Objectives

In the scope of this practical work, you will learn about the exception vector, system calls, and weak symbols.

Remarks

You need a Gecko board with the most up-to-date bitstream, necessary accessories, and a computer with the toolchain installed. Make sure that you have access to following documents (available on Moodle):

1. OpenRISC 1000 Architecture Manual
2. The GNU Assembler Manual (version 2.41)
3. The GNU Linker Manual (version 2.41)

You might also need to use some online resources to answer some of the questions in this practical work.
Make sure that you have downloaded and extracted the project files. We refer to file paths relative to the project directory.

Makefiles

The project template uses makefiles to automate the build process:

```
# cd in to the directory with the makefile

# before executing make, make sure that the toolchain is in the PATH
export PATH=/opt/or1k_toolchain/bin/:$PATH

# build the mem file ready to be uploaded
make mem

# clean the built files
make clean

# build and keep the intermediate assembler files for inspection
make mem CFLAGS=-save-temps

# build and also print the linker map
make mem CFLAGS=-save-temps LDFLAGS=-Wl,--print-map
```

Built files can be found in `built-release/` directory. You are encouraged to read and understand the makefile content.

> You can integrate makefile-based projects with Visual Studio Code.
> Modify the first line of the makefile (`PROJECT = template`), which describes the name of the project, and the directory name to reuse the template.

Better `printf`, some standard library primitives, and global variables

We updated the project template with a better open-source `printf` implementation (see https://github.com/mpaland/printf). We also provide some useful standard library primitives, such as `putchar` and `getchar`.

```
void *ptr = 0x8000;

// print a pointer
printf("ptr = 0x%08p\n", ptr);

// print a floating point number
// note: enable support by modifying ‘support/printf_config.h‘.
printf("f = %f\n", 3.14f);
```

Check out the header files in the `support/include` to learn more about the provided functionality.
We now support global variables with the `__global` modifier:

```
#include <defs.h> // provides __global
__global const char* cache_assoc[] = { "dm", "2-way", "4-way" };
```

1 Exception Vector

An *exception vector* is an array of function pointers. In case of a special condition (such as an unaligned access, or an interrupt) the processor jumps to the relevant *exception handler* as pointed by the corresponding element of the array.

Task 1.1. Read the relevant sections of the OpenRISC 1000 Architecture Manual to learn more about (1) exception classes and (2) exception processing. What is the role of EPC (a special-purpose CPU register)?

Task 1.2. Open `support/src/crt0.s` in the text editor. Where in this file is the exception vector *defined*? What is the purpose of the `.global _vectors` directive? You can refer to the GNU Assembler Manual.

Task 1.3. What does the `_exception_handler` do in `crt0.s`? Explain the `l.rfe` instruction by referring to the OpenRISC 1000 Architecture Manual.

Task 1.4. Open `support/include/exception.h` in the text editor. This header file allows interfacing with the exception vector from the C code. Notice that the vector is *declared* as:

```c
typedef void (*exception_handler_t)(void); /* function pointer */
extern exception_handler_t _vectors[EXCEPTION_COUNT];
```

What is the purpose of the `extern` keyword? How does `extern` relate to the `.global _vectors` directive?

Task 1.5. Consider the following code snippet:

```c
int *addr = (int *) 0x100; // choose whatever address you want
int data;
asm volatile("l.lwz %[d], 0(%[s])" : [d]"=r"(data) : [s]"r"(addr));
```

What are the possible exceptions that this code snippet can generate by setting the value of the `addr` variable? (Hint: you should have at least two.) Demonstrate on the Gecko board. What purpose does the `volatile` keyword serve?

Here are some useful online resources for GCC inline assembly:

1. https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/how-to-use-inline-assembly-language-in-c-code.html
2. http://www.ethernut.de/en/documents/arm-inline-asm.html

Task 1.6. In the `src/main.c`, define your own exception handler for one of the exceptions that you generated. The name of the exception handler should start with `my_`. Modify the exception vector to enable the new exception handler. You can use the enumeration with symbols `EXCEPTION_*` to index the correct exception vector entry. Do the modification only in `main.c`, never modify `crt0.s` or `exception.c`. Do not forget to `#include <exception.h>`.

2 System Calls

A *system call* (usually abbreviated as syscall) enables programs to request a service from the operating system. For example, on POSIX-compliant operating systems, syscalls like `read` and `write` allow for reading from/writing to file descriptors or sockets. Each platform provides various ways to invoke system calls. OpenRISC ISA provides a single system call instruction (`l.sys`, see the architecture documentation) that raises a system call exception. An operating system provides the syscall functionality by impleming the system call exception handler. In this part of the practical work, you will implement a simple system call exception handler.

Task 2.1. Make a syscall using the `SYSCALL(n)` macro defined in `exception.h`. `n` is the syscall number chiefly used for identifying the syscall type. Choose `n` as `0xAA` for debugging purposes.

Task 2.2. Write your custom handler for the system call exception (whose name should start with `my_`), then modify the exception vector to enable it. The handler should print the value of EPC. `support/include/spr.h` defines helper macros to read to/write from the special purpose registers (SPRs). You can include it by `#include <spr.h>`. Use `printf("0x%08x", ...)"` for printing the instructions.

Task 2.3. Print the values of the instruction pointed by `EPC`, and the previous instruction (`EPC - 4`). Which instruction do you think is the system call instruction that has just triggered the exception?

Task 2.4. Decode the system call instruction to extract the syscall number. Modify the syscall handler to print the syscall number as well. Clear the VGA screen for syscall `0xE0`. VGA functions are defined in `support/include/vga.h`, which you can include by `#include <vga.h>`.

Task 2.5. Open the `support/src/exception.c`. What is the purpose of `__weak` modifier in front of the exception handler definitions? `__weak` is defined as in `support/include/defs.h`:

```c
/**
 * @brief Defines a weak symbol.
 *
 */
#define __weak __attribute__((weak))
```

Task 2.6. Now, rename your exception handler to `system_call_handler` (the same name as in the `exception.c`). Do not modify the exception vector. Should your code still work given that there are now two symbols with the same name? Why or why not? Test it.