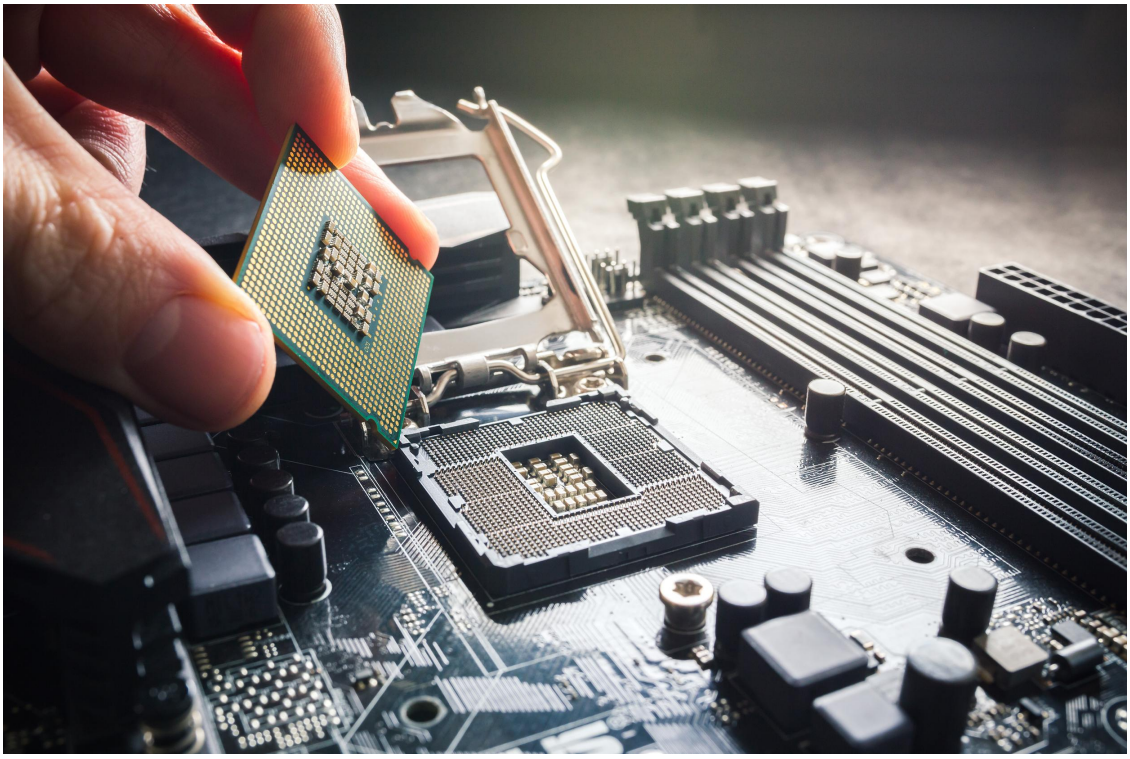
BFCPU 1.0

Joren Heit, Arthur Topal



Contents

1 Acknowledgements

Before diving into the book, I want to give out a special gratitude for: Joren Heit, for the invitation to take part in building the processor and for providing enough materials, tools, and components (see 3); And the school administration for providing a separate room in the building for storage and the development of the processor, and for the whiteboard used to make circuits, diagrams, and analyzing wrong behavior.

2 Preface

This documentation will encapsulate underlying principles of the processor based on brainf*ck programming language. Even though the documentation does not dive deeply in electronics or underlying science of computers, there are some recommendations for readers to provide the most satisfying reading process.

A reader should be familiar with:

- some elements of fundamental electronics such as capacitors, resistors, filters, voltage biasing, and basic electromagnetism;
- intermediate understanding of digital electronics and boolean algebra;
- comprehension of programming language(s), such as C/C++ or brainf*ck.

2.1 Useful resources

If the reader does not meet certain prerequisites, the following references may be helpful for acquaintance:

- "Digital Electronics by Morris Mano" for basic digital electronics and boolean algebra;
- "The Feynman Lectures on Physics Volume 2" for introductory electromagnetism;
- "Introduction to Electrodynamics by David J. Griffiths" for intermediate electrodynamics and applications;
- "learncpp.com" is a great learning resource for C++;
- for further resources see 12.2

3 Inventory

For the creation of the processor we used a vast inventory of tools and elements, like resistors, LEDs, and capacitors. However, other components are, mostly, digital chips created by different manufacturers and have also been utilized. In this section, we provide the reader with an approximate list of all tools and components utilized or used during the process.

3.1 Electronic components

- LEDs with different colors (white, yellow, red, blue, and green);
- Capacitors of a wide range (from 0.1 (nano Farad) up to 4000 (micro Farad));
- Resistors of a wide range (from 10 Ohm up to 1M Ohm);
- Seven-segment displays;
- Ribbon cables;
- Diodes;
- Voltage sources (batteries and AC to DC converters);
- Arduino Uno/Nano;
- ESM32;
- NAND (00 series);
- NOR (02 series);
- NOT (04 series);
- AND (08 series);
- OR (32 series);
- XOR (86 series);
- Seven-segment display decoders (48 series);
- JK Flip-flops (76 series);
- 4-bit count registers (74LS161);
- 4-bit registers (74LS173);
- 4-bit registers with up/down (74LS193);
- 8-bit shift registers (595 series);
- 8-bit bus tranceivers (245 series);
- 555 timers
- 3-to-8 demultiplexers (138 series);
- 4-bit NANDs (20 series);
- 4-bit NORs (21 series);
- 3-to-8 multiplexers (151 series);
- ROM
- RAM (AS6C4008);
- et cetera.

3.2 Tools and instrumentation

- wire cutters;
- pliers;
- multimeters;
- oscilloscope;
- probes;
- soldering machine;
- et cetera.

3.3 Digital tools

- Linux (Fedora);
- Emacs;
- Logisim;
- Inkscape;
- TeXstudio;
- Git/Github;
- Visual Studio Code;
- C++ compilers (g++), C++ build tools (CMake);
- et cetera.

4 Introduction

The Brainf*ck[1] (BF) programming language is an esoteric programming language that is basically impossible, or at least very unpractical, to actually write useful programs in. Even if you would become a very skilled programmer in this language, the resulting programs would be incredibly slow to execute. Despite this, many programmers have challenged themselves to write stunning pieces of code just for fun, or for the learning experience it offers. In doing so, it teaches us about computer architecture, compilers/interpreters, memory, pointers and much more. For more information on the language itself, see chapter 5.

The goal of this project is to build a computer that can actually run BF code natively. Normally, after having written some new piece of BF, the programmer must run this code in another program to either compile or interpret it to see whether it works. However, when looking more closely at the language, it seems a lot like an instruction set for a (not yet) existing processor. This is what we aim to do in this project: build that thing.

The aim is to build a Brainf*ck CPU without making use of any programmable chips. We will only use discrete logic and relatively simple integrated circuits like registers, buffers and (de)multiplexers. The computer will be built on breadboards, as it was inspired by Ben Eater's 8-bit breadboard computer [3]. This report will document the design and implementation of the BFCPU.

[1] The asterisk was inserted by the authors and is not part of the official name.

5 Brainf*ck

5.1 BF as a Language

Brainf*ck is often introduced as a programming language, which it is. However as we will soon see, it can also be viewed as an instruction set to a yet non-existent CPU. Just like any other programming language, it allows the programmer to write programs consisting of commands that are executed in order. There is a total of 8 commands available to the programmer, written as single characters: "`+-<>[].,`". Each of these commands corresponds to an operation on an array of memory or a pointer, pointing to some location within this memory. At the start of the program, every cell in memory is initialized to 0 and the pointer is pointing to the very first element (index 0, see Figure 1).

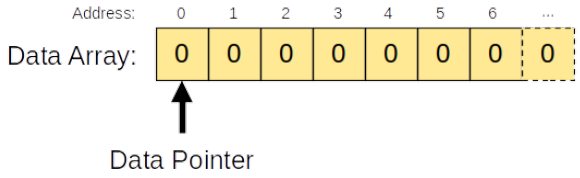


Figure 1: Initial state of the BF machine.

The commands then modify the contents of memory and the pointer as follows:

- `+` : add 1 to the current cell;
- `-` : subtract 1 from the current cell;
- `<` : move the pointer 1 cell to its left;
- `>` : move the pointer 1 cell to its right;
- `[` : if the current cell is nonzero, continue. Otherwise, skip to the matching closing `]`;
- `]` : if the current cell is zero, continue. Otherwise, loop back to its matching opening `[`;
- `.` : send the value in the current cell to the output;
- `,` : read a value from the input and store it into the current cell.

Although this might seem like a very limited set of instructions, it has been proven to be sufficient for performing any possible computation or program, also known as Turing-completeness [1]. The catch is that this requires an unbounded (or infinite) amount of memory, which is obviously impossible. However, the same caveat holds for traditional (von Neumann architecture) systems, so we should be safe to assume that BF is Turing complete for all practical purposes.

5.2 Architectures

5.2.1 Von Neumann

Modern computers are built according to the von Neumann architecture [2], which specifies a CPU (containing registers and an ALU), a single unit of memory and input/output devices (Figure 2). The registers of the CPU can be loaded with data from the memory unit and operated on by the ALU (Arithmetic and Logic Unit). Typical about this kind of architecture is the fact that not only data, but also the instructions (the program) are stored in memory. The program is therefore just as much part of the data as the data itself and can even be modified by itself.

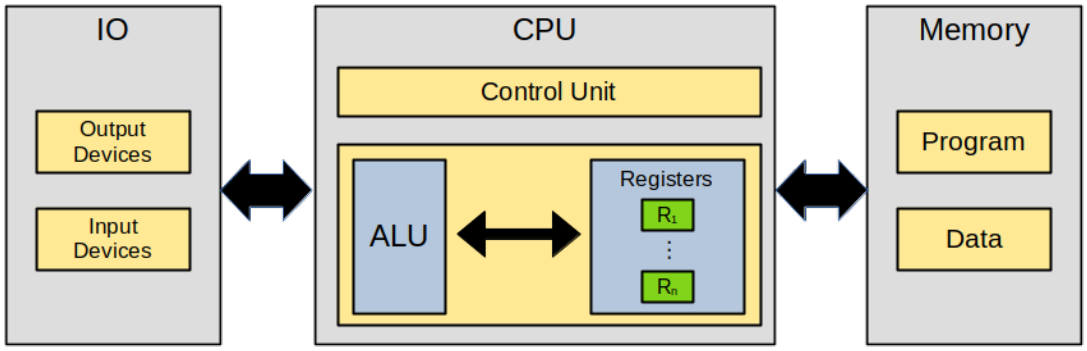


Figure 2: Schematic overview of the Von Neumann architecture.

5.2.2 Harvard

Unlike within the Von Neumann architecture, the Harvard architecture specifies two kinds of memory: program memory and data memory (Figure 3). The program memory contains only the instructions to be carried out and cannot be modified at runtime. Other than that, the architecture is similar to Von Neumann, in that it consists of a CPU (again containing registers and an ALU), memory (program and data) and input/output devices.

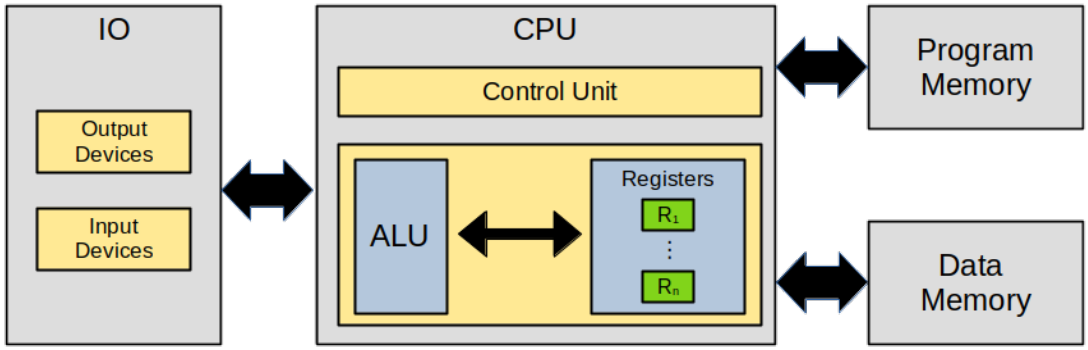


Figure 3: Schematic overview of the Harvard architecture.

5.2.3 BF Architecture

The architecture assumed by the BF language is similar to the Harvard architecture, in that the memory does not contain the program itself. This implies that the program is stored somewhere else and cannot be addressed by the pointer. The ALU is very limited and can only perform increment, decrement and comparison to zero.

5.3 BF as an Instruction Set

Instead of viewing BF as a language that needs to be compiled or interpreted on a traditional machine, it can also be seen as an instruction set to a processor, built according to the BF architecture described above. An instruction set of size 8 is truly tiny compared to more traditional instruction sets such as those implemented by modern processors or even microcontrollers and older 8-bit systems. Broadly speaking, Complex Instruction Set Computers (CISC) are designed to do as much work as possible in the least number of clock cycles, whereas Reduced Instruction Set Computers (RISC) focus on having a small instruction set with basic operations. For comparison, the x86 instruction set is massive with over 1500 instructions

implemented in hardware, whereas RISC processors only need to implement 50 to 100 instuctions. Even compared to RISC, the BF instruction set (BFISC from hereon) is tiny even compared to the smallest instruction sets in use today. This isn't necessarily a good thing; a smaller number of instructions simply means you need more of them to perform meaningful computations, which is reflected by the fact that complex BF programs are typically very large in size.

5.4 Brainfix

In 2013, one of the authors wrote a compiler, Brainfix, that takes a human-readable programming language (also called Brainfix) and compiles this into BF. This project was then abandoned and rebooted in late 2021 with a complete rewrite of the compiler. While working on this project, the idea to build an actual computer to go along with the compiler arose. The Github project page holds an extensive manual to the Brainfix language and compiler; see [4]. Chapter 11 goes into detail about some of the BF algorithms implemented by the compiler.

6 Architecture

In simple terms, a BF machine consists of an array of memory-cells, together with a pointer pointing to one of these cells. The pointer can move along the array while modifying its contents one step at a time. An example of this representation in some intermediate state is shown in Figure 4. Consider the BF program "`>>>>>+.`", applied to the initial conditions shown in the example. The pointer would take 5 steps to the right, landing on cell 9 which contains the number 41. It will then increment and output this value, displaying 42 on the screen (assuming a screen of some sort is used as the output device and it is displaying numbers directly rather than interpreting them as ASCII).

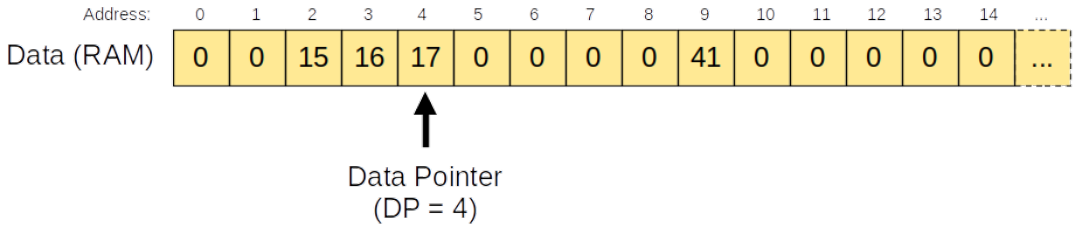


Figure 4: Example state of a BF machine.

In practice, a physical implementation of a machine like this needs many more components in addition to the RAM and Data Pointer in order to perform all the necessary operations. This chapter will list the modules that comprise the BF processor and give an overview of the way these modules function and communicate. The actual implementation on the logic/hardware level is described in section 8.

6.1 Overview

The processor consists of three basic building blocks: registers, memory and a control unit. The ALU is missing from this list because the only operations that it needs to perform are addition and subtraction of the value 1, which can be done directly at the register-level when using up/down binary counters like the 74LS193 integrated circuit. The program (a sequence of BF instructions) is stored into Read Only Memory (ROM), whereas the data is stored in Random Access Memory (RAM). Instructions (4-bits) are loaded from ROM into the instruction register (I), together with some flags that encode the state of the machine. Depending on the state and current instruction, the Control Unit sets the appropriate control signals for each of the modules in order for the system to perform the next computation. Figure 5 shows how each of the modules is communicating with other modules. In the sections below, each of these connections will be clarified further.

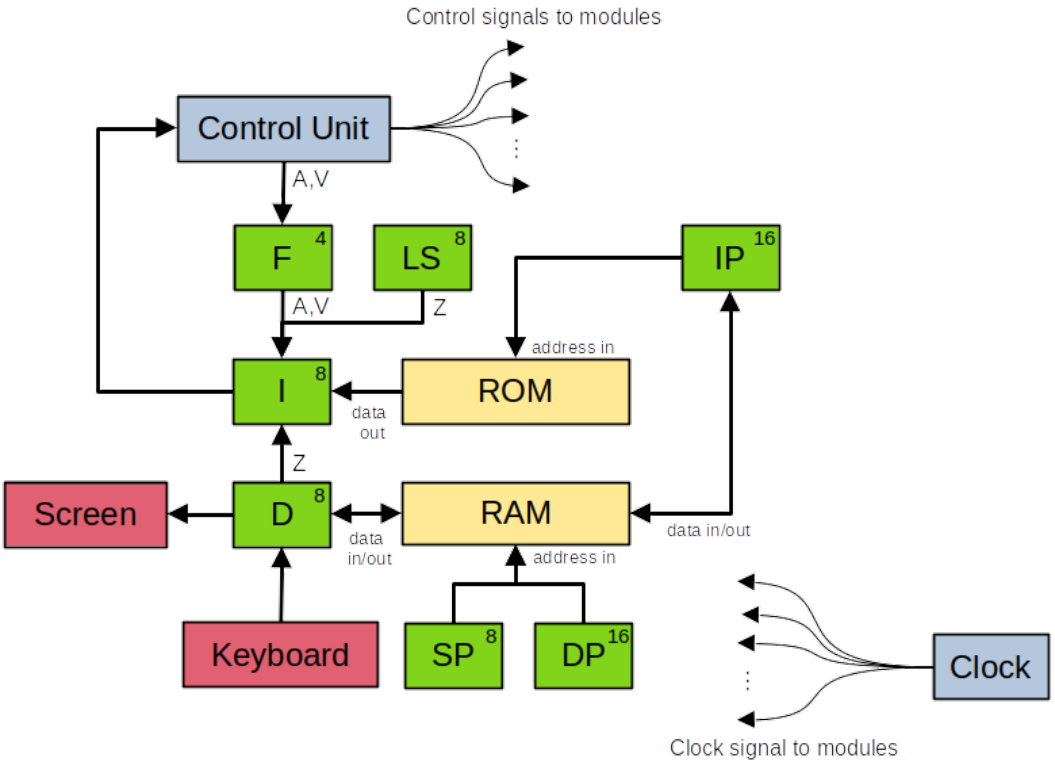


Figure 5: Connections between modules in the BF processor.

6.2 Instruction Pointer Register (IP)

The instruction pointer is a 16-bit value, kept in the IP register, which keeps track of the current instruction that is being executed. It points to a certain address in ROM (which stores the program) and is usually incremented after each instruction has finished executing, in order to move to the next instruction. However, when the processor encounters the `[`-instruction (and the loop is entered), it needs to store the current value of the IP somewhere in order to be able to jump back to this location. The value is therefore stored on the stack (the first part of RAM). When the matching `]`-instruction is encountered, this value is loaded back into the IP instead of simply incrementing the previous value. This has the effect of jumping back in the program, which is how loops are implemented in BF.

6.3 Stack Pointer (SP)

The stack is the first part of RAM (addresses 0x0000 - 0x00ff) which is reserved to keep track of addresses that might need to be jumped to. The stack-pointer (SP) is incremented whenever a new value is stored on the stack and decremented whenever a value is popped off the stack. In this implementation, the SP is an 8-bit value, which means that at most 256 different values can be stored onto the stack before it the SP overflows (wraps around back to 0) and starts overwriting previous values. This would happen if a BF program was loaded that has more than 256 nested `[]`-pairs. Although possible, it is very unlikely to happen for the simple programs we intend to run.

6.4 Instruction Register (I)

The instruction register is an 8-bit register that stores the current instruction, which was loaded from ROM according to the IP value. The BF-instruction itself is only 4 bits wide, which leaves another 4 bits for encoding the state of the machine, using flags. There are 4 flags that determine the state of the machine:

- Z(D): the zero-flag set by the D-register, indicating that there is currently a 0 stored in this register;

- Z(LS): the zero-flag set by the LS-register (see 6.7);
- A: the address-changed-flag, set by the control-unit, indicating that the previous instruction has changed the value of the data-pointer. When this flag is set, the value in the D-register does no longer correspond to the cell pointed to by the data-pointer;
- V: the value-changed-flag, set by the control-unit, indicating that the previous instruction has altered the value in the D-register. When this flag is set, the value in RAM is outdated and needs to be updated before moving the pointer to a different cell.

6.5 Data Register (D) and Data Pointer Register (DP)

The data-pointer corresponds to the pointer as specified in the BF-language. It points to some value in memory beyond the stack ($\geq$ 0x0100) and can be either incremented (moved right) or decremented (moved left) using the `>` and `<` instructions. Whenever the value pointed to by DP is modified by `+` or `-`, it is loaded into the D-register, where it can be modified before being stored back into RAM. This happens in conjunction with the flag-register, which dictates whether or not synchronization has to take place between RAM and D (see also section 6.6).

6.6 Flags and the Flag Register (F)

6.6.1 A and V

A naive and fail-safe way of synchronizing the contents of RAM with the contents of the D-register, is to perform the following sequence on each `+` or `-` instruction:

1. Load the current value (pointed to by DP) into D;
2. Modify the value;
3. Write the value back into RAM.

However, it is very common to have sequences of many repeating `+`'s or `-`'s and it would be a waste to keep writing the contents of D back to RAM, only to read them back into D during the very next instruction. This is why the Control Unit sets a flag whenever either the value of the DP has changed (A, the address-changed-flag) or the value in the D-register has changed (V, the value-changed-flag). Depending on the state of these flags, the Control Unit can determine whether it has to synchronize the RAM with the D-register, or not (yet). The Control Unit buffers these flags in the flag-register (F), which is loaded into the instruction register (I) together with the next instruction from ROM.

6.6.2 Z(D) and Z(LS)

In addition to the A and V flags, there are two flags which are not buffered in F, but can be directly loaded into I: the zero-flags (Z) from the data-register and loop-skip-register (see section 6.7). These flags are both used in the context of conditional jumps:

- Z(D) : When encountering either an opening `[`, the flag indicates whether or not to enter or skip the loop. On a closing `]`, the flag indicates whether to loop back or exit from the loop.
- Z(LS) : The zero-flag from the loop-skip register indicates whether or not we are currently in the process of skipping an entire piece of code. This happens when the current value is 0 when encountering an opening `[`. In this case, execution must resume after the matching closing `]`. Only when the LS has its zero-flag set, should the current instruction actually be executed.

6.7 Loop Skip Register (LS)

The Loop Skip (LS) register is a counter that indicates whether or not we're in the process of sckipping a loop. In BF, a loop (`[`) is only entered when the value currently pointed to is nonzero. In the case that it is zero, execution resumes beyond its matching `]`. When it is determined (by the Control Unit) that a loop must be skipped (based on the zero-flag set by the D-register), the LS register is incremented from 0 to 1. Subsequent instructions are then skipped until either another (nested) loop opening `[` or a closing `]` is encountered. On the former, the LS is incremented again while on the latter the LS is decremented. This has the effect that the LS becomes 0 again after the `]` that matches the original `[` which led to the skip. Normal execution occurs as soon as LS has become 0 again.

6.7.1 Preprocessing

The LS-register is solving a problem that can be avoided by preprocessing the BF code. We could have chosen to somehow include jump-locations in the program and render scanning during runtime (keeping track of the counter to determine when to resume execution) unnecessary. This would however introduce its own set of issues but would also require a fundamental modification to the BF instruction set. We therefore chose not to preprocess the program and require the computer to handle 'vanilla' BF.

6.8 Control Unit and Control Signals

Each of the aforementioned components/modules has one or more control inputs that determine what happens on the next clock cycle. For example, some register-modules can be told to either load a value from their input, increment or decrement the currently stored value, or do nothing at all. It is the Control Unit (CU) that supplies the appropriate control signals to each of the modules before the next clock pulse occurs. The implementation details of how this is done in hardware are discussed in section 8. The following sections will provide an overview of each of the control signals for each module. Section 7 will discuss the control sequences necessary to perform each BF instruction.

6.8.1 Control Unit

Even though the control unit is setting the control signals for all other modules, it has some control signals that it should be able to set for itself. Depending on the instruction that is being executed, it should write values to the flag register. The control signals to indicate that this is being done are called AE and VE respectively. Moreover, its internal instruction counter should be reset in the final cycle of each instruction. This is indicated with the CR (cycle reset) signal.

- AE: A-flag enable - outputs a high value into the A-bit of the flag register;
- VE: V-flag enable - outputs a high value into the V-bit of the flag register.
- CR: Cycle reset - reset the internal cycle counter to 0 to prepare for the next instruction.

6.8.2 Register Control Signals

Not all registers (green modules in Figure 5) have the exact same functionality. All of them store a value, but not all are required to count, let alone count in both directions. Table 1 shows which of the registers are able to perform certain actions. The control signals have been abbrivated as follows:

- EN: output enable - if this signal is unavailable, its output is either always available or not used at all;
- LD: load a value from the input;
- CE: count enable - increment the current value;
- DEC: if the CE signal is supplied, decrement instead of increment;

Register	#Bits	EN	LD	CE	DEC
D	8	x	x	x	x
I	8		x		
DP	16	x		x	x
SP	8	x		x	x
IP	16	x	x	x	
F	4		x		
LS	8			x	x

Table 1: Control signals available to each of the registers.

6.8.3 Memory (RAM)

The memory module has an address input and a set of I/O lines which can be used for either storing a new value into memory (input) or outputting the value stored at the current address. It is therefore sufficient to have only two control signals that go into the RAM module:

- OE: output enable - the value at the current address (supplied on the address lines) will be available at the output;
- WE: write enable - writes the value at the input to the current address.

6.8.4 Screen (Output)

The output module (which is assumed to be a screen) will be directly attached to the data-register and will display whatever is in there (on every clock pulse), when the PRE signal (print enable) is active.

- PRE: Print Enable - Display the contents of D on the next clock pulse.

6.8.5 Keyboard (Input)

The input device to the computer is assumed to be a keyboard of some sort, which contains a buffer from which some 8-bit value can be read. Immediately, a design choice has to be made: what happens if the buffer is empty? Either the program continues to run (immediate mode) or the program waits for something to appear in the buffer (buffered mode). The former is convenient for instance when playing games, while the latter is convenient for programs that require user input in order to continue meaningfully. Rather than deciding on either of these modes, we can just implement both. In the microcode table (Table 2, Section **??**) there are two variants of the `,` command listed (the other one denoted by `'`), each of which has a different binary representation. When assembling a BF program, we simply pick one or the other and the program should run in the appropriate mode: `,` for buffered and `'` for immediate mode.

The buffer of the module contains all zero's when nothing is present. When running in immediate mode, the programmer should programmatically determine if a value was read and how to act accordingly. In buffered-mode however, after reading from the buffer, the control unit will use the Z-flag from the D-register to determine whether or not to continue. As long as the value in D is zero, control flow will loop around, trying to get a value from the input buffer.

The only control signal going in to the input-module should therefore be the enable-signal, which makes the contents of the buffer available to the input of the D-register.

- EN: Output Enable - enables the contents of the input buffer to be loaded into D.

7 Control Sequences

By setting the control signals as described in Section 6.8 appropriately, the modules can work together to perform each of the BF instructions. Table 2 shows the control sequences that are executed per BF instruction (also known as microcode instructions). The Control Unit implements this as a lookup table in ROM, where the instruction, flags and cycle counter act as an address into this table and the control signal configuration is stored at this address in ROM (Figure 6).

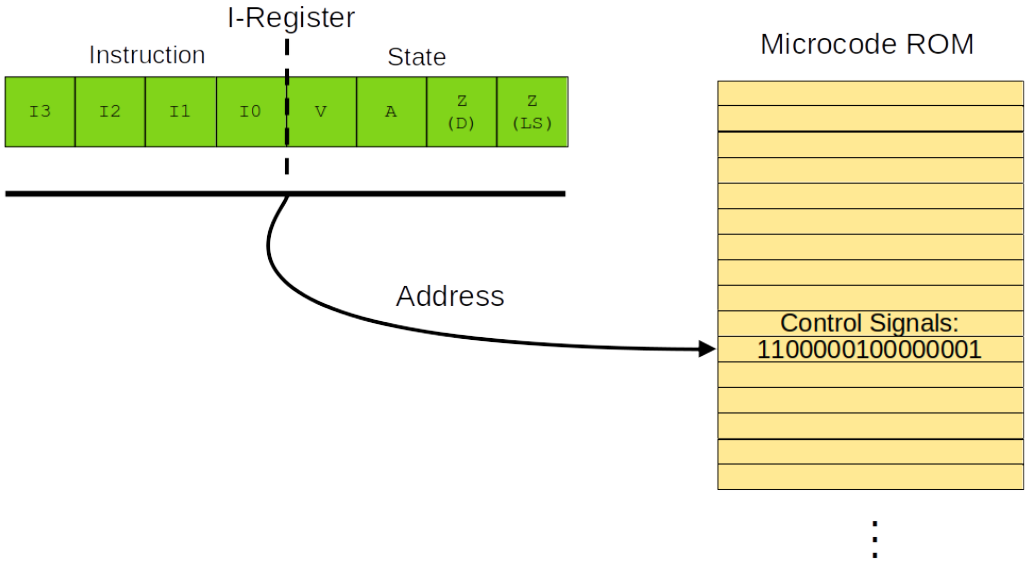


Figure 6: Decoding an instruction: the current instruction and state index the ROM, which returns the control signal configuration.

It is important to note that, because of the choice of driving all of the (counting) registers with a common interface (Section 8.1), only one register can be driven each clock cycle. That is, the CE and DEC signals can be applied to only one register in a clock cycle. Below we will go through each of the instructions in order to annotate the contents of Table 2.

7.1 Cycle 0

The first cycle of each instruction is identical. In fact, it's not really part of any instruction yet, because its purpose is to fetch the next instruction from ROM. In order to do so, the IP is enabled (address-in to the ROM module) and the resulting instruction is loaded into the I-register. Note that this also means that the A and V flags from the F register are loaded in (F is always enabled), together with the Z flags supplied by both the D and LS registers.

7.2 Modifying Data: + and -

The operations performed by the + command depend on the state of the system. If the A flag is not set (the address has not changed since the last instruction), this means that the value in D already corresponds to the current cell in memory. In that case, the CE signal for the D register is set for it to increment on the next clock pulse. In addition to this, the V-flag is set to indicate that the value in D has been changed, which is then loaded into the F register. In the next cycle, the CE is set for the IP register to load the next instruction and the cycle count of the CU is reset to 0.

On the other hand, when the A flag *was* set, we first need to fetch the new value from RAM by enabling the DP register and loading the resulting value into D. From hereon, the next set of control signals is identical

to that described above in the case where A was not set.

The control signals necessary to perform the `-` command are similar to those of the `+` command, the only difference being the DEC(D) signal to perform a subtraction rather than addition.

None of the actions above need to be performed when the Z(LS) flag is zero, which means that we're in the process of skipping a loop-block. In this case, we ignore the command and increment the IP immediately.

7.3 Moving the Pointer: < and >

Moving the datapointer around requires similar instructions compared to modifying the dataregister, the difference being that we increment or decrement the DP-register instead of the D-register. If the V-flag was set in the previous instruction, we need to write the updated value in the D-register back to RAM before moving the pointer. If not, we can immediately move the pointer. In either case do we need to set the A-flag in the F-register in order for the next instruction to take into account that the address has changed and the contents in the D-register are therefore not synchronized with the RAM. Like before, this instruction is ignored when the Z(LS) flag is zero.

7.4 Conditional Jumping: [and]

These are by far the most complicated instructions that require a lot of additional logic. Because the BFISC lacks a JMP-instruction where the argument holds the destination address, the computer has to store the address of the opening `[`-command in case it needs to loop back when the time comes. Also, if the loop is not entered, the LS-register is used to determine when execution resumes.

Loop Start: In the first scenario, where the D-register is up-to-date (A-flag not set) and its Z-flag is high, we can immediately conclude that this loop should be skipped over. Hence, the LS-register is incremented and the next instruction is loaded (to be ignored until the LS-register becomes 0 again).

In the second scenario, the A-flag is still not set but the Z-flag for the D-register is not set either, meaning that control should enter the loop. It takes 3 cycles to do so: increment the stack-pointer (cycle 1), write the current IP to this address on the stack (cycle 2) and move to the next instruction (cycle 3).

In scenario 3, the A-flag is set, which means that we should first load the new data from RAM into the D-register (cycle 1). After loading a new value into D, flags and cycle count are reset to 0 (without incrementing the instruction pointer). This means the same instruction is reloaded with updated flags on the next iteration, putting the system into either one of the states above.

In the case that we were already skipping code (Z(LS) low), we need to increment the LS-register once more to account for another pair of nested `[]`'s (cycle 1) and then continue to the next instruction (cycle 2).

Loop End: In the first scenario, which takes 2 cycles to execute, there is a known (synchronized) zero in the D-register. This means we can immediately choose to exit the loop. To do so, the stack-pointer is decremented (cycle 1) to point at the previous value on the stack. In cycle 2, the IP is incremented as usual.

If there is a known nonzero value in D (scenario 2: Z(D) is low), this means we must loop back to the IP stored on the top of the stack. This value is loaded into the IP-register by enabling the SP and RAM and setting the LD signal for the IP-register (cycle 1). On the second cycle, this new IP (pointing to a `[`) is incremented to re-enter the loop.

In the thrid scenario, the contents of D are not yet synchronized with the RAM (A-flag is set), so we first need to load it in. After loading the value into D, the flags and cycle counter are reset to put the system back into one of the previously defined states.

Finally, when already in the process of skipping a loop, the LS-register is decremented before moving to the next instruction.

7.5 Output: .

There are three states that need to be taking into account when implementing the output-command. In the first state, the value in D is already in sync with the RAM and can be sent to the output device immediately. The output of D is enabled and PRE signal for the screen is set, in addition to incrementing the IP to move

to the next instruction. When A was set, we first load the value from RAM into D before sending its contents to RAM. Lastly, when Z(LS) is low, this instruction can simply be skipped.

7.6 Input: , and '

As mentioned before in Section 6.8.5, the architecture implements two versions of the input command, buffered (`,`) and immediate (`'`). Because buffered inputs are more common in most (BF) programs other than game-like applications, this is the default mode (even though it is more complicated).

Buffered Mode: In this mode, control flow is stuck in a loop, waiting for something nonzero to appear in its D-register. A nonzero value means something was read from the input module, whereas a zero value indicates that the input buffer was empty. When the V-flag is not set, the contents of D can be overwritten immediately by loading the keyboard-buffer into D. After doing so, the instruction is reloaded to update the Z(D) flag, which is then used to decide between either moving to the next instruction (when Z(D) is 0) or looping back to cycle 0 (without incrementing the IP). When the V-flag *is* set however, the current value stored in D must first be written to RAM in order to synchronize. After having done so, the cycle counter is reset to 0 to put the system back into the state above where the keyboard-buffer can be read into the D-register. Of course, when Z(LS) is zero, we can simply skip all of this.

Immediate Mode: In this mode, we don't care what was loaded into D, even if there was nothing there in the keyboard buffer; it is up to the programmer to handle the case where no keys were pressed. This makes the implementation a lot easier, taking at most 2 cycles to complete. In the first case, the V-flag is not set, so we can simply overwrite the contents of D by loading in whatever is in the keyboard-buffer and move to the next instruction. In the second case, where V *is* set, we write the contents of D to RAM before copying the keyboard buffer into D. Again, when Z(LS) is zero, this instruction is skipped.

7.7 Non-BF instructions: `NOP` and `ERR`

For debugging purposes, two non-BF instructions have been included. The `NOP` instruction does nothing. It simply increments the IP and resets the cycle count to move to the next instruction. The `ERR` instruction is inserted in all non-reachable states. If for some reason a state occurs that maps to the `ERR` command, the clock will be halted and some indicator on the Control Unit should light up to let its users know that something has gone wrong.

7.8 Microcode table

Instr	V	A	Z(LS)	Z(D)	Cycle	Control Signals				
Any					0	EN(IP)	LD(I)			
+		0	1		1	CE(D)	VE(CU)	LD(F)		
		0	1		2	CE(IP)	CR(CU)			
		1	1		1	EN(DP)	LD(D)	OE(RAM)	LD(F)	CR(CU)
			0		1	CE(IP)	CR(CU)			
-		0	1		1	CE(D)	DEC(D)	VE(CU)	LD(F)	
		0	1		2	CE(IP)	CR(CU)			
		1	1		1	EN(DP)	LD(D)	OE(RAM)	LD(F)	CR(CU)
			0		1	CE(IP)	CR(CU)			
>	0		1		1	CE(DP)	AE(CU)	LD(F)		
	0		1		1	CE(IP)	CR(CU)			
	1		1		1	EN(D)	EN(DP)	WE(RAM)	LD(F)	CR(CU)
			0		1	CE(IP)	CR(CU)			
<	0		1		1	CE(DP)	DEC(DP)	AE(CU)	LD(F)	
	0		1		2	CE(IP)	CR(CU)			
	1		1		1	EN(D)	EN(DP)	WE(RAM)	LD(F)	CR(CU)

		0		1	CE(IP)	CR(CU)				
[	0	1	1	1	CE(LS)					
	0	1	1	2	CE(IP)	CR(CU)				
	0	1	0	1	CE(SP)					
	0	1	0	2	WE(RAM)	EN(SP)	EN(IP)			
	0	1	0	3	CE(IP)	CR(CU)				
	1	1		1	EN(DP)	LD(D)	OE(RAM)			
	1	1		2	LD(F)	CR(CU)				
		0		1	CE(LS)					
		0		2	CE(IP)	CR(CU)				
]	0	1	1	1	CE(SP)	DEC(SP)				
	0	1	1	2	CE(IP)	CR(CU)				
	0	1	0	1	EN(SP)	EN(RAM)	LD(IP)			
	0	1	0	2	CE(IP)	CR(CU)				
	1	1		1	EN(DP)	EN(RAM)	LD(D)			
	1	1		2	LD(F)	CR(CU)				
		0		1	CE(LS)	DEC(LS)				
		0		2	CE(IP)	CR(CU)				
.	0	1		1	PRE(SCR)	EN(D)	CE(IP)	CR(CU)		
	1	1		1	EN(DP)	EN(RAM)	LD(D)			
	1	1		2	PRE(SCR)	EN(D)	CE(IP)	CR(CU)		
		0		1	CE(IP)	CR(CU)				
,	0	1		1	EN(KB)	LD(D)				
	0	1		2	EN(IP)	LD(I)				
	0	1	0	3	VE(CU)	LD(F)	CE(IP)	CR(CU)		
	0	1	1	3	CR(CU)					
	1	1		1	EN(D)	EN(DP)	WE(RAM)			
	1	1		2	LD(F)	CR(CU)				
		0		1	CE(IP)	CR(CU)				
’	0	1		1	EN(KB)	LD(D)	VE(CU)	LD(F)	CE(IP)	CR(CU)
	1	1		1	EN(D)	EN(DP)	WE(RAM)			
	1	1		2	EN(KB)	LD(D)	VE(CU)	LD(F)	CE(IP)	CR(CU)
		0		1	CE(IP)	CR(CU)				
NOP				1	CE(IP)	CR(CU)				
ERR					ERR(CU)	HLT(CLC)				

Table 2: Control signals for each of the BF instructions in different scenario's, depending on the state flags. Note that in order to distinguish between the two input modes, the regular comma (`,`) and the apostrophe (`’`) are used for buffered and immediate inputs respectively. See also Section 6.8.5.

8 Implementation

8.1 Registers

This section elucidates the intricate facets concerning the register-module implementation. It delves into the architectural nuances encompassing the driver module, data-pointer register, data structures, stack functionality, loop-skip mechanisms, instruction parsing, instruction-pointer management, and the flag register. By the conclusion of this chapter, a comprehensive overview integrating these components will be presented, encapsulating the holistic framework of the register-module.

8.1.1 Overview

Within the BFCPU framework, the ALU was intentionally omitted due to the constrained mathematical operations inherent in brainfck language, limited to singular addition and subtraction by one. To streamline complexity, our approach involved the utilization of 74LS193 and 74LS161 integrated chips, enabling the creation of discrete registers. These chips offer both memory storage capabilities and the ability to increment values. Moreover, the 74LS193 chip encompasses the added functionality of decrementing stored values, although such functionality remains unnecessary for the F-R component. Hence, the choice of 74LS** was deliberate, selected for its singular provision of 4-bit value storage, aligning precisely with the requirements of the system.

8.1.2 Common INC-DEC functionality problem

The Register module's intricate configuration involves numerous INC and DEC pins, necessitating manipulation by the control unit to modify register values. To streamline and unify this process, the Driver module acts as a centralized interface for all registers within the entire register-module structure. The fundamental concept underlying the Driver module revolves around consolidating multiple INC and DEC pins into a single pair, accompanied by a CLK (clock) input.

When both INC and DEC signals maintain identical voltage states (either low or high), the selected register operates in a mode referred to as "none" during the rising edge of the CLK. This signifies that the register's value remains unchanged—neither incremented nor decremented. Conversely, when the digital value of the INC-DEC pair registers as 01 or 10, the corresponding selected register experiences either an increment or a decrement in its value.

The output generated by the Driver module, in the form of the UP-DOWN signal pair, directs the appropriate action to the designated register. Notably, for registers utilizing the 74LS161 chip that exclusively permits incrementing, only an UP signal is necessary.

To facilitate register selection, a selecting interface is imperative to designate the specific register under manipulation. The inner circuitry (refer to figure 7) adeptly fulfills this function. The accompanying figure provides a broad overview of the register-module, showcasing the existence of three select pins capable of establishing eight distinct paths. This suffices, given the presence of fewer than eight unique registers within the system's architecture.

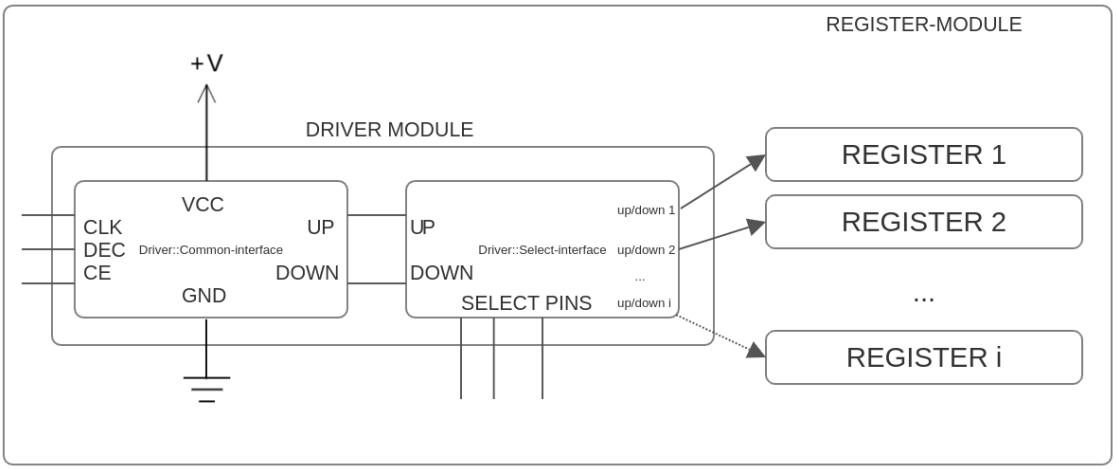


Figure 7: Register-module Overview

8.1.3 74LS193 Chaining

The utilization of 74LS193 (or 74LS161) chips in the construction of registers provides both storage capacity and INC-DEC functionality. However, a singular 74LS193 chip inherently provides storage for 4 bits, which falls short of the 8 or 16 bits necessary for the registers' intended functionality. Fortunately, the 74LS193 chip offers a chaining capability, mitigating this limitation.

Through a chaining configuration wherein the Carry-Out (CO) of one chip connects to the Up (UP) pin of another, and the Borrow Out (BO) links to the Down (DOWN) pin, an expansion of available bits becomes feasible (refer to figure 8). This method effectively extends the storage capacity, enabling the aggregation of multiple chips to accommodate the required bit sizes. This chaining technique stands as a foundational element implemented across registers demanding more than 4 bits, ensuring their requisite storage depth and functionality.

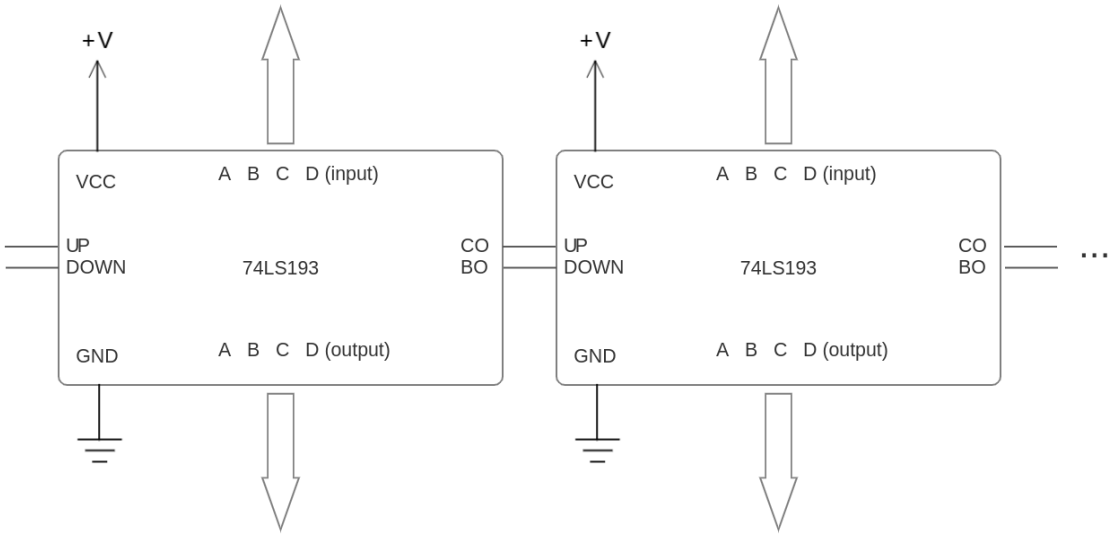


Figure 8: Register-module 74LS193 chaining

8.1.4 Circuit: Driver Module

In the computer's initial design phase, various concepts were explored for implementing this module. Initially, the demux-circuit was incorporated; however, it exhibited partial functionality, encountering persistent issues

such as potential propagation delays and compatibility concerns with the 74LS193 chip. Resolving these issues in a sustainable and efficient manner posed considerable challenges.

Consequently, the decision was made to discard the initial driver module design (refer to Appendix A(12.1) for an in-depth exploration of the former circuitry) in favor of developing an alternative circuit. This revised circuit now stands as the official and integral component of the computer's architecture, addressing the shortcomings and inefficiencies encountered in the prior design iteration.

The concept underlying the development of this stable driver module is rather straightforward:

- Configure UP-DOWN[2] to a HIGH state when both INC and DEC signals are either high or low simultaneously.
- Generate a pulse on the UP signal while maintaining the DOWN signal high when INC is high and DEC is low.
- Generate a pulse on the DOWN signal while maintaining the UP signal high when INC is low and DEC is high.

The practical implementation details are visually represented in Figure 9, accompanied by the following corresponding solutions:

- Common Interface:
  - Utilize
    $$CLK + (INC \oplus DEC)$$
    to prevent pulsing when INC and DEC have identical values.
  - Connect the resultant signal with the inverted INC-DEC into a NAND gate to ensure favorable behavior when INC and DEC have different values.
  - Integration of additional inverters is imperative to ensure compatibility with both the registers and the select interface[3].
- Select Interface:
  - Establish connection of the UP-DOWN pair to the 3-to-8 demultiplexer, facilitating the routing of this pair to the input of the selected register.

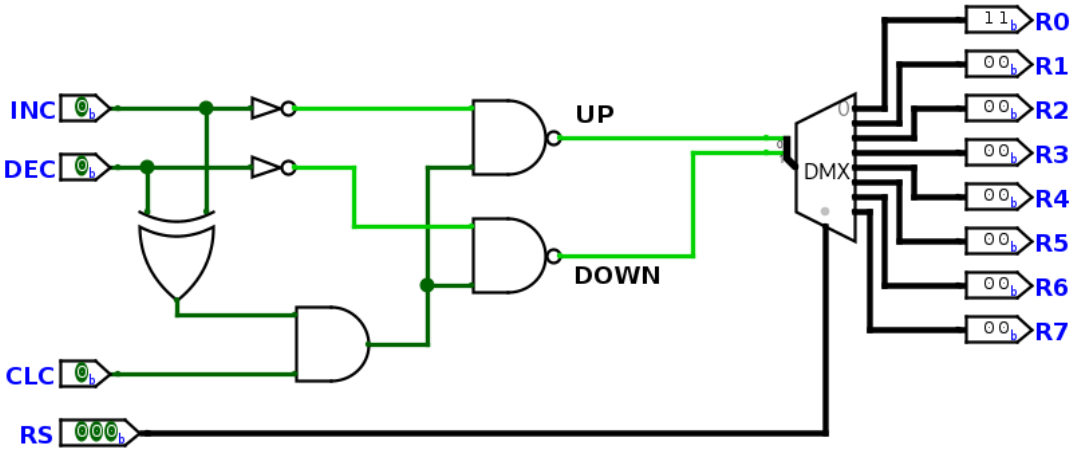


Figure 9: The Actual Driver Module Implementation

[2]The output of the driver module
[3]The 3-to-8 demultiplexer used inverts the output

8.1.5 Circuit: Data Register

The data register functions as an 8-bit valued flip-flop, equipped with both increment and decrement functionalities, which led to the adoption of the 74LS193 chip. An integral feature of the data register is its generation of a zero flag (Z), indicating a value of 1 exclusively when the stored value within the register is 0. The implementation of this particular feature involves the use of OR gates and direct connections to the stored bits (refer to Figure 10 for a visual representation).

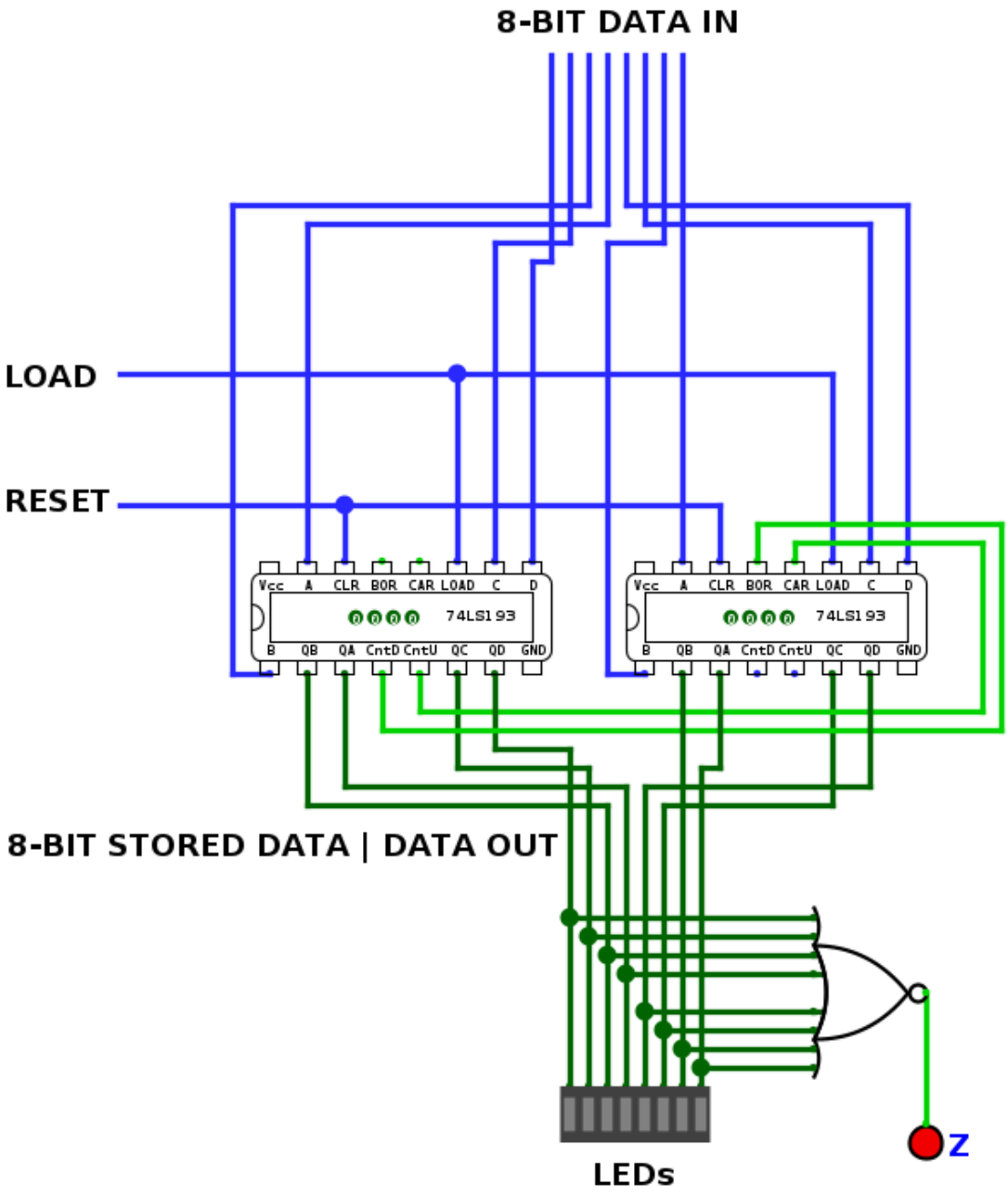


Figure 10: Data, and Loop-Skip Register Implementation (8.1.8)

8.1.6 Circuit: Data-Pointer Register

The Data-Pointer register operates as a 16-bit valued flip-flop equipped with both increment and decrement functionalities, consequently utilizing the 74LS193 chip. While the Data-Pointer register itself holds no distinct exceptional properties, its output (the stored value) maintains a connection to the bus transceiver, a characteristic shared among other registers as well[4]. Refer to Figure 11 for an illustrative depiction of its implementation.

8.1.7 Circuit: Stack-Pointer Register

The Stack-Pointer register operates as a 8-bit valued flip-flop equipped with both increment and decrement operations. Comparing with the DP-Reg, the SP-Reg could have been implemented in the same way. Thus, instead of using four 74LS193 chips for DP-Reg, only two chips were sufficient.

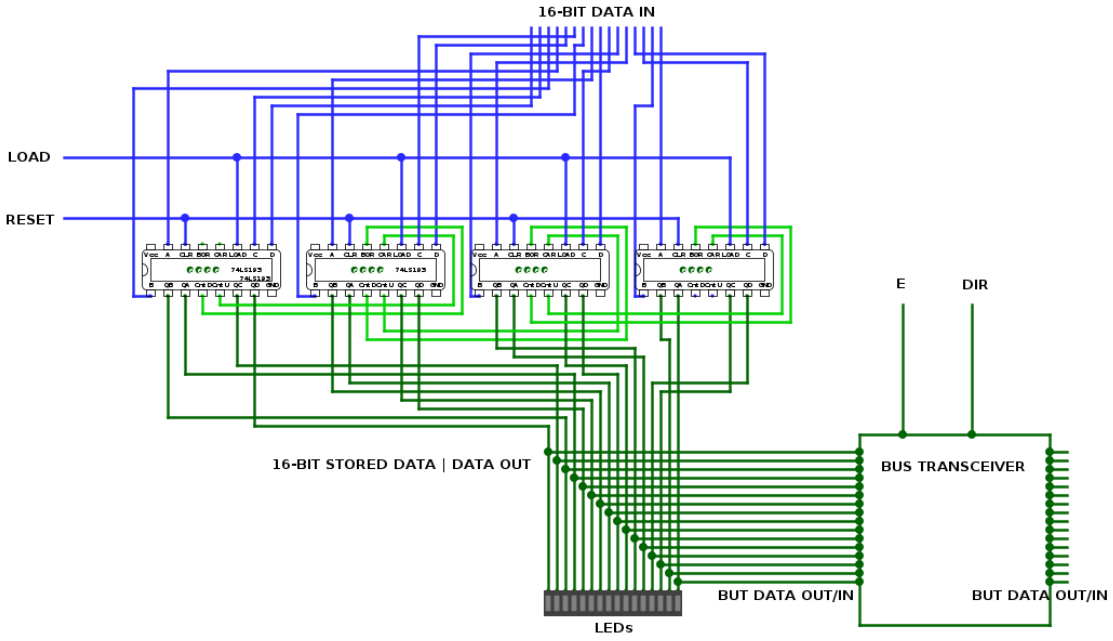


Figure 11: Data-Pointer Register Implementation

8.1.8 Circuit: Loop-Skip Register

At its core circuitry, the loop-skip register (LS-R) shares an identical implementation with the data register (D-R): both possess an 8-bit storage capacity, facilitate increment and decrement operations, and include a loop-flag functionality[5]. For an illustration of this implementation, please refer to Figure 10.

8.1.9 Circuit: Instruction Register

As stated above for Loop-Skip Registers, the instruction register shares an identical implementation with the data register: both possess an 8-bit storage capacity. However, an internal circuitry contains 74LS173 chips instead of 74LS193. The underlying reason is derived from unnecessary decrement functionality [6]. We will dive deeper in 10.

[4]The implementation of the bus transceiver involves a straightforward approach: connecting only outputs alongside the DIR and CE (chip enable) pins as inputs, employing the 74LS245 chips

[5]The loop-flag mirrors the zero-flag in the data register, thus is implemented in a similar fashion as in the data register

[6]Instruction Register stores current operation machine code, which can only be loaded from the ROM

8.1.10 Circuit: Instruction-Pointer Register

The instruction-pointer register (IP-R) shares its implementation framework with the data-pointer register. However, as the IP-R specifically denotes the machine code's location stored in the ROM, its sole requirement is for increment functionality. The procedural nature of brainf*ck dictates sequential code processing. Consequently, rather than utilizing the 74LS193, the choice fell upon the 74LS161 chip for its capacity to exclusively handle increments.

8.1.11 Circuit: Flag Register

The Flag Register is designed to manage precisely two bits, functioning as a 2-bit latch. Notably, our available inventory comprises chips configured for 4-bit operations, hence, two additional bits within the Flag Register remain reserved for potential future modifications. For this purpose, the LS74173 chip was selected to serve as the Flag Register.

8.2 Memory

8.2.1 RAM

In our architecture, we utilized 16-bit address, 8-bit word RAM chip (AS6C4008). Important to note, because some cells will need a bigger value than 8-bit, two RAMs are chained together with addresses and control pins connected. First 256 words with address from 0 to 255 store stack values pointed by the stack pointer. All other words are data in individual BF-cells pointed by data-pointer. Therefore, it was necessary to combine DP-Reg (see 8.1.6) and SP-Reg (see 8.1.7). The problem was that we cannot hard-connect the output of two pointers directly to the address lines of the RAM chip since it would cause voltage interference, and other hard to debug, impaired functionality. The solution was to use several 8-bit bus transceivers (see 12).

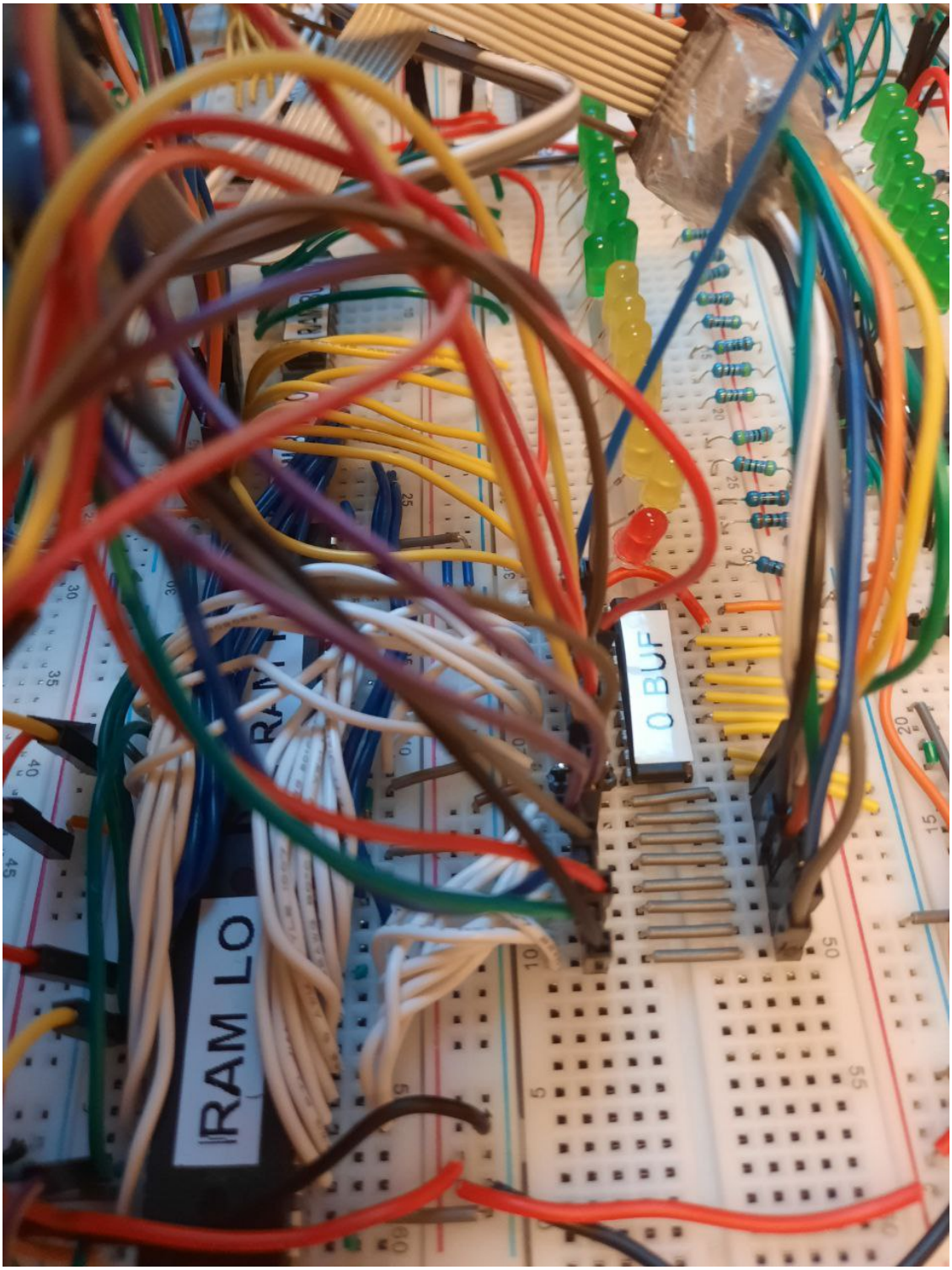


Figure 12: RAM and 8-bit bus transceivers

Firstly, let us explore the reasoning how to split two pointers. SP-Reg has 8 bits whereas DP-Reg has 16 bits. Suppose we connect DP-Reg directly to the address lines of the RAM chip. Then, add SP-Reg's 8-bits of output to the first eight address lines of the RAM. In this configuration, if both registers output voltage simultaneously, voltage interference happens as high and low bias equals high bias. The first step is

to connect the output of both pointers to the bus transceivers (from now on we use the abbreviation nBT8 for n 8-bit bus transceivers): SP-Reg—1BT8, and DP-Reg—2BT8. Control Unit can either activate a BT8 or disable for the pointer registers. Control unit must not activate both BT8s at the same time to prevent voltage interference.

From the first sight, it may be reasonable to state that this is a complete solution. However, the problem arises when we try to output SP-Reg: first 8-bit address lines have correct input from the register while other 8-bit address lines (remember that the RAM has sixteen address lines) are noise, as they are connected to the disabled BT8 of the DP-Reg. Therefore, we required to ground these 'higher,' disabled 8-bits of address lines. Another bus was introduced, right in the vicinity of the RAM with all the values connected to the ground. We call it the GND-Reg. The necessity to introduce a new bus rather than connect the values to the ground was interference with DP-Reg input - we do not want higher 8-bits to be always at the low potential. On this point, address lines are properly configured. The complete picture you can see in 13.

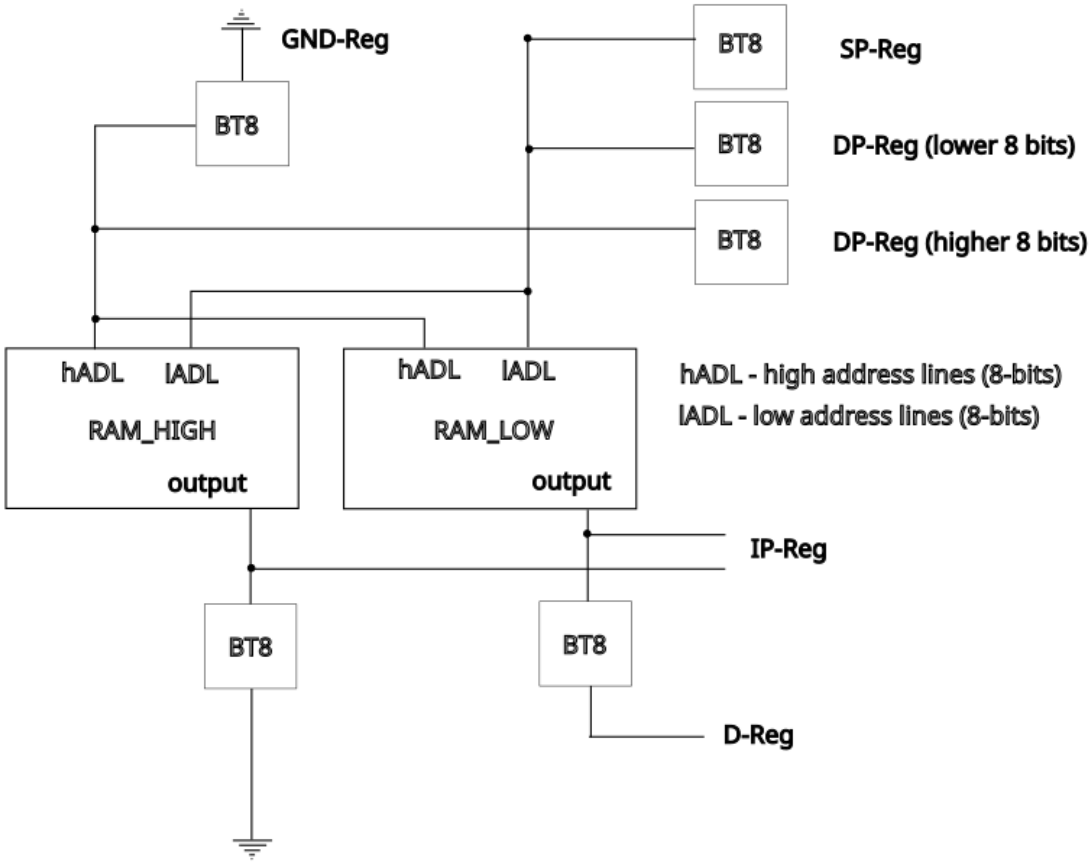


Figure 13: Encapsulated overview of RAM

Now, we can shed light on the output of the RAM. The chip that we are using have three control pins: Write-Enable (WE), Output-Enable (OE), and Chip-Enable (CE). The map of possible combinations and corresponding actions you can see in the official data-sheet for AS6C4008. Depending on the combination of WE, OE, and CE, BT8s of SP-Reg, DP-Reg, and GND-Reg must be configured appropriately. In particular, its direction and output-enable pin. This was done by several elongated wires that you can see in figure 14.

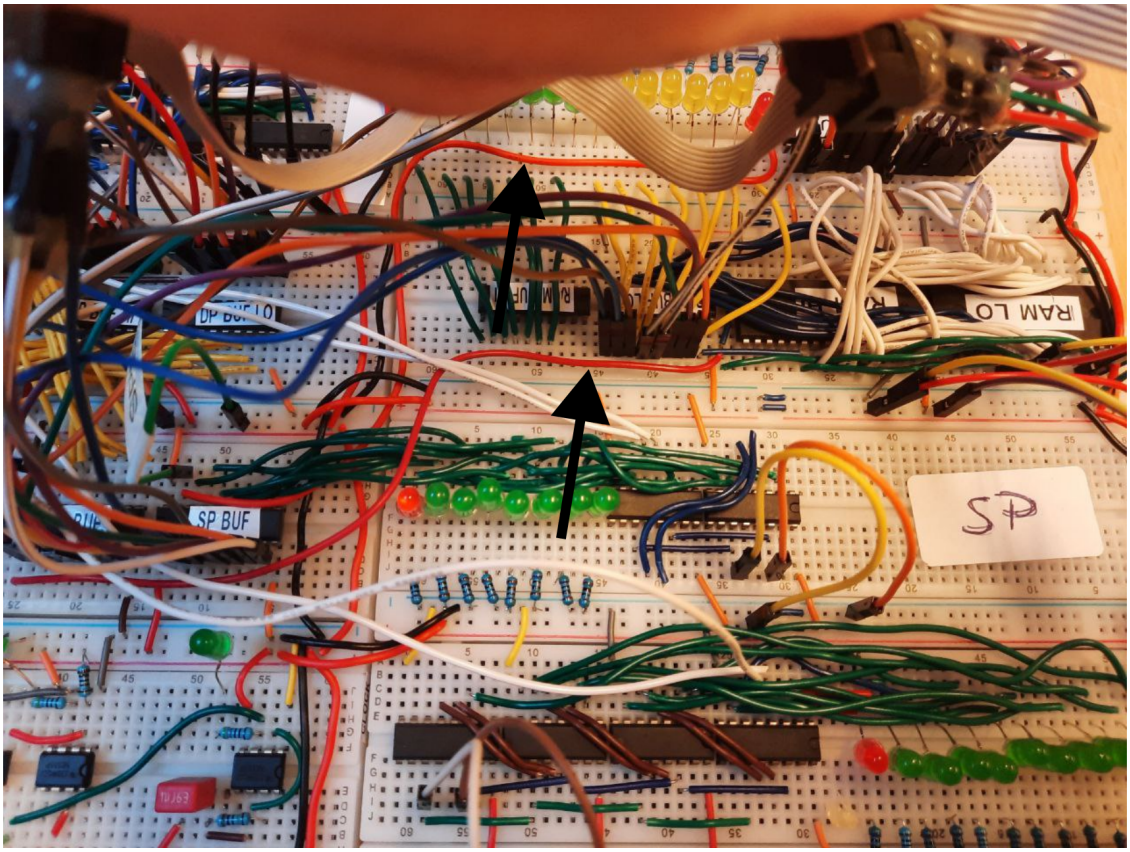


Figure 14: Elongated wires of RAM

When RAM outputs the data, it must be stored in the D-Reg and IP-Reg. However, since the two RAMs output 16-bit value, D-Reg is not able to store all 16-bits of information. Therefore, we divided 16-bit RAM output in higher and lower 8-bit values (HV AND LV for compactness). HV and LV are connected to two BT8s. The principle is actually the same as we dealt with address lines. One BT8 is connected to ground, the second to D-Reg, and, finally, both are connected to the IP-Reg for loading. Proper configuration of BT8's direction and output-enable pins is necessary, so proper wires have been added[7].

8.3 Control Unit

For the proper functionality of the Control Unit (see 10.1), it requires two 4-bit unary counters, particularly cycle counter (CC) and bank counter (BC), and three 8-bit registers that we refer to as CUR1, CUR2, and CUR3 (or just CUR(s)). While all counters and registers are not chained and are the same in implementation, we provide the reader with the implementation of one counter and one register.

8.3.1 CUR - Control-Unit Register

Each CUR only needs to be able to load 8-bit data, reset, and to have a 3-state buffer[8]. Luckily, we have this 4-bit 3-state flip-flop covered in one conventional chip, 74LS173. Two chips are chained together with OE and address lines connected. The abstract circuit and breadboard implementation you can observe in the figures below and 23.

[7] if you would like to know the location of these configuration wires, you can search them in the real circuit or in the photos provided in Appendix B

[8] That is, to have Output-Enable pin.

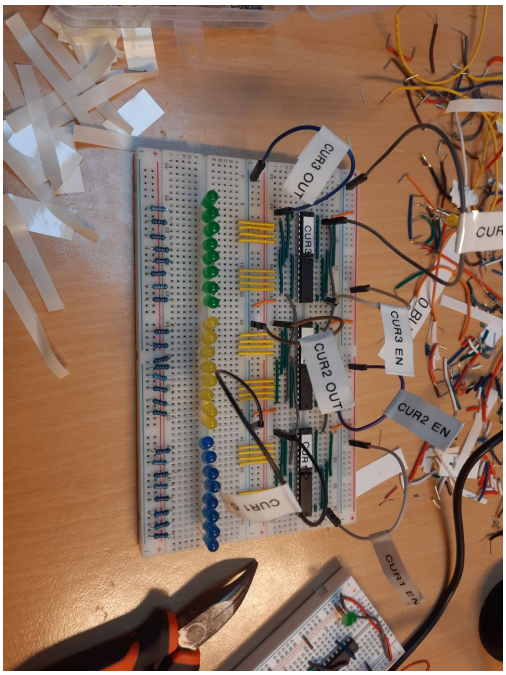


Figure 15: CURs on breadboard 1

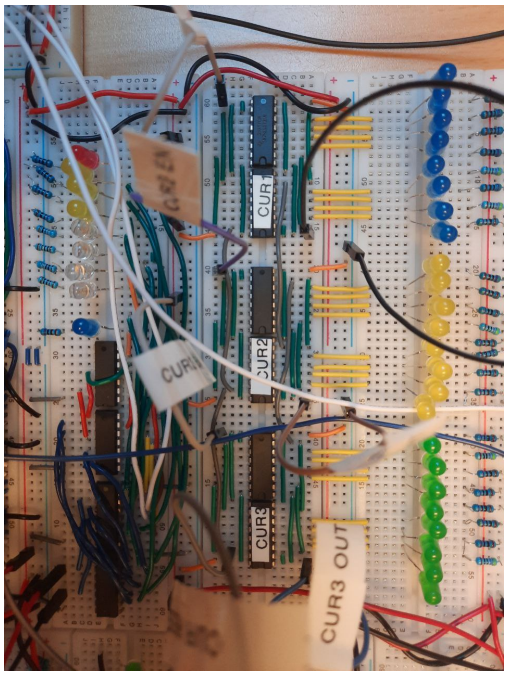


Figure 16: CURs on breadboard 2

8.3.2 Counters

The counter needs to count in this order: 00 — 01 — 10 — 11 — 00. It should have reset and output pin. Luckily, our inventory did contain a conventional chip with appropriate logic, 74LS161. Because the circuit is not complicated, we provide you with the abstract circuit and the real one, which you can find in the figures below.

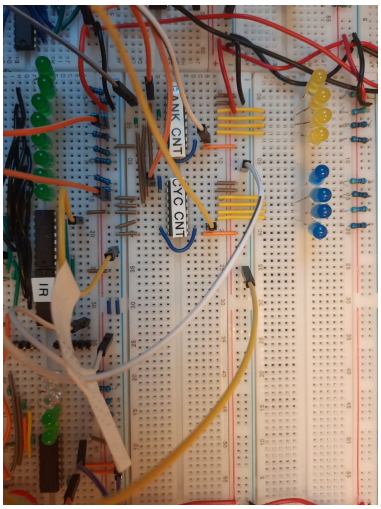


Figure 17: Counters on breadboard

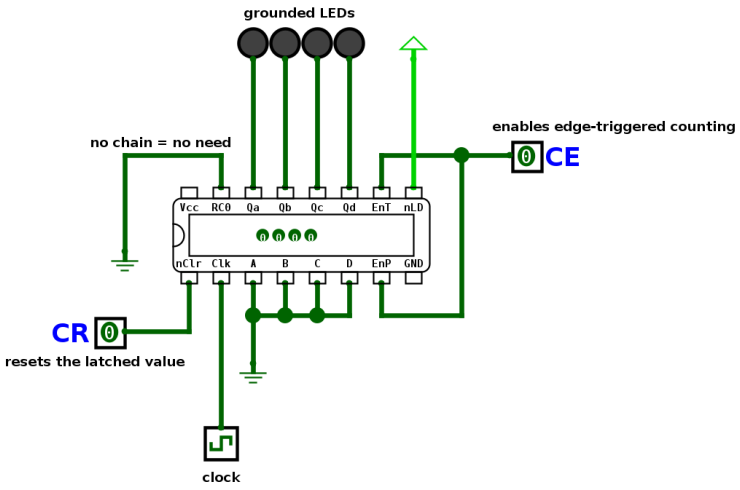


Figure 18: Counters - circuit

Before diving in the following section it is important to note that the implementation details of input and output are not provided as it deviates from the main course of this book.

8.4 Input

The input of the processor is a program written in an appropriate machine code. However, following the principle of abstraction, the user can make a program in brainf*ck programming language, use brainfix (**??**), or any other programming language translated in brainf*ck. Afterward, an arduino program takes up the code and converts it in the appropriate machine code, recognizable by bfcpu. However, this machine code is not yet stored in the processor's memory rather on a separate module of the processor. Therefore, a so-called ROM-Programmer comes into play. It is based on Arduino, and it loads machine code into the processor's ROM for the execution process.

8.5 Output

The output of the processor equals the output of the program. The process of execution is visible at different frequencies, set by a resistor with changeable resistance. Furthermore, the overall result will be displayed on a small screen that you can see in 19. The screen module is driven by Arduino for simplicity of its implementation. The brightness and symbol representation (binary, hex, etc.) could be explicitly set by the user.

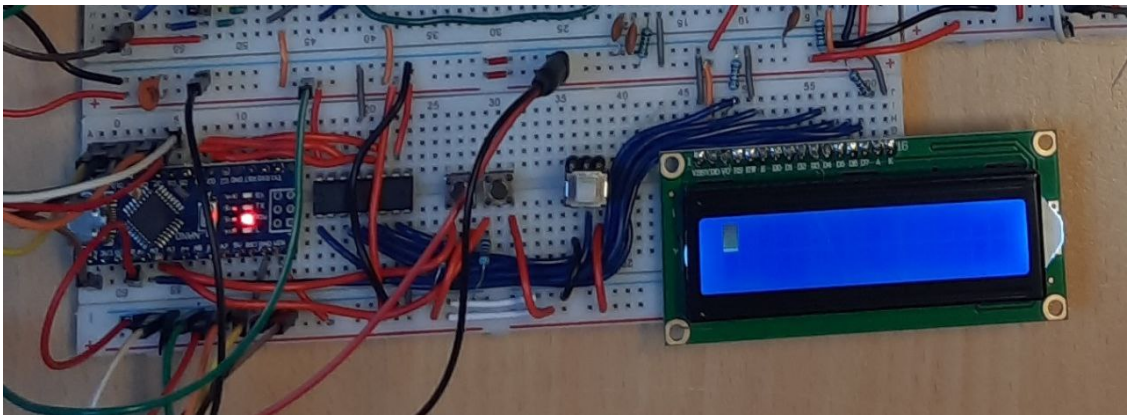


Figure 19: Arduino-driven output module (Screen)

9 Clock

Throughout previous sections, we mentioned a pretty basic term in almost every device of modern technology - (voltage) pulse. In this section, we will answer two questions regarding clocks and corresponding pulses:

- Why do the processor require pulsing? (9.1)
- How do we generate pulses in the processor? (9.2)

9.1 Why pulsing?

Every digital computer processor, not regarding quantum computing, requires the separation between different cycles, each containing crucial actions such as loading from a register in RAM or performing binary calculations in adders. We achieve such a behavior by biasing an approximate square-wave threshold voltage. It means that at some time intervals, potential difference is 3.3V or, in our case, 5V. At the subsequent interval, the voltage level is grounded. Consequently, at high bias levels operations (or cycles) are being performed whereas low voltage levels are devoid of any action. However, most chips nowadays are edge-triggered. It means that the cycles is executed on the edge of pulse where the derivative (dV/dt) is increased or decreased. This behavior is achieved by connecting the voltage square-wave to a filter, rectifier, or some other more primitive techniques. Edge-triggering is indicated on the manufacturer data-sheet.

9.2 How do we generate pulses?

Firstly, let us state the goal explicitly: we want to have an automatic non-stop square wave, manual-triggered square pulse, and the way to switch between two methods. In our implementation we used a frequently-used chip, the 555 Timer (see 20).

Automatic non-stop square wave, or the astable multivibrator, uses the following circuit:

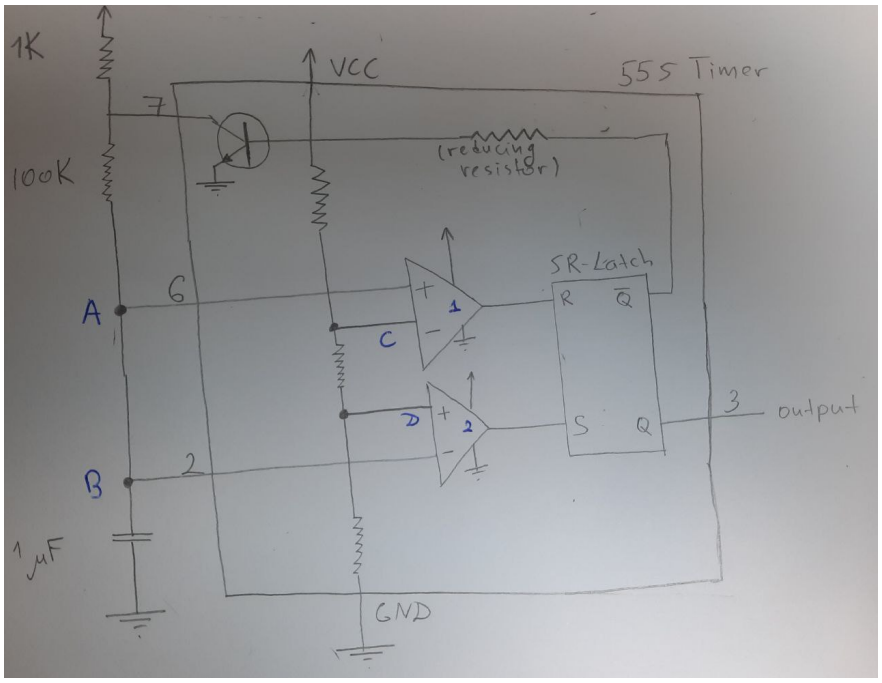


Figure 20: Astable Multivibrator using 555-timer

The three resistors inside the 555 timer constitute a voltage divider, and configure Op-Amps. The resistors have the same resistance, therefore, if the supplied power is 5V, then point C is at 3.33V whereas the point D is at 1.67V. In the beginning, point A and B[9] are at 0V due to the capacitor. Therefore, only

[9]Note, that they are always at the same potential level, thus we will referring to them simply as A.

the second Op-Amp outputs voltage (5V), whereas the first one is at 0V. Now, look at the SR-Latch with S set high and R set low. It will make Q high, thus the third pin of the 555 timer is going to be high, which corresponds to the high voltage level of the square wave. During capacitor charging the output is constant. However, when the capacitor is charged enough to set point A at 1.67V, the second operational amplifier turns off, while the first one is not altered. Therefore, S = 0 and R = 0 which means that the output of the 555 is not altered[10]. The capacitor continues charging and when it has a 3.33V-level, the first Op-Amp is activated. The consequence is that the R of the SR-Latch is high, which makes the output of the 555 timer low, which correspond to the grounded output of the square wave. However, the SR-Latch now outputs voltage to the base of the transistor at pin 7. This makes the collector collect charge and emit it to the ground, thus making two flows to ground: from the voltage supply AND from the capacitor. This action will discharge the capacitor until 1.67V where the SR-Latch will again output high voltage. This is the basic principle of the 555-timer-driven astable multivibrator. The real oscilloscope output is shown in the following figure:

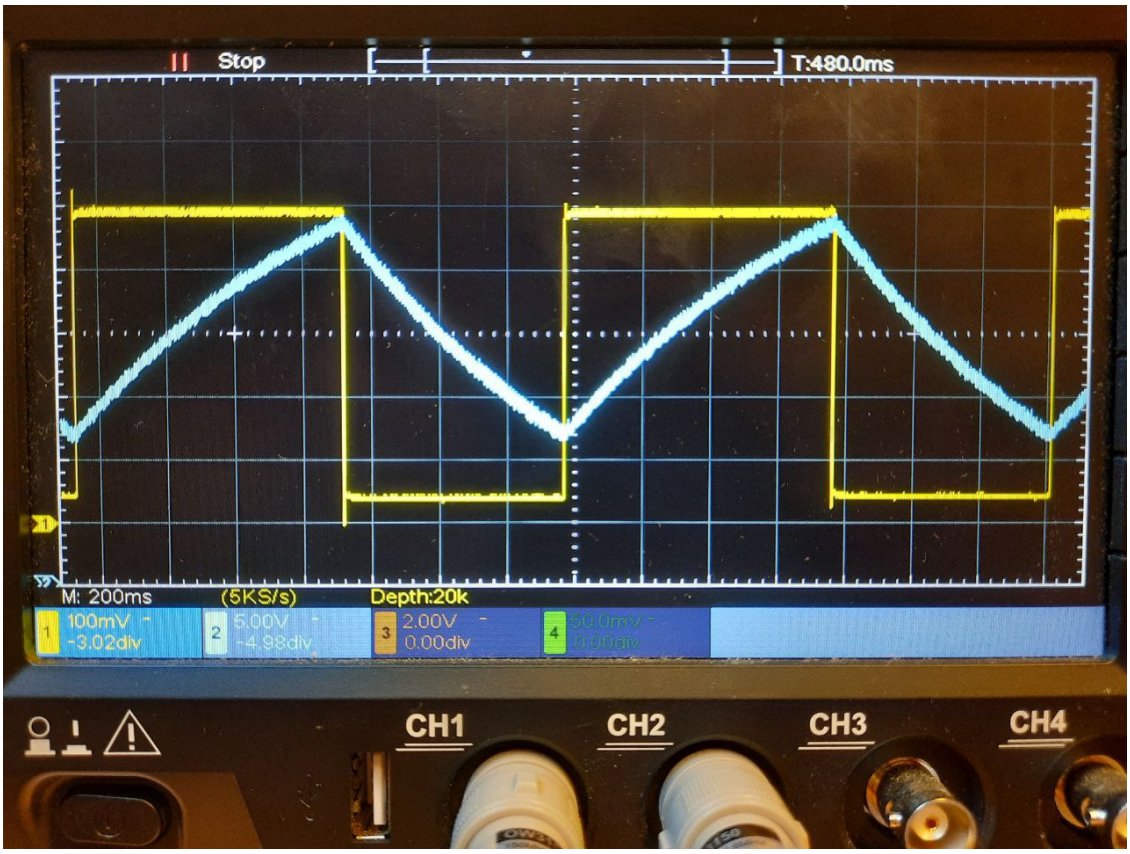


Figure 21: Oscilloscope of the clock and capacitor used with the 555 timer

Manual-triggered square pulse, or the monostable multivibrator, uses similar principle but voltage to the second pin is supplied only when the button is pressed, see 22. This means that ones pressed, the trigger immediately sets the output high, and keeps it so while being pressed. Once the button is not pressed, the discharging mechanism sets the output low.

[10]The SR-Latch will change the output from High to Low only when R is high.

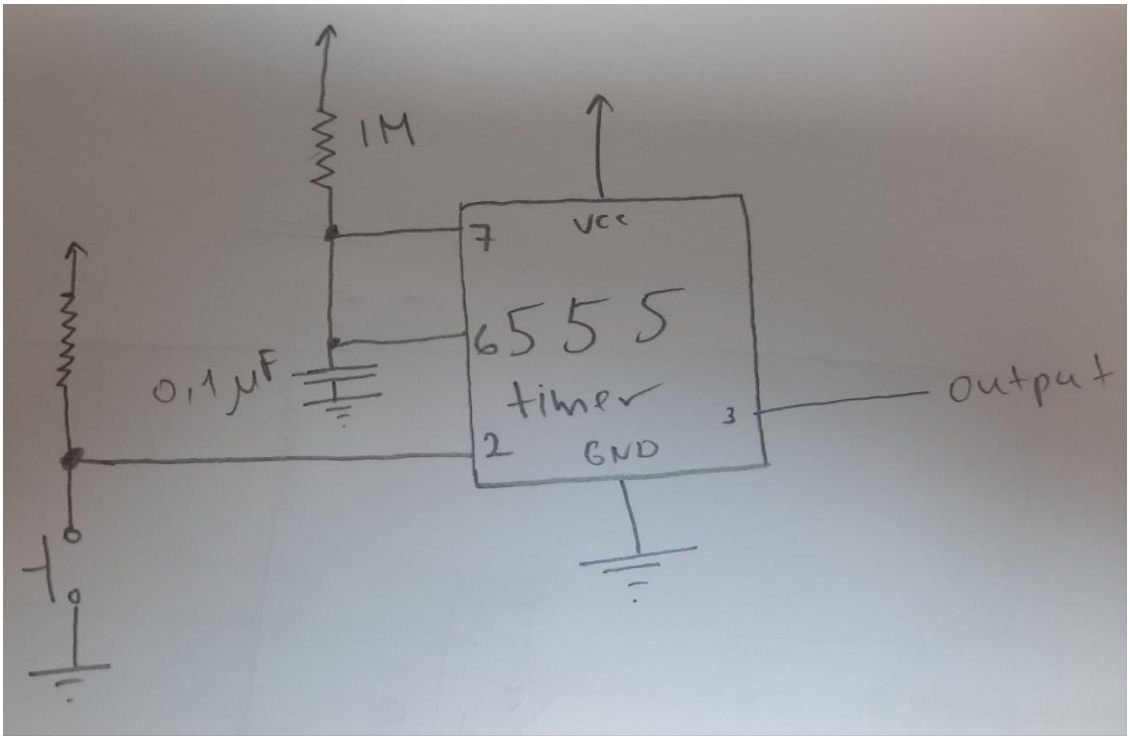


Figure 22: Monostable Multivibrator using 555-timer

The switcher between two modes is implemented by combining several digital gates (NANDs). Further investigation of this part is of no interest and can be traced by looking at the real circuit. The only important thing to note here is that the clock output (so either mono- or astable multivibrator) is ANDed with the hatch pin, which is used for the processor's master reset button. If Hatch is high, the clock is low no matter what is the output of multivibrators.

10 Control Unit

10.1 How do things get done?

The main parts of the control unit, which we will refer to as CU, are IP-R, Program ROM (PR), Machine Code ROM (MCR), bank counter (BC), cycle counter (CC), CURs (CU-Registers 1, 2, and 3), and F-R (Flag Register). Before we delve in the working principle, skim through the diagram below:

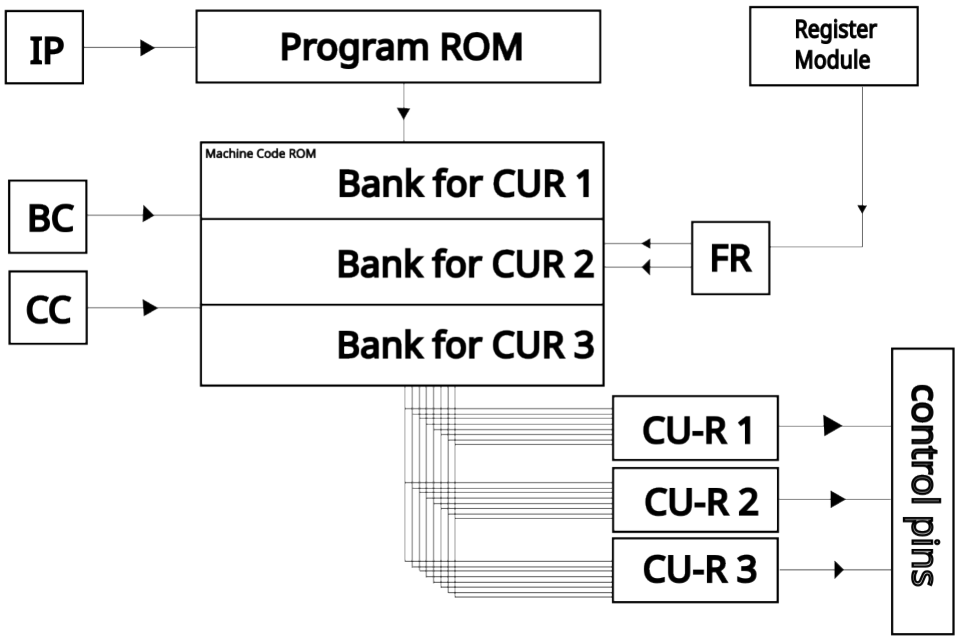


Figure 23: Control Unit Diagram

Firstly, the Machine Code ROM has a certain number of address lines composed of the output of the two counters (BC and CC), flag register, and program ROM.

As described in **??**, the Program ROM contains the actual executable that the processor aims to execute. Each address contains a specific sequence of bits (called instructions) that are to be loaded in the Machine Code ROM to search for and trigger the responsible physical action. Each Program ROM's instruction is referred to by the IP. To conclude, the first part of the Machine Code ROM address involves the output of the Program ROM.

The next part is counter-part, BC and CC. On each cycle, which we will describe later, the counter links to the next value and gives it up as an address line for the Machine Code ROM. As indicated in 24, the given BC value is the two most significant bits. We will explore the importance of this configuration later.

Flag Register contributes to the two bits of the address lines. It is necessary to get the correct value for the executable machine code from of the Machine Code ROM.

In total, we have 20 control pins. Those are pins like loading, resetting, incrementing, etc. of individual registers, counters, and modules. Because we could not store a 20-bit value stored in the ROM (Machine Code ROM), it was required to separate the value in three different ROM memory locations. We called such a location the bank with appropriate number from 1 to 3. The first third of the total number of memory of the ROM is dedicated for the bank 1, the second third - for the bank 2, and the same for the bank 3. C++-generated example you can see in the figure below:

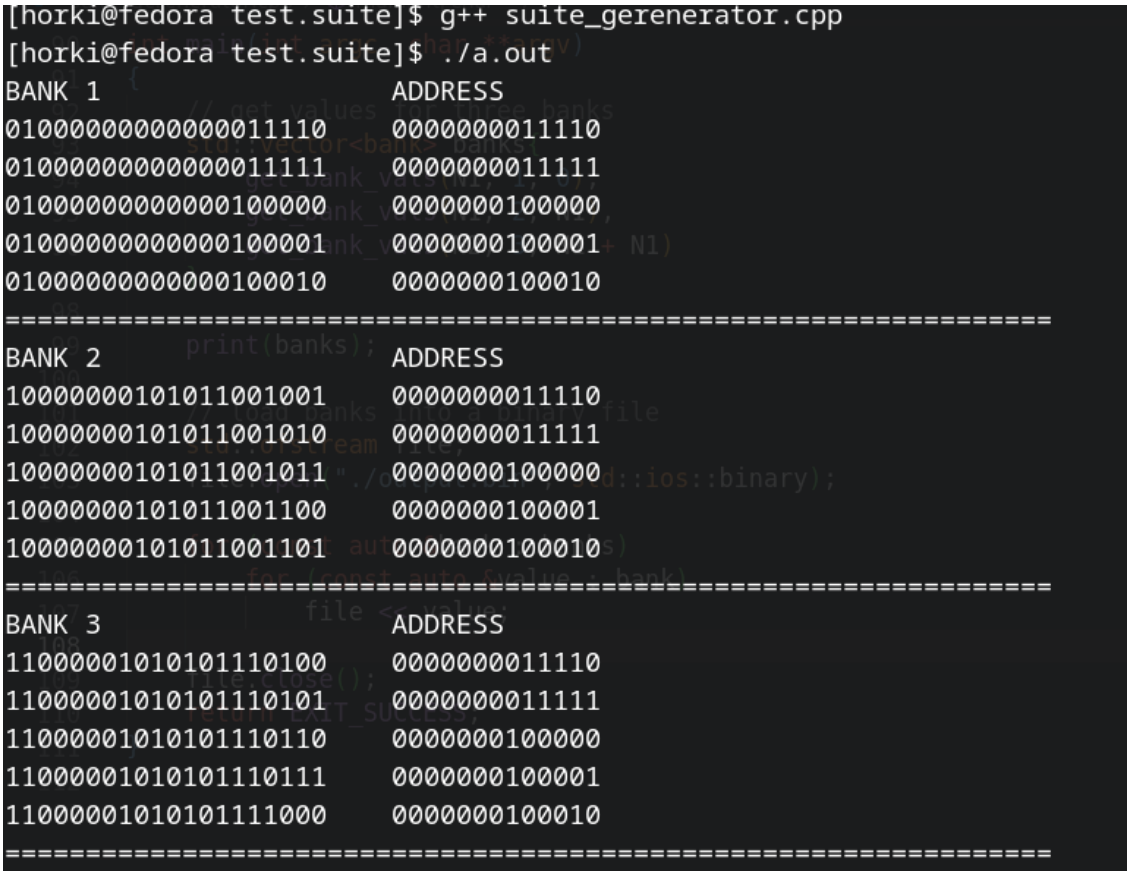


Figure 24: Banks in the Machine Code ROM

The first two bits (most significant) are decided by BC, and are responsible for the bank selection. For example, 00 selects for the bank 1, 01 - for the bank 2, and 10 - for the bank 3. However, what would we do with 11 value[11]? We deal with this problem by using all four possible cycles of the cycle counter.

Providing a reasonable address to the Machine Code ROM, the output is the 8-bit bank value, which is a part of the entire 20-bit action that is directly connected to control pins via CURs.

The major problem is how do we load the 20-bit values if we are restricted by 8-bit output from the Machine Code ROM? This is solved by the cycle counter, and 3-state behavior of the CURs.

When we start the computer, the first value on CC is 00 - a so-called beginning cycle. At the same time, the value on BC is 00, thus, the first bank is selected. Remembering, that each action is edge-triggered, we can separate some number of actions during the same clock pulse: on raising and falling edge. On the raising edge, we load data from the Machine Code ROM and select the CU-R 1, because we do not want the first 8-bit of the machine code to be loaded in the CU-R 2 and CU-R 3. On the falling edge of the same pulse, we load this 8-bit value of the bank 1 in the CU-R 1; and we increase the value of counters. Now, we are on the cycle 01, where we perform absolutely the same operations, but now we select a different CU-R. The same applies for the cycle 10. After the machine code is loaded in every CU-R, we need some way to combine them all and execute, and also to set BC to 00. This all is done on the raising and falling edge of the fourth (11) cycle: enable the output of all CURs (3-state behavior), increment BC two times (10-¿11-¿00), and some minor configurations. As the last step, we execute control pins on the same clock cycle. To shed more light here, you can scrutinize the time-diagram below, and skim through an example in **??**[12].

[11] Remember, the bank counter is a 4-bit counter, thus it traverses this cycled linked-list: 00 -¿ 01 -¿ 10 -¿ 11 -¿ 00

[12] Before it, remember that the falling edge is implemented via inverting the actual clock, so the falling edge is equal to the inverted raising edge

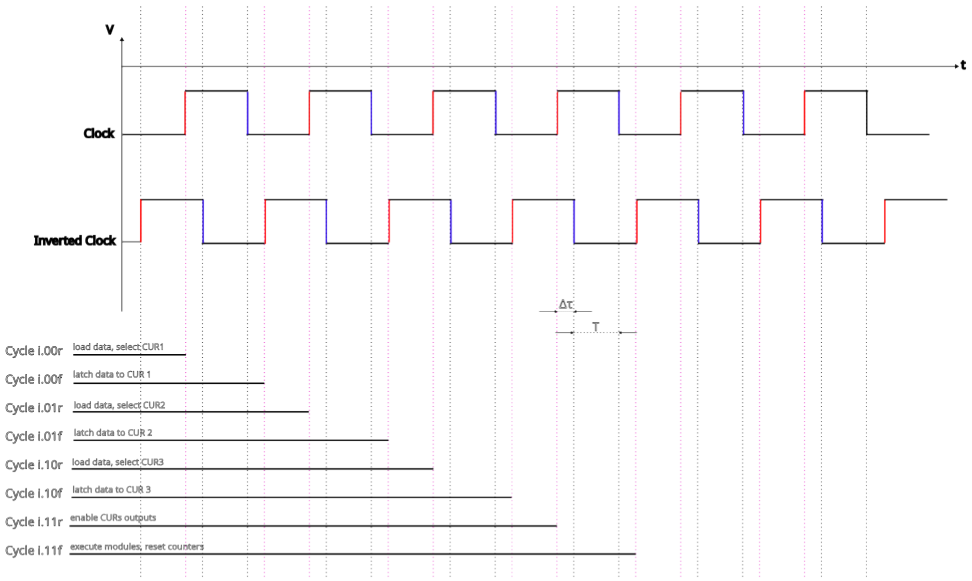


Figure 25: Clock Diagram for the CU

10.2 Performance

It is overtly difficult to get an exact formula or graph for performance. Therefore, it is only possible to get an estimate formula (see 10.2.1).

10.2.1 Theoretical Estimate

Introduce new symbols:

$\delta\tau$- inverting delay (see 25).

T - clock pulse period (see 25).

t_i^j - execution time of performing an action j in i-th component (such as D-R resetting).

t_{max} - time of execution.

N - number of instructions in the Program ROM.

Time of execution must be less than the clock pulse period. Otherwise, a new pulse will be generated before previous commands have been still being executed. Because action of components are occurring simultaneously, only the maximum execution time is important. This results in this possible formula for the highest frequency of the clock (see **??**) for the instruction k:

$$f_{k_{max}} = \frac{1}{\delta\tau + \max\left(t_i^j\right)}$$

This suggests the following estimate formula for the average frequency for the given program with N instructions:

$$< f >= \frac{1}{\frac{1}{N}\sum_{k=1}^{N}(\delta\tau + \max\left(t_i^j\right))}$$

However, despite the formula being given, it is quite cumbersome to check the error. This results from the fact that under given conditions, temperature, pressure, humidity, manufacturer, etc. each component might have differing execution timings. This results in the lack of data.

11 Algorithms

The BFX compiler implements a series of algorithms to perform basic operations on the data stored in the data-array, which map to common operators and statements in most programming languages. This section will document each of these algorithms, starting with the most fundamental ones, which we can then use to implement the more complex operations. In the process, we will develop a pseudo language that maps directly to BF-code. The statements of the pseudo language will always be written within a set of curly braces, in order to seperate them from the BF operators.

11.1 Moving the Pointer: {x}

Whenever we need to operate on a cell, we first need to move the datapointer to this cell. It is assumed that the compiler is aware of the current pointer position and will therefore be able to move the pointer to any other cell, as long as the address of this cell is also known at compile-time. From now on, the operation of moving to a cell designated by some variable `x` will be written in as {x}.

In future expressions enclosed by curly braces, the pointer is always left implicitly at the address of the left-hand-side operand.

11.2 Setting a Fixed Value: {x <- n}

The next elementary operation is setting some cell to a specific (known) value. We will not assume any prior knowledge of the current contents of this cell, which means the first step is to clear it completely, which we can do by iteratively decrementing the cell until it becomes zero: `[-]`. We then follow it up by the desired amount of increments. For example, setting cell `x` to `5` can be done with:

```
{x} [-] +++++
```

From now on, we will use the following shorthand for setting a cell (x) to a fixed value (n): {x <- n}.

11.2.1 Using + Instead: {x <- n+}

Instead of decrementing the cell until it becomes zero, we could also choose to increment it using `[+]`. The cell will eventually overflow and wrap back to zero, achieving the same result. This might seem unnecessary and dangerous, especially for 16-bit or larger values, because it can take a lot of time before the cell overflows. However, for some algorithms, where a cell is intentionally being *underflown*, this is a helpfull tool to have at hand. When we need it, we will use a slightly different shorthand: {x <- n+}.

11.3 Moving Data: {x <- y}

It turns out that moving data is a lot easier than copying data. When moving data from one cell to another, the source-cell will be left empty. Below is a listing for the algorithm that moves the contents of cell x into cell y, leaving cell x empty after the move:

```
{y <- 0}
{x} [ {y} + {x} - ]
```

After clearing `y`, we move the pointer to `x` and decrement it until it becomes zero, while at the same time incrementing cell `y`. This algorithm can be extended to move the data into arbitrarily cells, simply by adding variables to the loop and incrementing those as well. Moving data from `x` to `y, z, ...` will from now on be written as {(y, z, ...) <- x}, analogous to moving a fixed value into a cell.

11.4 Assignment/Copy: {x = y}

An assignment of the form `x = y` will assign the contents of cell `y` to cell `x`, overwriting the previously present value of `x`. This can be achieved by moving the contents of `y` into both `x` and a temporary value `tmp`. This will leave `y` empty, but we can restore it by moving the contents of `tmp` back into `y`.

```
{(x, tmp) <- y}
{y <- tmp}
```

The shorthand for this operation will from now on be: {x = y}

11.5 Addition

11.5.1 In-Place Addition: {x += y}

It turns out that adding some value to a cell looks just like a copy, but without resetting the target cell first. Below is the pseudocode for adding the contents of y to x, leaving y intact.

```
{tmp <- 0}
{y} [ {x} + {tmp} + {y} - ]
{y <- tmp}
```

This operation will be denoted as {x += y}.

11.5.2 Return Variable: {z <- x + y}

Rather than adding the contents of a cell to another one and changing the value of the cell on the receiving end, we should also have an algorithm that stores the sum in a new cell, leaving both operands untouched (or at least unchanged). This can be achieved simply by copying the contents of one of the cells to a third cell and applying the addition algorithm:

```
{z = x}
{z += y}
```

11.5.3 Subtraction

Subtraction works exactly analogously to addition: we just replace the + with a -. This leads to the following shorthands:

- {x -= y} for subtracting a cell in-place and
- {z = x - y} for storing the difference in a return-variable.

11.6 Multiplication

Multiplication can be implemented as repeated addition, for which we already have the tools. The same strategy will be used as before, where we first develop an algorithm aimed at multiplying a cell in-place.

11.6.1 In-Place Multiplication: x *= y

To multiply x by the value stored in y, we need to add x to itself, y times. To achieve this, we copy the value of y to a temporary cell, which we can decrement while adding a copy of the original value of x to itself:

```
{xCopy = x}
{yCopy = y}
{yCopy}
[
  {x += xCopy}
  {yCopy} -
]
```

11.6.2 Return Variable: {z <- x * y}

Like before, we can copy the value of one of the operands into the return address and apply the in-place multiplication algorithm:

```
{z = x}
{z *= y}
```

11.7 Logical Operators

11.7.1 NOT: {x <- NOT y}

The NOT operator basically checks whether the argument is 0, in which case it returns 1. In all other cases, it should return a zero. This can be implemented using the loop-operators, which are the only conditional instructions available to us in BF.

```
{x <- 1}
{yCopy = y}
[
  {x     <- 0}
  {yCopy <- 0}
]
```

11.7.2 AND: {z <- x AND y}

The AND operator will only return 1 when both arguments (x and y) are nonzero:

```
{z <- 0}
{yCopy = y}
{xCopy = x}
[
  {yCopy}
  [
    {z     <- 1}
    {yCopy <- 0}
  ]
  {xCopy <- 0}
]
```

11.7.3 OR: {z <- x OR y}

The OR operator returns a 1 when either of its arguments are nonzero:

```
{z <- 0}
{xCopy = x}
[
  {z     <- 1}
  {xCopy <- 0}
]
{yCopy = y}
[
  {z     <- 1}
  {yCopy <- 0}
]
```

11.7.4 Greater Than: {z <- x > y}

To check whether x has a higher value than y, we simply decrement x until it becomes 0. At each step, we also decrement y and check whether it is being decremented beyond zero (check for underflow). If that happens, this means that x is indeed the biggest.

```
{z <- 0}
{underflow <- 0}
{yCopy = y}
{xCopy = x}
[
  {underflow <- NOT yCopy}
  {yCopy} -
  {z <- z OR underflow}
  {underflow}
  [
    {yCopy <- 0+}
    {underflow <- 0}
  ]
  {xCopy} -
]
```

11.7.5 Less Than: {z <- x < y}

The less than algorithm can just be expressed as a greater than algorithm, for which the arguments have been swapped around. It will therefore not be repeated below.

11.7.6 Equals: {z <- x == y}

Once the 'greater than' and 'less than' operators have been properly defined, the 'equals' operator can be defined in terms of these:

```
{z <- {NOT x < y} AND {NOT (x > y)}}
```

11.8 Division and Modulo: {(div, mod) <- x / y}

Division and modulo are done in a single algorithm. Division is implemented as a series of subtractions, counting how many times the numerator fits in the denominator. Two special cases are handled:

1. When the numerator is zero, the division algorithm is skipped and zero is returned.
2. When the denominator is zero, the algorithm is skipped and the maximum cell value is returned (which is as close to infinity as we can get).

To indicate whether either of these special cases was handled, we set a flag (the loop-flag) which is reset while handling these cases. Only when this flag is still set will the division algorithm be executed. Below is the pseudocode for the expression {(div, mod) = x / y}, where `div` is the result of the division and `mod` is the remainder after the division.

```
{div <- 0}
{mod <- 0}
{zeroFlag <- 1}
{loopFlag <- 1}

# Special case 1: denominator is 0
# ==> return max value (255 on 8-bit arch)
```

```
{zeroFlag <- NOT y}
[
  {div <- 255}
  {mod <- 255}
  {loopFlag <- 0}
  {zeroFlag <- 0}
]

# Special case 2: the numerator is 0
# ==> return 0 (even in 0/0 case)
{zeroFlag <- NOT x}
[
  {div <- 0}
  {mod <- 0}
  {loopFlag <- 0}
  {zeroFlag <- 0}
]

# Division algorithm: repeated subtraction
{xCopy = x}
{yCopy = y}

{loopFlag}
[
  {xCopy} -
  {yCopy} -
  {mod}   +

  # yCopy became 0?
  # ==> increase result by one and reset yCopy
  {zeroFlag <- NOT yCopy}
  [
    {div} +
    {mod <- 0}
    {yCopy = y}
    {zeroFlag <- 0}
  ]

  # xCopy became 0?
  # ==> done
  {zeroFlag <- NOT xCopy}
  [
    {loopFlag <- 0}
    {zeroFlag <- 0}
  ]
  {loopFlag}
]
```

12 Appendix

12.1 Appendix A: Old Driver Module Design

As visible from figure 7, the old driver module has two parts: common interface, select interface. Connection to registers is the bridge between the driver module and registers in the register module.

We used two demuxers to create the common interface. The basic idea behind this implementation is the following: DEC pin defines the way where CE and CLK signals are propagated, so DEC is connected to the select pin of both demuxers. CE switches the signals off or on, while CLK stimulates the alterations in UP/DOWN pairs, contributing to the change in the value of a register.

Because of the inner circuit of 74LS193, we need to avoid simultaneous pulsing and UP/DOWN pair grounded since it unfavorably alters the value of a register. The solution was quite straightforward: CLK pin of the driver module is connected to the output of anded clock signal and CE pin.

The resulting UP/DOWN pair is directed to the select interface, that redirects the pair to a specific, selected, register. This behavior is accomplished by utilizing 3-to-8 demuxers, because we have less than 8 registers but more than 4.

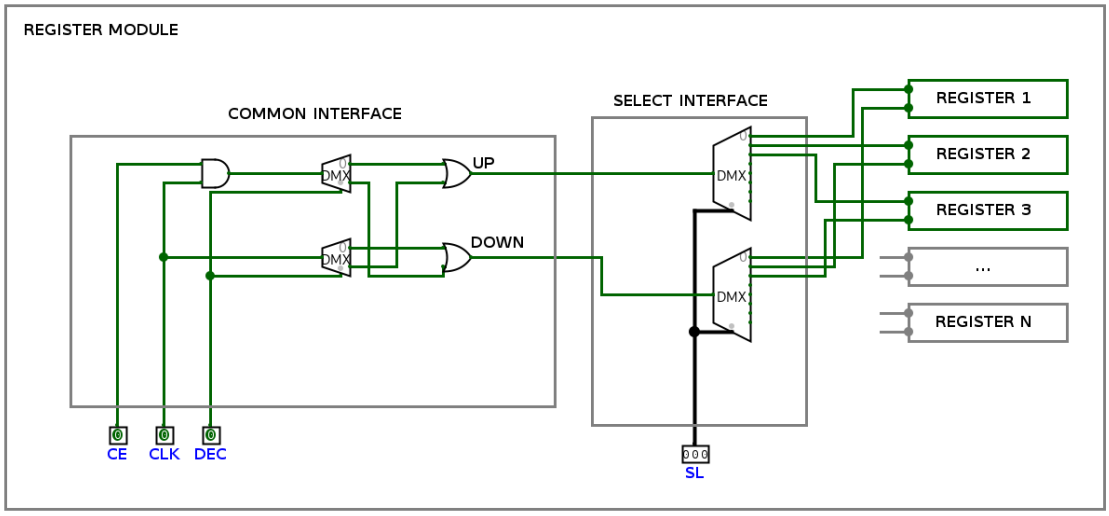


Figure 26: The Old Driver Module Implementation

12.2 Appendix B: Gallery - real-life process

In this appendix, you can explore the real-life process. What circuit we built first and next.[13]

[13] You are free to skip this appendix as there is nothing important going on here. It is a necessary step for the project in secondary school (profielwerkstuk, PWS).

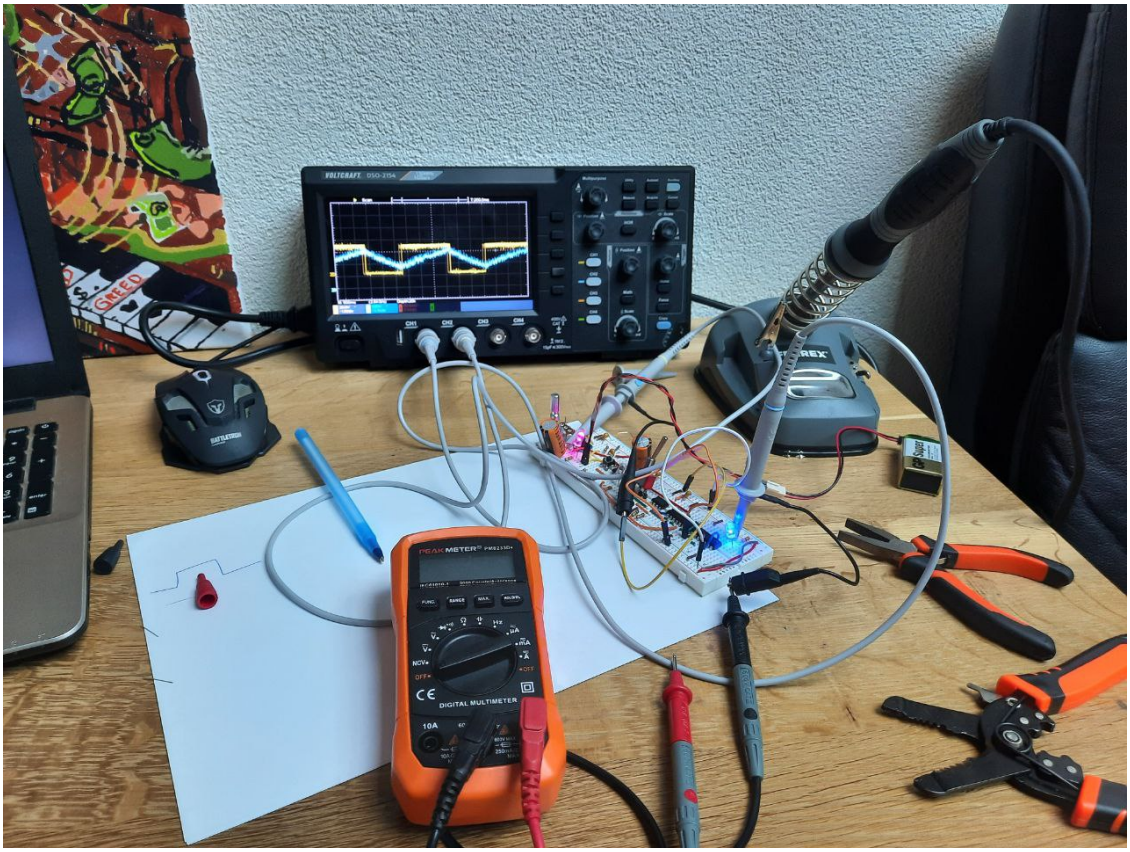


Figure 27: Photo number 1

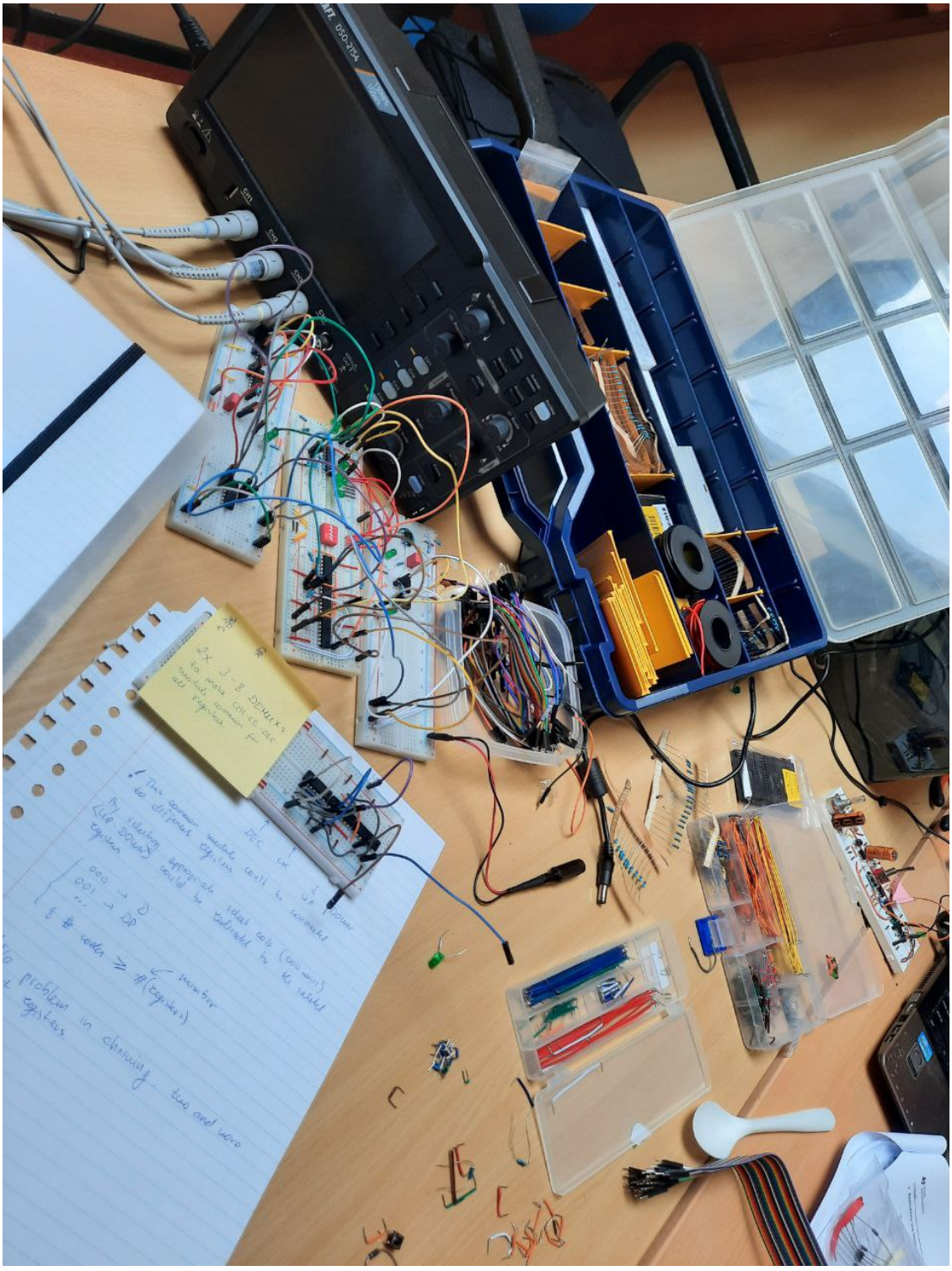


Figure 28: Photo number 2

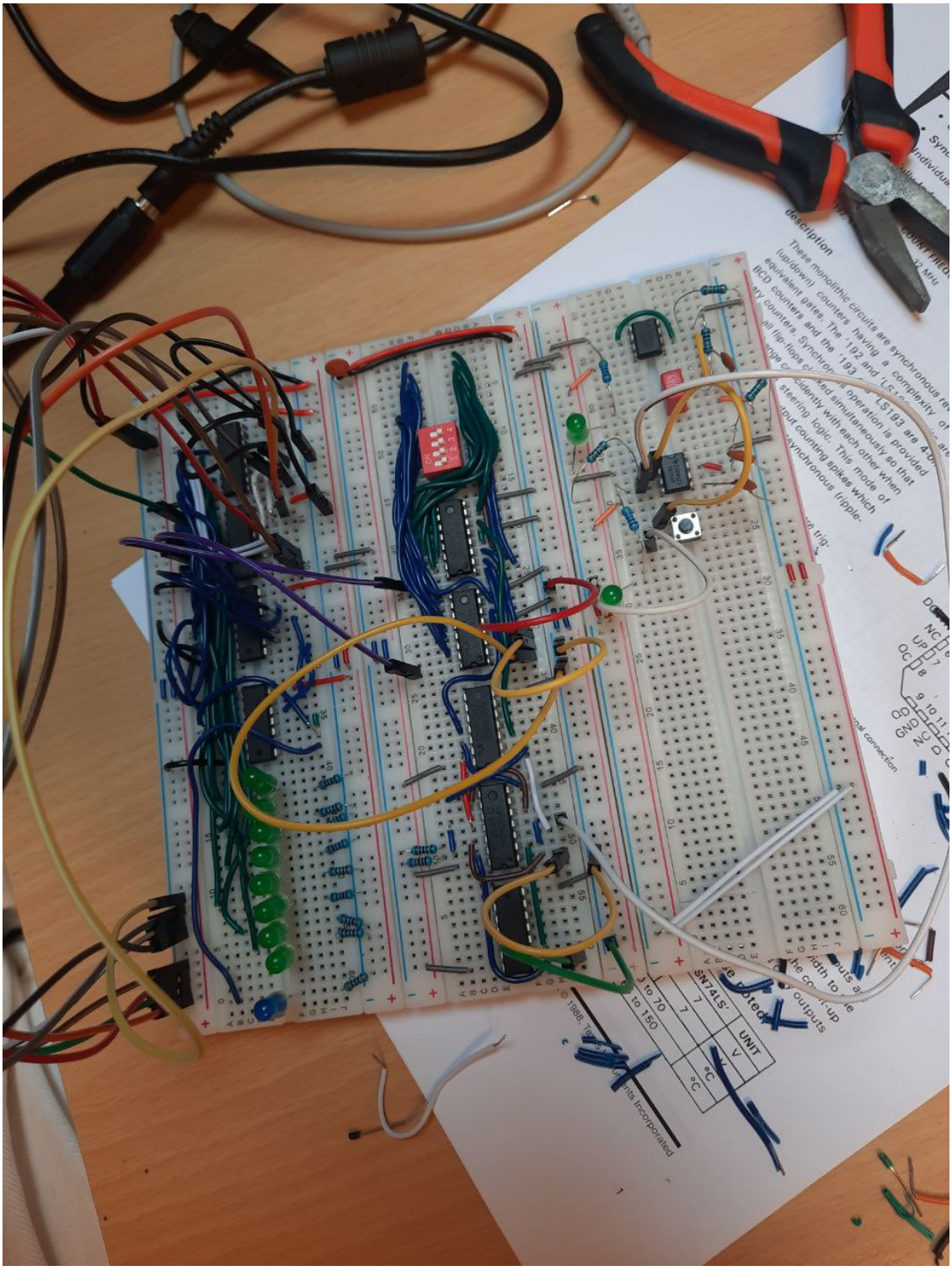


Figure 29: Photo number 3

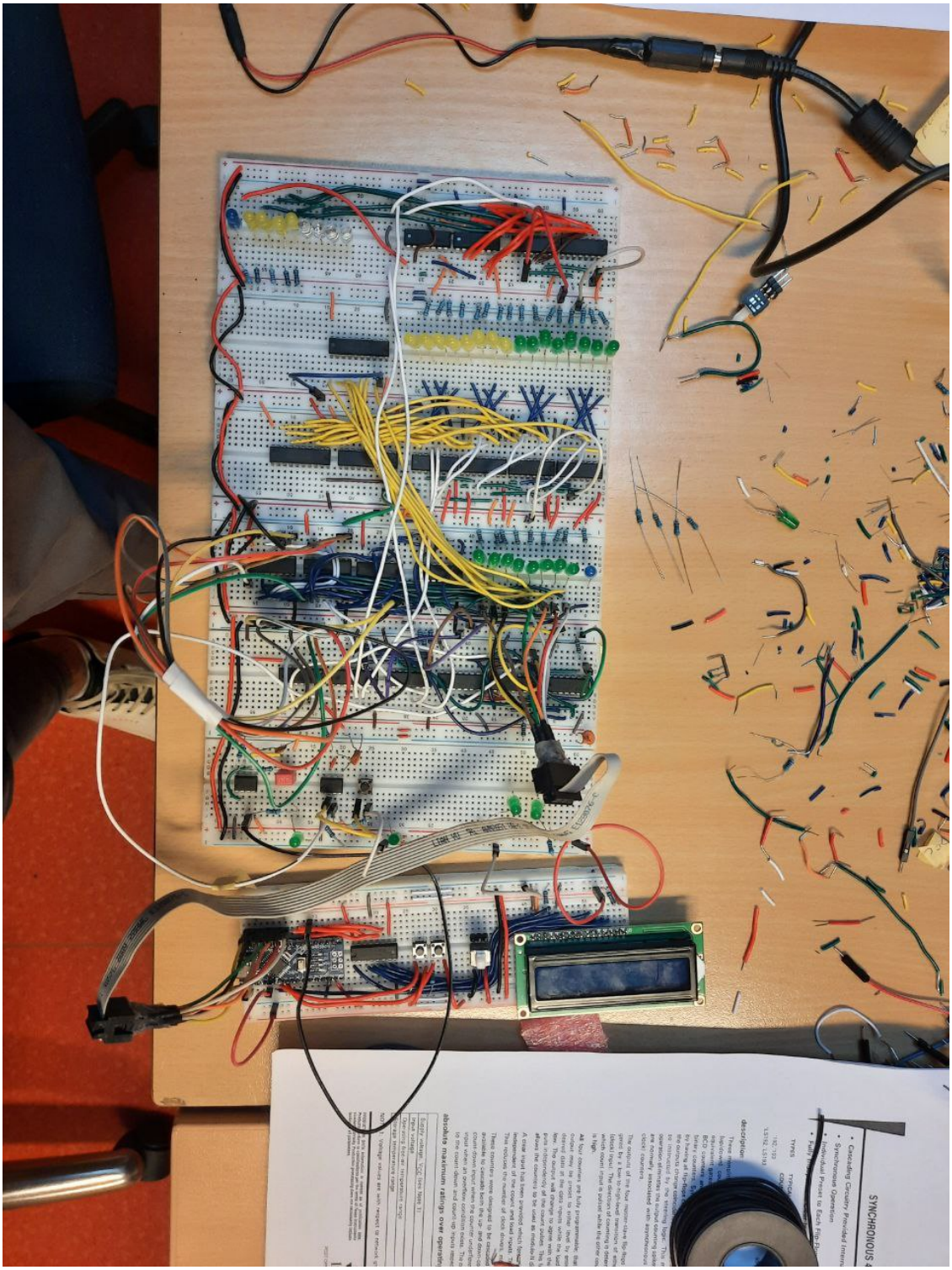


Figure 30: Photo number 4

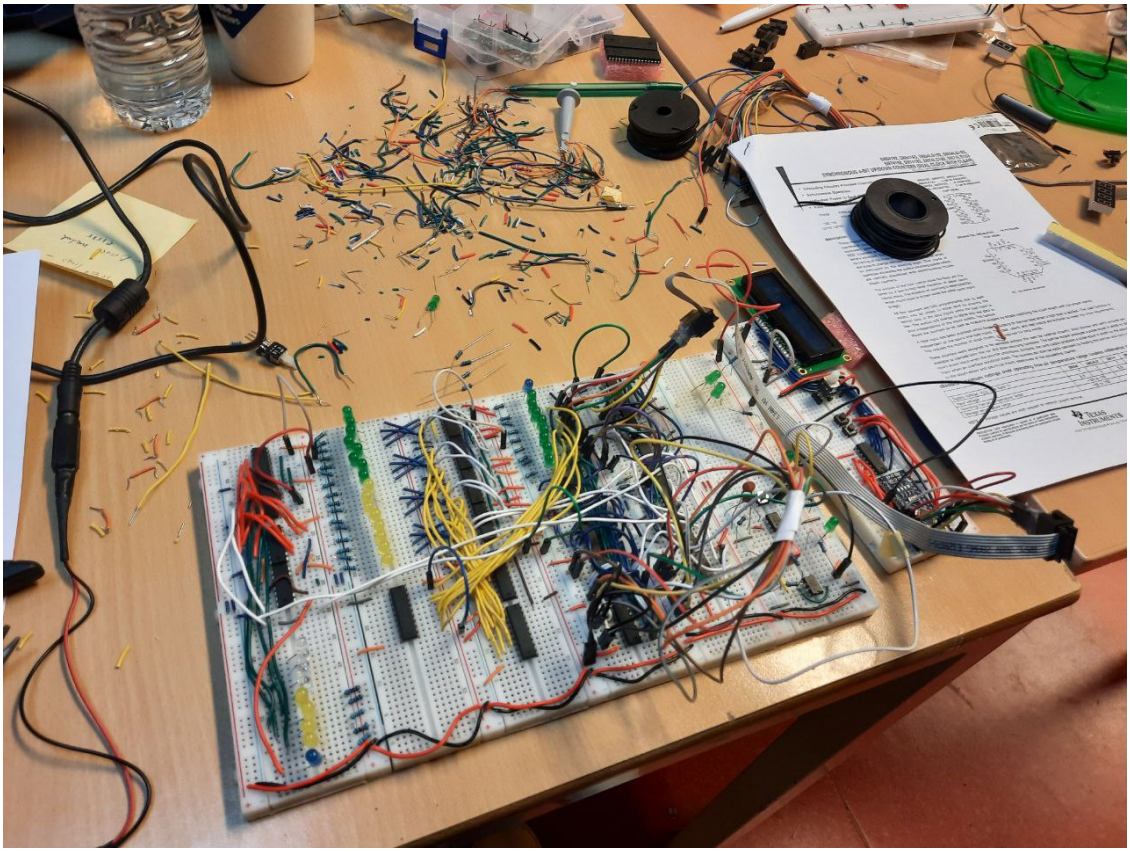


Figure 31: Photo number 5

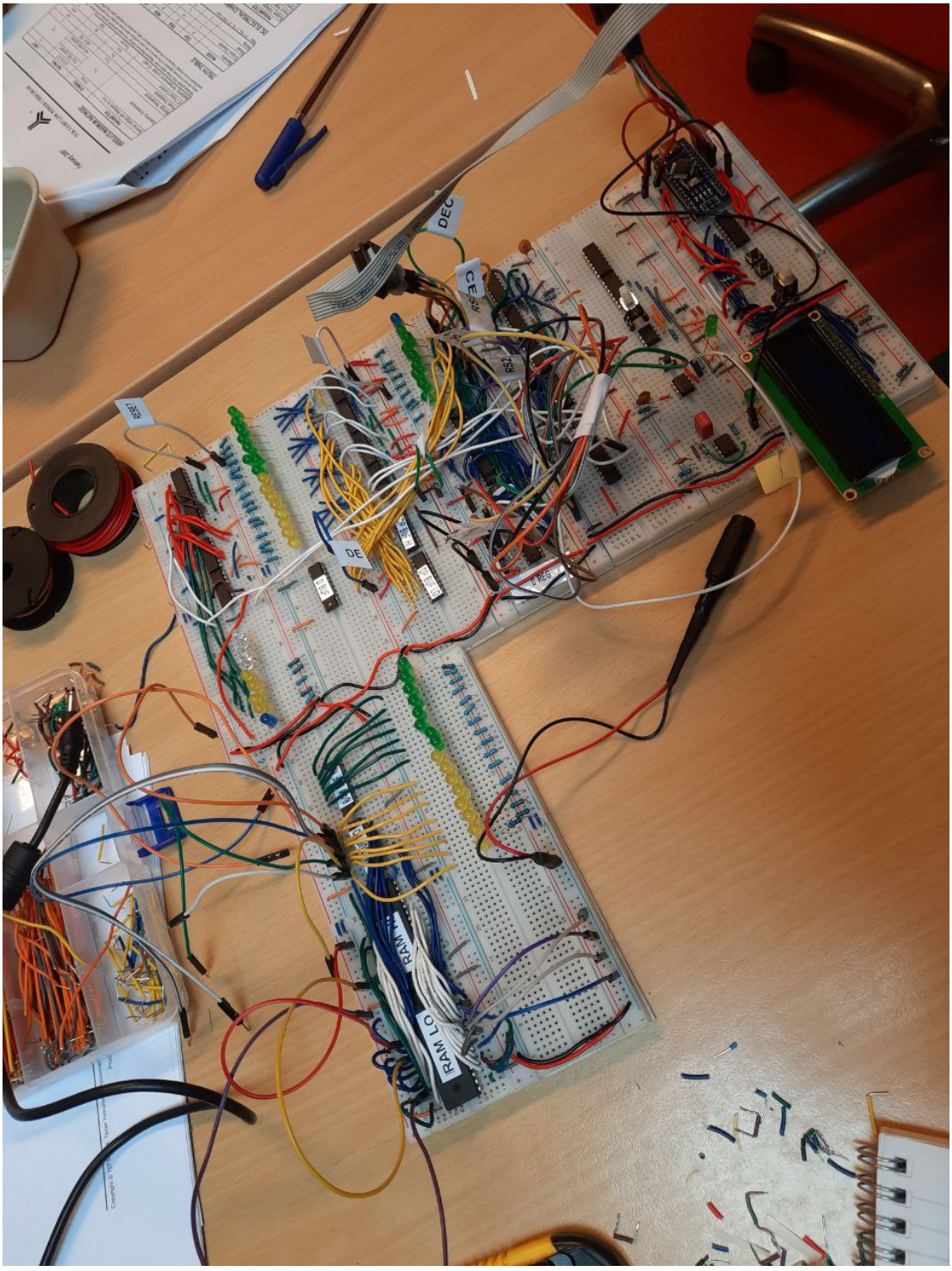


Figure 32: Photo number 6

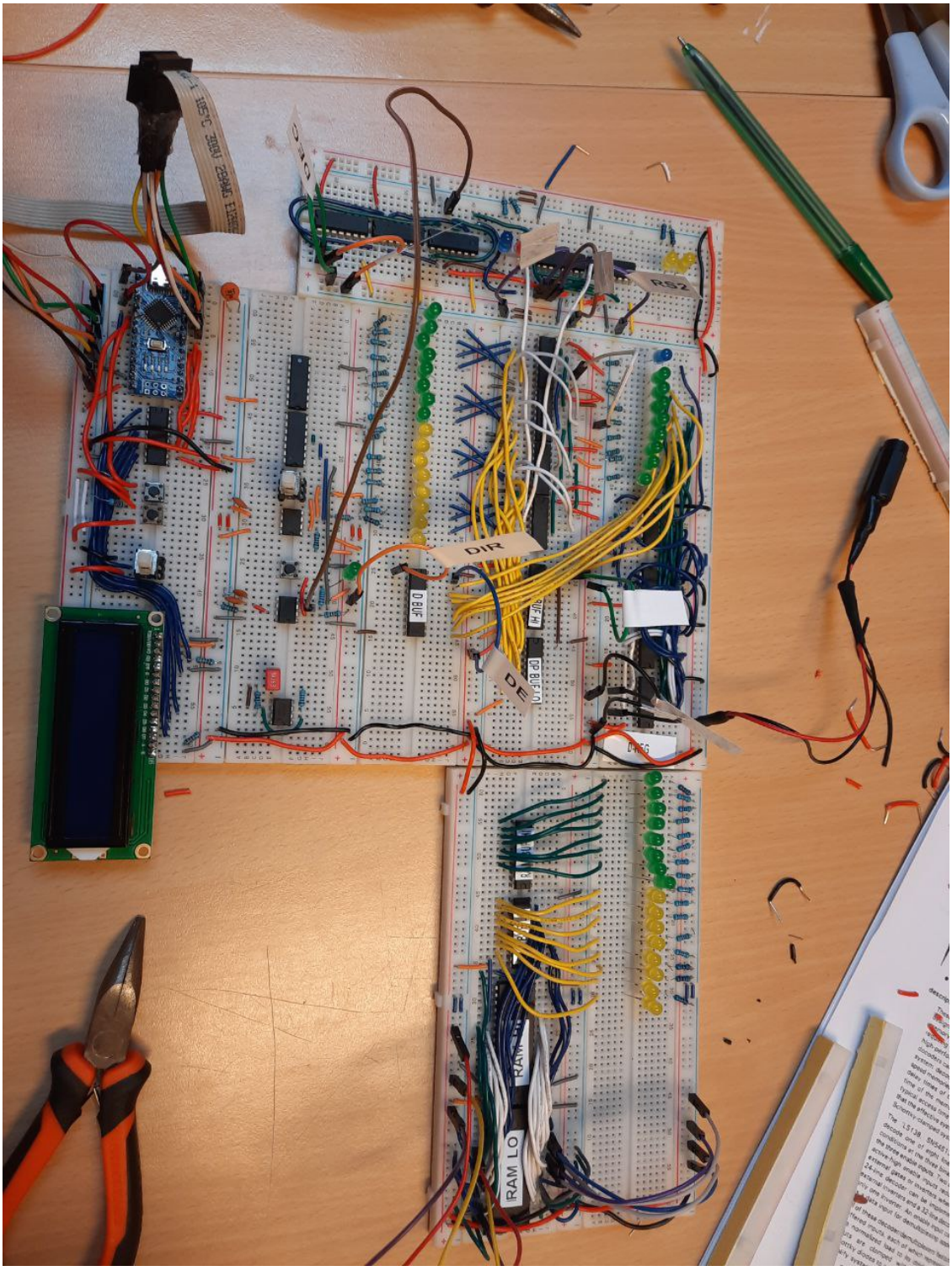


Figure 33: Photo number 7

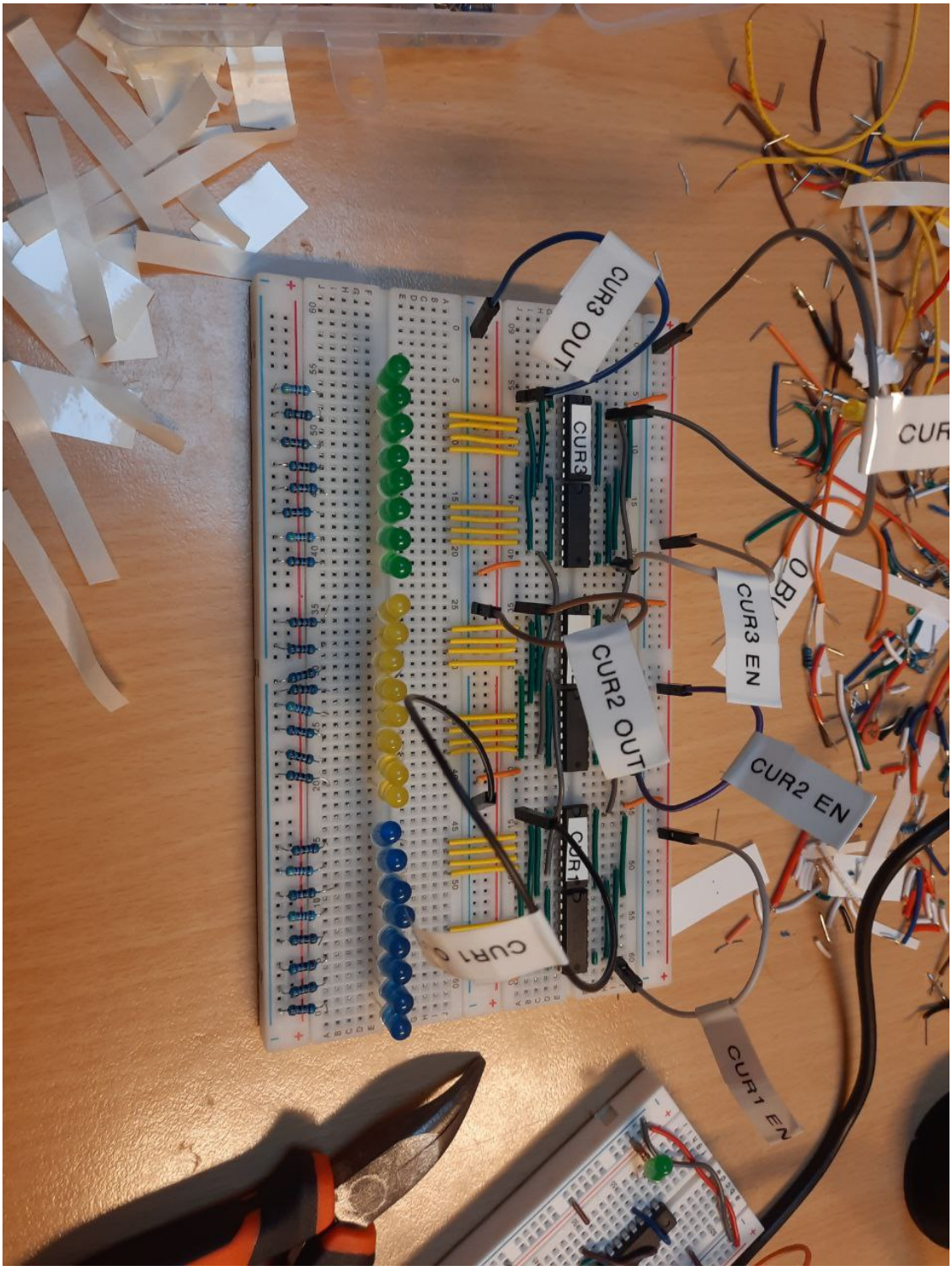


Figure 34: Photo number 8

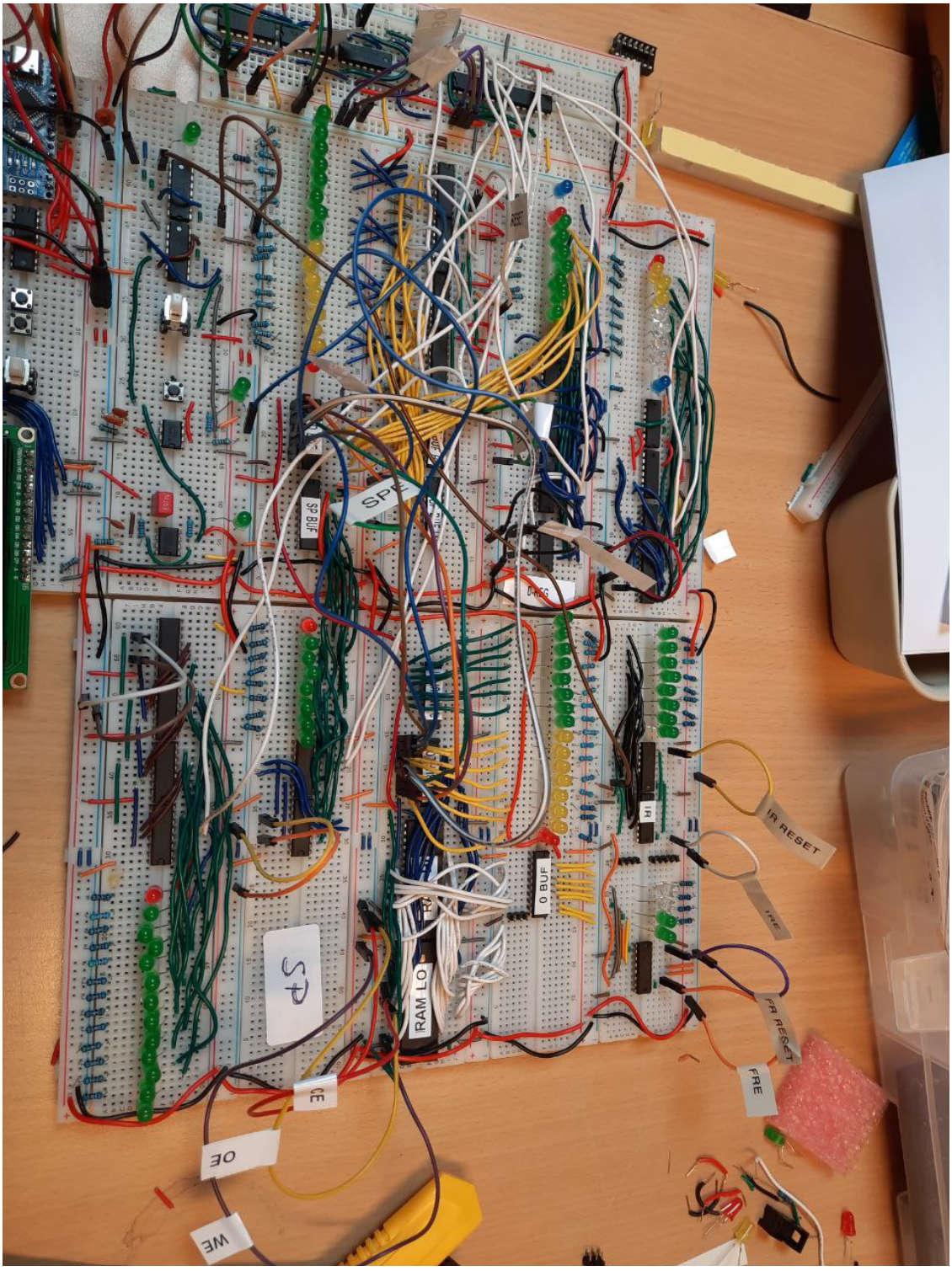


Figure 35: Photo number 9

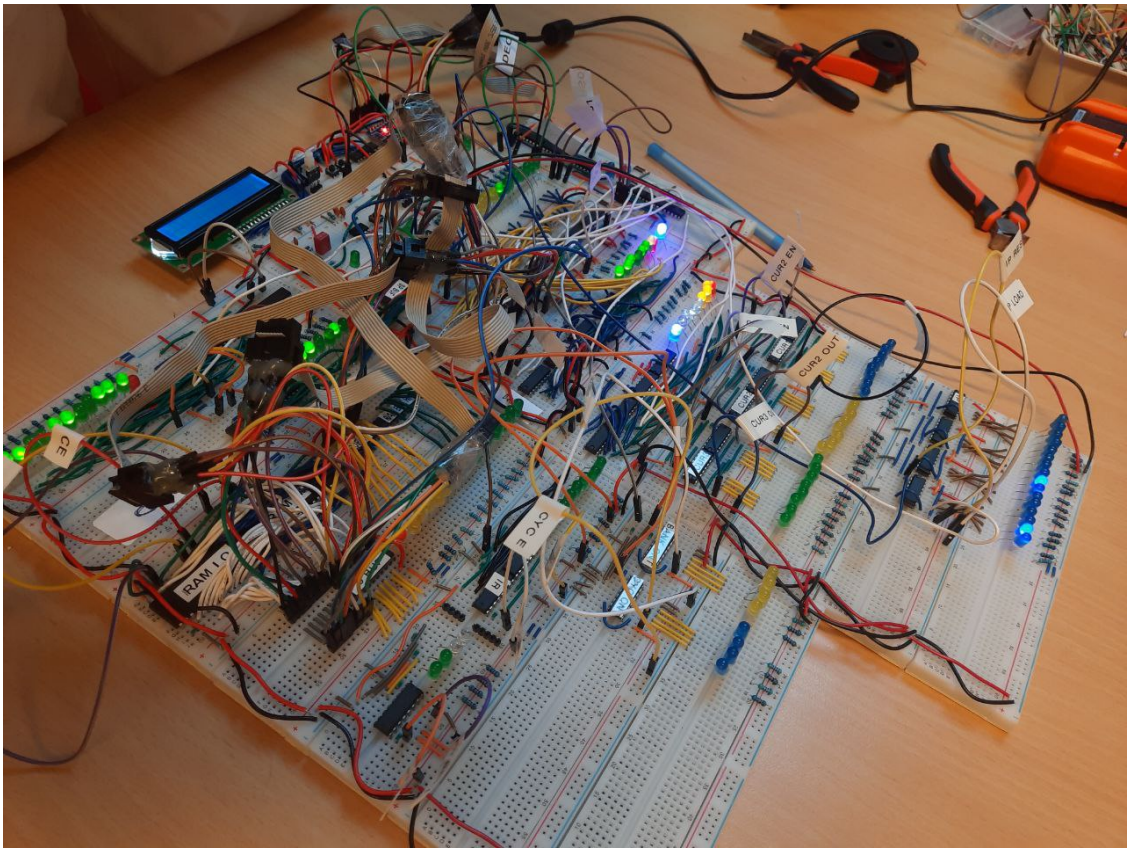


Figure 36: Photo number 10

References

[1] Wikipedia, *Brainfuck*, https://esolangs.org/wiki/Brainfuck

[2] Wikipedia, *Von Neumann Architecture*, https://en.wikipedia.org/wiki/von_neumann_architecture

[3] Ben Eater, *Build an 8-bit computer from scratch*, https://eater.net/8bit

[4] Joren Heit, *Brainfix*, https://github.com/jorenheit/brainfix

[5] All data-sheets are available online in the pdf format.