

# Platinencomputer - Handbuch

Alexander Wersching und Simon Walter

2021

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>3</b>
1.1	Ziel diese Dokuments . . . . .	3
1.2	Geschichte . . . . .	3
<b>2</b>	<b>Spezifikation des Computers</b>	<b>3</b>
2.1	Instruktions-Satz . . . . .	4
2.2	Supervisor-/Usermode . . . . .	4
2.3	Registers . . . . .	6
2.3.1	Kopieren und Bewegen von Daten . . . . .	7
2.3.2	Flaggen . . . . .	7
2.3.3	Arithmetische und Logische Operationen . . . . .	8
2.4	Memory . . . . .	9
2.4.1	Banking . . . . .	10
2.4.2	Hardware Stack . . . . .	11
2.5	Input/Output . . . . .	12
2.5.1	Berechtigungsliste . . . . .	15
2.6	Interrupts . . . . .	15
2.6.1	Timed-Interrupts . . . . .	16
2.6.2	. . . . .	17
<b>3</b>	<b>Kommentar zur Specification</b>	<b>17</b>
3.1	16 bit Daten/Address Länge . . . . .	17
3.2	Register Anzahl . . . . .	18
3.3	Gedanken Hinter dem Supervisormode . . . . .	19

# 1 Einführung

## 1.1 Ziel diese Dokuments

Dieses Dokument soll die Funktionsweise unseres Platinencomputers beschreiben. Dabei soll nicht auf die tatsächliche auf Implementation wertgelegt werden, aber viel mehr auf die einzelnen Features und Funktionen. Diese Spezifikation wurde vor dem Bau des Computers bzw. des Emulators geschrieben, um eine Grundlage für alle weiteren Konstruktionsschritte zu haben.

## 1.2 Geschichte

Dieses Dokument hat sehr viel Änderung durchleben müssen, diese kann unter <https://github.com/jufo-ufo/Breadboard-Computer> eingesehen werden. Zudem hatte diese Spezifikation zwei, wenn auch nicht vollständige, Vorgänger: [https://github.com/jufo-ufo/Breadboard-Computer/blob/master/specification\\_v0.1.txt](https://github.com/jufo-ufo/Breadboard-Computer/blob/master/specification_v0.1.txt) und [https://github.com/jufo-ufo/Breadboard-Computer/blob/master/specification\\_v0.2.txt](https://github.com/jufo-ufo/Breadboard-Computer/blob/master/specification_v0.2.txt). Wobei die zweite Fassung als Basis dieses Dokuments diene.

# 2 Spezifikation des Computers

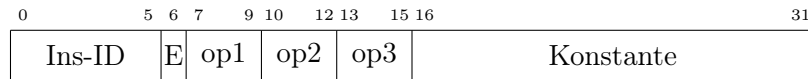
Ein Computer hat eine Reihe verschiedener “high-level“ Aufgaben und Features, die es ihm erlauben, seiner Tätigkeiten nach zu gehen! Diese Features können wie folgt auf gegliedert werden: Register, Memory, Input/Output, Interrupts, Supervisor-/Usermode und der Instruktionssatz.

Aber um zuerst ein paar Standards festzulegen: Der Computer hat eine Daten-/Adressenlänge von 16 Bit, d.h. er kann maximal 16 Bit auf einmal operieren bzw. können maximal 16 Bit an Daten/Adressen im Computer auf einmal bewegt werden. Mehr dazu in Unterabschnitt 3.1

Unsere Computer ist eine von Neuman-Maschine, d.h. sowol Daten als auch Programm während in einem Hauptspeicher (Memory) gespeichert

## 2.1 Instruktions-Satz

Das Programm, welches den Computer vorschreibt, was er abzuarbeiten hat, besteht aus Instruktionen. Jede Instruktion enthält die 6 Bit langen Ins-ID, ein 16 Bit Konstante und 3 gewählte Register in Form einer 3 Bit Register Nummer (RN). Damit ergibt sich eine 32 Bit, also 2 words, lange Instruktion. Die Codierung ist dabei in Unterabschnitt 2.1 festgehalten.



Jeder zur Verfügung stehenden Instruktion ist dabei exakt einer oben erwähnten Ins-ID zugeordnet. Alle Ins-ID und dazugehörige Instruktion ist in Tabelle 1

Jede Instruktion hat 3 Operatanten (op), welche Register sind. Im Falle des dritten Operatanten kann auch eine Konstante verwendet werden. Das E Bit gibt dabei an, welches von beiden genutzt werden soll. Bei gesetztem E Bit wird die Konstante verwendet, bei nicht gesetztem E Bit das Register.

## 2.2 Supervisor-/Usermode

Der Computer hatte ein eingebautes Rechte-System. Er kann dabei in zwei Modi operieren: dem Supervisor- und Usermode. Im Supervisor-Mode kann der Computer alle Funktion ohne Einschränkungen nutzen. Im Usermode hingegen ist der Computer in einigen Funktion eingeschränkt. Folgendes kann der Computer im Usermode *nicht* machen:

- Setzen der SUP Flagge
- Änderung der OINT Flagge
- Lesen und/oder schreiben im Bereich von 0x0000 - 0x7fff
- Änderung der Banken
- Die Modifikation der IO-Berechtigungsliste
- Zugriff auf IO-Geräte, welche nicht in der Berechtigungsliste gespeichert sind (siehe Unterunterabschnitt 2.5.1)

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	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
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	MOF	reg		reg/c	op1 = op3 if O
0x15	MG	reg		reg/c	op1 = op3 if G
0x16	ML	reg		reg/c	op1 = op3 if L
0x17	MS	reg		reg/c	op1 = op3 if SUP
0x18	MI	reg		reg/c	op1 = op3 if OINT
0x19	IOUT	reg		reg/c	IIO[op1] = op3/c
0x1a	DOUT	reg		reg/c	DIO[op1] = op3/c
0x1b	DIN	reg		reg	op3 = DIO[op1]
0x1c	AIO	reg			Aktiviert IO[op1]
0x1d	DIO	reg			Deaktivieren IO[op1]
0x1e	INT				Löst einen Interrupt aus
0x1f	TM1			reg/c	Modi Timer1 = op3
0x20	TM2			reg/c	Modi Timer2 = op3
0x21	SSTL1			reg/c	lower Stop Timer1 = op3
0x22	SSTL2			reg/c	lower Stop Timer2 = op3
0x23	SSTH1			reg/c	higher Stop Timer1 = op3
0x24	SSTH2			reg/c	higher Stop Timer2 = op3
0x25	CFL	reg			op1 = lower Clock Freq
0x26	CFH	reg			op1 = higher Clock Freq

Der Modus des Computers (Supervisor or User) wird dabei in der SUP Flagge gespeichert. Eine gesetzte SUP Flagge entspricht dem Supervisormode und eine nicht gesetzte SUP dem Usermode. Der Computer startete dabei mit gesetzter SUP Flagge.

Zudem ändert sich je nach Modus des Computer die Start position des Stacks wie in Unterunterabschnitt 2.4.2 erläutert.

Mehr zu der Motivation hinter diesem System in Unterabschnitt 3.3

## 2.3 Registers

Register sind kleine, sehr schnelle Speichereinheiten, die genutzt werden können, um Zwischenergebnisse, lokal Variablen, Argumenten für Funktionen zu speichern oder um Daten im Computer zwischen den einzelnen Komponenten zu verschieben. Aus jedem Register kann *gelesen* (*Kopierten* des alten Wertes an einen anderen Ort) oder *geschrieben* werden (den alten Wert mit einem neuen Wert *überschreiben*)

Der Computer besitzt 8 verschiedene, jeweils 16 Bit lange Register. Jedes Register hat dabei eine eigene Registernummer (RN). Machen Register erfüllen zudem noch spezielle Aufgaben und sollten auch nur so verwendet werden. Diese Funktionen sind in Tabelle 2 zu sehen.

Mehr zur Registeranzahl in Unterabschnitt 3.2

Tabelle 2: Registeraufteilung

RN	Name	Funktion
0	ZERO	Konstante 0x0000; Schreiben hat keinen Effekt
1	IP	Instruction Pointer, zeigt auf aktuell auszuführende Instruktion
2	SP	Stack Pointer, zeigt auf obersten Wert des Stacks, siehe 2.4.2
3	A	Allzweck Register ohne besonderen Aufgaben
4	B	Allzweck "
5	C	Allzweck "
6	D	Allzweck "
7	FLAG	Enthält alle wichtigen Flaggen, siehe Unterunterabschnitt 2.3.2

Die Register A-D sind dabei Register, welche frei für jede Nutzung zur Verfügung stehen. Es ist *stark zu empfehlen*, die Spezialregister (ZERO, IP, SP und FLAG) *nicht* für die Speicherung von Daten zu verwenden, da dies evtl. Programmabläufe stark stören könnte

oder im Falle von ZERO oder den letzten 8 bit des Flaggen Registers im Usermode nicht möglich ist. Sie sind ausdrücklich nur für ihre zugewiesenen Aufgaben da.

### 2.3.1 Kopieren und Bewegen von Daten

Das Kopieren eines Wertes in ein Register ist mit der MOV-Instruktion möglich. Die MOV-Instruktion nimmt dabei als ersten Operanden das Ziel Register und als zweiten Operanden das Quellenregister oder eine Konstante an. MOV kopiert nun den Wert des zweiten Operanden in den des Ersten. Die Notation ist dabei wie folgt:

MOV <Ziel-Register> <Quellen-Register/Konstante>

### 2.3.2 Flaggen

Das Register FLAG enthält alle Statusflaggen. Eine Flagge kann entweder gesetzt oder nicht gesetzt sein, daher kann jede Flagge als ein Bit repräsentiert werden. Jedes Bit in FLAG stellt daher eine Flagge da. Flaggen werden bei unterschiedlichen Ereignissen geschaltet. Dies und welche Flaggen es gibt, ist in Tabelle 3 zusehen.

Tabelle 3: Flaggenaufteilung in FLAG

Bit	Name	Funktion	Schaltung
0	OF	Integer Overflow einer Rechenoperation	Bei jeder ALU Operation
1	E	Gleichheit zweier Werte	TEST arg[0] = arg[1]
2	G	Ungleichheit (>) zweier Werte	TEST arg[0] > arg[1]
3	L	Ungleichheit (<) zweier Werte	TEST arg[0] < arg[1]
4			
5			
6			
7			
8	OVTI1	Override Timed-Interrupt1, siehe Unterabschnitt 2.6	
9	OVTI2	Override Timed-Interrupt2, siehe Unterabschnitt 2.6	
10	OVIOI	Override IO-Interrupt, siehe Unterabschnitt 2.6	
11			
12			
13			
14			
15	OINT	Override Interruption	
15	SUP	Supervisor-/Usermode, siehe Unterabschnitt 2.2	

Alle Flaggen, die mit TEST als Schaltung gekennzeichnet sind, werden bei der Ausführung der TEST Operation geschaltet.

Das FLAG Register verhält sich ebenfalls wie ein Register, daher können Flagge auch per Hand gesetzt oder nicht gesetzt werden. Eine Ausnahme machen die letzten 8 Flaggen darunter SUP und OINT die nur geschaltet werden können, wenn der Computer sich im Supervisormode befindet, siehe Unterabschnitt 2.2. Ein Schalten im Usermode hat keinen Effekt.

Neben der MOV-Instruktion gibt es auch die sogenannten konditionalen MOV-Instruktionen. Diese funktionieren genauso wie die MOV-Instruktion, indem sie das Quellen-Register/Konstante in das Ziel-Register kopiert, aber dies jedoch nur tun, wenn die entsprechende Flagge gesetzt ist. Die Notation dafür ist wie folgt:

M<Flagge> <Ziel-Register> <Quellen-Register/Konstante>

Daher ergeben sich folgende konditionalen MOV-Instruktionen: ME (E Flagge), MOF (OF Flagge), MG (G Flagge), ML (L Flagge), MI (ONIT Flagge), MS (SUP Flagge).

### 2.3.3 Arithmetische und Logische Operationen

Der Computer hat zudem die Fähigkeit, arithmetische und logische Operation durchzuführen. Alle Operationen sind dabei binär daher Kombinieren zwei Eingangswerte oder Operanden zu einem Ausgangswert. Die Ausnahme macht dabei die NOT-Instruktion, die nur einen Eingangswert annimmt und auf diesen ein Bitweises Nicht durchführt.

Alle arithmetischen und logische Operationen funktionieren dabei immer gleich: Sie kombinieren das erste Quelle-Register mit dem zweiten Quellen-Register und schreiben das Ergebnis in das Ziel-Register. Dabei ist zu beachten, dass das zweite Quellen-Register durch einen konstanten Wert ersetzt werden kann. Die Notation dafür sie wie folgt aus:

<Operation> <Ziel-Register> <Quellen-Register 1> <Quellen-Register 2/Konstante>

Der Computer unterstützt dabei folgende Operationen: Addition (ADD), Subtraktion (SUB), Multiplikation (MUL), Division (DIV), Bitweises exklusives Oder (XOR), Bitweise Und (AND), Bitweises Oder (OR), Bitweises Nicht (NOT)

Wichtig zu sagen ist, dass bei alle Operationen ( $f(x_1, x_2) = y \bmod 65536$ ) gilt  $65535 \geq y \geq 0$  ist wo bei gilt  $y \in \mathbb{N}$ . Der springende Punkt ist, dass bei einer Rechnung wie



$65535 + 1$  das Ergebnis nicht 65536 ist, sondern 0. Bei so einem “Overflow“ wird die OF-Flagge gesetzt. Wenn diese bei einer Rechenoperation nicht eintritt, dann wird die OF-Flagge gelöscht. Dieses Problem tritt auf da wir nur 16 Bit als Datenlänge zu Verfügung haben, mehr dazu in Unterabschnitt 3.1

## 2.4 Memory

Während die Register für lokale Variablen und Zwischenergebnisse gedacht sind, ist Memory (Arbeitsspeicher) für die Speicherung von größeren Datenmengen gedacht.

Es ist in 65536 einzelne und unabhängige Speicherzellen (Jede 16 Bit große) aufgeteilt. Jeder Zelle ist einer Adresse, welche von 0 bis 65535 reicht, zugeordnet. Der Computer kann damit auf 65536 words oder 131072 bytes ( $\approx 131$  kb) zugreifen<sup>1</sup>. Wichtig ist, dass die Gesamt Größe, dank Banking siehe Unterabschnitt 2.4.1 theoretisch weitaus größer sein kann.

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. Die Adresse wird zusammen gesetzt durch die Addition des Address-Register1 und entweder in dem Address-Register2 oder einer Konstante. Die Notation sieht dabei wie folgt aus:

LD <Ziel-Register> <Address-Register1> <Address-Register2/Konstante>

STR <Quellen-Register> <Address-Register1> <Address-Register2/Konstante>

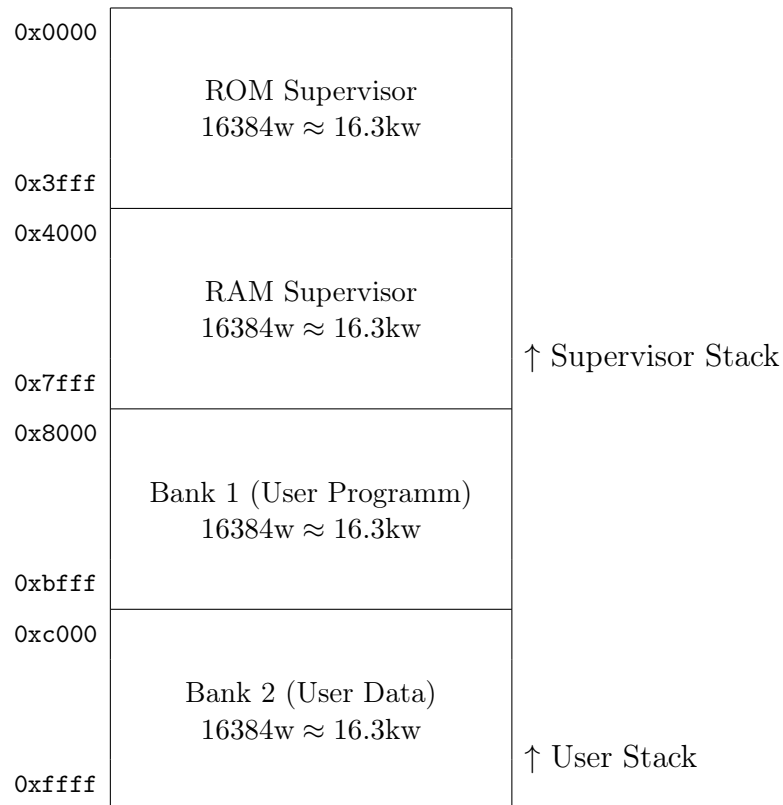
Zudem wurden bestimmte Bereichen bestimmte Aufgaben bzw. Funktionen zugewiesen, wie in Abbildung 1 zu sehen. Die zwei Hardware Stacks sind dort gekennzeichnet mit  $\uparrow$ , mehr dazu Stacks in Unterabschnitt 2.4.2

Die mit Supervisor gekennzeichneten Bereich können nur dann gelesen oder geschrieben werden, wenn sich der Computer im Supervisormode befindet. Mehr zu Supervisormode in Unterabschnitt 2.2. Der Bereich, der mit ROM versehen wurde, ist dabei nicht beschreibbar, sondern kann nur gelesen werden. Eine STR-Instruktion hat keine Wirkung. Eine Änderung des Inhalts der ROM Bereiche ist nur durch den Austausch des darunterliegenden physikalischen ROM-Chips möglich.

---

<sup>1</sup>Die Größe messen wir in words (w) oder in kilowords (kw), wobei 1 word = 2 bytes = 16 Bit.

Abbildung 1: Memory Karte



### 2.4.1 Banking

Banken sind externe Memory Module, die eingesteckt werden können. Alle Banken sind gleich groß (16.3kw = 32.6 kb) 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). Er kann nun auf jede dieser Slots eine der theoretisch 65536 Bank legen <sup>2</sup>. Das Banken legen lässt sich über die SBK1- (Select Bank Slot 1) und SBK2- (Select Bank Slot 2) Instruktion erreichen. Die BIN kann dabei als Konstante oder als Register vorliegen. Die Notation ist dabei wie folgt:

SBK<1/2> <BIN-Register/Konstante>

Somit führt das Beschreiben/Lesen an einer Adresse welche einem Bank-Slot zugeordnet

<sup>2</sup>Es ist dabei möglich, die gleiche Bank auf beide Slots zu legen, auch wenn das nur bedingt nützlich ist.

ist, zum Beschreiben/Lesen der ausgewählte Bank.

Durch dieses System lassen sich  $65536 \cdot 16.3 \text{ kw} = 1.06 \text{ Gw} = 2.12 \text{ Gb}$  an Arbeitsspeicher ansprechen.

Das Bankensystem dient zudem dazu, Prozesse voneinander zu isolieren, da der Computer im Usermode die Bank nicht verändern kann, siehe Unterabschnitt 2.2.

### 2.4.2 Hardware Stack

Der Stack (oder, auch Stapelspeicher) ist eine LIFO (Last In First Out) Datenstruktur. Sie dient hauptsächlich dazu, Subroutinen zu verwalten.

Er funktioniert dabei wie ein Papierstapel. Es kann ein Wert auf den Stack gelegt, werden (PUSH-Instruktion) oder ein Wert von oben abgehoben werden (POP-Instruktion). Wobei der Wert, der auf den Stack gelegt bzw. abgehoben werden soll, entweder im Quellen-Register bzw. Ziel-Register oder als Konstante vorliegt. Die Notation ist dabei wie folgt:

PUSH <Quellen-Register/Konstante>

POP <Ziel-Register>

Der Stack liegt dabei physisch in Memory vor. Jedes Element des Stacks okkupiert dabei eine Memory-Zelle (16 Bit). Das Stack-Point Register-(SP) hält die Adresse, welche die nächste freie Memory-Zelle zeigt, die eins über dem obersten Wert des Stacks liegt. Der Stack wächst dabei von hohen Adressen zu niedrigen Adressen, wie in Abbildung 2 zusehen.

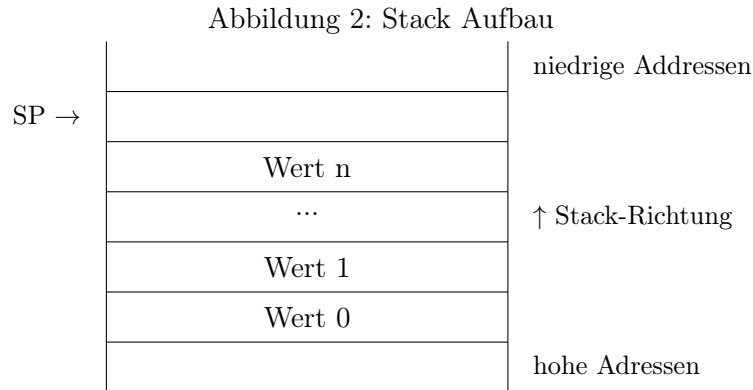
Bei einer PUSH Instruktion wird zuerst an die Stelle auf die SP zeigt der Wert des Quellen-Register/Konstante geschrieben und darauf der SP um 1 dekrementiert. POP dagegen inkriminiert zuerst den SP um 1 und kopiert dann den Wert von der Stelle auf welche SP zeigt in das Ziel-Register.

Zudem gibt es noch die RET- und CALL-Instruktionen. RET ist äquivalent zu (POP IP) und CALL ist äquivalent zu PUSH IP; MOV IP <Quellen-Register/Konstante>. Die Notation ist dabei wie folgt:

RET

CALL <Quellen-Register/Konstante>

Fundamental hat der Computer zwei Stacks. Der Supervisor-Stack beginnt bei `0xc7fff` und wird genutzt, wenn die SUP Flagge gesetzt ist. Der User-Stack beginnt bei `0xffff`, also am Ende des Bank Slots 2 und wird genutzt, wenn die SUP Flagge nicht gesetzt ist.



## 2.5 Input/Output

Der Computer hat zudem die Möglichkeit mit verschiedenen IO-Geräte zu interagieren. Die Kommunikation läuft über die sogenannten IO-Ports. Der Computer hat physische Anschlüsse an denen IO-Geräte angesteckt werden können. Die Struktur dieser Anschlüsse sieht wir folgt aus:

Tabelle 4:

#	Funktion
00	GND
01	Vcc
02-17	16Bit Daten
18	Iin-Enable
19	Din-Enable
20	Dout-Enable
21	Clock

Dabei bekommt jedes IO-Gerät eine eigene ID (IO-ID), welche im Bereich von 0 bis 65535 liegen darf, (wobei davon aufgrund von physikalischen beschränkungen vermutlich nur die ersten -zig verwendet werden). Die Interaktion funktioniert dabei über drei Instruktionen, wie in Tabelle 5 zusehen ist.

Tabelle 5:

kurzform	Ausgeschrieben	funktion
Iin	Instruktion Eingang	Computer schreibt eine Instruktion ans I/O-Gerät
Din	Daten Eingang	Computer schreibt 16-Bit Daten ans I/O-Gerät
Dout	Daten Output	Computer liest 16-Bit Daten vom I/O-Gerät

Jede Interaktion benötigt dabei mehrere Schritte, die genaue Definition der Kommunikationsschritte wird dabei durch jede Instruktion eigen festgelegt. Ein allgemeiner standard Ablauf (wie z.B. der der später definierten IO-Instruktion "rmn") ist in Tabelle 6 zusehen.

Tabelle 6:

Bereich	Instr 1	Instr 2	...	Instr n	Instr $n + 1$	...
Iin	1	0		0		
Din	0	1		0		
Dout	0	0		1		
Data0 – 7	InstructionID	Variable		Rückgabewert	Rückgabewert	
Data8 – 15	8b Variable	Variable		Rückgabewert	Rückgabewert	

Die Pins Iin und Din werden dabei durch den Computer für einen Clock-Tick gesetzt, sobald die Daten/instruktionsinformation am Dateneingang an liegen; der Dout Pin wird eine Clock-Cyclus durch das I/O-Gerät selbst gesetzt, wenn die Daten geschrieben werden.

Die Momentan bestehenden I/O-Instruktionen sind in Tabelle 5, aber diese sind im Programm frei veränderlich, solange das externe Gerät die gleiche Interpretation der Anweisungen hat.

Tabelle 7: I/O-Instruktionen

ID	Gerätetyp	Name	Funktion	Verhalten
0x00	All	ls	gibt alle normalen Funktionen des Geräts zurück	Iin(Instruktion); n*Dout(Antworten)

Tabelle 7: I/O-Instruktionen

0x01	Drive	rsn (read singel normal)	Lesen eines einzelnen 16-Bit werts von einer 24-Bit Adresse	Iin(Instruktion); Din(Adresse); 1*Dout(Antwort)
0x02	Drive	rmn (read multiple normal)	Lesen einer reihe von bis zu 65535 16 Bit Werten von 24-Bit Adres- sen aus	Iin(Instruktion); Din(startAdresse); Din(Länge); n*Dout(Antworten)
0x03	Drive	wsn (write singel normal)	Schreiben eines einzelnen 16 Bit werts von einer 24-Bit Adresse	Iin(Instruktion); Din(Adresse); Din(Daten); Dout(bestätigung)
0x04	Drive	wmn (write multiple normal)	Schreiben einer reihe von bis zu 65535 Werten an 24-Bit Adressen	Iin(Instruktion); Din(startAdresse); n*Din(Daten); Dout(bestätigung)
0x05	Drive	rsb (sin- gel read big)	Lesen eines einzelnen 16-Bit werts von einer 40-Bit Adresse	Iin(Instruktion); Din(Adresse); Din(Adresse); 1*Dout(Antwort)
0x06	Drive	rmb (multi- ple write big)	Lesen einer reihe an von bis zu 65535 Werten von 40-Bit Adressen aus	Iin(Instruktion); Din(startAdresse); Din(startAdresse2); Din(Länge); n*Dout(Antworten)
0x07	Drive	wmb (singel write big)	Schreiben eines einzelnen 16-Bit werts von einer 40-Bit Adresse	Iin(Instruktion); Din(Adresse); Din(Adresse);Din(Daten); 1*Dout(bestätigung)
0x08	Drive	wmb (write multiple big)	Schreiben einer reihe von bis zu 65535 Werten 16 Bit werten an 40- Bit Adressen	Iin(Instruktion); Din(startAdresse); n*Din(Daten); Dout(bestätigung)
0x09	Drive	rfn (read file nor- mal)	Lesen einer gesamten Datei (24-bit Namen)	Iin(Instruktion); Din(Name); n*Dout(Antwort)
0x0a	Drive	wfn (write file nor- mal)	Schreiben einer neuen Datei (24-bit Namen)	Iin(instruction); Din(Name); n*Din(daten); Dout(bestätigung)

Tabelle 7: I/O-Instruktionen

0x0b	Drive	rfpb (read file-part big)	Lesen eines teils einer Datei (24-bit Namen)	Iin(instruction); Din(Name); 2*Din(läge); n*Dout(Daten)
0x0c	Drive	wfpb (write file-part big)	Anhängen von Daten an eine Datei(24-bit Namen)	Iin(instruction); Din(Name); Din(länge); n*Din(Daten); Dout(bestätigung)

### 2.5.1 Berechtigungsliste

Der Supervisor-mode hat die berechtigungen zu allen I/O-Interaktionen, für normale Prozesse wird die Berechtigungsliste in einer reihe an RAM-chips gespeichert, deren Adressen sich mit dem wechseln zwischen den verschiedenen Banks ändern. gesetzt werden können sie nur durch den Supervisor, wobei auch dieser die RAM-Adressen nicht manuel ändern kann

Der Supervisor-mode hat die berechtigungen zu allen I/O-Interaktionen, für normale Prozesse wird die Berechtigungsliste in einem separaten Speicher gespeichert, welcher an die BIN des ersten Bank Slots geknüpft ist. Der Supervisor kann nun einzelne IO-IDs frei geben oder diese wieder sperren.

## 2.6 Interrupts

Interrupts sind Unterbrechungen im Programmablauf. Wenn ein Interrupt gemeldet wird, wird dieser im nächsten Instruktions-Zyklus bearbeitet. Für diesen Zyklus, wird, je nach Interrupt-Quelle, nicht die Instruktion ander Stelle von IP sondern von einem IV (Interrupt Vektor) geladen. Es gibt im Computer vier verschieden Interrupt Quellen, wobei jede ein hard-gewired IV hat, dies ist in Tabelle 8 zusehen.

Interrupts können general (sowohl im Usermode als auch im Supervisor-mode) deaktiviert werden, mit Ausnahme von Software-Interrupts, durch das setzen der OINT Flagge. Für ein feinere Einstellung gibt es die OVIOI, OVTI1, OVTI2 Flaggen welche wie in Tabelle 8 zusehen, einnige spezielle Flaggen deaktivieren. Diese findet aber nur im Supervisor-mode

stadt. Im Usermode hingegen lassen sich Interrupts nur über die OINT-Flagge deaktivieren. Alle diese Flaggen befinden sich übrigens in den letzten 8 Bit des FLAG Register, was sie nur für den Supervisor veränderbar macht.

Die INT Instruktion welche keine Operatanten annimmt löst ein Software-Interrupt aus. Dieser kann nicht deaktiviert werden, da dieser Interrupt von laufenden Prozess ausgeführt werden muss und das Prozess somit selber weiß wann und ob es eine Interrupt auslösen soll. Wenn ein Prozess ungestört seine Arbeit machen will, ist er somit selbst verantwortlich ob er eine INT-Operation ausführen soll.

Tabelle 8:

Interrupt-Quelle	IV Name	IV Standard Wert	Flagge zum Deaktivieren
IO-Interrupts	IV-IO	0x3fff8	OVI0I
INT-Instruktion	IV-INT	0x3fffa	
Timed-Interrupts	IV-TI1	0x3fffc	OVTI1
Timed-Interrupts	IV-TI2	0x3fffe	OVTI2

### 2.6.1 Timed-Interrupts

Der Computer besitzt zwei interne Timer je 32 Bit groß. Jeder Timer startet bei 0 und zählt (bei jedem Clock Schlag) um 1 bis zu seinem eingestellten End Wert. Bei Erreichen dieses Werts löst der Timer einen dementsprechenden Interrupt aus.

Jeder Timer kann in drei Modi Arbeit, wie in Tabelle 9 zusehen. Der Timer Modus kann über die TM1 und TM2 (für Timer1 Mode, Timer2 Mode steht) Instruktion gesetzt werden, welcher entweder ein Quellen Register oder eine Konstante annimmt, wobei die Notation wie folgt aussieht:

TM<1/2> <Quellen-Register/Konstante>

Der Endwert wird über die SSTL1, SSTL2 und SSTH1, SSTH2 erledigt. Wobei SSTL<1/2> die ersten 16 Bit und SSTH die letzten 16 Bit des Endwert-Registers setzt. Der Endwert kann dabei als Konstante oder als Quellen-Register vorliegen. Die Notation ist dabei wie folgt:

SSTL<1/2> <Quellen-Register/Konstante>

SSTH<1/2> <Quellen-Register/Konstante>



Tabelle 9:

Timer Modus	Modus ID	Funktion
Off	0x0	Timer ist Inaktive
Continues	0x1	Timer zählt hoch; bei erreichen des Ziels, löst Interrupt aus und fängt von vorne an zuzählen.
Singel	0x2	Timer zählt hoch; bei erreichen des Ziels, löst Interrupt aus und schalte in Off Modus um

Die aktuell Clock Geschwindigkeit, ist in mit CFL und CFH instruktion zu bekommen. CFL lädt die ersten 16 bit der Clock Geschwindigkeit in das Ziel-Register und CFH die letzten 16 bit in das Ziel-Register. Die Frequenz ist dabei In Herz (Schläge pro Sekunde) angegeben. Die Notation ist wie folgt:

CFL <Ziel-Register>

CFH <Ziel-Register>

### 2.6.2

Auch IO-Geräte können als Interrupt-Quelle fungieren. Dabei gibt es keine Möglichkeit, vom Computer her Interrupt eines IO-Geräts zu deaktivieren oder aktivieren. Dies muss gegebenenfalls über das IO-Protokoll, siehe Unterabschnitt 2.5, gemacht werden. Es können natürlich alle IO-Interrupts auf einmal mit der OVIOI-Flagge gemacht werden.

## 3 Kommentar zur Specification

### 3.1 16 bit Daten/Address Länge

Die Address- und Datenlänge besagt viele Binäre Ziffern (bits) für eine Address 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 Binary Ziffern (= Bits) ist und wenn man mit 0 anfängt zu zählen. Sie Tabelle 10

Ein eine  $n$  stellige Binär Zahl wobei  $n$  teilbar durch 8 ist, ist dabei relative angenehm, 8 bit = 1 byte. Das byte ist dabei die Basis Einheit für so gut wie alle informationstechnischen Standards. Daher ist es gute eine solches  $n$  zu wählen. Zudem kommen einige elementar ICs, z.B. Buffer oder Register, immer mit 8 bits.

Jedoch gilt je mehr Bits man verwendet, desto mehr Schaltung brauchen man, desto mehr Stromverbrauch. Da wir wie zwar nicht speziell auf Platz und Stromverbrauch optimieren, aber trotzdem ein limitiertes Buget haben, können wir auch nicht eine beliebig große Daten/Address Länge wählen.

Wir haben uns daher für 16 bit entschieden, der Codea man laut Tabelle 10 Zahlen bis zwischen 0 und 65535 darstellen kann. Diese Menge ist gerade große genug, um für die meisten Programme große genüge Zahl abzubilden <sup>3</sup>.

Die Entscheidung zur größe der Daten/Address Länge ist eine sehr ausschlag gebende, daher haben wir diese zuerst festgelegt. Sie wird für viele weitere Entscheidung eine wichtige Rolle spielen. Wichtig ist nur das wir maximal 65536 Zustände oder Zahlen von 0 bis 65535 darstellen können mit unseren 16 bit

Tabelle 10: Maximalerepersäsentierbare Zahlen (startend mit 0)

$n$	$n^2 - 1$
2	3
4	15
8	255
16	65535
32	4294967295
64	18446744073709551615

## 3.2 Register Anzahl

Der Computer hat 8 Register. Wir haben uns für Acht ausfolgenden Gründen entschieden: Zu erst die minimale Anzahl anregisteren die wirklich gebraucht würde, ist 5! Man braucht mindestens zwei all Zweck Register um binäre Rechenoperation durch zu führen, einen Stackpointer, einen Instruktion Point und einen Ort wo man die Flaggen speichern kann <sup>4</sup>.

<sup>3</sup>Für Programme die Zahlen welche  $> 65535$  darstellen wollen ist es ratsam immer zwei oder mehr Spreichereinheiten zunehmen oder bei Berechnungen, selbige auf zwei oder mehr Schritte auf zuteilen

<sup>4</sup>Theorthisch ist es möglich das Flaggen Register wegzulassen, wir haben uns aber für eine solches Register entschieden siehe ??

Fünf lässt sich nicht mehr mit 2 bits repräsentieren ( $2^2 = 4$ ), daher muss man 3 bit ( $2^3 = 8$ ) nehmen. Der Grund warum wir nicht mehr genommen haben hat mit der Encodierung von Instruktionen zutun siehe ?? . Somit haben sich 8 Register als das Optimum ergeben

### 3.3 Gedanken Hinter dem Supervisormode

Wir haben uns entschieden eine Artberechtigungs System in der Computer einzubauen. Eine Rechner der nur einfache Programm ausführt um zu Beispiel eine LED oder ein LCD-Display zusteueren, wie es z.B. ein ESP32 oder ein ATmega328p macht, ist so ein System nicht nötig. Wir hatten jedoch vor mit unserem Computer die Möglichkeit zuschaffen Betriebssystem ähnliches Programm (Supervisor) laufen zulassen.

Hier für hat man einmal den Supervisor, der bei Start des Computer anfängt zulaufen, und Programme, welche im Usermode ausgeführt werden, sogenannte Prozesse. Dabei sieht ein typischer Programmablauf wie folgt aus:

Wichtig ist, das sobald ein Prozess benendet wurde, der Supervisor an dieser Stelle wieder anfängt und den nächsten Prozess started.

Ein Prozess kann auch Funktionen des Supervisor nutzen über Software Interrupts, da bei einem Interrupt, der Computer in den Supervisormode wechselt. So kann der Prozess, bevor er den Interrupt auslegt, ein oder mehrer Register nutzen um Werte an den Supervisor zu übergeben. Z.B. kann er eine Register nutzen um dem Supervisor zusagen was er tun soll, z.B. dem Prozess zugang zu einem IO-Gerät zugeben oder etwas in eine Datei zuschreiben. Diese Art von Kommuntiation werden auch System-Calls genannt und sind die Basis für jedes moderne Betriebssystem.

Ein solcher Supervisor kann sehr mächtig sein:

1. Abstraktion der Hardware: Oft ist die IO-Configuration von Setup zu Setup sehr verschieden. Ein Prozess was IO-Nutzen möchte hat oft den Nachteil dass es für ein bestimmte Hardware setup geschrieben wurde. So z.B. ein Prozess was Daten einer SD-Karte auslieft und diese anzeigt. Wenn jedoch man jetzt eine Floppy-Laufwerk anschließt, so kann man das Prozess zum Auslesen nicht verwenden, da es nur die Kommuntiation mit einer SD-Karte kennt. Der Supervisor könnte an dieser Stelle ein Interface über System-Calls bieten welches unabhängige von dem Gerät ist. So im diesem Beispiel ein Dateisystem, auf welches Prozesse zugreifen können. Somit übernimmt der Supervisor die Kommuntiation mit der Hardware.
2. Teilung der Rechenzeit: Der Computer so viele Features er doch hat, ist stark limitiert vorallem was Rechengeschwindigkeit betrifft. Wenn man nun mehrer Pro-

programm hat, welche ausgeführt werden sollen, so wäre es natürlich sehr einfach sie der Reihe nach auszuführen. Jedoch gibt es bessere Wege, so z.B. das der Computer jedem Prozess nur eine Bestimmte Rechenzeit gibt und dann den nächsten Prozess ausführt. Das kann erreicht werden durch folgenden Ablauf: Der Supervisor setzt vor dem ausführen des Prozesse eine Timed-Interrupt. Dann übergibt der Supervisor den Computer dem Prozess, welcher dann seine Programm ausführt. Dies macher bis der Timed-Interrupt ausgelöst wird und der Computer wieder in den Supervisormode wechselt. Dieser speichert nun alle Register und Banken Configuration des Prozesse und übergibt einem anderen Prozesse den Computer. Wenn nun ein schon angefangen Prozess wieder gestartet werden soll läd der Supervisor wieder die Register und Banken des Prozesse und übergibt die Außführung wieder an den Prozess. Somit kann der Computer zwischen Prozessen hin und herschalten, was die Illusion gibt, das der Computer diese gleichzeitig ausführt.

3. Isolation von Prozessen: Programme sind meist nie perfekt. Vorallem sowas wie Bufferoverflows und andere Fehler dieser Art treten in Low-Level Systeme gerne auf. Aus diesem Grund ist es wichtig Prozesse von einander zu isolieren. Nehmen wir an Prozess A hat ein Fehler welcher dafür sorgt das er den gesamten RAM anfängt voll zuschreiben. Um jetzt Prozess A daran zuhinern Prozess B zu überschreiben, ist es dem Prozess nicht gestattet auf den Arbeitsspeicher von Prozess B zuzugreifen. Nur der Supervisor kann zwischen den Prozessen wechseln <sup>5</sup>.

Wenn man so etwas will kann das auch auf einem ATmega328p gemacht werden, jedoch nicht ohne massive Leistungs Verluste. Wenn so ein Berechtigungssystem jedoch direkt in die Hardware des Computers einfließt sorgt das für eine viel schnellern Computer.

Wenn man tatsächliche nur Programme schreiben will welche iür ein LED oder LCD-Display steuern soll, kann man das mit unserem Computer auch machen, indem man einfach diese Programm im Supervisor laufen lässt.

---

<sup>5</sup>Natürlich kann auch der Supervisor fehlerhaft sein. Jedoch hatten wir bei dieser Maßnahme nicht nur fehlerhaft sondern auch schlicht böswillige Prozesse im Kopf. Natürlich ist dies unwahrscheinlich dass so etwas jemals eintreten wird, jedoch war unsere Idee von Anfang an eine Computer zu bauen der im Internet stehen könnte und die dementsprechenden Sicherheitsmaßnahmen hat