# Exceptions in MIPS

## Objectives

After completing this lab you will:

- know the exception mechanism in MIPS
- be able to write a simple exception handler for a MIPS machine

## Introduction

Branches and jumps provide ways to change the control flow in a program. *Exceptions* can also change the control flow in a program.

The MIPS convention calls an *exception* any unexpected change in control flow regardless of its source (i.e. without distinguishing between a within the processor source and an external source).

An exception is said to be synchronous if it occurs at the same place every time a program is executed with the same data and the same memory allocation. Arithmetic overflows, undefined instructions, page faults are some examples of synchronous exceptions. Asynchronous exceptions, on the other hand, happen with no temporal relation to the program being executed. I/O requests, memory errors, power supply failure are examples of asynchronous events.

An *interrupt* is an asynchronous exception. Synchronous exceptions, resulting directly from the execution of the program, are called *traps*.

When an exception happens, the control is transferred to a different program named **exception handler**, written explicitly for the purpose of dealing with exceptions. After the exception, the control is returned to the program that was executing when the exception occurred: that program then continues as if nothing happened. An exception appears as if a procedure (with no parameters and no return value) has been inserted in the program.

Since the code for the exception handler might be executed at any time, there can be no parameters passed to it: passing parameters would require prior preparation. For the same reason there may not be any return value. It is important to keep in mind that the exception handler must preserve the state of the program that was interrupted such that its execution can continue at a later time.

As with any procedure, the exception handler must save any registers it may modify, and then restore them before returning control to the interrupted program. Saving registers in memory poses a problem in MIPS: addressing the memory requires a register (the base register) in which the address is formed. This means that

© 1996, Virgil Bistriceanu

a register must be modified before any register can be saved! The MIPS register usage convention (see Laboratory 4) reserves registers **`$26`** and **`$27`** (**`$k0`** and **`$k1`**) for the use of the interrupt handler. This means that the interrupt handler can use these registers without having to save them first. A user program that uses these registers may find them unexpectedly changed.

## The MIPS exception mechanism

The exception mechanism is implemented by the coprocessor 0 which is always present (unlike coprocessor 1, the floating point unit, which may or may not be present). The virtual memory system is also implemented in coprocessor 0. Note however that SPIM does not simulate this part of the coprocessor.

The CPU operates in one of the two possible modes, **user** and **kernel.** User programs run in user mode. The CPU enters the kernel mode when an exception happens. Coprocessor 0 can only be used in kernel mode.

The whole upper half of the memory space is reserved for the kernel mode: it can not be accessed in user mode. When running in kernel mode the registers of coprocessor 0 can be accessed using the following instructions:

**Instructions which access the registers of coprocessor 0**

| Instruction | Comment |
|---|---|
| `mfc0 Rdest, C0src` | Move the content of coprocessor's register `C0src` to `Rdest` |
| `mtc0 Rsrc, C0dest` | Integer register `Rsrc` is moved to coprocessor's register `C0dest` |
| `lwc0 C0dest, address` | Load word from `address` in register `C0dest` |
| `swc0 C0src, address` | Store the content of register `C0src` at `address` in memory |

The relevant registers for the exception handling, in coprocessor 0 are

**Exception handling registers in coprocessor 0**

| Register Number | Register Name | Usage |
|---|---|---|
| 8 | BadVAddr | Memory address where exception occurred |
| 12 | Status | Interrupt mask, enable bits, and status when exception occurred |
| 13 | Cause | Type of exception and pending interrupt bits |
| 14 | EPC | Address of instruction that caused exception |

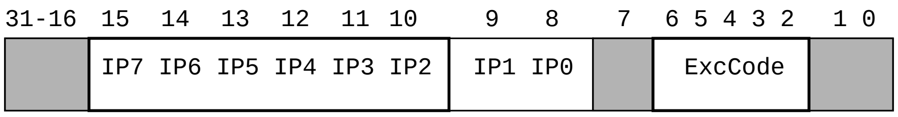
## The BadVAddr register

This register (its name stands for **Bad V**irtual **Addr**ess) will contain the memory address where the exception has occurred. An unaligned memory access, for instance, will generate an exception and the address where the access was attempted will be stored in BadVAddr.

© 1996, Virgil Bistriceanu

## The Cause register

The Cause register provides information about what interrupts are pending (IP2 to IP7) and the cause of the exception. The exception code is stored as an unsigned integer using bits 6-2 in the Cause register. The layout of the Cause register is presented below.



Bit IPi becomes 1 if an interrupt has occurred at level i and is pending (has not been serviced yet). The bits IP1 and IP0 are used for simulated interrupts that can be generated by software. Note that IP1 and IP0 are not visible in SPIM (please refer to the SPIM documentation).

The exception code indicates what caused the exception.

**Exception codes[a] implemented by SPIM**

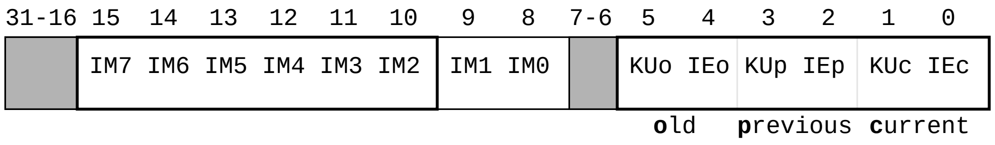
| Code | Name | Description |
|---|---|---|
| 0 | INT | Interrupt |
| 4 | ADDRL | Load from an illegal address |
| 5 | ADDRS | Store to an illegal address |
| 6 | IBUS | Bus error on instruction fetch |
| 7 | DBUS | Bus error on data reference |
| 8 | SYSCALL | `syscall` instruction executed |
| 9 | 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.

Code 0 indicates that an interrupt has occurred. By looking at the individual IPi bits the processor can learn what specific interrupt happened.

## The Status register

The Status register contains an interrupt mask on bits 15-10 and status information on bits 5-0. The layout of the Status register is presented below.



If bit IMi is 1 then interrupts at level i are enabled. Otherwise they are disabled. In SPIM IM1 and IM0 are not visible to the programmer.

KUc (bit 1 in the register) indicates whether the program is running in user (KUc = 1) or kernel (KUc = 0) mode. KUp (bit 3 in the register) indicates whether the processor was in kernel (KUp = 0) or user mode when last exception occurred. This information is important since at the return from the exception handler the processor must be in the same state it was when the exception happened. Bits 5-0 in the Status register implement a simple, three level stack with information about previous exceptions. When an exception occurs, the `previous` state (bits 3 and 2) is saved as the `old` state and the `current` state is saved as the `previous` state. The `old` state is lost. The `current` state bits are both set to 0 (kernel mode with interrupts disabled). At the return from the exception handler (by executing a `rfe` instruction), the `previous` state becomes the `current` state and the `old` state becomes the `previous`. The `old` state is not changed.

The Interrupt Enable bits (IEj) indicate whether interrupts are enabled (IEj = 1) or not (IEj = 0) in the respective state. If for instance IEc is zero, then the processor is currently running with the interrupts disabled.

## The EPC register

When a procedure is called using `jal`, two things happen:

- control is transferred at the address provided by the instruction
- the return address is saved in register **`$ra`**

In the case of an exception there is no explicit call. In MIPS the control is transferred at a fixed location, `0x80000080` when an exception occurs. The exception handler must be located at that address.

The return address can not be saved in **`$ra`** since it may clobber a return address that has been placed in that register before the exception. The Exception Program Counter (EPC) is used to store the address of the instruction that was executing when the exception was generated.

## Returning from exception

The return sequence is standard:

**Ex 1:**

```
#
# the return sequence from the exception handler for the case of an
# external exception (interrupt)
#
      mfc0 $k0, $14      # get EPC in $k0
      rfe                # return from exception
      jr $k0             # replace PC with the return address ■
```

© 1996, Virgil Bistriceanu

The architecture makes a clear distinction between interrupts (external exceptions) and traps (including explicit software invocations such as `syscall`). In the case of an interrupt the Program Counter has already been advanced to point to the next instruction at the moment the control was transferred to the exception handler. In other words, the EPC contain the address of the instruction to execute after the return from the exception handler. In the case of a trap or a `syscall`, the EPC contains the address of the instruction that has generated the trap. To avoid executing the instruction again at the return from the exception handler, the return address must be incremented by four, thus making sure the instruction that follows in the flow will be executed.

**Ex 2:**

```
#
# the return sequence from the exception handler for the case of a
# trap (including a syscall).
#
      mfc0 $k0, $14     # get EPC in $k0
      addiu $k0, 4      # make sure it points to next instruction
      rfe               # return from exception
      jr $k0            # replace PC with the return address ■
```

© 1996, Virgil Bistriceanu

# **Laboratory 7:** Prelab

Date ______________________ Section ______________________

Name ______________________________________________

## Making Sense Of An Exception Handler

During the prelab you will study an existing exception handler, the one that comes with the SPIM simulator. You will then be able to write one on your own (this will be the inlab task).

## Introduction

When an exception occurs, the processor enters the kernel mode and control is transferred at address `0x80000080`. Part of the invocation process is disabling interrupts: the hardware takes care of it. This is needed since the exception handler has to save the state of the processor before doing any processing for that interrupt. At the very least, the processor must be able to execute uninterrupted long enough to

- find out what caused the exception and save the relevant information (EPC, the Cause register, the Status register)
- save the state of the processor (actually save all registers the trap handler may use)

Disabling interrupts does not mean they are eliminated. It only means their effect is deferred. Since interrupts inform the processor about significant events, one may not indefinitely disable them. But it is safe to ignore interrupts for short periods of time. At the time interrupts are enabled again, the hardware will treat any pending interrupt as if it just occurred and immediately invoke the exception handler.

As noted before, the exception handler is invoked with the interrupts being disabled. However, there is nothing to prevent traps from occurring within the handler. It is the programmer's responsibility to make sure the exception handler does not execute any instruction that cause an exception.

If the exception handler returns control to the interrupted program after a short period of time, then it may run without enabling interrupts. In this case there is no need to save the exception information (EPC, Cause, Status). The registers the handler uses must be saved. Of course the two dedicated registers, **`$k0`** and **`$k1`**, need not be saved and can be used right away.

If the exception handler contains code that may run for long times, then it would be unsafe to execute that code with the interrupts disabled since some events may be lost. Imagine for instance the exception handler executing code for a communication interface and ignoring an interrupt that indicates a power failure. In general, a correctly written exception handler should be itself interruptible. Such a handler is said to be **reentrant**.

If interrupts are enabled within the exception handler, then the relevant information (EPC, Cause, Status) must be saved before enabling interrupts exception. All registers the handler may use must be saved anyway.

© 1996, Virgil Bistriceanu

Two things must be properly set to enable interrupts:

- IEc (bit 0 in the Status register) must be 1: this indicates that interrupts are allowed in the current state
- the interrupt mask (IM) which indicates what specific interrupt(s) to enable

The IEc bit enables or disables all interrupts while the individual bits (IMi) in the interrupt mask enable or disable individual interrupts. Either can disable an interrupt but both (IEc and IMi must be set for interrupt i to be enabled).

The Interrupt Pending bits (IPi) in the Cause register indicate whether or not an interrupt is pending. Note here that an IPi bit in this register will be set, thus indicating a pending interrupt, even if the interrupts are disabled. This way the exception handler can determine whether there are pending interrupts which would cause an exception had they not been disabled.

## Step 1

Begin by making a copy of the file called *trap.handler*: call the copy *trap.handler.original*. You will work on the file *trap.handler* which the SPIM simulator loads by default. Look at the file and, based on what you see, try to answer a few questions. The '.set noat' assembler directive instructs the assembler not to complain about using the **`$at`** register (which is reserved for the assembler's usage).

## Q 1:

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

## 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.

## Q 3:

Since a `syscall` creates an exception, do you think it is appropriate to have `syscall`s inside an exception

© 1996, Virgil Bistriceanu

handler? Explain.

## 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?

## Q 5:

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

### Step 2

As you can see the exception handler does not worry about interrupts. We ignore them for now. What the handler does is that it only prints out a message indicating what happened and then returns. The message SPIM prints has two parts, the first part comes from the simulator itself and the second is printed within the exception handler.

Write a program called *lab7.1.asm* which:

- declares a variable called MAX_INT with the initial value the maximum value of a signed integer
- loads the value of MAX_INT in register **`$t0`** and adds it to itself (use signed addition)

## Q 6:

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

© 1996, Virgil Bistriceanu

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 message generated by** *lab7.1.asm*

| Error message | Source |
|---|---|
| | |
| | |
| | |
| | |

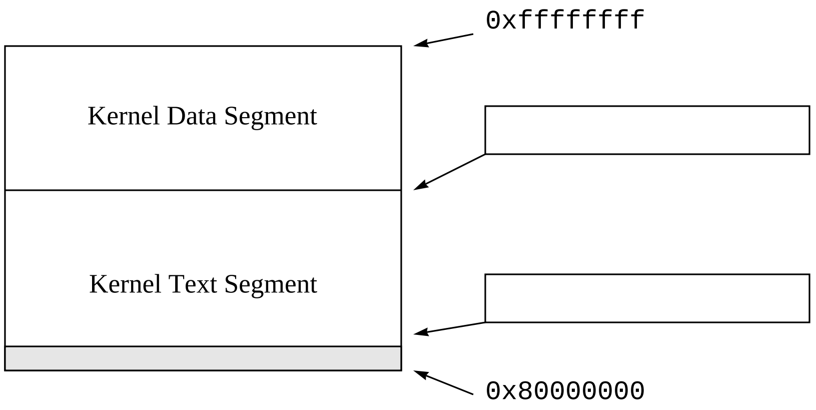
If you don't like having SPIM print a message every time an exception is generated, then you can run the simulator in quiet mode: use the `-quiet` flag in the command line to do so. In this case the only error messages will be the ones printed from within the exception handler.

## 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.

## 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 |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## 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

Use the program *lab7.1.asm* for testing. If you attempt to run the program and it hangs or produces exceptions other than what *lab7.1.asm* does when running with the unmodified exception handler, then you may want to go step by step and watch the content of relevant registers and memory.

© 1996, Virgil Bistriceanu

# **Laboratory 7:** Inlab

Date ______________________ Section ______________________

Name ______________________________________________

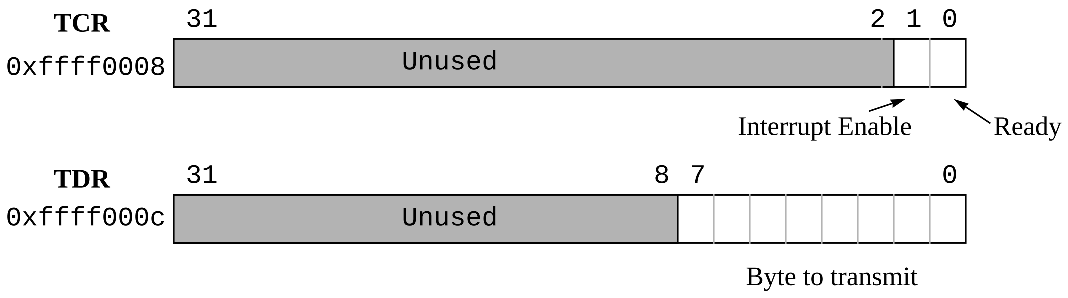
## Interrupt Driven Output

Your job is to build the set of procedures needed to do interrupt driven output to the terminal.

## Introduction

As described in the Prelab exercise, 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.

The processor accesses the transmitter using two memory-mapped device registers:



The Transmitter Control Register (TCR) is located at address 0xffff0008. Bit 0 (LSB) called 'Ready' is read-only. If it is one, then the transmitter is ready to accept a new character. Bit 1 called 'Interrupt Enable' can be read or written. If the user sets it to one, then an interrupt will be requested on level one whenever the ready bit is one (whenever the transmitter is ready to accept a new character).

The Transmitter Data Register (TDR) is located at address 0xffff000c. The least significant byte can be written and will be interpreted as an ASCII character to output to the display.

## Step 1

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-

cedure will copy the output characters in a buffer shared with the interrupt routine

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.

## Step 2

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.

© 1996, Virgil Bistriceanu