► **Question 1.** Write the initialization code.

► **Question 2**. When are error messages appearing? Clue: look at the program return code, that is located at the end of the handler (there is a comment `## Return to program`). Is the return to program code correct?

► **Question 3.** Write the intruction line append to the program.

► **Question 4.** Why so many asterisks appear when pressing a key?

► **Question 5.** Write the instruction lines added to cancel the keyboard interrupt.

► **Question 6.** Why the user program ends before expected when a key is pressed? **Clue: The handler changes register *$t0* that is used inside the user program *loops.asm*.**

► **Question 7.** With the text editor, open *keyboard.handler*:

- On the handler data segment: Declare a variable named `context` with 4 words capacity.

- In the system starting code: Write into *$k1* the address of variable `context`

- At the beginning of the handler code, where it says “save the user program context”, write the intructions that save the four register former mentioned into `context`, as shown in figure 7.

- At the end of the handler code, where it says “restore the context”, write the code that reads the context content and copies it back to the registers.

► With the simulator: Check if the system works.

► **Question 8.** With the text edtior open *keyboard.handler* and save it as *keyboard_and_clock.handler*.

- In the starting code add the instructions that enable the clock interrupt.

- Change the coprocessor state register to set the interrupt line *int2** unmasked.

► With the simulator: Check the system. If the clock interrupt has been enabled properly the console will show asterisks without stopping.

► **Question 9.** Explain why the system behaves as described previously. Clue: is the clock interrupt cancelled somewhere on the hadler?

► **Question 10**. With the text editor modify *keyboard_and_clock.handler* file.

- Add a new tag (in figure 9 appears as `retexc`) that will mark the point on the handler that restores the context and gives the control back to the interrupted program. From now, jumping to this tag inside the handler will be the same as *ending*.

- Add a new tag `int0` to apply the handling that from steps 4 and 5 are applied to the keyboard interrupt. At the end of the interrupt handling we have to add a jump to *ending*.

- Add a tag `int2` followed by the clock interrupt handling that will be just cancelling the interrupt on the interface avoiding to unable the interrupt. Don't forget jumping to *ending*.

- Add the instructions that will read and analyse the exception cause word from the exception coprocessor (the cause word explanation is on appendix 2) in the section marked with "`## Identify and handle exceptions`". If the cause is an interrupt the analysis will continue, otherwise a jump will be done to *ending*.

- Once it has been identified that the cause of a given exception is an interrupt, what remains is to add the instructions that will analyse bits *IP0* and *IP2* from the state word in the exception coprocessor followed by a jump to tags `int0` e `int2`., respectively. If none of these bits is active then jump to *ending*.

►Take carea ll the time over the registers used by the handler. In step 5 we have written the code that saves and restores registers *$at*, *$t0*, *$v0 and $a0*. You can restrict yourself to use only those four registers on the handler, or you can use other registers like (like *$t1*) if so you have to get aware that the context variable size should be increased to get space for those extra registers.

► With the simulator, check the system. If everything is fine, it will appear on the console one and only one "`*`" every time a key is pressed; the user program will smoothly end when the time comes.