

General Instructions

- You can download all required configuration, make and source files, that you will need for the following exercises, from Moodle.
- Each group should submit a report, written in French or English, in the PDF file format via Moodle.
- Each student has to be member of a group, where groups may vary in size from 3 to 4 students. Details on the groups are available on Moodle.

1 Setup

For the following exercises you will need to download an archive from Moodle containing configuration, as well as make and source files for the exercises. After extracting the archive, you should find the following files in the `PR2` directory:

```
brandner@kairon:~/PR2> find . -type f
./insertion-sort.c
./Makefile
```

- **insertion-sort.c**
C source code of the program that should be analyzed in the exercises. The source code is compiled, using the RISC-V C compiler, into a binary program (`insertion-sort.bin`) suitable for simulation by the `Ripes` simulator.
- **Makefile**
Makefile to compile `insertion-sort.c`.

To get started, simply launch a terminal, go into the directory containing the `Makefile`, and type `make` (or `make all`). This should build a RISC-V binary automatically. After completion you should find the following files:

```
brandner@kairon:~/PR2> find . -type f
./insertion-sort.c
./insertion-sort.elf
./Makefile
```

- **bin/insertion-sort.elf**
This is the binary produced by the RISC-V compiler in the ELF format, which can be processed by `Ripes`.

You can remove the generated file using the usual `make clean` command.

1.1 Using Ripes Interactively

Ripes provides a graphical user interface, which visualizes the execution of the program (similar to pipeline drawings and pipeline diagrams that we saw in the lecture), allows single-stepping (i.e., simulating the program cycle-by-cycle), and allows to inspect the register and memory values.

In order to start Ripes, simply open a terminal, and type the following command: `Ripes`. This should open a Ripes window ... you are good to go.

You may install Ripes on your personal computer by downloading a binary release from the following website: <https://github.com/mortbopet/Ripes/releases/tag/v2.2.3>.

2 RISC-V Tool Chain

Aims: Get familiar with the RISC-V tools and the Ripes interface.

Just follow the instructions below for the various exercises. Answer the questions in your report and illustrate them, e.g., using screen shots – whenever this appears helpful to you.

2.1 RISC-V Tools

- Compile `insertion-sort.c` using `make` and check which commands are executed.

Hint: if `make` tells you `make: Nothing to be done for 'all'`, either use `make -B` or `make clean all` – try both and explain the difference.

- Explain what the compiler options `-nostdlib` and `-nostartfiles` do.

Hint: Use `man gcc`.

- Look at the assembly code of the resulting ELF binary using the following command:
`riscv64-linux-gnu-objdump -d bin/insertion-sort.elf`

The output of this command consists of three columns the addresses of the various instructions (on the left), the instructions' binary representation (as hexadecimal numbers), and the human readable assembly code.

Hint: Using `riscv64-linux-gnu-objdump -d bin/insertion-sort.elf | less` might be even more convenient.

- Look at the assembly code of the ELF binary using the command:
`riscv64-linux-gnu-objdump -S bin/insertion-sort.elf`, which in addition to the assembly code also shows the original C code. Find the assembly code corresponding to the first `for`-loop in function `main`. At which address range is the code? Briefly explain what the instructions of that loop do.

- Look at the symbols defined in the ELF binary (functions and global variables) using the command:








`riscv64-linux-gnu-objdump -t -j.data -j.text bin/insertion-sort.elf`

The output of this command consists of several columns (not all of them are interesting for us). The leftmost column shows the address of the symbols, while the rightmost shows the symbols' names.


Find the address of symbol `input`. This symbol represents the variable `input` from the C code.

- Find an instructions that makes use of the address of `input` (maybe a memory load/store, or some instruction loading the address into a register). Explain the respective instruction/instruction sequences.
- Search on the internet what position independent code (PIC) is. Explain in which situations PIC-code might be useful. Explain why PIC-code is essential to many modern security mechanisms. Which role does the instruction `auipc` play on the RISC-V architecture in the context of PIC-code?
- Is the compiled binary code position-independent (PIC)? Justify your answer. If yes, indicate and explain instructions sequences that show PIC code. If no, explain the difference between the actual binary code in the ELF file and what one would expect from PIC code on RISC-V.

2.2 Ripes

- Read the documentation of the Ripes simulator:
<https://github.com/mortbopet/Ripes/wiki/Ripes-Introduction>
- Launch Ripes and load the ELF program `insertion-sort.elf` into the simulator via the menu *File · Load Program* – then follow the instructions of the dialog window.
- Go to the *memory* tab ( on the left) and show that the content of the `input` array within the simulator matches that of the source/ELF file. You can select an address or a section under the drop-down-list "Go to section". From there on you may scroll through the memory content using your mouse wheel (touch-pad gestures may also work).
- The values of the array `input` are copied by the first `for`-loop of `main` into another array `buf`, which is stored on the stack. Go to the binary-code tab () find the store instruction within this loop and set a breakpoint by clicking in the side-bar on the left of the instruction. Run the program up to that breakpoint and determine the address of the array `buf` by examining the register values. The simulator can run either in an interactive mode (buttons /||, slow) or quickly execute up to the next breakpoint/the program's completion (button , faster).
- Open again the *memory* tab and show that the values of array `buf` get actually sorted. You can even watch memory content while the program executes using the interactive mode (see above).
- Open the cache tab () and play with different cache options (number of lines & ways, block size). You can zoom into the content of the cache by pressing the *Ctrl* key and at the same time turning your mouse wheel (touch-pad gestures may also work).
- Finally go to the processor tab () and summarize the main characteristics when executing the program to completion (number of cycles & executed instructions, CPI, IPC). Click on button on the right of the simulator controls () to see a simplified pipeline diagram.




Note: The diagram only shows the last couple of instructions executed by the processor. You can change this number via the menu *Edit · Settings* under the list item *Environment* on the left, and the option *Max. pipeline diagram cycles*.

- The simulator allows to use different processor models using the CPU button ( on the top next to the simulator control buttons). Select another processor model (32-bit) and summarize and compare the characteristics with the numbers that you have obtained before.

3 Simple Pipelining

Aims: Deepen the understanding of the operation of a simple, but realistic, pipelined processor.



You now saw pretty much everything `Ripes` can do. In the next exercises you will use the various functions to explore the operation of the processor pipeline.

- Select the *5-stage processor*, using the CPU button () and simulate the insertion sort ELF binary for the following questions.
- Execute the program for a few cycles (100 cycles or so). Can you tell from the pipeline diagram ( on the Processor tab ) whether the simulated processor supports forwarding? You can also have a look at the processor's data path shown on the same tab.
Explain the capabilities of the processor and illustrate them through examples. Can you identify special cases for certain kinds of instructions?
- How does the simulator know how that the program has completed? Look at the pipeline diagram/the executed instructions and look at the simulator's documentation (via the menu *Help*). Research the involved instruction(s) and explain what they are usually intended for.

4 Branches and Multiple-Issue

Aims: Understand the impact of branches and more complex pipelines.


The following exercises will focus on issues related to branches.

- Select the *5-stage processor*, using the CPU button () , and simulate the insertion sort ELF binary for the following questions. Again, execute the program for a couple of cycles.
 - When you look at the pipeline diagram, you will see quite a few instructions that do not complete their execution. Find examples in the pipeline diagram illustrating this situation. Explain why the respective instruction did not finish its execution. What happens and why?
 - Have a close look at the way branches and jumps are handled. Can you determine whether the processor has a branch predictor? If yes, explain and illustrate how it works? If not, explain through examples and illustrations why not.
 - You will see that the processor achieves an CPI of 1.38 when executing the program to completion. What would the optimal CPI be for such a simple pipelined processor? Explain why the processor did not achieve this optimal value and compare this to the analysis of the CPI of the processor discussed in the lecture.
 - Select the *6-stage dual issue processor*, using the CPU button () , and simulate the insertion sort ELF binary for the following questions.
 - Analyze the pipeline's behavior as you did before for the simple 5-stage processor. Try to identify interesting situations, analyze, explain, and illustrate them in your report.
- Hint:** Think about data hazards between instructions.
- The performance of this dual-issue processor is better, with an CPI of 1.26. Still it is a bit disappointing. Discuss what the optimal CPI for this processor would be and explain why it is not achieved for this particular program.

5 Caches

Aims: Understand the performance of caches.

The following exercises will focus on data/instruction caching. However, the pipeline simulator is not really integrated with the cache simulator, i.e., you cannot observe the impact of stalling due to memory accesses. Also, complex pipelines may distort the accuracy of the cache simulation ...

- Analyze the program's C and assembly code (using `riscv64-linux-gnu-objdump` seen before) and try to determine, on paper, the number of distinct memory words accessed by the program, distinguishing between instruction fetches and data accesses. Discuss in detail how and when the program accesses certain ranges of data/instruction addresses.
- Select the *Single-cycle processor*, using the CPU button () and simulate the insertion sort ELF binary for the following questions.
- Using the cache parameters (number of lines & ways and block size) propose cache configurations for both, the data and instruction cache, that allow you to validate your analysis regarding the number of memory accesses performed. Configure the cache as a *write-through* cache with *write-allocate*. First describe and explain the cache configurations in the report. Then validate them by running the simulator. If you find any discrepancy between your analysis and the results, explain your mistake and try to correct it.
- Now describe cache configurations for the data/instruction caches respectively that results in the minimum number of cache misses, while also minimizing the cache size (shown in the *Statistics* view as *Size (bits)*). Again describe the idea first in the report, then validate your idea in the simulator, and explain any mistakes that you encountered in that process.

Is the minimal numbers of cache misses the same for the data and instruction cache? Explain why/why not.

Hints: You have to get the number of cache misses down to single digit numbers. The same goes for the total size of the caches in bits, it should not be larger than a single-digit factor of the size of the accessed data/instructions.

- Given the results from the last two questions, try to define a metric that allows you to find the best data cache configuration in terms of efficiency, i.e., that provides the best balance among the design criteria you have seen in the lecture. Describe, before running any simulations, what you think might give you this best data cache configuration. Then, run experiments in order to validate your intuition.
- Discuss how the data access patterns that you observe during the execution of the program impact the cache policy. Do the data access patterns match nicely with the available cache policies? Why/why not?

Hint: Think of concepts that you have heard in the lecture, such as locality, working set size, and the trade-off in terms of cache design parameters.

- You can of course observe the evolution of the cache contents during the simulation. Using a cache configuration of your choice, illustrate the different scenarios that explain

a cache miss: (a) compulsory misses, (b) conflict misses, or (c) capacity misses. In a similar vain, illustrate through examples how the parameter *Write allocate* compares with a configuration using *No write allocate*. Provide a detailed discussion of these examples/illustrations.