

# Platinencomputer - Langfassung

Alexander Wersching und Simon Walter

2022

## Inhaltsverzeichnis

<b>1</b>	<b>Idee</b>	<b>2</b>
<b>2</b>	<b>Nutzen von Platinen</b>	<b>2</b>
2.1	Warum Platinen? . . . . .	2
2.2	Erste Versuche der Umsetzung . . . . .	3
2.3	Design . . . . .	3
<b>3</b>	<b>Anforderungen V2</b>	<b>4</b>
3.1	Daten- und Adresslänge . . . . .	4
3.2	Memory . . . . .	5
3.3	ROM . . . . .	5
3.4	Instruktionen . . . . .	6
3.5	IO-Ports . . . . .	7
3.6	Berechtigungssystem . . . . .	8
<b>4</b>	<b>Tests</b>	<b>8</b>
4.1	Breadboard-Test . . . . .	8
4.2	Geschwindigkeit . . . . .	8
4.3	Funktionsweise der Chips . . . . .	9
<b>5</b>	<b>Umsetzung</b>	<b>10</b>
5.1	Programmierung . . . . .	10
5.2	Weitere Tests zum Ätzen . . . . .	10
5.3	Spezifikation . . . . .	12
5.4	Nutzung eines zwei Busses . . . . .	12
5.5	Design der Schaltungen . . . . .	13

# 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 mein (Simons) Interesse 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.

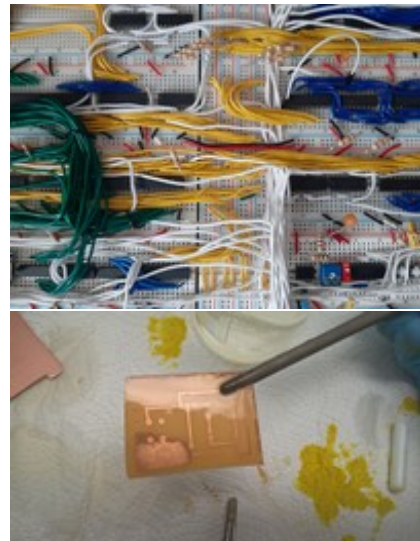


## 2 Nutzen von Platinen

### 2.1 Warum 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.

Als Alternative für Breadboards haben wir uns für Platinen (PCBs) entschieden. 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, währenddessen wir natürlich nicht Ätzen können. Wir hätten es natürlich zu Hause versuchen können, hatten da aber keine adäquate 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, modulares Design ausgedacht. Dieses ist in Abbildung 1

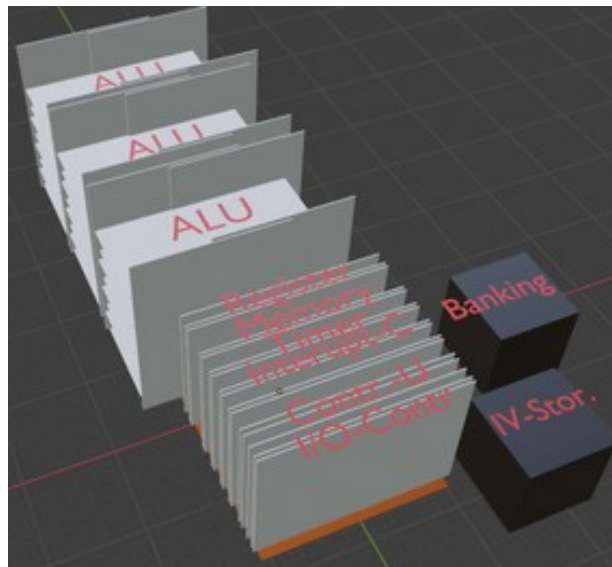


Abbildung 1: Offensichtlich ist das mit unseren aktuell geplanten Modulen: am Boden sitzt eine große Bus-Platine, auf der wir alle Module senkrecht aufstecken.

### 3 Anforderungen V2

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

#### 3.1 Daten- und Adresslänge

Aus den ursprünglich geplanten 8-Bit wurden 16-Bit-Daten- und Adresslänge:

Die Daten- und Adresslänge 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. Dabei ist die maximal repräsentierbare Zahl  $2^n - 1$  ist, wenn  $n$  die Anzahl der Binären Ziffern (= Bits) ist.

Ein  $n$  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 elementare ICs (Integrated Circuits), z.B. Buffer oder Register, immer mit 8 Bits.

Jedoch gilt, je mehr Bits man verwendet, desto mehr Schaltungen werden benötigt 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 groß genug, um Zahlen abzubilden, damit die meisten Programme funktionieren können, aber trotzdem noch auf Platinen handhabbar ist.

## 3.2 Memory

Für Daten und Programme verwenden wir in unserem Computer ein Modul: Memory

Unser Memory-Modul ist in 65536 einzelne und unabhängige Speicherzellen (jede 16 Bit groß) aufgeteilt. (weil wir genau so viele Adressen in unseren 16 Bit speichern können)

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 das Ziel-Register, während STR an der angegebenen Adresse den Wert mit dem Wert des Quellen-Registers überschreibt.

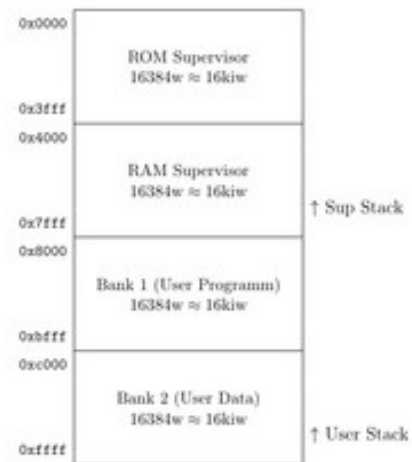


Abbildung 2: Speicher Aufteilung: Von niedrigen zu hohen Adressen in Hexadezimalschreibweise

Um aber noch mehr Speicher zu nutzen und das später erwähnte Berechtigungssystem möglichst leicht 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), auf den jeweils eine der Banken gelegt werden kann.

Bank-Slot 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-Slot 2 dient als potentieller zusätzlicher Speicher des Prozesses, oder als allgemeiner Speicherort, um zwischen den Prozessen Daten zu übermitteln.

## 3.3 ROM

Um den Computer komplett unabhängig funktionsfähig machen zu können, haben wir uns dann überlegt

einen Teil des RAMs (0x0000 - 0xffff) durch ROM zu ersetzen, um automatisch Programme von einem externen Speichermedium laden zu können. Zusehen in Abbildung ??

### 3.4 Instruktionen

Um die neuen Möglichkeiten durch 16-Bit auch sinnvoll nutzen zu können, haben wir dann noch viele Logikoperationen und komplexere konditionale Jumps hinzu gefügt: Der Computer soll die Grundoperationen: Addition (ADD), Subtraktion (SUB), Multiplikation (MUL), Division (DIV), so wie die bitweisen Operationen: Exklusives Oder (XOR), Und (AND), Nicht (NOT) und Oder (OR) beherrschen.

Der Computer besitzt zudem folgende Instruktion um mit Memory zu interagieren: Laden (LD), Speicher (STR), Bank 1 und Bank 2 wechseln (BNK1 und BNK2) und folgende Instruktion um mit dem Hardware Stack zu interagieren: push (PUSH), pop (POP), Funktionsaufruf (CALL) und Funktionsrückgabe (RET).

Zudem die folgenden IO-Instruktionen: Instruktion Send (IOUT), Daten Senden (DOUT), Daten lesen (DIN), IO-Gerät Aktivieren (AIO) und IO-Gerät Deaktivieren

Aber auch noch folgende Instruktion um Interrupts zu verwalten: Software Interrupt auslösen (INT), Modus des Timers 1 und 2 setzten (TM1/TM2) und die Stop Werte des Timers 1 und 2 setzten (SSTL1/SSTL2/SSTH1/SSTH2)

Insgesamt hat unser Computer damit folgende Instruktionen: Tabelle 1

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 $\oplus$ op3
0x07	AND	reg	reg	reg/c	op1 = op2 $\wedge$ op3
0x08	OR	reg	reg	reg/c	op1 = op2 $\vee$ op3
0x09	NOT	reg		reg/c	op1 = $\neg$ 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

Tabelle 1: Instruktions-Satz

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
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	erster Stop Timer1 = op3
0x23	SSTL2		reg/c	zweiter Stop Timer2 = op3
0x24	SSTH1		reg/c	erster Stop Timer1 = op3
0x25	SSTH2		reg/c	zweiter Stop Timer2 = op3
0x26	CFL	reg		op1 = erster Clock Freq
0x27	CFH	reg		op1 = zweiter Clock Freq

### 3.5 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 oder 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 Befehl an das Gerät schreiben;
3. Die Anfrage des 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;

### 3.6 Berechtigungssystem

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

## 4 Tests

### 4.1 Breadboard-Test

Zwischendurch haben wir dann für die Einladung zu der lokalen Veranstaltung "Perspektive P" am 15.07.2021 wieder einen unvollständigen Prototyp auf dem Breadboard gebaut.

### 4.2 Geschwindigkeit

Um zu errechnen, wie schnell wir den Computer maximal laufen lassen können, ohne fehlerhafte Ergebnisse zu erhalten, haben wir den "Propagation-delay" gemessen. Der "Propagation-Delay" ist die Zeit, bis der Chip den Output liefert, den der laut der Eingabe haben müsste. Wir haben diese Messungen bei einigen unserer wichtigsten Komponenten durchgeführt. Die meisten Chips, die wir jetzt vorhaben zu verwenden, haben eine Propagation-Delay von zwischen 10 und 20 ns wie z.B. bei NAND, AND oder OR-Gattern. Allerdings haben wir bei den einigen Versionen, die wir getestet haben, teilweise weitaus höhere Werte bis zu 150 ns gemessen. Dies ist an sich noch nicht so schlecht, aber wenn wir damit rechnen, dass wir Reihen von bis zu 40 Chips verwenden, limitiert das die Geschwindigkeit schon sehr. Der Messaufbau den wir verwendet haben ist dabei in Abbildung 3 zu sehen.

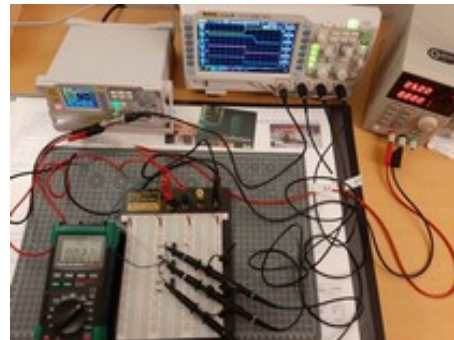


Abbildung 3: Mess



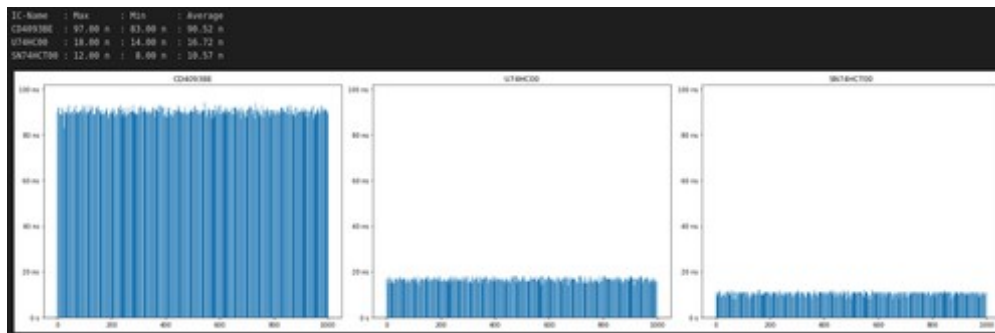
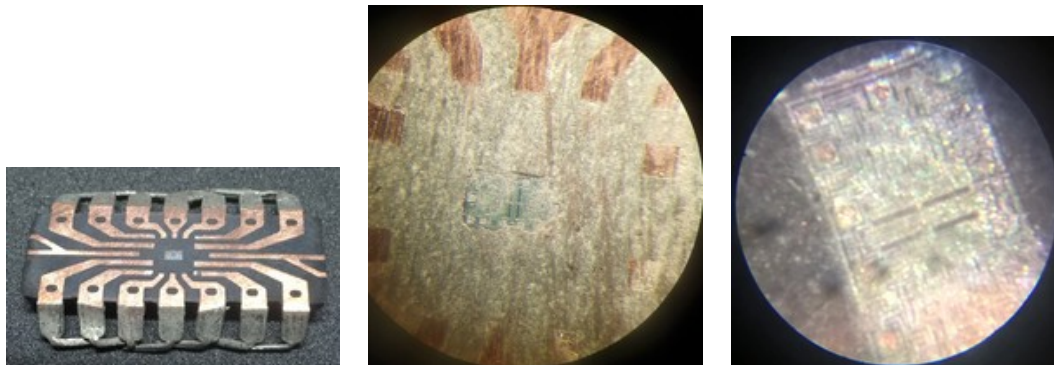


Abbildung 4: Messergebnisse der Propagation-Delay Messung von drei Quod-NAND-Gattern in Verschieden Ausführungen (CD4096BE, SN74HC00, SN74HCT00)

### 4.3 Funktionsweise der Chips

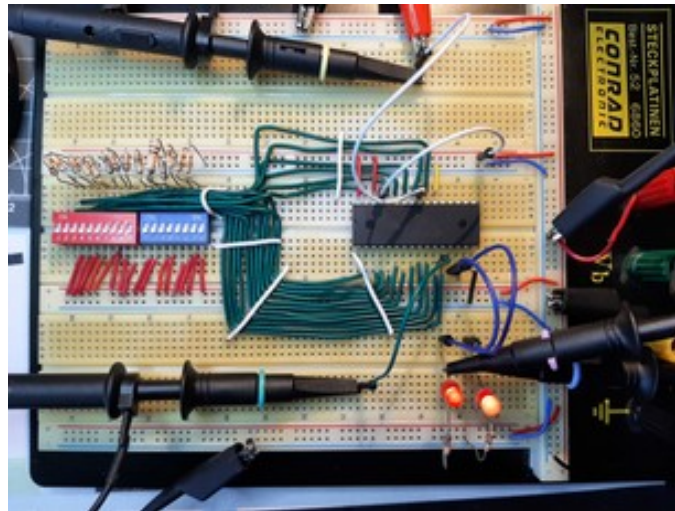
Wir wollen als Basis für unseren Computer ja Chips kaufen, aber wie funktionieren die eigentlich? Wir haben also versucht den Aufbau herauszufinden, indem wir einfach einen geöffnet haben, wobei "Einfach" nicht so ganz stimmt, weil die Hülle aus Epoxidharz besteht, was sich fast nicht auflösen lässt.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



```
323 # Converting Instruction to binary
324 for i, token in enumerate(program):
325     if token.type == "Instruction":
326         instruction_body = INSTRUCTIONS.get(token.value[0].lower())
327
328         if not instruction_body:
329             throw_error("Unknown Instruction {}".format(token.value[0]), token.line, token.file)
330
331         instruction_parameter_order = list(instruction_body[:-1])
332         while "" in instruction_parameter_order:
333             instruction_parameter_order.remove("")
334
335         for j in REGISTERS:
336             while j in instruction_parameter_order:
337                 instruction_parameter_order.remove(j)
338
339         if len(token.value) - 1 != len(instruction_parameter_order):
340             throw_error("Invalid Amount of parameters! {} requires {}".format(
341                 token.value[0], "".join("{}<{}> ".format(j) for j in instruction_parameter_order)
342             ), token.line, token.file)
343
344         for expected, parameter in zip(instruction_parameter_order, token.value[1:]):
345             if expected == "r" and type(parameter) == str and parameter.lower() not in REGISTERS:
346                 throw_error("{} is not a register, expected register!".format(
347                     parameter.value if type(parameter) == LiteralValue else parameter
348                 ), token.line, token.file)
349             elif expected == "c" and type(parameter) != LiteralValue:
350                 throw_error("{} is not a literal, expected literal".format(parameter), token.line, token.file)
351
352         instruction_binary = instruction_body[-1]
353         j_body = 0
354         j_ins = 1
355
356         while j_body < len(instruction_body) - 1:
357             token.binary_data = b"\x00\x00"
358             if instruction_body[j_body] != "":
359                 if type(token.value[j_ins]) == LiteralValue:
360                     instruction_binary |= 0b1000000
361                     token.binary_data = token.value[j_ins].bin_value
362                 else:
363                     instruction_binary |= REGISTERS.index(token.value[j_ins].lower()) << (7 + j_body*3)
364                     j_ins += 1
365             j_body += 1
366         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 <sub>UV</sub> in min	d <sub>UV</sub> in cm	c(NaOH) in g/100ml	θ(NaOH) in °C	t <sub>Ätz</sub> in min	c(FeCl <sub>3</sub> ) in g / 100ml	θ(FeCl <sub>3</sub> ) in °C	t <sub>Ätz</sub> 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%

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

Um Memory innerhalb eines Clock-Zyklus zu beschreiben, muss diesem sowohl eine 16-Bit-Adresse als auch den 16-Bit-Daten-Wert übermittelt werden. Damit diese beiden Daten parallel übermittelt werden können brauchen wir also  $2 \cdot 16$  Bit. Wir hätten dafür natürlich auch irgendwelche Kabel verlegen können, aber wir wollen in unserem System die Daten immer auf dem Bus transportieren, damit diese auch von anderen Modulen gelesen werden könnten. Die Adresse muss im Anwendungsfall des Stacks vor Benutzung um eins erhöht oder verringert werden, was in unserem System durch die ALU erfolgt, wobei diese ihre Daten nur auf den Bus ausgeben kann.



## 5.5 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.

