SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

Systemarchitektur SS 2021 – Projekt 2

Systemprogrammierung und Ausnahmebehandlung

Abgabemodalitäten

Das Projekt beginnt am 2. Juli 2021 mit Herausgabe dieser Beschreibung. Wir empfehlen, das Projekt *so bald wie möglich* zu beginnen. Nutzen Sie die jeweiligen Übungsgruppen, um etwaige Verständnisprobleme und/oder Probleme mit dem MARS-Simulator auszuräumen.

Sie dürfen das Projekt in *Gruppen von zwei bis drei Personen* bearbeiten. Wenn Sie eine Gruppe gebildet haben, erstellen Sie bitte auf Ihrer persönlichen Statusseite im CMS bis zum

Montag, dem 5. Juli 2021, 23:59 Uhr

ein entsprechendes Team.

Eine Person pro Gruppe muss die Ergebnisse Ihres Projekts bis zum

Freitag, dem 16. Juli 2021, 23:59 Uhr

in unserem CMS-System hochladen. Insgesamt sind 32 Punkte (plus 6 Zusatzpunkte) zu erreichen. Zu spät abgegebene Projekte werden mit **0 Punkten** bewertet. Verwenden Sie zur Abgabe eine ZIP-Datei oder ein gzip-komprimiertes tar-Archiv (d.h. `*.tar.gz`). Das Archiv soll unmittelbar alle *bearbeiteten* Assemblerdateien enthalten. Verwenden Sie die von uns zur Verfügung gestellten Gerüst-Dateien, welche Sie in unserem CMS[1] finden. Falls Sie eine Aufgabe nicht bearbeiten, reichen Sie die zugehörige Assemblerdatei auch nicht ein.

Kommentieren Sie Ihren Code sinnvoll, d.h. beschreiben Sie innerhalb von Kommentaren die logischen Einzelschritte Ihrer Ausnahmebehandlung, nicht jedoch jede einzelne Instruktion. Orientieren Sie sich an den Kommentaren der Gerüst-Dateien. Unverständlicher und nicht kommentierter Code wird zu **Punktabzug** führen.

Fügen Sie dem Archiv außerdem eine Datei `beitraege.txt` hinzu, die kurz beschreibt, welchen Beitrag die einzelnen Teammitglieder zur Implementierung geleistet haben. Wir behalten uns vor, das Projekt für einzelne Mitglieder mit 0 Punkten zu bewerten, wenn diese keinen signifikanten Beitrag geleistet haben.

*Hinweise: Eine Zusammenarbeit mit Personen, die nicht zur eigenen Gruppe gehören, ist **nicht** erlaubt. Wir werden alle Abgaben auf Plagiate prüfen. Als Plagiate gelten auch Abgaben, die durch Modifizieren eines anderen Projekts entstanden sind, wie z.B. durch Ändern von Variablennamen. Plagiierte Abgaben werden wir mit **0 Punkten** bewerten und als Täuschungsversuch an den Prüfungsausschuss melden. Dies kann ggf. zur Exmatrikulation führen!*

*Falls Sie in der Vergangenheit schon einmal an der Systemarchitektur-Vorlesung teilgenommen haben und Ihre damalige Abgabe als Grundlage für das aktuelle Projekt verwenden möchten: Dies ist nur zulässig für Teile, die Sie selbst implementiert haben; Code, der von anderen Mitgliedern des damaligen Teams geschrieben wurde, darf **nicht** wiederverwendet werden. Fügen Sie in diesem Fall zu der Abgabe eine Datei `vorjahr.txt` hinzu, die genau beschreibt, welche Teile übernommen worden sind. Beachten Sie jedoch, dass das aktuelle Projekt nicht identisch zu den Projekten aus den Vorjahren ist. Abgaben, die zur Aufgabenstellung eines alten Projekts gehören, werden wir mit **0 Punkten** bewerten.*

[1] `https://cms.sic.saarland/sysarch21/materials/`

Werkzeuge und Dokumentation

Um Ihre Assemblerprogramme zu testen, verwenden Sie die von uns angepasste Variante des MARS-Simulators, die insbesondere Unterstützung für Timerinterrupts bietet. Diese Version entspricht der Version des ersten Projekts, **unterscheidet** sich aber vom Original sowie der Version für Programmierung 2.

Die wichtigsten Informationen, die Sie für die Bearbeitung des Projekts benötigen, stellen wir im folgenden Abschnitt vor. Darüber hinaus bietet Teil 3 der offiziellen MIPS-Dokumentation zusätzliche Informationen[2] zur Ausnahme- und Unterbrechungsbehandlung. Unter [3] finden Sie, wie im ersten Projekt, eine Beschreibung aller MIPS-Instruktionen.

Hintergrundinformationen

Im folgenden finden Sie für das Projekt benötigte Hintergrundinformationen. In sämtlichen Aufgaben werden Sie jeweils eine Routine zur Ausnahmebehandlung schreiben. Solche Routinen sind üblicherweise Bestandteil des Betriebssystems.

Eine Ausnahme (engl. *exception*) kann aus mehreren Gründen eintreten, z.B. durch ein Systemaufruf oder durch die Ausführung einer ungültigen Instruktion. Eine Ausnahme kann aber auch unabhängig vom ausgeführten Programm aufgrund von externen Geräten (Tastatur, Bildschirm) eintreten, z.B. wenn ein Zeichen auf der Tastatur eingegeben wurde. Solche Ausnahmen durch externe Geräte nennt man Unterbrechung (engl. *interrupt*).

Die Behandlung von Ausnahmen benötigt eine gewisse Hardwareunterstützung. In MIPS-Prozessoren stellt der sogenannte *System Coprozessor* diese Funktionalität zur Ausnahmebehandlung bereit. Ein Coprozessor hat allgemein eine eigene Menge von Registern und implementiert eigene zusätzliche Befehle. Er teilt sich aber auch Logik mit dem Prozessor, z.B. zum Fetchen von Instruktionen; daher *Co*prozessor.

Sobald eine Ausnahme eintritt, wechselt der Prozessor vom *user mode* zum sogenannten *kernel mode*. Im User Modus hat ein Programm nur eingeschränkten Zugriff auf die Maschine, z.B. kann es weder mit dem System Coprozessor kommunizieren noch bestimmte Instruktionen verwenden noch auf externe Geräte zugreifen. Im Kernel Modus hingegen hat ein Programm uneingeschränkten Zugriff auf die Maschine inklusive des System Coprozessors und externer Geräte.

Register des System Coprozessors

Die Register des Coprozessors 0 (CP0), des System Coprozessors, nennt man auch *special-purpose Register* im Gegensatz zu den 32 meist frei verwendbaren *general-purpose Register* des Hauptprozessors. Für unsere Zwecke der Ausnahmebehandlung betrachten wir die folgende Teilmenge der 32-Bit breiten special-purpose Register: *epc*, *status*, *cause* zur allgemeinen Ausnahmebehandlung und *count*, *compare* zur Programmierung von zeitgesteuerten, periodischen Interrupts (Timer interrupts).

Exception Program Counter *epc*

Kommt es zu einem Ausnahmefall, wird der aktuelle Programmzähler vom System Coprocessor in *epc* gesichert. Dieses Register enthält somit die Adresse der Instruktion, an welcher die Ausführung nach der Ausnahmebehandlung (meistens) fortgesetzt wird. Der *epc* ist CP0-Register 14 und kann gelesen und geschrieben werden.
Hinweis: Der MARS verwendet (standardmäßig) keine Delay Slots, was die Implementierung vereinfacht. Passagen in der MIPS-Dokumentation zu Delay Slots können Sie also ignorieren.

Status Register *status*

Das Status Register beinhaltet Informationen zum Zustand des System Coprozessors und ermöglicht eine gewisse Konfiguration der Ausnahmefunktionalität. Der *status* ist CP0-Register 12 und kann gelesen und geschrieben werden. Für uns sind lediglich die folgenden Bits relevant.

[2] `https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02`
[3] `https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06`

Bit 0: *Interrupt Enable IE* gibt an, ob Ausnahmen global erlaubt sind (1). Ist das Bit auf 0 gesetzt, werden keine Ausnahmen generiert.
Bit 1: *Exception Level EXL* gibt an, ob momentan eine Ausnahme behandelt wird. Während der Ausnahmebehandlung befindet sich der Prozessor im Kernel Mode.
Bit 4 – Bit 3: *KSU* gibt an in welchem Mode sich das System außerhalb der Ausnahmebehandlung befindet. Wir nehmen hier vereinfachend an, dass dies immer der *user mode* (Bits 10) ist.
Bit 15 – Bit 10: *Interrupt Mask IM[7] - IM[2]* gibt an, ob auf Unterbrechungsanfragen von externen Geräten reagiert werden soll.

Cause Register *cause*

Das Cause Register beinhaltet zum Zeitpunkt einer Ausnahme den jeweiligen Grund für diese Ausnahme. Der *cause* ist CP0-Register 13 und kann *nur gelesen* werden. Für uns sind lediglich die folgenden Bits relevant.

Bit 6 – Bit 2: *ExcCode* gibt den Grund für die Ausnahme an. Eine Tabelle mit allen möglichen Belegungen finden Sie in Tabelle 9.53 auf Seite 212 der MIPS-Dokumentation Teil 3. Wichtig ist die Belegung 00000 — sie signalisiert einen Interrupt.
Bit 15 – Bit 10: *Interrupt Pending IP[7] – IP[2]* gibt an, ob eine aktuelle Unterbrechungsanfrage eines externen Gerätes besteht. Die Positionen entsprechen den jeweiligen Positionen von *IM[7] – IM[2]* im Status Register.

Timer Interrupts

Unser MIPS-System unterstützt programmierbare, zeitgestützte Unterbrechungen. Der Timer wird über die Count und Compare Register gesteuert. Der Timer Interrupt ist assoziiert mit IP[7] und IM[7]. Das Bit IP[7] wird vom Timer auf 1 gesetzt sobald ein gewisser Zeitstempel erreicht ist, d.h. Compare und Count den gleichen Wert haben. Das Pending Bit IP[7] wird vom Timer wieder auf 0 gesetzt, sobald das Compare- oder Count-Register neu beschrieben wird.

Count Register *count* Das Count Register, CP0 Register 9, wird in jedem Zyklus um eins inkrementiert, es sei denn es wird mittels einer Instruktion beschrieben.

Compare Register *compare* Das Compare Register, CP0 Register 11, enthält einen Zeitstempel. Erreicht das Count Register diesen Zeitstempel, so wird ein Timer-Interrupt signalisiert. Das Register ist sowohl les- als auch schreibbar.

Kommunikation mit dem Coprozessor

Der Code der Ausnahmebehandlung selbst wird auf dem Hauptprozessor ausgeführt. Um innerhalb dieser Behandlungsroutine Zugriff auf die Register des Coprozessors zu haben, gibt es zwei Instruktionen zur Kommunikation. Mithilfe von `mfc0` kann man den Wert eines special-purpose Registers in ein general-purpose Register kopieren und umgekehrt mit `mtc0` einen Wert in ein special-purpose Register kopieren.

Zusätzliche Befehle

Die Instruktion `eret` kehrt von der Routine zur Ausnahmebehandlung zum eigentlichen Programm zurück. Mithilfe der `syscall` Instruktion können Sie vom Nutzerprogramm aus einen Systemaufruf vom Betriebssystem anfordern. Es ist eine Konvention des sogenannten *Application Binary Interfaces*, die Nummer des geforderten Systemaufrufs in `$v0` und ein mögliches Argument in `$a0` zu übergeben. Für weitere Informationen konsultieren Sie bitte die angegebene MIPS-Dokumentation.

Konvention: Reservierte General-Purpose Register

Obwohl alle 32 Register eines MIPS-Prozessors (mit Ausnahme des Registers 0) frei verwendbar sind, gibt es bestimmte Konventionen bezüglich ihrer Verwendung. Ein bereits bekanntes Beispiel ist Register 31 (genannt

`$ra`), welches die Rücksprungadresse bei Funktionsaufrufen verwaltet. Eine für dieses Projekt wichtige Konvention ist es, Register 26 (`$k0`) und 27 (`$k1`) für das Betriebssystem — z.B. unsere Ausnahmebehandlung — zu reservieren. Somit dürfen Register 26 und 27 *nicht* von gewöhnlichen Programmen verwendet werden.

Beispiel Ablauf Ausnahmebehandlung

Wir möchten folgendes Programm ausführen:

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

Wie sieht die Abarbeitung auf einer MIPS-Maschine aus? `//`-Kommentare beziehen sich auf die Hardware, `#` beziehen sich auf die Routine zur Ausnahmebehandlung.

```
...
108: syscall
// Generiere Ausnahme:
// - Setze epc auf 108
// - Setze kernel mode status[EXL] = 1
// - Setze cause[ExcCode] auf 8
// - Setze pc auf den Anfang der Routine zur Ausnahmebehandlung 0x80000180
# Führe Routine zur Ausnahmebehandlung aus
# - Sichere Register, die innerhalb dieser Routine verwendet werden
# - Untersuche ExcCode
# - Reagiere auf Exception/Interrupt
# - Addiere (ggfls.) 4 auf epc, hier: um Systemaufruf nicht zu wiederholen
# - Führe eret aus
// Kehre zu Programm zurück
// - Setze pc auf epc
// - Setze user mode status[EXL] = 0
10c: add $a0, $a0, 1
...
```

Ausnahmegenerierung

Wir haben bereits den generellen Ablauf einer Ausnahmebehandlung gesehen, aber nicht wann eine Ausnahme generiert wird. Hierbei gibt es zwei Möglichkeiten. Erstens kann eine Ausnahme durch das ausführende Programm generiert werden, z.B. beabsichtigt durch einen Systemaufruf oder unbeabsichtigt durch eine Division durch 0. Zweitens kann eine Ausnahme durch die Unterbrechungsanfrage eines externen Gerätes generiert werden. Der Coprozessor generiert eine Ausnahme sobald eine nicht-maskierte Unterbrechungsanfrage vorliegt (*cause[15:10]* & *status[15:10]* $\neq$ 000000), interrupts global aktiviert sind (*status[IE]* = 1), und der Prozessor sich nicht innerhalb einer Ausnahmebehandlung befindet (*status[EXL]* = 0).

Memory-Mapped I/O Devices

Hier liefern wir etwas Hintergrundinformationen zu den externen Ein-/Ausgabegeräten. Ein bestimmter Teil des Adressraums (von `0x00000000` bis `0xffffffff`) ist nicht mit dem Hauptspeicher verbunden, sondern mit sogenannten *Ports* von externen Geräten. Sie können mit einem externen Gerät also durch Lesen und Schreiben auf die Adressen dieser Ports kommunizieren. Daher wird dieses Vorgehen auch *memory-mapped input/output* genannt.

Der MARS-Simulator stellt zwei (virtuelle) externe Geräte bereit: eine Tastatur und einen Bildschirm. Jedes dieser Geräte hat zwei Ports: einen Kontrollport und einen Datenport. Das unterste Bit des Kontrollports, genannt

ready-Bit, gibt an, ob das Gerät bereit zur Kommunikation ist. Das zweite Bit des Kontrollports, genannt *interrupt enable*-Bit, gibt an, ob das Gerät eine Unterbrechung generieren soll, sobald das Gerät bereit wird. Das untere Byte des Tastatur-Datenports hält das letzte getippte Zeichen bereit. Das untere Byte des Bildschirm-Datenports kann mit dem nächsten auszugebenden Zeichen befüllt werden, wenn das Gerät bereit ist. Alle anderen Bits haben einen undefinierten Wert. Die Adressen der jeweiligen Ports finden Sie in Tabelle 2.

Das Verhalten der externen Geräte können Sie im MARS-Simulator unter `Tools->Keyboard and Display MMIO Simulator` beobachten (Bildschirm) und beeinflussen (Tastatur). Drücken Sie innerhalb des *Keyboard and Display Simulator* auf `Connect to MIPS` bevor Sie Ihre Programmausführung starten.

Sobald Sie eine (virtuelle) Taste drücken, wird das *ready*-Bit auf 1 gesetzt. Laden Sie das aktuelle Zeichen vom Datenport in ein Prozessorregister, wird das *ready*-Bit zurück auf 0 gesetzt. Sobald Sie ein Zeichen in den Datenport des (virtuellen) Bildschirms laden, wird das *ready*-Bit auf 0 gesetzt. Nach einer (konfigurierbaren) Verzögerung erscheint das Zeichen auf dem Bildschirm und das *ready*-Bit wird wieder auf 1 gesetzt. Ist ein externes Gerät *ready* und das jeweilige *interrupt enable*-Bit gesetzt, wird eine Unterbrechungsanfrage gestellt. Hierzu wird das jeweilige interrupt-pending Bit im *cause*-Register auf 1 gesetzt, *IP[2]* für die Tastatur und *IP[3]* für den Bildschirm. Das interrupt-pending Bit wird 0, sobald das entsprechende externe Gerät nicht mehr *ready* ist oder das *interrupt enable*-Bit auf 0 gesetzt wird.

Aufgabe 2.1: Systemaufrufe — 5+5=10 Punkte

Implementieren Sie die folgenden Systemaufrufe:

- Systemaufruf mit Nummer 11, welcher das in `$a0` übergebene ASCII-Zeichen auf dem externen Bildschirm ausgibt, und
- Systemaufruf mit Nummer 4, welcher den nullterminierten String an der in `$a0` übergebenen Adresse auf dem externen Bildschirm ausgibt.

Sie können analog zum Abschnitt *Beispiel Ablauf Ausnahmebehandlung* vorgehen. Für andere Ausnahmen sowie andere Systemaufrufe soll Ihre Ausnahmebehandlung nichts tun und einfach zum Userprogramm zurückkehren.

*Hinweis: Zur Ausgabe dürfen Sie eine Warteschleife (*busy wait*) verwenden, welche darauf wartet, dass das* ready*-Bit des Bildschirms gesetzt wird. Insbesondere müssen Sie keine Interrupt-Behandlung für den Bildschirm programmieren.*

Aufgabe 2.2: Prozesswechsel — 16 Punkte

In dieser Aufgabe soll ein periodischer Prozesswechsel zwischen zwei nicht-kooperativen Programmen implementiert werden. Die beiden vorgegebenen Programme sollen abwechselnd für jeweils circa 100 Takte ausgeführt werden.

Überlegen Sie sich zunächst, wie Sie periodisch die Kontrolle des Prozessors an das Betriebssystem übergeben können. Im zweiten Schritt überlegen Sie sich, wie Sie den eigentlichen Prozesswechsel ausführen können. Betrachten Sie hierzu die jeweiligen Prozesskontrollblöcke. Sie müssen den vorgegebenen Kontrollblöcken noch Felder hinzufügen. Löschen Sie keine der vorgegebenen Felder. Sorgen Sie dafür, dass alle Felder im Prozesskontrollblock jeweils korrekt gesetzt sind. Fügen Sie schließlich beide Teile zusammen und implementieren Sie eine entsprechende Ausnahmebehandlung in MIPS-Assembler.

Stellen Sie durch Simulation mit dem MARS-Simulator sicher, dass beide Programme abwechselnd ausgeführt werden. Insbesondere dürfen Sie den Code der beiden vorgegebenen Prozesse *nicht* ändern.

Hinweis: Es ist ausreichend, die Register im Kontrollblock zu sichern, die tatsächlich von mindestens einem unserer User-Programme verwendet werden.

Tabelle 1: Die Speicheradressen zur Kommunikation mit externen Geräten.

	Kontrollport	Datenport
Tastatur	`0xffff0000`	`0xffff0004`
Bildschirm	`0xffff0008`	`0xffff000c`

Aufgabe 2.3: Memory-Mapped Ein-/Ausgabe 6 Punkte+6 Bonuspunkte

In dieser Aufgabe soll die Kommunikation mit externen Ein-/Ausgabegeräten realisiert werden, genauer mit einer Tastatur und einem Bildschirm. Schreiben Sie ein Programm, welches Zeichen von der Tastatur des *Keyboard and Display Simulator* einliest und in der eingegebenen Reihenfolge auf dem dortigen Bildschirm wieder ausgibt. Wenn die Tastatur Eingaben schneller liefert als der Bildschirm Ausgaben verarbeiten kann, dürfen Sie zusätzliche Eingaben verwerfen.

Hinweis: Wählen Sie im `Keyboard and Display Simulator` *einen hohen Wert für die Transmitter Delay Length um die tatsächliche Langsamkeit von I/O-Geräten nachzuempfinden.*

1. Implementieren Sie die Funktionalität mit *Polling*. Beim *Polling* fragen Sie in einer Schleife ständig den Status Ihrer Geräte ab (engl. *to poll*) und handeln dementsprechend. Insbesondere kommen keine Interrupts zum Einsatz.

2. Bonus: Implementieren Sie die oben beschriebene Ein-/Ausgabe-Funktionalität mithilfe von *Interrupts*. Im Gegensatz zum *Polling* lassen Sie sich hier von Statusänderungen Ihrer externen Geräte via Interrupts benachrichtigen. Es wird also nur im Falle einer Statusänderung Arbeit verrichtet. Sie werden einen Puffer zwischen Ein- und Ausgabe benötigen. Verwenden Sie einen 16 Bytes großen Puffer. Ein Byte davon darf immer unbenutzt bleiben.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Prof. Dr. Jan Reineke
Dr. Andreas Abel
Lucas Biehl

System Architecture SS 2021 – Project 2

System Programming and Exception Handling

Submission Modalities

The project starts on July 2, 2021 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 5, 2021, 23:59.

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

Friday, July 16, 2021, 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[4]. 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**.*

[4] `https://cms.sic.saarland/sysarch21/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[5] on exception and interrupt handling. See [6] 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.

[5] `https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02`
[6] `https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06`

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    andi $a0 $a0 0  # clean $a0
// - 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 2.

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 2: 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 11, which prints the ASCII character passed in `$a0` to the external display, and
- system call with the number 4, which prints the null-terminated string at the address passed in `$a0` on 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.