8 Exceptions and I/O

8.1 Objectives

After completing this lab, you will:

- Understand the exception mechanism in MIPS
- Understand Coprocessor 0 instructions
- Write exception handlers
- Understand Memory Mapped I/O

8.2 Exception Mechanism in MIPS

Branches and jumps change the control flow in a program. Exceptions also change the control flow. The MIPS architecture calls an **exception** any **unexpected** change in control flow, regardless of its source.

An exception is said to be **synchronous** if it is caused by an instruction in the running program. Examples of synchronous exceptions are arithmetic exceptions, invalid memory addresses generated by load and store instructions, and trap instructions.

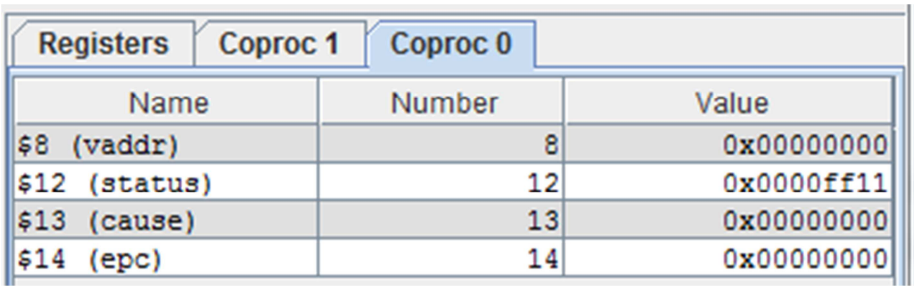
An exception is said to be **asynchronous** if it is caused by an I/O device requesting the processor. This is also known as a hardware **interrupt**, which is not related to program execution. Interrupts can be caused by a variety of I/O devices, such as the keyboard, timer, and disk controller.

When an exception happens, control is transferred to an **exception handler**, written specifically for the purpose of dealing with exceptions. After executing the exception handler, control is returned back to the program. The program continues as if nothing happened. The exception handler appears as a procedure called suddenly in the program with no parameters and no return value.

The MIPS processor operates either in **user** or **kernel mode**. User programs (applications) run in user mode. The CPU enters the kernel mode when an exception happens. The exception handling mechanism is implemented by **Coprocessor 0**, which has several important registers, such as: **`vaddr`**, **`status`**, **`cause`**, and **`EPC`**, which record information about an exception.

1. **`Vaddr`** (**`$8`**): Contains the invalid memory address caused by load, store, or fetch.
2. **`Status`** (**`$12`**): Contains the interrupt mask and enable bits (see below).
3. **`Cause`** (**`$13`**): Contains the type of exception and any pending bits (see below).
4. **`EPC`** (**`$14`**): Contains the address of the instruction when the exception occurred.

The MARS tool shows the values of registers: **vaddr**, **status**, **cause**, and **epc** under the Coprocessor 0 tab, as shown in Figure 8.1.



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff11
$13 (cause)	13	0x00000000
$14 (epc)	14	0x00000000

Figure 8.1: Coprocessor 0 registers in the MARS tool

Examples of exceptions are shown in Figure 8.2. The first example initializes register **$t0** with **0x7fffffff** and **$t1** with **1**. The **addu** instruction ignores overflow. However, **add** detects and causes an arithmetic overflow exception. The second example is about storing data at an illegal address in memory. The third example is about loading a word from a misaligned address in memory. The last example is about inserting a breakpoint (**break** instruction) inside the program. All of these examples cause exceptions and require special handling.

Examples of Exceptions	Coprocessor 0 Registers

`# Arithmetic Overflow Exception`
`li $t0, 0x7fffffff # $t0 = MAX_INT`
`li $t1, 1 # $t1 = 1`
`addu $t2, $t0, $t1 # Ignores Overflow`
`add $t3, $t0, $t1 # Detects Overflow`
	Registers | Coproc 1 | Coproc 0
	Name / Number / Value
	$8 (vaddr) 8 0x00000000
	$12 (status) 12 0x0000ff13
	$13 (cause) 13 0x00000030
	$14 (epc) 14 0x00400010

`# Store Address Exception`
`# Cannot store at address 4`
`li $t0, 4`
`li $a0, 5`
`sw $a0, ($t0)`
	Registers | Coproc 1 | Coproc 0
	Name / Number / Value
	$8 (vaddr) 8 0x00000004
	$12 (status) 12 0x0000ff13
	$13 (cause) 13 0x00000014
	$14 (epc) 14 0x0040001c

`# Misaligned Load Address Exception`
`.data`
`arr: .word 12, 17`
`. . .`
`la $t0, arr`
`lw $t0, 1($t0)`
	Registers | Coproc 1 | Coproc 0
	Name / Number / Value
	$8 (vaddr) 8 0x10010001
	$12 (status) 12 0x0000ff13
	$13 (cause) 13 0x00000010
	$14 (epc) 14 0x00400028

`# Breakpoint Exception`
`# Caused by the break instruction`
`.text`
`. . .`
`break`
	Registers | Coproc 1 | Coproc 0
	Name / Number / Value
	$8 (vaddr) 8 0x00000000
	$12 (status) 12 0x0000ff13
	$13 (cause) 13 0x00000024
	$14 (epc) 14 0x0040002c

Figure 8.2: Examples of exceptions and corresponding values of Coprocessor 0 registers

The second column of Figure 8.2 shows Coprocessor 0 registers when an exception happens. The Exception Program Counter (**EPC**) register **$14** stores the address of the instruction that caused the exception. The value of **EPC** in Figure 8.2 is **0x00400010**, which is the address of the **add** instruction that caused arithmetic overflow. It is **0x0040001c**, which is the address of **sw** that attempted to write to an illegal address in memory. It is **0x00400028**, which is the address of **lw** that generated a misaligned address in memory.

The **cause** register **$13** provides information about the cause of an exception (**exception code**) and about pending interrupts, if any. The exception code is stored in bits 2 to 6 of the cause register. The bit fields of the cause register is shown in Figure 8.3.

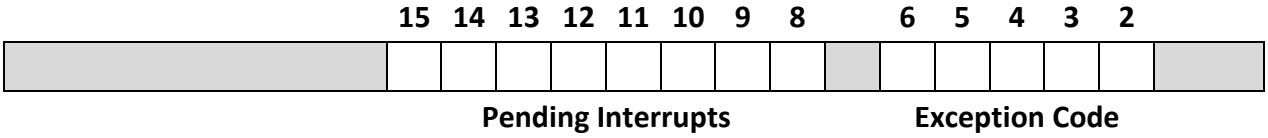


Figure 8.3: The **Cause** Register **$13**

The MIPS architecture defines exception codes for different types of exceptions. Some are listed in Figure 8.4. The MARS tool simulates some of these exception codes.

Code	Name	Description
0	INT	Hardware Interrupt
4	ADDRL	Address error exception caused by load or instruction fetch
5	ADDRS	Address error exception caused by store
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data load or store
8	SYSCALL	System call exception caused by the **syscall** instruction
9	BKPT	Breakpoint exception caused by the **break** instruction
10	RI	Reserved instruction exception
12	OVF	Arithmetic overflow exception
13	TRAP	Exception caused by a trap instruction
15	FPE	Floating-Point exception cause by a floating-point instruction

Figure 8.4: Some of the MIPS exception codes

If a load, store, jump, or branch instruction generates an exception, then **vaddr** (register **$8**) contains the invalid memory address that caused the exception. Consider again the examples of Figure 8.2, the illegal data address of **sw** that caused the exception is **vaddr = 0x00000004** and the misaligned data address of **lw** that caused the exception is **vaddr = 0x10010001**.

The **status** (register **$12**) is shown in Figure 8.5. Bit 0 is the Interrupt Enable (**IE**), which enables or disables interrupts. Bit 1 is the Exception Level (**EL**). The **EL** bit is normally 0, but is set to 1 after an exception occurs.

The **interrupt mask field** contains 8 bits and supports a bit for each of the six hardware and two software interrupt levels. A mask bit that is 1 allows interrupts at that level to interrupt the processor. A mask bit that is 0 disables interrupts at that level. When an interrupt arrives, it sets its

interrupt pending bit in the cause register, even if the mask bit is disabled. When an interrupt is pending, it will interrupt the processor when its mask bit is subsequently enabled.

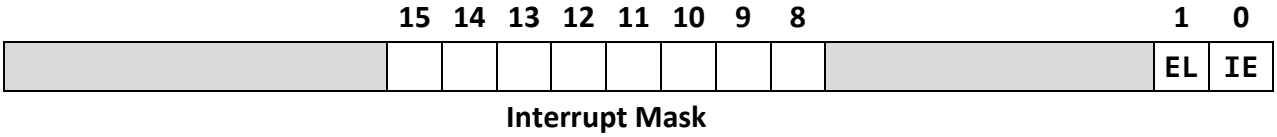


Figure 8.5: The **`Status`** Register **`$12`**

8.3 Coprocessor 0 Instructions

The MIPS architecture defines special instructions that cause exceptions and switch the processor from user to kernel mode. These are the **`trap`**, **`break`**, and **`syscall`** instructions:

Instruction	Meaning
`teq Rs, Rt`	Raise the trap exception if register **`Rs`** is equal to register **`Rt`**
`tne Rs, Rt`	Raise the trap exception if register **`Rs`** is not equal to register **`Rt`**
`tlt Rs, Rt`	Raise the trap exception if register **`Rs`** is less than register **`Rt`**
`. . .`	There are other trap instructions not listed here (see Appendix B)
`break code`	Raise the breakpoint exception. Code 1 is reserved for the debugger
`syscall`	Raise the system call exception. Service number is specified in **`$v0`**

The trap instructions (**`teq`**, **`tne`**, … etc.) raise the TRAP exception code 13. The **`break`** instruction raises the breakpoint exception code 9. However, the MARS simulator provides services for the **`syscall`** instruction without raising an exception. On a real system, the **`syscall`** instruction raises the system call exception code 8, which is serviced by the operating system.

When an exception occurs, the processor switches to kernel mode. Coprocessor 0 registers can be accessed only when the processor is servicing an exception in kernel mode. Register values can be transferred from and to coprocessor 0 using the following instructions. Load and store instructions that transfer data between coprocessor 0 registers and memory are also available. These instructions can be used when writing an exception handler.

After an exception is processed, the exception handler can return back and resume the execution of a program. The **`eret`** instruction is used to return from an exception.

Instruction	Meaning
`mfc0 Rd, C0src`	Move from Coprocessor 0 register **`C0src`** into destination register **`Rd`**
`mtc0 Rs, C0dst`	Move to Coprocessor 0 register **`C0dst`** the value of register **`Rs`**
`lwc0 C0dst, addr`	Load a word from memory into Coprocessor 0 register **`C0dst`**
`swc0 C0src, addr`	Store Coprocessor 0 register **`C0src`** in memory
`eret`	Reset **`EL = 0`** (back to user mode) and return: **`PC = EPC`**

8.4 Exception Handlers

The layout of a MIPS program is shown in Figure 8.6. The Operating System appears in the upper half of the address space, which can only be accessed when the processor is running in kernel mode. The OS kernel text segment starts at address **0x80000000** and the kernel data segment starts at address **0x90000000**. The last segment of the address space is mapped to I/O devices, starting at address **0xffff0000**. This is known as Memory-Mapped I/O (MMIO). The default memory configuration is shown in Figure 8.6. It is also possible to change the memory configuration under the MARS tool settings.

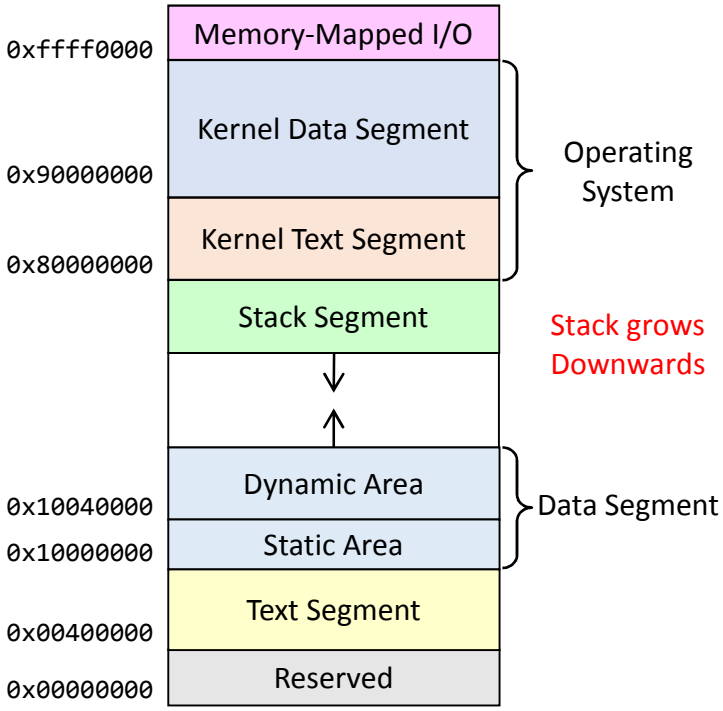


Figure 8.6: The layout of a MIPS program in memory

When a function is called using **jal**, control is transferred at the address provided by the instruction and the return address is saved in register **$ra**. In the case of an exception there is no explicit call. In MIPS, when an exception occurs, control is transferred at the fixed address **0x80000180**. The exception handler must be located at that address.

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

An exception handler can be written in the same file as the regular program, or in a separate file. The exception handler must start at the fixed address **0x80000180**. This address is in the kernel text segment. If there is no instruction at address **0x80000180**, MARS will terminate the MIPS program with an appropriate error message. An example of an exception handler that prints the address of the instruction that caused the exception and the exception code is shown below:

```
# Exception Handler starts here
.ktext  0x80000180

move   $k0, $at          # $k0 = $at
la     $k1, _regs        # $k1 = address of _regs
sw     $k0, 0($k1)       # save $at
sw     $v0, 4($k1)       # save $v0
sw     $a0, 8($k1)       # save $a0

la     $a0, _msg1        # $a0 = address of _msg1
li     $v0, 4            # $v0 = service 4
syscall                  # Print _msg1
mfc0   $a0, $14          # $a0 = EPC
li     $v0, 34           # $v0 = service 34
syscall                  # print EPC in hexadecimal

la     $a0, _msg2        # $a0 = address of _msg2
li     $v0, 4            # $v0 = service 4
syscall                  # Print _msg2
mfc0   $a0, $13          # $a0 = cause
srl    $a0, $a0, 2       # shift right by 2 bits
andi   $a0, $a0, 31      # $a0 = exception code
li     $v0, 1            # $v0 = service 1
syscall                  # Print exception code

la     $a0, _msg3        # $a0 = address of _msg3
li     $v0, 4            # $v0 = service 4
syscall                  # Print _msg3

la     $k1, _regs        # $k1 = address of _regs
lw     $at, 0($k1)       # restore $at
lw     $v0, 4($k1)       # restore $v0
lw     $a0, 8($k1)       # restore $a0

mtc0   $zero, $8         # clear vaddr
mfc0   $k0, $14          # $k0 = EPC
addiu  $k0, $k0, 4       # Increment $k0 by 4
mtc0   $k0, $14          # EPC = point to next instruction

eret                     # exception return: PC = EPC

# kernel data is stored here
.kdata

_msg1: .asciiz   "\nException caused by instruction at address: "
_msg2: .asciiz   "\nException Code = "
_msg3: .asciiz   "\nIgnore and continue program ...\n"
_regs: .word 0:3         # Space for saved registers
```

Figure 8.7: Example of an exception handler

Writing an exception handler is no easy job. Things that should be done are:

- Save registers before modifying them by the exception handler.
- Read the coprocessor registers to determine what exception has occurred.
- Execute the specific handler for the exception code, usually via a jump table.
- Restore all the registers that were modified by the exception handler.
- Restart the user-mode program, or kill the program if it cannot be restarted.

The exception handler must preserve the value of any register it may modify, such that the execution of the interrupted program can continue at a later time. The MIPS architecture reserves register **`$26`** and **`$27`** (**`$k0`** and **`$k1`**) for the use of the exception handler. The handler can modify these two registers without having to save them first. Additional registers should be saved in memory before they can be modified by the exception handler. For example, the exception handler of Figure 8.7 saves register **`$at`**, **`$a0`**, and **`$v0`** in the kernel data segment.

The **`EPC`** contains the address of the instruction that has generated the software exception. The return address must be incremented by 4 to avoid executing the same instruction again. The return sequence from a software exception can be written as follows:

```
mfc0   $k0, $14        # $k0 = EPC
addiu  $k0, $k0, 4     # increment $k0 by 4
mtc0   $k0, $14        # EPC = point to next instruction
eret                   # exception return: PC = EPC
```

Rather than writing the exception handler at the end of every MIPS program, it is better to write the exception handler in a separate file, then open the "Exception Handler …" dialog box in the MARS settings and include the exception handler file in all your MIPS programs.

8.5 Memory Mapped I/O

In any computer system, input and output devices are outside the processor chip. A MIPS processor communicates with I/O devices using a technique called **memory-mapped I/O**. Using memory-mapped I/O, there is no need to add additional instructions to the MIPS instruction set. Any **`lw`** or **`sw`** instruction with an effective address of **`0xffff0000`** or greater will **not access** the main memory. These addresses are reserved to make access to registers in I/O devices. The I/O device controllers must be connected to the system I/O bus, as shown in Figure 8.8.

Unique address decode logic is associated with each I/O register. When the MIPS processor reads from or writes to one of these addresses, the processor is actually reading from or writing to a selected register in one of the I/O device controllers. The processor can read data about the state of the I/O device and write control data to change the state of the device.

The two registers associated with the keyboard are the receiver control and data registers. These are mapped to addresses **`0xffff0000`** and **`0xffff0004`** respectively. To communicate with the keyboard, the processor reads the control register. As long as the **ready bit** is zero, the processor keeps reading the control register in a loop. This approach is known as **polling**.

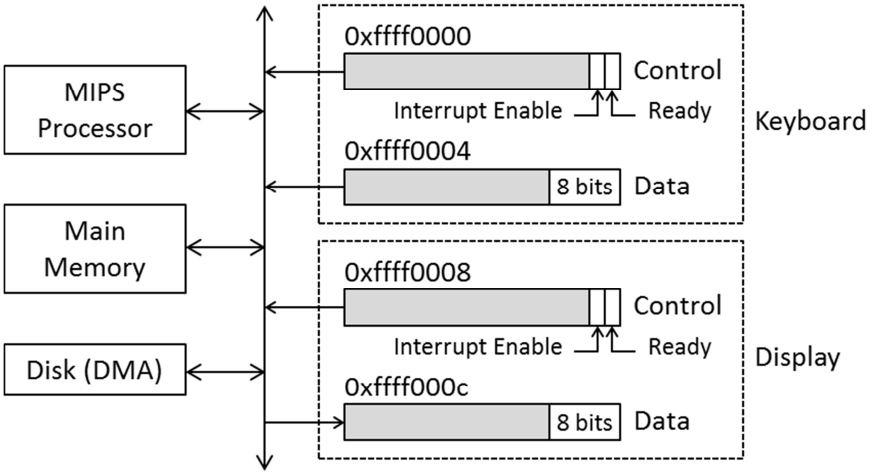


Figure 8.8: MIPS System I/O Bus

When a key is pressed, the data register stores the character and the **ready** bit is set. Then the processor reads the character. The following MIPS code provides an example of memory-mapped access to the keyboard control and data registers **via polling**:

```
   li     $t0, 0xffff0000      # Address of keyboard control register
   li     $t1, 0               # Initialize wait_counter = 0
wait_keyboard:
   lw     $t2, ($t0)           # Read the keyboard control register
   andi   $t2, $t2, 1          # Extract ready bit
   addiu  $t1, $t1, 1          # wait_counter++ (counts iterations)
   beqz   $t2, wait_keyboard   # loop back while not ready
   lw     $a0, 4($t0)          # Get character from keyboard
```

The rate that characters are typed on the keyboard is very slow compared to the rate that the MIPS processor can execute instructions. Typically, millions of instructions are executed until a key is pressed. Register **`$t1`** keeps track of the number of iterations in the **`wait_keyboard`** loop. The **`lw`** instruction outside the loop gets the character from the keyboard data register, which also clears the ready bit. A MIPS programmer can only read the keyboard data register. Writing to the keyboard data register has no effect.

Communicating with the display controller can be done via polling in a similar way. The display registers are mapped to addresses **`0xffff0008`** and **`0xffff000c`**. As long as the ready bit is zero, the processor keeps reading it in a loop. We must not store a character in the data register until the display is ready to receive it. The display controller clears the ready bit when a character is stored in the data register. It then sets the ready bit again after the character is displayed and it is ready to receive the next character to display.

```
   li     $t0, 0xffff0008      # Address of display control register
wait_display:
   lw     $t2, ($t0)           # Read the display control register
   andi   $t2, $t2, 1          # Extract ready bit
   beqz   $t2, wait_display    # loop back while not ready
   sw     $a0, 4($t0)          # Send character to display
```

The MARS simulator provides a tool to simulate the keyboard and display, as shown in Figure 8.9. Press "Connect to MIPS" to connect this tool to the MIPS program. You must activate this tool to communicate character-by-character with the keyboard and display. The code that communicates with I/O devices at this level is known as a **device driver**. This is one of the advantages of using a simulator to learn how to communicate directly with I/O devices.

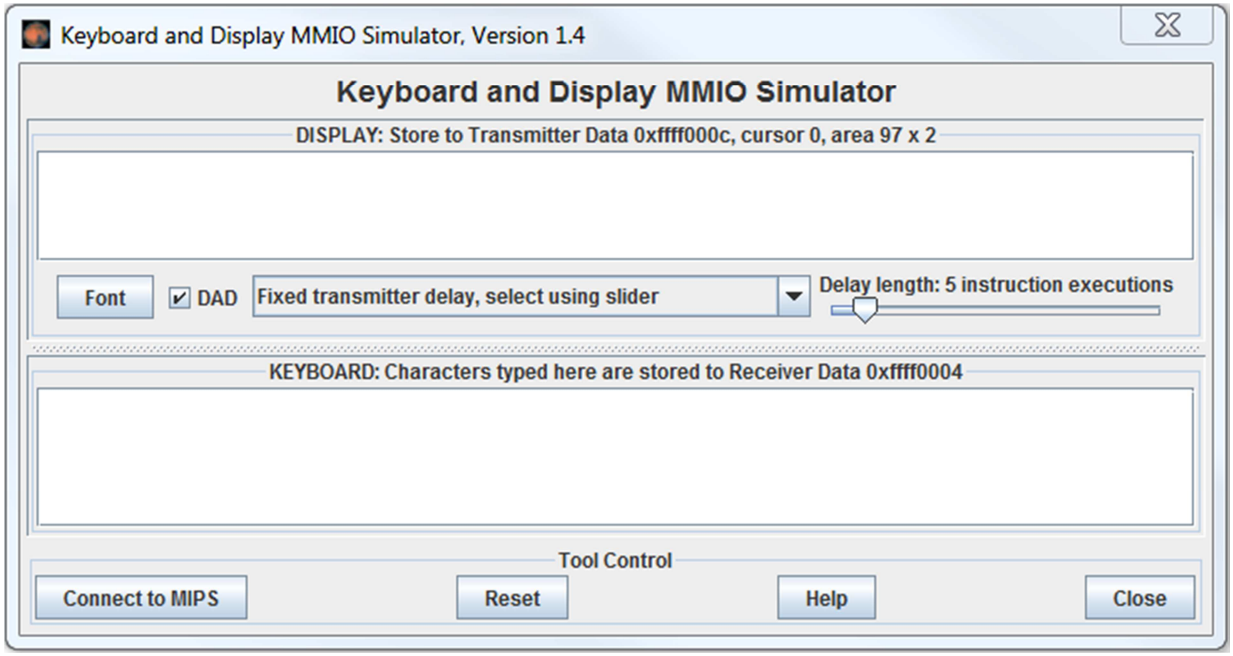


Figure 8.9: Keyboard and Display MMIO Simulator under the MARS Tools

The main drawback of polling is that it keeps the processor busy, wasting millions of cycles before the I/O device (such as the keyboard) becomes ready. An alternative to polling is to use interrupts. Interrupts can be enabled for the keyboard by setting the Interrupt Enable bit in the keyboard control register as follows:

```
li      $t0, 0xffff0000      # Address of keyboard control register
li      $t1, 2
sw      $t1, ($t0)           # Enable keyboard interrupt
```

When a key is pressed, the keyboard sends an interrupt signal to the MIPS processor and sets the **`cause`** register **`$13`** to the value **`0x00000100`**. Bit 8 in the cause register is set to indicate that the keyboard has interrupted the processor. An interrupt handler must be written to read the character from the keyboard and returning its value to the running program. This requires modifying the code of the exception handler shown in Figure 8.7 to deal with interrupts.

The Disk and Ethernet I/O device controllers use Direct Memory Access (DMA) to transfer blocks of data directly between the device and memory. As controllers become more complex, more than two registers are associated with each device controller. These devices are not part of the MARS simulator. In general, the supplier of any I/O device must provide programmers with an explanation of how to properly communicate with the I/O device controller.

8.6 In-Lab Tasks

1. Just before dividing **$s0** by **$s1**, the programmer might want to check that **$s1** is not equal to zero. Use **teq $s1, $zero** (trap if equal) to cause an exception. What is the address of the **teq** instruction in your program? What is the value of the **cause** register, the exception code, and the **epc** when the exception occurs?
2. Use **lb $t1, 5($zero)** to cause an exception when attempting to load a byte from address **5**. What is the address of the **lb** instruction in your program? What is the value of the **cause** register, the exception code, the **vaddr**, and the **epc** when the exception occurs?
3. Write a complete program that uses the exception handler of Figure 8.7 to generate multiple exceptions. The exception handler should report the address of the instruction that caused the exception, the exception code, and should resume the program after handling each exception. Insert instructions that cause overflow, invalid memory addresses, trap and break instructions.
4. Modify the exception handler of Figure 8.7 to print the invalid **vaddr**, when it is caused by a load, store, or instruction fetch (exception code 4 or 5). Test your exception handler by writing load and store instructions that generate invalid memory addresses.
5. Using memory-mapped I/O and polling, write a function **print_string** that prints a string on the display, without using any system call. The address of the string is passed in register **$a0** and the string must be null-terminated. Test this function by calling it from the main function. Make sure to activate the "Keyboard and Display MMIO Simulator".
6. Using memory-mapped I/O and polling, write a program that reads characters directly from the keyboard. To demonstrate how slow the keyboard device is, print the character pressed and the number of iterations after exiting the **wait_keyboard** loop. Repeat the execution of the program until the newline character is pressed. Make sure to activate the "Keyboard and Display MMIO Simulator" and to run the MARS simulator at maximum speed.
7. If the keyboard interrupt enable bit is set then the keyboard will interrupt the processor each time a key is pressed. Write a simple interrupt handler that returns the character pressed in register **$v0**. Rewrite the main function of question 6 using keyboard interrupts.

8.7 Bonus Problem

8. Using memory-mapped I/O and polling, write a function **read_string** that reads a string directly from the keyboard. The function will get characters from the keyboard and stores them in an array pointed by register **$a0**. The function should continue until **n-1** characters are read or the newline character is pressed. The parameter **n** should be passed in register **$a1**. The function should insert a NULL byte at the end of the string. It should return the actual number of characters read in register **$v0**. Make sure to activate the "Keyboard and Display MMIO Simulator". Write a main function to test **read_string** repeatedly, and to print the string after each call.