Architecture II Coursework

There are three central aims of this coursework:

- Solidify your understanding of how an instruction processor actually functions. The overall functionality of how a processor works is relatively easy to grasp, but there is lots of interesting detail which gives you some insight (both into CPUs, but also into software and digital design).
- Understand the importance of having good specifications, in terms of functionality, APIs, and requirements. This is fundamental to CPU design and implementation, but is also true in the wider world (again) of software and digital design.
- Develop your skills in coding from scratch. There is not much scaffolding here, I am genuinely asking you to create your own CPU simulator from scratch. You will also hopefully learn some important lessons about reducing code repetition and automation.

Meta-comment

You might find this document very verbose, and there are lots of clauses, clarifications, and restrictions on what you can do. So try to think of it from the other side, and imagine you're trying to write a spec that: 1 - will allow around 15 different simulators and testbenches to inter-operate perfectly with each other. 2 - gives as much freedom as possible in the implementation of both simulators and testbenches. 3 - allows both the simulators *and the testbenches* to be accurately tested/asssessed.

The only way of achieving this is to try to define very clear APIs. You then have to try to imagine all the possible ambiguities and corner cases in the interpretation and implementation of this API, and try to close them down or disambiguate them. So many of the clauses and restrictions here will not seem relevant, unless you happen to think of doing something which hits one of the anticipated problems.

This specification will still be imprecise, and *will* evolve. Where there is still a lack of clarity, it will be fixed.

Specification

Your task is to develop a MIPS CPU simulator, which can accurately execute MIPS-1 big-endian binaries. You will also need to develop a testbench which is able to test a MIPS simulator, and try to work out whether it is correct.

Terminology

For the sake of clarity, this document will use the following terms:

- *Simulator* : The MIPS CPU simulator being developed by you. This program is running natively under a Linux/Windows/OSX Environment and has direct access to your files, the keyboard (stdin), the screen (stdout), etc. The simulator will be responsible for implementing a register file, program counter, and memory, and then sequentially executing MIPS instructions according to the MIPS ISA. This is the thing that you will spend the most time working on, and it is up to you to make sure that it implements the interface expected by a Binary, while interacting correctly with the Environment.

- *Binary* : The MIPS binary/program/executable which is currently being executed/run/simulated by your *Simulator*. Each time your simulator is run it will need to be given a binary, as by itself the simulator does nothing (just like a "real" CPU does nothing if you switch it on but don't give it instructions to execute). While a simulator can only execute one binary each time it is run, the set of binaries that it can run is unrestricted. You will develop your own test binaries, as well as executing binaries from 3rd-party sources.

- *Environment* : This is the thing which is hosting and executing the Simulator. Part of it is the operating system, but it also contains elements of the C run-time library (e.g. libc), and also some elements of the compiler itself. The distinction between OS and language run-time may not be obvious to you at the moment, but an example is `std::cout` and `printf`, which are part of the C++ and C run-time libraries (respectively). Neither of these functions is provided by Linux, instead it provides lower-level functions like `write`, while Windows provides `WriteFile`, and OSX has... something. A standards-conforming C++ program should not use OS specific calls like `write`, but instead relies on the run-time library to provide a compliant environment.

- *Testbench* : This is your testing framework, which can take a given Simulator, and through running tests attempt to ascertain what features of the Simulator work. This should serve both to help you test and develop your own Simulator, but also to act as a check on the functionality of any other Simulator. The aim is that your Testbench should be able to check the functionality of a Simulator at an Instruction granularity.

The MIPS ISA acts as the boundary between the Simulator and the Binary, so any correct Binary should run on any correct Simulator, and should deterministically do exactly the same thing. This is the same principle as for the Environment that is running your Simulator; you would assume that Linux+glibc are going to run your Simulator correctly as long as your code plays by the rules, and the creator of any Binary will assume the same of your Simulator.

The target Evironment will be Ubuntu 16, with the standard GNU toolchain installed (i.e. `g++`, `make`), standard command line utilities, and bash. The lab Unix install should be a model of this environment, so anything that works in the lab should be correct. Feel free to use other environments during testing and development, but you should test in the target environment too.

Simulator Input/Output

Your Simulator will be a single executable, and has the following behaviour:

- *Binary* : the Binary location is passed as a command-line parameter, and should be the path of a binary file containing MIPS-1 big-endian instructions. These instructions should be loaded into a fixed region of "RAM" with a known address, then execution should start at the first address in this region.

- *Input* : input to the simulated Binary will be passed in over the Simulator's standard input (`std::cin` or `stdin`), and mapped into a 32-bit memory location. If the Binary reads from the nominated memory location, it should be logically equivalent to calling `std::getchar` or `getchar` (and one approach would be for the Simulator to call these functions on behalf of the Binary).
- *Output* : output from the simulated Binary will be produced by writing to a mapped 32-bit memory location. Writing to the nominated memory location should be equivalent to calling `std::putchar` or `putchar` (and again, the Simulator could call these functions on behalf of the Binary).
- *Exit* : A Binary signals successful termination/completion by executing the instruction at address 0. This tells the Simulator that there are no more instructions to execute, and that it should exit. The return code of the Simulator is given by the low 8-bits of the value in register `$2`. These 8-bits should be used as a non-negative value to pass to `std::exit` or `exit`.

- *Exceptions* : The Binary may execute instructions which are illegal, and so result in exceptions which should terminate execution of the Binary. To indicate this, the Simulator should return one of the negative exit codes detailed later on.

- *Errors* : Errors may occur within the Simulator (as opposed to exceptions which are due to part of the Binary's logic). Examples might include instructions which aren't implemented (limited functionality in the Simulator), or IO failures (problems which occur due to run-time interactions between the Simulator and the Environment).

- *Logging* : A Simulator may choose to emit diagnostic/debugging messages at various points, in order to record what is going on. This is completely fine, but any diagnostic information *must* be written to `std::cerr` / `stderr`. Any output written to `std::cout` / `stdout` will be interpreted as output from the Binary.

Your Simulator *may* take other private command line parameters, for example to enable or disable extended debug features during development. These should have the form `--ext-XXX`, for any value of XXX, and may take optional values if you wish. Note that your Testbench should not rely on a Simulator supporting any private extensions, as they are not part of the API. Nor should your Simulator rely on any extended command line parameters being passed at run-time, as nobody else will know about the existence of these parameters.

Simulator build and execution

The compiler should be buildable using the command:

```
make simulator
```

in the root of the respository. This should result in a binary called `bin/mips_simulator`. An artificial requirement of this coursework for assessment purposes (i.e. it isn't really required for API reasons) is that the simulator is:

- A binary compiled from C++ sources.
- It can be compiled in the target Environment. This means that if the following sequence is executed:

```
rm bin/mips_simulator
make simulator
```

Then a new binary will be compiled from C++ sources that are included in the submission.

If we assume the existence of a Binary called `x.bin`, we would simulate it using:

```
bin/mips_simulator x.bin
```

On startup all MIPS registers will be zero, any uninitialised memory will be zero, and the program counter will point at the first instruction in memory.

A Simulator should not assume it is being executed from any particular directory, so it should not try to open any data files. It should also not create or write to any other files.

Testbench Input/Output

A Testbench should take a single command-line parameter, which is the path of the Simulator to be tested.

As output, the Testbench should print a CSV file, where each row of the file corresponds to exactly one execution of the Simulator under test. Each row should have the following fields:

```
TestId , Instruction , Status , Author [, Message]
```

Whitespace between fields and commas is not important.

The meaning of the fields is as follows:

- `TestId` : A unique identifier for the particular test. This can be composed of the characters `0-9`, `a-z`, `A-Z`, `-`, or `_`. So for example, ascending integers would be fine, or combinations of words and integers, as long as there are no spaces. Running the test-bench twice should produce the same set of test identifiers in the same order, and this should reflect the order in which tests are executed.
- `Instruction` : This should identify the instruction which is the *primary* instruction being tested. Note that many (actually, most) instructions are impossible to test in isolation, so a given test may fail either because the instruction under test doesn't work, or because some other instruction necessary for the tests is broken. The test should be written to be particularly sensitive to the instruction under test, so it looks for a failure mode of that particular instruction.
- `Status` : This will either be `Pass` or `Fail`. Note that a given test can only test so much, so it is entirely possible that a test might pass even if an instruction is broken. However, a `Fail` should be only be returned if the instruction under test (or another instruction) has clearly done something wrong.
- `Author` : The login of the person who created the test.

- `Message` : This is an optional field which gives more details about what exactly went wrong. This field is free-form text, but it must not contain any commas, and should only be a single line.

All fields are case insensitive, including `TestId`.

Testbench build and executable

The Testbench should be built (or otherwise setup) using:

```
make testbench
```

Note: it is entirely possible that nothing needs to happen when this is executed. It is to allow for freedom of implementation.

This should result in an executable called:

```
bin/mips_testbench
```

Note: this only needs to be an executable file; so unlike the Simulator it does not need to be binary built from C++, and could be a bash script.

The Testbench will *always* be executed from within the root directory of the submission, so you can use relative paths to data files.

Any temporary or working files created during execution should be created in a directory called `test/temp`. Any files considered to be output of the Testbench (for example per-test logfiles) should be created in `test/output`. However, there is no requirement that output is created in either directory.

An example of running the Testbench on it's own Simulator would be:

```
bin/mips_testbench  bin/mips_simulator
```

corresponding output might be:

```
0, ADDU, Pass, dt10
1, ADD, Pass, dt10
2, ADDI, Pass, dt10
```

If we assume a different Testbench, and have a Simulator at the path `../other-simulator/bin/mips_simulator`, then we could execute with:

```
bin/mips_testbench   ../other-simulator/bin/mips_simulator
```

and the corresponding output might be:

```
jr1 ,   jr,   Pass, dt10,   Single JR statement back to NULL
addi1 , addi, Pass, hes2,   Add 5 to $0
addi2 , addi, Fail, hes2,   Add -5 to $0
jr2 ,   jr,   Pass, hes2,   JR->NOP->JR->NOP
```

Memory-Map

The memory map of the simulated process is as follows:

```
Offset     |  Length     | Name       | R | W | X |
-----------|-------------|------------|---|---|---|-----------------------
--------------------------------------------
0x00000000 |        0x4  | ADDR_NULL  |   |   | Y | Jumping to this address
means the Binary has finished execution.
0x00000004 |  0xFFFFFFC  | ....       |   |   |   |
0x10000000 |  0x1000000  | ADDR_INSTR | Y |   | Y | Executable memory. The
Binary should be loaded here.
0x11000000 |  0xF000000  | ....       |   |   |   |
0x20000000 |  0x4000000  | ADDR_DATA  | Y | Y |   | Read-write data area.
Should be zero-initialised.
0x24000000 |  0xC000000  | ....       |   |   |   |
0x30000000 |        0x4  | ADDR_GETC  | Y |   |   | Location of memory mapped
input. Read-only.
0x30000004 |        0x4  | ADDR_PUTC  |   | Y |   | Location of memory mapped
output. Write-only.
0x30000008 | 0xCFFFFFF8 | ....       |   |   |   |
-----------|-------------|------------|---|---|---|-----------------------
--------------------------------------------
```

The Binary is not allowed to modify its own code, nor should it attempt to execute code outside the executable memory.

When a simulated program reads from address `ADDR_GETC`, the simulator should

- Block until a character is available (e.g. if a key needs to be pressed)
- Return the 8-bit extended to 32-bits as the result of the memory read.
- If there are no more characters (EOF), the memory read should return -1.

When a simulated program writes to address `ADDR_PUTC`, the simulator should write the character to stdout. If the write fails, the appropriate Error should be signalled.

Exceptions and Errors

Exceptions are due to instructions which the Binary wants to execute which result in some kind of exceptional or abnormal situation. Exceptions should not occurr due to bugs or errors within the Simulator. All exceptions are classified into three types, each of which has a numeric code:

- Arithmetic exception (-10) : Any kind of arithmetic problem, such as overflow, divide by zero, ...
- Memory exception (-11) : Any problem relating to memory, such as address out of range, writing to read-only memory, reading from an address that cannot be read, executing an address that cannot be executed ...
- Invalid instruction (-12) : The Binary tries to execute a memory location that does not contain a valid instruction (this is not the same as trying to read a value that cannot be executed).

If any of these exceptions are encountered, the Simulator should immediately terminate with the exit code given using `std::exit`. Please note than an exception does not automatically mean that a Binary must be incorrect or buggy. For example, there are very well-defined situations where arithmetic overflow occurs, and a Binary may choose to rely on this behaviour for performance reasons, rather than explicitly checking for overflow all the time. Indeed, this performance argument is a big reason for hardware overflow exceptions, so a Binary *must* be able to rely on them being correctly reported.
Errors are due to problems occuring within the simulator, rather than something that the Binary did wrong. As with exceptions, an error may indicate a genuine problem with the Simulator, or it may be due to an interaction between the Simulator and the Environment. An example of the former is where a Simulator doesn't support a particular op-code (yet), so cannot execute a correct Binary.

An example of an error which is *not* the Simulator's fault is where the Binary has tried to output a character, but the request to the Environment has failed in some way. You may never have worried about it, but `std::cin >> x` can fail in various ways, and this would not be the fault of the Binary (so is not an exception).
Error codes are:

- Internal error (-20) : the simulator has failed due to some unknown error
- IO error (-21) : the simulator encountered an error reading/writing input/output

Instructions

Instructions of interest are:

Code	Meaning
ADD	Add (with overflow)
ADDI	Add immediate (with overflow)
ADDIU	Add immediate unsigned (no overflow)
ADDU	Add unsigned (no overflow)
AND	Bitwise and
ANDI	Bitwise and immediate
BEQ	Branch on equal
BGEZ	Branch on greater than or equal to zero
BGEZAL	Branch on non-negative (>=0) and link
BGTZ	Branch on greater than zero
BLEZ	Branch on less than or equal to zero
BLTZ	Branch on less than zero
BLTZAL	Branch on less than zero and link
BNE	Branch on not equal
DIV	Divide
DIVU	Divide unsigned

Code	Meaning
J	Jump
JALR	Jump and link register
JAL	Jump and link
JR	Jump register
LB	Load byte
LBU	Load byte unsigned
LH	Load half-word
LHU	Load half-word unsigned
LUI	Load upper immediate
LW	Load word
LWL	Load word left
LWR	Load word right
MFHI	Move from HI
MFLO	Move from LO
MTHI	Move to HI
MTLO	Move to LO
MULT	Multiply

Code	Meaning
MULTU	Multiply unsigned
OR	Bitwise or
ORI	Bitwise or immediate
SB	Store byte
SH	Store half-word
SLL	Shift left logical
SLLV	Shift left logical variable
SLT	Set on less than (signed)
SLTI	Set on less than immediate (signed)
SLTIU	Set on less than immediate unsigned
SLTU	Set on less than unsigned
SRA	Shift right arithmetic
SRAV	Shift right arithmetic
SRL	Shift right logical
SRLV	Shift right logical variable
SUB	Subtract
SUBU	Subtract unsigned

Code	Meaning
SW	Store word
XOR	Bitwise exclusive or
XORI	Bitwise exclusive or immediate
--------	--
INTERNAL	Not associated with a specific instruction
FUNCTION	Testing the ability to support functions
STACK	Testing for functions using the stack

The final instructions are pseudo-instructions, for cases where they don't map to a single instruction. You are not required to use them, but they may be useful for tests which are looking at more complex functionality, rather than narrowly looking at one.