Computer Organisation COMP2008

Lab Sheet 10 (starts session week 12, due in week 13)

Student Name and Number	
Date, Grade and Tutor signature, **max mark 5**	

Keep this cover sheet marked and signed by the tutor.

1. Preparation [Total max. mark: 1]

PCSpim version note: this lab will not run on older PCSpim versions 6.x.

DO NOT ATTEMPT TO DO THIS LAB unless you have studied thoroughly the materials listed below.

The main goal of today's lab is to explore the exception handling in MIPS architecture. Please study the following materials: the textbook Ed2, Ed3, chapter 5.6; Ed4, chapters 4.9, 6.6; Ed5, chapter 4.9 (changes in PH5: rather than isolating I/O into a single chapter, it has been spread throughout the book); Ed6, chapter 4.10; the **HP_AppA.pdf**: sections **A.5, A.7**, and **"Exceptions Handling in MIPS32 Architecture"** section at the end of this lab sheet; also online materials if needed.

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

Question [1 mark] Refer to the Example Interrupt Routine below (this is a slide taken from Lecture 7 notes) and describe the workings of this Interrupt Routine. Note that, this is only a fragment of larger code to illustrate the general approach, thus some parts are missing. While you can mention the missing parts where it helps to explain the flow of code, you do not have to recreate them.

1) To illustrate your explanation, please draw a block diagram (flow-chart like schematic drawing) to show the logical and functional structure of this Example Interrupt Routine. Hint: the information shown in the Example Interrupt Routine is about *exception handling*, where determining the types of exceptions and taking corresponding actions are certainly required.

2) Explain the role of each individual *special register* used in this Example Interrupt Routine.

/*Example Interrupt Routine*/

- Place at 0x80000180

```
.text 0x80000180
mfc0 $k0,$13                # $13 is Cause reg
mfc0 $k1,$14                # $14 is EPC reg
# Why don't have to save $k0, $k1?
```

- Exception field is bits 5 to 2; 0000 ⇒ I/O

```
andi $k0,$k0,0x003c         # select 5-2
bne $k0,$zero, OtherException
```

- Read byte

```
sw $ra, save0($0)           # save old $31
jal ReadandStoreByte
lw $ra, save0($0)           # restore $31
jr $k1
```

2. Workshop Task I [Total max. mark: 1.5]

1. Run program *instructions.s* It generates six Bad Data/Stack Address Exceptions (Exception 7).
   Question 1 (0.6): List all six instructions which cause Exception 7, and explain why they fail (do not fix up these errors in the program!).

2. Single step the *instructions.s* program through at least two exceptions. Make sure that you see how the flow of control is changed by comparing the source code with PCSpim windows, pay particular attention to all registers.
   Question 2 (0.4): how do you recognise that the normal program flow has been interrupted?

   Question 3 (0.5): how can you find out what exception condition has occurred and what caused it?

3. Workshop Task II [Total max. mark: 1.0]

Use PCSpim menu: Simulator –> Settings, and note the location of default Exception file. Write down the location of the Exception file (if using QtSpim, the location of the default exception handler isn't given explicitly, ignore this step):

_ _

Question (1 mark): analyse how *exceptions.s* works (*exceptions.s* is provided with this lab). Draw a one-page block diagram (flow-chart like) to show the logical and functional structure of this code. Note that, this program is about exception handling, where determining the types of exceptions and taking corresponding actions should be illustrated.

4. Workshop Task III [Total max. mark: 1.5]

In PCSpim/QtSpim Settings, change the default exception file *exceptions.s* to *modified_exceptions.s* provided with this lab. At this stage both exceptions files are identical.

Question 1 (1 mark): add code to *modified_exceptions.s* to change its functionality when it encounters Exception 7. The modified code should do the following:

a. print out a special message indicating that it is adjusting for Exception 7
b. add value 0x10010000 to contents of register $t1
c. resume the current instruction that caused the Exception

Notes: the code to implement the requirements above can be written in different ways. For instance, you can do your coding in a couple of steps to address these requirements incrementally and integrate it finally.

Question 2 (0.5): Run the program *instructions.s* You should now get only two Exception 7 occurrences (note that PCSpim is now using *modified_exceptions.s* that you changed as per the instructions above). Which two instructions are causing two exceptions and why?

Note: After completing Workshop Task III, please restore the original *exceptions.s* file in PCSpim (in QtSpim, use default exception handler).

5. Summary of the Lab Task

The lab task is to modify the exception handler so it actually does something in addition to printing the error message. To protect the original *exceptions.s* handler you are provided with a copy named *modified_exceptions.s* which is initially the same as the original exception handler. **Do not** modify the original version of the exception handler, and remember to restore it when you are done. **Do all modifications to *modified_exceptions.s* only!**

Your modified exception handler should print a special console message, adjust for exception 7 by shifting data request from address zero up to address 0x 1001 0000, and restart instruction which failed (do not bump EPC). For detailed instructions see the lab sheet.

Hint: somewhere in the *modified_exceptions.s* code you need to check for "exception 7", and if it occurs, jump to new block of code which you have to write. If there was no "exception 7" the exception handler continues executing unmodified code.

6. Assessment notice

When you ready, present to the tutor a printed copy of your program source code, with your name and student number included in the comments (#…), and typed or neatly written answers to questions, if there are any listed in the lab sheet. Your tutor may decide to keep the source code printout, but you should keep marked and signed cover sheet.

Warning: Any source code duplicated amongst students will result in a zero mark, and possible further action according to the WSU policy on plagiarism.

Exceptions Handling in MIPS32 Architecture

This lab requires little coding, but **you need to have a very good understanding** of exceptions and interrupts handling, and understand the role of PCSPIM control program *exceptions.s.* Specifically you have to understand what mechanism is triggered when you run provided program *instructions.s*, understand why it fails, and how Bad Data/Stack Address Exceptions messages are generated by *exceptions.s*

The textbook edition differences: the second edition of the textbook refers to the older MIPS-1 architecture, while the third edition (and this note) refers to more current MIPS32 architecture. If you have the second edition, please use Appendix from the third addition which is provided in PDF format. This note quotes Appendix page references from Ed.3.

1. Kernel / User mode

Generally all programs can run in one of two modes: **user mode** (in user address space in memory) or **kernel mode** (in kernel address space in memory). Controlling the mechanism of switching between both modes is challenging, it varies between different architectures, and moreover terminology is not always the same in different implementations. Terminology used here complies with MIPS convention.

Kernel mode is used by different routines serving different purpose, but the principle remains the same: when an exception, interrupt or system call occurs (see terms explanation below), the processor stops processing instructions, saves sufficient state to later resume the interrupted instruction stream, enters Kernel mode, and starts a piece of software which handles the exception (exception handler). What to save, and what memory address to go to depends on both the type of exception and the current state of the processor, and is described in more details below.

2. Exceptions, Interrupts and Exception Handler Defined

Exceptions

Synchronous exception (implicit transfer to operating system or unprogrammed trap) occurs always at the same place when a program is executed. Examples are: arithmetic overflow, using an undefined instruction (see the textbook chapter 5.6).

System calls (explicit transfers to operating system, or programmed traps) are also implemented using synchronous exception mechanism (not covered here).

Asynchronous exceptions may happen any time during a program execution. Examples are: I/O requests, memory errors, and hardware errors.

Interrupts

Interrupts refer to externally caused asynchronous exceptions. Interrupts are used by I/O devices to communicate with the processor.

Exception Handler

Exception Handler (sometimes also referred to as trap handler) is a piece of software which runs in kernel address space and handles the above conditions (details follow). You will see examples of synchronous exceptions when running program *instructions.s* in the lab, and you will also be modifying PCSpim handler *exceptions.s*

3. How MIPS Handles Exceptions

In MIPS architecture **coprocessor 0** records all information the software needs to handle exceptions. SPIM implements only some registers from full MIPS architecture, specifically registers numbered: 8,12,13,14 and named: **BadVAddr, Status, Cause** and **EPC** – you can see them in the top area of PCSPIM window (the textbook also explains role of registers Count, Compare and Config, but this is beyond the scope of this paper). These registers can be accessed by load, store and move type of instructions: lwc0, mfc0, mtc0 and swc0, which are described in more details in the textbook Appendix A section A.7 and below. Note part 'c0' which indicates that the instructions are used by coprocessor 0.

Changing normal program flow when an exception occurs: the hardware automatically copies PC into EPC, puts correct code into Cause register, and PC is set automatically to 0x80000180. As a result the user program execution is suspended: the address of the instruction which was executed when the exception occurred can be found in the register EPC. The control is transferred to a different program, which is located in the kernel memory address space starting from the address 0x80000180. It is named *exceptions.s*.

How to handle different exceptions? Refer to *exceptions.s* code line 87, instruction mfc0:
exception handler checks Cause register bits 2 to 6 (see the Appendix A Fig A.7.2), prints appropriate messages, and jumps to the part of the code which handles the current exception.

What is the role of *exceptions.s* when you run user code *instructions.s* ? It only prints messages on the screen informing about errors, but it does not fix or change any conditions. Why? Because the purpose of *exceptions.s* is mainly to demonstrate how the mechanism works, it does not do any "real work". Its role is to simulate a part of operating system by providing an empty frame, which could be filled in with additional functionality (as for example in the lab task).

When the exception handling is completed *exceptions.s* has the task of restoring any registers it may have modified, and returning control to the original (user) program, which then continues. This is done as follows: *exceptions.s* copies the value from EPC to the PC, which returns the process to user mode, and resets state to the way it was before the interrupt: see *exceptions.s* code line 158, instruction eret (exception return). To skip the instruction which caused the exception EPC value is bumped by one word (*exceptions.s* code line 138).

Because the registers convention usage allocates registers $k0 and $k1 for the use of interrupt handler (see the Appendix A Fig A.6.1: "reserved for OS kernel"), the interrupt handler can use them without having to save them first, because by convention user programs are not supposed to use these registers. Note that register using conventions are not enforced, and ignoring them while coding causes insidious bugs.

4. Some Instructions Used in *exceptions.s*

Instruction	Explanation
`eret`	**exception return** (or return from exception)

`mfc0 rt, rd`
 	
example:
`mfc0 $s1, $epc`
	(rt←rd) – "**move from coprocessor 0**": move the contents of coprocessor's 0 register rd to general purpose register rt, example:
	
	$s1 gets contents of $epc

`mtc0 rt,rd`
 	
example:
`mtc0 $0, $13`
	(rt→rd) – "**move to coprocessor 0**": moves the data from general purpose register rt to coprocessor's 0 register rd, example:
	
	Copies value 0 to $13, or: resets $13 to 0

`lwc0 C0dest, address`	**load** word from address in register C0dest
`swc0 C0src, address`	**store** the content of register C0src at address in memory