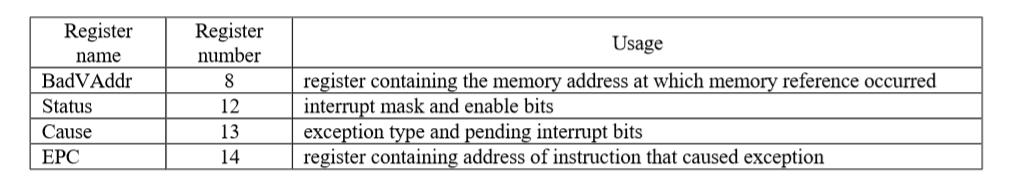
This is an interrupt for reading a value from the keyboard. Assuming that the keyboard's I.E. bit is set to '1', this will run when someone hits a key on the keyboard. The first line, ".text 0x80000180" places the code explicitly at the memory location where the interrupt service routine is called. It checks the cause field by “AND” operation with 0x003c; branches if this is not the exception; otherwise it has a nested function which gets called (read and store byte). **Special Registers used in this particular case:**$k0 stores cause; such that the kernel can use this as a way to determine the reason surrounding the interrupt.

$k1 stores EPC, such that the kernel can use this as a way to determine to go back to original instruction flow.

In the normal microprocessor, the registers $k0 and $k1 are both used as temporary variables in interrupt servicing routines. Coprocessor 0 is also used with interrupts. In Coprocessor 0, registers $8, $12, $13, and $14 are all important in servicing interrupts. These registers can be read and modified using the instructions mfc0 (move from coprocessor 0) and mtc0 (move to coprocessor 0).

**Information about Special Registers:**

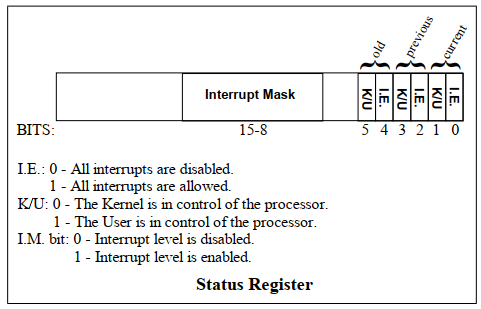
**“BadVaddr” Register**

Contains the memory address of the bad memory read which had occurred.

**Status Register**

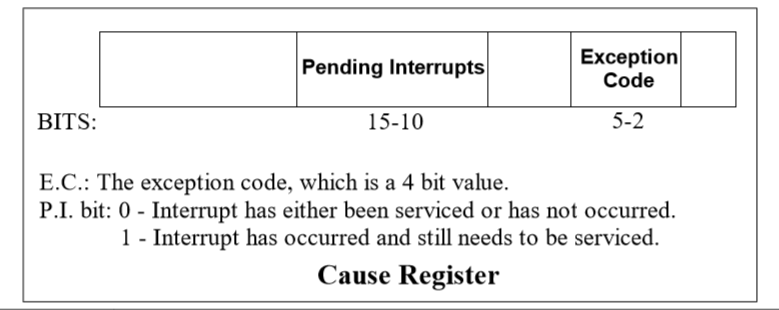
The status register gets modified as the program is running; it is used to understand the current state of the program and if it is accepting interrupts or currently managing one.

The status register is composed of two major components, the *interrupt mask* and *kernel/user - interrupt enable* bits.



**Cause Register**

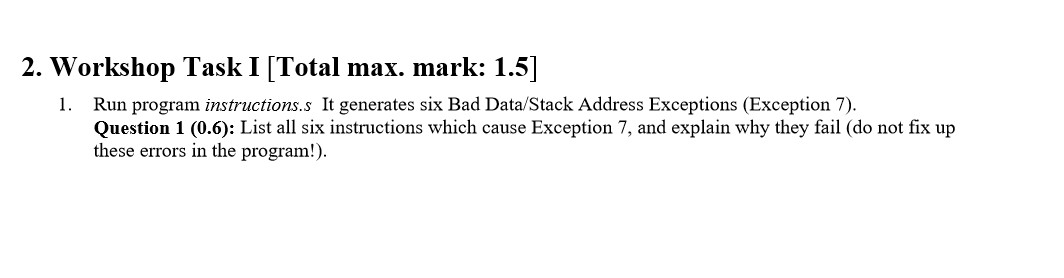
The cause register is also composed of two major components. A pending interrupts field and an exception code field



The pending interrupts field consists of bits, each corresponding to a separate interrupt level. If a bit is '1', then an interrupt has occurred and still needs to be serviced. The exception code field consists of a 4 bit value that tells us what exception occurred to cause the interrupt.

**EPC Register**

The EPC register contains the value of the program counter at the time of the interrupt. This is where the program will return after handling the interrupt.

$t1’s address is being set to the value of 0; this is obviously an incorrect address. Any other value which is stored or loaded from this point (using $t1 as the base address) will cause a “bad address read”; which is exception 7.  **The corresponding instruction involved in the interrupt:**

lw $t4, 84($t1) # Constant 4 stored in $t4

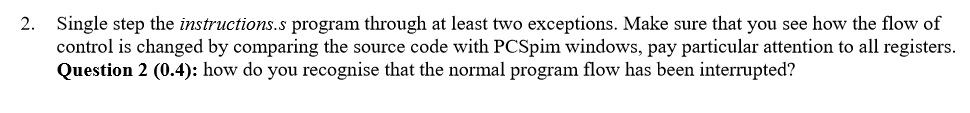
lw $t0, 88($t1) # Constant 9 stored in $t0 (i)

lw $t5, 92($t1) # Constant -1 stored in $t5

lw $t3, 0($t1) # $t3 has A[k]

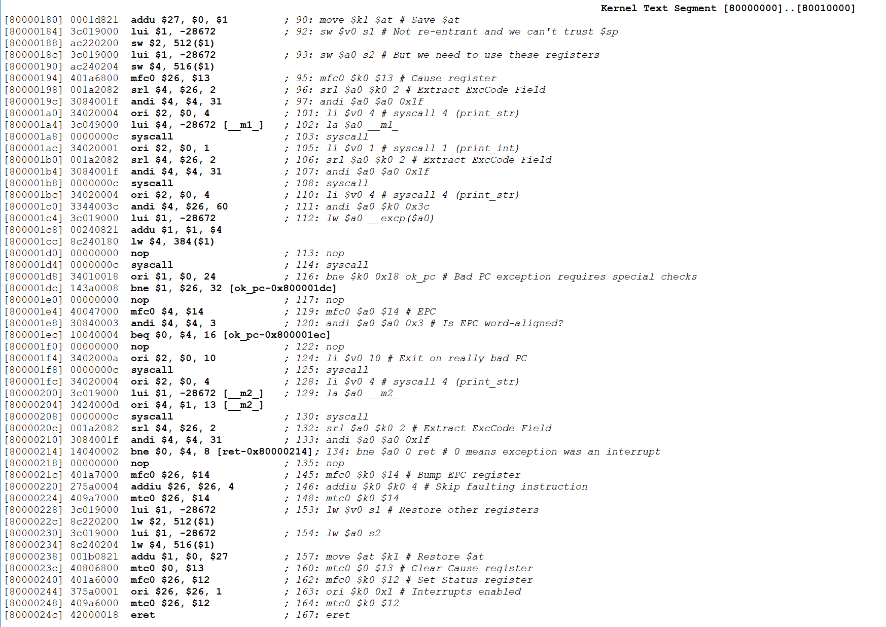
sw $t2, 40($t1) # B[k] = sum

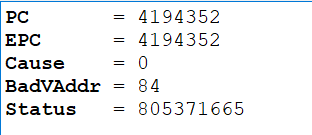
sw $t2, 0($t1) # sum is stored

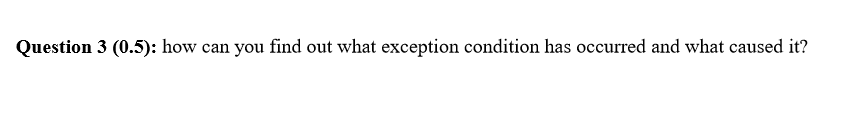


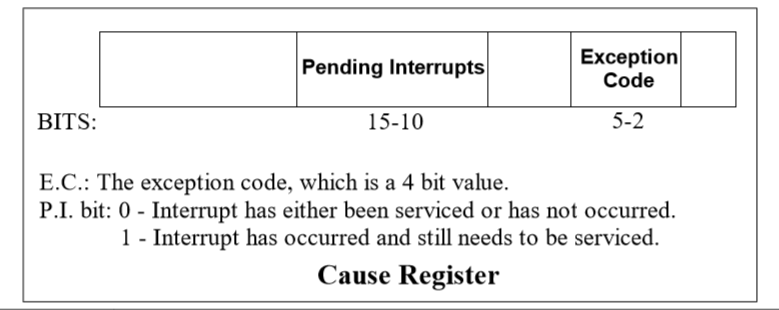
We can determine that an interrupt occurred by taking observation of the PC, EPC, cause, BADVAddr and Status counters. Once the PC is set to 0x80000180, the instruction jumps to the kernel address, and executes the instructions required for interrupts and we can observe this in the second diagram. This is two ways we can determine if the program flow:

1. Noticing Coprocessor 0’s special registers
2. See if instructions are at kernel

****

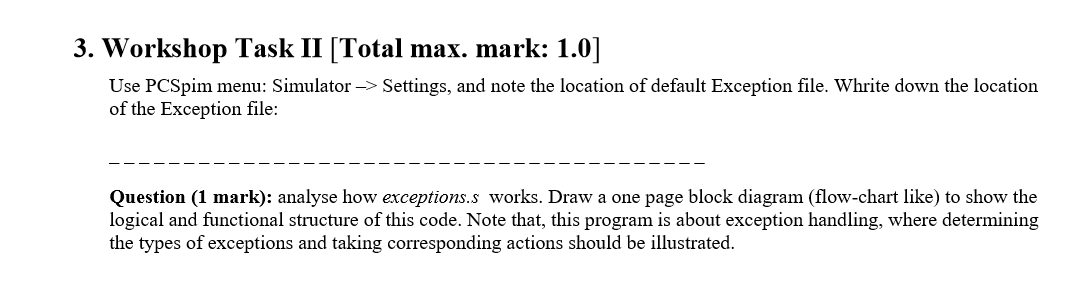
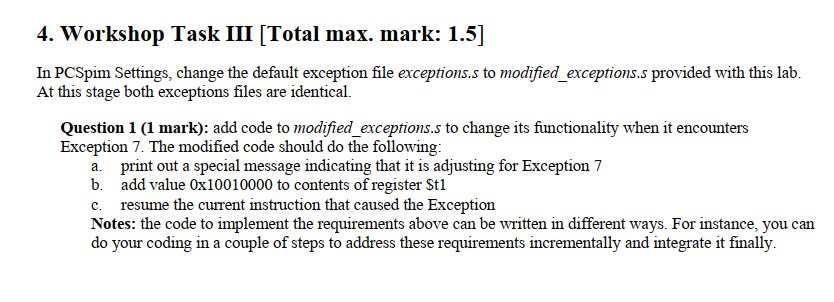
****

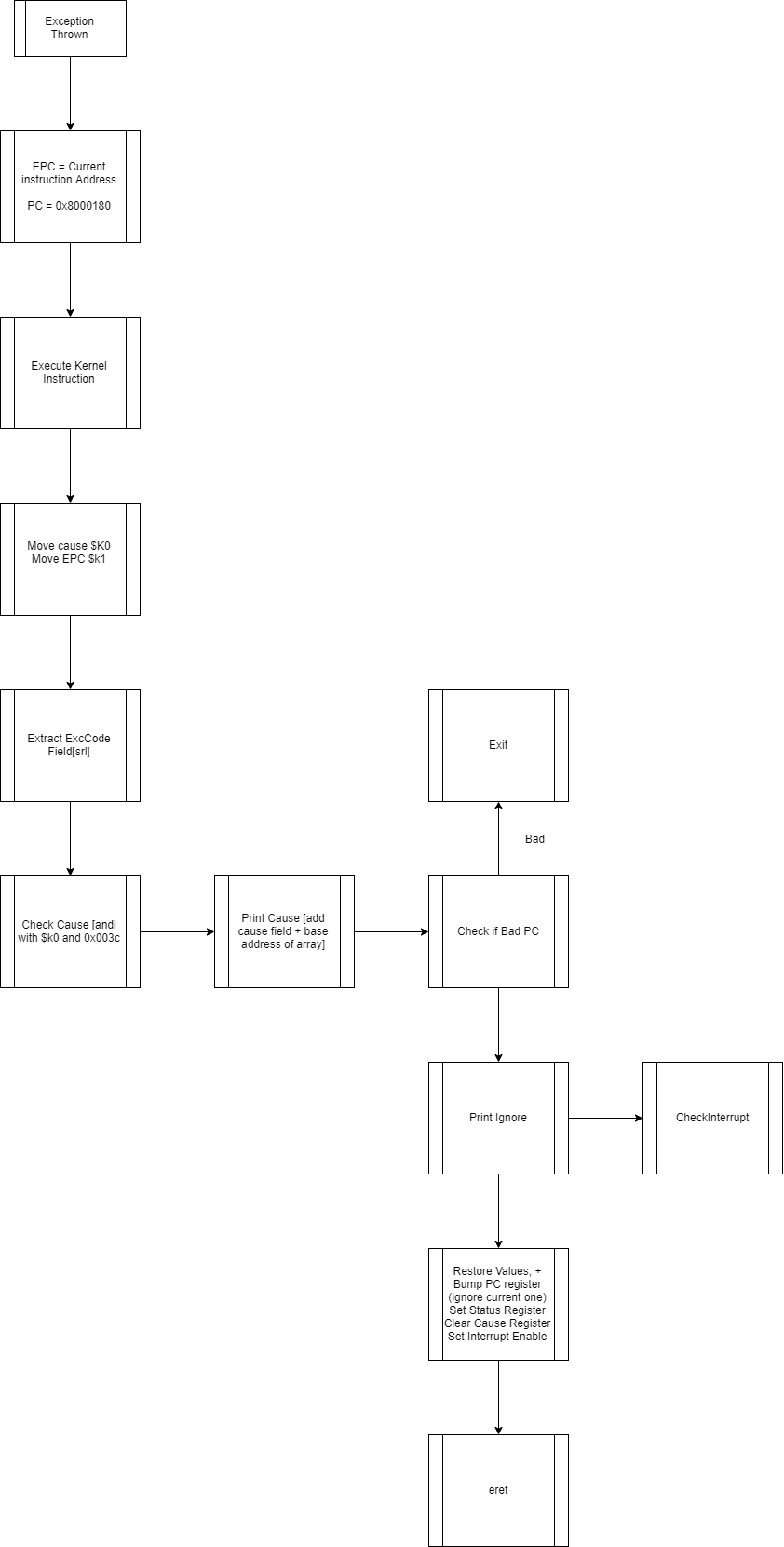
****We can determine the exception by looking at the cause register (refer special registers question 1).

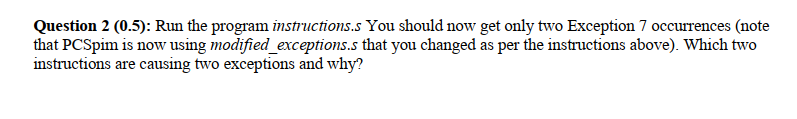


The pending interrupts field consists of bits, each corresponding to a separate interrupt level. If a bit is '1', then an interrupt has occurred and still needs to be serviced. The exception code field consists of a 4 bit value that tells us what exception occurred to cause the interrupt.

The program jumps through sections of program and uses a ***mask*** to determine exactly what the cause was, and from this point, it can determine the way it should handle this problem and the corresponding message thrown back to the receiver.

 ****It has been completed.





$t1 was pointing to an invalid address (0); as we went through the interrupt, the program detected this at the first bad memory address read. When the program went to the kernel; the modified exception program changed the address (by adding 0x10010000) which corrected the issue; and returned this value to $t1. After the loop had finished the program reset the value of $t1; and we see that another bad memory read occurs from this point (and therefore we notice two exception occurrences).