# Building a Computer from Scratch

https://yurix.ch/blog

December 3, 2023

## 1 Abstract

Modern computers are truly a miracle. On an area of less than two by two centimeters, they manage to pack billions of transistors connected up in a complicated system able to aid in almost any computational task. Today, they play an important role in almost every aspect of our life. This work aims to understand and then build a computer from scratch out of electronic components.

It is of course not realistic to use billions of transistors for this purpose. But computers have not always been this complicated. In the 1960s, it was not uncommon to build computers from a few hundred logic chips implementing logic gates, connected up by wires. The 1970s brought more advanced logic chips implementing whole function units like counters in one package. Using these technologies, it is still possible to build a simple computer today. Besides building the computer hardware, an instruction set and electronic design for the computer were created, as well as the tools required to program and interact with it.

By modern standards is this computer not powerful, but by comparing its specifications it should be roughly comparable to the Apollo Guidance Computer, the computer used to land the first people on the moon. Additionally, to visualize and teach about what is going on when the computer executes a program, it includes LEDs to show the state of all major components.

While a homebrew computer like this lacks any practical applications, it can still be used to demonstrate many principles found in computer design and computer science.

# Contents


**1 Abstract** **1**

**2 Introduction** **4**
2.1 Prerequisites . . . 4
2.1.1 Logic Gates . . . 4
2.1.2 Flip-Flops . . . 4
2.1.3 Propagation Delay . . . 4
2.1.4 Word . . . 5
2.1.5 ALU . . . 5

**3 Technology** **5**
3.1 Technology Choice . . . 5
3.1.1 Option A: Transistors . . . 5
3.1.2 Option B: Glue-Logic . . . 6
3.2 Glue-Logic: Building Logic Circuits Like Lego . . . 6
3.2.1 Glue-Logic-Chip-Families . . . 6
3.2.2 Data Topology . . . 6
3.2.3 Bus-Hold-Circuit . . . 8
3.2.4 Memory . . . 9

**4 Architecture** **10**
4.1 TTAs . . . 10
4.2 Connecting Technology and Architecture . . . 10
4.3 Micro-Architecture . . . 11
4.3.1 Execution Stages . . . 11
4.3.2 Memory-Mapped Computation . . . 13
4.3.3 Data Paths . . . 15
4.3.4 Putting it together . . . 16
4.4 Function Units . . . 19
4.4.1 Memory . . . 19
4.4.2 Subtraction . . . 19
4.4.3 Control Flow . . . 19
4.4.4 SPI . . . 19
4.4.5 EEPROM . . . 20

**5 Programming** **20**
5.1 Assembly Source Code . . . 20
5.1.1 Moves . . . 21
5.1.2 Raw Constants . . . 21
5.1.3 Labels and Pointers . . . 21
5.1.4 Numbers . . . 23
5.1.5 Other Instructions . . . 23

**6 Construction** **23**
6.1 Breadboards . . . . . 24
6.2 Wire Management . . . . . 24
6.3 Electronic Design . . . . . 25

**7 Results** **25**

**8 Review** **27**
8.1 Construction . . . . . 27
8.2 Architecture . . . . . 27
8.2.1 No Position Independent Code . . . . . 28
8.2.2 16-Bit instead of 8-Bit . . . . . 28
8.3 Time Management . . . . . 28

**9 Conclusion** **29**

**10 Sources** **29**
10.1 Literature . . . . . 29
10.1.1 With Author . . . . . 29
10.1.2 Without Author . . . . . 29
10.2 List of Figures . . . . . 30

# 2 Introduction

Since the construction of the first computers in the 1940s, they have become ever more powerful and capable. Today, they play an integral part in our lives and will probably continue to increase in importance. Therefore, a basic understanding of how computers work can be beneficial. A common first step in understanding how a computer works is to learn how to program one. Software plays an important part, as it allows the usage of computers for various useful tasks, but it cannot run without the underlying hardware.

The question of how the hardware works has interested me for a long time, but the knowledge I acquired always stayed theoretical. When the matura project approached, I took the chance to apply this interest to a project, and the idea to build a computer formed in my head.

My goal was to **build a computer**. It is not to **teach how a computer works**. This means the report focuses on the specifics of the computer I built instead of generally explaining how computer hardware works. There are plenty of books that do a better job of teaching this.

## 2.1 Prerequisites

Computer hardware is a huge topic, and this report cannot focus on explaining all fundamentals. A few important terms are explained here, but this might not be enough. In this case, I recommend the reader to do his own research. Furthermore, some programming and electronic knowledge is recommended.

### 2.1.1 Logic Gates

Logic gates are the equivalent of operators like addition, multiplication, or division, but for binary signals. They are the basic building blocks out of which a computer is constructed. Common logic gates are for example AND, which outputs HIGH if all inputs are HIGH, or NOT, which turns LOW into HIGH and HIGH into LOW.

### 2.1.2 Flip-Flops

A flip-flop is a circuit out of logic gates able to store a signal. They are in a sense a 1-bit RAM cell and are used in the computer for temporary results.

### 2.1.3 Propagation Delay

No logic gate is perfect. When an input signal is changing, the outputs do not flip immediately. Instead, there is a certain delay, the propagation delay, before the signal propagates through a logic gate.

### 2.1.4 Word

When computers perform arithmetic, they use fixed-width numbers. A fixed-width number only has a certain fixed number of digits. Assuming a computer can represent 4-digit decimal numbers, it can represent the numbers 0000 to 9999, but 10'000 cannot be represented since it doesn't fit into 4 digits. A computer using `n` bits is often called an n-bit computer, for example with 64 bits a 64-bit computer.

A word is the term used to refer to the kind of fixed-width number a computer can use the most efficiently. The computer in this report is a 16-bit computer. Therefore, the size of the words it uses is also 16 bits, and it operates the most efficiently if all numbers fit within 16 bits.

### 2.1.5 ALU

ALU stands for arithmetic-logic unit and is the part of the computer that performs mathematical and logical calculations.

# 3 Technology

## 3.1 Technology Choice

Producing a computer as an integrated circuit "chip" is out of reach for a matura-project, as the technology is financially out of reach for private customers. Even "cheap" IC manufacturing offers for prototyping, research and education like those offered by Efabless Corporation costs from $10'000 upwards[1].

However, looking at the history of computing, there have been other ways of building computers. In the very early days computers were built out of discrete transistors because manufacturing integrated circuits was not yet possible. Additionally, in the 1970s and 1980s, there was a time when some computers were made out of so-called glue-logic. This was a series of small and simple integrated circuits implementing functions like logic gates that then can be combined to build complete computers. It is still possible to build a computer out of transistors or out of glue logic today.

### 3.1.1 Option A: Transistors

Using transistors means going all the way down to the lowest level of what's happening in terms of electronics in a computer. This poses a few interesting electronic engineering challenges, but the number of transistors required for even the simplest processor is very high. For example, the Intel 4004 processor from 1971 already has more than 1000 transistors despite being one of the simplest computers possible[2]. Additionally, large parts like registers will be

[1] efabless.com: efabless, 11.4.2023
[2] Intel 4004, in: Wikipedia, 9.6.2023

identically repeated next to each other over and over, so I suspect that when building a computer out of transistors I will loose interest in the project while soldering it together. For this reason, I decided against building a computer out of transistors.

#### 3.1.2 Option B: Glue-Logic

Glue-Logic is a bit like Lego: It is a family of integrated chips that can be combined almost arbitrarily to form complex systems from simple parts. It's relatively cheap, easily available and can be used on breadboards, allowing rapid prototyping and experimentation. Because I never attempted an electronics project of a comparable size, being able to change and iterate rapidly is important. But even more important, glue-logic offers an almost perfect level of abstraction for this project: There are parts implementing logic gates, but also more complex parts implementing whole registers or adders. One could argue that registers or adders can be built from logic gates, but then we again run into the trap of having to spend too much time and resources on assembling the computer, like with transistors. Being able to avoid this trap with slightly higher-level components is exactly what I'm searching for for this project. These factors combined shaped my decision to use glue logic for the computer.

### 3.2 Glue-Logic: Building Logic Circuits Like Lego

While glue-logic is almost as simple as Lego, there are a few caveats to consider when using it.

#### 3.2.1 Glue-Logic-Chip-Families

First of all, multiple companies sell different series of glue logic. The most widespread, and the one I decided to go with, is the 74-series of integrated circuits. First introduced in the 1960s, the 74-series has multiple sub-families operating at different speeds and using different manufacturing technologies to accommodate the advancement in manufacturing since then. Chips from different families aren't always electrically compatible with chips from other families, so sticking with one family is advisable. My choice fell on the HC-family. Its CMOS-based manufacturing technology offers low power consumptions at decent speeds. While it isn't the fastest family anymore, it is easy to source, cheap and has a very diverse range of chips covering all possible functions.

#### 3.2.2 Data Topology

It is common that components conditionally need to receive inputs from different sources. This is a problem with the push-pull outputs of the 74HC family. When outputting a logical LOW, they connect ground via a low resistance directly to the output. Conversely, a logical HIGH means a low resistance connection from the power supply to the output. If multiple outputs are directly connected, e.g. to both send a signal to an input of another IC, it can happen that one output

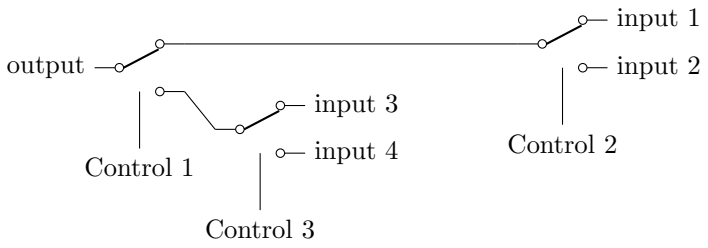


Figure 1: Multiplexer-tree reducing four inputs into one output. More stages for more inputs can be added at the expense of propagation delay.

writes a logical HIGH while the other writes a logical LOW. This forms a low resistance connection between the power supply rails, leading to a high current flow through the ICs which damages them. Generally, there are two solutions to this problem: multiplexers and tri-stated buses.

A multiplexer can be imagined like a selector switch between multiple positions: An input selection signal selects one of multiple inputs which is forwarded to an output. This is perfectly safe, as at no time multiple outputs can be active and cause a short. The disadvantage is that usually, the number of inputs that can be reduced to one output by a single multiplexer is limited, usually to something like 2:1 or 4:1. This can be solved by creating a tree of multiplexers (see fig. 1) at the expense of propagation delay, as the signal has to travel through multiple multiplexers. Additionally, it makes the wiring more challenging, as usually there aren't more than four 2:1 or two 4:1 multiplexers on one chip. This means wires are more likely to need to connect to multiple different chips. Spread-out cables lead to more chaos in the wiring because they can't be bundled together, each wire has to go individually. Also, the cables are also more prone to fall out or lose contact, whereas a group of wires would support each other mechanically.

Another solution is using a bus. In this context a bus is a term used to describe multiple parallel wires with a similar purpose, for example carrying the bits of a byte, that connect multiple things. It solves the problem of connecting multiple outputs to a single input by using tri-state-buffer outputs. Besides the usual HIGH and LOW push-pull outputs, they have a third state, HIGH-Z. In this state, the output is effectively disconnected, and other signals can be applied without causing a short. As long as it is ensured that all except one tri-state output is in HIGH-Z mode, multiple outputs can be connected to a bus without a problem. Ensuring that only one output is in a push-pull mode can be done by using a decoder. Another issue arises when all outputs are in HIGH-Z. In this case, the bus is floating, meaning it has neither a HIGH nor a LOW level. These in-between levels can damage CMOS logic gates because they activate a HIGH and LOW level at the same time, creating a short. Despite these disadvantages, a tri-state-buffer-based system also offers a few advantages. Compared to a tree

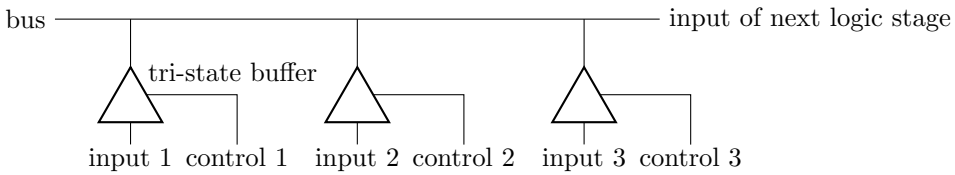


Figure 2: Tristate-based bus with 3 inputs. Only one control signal can be active at a time without causing a short. The concept can be expanded to any number of inputs, giving a linear propagation time regardless of input count.

of multiplexers, the propagation time of the signals connected to a bus remains constant regardless of the number of signals. Additionally, wiring is easier since a single tri-state buffer can handle 8 bits at a time, making it possible to bundle wires. An example of a tri-state-bus-based system can be seen in figure 2.

### 3.2.3 Bus-Hold-Circuit

A tri-state-buffer-based system is desirable due to its superior propagation delay and wiring capabilities. The problem of only allowing one output to be active at a time can easily be overcome by using a decoder to control the activations of the buffers. However, especially with large buses the amount of buffer outputs and decoder outputs are most likely not equal, leaving some decoder states unconnected. This can lead to a situation where no input is passed through to the bus, creating a floating condition that potentially harms the subsequent inputs.

The easiest solution is to add pull-up or pull-down resistors. This works great for low frequencies, but at higher frequencies the capacitance of the wire gets too large for the pull-down resistor to pull the bus to a valid level within a clock cycle. There is a frequency range where reducing the resistance of the push/pull resistor allows running at higher frequencies, but there is a limit to how low you can go with the resistance, as the signal level quality decreases while the power usage increases.

For the clock frequency of 1 MHz I targeted, pull-up/pull-down resistors would still be more than enough even with a very conservative estimation of breadboard capacitance. However, the chips used in this computer also have rise and fall time requirements, meaning that a transition between HIGH and LOW has to happen within a certain time. Achieving these times using pull-down/pull-up resistors is impossible, as the resistance of the pull-up/pull-down resistor would need to be unacceptably low.

One solution is to use a bus-hold-circuit[3]. In a bus hold circuit the bus is connected to a buffer that's always active. This way, it acts like an amplifier. It picks up the signal from the bus and outputs a clean version of it. This output

[3]Texas Instruments (Hg.): An Overview of Bus-Hold Circuit, S. 8

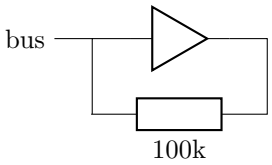


Figure 3: Bus-Hold circuit

is through a resistor connected back to the bus. If no signal is written to the bus, the output of the buffer is fed back to its inputs, thus keeping the bus on a well-defined level, and avoiding a floating bus. Once a new signal is written to the bus, this signal is stronger than the buffer's output since the buffer's output has to pass through a resistor first. Therefore, we have the new signal on the bus and additionally, the buffer starts to pick up the new signal as well. This amounts to a bus that can transport data like without a bus-hold-circuit, but it also keeps its old signal once no signal is applied to the bus, in this way eliminating the issue of not having any signal on the bus. Figure 3 shows an example of a bus-hold circuit.

A bus-hold-circuit also works at higher frequencies because even though the output of the buffer is connected through a high resistor to the bus, the bus always already has the same level as the buffer output. This means the bus-hold-circuit never needs to pull the bus to an opposite state. For this reason, the capacitance of the wires connected and the resistance of the resistors used don't matter.

#### 3.2.4 Memory

A fundamental problem in modern computers is the memory latency. With increasing clock speeds in the gigahertz range the memory for instruction fetching must get faster too. Since memory speeds haven't increased at the same rate as processor clock speeds, modern processors include caches to prevent being slowed down by memory. A cache is a small and fast memory containing a copy of parts of the main memory. When the processor needs a certain value from memory, it first looks in the fast cache before resorting to the slow main memory. A cache has a limited capacity, so sometimes the processor still has to load data from the slower main memory, but the overall number of slow reads is greatly reduced. Bandwidth on the other hand is rarely a problem, as more memory access lanes can be added.

When building this computer, I encountered the exact opposite. Memory latency is no problem at all, the memory chips used in the computer are one of the fastest components used in the whole processor. Bandwidth however is a problem. The options for suitable memory chips, meaning memory chips with a parallel interface and in a breadboard-mountable form factor, are very limited. All of them have in common that they only have one eight-bit wide memory

lane. This means we can either read or write 1 byte per clock cycle. This is not a lot. For example, the modern RISC-V architecture consumes 4 bytes per instruction[4]. With the memory chips I used it would take 4 clock cycles just for fetching the instruction.

As will be shown in later sections, these differences between modern computers and the technology available for my project had a strong influence on the design decisions.

# 4 Architecture

## 4.1 TTAs

This computer is based on a transport-triggered architecture, short TTA. Much inspiration was taken from Douglas W. Jones' article on *The Ultimate RISC*[5], describing a simple TTA-based computer. The computer described here is a very close relative to it.

TTAs differ significantly in how they work compared to a typical register or accumulator machine. A register machine program specifies operations that are applied to data, for example, `NEGATE R1 -> R2`. It doesn't matter what `R1` or `R2` mean in this example, the significant part is that the instruction includes an explicit action: `NEGATE`.

A TTA on the other hand specifies data movements. A TTA instruction is very similar to a move instruction in a register machine: `MOVE source, destination`. However, TTAs only offer this instruction and nothing else, while register machines also have other instructions.

Moving data between different storage sites alone is not enough to perform computations. TTA architectures solve this by performing computation as a side-effect of the data movements. You can imagine this like a car washing plant: While the car drives through it, it gets cleaned. While moving through, an action was performed on the car. Something similar is done in a TTA with data: While it is being moved around, it gets processed.

A TTA instruction is a movement, which can be defined as having a source and a destination. This is also what a TTA instruction executed by the computer represents: It contains a source and a destination for a data movement. An action, like in a register-machine, can be omitted because only a single action, `MOVE`, exists. It is therefore not necessary to explicitly say it is a move.

For the following sections, it can be assumed the movements happen mostly within memory.

## 4.2 Connecting Technology and Architecture

Because the memory latency in the available technology is negligible and a memory access can happen within one clock cycle, the computer is designed

[4] Waterman/Asanovìc: RISC-V Instruction Set, (2017) S.5

[5] Douglas W. Jones: The Ultimate RISC

as a direct memory-memory architecture. This means that values don't have to be loaded into registers before the processor can operate on them like in a register-register architecture. In modern processors, having few fast registers is faster than having the whole memory space available, because the latency of accessing memory is much higher than the latency of accessing one of typically 32 registers. In the computer described here, we have a different situation. The memory latency is negligible compared to logic propagation times. It is faster if we save the overhead of first loading a value from memory into a register and directly operate on the values in memory.

Using a memory-memory architecture comes with other trade-offs though. For example, the instructions must be wider since we need to include a much larger memory address instead of a register number. With a 64KB-address-space like in my computer, this means we need a 16-bit address instead of a 5-bit register number like in a register machine with 32 registers. On the other hand, we also save the OP-Code since we don't need to specify what action we want to perform in a transport-triggered architecture. The operation to perform is already included in the 16-bit memory address. In total we need two memory addresses for a TTA instruction, a source and a destination, therefore giving us a total instruction size of 32 bits. This is the same as in register machines like RISC-V[6] or ARM[7].

A disadvantage of this approach is that we need multiple cycles to fetch a single instruction. The memory chips used here have a low bandwidth of only one byte per clock cycle, meaning we'd need 4 cycles for a single 32-bit wide instruction. Two memory chips are connected in parallel to overcome this problem. The first memory chip holds the lower 8 of the 16 bits required for a single memory address while the second chip holds the upper 8 bits. This means half an instruction can be fetched per cycle. In total, we need two cycles to fetch one instruction. By using two memory chips, the bandwidth problem gets cut in half, but it isn't eliminated.

## 4.3 Micro-Architecture

The following section explains the details of how the computer executes instructions in hardware. This structure is often called the micro-architecture.

### 4.3.1 Execution Stages

After we load the instruction telling the computer what to do, we have to execute the instruction. Since this computer employs a transport-triggered architecture, executing an instruction means moving data around. Moving data around consists of two stages: Getting the data, and then storing the data at the new location.

The source of the data might be located in the memory. Therefore, we need to plan with an additional memory cycle to be able to retrieve data from sources

[6] Waterman/Asanovìc: RISC-V Instruction Set, (2017) S.5
[7] ARM: ARM Architecture Reference Manual, About the Instruction Set

in memory. The second part of executing the instruction is storing the data at the destination, which might also be located in the memory. To be able to serve these destinations, we need another memory cycle.

These four steps are already all the steps the CPU goes through:

1. load the source address (part 1 of an instruction)
2. load the destination address (part 2 of an instruction)
3. load the data to be moved located at the source address
4. store the data at the destination address

To know where in memory the source and destination are located, the computer internally has a counter that counts up after each instruction is executed, thus pointing to the next instruction to be executed. This counter is often referred to as the instruction pointer since it points to the instruction to be executed.

The pseudo-code to simulate this architecture is the following:

```
instruction_pointer = 0
while (true) {
    // first stage: load source address
    source_address = memory[instruction_pointer]
    instruction_pointer += 1
    // second stage
    destination_address = memory[instruction_pointer]
    instruction_pointer += 1
    // third stage: load data
    temporary_data = memory[source_address]
    // fourth stage
    memory[destination_address] = temporary_data
}
```

There is still one small optimization left to apply before we reach what is implemented in hardware. It is desirable to reduce the number of different variables used in the pseudocode as much as possible, as each variable requires one additional flip-flop. We have to apply a few transformations to do this: Stage 3 only depends on inputs from stage 1. It doesn't matter if we execute stage 2 or 3 first. We can therefore swap the order of these two stages.

```
instruction_pointer = 0
while (true) {
    // first stage: load source address
    source_address = memory[instruction_pointer]
    instruction_pointer += 1
    // second stage: load data
    temporary_data = memory[source_address]
```

```
    // third stage
    destination_address = memory[instruction_pointer]
    instruction_pointer += 1
    // fourth stage
    memory[destination_address] = temporary_data
}
```

With this, we can reduce the number of variables. The source address is used in stages 1 and 2 while the destination address is used in stages 3 and 4. Their usages are therefore not overlapping and we can use one variable for both. With this optimization, we get the following pseudo-code:

```
instruction_pointer = 0
while (true) {
    // first stage: load source address
    temporary_address = memory[instruction_pointer]
    instruction_pointer += 1
    // second stage: load data
    temporary_data = memory[temporary_address]
    // third stage
    temporary_address = memory[instruction_pointer]
    instruction_pointer += 1
    // fourth stage
    memory[temporary_address] = temporary_data
}
```

This algorithm is the core of the computer! It can be found implemented in hardware on the right side of figure 4. .

### 4.3.2 Memory-Mapped Computation

Until now, the computer is limited to moving around data in memory. This is not enough to perform arithmetic operations like addition or subtraction. Transport-triggered architectures solve this problem by performing computations using memory mapping. This is a technique where not all memory addresses are backed by actual memory. Rather, certain memory addresses are connected to special registers that instead of storing data compute something with the data.

I like to imagine memory mapping as a wall of mailboxes. Mailboxes are almost like memory. You put a letter in and later you retrieve it. Most of the mailboxes on this wall are normal mailboxes. A few of them are memory-mapped and therefore special. While from the outside these special boxes look the same as the other mailboxes next to them, "memory-mapped mailboxes" have a machine built into them that opens your letter, reads it, and writes a summary of the letter. When you open up this special mailbox, it gives you

[9] Abb. 4: Micro-Architecture-Core

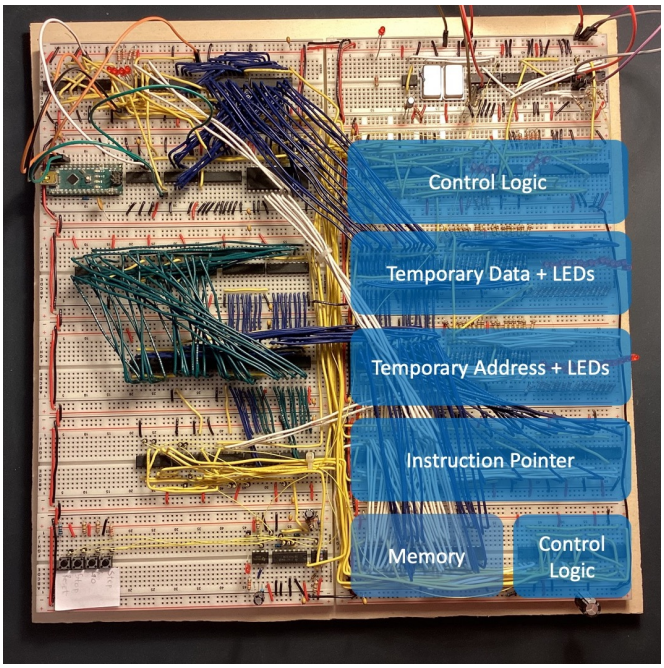


Figure 4: The core of the computer responsible for initiating movements[9]. Control logic controls the computer, so it is doing the right thing at the right time.

the summary of the letter instead of the normal letter. The mailbox takes in data (the letter), computes something based on it and returns the result of the computation instead of the original data.

In a computer, we don't want to summarize letters, but we want to for example add numbers. Addition requires two operands. So to continue the analogy of a wall of mailboxes, we have two mailboxes beside each other that work together. You put one operand into each mailbox. They are connected inside and work together to add up the two numbers you put into them. Next, you open another connected mailbox next to them where you will find the result of the addition. This is roughly how operations like addition work in the computer described here.

Of course, instead of addresses with a street and city, the computer uses memory addresses to identify a "mailbox".

We don't have mailboxes in hardware, but we have registers we can write values to. Since most operations require multiple registers that work together, a group of registers that form the interface for a computational unit is often referred to as a function unit. A function unit always has one or more registers to interface with and a task it performs, for example, addition.

In a sense, memory can be treated as a function unit as well. Memory doesn't

behave different than a function unit with many general-purpose registers that store values and can be written to and read from. The difference is that adding more than 65,000 register chips to a breadboard-based computer is not feasible but adding 2 memory chips is.

Function units are one of the main reasons I decided to go with a transport-triggered architecture approach. Since function units are one unit on their own and have a simple interaction surface with the rest of the computer, they allow for a great amount flexibility. After the core responsible for the movements between registers is built, it is possible to add and remove function units without having to plan for them from the start. Each function unit is an independent piece of hardware that doesn't depend on any other function unit. When I started building the core, I had no idea how I would implement the SPI input-output system, but it still allowed me to confidently start building the core, knowing I could add them later without giving too much thoughts back then.

### 4.3.3 Data Paths

When considering memory as a function unit, we are always moving data between function units. Therefore, the inputs and outputs of function units must be connected. This is done by the so-called data bus. However, data isn't moved directly: It first moves to either the temporary data or the temporary instruction register described in section 4.3.1 about execution stages. These temporary registers can be found at the top of the data bus. Besides the data bus, two more control lines are part of the data bus, controlling whether we are moving a value from a function unit to these registers or the other way around.

There also needs to be a way for function units to know when to release or accept data, as multiple function units talking at the same time leads to invalid data. For this purpose, there is another bus parallel to the data bus controlling the activation of function units. This so-called address-bus selects the function unit we are reading from or writing to.

The content of the address bus has two sources depending on which execution stage we are in. During the two instruction fetch stages (which load the source and destination of the move operation) the address bus contains the instruction pointer. Most of the time, the instruction pointer will point to memory, but in theory it can execute from any function unit. In reality, only memory makes sense though. As soon as we signal a read and have the instruction pointer on the address bus, we begin loading the instruction on the data bus. At the end of the execution stage this value is then saved in the temporary instruction register. In the next execution stage we can swap the content of the address bus to the just-updated temporary instruction register. As a consequence, we are now reading or writing to the address specified in the instruction, allowing us to perform the move operation.

Both of these busses are visible in the computer as seen in figure 6. The data bus is built out of blue wire and spans almost the entire computer, from the core to each function unit. The address bus is built out of white wire and can mostly be found around the core which performs the movements.

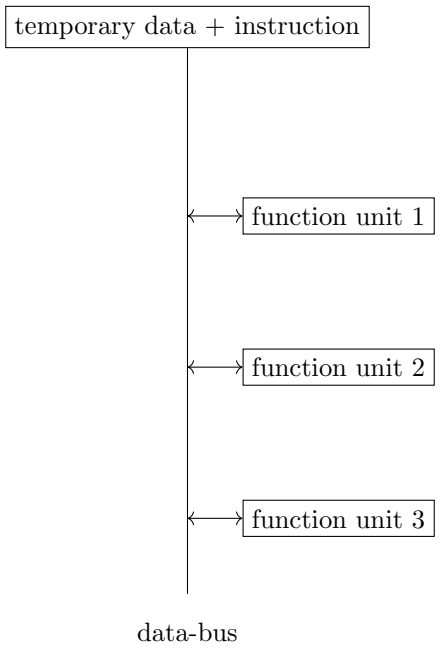


Figure 5: data-bus with three function units

#### 4.3.4 Putting it together

This is everything required to understand how the computer works at its core. To put everything together, we will step through one instruction using figure 7 as a visualization. Additionally, figure 8 helps to visualize where in the computer the actions are taking place. We will need to go through all four execution stages.

The first stage is the source fetch stage. The computer loads the address of the data to be moved. The source address usually resides somewhere in memory, the exact location is specified in the instruction pointer. For this reason, the control logic instructs the instruction pointer to release its value onto the address bus and informs all function units that we are reading data. Since the address on the address bus is within the range of the memory function unit and we are reading, it retrieves the value and releases it on the data bus. At the end of the cycle, the temporary address register takes the value from the data bus and stores it. Additionally, the instruction pointer is incremented by one.

The next stage is the data fetch stage. The computer loads the data to be moved. As soon as we switch to this stage, the control logic stops instructing the instruction pointer to output its value. Instead, it tells the temporary address

[10]Fig. 6: Data Paths visualized
[12]Fig. 8: All major parts and their function labeled

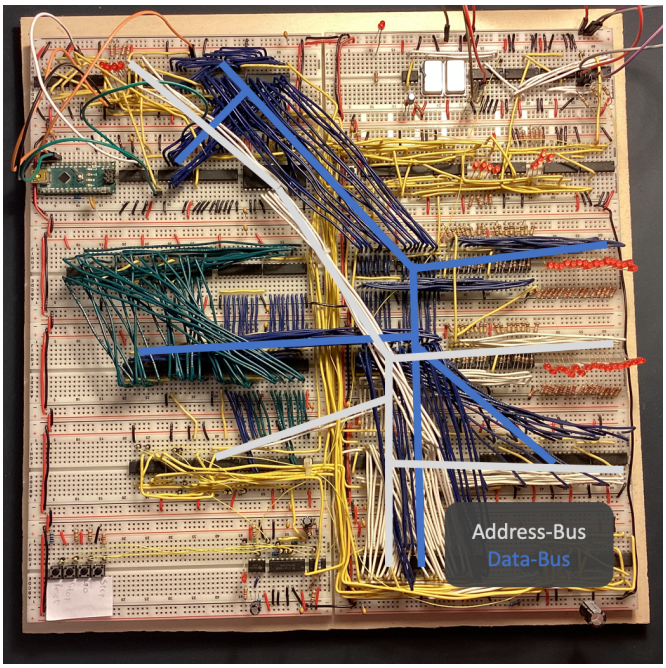


Figure 6: The address and data bus visualized in the cpu.[10]

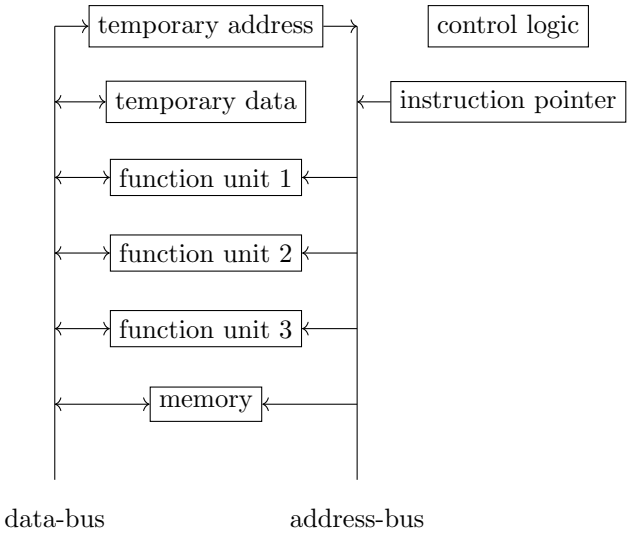


Figure 7: Complete micro-architecture of the core of the computer put together

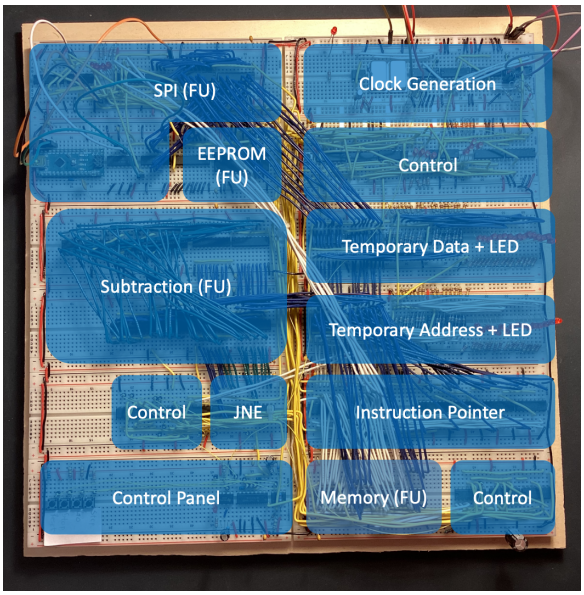


Figure 8: All major parts of the computer and their function labeled. FU stands for function unit, JNE stands for jump not negative and is part of the logic for conditional execution. Control is the logic responsible for initiating the right actions at the right time. [12]

register to release its value onto the address bus. The control logic still tells the function units we are reading. This means one function unit will recognize that it should output data on the data bus. At the end of the cycle, the value released on the data bus will be stored in the temporary data register.

The third stage is the destination fetch stage. It performs the same steps as stage one to load the second part of the instruction into the temporary address register.

The last stage is the write stage. The data loaded in stage two will be written to the function unit stored in the temporary address register. This stage differs quite a bit from the others. The control logic instructs the computer to release the contents of the temporary address register onto the address bus. At the same time, it instructs the temporary data register to release the data onto the data bus. Instead of the read control signal it enables the write control signal this time. This means that the selected function unit doesn't output a value, instead, it listens for values on the data bus, receiving the contents of the temporary data register. At the end of the cycle, the function unit takes the value it just received and stores it for either computation or in the case of memory retrieval later on.

With this, we have successfully gone through one instruction. To execute the next instruction, we jump back to stage one.

## 4.4 Function Units

Until now, we have always talked about function units and how they control computation, but there hasn't been an explanation of what kind of function units are available. The following section aims to explain this.

### 4.4.1 Memory

This is the only function unit that is required to execute any program, as it holds the program to execute. The computer has 64 kilobytes of memory organized as 32768 2-byte-words. They are addressed through the function unit addresses 32768 to 65535. Half of the function unit address space is taken up by the memory.

### 4.4.2 Subtraction

Due to time constraints, the only arithmetic operation available is subtraction. However, this is enough to also perform addition by calculating $a - (0 - b) = a - (-b) = a + b$. To perform a subtraction, the two operands must be loaded into the `ALU_A` and `ALU_B` function unit register. The result from the subtraction can be loaded from the `ALU_SUB` register.

### 4.4.3 Control Flow

Being able to control the instruction pointer allows for constructing loops or conditional execution like if-statements. Even though the instruction pointer is part of the core, it can be written like any other function unit to perform a jump in the instruction sequence. The computer supports two kinds of jump instructions to control the control flow: conditional and unconditional jumps.

Unconditional jumps are achieved by writing the new instruction pointer to the `IP` register located at address 28672. Unconditional means that the instruction pointer is always updated with the new value.

Conditional jumps only update the instruction pointer if a condition is fulfilled. In this computer, a conditional jump is initiated by loading the value that should be checked into the `ALU_A` register. `ALU_A` plays a role in subtraction but also holds conditions for conditional jumps. In a second step, the new value for the instruction pointer is moved to the `CIP`(conditional instruction pointer) register similar to an unconditional jump. However compared to conditional jumps, writing to `CIP` only updates the instruction pointer if the contents of `ALU_A` are not equal to 65535.

### 4.4.4 SPI

A computer that can't interact with the outside world isn't that useful. In small microprocessors like the Arduino, a commonly used interface is SPI, short for serial peripheral interface. I initially planned to add support for SPI to the computer. However, I ended up misunderstanding the SPI protocol specification

and implemented a protocol that's very close to SPI, but not quite the same. The small difference is significant enough that the computer is unable to interface with SPI devices. The protocol can still be used to communicate with a "big" computer using an Arduino as an translator from an SPI-like protocol to USB. This is also the primary way to download and execute programs.

#### 4.4.5 EEPROM

Until now, we have always assumed that we have a program in memory to execute. But since the memory is implemented in SRAM technology, it doesn't retain it's content when losing power. The software running on the computer has to be reloaded after each startup. To simplify this process, there is one function unit that provides a small read-only-memory containing a program, called the loader, that loads the actual software to execute from a computer via SPI.

The read-only memory is provided by an EEPROM chip. EEPROM stands for electronically erasable programmable read-only memory. It is mainly a read-only-memory but can also be reprogrammed.

The EEPROM is connected starting from address 4096 and has 64 bytes of addressable memory. Programming a loader program for the EEPROM has a catch though: The EEPROM can just control 8 bits of the 16 required for a full source or destination address of an instruction. All other bits are set to 0. By wisely choosing which bits of the address bus the EEPROM can address, it is still possible to use all function units. At the same time, it makes it difficult to program and the loader must remain fairly small.

# 5 Programming

An assembler is available for programming. This is a program that takes in a human-readable form of the instructions that should be executed and then translates it into the binary the computer executes. Assembly programs are very close to the hardware and features typically found in programming languages like if-statements, loops and variables are not available. The assembler offers only the most fundamental features required to reasonably program the computer.

## 5.1 Assembly Source Code

The assembler takes in a text file containing the source code. An example could look like this:

```
// This program forms an infinite loop
main:
    // loop around forever
    mv &main_addr, IP
    main_addr: raw_const @main
```

The first line begins with a comment. Everything behind the // until the end of the line is ignored and can be used for documentation. The following lines contain instructions for the assembler of what the program does. The following section describes the most important assembler instruction.

### 5.1.1 Moves

Since the computer is a TTA, the most basic operation is moving data around. This fundamental operation is therefore also available in the assembler using the `mv` keyword. `mv` takes two parameters, the source and the destination. A full move instruction can look like this:

```
mv 5, 6
```

In this example, it means we move a value from address 5 to address 6.

### 5.1.2 Raw Constants

A move instruction consists of two words in the constructed executable, the source and the destination. However, sometimes you might want to emit just one word. This can be done with the `raw_const` instruction. It takes the word to emit as an argument.

```
raw_const 42 // emits the word 42
```

### 5.1.3 Labels and Pointers

To perform a jump or load constants into for example the subtraction unit, it is required to specify a concrete value that should be moved into the function unit instead of a source address from where the value should be fetched. This can be done by placing the value to be moved somewhere at the end of the program and then using the address of the location where the value is stored as a source:

```
// abstract
mv address_of_data_to_move, destination     // address: 0
raw_const data_to_move                      // address: 2
```

When executing the move instruction, it loads the value at address `address_of_data_to_move` from memory. Since we defined `address_of_data_to_move` to be the memory address where our constant `data_to_move` is stored, we will read `data_to_move` from memory. In the next step, it is then stored at the destination as usual.

The assembler can't understand pseudocode though. Therefore, we insert actual values by hand:

```
// with addresses filled in:
mv 2, 444                                   // address: 0
raw_const 42                                // address: 2
```

This works and can be done but has a few catches: First, it is not visible what the 2 stands for. It has a clear meaning as the address of the constant, but the code doesn't convey that meaning. Second, and arguably worse, when we insert another instruction before our move, all addresses move around:

```
// with addresses filled in:
mv 0, 0          // address: 0
mv 2, 444        // address: 2
raw_const 42     // address: 4
```

Our constant has moved by two addresses, but the code still contains a reference to the old location and needs to be updated. In a small example like this, that could still be done by hand, but especially with larger programs the effort to update all references, and the risk of missing one grows massively. It would be nice to automate this.

To accomplish this, you can label assembler instructions:

```
// with addresses filled in:
mv 2, 444        // address: 0
data_to_move:
raw_const 42     // address: 2
```

This gives the instruction `raw_const 42` a name you can reference, in this case `data_to_move`.

In a second step, we can reference that label in the move instruction:

```
// with addresses filled in:
mv &data_to_move, 444       // address: 0
data_to_move:
raw_const 42                // address: 2
```

What this does is effectively tell the assembler to search for the label `data_to_move` and take the address of the associated instruction. It still generates the same executable as when writing the address constants directly, but is more readable and doesn't break when the addresses move around. The syntax of `&label_name` is called taking a pointer, since it takes the address pointed to by `label_name`. Oftentimes, this is shortened to calling `&label_name` a pointer.

Besides moving constant values into function units, pointers and labels are also useful for control flow jumps. Jumps are implemented by moving the new memory address of where the code is located to the `IP` or `CIP` function unit register. In much the same way, we label the instruction we want to jump to. In the next step, we can move a pointer to a constant containing the address of the label into `IP` or `CIP`. Please note that we don't use `&label` and instead `@label`. This is because jumps require their destination address to be one below where the code is located.

```
loop:
    mv &loop_addr, IP
loop_addr:
    raw_const @loop
```

### 5.1.4 Numbers

Instructions like `mv` or `raw_const` take in numbers as argument. We have already encountered constant numbers like just `64`, but also more complex expressions for numbers like pointers. Besides these two there are a few more kinds of numbers.

A variant of normal numbers are binary numbers. A binary number can be input by writing `b1000100010001000`. This is equivalent to the decimal number 34952, but depending on the situation it's easier to read.

Another category of numbers are predefined constants. When interacting with function units, one could write their address directly, but that decreases readability. Therefore, all function units can be referred to by their name. In the case of the instruction pointer, the constant name is `IP`, or in the case of the ALU it is `ALU_A` for the a-operand register.

The last kind of number are `const` numbers. In the examples above, we often used the following pattern to load constants into function units:

```
mv &data, ALU_A
data: raw_const 6
```

This pattern is very common, but also a bit cumbersome to write. Therefore, there is a shorthand for it. Writing `mv const 6, ALU_A` is equivalent to the example above with the difference that the assembler takes care of generating a label and raw constant automatically. Additionally, it can employ a few optimizations like removing duplicate constants with the same value.

### 5.1.5 Other Instructions

When reading code written for the computer, there are a few more instructions of lesser importance to discover. The most common are described here.

A program must sit somewhere in the memory of the computer. This location is specified using the `start_section` instruction. It takes the memory address where the code starts as an argument and is typically found somewhere at the top of the file.

Defining exports (`export`) is a way to tell various information to the assembler. One example is where in the program section the execution should start, as it doesn't need to be the first instruction.

# 6 Construction

Besides designing an architecture and writing the tools to program the computer, a large part of this project is actually building and later testing the computer hardware.

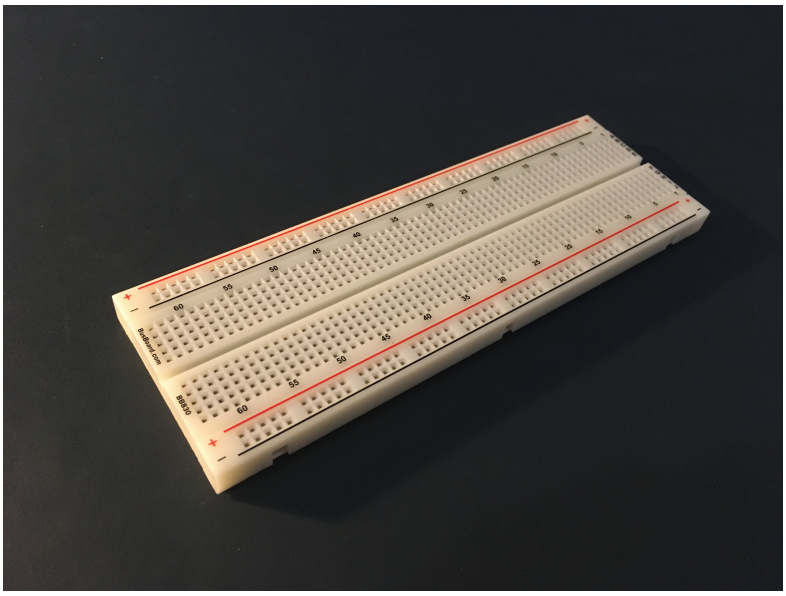

Figure 9: A breadboard without any components connected.[13]

## 6.1 Breadboards

I decided to build the computer on breadboards. They are an electronic prototyping method allowing for rapid prototyping and iteration. While breadboard circuits are not known for being particularly sturdy, I found this to be no issue. Even transports to school and back didn't pose any problems besides transporting a large object while ensuring it was never held upside down. To hold the breadboards together, I mounted them on a piece of wood.

## 6.2 Wire Management

From the beginning, I decided to use color coding to mark what a wire is doing. This helps me while building and helps with presenting the computer to others, as it is easier to follow explanations. In retrospect, I think I could have used more colors to achieve a finer granularity of functions to be distinguished. The problem is that this has to be done from the start to end up with a consistent coloring system. Another issue was that I repeatedly underestimated the amount of wire required to complete the task. Ordering resupplies takes time, during which I could not make progress. This means that some sections are colored differently than what their purpose is.

The following table shows what different colors means:

[13] Fig. 9: Breadboard

| Color | Role | Explanation |
|---|---|---|
| yellow | control logic | These wires control when the computer is doing what, like for example activating outputs of certain registers |
| green | ALU-internal wire | Green wires carry around temporary results inside the ALU. |
| red | +5V | Red wires carry +5 Volts to supply the computer with power. Additionally, they are equivalent to a HIGH logic level. |
| black | ground | Black wires carry the electric ground and are equivalent to a LOW logic level |
| blue | data-bus | Blue wires transport data on the data-bus across the computer. |
| white | address-bus/clock generation | Just like the data-bus, the address-bus using white wires spans across the whole computer. Additionally, white wires are used in the clock generation module. |

### 6.3 Electronic Design

While building the computer, I've also been drawing a model of the circuit in a online Circuit-CAD program. The drawings have helped me while debugging or changing circuits I haven't touched in a few months. They are also incomplete. In case they prove to be useful, I have included them alongside the report. One page represents one breadboard.

## 7 Results

One of the goals has been to build the computer physically. This has been done and figures 10 and 11 show a few images of the product.

The computer actually works. All function units were tested and are working. Even if this seems unimpressive and abstract, it means that there is the potential for applications when more time is spent developing software for the computer.

One example of what the computer is capable of calculating is the Fibonacci series. In this series, we start with the values `1,1`. The next element in the series is always the addition of the two previous elements: `1 1 2 3 5 8 13`, and so on. The output of a program producing this series can be seen in figure 12. Since only the lower eight bits can be sent to the computer displaying the results, after 233 the values wrap around.

When planning the electronic design, I aimed to run the computer at a frequency of 4 MHz. All programs besides the loader, which must interact with

[14] Fig. 10: The computer from the top.
[15] Fig. 11: The computer from the side

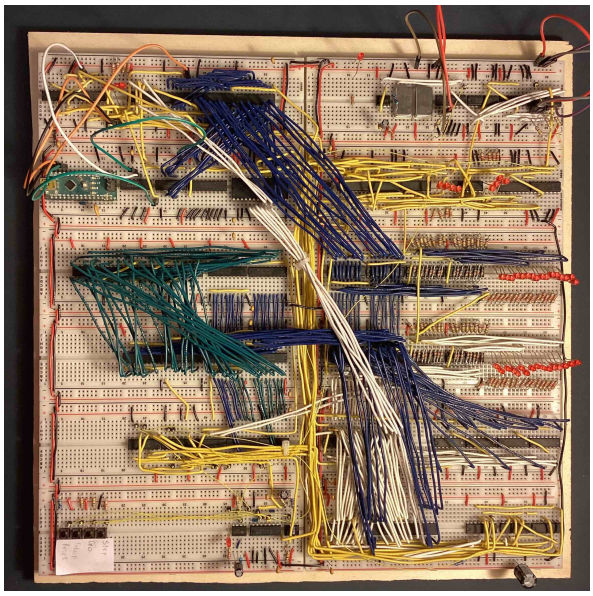

Figure 10: The computer from the top.[14]

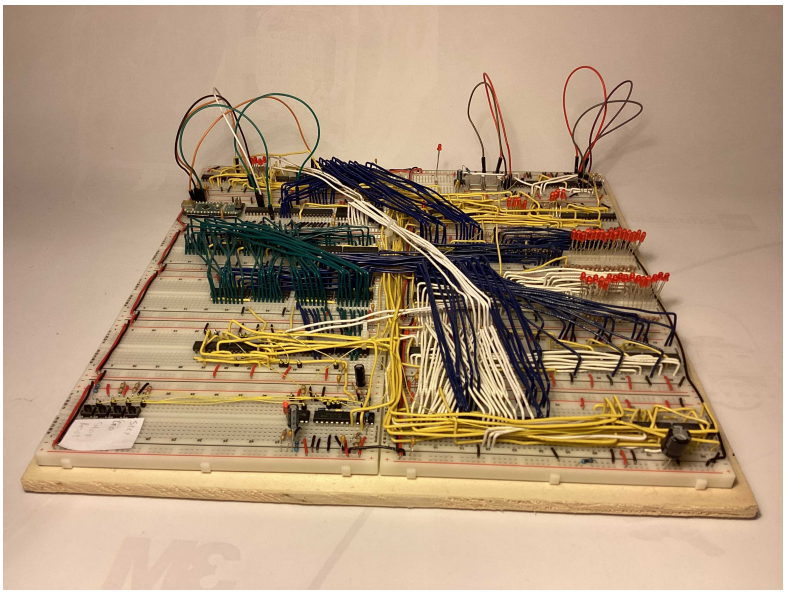

Figure 11: The computer from the side.[15]

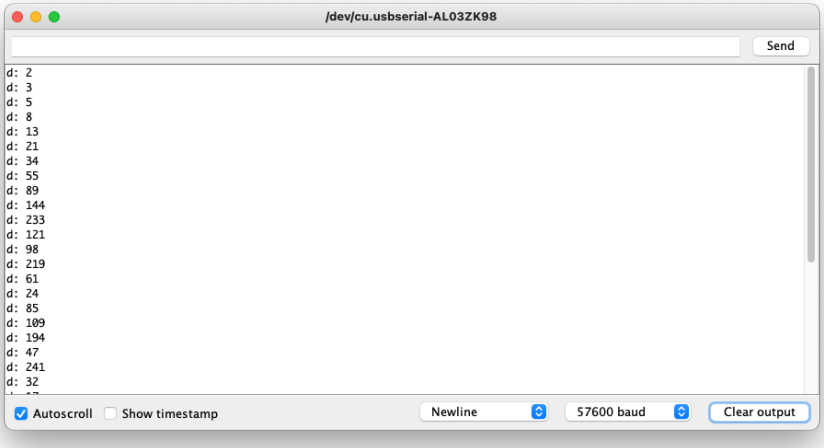


Figure 12: Output of the computer calculating the Fibonacci series. Only the lower eight bits of a word can be output, therefore the output wraps after 255.[17]

the Arduino, run fine at 4 MHz. I do plan to continue testing with a 16 MHz clock at some point.

Besides that, my focus has mostly been on the hardware side and the software to fully take advantage of the hardware has been lacking a bit.

# 8 Review

## 8.1 Construction

Besides taking significantly longer to wire than I expected, the breadboards used to build the computer didn't bring any surprises and were pleasant to work with. However, in September I learned about wire wrapping[18], a technique for fast and reliable prototypes that could also have been used in this project. At that point, it was too late to fully integrate it into the project though.

## 8.2 Architecture

While the architecture chosen for this project allows great flexibility with function units, once the core connecting the function unit stands, it is next to impossible to change it. The first weaknesses of this architecture started to appear around the summer holidays. All issues presented here aren't severe enough to

[17] Fig. 12: Fibonacci-Series output.
[18] Smith: Wire Wrap, 4.9.2023

render the architecture infeasible, but they limit the potential of the computer. Due to the fixed time available to complete this project, I wasn't able to redesign the computer and had to continue with what I had.

### 8.2.1 No Position Independent Code

All jumps and pointer look-ups require an absolute address. This means the program must be at a fixed position in memory. Even moving it by one word causes the program to fail. While I thought about this issue when designing the architecture, I underestimated the implications for programming.

Position-independent code means that it doesn't matter where in the memory your code resides. This is achieved by reworking how jumps work: Instead of providing a new value for the instruction pointer, they contain an offset that is added to the instruction pointer when performing a jump. The same is done to pointer look-ups: They are an offset to the instruction pointer instead of an absolute memory address.

That being said, it is still possible to program without having to worry about the lack of position-independent code for the most part, but the program loader, one of the most comprehensive programs I wrote, could benefit from it.

### 8.2.2 16-Bit instead of 8-Bit

When designing the architecture, I had to decide between using a 16 or 8-bit word size. I decided to go with 16 bits because it is the same width as the memory addresses, so that addresses in e.g. pointers can be treated the same as a value. Additionally, I wanted the computer to be somewhat capable, it should be able to compute something useful. This is easier with 16 bits.

The second argument turned out to backfire quite hard. 16-bit means twice as many bits as 8 bits, but that also implies twice as many wires. I did not expect that laying wires would take as much time as it did in the end, so going down the 16-bit route probably made the computer less powerful than a well-designed 8-bit machine, as having to spend less time on wiring would have allowed me to implement more features. In the end, I implemented fewer features than I expected and just barely reached the specified minimum.

If I could start over or give advice to anyone else attempting such a project, this argument weighs more than any other argument.

## 8.3 Time Management

It was clear to me that this is an ambitious project from the beginning. This proved to be correct, and this mostly in the time aspect. While I did manage to construct a working system, it is fairly limited and sometimes difficult to work with. What I mean by this is that a lot of features for efficient computation are missing, since I didn't have time to include them. For example, the computer cannot do comparisons of two numbers (`a < b`) natively. Instead, it has to compute the set of all values below `b` and check if `a` is in them. While it

works, it is slow compared to if the computer could directly compare numbers. More time would have allowed me to add the functionality required to perform comparison natively.

Another goal I didn't reach in time was to add a user-friendly control panel to operate the computer.

# 9 Conclusion

In this project, I was able to successfully design, build, and test a working computer based on a TTA-Architecture and demonstrate simple programs running on the computer. While the computer has all essential features to consider it a computer, due to a lack of time in the end, it hasn't unfolded it's full potential. This is also where I see the most room for improvements: Adding more features in the form of function units. Despite these shortcomings, the main goal of building a computer was achieved, so I consider the project a success.

# 10 Sources

## 10.1 Literature

### 10.1.1 With Author


- Douglas W. Jones, The Ultimate RISC, `https://homepage.cs.uiowa.edu/~dwjones/arch/risc/`, downloaded: 27.9.2023
- Hennessey, John L./Patterson, David A.: Computer Architecture, A Quantitative Approach, Waltham: Elsevier, 2012 (Fifth Edition)
- Smith, Paul: Working with Wire, How to Use a Wire Wrap Tool, `https://learn.sparkfun.com/tutorials/working-with-wire/how-to-use-a-wire-wrap-tool`, downloaded 4.9.2023
- Waterman, Andrew/Asanovìc, Krste: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, RISC-V Foundation, 2017


### 10.1.2 Without Author


- ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, `https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/About-the-instruction-sets`, downloaded: 24.9.2023
- efabless.com, auf: efabless.com, `https://efabless.com/`. downloaded: 11.4.2023
- Intel 4004: in: Wikipedia, `https://en.wikipedia.org/wiki/Intel_4004`, downloaded: 9.6.2023

- Texas Instruments: An Overview of Bus-Hold Circuit and the Applications (Rev. B), `https://www.ti.com/lit/an/scla015b/scla015b.pdf`, downloaded: 24.4.2023

## 10.2 List of Figures

- Fig. 1: Multiplexer-Tree for Signal Routing
- Fig. 2: Tristate-Bus for Signal Routing
- Fig. 3: Bus-Hold Circuit
- Fig. 4: Micro-Architecture-Core.
- Fig. 5: Data Bus with Function Units
- Fig. 6: Data Paths visualized
- Fig. 7: Complete Micro-Architecture with Data Paths
- Fig. 8: All major parts of the computer and their function labeled.
- Fig. 9: Breadboard
- Fig. 10: The computer from the top
- Fig. 11: The computer from the side
- Fig. 12: Fibonacci-Series Output form Computer in a Console