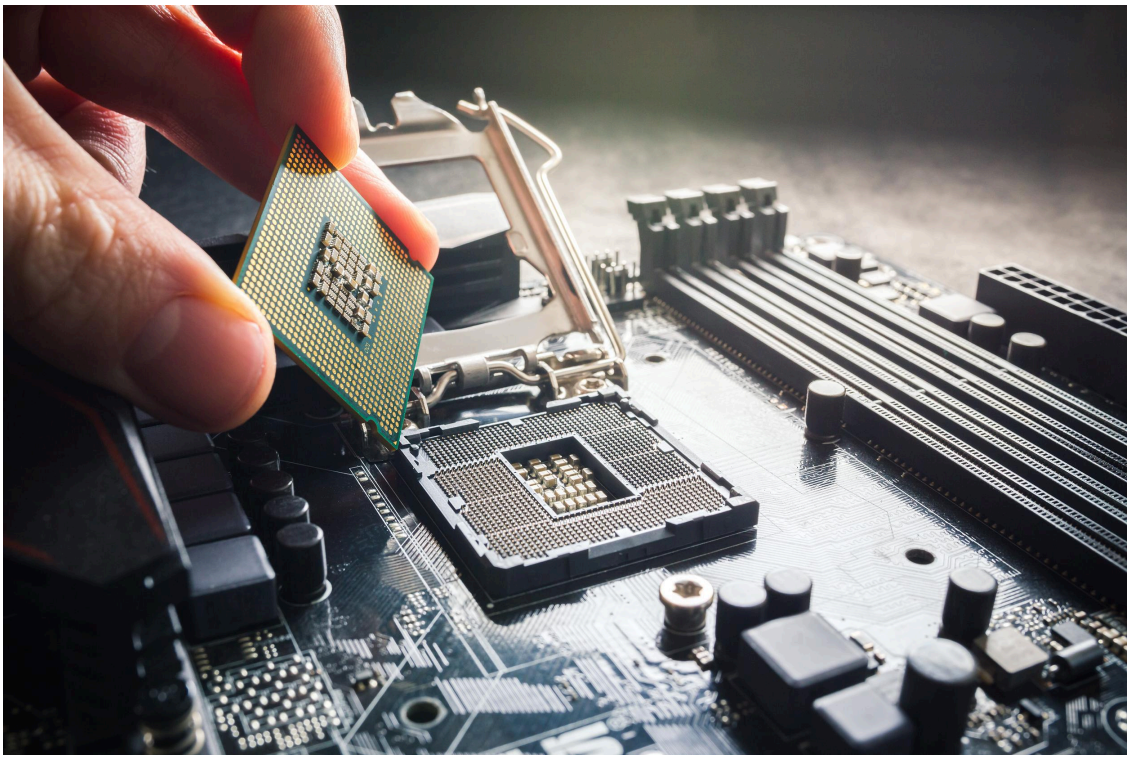
BFCPU 1.0

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 2.

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

The aim is to build a Brainf*ck CPU without making use of any programmable chips. We will only use discrete logic and relatively simple integrated circuits like registers, buffers and (de)multiplexers. The computer will be built on breadboards, as it was inspired by Ben Eater's 8-bit breadboard computer [4]. This report will document the design and implementation of the BFCPU.

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

2 Brainf*ck

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 1).

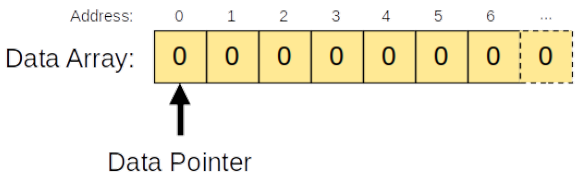


Figure 1: Initial state of the BF machine.

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

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

Although this might seem like a very limited set of instructions, it has been proven to be sufficient for performing any possible computation or program, also known as Turing-completeness [2]. The catch is that this requires an unbounded (or infinite) amount of memory, which is obviously impossible. However, the same caveat holds for traditional (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 1 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 [1] is fed into this program, it prints out the string `Hello World!`, as expected.

```
1  $ ./bfint "++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-] \
2  >>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++."
3 Hello World!
```

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

2.1 Architectures

2.1.1 Von Neumann

Modern computers are built according to the von Neumann architecture [3], which specifies a CPU (containing registers and an ALU), a single unit of memory and input/output devices (Figure 2). The registers of the CPU can be loaded with data from the memory unit and operated on by the ALU (Arithmetic and Logic Unit). Typical about this kind of architecture is the fact that not only data, but also the instructions (the program) are stored in memory. The program is therefore just as much part of the data as the data itself and can even be modified by itself.

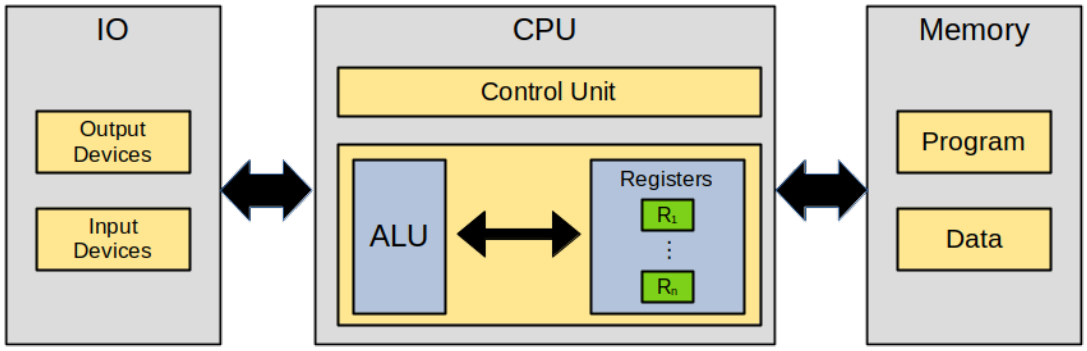


Figure 2: Schematic overview of the Von Neumann architecture.

2.1.2 Harvard

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

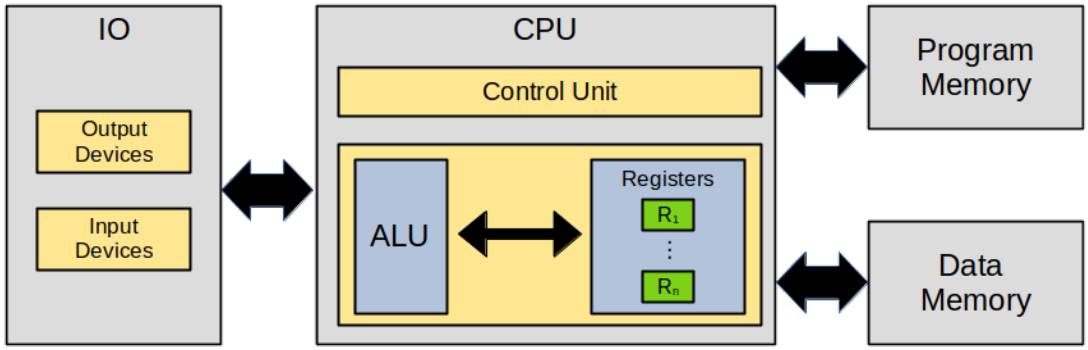


Figure 3: Schematic overview of the Harvard architecture.

The architecture assumed by the BF language is similar to the Harvard architecture, in that the memory does not contain the program itself. This implies that the program is stored somewhere else and cannot be addressed by the pointer, like in Listing 1, where the program was stored in a seperate array to the data. The ALU is very limited and can only perform increment, decrement and comparison to zero.

2.2 BF as an Instruction Set

Instead of viewing BF as a language that needs to be compiled or interpreted on a traditional machine, it can also be seen as an instruction set to a processor, built according to the BF architecture described above. An instruction set of size 8 is truly tiny compared to more traditional instruction sets such as those implemented by modern processors or even microcontrollers and older 8-bit systems. Broadly speaking, Complex Instruction Set Computers (CISC) are designed to do as much work as possible in the least number of clock cycles, whereas Reduced Instruction Set Computers (RISC) focus on having a small instruction set with basic operations. For comparison, the x86 instruction set is massive with over 1500 instructions implemented in hardware, whereas RISC processors only need to implement 50 to 100 instuctions. Even compared to RISC, the BF instruction set (BFISC from hereon) is tiny even compared to the smallest

instruction sets in use today. This isn't necessarily a good thing; a smaller number of instructions simply means you need more of them to perform meaningful computations, which is reflected by the fact that complex BF programs are typically very large in size.

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 4. Consider the BF program "`>>>>>+.`", applied to the initial conditions shown in the example. The pointer would take 5 steps to the right, landing on cell 9 which contains the number 41. It will then increment and output this value, displaying 42 on the screen (assuming a screen of some sort is used as the output device and it is displaying numbers directly rather than interpreting them as ASCII).

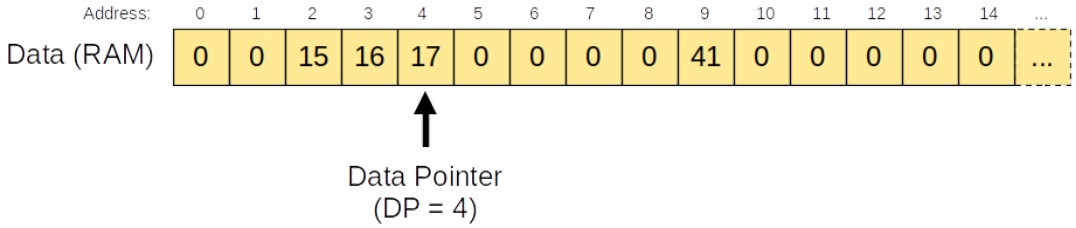


Figure 4: Example state of a BF machine.

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 four flags (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 5 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 5.

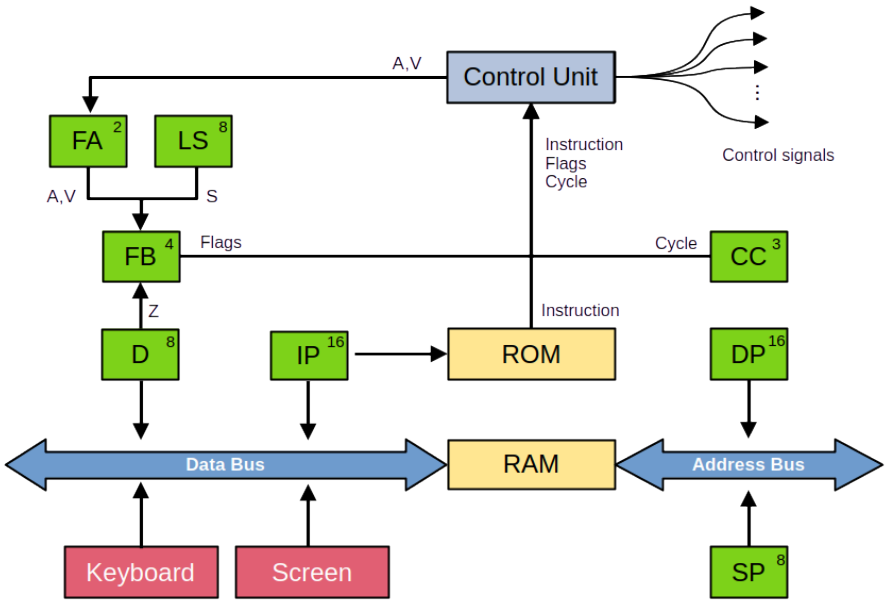


Figure 5: Connections between modules in the BF processor.

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

Inputs

The DP should be able to increment and decrement (corresponding to the `<` and `>` commands), and should be able to be enabled/disabled because of its connection to the address bus of the RAM (the Stack Pointer (SP, see 3.5, is also connected to this bus). Therefore, the DP has 3 control inputs:

- EN - Enable - Put 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. While all other modules have the ability to be reset, this is the only module where this signal is listed as it is the only module that needs to be reset during runtime (at initialization, see 4.8).

Outputs

- DP_OUT - 16 bits, connected to 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 leads to a synchronization issue D and RAM, which is handled by the control unit as described in Section 4. This register provides the Z flag to indicate 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 - Put 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 3.5). 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 - Put 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 that might need to be jumped to. The stack-pointer (SP) points to an address in this space; it is incremented whenever a new value is pushed to the stack and decremented whenever a value is popped off the stack. In this implementation, the SP is an 8-bit value, which means that at most 256 different values can be stored onto the stack before it the SP overflows (wraps around back to 0) and starts overwriting previous values. This would happen if a BF program was loaded that has more than 256 nested `[]`-pairs. Although possible, it is very unlikely to happen for the simple programs we intend to run.

Inputs

- EN - Enable - Put the stored value onto the databus.
- U - Up - Increment the stored value.
- D - Down - Decrement the stored value.

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 `]`. 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 opening `[` or a closing `]` is encountered. On the former, the LS is incremented again while on the latter the LS is decremented. This has the effect that the LS becomes 0 again after the `]` that matches the original `[` which led to the skip. Normal execution occurs as soon as LS has become 0 again 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. 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.

Inputs

- SETA - Set the address-change-flag.
- SETV - 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 CU (See **??**).

3.8 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 5.3) to drive register modules that supports modification of its 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 1 contains an overview of each of the registers and the control signals they support.

Register	#Bits	EN	LD	INC	DEC	RS
D	8	x	x	x	x	001
DP	16	x		x	x	010
SP	8	x		x	x	011
IP	16	x	x	x		100
LS	8			x	x	101
FA	4		x			-
FB	4		x			-

Table 1: Control signals available to each of the registers. The F and I register 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.

3.9 Cycle Counter (CC)

Almost every BF instruction will require 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 to prepare for the next instruction.

Inputs

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

Outputs

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

3.10 Data Memory (RAM)

The RAM is divided into two parts: stack and data. The first 256 bytes (0x0000 - 0x00ff) make up the stack and are indexed by the stack pointer (3.5). The data (corresponding to the BF tape) is stored at addresses 0x0100 through 0xffff and are indexed by the data pointer (3.2).

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 - Put 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 lines as DATA_IN).

3.11 Program Memory (ROM)

The actual BF instructions are stored in Read-Only-Memory (ROM) and are addressed by the IP (3.4). 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;

3.12 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.

Inputs

- DATA_IN - 8 bits - connected to the data bus;
- EN: Enable - Display the contents of the bus. The way output is displayed could vary depending on the implementation of the output device.

Outputs

None.

3.13 Keyboard (KB)

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

Inputs

- EN - Enable - Make the contents of the input buffer available on the databus.

Outputs

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

3.14 Control Unit

Each of the aforementioned components/modules has one or more control inputs that determine what happens on the next clock cycle. For example, some register-modules can be told to load a value from their input, increment or decrement the currently stored value, or do nothing at all. It is the Control Unit (CU) that supplies the appropriate control signals to each of the modules before the next clock pulse occurs, depending on the current instruction and state determined by the flags and cycle counter. The implementation details of how this is done in hardware are discussed in section 5.

Inputs

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

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 [2]
- OE_RAM - Output Enable RAM
- WE_RAM - Write Enable RAM
- EN_IN - Enable input (keyboard) to databus
- 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
- ERR - Error Signal

[2]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.

4 Control Sequences

4.1 Instruction Decoding

By setting the control signals as described in Section 3 appropriately, the modules can work together to perform each of the BF instructions. Table 2 shows the control sequences that are executed per BF instruction (also known as microcode instructions). The Control Unit implements this as a lookup table in 3 ROM chips, where the instruction (4 bits), flags (4 bits) and cycle counter (3 bits) act as an address into this table (Figure 6). There are 22 different control signals, so 3 8-bit EEPROM chips have been used to store the entire table. More details on the implementation can be found in Section 5.10. Below we will go through each of the instructions in order to annotate the contents of Table 2.

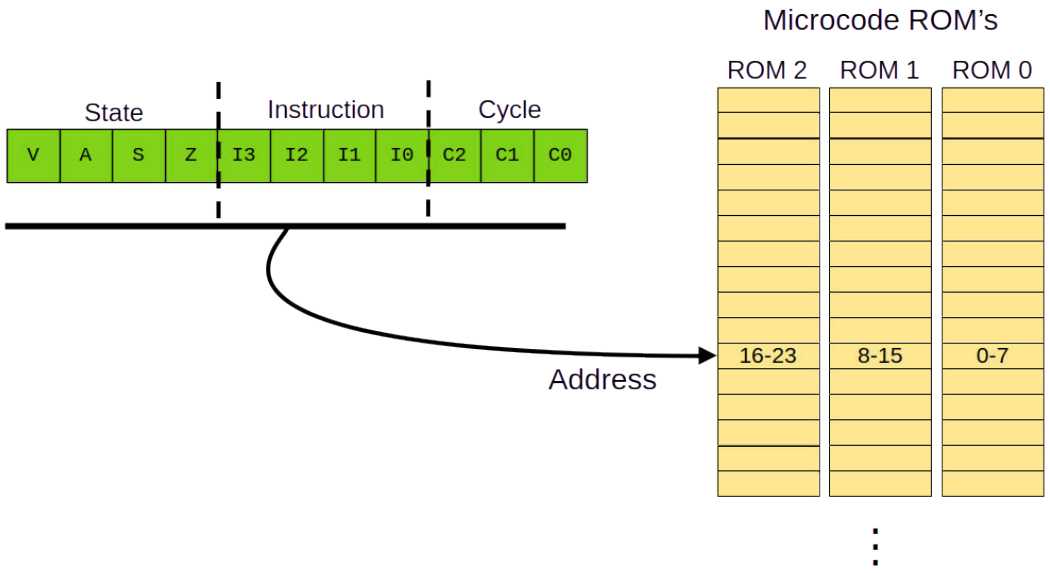


Figure 6: Decoding an instruction: the current instruction, state and cycle-count function as an index to the ROMs, which return the control signal configuration.

It is important to note that, because of the choice of driving all of the (counting) registers with a common interface (Section **??**), 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

The first cycle of each instruction is identical: all flags (A and V from FB and S and Z from LS and D) are loaded into the FB register. This provides the CU with all the necessary information to determine the control signals for the next cycle (the instruction is loaded directly from ROM and does not have to be enabled).

4.3 Modifying Data: + and -

The sequence of instructions necessary to execute a + command depends on the state of the system. If the A flag is not set, the value in D already corresponds to the value currently pointed to by the DP. In that case, its INC signal is set in order for it to increment on the next clock pulse. Additionally, the V-flag is set to indicate that the value in D has been changed, which is then loaded into the FA register. In the next cycle, the INC signal is set for the IP register and the cycle counter is reset to prepare for the next instruction.

However, when the A flag *was* set, we first need to fetch the correct value from RAM by enabling the DP register and loading the resulting value into D. From hereon, the next set of control signals is identical to that described above in the case where A was not set.

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

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.

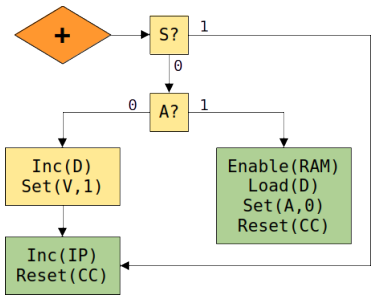


Figure 7: Block diagram for the `+` command. The diagram for the `-` command is equivalent (using `Dec` rather than `Inc`).

4.4 Moving the Pointer: < and >

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

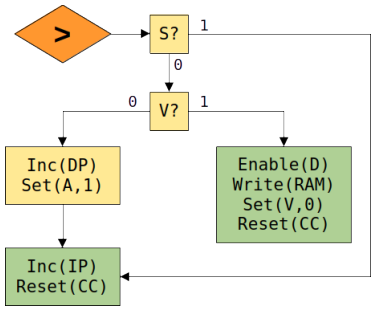


Figure 8: Block diagram for the `>` command. The diagram for the `<` command is equivalent (using `Dec` rather than `Inc`).

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. Also, if the loop is not entered, the LS-register is used to determine when execution should resume.

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

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

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

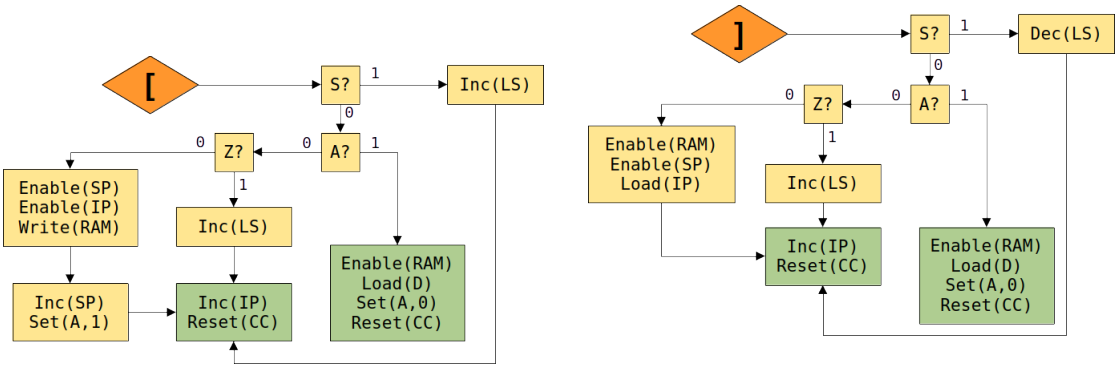
In the case that we were already skipping code (S set), we need to increment the LS-register once more to account for another pair of nested `[]`'s (cycle 1) and then continue to the next instruction (cycle 2).

Loop End: In the first scenario, which takes 2 cycles to execute, there is a known (synchronized) zero in the D-register (Z set, A not set). This means we can immediately choose to exit the loop. To do so, the stack-pointer is decremented (cycle 1) to point at the previous value on the stack. In cycle 2, the IP is incremented as usual.

If there is a known nonzero value in D (scenario 2: Z not set, A not set), this means we must loop back to the IP stored on the top of the stack. This value is loaded into the IP-register by enabling the SP and RAM and setting the LD signal for the IP-register (cycle 1). On the second cycle, this new IP (pointing to a `[`) is incremented to re-enter the loop.

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

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



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

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

Figure 9

4.6 Output: .

There are three states that need to be taking into account when implementing the output-command. In the first state, the value in D is already in sync with the RAM and can be sent to the output device immediately. The output of D is enabled and PRE signal for the screen is set, in addition to incrementing the IP to move to the next instruction. When A was set, we first load the value from RAM into D before sending its contents to RAM. Lastly, when S is set, this instruction can simply be skipped.

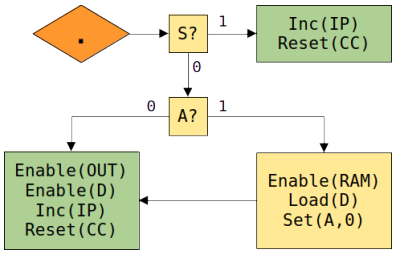


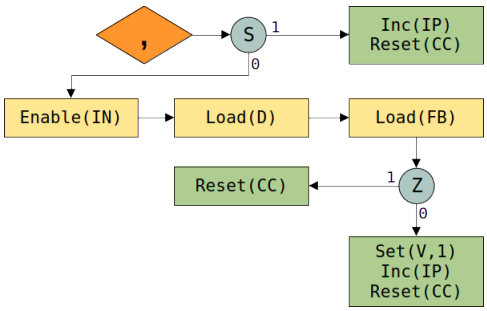
Figure 10: Block diagram for the output command.

4.7 Input: , and ’

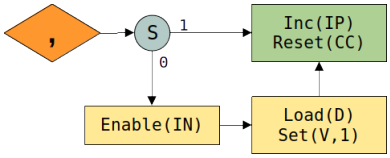
As mentioned before in Section 3.13, the architecture implements two versions of the input command, buffered (,) and immediate (’). Because buffered inputs are more common in most (BF) programs other than game-like applications, this is the default mode (even though it is more complicated).

Buffered Mode: In this mode, control flow is stuck in a loop, waiting for something nonzero to appear in its D-register. A nonzero value means something was read from the input module, whereas a zero value indicates that the input buffer was empty. When the EN signal is presented to the keyboard module, this module should provide the next character in the input-buffer (implementation defined) onto the databus on the next clock cycle, when it is loaded into the D register. After doing so, the flag register (FB) is reloaded to update the Z flag, which is then used to decide between either moving to the next instruction (when Z is 0) or looping back to cycle 0 (without incrementing the IP). If Z is now nonzero, the V flag is set to indicate that the contents of the cell have changed and the IP is incremented to move to the next instruction. If Z is 0, the cycle counter is reset to 0 to put the system back into the state above where the keyboard-buffer can be read into the D-register. Of course, when S is set, we can simply skip all of this.

Immediate Mode: In this mode, we don’t care what was loaded into D, even if there was nothing there in the keyboard buffer; it is up to the programmer to handle the case where no keys were pressed. This makes the implementation a lot easier, taking only 3 cycles to complete. We simply read whatever is in the keyboard buffer into D, set the V flag and move to the next instruction. Again, when S is set, this instruction is skipped.



(a) Block diagram for the bufferen input command.



(b) Block diagram for the immediate input command.

Figure 11

4.8 Non-BF instructions

Several non-BF instructions have been implemented for debugging purposes and to have the computer to be initialized at startup.

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 INIT

BF assumes that memory is zero-initialized. In practice, SRAM-modules will contain random values at startup, so the assembler must add a preamble to the main code in order to initialize the RAM (or 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. By incrementing the LS data pointer at the same time and inspecting the S flag on the next cycle, the computer keeps track of the number of cells that have been initialized. While the LS register has not looped around back to 0, resetting the S flag, the data pointer is incremented to move to the next cell. When finally the S flag resets, the datapointer has to return to its starting value. For this purpose alone, the `HOME` instruction has been added.

4.8.3 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.

4.8.4 HLT

The HLT instruction halts the clock and (temporarily) stops the program. When enabled, the assembler will interpret an exclamation mark (`!`) as `HLT` in the BF-code. Moreover, the assembler will insert the HLT instruction at the end of the program and provides the option to insert a HLT instruction after initialization to pause the program before the code is executed.

4.8.5 ERR

The `ERR` instruction is inserted in all non-reachable states. If for some reason a state occurs that maps to the `ERR` command, the clock will be halted and some indicator on the Control Unit should light up to let its users know that something has gone wrong. It is therefore functionally the same as the HLT instruction, the only difference being that an addidional light indicates the error status.

4.9 Microcode table

Table 2 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 2 as INC(SP).

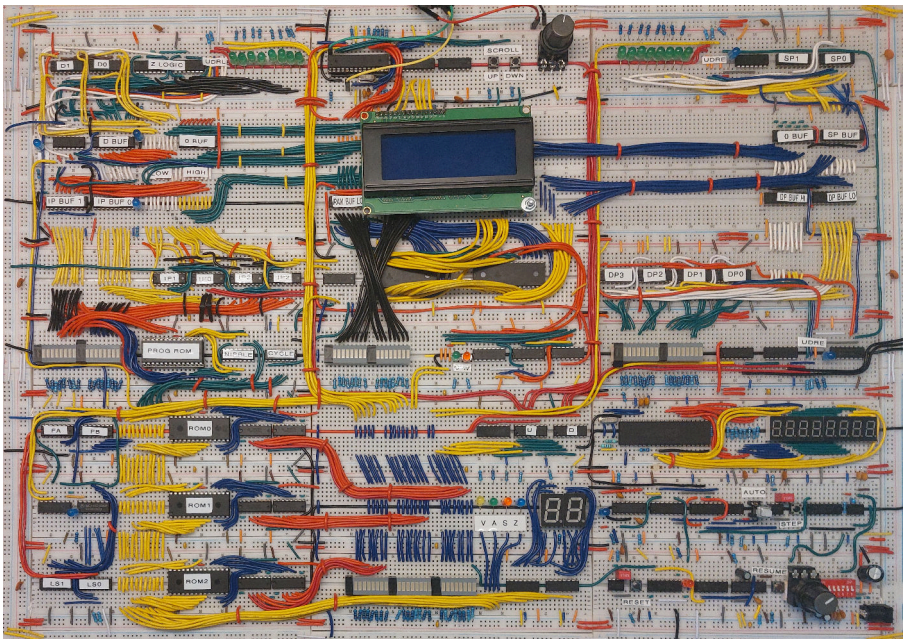
Instr	V	A	S	Z	Cycle	Control Signals				
Any					0	LD(FB)				
+		0	0		1	INC(D)	SETV(FA)	LD(FA)		
		0	0		2	INC(IP)	CR(CU)			
		1	0		1	LD(D)	OE(RAM)	LD(FA)	CR(CU)	
			1		1	INC(IP)	CR(CU)			
-		0	0		1	DEC(D)	SETV(FA)	LD(FA)		
		0	0		2	INC(IP)	CR(CU)			
		1	0		1	EN(DP)	LD(D)	OE(RAM)	LD(FA)	CR(CU)
			1		1	INC(IP)	CR(CU)			
>	0		0		1	INC(DP)	SETA(FA)	LD(FA)		
	0		0		2	INC(IP)	CR(CU)			
	1		0		1	EN(D)	EN(DP)	WE(RAM)	LD(FA)	CR(CU)
			1		1	INC(IP)	CR(CU)			
<	0		0		1	DEC(DP)	SETA(FA)	LD(FA)		
	0		0		2	INC(IP)	CR(CU)			
	1		0		1	EN(D)	EN(DP)	WE(RAM)	LD(FA)	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)	
		1	0		2	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)	
		1	0		2	CR(CU)				
			1		1	DEC(LS)				
			1		2	INC(IP)	CR(CU)			
.		0	0		1	EN(SCR)	EN(D)	INC(IP)	CR(CU)	
		1	0		1	OE(RAM)	LD(D)	LD(FA)		
		1	0		2	EN(SCR)	EN(D)	INC(IP)	CR(CU)	
			1		1	INC(IP)	CR(CU)			
,			0		1	EN(KB)				
			0		2	LD(D)				
			0		3	LD(FB)				
			0	0	4	SETV(FA)	LD(FA)	INC(IP)	CR(CU)	
			0	1	4	CR(CU)				
			1		1	INC(IP)	CR(CU)			
’			0		1	EN(KB)				
			0		2	LD(D)	SETV(FA)	LD(FA)		
			0		3	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)			
ERR						ERR(CU)	HLT(CLC)			

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	R(DP)		

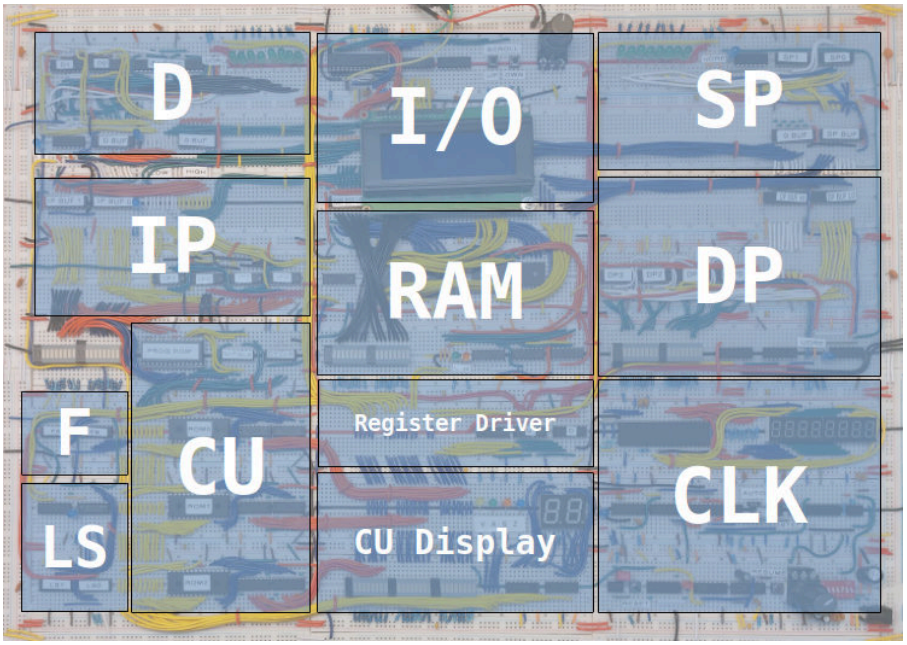
Table 2: Control signals for each of the BF instructions in different scenario's, depending on the state flags. Note that in order to distinguish between the two input modes, the regular comma (`,`) and the apostrophe (`'`) are used for buffered and immediate inputs respectively. See also Section **??**.

5 Implementation

This section will discuss the implementation of each module and the way they integrate together to make the computer. Figure 12a shows the computer as it was in February 2025. The overlays shown in Figure 12b 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

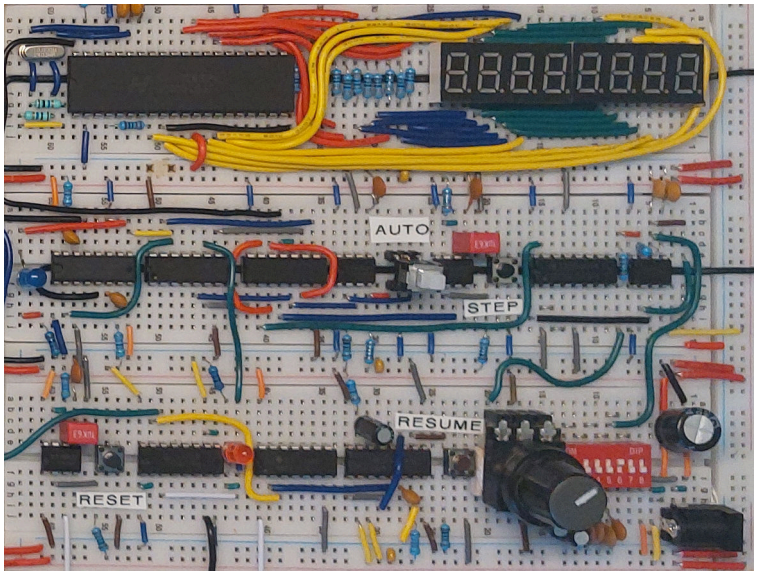


Figure 13: Close up of the Master Clock and Reset/Resume Modules.

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 design of the clock is taken directly from Ben Eater's 8-bit computer video's [4]. The automatic clock is generated by a 555 timer in astable mode. A set of DIP switches is used to select the capacitor of the RC-circuit that determines the output frequency for coarse control and a 100K linear potentiometer is used for fine control of the clock frequency. 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 divided 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 original clock), the control signals are loaded from the microcode EEPROMs into their registers (which connected to the modules), while the second pulse is connected to the modules to act upon. This approach guarantees a clean division between setting the control signals and clocking the modules. Figure 14 shows the timing diagram for the different signals discussed above.

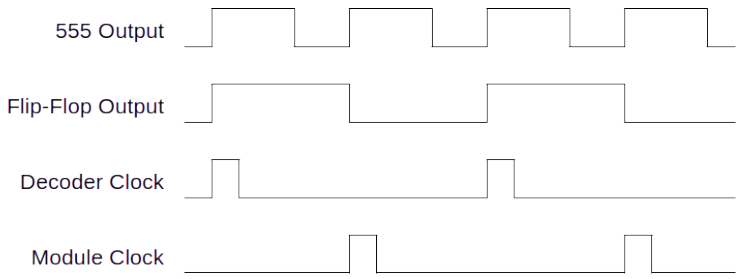


Figure 14: Timing diagram for the clock signals.

5.1.1 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)

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

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

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 3 should 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 3).

5.1.2 Frequency Display

To be able to see the frequency at which the computer is running, the module clock signal is connected to an ICM7226B [**?**], 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). For this module, no schematics will be provided.

5.2 Reset/Resume

The Reset/Resume module is located directly underneath the clock and contains logic necessary to be able to reset the computer (necessary after applying power) and resuming 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 resumes. This allows for pausing and resuming the computer, effectively adding breakpoints in 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.

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 14), 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 a second is achieved. Figure 15 shows a timing diagram to illustrate this process in more detail.

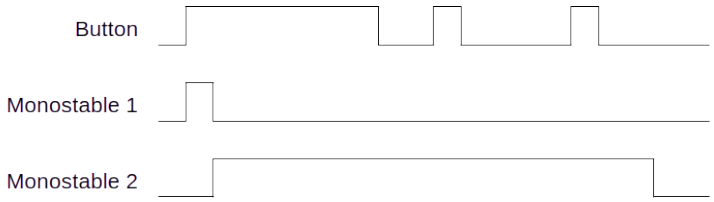
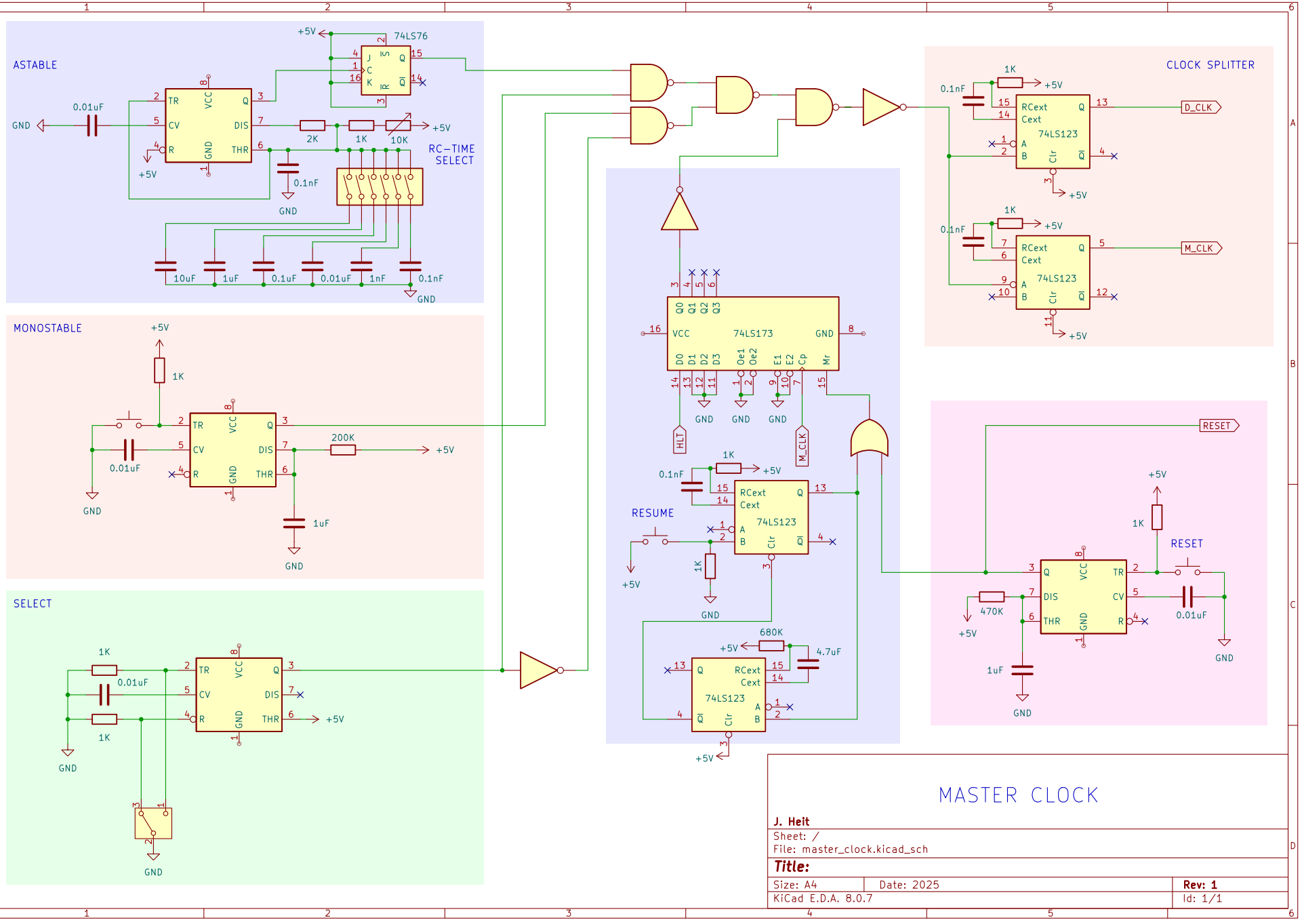


Figure 15: Timing diagram for the resume debouncer. The output of Monostable 1 is connected to the reset of the register that stores the HLT signal.

5.2.1 Schematic

A full schematic is provided on the next page.



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

5.3 Register Driver

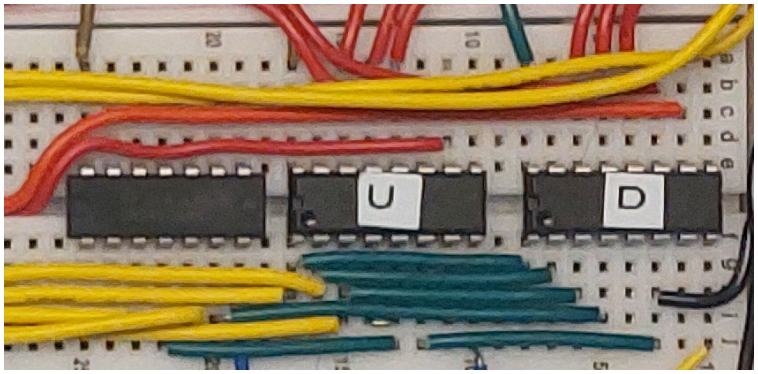


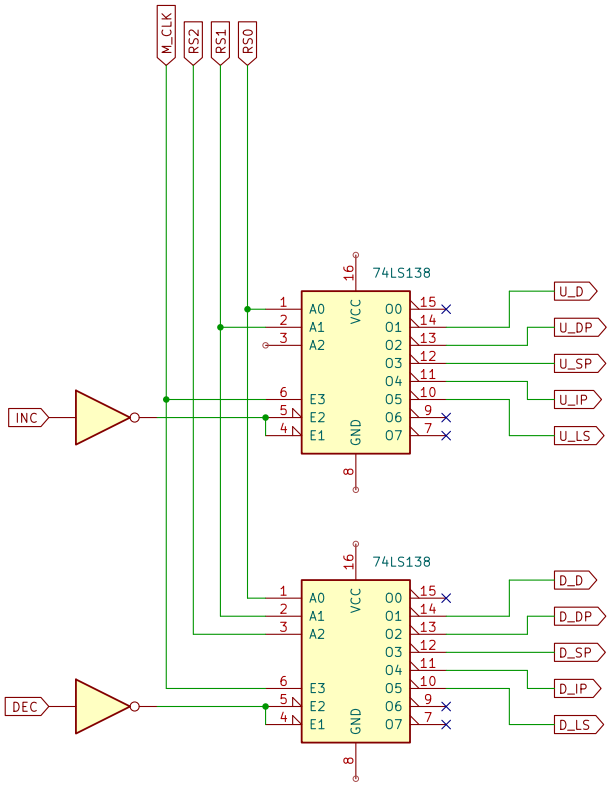
Figure 16: Close up of the Register Driver Module.

The register driver is responsible for sending the correct signals to the counting input of the D, DP, SP, IP and LS registers. All of these are based around the 74LS193, which has a U and D input and requires one of them to be pulsed low in order to execute the corresponding action. For example, to decrement this register, a low pulse must be sent to the D input while U is kept high. As explained in section **??**, we used a centralized driver to limit the number of logic IC's necessary to drive the registers and the total number of control signals necessary.

The driver module makes use of a pair of 74LS138 decoders. This chip takes 3 bits (A, B and C, connected to the register-select signals RS0, RS1 and RS2) to address one of its outputs and two gating inputs G1 and G2. When both gates are active, a low signal will be provided on the output pin determined by the address set on A, B and C. By letting the module-clock (M_CLK) pulse one of the gates, a low pulse is created on the output selected by the values present at A, B, and C and routed to the corresponding counting module. We use the first '138 for generating the U signal and the second '138 for the D signal. The INC and DEC signals are connected to the other gates to determine which of the decoders is generating the pulse.

5.3.1 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

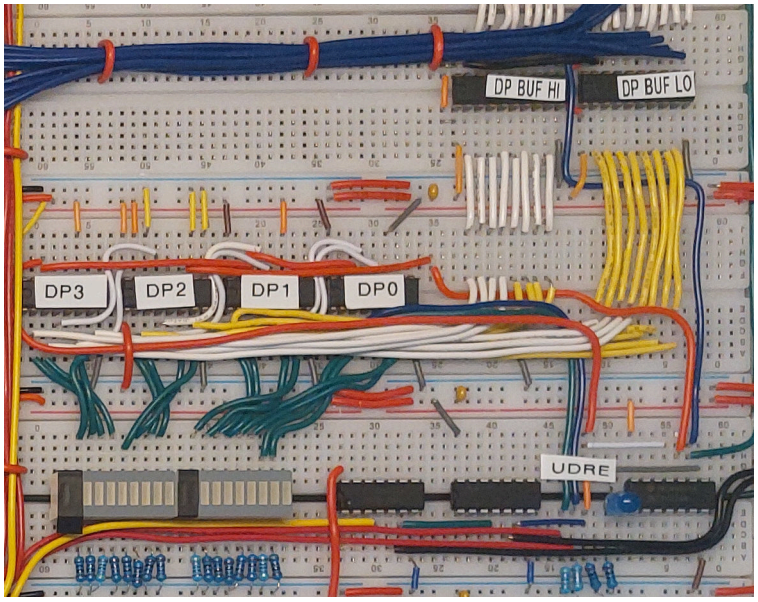


Figure 17: Close up of the Data Pointer Register Module.

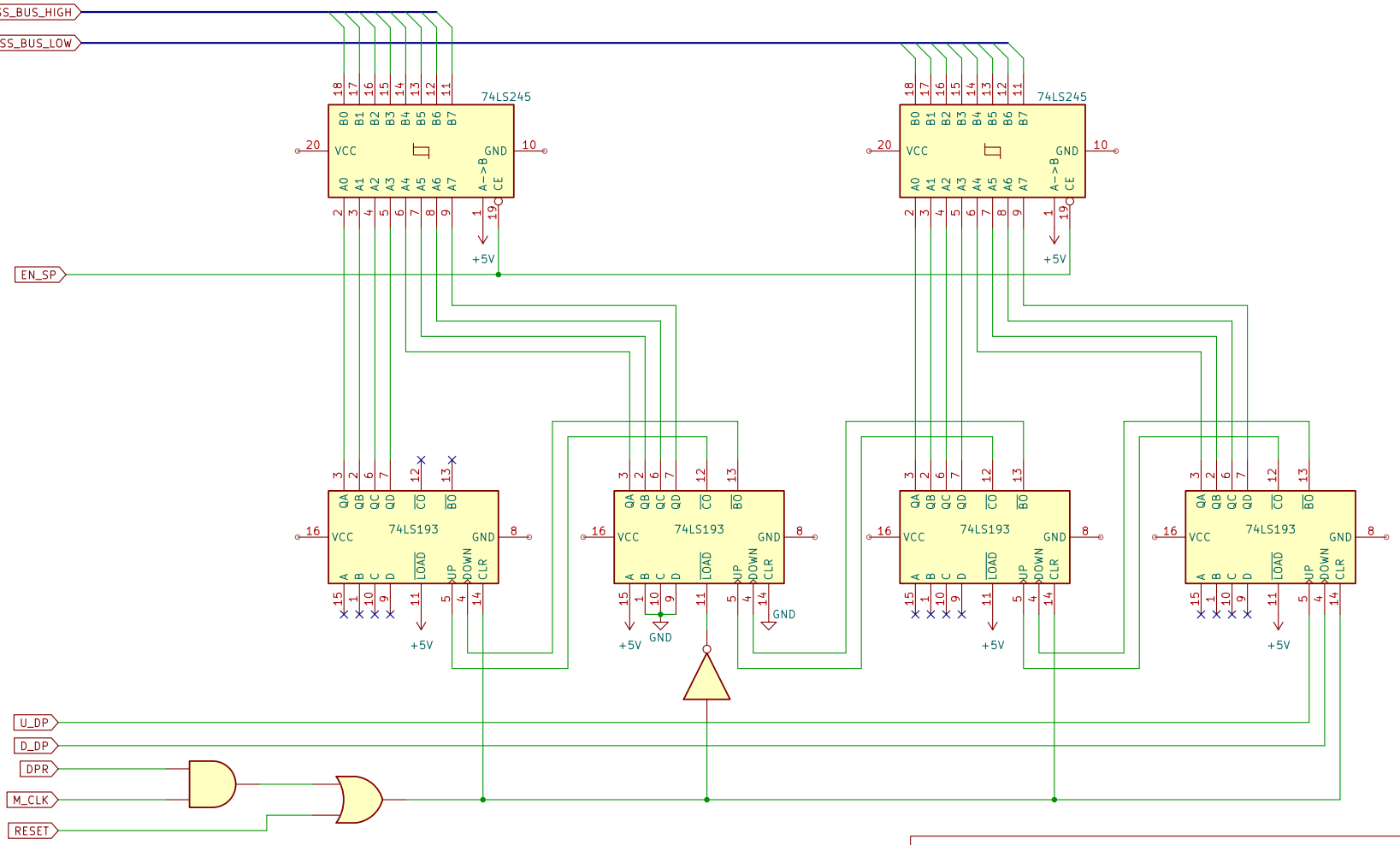
The datapointer register contains the address of the cell currently being operated on. It is a 16 bit value kept in four 74LS193 IC's because of the requirement that is should be able to be incremented and decremented. Their outputs are connected to the address bus through a pair of tristate buffers to prevent bus contention with the stack pointer. Because the first 256 bytes are reserved as the stack (where instruction pointer values can be stored to implement loops), the reset value of this register should be set to 0x0100. This is achieved by resetting all IC's except for the one containing the 3rd nibble, which is reset to one by hardcoding 0x1 to its inputs and connecting the reset line (through an inverter) to the load-pin instead (and clock for synchronization).

This register is special in the sense that it is the only register that should be able to be reset at runtime and without resetting any of the other modules (through the DPR signal). This allows it to be reset to its starting value after initializing the computer when power is first applied (see also Sections 4.8.2 and 4.8.3). The RESET signal is therefor fed into an OR gate together with the DPR signal before going to the reset pins of the IC's.

Perhaps somewhat confusingly, the schematic (see next page) shows that the SP_EN signal is used to enable the buffers. Since the bus is shared only between the stack pointer and data pointer, 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. Since the output enable pin of the 74LS245 is active low, the SP_EN can be fed directly into it. On the side of the stack pointer, the same signal goes through an inverter before going into the buffer. By default, when the stack pointer is not enabled, the datapointer will provide the address to the RAM. The RAM address input lines will never be floating which means that the contents of the current memory cell (in RAM) will always be visible in the computer.

5.4.1 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

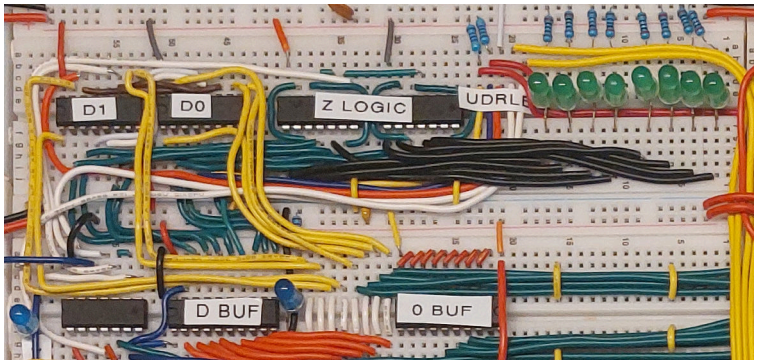


Figure 18: Close up of the Data Register Module.

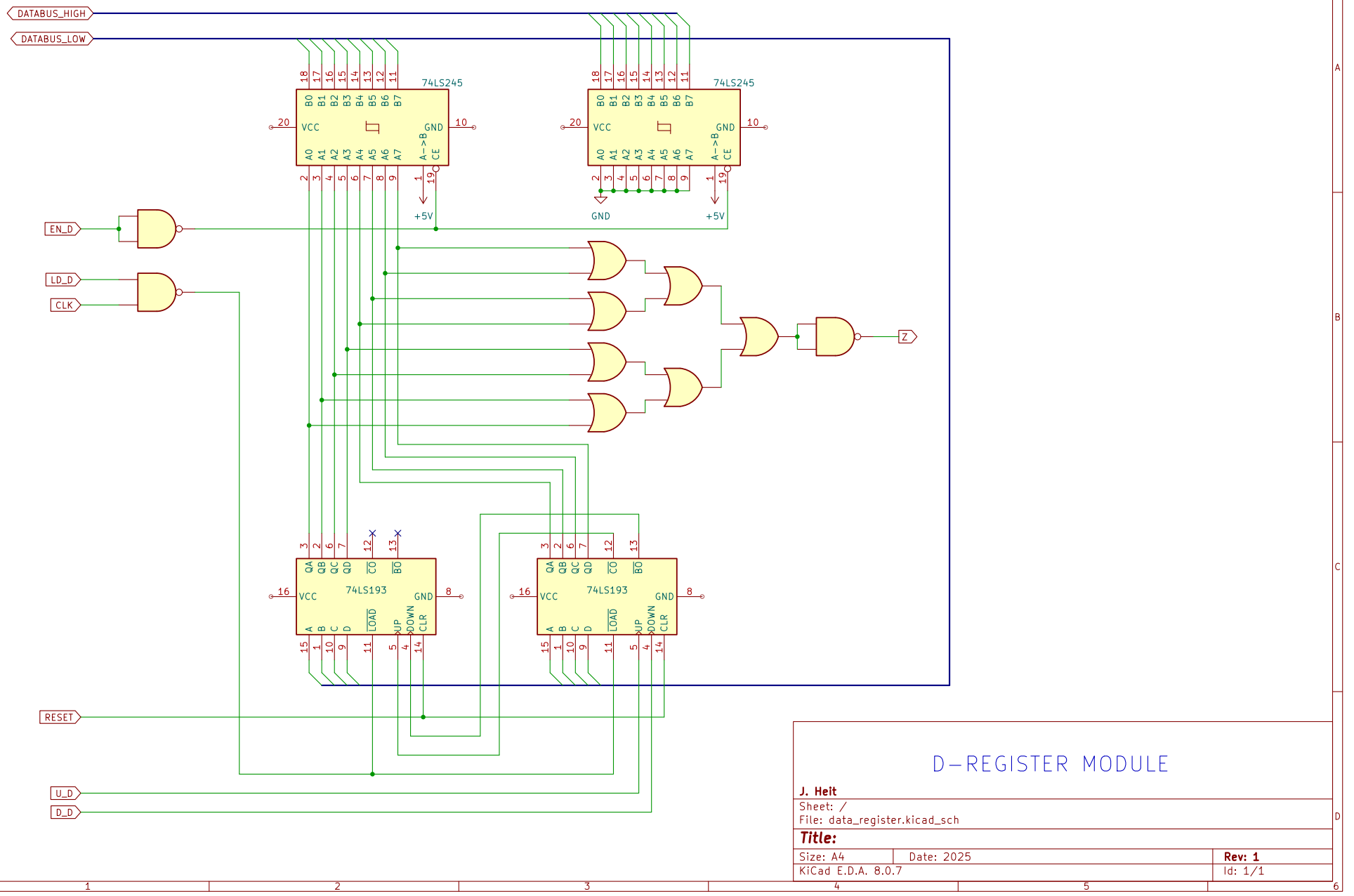
The data register holds (a copy of) the value in memory currently pointed to by the datapointer. In the computer, it is located in the top left corner. It is based around the 74LS193 counting register and driven by the register driver described in Section 5.3. The output is buffered in a tristate buffer (74LS245) before being connected to the databus. A second buffer is used to send all zeroes to the high byte of the databus since the data is only 8 bits wide. The buffers are set to output-mode only (even though the register is able to read from the bus as well) because the 74LS193 chips have seperate pins for incoming and outgoing data. The incoming data is read from the bus directly without going through a buffer.

This module also produces the Z flag, indicating that it is currently containing the value 0. This is achieved by sending the output through an array of OR gates and finally an inverter. The output is then sent to the second half of the instruction register which is responsible for keeping track of the flags.

Because the 74LS193 is loading asynchronously, the clock is gated together 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.

5.5.1 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. 8.0.7		Id: 1/1

5.6 IP Register Module

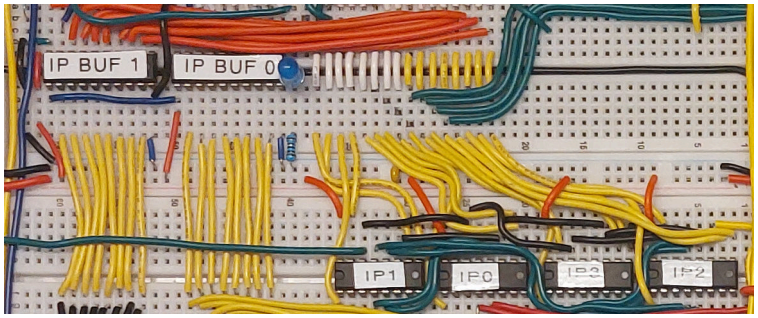


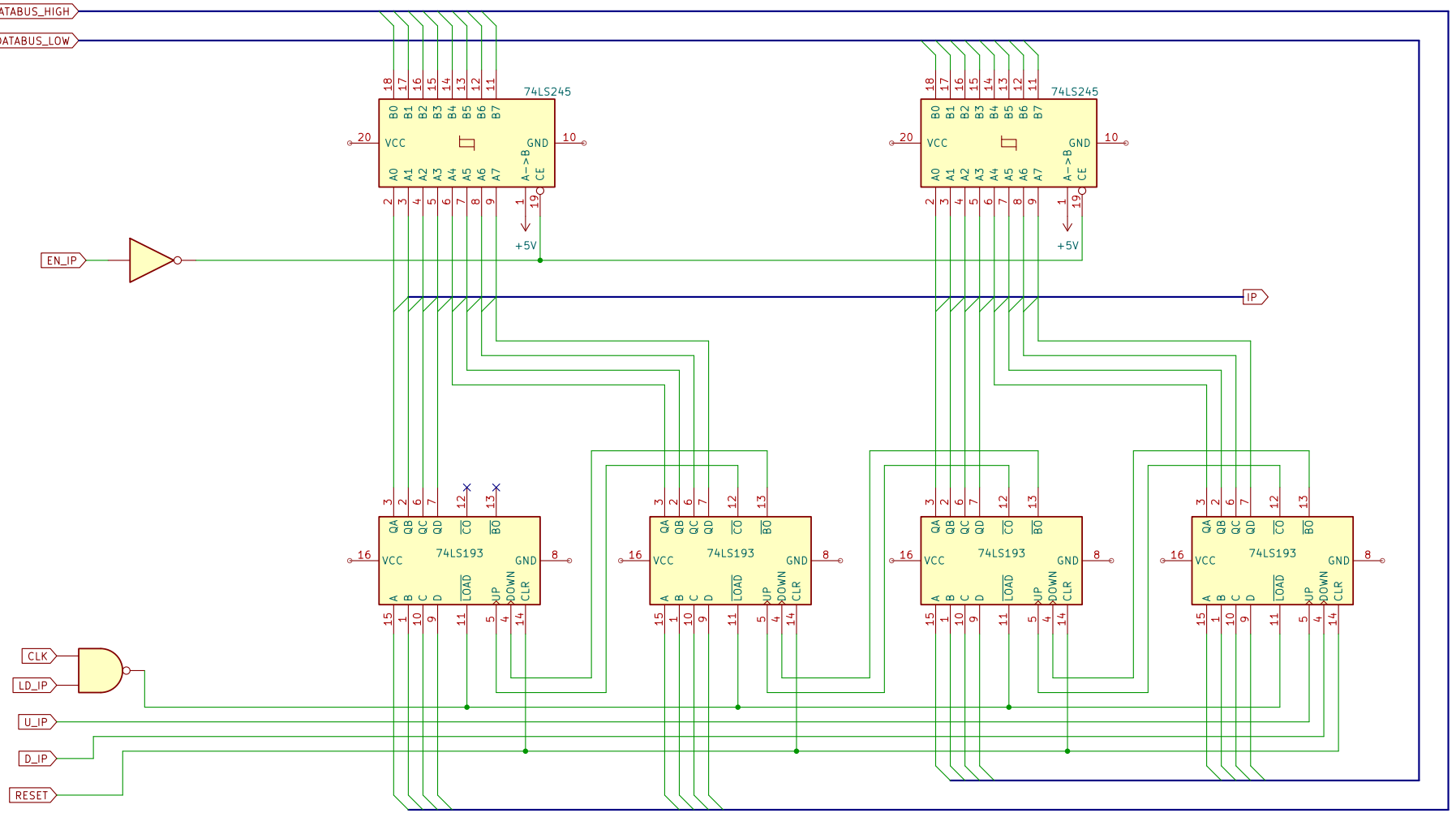
Figure 19: Close up of the Instruction Register Module.

The instruction pointer register holds a 16-bit value representing the address of an instruction in program-memory, stored in an EEPROM module (see **??**). Because the size of the available address space is 2^{14} instructions, the final two bits of the IP are not used. Red LEDs have been connected to these bits to quickly identify programs that try to address this non-existing part of program memory. The register IC's (74LS193) are driven by the register driver.

The IP is connected to the databus through two tristate buffers (74LS245) to avoid bus contention with the datapointer and keyboard module. It needs to be 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.

5.6.1 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

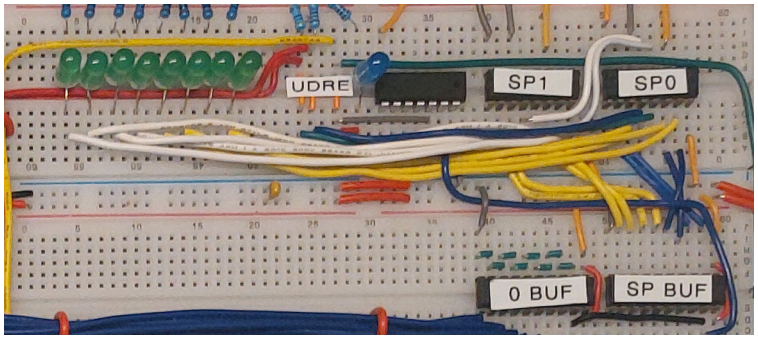
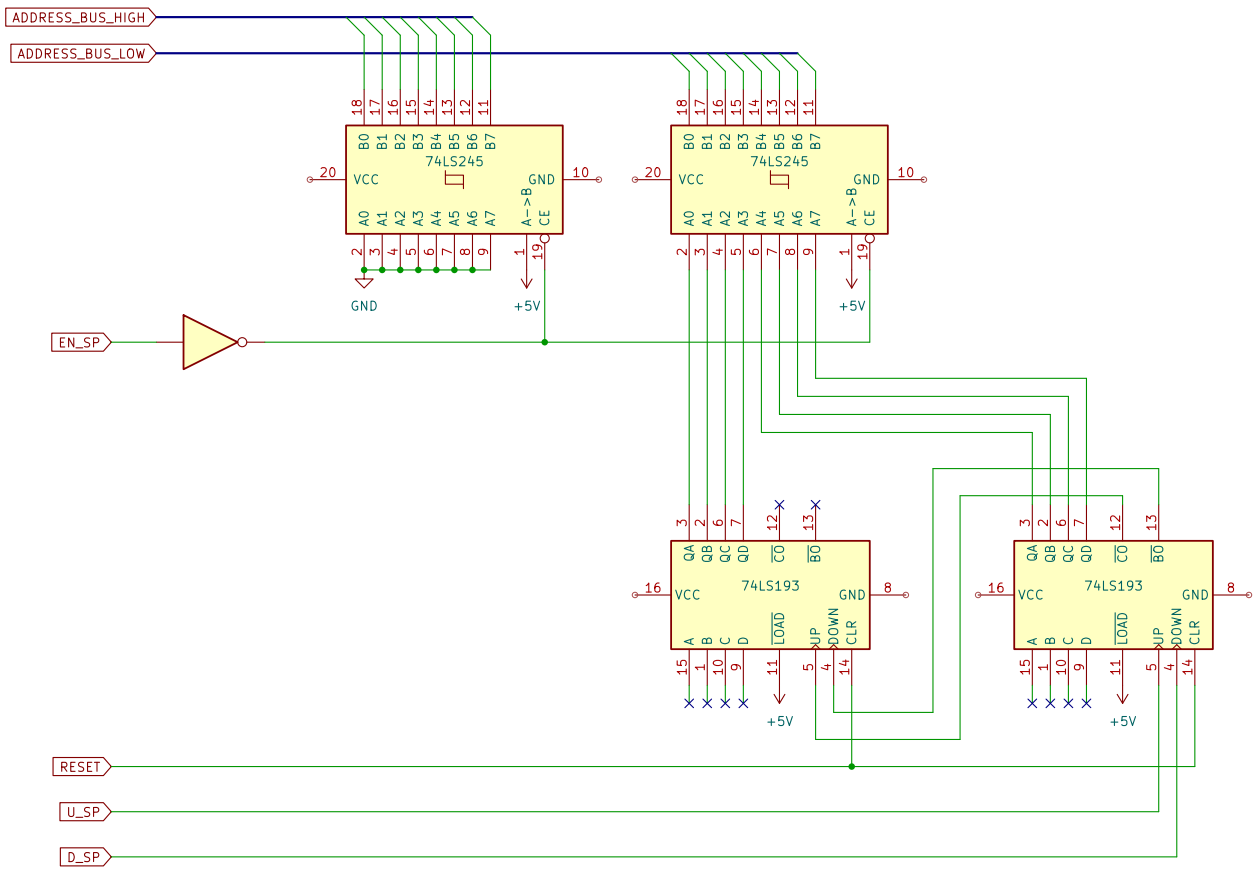


Figure 20: Close up of the Stack Pointer Register Module.

The stack pointer is an 8 bit value representing a memory address in the range 0x00 - 0xff, which has been reserved to hold instruction pointer values. This is necessary for implementing conditional loops. When a loop is entered, the current value in the IP register is stored on the stack at the address pointed to by the stack pointer. The SP module is therefore connected to the same RAM address bus as the datapointer, which means it should go through a tristate buffer to avoid bus contention.

5.7.1 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

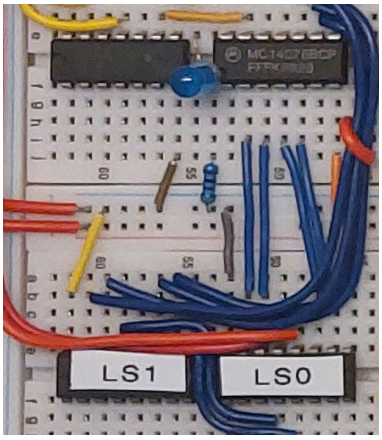
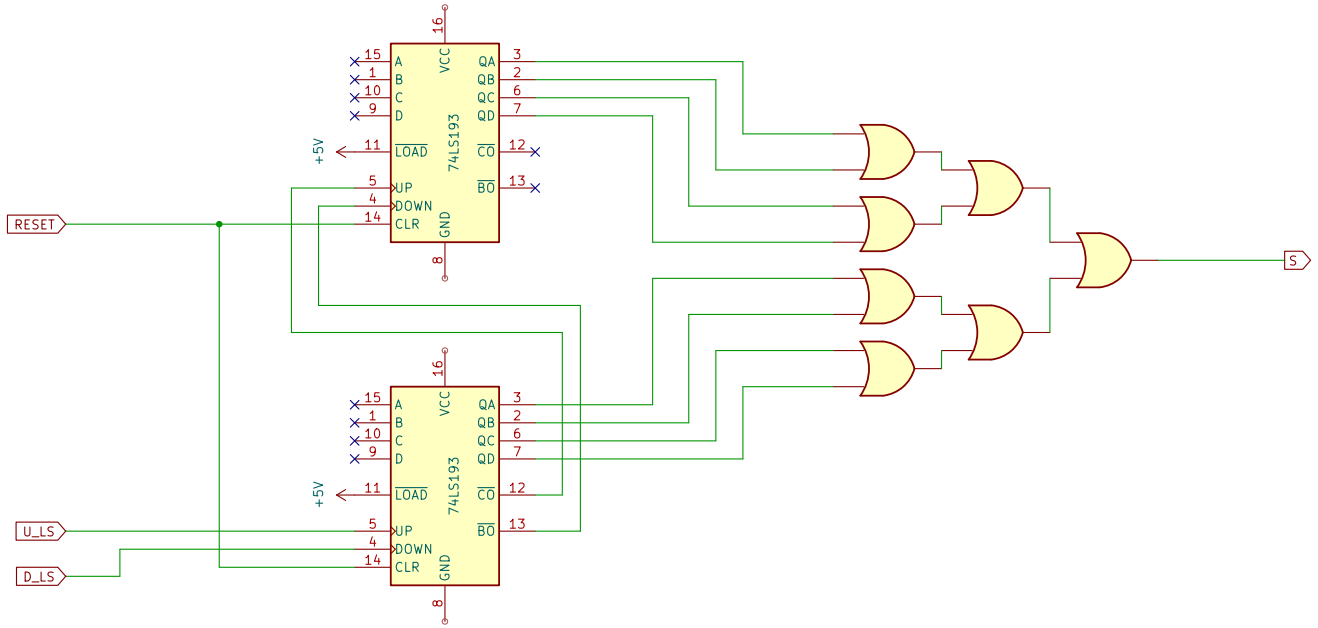


Figure 21: Close up of the Loop Skip Register Module.

The Loop Skip Register (LS) is used to provide the skip flag (S, see 3.6). It is implemented using two 74LS193 binary counters and is connected to the decoders of the Register Driver in order to be able to be selected by the control logic. The S flag is compiled from both register IC's by sending all 8 output-bits to an array of OR-gates; when any of these bits are high, the S flag will be asserted, indicating that the computer is in the process of skipping the current loop.

5.8.1 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

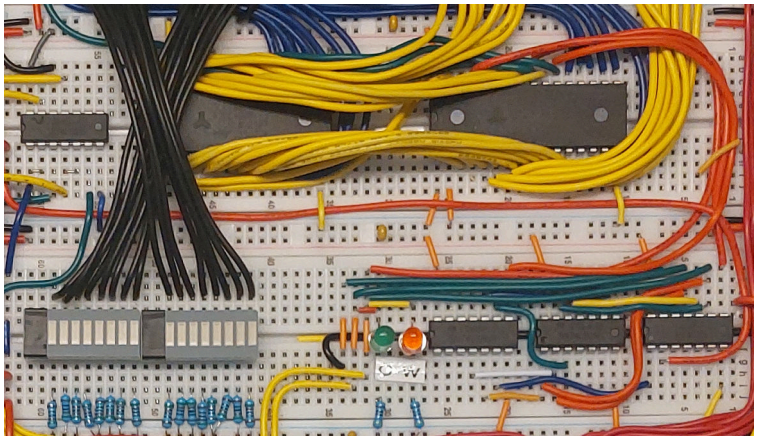


Figure 22: Close up of the RAM Module.

5.9.1 SRAM

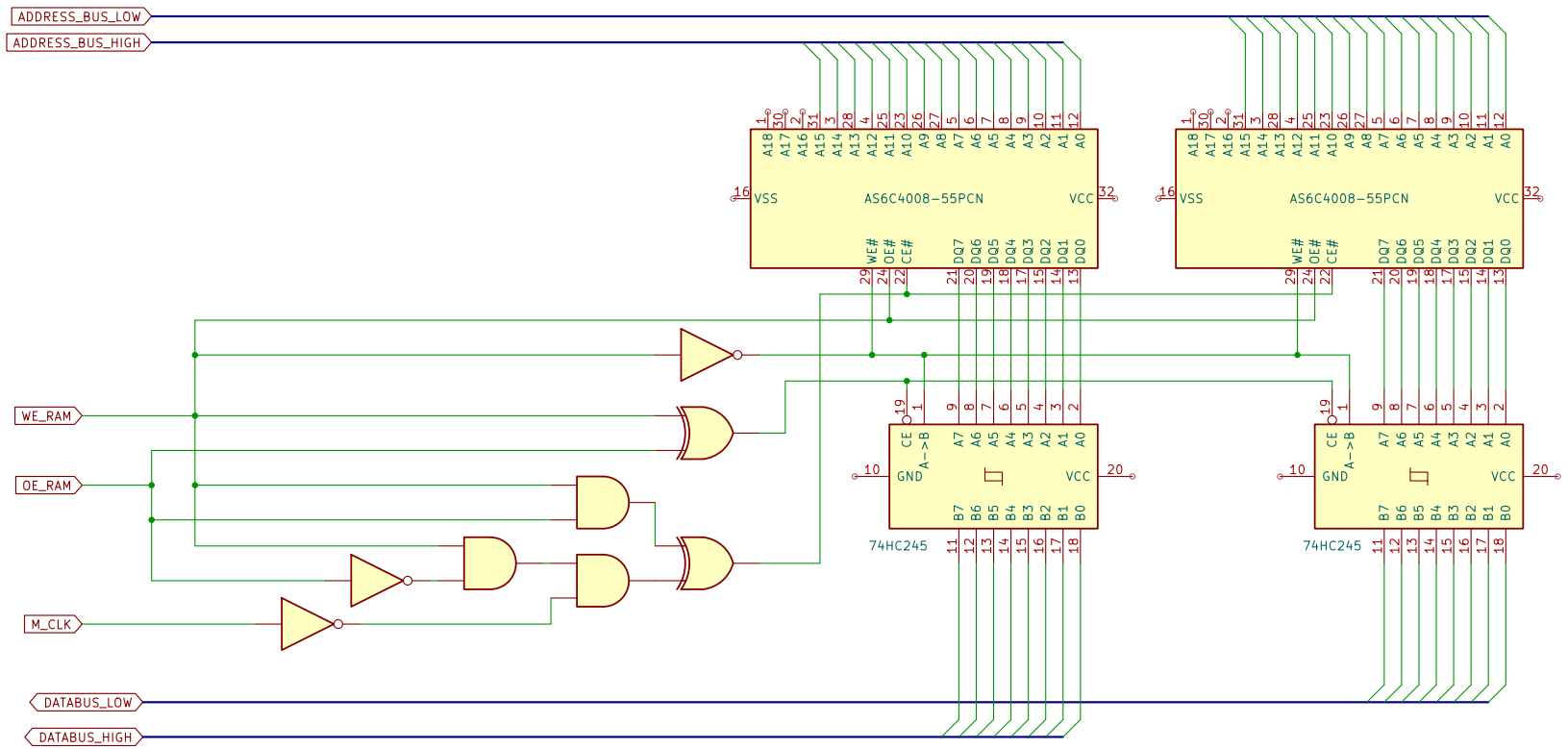
The RAM module is mainly used to store the 8-bit data of the BF memory-tape. However its secondary purpose is to also store the instruction-pointer values when loops are handled, which are 16-bit in size. Therefore the RAM module contains two 512K x 8-bit SRAM chips (AS6C4008) for a total of 512K 16-bit 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.

5.9.2 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.

5.9.3 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

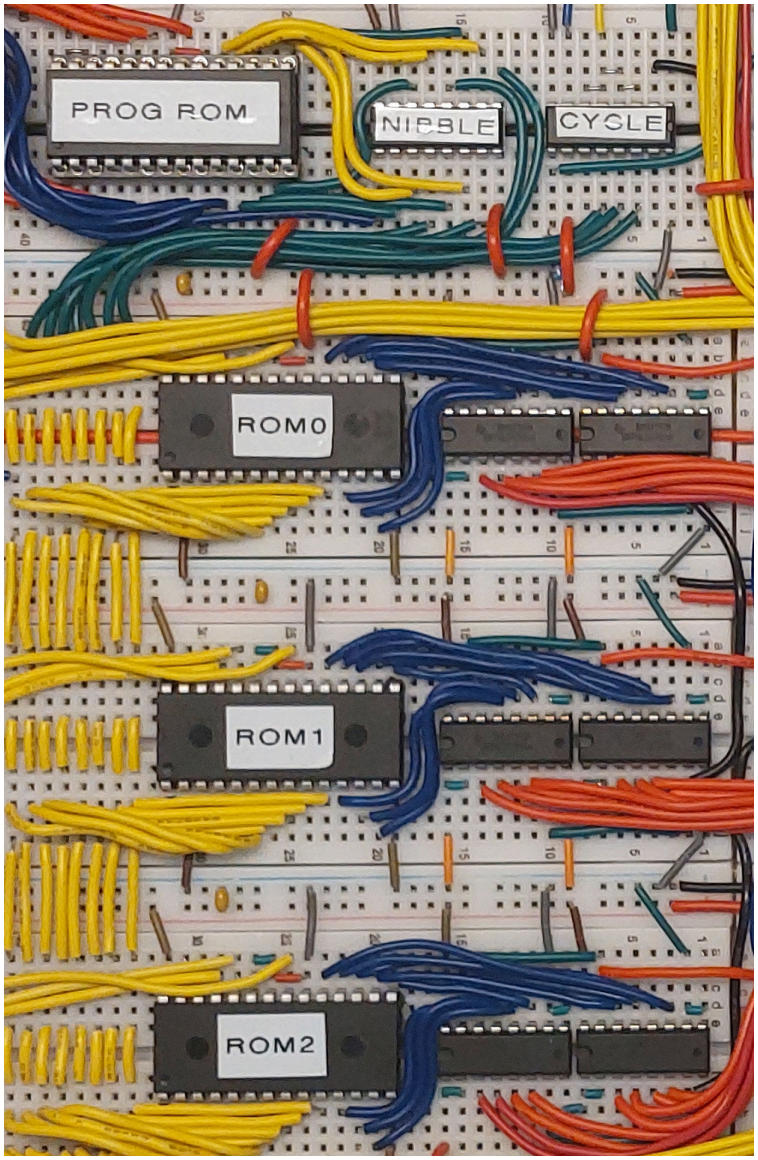


Figure 23: Close up of the Control Unit.

5.10.1 Partitioning

The control unit is responsible for sending the appropriate 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 4 bits: A, V, S and Z) and the cycle count (3 bits) combine together to form an (11 bit) address into a set of three EEPROM chips (AT28C64B), each of which contains part the signal configuration corresponding to that combination of state and instruction. When clocked by the decoder-clock (D_CLK), the values currently at this address are loaded into the registers (74LS173) and asserted onto their respective modules, which will act upon them on the next pulse of the M_CLK signal.

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)
8-11	Flags	(0000_2 - 1111_2)
12-13	Unused	

The three EEPROM's have been programmed using a custom built EEPROM programmer based around an Arduino Nano, combined with a python script (`bflash`) 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 specification for the BFCPU is shown in the listing below and is a direct reflection of the microcode shown in Table 2.

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

[rom] { 8192 x 8 }

[address] {
  cycle:   3
  opcode:  4
  flags:   4
}

[signals] {
  HLT
  RS0
  RS1
  RS2
  INC
  DEC
  DPR
  EN_SP
  OE_RAM
  WE_RAM
  EN_IN
  EN_OUT
  SET_V
  SET_A
  LD_FB
  LD_FA
  EN_IP
  LD_IP
  EN_D
  LD_D
  CR
  ERR
}

[opcodes] {
  NOP       = 0x00
  PLUS      = 0x01
  MINUS     = 0x02
  LEFT      = 0x03
  RIGHT     = 0x04
  IN_BUF    = 0x05
  IN_IM     = 0x06
  OUT       = 0x07
  LOOP_START    = 0x08
  LOOP_END  = 0x09
  INIT      = 0x0d
  HOME      = 0x0e
  HALT      = 0x0f
}


[microcode] {
```

```
54  x:0:xxxx      -> LD_FB
55
56  PLUS:1:x00x       -> INC, RS0, SET_V, LD_FA
57  PLUS:2:x00x       -> INC, RS2, CR
58  PLUS:1:x10x       -> LD_D, OE_RAM, LD_FA, CR
59  PLUS:1:xx1x       -> INC, RS2, CR
60
61  MINUS:1:x00x      -> DEC, RS0, SET_V, LD_FA
62  MINUS:2:x00x      -> INC, RS2, CR
63  MINUS:1:x10x      -> LD_D, OE_RAM, LD_FA, CR
64  MINUS:1:xx1x      -> INC, RS2, CR
65
66  LEFT:1:0x0x       -> DEC, RS1, SET_A, LD_FA
67  LEFT:2:0x0x       -> INC, RS2, CR
68  LEFT:1:1x0x       -> EN_D, WE_RAM, LD_FA, CR
69  LEFT:1:xx1x       -> INC, RS2, CR
70
71  RIGHT:1:0x0x      -> INC, RS1, SET_A, LD_FA
72  RIGHT:2:0x0x      -> INC, RS2, CR
73  RIGHT:1:1x0x      -> EN_D, WE_RAM, LD_FA, CR
74  RIGHT:1:xx1x      -> INC, RS2, CR
75
76  LOOP_START:1:x001 -> INC, RS0, RS2
77  LOOP_START:2:x001 -> INC, RS2, CR
78  LOOP_START:1:x000 -> INC, RS0, RS1
79  LOOP_START:2:x000 -> WE_RAM, EN_SP, EN_IP
80  LOOP_START:3:x000 -> INC, RS2, CR
81  LOOP_START:1:x10x -> LD_D, OE_RAM, LD_FA
82  LOOP_START:2:x10x -> CR # could be in cycle 1?
83  LOOP_START:1:xx1x -> INC, RS0, RS2
84  LOOP_START:2:xx1x -> INC, RS2, CR
85
86  LOOP_END:1:x001   -> DEC, RS0, RS1
87  LOOP_END:2:x001   -> INC, RS0, RS2, CR
88  LOOP_END:1:x000   -> EN_SP, OE_RAM, LD_IP
89  LOOP_END:2:x000   -> INC, RS2, CR
90  LOOP_END:1:x10x   -> OE_RAM, LD_D, LD_FA
91  LOOP_END:2:x10x   -> CR # could be in cycle 1?
92  LOOP_END:1:xx1x   -> DEC, RS0, RS2
93  LOOP_END:2:xx1x   -> INC, RS2, CR
94
95  OUT:1:x00x        -> EN_OUT, EN_D, INC, RS2, CR
96  OUT:1:x10x        -> OE_RAM, LD_D, LD_FA
97  OUT:2:x10x        -> EN_OUT, EN_D, INC, RS2, CR
98  OUT:1:xx1x        -> INC, RS2, CR
99
100 IN_BUF:1:xx0x     -> EN_IN
101 IN_BUF:2:xx0x     -> LD_D
102 IN_BUF:3:xx0x     -> LD_FB
103 IN_BUF:4:xx00     -> SET_V, LD_FA, INC, RS2, CR
104 IN_BUF:4:0x01     -> CR
105 IN_BUF:1:xx1x     -> INC, RS2, CR
106
107 IN_IM:1:xx0x      -> EN_IN
108 IN_IM:2:xx0x      -> LD_D, SET_V, LD_FA
109 IN_IM:3:xx0x      -> INC, RS2, CR
110 IN_IM:1:xx1x      -> INC, RS2, CR
111
112 NOP:1:xxxx        -> INC, RS2, CR
113 HALT:1:xxxx       -> HLT
114 HALT:2:xxxx       -> INC, RS2, CR
115
116 INIT:1:xxx1       -> EN_D, WE_RAM, INC, RS0, RS2
117 INIT:2:xxx1       -> LD_FB, INC, RS1
118 INIT:3:xx01       -> INC, RS2, CR
119 INIT:3:xx11       -> CR
120
121 HOME:1:xxxx       -> DPR, INC, RS2, CR
```

```
122
123   catch              -> ERR, HLT
124 }
```

5.10.2 Program ROM

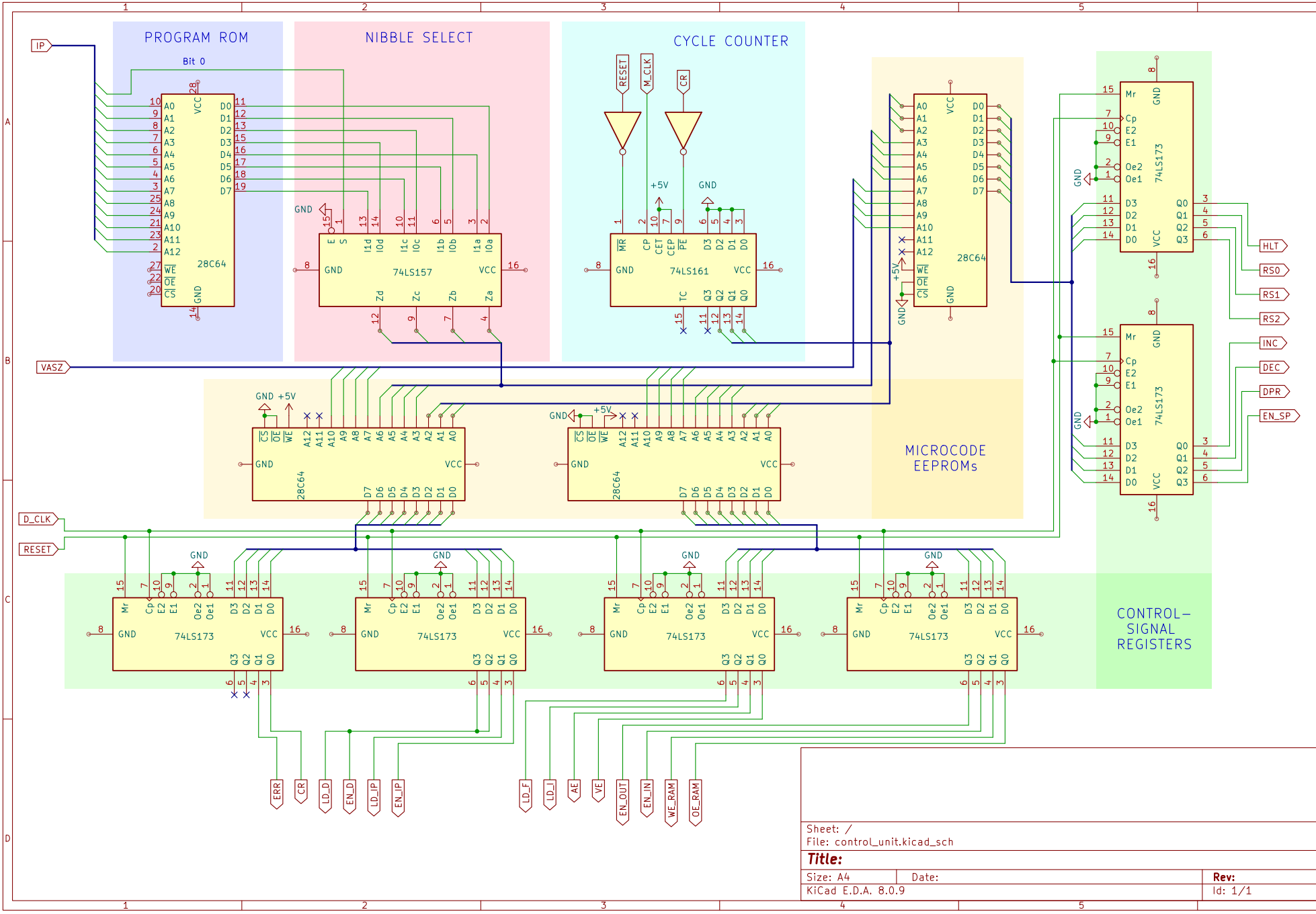
The actual BF program is stored in another 8K EEPROM chip (AT28C64) and is addressed by the instruction pointer as mentioned before. Since each BF instruction only needs 4 bits to be encoded (there are less than 16 different instructions), we can store up to 16K instruction in the chip by packing 2 consecutive instructions together in a single byte (handled by the assembler). 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 allowes 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.

5.10.3 Cycle Counter

The cycle counter is another 74LS161 that simply counts 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 an instruction has completed).

5.10.4 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

The IO system is handled by an ATMEGA328P microprocessor, commonly found in the Arduino Uno. It has three 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. Supply a menu system to alter its settings using two buttons.

5.11.1 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 24). This menu let's the user do the following:

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

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

5.11.2 Handling 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 2 shows the code for the interrupt routine running on the MCU.

5.11.3 Handling 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.

```
98  //----------isr_begin----------
99  void onSystemClock() {
100   ++tickCount;
101
102   enum KeyboardState : uint8_t {
103     IDLE,
104     WAIT,
105     RESET
106   };
107   static volatile KeyboardState kb_state = IDLE;
108
109   if (kb_state == IDLE && digitalRead<DISPLAY_ENABLE_PIN>()) {
110     lcdBuffer.enqueue(readByteFromBus());
111     return;
112   }
113
114   if (kb_state != IDLE || digitalRead<KEYBOARD_ENABLE_PIN>()) {
115     switch (kb_state) {
116       case IDLE:
117         {
118           setIOPinsToOutput();
119           writeByteToBus(kbBuffer.get());
120           kb_state = WAIT;
121           return;
122         }
123       case WAIT:
124         {
125           kb_state = RESET;
126           return;
127         }
128       case RESET:
129         {
130           setIOPinsToInput();
131           kb_state = IDLE;
132           return;
133         }
134     }
135   }
136 }
137 //----------isr_end----------
```

Listing 2: ISR running on the microcontroller to handle the input and output signals.

5.11.4 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.

5.11.5 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.

5.11.6 Keyboard

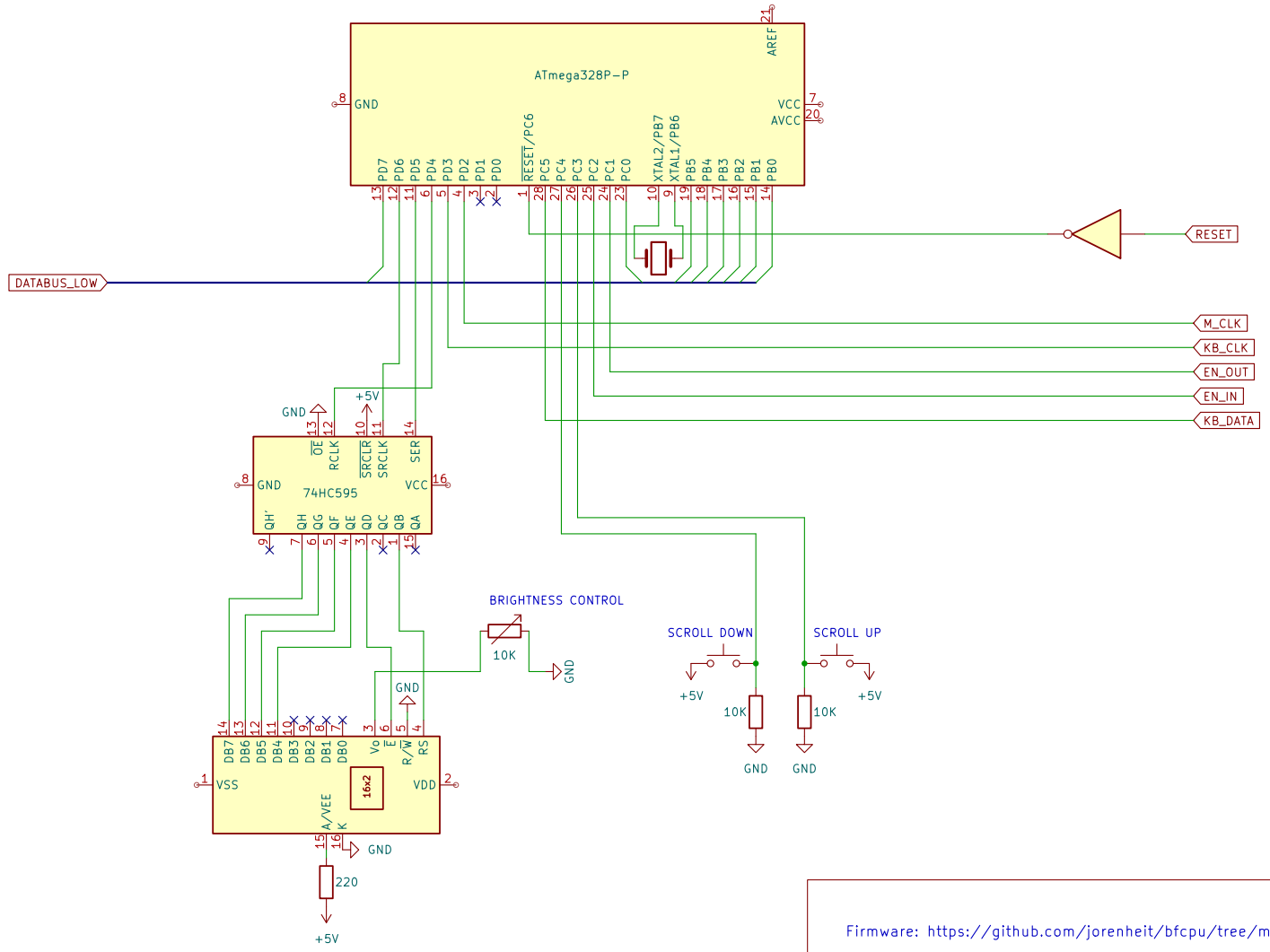
The IO module can only handle input from PS/2 compatible keyboards. A modified version of the `PS2Keyboard` library was used to implement the keyboard driver.

5.11.7 Software

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

5.11.8 Schematic

A full schematic is provided on the next page.



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 4 lists all the available commands and the values that map to these commands. As explained in Section 4, there are a few non-BF that have been added.

Command	Opcode
`NOP`	0x00
`+`	0x01
`-`	0x02
`<`	0x03
`>`	0x04
`,`	0x05
`'`	0x06
`.`	0x07
`[`	0x08
`]`	0x09
`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. 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`.

6.2 Programmer

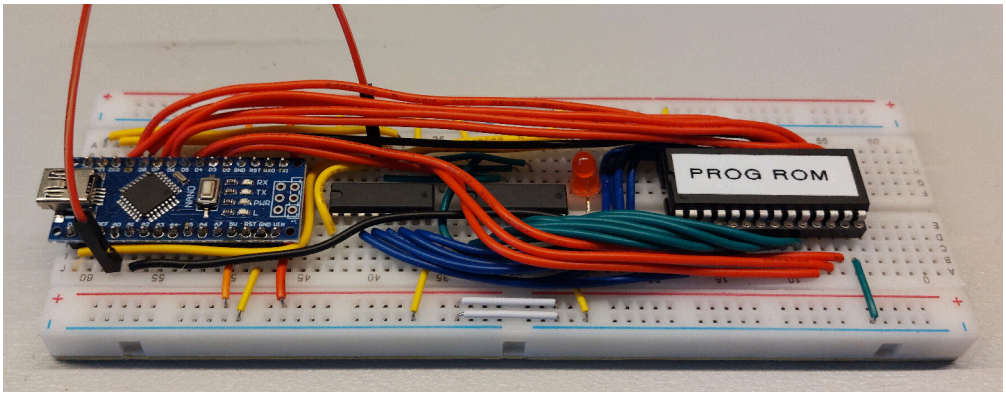
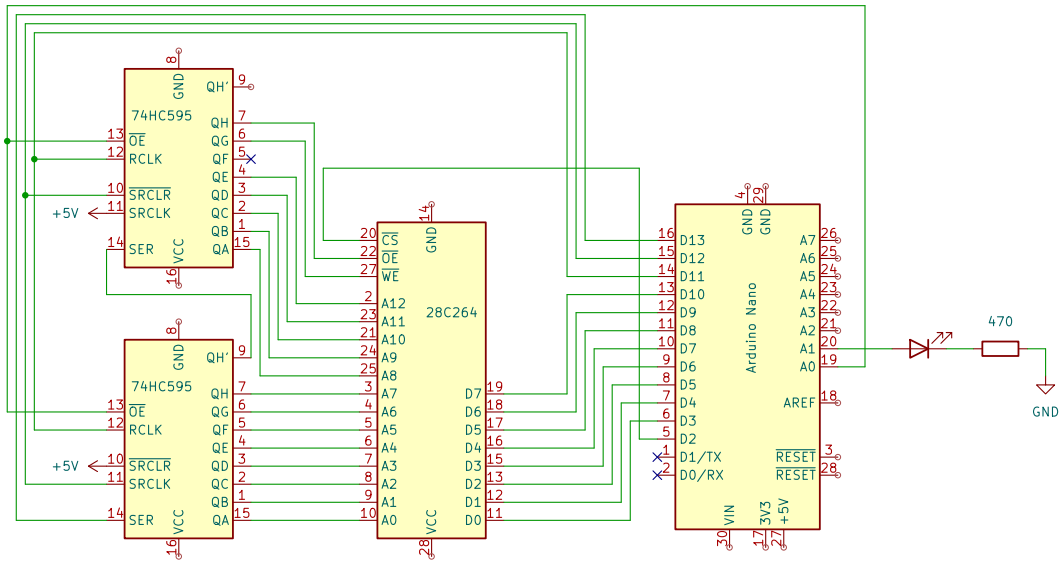


Figure 25: EEPROM chips were programmed using an Arduino Nano on a breadboard.

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 25) 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 5.10 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
}
```

References

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

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

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

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

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

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

7 Conclusion