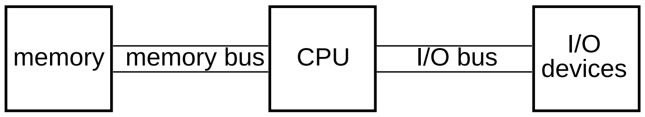
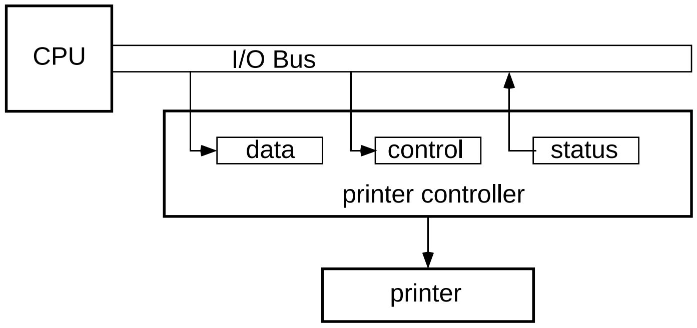
I/O Devices (Review)

- A typical computer system is organized as follows:



- The I/O (input / output) devices are accessed through registers in much the same manner as the memory is accessed



Communicating With a Keyboard (Review)

- A particular keyboard communicates with the CPU as follows:
  - A status register is addressed by the CPU at memory location 200. Bit 7 (little endian addressing) is used to indicate that a character is available. If so, that bit is 1; otherwise, it is 0. Other bits serve other purposes.
  - A data register, addressed by the CPU at memory location 201, stores the character typed on the keyboard.
- Assembly code:

```
top:    LOAD  R0,200      ; read status reg.
        AND   R0,R0,#80H  ; check bit 7
        BZ top            ; loop until it's 1
        LOAD  R9,201      ; read data
        JUMP  top         ; loop forever
```

Programmed I/O Using UARTs

- In *programmed I/O*, the program (CPU) is in direct control of the I/O
- The actual hardware element responsible for the I/O is a *UART* — a Universal Asynchronous Receiver / Transmitter
  - Can both transmit and receive
  - Transfers characters in serial fashion between a CPU and an external I/O device (e.g., keyboard, printer, modem)
  - Uses 4 (memory-mapped) registers:
    - Control register — specifies the communication method (bits to set transmission speed, parity, etc.)
    - Status register — maintains status of UART (transmit register full, receive register full, various errors, etc.)
    - Transmit register — holds next character to send
    - Receive register — holds last character received

Reception using a UART

- Status register (STATUS)
  - Bit 1 = 1   receive register (RCV) full
  - Bit 1 = 0   receive register (RCV) not full
- Internal operation of the UART
  - Bit 1 of status register initially set to 0
  - When new value is received
    - Value is stored in receive register
    - Bit 1 is set to 1
  - When value is read from receive register
    - Bit 1 is set to 0
- Program receiving characters
  - Example receiving 8-bit value into R3

```
wait_rcv:   LOAD.b   R2,STATUS
            AND      R2,R2,#2
            BRZ      wait_rcv
            LOAD.b   R3,RCV
```

Transmission using a UART

- Status register (STATUS)
  - Bit 0 = 1   transmit register (XMIT) full
  - Bit 0 = 0   transmit register (XMIT) not full
- Internal operation of the UART
  - Bit 0 of status register initially set to 0
  - When new value appears in transmit register
    - Bit 0 is set to 1
    - Value is transmitted
    - Bit 0 is set to 0
- Program transmitting characters
  - Example sending 8-bit value in R3

```
wait_snd:   LOAD.b    R2,STATUS
            AND       R2,R2,#1
            BRNZ      wait_snd
            STORE     XMIT,R3
```

Interrupts

- An *interrupt* is a signal generated by a device when it needs the CPU's attention
  - If a UART supports interrupts:
    - A UART can generate an interrupt whenever it receives a character
    - Interrupts can generally be *enabled* or *disabled* under software control (through a bit in the control register)
- CPU checks for interrupts after each instruction execution
  - When an interrupt occurs:
    - Program control is transferred to the appropriate interrupt handler
    - That interrupt handler processes the interrupt, and clears the interrupt request
    - The program that was interrupted then resumes its normal execution
  - Interrupts can occur at any time!

Interrupts (cont.)

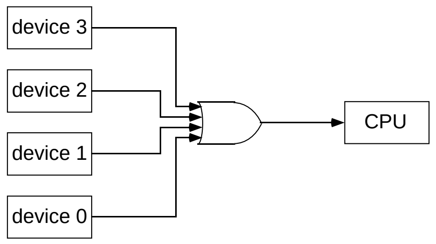
- Before control transfers to the interrupt handler, the CPU saves the PC and PSW
  - PC = Program Counter
  - PSW = Processor Status Word
    - Condition codes
    - Interrupt mask
  - Restored after interrupt handler finishes
- The interrupt handler must save and restore all the general-purpose registers
- With a UART, when the interrupt handler reads from the receive register, the interrupt gets cleared
  - What if interrupt handler gets interrupted?
  - Solution — Use special instruction to set interrupt mask bit in PSW to disable interrupts while in interrupt handler

Handling Multiple Interrupts

- It may be necessary for more than one device to interrupt the CPU
  - *Selective masking* — disable interrupts from certain selected devices
  - *Device identification* — identify the device that has an enabled and unserviced interrupt
  - *Priority scheme* — if more than once device has an enabled and unserviced interrupt, choose which one to service first
  - *Dispatching* — transfer control to the proper interrupt handling routine
- Interrupt Mask in PSW enables/disables all interrupts
  - Add Interrupt Mask Register, with enable/disable bit corresponding to each device

Handling Multiple Interrupts (cont.)

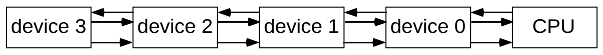
- Software polling



  - Interrupt signal to CPU is logical *or* of individual interrupt signals
  - Device identification, prioritization, and dispatching must be handled by software
    - Add an Interrupt Status Register with bit for each device
    - Software has to "poll" each bit in this ISR to see which device has an unserviced interrupt
    - Software prioritizes those interrupts, and branches to appropriate interrupt handler

Handling Multiple Interrupts (cont.)

- Daisy chaining



  - Interrupt request is sent to CPU, going through any neighbors to the right
  - CPU sends an "identify yourself" request back through the devices
  - When it receives that "identify yourself" request, the interrupting device sends its ID back to the CPU
    - It does <u>not</u> send the "identify yourself" request on to lower-priority devices
    - Devices closer to CPU have higher priority
  - CPU then services that interrupt
- Interrupt controller
  - Sits between the devices and the CPU
  - Prioritizes the requests, identifies the device, and sends that request to CPU