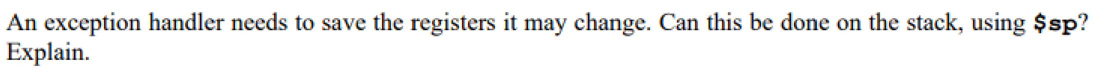
Instalación y Reemplazo de Componentes Internos 6to Info 2021
Trabajo Práctico Excepciones e I/O en assembly del MIPS R2000
Grupo D: Teo Baiocchi, Andrés Grillo, Santiago Libonati, Luca Seri, Mateo Regodesebes

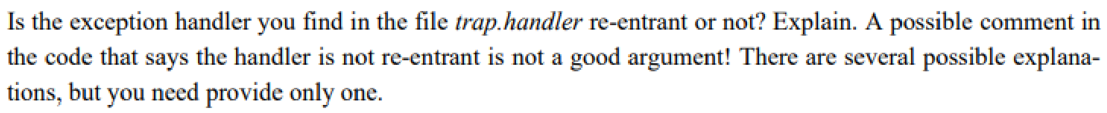
Step 1

Q 1:


An exception handler needs to save the registers it may change. Can this be done on the stack, using **$sp**? Explain.

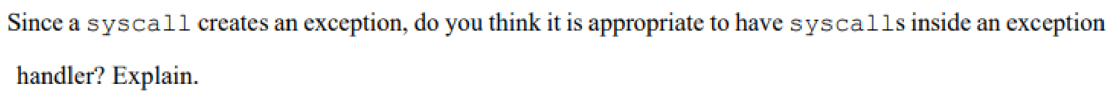
Un exception handler no puede guardar los datos de registros en la pila ($sp). Esto se debe a que la pila podría estar completa y por lo tanto sería imposible utilizarla sin borrar o sobreescribir algún valor relevante para el programa en sí.

Q 2:


Is the exception handler you find in the file *trap.handler* re-entrant or not? Explain. A possible comment in the code that says the handler is not re-entrant is not a good argument! There are several possible explanations, but you need provide only one.

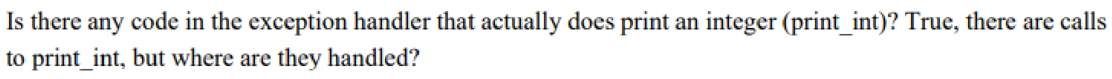
No lo es.
La re-entrabilidad se aplica en handlers complejos, que corren por largos periodos de tiempo, y por tanto donde el riesgo de que algo salga mal es aumentado. A más largo el potencial tiempo de ejecución, más conveniente es que el handler sea interrumpible (sea re-entrable).
Ahora bien, con esto en vista, el handler que encontramos es particularmente sencillo y corto. Este es el motivo por el que suponemos que no es re-entrante (Y está confirmado que no lo es en una línea de comentario)

Q 3:


Since a `syscall` creates an exception, do you think it is appropriate to have `syscalls` inside an exception handler? Explain.

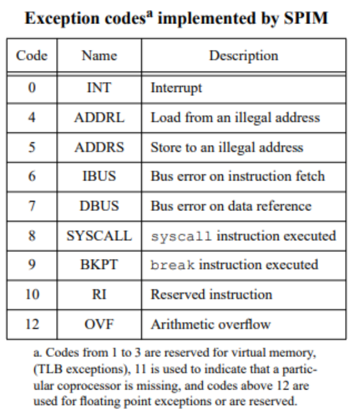
Sí.
Si bien no es ideal causar excepciones dentro del exception handler, el caso de las syscalls parece ser la *excepción.* Es necesario utilizar llamadas al sistema para realizar impresiones en pantalla o para salir. Comprobamos que esto es así porque en el exceptions.s por defecto del programa se utilizan syscalls 1, 4 y 10 múltiples veces.

Q 4:


Is there any code in the exception handler that actually does print an integer (print_int)? True, there are calls to print_int, but where are they handled?

Esas llamadas son manejadas en la sección del exception handler en la que se imprime en pantalla información sobre la excepción. Lo que imprime es el ExcCode Field, un número que indica la causa de la excepción:

__e12_: “[Arithmetic overflow]”


Exception codes[a] implemented by SPIM

Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	`syscall` instruction executed
9	BKPT	`break` instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic overflow

a. Codes from 1 to 3 are reserved for virtual memory, (TLB exceptions), 11 is used to indicate that a particular coprocessor is missing, and codes above 12 are used for floating point exceptions or are reserved.

Q 5:

Why is there a need to check whether the EPC is word aligned?

La importancia del ECP viene de que es utilizado para guardar la dirección de memoria de la instrucción que se estaba ejecutando en el momento que se produce una excepción, para poder volver allí una vez el exception handler termine su trabajo. Para poder continuar sin repetir la misma instrucción, esta dirección de memoria se incrementa en 4 bytes.
En pocas palabras, el ECP contiene una dirección de memoria vital para que el flujo del programa pueda proseguir junto al manejo de excepciones y para que esto suceda es necesario que se encuentre alineado correctamente

Step 2

Se pide en el enunciado elaborar un código que guarde el mayor valor posible dentro de un entero y luego intentar sumarlo consigo mismo. Él código que vamos a utilizar es el siguiente

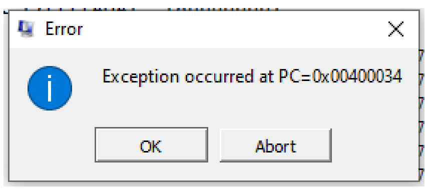
```
.data 0x10000000
numero: .word 2147483647
.text
main:
    lw $t0, numero
    lw $t1, numero
    add $t2, $t0, $t1
    li $v0, 10
    syscall
.end
```

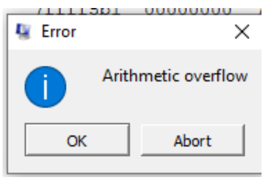
Q 6:

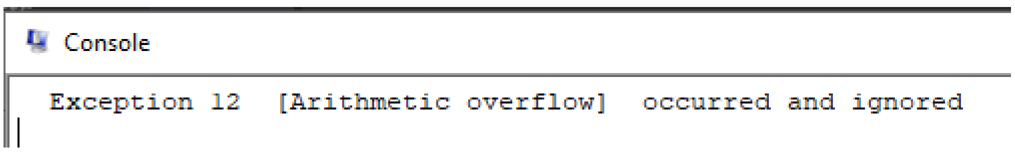
Will your program, when run, generate an exception? If you answer yes, then indicate whether it is a trap or an interrupt.

Si intentásemos correr nuestro programa, siempre obtendríamos una excepción. Y siempre ocurriría en el mismo punto: Cuando intentamos realizar la suma de nuestro entero de valor máximo. Como se trata de un Arithmetic Overflow, es una excepción que puede ser ignorada sin interrumpir el resto del código (aunque en este caso no tengamos) y por lo tanto es una **trap**

Run *lab7.1.asm* and write down the error message you get on screen. For each line in the error message clearly indicate the source in the 'Source' column: use a S to indicate the message is generated inside the simulator, and a T to indicate it is generated inside the trap handler.


Error
Exception occurred at PC=0x00400034


Error
Arithmetic overflow

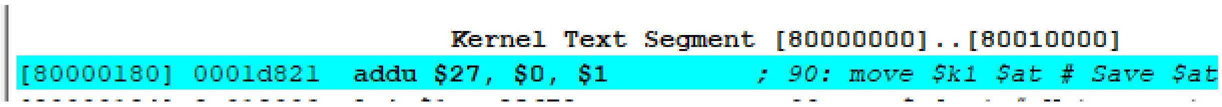


```
Exception 12  [Arithmetic overflow]  occurred and ignored
```

Step 3

Step through *lab7.1.asm* until you enter the exception handler. Some instructions load data from the kernel data segment. Based on what the simulator prints when you step, find out at what address the data segment starts. Show your work.

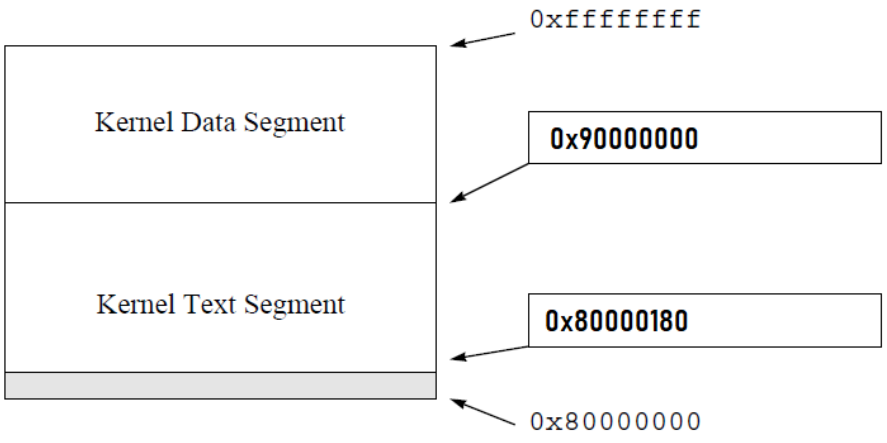
El segmento de kernel comienza en 0x80000180



```
                         Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1          ; 90: move $k1 $at # Save $at
```

Step 4

In the Laboratory #2 inlab exercise, the model of MIPS systems memory has been presented. However, there are no details about the kernel space. A more detailed image is presented below. Fill out the missing information on this figure.



Step 5

The trap handler prints the appropriate error message for each exception by using the exception code from the Cause register as an index in a table containing message addresses. At this step you are required to identify the table and show a few entries in it. In the 'Label' column, the first row will be the label where the table begins. In the 'Comment' column indicate what is the content stored at each of the addresses in the column 'Address'.

Label	Address	Contents	Comment
_ e4	0x9000004b	"Address error in inst/data fetch"	Cargar desde una dirección ilegal
_ e5	0x90000071	"Address error in store"	Guardar en una dirección ilegal
_ e6	0x9000008d	"Bad instruction address"	Intentar acceder a una dirección de memoria invalida, fuera del data o del stack segment
_ e7	0x900000aa	"Bus error on data reference"	Error de hardware: Se intenta acceder a memoria que no se puede localizar físicamente
_ e8	0x900000c0	"Error in syscall"	Error durante una llamada al sistema
_e9	0x900000d6	"Breakpoint"	Ejecutar una instrucción break (que genera una interrupción en el programa)
_e10	0x900000e6	"Reserved instruction"	Intentar ejecutar instrucciones que no son posibles para MIPS o no están permitidas. Por ejemplo: Ejecutar operaciones de 64 bits desde un modo 32
_e12	0x90000101	"Arithmetic overflow"	Intentar crear/guardar un valor numérico que supera el rango posible

Step 6

Printing inside the exception handler is done by using `syscall`. However, the code that actually prints a character on the console is not there. SPIM uses the services of the machine it is running on to do the actual character output or input.

We now want to see how it is like to do it, using the bare devices the machine offers. SPIM simulates a memory-mapped terminal connected to the processor. Memory mapped means that accessing some memory locations accesses the I/O devices in reality. Writing to a specific memory location will actually write to an output device, while reading from some specific memory location actually reads from an I/O device. The terminal device SPIM simulates consists of two independent units: the *receiver* and the *transmitter*. The receiver reads characters from the keyboard as they are typed. The transmitter unit writes characters to the terminal's display. Both the receiver and the transmitter can work in interrupt mode. At this step, however, you will do something simpler: you will modify the exception handler as to print using the transmitter unit. Modify the exception handler as follows:

- write a procedure called *PrintString* which receives in `$a0` the address of the string to print, and returns no value. The string to print is null-terminated
- the procedure uses busy-waiting for printing
- call 'PrintString' in the exception handler instead of print_str

A few hints are probably in place:

- you will probably need to save more registers when you enter the exception handler than it does now; save them in kernel data segment
- busy waiting means that you have a loop where you test the 'Ready' bit in the Transmitter Control Register until it becomes 1, thus signaling that the output device is ready to accept a new character
- start the spim simulator with the `-mapped_io` flag in the command line
- the null symbol is the byte 0x00

```
PrintString:
loop1:
    lb $v0, ($a0)
    bne $v0, $0, imprimir
    j endLoop1

imprimir:
    lb $k1, 0xffff0008 # TCR
    andi $k1, $k1, 0x01
    beqz $k1, imprimir

    sb $v0, 0xffff000c # TDR
    addi $a0, $a0, 1

    j loop1
endLoop1:
    jr $ra
```

Write a program called *lab7.2.asm* which does:

- loops 100 times: inside the loop it calls the procedure *PrintStr* to print the message “Hello world”
- the procedure ‘PrintStr’ receives a pointer to the string to print in **$a0**, and returns nothing. The pro-

The ‘PrintStr’ procedure will not manipulate the TDR: the interrupt routine will do it. The only interaction PrintStr has with the device registers is to make sure that interrupts are enabled.

The buffer ‘PrintStr’ shares with the interrupt routine is a circular buffer with a capacity of 256 bytes. The buffer is in the user space. The names for the two pointers in the circular buffer will be `t_in` (to indicate where to put a new char in the buffer) and `t_out` (to indicate where is the next character to send).

Since the program generates characters to print much faster than the transmitter can print, the buffer will be eventually filled: in a real system the operating system would block the currently executing process (the one who has filled up the buffer) and would let another process run. Since the SPIM simulator is a single-user mono-programming environment (as opposed to a multi-programming one), the procedure ‘PrintStr’ should handle the case when the buffer is full. One approach is to repeatedly check the buffer until it is no longer full.

Turn the interrupts on when ‘PrintStr’ deposits something in the buffer. This works together with the interrupt routine which will turn the interrupts off when the buffer is empty.

```
.data
PrinThing: .asciiz "Hola\n"
t_in: .word 0
t_out: .word 0
buffer: .space 256
FinBuffer: .word 0
.text
main:
    la $s1, t_out # dirección de t_out
    la $s2, t_in  # dirección de t_in
    la $t2, FinBuffer # puntero al final del buffer
    la $t3, buffer # puntero al principio del buffer
    sw $t3, ($s2) # t_in apunta al principio del buffer
    sw $t3, ($s1) # t_out apunta al principio del buffer
    lw $t4, ($s2) # contenido de t_in
    lw $t5, ($s1) # contenido de t_out
    li $t1, 5

    addi $sp, $sp, -4
    sw $ra, ($sp)
for:
    beq $t1, $0, final
    la $a0, PrinThing

    jal PrintStr

    addi $t1, $t1, -1
    j for
final:

    li $sp, 4
    lw $ra, ($sp)
    jr $ra



PrintStr:

    lb $t0, ($a0)
    addi $a0, $a0, 1
    sb $t0, ($t4) # meter el caracter en buffer
    addi $t4, $t4, 1 #  mover el t_in en un byte
    beq $t4, $t2, resetIn # si t_in es igual al final del buffer resetear el t_in
al principio del buffer
continuar:
    sw $t4, t_in

    # imprimir caracter
    mfc0 $t6, $12
    ori $t6, $t6, 0x201
    mtc0 $t6, $12

    li $t6, 0xffff0008
    lw $t7, 0($t6)
    ori $t7, $t7, 0x2
    sw $t7, 0($t6)
    # imprimir caracter

    addi $t5, $t5, 1 # mover t_out en un byte
    beq $t5, $t2, resetOut
proceder:
    sw $t5, t_out
    addi $a0, $a0, 1
    beqz $a0, FinPrint
    j PrintStr
resetIn:
    move $t4, $t3
    j continuar
resetOut:
    move $t5, $t3
    j proceder
FinPrint:
    jr $ra
```

Modify the exception handler as to accept interrupts at level 1. If the handler is invoked because of such an interrupt, then call a procedure named *PutChar* which will take a character from the circular buffer and store it into TDR. Of course, if the transmitter is not ready, then just return: an interrupt should not have been produced in the first place in this case, but it does not hurt to test.

If the buffer is empty then turn the interrupts off. Otherwise interrupts will be generated all the time to indicate the transmitter is ready.

Make also sure you properly set up the interrupt mask at level 1 (IM1) and the interrupt enable (IEc), every time you enable or disable interrupts.

```
PrintString:
    lb $v0, ($t5) # cargamos el contenido de t_out en $v0
imprimir:
    # corroboramos que se puede imprimir el caracter
    lb $k1, 0xffff0008
    andi $k1, $k1, 0x01
    beqz $k1, imprimir
    # colocamos el caracter en el TDR
    li $k0, 0xffff000c
    sb $v0, ($k0)
    # reseteamos el bit ready del TCR
    lw $k1, 0xffff0008($k0)
    xori $k1, $k1, 0x2
    sw $k1, 0($k0)

    j return
```