

CPS104
Computer Organization and Programming
Lecture 17: Interrupts and Exceptions

Robert Wagner

Interrupts Exceptions and Traps

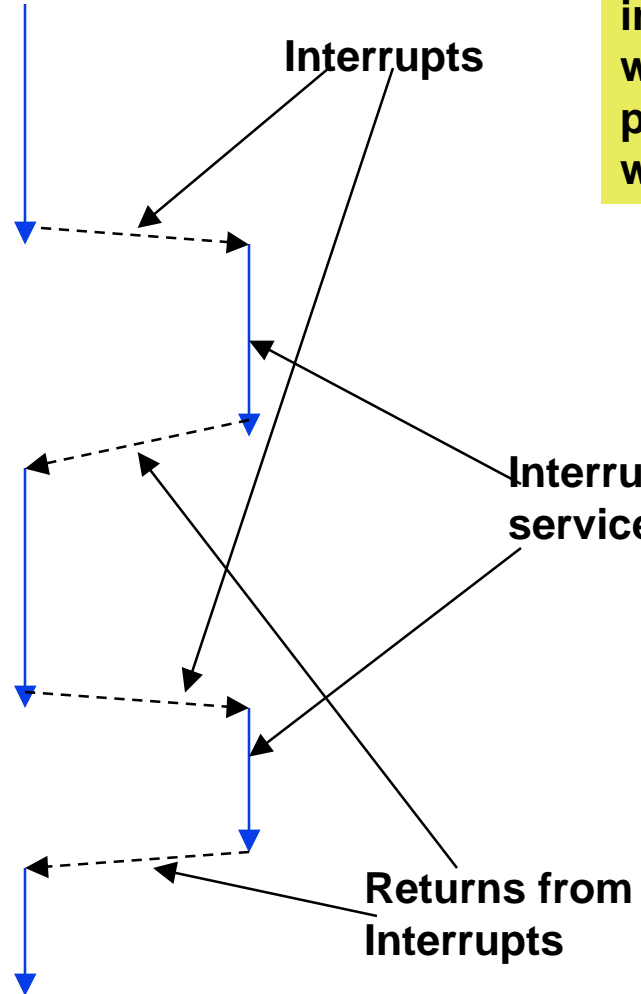
- **Interrupts, Exceptions and Traps** are asynchronous changes in the control flow. Interrupts and Exceptions can be viewed as asynchronous (unscheduled) procedure calls.
- Interrupts and exceptions are designed to provide:
 - ◆ **protection mechanisms:** Error handling, TLB management.
 - ◆ **efficiency:** Overlap I/O and execution, . . .
 - ◆ **Illusion of parallelism:** Timesharing, multithreading
 - ◆ **Ability to handle Asynchronous external events:** Network connections, keyboard input, DMA I/O, . . .

Visualizing an Interrupt

Executing Program



Executing Program



An interrupt service routine can return to the interrupted program, without affecting that program behavior in any way (except speed)

Another service routine could in fact resume a completely different interrupted program – a different “thread”

Interrupts and Exceptions

- **Exception:** a change in execution caused by a condition that occurs **within** the processor.
 - ◆ segmentation fault (access outside program boundaries, illegal access, . . .)
 - ◆ bus error
 - ◆ divide by 0
 - ◆ overflow
 - ◆ page fault (virtual memory...)
- **Interrupt:** a change in execution caused by an external event
 - ◆ devices: disk, network, keyboard, etc.
 - ◆ clock for timesharing (multitasking)
 - ◆ These are useful events, must do something when they occur.
- **Trap:** a user-requested exception
 - ◆ Operating system call (syscall)
 - ◆ Breakpoints (debugging mode)

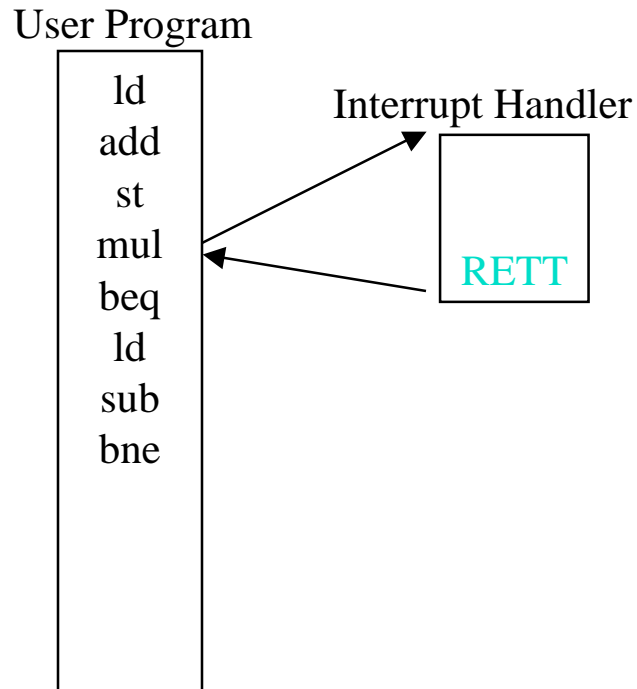
An Execution Context

- The state of the CPU associated with a thread of control (process)
 - ◆ general purpose registers (integer and floating point)
 - ◆ status registers (e.g., condition codes)
 - ◆ program counter, stack pointer
- Need to be able to switch between contexts
 - ◆ **timesharing**: sharing the machine among many processes
 - ◆ better utilization of machine (overlap I/O of one process with computation of another)
 - ◆ different **modes** (Kernel v.s. user)
- Maintained by operating system

Context Switches

- **Save current execution context**
 - ◆ **Save registers and program counter**
 - ◆ **information about the context (e.g., ready, blocked)**
- **Restore other context**
- **Need data structures in kernel to support this**
 - ◆ **process control block**
- **Why do we context switch?**
 - ◆ **Timesharing: HW clock tick**
 - ◆ **I/O begin and/or end**
- **What causes these events to occur?**
 - ◆ **Interrupts...**
- **When do they occur?**
 - ◆ **BETWEEN INSTRUCTIONS**

Handling an Exception



- The **same mechanism** is used to handle interrupts, exceptions, traps.

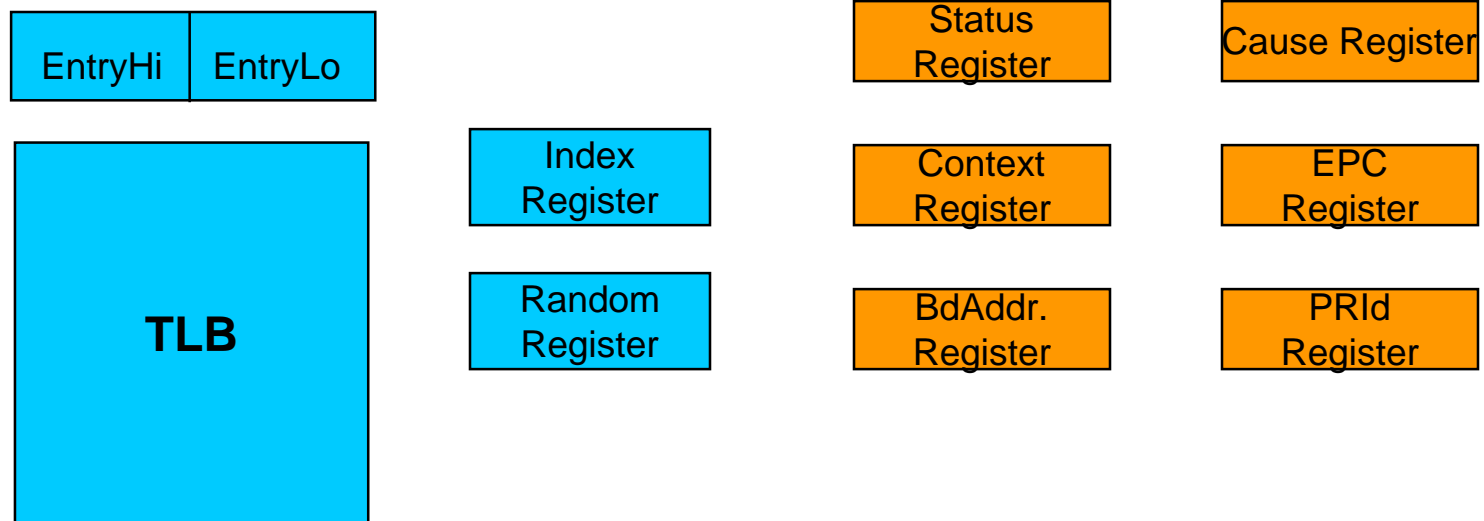
- Invoke specific kernel routine based on type of interrupt
 - interrupt/exception handler
- Must determine what caused interrupt
 - **Cause register** holds information on what caused the interrupt
 - $PC \leftarrow interrupt_handler$
- Vectored Interrupts
 - $PC = interrupt_table[i]$
- Clear the interrupt
- Return from interrupt (RETT) to different process (e.g, context switch)

Execution Mode

- What if interrupt occurs while in interrupt handler?
 - ◆ *Problem*: Could lose information for one interrupt clear of interrupt #1, clears both #1 and #2
 - ◆ *Solution*: **disable interrupts**
- Disabling interrupts is a protected operation
 - ◆ Only the kernel can execute it
 - ◆ user v.s. kernel mode
 - ◆ **mode bit in CPU status register**
- Other protected operations
 - ◆ installing interrupt handlers
 - ◆ manipulating CPU state (saving/restoring status registers)
 - ◆ Updating TLB, Changing page table, changing page protection.
- Changing modes
 - ◆ interrupts
 - ◆ system calls (syscall instruction)

Interrupts and exceptions in the MIPS 2000

- The MIPS has a special coprocessor **CP0**, that is dedicated to Exception handling.



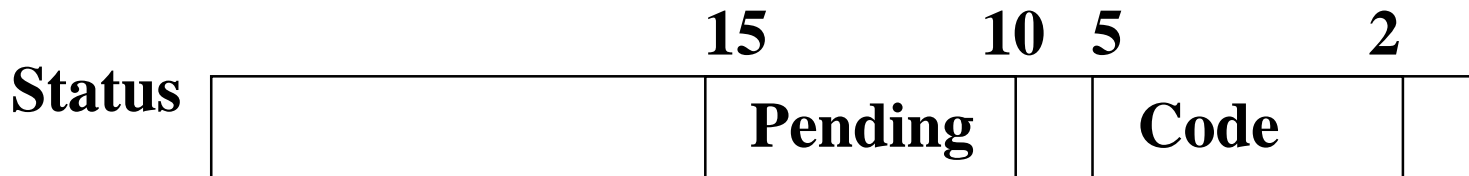
MIPS-2000 CP0

- **EPC** (Exception Program Counter): a 32-bit register used to hold the address of the affected instruction
 - ◆ register 14 of coprocessor 0
- **Cause**: a register used to record the cause of the exception.
 - ◆ In the MIPS architecture this register is 32 bits, though some bits are currently unused.
- **BadVAddr**: register contains memory address at which memory reference occurred
 - ◆ when memory access exception occurs
 - ◆ register 8 of coprocessor 0
- **Status**: interrupt mask and enable bits
 - ◆ register 12 of coprocessor 0
- **PRId**: Processor Revision Identifier register

Additions to MIPS ISA

- Special coprocessor instructions:
 - ◆ `mfc0 rt, rd` # register `rd` in CP0 is loaded into register `rt`
 - ◆ `mtc0 rt, rd` # register `rt` is loaded into CP0 register `rd`
 - ◆ `rfe` # Return from exception, restore Interrupt mask and status register, (allow change in execution mode).
- TLB special instructions:
 - ◆ `tlbp` # The index register is loaded with the address of the TLB entry whose contents match the contents of `EntryHi` and `EntryLo` registers.
 - ◆ `tlbr` # Read Indexed TLB entry: The `EntryHi` and `EntryLo` registers are loaded with the content of the TLB pointed at by the TLB `Index` register
 - ◆ `tlbwi` # Write Indexed TLB entry: The `EntryHi` and `EntryLo` registers are stored in the TLB entry pointed at by the TLB `Index` register
 - ◆ `tlbwr` # Write random TLB entry: The `EntryHi` and `EntryLo` registers are stored in the TLB entry pointed at by the TLB `random` register

Details of Cause register



- **Pending interrupt** 5 hardware levels: bit set if interrupt occurs but not yet serviced
 - ◆ handles cases when more than one interrupt occurs at same time, or records interrupt requests when interrupts disabled
- **Exception Code** encodes reasons for interrupt
 - ◆ 0 (INT) => external interrupt
 - ◆ 4 (ADDRL) => address error exception (load or instr fetch)
 - ◆ 5 (ADDRS) => address error exception (store)
 - ◆ 6 (IBUS) => bus error on instruction fetch
 - ◆ 7 (DBUS) => bus error on data fetch
 - ◆ 8 (Syscall) => Syscall exception
 - ◆ 9 (BKPT) => Breakpoint exception
 - ◆ 10 (RI) => Reserved Instruction exception
 - ◆ 12 (OVF) => Arithmetic overflow exception

How are Exceptions Handled?

- **Machine must save the address of the offending instruction in the EPC (Exception Program Counter)**
- **Then transfer control to the Operating System (OS) at some specified address**
- **OS performs some action in response**
- **OS terminates program or returns (eventually) using EPC**
 - ◆ **could return to different program (context switch)**
- **Exceptions are an “unscheduled procedure call” to a fixed location!**

How are Exceptions Handled?

- 2 types of exceptions in our current implementation
 - ◆ undefined instruction
 - ◆ arithmetic overflow
- Which Event caused Exception?
 - ◆ Option 1 (used by MIPS): **Cause register** contains reason
 - ◆ Option 2 Vectored interrupts: cause is part of the address of the subroutine called.
 - ✦ addresses separated by 32 instructions
 - ✦ E.g.,
 - ◆

<u>Exception Type</u>	<u>Exception Vector Address</u>
Undefined instruction	C0 00 00 00
Arithmetic overflow	C0 00 00 20

 - ✦ Jump to appropriate address

What happens to Instruction with Exception?

- **Some problems could occur in the way the exceptions are handled.**
- **Example:**
 - ◆ **in the case of arithmetic overflow, the instruction causing the overflow completes writing its result, because the overflow branch is in the state when the write completes.**
 - ◆ **However, the architecture may define the instruction as having no effect if the instruction causes an exception; MIPS specifies this.**
- **Virtual memory exceptions must prevent the interrupt handler from changing the machine state.**
- **This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard**

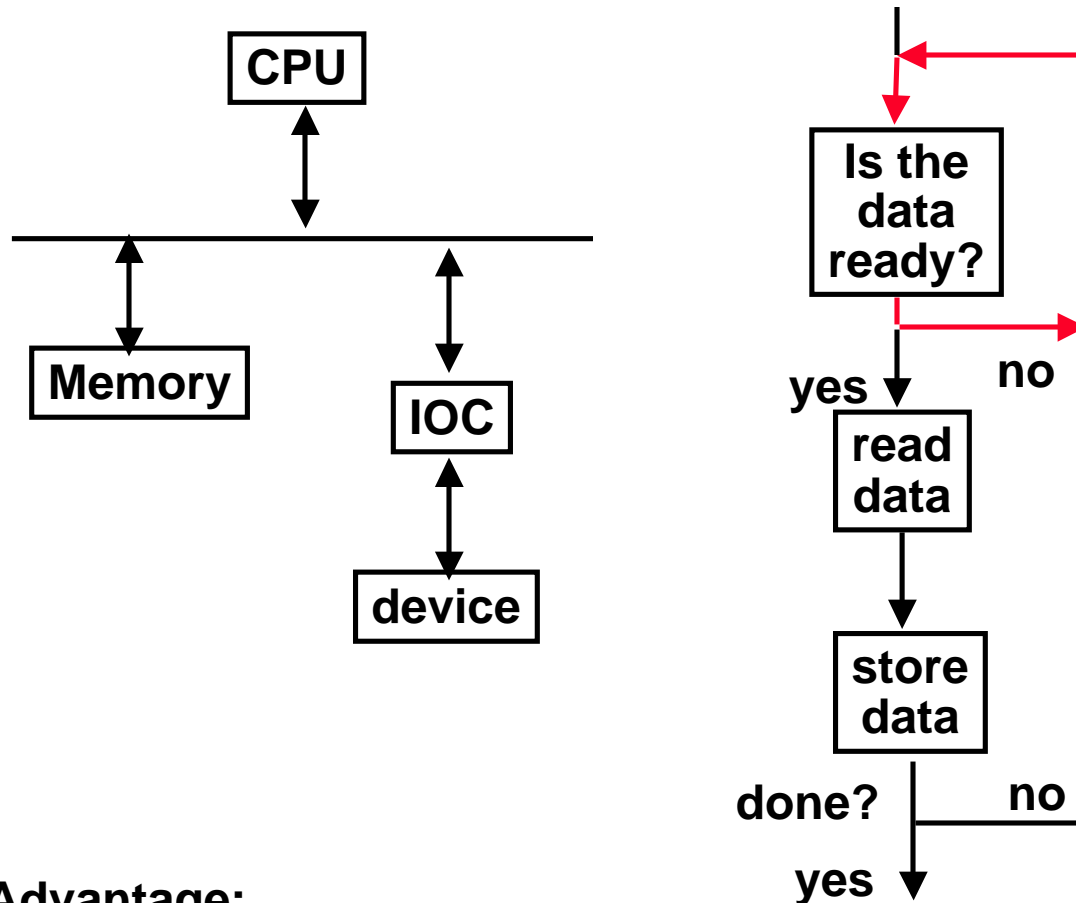
I/O, The OS, and Interrupts

- **Input output devices are SHARED across**
 - ◆ **all user programs running at same time**
 - ◆ **The OS, which switches between user programs**
- **The OS uses Interrupts to control**
 - ◆ **Access to I/O devices**
 - ◆ **Context switching between user processes**
 - ❧ **So processes act as if they run “in parallel”**
 - ❧ **So users can “timeshare” access to computer**

OS and I/O Systems Communication Requirements

- **The Operating System must be able to prevent:**
 - ◆ **The user program from communicating with the I/O device directly**
- **If user programs could perform I/O directly:**
 - ◆ **Protection to the shared I/O resources could not be provided**
- **Three types of communication are required:**
 - ◆ **The OS must be able to give commands to the I/O devices**
 - ◆ **The I/O device must be able to notify the OS when the I/O device has completed an operation or has encountered an error**
 - ◆ **Data must be transferred between memory and an I/O device**

Polling: Programmed I/O

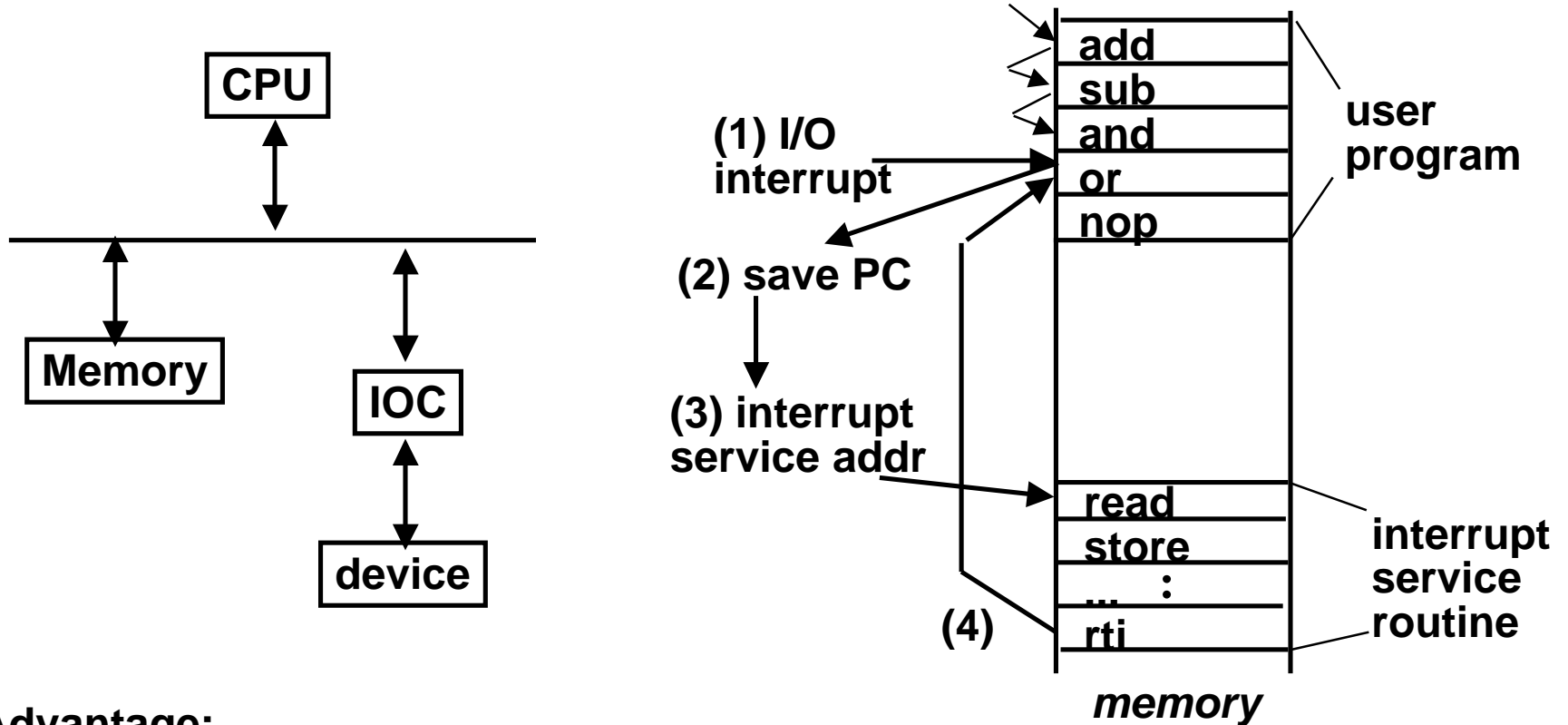


busy wait loop
not an efficient
way to use the CPU
unless the device
is very fast!

but checks for I/O
completion can be
dispersed among
computation
intensive code

- Advantage:
 - ◆ Simple: the processor is totally in control and does all the work
- Disadvantage:
 - ◆ Polling overhead can consume a lot of CPU time

Interrupt Driven Data Transfer



- **Advantage:**
 - ◆ User program progress is only halted during actual transfer
- **Disadvantage, special hardware is needed to:**
 - ◆ Cause an interrupt (I/O device)
 - ◆ Detect an interrupt (processor)
 - ◆ Save the proper states to resume after the interrupt (processor)

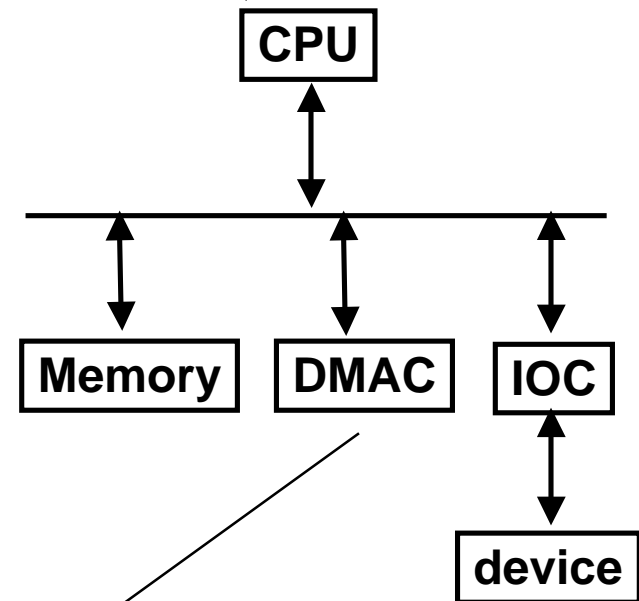
I/O Interrupt

- **An I/O interrupt is just like an exception except:**
 - ◆ **An I/O interrupt is asynchronous (could happen at any time)**
 - ◆ **Further information needs to be conveyed (what device interrupted)**
- **An I/O interrupt is asynchronous with respect to instruction execution:**
 - ◆ **I/O interrupt is not associated with any instruction**
 - ◆ **I/O interrupt does not prevent any instruction from completion**
 - ✎ **You can pick your own convenient point to take an interrupt**
- **I/O interrupt is more complicated than exception:**
 - ◆ **Needs to convey the identity of the device generating the interrupt**
 - ◆ **Interrupt requests can have different urgencies:**
 - ✎ **Interrupt request needs to be prioritized**

Delegating I/O Responsibility from the CPU: DMA

- **Direct Memory Access (DMA):**
 - External to the CPU
 - Act as a master on the bus
 - Transfer blocks of data to or from memory without CPU intervention

CPU sends a starting address, direction, and length count to DMAC. Then issues "start".



DMAC provides handshake signals for Peripheral Controller, and Memory Addresses and handshake signals for Memory.