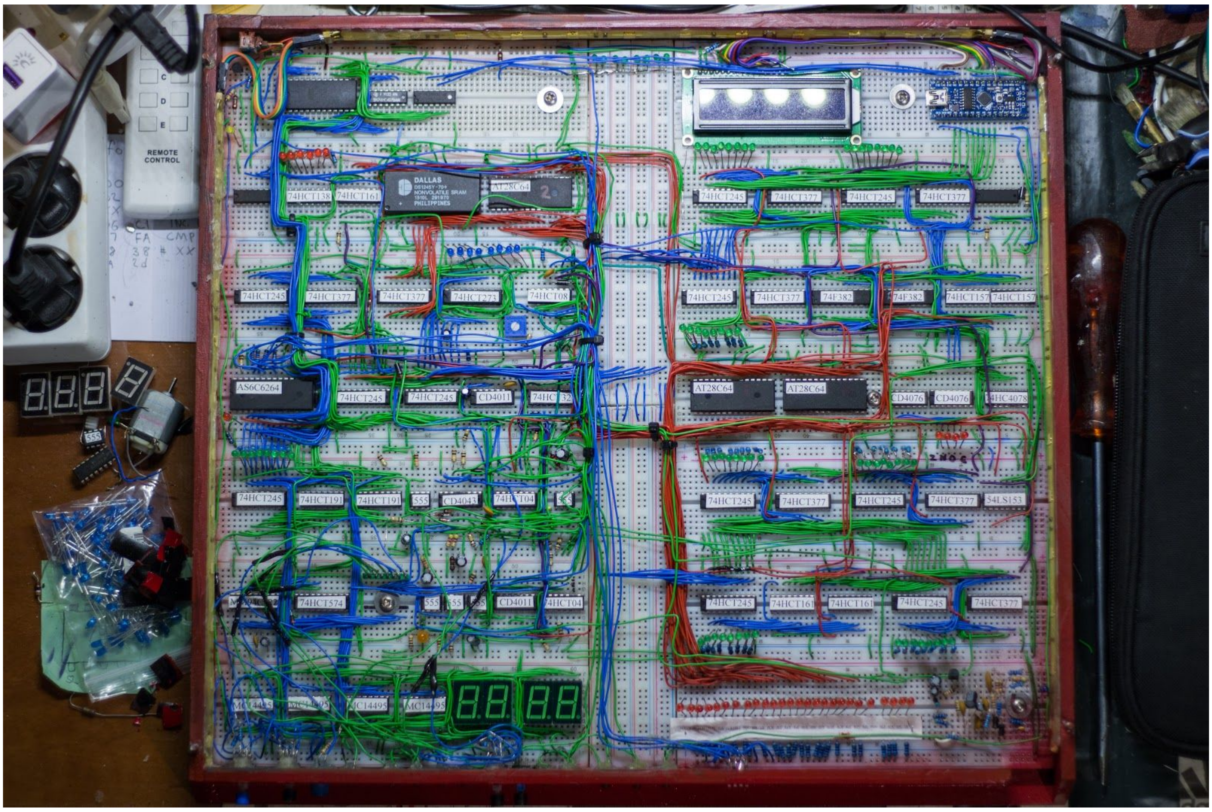
8-bit computer

Design and construction
2020/02/20



Emil Andréasson or Xrayer
Sweden

xehono@gmail.com
Youtube: Xrayer

20.2.2020

Foreword

Before I begin I want to tell you that I can't guarantee you that all the schematics of this computer is 100% correct, as im a human. If you encounter problems along the way according to incorrectness of the schematics, be sure to report it to me (xehono@gmail.com). I'm not responsible for any damage that may encounter along your build (IC:s, arduino, etc). You are responsible for what you build yourself. My biggest advise if you are going to build your own computer by the help of my schematics and documentation, make sure to test every module individually and se that they are working as they should before you set them all together, and of course, happy building!
Special thanks to James Bates who has given me a lot of inspiration of this computer, not to mention Ben Eater.

Abstract

The main purpose of this project is to design and build an 8-bit computer from scratch; this by building every module of the computer with individual ic:s, leds, resistors, capacitors, etc. The goal is to document as much as possible to make it easy for you to build along this documentation, and understand the basic concept of how a “simple” computer works. By the word “simple” I mean a computer that only has components that is necessary for the computer to work properly. This is a quite modified version of Ben Eaters famous 8-bit computer, with help of James Bates, that is on the next level. By this i mean integration of stack pointer, keypad, Arduino interface, etc.

Vocabulary

ALU (**A**rithmetic **L**ogic **U**nit) a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers

RAM (**R**andom **A**ccess **M**emory) a form of computer memory that can be read and changed in any order, typically used to store working data and machine code

CPU (**C**entral **P**rocessing **U**nit) an electronic circuitry within a computer that executes instructions that make up a computer program

Clock a unit that synchronise most modules within the CPU

Flag a bit that can be either true or false, the ALU have flags to indicate different states, like carry, etc

Logical 1 a state that can be described as “active” or “on” or “5 volts”

Logical 0 a state that can be described as “inactive” or “off” or “0 volts”

Logic gate an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs and produces a single binary output

Pin an IC have pins to connect to the breadboard

Register a quickly accessible location available to a computer's central processing unit (CPU).

Transistor can be used as an on-off switch for voltages; are the
elementary building blocks of logic gates

Tri-State allows an output port to assume a high impedance state, effectively removing the output from the circuit. This allows multiple circuits to share the same output line or lines

Control unit a component of a computer's central processing unit (CPU) that directs the operation of the processor.

truth tables a mathematical table used in logic; specifically in connection with Boolean algebra, boolean functions

IC (Integrated Circuit) a set of electronic circuits on one small flat piece (or "chip") of semiconductor material that is normally silicon

Computer Program a collection of instructions that performs a specific task when executed by a computer

Bus a communication system that transfers data between components inside a computer

Instruction cycle is the cycle which the central processing unit (CPU) follows from boot-up until the computer has shut down in order to process instructions. It is composed of three main stages: the **fetch stage**, the **decode stage**, and the **execute stage**. see (**3.1.9.2 Fetch and Execution cycle)**

Table of Contents

1. INTRODUCTION

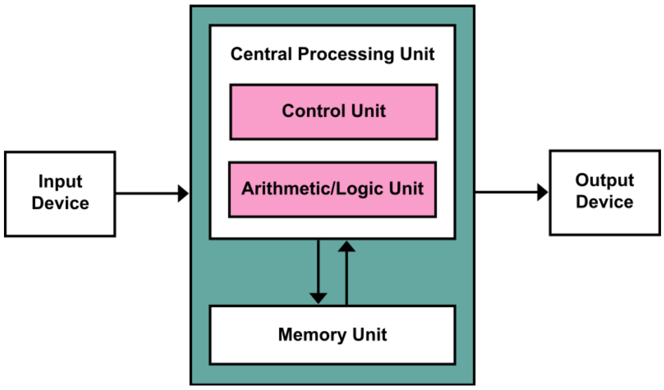
1.1 Background

The project covers the most fundamental parts of a computer and how it work in its most "simple" form. The goal of this project is to educate people who want to build their own computer from scratch.
This computer was mainly built after *Von-Neumann architecture*. The term "Von Neumann's architecture" refers to a design scheme of electronic computers invented by the mathematician *John Von Neumann* in 1951.
This project was inspired mostly by James Bates who have a youtube channel where he describes every module in his computer in detail. But the real credit should go to Ben Eater as he was the first on youtube that teaches you how to build an 8-bit computer.
I thought I would take one step further and document every single step for this projekt to make it easy to understand the core mechanics of computers. I have built three computers in total by now (old 8-bit computer, 16-bit computer and the new 8-bit computer). I thought it would be helpful to do a detailed documentation for people who want to build their own computer with instructions to follow along with.

The architecture design John Von Neumann invented, consisted of **three** parts

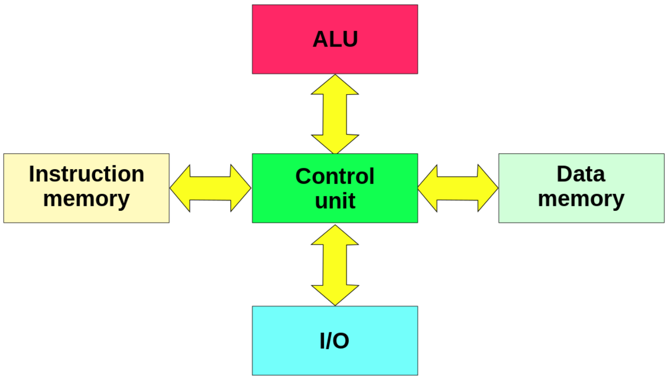
- CPU
  The part that do the calculation based on the machine code programmed in RAM.
  This part includes ALU, Control unit, Registers
- RAM (Memory)
  Computer memory where instructions is stored in form of machine code.
- Bus
  A physical connection made of wire between CPU, RAM and other modules in the computer. It enables the exchange of data.

There is also an architecture beside Von-Neumann architecture, called *Harvard Architecture.*
The main difference between Von-Neumann architecture and Harvard architecture is how they handle program memory (machine instructions) and data memory. The Harvard architecture stores machine instructions and data in separate memory units that are connected by different buses. This allows Harvard architecture to run a program and access data independently, and therefore simultaneously.
Harvard architecture is more complicated but separate pipelines remove the bottleneck that Von Neumann creates.



Von-Neumann architecture

This computer is a “modified Von-Neumann architecture”, that draws inspiration from Harvard architecture. There are no physical separation between instruction memory and data memory, it's all in the same RAM IC. In a “real” Harvard architecture there are two memory units separated from each other. In my case, they are instead separated by controlling what address space within the RAM is currently active; by this controlling the last address pin of the RAM, in my case address 12 (A12). I'm also just using one pathway (or bus) for both data and instructions.
The computer has a feature that let the programmer to be able to choose whether you want to access data memory or program memory by a flick of a switch (S4) when in program mode, see (**3.1.10 Input module**).



Harvard architecture

All modules in this computer:

- Clock
- A-Register
- B-Register
- C-Register
- D-Register
- Stack pointer
- Program Counter
- RAM
- MAR (Memory Address Register)
- ALU (Arithmetic Logic Unit)
- Control unit (Ring counter, Instruction register, EEPROM:s)
- Input module (including Arduino bootloader)
- Output LCD

A computer is a machine that is designed to do more than only one specific task such as a steam-engine, whose job is to drive an axle, or a lawn mower whose job is to cut grass. You can program a computer to do tasks that a programmer have programmed. This is why the computer is special against all other machines.
A computer need a program to work. A program in itself consist of a bunch of instructions, and an instruction in itself consist of both fetch and execution cycle, see (**3.1.9.5 Fetch and Execution cycle**).

1.2 Object

The goal of this project is to get an understanding of the core mechanics of a simple computer while holding it SAP (simple as possible) - sort of.

1.3 Framing of questions

How do you design the hardware of an 8-bit computer?
What functionality should it implement?
How does the computer work?

2. METHOD AND MATERIAL

2.1 Method

My work procedure intended to first of all gather information necessary to build the computer. It included what functionality to implement but also what architecture I would use, meaning what layout of every module should be placed in the computer such as registers, RAM, Control unit and so on. Much of the design is inspired by James Bates, who got his inspiration from Ben Eater. I decided to implement my own modifications such as the input module etc.

My method to avoid confusion about potential troubleshooting, was to test every module side by side. When one module was finished, tests on it was applied to see if it worked as it should. When every module worked as expected, I integrated them. This ensured that I had to do as little troubleshooting as possible. I recommend to have a plan well thought out to minimize troubleshooting but also to avoid having to redesign the computer, while you already built some parts.

As I mentioned the project include a lot of troubleshooting, witch was expected for such a big project.

To troubleshoot effectively it's good to work systematically, for example if an IC dont work as expected, use some kind of pin tester (probe) such as a simple LED with a resistor and a wire in series that have it’s cathode connected to minus, and its anode connected to the IC:s pin. Then test every pin on the IC to detect the issue. This is easier said than done though, believe me.

See (**3.1.11 Power distribution and tips**).

2.2 MATERIAL

Material that was used in this project are listed below.

IC:

x3 74HCT04
x2 74HCT08
x1 74HCT32
x1 74HCT138
x2 74HCT157
x3 74HCT161
x2 74HCT191
x11 74HCT245
x1 74HCT273
x8 74HCT377
x1 74HCT574
x2 74HC4078
x2 74F382
x1 54LS153
x2 CD4011
x1 CD4043
x2 CD4076
x1 MM94C922
x5 AT28C64
x1AS6C6264
x6 NE555
x4 MC14495
x1 74HC595

Other:

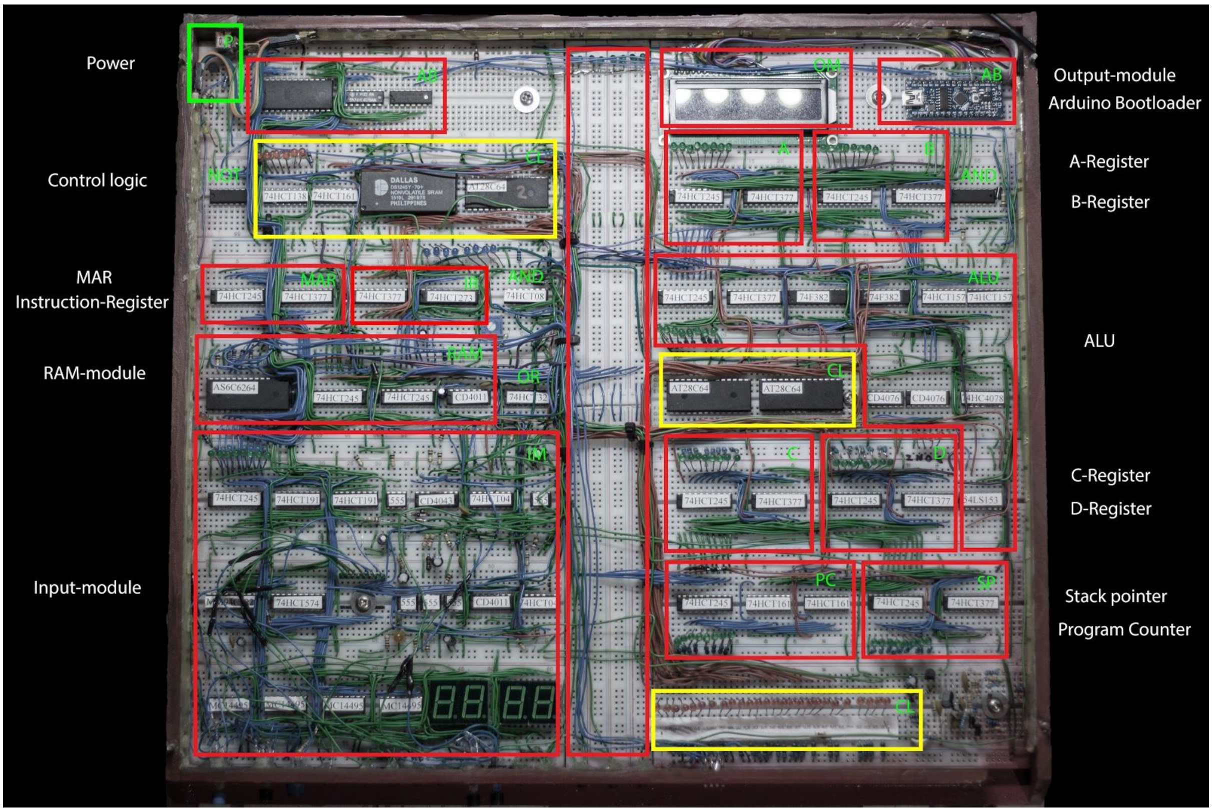
- 1x LCD 2x16 character
- Arduino nano (china clone)
- x14 Breadboards
- x8 Breadboard rails (you can get these when you start attach the breadboards together by removing one rail of every breadboard that connects together)
- A lot of single core wire (200 meters should be fine, pick different colors to color code: green, blue, red, white, etc)
- A lot of 3mm LED:s (different colors, green, red, blue and yellow)
- x3 3mm RG LED:s (red and green common cathode)
- PVC box for the computer (not necessary, but practical)
- Plexiglass front cover (with keypad built in)
- Power supply 5v (USB powered, 1A, 2A or 3A)
- Customized USB A 2.0 cable (male to male, solder on your own, or buy)
- USB A 2.0 connector female (integrated in chassi)
- PNP transistor (BC 640) (as power switch)
- 10 Pushbuttons
- 4 Switches
- 4 bit DIP switch
- 1 Potentiometer
- x4 7-segment displays
- Electrolytic condensators (different values, see schematics)
- Ceramic condensators (different values, see schematics)
- A lot of resistors (different values, see schematics)
- Diodes (see schematics)
- 4x4 keypad
- Male Pin Header Strip (attaching keypad and input module)

See list of references on the last page for links to different links and websites (very useful)

3. RESULT

3.1 How to design hardware?

The task of building a computer is pretty complex, it includes various components or modules, which I will address later. The most fundamental part of a computer is that it can save and manipulate data. If we dig ourself to the core mechanics of the computer we will find a circuit that can save one bit of data either a one or a zero. It's called a D-flipp flopp, and consist of an SR-latch and some other logic gates, see (**3.1.3.3 D-Flip-Flop**). If we put 8 of this circuits together then we have what we call an 8-bit register. A register IC is often 8-bits like the 74HCT377. This means that an 8-bit register can hold 8 bits of value, also known as one *byte* (8 bits).



Every module in the computer

3.1.1. Logic Gates

There are a total of 8 logical gates. These are the following, **AND, NAND, OR, NOR, XOR, XNOR, BUFFER, NOT** - gate. Logic gates can be combined in so called networks to achieve more complex boolean algebra. A logic gates usually takes in two inputs and generate an output depending on the function of the logic gate, but this is not every case. A logic gate can have theoretically, endless of inputs, but at least one input. The output is always only one output. In an IC like 74HCT08 or 74HCT32 there is a total of 4 logic gates in both of them. They are four two input And gates and four two input Or gates respectively. They are most common in that format, but there also exists “one eighth input gate” or “two four input gates” or even “three three input gates”.

3.1.1.1 AND/NAND Gate

First of, the **AND** gate. An **AND** gate is only true when both inputs are true, in other words, when input A **AND** input B is true. The **NAND** gate is the inverse of the **AND** **gate.** The only case where the output is false is when input A **AND** input B is true, inversing the logic.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

3.1.1.2 OR/NOR Gate

The **OR** gate is true when either input A **OR** input B is true. The **NOR** gate is the inverse of an **OR** gate, meaning that the output is false when input A or input B is true.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

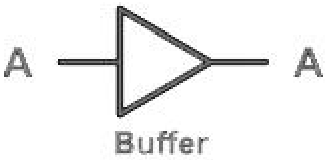
3.1.1.3 XOR/XNOR Gate

An **XOR** gate is a combination of an **OR** gate and an **NAND** gate. The output is true when input A or input B is true, but not both. **XNOR** is the inverse of **XOR**, meaning that the output is false when input a or input b is true, but not both.

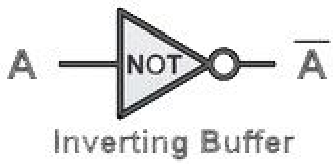
A	**B**	**Y**
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

A	**B**	**Y**
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

3.1.1.4 BUFFER/NOT Gate



A **NOT** gate inverse the input signal. A true on the input results as an false on the output. A false on the input results as an true on the output. One input, one output. The **buffer** gate keeps the input same as the output signal. The interesting thing about a buffer gate is that its mono directional, meaning that you only can send a signal from input to output, and not from output to input. A diode can work as a buffer gate.



A	**Y**
0	**0**
1	**1**

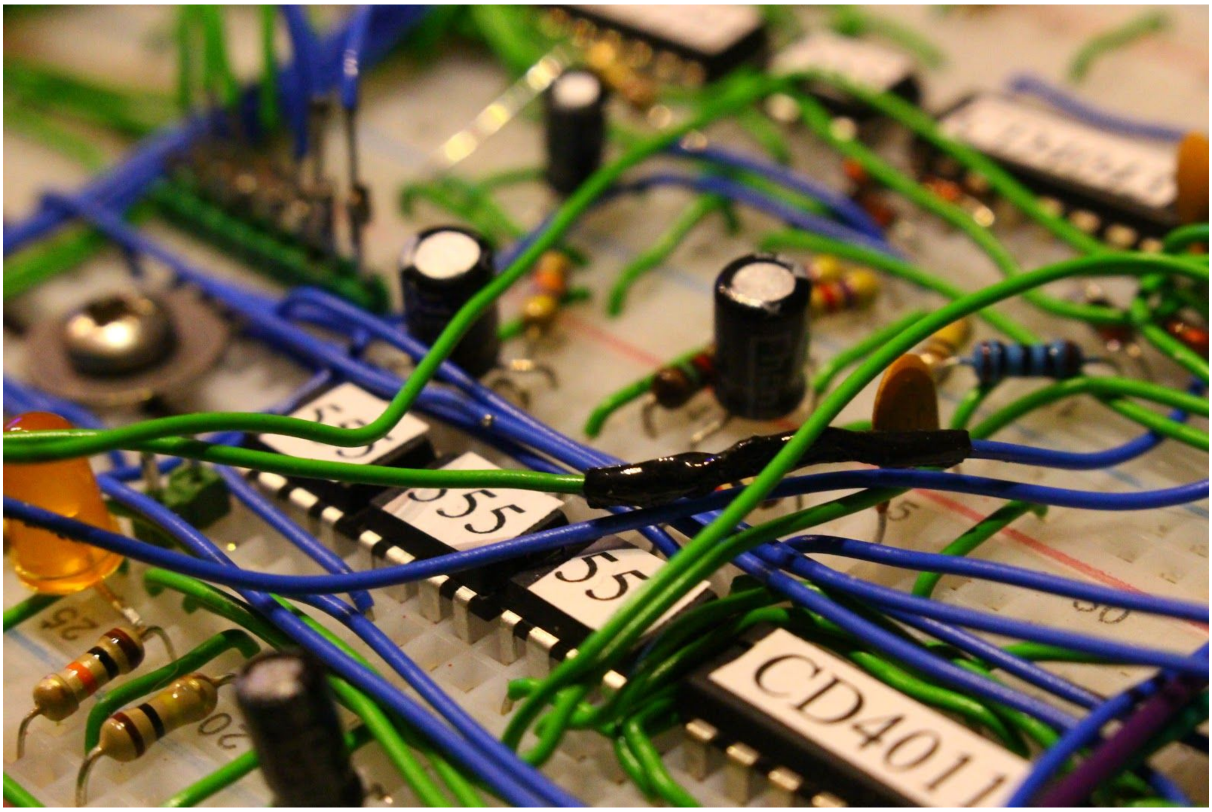
A	**Y**
0	**1**
1	**0**

3.1.2 Computer Clock

A computer clock may not be what you think it is. The clock is used for keeping the the computer in sync to work properly. Almost all modules in the computer uses the clock to stay in sync with the rest of the computer. This is to ensure that every module is enabled and write data in correct order. If we want to load A-register with a value, we can't move the value from RAM to the register if we haven't set the correct address in the MAR before the data is moved, this instruction can be as an example:

LDA F0

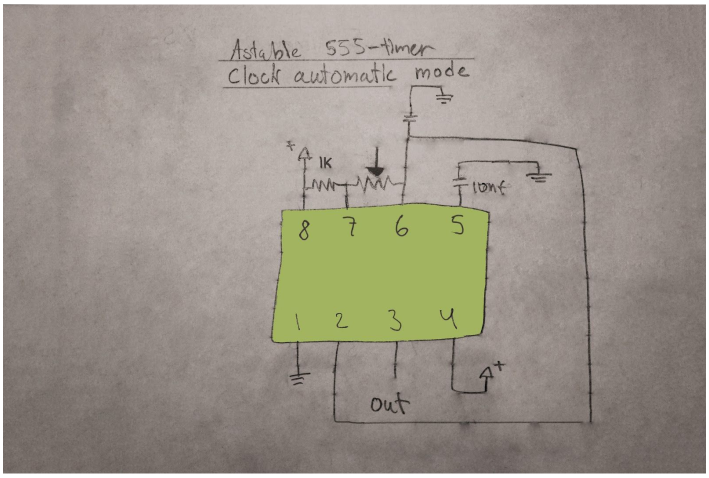
We load register A with the value in RAM at address F0. First we write the correct address at the MAR module, then the RAM can output it's value to the A-register



The clock module should be able to halt, resume and change its frequency. The frequency should be based on a potentiometer. A 555-timer can be wired in monostable, astable mode and bistable.

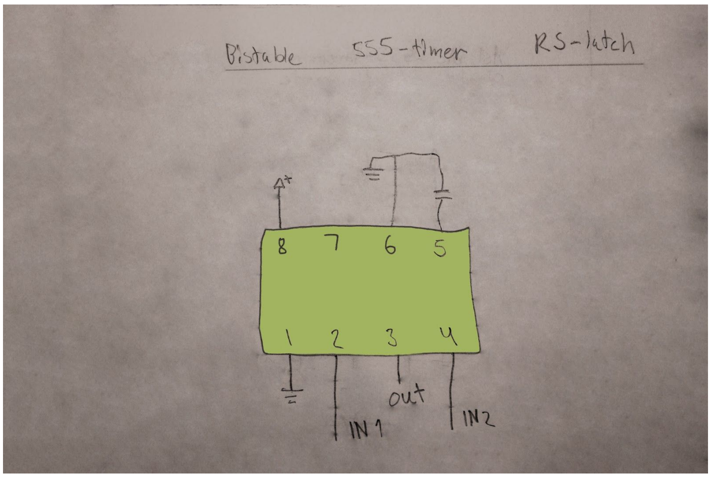
The monostable circuit is stable in one state. It's commonly used in debounce-circuits to eliminate the problem where a switch "bounce" more than once. This causes problem if you for example press the single step clock button once, but the clock pulses more than once. This I due to the bounce problem and can be easily fixed with this circuit.

3.1.2.1 Three different hookups for the 555-timer



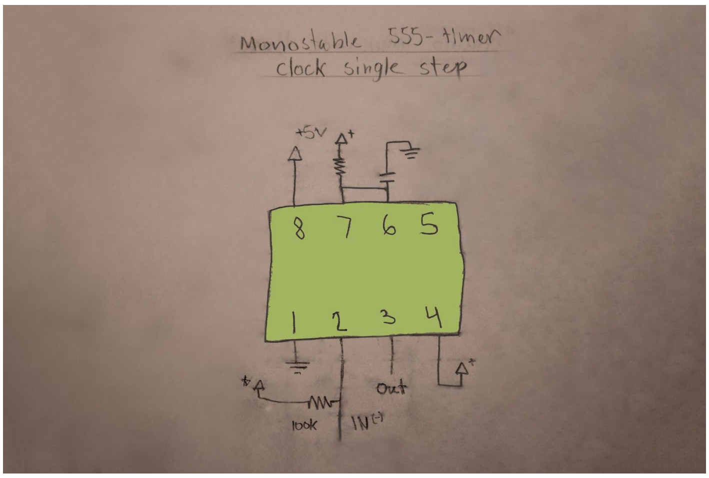
Astable-mode

This circuit allows a pulse to toggle on and of automatically, the speed is controlled by a potentiometer. This is an excellent choice for our clock.



Bistable-mode

This circuit is similar to an SR-latch circuit, though using a 555-timer. This is great because it's small size.

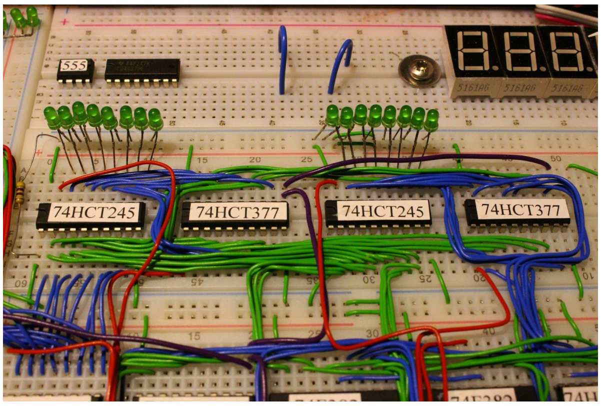


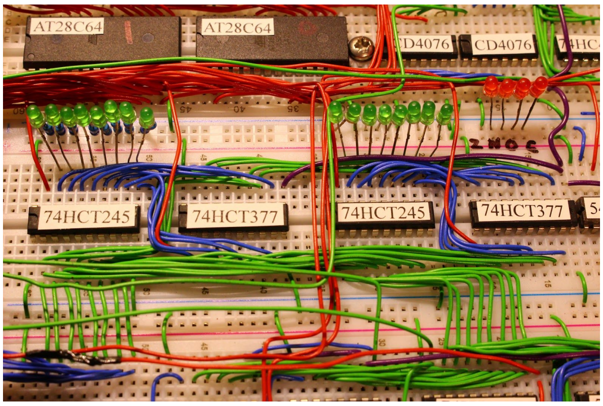
Monostable-mode

This circuit behaves like a delay circuit. When you send a pulse trough pin 2, you get an extended out pulse of pin 3, depending on the resistor and capacitor value on pin 6 and 7.

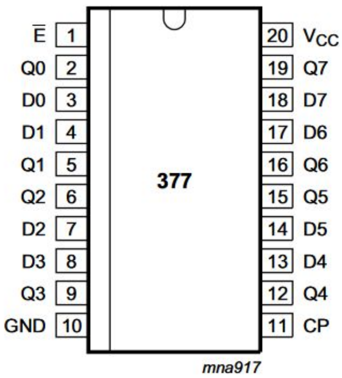
3.1.3 Register module (special purpose register and general purpose register)

There are different kinds of registers, registers that only the computer has access to and registers that the programer and computer have access to. They are called special purpose register and general purpose register respectively. The Instruction register, Status register, ALU-register, Program counter, Stack Pointer and MAR (memory address register) are special purpose registers, and the A, B, C, D registers are general purpose registers.

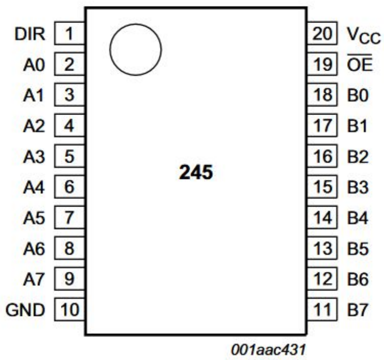




The general purpose register is made of two IC:s, one buffer (74HCT245) and one 8-bit d-type flip flop (74HCT377). Some special purpose registers differ from each other, therefore I go through each one in its own context, however the design of all the registers are almost the same, what differ is what they are used for.



74HCT377



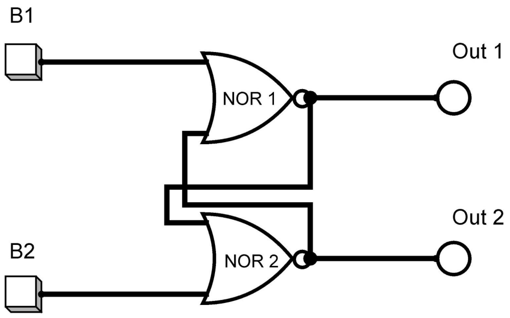
74HCT245

3.1.3.1 SR-latch

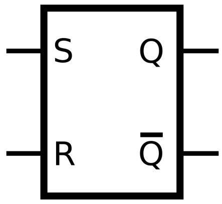
SR-Latch (set reset latch) is a circuit that feedback to it self. The principle of this circuit is to store a single “bit”, either as a one (1) or zero (0).

NOR truth table

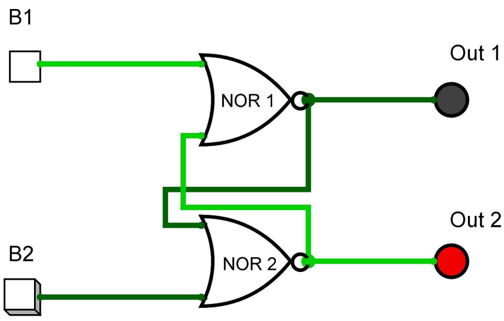
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

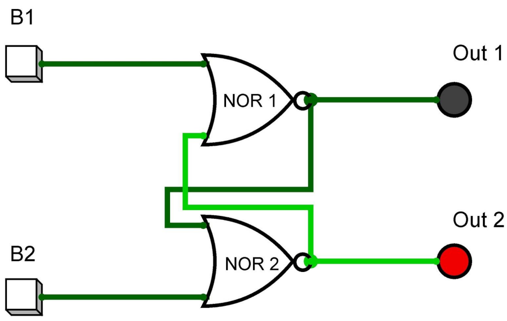


Logic symbol

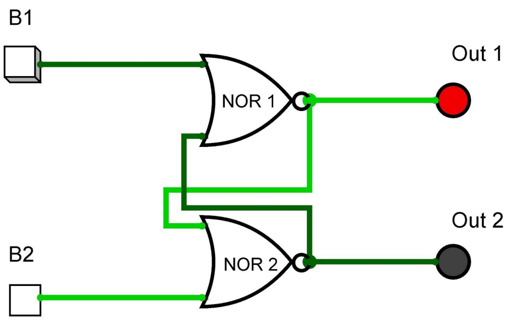


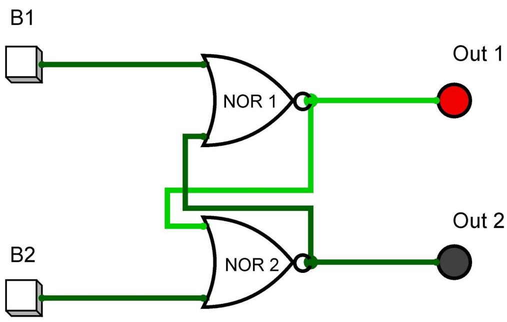
When B1 is pressed NOR gate 1 gets disabled. NOR gate 2 gets enabled because of the disabling of NOR gate 1, allowing output 2 to be true or 1, and output 1 to be false or 0.





When B2 gets pressed vice versa.





3.1.3.2 D-latch

The D-latch allows us to control when we want to store a bit, with an enable signal.

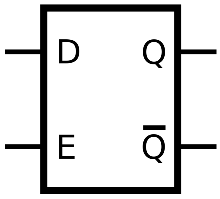
AND truth table

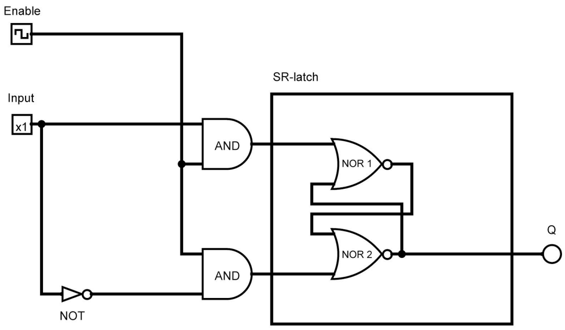
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

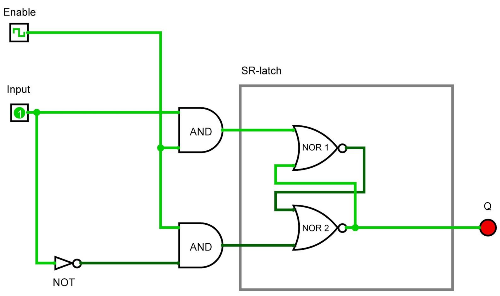
NOT truth table

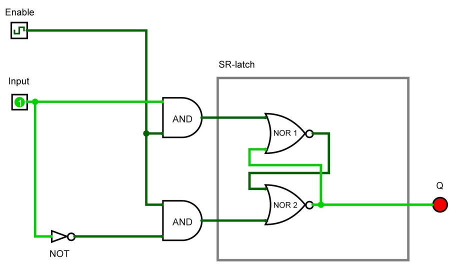
A	Y
0	1
1	0

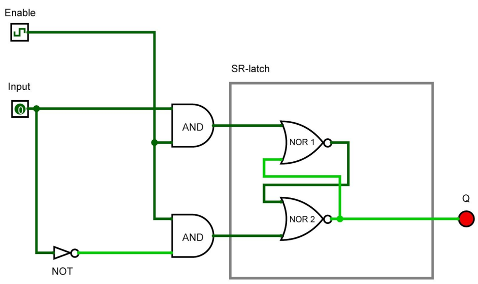
Logic symbol

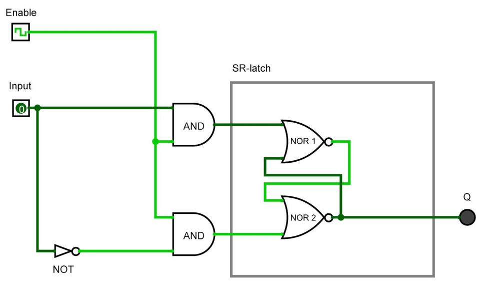






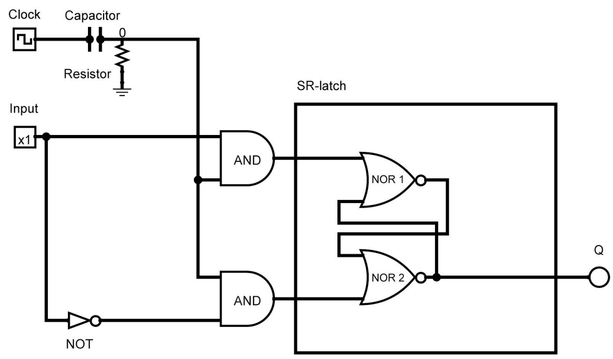




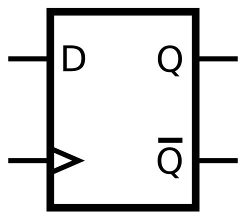


3.1.3.3 D-Flip-Flop

The D-Flip Flop is an identical copy of the D-latch with one exception, the enable signal is pulse triggered which means that enable signal only reads the data input on the rising edge of the clock. This is achieved by having a rc circuit; by this having a capacitor and resistor in series, to make a detection edge. The value of the resistor and the capacitor determines the time it takes for the capacitor to go from fully charged to it’s discharged.

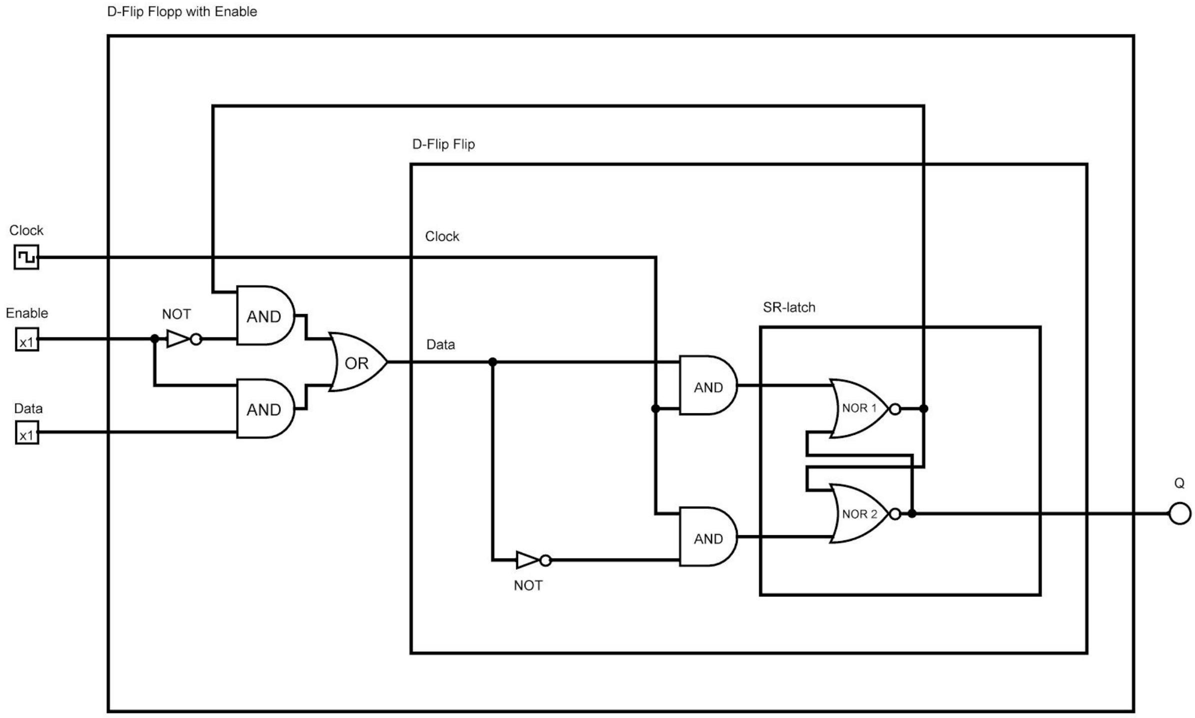


Logic symbol

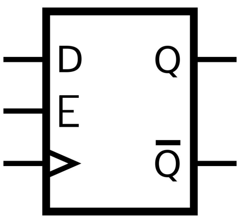


3.1.3.4 D-Flip-Flop with Enable

The great thing with Flip Flops are that they are stackable allowing us to store more than one bit at a time. Often you want to store a byte of data consisting of 8 bits. All you have to do is to connect all of the clock signals and the Enable signals together respectively. The data pins are left individually, making the pinout look like this: 8 data input pins (D0 - D7), 1 clock pin, 1 enable pin, 8 data output pins (Q0 - Q7). 8 bit registers can also have things like **Master Reset**, or **Output Enable**, depending on IC.

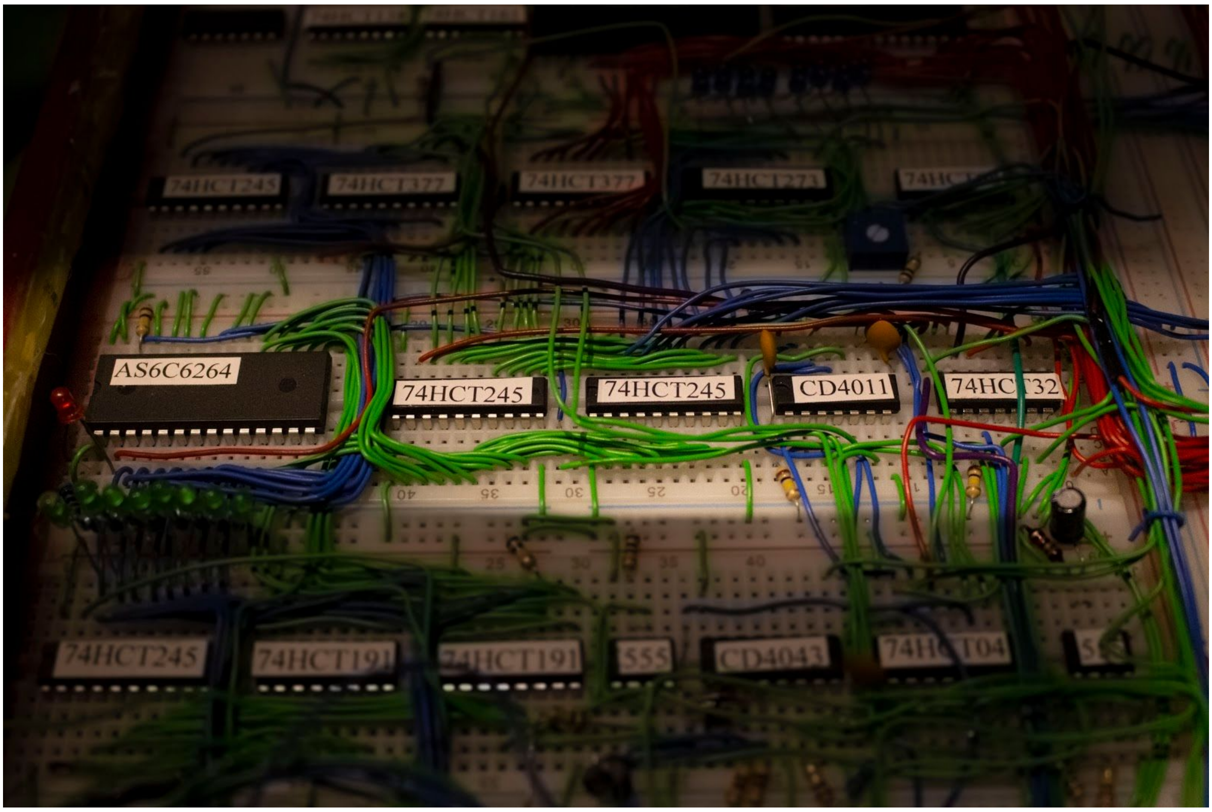


Logic symbol



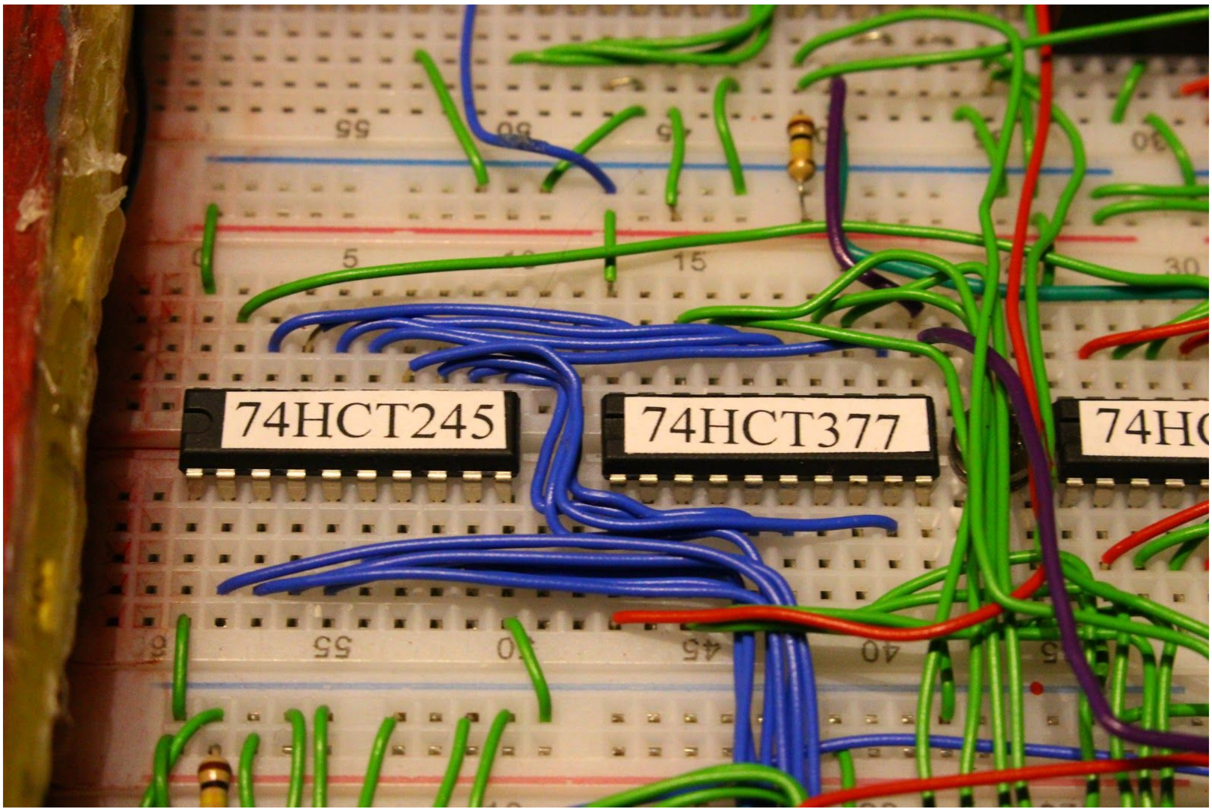
3.1.4 RAM (Random Access Memory)

The RAM module of the computer works like registers clumped together, it can hold much more data though. The difference is that it can address a location in the RAM dependent on what address you send to it. It's like you are choosing a register within it. A RAM can hold a lot of data unlike a register that can hold one byte. The RAM size is printed on the IC, but googling the name of the IC for a datasheet is recommended for more details. In this computer the AS6C6264 RAM chip is used. It's a 65,536-bit or 8KB static RAM. RAM is used to store both data and instructions. In my computer, I don't mix those, i'm using a "modified" *Harvard Architecture,* wich in simple terms means that data is separated from the instructions in the manner of avoiding the programmer to change the instructions during the run of a program. The red LED to the left on the RAM is indicating if the computer is accessing the upper or the lower part of the memory meaning the *data or instruction memory*.



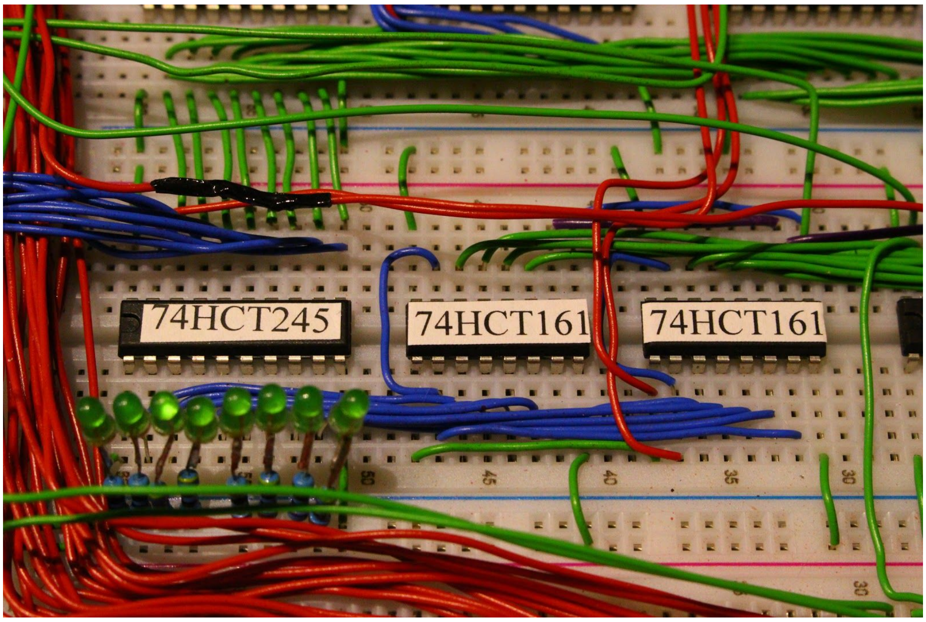
3.1.5 MAR (Memory Address Register)

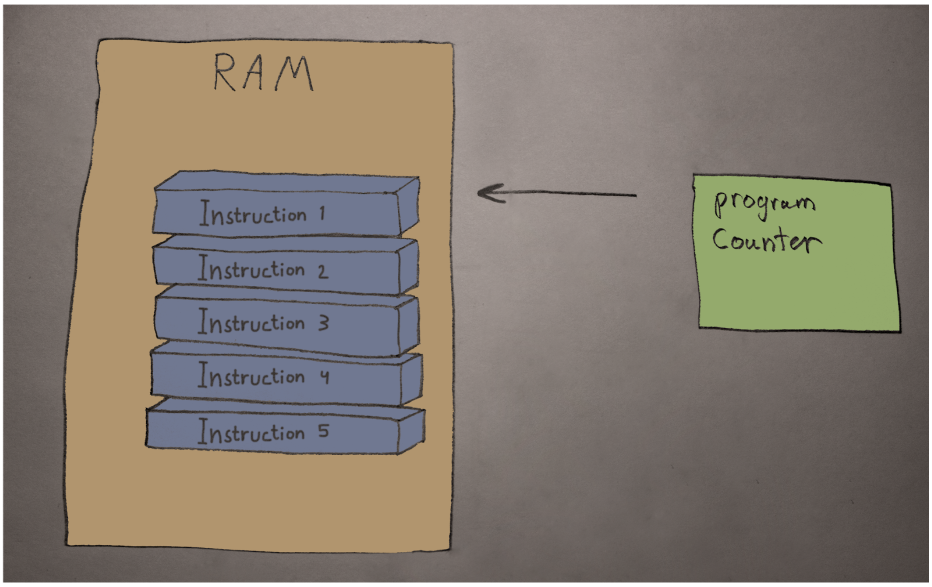
The MAR is a special purpose register however, the design is exactly like the A, B, C, D-registers, i.e. it consists of one buffer (74HCT245) and one 8-bit d-type flip flop (74HCT377). The purpose of the MAR is to point at a specific address in the RAM. MAR takes data from the buss and send it to the address pins of the RAM chip (AS6C6264). It sends and takes data only when it’s told to by the the control matrix. The MAR however unlike the general purpose register does not have the ability to put its data back a the bus again, because the 74HCT377 isn't connected to it.



3.1.6 Program Counter

The program counter is one type of register. The difference is that it can increment by one when it told to. It uses two IC 74HCT161, which is a 4 bit counter with load (useful for jump instruction). Because I have two 4-bit counters, the program counter is 8-bit wide. The program counters task is to hold track where in the program the computer currently are, in fancier words, hold track of what instruction the computer is currently executing at the moment. When an instruction is executed, it points to the next instruction that needs to be fetched and executed. See the program counter as a pointer that points to the next instruction, see picture below. Normally the program counter advance by one each *execution cycle,* see (**3.1.8.5 Fetch and Execution cycle**) but sometimes you want to skip to another address in the program counter to make a so called "jump" in the program, which is very useful for if-statements for example, see (**3.3.2 Example programs**).

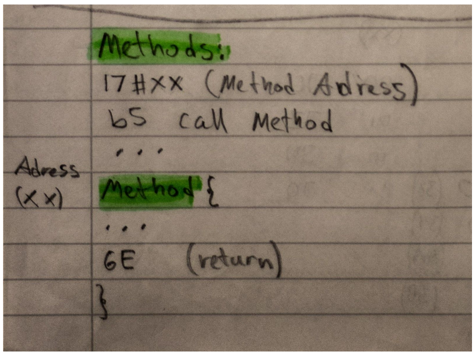


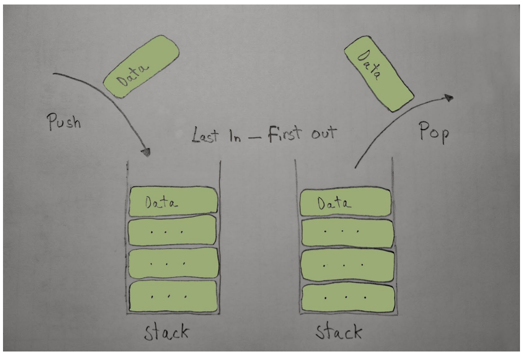


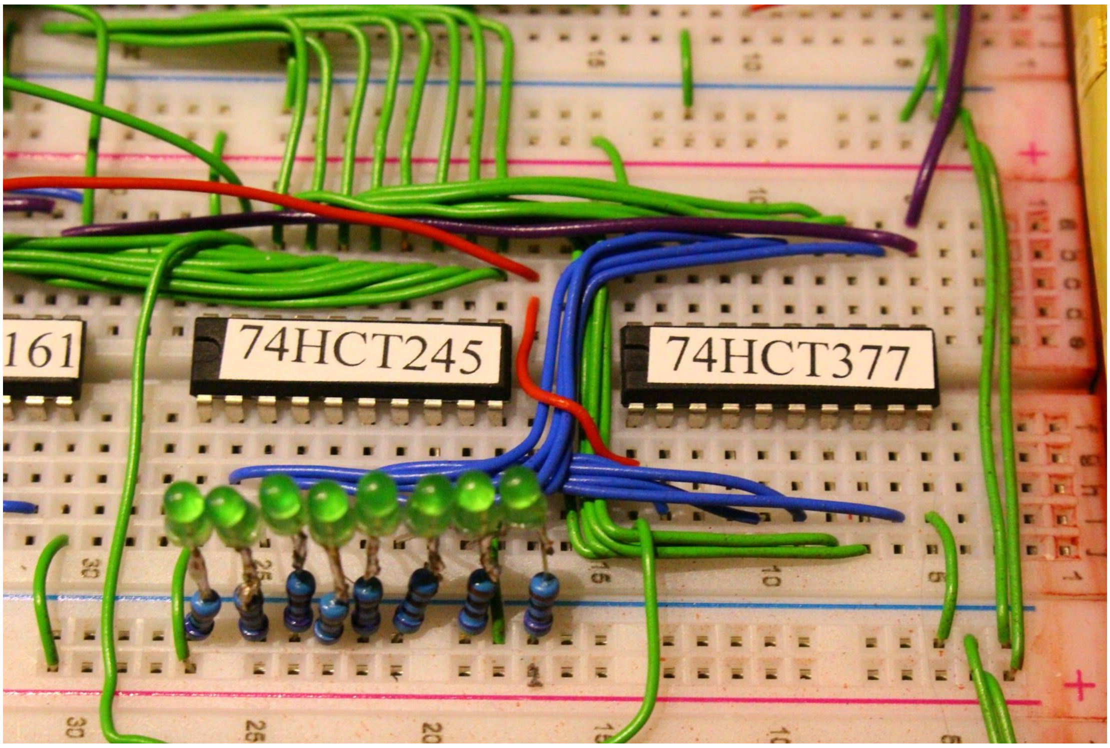
3.1.7 Stack Pointer

The Stack Pointer is a special purpose register, however, the design is exactly like the A, B, C, D-registers, i.e. it consists of one buffer (74HCT245) and one 8-bit d-type flip flop (74HCT377). A stack is a concept of storing data that is based around the term "LIFO" (Last In First Out), see illustration below. It reverses the order that data arrives and is sent. The stack pointer can be used to store the address of current instruction. This is useful for making so called "methods"
see (**3.3.2.7 Program - Methods**).

A little about methods:
17#xx mean that we load register C with the address xx where the method is in the RAM. b5 is used for calling the method and is a push instruction from program counter to stack pointer. We push data to the stack pointer to store the address of the program counter. The actual method is at address xx, there we can write some code, but we always need to write 6e to exit the method or to "return". 6e is a pop instruction from the stack pointer to the program counter we stored earlier.

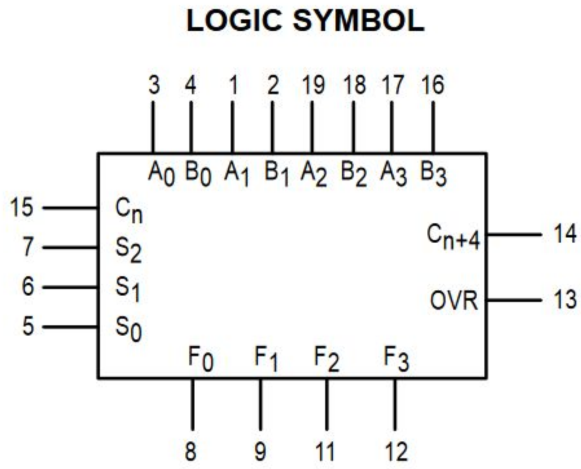




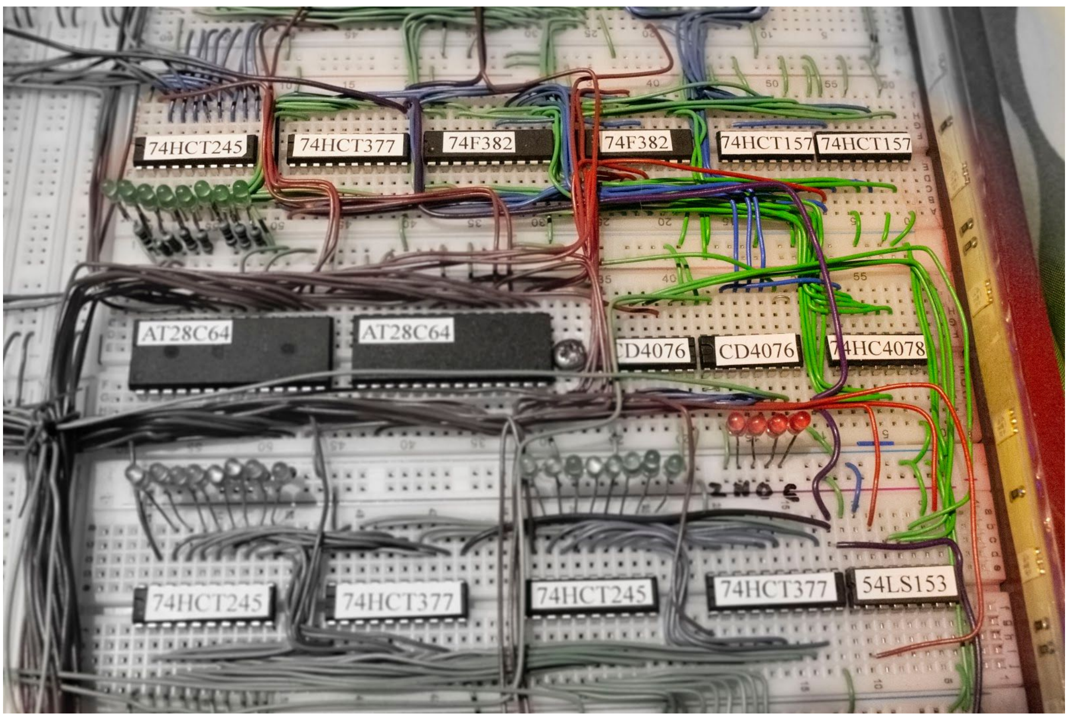


3.1.8 ALU (Arithmetic logic unit)

In this module, all the computer calculations take place here. I'm using two ALU 74F382 ic:s. The IC is completely reliant on combinational logic. It have no memory storage whatsoever. You could actually build an ALU by programing an EEPROM the correct way (EEPROM are also only combinational logic). This specific ALU IC have 6 different functions. Depending on what function is selected by the 3 address pins, the IC can perform addition and subtraction, and bitwise operation such as OR, AND, XOR. The ALU module does not only consist of two 74F382, but also logic that saves the outputs from the ALU chip. It needs to save both the calculated value, but also potential flags. The 4 bit d-register 4076 is used to store the flags, where as the 74HCT377 which is a 8-bit d-register is used to store the calculated value.



There are 4 flags in this computer, **Z**ero flag, **N**egative flag, **O**verflow Flag and **C**arry flag. The ALU:s task is to save these values while the control units task is to make actions based on those flags.



FUNCTION SELECT TABLE

Select			
S_0	S_1	S_2	Operation
L	L	L	Clear
H	L	L	B Minus A
L	H	L	A Minus B
H	H	L	A Plus B
L	L	H	A⊕B
H	L	H	A + B
L	H	H	AB
H	H	H	Preset

H = HIGH Voltage Level L = LOW Voltage Level

3.1.8.1 ALU-Register

The ALU outputs pins isn’t directly connected to the bus. Instead, the output data is feeded trough an ALU-register were the data can be stored temporary. Why it's important with this register, is that both registers that feeds the ALU operands are not tied directly to the ALU:s operands, only the B-register is fixed. The programmer can choose one of the registers freely what he want to operate with. The register that is not directly connected to the ALU will need to output it’s value to the bus.

Here is where the ALU-register takes place. We must save the ALU calculation because the *bus is already occupied* of the second register that feeds one of the operands of the ALU. After the value is stored in the ALU-register, the register can send it data to the disaried register.

Like in this in initialization [A = A + B]

(*A-register adds the value of B-register and store it in A-register*)

3.1.9 Control unit

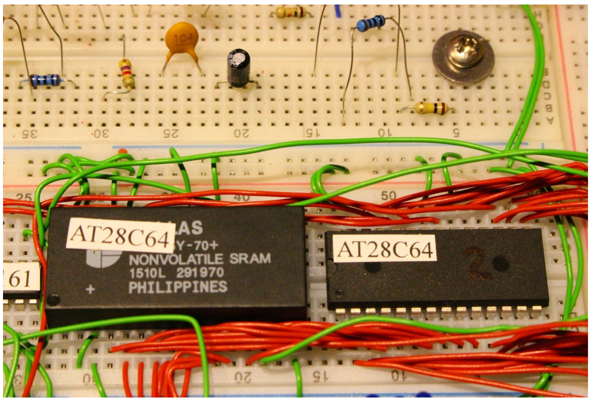
Without a control unit nothing would work. This module control the whole computer, (thereof the name) by this; all of the computers modules such as the registers, RAM etc. The control unit task is to enable or disable specific modules in a specific micro time depending on the current instruction and status register. The control unit consist of several modules. In the illustration at page 7, I have marked the Control unit by yellow. This includes following modules, ring counter see (**3.1.8.3 Ring counter**), instruction register (**3.1.8.2 Instruction register**), status register (**3.1.8.4 Status Register (Flags)**) and eeproms (**E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory).

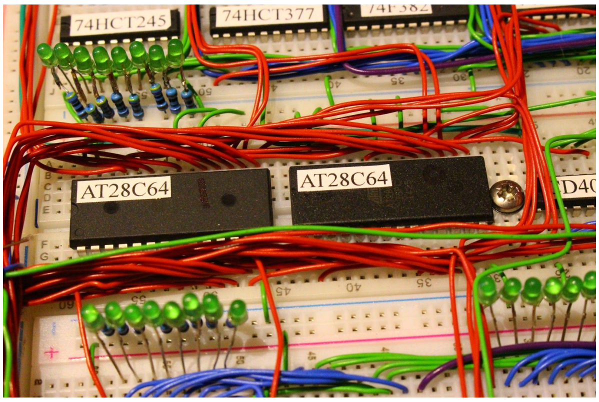
The main part of the control unit is the four EEPROMs (AT28C64). Their outputs controls the other computers parts, as I mentioned. The address (13-bit wide) of the EEPROMs are feeded by three components, the instruction register (8-bit, OPCODE), ring counter (3-bit microtime T) and flags + carry (2-bit).

```
EEPROM address:   bit position     12       11      10  ....  8     7   .....   0
                  meaning         flags    carry   (microtime) T                opcode
```

3.1.9.1 EEPROM:s

Their task is to store the control word for the computer. This control word is controlled by the instruction register ring counter, status register combined.





3.1.9.1.1 EEPROM Software

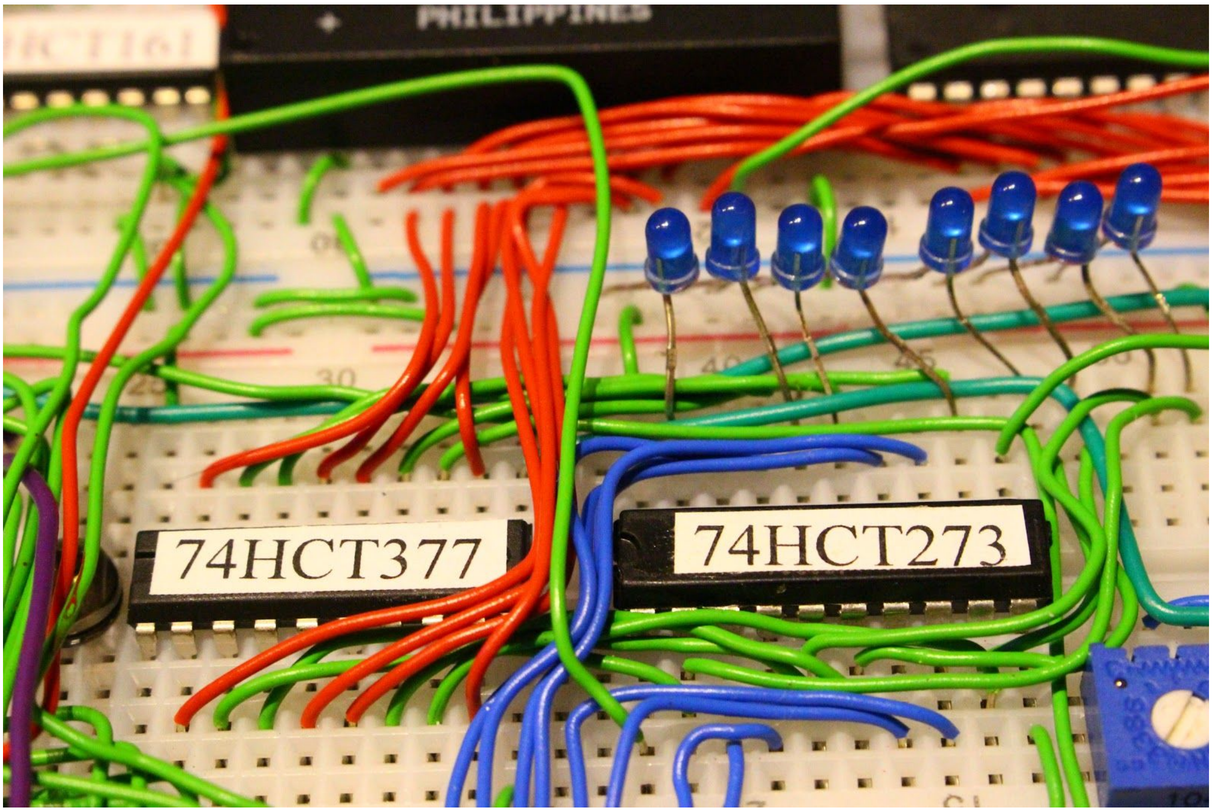
I have copied James bates Microcode for the EEPPROM:s pretty much straight of. You can go to his github and find what you need by yourself, or follow my links to James that I have included below.
I can at least say that you have to write one EEPROM at a time with only one class instruction at a time, so be prepared to write to the EEPROM:s a total of 4*4 = 16 times (4 EEPROMS each with 4 instruction Classes [**MOV**, **LOD**, **STO** and **ALU**]).

It's important that you use James Bates layout of the Arduino programmer as its mirrored unlike Ben Eaters.

James bates github:
https://github.com/jamesbates/jcpu
James Bates Microcode (Arduino):
https://github.com/jamesbates/jcpu/blob/master/arduino/Microcode/Microcode.ino
James Bates Arduino programmer:
https://github.com/jamesbates/jcpu/blob/master/schematics/Arduino_programmer.pdf
James Bates EEPROM librierary (Arduino):
https://github.com/jamesbates/jcpu/tree/master/arduino/libraries/MyEEPROM

3.1.9.2 Instruction register

The instruction register is 8-bit and are feeded to the least significant bits on every EEPROM. The instruction register design is exactly like the **A**, **B**, **C**, **D** or the **stack pointer** design. The instruction register however, does not have the ability to put out it's contents to the buss like the general purpose registers. The task of the instruction register is to inform the EEPROM:s along with the ring counter and flags what the output from the EEPROMs should be.

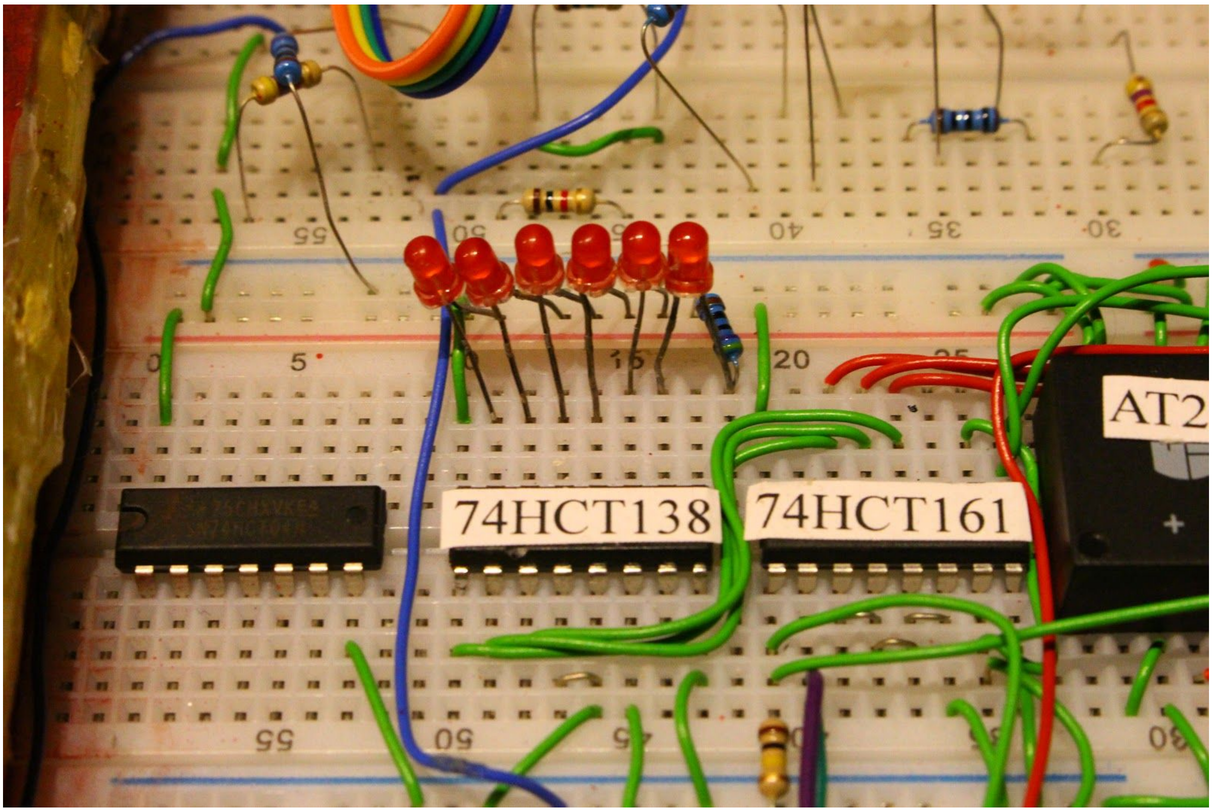


To keep everything run smooth it's necessary to have a *mirror instruction register.* The usual instruction register have its clock input connected to the main clock. The mirror instruction register have it's clock input connected to the inverted clock. This is to make sure that all modules holds it enable/disable state when a new instruction enter the instruction register. As soon as the instruction register changes its value when the computer is in it's execution cycle (micro time T >= 2), the modules are going to change their enable/disable state depending on the instruction.

To summarize this, a mirror register is necessary to make sure that every instruction is completed before next enters the instruction register, this happens on the mirrored clock edge.

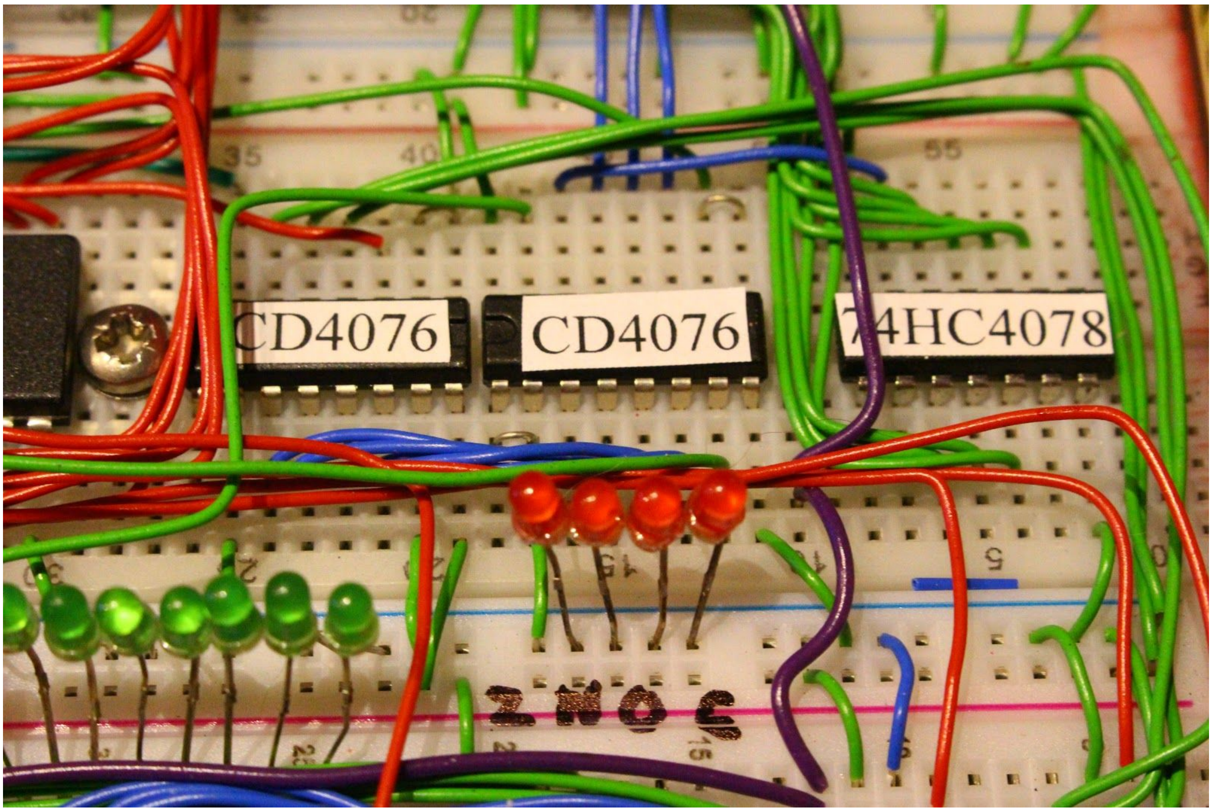
3.1.9.3 Ring counter

The ring counter is a 74HCT161 4-bit binary counter plus a 74HCT138 decoder. It only uses 3-bits and is feeded to the middle significant bits to the every EEPROM. The ring counter task is to hold track of what micro time the computer currently is executing. This acts like the micro time **T** in the arduino Microcode. Every instruction is divided into a number of microtimes. The micro time is usually between 2 to 6 clocks cycles long in my computer, depending on the code executed. The ring counter resets when the execution is complete, and next instruction cycle can enter.



3.1.9.4 Status Register (Flags)

The flags is 2-bits and is feeded to the most significant bits (MSB) to the every EEPROM. There are four flags; **Z**ero flag, **N**egative flag, **O**verflow Flag and **C**arry flag. the status register also have a mirrored register that is connected to the inverted clock, the same reason as the instruction register, as it also directly connected to the EEPROM:s.



3.1.9.5 Fetch and Execution cycle

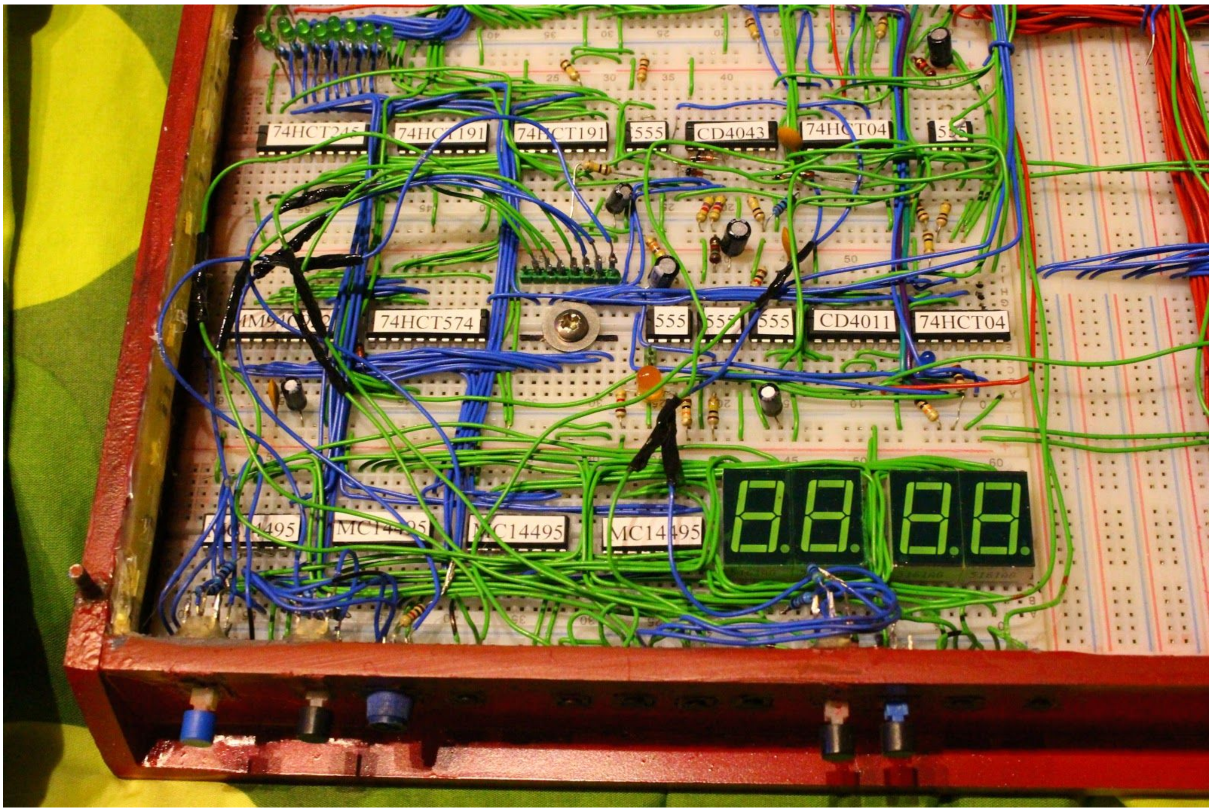
An instruction consist of both fetch and execution cycle. In the fetch cycle, the program counter sends it data (the current address) to the memory address register (MAR). The data in that address in the RAM is copied. This is the first instruction in our program. This data is then copied to the instruction register see (**Instruction register 3.1.8.2**). The computer have *fetched* the instruction we have programmed. These two steps are always the same, it doesn't matter what instruction the instruction register is loaded with in the fetch cycle.

In the execution cycle however, the next steps after the fetch cycle depends on what instruction the instruction register is loaded with. For example, an ADD instruction is the same as the SUB instruction in the fetch cycle, but the execution cycle differ from each other.

3.1.10 Input module

The biggest difference between my computer and James Bates computer is the input module. The Input module is located in the left down corner, as marked in the picture. The input module also consist of the input panel as shown in (pic 1.2), but also the Arduino “bootloader” which is located in the up right corner and are used to load the EEPROM loader with data from a text file. I talk more about this in (**3.1.10.1 Arduino “bootloader”**).

The input module allows the programmer to input data to specific addresses in RAM to make a program. The computer automatically goes to the next address when the programer have write a byte to RAM, or pressed on the ***decrease address*** or the ***increase address*** button. The two left 7 segment displays shows the address, and the two on the right shows the data on that address.



The big black buttons are as follows from left to right:
***On/Off (blue)**, **Manual clockmode/Auto clockmode**, **Programingmode/Runmode**.*

The small black buttons are as follows from left to right:
***single step**, **write data**, **decrease address**, **increase address**, **goto address xx**, **instruction step**.*
The blue potentiometer controls the frequency of the clock and the red dip switch to the right is used to chose what address space you want to work on. For example, you can have up to 16 (4 bit) programs loaded in the EEPROM.

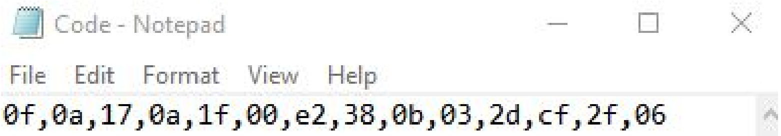
3.1.10.1 Arduino “Bootloader”

This part is not necessary for the computer to function properly, it just add a bit of comfort as you don't need to program the computer by the keypad, you can just write a program in HEX code on a .txt file on a computer and then send it over to the 8-bit computer, using Processing and Arduino.

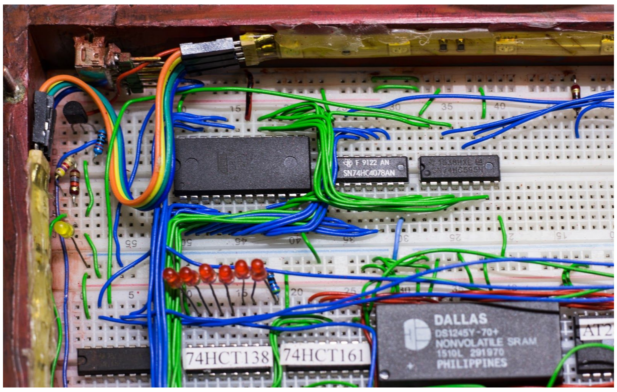
This requires that you have both Arduino and Processing installed on your computer.

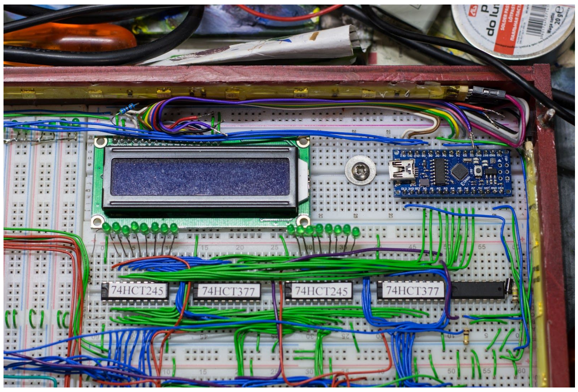
https://www.arduino.cc/en/Main/Software
https://processing.org/

You can get the code for both Processing and Arduino here:
https://github.com/xrayer2000/Bootloader-and-LCD-BIN-to-DEC



This is based on the idea that Processing reads a text file with HEX code on the computer. More specifically, Processing read one character at a time, the ASCII value, and sends it over to the Arduino via serial port with the “Serial.read()” function. Due to that a HEX byte is two characters, the Arduino need to read twice. With some clever techniques the Arduino builds a hexadecimal value from the two characters (0-F). The value is sent to the shift register (74HC595) with the “shiftOut()” function, which is directly connected to the EEPROM data pins. This work due to the built in OE signal in the 74HC595.





3.1.11 Power distribution and tips

Something annoying about troubleshooting something that dont work, is that you either have a bad powersource or a bad power distribution across the computer. The easiest thing to do if you don't know if you suffer from this, is to use a multimeter to measure the voltage across the power rails on all breadboards. You could also measure the resistance between the rails. This value should be as high as possible (20K + ohm, more the better). I have experienced this alot, and so have others, I suggest you go to this reddit post were a kind soul have written about the most common problems you can encounter. He talks about the importance of not mixing logic families (I know I do that on some ic:s but i'm trying to avoid it at all cost), have decent connections between all breadboards, etc.
https://www.reddit.com/r/beneater/comments/dskbug/what_i_have_learned_a_master_list_of_what_to_do/
One trick I came up with is that you can solder a resistor directly to the led leg, to take no additional place on the breadboard. It's important that you don't skip the resistors between the leds and the outputs of a module, you can damage the IC and it draws more current.

3.2 How does the computer work?

3.2.1 Power

First of all, connect the output port to USB charger (2A or what you have in hands, doesn't really matter if its 1A, 2A or 3A, the computer aren't gonna draw more than 1 amp, and if it does, something is wrong in the circuit), you need to make a customized usb cable that have male contact on both ends, this can be achieved by taking two usb cables and solder the male contacts together, and discard the female ones. Then, you can press the on/off button, which is the most left one, to start the computer. It's recommended to always reset the computer every time you power it, to ensure the instruction register and program counter is set to 0.

3.2.2 Program basics

First of all, to make the computer run, it need a program to be executed. The program is a list of instructions that the computer understands and can execute. Some programs a linear, and some are nonlinear. Linear programs go from the first address to the last program address and executes each instruction in a linear line with no exception. Nonlinear programs have so called "jump" instructions that allow the computer to skip to another address. This jump instruction can be conditional or unconditional. A conditional jump instruction only jump to a specific address when a certain condition is met. In this computer there is four conditional jump instructions (there are more, but not in my computer); jump if **Zero**, jump if **Negative**, jump if **Carry**, jump if **Overflow**. A unconditional jump is just called "jump", and jumps to a specific address no matter what. The conditional and unconditional jump instructions allow the programer to make so called "if-statements", similar to those in modern programming languages, like Java, C++, or C#.

The "#" symbol indicates that the instruction is two bytes long, meaning it has an instruction with a immediate value, meaning that the computer don't need to look a specific address in RAM to but instead take the value from the instruction after the #. For example let say that we want the program counter to load the value 0 (jump to address 0) we can write (38 # 00), this takes two bytes.
that looks like this in machine code:

00 [00111000]
01 [00000000]

3.2.2.1 How to program it?

To program it, simply put the Programmode/runmode switch to programmode (LED light red). Then enter the program code via the keypad. Each instruction is between one and two bytes. One byte consist of two hex symbols (00-FF). When two keys have been pressed in sequence, you can either reset the keypad data or push the button for write. To reset the keypad you can simply just wait a few seconds, or by press any of the following buttons; **address step right**, **address step left**, **address step to address xx**, or the most convenient option, the **reset button**. You can also program it by letting the arduino read from a text file, see (**3.1.10.1 Arduino "bootloader"**).

Down below are two sheets explaining a little about how an instruction is built up. There are 4 instruction classes, **MOV** (Move) **LOD** (Load) **STO** (Store) **ALU** (Arithmetic and Logic). MOV, LOD and STO are classes that have operand **x** and **y**, and is 3 bits wide respectively to encode the 6 registers (**A [000], B [001], C [010], D [011], Stack Pointer [100]**, **Program Counter [101**]). The class is built up by 2 bits.
In Total there are 8 bits building a byte (2 + 3 + 3 bits).

The **MOV** class can move data from register x to register y.
The **LOD** class can move data from RAM address x to register y
The **STO** class can move data from register x to RAM address y
The **ALU** class performs calculations on the B-Register (fixed) and one general purpose register (A, B, C or D-Register)

GreenStone - 8 bit computer

Instructionset   XXXXXXXX

4 klassen: MOV, LOD, STO o ALU (00, 01, 10, 11)

MOV (Data): instruktion för att ladda imidete värden

Data	Ra	07	#XXXXXXXX
	Rb	0F	
	Rc	17	
	Rd	1F	
	PC	2F	(Jump)
	SP	27	

MOV:	Rx ← Ry	00 XXX XXX (X, Y)	
LOD:	R[X] ← Ry	01 XXX XXX (X, Y)	(01XXX111) # X
STO:	Rx ← R[Y]	10 XXX XXX (X, Y)	(10111XXX) # X
ALU:		11 X XX XXX (A/C, OP, Rx)	(Rb, Rx)

MOV Class

Conditional jumps:

00 1110XX

(XX)	
00	JC
01	JZ
10	JN
11	JO

⊗⊗⊗⊗
ZNOC

CMP

- JC – Jump if >=0 (38)
- JZ – Jump if == 0 (39)
- JN – Jump if < 0 (3A)
- JO – (3B)

ALU klassen:

		S2 S1 S0	An	Bn	Cn	opcode
ADD	Rx, Rb	A Plus B	Rx	Rb	0	11 0 011 XX
ADC	Rx, Rb	A Plus B	Rx	Rb	CF	11 1 011 XX
SUB	Rx, Rb	A Minus B	Rx	Rb	1	11 0 010 XX
SBC	Rx, Rb	A Minus B	Rx	Rb	CF	11 1 010 XX
SUB	Rb, Rx	B Minus A	Rx	Rb	1	11 0 001 XX
SBC	Rb, Rx	B Minus A	Rx	Rb	CF	11 1 001 XX
CMP	Rx, Rb	A Minus B	Rx	Rb	1	11 1 110 XX
AND	Rx, Rb	A AND B	Rx	Rb	0	11 0 110 XX
OR	Rx, Rb	A OR B	Rx	Rb	0	11 0 101 XX
XOR	Rx, Rb	A XOR B	Rx	Rb	0	11 0 100 XX
NOT	Rx	B Minus A	Rx	0	0	11 0 111 XX
TST	Rx	A Plus B	Rx	0	0	11 1 111 XX
INC	Rx	A Plus B	Rx	0	1	11 0 000 XX
DEC	Rx	A Minus B	Rx	0	0	11 1 000 XX
push	Rx	A Minus B	SP	0	0	10 110 XXX
pop	Rx	A Plus B	SP	0	1	01 XXX 110

HLT: 2d

CMP: C=1 IF $(a-b) >= 0$

Methods:

17#XX (Method Adress)

b5 call Method

...

Adress (XX) Method {

...

6E (return)

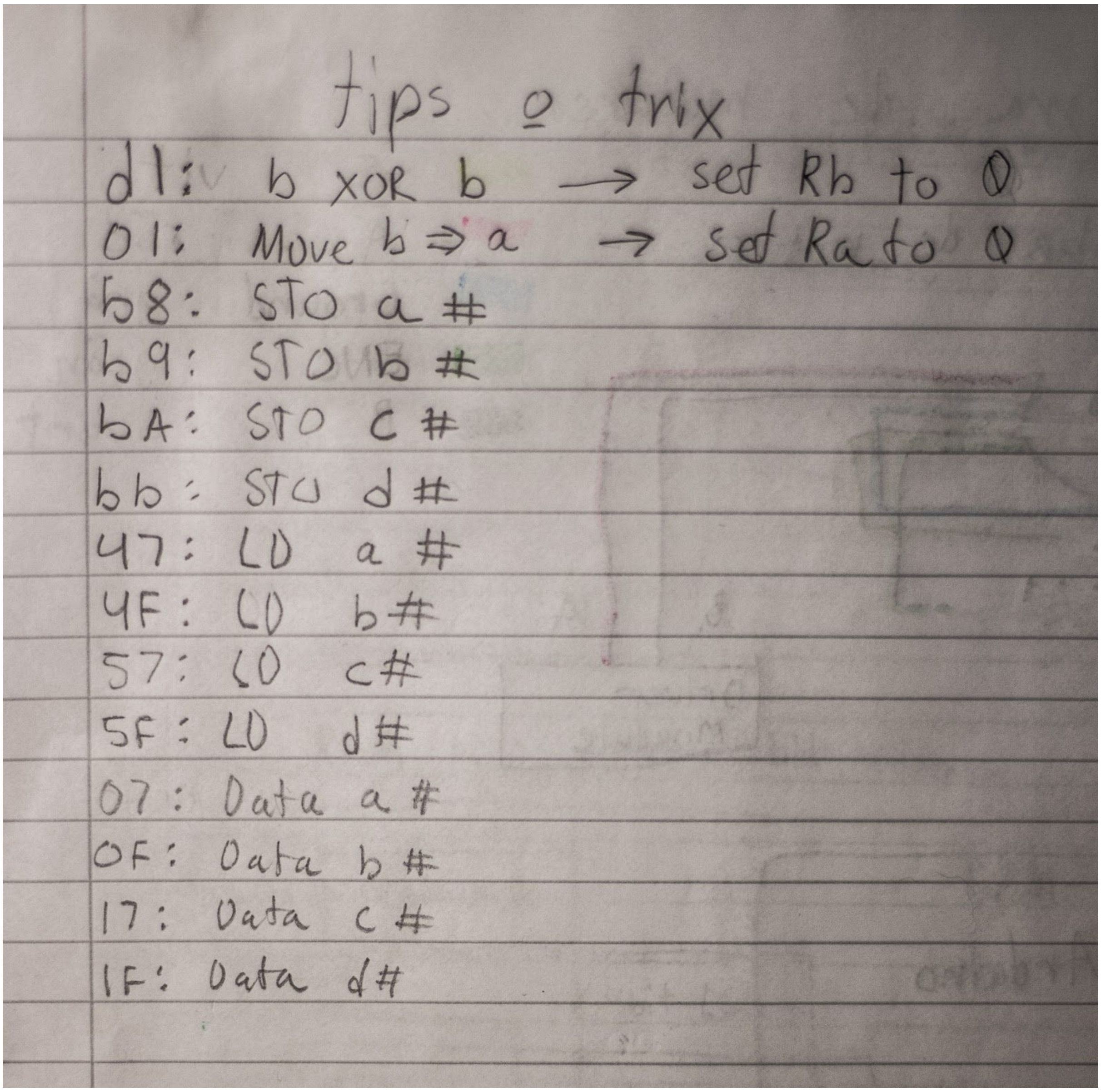
}

Push:

10110XXX (Rx)

Pop:

01XXX110 (Rx)



tips o trix

d1: b XOR b → set Rb to 0
01: Move b ⇒ a → set Ra to 0
b8: STO a #
b9: STO b #
bA: STO c #
bb: STO d #
47: LD a #
4F: LD b#
57: LD c#
5F: LD d#
07: Data a #
0F: Data b #
17: Data c #
1F: Data d#

 # - immediate instructions

3.2.3 Example Programs

Where it says “#a” and “#b” it means that its variable a and variable b, like in a x b = c.

You can have a text file that looks like this to store more than one program. Processing only gonna read until it collides with a blank space. “,” works as byte separators. Make sure that the current program you want to upload is at the bottom of the text file and that there are no blank spaces after or under the program text, you can only upload one program at a time.

Code - Anteckningar

Arkiv Redigera Format Visa Hjälp

```
// Addition
07,30,0f,38,cc,2d

// Multiplication
0f,0a,17,0b,1f,00,e2,38,0b,03,2d,cf,2f,06

// Division
07,07,0f,02,1f,00,c8,c3,f8,38,06,03,2d

// Modulo
07,07,0f,02,f8,38,09,2f,0c,c8,2f,04,2d

// Mov without Mov instruction
07,01,0f,02,90,c2,91,e2,4a,c2,42,2d

// Adding one Loop
0f,01,07,00,cc,38,00,2f,04

// Fibonacci numbers
d1,01,10,c1,ce,38,00,02,11,08,2f,04
```

3.2.3.1 Program - Addition

```
// Addition
07,30,0f,38,cc,2d
```

The program is a linear program and goes as follows:

MOV initialize register A with the variable a with value a (10 DEC)
MOV initialize register B with the variable b with value b (11 DEC)
ALU Add value of register B to register A
MOV Halt

3.2.3.2 Program - Multiplication

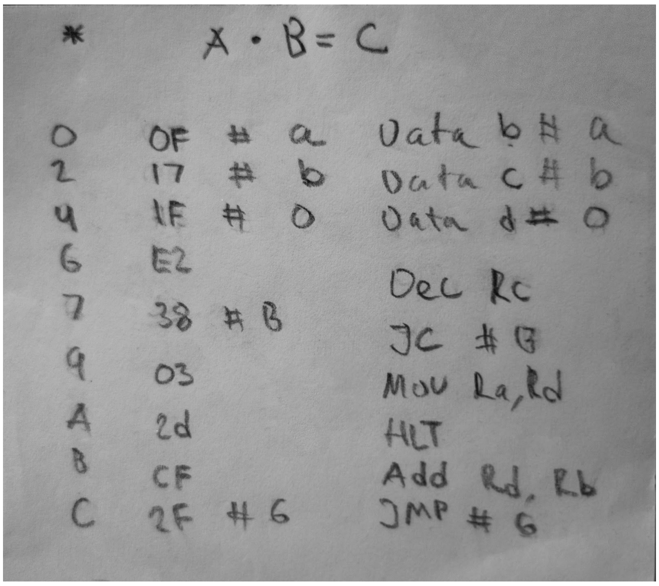
The program is a nonlinear program and goes as follows:

```
// Multiplication
0f,0a,17,0b,1f,00,e2,38,0b,03,2d,cf,2f,06
```

MOV initialize register B with the variable a with value a (10)
MOV initialize register C with the variable b with value b (11)
MOV initialize register D with the value 0
06 **ALU** decrease value of register C with one
MOV jump to address 0b if carry (register C > 0)
MOV move value in register D to A
MOV halt
0b **ALU** Add value of register B to register D
MOV jump to address 06

//or this version that looks more familiar to modern programing languages

MOV initialize register B with the variable a with value a (10)
MOV initialize register C with the variable b with value b (11)
MOV initialize register D with the value 0
06 **ALU** decrease value of register C with one
if carry (register C > 0)
{
    MOV jump to address 0b
0b **ALU** Add value of register B to register D
    MOV jump to address 06
}
else
{
    MOV move value in register D to A and halt
}



3.2.3.3 Program - Division

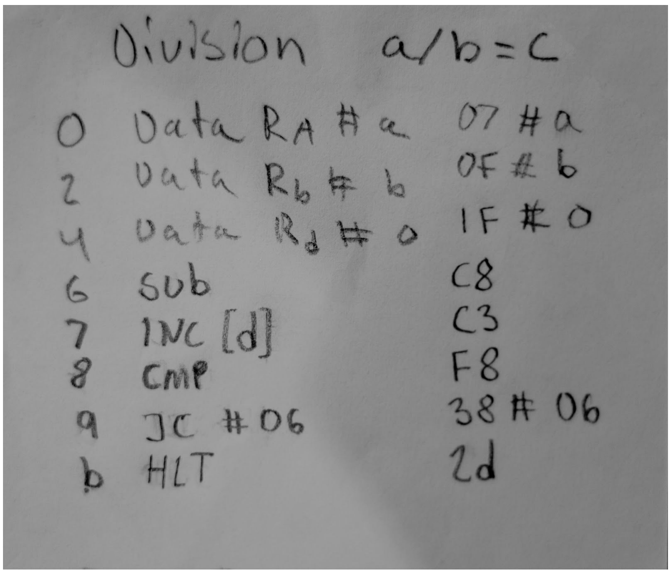
```
// Division
07,0a,0f,0b,1f,00,c8,c3,f8,38,06,2d
```

The program is a nonlinear program and goes as follows:

MOV initialize register A with the variable a with value a (10)
MOV initialize register B with the variable b with value b (11)
MOV initialize register D with the value 0
06 **ALU** subtract value of register B from register A
ALU increase register D with one
ALU Compare (A minus B, used to se if register A is bigger than register B)
MOV jump to address 06 if carry
MOV Halt

//or this version that looks more familiar to modern programing languages

MOV initialize register A with the variable a with value a (10)
MOV initialize register B with the variable b with value b (11)
MOV initialize register D with the value 0
06 **ALU** subtract value of register B to register A
ALU increase register D with one
ALU Compare (A minus B, used to se if register A is bigger than register B)
if carry
{
  MOV jump to address 06
}
else
{
  MOV Halt
}

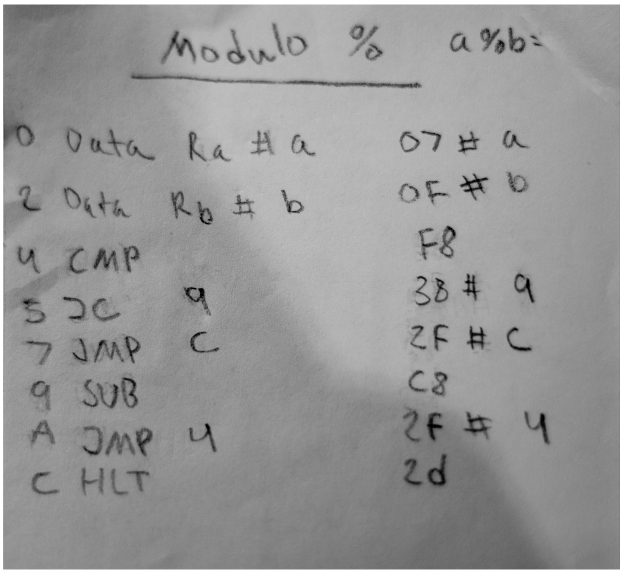


3.2.3.4 Program - Modulo

```
// Modulo
07,0a,0f,0b,f8,38,09,2f,0c,c8,2f,04,2d
```

The reminder in integer division.
The program is a nonlinear program and goes as follows:

	MOV initialize register A with the variable a with value a (10)
	MOV initialize register B with the variable b with value b (11)
04	**ALU** Compare (A minus B, used to se if register A is bigger than register B)
	MOV jump to address 09 if carry
	MOV jump to address 0c
09	**ALU** subtract value of register B to register A
	MOV jump to address 04
0c	**MOV** Halt

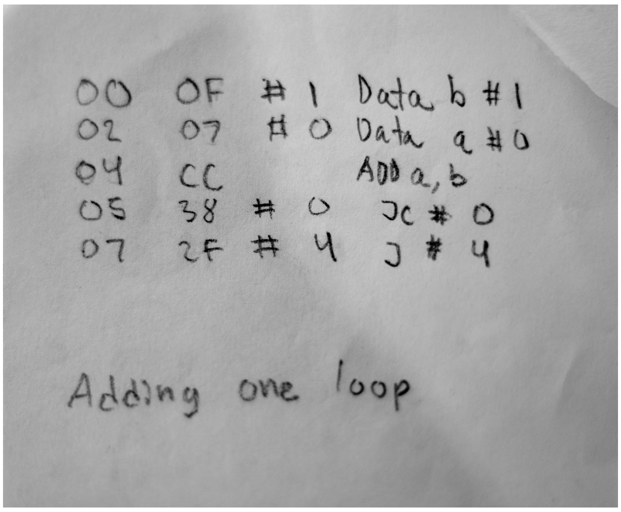


3.2.3.5 Program - Adding one loop

```
// Adding one loop
0f,01,07,00,cc,38,00,2f,04
```

The program is a nonlinear program and goes as follows:

00	**MOV** initialize register B with the value 1
	MOV initialize register A with the value 0
04	**ALU** Add value of register B to register A
	MOV jump to address 0 if carry
	MOV jump to address 04

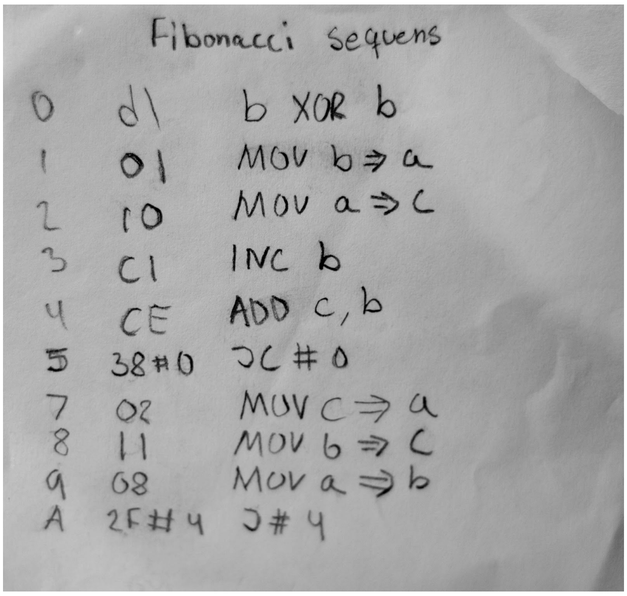


3.2.3.6 Program - Fibonacci numbers

```
// Fibonacci numbers
d1,01,10,c1,ce,38,00,02,11,08,2f,04
```

The program is a nonlinear program and goes as follows:

0 **ALU** register B XOR register B (shortcut for initialize register B with 0)
MOV value of register B to register A (initialize register A with register B [value 0])
MOV value of register A to register C (initialize register C with register A [value 0])
ALU increase register B with one
04 **ALU** Add value of register B to register C
MOV jump to address 0 if carry
MOV value of register C to register A
MOV value of register B to register C
MOV value of register A to register B
MOV jump to address 04



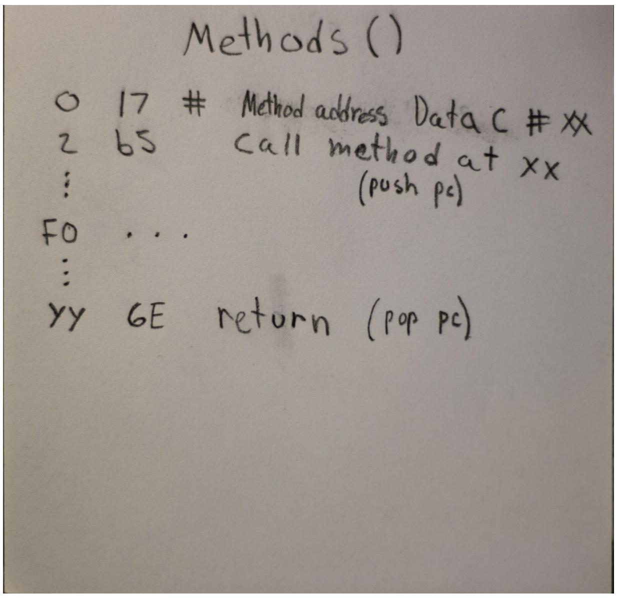
3.2.3.6 Program - Methods

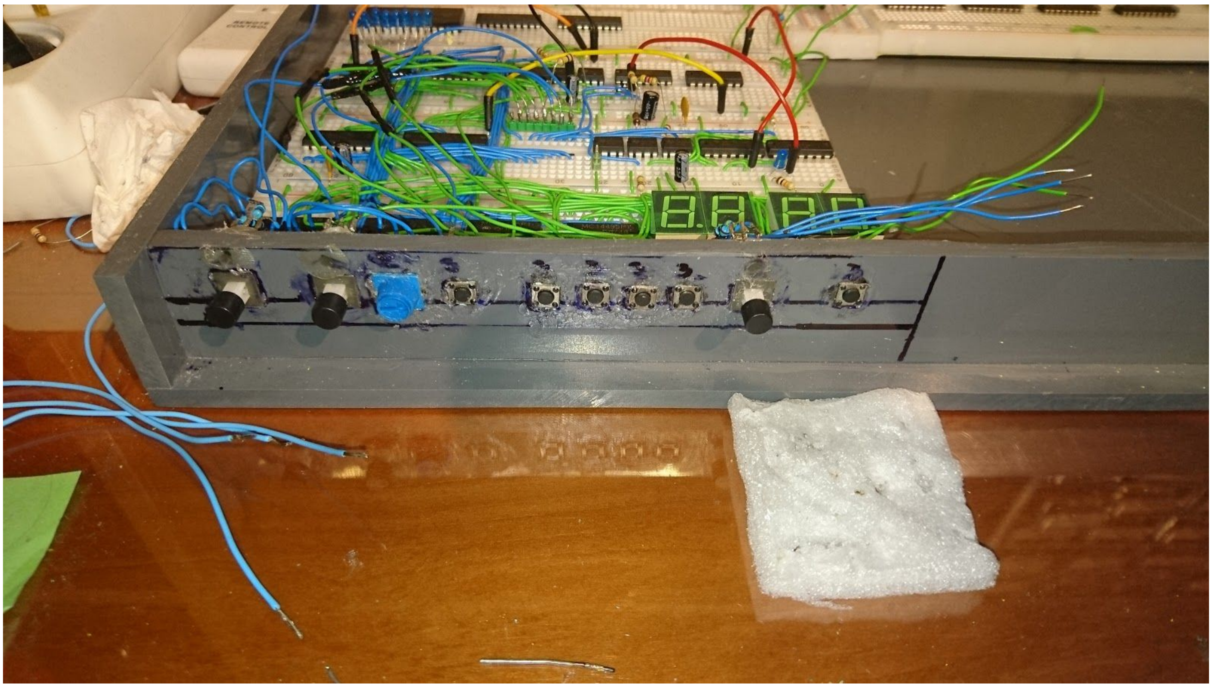
```
//Methods
17,f0,b5
.
.
.
2d
(Method)
.
.
.
6e
```

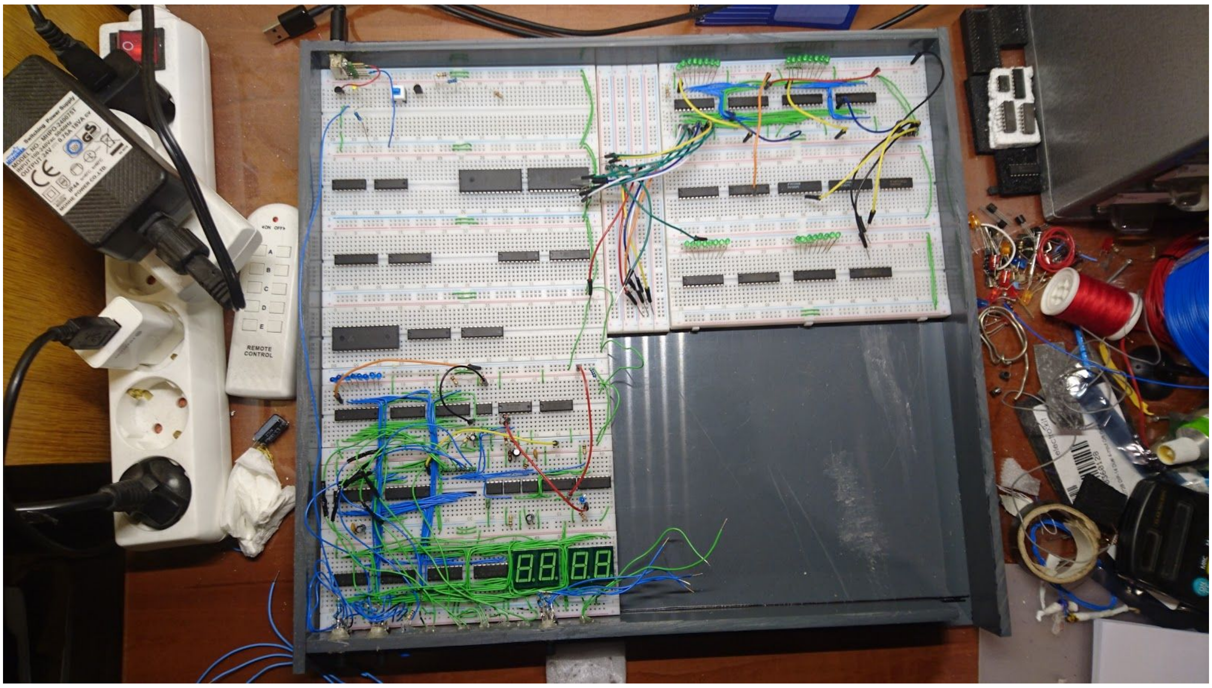
A method can be whatever you want. Methods is useful for reuse code without actually rewrite it, you just have to write it once. In this example I show you the multiplication program as a method. Be aware that if you have something stored in the C-register, that data is gonna be replaced with the method address, due to that it's bind to the call instruction (b5) for saving the address from the program counter, which is a bit of a down side.

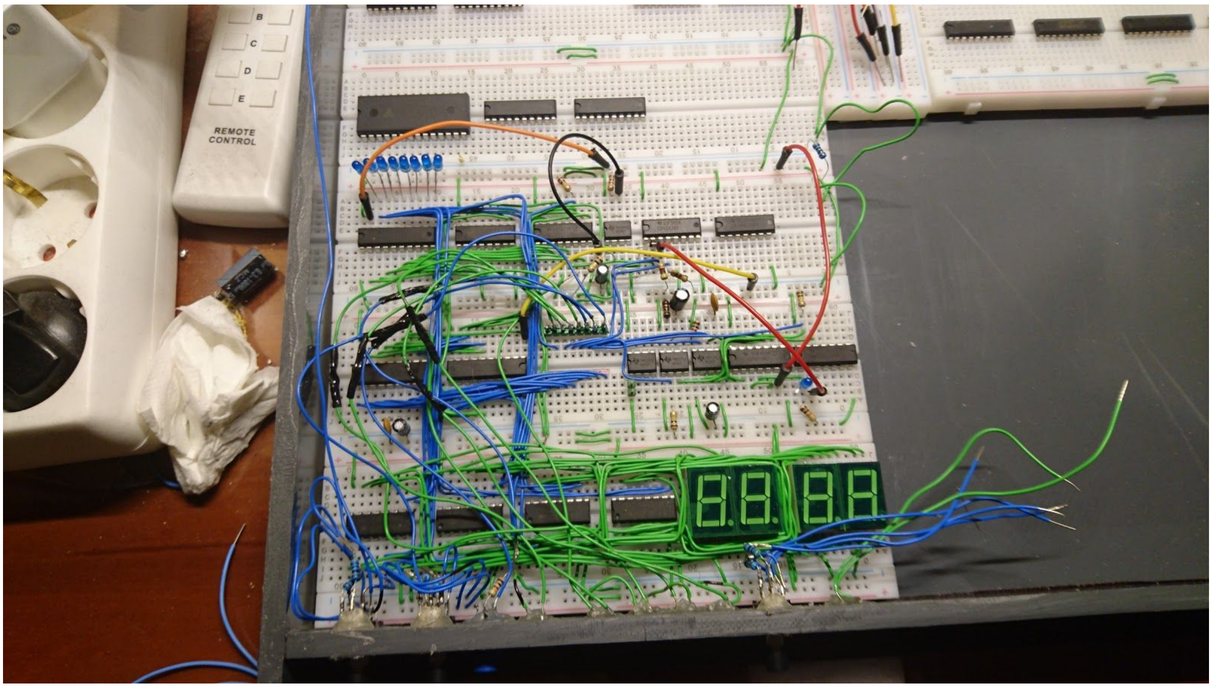
The program is a nonlinear program and goes as follows:

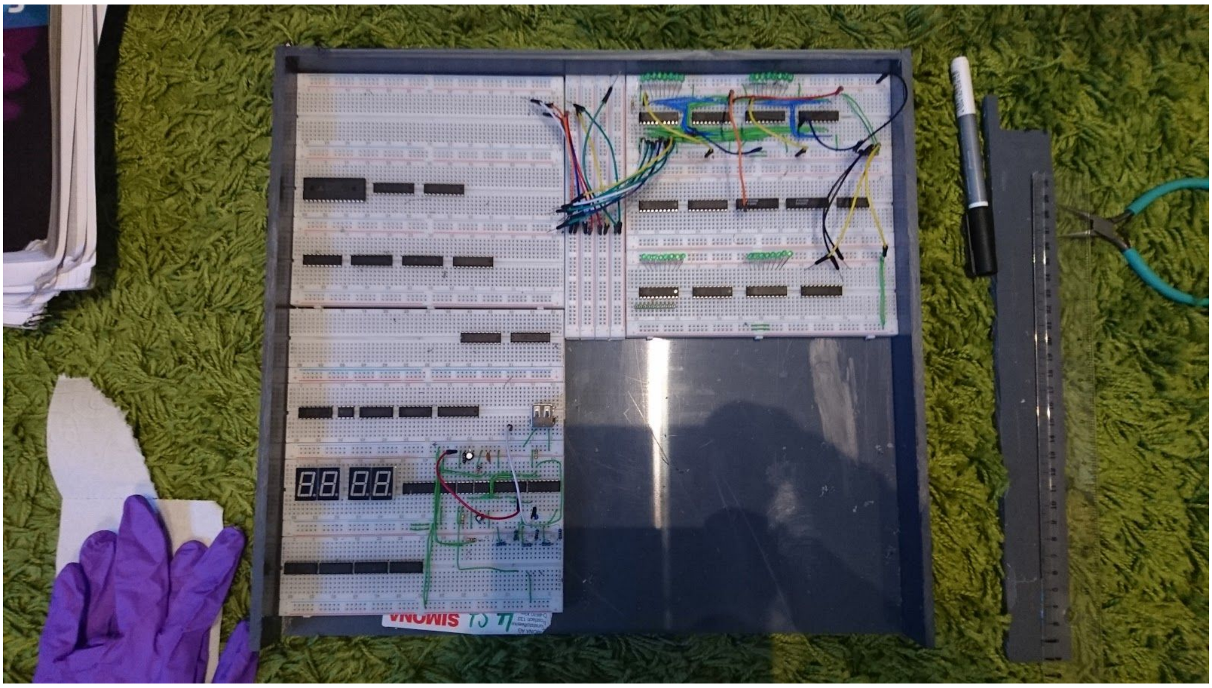
00 **MOV** initialize register C with address of method (F0)
02 **STO** push to program counter (call method)
.
.
.
Method()
F0 **MOV** initialize register B with the variable a with value a (10)
MOV initialize register C with the variable b with value b (11)
MOV initialize register D with the value 0
F6 **ALU** decrease value of register C with one
MOV jump to address 0b if carry (register C > 0)
MOV move value in register D to A
MOV halt
Fb **ALU** Add value of register B to register D
MOV jump to address 06
LOD pop from program counter (return from method)

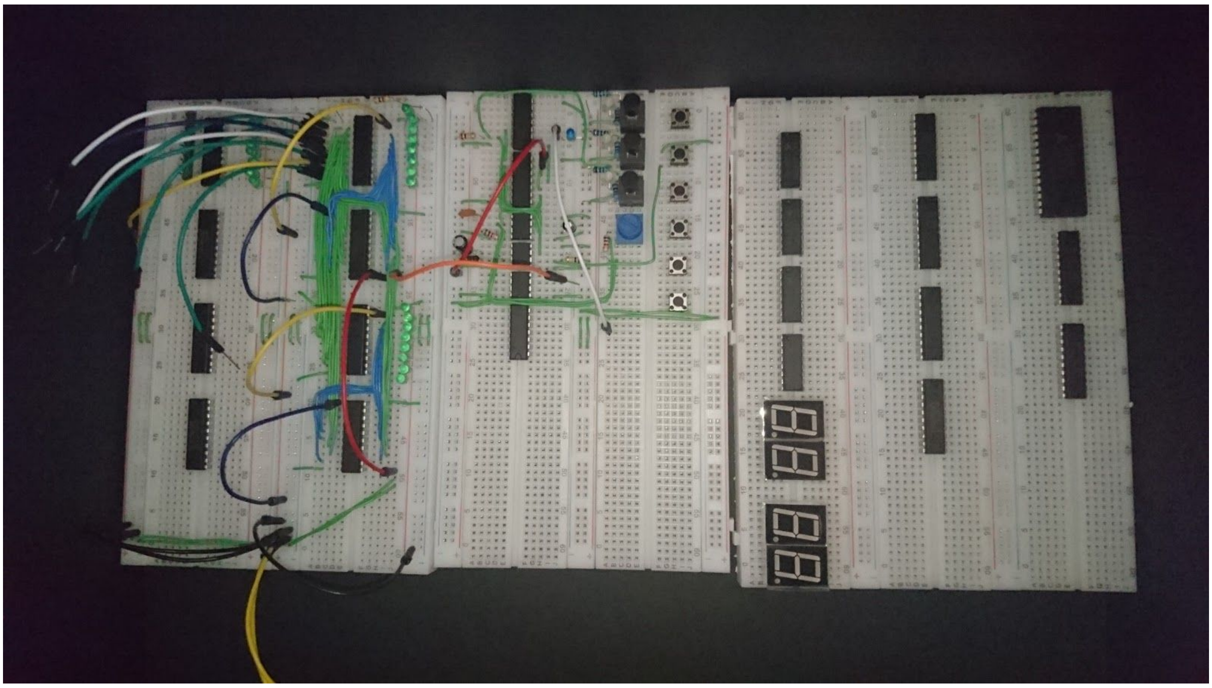


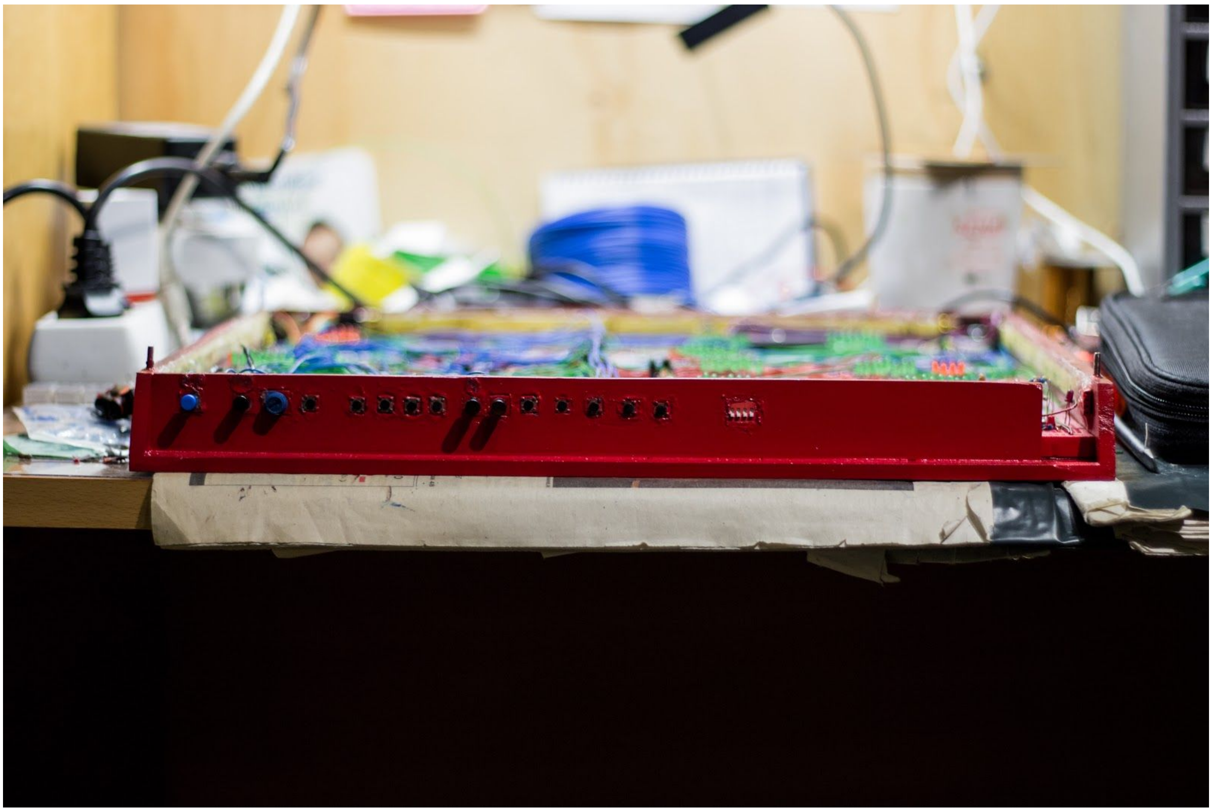





REMOTE
CONTROL







4. CONCLUSION AND DISCUSSION

if I had to remake the computer, there are some things that I wanted to change. First of all I would like to add more conditional jumps, like “jump if no carry”, “jump if no zero” and “jump if no negative”. That would make some part of the code smaller and more effective.
I think it would be really nice if I included interruptions. That would mean I could design a input command that ask the user for example to press a button, and then execute code based on that. But these things are out of my knowledge range, but who knows, you who are reading this can maybe put a thought in it, and if you figure out some modifications I would be more than happy to know.
I want to say that it's a good to be stubborn. This project is gonna take a while. You are going to encounter a **LOT** of problems when you build your computer. Some great tools for diagnostic testing, is a multimeter of course, but you can also build your self a simple test probe by soldering a common 5 or 3mm led to a resistor (470 ohm) and a wire, preferably a multithreaded. By using these tools you can detect errors you make, or errors that just arises “by itself”, believe me, this can happen. led resistor

5. SUMMARY

The goal of this project was to educate people who are new to 8-bit computers. The documentation addresses the range from the most basic concept of computers, which is to store a 1 or a 0, apply boolean functions for bits, to more advanced modules, such as the Arduino “bootloader”. This project had taken me long time to be “finished”, with finished I mean I have acquired the level of complexity I was aiming at. You can always add more things to a project like this, the thing that holds you back is your imagination. It started back when I was 15 year old, when I found Ben Eaters youtube channel watching his 4 bit adder. Now, im almost 20 years. After that, he started making the 8-bit computer series that I followed along with from the first to the last video. Ben Eater have teached many of us some important knowledge about computers that we should be really thankful for, and I know that many are.

If you're reading this, Thank you, **Ben Eater and James Bates**!

6. LIST OF REFERENCES

My Youtube channel (Xrayer):
https://www.youtube.com/channel/UCh3Q-7RZbRZGIu6rLeAsR-Q?view_as=subscriber
Ben Eater Youtube channel:
https://www.youtube.com/channel/UCS0N5baNlQWJCUrhCEo8WlA
James Bates Youtube channel:
https://www.youtube.com/channel/UCH09NwwJsfThwLKvc6kxl4
James Bates github:
https://github.com/jamesbates/jcpu
James Bates Microcode (Arduino):
https://github.com/jamesbates/jcpu/blob/master/arduino/Microcode/Microcode.ino
James Bates Arduino programmer:
https://github.com/jamesbates/jcpu/blob/master/schematics/Arduino_programmer.pdf
James Bates EEPROM librierary (Arduino):
https://github.com/jamesbates/jcpu/tree/master/arduino/libraries/MyEEPROM
Arduino download page:
https://www.arduino.cc/en/Main/Software
Processing download page:
https://processing.org/
Code for Arduino and Processing (Bootloader and LCD code)
https://github.com/xrayer2000/Bootloader-and-LCD-BIN-to-DEC