First Partial Exam
MIPS Exceptions & Interrupts

1 Learning outcomes

- To understand how exceptions and interrupts are configured and processed in the Microprocessor without Interlocked Pipeline Stages (MIPS) Microprocessor (μP).
- To understand how Input/Output (IO) devices, such as a keyboard, communicates with μPs using interrupts.

2 Pre-requisites

For this exam, MIPS Assembler and Runtime Simulator (MARS) must be already installed and you must be familiar the basic steps for running and debugging MIPS assembly programs.

3 Background

MIPS μP consists of a Central Processing Unit (CPU) and two coprocessors, CoProcessor 0 (CP0) and CoProcessor 1 (CP1), as shown in Figure 1.

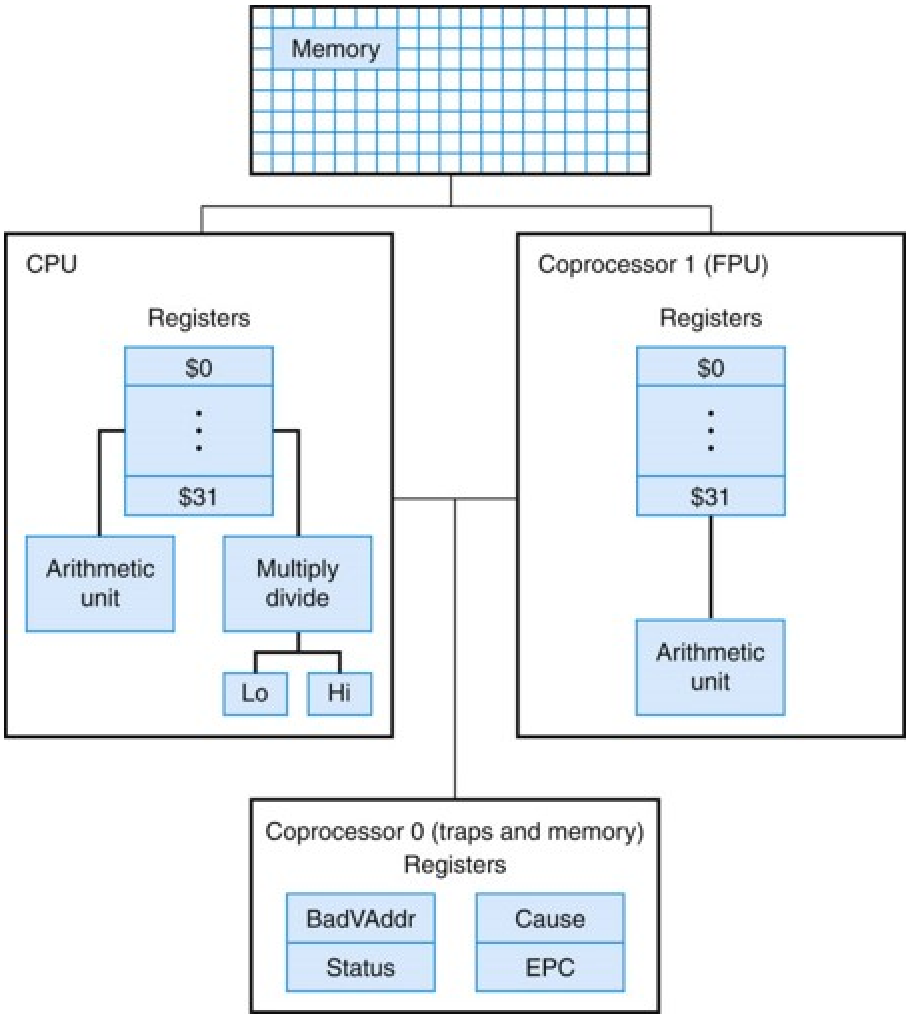


Figure 1: MIPS coprocessors.

Here, the CPU deals with integer arithmetic and logic operations using an Arithmetic and Logic Unit (ALU) and a multiply/divide unit, which store their result in general purpose registers. In contrast to this, CP1 is a Floating Point Unit (FPU) processor. Finally, CP0 contains a set of registers for handling exceptions and interrupts, among other special functions.

3.1 MIPS CP0

MIPS CP0 comprises 32 registers. Four of those 32 registers are reserved for handling exceptions and interrupts and briefly described in the following sections.

BadVAddr

Located at CP0 address 8. This register contains the address that caused a bad address exception.

Status register

Located at CP0 address 12. This register is used both for configuring exceptions and interrupts and for gathering information about the exception or interrupt that occurred. Figure 2 and Table 1 show the relevant fields of the status register.

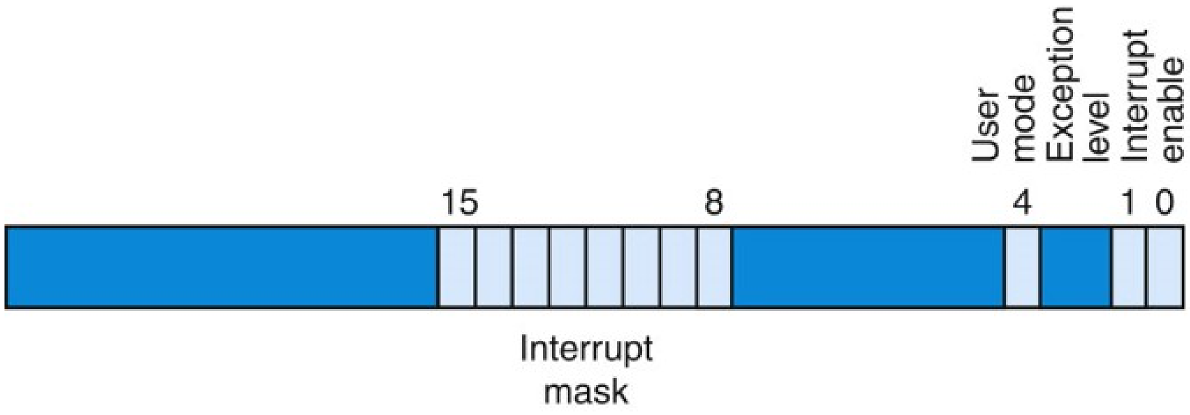


Figure 2: MIPS CP0 status register.

Table 1: MIPS CP0 status register description.

Bit	Name	Description
0	Interrupt enable	Set to 1 to enable global interrupts.
1	Exception level	Set to 1 after an exception occurs. When set to 1, interrupts are disabled.
2-3	-	Unused.
4	User mode	Set to 0 when MIPS runs in kernel mode. Set to 1 when MIPS runs in user mode.
5-7	-	Unused
8-15	Interrupt mask	Each bit enable a possible interrupt source.
16-31	-	Unused.

As described in Table 1, bit 0 of the status register must be set to 1 in order to enable global interrupts to occur in the MIPS. The exception level bit, normally 0, is set to 1 after an exception occurs. When this bit is 1, interrupts are disabled, preventing an exception handler from being disturbed by an interrupt or exception, but it should be reset when the handler finishes. The user mode bit, indicates whether a program is running in user of kernel mode. However, MARS does not implement kernel mode and it defaults program execution to user mode. The interrupt mask bits are used for enabling each of the 8 possible sources of interrupts in the MIPS. More specifically, the 6 Most Significant Bit (MSB) of the interrupt mask bits are related to interrupts such as timers, real-time clocks and serial ports. Similarly, bit 1 and 0 of the interrupt mask correspond to the transmitter (terminal output) and receiver (keyboard receiver), respectively.

NOTE: MARS initializes this register by default. In your exam problems described in Section 4, you should first investigate the initial (startup) value of this register.

Cause register

Located at CP0 address 13. MIPS cause register of Figure 3 indicates which interrupt or exception has occurred.

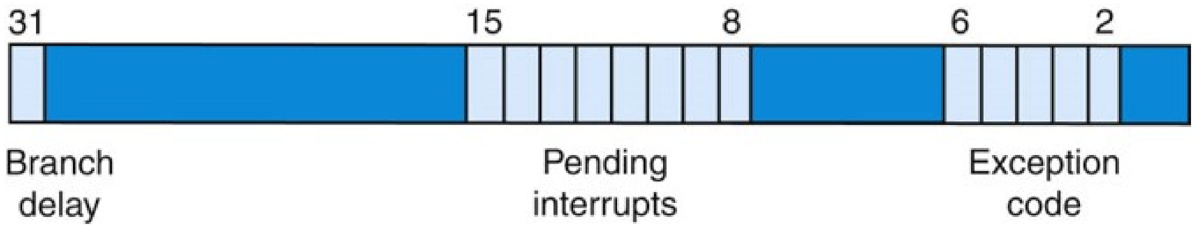


Figure 3: MIPS CP0 cause register.

Exception code bits of Figure 3 indicate the reason of the exception or interrupt, as shown in Table 2

Table 2: MARS exception codes.

Exception code	Name	Exception source
0	Int	Interrupt (hardware)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error on instruction fetch
7	DBE	Bus error on data load or store
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor unimplemented
12	Ov	Arithmetic overflow exception
13	Tr	Trap
15	FPE	Floating point

Note that Table 2 does not include all the 32 possible exception codes. This is due to MARS supporting only the 12 exception sources of Table 2. The interrupt pending bits of Figure 3 become 1 whenever an interrupt occurs. Finally, the branch delay bit of Figure 3 is set to 1 if the last exception occurred in an instruction executed in the delay slot of a branch.

Exception Program Counter (EPC)

Located at CP0 address 14. EPC stores the address of the instruction that was being executed when the exception or interrupt occurred. When an interrupt or exception occurs, the contents of Program Counter (PC) are copied to EPC before executing the handler routine. In this way, the program that was executed before the arrival of the exception or interrupt may be restored and resumed.

4 Exam problems

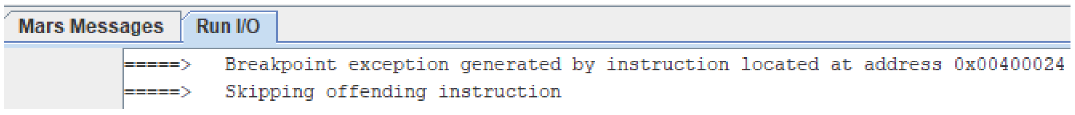
You are provided with the assembly language source file `first_partial_exam.s`. This file contains a basic template for performing each of the different problems to solve in this exam, which are specified in the following sections.

Each exam problem is independent from each other, *i.e.*, there's no specific order in which you must solve the exam. In order to work on a specific problem at the time, you may simply comment or uncomment the line of code that calls each subroutine for each problem. For example, if you are working exclusively on Problem 2, you may comment out calls to Problems 1, 3 and 4. However, note that all problems may have instructions in common, as detailed in Section 5.

4.1 [15 points] Problem 1: Resolve a breakpoint exception

Once the subroutine for this program is called, it executes a `break` instruction, which triggers an exception. In order to solve this problem, you should:

1. Detect the breakpoint exception.
2. Create an exception handler for this exception. This exception handler must display a message similar to that of Figure 4 for indicating that this exception has occurred and the address of the instruction that caused it.

   ```
   =====>   Breakpoint exception generated by instruction located at address 0x00400024
   =====>   Skipping offending instruction
   ```

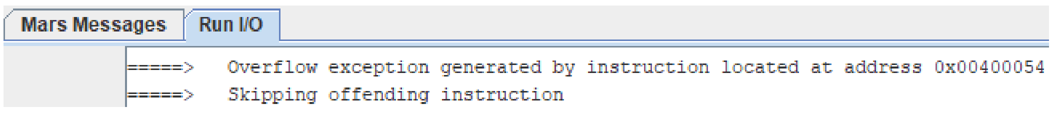
   Figure 4: Breakpoint exception detected.

   This exception handler must also be able to skip the offending `break` instruction.
3. After the handler finishes its execution, the program should return to main.

4.2 [20 points] Problem 2: Generate and solve an overflow exception

In order to solve this problem, you should:

1. Generate an arithmetic overflow exception.
2. Detect such exception.
3. Detect the address of the offending instruction.
4. Create an exception handler for this exception. This exception handler must display a message similar to that of Figure 5 for indicating that this exception has occurred and the address of the offending instruction.



```
=====>   Overflow exception generated by instruction located at address 0x00400054
=====>   Skipping offending instruction
```

Figure 5: Arithmetic overflow exception detected.

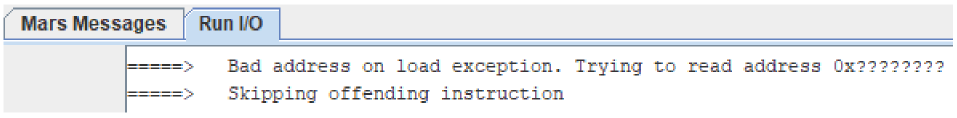
   This exception handler must also be able to skip the offending instruction.

5. After the handler finishes its execution, the program should return to main.

4.3 [25 points] Problem 3: Generate and solve a bad address exception on both a load and a store instruction

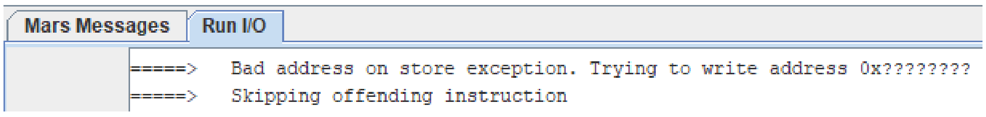
In order to solve this problem, you should:

1. Generate two exceptions: one caused by a bad load address, one caused by a bad store address.
2. Detect such exceptions.
3. Detect the wrong address used in the load or store instruction.
4. Create an exception handler for each of these two exceptions. Each exception handler must display a message similar to that of Figure 6 and Figure 7 for indicating that these exception have occurred and the bad address (in hexadecimal) that triggered the exception.

   ```
   =====>   Bad address on load exception. Trying to read address 0x????????
   =====>   Skipping offending instruction
   ```

   Figure 6: Bad address on load exception detected.

   ```
   =====>   Bad address on store exception. Trying to write address 0x????????
   =====>   Skipping offending instruction
   ```

   Figure 7: Bad address on store exception detected.

   NOTE: You should replace ???????? in Figure 6 and Figure 7 with the actual bad addresses. These exception handlers must also be able to skip their offending instructions.

5. After the handler finishes its execution, the program should return to main.

4.4 [40 points] Problem 4: Read keyboard characters using interrupts

This program processes keyboard interrupts. All American Standard Code for Information Interchange (ASCII) letter characters received from the keyboard through MARS **MMIO Simulator** must convert their case from upper to lower and from lower to upper. For example, if the letter `a` is pressed in the keyboard, the ASCII character `A` must be printed in MARS **Run IO** pane. Similarly, if `B` has been pressed, `b` must be displayed. All other characters such as numbers and special characters do not require any processing, *i.e.*, they can be printed exactly as they are received. In order to simplify this problem, you may simply use `syscall` instructions for printing the characters in MARS **Run IO** pane. This problem must enter into an infinite loop. In this infinite loop, your program should simply wait for keyboard interrupts to occur and to be processed by the handler. Keyboard polling is not allowed.

5 General considerations

Some lines of code in your solution may be shared across all four exam problems. More specifically, instructions for detecting the source of the exception/interrupt may be shared for all four problems.

In addition to the solution provided for solving each of the exam problems, the following items will be considered when marking your exam.

- Code readability.
  - It is easier to read meaningful names of registers than hard-coded addresses. For example, it is easier to understand the instruction

    ```
    sw $t0, DISPLAY_CTRL
    ```

    than

    ```
    sw $t0, 0xffff0008.
    ```
  - Meaningful comments.
- Code re-usability.
  - Use macros or subroutines for frequently used code.

6 Deliverables and submission

1. `first_partial_exam.s`, which should include your code for all exam problems.

Submit your exam through Canvas before 18:30 hours on Thursday September 10th 2020. Please send any questions to isaac.perez.andrade@tec.mx.