Chris Thomas
10/24/2022
HW 3

1) General exception handling and Multiple Interrupts handled by one ISR.
a) How can the service routine invoked to handle general exceptions determine what exception it is handling?

The EXCCODE is used by the general exception handler (at offset 0X180) to determine what event caused the exception and what service is needed.

b) On some PIC32 processors, there are more IRQs than vectors. Even assuming the EIC is in multi-vector mode, this means that some ISRs must handle multiple IRQs. However, in this case, the IRQs are generally related and there would never be more than 3 IRQs handled by one service routine. For example, the PIC32MX UART transmit, receive and Error IRQs for a given UART all share the same vector. (Note that these related IRQs will not be assigned different priorities because PIC processors assign priority by vector.)
How can a service routine handling a few related IRQs best determine what IRQ caused the interrupt in order to determine what service is needed?

Check the interrupt service flag.

c) Now, assume that the EIC is in single-vector mode. How can the interrupt service routine determine which of several interrupts it should be handling (or handle first assuming there are multiple simultaneous interrupts)? In this case there is potentially many more IRQs that need to be handled by the single service routine than there were in part b. There are two different ways this can be done. Give both. Note that one way is related to the answer for part b, but there are some additional things that must be considered in this case.

The SRIPL bits (Requested Priority Level bits for Single Vector Mode bits) determine the priority level of the latest interrupt presented to the CPU.

Check the interrupt service flag and check that the IEC is cleared.

d) In the PIC32MZ, each IRQ, even related IRQ's all get their own vector. Suppose a programmer wanted to use multivector mode but also wanted to handle two related

IRQs, specifically IRQs 53 and 54, for one particular device in one ISR. How could this be accomplished in the PIC32MZ? There are two ways this can be done, one involves using a dispatch function and one does not.
i) Show how to use #pragma interrupt statements so that an interrupt on IRQ 53 will go directly to the combined ISR in the vector table and IRQ 54 will have a dispatch function in its entry in the vector table that jumps to the combined ISR.

#pragma interrupt IRQ543_ISR IPL5SOFT vector @53, 54

ii) Show how to arrange things so that neither IRQ needs a dispatch function to get to the combined ISR in the vector table. Give the modified #pragma interrupt statement and a line of code that would go prior to the asm("ei") in the main.

#pragma interrupt ISR_TEST2 IPL5SRS vector @53
In main: PRISS = (5 << _PRISS_PRI5SS_POSITION) & _PRISS_PRI5SS_MASK;

2) Interrupt service routines and Shadow Sets
a) The instruction `rdpgpr sp, sp` appears at the top of all interrupt service routine prologues.
i) What exactly does this instruction do?

This line of code stands for Read Previous General Purpose Register and it copies the source stack pointer into the current stack pointer in case the register set changes.

ii) In what situation does this instruction actually serve a useful purpose in an ISR prologue?

This instruction serves a useful purpose in an ISR prologue when using Shadow Register Set since the register of the SS can easily be a different register set then we are currently in.

b) We note that the rdpgpr instruction is used even in service routines that use the XC32 IPLnSOFT specifier. This is actually useful because an IPL4 service routine that uses register set 0 could interrupt a priority 3 service routine that used a shadow set. Even if this particular situation couldn't arise in a given system, the rdpgpr

instruction in the prologue can make debugging easier.
To explore this idea, assume we have only **one interrupt** in our system, that its ISR specifies IPL4SOFT, and that we did **not** have the `rdpgpr sp, sp` instruction at the top of the ISR prologue.
i) How could this service routine end up with a shadow set?

It would be an error by the coder the instruction of IPL4SOFT just lets the complier know we do not intend to use a shadow set but that does not stop us from later using one in our interrupt handler.

ii) What happens if the ISR gets a shadow set but that shadow set's `sp` register points to non-existent memory or that it points to existing RAM memory, but that the sp is not properly aligned?

You will throw an exception and will shutdown.

iii) What could happen in this case if the shadow set's `sp` register points to a random (but aligned) place in existing RAM?

You would be messing with memory you don't know what for sure. You could be overwriting something you need or trying to access memory you cant.

iv) Which of ii and iii would be harder to debug? Why?

3 would be harder to debug because things could be working but not as intended.

c) What doesn't have to be done in the service routine prologue or epilogue of an ISR that uses a different register set than the register set(s) that all lower priority routines use?

You don’t need to use the wrpgpr. The wrpgpr is needed any time a priority other than the highest priority uses a shadow set and priorities higher and lower than the on that gets the shadow set use a different register set.

3) Other than running at priority 7, there are two ways we can prevent interrupts from nesting inside an ISR. One involves clearing Status.IE for the body of the ISR and the other involves leaving EXL set for the duration of the ISR. Note that a set EXL and a clear IE only disable interrupts but not other general exceptions. However, if a general exception occurs while EXL is set, EPC is not updated. This means that a general exception that occurred while EXL was set could not be returned from.
a) The IPL6SOFT interrupt prologue handed out in class allows for nested interrupts from the moment the Status register is updated. How can the prologue be changed in order to disable nested interrupts by clearing IE? This can be done by modifying one instruction. Give the changed instruction on the right of the instruction it replaces in the code listing below.
b) What other things don't have to be done in service routine prologues if the service routine doesn't allow interrupt nesting, through clearing IE, that otherwise would have to be done? Show your changes on the right in the code listing below. Place an X in the right hand column next to an instruction to indicate that it is not needed. Assume that we still want to be able to handle other general exceptions in the body of the service routine. Don't worry about adjusting the stack space allocation or stack references. Note that in the body of the ISR, EXL will be cleared, so returning from a nested exception will be possible. Also note that the exception handler may well use a register set different from any of the possibly nested interrupt service routines.
c) Now, assume that we don't want to allow nested interrupts and that we don't expect to have (or to have to be able to return from) any exceptions in the body of the service routine. Indicate the instructions that change or are no longer needed by placing X's next to them in the left hand column in the code listing below.

	`rdpgpr sp, sp`	
X	`mfc0 k0, EPC`	
	`mfc0 k1, Status`	
	`addiu sp, sp, -32`	
x	`sw k0, 28(sp)`	
	`mfc0 k0, SRSCtl`	**x**
	`sw k1, 24(sp)`	
	`sw k0, 20(sp)`	**x**
	`ins k1, zero, 1,15`	**ins k1, zero, 0, 15**
	`ori k1, k1, 0x1800`	
	`mtc0 k1, Status`	

	sw s8, 12(sp)
	sw v1, 8(sp)
	sw v0, 4(sp)
	addiu s8, sp, zero	

4)
a) The IPL6SOFT interrupt prologue in problem 3 above allows for nested interrupts from the moment the Status register is updated. What mechanism kept interrupts from nesting inside the prologue prior to the Status register's update?

Having the EXL bit non zero disables interrupts so until we zero it out interrupts are disabled.

b) Describe what will happen if the mechanism in part a wasn't implemented, allowing interrupts in the prologue prior to the Status register update?

You would get interrupts before critical information is saved leading to not being able to get back and finish the interrupt.

c) General (non-interrupt) exceptions can occur during this period of time, but if they do, the EPC register will not be updated. And, if the EPC register is not updated, there will be no way for the exception handler to return to the point in the prologue where the exception occurred.
What is the only way that we could have a syscall, reserved instruction, coprocessor exception, breakpoint instruction exception, overflow exception, or trap exception, in the prologue prior to the Status register update? One reason applies to all these exceptions. (This would also be true of the DSP ASE exception, the execute inhibit and the machine check exceptions, but don't worry about these.)

if the code instructions changes into a syscall or an instruction that is not properly defined through memory corruption.

d) Two other general exceptions that could occur in the prologue are the address error or the bus error exception. What could cause these exceptions to occur in the

prologue before the Status register update? (This could also cause the other TLB exceptions -- TLB miss or invalid TLB entry.)

unaligned memory reference or a reference to nonexistent memory from overflowing the stack or the stack pointer has become unaligned.

e) What should the general exception handler do in response to any of the exceptions listed in part c and the other exceptions listed in part d?

Shutdown it cannot do anything to salvage the situation.

f) Given your answer to part e, does it matter that EPC is not updated for any general exceptions prior to the Status register update?

No since a general exception shuts down the operation.

g) Why were the writers of the prologue code careful to use `addiu sp, sp, -32` instead of `addi sp, sp, -32`?

Because addi will generate an exception when overflowed but addiu will not.

5) k0, k1.
a) What would have to be done if instead of using k0 and k1 in an IPLnSOFT prologue, we used other registers like v0 and v1?

You would have to make sure that v0 and v1 could not be overwritten by anything but the kernel.

b) So, what advantage does using k0 and k1 in such a prologue give?

They can only be overwritten by the kernel itself because they are kernel registers.

c) Why would what was done in part a not have to be done for an IPLnSRS prologue?

It will be stored in a shadow set register so we could easily retrieve it and know it wont be modified.

6) Priority mismatch.
a) What problem can happen if a service routine uses an IPL4 IPL specifier (could be either IPL4SOFT or IPL4SRS), but the interrupt that uses this service routine actually has a priority of 5 (assigned using the EIC's IPC registers)? Also describe the sequence of events that lead to this problem.

You will get stuck in an infinite loop of trying to interrupt since it will be at level 4 see the level 5 interrupt and start over.

b) What problem**s** can happen if the service routine used in a single vector system use a fixed IPL specifier rather than the RIPL or SINGLE IPL specifier? Assume that different vectors will have different priorites (so that higher priority interrupts that interrupt during a lower priority routine can nest giving them lower latency).

If you don't use RIPL you have no way of knowing the actual priority so nesting will not be possible.

c) How does the prologue generated by the compiler for the RIPL specifier set the priority of the service routine?

It gets the RIPL from the Cause register and then moves it to the lsbs and adjusts IPL to RIPL.