CS-473:
System programming
for
Systems on Chip

Practical work 3

Profiling and memory distance

Version:
1.0

Contents

1 Introduction

1.1 Prerequisites

On moodle you find the file `mandelbrot_fxpt.zip`; it contains the complete source code of the fixed-point version of the Mandelbrot set. Download this file, and unzip it in the directory `programms`, where you also have the floating point version of the last PW.
Make sure that you have access to following documents (available on Moodle):

- OpenRISC 1000 Architecture Manual
- The GNU Assembler Manual (version 2.41)
- 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.

1.2 Preparations

In the theory session we have seen the performance counters that are present in the or1300. To use these counters there are support functions. These functions can be found in:
`programms/support/include/perf.h`
Please familiarize yourself with these functions.

1.3 Memories

In the theory session we have seen that SDRAM's have some particularities, like the burst mode and the CPU-Memory distance. In this practical work we are going to see which influences this can have on your program. Please note that the VGA-controller *fetches* each line in a burst of `nrOfPixelsPerLine/2` 32-bit words.

2 Exercises

2.1 Prerequisites

Familiarize yourself with the fixed-point version of the Mandelbrot set. Compile the program by:

```
cd programms/fractal_fxpt_sol
make mem1300
```

Make sure that you do not have any errors. Then, download the `fractal_fxpt.cmem` file to your virtual prototype and see the result.

Of course you can also use your own fixed-point version of the Mandelbrot set. Please make sure that you insert the commands required for profiling and memory distance in your `main_fxpt.c`. Also make sure that the flushing of your data-cache (`dcache_flush()`) is the last command in your `main()` function.

2.2 Profiling

In the provided fixed point version of the mandelbrot algorithm, already two profiling counters are defined as:

```
perf_set_mask(PERF_COUNTER_0, PERF_STALL_CYCLES_MASK |
                              PERF_ICACHE_NOP_INSERTION_MASK);
perf_set_mask(PERF_COUNTER_1, PERF_BUS_IDLE_MASK);
```

The profiling counter 0 counts the stall cycles. To understand why also the `PERF_ICACHE_NOP_INSERTION_MASK` is used, the behavior of the fetch-stage needs to be known. In case the fetch-stage has not yet a new instruction, it will not stall the CPU, but it will insert `l.nop` instructions. This is counted with the given mask.

The profiling is started, respectively stopped with the functions:

```
perf_start();
perf_stop();
```

After stopping the profiling, please wait at least 5 CPU-cycles, as the profiling module is pipe-lined.

The results of the profiling counters can than be shown with:

```
perf_print_time(PERF_COUNTER_0, "Stall cycles    ");
perf_print_time(PERF_COUNTER_1, "Bus idle cycles ");
perf_print_time(PERF_COUNTER_RUNTIME, "Runtime cycles  ");
```

There exists also another support function that prints the results in decimal format:

```
perf_print_cycles(PERF_COUNTER_RUNTIME, "Runtime cycles  ");
```

In this exercise we are going to add two profiling counters.

Add and print out the number of:

- Instruction cache fetches.
- Instruction cache misses.

What can you observe?

2.3 Memory distance

Your virtual prototype has the ability to *emulate* a speed difference between the CPU and the external SDRAM. This emulation consists of *slowing down* the SDRAM-accesses. The macro used for this purpose is `perf_memdist_set( val )` that is defined in `perf.h`. The value-range of `val` is 0 (the SDRAM is running at the same frequency as the CPU) up to 63 (one SDRAM clock cycle equals to 63 CPU-clock cycles).

In this exercise we are going to *play* with the memory distance. Up to this moment we executed the mandelbrot algorithm with a memory distance value 0.

Execute the mandelbrot algorithm with the memory-distance values 5, 25 and 63.
What can you observe, and how can you explain it?

2.4 Exception vectors

The *exception vectors* are 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.
For this and the following exercises, you can use the template `programms/exceptions`.

Tasks:

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)?
2. Open `support/src/crt0.s` in the text editor. Where in this file are the exception vectors *defined*? What is the purpose of the `.global _vectors` directive? You can refer to the GNU Assembler Manual.
3. What does the `_exception_handler` do in `crt0.s`? Explain the `l.rfe` instruction by referring to the OpenRISC 1000 Architecture Manual.
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 each 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?

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

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 vectors 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>`.

2.5 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 implementing the system call exception handler. In this part of the practical work, you will implement a simple system call exception handler.

Tasks:

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.
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.
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?
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>`.

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:

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

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.