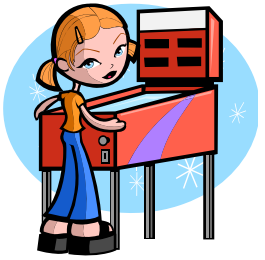
Lecture 7
I/O and interrupts

Lecture overview

- Embedded & real-time systems
- I/O
- Interrupt handling

Embedded systems

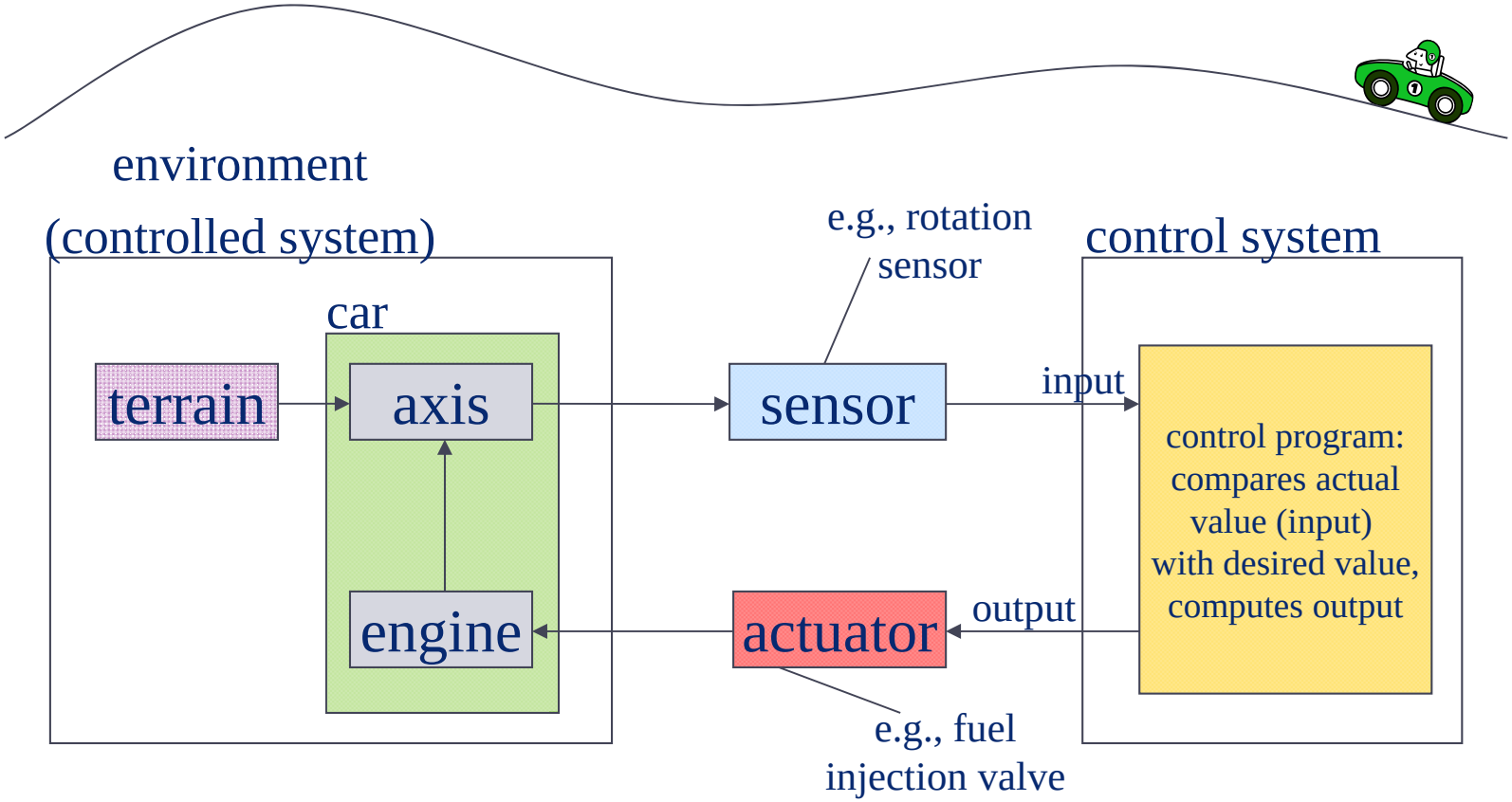
- Hard to define
- “A computer that is not perceived as a computer”
  - E.g., microcontrollers + control programs in cars, household appliances, pinball machines, etc.
  - Typically have a very specific task
    - Not general-purpose computation
- Should be small & cheap to mass produce
  - Cheap components
    - E.g., slow processors, slow buses, small memory
- Typical task: feedback control loop



Feedback control loop

- Measure value (typically quantifiable aspect of an *environment*)
- Compare *actual value* (“*ärvärde*”) with *desired value* (“*börvärde*”)
- Compute control signal, aim to make actual value closer to desired value
- Repeat loop
- Example: cruise control

Feedback control loop example: cruise control



Real-time systems

- Embedded systems are often **real-time systems**
- A real-time system is a system
  - that **interacts** with an **environment**
  - whose state is **affected by the passage of time**
  - where **predictability** and **sufficient efficiency** are important
- Predictability: resource requirements **known** and **bounded**
- Sufficient efficiency: there are enough resources
  - Possible to perform all critical tasks in **worst-case** scenario

Real-time systems: tasks and jobs

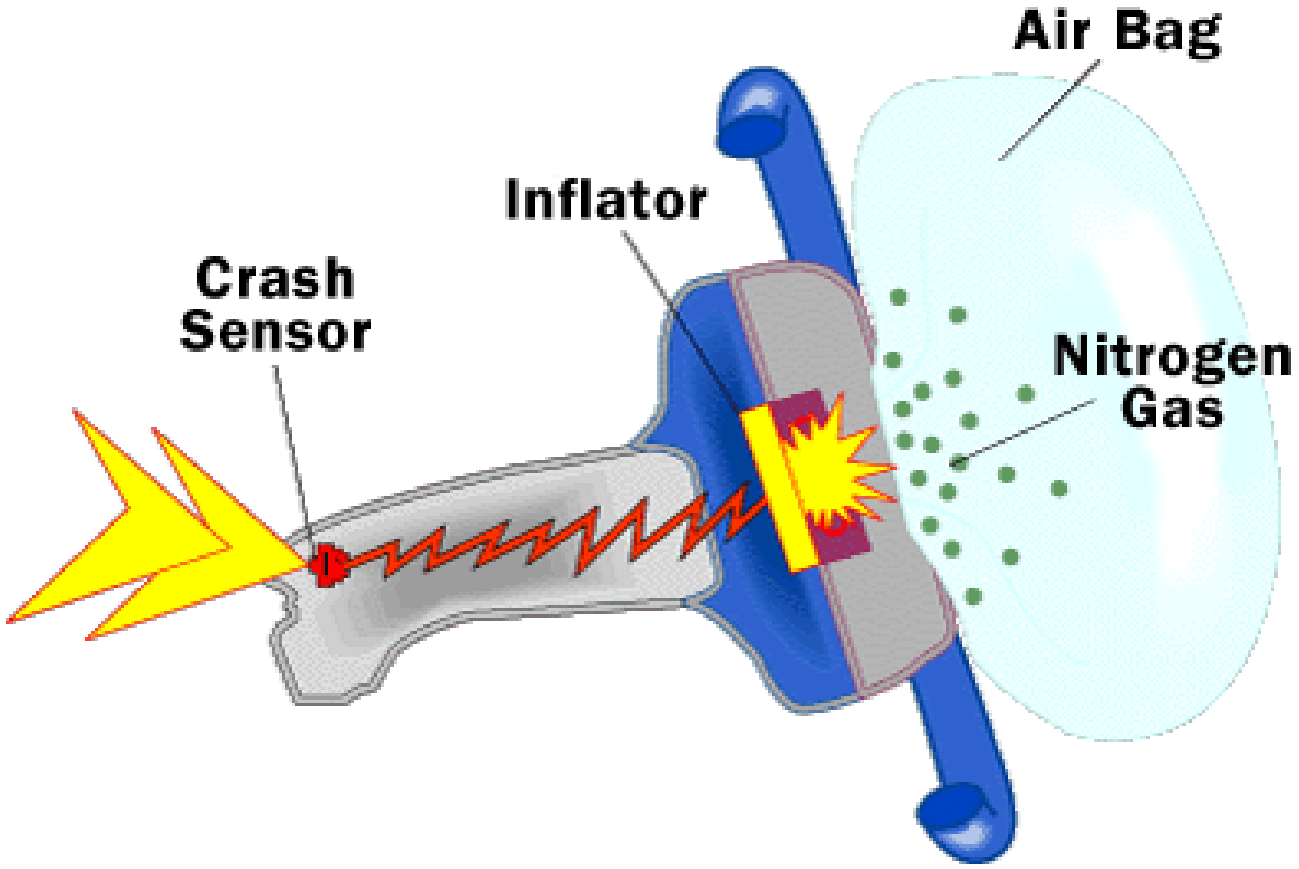
- A **task** is something the RTS must perform periodically or sporadically
  - E.g., check reactor heat, shut down if too high
- A **job** is an instance of a task
  - I.e., a single task activation
- Each job has a relative **deadline**
  - Finishing a job after its deadline has detrimental effects
  - Soft deadline: value of job decreases after deadline
  - Firm deadline: value of job = 0 after deadline
  - Hard deadline: value of job = $-\infty$ after deadline

Soft deadline

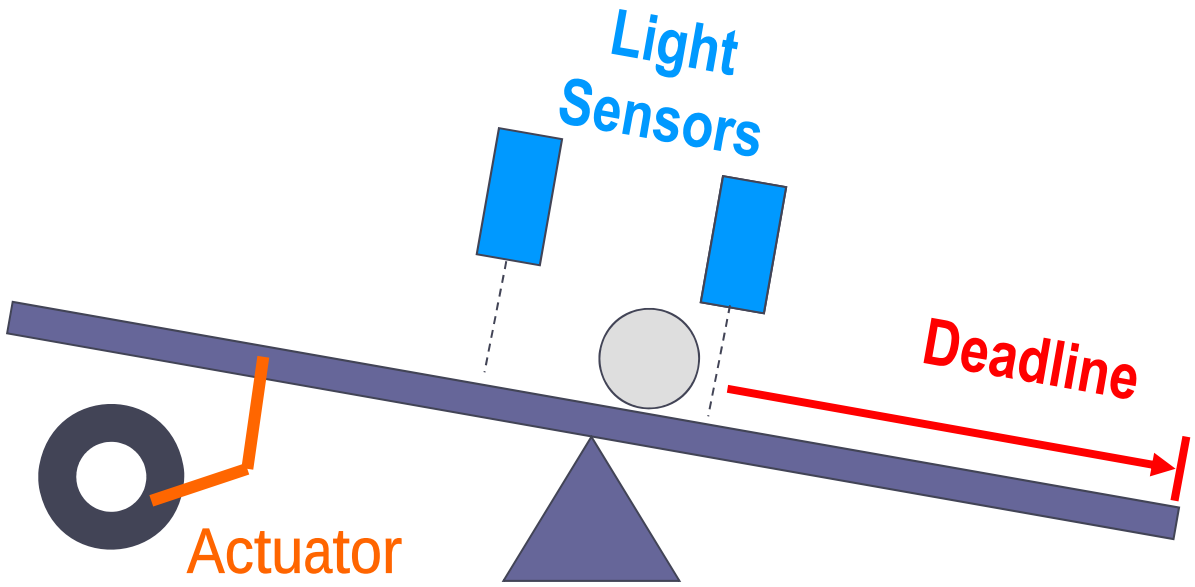
Nobina 7180 vid Skövde resecentrum den 19 april 2011. Foto: Mikael T Nilsson

Firm deadline

Hard deadline



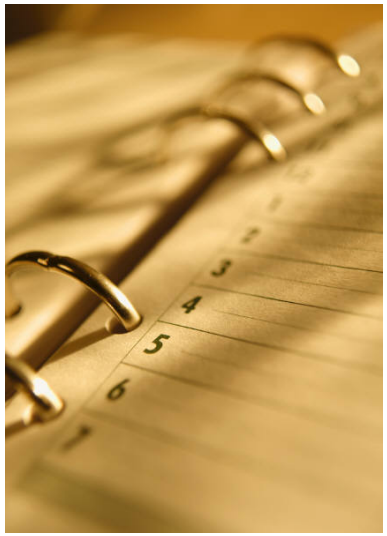
Real-time system, illustration



Predictability

- Real-time systems are often safety-critical …
  - Deadline miss = risk for lives or expensive damage
- … and even if they aren't
  - Value comes from meeting deadlines
- To **guarantee** that a system meets requirements
  - The system must be **analyzable** and **predictable**

Fundamental RTS problem: scheduling



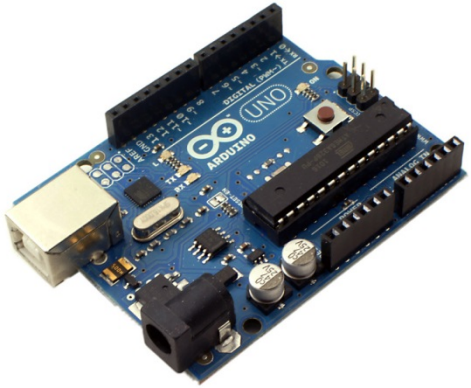
- Choosing which job to run next
  - Example metrics:
    - Throughput (number of jobs completed)
    - Value of completed jobs
  - Safety must be guaranteed
    - Hard deadlines must never be missed
    - But value should still be maximized (QoS)
  - Example **scheduling policies**
    - Shortest job first (maximizes throughput)
    - Earliest deadline first (avoids starvation, ensures all deadlines met if possible)
  - Scheduling depends on ability to determine job execution time

Achieving predictability

- What makes predictability hard to achieve?
  - **Nondeterminism** as introduced by e.g.,
    - Thread/processor scheduling
    - Memory allocation/fragmentation
    - Virtual memory
    - Caches and pipelines
    - Bugs
    - Unpredictable environments
  - **Opaque system features** such as
    - Virtual machines
    - Compiler optimization
    - High-level, complex language constructs (e.g., object-orientation)
    - I/O drivers
    - External systems (e.g., database managers)

Practical concerns for embedded & real-time systems

- Compile/assemble code
  - Typically no compilers running on the hardware itself
  - Solution: **cross-compilers**
- Transfer program to hardware
- Debug the system
  - Typically hard to do on the final hardware
  - **Emulators** are often used
- Perform I/O
  - Read from sensors, write to actuators
- We now focus on I/O



I/O overview

- Two main ways of programming I/O
  - Special instructions (“port-mapped I/O”)
  - Memory-mapped I/O
- Two ways of transferring data
  - Programmed I/O
  - Direct memory access (DMA)
- Two ways of sending data
  - Serial communication
  - Parallel communication
- Two timing models for I/O
  - Polling
  - Interrupt-based I/O

Port-mapped I/O

- Special instructions to access I/O devices
  - E.g., **`in`** & **`out`** on Intel/Atmel processors
    - Reads/writes a byte to an I/O device
- Possibly specific instructions for each I/O port
  - E.g., **`dia`** (data input from register A), **`doa`** (data output to register A) **, `dib`, `dob`, `dic`, `doc`** on the Nova computer
  - Additional instructions to control flow based on I/O status
    - E.g., **`skpbz`** (skip if device not busy), **`skpdn`** (skip if device done)

Port-mapped I/O, pros and cons

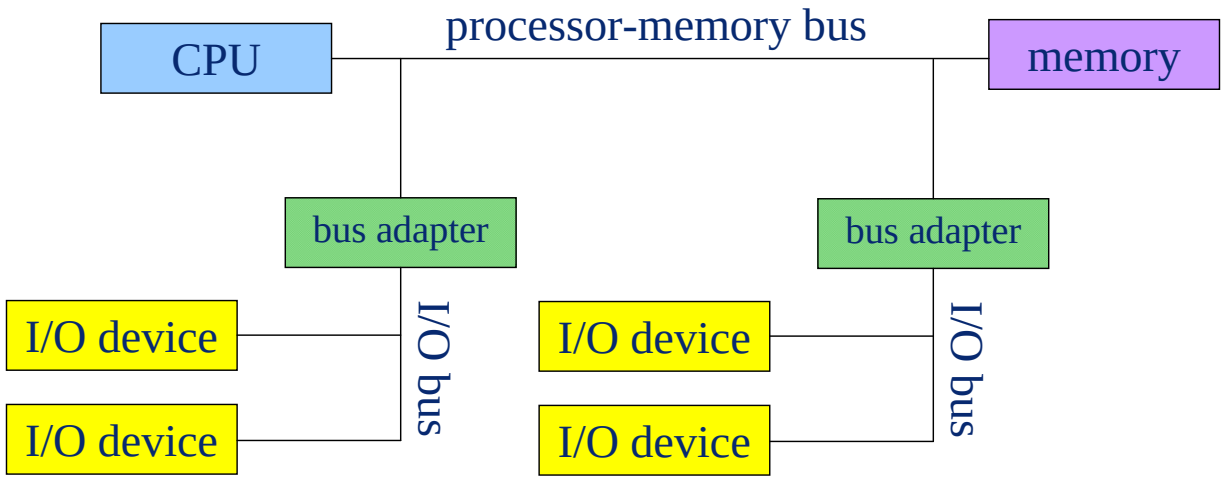
- Does not use space in the address range
  - May be small on embedded devices
  - Complex devices require a large address range (e.g., graphics cards)
- Explicit I/O makes code easy to read / optimize
  - I/O looks different from regular memory access
- I/O range smaller than address range
  - No need to perform full address decoding
  - If specialized instructions for each register: no decoding necessary!
- Hard to do in high-level languages such as C/C++
  - No inherent support for port-mapped I/O
  - Requires inline assembler
- Architecture becomes more complex, additional instructions

Memory-mapped I/O

- Do I/O by accessing specific memory addresses
  - Uses “normal” instructions for memory access
  - E.g., **`mov`** on H8 processors, **`lw`**/**`sw`** on MIPS, **`ld`**/**`st`** on ATmega
- Avoiding use of I/O address range by applications
  - Operating system protects address range
  - Systems without O/S: the programmer’s responsibility

Advantages of memory-mapped I/O

- Simpler internal architecture, fewer instructions
- Can use memory bus to communicate with I/O devices
  - Devices either connected directly to system bus, or bus hierarchy
- Possible to use all available addressing modes
- Possible in high-level languages that allow direct memory addressing



Disadvantages with memory-mapped I/O

- Requires address decoding
- Code becomes harder to read
- **Caching becomes complex**
  - Hard to distinguish between memory access and I/O access
  - Values of I/O ports cached
    - Major problem; operations performed on cache
    - Solution: explicitly remove lines from cache, prevent caching
- **Compilers may optimize incorrectly**
  - May remove operations that look redundant
  - Solution (in C/C++) the **`volatile`** keyword

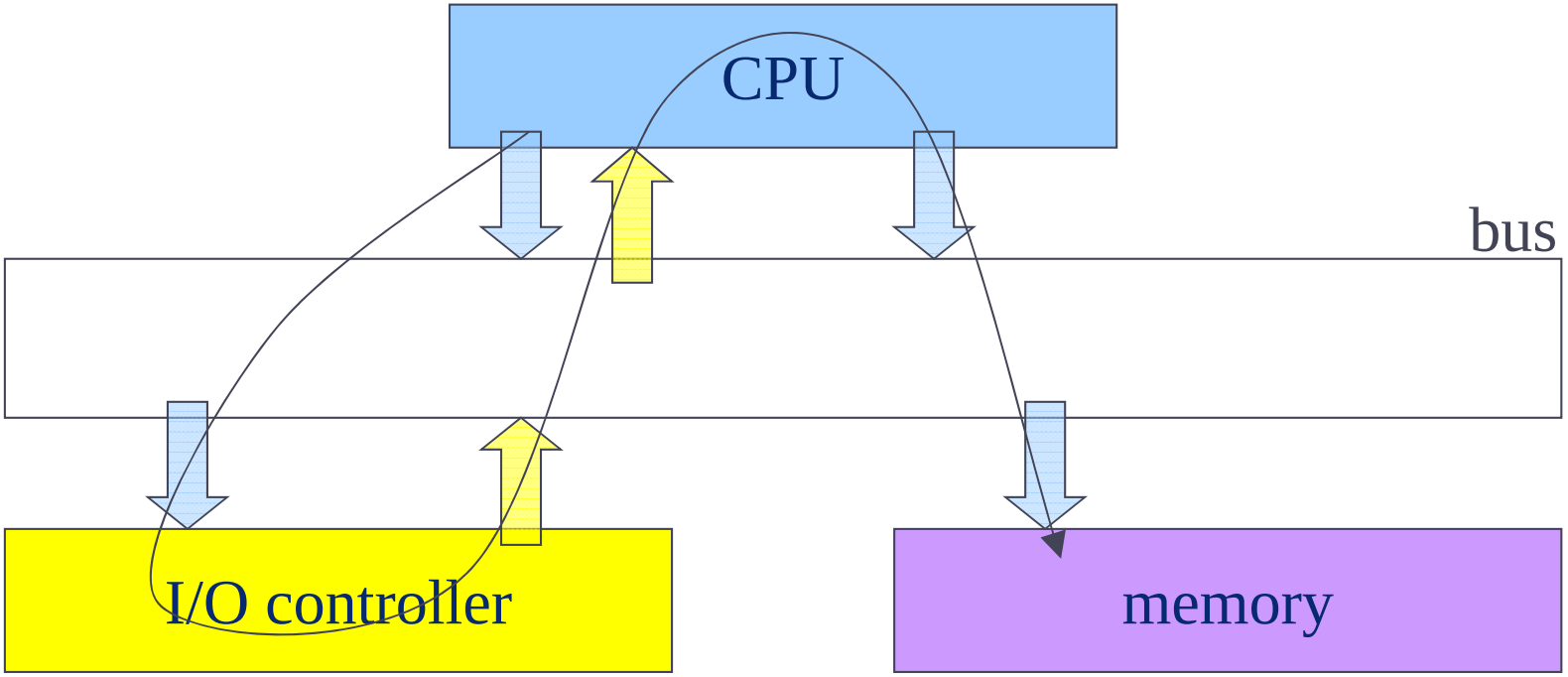
Example

```
void *p = 0x000000FF;
volatile char a = *p;
while(a == 0) {
   a = *p;
}
```

Data transfer: programmed I/O (PIO)

- In some architectures (e.g., some real-time systems), the **processor** performs I/O
  - CPU busy during entire I/O period
  - Sometimes not a problem
    - If “normal” processing cannot continue until data available, and
    - operation speed is not critical
  - Problematic if it wastes processor time
    - E.g., transferring data from hard drive to memory

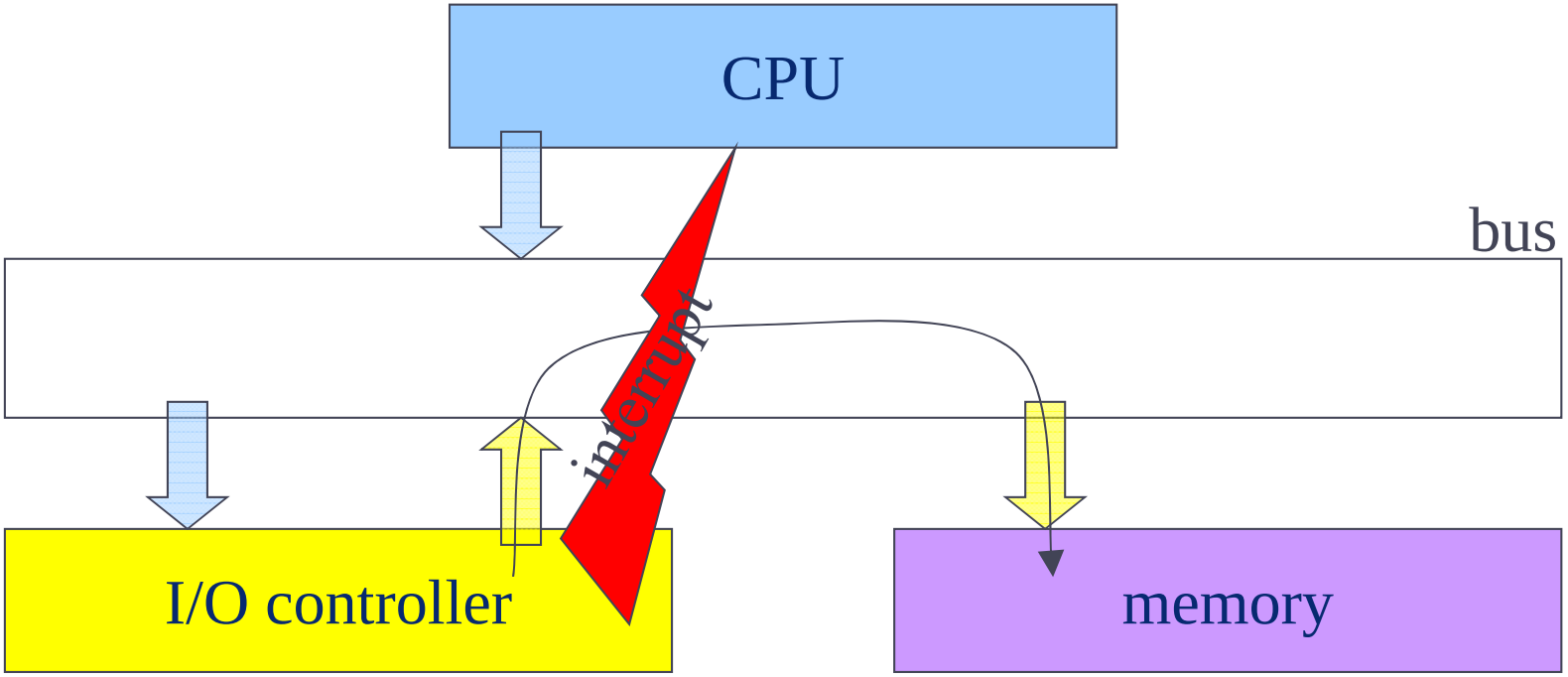
Programmed I/O



Data transfer: Direct Memory Access (DMA)

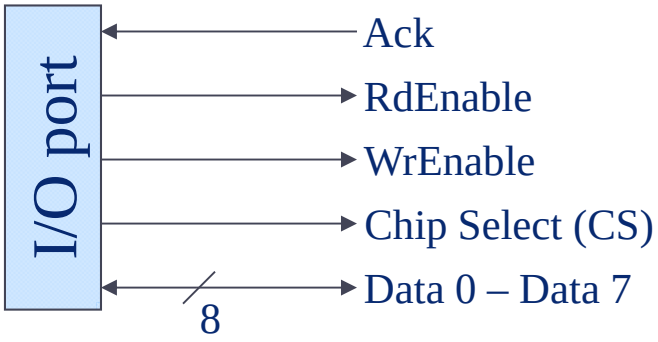
- Allow I/O devices to transfer data blocks directly
  - Typically between devices and system memory
  - Either via *third-party* DMA …
    - A device-independent DMA controller does the transfer
  - … or via *first-party* DMA
    - Devices take control of system bus (*bus mastering*)
  - Much faster than programmed I/O
  - Can be realized in hardware
  - Time gain even if processor idles during DMA transfer
    - Best performance by using multiple *threads*

Data transfer with DMA



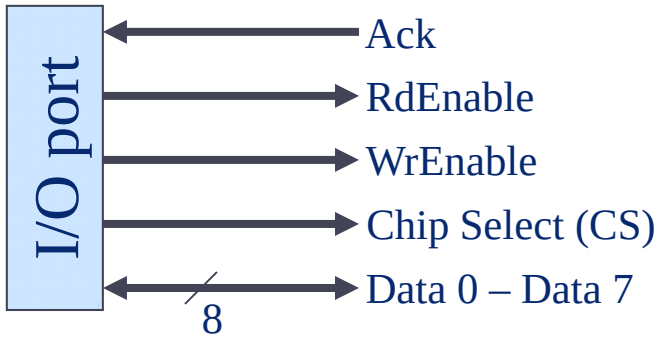
Parallel communication

- Several signal wires
  - Fast communication (1 bit per wire, in parallel)
  - Can only be used over short distances due to crosstalk and clock skew
- 8-bit parallel interface:



Parallel communication example: input

- Processor puts I/O mapped address on address bus
  - Causes RdEnable and CS to be activated
- I/O device notices that CS is active
  - Checks RdEnable, WrEnable, detects active RdEnable
- I/O device puts data on data lines
- I/O device activates Ack line
- CPU notices that Ack is active
- Data is collected from data lines



Serial communication

- Only three wires
  - One of each direction of communication + ground
  - Slower communication, but longer distances

- Depends on known transmission speed
  - Synchronization only at beginning of transmission
  - A *start bit* (0) signals new transmission
  - Each transmission short enough (e.g., one byte) to ensure that clocks do not diverge
  - Parity bits used to detect errors
    - Retransmission used for corrupt data

Polling

- CPU “asks” I/O device for data
  - Busy-wait until data is available

- Phone analogy: phone with malfunctioning signal

Polling example

- Read data from keyboard
  - Assume that only one key can be pressed at a time
- I/O port for keyboard on address **`0xbfcc0000`**
- I/O port for status (bit 7 signals that a key is pressed) on address **`0xbfcc0020`**

Polling example, C

- I/O port for keyboard on address **0xbfcc0000**
- I/O port for status (bit 7 signals that a key is pressed) on address **0xbfcc0020**

```
unsigned char *status =
    (unsigned char *) 0xbfcc0020;
unsigned char *keyboard =
    (unsigned char *) 0xbfcc0000;

char getchar() {
  /* bit mask: 0x80 = 10000000 */
  while(!(*status & 0x80)) ;
  return *keyboard;
}
```

Polling example, MIPS assembler

- I/O port for keyboard on address **0xbfcc0000**
- I/O port for status (bit 7 signals that a key is pressed) on address **0xbfcc0020**

```
getchar:
  la      t0,0xbfcc0000
L1:
  lb      t1,0x20(t0)     # status->t1
  nop
  andi    t1,t1,0x80      # mask bit 7
  beq     t1,zero,L1      # branch if no key
  nop
  lb      v0,0(t0)
  jr      ra
```

Interrupts

- I/O device signals whenever it needs the CPU's attention, e.g., when …
  - … an external event occurred (e.g., temperature > 90)
  - … it finished a task assigned by the CPU (e.g., DMA transfer)
  - … a certain amount of time has passed
- Phone analogy: regular phone

INTERRUPTS

Nobody expects them

External and internal interrupts

- External interrupts are caused by I/O devices
- Interrupts can also be generated by the CPU
  - Often called *exceptions* (MIPS) or *internal interrupts*
  - Examples:
    - **`break`** instruction executed (used for debugging)
    - System call (**`syscall`**) executed
    - An error occurred (arithmetic, address alignment, instruction decode, …)

Interrupt handling

- Interrupts cause execution of **interrupt handlers** (**interrupt routines**)
- Require special treatment
  - Interrupts can happen anywhere
  - Not possible to use, e.g., **`$ra`** for return address
- MIPS has three interrupt-specific registers:
  - Status: controls which user interrupts are allowed
  - Cause: contents reflect which interrupt has occurred
  - EPC: return address after interrupt handling
    - Exception Program Counter

Interrupt handling in MIPS (brief)

- Execution is stopped
- Cause and EPC registers are updated
- Program jump to address **`0x80000080`**
  - Contains interrupt handling routine
  - Typically checks Cause register and calls specific handler

Installing interrupt routines

- Problem: ensuring that interrupt-handling code is in the right location
  - MIPS handling must start at **`0x80000080`**
  - Sometimes assembler allows specification of addresses
    - E.g., AVR studio to ATmega328p (**`.ORG`** directive)
  - If not, necessary to programatically copy code

Installing interrupt routines, cont.

- Assume that the interrupt handler is specified in a routine **`int_routine`**
- We could install **`int_routine`** at adress **`0x80000080`**
- Shortcut: install the following code

```
intstub:
  j  int_routine
  nop
```

should be copied to **`0x80000080`**

Copying code

- The following code installs **`intstub`**

```
la  t0,intstub
la  t1,0x80000080
lw  t2,0(t0)
lw  t3,4(t0)
sw  t2,0(t1)
sw  t3,4(t1)
```

- Executed at program initialization
- All that remains is coding **`int_routine`**…

Atmega 328p interrupt vectors, example

```
.CSEG                   // Code segment
.ORG 0x0000
jmp init                // Reset vector
.ORG 0x0020
jmp isr_timerOF     // Timer0 overflow
.ORG INT_VECTORS_SIZE
// Rest of code
```

Polling vs. interrupts

- Advantages of polling
  - Easier to implement
  - Makes system easier to analyze
    - Easier to ensure predictability (*time-triggered* real-time systems)
- Disadvantages of polling
  - May cause unnecessary load on CPU
    - If value changes infrequently
  - May miss high-frequency fluctuations in input value

Polling vs. interrupts, cont.

- Advantages of interrupts
  - Avoids CPU use when nothing happens
- Disadvantages of interrupts
  - Handled outside (i.e., “interrupts”) normal program flow
  - Makes system much harder to analyze
    - Used in **event-triggered** real-time systems
      - Much harder to guarantee correctness
      - Much harder to test
- Use the technique that best fits program semantics
  - Is system behavior cyclic/predictable or hard to predict?

Combination of techniques

- Possible to combine polling and interrupts
- A timer generates interrupts with a given period
  - E.g., a timer interrupt every 10ms
- Interrupt handling routine polls I/O device
  - (Or sets a flag that it should be polled)
- More “wasteful” than pure interrupts
  - But not much; timer interval can be carefully chosen
- System becomes much more predictable
- Common in time-triggered real-time systems