. . . A SMALL GROUP OF MEN AND WOMEN, LED BY JOHN VON NEUMANN AT THE INSTITUTE FOR ADVANCED STUDY IN PRINCETON, NEW JERSEY, WHO BUILT ONE OF THE FIRST COMPUTERS TO REALIZE ALAN TURING'S VISION OF A UNIVERSAL MACHINE. THEIR WORK WOULD BREAK THE DISTINCTION BETWEEN NUMBERS THAT *MEAN* THINGS AND NUMBERS THAT *DO* THINGS - AND OUR UNIVERSE WOULD NEVER BE THE SAME.

DESCRIPTIVE TEXT, TO TURING'S CATHEDRAL BY GEORGE DYSON

. . . THERE'S NOTHING NECESSARILY PHYSICAL OR EXPENSIVE OR EVEN SLOW IN THE PROCESS OF PARADIGM CHANGE. IN A SINGLE INDIVIDUAL IT CAN HAPPEN IN A MILLISECOND. ALL IT TAKES IS A CLICK IN THE MIND, A FALLING OF SCALES FROM EYES, A NEW WAY OF SEEING.

DONELLA H. MEADOWS - LEVERAGE POINTS: PLACES TO INTERVENE IN A SYSTEM

THE PROCESS OF PREPARING PROGRAMS FOR A DIGITAL COMPUTER IS ESPECIALLY ATTRACTIVE, NOT ONLY BECAUSE IT CAN BE ECONOMICALLY AND SCIENTIFICALLY REWARDING, BUT ALSO BECAUSE IT CAN BE AN AESTHETHIC EXPERIENCE MUCH LIKE COMPOSING POETRY OR MUSIC.

DONALD E. KNUTH - THE ART OF COMPUTER PROGRAMMING - VOLUME 1 - FUNDAMENTAL ALGORITHMS

OLA DAHL

INTO COMPUTERS

theintobooks.com

A Books with Views™ production

Typeset using Tufte LaTeX, tufte-latex.googlecode.com

Draft printing, February 9, 2017

Contents

List of Figures

List of Tables

Welcome

Do you want to create a computer? All by yourself?

Do you want to do it step-by-step, starting with a single bit and ending with a design that can run real programs, written in C?

Do you also want to see how the design can be done using different languages, such as VHDL and Verilog?

Then this might be a book for you. The book will show you, in a step-by-step manner, how a simple computer can be created. While doing so, it will also provide an introduction to how computers work, and how their main parts can be constructed, and put together into a functioning design.

We start with a simple building block that can store one bit, and we end with a computer that can run software that is compiled and linked using gcc.

We choose to implement a subset of a real computer architecture - the OR1K architecture[1]. In this way, we can convey the experience of building a real system, while at the same time - since we choose a suitable small subset - making the task small enough to be completed without a large implementation effort.

[1] http://openrisc.io/architecture

Using an already available architecture also allows us to use available tools, such as the OpenRISC GNU tool chain[2].

[2] http://opencores.org/or1k/OpenRISC_GNU_tool_chain

The book is designed as a Book with Views[3]. This means that there are common parts, covering the general aspects of computer design, but also specific parts, treating view-specific material. We have chosen the views as *languages*. This gives us a possibility to treat the topic of computer design using different languages, but still keeping the material contained in one book.

[3] http://bookswithviews.com

The book has the following views.

- SystemC/TLM
- VHDL
- Verilog

The view you are reading now is Verilog. The purpose of this view is to show how Verilog can be used to construct a computer that

implements a specific architecture.

Choosing a language

We choose to describe our computer using a *language*. In this way, we can have a textual representation of the computer, and we can use the textual representation as input to *software tools*, that will help us to simulate the behavior of our computer.

Verilog is a hardware description language. We use Verilog to describe our computer, and to simulate its functionality. It is also possible to use Verilog to actually synthesize a computer, in real hardware, for example in an FPGA[4].

Verilog is standardized by IEEE[5].

Information about Verilog can be found e.g. from Doulos[6].

[4] https://en.wikipedia.org/wiki/Field-programmable_gate_array

[5] http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1620780

[6] https://www.doulos.com/knowhow/verilog_designers_guide

Hello world

A simple example will get us started. We use a classical "Hello, world" example, which will do nothing meaningful except printing a text string.

The code for the example is shown in Figure 1.

The code for the example is from the Icarus Verilog User Guide[7]. We will start using Icarus Verilog in Section Getting some tools.

[7] http://iverilog.wikia.com/wiki/Getting_Started

Figure 1: A hello world example in Verilog.

```
module main;

initial
  begin
    $display("Hello, world");
    $finish;
  end

endmodule
```

The code in Figure 1 contains a *module*, which is named `main`.

An *initial block* is used, for the purpose of defining the behavior of the module. The block contains a statement for displaying a string, and a statement for finishing the simulation.

We remark that the code in Figure 1 generates an artificial, simulated behavior. It does not provide any code that can be used for synthesizing actual hardware.

You can read about Verilog in Wikipedia[8], and at other places, such as Doulos, who provides this Verilog Designer's Guide[9].

[8] https://en.wikipedia.org/wiki/Verilog

[9] https://www.doulos.com/knowhow/verilog_designers_guide

Getting some tools

We need some tools, in the form of software. We search for software that can be obtained without cost.

We use a Linux computer with Ubuntu 16.04, and a Mac computer with OS X El Capitan.

We decide to use Icarus Verilog[10].

[10] http://iverilog.icarus.com

We can install Icarus Verilog in Ubuntu, by doing

```
sudo apt-get install iverilog
```

We can install Icarus Verilog on Mac, by downloading its source code from this Icarus Verilog FTP repository[11].

[11] ftp://ftp.icarus.com/pub/eda/verilog/snapshots

The file `verilog-20150513.tar.gz` can be downloaded from the pre-v10 directory[12]. The source can then be extracted and built, using the commands

[12] ftp://ftp.icarus.com/pub/eda/verilog/snapshots/pre-v10

```
tar xvf verilog-20150513.tar.gz
cd verilog-20150513
./configure
make
sudo make install
```

Make it run

The code in Figure 1 can be compiled and run.

Assuming that the program is stored in a file `hello.v`, compiling can be done as

```
iverilog -o hello hello.v
```

The above command generates the file `hello`, which can be run, by doing

```
vvp hello
```

which results in the printout

```
Hello, world
```

Building a computer

We have chosen a language, to describe our computer. We have taken a first, tiny step, and we have seen how we can get hold of some tools.

Before we take our next steps towards building our computer, let's spend some time talking a bit of what we want to build.

Our goal is to create a computer[13]. A computer reads instructions

[13] https://en.wikipedia.org/wiki/Computer

from a memory. Each instruction is represented as a sequence of bits. The values of the bits determine the type of instruction, and sometimes also arguments that the instruction shall use. The allowed instructions, for a given computer, belong to the computer's instruction set[14].

[14] https://en.wikipedia.org/wiki/Instruction_set

Most computers have instructions for loading data from a memory, and storing data to a memory. Other common instructions are instructions for doing mathematical operations, such as addition and subtraction, and instructions for making decisions. The decisions can be based on evaluations of certain conditions, such as checking if a number is zero, or if a certain bit is set in a piece of data.

An instruction that has been read is decoded, meaning that the computer interprets the bits of the instruction, and then, depending on the values of the bits, takes different actions.

The action taken is determined by the instruction. As an example, an instruction for addition results in the actual addition of two numbers, and most often also the storing of the result of the addition.

As we move on, we start with a small building block, that can store only one bit. We then extend the building block, so that we can store larger pieces of information. At a certain stage in our development, we are ready to implement our first instruction.

Storing one bit

A bit can have the values 0 or 1. In a computer, these values are represented by a low and a high value of an electrical signal.

The value of a bit can be stored. This means that the value is remembered, as long as it is stored. While the value is stored, the value can be read, and used, for the purpose of performing different operations. Examples of operations could be to store the value somewhere, for example in memory, or using the value in an addition operation.

A D flip-flop

The value of a bit can be stored in a building block called D flip-flop[15].

[15] https://en.wikipedia.org/wiki/Flip-flop_(electronics)

A D flip-flop stores one bit of data. A new value can be stored when a clock signal changes value. A component which can change its stored value only when a clock signal changes is called a *synchronous* component.

A D flip-flop implementation in Verilog is shown in Figure 2.

The code in Figure 2 defines a module. The module has two inputs, called `clk` and `data_in`, and one output, called `data_out`.

The input variables `clk` and `data_in` are defined using the keyword `input` and the output variable `data_out` is defined using the keyword `output`.

The input variables `clk` and `data_in` are also defined using the keyword *wire*.

The module defines a *register variable* called `reg_value`. The variable `reg_value` is defined using the keyword `reg`.

The variable `reg_value` will contain the actual value stored in the D flip-flop.

The variable `reg_value` is called a *state variable*.

An *always block* is defined using the keyword `always`. Following the keyword `always` is an indication, using the word `posedge`, stating that the actions in the always block shall take place at every rising edge of the clock signal. We see that the only action taken is to assign the value of the input `data_in` to the state variable `reg_value`. This

```
module d_ff(clk, data_in, data_out);

   input clk;
   input data_in;
   output data_out;

   wire clk, data_in;

   reg  reg_value;

   always @(posedge clk)
     reg_value <= data_in;

   assign data_out = reg_value;

endmodule
```

Figure 2: A D flip-flop in Verilog.

assignment ensures that the state variable `reg_value` is updated at every rising edge of the clock.

An assignment of the variable `data_out` is done, outside of the always block. This assignment ensures that the output `data_out` has the same value as the current value of the state variable `reg_value`.

A testbench

The D flip-flop implementation in Figure 2 has inputs and outputs. An external module, referred to as a *testbench*, can be used for the purpose of generating input signals to the D flip-flop, and observing output signals from the D-flip-flop.

A Verilog testbench is shown in Figure 3.

The testbench in Figure 3 starts with a definition of the time scale, stating that nanoseconds (ns) will be used as the time unit.

A register variable called `clk` is defined. This variable represents the *clock signal*. The actual shape of the clock signal is defined by the line

```
always #2 clk = !clk;
```

The input signal to the D flip-flop is defined by the register variable `d_ff_data_in`. The values used for the input signal are defined in an *initial block*, as

```
initial begin
   #1 d_ff_data_in = 0;
```

Figure 3: A D flip-flop testbench in Verilog.

```
‘timescale 1ns / 1ns

module d_ff_tb;

   reg clk = 1;

   reg d_ff_data_in = 1;
   wire d_ff_data_out;

   initial begin
      $monitor("At time %t, data_in=%b, data_out=%b",
               $time, d_ff_data_in, d_ff_data_out);
      #16 $finish;
   end

   initial begin
      #1 d_ff_data_in = 0;
      #5 d_ff_data_in = 1;
      #3 d_ff_data_in = 0;
   end

   initial begin
      $dumpfile("d_ff_tb_wave.vcd");
      $dumpvars(0,d_ff_0);
   end

   always #2 clk = !clk;

   d_ff d_ff_0(clk, d_ff_data_in, d_ff_data_out);

endmodule
```

```
    #5 d_ff_data_in = 1;
    #3 d_ff_data_in = 0;
  end
```

The testbench in Figure 3 is a *behavioral model*. A behavioral model can be used in simulation, but can not be synthesized into a working digital system, for use in e.g. an FPGA or an ASIC.

Build and run

A system, containing the D flip-flop in Figure 2 and the testbench in Figure 3, can be analyzed and built using the command

```
iverilog -o d_ff_tb d_ff.v d_ff_tb.v
```

The simulation can be run by giving the command

```
vvp d_ff_tb
```

The resulting printout is shown in Figure 4.

```
VCD info: dumpfile d_ff_tb_wave.vcd opened for output.
At time                    0, data_in=1, data_out=1
At time                    1, data_in=0, data_out=1
At time                    4, data_in=0, data_out=0
At time                    6, data_in=1, data_out=0
At time                    8, data_in=1, data_out=1
At time                    9, data_in=0, data_out=1
At time                   12, data_in=0, data_out=0
```

Figure 4: Printout from running the testbench in Figure 3.

The printout in Figure 4 shows the values of `data_in` and `data_out` for a sequence of time instants. The time instants are defined by a `$monitor` statement inside an *initial block* in Figure 3, as

```
initial begin
   $monitor("At time %t, data_in=%b, data_out=%b",
            $time, d_ff_data_in, d_ff_data_out);
   #16 $finish;
end
```

with the effect that a printout is done whenever the time changes, or one of the variables `d_ff_data_in` or `d_ff_data_out` changes value. The changes for the variable `d_ff_data_in` are defined in an initial block in Figure 3 as

```
initial begin
   #1 d_ff_data_in = 0;
```

```
    #5 d_ff_data_in = 1;
    #3 d_ff_data_in = 0;
  end
```

The printuout in Figure 4 also contains a printout of the file name `d_ff_tb_wave.vcd`. This is a file where *waveform* data are stored. The display of waveforms is treated in Section Making waves.

Making waves

The testbench in Figure 3 generates printouts as shown in Figure 4. The printouts show values of digital signals[16], each having the value one or zero. We can represent these signals as waveforms, with the level of the waveform being one or zero. Thinking of the value one as a high voltage level, and the value zero as a low voltage level, we can think of the waveforms as representing actual voltages, in an actual digital system.

[16] https://en.wikipedia.org/wiki/Digital_signal

A waveform can be visualized using the GTKWave[17] program. We can download a GTKWave version for Mac[18], in the form of a zip-file that contains an executable GTKWave program. The GTKWave program can be started from a Mac Terminal, by giving the command `open` followed by the app file name of the program. As an example, I could start the program by doing

[17] http://gtkwave.sourceforge.net

[18] http://gtkwave.sourceforge.net/gtkwave.zip

```
open /Users/oladahl/prog/gtkwave/gtkwave.app
```

A GTKWave version for Ubuntu can be installed in Ubuntu, by giving the command

```
sudo apt-get install gtkwave
```

The program can then be started by giving the command `gtkwave`.

A waveform can be generated from Verilog by insertion of a `$dumpfile` statement, followed by a `$dumpvars` statement indicating from which module the waveform data shall be recorded. An example is seen in the testbench code in Figure 3.

Waveforms, generated from the testbench in Figure 3, is shown in Figure 5.

We see in Figure 5 how the waveforms correspond to the printouts shown in Figure Figure 4

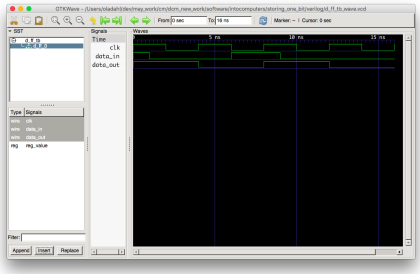

Figure 5: Waveforms, obtained from running the testbench in Figure 3.

Storing data in registers

When a computer executes instructions, it often needs intermidiate storage places. Reading instructions from memory, writing results back to memory. For example adding numbers, and writing back only when all numbers have been added. Then registers can be used, to hold the intermediate sum, while the calculation is ongoing. We can refer to such a row using the term register[19]. Another use of registers is for addressing. In this scenario, the value stored in the register is an address, addressing a part of the memory. One such register is holding an address pointing to the next instruction to be executed. This register is referred to as the program counter[20].

[19] https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Overall/register.html

[20] https://en.wikipedia.org/wiki/Program_counter

A register

A D flip-flop can store one bit. We can imagine a register as a row of D flip-flops, each storing one bit, with the possibility to load new values into all D flip-flops simultaneously.

A register implementation in Verilog is shown in Figure 6.

The code in Figure 6 defines a module. The module has two inputs, called `clk` and `data_in`, and one output, called `data_out`.

The module defines a *register variable* called `reg_value`, that will contain the actual value stored in the register.

An *always block* ensures that the state variable `reg_value` is updated at every rising edge of the clock.

An assignment of the variable `data_out` is done, outside of the always block. This assignment ensures that the output `data_out` has the same value as the current value of the state variable `reg_value`.

A testbench

An external module, referred to as a *testbench*, can be used for the purpose of generating input signals to, and observing output signals from, the register in Figure 6.

In the testbench module, we use a parameter, to specify the width of the register.

```
module n_bit_register(clk, data_in, data_out);

   parameter N = 8;

   input clk;
   input[N-1:0] data_in;
   output[N-1:0] data_out;

   wire   clk;
   wire [N-1:0]  data_in;

   reg [N-1:0]   reg_value;

   always @(posedge clk)
     reg_value <= data_in;

   assign data_out = reg_value;

endmodule
```

Figure 6: A register in Verilog.

The parameter is defined as a Verilog parameter, as

```
parameter N=4;
```

The clock signal is generated using a register variable named clk, defined as

```
reg clk = 1;
```

The actual clock generation is done in an always block, as

```
always #2
  clk = !clk;
```

The generation of input signals to the register in Figure 6 is done using
a Verilog process, as

```
always @(posedge clk)
  reg_data_in <= reg_data_in + 1;
```

The input signal and the output signal are defined as

```
reg [N-1:0] reg_data_in = 1'b1;
wire[N-1:0] reg_data_out;
```

The signals are used in the instantiation of the register, which is done as

```
n_bit_register #(.N(N)) reg_0(clk, reg_data_in, reg_data_out);
```

The reporting of the results is done in a process, as

```
initial begin
   $monitor("At time %t, data_in=%b, data_out=%b",
            $time, reg_data_in, reg_data_out);
   #16 $finish;
end
```

Build and Run

The register in Figure 6 and a testbench, with code as shown in in Section A testbench, can be built and run.

A makefile[21] can be created. The makefile can contain commands for building and running the register and the testbench.

A makefile is shown in Figure 7.

[21] https://en.wikipedia.org/wiki/Makefile

Figure 7: A makefile for building and running the register in Figure 6.

```
SOURCES := n_bit_register.v n_bit_register_tb.v

n_bit_register_tb: $(SOURCES)
iverilog -o $@ $^

.PHONY: clean

clean:
rm n_bit_register_tb
```

It can be seen, in the makefile in Figure 7, that the `iverilog` command is used, in the same way as described in Section Build and run in Chapter Storing one bit.

Assume the register is stored in a file named `n_bit_register.v`, and the testbench is stored in a file named `n_bit_register_tb.v`. Running the makefile, by giving the command `make` results in printouts, as

```
$ make
iverilog -o n_bit_register_tb n_bit_register.v n_bit_register_tb.v
```

A script file can be created, and used for running the simulated register and the testbench. Using a script file named `run.sh`, with contents as

```
#!/bin/bash

vvp n_bit_register_tb
```

```
$ ./run.sh
VCD info: dumpfile n_bit_register_tb_wave_verilog.vcd opened for output.
At time                    0, data_in=0001, data_out=0001
At time                    4, data_in=0010, data_out=0001
At time                    8, data_in=0011, data_out=0010
At time                   12, data_in=0100, data_out=0011
At time                   16, data_in=0101, data_out=0100
```

Figure 8: Printouts from a simulation of the register in Figure TBD.

for running the simulation, gives the result as shown in Figure 8.

We can generate waveforms, in the same way as described in Section TBD. The resulting waveform, for the register with printouts as shown above, is displayed in Figure 9.

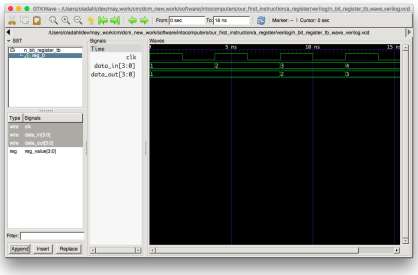

Figure 9: Waveforms from a simulation with printouts as shown in Figure TBD.

Our first instruction

A computer executes programs by following instructions. The instructions belong to an an instruction set. As mentioned in Chapter Welcome, we will use a subset of the OR1K instruction set[22] as the instruction set for our computer.

As a first step, we will try to build a computer with only one instruction. Although somewhat restricted, this computer will be able to

- Read instructions stored in a memory
- Decode each instruction
- Take action, based on the instruction read - in this case, it means that the computer will store a value in a register

We will start with deciding on a program to run our our computer. The program will be stored in a memory, and its instructions will be read, one by one, and actions will be taken.

[22] http://openrisc.io/architecture

A program

Our computer needs a program in order to run. Before creating the program, we select an instruction to use.

From the OpenRisc Architecture page[23] we can find the OpenRISC 100 Architecture Manual[24].

We look in the OpenRISC 100 Architecture Manual[25], and we find the instruction `l.movhi rD, K` on page 81. This instruction takes a 16-bit value K, and shifts it left[26] by 16-bits, and then places the resulting value in the register rD.

We see the instruction format for `l.movhi rD, K`, with its different fields. There are

- a 6-bit opcode (TBD, URL) field, in bits 31-26 (where bit 31 is the most significant byte (URL)), with the value 6
- a 5-bit register destination field D, in bits 25-21, with the value being the number of the register rD

[23] http://openrisc.io/architecture

[24] https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.1-rev0.pdf

[25] https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.1-rev0.pdf

[26] http://stackoverflow.com/questions/141525/what-are-bitwise-shift-bit-shift-operators-and-how

- 4 reserved bits, in bits 20-17
- a 1-bit opcode field, in bit 16, with the value 0
- a 16-bit field K, in bits 15-0, holding the value that shall be written into register rD

We can write the instruction, with the fields as described above, as a 32-bit binary word. This results in

```
000110DDDDD----0KKKKKKKKKKKKKKKK
```

The binary instruction format for `l.movhi rD, K` can also be seen in Section 17 of the OpenRISC 100 Architecture Manual[27].

[27] https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.1-rev0.pdf

Suppose we want to make a program that first puts the value 1 in r0, 2 in r1 and 3 in r2. The program should then put the value 0 in registers r0, r1, and r2. Such a program would then have the binary representation

```
00011000000----00000000000000001
00011000001----00000000000000010
00011000010----00000000000000011
00011000000----00000000000000000
00011000001----00000000000000000
00011000010----00000000000000000
```

Grouping the binary digits in groups of four gives

```
0001 1000 000- ---0 0000 0000 0000 0001
0001 1000 001- ---0 0000 0000 0000 0010
0001 1000 010- ---0 0000 0000 0000 0011
0001 1000 000- ---0 0000 0000 0000 0000
0001 1000 001- ---0 0000 0000 0000 0000
0001 1000 010- ---0 0000 0000 0000 0000
```

We can now convert the program to a representation where we use hexadecimal[28] numbers. This conversion results in the program

[28] https://en.wikipedia.org/wiki/Hexadecimal

```
18000001
18200002
18400003
18000000
18200000
18400000
```

Figure 10: A program using a l.movhi instruction for writing values into registers

Addressing a memory

We need to put our instructions in memory.

Design a memory where the program can be stored.

A memory implementation in Verilog is shown in Figure 11.

Figure 11: A memory in Verilog.

```
module memory(clk, write_enable, address, data_in, data_out);

   parameter address_width = 32;
   parameter data_width = 32;
   parameter size = 256;

   input clk;
   input write_enable;
   input [address_width-1:0] address;
   input [data_width-1:0] data_in;
   output[data_width-1:0] data_out;

   wire clk;
   wire write_enable;
   wire [address_width-1:0] address;
   wire [data_width-1:0] data_in;

   reg [data_width-1:0] memory [0:size-1];

   initial begin
     $readmemh("memory_contents.txt", memory);
   end

   always @(posedge clk) begin
     if (write_enable == 1)
       memory[address] <= data_in;
   end

   assign data_out = memory[address];

endmodule
```

Create a pc that reads addresses expressed in bytes. Meaning that it increments itself with four for each instruction read.

A program counter implementation in Verilog is shown in Figure 12.

Connect the pc and the memory into a design, so that when it runs, the program is read, and printed.

```
module pc(clk, pc_out);

   parameter pc_width = 32;

   input clk;
   output[pc_width-1:0] pc_out;

   wire clk;

   reg [pc_width-1:0] pc_value = 'b0;

   always @(posedge clk)
     pc_value <= pc_value + 4;

   assign pc_out = pc_value;

endmodule
```

Figure 12: A program counter in Verilog.

Decoding the instruction

We must interpret the instruction. We must take actions, in the form of writing parts of the instruction - the K value - into a register. We make a register bank with a destination register selection, and a data input, and a write enable (feels good to have that).

We take out the different pieces of the instruction, and feed the destination register and the sign-extender, from which we take the 32-bit value back to the register bank, so that when the clock comes, the value is written to the correct register. This sounds really cool! Perhaps we do not need a full program for this section. No, I do not think so, only code snippets from the final program, which is presented in total in the next section!

Running the program

Here we wire the pieces together, and create a functioning computer, albeit with only one instruction! More to come, continue reading!