Elektronikpraktikum – Versuch 8: Mikroprozessor

Jonas Wortmann*[1] and Angelo V. Brade†[1]

[1]Rheinische Friedrich-Wilhelms-Universität Bonn

20. September 2024

*s02jwort@uni-bonn.de
†s72abrad@uni-bonn.de

Inhaltsverzeichnis

1 Einleitung

In diesem Versuch werden wir einen Prozessor angefangen von einigen wenigen logischen Verknüpfunkt bis zu einem vollständig mit Assambly programmierbaren 8080 Mikroprozessor verstehen. Dafür wird zunächst die Arithmetic Logic Unit und dann dessen Ereweiterungen betrachtet. Zum Schluss führen wir ein Programm zu Multiplikation von Zahlen aus.

2 Theorie

2.1 Grundlagen

Als Grundlage unserer logischen Operationen dient, wie schon zuvor untersucht und verstanden, die Binärarithmetik. Es lassen sich z.B. Zahlen einfach addieren, wobei auf ein Überlauf geachtete werden muss. Ein Überlauf dritt dann auf, wenn das Ergebnis größer, als die Speichergröße der Zahl ist. Es lässt sich auch eine Subtraktion durch umwandlung des Subtrahenden in eine Addition überführen. Dafür wird der Subtrahend bitweise invertiert und von dem gesammten Subtrahenden 1 abgezogen. Im allgemeinen lassen sich mit Binärzahlen aber genauso rechnen, wie wenn man die Dezimalbasis, statt der Binärbasis, benutzt. Neben den Binärzahlen werden auch Hexadezimalzahlen verwendet, da diese sich mit genau 4 Bits darstellen lassen. Durch die verkürtzte schreibweise werden sie zur Datenspeicherung verwendet. So können auch zwei Hexadezimalzahlen einen Byte (8 Bit) bilden.

2.2 Arithmetic Logic Unit

Die Arithmetic Logic Unit (Abb. 1), kurz ALU, ermöglich die Verknüpfung von AND, OR und XOR, sowie die Verknüpfungen des integrierten 8-Bit-Addier-Subtrahierwerks (Abb. 2).

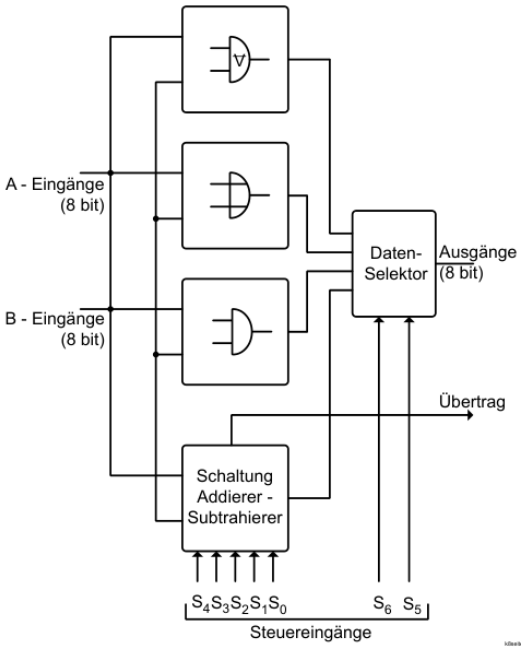

Abbildung 1: Arithmetic Logic Unit

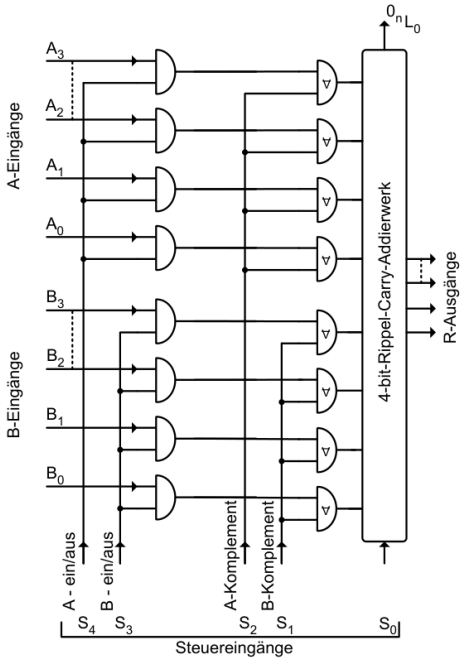

Abbildung 2: 8-bit Addier-Subtrahierwerk

2.3 Akkumulator

Erweitert man den ALU mit einem ROM, sowie einem Register und einer Übertragsflag, so erhält man den Akkumulator (Abb. 3). Der Akkumulator kann mithilfe des Registers Ergebnisse zwischenspeichern. Tritt eine Übertrag

auf, so wird er in der Übertragsflag gespeichert. Der Rom dient der Umkodierung, der Befehle, da von den 32 möglichen Funktionen (Abb. 4) nur 13 (Abb. 5) sinnvoll sind.

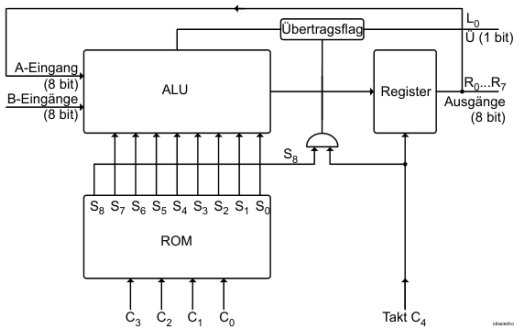

Abbildung 3: Akkumulator

S_4	S_3	S_2	S_1	S_0	Ausgangsfunktic
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	−1
0	0	0	1	1	0
0	0	1	0	0	−1
0	0	1	0	1	0
0	0	1	1	0	−2
0	0	1	1	1	−1
0	1	0	0	0	B
0	1	0	0	1	B+1
0	1	0	1	0	$-B-1 = \overline{B}$
0	1	0	1	1	−B
0	1	1	0	0	B−1
0	1	1	0	1	B
0	1	1	1	0	−B−2
0	1	1	1	1	$-B-1 = \overline{B}$
1	0	0	0	0	A
1	0	0	0	1	A+1
1	0	0	1	0	A−1
1	0	0	1	1	A
1	0	1	0	0	$-A-1 = \overline{A}$
1	0	1	0	1	−A
1	0	1	1	0	−A−2
1	0	1	1	1	$-A-1 = \overline{A}$
1	1	0	0	0	A+B
1	1	0	0	1	A+B+1
1	1	0	1	0	A−B−1
1	1	0	1	1	A−B
1	1	1	0	0	B−A−1
1	1	1	0	1	B−A
1	1	1	1	0	−A−B−2
1	1	1	1	1	−A−B−1

Abbildung 4: 8-bit Addier-Subtrahierwerk Befehle

C_3	C_2	C_1	C_0	Funktion
0	0	0	0	A
0	0	0	1	1
0	0	1	0	$\overline{A}$
0	0	1	1	B
0	1	0	0	0
0	1	0	1	A+1
0	1	1	0	A−1
0	1	1	1	A+B
1	0	0	0	A−B
1	0	0	1	$A \wedge B$
1	0	1	0	$A \vee B$
1	0	1	1	$A\overline{\vee}B$
1	1	0	0	−1
1	1	0	1	
1	1	1	0	
1	1	1	1	

Abbildung 5: Sinnvolle Befehle des ALUs

2.4 Akkumulator mit Datenspeicher

Fügt man dem Akkumulator einen Datenspeicher, hier RAM, hinzu, können jetzt auch zwischenergebnisse nicht nur abgespeichert, sondern auch wieder ausgelesen werden. Dieser ist in Abb. 6 zu sehen.

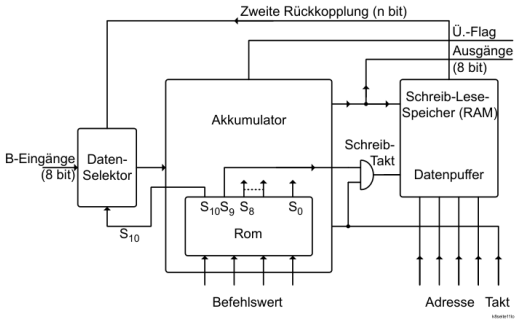

Abbildung 6: Sinnvolle Befehle des ALUs

2.5 Prozessor

Um nun die Konstruktion zu einem Prozessor (Abb. 7) zu vervollständigen, wird ein Programmspeicher mit einem Befehlszähler hinzugefügt. Aus dem Programmspeicher werden mithilfe des Befehlszählers nacheinander die Befehle des Programms aufgerufen. Ist der Befehl ausgeführt, so wird der Befehlszähler um Eins hochgeschalten. So ein Prozessor bildet der Mikroprozessor 8080.

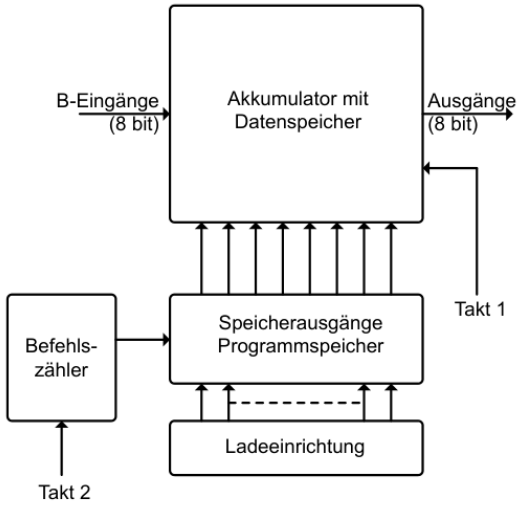

Abbildung 7: Sinnvolle Befehle des ALUs

3 Voraufgaben

A

Die folgenden Dualzahlen, lassen sich in die entsprechenden Hexadezimal- und Dezimalzahlen umwandeln:

Binär	Hexadezimal	Dezimal
1101 1111 0010 1110	DF2E	57134
1111 1111	FF	255

B

Die Zahl 2115_{10} ist auch 100001000011_2 oder 843_{16}.

C

Die Zahl $B75F_{16}$ ist auch 1011011101011111_2 oder 46943_{10}.

D

Es können die folgenden Rechenoperationen durchgeführt weden.

$$\begin{array}{r} 01011011 \\ +01101011 \\ \hline 11000110 \end{array} \qquad \begin{array}{r} 011111111 \\ +000000001 \\ \hline 100000000 \end{array}$$

$$\begin{array}{r} 11000000 \\ -10110101 \\ \hline 00001011 \end{array}$$

Bei Zweierkomplementzahlen gibt die erste Stelle mit 1 an, ob sie negativ, oder mit 0 and, ob sie positiv ist.

Zahl	Vorzeichen
10110111	-
11110000	-
01111111	+
11111111	-

Für die Konvertierung von einer normalen Binärzahl zu einer Zweikomplimentzahl, wird zuerst die Binär zahl invertiert und dann mit 1 addiert. So lassen sich subtraktion mit Zweierkomplementzahlen durch addition darstellen.

$$\begin{array}{r} 11011011 \\ -01101011 \end{array}$$

wird so zu

$$\begin{array}{r} 11011011 \\ +10010101 \\ \hline 101110000 \end{array}$$

Ein Überlauf tritt dann auf, wenn zwei Zahlen addiert werden, wobei das Ergebnis größer ist, als die bereitgestellte Speichergröße. Passiert dies, wird der resultierende zusätzliche Bit nicht gespeichert und es scheint so, als ob die Zahl wieder um 1 plus die maximale Zahl, die in dem Speicher gespeichert werden kann, verringert wird.

Eine Übetrag passiert dann, wenn bei einer Addition der nächste Bit verwendet werden muss. So führt nur die Addition von Zahlen, die an der gleichen Bit-Stelle eine 1 haben zu einem Übertrag.

$$\begin{array}{rrr} 11011111 & 01011011 & 01011011 \\ +00111000 & -10111011\hat{=} & 01000101 \\ \hline 100010111 & & 10100000 \end{array}$$

Bei der ersten Operation entseht ein Überlauf, wobei die Zahl mit einem 9-bit Rechner immernoch mit -23_{10} immernoch im Definitionsbereich eines 8-bit Rechners wäre. Die zweite Operation ist mit -32_{10} hat keinen Überlauf und ist immernoch im Definitionsbereich eines 8-bit Rechners.

Es lassen sich auch Produkte berechnen: 1101 · 1001=1110101.

Genauso auch Quotienten: 1110111/101=10111 mit Rest 100

E

ROM steht für Read-Only-Memory und ist ein Speicher der, wie der Name wage ahnen lässt, nur gelesen werden und nur einmal beschrieben werden kann. RAM steht für Random-Access-Memory und ist ein Speicher, der beliebig gelesen und beschrieben werden kann.

F

Um analoge Signal in Digitale umzuwandeln benötigen wir einen DAC (Digital Analog Converter). Dieser ist durch seine Bitzahl limitiert.

G

Ein ALU hat verschiedene logische Verknüpfungen, wie z.B. AND, OR, XOR und noch grundleregendere, wie z.B. $A + 1$. Erweitert man diesen mit einem Register, einer Übertragungsflagge und einem ROM, so erhält man ein Akkumulator. Mithilfe von einer Taktung kann nun das Ergebniss gespeichert werden und ist somit kein statisches Netzwerk.

H

Erweitert man den Akkumulator mit einem RAM und Programmspeicher, so erhält man einen vollständigen Rechner.

I

Hat ein Rechner ein dedizierten Daten-Bus, Adressen-Bus und Steuer-Bus, so hat er eine sog. Busstruktur. Er rechnet dabei mit Bit in Binärzahlen oder Hexadezimalzahlen für die Befehle.

J

Ein Taktzyklus durchläuft ein Takt der Clock. Ein Befehlszyklus sind alle Takte, die benötigt werden, um einen Befehl auszuführen. Ein Operationszyklus sind alle Takte, die benötigt werdne, um eine Operation durchzuführen.

K

Der 8080-Mikroprozessor kann mit 8 Bits 2^8 also 256 Befehle speichern. Hierbei hat jeder Befehl ein bestimmte Zahl. Diese Zahl wird Operations-Code genannt. Allerdings hat der 8080 nur 115 Befehle.

L

Ein Zweiweg-Tri-state-Datenbus hat nicht nur die Zustände 1 und 0, sondern auch Z, der sog. hochohmige Zustand. Er signalisiert, dass es sich werder noch um 1 oder 0 handelt.

M

Die Länge eines Befehlszyklus hängt davon ab, mit wie vielen Takten der Befehl geholt, decodiert und dann ausgeführt wird. Beim ausführen wird ein Operationszyklus durchlaufen, weshalb der Befehlszyklus auch von ihm abhängt.

N

Der DMA (Direct Memory Access) ermöglicht den Zugriff von Befehlen auf abgespeichert Daten und kann so wiederverwendbare Ergebnisse erreichen, die sonnst erneut berechnet weden müssen. Somit wird weniger Rechenzeit beansprucht.

O

Der Befehlszähler zählt die Befehle und weiß so die Zahl, des als nächstes auszuführenden Befehls. Der Stackpionter speichert die Adressen für abgespeicherte Werte in dem Stack.

P

Der Stack speichert Werte von Größen, die schon zu kompilierzeit bekannt sind. Wie der Name schon suggeriert handelt es sich um ein Stapel an Befehlen. Von diesem Stapel können nur von unten oder oben Befehle hinzugefügt oder entfernt weden. So werden dort Werte die schon beim kompilieren bekannt sind dort gespeichert, sowie Werte dessen benötigte Speichergröße bekannt sind, aber erst beim Ausführen ermittelt werden, dort gespeichert. Während das Programm läuft lässt kein weiterer Speicher mehr reservieren.

Q

Es gibt die volgenden Adressierungsarten: Unmittelbare Adressierung: Der Befehl hat den Wert selber. Direkte Adressierung: Der Befehl kennt direkte Adresse des Werts. Indirekt Adressierung: Der Befehl kennt die Adresse des Werts durch den Stackpointer. Relative Adressierung: Der Befehl kennt durch das Addieren von einer bestimmten Größe die Adresse in Abhängigkeit eines Ausgangpunktes (Arrays auf dem Stack). Indizierte Adressierung: Der Befehl kennt durch Indexen die Adressen einer Reihe an Werten.

R

Der 8080 hat die folgenden Operationszyklen Befehlsaufruf, Speicher lesen, Speicher schreiben, Stack lesen, Stack einschreiben, Eingabe, Ausgabe, Unterbrechung und halten.

S

Der erste Operationszyklus liest den Befehl ein, damit er dann mit dem Befehlregister decodiert werden kann.

4 Auswertung

4.1 Versuchsaufgabe 1

Zu Beginn des Versuches wird das System 0 ausgewählt und der FPGA springt in den Addier–Subtrahierwerk Modus. Mit den Eingängen S_0 bis S_4 können verschiedene Operationen mit den Registern A und B durchgeführt werden. Diese Operationen sind in (2) dargestellt.

Die einzelnen Bits der Register bzw. des Steuerwerks werden mit Tastern auf dem Schaltbrett gesetzt und in der LED–Matrix angezeigt. Auf der LED–Matrix ist C das Steuerregister S. R ist das Result Register. Sollte eine negative Zahl angezeigt werden, dann schaltet die flag–LED an und die zahl wird im zweierkomplement angezeigt. Einige Beispiele sind:

- $A = 2_{10} = 10_2$, $B = 1_{10} = 1_2$ und $C = 11111_2$: $R = -5_{10} = 11001_2$ mit flag gleich 1. Der Befehl ist –A–B–1.
- $A = 1_{10} = 1_2$, $B = 1_{10} = 1_2$ und $C = 11000_2$: $R = 2_{10} = 10_2$. Der Befehl ist A+B.

4.2 Versuchsaufgabe 2

Der Mikroprozessor wird auf System 2 eingestellt. Diese Konfiguration ist das Addier–Subtrahierwerk mit einer ALU. Es wird getestet für $A = 01111010_2$ und $B = 10000111_2$ Hier ist die Steuerung abhängig von C_0 bis C_3.

- Der Befehl $\overline{A}$ ist 0100_2. Daraus folgt $R_A = 10000101_2$ und $R_B = 01111000_2$, was genau das Komplement von A bzw. B ist.
- Der Befehl $A \wedge B$ ist 1001_2. Daraus folgt $R = 000000010_2$. Dieses Ergebnis kommt durch das bitweise vergleichen von A und B mit einem logischen AND.
- Der Befehl $A \vee B$ ist 0101_2. Daraus folgt $R = 111111111_2$. Dieses Ergebnis kommt durch das bitweise vergleichen von A und B mit einem logischen OR.

B FEHLT

4.3 Versuchsaufgabe 3

Unter Verwendung von System 2 können Operationen auf zwei oder mehr Zahlen angewandt werden, indem sie im RAM gespeichert werden. Zu erst wird eine Zahl in das B Register geschrieben, dann mit einem Takt in den Akku geladen. B wird nun am R Register angezeigt. Wird eine zweite Zahl in B geschrieben, so kann sie mit dem passenden Steuerbefehl addiert werden. Für jede Operation wird auch getaktet. Das Ergebnis lässt sich dann am R Register ablesen.

Rechnet man $B1_{16} - 18_{16}$ geht man nach dem oben beschriebenen Prozedere vor. Das Ergebnis ist $99_{16} = 153_{10}$. Für die Rechnung $22_{16} - 08_{16} + 1_{16} + 10_{16}$ wird das selbe Verfahren verwendet. Da das Ergebnis direkt in R gespeichert bleibt, kann die nächste Zahl sofort addiert bzw. subtrahiert werden, indem sie über B eingegeben und dann getaktet wird.

4.4 Versuchsaufgabe 4

Der RAM ist dafür da, um als Puffer zu agieren. Er speichert Daten, die für spätere Berechnungen benötigt werden und erspart so das wiederholte Eingeben von Zahlen. Zudem können aus ihm auch Daten gelesen werden, was es möglich macht mehrmals oder zu verschiedenen Zeitpunkten auf Daten zuzugreifen. All dies ermöglicht es komplexere Schaltungen zu bauen, in denen viele Zahlen / Daten zu unterschiedlichen Zeitpunkten benötigt werden.

Der Programmspeicher wird dafür verwendet, um Programme bzw. Befehlsfolgen zu speichern und zu anderen Zeitpunkten auszuführen. Zudem benutzt man ihn, um Befehle einzulesen und in der richtigen Reihenfolge auszuführen.

4.5 Versuchsaufgabe 5

Zu erst wird ein Sägezahlsignal in den Mikroprozessor eingegeben, indem er mit einer Tastatur verbunden wird und die Befehle in Hexadezimal eingegeben werden. Dieses ist in Abb. 8 gezeigt.

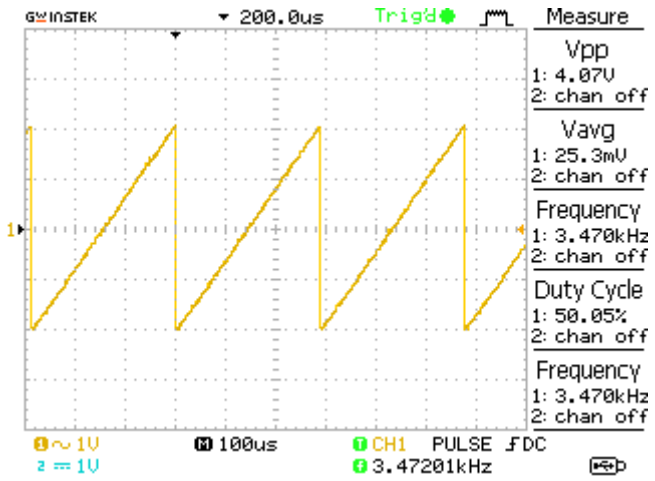

Abbildung 8: Sägezahn-Signal

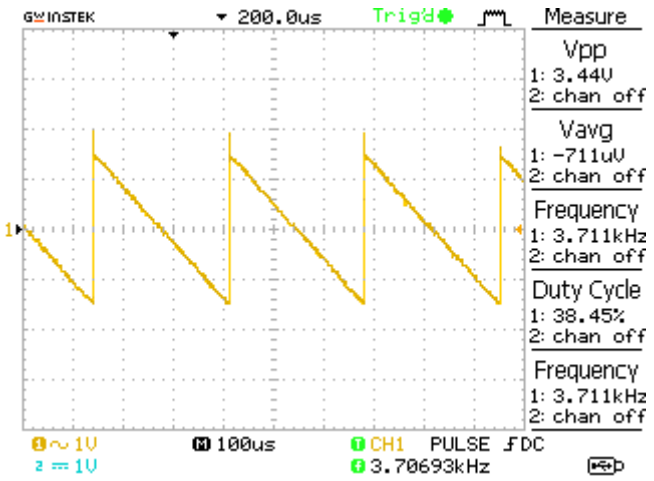

Abbildung 9: Lange Periode

Die Periodendauer des Signals ist $T = \frac{1}{f} = 288\,\mu s$. Das Programm funktioniert so, dass es das Register A anfangs mit Nullen füllt und dann mit jedem jump um eins erhöht. Erreicht A die maximale Größe, so wird es wieder mit Nullen gefüllt, da der nächst größere Bit Überschüssig ist und nicht mehr gespeichert werden kann. Dadurch entsteht eine ansteigende Rampe und ein instantaner Abfall zurück auf Null.

Das Programm kann nun so verändert werden, dass die Periodendauer über das Register B bestimmt wird. Dies wird über den Code in Tab. 1 erreicht. Zu Beginn wird das Input B-Register eingelesen und gespeichert, sodass bei der Schleife, die hier mit F1 makiert ist, der Iterator immer auf den die gewünschte Größe zurückgestellt wird, sobald er Null erreicht.

Befehl	**Hexadzimal**
IN 01	DB 01
MOV A, B	47
MVI A, 0	3E 00
F1: INR A	3C
OUT 04	D3 04
CMP B	B8
JNZ F2	C2 0E 00
MVI A, 0	3E 00
F2: JMP F1	C3 05 00

Tabelle 1: Variables Sägezahn

Als Singal erhalten wir die folgenden Abb. 9, 10 und 11.

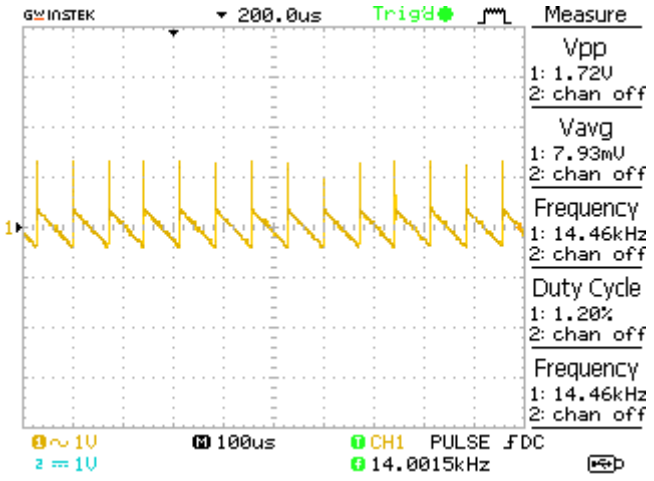

Abbildung 10: Mittlere Periode

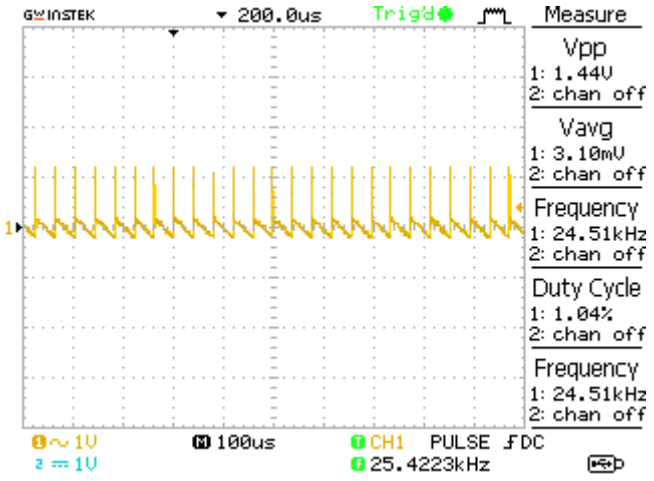

Abbildung 11: Kurze Periode

Nun lassen wir uns unserer Fantasie freien lauf und erschaffen ein Rechteck-Signal. Der Code ist unter Tab. 2 zu

finden. Der Code ist ein variable einstallbares Rechtecksignal, welches das D-Register nutzt um es abwechselnd auf 0 und 1 zusetzten. Durch diese Alternation wird abwechselnd SIG_ON und SIG_OFF aufgerufen, welches das Ausgangssignal auf 0 bzw. 255 setzt.

Befehl	**Hexadzimal**
MVI D, 1	16 01
F1: IN 01	DB 01
MOV B, A	47
MOV C, B	48
F2: DCR C	0D
JNZ F2	C2 06 00
MOV A, D	7A
ADI 0	C6 00
CZ SIG_ON	CC 18 00
CNZ SIG_OFF	C4 1D 00
OUT 04	D3 04
JMP F1	C3 02 00
SIG_ON: MVI A, 255	3E FF
MVI D, 1	16 01
RET	C9
SIG_OFF: MVI A, 0	3E 00
MVI D, 0	16 00
RET	C9

Tabelle 2: Rechteck-Signal

Wir erhalten somit die in Abb. 12 gezeigte Signal.

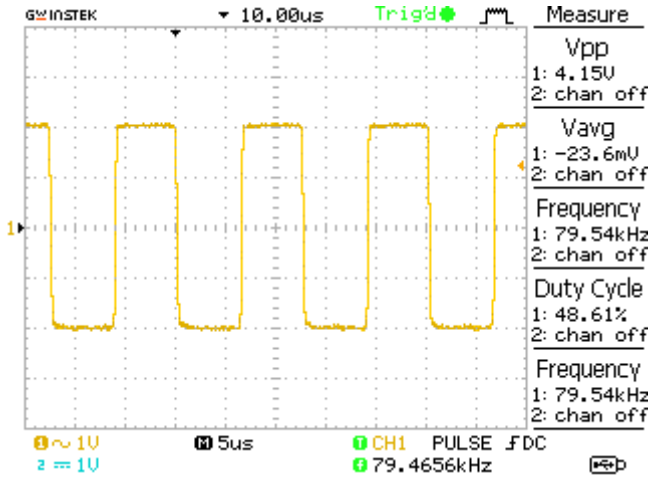

Abbildung 12: Einstellbares Rechtecksignal

Nun soll am X-Register gezeigt werden, ob der 4te Bit am B-Register an oder aus ist. Dafür lesen wir das B-Register ein und können mit einer AND Operation mit der Zahl 8 (4ter bit an) nur den 4ten Bit errausfiltern. Wir subtrahieren dann 8. Sollte der 4te Bit an gewesen sein, so wird das komplette Register nun 0 sein und die Zero-Flag erschein. Ist der 4te Bit aus, so erhalten wir $0 - 8 \neq 0$, womit die Zero-Flag nicht erscheint. Anhand der Zero-Flag können wir dann an dem X-Register 255 (alle 8 Bit leuchten) oder 0 (kein Bit leuchtet) ausgeben. Der verwendetet Code kann in Tab. 3 eingesehen werden.

Befehl	**Hexadzimal**
IN 01	DB 01
ANI 8	E6 08
CPI 8	FE 08
CZ SET_1	CC 10 00
CNZ SET_0	C4 13 00
MOV A, C	79
OUT 00	D3 00
HLT	76
SET_1: MVI C, 255	0E 01
RET	C9
SET_0: MVI C, 0	0E 00
RET	C9

Tabelle 3: Überprüfe 4ten Bit

Der Code wurde erfolgreich in dem Versuch mit verschiedenen Kombinationen getestet.

Nun war es die Aufgabe eine Multiplikation zu implementieren. Dafür gab es mehrere Möglichkeiten. Den Code wie die schriftliche Multiplikation zu konstruieren erfordert mehr Programmier-Aufwand, ist aber für große Operation relativ schnell. Einfach die Zahl mit sich selber so oft auf zu addieren, wie der Multiplikant groß ist, beansprucht weniger Programmier-Aufwand, ist aber dafür für große Operationen langsam. Da es sich bei uns um eine 8 Bit Operation handelt und somit die größte Zahl 255 ist, haben wir uns für letzteres entschieden. Somit wird genau dies im Code von Tab. 4 verwirklicht. Das Programm macht genau das wie es schon beschrieben ist. Er liest A und B-Register ein und führt dann so groß, wie das B-Register ist, eine addition von dem A-Register auf sich selber aus. Dies wird allerdings mit einer Doppelgenauigkeit getan, weil das Ergebnis durchaus bis zu 16 Bit beanspruchen kann.

Befehl	**Hexadzimal**
F1: MVI L, 0	2E 00
MVI H, 0	26 00
MVI D, 0	16 00
IN 00	DB 00
MOV E, A	5F
IN 01	DB 01
MOV C, A	4F
INR C	0C
F2: DCR C	0D
JZ F3	CA 15 00
DAD D	19
JMP F2	C3 0D 00
F3: MOV A, H	7C
OUT 00	D3 00
MOV A, L	7D
OUT 01	D3 00
JMP F1	C3 00 00

Tabelle 4: Multiplikation

Die Programm wurde für verschiedene Zahlen erfolgreich im Versuch getestet.

5 Fazit

In diesem Versuch wurde nach und nach aus elementaren Bauteilen ein Mikroprozessor entworfen. Alle Teile wurden einzeln auf ihre Funktionen mit verschiedenen Dualzahlen überprüft. Die Bauteile selbst wurden nicht aufgebaut, stattdessen werden sie von einem FPGA simuliert.

Zuerst wird ein Addier–Subtrahierwerk eingestellt mit dem erfolgreich verschiedene Zahlen addiert und subtrahiert werden können. Dann schaltet man eine ALU dazu um logische Operationen zwischen zwei Zahlen auf verschiedenen Registern durchzuführen. Mit Hilfe eines RAM konnten Zahlen gespeichert werden und größere bzw. längere Operationen waren möglich. Mit dem Programmspeicher konnten dann Befehlsfolgen über eine Tastatur an den Mikroprozessor gegeben werden, mit denen unter anderem verschiedene Signale, wie ein Sägezahnsignal, aber auch die Multiplikation zweier Dualzahlen möglich war.