Recap from last class

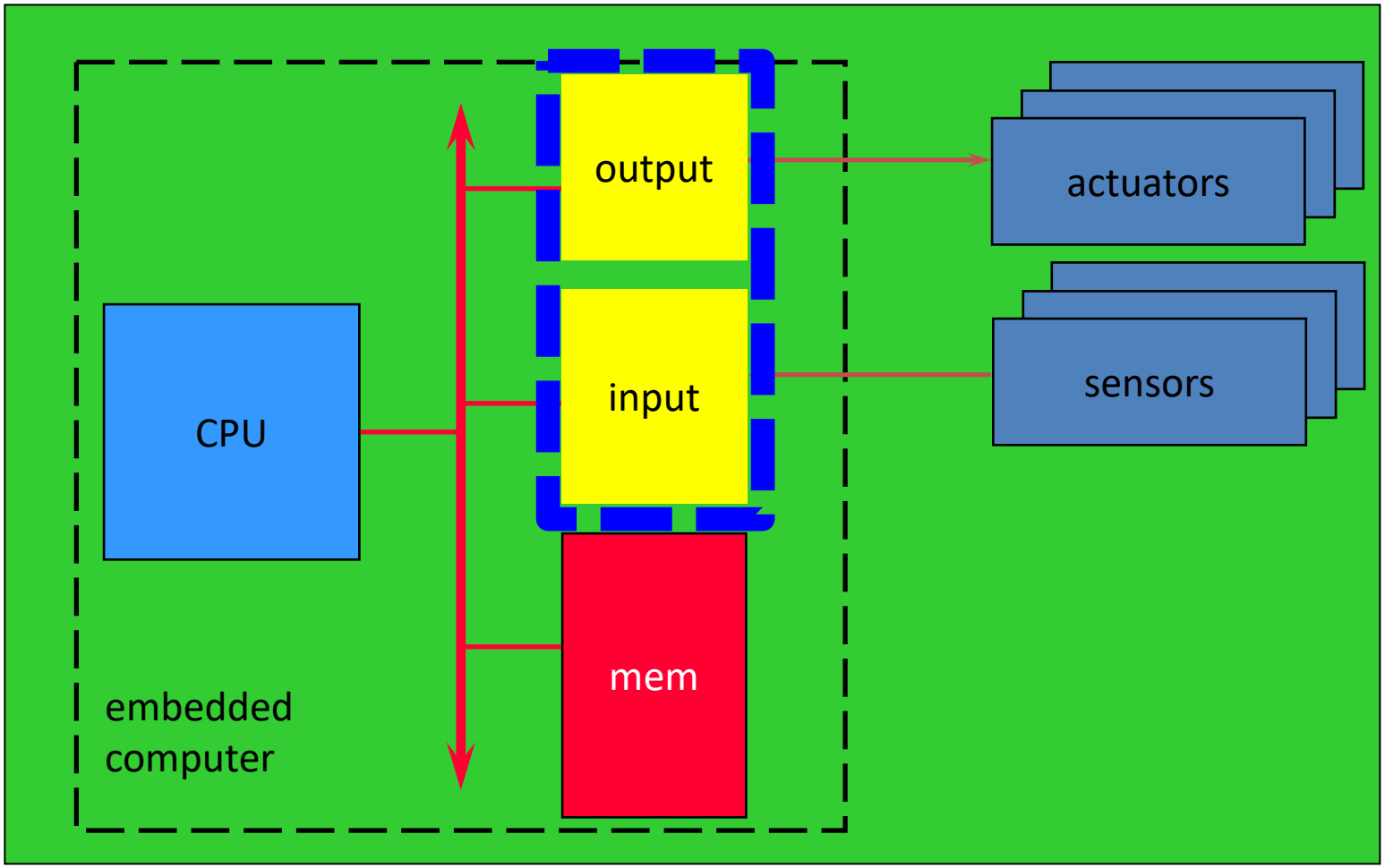
- Taxonomy of microprocessor architecture
  - Von Neumann
    - Memory for both data and instructions, single bus
    - Easier to write
  - Harvard
    - Separate memories for data and instructions, two buses
    - Higher throughput
- RISC vs. CISC
- Multiple implementations of microprocessor
  - ARM: von Neumann + RISC
  - SHARC: Harvard + CISC, optimized for DSP

ECE 1175
Embedded Systems Design

I/O and Interrupt

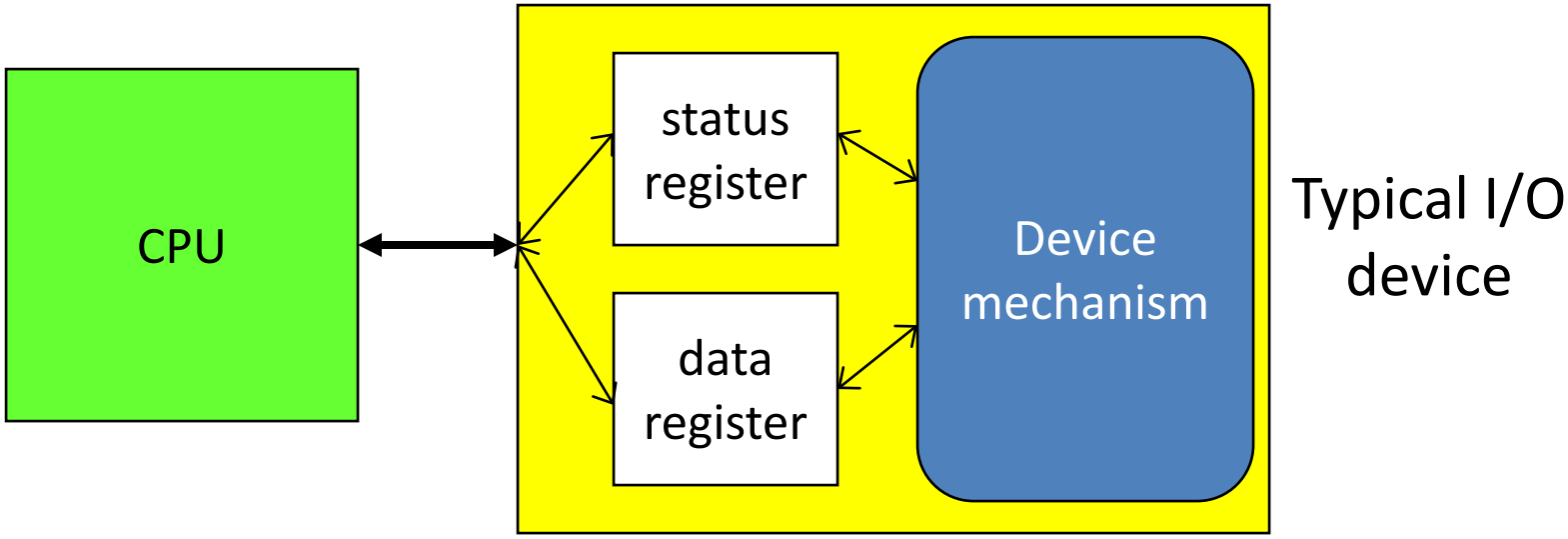
Wei Gao

Embedding A Computer: I/O devices



I/O Devices

- Digital interface to I/O devices
  - Some devices have non-digital components,
    - e.g., electronics for rotating disk and analog read/write in hard drive
- Devices usually use registers to talk to CPU
  - Status: info, e.g. if the data is ready to read, 1: ready, 0: finished
  - Data: holds data outputted to or inputted from the device



Programming I/O Devices

- Two types of instructions can support I/O:
  1. special-purpose I/O instructions;
  2. memory-mapped load/store instructions.
- Intel x86 provides special `in`, `out` instructions.
  - Separate address space for I/O devices
- Most CPUs use memory-mapped I/O.
  - Registers in I/O devices have normal memory address
  - Use CPU's normal read/write instructions to access I/O
  - I/O instructions do not preclude memory-mapped I/O

Peek and Poke

- Traditional high-level functions to implement memory-mapped I/O
  - Read: dereference a given location pointer

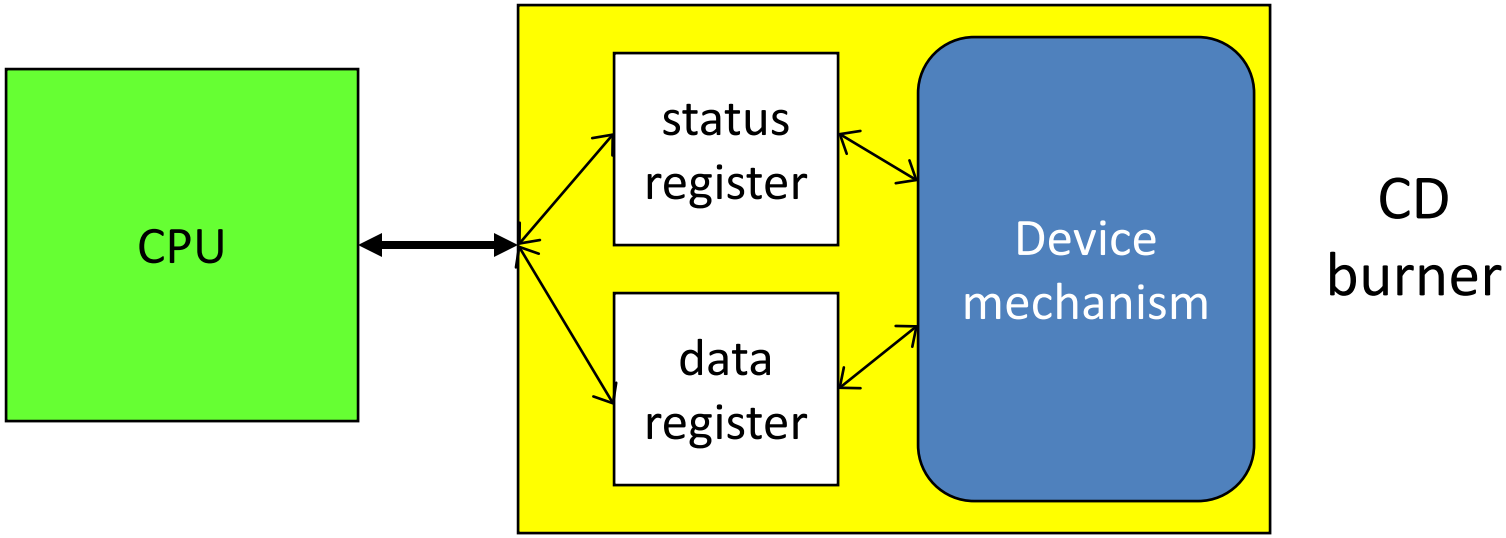
```
int peek(char *location) {
    return *location;
}
```

  - Write to a certain location

```
void poke(char *location, char newval) {
    (*location) = newval;
}
```

Busy-Wait I/O Programming

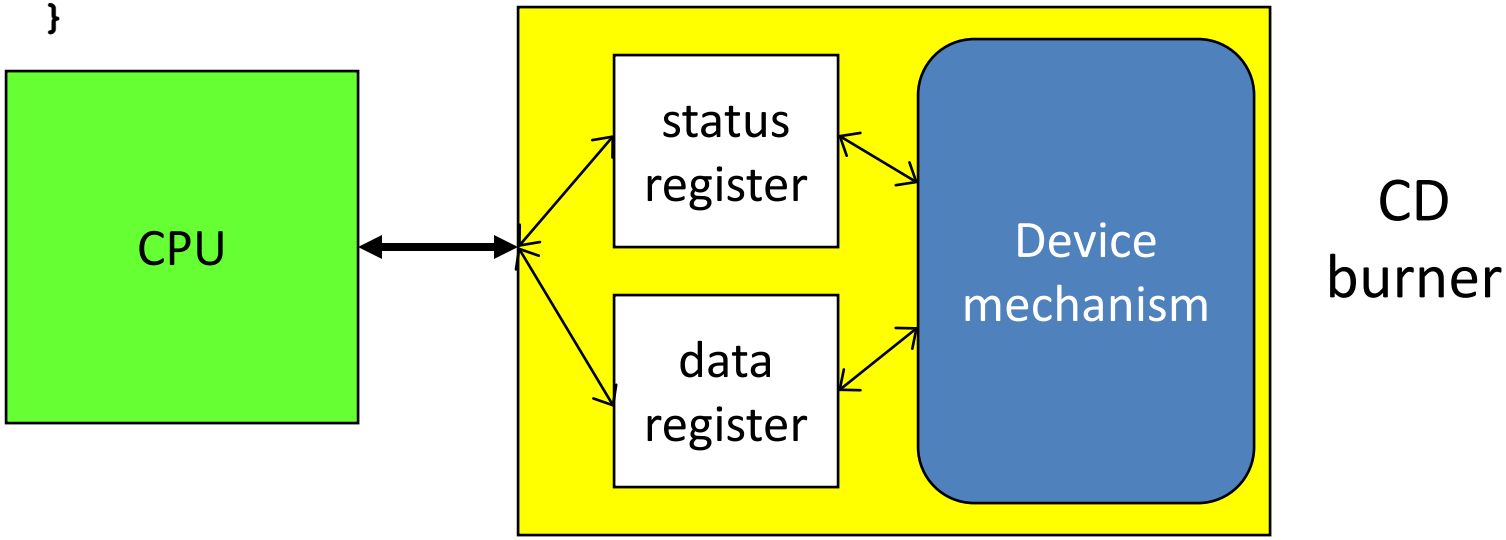
- Simplest way to program I/O devices.
  - Devices are much slower than CPU and require more cycles
  - CPU has to wait for device to finish before starting next
  - Use **`peek`** instruction to test when device is finished



Busy-Wait I/O Programming

```
//send a string to device using Busy-Wait handshaking
current_char = mystring;
while (*current_char != '\0') {
    //send character to device (data register)
    poke(OUT_CHAR,*current_char);
    //wait for device to finish by checking its status
    while (peek(OUT_STATUS) != 0);
    //advance character pointer to next one
    current_char++;
}
```

"test" as an atomic operation



Simultaneous I/O using Busy-Wait

- Repeatedly read a character from the input device and then write it to the output device

```
while (TRUE) {
   /* read a character into variable achar */
   while (peek(IN_STATUS) == 0);/* wait until ready
   */
   achar = (char)peek(IN_DATA);/* read the character
   */
   /* write achar to output device*/
   poke(OUT_DATA,achar);
   poke(OUT_STATUS,1); /* turn on the device */
   while (peek(OUT_STATUS) != 0); /* wait until
   done*/
}
```

Mutual Exclusion using Busy-Wait

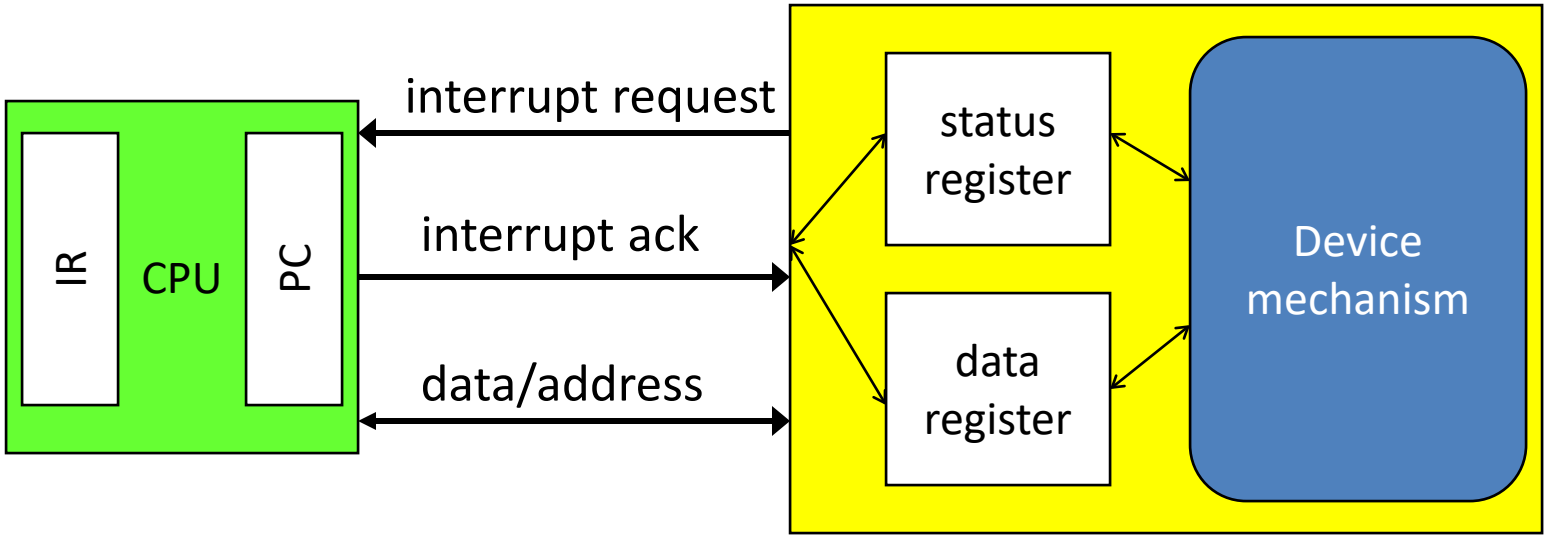
- Exclusive access on the I/O device
  - Multi-threaded systems

```
TestAndSet() {
    oldValue=peek(LOCK); //read
    poke(LOCK, true); //write
    return oldValue;
}
.
.
//wait until LOCK is acquired
while (TestAndSet());
//send character to device (data register)
poke(OUT_CHAR,*current_char);
//wait for device to finish by checking its status
while (peek(OUT_STATUS) != 0);
//advance character pointer to next one
current_char++;
poke(LOCK, false); //release lock
```

"TestAndSet" for mutual exclusion

Interrupt-based I/O

- Busy-wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow to change the flow of control in the CPU.
  - Call interrupt handler (i.e. device driver) to handle device.



Interrupt Behavior

- Based on subroutine call mechanism.
  - Interrupt forces next instruction of CPU to be a subroutine call to the interrupt handler.
  - **Context switch:** return address is saved to resume executing foreground program.
- CPU may not service a request immediately
  - e.g. CPU needs to finish a disk transaction before handling a keyboard interrupt

Interrupt Physical Interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
  - Device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.

Simultaneous I/O using Interrupt

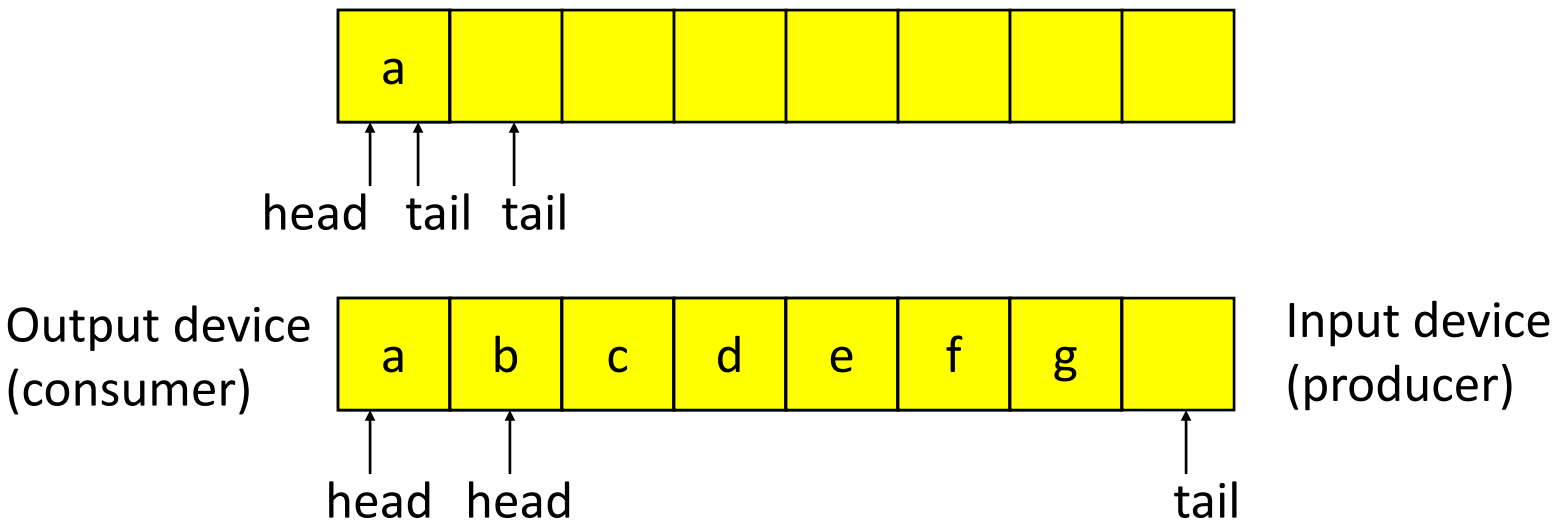
```
void input_handler() { /*get a char and put in global*/
   achar = peek(IN_DATA); /*get a character*/
   gotchar = TRUE; /*Signal to main program*/
   poke(IN_STATUS,0); /*reset status for next transfer */
}
main() {
  while (TRUE) { /*read then wait forever*/
    if (gotchar) { /*write a character*/
       poke(OUT_DATA,achar); /*put character in device*/
       poke(OUT_STATUS,1); /*set status to write*/
       gotchar = FALSE; /*reset flag*/
    }
     // do some other jobs here.
  }
}
```

Request interrupt when input comes

Write data as interrupt handler

Interrupt I/O with Buffers

- No waiting; CPU can read and write simultaneously
- Use a queue to store characters, producer/consumer
- Allow input/output devices to run at different rates

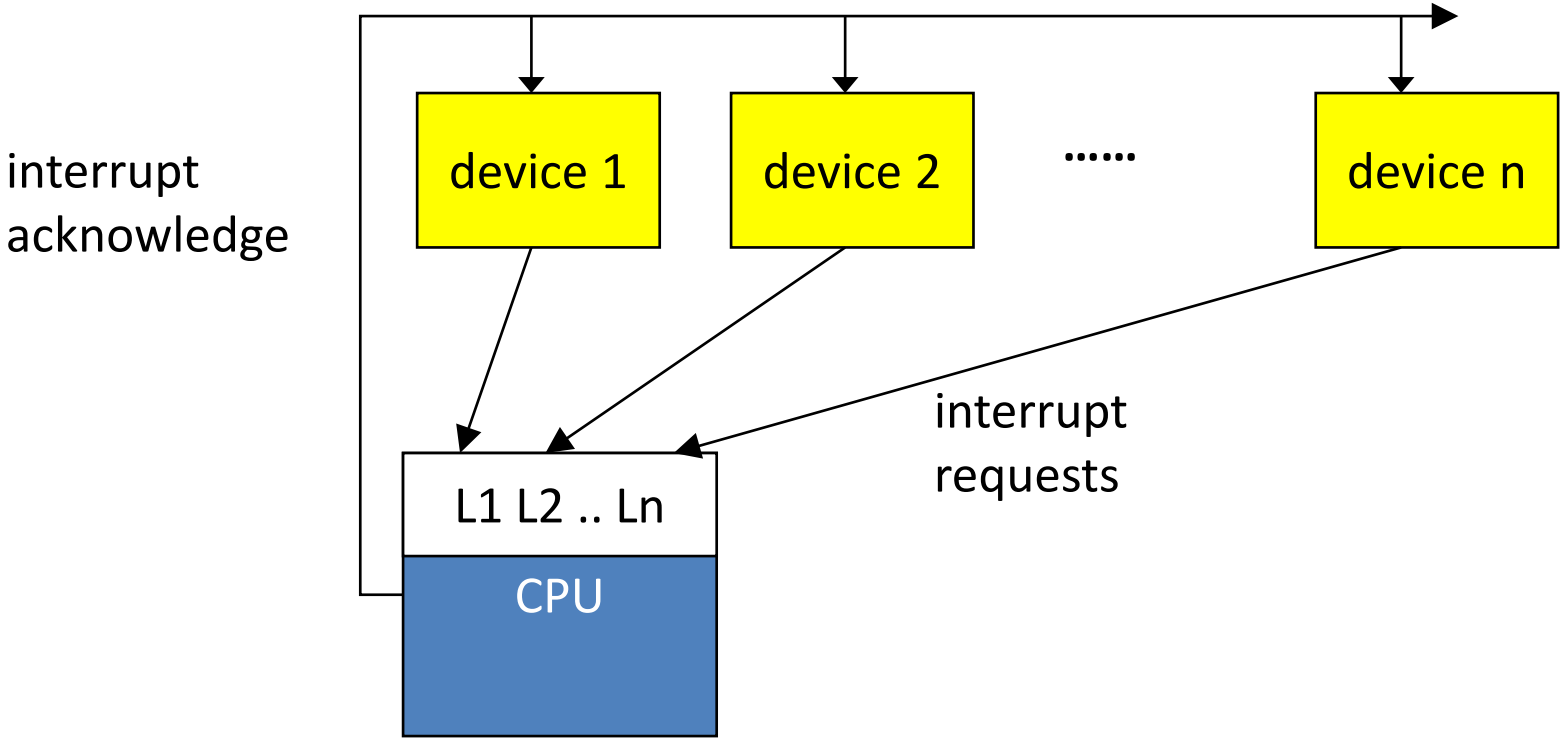


Priorities and Vectors

- Need to handle interrupts from multiple devices
- Two mechanisms allow us to make interrupts more general:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Mechanisms are orthogonal: most CPUs provide both.

Prioritized Interrupts

- Some interrupts are more important
- Lower-numbered interrupt lines have higher priority
- Re-connect lines to change priorities

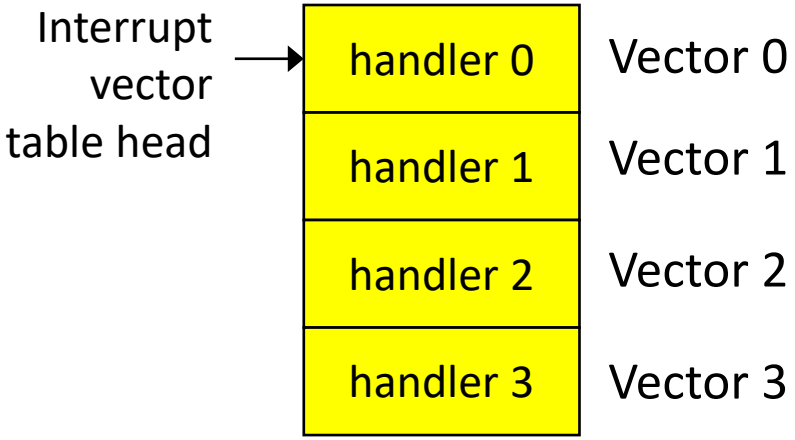


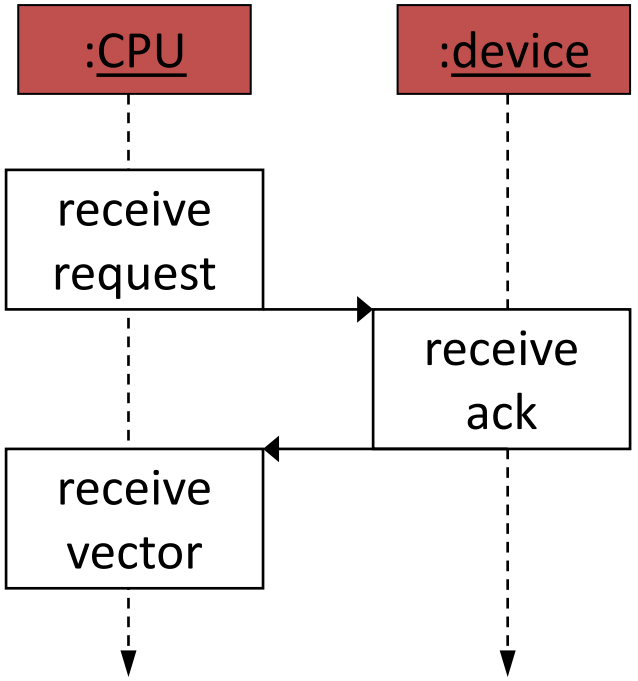
Interrupt Prioritization

- Masking: interrupt with priority lower than current priority is not recognized until current interrupt is complete.
  - In TinyOS, event handlers should not do long processing because the priority is highest by default
  - Use tasks because they can be preempted
- Highest-priority is non-maskable interrupt (NMI) which is never masked.
  - Often used to save critical states of memory and turn off devices for power-down.

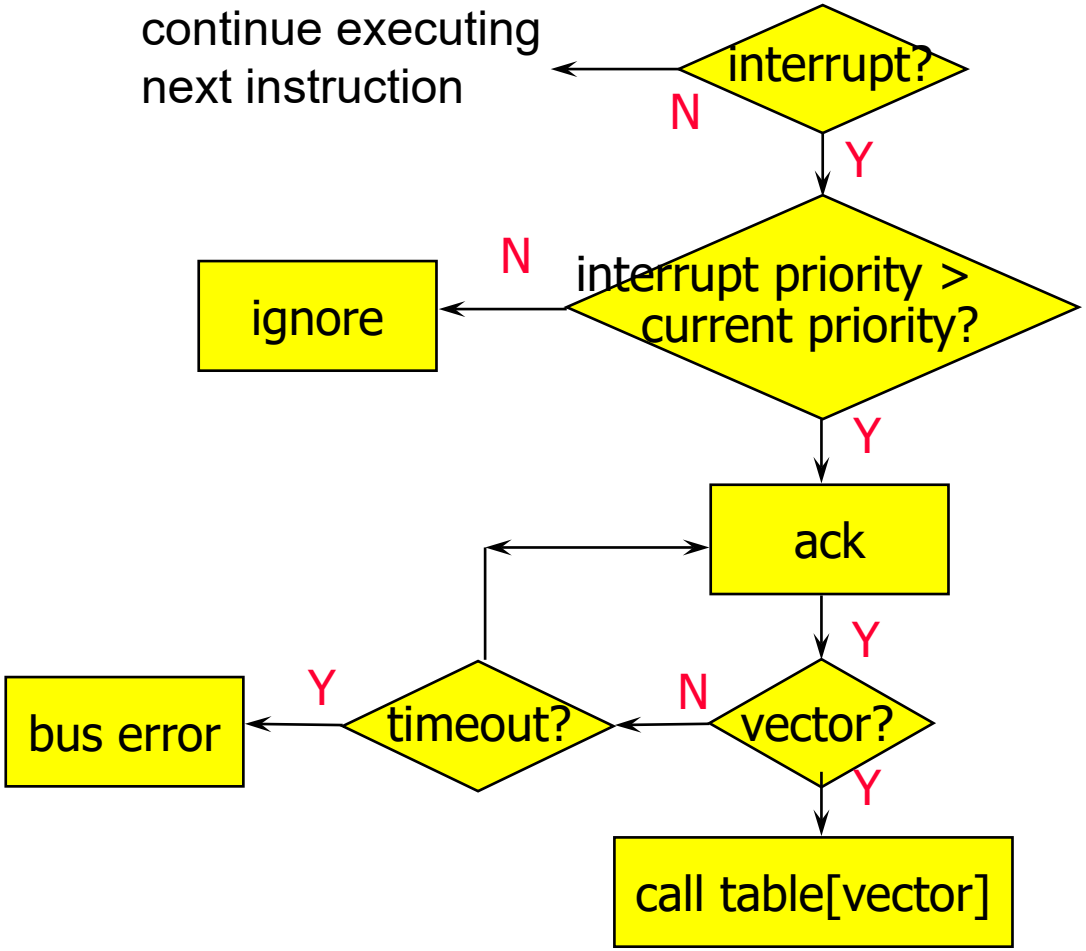
Interrupt Vectors

- Flexibility: allow different devices to be handled by different handlers; mapping is changeable
- Devices store the vector #
- Interrupt vector table:





Generic Interrupt Mechanism



Interrupt Sequence

1. Device requests interrupt
2. CPU checks for pending interrupts and acknowledges the highest priority request.
3. Device receives acknowledge and sends CPU the interrupt vector.
4. CPU saves current states, looks up and calls the corresponding handler.
5. Handler processes request.
6. CPU restores states to foreground program.

Interrupt Overhead

- Context switch: registers (e.g., PC) save/restore when handler is called.
- Handler execution time.
- Extra cycles for requests, acknowledges, vectors, etc.
- Other overhead
  - Pipeline-related penalties.
  - Cache-related penalties.

Summary

- I/O programming
  - Memory-mapped I/O vs. special-purpose I/O instructions
  - Busy-wait is simplest but very inefficient
    - Devices are usually slower than CPU
- Interrupts
  - Using buffer to allow input/output at different rates
  - Priorities and vectors allow to handle multiple interrupts