UNIVERSITÄT DES SAARLANDES

Prof. Dr. Jan Reineke

# System Architecture SS 2024 – Project II

# System Programming and Exception Handling

## Project Modalities

The project starts on July 05, 2024 with the release of this description.

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

Wednesday, July 10, 2024, 23:59.

One person per group must submit the group's solution of the project to our CMS system by

Sunday, July 21, 2024, 23:59.

**Follow the instructions in the next section to generate your submission.** A total of 32 points (plus 6 bonus points) can be achieved. Projects submitted late will be graded with 0 points.

## Tools and Setup

The skeleton files for the project are available at the following public repository:

`https://gitlab.cs.uni-saarland.de/reineke/sysarch-project-two`.

Simply clone the repository to get started. We do recommend using a version control system like `git` to manage your project within your team. For instance, you can use the department's GitLab server to host your repository.

As in the first project, you will need *Java* and *Scala*, including *sbt* ("Simple Build Tool"). To install all of these, follow the instructions at `www.scala-lang.org/download/`. This should leave you with a working Scala environment, including *sbt*, and if not already installed, *Java*. In addition, you will need Docker to run the RISC-V simulator. You can install Docker from `https://docs.docker.com/get-docker/`.

Call `docker compose up -d`. This will download and start the required Docker containers. You can access the RISC-V simulator at `http://localhost:9000`. You can shut down the Docker containers with `docker compose down`. In case of any problems you can see the logs with `docker compose logs`. Once you are ready to submit your solution, run `sbt createSubmission` and upload the generated zip file to our CMS.

If the provided docker images do not work on your setup, you can build them yourself by executing `docker compose up -build -d`.

| Command | Description |
|---|---|
| `docker compose up -d` | Start the Docker containers. |
| `docker compose down` | Shut down the Docker containers. |
| `docker compose logs` | Show the logs of the Docker containers. |
| `sbt createSubmission` | Create a submission file (inside folder `submissions`). |

## Working on the Project

The tasks for the project are given at the end of this document. For each task, you will find skeleton files in the `programs` folder and a test configuration in the `configs` folder.

You need to only modify[1] the assembly files in the `programs` folder.

To test your work, start the Docker containers and open `http://localhost:9000` in your browser for the RISC-V simulator. There, you can upload a configuration file using the upload button. If there are any syntax errors in the configuration file or the assembly code, the simulator will show a corresponding error message. If the configuration file is correct, the simulation will start and you can control the simulation with the buttons at the top.

The provided assembler supports RV32I, RV32M, and Zicsr instructions. Additionally a few useful pseudo-instructions are supported, such as `la` to load the address of a code-label, `li` to load a 32-bit immediate value, and `mv` to move a value between registers. `beqz` or `bnez` are also supported as conditional branches. `j` and `ret` can be used to call and return from a subroutine. `csrr` and `csrw` are supported to read and write control and status registers. You cannot use data segments in your assembly code. Please specify initial memory values in the configuration file.

Enabling the virtual keyboard allows you to generate memory-mapped inputs to the processor, which will be useful to test your solution of the second task.

## Collaboration Policy

You may work on the project in groups of up to three people. Briefly describe how each team member contributed to the project in the file `CONTRIBUTIONS.md`. There, also include a brief statement whether you have used any AI support for the project, and if so, which tools were used. We may ask individual team members to explain their contributions in more detail orally. We also expect that all team members have a good understanding of the entire project. We may grade the project with 0 points for individual members if they have not made a significant contribution or do not have a reasonable understanding of the project as a whole.

**The work you hand in as a team must be your own.** Copying another group's work or allowing your work to be copied by others is a serious academic offense and will be treated as such.

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.

### Examples of permitted collaboration:

- Talk to other groups about how to complete the project **at a high level**.
- Ask a tutor for help if you are confused or stuck.

### Examples of collaboration that is not permitted:

- Help another group debug their code by bringing up your own solution and comparing the two.
- Using a friend's code from a previous year as a starting point and then modifying it.
- Copying any portion of another group's code even if you modify it.
- Sharing portions of your code with another group.

## Questions and Problems

If you encounter any issues while working on the project that you cannot solve in your group, we will be happy to help you in the forum or during the office hours. In general, however, we expect that you have already done some work on your own (own tests, debugging outputs, online research, etc.) to solve the issue. Non-specific questions like "Does the code compile?", "Is this correct?", or "What needs to be done in problem X.Y?" will not be answered.

[1] You may also change or create new configuration files, but make sure that your submitted solution works with the original configuration files. The configuration files are not included in the submission.

# Background Information

In the following, we will describe background information required for the project. For all tasks, you will write exception handling routines. 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 expiring timers or 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 hardware support. In RISC-V processors, the *machine mode* provides exception handling functionality. As soon as an exception occurs, the processor switches from *user mode* to *machine mode* and jumps to a predefined address in the exception handling routine. In contrast to user mode, machine mode is a privileged mode that has full access to the machine's resources, including the control and status registers (CSRs) and external devices.

## System Calls in RISC-V

System calls are a way for user programs to request services from the operating system. In RISC-V, system calls are implemented using the `ecall` instruction. It is a convention of the so-called *Application Binary Interface* (ABI) to pass the number of the requested system call in register `a7` and possible arguments in registers `a0`, ..., `a5`. The `mret` instruction is used to return to user mode from an exception.

## Control and Status Registers in RISC-V

The RISC-V ISA contains 32 general-purpose registers (GPRs). These registers are defined in the unprivileged specification and are sometimes referred to as integer registers. The RISC-V privileged specification (see `https://riscv.org/technical/specifications/`) defines additional registers referred to as *Control and Status Registers* (CSRs). While GPRs are accessible at any privilege level, CSRs are defined at a specific privilege level and can only be accessed by that level and any levels of higher privilege. In this project, we will work with an instance of RISC-V that has two privilege levels: *user mode* and *machine mode*, and we will use CSRs that are only accessible in machine mode.

The privileged specification defines a common set of CSRs and address ranges that all CSRs must reside within. CSR addresses are 12 bits, encoded as an immediate in the instructions that are used to access them. As a consequence, up to $2^{12} = 4096$ CSRs could in principle be implemented.

We consider the following subset of the CSRs: *mepc*, *mtvec*, *mstatus*, *mcause*, *mscratch*, *mie*, *mip* for general exception handling. We will also consider the machine-level memory-mapped registers *mtimecmp* and *mtime* for programming timer interrupts.

### Machine Exception Program Counter *mepc*

The machine exception program counter *mepc* is CSR register 0x341 and can be read and written.

If an exception occurs, the current program counter is saved in the CSR *mepc* to allow the program to continue after the exception has been handled. To this end, the `mret` instruction is used to switch to user mode and jump to the address stored in the *mepc*. In some cases, the exception handler needs to modify the *mepc* before calling `mret`:

- After handling a system call, the *mepc* needs to be incremented by 4 to prevent the system call from being executed again.
- After the boot code, the *mepc* needs to be set to the address of the first user program to execute.
- To perform a process switch, the *mepc* needs to be set to the address of the next process to execute.

### Machine Trap-Vector Base Address *mtvec*

The machine trap-vector base address register *mtvec* is CSR register 0x305 and can be read and written. It contains the base address for the exception handling routine and the mode in which the *mtvec* register is used.

Exception handling is performed in one of two different modes:

- *Direct Mode*: The exception handler is located directly at the base address stored in the *mtvec* register.
- *Vectored Mode*: In this case, the address of the exception handler is determined by the base address *and* the exception code: The address is the base address in the *mtvec* register plus four times the exception code.

The mode is determined by the least significant bit of the *mtvec* register:

- If the two least significant bits are $00$, the *mtvec* register is in direct mode.
- If the two least significant bits are $01$, the *mtvec* register is in vectored mode.

In both cases, the remaining bits of the *mtvec* register determine the base address, where the two least significant bits are omitted and implicitly assumed to be $00$, which means that the base address is aligned to a 4-byte boundary.

### Machine Status Register *mstatus*

The machine status register contains information on the state of the core and it allows some configuration of the exception functionality. The machine status register is CSR register 0x300 and can be read and written. For us, only the following bits are relevant:

**Bit 12 – Bit 11:** *Machine Previous Privilege Mode (MPP)* indicates the privilege mode before the exception occurred. As we only have two privilege levels, these are either $00$ (user mode) or $11$ (machine mode).
**Bit 7:** *Machine Previous Interrupt Enable (MPIE)* indicates whether interrupts were enabled before the current exception occurred.
**Bit 3:** *Machine Interrupt Enable (MIE)* indicates whether interrupts are enabled.

### Machine Cause Register *mcause*

The machine cause register contains at the time of an exception the reason for this exception. The machine cause register is CSR register 0x342 and can be read and written. For us, only the following bits are relevant:

**Bit 31:** *Interrupt* indicates whether the exception was caused by an interrupt.
**Bits 30 – 0:** *Exception Code* contains the reason for the exception.

The exception code is used to distinguish between different types of exceptions. Common case for the exception code are system calls (code 8), timer interrupts (code 7), and external interrupts (code 11).

### Machine Scratch Register *mscratch*

The machine scratch register is a general-purpose register that can be used by the exception handling routine. The machine scratch register is CSR register 0x340 and can be read and written. It is used to store temporary values that are needed during exception handling. The register is not modified by the processor and can be used freely by the exception handling routine.

### Machine Interrupt Registers *mie* and *mip*

The machine interrupt enable register *mie* and the machine interrupt pending register *mip* are used to enable and handle interrupts. The machine interrupt enable register is CSR register 0x304 and can be read and written. The machine interrupt pending register is CSR register 0x344 and can be read and written.

For us, only the following bits are relevant:

**Bit 7 of *mie***, *Machine Timer Interrupt Enable (MTIE)* enables timer interrupts.
**Bit 11 of *mie***, *Machine External Interrupt Enable (MEIE)* enables external interrupts.
**Bit 7 of *mip***, *Machine Timer Interrupt Pending (MTIP)* indicates that a timer interrupt is pending.
**Bit 11 of *mip***, *Machine External Interrupt Pending (MEIP)* indicates that an external interrupt is pending.

## Control and Status Register Instructions

The Zicsr RISC-V ISA extension provides instructions to read and write CSRs.

The `csrrw` (atomic read/write CSR) instruction atomically swaps values in the CSRs and integer (i.e., general-purpose) registers. More precisely, `csrrw rd, csr, rs1` reads the value from the CSR `csr`, zero-extends

the value to 32 bits, writes the value from the source register `rs1` to the CSR, and stores the old CSR value in the destination register `rd`. If `rd` is `x0` the instruction does *not* read the CSR and does not cause any side effects that might occur on a CSR read.

The `csrrs` (atomic read/set CSR) instruction atomically reads, sets, and writes values in the CSRs and integer registers. More precisely, `csrrs rd, csr, rs1` reads the value from the CSR `csr`, zero-extends the value to 32 bits, sets the bits in the CSR that are set in the source register `rs1`, writes the new value to the CSR, and stores the old CSR value in the destination register `rd`.

Similarly, the `csrrc` (atomic read/clear CSR) instruction atomically reads, clears, and writes values in the CSRs and integer registers. More precisely, `csrrc rd, csr, rs1` reads the value from the CSR `csr`, zero-extends the value to 32 bits, clears the bits in the CSR that are set in the source register `rs1`, writes the new value to the CSR, and stores the old CSR value in the destination register `rd`. Any bit that is high in `rs1` will be cleared in the CSR.

For both `csrrs` and `csrrc`, if `rs1` is `x0`, then the instruction does not write to the CSR and does not cause any side effects that might occur on a CSR write.

The `csrrwi`, `csrrsi`, and `csrrci` variants are similar to `csrrw`, `csrrs`, and `csrrc` respectively, except they update the CSR using a 32-bit value obtained by zero-extending a 5-bit unsigned immediate.

## Example Workflow Exception Handling

We would like to execute the following program:

```
...
100: li a0, 1234
104: li a7, 1
108: ecall
10c: addi a0, a0, 1
...
```

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

```
...
108: ecall
// Generate exception:
// - Set mepc to 108
// - Switch to privilege mode 11 (machine mode)
// - Set mcause[Exception Code] to 8
// - Set pc to the start of the exception handling routine defined by mtvec
# Execute exception handling routine
# - Save registers that are used by this routine
# - Examine Exception Code
# - React to exception/interrupt
# - If interrupt: clear pending interrupt bit in mip
# - Add (if necessary) 4 to mepc, here: to not repeat system call
# - Execute mret
// Return to the program
// - Set pc to mepc
// - Switch to privilege mode 00 (user mode)
10c: addi a0, a0, 1
...
```

## Memory-Mapped Timer Registers and I/O Devices

### Timer Registers *mtimecmp* and *mtime*

Our RISC-V system supports programmable timer interrupts. The timer is controlled by the memory-mapped *mtimecmp* and *mtime* registers. Both registers are 64-bit wide and can be read and written.

These registers are not CSRs, but rather memory-mapped registers. Memory-mapped registers are registers that are accessible in the same way as memory, i.e., they can be read and written using load and store instructions to particular addresses that are reserved for these registers. Therefore, this approach is also called *memory-mapped I/O*.

**Machine Time Register *mtime*** The *mtime* register increments at a constant frequency unless it is written to by an instruction and it wraps around if the count overflows. In our implementation, the *mtime* register is incremented by one every cycle.

**Machine Time Compare Register *mtimecmp*** The *mtimecmp* register contains a timestamp. A machine timer interrupt becomes pending whenever the value of the *mtime* register is greater than or equal to the value of *mtimecmp*, treating both values as unsigned integers. The interrupt remains posted until *mtimecmp* becomes greater than *mtime* (typically as a result of writing *mtimecmp*).

**Note:** In our implementation you need to manually clear the interrupt pending bit in the *mip* register after handling an interrupt. The interrupt pending bit will not be cleared automatically by modifying the *mtimecmp* or the *mtime* register.

The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the *mie* register.

In RV32, both registers are split into two 32-bit halves, with the lower half at the lower address.

When updating the *mtimecmp* register to a new value in two steps, one needs to take care that the intermediate value does not accidentally trigger an interrupt. Here is sample code from the RISC-V privileged specification that avoids such a scenario:

```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1) # No smaller than old value.
sw a1, 4(t1) # No smaller than new value.
sw a0, 0(t1) # New value.
```

Using our assembler, you can refer to the base addresses of the two 64-bit memory-mapped registers by their names, e.g., `mtimecmp` and `mtime`. The upper halves of the registers can be accessed by adding 4 to the base address, e.g., `mtimecmp+4` and `mtime+4`, which are also directly accessible by their names `mtimecmph` and `mtimeh`.

### Keyboard and Display

In addition to the timer, our RISC-V system also provides two external memory-mapped input/output 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 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. Using our assembler you can refer to the addresses of the ports by their names, i.e., `keyboard_ready` and `keyboard_data` for the keyboard, and `display_ready` and `display_data` for the display.

You can observe (display) and influence (keyboard) the behavior of the external devices in the RISC-V simulator that we are providing.

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 delay, the character appears on the display and the *ready* bit is set back to 1. If an external device is *ready* , an external interrupt request is generated. For this, the corresponding interrupt pending bit in the *mcause* register is set to 1. However, as both devices share the same interrupt code, the *mcause* register does not distinguish between keyboard and display interrupts.

**Note:** As with timer interrupts, in our implementation you need to manually clear the interrupt pending bit in the *mip* register after handling an interrupt. The interrupt pending bit will not be cleared automatically by writing to the display or reading from the keyboard.

# Your Tasks

For each of the following tasks, you will find skeleton files in the provided repository. Your task is to complete these files to implement the described functionality. The skeleton files also include user code that interacts with your implementation. For testing purposes, we may replace this user code with our own test code.

## Task 1: System Calls and I/O (4+4=8 Points)

Implement the following system calls:

1. System call number 11, which prints the ASCII character passed in register `a0` to the external display.
2. System call number 4, which prints the null-terminated string at the address passed in register `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. You can use memory addresses `0x0` to `0xff` for storing saved registers while handling system calls.

*Note: For the output, you may use a wait loop (*busy wait*) that 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.*

## Task 2: Memory-Mapped I/O (6 Points+6 Bonus Points)

In this task, your code shall interact with both external devices, i.e., keyboard and display. Write a program that reads characters from the keyboard and outputs them in the same order on the display. If you wish, you may reuse code from the previous task.

*Note: You can figure the delay of the display (in cycles) in the JSON configuration file.*

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. If the keyboard provides inputs faster than the display can process outputs, you may discard additional inputs.
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. Use a buffer of 16 bytes to store inputs received from the keyboard that have not yet been sent to the display. One byte of the buffer may always remain unused. Between handling interrupts your code should execute the given user program (which does not generate exceptions itself). You can use memory addresses `0x0` to `0xfff` for your buffer and for storing saved registers of the user program while handling interrupts.

## Task 3: Process Switch and Round-Robin Scheduling (10 Points)

In this task, you shall implement a periodic process switch between two non-cooperative processes. The two given programs shall be executed alternately for about 300 cycles each.

First, consider how to periodically hand over control of the processor to the operating system. Then consider how to perform the actual process switch. To this end, you will need to save and restore the registers of the currently running process in its process control block. Make sure that all fields in the process control block are always set correctly. You may use memory addresses `0x0` to `0xfff` for the process control blocks of the two processes as you see fit. Combine both parts and implement a corresponding exception handler in RISC-V assembler.

Ensure that the two given programs are executed alternately using our simulator.

*Note: Save all registers in the process control block so that your process switch works for an arbitrary pair of user programs.*

## Task 4: Cloning Processes (8 Points)

In the previous task, you implemented a process switch between two non-cooperative processes.

In this task, you shall implement two system calls that allow to create new processes and to determine the process ID of the currently running process:

- System call number 220, which creates a new process by cloning the calling process. We call the calling process the "parent" process and the new process the "child" process. The child process shall execute the same code as the parent process continuing execution at the instruction following the system call. The system call shall return the process ID of the new child process in register `a0` to the parent process and it shall return 0 in the same register to the child process. Execution should continue with the parent process. Process IDs shall be unique and start at 1.
- System call number 172, which returns the process ID of the currently running process in register `a0`.

Each process will need its own process control block. You may use memory addresses `0x0` to `0xfff` for the process control blocks of the processes as you see fit. Your system should support the creation of 8 processes. System calls to create more than 8 processes shall fail, indicated by returning $-1$ in register `a0`.

Implement round-robin scheduling among all processes with a time slice of 300 cycles.