7

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 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

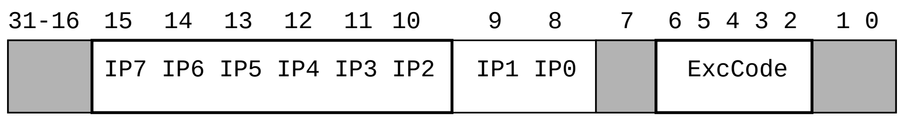
Register Number	Register Name	Usage
8	BadVAddr	Memory address where exception occurred
12	Status	Interrupt mask, enable bits, and status when exception occurred
13	Cause	Type of exception and pending interrupt bits
14	EPC	Address of instruction that caused exception

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.

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

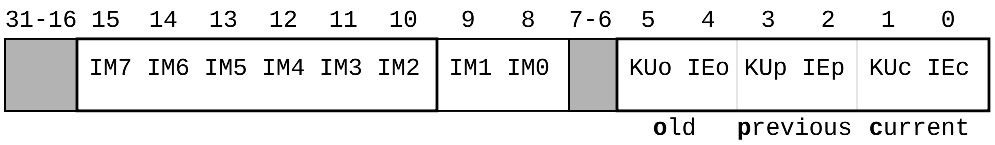
Code	Name	Description
0	INT	Interrupt
4	ADDRL	Load from an illegal address
5	ADDRS	Store to an illegal address
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data reference
8	SYSCALL	`syscall` instruction executed
9	BKPT	`break` instruction executed
10	RI	Reserved instruction
12	OVF	Arithmetic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

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 ■
```