G1
Interpreting Instructions — Simulating Machines

Annie, Oleks, ARK15 Course Team, see also Major Contributors

Version 1.2, September 17, 2015

Table of Contents

This is the first in a series of three G-assignments ("G" for "Godkendelse" and/or "Gruppeopgave") which you must pass in order to be eligible for the exam in the course Machine Architecture (ARK) at DIKU. We encourage pair programming, so please form groups of 2-3 students.

Although this G-assignment is fairly self-contained, we strongly encourage you to at least skim the first two chapters of [COD], as well as Appendices A.1 to A.6. You will find it extremely convenient to have the "green card" by your side.

If you have any comments or corrections to the text, visit our public GitHub repository at https://github.com/onlineta/ark15.

Happy hacking :-)

1. Assignment

This is a short overview of your assignment. Flip back to this if you are ever in doubt about what you are doing.

Your task is to write a simulator for a subset of the MIPS32 ISA. The simulator must be written in C. It must support the following MIPS instructions: `addu`, `addiu`, `and`, `andi`, `beq`, `bne`, `j`, `jal`, `jr`, `lw`, `lui`, `nor`, `or`, `ori`, `sll`, `slt`, `slti`, `srl`, `subu`, `sw`, and halt when it sees a `syscall` instruction.

The simulator must take two command-line arguments: A configuration file, and an executable ELF file. The configuration file contains the initial values for the registers `t0`-`t7`. The values are written as ASCII integers, no bigger than would fit in a `uint32_t`. For instance, if we want `t5` to be initialized to the value `42`, we should be able to provide a configuration file that looks like this:

```
0
0
0
0
42
0
0
```

After simulating the program in the given ELF file, the simulator should print the number of instructions executed and the contents of some prominent registers in hex. The output must be in the following "printf syntax":

```
Executed %zu instruction(s).
pc = 0x%x
at = 0x%x
v0 = 0x%x
v1 = 0x%x
t0 = 0x%x
t1 = 0x%x
t2 = 0x%x
t3 = 0x%x
t4 = 0x%x
t5 = 0x%x
t6 = 0x%x
t7 = 0x%x
sp = 0x%x
ra = 0x%x
```

OBS! Please follow this format precisely as your code will be subject to automated testing.

For the configuration given above, and a program that contains *just* the `syscall` instruction, your simulator must halt with the following output:

```
Executed 1 instruction(s).
pc = 0x40001c
at = 0x0
v0 = 0x0
v1 = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
t3 = 0x0
t4 = 0x0
t5 = 0x2a
t6 = 0x0
t7 = 0x0
sp = 0x4a0000
ra = 0x0
```

Lastly, follow the instructions in Submitting Your Solution.

2. Introduction

We begin with the exact same introduction (except for the last section) as in our tutorial on Linux, Toolchains, and Assembly. We assume in the rest of this assignment that you have completed this tutorial.

An executable file is a series of instructions, called machine code, packed into a format that your operating system (Linux, OS X, Windows, etc.) understands. Machine code itself is *architecture dependent*: its precise format depends on the *instruction set architecture* for which it is intended. Your own machine is probably an x86-64 or ARM architecture, whereas in this course, we will study a variant of the MIPS instruction set: MIPS processors thrive in network routers and video game consoles.

Machine code is a series of bits which are understood by a CPU. Dealing with bits is not very programmer friendly. An XKCD comic puts text editors and bits into perspective:

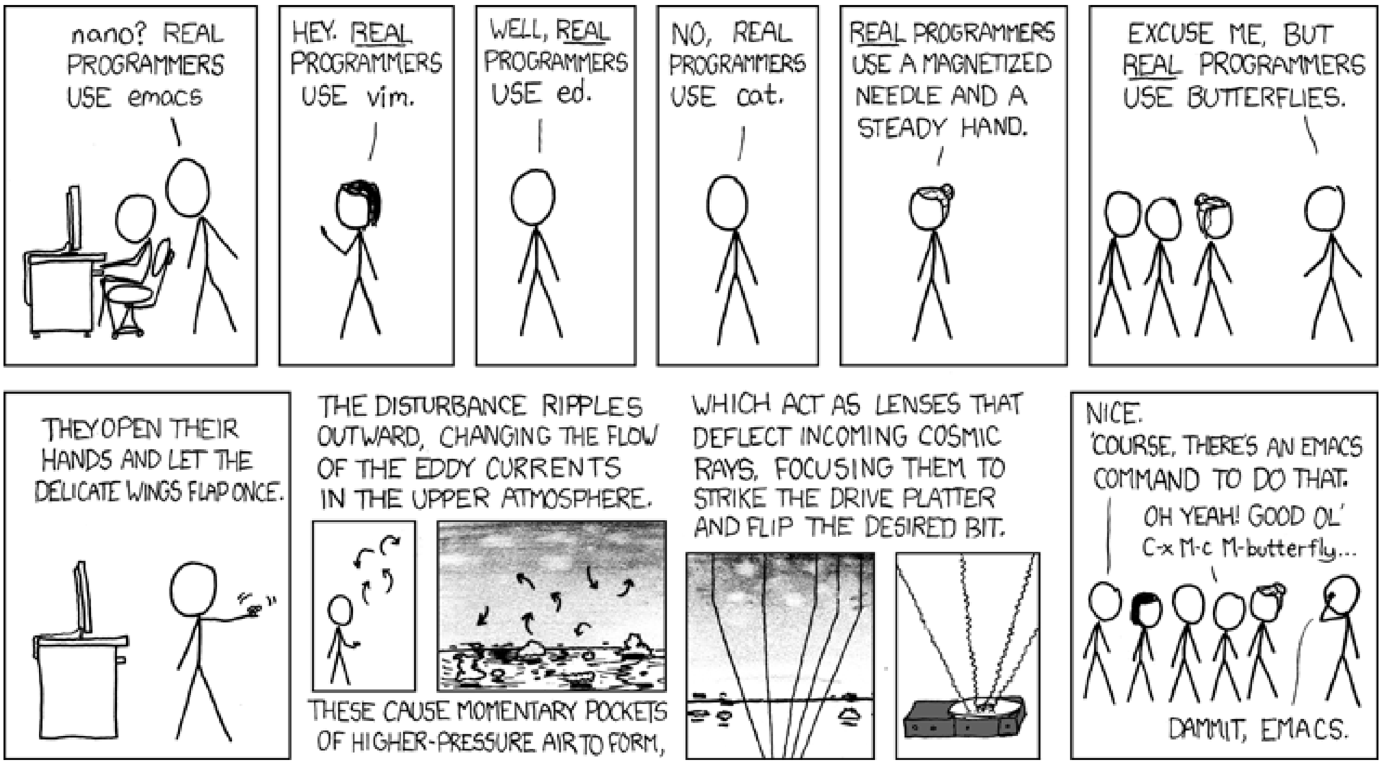

Figure 1. XKCD: Real Programmers (source: http://xkcd.com/378/).

Assembly language instructions are textual mnemonics (representations) for sequences of bits. Writing assembly is the closest to the hardware a programmer usually gets. Like machine code, assembly code is architecture dependent.

Assembly language is translated into machine code by an *assembler*. Higher level languages, such as C, are usually architecture *independent*, and code is converted into machine code (sometimes, assembly code) by a *compiler*.

The machine code is read into main memory upon execution. Small, fast CPU registers contain fixed-size bit-sequences, and are used for executing individual instructions: machine instructions operate on registers. In MIPS32, registers are 32 bits wide. We say that instructions operate on 32-bit *words*.

Computation on values in memory means that values must be explicitly read from memory into registers, and explicitly written from registers to memory. This wastes precious clock cycles. Working with memory is significantly slower than working exclusively with registers.

In MIPS32, there are 32 *general-purpose* registers. There are also a couple of *special-purpose* registers. The Program Counter (PC) register, contains the memory address of the instruction to be executed next. Instructions in MIPS32 are always one word, or 32 bits wide. Memory on the other hand, is addressed in bytes, that is, in terms of 8-bit sequences. After each instruction is executed, the PC must therefore be incremented by 4 to point to the immediately following instruction.

The exact format of an instruction depends upon the type of operation being carried out, but the most significant 6 bits always denote the "opcode", which designates the type of instruction to be executed. The rest of the bits making up the instruction contain register numbers, partial memory addresses, or additional parameters for the instruction.

2.1. Assembly and Linking

An assembler assembles assembly code into an *object file*. An object file is not directly executable, but it begins with a header, providing information about the remaining contents of the file. The contents may include:

- A text section containing the machine code.
- A (static) data section containing the data that must persist throughout the lifetime of a program.
- Relocation information, allowing the text and data segments to be moved around in memory.
- A symbol table, matching externally visible labels to machine code addresses.

The format of an object file (the way the file is structured) varies across operating systems. For Linux, this is typically the Executable and Linkable Format, or ELF.

One or more object files can be linked together by a *linker*. A linker resolves internal references within an object file, and externally, to other files. A linker produces a file that your operating system knows how to execute. As you might have guessed, for Linux, this is typically also the Executable and Linkable Format (ELF).

Similar to an object file, an executable file begins with a header, and may contain, among other things, a text and data segment. The header provides information on how to set up the memory before executing the program: how to place the text and data segments for all branches, jumps, and loads to work correctly. The header also lists the address of the *entry point* instruction. This is the instruction which will be executed first.

If all this seems mysterious to you, take a look at the Appendices A.1 to A.5 in [COD].

2.2. Compiling

A compiler for a high-level language, e.g. C, produces either an assembly-, object-, or executable file.

2.3. Simulation

Simulation is the imitation of the operation of some real-world system on another system.

Computer architecture simulation is often used in connection with computer architecture design to measure the costs and benefits of various design choices, without putting the decisions down in silicon. The task of taking a *new* computer architecture from design to physical chip, requires a substantial amount of money, manpower, and months of hard work. Software simulators, on the other hand, are quick to write and easy to change.

In this course, we will study the MIPS32 instruction set. The machine you are using to read (or used to print) this text is likely an x86-64 or ARM architecture. To execute MIPS32 instructions on your machine, you will need a simulator: a piece of software that imitates the operation of MIPS32 instructions using your native x86-64 or ARM instructions. You won't write the simulator in x86-64 or ARM assembly—that too is a matter of months of hard work. Instead, you will write it in C.

You have already heard of some simulators. Last week, we used the MARS MIPS Simulator to play around with MIPS32 assembly. Appendix A in [COD] refers to another simulator, called SPIM. Until you have a functioning simulator of your own, we recommend that you continue to use the MARS MIPS simulator when playing around with MIPS32 assembly.

3. Writing Your Simulator

3.1. Start With The Handout

Download and unpack the handed out g1-handout-v1.1.tar.gz archive.

"tar" is a classic archiving format on Unix-like systems. An archiving format packs multiple files and directories into one file. "gz" stands to signify that the archive is also compressed. Another archiving and compression format you might be familiar with is the ZIP file format.

The archive contains the following files and directories:

1. `mips32.h` with some very useful MIPS32 macros.
2. A simple ELF file parsing module in `elf.h` and `elf.c`.
3. A default configuration file, `default.cfg`, for your simulator.
4. A folder `asm` with a couple of MIPS32 assembly test programs and a Makefile. Once in the folder, type `make` to build MIPS ELF executables from the assembly source files.
5. A folder `c` with a simple C test program, and a Makefile. Once in the folder, type `make` to build MIPS ELF executables from the C source files.

NOTE Makefiles are explained in detail below.

You can use the `tar` command-line utility to unpack the archive:

```
~/ark$ tar xvf g1-handout-v1.1.tar.gz
```

3.2. Getting Started with C

3.2.1. Hello, World!

The `main` function is the entry point for your program. It takes two arguments, the argument count (`argc`) and the arguments themselves as an array of strings (`argv`).

Create a file called `sim.c`, with the following code:

~/ark/1st/sim.c

```
int main(int argc, char *argv[]) {
  // ...
}
```

The `main` function always returns an integer indicating whether execution went as planned (0) or resulted in an error (any other integer).

In order to be able to read and write to files or the terminal, the library `stdio.h` must be included. This is done using a preprocessor directive, or "macro". The C preprocessor runs immediately before the compiler compiles the program, and does a search-and-replace to expand all macros. Preprocessor directives are lines starting with a `#` sign. Add the line:

~/ark/1st/sim.c

```
#include <stdio.h>
```

above your defined `main` function. Then, replace the ellipsis with:

~/ark/1st/sim.c

```
printf("Hello, world!\n");
return 0;
```

Your final program should now look like this:

~/ark/1st/sim.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, world!\n");
  return 0;
}
```

The `\n` is a control code for a new line. No code after the `return` statement will be executed. Now compile and link:

```
~/ark/1st$ gcc -c sim.c
~/ark/1st$ gcc -o sim sim.o
```

The first command compiles `sim.c` to an object file, `sim.o`. The second command links the object file, producing an executable. The `-o sim` flag tells `gcc` to name the executable `sim`. The executable can now be run from the terminal:

```
~/ark/1st$ ./sim
Hello, world!
~/ark/1st$
```

TIP

Why `./sim`, and not just `sim`?

Normally, when you type a command in your shell and press enter, your shell searches the directories in your `$PATH` environment variable for an executable file matching the name of the command. To see the content of `$PATH` on your system, type

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
$
```

To see which file actually gets executed when you run a given command, use `which`:

```
$ which gcc
/usr/bin/gcc
$
```

Since our simulator is not located in a directory referenced by `$PATH`, we use the `./` prefix to tell the shell to look for a locally referenced executable.

3.2.2. More on Header Files

Header files have the filename extension `.h`, and are usually used for preprocessor definition directives, data structure definitions, and function prototypes. Function prototypes are function definitions with no corresponding body, which specify the *shape* of the function. The compiler then knows to look in the corresponding `.c`, or compiled `.o` file for the actual implementation of the function.

The angular brackets in the directive

```
#include <stdio.h>
```

caused the preprocessor to search system library directories for the file `stdio.h`. To include local header files, use double quotes (`""`) instead of chevrons (`<>`).

A header file, mips32.h, containing some useful MIPS32 macros has been provided for your convenience. Add the following line at the top of your `sim.c`:

```
#include "mips32.h"
```

When using quotes (`""`) rather than chevrons (`<>`), the C preprocessor will first and foremost look for the header file in the local directory relative to the including file. In our case, `~/ark/1st/sim.c`. Conversely, when using chevrons, the C preprocessor will start by looking at the system-wide include directories first, such as `/usr/include/`.

EXERCISE

Make sure that `sim.c` still compiles.

TIP

Compiling directly to an executable

`gcc` can compile a C file directly to an executable, keeping the intermediate object file in memory:

```
~/ark/1st$ gcc -o sim sim.c
```

Note, that we pass the C filename as an argument, not the object file as we did before.

3.2.3. Your Canonical Build System — `make`

`make` is a canonical command-line utility, used in Unix-like programming environments for building software projects of all shapes and sizes. We will only briefly mention some of the aspects of `make`. If you want to know more about make, we humbly recommend this tutorial.

The `make` command revolves around the notion of a `Makefile`. Create a file called `Makefile` in your `~/ark/1st/` directory. Start with these lines:

~/ark/1st/Makefile

```
CC=gcc
CFLAGS=-Werror -Wall -Wextra -pedantic -std=c11
```

`CC` and `CFLAGS` are now variables that can be used throughout the Makefile. For instance, instead of writing

```
gcc -Werror -Wall -Wextra -pedantic -std=c11 -o sim sim.c
```

in our Makefile (or in the terminal), we could now write

```
$(CC) $(CFLAGS) -o sim sim.c
```

in our Makefile, with the same result. This way, every time we compile, we use the same compiler and command-line arguments. Typing out all those arguments every time would've been laborious, non-maintainable, and error-prone.

IMPORTANT

Arguments to `gcc`

Out of the box, the `gcc` compiler is rather naïve. It is easy to write bad programs. Travel a bit safer by always using these flags:

1. `-Werror` makes `gcc` treat all warnings as errors; the program will fail to compile if `gcc` has any warnings to report about your code.
2. `-Wall` enables all warnings; `gcc` comes with most warnings turned off; this makes it easy to get started with `gcc`, but also easy to write bad programs.
3. `-Wextra` enables additional, extra warnings.
4. `-pedantic` enables even more warnings, making `gcc` almost as pedantic as the teaching assistant who will mark your assignment.
5. `-std=c11` makes `gcc` compile with the C11 standard in mind; this is the most recent C language standard.

IMPORTANT

Unused parameters

If you try to compile `sim.c` with the above arguments to `gcc`, compilation will fail, and `gcc` will tell you that `argc` and `argv` are "unused parameters". Add the line `argc = argc; argv = argv;` to the beginning of `main` to trick `gcc` into thinking that these variables are in use. At some later stage, when `argc` and `argv` are in actual use (i.e. serve a purpose in the body of your `main` function), you can remove this line again.

A Makefile is structured in terms of *rules*. A rule is a list of *targets*, followed by a list of *prerequisites*, and a *recipe*.

A target is first-and-foremost a file that we build using a recipe. A recipe is a list of shell commands. In this case, we need to build `sim`. This target has some *prerequisites* (dependencies), namely `sim.c` and `mips32.h`: Whenever we change `sim.c` or `mips32.h`, `sim` becomes outdated. (We typically assume that system libraries, such as `stdio.h`, don't change very often, so they don't count as dependencies.) The recipe is also used for bringing the target *up to date* with its prerequisites.

Add the following rule to your Makefile (below the `CC` and `CFLAGS` variables):

```
sim: mips32.h sim.c
	$(CC) $(CFLAGS) -o sim sim.c
```

The general format of a `Makefile` rule goes as follows:

```
TARGETS : PREREQUISITES LINE-BREAK
TAB COMMAND LINE-BREAK
TAB COMMAND LINE-BREAK
TAB COMMAND LINE-BREAK
...
```

IMPORTANT

Every line of a recipe must begin with a **tab character**.

To quote the GNU `make` manual: "This is an obscurity that catches the unwary."

Remove your previous build of `sim`, and type `make` in the terminal to build it once again:

```
~/ark/1st$ rm sim
~/ark/1st$ make
gcc -Werror -Wall -Wextra -pedantic -std=c11 -o sim sim.c
~/ark/1st$ make
make: 'sim' is up to date.
```

`make` saves us some work by not compiling things again, if everything is up to date. Try modifying `sim.c` (e.g. add some comments) and see what happens when you run `make` again.

The reason you can get away with just typing `make` is that `sim` is the *default target* in our Makefile. The default target is the uppermost target in the Makefile, and is usually called `all`. Add an `all` target, listing `sim` as a prerequisite, just below your variables:

~/ark/1st/Makefile

```
all: sim
```

Add another canonical target to the bottom of your Makefile, called `clean`.

~/ark/1st/Makefile

```
clean:
	rm -f sim
```

TIP The `-f` argument "forces" `rm` to remove the file. In practice this means that warnings are suppressed if the file doesn't exist, and read-only files are deleted as well. `-f` should otherwise be used with caution.

We stated previously that a target is first-and-foremost a file, but `all` and `clean` are not files! Their recipes produce no such files. Such targets are called *phony targets*. Phony targets are always out of date. (Because otherwise, if the files `all` or `clean` did exist, they would always be up to date!)

At the very top of your Makefile, declare `all` and `clean` to be phony targets like so:

~/ark/1st/Makefile

```
.PHONY: all clean
```

Your Makefile should now look like this (remember the tabs):

~/ark/1st/Makefile

```
.PHONY: all clean

CC=gcc
CFLAGS=-Werror -Wall -Wextra -pedantic -std=c11

all: sim

sim: mips32.h sim.c
    $(CC) $(CFLAGS) -o sim sim.c

clean:
    rm -f sim
```

The handout included two directories `c` and `asm`. Each contains a Makefile we've written which will compile all the c and assembly code in the directories `c` and `asm`, respectively.

We can call `make` recursively on those directories using the `-C` command-line argument. For instance, try this out in your terminal:

```
~/ark/1st$ make -C asm
make: Entering directory '/home/archimedes/ark/1st/asm'
...
make: Leaving directory '/home/archimedes/ark/1st/asm'
```

Or this:

```
~/ark/1st$ make -C asm clean
make: Entering directory '/home/archimedes/ark/1st/asm'
rm -f *.o
rm -f *.elf
make: Leaving directory '/home/archimedes/ark/1st/asm'
```

It would be useful as we are developing the simulator to add new assembly and C files for testing, and quickly compile them as we go. Let's make this a part of our Makefile:

OPTIONAL EXERCISE

As part of your `all` target, make sure to recursively make the directories `c` and `asm`. As part of your `clean` target, make sure to recursively clean the directories `c` and `asm`.

TIP

Entering, leaving directory

You can silence recursive invocations of `make` a little bit by adding the command-line argument `--no-print-directory`.

3.3. The Interface

The simulator will be a program which takes two command-line arguments:

1. the path to a text file specifying the initial values of the temporary registers (this is useful for testing); and
2. the path to an ELF executable.

The 8 initial temporary values are given in a simple text file, separated by whitespace. For instance, if we want `t5` to be initialized to the value `42`, we can provide a configuration file like this:

```
~/ark/1st$ cat default.cfg
0
0
0
0
0
42
0
0
```

So if we have, `default.cfg` and an ELF executable `~/ark/1st/asm/addu.elf`, we want to execute the simulator (which we'll call `sim`) like this:

```
~/ark/1st$ ./sim default.cfg asm/addu.elf
```

The `argc` and `argv` arguments in your `main` function can be used to fetch command-line arguments. We must first check that the right number have been given, before calling any other functions which use these arguments.

EXERCISE

Write an `if` statement in your `main` function, that checks if the number of arguments is 3. The first argument will always be the name of the executable (e.g. `"./sim"`). The rest, as passed on the command line. For instance, for the example above, the three arguments will be `"./sim"`, `"default.cfg"`, and `"asm/addu.elf"` (in that order).

If there are not three arguments as expected, print a usage message informing the user how the program is intended to be used, and return `ERROR_INVALID_ARGS`. Use the `#define` to define `ERROR_INVALID_ARGS` at the top of your `sim.c`.

Otherwise, call a function called `read_config`, which takes one parameter, namely the second argument (`argv[1]`), and returns an integer. If the return value from `read_config` is non-zero, assume that an error has occured, and return this value from `main`.

In order to compile this, you will need to define a function stub for `read_config`, which does nothing, prior to the declaration of `main`:

~/ark/1st/sim.c

```
int read_config (const char *path) {
  path = path;
  printf("Readfile!\n");
  return 0;
}
```

The assignment of `path` to itself is necessary to suppress compiler warnings about unused arguments.

In C, a string is an array of characters in memory. A string is thus a pointer (denoted with an asterisk in the above) to the first character in the string. Strings are terminated with a nul (`\0`) byte.

Recompile and run your code. You should do this often to catch bugs before they multiply!

3.4. Setting up the Registers

In MIPS32, there are 32 general-purpose registers. In assembly, a register is referenced using either its name or number. In machine code, 5 bits (2^5 = 32) are sufficient to identify a register. We can use a static array `regs` to model the register file, where the array index corresponds to the register number. These registers will initially be set to 0.

MODELLING CONCEPT

Variables are "statically allocated" when declared outside of a function declaration. The conventional place to put a static variable declaration is at the top of a C file. **Think:** Why shouldn't we put static variables in header files?

The distinctive thing about static variables is that they have a static size, predetermined at compile-time. They are 0-initialized at program start-up (before running `main`), and are available to all functions in the C file throughout the lifetime of the program.

Static variables are convenient for modelling elements of a predetermined size, which should persist throughout the lifetime of a program. For instance, the registers in a MIPS32 simulator.

In MIPS32, all registers are 32 bits in size. Many standard C types, including `int`, are not guaranteed to be of some exact size on all CPU architectures. Yet, it is most appropriate to model a register with a type we know is always exactly 32 bits in size. Exact integer types are available in the standard C library, under the `stdint.h` header.

MODELLING CONCEPT

`uint32_t` is a data type defined in `stdint.h`, representing a 32-bit unsigned (nonnegative) integer. `uint32_t` is useful for modeling 32-bit registers.

EXERCISE

Include `stdint.h` at the top of your `sim.c`. Declare a static `uint32_t` called `PC`, and a static array `regs` for the general-purpose registers. You might find it useful to `#define` macros like `AT`, `V0`, `V1`, `SP`, `RA`, etc. standing in for the respective elements of the `regs` array.

It is time to fill in the `read_config` stub and initialize the temporary registers as specified in the given configuration file (`default.cfg`). This is done in 3 steps:

1. open the file for reading;
2. read the file; and
3. close the file.

3.4.1. Opening and Closing Files

It is convenient to handle the files on your computer as streams of bytes. The function `fopen` (also in `stdio.h`) can be used to open a file as a stream of bytes. The *manual page* for `fopen` can be found online, or read in the terminal by typing:

```
$ man fopen
```

We can see the function prototype:

```
FILE *fopen(const char *path, const char *mode);
```

Replace the first two lines of the body of your `read_config` function with the single line:

```
FILE *stream = fopen(path, "r");
```

which gives a file stream, opened in read-only mode, i.e. we can only read bytes off of the stream, but not e.g. write bytes to it.

EXERCISE

We need to check that this call is successful, so it should be wrapped in an `if` statement checking if the return value from `fopen` is `NULL`. You should always check function return values to see if a call was successful; the man-page section `RETURN VALUE` will specify which values indicate success and failure respectively. If the return value from `fopen` was `NULL`, return the constant `ERROR_IO_ERROR` from `read_config` (use `#define` to define the constant).

EXERCISE

Define a function stub `read_config_stream`, which takes a single variable of type `FILE *` as parameter and returns an integer. Call this function in `read_config` with `stream` as the argument, once the file has been properly opened. Use the man page for `fclose` to find out how to close the stream again in `read_config`, after the call to `read_config_stream`. Return `ERROR_IO_ERROR` if `fclose` returns anything other than 0. If `fclose` does return 0, return the return value of `read_config_stream` from `read_config`.

3.4.2. Reading Integers Off of a Stream

EXERCISE

Write a loop in `read_config_stream` which loops 8 times, calling `fscanf` in every iteration to read an unsigned integer into the appropriate slot of the `regs` array. You should only initialize the registers `t0` to `t7`.

The `fscanf` function takes three (or more) arguments: A file stream (`FILE *`), a format string, and one or more pointer arguments (whose type(s) must correspond appropriately to the pattern in the format string). For instance, the code

```
uint32_t v;
fscanf(stream, "%u", &v);
```

scans from the current position of `stream`, finds the first group of bytes that looks like a textual representation of an unsigned integer (`%u`), constructs an actual `uint32_t` value based on these bytes, and puts that value into the variable `v`. With each call to `fscanf`, we are searching further down the stream for bytes that look like the string representation of an unsigned integer.

Remember to check for error conditions from `fscanf`. If an error occurs, return an appropriate error value from `read_config_stream`.

For more, see the man page for `fscanf`.

3.4.3. Showing Status

It is worthwhile to check whether the initialization procedure actually works. Do this by getting a bit ahead and write a general status function for the simulator. The simulator status consists of the number of instructions the simulator has executed so far, and a printing of the PC and some of the prominent registers.

EXERCISE

Declare a static `size_t instr_cnt` which will be used to keep track of how many instructions the simulator has executed. `size_t` is the largest one-word integer data type on your machine. `size_t` is already defined in `stdio.h`.

Write a function `show_status` which takes no arguments, and call it from `main` after the registers are successfully initialized. Use the `printf` function to print the simulator status to the standard output. Use the formant `%zu` for `size_t` types, and `%x` for printing integers in hex. See the man page for `printf` for details on how to format output.

The printing should follow the format as given below in "printf syntax":

```
Executed %zu instruction(s).
pc = 0x%x
at = 0x%x
v0 = 0x%x
v1 = 0x%x
t0 = 0x%x
t1 = 0x%x
t2 = 0x%x
t3 = 0x%x
t4 = 0x%x
t5 = 0x%x
t6 = 0x%x
t7 = 0x%x
sp = 0x%x
ra = 0x%x
```

OBS! Please follow this format precisely as your code will be subject to automated testing.

For the `default.cfg` given above, this should result in the following output:

```
Executed 1 instruction(s).
pc = 0x0
at = 0x0
v0 = 0x0
v1 = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
t3 = 0x0
t4 = 0x0
t5 = 0x2a
t6 = 0x0
t7 = 0x0
sp = 0x0
ra = 0x0
```

3.5. Setting up the Memory

MODELLING CONCEPT

Memory, similarly to registers, can be modelled with a static array. Unlike registers, memory is byte addressed; it is more natural to model memory by a static array of a byte-sized data type. C does not have a dedicated "byte" type, but C programmers canonically use the byte-sized `unsigned char`.

EXERCISE

Declare a static `unsigned char` array `mem` at the top of `sim.c` of size 640KB. For now, this will be our memory component. Define a macro `MEMSZ` which holds the static size of `mem`.

We would like our simulator to run MIPS32 ELF executables. That way, we can easily run both simple programs written in assembly, as well as more complicated programs written in e.g. C. Dealing with the ELF file format directly is a somewhat laborious task. Although there are some useful build utilities like `mips-elf-objcopy`, we find it more flexible to offer you a simple ELF parser written in C. You will find the ELF parser in the handed out `elf.h` and `elf.c`.

To use the parser, you need to do three things:

EXERCISE

1. Include the header file `elf.h` at the top of your `sim.c`.
2. Add an `elf.o` target to your Makefile. List `elf.h` and `elf.c` as its prerequisites. Use the `-c` flag to `gcc` to compile `elf.c` into an object file.
3. Modify the `sim` target in your Makefile. List `elf.o` as a prerequisite. It is now necessary to pass `elf.o` on to `gcc` to successfully compile `sim.c`:

~/ark/1st/Makefile

```
sim: mips32.h elf.o sim.c
	$(CC) $(CFLAGS) -o sim elf.o sim.c
```

1. Modify the `clean` target in your Makefile to also delete all `*.o` files.

~/ark/1st/Makefile

```
clean:
	rm -f sim
	rm -f *.o
```

The ELF parser offers exactly one function:

~/ark/1st/elf.h

```
int elf_dump(const char *path, uint32_t *entry,
  unsigned char *mem, size_t memsz);
```

Like `fscanf`, `elf_dump` is an example of a function which takes both "regular" arguments (`path` and `memsz`), and result arguments, i.e. the addresses of variables in which to store the result of the function call (`entry` and `mem`). In particular, it takes a path to an ELF executable, reads the file, stores the entry point at the given `entry` address, and copies all program segments (including sections like `.text` and `.data`) into the memory starting at `mem`, writing at most `memsz` bytes past `mem`.

NOTE `size_t` is defined as an integer data type large enough to store any memory size or offset on your machine. `size_t` is already defined in `stdio.h`.

The effect of `elf_dump` on the memory starting at `mem` is the exact same as doing

```
~/ark/1st/asm$ mips-elf-objcopy -O binary addu.elf addu.bin
```

And then reading the contents of `~/ark/1st/asm/addu.bin` directly into the memory starting at `mem`.

EXERCISE

Call the `elf_dump` function from your `main` function, after the registers are successfully initialized. Here is how you might call `elf_dump` from `main`:

~/ark/1st/sim.c

```
elf_dump(argv[2], &PC, &mem[0], MEMSZ);
```

Make sure to check the return value of `elf_dump`. The entry point and memory is invalid so long as `elf_dump` returns a non-zero value.

C quirk

In our sample call to `elf_dump` we used `&mem[0]`. You might wonder why we couldn't just use `mem`?

It is a common misconception that C array names are mere pointers. The names of statically- or stack-allocated arrays are pointers to arrays of a particular size. So `mem` is a pointer to an array of size `MEMSZ`, whatever that is. A pointer with size information is different from a bare pointer. However, it is easy to construct a bare pointer from a value in C, by taking the address of (`&`) that value.

Integral types

The range of integral types in use, and their headers, is perhaps starting to get a bit overwhelming, so let's recap:

Type	Sort of values	Our use	`#include`
`int`	Positive and negative two's complement integers.	Function (and program!) return values.	(Nothing)
`uint8_t`	Unsigned (nonnegative) 8-bit integers.	(None)	`stdint.h`
`uint16_t`	Unsigned (nonnegative) 16-bit integers.	(None)	`stdint.h`
`uint32_t`	Unsigned (nonnegative) 32-bit integers.	Modelling 32-bit registers.	`stdint.h`
`unsigned char`	Smallest addressable unit of the machine that can contain a basic character set (typically an unsigned 8-bit integer).	Modelling bytes in memory.	(Nothing)
`size_t`	Memory sizes and offsets.	`instr_cnt` and fourth argument to `elf_dump`.	`stdio.h`

3.6. Interpreting Instructions

3.6.1. Memory Layout and Endianness

If you try to print the hex-value of `PC` after performing an `elf_dump`, you will see that the entry point is something like `0x400018` (somewhere beyond 4MB). How come the entry point is so high? If there are this many instructions, how come `elf_dump` succeeds even though we only allocate 640KB of memory?

This has to do with the conventional memory layout of a MIPS32 process, and how `elf_dump` supports this memory layout. You can read more about the MIPS32 memory layout in Appendix A.5 of [COD]. The crucial detail is that the lower 4MB are reserved for the operating system. Your linker assumes that this is always the case and *offsets* all machine code addresses (including the entry point) by `0x400000` (4MB).

`elf_dump` supports this memory layout in the sense that it fills `mem` starting at the first byte past address `0x400000`. We can illustrate this as follows:

```
=========================================================
=      MIPS32 Memory Layout   =     mem Memory Layout   =
=========================================================
=           +--------------+  =     +--------------+    =
=           |   Stack top  |  =     |   Stack top  |    =
=                 ...         =           ...           =
=           | Data segment |  =     | Data Segment |    =
= 0x400000  | Text segment |  = 0x0 | Text Segment |    =
= 0x0       | Reserved     |  =     +--------------+    =
=           +--------------+  =                         =
=========================================================
```

Luckily for you, the header file `mips32.h` already defines the macros `GET_BIGWORD` and `SET_BIGWORD` which take care of offsetting the address before dealing with the memory component. For instance, we can use `GET_BIGWORD(mem, PC)` to get the `mem` instruction currently pointed to by `PC`.

> **EXERCISE**
>
> After successfully initializing registers and memory in `main`, set the `SP` (stack pointer register) to point to the top of the stack. This is the 4th last byte in `mem`. As with any other memory address, the stack pointer should be offset by `MIPS_RESERVE` (`0x400000`, defined in `mips32.h`).

`GET_BIGWORD` and `SET_BIGWORD` do more than merely offset the addresses. You might have noticed that MIPS32 is a so-called "big-endian" architecture, while your own machine is likely an x86-64 architecture, which is "little-endian".

The best choice of endianness is subtle, often accidental, and there is no clear benefit of one over the other. In a big-endian architecture, the bytes of a word (or half-word) are stored in order of decreasing significance (most-significant byte first). In a little-endian architecture, the bytes of a word (or half-word) are stored in order of increasing significance (least-significant byte first). The choice of endianness has no effect on the order of the words (or half-words) themselves.

We can illustrate this difference with our go-to example-instruction, `addu $2, $4, $5`:

`addu $2, $4, $5` *in big-endian format*

```
00000000 10000101 00010000 00100001
```

`addu $2, $4, $5` *in little-endian format*

```
00100001 00010000 10000101 00000000
```

In the ELF file format, the memory segments are stored with an endianness expected by the target architecture. So for MIPS32, big-endian. `elf_dump` performs no endianness conversion before storing data in `mem`, and so all data in `mem` is in big-endian format. Luckily, you don't have to think too much about this as long as you use the macros `GET_BIGWORD` and `SET_BIGWORD` whenever you're dealing with memory.

MODELLING CONCEPT

We used an array of `uint32_t` values to model registers, and an array of `unsigned char` values to model memory. If you are running a little-endian architecture, this means that we model e.g. registers with little-endians, and memory with big-endians. This inconsistency is unsettling, but dealing in native data types, such as `uint32_t`, is a lot more simple than juggling bytes.

3.6.2. The Interpretation Loop

`mips32.h` also defines many other useful macros for dealing with MIPS32 instructions.

EXERCISE

Define a function stub `interp`, which takes no arguments and returns an integer.

Modify your `main` function to do 5 things:

1. initialize the registers;
2. read the ELF file;
3. initialize `SP`;
4. call `interp`; and
5. show resulting simulator status.

At least 3 of these steps may fail. Check the return values appropriately before going from step to step.

EXERCISE

`interp` should run an infinite loop, counting up the `instr_cnt` variable in every iteration. To perform an iteration, you should get the instruction currently pointed to by `PC`, and increment the `PC`.

To actually interpret the instructions (which might have an effect on either the contents of the `regs` or `mem`), define a function stub `interp_inst`, which takes the instruction as a `uint32_t` argument.

The return value of `interp_inst` should determine whether to continue or break out of the loop:

- You should break out of the loop, and finish simulation gracefully if `interp_inst` sees the special `syscall` instruction.
- You should break out of the loop and report an error if some error occurred in `interp_inst` (e.g. unsupported instruction).
- You should continue the infinite loop if `interp_inst` did not see a `syscall` instruction, and no error occurred during `interp_inst`.

MISSING ON THE "GREEN CARD"

You won’t find the `syscall` instruction on the "green card" in [COD]. It is an R-type instruction with funct `0xc`. There is already a `FUNCT_SYSCALL` in `mips32.h`. See below for details on how to handle R-type instructions.

The opcode of an instruction is located in its 6 most significant bits. A macro, `GET_OPCODE` has been defined in `mips.h` to extract these bits for you. The code `GET_OPCODE(inst)` returns the opcode of the instruction `inst`.

> **EXERCISE**
>
> In `interp_inst`, use a `switch … case` construct on the extracted opcode. Symbols have been defined corresponding to the different instructions by `#define` preprocessor directives in the provided `mips32.h` header file. If the instruction being handled is an R-type instruction, then call a function `interp_r`, which also takes the instruction as an argument. All the other defined opcodes should result in a call to a specific function for that instruction. Remember to break after each case!
>
> The default case should return `ERROR_UNKNOWN_OPCODE`.
>
> Come up with your own special return value in case you see a `syscall` instruction.

3.6.3. R-Type Instructions

R-type instructions all have the same opcode, 0. The actual operation to be carried out is specified by the `funct` field. As we have seen earlier, an R-type instruction has the following format:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[      opcode     ] [      rs      ] [      rt      ] [      rd      ] [ shamt   ] [  funct   ]
```

Luckily for you, the macros `GET_RS`, `GET_RT`, `GET_RD`, `GET_SHAMT` and `GET_FUNCT` have already been defined in `mips32.h`, which extract the corresponding fields from the 32-bit instruction using bitwise operations.

Both signed and unsigned versions of some of the instructions are implemented by the MIPS32 architecture. The difference between these is that the signed versions can cause an overflow exception. This causes the CPU to jump to a special memory address, known as an exception handler, before execution continues. You will learn more about exceptions later in the course/in the Operating Systems course. We will not handle this right now, and just assume that everything goes swimmingly.

EXERCISE

In interp_r, depending upon the value of the funct field of the instruction (use the predefined FUNCT_* constants), update the **contents** of the register corresponding to rd with the result of the operation on the source registers rs and rt (and shamt for the logical shift operations.) The semantics of these operations are described in Verilog on the "green card" in [COD]. Remember to break after each case. The default case should return ERROR_UNKNOWN_FUNCT.

These operations would be carried out by the ALU on a physical architecture.

Implement support for all the funct constants defined in mips32.h.

3.6.4. J-Type Instructions

The J-type instructions have the format:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[     opcode     ] [                              address                            ]
```

The address field of a J-type instruction is 26 bits long. As instructions are word aligned, we left-shift the address field by 2, to provide a word-aligned address. The (remaining) upper 4 bits are taken from the upper 4 bits of the incremented Program Counter (PC). This is called "pseudodirect addressing" (see also [COD]).

EXERCISE

Implement the j and jal instructions in their own functions, called from interp. The macros GET_ADDRESS and MS_4B has been defined for you in mips32.h.

3.6.5. I-Type Instructions.

The immediate instructions have the format:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[     opcode     ] [      rs      ] [      rt      ] [              immediate            ]
```

The immediate instructions operate on two registers and a 16 bit constant. This constant must be extended to 32 bits. Depending on the instruction, the constant is either sign-extended or zero-extended (or in the case of lui, shifted left). If you are in doubt about what to choose, take a look at the Verilog examples on the "green card" in [COD]. The macros GET_IMM, SIGN_EXTEND, and ZERO_EXTEND have been provided. After using GET_IMM to extract the immediate field (the 16 bit constant), you can use one of the `_EXTEND` macros to either sign- or zero-extend the constant.

EXERCISE

1. Implement the `beq` and `bne` instructions. Note, that the addresses for the branching instructions are contained in the immediate field, and are used to construct a word-aligned, PC-relative addresses. See also [COD] or the "green card" for implementation details.
2. Implement the `lw` and `sw` instructions. Remember to use the `GET_BIGWORD` and `SET_BIGWORD` macros for accessing the memory in `mem`.
3. Implement the `addiu`, `andi`, `lui`, `ori`, and `slti` instructions.

3.6.6. Pseudoinstructions

You already support the `move` and `nop` pseudoinstructions by supporting the instructions `addu` and `sll`, respectively. **Mental exercise:** How come?

3.6.7. Branch Delay Slot

For technical reasons (which will first become relevant in G2), the instruction immediately following a branch or jump instruction (e.g. `beq`, `bne`, `j`, `jal` and `jr`) is always executed. To accommodate this, the assembler will reorder your assembly, and if necessary, add a `nop` after the branch instruction.

You can turn this off by adding the assembler directive `.set noreorder` at the top of your assembly files. Alternatively, you can implement a branch delay slot. For the scope of this assignment, this will almost be a "hack" or "spoof". In G2, this will be a lot more elegant.

4. Submitting Your Solution

Follow these steps to submit your solution.

4.1. Finalize Your Solution

Clean up your code, remove superfluous code, and add comments for the non-trivial parts.

Write a **short** report (`g1-report.txt` or `g1-report.pdf`) documenting your solution. Discuss what works, what doesn’t, if anything. Discuss the design decisions you have had to make, if any. To back your claims, we humbly encourage you to fill `~/ark/1st/asm` with a plethora of tests. Discuss your tests (and how to run them) in your report.

Your report should be sufficient to get a good idea of the extent and quality of your implementation. **Your code will only be used to verify the claims you make in your report**.

4.2. Package Your Code

Use the `tar` command-line utility to package your code:

```
~/ark$ tar cvzf g1-code.tar.gz 1st
```

4.3. Submit on Absalon

Submit **two files** on Absalon:

1. Your report (`g1-report.txt` or `g1-report.pdf`)
2. Your archive (`g1-code.tar.gz`)

Remember to **mark your team members** on Absalon.

5. Optional: Interpreting C

We are not far from the bare essentials necessary to run simple C code on our simulator. Getting this to work is a completely optional, supplementary exercise.

Consider a very simple C program:

~/ark/1st/c/universe.c

```
int main() {
  return 42;
}
```

We already know how to compile programs using GCC, and you might've already guessed that we've already installed something called `mips-elf-gcc`. A naïve way to compile `universe.c` would be:

```
~/ark/1st/c$ mips-elf-gcc -mips32 -o universe.elf universe.c
mips-elf/bin/ld: warning: cannot find entry symbol _start; defaulting to
0000000000400018
~/ark/1st/c$ file universe.elf
universe.elf: ELF 32-bit MSB executable, MIPS, MIPS32 version 1 (SYSV),
statically linked, not stripped
```

We can see why this is naïve if we try to *disassemble* `universe.elf`:

```
~/ark/1st/c$ mips-elf-objdump -d universe.elf
...
00400018 <_init>:
...
004001a4 <frame_dummy>:
...
0040020c <main>:
...
00400230 <__do_global_ctors_aux>:
...
00400290 <_fini>:
```

It looks like GCC has generated a great deal of "bloat" around a rather simple C program. This is because GCC will by default package a couple of things for your convenience, should you choose to link your program against e.g. the standard C library.

We certainly don't need any such convenience here! `-nostdlib` to the rescue:

```
~/ark/1st/c$ mips-elf-gcc -mips32 -nostdlib -o universe.elf universe.c
mips-elf/bin/ld: warning: cannot find entry symbol _start; defaulting to
0000000000400018
~/ark/1st/c$ mips-elf-objdump -d universe.elf

universe.elf:     file format elf32-bigmips


Disassembly of section .text:

00400018 <main>:
  400018: 27bdfff8  addiu sp,sp,-8
  40001c: afbe0004  sw  s8,4(sp)
  400020: 03a0f021  move  s8,sp
  400024: 2402002a  li  v0,42
  400028: 03c0e821  move  sp,s8
  40002c: 8fbe0004  lw  s8,4(sp)
  400030: 27bd0008  addiu sp,sp,8
  400034: 03e00008  jr  ra
  400038: 00000000  nop
```

From the look of the output above, all we need to support in our simulator are the `addiu`, `sw`, `move`, `li`, `lw`, `jr`, and `nop` instructions, where `move`, `li`, and `nop` are pseudoinstructions.

NOTE

`li` *pseudoinstruction*

The `li` pseudoinstruction has 3 possible machine-code implementations, depending on the size of the constant written in the assembly:

1. For the interval [0..65,535] it is implemented as an `ori` instruction.
2. For the interval [-32,767..0], `addiu` is used instead.
3. For all other constants, it is implemented as a `lui` instruction, followed by an `ori`.

So as long as your simulator supports `ori`, `addiu`, and `lui`, it supports the `li` pseudoinstruction.

Let us briefly recap what this program does. First, it allocates 8 bytes of stack space, and uses 4 of those bytes to store the callee-saved frame pointer (`s8` is a synonym for `fp`). It then sets the stack pointer as the new frame pointer. This frame is never used, but it is set up. The program then stores the value `42` in register `v0`, clears the frame and returns to `ra`.

Crucially, `main` assumes that someone has already jump-and-linked to it. This is typically done by the operating system. As we have no operating system embedded in our simulator, we will wrap our C programs with some assembly. This assembly will define a `_start` label, jump-and-link to `main`, linking it back to a terminating `syscall` instruction:

~/ark/1st/c/_start.S

```
.globl _start
_start:
  jal main
  syscall
```

To wrap a C program, we merely need to list `_start.S` together with the main C file when compiling:

```
~/ark/1st/c$ mips-elf-gcc -mips32 -nostdlib -o universe.elf _start.S universe.c
~/ark/1st/c$ mips-elf-objdump -d universe.elf

universe.elf:     file format elf32-bigmips


Disassembly of section .text:

00400018 <_ftext>:
  400018: 0c100009  jal 400024 <main>
  40001c: 00000000  nop
  400020: 0000000c  syscall

00400024 <main>:
  400024: 27bdfff8  addiu sp,sp,-8
  400028: afbe0004  sw  s8,4(sp)
  40002c: 03a0f021  move  s8,sp
  400030: 2402002a  li  v0,42
  400034: 03c0e821  move  sp,s8
  400038: 8fbe0004  lw  s8,4(sp)
  40003c: 27bd0008  addiu sp,sp,8
  400040: 03e00008  jr  ra
  400044: 00000000  nop
```

`universe.elf` is no special ELF file, so it should be straightforward to run it on our simulator:

```
~/ark/1st$ ./sim default.cfg c/universe.elf
Executed 10 instruction(s).
pc = 0x400024
at = 0x0
v0 = 0x2a
v1 = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
t3 = 0x0
t4 = 0x0
t5 = 0x2a
t6 = 0x0
t7 = 0x0
sp = 0x4a0000
ra = 0x400020
```

Of course, we still have a long way to go before we can simulate our simulator on our simulator.

6. References

1. [COD] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Elsevier. 5th or 4th edition.

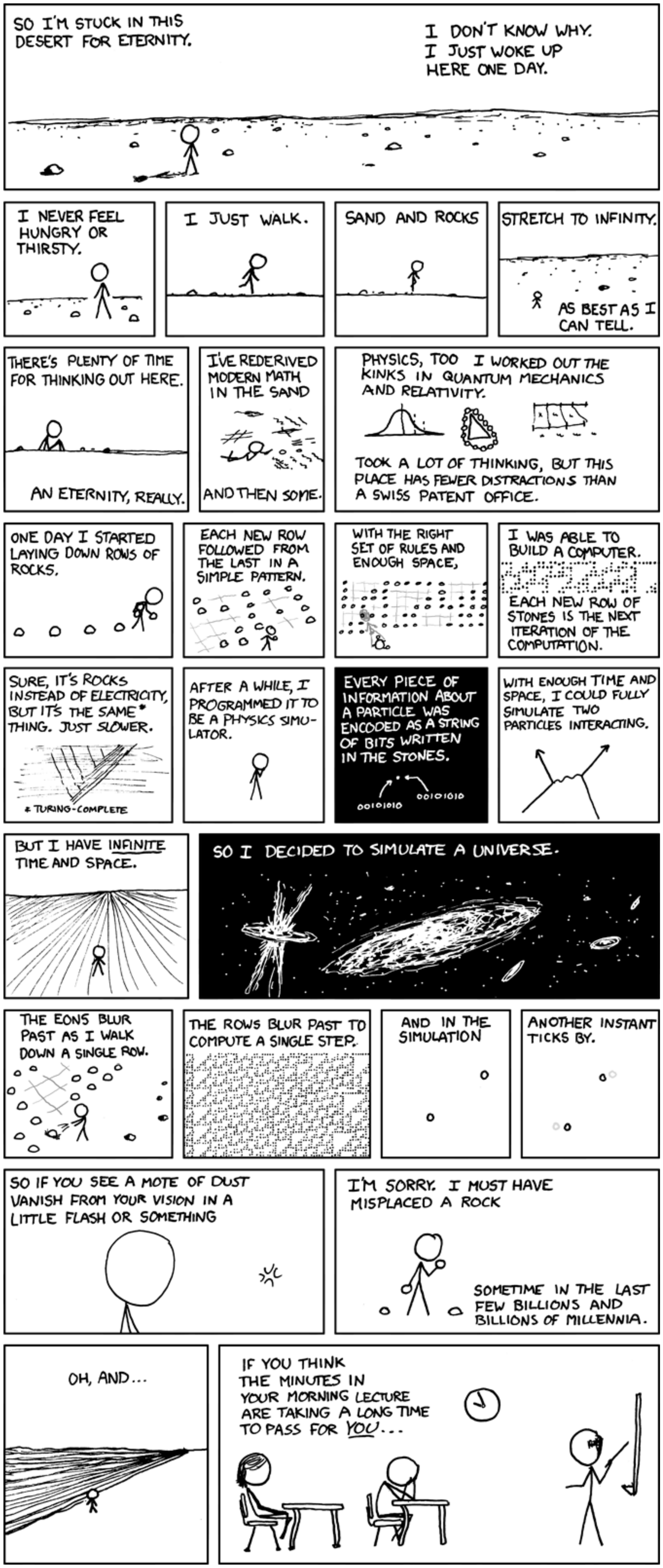
SO I'M STUCK IN THIS DESERT FOR ETERNITY.
I DON'T KNOW WHY. I JUST WOKE UP HERE ONE DAY.
I NEVER FEEL HUNGRY OR THIRSTY.
I JUST WALK.
SAND AND ROCKS
STRETCH TO INFINITY.
AS BEST AS I CAN TELL.
THERE'S PLENTY OF TIME FOR THINKING OUT HERE.
AN ETERNITY, REALLY.
I'VE REDERIVED MODERN MATH IN THE SAND
AND THEN SOME.
PHYSICS, TOO. I WORKED OUT THE KINKS IN QUANTUM MECHANICS AND RELATIVITY.
TOOK A LOT OF THINKING, BUT THIS PLACE HAS FEWER DISTRACTIONS THAN A SWISS PATENT OFFICE.
ONE DAY I STARTED LAYING DOWN ROWS OF ROCKS.
EACH NEW ROW FOLLOWED FROM THE LAST IN A SIMPLE PATTERN.
WITH THE RIGHT SET OF RULES AND ENOUGH SPACE,
I WAS ABLE TO BUILD A COMPUTER.
EACH NEW ROW OF STONES IS THE NEXT ITERATION OF THE COMPUTATION.
SURE, IT'S ROCKS INSTEAD OF ELECTRICITY, BUT IT'S THE SAME* THING. JUST SLOWER.
* TURING-COMPLETE
AFTER A WHILE, I PROGRAMMED IT TO BE A PHYSICS SIMULATOR.
EVERY PIECE OF INFORMATION ABOUT A PARTICLE WAS ENCODED AS A STRING OF BITS WRITTEN IN THE STONES.
00101010
00101010
WITH ENOUGH TIME AND SPACE, I COULD FULLY SIMULATE TWO PARTICLES INTERACTING.
BUT I HAVE INFINITE TIME AND SPACE.
SO I DECIDED TO SIMULATE A UNIVERSE.
THE EONS BLUR PAST AS I WALK DOWN A SINGLE ROW.
THE ROWS BLUR PAST TO COMPUTE A SINGLE STEP.
AND IN THE SIMULATION
ANOTHER INSTANT TICKS BY.
SO IF YOU SEE A MOTE OF DUST VANISH FROM YOUR VISION IN A LITTLE FLASH OR SOMETHING
I'M SORRY. I MUST HAVE MISPLACED A ROCK
SOMETIME IN THE LAST FEW BILLIONS AND BILLIONS OF MILLENNIA.
OH, AND...
IF YOU THINK THE MINUTES IN YOUR MORNING LECTURE ARE TAKING A LONG TIME TO PASS FOR YOU...

Figure 2. XKCD: A Bunch of Rocks (source: http://xkcd.com/505/).

7. Major Contributors

This text was made possible by the hard and enduring work of the entire ARK15 Course Team, and in particular the following members of the team:

- Annie Jane Pinder <anpi@di.ku.dk>
- Oleksandr Shturmov <oleks@oleks.info>

A special thanks to Phillip Alexander Roschnowski <roschnowski@gmail.com> for the meticulous proof-reading.