

Synapse-191

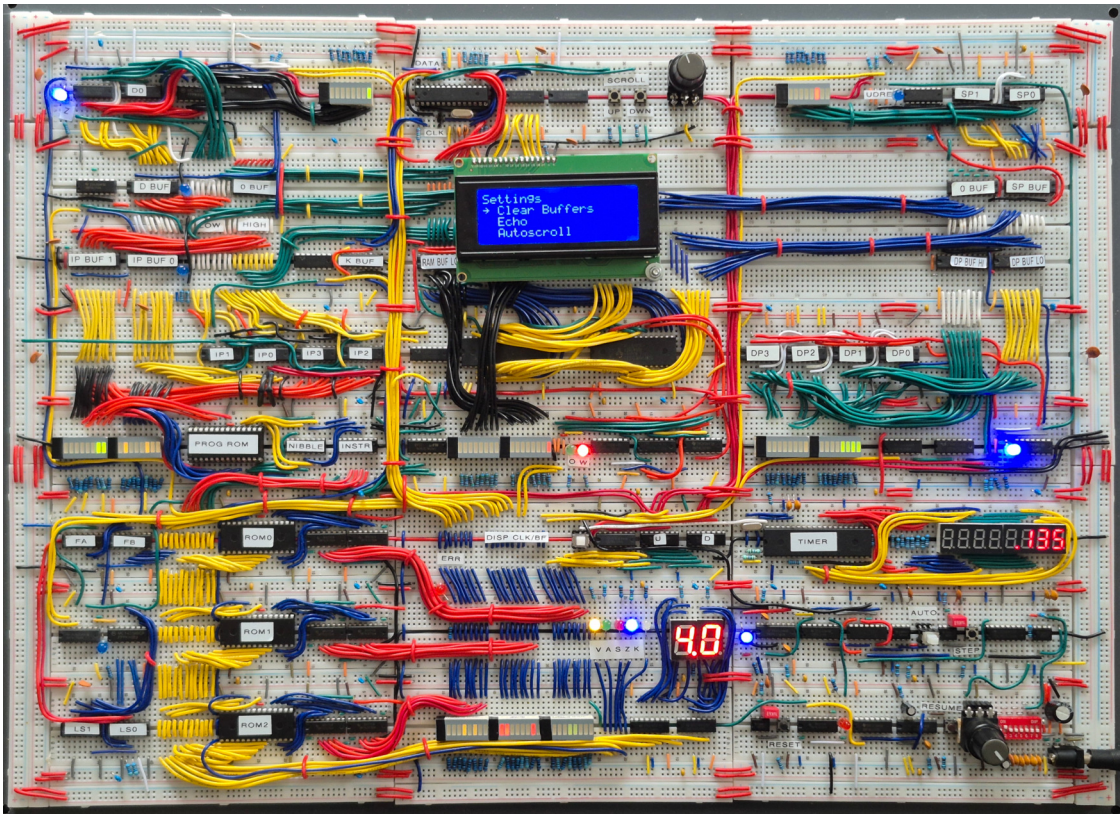
Building a Brainfuck Computing Environment



Joren Heit
2025

Abstract

This report documents the design and implementation of Synapse-191, a fully functional computer built on breadboards to execute the Brainfuck (BF) programming language natively. Unlike traditional approaches that interpret or compile BF code on conventional architectures, Synapse-191 treats BF as its instruction set, realized through a microcoded CPU constructed entirely from TTL logic components, RAM, and ROM—without programmable logic in the core. The system features a Harvard-like architecture, a two-phase clock capable of operating at over 250 kHz, and a modular design comprising registers, memory units, and a control unit driven by EEPROM-based microcode. A dedicated I/O module, powered by an ATmega328P, manages keyboard input, LCD output, and random number generation, providing a user interface for program selection and runtime configuration. Supporting tools—including an assembler, EEPROM programmer, microcode compiler, and emulation framework—enable efficient development and testing. The project demonstrates the feasibility of implementing an esoteric language as a native instruction set and offers insights into computer architecture, timing, and control logic at the hardware level. The system ended up being able to run all programs from a diverse test suite of BF programs, taken verbatim from the internet, without running into any hardware limitations, and can be considered a very capable Brainfuck execution machine, if such a thing even makes sense.



Acknowledgement

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

Thank you, Artur.

Contents

Abstract	ii
Acknowledgement	v
I The Brainfuck Computer	1
1 Introduction	2
1.1 Brainfuck	5
1.1.1 Language Description	5
1.1.2 Interpreters	5
1.1.3 Brainfuck Instruction Set	7
2 Synapse-191 Architecture	8
2.1 Hardware Overview	8
2.2 Flags	10
2.3 Data Pointer Register (DP)	11
2.4 Data Register (D)	12
2.5 Instruction Pointer Register (IP)	12
2.6 Stack Pointer Register (SP)	13
2.7 Loop Skip Register (LS)	13
2.8 Flag Registers (FA and FB)	14
2.9 Instruction Register (I)	14
2.10 Register Driver	15
2.11 Cycle Counter (CC)	16
2.12 Data Memory (RAM)	16
2.13 Program Memory (ROM)	17
2.14 Screen (SCR)	17
2.15 Keyboard (KB)	17
2.16 Control Unit	18
3 Opcodes and Control Sequences	20
3.1 Binary Format	20
3.2 Instruction Decoding	21
3.3 Cycle 0: Instruction Fetch	22
3.4 Modifying Data: + and -	22
3.5 Moving the Pointer: < and >	23
3.6 Conditional Jumping: [and]	24

3.7	Output:	25
3.8	Input: ,	27
3.9	Initialization and Bootloading	28
3.10	Convenience Opcodes	30
4	Hardware Implementation	32
4.1	Wiring	34
4.1.1	Breadboards	34
4.1.2	Wires	34
4.1.3	Busses	34
4.1.4	Bus Pull-Down	34
4.1.5	Power	35
4.1.6	Clock and Reset	35
4.1.7	LED Indicators	35
4.2	Clock and Controls	36
4.2.1	Clock	36
4.2.2	Reset/Resume	38
4.3	Register Driver	40
4.4	DP Register	41
4.5	D Register	43
4.6	IP Register	44
4.7	SP Register	45
4.8	LS Register	46
4.9	Flag Registers	47
4.10	RAM	48
4.11	Control Unit	50
4.11.1	Overview	50
4.11.2	Early Segmented Implementation	51
4.12	IO Module	52
4.12.1	Features	52
4.12.2	System Interactions	53
4.12.3	Shift Register	55
4.12.4	LCD Screen	55
4.12.5	Keyboard	55
4.13	IO Module Firmware	55
4.13.1	Interrupts	56
4.13.2	Ring Buffers	56
4.13.3	Direct Port Access	57
4.13.4	Compile-time Menu Structure	57
5	Supporting Tools	59
5.1	Assembler: <code>bfasm</code>	59
5.1.1	Features	60
5.2	Programmer: <code>bflash</code>	62
5.2.1	Overview	62
5.2.2	Flashing the AT28C64B	62
5.2.3	Shift Register	62
5.3	Microcode Compiler (Mugen)	63

5.3.1	Motivation	63
5.3.2	File Structure	63
5.4	Emulation Framework (Rinku)	64
6	Runtime Results	65
6.1	Stability	65
6.2	Test Suite	66
6.2.1	hello.bf	66
6.2.2	euler.bf	67
6.2.3	phi.bf	68
6.2.4	factorial.bf	68
6.2.5	primes.bf	69
6.2.6	primes2.bf	70
6.2.7	tictactoe.bf	70
II	Brainfuck Compilation	72
7	Introduction to Acus	73
7.1	What This is Not	73
7.2	Motivation and History	73
7.3	Challenges and Design Goals	74
8	Acus Program Model	76
8.1	Basic Block Scheduling	76
8.2	Frames and Initialization	77
8.3	Block Guards	77
8.4	Example: Fibonacci	78
9	Tape Abstractions	80
9.1	Overview	80
9.2	Cells and Macro-Cells	81
9.2.1	Value0 and Value1	82
9.2.2	Scratch0 and Scratch1	82
9.2.3	Flag	82
9.2.4	FrameMarker	82
9.2.5	SeekMarker	83
9.2.6	Payload0 and Payload1	84
9.3	Slots	84
9.3.1	Data Types	84
9.3.2	Slot Proxies and Materialization	85
9.3.3	Local Memory Management	86
9.4	Frames and Calls	87
9.4.1	Reference Frames	87
9.4.2	Global Data Frame	87
9.4.3	Frame Layout	87
9.4.4	Calls, Returns and Meta-Blocks	88

10	Memory Management	90
10.1	Overview	90
10.1.1	Relative Offsets	90
10.1.2	Free vs Clear	90
10.1.3	Heap Allocation	91
10.2	Splitting and Merging	91
10.2.1	Slot Splitting	91
10.2.2	Slot Merging	91
11	Code Generation	93
11.1	Clearing, Moving and Copying Values	93
11.1.1	Clearing Cells	93
11.1.2	Moving Data	93
11.1.3	Copying Data	93
11.2	Constant Factories	94
11.3	Addition and Subtraction	95
11.3.1	Addition (8-bit)	95
11.3.2	Addition (16-bit)	96
11.3.3	Subtraction	96
11.3.4	Signed Integers	96
11.4	Multiplication and Division	97
11.4.1	Multiplication (8-bit)	97
11.4.2	Multiplication (16-bit)	97
11.4.3	Division (8-bit and 16-bit)	98
11.4.4	Signed Multiplication and Division	99
11.4.5	Division by Zero	99
11.5	Boolean Operations	99
11.5.1	Conversion to Bool	99
11.5.2	Logical Not	100
11.5.3	Logical And	100
11.5.4	Logical Or	100
11.5.5	Equality	101
11.5.6	Less Than	101
11.5.7	Signed Comparisons	101
11.6	Pointer Movement	102
11.6.1	Static Pointer Movement	102
11.6.2	Dynamic Pointer Movement	102
11.6.3	Payload Transport	104
12	Caching	105
12.1	Cache Graphs	105
12.2	Flushing and Invalidation	106
12.2.1	Dirty Entries	106
12.2.2	Flushing	106
12.2.3	Invalidation	107
12.2.4	Writes	107
12.2.5	Example	107

13 Other Optimizations	109
13.1 Move Semantics	109
13.2 Literal Evaluation	109
13.2.1 Constant Folding	109
13.2.2 Specialized Constant Algorithms	109
13.3 Primitive and BF Merging	110
13.3.1 Merging Primitives	110
13.3.2 Merging BF	112
 III Reflection	 113
14 Reflection	114
14.1 Conclusion	114
14.2 Final Thoughts	115
14.2.1 Virtual Registers	115
14.2.2 No Register Driver	116
14.2.3 Designing a PCB	116
14.2.4 Going Full 16-bit	116
14.2.5 Using the '191	116
14.2.6 Decoupling Capacitors	117
14.2.7 Using AI	117
14.2.8 IO Module: Cheating?	117
14.2.9 Higher Clock Speeds	117
 A Microcode Table	 119
 B Mugen Specification	 122
 C Bill of Materials	 127
C.1 Integrated Circuits	128
C.2 Passive Components	129
C.3 Other Parts	129
 D BF Test Suite	 130
D.1 Hello World	130
D.2 Factorial	130
D.3 Euler's Number	131
D.4 Golden Ratio	131
D.5 Primes	132
D.6 Primes 2	132
D.7 Tic Tac Toe	133
 E IO Module ISR	 134
 F Constant Factory Lookup Table	 136
 G Schematics	 137

Part I

The Brainfuck Computer

Chapter 1

Introduction

The Brainfuck (BF) programming language is an esoteric programming language that is essentially impossible—or at least highly impractical—to actually write useful programs in. Even if you became a very skilled programmer in this language, the resulting programs would be incredibly slow to execute. Despite this, many programmers have challenged themselves to write stunning pieces of code just for fun or for the learning experience it offers. Doing so teaches us about computer architecture, compilers/interpreters, memory, pointers and much more. For more information on the language itself, see Section 1.1.

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

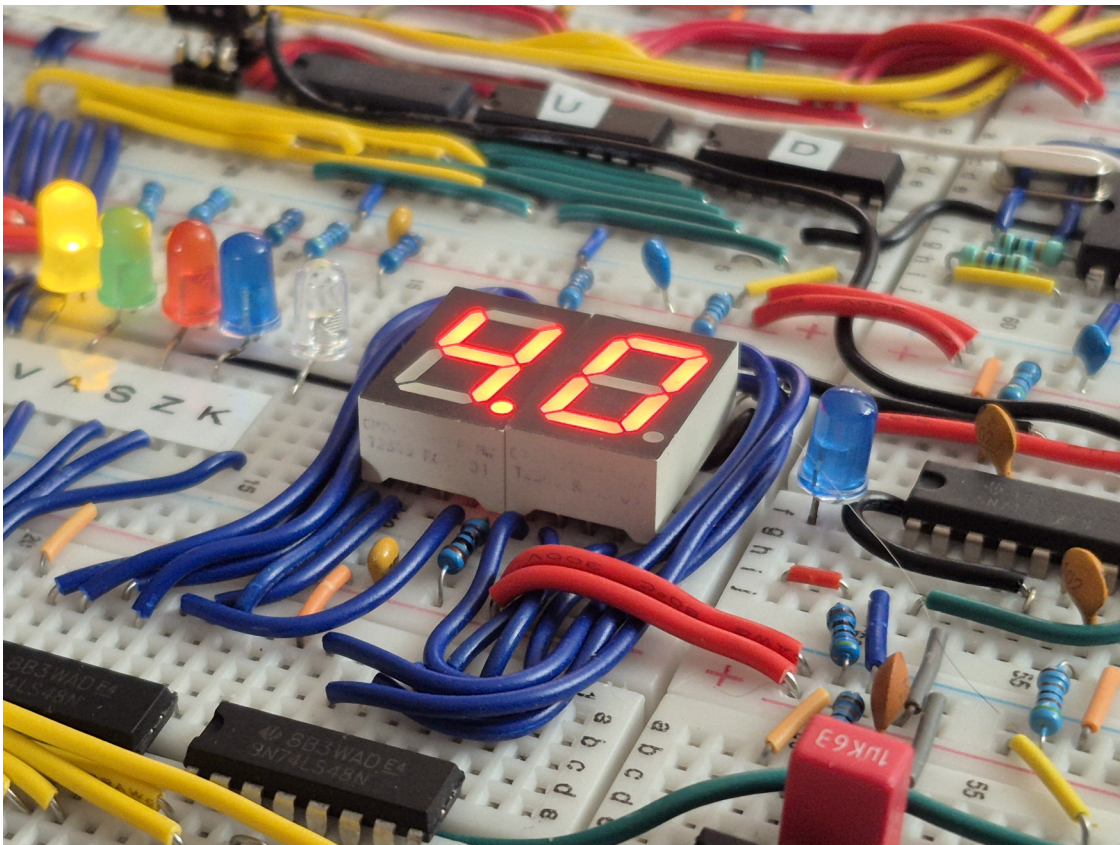
Our goal was to build this BF-CPU without making use of any programmable chips, relying solely on Transistor-Transistor-Logic (TTL) chips such as registers, buffers and (de)multiplexers in addition to the necessary RAM and ROM. The computer was to be implemented entirely on breadboards, as it was inspired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

Outcome Over roughly 2.5 years (intermittently), the design evolved into a stable microcoded CPU with a Harvard-like memory map, a two-phase clock that can drive the system at over 250 kHz, 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 input (both random numbers and keyboard input) and output (to a 20×4 character LCD display). This IO module is driven by an ATmega328P to be able to manage IO buffers, implement the PS/2 protocol and drive the screen, in addition to managing all the IO settings (echo, autoscroll, output modes, etc.) and providing a user interface for selecting the active program at runtime. A deliberate choice was made to use a programmable chip in this case, since it is not really part of the CPU itself and makes the IO capabilities a lot more advanced and convenient. To the CPU core, the outside world is a black box that accepts certain control signals and acts accordingly, similar to how the BF input and output commands (`.` and `,`) interact with an abstract external world.

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

Document structure

- Section 1.1 recaps the BF programming language: how does this language work, what does a simple interpreter look like and how does it relate to the architecture of Synapse-191?
- Chapter 2 describes the architecture from a modular perspective: what is the purpose of each of the high-level modules, what control signals do they accept and what actions do they perform when clocked?
- Chapter 3 explains microcode and control sequences; how do all these signals and modules work together to perform the computations necessary to run BF programs?
- Chapter 4 discusses hardware implementation: what kinds of chips and techniques were used to implement the modules on a hardware level?
- Chapter 5 covers supporting utilities, like the assembler, microcode compiler, EEPROM programming utilities, etc.
- Chapter 6 goes through a number of test programs that were used during development.
- Chapter 7 introduces Acus, the higher-level compiler targeting BF.
- Chapter 14 lists a number of possible improvements that could still be implemented in the future, as well as some more general final thoughts.
- We conclude in Section 14.1 with a brief retrospective on the project.



1.1 Brainfuck

1.1.1 Language Description

Brainfuck is a popular esoteric programming language. Just like any other programming language, it allows the programmer to write programs consisting of commands that are executed in order. The key limitation is that the language provides only eight commands to the programmer, all written as a single character: “+-<>[].,”. Each of these commands corresponds to an operation on an array of memory or a pointer, pointing to some location within this memory. At the start of the program, every cell in (an infinite amount of) memory is initialized to 0 and the pointer is pointing to the very first element (index 0, see Figure 1.1).

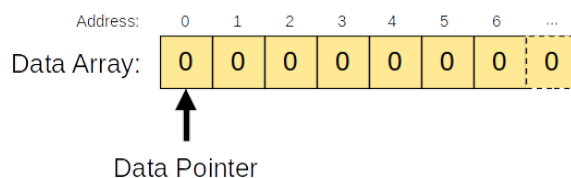


Figure 1.1: Initial state of the BF machine.

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

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

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

1.1.2 Interpreters

To run a BF program, one usually feeds these commands into an interpreter written in a more common language. These interpreters are very straightforward to write. Listing 1 shows a very basic implementation (about 40 lines) of a BF interpreter written in C. This implementation initializes a

block of memory to zero and defines a pointer to its first element. This pointer can be incremented or decremented to move along the array, increment or decrement the value it points to, or print it to the standard output. Most of its complexity is embedded in the handling of the loop operators. When an opening bracket is encountered, the index of this instruction is stored in a jump table. When control reaches its matching closing bracket and the value of the cell pointed to is nonzero, this stored index is reloaded in order to return to the start of the loop. When skipping a loop, i.e. the current cell holds a zero when the opening bracket is evaluated, all commands up to and including the matching closing bracket are skipped.

```

16 //-----bfint_begin-----
17 void bfint(char const *program) {
18     unsigned char mem[MEM_SIZE];
19     memset(mem, 0, MEM_SIZE);
20     unsigned char *ptr = mem;
21
22     int jmp_table[JMP_TABLE_SIZE];
23     int jmp_index = 0;
24
25     int program_size = strlen(program);
26     int index = 0;
27
28     while (index < program_size) {
29         switch (program[index]) {
30             case '+': ++(*ptr); break;
31             case '-': --(*ptr); break;
32             case '<': --ptr; break;
33             case '>': ++ptr; break;
34             case '.': putchar(*ptr); break;
35             case ',': (*ptr) = getchar(); break;
36             case '[': {
37                 if (*ptr) jmp_table[jmp_index++] = index;
38                 else {
39                     int count = 1;
40                     while (count != 0) {
41                         switch (program[++index]) {
42                             case '[': ++count; break;
43                             case ']': --count; break;
44                         }
45                     }
46                 }
47                 break;
48             }
49             case ']': {
50                 --jmp_index;
51                 if (*ptr) index = jmp_table[jmp_index++];
52                 break;
53             }
54             ++index;
55         }
56     }
57 //-----bfint_end-----

```

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

When the Hello World program from Wikipedia [1] is fed into the function of Listing 1, it prints

out the string `Hello World!`, as expected.

```

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

```

1.1.3 Brainfuck Instruction Set

BISC 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; let's refer to this as a Brainfuck Instruction Set Computer (BISC). A true BISC implementation would operate on the basis of only the 8 aforementioned instructions, which is truly tiny compared to more conventional instruction sets such as those implemented by modern processors or even microcontrollers and older 8-bit systems. Broadly speaking, Complex Instruction Set Computers (CISC) are designed to do as much work as possible in the least number of clock cycles, whereas Reduced Instruction Set Computers (RISC) focus on having a small instruction set with basic operations. For comparison, the x86 instruction set is massive with over 2000 instructions implemented in hardware (depending on the way you count, [6]), whereas RISC-V processors have a fixed opcode width of only 7 bits, allowing for a maximum of 128 different opcodes [7]. BISC is therefore tiny even compared to the smallest instruction sets in use today. While compact, such a small instruction set inevitably leads to less efficient execution; a smaller number of instructions simply means you need more of them to perform meaningful computations, which is reflected by the fact that complex BF programs are typically very large in size.

Harvard Architecture Modern computers are built according to the von Neumann architecture [4], which specifies a CPU (containing registers and an ALU), a single unit of memory and input/output devices. In this architecture, both the program's instructions and data are stored in RAM. This allows programs to change their own code at runtime (self-modifying code) because the program itself is data in a sense. This model does *not* naturally fit the Brainfuck model, which has a strict separation between instructions and data; the data pointer can never point to the instructions that govern it. This is why a Harvard architecture was adopted. In the Harvard architecture, two kinds of memory exist: program memory containing the instructions—stored in read-only memory (ROM)—and data memory—stored in random-access memory (RAM)—containing the modifiable data (Figure 1.2).

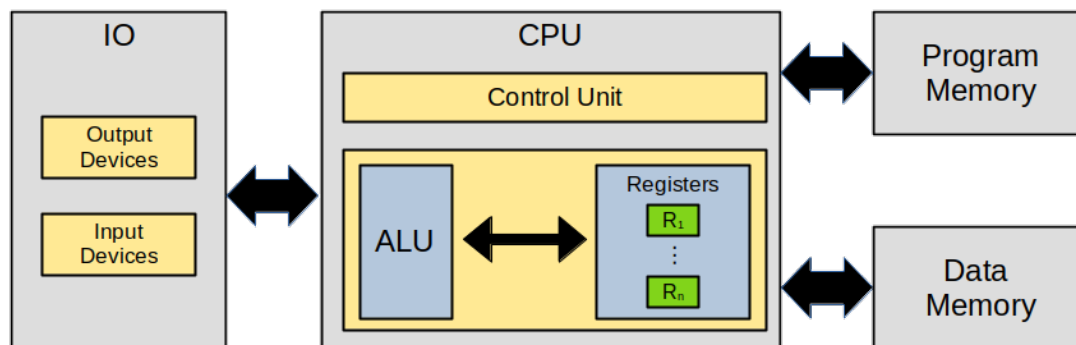


Figure 1.2: Schematic overview of the Harvard architecture.

Chapter 2

Synapse-191 Architecture

2.1 Hardware Overview

A true BISC implementation is a direct representation of the abstract BF machine described in Section 1.1, consisting 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. Figure 2.1 illustrates an example intermediate state of such a system. Consider the BF program “>>>>+.”, applied to the initial conditions shown in the example. The pointer would take 5 steps to the right, landing on cell 9 which contains the number 41. It will then increment and output this value, displaying 42 on the screen (assuming a screen of some sort is used as the output device and it is displaying numbers directly rather than interpreting them as ASCII).

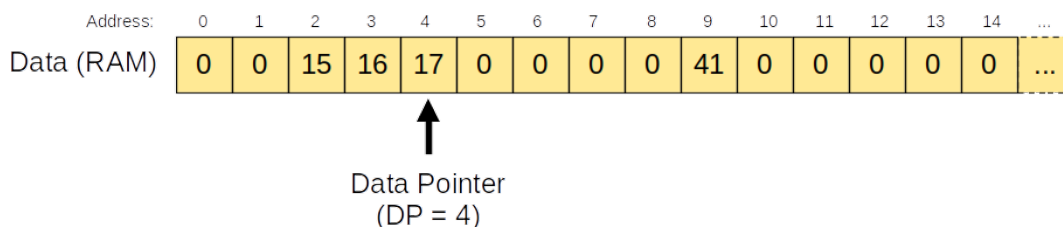


Figure 2.1: Example state of a BF machine.

In a physical system, the data-tape would be a large memory space in RAM and the data pointer is a register that holds a value representing an address within that space. The pointer is moved by simply changing the value stored in the data pointer register. Modifying values in regular systems happens inside the ALU (Arithmetic and Logic Unit), which is capable of addition, subtraction, multiplication and many more mathematical and logical operations. No full-blown ALU is needed in a BISC implementation because of the simplicity of the modifying operations: values only need to be incremented or decremented (this goes for both the data pointer and the data itself), which can be done directly at the register level when up/down binary counters like the 74LS193 integrated circuit are used. The program (a sequence of BF instructions) is stored in Read-Only Memory (ROM), whereas the data is stored in Random Access Memory (RAM). Instructions (4-bit) are fed from ROM into a control unit (CU) together with a set of flags (see 2.2) 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 (memory, registers and I/O) in order for the system to perform the appropriate actions. Table 2.1 lists the components of this BISC

implementation, whereas Figure 2.2 shows the (conceptual) connections between those modules. In the sections below, each of these connections will be clarified further. The actual implementation on the logic/hardware level is described in Chapter 4.

Full Name	Short Name	Description
Data Memory	RAM	Represents the data tape in the BF model; this is where the data is stored.
Program Memory	ROM	Separate from RAM, which is where the program's instructions are stored.
Data Pointer Register	DP	16-Bit counting register that represents the pointer in the BF model; points to data in RAM.
Data Register	D	8-Bit counting register that represents the value currently pointed to by the DP.
Instruction Pointer Register	IP	16-Bit Counting register that contains the address of the current instruction; points to data in ROM.
Stack Pointer Register	SP	8-Bit counting register that holds a value pointing inside the stack-space in RAM; used to implement loops.
Loop Skip Register	LS	8-Bit counting register, used to skip over loops.
Flag Registers A & B	FA/FB	4-Bit registers used to buffer flag values that encode the state of the system.
Instruction Register	I	4-Bit register that buffers the currently active instruction.
Control Unit	CU	System that translates the current instruction into sequences of control signals.
Register Driver	RD	Subsystem of the CU that centralizes increment and decrement operations.

Table 2.1: Overview of the modules that make up the physical BISC implementation.

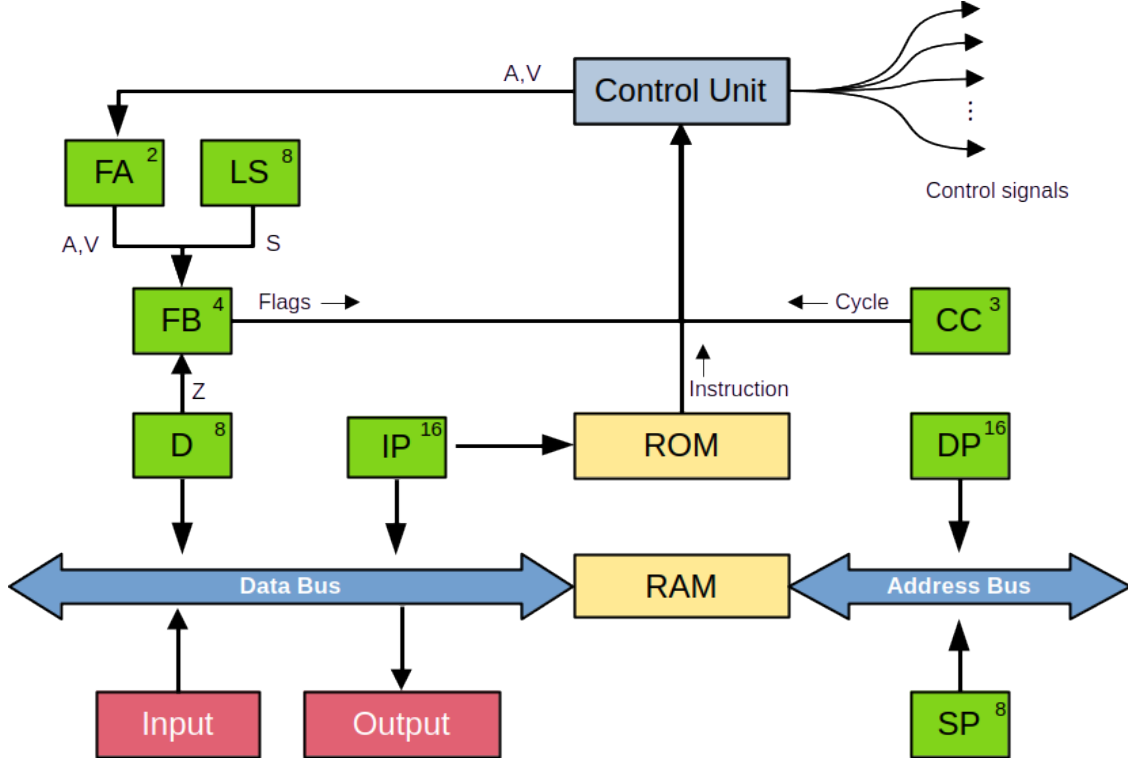


Figure 2.2: Connections between modules in the BF processor.

2.2 Flags

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

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

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

V-Flag The V-flag allows for another optimization, similar to that of the A-flag. Whenever the value in the D register has been modified, the V-flag is set. When the DP is about to change its value, through the < and > commands, the V-flag is checked. If set, we store the modified value back into RAM before moving the pointer. If not, the value hasn't changed and does not

need to be synchronized into RAM. The number of cycles to complete long sequences of pointer move-commands is decreased substantially using this approach.

S-Flag The S-flag is used to skip over blocks of code, primarily when skipping over loop-blocks. It is set when a loop should not be entered, causing all following commands to be ignored until the matching closing bracket is reached. Similarly, it is used at boot to skip over entire programs until the selected program is reached (when multiple BF programs have been written to the same program ROM).

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

Performance Gains The number of cycles saved from using the A and V flags depends on the program that is running; programs with long and frequent sequences of `+`, `-`, `<` or `>` commands will benefit more than those with fewer or shorter sequences. To investigate the actual impact, a test suite of BF programs was put together and run in an emulator (a modified version of the C example shown in Listing 1). The BF source for each of these programs is shown in Appendix D. The modified interpreter contains a table of the number of cycles necessary to execute each instruction, depending on the aforementioned flags. This allows for a quantitative comparison between the naive implementation (load, modify, store) and the cache-like approach using the A and V flags. The results are shown in Figure 2.3; a 16 to 22% decrease was observed when using the A and V flags.

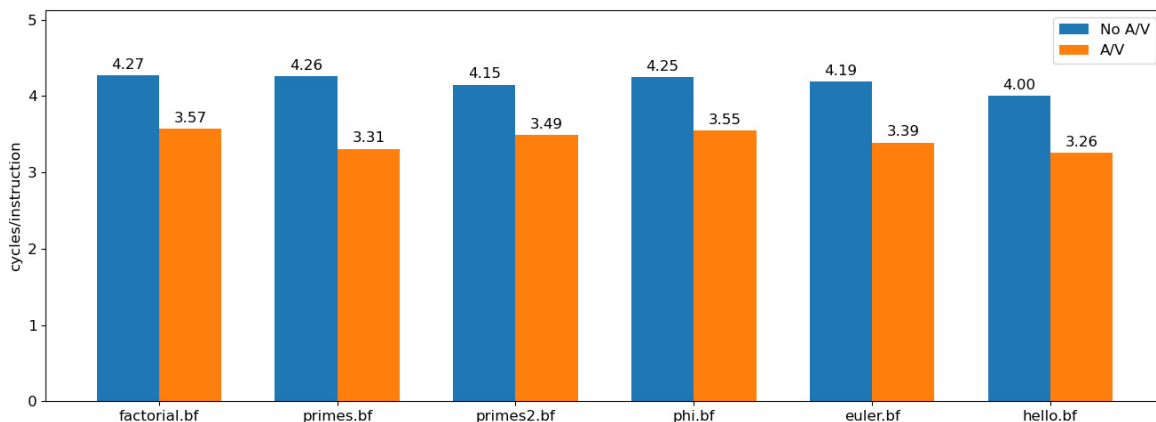


Figure 2.3: The number of cycles per BF instruction drops by around 20% when using the A and V flags.

2.3 Data Pointer Register (DP)

The data-pointer corresponds to the pointer as specified in the BF-language. It points to some value in memory beyond the stack ($\geq 0x0100$, see 2.6) and can be either incremented (moved right) or decremented (moved left) using the `>` and `<` instructions. Whenever the value pointed to by DP is modified by `+` or `-`, it is loaded into the D-register (see 2.4), where it can be modified before being stored back into RAM.

Inputs

The DP should be able to increment and decrement (corresponding to the < and > commands), and should be able to be enabled/disabled because of its connection to the address bus of the RAM (the Stack Pointer (SP; see 2.6) is also connected to this bus). While all other modules have the ability to be reset, the DP is the only register that can be reset (to 0x0100) at runtime. This is necessary during boot, when all the data cells need to be initialized to 0 (see 3.9).

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

Outputs

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

2.4 Data Register (D)

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

Inputs

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

Outputs

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

2.5 Instruction Pointer Register (IP)

The IP Register stores the instruction pointer (16-bits), which keeps track of the instruction that is currently being executed. It points to a certain address in ROM (which stores the program) and is usually incremented after each instruction has finished executing, in order to move to the next instruction. However, when the processor encounters the [-instruction (and a loop is entered), its value is stored in RAM at the location pointed to by the stack pointer (SP, see 2.6). When the matching]-instruction is encountered, this value can be loaded back into the IP in order to jump back to the start of the loop if needed.

Inputs

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

Outputs

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

2.6 Stack Pointer Register (SP)

The stack is the first part of RAM (addresses 0x0000 - 0x00FF) and is reserved to keep track of addresses in ROM that might need to be jumped to when flow encounters a loop-end instruction (I). 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 the SP overflows (wraps around back to 0) and starts overwriting previous values. This would happen if a BF program was loaded that has more than 256 nested []-pairs (and each of those loops is entered). Although possible, it is very unlikely to happen for the simple programs we intend to run. In Chapter 6, where the required stack size is measured for a suite of test programs, this is confirmed; the largest value found for any required stack size in these programs was only 17, leaving plenty of space for even more complex programs.

Inputs

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

Outputs

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

2.7 Loop Skip Register (LS)

The Loop Skip (LS) register is a counter that indicates whether or not we're in the process of skipping a loop. In BF, a loop (I) 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 (J). When it is determined that a loop must be skipped (based on the Z-flag provided by the D-register), the LS register is incremented from 0 to 1 and the S-flag is set. This flag will remain set as long as the value in LS is nonzero, indicating that the system is in a skip-state. Subsequent instructions are then skipped until either another (nested) loop-start or a closing loop-end is encountered. On the former, the LS is incremented again while on the latter the LS is decremented. This has the effect

that the LS becomes 0 again after the] that matches the original [which led to the skip. Normal execution occurs as soon as LS has become 0 again and the S-flag is reset back to 0.

Inputs

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

Outputs

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

2.8 Flag Registers (FA and FB)

FA The first flag register (FA) holds two flag values, A and V, which are used to indicate that either the address (A) or value (V) has changed during one of the previous instructions. For instance, if D was incremented (or decremented), the V-flag is set to indicate a change of the *value* being pointed to: the value in RAM is now outdated. When DP is incremented (or decremented), the A-flag is set to indicate a change of the current *address*, meaning that the value in D is now outdated. For a more detailed description of the function and application of these flags, refer to Sections 3.4 and 3.5.

FB On the zeroth cycle of every instruction, these flags are latched into the FB register together with the Z and S-flags (set by the D and LS registers) for a total of 4 flags. This happens simultaneously with loading the next instruction into the instruction register (I, see 2.9) and is not refreshed until loading the next instruction to make sure that the flag-state remains constant throughout the execution of the current instruction.

K-Flag The previously mentioned K-flag is specific to interactions with the IO-module and is not buffered in either of the flag registers. It is discussed more thoroughly in Sections 3.7 and 3.8.

Inputs

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

Outputs

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

2.9 Instruction Register (I)

The instruction register I buffers the current instruction pointed to by the IP. The instruction is loaded from program ROM into I at the start of every new instruction (cycle 0), right after IP has

been incremented. Its outputs are used as part of the microcode address that goes into the decoder of the CU (see Figure 3.2 in Chapter 3).

Inputs

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

Outputs

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

2.10 Register Driver

Rather than having a separate signal for each of the INC/DEC inputs of each register (e.g. INC_D, INC_LS, etc.), a driver module was designed (see 4.3) to drive registers that support modification of their contents (increment/decrement). In addition to a universal INC/DEC signal, three Register Select (RS) bits are used to index the target register. This approach has two advantages:

1. It decreases the number 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 2.2 contains an overview of each of the registers and the control signals they support. When all RS-signals are off, no register is selected.

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

Table 2.2: Control signals available on each of the registers. The flag and instruction registers are not connected to the register driver.

Inputs

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

Outputs

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

2.11 Cycle Counter (CC)

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

Inputs

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

Outputs

1. CC_OUT - 3 bits - Current value of the register (0–7).

2.12 Data Memory (RAM)

RAM is divided into two parts: stack and data. The first 256 bytes (0x0000 - 0x00FF) make up the stack and are indexed by the stack pointer (2.6). The data (corresponding to the BF tape) is stored at addresses 0x0100 through 0xFFFF and is indexed by the data pointer (2.3). Its address lines are connected to the address bus, which in turn receives its value from either the DP or the SP. Its data lines are bidirectional and are connected to the data bus. When the Write Enable (WE) signal is active, data can be read from the bus and written into RAM. When instead the Output Enable (OE) signal is active, the current value in RAM (determined by the address on the address bus) is asserted onto the data bus.

Inputs

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

Outputs

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

2.13 Program Memory (ROM)

The actual BF instructions are stored in Read-Only-Memory (ROM) and are addressed by the IP (2.5). A 4-bit instruction is stored at the address pointed to by the IP. It is sent to the CU where it is used to determine the set of control signals, together with the flags and cycle counter.

Inputs

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

Outputs

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

2.14 Screen (SCR)

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

Inputs

1. DATA_IN - 8 bits - connected to the data bus.
2. EN: Enable - Display the contents of the bus. The format of the output (ASCII, hex, etc) may vary depending on the implementation of the output device.
3. CLR_K - Clears the K-flag - connected to the CU (this signal is shared with the input module, KB; see 2.15).

Outputs

None.

2.15 Keyboard (KB)

The input device to the computer is assumed to be a keyboard of some sort¹ that implements a buffer from which some 8-bit value can be requested. The control unit can assert the enable signal of this device and should then wait until the K-flag is set, at which point the data should be ready to be read from the data bus. It is left up to the implementation of the peripheral to decide what to do when there is nothing in the input buffer (either wait for user input or return 0). The K-flag, which is shared with the output peripheral, is reset to communicate that the data has been transferred and the input device can yield control of the data bus back to the system.

¹At 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 Brainfuck extension, where the ?-command is used to generate a random number in the currently active cell.

Inputs

1. EN - Enable - Make the contents of the input buffer available on the data bus.
2. CLR_K - Clears the K-flag - connected to the CU (this signal is shared with the output module, SCR; see 2.14).

Outputs

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

2.16 Control Unit

Each of the aforementioned components/modules has one or more control inputs that determine what happens on the next clock cycle. For example, some register-modules can be told to load a value from their input, increment or decrement the currently stored value, or do nothing at all. It is the Control Unit (CU) that supplies the appropriate control signals to each of the modules before the next clock pulse occurs, depending on the current instruction and state determined by the flags and cycle counter. The implementation details of how this is done in hardware are discussed in Chapter 4.

Inputs

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

Outputs

1. HLT - Halt Clock
2. RS0 - Register Select, bit 0
3. RS1 - Register Select, bit 1
4. RS2 - Register Select, bit 2
5. INC - Increment selected register
6. DEC - Decrement selected register
7. CLR_DP - Reset DP
8. EN_SP - Enable SP to address bus ²
9. OE_RAM - Output Enable RAM
10. WE_RAM - Write Enable RAM
11. EN_IN - Enable input (keyboard) to data bus
12. EN_OUT - Enable output device
13. SET_V - Set V-flag in FA

²Note that there is no EN_DP since this signal is mutually exclusive with EN_SP; whenever one is set, the other is unset and vice versa.

14. SET_A - Set A-flag in FA
15. LD_FBI - Load FB and I
16. LD_FA - Load FA
17. EN_IP - Enable IP
18. EN_D - Enable D to data bus
19. LD_D - Load D
20. LD_IP - Load IP
21. CLR_CC - Reset Cycle Counter
22. CLR_K - Clear the K-flag in the IO module
23. END - Program End Signal
24. ERR - Error Signal

Chapter 3

Opcodes and Control Sequences

A control sequence is a sequence of control signals being sent out to various modules on subsequent clock cycles to implement a specific opcode. This chapter will go through each of the control sequences that implement the BF instruction set, including supporting instructions. All sequences have been summarized in Table A.1 for quick reference (Appendix A).

3.1 Binary Format

In Section 2.1, it was mentioned that the BF program is stored in ROM as a series of opcodes, each of which corresponds one-to-one to a BF command. However, the system needs to perform some additional non-BF actions to prepare itself for execution. For one, the data-tape is assumed to be zero-initialized; a freshly powered-on SRAM chip will contain random garbage, so we need to take care of that before the first BF command is executed. Moreover, the system is designed to accept a ROM chip containing multiple programs for convenience. This means that before blindly running the first program, there must exist some mechanism that looks for the index corresponding to the program the user is intending to run. To account for these issues, the binary format on the EEPROM expected by the system is as listed below. The `bfasm` utility was designed to translate any canonical BF source to the corresponding opcodes and dresses them with the necessary housekeeping instructions. The control sequences that implement the initialization and bootloading steps are further discussed in Section 3.9.

1. One or more `INIT` instructions to (partially) initialize the data-tape.
2. One `INIT_FINISH` instruction to return the data pointer to the start of the data section. This instruction also signals to the IO module that the system has been fully initialized.
3. One `LOAD_SLOT` instruction. The program index is supplied to the bus by the IO module and used to move the IP to the start of the corresponding program.

INIT
INIT_FINISH
LOAD_SLOT
PROG_START
<i>Program Slot 0</i>
PROG_END
PROG_START
<i>Program Slot 1</i>
PROG_END
...

Figure 3.1: Binary format expected by the system: initialization, bootloading and the BF programs themselves.

4. A sequence of BF programs, each preceded by a PROG_START instruction and ended by a PROG_END instruction. The PROG_START instruction serves as a barrier between programs and is used to skip over programs in order to find the correct program to run. PROG_END simply halts the clock and lights up an LED to indicate successful termination of the program.

3.2 Instruction Decoding

By setting the control signals as described in Chapter 2 appropriately, modules can work together to perform each of the BF and supporting instructions. The Control Unit implements this as a lookup-table in 3 ROM chips, where the instruction taken from program ROM (4 bits), flags (5 bits) and cycle counter (3 bits) act as an address into this table (Figure 3.2). Given that the CU has to be able to supply a total of 24 different control signals, three 8-bit EEPROM chips have been used to store the microcode lookup-table for a maximum of 24 signals (all of them used). More details on the implementation can be found in Section 4.11.

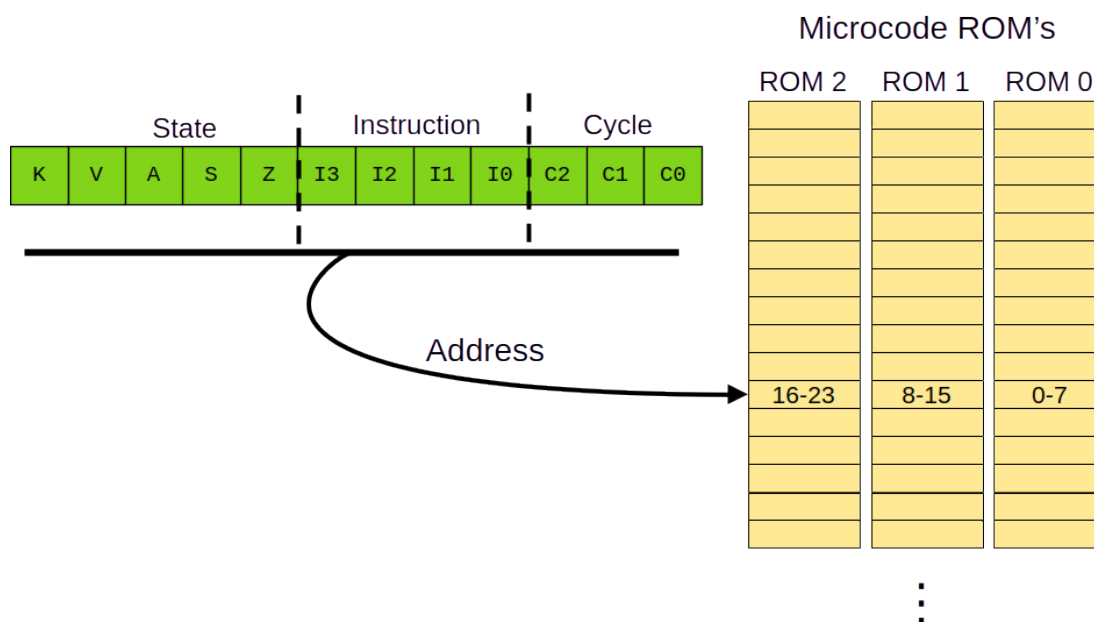


Figure 3.2: Decoding an instruction: the current instruction, flags and cycle-count are used as an index to the three ROM chips that output the control signals corresponding to the current state of the system.

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 (the Register Driver, see Section 2.10), only one register can be driven per clock cycle. In other words, the INC and DEC signals can be applied to only one register at a time.

3.3 Cycle 0: Instruction Fetch

The first cycle of most instructions is identical¹: A, V, S and Z are loaded into the FB register and the current instruction (pointed to by IP) is loaded from program ROM into the instruction register (I). This provides the CU with all the necessary information to determine the control signals for cycle 1: LD_FBI.

3.4 Modifying Data: + and -

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

Scenario 1: ($A = 0, S = 0$) - If the A-flag (address-change-flag) is not set, the value in D already corresponds to the value currently pointed to by the DP and no synchronization has to be performed. In that case, its INC signal is immediately asserted to the D register in order for it to increment on the next clock pulse. Referring to Table 2.2, we see only RS0 has to be asserted in conjunction with the INC signal to increment D (register address 0b001). The V-flag must also be set in order to indicate that the value in D has been changed: this is done by asserting the SET_V signal to FA and latching in the value using the LD(FA) signal. Now that the value has been incremented and the corresponding flag has been set, the IP is incremented using the register-driver on cycle 2. The cycle reset signal is asserted at the same time.

Cycle 1: INC, RS0, SET_V, LD_FA

Cycle 2: INC, RS2, CR

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

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

Cycle 2: INC, RS0, SET_V, LD_FA

Cycle 3: INC, RS2, CR

Scenario 3: ($S = 1$) - None of the actions above need to be performed when the S-flag is set, which means that we're in the process of skipping a loop-block. In this case, we ignore the command and increment the IP immediately and reset the cycle counter: INC, RS2, CR.

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

¹The OUT, IN and RAND instructions use specialized cycle-0 sequences. For more information, refer to Sections 3.7, 3.8 and 3.10.

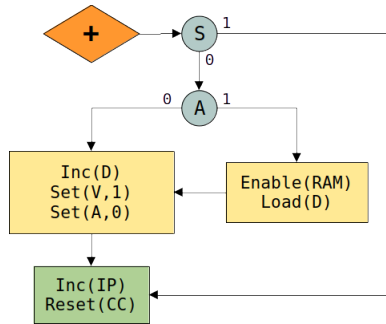


Figure 3.3: Block diagram for the + command. The diagram for the - command is equivalent (using *Dec* rather than *Inc*).

3.5 Moving the Pointer: < and >

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

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

Cycle 1: INC, RS1, SET_A, LD_FA

Cycle 2: INC, RS2, CR

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

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

Cycle 1: EN_D, WE_RAM, LD_FA

Cycle 2: INC, RS1, SET_A, LD_FA

Cycle 3: INC, RS2, CR

Scenario 3: ($S = 1$) - None of the actions above need to be performed when the S-flag is set, which means that we're in the process of skipping a loop-block. In this case, we ignore the command and increment the IP immediately and reset the cycle counter.

Cycle 1: INC, RS2, CR

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

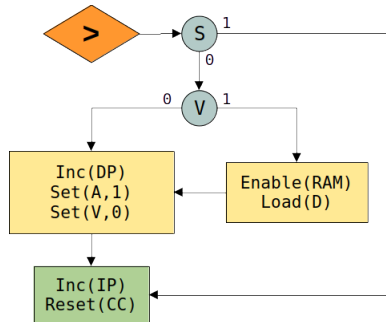


Figure 3.4: Block diagram for the > command. The diagram for the < command is equivalent (using Dec rather than Inc).

3.6 Conditional Jumping: [and]

These are by far the most complicated instructions and require lots of additional logic. Because the BF instruction set lacks a JMP instruction where some argument holds the destination address, the computer has to store the address of the opening [command in case it needs to loop back when the time comes. When a loop is skipped, the LS (Loop Skip) register is used to determine when execution should resume. This leads to multiple scenarios depending on the state of A, Z and S.

LOOP_START

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

Cycle 1: INC, RS0, RS2

Cycle 2: INC, RS2, CR

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

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

Cycle 1: INC, RS0, RS1

Cycle 2: WE_RAM, EN_SP, EN_IP

Cycle 3: INC, RS2, CR

Scenario 3: ($A = 1, S = 0$) - In the third scenario the A-flag *is* set, which means that we should first load the current value from RAM into the D-register (cycle 1) and reset the A-flag. The cycle count is immediately reset to 0 without incrementing the instruction pointer. This means the same instruction is reloaded with updated

flags on the next iteration, putting the system into either one of the states above (either scenario 1 or 2, depending on the value of Z).

Cycle 1: OE_RAM, LD_D, LD_FA, CR

Scenario 4: ($S = 1$) - In the final scenario, we are in the process of skipping code, indicated by the S-flag ($S = 1$). In this case, we have encountered a nested loop that needs to be skipped over, so we increment the LS register once more to account for another pair of nested `[]` brackets (cycle 1) and then continue to the next instruction (cycle 2). The control sequences are therefore identical to those in the first scenario:

Cycle 1: INC, RS0, RS2

Cycle 2: INC, RS2, CR

LOOP_END

Scenario 1: ($A = 0, Z = 1, S = 0$) - In the first scenario, which takes 2 cycles to execute, there is a known (synchronized) zero in the D-register ($A = 0$). This means we can immediately choose to exit the loop. To do so, the stack-pointer is decremented (cycle 1) to point at the previous value on the stack. In cycle 2, the IP is incremented as usual.

Cycle 1: DEC, RS0, RS1

Cycle 2: INC, RS2, CR

Scenario 2: ($A = 0, Z = 0, S = 0$) - In the second scenario, there is a known nonzero value in D. This means we must loop back to the IP-value stored on the top of the stack. This value is loaded into the IP-register by enabling the SP and RAM and setting the LD signal for the IP-register (cycle 1). In the second cycle, this new IP (pointing to a `[]`) is incremented to re-enter the loop.

Cycle 1: EN_SP, OE_RAM, LD_IP

Cycle 2: INC, RS2, CR

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

Cycle 1: OE_RAM, LD_D, LD_FA, CR

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

Cycle 1: DEC, RS0, RS2

Cycle 2: INC, RS2, CR

3.7 Output: .

Handshake Due to the asynchronous nature of the output peripheral, it is necessary to enter into a handshake protocol whenever a byte is put on the bus for display. In this protocol, the K-flag is used to communicate between the CPU and the IO module: it is set by the IO module to

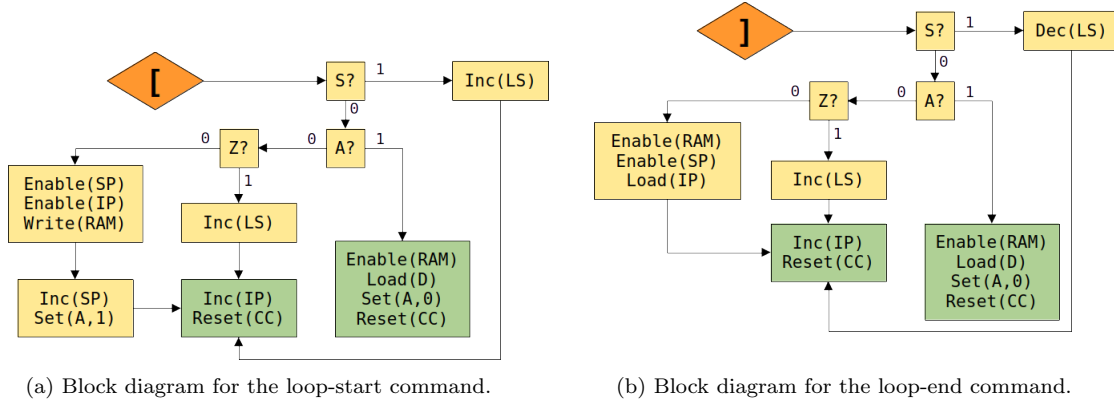


Figure 3.5

indicate that it has read the data from the bus, and reset by the CU to indicate that the handshake has been received and completed. The global step-by-step process is described below. The control sequences (on the side of the CU) that implement these steps follow after. Section 4.12 and the code in Appendix E show how the IO Module's side is implemented.

- Step 1:** First, the data is loaded into D if not already present ($A=1$). The CU then asserts the EN_OUT signal and enables D such that its contents appear on the data bus. It will keep EN_OUT high and keep the data asserted onto the data bus for as long as necessary, as indicated by the K-flag.
- Step 2:** The output module notices the EN_OUT signal and reads the value from the bus. When done, it sets the K-flag.
- Step 3:** When K is seen to go high, the bus is deasserted, K is reset and the transaction has finished.

Implementation When the OUT instruction hits its first cycle, it will first synchronize D with RAM if the A-flag or V-flag is set. This is necessary because these flags are then both set to 1 to indicate a wait state; synchronizing D allows these flags to both be safely reset when the instruction has finished. In cycle 2, A and V are latched into FB and the CC is reset, in order for the microcode to branch into the wait loop on the next cycle 0. Now that the system is back in cycle 0 with both A and V set, the system goes into a busy loop. The EN_OUT and EN_D signals are continuously asserted for as long as the K-flag is still low; an immediate cycle reset leaves the system in cycle 0. All other commands reload the flags and instruction in cycle 0; to maintain stable data on the bus without interruptions (due to resetting to the default cycle 0), we need the special $A=V=1$ state to encode this situation. When K finally goes high (K does not have to be latched, so its current value is always available), cycle 0 branches to a different sequence, where the A, V and K flags are cleared and the IP is incremented to move to the next instruction.

Scenario 1: ($A = 0, V = 0$) - No need to synchronize; go into wait-state by setting A and V to 1. Latch these in FB immediately to make them available on the next cycle 0.

Cycle 1: SET_A, SET_V, LD_FA

Cycle 2: LD_FBI, CR

Scenario 2: ($A = 0, V = 1$) - Synchronize D to RAM and go into the wait-state like before.

Cycle 1: SET_A, SET_V, LD_FA, EN_D, WE_RAM

Cycle 2: LD_FBI, CR

Scenario 3: ($A = 1, V = 0$) - Synchronize RAM to D and go into the wait-state like before.

Cycle 1: SET_A, SET_V, LD_FA, LD_D, OE_RAM

Cycle 2: LD_FBI, CR

Scenario 4: ($A = 1, V = 1, K = 0$) - Still waiting for K to go high; keep the data asserted and reset the cycle counter.

Cycle 0: EN_OUT, EN_D, CR

Scenario 5: ($A = 1, V = 1, K = 1$) - K has gone high; break out of the wait-state by clearing A and V (loading FA without asserting any signals) and move to the next instruction.

Cycle 0: LD_FA, CLR_K, INC, RS2

Cycle 1: LD_FBI, CR

3.8 Input: ,

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

Step 1: The CU asserts the EN_IN signal and will keep this control line asserted until K is set by the IO module, indicating that the data is ready.

Step 2: The input module notices EN_IN, waits until the keyboard buffer contains a value and puts this on the bus, then it sets K.

Step 3: When K is seen to go high, the value on the bus is loaded into D and K is cleared to complete the handshake.

Implementation The microcode for the sequences described above is very similar to that of the OUT instruction. Again, the wait state is encoded by setting A and V to 1, allowing the system to spin in cycle 0 while K is not yet set by the IO module. When K goes high, the data on the bus is latched into D, the A and K flags are cleared, V is set and the next instruction is reloaded. As was the case with the OUT instruction, we need an explicit LD_FBI before resetting the CC to compensate for the fact that the flags already have to be up to date when cycle 0 loads in order to implement the busy loop (all other commands refresh the flags in cycle 0). In this case, there is no need to synchronize D and RAM before setting the flags to the wait state. Since we expect a new value to be read into D as a result of this instruction, the V-flag should be set and the A-flag should be reset regardless of the state of the computer on entering this instruction.

Scenario 1: ($A = 0, V = 0$) - Go into wait-state.

Cycle 1: SET_A, SET_V, LD_FA

Scenario 2: ($A = 0, V = 1$) - Go into wait-state.

Cycle 1: SET_A, SET_V, LD_FA

Scenario 3: ($A = 1, V = 0$) - Go into wait-state.

Cycle 1: SET_A, SET_V, LD_FA

Scenario 4: ($A = 1, V = 1, K = 0$) - Still waiting for K to go high; keep enable signal asserted and reset the cycle counter.

Cycle 0: EN_IN, CR

Scenario 5: ($A = 1, V = 1, K = 1$) - K has gone high; load the data, break out of the wait-loop by setting V and resetting A, clear K and move to the next instruction.

Cycle 0: LD_D

Cycle 1: SET_V, LD_FA, CLR_K, INC, RS2

Cycle 2: LD_FBI, CR

Input Modes The input peripheral (probably) manages an internal buffer to serve subsequent bytes 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 two options:

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

In our implementation of the IO-module (see 4.12), both modes are supported and can be selected from the options menu.

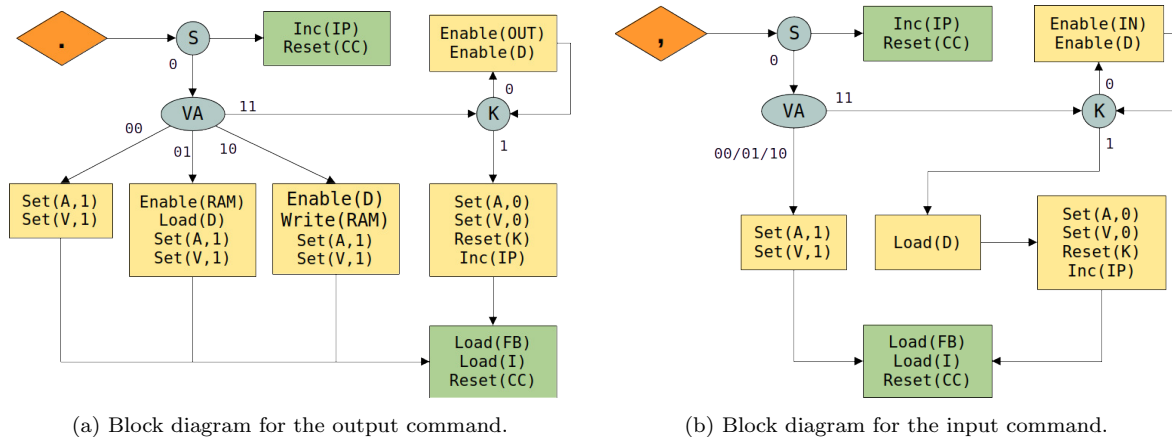


Figure 3.6

3.9 Initialization and Bootloading

In Section 3.1 a general overview of the contents of the program ROM was given. Before running the program, two main steps have to be taken: initialize the data-tape to zero and jump to the correct program (the EEPROM can contain multiple programs). These two steps are implemented using the INIT, INIT_FINISH, LOAD_SLOT and PROG_START opcodes.

INIT In any BF program it is assumed that all memory is zero-initialized. In practice, SRAM-modules will contain random values at startup, so the assembler must add a preamble to the main code in order to initialize the RAM (or at least part of it) to 0. While this could in principle be

implemented in terms of canonical BF commands (initializing one cell at a time using a sequence of [-] commands) it is much faster to write a bunch of known zeroes directly to RAM. This is the purpose of the INIT instruction: for each INIT instruction, a contiguous chunk of 256 memory-cells will be zero-initialized. Since it is guaranteed that the D register contains a zero after reset, this value can be directly written into RAM on the first cycle of INIT. The LS and DP registers are incremented during the first and second cycles, respectively. DP is incremented to move through the memory-space whereas LS is incremented in order to count to 256, at which point the S-flag will go low again due to the LS register overflowing back to 0. If more memory needs to be initialized, the assembler can simply concatenate multiple INIT instructions.

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

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

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

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

Scenario 2: ($S = 1$) - CR

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

INIT_FINISH The INIT_FINISH opcode is designed to do two things: it resets the DP to the start of the data section (0x0100) and it resets the K-flag. The former is necessary after initialization, which leaves the DP at whatever the last cell was that was initialized, and is achieved by asserting the CLR_DP signal. The latter will tell the IO module that initialization is finished (the IO module sets the K-flag at boot). The IO module will respond to this by taking ownership of the bus and writing the program slot to it. The LOAD_SLOT instruction (which always immediately follows the INIT_FINISH instruction) will use this value to skip to the desired program.

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

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

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

PROG_START The only opcode that performs non-trivial actions while the system is in skip-mode (initiated by incrementing LS as described above) is PROG_START. It checks the value in D, which was initially populated with the index of the desired program (0, 1, 2, ...), to see if the IP has arrived at the correct location: if the Z-flag is set ($D = 0$), we exit skip-mode and enter the program following the current PROG_START instruction. If the Z-flag is not set ($D > 0$), we

decrement D and keep scanning. In other words, D is decremented on each PROG_START until it hits zero, i.e. the selected program slot has been reached.

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

Cycle 1: DEC, RS0

Cycle 2: INC, RS2, CR

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

Cycle 1: DEC, RS0, RS2

Cycle 2: INC, RS2, CR

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

3.10 Convenience Opcodes

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

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

Cycle 1: INC, RS2, CR

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

Cycle 1: HLT

Cycle 2: INC, RS2, CR

PROG_END The PROG_END instruction is basically a HALT instruction that also lights up an LED by asserting the END signal. This lets the user know that the end of the program was reached gracefully. Unlike the HALT instruction, no Cycle 2 has been defined. Trying to resume

the clock after a PROG_END was reached will therefore result in the ERR signal being raised (see below).

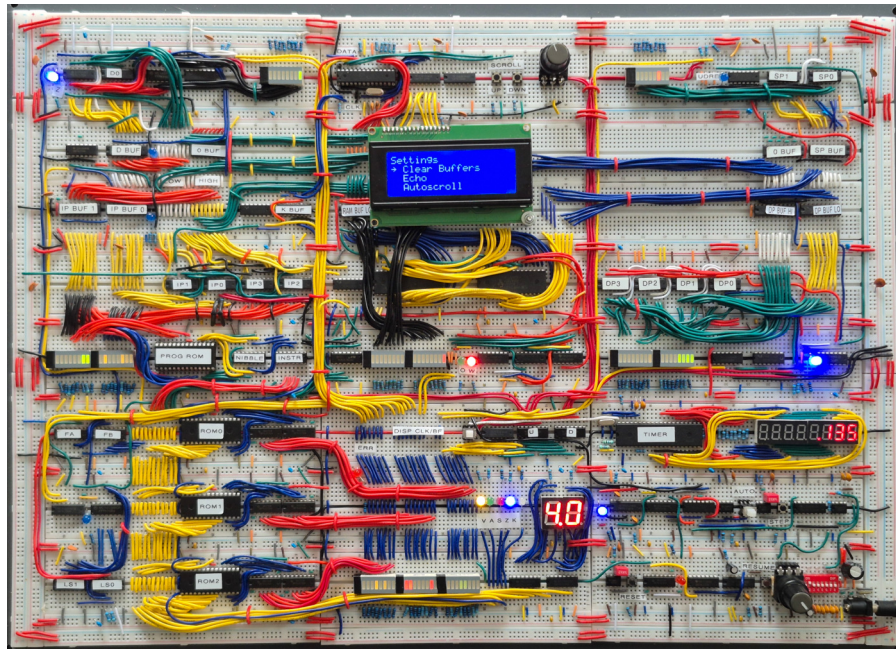
Cycle 1: HLT, END

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

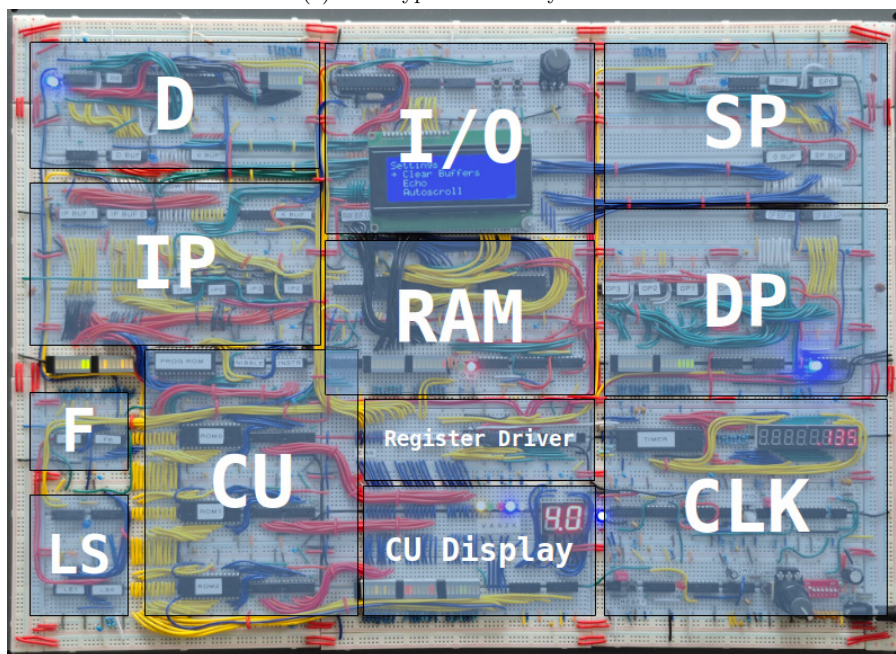
Chapter 4

Hardware Implementation

This chapter will discuss the implementation of each module and the way they integrate together to make the computer. Figure 4.1a shows the computer as it was in February 2025. The overlays shown in Figure 4.1b show where each of the modules described in the previous chapter is located on the computer. Schematics for each of the modules can be found in Appendix G.



(a) Prototype of February 2025



(b) Location of the modules on the prototype

Figure 4.1

4.1 Wiring

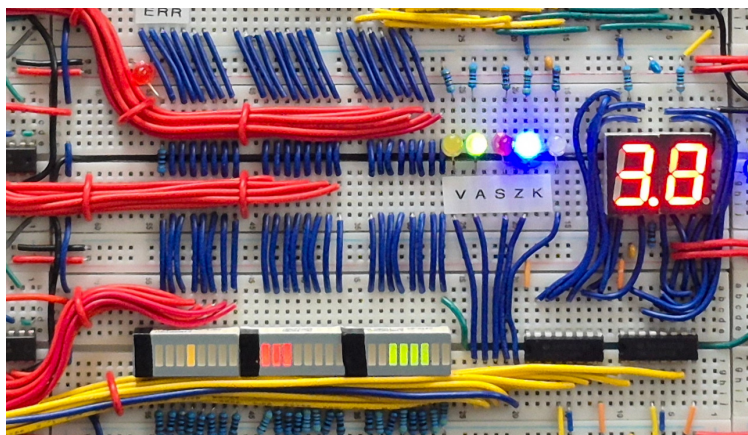


Figure 4.2: Close-up of the wiring inside the CU.

4.1.1 Breadboards

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

4.1.2 Wires

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

4.1.3 Busses

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

4.1.4 Bus Pull-Down

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

the DP or the SP has its outputs enabled at any time during execution. Therefore, no pull-down resistors were used on the address bus.

4.1.5 Power

Power is supplied by a 5V, 3A power supply (the system draws around 930mA), connected to power rails that run around the perimeter of the system and in between the breadboards. As many interconnections as possible between different parts of the power rails were made to ensure that all segments of the board receive a stable power supply and connection to ground. At short intervals, 100nF capacitors were placed across the rails to filter high-frequency noise.

4.1.6 Clock and Reset

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

4.1.7 LED Indicators

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

Additionally, two 7-segment displays, each driven by a 74LS48, are connected to the instruction register and cycle counter to display the current instruction and cycle as decimal numbers (e.g. 4.2 for cycle 2 of the > instruction, see 5.1). Unfortunately, the '48 does not support values over 9 (no hexadecimal representation) so non-BF opcodes like INIT lead to a blank display.

4.2 Clock and Controls

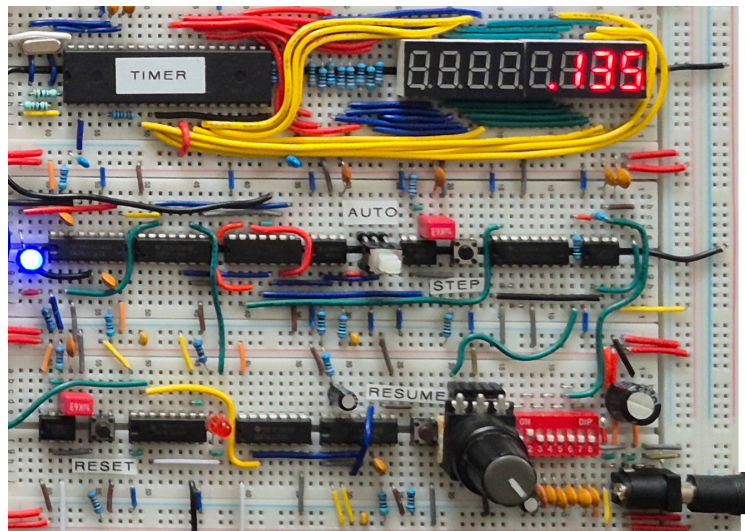


Figure 4.3: Close up of the Master Clock and Reset/Resume Modules.

4.2.1 Clock

The clock module is located at the bottom right of the computer and is responsible for providing a heartbeat to (most of) the modules. The core design of the clock, based around a 555 timer in astable mode, is taken directly from Ben Eater’s 8-bit computer videos [5]. The output frequency can be set using an array of DIP switches to select the capacitor of an RC circuit for coarse control and a 10K linear potentiometer for fine control. Two additional 555 timers are used to debounce both the pushbutton for the manual clock and the latching pushbutton which acts as a selector 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 edge and another on the falling edge of the output of the flip-flop. The 200ns pulse is generated by connecting a 100pF capacitor and a 5K resistor to the ’123, in accordance with the timing diagram from the datasheet [27], shown in Figure 4.4.

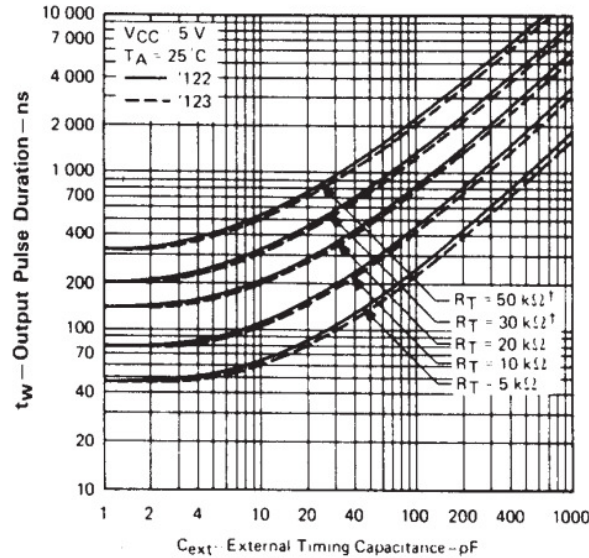


Figure 4.4: Pulse width of the 74LS123, based on the connected RC values. Taken directly from [27], page 7.

This results in two sets of clean signals at constant intervals. On the first pulse (rising edge of the output of the flip-flop), control signals are loaded from the microcode EEPROMs into a set of registers (74LS173) that buffer these control signals for stability; even when the inputs to the EEPROM address-pins change during execution of an opcode, this will not affect the control signals presented at the modules. The second pulse (generated by the falling edge of the flip-flop) is used as a clock to the modules; this is when the modules execute their command, like loading a value into RAM or incrementing the contents of a register. This approach guarantees a clean division between setting the control signals and clocking the modules. Figure 4.5 shows the timing diagram for the different signals discussed above.

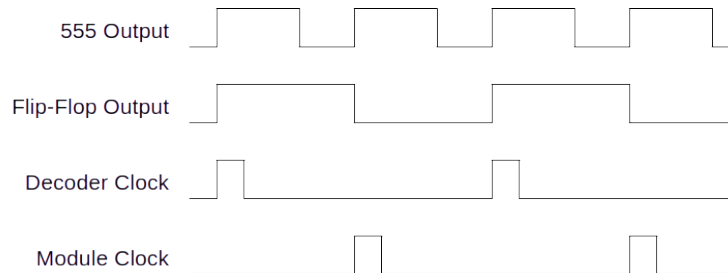


Figure 4.5: Timing diagram for the clock signals.

Frequency Control The frequency of the master clock can be set using DIP switches to select the capacitor value and a 10K potentiometer to select the resistance of the RC circuit connected to the 555 timer. The capacitor, in conjunction with a fixed 2K resistor, sets a broad range (lower capacitance corresponding to higher frequencies), while the potentiometer is used for fine-grained selection of the frequency within this range. This potentiometer is wired in series with a 1K resistor to ensure stability when the potentiometer resistance drops toward zero. Table 4.1 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

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

the frequencies displayed in Table 4.1). 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 clock frequency 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 [39], which is configured as an 8-digit frequency timer. It drives two 4-digit 7-segment displays at a one-second interval (the frequency is measured and updated every second).

4.2.2 Reset/Resume

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

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

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

the duration of the reset pulse can be calculated using

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

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

Resume The Resume button needs more sophisticated debounce circuitry due to the following scenario: when multiple HLT instructions are separated by a relatively small number of other instructions, a pulse on 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 equal in width to the clock pulses (Figure 4.5), 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 multivibrators 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 multivibrator. When the initial (short) pulse goes low, the second multivibrator generates a much longer pulse that is connected to the reset input of the first one, making sure it cannot be reactivated for some time.

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

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

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

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

By selecting a 680K resistor and a $4.7\mu F$ capacitor, a cooldown period of around 1 second is achieved. Figure 4.6 shows a timing diagram to illustrate this process in more detail.

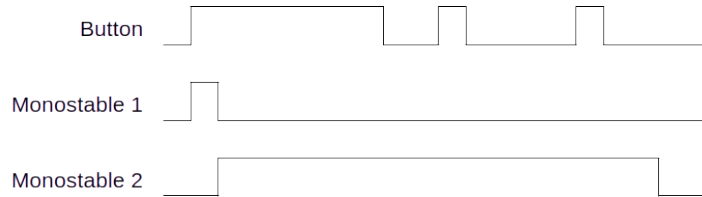


Figure 4.6: Timing diagram for the resume debouncer. The output of Monostable 1 is connected to the reset of the register that stores the HLT signal.

4.3 Register Driver

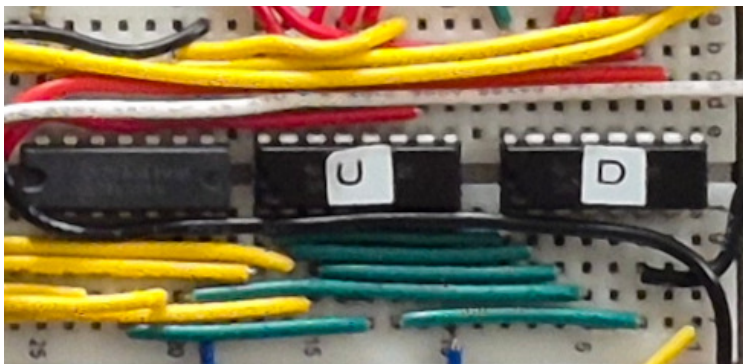


Figure 4.7: Close-up of the Register Driver Module.

The register driver is responsible for driving the U and D inputs of the 74LS193 counting registers that are used to implement the D, DP, SP, IP and LS register modules. To increment the '193, its D input needs to be held high while providing a low pulse to the U input. As explained in Section 2.10, a centralized driver was used to limit the number of logic ICs necessary to drive the registers and the total number of control signals necessary.

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

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

4.4 DP Register

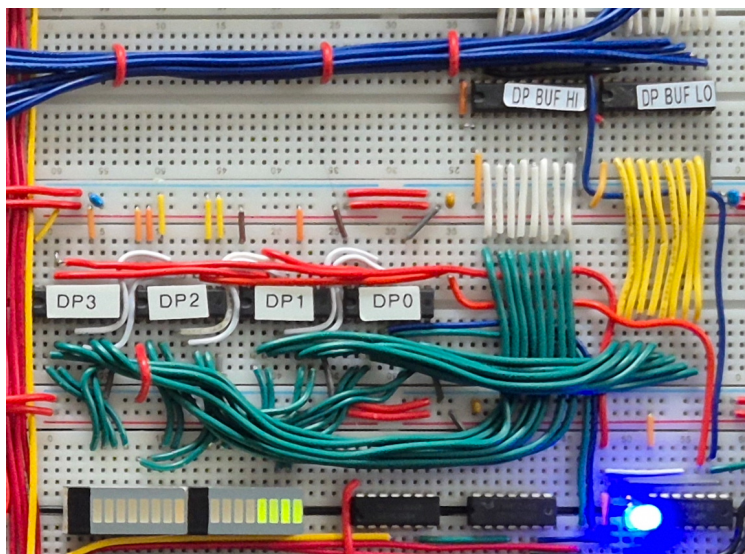


Figure 4.8: Close-up of the Data Pointer Register Module.

The DP Register Module is the module that is responsible for managing the data pointer; it contains a 16-bit value that is connected to the address bus of the RAM and points to the memory cell 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 2.2) 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 Output) 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 '193s corresponding to nibbles 0, 1 and 3, but to the load pin of nibble 2, whose 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 data section before the main program starts (see also Section 3.9). The global RESET signal is therefore ORed with the CLR_DP signal before going to the reset (and load) pins of the ICs.

Enabling Output Perhaps somewhat confusingly, the schematic (see below) shows that the SP_EN signal is used to enable the buffers. Because the DP shares the address bus with only the stack pointer (SP)—and their outputs should be mutually exclusive—the same signal can be used to enable and disable their respective buffers: when the stack pointer is enabled, the data pointer should be disabled and vice versa. Given that the output enable pin of the 74LS245 is active low, the SP_EN signal can be fed directly into the enable pin of the DP buffers. On the side of the SP,

the same signal goes through an inverter before going into the enable pin of its respective buffer. By default, when the SP is not enabled, the DP will provide its address to RAM. The address bus will therefore never be left floating, which has the nice side effect of always being able to visually see the current value in RAM by the LEDs connected to its outputs.

4.5 D Register

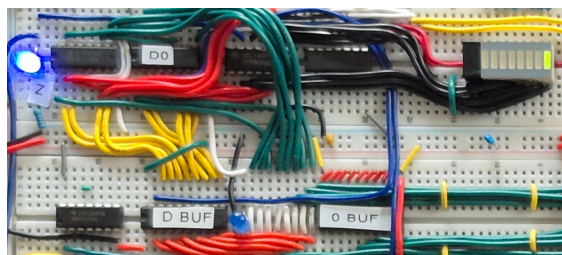


Figure 4.9: Close-up of the Data Register Module.

The data register (D) holds (a copy of) the value in memory currently pointed to by the data pointer (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 4.3 at address 1 (001, see Table 2.2). 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 data bus. Because the data bus is 16 bits 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 zeros. Storing nonzero values in the high byte of the data section 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 separate 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.

4.6 IP Register

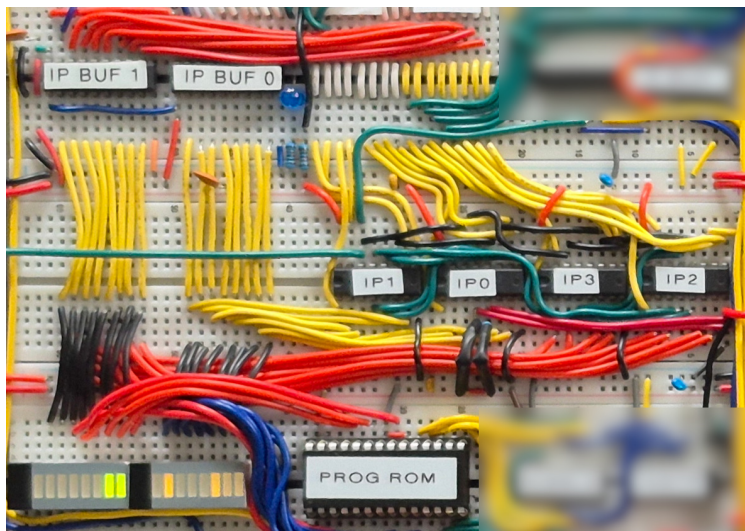


Figure 4.10: Close-up of the Instruction Pointer Register Module.

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

Forbidden Decrement The IP, like the D and DP registers, is driven by the Register Driver at address 4 (100), but should in principle never be decremented; it either moves to the right (next instruction) or jumps back by loading a value from the data bus. Its inability to move left (decrement) 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 data bus through two tristate buffers (74LS245) to avoid bus contention with the D register 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.

4.7 SP Register

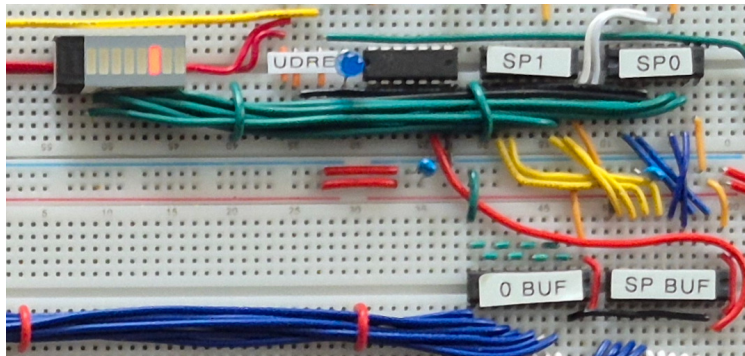


Figure 4.11: Close-up of the Stack Pointer Register Module.

The stack pointer (SP) holds an 8-bit value in the range 0x00 - 0xFF, which corresponds to addresses within the stack space of RAM, where IP values can be stored and loaded from when the system sees the [and] loop instructions. When a loop is entered, the SP first increments its value and the IP register then stores its value on the stack at the resulting address. The SP is now ready for the next value to be stored on the stack when a nested loop is encountered. It is therefore implemented using the 74LS193 binary counter and connected to the register driver at address 3 (011). The SP module is connected to the same RAM address bus as the DP, which means it should go through a tristate buffer to avoid bus contention. As mentioned before (4.4), the SP buffer shares its enable line (though inverted) with the DP. A second buffer that, when enabled, only outputs zeros on the address bus is used to make sure that only the stack is addressed by the SP and no accidental reads or writes happen in the data section.

4.8 LS Register

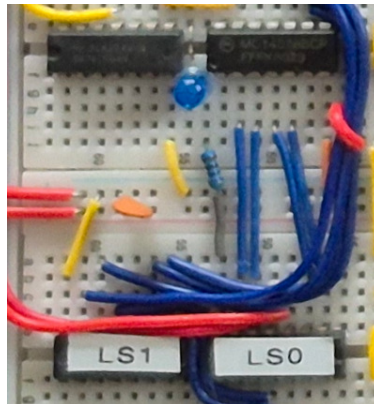


Figure 4.12: Close-up of the Loop Skip Register Module.

The Loop Skip Register (LS) is used to produce the loop skip flag (S, see 2.7). It is implemented using two 74LS193 binary counters and is connected to the register driver at address 5 (101). Like the Z-flag, the S-flag is produced by sending the outputs of the binary counters through an 8-input NOR gate, the output of which is then inverted and connected to the FB flag register. When any of these bits are high, the S-flag will be raised, indicating that the computer is in the process of skipping the current loop.

4.9 Flag Registers

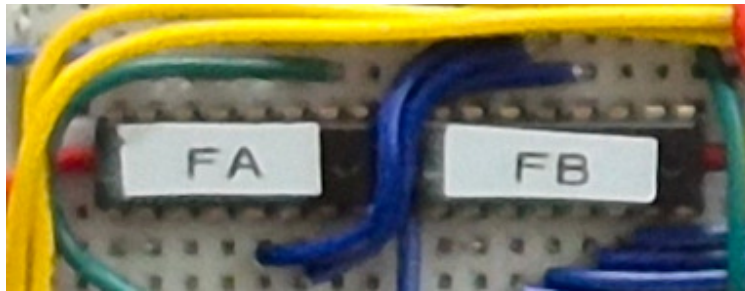


Figure 4.13: Close up of the flag registers FA and FB.

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

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

FB At each cycle 0, the A and V-flags are loaded from FA into FB, together with S and Z coming from the LS and D registers respectively. Since this always happens in conjunction with loading the instruction from program ROM into the instruction register (see 4.11), the control signal has been named LD_FBI. The flags in the FB register will (mostly) remain constant during the execution of an opcode.

4.10 RAM

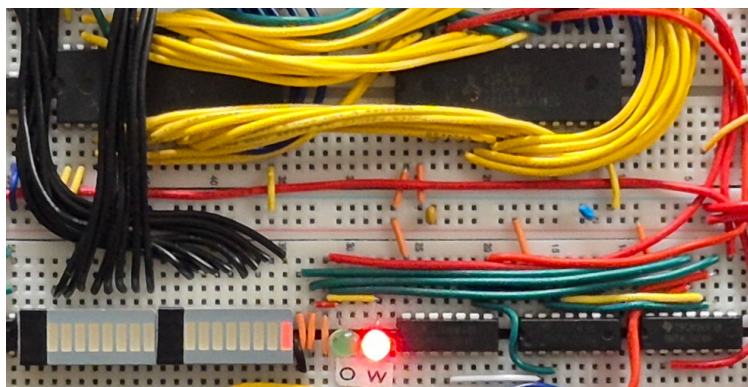


Figure 4.14: Close-up of the RAM Module.

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 bits in size. Therefore, the RAM module contains two $512\text{K} \times 8\text{-bit}$ SRAM chips (AS6C4008). Although each chip physically contains 512 KiB, the 16-bit address bus exposes only 65,536 paired addresses, for a total of 64K accessible 16-bit memory cells. When the data register's output is enabled, its value will be stored in the lower byte of RAM (chip 0); the high byte (chip 1) is filled with zeros. The second chip is therefore only used to store the high byte of the IP. Since the stack can only store up to 256 values (in the address range $0\text{x}0000\text{--}0\text{x}00\text{FF}$), the remainder of the accessible capacity of the RAM chip storing the high byte will never be used. Though this might look wasteful, it was preferred over the alternative of using two sequential bytes on the same chip to store the low and high bytes separately. This would have introduced more complexity in the surrounding logic and would have led to multiple cycles for reading from/writing to the stack, negatively impacting performance.

Buffering The AS6C4008 already provides a Chip Enable input which is supposed to be used when the data is connected to a data bus. 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 LEDs, 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 truth table for this logic is shown in Table 4.2. The LEDs are not shown in the schematic, but have been connected directly to the data lines of the RAM in this configuration.

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

Table 4.2: Truth table for the logic that drives the RAM in such a way that its output is always enabled, in order for the data to be visualized on LEDs without enabling it to the data bus.

4.11 Control Unit

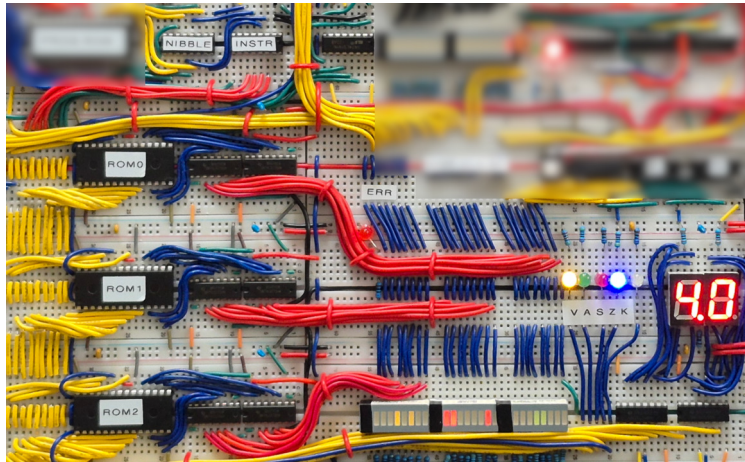


Figure 4.15: Close-up of the Control Unit.

4.11.1 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), combines to form a 12-bit address into a set of three EEPROM chips (AT28C64B), each of which contains part of 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	$(000_2 - 111_2)$
3-6	Instruction	$(0000_2 - 1111_2)$
7-11	Flags	$(00000_2 - 11111_2)$
12	Unused	

Generating and Programming Microcode The three EEPROMs 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 5.3), a utility developed to make the microcode programming more maintainable. Mugen generates the images from a specification file. The full Mugen specification for Synapse-191 is shown in Appendix B. This is a direct representation of the microcode shown in Table A.1 (Appendix A).

Instruction Nibbles The actual BF program is stored in another 8K EEPROM chip (AT28C64) and is addressed by the instruction pointer as mentioned before. Since each BF instruction only

needs 4 bits to be encoded (there are fewer than 16 different opcodes), we can store up to 16K instructions 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 data bits. 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 number of instructions that can be stored and retrieved.

Instruction Register The selected 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 on 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 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 after an instruction has completed).

4.11.2 Early Segmented Implementation

In the initial implementation of the CU, the entire microcode lookup-table was implemented in a single EEPROM chip. Two additional address bits, the segment bits, were included to address at most 4 different sections of the ROM (at the time, the K-flag did not exist yet, leaving the final two address bits 11 and 12 free to use for this purpose). Each section contained an 8-bit control word, such that rather than latching the entire 24-bit control word in a single clock, the logic was designed to access each of the 3 control words sequentially and latch them one by one before clocking the modules. This design choice was made back then because of the low availability of the AT28C64B chip; only two of these chips were acquired, one for storing the program and the other for storing the microcode. While chip-efficient, this approach requires 3 times as many clock cycles per instruction and was abandoned as soon as we got our hands on more EEPROM chips. Moreover, in our current implementation only one address bit (12) is free to use due to the K-flag being added in a later stage of the build, leaving no more room for segmentation even if we wanted to. Figure H.11 in Appendix H shows the system at a point in time where the segmented CU was still present.

4.12 IO Module



Figure 4.16: Close-up of the IO Module.

4.12.1 Features

The IO system is handled by an ATmega328P microprocessor, commonly found in the Arduino Uno. It has five main functions:

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

Buttons and Menu Two buttons are provided to interact with this system. They can be used to scroll the screen or to access and navigate a menu (Figure 4.17). This menu lets the user do the following:

1. Select the program slot that is loaded by the system on reset.
2. Clear the screen and keyboard buffer.
3. Change the display-mode. By default, incoming data is interpreted as ASCII characters. When it should be displayed as raw numerical values (either in base 10 or 16), this option can be selected from the menu. When in either of these numerical modes, a delimiter character can be selected to separate bytes visually.
4. Set autoscrolling on/off. By default, the screen will scroll its contents when they overflow to always keep the most recent data in view. When new data is displayed, the screen is always scrolled to display this data. Setting autoscroll to 'off' will disable these features.

5. Echo on/off. When running an interactive program that requires keyboard input, the user probably wants to see what is being typed. This is the default behavior (echo on). If for some reason the keypresses should not be displayed, this option can be disabled.
6. Set the input-mode. By default, the IO module will wait for the input-buffer to contain a value before putting anything on the bus and notifying the CU through the K-flag (buffered input-mode). However, an alternative mode (immediate) can be selected, in which case the IO module will put a zero on the bus when the buffer is empty and set the K-flag regardless. This can be helpful if programs require real-time inputs (e.g. for simple games).
7. Set the RNG seed. For programs that use the Random Number Generator as an input device, the seed can be set through this option. Since the same seed will produce the same sequence of numbers, this option can be used to control the randomness of the application. A ‘true’ random seed can normally be emulated by seeding the generator with the reading of a floating analog input, for example, but sadly no free analog inputs were left available on the MCU.
8. Reset to default settings. Whenever settings have been changed, the new settings will be saved to the persistent EEPROM memory of the MCU and loaded back on startup to make the settings persist when the MCU is powered down. This option allows you to revert all changes and load the default settings back in.

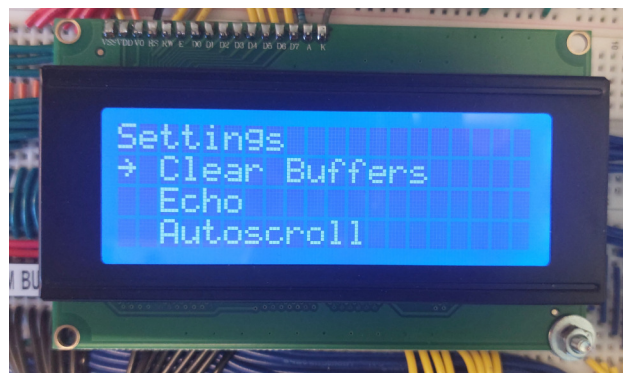


Figure 4.17: Part of the menu that is accessible by pressing both scroll-buttons simultaneously.

4.12.2 System Interactions

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

'74 has asynchronous clear lines, active low) was implemented in terms of NOR gates, requiring only a single 74LS02 quad NOR chip. Setting $R = \text{RESET}$, $K = \text{CLR_K}$ and $C = \text{M_CLK}$, the logic required to drive the clear lines of the flip-flops (Q) can be written as

$$Q = \neg(R \vee (K \wedge C)).$$

The output Q is driven low when either the RESET line goes high or the CLR_K is asserted together with the clock. This can be rewritten in terms of NOR-operations (using ∇ to denote the NOR operator) as

$$Q = R \nabla (\neg K \nabla \neg C)$$

and finally, replacing the NOT-operators with NOR-operators as

$$Q = R \nabla ((K \nabla K) \nabla (C \nabla C))$$

for a total of 4 NOR gates.

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

Output An interrupt attached to a timer fires at a set interval. When the system is in its IDLE state, it will check the EN_IN and EN_OUT lines to determine if it should initiate a read or write sequence to the data bus. When EN_OUT is found to be high, it will copy the byte currently present on the bus into its screen buffer, set the K-flag and wait for the CPU to reset K before waiting for the next instruction. When K is seen to go low, the handshake has been completed and the module returns to its IDLE state (see Figure 4.18).

Input If the EN_IN signal is found to be asserted, the system has two possible courses of action, depending on the input mode currently set by the user (through the on-screen settings menu). In the (default) buffered mode, the system will change its state to WAIT_KB and wait for a byte to become available in the keyboard buffer. As soon as it does, it will provide this value onto the bus and notify the CU by setting the K-flag. It then waits for K to go low again, confirming that the CU has taken the value from the bus and that the module should release the bus again. In immediate mode, all zeros will be written to the bus and the K-flag is set immediately even when the keyboard buffer is still empty. If, in addition to EN_IN, EN_OUT is asserted as well, a random byte is put onto the bus (see Figure 4.18).

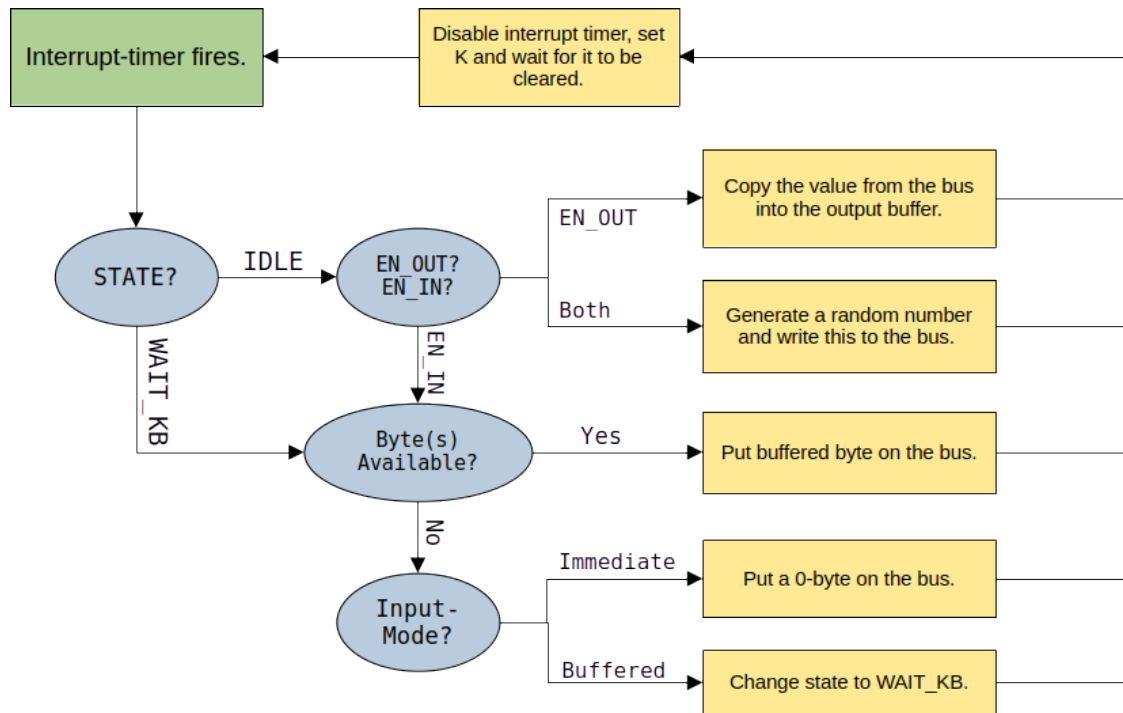


Figure 4.18: Control flow inside the ISR running on the microcontroller.

4.12.3 Shift Register

A shift register (74HC595) had to be used to decrease the number of pins on the ATmega328P necessary to drive the LCD screen. Every IO pin has been used, so the shift register proved invaluable for this application.

4.12.4 LCD Screen

The software was written in such a way that most common LCD character screens (compatible with the Hitachi HD44780 driver) will be handled appropriately. Both a 16x2 and a 20x4 display have successfully been installed in the computer. A modified version of the `LiquidCrystal_74HC595` library was used to implement the LCD driver.

4.12.5 Keyboard

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

4.13 IO Module Firmware

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

4.13.1 Interrupts

To make sure the module remains responsive during operation, regardless of I/O throughput, its input signals (EN_IN and EN_OUT) are checked at regular intervals, triggered by a timer interrupt. This timer fires at a rate of 200 Hz and triggers a highly optimized, low-cost interrupt service routine (ISR) that inspects the input signals and performs the corresponding action. Even when the ISR triggers at only 200 Hz, this will feel close to instantaneous to the user operating the computer. This leaves plenty of time for the microcontroller to drive the screen, handle the button input and manage the menu system. When the ISR is triggered by the timer and it is determined that some action needs to be taken, the K-flag is set to indicate to the external system that the action has been completed (see below). When K is finally cleared, the microcontroller needs to respond immediately because the system expects it to release control of the bus immediately after K is cleared (e.g. after responding to the EN_IN command). Therefore, the same 200 Hz interrupt cannot be used for checking K. Instead, the timer is stopped before checking K and a second ISR is attached to the falling edge of K. This ISR will trigger immediately when K goes low, allowing the microcontroller to respond quickly and set its data pins back to a high-impedance state (`pinMode(X, INPUT)` in Arduino nomenclature). It will then re-enable the timer to go back into normal operation. The C++ code that represents the state machine previously shown in Figure 4.18 is shown in Listing 16 (Appendix E).

4.13.2 Ring Buffers

Overview The IO module employs a set of custom ring buffers that provide high-speed, non-blocking data transfers between the asynchronous microcontroller firmware and the synchronous TTL-based CPU. The ring buffer (or circular buffer) is a fixed-size, first-in-first-out (FIFO) data structure that uses two indices, commonly called the **head** and **tail**. The **head** marks the position where the next incoming element will be written, while the **tail** indicates the position of the next element to be read. When either index reaches the end of the buffer, it wraps around to the beginning, forming a logical ring in memory. This structure eliminates the need for memory allocation or data shifting, providing constant-time access for both enqueue and dequeue operations. See Figure 4.19.

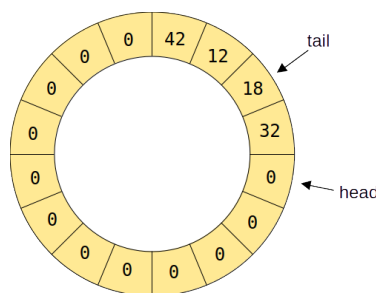


Figure 4.19: In this ring buffer, the value 18 (pointed to by **tail**) will be returned on the next query. Data entering the ring buffer will be stored at the cell pointed to by **head**. The value 42 is the first value to be overwritten when the **head** pointer wraps around to the top of the buffer.

In the Synapse-191 I/O module, one ring buffer is dedicated to temporarily storing incoming data (bytes coming from the CPU that need to be written to the LCD screen), the screen-buffer, and another is used for outgoing data (keyboard input that needs to be put on the data bus), the keyboard-buffer. The main loop periodically flushes the screen-buffer to the display and refills the

keyboard-buffer with new data when it arrives. Meanwhile, the interrupt service routine (ISR) accesses these buffers asynchronously to satisfy CPU read and write requests. Since all the heavy lifting (interpreting incoming data, managing the LCD screen, decoding keyboard scan-codes) is done in the main-loop, the ISR is kept very compact.

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

- **Index Type Specialization:** The index type (data type used to store the `head` and `tail` indices) is selected automatically at compile time using template metaprogramming. For small buffers, 8-bit indices are used instead of 16- or 32-bit counters, reducing instruction count and register usage and thus ensuring the fastest possible integer operations on our 8-bit MCU.
- **Integer Wrapping Optimization:** For buffers that are exactly as big as the maximum number representable by some type, that type is used to index the buffers as per the logic above. For example, a 256-byte buffer can be indexed fully using an 8-bit type like `unsigned char`. A value of 255 represented by this type will automatically wrap around when incremented, rendering any additional modular arithmetic to wrap around to the start of the buffer unnecessary.
- **Power-of-Two Optimization:** When the buffer size does not match the condition above, but *is* a power of two, the modulo operation used for index wrapping is replaced with a simple bitwise AND operation: $i = (i + 1) \& (N - 1)$. This reduces the naive modulo operation from multiple CPU cycles to a single instruction.

Volatile Access and Memory Barriers The `head`, `tail`, and data array are declared `volatile` so that their accesses are emitted rather than cached or optimized away across the ISR and main loop. Additionally, an inline assembly memory barrier prevents the compiler from reordering memory accesses across the barrier, ensuring that writes to the buffer occur before the corresponding index is updated.

4.13.3 Direct Port Access

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

4.13.4 Compile-time Menu Structure

Finally, the on-screen configuration interface uses a statically defined menu tree that is fully constructed at compile time using template metaprogramming techniques. This eliminates dynamic memory allocation and object initialization overhead during runtime, minimizing both execution

latency and memory footprint. Together, these optimizations produce a maintainable, fast and memory-efficient IO subsystem capable of operating reliably at high CPU-core clock speeds.

```

1  using Menu = MainMenu <
2      SelectSlot<
3          SlotSelector<0, N_SLOTS - 1>
4      >,
5      Clear,
6      Echo<
7          EchoOn,
8          EchoOff
9      >,
10     Autoscroll<
11         AutoscrollOn,
12         AutoscrollOff
13     >,
14     DisplayMode<
15         TextMode,
16         DecMode <
17             CommaDelim,
18             SemiDelim,
19             BarDelim,
20             SpaceDelim
21         >,
22         HexMode<
23             CommaDelim,
24             SemiDelim,
25             BarDelim,
26             SpaceDelim
27         >
28     >,
29     InputMode<
30         BufferedInput,
31         ImmediateInput
32     >,
33     SetRNGSeed<
34         SeedSelector<RNG_MIN_SEED, RNG_MAX_SEED>
35     >,
36     Defaults,
37     Exit
38 >;

```

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

Chapter 5

Supporting Tools

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.

5.1 Assembler: bfasm

Even though the computer is designed to run BF natively, we can't just burn any text file containing BF commands onto the program ROM and expect it to execute them. Instead, each of these commands has to be translated into its corresponding binary opcode. Table 5.1 lists all the available commands and the values that map to these commands. As explained in Chapter 3, there are a few non-BF instructions that have been added.

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

Table 5.1: Opcode values for each of the available commands.

bfasm performs pretty much a one-to-one transformation of the BF commands in the provided

text file into these values. It will add the preamble commands to initialize and bootload the system, as was discussed in Section 3.1 and shown in Figure 3.1.

5.1.1 Features

The **bfasm** utility provides the following features:

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

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

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

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

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

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

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

```

1 Usage: ./src/bfasm/bfasm [options] <file1, file2, ...>
2 Options:
3 -h, --help                Display this text.
4 -H, --halt-enable         Interpret '!' as HLT in the BF code
5 -r, --rand-enable         Interpret '?' as RAND in the BF code
6 -g, --debug               Place a breakpoint (!) after each instruction.
7 -e, --echo                Follow each input command (,) up by an output command (.)
                        to echo keyboard input.
8 -d, --max-depth [N]      Maximum nesting depth of []-pairs.
```



```
9 -p, --print-filename      Add BF code to print the source filename before the
    program starts.
10 -z [N]                   Initialize N chunks of 256 bytes with zero's. Default: N =
    1.
11 -u, --allow-unbalanced-loops
12                           By default, the assembler will refuse to produce a program
    with unbalanced
13                           loops ([ and ] do not match). Using this option will allow
    for this to occur.
14 -o [file, stdout]        Specify the output stream/file (default stdout).
15
16 Example: ./src/bfasm/bfasm -p -o image.bin program1.bf program2.bf
```

5.2 Programmer: bflash

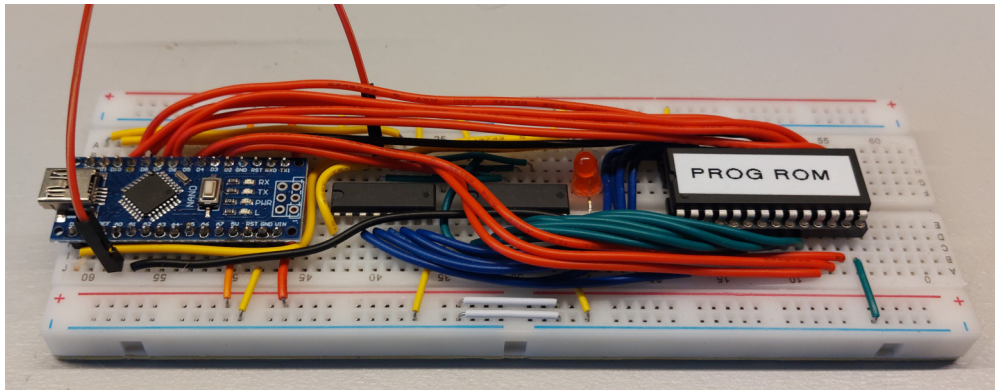


Figure 5.1: EEPROM chips were programmed using an Arduino Nano on a breadboard.

5.2.1 Overview

Given that there are four EEPROM chips embedded in the computer (one containing the program and three containing the microcode), we had to develop a toolkit for programming these. Specialized programmers can be pretty expensive and relatively hard to acquire, so an Arduino Nano was used to carry out that task. It waits for a serial connection and transfers incoming data byte by 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 5.1) is shown in Appendix G.

5.2.2 Flashing the AT28C64B

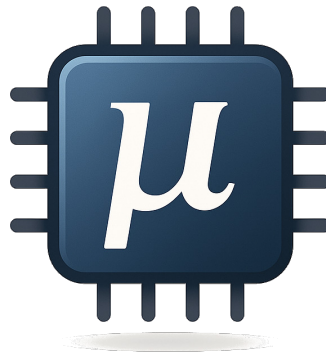
The AT28C64B $8K \times 8$ EEPROM chip is used for both microcode and program storage. A value can be written to a specific address using the following sequence of inputs:

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

5.2.3 Shift Register

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

5.3 Microcode Compiler (Mugen)



5.3.1 Motivation

Initially, the binary images that were burnt onto the microcode EEPROM chips were generated using a simple Octave/MATLAB script. This meant that both the microcode and the logic to generate the images had to be expressed in this language. While this certainly worked (albeit a bit slowly), 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 at <https://github.com/jorenheit/mugen>. The listing in Appendix B shows the contents of the Mugen specification file for this project.

5.3.2 File Structure

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

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

5.4 Emulation Framework (Rinku)



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

Chapter 6

Runtime Results

In this chapter we show snapshots of the output during the execution of each of the BF programs listed in Appendix D. Each of these programs was able to run successfully at the maximum clock speed of around 250 kHz ($5 \times 256 = 1280$ bytes of memory were initialized to 0). For each program, the following statistics have been collected in a modified version of the interpreter shown in Listing 1:

- *Program Size* - the number of BF commands in the program, i.e. the number of bytes in the .bf file, not counting non-BF characters.
- *Stack Size Required* - size of the stack required to run the program, i.e. the maximum nesting depth encountered while executing loops.
- *Memory Cells Required* - the minimum number of memory cells required to execute the program. In our case, this corresponds to the minimal amount of cells that needs to be zero-initialized at startup in order for the program to execute correctly.
- *Runtime Instructions* - the number of BF instructions that are executed when the program is run. Due to the existence of loops, this number differs (for most programs) from the program size.
- *Runtime Clock Cycles* - the number of clock cycles that elapsed over the course of execution.
- *Cycles per Instruction* - the average number of clock cycles needed to execute a single BF instruction. This value changes from program to program and depends on the structure of the program, the number of skipped loops, commonality of command sequences that leverage the A and V flags, etc. These values have been measured in the interpreter and have been (roughly) verified on the physical hardware.

6.1 Stability

Throughout development, the system was plagued by instabilities. At first, these could be traced back to bugs in the microcode, subtle architectural issues or even wrong connections that only sometimes caused problems. Then, the clock speed and overall stability were highly influenced by bad IO design. The IO module was connected to the system clock and expected to run its interrupt routine (ISR) within the time between two ticks; this left the module unresponsive at high clock speeds (there was almost no time between calls to the ISR to manage other functionality

like polling the keyboard) and would sometimes cause bus contention if it could not reset its outputs in time. With a proper handshake implemented using the K-flag and a completely asynchronous setup where the IO module only polls its input at a very low frequency, leaving the rest of the microcontroller’s capacity for handling the user interface, all these problems went away. New clock speeds were achieved to the point that it became stable up to the limits of the clock module itself. It has not been pushed any further at the time of writing, though that may very well be possible.

6.2 Test Suite

6.2.1 hello.bf



Figure 6.1: *Hello World*

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

Program Size	112
Stack Size Required	1
Memory Cells Required	4
Runtime Instructions	390
Runtime Clock Cycles	1273
Cycles per Instruction	3.3

Table 6.1: Properties of `hello.bf`, see Appendix D.1

6.2.2 euler.bf



Figure 6.2: Calculating Euler's Number

The Euler number (e) is an irrational number, so calculating its digits is an infinite process; the `euler.bf` program (written by the notorious BF developer Daniel Cristofani [13]) never ends. Running this program for a long time exposes a weakness of the system; sometimes the system gets into some erroneous state and goes haywire. It is still not fully understood what causes this to happen; based on qualitative observations, it seems as if the probability of failure increases with time and persists afterwards even after a reset. Shorting the power rails (after power has been disconnected) seems to stabilize the system again, so maybe the instability stems from charge buildup due to bad connections to power. Such bad connections have been found throughout the build process, so there may still be subtle bad connections present around the board.

Since this program runs indefinitely, there is no bound to the number of runtime instructions. The number of memory cells required also keeps growing with time; it was found to grow proportionally to the square root of the number of BF instructions that had been executed up to that point according to $M = 0.36\sqrt{N}$. A quick back-of-the-envelope calculation shows that for our case, with 65,280 data cells available (the first 256 addresses are reserved for the stack), we'd be able to execute $\left(\frac{65,280}{0.36}\right)^2 = 3.3 \times 10^{10}$ BF instructions, which would take 1.1×10^{11} cycles to complete (using the measured value of 3.4 cycles per instruction), or 4.5×10^5 seconds at 250 kHz (over 124 hours). It's probably safe to assume that those limits won't be reached any time soon.

Program Size	1424
Stack Size Required	11
Memory Cells Required	$\sim \sqrt{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.4

Table 6.2: Properties of `euler.bf`, see Appendix D.3

6.2.3 phi.bf

Figure 6.3: Calculating ϕ

Like e , the Golden Ratio $\phi = \frac{1+\sqrt{5}}{2}$ is irrational and thus has infinitely many digits. The `phi.bf` program [12] runs indefinitely and has growing memory demands when it runs for a longer amount of time. Again, the number of memory cells required grows as the square root of the number of BF commands executed according to $M = 71 + 0.033\sqrt{N}$. This number grows even more slowly than that required to calculate Euler's number, so there won't be any memory issues here either.

Program Size	1950
Stack Size Required	17
Memory Cells Required	$\sim \sqrt{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.6

Table 6.3: Properties of `phi.bf`, see Appendix D.4

6.2.4 factorial.bf



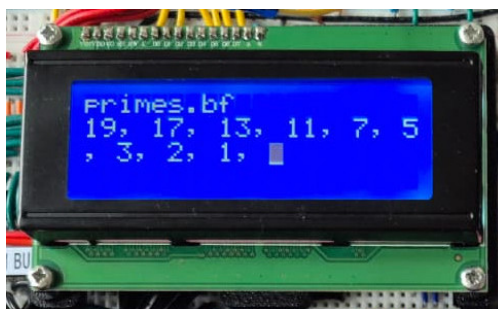
Figure 6.4: Calculating increasingly larger factorials

Another program that runs indefinitely is `factorial.bf`; it calculates $n!$ for ever increasing n ad infinitum. Unfortunately, this program was found and downloaded long before this report ever needed to mention it; we were unable to trace and credit its source at the time of writing. The memory requirement grows according to a cube root, so this program can run for an even longer time compared to `phi.bf` and indeed `euler.bf` before memory runs out (provided it runs without issue): $M = 12.5\sqrt[3]{N}$.

Program Size	964
Stack Size Required	15
Memory Cells Required	$\sim \sqrt[3]{N}$
Runtime Instructions	∞
Runtime Clock Cycles	∞
Cycles per Instruction	3.6

Table 6.4: Properties of `factorial.bf`, see Appendix D.2

6.2.5 `primes.bf`

Figure 6.5: *Calculating primes < 20.*

Found on Reddit by the user Troemax [14], this program checks primality for all numbers ≤ 20 (this value can be adjusted by changing the number of `+` commands at the very start of the program). It runs stable at all tested clock speeds up to 250 kHz. The algorithm reuses memory very efficiently during the course of its execution, requiring only 18 cells.

Program Size	790
Stack Size Required	7
Memory Cells Required	18
Runtime Instructions	3.6M
Runtime Clock Cycles	12M
Cycles per Instruction	3.3

Table 6.5: Properties of `primes.bf`, see Appendix D.5

6.2.6 primes2.bf



Figure 6.6: *Calculating primes < 20 (entered at runtime).*

Like the former prime-generating program, this program calculates all primes below 20. The difference between `primes.bf` and `primes2.bf` is that the latter asks for the upper bound at runtime (expecting keyboard input). The program was posted on the Code Golf section of Stack Exchange by user Alexandru, challenging other users to implement small interpreters that can successfully run it.

Program Size	3880
Stack Size Required	6
Memory Cells Required	10
Runtime Instructions	5K
Runtime Clock Cycles	1.8M
Cycles per Instruction	3.5

Table 6.6: Properties of `primes2.bf`, see Appendix D.6

6.2.7 tictactoe.bf



Figure 6.7: *Playing Tic Tac Toe against Synapse-191.*

Found on GitHub [16], from a developer by the username Mixtela [16], we found a BF program that actually implements a game of Tic Tac Toe, including a very capable ‘AI’ opponent. The statistics in Table 6.7 have been gathered by playing a single game (which ended in a draw).

Program Size	6880
Stack Size Required	12
Memory Cells Required	72
Runtime Instructions	224K
Runtime Clock Cycles	774K
Cycles per Instruction	3.5

Table 6.7: Properties of `tictactoe.bf`, see Appendix D.7

Part II

Brainfuck Compilation

Chapter 7

Introduction to Acus

7.1 What This is Not

Acus¹ is not a BF compiler: it is a C++ library for the development of such compilers. This chapter is *not* a guide on how to use Acus. For examples and extensive API details, please visit the GitHub repository at [10]. Instead, this chapter goes through the most important concepts, methods and abstractions that allow for the translation of high-level programming concepts (like function calls and flow control) into BF.



7.2 Motivation and History

Up to this point, almost every BF program used to test the computer was found online and used verbatim. On some occasions, simple BF routines were written to test very specific functions of the computer, but these were borderline trivial to produce. The next goal, therefore, was to create a compiler that takes human-readable source code and turns that into BF, which in turn can be assembled into the binary format understood by Synapse-191.

The author has attempted to create such a language and compiler in the past: *BrainFix* [9] is a compiler that takes a C-style language and turns it into BF. This compiler works reasonably well but has some serious limitations:

- No (true) function calls: all functions are inlined. Recursion is therefore *not* possible.

¹Latin for *needle*, referring to the pointer in the BF computation model.

- Dynamic array accesses were implemented in a way that limits the maximum array size to 256 elements.
- Only 8-bit numbers are supported (assuming the BF tape stores 8-bit integers in its cells).
- No signed integers.
- No pointers or function pointers.

These limitations derive from naive choices during early development of the compiler, where the focus was on parsing and language design: features were bolted on rather than planned for in the early stages of development. In this second, more mature attempt to build a compiler, we started out by building a backend rather than a full compiler: *Acus*. *Acus* is a C++ library that exposes a set of elementary assembly-like instructions (like arithmetic operations, conditional expressions and jumps) that are compiled into BF. *Acus* is meant to be used as a compiler backend: a layer that accepts higher-level operations and emits BF code. A frontend can then link with *Acus*, parse an actual programming language, call the necessary *Acus* functions and produce the corresponding BF. *Acus* can also be used on its own, since its library functions were designed to feel a bit like an actual language, as can be seen from the *Hello World* snippet of Listing 3²

```

1  #include <iostream>
2  #include "acus/assembler/assembler.h"
3  using namespace acus;
4
5  int main() {
6      Assembler a;
7
8      a.program("hello", "main").begin(); {
9
10     a.function("main").begin(); {
11         a.print(literal::string("Hello, World!\n"));
12         a.returnFromFunction();
13     } a.endFunction();
14
15     } a.endProgram();
16
17     std::cout << a.brainfuck("hello") << '\n';
18 }
19

```

Listing 3: C++ code that calls into the *Acus*-library to produce an executable that outputs BF source code for a *Hello World* program.

7.3 Challenges and Design Goals

The main goal was to make *Acus* capable enough that a frontend for a language like C would have no problem implementing most of its features. These include:

²Note that using *Acus* in this way is unintended and inconvenient: the resulting binary is a program that, when run, prints a string of BF commands to the output. That string of commands then has to be written to a file, which can then be executed by a BF computer or interpreter. When the program is modified, the C++ source has to be recompiled and the resulting executable rerun to produce the updated BF source.

1. 8-bit and 16-bit integer data types;
2. signed integers (both 8-bit and 16-bit);
3. arrays and strings;
4. user-defined structured types (structs);
5. arithmetic operators (+, -, *, /, %);
6. logic operators (**and**, **or**, **not**, **xor**);
7. function calls and recursion;
8. pointers and pointer arithmetic;
9. labels and conditional jumps (**goto**);
10. global data.

Familiar features that were *not* implemented include:

1. floating/fixed-point numbers;
2. bitwise operators;
3. member functions;
4. control-flow mechanisms like **for**, **while**, **switch**, **if**.

Many of the above can be implemented in terms of the features provided by Acus; it was a deliberate choice not to make these part of the library in order for the library to remain as fundamental as possible.

Chapter 8

Acus Program Model

8.1 Basic Block Scheduling

In the Acus model, a program consists of a series of basic blocks, sequences of code that are always executed in order, and jumps between those blocks. Each basic block (B_i) is compiled into its corresponding BF instructions and wrapped in a BF loop pair (i.e. between [and]) whose entry is controlled by generated guard code G_i . The complete sequence of blocks is enclosed in a main loop, conditional on the value in a special BF cell, the *Run* cell, which can be reset by any block when the program should be terminated. Prior to entering the main loop, a *head* section H is responsible for the initial setup of the data tape and the tail code T is responsible for resetting the pointer to the *Run* cell before the closing bracket of the main loop is encountered:

$$H \left[G_1 [B_1] G_2 [B_2] \dots G_N [B_N] T \right]$$

The aforementioned guard code G_i compares the block index i to the value stored in another predefined cell: the *TargetBlock* cell. Only when this cell contains the exact index of the block *and* the *Run* cell is still nonzero is the loop entered and its code executed. Therefore, each block should at some point set the index of the next block to be executed to keep the program going (not setting a different *TargetBlock* will cause the same block to be executed ad infinitum). This approach basically implements a state machine that can be expressed in pseudocode as shown in Listing 4. Note that, on each iteration of the main loop, all blocks are scanned until the scheduled block is encountered. In a worst-case scenario, when block i is scheduled by block $i+1$ (the currently executed block), the system will first need to check all blocks from $i+2$ up to N before looping back to block 1 and eventually block i .


```

1   B = {B1, B2, ..., Bn}
2
3   Run = 1
4   TargetBlock = 1 # not necessarily block 1
5
6   while Run:
7       for i = 1 ... N:
8           if Run AND TargetBlock == i:
9               execute B[i]
10

```

Listing 4: Pseudocode of an Acus-generated program.

8.2 Frames and Initialization

Each function has its own chunk of memory that it operates on, called a frame. Calling a function has the effect of advancing the data pointer one cell beyond the end of the current function's frame. When a BF program is executed, the pointer starts at cell 0 in frame 0, which is dedicated to holding the global data (Section 9.4.2). Frame 1 starts just beyond the end of frame 0; the exact offset depends on the number of global variables and their sizes. The initialization code that runs prior to entering the main loop needs to set up both frame 0 and frame 1:

1. Set a marker (the **SeekMarker**, Section 9.2.5) at the start of the data-tape (cell 0). This marker can be used as a stop-sign during dynamic pointer movements to locate the start of the global data frame (and to prevent the pointer from moving into the (undefined) negative address-space).
2. Set another marker (the **FrameMarker**) that marks the start of frame 0 (the global data frame, Section 9.4.2). This might seem redundant as we already marked the start of this frame with a **SeekMarker**, but since every frame has a **FrameMarker**, the global frame needs it as well for generalized algorithms to work.
3. Calculate the size of frame 0 and move the pointer to the start of frame 1 (the frame that will be used by the function that is set as the entrypoint for the program).
4. Mark the start of frame 1 by setting the **FrameMarker** here as well.
5. Move the data pointer to the *Run* cell of frame 1 and set its value to 1.
6. While the data pointer points to the *Run* cell of frame 1, open the main loop (guaranteed to be entered as we just set this cell to 1).

8.3 Block Guards

The guards G_i compare the value stored in the **TargetBlock** cell to their index and check if the *Run* flag is still set. The results of these comparisons are ANDed together and used as the loop condition for the block that follows immediately after. Because the block index is known at compile time, it can simply be subtracted from (a copy of) the **TargetBlock** value. If the result is 0, this means we should enter the block given that the *Run* cell contains a nonzero (true) value. The logic necessary to implement these checks is discussed in more detail in Section 11.5. A simplified BF

algorithm that implements this logic could look like the snippet of Listing 5; the real algorithm is more complicated because it needs to support 16-bit block indices (otherwise each program would be limited to only 255 code blocks), which requires 16-bit arithmetic and logic routines (Section 11.3).

```

1  # Guard code G3: computes ‘r AND i == 3’
2  # r: cell that contains a copy of the run-flag
3  # i: cell that contains a copy of the target-index
4  # f: cell that will contain the guard-flag (result of r AND i == 3)
5
6  f [-]          # reset f
7  r [           # if run
8    f +         # set guard-flag to 1
9    i --- [     # subtract 3 from i, check if nonzero
10     f -        # still nonzero -> set flag to zero
11     i [-]      # exit i-loop
12  ]
13  r -          # exit r-loop
14  ]
15
16 # At this point, f is set only when r was set and i == 3
17 f [           # move pointer to guard-flag
18 ...          # user code (block 3)
19  ]

```

Listing 5: Simplified guard code for block 3 (G_3). The contents of block 3 will be executed only if **f** is nonzero.

8.4 Example: Fibonacci

To illustrate how a program is divided into basic blocks, a simple Fibonacci number generator is shown in Figure 8.1. It consists of three functions, each of which contains multiple basic blocks. The block IDs are shown in square brackets at the start of each block. Blocks end at control boundaries, i.e. places in the code where potential jumps take place, or explicitly at predefined labels.

The program starts execution in block 10, which is set as the entry point by the initialization code (note that the initialization code of Listing 4 sets the entry point to block 1; the specific example below requires the initialization sequence to set this value to 10 instead). A function call is immediately encountered, ending the current block. After initializing a new stack frame (this will be covered later, in Section 9.4), the *TargetBlock* is set to block 1: the entry block index of the callee. This will cause control flow to skip over block 11 and loop back to block 1, which will then be entered. When control hits the conditional statement of block 3, either block 4 or 5 will be set as the next target block, depending on the result of evaluating $i < n$. In this case, both the **if** and **else** bodies contain only a single block that immediately terminates in a jump. When control eventually reaches the end of **main**, the *Run* cell is set to zero, causing flow to exit the main loop and terminate the program.

```

function fib(a, b, n):
  i = 0 [1]
  loop:
    c = a + b [2]
    a = b
    b = c
    i = i + 1
    println(b)

  if i < n: [3]
    jmp loop [4]
  else:
    jmp done [5]

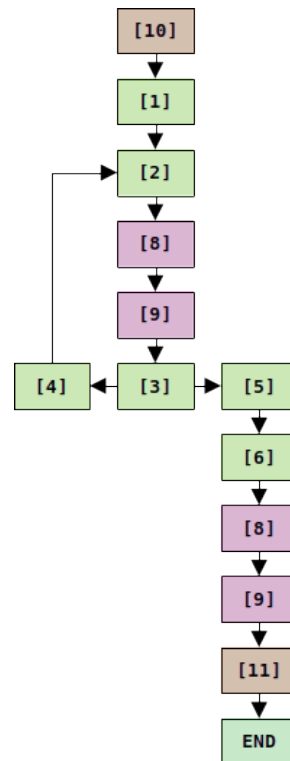
  done:
    println("done") [6]
    return [7]

function println(x):
  // code to print x [8]
  return [9]

function main():
  fib(0, 1, 10) [10]
  return [11]

```

(a) Fibonacci program



(b) Control flow

Figure 8.1: Figure (a) shows a simple Fibonacci program (unspecified syntax) and how that program is split into blocks. Figure (b) shows how control flows from one block to another.

Chapter 9

Tape Abstractions

9.1 Overview

The data-tape is considered a stack, where each function owns a stack frame. Each frame is composed of slots: a return-slot (if the function returns a value), argument-slots (if the function takes arguments) and local data-slots (variables and temporaries used in the function). A slot is a series of (logical) cells that contains a variable of some data-type, used by the function. The size of a slot depends on the number of cells required to represent the type of the variable. Instead of using single BF-cells to compose slots, logical cells or macro-cells are used. Each macro-cell contains 9 actual BF-cells, referred to as *fields* from now on, each of which serves a different purpose.

$$\text{BF Cell} \longrightarrow \text{MacroCell} \longrightarrow \text{Slot} \longrightarrow \text{Frame} \longrightarrow \text{Tape}$$

A macro-cell contains the following fields (in arbitrary order), each of which will be discussed in more detail in Section 9.2.

1. **Value0**: Low-byte of the semantic value of the macro-cell.
2. **Value1**: High-byte of the semantic value of the macro-cell.
3. **Scratch0**: General-purpose scratch-cell.
4. **Scratch1**: General-purpose scratch-cell.
5. **Payload0**: Low-byte of a payload (or another scratch-cell).
6. **Payload1**: High-byte of a payload (or another scratch-cell).
7. **Flag**: Dedicated flag cell to guard BF-loops.
8. **FrameMarker**: Dedicated cell to mark the start of a new frame.
9. **SeekMarker**: Dedicated cell to mark arbitrary locations on the tape.

Central to the macro-cells are the two value-fields that hold the actual value of the logical cell. By having two value fields, we allow for 16-bit data types to be implemented within the same logical-cell, rather than having to allocate two macro-cells for a single 16-bit variable. The other fields hold temporary data or metadata required to perform higher level functions like arithmetic, flow control and dynamic pointer movement. When the compiler needs to generate code that moves the BF-pointer to location known only at runtime (for example when dereferencing pointers or accessing array elements by a dynamic index), these fields can hold temporary values or metadata necessary to perform the move. Other non-value fields include temporary scratch-pads (heavily used when

performing arithmetic operations), payload-fields (used to transport data to locations known only at runtime) and a flag cell that is dedicated to the evaluation of loop-conditions. Having all these fields near one another on the BF-tape has the nice consequence of BF-algorithms being relatively short due to tight pointer-move-sequences.

Example: Fibonacci (continued)

Figure 9.1 shows the tape hierarchy described above, applied to the Fibonacci example from before and evaluated immediately after `fib` was first called. Even though this program made no use of any global data, this frame is still depicted for completeness' sake. Zooming in on Frame 2, used by the `fib` function, we see how a frame starts with the slots we previously encountered in Section 8.1: the **TargetBlock** and **Run** slots, which are used to determine where to jump next and whether to continue execution at all. Since `fib` does not return anything, no return slot has been allocated to this frame. Instead, the local data immediately follows: first the arguments passed to the function (initialized by the caller), then the local data used within the function. Only the variables `c` and `i` have been explicitly allocated by the programmer, but temporary slots have been created by the compiler to store the result of `a + b` and `i < n`.

Zooming in further on two of the slots, we see that they both contain only a single macro-cell and its 9 constituents. The first slot of the frame (**TargetBlock**), which still holds ID 1 (the block-ID of the first block of the function) in its value-field, also has its **FrameMarker** field set, indicating that this is where a new frame starts on the data-tape. The `b`-slot has its value-field set to 1 as well, corresponding to the value of `b` that was passed to `fib` from `main`.

The sections below go into more detail about each of the layers of abstraction.

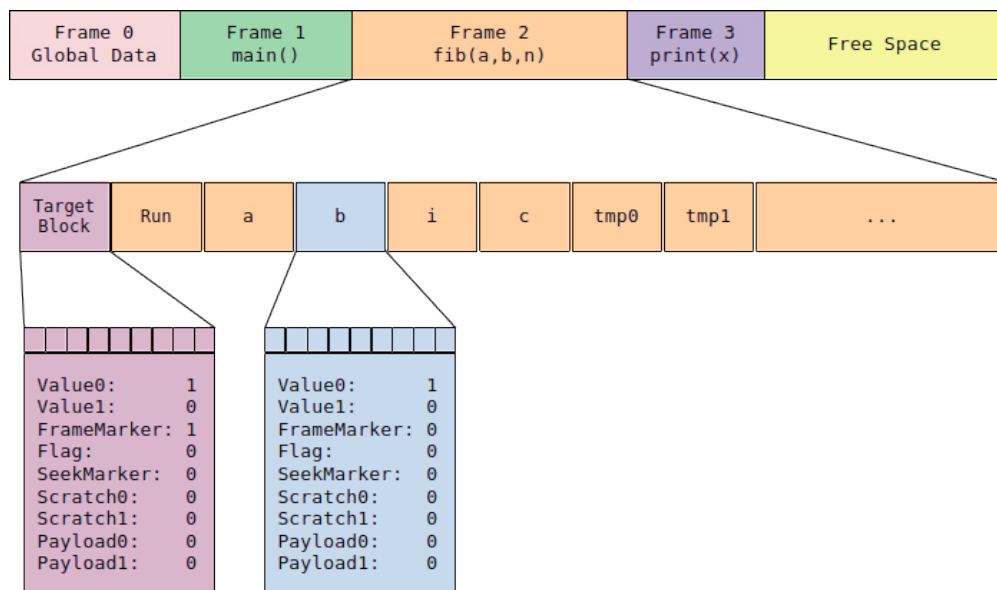


Figure 9.1: Acus tape hierarchy: the tape is divided into frames, which are divided into slots, which consist of macro-cells containing 9 fields (BF-cells).

9.2 Cells and Macro-Cells

BF operates on the most fundamental level of raw cells. Each consecutive sequence of 9 such cells constitutes a logical macro-cell as mentioned before in Chapter 9. In this section, we will cover

each of the 9 fields of a macro-cell in more detail and cover some of their use cases.

9.2.1 Value0 and Value1

The value fields together hold the semantic value of the macro-cell. The **Value0** field contains the low byte and the **Value1** field holds the high byte of an integer of at most 16 bits. Acus implements four integer data types using these two fields: unsigned 8-bit (**u8**), unsigned 16-bit (**u16**), signed 8-bit (**s8**) and signed 16-bit (**s16**). When referring to 8-bit values, the **Value1** field is simply ignored. Signed integers are represented using two's complement; some algorithms need to do some additional work (like evaluating the sign bit) before applying the usual operations.

9.2.2 Scratch0 and Scratch1

Many BF algorithms need temporary cells as scratch-pads. These cells can be temporarily claimed by an algorithm and must be left behind empty for the next algorithm to use. An example of a simple algorithm that makes use of these values is the non-destructive addition. Say we want to add the contents of cell **x** to those of cell **y** without destroying **x** in the process. We can achieve this by decrementing **x** while at the same time incrementing **y** until **x** becomes zero. However, in order to restore **x** to its original value, we need to use a scratch-pad to construct a copy of **x** which is moved back after the addition has been completed. This algorithm is shown in Listing 6, where instead of denoting pointer-movement with sequences of **<** and **>**, we simply write the name of the variable/slot to indicate a move to that (known) location on the tape.

```

1  # Algorithm for y += x
2  # tmp is a scratch-cell assumed 0
3
4  # Step 1: add x to both y and tmp, destroying x
5  x[- y+ tmp+ x]
6
7  # Step 2: add tmp back to the now empty x
8  tmp[- x+ tmp]
```

Listing 6: Non-destructive addition of one cell to another, using a temporary scratch cell.

9.2.3 Flag

The **Flag** field is reserved to store a value that determines whether or not a BF-loop should be entered. An algorithm can store an intermediate, temporary, value in the flag-cell and use that as a branch or loop-condition. Whereas scratch-cells are general purpose temporary storage locations, the flag-cell is meant to be used for this purpose alone to avoid confusion. The subsection below includes an example that also utilizes this field in Listing 7.

9.2.4 FrameMarker

The **FrameMarker** field is only set when the macrocell marks the start of a new frame. When performing dynamic pointer movements, a marker like this (the **SeekMarker** is used in a similar way) can be used to move from a still unknown to a known location on the tape. Let's say we know the data-pointer is somewhere in Frame 2, but we don't know where because we just performed some (dynamic) movement (i.e. an algorithm that moves the pointer by an amount that depends

on runtime data). We know that the start of the frame is to the left of the current position, so we can perform the algorithm below and be sure that we're pointing at the start of the frame when the algorithm is finished. This algorithm is shown in the pseudo-BF of Listing 7. This works because every macro-cell has the same width and because exactly one `FrameMarker` is set at the start of the active frame.

1. Move the pointer to the `FrameMarker` field of the current cell. Even though the current offset is unknown, the current field *is* known so we are able to switch between fields reliably.
2. Construct the logical NOT of the `FrameMarker` and store the result in the `Flag` field, then move the pointer to the `Flag` field.
3. Enter a loop, conditioned on the flag, where the pointer is moved 9 cells to the left on each iteration until a non-zero `FrameMarker` is encountered. In each iteration, the logical NOT has to be constructed and stored in the `Flag` field as before. Note that when inside a loop (meaning the flag was nonzero), the flag needs to be cleared before moving to the adjacent cell.

```

1  # Algorithm to move the pointer back to the start of the frame
2  # Current macro-cell offset is unknown; all moves are relative
3  # field-switches within the same macro-cell, except for the
4  # 9 left-moves, which take us exactly one macro-cell to the left.
5
6  # Assumed: there are already implementations for assignment and
7  #           the logical NOT operator.
8
9  ASSIGN(Flag, NOT(FrameMarker))
10 Flag [-
11     FrameMarker <<<<<<<<<      # Reset flag before moving away from it
12                                     # Move to the frame-marker field of
13     ASSIGN(Flag, NOT(FrameMarker)) # neighboring macro-cell
14     Flag                               # Compute exit-condition
15 ]                                     # Check exit-condition
16
17 # Out of the loop: pointer is at the start of the frame
18
```

Listing 7: Algorithm to seek left to the start of the current frame from an unknown initial position.

9.2.5 SeekMarker

The purpose of the `SeekMarker` field is very similar to that of the `FrameMarker` field, in the sense that it is used to mark certain cells to make dynamic movement possible. For example, a known location can be marked before performing a dynamic move, in order to be able to move back to that known location using an algorithm similar to that of Listing 7. Also, when a dynamic move has been completed, its location can be marked to make future excursions to this location more efficient. Contrary to the `FrameMarker`, which is specifically used to mark the start of a frame, the `SeekMarker` field may be set for any macro-cell on the tape.

9.2.6 Payload0 and Payload1

The payload fields are used to store values that need to be transported along to dynamically determined locations. Algorithms that move the pointer to such locations can be trivially extended to move data along using these fields: one for the low byte and one for the high byte of the value. Additionally (but this is not the intended use-case), a payload cell may be used as a scratch-pad (when an algorithm has exhausted the available scratch cells), provided it is not being used to transport any values at that time.

9.3 Slots

A slot is a storage location for one object. Single 8 or 16-bit integers occupy a single macro-cell, while larger objects like arrays or structs occupy a sequence of macro-cells. A slot is bound to a stack frame; its offset is defined as the index of its first macro-cell with respect to the start of the frame, not the start of the actual BF data-tape.

In addition to a name and offset, a slot holds information about its scope (to determine access, allow for name-shadowing and keep track of lifetime) and its *kind*. Most slots are of the *local* kind, representing data local to the current function, but slots can also be tagged *global*, *available* (representing a free block of memory that was previously occupied), *temp* (for temporary slots that should be freed after use) and *cache* (Chapter 12). Figure 9.2 shows a section of a frame and how local slots could be organized given their data-types and sizes. Note that the size of a slot is expressed in terms of the number of macro-cells rather than raw BF cells.

x: u8 offset = 3 size = 1	x: u16 offset = 4 size = 1	arr: array<u8, 10> offset = 5 size = 10	p: Point<u8, u8> offset = 15 size = 2
---------------------------------	----------------------------------	---	---

Figure 9.2: Slots represent variables, which may span multiple macro-cells.

9.3.1 Data Types

Slots reserve space on the tape, but the compiler also needs to know how that space should be interpreted. This is the role of data types. A data type determines how many macro-cells are needed to store a value, which sub-slots exist inside that value, and which algorithms may be used to copy, compare, increment or otherwise manipulate it.

Primitive integer types (such as `u8` or `s16`) are stored directly in the value fields of a single macro-cell (even 16-bit integers fit in a single logical cell). Compound types (arrays and structures) are constructed from these primitives. An array is represented as a sequence of equally sized elements, while a structure is represented as a sequence of fields at fixed offsets. As a result, the compiler can compute the size of every type and the offset of every field at compile-time.

Type	Size (# macro-cells)	Description
u8	1	Unsigned 8-bit integer stored in one value field.
s8	1	Signed 8-bit integer, stored using the same physical layout as u8.
u16	1	Unsigned 16-bit integer stored across both value fields of a macro-cell.
s16	1	Signed 16-bit integer, stored using the same physical layout as u16.
array<T, N>	$N \cdot \text{sizeof}(T)$	Fixed-size sequence of N values of type T, stored contiguously.
string<N>	N	Fixed-size character buffer, equivalent in layout to an array of N byte-sized characters.
ptr<T>	2	Pointer to a value of type T. The type parameter determines how dereferenced storage is interpreted. The first cell contains the FrameDepth of the pointer, i.e. how many frames back the pointee is actually stored. The second cell stores the offset of the pointee within its frame. Refer to Section 11.6 for more details on pointers.
struct-defined type	sum of field sizes	User-defined aggregate type. Fields are stored consecutively at statically known offsets.

Table 9.1: Overview of the data types supported by Acus. Sizes are measured in macro-cells rather than raw Brainfuck cells.

9.3.2 Slot Proxies and Materialization

Most expressions encountered by the compiler can be resolved directly into a slot. For example, assume **x** is an array of structs that contain an **age** field, e.g.

```

1  struct Person {
2      id:  u16
3      age: u8
4  };
5
6  let x: array<Person, 5>;

```

In this context, the expressions **x**, **x[2]** and **x[2].age** all refer to known slots or sub-slots (assuming square brackets represent array indexing and a dot represents field access for struct objects). The compiler knows in which slot the array **x** is stored, how to find the offset of element 2 and how to find the offset of the **age** field.

However, when the index is not a compile-time constant, things change. The expression **x[i].age** has the same shape as **x[2].age** (they might even refer to the same data), but now the compiler has no way to compute the desired offset at compile-time. At this point, the expression can no longer be represented by a single statically known slot. The array itself still has a known base location, and the layout of each **Person** object is still known, but the particular element that should be accessed depends on a value computed at runtime. In Brainfuck terms, the compiler must now generate code that moves through the tape dynamically, using the value of **i** to reach the correct element before applying the fixed offset of the **age** field.

This is where Acus introduces the notion of a *slot proxy*. A slot proxy is not necessarily a slot by itself, but a description of how a value can be accessed. Some proxies are direct: they simply refer to an already known slot. Others are indirect: they describe an access path that must be resolved at runtime. The expression `x[2].age` can be reduced to a direct sub-slot, while `x[i].age` becomes a proxy that says, in effect: start from the slot occupied by `x`, use the runtime value of `i` to select an element, and then access the `age` field within that element. In general, modifications made to that element (e.g. `x[i].age += n`) will need a read-modify-write cycle where the element is first *materialized* into a known, temporary slot (much like a hardware register), modified and written back using a dynamic write-algorithm. This process of materialization and write-back is shown conceptually in Figure 9.3.

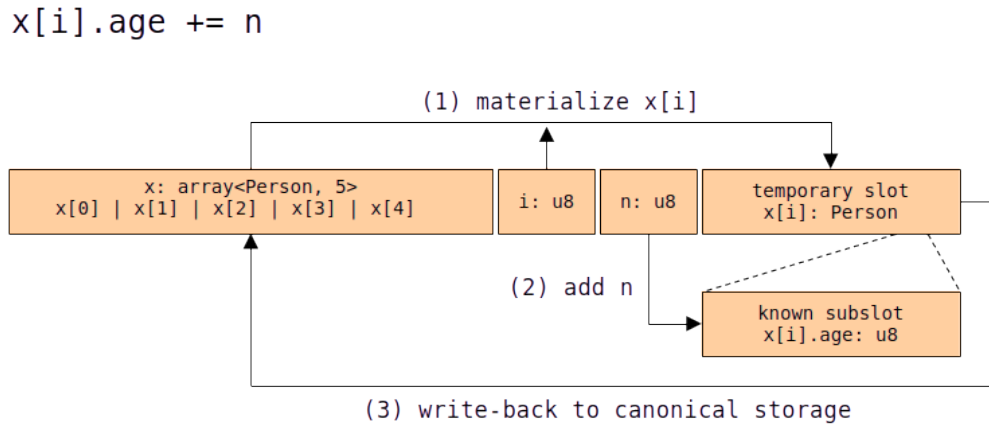


Figure 9.3: Materialization process for a modification to a dynamically indexed array.

The distinction between actual slots and proxies that represent them allows the compiler to treat simple variables, struct fields, array elements and pointer dereferences in a uniform way. Code that needs to read from or write to an expression does not have to know immediately whether the target is a fixed slot or a dynamically computed location. It can ask the proxy to materialize the value into a concrete slot, or to write a concrete value back through the described access path.

For the purposes of this chapter, slot proxies should therefore be understood as compiler-level access paths. They bridge the gap between the static memory layout known at compile-time and the dynamic addresses that sometimes have to be computed at runtime. This idea becomes especially important in the caching system (Chapter 12), which is designed to limit unnecessary materialization and synchronization events.

9.3.3 Local Memory Management

Declaring a local variable means reserving a slot inside the current function frame. The slot is assigned a type, a size and an offset relative to the start of the frame. From that point onward, the variable name refers to this slot until it goes out of scope. Since the offset is fixed, accessing a local variable does not require a runtime search through the tape.

Temporary slots are allocated in the same frame when the compiler needs short-lived storage. This happens, for example, when evaluating intermediate expressions, materializing a slot proxy, or running algorithms that require additional scratch space. Unlike named locals, temporaries are not part of the source-level program and may be released as soon as the generated code no longer depends on their contents.

When a slot is freed, either because a local variable leaves scope or because a temporary can be dismissed, its region is marked as available. The contents of the underlying macro-cells are not necessarily cleared immediately; freeing a slot only means that the compiler is allowed to reuse that region for later storage. When a new slot is allocated, a sufficiently large available region may be split into an occupied part and a smaller remaining available part. Conversely, when a slot is freed, neighbouring available regions are merged back together. This prevents the local frame from becoming unnecessarily fragmented and allows larger slots to be allocated later when enough adjacent macro-cells have become free. This management system is discussed in more detail in Chapter 10.

9.4 Frames and Calls

A frame groups all data belonging to one function activation. Each frame contains the local control cells, optional return storage, arguments, named locals and compiler-generated temporaries (and cached values). Function calls can then be viewed as transitions between frames: the caller prepares the callee frame, moves the data-pointer to the start of the frame and sets the **TargetBlock** value to the callee's entry block. When a function returns, the data-pointer is moved back to the caller's frame and execution continues at the first block beyond the call. Each of these steps will be described in more detail in the sections below.

9.4.1 Reference Frames

Each stack frame acts as a frame of reference, or coordinate system, to the data-pointer. Since function calls happen dynamically, stack frames are allocated dynamically as well. This means that the compiler has no way of knowing where the data-pointer is located in an absolute sense (i.e. with respect to the start of the BF-tape). Instead, it keeps track of the pointer-location with respect to the start of the current frame. Pushing a new frame on top of the stack and moving the pointer to the start of this frame can then be seen as performing a coordinate transformation—or rebase—of the pointer: the origin is now in the frame of the callee rather than the caller.

9.4.2 Global Data Frame

Global data is allocated in its own frame at the very start of the data-tape, i.e. frame 0. This frame is not owned by any function and is accessible from any other frame to fetch data from or write to. Like dynamic array accesses or pointer dereferences, a global slot is accessed through a slot-proxy and will be materialized locally before any work can be done on it. This introduces synchronization issues across call boundaries, which will be addressed in Chapter 12. In the sections below, the concept of (stack) frames refers to frames actually tied to the functions being executed, not the global frame.

9.4.3 Frame Layout

The exact layout of a frame depends on the nature of the function that it belongs to, depending on the return-type and number of arguments it takes. However, each instance of a function-call will cause an identical frame-layout to be generated as a result. In general, a frame is laid out as follows (in terms of macro-cells):

1. **TargetBlock**: a macro-cell whose **Value** fields are set to the index of the next block scheduled for execution and whose **FrameMarker** is set to 1, indicating the start of a new frame.

2. **Run**: a macro-cell dedicated to storing a local copy of the program's *run*-state. As soon as this flag is set to 0 (from any frame), none of the blocks will execute and the main loop will be aborted.
3. **ReturnSlot**: Depending on the return-type of the function, a slot is allocated to hold the return-value. Note that this slot is located at a fixed offset, so it can be reached easily even from within the caller's frame in order to fetch the return-value. If the function returns **void**, this slot does not exist at all.
4. **Locals**: Data local to the frame, which can be categorized further:
 - (a) local variables (including function arguments), organised in slots as discussed before in Section 9.3.
 - (b) compiler-generated temporaries
 - (c) cached materializations

9.4.4 Calls, Returns and Meta-Blocks

Calls To call a function, the caller has to perform the following actions:

1. Set the caller's **TargetBlock** to the ID of a compiler-generated meta-block. This is the continuation that will run after the callee returns (described in the next paragraph).
2. Copy or materialize the function arguments into the argument slots of the next frame. The offsets are known relative to the caller's frame, so this does not require a runtime search.
3. Initialize the control cells of the callee frame: its **Run** cell is set and its **TargetBlock** is set to the callee's entry block.
4. Move the data pointer to the start of the callee frame and treat this position as the new origin.

After these steps, the runtime naturally dispatches to the block whose ID is now stored in the **TargetBlock** cell of the freshly initialized stack frame.

Meta-blocks These are compiler-generated code-blocks that are scheduled at the time of the call, while still in the caller's frame. When the pointer is popped back to this frame after the callee returns, this is the value that is observed and used to determine the next block to be executed. This block is then responsible for making sure that

1. the return value is fetched from the callee's frame and copied into the return-slot of the caller (if provided; it may also be ignored),
2. the state of the *run*-flag is copied back into the caller's frame (to propagate its state in case the callee changed it), and
3. the **TargetBlock** cell is set to the ID of the block following the call.

Propagating the *run*-flag like this makes early termination compositional: if a callee stops the program, the caller observes the same stopped state after the return metablock has run. This allows a (builtin) function like **abort()** to be implemented.

Returns To return control to the caller, we only need to make sure that the return-slot is populated appropriately and pop the data-pointer back to the frame below. The meta-block has already been scheduled and will take control immediately, finishing the transaction as described above.

Figure 9.4 shows data transfers and the setup that occurs while performing a call and returning from it. At the call-site, the target-blocks are set and arguments are copied over to the next frame. When returning and control has been passed to the meta-block, return-data is copied back from the callee’s frame into the current one.

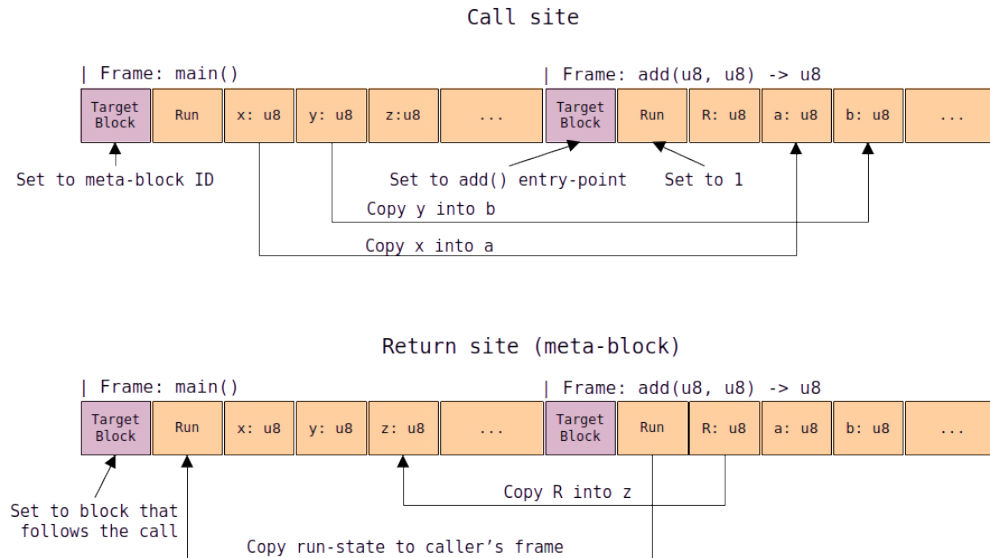


Figure 9.4: Conceptual overview of data-tape manipulations that occur when calling a function and returning from it. This graphic assumes that in between these two moments, the `add` function has populated the return-slot (`R`).

Notice that the return address is not stored on a hardware stack. It is encoded as a block ID in the caller’s `TargetBlock` cell. Returning to the caller therefore means restoring the caller frame; the scheduler will then execute the metablock whose ID was already waiting there.

Chapter 10

Memory Management

10.1 Overview

Whenever a variable is declared in Acus, the allocator will look for available space in the current stack-frame. From the allocator's point of view, named locals, compiler temporaries and cache slots are all typed slots inside the current frame. They differ only in who requests them and when they are released.

This allocation can always succeed in the compiler model, because Acus assumes an unbounded Brainfuck tape and therefore allows frames to grow as needed. Frame sizes are only evaluated when the program has been fully defined and algorithms are retroactively adjusted to fit the final configuration. However, slots may also be recycled to keep the frames as tightly packed as possible; this is important to reduce the number of pointer movements. To accomplish this, the management system provides the following features:

1. Freed slots are marked **Available** and can be reused.
2. Declarations can be scoped to reduce the lifetime of variables. Variable slots will become **Available** when the variable goes out of scope.
3. Temporary slots declared by the compiler are freed immediately after use.
4. Adjacent **Available** slots are merged to accommodate larger data types.
5. **Available** slots will split into smaller parts when a variable with a size smaller than that of the slot is allocated to it.

10.1.1 Relative Offsets

The memory management system assigns each slot an offset relative to the start of its stack-frame. Absolute tape positions are not known at compile-time, because the same function frame may appear at different runtime locations on the tape. The compiler only needs to know the layout of one frame; generated movement algorithms are later adjusted to the final frame size.

10.1.2 Free vs Clear

When a slot is freed, its contents are not cleared. This means that no algorithm may assume that a newly allocated slot contains zero, even if that slot has just been created from previously available space. In particular, the **Value** fields of a slot must be treated as unknown until explicitly initialized. Freeing a slot is therefore purely a compiler-side bookkeeping operation and does not incur any runtime cost.

10.1.3 Heap Allocation

While technically possible, Acus does not support heap allocation. This would require a separate heap frame and a runtime system that keeps track of the memory layout in that frame (including deallocation, block splitting and block merging). The runtime overhead that would arise from managing the heap at runtime would be so costly that it was decided not to implement this as a feature. However, a front-end might emulate heap allocation using the global frame. One could certainly write algorithms on top of Acus to reserve a specific amount of raw data in the global frame and manage it using existing features.

10.2 Splitting and Merging

10.2.1 Slot Splitting

When a variable of type T is allocated, Acus looks for the first slot that is marked **Available** and is of size $\geq$ the size of T (first fit strategy, [11]). If the sizes match perfectly, the process ends here. If the available slot's size exceeds the variable size, the remaining part of the slot is split off into its own available slot (Figure 10.1). If no available slots can be found at all, a new slot is allocated at the end of the current frame (increasing the eventual frame-size).



Figure 10.1: Available slots are split when a smaller variable is allocated to them.

10.2.2 Slot Merging

Named local variables live until the end of their lexical scope. Temporaries usually live only for the duration of one generated algorithm. Cache slots live until their cache entries are flushed, invalidated, or discarded at a boundary. All three are returned to the same pool of available slots. In each of these scenarios, the slot is marked **Available** by the memory management system and can be used to accommodate other data. Often, this will result in adjacent **Available** slots, which can be merged as shown in Figure 10.2.

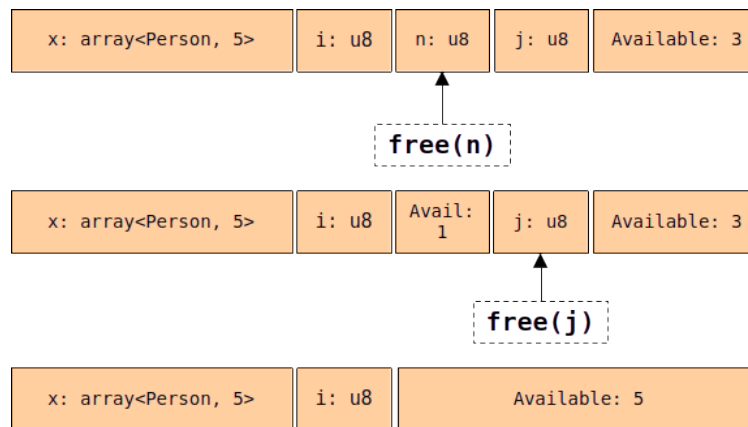


Figure 10.2: Adjacent available slots are merged.

Chapter 11

Code Generation

Before discussing larger compiler structures such as caching and dynamic access, it is useful to look at the small Brainfuck idioms from which Acus builds its generated code. Many of the algorithms in this chapter are simple in isolation, but together they form the vocabulary used to express higher-level operations on the tape. Moving or copying values, constructing constants or adding numbers together are all small patterns that appear repeatedly in the generated output.

This chapter is therefore not meant to be a complete catalogue of every algorithm implemented by Acus. Instead, it presents a representative selection: the basic destructive operations that make Brainfuck programming work, a few arithmetic operations, and the dynamic pointer-movement techniques that are most specific to compiling higher-level code to Brainfuck.

11.1 Clearing, Moving and Copying Values

One of the simplest useful BF idioms is destructive movement: consume one cell while adding its value somewhere else. From that, one can build addition and non-destructive copying by introducing temporary storage. Basically all other algorithms rely on repeated applications of these idioms.

11.1.1 Clearing Cells

A cell can be cleared simply by moving the pointer onto it and repeatedly decrementing its value until it becomes zero: `[-]`. If the value is known to be larger than 128, `[+]` may be faster at runtime, since the cell reaches zero by overflowing rather than by decrementing all the way down.

11.1.2 Moving Data

The contents of a cell can be moved to another by decrementing its contents while incrementing that of the other. For example, to move the contents of cell `x` into cell `y` (leaving `x` empty):

```
1  # MOVE(x, y)
2  y[-]
3  x[- y+ x]
```

11.1.3 Copying Data

When the contents of a cell must be copied rather than moved, we need a temporary cell. The value stored in `x` is then simultaneously moved into both `y` and the temporary scratch cell `s`. When

the move completes, **s** is moved back into **x** to restore its value. The scratch cell **s** is assumed to be zero before and after the operation.

```

1  # COPY(x, y)
2  # Assumes s is an empty scratch cell
3
4  y[-]
5  x[- y+ s+ x]
6  s[- x+ s]
```

11.2 Constant Factories

Constructing known values is fundamental to creating meaningful programs. Even a simple *Hello World* requires the compiler to construct those characters somewhere in memory. The simplest and most naive way of setting a BF cell to some known value, e.g. **n**, is to concatenate **n** plus-commands. For example, the expression **x = 20** can be compiled to:

```

1  # SET(x, 20)
2  x [-] ++++++ ++++++ # 23 characters + initial move sequence
```

Notice that the cell containing **x** is first cleared using **[-]**, since we generally cannot assume that it already contains zero (even if it was freshly allocated). This particular example adds 23 characters to the generated BF output (excluding the initial move); larger values obviously produce more characters. The number of characters needed to set a cell to some constant value *c* is simply:

$$N_1(x \leftarrow c) = 3 + c + m(x) \quad (11.1)$$

where $m(x)$ is the number of move-commands necessary to reach the slot of **x** from the current pointer position.

To reduce code-size, we can use a scratch-cell to implement a multiplication of two smaller numbers instead. For example, instead of a series of 20 plus-commands, we can have the compiler multiply 4 and 5 to get the same result at the expense of having to use a scratch-cell **s**. Since scratch cells are allocated in a known-zero state and restored to zero after use, there is no need to clear **s** before this sequence.

```

1  # SET(x, 20)
2  # Assumes s is an empty scratch cell
3
4  x [-]
5  s +++++ [
6      x +++++
7      s -
8  ]
```

Instead of 23 characters, this algorithm adds only 15 non-movement characters to the program. In general, when *c* can be factored into *a* and *b*, the number of BF characters necessary to construct this value is:

$$N_2(x \leftarrow ab, s) = 6 + a + b + m(x) + 3m(x, s) \quad (11.2)$$

where $m(x, s)$ is the number of move-commands necessary to move between the scratch-cell **s** and the target cell **x**. This value depends on the exact layout of a macro-cell but can be as low as 2 when

the scratch-fields are right next to the value-fields. In the case where $m(x, s) = 2$, the difference between both methods for this specific example is

$$N_1 - N_2 = 3 + 20 + m(x) - (6 + 4 + 5 + m(x) + 3 \cdot 2) \quad (11.3)$$

$$= 23 - 21 = 2. \quad (11.4)$$

In this small example the saving is only two characters, while the runtime cost increases because the generated loop must be executed. Luckily the benefit from the second method increases along with the value of c . Acus has a precomputed table of factors for all numbers up to 128 (listed in Appendix F). Larger values can be reached by decrementing using the minus command to come down from 255 rather than up from 0 (BF cells wrap around after being decremented beyond 0). Each factorization $a \times b$ is chosen in such a way that it minimizes the sum $a + b$ (and consequently the number of output characters). In some cases (e.g. when c itself is prime), it is better to factor a number close to c and correct for the difference after doing the multiplication. For example, the optimal way to construct the value 31 is to first construct 30, then add 1, i.e. $31 = 5 \times 6 + 1$ for a total cost of 24 characters (as opposed to 34) for a gain of 10 characters per construction. Figure 11.1 shows the number of characters generated for every constant up to 128 (the graph would be symmetric around this number when going from 129 to 255), for both methods. Acus implements both the naive and factorizing method and picks whichever produces the least amount of code.

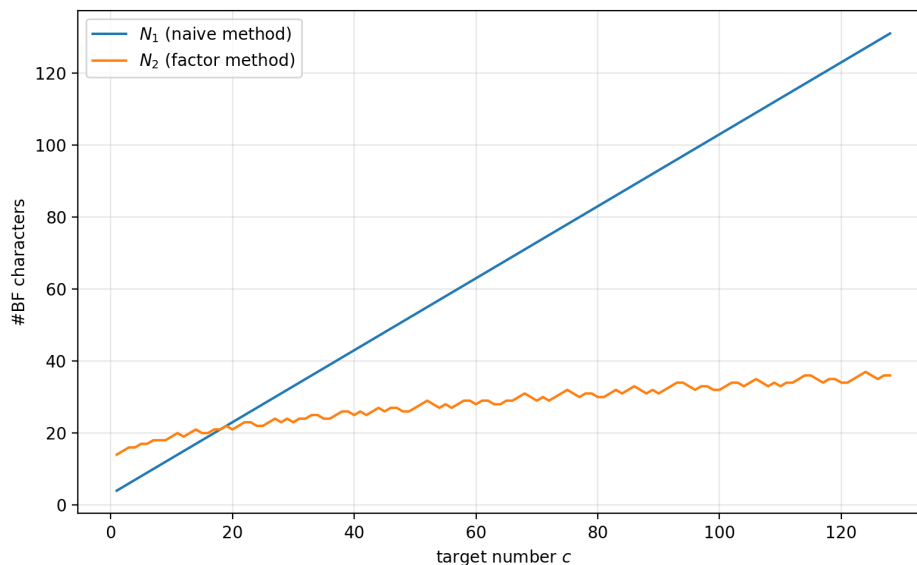


Figure 11.1: Cost of generating constants for both methods, expressed as the number of generated BF characters, as a function of their values. The common term $m(x)$ from equations 11.1 and 11.2 is set to zero for the purpose of this comparison.

11.3 Addition and Subtraction

11.3.1 Addition (8-bit)

The algorithm for addition is much like that of copying, the difference being that the target cell is never cleared in advance. When the right-hand side (rhs) of an expression of the form $x += y$ may

be destroyed, the algorithm is very short and efficient:

```
1  # DESTR_ADD_TO(x, y)
2  y[- x+ y]
```

The non-destructive version requires a scratch cell to hold a copy of the rhs operand, which can then be destroyed in the process.

```
1  # ADD_TO(x, y), preserving y
2  # Assumes s is an empty scratch cell
3
4  COPY(y, s)
5  DESTR_ADD_TO(x, s)
```

11.3.2 Addition (16-bit)

Adding 16-bit numbers together is much more involved and no BF pseudocode will be listed for this reason. Instead the higher-level pseudocode of Listing 8 illustrates the process. First, the low bytes are added together as usual. If the resulting value is less than its original value, this means that the cell has overflowed and we need to carry a 1 to the high byte. This requires multiple scratch-cells to hold temporary copies and helpers. Moreover, we need to be able to do a less-than comparison and branch on the result to carry if necessary. The implementation of this operator is discussed below in Section 11.5.6.

```
1  # 16-bit version of x += y
2  xLowBefore = low(x)
3  low(x) += low(y)           # normal 8-bit add
4  if low(x) < xLowBefore:
5      high(x) += 1           # carry on overflow
6  high(x) += high(y)         # normal 8-bit add
```

Listing 8: High-level pseudocode for 16-bit addition with carry propagation.

11.3.3 Subtraction

The subtraction algorithms are symmetric to the ones for addition, where increments are substituted for decrement operations and instead of checking for overflow, we check for underflow in the 16-bit case (i.e. whether the result has grown after doing the 8-bit subtraction on the low byte).

11.3.4 Signed Integers

For signed integers, no special algorithms have to be used at all. Since signedness is encoded using two's complement representation, addition and subtraction already work out of the box. Signedness becomes more important for comparisons, where the same bit pattern may represent a different ordering depending on how it is interpreted (as signed or unsigned).

11.4 Multiplication and Division

11.4.1 Multiplication (8-bit)

Multiplication can be implemented in terms of repeated addition, much like we already did in the constant factory optimization. To evaluate the expression `x *= y`, we do the following:

1. Make a copy of the operand `x`.
2. Clear `x`.
3. Repeatedly decrease `y` while adding the original value of `x`, stored in the copy, to `x` itself.

The steps above are the generalized version of the algorithm, which can be optimized further for edge cases where `y = 0`, `y = 1` or even for powers of 2.

```

1  # DESTR_MUL_BY(x, y)
2  # Assumes s is an empty scratch cell
3
4  COPY(x, s)
5  x [-]
6  y [-
7    ADD_TO(x, s)
8  ]

```

Like before, this algorithm destroys the rhs operand in the process. A non-destructive version can be written in terms of the destructive version:

```

1  # MUL_BY(x, y)
2  # Assumes s is an empty scratch cell
3
4  COPY(y, s)
5  DESTR_MUL_BY(x, s)

```

11.4.2 Multiplication (16-bit)

A naive 16-bit algorithm can simply follow the exact same procedure by relying on 16-bit addition. However, this leads to a lot of additions for large factors, which can slow down the runtime significantly. Instead, we can perform a small series of 8-bit multiplications and additions to get the same result, realizing that the 16-bit number is stored as a 2-digit base-256 number. That is, every 16-bit integer x can be written as

$$x = x_0 + 256 \cdot x_1, \quad (11.5)$$

where x_0 and x_1 are the low and high bytes of x respectively. Multiplication then becomes:

$$z = x \times y = (x_0 + 256x_1)(y_0 + 256y_1) \quad (11.6)$$

$$= x_0y_0 + 256 \cdot (x_1y_0 + x_0y_1) + 256^2 \cdot (x_1y_1) \quad (11.7)$$

The final term only contributes to bits beyond the lowest 16 bits, so it can be discarded when the result is stored as a 16-bit value. This means that the low and high byte of the product z can be

expressed as

$$z_0 = x_0y_0 \bmod 256 \quad (11.8)$$

$$z_1 = \left(x_0y_1 + x_1y_0 + \left\lfloor \frac{x_0y_0}{256} \right\rfloor \right) \bmod 256 \quad (11.9)$$

Therefore, the 16-bit product can be computed using three 8-bit multiplications. The low byte of the result comes from the low byte of x_0y_0 , while the high byte receives the carry from this product together with the low bytes of the two cross-products.

```

1  p0 = low(x) * low(y)
2  p1 = low(x) * high(y)
3  p2 = high(x) * low(y)
4
5  low(z) = low(p0)
6  high(z) = low(p1 + p2 + high(p0))

```

11.4.3 Division (8-bit and 16-bit)

Unsigned division is implemented using repeated subtraction. Given two values x and nonzero y , the goal is to compute a quotient q and remainder r such that

$$x = qy + r, \quad 0 \leq r < y. \quad (11.10)$$

The simplest way to do this is to repeatedly subtract y from a running remainder until doing so would make it negative. The number of successful subtractions is the quotient, while the value left behind is the remainder.

For 8-bit unsigned values, this can be expressed by first copying the dividend into r and clearing q . Then, as long as r is greater than or equal to the divisor, the divisor is subtracted from r and q is incremented. In high-level pseudocode:

```

1  # unsigned 8-bit division: x / y
2  q = 0
3  r = x
4
5  while r >= y:
6      r -= y
7      q += 1

```

The actual BF implementation cannot test $r \geq y$ directly. Instead, this comparison is implemented using scratch cells: temporary copies of r and y are decremented together until one of them reaches zero. If the copy of y reaches zero first or at the same time as r , the subtraction is safe and the loop may continue. If the copy of r reaches zero first, the next subtraction would underflow and the algorithm terminates. The original values must either be preserved during this test or reconstructed afterwards, which makes division considerably more expensive than addition or multiplication.

The same idea also applies to 16-bit unsigned division. The quotient is repeatedly incremented while the divisor is subtracted from a 16-bit remainder. The only difference is that the comparison and subtraction are now multi-byte operations, using the borrow and comparison mechanisms described earlier. This makes the algorithm correct but relatively slow for large quotients, since the number of loop iterations is proportional to the resulting quotient.

The generated implementation handles a zero divisor separately according to the rules in Section 11.4.5.

11.4.4 Signed Multiplication and Division

Signed multiplication and division are implemented as thin layers around the corresponding unsigned algorithms. First, the sign bit of the left-hand side is inspected. If it is set, the value is negated in place and a temporary flag is set to indicate that the final result should be negative. The right-hand side is copied into temporary storage before its sign is inspected, so the original divisor or multiplier does not have to be modified. If this copy is negative, it is negated as well and the result-sign flag is toggled. At this point, both operands have been converted to positive values, while the temporary flag records whether the original operands had opposite signs.

The actual multiplication or division can then be performed using the unsigned algorithm. After this operation has completed, the result-sign flag is checked. If it is set, the result is negated back into two's complement form. This approach avoids having to implement separate signed versions of the core multiplication and division algorithms; signedness is handled only by pre-processing the operands and post-processing the result.

When division also produces a remainder, Acus follows the same convention as C++ integer division: the quotient is truncated toward zero and the remainder has the same sign as the dividend. In other words, for x / y and $x \% y$, the generated code preserves the relation

$$x = (x/y) \cdot y + (x\%y), \quad (11.11)$$

where $x \% y$ follows the sign of x . This convention was chosen because it is familiar from C++ and maps naturally onto the implementation: expressions evaluated at compile-time (constant-folding) using the C++ `%` operator will therefore produce the same result as the runtime.

11.4.5 Division by Zero

To deal with division by zero, Acus will simply fill the result with the maximum value allowed by the width of the type: 255 for `u8` or 65535 for `u16` unsigned integers. For signed integers, the same bit-representation will be used, resulting in (a somewhat confusing) value of -1. The exception to this rule is when the numerator is 0 as well, in which case the result will become 0. Modulo operations where the denominator is 0 will always result in a value of 0.

11.5 Boolean Operations

All of the usual logic and comparison operators have been implemented in Acus, which allow the programmer to build complex logical expressions. Some of these logical building blocks will be explained below.

11.5.1 Conversion to Bool

The simplest boolean conversion is one where an integer is transformed into either a 0 or 1, depending on its truth-value (any non-zero is considered *true*). To accomplish this, the value of the operand `x` is moved into a scratch cell, leaving `x` empty. We then branch off the value of `s`: if non-zero, `x` is set back to 1. If the conditional block is not entered, `x` remains 0.

```

1  # BOOL(x)
2  # Assumes s is an empty scratch cell
3
4  MOVE(x, s)
5  s [[-] x+ s]

```

11.5.2 Logical Not

The not-operator does the exact opposite of a boolean conversion. It transforms its operand into 1 when its value equals 0 and vice versa for any non-zero values. The algorithm is almost identical to that of `BOOL(x)`: instead of setting `x` to 0 and incrementing it in the loop-block, we set it to 1 and decrement it inside the loop.

```

1  # NOT(x)
2  # Assumes s is an empty scratch cell
3
4  MOVE(x, s)
5  x +
6  s [[-] x- s]

```

11.5.3 Logical And

The AND operator is implemented using a nested BF-loop block. The operands are used to decide if those blocks should be entered. The result is then set in the inner loop, which can only be reached if both operands evaluate to *true*.

```

1  # DESTR_AND(x, y)
2  # Assumes s is an empty scratch cell
3
4  MOVE(x, s)
5  s [[-]
6    y [[-] x+ y]
7    s
8  ]

```

Note that the algorithm above destroys the contents of `y` and replaces the original value of `x` by the result of the operation. A non-destructive version can be implemented simply by letting the destructive version operate on copies of the operands.

11.5.4 Logical Or

If we compare the logical AND algorithm to two switches wired in series, where current can only flow if both switches are turned on, the OR algorithm is its parallel counterpart. Each of the operands guards a loop and if either one of them is entered, the result is set to *true*.


```

1  # DESTR_OR(x, y)
2  # Assumes s is an empty scratch cell
3
4  MOVE(x, s)
5  s [[-]
6    x+
7    s
8  ]
9  y [[-]
10   x[-]+
11   y
12  ]

```

Again, this algorithm destroys the rhs of the expression but this is easily managed by operating on a copy if necessary. Note as well that in the second loop, x has to be reduced to 0 first in case the first branch was entered as well (a short-circuiting algorithm would only increase the complexity and runtime rather than cutting down on it).

11.5.5 Equality

In testing for equality, we rely on the fact that $x - y = 0$ when $x = y$ to implement a routine that calculates NOT($x - y$).

```

1  # EQ(x, y)
2
3  SUB_FROM(x, y)
4  NOT(x)

```

11.5.6 Less Than

This operator was already used before in the implementation of 16-bit addition, to determine the value of the carry-bit. It is implemented by reducing both of its operands simultaneously, until either one of them reaches zero. If the rhs is still non-zero when this happens, the lhs must have been the smaller of the two. This turns out to be a pretty tedious algorithm when expressed in BF, so instead it is presented in pseudocode below. Other comparison operators are implemented similarly.

```

1  # LT(x, y)
2
3  while x AND y:
4      --x
5      --y
6
7  x = NOT(y)

```

11.5.7 Signed Comparisons

Comparisons on signed integers cannot use the unsigned implementations discussed before. Instead, the exact form of the algorithm depends on the sign bits of both arguments. A runtime check has

to be done to determine which one of four branches must be taken. For example, for the less-than operator, if its operands are both signed:

1. If both are nonnegative, use the unsigned algorithm.
2. If the lhs is negative and the rhs is positive (or zero), return 1 immediately.
3. If the lhs is nonnegative and the rhs is negative, return 0 immediately.
4. If both are negative, use the unsigned greater-than algorithm on their absolute values.

This requires an algorithm to evaluate the sign of an integer and one to take its absolute value. To get the sign bit of an 8-bit value x (1 if negative, 0 otherwise), we simply compute the result of $x \geq 128$ (using the unsigned greater-equal operator). Values with bit patterns identical to unsigned values from 128 on are interpreted as negative numbers (i.e. have their highest bit set) in two's complement representation.

To calculate the absolute value of a signed integer, its sign bit is first calculated. If not set, the value can remain unmodified since it is already positive. If the sign bit *is* set, it can be negated simply by subtracting it from 0.

```

1  # ABS(x)
2  # Assumes s to be an empty scratch-cell
3
4  if SIGN_BIT(x):
5      # x = 0 - x
6      SUB_FROM(s, x)
7      MOVE(s, x)

```

11.6 Pointer Movement

BF has no random memory access instruction; the only way to reach a cell is to move the data-pointer (simply referred to as the *pointer* from now on) step by step, which requires the compiler to keep track of the pointer at all times. This poses a challenge for situations where the pointer has to move dynamically, i.e. to tape-locations not known at compile-time. This occurs for example when dereferencing pointers or when interacting with runtime indices into arrays.

11.6.1 Static Pointer Movement

In most scenarios, when the source and destination locations are known at compile time, pointer movement is straightforward: emit the required number of < or > commands. This is the common case for local slot operations within a frame, where the compiler can keep track of the current pointer position relative to the frame origin.

11.6.2 Dynamic Pointer Movement

The most interesting pointer-movement algorithms are the ones where the target is not known until runtime. Examples include array access by a runtime index and pointer dereferencing. These algorithms rely on metadata stored in macro-cells: markers, counters and payload fields are used so that the BF pointer can search through structured tape data and stop at the correct logical cell. We can distinguish between two different classes of dynamic pointer movement: *marker-based* and

offset-based moves. In marker-based moves, the pointer moves along the tape until it hits some pre-set marker. In offset-based moves, the offset is stored in the payload cells and dragged along with the pointer while being reduced; when the payload becomes zero, the destination has been reached.

Marker-Based

This method relies on a marker-field, the `FrameMarker` or `SeekMarker`, being set beforehand. We can then move the pointer into the corresponding field of the current cell and move in some direction until the marker is hit.

```

1  # SEEK_LEFT()
2  # Assumes s to be an empty scratch-field
3
4  COPY(SeekMarker, s)
5  NOT(s)
6  s [[-]
7      <<<<<<<<<      # skip to left-adjacent macro-cell
8      COPY(SeekMarker, s)
9      NOT(s)
10     s
11 ]
```

Because the pointer ends up at some unknown location, we commonly set another marker that can be used to bring the pointer back afterwards.

Offset-Based

The second class of dynamic pointer movement relies on a runtime value representing the offset being stored on the tape itself. For example, in the expression `x[i]`, `i` is a variable stored on the tape that contains the element index with respect to the start of the array `x`. This index is converted into a macro-cell offset based on the size of the element type. To reach a specific offset, the pointer is moved to the start of the array (known at compile-time) and the offset-value is copied or moved into its payload-fields (a single one for 8-bit values and both of them for 16-bit values). The pointer is then moved to the right while the payload is moved along and decremented. When the payload hits zero, the required offset has been reached.

```

1  # TO_OFFSET(b, i)
2  # b = base address
3  # i = macro-cell offset wrt base
4
5  COPY(i, b.payload)
6  b.payload [-
7      MOVE(b.payload, 9)      # move payload one macro-cell to the right
8      >>>>>>>>>      # follow with data-pointer
9  ]
10 # Pointer now points at the (empty) payload field of the target-element.
```

Note that, depending on the element-type of the array, the index does not always equal the actual offset in terms of the number of macro-cells. Some preprocessing needs to be done on arrays of type `T` when `sizeof(T)` is larger than 1, by multiplying the index by that size to get the actual offset.

11.6.3 Payload Transport

Dynamic writes are harder than dynamic reads because data has to be carried to a location that is not statically known. Acus solves this by moving values through designated payload fields. The payload travels along with the dynamic traversal and is deposited when the target macro-cell is found.

For marker-based movements, the payload can simply be moved along with the pointer because the payload fields are not used to perform the move itself. For offset-based movements, the payload fields already serve the purpose of containing the offset itself and cannot be used to transport the new value. Instead, the usual offset-based method described in Section 11.6.2 is used to find the element, where a marker is planted. Then, the marker-based method is used to move the payload to that position and overwrite its value field(s).

Chapter 12

Caching

Slot proxies (Section 9.3.2) allow expressions to remain abstract until their value is needed. However, repeatedly materializing the same proxy can duplicate work. For example, accessing the same array element several times in a block should not require materializing the element locally each time. Instead, multiple accesses can be cached and a writeback occurs only when necessary, for example at control-flow boundaries.

```
1  let x: array<u8, 5>
2  let i: u8
3  ...
4
5  let y: u8 = x[i]  # materializes x[i] in a local cache before assigning to y
6  x[i] = 42        # operates on the same cached value and marks it dirty
7  x[i] *= 2        # operates directly on the dirty entry
8  let y: u8 = f(x)  # before passing x to f(), x[i] is written
9                  # back to canonical storage
```

To implement this caching system, dependencies between proxies need to be carefully maintained. During compilation, a dependency graph is built and used to determine which cache entries have become dirty and which ones need to be flushed back into their actual storage location. This chapter will go over the different techniques and compromises that were made in their implementation.

12.1 Cache Graphs

A cache entry connects a logical proxy to a concrete slot that currently holds its materialized value. Entries can be marked dirty when modified or invalidated/removed when there is no guarantee that their values still represent the actual referred-to data.

Figure 12.1 shows a simplified representation of a cache graph, where each entry (node) contains information about the proxy that it represents, the physical location of the cached slot, and its parent and child nodes. A proxy **A** is considered a parent of proxy **B** if the canonical storage location of **B** is a subset of that of **A**. For example, given the expression `x[i][j]`, the expression `x[i]` is represented by a proxy that itself will be cached (assuming `i` is not known at compile time). Therefore, to materialize the element `x[i][j]`, we first need to materialize `x[i]`, and their cache entries are in a parent-child relation: `x[i]` is the parent of `x[i][j]` because the storage slot of `x[i]` encloses that of `x[i][j]`.

In the example of Figure 12.1, the 3×3 matrix x is indexed by the index variables i , j , k and l through the expressions at the top. The first line causes the left branch of the tree to be generated: in order to materialize $x[i][j]$, its parent $x[i]$ is materialized in a cache slot, which can be generated from the statically known starting address of its parent x (which does not have to be cached). In this example, $x[i]$ is materialized right after its parent x (as can be confirmed from the offsets and size data) but this does not necessarily have to be the case.

The second expression causes a second cache-branch to be generated because the proxies $x[i]$ and $x[k]$ have different proxy-identities (even though they could be referring to the same objects at runtime, when $i == k$).

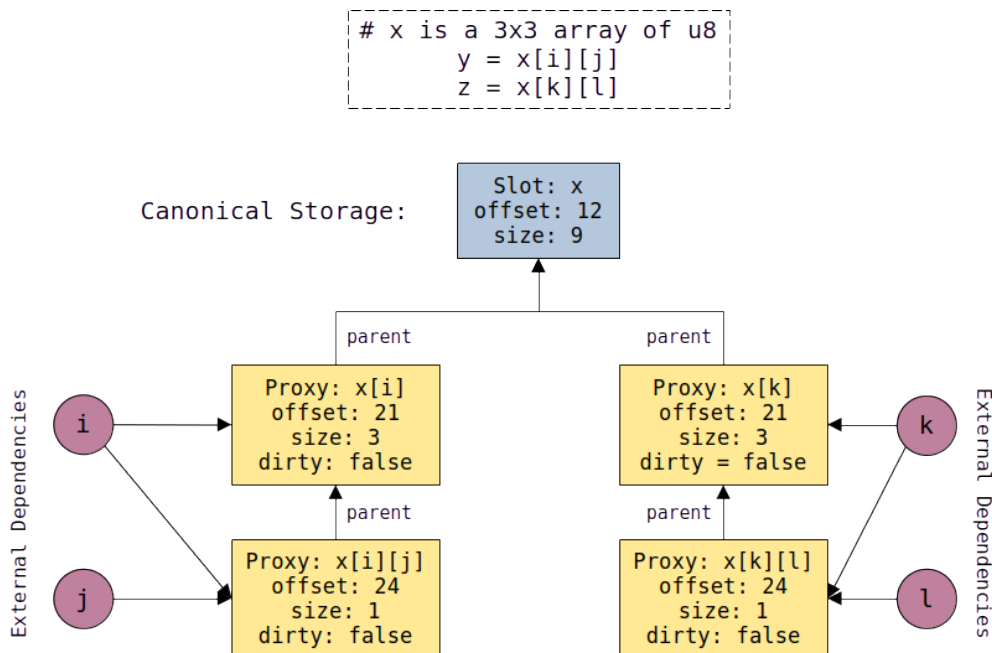


Figure 12.1: Resulting cache graph after materializing $x[i][j]$ and $x[k][l]$. The blue box represents the canonical slot containing x , whereas the yellow boxes represent cached subslots (at unknown offsets with respect to the canonical storage) that have been materialized in known locations.

12.2 Flushing and Invalidation

12.2.1 Dirty Entries

In the example of Figure 12.1, all entries are *clean* (i.e. not *dirty*), since none of the expressions involving them modified their contents. The expression $x[i][j] = 3$ would have the effect of writing the value 3 to the cache slot and marking its corresponding entry as dirty. Dirty entries must at some point be flushed into their parent to make sure that all changes applied to a cached object eventually propagate into canonical storage.

12.2.2 Flushing

Flushing an entry means writing its contents back into its parent (either another cached slot or canonical storage). An entry needs to be flushed in the following scenarios:

1. The entry is marked *dirty* and a parent somewhere up in the cache hierarchy is materialized. If the dirty entry is multiple parents removed, this causes all intermediate parents to be flushed as well. For example, if $x[i][j]$ is dirty and x is materialized, the dirty $x[i][j]$ is flushed into $x[i]$, which then becomes dirty and is subsequently flushed into x itself. The entire path from $x[i][j]$ to x is now clean.
2. Whenever a control-flow boundary is crossed. For example, at jumps or function calls, the entire cache is cleared by flushing all entries (starting at the deepest child nodes) back into their canonical storage. This makes sure that all flow paths see the correct data.
3. Whenever a pointer is dereferenced. Because pointers can refer to anything (of some type) at runtime, the entire cache is cleared to make sure that the pointee is up-to-date when fetched by the dereferencing algorithm.
4. Before cache invalidation. In scenarios where a cache entry is invalidated (removed from the tree), it is flushed back before doing so. Cache invalidation is discussed in more detail below.

At a return boundary, local cache entries can be discarded because their frame is about to disappear. Entries referring to storage that outlives the frame, such as globals and dereferenced pointers, must first be flushed. Return values are transferred separately by the return mechanism described earlier.

12.2.3 Invalidation

Entries can be invalidated when an external dependency or (indirect) parent is modified or freed by the memory management system. In the example of Figure 12.1, the entries $x[i]$ and $x[i][j]$ have been materialized in cache using the value stored in slot i at that time. When the value of i is modified, the proxy refers to a new object and the entry has become invalid. When this is about to happen, the invalidated proxy has to be flushed (if dirty) *before* modifying the dependency, after which it (and its descendants) can be removed from the cache tree.

12.2.4 Writes

Writing to a proxy will, as mentioned before, cause its cached entry to be marked *dirty*. When the object is overwritten independently of its original value, there is no need to materialize that value at all. For example, when compiling the expression $x[i][j] = 3$, the value stored initially in $x[i][j]$ is completely irrelevant. In such cases, a new entry for the proxy $x[i][j]$ is created and marked dirty immediately and no materialization algorithm has to be executed.

12.2.5 Example

The listing below shows a more complete version of code that leads to the cache hierarchy of Figure 12.1. At the lines marked (a) and (b), the graph from Figure 12.1 is generated. Then, at line (c), a new value is assigned to $x[i][j]$. This assignment expression will write the value 3 directly to the already cached slot for the proxy $x[i][j]$ and mark this entry as *dirty*. At the next expression (line (d)), a new proxy ($x[i][k]$) is materialized, which shares a parent ($x[i]$) with the now dirty entry $x[i][j]$. Therefore, before materializing $x[i][k]$, the dirty entry is flushed upwards into $x[i]$, after which its dirty state is cleared and $x[i][k]$ can be materialized. Finally, at line (e), the value of i is modified. All cache entries that depend on i (the left branch of the graph)

therefore need to be invalidated. In this example, these entries have just been flushed and are still clean, so there is no need to flush them prior to deletion.

```
1  using Vec3 = array<u8, 3>
2  using Mat  = array<Vec3, 3>
3  let x: Mat
4
5  # Indices (not known at compile-time)
6  let i: u8
7  let j: u8
8  let k: u8
9  let l: u8
10 ...
11
12 # Materialization and caching
13 let y: u8 = x[i][j]    # (a)
14 let z: u8 = x[k][l]    # (b)
15
16 # Flushing and invalidating
17 x[i][j] = 3            # (c)
18 let q: u8 = x[i][k]    # (d)
19 i = 0                  # (e)
20 ...
```


Chapter 13

Other Optimizations

The caching system is one of the optimizations used by Acus to try to somewhat limit the number of generated BF instructions, which tends to grow very fast for complex programs. Even though it was never a goal to produce highly optimized BF, several other optimizations have been implemented to fight the explosive nature of the generated output. Each of these techniques will be briefly outlined in this chapter.

13.1 Move Semantics

The algorithms for moving and copying BF cells are very similar: copying means we simultaneously move to the target cell and a scratch cell, only for the scratch cell to be emptied back into the source location. Obviously, we should move and avoid the full copy algorithm whenever possible. Therefore, if we can prove that the source is expendable, we can just move its contents out of it to a new location. Acus tries to find as many opportunities as possible to move instead of copy. For example, this is possible in the following situations:

1. Assignment from a temporary expression. e.g. $x = y + z$, where $y + z$ is a temporary which does not outlive the statement anyway.
2. Flushing cache entries that are about to be deleted. When a cache tree is flushed prior to it being deleted, we can move instead of copy towards the root.

13.2 Literal Evaluation

13.2.1 Constant Folding

While Acus does *not* implement advanced constant-expression evaluation (it does not keep track of known values of runtime variables), it *will* apply constant folding to literal expressions. For example, consider the expression $3 + 4$. In a naive implementation, the compiler might need to allocate 2 temporary slots for the literal expressions 3 and 4 and call a runtime algorithm on those slots to compute the value 7. Acus will simply collapse this to the literal expression 7 without generating any runtime code.

13.2.2 Specialized Constant Algorithms

All operators come in 2 flavors: one to apply its effect to two slots and one to apply its effect to a slot paired with a constant. For example, the expressions $x * y$ and $x * 8$ might be evaluated

differently. In the latter case, the constant can be evaluated to determine if the algorithm can be simplified or optimized. Multiplication by 0 will instantly result in the cell being zeroed out, whereas multiplication by 1 can return without any modifications at all. Similar optimizations have been implemented for other arithmetic and logical operators.

1. Addition and subtraction by 0 result in a no-op.
2. Multiplication by 0 results in immediate 0.
3. Multiplication by 1 results in a no-op.
4. Division by 0 rules are hard-coded (see 11.4.5).
5. Unsigned comparisons to 0 only need to check if the value is 0 (can't be less than 0 anyway).
6. Signed comparisons to 0 can use the sign-bit algorithms.

13.3 Primitive and BF Merging

13.3.1 Merging Primitives

Before generating actual BF characters, Acus will produce a series of primitives, each of which represents a relatively simple sequence of BF instructions on a slightly higher level (Table 13.1). Sequences of primitives can sometimes be merged into shorter sequences or even no-ops. For example, if the primitive `MovePointerRelative(3)` is immediately followed by the primitive `MovePointerRelative(6)`, they can be merged into a single `MovePointerRelative(9)`. This leads to a series of merge-rules, listed in Table 13.2. By applying these rules iteratively to the generated sequence of primitives, the program can be shrunk significantly in post-processing.

Primitive	Description
<code>LoopOpen</code>	Open a BF loop, compiles to simply <code>[</code> .
<code>LoopClose</code>	Close a BF loop, compiles to simply <code>]</code> .
<code>MovePointerRelative(x)</code>	Move the pointer by <code>x</code> cells (positive means to the right).
<code>ChangeValue(c)</code>	Change the current value by an amount <code>c</code> .
<code>ZeroCell</code>	Set the current value to 0 using <code>[-]</code> .
<code>ZeroCellPlus</code>	Set the current value to 0 using <code>[+]</code> .
<code>ConstructConstant(c)</code>	Construct the value <code>c</code> in the current cell.
<code>MoveData(x)</code>	Move the contents of the current cell to relative position <code>x</code> .
<code>CopyData(x)</code>	Copy the contents of the current cell to relative position <code>x</code> .
<code>Add(x)</code>	Add the value of the cell at relative position <code>x</code> to the value of the current cell.
<code>Subtract(x)</code>	Subtract the value of the cell at relative position <code>x</code> from the value of the current cell.
<code>Boolean</code>	Set the current cell to its boolean value.
<code>Not</code>	Set the current cell to its logical not value.
<code>Or(x)</code>	Set the current value to the logical OR of this value and the value inside the cell at relative offset <code>x</code> .
<code>And(x)</code>	Set the current value to the logical AND of this value and the value inside the cell at relative offset <code>x</code> .
<code>Xor(x)</code>	Set the current value to the logical XOR of this value and the value inside the cell at relative offset <code>x</code> .

Primitive	Description
Equal(x)	Set the current value to 1 if it is equal to the value in the cell at relative offset x, 0 otherwise.
Less(x)	Set the current value to 1 if it is less than the value in the cell at relative offset x, 0 otherwise.
LessOrEqual(x)	Set the current value to 1 if it is less than or equal to the value in the cell at relative offset x, 0 otherwise.
Greater(x)	Set the current value to 1 if it is greater than the value in the cell at relative offset x, 0 otherwise.
GreaterOrEqual(x)	Set the current value to 1 if it is greater than or equal to the value in the cell at relative offset x, 0 otherwise.
Cmp(c)	Set the current value to 1 if it is equal to the compile-time constant c, 0 otherwise.

Table 13.1: Acus primitives.

Primitive	Followed By	Merged Into
ZeroCell	ZeroCell	ZeroCell
ZeroCell	ZeroCellPlus	ZeroCell
ZeroCellPlus	ZeroCellPlus	ZeroCellPlus
ZeroCellPlus	ZeroCell	ZeroCellPlus
LoopClose	ZeroCell	LoopClose
LoopClose	ZeroCellPlus	LoopClose
MovePointerRelative(x)	MovePointerRelative(y)	MovePointerRelative(x + y)
ConstructConstant(c)	ChangeValue(d)	ConstructConstant(c + d)
ConstructConstant(c)	ConstructConstant(d)	ConstructConstant(d)
ConstructConstant(c)	ZeroCell	ZeroCell
ConstructConstant(c)	ZeroCellPlus	ZeroCellPlus
ConstructConstant(c)	Boolean	ConstructConstant(c != 0)
ConstructConstant(c)	Not	ConstructConstant(c == 0)
ConstructConstant(c)	Cmp(d)	ConstructConstant(c == d)
ChangeValue(c)	ChangeValue(d)	ChangeValue(c + d)
ChangeValue(c)	ZeroCell	ZeroCell
ChangeValue(c)	ZeroCellPlus	ZeroCellPlus
ChangeValue(c)	ConstructConstant(d)	ConstructConstant(d)
MoveData(x)	ZeroCell	MoveData(x)
MoveData(x)	ZeroCellPlus	MoveData(x)
CopyData(x)	ZeroCell	MoveData(x)
CopyData(x)	ZeroCellPlus	MoveData(x)
Cmp(c)	ZeroCell	ZeroCell
Cmp(c)	ZeroCellPlus	ZeroCellPlus
Cmp(c)	Boolean	Cmp(c)
Boolean	Boolean	Boolean
Boolean	Not	Not
Not	Boolean	Not
Not	Not	Boolean
Or(x)	Boolean	Or(x)

Primitive	Followed By	Merged Into
And(x)	Boolean	And(x)
Xor(x)	Boolean	Xor(x)
Add(x)	ZeroCell	ZeroCell
Add(x)	ZeroCellPlus	ZeroCellPlus
Add(x)	ConstructConstant(c)	ConstructConstant(c)
Subtract(x)	ZeroCell	ZeroCell
Subtract(x)	ZeroCellPlus	ZeroCellPlus
Subtract(x)	ConstructConstant(c)	ConstructConstant(c)
Equal(x)	ZeroCell	ZeroCell
Equal(x)	ZeroCellPlus	ZeroCellPlus
Equal(x)	ConstructConstant(c)	ConstructConstant(c)
Equal(x)	Boolean	Equal(x)
Less(x)	ZeroCell	ZeroCell
Less(x)	ZeroCellPlus	ZeroCellPlus
Less(x)	ConstructConstant(c)	ConstructConstant(c)
Less(x)	Boolean	Less(x)
LessOrEqual(x)	ZeroCell	ZeroCell
LessOrEqual(x)	ZeroCellPlus	ZeroCellPlus
LessOrEqual(x)	ConstructConstant(c)	ConstructConstant(c)
LessOrEqual(x)	Boolean	LessOrEqual(x)
Greater(x)	ZeroCell	ZeroCell
Greater(x)	ZeroCellPlus	ZeroCellPlus
Greater(x)	ConstructConstant(c)	ConstructConstant(c)
Greater(x)	Boolean	Greater(x)
GreaterOrEqual(x)	ZeroCell	ZeroCell
GreaterOrEqual(x)	ZeroCellPlus	ZeroCellPlus
GreaterOrEqual(x)	ConstructConstant(c)	ConstructConstant(c)
GreaterOrEqual(x)	Boolean	GreaterOrEqual(x)

Table 13.2: Primitive merge rules.

13.3.2 Merging BF

As a final pass, after all primitives have been lowered into actual BF characters, Acus will scan the resulting source code for cancellation opportunities: sequences of consecutive + and - or < and > commands that can be collapsed. The primitive-merging rules already do this to some extent by merging `ChangeValue` and `MovePointerRelative` primitives, but that does not capture all opportunities. For example, when a `MovePointerRelative` primitive is followed by another that, when lowered to BF, starts with a sequence of > commands, these can be merged or cancelled in the final result.

Part III

Reflection

Chapter 14

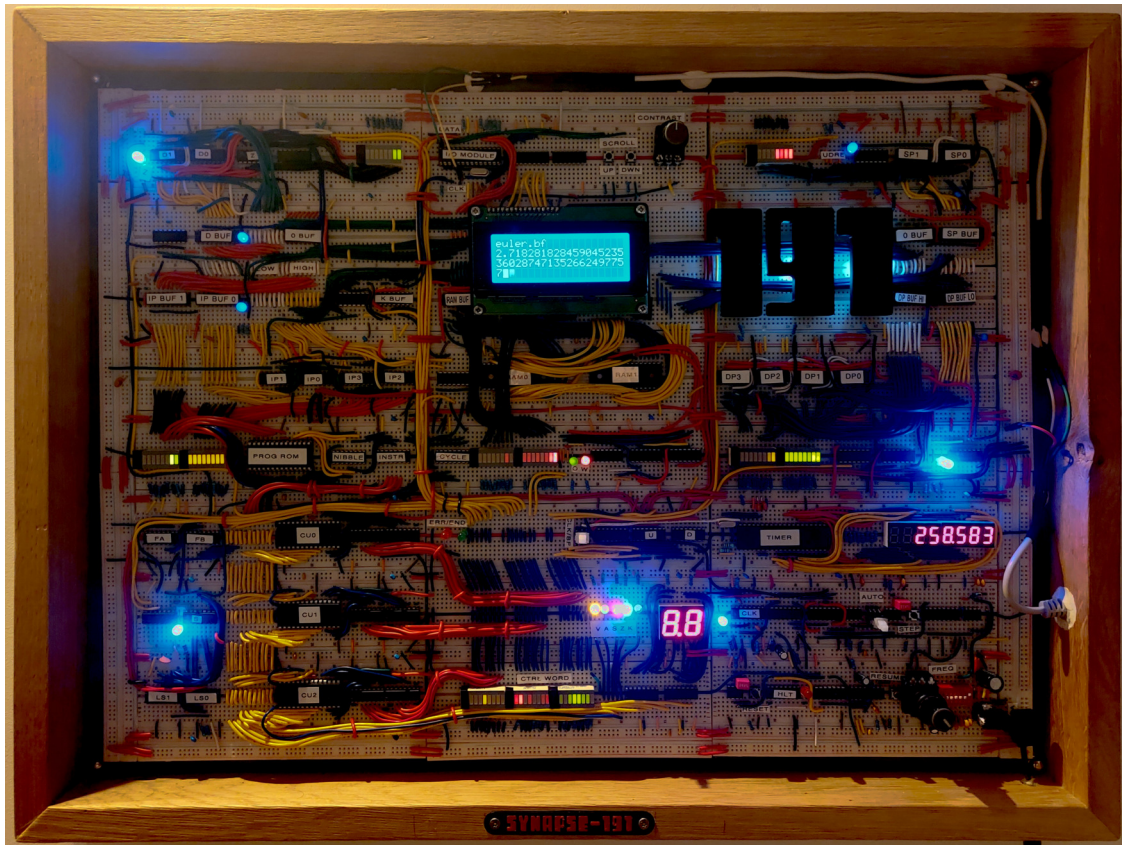
Reflection

14.1 Conclusion

This project set out as a “What if?” for which expectations were deliberately kept low. A functional system was the goal; not a fast one. We had no idea what kinds of speeds we could expect, how stable the system would be and frankly whether we would even end up with anything functional. The only goal was simply to get something going. Given those goals, this project has certainly delivered to an unexpected extent.

Building Synapse-191 has been a tremendously powerful learning experience. Above all, the joy of seeing months of tedious work, planning and debugging culminate into a working system is unparalleled. The system’s specifications themselves never prevented any of the tested BF programs from running; it has never run low on memory, has a large enough program-ROM to store BF code and has plenty of stack-space to run even computationally demanding programs. Features like the IO system, being able to store multiple programs to select from, single-stepping and easy control of the clock make it an enjoyable process to operate the computer. Moreover, with the toolchain developed to a mature state, adding features through modification of the firmware and microcode was relatively straightforward and thoroughly rewarding to do.

The system has at the time of writing been framed and hung from a wall. External ports have been added to the frame such that even when hanging, it can still easily be plugged in and controlled using the keyboard. Some 3D-printed parts have been added for visual effects, but no further extensions will be added to the system as-is. There are currently no plans for future development of a next generation (e.g. on a PCB), because *what’s the point?* Then again, what *was* the point, ever?



14.2 Final Thoughts

14.2.1 Virtual Registers

A big limitation of using the register driver as a centralized system to increment or decrement each register is the fact that only one register can be addressed per clock cycle. Having two addresses on the register driver still unconnected (addresses 110 and 111), their corresponding outputs could be connected to multiple registers that commonly need to increment together. For example, address 110 (bits RS2 and RS1) would then correspond to the virtual register D/IP. The corresponding U and D output signals would then be connected (through OR gates) to the counting inputs of both the D and IP register. Addressing this virtual register would then allow for incrementing or decrementing these registers simultaneously, which saves a clock cycle in the PLUS instruction. The MINUS instruction does not benefit from this trick, since we can't both decrement D and increment IP at the same time. The same could be done for DP and IP, saving a clock cycle in the RIGHT instruction.

Some preliminary tests on the programs from the test suite (using again the modified interpreter to acquire statistics) show that 30 to 40% of all instructions executed at runtime are either + or >. Based on this data, we can set some boundaries on the estimated increase in performance when the aforementioned virtual registers (D/IP and DP/IP) would be implemented. Assuming 3.5 cycles per instruction under current circumstances (see Chapter 6), this number would be reduced by 1 cycle in 40% of all instructions (in the optimal case), leading to a decrease of $3.5 - 0.4 \times 1 = 3.1$ cycles per instruction on average, or about a 10% performance increase.

14.2.2 No Register Driver

Taking the considerations above even further, one could argue that the system would do even better with no register driver at all. Granted, this would have required four additional control lines (5 INC + 4 DEC for a total of 9 versus 1 INC + 1 DEC + 3 RS-bits) and would have required each register to implement logic that properly drives the U and D inputs of the '193, but it is still worth considering what kind of performance improvement could be expected. As before, the gathered statistics allow us to estimate the fraction of instructions that would benefit from driving the registers in parallel, allowing for multiple INC and DEC instructions to execute simultaneously. Given the fact that we could even issue an INC and another DEC together (which wouldn't have been possible with the virtual register), the +, -, < and > commands would all be improved by one cycle. These commands make up 70 to 80% of all instructions executed at runtime. A similar calculation then shows that the average number of cycles per instruction could drop by at least 0.7, all the way down to 2.8 cycles per instruction for a performance increase of 20%. However, since there currently are no spare control lines available, this would have required adding a fourth EEPROM to extend the control word beyond 24 bits. The system's layout was frozen long before this consideration even came to mind, rendering this idea next to impossible to implement on the current version of the system without requiring a complete teardown.

14.2.3 Designing a PCB

Even though the system appears to run very stably at the time of writing, it is still very vulnerable given that it relies on springy connections that could oxidize or lose their tension over time. The durable approach would be to design a PCB with proper voltage and ground planes, stable traces between ICs and durable soldered connections. If this direction is ever going to be pursued, most LEDs would probably be removed to minimize the PCB's footprint. We would also have to consider how the program ROM can be inserted and removed easily; maybe even using ROM on an expansion card like the NES of the old days. For maximal modularity, the IO module could be implemented as an expansion card; different implementations (perhaps even using graphics or sound instead of ASCII characters as their output mode) could easily be swapped in and out.

14.2.4 Going Full 16-bit

In Section 4.10 it was mentioned that the vast majority of memory cells in the second SRAM chip are left unused. Only the high byte of the IP values is stored in this chip when the IP is pushed to the stack on entering a loop, for a maximum of 256 bytes stored on a 512 KiB memory chip (of which 64 KiB is addressable through the 16-bit address bus); none of the other accessible cells are ever used. Since the data bus is already 16 bits wide, it would have been fairly easy to make the D register a 16-bit value as well. This would allow for BF programs that assume 16-bit data cells. Even though this was briefly considered in the early stages of the project, we chose not to go through with it because a) it is not canonical BF and b) we did not want to make the system more complex than necessary at the time (we would effectively need a 16-input NOR gate, requiring two more ICs on top of the two additional register ICs). In hindsight, this would have been a nice (and pretty achievable) feature to have.

14.2.5 Using the '191

The 74LS191 seems like the perfect candidate to implement the counting registers. It simply has a U/D input to select the counting direction, a count-enable input (CTEN) and is connected directly

to the clock. This is exactly how the register modules behaved in early abstract designs of the system. The '191 was never used for the simple reason that we never knew about its existence and ordered the '193 instead; that was the first chip we encountered that could do the job. In hindsight—and purely by luck—the '193 was indeed the better choice because of its ability to (asynchronously) clear its values; the '191 has no clear input. This would have made it more difficult to implement clean reset functionality. Also, the number of control lines to the modules would not have been different ($U/D + CTEN$ instead of $U + D$), so the need for a register driver would still be there.

14.2.6 Decoupling Capacitors

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

14.2.7 Using AI

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

14.2.8 IO Module: Cheating?

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

14.2.9 Higher Clock Speeds

Once all the instabilities faded even at higher clock speeds (after a proper implementation of the K-flag to act as a synchronization barrier between the CPU and the IO module), it should in principle be possible to push the clock far beyond its current limits. These limits have now been imposed by

the choice of RC combinations on the 555 timer and can be pushed even higher. Other breadboard implementations of homebrew CPUs have reported stable clock speeds in the MHz range. The reason for not doing this is purely pragmatic. Enough time has been sunk into this project as it is and it needs to end somewhere, exposing a human limit rather than a technical one.

Appendix A

Microcode Table

Instr	K	V	A	S	Z	Cycle	Control Signals				
Any except “.”, “,” and “?”						0	LD(FBI)				
+			0	0		1	INC(D)	SETV(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	LD(D)	OE(RAM)			
			1	0		2	INC(D)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
			1			1	INC(IP)	CLR(CC)			
-			0	0		1	DEC(D)	SETV(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	LD(D)	OE(RAM)			
			1	0		2	DEC(D)	SETV(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
			1			1	INC(IP)	CLR(CC)			
<			0	0		1	DEC(DP)	SETA(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	EN(D)	WE(RAM)			
			1	0		2	DEC(DP)	SETA(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
			1			1	INC(IP)	CLR(CC)			
>			0	0		1	INC(DP)	SETA(CU)	LD(FA)		
			0	0		2	INC(IP)	CLR(CC)			
			1	0		1	EN(D)	WE(RAM)			
			1	0		2	INC(DP)	SETA(CU)	LD(FA)		
			1	0		3	INC(IP)	CLR(CC)			
			1			1	INC(IP)	CLR(CC)			
[			0	0	1	1	INC(LS)				
			0	0	1	2	INC(IP)	CLR(CC)			
			0	0	0	1	INC(SP)				
			0	0	0	2	WE(RAM)	EN(SP)	EN(IP)		
			0	0	0	3	INC(IP)	CLR(CC)			
			1	0		1	EN(DP)	LD(D)	OE(RAM)	LD(FA)	CLR(CC)
			1			1	INC(LS)				
			1			2	INC(IP)	CLR(CC)			
]			0	0	1	1	DEC(SP)				
			0	0	1	2	INC(IP)	CLR(CC)			
			0	0	0	1	EN(SP)	OE(RAM)	LD(IP)		
			0	0	0	2	INC(IP)	CLR(CC)			
			1	0		1	EN(DP)	OE(RAM)	LD(D)	LD(FA)	CLR(CC)
			1			1	DEC(LS)				

		1	2	INC(IP)	CLR(CC)			
.	0	0	0	0	LD(FBI)			
	0	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	
	0	0	0	2	LD(FBI)	CLR(CC)		
	0	1	0	0	LD(FBI)			
	0	1	0	1	SETV(CU)	SETA(CU)	LD(FA)	LD(D) OE(RAM)
	0	1	0	2	LD(FBI)	CLR(CC)		
	1	0	0	0	LD(FBI)			
	1	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	EN(D) WE(RAM)
	1	0	0	2	LD(FBI)	CLR(CC)		
	0	1	1	0	EN(D)	EN(SCR)	CLR(CC)	
	1	1	1	0	LD(FA)	CLR(K)	INC(IP)	
	1	1	1	0	LD(FBI)	CLR(CC)		
		1	0	LD(FBI)				
		1	1	INC(IP)	CLR(CC)			
,	0	0	0	0	LD(FBI)			
	0	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	
	0	0	0	2	LD(FBI)	CLR(CC)		
	0	1	0	0	LD(FBI)			
	0	1	0	1	SETV(CU)	SETA(CU)	LD(FA)	LD(D) OE(RAM)
	0	1	0	2	LD(FBI)	CLR(CC)		
	1	0	0	0	LD(FBI)			
	1	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	EN(D) WE(RAM)
	1	0	0	2	LD(FBI)	CLR(CC)		
	0	1	1	0	EN(KB)	CLR(CC)		
	1	1	1	0	LD(D)			
	1	1	1	0	SETV(CU)	LD(FA)	CLR(K)	INC(IP)
	1	1	1	0	LD(FBI)	CLR(CC)		
		1	0	LD(FBI)				
		1	1	INC(IP)	CLR(CC)			
?	0	0	0	0	LD(FBI)			
	0	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	
	0	0	0	2	LD(FBI)	CLR(CC)		
	0	1	0	0	LD(FBI)			
	0	1	0	1	SETV(CU)	SETA(CU)	LD(FA)	LD(D) OE(RAM)
	0	1	0	2	LD(FBI)	CLR(CC)		
	1	0	0	0	LD(FBI)			
	1	0	0	1	SETV(CU)	SETA(CU)	LD(FA)	EN(D) WE(RAM)
	1	0	0	2	LD(FBI)	CLR(CC)		
	0	1	1	0	EN(KB)	EN(OUT)	CLR(CC)	
	1	1	1	0	LD(D)			
	1	1	1	0	SETV(CU)	LD(FA)	CLR(K)	INC(IP)
	1	1	1	0	LD(FBI)	CLR(CC)		
		1	0	LD(FBI)				
		1	1	INC(IP)	CLR(CC)			
NOP				1	INC(IP)	CLR(CC)		
HLT		0		1	HLT(CLK)			
		0		2	INC(IP)	CLR(CC)		
		1		1	INC(IP)	CLR(CC)		
INIT	0			1	CLR(CC)			
	1		1	1	EN(D)	WE(RAM)	INC(LS)	
	1		1	2	LD(FBI)	INC(DP)		
	1	0	1	3	INC(IP)	CLR(CC)		
	1	1	1	3	CLR(CC)			
INIT_FINISH	1			1	CLR(DP)	CLR(K)	INC(IP)	CLR(CC)
LOAD_SLOT	0			1	CLR(CC)			
	1			1	LD(D)	INC(LS)		
	1		1	2	CLR(K)	INC(IP)	CLR(CC)	

PROG_START	1 0	1	DEC(D)	
	1 1	1	DEC(LS)	
	1	2	INC(IP)	CLR(CC)
PROG_END	0	1	END(CU)	HLT(CLK)
	1	1	INC(IP)	CLR(CC)

Table A.1: Control signals for each of the BF instructions.

Appendix B

Mugen Specification

```
1  # This file can be compiled with Mugen, see https://github.com/jorenheit/mugen.
2
3  [rom] { 8192 x 8 x 3 }
4
5  [address] {
6    cycle: 3
7    opcode: 4
8    flags: K, V, A, S, Z
9  }
10
11 [signals] {
12   HLT
13   RS0
14   RS1
15   RS2
16   INC
17   DEC
18   CLR_DP
19   EN_SP
20
21   OE_RAM
22   WE_RAM
23   EN_IN
24   EN_OUT
25   EN_V
26   EN_A
27   LD_FBI
28   LD_FA
29
30   EN_IP
31   LD_IP
32   EN_D
33   LD_D
34   CR
35   CLR_K
36   END
37   ERR
38 }
39
40 [opcodes] {
41   NOP          = 0x00
42   PLUS         = 0x01
```

```

43 MINUS          = 0x02
44 LEFT           = 0x03
45 RIGHT          = 0x04
46 IN             = 0x05
47 OUT            = 0x06
48 LOOP_START     = 0x07
49 LOOP_END       = 0x08
50 RAND           = 0x09
51 PROG_START     = 0x0a
52 PROG_END       = 0x0b
53 LOAD_SLOT      = 0x0c
54 INIT           = 0x0d
55 INIT_FINISH    = 0x0e
56 HALT           = 0x0f
57 }
58
59
60 [macros] {
61     # Register Select
62     R_D  = RS0
63     R_DP = RS1
64     R_SP = RS0, RS1
65     R_IP = RS2
66     R_LS = RS0, RS2
67
68     # Load/store to RAM
69     LOAD_D  = LD_D, OE_RAM
70     STORE_D = EN_D, WE_RAM
71     LOAD_IP = LD_IP, EN_SP, OE_RAM
72     STORE_IP = EN_IP, EN_SP, WE_RAM
73
74     # Set A and V flags
75     SET_V  = EN_V, LD_FA
76     SET_A  = EN_A, LD_FA
77     SET_VA = EN_V, EN_A, LD_FA
78     CLR_VA = LD_FA
79
80     # INC/DEC Registers
81     INC_D = INC, R_D
82     DEC_D = DEC, R_D
83     INC_DP = INC, R_DP
84     DEC_DP = DEC, R_DP
85     INC_SP = INC, R_SP
86     DEC_SP = DEC, R_SP
87     INC_LS = INC, R_LS
88     DEC_LS = DEC, R_LS
89     INC_IP = INC, R_IP
90
91     # Move to next instruction
92     NEXT = INC_IP, CR
93
94     # RNG
95     EN_RNG = EN_IN, EN_OUT
96 }
97
98 [microcode] {
99
100     # Most opcodes do the same thing in cycle 0:
101     # load the new instruction into I and latch flags into FB

```

```

102  NOP:0:()                -> LD_FBI
103  PLUS:0:()              -> LD_FBI
104  MINUS:0:()             -> LD_FBI
105  LEFT:0:()              -> LD_FBI
106  RIGHT:0:()             -> LD_FBI
107  LOOP_START:0:()        -> LD_FBI
108  LOOP_END:0:()          -> LD_FBI
109  INIT:0:()              -> LD_FBI
110  INIT_FINISH:0:()       -> LD_FBI
111  LOAD_SLOT:0:()         -> LD_FBI
112  PROG_START:0:()        -> LD_FBI
113  PROG_END:0:()         -> LD_FBI
114  HALT:0:()              -> LD_FBI
115
116
117  # BF opcodes
118  PLUS:1:(A=0,S=0)        -> INC_D, SET_V
119  PLUS:2:(A=0,S=0)        -> NEXT
120  PLUS:1:(A=1,S=0)        -> LOAD_D
121  PLUS:2:(A=1,S=0)        -> INC_D, SET_V
122  PLUS:3:(A=1,S=0)        -> NEXT
123  PLUS:1:(S=1)           -> NEXT
124
125  MINUS:1:(A=0,S=0)       -> DEC_D, SET_V
126  MINUS:2:(A=0,S=0)      -> NEXT
127  MINUS:1:(A=1,S=0)      -> LOAD_D
128  MINUS:2:(A=1,S=0)      -> DEC_D, SET_V
129  MINUS:3:(A=1,S=0)      -> NEXT
130  MINUS:1:(S=1)          -> NEXT
131
132  LEFT:1:(V=0,S=0)        -> DEC_DP, SET_A
133  LEFT:2:(V=0,S=0)        -> NEXT
134  LEFT:1:(V=1,S=0)        -> STORE_D
135  LEFT:2:(V=1,S=0)        -> DEC_DP, SET_A
136  LEFT:3:(V=1,S=0)        -> NEXT
137  LEFT:1:(S=1)           -> NEXT
138
139  RIGHT:1:(V=0,S=0)       -> INC_DP, SET_A
140  RIGHT:2:(V=0,S=0)       -> NEXT
141  RIGHT:1:(V=1,S=0)       -> STORE_D
142  RIGHT:2:(V=1,S=0)       -> INC_DP, SET_A
143  RIGHT:3:(V=1,S=0)       -> NEXT
144  RIGHT:1:(S=1)          -> NEXT
145
146  LOOP_START:1:(A=0,Z=1,S=0) -> INC_LS
147  LOOP_START:2:(A=0,Z=1,S=0) -> NEXT
148  LOOP_START:1:(A=0,Z=0,S=0) -> INC_SP
149  LOOP_START:2:(A=0,Z=0,S=0) -> STORE_IP
150  LOOP_START:3:(A=0,Z=0,S=0) -> NEXT
151  LOOP_START:1:(A=1,S=0)    -> LOAD_D, CLR_VA, CR
152  LOOP_START:1:(S=1)       -> INC_LS
153  LOOP_START:2:(S=1)       -> NEXT
154
155  LOOP_END:1:(A=0,Z=1,S=0)  -> DEC_SP
156  LOOP_END:2:(A=0,Z=1,S=0) -> NEXT
157  LOOP_END:1:(A=0,Z=0,S=0) -> LOAD_IP
158  LOOP_END:2:(A=0,Z=0,S=0) -> NEXT
159  LOOP_END:1:(A=1,S=0)     -> LOAD_D, CLR_VA, CR
160  LOOP_END:1:(S=1)        -> DEC_LS

```



```

161 LOOP_END:2: (S=1)          -> NEXT
162
163 OUT:1: (A=0,V=0,S=0 )      -> SET_VA
164 OUT:1: (A=0,V=1,S=0 )      -> STORE_D, SET_VA
165 OUT:1: (A=1,V=0,S=0 )      -> LOAD_D, SET_VA
166 OUT:2: (A=0,V=0,S=0 )      -> LD_FBI, CR
167 OUT:2: (A=0,V=1,S=0 )      -> LD_FBI, CR
168 OUT:2: (A=1,V=0,S=0 )      -> LD_FBI, CR
169 OUT:0: (A=1,V=1,S=0,K=0)    -> EN_OUT, EN_D, CR
170 OUT:0: (A=1,V=1,S=0,K=1)    -> CLR_VA, CLR_K, INC_IP
171 OUT:1: (A=1,V=1,S=0 )      -> LD_FBI, CR
172 OUT:0: (A=0,V=0,S=0 )      -> LD_FBI
173 OUT:0: (A=0,V=1,S=0 )      -> LD_FBI
174 OUT:0: (A=1,V=0,S=0 )      -> LD_FBI
175 OUT:1: (          S=1 )      -> NEXT
176 OUT:0: (          S=1 )      -> LD_FBI
177
178 IN:1: (A=0,V=0,S=0 )        -> SET_VA
179 IN:1: (A=0,V=1,S=0 )        -> SET_VA
180 IN:1: (A=1,V=0,S=0 )        -> SET_VA
181 IN:2: (A=0,V=0,S=0 )        -> LD_FBI, CR
182 IN:2: (A=0,V=1,S=0 )        -> LD_FBI, CR
183 IN:2: (A=1,V=0,S=0 )        -> LD_FBI, CR
184 IN:0: (A=1,V=1,S=0,K=0)     -> EN_IN, CR
185 IN:0: (A=1,V=1,S=0,K=1)     -> LD_D
186 IN:1: (A=1,V=1,S=0,K=1)     -> SET_V, CLR_K, INC_IP
187 IN:2: (A=1,V=1,S=0 )        -> LD_FBI, CR
188 IN:0: (A=0,V=0,S=0 )        -> LD_FBI
189 IN:0: (A=0,V=1,S=0 )        -> LD_FBI
190 IN:0: (A=1,V=0,S=0 )        -> LD_FBI
191 IN:1: (          S=1 )      -> NEXT
192 IN:0: (          S=1 )      -> LD_FBI
193
194 # Random BF extension (almost same as IN)
195 RAND:1: (A=0,V=0,S=0 )      -> SET_VA
196 RAND:1: (A=0,V=1,S=0 )      -> SET_VA
197 RAND:1: (A=1,V=0,S=0 )      -> SET_VA
198 RAND:2: (A=0,V=0,S=0 )      -> LD_FBI, CR
199 RAND:2: (A=0,V=1,S=0 )      -> LD_FBI, CR
200 RAND:2: (A=1,V=0,S=0 )      -> LD_FBI, CR
201 RAND:0: (A=1,V=1,S=0,K=0)    -> EN_RNG, CR
202 RAND:0: (A=1,V=1,S=0,K=1)    -> LD_D
203 RAND:1: (A=1,V=1,S=0,K=1)    -> SET_V, CLR_K, INC_IP
204 RAND:2: (A=1,V=1,S=0 )      -> LD_FBI, CR
205 RAND:0: (A=0,V=0,S=0 )      -> LD_FBI
206 RAND:0: (A=0,V=1,S=0 )      -> LD_FBI
207 RAND:0: (A=1,V=0,S=0 )      -> LD_FBI
208 RAND:1: (          S=1 )      -> NEXT
209 RAND:0: (          S=1 )      -> LD_FBI
210
211 # Housekeeping opcodes
212 NOP:1: ()                   -> NEXT
213 HALT:1: (S=0)                -> HLT
214 HALT:2: (S=0)                -> NEXT
215 HALT:1: (S=1)                -> NEXT
216
217 INIT:1: (K=0)                -> CR
218 INIT:1: (K=1, Z=1)           -> STORE_D, INC, R_LS
219 INIT:2: (K=1, Z=1)           -> LD_FBI, INC, R_DP

```

```

220 INIT:3:(K=1, Z=1, S=1)      -> CR
221 INIT:3:(K=1, Z=1, S=0)      -> NEXT
222
223 INIT_FINISH:1:(K=1)         -> CLR_DP, CLR_K, NEXT
224
225 LOAD_SLOT:1:(K=0)           -> CR
226 LOAD_SLOT:1:(K=1)           -> LD_D, INC, R_LS
227 LOAD_SLOT:2:(K=1)           -> CLR_K, NEXT
228
229 PROG_START:1:(Z=0, S=1)      -> DEC, R_D
230 PROG_START:1:(Z=1, S=1)      -> DEC, R_LS
231 PROG_START:2:(S=1)          -> NEXT
232
233 PROG_END:1:(S=0)             -> END, HLT
234 PROG_END:1:(S=1)            -> NEXT
235
236 catch                        -> ERR, HLT
237 }

```

Appendix C

Bill of Materials

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

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

C.1 Integrated Circuits

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

Table C.1: ICs used in the project.

C.2 Passive Components

Value	Quantity	Value	Quantity	Value	Quantity
220 Ω	1	470 μF	1	10 MHz crystal	1
330 Ω	8	10 μF	1	16 MHz crystal	1
470 Ω	80	4.7 μF	1	10 k Ω potentiometer	2
1 k Ω	5	1 μF	3	Toggle switch	2
2 k Ω	1	100 nF (104)	50	DIP switch (x8)	1
4.7 k Ω	10	10 nF (103)	5	Tactile button	5
5 k Ω	3	2.2 nF (222)	1	20-22AWG wire	?
10 k Ω	5	1 nF (102)	4	(c) Other passive components	
100 k Ω	2	470 pF (471)	1		
220 k Ω	1	220 pF (221)	1		
470 k Ω	1	100 pF (101)	1		
10 M Ω	2	47 pF	2		
(a) Resistors		22 pF	2		
		(b) Capacitors			

Table C.2: Resistors, capacitors and other passive components used in the project.

C.3 Other Parts

Part	Quantity	Description
Breadboard	~25	830-hole
Single 7-Segment Display	2	
Quad 7-Segment Display	2	
Single LED	15	Various colors
Bar LED strip	11	10 LEDs per strip
HD44780-Compatible LCD	1	20 $\times$ 4 characters

Table C.3: Other parts used in the project.

Listing 12: phi.bf, [12]

```

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

```

Listing 13: primes.bf, [14]

[illegible]

Listing 14: primes2.bf, [15]

D.7 Tic Tac Toe

[illegible]

Appendix E

IO Module ISR

```

163 //-----isr_begin-----
164 namespace ISR {
165     void panic(Error::ErrorType err) {
166         Timer1::stop();
167         detachInterrupt(digitalPinToInterrupt(K_IN_PIN));
168         Error::errno = err;
169     }
170
171     static volatile enum : uint8_t {
172         IDLE,
173         WAIT_FOR_KEYBOARD,
174         WAIT_AFTER_WRITE,
175         WAIT_AFTER_READ
176     } state = IDLE;
177
178     void onKCleared() {
179         switch (state) {
180             case IDLE: return;
181             case WAIT_AFTER_WRITE: setIOPinsToInput(); break;
182             case WAIT_AFTER_READ: break;
183             default: return panic(Error::REACHED_DEFAULT_1);
184         }
185         state = IDLE;
186         Timer1::restart();
187     }
188
189     void writeByteToBusAndWait(uint8_t const value) {
190         Timer1::stop();
191         setIOPinsToOutput();
192         writeByteToBus(value);
193         setK();
194         state = WAIT_AFTER_WRITE;
195     };
196
197     void enqueueByteFromBusAndWait() {
198         Timer1::stop();
199         lcdBuffer.enqueue(readByteFromBus());
200         setK();
201         state = WAIT_AFTER_READ;
202     };
203
204     void onTimer() {
205         switch (state) {
206             case IDLE: break;
207             case WAIT_FOR_KEYBOARD: {
208                 if (kbBuffer.available()) {
209                     writeByteToBusAndWait(kbBuffer.get());
210                 }
211                 return;

```

Appendix F

Constant Factory Lookup Table

Table F.1 lists the factor pair selected for each constant (c) from 0 through 128. The generated code first constructs the product (ab), then corrects for any remaining difference ($c - ab$) using increment or decrement operations. The entry for ($c = 0$) is unused, since zero is produced simply by clearing the target cell.

Table F.1: Factor pairs used by the constant factory.

c	$a \times b$	c	$a \times b$	c	$a \times b$	c	$a \times b$	c	$a \times b$
0	-	26	5×5	52	4×13	78	6×13	104	8×13
1	1×1	27	3×9	53	6×9	79	8×10	105	7×15
2	1×2	28	4×7	54	6×9	80	8×10	106	7×15
3	1×3	29	4×7	55	5×11	81	9×9	107	9×12
4	2×2	30	5×6	56	7×8	82	9×9	108	9×12
5	2×2	31	5×6	57	7×8	83	7×12	109	9×12
6	2×3	32	4×8	58	7×8	84	7×12	110	10×11
7	2×3	33	4×8	59	6×10	85	7×12	111	10×11
8	2×4	34	5×7	60	6×10	86	7×12	112	8×14
9	3×3	35	5×7	61	6×10	87	8×11	113	8×14
10	2×5	36	6×6	62	7×9	88	8×11	114	8×14
11	2×5	37	6×6	63	7×9	89	8×11	115	9×13
12	3×4	38	6×6	64	8×8	90	9×10	116	9×13
13	3×4	39	5×8	65	8×8	91	7×13	117	9×13
14	2×7	40	5×8	66	6×11	92	7×13	118	9×13
15	3×5	41	5×8	67	6×11	93	7×13	119	10×12
16	4×4	42	6×7	68	6×11	94	8×12	120	10×12
17	4×4	43	6×7	69	7×10	95	8×12	121	11×11
18	3×6	44	4×11	70	7×10	96	8×12	122	11×11
19	3×6	45	5×9	71	7×10	97	8×12	123	11×11
20	4×5	46	5×9	72	8×9	98	7×14	124	9×14
21	3×7	47	6×8	73	8×9	99	9×11	125	9×14
22	3×7	48	6×8	74	8×9	100	10×10	126	9×14
23	4×6	49	7×7	75	5×15	101	10×10	127	9×14
24	4×6	50	5×10	76	7×11	102	10×10	128	8×16
25	5×5	51	5×10	77	7×11	103	8×13		

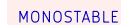
Appendix G

Schematics

This appendix contains the full hardware schematics for the Synapse-191 computer. Each schematic corresponds to a specific module described in Chapter 4 of this report, including the Control Unit, Register Driver, IO Module, and various registers and memory components.

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

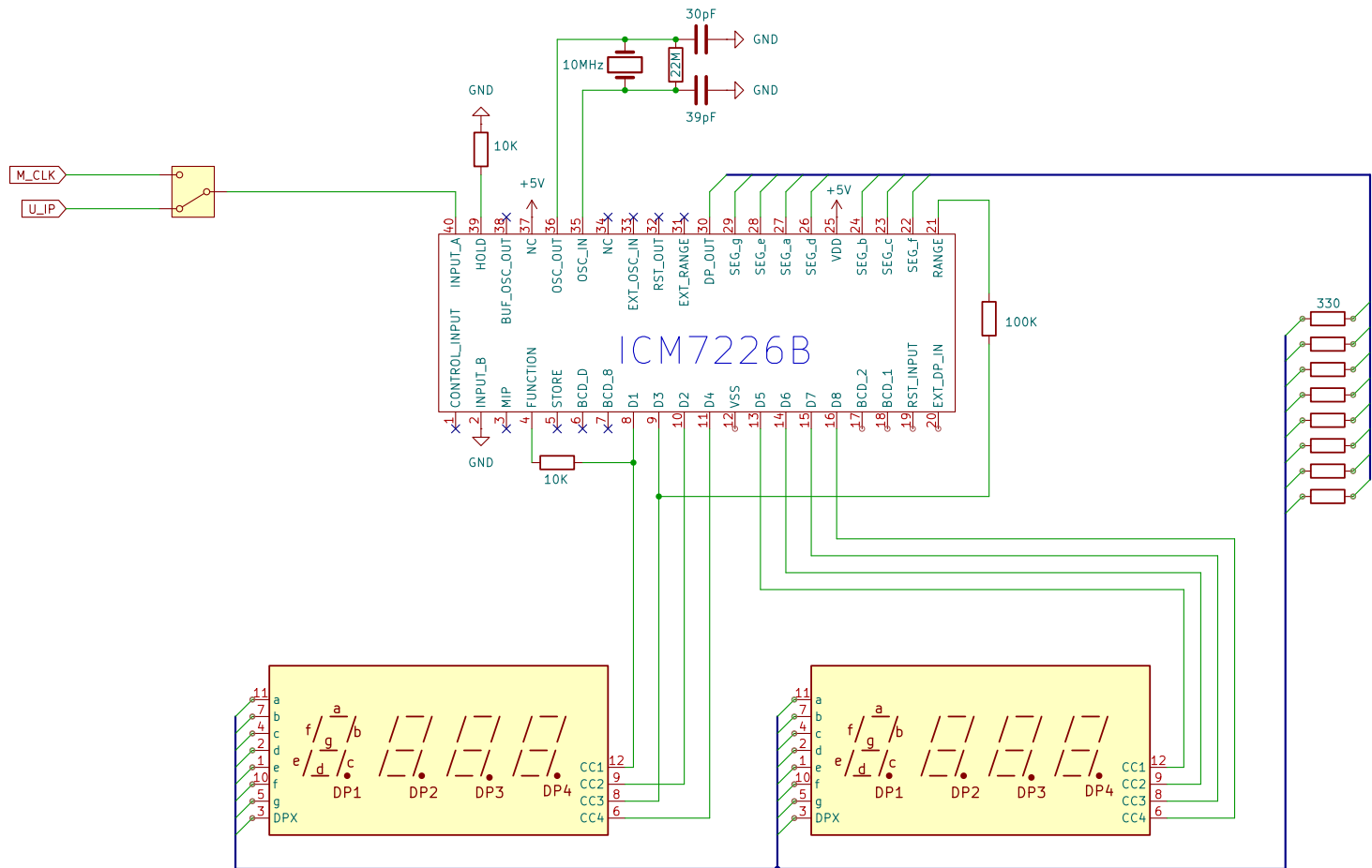
All schematics were created during the final build phase and reflect the working configuration of the system as tested.



J. Heit

Title:

Rev: 1
Id: 1/1



FREQUENCY DISPLAY MODULE

Sheet: /

File: timer.kicad_sch

Title:

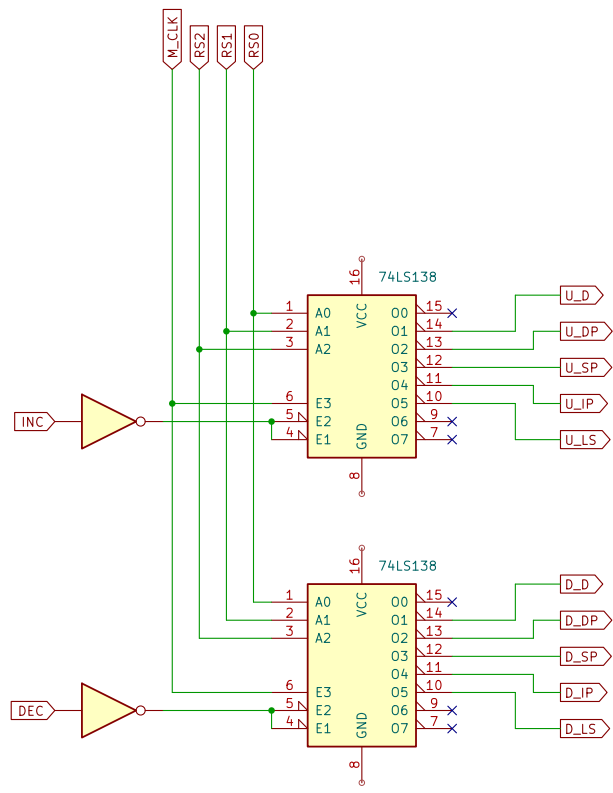
Size: A4

Date:

KiCad E.D.A. 8.0.9

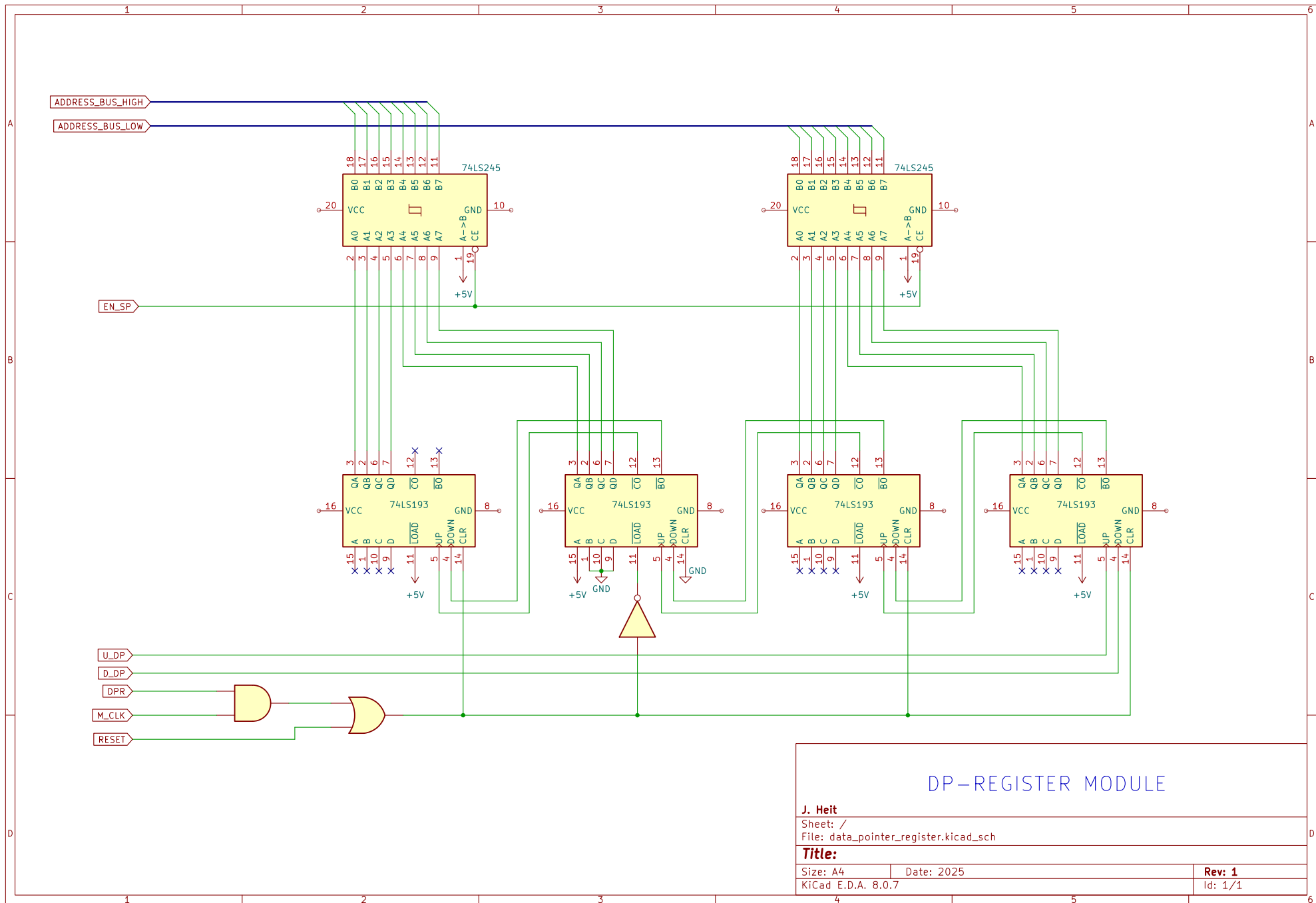
Rev:

Id: 1/1



REGISTER DRIVER

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



DP-REGISTER MODULE

J. Heit

Sheet: /

File: data_pointer_register.kicad_sch

Title:

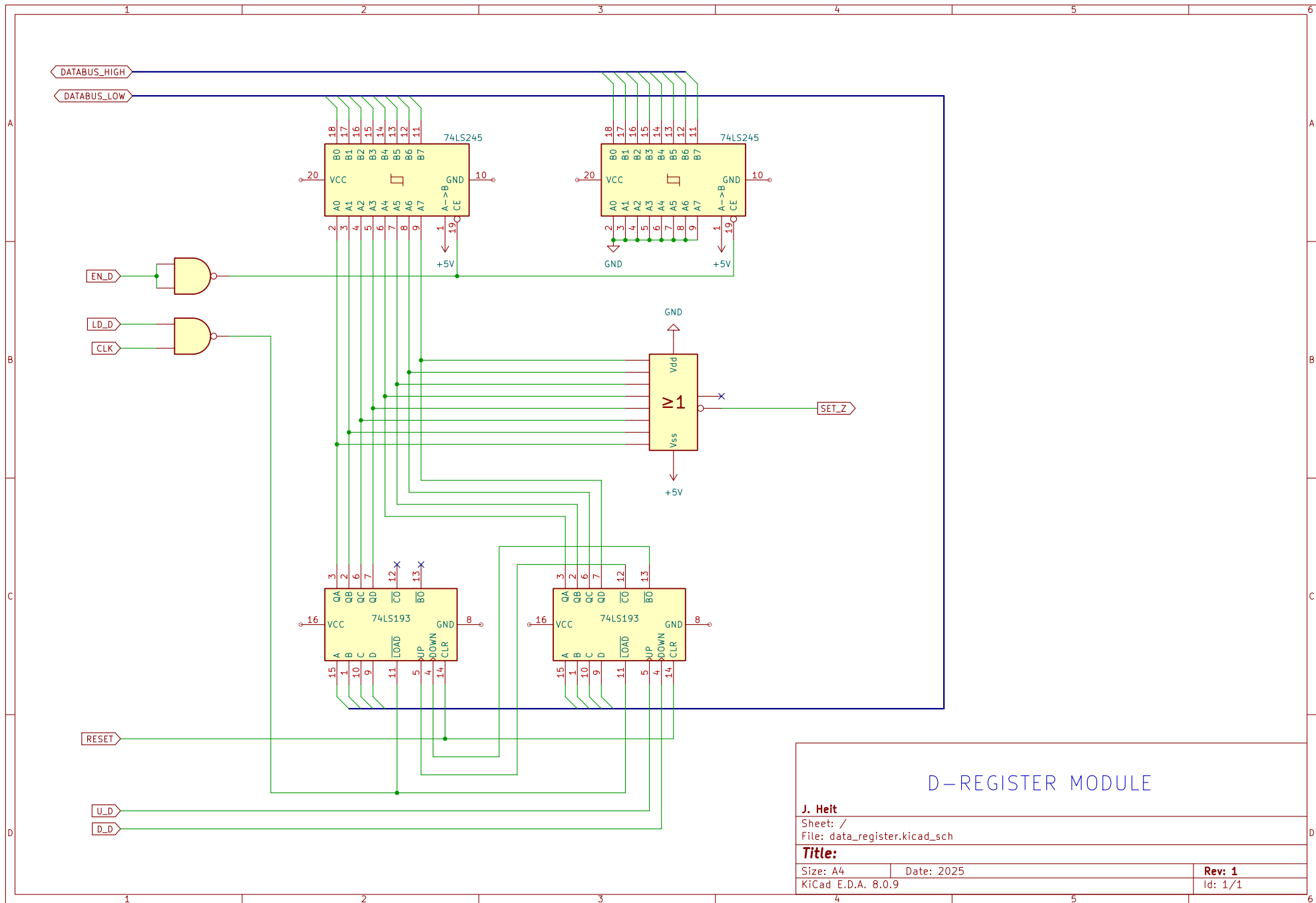
Size: A4

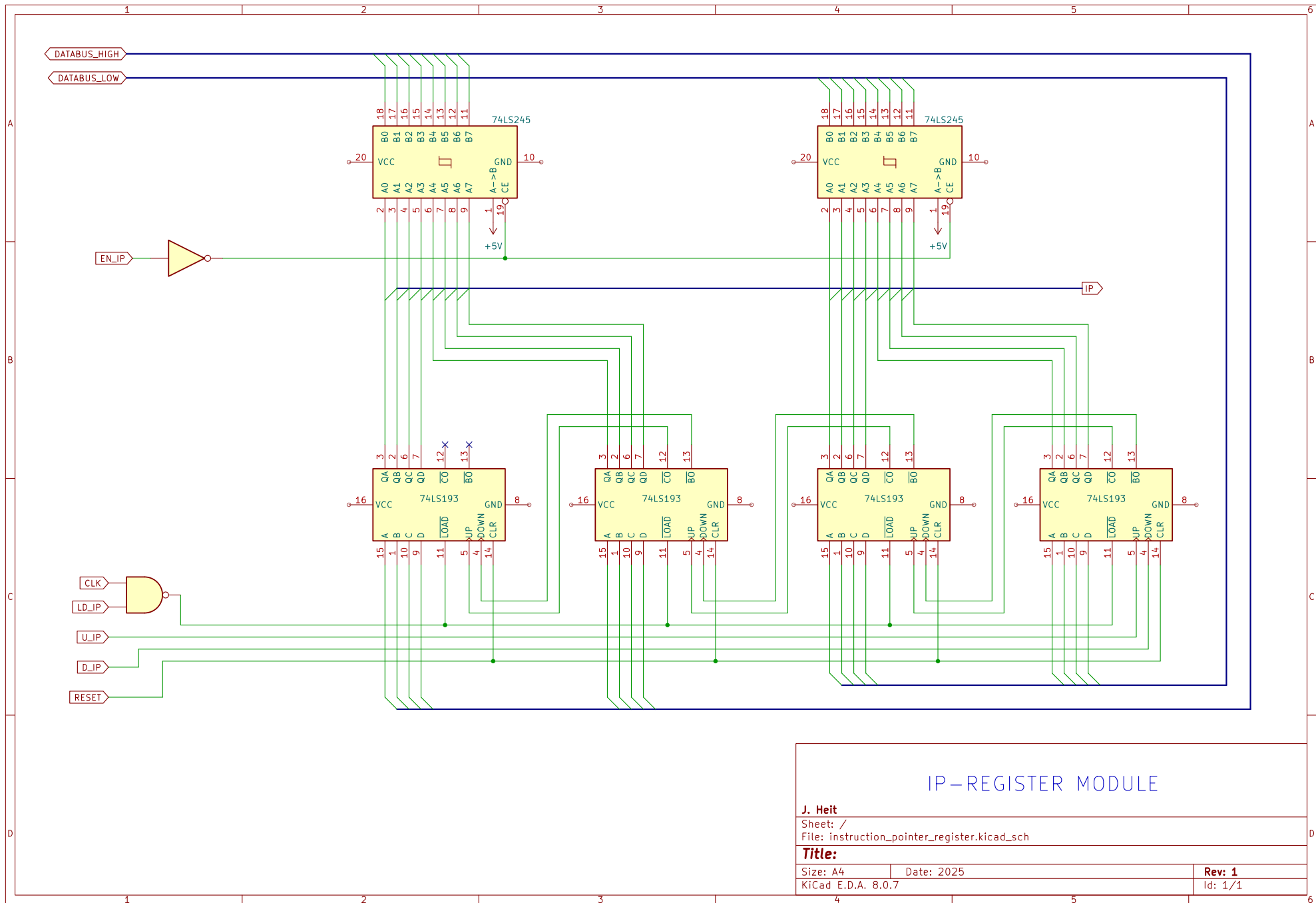
Date: 2025

Rev: 1

KiCad E.D.A. 8.0.7

Id: 1/1





IP-REGISTER MODULE

J. Heit

Sheet: /

File: instruction_pointer_register.kicad_sch

Title:

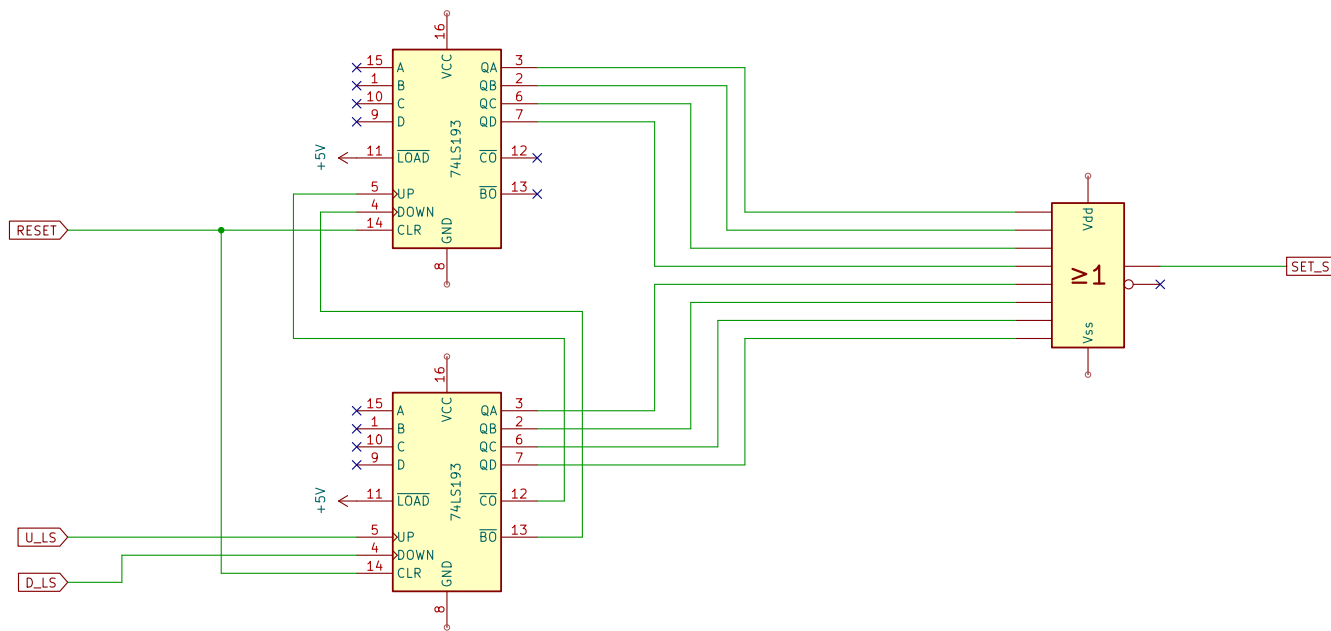
Size: A4

Date: 2025

Rev: 1

KiCad E.D.A. 8.0.7

Id: 1/1



LS-REGISTER MODULE

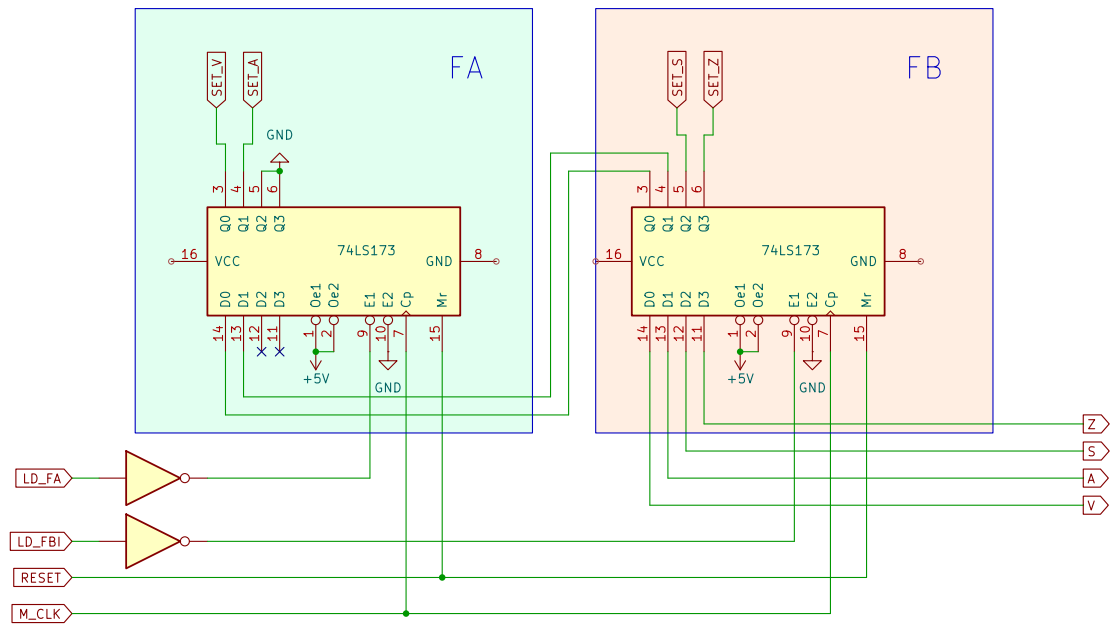
Sheet: /
File: loop_skip_register.kicad_sch

Title:

Size: A4
KiCad E.D.A. 8.0.9

Date:

Rev:
Id: 1/1



FLAG REGISTERS

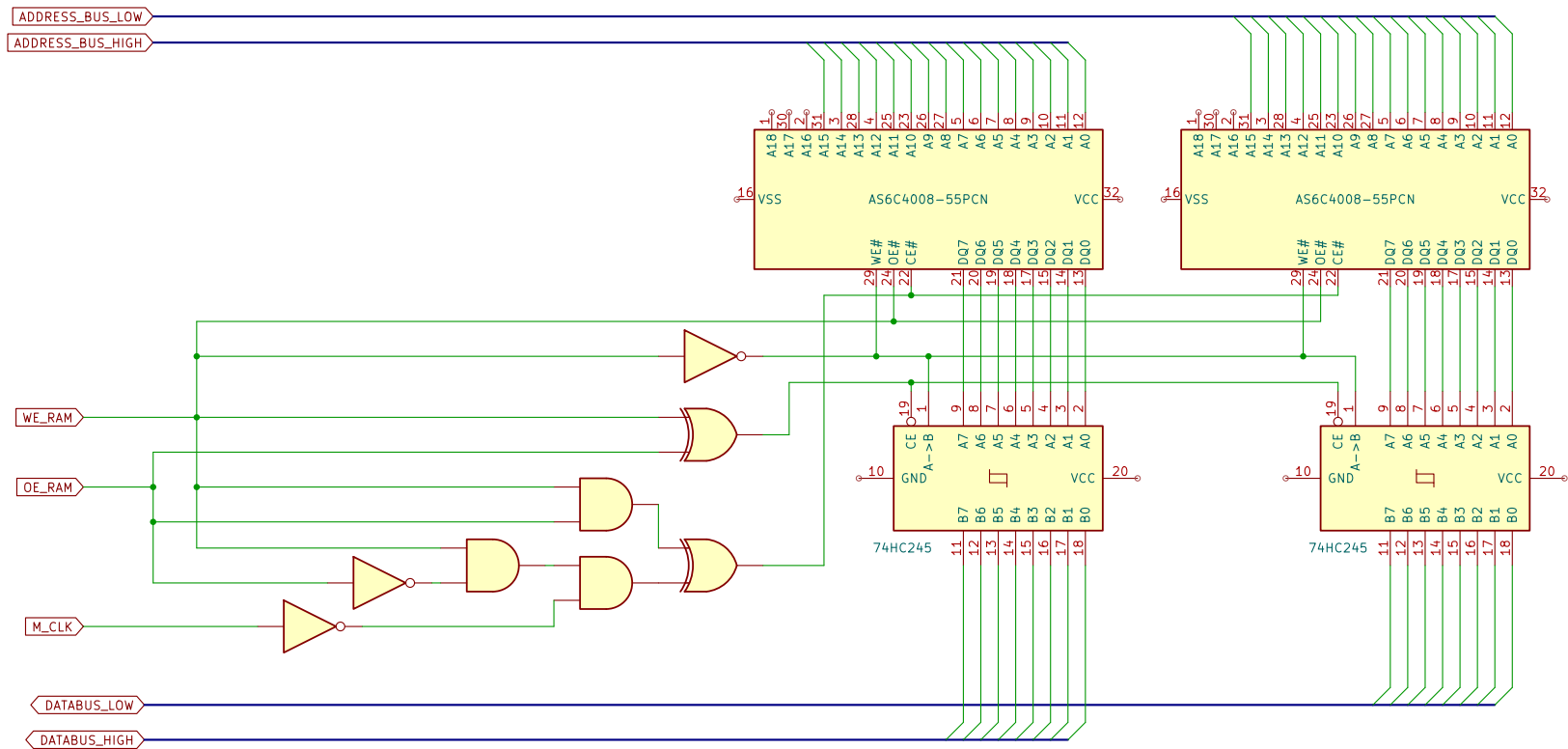
Sheet: /
File: flag_registers.kicad_sch

Title:

Size: A4
KiCad E.D.A. 8.0.9

Date:

Rev:
Id: 1/1



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

RAM MODULE

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

Sheet: /

File: ram.kicad_sch

Title:

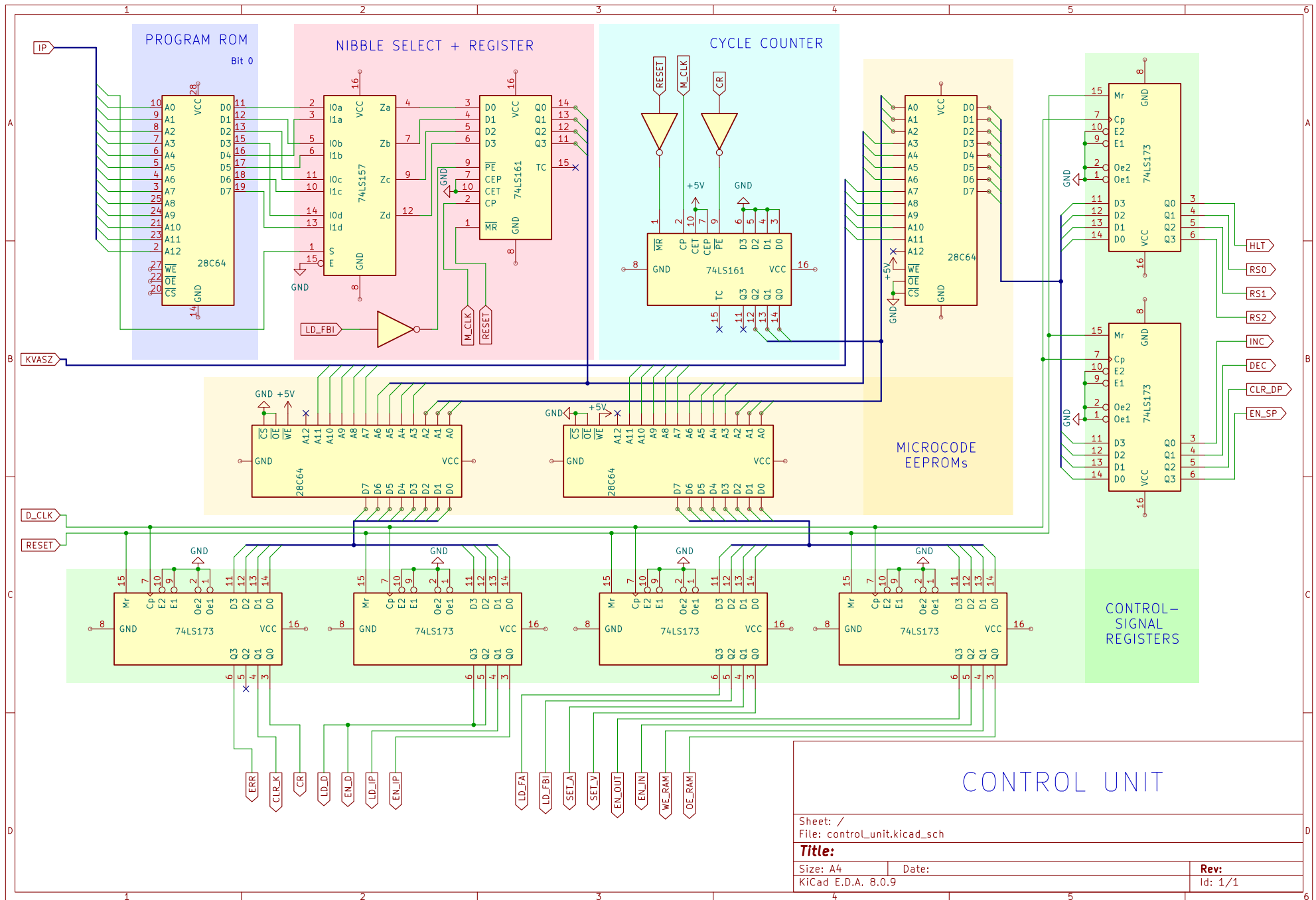
Size: A4

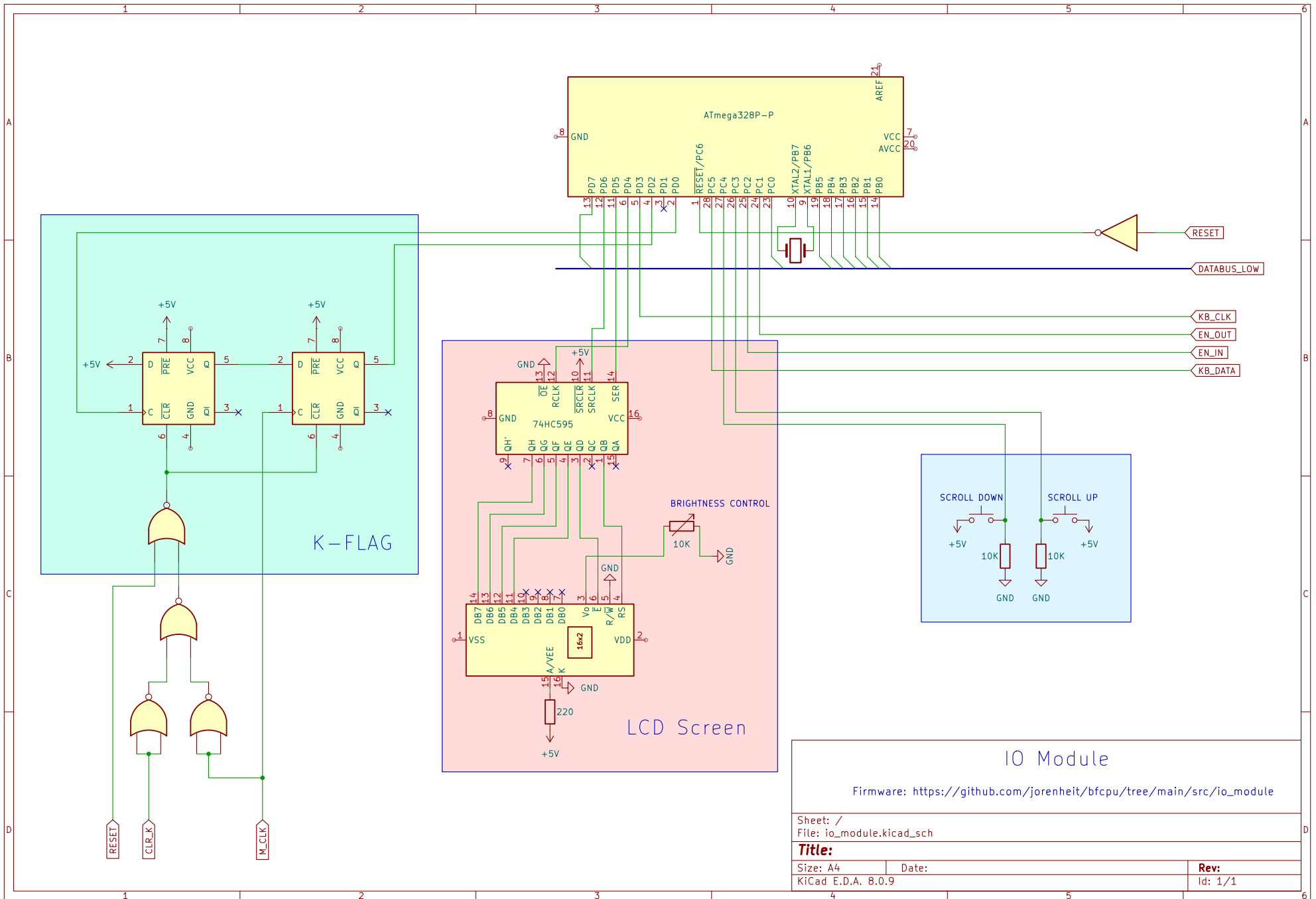
Date:

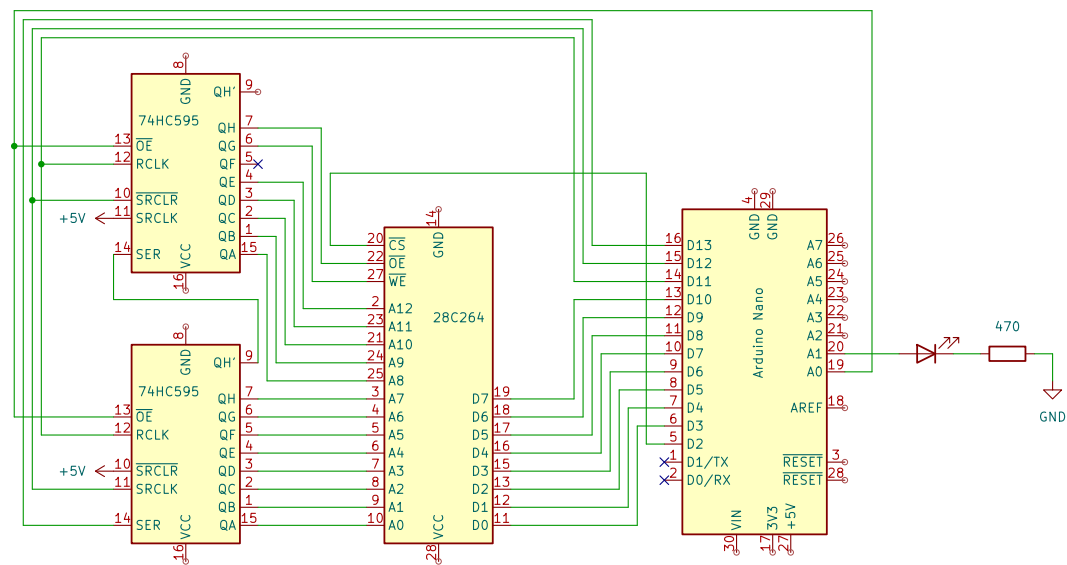
KiCad E.D.A. 8.0.7

Rev:

Id: 1/1







EEPROM PROGRAMMER

J. Heit

Sheet: /

File: eeprom_programmer.kicad_sch

Title:

Size: A4

Date: 2025

Rev: 1

KiCad E.D.A. 8.0.7

Id: 1/1

Appendix H

Flashback

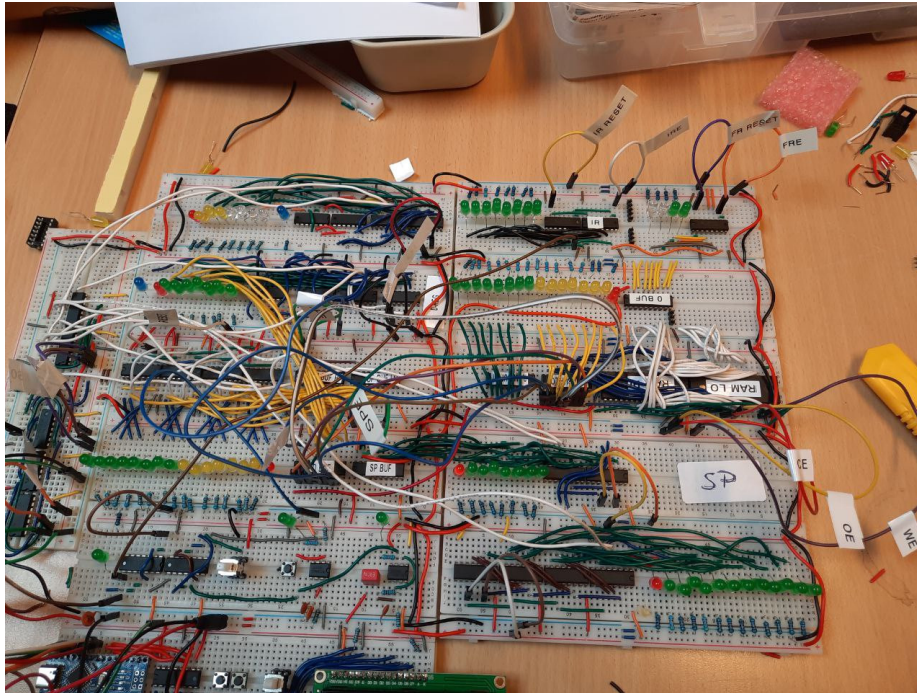


Figure H.1: Early prototyping in 2024.

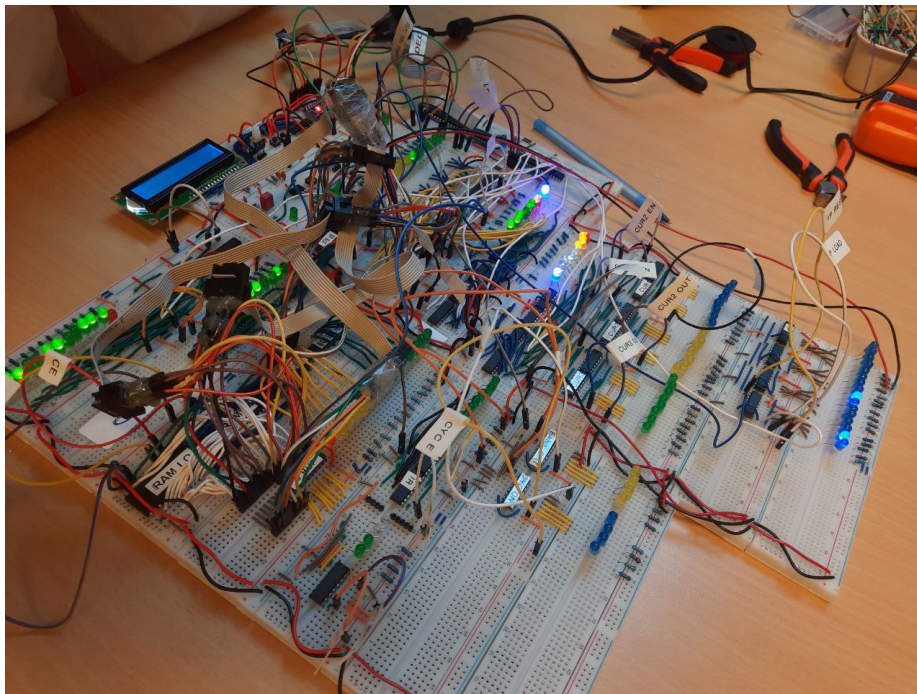


Figure H.2: Early prototyping in 2024.

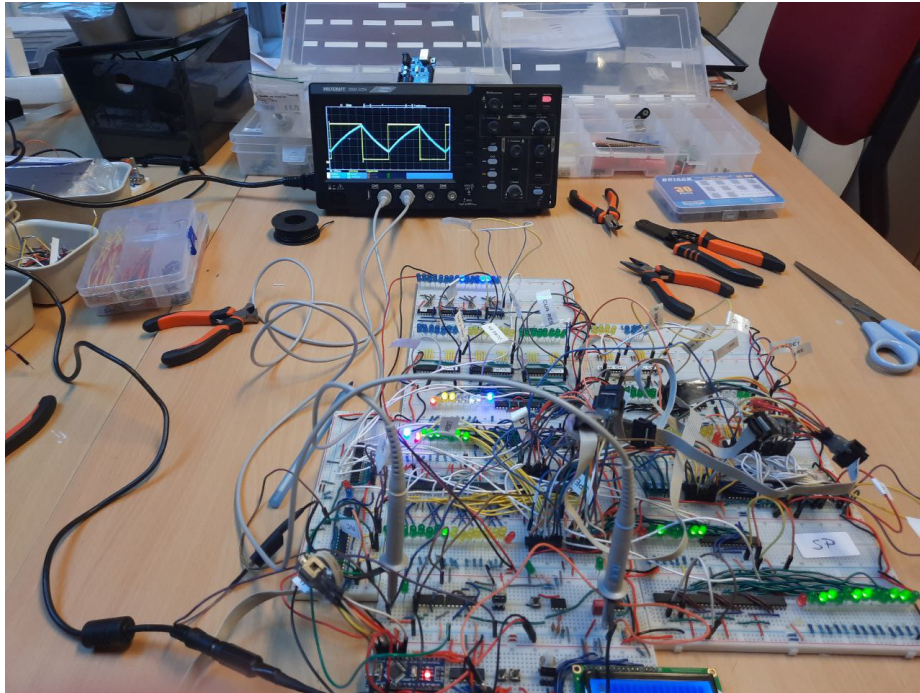


Figure H.3: Early prototyping in 2024.

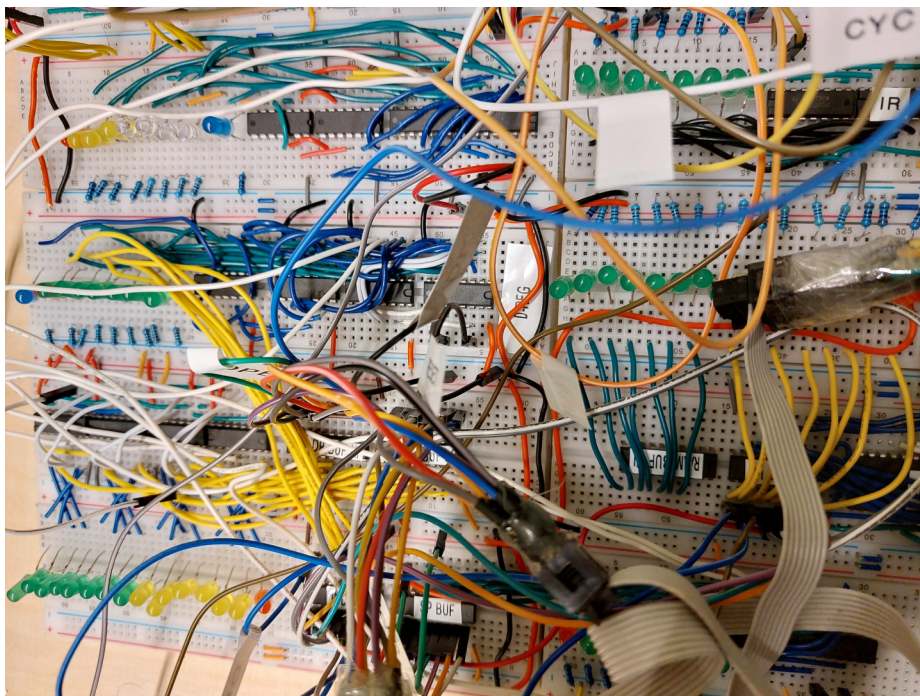


Figure H.4: Early prototyping in 2024.



Figure H.5: Wire stripping can quickly lead to a mess.



Figure H.6: A typical view when building and debugging the system.

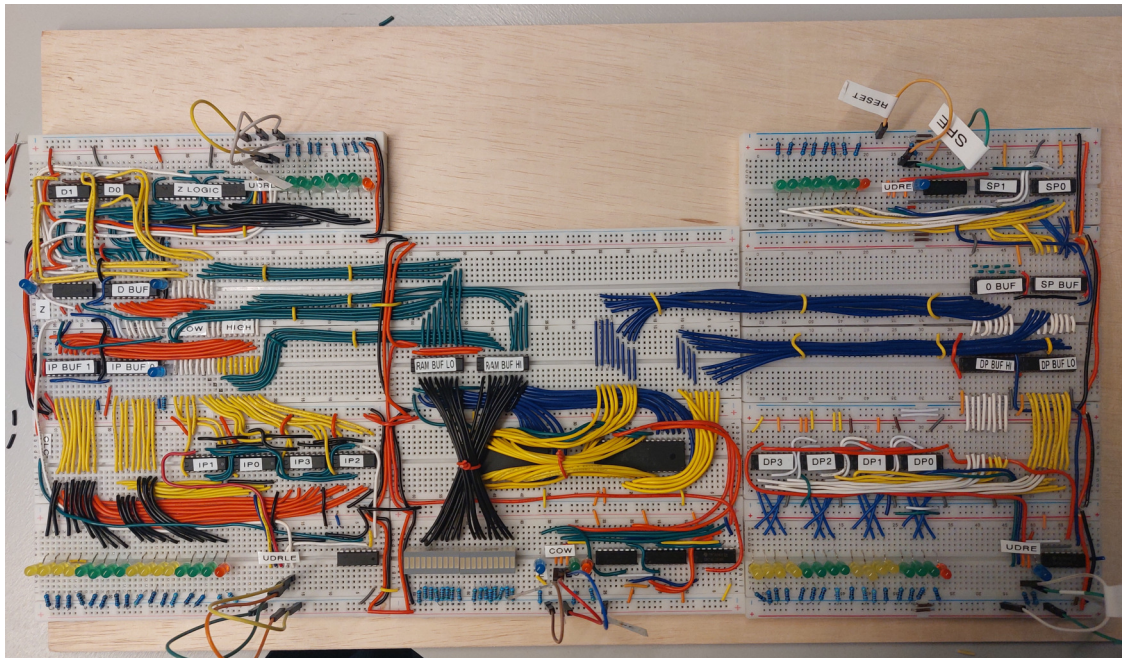


Figure H.7: Registers and RAM first put together, the system starts to take shape.

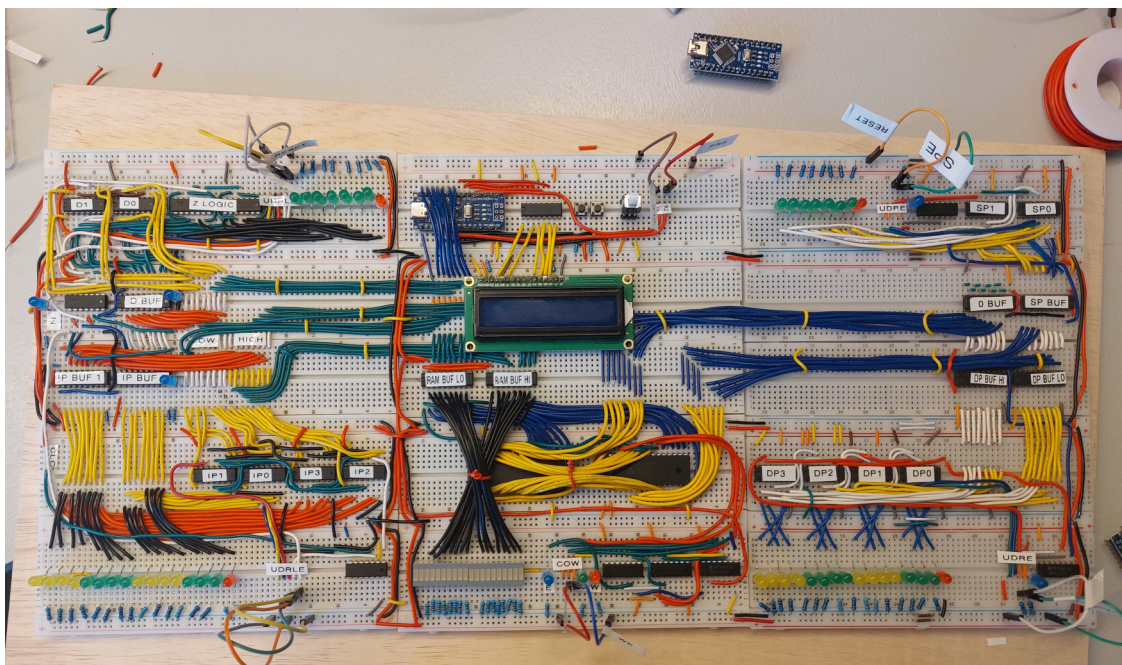


Figure H.8: An early version of the IO module, including an Arduino Nano, was added to the system.

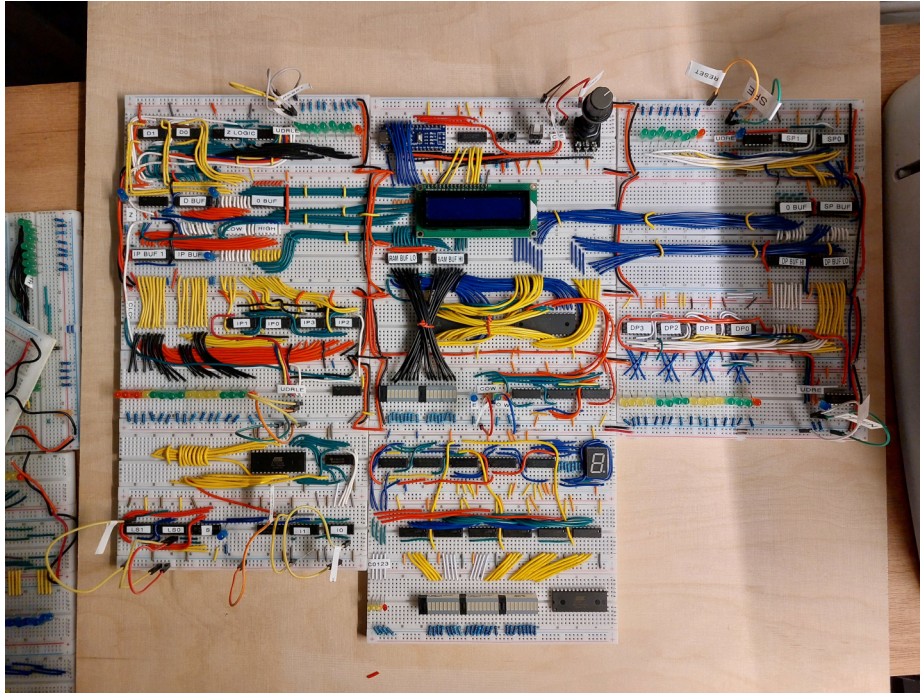


Figure H.9: The (segmented) control unit starts to take shape. The bottom-middle section contains the logic for sequentially fetching the three parts of the control word from different sections in the MC EEPROM chip.

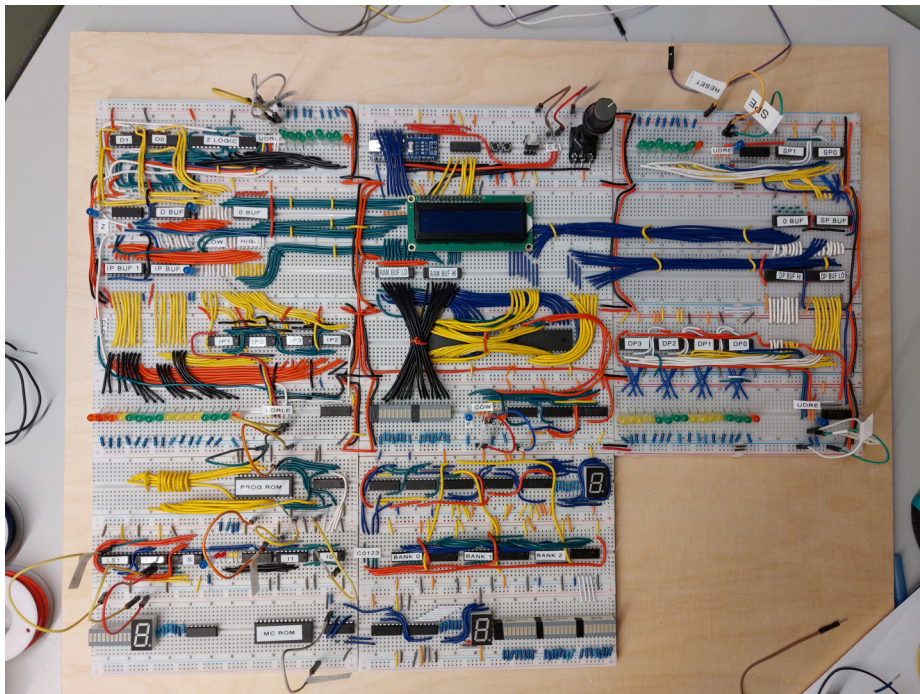


Figure H.10: The (segmented) control unit starts to take shape even more. A display for showing the current instruction was added (7-segment display, bottom left).

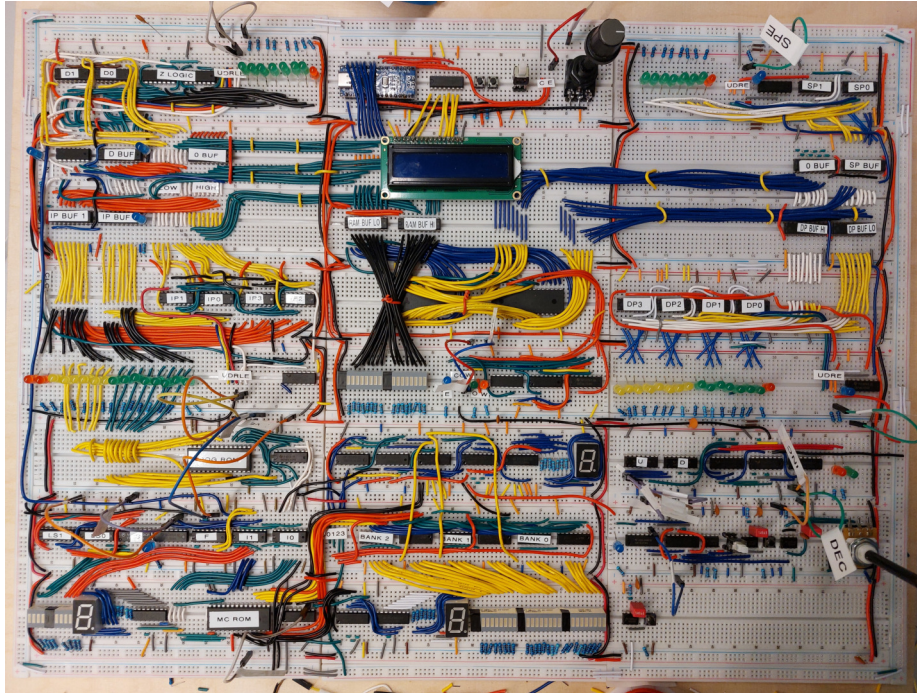


Figure H.11: First (somewhat) working version of the system.

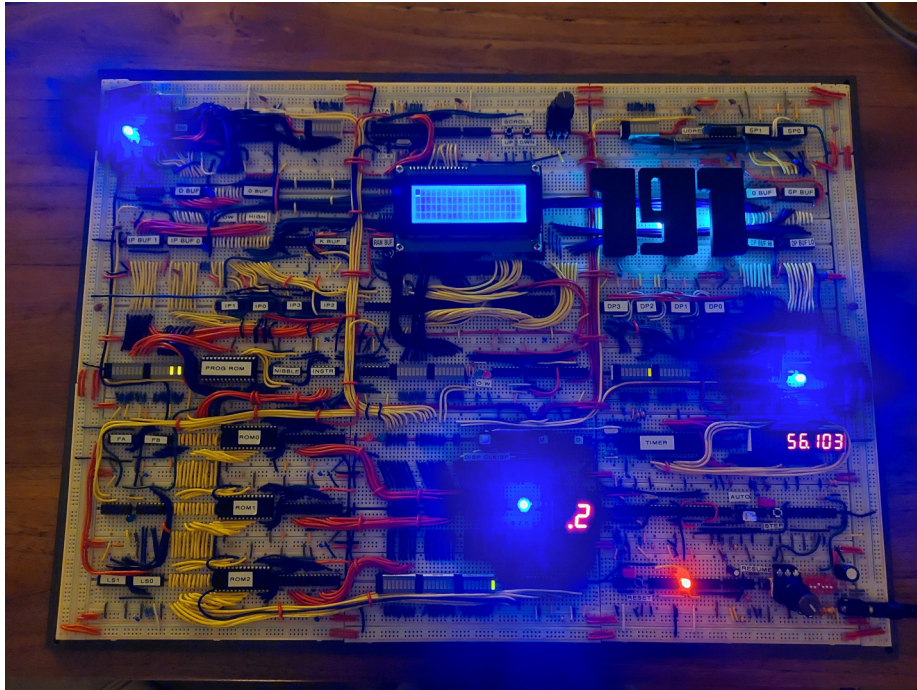


Figure H.12: Final build of the system, including 3D-printed parts and many pretty lights.

Bibliography

- [1] Wikipedia, *Brainfuck*, <https://en.wikipedia.org/wiki/Brainfuck>
- [2] Esolangs, *Brainfuck*, <https://esolangs.org/wiki/Brainfuck>
- [3] Esolangs, *Brainfuck*, https://esolangs.org/wiki/Random_Brainfuck
- [4] Wikipedia, *Von Neumann Architecture*,
https://en.wikipedia.org/wiki/von_neumann_architecture
- [5] Ben Eater, *Build an 8-bit computer from scratch*, <https://eater.net/8bit>
- [6] Stefan Heule, *How Many x86-64 Instructions Are There Anyway?*,
<https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/>, 2016
- [7] Daniel Mangum, *RISC-V Bytes: Introduction to Instruction Formats*,
<https://danielmangum.com/posts/risc-v-bytes-intro-instruction-formats/>, 2021
- [8] unknown author, *Wikipedia* <https://en.wikipedia.org/wiki/Brainfuck>
- [9] Joren Heit, *GitHub* <https://github.com/jorenheit/brainfix>
- [10] Joren Heit, *GitHub* <https://github.com/jorenheit/acus>
- [11] Wikipedia, *First-fit bin packing* https://en.wikipedia.org/wiki/First-fit_bin_packing
- [12] Marijn Stevens, *GitHub*,
<https://github.com/MarijnStevens/brainfuck-collection/blob/master/bf/golden2.b>
- [13] Daniel B. Cristofani, *brainfuck.org*, <https://brainfuck.org/e.b>
- [14] Troemax, *Reddit*,
https://www.reddit.com/r/brainfuck/comments/847vl0/prime_number_generator_in_brainfuck/
- [15] Alexandru, *Stack Exchange*,
<https://codegolf.stackexchange.com/questions/84/interpret-brainfuck>
- [16] Mixtela, *GitHub*, <https://github.com/mitxela/bf-tic-tac-toe>
- [17] Texas Instruments, *Quadruple 2-Input Positive-NAND Gates*, 74LS00 Data Sheet, Dec. 1983 (Rev. 2003)
- [18] Texas Instruments, *Quadruple 2-Input Positive-NOR Gates*, 74LS02 Data Sheet, Dec. 1983 (Rev. 1988)

- [19] Texas Instruments, *Hex Inverters*, 74LS04 Data Sheet, Dec. 1983 (Rev. 2004)
- [20] Texas Instruments, *Quadruple 2-Input Positive-AND Gates*, 74LS08 Data Sheet, Dec. 1983 (Rev. 1988)
- [21] Texas Instruments, *Hex Schmitt-Trigger Inverters*, 74LS14 Data Sheet, Dec. 1983 (Rev. 2016)
- [22] Texas Instruments, *Quadruple 2-Input Positive-OR Gates*, 74LS32 Data Sheet, Dec. 1983 (Rev. 1988)
- [23] Texas Instruments, *BCD-To-Seven-Segment Decoders/Drivers*, 74LS48 Data Sheet, Mar. 1974 (Rev. 1988)
- [24] Texas Instruments, *Dual D-Type Positive-Edge-Triggered Flip-Flops with Preset and Clear*, 74LS74 Data Sheet, Oct. 1983 (Rev. 1988)
- [25] Texas Instruments, *Dual JK-Flip-Flops with Preset and Clear*, 74LS76 Data Sheet, Dec. 1983 (Rev. 1988)
- [26] Texas Instruments, *Quadruple 2-Input Exclusive-OR Gates*, 74LS86 Data Sheet, Dec. 1972 (Rev. 1988)
- [27] Texas Instruments, *Retriggerable Monostable Multivibrators*, 74LS123 Data Sheet, Dec. 1983 (Rev. 1988)
- [28] Texas Instruments, *3-Line to 8-Line Decoders/Demultiplexers*, 74LS138 Data Sheet, Dec. 1972 (Rev. 1988)
- [29] Texas Instruments, *Quadruple 2-Line to 1-Line Data Selectors/Multiplexers*, 74LS157 Data Sheet, Dec. 1982 (Rev. 2022)
- [30] Texas Instruments, *Synchronous 4-Bit Counters*, 74LS161 Data Sheet, Oct. 1976 (Rev. 1988)
- [31] Texas Instruments, *4-Bit D-Type Registers with 3-State Outputs*, 74LS173 Data Sheet, Oct. 1976 (Rev. 1999)
- [32] Texas Instruments, *Synchronous 4-Bit Up/Down Counters*, 74LS193 Data Sheet, Dec. 1972 (Rev. 1988)
- [33] Texas Instruments, *Octal Bus Transceivers With 3-State Outputs*, 74LS245 Data Sheet, Oct. 1976 (Rev. 2016)
- [34] Texas Instruments, *8-Bit Shift Registers With 3-State Output Registers*, 74HC595 Data Sheet, Dec. 1982 (Rev. 2021)
- [35] Texas Instruments, *xx555 Precision Timers*, NE555 Data Sheet, Sep. 1973 (Rev. 2014)
- [36] Atmel, *64K (8K x 8) Parallel EEPROM with Page Write and Software Data Protection*, AT28C64B Data Sheet, 2009
- [37] Alliance Memory Inc., *512K X 8 BIT LOW POWER CMOS SRAM*, AS6C4008 Data Sheet, Aug. 2009
- [38] Motorola, *B-Suffix Series CMOS Gates*, MC14078B Data Sheet, Aug. 2009
- [39] Intersil, *8-Digit, Multi-Function, Frequency Counter/Timer*, ICM7226B Data Sheet, Aug. 2009

