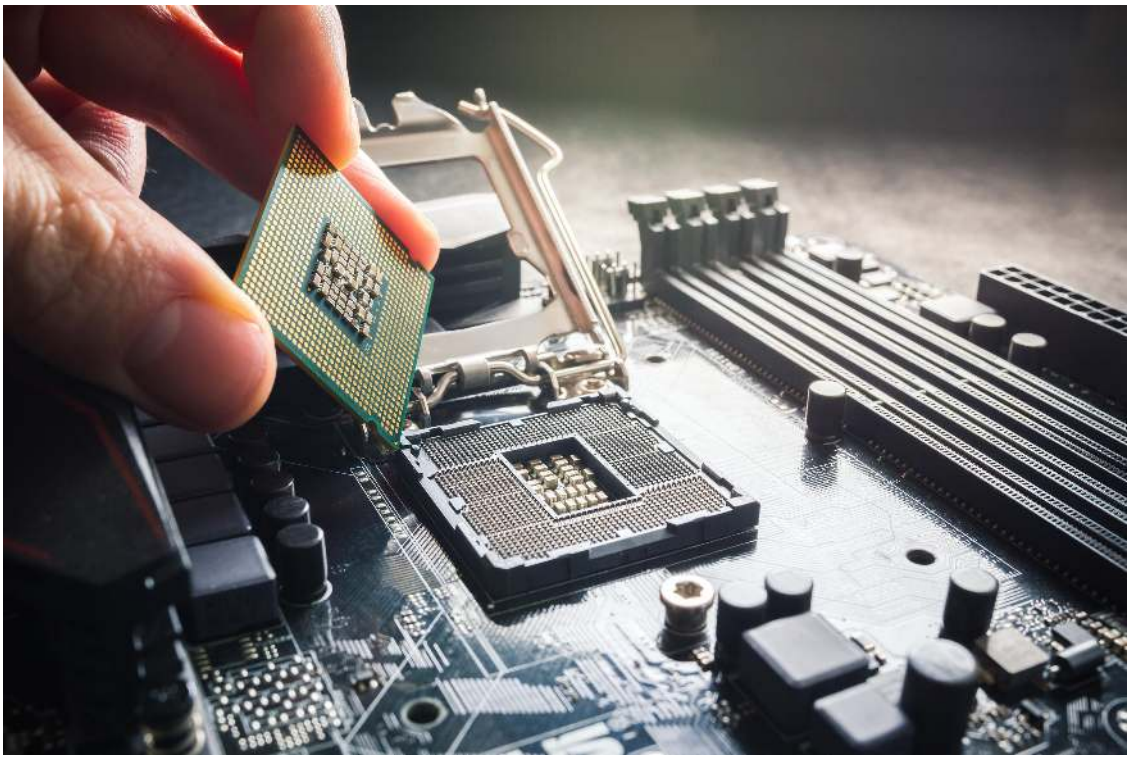
Synapse-191

Joren Heit, Arthur Topal

2025



Contents

1 Introduction

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

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 Brainf*ck machine. However, rather than viewing BF as a language that needs to be compiled or interpreted, why not view it as the instruction set of a, yet to exist, Brainf*ck-CPU?

We wanted to build this Brainf*ck CPU without making use of any programmable chips; only use Transistor-Transitor-Logic (TTL) chips such as registers, buffers and (de)multiplexers in addition to the necessary RAM and ROM. The computer had to be built on breadboards, as it was inspired by Ben Eater's 8-bit breadboard computer [**?**]. 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, interpret the PS2 protocol and drive the screen, in addition to managing all the IO settings (echo, autoscroll, output-modes, etc). 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. 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`.

Document structure. Section **??** recaps BF as an ISA, Section **??** describes the architecture, Section **??** explains microcode and control sequences, Section **??** discusses hardware implementation and debugging and Section **??** covers supporting utilities. We conclude in Section **??**. Appendices contain tables, microcode listings, BOM, and schematics.

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

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 catch is that there is a total of only 8 commands available 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 **??**).

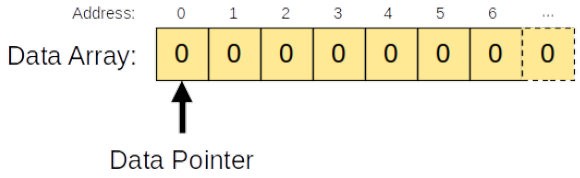


Figure 1: Initial state of the BF machine.

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

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

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

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 **??** shows a very simple implementation (about 40 lines) of such a program 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 value it's pointing or print it to standard output. The most complexity is embedded in the handling of the loop-operatores. When an opening bracket is encountered, the index of this instruction is stored in a jump-table. When the matching closing bracket is encountered and it is determined that flow should loop back, this stored index is reloaded. When a loop is being skipped, all commands within it are being skipped until the matching closing brace is found.

When the Hello World program from Wikipedia [**?**] is fed into this program, it prints out the string `Hello World!`, as expected.

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

```c
//----------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.

2.2 Brainf*ck Architecture

Von Neumann. Modern computers are built according to the von Neumann architecture [**?**], which specifies a CPU (containing registers and an ALU), a single unit of memory and input/output devices (Figure **??**). 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.

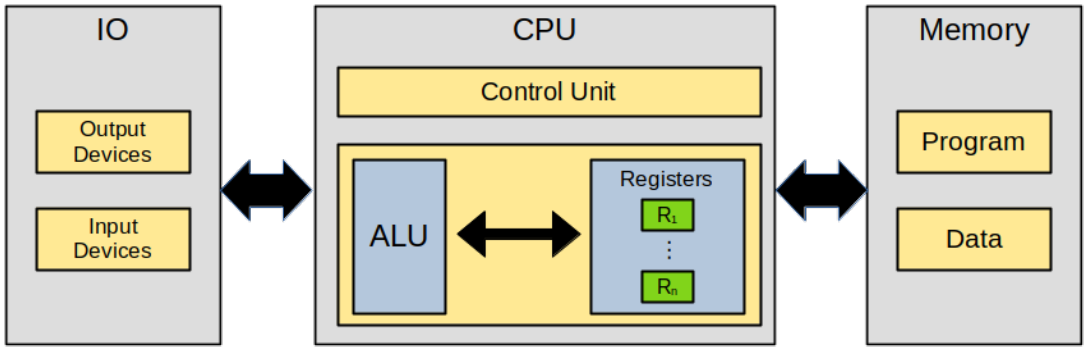


Figure 2: Schematic overview of the Von Neumann architecture.

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

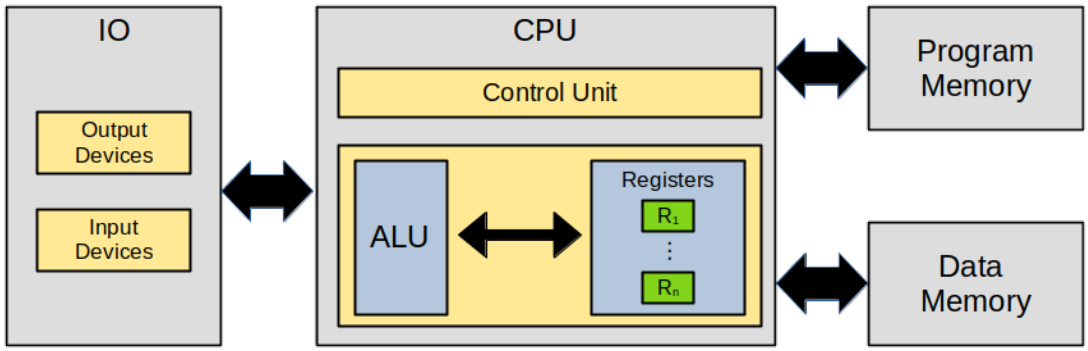


Figure 3: Schematic overview of the Harvard architecture.

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 **??**, where the program was stored in a seperate array to the data. The ALU is very limited and can only perform increment, decrement and comparison to zero.

2.3 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 1500 instructions implemented in hardware, whereas RISC processors only need to implement 50 to 100 instuctions. Even compared to RISC, the BF instruction set is tiny even compared to the smallest instruction sets in use today. This isn't necessarily a good thing; a smaller number of instructions simply means you need more of them

to perform meaningful computations, which is reflected by the fact that complex BF programs are typically very large in size.

3 Architecture

3.1 Overview

In simple terms, a BF machine consists of an array of memory-cells, together with a pointer pointing to one of these cells. The pointer can move along the array while modifying its contents one step at a time. An example of this representation in some intermediate state is shown in Figure **??**. 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).

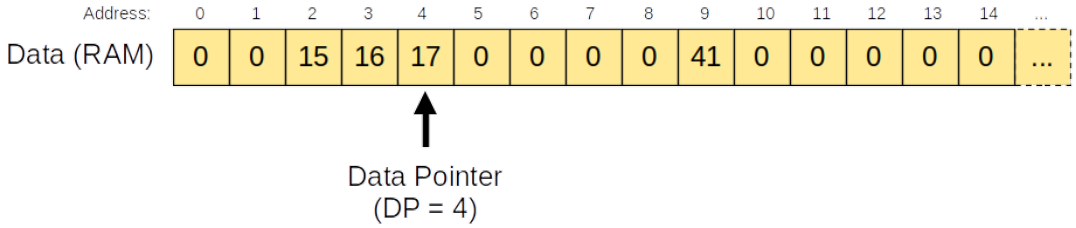


Figure 4: Example state of a BF machine.

The processor consists of three basic building blocks: registers, memory and a control unit. The ALU is missing from this list because the only operations that it needs to perform are addition and subtraction of the value 1, which can be done directly at the register-level when using up/down binary counters like the 74LS193 integrated circuit. The program (a sequence of BF instructions) is stored into Read Only Memory (ROM), whereas the data is stored in Random Access Memory (RAM). Instructions (4-bits) are 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 next computation. Figure **??** shows how each of the modules is communicating with 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 1.

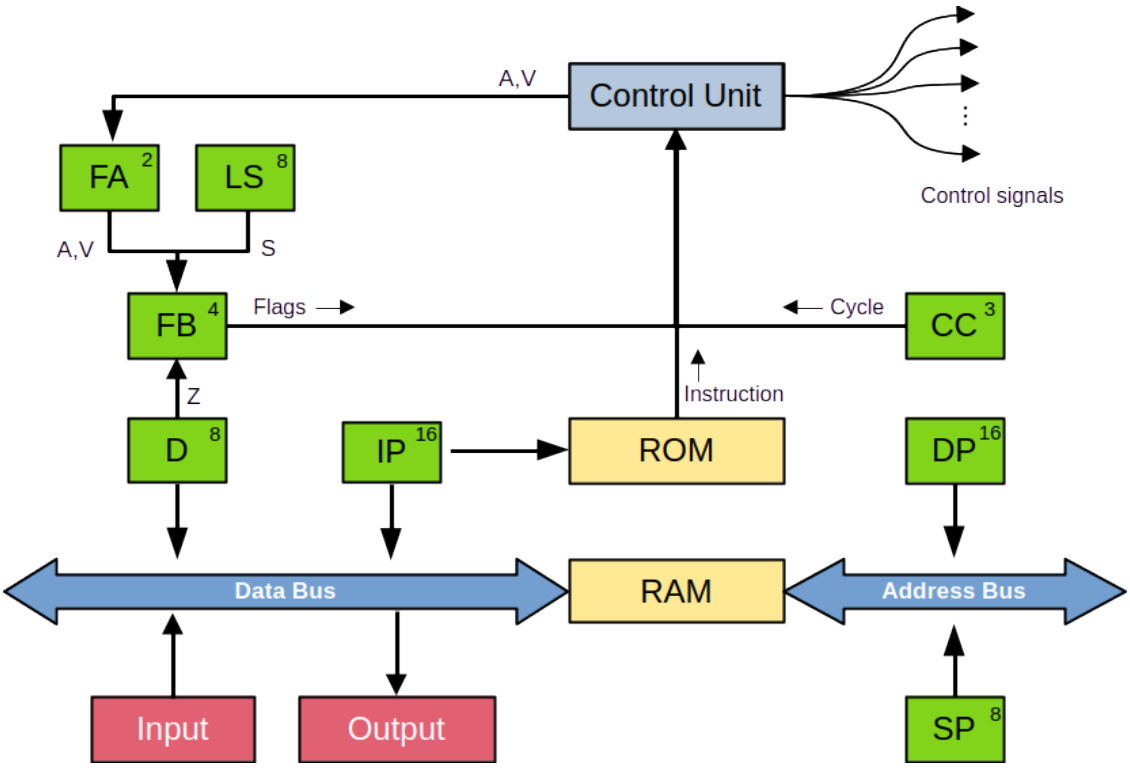


Figure 5: Connections between modules in the BF processor.

3.2 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 **??**) 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 **??**), 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 **??**, 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 **??**).

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

Outputs

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

3.3 Data Register (D)

The data register holds a representation of the value currently pointed to by the DP and can be incremented and decremented (corresponding to `+` and `-`). 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

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

Outputs

- D_OUT - 8 bits - Data outputs, connected to the databus.
- Z - Zero Flag - High when the register stores a zero.

3.4 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 **??**). 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

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

Outputs

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

3.5 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 instructoin (`]`). 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. Although possible, it is very unlikely to happen for the simple programs we intend to run.

Inputs

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

Outputs

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

3.6 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 of the D-register), the LS register is incremented from 0 to 1 and the S-flag is set. 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

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

Outputs

- S - Skip flag - set when it value is nonzero.

3.7 Flag Registers (FA and FB)

The first flag register (FA) holds two flag values A and V which are used to indicate that either the address or value respecively has changed during one of the previous instructions. For instance, is D was incremented, the V-flag is set to indicate a change of the *value* being pointed to (the value in RAM is now outdated). For a more detailed description of the function and application of these flags, refer to Section **??** and **??**. 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. The previously mentioned K-flag is specific to interactions with the IO-module and will be discussed in Section **??**.

Inputs

- SET_A - Set the address-change-flag.
- SET_V - Set the value-change-flag.
- LD(FA) - Load A and V into FA.
- LD(FB) - Load A, V (previously buffered in FA), Z and S (from D and LS) into FB.

Outputs

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

3.8 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, right after IP has been incremented. Its outputs are used as the instruction-part that go into the decoder of the CU (see Figure **??** in Section **??**).

Inputs

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

Outputs

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

3.9 Register Driver

Rather that having a seperate signal for each of the INC/DEC-inputs of each register (e.g. INC_D, INC_LS, etc), a driver module was designed (see 1.3) to drive register modules that support modification of their contents. 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 **??** contains an overview of each of the registers and the control signals they support.

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

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

Outputs

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

3.10 Cycle Counter (CC)

Almost every BF instruction requires multiple cycles to complete. Therefore, in addition to the instuction and state, a cycle counter is used to determine the control signals that should be sent out. This cycle counter is 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 Cycle Reset (CR) signal which resets the count in order to fetch the next instruction (which happens on cycle 0).

Inputs

- CR - Cycle Reset - Reset the count to 0;

Outputs

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

3.11 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 (**??**). The data (corresponding to the BF tape) is stored at addresses 0x0100 through 0xffff and are indexed by the data pointer (**??**).

Inputs

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

Outputs

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

3.12 Program Memory (ROM)

The actual BF instructions are stored in Read-Only-Memory (ROM) and are addressed by the IP (**??**). 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

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

Outputs

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

3.13 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 acknowlegde 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). This flag can only be reset by the CU, indicating that the transfer procedure has been completed.

Inputs

- DATA_IN - 8 bits - connected to the data bus.
- 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.
- CLR_K - Clears the K-flag - connected to the CU

Outputs

None.

3.14 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, indicating that the data is 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 buffer (either wait for user input or return 0). Shared with the output-peripheral, the K_REC signal is used to indicate that the K-flag has been received and reset, indicating that the data has been transferred and the input-device can yield control of the databus back to the system.

Inputs

- EN - Enable - Make the contents of the input buffer available on the databus.
- CLR_K - Clears the K-flag - connected to the CU.

Outputs

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

3.15 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 1.

Inputs

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

Outputs

- HLT - Halt Clock
- RS0 - Register Select, bit 0
- RS1 - Register Select, bit 1
- RS2 - Register Select, bit 2
- INC - Increment selected register
- DEC - Decrement selected register
- DPR - Reset DP
- EN_SP- Enable SP to address bus [3]
- OE_RAM - Output Enable RAM
- WE_RAM - Write Enable RAM
- EN_IN - Enable input (keyboard) to 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 `?` is used to generate a random number in the currently active cell.

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

- EN_OUT - Enable output device
- SET_V - Set V flag in FA
- SET_A - Set A flag in FA
- LD_FB - Load FB
- LD_FA - Load FA
- EN_IP - Enable IP
- EN_D - Enable D to databus
- LD_D - Load D
- LD_IP - Load IP
- CR - Cycle Reset
- CLR_K - Clear the K-flag in the IO module
- ERR - Error Signal

4 Control Sequences

4.1 Instruction Decoding

By setting the control signals as described in Section **??** appropriately, modules can work together to perform each of the BF instructions. Table **??** shows the control sequences that are executed for each BF instruction. The Control Unit implements this as a lookup-table in 3 ROM chips, where the instruction (4 bits), flags (5 bits) and cycle counter (3 bits) act as an address into this table (Figure **??**). 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. More details on the implementation can be found in Section 1.10. Below we will go through each of the control-sequences listed in Table **??**.

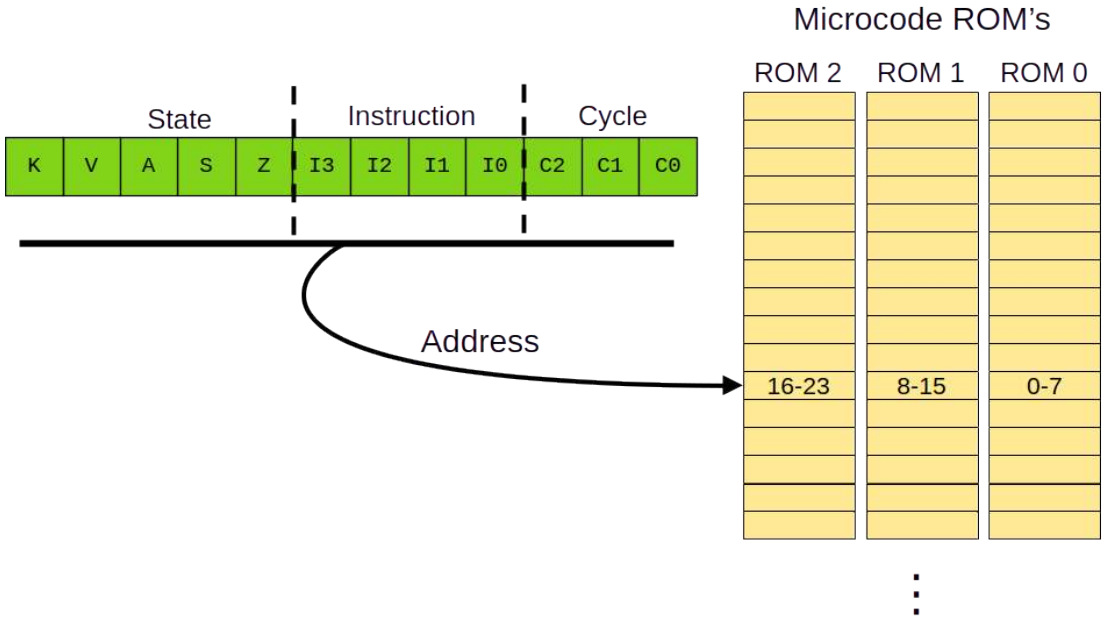


Figure 6: Decoding an instruction: the current instruction, flags and cycle-count are used as an index to the three ROM chips that output the control signals corresponding to the current state of the system.

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 (Section **??**), only one register can be driven during each clock cycle. That is, the INC and DEC signals can be applied to only one register at a time.

4.2 Cycle 0: Instruction Fetch

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

4.3 Modifying Data: + and -

PLUS-instruction The sequence of instructions necessary to execute a + command depends on the state of the system. 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 **??**, 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

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

is done by asserting the SET_V signal to FA and latching in the value using the LD(FA) signal: INC, RS0, SET_V, LD_FA. Now that the value has been incremented and the corresponding flag has been set, the IP is incremented using the register-driver. The cycle reset signal is asserted at the same time to make sure the next instruction will be loaded on cycle 0: INC, RS2, CR.

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, enabling the RAM to output its content on the databus and loading the resulting value into D: EN(DP), OE_RAM, LD(D). From hereon, the next set of control signals is identical to that described above in the case where A was not set.

MINUS-instruction 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.

Skipping: 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.

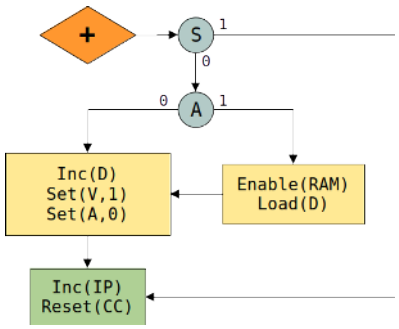


Figure 7: Block diagram for the `+` command. The diagram for the `-` command is equivalent (using `Dec` rather than `Inc`).

4.4 Moving the Pointer: < and >

RIGHT-instruction Moving the datapointer 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. 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. 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: INC, RS1, SET_A, LD_FA.

In the case that V *was* set during one of the previous instructions, we need to write the updated value in the D-register back to RAM before moving the pointer. This is achieved by enabling the value in D onto the databus, 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 set both A and V back to zero: EN_D, WE_RAM, LD_FA.

LEFT-instruction 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.

Skipping: 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.

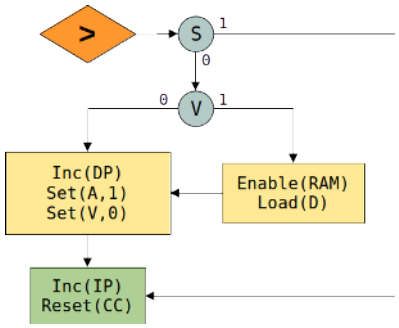


Figure 8: Block diagram for the `>` command. The diagram for the `<` command is equivalent (using `Dec` rather than `Inc`).

4.5 Conditional Jumping: [and]

These are by far the most complicated instructions that require a lot 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.

LOOP_START-instruction In the first scenario, where A is not set (the D-register is up-to-date) and the Z-flag is set (A = 0, Z = 1), we can immediately conclude that this loop should be skipped over. Hence, the LS-register is incremented and the next instruction is loaded (to be ignored until the LS-register becomes 0 again). Since LS is addressed by the register driver at address 101, this is achieved by the following control sequence: INC, RS0, RS2. In the next cycle, the IP is incremented and the cycle-counter is reset to move to the next instruction: INC, RS2, CR.

In the second scenario, the A-flag is still not set but the Z-flag for the D-register is not set either (A = 0, Z = 0), meaning that control *should* enter the loop. It takes 3 cycles to do so: increment the stack-pointer (cycle 1), write the current IP to this address on the stack (cycle 2) and move to the next instruction (cycle 3). The corresponding control sequences are therefore:

1. INC, RS0, RS1
2. WE_RAM, EN_SP, EN_IP
3. INC, RS2, CR

In the third scenario the A-flag *is* set, which means that we should first load the new data from RAM into the D-register (cycle 1) and reset the flag: OE_RAM, LD_D, LD_FA, CR. 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).

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).

1. INC, RS0, RS2
2. INC, RS2, CR

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

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

1. DEC, RS0, RS1
2. INC, RS2, CR

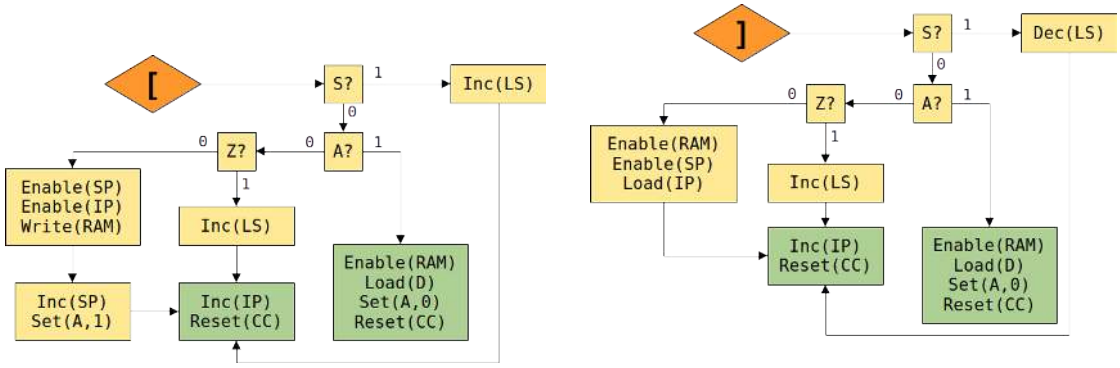
In the second scenario, there is a known nonzero value in D (A = 0, Z = 0), this means we must loop back to the IP stored on the top of the stack. This value is loaded into the IP-register by enabling the SP and RAM and setting the LD signal for the IP-register (cycle 1). On the second cycle, this new IP (pointing to a [) is incremented to re-enter the loop.

1. EN_SP, OE_RAM, LD_IP
2. INC, RS2, CR

In the thrid scenario, the contents of D are not yet synchronized with the RAM (A = 1), 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: OE_RAM, LD_D, LD_FA, CR.

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:

1. DEC, RS0, RS2
2. INC, RS2, CR



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

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

Figure 9

4.6 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. The handshake protocol is therefore defined by the following steps:

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

Cycle 0: As mentioned briefly before, the OUT instruction is the only instruction that has a slighty 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: LD_FB, LD_I, (EN_D or EN_RAM).

Skipping: 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.

4.7 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.

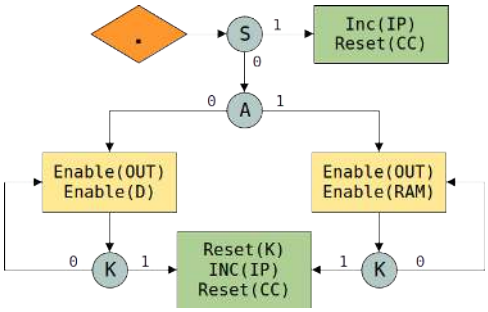
1. The CU asserts the EN_IN signal to let the input-device know that it can claim ownership of the bus.
2. The input-devices receives the EN_IN signal and puts a value onto the bus (see below for the difference between immediate and buffered input-mode). When the data is ready, it sets the K-flag.
3. 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:
   - K = 0: CR
   - K = 1: LD_D, SET_V, LD_FA
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:

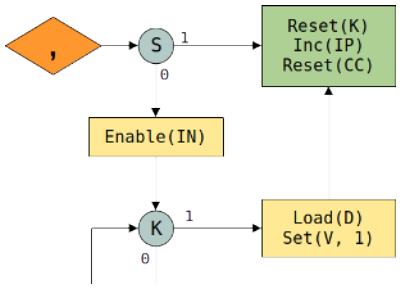
1. Wait for the buffer to contain data. Only then set the K-flag (buffered mode).
2. Assert zero's to the bus and set the K-flag immediately (immediate mode).

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

Skipping: 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.



(a) Block diagram for the output command.



(b) Block diagram for the input command.

Figure 10

4.8 Non-BF instructions

Several non-BF instructions have been implemented for debugging purposes and to initialization at startup. Furthermore, the common extension *Random Brainf*ck* [**?**] 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.

4.8.1 `NOP`

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

4.8.2 `WAIT_EXT`

The WAIT_EXT instruction (*wait for external device*) should be the first instruction in the program binary and simply loops on the K-flag, waiting for an external peripheral to set it to 1. When this happens, the CU immediately resets the flag using the CLR_K signal. Both the BF system and the IO module will now be ready for operation:

- K = 0: CR
- K = 1: CLR_K, K_REC, INC, RS2, CR

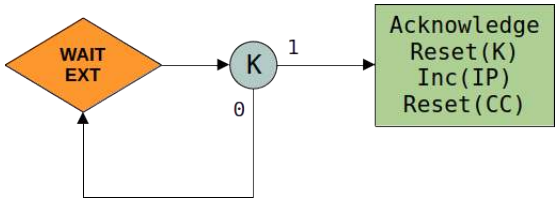


Figure 11: Block diagram for the WAIT_EXT instruction.

4.8.3 `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 (after the initial WAIT_EXT handshake) in order to initialize the RAM (or at least part of it) to 0. While this can be handled using canonical BF commands, initializing one cell at a time using a sequence of `[-]` commands, it is much faster to write 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 be able 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.

1. EN_D, WE_RAM, INC, RS0, RS2
2. LD_FB, INC, RS1
3. - S = 0: INC, RS2, CR
   - S = 1: CR

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

4.8.4 `HOME`

In order for the datapointer to return to its starting position after initialization (and before the main program runs), the `HOME` instruction is provided. The only thing it does is send the reset signal to the datapointer using the DPR signal.

4.8.5 `HLT`

The `HLT` instruction halts the clock and (temporarily) stops the program by asserting the HLT signal. The assembler will place a HLT instruction after initialization (`INIT`) and after the final instruction of the main program. This 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 interpret an exclamation mark (`!`) as a `HLT` in the BF-code to set breakpoints for debugging.

When the system is resumed and cycle 2 of the HLT instruction is reached, the IP is incremented as usual in the final cycle of any instruction: INC, RS2, CR.

4.8.6 `ERR`

The `ERR` instruction enables the ERR signal and halts the clock: ERR, HLT. It therefore acts similarly to a regular `HLT`, but indicates to the end user that the system was halted due to an error (and it is therefore probably not wise to resume the system anyway). Furthermore, there is no second cycle defined to increment IP and go beyond this point in the program.

Every undefined state in the microcode table implicitly contains the `ERR` sequence. If for some reason a state occurs that maps to the `ERR` command, the clock will be halted and some indicator on the Control Unit should light up to let its users know that something has gone terribly wrong and the computer has reached an unreachable state.

4.9 Microcode table

Table **??** shows each of the control sequences described in previous sections sections. Please note that in order to simplify notation, the control signals RS0, RS1 and RS2 have not been used to indicate register selection. Instead, the module to which the instruction (INC/DEC) is applied is provided in brackets. For example, incrementing the SP register would require control signals INC, RS0, RS1 which is denoted in Table **??** as INC(SP).

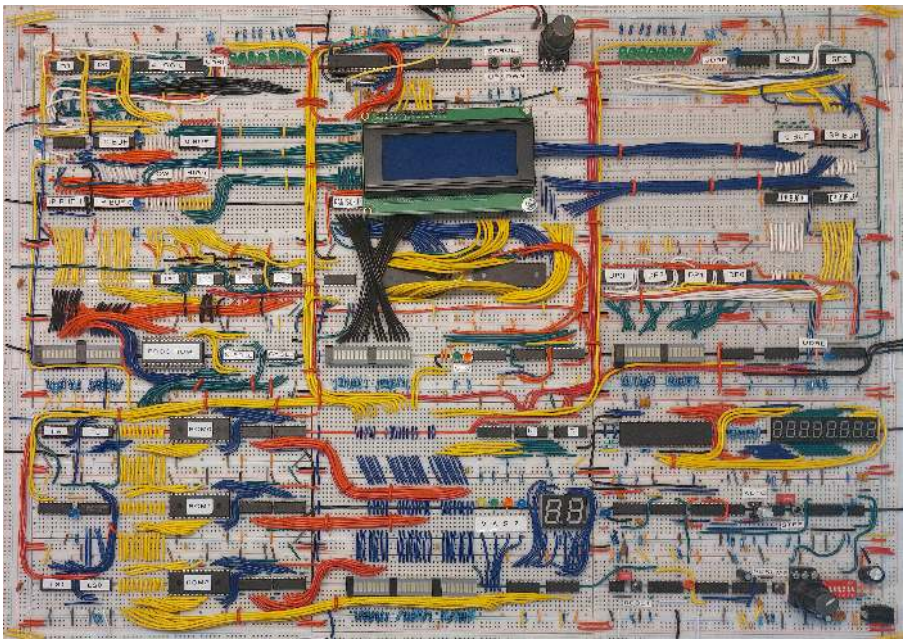
Instr	K	V	A	S	Z	Cycle	Control Signals				
Any except "."						0	LD(FB)	LD(I)			
+			0	0		1	INC(D)	SETV(CU)	LD(FA)		
			0	0		2	INC(IP)	CR(CU)			
			1	0		1	LD(D)	OE(RAM)			
			1	0		2	INC(D)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CR(CU)			
				1		1	INC(IP)	CR(CU)			
-			0	0		1	DEC(D)	SETV(CU)	LD(FA)		
			0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

		1		1	INC(IP)	CR(CU)		
RAND		0		1	EN(KB)	EN(SCR)		
	0	0		2	CR(CU)			
	1	0		2	LD(D)	SETV(CU)	LD(FA)	
	1	0		3	CLR(K)	INC(IP)	CR(CU)	
		1		1	INC(IP)	CR(CU)		
NOP				1	INC(IP)	CR(CU)		
HLT		0		1	HLT(CLC)			
		1		1	INC(IP)	CR(CU)		
INIT			1	1	EN(D)	WE(RAM)	INC(LS)	
			1	2	LD(FB)	INC(DP)		
		0	1	3	INC(IP)	CR(CU)		
		1	1	3	CR(CU)			
HOME				1	RESET(DP)			

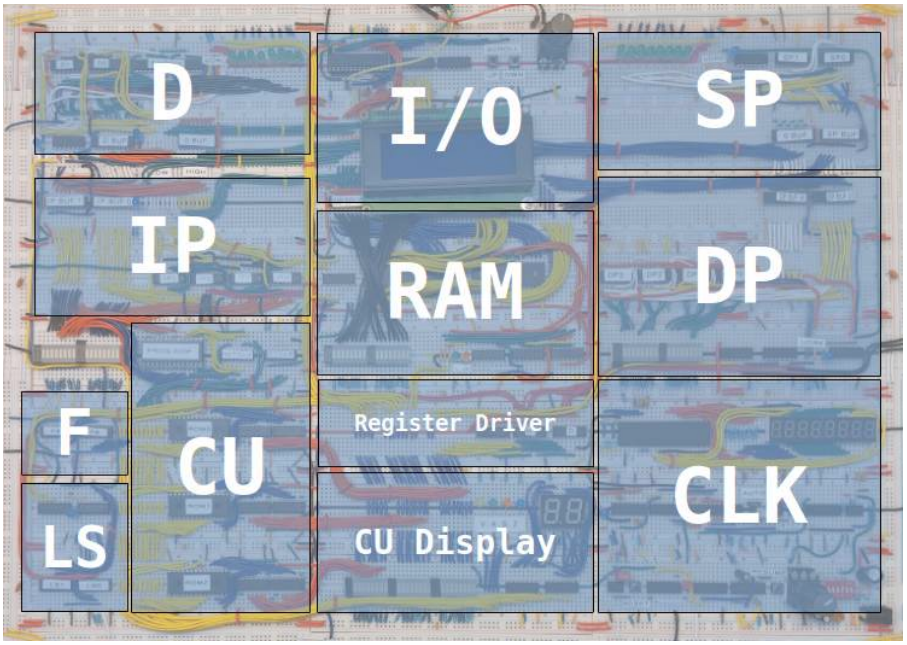
Table 2: Control signals for each of the BF instructions in different scenario's, depending on the state flags. Note that in order to distinguish between the two input modes, the regular comma (`,`) and the apostrophe (`'`) are used for buffered and immediate inputs respectively. See also Section **??**.

5 Implementation

This section will discuss the implementation of each module and the way they integrate together to make the computer. Figure 1a shows the computer as it was in February 2025. The overlays shown in Figure 1b show where each of the modules described in previous sections is located on the computer.



(a) Prototype of February 2025



(b) Location of the modules on the prototype

Figure 12

5.1 Clock Module

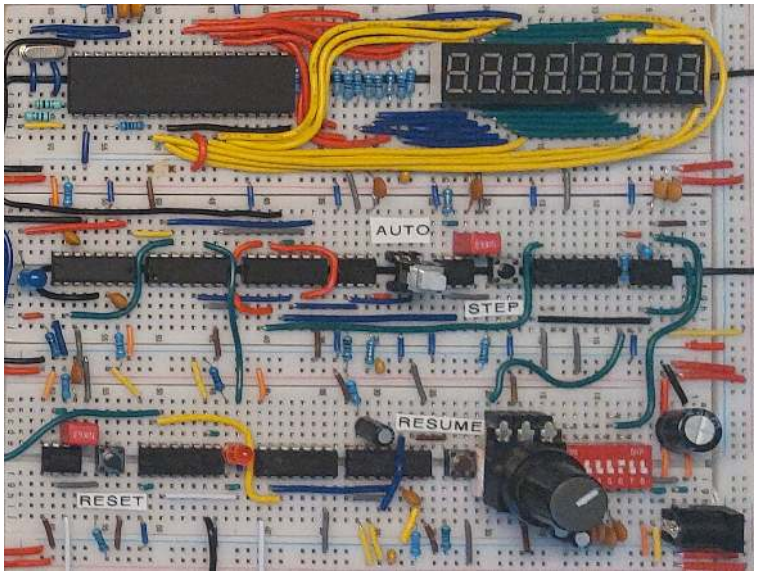


Figure 13: Close up of the Master Clock and Reset/Resume Modules.

5.1.1 Overview

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 [**?**]. 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. 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 3 shows the timing diagram for the different signals discussed above.

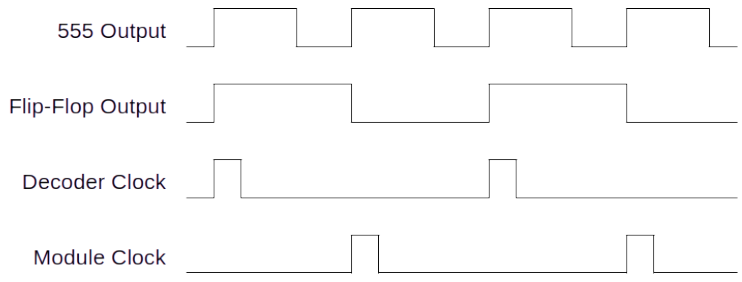


Figure 14: Timing diagram for the clock signals.

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

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 finegrained selection of the frequency within this range. This potentiometer is wired in series with a 1K resistor to enure stability when the potentiometer resistance drops toward zero. Table 1 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 1). 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.

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[**?**], 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 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.

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 3), 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. By selecting a 680K resistor and a $4.7\mu F$ capacitor, a cooldown period of around 1 second is achieved. Figure 4 shows a timing diagram to illustrate this process in more detail.

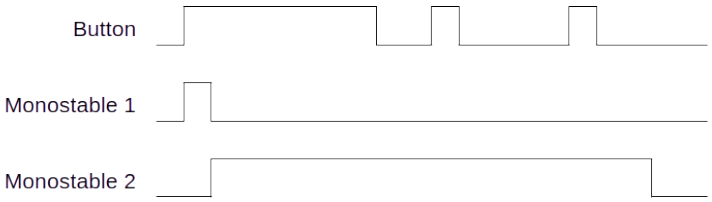
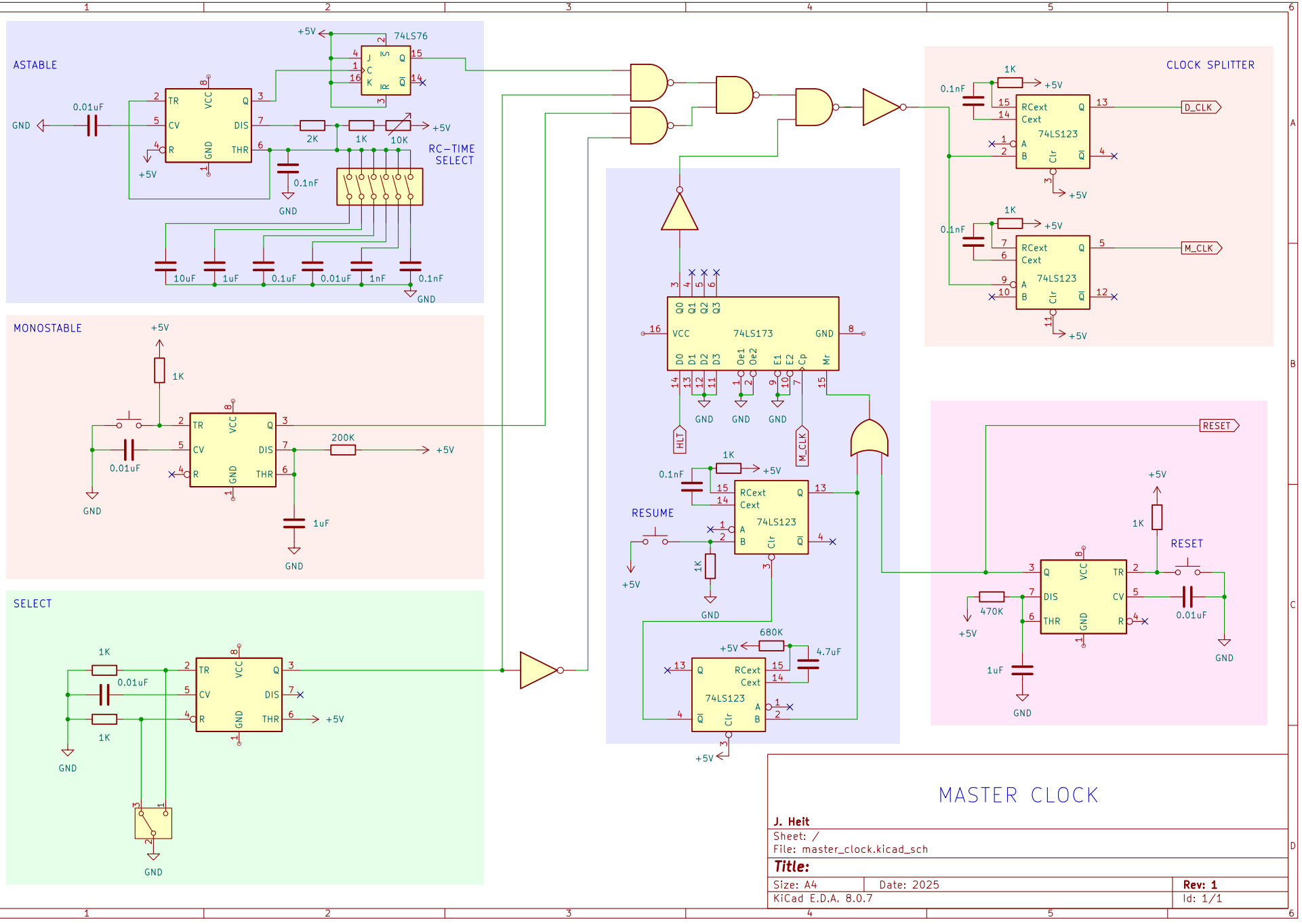
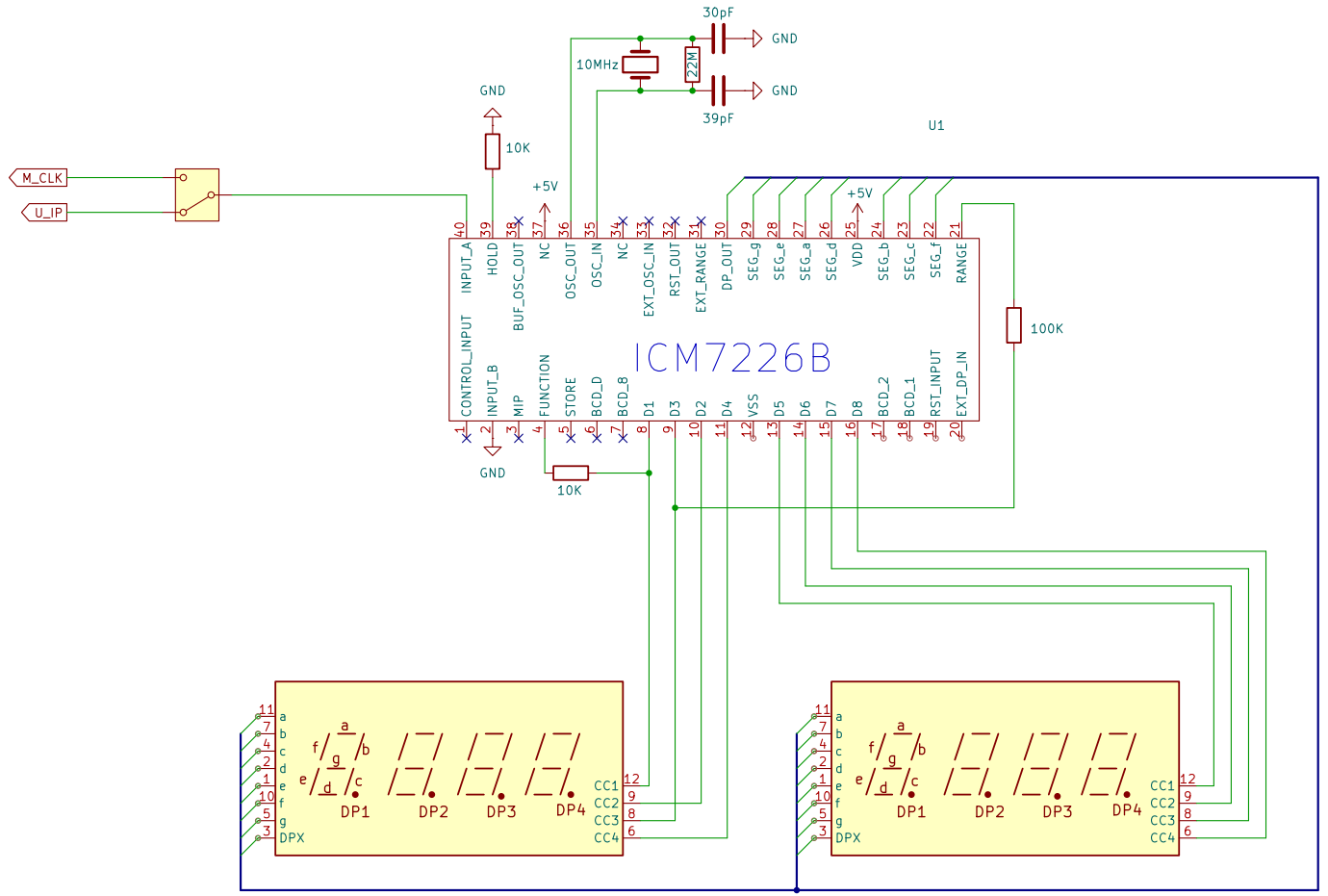


Figure 15: Timing diagram for the resume debouncer. The output of Monostable 1 is connected to the reset of the register that stores the HLT signal.

5.2.1 Schematic

Full schematics for the clock module, reset/resume circuitry and the frequency display section are provided on the next pages.


ASTABLE
74LS76
0.01uF
GND
+5V
2K
1K
10K
RC-TIME SELECT
0.1nF
10uF
1uF
0.01uF
1nF
MONOSTABLE
200K
SELECT
CLOCK SPLITTER
D_CLK
M_CLK
74LS123
74LS173
HLT
RESUME
680K
4.7uF
RESET
470K
MASTER CLOCK
J. Heit
Sheet: /
File: master_clock.kicad_sch
Title:
Size: A4
Date: 2025
Rev: 1
KiCad E.D.A. 8.0.7
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

5.3 Register Driver

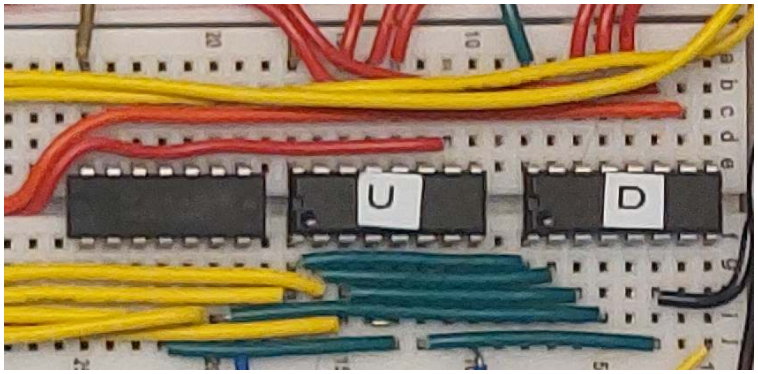


Figure 16: Close up of the Register Driver Module.

5.3.1 Overview

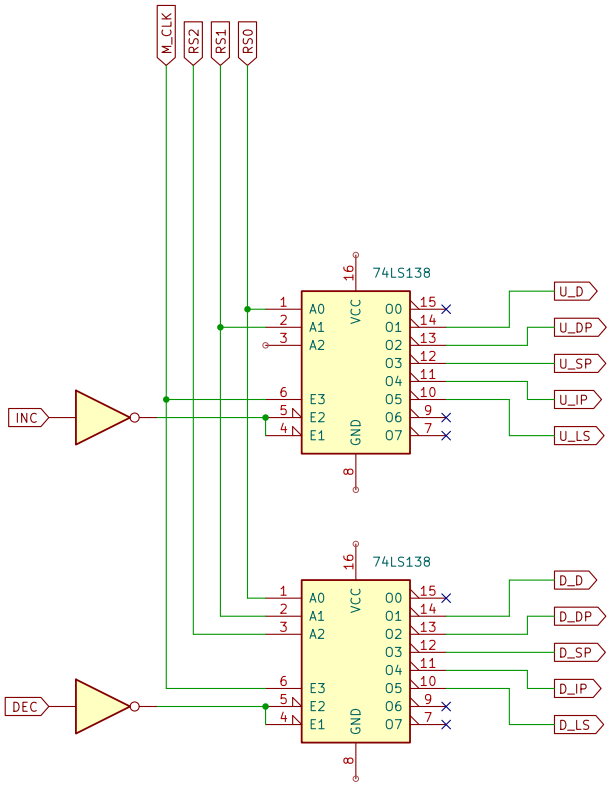
The register driver is responsible for driving the U- and D-inputs of the 74LS193 counting registers that are used to implement the D, DP, SP, IP and LS register modules. To increment the '193, its D-input needs to be held high whileproviding a low pulse to the U-input. As explained in section **??**, 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.

The driver module uses a pair of 74LS138 decoders: 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-selects 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.

5.3.2 Schematic

A full schematic is provided on the next page.



REGISTER DRIVER

Sheet: /		
File: register_driver.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. 8.0.7		Id: 1/1

5.4 DP Register Module

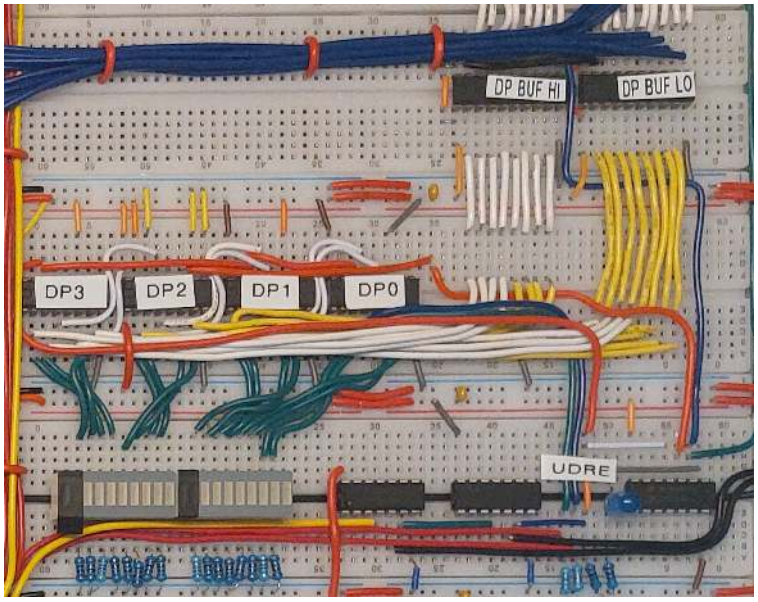


Figure 17: Close up of the Data Pointer Register Module.

5.4.1 Overview

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 **??**) 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.

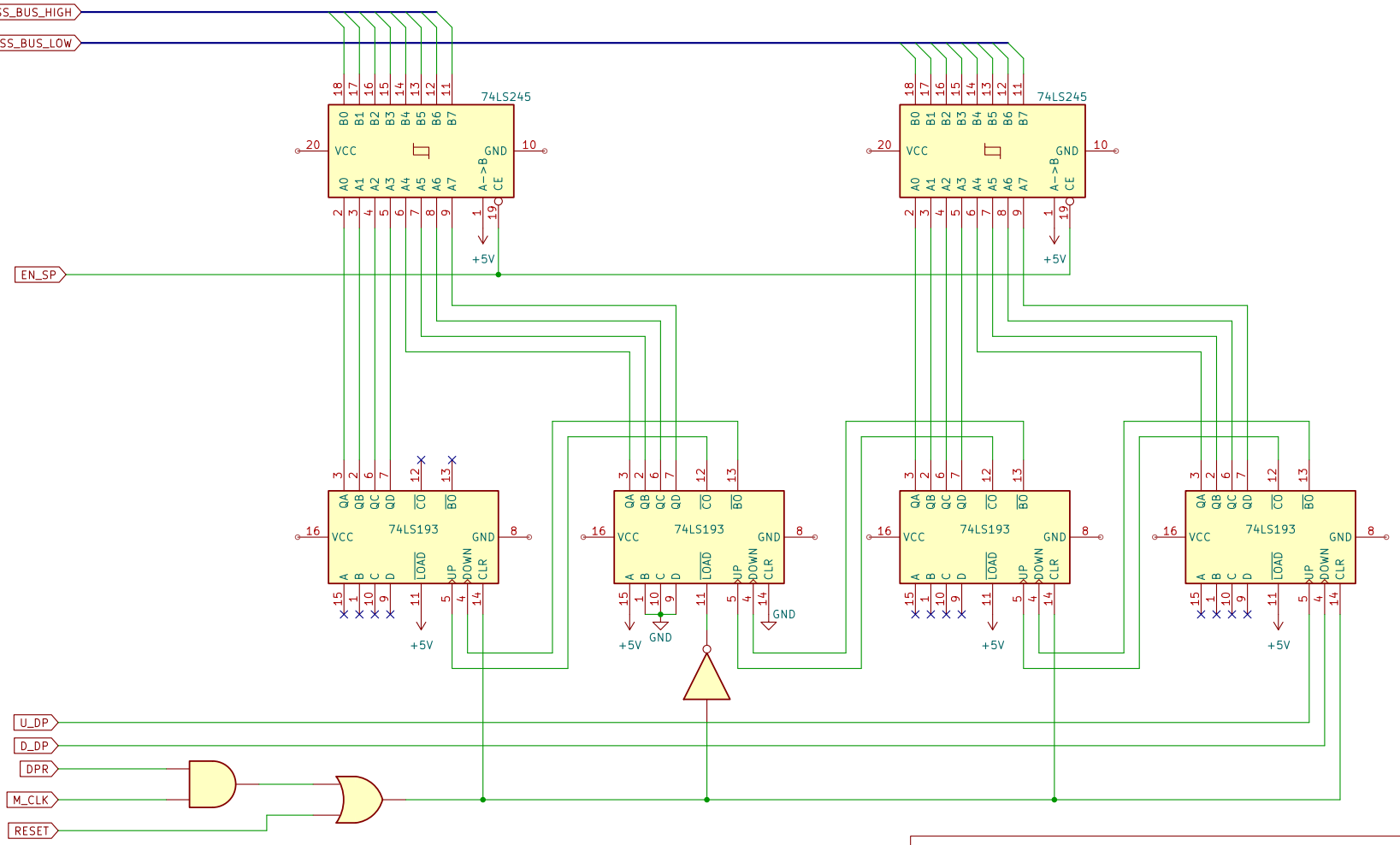
Reset Vector. Since the data section in RAM starts at 0x0100 (0x0000 through 0x00ff are reserved for the stack), this is the value that the register should start at right after booting up the system (all other registers start out with an initial value of zero). To achieve this, the global reset line 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 Sections **??** and **??**). 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 mututally 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.

5.4.2 Schematic

A full schematic is provided on the next page.



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

5.5 D Register Module

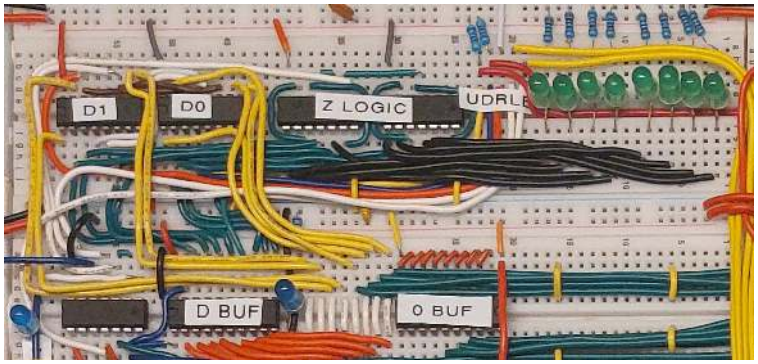


Figure 18: Close up of the Data Register Module.

5.5.1 Overview

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 1.3 at address 1 (0b001, see Table **??**). 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.

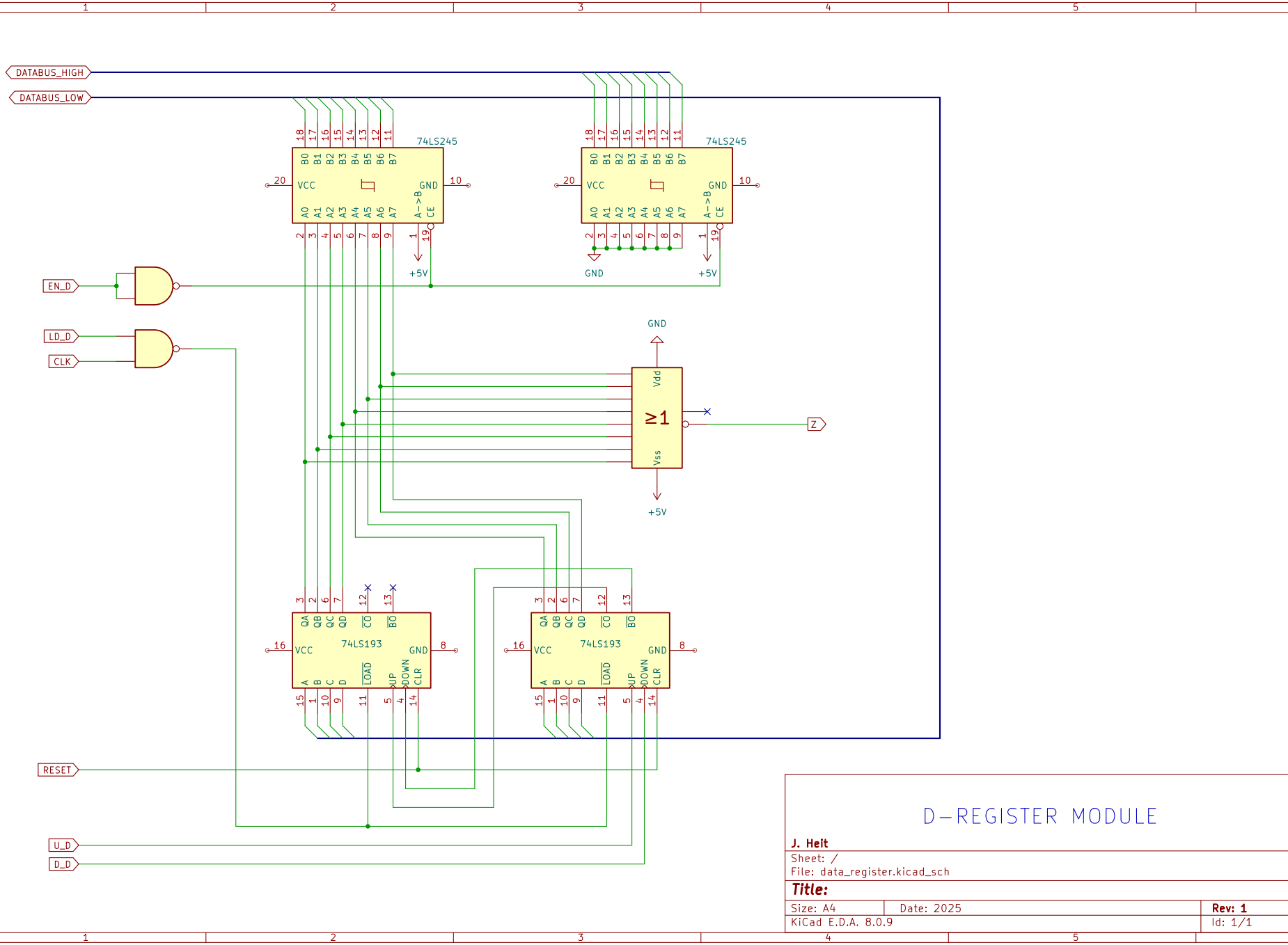
Enabling Output. The outputs of the '193s are buffered by a single 8-bit tristate buffer (74LS245) before being connected to the databus. Because the databus is 16-bit wide (necessary to store IP-values on the stack), the high-byte is set explicitly to 0 when D is enabled by a second buffer that always 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.

5.5.2 Schematic

A full schematic is provided on the next page.



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

5.6 IP Register Module

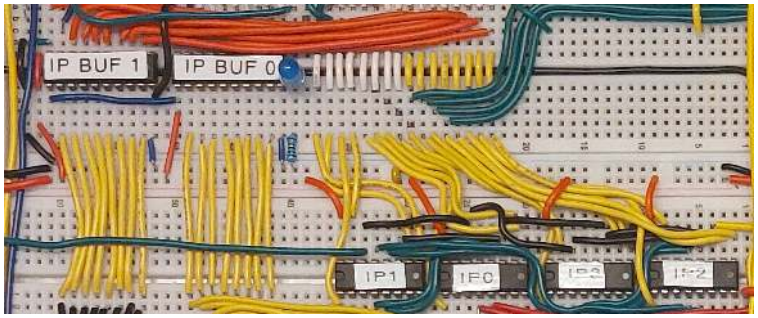


Figure 19: Close up of the Instruction Register Module.

5.6.1 Overview

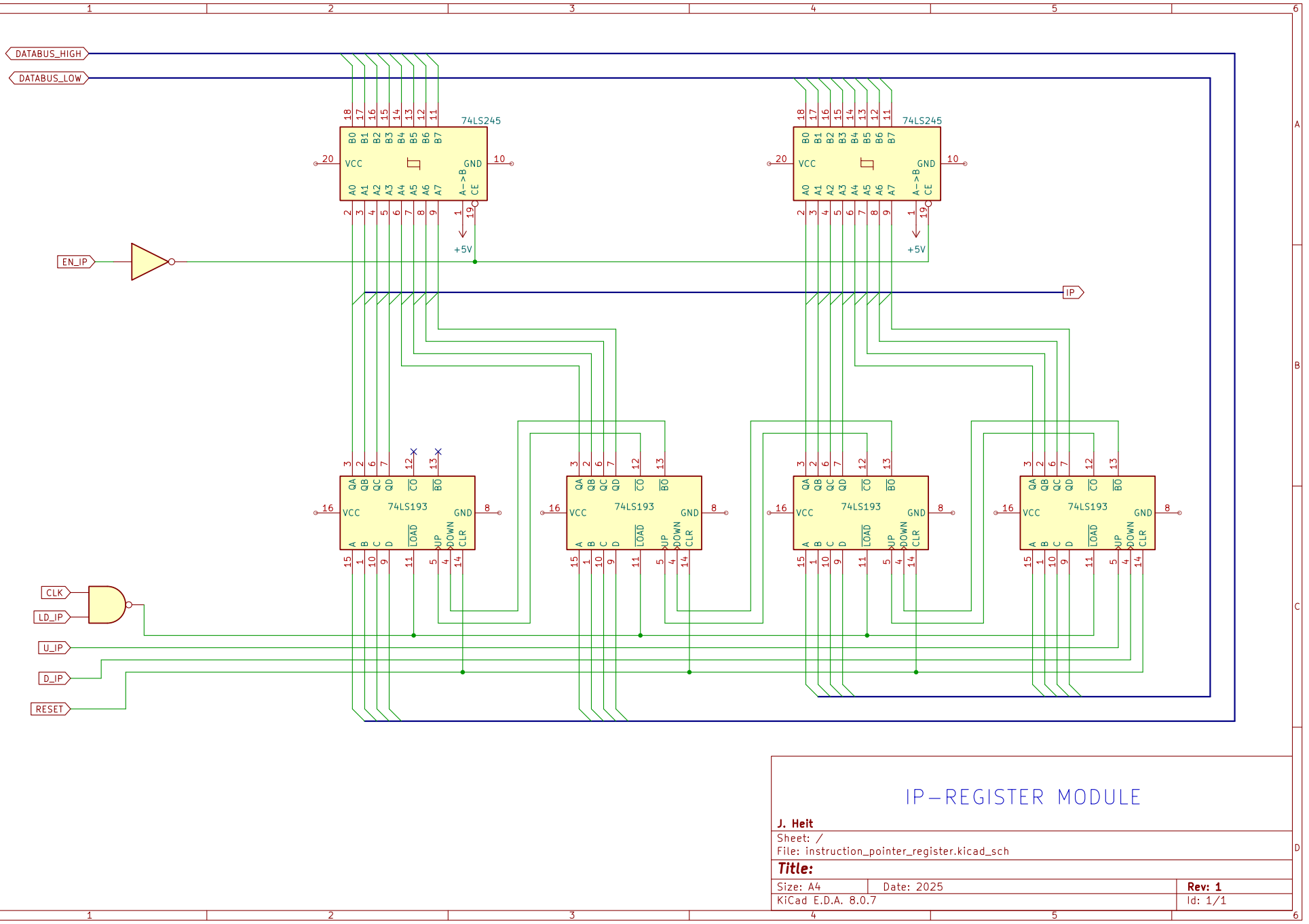
The instruction pointer register holds a 16-bit value representing the address of an instruction in the program-ROM (implemented using an EEPROM chip (see **??**). The size of the available address space in program-memory is 2^{14} instructions, so the two uppermost bits (bits 14 and 15) of the IP are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

Forbidden Decrement. The IP, like the D and DP registers, is driven by the Register Driver at address 4 (0b100), 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.

5.6.2 Schematic

A full schematic is provided on the next page.



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

5.7 SP Register Module

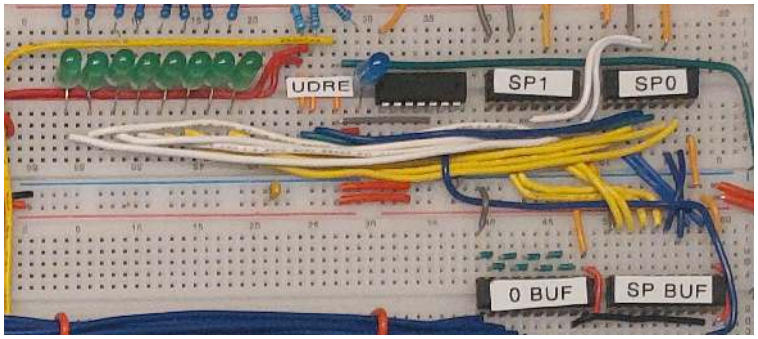


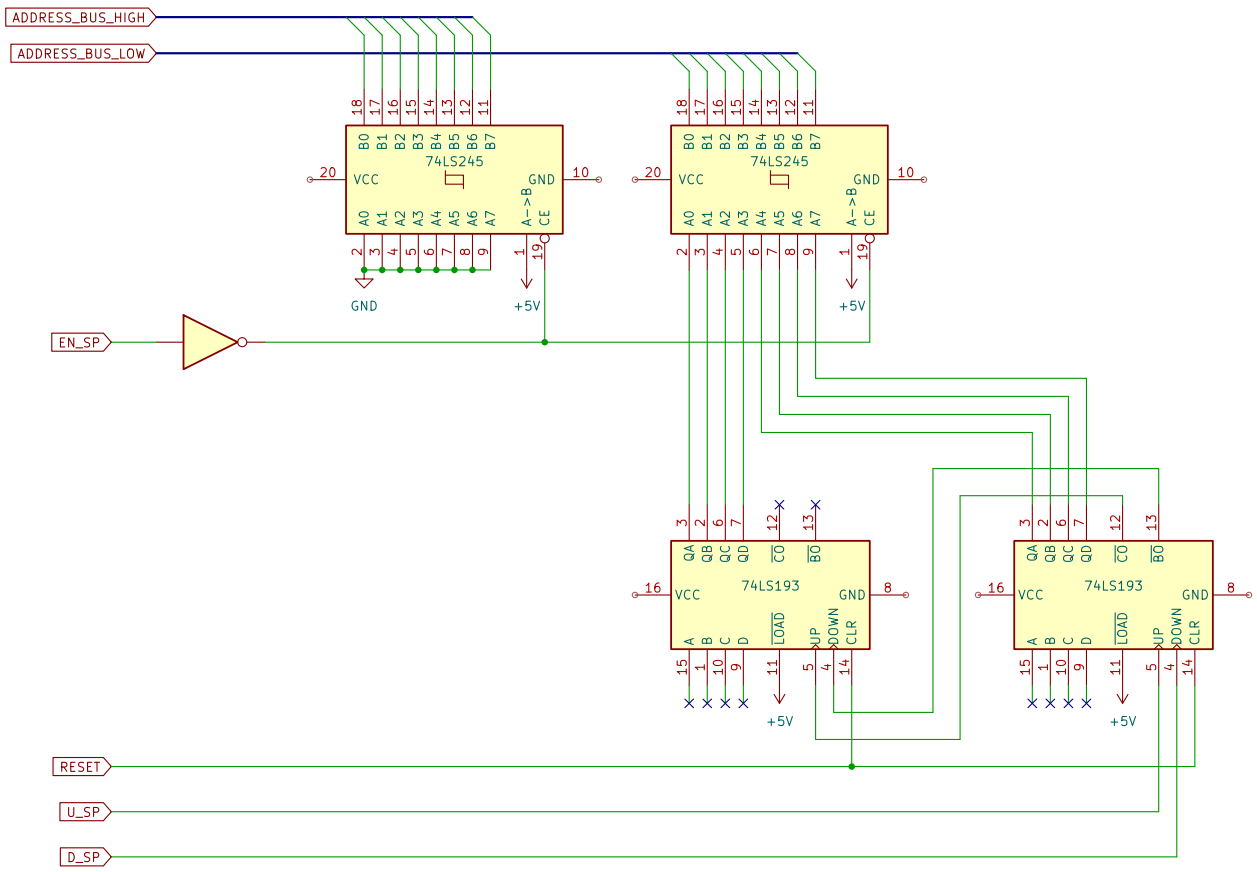
Figure 20: Close up of the Stack Pointer Register Module.

5.7.1 Overview

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 (0b011). 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 (**??**), 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 datasection.

5.7.2 Schematic

A full schematic is provided on the next page.



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

5.8 LS Register Module

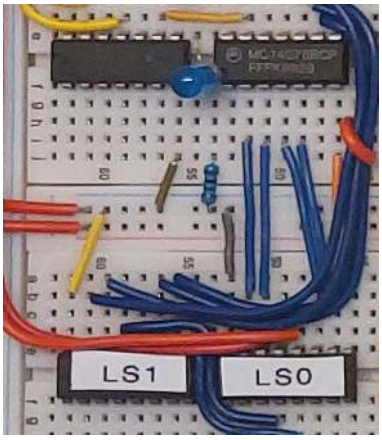


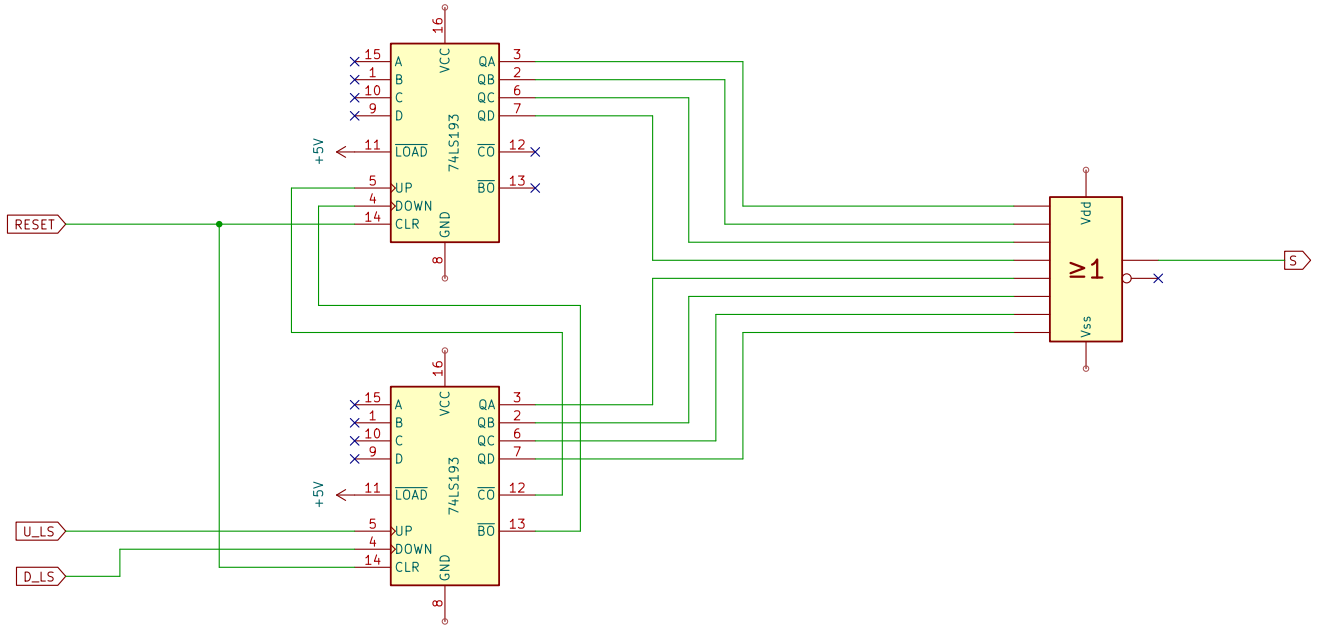
Figure 21: Close up of the Loop Skip Register Module.

5.8.1 Overview

The Loop Skip Register (LS) is used to produce the loop-skip-flag (S, see **??**). It is implemented using two 74LS193 binary counters and is connected to register driver at address 5 (0b101). 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.

5.8.2 Schematic

A full schematic is provided on the next page.



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

5.9 RAM Module

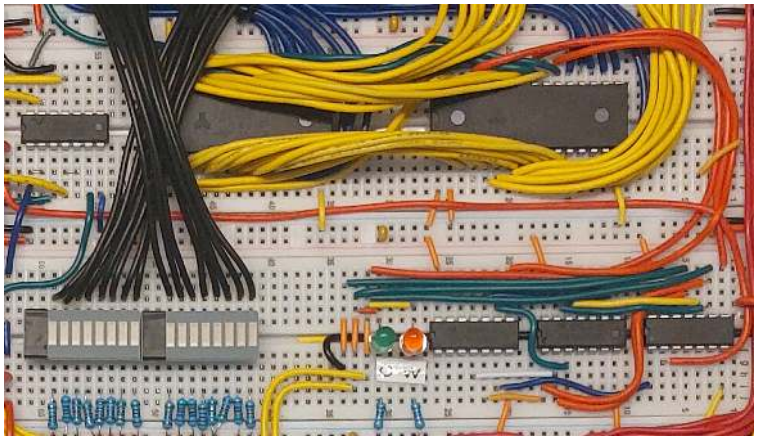


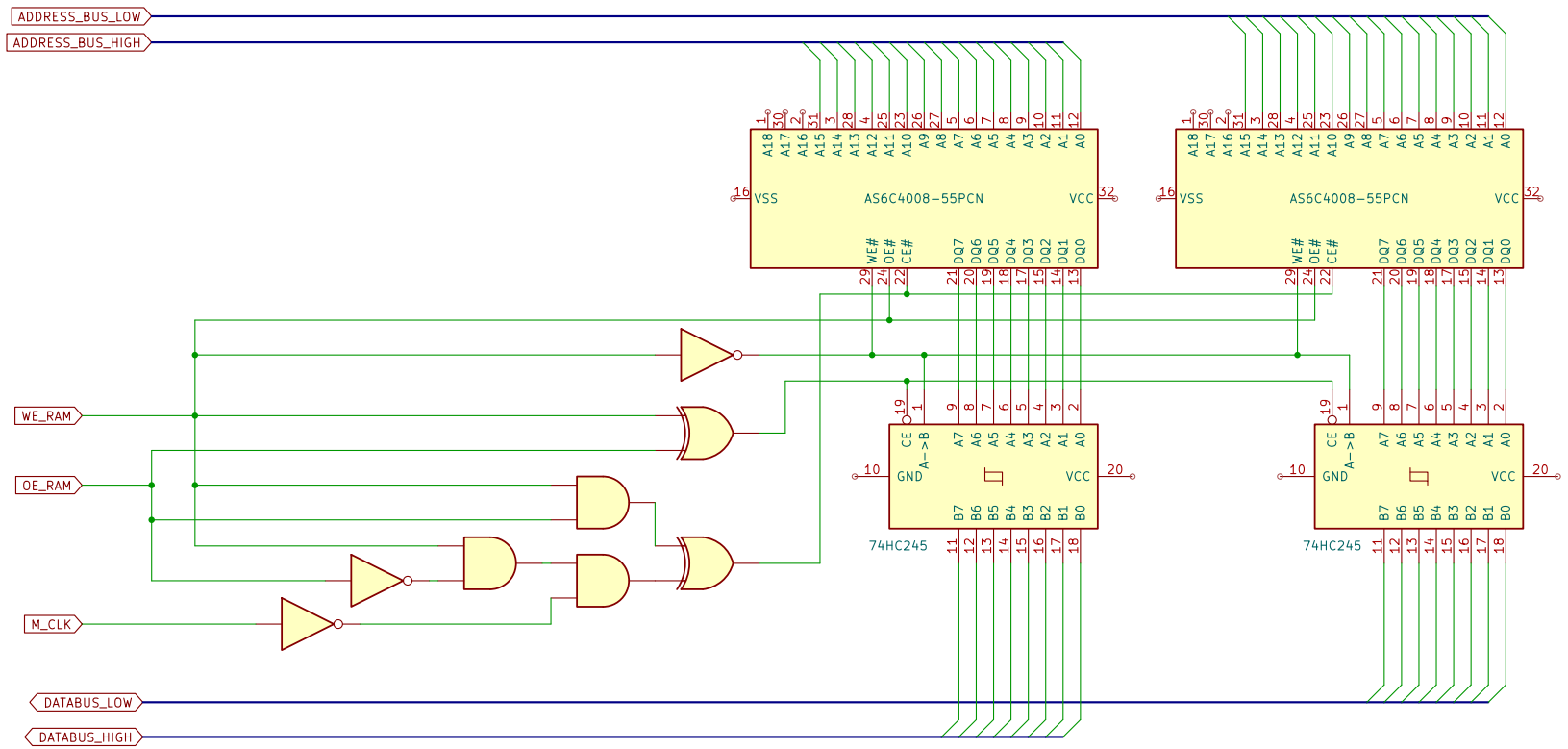
Figure 22: Close up of the RAM Module.

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. The second chip is therefor only used to store the high-byte of the IP's stored on the stack (at most 256 values). The remainder of the capacity of this chip is not used at all, since the datavalues are only 8 bits in size.

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 incorporated in the schematics below. The LED's are not shown in the schematic, but can be connected directly to the datalines of the RAM in this configuration.

5.9.1 Schematic

A full schematic is provided on the next page.



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

5.10 Control Unit

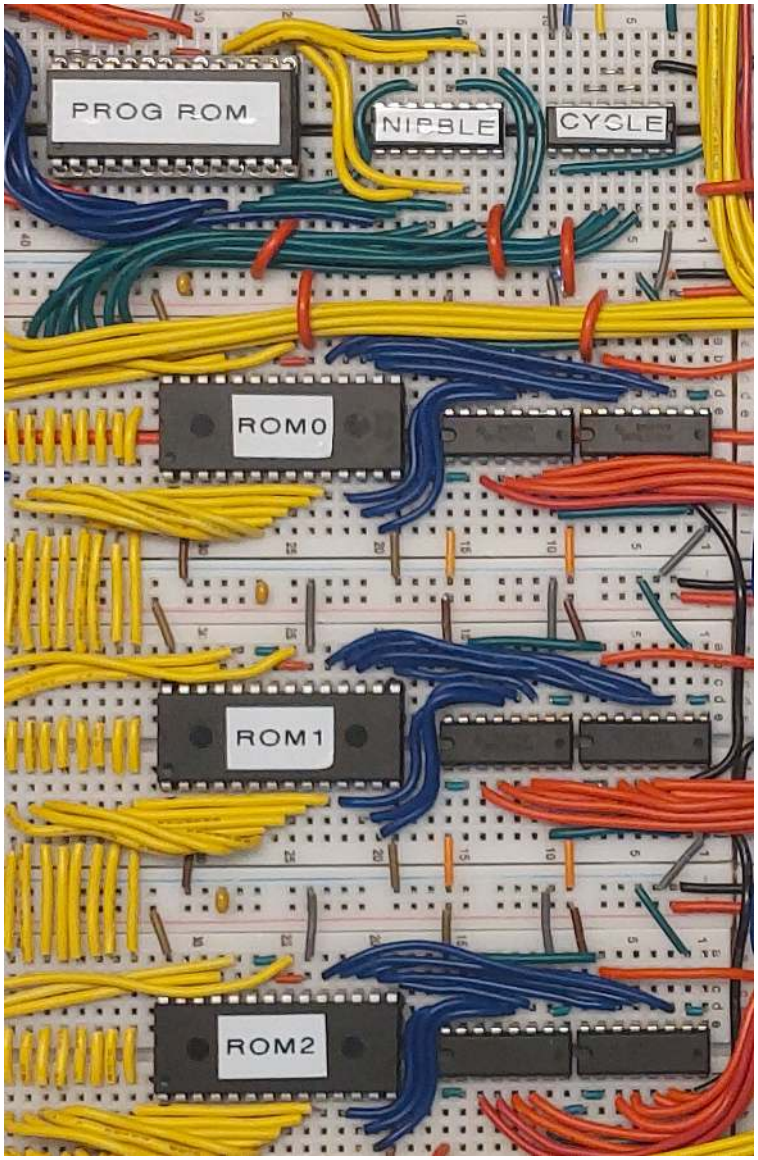


Figure 23: Close up of the Control Unit.

5.10.1 Overview

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 **??**), a utility developed to make the microcode programming more maintainable. Mugen generates the images from a specification file. The relevant part of the specification for the BFCPU is shown in the listing below and is a direct representation of the microcode shown in Table **??**.

```
[microcode] {

  NOP:0:xxxxx           -> LD_FBI
  PLUS:0:xxxxx          -> LD_FBI
  MINUS:0:xxxxx         -> LD_FBI
  LEFT:0:xxxxx          -> LD_FBI
  RIGHT:0:xxxxx         -> LD_FBI
  IN:0:xxxxx            -> LD_FBI
  OUT:0:xx0xx           -> LD_FBI, EN_D     # when OUT is spinning, EN_D must be kept high
  OUT:0:xx1xx           -> LD_FBI, OE_RAM   # OE_RAM in this case, for the same reason
  LOOP_START:0:xxxxx    -> LD_FBI
  LOOP_END:0:xxxxx      -> LD_FBI
  RAND:0:xxxxx          -> LD_FBI
  WAIT_EXT:0:xxxxx      -> LD_FBI
  INIT:0:xxxxx          -> LD_FBI
  HOME:0:xxxxx          -> LD_FBI
  HALT:0:xxxxx          -> LD_FBI

  PLUS:1:xx00x          -> INC, RS0, SET_V, LD_FA
  PLUS:2:xx00x          -> INC, RS2, CR
  PLUS:1:xx10x          -> LD_D, OE_RAM
  PLUS:2:xx10x          -> INC, RS0, SET_V, LD_FA
  PLUS:3:xx10x          -> INC, RS2, CR
  PLUS:1:xxx1x          -> INC, RS2, CR

  MINUS:1:xx00x         -> DEC, RS0, SET_V, LD_FA
  MINUS:2:xx00x         -> INC, RS2, CR
  MINUS:1:xx10x         -> LD_D, OE_RAM
  MINUS:2:xx10x         -> DEC, RS0, SET_V, LD_FA
  MINUS:3:xx10x         -> INC, RS2, CR
  MINUS:1:xxx1x         -> INC, RS2, CR

  LEFT:1:x0x0x          -> DEC, RS1, SET_A, LD_FA
  LEFT:2:x0x0x          -> INC, RS2, CR
  LEFT:1:x1x0x          -> EN_D, WE_RAM
  LEFT:2:x1x0x          -> DEC, RS1, SET_A, LD_FA
  LEFT:3:x1x0x          -> INC, RS2, CR
  LEFT:1:xxx1x          -> INC, RS2, CR

  RIGHT:1:x0x0x         -> INC, RS1, SET_A, LD_FA
  RIGHT:2:x0x0x         -> INC, RS2, CR
  RIGHT:1:x1x0x         -> EN_D, WE_RAM
  RIGHT:2:x1x0x         -> INC, RS1, SET_A, LD_FA
  RIGHT:3:x1x0x         -> INC, RS2, CR
  RIGHT:1:xxx1x         -> INC, RS2, CR

  LOOP_START:1:xx001    -> INC, RS0, RS2
  LOOP_START:2:xx001    -> INC, RS2, CR
  LOOP_START:1:xx000    -> INC, RS0, RS1
  LOOP_START:2:xx000    -> WE_RAM, EN_SP, EN_IP
  LOOP_START:3:xx000    -> INC, RS2, CR
```

```
52    LOOP_START:1:xx10x     -> OE_RAM, LD_D, LD_FA, CR
53    LOOP_START:1:xxx1x     -> INC, RS0, RS2
54    LOOP_START:2:xxx1x     -> INC, RS2, CR
55
56    LOOP_END:1:xx001       -> DEC, RS0, RS1
57    LOOP_END:2:xx001       -> INC, RS2, CR
58    LOOP_END:1:xx000       -> EN_SP, OE_RAM, LD_IP
59    LOOP_END:2:xx000       -> INC, RS2, CR
60    LOOP_END:1:xx10x       -> OE_RAM, LD_D, LD_FA, CR
61    LOOP_END:1:xxx1x       -> DEC, RS0, RS2
62    LOOP_END:2:xxx1x       -> INC, RS2, CR
63
64    OUT:1:xxx1x            -> INC, RS2, CR
65    OUT:1:xx00x            -> EN_OUT, EN_D
66    OUT:1:xx10x            -> EN_OUT, OE_RAM
67    OUT:2:0x00x            -> EN_D, CR
68    OUT:2:0x10x            -> OE_RAM, CR
69    OUT:2:1xx0x            -> CLR_K, INC, RS2, CR
70
71    IN:1:xxx1x             -> INC, RS2, CR
72    IN:1:xxx0x             -> EN_IN
73    IN:2:0xx0x             -> CR
74    IN:2:1xx0x             -> LD_D, SET_V, LD_FA
75    IN:3:1xx0x             -> CLR_K, INC, RS2, CR
76
77    RAND:1:xxx1x           -> INC, RS2, CR
78    RAND:1:0xx0x           -> EN_IN, EN_OUT
79    RAND:2:0xx0x           -> CR
80    RAND:1:1xx0x           -> LD_D, SET_V, LD_FA
81    RAND:2:1xx0x           -> CLR_K, INC, RS2, CR
82
83    WAIT_EXT:1:0xxxx       -> CR
84    WAIT_EXT:1:1xxxx       -> CLR_K, INC, RS2, CR
85
86    INIT:1:xxxx1           -> EN_D, WE_RAM, INC, RS0, RS2
87    INIT:2:xxxx1           -> LD_FBI, INC, RS1
88    INIT:3:xxx01           -> INC, RS2, CR
89    INIT:3:xxx11           -> CR
90
91    NOP:1:xxxxx            -> INC, RS2, CR
92    HALT:1:xxxxx           -> HLT
93    HALT:2:xxxxx           -> INC, RS2, CR
94
95    HOME:1:xxxxx           -> DPR, INC, RS2, CR
96
97    catch                  -> ERR, HLT
98  }
```

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.

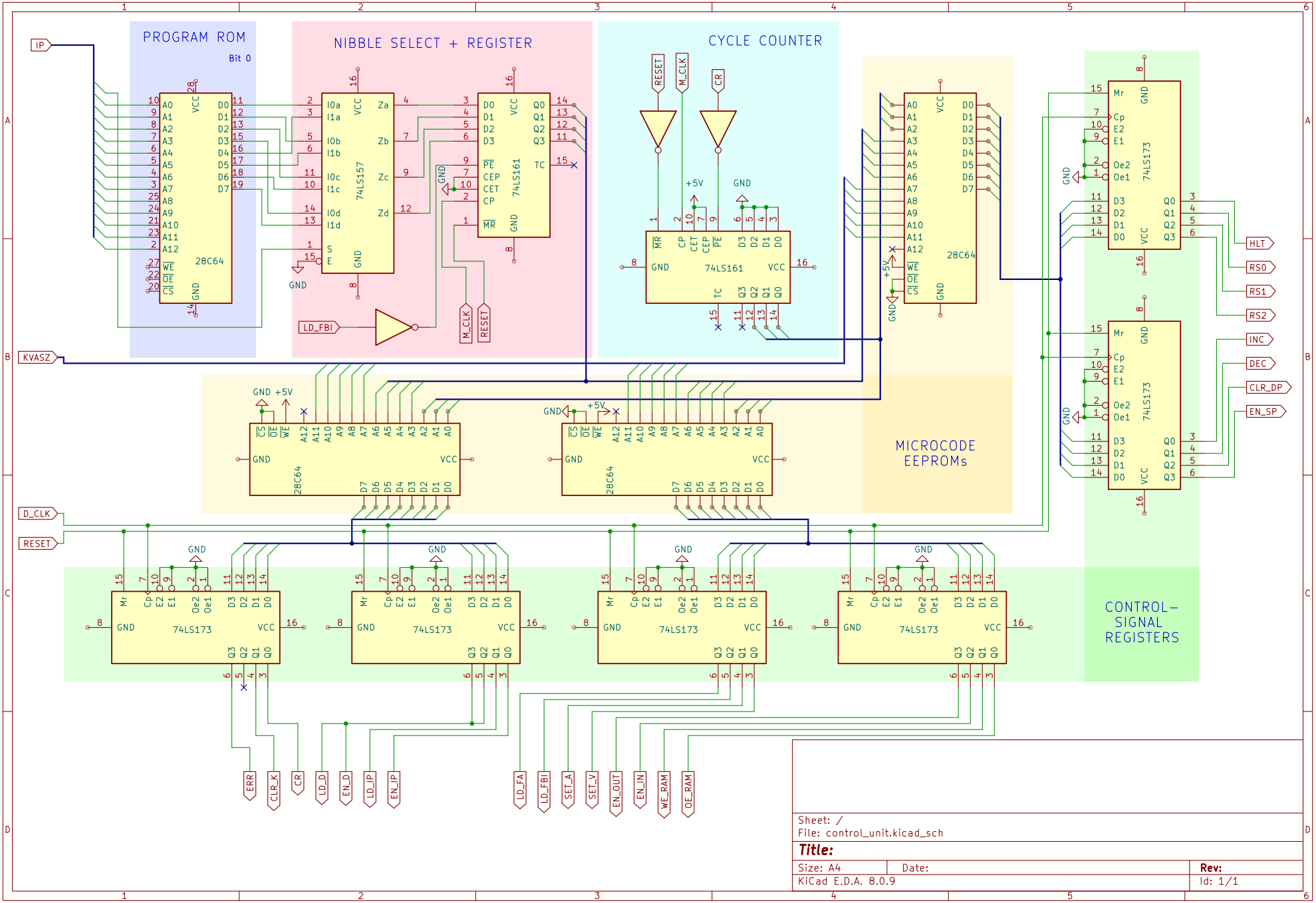
Instruction Register. The seleted nibble is loaded into the instruction register (I) at the same time as the V, A, S and Z flags are loaded into the FB register. For this reason both the FB and I registers can operate on the same control-signal: LD_FBI. The I register is implemented using the 74LS161, 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).

5.10.2 Schematic

A full schematic is provided on the next page.



Sheet: /
File: control_unit.kicad_sch

Title:

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

5.11 IO Module

5.11.1 Overview

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

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

Buttons and Menu Two buttons are provided to interact with this system. They are mainly used to scroll the screen but can also be used to access and navigate a menu (Figure 13). This menu let's the user do the following:

1. Clear the screen and keyboard buffer.
2. 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.
3. 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.
4. 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.
5. 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).
6. 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.
7. 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.

5.11.2 Handshake Protocol

Output The `M_CLK` signal is connected to an interrupt pin of the MCU. On every interrupt triggered by the clock, the MCU will check the status of the pin connected to the `EN_OUT` signal. When it is high, all of the pins connected to the databus are read to reconstruct the byte present on the bus. This value is then displayed on the LCD. Listing **??** shows the code for the interrupt routine running on the MCU.

Figure 24: Part of the menu that is accessible by pressing both scroll-buttons simultaneously.

Input Handling input from the keyboard is slightly more involved. When interrupted by the clock, if the `EN_IN` signal is found to be high, the MCU goes into a 3-clock-cycle routine. During the first cycle (when the `EN_IN` signal went high), the IO pins are set to output-mode, data is fetched from the keyboard and put on the bus. On the second cycle, it does nothing at all; this is when the computer reads the value into the D register. Then on the 3rd cycle, when the computer should have completed reading the value into its register, the pins can be reset to input-mode (high Z) and the MCU will wait for its next instruction. The flow of the state-machine that implements this algorithm is shown in Figure 14.

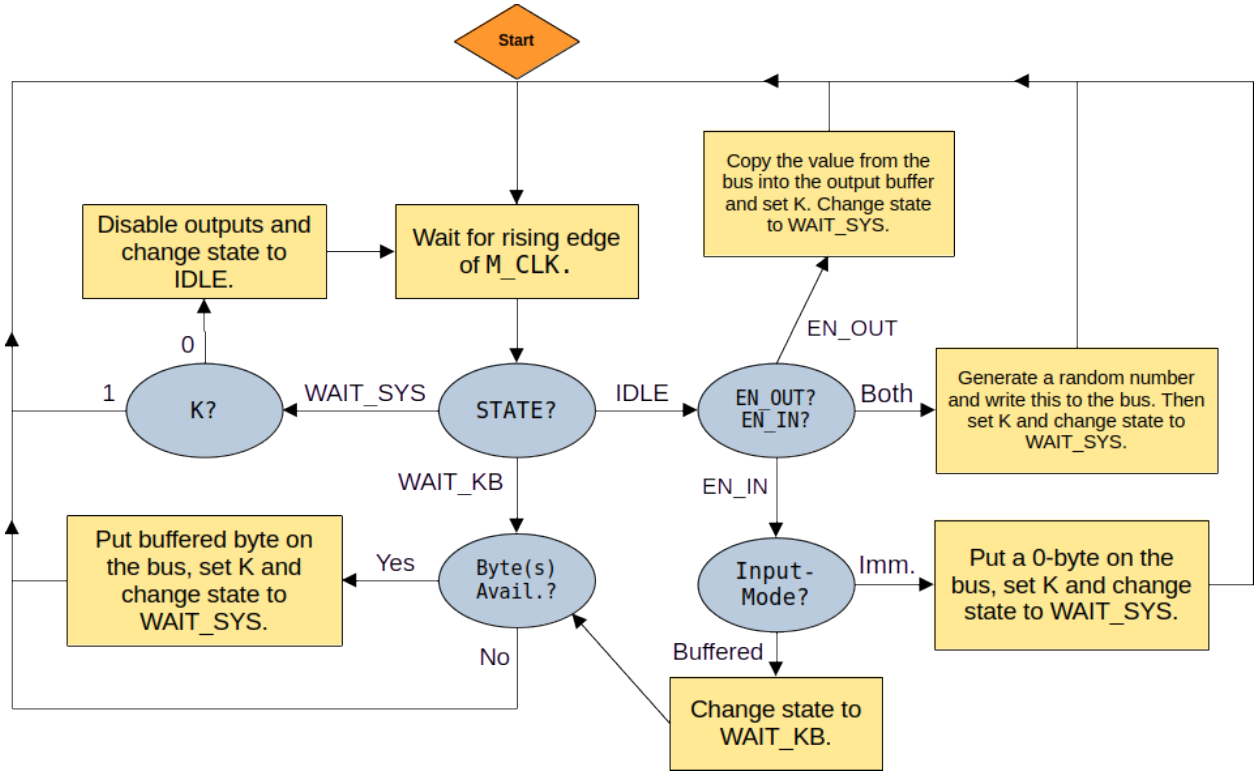


Figure 25: Control flow inside the ISR running on the microcontroller.

5.11.3 Shift Register

To decrease the number of pins needed to drive the LCD module, a shift register is used. In the current implementation, every pin except the RX/TX pins (reseverd for debugging over a serial connection) is used so using the shift register (74HC595) was vital.

5.11.4 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.

5.11.5 Keyboard

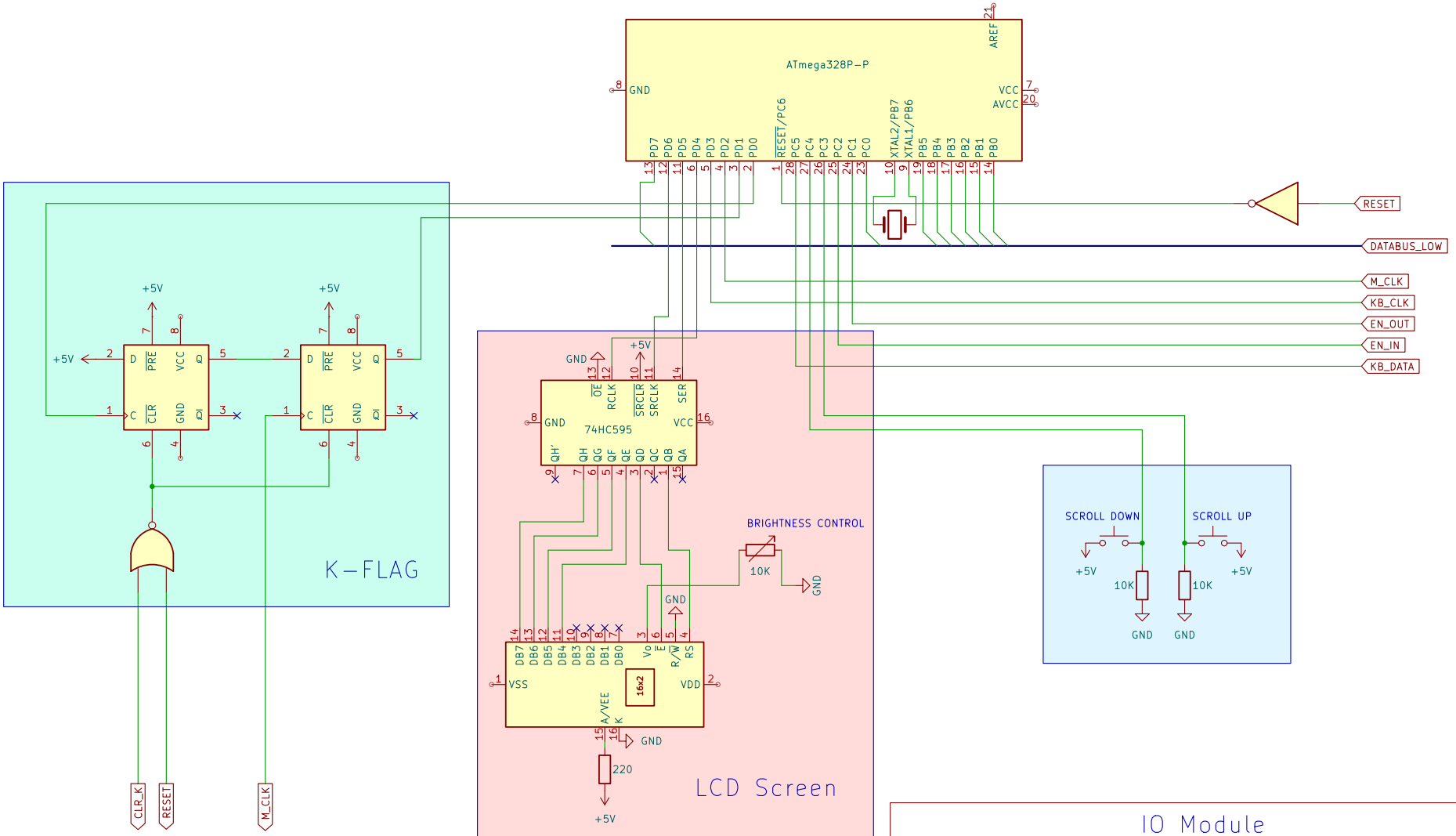
The IO module can only handle input from PS/2 compatible keyboards. A modified version of the `PS2Keyboard` library was used to implement the keyboard driver.

5.11.6 Software

The full source code for the IO module can be found on Github: `https://github.com/jorenheit/bfcpu/tree/main/src/io_module`.

5.11.7 Schematic

A full schematic is provided on the next page.



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

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 the binary opcodes that correspond to these commands. Table **??** lists all the available commands and the values that map to these commands. As explained in Section **??**, there are a few non-BF that have been added.

Command	Opcode
`NOP`	0x00
`+`	0x01
`-`	0x02
`<`	0x03
`>`	0x04
`,`	0x05
`'`	0x06
`.`	0x07
`[`	0x08
`]`	0x09
`INIT`	0x0d
`HOME`	0x0e
`HLT`	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 some preample commands to initialize the system and puts a `HLT` instruction at the end of the program to stop the computer when the program has finished (Figure **??**). Part of the source-code is listed in the listing below; for the full source, refer to `https://github.com/jorenheit/bfcpu/blob/main/src/bfasm/bfasm.cc`.

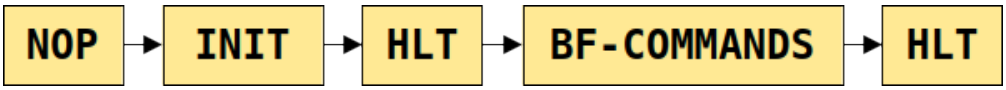


Figure 26: Result of assembling a BF-file.

6.2 Programmer

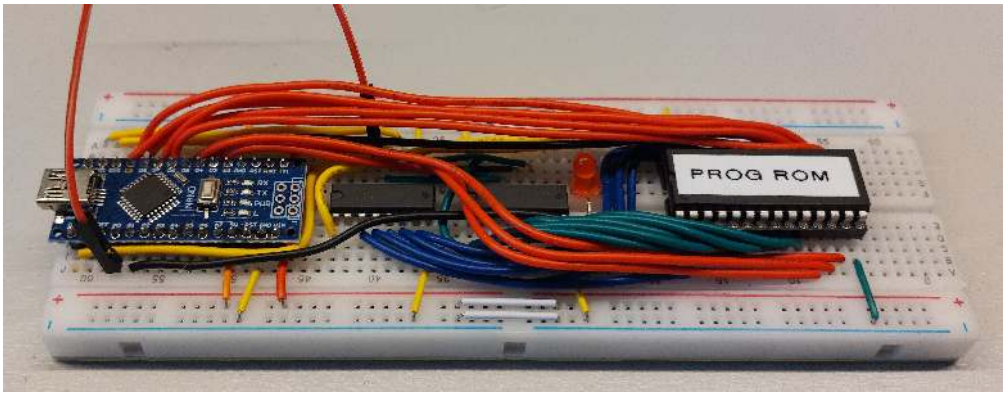
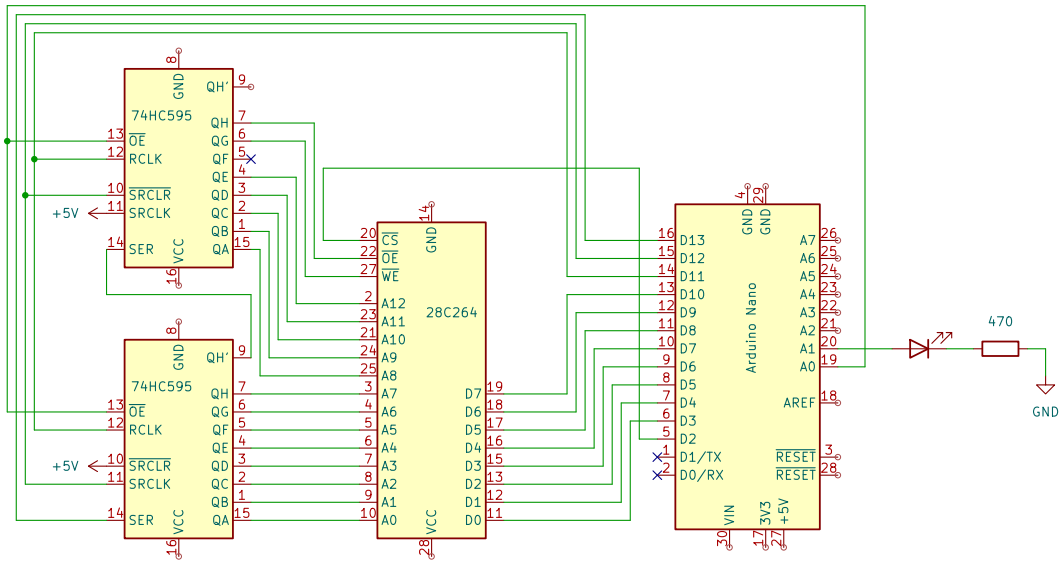


Figure 27: EEPROM chips were programmed using an Arduino Nano on a breadboard.

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 **??**) is shown on the next page.



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

6.3 Microcode Generation (Mugen)

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`. Section 1.10 shows the Mugen specification file for this project. When this file is passed to Mugen, it shows the resulting memory layout. which corresponds to the layout as shown in the schematics of the Control Unit.

```
$ mugen bfcpu.mu bfcpu.bin --layout
Successfully generated 3 images from bfcpu.mu:
  ROM 0 : bfcpu.bin.0
  ROM 1 : bfcpu.bin.1
  ROM 2 : bfcpu.bin.2

[ROM 0, Segment 0] {
  0: HLT
  1: RS0
  2: RS1
  3: RS2
  4: INC
  5: DEC
  6: DPR
  7: EN_SP
}

[ROM 1, Segment 0] {
  0: OE_RAM
  1: WE_RAM
  2: EN_IN
  3: EN_OUT
  4: VE
  5: AE
  6: LD_FB
  7: LD_FA
}

[ROM 2, Segment 0] {
  0: EN_IP
  1: LD_IP
  2: EN_D
  3: LD_D
  4: CR
  5: ERR
  6: UNUSED
  7: UNUSED
}

[Address Layout] {
  0: CYCLE 0
  1: CYCLE 1
  2: CYCLE 2
  3: OPCODE 0
  4: OPCODE 1
  5: OPCODE 2
  6: OPCODE 3
  7: FLAG 0
  8: FLAG 1
  9: FLAG 2
  10: FLAG 3
  11: UNUSED
  12: UNUSED
}
```

7 Conclusion

Looking back on the project, it was a very challenging, sometimes frustrating process that in the end paid off with a great feeling of accomplishment. It provided many tangible insight into computer architecture and proved that it is definitely possible to implement a Turing Complete system using basic parts and very little practicle experience in the field. There are most certainly many areas in which the computer can be improved, both in its efficiency and reliability. A natural next step might be to design a PCB that implements the architecture. This would probably increase the reliability and would allow the computer to run stable at much higher clock-frequencies, but would come at the cost of having less visually clear connections and therefore less educational value.

A big part of the project was the devlopment of the toolchain. We hope that tools like Mugen and the programmer hardware/software could be of value to the community, for others that are implementing a custom architecture or even following Ben Eater's instructions on the 8-bit breadboard CPU. As previously mentioned, all software is available through Github at `https://github.com/jorenheit/bfcpu`.

7.1 Parts List

Table **??** lists most of the parts used in this project. It is not meant to be a comprehensive list of every part used because this document is not meant to serve as a guide on building an identical system.

Part	Description	Quantity
Breadboard	830 hole	?
Wire	22AWG	A lot
74LS00 [**?**]	NAND	?
74LS04 [**?**]	NOT	?
74LS08 [**?**]	AND	?
74LS14 [**?**]	Schmitt-Trigger	?
74LS32 [**?**]	OR	?
74LS48 [**?**]	7-Segment Driver	?
74LS76 [**?**]	JK-Flip-Flop	?
74LS86 [**?**]	XOR	?
74LS123 [**?**]	Monostable Multivibrator	?
74LS138 [**?**]	3-to-8 Decoder	
74LS157 [**?**]	Data Selector	
74LS161 [**?**]	8-Bit Counter	
74LS173 [**?**]	8-Bit Register	
74LS193 [**?**]	8-Bit U/D Counter	
74LS245 [**?**]	8-Bit Bus Transceiver	
74HC595 [**?**]	8-Bit Shift Register	
NE555 [**?**]	555-Timer	
AT28C64B [**?**]	64K EEPROM	4
AS6C4008 [**?**]	512K SRAM	2
MC14068B [**?**]	8-Input NAND	1
ICM7226B [**?**]	Frequency Counter	1
7-Segment Display	-	2
Quad 7-Segment Display	-	2
LED (single/array)	Various colors	A lot
HD44780 Compatible LCD	4x20 Characters	1
Resistors	Different values	A lot
Capacitors	Different values	A reasonable amount

Table 5: Incomplete list of parts used in the project.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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