Compile C Code to Turing Machines

This document describes a suite of tools I developed to demonstrate concepts like Turing completeness and the Church-Turing thesis based on real modern processors. It is part of a lecture series for computer scientists at the Hamburg University of Applied Sciences.

If you spot any errors in this document or the accompanying software, please send me an email.

Installation

Everything you need for the following pages is run inside a Docker container, so you should be able to follow along no matter whether you are using Linux, Windows, or macOS. You just have to install Docker first. Some familiarity with Docker is helpful, but not really necessary. It is, however, assumed that you have used a command-line interface before and that you know the basics of working with a Unix shell. I believe this is not too much to ask for someone studying CS.

Once you have Docker up and running on your computer, you start the suite like this:

```
docker run -it hunchentoot/turing
```

If you do this for the first time, the image will have to be downloaded which might take several minutes; the next start will be much faster. If the command succeeds, you're in a Bash shell in a simulated computer running Linux and can continue with the next sections of this document. You can leave at any time by entering the command `exit` or by typing `Ctrl-D`.

A more elaborate version of the `docker` command above could look like this:

```
docker run -it \
       --hostname turing \
       -v "/home/fz/data:/home/tm/private:z" \
       --security-opt seccomp=unconfined \
       --name frunobulax \
       --rm \
```

```
       hunchentoot/turing
```

Everything except for the first and the last line is optional. The meaning of these options is as follows:

- `--hostname` is a name for the simulated machine. (You don't have to name it `turing`.) The main effect is that you'll get a nicer prompt.
- The `-v` line maps a part of your local file system into the simulated computer. This will enable you to transfer files between the container and your PC. Replace `/home/fz/data` with an existing directory on your real computer.
- The `-security-opt` line will suppress some warnings (about operations which are not permitted) you might see when running QEMU inside the container. They are harmless and can be ignored, but if they annoy you, use this option.
- With `--name` you can give a name of your choosing (not necessarily `frunobulax`) to the container. This can be useful if you've exited and later want to continue where you've left off. You would do that like so:

  ```
  docker start frunobulax
  docker attach frunobulax
  ```

  It is not strictly necessary to name a container to be able to re-attach to it later. If you don't do it, Docker will make up some name automatically. You can get a list of all your containers with `docker ps -a`.
- Finally, `--rm` means that the container will be immediately deleted after you leave it. Of course, in this case it is impossible to re-attach to it and it doesn't make sense to use the `--name` option together with `--rm`.

Updates

I might occasionally update the software to fix errors or add features. But Docker will not automatically pick up updates. The easiest way to make sure you're using the newest version is to run

```
docker pull hunchentoot/turing
```

every now and then. An alternative is to compare the output of

```
docker images hunchentoot/turing \
       --format "table {{.Tag}}\t{{.CreatedAt}}"
```

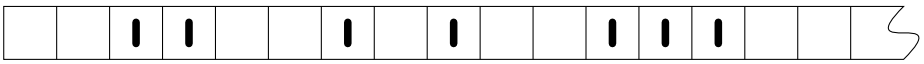
with https://hub.docker.com/r/hunchentoot/turing/tags.

The document you are reading might also get updated. The newest version will always be available at https://weitz.de/files/turing.pdf. Check the timestamp at the end of the last page.

The first version of the Turing machine

Turing machines are hypothetical devices invented by Alan Turing in a famous paper published in 1936. Although these machines were — by design — already pretty primitive, we will look at an even simpler version based on modifications by Stephen Kleene.

A Turing machine consists of a *tape*, a *read-and-write head*, and a (finite) set of *instructions*. The tape can be imagined as some kind of ribbon subdivided into cells where each cell is either empty or contains a single symbol which we'll denote as a vertical bar. The beginning of the tape could look like this, where we have bars in the third and fourth cell, in the seventh one, and so on:



The tape has a defined origin on the left but no fixed length. It is not infinite, but it is unbounded in the sense that it will always be "long enough" for what we want to do. We can imagine it to grow as needed.

The read-and-write head (which we'll just call *head* from now on) can move along the tape. It is always situated above one particular cell and it can only "see" this cell. It can also change the contents of the cell it's looking at, i. e., it can remove the bar symbol if there is one or it can add a bar symbol into an empty cell.

The machine is always in one of a finite set of *states* and how it operates depends on the contents of the tape and the aforementioned instructions. To be more precise, the machine operates in steps and each step consists of the following parts:

(i) Look at the current cell

(ii) Maybe change the contents of the current cell

(iii) Maybe move one cell to the left or right

(iv) Maybe switch into another state

What will happen in (ii) to (iv) depends on the instructions and these refer *only* to the current state and the symbol seen in (i). In particular, the machine doesn't "know" where on the tape it is or what else is on the tape apart from the current cell!

In our particular implementation of this kind of Turing machine, a single instruction always consists of three characters followed by a number like so:

```
I-R2
```

The first two characters are either `I` or `-` which stand for a bar or an empty cell, respectively.[1] The first character denotes the current contents of the cell the head is looking at, the second tells the head how the cell should look like afterwards. The third character is one of `L`, `R`, or `S` and it controls the movement of the head: move one cell to the left, move one cell to the right, or stay where you are. The number at the end is the number of the next state. So, the example above means: If the cell you're on has a bar in it, remove it, then move one cell to the right and switch to state number 2.

Of course, the machine wouldn't know what to do if the cell were empty, so typically you'd need two such instructions — one for `-` and one for `I` — and they have to be separated by the pipe character:[2]

```
I-R2 | --L3
```

This would mean that in case the cell is empty we leave it as is, move one position to the left and switch to the third state.

But that's not all. In order to keep things simple, I've left out two details so far. First, the third character can also be `H` which stands for *halt*, meaning that the machine will stop processing. (There's no "next" state after the machine has halted, but for the sake of a consistent syntax my program demands that a state number follows nevertheless.)

And second, what happens if the head moves too far to the left, beyond the start of the tape? We solve this problem by adding a pseudo symbol. If the head moves left from the first cell, it "sees" the symbol `*` (which is not really there but signifies "forbidden territory"). This symbol cannot be changed, nor can you move left from there. In other words, instructions starting with `*` must have as their second character another `*` and the third character must not be `L`. Also, as `*` is not really a symbol, it must not be the second character behind `-` or `I`. A complete set of instructions could thus look this:

```
I-R2 | --L3 | **R7
```

You might be wondering where the states are. Well, lines like this one essentially *are* the states. A "program" for a Turing machine is a text file consisting of a list of lines like the one above. The first line describes the behavior in the first state, the second line is for the second state, and so on. So, finally, here's our first complete "program":[3]

```
--R2 | I-R1 | **R1   # first state
--H2 | I-R1 | **R2   # state number 2
```

[1] We will sometimes call these *symbols* as well. In that sense, our machine has two symbols `I` and `-` and each new cell is "born" with the symbol `-` on it.

[2] It doesn't matter whether `I` or `-` comes first.

[3] By the way, `#` is treated as a comment character: everything following it up to the end of the line is ignored. This also applies to the applications introduced in the following sections.

The machine always starts at the first cell of the tape and is initially in the first state. The *input* to the machine is what's on the tape when it starts and its *output* is what's on the tape once it comes to a halt. The initial tape contents consist of a finite set of bars and empty cells at the beginning of the tape. All other cells — if more are needed — are initially empty.

> **Exercise 1**. I will sometimes interrupt the text with exercises like this one. If you're reading this in order to learn something, you should try to answer the questions first before reading on.
>
> What does the program do with its input?

The Docker container you installed in the first section contains a program called `tm` which can simulate the Turing machine just described. It expects the list of instructions in a simple text file. You could create this file from within the Docker container using the included Vim text editor or from the outside using a tool of your choice if you're using the `-v` option described above. For a tiny two-line file like this one, you can also create it directly like so:

```
echo $'--R2 | I-R1 | **R1\n--H2 | I-R1 | **R2\n' > Code/test.tm
```

Now run the simulation:

```
tm --tape=III Code/test.tm
```

This means that the code in the file `Code/test.tm` is executed with a tape that starts with three cells containing a bar. (Remember that all other cells are empty by definition.) What you should see is this:

```
*[1]III
*-[1]II
*--[1]I
*---[1]-
*----[2]-
*----[2]-
```

The program shows the relevant initial segment of the tape[4] for each step of the Turing machine. The square brackets show the current position of the head (which is always looking at the cell to the right of the closing bracket) and the current state (which is the number inside the brackets).

Do the same with other tapes like `--tape=II-II-I`. You can also use the abbreviated form `-t III-I` (with or without a space after `t`). As a general rule, all programs described in this document can be called with `--help` or `-h` to show a list of the available options. If you try `tm -h`, you will see some options which will only make sense later. Right now, only `-t`, `-s`, and `-p` are relevant: The argument to `-t` can also be the name of a file containing a tape or it can be a list like `5I2-3I` which would mean: five bars, two empty cells, three bars. The

[4] Usually at least as many cells as specified by the input and maybe more if the head went further to the right.

option `-s` controls how often the tape is shown. By default, the value is 1 which means that every step is shown. If you only want to see, say, every fifth step of a long-running program, enter `-s5`. With `-s0`, only the input and the result are shown. The option `-p` can be used to insert a small delay after each step is shown; this might be useful for debugging if you write your own programs.

Hopefully, it should now be clear what this particular set of instructions does: The machine moves to the right and stays in state 1 as long as it encounters bars, deleting each bar it sees. Once it comes across an empty cell, it switches to state 2. If it sees a bar in state 2, it also removes it and switches back to state 1. If, however, it sees an empty cell in state 2, it halts. In other words, the machine removes all bars from the tape up to a point where there are at least two empty cells in a row. (Try something like `-tII-I--I` or even `-t--I`.)

Before we continue, let me mention some "convenience features" as writing code for a Turing machine can become pretty painful once the logic is more complicated.

First, while each line should in theory contain an instruction for each of the three symbols `-`, `I`, and `*`, one often knows that certain situations can never occur. You can simply omit these instructions. In the case of `test.tm`, it is obvious that the machine will never see `*` because it never moves left. We could thus replace the program with this shorter one:

```
--R2 | I-R1
--H2 | I-R1
```

But what will happen if you make a mistake and omit an instruction that was needed after all? For each missing instruction, the compiler will insert a default statement of the form "stay in this state and at this cell without changing anything." The full version of the shortened code from above is thus:

```
--R2 | I-R1 | **S1
--H2 | I-R1 | **S2
```

Note that this can lead to an infinite loop if you omit an instruction for a situation that can actually happen! (In such a case, you can usually stop the simulation with `Ctrl-C`. If you started it with `-s0` and there is no output, it might be necessary to use `Ctrl-Z` to suspend the process and then kill it.)

The second convenience feature is about the fourth part of each instruction, the next state. You can specify it by the number of the state as above, but you can also use *relative targets* by prepending `+` or `-` to the number. `+2` in the third line would for example be interpreted as state 5. This means, we could have written our program like so:

```
--R+1 | I-R+0
--H+0 | I-R-1
```

Even better, you can also use *symbolic targets*. That means you can optionally start each line with an alphanumeric[5] label (followed by a colon) and use such labels instead of numbers. So, another equivalent version of our program could be the following one which also demonstrates that gratuitous whitespace is ignored. (But all instruction pertaining to the same state must still be on the same line!)

```
initial: --Rempty  | I-Rinitial
empty:
- - H empty | I - R initial
```

The Docker container contains some small demo programs for the Turing machine:

- `addBars.tm` adds two bars to the end of a sequence of bars, then moves back to the beginning and removes the first one.
- `incFirst.tm` interprets the tape contents as a list of two numbers separated by an empty cell: `IIII-II` would for example be $4,2$.[6] It increases the first number by one and makes sure the gap between the two numbers remains the same.
- `decSecond.tm` interprets the tape contents as a list of at least two numbers separated by empty cells: `II-I-IIII` would be $2,1,4$ while `-III-I-I` would be $0,3,1,1$. It subtracts one from the second number (unless it's already a zero) and shifts the following numbers one cell to the left so that the gaps between the numbers don't increase.
- `add.tm` adds two "numbers". The tape `II-III` will for example be converted to `IIIII`.
- `sub.tm` subtracts two "numbers". The tape `IIIIIII-III` will for example be converted to `IIII`. You will notice that subtraction is a bit more complicated than addition ...

You should look at the code, try to understand it, and play with the demos a bit.

Exercise 2. What will `addBars.tm` do if the tape contains more than one block of bars, like `III-I`, or if it doesn't start with a bar, like `-I`? Try to predict the behavior before you run the program. Try to "fix" the program if you don't like this behavior.

Exercise 3. Do `incFirst.tm` and `decSecond.tm` handle all edge cases correctly? What if there are fewer numbers than the program expects or too many? Can they cope with zeros? Try to improve them.

Exercise 4. Write a program that multiplies a number by 2, i. e., it should for example convert the tape `III` to `IIIIII`.

Exercise 5. If you're adventurous, write a program that multiplies two numbers.

[5] Labels must consist of characters of the latin alphabet, decimal digits, or the underscore character and must not start with a digit.

[6] This is the unary numeral system.

The `tm` program we've been using as well as most other programs in this suite were written in the programming language Julia. This is due to my personal preferences and I could have used Common Lisp, Python, or pretty much any other language instead. But it will be relevant for the next sections that the core of the program which performs the actual simulation of the Turing machine was written in C — although of course it code have been written in Julia as well. The source code for the C part[7] can be found in the file `~/TM/src/sim.c`. The Julia program converts the tape and the list of instructions into binary representations (an array of 8-bit bytes and an array of 32-bit words, respectively) and calls the C function with the corresponding addresses.

By the way, technically, there are infinitely many Turing machines. Each "program" (i. e., each set of instructions) defines one particular Turing machine: the code is "hard-wired." We might, however, sometimes refer to *the* Turing machine when we talk about the simulation program `tm`.

What is a program?

The previous section showed how painful even a simple task like the subtraction of two positive integers can be on a Turing machine — let alone multiplication or division. These extremely primitive devices were invented for mathematical proofs and never meant to be built and used for real-world computations. But the main purpose of this document is to demonstrate that they are in a certain sense as powerful as any modern computer.

We first need to clarify how this is meant. If you are using an application like Windows Calculator (or a similar tool for another operating system), you can type "2 + 5 = " and will get the answer 7. The Turing machine `add.tm` from above can turn the input `II-IIIII` into `IIIIIII`. But is that really the same? It is tempting to argue that `add.tm` just moves bars around that we are only *interpreting* as numbers while the calculator program really performs addition. But that's only true if we're ignoring a lot of what's going on "under the hood."

If we are typing, the actual input is a physical action (hitting keys) that is converted by the keyboard to electronic signals which in a complicated sequence of events will eventually be processed by the operating system and sent to the calculator application. Likewise, the answer 7 we're seeing consists of some pixels on our screen that the operating system has arranged in a certain way on behalf of the calculator. We're just *interpreting* this as the number 7 because we're used to the visual appearance of the symbol. What the calculator does internally is essentially nothing more than "moving bars around." The difference is that the "bars" are bits which are represented by the presence or absence of electrons in the PC's RAM.

The relevant point here is that from the viewpoint of theoretical computer science we are only interested in programs that convert some kind of input to some kind of output and we don't care how the input is delivered to the program or how we get at its output as long as we have a consistent *interpretation* of both. In other words, we abstract away the irrelevant details. In

[7] For more information about the source code see the last section.

that sense, when the calculator application adds two numbers it does the same thing as the Turing machine `add.tm`.

This will be the perspective used on the following pages. It also implies that we don't care about performance. A modern CPU will of course add two numbers several orders of magnitude faster than a Turing machine simulated by a Julia program running on the same CPU. But in this document we only want to know whether something can be done at all — not how long it takes. (The latter is the subject of computational complexity theory.)

Better Turing machines

Before we proceed, let's improve the Turing machines we have so far. It would be nice if we had another symbol so that we could use binary code instead of the unary number system we've employed so far.

The "trick" is to combine two tape cells into one: We interpret `I-` as 0, `II` as 1, and `--` as an empty cell. We could then write instructions like this:[8]

```
00R1 | 10S2 | --L1 | **H1
```

This will of course make the logic a bit trickier. Here's how a translation of these instructions into the "old" version could look like:

```
X1: --R+1 | IIR+2 | **H X1   #  1
--L+2                       #  2
IIL X1a | --L X1b           #  3
--R+1                       #  4, combined symbol "-"
--L+1                       #  5
--L+1 | IIL+1               #  6
--L X1 | IIL X1 | **S X1    #  7
X1a:IIR+1                   #  8, combined symbol "1"
I-L X2                      #  9
X1b:IIR+1                   # 10, combined symbol "0"
--RX1                       # 11
```

What's happening here? We introduce labels like `X1`, `X2`, and so on for lines of the code we want to translate. Line 1 looks at the first part of the combined symbol. If it is a star, we can take immediate action. If it is `-`, we know another hyphen must follow, but just in case (the tape might be corrupted) we look at it in line 2, then move back left and go to line 4. If, however, we've seen `I` in line 1, we move right, go to line 3 and from there branch to `X1a` or `X1b` depending on the next cell — which decides whether the combined symbol is `1` or `0`.

We're now at one of the lines 4, 8, or 10 and we know which combined symbol we've seen

[8] An empty cell in this version is again represented by one hyphen.

(and on the tape we're back where we were in line 1). We then write the new symbol (which of course consists of two parts) and then move the head according to the instructions. (The latter might again involve two steps if the movement is `L`).

Eleven lines of "old" code for just one line of "new" code! This is certainly something you don't want to do for a complicated set of instructions. The good news is that I've written a small Julia program that will perform the translation for you.[9] For example, `Code/inc.tm2` is a set of instructions to add 1 to a binary number. To look at the code and then at its translation you can do this:

```
cat Code/inc.tm2
transform Code/inc.tm2
```

And as `tm` will also accept its instruction from `stdin`, you can pipe the translation directly into it:

```
transform Code/inc.tm2 | tm -t IIII
```

This will turn the input `IIII` into `III-I-` which in translated form means that `11` (3) becomes `100` (4). As another convenience feature you can ask `tm` to do this translation for you:

```
transform Code/inc.tm2 | tm -t 100 -b
```

Exercise 6. We didn't use the sequence `-I` which is a bit wasteful. How would the translations change if `-I` represented another new symbol like 2?

Exercise 7. What would have to be done if we wanted to represent even more symbols? Imagine creating a Turing machine based on `tm` which can handle all ten decimal digits.

By now, you should hopefully be convinced that a Turing machine with an arbitrary number of symbols isn't more powerful than our orginal one — but maybe more convenient. Therefore, we will from now on use a more sophisticated version which can work with the symbols `0` und `1` and also with other sets of symbols.[10] Here are some things you can try:

```
tm2 -t 111 Code/inc.tm2     # increase binary number (code from above)
tm2 -t 1000 Code/dec.tm2    # decrease binary number
tm2 -t 10101 Code/palindrome.tm2          # 1 if palindrome, 0 if not
tm2 -c ABCD -t AABDCC Code/cycle.tm2      # other symbols than 0 and 1
```

Exercise 8. Imagine a Turing machine with two tapes and one head which scans both tapes in parallel. The head sees two symbols (one on each tape) at the same time and can of course also

[9] If you study its source you will notice that the translations could be optimized in some cases. I opted instead for easier code as this is just a demonstration.

[10] In particular, `tm2` does the same as `tm` if it is invoked with `-c I`. The default for `-c` is `01`.

write two symbols in one go. Convince yourself that simulating such a machine with `tm` is not harder than what we did above. (And the same is true if we want more than two tapes.)

Exercise 9. What *is* harder to simulate is a Turing machine with several tapes where each tape has its own head with its own set of instructions. Do you have an idea how this could be done? (Hint: Use a machine like in the previous exercise with twice the number of tapes. One half of the tapes will only be used to mark the current positions of the heads in the other half.)

A video of a "real" Turing machine can be seen at `https://aturingmachine.com/`.

Exercise 10. The video mentioned above shows a Turing machine with a tape that is unbounded in both directions. How could that be simulated using `tm`?

The RISC-V 32I CPU

There's a plethora of programming languages available today.[11] But in the end, every program you write needs to be translated (by a compiler or an interpreter) to the computer's "native language" which is the machine code of the CPU. So, if we're talking about what a computer can do, it suffices to contemplate the abilities of its CPU.

Modern CPUs like those of the x86-64 variety found in most Linux and Windows PCs (and also in older Macs) can do a lot of things. Besides simple integer and floating-point arithmetic (addition, multiplication, subtraction, division) they can also perform more complex tasks like computing logarithms or trigonometric functions. But what of this is really necessary and what are just convenience features?

One way to answer this question would be a comparison to older computers that could perform essentially the same tasks as contemporary ones, albeit slower. They had much more primitive CPUs like the Intel 8088 used in the original IBM PC or the MOS Technology 6502 used in the Apple II. The 6502, for example, couldn't even multiply or divide nor did it have any facilities to work with floating-point numbers. But these computers already had spreadsheet applications, word processors, and — most importantly — compilers and interpreters for several programming languages.

We don't have to look at the past, though. Even today, very simple CPUs are in widespread use. Some well-known examples are based on the RISC-V architecture developed by the University of California, Berkeley. This architecture has the advantage of a very clean, almost "academic" design on the one hand while on the other hand several companies are actually building and selling processors based on it. It is also supported by various open-source tools. That's why we'll be using it here and we'll concentrate on the most basic variant called RV32I.

Let's start with a simple demonstration, the traditional first program, written in C:

[11] Some people estimate that more than 10,000 languages have been designed in the last decades. However, most of them are probably niche languages or no longer in use.

```
buildRV32I RISC/hello.c
runQEMU RISC/hello.elf
```

You should see a message printed to the console. What's happening here?

`buildRV32I` is a script that calls the well-known GCC program which in turn compiles the C source code in the file `hello.c` to RISC-V 32I machine code. It will generate a file `hello.elf`[12] which is then fed (by the script `runQEMU`) into the program QEMU which simulates a RISC-V 32I CPU.

If you look at the source of `runQEMU`,[13] you will see that QEMU is instructed to simulate a virtual machine with no BIOS and with `hello.elf` as the kernel. This means that we are running a "bare-metal" simulation where there is no operating system or other supporting machinery whatsoever — just a CPU running our code.

But how could the program write text to the console? Well, QEMU simulates a minimal set of peripheral devices including a UART. The UART is associated with a dedicated memory location and every byte written into this part of the simulated RAM ends up on our console.

Let's have a look at `hello.c`:

```
#include "wrapper.c"
#include "lib.c"

void main(void) {
  putsRaw("The crux of the biscuit is the apostrophe.");
}
```

The first line includes a few lines of assembly which just set up the call stack, then invoke the function `main`, and finally make sure the whole program ends cleanly with the virtual machine shutting down. Every C program written for this kind of simulation must start with this particular line. (And if you want to experiment with your own programs, make a copy of this file and replace the `putsRaw` line with your code.)

The second line is optional. It adds some convenience functions to make sending output to the console a bit easier.[14] One of them is the function `putsRaw` which is used in `main` und which can be used like `puts` of the C standard library. A version of `hello.c` that doesn't use `lib.c` is in the file `hello2.c`:[15]

```
#include "wrapper.c"
```

[12] It actually generates two files. We'll talk about the second one later.
[13] Located in `~/RISC/runQEMU.sh`.
[14] You'll find the source code of both `wrapper.c` and `lib.c` in the `RISC` directory.
[15] The `volatile` qualifier is there to prevent GCC from performing unwanted optimizations involving `uart`.

```
static volatile char * const uart = (volatile char *)0x10000000;
static char msg[] = "The crux of the biscuit is the apostrophe.";

void main(void) {
  char *s = msg;
  while (*s)
    *uart = *s++;
  *uart = '\n';
}
```

OK, we can create output, but how do we get input into our RV32I programs? We can use `runQEMU` to write into the simulated machine's RAM before the program is started. To demonstrate this, the file `hello3.c` contains yet another version of the program above. In this case, the string to print is read from a hard-coded memory location.[16]

```
#include "wrapper.c"

static volatile char * const uart = (volatile char *)0x10000000;
static volatile char * const msg = (volatile char *)0x81000000;

void main(void) {
  volatile char *s = msg;
  while (*s)
    *uart = *s++;
  *uart = '\n';
}
```

If you compile and run `hello3.c` as before, you will see nothing, because QEMU fills the simulated RAM with zeros before any code runs.[17] But you can do this:

```
buildRV32I RISC/hello3.c
echo -n "Hello World!" > /tmp/hello
runQEMU RISC/hello3.elf 0x81000000:/tmp/hello
```

As an added convenience, the linker script `link.ld` used by `buildRV32I` sets up a couple of symbolic addresses so that we don't have to work with numbers. One of them is `mem` so that you could replace the two lines using `msg` in `hello3.c` with these:

[16] In QEMU, the program will start at `0x80000000` and everything before that address is reserved.

[17] Actually, QEMU allocates RAM and the Linux system in the Docker container fills it with zeros. But that minor difference doesn't matter here.

```
extern char mem[];              // instead of line 4
  volatile char *s = mem;       // instead of line 7
```

runQEMU also knows these names so that we could have called it like so:

```
runQEMU RISC/hello3.elf mem:/tmp/hello
```

mult.c in the RISC directory is another example program with input and output. It multiplies two 32-bit integers read from RAM and displays the result:

```
#include "wrapper.c"
#include "lib.c"                // declares mem address

void main(void) {
  i32 *p = (i32 *) mem;         // i32 is int32_t from <stdint.h>
  printInt(*(p++) * *p);        // printInt is defined in lib.c
  putcharRaw('\n');
}
```

We can use Julia to create a "RAM image" that QEMU can use:

```
buildRV32I RISC/mult.c
julia -e 'write("/tmp/factors", Int32[6,7])'
hexdump -X /tmp/factors                      # optional, just checking
runQEMU RISC/mult.elf mem:/tmp/factors
```

Finally, there's the example RISC/log.c which computes the natural logarithm of a 32-bit IEEE 754 floating-point number. You can use it as follows:[18]

```
buildRV32I RISC/log.c -lm
julia -e 'write("/tmp/input", Float32[1.74e18])'
runQEMU RISC/log.elf mem:/tmp/input
```

You're probably not very impressed by this. But the main point is that the RISC-V 32I CPU we're using here is essentially as primitive as the 6502 chip from 1975 (see above). It can't multiply or divide nor can it work with floating-point numbers! What we saw in programs like mult.c or log.c was done with the help of GCC libraries which simulate the "missing" capabilities.

If you want to see the assembly for the examples above, you can do something like this:[19]

[18] Note the -lm option for the C math library.

[19] The listing for hello.elf will be longer because it contains several functions from lib.c although not all of them are actually used. https://www.robertwinkler.com/projects/riscv_book/riscv_book.html is a free book that teaches RISC-V assembly programming, in case you're interested.

```
riscv64-unknown-elf-objdump -d RISC/hello2.elf
```

If you do the same for `log.elf`, you will see more than 5000 lines of code which simulate the computation of the natural logarithm of a floating-point number, but in the end this is all done with integer addition and subtraction plus some bitwise operations.

Simulating Turing machines with the RISC-V 32I CPU

At any rate, the consequence of what we saw in the previous section is that whatever you can do with your computer at home or even with a supercomputer can essentially also be done with a RISC-V 32I machine — if we ignore factors like speed or storage space. In principle, any C program can be compiled to this architecture. And if it can be done for C, it can — with enough effort — be done for other languages as well.[20]

For the purpose of this document, we are mostly interested in one particular program: the Turing machine simulator. And this can indeed be done with less than hundred lines of code — half of which are the same the Julia program `tm` uses. We just have to get the code and the tape into QEMU's simulated RAM.

That's what the `-d` option of `tm` is for. Try this:

```
tm -t IIIII-II Code/sub.tm -d /tmp/sub
```

Without `-d /tmp/sub`, the subtraction code would compute $5-2=3$. But now the program instead created two files `sub.tape` and `sub.code` in the `/tmp` directory. These are the binary representations (of the tape and of the instructions in `sub.tm`) we talked about earlier.

Now compile the RISC-V 32I version of the Turing machine simulator and start it with these files loaded into QEMU:

```
buildRV32I RISC/tm.c
runQEMU RISC/tm.elf code:/tmp/sub.code tape:/tmp/sub.tape
```

As you can see, this works exactly like before! (And it should be obvious that we could likewise simulate `tm2` Turing machines.)

The primitive counter machine

Our original goal was to show that Turing machines can do anything a modern computer can do. We can now be more specific and stipulate that we want to demonstrate that a RISC-V 32I

[20] Some features that usually require an operating system, for example dynamic memory allocation, would have to be implemented if the code is supposed to run on "bare metal." But that is of course possible. After all, the Linux kernel is written in C.

machine can be simulated by a Turing machine. As you can imagine, this won't be easy. We will thus proceed step by step.

The "improved" Turing machine `tm2` is easier to use for computations than the original `tm`, but it is still pretty clumsy. Our next step will be something that feels a bit more like a real computer, a so-called *counter machine*. We will start with a very primitive counter machine, due to Marvin Minsky.[21]

Our counter machine has an arbitrary number of *registers* called `R1`, `R2`, and so on. Each register can contain one natural number. Negative numbers aren't allowed, but other than on a real computer there's no size limit. A program for a counter machine consists of a list of instructions and only three types of instruction are possible:

- `INC R`*n* increases the number in the *n*th register by one.
- `DEC R`*n* decreases the number in the *n*th register by one if it is positive. If the number is zero, it is not changed.
- `JEQ R`*n*`,`*m* continues at the *m*th instruction if the number in the *n*th register is zero.

Apart from the possible "jumps" invoked by `JEQ`, instructions are executed one by one in the order given and the program halts once it reaches the line following the last instruction. Similar to what was possible in `tm` and `tm2`, the "target" in a `JEQ` instruction can also be specified relative to the current line (like `+3` or `-5`) or as a symbolic (alphanumeric) label.

The file `Code/add.reg` contains a very simple first program and shows another convenience feature:

```
JEQ R2,+4
DEC R2
INC R1
JEQ T0,1
```

The program adds the contents of `R1` and `R2` by accumulating the sum in `R1`: In a loop it decrements `R2` until this register has reached the value zero while at the same time incrementing `R1`.

But what is `T0`? The `T` here is for *temporary* and it works like this: Among the "normal" registers used in the program we look for the one with the highest number (in this case `R2`) and convert `T0` to the register following it (in this case `R3`). If there were also a mention of `T1`, it would be the next one (`R4`), and so on. Although this doesn't look like much, it will turn out to be pretty helpful later.

We will also use the convention that `T0` will always contain the number zero. That implies that a `JEQ` instruction with `T0` (like the one in the fourth line above) will always be an *unconditional* jump.

[21] Counter machines are a special type of *register machines*. See the corresponding Wikipedia article for information about the history of this type of abstract machine.

To run this program, do the following:

```
reg -r 35,7 Code/add.reg
```

Here, a comma-separated list of values with the `-r` option is used to fill the registers starting at `R1` with initial values. (All other registers used are initially set to zero.) The output of `reg` should be self-explanatory. The program also accepts the options `-h`, `-s` and `-p` we already know from `tm`. In addition to that, you can enter numbers in binary or hexadecimal format and, using `-b`, choose a basis for the display of the register contents. With the option `-m`, you can limit the amount of registers shown.

```
reg -r 0x20,0b1010 Code/add.reg
reg -r 32,34 -b 16 Code/add.reg
reg -r 20,22 -m 1 Code/add.reg
reg -r 20,22,42 -s0 Code/add.reg                    # note the warning
```

Exercise 11. Write a program that computes the same result as `add.reg` in `R1` without destroying the contents of `R2`. (Hint: Move the contents of `R2` into a temporary register and then use that to increase `R1` and at the same time restore `R2`.)

Exercise 12. Play with the program `Code/add2.reg` which computes the sum of `R2` and `R3` in `R1` while preserving the original values. Study its code and then write a similar program to multiply two numbers. (A solution can be found in `Code/mul2.reg`.)

The macro preprocessor

To make our life a bit easier for the next steps, we will now introduce a macro preprocessor. This is a simple program that doesn't know the instruction syntax for `tm` or `reg` or any other application; it just replaces text with other text using templates with parameters. In its most basic form, it works like this:

```
echo 'bad() {good}' > /tmp/macros
echo 'This is @bad' | mac /tmp/macros
```

Here a macro named `bad` was defined with its *body* (the template) being the text enclosed in curly brackets. If the program `mac` is called with a file containing this definition it reads text from `stdin` and writes it to `stdout` after replacing each occurrence of `@mac` (note the at sign) with the body.[22] You can also specify files with the `-i` and `-o` options which will then be used instead of `stdin` and `stdout`, respectively.

A macro with a *parameter* looks like this:

[22] To be more precise: each occurrence which is followed by whitespace or a line ending.

```
zero(dst) {
JEQ $dst,+3
DEC $dst
JEQ T0,-2
}
```

If the `mac` program with this macro loaded will come across a line like `@zero R2`,[23] it will replace it with the macro's body after first substituting `R2` (the so-called *argument*) for each `$dst` (note the dollar sign) in the body. Macros can have more than one parameter and should then of course be called with the same number of arguments. Parameters and arguments have to be separated by commas.

The preprocessor has a few more features which will be mentioned once we need them.

Simulating the counter machine with a Turing machine

We will now see that our counter machine can be simulated inside `tm2` (and thus also inside `tm`). The idea behind this is relatively easy as `reg` is pretty simple. The register contents will be encoded as binary numbers on the tape separated by empty cells. For example, if only the three registers `R1` to `R3` were used with the values 3, 42, and 10, the tape would look like this:

```
111-101010-1010
```

We now only need code to increase and decrease one of these numbers (which we already have), some "housekeeping" (shifting tape contents if numbers get longer or shorter, moving to a particular register on tape, initializing temporary registers, ...), and an implementation of `JEQ`. This code, in the form of macros, is in the file `Code/tm2.mac`. For example, a macro that always moves the head back to the tape's start looks like this:

```
toStart() {
**R+1 | 00L+0 | 11L+0 | --L+0
}
```

The `reg` application offers a `-t` option which will convert its list of instructions into a list of such macros which can then, after running through `mac`, be given to `tm2`:[24]

```
reg -r 0,3,5 -s0 Code/mul2.reg
reg -t Code/mul2.reg
```

[23] `mac` is designed to work line-oriented, so an invocation like this must be at the end of a line or followed by a comment starting with `#`.

[24] Again, the generated code is pretty inefficient and could easily be optimized so that the head doesn't have to move that much. But as I already said, that's not the point here.

```
reg -t Code/mul2.reg | mac Code/tm2.mac
reg -t Code/mul2.reg | mac Code/tm2.mac | tm2 -s0 -t 0-11-101
reg -t Code/mul2.reg | mac Code/tm2.mac | transform
reg -t Code/mul2.reg | mac Code/tm2.mac | transform \
                     | tm -s0 -b -t 0-11-101
```

The last line eventually shows how `tm` multiplies two numbers (encoded in binary form). Compare this with Exercise 5 and maybe run the program with `-s1` instead of `-s0` to see what has to be done to achieve this.

A better counter machine

Like `tm`, our first counter machine `reg` is too cumbersome to do anything meaningful with it. We will therefore switch to an improved version, called `reg2`, that offers a lot more commands. You can generate a list of them like so:

```
reg2 -l
```

The commands starting with `J` are variants of `JEQ` which can for example compare two numbers and branch accordingly. They can be used to mimic `if-then` constructs in typical programming languages. We also have real arithmetic, including exponentiation and modulo. For example, the program in `Code/gcd.reg2` computes the greatest common divisor of two numbers while the one in `Code/invmod.reg2` can compute modular multiplicative inverses:

```
reg2 -r 0,97,67 -s0 Code/invmod.reg2
reg2 -r 0,126,210 -s0 Code/gcd.reg2
```

You might want to study these files and also `Code/log2.reg2` to familiarize yourself with what `reg2` can do.

You might have wondered about the somewhat cryptic commands `CONS` and `CARCDR`. These are named after traditional Lisp functions and have the same purpose here: The *cons* of two numbers x and y is defined in `reg2` as $z = 2^x \cdot (2y+1)$. That's a clever way of encoding two numbers in one[25] as it is relatively easy to extract the individual values x and y from z. (That's what `CARCDR` does. It checks how often z can be divided by 2 to get x and then subtracts 1 from the quotient $z/2^x$ and divides the difference by 2 to get y.)

We can interpret the number z as the ordered pair (x, y). And once you have ordered pairs, you have arbitrary tuples as you can encode (a, b, c) as $(a, (b, c))$, (a, b, c, d) as $(a, (b, c, d))$, and so on. As you might have noticed, these are essentially linked lists "disguised" as numbers.

[25] Such a code is called a Gödel numbering after the Austrian mathematician Kurt Gödel.

Two small programs demonstrate how we will later use this feature to simulate the memory of a "normal" computer. `Code/list.reg2` accepts eight numbers and interprets them as address/value pairs.

```
reg2 -r 0,2,4,5,1,3,42,1,3 Code/list.reg2
```

The number which ends up in `R1` encodes a list with the four components $(2,4)$, $(5,1)$, $(3,42)$, and $(1,3)$. This is meant in the sense that for example the "RAM" cell with the address 3 holds the value 42. The program `Code/find.reg2` can extract such a value, given the address:

```
ARRAY=$(reg2 -r 0,2,4,5,1,3,42,1,3 Code/list.reg2 | tail -1 | \
        awk -F '[, ]' '{print $4}')     # store R1 in a shell variable
echo $ARRAY
reg2 -r 0,$ARRAY,3 Code/find.reg2
```

We are once again reminded that this is about *theoretical* computer science because the numbers get very large. This only works because our hypothetical counter machine can deal with natural numbers of any size. But it works!

More sophisticated macros

As you will have expected by now, our next step is to simulate `reg2` with `reg`. This will again be done with macros. We will use some features of `mac` we haven't seen so far. To demonstrate them let's look at the macro that implements `reg2`'s command `ZERO` to set a register to zero:

```
zero(dst) {
JEQ $dst,§end
DEC $dst
@jmp -2
§end:
}
```

The first feature is that macros can be *nested*: The body of the `zero` macro contains an invocation of another macro, `jmp`, which will also be expanded. This is done recursively so that no at sign remains once the preprocessor is finished.

The second thing that stands out is the section sign `§` in front of the label `end`. The reason for this is that macros are usually meant to be used more than once. Without the `§`, we would then end up with several different lines with the same label `end`. This is certainly not what we want (and the counter machine would complain anyway). Therefore, labels starting with `§` will automatically be replaced by numbered labels like `_lbl1`, `_lbl2`, and so on so that each one is unique.

To demonstrate the third feature, let's look at the macro to simulate the command ADDTO:

```
addTo(dst,src) {
@copy%1 T%1,$src
§add: JEQ T%1,§end
DEC T%1
INC $dst
@jmp%1 §add
§end:
}
```

What are the percent signs for? This macro uses the temporary register T1. But it also makes use of other macros like copy and jmp. What if these macros also use and maybe modify T1? That could certainly be a problem. Therefore, if we invoke, say, copy with @copy%1 instead of just @copy, we instruct mac to increase each number following a percent sign in the body of copy by 1. Should copy really use T1 (in the form of T%1), this would then be changed to T2 so that there's no conflicting use of the same register.

So, in addTo, the macro invocations copy%1 and jmp%1 make sure that these macros don't use T1 while the lines with T%1 prepare addTo itself to be called by other macros. If some other macro would invoke it as addTo%2, then in the body of addTo, the first line would for example become @copy%3 T3,$src.

This should have prepared you to understand the file Code/reg.mac which contains macros that "translate" all the new reg2 commands to the meager three commands understood by reg. reg2 has an option -t (similar to reg) that instructs it to convert its instruction into these macros:

```
reg2 -t Code/gcd.reg2
reg2 -t Code/gcd.reg2 | mac Code/reg.mac
reg2 -t Code/gcd.reg2 | mac Code/reg.mac | reg -r 0,15,25
```

We've just seen how the primitive counter machine reg computed the greatest common divisor of 15 and 25. We can even go further and translate this down to our very first Turing machine:

```
reg2 -t Code/gcd.reg2 | mac Code/reg.mac | reg -t | \
     mac Code/tm2.mac | transform | tm -s0 -b -t 0-1001-1100
```

You might want to do the same without the -s0 option to see how much work this is for tm although this is just about the numbers 9 and 12. Obviously, this translation business quickly becomes unfeasible even for moderately complex tasks. And the "RAM" examples from above with their very large numbers would already bring reg to its knees. As it can't do more than add or subtract 1, you won't live long enough to see the simulation come to an end! But as

I already said, this is all about what would be possible *in principle* without time and space constraints. And in that sense, a primitive Turing machine can simulate `reg2`.

The best counter machine

Like in the fairy tale *The Fisherman and His Wife*, we are not satisfied with what we have and want an even better counter machine. Our wish is granted by the application `reg3` which — I promise — will be the last step in our growing collection. Remember that the original goal was to simulate a real CPU. That's why `reg3` introduces several commands which mimic what a RISC-V 32I processor can do. You can see all of them like so:

```
reg3 -l
```

Several command names end in `32`. These commands are specifically meant to work with 32-bit (potentially signed) integers. And many of them implement bitwise operations. While these are easy to realize in hardware, they can become pretty convoluted when you only have arithmetic available. For example, to check whether the least significant bit of a number is set, you have to compute the remainder you'd get if you divided the number by 2. To check other bits, you'd first have to "shift" the number by dividing repeatedly by 2. And so on.

`reg3` also has commands with `LOAD` or `STORE` in their names. They implement some kind of "RAM" with 8-bit cells as well as a collection of 32-bit registers. We've seen above that this could be done (albeit clumsily) with linked lists. To see how the "RAM" in `reg3` can be used, have a look at the file `Code/era.reg3`. It contains code to compute values of the prime-counting function using the sieve of Eratosthenes:

```
reg3 -r 1000 Code/era.reg3
```

As you might have guessed, there are macros to translate all the new commands in `reg3` back to `reg2`:

```
reg3 -t Code/era.reg3
reg3 -t Code/era.reg3 | mac Code/reg2.mac
reg3 -t Code/era.reg3 | mac Code/reg2.mac | reg2 -s0 -r 10
```

Run this with `-s1` to see the insanely large numbers `reg2` needs for the linked lists. You should better *not* try much larger numbers than 10! But again, it can be done in principle and we have the means to translate `era.reg3` into a set of instructions for `tm`.

The final step

`reg3` finally enables us to write a RISC-V 32I simulation. The set of instructions which achieve this can be found in `Code/risc.reg3`. At not much more than 400 lines, this can still be

considered a fairly simple program. It mainly consists of one big loop which continuously pulls 32-bit words from RAM and interprets them as RV32I machine code instructions. This is done by dissecting the word into parts and going through a large case-by-case analysis. ("Is this `add`? Is this `xor`?")

To test `risc.reg3`, let's return to the very first C program, `hello.c`. When we compiled it using `buildRV32I`, two files were created. The first one was the ELF file `hello.elf` which we already used for QEMU. The second one is `hello.bin`. This is a binary file containing nothing but the machine code. Using the `-d` option we can load it into the "RAM" `reg3` simulates at the same address QEMU uses.

```
reg3 Code/risc.reg3 -s0 -d RISC/hello.bin
```

If the code in `risc.reg3` ends with a zero in `R1` that's a sign that everything worked as expected. But shouldn't we see a message? Well, in QEMU, we sent the message through the simulated UART. `reg3` can simulate such a device as well, but we have to turn it on with the `-u` option. We can also set `-m` to a negative value to completely suppress register output:

```
reg3 Code/risc.reg3 -s0 -m-1 -u -d RISC/hello.bin
```

Voilà! And `reg3` can also execute the little floating point program from above although `reg3` "knows" only integers (and not even negative ones).[26]

```
julia -e 'write("/tmp/input", Float32[1.74e18])'
reg3 Code/risc.reg3 -s0 -m-1 -u -d RISC/log.bin mem:/tmp/input
```

Finally, we can even run the RISC-V 32I Turing machine simulation within `reg3`:

```
tm -t IIIII-II Code/sub.tm -d /tmp/sub
reg3 Code/risc.reg3 -s0 -m-1 -u -d RISC/tm.bin code:/tmp/sub.code \
                                               tape:/tmp/sub.tape
```

This is where the circle closes. We are now able to simulate arbitrary Turing machines in `reg3`. And the code in `risc.reg3` can in theory be converted to a Turing machine we could run on `tm`.

> **Exercise 13**. Look at the code of `Code/risc.reg3`. Why would it not be a good idea to use our existing tools to translate this to `reg` code?

I hope the previous pages could convince you that the so-called Church-Turing thesis is correct. It claims that every intuitively calculable function can be computed with a Turing machine. This is not a mathematical theorem as the notion of being "intuitively calculable" cannot be rigorously defined. However, there's a general agreement among computer scientists

[26] `-d` accepts more than one file. The syntax is otherwise the same as that of `runQEMU`.

and mathematicians that if we know an algorithm to compute something, we can implement this algorithm on a computer. And we have just seen that everything that runs on a computer can in theory be translated to a Turing machine.

A programming language — or, more generally, a set of data-manipulation rules — is therefore called *Turing-complete* if it can simulate any Turing machine. In that sense, all modern programming languages are Turing-complete.[27] And none of them is more powerful than that. That would imply that a language could express computations a Turing machine can't perform. So far, nobody could come up with something like that.

Source code

Within the Docker container you will find the source code for all programs described in this document. Most of it is written in Julia and the code is distributed over several modules in directories like `~/TM`, `~/Reg2`, and so on. Note that the corresponding programs use precompiled images in order to decrease startup time. This means that they won't be affected by changes you make to their source code unless you remove the `-JCompile/...` option in their corresponding start scripts — named `tm.jl`, `reg2.jl`, and so on. (Programs like `tm`, without the `.jl` suffix, are just symbolic links to a Bash script `launch.sh` which in turn calls the Julia scripts.)

Prof. Dr. Edmund Weitz Hamburg, May 12, 2026

[27] With the caveat that real computers only have a finite amount of memory while the tape of a Turing machine is unbounded.