Platinencomputer - Langfassung

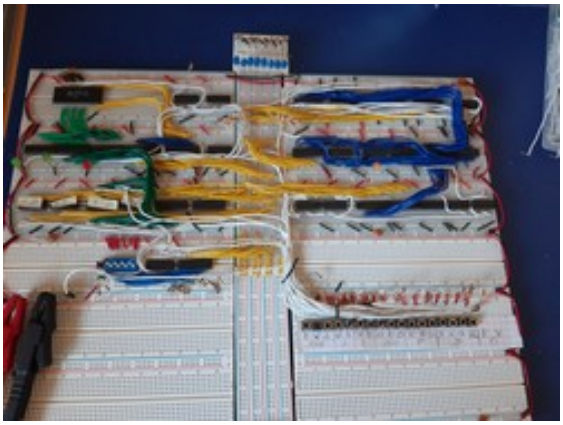
Alexander Wersching und Simon Walter

2022

Inhaltsverzeichnis

1 Idee

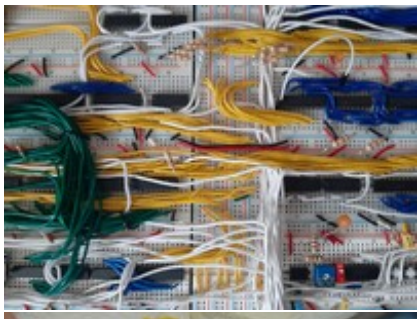
Vor ca. 2 Jahren hatte Alex die Idee, er wolle einen 8-bit Computer bauen, die vor 2 Jahren beim Regionalwettbewerb München-West eingereichte Version eines anderen Teilnehmers hat dann auch das Inter resse von mir (Simon) geweckt. Ein paar Monate später haben wir uns dann dazu entschieden, tatsächlich einen eigenen 8-bit-Computer zu bauen und haben auch bald angefangen erste Versuche auf dem Breadboard zu entwickeln. Es gab noch weder eine Möglichkeit diese erste Version automatisch zu betreiben, noch Daten zu schreiben/speichern, oder irgendwie mit der Umgebung zu interagieren, nur 3 Register, 2 Operationen und viele Knöpfe zum Bedienen.



2 Gebrauch von Platinen

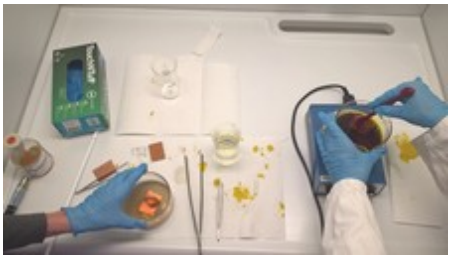
2.1 Weshalb Platinen?

Bald war aber klar, dass ein einfaches Verwenden von flachen Kabeln als Verbindung zwischen den Breadboards, das Kabelgewirr, das wir bei dem Projekt von vor 2 Jahren gesehen haben und nach Möglichkeit vermeiden wollten, um ein übersichtlicheres Design zu erreichen, nicht lösen kann.



Auf der Suche nach einer Alternative sind wir fast zwangsläufig auf Platinen gekommen, aber weil unser Computer ja nicht mehr wirklich selbst gemacht wäre, wenn wir diese nur professionell Ätzen lassen würden, wollten wir versuchen - zumindest einen Teil - selbst zu ätzen, auch wenn wir dadurch natürlich auf viele neue Probleme gestoßen sind...

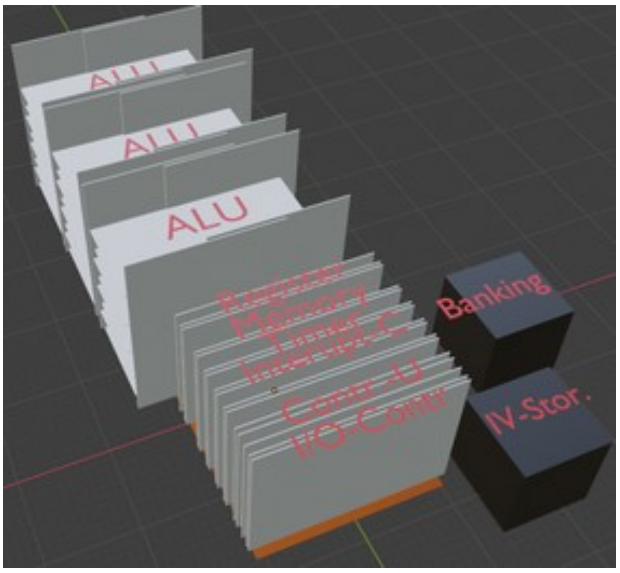
2.2 Erste Versuche der Umsetzung



Was zu Beginn das Projekt vorangetrieben hat, hat ab diesem Punkt aber natürlich ein Problem dargestellt:Das Homeschooling, bei dem wir natürlich nicht Ätzen können. Wir hätten es natürlich zu Hause versuchen können, hatten da aber keine adequate Ausrüstung v.a. bzgl. Sicherheit bis wir dann so weit waren erste Versuche zu machen war es dann schon Ende 2020.

2.3 Design

Nachdem es auch auf geätzten Platinen noch nicht automatisch ordentlich ist und eine derart große Platine für uns nicht machbar gewesen wäre, haben wir uns dann noch ein 3-Dimensionales Design ausgedacht.



Offensichtlich ist das mit unseren aktuell geplanten Modulen: am Boden sitzt eine große Bus-Platine, auf der wir alle Module senkrecht aufstecken.

3 Anforderungen V2

Im nächsten Lockdown hatten wir dann wieder viel Zeit zu planen, und der geplante Computer wurde immer komplexer.

3.1 Datenlänge

Aus den ursprünglich geplanten 8-bit wurden 16-bit Datenlänge:

Die Address- und Datenlänge besagt, wie viele binäre Ziffern (bits) für eine Adresse und Daten verwendet werden.
Je mehr Ziffern man verwendet desto größere Zahlen können in einer Operation verarbeitet werden wobei die maximal repräsentierbare Zahle $2^n - 1$ ist wenn n die Anzahl der Binären Ziffern (= Bits) ist.

n teilbar durch 8 zu machen, ist dabei relativ angenehm, weil 8 bit = 1 byte,
und Bytes sind nun mal die Grundlage für das meiste in der Informatik,
zudem kommen einige elementar ICs, z.B. Buffer oder Register, immer mit 8 bits.

Jedoch gilt, je mehr Bits man verwendet, desto mehr Schaltung braucht man und desto größer der Stromverbrauch, wodurch es für uns relativ schnell unrealistisch aufwendig wird

Wir haben uns daher für 16 bit entschieden, womit wir Zahlen bis zwischen 0 und 65535 darstellen können.
Diese Menge ist gerade große genug, um eine Zahl abzubilden, die für die meisten Programme groß genug ist, aber trotzdem auf Platinen noch einigermaßen handhabbar.

3.2 Memory

Um auch Daten oder Programme verwenden zu können haben wir dann als erstes noch einen größeren Speicher hinzu gefügt: Memory

Memory ist in 65536 (weil wir genau so viele Adressen in unseren 16bit speichen können) einzelne und unabhängige Speicherzellen (jede 16 Bit groß) aufgeteilt.
Der Computer kann damit auf 65536 words oder 131072 bytes (= 128 kib) direkt zugreifen.
(Die Größe messen wir in words (w) oder in kilowords (kiw), wobei 1 word = 2 bytes = 16 Bit sind.)

Memory wird dabei über LD- und STR-Instruktionen gesteuert.
Die LD-Instruktion kopiert einen Wert aus Memory und schreibt ihn in ein Ziel-Register, während STR an der angegebenen Adresse den Wert mit dem Wert des Quellen-Register überschreibt.

Um aber noch mehr Speicher zu bekommen und das später erwähnte Berechtigungssystem möglichst einfach umsetzen zu können haben wir noch Banken hinzugefügt:

Banken sind externe Memory Module, die eingesteckt werden können. Alle Banken sind gleich groß (16 kiw = 32 kib) und der Computer unterstützt maximal 65536 unterschiedliche Banken. Jede Bank hat dabei eine ID (BID), die von 0 bis 65535 reicht.
Der Computer hat zwei Bank-Slots (Slot 1 `0x8000 - 0xbfff` und Slot 2 `0xc000 - 0xffff`).

Es kann auf jede dieser Slots eine der theoretisch 65536 Banken gelegt werden.
Bank 1 ist dafür gedacht, dass jeder Prozess dort in seiner eigenen Bank lebt, sodass in dem Prozess absolute Memory Adressen verwendet werden können, weil das Programm immer an der gleichen Stelle liegt. Außerdem wird in der Bank auch der Stack des jeweiligen Prozesses gespeichert, sodass dieser von anderen Prozessen nicht überschrieben werden kann.
Bank 2 dient als potentieller zusätzlichen Speicher des Prozesses, oder als allgemeineren Speicherort, um zwischen den Prozessen Daten zu übermitteln.

3.3 Logik-Operationen und konditionale Jumps

Um die neuen Möglichkeiten durch 16-bit auch sinnvoll nutzen zu können haben wir dann noch eine ganze Reihe an neuen Operationen hinzugefügt: Viele Logikoperationen und komplexere konditionale Jumps.

Tabelle 1: Instruktions-Satz

Ins-ID	Name	op1	op2	op3	Beschreibung
0x00	NOP				Kein Effekt
0x01	MOV	reg		reg/c	op1 = op3
0x02	ADD	reg	reg	reg/c	op1 = op2 + op3
0x03	SUB	reg	reg	reg/c	op1 = op2 − op3
0x04	MUL?	reg	reg	reg/c	op1 = op2 ∗ op3
0x05	MULOF	reg	reg	reg/c	op1 = Overflow op2 ∗ op3
0x05	DIV?	reg	reg	reg/c	op1 = op2 / op3
0x06	XOR	reg	reg	reg/c	op1 = op2 ⊕ op3
0x07	AND	reg	reg	reg/c	op1 = op2 ∧ op3
0x08	OR	reg	reg	reg/c	op1 = op2 ∨ op3
0x09	NOT	reg		reg/c	op1 = ¬op3
0x0a	STR	reg	reg	reg/c	memory[op2 + op3] = op1
0x0b	LD	reg	reg	reg/c	op1 = memory[op2 + op3]
0x0c	BNK1			reg/c	Setzt BID für Bank-Slot1
0x0d	BNK2			reg/c	Setzt BID für Bank-Slot2
0x0e	PUSH			reg/c	SP–; memory[SP] = op1
0x0f	POP			reg	op3 = memory[SP]; SP++
0x10	CALL			reg/c	memory[SP] = IP; SP++; IP = op3
0x11	RET				IP = memory[SP]; SP++
0x12	TEST	reg		reg/c	Vergleicht op1 und op3
0x13	ME	reg		reg/c	op1 = op3 if E
0x14	MG	reg		reg/c	op1 = op3 if G
0x15	ML	reg		reg/c	op1 = op3 if L
0x16	MS	reg		reg/c	op1 = op3 if SUP
0x17	MI	reg		reg/c	op1 = op3 if OINT
0x18	MOFadd	reg		reg/c	op1 = op3 if O
0x19	MOFsub	reg		reg/c	op1 = op3 if O
0x1a	IOUT	reg		reg/c	IIO[op1] = op3/c
0x1b	DOUT	reg		reg/c	DIO[op1] = op3/c
0x1c	DIN	reg		reg	op3 = DIO[op1]
0x1d	AIO	reg			Aktiviert IO[op1]
0x1e	DIO	reg			Deaktivieren IO[op1]
0x1f	INT				Löst einen Interrupt aus
0x20	TM1			reg/c	Modi Timer1 = op3
0x21	TM2			reg/c	Modi Timer2 = op3
0x22	SSTL1			reg/c	lower Stop Timer1 = op3
0x23	SSTL2			reg/c	lower Stop Timer2 = op3
0x24	SSTH1			reg/c	higher Stop Timer1 = op3
0x25	SSTH2			reg/c	higher Stop Timer2 = op3
0x26	CFL	reg			op1 = lower Clock Freq
0x27	CFH	reg			op1 = higher Clock Freq

3.4 IO-Ports

Dazu hat sich die Idee von komplexer IO-Interaktion mit dem Computer ergeben: universelle Ports zum Anschließen von Festplatten, Sensoren, einfachen Tastaturen, LCD-Displays und einer Grafikkarte.

Die theoretisch 65536 Anschlüsse, wobei jedem eine Anschluss-ID zugewiesen ist, kommunizieren mit IO-Geräten über 4 Kommandos, die als 4 separate Pins zwischen dem Controller und dem Gerät umgesetzt sind: 1. Einen Befehl an das IO-Gerät schicken (auf Programm-Ebene); 2. Daten zu einem vorherigen Befahl an das Gerät schreiben; 3. Die Anfrage der IO-Gerätes Daten an den Computer zurück zu schreiben, was bei diesem normalerweise einen Interrupt auslöst; 4. Bestätigen des Computers, das das Externe Gerät schreiben darf.

3.5 Berechtigungssysten

Die vermutlich die größte Änderung war danach, dass wir ein Berechtigungssystem einführen wollten. Dafür haben wir dann Flaggen, Interrupts und noch mehr Instruktionen hinzugefügt.

3.6 ROM

Um den Computer komplett unabhängig funktionsfähig machen zu können, haben wir uns dann überlegt einen Teil des RAMs durch ROM zu ersetzen, um automatisch Programme von einem externen Speichermedium laden zu können.

3.7 Nutzung eines 2. Busses

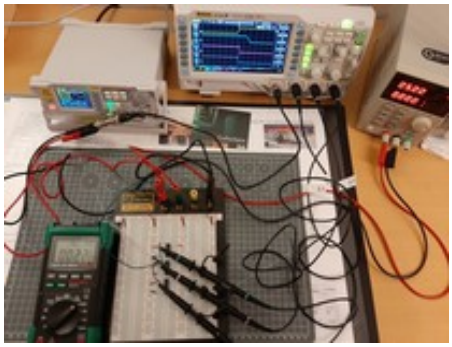
Um Memory innerhalb einer Instruktion zu beschreiben, muss dieses sowohl eine 16-bit Adresse über die Position in Memory erhalten, als auch den 16-bit Wert, der dort gespeichert werden soll, damit diese beiden Daten parallel übermittelt werden können brauchen wir also 2*16bit. Wir hätten dafür natürlich auch irgendwelche Kabel verlegen können, aber wir wollen in unserem System die Daten immer auf dem Bus haben, damit diese auch von anderen Modulen gelesen werden könnten, und die Adresse muss im Anwendungsfall als Stack vor Benutzung um eins erhöht oder verringert werden, was in unserem System durch die ALU erfolgen sollte, wobei diese nur auf den Bus ihre Daten auseben kann.

4 Tests

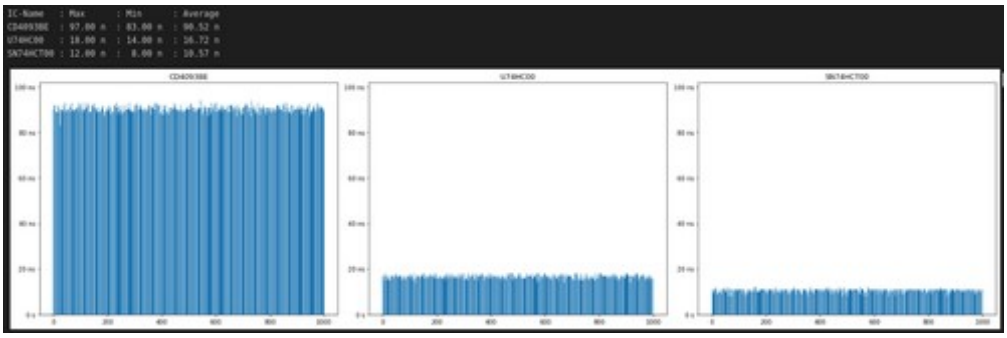
4.1 Breadboard-Test

Zwischendurch haben wir dann für die Einladung zu der lokalen Veranstaltung "Persvektive P" am 15.07.2021 wieder einen Prototyp auf dem Breadboard gebaut, wobei auch diese Version sehr unvollständig war.

4.2 Geschwindigkeit



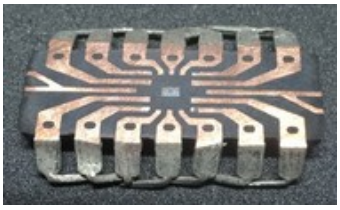
Um einen Eindruck zu bekommen, wie komplex unsere Programme werden können, ohne übermäßig lange zu brauchen, haben wir dann versucht zu errechnen, wie schnell wir den Computer maximal laufen lassen können, ohne fehlerhafte Ergebnisse zu erhalten. Also haben den "Propagation-delay", die Zeit, bis der Chip den Output liefert, den der laut der Eingabe haben müsste, aller Komponenten gemessen, die wir verwenden wollen. Die meisten Chips, die wir jetzt vorhaben zu verwenden, haben eine Verzögerung von ungefähr 8 Nanosekunden, allerdings haben wir bei den verschiedenen Versionen, die wir getestet haben, teilweise weitaus höhere Werte bis zu 150ns erreicht, was an sich noch nicht so schlecht ist, aber wenn wir damit rechnen, dass wir Reihen von bis zu 40 Chips in einem Clock-Zyklus haben, limitiert das die Geschwindigkeit schon sehr.

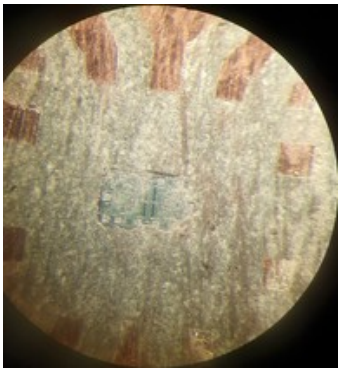


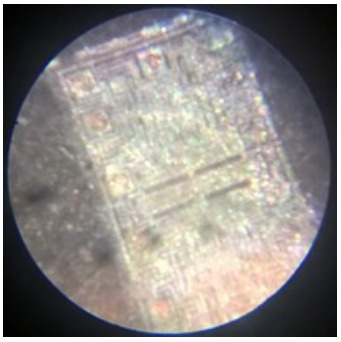
4.3 Funktionsweise der Chips

Wir wollen als Basis für unseren Computer ja Chips kaufen, aber wie funktionieren die eigentlich? Wir haben also versucht den Aufbau herauszufinden, indem wir einfach einen geöffnet haben, wobei "Einfach" nicht so ganz stimmt, weil die Hülle aus Epoxidharz

besteht, was sich fast nicht auflösen lässt. Also mussten wir mechanisch abtragen, also Schleifen. Leider haben wir es damit nie geschafft das gesamte Silizium/die Verbindungsbahnen frei zu legen, sodass wir keinen Schaltplan erstellen könnten, es gab uns aber trotzdem eine beeindruckende persönliche Einsicht:

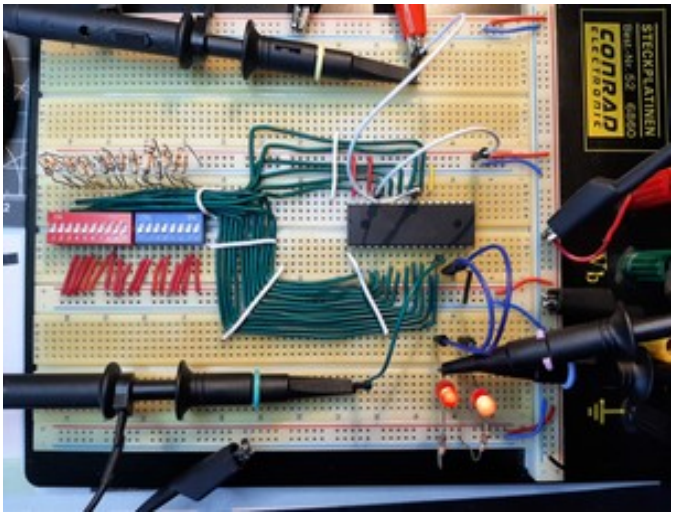






5 Umsetzung

5.1 Programmierung



Um den ROM-Chip zu programmieren haben wir zwar mal kurz ein bisschen manuell über Schalter programmiert, aber nachdem das offensichtlich zu lange brauchen würde haben wir dann einen Arduino (um genau zu sein, einen ESP32) programmiert und an die pins des ROM-Chips angeschlossen. Der Arduino bekommt die zu schreibenden bits wiederum aus einem Python-Programm, das den Assembler in Maschinensprache (eine binary Datei) umwandelt.

```
# Converting Instruction to binary
for i, token in enumerate(program):
    if token.type == "instruction":
        instruction_body = INSTRUCTIONS.get(token.value[0].lower())

        if not instruction_body:
            throw_error("Unknown Instruction \"{}\"".format(token.value[0]), token.line, token.file)

        instruction_parameter_order = list(instruction_body)[:-1]
        while "" in instruction_parameter_order:
            instruction_parameter_order.remove("")

        for j in REGISTERS:
            while j in instruction_parameter_order:
                instruction_parameter_order.remove(j)

        if len(token.value) - 1 != len(instruction_parameter_order):
            throw_error("Invalid Amount of parameters! {} requires {}".format(
                token.value[0], "".join(["<{}> ".format(j) for j in instruction_parameter_order])
            ), token.line, token.file)

        for expected, parameter in zip(instruction_parameter_order, token.value[1:]):
            if expected == "r" and type(parameter) == str and parameter.lower() not in REGISTERS:
                throw_error("\"{}\" is not a register, expected register!".format(
                    parameter.value if type(parameter) == LiteralValue else parameter
                ), token.line, token.file)
            elif expected == "c" and type(parameter) != LiteralValue:
                throw_error("\"{}\" is not a literal, expected literal".format(parameter), token.line, token.file)

        instruction_binary = instruction_body[-1]
        j_body = 0
        j_ins = 1

        while j_body < len(instruction_body) - 1:
            token.binary_data = b"\x00\x00"
            if instruction_body[j_body] != "":
                if type(token.value[j_ins]) == LiteralValue:
                    instruction_binary |= 0b1000000
                    token.binary_data = token.value[j_ins].bin_value
                else:
                    instruction_binary |= REGISTERS.index(token.value[j_ins].lower()) << (7 + j_body*3)
                j_ins += 1
            j_body += 1
        token.binary_data = token.binary_data + instruction_binary.to_bytes(2, ENDIANNESS)
```

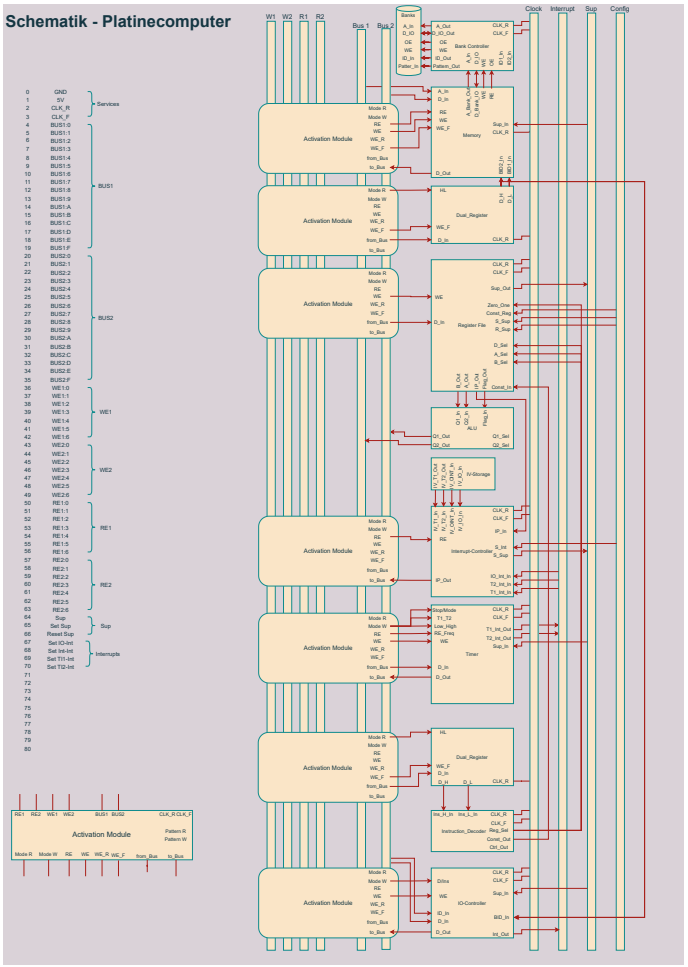
5.2 Weitere Tests zum Ätzen

Im Schuljahr 2021-2022 haben wir dann wieder weitere Tests zum Ätzen - diesmal sowohl erfolgreicher, als auch besser dokumentiert, um die optimalen Belichtungs- und Ätzzeiten zu finden. Diese sind in der nebenstehenden Tabelle aufgelistet.

Nummer	UV-Lampe	t_{uv} in min	d_{uv} in cm	c(NaOH) in g/100ml	ϑ(NaOH) in °C	t_{NaOH} in min	c($FeCl_3$) in g / 100ml	ϑ($FeCl_3$) in °C	t_{FeCl3} in min	Qualität
-1	alt	1.00	?	1.5	21	?	80	40	15.00	5.00%
-2	alt	10.50	?	1.5	21	?	80	38	15.00	15.00%
-3	alt	20.00	?	1.5	21	?	80	39	15.00	85.00%
-4	alt	20.00	10.00	1.5	21	?	100	42	15.00	80.00%
-5	alt	25.00	10.00	1.5	21	?	100	48	15.00	???
-6	alt	20.00	10.00	1.5	21	?	100	48	15.00	0%
-7	alt	20.00	10.00	1.5	21	20.00	100	52	15.00	???
4	neu KW	15.00	10.00	1.5	21	6.40	80	39	20.00	70.00%
7	neu KW	20.00	10.00	1.5	20	6.00	84	38	6.50	90.00%
8	neu KW	30.00	10.00	1.5	20	6.50	82	32	4.85	95.00%
11	neu KW	20.00	10.00	1.5	21	10.60	80	38	9.85	90.00%
12	neu KW	25.00	10.00	1.5	21	6.25	80	42	6.08	70.00%
22	neu KW	20.00	10.00	1.5	20	21.55	80	40	14.48	0.00%
14	neu KW	30.00	10.00	1.5	20	6.60	80	49	9.85	65.00%

5.3 Spezifikation

Gleichzeitig haben wir auch angefangen unsere erste vollständige Version der Spezifikation zu schreiben siehe: Handbuch.pdf



5.4 Design der Schaltungen

Zum designen haben wir uns schlussendlich darauf festgelegt, dass wir die Schaltungen zuerst in dem Logiksimulationsprogramm Logisim erstellen, und danach den Plan noch einmal als Ätzbaren Schaltplan erstellen, wobei wir hierfür kein wirklich zufriedenstellendes Produkt finden konnten. Schließlich haben wir uns vor allem wegen der einfachheit es zu bekommen und Installieren für KiCad entschieden, was uns allerdings ziemlich viel frust beschert hat.