The Architecture of a Simple 8-bit Computer

Marc Widmer

supervised by

Mario Häfeli

Maturitätsarbeit 2017

MNG Rämibühl

April 15, 2017

Abstract

This matura project consists of two parts: Building a simple 8-bit computer based on the SAP-1 architecture and a documentation explaining my improved version of this architecture. The paper firstly provides some basic knowledge necessary to understand this computer architecture and then continues with an explanation of the different parts which this architecture is made up with. In order to establish an understanding of how the computer operates, the execution of a program will be discussed in great detail. The explanation concludes by exploring its capabilities. The second to last section gives some advice on building such a computer and is accompanied by the computer's schematics in the Appendix. The end will reflect on the problem of using breadboards for such a project and enumerate possible improvements.

Contents

1 Introduction 2

2 Prerequisites 4

2.1 The Binary Numeral System . . . 4

2.2 Logic Gates . . . 6

2.3 Addition and Subtraction in a Digital Circuit . . . 7

3 Architecture 9

3.1 Units . . . 10

3.2 Operation . . . 13

3.3 Capabilities . . . 23

4 Advice on Building a 8-bit Computer 26

5 Conclusion 27

6 Appendix 28

6.1 Schematics . . . 29

1 Introduction

In our modern world, computers are found almost everywhere. Yet most people know barely anything about how they work. Being personally very interested in computer programming, I wanted to know how a computer executes code on the lowest level. After spending multiple evenings trying to find an answer on the internet, I was very disappointed as I could not find a satisfactory explanation. Eventually, I stumbled over a Youtube video [1] from someone called Ben Eater in which he presents his home-built 8-bit computer. After watching another one of his videos [2], explaining how a simple program runs on his computer, I was determined to build such a computer.

The result of this project was a 8-bit computer built on prototyping board called breadboards. It has 16 bytes of memory used to store a program, which is capable of performing automated computation with minimalistic conditional branching. It expands on the SAP-1 (Simple-As-Possible) computer which is a very basic model of a computer designed as example architecture for beginners. As it is meant to demonstrate how a computer works, it has arrays of LEDs at key points in the computer providing insight into what happens within the circuits. It uses 7-segment displays to present the results of calculations in decimal form.

Building the computer was the main part of this project yet this paper focuses on the explanation of the architecture and its operation. Additionally I provide the necessary plans to build it.

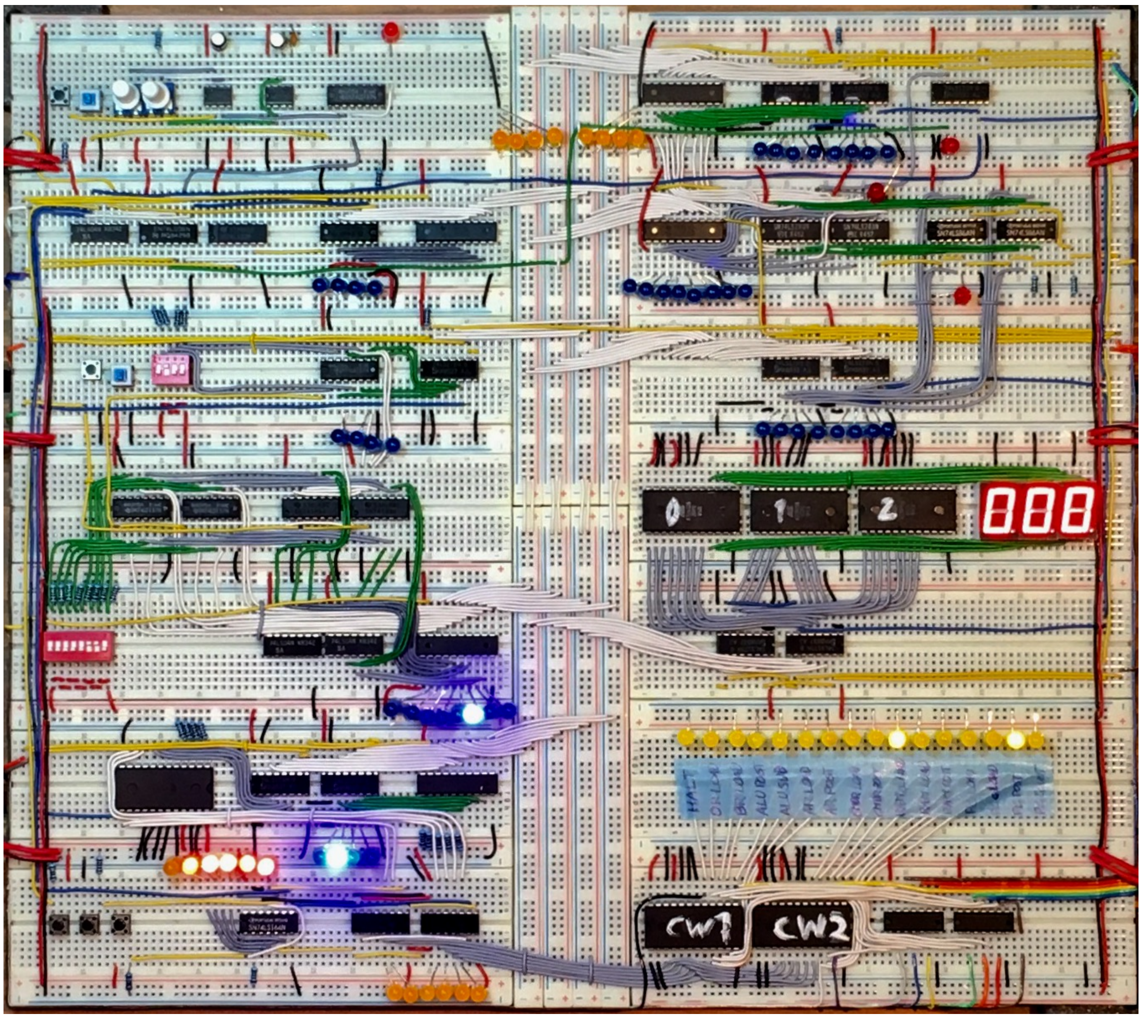


Figure 1: 8-bit computer

2 Prerequisites

2.1 The Binary Numeral System

Inside a computer there can either be a voltage or no voltage, represented by 1 and 0. Therefore, a computer expresses numbers using the binary numeral system, also known as base-2 numeral system. The binary numeral system uses positional notation with a radix of 2. The decimal system also uses positional notation but with a radix of 10. This means a binary number B is represented as

$$B = a_0 2^0 + a_1 2^1 + a_2 2^2 + ... + a_n 2^n$$

where the coefficients a_i are either 0 or 1.
For example, the binary number 00101001 equals 41 in the decimal system. The leading zeros may be omitted. However, as this paper discusses a 8-bit computer, all binary numbers are represented in 8-bit length.

For negative numbers one is used to place a minus sign in front of the number. When using the binary system this works too, yet this cannot be implemented in a computer because it cannot represent a minus sign. It can only represent a 1 or a 0. As a work around for this, various methods exist by which negative numbers can be represented in binary and can be implemented in a computer. The one used in this architecture is the so called "two's complement". This method is not as straight forward as other methods however it makes the implementation of subtraction in a digital circuit fairly simple. To make a number's "two's complement" there are two steps. Firstly, it has to be inverted. This means all the ones have to be turned into zeros and all the zeros into ones. After this, 1 has to be added to it. So for example the "two's complement" of 00101001 is 11010111.

Calculations with binary numbers are performed in the same manner as with with decimal numbers. As an example, you below find an addition and a subtraction of two binary numbers using long addition. For the subtraction, the "two's complement" of the subtrahend is added to the minuend.

$$\begin{array}{r}
1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
+\ 0\ 0\ 0\ 0_1\ 1_1\ 1\ 0\ 0 \\
\hline
1\ 0\ 0\ 1\ 1\ 0\ 1\ 1
\end{array}
\qquad\qquad
\begin{array}{r}
1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
+_1 1_1\ 1_1\ 1_1\ 1_1\ 0_1\ 1\ 0\ 0 \\
\hline
1\ 0\ 0\ 0\ 0\ 0\ 1\ 1
\end{array}$$

Table 1: Addition and subtraction in binary

The results seem to be correct yet one might be confused by the ignored last carry in the subtraction. The reason for its neglection is that it exceeds the 8-bit limit of this architecture and therefore creates a so called overflow. Ignoring overflow usually results in a wrong result yet, when using the "two's complement", this is meant to happen.

2.2 Logic Gates

The fact that computers are built up of transistors is well known and correct. However, going down to the level of transistors to explain how a computer works makes everything extremely complicated and confusing. For this reason, the explanation in this paper will not dive to deeper levels than the level of logic gates. Logic gates are small electric circuits considered as being the basic building block of digital logic. They take a certain number of inputs, usually two, and have one output. The following listing explains the behaviour of some of the most common logic gates.

- An **AND gate** outputs a high voltage if all of its inputs are high.
- An **OR gate** outputs a high voltage if one or more of its inputs are high.
- A **XOR gate** outputs a high voltage iff one of its inputs is high. Another common name for this gate is exclusive OR gate.
- A **NOT gate** outputs the inverse of its input. They have only one input.

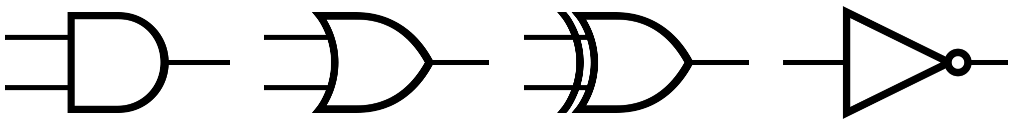


Figure 2: The common symbols of the before described logic gates in the order of their description.

A more abstract but very common way of describing binary logic are truth tables (Table 2). These are tables with two sides, the left denotes all possible input combinations and the right denotes the corresponding output.

A	B	Q
0	0	0
1	0	0
0	1	0
1	1	1

A	B	Q
0	0	0
1	0	1
0	1	1
1	1	1

A	B	Q
0	0	0
1	0	1
0	1	1
1	1	0

A	Q
0	1
1	0

Table 2: Truth tables of the before described logic gates

2.3 Addition and Subtraction in a Digital Circuit

Addition is one of the most fundamental elements in a computer. Its implementation in a digital circuit is called **full adder** and based on long addition as it is taught in primary school. A full adder is an electronic circuit built up of logic gates which computes two bits and a carry-in signal into a result and a carry-out. As each full adder performs its computation on two bits, one bit of each adders, eight full adders are needed to add two 8-bit numbers. These full adders are cascaded together, meaning the carry-out signal of one full adder is fed into the carry-in signal of the next full adder. This allows for the carry signal to ripple through the full adder.

A full adder itself consists of two half adders, which are simple circuits capable of computing only two bits into a result and a carry-out signal, hence lack the capability to handle a carry signal.

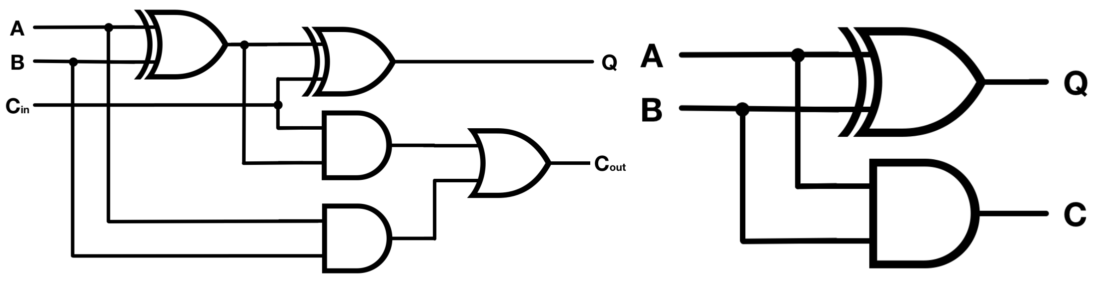


Figure 3: One full adder (left) and one half adder (right)

The behaviour of a half adder can be described using a truth table (Table 3). A and B are the two input bits, Q is the result and and C_{out} is the carry-out signal.

A	B	Q	C_{out}
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Table 3: Half adder truth table

Just as with the half adder, the below truth table (Table 4) depicts the behaviour of a full adder. On the left side there are the input bits A, B and an additional carry-in signal which is fed into the circuit. On the right are the result Q and the Carry out signal

A	B	C_{in}	Q	C_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Table 4: Full adder truth table

An array of full adders can easily add two numbers but cannot subtract. This is where the "two's complement" comes into play as adding the "two's complement" of a number to another number results in their difference. Therefore, in order to subtract, all bits of B have to be inverted and one added to it. In order to switch between addition and subtraction, a new signal S is introduced.

For this circuit, every bit of B is connected to a XOR gate with the other input connected to S. The XOR gates' outputs now toggle between inverted B if S is 1 and not inverted B if S is 0. The outputs of the XOR gates are connected to the second input of the full adder array. The only missing piece is the addition of 1. This can easily be accomplished by feeding the S signal into the carry-in of the first full adder, hence adding 1.

3 Architecture

A computer consists of different units, each of them with a specific set of functions. These functions may be adding numbers, storing data or controlling other parts. What makes them into a computer is that all these units work together in a well ordered manner. They do this by passing data, respectively numbers, between them. The units may modify the number before passing it on, pass on another number based on the original number or not pass any new number. The exchange of data has to be controlled in some manner. This achieved with a dedicated control unit, which follows the instructions given by the program code. All of this put together forms a simple computer capable of performing automated calculations.

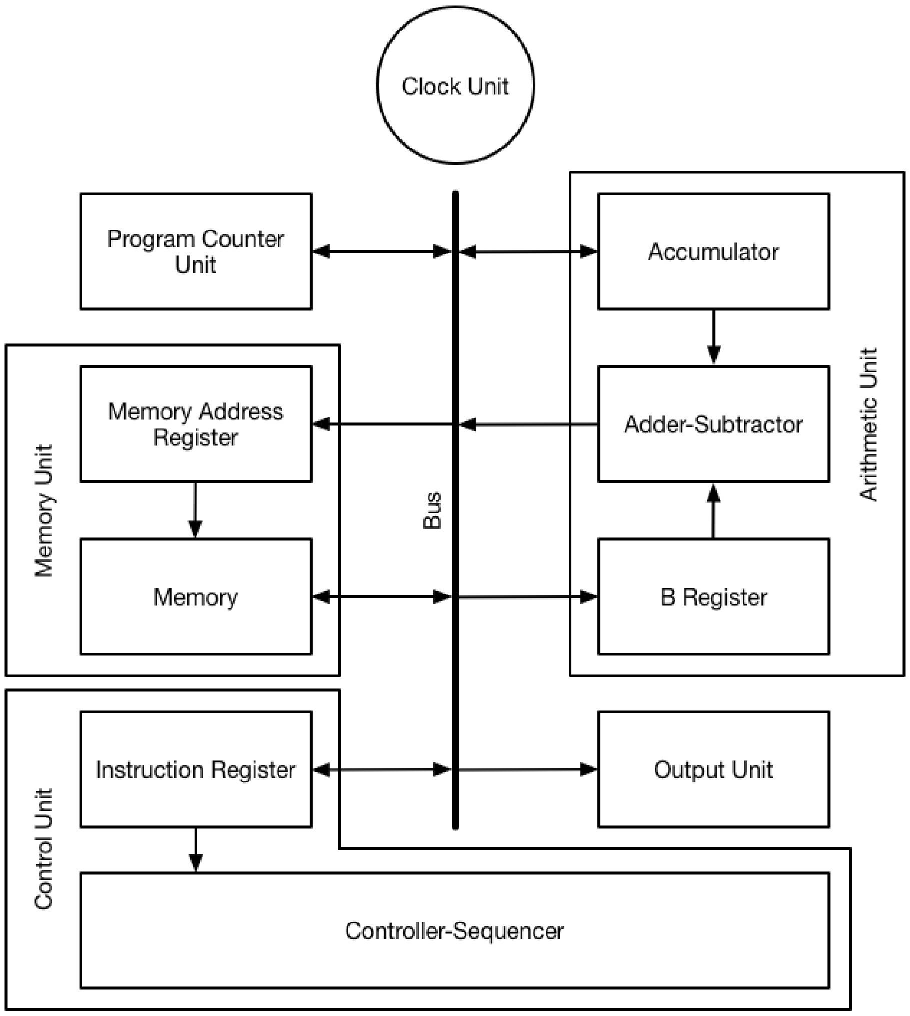


Figure 4: Block diagram of the architecture. The arrows show possible data movements. Additional signals are not depicted

3.1 Units

This section explains the functions of the different parts making up this computer architecture. The exact build-up of all these elements is too complicated to be described here yet a deeper understanding can be obtained by studying the schematics in the Appendix.

As already mentioned, a computer basically just moves data between different units. For this transfer there are two dedicated structures: The bus and the clock unit.

The **bus** is a set of wires to which all the units except the clock are connected. This set of common connections is used for bus transfer. In such a bus transfer one unit posts data onto the bus and another one loads the data from the bus, thus transfer data via the bus. As the bus in this architecture consists of 8 wires, it can transfer and therefore manipulate numbers with a maximal size of 8 bits. Since it is simply a set of wires and does not perform any actions, I omit calling it a unit.

The **clock unit** is an oscillator circuit, creating a clock signal which paces all operations of the computer and therefore synchronises the data transfer between units. When given the command, it inhibits this clock signal, thus halts all operations of the computer. The clock unit is different from all other units as it does not transfer any data.

When a unit loads data from the bus, it always stores this data before manipulating it. For storing data there are two structures: Registers and the memory.

Registers are the most common structure of this computer architecture and part of every unit but the clock. Their purpose is to store a series of bits, put differently, a number in binary. Since this is a 8-bit computer registers have a size of 8 bits unless otherwise noted.

Memory is a collection of registers. Every register, now called memory location, has an address through which it can be accessed. Memory is only used in the **memory unit** (MU), which is the unit that stores the program running on the computer. The memory unit consists of two parts, the aforementioned memory with 16 memory locations and a **memory address register** (MAR). The memory address register is 4 bits in size and stores the address of the currently accessed memory location.

A program consists of a series of instructions and followed by variables if needed. Every element of a program occupies one memory location, hence the memory of this computer can store programs with a maximum of 16 instructions and variables. An instruction is composed of 8-bits. The first four bits are the opcode, which stands for operation code and specifies what operation should be executed The last four bits are the operand, which stores additional information if needed for this operation. Variables may store numbers needed in the program or serve as storage location for interim results.

The execution of a simple program starts with its first instruction (referring to the instruction in the first memory location) and steps through one instruction after the other. In order to keep track of which instruction should be executed next, there is a **program counter unit** (PCU). The program counter unit stores the address of the first instruction and is incremented by one for the execution of every instruction. For some more complex programs, the address stored in the memory address register can be replaced with another one, hence allowing for simple loops and other forms of conditional branching.

The **arithmetic unit** (AU)[1] performs mathematical operations. This unit can add and subtract two numbers which have to be stored in two registers beforehand. These two registers are the **accumulator** (ACC) and the **B register** (BR), with the number stored in accumulator serving as augend or minuend and the number stored in the B register serving as addend or subtrahend. The numbers stored in these two registers are fed into the adder-subtractor which performs the calculation. Its result is then stored in the accumulator. Storing the result in the accumulator instead of a separate register simplifies subsequent calculations as the result does not have to be explicitly loaded into the accumulator for the next calculation to take place. The arithmetic unit can also detect when a calculation results in a overflow. Together with the PCU, this detection is used for the aforementioned manipulation of the program's execution flow. The actual build-up of the adder-subtractor is based on the circuit described in the section *Addition and Subtraction in a Digital Circuit.*

The most complex unit is the **control unit** which, as the name implies, controls the entire computer. This control unit can be divided into two parts: an **instruction register** (IR) and a **controller-sequencer**. The instruction register stores an instruction from the memory unit, passes on the opcode to the controller-sequencer and posts the operand

[1]Normally called ALU for arithmetic logic unit but, as this computer only performs arithmetic operations, I omit the logic part

onto the bus when needed. The controller-sequencer decodes the opcode into a sequence of steps needed to execute the given instruction. Every step is accompanied by a control word, representing is a series of control signals that specify what parts of the computer have to perform what action. These actions are usually one unit writing data onto the bus and another unit reading data from the bus.

In order to present results to the operator, the computer has an **output unit** (OU). It consists of a register, which stores the output, and a display capable of displaying positive decimal numbers.

3.2 Operation

A program in its simplest form is a list of instructions telling a computer what to do. The computer then executes one instruction after the other. In this architecture every execution of an instruction consists of six steps called microsteps. The execution of the individual microsteps is in response to the clock signal.

The first three microsteps are the same for every instruction and comprise the **fetch cycle** in which the control units loads the current instruction. The actual instruction is executed in the last three microsteps, forming the **execution cycle**. These microsteps are the steps controlled by the controller-sequencer mentioned in the section *Units*.

The fetch cycle's microsteps are as follows: In the first microstep, the program counter posts the address of the current instruction onto the bus which is then fetched by the memory address register. In the second microstep, the memory unit posts the instruction stored at that address, and the instruction register fetches it. The controller-sequencer then decodes this instruction into the three microsteps assigned to its execution cycle. Before these will be executed, the third microstep advances the program counter by one.

As the execution cycle for every command is different, only the execution of the LDA instruction, which loads a variable from memory into the accumulator, will be discussed here. The operand of this instruction is a memory address, more precisely the memory address of the variable that should be loaded. In the fourth microstep, the first microstep of the execution cycle, the instruction register posts operand onto the bus, and the memory address register reads it. Then, in the next microstep, the memory unit posts the variable's value, which is fetched by the accumulator. After these five microsteps all necessary operations are completed and no further operation takes place in the sixth microstep.

After the execution cycle is finished, the fetch cycle for the the next instruction begins.

Just looking at one instruction does not show the operation of the computer very well. For this reason, we will look at a program which adds the numbers 12 and 143 and displays the result.

The program is depicted in table 5 and concisely explained underneath. The left half of the table shows the program in mnemonic form whereas the right half is the actual machine code. A table of all available instructions (Table 6) can be found in the next subsection *Capabilities*.

Address	Instruction	Address	Instruction
0	LDA [4]	0000	0000 0100
1	ADD [5]	0001	0010 0101
2	OUTA	0010	1001 0000
3	HLT	0011	1100 0000
4	12	0100	0000 1100
5	143	0101	1000 1111

Table 5: Program for adding two numbers and displaying the result

LDA[4]	Memory location 4 is accessed and its data (12) is loaded into the accumulator.
ADD[5]	Memory location 5 is accessed and its data (143) is loaded into the B register. The Adder-Subtractor adds the two numbers and stores the result in the accumulator.
OUT	The content of the accumulator is loaded into the output unit and presented on its display.
HLT	The HLT signal inhibits the clock signal and stops the computer.

The Python function below should serve as reference to the machine code. The *print()* function represents the OUT instruction.

```python
def addition():
    var x = 12
    var y = 143
    print(x + y)
```

Code 1: Addition example written in Python

The following pages will step through the program using pictures of the computer executing this calculation and drawings where arrows show the data transfers. The arrangement of the units in the drawing is similar to the actual computer, with the clock unit being located above the program counter unit. The instruction register in the actual computer displays only the operand and the drawing of the controller-sequencer shows the microsteps, counting form left to right and 0 being the current one. Its important to note, that the actual data transfers happen in response the clock signal, in other words, between the images.

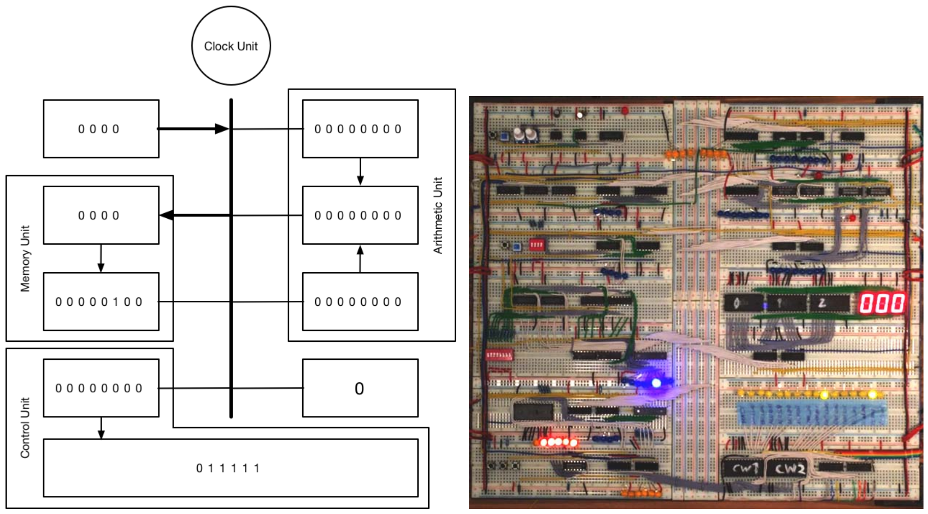


Figure 5: LDA[4] Microstep 1: PCU transfers current instruction address to MAR

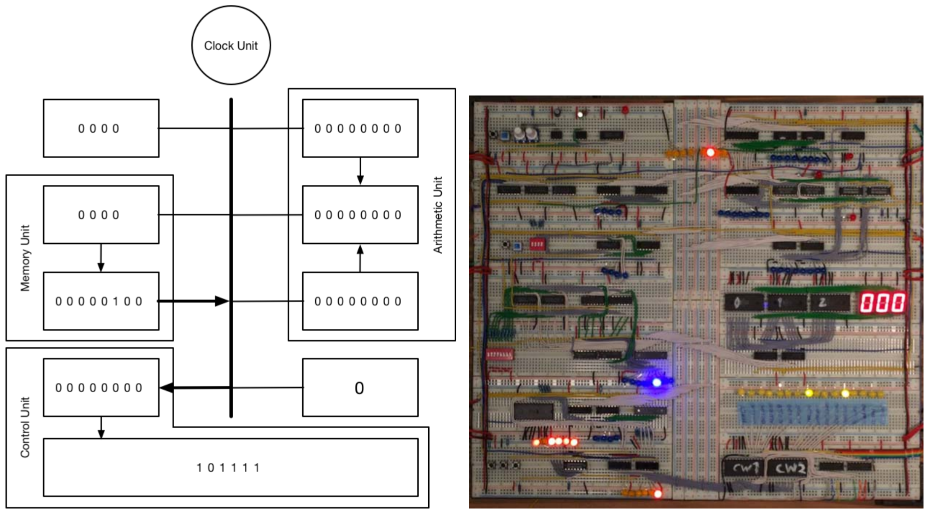


Figure 6: LDA[4] Microstep 2: MU transfers instruction to IR

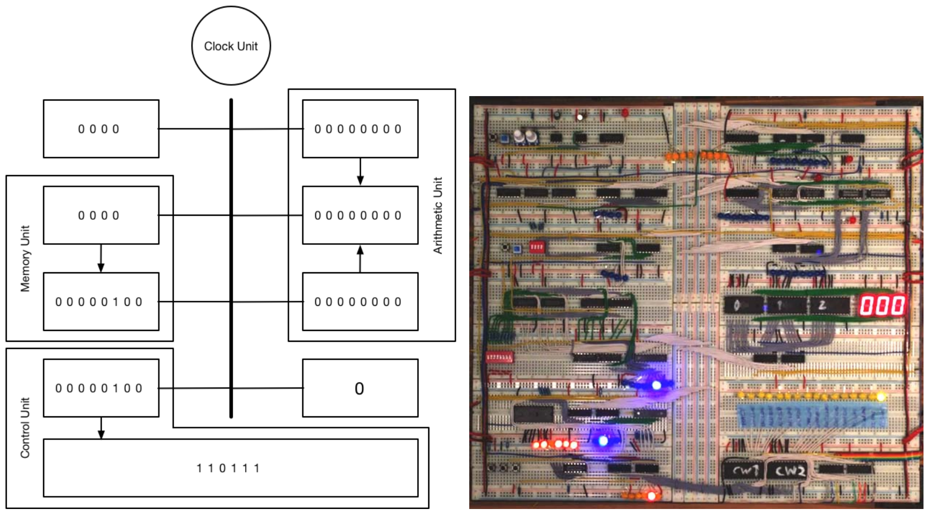


Figure 7: LDA[4] Microstep 3: PCU advances by one

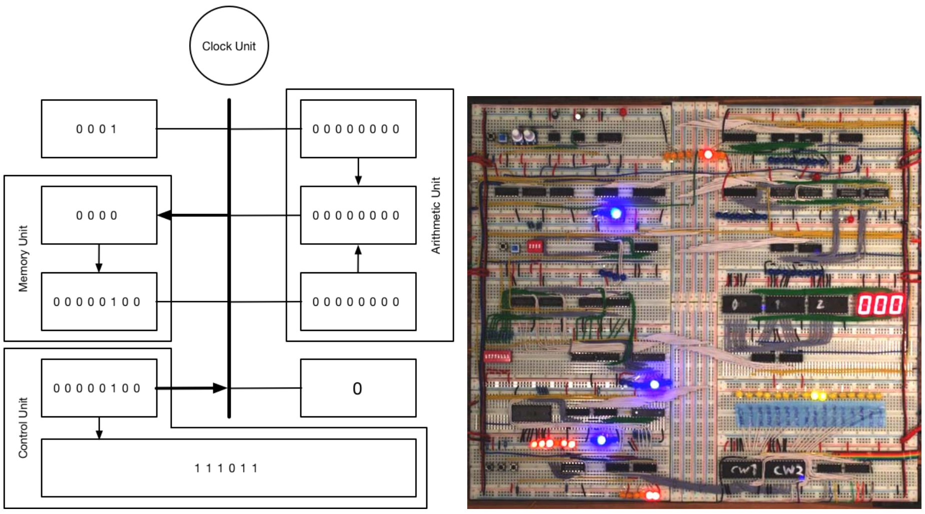


Figure 8: LDA[4] Microstep 4: IR transfers operand (address) to MAR

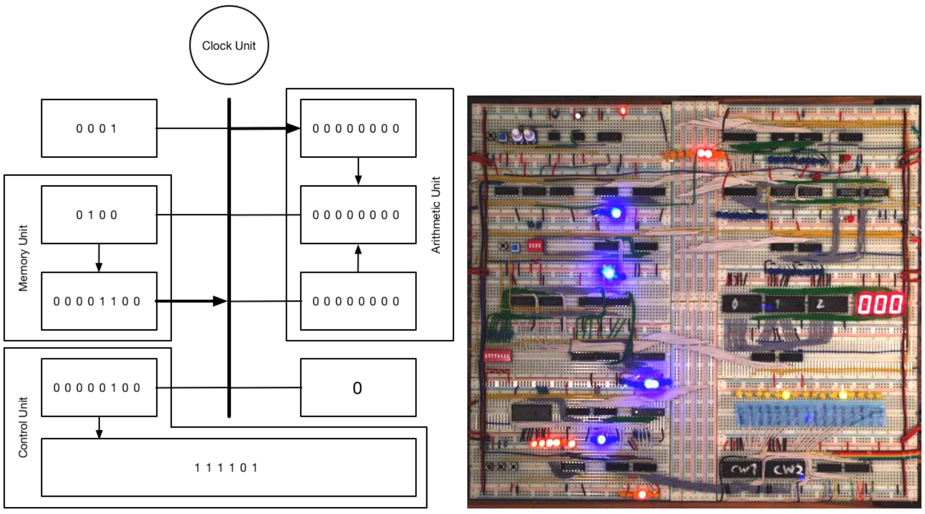


Figure 9: LDA[4] Microstep 5: MU transfers number to AAC

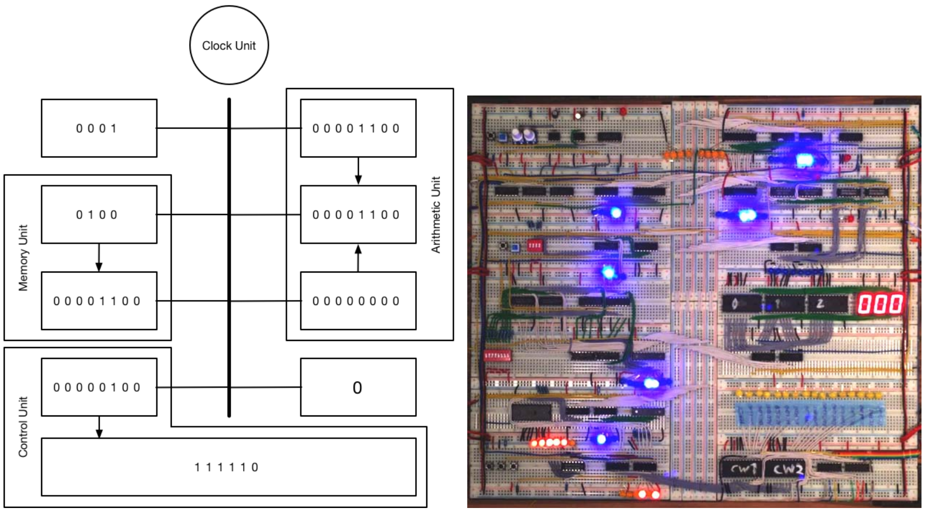


Figure 10: LDA[4] Microstep 6: No operation

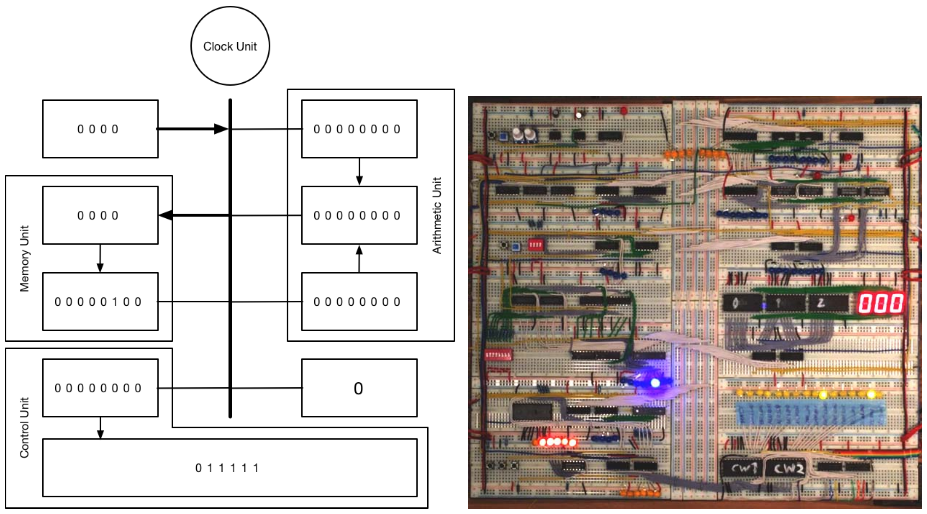


Figure 11: ADD[5] Microstep 1: PCU transfers current instruction address to MAR

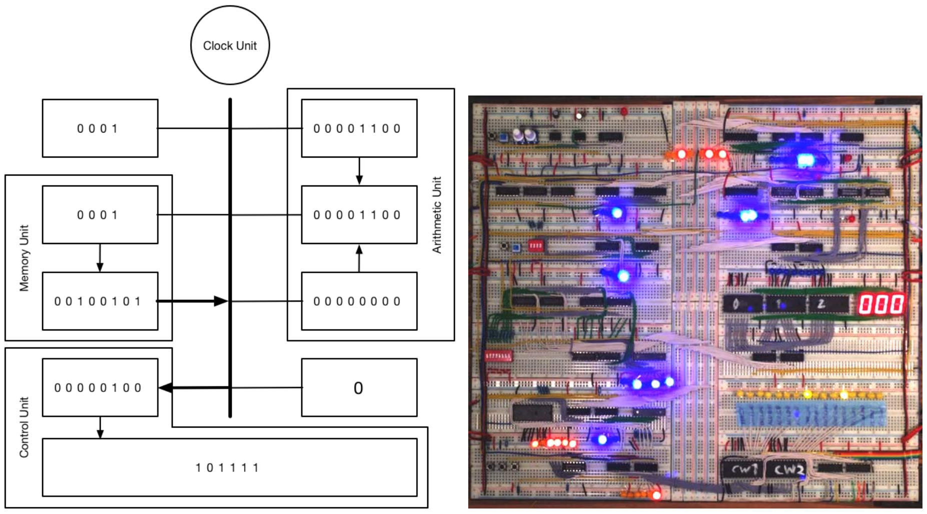


Figure 12: ADD[5] Microstep 2: MU transfers instruction to IR

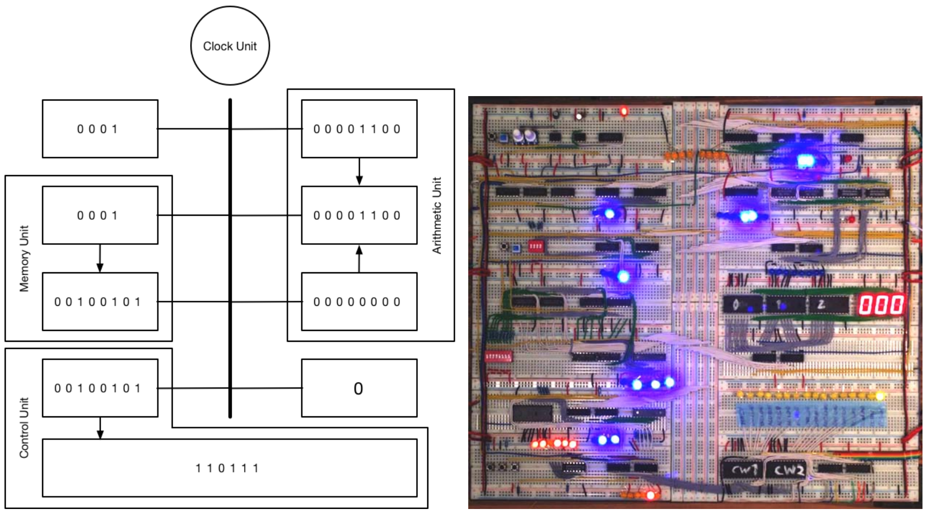


Figure 13: ADD[5] Microstep 3: PCU advances by one

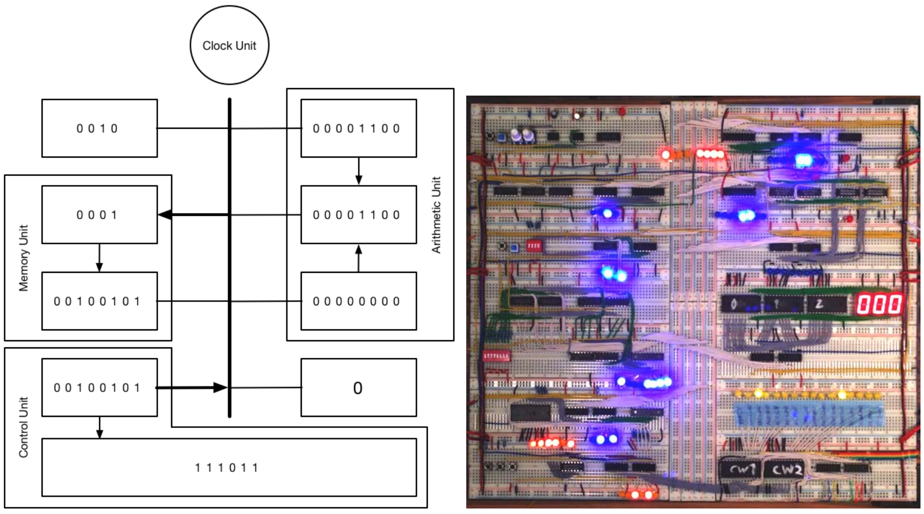


Figure 14: ADD[5] Microstep 4: IR transfers operand (address) to MAR

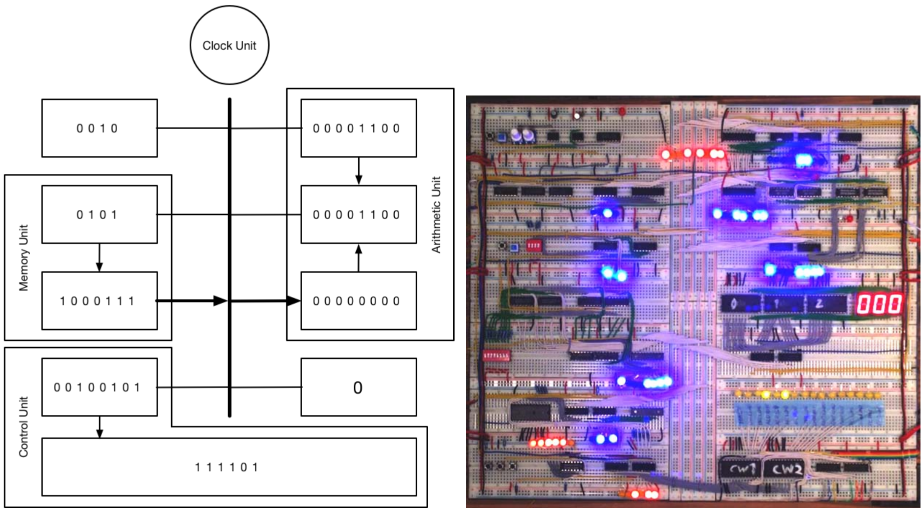


Figure 15: ADD[5] Microstep 5: MU transfers number to BR

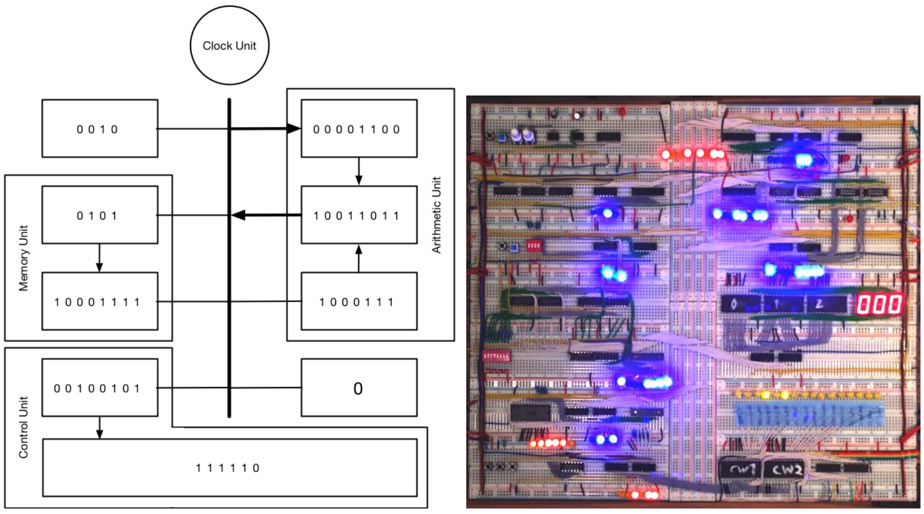


Figure 16: ADD[5] Microstep 6: AU transfers result to ACC

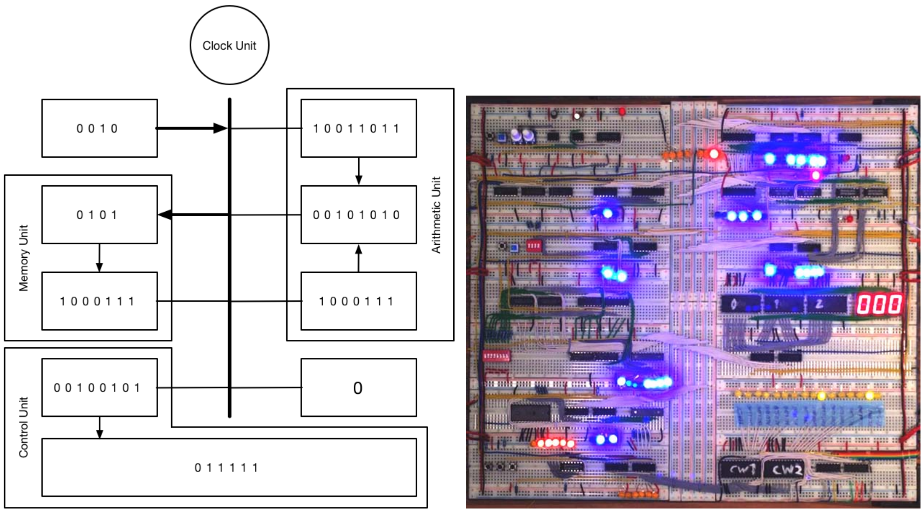


Figure 17: OUTA Microstep 1: PCU transfers current instruction address to MAR

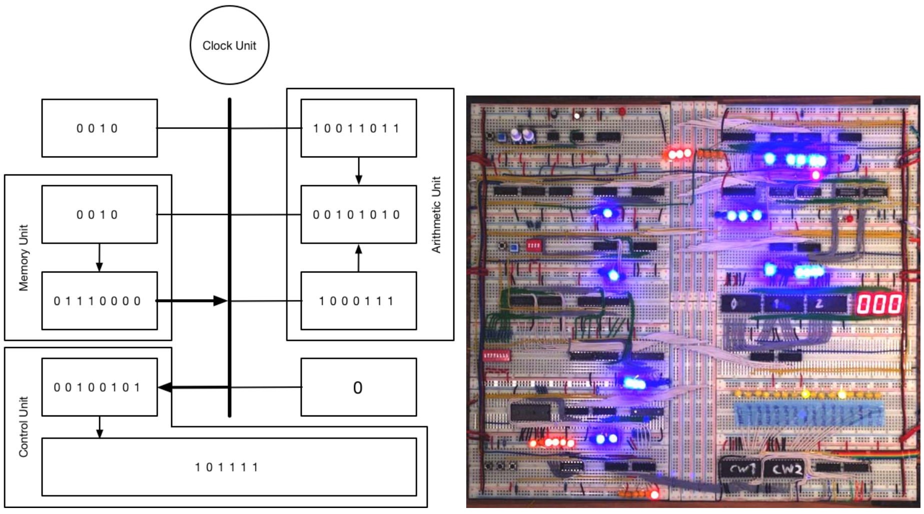


Figure 18: OUTA Microstep 2: MU transfers instruction to IR

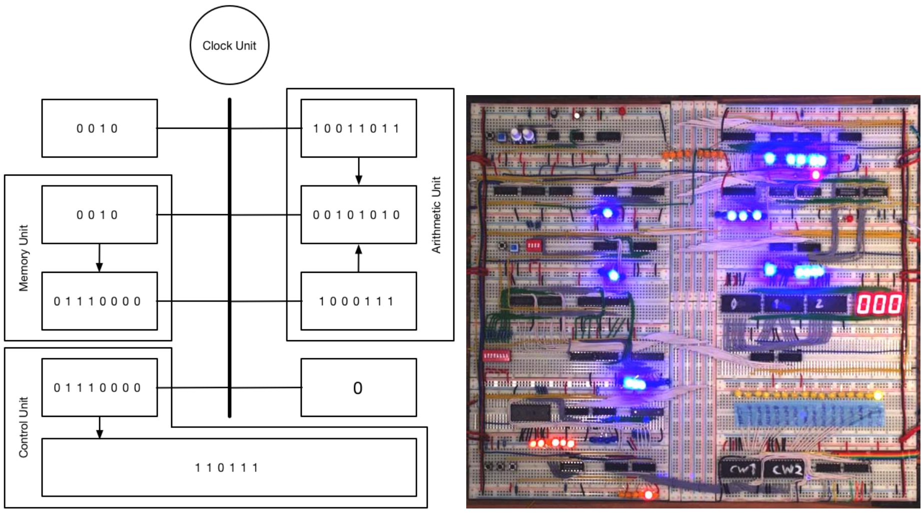


Figure 19: OUTA Microstep 3: PC advances by one

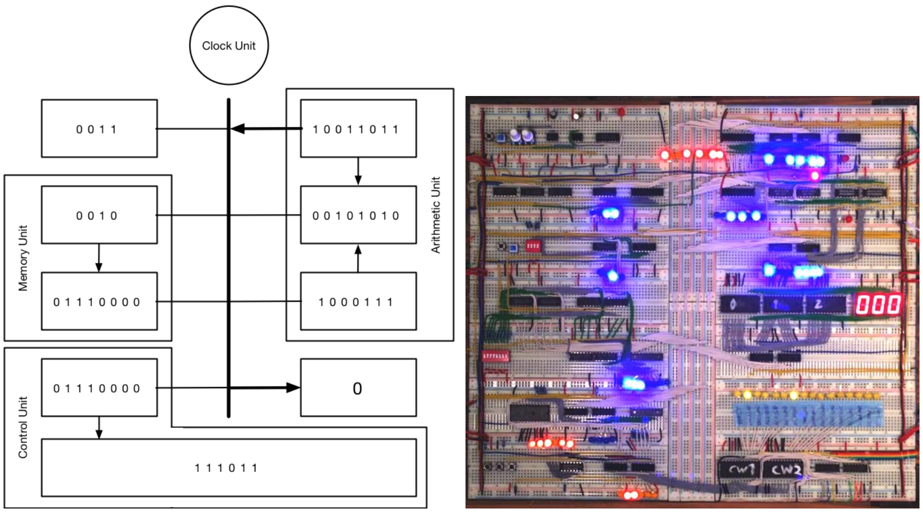


Figure 20: OUTA Microstep 4: ACC transfers result to OU

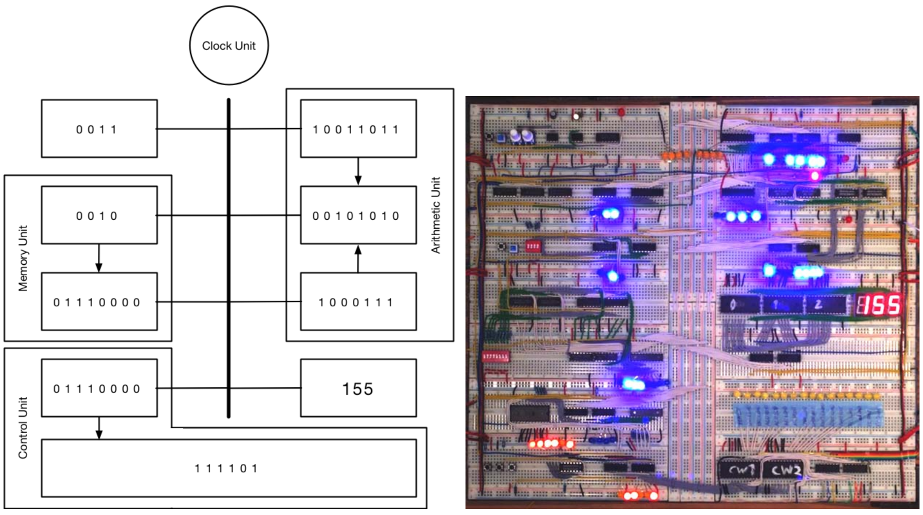


Figure 21: OUTA Microstep 5: No operation

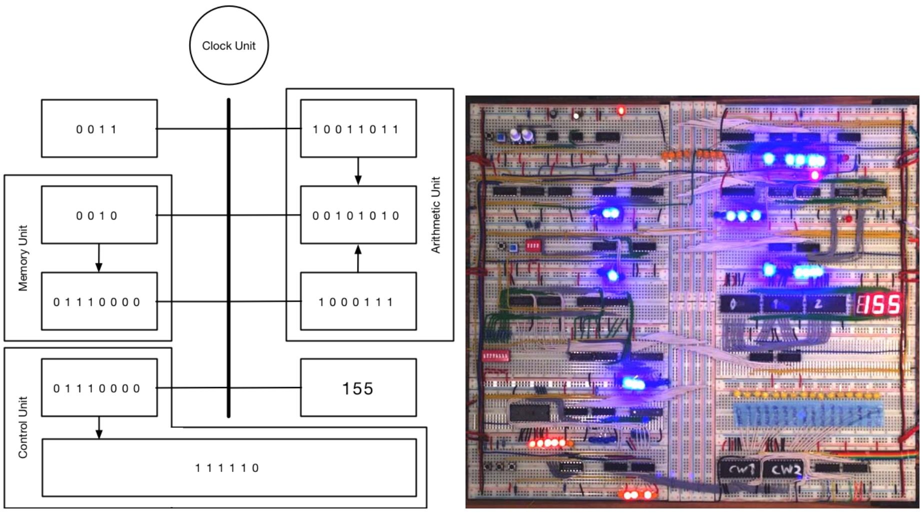


Figure 22: LDA[4] Microstep 6: No operation

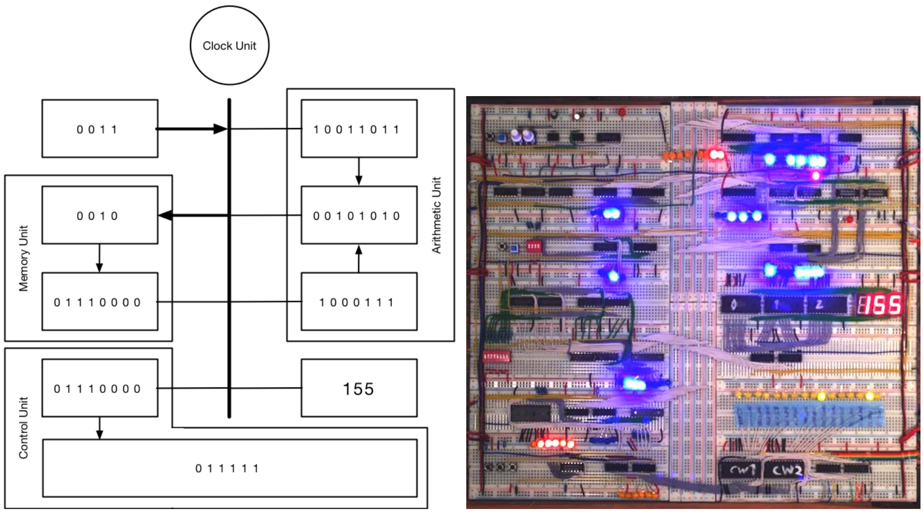


Figure 23: HLT Microstep 1: PCU transfers current instruction address to MAR

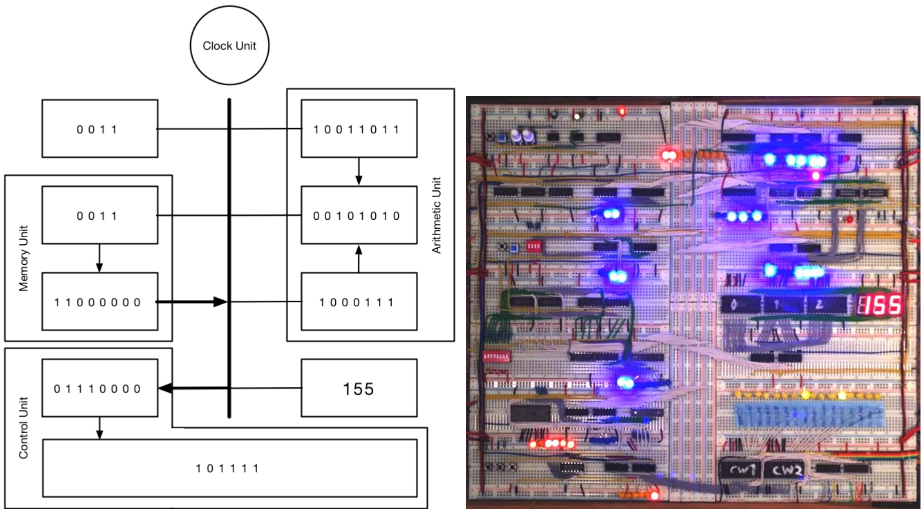


Figure 24: HLT Microstep 2: MU transfers instruction to IR

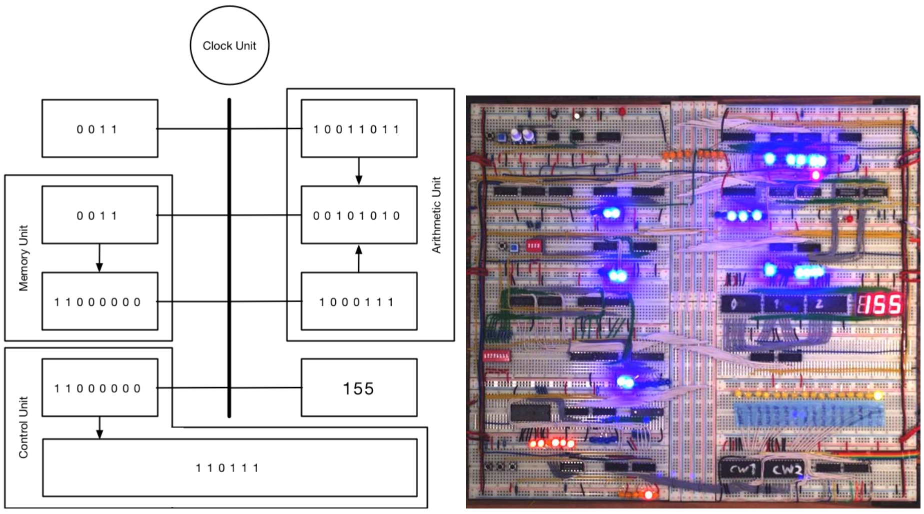


Figure 25: HLT Microstep 3: PC advances by one

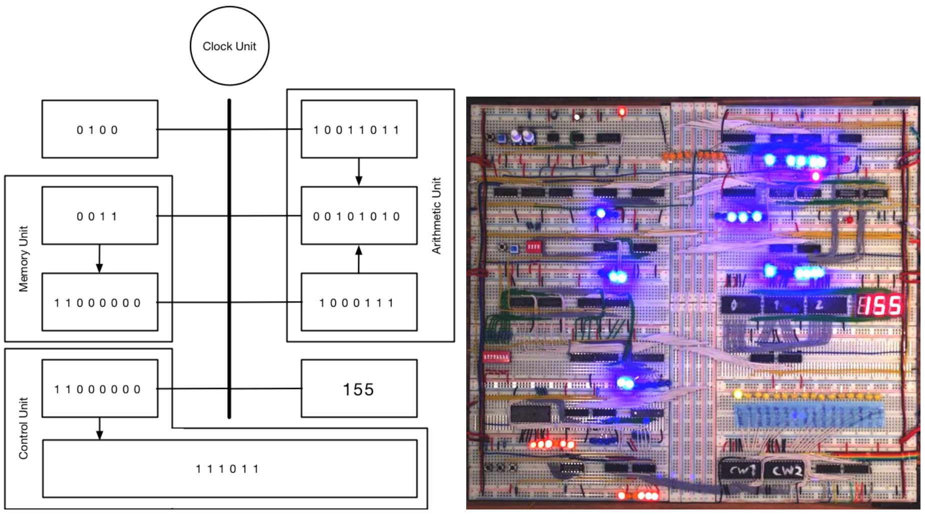


Figure 26: HLT Microstep 4: Clock halts all operations. Program has finished

3.3 Capabilities

The already given explanations may give insight into how this computer architecture works but hardly demonstrates its capabilities. This computer is capable of performing automated calculations comprised of additions and subtractions. Such a calculation has to be programmed into the computer's memory using the instructions denoted in its instruction set (see Table 6). The number of instructions and variables cannot exceed 16 since that is the maximal memory capacity.

Mnemonic	Opcode	Operand	Explanation
LDA	0000	Variable Address	Load variable into ACC (accumulator)
LDAC	0001	4-bit constant	Load 4-bit constant into ACC
ADD	0010	Variable Address	Add variable to ACC data and store result in ACC
ADDC	0011	4-bit constant	Add constant and store result in ACC
SUB	0100	Variable Address	Subtract variable and store result in ACC
SUBC	0101	4-bit constant	Subtract 4-bit constant and store result in ACC
STR	0110	Variable Address	Store number in variable
OUTA	0111	-	Output number from ACC
OUTC	1000	4-bit constant	Output 4-bit constant
OUT	1001	Variable Address	Output variable
JMP	1010	Program Position	Jump to position in program
JMPO	1011	Program Position	Jump to position if overflow occurred
HLT	1100	-	Halt all operations

Table 6: Instruction set

Many instructions have different forms where one form uses a variable and another uses a constant or similar. This can make programs more compact and allows for more complex programs. Such complex programs could calculate a sequence and restart the sequence calculation when the calculated elements exceed the 8-bit limit.

For example, the calculation of the fibonacci sequence [3].

Address	Intruction/Data	Address	Instruction/Data
0	LDAC [1]	0000	0001 0001
1	STR [12]	0001	0110 1100
2	LDAC [0]	0010	0001 0000
3	OUTA	0011	0111 0000
4	ADD [12]	0100	0010 1100
5	STR [13]	0101	0110 1101
6	LDA [12]	0110	0000 1100
7	STR [14]	0111	0110 1110
8	LDA [13]	1000	0000 1101
9	STR [12]	1001	0110 1100
10	JMPO [0]	1010	1011 0000
11	JMP [3]	1011	1001 0011
12	0	1100	0000 0000
13	0	1101	0000 0000
14	0	1110	0000 0000

Table 7: Fibonacci sequence calculation.
Last three positions used as variables

The Python function (Code 1) is similar and should serve as a reference for the understanding program in table 7. The *print()* function once again represents the OUT instruction. The two programs are not the same because the loading of variable y is optimised in the program for the 8-bit computer.

```
def fibonacci():
    var x, y, z
    while(true):
    x = 0
    y = 1
    while(x < 255):
        print(x)
        z = x + y
        x = y
        y = z
```

Code 2: Fibonacci sequence example written in Python

The example program in table 7 uses constants to initialise a loop. This loop then calculates the individual elements. When the elements overflow, the calculation restarts. The SAP-1 architecture, which this architecture is based on, could not run such a program as its instruction set only consists of LDA, ADD, SUB, OUT, HLT. Therefore the SAP-1 architecture can only perform extremely limited calculations without any loops, conditional branching, storing of interim results or constants.

4 Advice on Building a 8-bit Computer

In case you are interested in building an 8-bit computer there are two ways to tackle this project. The first is to design all the circuits yourself and the second is to use the plans in the Appendix.

Regarding the first option, the most important thing to do is to obtain a good overview of the entire project. Then, it is all about working out how to build the computer. Due to its complexity it is advisable to split the computer up into different pieces and instead of trying to figure out everything at once. The focus should always be exclusively on one single piece The workflow on each piece is always the same: Plan, build, test and troubleshoot. For more complicated parts, I would even advise to build subsections of it to make sure they work before planning the entire part.

As soon as multiple parts work correctly, they can be connected with each other to make sure they also work together. As the computer starts to take form, you will maybe realise that you have a critical design flaw. If this happens, do not rip the unit apart and rebuild it right way but rather stop working on it for a while since it might not be a flaw of your design but rather in your thinking at the moment. I ran into such situations numerous times and then quite quickly redid that piece and then eventually had to change it back again.

If one follows the plans given in the Appendix the procedure is practically the same but just without the designing of the computer. It is still advisable to focus on smaller parts first and testing them before building another one.

A problem you will most certainly run into are faulty connections and the thing is there will always be some loose connections as breadboards simply do not provide good connections. I found that the most efficient method to correct this is to move some of the wires ever so slightly and then replace the ones which proofed to be faulty.

5 Conclusion

In the conclusion I'll focus on the physical part of this project meaning the planing and building of the computer. Planning the computer filled me with great pleasure as I enjoy tackling problems of great complexity. One aspect that I liked particularly was to derive how the different elements should work, especially the control unit as it is the most complex.

Building the actual elements needed more concentration as I expected. The first few test parts looked terrible with connections all over the place so I quickly devised some layout guidelines and became quite efficient at actually making the connection. Still, spending hours connecting different integrated circuits with dozens of wires is a very cumbersome process. And, as a newly built part generally malfunctions during testing or sometimes does not work at all, this can create quite some frustration. This usually meant a few more hours or entire afternoons probing the circuits with a multimeter, trying to find the mistake(s).

Creating the schematic of the circuits turned out to be very time consuming. Nevertheless, I consider this part to be necessary since I am planning on building a second version of this 8-bit computer using real, homemade circuit boards instead of breadboards. In addition, I would also like to build a version with multiple buses rather than just one for instructions, addresses and data. This would allow for a bigger addresses and therefore longer, more complex programs. Another improvement would be to build an external device to access the memory of the computer. This device could then store a couple of programs for the computer and be used to program it automatically. Programming is currently done by hand using small switches, which is prone to errors. Automating this process would eliminate a great source of problems.

Even though there is still room for improvement, I'm very happy with the result of this project. Actually seeing Alan, this is what I call my computer, perform calculations mesmerises me. I gave my computer this codename in honour of Alan Turing, the father of computer science and artificial intelligence, an ingenious mind behind Britains codebreaking endeavours during the Second World War and a victim of the persecution of homosexuals which led to his tragic suicide.

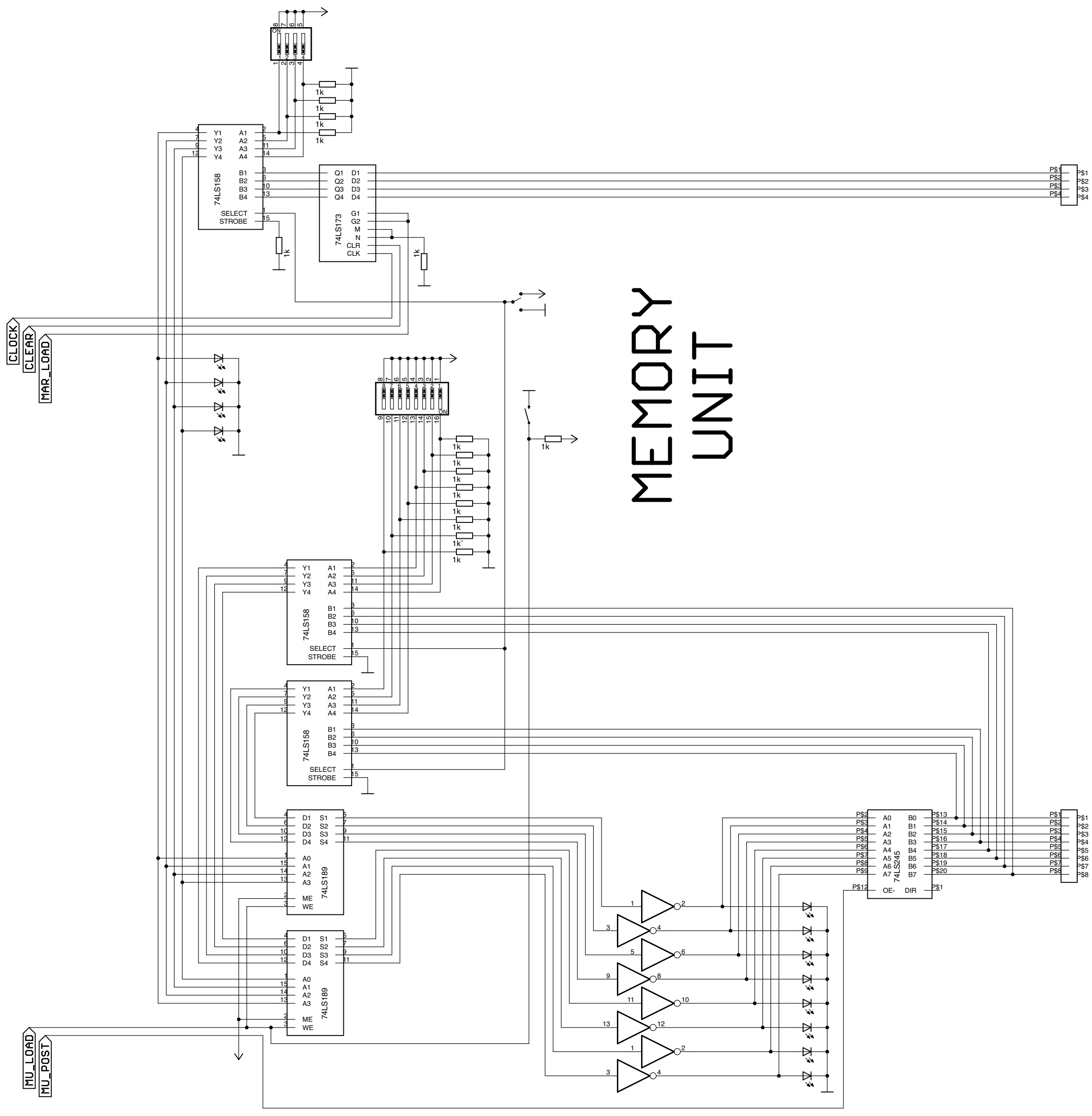
6 Appendix

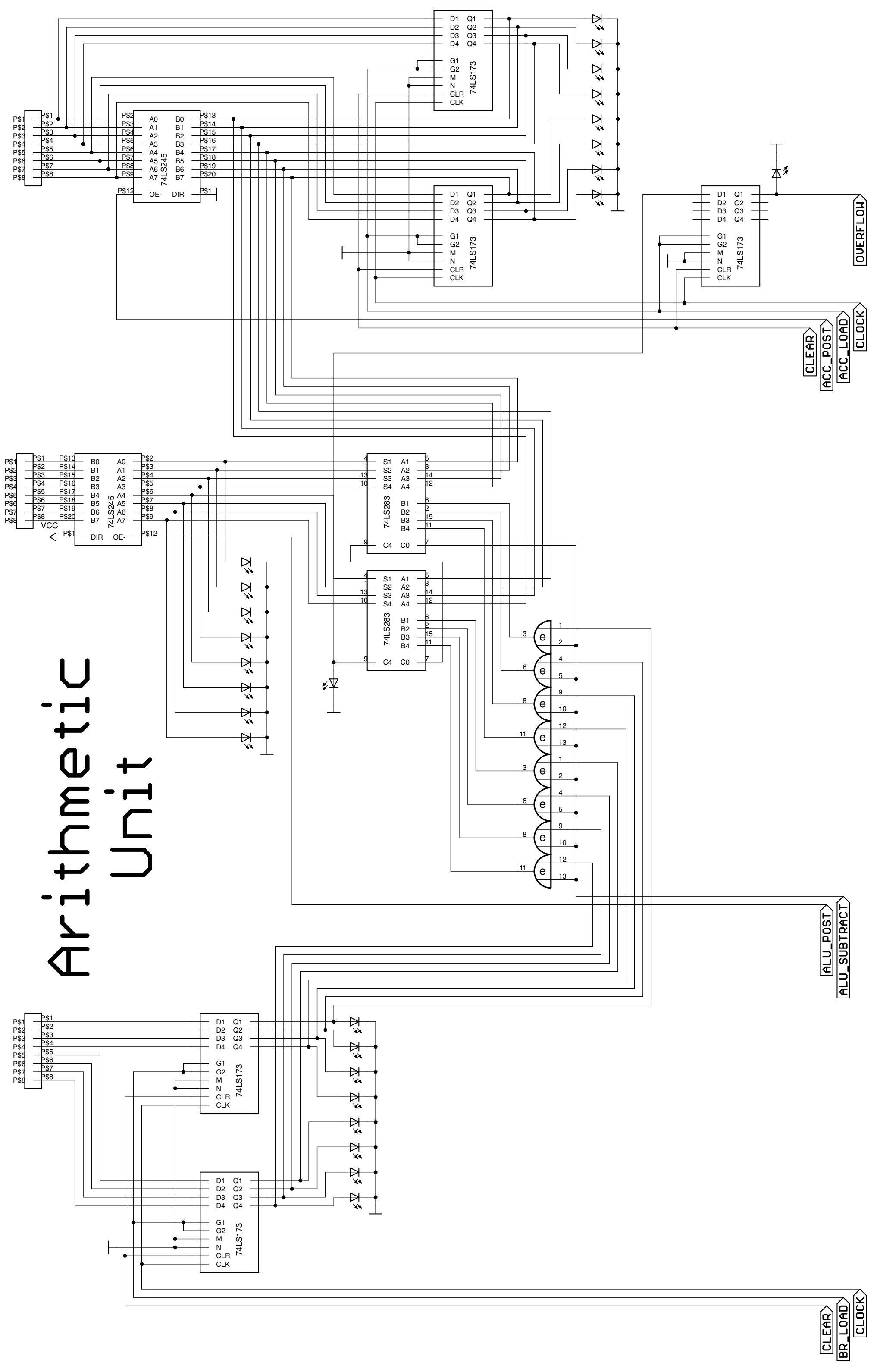
This Appendix contains all necessary schematics. The given information is not explained in any way. Due to their size many schematics cannot be depicted very well. For this reason, all the schematics are available online underr

`https://github.com/komplexon3/Alan`

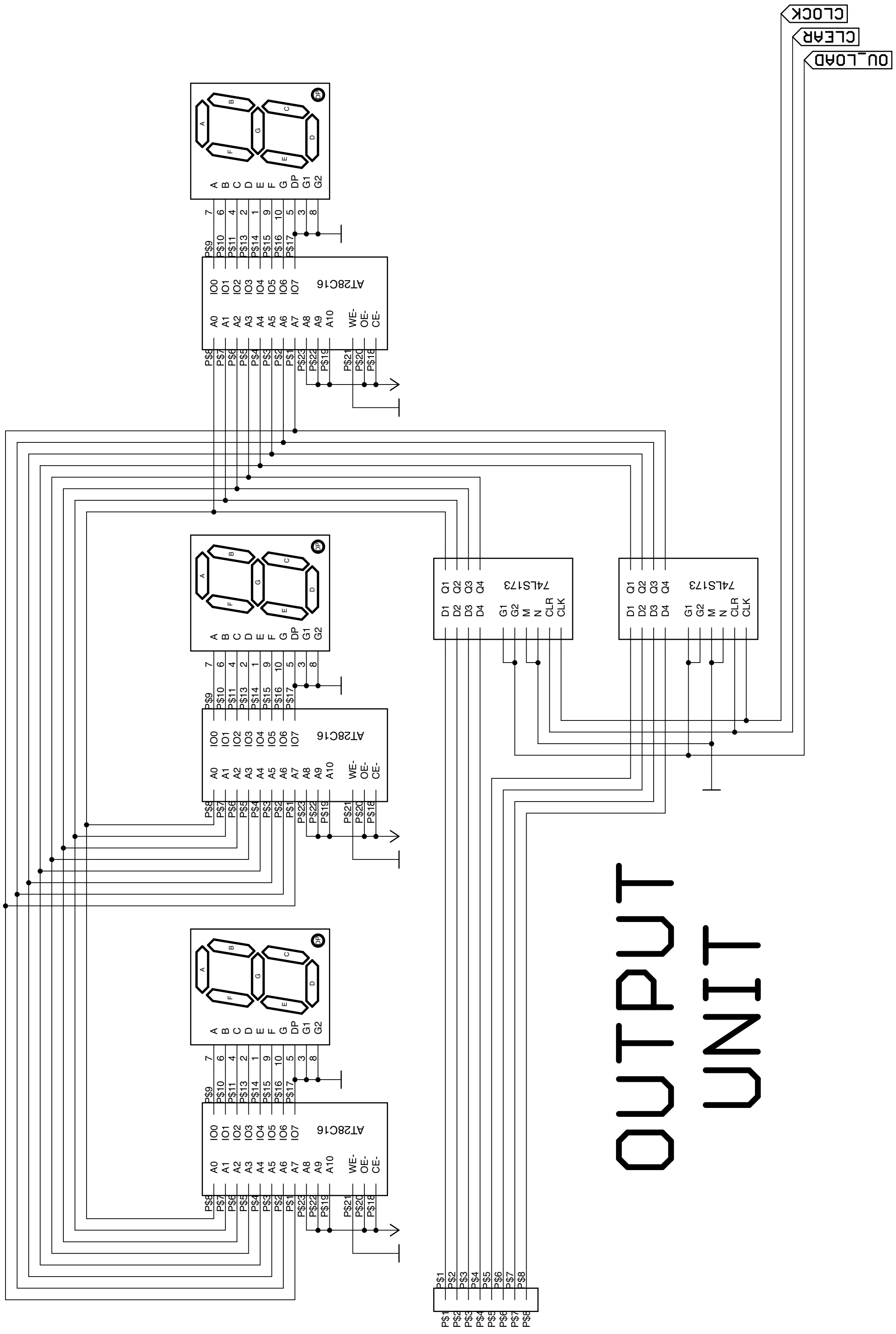
Mobile devices may scan the following QR code.

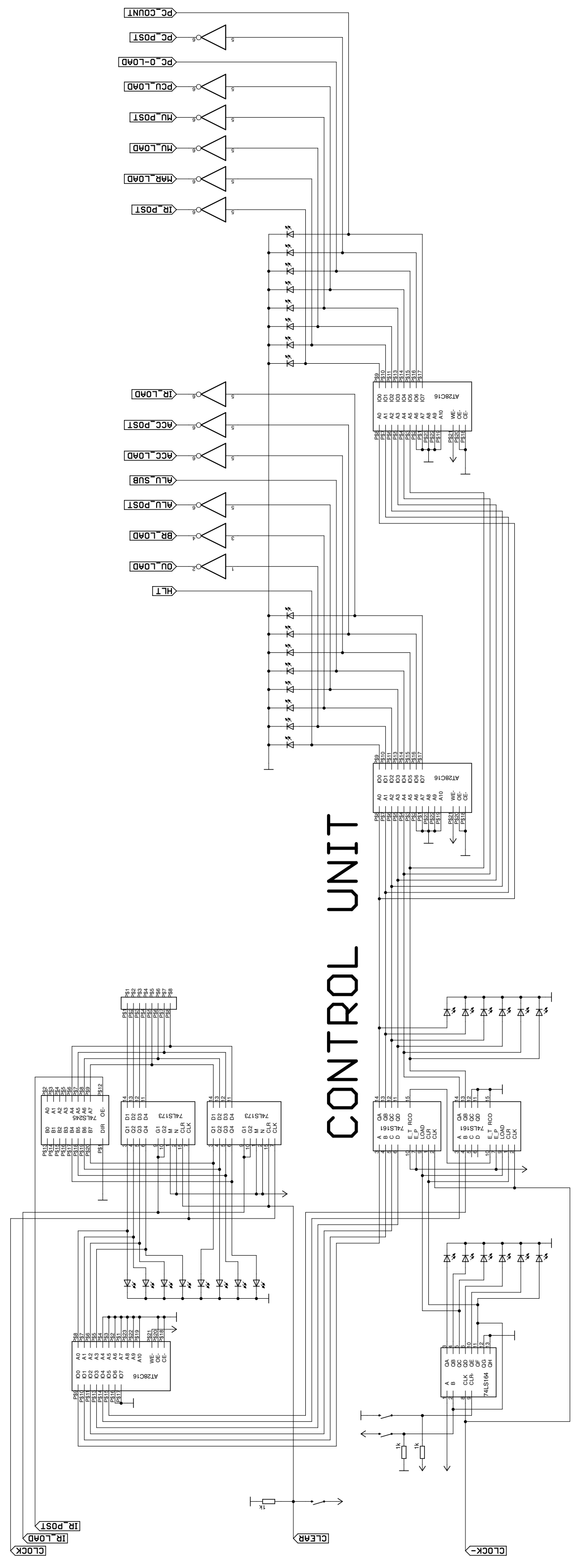
6.1 Schematics





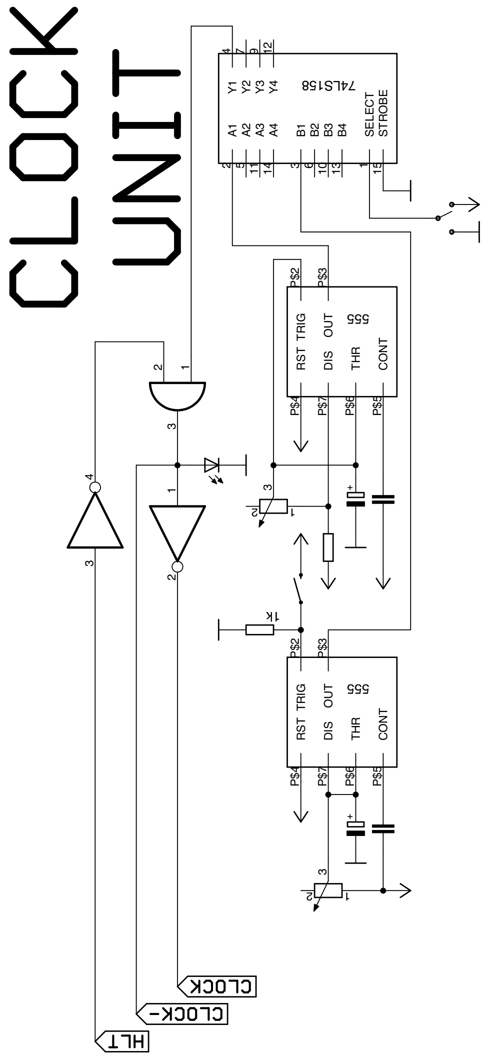
Arithmetic Unit
74LS173
74LS245
74LS283
VCC
OVERFLOW
CLEAR
ACC_POST
ACC_LOAD
CLOCK
ALU_POST
ALU_SUBTRACT
CLEAR
BR_LOAD
CLOCK

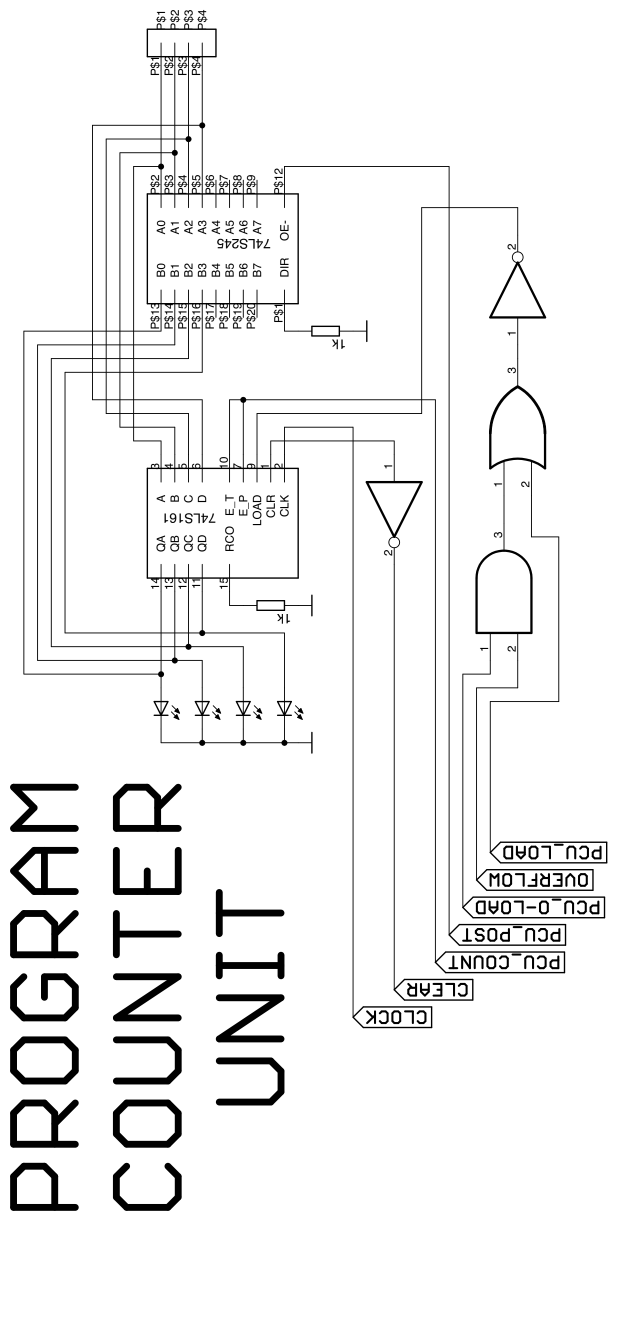

OUTPUT UNIT
CLOCK
CLEAR
OU_LOAD
AT28C16
AT28C16
AT28C16
74LS173
74LS173

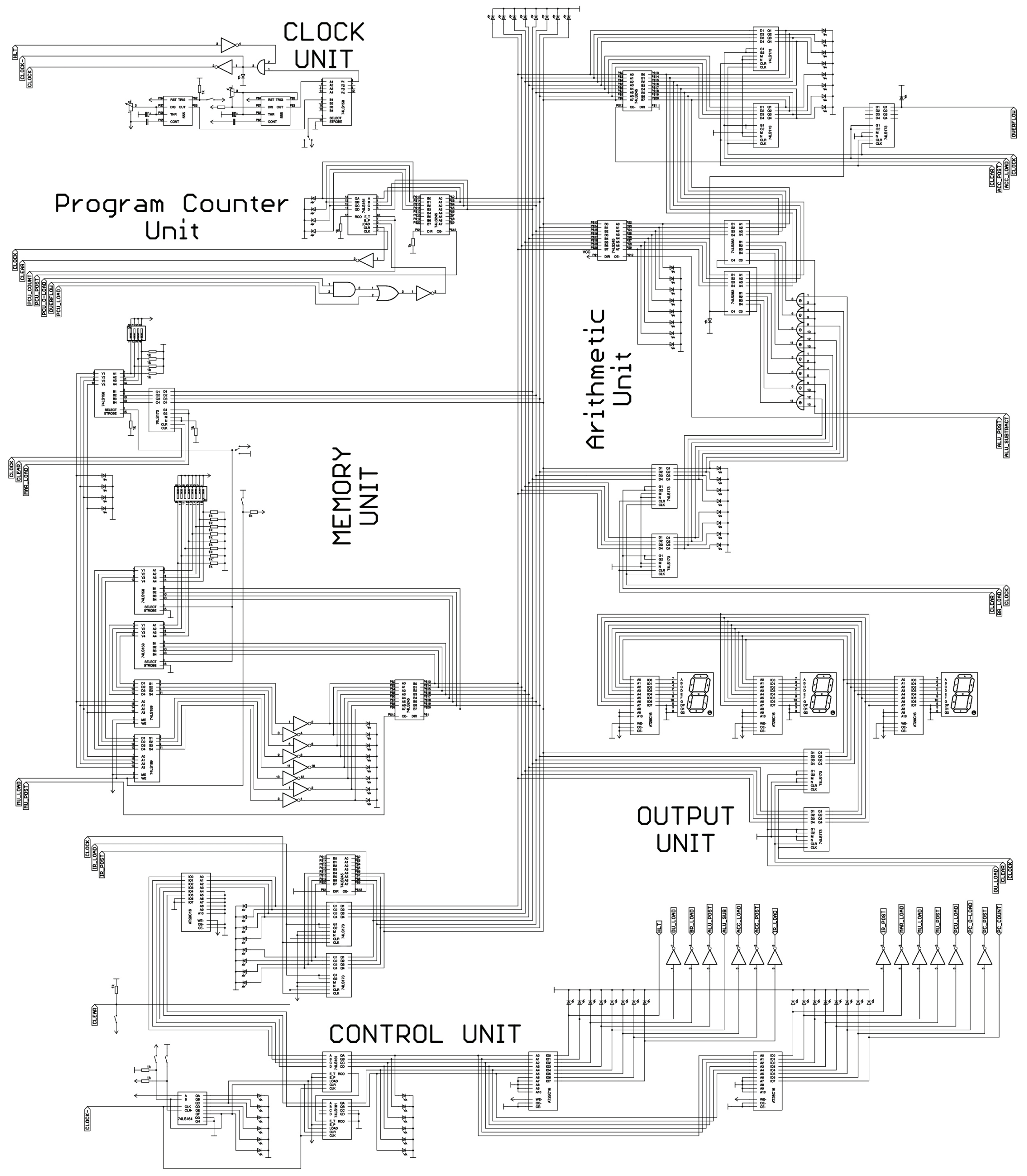


CONTROL UNIT
PC_COUNT
PC_POST
PC_O-LOAD
PCU_LOAD
MU_POST
MU_LOAD
MAR_LOAD
IR_POST
IR_LOAD
ACC_POST
ACC_LOAD
ALU_SUB
ALU_POST
BR_LOAD
OU_LOAD
HLT
AT28C16
74LS161
74LS164
74LS173
74LS245
IR_POST
IR_LOAD
CLOCK
CLEAR
CLOCK-

CLOCK UNIT


74LS158
A1
A2
A3
A4
B1
B2
B3
B4
Y1
Y2
Y3
Y4
SELECT
STROBE
555
RST
TRIG
DIS
OUT
THR
CONT
P$2
P$3
P$4
P$5
P$6
P$7
1K
CLOCK
CLOCK-
HLT


PROGRAM
COUNTER
UNIT
74LS245
A0 A1 A2 A3 A4 A5 A6 A7
B0 B1 B2 B3 B4 B5 B6 B7
DIR
OE-
74LS161
A B C D
QA QB QC QD
RCO
E_T
E_P
LOAD
CLR
CLK
1K
1K
CLOCK
CLEAR
PCU_COUNT
PCU_POST
PCU_0-LOAD
OVERFLOW
PCU_LOAD


CLOCK UNIT
Program Counter Unit
MEMORY UNIT
Arithmetic Unit
OUTPUT UNIT
CONTROL UNIT

	Output Register	A Register		B Register	Command Register		RAM			Program Counter				ALU		Clock
	Load	Load	Out	Load	Load	Out	aLoad	Load	Out	Load	oLOAD	Out	Count	Out	Sub	Stop
Start	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Load	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Load Const.	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Add	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Add Const.	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sub	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
Sub Const.	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Store	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Jump	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Overflow Jump	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Stop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Output	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Output Var	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Output Const	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

References

[1] Ben Eater (12.12.2015). Programming my 8-bit Computer.

https://www.youtube.com/watch?v=9PPrrSyubG0&t=9

[2] Ben Eater (12.12.2015). Stepping through a program on the 8-bit breadboard computer.

https://www.youtube.com/watch?v=35zLnS3fXeA

[3] Ben Eater (12.12.2015). Programming Fibonacci on a breadboard computer.

https://www.youtube.com/watch?v=a73ZXDJtU48&t=5s

[4] Albert Paul Malvino, Jerald A. Brown (1999). Digital Computer Electronics.

[5] Harry Henderson (2008). Encyclopedia of computer science and technology.

[6] Christine R. Wright (14.11.2016). The Binary System.

http://www.math.grin.edu/~rebelsky/Courses/152/97F/Readings/student-binary

[7] (7.11.2016). Cambridge Igcse Computer Studies Revision Guide - Chapter 9: Logic gates.

http://education.cambridge.org/media/577240/cambridge_igcse_computer_studies__revision_guide___cambridge_education___cambridge_university_press_samples.pdf

Eigenständigkeitserklärung

Der Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbstständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

Marc Widmer	Ort & Datum