Computer Architecture Fall 2022
Assignment 2: ISA emulation

Deadline: Friday, November 11, 2022

In this second assignment you will develop an emulator that is capable of executing simple OpenRISC programs. The OpenRISC architecture has been chosen because of its simplicity. The main learning objective of this assignment is to acquire a deep understanding of how processors execute instructions, the challenges of pipelining, and how the different categories of instructions are implemented.

We will limit ourselves to the execution of instructions. We will not concern ourselves with the detailed simulation of the characteristics of DDR memory nor will we model and simulate a cache subsystem. Within this assignment we will assume that a memory load/store operation can be performed in a single clock cycle. Of course, these things can be simulated in detail if the emulator were to be further extended.

You do not have to start from scratch. From BrightSpace a skeleton, or rather a starting point, can be downloaded[1]. A number of things have already been implemented within this starting point: loading programs in ELF format, a simplified emulation of a memory bus and memory, register file and simple "devices" to output characters and to the halt the system.

During the development of the emulator it is very important to test your code. The starting point provides functionality to test the decoding of individual instructions. You will also have to write micro programs (unit tests) to test the execution of individual instructions. To help you get started, the starting point contains a test harness that will be elaborated on below. The goal is to work towards the capability to run larger programs. A collection of test programs can be obtained from BrightSpace, which increase in complexity: `basic.bin`, `hellonods.bin`, `hello.bin`, `comp.bin` `hello4.bin`, `comp4.bin`, and `brainfuck.bin` (an interpreter of the well-known programming language). These test programs will also be used to determine the correctness score (the achieved level) of your program when grading the assignment.

Background Information

For general theory on Instruction Set Architectures we refer to Appendix A of the textbook (Hennessy and Patterson). In Appendix C the "classic RISC" pipeline is described, which consists of five steps to execute an instruction: Instruction Fetch, Instruction Decode, Execute, Memory, Write Back. In this assignment, we will process instructions according to these five steps. Appendix C.3 describes the implementation of a pipeline for a simple RISC-V processor, which can be adapted to OpenRISC. More detailed treatment can also be found in Chapter 4 of *Computer Organization and Design* also by Hennessy and Patterson.

Detailed information on the OpenRISC instruction set can be found in the architecture reference manual: `https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.4-rev0.pdf`. Note that in this assignment we will consider *32-bit* OpenRISC (ORBIS32), not 64-bit. Particularly useful is Chapter 5, which describes the instructions in detail, and a table of all instructions, including their instruction formats, that can be found in Chapter 18 (page 362). On BrightSpace an OpenRISC cheatsheet can be found, which clarifies/summarizes some of the instruction formats that are used (the ISA manual is not very clear in this regard).

Another aspect of OpenRISC that is important to be aware of is that the OpenRISC architecture is *big endian*, whereas Intel x86 is little endian. The skeleton code in the starting point has been modified to account for this (by converting all data read from the memories to little endian) and therefore you probably do not have to specifically deal with this difference.

[1] The files are also available on the University workstations: `/vol/share/groups/liacs/scratch/ca2022/files/`.

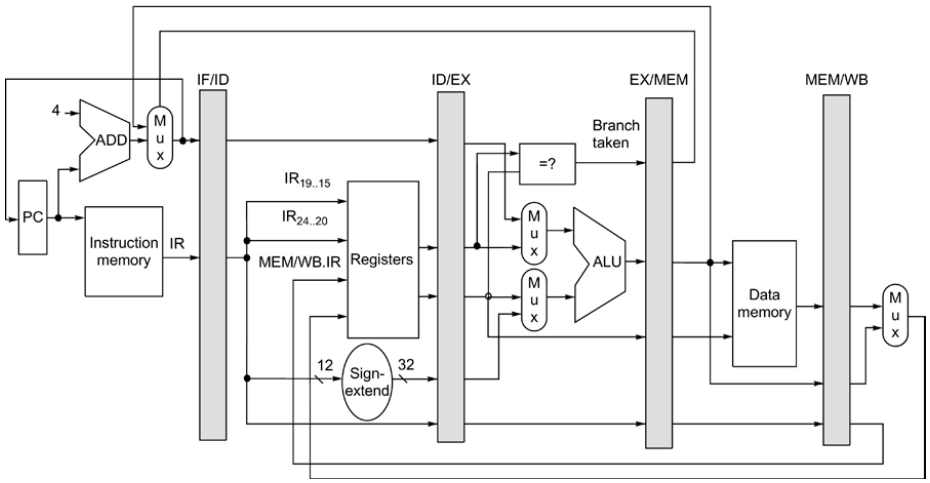

Figure 1: Possible data path for a simple RISC-V processor, which can be adapted to OpenRISC. **Source:** Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 6th edition. Figure C.19.

The Big Picture

Essentially what this assignment is about is to implement the data path and necessary control for a simple processor in a C++ code base. Many of the classes that will be implemented are similar to hardware components. Using getter and setter methods these components will be connected with each other.

Consider the data path in Figure 1 as discussed by Hennessy and Patterson. You will find several of these components in the C++ skeleton that is provided to you, such as the register file, pipeline registers and instruction and data memories. Not all of the components have been fully implemented. Because we will start with a non-pipelined design that is later extended to a pipelined design, we will use the pipeline registers in all cases. The data path consists of 5 stages. Each clock cycle, one of the stages is executed. This execution is done in two steps: propagate and clock pulse. During the propagate step the (source) pipeline register may be read and inputs of the different components are set. In fact, the inputs from the pipeline register propagate through the various components. In the clock pulse step the destination pipeline register is written. Also in components that contain state (such as the register file) a write is carried out.

So, the overall rule is: *only during propagate pipeline registers can be read; any components that contain state (pipeline registers, register file, data memory, PC, etc.) can only be written during clock pulse.*

In the source files `stages.cc` and `stages.h` you can find the different stages and pipeline registers stubbed out. They are still mostly empty! In this file you will be writing the necessary code to interconnect the components. For example, for the execute stage you will need to read fields from the `id_ex` pipeline register (`struct ID_EXRegister`) and configure the inputs of the ALU during the propagate step. In the clock pulse step of EX you need to set fields in the `ex_m` pipeline register (`struct EX_MRegister`) with the output value of the ALU and any other data that needs to be passed along for the future stages.

To understand how we model components, let us consider the register file as an example. An implementation of this component is provided to you (contrary to many other components). Figure 2 shows the C++ methods on the left and a schematic of the component on the right. The `RS1` and `RS2` inputs can be set right after instruction decoding in the propagate phase of the ID stage. In the clock pulse phase the outputs of the register file can be read and stored in a pipeline register. The `RD` and `writeEnable` signals are to be set from the propagate phase of the writeback stage. During the clock pulse stage of the writeback stage the `clockPulse` method is called to

```
void setRS1(const RegNumber);
void setRS2(const RegNumber);
void setRD(const RegNumber);
void setWriteData(const RegValue);
void setWriteEnable(bool);

RegValue getReadData1();
RegValue getReadData2();

void clockPulse();
```

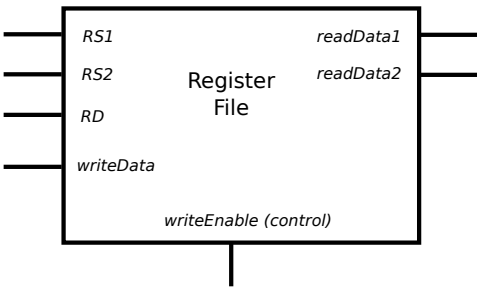

Figure 2: Left: C++ methods of the register file component. Right: schematic of the register file component.

trigger the actual write to the register file. Compare this with the lines drawn in the data path of Figure 1!

As can be seen in the data path there are several multiplexers (mux) and other components that need to be controlled. The multiplexers need to be told which of the inputs to relay to the output. The ALU needs an input that indicates the operation to be carried out. These are not data inputs (contrary to instruction word, register numbers, ALU output), but rather control inputs. So, next to the data path we need control signals, sometimes referred to as the control of the processor. Note that in Figure 1 the control lines are *not* shown. How to determine the correct values of the different control signals? Well, this is all encoded into the instruction. When decoding the instruction, we need to determine the control signals such that exactly this instruction is being executed. In case of OpenRISC, we can determine the control signals from the combination of opcode and function codes.

For the instruction decoding logic this means that we want to have an initial stage which splits the instruction into the different fields: A (RS1), B (RS2), D (RD), opcode, function code and the immediate. We suggest to implement this in the class `InstructionDecoder`. Some of these fields are passed directly to components (such as passing RS1 to the register file). And some of these fields also need to be passed to the pipeline register for use in a later step (for instance RD).

To implement the control signals, we suggest to implement a `ControlSignals` class. This will form a separate component. The instruction decode stage should set the opcode and function code as inputs on this component. After that, the control signals can be obtained using various getters. For example, you would want getters for determining the ALU operation (which is to be set as input on the ALU in EX), setting the selectors on the multiplexers in front of the A and B inputs of the ALU, and so on. You also want to pass the instance of the `ControlSignals` class to later stages through the pipeline registers. Figure 3 depicts the interplay between the initial instruction decoder and the control signals component.

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.

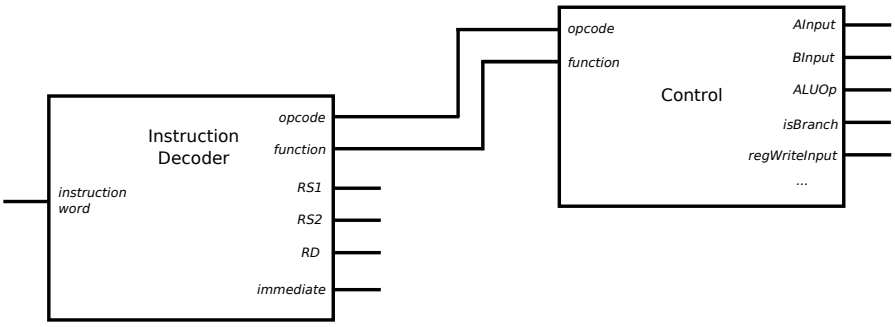

Figure 3: Interconnection between instruction decoder and control signals component.

You can also work remotely on the University Linux workstations (Ubuntu 18). To be able to successfully compile the project, you need to use the g++ 8 compiler (`g++-8`) instead of the system default. In this case use the following `make` command to compile the project:

```
make CXX=g++-8
```

Or, edit the `Makefile` and change the `CXX` variable at the top of the file.

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`

Skeleton Code

After successfully compiling the skeleton code, try running the emulator with a test program:

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

You will probably notice that the emulator gets stuck in an infinite loop. This is actually the expected result! The given skeleton code is not functional and many essential parts are missing. It is up to you to implement these. We will first briefly describe how the code base is organized and give hints on how to get started.

Structure

The core of the code base can be found in the files `stages.h` and `stages.cc`. Within these files the five stages of instruction execution are to be implemented. The classes are already stubbed out. In the file `pipeline.cc` these stages are constructed and the methods `Pipeline::propagate()` and `Pipeline::clockPulse()` will run the stages in sequence. The `propagate` and `clockPulse` methods are called by `Processor::run`. This run method is in fact the main loop of the program.

At the bottom of the definition of `class Processor` in `processor.h` you can find member variables that hold the pipeline and a number of shared components (such as the register file). When a stage needs access to a shared component a reference is passed to the stage upon construction as can be seen in the constructors of the `Pipeline` and `Processor` classes.

Start With The Instruction Decoder

We recommend to start this assignment by working on the instruction decoder for OpenRISC instructions. The task of an instruction decoder is easily described: given an *instruction word*, isolate the different fields from the instructions. The decoder is a separate class, `InstructionDecoder`, which can be found in `inst-decoder.h` and `inst-decoder.cc`. In these files, you can find a number of methods that need to be implemented. More methods need to be added to this class, for instance to decode immediate values and to obtain opcodes and function codes. When writing your decoder, avoid the use of "magic numbers" in the code and instead use enums and constants where desirable. You probably want to create enums for opcodes and function codes. Ensure that when a malformed or illegal instruction is given (for instance a non-existent or non-implemented opcode) the exception `IllegalInstruction` is thrown.

To test the instruction decoder, we need the capability to print the decoded instructions – so the state of the class `InstructionDecoder` – to the terminal. A stub for a method to do so can be found in `inst-formatter.cc`. The skeleton code provides a way to test just the instruction decoder and the formatter without implementing any of the pipeline stages. Essentially, this boils down to a simple disassembler. The following command decodes a single instruction specified on the command line:

```
./rv64-emu -x 0x15000000
```

you can also create an ASCII (text) file containing instructions in hexadecimal format and specify this to the program:

```
./rv64-emu -X myfile.txt
```

The `-X` option also accepts ELF files, such as any of the `.bin` files. In this case, the text segment of the ELF file will be disassembled.

We strongly recommend to work on the decoder first. There will be a subdeadline (October 21), at which time we will release a list of instructions and reference output, such that you can test your instruction decoder.

Getting The First Instructions To Work: ALU Operations

Once we have some confidence in our instruction decoder, it is time to get the first instructions to work. The easiest instructions to get going are ALU operations (R-type) that simply operate on registers. An example is the `l.add` instruction and a unit test for this (`add.bin`) is already provided with the skeleton code. To run this program, use:

```
./rv64-emu -t ./tests/add.bin
```

you can also add the `-d` command-line option to print decoded instructions to the terminal using your instruction formatter. At first, the program will get stuck in an infinite loop. But once you have wired up all the stages of the pipeline correctly (in `stages.cc` and `stages.h`), the instructions will be executed. So, what needs to be done in the different stages?

- **IF** Fetch an instruction word from memory. This is done by sending the PC to the instruction memory *setAddress* line and setting the size to 4 bytes, as OpenRISC instructions are 32 bits in size. Subsequently, the instruction can be loaded by reading the value lines. The definition of the instruction memory can be found in `memory-control.h`. These steps need to be implemented in the instruction fetch stage. Also pass the instruction word to the next stage through the pipeline register.

- **ID** In the instruction decode stage, we are going to decode the instruction word that was placed in the pipeline register. As a first step, the instruction word needs to be set on the decoder using the appropriate method. Secondly, we can configure the register file in `propagate`, to be able to fetch the register values in `clockPulse` (second half of the clock cycle). To do so, we will call the different methods of the decoder from this instruction decode stage. To fetch register values we need to call the `getA` method to get the register number. This number is set on the register file component, see Figure 1. Additionally, control signals need to be determined to drive the ALU and multiplexers in later stages. We suggest that you create a control signals class, as described earlier. On this class, opcode and function code need to be set, as obtained from the instruction decoder. The control signals class should have several getter-methods that indicate how the different components of the data path must be configured. See Figure 3 for an example.
- **EX** After an instruction has been decoded, it can be executed. Most instructions need to perform an ALU operation. In order to do so, the control signals A, B and op (refer to Figure 1) need to be set. ALU operations are implemented in `alu.cc` and `alu.h`. Methods are present in the `ALU` class to configure the operands and to read the result. Left to be added are ALU operations to be performed. Use the `ALUop` enum. Your control signals class needs a method to translate opcode and function code to the corresponding `ALUop`. So, within the execute stage you will obtain the `ALUop` from the control signals class and set this value on the ALU. All values must be set on the ALU during the propagate stage. In the clock pulse stage, one calls the method `getResult` to obtain the result and write this result to the pipeline register to be used in the next stage.
- **M** For ALU operations nothing is done during this stage. We simply forward the result from *EX* to the next stage (*WB*).
- **WB** We configure the register file to write the result to the destination register. The register file is pulsed in `propagate` as an exception, to be able to write register values in the first half of the clock cycle (recall the lectures on pipelining).

Once the stages have been wired up and the add operation implemented in the ALU, your emulator should be capable of executing `add.bin`. What's next? Implement more ALU instructions and write unit tests, consider the instructions used by `basic.bin`. After that, it is time for a different instruction type.

Moving On To Memory Operations

To implement memory instructions, you need to implement the memory stage in `stages.cc`. The effective address should be computed by the ALU during the execute stage (so, your control signals class needs to generate the "add" `ALUop` when it encounters a load or store instruction). In the memory stage, you need to send the correct control signals to the data memory. Refer to the `DataMemory` class in `memory-control.h` for details. Additionally, you still need to implement some of the `DataMemory` methods in `memory-control.cc`.

Branch Delay Slots

A notable characteristic of the OpenRISC architecture is it that uses branch delay slots. We have discussed these delay slots in the lectures on pipelining. Essentially, a branch delay slot is positioned immediately after a branch instruction. The instruction in this delay slot is always executed, regardless of whether the branch is taken. Also jumps have branch delay slots. After executing the instruction in the branch delay slot, control flow moves to the jump or branch target.

You will note that this has been designed such that no branch penalty will be incurred if the branch target and condition are known by the end of the *ID* stage of the branch instruction. To facilitate this, OpenRISC uses separate set flag instructions to perform register comparison

and branch if flag instructions. This simplifies the branch instruction and the flag can easily be verified in the *ID* stage. We have added the flag as shared component to the `Processor` class. An additional adder can be added to the *ID* state just for the purpose of computing the branch target (otherwise this needs to be done during *EX*, which will incur another stall cycle). Consider all of this in your design.

The first few levels of the assignment have been designed such that you can ignore branch delay slots. In these cases, the delay slots will always contain `l.nop` instructions, so it does not matter whether the branch delay slot is executed. Starting with level 4 delay slots should be implemented correctly, first without pipelining. To accomplish this, you will have to implement special logic to control updates to the program counter (PC).

Towards Pipelined Execution

Within this assignment we will assume that each of the five steps requires one clock cycle. To execute a reg-reg ALU instruction all five steps must be performed, therefore this takes five clock cycles. For the first levels in this assignment we will assume that the emulator only works on a single instruction at a time, so non-pipelined execution. After implementing a basic set of instructions, the higher levels, starting from level 5, will require you to implement pipelining. In this case, you will run the programs with the `-p` option of the emulator. This option will cause the emulator to run all five stages at the same time, instead of calling propagate and clock pulse of the different stages one after the other. So, in effect, the emulator will work on five instructions at the same time.

We consider two pipelining modes:

1. *Pipelining with hazard detection.* For this step you will implement support for detecting data and control hazards and automatically inserting stalls when necessary. To do so, we recommend to implement a hazard detector class/component that gets access to all pipeline registers. The clock pulse phase of the different stages can access the hazard detector's getter methods to see whether a stall needs to be inserted (which essentially means placing a `l.nop` instruction in the destination pipeline register). For control hazards, no complicated logic should be required to get branches right, as long as the branch target and condition are known at the end of *ID* and the PC is adapted at the end of *ID*. This is because of the use of branch delay slots in the architecture. One thing to be aware of is the dependency on the flag register between set flag and branch instructions! It might be necessary to insert stalls, so hazard detection should also take the flag register into account. When implemented correctly, it should be possible to run all unmodified test programs with pipelining enabled.

2. *Pipelining with hazard detection and forwarding.* Now also forwarding is implemented. This should reduce the number of stall cycles compared to pipelining without forwarding. Also consider whether forwarding is required/possible for the flag register.

Testing

We strongly recommend to implement the instructions one by one, or in small groups of similar instructions. Start with the implementation of the decoding logic for the instruction, followed by formatting. You can test this separately using the `-x` and `-X` options as described above. You can add unit tests just for formatting to the `testdata/` directory (you can look at `testdata/decode-file-testfile.test` for an example) and run all these tests with the `test_output.py` script.

After that, implement the necessary logic in the ALU and whatever is needed in the different stages. Subsequently, this individual instruction can be tested. To do so, we recommend to write a micro program. It is possible to write micro programs that consist of just a single instruction. Look at `tests/add.s` and `tests/add.conf` as an example. In the `.conf`-file the pre- and post-conditions of the test are specified: so how the registers must be initialized and what values the

registers must contain after completion of the program. Construct the pre- and post-conditions in such a way that you can ensure that the tested instruction functions correctly. When you want to add tests for a new instruction, say `nop`, you need to add files `nop.s` and `nop.conf`.

To be able to compile the micro programs, you need to have access to the OpenRISC toolchain. This toolchain is available in the `ca2022` environment:

```
source /vol/share/groups/liacs/scratch/ca2022/ca2022.bashrc
```

With the environment enabled, the command `make nop.bin` in the `tests` subdirectory will generate the binary program. Subsequently, `make check` will execute all micro programs in the `tests` subdirectory. Alternatively, you can also run `./test_instructions.py -v` to get verbose output. The `-f` option will stop on the first failed test. And `-p` will run the test with pipelining enabled.

It is also possible to manually execute a micro program using the special unit test mode of the emulator:

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

A "normal" program can be executed as follows:

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

When the emulator exits, all current values of all registers are printed to the terminal. Using the option `-d` the instruction dump functionality can be enabled (for which you need to implement the instruction formatter).

Framebuffer Device

For level 8 two additional programs must be made to run on your emulator: `gfxtest.bin` and `mandel-gfx.bin`. These programs can be obtained from BrightSpace in the separate `gfx-test-programs` tarball. Both of these programs use the framebuffer device that is implemented in the simulator. This framebuffer device is not compiled by default. To be able to compile the simulator with framebuffer support, the SDL2 library must be present on your system. It has been installed in the `ca2022` environment (which must be enabled prior to compilation). To compile the simulator with framebuffer support, use the following make command:

```
make ENABLE_FRAMEBUFFER=1 CXX=g++-8
```

`gfxtest.bin` is the easier of the two programs to get working. It does use the framebuffer and only consists of integer instructions. `mandel-gfx.bin` on the other hand also uses a number of floating-point instructions. These need to be implemented in your emulator. Note that OpenRISC does *not* use a separate register file for floating-point data!

Submission and Grading

Teams may be formed that consist of at most *two* persons. The following needs to be submitted:

- Source code of the emulator. It *MUST* be based on the starting point of this academic year, otherwise your submission will not be graded. Please do *not* submit binaries, object files (with the exception of micro programs) or PDF files of the assignment or OpenRISC ISA manual. Use `make clean`!
- The micro programs that have been written.
- A concise report in text format, `report.txt`. The report must describe:

- The contents of the different pipeline registers and why it was necessary to put these fields in the pipeline register.
- Which parts of your emulator do and do not work.
- Your approach to testing: which micro programs have been written and why?
- Any particular choices you have made for your implementation.
- For level 4: describe how you have dealt with delay slots in the non-pipelined case.
- For level 5: report for the different test programs the CPI with and without pipelining. How does it compare to the ideal CPI?
- In case you have worked in a team of two: each student should write a paragraph *individually* which describes your contribution to the project (what parts did you work on?) and how you experienced the collaboration with your lab partner (was the collaboration efficient? Did your lab partner put in enough effort in your opinion? Etc.). If there are disagreements within the team, this text may also be directly submitted to the lecturer by e-mail on an individual basis.

Ensure that all files that are submitted (source code, report, etc.) contain your names and student IDs! Please create a "gzipped tar" archive of your source code:

```
tar -czvf lab2-sXXXXXXX-sYYYYYYY.tar.gz lab2/
```

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs. Submit the tar-archive and your report through the BrightSpace submission site. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team members submits the assignment, such that there is a single submission per team!*

Deadline: Friday, November 11, 2022.

Consider Friday, October 21 as an intermediate deadline for your instruction decoder and formatter to be fully functional for a substantial number of instructions. We will release a test file and reference output on BrightSpace this day.

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- [**1.5 out of 10**] Quality of the submission: report, layout and modularity of the code, quality and coverage of the micro programs that have been written as unit tests.
- [**6.5 out of 10**] Correctness. This will be judged by having your emulator execute different test programs in increasing complexity. We will stop as soon as the emulator does not execute a program correctly. This can be seen as a "level" that has been attained. The attained level determines the score for this component:

Level 0	0.5 points	`basic.bin` (non-pipelined, no delay slots)
Level 1	1.25 points	`hellonods.bin` (non-pipelined, no delay slots)
Level 2	1.75 points	`hello.bin` (non-pipelined, no delay slots)
Level 3	2.5 points	`comp.bin` (non-pipelined, no delay slots)
Level 4	3.0 points	Implement delay slots without pipelining. `hello4.bin`, `comp4.bin` should now work, as well as all preceding test programs.
Level 5	4.0 points	Implement hazard detection and stall insertion. Your emulator should be capable of running `basic.bin`, `hellonods.bin`, `hello.bin`, `comp.bin`, `hello4.bin`, `comp4.bin` unmodified with the `-p` option enabled.
Level 6	4.75 points	`brainfuck.bin` (non-pipelined and pipelined).
Level 7	5.5 points	Implement forwarding. Your emulator must be capable of correctly executing all prior test programs with the `-p` option enabled.
Level 8	6.5 points	`gfxtest.bin` and `mandel-gfx.bin` (non-pipelined and pipelined).

Note: we do *not* consider scores between levels (partially achieved levels). A score corresponding to a level can only be attained if the program for that level can be executed correctly.

- **[0.5 out of 10]** Robustness and correctness of the implemented instruction decoder and formatter. This will be tested using our own, private, unit test files.
- **[1.5 out of 10]** Robustness of the implemented instructions. This will be tested using our own, private, unit test suite.

In order to receive a sufficient grade (≥ 5.5), at least level 3 (`comp.bin`) must be attained, together with good scores for quality and robustness. Given that getting a perfect score for robustness is very hard, count on attaining at least level 4.

The programs that are submitted will be graded automatically (with the exception of the assessment of the quality of the submission). All source code that is submitted will be subjected to plagiarism checks.

Additional Help

To help you get started, below follows some more detailed information in addition to the section *Structure* above.

- Start with the instruction decoder and formatter, as described above under *Start With The Instruction Decoder*. Make sure a small number of instructions can be handled by your code. To implement the methods of the instruction decoder, you will need to refer to the ISA manual to learn how to decode, e.g., an `l.add` instruction. In particular, the ISA manual specifies where the different fields of the instruction are located in the instruction word and which opcodes correspond with which operations and instruction formats. There's also a short video on instruction decoding available on BrightSpace, which focuses on RISC-V, but translates to OpenRISC as well. An OpenRISC cheatsheet is available on BrightSpace.

- Once your decoder can handle a number of instructions, continue with making the necessary modifications such that the emulator is able to execute the `l.add` instruction (non-pipelined). Consider the description under *Getting The First Instructions To Work: ALU Operations*.

  For the instruction fetch stage: don't forget to also increment the `PC` with 4 (to start with, later to be modified to also take branch targets into account). Uncomment the code that checks for the "test end marker". This marker is a special codeword, with the value `0x40ffccff`, that will stop the emulator once this codeword is encountered (when in "unit test mode"). Note that if this marker were not present, the emulator would at some point read beyond the end of the memory (text segment in our case) which would lead to an abnormal program termination.

  Once fetch and decode works, you need to implement the required logic in the ALU (`alu.cc`, `alu.h`). After that, you can complete the implementation of the instruction by tying everything together in the different stages: fetch the instruction in IF, call the instruction decoder and fetch the register values in ID, configure the ALU and write the ALU result to the next pipeline register in EX, pass through the result in M and in WB write the result back to the destination register.

  You can verify the implementation of the instruction by running the provided example micro program `add.conf` in the `tests` subdirectory.

- As the next step, determine which instructions need to be implemented to be able to execute `basic.bin`. Use the disassembler ("or1k-elf-objdump", or the `-X` option of the emulator!).

- Implement all of these instructions one by one (non-pipelined). Write new and separate micro programs to test these instructions. Note that `basic.bin` is still a simple test that is terminated using the test end marker as described above.
- After `basic.bin`, continue with `hellonods.bin`. Note that the last instruction that is executed by `hellonods.bin` is a store instruction to a specific memory address. At this particular memory address a system control module is connected. This module ensures that the emulator will halt and is cleanly shutdown. So, a correct execution of `hellonods.bin` will not lead to an "abnormal program termination", instead "System halt requested" should appear in the terminal.
- Continue in the same way for the more complicated test programs. For implementing delay slots and later pipelining, also see the hints under *Branch Delay Slots* and *Towards Pipelined Execution* respectively.