SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Shrey Sharma

System Architecture SS 2022 – Project 2

System Programming and Exception Handling

Submission Modalities

The project starts on July 1, 2022 with the release of this description. We recommend that you start working on the project *as soon as possible*. Use the tutorials to clear up any comprehension issues and/or problems with the MARS simulator.

You may work on the project in *groups of two to three people*. If you have formed a group, please create a corresponding team on your personal page in our CMS until

Monday, July 4, 2022, 23:59.

One person per group must upload the solutions of *both* parts of the project together to our CMS system by

Friday, July 15, 2022, 23:59.

A total of 32 points (plus 6 extra points) can be achieved. Projects submitted late will be graded with **0 points**. Use a ZIP file or a gzip compressed tar archive (i.e., `*.tar.gz`) for your submission. The archive should immediately contain all assembler files (i.e., do not use subfolders). Use the skeleton files we provide, which can be found in our CMS[1]. If you do not work on some of the problems, please do not submit the corresponding assembler files.

Comment your code in a meaningful way, i.e., describe the logical steps of your exception handling, but not every single instruction. Use the comments in the skeleton files as a guide. Unintelligible and uncommented code will result in **point deductions**.

If you worked on the project in a group, add a `contributions.txt` file to the archive that briefly describes how each team member contributed to the implementation. We may grade the project with 0 points for individual members if they have not made a significant contribution.

*Notes: Any collaboration with people who are not part of your own group is **not** allowed. We will check all submissions for plagiarism (against submissions from other groups, as well as submissions from previous years). Submissions that were created by modifying another project, for example by changing variable names, are also considered to be plagiarized. Plagiarized submissions will be graded with **0 points** and will be reported to the examination board as a cheating attempt; this may lead to expulsion from the university..*

*If you have attended the system architecture course in the past and would like to use your previous submission as a basis for the current project: This is only allowed for parts that you implemented yourself; code written by other members of your previous team may **not** be reused. In this case, add a file `previousyear.txt` to the submission that describes exactly which parts have been reused. Note, however, that the current project is not identical to projects from previous years. Submissions that only solve an old project will be graded with **0 points**.*

[1] `https://cms.sic.saarland/sysarch22/materials/`

Tools and Documentation

To test your assembler programs, use our adapted version of the MARS simulator, which in particular provides support for timer interrupts. This version is the same as the version used for the first project, but **different** from the original and the version for the Programming 2 course.

We present the most important information that you need for working on the project in the following sections. In addition to that, Part 3 of the official MIPS documentation provides additional information[2] on exception and interrupt handling. See [3] for a description of all MIPS instructions.

Background Information

In the following, we will describe background information required for the project. For all problems, you will write an exception handling routine. Such routines are usually part of the operating system.

An *exception* can occur for several reasons, such as a system call or the execution of an invalid instruction. However, an exception can also occur independently of the executed program due to external devices (keyboard, display), e.g., if a key was pressed on the keyboard. Such exceptions due to external devices are called *interrupts*.

The handling of exceptions requires some hardware support. In MIPS processors, the so-called *system coprocessor* provides this exception handling functionality. A coprocessor generally has its own set of registers and implements its own additional instructions. However, it also shares logic with the processor, e.g., for fetching instructions; hence *co*processor.

As soon as an exception occurs, the processor switches from *user mode* to the so-called *kernel mode*. In user mode, a program has only limited access to the machine, e.g., it can neither communicate with the system coprocessor nor use certain instructions nor access external devices. In kernel mode, on the other hand, a program has unrestricted access to the machine including the system coprocessor and external devices.

Registers of the System Coprocessor

The registers of the coprocessor 0 (CP0), i.e., the system coprocessor, are also called *special-purpose registers*, in contrast to the 32 mostly freely usable *general-purpose registers* of the main processor. For our exception handling purposes, we consider the following subset of the 32-bit wide special-purpose registers: *epc*, *status*, *cause* for general exception handling, and *count*, *compare* for programming timed periodic interrupts (timer interrupts).

Exception Program Counter *epc*

If an exception occurs, the current program counter is saved by the system coprocessor in *epc*. This register thus contains the address of the instruction at which the execution is (usually) resumed after exception handling. The *epc* is CP0 register 14 and can be read and written.
Note: The MARS simulator does not use delay slots (by default), which simplifies the implementation. So you can ignore information in the MIPS documentation about delay slots.

Status Register *status*

The status register contains information on the state of the system coprocessor, and it allows some configuration of the exception functionality. The status register is CP0 register 12 and can be read and written. For us, only the following bits are relevant.

Bit 0: *Interrupt Enable (IE)* indicates whether exceptions are allowed globally (1). If the bit is set to 0, no exceptions are generated.
Bit 1: *Exception Level (EXL)* indicates whether an exception is currently being handled. During exception handling, the processor is in kernel mode.
Bit 4 – Bit 3: *KSU* indicates in which mode the system is outside of the exception handling. We assume here for simplicity that this is always the *user mode* (bits 10).
Bit 15 – Bit 10: *Interrupt Mask IM[7] – IM[2]* indicates whether to respond to interrupt requests from external devices.

[2] https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00090-2B-MIPS32PRA-AFP-06.02.pdf
[3] https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf

Cause Register *cause*

The cause register contains at the time of an exception the reason for this exception. The cause register is CP0 register 13 and can be *read only*. For us, only the following bits are relevant.

Bit 6 – Bit 2: *ExcCode* indicates the reason for the exception. See Table 9.53 on page 212 of the MIPS documentation Part 3 for a table with all possible assignments. The assignment 00000 is important — it signals an interrupt.
Bit 15 – Bit 10: *Interrupt Pending IP[7] – IP[2]* indicates whether there is a current interrupt request from an external device. The positions correspond to the respective positions of *IM[7] – IM[2]* in the status register.

Timer Interrupts

Our MIPS system supports programmable timer interrupts. The timer is controlled by the count and compare registers. The timer interrupt is associated with IP[7] and IM[7]. The bit IP[7] is set to 1 by the timer as soon as a certain timestamp is reached, i.e., the compare and count registers have the same value. The timer sets the pending bit IP[7] back to 0 as soon as the compare or count register is overwritten.

Count Register *count* The count register, CP0 register 9, is incremented by one in each cycle unless it is written to by an instruction.

Compare Register *compare* The compare register, CP0 register 11, contains a timestamp. When the count register reaches this timestamp, a timer interrupt is signaled. The register is readable and writable.

Communication with the Coprocessor

The exception handling code itself is executed on the main processor. In order to have access to the coprocessor registers within this exception handling routine, there are two instructions for communication. With the `mfc0` instruction one can copy the value of a special-purpose register into a general-purpose register, and with the `mtc0` instruction one can copy a value into a special-purpose register.

Additional Instructions

The `eret` instruction returns from the exception handling routine to the main program. You can use the `syscall` instruction to request a system call from the operating system by a user program. It is a convention of the so-called *Application Binary Interface* to pass the number of the requested system call in `$v0` and a possible argument in `$a0`. For more information, please consult the MIPS documentation.

Convention: Reserved General-Purpose Registers

Even though all 32 registers of a MIPS processor (except register 0) are freely usable, there are certain conventions regarding their usage. An example that we have already seen is register 31 (called `$ra`), which manages the return address for function calls. An important convention for this project is to reserve registers 26 (`$k0`) and 27 (`$k1`) for the operating system — e.g., for our exception handling. Thus, registers 26 and 27 must *not* be used by ordinary programs.

Example Workflow Exception Handling

We would like to execute the following program:

```
...
100: li $a0, 1234
104: li $v0, 1
108: syscall
10c: add $a0, $a0, 1
...
```

What does the execution look like on a MIPS machine? // comments refer to the hardware, # comment refer to the exception handling routine.

```
...
108: syscall
// Generate exception:
// - Set epc to 108
// - Set kernel mode status[EXL] = 1
// - Set cause[ExcCode] to 8
// - Set pc to the start of the exception handling routine 0x80000180
# Execute exception handling routine
# - Save registers that are used by this routine
# - Examine ExcCode
# - React to exception/interrupt
# - Add (if necessary) 4 to epc, here: to not repeat system call
# - Execute eret
// Return to the program
// - Set pc to epc
// - Set user mode status[EXL] = 0
10c: add $a0, $a0, 1
...
```

Exception Generation

We have already seen the general exception handling workflow, but not when an exception is generated. There are two possibilities here. First, an exception can be generated by the executing program, e.g., intentionally by a system call or unintentionally by a division by 0. Second, an exception can be generated by an interrupt request of an external device. The coprocessor generates an exception as soon as a non-masked interrupt request is present (*cause[15:10]* & *status[15:10]* $\neq$ 000000), interrupts are globally enabled (*status[IE]* = 1), and the processor is not currently handling an exception (*status[EXL]* = 0).

Memory-Mapped I/O Devices

Here, we provide some background information on the external input/output devices. A specific part of the address space (from `0x00000000` to `0xffffffff`) is not connected to the main memory, but to so-called *ports* of external devices. So you can communicate with an external device by reading from and writing to the addresses of these ports. Therefore, this approach is also called *memory-mapped input/output*.

The MARS simulator provides two (virtual) external devices: a keyboard and a display. Each of these devices has two ports: a control port and a data port. The lowest bit of the control port, called the *ready* bit, indicates whether the device is ready for communication. The second bit of the control port, called the *interrupt enable* bit, indicates whether the device should generate an interrupt when the device becomes ready. The lower byte of the keyboard data port contains the last typed character. The lower byte of the display data port can be filled with the next character to be output when the device is ready. All other bits have undefined values. The addresses of the ports can be found in Table 1.

You can observe (display) and influence (keyboard) the behavior of the external devices in the MARS simulator via `Tools->Keyboard and Display MMIO Simulator`. Within the *Keyboard and Display Simulator*, press `Connect to MIPS` before you start executing your program.

Table 1: The memory addresses for communication with external devices.

	Control port	Data port
Keyboard	`0xffff0000`	`0xffff0004`
Display	`0xffff0008`	`0xffff000c`

As soon as you press a (virtual) key, the *ready* bit is set to 1. If you load the current character from the data port into a processor register, the *ready* bit is set back to 0. As soon as you load a character into the data port of the (virtual) display, the *ready* bit is set to 0. After a (configurable) delay, the character appears on the display and the *ready* bit is set back to 1. If an external device is *ready* and the corresponding *interrupt enable* bit is set, an interrupt request is generated. For this, the corresponding interrupt pending bit in the *cause* register is set to 1, *IP[2]* for the keyboard and *IP[3]* for the display. The interrupt pending bit becomes 0 as soon as the corresponding external device is no longer *ready*, or the *interrupt enable* bit is set to 0.

Problem 2.1: System Calls — 5+5=10 points

Implement the following system calls:

- system call with the number 1, which prints the binary number passed in `$a0` in two's complement representation as a signed decimal number (without leading zeros) on the external display, and
- system call with the number 11, which prints the ASCII character passed in `$a0` to the external display.

You can proceed analogously to the section *Example Workflow Exception Handling*. For other exceptions and other system calls, your exception handler shall do nothing and simply return to the user program.

*Note: For the output, you may use a wait loop (*busy wait*) which waits for the* ready *bit of the display to be set. In particular, you do not need to implement an interrupt handler for the display.*

Problem 2.2: Process Switch — 16 Points

In this problem, a periodic process switch between two non-cooperative programs is to be implemented. The two given programs shall be executed alternately for about 100 cycles each.

First, consider how to periodically hand over control of the processor to the operating system. In the second step, consider how to execute the actual process switch. For this, consider the corresponding process control blocks. You will need to add fields to the given control blocks. Do not delete any of the predefined fields. Make sure that all fields in the process control block are always set correctly. Finally, combine both parts and implement a corresponding exception handler in MIPS assembler.

Ensure that both programs are executed alternately by simulation with the MARS simulator. In particular, do *not* modify the code of the two given processes.

Note: It is sufficient to save those registers in the control block that are actually used by at least one of our user programs.

Problem 2.3: Memory-Mapped Input/Output — 6 Points+6 Bonus Points

In this problem, communication with external input/output devices is to be realized, more precisely with a keyboard and a display. Write a program which reads characters from the keyboard of the *Keyboard and Display Simulator* and outputs them in the same order on the display there. If the keyboard provides inputs faster than the display can process outputs, you may discard additional inputs.

Note: In the `Keyboard and Display Simulator`*, select a high value for the Transmitter Delay Length to mimic the actual slowness of I/O devices.*

1. Implement the functionality with *polling*. With *polling*, you constantly poll the status of your devices in a loop and act accordingly. In particular, no interrupts are used.
2. Bonus: Implement the input/output functionality described above using *interrupts*. Unlike *polling*, here you let yourself be notified of status changes of your external devices via interrupts. So, work will only be performed in case of a status change. You will need a buffer between input and output. Use a buffer with 16 bytes. One byte of it may always remain unused.