


A Timely Question.

- Most modern operating systems **pre-emptively** schedule programs.
 - If you are simultaneously running two programs A and B, the O/S will periodically switch between them, as it sees fit.
 - Specifically, the O/S will:
 - Stop A from running
 - Copy A's register values to memory
 - Copy B's register values from memory
 - Start B running
- How does the O/S stop program A?

← HAN INTERRUPT
← store
← loads
← jr \$__

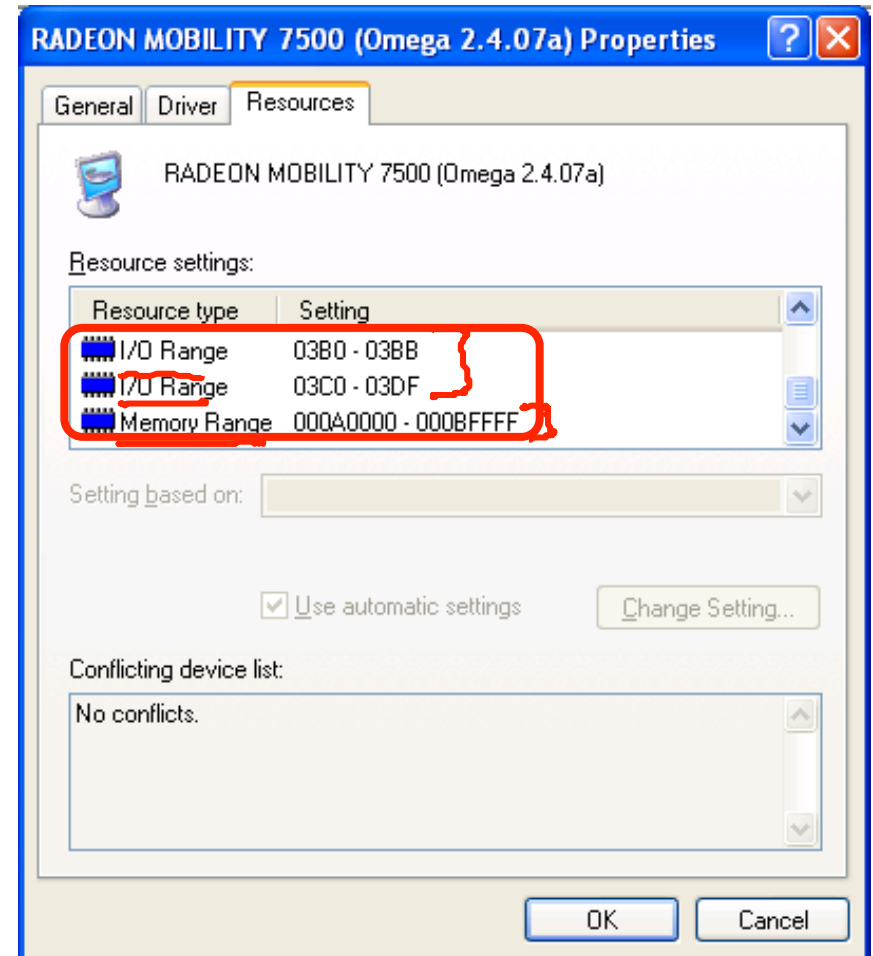
NO HANDOUT

I/O Programming

- I/O requests are made by applications or OS
 - involve moving data between peripheral device and main memory
- Two main ways for programs to communicate with devices:
 - Memory-mapped I/O  MIPS
 - Isolated I/O
- Several ways to transfer data between devices and main memory:
 - Programmed I/O
 - Interrupt-driven I/O
 - Direct memory access
- We will explore some of these in more detail in the MPs

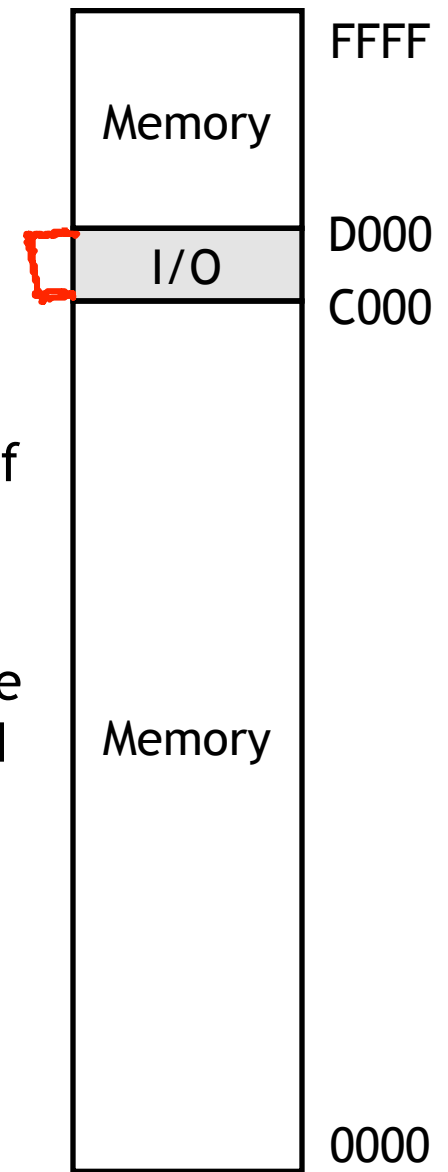
Communicating with devices

- Most devices can be considered as memories, with an “address” for reading or writing
- Many ISAs often make this analogy explicit — to transfer data to/from a particular device, the CPU can access special addresses
 - *Example:* Video card can be accessed via addresses 3B0-3BB, 3C0-3DF and A0000-BFFFF
- Two ways to access these addresses:
 - Memory-mapped I/O
 - Isolated I/O

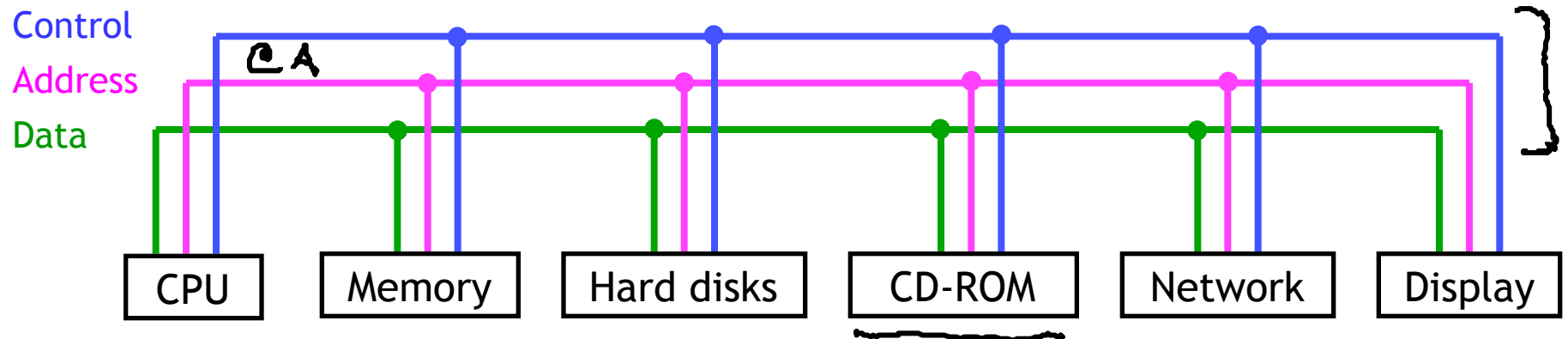


Memory-mapped I/O

- With **memory-mapped I/O**, one address space is divided into two parts.
 - Some addresses refer to physical memory locations.
 - Other addresses actually reference peripherals.
- For example, my old Apple IIe had a 16-bit address bus which could access a whole 64KB of memory.
 - Addresses **C000-CFFF** in hexadecimal were not part of memory, but were used to access I/O devices.
 - All the other addresses did reference main memory.
- The I/O addresses are shared by many peripherals. In the Apple IIe, for instance, C010 is attached to the keyboard while C030 goes to the speaker.
- Some devices may need several I/O addresses.



Programming memory-mapped I/O

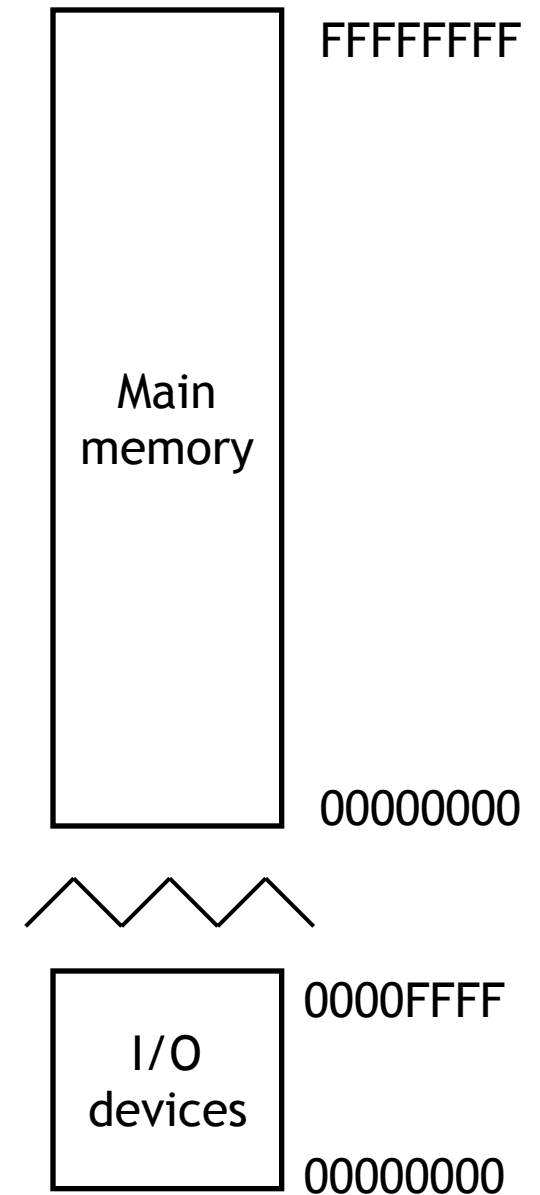


- To send data to a device, the CPU writes to the appropriate I/O address. The address and data are then transmitted along the bus.
- Each device has to monitor the address bus to see if it is the target.
 - The Apple IIe main memory ignores any transactions whose address begins with bits 1100 (addresses C000-CFFF).
 - The speaker only responds when C030 appears on the address bus.



Isolated I/O

- Here, there are *separate* address spaces for memory and I/O devices
 - special instructions that access the I/O space
- *Example (x86):*
 - regular instructions like **MOV** reference RAM
 - special instructions **IN** and **OUT** access a separate I/O address space
- An address could refer to *either* main memory *or* an I/O device, depending on the instruction used



MIPS/SPIMbot uses memory-mapped I/O

// \$t0, 10

- SPIMbot uses memory-mapped I/O:

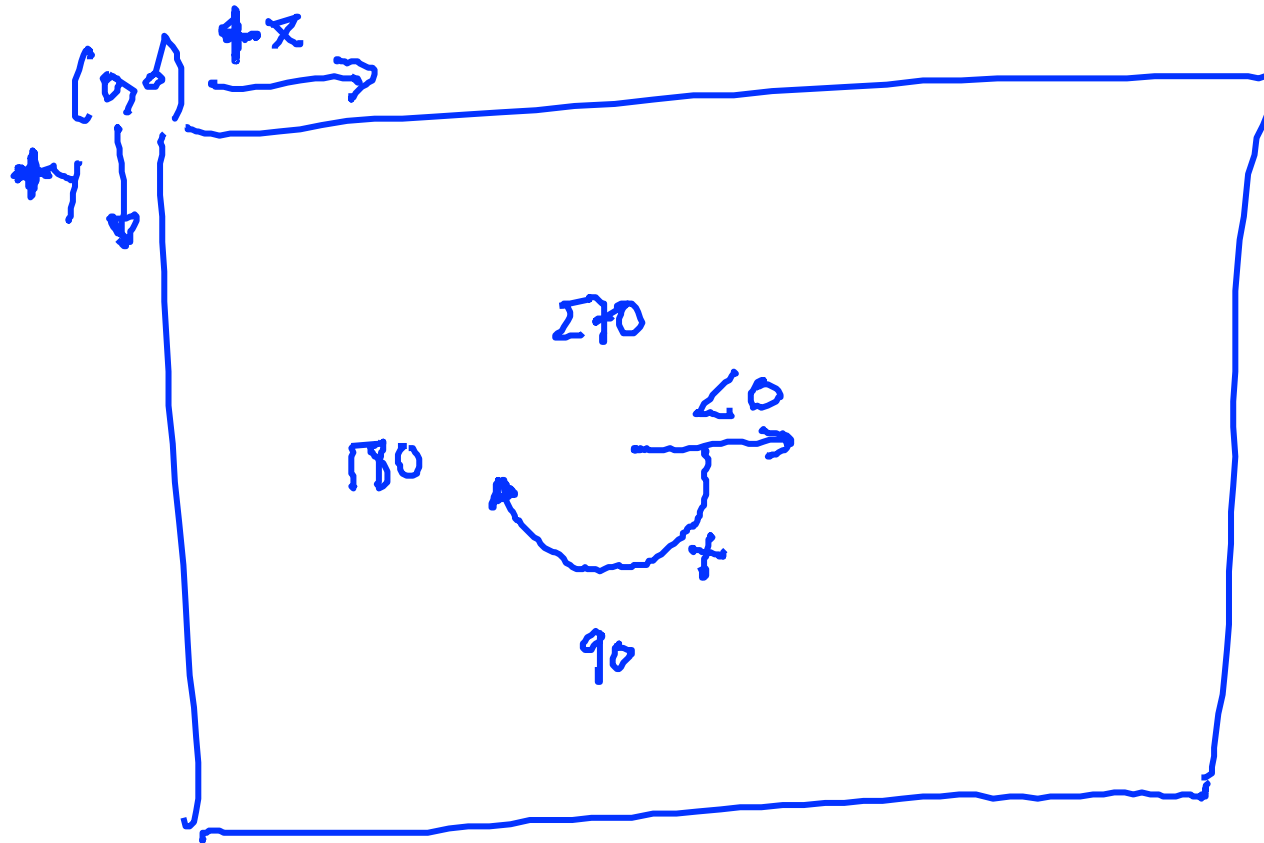
```
sw    $reg, 0xffff0080($0)    # prints integer $reg
sw    $reg, 0xffff0010($0)    # sets bot speed = $reg
      t0                ~10 ~ 10
```

// \$t1, 270

- Some things require a sequence of instructions

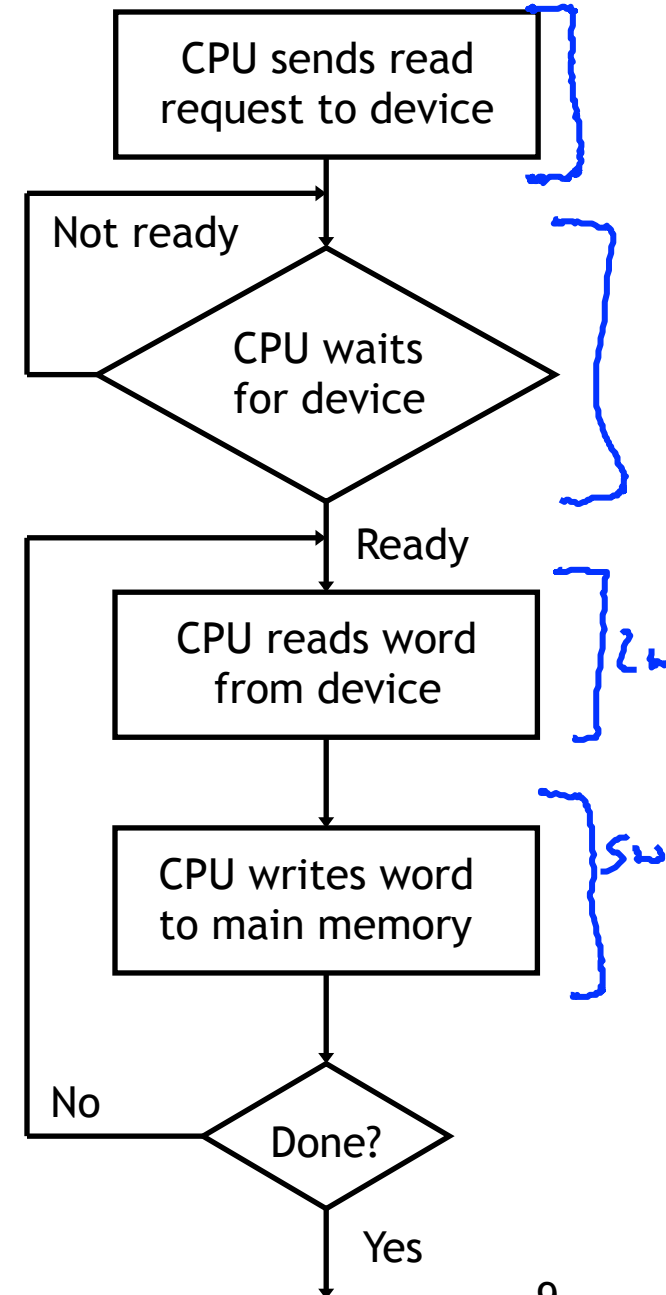
```
sw    $reg, 0xffff0014($0)
      reg
li    $t0, 1
sw    $t0, 0xffff0018          # sets bot angle = $reg
```

SPIMbot coordinates



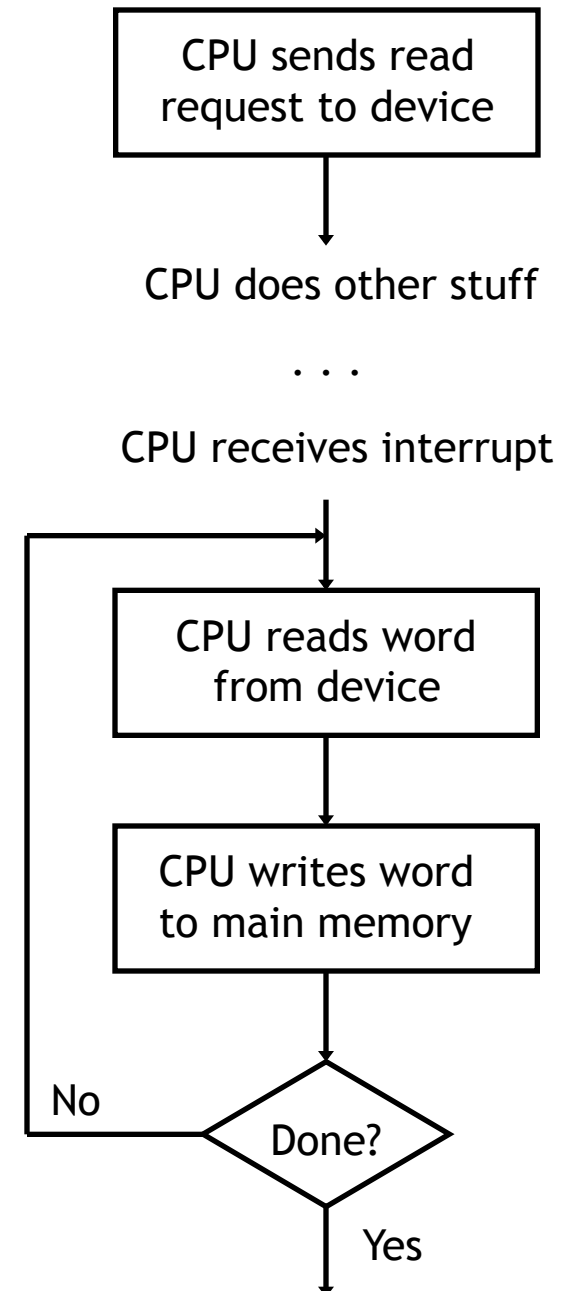
Transferring data with programmed I/O

- How data is transferred?
- With **programmed I/O**, it's all up to the program or the OS
 - CPU makes a request and then waits (loops) until device is ready (loop 1)
 - Buses are typically 32-64 bits wide, so loop 2 is repeated for large transfers
- A lot of CPU time is needed for this!
 - most devices *are* slow compared to CPUs
 - CPU also “wastes time” doing actual data transfer




Interrupt-driven I/O

- Continually checking to see if a device is ready is called **polling**
- Wastes CPU time:
 - CPU must ask repeatedly
 - CPU must ask often enough to ensure that it doesn't miss anything, which means it can't do much else while waiting
- Solution: **Interrupt-driven I/O**
 - Instead of waiting, the CPU continues with other calculations
 - The device **interrupts** the processor when the data is ready
- CPU still does the data transfer





In what ways is Interrupt-driven I/O better than Polling

- A. It has better response time
- B. The CPU can do something else while it is waiting  BEST
- C. It is easier to implement
- D. It makes it possible to wait for multiple things

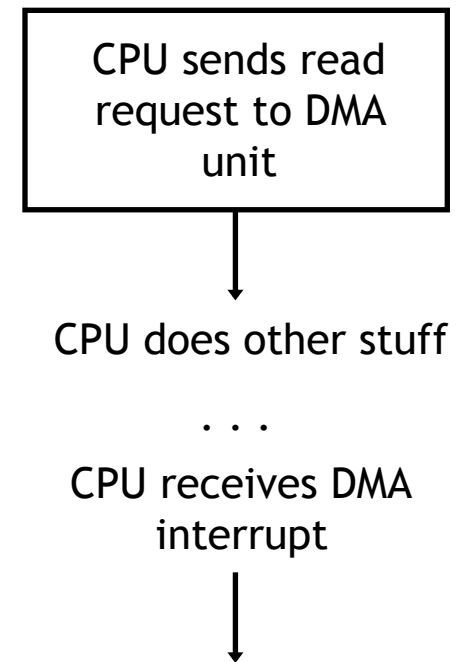

easier

In what ways is Polling better than Interrupt-driven I/O

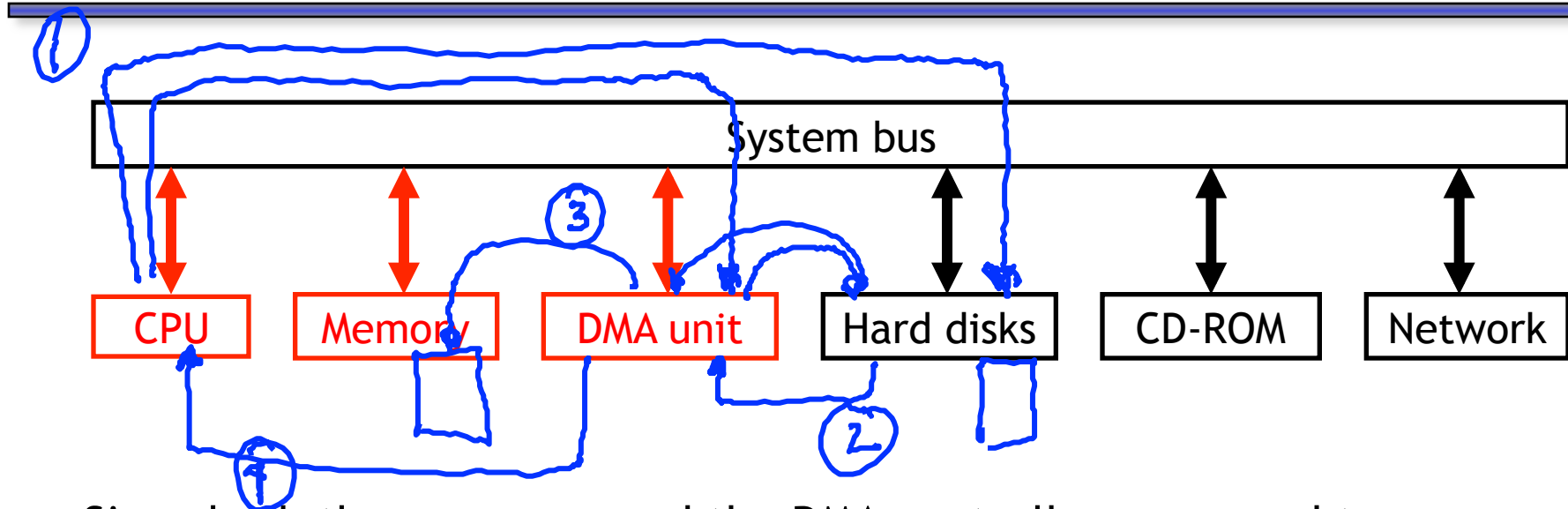
- A. It has better response time 
- B. The CPU can do something else while it is waiting
- C. It is easier to implement 
- D. It makes it possible to wait for multiple things

Direct memory access

- One final method of data transfer is to introduce a **direct memory access**, or **DMA**, controller
- The DMA controller is a simple processor which does most of the functions that the CPU would otherwise have to handle
 - The CPU asks the DMA controller to transfer data between a device and main memory. After that, the CPU can continue with other tasks
 - The DMA controller issues requests to the right I/O device, waits, and manages the transfers between the device and main memory
 - Once finished, the DMA controller interrupts the CPU
- This is yet another form of parallel processing



DMA pictorially



- Since both the processor and the DMA controller may need to access main memory, some form of arbitration is required

Interrupts vs. Exceptions

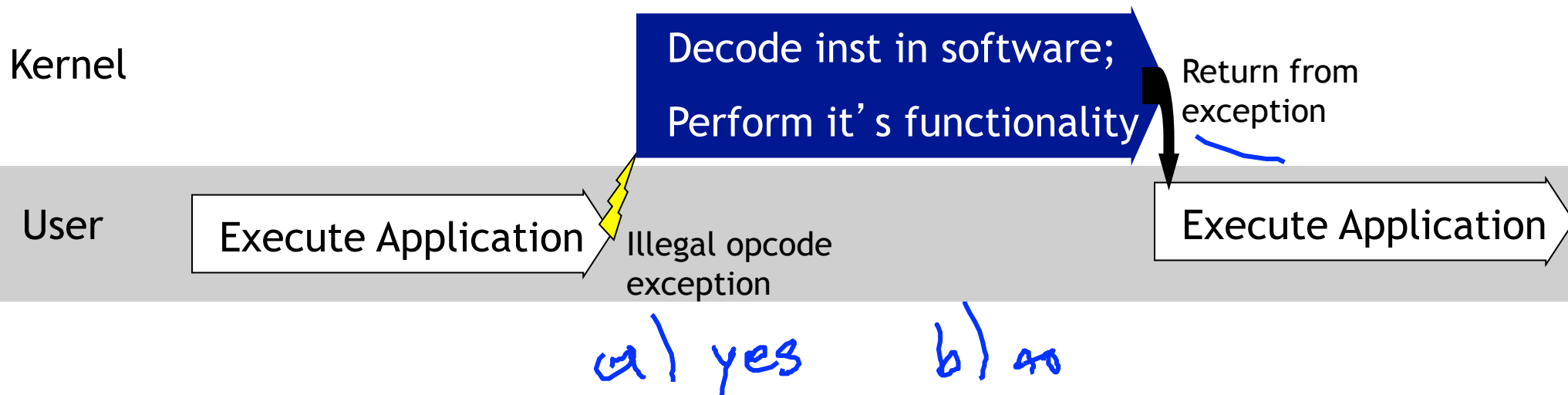
- **Interrupts** are external events that require the processor's attention
 - *Examples:* I/O device needs attention, timer interrupts to mark cycle
- These situations are normal, not errors
 - All interrupts are recoverable: interrupted program will need to be resumed after the interrupt is handled
- OS responsible to do the right thing, such as:
 - Save the current state and shut down the hardware devices
 - Find and load the correct data from the hard disk
 - Transfer data to/from the I/O device, or install drivers
- **Exceptions** are typically errors that are detected within the processor
 - *Examples:* illegal instruction opcode, arithmetic overflow, or attempts to divide by 0
- There are two possible ways of resolving these errors:
 - If the error is **un-recoverable**, the operating system kills the program
 - Less serious problems can often be fixed by OS or the program itself

User handled exceptions

- Sometimes users want to handle their own exceptions:
 - e.g. numerical applications can scale values to avoid floating point overflow/underflow
- Many operating systems provide a mechanism for applications for handling their exceptions
 - Unix lets you register “signal handler” functions
- Modern languages like Java provide programmers with language features to “catch” exceptions (this is much cleaner)

Instruction Emulation: an exception handling example

- Periodically ISA's are extended with new instructions
 - e.g., MMX, SSE, SSE2, etc
- If programs are compiled with these new instructions, they will not run on older implementations (e.g., a 486)
 - “**Forward compatibility**” problem
- Can't change existing hardware, so we add software to “emulate” these instructions



How interrupts/exceptions are handled (Part 1)

- For simplicity exceptions and interrupts are handled the same way
- When an exception/interrupt occurs, we stop execution and transfer control to the operating system, which executes an “interrupt handler” to decide how it should be processed.
- This control transfer to the operating system involves 2 steps:
 1. The current PC (either the excepting instruction or the next instruction to be executed) is saved, so that we can continue executing at that point. On MIPS this is saved into a special register called the “exception program counter” (EPC)
 2. The PC is set to a pre-defined address where the operating system has placed the interrupt/exception handling code.

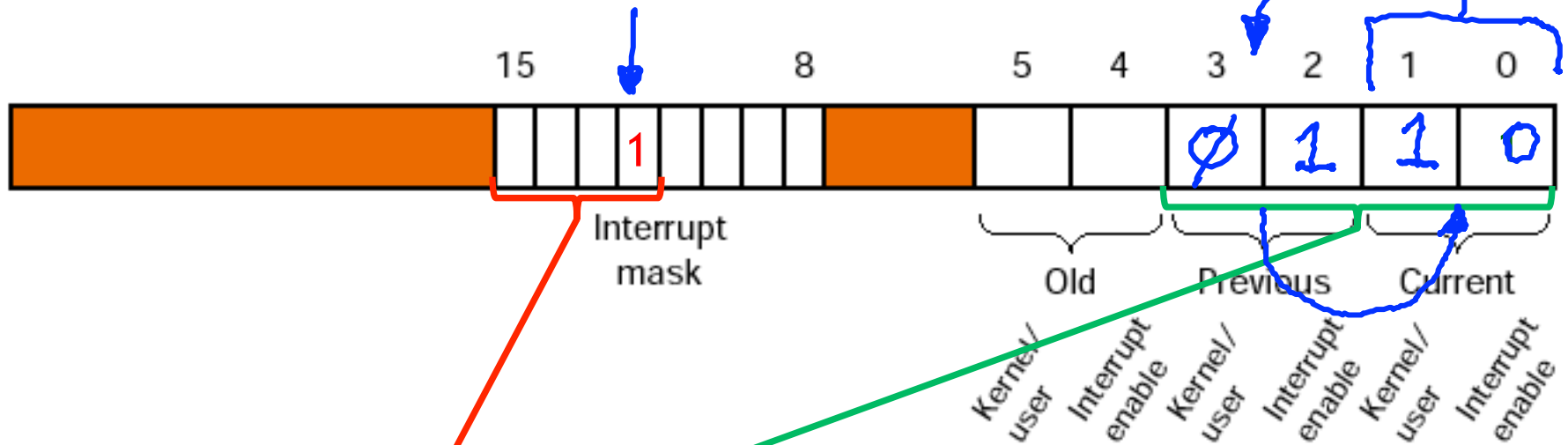
How interrupts/exceptions are handled (Part 2)

- Once in the operating system, the interrupt handler needs to know the *cause* of the interrupt/exception (e.g. overflow, illegal opcode)
 - This is again stored in a special-purpose register called the “cause” register.
- The EPC and Cause register are good examples of interaction between software and hardware, as the cause and current instruction must be supplied to the operating system by the processor.

Make the common case fast

Receiving Interrupts

- To receive interrupts, the software has to enable them
 - MIPS: done by writing to the Status register (on the co-processor)
 - Enable interrupts by setting bit zero
 - Select which interrupts to receive by setting one or more of bits 8-15



```
li    $t0, 0x1001    # enable interrupts and interrupt 12
mtc0  $t0, $12       # set Status register = $12
```

move to co-processor 0

rfe

Handling Interrupts

- When an interrupt occurs, the Cause register indicates which one
 - For an **exception**, the **exception code field** holds the exception type
 - For an **interrupt**, the **exception code field** is **0000** and bits will be set for pending interrupts

