Synapse-191

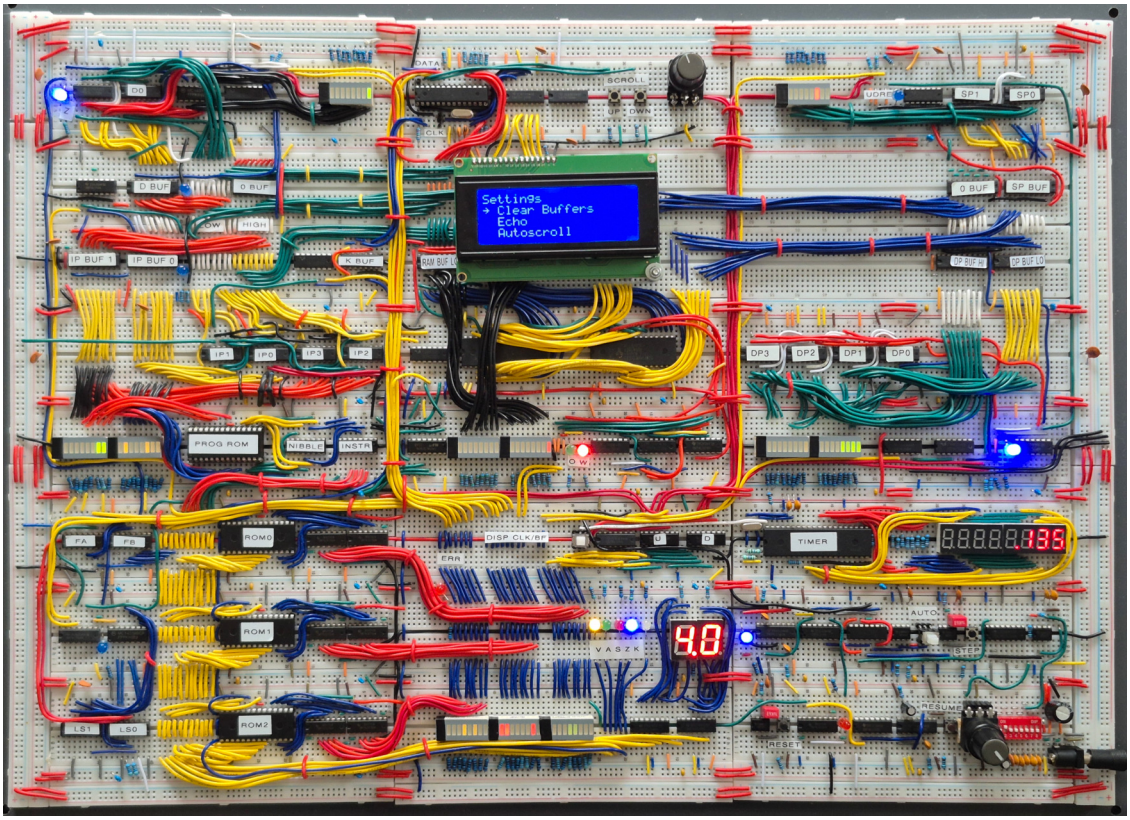
*Building A Brainf*ck Computer on Breadboards*

Joren Heit

2025

Abstract

This report documents the design and implementation of Synapse-191, a fully functional computer built on breadboards to execute the Brainfuck (BF) programming language natively. Unlike traditional approaches that interpret or compile BF code on conventional architectures, Synapse-191 treats BF as its instruction set, realized through a microcoded CPU constructed entirely from TTL logic components, RAM, and ROM—without programmable logic in the core. The system features a Harvard-like architecture, a two-phase clock capable of operating at over 200 kHz, and a modular design comprising registers, memory units, and a control unit driven by EEPROM-based microcode. A dedicated I/O module, powered by an Atmega328P, manages keyboard input, LCD output, and random number generation, providing a user interface for program selection and runtime configuration. Supporting tools—including an assembler, EEPROM programmer, microcode compiler, and emulation framework—enable efficient development and testing. The project demonstrates the feasibility of implementing an esoteric language as a native instruction set and offers insights into computer architecture, timing, and control logic at the hardware level. Outcomes, limitations, and potential improvements—such as PCB migration and extended data width—are discussed, highlighting both the educational value and practical challenges of building a CPU from first principles.



Acknowledgement

I would like to express my sincere gratitude to Artur Topal, a former student of mine, with whom this project first began. Together, we set out on this journey when he was looking for a project for his *profielwerkstuk*—a kind of final high school research project in the Netherlands. Artur played a vital role in the early stages of the design, from theoretical planning and debugging to constructing logic circuits and keeping the project's momentum alive. His sharp mind, perseverance, and enthusiasm were invaluable and have significantly contributed to the eventual success of this project.

Thank you, Artur.

Contents

1 Introduction

The Brainf*ck[1] (BF) programming language is an esoteric programming language that is essentially impossible - or at least highly impractical - to actually write useful programs in. Even if you became 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 2.

Goals The main goal of this project was to build a computer that can actually run BF code natively. Normally, after having written some new piece of BF, the programmer presents this code to some program that either compiles it to an executable native to the host architecture, or interprets it in a virtual BF machine. However, instead of treating BF as a language that requires compilation or interpretation, why not view it as the instruction set of a, yet to exist, BF-CPU?

Our goal was to build this BF-CPU without making use of any programmable chips, relying solely on Transistor-Transistor-Logic (TTL) chips such as registers, buffers and (de)multiplexers in addition to the necessary RAM and ROM. The computer was to be implemented entirely on breadboards, as it was inspired by Ben Eater's 8-bit breadboard computer [5]. It should be able to run any compliant BF program directly, as long as it fits in the program ROM and does not exhaust the available amount of memory or stack-space. In other words: the computer should be capable of running canonical BF without doing any preprocessing steps like pre-calculating jump-addresses.

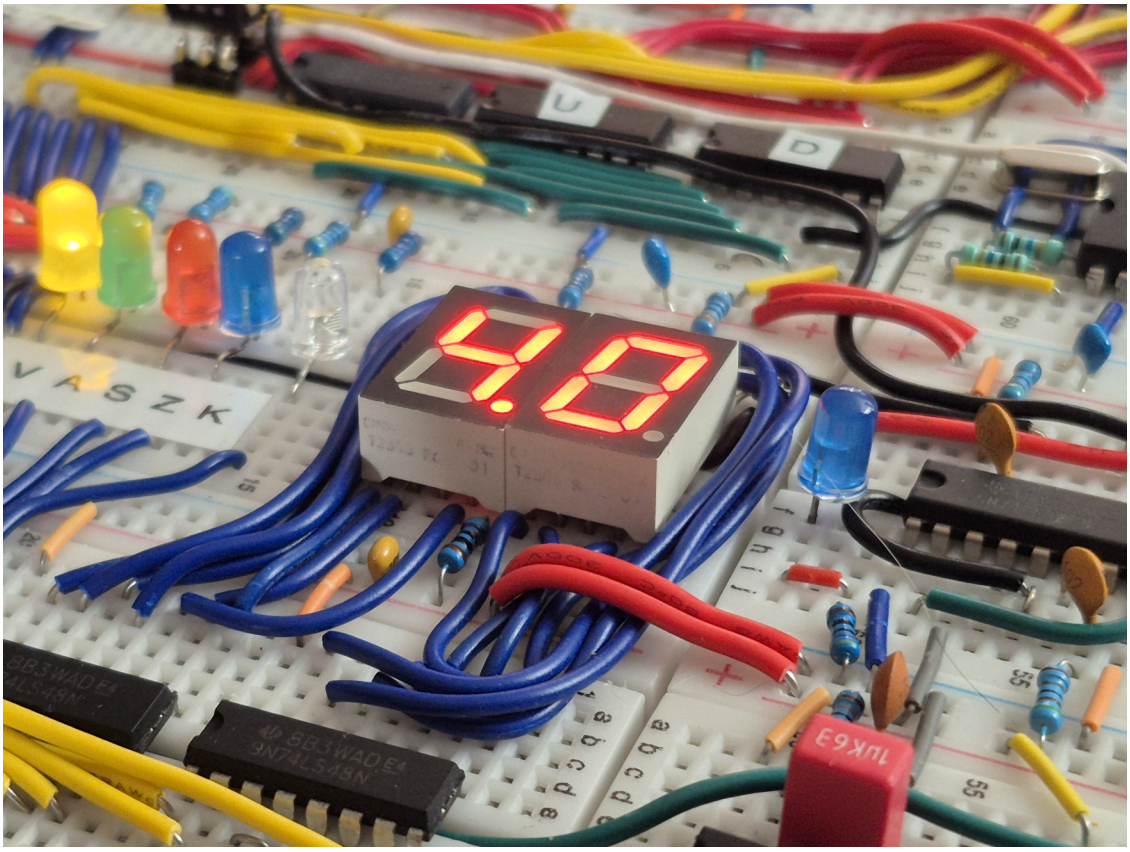
Outcome Over roughly 2.5 years (intermittently), the design evolved into a stable microcoded CPU with a Harvard-like memory map, a two-phase clock that can drive the system at over 200kHz, and a supporting software toolchain (assembler, EEPROM programmer, microcode compiler and emulation library) that makes prototyping, editing, assembling, and flashing programs and microcode practical. It has a sophisticated IO module that handles both input (both random numbers and keyboard input) and output (to a 4x20 character LCD display). This IO module is driven by an Atmega328P to be able to manage IO buffers, implement the PS2 protocol and drive the screen, in addition to managing all the IO settings (echo, autoscroll, output modes, etc) and to provide a user interface for selecting the active program at runtime. A deliberate choice was made to use a programmable chip in this case, since it is not really part of the CPU itself and makes the IO capabilities a lot more advanced and convenient. To the CPU core, the outside world is a black box that accepts certain control signals and acts accordingly; similar to the fact that the BF input and output commands (`.` and `,`) are interacting with an abstract external world.

The machine has been tested by running many different BF programs on it that can be found online, validating its stability and BF-standard conformity (if such a thing even exists). The source code for all supporting software is available on Github at `https://github.com/jorenheit/bfcpu`.

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

Document structure

- Section 2 recaps the BF programming language: how does this language work, what does a simple interpreter look like and how does it relate to the architecture of Synapse-191?
- Section 3 describes the architecture from a modular perspective: what is the purpose of each of the high-level modules, what control signals do they accept and what actions do they perform when clocked?
- Section 4 explains microcode and control sequences; how do all these signals and modules work together to perform the computations necessary to run BF programs?
- Section 5 discusses hardware implementation: what kinds of chips and techniques were used to implement the modules on a hardware level?
- Section 6 covers supporting utilities, like the assembler, microcode compiler, EEPROM programming utilities, etc.
- Section 7 goes through a number of test programs that were used during development.
- We conclude in Section 8 with a brief retrospective and some ideas for future endeavours.



2 Brainf*ck

2.1 Language Description

Brainf*ck is a popular esoteric programming language. Just like any other programming language, it allows the programmer to write programs consisting of commands that are executed in order. The key limitation is that the language provides only eight commands to the programmer, all written as a single character: "`+-<>[].,`". 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 (an infinite amount of) memory is initialized to 0 and the pointer is pointing to the very first element (index 0, see Figure 1).

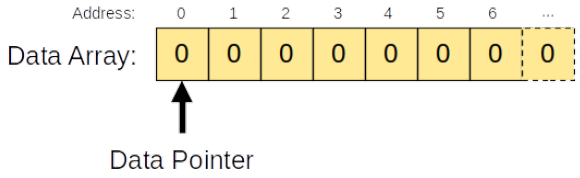


Figure 1: Initial state of the BF machine.

The commands then modify the contents of memory or 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 to the next instruction. Otherwise, skip all instructions and continue beyond the matching closing `]`;
- `]` : if the current cell is zero, exit the loop and continue to the next instruction. Otherwise, loop back to its matching opening `[`;
- `.` : send the value in the current cell to the output device;
- `,` : read a value from the input device and store it into the current cell.

Although the instruction set is minimal, it has been proven to be sufficient for performing any possible computation or program, also known as Turing-completeness [2]. The catch is that this requires an unbounded (or infinite) amount of memory, which is obviously impossible. However, the same caveat holds for traditional systems, so we should be safe to assume that BF is Turing complete for all practical purposes.

2.2 Interpreters

To run a BF program, one usually feeds these commands into an interpreter written in a more common language. These interpreters are very straightforward to write. Listing 1 shows a very basic implementation (about 40 lines) a BF-interpreter written in C. This implementation initializes a block of memory to zero and defines a pointer to its first element. This pointer can be incremented or decremented to move along the array, increment/decrement the value it's pointing to or print it to the standard output. Most of its complexity is embedded in the handling of the loop-operators. When an opening bracket is encountered, the index of this instruction is stored in a jump-table. When control reaches its matching closing bracket and the value of the cell pointed to is nonzero, this stored index is reloaded in order to return to the start of the loop. When skipping a loop, i.e. the current cell holds a zero when the opening bracket is evaluated, all commands up to and including the matching closing bracket are skipped.

```
//----------bfint_begin----------
void bfint(char const *program) {
  unsigned char mem[MEM_SIZE];
  memset(mem, 0, MEM_SIZE);
  unsigned char *ptr = mem;

  int jmp_table[JMP_TABLE_SIZE];
  int jmp_index = 0;

  int program_size = strlen(program);
  int index = 0;

  while (index < program_size) {
    switch (program[index]) {
    case '+': ++(*ptr); break;
    case '-': --(*ptr); break;
    case '<': --ptr; break;
    case '>': ++ptr; break;
    case '.': putchar(*ptr); break;
    case ',': (*ptr) = getchar(); break;
    case '[': {
      if (*ptr) jmp_table[jmp_index++] = index;
      else {
        int count = 1;
        while (count != 0) {
          switch (program[++index]) {
          case '[': ++count; break;
          case ']': --count; break;
          }
        }
      }
      break;
    }
    case ']': {
      --jmp_index;
      if (*ptr) index = jmp_table[jmp_index++];
      break;
    }}
    ++index;
  }
}
//----------bfint_end----------
```

Listing 1: Very basic implementation of a BF interpreter in C.

When the Hello World program from Wikipedia [1] is fed into the function of Listing 1, it prints out the string `Hello World!`, as expected.

```
  $ ./bfint "++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.> \
             ---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++."
Hello World!
```

2.3 Brainf*ck Architecture

Von Neumann Modern computers are built according to the von Neumann architecture [4], 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.

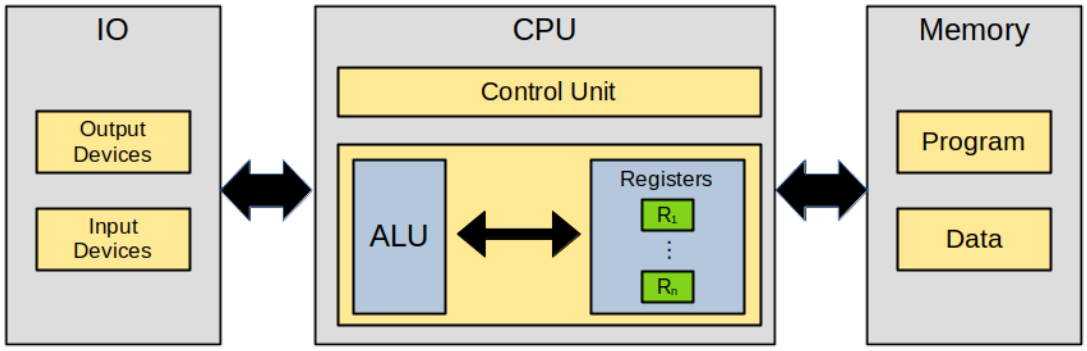


Figure 2: Schematic overview of the Von Neumann architecture.

Harvard The Harvard architecture specifies two kinds of memory: program memory and data memory (Figure 3). Program memory contains just the instructions that make up the program 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.

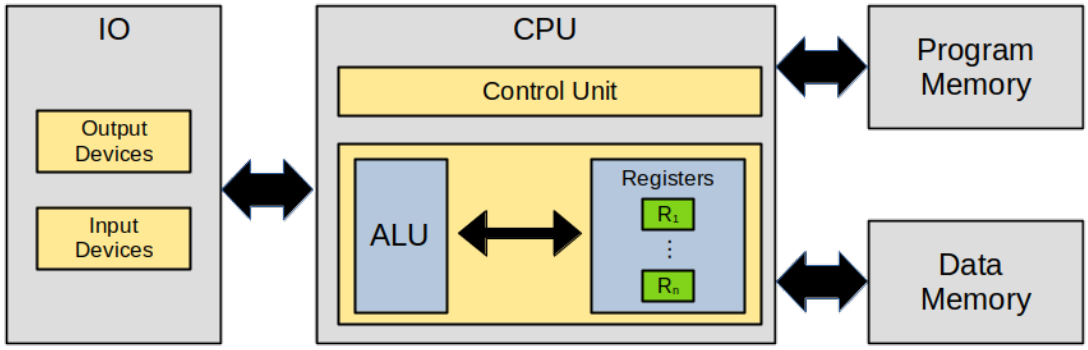


Figure 3: Schematic overview of the Harvard architecture.

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, like in Listing 1, where the program was stored in memory separate from the data. The ALU is very limited and can only increment values, decrement values and compare values to zero.

2.4 BF 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 2000 instructions implemented in hardware (depending on the way you count, [6]), whereas RISC-V processors have a fixed opcode width of only 7 bits, allowing for a maximum of 128 different opcodes [7]. Even compared to RISC, the BF instruction set is tiny even compared to the smallest instruction sets in use today. While compact,

such a small instruction set inevitably leads to less efficient execution; 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.

3 Architecture

3.1 Hardware Overview

In simple terms, a BF machine consists of an array of memory-cells and a pointer pointing to one of these cells. The pointer can move along the array while modifying its contents one step at a time. Figure 4 illustrates an example intermediate state of such a system. 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).

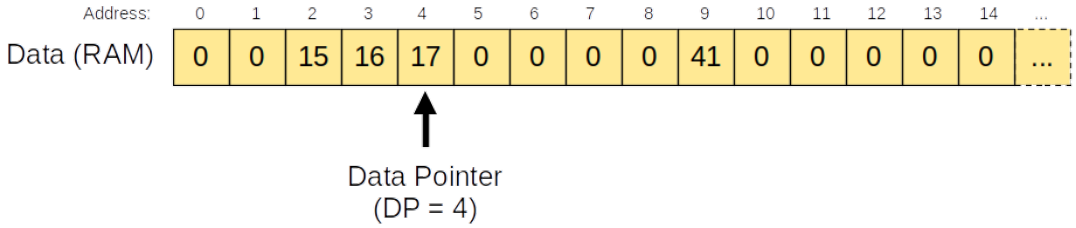


Figure 4: Example state of a BF machine.

The physical BF core needs three basic building blocks to implement the machinery described above: 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 fed from ROM into the control unit (CU) together with five flags (K, A, V, S and Z) that encode the state of the machine. Depending on the state and current instruction, the CU sets the appropriate control signals for each of the modules in order for the system to perform the appropriate actions. Figure 5 shows how each of the modules is connected to other modules. In the sections below, each of these connections will be clarified further. The actual implementation on the logic/hardware level is described in Section 5.

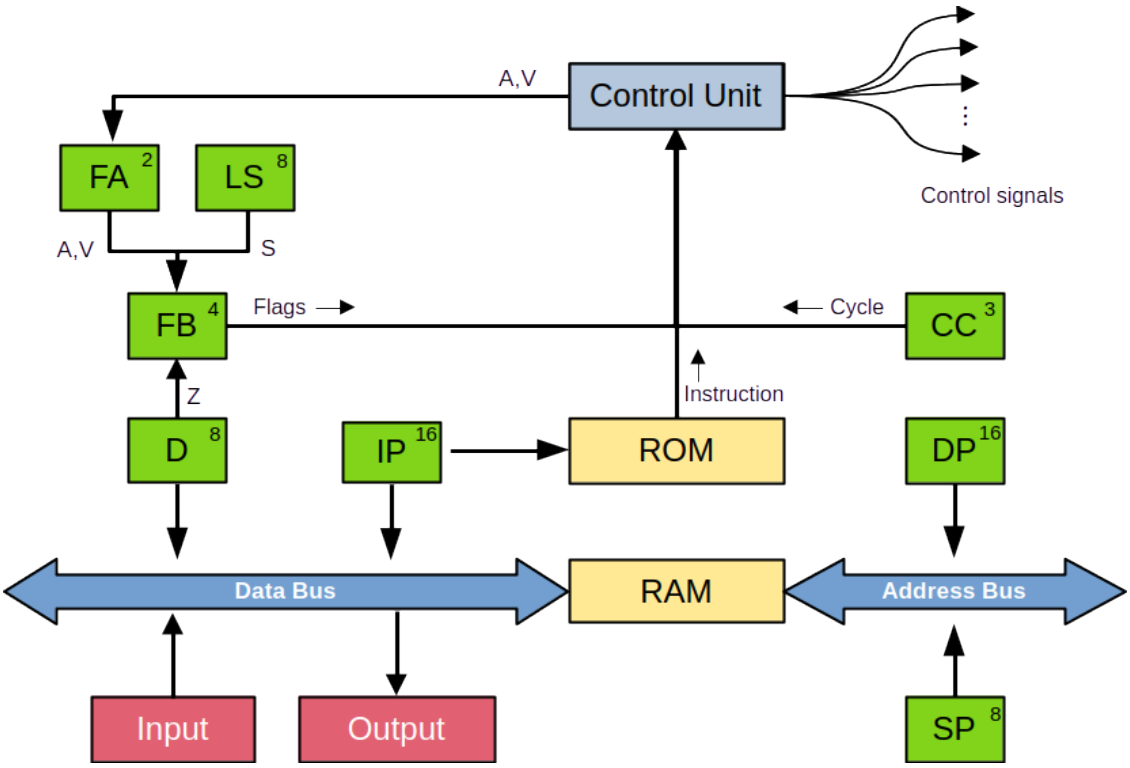


Figure 5: Connections between modules in the BF processor.

3.2 Flags

The state of the system is encoded using five flags: K, V, A, S, and Z. These flags were introduced to simplify control logic and enable performance optimizations during execution. Each flag serves a specific purpose, as explained below:

K-Flag The K-flag is used for communication between the system and the IO Module, which run asynchronously. The flag is set by the IO Module and reset by the system to let each other know when certain data transactions have been completed.

A-Flag The A-flag is set when the value in the DP register was modified, i.e. when the data pointer has been shifted to a new cell. This is an architectural optimization to minimize the number of reads from and writes to RAM. Consider a long sequence of `+` or `-` instructions, which are very common in BF programs. A naive implementation of, for example, the `+` instruction, would consist of 3 steps: read the current value from RAM into D, increment the value in D, write the result back to RAM. However, when it is known that the data pointer has not been changed, we can keep operating on the copy in D. Only when the address has been changed (A was set), should we load the new value into D before modifying it.

V-Flag The V-flag allows for another optimization, similar to that of the A-flag. Whenever the value in the D register has been modified, the V-flag is set. When the DP is about to change its value, through the `<` and `>` commands, the V-flag is checked. If set, we store the modified value back into RAM before moving the pointer. If not, the value hasn't changed and does not need to be synchronized into RAM. The number of cycles to complete long sequences of pointer move-commands is decreased substantially using this approach.

S-Flag The S-flag is used to skip over blocks of code, primarily when skipping over loop-blocks. It is set when a loop should not be entered, causing all following commands to be ignored until the matching closing

bracket is reached. Similarly, it is used at boot to skip over entire programs until the selected program is reached (when multiple BF programs have been written to the same program ROM).

Z-Flag The Z-flag is set by the D register whenever it contains the value zero. This allows control-flow to branch on the value of the current cell, allowing for the implementation of loops.

Performance Gains The number of cycles saved from using the A and V flags depends on the program that is running; programs with long and frequent sequences of `+`, `-`, `<` or `>` commands will benefit more than ones with less or shorter sequences. To investigate the actual impact, a test suite of BF programs was put together and run in an emulator (a modified version of the C example shown in Listing 1). The BF source for each of these programs is shown in Appendix D. The modified interpreter contains a table of the number of cycles necessary to execute each instruction, depending on the aforementioned flags. This allows for a quantitative comparison between the naive implementation (load, modify, store) versus the cache-like approach using the A and V flags. The results are shown in Figure 6; a 16 to 22% decrease was observerd when using the A and V flags.

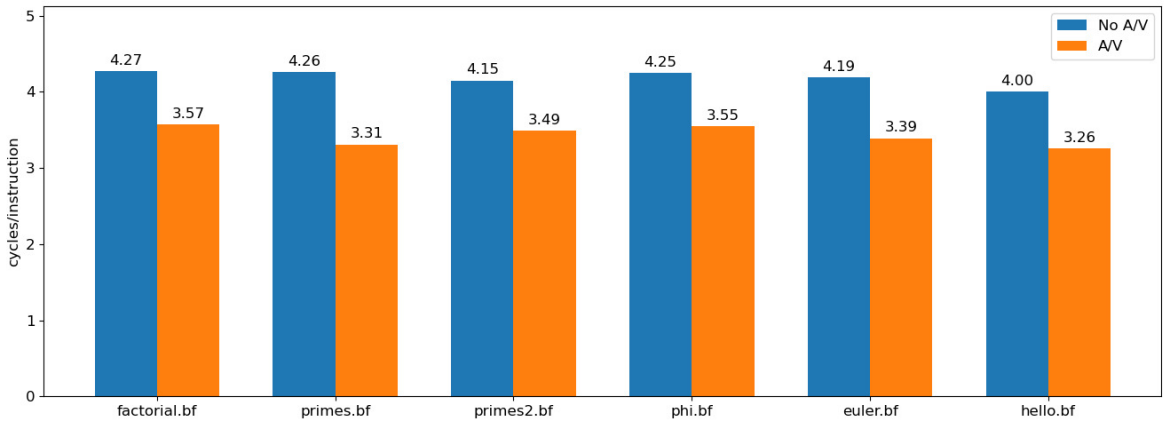


Figure 6: The number of cycles per BF instruction drops by around 20% when using the A and V flags.

3.3 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, see 3.6) 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 (see 3.4), where it can be modified before being stored back into RAM.

Inputs

The DP should be able to increment and decrement (corresponding to the `<` and `>` commands), and should be able to be enabled/disabled because of its connection to the address bus of the RAM (the Stack Pointer (SP, see 3.6, is also connected to this bus). While all other modules have the ability to be reset, the DP is the only register that can be reset (to 0x0100) at runtime. This is necessary during boot, when all the datacells need to be initialized to 0 (see 4.9).

1. EN - Enable - Assert the stored 16-bit value onto the address bus.
2. U - Up - Increment the stored value.
3. D - Down - Decrement the stored value.
4. R - Reset - Reset the value to 0x0100, the start of the datasection of RAM.

Outputs

1. DP_OUT - 16 bits, asserted onto the address bus when enabled (EN high).

3.4 Data Register (D)

The data register holds a representation of the value currently pointed to by the DP and can be incremented and decremented (using the `+` and `-` commands). This register provides the Z-flag to signify that its current value is 0. Among other things, this flag can be used to determine whether or not to enter a loop.

Inputs

1. D_IN - 8 bits - Data inputs, connected to the databus.
2. EN - Enable - Assert the stored value onto the databus.
3. LD - Load - Load data from the bus into D.
4. U - Up - Increment the stored value.
5. D - Down - Decrement the stored value.

Outputs

1. D_OUT - 8 bits - Data outputs, connected to the databus.
2. SET_Z - Set Zero Flag - High when the register stores a zero, connected to FB.

3.5 Instruction Pointer Register (IP)

The IP Register stores the instruction pointer (16-bits), which keeps track of the instruction that is currently 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 a loop is entered), its value is stored in RAM at the location pointed to by the stack pointer (SP, see 3.6). When the matching `]`-instruction is encountered, this value can be loaded back into the IP in order to jump back to the start of the loop if needed.

Inputs

1. IP_IN - 16 bits - Data inputs, connected to the databus.
2. EN - Enable - Assert the stored value onto the databus.
3. LD - Load - Load data from the bus into IP.
4. U - Up -Increment the stored value.

Outputs

1. IP_OUT - 16 bits - Data outputs, connected to the databus and the address inputs of program-ROM;

3.6 Stack Pointer Register (SP)

The stack is the first part of RAM (addresses 0x0000 - 0x00FF) and is reserved to keep track of addresses in ROM that might need to be jumped to when flow encounters a loop-end instruction (`]`). The stack-pointer (SP) points to an address in this space; it is incremented whenever a new jump-address is pushed to the stack and decremented whenever an address 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 (and each of those loops is entered). Although possible, it is very unlikely to happen for the simple programs we intend to run.

Inputs

1. EN - Enable - Assert the stack-pointer onto the address-bus.
2. U - Up - Increment the stack-pointer.
3. D - Down - Decrement the stack-pointer.

Outputs

1. SP_OUT - 8 bits - connected to the address bus of RAM.

3.7 Loop Skip Register (LS)

The Loop Skip (LS) register is a counter that indicates whether or not we're in the process of skipping 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 loop-end instruction (`]`). When it is determined that a loop must be skipped (based on the Z-flag provided by the D-register), the LS register is incremented from 0 to 1 and the S-flag is set. This flag will remain set as long as the value in LS is nonzeros, indicating that the system is in a skip-state. Subsequent instructions are then skipped until either another (nested) loop-start or a closing loop-end 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 and the S-flag is reset back to 0.

Inputs

1. U - Up - Increment the stored value.
2. D - Down - Decrement the stored value.

Outputs

1. SET_S - Skip flag - set when it value is nonzero, connected to FB.

3.8 Flag Registers (FA and FB)

FA The first flag register (FA) holds two flag values, A and V, which are used to indicate that either the address (A) or value (V) has changed during one of the previous instructions. For instance, if D was incremented (or decremented), the V-flag is set to indicate a change of the *value* being pointed to: the value in RAM is now outdated. When DP is incremented (or decremented), the A-flag is set to indicate a change of the current *address*, meaning that the value in D is now outdated. For a more detailed description of the function and application of these flags, refer to Section 4.4 and 4.5.

FB On the zeroth cycle of every instruction, these flags are latched into the FB register together with the Z and S-flags (set by the D and LS registers) for a total of 4 flags. This happens simultaneous with loading the next instruction into the instruction register (I, see 3.9) and is not refreshed until loading the next instruction to make sure that the flag-state remains constant throughout the execution of the current instruction.

K-Flag The previously mentioned K-flag is specific to interactions with the IO-module and is not buffered in either of the flag registers. It is discussed more thoroughly in Sections 4.7 and 4.8.

Inputs

1. SET_A - Assert the address-change-flag onto FA.
2. SET_V - Assert the value-change-flag onto FA.
3. LD(FA) - Load A and V into FA (if set).
4. LD(FB) - Load A, V (previously buffered in FA), Z and S (from D and LS) into FB.

Outputs

1. F_OUT - 4 bits - connected to the instruction decoder inside the Control Unit.

3.9 Instruction Register (I)

The instruction register I buffers the current instruction pointed to by the IP. The instruction is loaded from program ROM into I at the start of every new instruction (cycle 0), right after IP has been incremented. Its outputs are used as part of the microcode address that goes into the decoder of the CU (see Figure 8 in Section 4).

Inputs

1. LD(I) - Load the instruction pointed to by IP

Outputs

1. I_OUT - 4 bits - connected to the instruction decoder inside the Control Unit.

3.10 Register Driver

Rather than having a separate signal for each of the INC/DEC-inputs of each register (e.g. INC_D, INC_LS, etc), a driver module was designed (see 5.3) to drive registers that support modification of their contents (increment/decrement). In addition to a universal INC/DEC signal, three Register Select (RS) bits are used to index the target-register. This approach has two advantages:

1. It decreases the amount of control signals needed;
2. The logic needed to drive the counting registers (74LS193) only needs to be implemented once.

The driver module accepts 5 control signals: 3 register-select signals (RS0 through RS2), INC and DEC. Using 3 register-select signals, up to 8 (2^3) registers can be selected, though only 5 need to be driven by the driver. Table 1 contains an overview of each of the registers and the control signals they support. When all RS-signals are off, no register is selected.

Register	#Bits	EN	LD	INC	DEC	RS2|RS1|RS0
D	8	x	x	x	x	0|0|1
DP	16	x		x	x	0|1|0
SP	8	x		x	x	0|1|1
IP	16	x	x	x		1|0|0
LS	8			x	x	1|0|1
FA	4		x			not addressable
FB	4		x			not addressable
I	4		x			not addressable

Table 1: Control signals available on each of the registers. The flag and instruction registers are not connected to the register driver.

Inputs

1. RS0 - Register Select Bit 0
2. RS1 - Register Select Bit 1
3. RS2 - Register Select Bit 2
4. INC - Increment selected register
5. DEC - Decrement selected register

Outputs

1. U - 5 bits - Up signals - Connected to the U input of all registers that support the INC operation.
2. D - 5 bits - Down-signals - Connected to the D input of all registers that support the DEC operation.

3.11 Cycle Counter (CC)

Almost every BF instruction requires multiple cycles to complete. Therefore, in addition to the instruction and state, a cycle counter is used to determine the control signals that should be sent out at each step of the instruction. This cycle counter is implemented as a 3-bit counting register (allowing for at most 8 cycles per instruction) that increments on every clock cycle and sends its output to the control unit. Its only control signal is the clear signal (CLR) which resets the count to 0, in order to fetch the next instruction.

Inputs

1. CLR - Cycle Reset - Reset the count to 0;

Outputs

1. CC_OUT - 3 bits - Current value of the register (0-8).

3.12 Data Memory (RAM)

RAM is divided into two parts: stack and data. The first 256 bytes (0x0000 - 0x00FF) make up the stack and are indexed by the stack pointer (3.6). The data (corresponding to the BF tape) is stored at addresses 0x0100 through 0xFFFF and are indexed by the data pointer (3.3). Its address lines are connected to the address bus, which in turn receives its value from either the DP or the SP. Its data lines are bidirectional and are connected to the data bus. When the Write Enable (WE) signal is active, data can be read from the bus and written into RAM. When instead the Output Enable (OE) signal is active, the current value in RAM (determined by the address on the address bus) is asserted onto the data bus.

Inputs

1. DATA_IN - 16 bits - Input data, connected to the databus.
2. ADDR_IN - 16 bits - Address lines, connected to the address bus;
3. OE - Output Enable - Assert the value stored at the current address onto the databus;
4. WE - Write Enable - Write the value on the databus into the current address.

Outputs

1. DATA_OUT - 16 bits - Output data, connected to the databus (same physical lines as DATA_IN).

3.13 Program Memory (ROM)

The actual BF instructions are stored in Read-Only-Memory (ROM) and are addressed by the IP (3.5). A 4-bit instruction is stored at the address pointed to by the IP. It is sent to the CU where it is used to determine the set of control signals, together with the flags and cycle counter.

Inputs

1. ADDR_IN - 16 bits - Address lines, connected to the IP.

Outputs

1. INS_OUT - 4 bits - Instruction data, connected to the CU.

3.14 Screen (SCR)

The output module (which is assumed to be a screen) will be attached to the data bus and will display whatever is on the bus when enabled using the EN signal. Because the output is handled asynchronously by some peripheral that will, from the perspective of the CPU, be viewed as a black box, it needs a flag to acknowledge a successful data-transfer. This is done through the K-flag, which is set by the peripheral after reading data from the databus (this flag is shared with the input device for a similar purpose). The K-flag can only be reset by the CU using the CLR_K signal, indicating that the transfer procedure has been completed.

Inputs

1. DATA_IN - 8 bits - connected to the data bus.
2. EN: Enable - Display the contents of the bus. The format of the output (ASCII, hex, etc) may vary depending on the implementation of the output device.
3. CLR_K - Clears the K-flag - connected to the CU (this signal is shared with the input module (KB, see 3.15).

Outputs

None.

3.15 Keyboard (KB)

The input device to the computer is assumed to be a keyboard of some sort[2], that implements a buffer from which some 8-bit value can be requested. The control unit can assert the enable-signal of this device and should then wait until the K-flag is set, at which point the data should ready to be read from the databus. It is left up to the implementation of the peripheral to decide what to do when there is nothing in the input-buffer (either wait for user input or return 0). Shared with the output-peripheral, the K-flag is reset to communicate that the data has been transferred and the input-device can yield control of the databus back to the system.

Inputs

1. EN - Enable - Make the contents of the input buffer available on the databus.
2. CLR_K - Clears the K-flag - connected to the CU (this signal is shared with the output module (SCR, see 3.14).

Outputs

1. DATA_OUT - 8 bits - Output data, connected to the databus.

[2]At a later stage in the process of building the CPU, a Random Number Generator (RNG) was implemented as a second input device, to support a common Brainf*ck extension, where the `?`-command is used to generate a random number in the currently active cell.

3.16 Control Unit

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 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, depending on the current instruction and state determined by the flags and cycle counter. The implementation details of how this is done in hardware are discussed in Section 5.

Inputs

1. CC_IN - 3 bits - Cycle counter input lines.
2. INS_IN - 4 bits - Instruction input lines (from program ROM).
3. FLAGS_IN - 5 bits - Flag input lines (from FB and K).

Outputs

1. HLT - Halt Clock
2. RS0 - Register Select, bit 0
3. RS1 - Register Select, bit 1
4. RS2 - Register Select, bit 2
5. INC - Increment selected register
6. DEC - Decrement selected register
7. CLR_DP - Reset DP
8. EN_SP- Enable SP to address bus [3]
9. OE_RAM - Output Enable RAM
10. WE_RAM - Write Enable RAM
11. EN_IN - Enable input (keyboard) to databus
12. EN_OUT - Enable output device
13. SET_V - Set V-flag in FA
14. SET_A - Set A-flag in FA
15. LD_FBI - Load FB and I
16. LD_FA - Load FA
17. EN_IP - Enable IP
18. EN_D - Enable D to databus
19. LD_D - Load D
20. LD_IP - Load IP
21. CLR_CC - Reset Cycle Counter
22. CLR_K - Clear the K-flag in the IO module
23. END - Program End Signal
24. ERR - Error Signal

[3]Note that there is no EN_DP since this signal is mutually exclusive with EN_SP; whenever one is set, the other is unset and vice versa.

4 Opcodes and Control Sequences

A control sequence is a sequence of control signals being sent out to various modules on subsequent clock cycles to implement a specific opcode. This chapter will go through each of the control sequences that implement the BF instruction set, including supporting instructions. All sequences have been summarized in Table 12 for quick reference (Appendix A).

4.1 Binary Format

In Section (3.1) it was mentioned that the BF program is stored in ROM as a series of opcodes, each of which corresponds 1-to-1 to a BF command. However, the system needs to perform some additional non-BF actions to prepare itself for execution. For one, the data-tape is assumed to be zero-initialized; a freshly powered on SRAM chip will contain random garbage, so we need to take care of that before the first BF command is executed. Moreover, the system is designed to accept a ROM chip containing multiple programs for convenience. This means that before blindly running the first program, there must exist some mechanism that looks for the index corresponding to the program the user is intending to run. To account for these issues, the binary format on the EEPROM expected by the system is as listed below. The `bfasm` utility was designed to translate any canonical BF source to the corresponding opcodes and dresses them with the necessary housekeeping instructions. The control sequences that implement the initialization and bootloading steps are further discussed in Section 4.9.

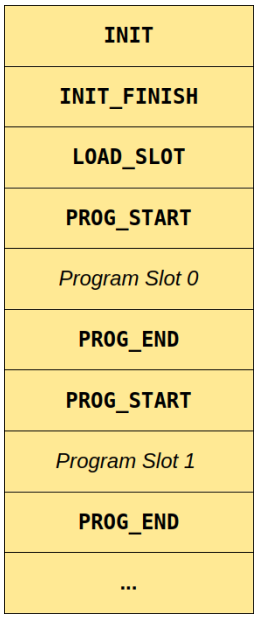


Figure 7: Binary format expected by the system: initialization, bootloading and the BF programs themselves.

1. One or more INIT instructions to (partially) initialize the data-tape.
2. One INIT_FINISH instruction to return the datapointer to the start of the datasection. This instruction also signals to the IO module that the system has been fully initialized.
3. One LOAD_SLOT instruction. The program index is supplied to the bus by the IO module and used to move the IP to the start of the corresponding program.
4. A sequence of BF programs, each preceded by a PROG_START instuction and ended by a PROG_END instruction. The PROG_START instructions serves as a barrier between programs and is used to skip over programs in order to find the correct program to run. The PROG_END simply halts the clock and lights up an LED to indicate successful termination of the program.

4.2 Instruction Decoding

By setting the control signals as described in Section 3 appropriately, modules can work together to perform each of the BF and supporting instructions. The Control Unit implements this as a lookup-table in 3 ROM chips, where the instruction taken from program ROM (4 bits), flags (5 bits) and cycle counter (3 bits) act as an address into this table (Figure 8). Given that the CU has to be able to supply a total of 24 different control signals, three 8-bit EEPROM chips have been used to store the microcode lookup-table for a maximum of 24 signals (all of them used). More details on the implementation can be found in Section 5.11.

No simultaneous INC/DEC signals It is important to note that, because of the choice of driving all of the (counting) registers with a common interface (the Register Driver, see Section 3.10), only one register can be driven per clock cycle. In other words, the INC and DEC signals can be applied to only one register at a time.

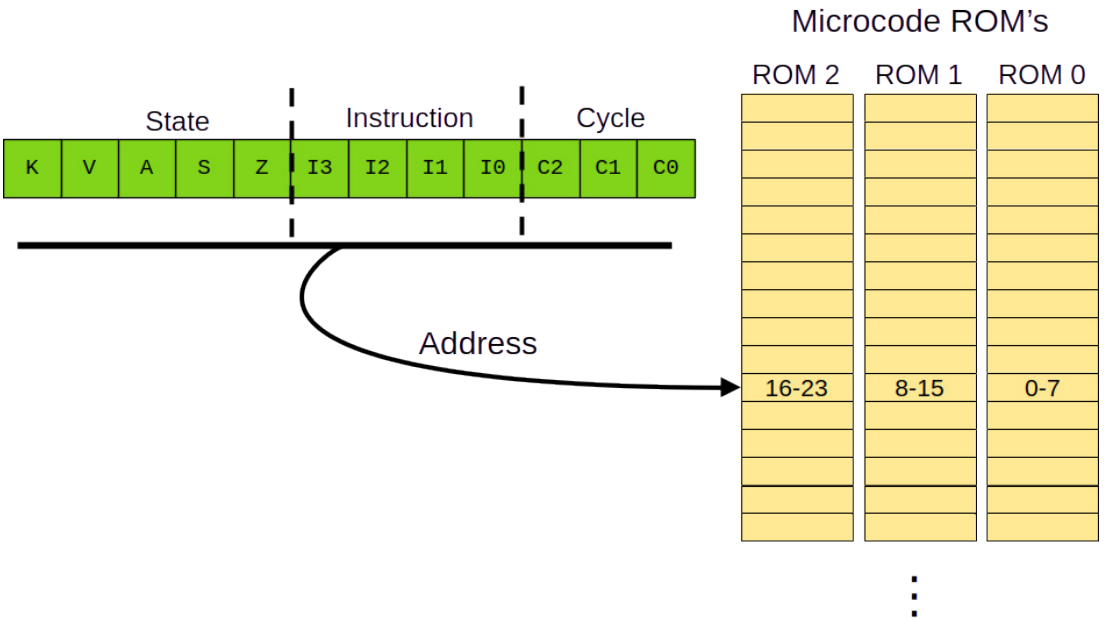


Figure 8: Decoding an instruction: the current instruction, flags and cycle-count are used as an index to the three ROM chips that output the control signals corresponding to the current state of the system.

4.3 Cycle 0: Instruction Fetch

The first cycle of each instruction is identical[4]: A, V, S and Z are loaded into the FB register and the current instruction (pointed to by IP) is loaded from program ROM into the instruction register (I). This provides the CU with all the necessary information to determine the control signals for cycle 1: LD_FB, LD_I.

4.4 Modifying Data: + and -

PLUS: The sequence of instructions necessary to execute a + command depends on the state of the system. Three different scenario's have to be taken into account:

Scenario 1: (A = 0, S = 0) - If the A-flag (address-change-flag) is not set, the value in D already corresponds to the value currently pointed to by the DP and no synchronization has to be performed. In that case, its INC signal is immediately asserted to the D register in order for it to increment on the next clock pulse. Referring to Table 1, we see only RS0 has to be asserted in conjunction with the INC signal to increment D (register address 0b001). The V-flag must also be set in order to indicate that the value in D has been changed: this is done by asserting the SET_V signal to FA and latching in the value using the LD(FA) signal. Now that the value has been incremented and the corresponding flag has been set, the IP is incremented using the register-driver on cycle 2. The cycle reset signal is asserted at the same time.

Cycle 1: INC, RS0, SET_V, LD_FA

Cycle 2: INC, RS2, CR

Scenario 2: (A = 1, S = 0) - However, when the A-flag *was* set, this means that the address has recently changed and the value inside D does not correspond to the value pointed to by the DP in RAM. We therefore need to fetch the current value from RAM by enabling the DP register on cycle 1, enabling the RAM to output its content on the databus and loading the resulting value into D. From hereon, the control signals are identical to those described above in the case where A was not set.

[4]The OUT instruction is slightly different but this can be ignored for now. For more information, refer to 4.7

Cycle 1: EN(DP), OE_RAM, LD(D)

Cycle 2: INC, RS0, SET_V, LD_FA

Cycle 3: INC, RS2, CR

Scenario 3: (S = 1) - None of the actions above need to be performed when the S-flag is set, 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 and reset the cycle counter: INC, RS2, CR.

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

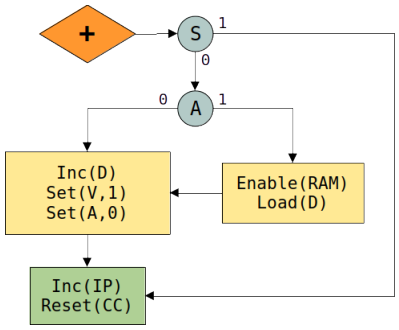


Figure 9: Block diagram for the `+` command. The diagram for the `-` command is equivalent (using `Dec` rather than `Inc`).

4.5 Moving the Pointer: < and >

RIGHT: Moving the data pointer one cell to the right requires similar instructions compared to PLUS instruction, the difference being that we increment the DP-register rather than the D-register. Similarly, we consider three different scenario's, branching on the V-flag instead of the A-flag:

Scenario 1: (V = 0, S = 0) - If the V-flag is not set, it means that the value we point to hasn't changed and we don't need to care about synchronization. The DP (register address 010) is incremented immediately and the A-flag is set to indicate we changed the address and are now out of sync. In the second cycle, we move to the next instruction and reset the cycle counter.

Cycle 1: INC, RS1, SET_A, LD_FA

Cycle 2: INC, RS2, CR

Scenario 2: (V = 1, S = 0) - In the case that V *was* set during one of the previous instructions, we need to write the updated value (present in the D-register) back to RAM before moving the pointer. This is achieved by enabling the value in D onto the databus and setting the RAM module to write-mode. Furthermore, the V-flag needs to be cleared. This is achieved by loading FA without setting any signals; this will effectively reset both A and V back to zero.

Now that the RAM contains the updated value, it is safe to move the DP to the next cell. The control sequence to do this is identical to the sequence described in the (V = 0)-scenario.

Cycle 1: EN_D, WE_RAM, LD_FA

Cycle 2: INC, RS1, SET_A, LD_FA

Cycle 3: INC, RS2, CR

Scenario 3: (S = 1) - None of the actions above need to be performed when the S-flag is set, 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 and reset the cycle counter.

Cycle 1: INC, RS2, CR

LEFT: The control signals necessary to perform the < command are similar to those of the > command, the only difference being the DEC signal to perform a subtraction rather than addition.

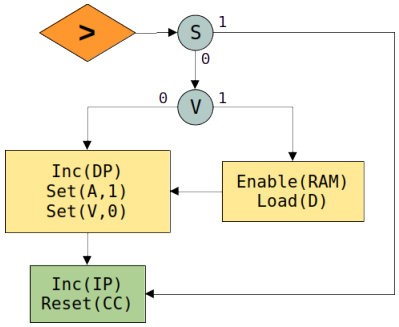


Figure 10: Block diagram for the > command. The diagram for the < command is equivalent (using `Dec` rather than `Inc`).

4.6 Conditional Jumping: [and]

These are by far the most complicated instructions that require lots of additional logic. Because the BF instruction set lacks a JMP-instruction where some 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. When a loop is skipped, the LS (Loop Skip) register is used to determine when execution should resume. This leads to multiple scenario's depending on the state of A, Z and S.

LOOP_START:

Scenario 1: (A = 0, Z = 1, S = 0) - In the first scenario, where A is not set (the D-register is up-to-date) and the Z-flag is set, we can immediately conclude that this loop should be skipped. Hence, the LS-register is incremented and the next instruction is loaded (to be ignored until the LS-register becomes 0 again). Since LS is addressed by the register driver at address 101, both RS0 and RS1 need to be asserted to the register driver. In the second cycle, the IP is incremented and the cycle-counter is reset to move to the next instruction.

Cycle 1: INC, RS0, RS2

Cycle 2: INC, RS2, CR

Note that these instructions could not take place in the same cycle due to the limitation of the register driver, which can only increment one register per cycle.

Scenario 2: (A = 0, Z = 0, S = 0) - 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 (the current value is nonzero). It takes 3 cycles to do so: increment the stack-pointer (cycle 1), write the current IP (address 011) to this address on the stack by enabling (cycle 2) and move to the next instruction (cycle 3). The corresponding control sequences are therefore:

Cycle 1: INC, RS0, RS1

Cycle 2: WE_RAM, EN_SP, EN_IP

Cycle 3: INC, RS2, CR

Scenario 3: (A = 1, S = 0) - In the third scenario the A-flag *is* set, which means that we should first load the current value from RAM into the D-register (cycle 1) and reset the A-flag. The cycle count is immediately 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 (either scenario 1 or 2, depending on the value of Z).

Cycle 1: OE_RAM, LD_D, LD_FA, CR

Scenario 4: (S = 1) - In the final scenario, we are in the process of skipping code, indicated by the S-flag (S = 1). In this case, we have encountered a nested loop that needs to be skipped over, so we 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). The control sequences are therefore identical to those in the first scenario:

Cycle 1: INC, RS0, RS2

Cycle 2: INC, RS2, CR

LOOP_END:

Scenario 1: (A = 0, Z = 1, S = 0) - In the first scenario, which takes 2 cycles to execute, there is a known (synchronized) zero in the D-register (A = 0). 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.

Cycle 1: DEC, RS0, RS1

Cycle 2: INC, RS2, CR

Scenario 2: (A = 0, Z = 0, S = 0) - In the second scenario, there is a known nonzero value in D. This means we must loop back to the IP-value 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). In the second cycle, this new IP (pointing to a `[`) is incremented to re-enter the loop.

Cycle 1: EN_SP, OE_RAM, LD_IP

Cycle 2: INC, RS2, CR

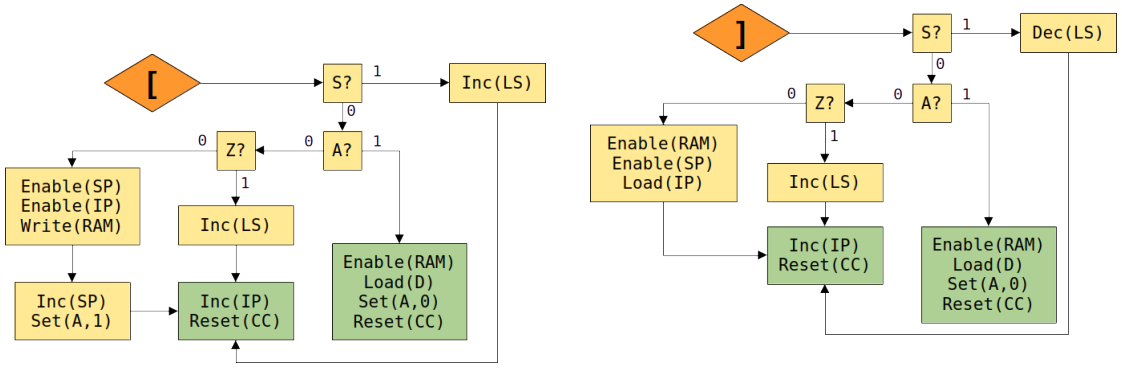
Scenario 3: (A = 1, S = 0) - In the third scenario, the contents of D are not yet synchronized with the RAM, so we first need to load it in. After loading the value into D, the flags and cycle counter are reset to put the system back into one of the previously defined states.

Cycle 1: OE_RAM, LD_D, LD_FA, CR

Scenario 4: (S = 1) - Finally, when already in the process of skipping a loop, the LS-register is decremented before moving to the next instruction before resetting the cycle counter and incrementing the IP.

Cycle 1: DEC, RS0, RS2

Cycle 2: INC, RS2, CR



(a) Block diagram for the loop-start command.

(b) Block diagram for the loop-end command.

Figure 11

4.7 Output: .

Handshake: Due to the asynchronous nature of the output peripheral, it is necessary to enter into a handshake protocol whenever a byte is put on the bus for display. In this protocol, the K-flag is used to communicate between the CPU and the IO module: it is set by the IO module to indicate that it has read the data from the bus, and reset by the CU to indicate that the handshake has been received and completed:

Step 1: The CU asserts the EN_OUT signal and enables the module that contains the current value (either D when A = 0 or RAM when A = 1), such that its contents appear on the databus:
EN_OUT, (EN_D or OE_RAM).

Step 2: The output module is notified of this through the EN_OUT signal and reads this value from the bus. When done, it sets the K-flag.

Step 3: After enabling the data on the bus, the CU repeatedly checks the K-flag. When it becomes 1, the CU knows that the data has been successfully taken from the bus, at which point the supplying module (D or RAM) can disable its outputs. The CU then finalizes the handshake by resetting the K-flag. As soon as the output module notices that the K-flag was reset, it starts waiting for its next instruction.

Scenario 1: (K = 0) - CR, (EN_D or OE_RAM)
Scenario 2: (K = 1) - CLR_K, INC, RS2, CR

Cycle Zero: As mentioned briefly before, the OUT instruction is the only instruction that has a slightly different cycle 0 control sequence associated with it. This is because the waiting-loop is implemented by simply resetting the cycle count, such that the instruction is loaded in again but may now branch differently depending on the value of K, which might change in the meantime. However, the data on the bus should remain valid for the entire duration of the loop, since the output module may read from it asynchronously. To accommodate for this, the 0-cycle for the OUT instruction enables either D or RAM, depending on the value of the A-flag. This does no harm whenever the sequence is encountered outside of the loop, but does keep the data valid during one.

Cycle 0: LD_FB, LD_I, (EN_D or EN_RAM).

4.8 Input: ,

Handshake: Similar to the output command, the input command implements a handshake protocol to make sure that the databus is claimed by the input-device for the exact right amount of time, in order for the system to reliably read its contents. This happens using the same K-flag that was used in the output handshake.

Step 1: The CU asserts the EN_IN signal to let the input-device know that it can claim ownership of the bus.

Step 2: The input-devices receives the EN_IN signal and puts a value onto the bus (see below for the difference between immediate and buffered input-mode). When the data is ready, it sets the K-flag.

Step 3: After asserting the EN_IN signal, the CU waits for the K-flag by continuously resetting the cycle counter and branching on the value of K. Once that becomes high, it loads the data into the D register and sets the V-flag:

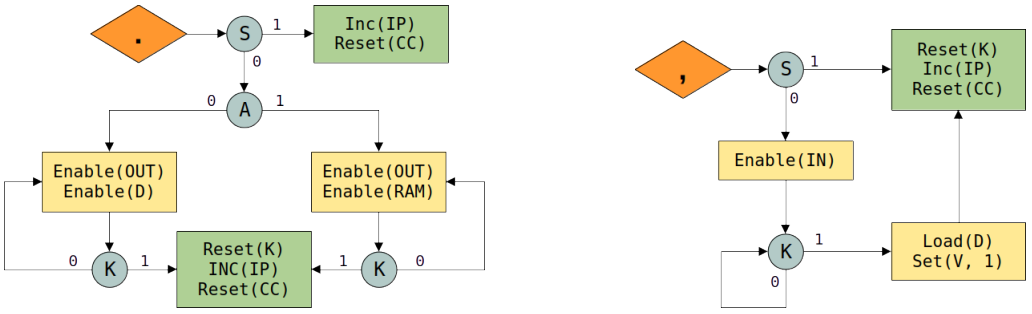
Scenario 1: (K = 0) - CR
Scenario 2: (K = 1) - LD_D, SET_V, LD_FA

Step 4: To finalize the handshake, the K-flag is cleared by the CU by setting the CLR_K signal.

Input-Modes: The input peripheral (probably) manages an internal buffer to serve subsequent bytes from to the system, which might be empty when the request for input arrives. When this happens, it is up to the peripheral to decide upon one of 2 options:

1. Wait for the buffer to contain data. Only then set the K-flag (buffered mode).
2. Assert zero's to the bus and set the K-flag immediately (immediate mode).

In our implementation of the IO-module (see 5.12), both modes are supported and can be selected from in the options menu.



(a) Block diagram for the output command.

(b) Block diagram for the input command.

Figure 12

4.9 Initialization and Bootloading

In Section 4.1 a general overview of the contents of the program ROM was given. Before running the program, two main steps have to be taken: initialize the data-tape to zero and jump to the correct program (the EEPROM can contain multiple programs). These two steps are implemented using the INIT, INIT_FINISH, LOAD_SLOT and PROG_START opcodes.

INIT: In any BF program it is assumed that all memory is zero-initialized. In practice, SRAM-modules will contain random values at startup, so the assembler must add a preamble to the main code in order to initialize the RAM (or at least part of it) to 0. While this could in principle be implemented in terms of canonical BF commands (initializing one cell at a time using a sequence of `[-]` commands) it is much faster to write a bunch of known zeroes directly to RAM. This is the purpose of the INIT instruction: for each INIT instruction, a contiguous chunk of 256 memory-cells will be zero-initialized. Since it is guaranteed that the D register contains a zero after reset, this value can directly be written into RAM on the first cycle of INIT while also incrementing the LS and DP registers. DP is incremented to move through the memory-space whereas LS is incremented in order to count to 256, at which point the S-flag will go low again due to the LS register overflowing back to 0. If more memory needs to be initialized, the assembler can simply concatenate multiple INIT instructions.

Cycle 1: EN_D, WE_RAM, INC, RS0, RS2 (write a zero to the current cell)

Cycle 2: LD_FBI, INC, RS1 (increment the LS-register)

Cycle 3: Depending on whether the LS-register wrapped around (256 cells initialized), either move to the next cell or finalize by incrementing the IP and resetting the cycle-counter.

Scenario 1: (S = 0) - INC, RS2, CR

Scenario 2: (S = 1) - CR

After the appropriate number of INIT instructions have been execured, the INIT_FINISH instruction (see below) must be called in order for the DP to return back to the start of the memory (0x0100).

INIT_FINISH: The INIT_FINISH opcode is designed to do two things: it resets the DP to the start of the data section (0x0100) and it resets the K-flag. The former is necessary after initialization, which leaves the DP at whatever the last cell was that was initialized, and is achieved by asserting the CLR_DP signal. The latter will tell the IO module that initialization is finished (the IO module sets the K-flag at boot).

The IO module will respond to this by taking ownership of the bus and writing the program slot to it. The LOAD_SLOT instruction (which is always immediately following the INIT_FINISH instuction) will use this value to skip to the desired program.

LOAD_SLOT: As described above, the IO module will have put the program slot (set by the user through the IO menu) onto the bus by the time the LOAD_SLOT opcode is executed. To avoid race conditions, we wait for the K-flag to go high, meaning that the data is ready to be read from the bus. Once it does, we leverage the existing loop-skip mechanism to skip over entire programs rather than loops inside a program. First, we load the program index from the bus into D and increment the LS register to go into skip-mode (LS now has value 1 so the S-flag goes high). We then clear the K-flag to let the IO module know that it should disable its outputs. Since the system is in skip now, every normal BF instruction will be skipped entirely.

Cycle 1: LD_D, INC, RS0, RS2 (load slot index and increment LS)

Cycle 2: CLR_K, INC, RS2 (clear the K-flag and increment IP)

PROG_START: The only opcode that performs non-trivial actions while the system is in skip-mode (initiated by incrementing LS as described above) is PROG_START. It checks the value in D, which was initially populated with the index of the desired program (0, 1, 2, ...), to see if the IP has arrived at the correct location: if the Z-flag is set (D = 0), we exit skip-mode and enter the program following the current PROG_START instruction. If the Z-flag is not set (D $\geq$ 0), we decrement D and keep scanning. In other words, D is decremented on each PROG_START until it hits zero, i.e. the selected program slot has been reached.

Scenario 1: (S = 1, Z = 0) - D still holds a nonzero value. Decrement D and continue in skip-mode:

Cycle 1: DEC, RS0

Cycle 2: INC, RS2

Scenario 2: (S = 1, Z = 1) - D is zero. Exit skip-mode (decrement LS) to run subsequent code.

Cycle 1: DEC, RS0, RS2

Cycle 2: INC, RS2

Note that, while skipping over entire programs, the LOOP_START and LOOP_END instructions will still have the effect of incrementing and decrementing LS. However, since LS starts out at 1 and programs are guaranteed to contain matching brackets (enforced by the assembler), it is guaranteed to contain the value 1 when the entire program is skipped over (it will have undergone an equal amount of increment and decrement operations). Decrementing it a final time (Cycle 1 of Scenario 2) is therefore guaranteed to bring it back to 0, disabling the S-flag.

4.10 Convenience Opcodes

While all of the opcodes above would be sufficient to run any BF program, it is still convenient to be able to halt the clock, indicate that an error occurred or tell the user that the end of the program was reached succesfully. Furthermore, the common extension *Random Brainf*ck* [3] is implemented as an additional instruction (RAND) that acts just like the IN instruction, except now a random number appears on the bus rather than user input (handled by the IO module as well). All non-BF opcodes are listed and described below.

NOP: The NOP instruction does nothing. It simply increments the IP and resets the cycle count to move to the next instruction.

Cycle 1: INC, RS2, CR

HALT: The `HALT` instruction halts the clock and (temporarily) stops the program by asserting the HLT signal. The assembler will place a HALT instruction before the first and after the final instruction of each program. The former allows the user to manually start the program after the system has been fully set up and the latter prevents the program from continuing into invalid memory after it has executed its final command. Furthermore, the assembler can (optionally) interpret an exclamation mark (`!`) as a `HALT` in the BF-code to set breakpoints for debugging. When the system is resumed and cycle 2 of the HALT instruction is reached, the IP is incremented as usual in the final cycle of any instruction.

Cycle 1: HLT

Cycle 2: INC, RS2, CR

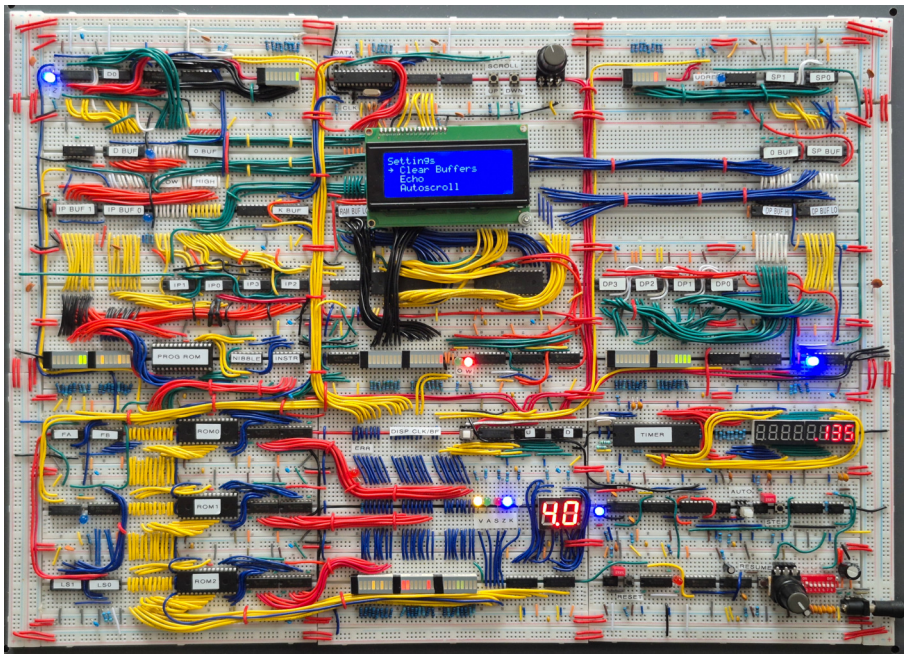
PROG_END: The PROG_END instruction is basically a HALT instruction that also lights up an LED by asserting the END signal. This let's the user know that the end of the program was reached gracefully. Contrary to the HALT instruction, no Cycle 2 has been defined. Trying to resume the clock after a PROG_END was reached will therefore result in the ERR signal being raised (see below).

Cycle 1: HLT, END

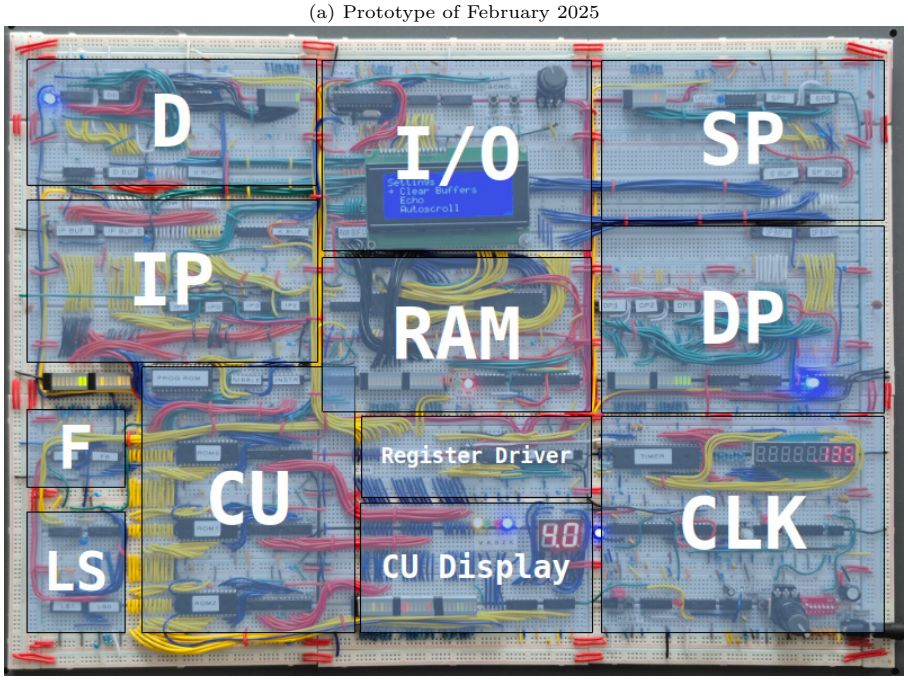
Error States: There is no opcode to put the system into an error state explicitly. However, every address in the microcode table that corresponds to an unspecified state contains the HLT and ERR signals to stop the clock and light up the error LED. The system should never go into such a state, no matter what errors the BF program contains. However, it proved invaluable during testing, when the microcode sequences themselves still contained mistakes.

5 Implementation

This section will discuss the implementation of each module and the way they integrate together to make the computer. Figure 13a shows the computer as it was in February 2025. The overlays shown in Figure 13b show where each of the modules described in previous sections is located on the computer. Schematics for each of the modules can be found in Appendix E.



(a) Prototype of February 2025



(b) Location of the modules on the prototype

Figure 13

5.1 Wiring

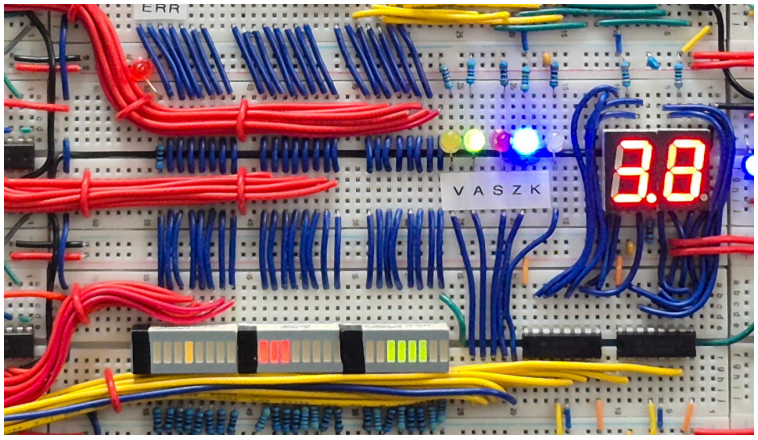


Figure 14: Close up of the wiring inside the CU.

5.1.1 Breadboards

The entire system was implemented on 830 hole breadboards. When the project started, relatively cheap breadboards were used, which had segmented power rails. These had to be bridged with jumper wires but these connections proved somewhat unreliable, introducing subtle points of failure. This is why in a later stage of the build we moved to more expensive but more reliable breadboards with continuous power rails.

5.1.2 Wires

Most of the longer wires were cut from spools of either 22AWG or 20AWG wire. While 22AWG proved reliable enough almost all the time, the thicker 20AWG had noticeably stronger connections to the breadboards and has been used mainly to make reliable power and ground connections. Thinner jumper wires have been used for short connections, e.g. connections from IC's VCC and GND pins to the power rails. These are very convenient to use - cutting and stripping short wires is a tedious job - but have sometimes been the source of hard-to-identify instabilities that were eventually identified as bad connections to ground, caused by these jumper wires.

5.1.3 Busses

In Ben Eater's implementation of the 8-bit CPU, power rails cut loose from breadboards were used to build the central bus, which makes it easy for modules to connect to it. Because our system has two busses (8 and 16 bits wide), this would have been impractical to replicate.

5.1.4 Bus Pull-Down

The data bus can be written to by 3 diffent modules: D, IP and the input device (usually KB). To prevent floating bits at times when none of these devices are active, each of the 8 bus-lines is connected to ground through a 1K resistor. Even though the microcode implementation prevents any module from reading from the bus when no other module is enabled, this was done anyway in the spirit of good practice. The 16-bit address lines are guaranteed to be in a valid state since either the DP or the SP has its outputs enabled at any time during execution. Therefore, no pull-down resistors were used on the address bus.

5.1.5 Power

Power is supplied by a 5V, 3A power supply (the system draws around 930mA), connected power rails that run around the perimeter of the system and in between the breadboards. As many interconnections between different parts of the power rails were made to ensure that all segments of the board receive a stable power

supply and connection to ground. At short intervals, 100nF capacitors were placed across the rails to filter high-frequency noise.

5.1.6 Clock and Reset

Many of the modules need a connection to the clock and reset lines, which is why power lines cut loose from spare breadboards were repurposed to function as clock/reset rails that run around the perimeter of the board. This provides easy access to those lines even for modules at the opposite end of the system relative to the clock and reset modules.

5.1.7 LED Indicators

To be able to monitor the state of the machine visual (and for dramatic effect), many LED indicators have been installed across the board. LED bars were used to visualize the register contents and control signals, while single LED's were used to show the status of flags and enable-signals. Each LED is wired in series with a 470Ω resistor to ground.

Additionally, two 7-segment displays, each driven by a 74LS48, are connected to the instruction register and cycle counter to display the current instruction and cycle as decimal numbers (e.g. 4.2 for cycle 2 of the `>` instruction, see 6.1). Unfortunately, the '48 does not support values over 9 (no hexadecimal representation) so non-BF opcodes like INIT lead to a blank display.

5.2 Clock and Controls

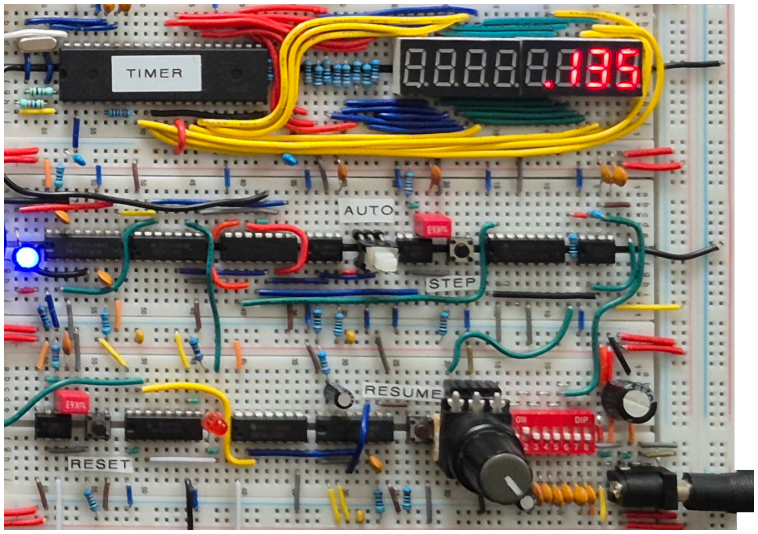


Figure 15: Close up of the Master Clock and Reset/Resume Modules.

5.2.1 Clock

The clock module is located at the bottom right of the computer and is responsible for providing a heartbeat to (most of) the modules. The core design of the clock, based around a 555 timer in astable mode, is taken directly from Ben Eater's 8-bit computer video's [5]. The output frequency can be set using an array of DIP-switches to select the capacitor of an RC-circuit for coarse control and a 10K linear potentiometer for fine control. Two additional 555 timers are used to debounce both the pushbutton for the manual clock and the latching push button which acts as a select between the two modes, as per Ben's design.

The frequency of the astable 555 is halved by sending it through a JK flip-flop to ensure a perfectly symmetric duty cycle, then fed into a 74LS123 monostable multivibrator to produce two short 200ns pulses: one on the rising and another on the falling edge the output of the flip-flop. The 200ns pulse is generated by connecting a 100pF capacitor and a 5K resistor to the '123, in accordance with the timing diagram from the datasheet [25], shown in Figure 16.

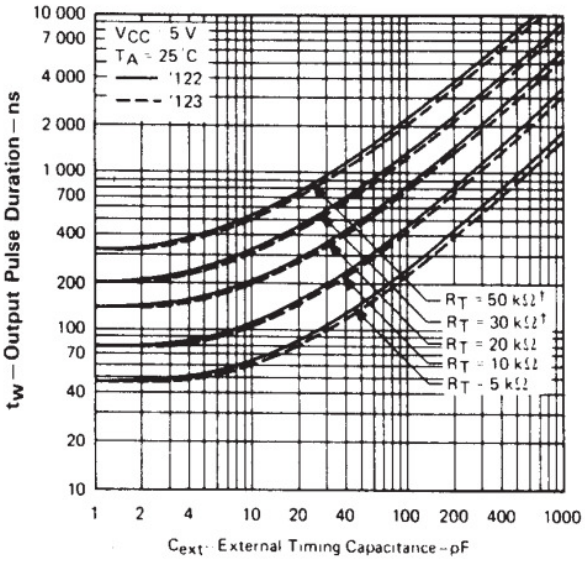


Figure 16: Pulse width of the 74LS123, based on the connected RC values. Taken directly from [25], page 7.

This results in two sets of clean signals at constant intervals. On the first pulse (rising edge of the output of the flip-flop), control signals are loaded from the microcode EEPROMs into a set of registers (74LS173) that buffer these control signals for stability; even when the inputs to the EEPROM address-pins change during execution of an opcode, this will not affect the control signals presented at the modules. The second pulse (generated by the falling edge of the flip-flop) is used as a clock to the modules; this is when the modules execute their command, like loading a value into RAM or incrementing the contents of a register. This approach guarantees a clean division between setting the control signals and clocking the modules. Figure 17 shows the timing diagram for the different signals discussed above.

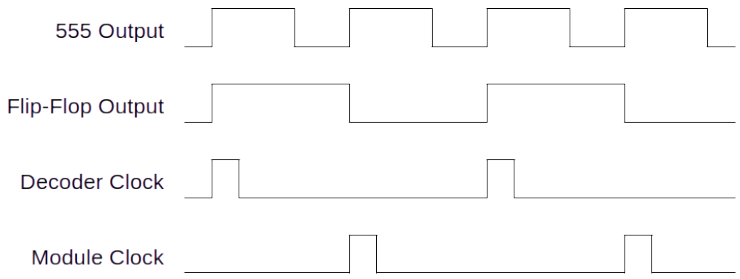


Figure 17: Timing diagram for the clock signals.

Frequency Control The frequency of the master clock can be set using DIP switches to select the capacitor-value and a 10K potentiometer to select the resistance of the RC circuit connected to the 555 timer. The capacitor, in conjunction with a fixed 2K resistor, sets a broad range (lower capacitance corresponding to higher frequencies) while the potentiometer is used fine-grained selection of the frequency within this range. This potentiometer is wired in series with a 1K resistor to ensure stability when the potentiometer resistance drops toward zero. Table 2 shows the frequency-ranges available for each of the currently selected capacitors. These values have been measured *after* the flip-flop (so the actual frequency of the 555 timer is around double the frequencies displayed in table 2). Having a broad range of frequencies available makes it possible to run at very low speeds for educational purposes, or at very high speeds for complicated, long running algorithms.

Capacitance (F)	f_{min} (Hz)	f_{max} (Hz)
10^{-5}	5	15
10^{-6}	40	140
10^{-7}	500	1,500
10^{-8}	5,000	14,000
10^{-9}	19,000	55,000
10^{-10}	38,000	108,000
10^{-11}	90,000	270,000

Table 2: Frequency ranges for each of the capacitors (approximate).

Frequency Display To be able to see the clockfrequency as well as the instruction-frequency (number of BF instructions executed per second), the module clock (M_CLK) and INC(IP) signals are connected through a switch to the input of an ICM7226B timer chip [37], which is configured as an 8-digit frequency timer. It drives two 4-digit 7-segment displays at a 1 second interval (the frequency is measured and updated every second).

5.2.2 Reset/Resume

Reset The Reset/Resume module is located directly underneath the clock and contains logic necessary to reset the computer (necessary after applying power) or resume the clock after it has been halted. The HLT signal coming from the decoder is latched into a register (74LS173) from which the corresponding output bit is connected to the HLT input of the clock module. When the system is reset (using the reset button) or when the resume button is pressed, the HLT bit is cleared and the clock output is enabled again. This allows for pausing and resuming the computer, effectively adding breakpoints to the code. The reset button itself is debounced in the same way as the manual clock button to ensure a stable transition with a debounce time of around 300ms.

Power on Reset A Power-on-Reset (POR) mechanism has been implemented to let the system reset itself when first powered on. This removes the necessity of resetting the system manually after connecting power, to make sure it is in a valid state before running the first command. A RC circuit is connected to the input of a Schmitt Trigger (74LS14). This has the effect of temporarily pull the input low before going high once the capacitor has charged to a sufficient potential for triggering the inverter. During this time, the output of the trigger is high; this output is connected through an OR gate (74LS32) to the reset line, together with the output of the manual reset button. The RC circuit consists of a 22k resistor and a $20\mu F$ capacitor for an RC-time of around 400ms. Given that the Schmitt Trigger transitions at a voltage U_T between 1.5V and 2.0V, and potential across the capacitor rises with time according to

$$U(t) = U_{vcc}\left[1 - \exp\left(\frac{t}{RC}\right)\right],$$

the duration of the reset pulse can be calculated using

$$t_R = RC \cdot \ln\left(\frac{U_{VCC}}{U_{VCC} - U_T}\right)$$

to be between 150ms and 225ms. This gives the system plenty of time to settle into its reset state.

Resume The Resume button needs more sophisticated debounce circuitry due to the following scenario: when multiple HLT instructions are seperated by a relatively small amount of other instructions, a pulse in the order of milliseconds (like the reset and pulse debouncers) will be far too long at high clock frequencies. The resume-signal will still be high when a second (or third, fourth, ...) HLT instruction is encountered, causing control flow to simply skip over these instructions. To remedy this situation, a debouncing circuit is required that first produces a pulse of equal width of the clock pulses (Figure 17), followed by a guaranteed period where the signal is low, even when the button bounces after the pulse. This is achieved by creating

a feedback loop between the two monostable vibrators present on the 74LS123. The first one will produce a 200ns pulse on the rising edge of the button. This pulse is sent to the reset of the register that holds the HLT signal in order to clear it, but is also connected to the second monostable vibrator. When the initial (short) pulse goes low, the second vibrator generates a much longer pulse that is connected to the reset-input of the first one, making sure it cannot be re-activated for some time.

To determine the RC-values for the '123, we aim for a cool-down period of around 1 second. The diagram from the datasheet is suitable for only short (up to 10 μs) pulses, so instead we use the formula from the same page to find suitable values:

$$t_W = 0.28 \times RC \left(1 + \frac{0.7}{R}\right)$$

Here, R is measured in $k\Omega$ and C is measured in pF. The result t_W is the pulse width measured in ns. Setting $R = 680k\Omega$ and $C = 4.7\mu F = 4.7 \times 10^6 pF$ (both commonly available values for these components), the factor between brackets nearly vanishes to 1, yielding a pulse-width of

$$t_W = \frac{0.28 \times 680k\Omega \times 4.7 \times 10^6 pF}{10^9 \text{ns/sec}} = 0.89s$$

By selecting a 680K resistor and a $4.7\mu F$ capacitor, a cool-down period of around 1 second is achieved. Figure 18 shows a timing diagram to illustrate this process in more detail.

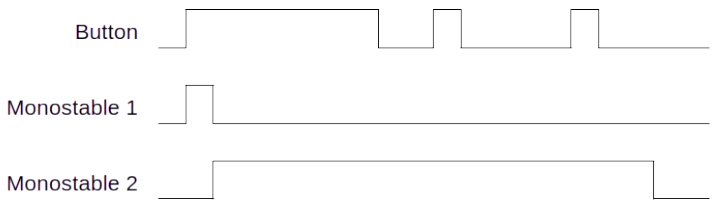


Figure 18: Timing diagram for the resume debouncer. The output of Monostable 1 is connected to the reset of the register that stores the HLT signal.

5.3 Register Driver

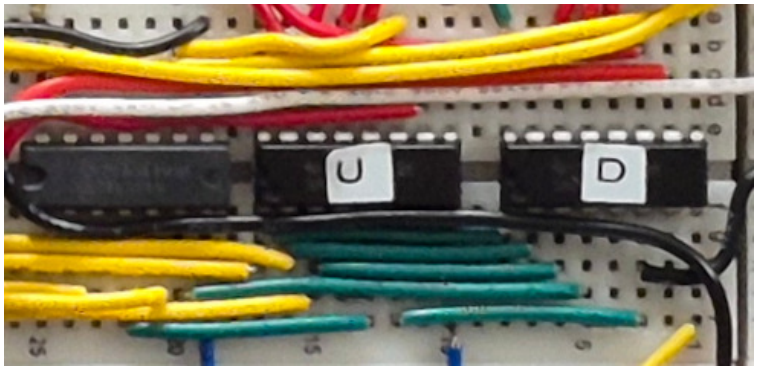


Figure 19: Close up of the Register Driver Module.

The register driver is responsible for driving the U- and D-inputs of the 74LS193 counting registers that are used to implement the D, DP, SP, IP and LS register modules. To increment the '193, its D-input needs to be held high while providing a low pulse to the U-input. As explained in Section 3.16, a centralized driver was used to limit the number of logic IC's necessary to drive the registers and the total number of control signals necessary.

Decoders The driver module uses a pair of 3-to-1 decoders (74LS138): one to drive the U-inputs and the other to drive the D-inputs of the '193. The '138 takes 3 address bits (A, B, and C) to select one of 8 outputs (Y0-Y7), which will be pulled low when selected (all other outputs remain high). Two gate-inputs G1 (active high) G2 (active low) are used to enable the outputs of the chip; the selected output is activated (pulled low) only when both gates are active. This is very convenient given the fact that the 74LS193 needs a low pulse to increment or decrement its value:

- The register-select signals RS0, RS1 and RS2 are connected to A, B, and C to select the required output.
- The INC and DEC signals are connected through inverters to the G2 gate.
- The module-clock signal is connected to G1: when pulsed, the selected output will produce a pulse that is effectively an inverted clock pulse (high-low-high) which is exactly what the '193 expects.

5.4 DP Register

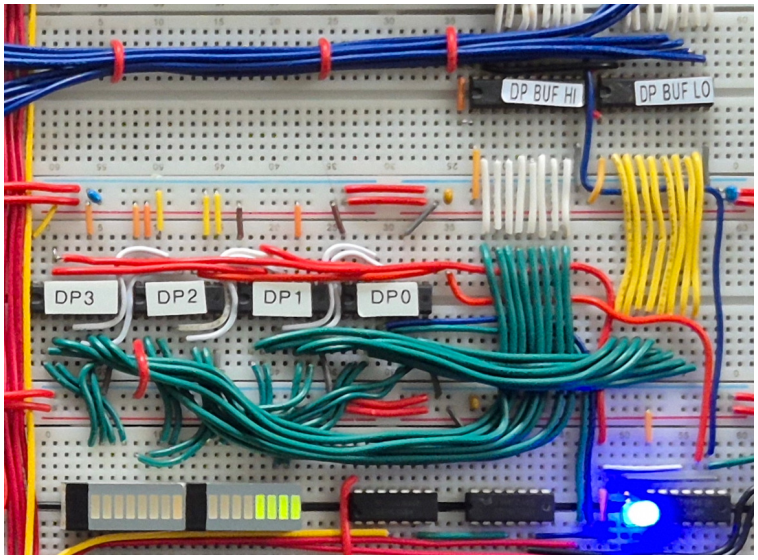


Figure 20: Close up of the Data Pointer Register Module.

The DP Register Module is the module that is responsible for managing the data-pointer; it contains a 16-bit value that is connected to the address bus of the RAM and points to the memorycell currently pointed to by the BF-pointer. The value is stored across four 74LS193 binary counters that are chained together (each holding 4 bits), making it possible to address a total of $2^{16} = 65,536$ different memory cells. The DP is connected to the register driver at address 2 (0b010, see Table 1) and as such can be incremented or decremented when the program hits a > or < respectively. The outputs of each '193 are connected to the address bus through a pair of tristate buffers (74LS245) to prevent bus contention with the stack pointer; see below (Enabling Ouput) for more information on the enable-signal.

Reset Vector Since the data section in RAM starts at 0x0100 (0x0000 through 0x00ff are reserved for the stack), this is the value that the register should start at right after booting up the system (all other registers start out with an initial value of zero). To achieve this, the global reset line of the system is connected to the reset pin of the 193's corresponding to nibbles 0, 1 and 3, but to the load-pin of nibble 2, who's inputs have been hardcoded to 0b0001 (0x1). This register is also special in the sense that it is the only register that needs to reset at runtime (through the CLR_DP signal), without resetting any of the other modules. After all, after initializing its memory to 0 by looping through (part of) its addressable space, the DP needs to be brought home to the start of the datasection before the main program starts (see also Section 4.9). The global RESET signal is therefor OR'd with the CLR_DP signal before going to the reset (and load) pins of the IC's.

Enabling Output Perhaps somewhat confusingly, the schematic (see below) shows that the SP_EN signal is used to enable the buffers. Because the DP shares the address-bus with only the stack pointer (SP) -and their outputs should be mutually exclusive- the same signal can be used to enable and disable their respective buffers: when the stack pointer is enabled, the data pointer should be disabled and vice versa. Given that the output-enable-pin of the 74LS245 is active low, the SP_EN signal can be fed directly into the enable-pin of the DP buffers. On the side of the SP, the same signal goes through an inverter before going into the enable-pin of its respective buffer. By default, when the SP is not enabled, the DP will provide its address to RAM. The address-bus will therefore never be left floating, which has the nice side-effect of always being able to visually see the current value in RAM by the LED's connected to its outputs.

5.5 D Register

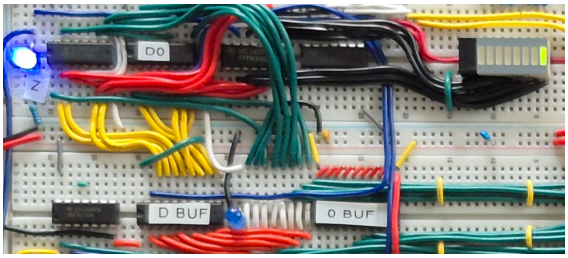


Figure 21: Close up of the Data Register Module.

The data register (D) holds (a copy of) the value in memory currently pointed to by the datapointer (DP). In the computer, it is located in the top left corner. Like the DP, it is implemented using 74LS193 counting registers and driven by the register driver described in Section 5.3 at address 1 (001, see Table 1). Since the data is only 8-bits in size, no more than two '193 chips have to be chained together to create the 8-bit register.

Enabling Output The outputs of the '193s are buffered by a single 8-bit tristate buffer (74LS245) before being connected to the databus. Because the databus is 16-bit wide (necessary to store IP-values on the stack), the high-byte is set explicitly to 0 when D is enabled by a second buffer that always outputs zero's. Storing nonzero values in the high-byte of the datasection would not have any consequences for the computation, but would be visually confusing. The buffers are set to output-mode only (even though the register is able to read from the bus as well) because the 193 chips have seperate pins for incoming and outgoing data. The incoming data is read from the bus directly without needing to go through a buffer.

Z-Flag This module also produces the Z-flag, indicating that it is currently containing the value 0. This is achieved by connecting its outputs through an 8-input NOR gate (MC14078B). The output of this gate is then connected to the FB flag register where it can be latched in by the CU in order to determine the next course of action.

Loading Data Because the '193 loads asynchronously, the clock has to be gated with the LD_D signal through a NAND gate in order to load synchronously with the clock when the LD_D signal is high (the load-pin on the '193 is active low). The necessity of a NAND gate meant it was easier to also implement any inverters needed in the circuit in terms of NAND gates.

5.6 IP Register

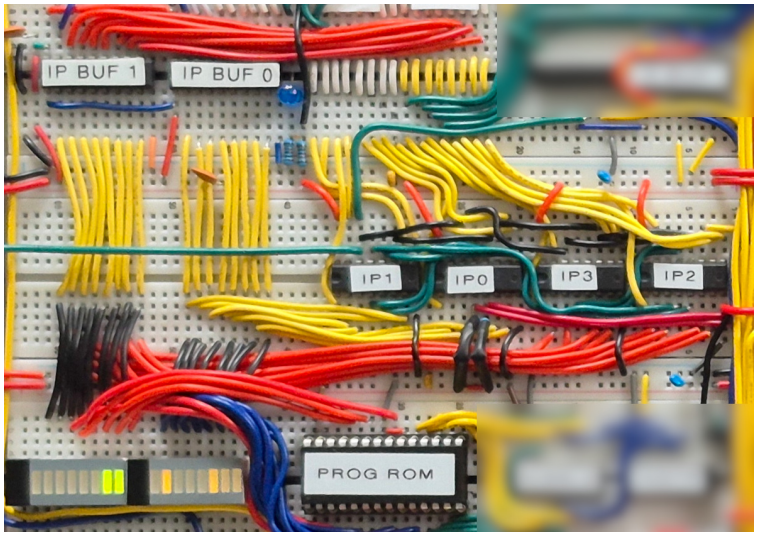


Figure 22: Close up of the Instruction Register Module.

The instruction pointer register holds a 16-bit value representing the address of an instruction in the program-ROM (implemented using an EEPROM chip (AT28C64B)). The size of the available address space in program-memory is 2^{14} instructions, so the two uppermost bits (bits 14 and 15) of the IP are left unused.

Forbidden Decrement The IP, like the D and DP registers, is driven by the Register Driver at address 4 (100), but should in principle never be decremented; it either moves to the right (next instruction) or jumps back by loading a value from the databus. Its inability to move left (decremented) is not enforced by the hardware itself, but should be taken care of by the microcode implementation.

Reading and Writing Data The IP is connected to the databus through two tristate buffers (74LS245) to avoid bus contention with the DP and IO-module. It is connected to this bus in order to write its value to the stack when a loop is entered. When exiting from a loop, a value is read back into the register through a direct connection to this bus (without going through a buffer). Because loading is done asynchronously on the '193, the load signal is NAND'ed with the clock to make loading synchronous again.

5.7 SP Register

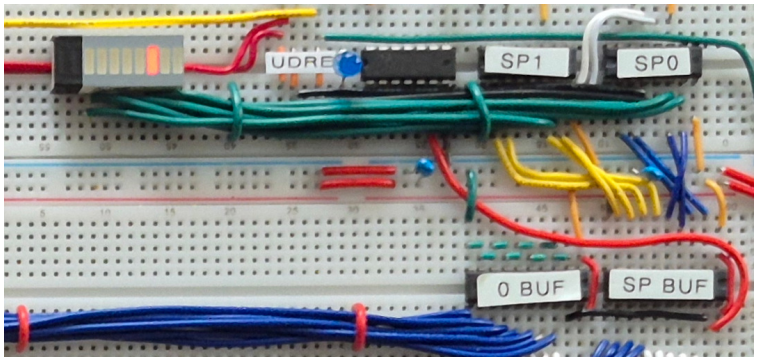


Figure 23: Close up of the Stack Pointer Register Module.

The stack pointer (SP) holds an 8-bit value in the range 0x00 - 0xff, which corresponds to addresses within the stack-space of RAM, where IP values can be stored and loaded from when the system sees the `[` and `]` loop-instructions. When a loop is entered, the IP register stores its value on the stack at the address pointed to by the SP. The SP then increments its value, ready for the next value to be stored on the stack when a nested loop is encountered. It is therefore implemented using the 74LS193 binary counter and connected to the register driver at address 3 (011). The SP module is connected to the same RAM address bus as the DP, which means it should go through a tristate buffer to avoid bus contention. As mentioned before (5.4), the SP buffer shares its enable-line (though inverted) with the DP. A second buffer that, when enabled, only outputs zero's on the address-bus is used to make sure that only the stack is addressed by the SP and no accidental reads or write happen in the data section.

5.8 LS Register

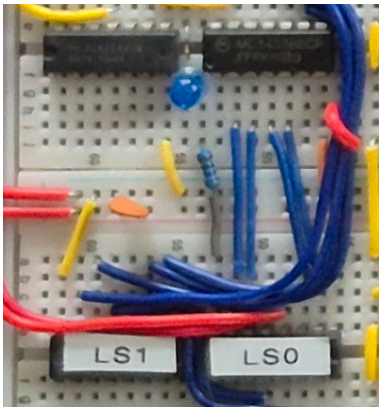


Figure 24: Close up of the Loop Skip Register Module.

The Loop Skip Register (LS) is used to produce the loop-skip-flag (S, see 3.7). It is implemented using two 74LS193 binary counters and is connected to register driver at address 5 (101). Like the Z-flag, the S-flag is produced by sending the outputs of the binary counters through an 8-input NOR-gate, the output of which is then inverted and connected to the FB flag-register. When any of these bits are high, the S-flag will be raised, indicating that the computer is in the process of skipping the current loop.

5.9 Flag Registers

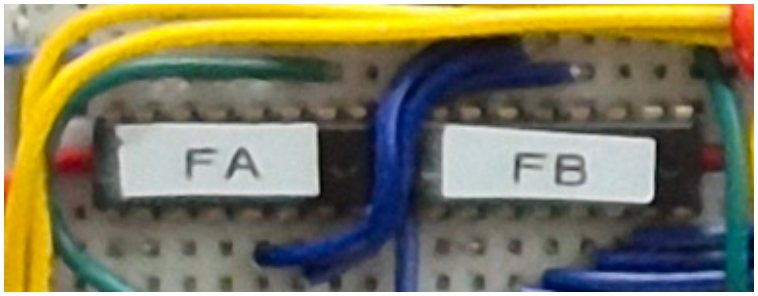


Figure 25: Close up of the flag registers FA and FB.

There are two stages of flag-buffering; the first stage is the FA register (in which the A and V-flags are stored) and the second the FB register (in which all flags except K are latched when the instruction is loaded). Both of these registers have been implemented using a 74LS173 4-bit register.

FA The A and V-flag are stored in FA by the CU whenever the value in D is changed (V) or whenever the pointer is changing its position (A). This can happen mid-instruction without changing the address on the microcode EEPROMs.

FB At each cycle 0, the A and V-flags are loaded from FA into FB, together with S and Z coming from the LS and D registers respectively. Since this always happens in conjunction with loading the instruction from program ROM into the instruction register (see 5.11), the control signal has been named LD_FBI. The flags in the FB register will (mostly) remain constant during the execution of an opcode.

5.10 RAM

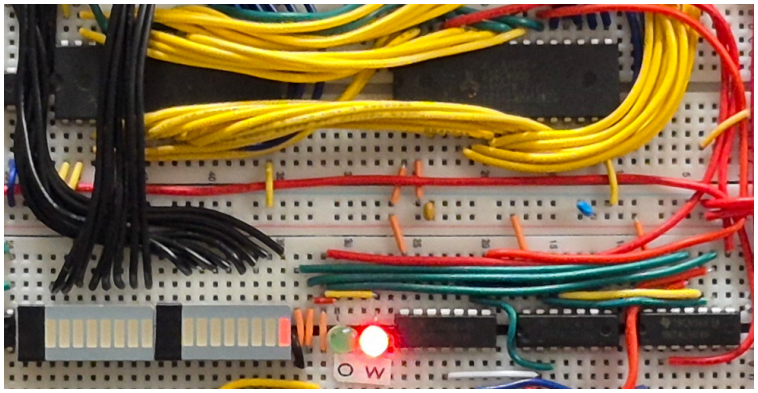


Figure 26: Close up of the RAM Module.

Capacity The RAM module is mainly used to store the 8-bit data of the BF memory-tape. However its secondary purpose is to also store the instruction-pointer values when loops are handled, which are 16-bit in size. Therefore the RAM module contains two 512K x 8-bit SRAM chips (AS6C4008) for a total of 512K 16-bit memory-cells. When the data register's output is enabled, its value will be stored in the lower byte of RAM (chip 0); the high byte (chip 1) is filled with zero's. The second chip is therefore only used to store the high-byte of the IP. Since the stack can only store up to 256 values (in the address range 0x0000-0x00FF), the remainder of the capacity of the RAM chip storing the high byte will never be used. Though this might look wasteful, it was preferred over the alternative of using two sequential bytes on the same chip to store the low and high byte seperately. This would have introduced more complexity in the surrounding logic and would led to multiple cycles for reading from/writing to the stack, negatively impacting performance.

Buffering The AS6C4008 already provides a Chip Enable input which is supposed to be used when the data is connected to a databus. When this input is inactive, its outputs are in a high impedance state to avoid bus contention with other devices. However, in this project we need the data currently pointed to to be visible on an array of LED's, which means that the chip should be enabled basically at all times (except when writing to it). Additional logic is used in conjunction with a pair of 74LS245 tristate buffers to intercept the outputs before making them available on the bus through the buffers. The truthtable for this logic is shown in Table 3. The LED's are not shown in the schematic, but have been connected directly to the datalines of the RAM in this configuration.

Module			RAM			Buffers		
OE	WE	CLK	CE#	OE#	WE#	A→B	EN	
0	0	0	0	0	1	1	0	Show data, but do not send to the bus.
0	0	1	0	0	1	1	0	Show data, but do not send to the bus.
0	1	0	1	1	0	0	1	Prepare to load value from the bus.
0	1	1	0	1	0	0	1	Load value from the bus.
1	0	0	0	0	1	1	1	Send value to the bus.
1	0	1	0	0	1	1	1	Send value to the bus.
1	1	0	1	1	0	0	0	Should not happen (chip disabled).
1	1	1	1	1	0	0	0	Should not happen (chip disabled).

Table 3: Truth table for the logic that drives the RAM in such a way that its output is always enabled, in order for the data to be visualized on LED's without enabling it to the data bus.

5.11 Control Unit

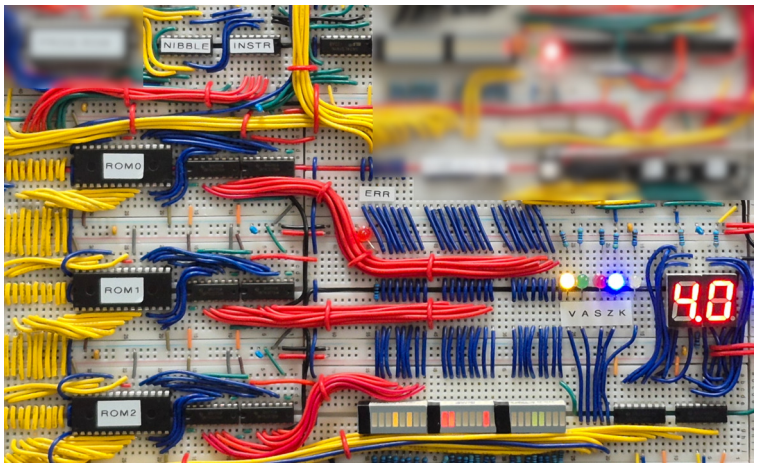


Figure 27: Close up of the Control Unit.

The control unit is responsible for sending the appropriate control-signals to each of the modules. The general idea is that the current instruction pointed to by the IP (4 bits) together with the state flags (another 5 bits: K, A, V, S and Z) and the cycle count (3 bits) combine together to form a 12-bit address into a set of three EEPROM chips (AT28C64B), each of which contains part the signal configuration corresponding to the current state of the system. When clocked by the decoder-clock (D_CLK), the values currently at this address are loaded into six 74LS173 registers (two per EEPROM) and asserted onto their respective modules, which will act upon them on the next pulse of the M_CLK signal.

Address Layout Based on the physical layout of the board, the following configuration was used to construct an address into the EEPROMs.

Address Bits		
0-2	Cycle count	(000_2 - 111_2)
3-7	Instruction	(0000_2 - 1111_2)
8-12	Flags	(00000_2 - 11111_2)
13	Unused	

Generating and Programming Microcode The three EEPROM's have been programmed using a custom built EEPROM programmer based around an Arduino Nano, combined with a python script (`bflash.py`) that is able to send a binary image to the Nano over a serial connection. The images that store the microcode tables have been generated by Mugen (see 6.3), a utility developed to make the microcode programming more maintainable. Mugen generates the images from a specification file. The full Mugen specification listing the Synapse-191 is shown in Appendix B. This is a direct representation of the microcode shown in Table 12 (Appendix A).

Instruction Nibbles The actual BF program is stored in another 8K EEPROM chip (AT28C64) and is addressed by the instruction pointer as mentioned before. Since each BF instruction only needs 4 bits to be encoded (there are less than 16 different opcodes), we can store up to 16K instruction in the chip by packing 2 consecutive instructions together in a single byte (handled by the assembler, `bfasm`). Rather than using bit 0 from the IP directly as address bit 0 on the EEPROM, it is used as the data-select signal to a 74LS157 multiplexer. This multiplexer takes 1 select-bit and two sets of 4 databits. Depending on the value of the select-bit, one of the sets of 4-bit data is sent to its outputs. This allows us to select either the low or high nibble of the data in the EEPROM, effectively doubling the amount of instructions that can be stored and retrieved.

Instruction Register The seleted nibble is loaded into the instruction register (I) at the same time as the V, A, S and Z-flags are loaded into the FB register. For this reason both the FB and I registers can operate on the same control-signal: LD_FBI. The I register is implemented using the 74LS161, which is actually a counting register, because at the time there was no '173 available anymore and these chips are functionally almost identical when counting is disabled on the '161. Initially, the outputs of the multiplexer ('157) were directly connected to the address lines of the microcode EEPROMs but when it turned out that this could cause instabilities in some rare occasions, the I register was added to buffer the instruction for the entire duration of the opcode execution.

Cycle Counter The cycle counter is implemented by a 74LS161 binary counter that simply increments on every M_CLK signal up and sends its outputs (bits 0-2) to address lines 0-2 of the microcode EEPROM chips. It is reset when it receives the CR signal (which becomes active when after an instruction has completed).

5.12 IO Module

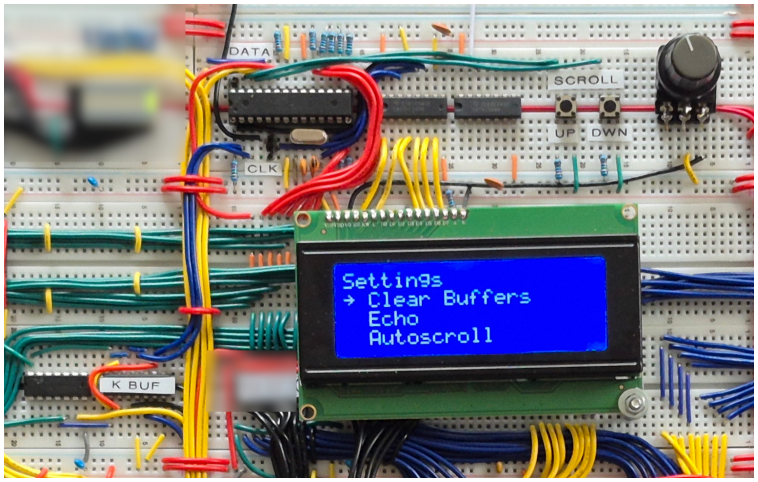


Figure 28: Close up of the IO Module.

5.12.1 Features

The IO system is handled by an ATMEGA328P microprocessor, commonly found in the Arduino Uno. It has five main functions:

1. Supply the program slot index to the system bus at boot.
2. Drive the screen and display contents from the bus when instructed to by the `EN_OUT` signal.
3. Handle keyboard input and provide input data to the bus when instructed to by the `EN_IN` signal.
4. Provide a random number to the bus when both signals are supplied (implementing the *Random Brainf*ck Extension*).
5. Supply a menu system to alter its settings using two buttons.

Buttons and Menu Two buttons are provided to interact with this system. They can be used to scroll the screen or to access and navigate a menu (Figure 29). This menu let's the user do the following:

1. Select the program slot that is loaded by the system on reset.
2. Clear the screen and keyboard buffer.
3. Change the display-mode. By default, incoming data is interpreted as ASCII characters. When it should be displayed as raw numerical values (either in base 10 or 16), this option can be selected from the menu. When in either of these numerical modes, a delimiter character can be selected to separate bytes visually.
4. Set autoscrolling on/off. By default the screen will scroll its contents when they overflow to always keep the most recent data in view. When new data is displayed, the screen is always scrolled to display this data. Setting autoscroll to 'off' will disable these features.
5. Echo on/off. When running an interactive program that requires keyboard input, the user probably wants to see what is being typed. This is the default behavior (echo on). If for some reason the keypresses should not be displayed, this option can be disabled.

6. Set the input-mode. By default, the IO module will wait for the input-buffer to contain a value before putting anything on the bus and notifying the CU throught the K-flag (buffered input-mode). However, an alternative mode (immediate) can be selected, in which case the IO module will put a zero on the bus when the buffer is empty and set the K-flag regardless. This can be helpful if programs require real-time inputs (e.g. for simple games).

7. Set the RNG seed. For programs that use the Random Number Generator as an input device, the seed can be set through this option. Since the same seed will produce the same sequence of numbers, this option can be used to control the randomness of the application. A 'true' random seed can normally be emulated by seeding the generator with the reading of a floating analog input for example, but sadly no free analog inputs were left available on the MCU.

8. Reset to default settings. Whenever settings have been changed, the new settings will be saved to the persistent EEPROM memory of the MCU and loaded back on startup to make the settings persist when the MCU is powered down. This option allows you to revert all changes and load the default settings back in.

Figure 29: Part of the menu that is accessible by pressing both scroll-buttons simultaneously.

5.12.2 System Interactions

K-Flag The K-flag acts as an asynchronous barrier between the IO module and the rest of the system. It can be set by the module and is reset by the main system; this allows for synchronized communication between two asynchronous systems. It is implemented as a dual D-Flip-Flop (74LS74) of which the first flip-flop is set asynchronously (w.r.t. the main system) by the IO module. The output of this flip-flop is then latched synchronously into the second flip-flop on every system clock edge (M_CLK), the output of which it is fed into the CU together with the other flags. The CLR_K signal emitted by the CU will reset both flip-flops, indicating to the module that some transaction has been completed. The flag is used during boot (to communicate the program slot index), during the OUT instruction (to indicate that the output-byte has been received) and during the IN instruction (to indicate that the input-byte has been received).

Bootloading The first transaction between the system and the IO module happens at boot: the IO module is expected to put the index of the program (selected by the user through the menu) onto the databus. However, when the system is reset, it will first initialize its memory. This process involves data transfers over the databus, so the IO module should wait for a signal from the system to indicate that the initialization is completed and the data bus can be used safely. To achieve this, the IO module enables the K-flag and waits for it to be set low by the system. Next, it will write the program index to the bus, again setting K to indicate that this value is ready. The system will then read this value from the bus and acknowlegde the transfer by resetting K once again. Once the IO module sees K going low again, it will disable its outputs and go to its normal operation, listening for input and output commands.

Output The M_CLK signal is connected to an interrupt pin of the MCU. On every interrupt triggered by the clock, when the system is in its IDLE state, it will check the EN_IN and EN_OUT lines to determine if it should initiate a read or write sequence to the databus. When EN_OUT is found to be high, it will copy the byte currently present on the bus into its screen-buffer, set the K-flag and change its state to WAIT_SYS. On every subsequent clock pulse, while in the WAIT_SYS state, it will check if the K-flag has been reset by the control unit. If so, the handshake has been completed and the module returns to its IDLE state (see Figure 30).

Input If the EN_IN signal is found to be asserted, the system has two possible courses of action, depending on the input-mode currently set by the user (through the on-screen settings menu). In (the default) buffered mode, the system will change its state to WAIT_KB and wait for a byte to become available in the keyboard-buffer. As soon as it does, it will provide this value onto the bus and notify the CU by setting the K-flag. It then moves into the WAIT_SYS state to await confirmation by the CU (which in turn resets the K-flag). In immediate mode, all zeros will be written to the bus and the K-flag is set immediately even when the keyboard-buffer is still empty. If, in addition to EN_IN, EN_OUT is asserted as well, a random byte is put onto the bus (see Figure 30).

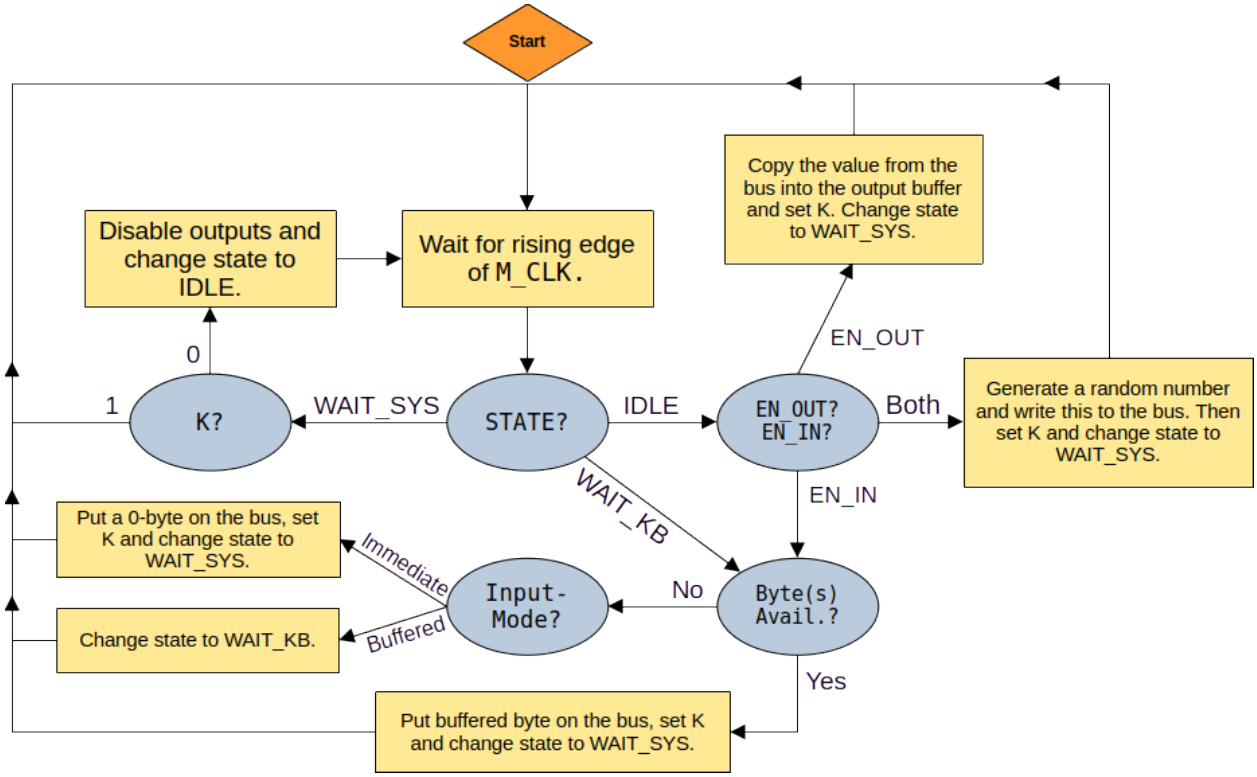


Figure 30: Control flow inside the ISR running on the microcontroller.

5.12.3 Shift Register

A shift register (74HC595) had to be used to decrease the number of pins on the Atmega328p necessary to drive the LCD screen. Not a single IO pin has not been used, so the shift register proved invaluable for this application.

5.12.4 LCD Screen

The software was written in such a way that most common LCD character screens (compatible with Hitachi the HD44780 driver) will be handled appropriately. Both a 16x2 and 20x4 have successfully been installed in

the computer. A modified version of the `LiquidCrystal_74HC595` library was used to implement the LCD driver.

5.12.5 Keyboard

The IO module can only handle input from PS/2 compatible keyboards. A modified version of the `PS2Keyboard` library was used to implement the keyboard driver.

5.13 IO Module Firmware

To ensure minimal latency and efficient CPU–peripheral communication, several low-level optimizations were applied to both the interrupt routine and the main execution loop of the module's firmware.

5.13.1 Ring Buffers

Overview The IO module employs a set of custom ring buffers that provide high-speed, non-blocking data transfers between the asynchronous microcontroller firmware and the synchronous TTL-based CPU. The ring buffer (or circular buffer) is a fixed-size, first-in-first-out (FIFO) data structure that uses two indices, commonly called the `head` and `tail`. The `head` marks the position where the next incoming element will be written, while the `tail` indicates the position of the next element to be read. When either index reaches the end of the buffer, it wraps around to the beginning, forming a logical ring in memory. This structure eliminates the need for memory allocation or data shifting, providing constant-time access for both enqueue and dequeue operations. See Figure 31.

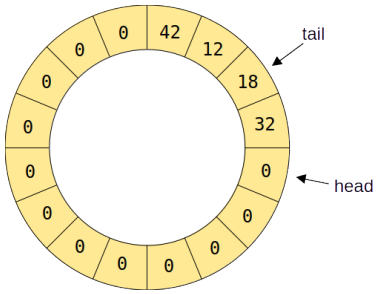


Figure 31: In this ring buffer, the value 18 (pointed to by `tail`) will be returned on the next query. Data entering the ringbuffer will be stored at the cell pointed to by `head`. The value 42 is the first value to be overwritten when the `head` pointer wraps around to the top of the buffer.

In the Synapse-191 I/O module, one ring buffer is dedicated to temporarily storing incoming data (bytes coming from the CPU that need to be written to the LCD screen), the screen-buffer, and another is used for outgoing data (keyboard input that needs to be put on the data bus), the keyboard-buffer. The main loop periodically flushes the screen-buffer to the display and refills the keyboard-buffer with new data when it arrives. Meanwhile, the interrupt service routine (ISR) accesses these buffers asynchronously to satisfy CPU read and write requests. Since all the heavy lifting (interpreting incoming data, managing the LCD screen, decoding keyboard scan-codes) is done in the main-loop, the ISR is kept very compact.

Optimizations The ring buffer is implemented as a templated C++ class, parameterized by buffer size and value type. Several compile-time and runtime techniques were employed to maximize performance and memory efficiency. A combination of compile-time specialization and minimal arithmetic overhead allows the ring buffers to perform low-overhead enqueue and dequeue operations, which is essential for keeping up with the 'high' frequency external system clock of the Synapse-191 CPU.

- **Index Type Specialization:** The index-type (datatype used to store the `head` and `tail` indices, is selected automatically at compile time using template metaprogramming. For small buffers, 8-bit indices are used instead of 16- or 32-bit counters, reducing instruction count, register usage and thus ensuring the fastest possible integer operations on out 8-bit MCU.
- **Integer Wrapping Optimization:** For buffers that are exactly as big as the maximum number representable by some type, that type is used to index the buffers as per the logic above. For example, a 256 byte buffer can be indexed fully using an 8-bit type like `unsigned char`. A value of 255 represented by this type will automatically wrap around when incremented, rendering any additional modular arithmetic to wrap around to the start of the buffer unnecessary.

- **Power-of-Two Optimization:** When the buffer size does not match the condition above, but *is* a power of two, the modulo operation used for index wrapping is replaced with a simple bitwise `AND` operation: `i -> (i + 1) & (N - 1)`. This reduces the naive modulo operation from multiple CPU cycles to a single instruction.

Volatile Access and Memory Barriers The `head`, `tail`, and data array are declared `volatile` to prevent compiler reordering and ensure consistency between the ISR and the main loop. Additionally, an inline assembly memory barrier guarantees that writes to the buffer are completed before the corresponding index is updated, avoiding race conditions.

5.13.2 Direct Port Access

The ISR itself is further optimized using *direct port manipulation*. Instead of relying on Arduino's `digitalRead()`, `digitalWrite()` and `pinMode()` functions, which incur significant overhead, the firmware performs raw register accesses (`PORTX` for writing `PINX` for reading and `DDRX` for changing modes) through custom inline functions. This facilitates faster control over timing-sensitive pins.

5.13.3 Compile-time Menu Structure

Finally, the on-screen configuration interface uses a statically defined menu tree that is fully constructed at compile time using template metaprogramming techniques. This eliminates dynamic memory allocation and object initialization overhead during runtime, minimizing both execution latency and memory footprint. Together, these optimizations produce a maintainable, fast, and memory-efficient IO subsystem capable of operating reliably at high clock speeds of the CPU core.

```
  using Menu = MainMenu <
    SelectSlot<
      SlotSelecter
    >,
    Clear,
    Echo<
      EchoOn,
      EchoOff
    >,
    Autoscroll<
      AutoscrollOn,
      AutoscrollOff
    >,
    DisplayMode<
      TextMode,
      DecMode <
        CommaDelim,
        SemiDelim,
        BarDelim,
        SpaceDelim
      >,
      HexMode<
        CommaDelim,
        SemiDelim,
        BarDelim,
        SpaceDelim
      >
    >,
    InputMode<
      BufferedInput,
      ImmediateInput
    >,
    SetRNGSeed<
      SeedSelecter
    >,
    Defaults,
    Exit
  >;
```

Listing 2: The IO system's menu tree is defined as a type and built at compile-time.

6 Utilities

While designing and implementing the computer, several supporting utilities were developed. The assembler (`bfasm`) is responsible for translating BF programs (text) into machine language (binary), the programmer and its software are used to write data to EEPROM chips and Mugen aids in having a more maintainable microcode definition. Each of these 3 utilities will be described in more detail below.

6.1 Assembler: `bfasm`

Even though the computer is designed to run BF natively, we can't just burn any text-file containing BF commands onto the program-ROM and expect it to execute them. Instead, each of these commands has to be translated into its corresponding binary opcode. Table 4 lists all the available commands and the values that map to these commands. As explained in Section 4, there are a few non-BF that have been added.

Command	Opcode
NOP	0x00
+	0x01
-	0x02
<	0x03
>	0x04
,	0x05
.	0x06
[	0x07
]	0x08
?	0x09
PROG_START	0x0a
PROG_END	0x0b
LOAD_SLOT	0x0c
INIT	0x0d
INIT_FINISH	0x0e
HALT	0x0f

Table 4: Opcode values for each of the available commands.

`bfasm` performs pretty much a one-to-one transformation of the BF commands in the provided textfile into these values. It will add the preamble commands to initialize and bootload the system as was discussed in Section 4.1 and shown in Figure 7).

6.1.1 Features

The `bfasm` utility provides the following features:

Zero Initialization By default, only a single INIT instruction will be emitted in the preamble of the resulting binary file. This will initialize a single 256 byte block of RAM when the system boots. In practice this proved to be enough for most programs but if more is needed, `bfasm` has the option to emit an arbitrary amount of INIT instructions.

Print Filenames Especially when multiple programs are written into a single binary, it can be helpful to see what program is running after loading a slot at runtime. The `-p` flag tells `bfasm` to generate BF code that prints the source filename before it actually runs. This code precedes the HALT instruction that is inserted right before the body of the program; the filename is displayed and the user can resume the clock knowing which program will run.

Interpret '!' as HALT The BF commands do not include a command to stop the program. For debugging purposes, `bfasm` can interpret an exclamation mark in the BF source as a HALT instruction. This can be used to set breakpoints in the BF program.

Interpret '?' as RAND The Random Brainf*ck Extension is supported by `bfasm`; when this option is enabled, each question mark in the BF source code will be assembled into a RAND instruction, which will place a random byte into the current cell (generated by the IO module).

Debug Mode In debug-mode, each BF command is followed by a HALT instruction. This allows the user to step through the program on a command-level rather than on the cycle-level.

Echo With the echo-mode enabled, each input-command will be followed by an output-command in order to see what's being typed on the keyboard (if the BF program does not already do this). This feature was deprecated when the IO-module itself started to provide this facility, but was left in `bfasm` anyway.

Bracket Matching The assembler will not allow programs to contain unmatched brackets. This would for example mess up the logic for the bootloading opcodes and points to invalid BF programs in general.

```
Usage: ./src/bfasm/bfasm [options] <file1, file2, ...>
Options:
-h, --help              Display this text.
-H, --halt-enable       Interpret '!' as HLT in the BF code
-r, --rand-enable       Interpret '?' as RAND in the BF code
-g, --debug             Place a breakpoint (!) after each instruction.
-e, --echo              Follow each input command (,) up by an output command (.) to echo
    keyboard input.
-d, --max-depth [N]     Maximum nesting depth of []-pairs.
-p, --print-filename    Add BF code to print the source filename before the program starts.
-z [N]                  Initialize N chunks of 256 bytes with zero's. Default: N = 1.
-u, --allow-unbalanced-loops
                        By default, the assembler will refuse to produce a program with
    unbalanced
                        loops ([ and ] do not match). Using this option will allow for this
    to occur.
-o [file, stdout]       Specify the output stream/file (default stdout).

Example: ./src/bfasm/bfasm -p -o image.bin program1.bf program2.bf
```

6.2 Programmer: `bflash`

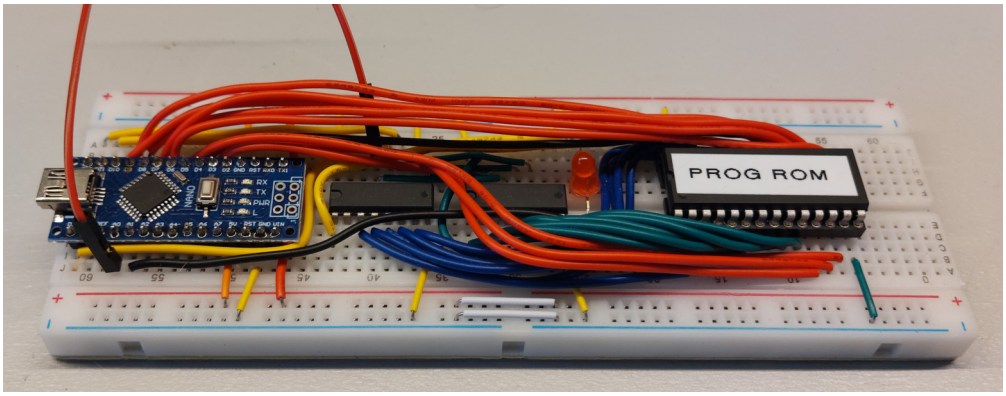


Figure 32: EEPROM chips were programmed using an Arduino Nano on a breadboard.

6.2.1 Overview

Given that there are four EEPROM chips embedded in the computer (one containing the program and three containing the microcode), we had to develop a toolkit for programming these. Specialized programmers can be pretty expensive and relatively hard to acquire, so an Arduino Nano was used to carry out that task. It waits for a serial connection and transfers incoming data byte per byte to the EEPROM chip. This serial connection is established by a Python script that accepts a binary blob and passes this on to the Arduino. The Python utility is called `bflash` (although it's not really BF-specific); its source and the Arduino sketch can be found at `https://github.com/jorenheit/bfcpu/tree/main/src/bflash`. A schematic for the programmer hardware (Figure 32) is shown in Appendix E.

6.2.2 Flashing the AT28C64B

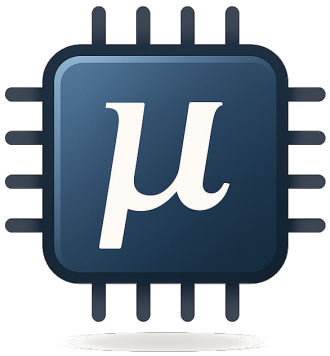
The AT28C64B 8x8K EEPROM chip is used for both microcode as program storage. A value can be written to a specific address by asserting using the following sequence of inputs:

1. Assert the value and address onto the data and address lines of the EEPROM.
2. Set WE (Write Enable) low and OE (Output Enable) high. Both of these pins are active low, so this puts the chip in write-mode.
3. Hold CE (Chip Enable) low for at least 100ns; we chose 1μs because this is the smallest delay that can be performed using standard Arduino library functions.

6.2.3 Shift Register

Because of the large number of connections to the EEPROM chip (13 address lines, 8 data lines and 3 control lines), two shift registers (74HC595) were used to buffer the address and WE/OE control lines.

6.3 Microcode Compiler (Mugen)



6.3.1 Motivation

Initially, the binary images that were burnt onto the microcode EEPROM chips were generated using a simple Octave/Matlab script. This meant that both the microcode and the logic to generate the images had to be expressed in this language. While this certainly worked (albeit a bit slow), we felt the need to develop a more general approach to generating microcode images. To satisfy this need, Mugen was developed. It takes a file in which the microcode can be expressed intuitively and generates the binary images from it. The Mugen project can be found in `https://github.com/jorenheit/mugen`. The listing in Appendix B shows the contents Mugen specification file for this project.

6.3.2 File Structure

A Mugen file consists of four or five different sections in any order:

- `rom`: specifies the ROM configuration of the system;
- `signals`: lists all the signals that make up the control word;
- `opcodes`: lists all the opcodes of the CPU and assigns numerical values to them;
- `address`: specifies how the microcode table address is formed from the opcode, cycle-count and flag-bits;
- `microcode`: specifies the control-sequences for each of the opcodes;
- `macros` (optional): lists common signal combinations that can be referred to by a single name in the microcode section (or within the macro section itself).

6.4 Emulation Framework (Rinku)

Before and during development of the physical system, a C++ framework was developed for emulating computational systems by defining modules and signals that connect them. The Synapse-191 has been emulated cycle-accurate using this framework, which could then be debugged interactively to identify issues with the logic of the real-life computer. Mugen was used to generate C++ source code containing the lookup tables that are normally flashed to the EEPROM chips, such that the same `.mu` files could be used to drive to real system as well as the emulated one. The Rinku project can be found on Github at `https://github.com/jorenheit/rinku/`.

7 Runtime Results

In this section we show snapshots of the output during the execution of each of the BF programs listed in Appendix D. Each of these programs is run multiple times at different clock speeds up to 250 kHz to investigate the stability of the system. All of the statistics have been collected in a modified version of the interpreter shown in Listing 1:

- *Program Size* - the number of BF commands in the program.
- *Memory Cells Required* - the minimum number of memory cells required to execute the program. In our case, this corresponds to the minimal amount of cells to zero-initialize at startup.
- *Runtime Instructions* - the number of BF instructions that are executed when the program is run. Due to the existence of loops, this number differs (for most programs) from the program size.
- *Runtime Clock Cycles* - the number of clock cycles necessary to run the prgram.
- *Cycles per Instruction* - the average number of clock cycles needed to execute a single BF instruction. This value changes from program to program and depends on the structure of the program, the number of skipped loops, commonality of command sequences that leverage the A and V flags, etc. These values have been measured in the interpreter but have been verified on the physical hardware.

7.1 `hello.bf`

Figure 33: *Hello World*

It should be obvious why *Hello World* is the first program to test. The output is simply the string *"Hello World!"* followed by a newline. This program performed as expected on any selected clock frequency within the supported range (and maybe even beyond). This program was taken straight from the Wikipedia page on Brainfuck [8].

Program Size	112
Memory Cells Required	4
Runtime Instructions	390
Runtime Clock Cycles	1273
Cycles per Instruction	3.3

Table 5: Properties of `hello.bf`, see Appendix D.1

7.2 `euler.bf`

Figure 34: *Calculating Euler's Number*

The Euler number (e) is an irrational number, so calculating its digits is an infinite process; the `euler.bf` program (written by the notorious BF developer Daniel Cristofani [11]) never ends. Running this program for a long time exposes a weakness of the system; more often than not will the system get into some erroneous state and go haywire. It is still not fully understood what causes this to happen. Among the most probable causes are bad connections to ground. We observed that some connections to ground have a resistance in the order of 10's of Ohms, where a value close to 0 would be expected. At high frequencies, this might cause issues especially in longer running programs. However, this hypothesis was not thoroughly tested as it proved very difficult todiagnose or get rid of these problems altogether.

Since this programs runs indefinitely, there was no bound to the number of runtime instructions. The number of memory cells required also keeps growing with time; it was found to grow proportional to the square root of the number of BF instructions that had been executed up to that point according to $M = 0.36\sqrt{N}$. A quick back-of-the-envelope calculation shows that for our case, with 64K (2^{16} cells) of RAM available (the stack is negligible in comparison), we'd be able to execute $\left(\frac{2^{16}}{0.36}\right)^2 = 3.3 \times 10^{10}$ BF instructions, which would take 1.1×10^{11} cycles to complete (using the measured value of 3.4 cycles per instruction), or 4.5×10^5 seconds at 250 kHz (over 124 hours). It's probably safe to assume that those limits won't be reached any time soon.

Program Size	1424
Memory Required	$\sim \sqrt{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.4

Table 6: Properties of `euler.bf`, see Appendix D.3

7.3 `phi.bf`

Figure 35: *Calculating ϕ*

Like e, the Golden Ration $\phi = \frac{1+\sqrt{5}}{2}$ is irrational and thus has infinitely many digits. The `phi.bf` program [10] runs indefinitely and has growing memory demands when it runs for a longer amount of time. Again, the number of memory cells requires grows as the square root of the number of BF commands executed according to $M = 71 + 0.033\sqrt{N}$. This number grows even slower than that required to calculate Euler's number, so there won't be any memory issues here neither.

In practice, this was the most difficult program to execute due to stability issues. Sometimes it would run for a long time without any problems; other times it would start going all over the place after only a few digits.

Program Size	1950
Memory Required	$\sim \sqrt{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.6

Table 7: Properties of `phi.bf`, see Appendix D.4

7.4 `factorial.bf`

Figure 36: *Calculating increasingly larger factorials*

Another program that runs indefinitely is `factorial.bf`; it calculates $n!$ for ever increasing n ad infinitum. The program runs more stable than for example `phi.bf`, but ultimately suffers from the same problems that the other long runners do. Unfortunately, this program was found and downloaded long before this report ever needed to mention it; we were unable to trace and credit its source at the time of writing.

The memory requirement grows according to a cube root, so this program can run for an even longer time compared to `phi.bf` and indeed `euler.bf` before memory runs out (provided it runs without issue): $M = 12.5\sqrt[3]{N}$.

Program Size	964
Memory Required	$\sim \sqrt[3]{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.6

Table 8: Properties of `factorial.bf`, see Appendix D.2

7.5 `primes.bf`

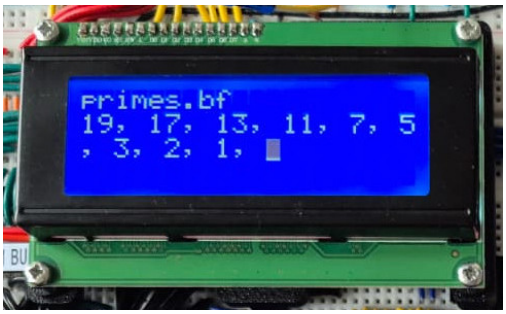


Figure 37: *Calculating primes* < 20.

Found on Reddit by the user Troemax [12], this program checks primality for all numbers ≤ 20 (this value can be adjusted by changing the number of `+` commands at the very start of the program). It runs stable at all tested clock speeds up to 250 kHz. The algorithm reuses memory very efficiently during the course of its execution, requiring only 18 cells.

Program Size	790
Memory Required	18
Runtime Instructions	3.6M
Runtime Clock Cycles	12M
Cycles per Instruction	3.3

Table 9: Properties of `primes.bf`, see Appendix D.5

7.6 `primes2.bf`

Figure 38: *Calculating primes* < 20 *(entered at runtime).*

Like the former prime generating program, this program calculates all primes below 20. The difference between between `primes.bf` and `primes2.bf` is that the latter asks for the upper bound at runtime (expecting keyboard input). The program was posted on the codegolf section of Stackexchange by user Alexandru, challenging other users to implement small interpreters that can successfully run it. For yet unknown reasons, the input is not aquired reliably at clock speeds over 125kHz. We expected the handshake algorithms including the K-flag to mitigate these types of problems and have not been able to identify the flaw that causes the system to misbehave at higher clocks.

Program Size	3880
Memory Required	10
Runtime Instructions	5K
Runtime Clock Cycles	1.8M
Cycles per Instruction	3.5

Table 10: Properties of `primes.bf`, see Appendix D.6
tab:primes2

7.7 `tictactoe.bf`

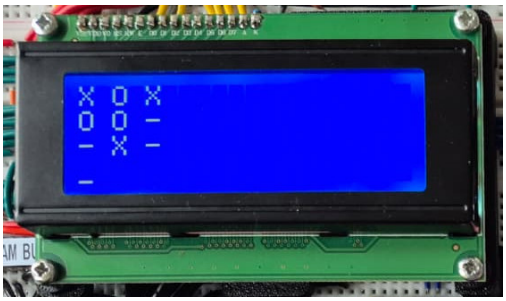


Figure 39: *Playing Tic Tac Toe against Synapse-191.*

Found on Github [14], from a developer by the username Mixtela [14], we found a BF program that actually implements a game of Tic Tac Toe, including a very capable 'AI' opponent. Like `primes2.bf`, the clock is bottlenecked by the input-commands such that it will only run properly at clocks below 125 kHz. The statistics in Table 11 have been gathered by playing a single game (which ended in a draw).

Program Size	6880
Memory Required	72
Runtime Instructions	224K
Runtime Clock Cycles	774K
Cycles per Instruction	3.5

Table 11: Properties of `tictactoe.bf`, see Appendix D.7

8 Conclusion

8.1 Retrospective

Building Synapse-191 has been a tremendously powerful learning experience. Above all, the joy of seeing months of tedious work, planning and debugging culminate into a working system is unparalleled. Even though the system is not fully stable in all programs, it works well enough to demonstrate that it does the job it was designed to do in principle. Features like the IO system, being able to store multiple programs to select from, single-stepping and easy control of the clock make it an enjoyable process to operate the computer. Moreover, with the toolchain developed to a mature state, adding features through modification of the firmware and microcode was relatively straightforward and thoroughly rewarding to do. During development, various ideas and opportunities for potential improvement came to mind, some of which are listed in the sections below.

8.2 Final Thoughts and Potential Improvements

Virtual Registers A big limitation of using the register driver as a centralized system to increment or decrement each register is the fact that only one register can be addressed per clock cycle. Having two addresses on the register driver still unconnected (addresses 110 and 111), their corresponding outputs could be connected to multiple registers that commonly need to increment together. For example, address 110 (bits RS2 and RS1) would then correspond to the virtual register D/IP. The corresponding U and D output signals would then be connected (through OR gates) to the counting inputs of both the D and IP register. Addressing this virtual register would then allow for incrementing or decrementing these registers simultaneously, which saves a clock cycle in the PLUS instruction. The MINUS instruction does not benifit from this trick, since we can't both decrement D and increment IP at the same time. The same could be done for DP and IP, saving a clock cycle in the RIGHT instruction. Even though these are common instructions in many BF programs, it remains questionable if it is worth the effort to implement this for such relatively small gains with respect to the rise in complexity.

Designing a PCB The system's instabilities would likely vanish if it was implemented on a PCB, due to very short connections to large voltage and ground planes and stable traces between IC's. If this direction is ever going to be pursued, most LED's would probably be removed to minimize the PCB's footprint. We would also have to consider how the program ROM can be inserted and removed easily; maybe even using ROM on an expansion card like the NES of the old days. For maximal modularity, the IO module could be implemented as an expansion card; different implementations (perhaps even using graphics or sound instead of ASCII-characters as its output-mode) could easily be swapped in and out.

Going Full 16-bit In Section 5.10 it was mentioned that the vast majority of memory-cells in the second SRAM-chip are left unused. Only the high byte of the IP values is stored in this chip when the IP is pushed to the stack on entering a loop, for a maximum of 256 bytes stored on a 512kb memory chip (effectively 64kb because of the 16-bit address bus); none of the other cells are ever used. Since the data bus is already 16-bits wide, it would have been fairly easy to make the D-register a 16-bit value as well. This would allow for BF programs that assume 16-bit data cells. Even though this was briefly considered in early stages of the project, we chose not to go through with it because a) it is not canonical BF and b) we did not want to make the system more complex than necessary at the time (we would effectively need a 16 input NOR-gate, requiring two more IC's on top of to the two additional register IC's). In hindsight, this would have been a nice (and pretty achievable) feature to have.

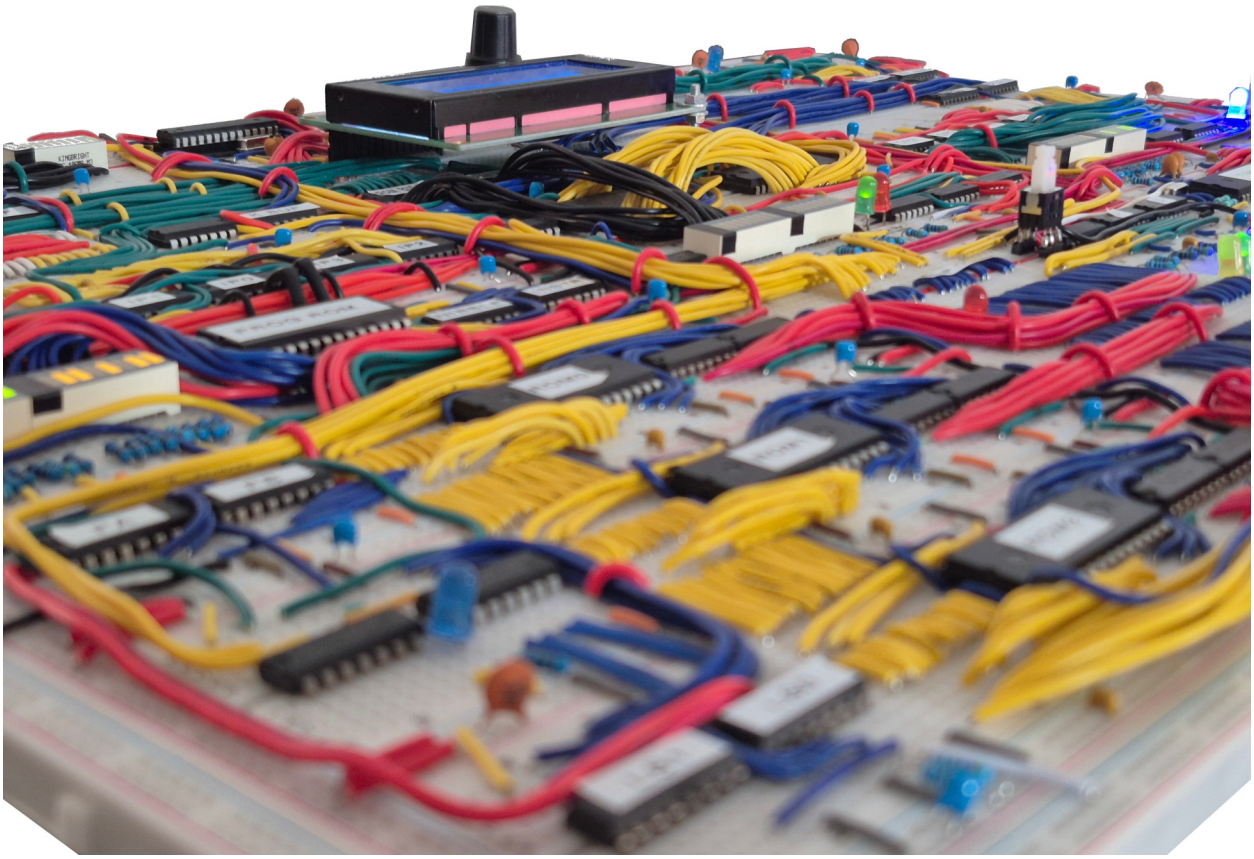
Using the '191 The 74LS191 seems like the perfect candidate to implement the counting registers. It simply has a U/D input to select the counting direction, a count-enable input (CTEN) and is connected directly to the clock. This is exactly how the register-modules behaved in early abstract designs of the system. The '191 was never used for the simple reason that we never knew about its existence and ordered the '193 instead; that was the first chip we encountered that could do the job. In hindsight - and purely by luck - the '193 was indeed the better choice because of its ability to (asynchronously) clear its values; the

'191 has no clear input. This would have made it more difficult to implement clean reset functionality. Also, the number of control-lines to the modules would not have been different (U/D + CTEN instead of U + D), so the need for a register driver would still be there.

Decoupling Capacitors When resorting to the internet (websites, forums, AI chatbots) to gather tips on stability, all of them mention the same potential solution: place decoupling capacitors (100 nF) as close as possible to the VCC and GND pins of every IC. Even though many of these have been placed around the board across the power rails, near the IC's, ideally they would have to be placed right across the IC's power pins (often located diagonally opposite one another). We never observed any improvements of placing such capacitors anywhere so - naively - we never sacrificed looks for potentially more stability (placing the caps across the IC's doesn't look very good in my opinion). If we were to start over again (and we won't), these capacitors would have to be integrated in the design, just to be sure.

Using AI LLM's like ChatGPT have proven a valuable tool while designing and debugging the system. Let's be clear: no LLM's have been used to generate the text of this document or even write a single line of supporting code in the toolchain. It was however a valuable brainstorming partner when trying to find subtle logic bugs in both software and hardware. It almost never got it right completely on the first try, but it almost always inspired new avenues to explore. This more then often led to us identifying the problem eventually. Thinking out loud together with a partner will often lead to new insights and believe it or not, it is hard to find (human) sparring partners on a specialized project like this one.

IO Module: Cheating? Should using an MCU in the implementation of the IO module be considered cheating? We set out to implement a BF computer without using any programmable chips, so clearly we broke our own rule, right? No. The entire BF core was implemented true to our mission goals: all BF commands are handled by TTL, RAM and ROM chips. The IO Module should be considered an abstract external device; its inner workings are irrelevant to the BF core logic in the same way that the BF language does not specify how user input is aquired or how data is displayed when issuing the `,` and `.` commands. Modern monitors and other peripherals often contain embedded driver chips that handle communication protocols or, in the case of monitors, display settings; this is no different.



Appendices

A Microcode Table

Instr	K	V	A	S	Z	Cycle	Control Signals				
Any except "."						0	LD(FB)	LD(I)			
+			0	0		1	INC(D)	SETV(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	LD(D)	OE(RAM)			
			1	0		2	INC(D)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
				1		1	INC(IP)	CLR(CC)			
-			0	0		1	DEC(D)	SETV(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	LD(D)	OE(RAM)			
			1	0		2	DEC(D)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
				1		1	INC(IP)	CLR(CC)			
<			0	0		1	DEC(DP)	SETA(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	EN(D)	WE(RAM)			
			1	0		2	DEC(DP)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
				1		1	INC(IP)	CLR(CC)			
>			0	0		1	INC(DP)	SETA(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	EN(D)	WE(RAM)			
			1	0		2	INC(DP)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
				1		1	INC(IP)	CLR(CC)			
[			0	0	1	1	INC(LS)				
			0	0	1	2	INC(IP)	CLR(CC)			
			0	0	0	1	INC(SP)				
			0	0	0	2	WE(RAM)	EN(SP)	EN(IP)		
			0	0	0	3	INC(IP)	CLR(CC)			
			1	0		1	EN(DP)	LD(D)	OE(RAM)	LD(FA)	CLR(CC)
				1		1	INC(LS)				
				1		2	INC(IP)	CLR(CC)			
]			0	0	1	1	DEC(SP)				
			0	0	1	2	INC(IP)	CLR(CC)			
			0	0	0	1	EN(SP)	OE(RAM)	LD(IP)		
			0	0	0	2	INC(IP)	CLR(CC)			
			1	0		1	EN(DP)	OE(RAM)	LD(D)	LD(FA)	CLR(CC)
				1		1	DEC(LS)				
				1		2	INC(IP)	CLR(CC)			
.			0			0	LD(FB)	LD(I)	EN(D)		
			1			0	LD(FB)	LD(I)	OE(RAM)		
			0	0		1	EN(D)	EN(SCR)			
	0		0	0		2	EN(D)	CLR(CC)			
			1	0		1	EN(D)	EN(SCR)			
	0		1	0		2	OE(RAM)	CLR(CC)			
	1			0		2	LD(D)	SETV(CU)	LD(FA)		
				1		1	INC(IP)	CLR(CC)			
,				0		1	EN(KB)				
	0			0		2	CLR(CC)				
	1			0		2	LD(D)	SETV(CU)	LD(FA)		
	1			0		3	CLR(K)	INC(IP)	CLR(CC)		
				1		1	INC(IP)	CLR(CC)			
RAND				0		1	EN(KB)	EN(SCR)			
	0			0		2	CLR(CC)				
	1			0		2	LD(D)	SETV(CU)	LD(FA)		
	1			0		3	CLR(K)	INC(IP)	CLR(CC)		
				1		1	INC(IP)	CLR(CC)			
NOP						1	INC(IP)	CLR(CC)			
HLT				0		1	HLT(CLK)				

		0		2	INC(IP)	CLR(CC)		
		1		1	INC(IP)	CLR(CC)		
INIT	0			1	CLR(CC)			
	1		1	1	EN(D)	WE(RAM)	INC(LS)	
	1		1	2	LD(FB)		INC(DP)	
	1	0	1	3	INC(IP)	CLR(CC)		
	1	1	1	3	CLR(CC)			
INIT_FINISH	1			1	CLR(DP)	CLR(K)	INC(IP)	CLR(CC)
LOAD_SLOT	0			1	CLR(CC)			
	1			1	LD(D)	INC(LS)		
	1		1	2	CLR(K)	INC(IP)	CLR(CC)	
PROG_START		1	0	1	DEC(D)			
		1	1	1	DEC(LS)			
		1		2	INC(IP)	CLR(CC)		
PROG_END		0		1	END(CU)	HLT(CLK)		
		1		1	INC(IP)	CLR(CC)		

Table 12: Control signals for each of the BF instructions.

B Mugen Specification

```
# This file can be compiled with Mugen, see https://github.com/jorenheit/mugen.

[rom] { 8192 x 8 x 3 }

[address] {
  cycle:   3
  opcode:  4
  flags:   K, V, A, S, Z
}

[signals] {
  HLT
  RS0
  RS1
  RS2
  INC
  DEC
  CLR_DP
  EN_SP

  OE_RAM
  WE_RAM
  EN_IN
  EN_OUT
  EN_V
  EN_A
  LD_FBI
  LD_FA

  EN_IP
  LD_IP
  EN_D
  LD_D
  CR
  CLR_K
  END
  ERR
}

[opcodes] {
  NOP           = 0x00
  PLUS          = 0x01
  MINUS         = 0x02
  LEFT          = 0x03
  RIGHT         = 0x04
  IN            = 0x05
  OUT           = 0x06
  LOOP_START    = 0x07
  LOOP_END      = 0x08
  RAND          = 0x09
  PROG_START    = 0x0a
  PROG_END      = 0x0b
  LOAD_SLOT     = 0x0c
  INIT          = 0x0d
  INIT_FINISH   = 0x0e
  HALT          = 0x0f
}


[macros] {
  # Register Select
  R_D  = RS0
  R_DP = RS1
  R_SP = RS0, RS1
  R_IP = RS2
```

```
  R_LS = RS0, RS2

  # Load/store to RAM
  LOAD_D   = LD_D, OE_RAM
  STORE_D  = EN_D, WE_RAM
  LOAD_IP  = LD_IP, EN_SP, OE_RAM
  STORE_IP = EN_IP, EN_SP, WE_RAM

  # Set A and V flags
  SET_V  = EN_V, LD_FA
  SET_A  = EN_A, LD_FA
  CLR_VA = LD_FA

  # Modify Data
  INC_D = INC, R_D, SET_V
  DEC_D = DEC, R_D, SET_V

  # Modify Data Pointer
  INC_DP = INC, R_DP, SET_A
  DEC_DP = DEC, R_DP, SET_A

  # Loops
  INC_SP = INC, R_SP
  DEC_SP = DEC, R_SP

  INC_LS = INC, R_LS
  DEC_LS = DEC, R_LS

  # Move to next instruction
  NEXT = INC, R_IP, CR
}

[microcode] {

  # Most opcodes do the same thing in cycle 0: load the new instruction into I and latch flags into FB
  NOP:0:()                      -> LD_FBI
  PLUS:0:()                     -> LD_FBI
  MINUS:0:()                    -> LD_FBI
  LEFT:0:()                     -> LD_FBI
  RIGHT:0:()                    -> LD_FBI
  IN:0:()                       -> LD_FBI
  LOOP_START:0:()               -> LD_FBI
  LOOP_END:0:()                 -> LD_FBI
  RAND:0:()                     -> LD_FBI
  INIT:0:()                     -> LD_FBI
  INIT_FINISH:0:()              -> LD_FBI
  LOAD_SLOT:0:()                -> LD_FBI
  PROG_START:0:()               -> LD_FBI
  PROG_END:0:()                 -> LD_FBI
  HALT:0:()                     -> LD_FBI

  # When OUT is spinning, EN_D must be kept high in cycle 0
  OUT:0:(A=0)                   -> LD_FBI, EN_D
  # OE_RAM in this case, for the same reason
  OUT:0:(A=1)                   -> LD_FBI, OE_RAM

  PLUS:1:(A=0,S=0)              -> INC_D
  PLUS:2:(A=0,S=0)              -> NEXT
  PLUS:1:(A=1,S=0)              -> LOAD_D
  PLUS:2:(A=1,S=0)              -> INC_D
  PLUS:3:(A=1,S=0)              -> NEXT
  PLUS:1:(S=1)                  -> NEXT

  MINUS:1:(A=0,S=0)             -> DEC_D
  MINUS:2:(A=0,S=0)             -> NEXT
  MINUS:1:(A=1,S=0)             -> LOAD_D
  MINUS:2:(A=1,S=0)             -> DEC_D
  MINUS:3:(A=1,S=0)             -> NEXT
```

```
MINUS:1:(S=1)                -> NEXT

LEFT:1:(V=0,S=0)             -> DEC_DP
LEFT:2:(V=0,S=0)             -> NEXT
LEFT:1:(V=1,S=0)             -> STORE_D
LEFT:2:(V=1,S=0)             -> DEC_DP
LEFT:3:(V=1,S=0)             -> NEXT
LEFT:1:(S=1)                 -> NEXT

RIGHT:1:(V=0,S=0)            -> INC_DP
RIGHT:2:(V=0,S=0)            -> NEXT
RIGHT:1:(V=1,S=0)            -> STORE_D
RIGHT:2:(V=1,S=0)            -> INC_DP
RIGHT:3:(V=1,S=0)            -> NEXT
RIGHT:1:(S=1)                -> NEXT

LOOP_START:1:(A=0,Z=1,S=0)   -> INC_LS
LOOP_START:2:(A=0,Z=1,S=0)   -> NEXT
LOOP_START:1:(A=0,Z=0,S=0)   -> INC_SP
LOOP_START:2:(A=0,Z=0,S=0)   -> STORE_IP
LOOP_START:3:(A=0,Z=0,S=0)   -> NEXT
LOOP_START:1:(A=1,S=0)       -> LOAD_D, CLR_VA, CR
LOOP_START:1:(S=1)           -> INC_LS
LOOP_START:2:(S=1)           -> NEXT

LOOP_END:1:(A=0,Z=1,S=0)     -> DEC_SP
LOOP_END:2:(A=0,Z=1,S=0)     -> NEXT
LOOP_END:1:(A=0,Z=0,S=0)     -> LOAD_IP
LOOP_END:2:(A=0,Z=0,S=0)     -> NEXT
LOOP_END:1:(A=1,S=0)         -> LOAD_D, CLR_VA, CR
LOOP_END:1:(S=1)             -> DEC_LS
LOOP_END:2:(S=1)             -> NEXT

OUT:1:(S=1)                  -> NEXT
OUT:1:(A=0,S=0)              -> EN_OUT, EN_D
OUT:1:(A=1,S=0)              -> EN_OUT, OE_RAM
OUT:2:(K=0,A=0,S=0)          -> EN_D, CR
OUT:2:(K=0,A=1,S=0)          -> OE_RAM, CR
OUT:2:(K=1,S=0)              -> CLR_K, NEXT

IN:1:(S=1)                   -> NEXT
IN:1:(S=0)                   -> EN_IN
IN:2:(K=0,S=0)               -> CR
IN:2:(K=1,S=0)               -> LD_D, SET_V
IN:3:(K=1,S=0)               -> CLR_K, NEXT

RAND:1:(S=1)                 -> NEXT
RAND:1:(K=0,S=0)             -> EN_IN, EN_OUT
RAND:2:(K=0,S=0)             -> CR
RAND:1:(K=1,S=0)             -> LD_D, SET_V
RAND:2:(K=1,S=0)             -> CLR_K, NEXT


# HOUSEKEEPING
NOP:1:()                     -> NEXT
HALT:1:(S=0)                 -> HLT
HALT:2:(S=0)                 -> NEXT
HALT:1:(S=1)                 -> NEXT

INIT:1:(K=0)                 -> CR # Wait for IO module to set K (bus is safe to use)
INIT:1:(K=1, Z=1)            -> STORE_D, INC, R_LS
INIT:2:(K=1, Z=1)            -> LD_FBI, INC, R_DP
INIT:3:(K=1, Z=1, S=1)       -> CR
INIT:3:(K=1, Z=1, S=0)       -> NEXT

INIT_FINISH:1:(K=1)          -> CLR_DP, CLR_K, NEXT

LOAD_SLOT:1:(K=0)            -> CR
```

```
  LOAD_SLOT:1:(K=1)             -> LD_D, INC, R_LS
  LOAD_SLOT:2:(K=1)             -> CLR_K, NEXT  # write slot to D and go into skip mode

  PROG_START:1:(Z=0, S=1)       -> DEC, R_D
  PROG_START:1:(Z=1, S=1)       -> DEC, R_LS
  PROG_START:2:(S=1)            -> NEXT

  PROG_END:1:(S=0)              -> END, HLT
  PROG_END:1:(S=1)              -> NEXT

  catch                         -> ERR, HLT
}
```

C Bill of Materials

The following section lists all components required to build the Synapse-191 computer, including integrated circuits, passive components, and other hardware parts. The system was constructed entirely on solderless breadboards using standard through-hole TTL logic and readily available electronic parts. Component quantities reflect the final, fully assembled configuration of the system, including the control unit, data path, clock circuitry, and I/O modules.

Note that in several cases, alternative or suboptimal parts were used based on component availability at the time of construction. These lists are therefore intended as a reference for replication and documentation purposes, not as precise shopping lists. Anyone attempting to build a similar system should expect to keep a stock of spare parts and make appropriate substitutions where necessary.

C.1 Integrated Circuits

Part	Quantity	Description
74LS00	3	4x NAND
74LS02	1	4x NOR
74LS04	6	6x NOT
74LS08	2	4x AND
74LS14	1	6x Schmitt-Trigger
74LS32	2	4x OR
74LS48	2	7-Segment Driver
74LS74	1	2x D-Flip-Flop
74LS76	1	2x JK-Flip-Flop
74LS86	1	4x XOR
74LS123	2	2x Monostable Multivibrator
74LS138	2	3-to-8 Decoder
74LS157	1	4-bit Data Selector
74LS161	2	8-Bit Counter
74LS173	9	8-Bit Register
74LS193	14	8-Bit U/D Counter
74LS245	10	8-Bit Bus Transceiver
74HC595	1	8-Bit Shift Register
MC14078B	2	8-Input NOR
NE555	4	555-Timer
AT28C64B	4	64K EEPROM
AS6C4008	2	512K SRAM
ICM7226B	1	Frequency Timer

Table 13: IC's used in the project.

C.2 Passive Components

Value	Quantity
220 Ω	1
330 Ω	8
470 Ω	80
1 $k\Omega$	5
2 $k\Omega$	1
4.7 $k\Omega$	10
5 $k\Omega$	3
10 $k\Omega$	5
100 $k\Omega$	2
220 $k\Omega$	1
470 $k\Omega$	1
10 $M\Omega$	2

(a) Resistors

Value	Quantity
470 μF	1
10 μF	1
4.7 μF	1
1 μF	3
100 nF (104)	50
10 nF (103)	5
2.2 nF (222)	1
1 nF (102)	4
470 pF (471)	1
220 pF (221)	1
100 pF (101)	1
47 pF	2
22 pF	2

(b) Capacitors

Value	Quantity
10 MHz crystal	1
16 MHz crystal	1
10 kΩ potentiometer	2
Toggle switch	2
DIP switch (x8)	1
Tactile button	5
20-22AWG wire	?

(c) Other passive components

Table 14: Resistors, capacitors and other passive components used in the project.

C.3 Other Parts

Part	Quantity	Description
Breadboard	∼25	830 hole
Single 7-Segment Display	2	
Quad 7-Segment Display	2	
Single LED	15	Various colors
Bar LED strip	11	10 LEDs per strip
HD44780 Compatible LCD	1	4x20 Characters

Table 15: Other parts used in the project.

D BF Test Suite

D.1 Hello World

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.--
----.--------.>+.>.
```

Listing 3: hello.bf, [8]

D.2 Factorial

```
++++++++++>>>+>>>>+>+<[[+++++[>++++++++<-]>.<++++++[>--------<-]+<<]<<[<<]<.>>>>+<[->[<+>-[<
+>-[<+>-[<+>-[<+>-[<+>-[<+>-[<+>-[<[-]>-+>[<->-]<[->>>[>>]<<[->[>>+<<-]>+<<<<]<]>[-]+>+
<<]]]]]]]]]]<[>+<-]+>>]<<[<<]>>[->>[>>]>>[-<<[<<]<<[<<]>[>[>>]>>[>>]>>[>>]>+>+<<<<[<<]<<[<<]
<<[<<]>-]>[>>]>>[>>]>>[>>]>[<<<[<<]<<[<<]<<[<<]>+>[>>]>>[>>]>>[>>]>-]<<<[<<]>[>[>>]>+>>+<<<<
<[<<]>-]>[>>]>[<<<[<<]>+>[>>]>-]>>[<[<<+>+>-]<[>>>+<<<-]<[>>+<<-]>>>-]<[-]>>+[>[>>>>]>[>>>>]
>[-]+>+<[<<<<]>-]>[>>>>]>[>>>>]>->-[<<<+>>+>-]<[>+<-]>[[<<+>+>-]<[<->-[<->-[<->-[<->-[<->-[<
->-[<->-[<->-[<->-[<-<---------->>[-]>>>>[-]+>+<<<<<]]]]]]]]]]<[>>+<<-]>>]<+<+<[>>>+<<<-]<<[
<<<<]<<<<<[<<]+>>]>>>>>[>>>>]+>[>>>>]<<<<[-<<<<]>>>>>[<<<<]<<<<<[<<]<<[<<]+>>]>>[>>]>>>>>[->
>>>]<<[<<<<]>>>>[>>>>]<<<<<<<<[>>>>[<<+>>->[<<+>>-]>]<<<<[<<]<<]<<<<<[->[-]>>>>>>>>[<<+>>->[
<<+>>-]>]<<<<[<<]<<<<<<<]>>>>>>>>>[<<<<<<<+>>>>>>>->[<<<<<<<+>>>>>>>-]>]<<<<<<<<<]
```

Listing 4: factorial.bf, [9]

D.3 Euler's Number

```
[.--..]>>>>++>+>++>+>>++<+[[>[>>[>>>>]<<<<[[>>>>+<<<<-]<<<<]>>>>>>]+<]>->>--[+[+++<<<<--]++>
>>>--]+[>>>>]<<<<[<<+<+<]<<[>>>>>>[[<<<<+>>>>-]>>>>]<<<<<<<<[<<<<]>>-[<<+>>-]+<<[->>>>[-[+>>
>>-]-<<-[>>>>-]++>>+[-<<<<+]+>>>>]<<<<[<<<<]]>[-[<+>-]]+<[->>>>[-[+>>>>-]-<<<-[>>>>-]++>>>+[
-<<<<+]+>>>>]<<<<[<<<<]]<<]>>>+[>>>>]-[+<<<<--]++[<<<<]>>>+[>-[>>[--[++>>+>>--]-<[-[-[+++<<<
<-]+>>>>-]]++>+[-<<<<+]++>>+>>]<<[>[<-<<<]+<]>->>>]+>[>>>>]-[+<<<<--]++<[[>>>>]<<<<[-[+>[<->
-]++<[[>-<-]++[<<<<]+>>+>>-]++<<<<-]>-[+[<+[<<<<]>]<+>]+<[->->>>[-]]+<<<<]]>[<<<<]>[-[-[++++
+[>++++++++<-]>-.>>>-[<<<----.<]<[<<]>>[-]>->>+[[>>>>]+[-[->>>>+>>>>>>>>-[-[+++<<<<[-]]+>>>>
-]++[<<<<]]+<<<<]>>>]+<+<<]>[-[->[--[++>>>>--]->[-[-[+++<<<<-]+>>>>-]]++<+[-<<<<+]++>>>>]<<<
<[>[<<<<]+<]>->>]<]>>>>[--[++>>>>--]-<--[+++>>>>--]+>+[-<<<<+]++>>>>]<<<<<[<<<<]<]>[>+<<++<]
<]>[+>[--[++>>>>--]->--[+++>>>>--]+<+[-<<<<+]++>>>>]<<<[<<<<]]>>]>],.,.,.
```

Listing 5: euler.bf, [11]

D.4 Golden Ratio

```
++++++[>++++++++<-]>+.---.>>>+>>+>++>+>+>+>>+>+++>+>>+>+>+>>+>>>>+>>>>+>>+>++>>>>+>+<[[>>>]<<<[
>>>>>>>>>+<<<<<<<<<[>>>>>>>>>+<<<<<<<<<-]<-<<<]>>>>>>+>>>+[<<<]<<<<<[>>>+<<[>>>+<<<<-]<-<<<]>>
>+<<<<<<<<<[<<]>>>>[->[<<<+>>>>[>>]>+<<<[<<]>-]<<<-[>>>+<<<<-]+>>>>[>>]+>[-[-[-<-><<<<<<[[>>>]>
>]<<<<<[>+<-<<<]<<[<<<]>>>[->>>[>>>]>>[>>>]+[<<<]<<[<<<]+>>>]>>>[<[-]+>->>>]<[>>>]<<<[>+++++
++++<<<<]<<[<<<]>>>[->[<<+>+>-]<[>+<-]>[>>[>>>]>>>>[>>>]<-<<[<<<]<<<<[<<<]>-]>>[>>>]>>>>[>>>
]+[<<<]<<<<[<<<]<[>>+<<-]>+>>>]>>>>[->>>]<<[>>>]<<<[>[-[-[-[-[-[-[-[-[-<->[<+>[-]>+<]]]]]]]]
]]<[[>>[<<<+>>>-]<[-]<-<<<]>>>]<<<]>+>[-<-<<<[[<<<]<<]<<<[<<]+>>->>[>>]>>>>>>>-<[[>>>]>>]<]<
[-<<<[[<<<]<<]<<<[<<]+[<<]>>->>[>>]>>>>>>[[>>>]>>]<<]<<<[[<<<]<<]]<[->>>>>>>+[<<+>+>-]<-[>+<
-]+<[-[-[-[-[-[-[-[-[-[-[->>-<<<<<<<<<<[<<]+>>>>>>>>>->>[>>]>>>>>]]]]]]]]]<<<<<]>]<[->>>>>>>[<<
<+>+>>-]<<[>>+<<-]+<[>-<[>>[>>>]>>[>>>]>>[>>>]<<<[-<<<]<<[<<<]<<[<<<]>>>[->[<<+>+>-]<[>+<-]<
[>>>>[>>>]>>[>>>]>>[>>>]>++<<<<[<<<]<<[<<<]<<[<<<]<-]>>>>[>>>]>>[>>>]>>[>>>]+[<<<]<<[<<<]<<[
<<<]+>>>]<<<[<<<]>-]>>[>>>]>>[>>>]>>[->[<+>>+<-]<[[<+>>+<-]<[>+<-]>->-[<->-[<->-[<->-[<->-[<
->-[<->-[<->-[<->-[<->-[<->]>---------->[-]+>+<<<]]]]]]]]]<]+>>[<+>-]>]<<<[[<<<]<<]>>>>>>[<<+>>
>+<-]>[<+>-]<<<[>[[>>>]>>]<+<<<<[[<<<]<<]>>>>-]>[[>>>]>>]>+<<<-[>+<-]>[>+++<-[>+++++<-[>--->>
+>+<<<<-[>->>>+<<<<-[>+>>>+<<<<-[>+++>>>+<<<<-[>-----<<<++<<<<-[>--->>>++<<<<-]]]]]]]]+[[<<<
]<<]>>>]>[->[[>>>]>>]+>>>>>+[[<<<]<<]>>>>]<<<<<]>]<[->>>>>>[[>>>]>>]<<<<<[<<<]>>>[-<<<[<<<]<
<[<<<]>>[>>>]+>[>>>]>>[>>>]>[<<<<[<<<]<<[<<<]>>[>>>]<<<+>[>>>]>>[>>>]>-]<+>>>]<<<[-<<<]<<[<<
<]>>[->>>]<<[<<<]>>>[+++++++++<[>-<-]>[>+<-]+>>>]<<<[<<<]>>>>+<[->[<<+>+>-]<[-[-[-[-[-[-[-[
-[<----------->>>>[-]+>+<<<<[-]]]]]]]]]]<[>>+<<-]>+>>>]<<[-<]<<[<<<]>>>->[<<+>>-]>>[>>>]<<<[
>>+<[>-<<[<<<]>]>[-<<-<]<<]+<[>>+<<-]>[[<<<]<<]<]<<[<<]+>>]>+>[>+++++[>++>++<<-]<-]>>>>>[<<<
+>+>+>-]<-[>+<-]+<<[<[<+>-]<<[>>+<<-]>[<+>-]>>-]<<<[-]>>[<+>>>-<<-]++++++[>++++++++<-]>>[<+>
--]<.[-]>>[>>>]>>]
```

Listing 6: phi.bf, [10]

D.5 Primes

```
++++++++++++++++++++[[>+>+<<-]>>[<<+>>-]<--[+>>[-]<<<[>>+>+<<<-]>>>[<<<+>>>-]<>+[<[>>>>>+>+<
<<<<<-]>>>>>>[<<<<<<+>>>>>>-]<<<<<<[>[-]<<[>>+>+<<<-]>>>[<<<+>>>-]<<[>>+>+<<<-]>>>[<<<+>>>-]
<[<->-]<>>>+<<<[>>>-<<<[-]]<-]>>>>>[<<<<<+>>>>>-]<[[-]<<<<<[>->+<<-]>>[<<+>>-]+>]<]<<<>>>>>>
>+<<<<<<<[>>>>>>>-<<<<<<<[-]]<--]>>>>>>>>>>+<[>-<[-]]>[-<<<<<<<<<<[>>>>>>>>>>+>+<<<<<<<<<<-]
>>>>>>>>>>[<<<<<<<<<<<+>>>>>>>>>>>-]<[>>+>+<<<-]>>>[<<<+>>>-]<<+>[<->[++++++++++<[->-[>+>>
]>[+[-<+>]>+>>]<<<<<]>[-]++++++++[<++++++>-]>[<<+>>-]>[<<+>>-]<<]>]<[->>++++++++[<++++++>-]]
<[.[-]<]<[-]>[-]<>++++[<++++++++++>-]<.----------.[-]]<<<<<<<<<<-]
```

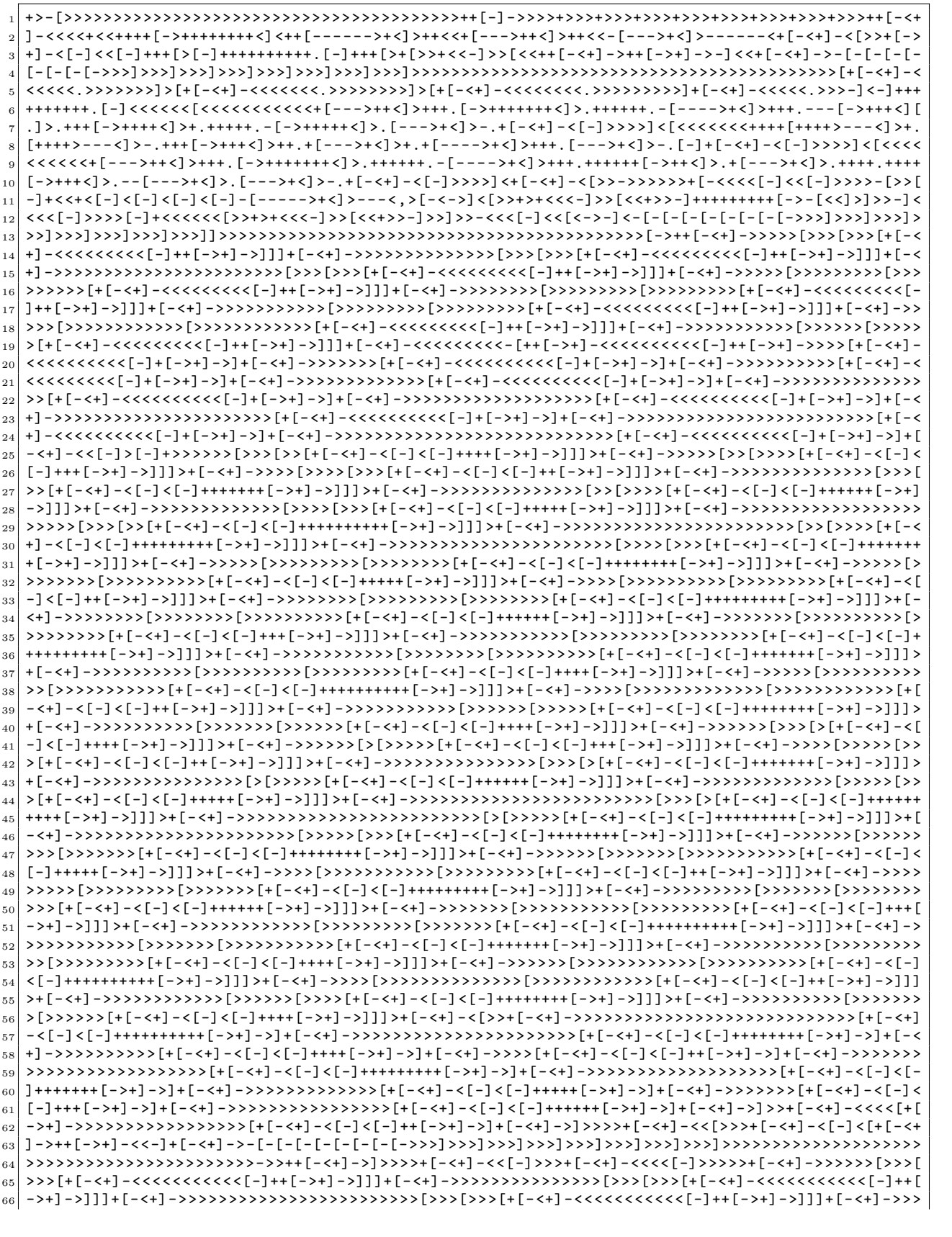
Listing 7: primes.bf, [12]

D.6 Primes 2

```
>++++++++[<++++++++>-]<+++++++++++++++.[-]>++++++++++[<++++++++++>-]<+++++++++++++.[-]>+++
+++++++[<++++++++++>-]<+++++.[-]>++++++++++[<++++++++++>-]<+++++++++.[-]>++++++++++[<+++++++
+++>-]<+.[-]>++++++++++[<++++++++++>-]<+++++++++++++++.[-]>+++++[<+++++>-]<+++++++.[-]>+++++
+++++[<++++++++++>-]<+++++++++++++++++.[-]>++++++++++[<++++++++++>-]<+++++++++++++.[-]>+++++[
<+++++>-]<+++++++.[-]>++++++++++[<++++++++++>-]<+++++++++++++++++.[-]>++++++++++[<++++++++++>
-]<++++++++++.[-]>+++++++[<+++++++>-]<+++++++++.[-]>+++++[<+++++>-]<+++++++.[-]+[->,.------
-----[<+>----------------------------------->[>+>+<<-]>>[<<+>>-]<>>>+++++++++[<<<[>+>+<<-]
>>[<<+>>-]<[<<+>>-]>>-]<<<[-]<<[>>+<-]]<]>>[<<+>>-]<<>+<-[>+[>+>+<<-]>>[<<+>>-]<>+<-->>>>>>>>
+<<<<<<<<<[>+<-<[>>>+>+<<<<-]>>>>[<<<<+>>>>-]<<<<[>>>+>+<<<-]>>>[<<<+>>>-]<<<<<>>>[>+>+<<-]>>[<
<+>>-]<<<[>>>>>+<<<[>+>+<<-]>>[<<+>>-]<[>>[-]<<-]>>[<<<<[>+>+<<-]>>[<<+>>-]<>>>-]<<<-<<-]+>>
[<<[-]>>-]<<>[-]<[>>>>>>[-]<<<<<<-]<<>>[-]>[-]<<<]>>>>>>>>[-<<<<<<<<[-]<<[>>+>+<<<-]>>>[<<<+>
>>-]<<<>>[>+<-]>[[>+>+<<-]>>[<<+>>-]<>+>>>>>>>>>[>>>+<<[>+>[-]<<-]>[<+>-]>[<<++++++++++>>>-]<
<-<-]+++++++++>[<->-]<[>+<-]<[>+<-]<[>+<-]>>>[<<<+>>>-]<>+++++++++<[>>>+<<[>+>[-]<<-]>[<+>-]
>[<<++++++++++>>>+<-]<<-<-]>>>>[<<<<+>>>>-]<<<<<[-]<<+>]<[[>+<-]+++++++[<++++++>-]<->>.[-]>
>[<<+>>-]<<-]>++++[<++++++++>-]<.[-]>>>>>>>]<<<<<<<<<>[-]<[-]<<-]++++++++++.[-]
```

Listing 8: primes2.bf, [13]

D.7 Tic Tac Toe



```
>>>[>>>>>>>>>>[>>>>>>>>>>[+[-<+]-<<<<<<<<<<<[-]++[->+]->]]]+[-<+]->>>>>>>>>>[>>>>>>>>>>[>>>>>>>>
>[+[-<+]-<<<<<<<<<<<[-]++[->+]->]]]+[-<+]->>>>>>>>>>>>>[>>>>>>>>>>[>>>>>>>>>>[+[-<+]-<<<<<<<<<<
<[-]++[->+]->]]]+[-<+]->>>>>>>[>>>>>>>>>>>>>[>>>>>>>>>>>>>[+[-<+]-<<<<<<<<<<<[-]++[->+]->]]]+[-
<+]->>>>>>>>>>>>>[>>>>>>>[>>>>>>>[+[-<+]-<<<<<<<<<<<[-]++[->+]->]]]+[-<+]-<[-]]++[->+]->]+[-<+]
-<+[-<+]-<]>>+[->+]->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+[-[-]<+]-<+[-[-]<+]-<+>]
```

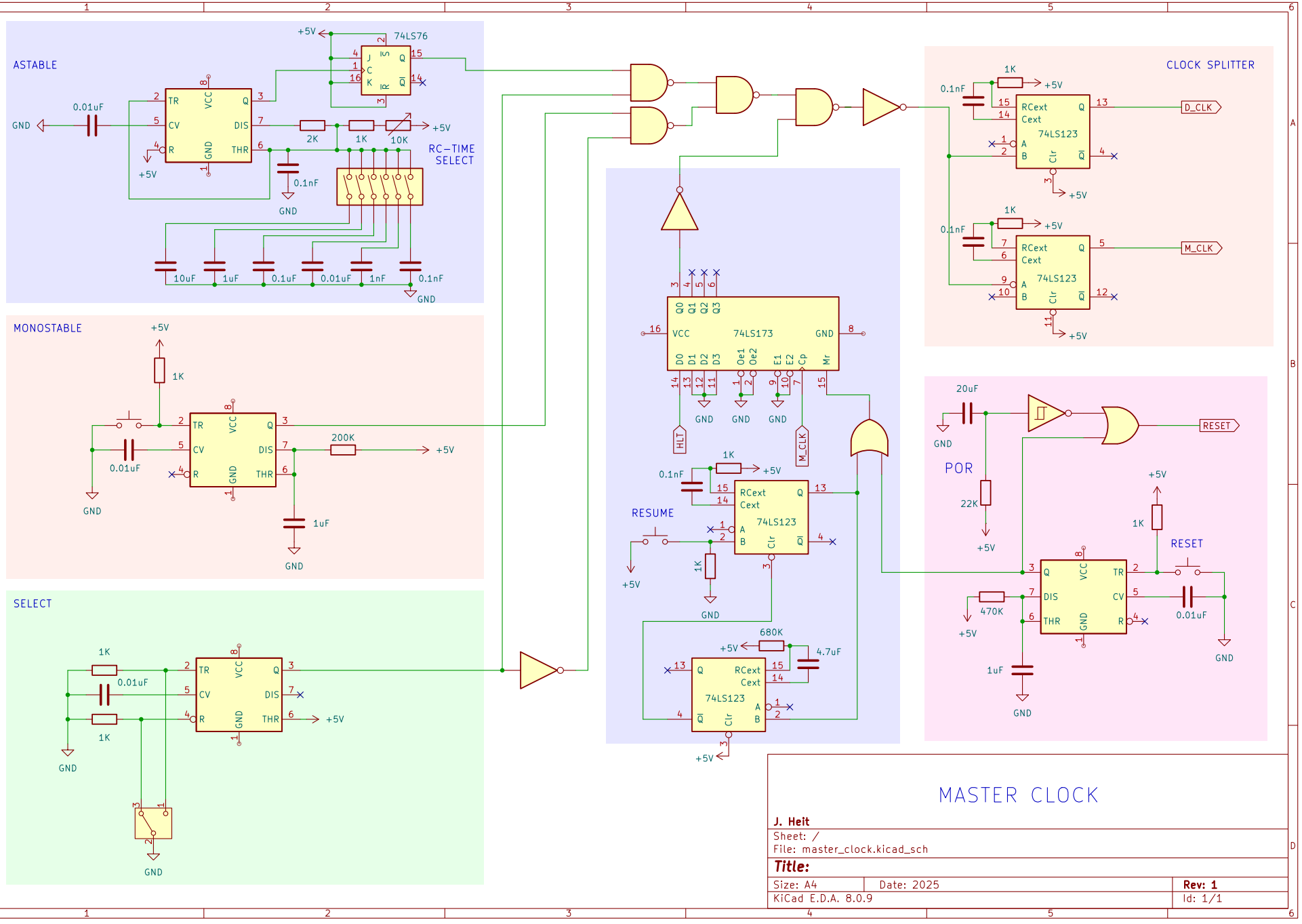
Listing 9: tictactoe.bf, [14]

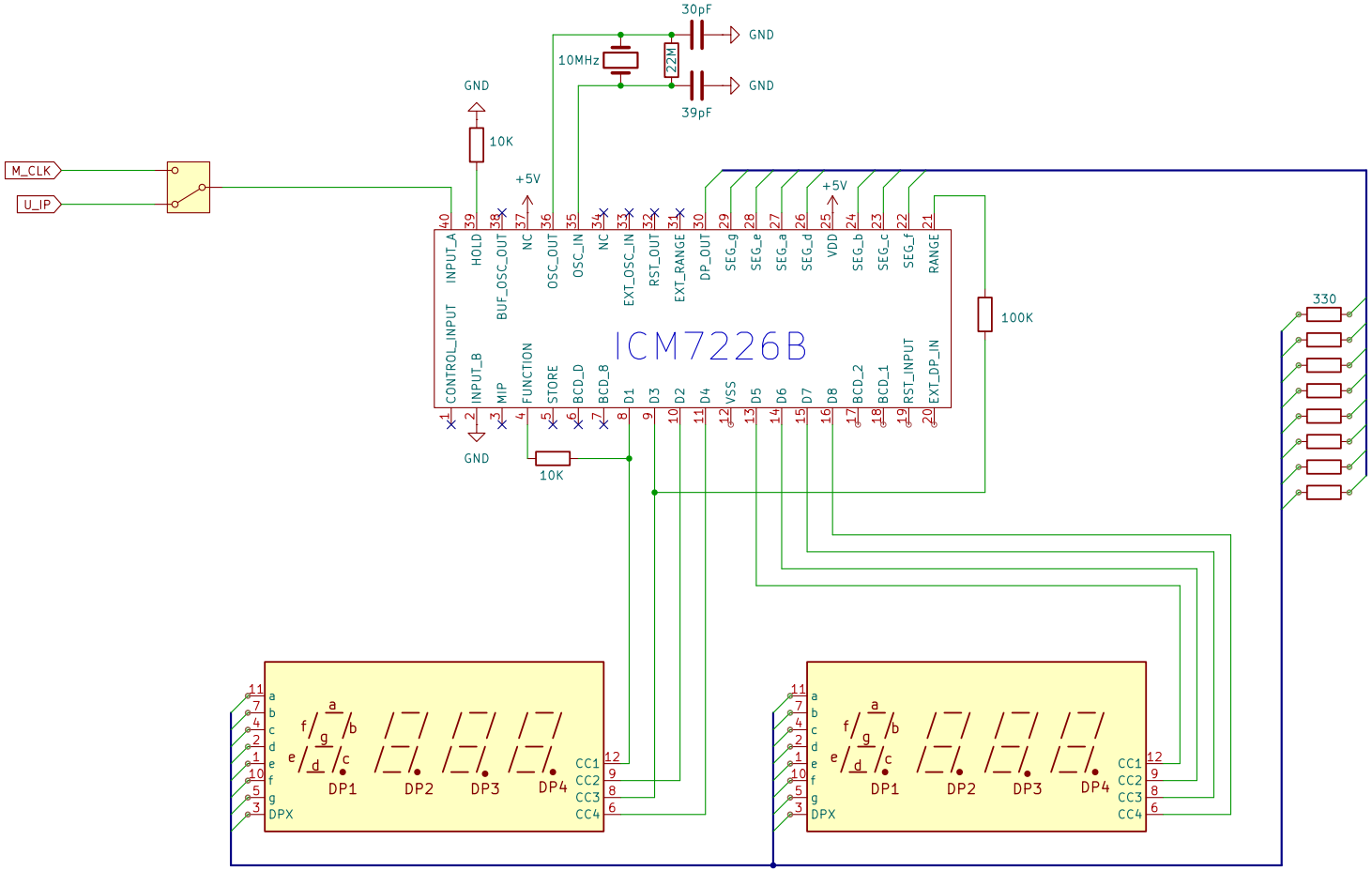
E Schematics

This appendix contains the full hardware schematics for the Synapse-191 computer. Each schematic corresponds to a specific module described in Section 5 of this report, including the Control Unit, Register Driver, IO Module, and various registers and memory components.

The schematics have been included pages for clarity and resolution. They provide a detailed view of the wiring, chip connections, and signal routing used in the physical implementation, but often do not include things like denoising capacitors or LED indicators. The diagrams are intended to complement the modular descriptions in the main text and serve as a reference for replication, debugging, or further development.

All schematics were created during the final build phase and reflect the working configuration of the system as tested.


ASTABLE
74LS76
0.01uF
GND
+5V
2K
1K
10K
RC–TIME SELECT
0.1nF
10uF
1uF
0.01uF
1nF
MONOSTABLE
200K
SELECT
74LS173
HLT
M_CLK
RESUME
680K
4.7uF
CLOCK SPLITTER
74LS123
RCext
Cext
D_CLK
M_CLK
POR
20uF
22K
470K
RESET
MASTER CLOCK
J. Heit
Sheet: /
File: master_clock.kicad_sch
Title:
Size: A4
Date: 2025
Rev: 1
KiCad E.D.A. 8.0.9
Id: 1/1

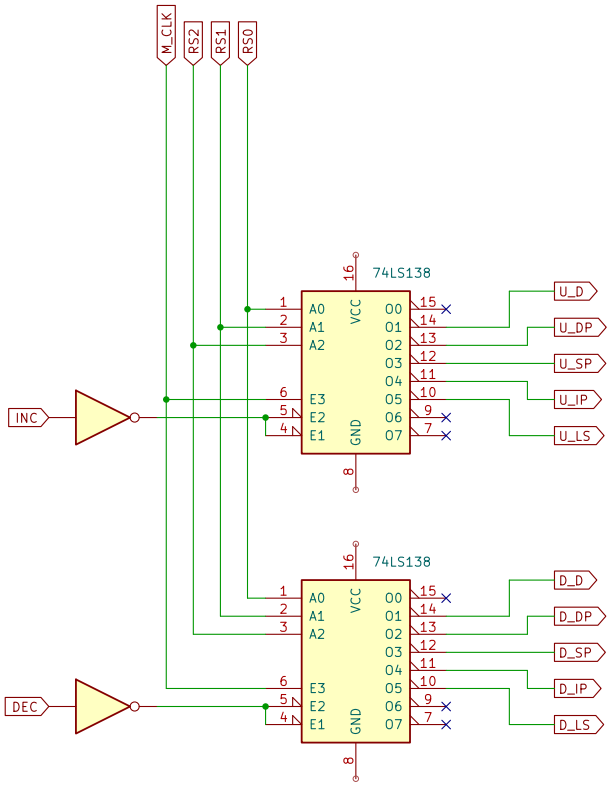


FREQUENCY DISPLAY MODULE

Sheet: /
File: timer.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.9		Id: 1/1

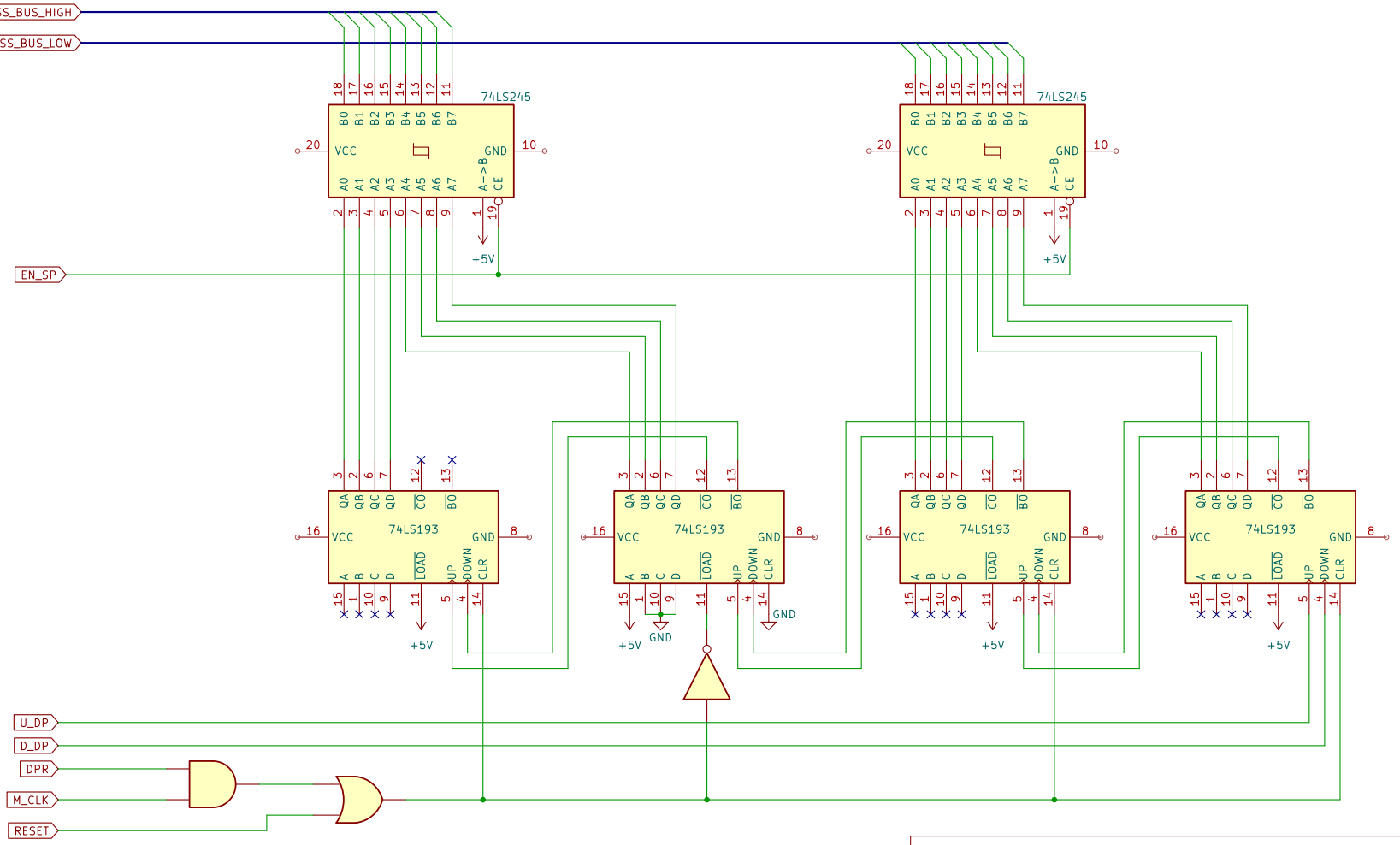


REGISTER DRIVER

Sheet: /
File: register_driver.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.9		Id: 1/1



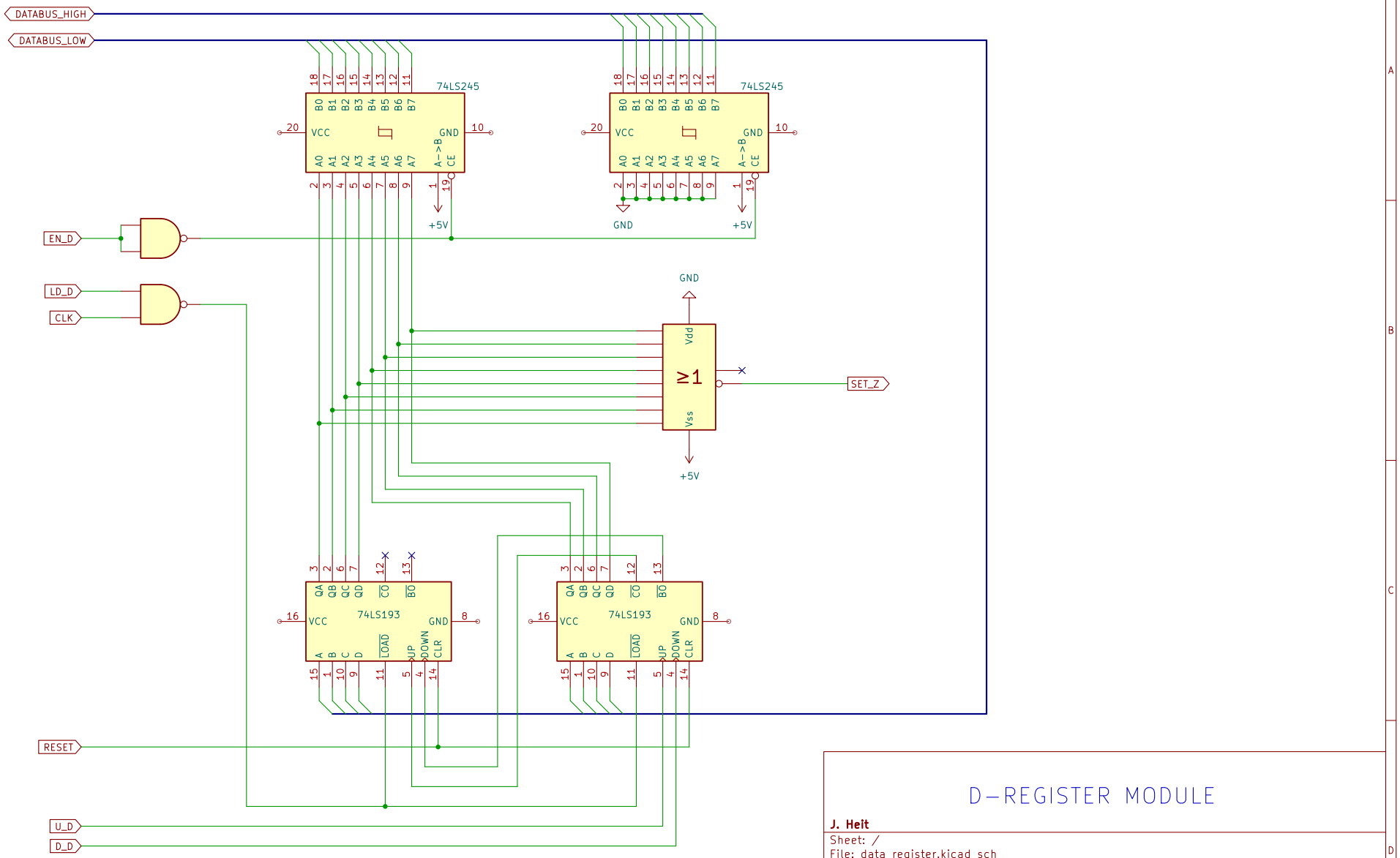
DP–REGISTER MODULE

J. Heit

Sheet: /
File: data_pointer_register.kicad_sch

Title:

Size: A4	Date: 2025	**Rev: 1**
KiCad E.D.A. 8.0.7		Id: 1/1



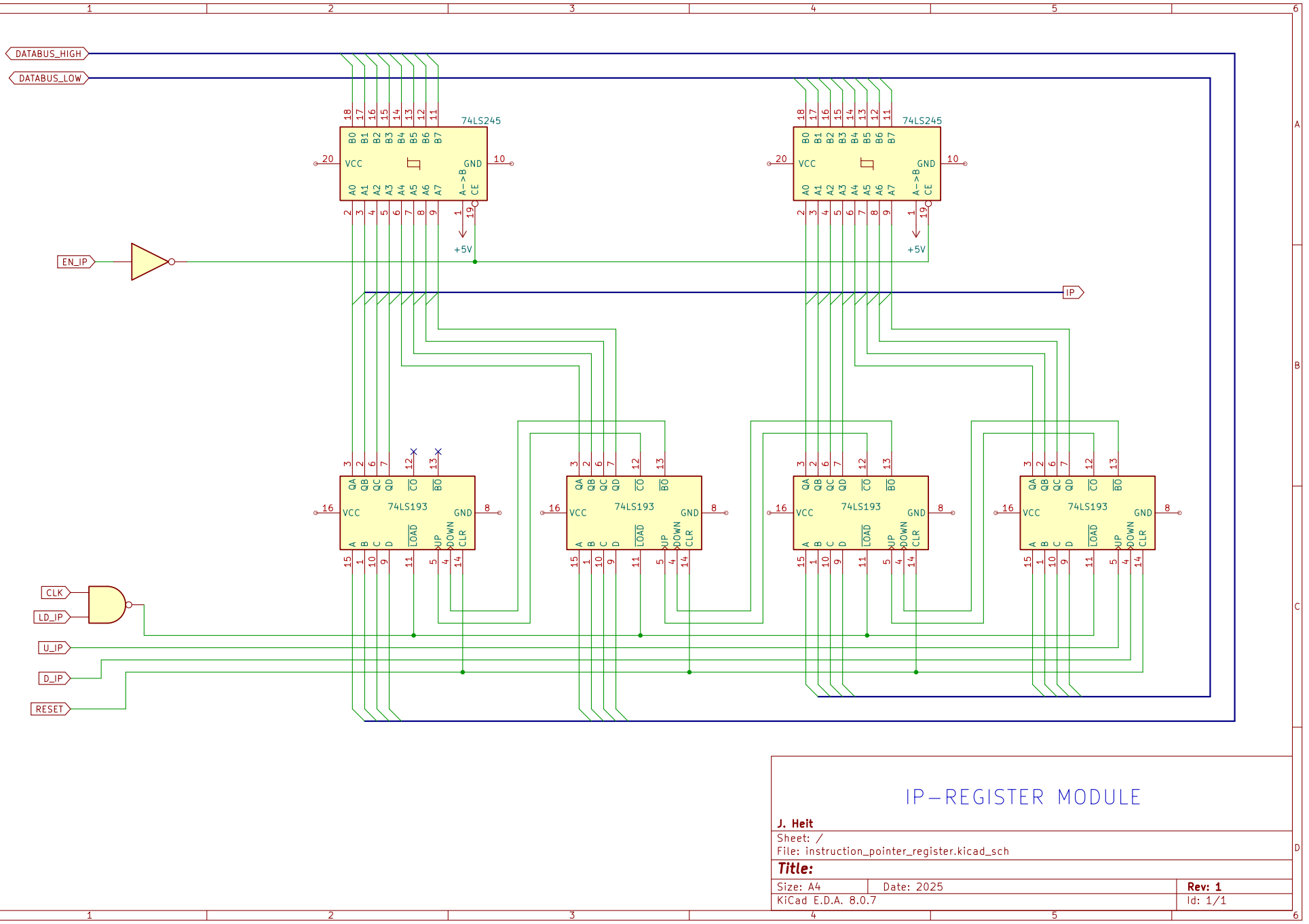
D-REGISTER MODULE

J. Heit

Sheet: /
File: data_register.kicad_sch

Title:

Size: A4	Date: 2025	**Rev: 1**
KiCad E.D.A. 8.0.9		Id: 1/1



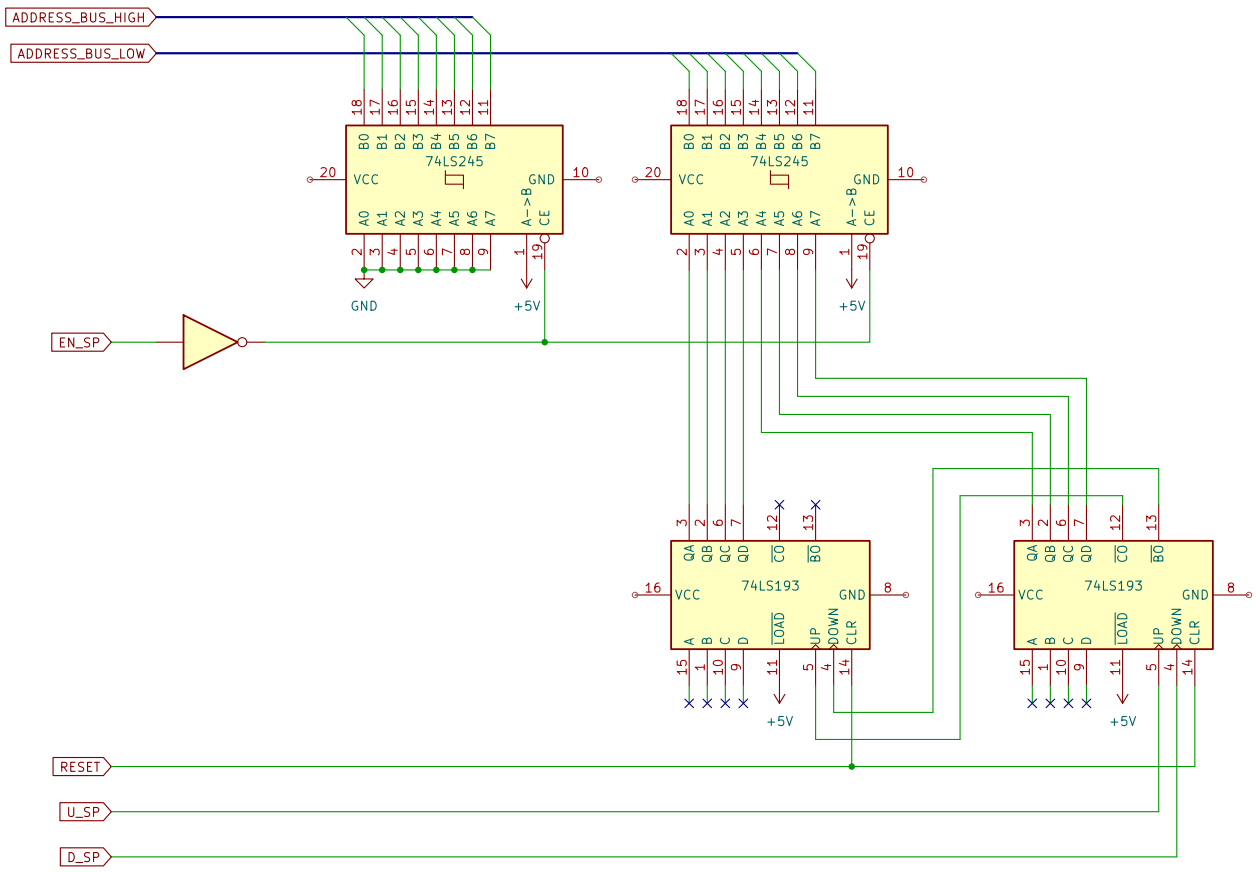
IP–REGISTER MODULE

J. Heit

Sheet: /
File: instruction_pointer_register.kicad_sch

Title:

Size: A4	Date: 2025	**Rev: 1**
KiCad E.D.A. 8.0.7		Id: 1/1



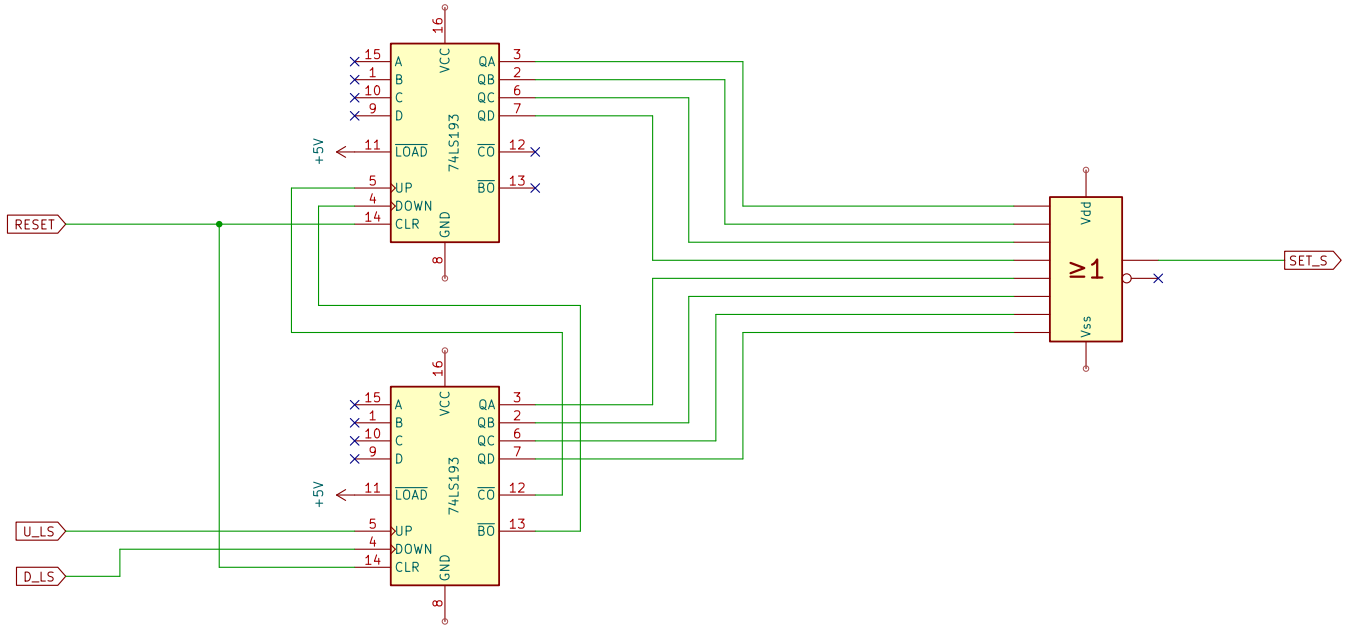
SP—REGISTER MODULE

J. Heit

Sheet: /
File: stack_pointer_register.kicad_sch

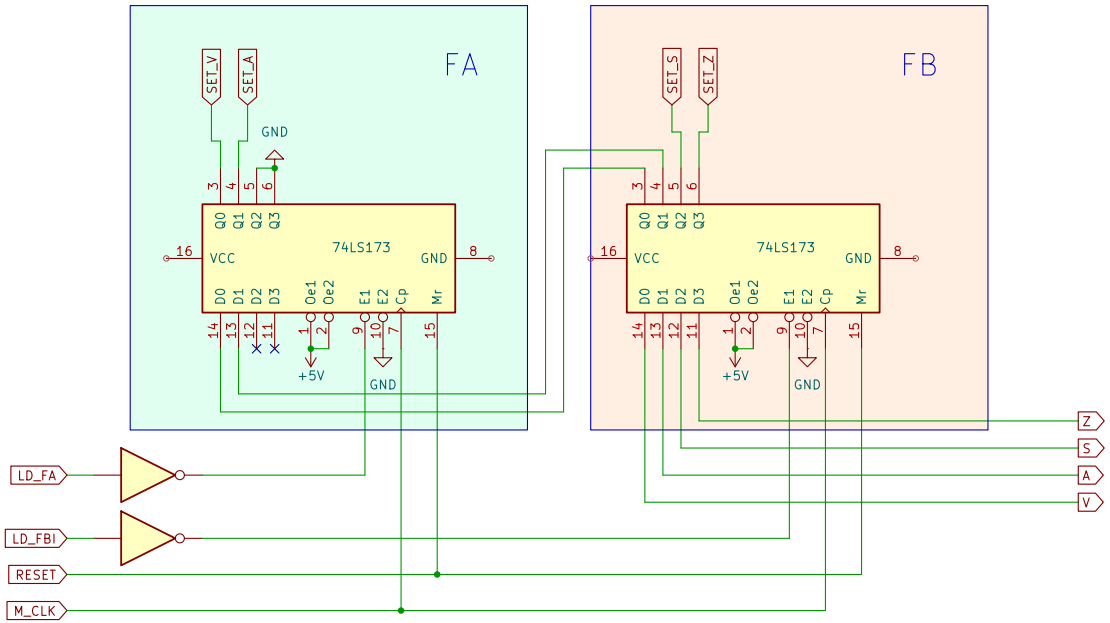
Title:

Size: A4	Date: 2025	**Rev: 1**
KiCad E.D.A. 8.0.7		Id: 1/1



LS-REGISTER MODULE

Sheet: /		
File: loop_skip_register.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.9		Id: 1/1

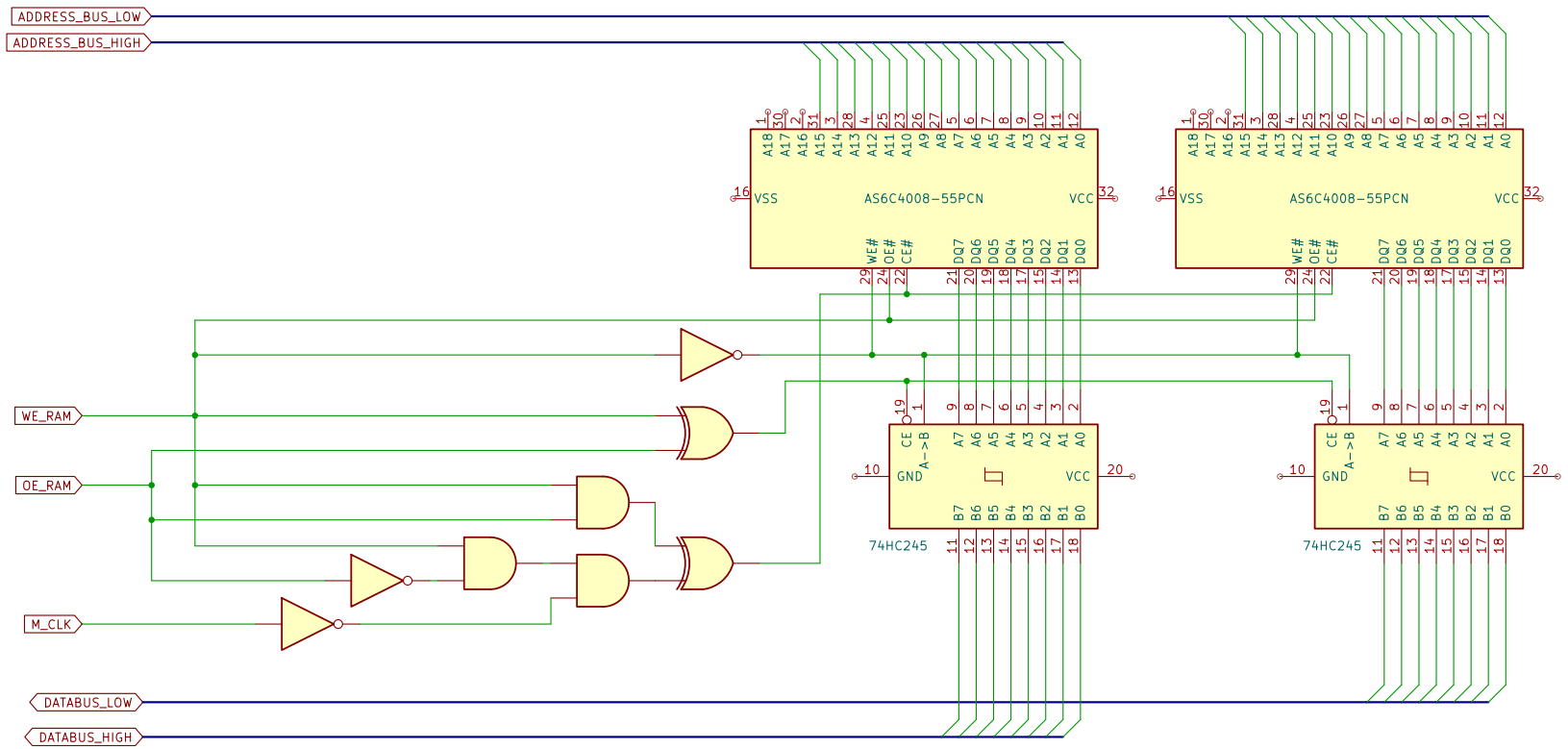


FLAG REGISTERS

Sheet: /
File: flag_registers.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.9		Id: 1/1



OE	WE	CLK	CE#	OE#	WE#	A->B	EN	
0	0	0	0	0	1	1	0	Show, but do not send to bus.
0	0	1	0	0	1	1	0	Show, but do not send to bus.
0	1	0	1	1	0	0	1	Prepare to load value from bus.
0	1	1	0	1	0	0	1	Load value from bus.
1	0	0	0	0	1	1	1	Send value to bus.
1	0	1	0	0	1	1	1	Send value to bus.
1	1	0	1	1	0	0	0	Should not happen (chip disabled).
1	1	1	1	1	0	0	0	Should not happen (chip disabled).

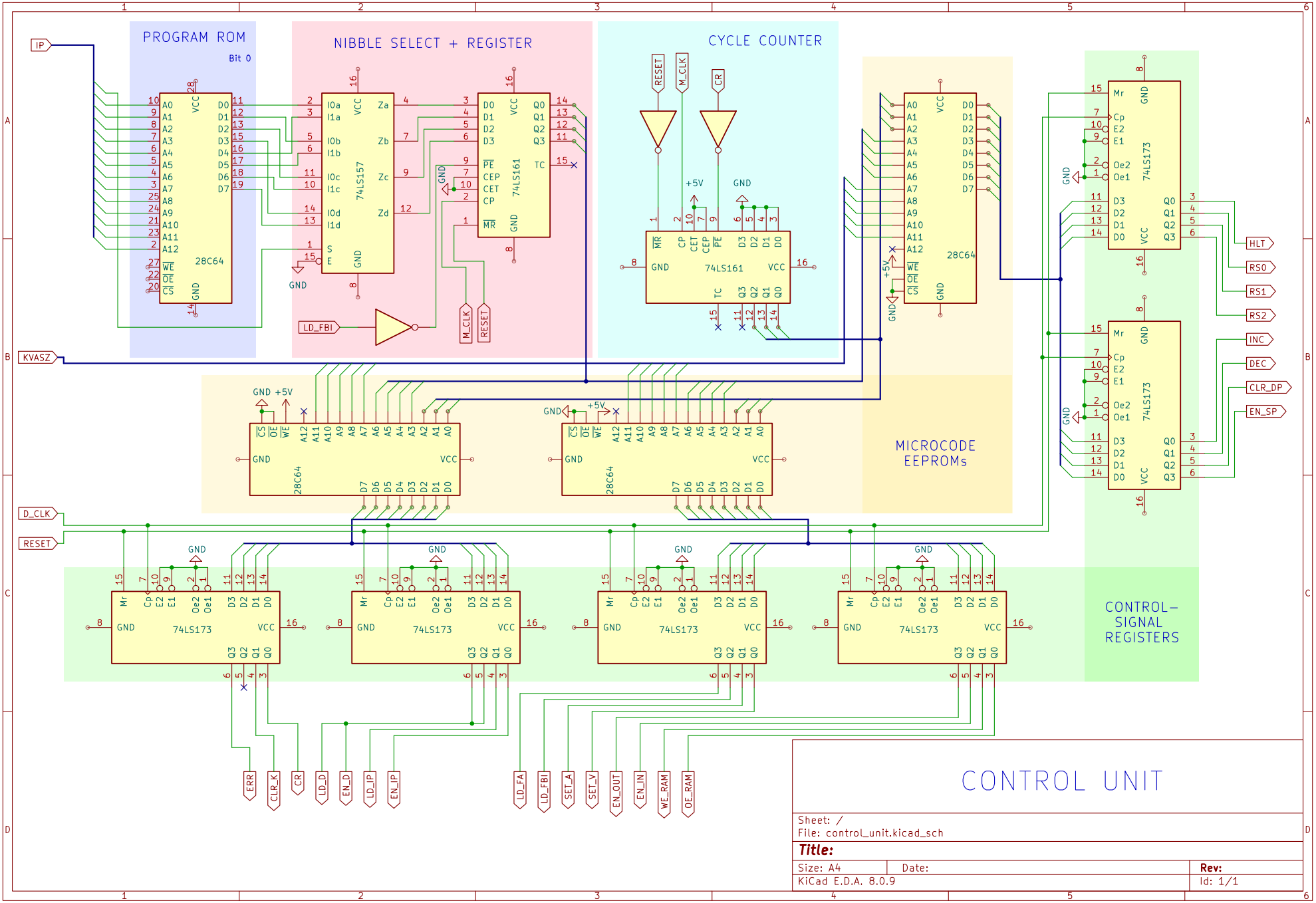
RAM MODULE

NOTE: Logic is needed to be able to intercept the output and show this on a LED array, while not making the data available on the bus.

Sheet: /
File: ram.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.7		Id: 1/1

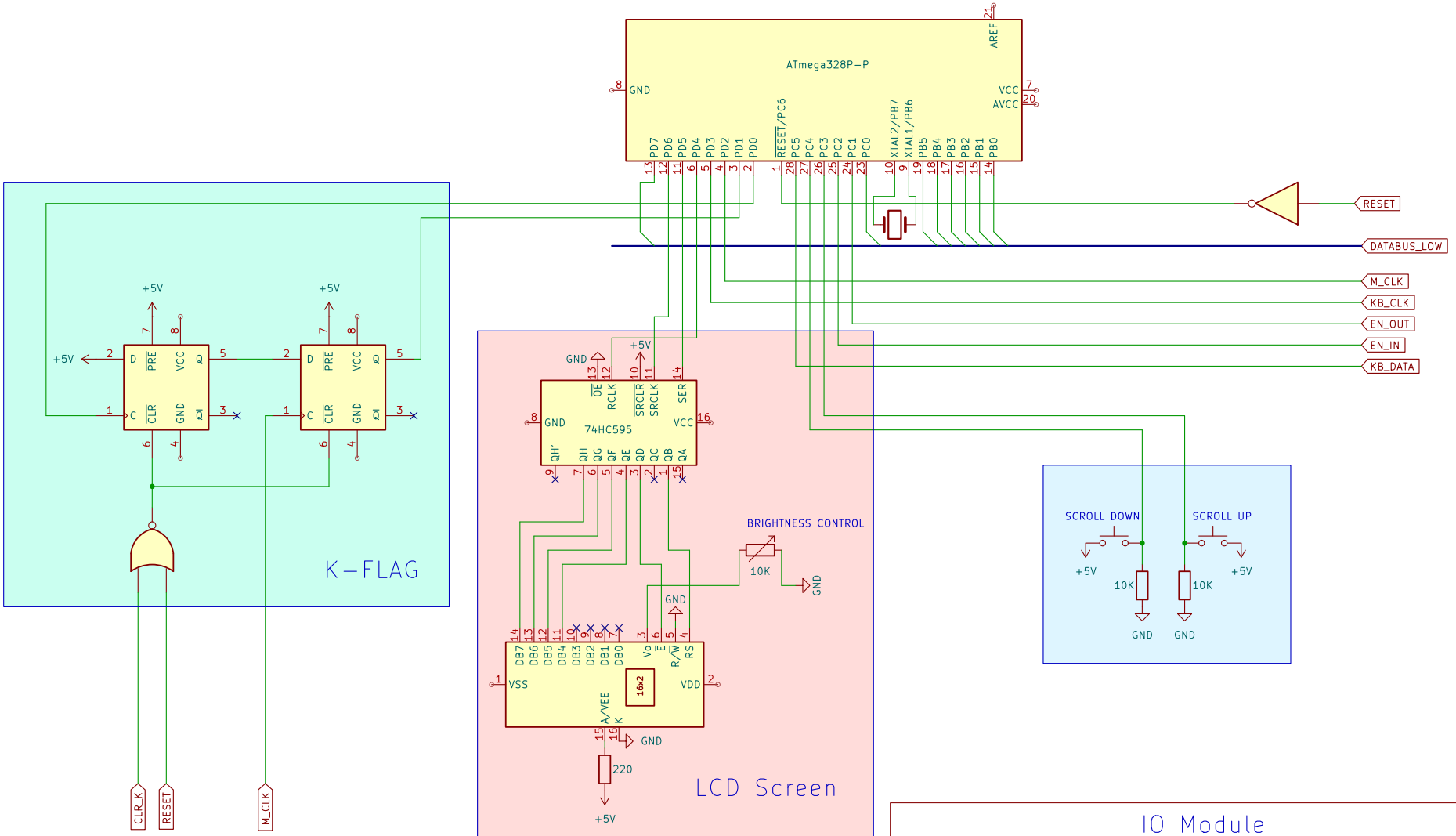


CONTROL UNIT

Sheet: /
File: control_unit.kicad_sch

Title:

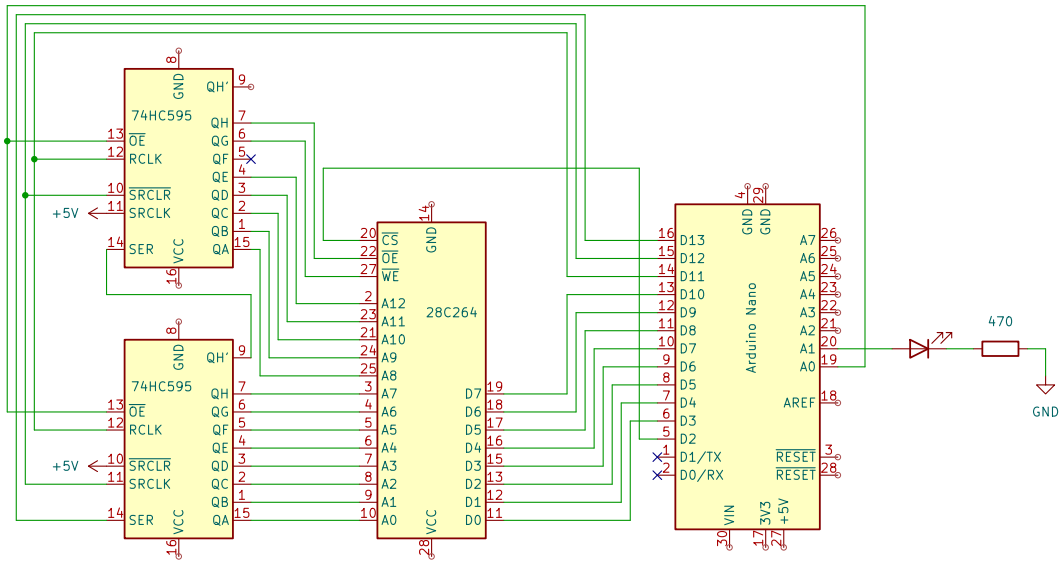
Size: A4 | Date: | **Rev:**
KiCad E.D.A. 8.0.9 | Id: 1/1



IO Module

Firmware: https://github.com/jorenheit/bfcpu/tree/main/src/io_module

Sheet: /		
File: io_module.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.9		Id: 1/1



EEPROM PROGRAMMER

J. Heit

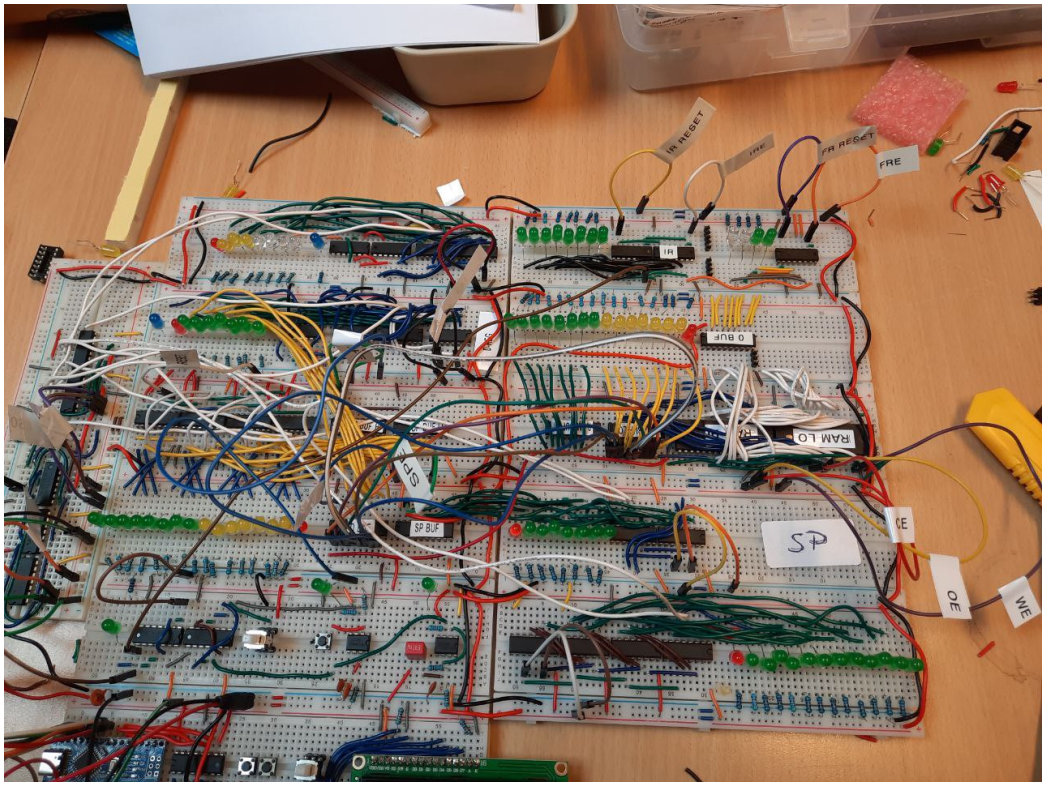
Sheet: /

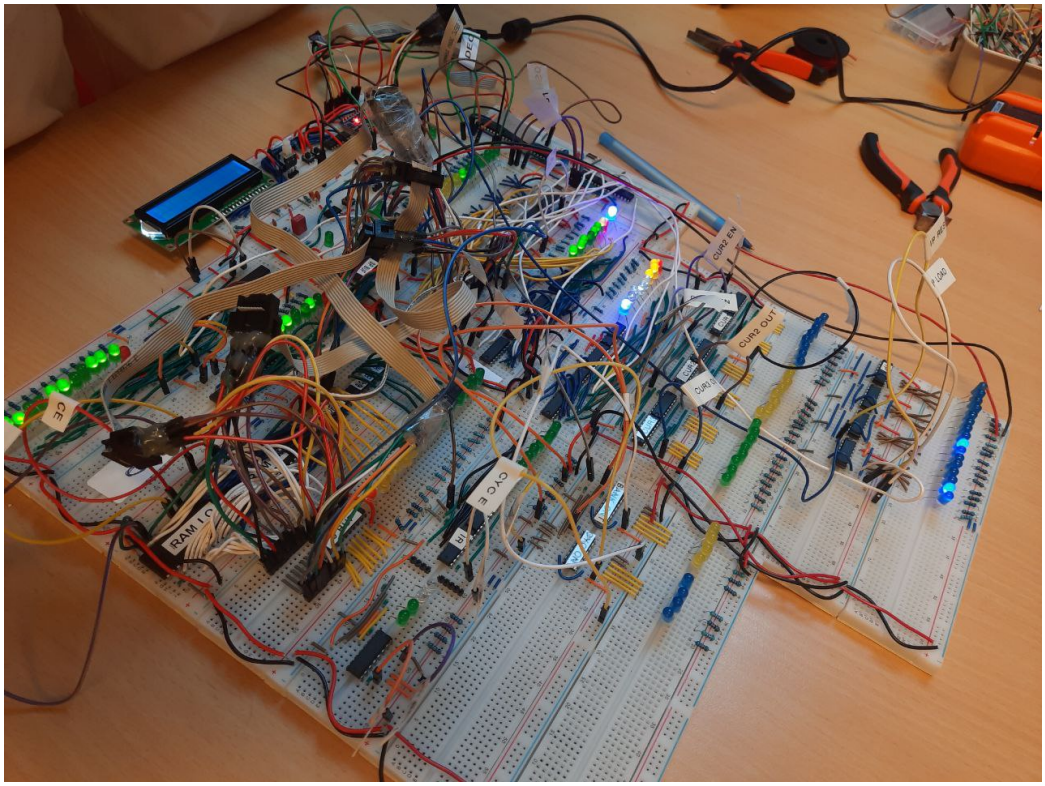
File: eeprom_programmer.kicad_sch

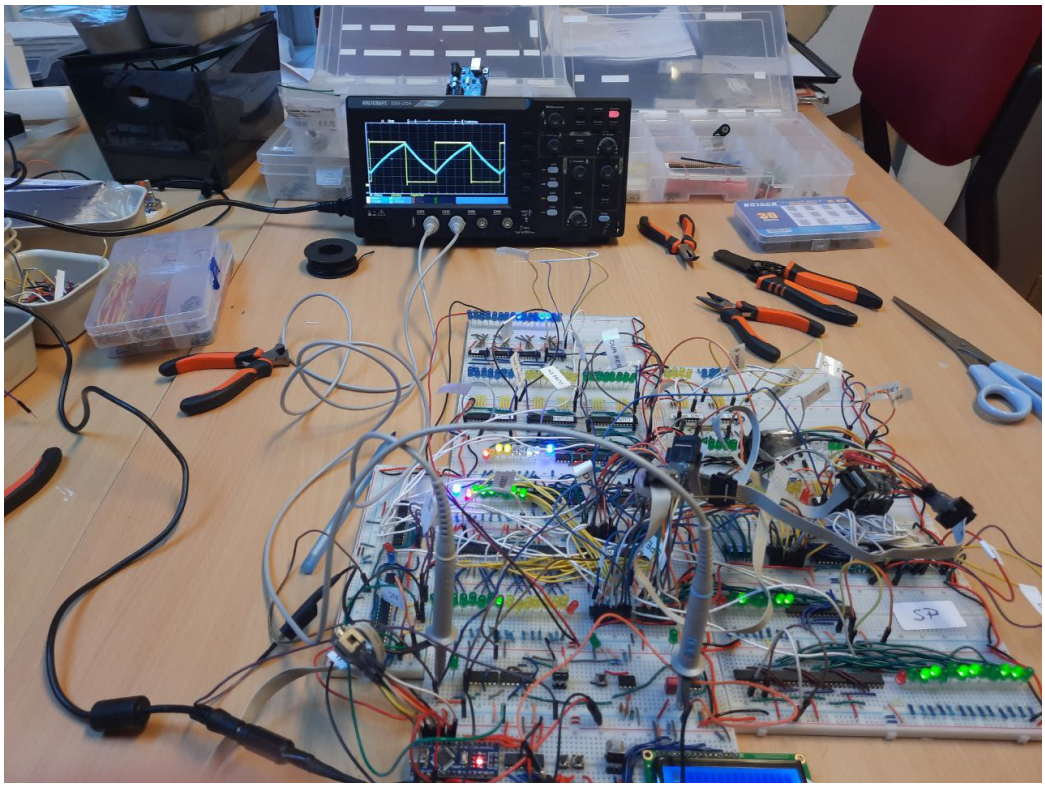
Title:

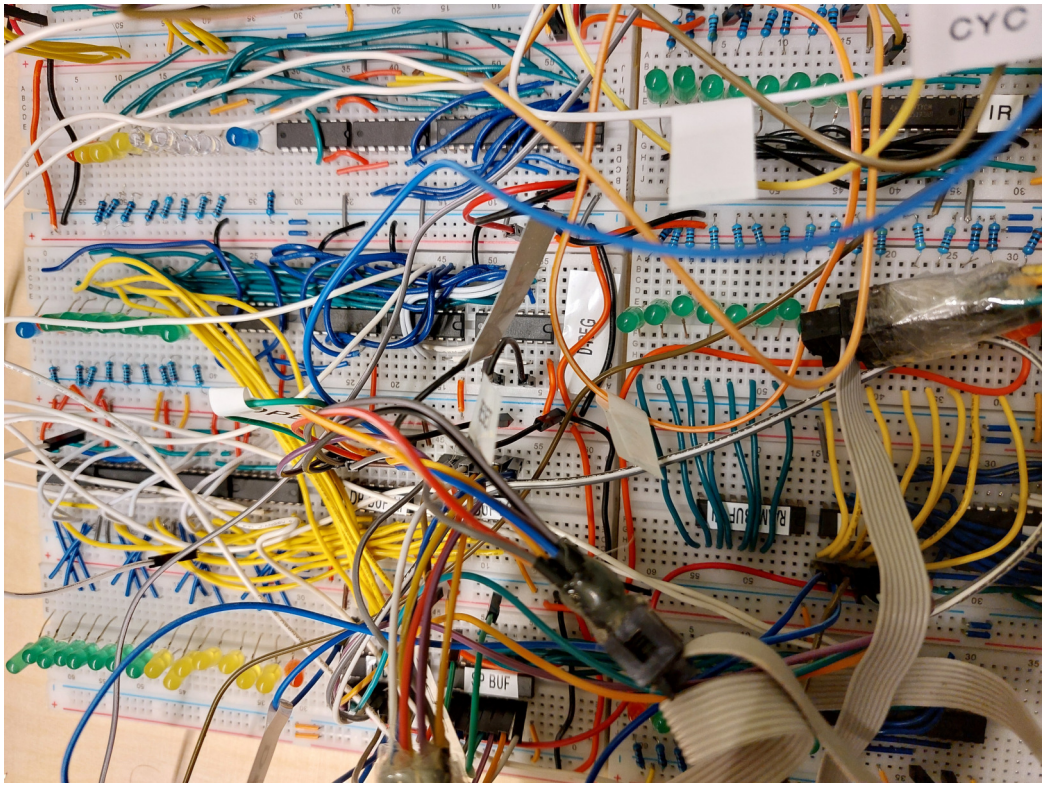
Size: A4	Date: 2025	**Rev: 1**
KiCad E.D.A. 8.0.7		Id: 1/1

F Flashback

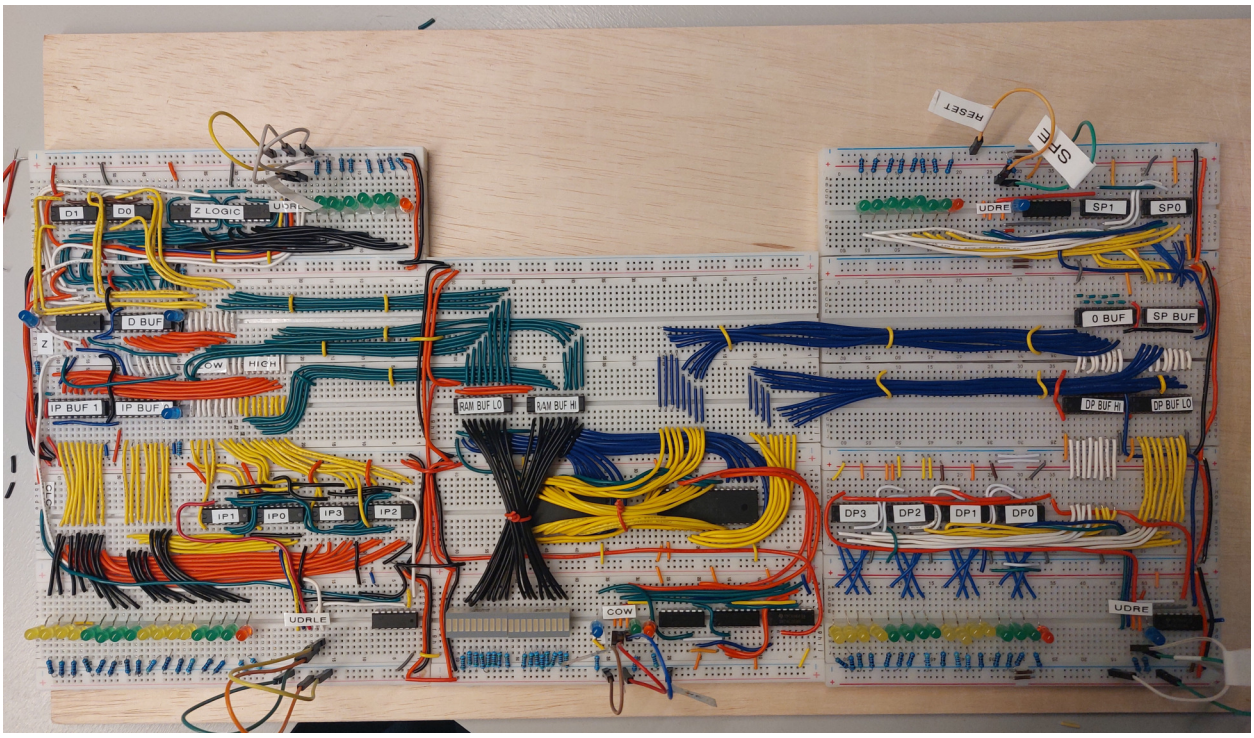


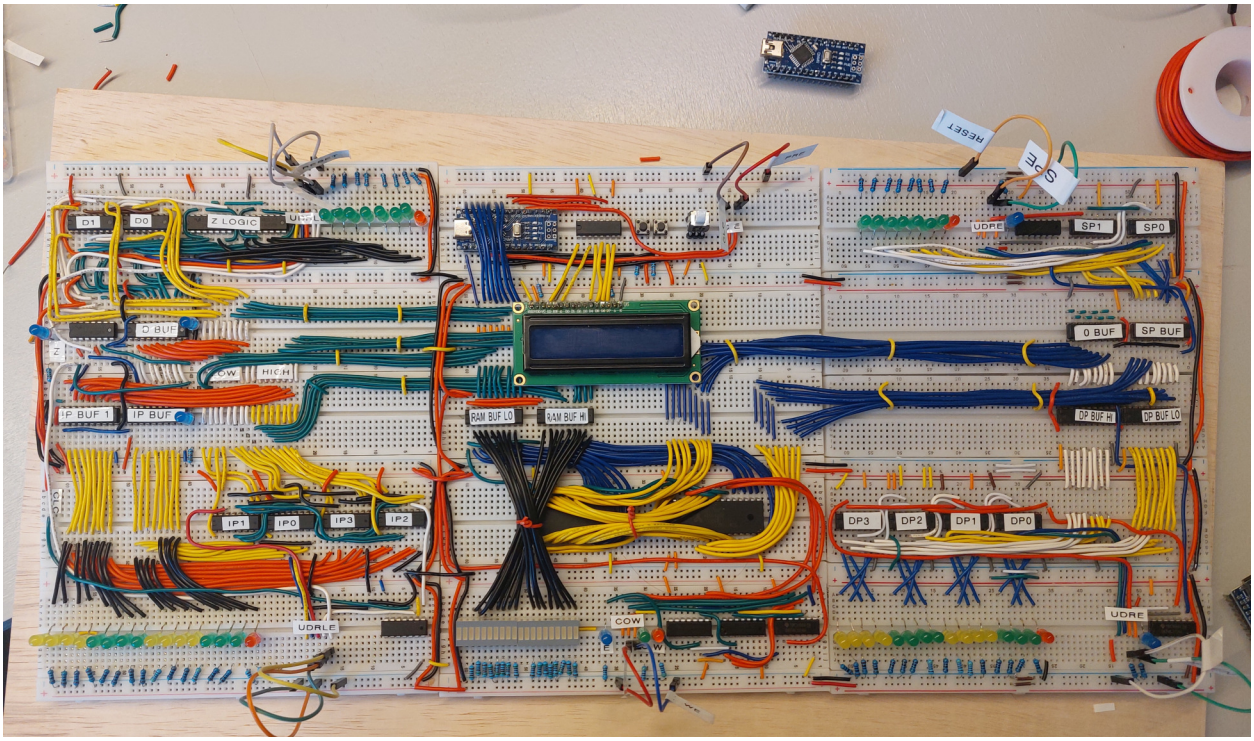


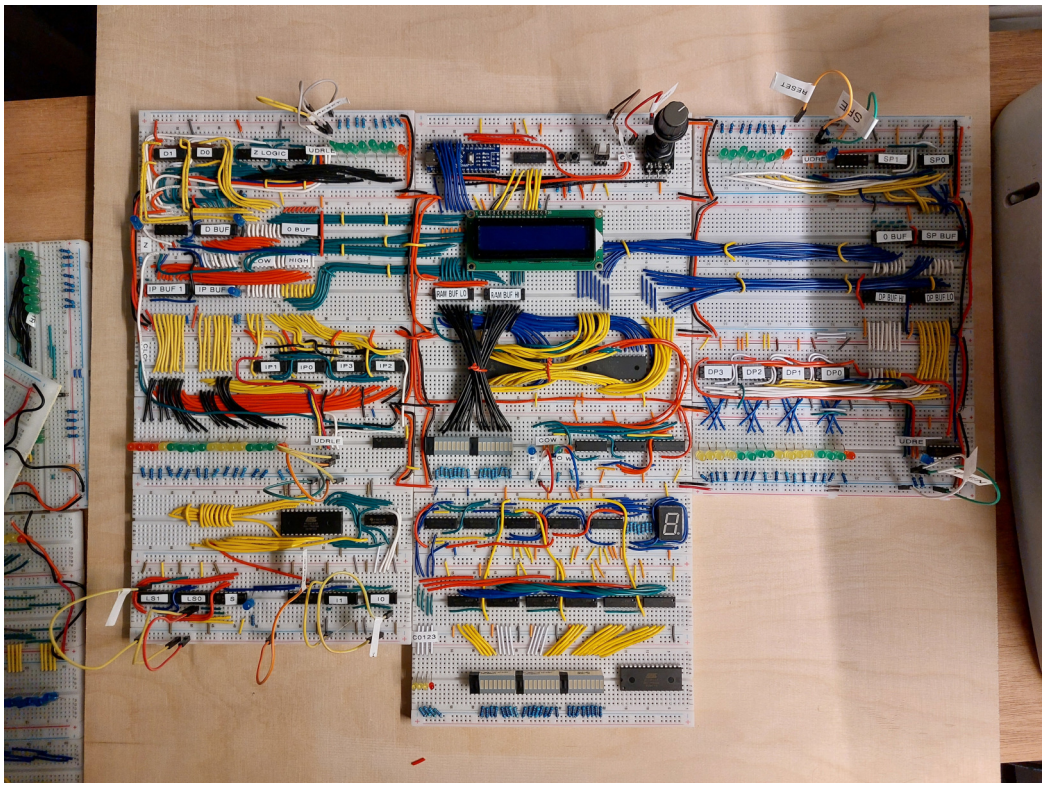

SP

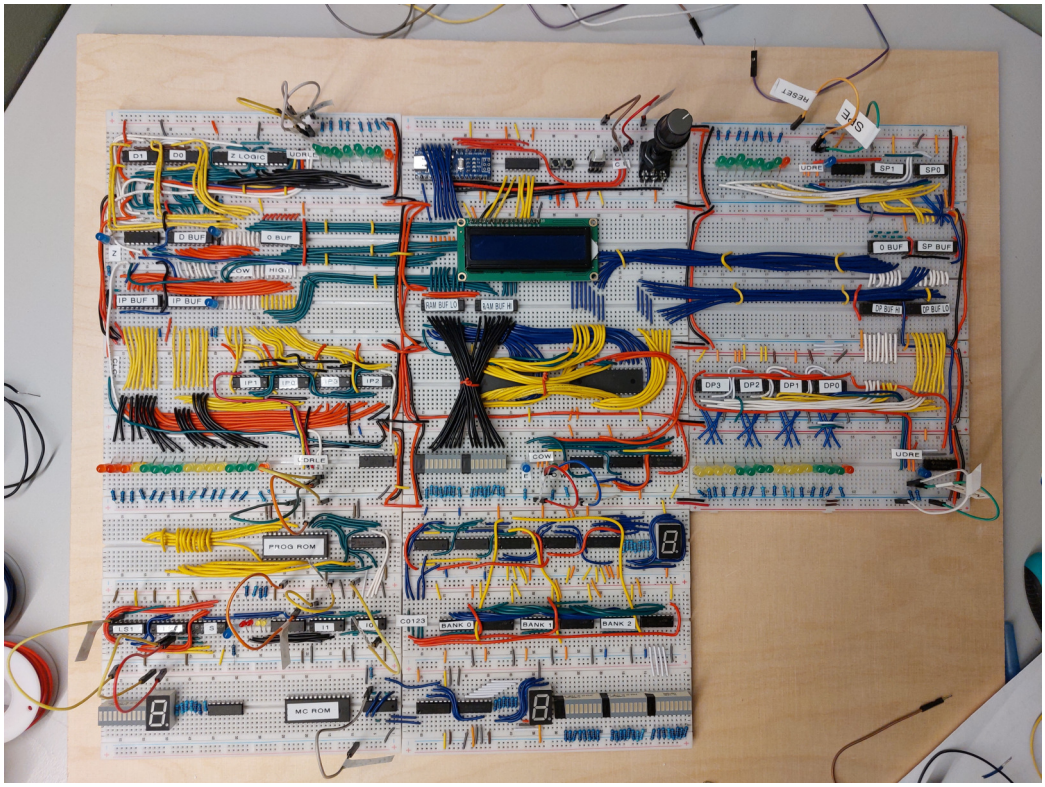

CYC
IR
BUF

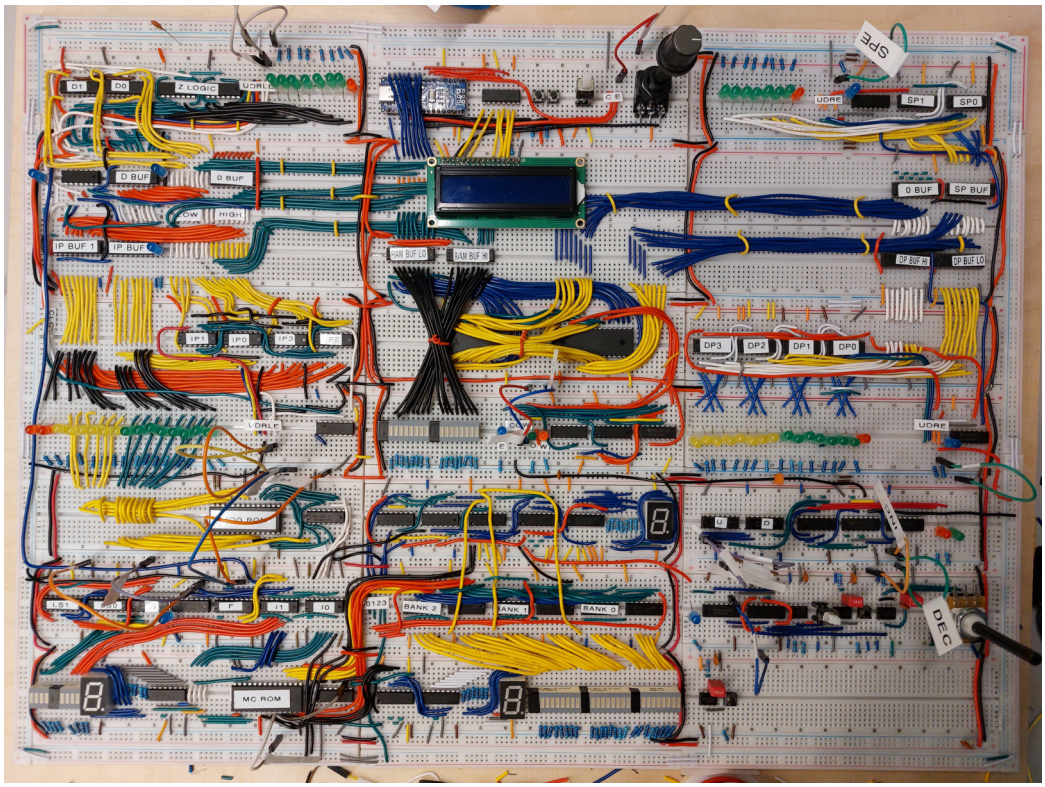

VOLTCRAFT DSO-2064

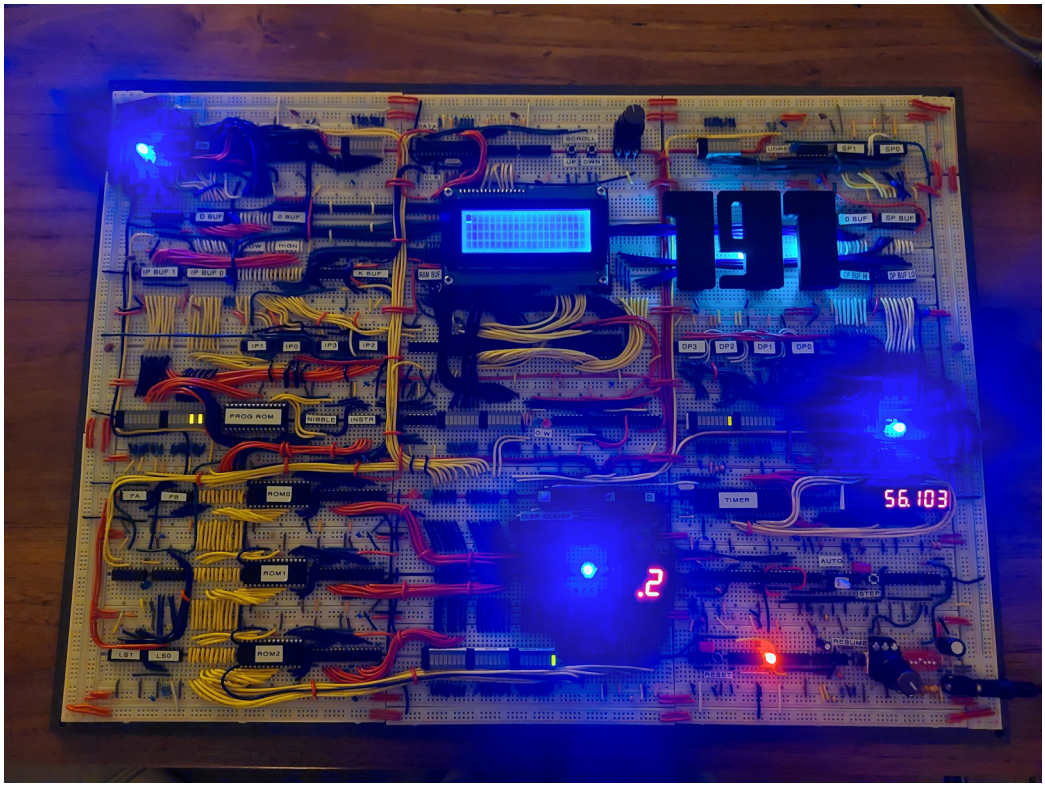

RESET
SPI
D1
D0
Z LOGIC
UDRE
SP1
SP0
D BUF
0 BUF
SP BUF
Z
COW
HIGH
IP BUF 1
IP BUF
RAM BUF LO
RAM BUF HI
DP BUF HI
DP BUF LO
IP1
IP0
IP3
IP2
DP3
DP2
DP1
DP0
UDRLE
COW
UDRE


RESET
SPI
D1
D0
Z LOGIC
UDRE
SP1
SP0
D BUF
0 BUF
SP BUF
Z
COW
HIGH
IP BUF 1
IP BUF
RAM BUF LO
RAM BUF HI
DP BUF HI
DP BUF LO
IP1
IP0
IP3
IP2
DP3
DP2
DP1
DP0
UDRLE
COW
UDRE


RESET
SP1
SP0
0 BUF
SP BUF
DP3
DP2
DP1
DP0
UDRE


RESET
SP1
SP0
0 BUF
SP BUF
DP3
DP2
DP1
DP0
UDRE
PROG ROM
MC ROM
BANK 0
BANK 1
BANK 2


D1
D0
Z LOGIC
UDRE
SP1
SP0
SPE
D BUF
0 BUF
SP BUF
IP BUF 1
IP BUF 0
DP3
DP2
DP1
DP0
MC ROM
BANK 2
BANK 1
BANK 0
DEC


191
56.103
.2
PROG ROM
NIBBLE
INSTR
ROM0
ROM1
ROM2
TIMER
AUTO
STEP
RESUME
RESET

References

[1] WikiPedia, *Brainfuck*, https://en.wikipedia.org/wiki/Brainfuck

[2] Esolangs, *Brainfuck*, https://esolangs.org/wiki/Brainfuck

[3] Esolangs, *Brainfuck*, https://esolangs.org/wiki/Random_Brainfuck

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

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

[6] Stefan Heule, *How Many x86-64 Instructions Are There Anyway?*, https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/, 2016

[7] Daniel Mangum, *RISC-V Bytes: Introduction to Instruction Formats*, https://danielmangum.com/posts/risc-v-bytes-intro-instruction-formats/, 2021

[8] unknown author, *Wikipedia* https://en.wikipedia.org/wiki/Brainfuck

[9] unknown author, *Unknown Source*

[10] Marijn Stevens, *Github*, https://github.com/MarijnStevens/brainfuck-collection/blob/master/bf/golden2.b

[11] Daniel B. Cristofani, *brainfuck.org*, https://brainfuck.org/e.b

[12] Troemax, *Reddit*, https://www.reddit.com/r/brainfuck/comments/847vl0/prime_number_generator_in_brainfuck/

[13] Alexandru, *Stackexchange*, https://codegolf.stackexchange.com/questions/84/interpret-brainfuck

[14] Mixtela, *Github*, https://github.com/mitxela/bf-tic-tac-toe

[15] Texas Instruments, *Quadruple 2-Input Positive-NAND Gates*, 74LS00 Data Sheet, Dec. 1983 (Rev. 2003)

[16] Texas Instruments, *Quadruple 2-Input Positive-NOR Gates*, 74LS02 Data Sheet, Dec. 1983 (Rev. 1988)

[17] Texas Instruments, *Hex Inverters*, 74LS04 Data Sheet, Dec. 1983 (Rev. 2004)

[18] Texas Instruments, *Quadruple 2-Input Positive-AND Gates*, 74LS08 Data Sheet, Dec. 1983 (Rev. 1988)

[19] Texas Instruments, *Hex Schmitt-Trigger Inverters*, 74LS14 Data Sheet, Dec. 1983 (Rev. 2016)

[20] Texas Instruments, *Quadruple 2-Input Positive-OR Gates*, 74LS32 Data Sheet, Dec. 1983 (Rev. 1988)

[21] Texas Instruments, *BCD-To-Seven-Segment Decoders/Drivers*, 74LS48 Data Sheet, Mar. 1974 (Rev. 1988)

[22] Texas Instruments, *Dual D-Type Positive-Edge-Triggered Flip-Flops with Preset and Clear*, 74LS74 Data Sheet, Oct. 1983 (Rev. 1988)

[23] Texas Instruments, *Dual JK-Flip-Flops with Preset and Clear*, 74LS76 Data Sheet, Dec. 1983 (Rev. 1988)

[24] Texas Instruments, *Quadruple 2-Input Exclusive-OR Gates*, 74LS86 Data Sheet, Dec. 1972 (Rev. 1988)

[25] Texas Instruments, *Retriggerable Monostable Multivibrators*, 74LS123 Data Sheet, Dec. 1983 (Rev. 1988)

[26] Texas Instruments, *3-Line to 8-Line Decoders/Demultiplexers*, 74LS138 Data Sheet, Dec. 1972 (Rev. 1988)

[27] Texas Instruments, *Quadruple 2-Line to 1-Line Data Selectors/Multiplexers*, 74LS157 Data Sheet, Dec. 1982 (Rev. 2022)

[28] Texas Instruments, *Synchronous 4-Bit Counters*, 74LS161 Data Sheet, Oct. 1976 (Rev. 1988)

[29] Texas Instruments, *4-Bit D-Type Registers with 3-State Outputs*, 74LS161 Data Sheet, Oct. 1976 (Rev. 1999)

[30] Texas Instruments, *Synchronous 4-Bit Up/Down Counters*, 74LS193 Data Sheet, Dec. 1972 (Rev. 1988)

[31] Texas Instruments, *Octal Bus Transceivers With 3-State Outputs*, 74LS245 Data Sheet, Oct. 1976 (Rev. 2016)

[32] Texas Instruments, *8-Bit Shift Registers With 3-State Output Registers*, 74LS245 Data Sheet, Dec. 1982 (Rev. 2021)

[33] Texas Instruments, *xx555 Precision Timers*, NE555 Data Sheet, Sep. 1973 (Rev. 2014)

[34] Atmel, *64K (8K x 8) Parallel EEPROM with Page Write and Software Data Protection*, AT28C64B Data Sheet, 2009

[35] Alliance Memory Inc., *512K X 8 BIT LOW POWER CMOS SRAM*, AS6C2008 Data Sheet, Aug. 2009

[36] Motorolla, *B-Suffix Series CMOS Gates*, MC14068B Data Sheet, Aug. 2009

[37] Intersil, *8-Digit, Multi-Function, Frequency Counter/Timer*, ICM7226B Data Sheet, Aug. 2009