Computerarchitectuur - Lab 2 Skeleton Code Overview

This document gives a brief overview of the skeleton code for assignment 2. We describe possible development environments, list the source code files and their contents and functions, and conclude with a summary of command line options to the program.

Development Environment

You should be able to work on this assignment using a Linux, Windows, or macOS machine. You can also use the University Linux environment (remotely) through ssh. In case you work on this assignment on your own computer, please do make sure everything compiles and works as expected on the University Linux environment before submitting.

Linux

A modern C++-compiler is required (at least g++ 8.3 or Clang 8.x) as well as Python 3.5 or higher. Any recent Linux distribution should suffice. You can compile using the provided Makefile by simply using the `make` command.

You can also work remotely on the University Linux workstations. Note that the workstations in the lab rooms run Ubuntu 22, whereas the 'huisuil' (remotelx) runs Ubuntu 20. This means that a code compiled on a lab room computer might not run on the 'huisuil'. In this case, simply recompile the project.

Windows

A Visual Studio project file is provided to compile a native Windows executable of the project. See the `Windows` subdirectory in the skeleton code. We do however strongly recommend to use Windows Subsystem for Linux (WSL). This will create a Linux environment on top of your Windows installation. Within this Linux environment you can use gcc and the Makefile as you would on Linux. More information on installing WSL can be found here:

`https://docs.microsoft.com/en-us/windows/wsl/install-win10`

Source Code Overview

The main parts of the processor are implemented in the following files and their corresponding headers. These will be your main focus when making changes.

`processor.cc`	The processor class, which consists of the pipeline, memory, as well as shared components that are used by multiple pipeline stages, such as the program counter.
`pipeline.cc`	The pipeline class. This class creates and initializes each of the pipeline stages. If a stage needs access to a shared component, a reference to that component is passed in its constructor here.
`stages.cc`	The implementation of each pipeline stage (for example `InstructionFetchStage`), as well as the declaration for the pipeline registers (for example `IF_IDRegisters`). The "wiring" of each stage is implemented here.

inst-decoder.cc	The instruction decoder. This is a component that splits an instruction into fields; it is used in the instruction decode stage, as well as in the instruction formatter.
inst-formatter.cc	The instruction formatter. This class takes an instruction and represents it as a string, like a disassembler would.
reg-file.h	The register file. This component is used in the decode stage, and contains the value of every register.
alu.cc	The arithmetic/logic unit (ALU). This component is used in the execute stage to perform the actual calculations (addition, logical OR, etc.).
mux.h	A multiplexer. Multiplexers are a hardware component used to choose between two different values, for example, whether the input of the ALU comes from the register file or from an immediate.

Memory and I/O devices

The simulated processor is connected to some memory, and some input and output devices. These devices are implemented in the following files.

memory-bus.cc	The memory bus. All I/O devices are represented as memory (memory mapped I/O). The bus sends memory operations (read or write) to the correct device.
memory-control.cc	The memory controller. This component connects the processor to the memory bus, and allows it to perform memory operations on instructions and data.
memory.cc	RAM, used as main memory.
serial.cc	The serial interface. This device is used to output text to the screen.
sys-status.cc	The system status device. A program can use this device to shut down the processor.

Loading and start-up

The following files relate to the loading and starting of programs. You won't need to make any changes here.

main.cc	The `main()` of the program; it handles command-line parsing and program initialization.
elf-file.cc	A loader for `.bin` files, which are ELF format executables.
config-file.cc	A loader for `.conf` files, which specify the contents of registers before and after a test is run.

Testing

The simulated processor has some useful features to test whether instructions work correctly.

`testing.cc`	This class starts tests and determines if they ran successfully.
`tests/`	This directory contains unit tests. Each test consists of a small program (source `.s`, assembled `.bin`) and a configuration file (`.conf`). The configuration file specifies the contents of registers before a test is run, and also the value they are expected to contain afterwards.
`testdata/`	This directory contains some tests for the instruction decoder. You can compare these with the results of your implementation.

Command Line Options Summary

A normal program is executed by simply providing the `.bin` file as argument:

```
$ ./rv64-emu ../lab2-test-programs/hello.bin
```

When the emulator exits, all current values of all registers are printed to the terminal.

You can manually execute a micro-program using the special 'unit test mode' of the emulator. This is triggered using the option `-t`:

```
$ ./rv64-emu -t tests/add.conf
```

Other relevant options when running programs are:

- `-d` will dump each fetched and decoded instruction to the terminal (for which you need to implement the instruction formatter).
- `-p` enables pipelining, which means that all stages will be executed at the same time instead of one after the other.

To aid in testing your instruction decoder and formatter, the `-x` and `-X` options are present. `-x` is used to test a single instruction, `-X` for a file of instructions. The file can be either a compiled `.bin` executable, or a `.txt` file with a list of hexadecimal instructions.

```
$ ./rv64-emu -x 0x12345678
$ ./rv64-emu -X tests/add.bin
$ ./rv64-emu -X testdata/decode-testfile.txt
```

A Python program `test_instructions.py` is available which will automatically run all micro-programs found in the `tests/` subdirectory. This is the default mode when executed without command line arguments. The following command line arguments can be specified:

- `-v` will give verbose output when running the micro-programs.
- `-f` will stop on the first failed test.
- `-p` will enable pipelining on the emulator, which means that each micro-program will be executed in pipelined mode.