In this lecture, I will elaborate on some of the details of the past few weeks, and attempt to pull some threads together.

System Bus and Memory (cache, RAM, HDD)

We first return to an topic we discussed in lectures 16 to 18. 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). These events cause exceptions, and hence a jump to the kernel where they are handled by exception handlers. Let's quickly review these events, adding in a few details of how the system bus is managed and used.

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 → physical translation is present: a TLB hit!)

From the discussion of the system bus last lecture, we now understand that, for the TLB miss handler to get the translation from the page table, the CPU needs to access the system bus. The page table entry is retrieved from the page table in main memory, and brought over the CPU where it can be analyzed. (Details on how that is done are not unspecified here.)

What if the pagevalid bit was 0? In that case, a page fault occurs. That is, 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 arranges that the the desired page is brought into main memory.[1]

After the discussion last lecture, we now understand that the page fault handler tells the hard disk controller (a DMA device) which page should be moved, and to where. The page fault handler then pauses the process and saves the process state, and the kernel switches to some other process. The disk controller meanwhile starts getting the page off the hard disk. Once it has done so, it will need to use the system bus to transfer the page to RAM. (Recall how DMA works.) After it has finished moving the page to RAM, it sends an interrupt request (see below) to the CPU and tells the CPU that it is done.

The CPU will eventually allow the process to continue. The page fault handler will need to update the page table to indicate that the new page is indeed there. It will then jump 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. *Again, note that these main memory accesses require the system bus.*

[ASIDE: I did not fill in all these details in the lecture slides, but rather gave only a tree diagram summarizing the steps. I am not expecting you to memorize all the details above. Just understand the steps and the order.]

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

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. Today we'll look at another way for devices to share the system bus, called interrupts.

Interrupts

With interrupts, an I/O device gains control of the bus by making 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. A 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 ?

I'll deal with these questions one by one.

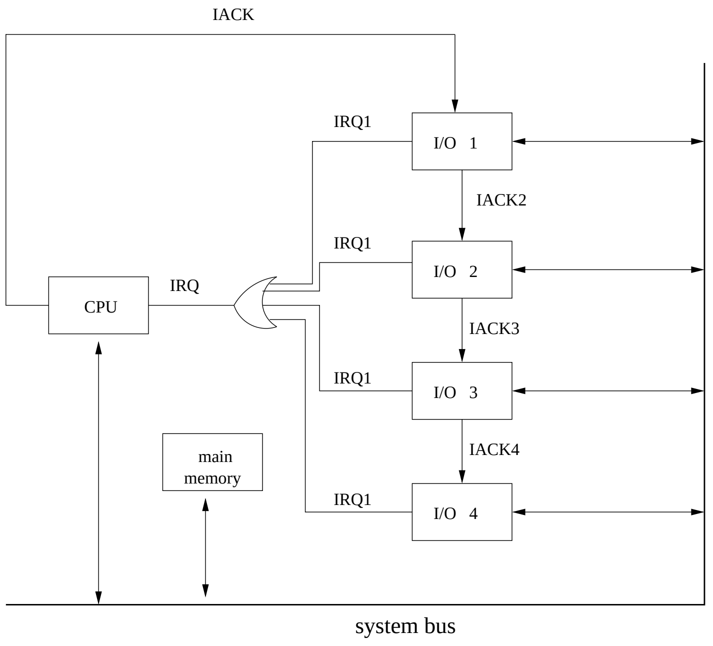
The mechanism by which an I/O device makes an interrupt request is similar to what we saw in DMA with bus request and bus granted. The I/O device makes an *interrupt request* using a control signal commonly called IRQ. The I/O device sets IRQ to 1. If the CPU does not ignore the interrupt request (under 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 use 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 the burden on the CPU in administering all these lines. THis is not such a problem, in fact, and many modern processors do use point to point connections betwen components. 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 use the system bus 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 a form of polling (although there is no infinite loop here as there was when I discussed polling last lecture).

Daisy chaining

A more sophisticated 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 the CPU is servicing the interrupt request from a higher priority device. This can be implemented with a classical method known as *daisy chaining*. As described 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 IACKin line of the highest priority I/O device would be the IACK line from the CPU. The IACKout line of one I/O device is the IACKin line of the next lower priority I/O device. 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.

Let's start by supposing that all IRQ and IACK signals are 0. Now suppose that an interrupt request is made (IRQ = 1) by some device, and the CPU responds by setting the IACK line to 1. If the highest priority device had requested the interrupt, then it leaves its IACKout at 0. Otherwise, it sets IACKout to 1 i.e. passing the CPU's IACK=1 signal down to the second highest priority device. That device does the same thing, and so on. Whenever IACKin switches from 0 to 1, the device either leaves its 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 the CPU is servicing the interrupt request from a lower priority device, and a higher priority device wishes to interrupt the CPU. 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 changes 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 just set to 0. It could be because a higher priority device wants to make an interrupt request, or it could be because the CPU has turned off its IACK signal. Whichever of these occurs doesn't matter. The lower priority device 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. Once it is done, 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 is just one way of coordinating interrupt requests. But it is not the only way, and often the CPU does the work of figuring out which interrupt request to handle based on priorities. Let's now put daisy chaining aside and think about interrupts from the CPU perspective.

Interrupt and exception handlers (kernel)

When a user program is running and an interrupt request occurs and the CPU has enabled interrupts, the current process branches to the exception handler, which in MIPS is located at 0x80000080 i.e. in the kernel. The kernel then examines the `Cause` and `Status` registers to see what caused the exception. It can determine that an interrupt request occurred from the `Cause` register. It then examines the interrupt enable bits to see if it should accept this interrupt and, if so, it branches to the appropriate exception handler. (Note that an interrupt is just another kind of exception.)

Because there is a jump to another part of code, interrupts are similar to 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.

Note that interrupts can be nested. This is reminiscent of recursive functions. However, the situation is slightly different here since interrupt requests can occur *at any time*, including while the interrupt handler is in the middle of saving the registers from the previous interrupt. 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. Service the interrupt.
5. Restore process state (restore values in registers)
6. Return from interrupt, and reset previous interrupt enable level

Interrupts in MIPS

Let's look at a few details about how MIPS treats interrupts, and how the Cause and Status registers are used.

The general interrupt enable bit is the LSB of the Status register. To turn this bit on, we use:

```
mfc0  $k0, $12          # the Status register is $12 in coprocessor 0
ori   $k0, $k0, 0x01    # turns on the LSB which is an IE bit
mtc0  $12,  $k0
```

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
```

Note that this code uses the register `$k0` which is register 26 in the main register array. User programs are not allowed to use this register. Only kernel programs can use it. (And error will occur if a user instruction uses this register.)

Finally, there are particular bits in the Status register which indicate whether interrupts of a particular priority are enabled, and there are corresponding bits in the Cause register which turn on when an interrupt request of particular priority has been made. The kernel would need to compare bits in these two registers to decide if a particular interrupt request should be serviced. (Detailed omitted.)