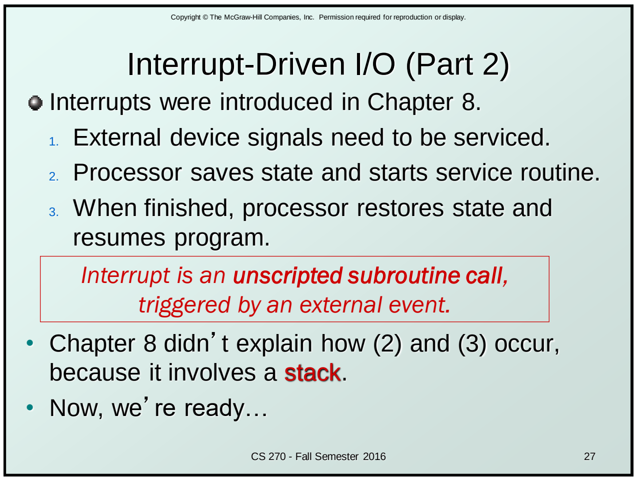
Interrupt-Driven I/O (Part 2)

- See Chapter 10.2



Exception: Internal Interrupt

- When something unexpected happens *inside* the processor, it may cause an exception.
- Examples:
  - Privileged operation (e.g., RTI in user mode)
  - Executing an illegal opcode
  - Divide by zero
  - Accessing an illegal address (e.g., protected system memory)
- Handled just like an interrupt
  - Vector is determined internally by type of exception
  - Priority is the same as running program

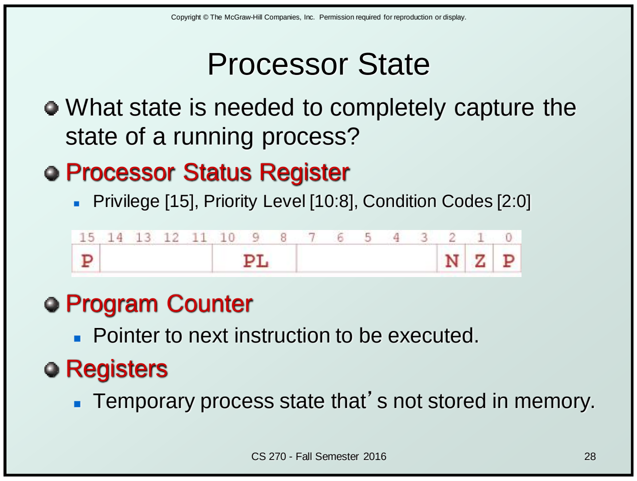


Interrupt-Driven I/O (Part 2)

- Interrupts were introduced in Chapter 8.
  1. External device signals need to be serviced.
  2. Processor saves state and starts service routine.
  3. When finished, processor restores state and resumes program.

*Interrupt is an **unscripted subroutine call**, triggered by an external event.*

- Chapter 8 didn't explain how (2) and (3) occur, because it involves a **stack**.
- Now, we're ready…



Processor State

- What state is needed to completely capture the state of a running process?
- **Processor Status Register**
  - Privilege [15], Priority Level [10:8], Condition Codes [2:0]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P						PL							N	Z	P

- **Program Counter**
  - Pointer to next instruction to be executed.
- **Registers**
  - Temporary process state that's not stored in memory.

Where to Save Processor State?

- Can't use registers.
  - Programmer doesn't know when interrupt might occur, so she can't prepare by saving critical registers.
  - When resuming, need to restore state exactly as it was.
- Memory allocated by service routine?
  - Must save state before invoking routine, so we wouldn't know where.
  - Also, interrupts may be nested – that is, an interrupt service routine might also get interrupted!
- **Use a stack!**
  - Location of stack "hard-wired".
  - Push state to save, pop to restore.

Supervisor Stack

- A special region of memory used as the stack for interrupt service routines.
  - Initial Supervisor Stack Pointer (SSP) stored in Saved.SSP.
  - Another register for storing User Stack Pointer (USP): Saved.USP.
- Want to use R6 as stack pointer.
  - So that our PUSH/POP routines still work.
- When switching from User mode to Supervisor mode (as result of interrupt), save R6 to Saved.USP.

Invoking the Service Routine (Details)

1. If Priv = 1 (user),
   Saved.USP = R6, then R6 = Saved.SSP.
2. Push PSR and PC to Supervisor Stack.
3. Set PSR[15] = 0 (supervisor mode).
4. Set PSR[10:8] = priority of interrupt being serviced.
5. Set PSR[2:0] = 0.
6. Set MAR = x01vv, where vv = 8-bit interrupt vector provided by interrupting device (e.g., keyboard = x80).
7. Load memory location (M[x01vv]) into MDR.
8. Set PC = MDR; now first instruction of ISR will be fetched.

Note: This all happens between the STORE RESULT of the last user instruction and the FETCH of the first ISR instruction.

Returning from Interrupt

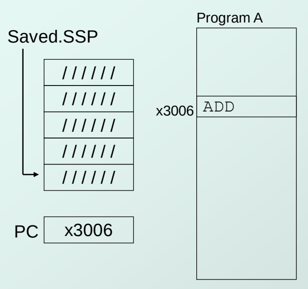
- Special instruction – RTI – that restores state.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1. Pop PC from supervisor stack:
   (PC = M[R6]; R6 = R6 + 1)
2. Pop PSR from supervisor stack:
   (PSR = M[R6]; R6 = R6 + 1)
3. If going back to user mode, need to restore User Stack Pointer:
   (if PSR[15] = 1, R6 = Saved.USP)

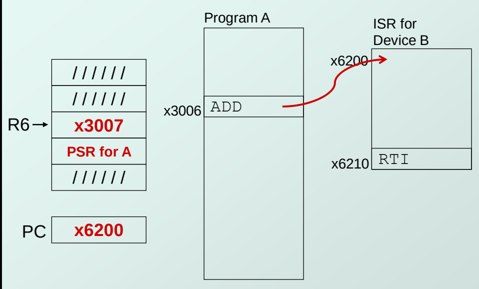
- RTI is a privileged instruction.
  - Can only be executed in Supervisor Mode.
  - If executed in User Mode, causes an exception. (More about that later.)

Example (1)



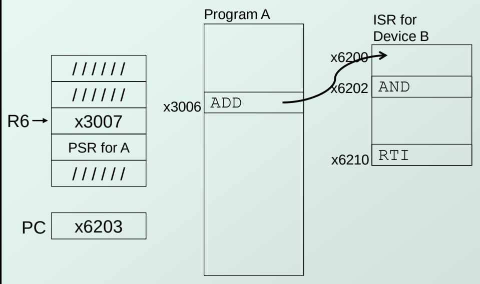
Executing ADD at location x3006 when Device B interrupts.

Example (2)



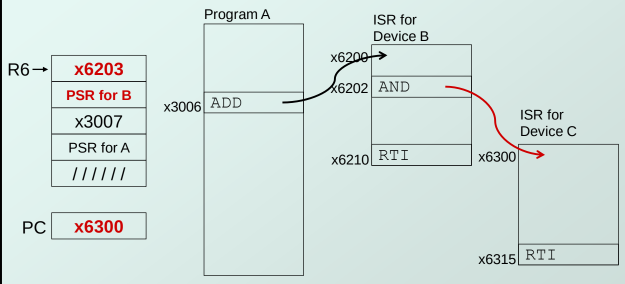
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to Device B service routine (at x6200).

Example (3)

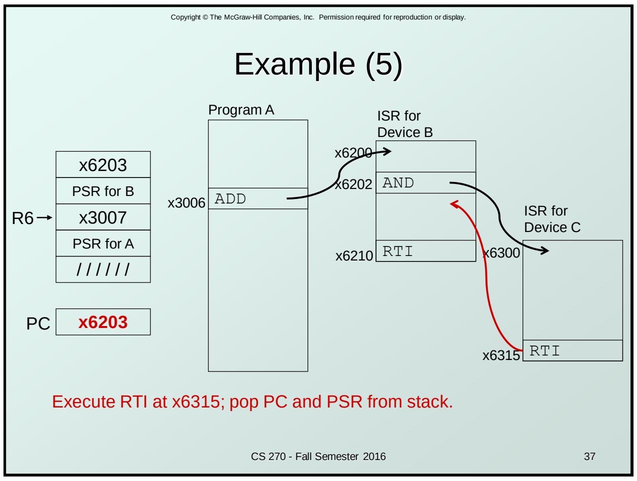


Executing AND at x6202 when Device C interrupts.

Example (4)

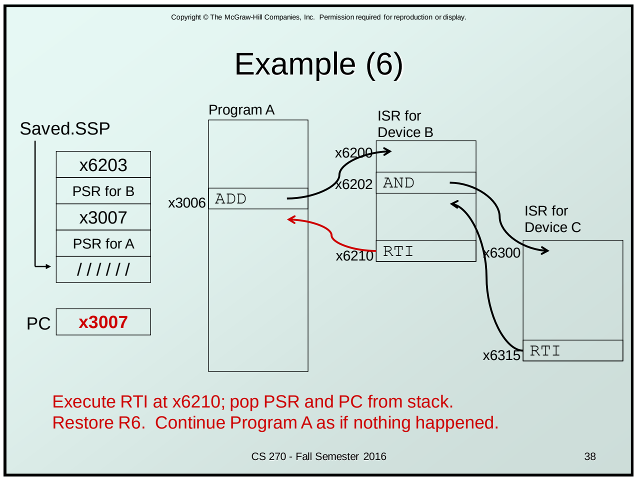


Push PSR and PC onto stack, then transfer to Device C service routine (at x6300).



Example (5)

Execute RTI at x6315; pop PC and PSR from stack.



Example (6)

Execute RTI at x6210; pop PSR and PC from stack.
Restore R6. Continue Program A as if nothing happened.