Last lecture we looked at polling and DMA as a way for the CPU and I/O to coordinate their actions. Polling is simple but it is inefficient since the CPU typically asks many times if the device is ready before the device is indeed ready.

Interrupts

Another method for the I/O device to gain control of the bus is an *interrupt* request. Interrupts are similar to DMA bus requests in some ways, but there are important differences. With a DMA bus request, the DMA device asks the CPU to disconnect itself from the system bus so that the DMA device can use it. The purpose of an interrupt is different. When an I/O device makes an interrupt request, it is asking the CPU to take specific action, namely to run a specific kernel program: an *interrupt handler*. Think of DMA as saying to the CPU, "Can you get off the bus so that I can use it?," whereas an interrupt says to the CPU "Can you stop what you are doing and instead do something for me?"

Interrupts can occur from both input and output devices. An extreme example of input device interrupts is the Ctl-Alt-Del sequence on an MS Windows operating system. A more typical example is a mouse click or drag or a keyboard press. Output interrupts can also occur e.g. when an printer runs out of paper, it tells the CPU so that the CPU can send a message to the user e.g. via the console.

There are several questions about interrupts that we need to examine:

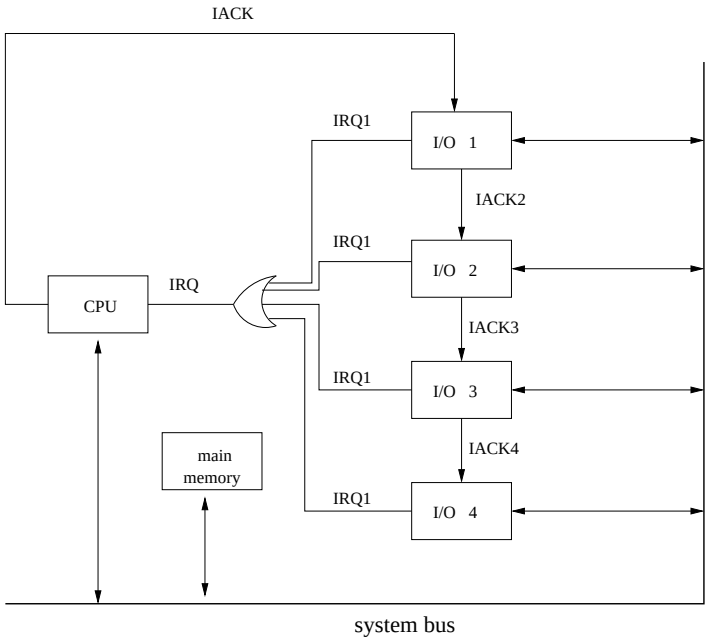
- how does an I/O device make an interrupt request?
- how are interrupt requests from multiple I/O devices coordinated?
- what happens from the CPU perspective when an I/O device makes an interrupt request ?

The mechanism by which an I/O device makes an interrupt request is similar to what we saw in DMA with bus request and bus granted. The I/O device makes an *interrupt request* using a control signal commonly called IRQ. The I/O device sets IRQ to 1. If the CPU does not ignore the interrupt request (under certain situations, the CPU does ignore interrupt requests), then the CPU sets a control signal IACK to 1, where IACK stands for *interrupt acknowledge*. The CPU also stops writing on the system bus, by setting its tristate gates to off. The I/O device then observes that IACK is 1, which means that it can write on the system bus.

Often there is more than one I/O device, and so there is more than one type of interrupt request than can occur. One could have a separate IRQ and IACK line for each I/O device. This requires a large number of dedicated lines and places a burden on the CPU in administering all these lines. Another method is to have the I/O devices all share the IRQ line to the CPU. They could all feed a line into one big OR gate. If any I/O device requests an interrupt, then the output of the OR gate would be 1. How then would the CPU decide whether the allow the interrupt. One way is for the CPU to ask each I/O device one by one whether it requested the interrupt, but using the system bus. It could address each I/O device and ask "did you request the interrupt?" Each I/O device would then have one system bus cycle to answer yes. This is just another form of polling.

Daisy chaining

A more elegant method is to have the I/O devices coordinate who gets to interrupt the CPU at any time. Suppose the I/O devices have a *priority ordering*, such that a lower priority device cannot interrupt the CPU when a higher priority device is currently interrupting the CPU. This can be implemented with a classical method known as *daisy chaining*. As mentioned above, the IRQ lines from each device meet at a single OR gate, and the output of this OR gate is sent to the CPU as a single IRQ line. The I/O devices are then *physically ordered* by priority. Each I/O device would have an IACKin and IACKout line. The IACKout line of one I/O device is the IACKin line of the next lower priority I/O device. The IACKin line of the highest priority I/O device would be the IACK line from the CPU. There would be no IACKout line from the lowest priority device.



Here is how daisy chaining works. At any time, any I/O device can interrupt the CPU by setting its IRQ line to 1. The CPU can acknowledge that it has received an interrupt request or it can ignore it. To acknowledge the interrupt request, it sets IACK to 1. The IACK signal gets sent to the highest priority I/O device.

Suppose that an interrupt request is made and the CPU sets the IACK line to 1. If the highest priority device had requested the interrupt, then it sets IACKout to 0. Otherwise, it sets IACKout to 1 i.e. passing the CPU's IACK down to the second highest priority device. Each device does the same thing. Whenever IACKin switches from 0 to 1, the device either sets IACKout = 0 (if this device requested an interrupt) or it sets IACKout = 1 (if it did not request an interrupt).

Let me explain the last sentence in more detail. I said that the I/O device has to see IACKin switch from 0 to 1. That is, it has to see that the CPU previously *was not* acknowledging an interrupt, but now *is* acknowledging an interrupt. This condition (observing the transition from 0 to 1) is used to prevent two I/O devices from simultaneously getting write access to the system bus.

What if a lower priority device is allowed to interrupt the CPU but then, a short time later, a higher priority device wishes to interrupt the CPU (while the CPU is still processing the lower order

interrupt). With daisy chaining, the IRQ from the higher priority device cannot be distinguished from that of the lower priority device since both feed into the same OR gate. Instead, the higher priority device has to kill the interrupt from the lower priority device first. It does so by changing its own IACKout signal to 0. This 0 value gets passed on to the lower priority device. The lower priority device doesn't know why its IACKin was set to 0, but it doesn't need to know. It just needs to know that its interrupt time is now over. It finishes up as quickly as it can, and then sets its IRQ to 0, at which point the CPU sets its IACK to 0. Once the higher priority device sees that the CPU has set IACK to 0, it can then make its interrupt request.

How does the CPU know which device made the interrupt request? When CPU sets IACK to 1, it also frees up the system bus. (It "tristates itself.") When the I/O device that made the interrupt request observes the IACK 0-to-1 transition, this device then identifies itself by writing its address (or some other identifier) on the system bus. The CPU reads in this address and takes the appropriate action. For example, if the device has a low priority, then the CPU may decide to ignore the interrupt and immediately set IACK to 0 again.

Think of the sequence as follows. "Knock knock" (IRQ 0-to-1). "Who's there and what do you want?" (IACK 0-to-1) and CPU tristates from system so that it can listen to the answer. "I/0 device number 6 and I want you to blablabla" (written by I/O device onto system bus). If CPU then sets IACK 1-to-0, this means that it won't service this request. In this case, the I/O device has to try again later. If the CPU doesn't set IACK 1-to-0, then it may send back a message on the system bus. (The I/O device needs to tri-state after making its request, so that it can listen to the CPU's response.)

Daisy chaining can be used for DMA as well. Rather than talking about an IRQ and IACK signals, we talk about bus request (BR) and bus granted (BG) signals, respectively. You can take the daisy chaining diagram above and replace IRQ by BR and replace IACK by BG.

Interrupt handler

When a user program is running and an interrupt occurs, the current process branches to the exception handler, in MIPS located at 0x80000080 i.e. in the kernel. The kernel then examines the `Cause` and `Status` registers to see what caused the exception, examines the interrupt enable bits to see if it should accept this interruptible and, if so, then branches to the appropriate exception/interrupt handler. (Note that an interrupt is just another kind of exception.)

Because there is a jump to another piece of code, interrupts are reminiscent of function calls. With functions, the caller needs to store certain registers on a stack so that when the callee returns, these registers have their correct values. Similarly, when an interrupt (more generally, an exception) occurs, the kernel needs to save values in registers ($0-$31, $f0-$f31, EPC, PC, Status, etc) that are being used by that process. Several issues arise.

First, the kernel disables other interrupts. (This will be explained briefly below, and you will learn alot more about it in COMP 310.) Second, the kernel saves certain key values that will allow it to return safely to program that was interrupted. Where does the kernel save these values? The kernel maintains a (data) structure for each process that keeps track of the administrative information for that process. This includes the page table, as well as information about the history of the process. For processes that are temporarily halted, it also stores values in each of the registers in use. This process data structure is in the kernel's area (above address 0x80000000). Once these values have been saved, the kernel changes its *interrupt enable* state to allow some limited set of

interrupts to occur.

Note that interrupts can be nested. This is reminiscent of recursive functions, but we need to be more careful here since interrupts can occur *at any time* e.g. while the interrupt handler is in the middle of saving the registers from the previous interrupt.

To avoid problems, and to ensure that the kernel can prioritize its work, a *priority* ordering is imposed on the various kernel routines and on possible interrupts. For example when saving the state information for a process, it would be bad for the kernel to be interrupted and to jump to yet another interrupt handler. So, for certain kernel routines, the processor can put itself into a non-interruptable state. Another example is that a lower priority I/O device should not be allowed to interrupt the interrupt handler of a higher priority device, whereas a higher priority device should be allowed to interrupt a lower priority device's interrupt handler.

In summary, here's the basic idea of what the kernel (interrupt handler) needs to do:

1. Disable all interrupts. (Don't acknowledge any interrupts.)
2. Save the process state (registers, etc).
3. Enable higher priority interrupts.
4. Attend to the interrupt.
5. Restore process state (restore values in registers)
6. Return from interrupt, and reset previous interrupt enable level

Finally, in the slides, I gave some details about MIPS, namely which bits in the Cause and Status registers serve which function. I also described how to test particular bits within a register. For example, to set all bits in the Cause register to 0 except for the bits that encode the cause of the exception, you could run:

```
mfc0  $k0, $13          # the Cause register is $13 in coprocessor 0
andi  $k0, $k0, 0x3c    # mask $k0 with 00000000000000000000000000111100
```

The register `$k0` is register 26 in the main register array.

Another example was to set certain interrupt enable (IE) bits in the Status register.

```
mfc0  $k0, $12          # the Status register is $12 in coprocessor 0
ori   $k0, $k0, 0x01    # turns on the LSB which is an IE bit
```

To disable this interrupt, we could turn off this bit.

```
mfc0  $k0, $12
lui  $1,      0xffff
ori  $1,  $1, 0xfffe    # sets a mask that is all 1's except LSB
and  $k0, $k0, $1       # turns off the LSB
mtc0 $12, $k0
```

Process scheduling

There is a timer in the CPU which keeps track of how many clock cycles the current process (or kernel function) is using. When this timer reaches 0, an interrupt occurs. One of the causes of an exception is this scenario. So when the timer is 0, the kernel detects that it is time for another process to have a chance at running. As described above, the state of the process is saved as part of a data structure (in the kernel's memory). Then, the kernel chooses another process to run. *This scheduling problem and other related issues such as whether an exception handler can be interrupted are treated in depth in COMP 310.*

More on exceptions

page fault

Here I return to an topic we discussed in lectures 17 to 19. There we examined the mechanism of a TLB access and cache access and considered what happens if there is a TLB miss or cache miss. We also examined what happens when main memory does not contain a desired word (page fault). All of these cause exceptions, and hence a jump to the kernel. Let's quickly review these, adding in a few details of how the system bus is managed.

When there is a TLB miss, the TLB miss handler loads the appropriate entry from the appropriate page table, and examines this entry. There are two cases: either the page valid bit in that entry is 1 or the page valid bit is 0. The value of the bit indicates whether the page is in main memory or not.

If the pagevalid bit is 1, then the page is in main memory. The TLB miss handler copies the translation from the page table to the TLB, sets the validTLB bit, and then control is given back to the user program. (This program again accesses the TLB as it tried to do before and now the virtual $\rightarrow$ physical translation is present, so there is a TLB hit.)

We now understand that, for the TLB miss handler to get the translation from the page table, it needs to access the system bus. This is possible provided the system bus is free. It may happen, however, that the system bus is busy (e.g. it is being used by a DMA device). In this case, the kernel may decide to stop the process, rather than taking over the system bus.

If the pagevalid bit is 0, then a page fault occurs. The page is not in main memory and it needs to be copied from the hard disk to main memory (very slow). The TLB miss handler jumps to the page fault handler, which brings the desired page in main memory.[1]

We now understand that the page fault handler tells the hard disk controller (a DMA device) which page(s) should be moved, and where. The page fault handler then pauses (and saves its state), and the kernel switches to some other process.... Later, when the hard disk (DMA) controller has transfered the page to main memory, it sends an interrupt request to the CPU. The page fault handler can then continue from where it left off, namely it can update the page table. When it is done updating the page table, , the page fault handler then jumps back to the TLB miss handler. Now, the requested page is in main memory and the pagevalid bit is on, so the TLB miss handler can copy the page table entry into the TLB and return control to the original process.

[1] We ignore the step that it also swaps a page out of main memory.

pipelining

One last consideration is about pipelining. Other than interrupts (which can occur at any time), which exceptions can occur at each stage of the pipeline?

- IF - page fault, memory protection violation (e.g. user asks for kernel address, fetched address is to user data area)
- ID - non-valid op code (some opcodes are not used)
- ALU - arithmetic exception (e.g. integer overflow)
- MEM - page fault, memory protection violation
- WB - none

What happens in a pipeline when there is an exception? For example, suppose there is a page fault at the MEM stage. In this case, the instruction just ahead in the pipeline would be allowed to finish before the process is paused. The address for the instruction in the MEM stage should then be saved in the kernel, along with the state of the process, namely the values of all the registers. Before the process resumes (after the page fault has been serviced), the registers must reloaded. Finally, the PC is reloaded, the instruction is re-fetched and it begins again at the beginning of the pipeline.