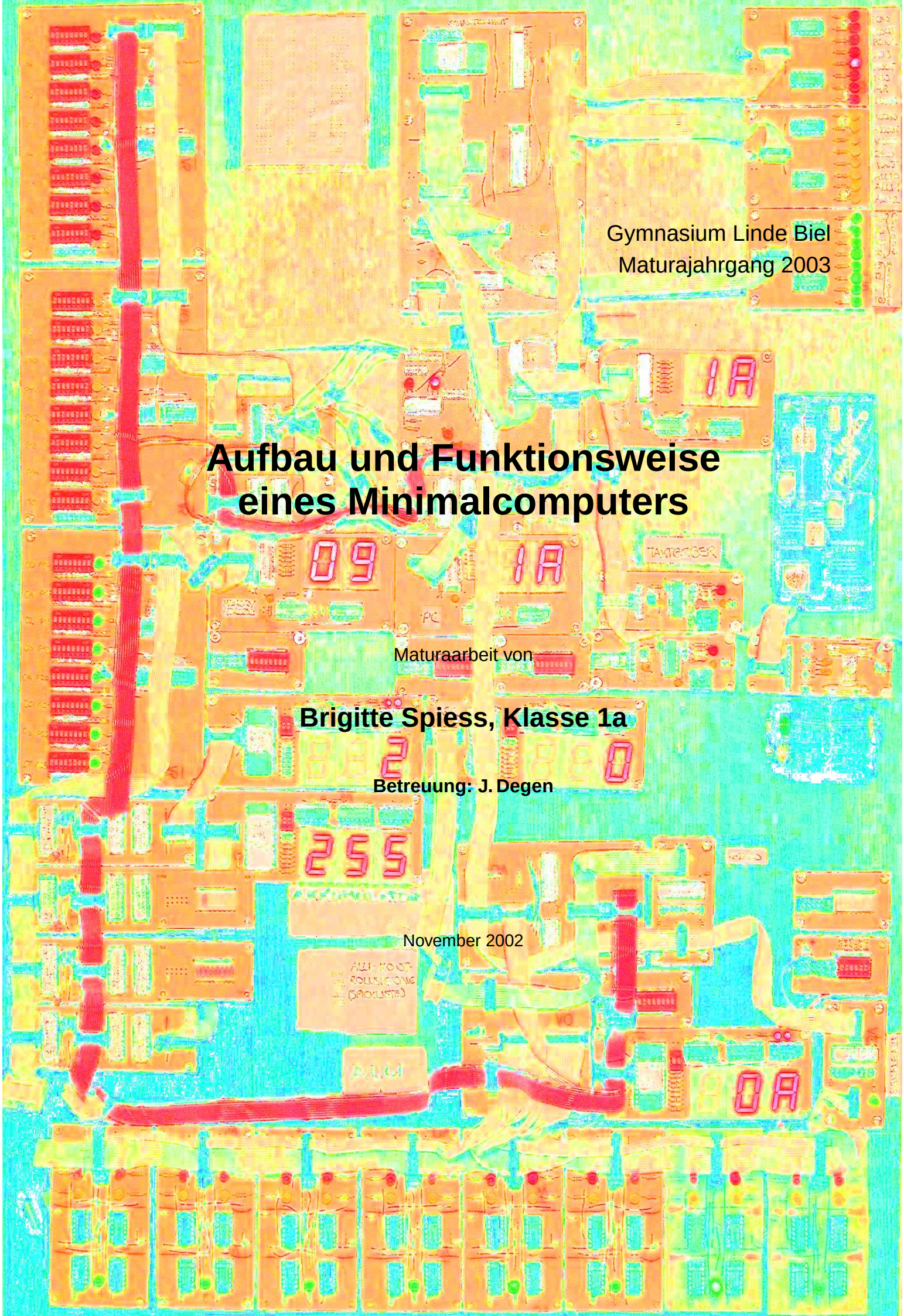


Gymnasium Linde Biel
Maturajahrgang 2003

Aufbau und Funktionsweise eines Minimalcomputers

Maturaarbeit von

Brigitte Spiess, Klasse 1a

Betreuung: J. Degen

November 2002

Vorwort

Kann man es sich heute noch leisten, nicht zu wissen, wie mit einem Computer umzugehen ist?

Ich denke nicht. Zwar muss man im Alltag nur wissen, wie der Computer zu bedienen ist, aber wie ein Computer funktioniert, hat keinerlei Bedeutung. Viele Personen, so auch ich, benützen den Computer täglich, haben aber keine Ahnung, was dahinter steckt.

Als es um die Auswahl eines Themas für die Maturaarbeit ging, diskutierte ich wieder einmal mit meinem Vater darüber, dass zwar alle den PC brauchen, aber keiner wirklich versteht, wie er oder auch nur ein ganz einfacher Computer funktioniert. Im Verlauf von diesem Gespräch entschied ich mich, dies für mich zu ändern. So kam es, dass ich als Thema für meine Maturaarbeit “Aufbau und Funktionsweise eines Minimalcomputers” wählte.

Mit meiner Maturaarbeit wollte ich die Funktionsweise eines einfachen Computers von unten her verstehen lernen, also aufbauend auf die Transistorlogik. Ich wollte aber nicht nur theoretisch verstehen, wie ein Computer funktioniert, nein, ich wollte auch praktisch soweit wie möglich einen Minimalcomputer bauen. So entstand mit der Zeit “MINICOMP”, mein Minimalcomputer, der alle Anforderungen, die ich an einen Minimalcomputer stelle, erfüllt.

Ich möchte mich hier bei allen bedanken, die mich während meiner Maturaarbeit begleitet haben. Ein besonderer Dank gilt meiner betreuenden Lehrkraft, Herr J. Degen, der mir meinen Freiraum liess, aber dennoch immer Zeit für mich hatte, und meinem Vater, der mich während dem ganzen Projekt unterstützt hat.

Inhaltsverzeichnis

Zusammenfassung

Das Hauptziel meiner Arbeit bestand darin, den grundlegenden Aufbau und die Funktionsweise eines Computers von unten her zu verstehen. Um dieses Ziel zu erreichen, stellte ich mir die Aufgabe, einen minimalen, d.h. auf das Allerwesentlichste beschränkten, einfachen Computer zu entwerfen und mindestens einen Teil davon, das Rechenwerk, selbst zu bauen.

Da die Welt der Digitaltechnik für mich ganz neu war, musste ich mir als Erstes die nötigen Grundlagen erarbeiten. Von der theoretischen Seite her bedeutete dies, mich mit der auf 0 und 1 beschränkten Zahlenwelt vertraut zu machen und mich in die boolsche Algebra, Wahrheitstabellen und die verschieden Möglichkeiten der digitalen Zahlendarstellung einzulesen. Meine elektronischen Grundkenntnisse habe ich mir, basierend auf dem Prinzip eines Transistors, von unten her aufgebaut. Aus Transistoren entstehen logische Gates, aus diesen lassen sich komplexere Funktionen bauen, wie z.B. Multiplexer, Volladdierer und Flipflops. Mit diesem Basiswissen war ich dann in der Lage, die eigentlichen Baugruppen eines Computers, die Register, das Rechenwerk, der Speicher und die Schalteinheit, zu verstehen.

Nun musste ich mich entscheiden, wie MINICOMP, so der Name meines Minimalcomputers, konkret aufgebaut ist. Das Problem, das ich zuerst lösen musste, war, die Grundidee des Computers soweit zu reduzieren, dass er mit meinen beschränkten Möglichkeiten baubar wurde, dass seine Funktionalität jedoch nicht so einfach wurde, dass alle damit noch möglichen Programme jeden Reiz verlieren würden.

Das Resultat war der Entwurf eines Computers mit folgenden Eigenschaften:

- 8-Bit Datenwort
- 8-Bit Instruktionen aufgeteilt in je 4 Bits für Befehlscode und Operand
- 16 einfache Befehle für Laden und Speichern, Addition, Subtraktion, logische Funktionen AND, OR, XOR und NOT, bedingte und unbedingte Sprungbefehle und einem Haltbefehl
- 32 adressierbare Speicherworte, aufgeteilt in 24 auf Schalter basierten Lesespeicher für Programme und Konstanten und 8 Lese-Schreib-Speicher für Programmvariablen
- 8-Bit Rechenwerk, das mittels 6 Kontrollsignalen für viele verschiedene arithmetische und logische Operationen verwendet werden kann
- 8-Bit Datenbus, welcher den Speicher, das Rechenwerk und die folgenden Register miteinander verbindet: Akkumulator, Programmzähler, Instruktionsregister und Adressregister
- Schalteinheit, die jeden Befehl in eine Abfolge von Mikrobefehlen zerlegt und mittels 16 Kontrollsignalen den Computer steuert

Zum Bau von MINICOMP verwendete ich vorwiegend CMOS Chips. Diese und die übrigen Komponenten lötete ich auf selbst entworfene und selbst geätzte Kupferplatinen, je eine Platine pro Funktionseinheit. Ich montierte die einzelnen Computerteile auf eine solide Spanplatte und verband sie mit Flachbandkabeln untereinander. Beim Design von MINICOMP achtete ich besonders darauf, dass möglichst alle internen Vorgänge mittels Leuchtdioden und 7-Segment-Anzeigen für das Auge sichtbar gemacht wurden.

Um die Fähigkeiten von MINICOMP demonstrieren zu können, schrieb ich einige Programme. Eines davon berechnet hintereinander die Quadratzahlen zwischen 1 und 225. Ein anderes multipliziert zwei eingegebene Werte miteinander.

Durch dieses Projekt habe ich viel Theoretisches und Praktisches gelernt, und ich bin froh, dass ich, trotz gewissen Schwierigkeiten, mein Ziel erreicht habe.

1 Einleitung

Der tägliche Gebrauch des Computers ist für mich, wie für viele andere Leute, eine Selbstverständlichkeit. Schularbeiten, E-Mails austauschen, Informationen beschaffen über das World Wide Web, digitale Fotos anschauen und bearbeiten – bei all diesen Tätigkeiten bin ich auf den Computer angewiesen.

Ich kann zwar mit dem Computer umgehen, doch weiss ich auch, warum und wie ein Computer überhaupt funktionieren kann?

Diese Frage gab mir die Idee zum Thema meiner Maturaarbeit.

1.1 Fragestellung

Für meine Maturaarbeit, habe ich mir folgende Fragen gestellt:

- Wie einfach darf ein Computer sein, damit er noch als Computer gilt?
- Aus welchen Grundbausteinen besteht ein solcher Minimalcomputer?
- Wie funktionieren die darin verwendeten Bauteile?
- Wie sind sie zu verbinden, damit daraus ein Computer entsteht?
- Ist es mir möglich, einen solchen Minimalcomputer nicht nur auszudenken, sonder ihn effektiv auch zu bauen?

1.2 Definition eines Minimalcomputers

Da der erste Punkt der Fragestellung den Rest dieser Arbeit bestimmt, möchte ich meine Definition eines Minimalcomputers schon hier erläutern:

Das wichtigste Merkmal eines Computer ist, dass er **programmierbar** sein muss. Das bedeutet, dass der Computer Befehle erkennen und selbstständig abarbeiten kann. Die Befehlsfolge (das “Programm”) wird jedoch nicht zum Vornherein beim Bau des Computers festgelegt, sondern kann vom Benutzer, innerhalb wohldefinierter Regeln, frei gestaltet werden.

Aus dieser Definition lässt sich schon schliessen, dass die Hauptsache an einem Computer nicht der Bildschirm, die Tastatur und die Maus ist. Damit mein Minimalcomputer rechnen, Werte lesen und speichern kann sowie programmierbar ist, genügt es völlig, wenn er folgende Eigenschaften hat:

- 7-Segment-Anzeigen anstelle eines Bildschirms
- Schalter für die Eingabe von Programmen und Zahlen anstelle einer Tastatur
- 16 Befehle, um einfache arithmetische, logische und Sprung-Funktionen auszuführen, anstelle von hunderten von verschiedenen Befehlsvarianten
- 8-Bit Worte anstelle von 32- oder gar 64-Bit Worten
- 28 Bytes Speicher für Programme und Daten anstelle von hunderten von Megabytes Hauptspeicher und “zig” Gigabytes Festplattenspeicher

1.3 Methodik

Ich wollte den Computer wirklich von Grund auf verstehen und so eignete ich mir als Erstes ein Grundwissen über Transistoren an, wie sie gebaut sind und wie sie funktionieren. Danach machte ich mich mit den Gates und anderen Hilfsmitteln (Wahrheitstabelle, Karnaugh-Diagramm, etc.) bekannt. Sobald ich diese kannte, fing ich an, die Gates zu komplexeren Komponenten zusammenzufügen. Ich baute relativ früh ein RS Flipflop und schaffte mich immer weiter hinauf, bis ich schliesslich alle wichtigen Teile eines Computers gebaut hatte. Als Informationsmaterial dienten mir die Bücher, welche im Literaturverzeichnis aufgelistet sind, sowie die Hilfe meines Vaters, wenn ich alleine nicht mehr weiter kam. Später als es ans Programmieren von GALs und EPROMs ging, benutzte ich, zum Programm schreiben, das Computerprogramm ispLever [12] und um die Chips zu brennen, das Computerprogramm Galep32 [11].

1.3.1 Angewandte Techniken

Um meinen Minimalcomputer zu bauen, verwendete ich drei verschiedene Techniken: Das Testboard mit Steckkontakten, die Wirewrap Technik und selbst geätzte Kupferplatinen, auf welche ich dann die Komponenten lötete.

Meine ersten Kontakte mit den verschiedenen Gates machte ich einfachheitshalber auf den Testboards. Da man auf Testboards die Komponenten nur stecken kann und auch einfach wieder entfernen kann, testete ich auch die Durchführbarkeit einzelner Teile meines Computers auf Testboards. Abbildung 1 zeigt ein RS Flipflop auf einem Testboard.

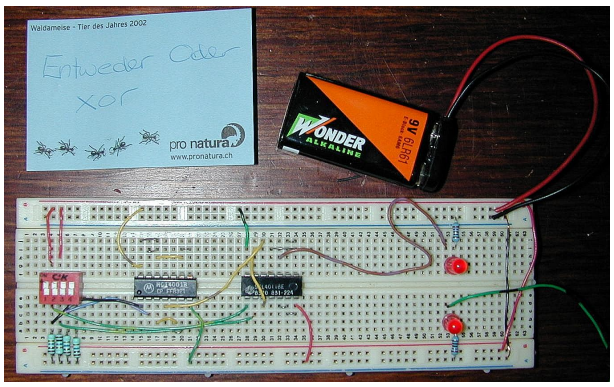


Abbildung 1: Einfaches RS-Flipflop auf Testboard

Mein erstes Alu-Bit fertigte ich mit Wirewrap Technik an. Bei der Wirewrap Technik wickelt man mit einem speziellen Werkzeug einen feinen Draht um einen Pfosten, das andere Ende des Drahtes wird mit einem anderen Pfosten verbunden, der Draht ist dann die Leitung zwischen diesen beiden Komponenten. Abbildung 2 zeigt den ersten Prototyp eines Bits der ALU von oben und unten.

Da die Wirewrap Technik sehr zeitaufwändig ist, vorallem wenn man mehrere gleiche Teile herstellen will, ätzte ich für den Bau des Minimalcomputers Kupferplatinen, die als Leiterbahnen und Träger für die elektronischen Komponenten dienen. Vor dem Ätzen muss man den Plan für die Platine kreieren. Einmal fertig legt man die Folie, auf welcher dieser Plan aufgedruckt ist, auf eine photobeschichtete Platine und belichtet sie für ca. drei Minuten mit UV-Strahlen. Danach gibt man die Platte in einen Entwickler (NaOH). Nach dem Entwickeln wäscht man die Platte mit Wasser ab. Die UV-Strahlen und der Entwickler bewirken, dass die Photoschicht nur entsprechend den Linien auf dem Plan fixiert bleibt und so das darunterliegende Kupfer vor der Ätzlösung schützt. Nun gibt man die Platine in die Ätzlösung (Natriumperoxodisulfat), bis dass alles überflüssige Kupfer abgeätzt ist. Nach diesem Vorgang spült man die Platine erneut mit Wasser. Nun ist auf den

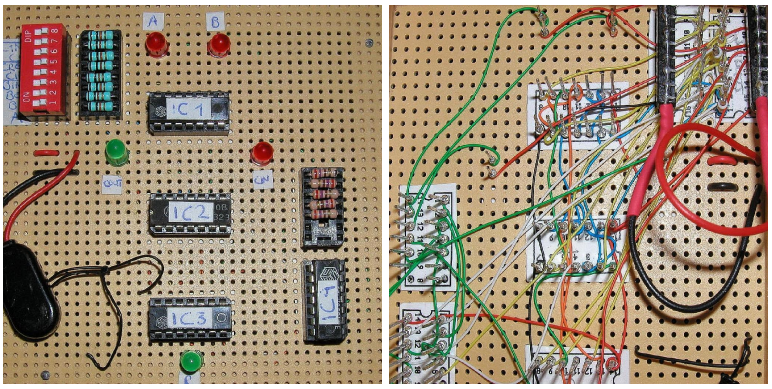


Abbildung 2: Prototyp Alu-Bit in Wirewrap-Technik (links von oben, rechts von unten)

Leiterbahnen noch überall ein Lack (Photoschicht), der an den späteren Lötstellen entfernt werden muss. Für das Entfernen belichtet man die Platte erneut mit UV-Strahlen durch eine Lötaugenmaske, danach gibt man sie noch einmal in den Entwickler. Nach dem erneuten Abspühlen mit Wasser ist der Ätzvorgang fertig. Nun werden die Löcher gebohrt, damit die Komponenten eingesetzt und angelötet werden können. Abbildung 3 zeigt die wichtigsten Arbeitsschritte der Platinenherstellung.

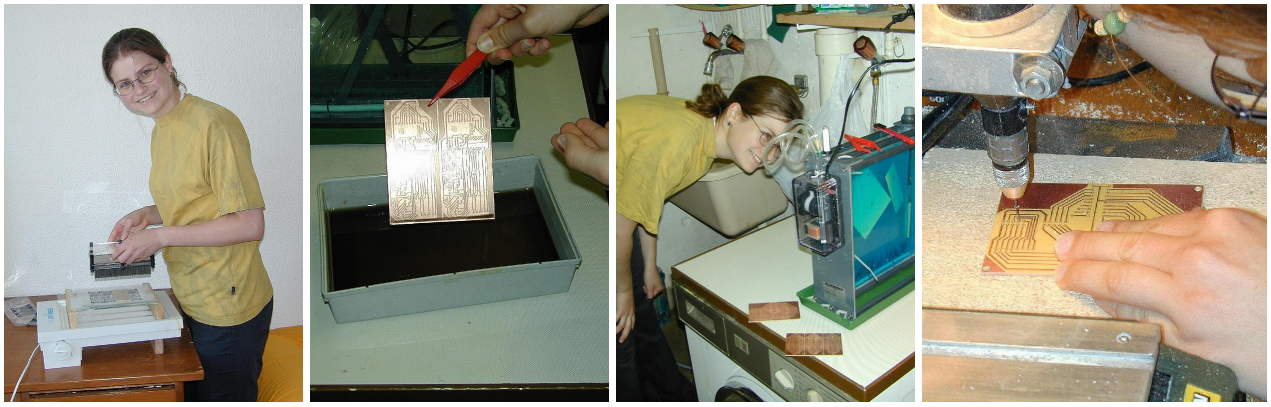


Abbildung 3: Herstellung einer Kupferplatine: Belichten, entwickeln, ätzen, bohren (v.l.n.r)

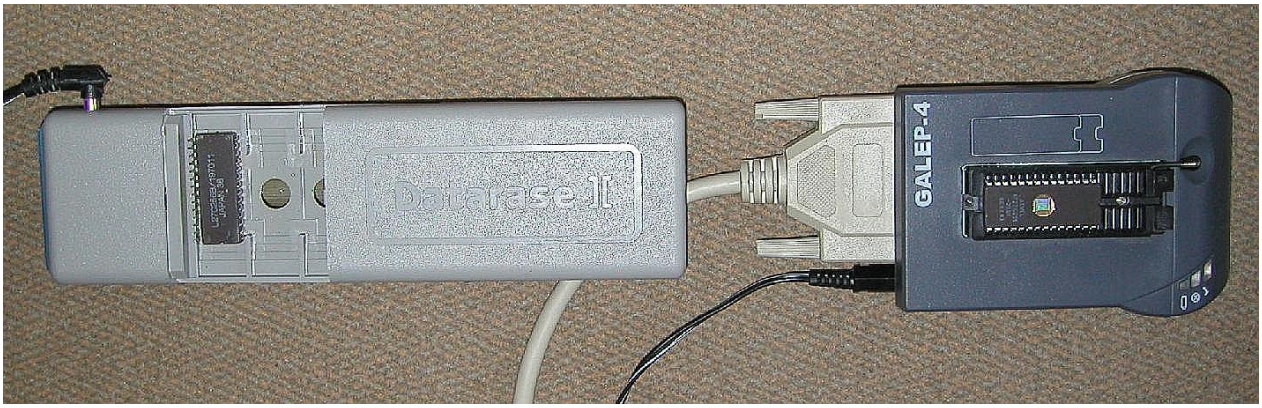


Abbildung 4: EPROM Löschgerät und Galep Programmiergerät

Nachdem ich mich mit den einfachen Bausteinen (Gates) vertraut gemacht hatte und mich an kompliziertere Schaltungen wagte, war es nicht mehr sinnvoll, diese ausschliesslich aus einfachen Gates zu bauen, da dies zuviel Zeit beansprucht hätte. Deshalb verwendete ich bald kompliziertere festverdrahtete Chips (z.B. Multiplexer-Demultiplexer) sowie programmierbare Bausteine, d.h.

EPROMs und GALs. Der Vorteil dieser Bauteile ist, dass sie elektrisch programmierbar sind und auch gelöscht und wieder beschrieben werden können. Das ist sehr praktisch, um Fehler zu korrigieren. EPROMs werden mit kurzwelligen UV-Strahlen gelöscht. Dazu besitzen sie ein kleines Glasfenster auf der Oberseite. Nach dem Programmieren ist es wichtig, dass dieses Fenster lichtdicht abgedeckt wird, da sonst das Umgebungslicht Teile des Inhalts unerwüschterweise löschen würde. Abbildung 4 zeigt ein EPROM-Löschgerät und das GALEP Programmiergerät, beide mit einem EPROM belegt.

1.3.2 Verwendete Programme

Auch für die Maturaarbeit zeigte sich wieder einmal, wie wichtig Computer sind. Ohne die folgende von mir verwendeten Computerprogramme, wäre diese Arbeit erschwert oder gar nicht möglich gewesen:

Programm:	*Verwendung:*	*Referenz:*
awk	Skriptsprache, welche ich zum Erstellen der EPROM-Dateien verwendete	[1]
galep32	Programm zum “Brennen” von GAL und EPROM Chips	[11]
ispLever	Programm, mit welchem ich die Logik der GAL-Bausteine definierte	[12]
LaTeX	Textsatzprogramm, mit welchem dieser Text formatiert wurde.	[6]
Linux	Freies Unix-artiges Betriebssystem, auf welchem ich diese Arbeit verfasste.	[14]
PCB	Platinen-Layout-Programm, mit welchem alle Platinen von MINICOMP entworfen wurden	[13]
Xfig	Vektororientiertes Zeichnungsprogramm, mit welchem ich alle Schaltpläne erstellte.	[15]
xv	Bildbearbeitungsprogramm, mit welchem ich die im Bericht erscheinenden Fotos bearbeitete.	[10]

2 Grundlagen

2.1 Theorie

2.1.1 Boolsche Algebra

Für die Operationen mit den Schaltfunktionen gilt eine Algebra, welche ähnlich der Algebra ist, die wir vom Mathematikunterricht her kennen. Diese Algebra nennt man Schaltalgebra oder boolsche Algebra.

Es gelten folgende Regeln und Voraussetzungen:

Wir verwenden das Binärsystem, das bedeutet, dass wir nur die Menge B von 0 (falsch) und 1 (wahr) haben, um damit zu rechnen.

Die boolsche Algebra kennt drei verschiedene Operationen:

AND	Logische “UND”-Verknüpfung zweier Binärwerte, Operator $\&$ (z.B. $a\&b$)
OR	Logische “ODER”-Verknüpfung zweier Binärwerte, Operator $\#$ (z.B. $a\#b$)
NOT	Logische Verneinung eines Binärwerts, Operator $\overline{\ }$ (z.B. $\overline{a}$)

Es gibt verschiedene Gesetze für $a, b, c \in B$

a) Kommutativgesetze

$$a\&b = b\&a$$

$$a\#b = b\#a$$

b) Assoziativgesetze

$$(a\&b)\&c = a\&(b\&c)$$

$$(a\#b)\#c = a\#(b\#c)$$

c) Absorptionsgesetze

$$a\&(a\#b) = a$$

$$a\#(a\&b) = a$$

d) Distributivgesetze

$$a\&(b\#c) = (a\&b)\#(a\&c)$$

$$a\#(b\&c) = (a\#b)\&(a\#c)$$

e) Neutrale Elemente

$$a\&1 = a$$

$$a\#0 = a$$

f) Komplementäres Element

$$a\&\overline{a} = 0$$

$$a\#\overline{a} = 1$$

$$\overline{\overline{a}} = a$$

g) Gesetz von Morgan

$$\overline{a}\&\overline{b} = \overline{a\#b}$$

In Anlehnung an die normale Algebra nennt man eine AND-Verknüpfung auch ein (boolsches) Produkt, und eine OR-Verknüpfung eine (boolsche) Summe. Wie in der normalen Algebra, gilt auch hier die Regel “Produkt vor Summe”, d.h. z.B. $a\&b\#c = (a\&b)\#c$.

2.1.2 Wahrheitstabelle

Da eine logische Funktion, auch boolsche oder binäre Funktion genannt, mit n Variablen immer einen Definitionsbereich von genau 2^n Wertekombinationen besitzt, ist jede logische Funktion eindeutig bestimmt durch das Aufzählen der 2^n dazugehörigen binären Funktionswerte.

Eine solche Auflistung aller möglichen Kombinationen der Eingangswerte und der entsprechenden Funktionswerten (Werte an den Ausgängen) nennt man **Wahrheitstabelle**.

Hier ein Beispiel einer Wahrheitstabelle einer Funktion mit drei Variablen $f(a, b, c)$:

Eingänge			Ausgang
a	b	c	$f(a, b, c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Oft kennt man von einer zu implementierenden logischen Funktion nur die Wahrheitstabelle. Um die Funktion effizient elektronisch realisieren zu können, stellt sich deshalb oft das Problem, wie man eine gegebene Wahrheitstabelle in einen boolschen Ausdruck umwandeln kann.

Eine sehr einfache Methode, eine Wahrheitstabelle in eine boolsche Funktion umzuwandeln, besteht darin, für jeden Funktionswert 1 das entsprechende boolsche Produkte zu bilden und diese mit der OR-Verknüpfung zu summieren. Für die obige Funktion ergibt dies:

$$f(a, b, c) = \overline{a}\&b\&c \# a\&\overline{b}\&c \# a\&b\&\overline{c} \# a\&b\&c$$

Als Alternative kann man die Funktion auch als Produkt der Summen aller Funktionswerte 0 darstellen. Für das obige Beispiel erhält man so:

$$f(a, b, c) = (a\#b\#c)\ \&\ (a\#b\#\overline{c})\ \&\ (a\#\overline{b}\#c)\ \&\ (\overline{a}\#b\#c)$$

2.1.3 Karnaugh-Diagramm

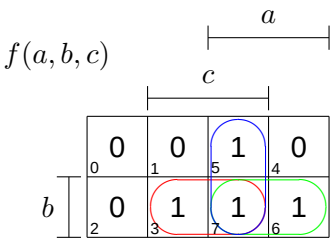
Die im vorherigen Abschnitt erklärten einfachen Methoden zur Umwandlung von Wahrheitstabellen in boolsche Ausdrucke sind zwar richtig, ergeben aber meistens Formeln, welche viel komplizierter als nötig sind. Zwar können immer die boolschen Gesetze verwendet werden, um diese Ausdrucke zu vereinfachen, dies ist jedoch recht knifflig.

Einfacher geht es mit der grafischen Methode des Karnaugh-Diagramms, welches auch Karnaugh-Veitch-Diagramm genannt wird. In dieser Methode werden die Funktionswerte in einer zweidimensionalen Tabelle so dargestellt, dass die Hälfte der unabhängigen Variablen horizontal, die andere Hälfte vertikal verlaufen. Die Werte der unabhängigen Variablen sind dabei so angeordnet, dass sich die Eingabewerte zweier benachbarten Funktionswerte immer nur durch ein Bit unterscheiden. Die horizontalen, resp. vertikalen Bereiche für welche eine Variable den Wert 1 hat, sind entlang der entsprechenden Kante des Diagramms angegeben.

In einem solchen Diagramm entsprechen alle rechteckigen Bereiche mit 2^n Zellen, welche alle den Funktionswert 1 enthalten, einem boolschen Produkt. Dabei ist zu beachten, dass im Diagramm diese Bereiche auch den einen Diagrammrand überschreiten können, d.h. nach der letzten Spalte, resp. Zeile, folgt wieder die erste. Je grösser dabei ein Bereich ist, desto einfacher das entsprechende boolsche Produkt.

Das Ziel der Vereinfachung ist es, alle Funktionswerte 1 mit möglichst wenigen, dafür möglichst grossen solchen 2^n grossen Rechteckbereichen abzudecken. Dabei ist es erlaubt, den gleichen Wert mehr als einmal abzudecken.

Für das in der Sektion 2.1.2 gezeigte Beispiel, sieht das Karnaugh-Diagramm wie folgt aus:

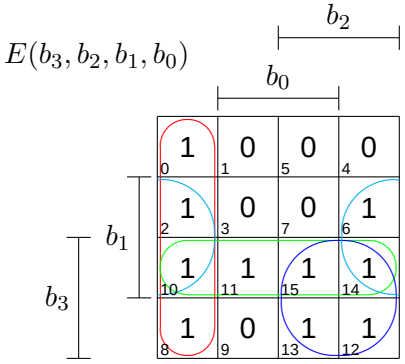


Aus den drei eingetragenen Bereichen, ergibt sich folgender vereinfachter Ausdruck:

$$f(a,b,c) = b\&c \# a\&b \# a\&c$$

Eine etwas kompliziertere Funktion, welche für den Bau der Hexadezimalanzeige (siehe Sektion 3.2.5) Verwendung findet, möchte ich als weiteres Beispiel anfügen.

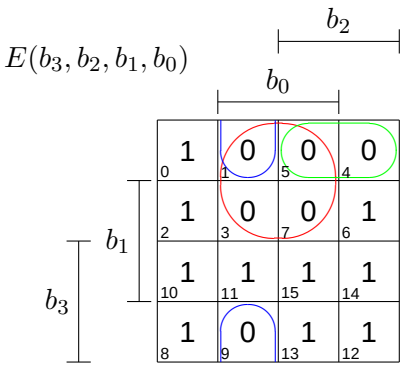
Das folgende Karnaugh-Diagramm basiert auf der Wahrheitstabelle für das Segment e (d.h. links unten) einer hexadezimalen 7-Segment-Anzeige:



Aus dem obenstehenden Karnaugh-Diagramm ergibt sich folgende vereinfachte Formel:

$$E(b_3,b_2,b_1,b_0) = \overline{b_0}\&\overline{b_2}\#b_1\&b_3\#b_2\&b_3\#\overline{b_0}\&b_1$$

Wenn in einem Karnaugh-Diagramm die 0-Werte einfacher abgedeckt werden können als die 1-Werte, was im obigen Diagramm der Fall ist, ist es einfacher, die gleiche Methode zur Bestimmung der Komplementärfunktion anzuwenden. Das ergibt folgende Abdeckung der 0-Werte:



Aus diesem Diagramm leiten wir folgende vereinfachte Formel ab:

$$E(b_3, b_2, b_1, b_0) = \overline{b_0 \& \overline{b_3} \# \overline{b_1} \& b_2 \& \overline{b_3} \# b_0 \& \overline{b_1} \& \overline{b_2}} = (\overline{b_0} \# b_3) \& (b_1 \# \overline{b_2} \# b_3) \& (\overline{b_0} \# b_1 \# b_2)$$

Durch Anwendung der boolschen Gesetze kann man zeigen, dass beide abgeleiteten Formeln für das Segment e identisch sind.

2.1.4 Zahlendarstellung

Wenn man mit Binärwerten andere Zahlen als 0 und 1 darstellen will, benötigt man dazu mehrere Binärwerte. Mit n Binärwerten $b_0, \ldots, b_{n-1}$ lassen sich 2^n verschiedene Zahlen darstellen. Wenn man sich auf den natürliche Zahlenraum $\mathbb{N}_0$ beschränkt, so entspricht je ein Binärwert einer Zweierpotenz, d.h. die Binärwerte $b_0, \ldots, b_{n-1}$ stehen für folgende Zahl z

$$z = \sum_{i=0}^{n-1} 2^i b_i$$

Hier ein Beispiel (beachte, das b_0 ganz rechts und b_{n-1} ganz links steht):

$$00110110_2 = 1 \cdot 0 + 2 \cdot 1 + 4 \cdot 1 + 8 \cdot 0 + 16 \cdot 1 + 32 \cdot 1 + 64 \cdot 0 + 128 \cdot 0 = 54$$

Wenn man aber neben den positiven auch negative Zahlen darstellen will, gibt es verschiedene Darstellungsarten:

Explizites Vorzeichen-Bit

Bei dieser Darstellungsart wird das Vorzeichen in einem dafür reservierten Bit (üblicherweise das höchste, hier a_7) gespeichert.

Für 8-Bit Zahlen, gibt es 127 positive und 127 negative Zahlen, dazu gibt es noch +0 und -0, das heisst, es gibt 255 verschiedene Zahlenwerte.

$$A = [v, a_6, a_5, \ldots, a_0] \rightarrow -A = [\overline{v}, a_6, a_5, \ldots, a_0]$$

Beispiele:	+0	=	00000000_2	-0	=	10000000_2
	+5	=	00000101_2	-5	=	10000101_2
	+126	=	01111110_2	-126	=	11111110_2

Achtung der Wert Null kommt zweimal vor!

Einer-Komplement

Beim Einer-Komplement wird eine Zahl durch Komplementierung aller Bits negiert.

$$A = [a_7, a_6, a_5, \ldots, a_0] \rightarrow -A = [\overline{a_7}, \overline{a_6}, \overline{a_5}, \ldots, \overline{a_0}]$$

Das Vorzeichen wird durch das oberste Bit (hier a_7) bestimmt.

Beispiele:						
	+0	=	00000000_2	-0	=	11111111_2
	+5	=	00000101_2	-5	=	11111010_2
	+126	=	01111110_2	-126	=	10000001_2

Achtung, auch hier kommt der Wert Null zweimal vor!

Zweier-Komplement

Die Idee des Zweierkomplements ist, dass der Wert Null nur auf eine Art dargestellt wird. Dies wird erreicht, in dem die negativen Zahlen gegenüber dem Einer-Komplement um den Wert 1 verschoben werden, d.h. dass der Binärwert 1111111_2 nicht -0 sondern -1 entspricht.

$$A = [a7, a6, a5, \ldots, a0] \rightarrow -A = [\overline{a7}, \overline{a6}, \overline{a5}, \ldots, \overline{a0}] + 1$$

Es bestehen die folgenden Zusammenhänge zwischen $\overline{A}$ und $-A$:

$$-A = \overline{A} + 1$$

$$\overline{A} = -A - 1$$

Einfachheitshalber zeige ich hier die Zahlendarstellung im Zweier-Komplement nur für 4-Bit Zahlen:

0000_2	0	1000_2	-8
0001_2	1	1001_2	-7
0010_2	2	1010_2	-6
0011_2	3	1011_2	-5
0100_2	4	1100_2	-4
0101_2	5	1101_2	-3
0110_2	6	1110_2	-2
0111_2	7	1111_2	-1

Für 8-Bit Zahlen lassen sich so alle Werte zwischen -128 und +127 darstellen.

Beispiele:						
	+0	=	00000000_2	-0	=	00000000_2
	+1	=	00000001_2	-1	=	11111111_2
	+5	=	00000101_2	-5	=	11111011_2
	+126	=	01111110_2	-126	=	10000010_2

Das Zweier-Komplement ist die gebräuchlichste Darstellungsart von negativen Zahlen und wird heute in fast allen Computern verwendet, so auch in MINICOMP.

2.2 Kombinatorische Digitalelektronik

Während man in der analogen Elektronik (Verstärker, Radio, Fernsehen) mit komplexen Spannungsverläufen zu tun hat, sind in der Digitalelektronik die Ein- und Ausgangsspannungen der Schaltungen im Wesentlichen auf zwei Zustände beschränkt. Diese beiden Zustände entsprechen den Spannungsbereichen, in welchem die Transistoren entweder ganz ein- oder ganz ausgeschaltet sind. Der Spannungsbereich nahe der Maximalspannung V_{dd} entspricht dem logischen Wert 1, derjenige nahe der Nullspannung V_{ss} dem logischen Wert 0.

Bei meinem Minimalcomputer ist die Maximalspannung V_{dd} etwa 5 Volt.

Die kombinatorische Digitalelektronik beschäftigt sich mit Schaltfunktionen, bei welchen die Ausgangswerte nur durch die momentanen Eingangswerte bestimmt sind. Das heisst, dass keine Werte gespeichert werden und Änderungen in den Eingangssignalen sich sofort auf die Ausgangssignale auswirken.

2.2.1 Transistoren

Ein Transistor ist der Grundbaustein der Digitalelektronik, welcher das Ein- und Ausschalten eines Stroms bzw. einer Spannung durch ein Kontrollsignal ermöglicht. Deshalb können aus Transistoren die verschiedenen logischen Gates gebaut werden, welche ich in Sektion 2.2.2 behandeln werde.

Ein Transistor besteht aus Silizium oder aus sonstigen Elementen aus der 4. Hauptgruppe des Periodensystems der Elemente. Das Silizium ist ein sogenannter Halbleiter, es leitet somit besser als isolierende Stoffe, wie Porzellan, aber nicht so gut wie ein Metall, ein Leiter. Wenn man Silizium nun aber mit Atomen der 3. oder der 5. Hauptgruppe dotiert, ist es besser leitend, da es “freie” Ladungen hat. Wenn Atome der 5. Hauptgruppe eingebaut werden (z.B. Arsen), so ist die positive Ladung ortsfest und die negative Ladung beweglich, es ist also n-leitend. Bei eingebauten Atomen der 3. Hauptgruppe (z.B. Gallium, Indium oder Aluminium) ist die positive Ladung beweglich und die negative ortsfest, es ist also p-leitend.

Es gibt verschiedene Möglichkeiten, Transistoren aus n- und p-Leitern zu bauen. Ich beschränke mich hier auf die Beschreibung der Isolierschicht-Feldeffekttransistoren (MOSFET = **M**etal **O**xid **S**emiconductor **F**ield **E**ffect **T**ransistor), da die von mir verwendeten CMOS-Chips auf dieser Technologie basieren.

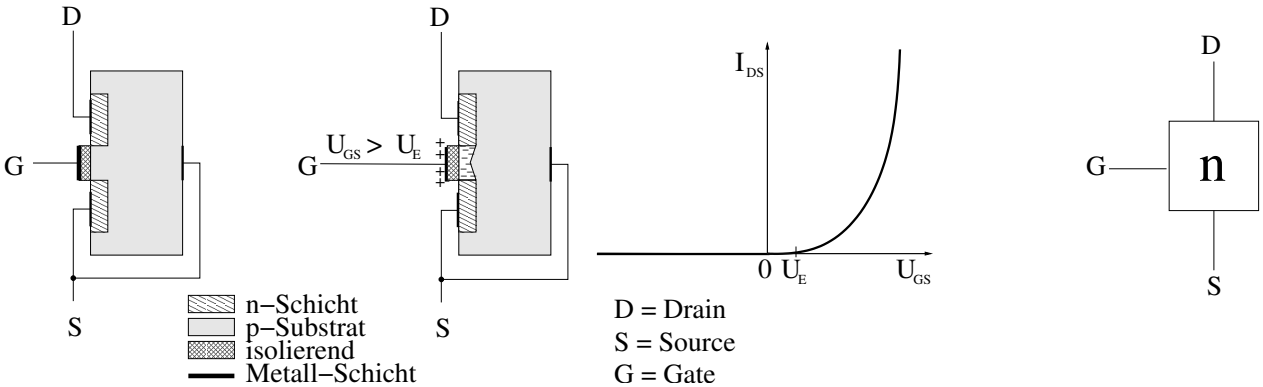


Abbildung 5: nMOS Transistor: Aufbau, n-Kanal-Leitung, Schaltkurve, Symbol

Abbildung 5 zeigt links den Aufbau eines n-Kanal MOSFET. Dieser besteht aus einem p-dotieren Substrat, in welches zwei n-dotierte Zonen (Drain **D** und Source **S**) eingelassen sind. Zwischen Source und Drain befindet sich ausserhalb und isoliert vom Substrat das sogenannte Gate **G** (nicht zu verwechseln mit den *logischen* Gates, welche ich im Rest der Arbeit nur als “Gate” bezeichne).

Wird nun am Gate eine genügend grosse positive Spannung angelegt, so werden die beweglichen Elektronen der **D** und **S** Zonen in Richtung **G** angezogen und bilden so einen leitenden Kanal zwischen Drain und Source. Der Strom I_{DS} kann somit durch Verändern der Spannung am Gate U_{GS} ein- und ausgeschaltet werden.

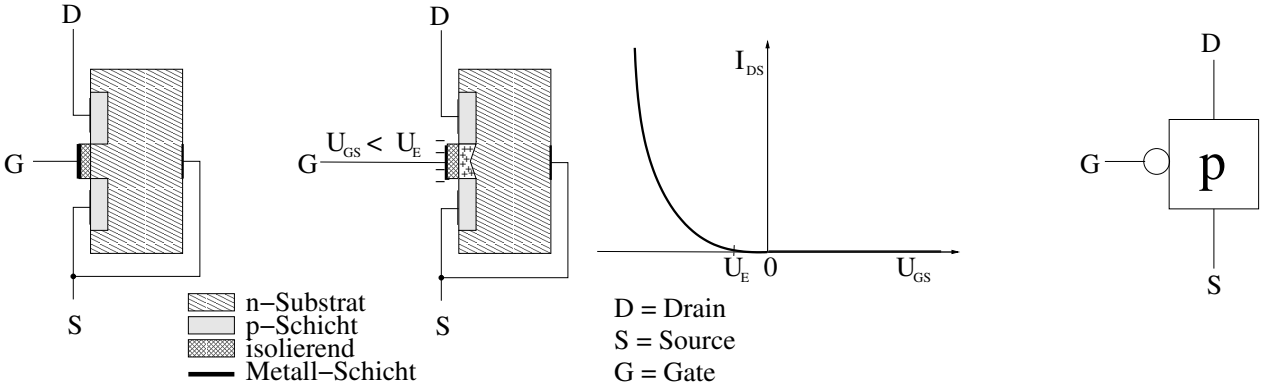


Abbildung 6: pMOS Transistor: Aufbau, p-Kanal-Leitung, Schaltkurve, Symbol

Komplementär zum n-Kanal MOSFET ist der p-Kanal MOSFET gebaut, wie in Abbildung 6 ersichtlich ist. Hier muss die Spannung U_{GS} genügend negativ sein, um einen leitenden p-Kanal zu erzeugen, und so den Strom I_{DS} einzuschalten.

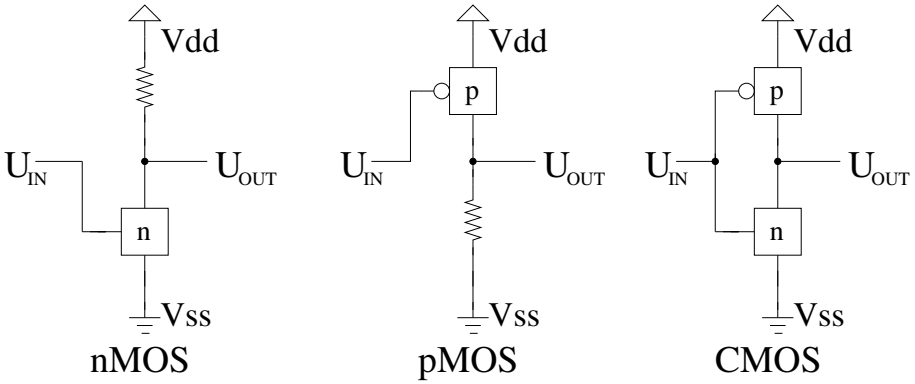


Abbildung 7: Drei Inverterschaltungen basierend auf nMOS und/oder pMOS Transistoren

Zur Herstellung von logischen Gates kann man entweder nur nMOS-Transistoren, nur pMOS-Transistoren oder eine Kombination von beiden verwenden. Auf dem letzteren basiert die CMOS Technologie (**C**omplementary **MOS**). In Abbildung 7 werden die drei möglichen Bauweisen für einen Inverter gezeigt, bei welchem die Ausgangsspannung U_{OUT} sich umgekehrt zur Eingangsspannung U_{IN} verhält. Damit die Transistoren vollständig eingeschaltet werden können, d.h. U_{GS} genügend positiv resp. negativ werden kann, befindet sich die Source der nMOS-Transistoren immer an V_{ss} und die Drain der pMOS-Transistoren immer an V_{dd}. Bei der nMOS und pMOS Varianten des Inverters fliesst jeweils ein Ruhestrom durch Transistor und Widerstand solange der Transistor eingeschaltet ist. Bei der CMOS Variante fliesst hingegen kein Ruhestrom, da immer einer der beiden Transistoren ausgeschaltet ist. Innere Ströme fliessen nur während den Schaltvorgängen. Dies ist der Grund, weshalb auf CMOS basierende elektronische Schaltungen sehr stromsparend sind. Die Stromaufnahme steigt mit zunehmender Schaltfrequenz.

Da über das Gate **G** eines MOSFET kein Strom fliessen kann, sind bei der Arbeit mit CMOS Schaltungen folgende Punkte wichtig:

- Alle Eingänge müssen einen klar definierten Spannungspegel haben. Unbenutzte Eingänge

müssen mit V_{ss} oder V_{dd} verbunden werden.

- CMOS-Bausteine sind sehr empfindlich bezüglich statischer Elektrizität.
- Im Gegensatz zu den anderen Technologien kann man einen CMOS-Ausgang mit beliebig vielen CMOS-Eingängen verbinden. Deshalb sagt man, dass CMOS-Bausteine ein hohes Fan-Out haben.

2.2.2 Logische Gates

Aus den Transistoren baut man logische Gates, welche als Grundbausteine meines Minimalcomputers dienen. Ein logisches Gate ist eine einfache Schaltung, die aus einem oder mehreren digitalen Eingangssignalen ein digitales Ausgangssignal generiert.

Es gibt verschiedene Gates, auf deutsch Gatter, wie AND, OR, NOT, NAND, NOR und XOR. Die Tabelle 1 enthält für jedes dieser Gates folgende Informationen:

- die Wahrheitstabelle, die für alle möglichen Eingangswerte den dazugehörigen Ausgangswert angibt;
- das amerikanische und das deutsche Schaltzeichen;
- den CMOS-Schaltplan des Gates;
- die Nummer und die Pinbelegung des gebräuchlichsten CMOS-Chips.

In meinem Text werde ich die englischen Namen sowie die amerikanischen Zeichen verwenden.

Stellvertretend für alle Gates der Tabelle 1, erkläre ich an dieser Stelle das NAND-Gate.

Das NAND-Gate besteht aus zwei parallel geschalteten pMOS- und zwei in Serie geschalteten nMOS-Transistoren. Solange einer der beiden Eingänge 0 ist, ist der Ausgang über einen der pMOS-Transistoren mit V_{dd} verbunden, d.h. logisch 1. Nur wenn beide Eingänge 1 sind, besteht keine Verbindung mit V_{dd}, sondern mit V_{ss}, da nun beide nMOS-Transistoren eingeschaltet sind. Durch Zufügen von weiteren parallelen pMOS- und seriellen nMOS-Transistoren können NAND-Gates mit mehr als zwei Eingängen gebaut werden.

Der gebräuchlichste CMOS NAND-Chip hat die Nummer 4011. Er enthält vier NAND-Gates mit je zwei Eingängen.

NAND-Gates sind von besonderer praktischer Wichtigkeit, da sie einfach zu realisieren sind und man jede beliebige logische Funktion mit ihnen herstellen kann. Deshalb sagt man, dass NAND-Gates alleine schon ein komplettes logisches Set bilden. Ausser für das NAND-Gate gilt dies nur noch für das NOR-Gate.

2.2.3 Multiplexer / Demultiplexer

Ein Multiplexer ist ein auswählendes Schaltnetz. Mit n Kontrollsignalen bestimmt man, welcher der 2^n Eingänge auf den einen Ausgang durchgeschaltet wird. Die Wahrheitstabelle zeigt die Funktion eines 4 zu 1 (4:1) Multiplexers:

Beim Demultiplexer ist der Datenfluss umgekehrt als beim Multiplexer. Hier wird mit n Kontrollsignalen ein Eingang auf einen von 2^n Ausgängen geschaltet, d.h. der Demultiplexer ist ein verteilendes Schaltnetz.

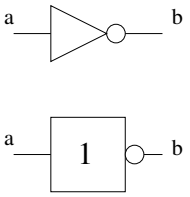
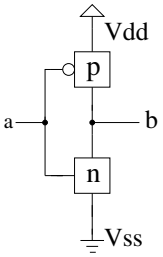
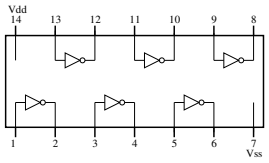
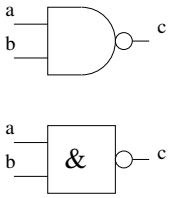
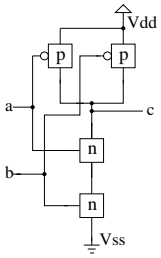
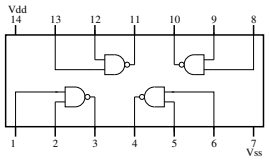
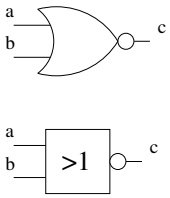
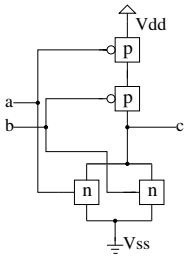
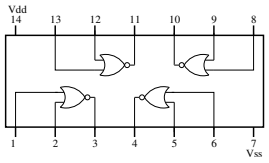
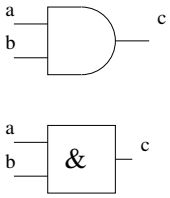
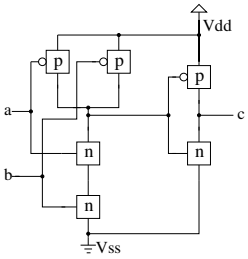
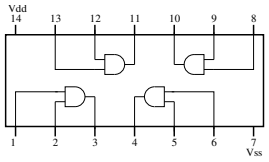
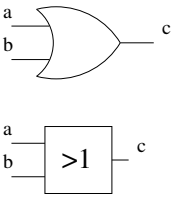
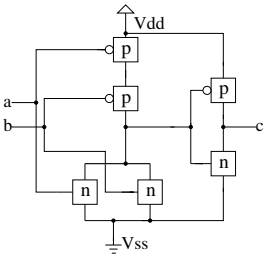
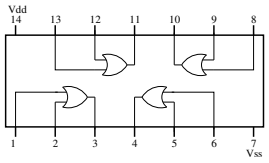
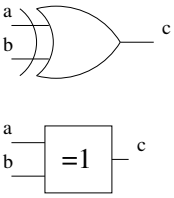
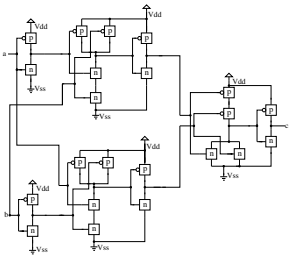
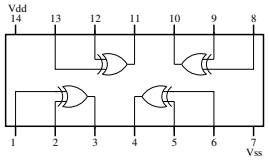
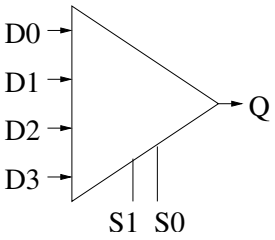
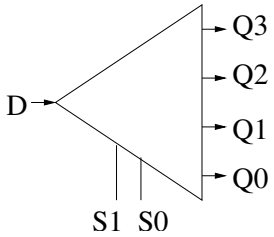
NAME Formel Wahrheitstabelle	Schaltzeichen USA DIN	Schaltplan	CMOS Chip
NOT $b = \overline{a}$ a | b 0 | 1 1 | 0			 CD4069
NAND $c = \overline{a\&b}$ b \ a | 0 1 0 | 1 1 1 | 1 0			 CD4011
NOR $c = \overline{a\#b}$ b \ a | 0 1 0 | 1 0 1 | 0 0			 CD4001
AND $c = a\&b$ b \ a | 0 1 0 | 0 0 1 | 0 1			 CD4081
OR $c = a\#b$ b \ a | 0 1 0 | 0 1 1 | 1 1			 CD4071
XOR $c = (a\&\overline{b})\#(\overline{a}\&b)$ b \ a | 0 1 0 | 0 1 1 | 1 0			 CD4070

Tabelle 1: Zusammenstellung der wichtigsten logischen Gates



Adresse		Ausgang
S_1	S_0	Q
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Abbildung 8: 4:1 Multiplexer: Symbol und Wahrheitstabelle



Adresse		Ausgänge			
S_1	S_0	Q_0	Q_1	Q_2	Q_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Abbildung 9: 1:4 Demultiplexer: Symbol und Wahrheitstabelle

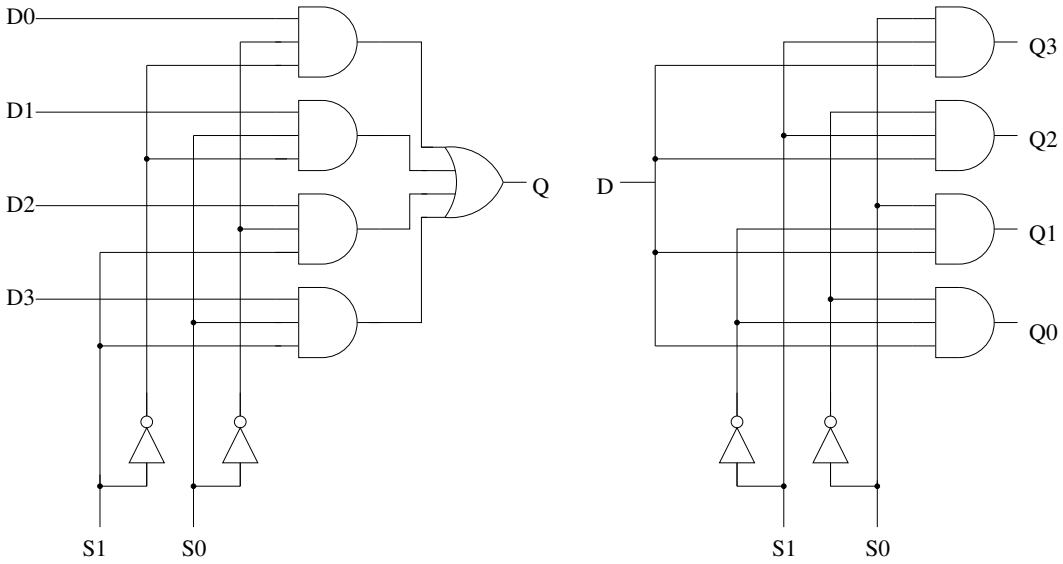


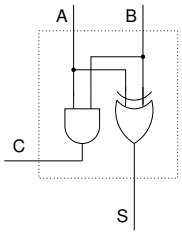
Abbildung 10: Schaltplan 4:1 Multiplexer und 1:4 Demultiplexer

Es gibt auch kombinierte Multiplexer/Demultiplexer. Hier wird eine richtungsunabhängige, meist analoge Verbindung zwischen dem einzelnen Ein-/Ausgang und dem durch die Adresssignale ausgewählten Aus-/Eingang hergestellt. Ein solcher kombinierter analoger 1:8 Multiplexer/Demultiplexer ist z.B. im CMOS Chip 4051 implementiert, welcher in meinem Lesespeicher Verwendung findet.

Bei meinem Minimalcomputer verwendete ich an verschiedenen Orten Multiplexer oder Demultiplexer. Ein Anwendungsort von einem Demultiplexer ist der Speicher, wo ich den Wert des Datenbusses an einen von vielen Speicherorten bringen will. Einen Multiplexer verwendete ich z.B. im Rechenwerk (ALU). Dank ihm kann man entweder die Summe oder das Behalte als Resultat haben.

2.2.4 Halbaddierer

Die einfachste arithmetische Operation, die man mit einer Verknüpfung logischer Gates berechnen kann, ist die Addition von zwei Binärwerten A und B. Da die Summe $A + B$ den Wert 0, 1 oder 2 annehmen kann, hat die Schaltung zwei binäre Ausgänge C und S. Diese Schaltung nennt man Halbaddierer und ihr Bauplan sowie ihre Wahrheitstabelle ist in Abbildung 11 dargestellt.



Eingänge		Ausgänge	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

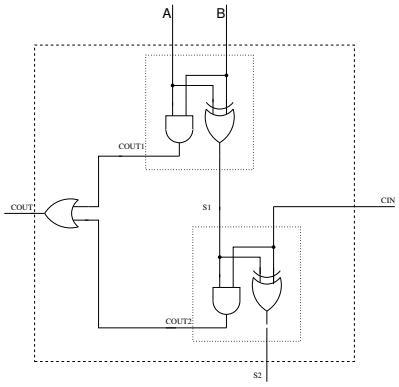
Abbildung 11: Halbaddierer: Addition von zwei Binärwerten

Die Wahrheitstabelle zeigt, dass der Ausgang C (**C**arry = Behalte) einem AND und der Ausgang S (**S**umme) einem XOR entspricht.

2.2.5 Volladdierer

Um Zahlen, die aus mehreren Bits bestehen, addieren zu können, genügt ein Halbaddierer nicht, auch kann man nicht mehrere davon parallel schalten, da der Halbaddierer kein Behalte-Eingang hat.

Dieser Mangel wird im Volladdierer behoben. Hier werden statt zwei, drei Binärwerte addiert, die ersten zwei, A und B, sind die entsprechenden Bits der zu addierenden Zahlen, während der dritte Wert das Behalte der niedrigstelligeren Bits einbringt. Der Volladdierer hat also ein Behalte, das hereinkommt (C_{in}), und eins, das herausgeht (C_{out}).



Eingänge			Ausgänge	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Abbildung 12: Volladdierer: Addition von drei Binärwerten

Wie in Abbildung 12 ersichtlich ist, besteht ein Volladdierer aus zwei Halbaddierern (deshalb deren Name) sowie einem OR, da nie beide Halbaddierer zur gleichen Zeit 1 als Behalte generieren können.

2.2.6 7-Segment-Anzeige

Wenn man hexadezimale oder dezimale Zahlen anzeigen will, geschieht dies meistens mit 7-Segment-Anzeigen. Diese bestehen aus 7 Leuchtdioden, die so angeordnet sind, wie es Abbildung 13 zeigt. Man benennt die Leuchtdioden mit den Namen a bis g. Segment a ist die oberste Leuchtdiode (der oberste Strich), danach geht es im Uhrzeigersinn weiter, g ist die Leuchtdiode, die in der Mitte ist.

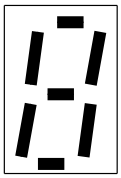


Abbildung 13: Schaltsymbol für eine 7-Segment-Anzeige

Es gibt zwei Typen von 7-Segment-Anzeigen, solche mit gemeinsamer Kathode und solche mit gemeinsamer Anode. Erstere muss man direkt mit dem darzustellenden Signal ansteuern, während für letztere der Komplementärwert benötigt wird. In jedem Fall muss der Strom, wie für alle Leuchtdioden, mit einem Widerstand auf maximal 20mA begrenzt werden.

2.2.7 Schmitt-Trigger

Wenn man aus einem analogen Signal ein digitales erzeugen will, verwendet man einen Schmitt-Trigger. Der Schmitt-Trigger benötigt eine gewisse Voltzahl (V_{on}), bis er auf 1 schaltet. Wenn dieser obere Schwellenwert überschritten wird, schaltet er mit einem klaren Signal ein (senkrechte Kurve). Umgekehrt, muss der untere Schwellenwert (V_{off}) unterschritten werden, damit das Signal auf 0 zurückkehrt. Das Band zwischen unterem und oberem Schwellenwert dient als Puffer, der auch leicht flackernde Signale stabilisiert. Die entsprechende Funktionskurve ist in Abbildung 14 gezeigt.

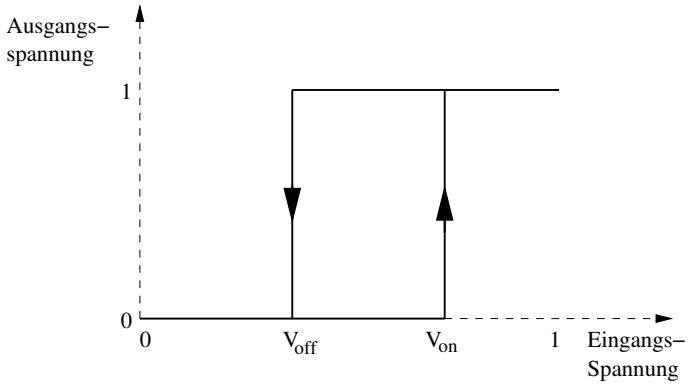


Abbildung 14: Funktionskurve eines Schmitt-Triggers (Hysterese-Kurve)

Abbildung 15: Schaltsymbole für Schmitt-Trigger NOT und NAND

Die Schmitt-Trigger Gates werden wie in Abbildung 15 dargestellt. Neben den dargestellten Varianten gibt es noch andere Schmitt-Trigger Gates, wie z.B. das NOR. Sie werden immer mit dem selben Zeichen in der Mitte dargestellt.

2.2.8 Transistor-Gate

Ein Transistor-Gate ist trotz des Namens weder ein logisches Gate noch das Gate eines MOSFET, sondern das Transistor-Gate ist ein elektronischer Schalter. Dieser erlaubt, mit einem Kontrollsignal K das Eingangssignal E mit dem Ausgangssignal A elektrisch zu verbinden ($K = 1$) oder zu trennen ($K = 0$). Wenn das Ausgangssignal vom Eingangssignal getrennt ist, so nennt man den Ausgang auch “hochohmig”.

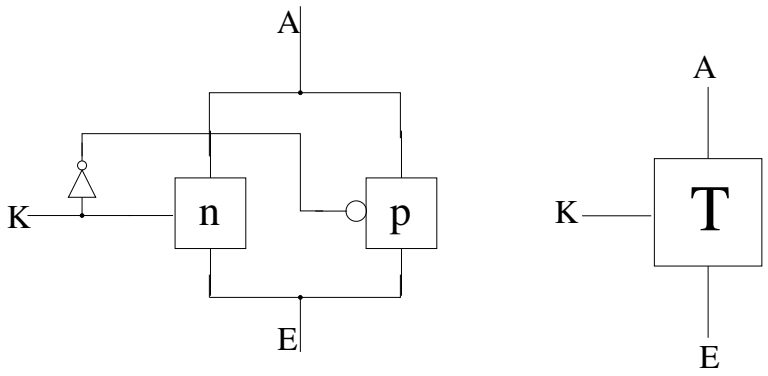


Abbildung 16: Schaltplan und Symbol des Transistor-Gates

Das Transistor-Gate besteht aus einem nMOS- und einem pMOS-Transistor, die parallel geschaltet sind und beide bei $K = 1$ leiten und bei $K = 0$ nicht leiten. Um dies zu realisieren, muss das Kontrollsignal K in invertierter Form an den G-Eingang des pMOS-Transistors gelangen. Abbildung 16 zeigt den Schaltplan und das Schaltsymbol eines Transistor-Gates.

2.3 Sequenzielle Digitalelektronik

Im Gegensatz zur kombinatorischen Digitalelektronik, beschäftigt sich die sequenzielle Digitalelektronik mit Schaltungen, deren Ausgangswerte nicht nur durch die momentanen Eingangswerte bestimmt sind, sondern auch von den vorherigen Zuständen abhängen. Dies bedeutet, dass Werte gespeichert und später wieder verwendet werden können.

2.3.1 Flipflop

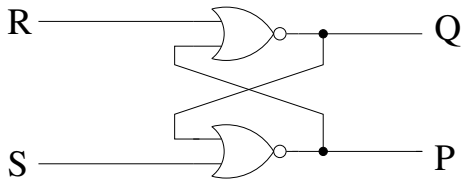
Es gibt verschiedene Arten von Flipflops (FF), welche ich in den nächsten Kapiteln kurz vorstellen werde. Ein Flipflop hat immer zwei stabile Zustände, d.h. man kann eine einstellige Binärzahl speichern. Das einfachste Flipflop besteht aus zwei NOT, welche wie in Abbildung 17 aneinander gehängt sind. Dieses einfache Flipflop ist aber in der Praxis nicht nützlich, da man seinen angenommenen Zustand nicht beeinflussen kann.

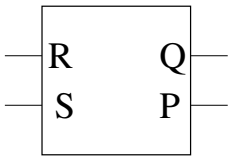
Abbildung 17: Grundschaltung eines Flipflops aus zwei NOT-Gates

RS-Flipflop

Das RS-Flipflop (RS-FF) ist das einfachste Flipflop nach dem Flipflop aus zwei NOT. Das RS-FF wird aus zwei NOR gebaut, dabei müssen sie so, wie in Abbildung 18 gezeigt, aneinandergehängt werden. Wobei der Name S des Eingangs “setzen” (**s**et) bedeutet und der Name R des anderen Eingangs “löschen” (**r**eset) meint.

Das RS-FF speichert die Zahl, die vorher im FF war, wenn beide Eingänge 0 sind. Wenn beide Eingäge hingegen 1 sind, ist der Zustand des Ausgangs Q nicht klar, d.h. man kennt ihn nicht im Voraus. Deshalb ist dieser Zustand verboten. Wenn jedoch nur der Eingang S auf 1 steht, so wird 1 gesetzt, $Q = 1$. Wenn nur der Eingang R auf 1 steht, so wird der Ausgang ein 0 sein, $Q = 0$.



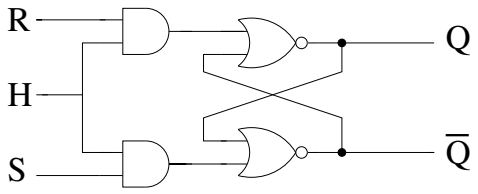


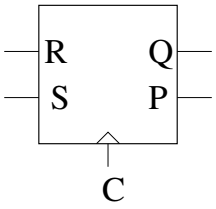
Eingänge		Ausgänge	
R	S	Q	P
0	0	speichern	
0	1	1	0
1	0	0	1
1	1	verboten	

Abbildung 18: Einfaches RS-Flipflop aus zwei NOR-Gates

RS Flipflop mit Taktsignal

Bei dieser Variante des RS-FF werden die Eingänge durch ein Taktsignal C (**C**lock) kontrolliert. Das heisst, der Zustand kann sich nur ändern, während C auf 1 steht. Dazu lässt man beide Eingänge zuerst durch ein AND, wobei der zweite Eingang des AND von dem Clock besetzt ist, vgl. Abbildung 19. Dieses FF macht dasselbe wie das einfache RS-FF, aber nur wenn der $C = 1$ ist, sonst ist es im Speicherzustand und bewahrt den vorherigen Zustand.



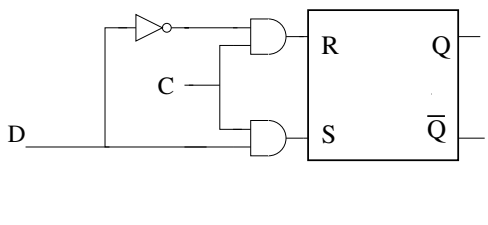


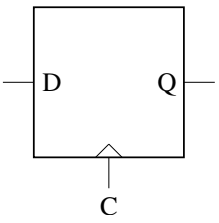
Eingänge		Ausgänge	
R	S	Q	P
0	0	speichern	
0	1	1	0
1	0	0	1
1	1	verboten	

Abbildung 19: RS-Flipflop mit Taktsignal

D Flipflop

Der verbotene Eingangszustand, der Nachteil des RS-FF, wird im D-Flipflop verhindert, indem man nur ein Eingangssignal D hat. Es gilt: $S = D$ und $R = \overline{D}$. Dadurch ist es unmöglich, dass einmal beide Eingänge 1 sind und somit gibt es den verbotenen Zustand gar nicht. Daneben gibt es noch das Taktsignal C. Somit kann sich nur etwas ändern, wenn $C = 1$ ist. Wie der Wahrheitstabelle in Abbildung 20 zu entnehmen ist, hat der Wert Q_{n+1} immer den gleichen Wert wie D_n. Das D-FF bildet den Grundstein für Mehrbitspeicher.



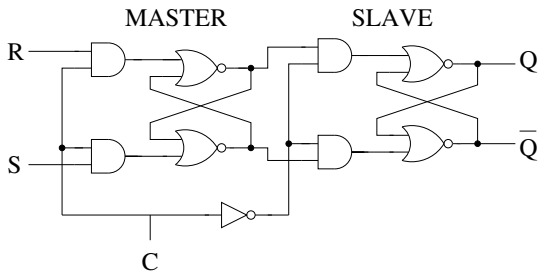


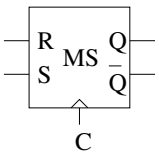
Eingänge		Ausgang
C	D_n	Q_{n+1}
0	0	Q_n
0	1	Q_n
1	0	0
1	1	1

Abbildung 20: D-Flipflop

Master-Slave RS-Flipflop

Das Master-Slave RS-Flipflop ist auch eine Art Weiterentwicklung des RS-FF. Wie aus der Abbildung 21 ersichtlich ist, besteht es aus zwei taktgesteuerten RS-FFs und einem Inverter. Dieser macht, dass der Takt vom zweiten Flipflop immer das Komplement des Taktsignals des ersten Flipflops ist, was sicherstellt, dass keine unkontrollierte Rückkopplung entstehen kann, wenn ein Ausgangssignal mit einem Eingang verbunden wird. Das erste RS-FF nennt man Master (Meister) das zweite Slave (Sklave). Master-Slave basierte Flipflops sind flankengesteuert, das bedeutet, dass sie ihren Zustand nur während der steigenden Flanke des Taktsignals C ändern (Abbildung 21).



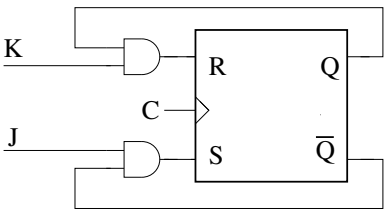


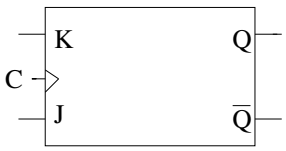
Eingänge		Ausgänge	
R	S	Q	P
0	0	speichern	
0	1	1	0
1	0	0	1
1	1	verboten	

Abbildung 21: Master-Slave RS-Flipflop

JK Flipflop

Das JK-Flipflop ist ein rückgekoppeltes Master-Slave RS-Flipflop, das heisst, die Ausgänge werden zum Eingang zurückgeführt. Am Eingang wird Q durch ein AND mit K und dem Taktsignal verbunden. $\overline{Q}$ wird durch das AND mit dem J und dem Taktsignal verbunden. Auch das JK-FF ist flankengesteuert. Durch diese Rückkoppelung wird der verbotene Zustand, also dass beide Eingänge 1 sind, vermieden, denn Q und $\overline{Q}$ sind stets komplementär.





Eingänge		Ausgang
J_n	K_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q_n}$

Abbildung 22: JK-Flipflop

Da der JK-Flipflop oft verwendet wird, um bestimmte Wertsequenzen zu generieren (z.B. in Zählern, siehe Sektion 2.3.3), ist es nützlich zu wissen, wie man die Eingangssignale setzen muss, um bestimmte Übergänge von Q zu generieren. Dies ist in Tabelle 2 dargestellt. Jeder mögliche Übergang kann auf zwei verschiedene Arten erfolgen. So kann z.B. der Wertübergang von 0 nach

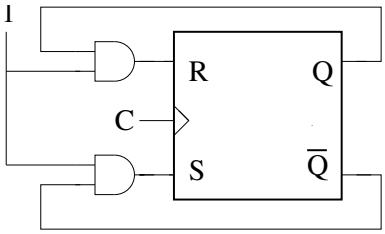
1 durch “1-Setzen” ($J = 1, K = 0$) oder “Invertieren” ($J = 1, K = 1$). Das heisst, der Wert von K hat auf diesen Wertübergang keinen Einfluss und erscheint daher in der Tabelle 2 als **x**.

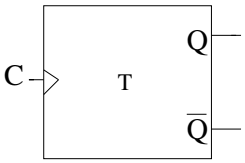
Q_n	Q_{n+1}	J_n	K_n
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Tabelle 2: Generieren von Wertübergängen im JK-Flipflop

T Flipflop

Das T-Flipflop entspricht einem JK-Flipflop, bei dem beide Eingänge konstant zu 1 verbunden sind, d.h. $J = 1$ und $K = 1$. Es invertiert bei jedem Takt seinen Zustand. Sein Name „*T*“ stammt aus dem Englischen: „*toggle*“, was hin- und herschalten bedeutet.





Eingang C	Ausgang Q_{n+1}
0	Q_n
1	$\overline{Q_n}$

Abbildung 23: T-Flipflop

2.3.2 Register

Ein Flipflop speichert immer einen einzigen Binärwert 0 oder 1. Wenn grössere Zahlen gespeichert werden müssen, so sind mehrere Flipflops nötig. Ein solcher einfacher n-Bit Speicher nennt man ein Register. Alle Bits eines Registers sind mit dem gleichen Taktsignal verbunden. Da ein Register meistens den Wert eines Eingangssignals speichern muss, sind sie fast immer aus D-Flipflops gebaut.

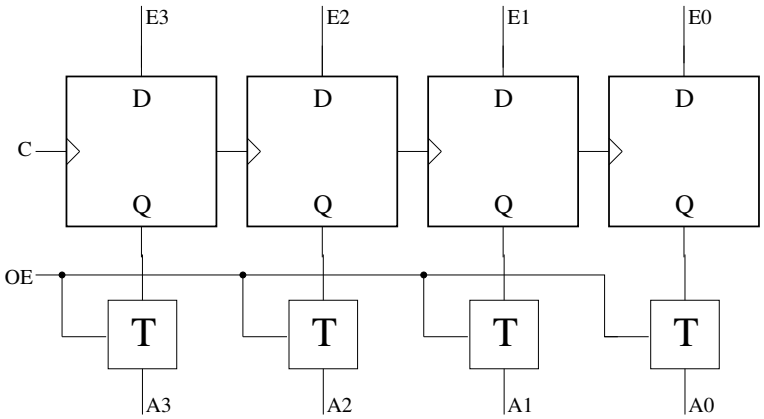


Abbildung 24: 4-Bit Register mit parallelen Ein- und Ausgängen

Sowohl der Eingang als auch der Ausgang eines Registers können parallel oder seriell sein, d.h. entweder alle Bits gleichzeitig auf verschiedenen Signallinien oder Bit für Bit auf einer einzigen

Linie. Da im MINICOMP keine Schieberegister vorkommen, beschränke ich mich hier auf Parallel-Register.

Da zu jedem Zeitpunkt höchstens ein aktives Ausgangssignal auf einen Bus geleitet werden darf, ist es wichtig, dass die Ausgangssignale eines mit dem Bus verbundenen Registers ein- und ausgeschaltet werden können. Dies geschieht am einfachsten mittels Transistor-Gates, welche den Flipflop-Ausgängen nachgeschaltet sind und durch ein gemeinsames Kontrollsignal OE (**O**utput **E**nable) aktiviert werden. So kontrollierte Ausgänge nennt man “Tri-State”, da sie neben den logischen Werten 0 und 1 auch einen dritten Zustand “nicht-verbunden” (oder auch “hoch-ohmig”) annehmen können.

Abbildung 24 zeigt ein einfaches 4-Bit Register mit parallelen Ein- und Ausgängen.

2.3.3 Zähler

Wie ein Register, ist auch ein Zähler aus mehreren Flipflops gebaut. Während aber ein Register beliebige Werte speichern und später wieder ausgeben kann, hat der Zähler den Zweck, eine genau festgelegte Abfolge (Sequenz) von Werten automatisch zu generieren, je ein Wert pro Taktsignal.

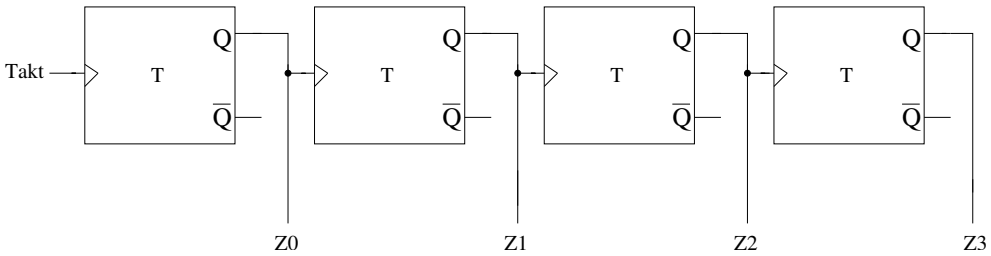


Abbildung 25: T-basierter 4-Bit Zähler für die Werte 0 bis 15

Oftmals zählt ein Zähler ganz normal von 0 beginnend bis $2^n - 1$ (n = Anzahl Bits) und beginnt dann wieder bei 0. Ein solcher einfacher Zähler entspricht einer Kette von n T-Flipflops, bei welcher immer der Ausgang eines T-Flipflops mit dem Takteingang des nächsten verbunden wird, wie in Abbildung 25 gezeigt wird.

Q_1^n	Q_0^n	Q_1^{n+1}	Q_0^{n+1}	J_1^n	K_1^n	J_0^n	K_0^n
0	0	0	1	0	x	1	x
0	1	1	0	1	x	x	1
1	0	0	0	x	1	0	x
1	1	x	x	x	x	x	x
			Funktion:	Q_0^n	1	$\overline{Q_1^n}$	$\overline{Q_1^n}$

Tabelle 3: Wahrheitstabelle eines einfachen Zählers aus zwei JK-Flipflops

Für kompliziertere Zähler wird die Abfolge der zu generierenden Werte durch eine Wahrheitstabelle festgelegt, in welcher die Werte des folgenden Takts ($n + 1$) als Funktion der Werte des jetzigen Takts (n) eingetragen sind. Wenn JK-Flipflops verwendet werden, werden die für die Übergänge nötigen Kontrollsignale (siehe Tabelle 2 auf Seite 25) als zusätzliche Kolonnen in die Wahrheitstabelle eingefügt. Durch Vereinfachung, wenn nötig mit Karnaugh-Diagrammen, können nun die Schaltfunktionen für die J- und K-Eingänge bestimmt werden.

Ein einfaches Beispiel ist ein Zähler, der die Wertesequenz 0, 1, 2, 0, 1, 2, ... generiert. Da die Sequenz nur aus drei Werten besteht, genügen hier zwei JK-Flipflops. In Tabelle 3 ist die entsprechende Wahrheitstabelle zu sehen, sowie in der untersten Zeile die resultierenden vereinfachten

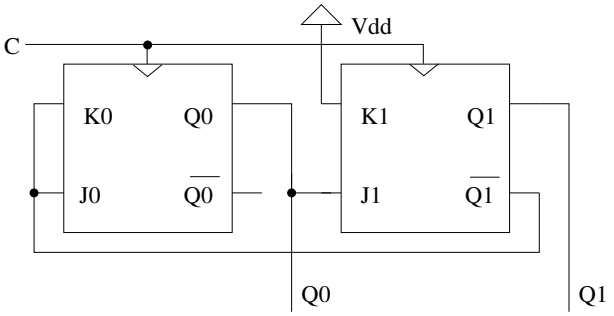


Abbildung 26: JK-basierter Zähler für die Sequenz 0-1-2

Funktionen für die J- und K-Eingänge. Daraus resultiert der in Abbildung 26 abgebildete Schaltplan.

2.3.4 Speicher

Ein Speicher besteht aus zwei Hauptteilen, nämlich:

- mehreren Registern, in welchen die Daten gespeichert und abgerufen werden können
- und einer Adressdekodierung, die entscheidet, für welches Register die Lese- oder Schreiboperation bestimmt ist.

Ein Speicher wird über die folgenden Signallinien angesprochen:

- ein Adressbus, welcher die Adresse des zu lesenden oder zu schreibenden Speicherwortes enthält;
- ein Datenbus, von welchem das zu schreibende Wort geholt oder auf welchen das zu lesende Wort ausgegeben wird;

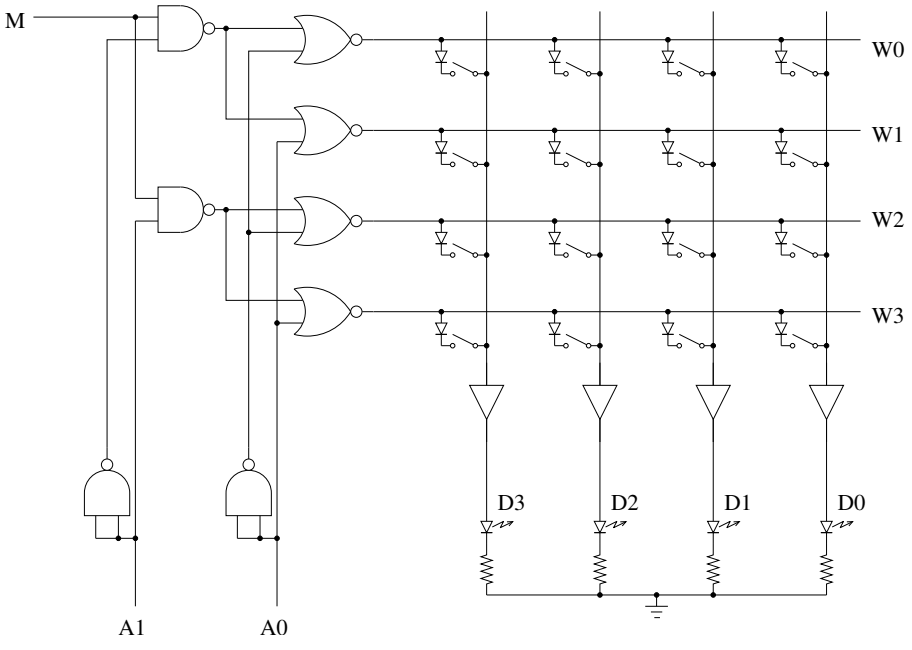


Abbildung 27: Grundidee des Lesespeichers (ROM)

- Kontrollsignale für das Lesen und Schreiben des Speichers.

Es gibt zwei verschiedene Arten von Speicher, den Lese-Schreib-Speicher und den Nur-Lese-Speicher. Der Lese-Schreib-Speicher erlaubt sowohl Lese- wie auch Schreiboperationen und basiert auf Registern, welche auf Flipflops basieren. Der Nur-Lese-Speicher, kurz Lesespeicher oder ROM (**R**ead **O**nly **M**emory) kann vom Programm her nur gelesen, jedoch nicht geschrieben werden, weil er nicht auf Flipflops basiert, sondern die Binärwerte mittels anderen Mechanismen definiert werden. Lesespeicher dienen dem Speichern von Programmen und Konstanten.

Abbildung 27 zeigt die Grundidee eines einfachen 4-Wort Lesespeichers mit 4 Bits pro Wort. Der Adressbus besteht aus den zwei Adresslinien A_0 und A_1, der Datenbus aus den vier Datenlinien D_0 bis D_3, welche in diesem Beispiel zur Darstellung des gelesenen Speicherwortes gepuffert und durch Leuchtdioden (LEDs) angezeigt werden. Die Werte der einzelnen Bits sind durch die Stellung der entsprechenden Schalter bestimmt. Die Dioden stellen sicher, dass die geschlossenen Schalter der nicht adressierten Speicherworte den Datenbus nicht kurzschliessen.

2.3.5 GAL - Generic Array Logic

Ein GAL (**G**eneric **A**rray **L**ogic) ist ein multi-funktioneller Baustein, dessen genaue Funktion mit einem “GAL-Brenner” (z.B. Galep) programmiert werden kann.

Da alle damit realisierbaren Funktionen (wie z.B. kombinatorische Funktionen, Multiplexer, Register, Zähler) in den vorherigen Sektionen schon erklärt wurden, kann ich mich hier kurz fassen.

Der Grund, warum ich für meinen MINICOMP GALs verwendete, hat nicht mit der Art der zu implementierenden Logik zu tun, sondern damit, dass ich mir damit viel Arbeit ersparen konnte.

Das GAL, das ich in meiner Arbeit haupsächlich anwendete, ist der Typ 16V8. Dieser hat 8 “Nureingänge” und 8 konfigurierbare Ein- und/oder Ausgänge. Jeder Ausgang kann als boolsche Summe von bis zu acht beliebigen boolschen Produkten definiert werden. Die resultierende logische Funktion kann entweder direkt zum Ausgangspin geleitet werden (kombinatorischer Ausgang) oder zum

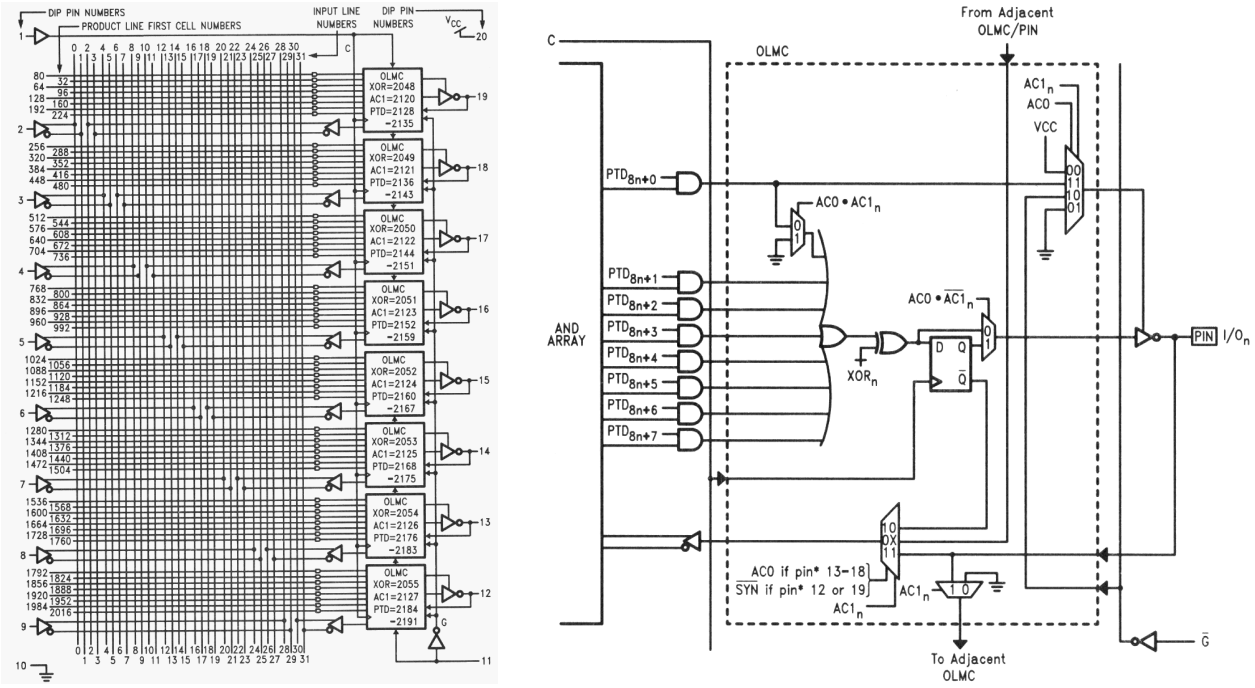


Abbildung 28: GAL: Gesamtschaltplan und Makrozelle OLMC (aus [7] Seite 2-142f)

Eingang eines dazwischengeschalteten D-Flipflops. Der Ausgang des D-Flipflops ist dann mit dem Ausgangspin verbunden und wird auch zurückgeleitet zu den Produktetermen, wo sie als Eingangssignale rückgekoppelt werden können, was z.B für die Realisierung von Zählern gebraucht wird. Alle Ausgänge können bei Bedarf als Tri-State-Augänge konfiguriert werden.

Abbildung 28 zeigt den generellen Aufbau des GAL-Bausteins 16V8 (links) sowie den detaillierten Schaltplan einer einzelnen Makrozelle. Weiterführende praktische Informationen zu GALs findet man in [7], mehr theoretische Erklärungen in [5], Seite 205ff.

3 MINICOMP - mein Minimalcomputer

3.1 Architektur

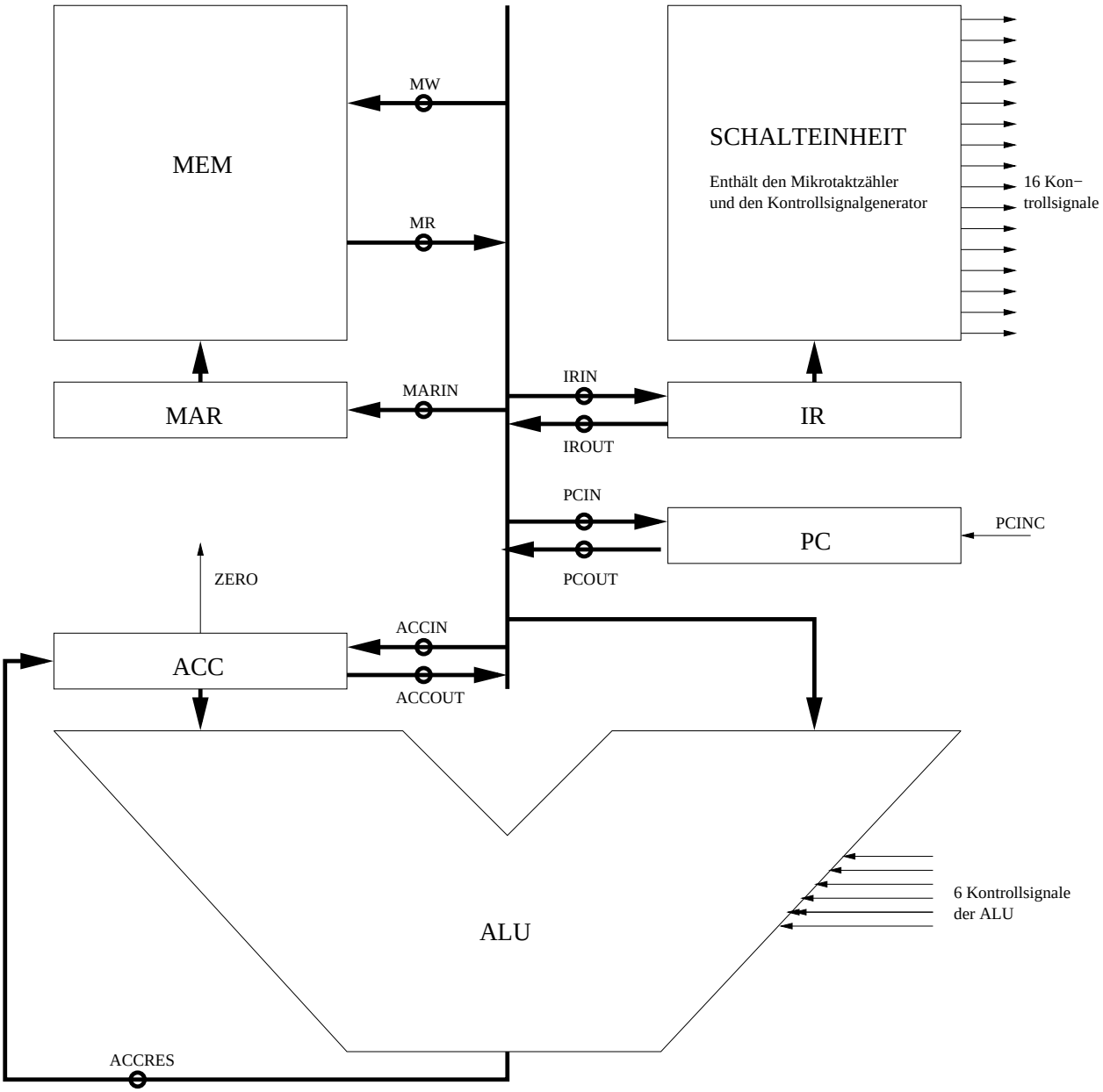


Abbildung 29: Die Systemarchitektur von MINICOMP

Man kann MINICOMP in sieben Teile unterteilen. Ich werde hier kurz zu jedem etwas sagen. Genauere Angaben folgen jedoch in den nächsten Kapiteln.

ALU Arithmetisch-Logische-Einheit:
Die ALU ist das Rechenwerk des Computers. In ihr geschehen alle Operationen, wie addieren, subtrahieren, AND oder OR.

ACC Akkumulator:
Der Akkumulator ist ein der ALU vorangehendes Register. Darin kann ein 8-Bit Wert kurzfristig gespeichert werden. Seinen Wert holt der Akkumulator entweder vom Datenbus oder

vom Resultat der ALU. Der Wert des Akkumulators ist ständig mit dem A Eingang der ALU verbunden. Der ACC hat aber auch einen Ausgang auf den Datenbus.

IR Instruktionsregister:
Die Funktion des IRs ist den abzuarbeitenden Befehl (die Instruktion) zu speichern. Dies ist notwendig, da er während des ganzen Programmschritts der Schalteinheit zur Verfügung stehen muss.

Schalteinheit :
Die Schalteinheit ist der Ort, wo die Kontrollsignale generiert werden. Sie besteht aus dem Mikrotaktzähler und dem Signalgenerator.

MEM Memory:
Im Memory werden sowohl das Programm, als auch Werte gespeichert. Das Memory von MINICOMP hat 28 8-Bit Speicherplätze, wovon 24 nur Lesespeicher sind und vier Lese-Schreib-Speicherplätze.

MAR Memory-Adress Register:
Das MAR ist ein Register, worin die Speicheradresse gespeichert wird, auf welche nächstens zugegriffen wird.

PC Programmzähler:
Der PC ist ein Register, welches die Adresse des laufenden Befehls speichert.

3.2 Einzelteile

Um die Computerteile zusammenzuhängen, verwendete ich Flachkabel, die aus mehreren voneinander isolierten Leitungen bestehen. Auf solche Flachkabel klemmt man Stecker, die dann auf Stiftleisten passen, welche auf die Platinen gelötet sind. Ich verwendete, wo immer möglich, 10er Flachkabel. Da mein Computer mit höchtesns 8-Bit Bussen arbeitet, können so immer die Bus-Signale und die Stromversorgung auf einem Kabel transportiert werden.

3.2.1 Hauptschalter

Der Hauptschalter dient zum Ein- und Ausschalten des Computers. Von hier gelangt die 5-Volt Stromversorgung auf den ersten Busstecker. Eine 1 Ampère (1 A) Sicherung verhindert grössere Schäden bei Kurzschluss durch eine fehlerhafte Platine oder falsche Verkabelung. Der einreihige Sockel neben dem 10er-Bus kann mit einem Pull-Down SIL-Widerstand (Komponente mit mehreren gleichen Widerständen) bestückt werden, wenn keine undefinierten Signale auf dem Bus sein dürfen.

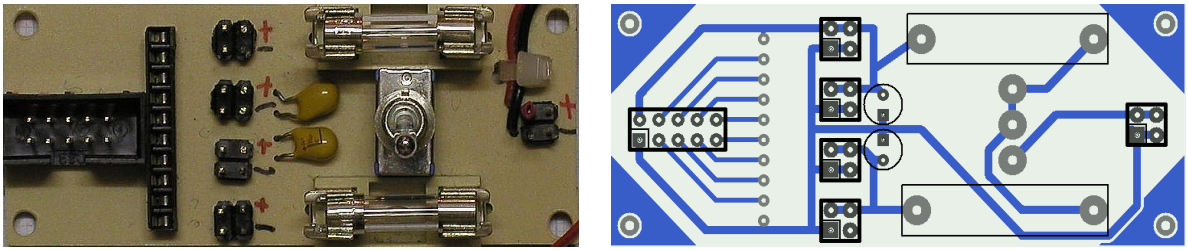


Abbildung 30: Foto und Platinen-Layout des Hauptschalters

3.2.2 Taktgenerator

Der Taktgenerator kann entweder einzelne Takte mittels Druckknopf oder ein fortlaufendes Rechtecksignal erzeugen. Ein IC mit 6 Schmitt-Trigger NOT Gates generiert Signale mit sauberen Flanken. Dies ist vorallem für die Einzeltakte wichtig, da der mechanische Druckknopf-Schalter sonst beim Ein- und Ausschalten immer mehrere ganz kurze Zwischentakte generieren würde.

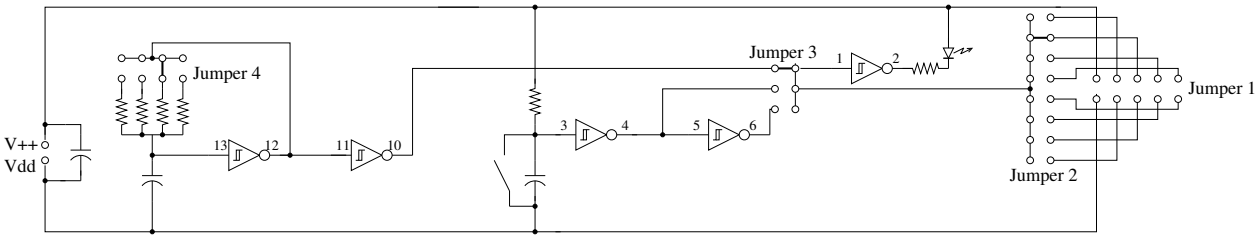


Abbildung 31: Schaltplan für den Taktgenerator

Die Funktionen des Taktgebers werden durch folgende Jumper-Blöcke bestimmt:

Jumper 1: Hier wird das Taktsignal in den 10er-Bus eingespeist.

Jumper 2: Hier wird entschieden, auf welche der acht Signallinien das Taktsignal eingespeist wird.

Jumper 3: Auswahl der Taktquelle: automatischer Takt (oben), einzelner Takt beim Drücken des Schalters (Mitte), einzelner Takt beim Loslassen des Schalters (unten).

Jumper 4: Auswahl von vier möglichen Taktfrequenzen im automatischen Taktmodus, von oben schnell bis unten langsam.

Die Frequenzen für den automatischen Modus werden durch das Produkt vom Kondensator C und dem Widerstand R bestimmt, wobei hier nur ein Kondensator C verwendet wird, aber vier verschiedene Widerstände $R1$ bis $R4$.

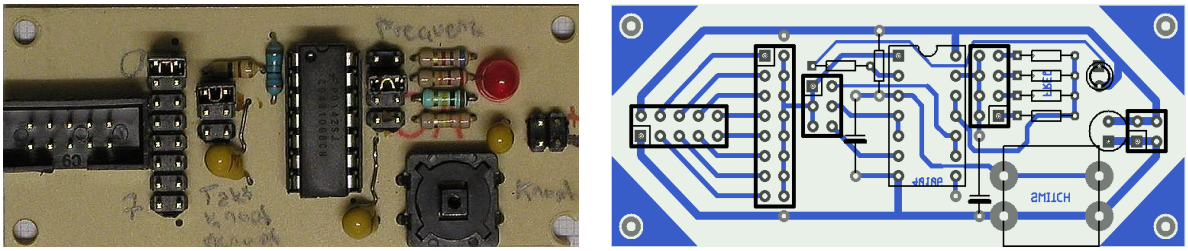


Abbildung 32: Foto und Platinen-Layout für Taktgenerator

3.2.3 8-Bit Schalter und ungepufferte LED-Anzeigen

8-Bit Schalter

Um einzelne Computerteile auszutesten, musste ich Werte herstellen können, die vom zu testenden Teil eingelesen werden und oder die Kontrollsignale dafür bilden. Um solche Daten herzustellen, baute ich mir 8-Bit Schalter. Dazu verwendete ich 8er-Dipschalter, wie ich sie auch für den Lesespeicher benutzte. Weil, wenn die Schalter geschlossen (on) sind, nicht unbegrenzt Strom fliessen darf, baute ich 1 $k\Omega$ Strombegrenzungswiderstände zwischen den Schalter und den Pluspol der Batterie. Da es auch ein klares Signal geben muss, wenn die Schalter offen sind, baute ich Pulldownwiderstände vom Wert 10 $k\Omega$ ein. Diese sollen das Signal auf 0 herunterziehen, sofern kein anderes Signal vorhanden ist.

Ungepufferte LED-Anzeige

Da man einem Bus nicht ansieht, welchen Wert er momentan hat, ich aber die Vorgänge auf MINICOMP auch darstellen wollte, baute ich ungepufferte LED-Anzeigen. Diese bestehen aus acht Leuchtdioden und einem Busanschluss. Die Leuchtdioden sind über einen Widerstand mit 0 verbunden und leuchten so nur auf, wenn der entsprechende Wert des Busses 1 ist. D0 wird immer mit der ganz rechten Leuchtdiode, D1 mit der zweiten von rechts und D7 mit der ganz links dargestellt.

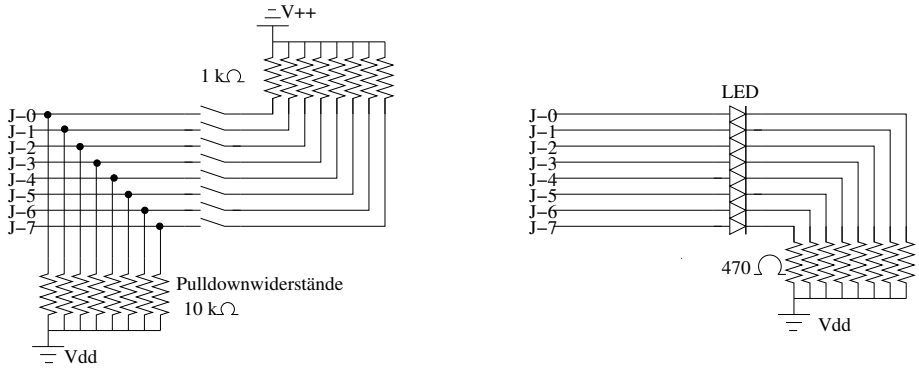


Abbildung 33: Schaltplan 8-Bit Schalter und ungepufferte LED-Anzeige

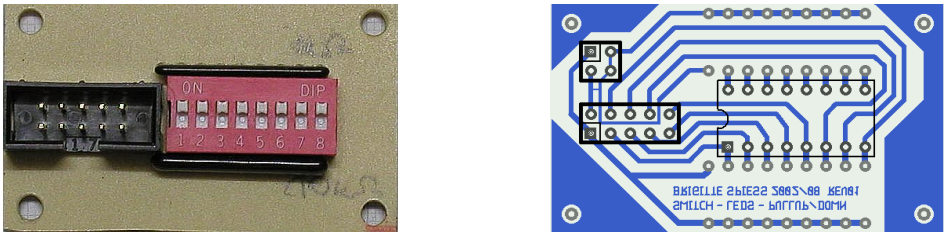


Abbildung 34: Foto und Platinen-Layout für 8-Bit Schalter und ungepufferte LED-Anzeige

3.2.4 Arithmetisch-logische Einheit (ALU)

Die Arithmetisch-logische Einheit (engl. **A**rithmetic **L**ogic **U**nit = ALU) ist das Rechenwerk des Computers. In ihr geschehen alle Operationen, wie zum Beispiel addieren und subtrahieren. Die ALU soll aber nicht nur arithmetische Verknüpfungen machen können, sie soll auch logische Verknüpfungen machen können. Meine ALU kann, neben vielen wenig nützlichen, die ich hier nicht aufzähle, folgende Operationen ausführen:

$f(A,B)$	Operation	Kontrollsignale					
		BEN	$\overline{\textit{BINV}}$	*CEN*	$\overline{\textit{CINV}}$	*CRES*	*C0*
$A+B$	Addition	1	1	1	1	0	0
$A-B$	Subtraktion	1	0	1	1	0	1
$A+1$	Inkrement	0	1	1	1	0	1
$A-1$	Dekrement	0	0	1	1	0	0
A	Identität	0	1	1	1	1	0
$\overline{A}$	Komplement	0	0	0	1	0	0
0	alle Bits auf 0	0	1	1	1	1	0
-1	alle Bits auf 1	0	0	0	0	1	1
$A\&B$	Bitweise AND	1	1	0	1	1	0
$A\#B$	Bitweise OR	1	1	0	0	1	1
$AxorB$	Bitweise XOR	1	1	0	1	0	0
$\overline{AxorB}$	Bitweise NOT-XOR	1	0	0	1	0	0

Meine ALU besteht aus acht identischen 1-Bit Modulen, welche parallel geschaltet sind und seitlich durch die Behalte Ein-/Ausgänge miteinander verbunden sind. Die Eingänge und Ausgänge eines Moduls entsprechen denjenigen eines Volladdierers: Eingänge A, B, C_{in}, Ausgänge R und C_{out}, vergleiche mit Abbildung 12 auf Seite 20. Dazu kommen Kontrollsignale, welche die ausgeführte Operation bestimmen und, mit Ausnahme von *C0*, für alle Bit-Modulen dieselben sind.

Ein ALU-Bit besteht aus einem Volladdierer, einem Multiplexer und anderen Gates, welche die Ein- bzw. Ausgänge kontrollieren, wie in Abbildung 35 dargestellt ist.

Die ALU hat folgende sechs Kontrollsignale:

BEN (**B**-**EN**able, „*enable*“ = ermöglichen)
BEN kontrolliert, ob das Eingangssignal *B* das Resultat beeinflusst. Wenn *BEN* $= 0$ ist, wird anstelle des effektiven Eingangssignals *B* ein 0 verwendet. Die *BEN*-Funktion besteht aus einen NAND, an dessen Ausgang das Signal $\overline{B\&BEN}$ entsteht.

$\overline{\textit{BINV}}$ (**B**-**INV**vert):
Wenn $\overline{\textit{BINV}} = 0$ ist, wird der Wert $\overline{B\&BEN}$ unverändert durch das XOR gelassen. Wenn aber $\overline{\textit{BINV}} = 1$ ist, wird der im *BEN*-NAND invertierte Wert $\overline{B\&BEN}$ nochmals invertiert, das heisst, man hat wieder $B\&BEN$.

CEN (**C**-**EN**able):
Das Kontrollsignal *CEN* funktioniert gleich wie *BEN*, nur dass *CEN* kontrolliert, ob das Behalte in das nächste ALU Bit weitergegeben wird.

$\overline{\textit{CINV}}$ (**C**-**INV**vert):
$\overline{\textit{CINV}}$ funktioniert gleich wie $\overline{\textit{BINV}}$, ausser dass $\overline{\textit{CINV}}$ das ausgehende Behalte C_{out} invertiert.

CRES (**C**-**RES**ultat):
Neben den genannten vier Kontrollsignalen gibt es noch *CRES*. *CRES* ist das Kontrollsignal für einen 2:1 Multiplexer. Wenn *CRES* $= 1$ ist, wird das Resultat des Volladdierers als Resultat ausgegeben, wenn es 0 ist, wird das Behalte als Resultat ausgegeben.

C0 (Behalte-Eingang am Bit 0):
C0 ist das Behalte, das im Bit 0 der ALU mit dem Eingang C_{in} verbunden ist. So ist das C_{in} im Bit 0 von aussen kontrollierbar. Bei allen übrigen Bits ist es das C_{out} (Behalte) des vorherigen Bits.

Auf dem Schaltplan in Abbildung 35 sieht man, dass die Werte aller Ein- und Ausgänge mittels Leuchtdioden (LEDs) angezeigt werden. Die Signale werden zuerst durch ein NOT geführt, bevor

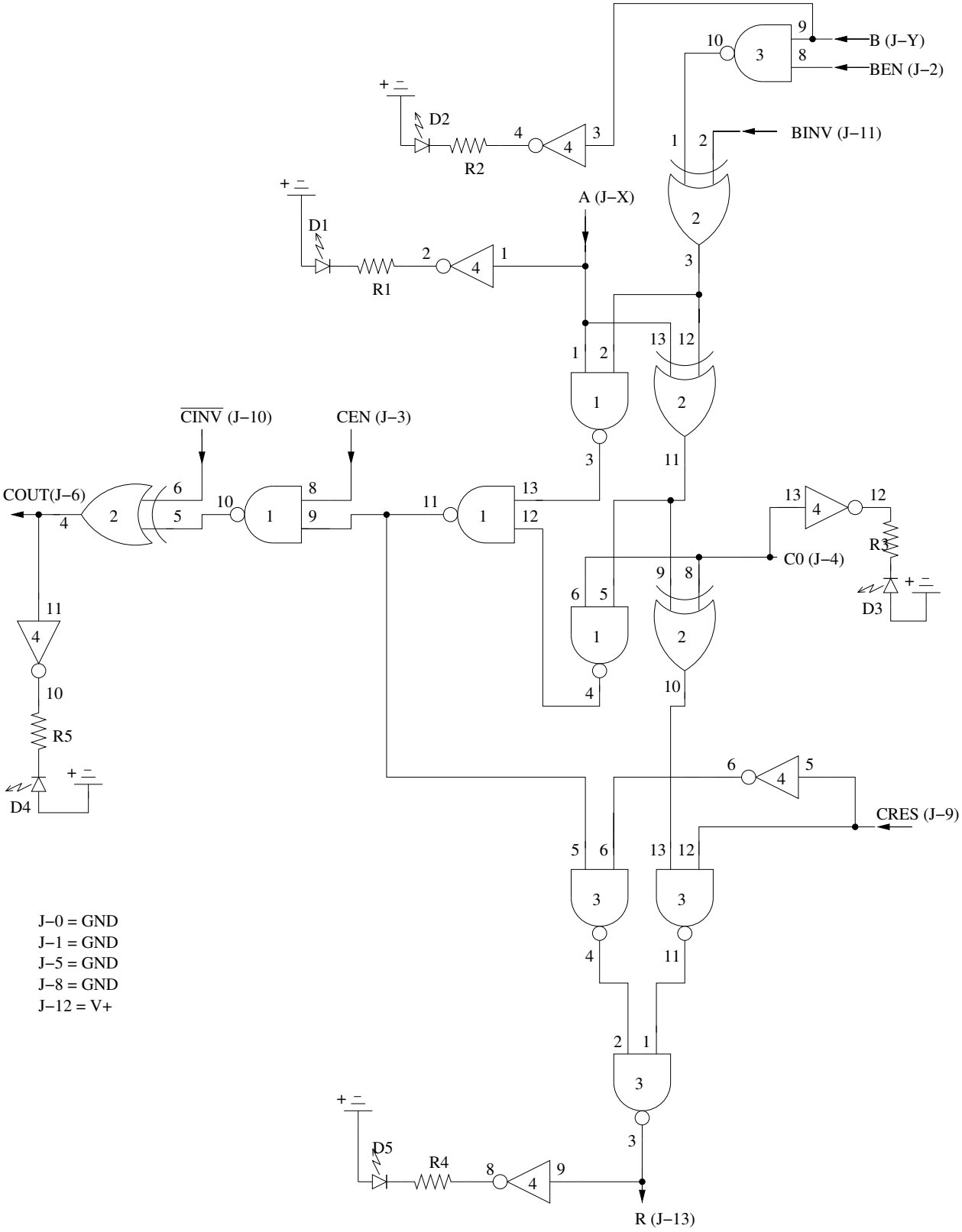


Abbildung 35: Schaltplan für ein Bit der ALU

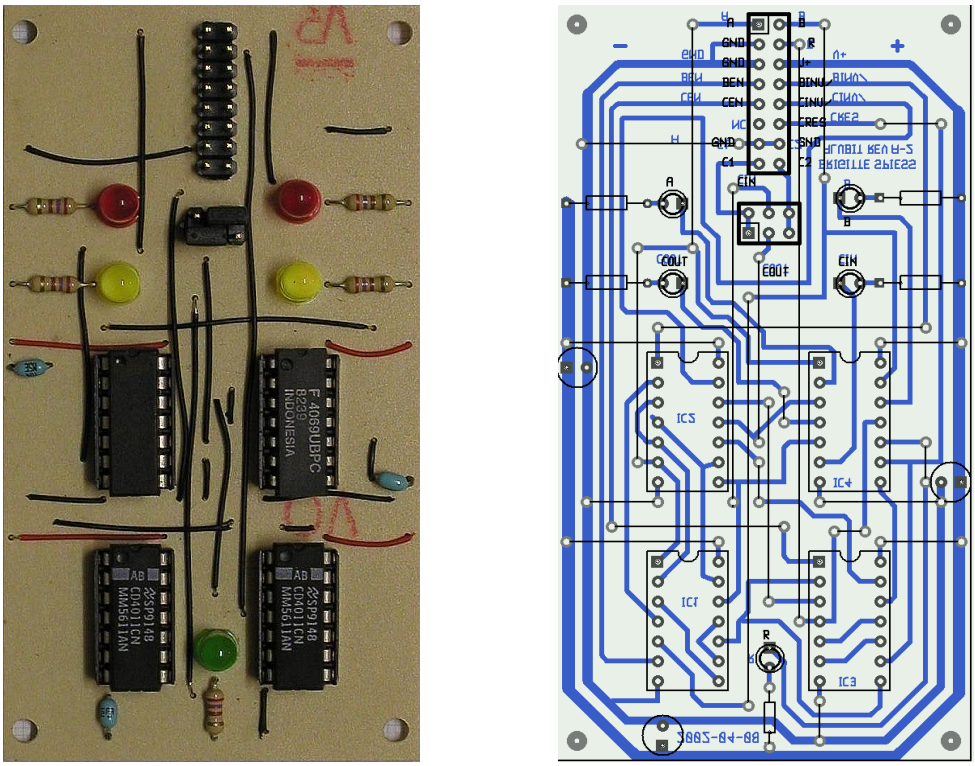


Abbildung 36: Foto und Platinen-Layout für ein Bit der ALU

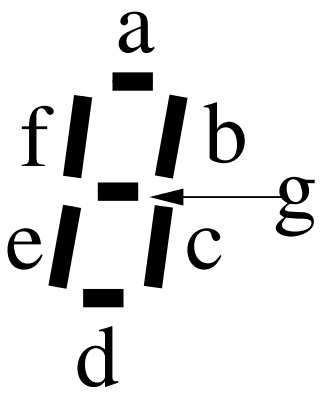
sie über einen Strombegrenzungswiderstand durch die LED zu $V+$ geführt werden. Das NOT dient somit als Puffer, damit die Spannung nicht heruntergezogen wird.

3.2.5 Zweistellige Hexadezimalanzeige

Die Hexadezimalanzeige dient der Darstellung von gespeicherten Werten oder solchen, die auf einem Bus laufen. Da jede Hexadezimalziffer genau vier Bits entspricht, ist diese Darstellungsart viel einfacher zu bauen als die Dezimaldarstellung, wo die Ziffern nicht einzelnen Bit-Gruppen zugeordnet werden können.

In der folgenden Wahrheitstabelle ist für jede der 16 Hexadezimalziffern (0 - F) angegeben, welche Segmente der 7-Segment-Anzeige leuchten müssen:

Dez-	Binärwerte				Hex-	Segmente						
Wert	i_3	i_2	i_1	i_0	Anzeige	a	b	c	d	e	f	g
0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	1	0	1	1	0	0	0	0
2	0	0	1	0	2	1	1	0	1	1	0	1
3	0	0	1	1	3	1	1	1	1	0	0	1
4	0	1	0	0	4	0	1	1	0	0	1	1
5	0	1	0	1	5	1	0	1	1	0	1	1
6	0	1	1	0	6	1	0	1	1	1	1	1
7	0	1	1	1	7	1	1	1	0	0	0	0
8	1	0	0	0	8	1	1	1	1	1	1	1
9	1	0	0	1	9	1	1	1	1	0	1	1
10	1	0	1	0	A	1	1	1	0	1	1	1
11	1	0	1	1	b	0	0	1	1	1	1	1
12	1	1	0	0	C	1	0	0	1	1	1	0
13	1	1	0	1	d	0	1	1	1	1	0	1
14	1	1	1	0	E	1	0	0	1	1	1	1
15	1	1	1	1	F	1	0	0	0	1	1	1



Durch Erstellen von Karnaugh-Diagrammen (siehe Sektion 2.1.3) habe ich für jedes Segment die entsprechende vereinfachte logische Funktion bestimmt. Daraus ergaben sich folgende boolsche Gleichungen:

$$
\begin{aligned}
a &= i_0\&\overline{i_3} \# i_1\&i_2 \# \overline{i_0}\&i_3 \# \overline{i_1}\&\overline{i_2}\&i_3 \# \overline{i_0}\&\overline{i_2} \# i_0\&i_2\&\overline{i_3} \\
b &= \overline{i_0}\&\overline{i_2} \# \overline{i_2}\&\overline{i_3} \# \overline{i_0}\&\overline{i_1}\&\overline{i_3} \# i_0\&i_1\&\overline{i_3} \# i_0\&\overline{i_1}\&i_3 \\
c &= \overline{i_1}\&\overline{i_3} \# i_0\&\overline{i_3} \# i_2\&\overline{i_3} \# \overline{i_2}\&i_3 \# i_0\&\overline{i_1} \\
d &= \overline{i_1}\&i_3 \# i_0\&\overline{i_1}\&i_2 \# i_0\&i_1\&\overline{i_2} \# \overline{i_0}\&i_1\&i_2 \# \overline{i_0}\&\overline{i_2}\&\overline{i_3} \\
e &= \overline{i_0}\&\overline{i_2} \# \overline{i_0}\&i_1 \# i_1\&i_3 \# i_2\&i_3 \\
f &= \overline{i_0}\&\overline{i_1} \# \overline{i_2}\&i_3 \# \overline{i_0}\&i_2 \# i_1\&i_3 \# \overline{i_1}\&i_2\&\overline{i_3} \\
g &= \overline{i_1}\&i_2\&\overline{i_3} \# \overline{i_2}\&i_3 \# \overline{i_0}\&i_1 \# i_0\&i_3 \# i_1\&\overline{i_2}
\end{aligned}
$$

Diese Funktionen sind für eine Hexadezimalziffer. Meine Hexadezimalanzeige soll jedoch 8 Bits, d.h. eine zweistellige Hexadezimalzahl, darstellen können. Um die ganze Logik nicht zweimal bauen zu müssen, wende ich hier eine Multiplex-Technik an, bei welcher abwechslungsweise immer nur eine Ziffer angezeigt wird. Da diese jedoch sehr schnell abwechseln, sieht das menschliche Auge immer beide Ziffern.

Die Implementierung der logischen Segmentfunktionen ist in einem GAL realisiert. Neben den acht darzustellenden Datenbits hat das GAL einen Eingang für die Zifferauswahl, 0 für die rechte, 1 für die linke Ziffer. Dieses Signal wird durch einen einfachen Oszillator erzeugt, der aus zwei Invertern und einem RC-Glied besteht. Das gleiche Signal wird auch zur Steuerung von Transistor-Gates benutzt, die immer nur Strom durch die Leuchtdioden der gerade ausgewählten Ziffer lassen.

Abbildung 37 zeigt den Schaltplan der Hexadezimalanzeige.

3.2.6 Schaltbare dreistellige Anzeige

Die Aufgabe dieser schaltbaren Anzeige ist das Darstellen von Werten, die entweder auf einem Bus laufen oder an einem Ort gespeichert sind. Da das Umrechnen von Binärwerten oder Hexadezimalwerten in das uns vertraute Dezimalsystem eher mühsam ist, baute ich neben den reinen Hexadezimalanzeigen auch noch schaltbare dreistellige Anzeigen. Das schaltbar bezieht sich darauf, dass man mit dieser Anzeige den Wert entweder im Hexadezimalsystem, im Oktalsystem,

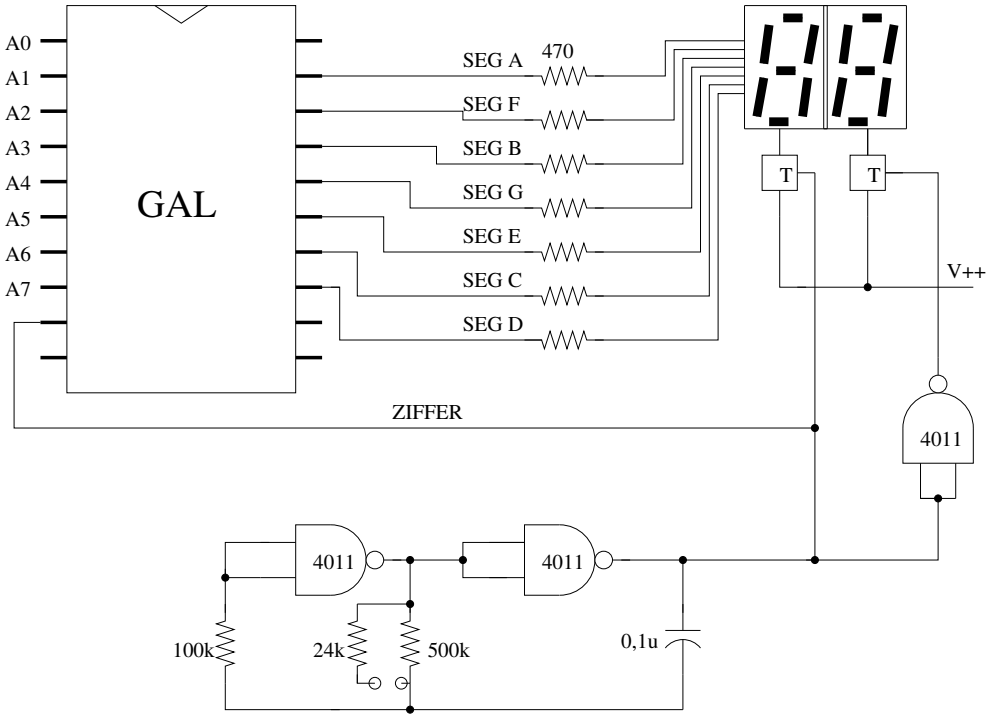


Abbildung 37: Schaltplan für die Hexadezimalanzeige

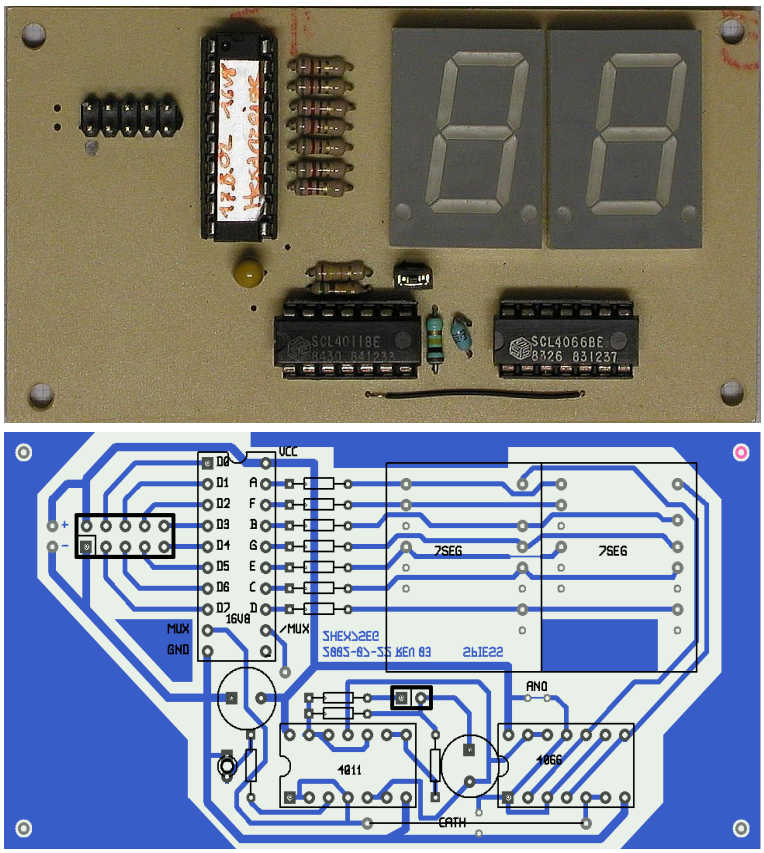


Abbildung 38: Foto und Platinen-Layout für zweistellige Hexadezimalanzeige

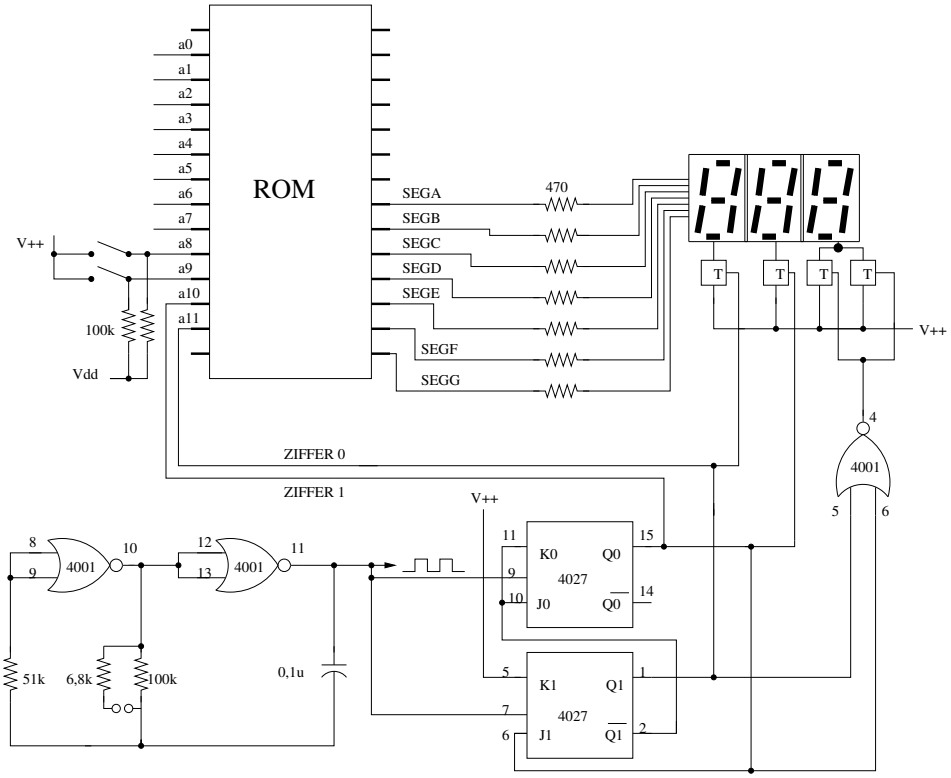


Abbildung 39: Schaltplan für schaltbare dreistellige Anzeige

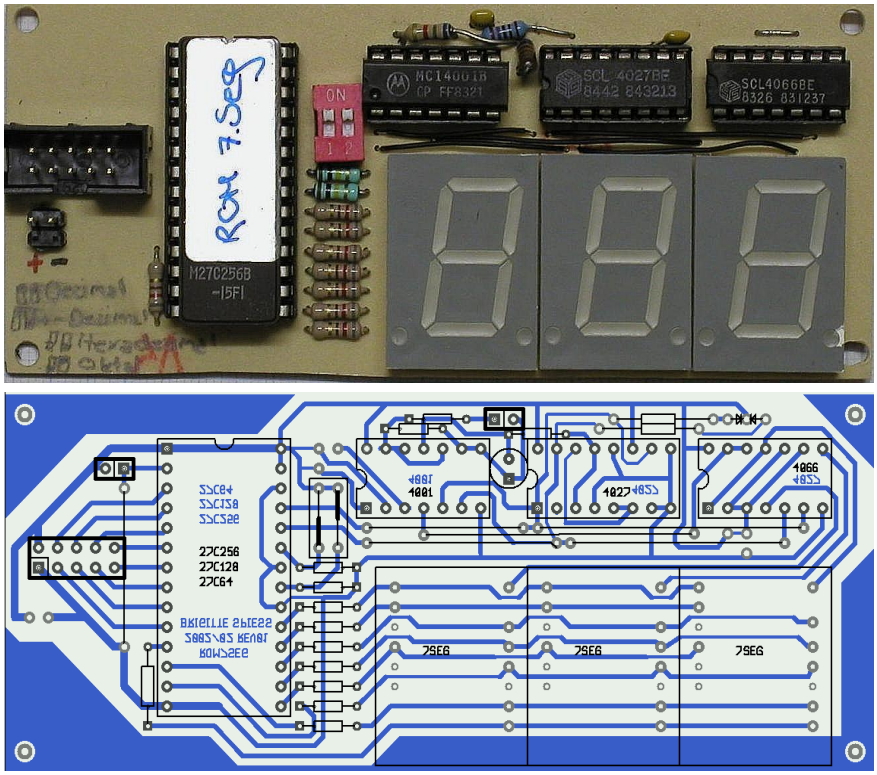


Abbildung 40: Foto und Platinen-Layout für schaltbare dreistellige Anzeige

	0000_2 0	0001_2 1	0010_2 2	0011_2 3	0100_2 4	0101_2 5	0110_2 6	0111_2 7	1000_2 8	1001_2 9	1010_2 10	1011_2 11	1100_2 12	1101_2 13	1110_2 14	1111_2 15
$0000xxxx_2$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
0	00_h	01_h	02_h	03_h	04_h	05_h	06_h	07_h	08_h	09_h	$0A_h$	$0B_h$	$0C_h$	$0D_h$	$0E_h$	$0F_h$
	000_8	001_8	002_8	003_8	004_8	005_8	006_8	007_8	010_8	011_8	012_8	013_8	014_8	015_8	016_8	017_8
$0001xxxx_2$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	+16	+17	+18	+19	+20	+21	+22	+23	+24	+25	+26	+27	+28	+29	+30	+31
16	10_h	11_h	12_h	13_h	14_h	15_h	16_h	17_h	18_h	19_h	$1A_h$	$1B_h$	$1C_h$	$1D_h$	$1E_h$	$1F_h$
	020_8	021_8	022_8	023_8	024_8	025_8	026_8	027_8	030_8	031_8	032_8	033_8	034_8	035_8	036_8	037_8
$0010xxxx_2$	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	+32	+33	+34	+35	+36	+37	+38	+39	+40	+41	+42	+43	+44	+45	+46	+47
32	20_h	21_h	22_h	23_h	24_h	25_h	26_h	27_h	28_h	29_h	$2A_h$	$2B_h$	$2C_h$	$2D_h$	$2E_h$	$2F_h$
	040_8	041_8	042_8	043_8	044_8	045_8	046_8	047_8	050_8	051_8	052_8	053_8	054_8	055_8	056_8	057_8
$0011xxxx_2$	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	+48	+49	+50	+51	+52	+53	+54	+55	+56	+57	+58	+59	+60	+61	+62	+63
48	30_h	31_h	32_h	33_h	34_h	35_h	36_h	37_h	38_h	39_h	$3A_h$	$3B_h$	$3C_h$	$3D_h$	$3E_h$	$3F_h$
	060_8	061_8	062_8	063_8	064_8	065_8	066_8	067_8	070_8	071_8	072_8	073_8	074_8	075_8	076_8	077_8
$0100xxxx_2$	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	+64	+65	+66	+67	+68	+69	+70	+71	+72	+73	+74	+75	+76	+77	+78	+79
64	40_h	41_h	42_h	43_h	44_h	45_h	46_h	47_h	48_h	49_h	$4A_h$	$4B_h$	$4C_h$	$4D_h$	$4E_h$	$4F_h$
	100_8	101_8	102_8	103_8	104_8	105_8	106_8	107_8	110_8	111_8	112_8	113_8	114_8	115_8	116_8	117_8
$0101xxxx_2$	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	+80	+81	+82	+83	+84	+85	+86	+87	+88	+89	+90	+91	+92	+93	+94	+95
80	50_h	51_h	52_h	53_h	54_h	55_h	56_h	57_h	58_h	59_h	$5A_h$	$5B_h$	$5C_h$	$5D_h$	$5E_h$	$5F_h$
	120_8	121_8	122_8	123_8	124_8	125_8	126_8	127_8	130_8	131_8	132_8	133_8	134_8	135_8	136_8	137_8
$0110xxxx_2$	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
	+96	+97	+98	+99	+100	+101	+102	+103	+104	+105	+106	+107	+108	+109	+110	+111
96	60_h	61_h	62_h	63_h	64_h	65_h	66_h	67_h	68_h	69_h	$6A_h$	$6B_h$	$6C_h$	$6D_h$	$6E_h$	$6F_h$
	140_8	141_8	142_8	143_8	144_8	145_8	146_8	147_8	150_8	151_8	152_8	153_8	154_8	155_8	156_8	157_8
$0111xxxx_2$	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	+112	+113	+114	+115	+116	+117	+118	+119	+120	+121	+122	+123	+124	+125	+126	+127
112	70_h	71_h	72_h	73_h	74_h	75_h	76_h	77_h	78_h	79_h	$7A_h$	$7B_h$	$7C_h$	$7D_h$	$7E_h$	$7F_h$
	160_8	161_8	162_8	163_8	164_8	165_8	166_8	167_8	170_8	171_8	172_8	173_8	174_8	175_8	176_8	177_8
$1000xxxx_2$	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
128	80_h	81_h	82_h	83_h	84_h	85_h	86_h	87_h	88_h	89_h	$8A_h$	$8B_h$	$8C_h$	$8D_h$	$8E_h$	$8F_h$
	200_8	201_8	202_8	203_8	204_8	205_8	206_8	207_8	210_8	211_8	212_8	213_8	214_8	215_8	216_8	217_8
$1001xxxx_2$	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
144	90_h	91_h	92_h	93_h	94_h	95_h	96_h	97_h	98_h	99_h	$9A_h$	$9B_h$	$9C_h$	$9D_h$	$9E_h$	$9F_h$
	220_8	221_8	222_8	223_8	224_8	225_8	226_8	227_8	230_8	231_8	232_8	233_8	234_8	235_8	236_8	237_8
$1010xxxx_2$	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
160	$A0_h$	$A1_h$	$A2_h$	$A3_h$	$A4_h$	$A5_h$	$A6_h$	$A7_h$	$A8_h$	$A9_h$	AA_h	AB_h	AC_h	AD_h	AE_h	AF_h
	240_8	241_8	242_8	243_8	244_8	245_8	246_8	247_8	250_8	251_8	252_8	253_8	254_8	255_8	256_8	257_8
$1011xxxx_2$	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
176	$B0_h$	$B1_h$	$B2_h$	$B3_h$	$B4_h$	$B5_h$	$B6_h$	$B7_h$	$B8_h$	$B9_h$	BA_h	BB_h	BC_h	BD_h	BE_h	BF_h
	260_8	261_8	262_8	263_8	264_8	265_8	266_8	267_8	270_8	271_8	272_8	273_8	274_8	275_8	276_8	277_8
$1100xxxx_2$	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
192	$C0_h$	$C1_h$	$C2_h$	$C3_h$	$C4_h$	$C5_h$	$C6_h$	$C7_h$	$C8_h$	$C9_h$	CA_h	CB_h	CC_h	CD_h	CE_h	CF_h
	300_8	301_8	302_8	303_8	304_8	305_8	306_8	307_8	310_8	311_8	312_8	313_8	314_8	315_8	316_8	317_8
$1101xxxx_2$	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
208	$D0_h$	$D1_h$	$D2_h$	$D3_h$	$D4_h$	$D5_h$	$D6_h$	$D7_h$	$D8_h$	$D9_h$	DA_h	DB_h	DC_h	DD_h	DE_h	DF_h
	320_8	321_8	322_8	323_8	324_8	325_8	326_8	327_8	330_8	331_8	332_8	333_8	334_8	335_8	336_8	337_8
$1110xxxx_2$	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
224	$E0_h$	$E1_h$	$E2_h$	$E3_h$	$E4_h$	$E5_h$	$E6_h$	$E7_h$	$E8_h$	$E9_h$	EA_h	EB_h	EC_h	ED_h	EE_h	EF_h
	340_8	341_8	342_8	343_8	344_8	345_8	346_8	347_8	350_8	351_8	352_8	353_8	354_8	355_8	356_8	357_8
$1111xxxx_2$	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
240	$F0_h$	$F1_h$	$F2_h$	$F3_h$	$F4_h$	$F5_h$	$F6_h$	$F7_h$	$F8_h$	$F9_h$	FA_h	FB_h	FC_h	FD_h	FE_h	FF_h
	360_8	361_8	362_8	363_8	364_8	365_8	366_8	367_8	370_8	371_8	372_8	373_8	374_8	375_8	376_8	377_8

Tabelle 4: 8-Bit Binärzahlen in vier Zahlensystemen

im Dezimalsystem ohne Vorzeichen, oder im Dezimalsystem mit Vorzeichen darstellen kann. Auf welche Art man es darstellt, hängt von der Schaltereinstellung ab:

Schalter		Darstellungsart	Bereich
off	off	Hexadezimal	$00_{\mathrm{h}} \ldots \mathrm{FF}_{\mathrm{h}}$
off	on	Oktal	$000_8 \ldots 377_8$
on	off	Dezimal mit Vorzeichen	$-128 \ldots 127$
on	on	Dezimal ohne Vorzeichen	$0 \ldots 255$

Tabelle 4 zeigt alle 256 möglichen 8-Bit Werte in den vier Darstellungsarten.

Da ich hier jedoch die Binärwerte nicht nur ins Hexadezimalsytem, sondern auch ins Oktal-, und ins Dezimalsystem mit und ohne Vorzeichen, umwandeln musste, reichte ein GAL nicht mehr. An dessen Stelle verwendete ich ein EPROM, in welchem die Wahrheitstabellen für alle möglichen Werte und alle vier Zahlensysteme direkt gespeichert sind.

Die Adresslinien des EPROMs sind dabei wie folgt belegt:

Adresslinien	Bedeutung
$a_7 \ldots a_0$	darzustellendes Datenwort
$a_9 \ldots a_8$	Darstellungsart (Schaltereinstellung)
$a_{11} \ldots a_{10}$	Ziffer

Da 12 Adresslinien benötigt werden, würde ein 4kx8 EPROM genügen. Der besseren Erhältlichkeit wegen verwendete ich jedoch grössere 64kx8 EPROMs vom Typ 27C256.

Ein Unterschied zur Hexadezimalanzeige ist, dass ich hier drei 7-Segment Anzeigen benötige, da ich mit 8 Binärwerten im Dezimal- und Oktalsystem mindestens drei Stellen brauche. Theoretisch bräuchte man sogar vier Stellen, um z.B. die Zahl `-128` darzustellen. Da das Minus an der vierten Stelle jedoch nur in Kombination mit einer 1 in der dritten Stelle vorkommt, “vermische” ich diese zwei zu einer “Kunstziffer”, bei welcher die Segmente g (Minus) und b und c (1) zugleich leuchten.

Auch hier verwende ich eine Multiplex-Technik, bei welcher die Segmenteingänge aller drei Ziffern parallel geschaltet sind und abwechslungsweise vom gleichen EPROM angesteuert werden. Die Auswahl der Ziffer ist hier jedoch komplizierter als bei der zweistelligen Hexadezimalanzeige. Um die Ziffern 0, 1 und 2 der Reihe nach zu decodieren und einzuschalten, verwende ich den in der Tabelle 3 (Seite 26) berechneten 2-Bit JK-Zähler. Die Ausgänge dieses Zählers werden einerseits auf die Adresslinien a_{10} und a_{11} des EPROMs geleitet, um die der Ziffer entsprechenden Segmente auszulesen, und andererseits zu den Transistor-Gates, welche die Anodenausgänge der 7-Segment-Anzeigen ein- und ausschalten.

3.2.7 Akkumulator ACC

Der Akkumulator ACC ist ein 8-Bit Register, welches der ALU vorgeschaltet ist. Der Wert, den er speichern soll, kann entweder vom Datenbus gelesen werden, oder vom Resultat der ALU. Der ACC hat zwei Ausgänge. Der eine führt auf den Datenbus, der andere auf den A-Eingang der ALU. Derjenige, der zur ALU führt, ist immer offen, der andere nur wenn das entsprechende Kontrollsignal aktiv ist.

Der ACC benötigt folgende drei Kontrollsignale:

ACCIN (**ACC**umulator **IN**put):
ACCIN ist ein Flankensignal. Wenn sein Wert von 0 auf 1 wechselt, wird ein Wert in das Regi-

ster geschrieben. Welcher Eingang benutzt wird, hängt vom momentanen Wert von *ACCRES* ab.

ACCRES (**ACC**umulator **RES**ultat):
ACCRES bestimmt, woher der Wert stammt, der in den Akkumulator geschrieben wird. Wenn *ACCRES* $= 1$ ist, wird das Resultat der ALU eingelesen, sonst der Datenbus.

ACCOUT (**ACC**umulator **OUT**put):
ACCOUT kontrolliert, ob der Wert, der im Akkumulator gespeichert ist, auf den Datenbus geschrieben wird oder nicht. Der Akkumulator schreibt nur auf den Datenbus, wenn *ACCOUT* $= 1$ ist.

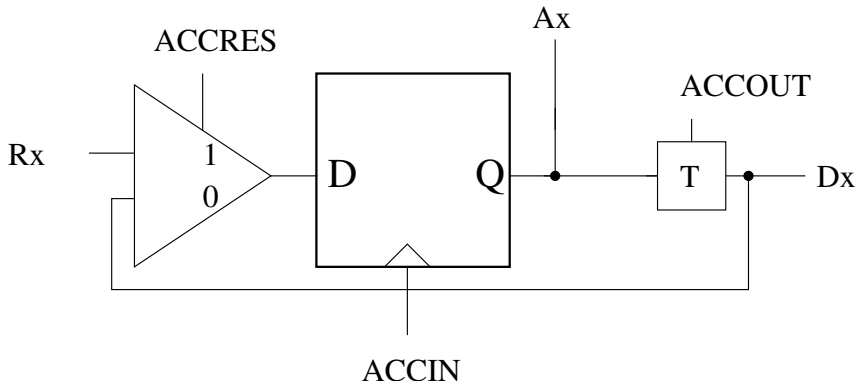


Abbildung 41: Schaltplan für ein Bit des Akkumulators

3.2.8 Instruktionsregister IR

Der Zweck des **I**nstruktions**R**egister IR ist, den abzuarbeitenden Befehl (Instruktion = Opcode und Operand) zu speichern, damit dieser während des ganzen Programmschritts der Schalteinheit zur Verfügung steht.

Technisch funktioniert das IR ähnlich wie der Akkumulator: Es kann seinen Inhalt vom Datenbus lesen und auf ihn schreiben. Seinen Inhalt gibt es zudem immer an einen permanenten Ausgang zur Schalteinheit ab. Deshalb verwendete ich die gleiche Platine wie für den Akkumulator (vgl. Abbildung 42), wobei das Kontrollsignal *ACCRES* immer 0 ist, da der Resultateingang beim IR nie benutzt wird.

Ein weiterer Unterschied ist, dass beim IR nur die ersten vier Bits sowie das höchste Bit auf den Datenbus gelassen werden. Dies ist notwendig, da das MAR (Memory-Adress-Register) nur fünf Bit Adressen hat. Die Opcodes sind so angeordnet, dass für Befehle, welche den Datenspeicher ansprechen, immer $\mathrm{IR}_7 = 1$ ist. Dies stellt die automatische Umwandlung von Datenspeicher-Operanden in Datenspeicher-Adressen sicher. Dass nur diese fünf Bits aus dem IR gelangen, wird mit der Programmierung des GALs gemacht. Siehe Anhang A.4.

Da MINICOMP immer bei dem Programmschritt P00 anfangen soll, muss das IR zu Beginn so initialisiert werden, dass sich der Befehl “GOTO P00” darin befindet. Dadurch springt der Computer zu Beginn zum Programmschritt P00 und beginnt dann mit der Abarbeitung des dort beginnenden Programms. Da ich den Opcode des Befehls “GOTO” so gewählt habe, dass er 0 ist, entspricht diese Initialisierung einfach einer Null-Setzung des IR.

Wie ich diese Initialisierung implementierte, zeigt Abbildung 43. Die Idee hinter dieser Initialisierung ist, dass beim Einschalten des Computers ein Kondensator C über einen Widerstand R aufgeladen

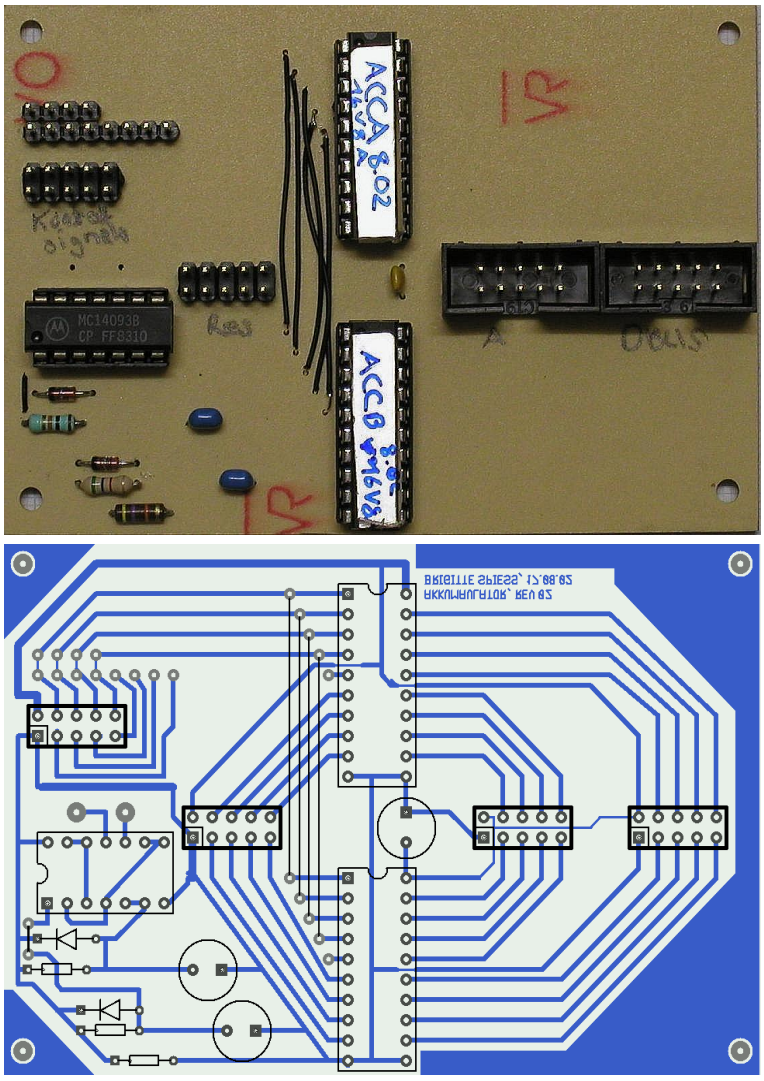


Abbildung 42: Foto und Platinen-Layout für Akkumulator und IR

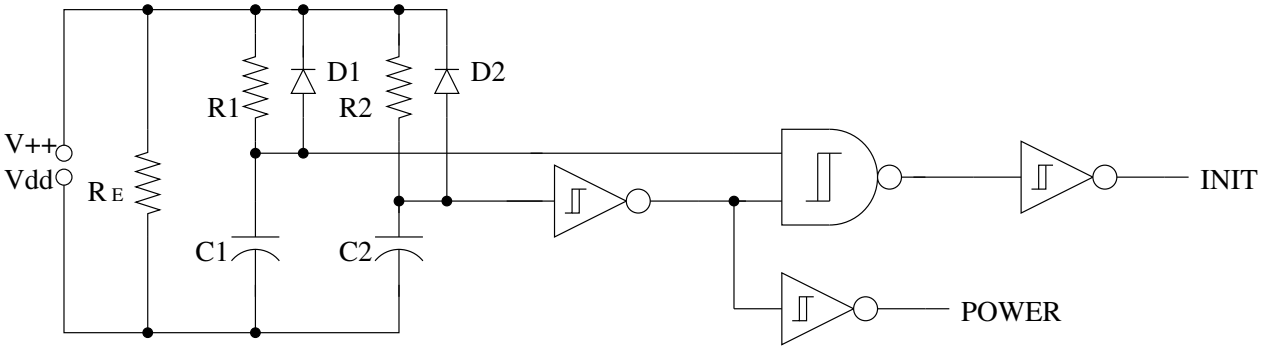


Abbildung 43: Schaltplan für *INIT* und *POWER*

wird und damit ein Schmitt-Trigger verzögert eingeschaltet wird. Die Verzögerungszeit ist ungefähr $t \approx R * C$. Für die automatische Initialisierung beim Einschalten brauchte ich zwei solche RC-Glieder, ein schnelleres und ein langsameres, welche die steigende und die fallende Flanke des Signals *INIT* bestimmen. Sobald der *INIT* Impuls beendet ist, ist das System in einem stabilen Zustand, was durch *POWER* $= 1$ angezeigt wird.

Für R_1 wählte ich 100 kΩ, für R_2 240 kΩ, für die beiden Kondensatoren C_1 und C_2 je $2,2\,\mu F$. Daraus ergibt sich ein *INIT* Signal das 220 ms nach dem Einschalten beginnt und 308 ms dauert. Nach dieser Zeit (528 ms) schaltet das *POWER* Signal auf 1.

Das Signal *INIT* verknüpfte ich dann durch ein Dioden-OR mit dem *IRIN* Schaltsignal. Somit gibt es auch einen Taktimpuls, wenn *INIT* $= 1$ ist und *IRIN* $= 0$ ist. Wenn aber *INIT* $= 1$ ist, so ist *POWER* immer 0. Also habe ich das GAL so programmiert, dass es immer 0 speichert, solange *POWER* $= 0$ ist.

Dafür, dass sich beim Ausschalten die Kondensatoren C_1 und C_2 möglichst schnell entladen können und somit wieder bereit sind für das nächste Einschalten, sorgen die umgekehrt gepolten Dioden, welche im ausgeschalteten Zustand die Ladung über R_E ableiten.

3.2.9 Programmzähler PC

Der Programmzähler PC (englisch **P**rogramm **C**ounter) ist ein Register, welches die Programmadresse des laufenden Befehls speichert. Dies ist notwendig, damit der Computer weiss, bei welchem Programmschritt er als nächstes weiter machen muss. Die nächste Programmadresse ist im Normalfall PC+1 (Inkrementierung), ausser nach einem Sprungbefehl (GOTO, IFZ, IFNZ), bei welchem der Operand in den Programmzähler geschrieben wird.

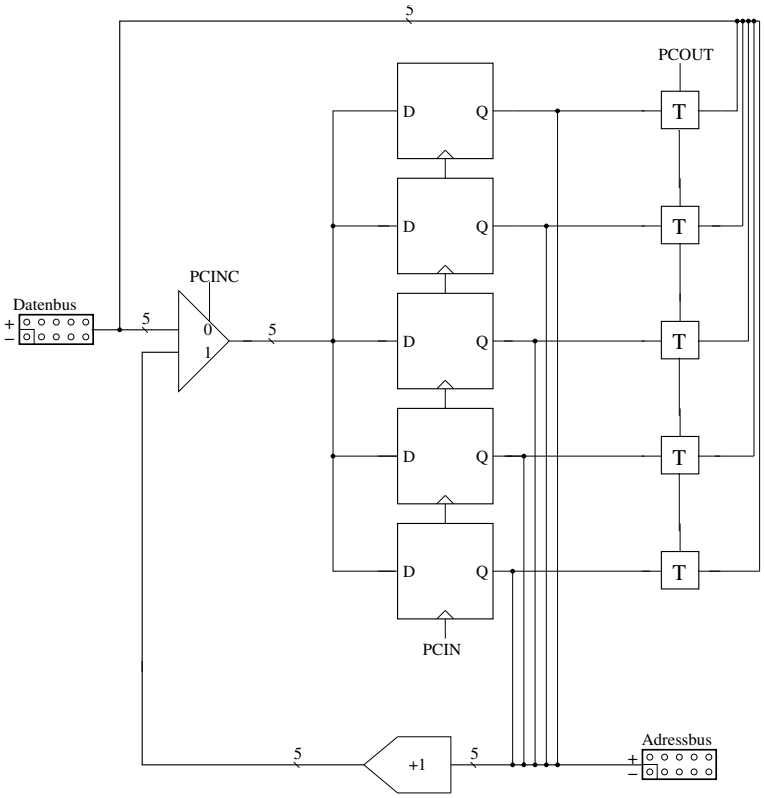


Abbildung 44: Schaltplan des Programmzählers

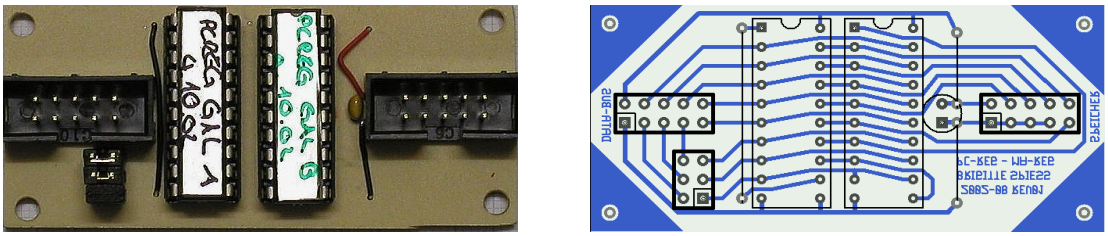


Abbildung 45: Foto und Platinen-Layout für Programmzähler

Das PC Register muss einen Wert vom Datenbus lesen und speichern können. Da die Speicheradressen nur 5 Bits lang sind, benötigt der PC auch nur 5 Bits Speicherplatz. Diese sind mit D-Flipflops in einem GAL implementiert, welches auch die Logik für die Inkrementierung enthält.

Um all diese Funktionen zur rechten Zeit zu erfüllen, benötigt der PC folgende drei Kontrollsignale:

PCIN (**PC IN**put):
: *PCIN* ist ein Flankensignal. Wenn sein Wert von 0 auf 1 wechselt, wird ein Wert in das Register geschrieben. Ob der vorhandene Wert inkrementiert wird oder ein neuer Wert vom Datenbus gelesen wird, hängt vom momentanen Wert von *PCINC* ab.

PCINC (**PC INC**rementieren):
: Wenn $PCINC = 1$ ist, wird bei steigender Flanke des Signals *PCIN* der Wert, der im PC gespeichert ist, inkrementiert. Wenn dagegen $PCINC = 0$ ist, wird der Wert vom Datenbus eingelesen.

PCOUT (**PC OUT**put):
: *PCOUT* bestimmt, ob der Wert, der im PC gespeichert ist, auf den Datenbus geschrieben wird oder nicht. Der PC schreibt nur auf den Datenbus, wenn $PCOUT = 1$ ist.

Wie Abbildung 44 zeigt, hat der PC zwei Busanschlüsse. Der Datenbusanschluss wird gebraucht, um die Daten, die auf dem Datenbus sind, zu lesen oder um auf den Datenbus zu schreiben. Der zweite Anschluss dient beim PC alleine zur Darstellung, für die Funktionalität des PCs spielt er keine Rolle.

3.2.10 Memory-Adress-Register MAR

Das **M**emory-**A**ddress-**R**egister MAR dient der Zwischenspeicherung einer Speicheradresse, auf die nächstens zugegriffen wird. Dies ist nötig, da der Datenbus nicht gleichzeitig die Adresse und den Speicherinhalt leiten kann.

Technisch gesehen ist das MAR genau gleich gebaut wie der PC in Abbildungen 44 und 45. Das MAR muss sich zwar nicht selbständig inkrementieren und auch nicht auf den Datenbus schreiben können. Wenn es diese Funktionen jedoch auch hat, so schadet dies nicht. Die Kontrollsignale *PCINC* und *PCOUT* sind beim MAR einfach immer 0.

Ein weiterer Unterschied zwischen dem MAR und dem PC ist, dass beim MAR der zweite Busanschluss nicht nur zur Darstellung dient, sondern funktionell als Adressbus notwendig ist.

3.2.11 Schalterspeicher für Programme und Konstanten

Für meinen Minimalcomputer wollte ich einen möglichst einfachen Lesespeicher bauen, der aber trotzdem programmierbar ist. Er soll ein Programm sowie konstante Daten speichern können. Zu-

erst überlegte ich mir, ihn mit einem EPROM zu bauen. Ich unterliess es dann aber, da es sehr umständlich wäre, jedesmal ein neues Programm zu schreiben. Man müsste ja immer das EPROM mit dem Löschgerät löschen und danach neu programmieren. Ich hatte das Ziel, dass mein Minimalcomputer direkt, das heisst ohne Hilfsmittel, programmierbar ist. Deshalb verwendete ich schlussendlich 8er DIP-Schalter, nach dem Prinzip, das ich schon bei der Grundidee des Speichers erklärt habe (Abbildung 27 auf Seite 27). Auf jedem DIP-Schalter kann man 8 Binärwerte einstellen, diese acht Werte entsprechen dem gespeicherten 8-Bit Wert.

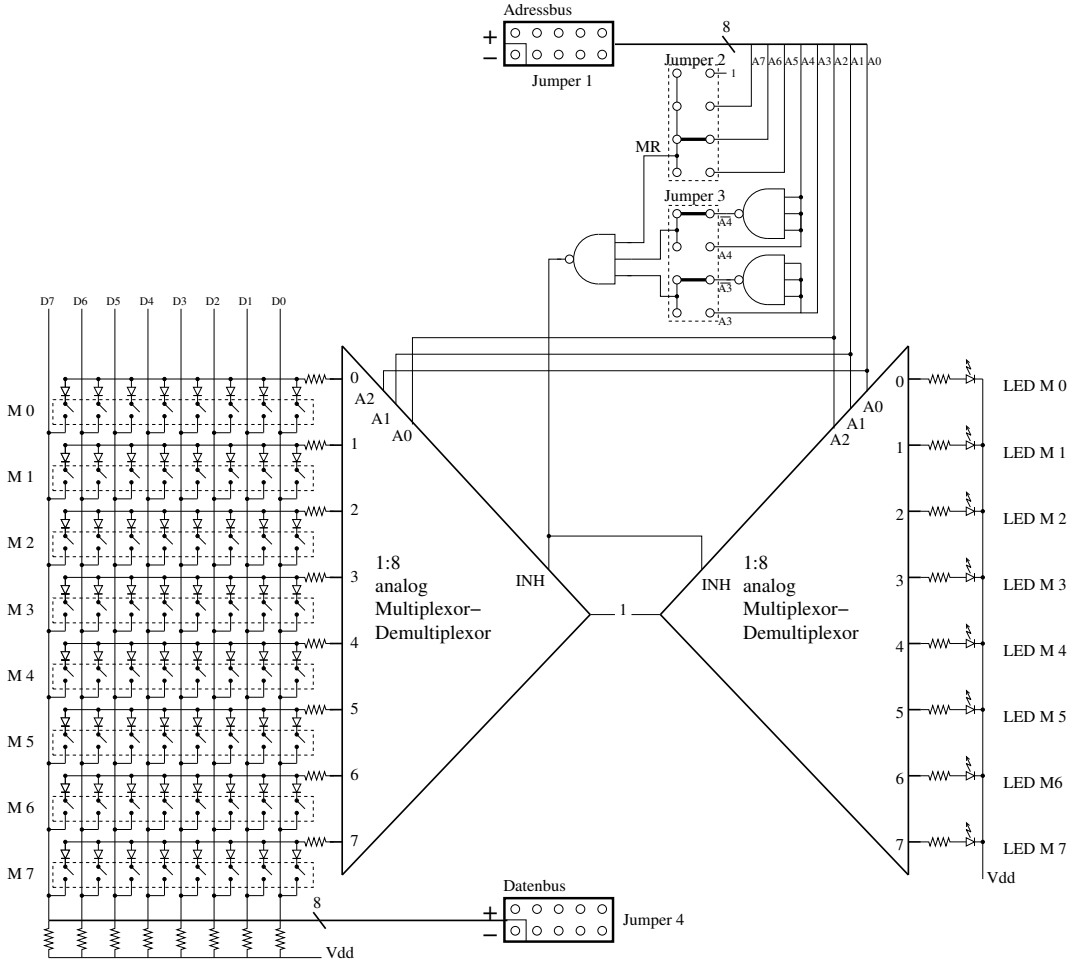


Abbildung 46: Schaltplan für Schalterspeicher

Mein Speicher besteht aus 4 Seiten (Modulen). Davon sind drei Schalterspeicher und ein Modul besteht aus den Lese-Schreibregister. Eine Schalterspeicherseite kann 8 Bytes speichern, das heisst 8 Wörter mit 8 Binärwerten. Diese 4 Seiten müssen nun verschieden adressiert sein, da ja immer nur aus einer Seite aufs Mal gelesen werden darf, also nur von der Seite mit der richtigen Adresse. Damit ich vier verschiedene Adressen auf einer Platine einstellen kann, habe ich einen 4er Jumper aufgesetzt, wie Abbildung 46 zeigt. Man muss immer zwei Jumper gesetzt haben, einen entweder bei $A3$ oder $\overline{A3}$ und den anderen entweder bei $A4$ oder $\overline{A4}$. Aber auch wenn die Adresse stimmt, darf der Wert nur gelesen werden, sofern das Kontrollsignal *MR* $= 1$ ist. Also wird die dekodierte Adresse mit dem *MR* durch ein NAND gelassen. Der so entstandene Wert gilt nun als Inhibit (english “verbieten”) des Demultiplexers.

Da die Adresse nur aus 5 Bits besteht, aber von einem 8-Bit Bus herangeführt wird, wird das Kontrollsignal *MR* auch über den Adressbus eingespeist. Ein Jumperfeld bestimmt, von welchem Bit *MR* bezogen wird.

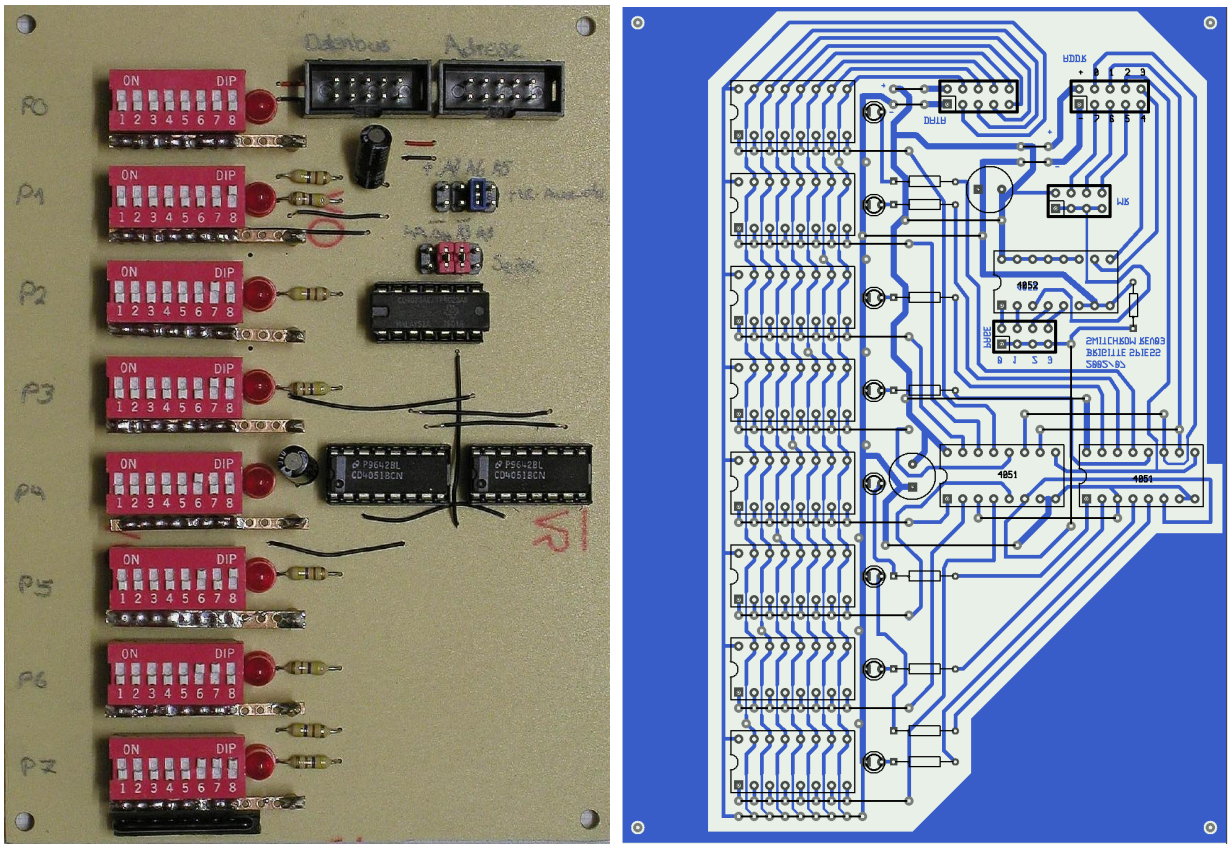


Abbildung 47: Foto und Platinen-Layout für Schalterspeicher

Welches Wort der angesprochenen Seite gemeint ist, wird durch die Adressensignale $A0$ bis $A2$ ausgesagt. Diese drei Werte sind die Adressierungswerte für den Demultiplexer, der immer nur das adressierte Wort aktiviert. Wie in Abbildung 47 gezeigt wird, verwendete ich zwei Demultiplexer. Für die Funktionalität ist jedoch nur der linke der beiden notwendig, der rechte dient alleine der Darstellung, d.h. dass immer eine Leuchtdiode bei dem Wort aufleuchtet, das gerade gelesen wird.

Um die Wörter dann auf den Datenbus zu lassen, verwendete ich ein Dioden-OR. Das bedeutet, dass ein Strang des Datenbuses 1 ist, sobald an einer Diode das Signal 1 anliegt. Da dies jedoch nur beim momentan adressierten Wort geschehen kann, müssen alle anderen zwingend 0 sein.

Am Ende des Datenbuses installierte ich Pulldown-Widerstände, die das Datenbussignal auf Null ziehen, wenn kein aktives Signal generiert wird.

Für die Funktionalität des Schalterspeichers sind folgende Jumper notwendig:

Jumper 1: Von hier bezieht der Schalterspeicher seine Adresse.
Jumper 2: Hier wird bestimmt, von welchem Bit des Adressbusses das *MR* Signal bezogen wird, oder ob es konstant auf 1 steht.
Jumper 3: Hier wird die Seitenadresse bestimmt, vgl. oben.
Jumper 4: Hier werden die 8 Werte auf den Datenbus geleitet.

Der Schalterspeicher braucht nur ein einziges Kontrollsignal:

MR (**M**emory **R**ead):
Nur wenn $MR = 1$ ist, kann etwas aus dem Speicher gelesen werden. Woher *MR* genommen wird, entscheidet ein Jumper (vgl. Jumperliste).

3.2.12 Lese-Schreib-Speicherregister für variable Daten

Das Lese-Schreib-Speicherregister ist ein Speicher für ein 8 Bit Wort, welches zum Speichern von Programmvariablen dient. Es muss sowohl Werte vom Datenbus lesen und speichern können, als auch die gespeicherten Werte auf den Datenbus geben können.

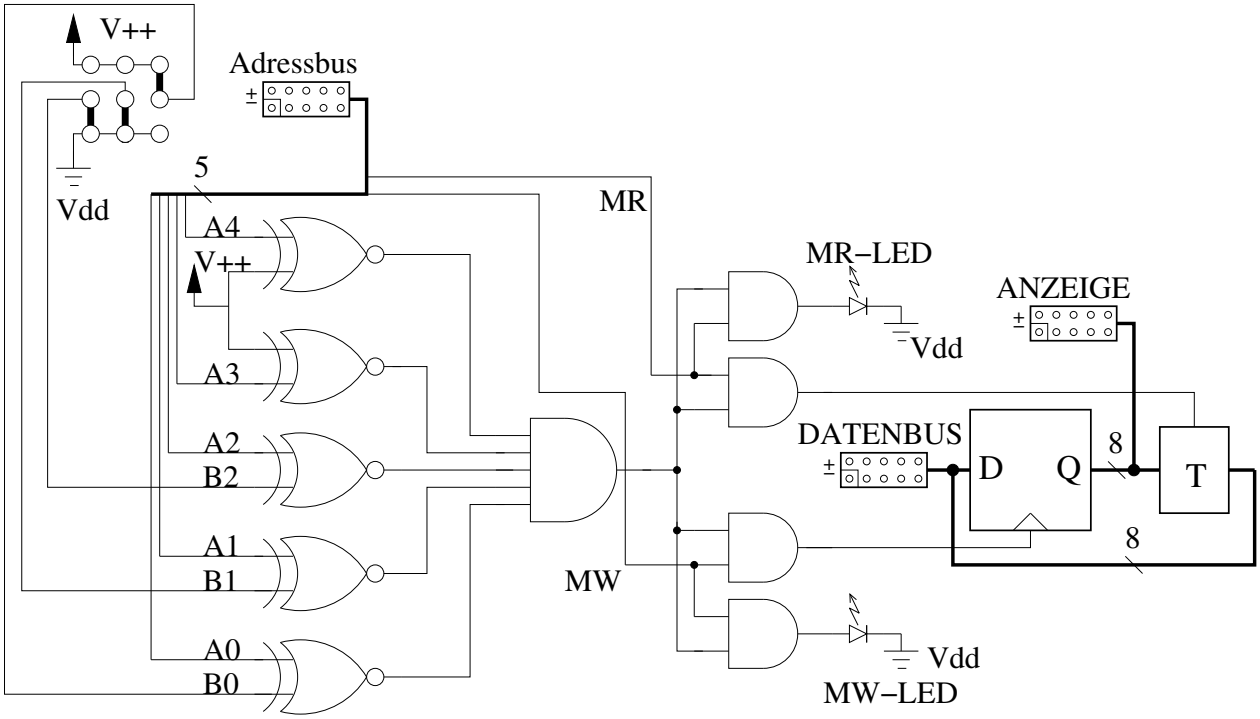


Abbildung 48: Schaltplan für den Lese-Schreib-Speicher

Wie ich schon erwähnt habe, hat MINICOMP vier Module (Seiten) für den Speicher. Der Lese-Schreib-Speicher bildet die vierte Seite. Deshalb müssen auch die obersten zwei Adressbits immer auf 1 sein, damit überhaupt ein Lese-Schreib-Speicher aktiviert wird. Ich implementierte vier Lese-Schreib-Speicher.

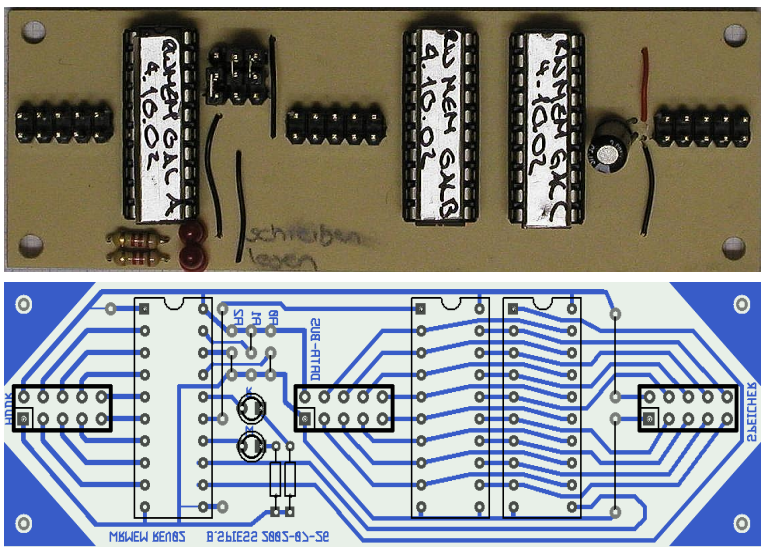


Abbildung 49: Foto und Platinen-Layout für Speicherregister

Die Adressdekodierung geschieht mit einem 3er Jumper und 5 $\overline{XOR}$ und einem 5-input-AND. Nur wenn die 3 Werte der Jumper identisch sind mit den 3 unteren Werten der Adresse, und die beiden oberen Adressbits 1 sind, kann das Speicherwort angesprochen werden. Abbildung 48 zeigt den logischen Aufbau der Adressdekodierung, welche hauptsächlich in einem GAL implementiert ist.

Damit aber wirklich ein Wert gespeichert oder gelesen werden kann, muss neben der korrekten Adresse, auch ein aktives Kontrollsignal *MW* (speichern) oder *MR* (lesen) vorhanden sein.

Um die achtbitigen Daten speichern zu können, verwendete ich acht D-Flipflops, die in einem zweiten GAL realisiert sind. Die AND-Verknüpfung zwischen *MW* und der Adressdekodierung gibt den Takt für den D-Flipflop an, das heisst der Flipflop speichert nur einen neuen Wert, wenn die Adresse korrekt ist und *MW* von 0 auf 1 wechselt (Flankensignal), sonst behält er den alten Wert.

Der gespeicherte Wert wird nur auf den Datenbus gespiesen, wenn der Inhalt des Adressbusses der Adresse des Speicherworts entspricht und $MR = 1$ ist.

Da ich bei MINICOMP aber möglichst alle Schritte darstellen möchte, ist es wichtig, den gespeicherten Wert auch dann anzeigen zu können, wenn er nicht auf den Datenbus gespiesen wird. Deshalb enthält der Datenspeicher einen direkten Ausgang, an welchen eine Anzeige angeschlossen werden kann. Die Verbindung zum Datenbus erfolgt über ein drittes GAL, bei dem im wesentlichen nur die Tri-State Ausgänge verwendet werden.

Der Lese-Schreib-Speicher benötigt zwei Kontrollsignale:

MR (**M**emory **R**ead):
Nur wenn $MR = 1$ ist, kann etwas aus dem Speicher gelesen werden.

MW (**M**emory **W**rite):
Nur wenn *MW* von 0 auf 1 wechselt, kann etwas in den Speicher geschrieben werden. Wenn aber etwas in den Speicher geschrieben werden soll, so ist es wichtig, dass vorher alle Werte wohldefiniert sind. Deshalb sagt man, dass *MW* ein Flankensignal ist.

3.2.13 Schalteinheit

Die Aufgabe der Schalteinheit ist es, den auszuführenden Befehl in die entsprechende Sequenz von Kontrollsignalen umzusetzen.

Jeder Befehl wird in mehreren Schritten ausgeführt. Jeder Schritt entspricht einem Mikrotakt. Für jeden Mikrotakt einer Instruktion muss die Schalteinheit für jedes Kontrollsignal einen genau vorgegebenen binären Wert erzeugen.

Die zu erzeugenden Kontrollsignale sind also durch den Opcode des sich im IR befindenden Befehls sowie der Schrittnummer des Mikrotakts bestimmt. Zusammen bilden diese zwei Informationen die Mikroadresse. Sie besteht aus 8 Bits, die oberen 4 entsprechen dem Opcode des Befehls, die unteren 4 der Schrittnummer. Dies erlaubt bis zu 16 Mikrotakte pro Instruktion, was für alle Befehle von MINICOMP bei weitem reicht (siehe Sektion 3.4.3).

Die Schalteinheit besteht aus einen Mikrotaktzähler, der die Schrittnummer generiert und aus dem Signalgenerator, der aus der Mikroadresse die richtigen Werte der Kontrollsignale generiert.

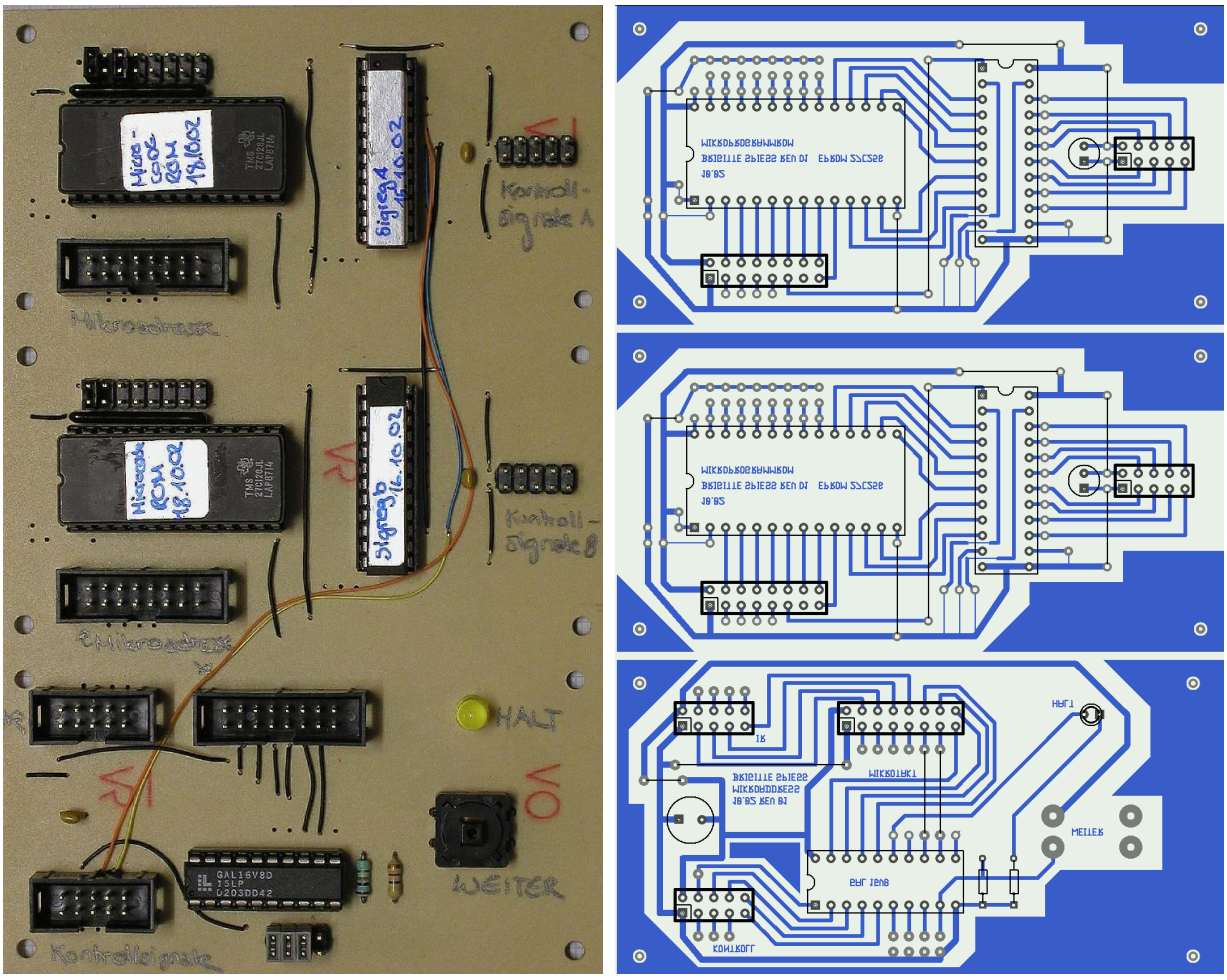


Abbildung 50: Foto und Platinen-Layout für die Schalteinheit

Mikrotaktzähler

Aufgabe des Mikrotaktzählers ist, die unteren 4 Bits der Mikroadresse zu generieren. Der Mikrotaktzähler ist ein Register, das sich selbst inkrementieren kann.

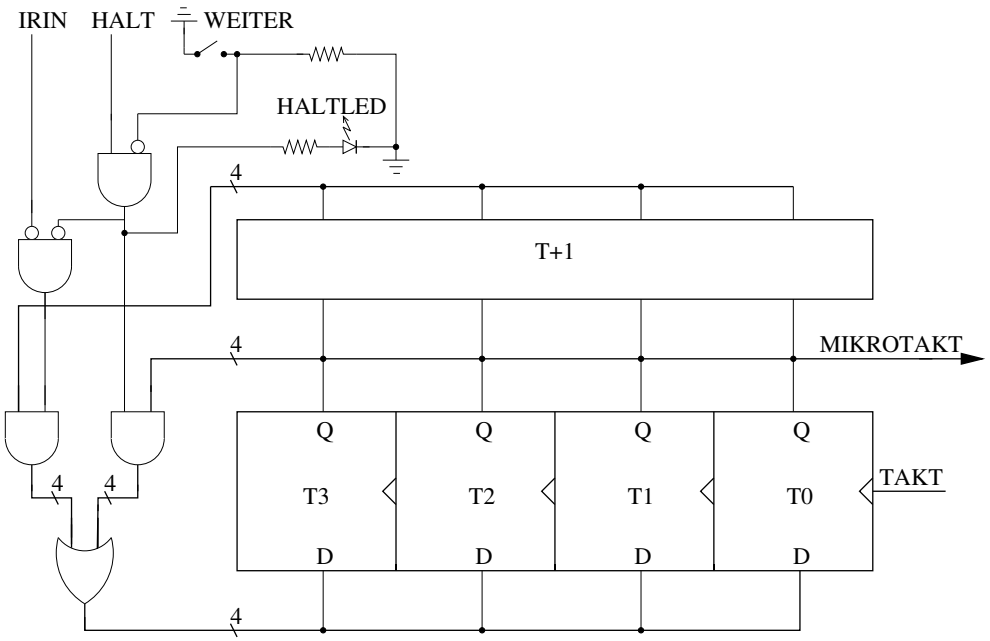


Abbildung 51: Schaltplan für den Mikrotaktzähler

Der Mikrotaktzähler basiert auf vier D-Flipflops, welche die momentane Schrittnummer speichern und zum Signalgenerator leiten. Bei jedem Mikrotakt, wird die nächste Schrittnummer aus einer der drei folgenden Möglichkeiten bestimmt:

Inkrementieren: Fortfahren mit der nächsten Mikroadresse.

Nullsetzen: Zum Schritt 0 eines (neuen) Befehls.

Beibehalten: Die Schrittnummer bleibt unverändert (HALT-Befehl).

Welche dieser drei Möglichkeiten gewählt wird, hängt von folgenden drei Kontrollsignalen ab:

IRIN (**I**nstruktions**R**egister **IN**put):
Immer wenn *IRIN* $= 1$ ist, wird der Takt auf 0 zurückgesetzt, da mit dem Einlesen eines neuen Werts ins IR immer der laufende Befehl abgeschlossen ist und somit ein neuer Befehl beginnt.

HALT :
Dieses Kontrollsignal wird ausschliesslich vom HALT-Befehl generiert. Wenn *HALT* $= 1$ ist, so bleibt der Mikrotaktzähler an derselben Mikroadresse stehen, solange bis das Signal *WEITER* $= 1$ ist.

WEITER :
Das Kontrollsignal *WEITER* wird nicht vom Signalgenerator generiert, sondern direkt auf dem Mikrotaktzähler mit Hilfe eines Druckschalters. Während man den Schalter drückt, so ist *WEITER* $= 1$, sonst ist es 0. Wenn MINICOMP durch einen HALT-Befehl angehalten wird (HALT-LED leuchtet), so wird durch Drücken der *WEITER*-Taste der HALT-Befehl beendet.

Signalgenerator

Im Signalgenerator werden für jede Mikroadresse, welche aus dem Mikrotakt und dem Opcode besteht, die richtigen Kontrollsignale generiert. Diese sind in zwei EPROMs gespeichert, je 8 Kontrollsignale in einem Chip. Die genauen Werte der Schaltsignale sind in Sektion 3.4.3 aufgelistet. Das AWK-Skript, welches die EPROM-Datei generiert, befindet sich im Anhang B.2. Da die verwendeten EPROMs vom Typ 27C128 oder 27C256 über ein Vielfaches der benötigten $16 \cdot 16 = 256$ Bytes Speicherplatz verfügen, speicherte ich die erste Hälfte der 16 Kontrollsignale in den Adressen 0-255 und die zweite Hälfte in den Adressen 256-511. Mittels eines Jumpers setzte ich dann für das erste EPROM die Adresslinie A_8 auf 0 und für das zweite auf 1. Auf diese Weise kann ich zwei identisch gebrannte EPROMs verwenden.

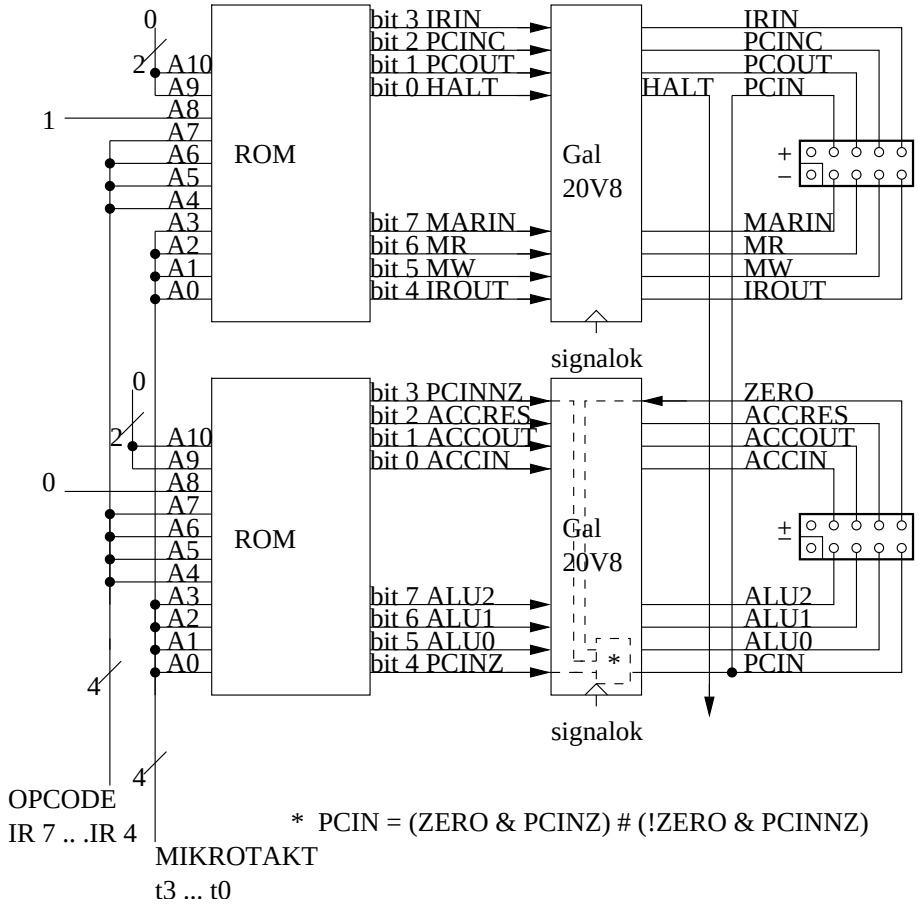


Abbildung 52: Schaltplan für Signalgenerator

Wie der Schaltplan des Signalgenerators in Abbildung 52 zeigt, werden die Ausgänge der EPROMs nicht direkt verwendet, sondern je in einem GAL zwischengespeichert, damit die generierten Kontrollsignale stabil bleiben, während die Mikroadresse geändert wird. Im ersten GAL wird zusätzlich das Kontrollsignal *PCIN* nach folgender Formel generiert:

$$PCIN = (ZERO \& PCINZ) \# (\overline{ZERO} \& PCINNZ) \ ,$$

wobei das Signal *ZERO* nicht durch die Schalteinheit generiert wird, sondern durch eine spezielle Schaltung beim Akkumulator, welche testet, ob der Wert im Akkumulator im Moment 0 ist.

3.2.14 Stecker-GAL-Stecker

Die mit Stecker-GAL-Stecker bezeichnete Platine besteht im Wesentlichen aus zwei Bus-Anschlüssen und einem dazwischengeschalteten GAL vom Typ 16V8. Der eine Bus-Anschluss führt zu den Eingängen des GALs, der andere zu den Ausgängen. Dies erlaubt das gleiche Platinen-Layout für ganz verschiedene Funktionen einzusetzen. Dazu muss man lediglich das zu diesem Zweck programmierte GAL einsetzen.

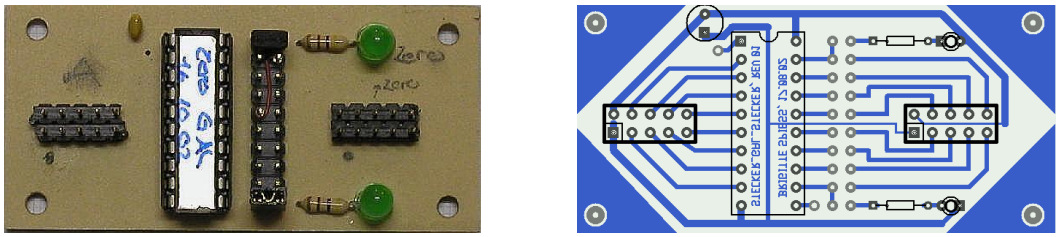


Abbildung 53: Foto und Platinen-Layout für Stecker-GAL-Stecker

Bei MINICOMP verwendete ich das Bauteil Stecker-GAL-Stecker an drei Orten.

Eines generiert die sechs Kontrollsignale der ALU (*C0*, *BEN*, $\overline{BINV}$, *CEN*, $\overline{CINV}$ und *CRES*) aus den drei vom Signalgenerator erzeugten Signalen *ALU0*, *ALU1* und *ALU2*.

Ein weiteres Stecker-GAL-Stecker Bauteil verwendete ich, um die Kontrollsignale für die Speicher, das MAR, das IR und den PC an die richtigen Stellen, wo nötig umsortiert, zu leiten, damit sie von dort aus einfach an ihren Bestimmungsort geleitet werden können.

Das letzte Stecker-GAL-Stecker Bauteil benutzte ich, um aus dem Inhalt des Akkumulators das Signal *ZERO* zu generieren. *ZERO* ist nur 1, wenn der Wert des Akkumulators 0 ist. Es ist für die bedingten Sprungbefehle IFZ und IFNZ notwendig.

Die eingesetzten GAL-Konfigurationen sind im Anhang A.6 aufgelistet.

3.3 Verknüpfung der einzelnen Computerteile

Nachdem ich nun den Aufbau und die Funktionsweise aller Einzelteile beschrieben habe, möchte ich noch kurz das Zusammensetzen dieser Teile zu einem funktionierenden Minimalcomputer beschreiben.

Bevor ich die Einzelteile mit Flachkabel verbunden habe, schraubte ich sie in einer ähnlichen Anordnung wie in Abbildung 29 (siehe Seite 30) auf eine solide Spanplatte der Grösse 49cm x 73cm. Der fertig zusammengesetzte MINICOMP ist in Abbildung 54 zu sehen. Abbildung 55 zeigt den Brettaufbau von MINICOMP als Blockdiagramm.

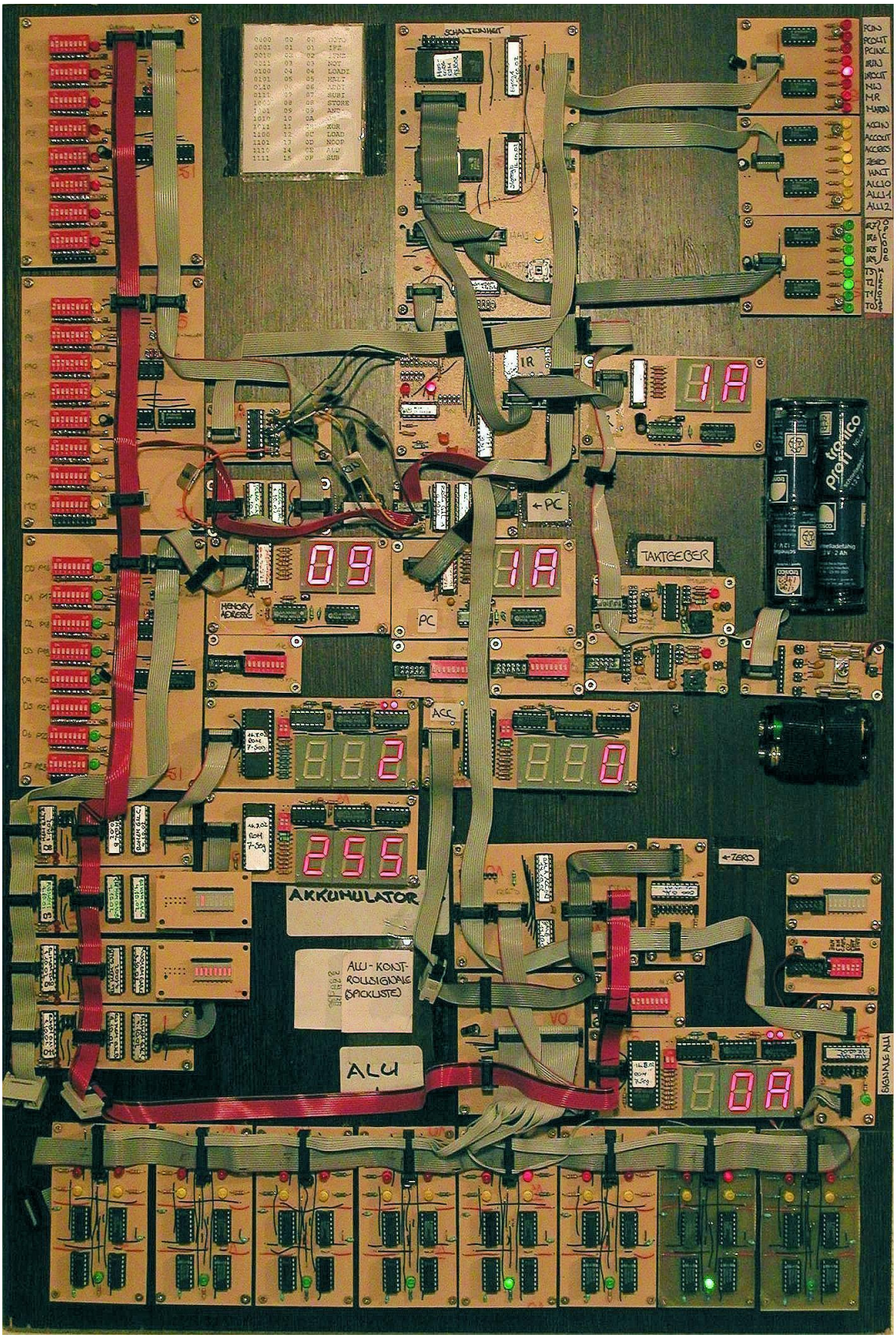


Abbildung 54: MINICOMP - mein Minimalcomputer

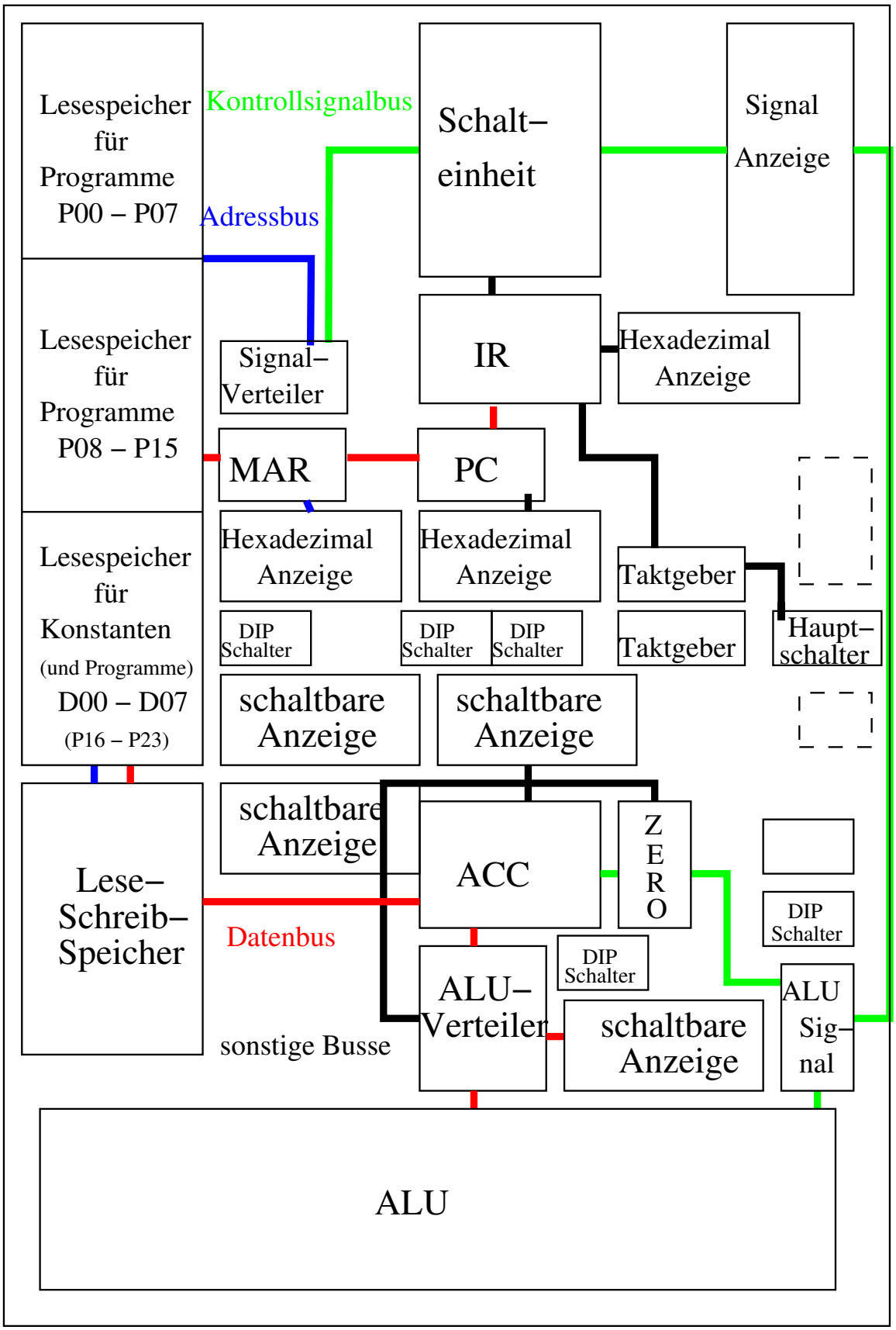


Abbildung 55: Ein Schema der Zusammensetzung von MINICOMP

3.4 Adressierung und Befehlsstruktur

Die Speicheradressen von MINICOMP bestehen aus 5 Bits, d.h. die maximale Anzahl Speicherworte ist $2^5 = 32$. Der Speicherraum ist teilt sich in zwei Bereiche auf:

Programmspeicher: Die ersten 16 Speicherworte (Adressen 00_{h} bis 0F_{h}) sind für die Programmschritte P00 – P15 reserviert. Es handelt sich dabei um Nur-Lese-Speicher, deren Werte vom Programmierer durch Einstellen der Schalter definiert werden.

Datenspeicher: Die zweiten 16 Speicherworte (Adressen 10_{h} bis 1F_{h}) dienen als Datenspeicher D00 – D15 für Konstanten und Programmvariablen.

Die Datenspeicher D00 – D07 schliessen direkt an den Programmspeicher an (Adressen 10_{h} bis 17_{h}) und sind gleich gebaut wie die Programmspeicher. Deshalb können darin bei Bedarf anstelle von Konstanten auch bis zu maximal 8 zusätzliche Programmschritte P16 – P23 gespeichert werden. Zu beachten gilt einzig, dass diese Programmschritte zwar Sprungbefehle enthalten können, jedoch nie selbst das Ziel eines Sprungbefehls sein können.

Die Datenspeicher D08 – D15 (Adressen 18_{h} bis 1F_{h}) sind für Lese-Schreib-Speicher reserviert, in welchen Programmvariablen gespeichert werden können. Da nur vier Lese-Schreibspeicher D08 – D11 in MINICOMP realisiert wurden, sind D12 – D15 nur in Theorie vorhanden.

Die Befehle für MINICOMP sind 8 Bits lang. Die ersten 4 Bits (Bit 0 bis Bit 3) bilden den Operand, während die Bits 4 bis 7 der Opcode sind. Der Opcode identifiziert den Befehl, das bedeutet, dass gleichartige Befehle immer den gleichen Opcode haben, ihre Operanden können jedoch verschieden sein. So hat z.B. der Opcode vom Befehl GOTO den Wert 0000_2.

3.4.1 Operanden

MINICOMP hat drei verschiedene Arten von Operanden:

Direkter Operand: 0 – 15

Der direkte Operand gibt den Wert an, der bei dieser Instruktion verrechnet werden muss. Die Instruktionen, die einen direkten Operand haben, sind durch ein I am Ende des Namens gekennzeichnet, wobei das I **I**mmediate (sofort) bedeutet. Ein Beispiel für eine solche Instruktion ist ADDI, wobei der Operand zum Wert des Akkumulators addiert wird.

Programmschritt-Operand: P00 – P15

Der Programmschritt-Operand gibt eine Programmadresse an, an welcher weiterzufahren ist. Diese Art von Operand wird nur in den Sprungbefehlen benötigt, d.h. wenn das Programm nicht bei der nächsten Programmadresse weiterfahren soll. Die Sprungbefehle von MINICOMP sind GOTO, IFZ und IFNZ.

Datenspeicher-Operand: D00 – D15

Der Datenspeicher-Operand gibt immer die Datenadresse an, deren Inhalt für die auszuführende Operation verwendet wird. Wenn also die Instruktion ADD D09 ausgeführt wird, addiert MINICOMP zum Akkumulator den Wert, der im Datenspeicher D09 befindet. Die Umwandlung des Operands (00_{h} – 0F_{h}) in die richtige Speicheradressen (10_{h} – 1F_{h}) besorgt das Instruktionsregister automatisch anhand des Opcodes.

3.4.2 Instruktionen

Da MINICOMPs Opcode aus 4 Bits besteht, ist sein Befehlssatz auf maximal 16 Befehle beschränkt. Jeder dieser 16 Befehle ist im Folgenden kurz beschrieben.

Befehl:	GOTO
Opcode:	Binär: 0000_2, hexadezimal: 0_h, dezimal: 0
Operand:	Gibt die Programmadresse an, P00-P15
Beschreibung:	Beim Befehl GOTO springt der Programmzähler (PC) zum Operand.
Beispiel:	00000110_2
	Der Befehl ist GOTO (0000_2) und der Operand 6 (0110_2). Bei diesem Beispiel würde der Programmzähler zu der P06 springen und somit würde es beim Programmschritt P06 weiter gehen.

Befehl:	IFZ
Opcode:	Binär: 0001_2, hexadezimal: 1_h, dezimal: 1
Operand:	Gibt die Programmadresse an, P00-P15
Beschreibung:	Der Programmzähler springt zum Operand, sofern *ZERO*=1 ist. Sonst fährt er mit dem darauffolgenden Programmschritt weiter.
Beispiel:	00010110_2, *ZERO*=0
	Der Befehl ist IFZ (0001_2), der Operand 6 (0110_2) und *ZERO* ist 0. Weil *ZERO*=0 ist, zählt der Programmzähler weiter und kommt so zum nächsten Programmschritt.
Beispiel:	00010110_2, *ZERO*=1
	Der Befehl ist IFZ (0001_2), der Operand 6 (0110_2) und *ZERO* ist 1. Weil *ZERO*=1 ist, springt der Programmzähler auf die 6 und das Programm macht mit dem Programmschritt P06 weiter.

Befehl:	IFNZ
Opcode:	Binär: 0010_2, hexadezimal: 2_h, dezimal: 2
Operand:	Gibt die Programmadresse an, P00-P15
Beschreibung:	Der Programmzähler springt zum Operand, sofern *ZERO*=0 ist. Wenn *ZERO*=1 ist, zählt er weiter und kommt somit zum nächsten Programmschritt.
Beispiel:	00100110_2, *ZERO*=1
	Der Befehl ist IFZ (0001_2), der Operand 6 (0110_2) und *ZERO* ist 1. Weil *ZERO*=1 ist, zählt der Programmzähler weiter und kommt so zum nächsten Programmschritt.
Beispiel:	00100110_2, *ZERO*=0
	Der Befehl ist IFZ (0001_2), der Operand 6 (0110_2) und *ZERO* ist 0. Weil *ZERO* 0 ist, springt der Programmzähler auf die 6 und das Programm macht mit dem Programmschritt 06 weiter.

Befehl:	NOT
Opcode:	Binär: 0011_2, hexadezimal: 3_h, dezimal: 3
Operand:	Kein Einfluss
Beschreibung:	Der Wert, der im Akkumulator gespeichert ist, wird invertiert.
Beispiel:	00110000_2, ACC=11000101_2 Der Befehl ist NOT, der Akkumulator ist 11000101_2. Der Akkumulator wird nun invertiert und ist somit am Schluss 00111010_2.

Befehl:	LOADI
Opcode:	Binär: 0100_2, hexadezimal: 4_h, dezimal: 4
Operand:	Gibt den Wert an
Beschreibung:	Der Befehl LOADI lädt den direkten Operand in den Akkumulator.
Beispiel:	01000110_2 Der Befehl ist LOADI (0100_2) und der Operand 6 (0110_2), somit wird die Zahl sechs in den Akkumulator geladen.

Befehl:	HALT
Opcode:	Binär: 0101_2, hexadezimal: 5_h, dezimal: 5
Operand:	Kein Einfluss
Beschreibung:	Der Befehl HALT macht, dass der Programmzähler nicht mehr weiterzählt. Somit stoppt er den Verlauf des Programms, bis die Taste WEITER gedrückt wird.
Beispiel:	01010000_2 Der Befehl ist HALT. Somit bleibt der Computer dort stehen, wo er ist und die HALT-LED leuchet.

Befehl:	ADDI
Opcode:	Binär: 0110_2, hexadezimal: 6_h, dezimal: 6
Operand:	Gibt den Wert an
Beschreibung:	Der Befehl ADDI addiert den direkten Operanden zum Akkumulator. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.
Beispiel:	01100011_2, ACC=00100100_2 (36) Der Befehl ist ADDI, der Operand ist 6 und der Akkumulator ist 36. Somit rechnet der Computer 00100100_2+00000110_2 (36+6) und schreibt das Resultat in den Akkumulator. Deshalb steht am Ende 00101010_2 (42) im Akkumulator.

Befehl:	SUBI
Opcode:	Binär: 0111_2, hexadezimal: 7_h, dezimal: 7
Operand:	Gibt den Wert an
Beschreibung:	Der Befehl SUBI subtrahiert den direkten Operanden vom Akkumulator. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.
Beispiel:	01110110_2, ACC=00100100_2 (36) Der Befehl ist SUBI, der Operand ist 6. Nun rechnet der Computer; 00100100_2-00000110_2 (36-6) und schreibt das Resultat in den Akkumulator. Somit steht am Ende 00011110_2 (30) im Akkumulator.

Befehl:	STORE
Opcode:	Binär: 1000_2, hexadezimal: 8_h, dezimal: 8
Operand:	Gibt die Datenadresse an, D00-D15
Beschreibung:	Der Befehl STORE speichert den Wert, der im Akkumulator ist, in die Speicheradresse, die im Operand angegeben ist.
Beispiel:	10000110_2, ACC=00110011_2 (51) Der Befehl ist STORE, der Operand ist 6. Somit speichert der Computer den Wert des Akkumulators, 00110011_2, in D06.

Befehl:	AND
Opcode:	Binär: 1001_2, hexadezimal: 9_h, dezimal: 9
Operand:	Gibt die Datenadresse an, D00-D15
Beschreibung:	Der Operand gibt die Datenadresse an. Der Wert, der in dieser Adresse gespeichert ist, wird Bit für Bit mit dem Akkumulator durch ein AND verbunden. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.
Beispiel:	10010110_2, ACC=00110011_2, D06=00000110_2 Der Befehl ist AND, der Operand ist 6, folglich verbindet der Computer den Wert 00000110_2, der in D06 gespeichert ist, mit dem Wert des Akkumulators durch ein AND, das heisst $00000110_2 \& 00110011_2 = 00000010_2$. Der Akkumulator zeigt somit 00000010_2 an.

Befehl: OR

Opcode: Binär: 1010_2, hexadezimal: $\mathrm{A_h}$, dezimal: 10

Operand: Gibt die Datenadresse an, D00-D15

Beschreibung: Der Operand gibt die Datenadresse an. Der Wert, der in dieser Adresse gespeichert ist, wird Bit für Bit mit dem Akkumulator durch ein OR verbunden. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.

Beispiel: 10100110_2, ACC=00110011_2, D06=00000110_2

Der Befehl ist OR, der Operand ist 6, folglich verbindet der Computer den Wert 00000110_2, der in D06 gespeichert ist, mit dem Akkumulator, 00110011_2, durch ein OR. $00000110_2 \# 00110011_2$=00110111_2. Der Akkumulator zeigt somit 00110111_2 an.

Befehl: XOR

Opcode: Binär: 1011_2, hexadezimal: $\mathrm{B_h}$, dezimal: 11

Operand: Gibt die Datenadresse an, D00-D15

Beschreibung: Der Operand gibt die Datenadresse an. Der Wert, der in dieser Adresse gespeichert ist, wird Bit für Bit mit dem Akkumulator durch ein XOR verbunden. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.

Beispiel: 10110110_2, ACC=00110011_2, D06=00000110_2

Der Befehl ist XOR, der Operand ist 6, folglich verbindet der Computer den Wert 00000110_2, der in D06 gespeichert ist, mit dem Akkumulator, 00110011_2, durch ein XOR. 00000110_2*xor*00110011_2=00110101_2. Der Akkumulator zeigt somit 00110101_2 an.

Befehl: LOAD

Opcode: Binär: 1100_2, hexadezimal: $\mathrm{C_h}$, dezimal: 12

Operand: Gibt die Datenadresse an, D00-D15

Beschreibung: Der Befehl LOAD lädt eine Zahl aus dem Speicher (Memory) in den Akkumulator. Der Wert des Operanden gibt die Datenadresse an, aus welchem die Zahl zu lesen ist.

Beispiel: 11000110_2, D06=00110011_2

Der Befehl ist LOAD (1100_2) und der Operand ist 6 (0110_2), folglich wird die Zahl 00110011_2, die an der Datenadresse 6 (D06) gespeichert ist, in den Akkumulator geladen.

Befehl:	NOOP
Opcode:	Binär: 1101_2, hexadezimal: D_h, dezimal: 13
Operand:	Kein Einfluss
Beschreibung:	Der Befehl NOOP macht, dass das Programm automatisch zum nächsten Programmschritt springt. Sonst beinhaltet dieser Befehl keine besondere Funktion (**NO OP**eration).

Befehl:	ADD
Opcode:	Binär: 1110_2, hexadezimal: E_h, dezimal: 14
Operand:	Gibt die Datenadresse an, D00-D15
Beschreibung:	Der Befehl ADD addiert einen Wert zu dem Akkumulator. Der Operand gibt die Datenadresse, aus welcher der Wert genommen wird, an. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.
Beispiel:	11100110_2, ACC=00000010_2, D06=00000001_2 Der Befehl ist ADD, der Operand 6. D06 ist 00000001_2 (1) und der Akkumulator ist 00000010_2 (2), also rechnet der Computer 1+2. Somit steht am Schluss 000000011_2 (3) im Akkumulator.

Befehl:	SUB
Opcode:	Binär: 1111_2, hexadezimal: F_h, dezimal: 15
Operand:	Gibt die Datenadresse an, D00-D15
Beschreibung:	Der Befehl SUB zieht einen Wert vom Akkumulator ab. Der Operand gibt die Datenadresse an, aus welcher der Wert genommen wird. Das Resultat ersetzt den vorgängigen Wert des Akkumulators.
Beispiel:	11110110_2, ACC=000000011_2, D06=00000010_2 Der Befehl ist SUB, der Operand ist 6. Somit wird der Wert aus D 6 benötigt, also 00000010_2 (2). Nun nimmt der Computer den ACC und zieht davon 2 ab. 00000011_2-00000010_2=00000001_2. Somit zeigt der Akkumulator am Schluss 00000001_2 (1) an.

3.4.3 Codierung der Schaltsignale

Auf den vorherigen Seiten wurden alle 16 Befehle von MINICOMP so beschrieben, wie sie ein MINICOMP-Programmierer kennen muss.

Um MINICOMP bauen zu können, müssen die Befehle jedoch noch viel genauer beschrieben werden. Sie werden dazu in noch kleinere Einheiten (Mikrobefehle) eingeteilt, wobei jedem Mikrotakt eines Befehls ein Mikrobefehl zugeordnet ist. Ein Mikrobefehl besteht aus der Liste aller Kontrollsignale, die während diesem Mikrotakt aktiviert werden müssen.

Jedem Kontrollsignal ist ein Bit in einem der beiden EPROMs A oder B des Signalgenerators zugeordnet, d.h. ein Wert 2^i. Diese Werte sind in den folgenden Tabellen aufgelistet. Um die Lesbarkeit der Mikrobefehle zu verbessern, kommen darin auch zusammensetzte Signale vor. Diese sind in der rechten Spalte ersichtlich. So bedeutet z.B. das zusammengesetzte Signal PCIN (A=24), das gleichzeitig PCINNZ (A=8) und PCINZ (A=16) aktiv sind.

Signalname:	A	B
ACCIN *	1	0
ACCOUT	2	0
ACCRES	4	0
PCINNZ *	8	0
PCINZ *	16	0
ALU0	32	0
ALU1	64	0
ALU2	128	0

Signalname:	A	B
HALT	0	1
PCOUT	0	2
PCINC	0	4
IRIN *	0	8
IROUT	0	16
MW *	0	32
MR	0	64
MARIN *	0	128

Signalname:	A	B
PCIN *	24	0
ALUADD	0	0
ALUSUB	32	0
ALUNOT	128	0
ALUAND	160	0
ALUOR	192	0
ALUXOR	224	0

Die mit * bezeichneten Kontrollsignale sind *Flankensignale*. Diese dürfen erst dann von 0 auf 1 gewechselt werden, wenn alle andern benötigten Kontrollsignale und Businhalte einen stabilen Wert haben. Deshalb geht jedem Mikrobefehl mit Flankensignal ein Vorbereitungsschritt voran.

Die folgende Tabelle enthält für jeden Befehl die entsprechenden Mikrobefehle. Um das Verständnis zu erleichtern, ist jeder Mikrobefehl kommentiert. Rechts vom Kommentar erscheint in der Tabelle die entsprechende Mikroadresse (μAdr) sowie der zugehörige Inhalt der zwei EPROMS des Signalgenerators (A und B).

Befehl	Takt	Schaltsignale	Kommentar	μAdr	A	B
GOTO	1	IROUT	Operand → Datenbus	1	0	16
	2	IROUT PCIN MARIN	Adresse → PC und MAR	2	24	144
	3	MR	Speicherinhalt → Datenbus	3	0	64
	4	MR IRIN	nächste Instruktion → IR	4	0	72
IFZ	1	PCINC	PC-Inkrement vorbereiten	17	0	4
	2	PCINC PCIN	PC inkrementieren	18	24	4
	3	IROUT	Operand → Datenbus	19	0	16
	4	PCINZ IROUT	falls ACC=0: Adresse → PC	20	16	16
	5	PCOUT	PC → Datenbus	21	0	2
	6	PCOUT MARIN	PC → MAR	22	0	130
	7	MR	Speicherinhalt → Datenbus	23	0	64
	8	MR IRIN	nächste Instruktion → IR	24	0	72
IFNZ	1	PCINC	PC-Inkrement vorbereiten	33	0	4
	2	PCINC PCIN	PC inkrementieren	34	24	4
	3	IROUT	Operand → Datenbus	35	0	16
	4	PCINNZ IROUT	falls ACC≠0: Adresse → PC	36	8	16
	5	PCOUT	PC → Datenbus	37	0	2

Befehl	Takt	Schaltsignale	Kommentar	μAdr	A	B
	6	PCOUT MARIN	PC → MAR	38	0	130
	7	MR	Speicherinhalt → Datenbus	39	0	64
	8	MR IRIN	nächste Instruktion → IR	40	0	72
NOT	1	PCINC ACCRES ALUNOT	PC-Inkrement vorbereiten ACC+ALU vorbereiten für NOT	49	132	4
	2	PCINC PCIN ACCRES ACCIN ALUNOT	PC inkrementieren ACC → NOT(ACC)	50	157	4
	3	PCOUT	PC → Datenbus	51	0	2
	4	PCOUT MARIN	PC → MAR	52	0	130
	5	MR	Speicherinhalt → Datenbus	53	0	64
	6	MR IRIN	nächste Instruktion → IR	54	0	72
LOADI	1	PCINC IROUT	PC-Inkrement vorbereiten Direkter Operand → Datenbus	65	0	20
	2	PCINC PCIN IROUT ACCIN	PC inkrementieren Direkter Operand → ACC	66	25	20
	3	PCOUT	PC → Datenbus	67	0	2
	4	PCOUT MARIN	PC → MAR	68	0	130
	5	MR	Speicherinhalt → Datenbus	69	0	64
	6	MR IRIN	nächste Instruktion → IR	70	0	72
HALT	1	HALT	Anhalten bis WEITER=1	81	0	1
	2	PCINC	PC-Inkrement vorbereiten	82	0	4
	3	PCINC PCIN	PC inkrementieren	83	24	4
	4	PCOUT	PC → Datenbus	84	0	2
	5	PCOUT MARIN	PC → MAR	85	0	130
	6	MR	Speicherinhalt → Datenbus	86	0	64
	7	MR IRIN	nächste Instruktion → IR	87	0	72
ADDI	1	PCINC IROUT ACCRES ALUADD	PC-Inkrement vorbereiten Direkter Operand → Datenbus ACC+ALU vorbereiten f.Addition	97	4	20
	2	PCINC PCIN IROUT ACCIN ACCRES ALUADD	PC inkrementieren ACC → ACC+Operand	98	29	20
	3	PCOUT	PC → Datenbus	99	0	2
	4	PCOUT MARIN	PC → MAR	100	0	130
	5	MR	Speicherinhalt → Datenbus	101	0	64
	6	MR IRIN	nächste Instruktion → IR	102	0	72
SUBI	1	PCINC IROUT ACCRES ALUSUB	PC-Inkrement vorbereiten Direkter Operand → Datenbus ACC+ALU vorbereiten f.Subtrakt.	113	36	20
	2	PCINC PCIN IROUT ACCIN ACCRES ALUSUB	PC inkrementieren ACC → ACC-Operand	114	61	20
	3	PCOUT	PC → Datenbus	115	0	2
	4	PCOUT MARIN	PC → MAR	116	0	130
	5	MR	Speicherinhalt → Datenbus	117	0	64
	6	MR IRIN	nächste Instruktion → IR	118	0	72
STORE	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	129	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	130	24	148
	3	ACCOUT	ACC → Datenbus	131	2	0

Befehl	Takt	Schaltsignale	Kommentar	μAdr	A	B
	4	ACCOUT MW ‘	ACC → Speicher	132	2	32
	5	PCOUT	PC → Datenbus	133	0	2
	6	PCOUT MARIN	PC → MAR	134	0	130
	7	MR	Speicherinhalt → Datenbus	135	0	64
	8	MR IRIN	nächste Instruktion → IR	136	0	72
AND	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	145	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	146	24	148
	3	ACCRES ALUAND MR	ACC+ALU vorbereiten für AND Datenspeicherinhalt → Datenbus	147	164	64
	4	MR ACCRES ALUAND ACCIN	ACC → ACC & Speicherinhalt	148	165	64
	5	PCOUT	PC → Datenbus	149	0	2
	6	PCOUT MARIN	PC → MAR	150	0	130
	7	MR	Speicherinhalt → Datenbus	151	0	64
	8	MR IRIN	nächste Instruktion → IR	152	0	72
OR	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	161	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	162	24	148
	3	ACCRES ALUOR MR	ACC+ALU vorbereiten für OR Datenspeicherinhalt → Datenbus	163	196	64
	4	MR ACCRES ALUOR ACCIN	ACC → ACC # Speicherinhalt	164	197	64
	5	PCOUT	PC → Datenbus	165	0	2
	6	PCOUT MARIN	PC → MAR	166	0	130
	7	MR	Speicherinhalt → Datenbus	167	0	64
	8	MR IRIN	nächste Instruktion → IR	168	0	72
XOR	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	177	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	178	24	148
	3	ACCRES ALUXOR MR	ACC+ALU vorbereiten für XOR Datenspeicherinhalt → Datenbus	179	228	64
	4	MR ACCRES ALUXOR ACCIN	ACC → ACC XOR Speicherinhalt	180	229	64
	5	PCOUT	PC → Datenbus	181	0	2
	6	PCOUT MARIN	PC → MAR	182	0	130
	7	MR	Speicherinhalt → Datenbus	183	0	64
	8	MR IRIN	nächste Instruktion → IR	184	0	72
LOAD	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	193	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	194	24	148
	3	MR	Speicher → Datenbus	195	0	64
	4	MR ACCIN	Speicher → ACC	196	1	64
	5	PCOUT	PC → Datenbus	197	0	2
	6	PCOUT MARIN	PC → MAR	198	0	130
	7	MR	Speicherinhalt → Datenbus	199	0	64
	8	MR IRIN	nächste Instruktion → IR	200	0	72
NOOP	1	PCINC	PC-Inkrement vorbereiten	209	0	4

Befehl	Takt	Schaltsignale	Kommentar	μAdr	A	B
	2	PCINC PCIN	PC inkrementieren	210	24	4
	3	PCOUT	PC → Datenbus	211	0	2
	4	PCOUT MARIN	PC → MAR	212	0	130
	5	MR	Speicherinhalt → Datenbus	213	0	64
	6	MR IRIN	nächste Instruktion → IR	214	0	72
ADD	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	225	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	226	24	148
	3	ACCRES ALUADD MR	ACC+ALU vorbereiten f.Addition Datenspeicherinhalt → Datenbus	227	4	64
	4	MR ACCRES ALUADD ACCIN	ACC → ACC + Speicherinhalt	228	5	64
	5	PCOUT	PC → Datenbus	229	0	2
	6	PCOUT MARIN	PC → MAR	230	0	130
	7	MR	Speicherinhalt → Datenbus	231	0	64
	8	MR IRIN	nächste Instruktion → IR	232	0	72
SUB	1	PCINC IROUT	PC-Inkrement vorbereiten D-Operand → Datenbus	241	0	20
	2	PCINC PCIN IROUT MARIN	PC inkrementieren D-Operand → MAR	242	24	148
	3	ACCRES ALUSUB MR	ACC+ALU vorbereiten f.Subtrakt. Datenspeicherinhalt → Datenbus	243	36	64
	4	MR ACCRES ALUSUB ACCIN	ACC → ACC - Speicherinhalt	244	37	64
	5	PCOUT	PC → Datenbus	245	0	2
	6	PCOUT MARIN	PC → MAR	246	0	130
	7	MR	Speicherinhalt → Datenbus	247	0	64
	8	MR IRIN	nächste Instruktion → IR	248	0	72

3.5 Programmierung

Aus einzelnen Instruktionen kann man nun Programme herstellen, indem man sie sinnvoll hintereinander in den Programmspeicher eingibt.

Auf den folgenden Seiten zeige ich anhand von ein paar Beispielen, wie man MINICOMP programmieren kann. Zuerst werden die Programme kurz beschrieben, danach die Belegung des Speichers (Instruktionen, Konstanten und Variablen) in einer Tabelle aufgezeigt. Für den Inhalt des Lesespeichers ist auch die Schalterstellung angegeben, um die Eingabe des Programms zu erleichtern. Rechts davon sind der entsprechende Speicherwert als Hexadezimalzahl, die Instruktion bestehend aus Opcode und Operand sowie eine kurze Beschreibung zur Funktion dieses Speicherworts.

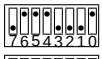
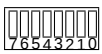
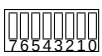
Nach den Programmbeispielen habe ich eine leere Programmiervorlage angefügt, damit ein interessierter Leser auch MINICOMP-Programme selbst entwickeln kann. Es würde mich natürlich freuen, wenn ich Rückmeldungen über solch selbstgeschriebene Programme erhalten würde!

3.5.1 Dekrementierschleife

Ein einfaches Programm, das vom an der Adresse D00 gespeicherten Wert bis nach Null rückwärts zählt.

Titel: **Schlaufe**

Dieses Programm macht so lange eine Schlaufe, bis der Akkumulator den Wert 0 ergibt. Es zählt immer den Akkumulator minus 1.

P00:		C0	`LOAD D00`	`; ACC ← D00`
P01:		71	`SUBI 1`	`; ACC ← ACC-1`
P02:		14	`IFZ P04`	`; Wenn ACC=0 ist, nach Programmadresse P04 springen`
P03:		01	`GOTO P01`	`; Zurück zur Programmadresse P01`
P04:		50	`HALT`	`; das Programm hält an, hier ist das Ende der Schlaufe`
P05:		00	`GOTO P00`	`; Nach Drücken der Weiter-Taste von vorne beginnen`
P06				
P07				
P08				
P09				
P10				
P11				
P12				
P13				
P14				
P15				
D00:				`; Anfangswert hier eingeben`

3.5.2 Bitzähler

Dieses Programm testet jedes Bit des in D07 eingegeben Wertes und zählt die Anzahl eingeschaltener Bits.

Titel: **Bitzähler**

Dieses Programm zählt die eingeschalteten Bits des Werts D07 und zeigt das Endresultat in D08 an, sobald die HALT-LED leuchtet

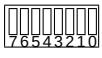
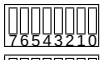
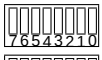
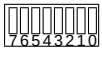
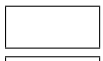
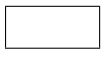

```
P00:  40  LOADI 0     ; ACC ← 0
P01:  88  STORE D08   ; D08 ← ACC, Resultat = 0
P02:  41  LOADI 1     ; ACC ← 1
P03:  89  STORE D09   ; D09 ← ACC, Bitmask = 1
P04:  97  AND D07     ; ACC ← ACC & D07 (höchtens ein ACC-Bit = 1)
P05:  19  IFZ P09     ; Falls ACC=0 zum Programmschritt 9 springen
P06:  C8  LOAD D08    ; ACC ← D08, Resultat in ACC laden
P07:  61  ADDI 1      ; ACC + 1, Resultat inkrementieren

P08:  88  STORE D08   ; D08 ← ACC, Resultat speichern
P09:  C9  LOAD D09    ; ACC ← D09, Bitmask in ACC laden
P10:  E9  ADD D09     ; ACC ← ACC + D09, Bitmask 1 Bit nach links schieben
P11:  89  STORE D09   ; D09 ← ACC, Bitmask speichern
P12:  24  IFNZ P04    ; Falls ACC≠0 zum Programmschritt 4 springen
P13:  50  HALT        ; anhalten bis man WEITER drückt
P14:  00  GOTO P00    ; Nach Drücken der Weiter-Taste von vorne beginnen
P15
```

```
D00
D01
D02
D03
D04
D05
D06
D07:                  ; Wert von welchem die Bits gezählt werden

D08:                  ; Resultat
D09:                  ; Bitmask
D10
D11
```

3.5.3 Quadratzahlen

Dieses Programm berechnet die Quadratzahlen 1, 4, 9, 16, ... 225. Es beruht auf der folgenden Rekursionsformel: $(n+1)^2 = n^2 + 2n + 1$ Nach jedem Wert hält das Programm an und wartet bis die Weiter-Taste gedrückt wird.

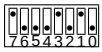
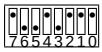
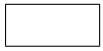
Titel: **Berechnen von Quadratzahlen**

P00:		40	`LOADI 0`	`; ACC ← 0`
P01:		8A	`STORE D10`	`; D10 ← ACC Quadratzahl 0*0=0`
P02:		41	`LOADI 1`	`; ACC ← 1`
P03:		8B	`STORE D11`	`; D11 ← ACC Differenz 1*1-0*0`
P04:		CA	`LOAD D10`	`; ACC ← D10 n*n`
P05:		EB	`ADD D11`	`; ACC ← ACC+D11 n*n + (2*n+1) = (n+1)*(n+1)`
P06:		8A	`STORE D10`	`; D10 ← ACC (n+1)*(n+1) speichern`
P07:		50	`HALT`	`; Pause - Quadratzahl anzeigen (ohne Halt: IFZ P00)`
P08:		CB	`LOAD D11`	`; ACC ← D11 2*n+1`
P09:		62	`ADDI 2`	`; ACC ← ACC+2 (2*n+1)+2 = 2*(n+1)+1`
P10:		8B	`STORE D11`	`; D11 ← ACC 2(n+1)+1 speichern`
P11:		04	`GOTO P04`	`; Zurück und nächste Quadratzahl berechen`
P12				
P13				
P14				
P15				
D00				
D01				
D02				
D03				
D04				
D05				
D06				
D07				
D08				
D09				
D10:				`; Quadratzahl Q = n*n`
D11:				`; Differenz zwischen Quadratzahlen D = 2*n+1`

3.5.4 Multiplikation

Dieses Programm mulitpliziert zwei Zahlen, welche in D06 und D07 eingegeben werden. Durch sukzessive Verdoppelung werden der Summand $A * 2^n$ und die Bitmaske 2^n berechnet. Mit einer AND Operation wird getestet, ob Bit n von B den Wert 1 hat. Wenn ja, wird der Summand $A * 2^n$ zum Resultat addiert.

Titel: **Multiplikation von zwei Zahlen**

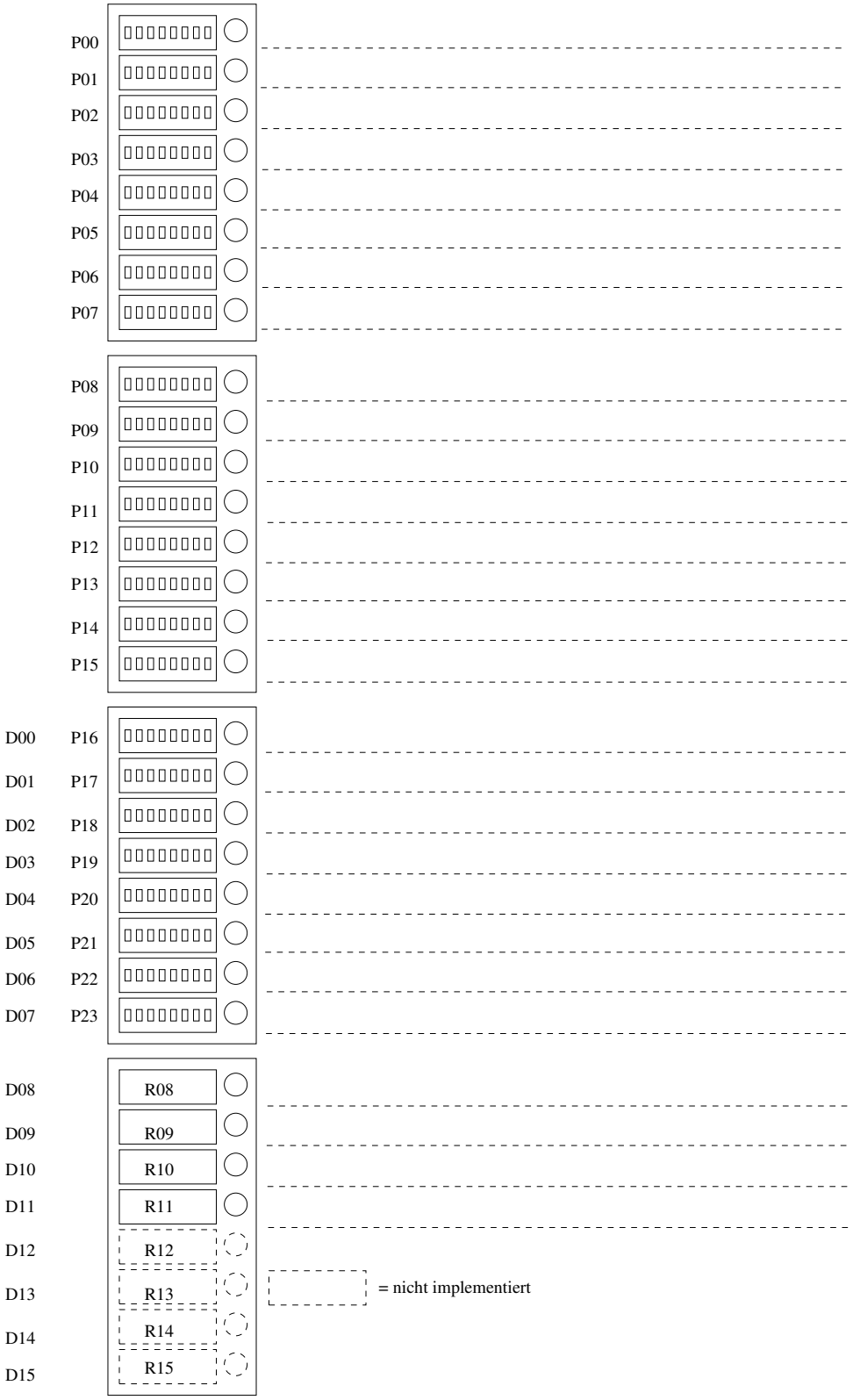
P00:		41	`LOADI 1`	`; ACC ← 1`
P01:		8B	`STORE D11`	`; M ← ACC Bitmaske initialisieren`
P02:		40	`LOADI 0`	`; ACC ← 0`
P03:		8A	`STORE D10`	`; R ← ACC Resultat initialisieren`
P04:		C6	`LOAD D06`	`; ACC ← A`
P05:		88	`STORE D08`	`; S ← ACC Summand beginnt mit A`
P06:		CB	`LOAD D11`	`; ACC ← M`
P07:		97	`AND D07`	`; ACC ← ACC&M n-tes Bit maskieren`
P08:		1C	`IFZ P12`	`; Test, wenn Bit 0 → überspringen`
P09:		CA	`LOAD D10`	`; ACC ← R`
P10:		E8	`ADD D08`	`; ACC ← ACC + S Summand addieren`
P11:		8A	`STORE D10`	`; R ← ACC Resultat speichern`
P12:		C8	`LOAD D08`	`; ACC ← S`
P13:		E8	`ADD D08`	`; ACC ← ACC + S Summand verdoppeln`
P14:		88	`STORE D08`	`; S ← ACC und speichern`
P15:		CB	`LOAD D11`	`; ACC ← M`
P16:		EB	`ADD D11`	`; ACC ← ACC+M Bitmaske nach links schieben`
P17:		8B	`STORE D11`	`; M ← ACC und speichern`
P18:		27	`IFNZ P07`	`; Zurück und nächstes Bit bearbeiten`
P19:		CA	`LOAD D10`	`; Resultat in Akkumulator laden (Anzeige)`
P20:		50	`HALT`	`; Anhalten (A un B können jetzt geändert werden)`
P21:		00	`GOTO P00`	`; Zurück zum Anfang und neue Multiplikation berechnen`
D06:				`; Multiplikator A`
D07:				`; Multiplikand B`
D08:				`; Summand S`
D09				
D10:				`; Resultat R = A*B`
D11:				`; Maske M`

3.5.5 Programmiervorlage

Leeres Formular für eigene Programme:

Vorlage für Programm: _____

Autor: _____ Datum: _____



4 Persönliche Reflexionen und Schlussfolgerungen

Nun, da ich fast fertig bin mit meiner Maturaarbeit und das Produkt vor mir habe, bin ich froh, mein Ziel erreicht zu haben. Nun weiss ich, wie ein Minimalcomputer aufgebaut ist, aus welchen Teilen er besteht und warum er überhaupt funktionieren kann. Am meisten freut es mich aber, dass ich einen vollständigen Minimalcomputer bauen konnte.

Während meiner Maturaarbeit habe ich viel gelernt, denn zu Beginn war alles Neuland für mich. So musste ich zuerst die ganzen theoretischen und elektronischen Grundlagen lernen. Aber auch für den Bau musste ich mich mit verschiedenen neuen Techniken vertraut machen und mir viele praktische Fertigkeiten aneignen. So waren z.B. die Wirewrap-Technik, sowie das Herstellen und Bestücken einer Kupferplatine ganz neu für mich. Ich lernte aber auch viele neue Computerprogramme kennen. Am Anfang meiner Maturaarbeit, da hatte ich noch viel länger, um einen Schaltplan mit Xfig zu entwerfen, schon nur, weil ich immer wieder die richtigen Tasten suchen musste, um z.B. ein NOT zu plazieren. Da ich keine teuren Windows Programme kaufen wollte, arbeitete ich zum grössten Teil mit Linux, da dort alle von mir benötigten Programme gratis zu Verfügung stehen.

Meinen Minimalcomputer zu bauen, war sehr zeitaufwändig. Denn auch wenn man eine Platine entworfen, geätzt und bestückt hat, so heisst das noch lange nicht, dass man fertig ist. Danach geht es nämlich ans Testen, ob sie funktioniert. Wenn nicht, was leider oft der Fall ist, muss man die Fehler suchen, was oftmals sehr zeit- und nervenraubend ist. So musste ich lernen, dass es nicht leicht ist, den Aufwand und den Zeitbedarf für eine Arbeit abzuschätzen. Denn oft dachte ich, dass etwas schnell zu machen sei, doch danach benötigte ich viel mehr Zeit als angenommen.

Auch habe ich es nicht immer geschafft, alle Fehler zu finden und zu korrigieren. Zum Beispiel hat MINICOMP das Problem, dass der Mikrotaktzähler nur richtig zählt, wenn seine Ausgänge nicht verbunden sind. Sind sie jedoch verbunden, so überspringt er manchmal einen Takt. Um dieses Problem zu lösen, wendete ich viele Stunden auf – und trotzdem gelang es mir einfach nicht. So musste ich mich schlussendlich dafür entscheiden, dieses Problem mit einem Trick zu umgehen. So fangen die Mikrobefehle erst mit dem Takt 1 an, und nicht schon, wie ich eigentlich wollte, mit dem Takt 0. Zudem musste ich die Takte verdoppeln, so dass ein fehlender Takt keinen Funktionsfehler verursacht. Somit ist MINICOMP zwar nur noch halb so schnell, aber das stört mich nicht, denn er funktioniert!

Wenn ich frustriert nach einem Fehler suchte und die Zeit anfing knapp zu werden, so war ich öfters drauf und dran, aufzugeben und doch nicht den ganzen Minimalcomputer zu bauen. Aber nun, da ich fertig bin und sehe, dass er läuft, so bin ich sehr froh, dass ich durchgehalten habe, denn es macht Spass, dem eigenhändig gebauten Computer zuzusehen, wie er ein Programm abarbeitet!

Literaturverzeichnis

[1] Aho Alfred V., Kernighan Brian W. and Weinberger, 1988: „*The AWK Programming Language*“, Addison-Wesley Verlag

[2] Bredthauer Wilhelm et al., 1996: „*Impulse Physik 1*“, Klett und Balmer Verlag, Zug

[3] CMOS Databook, 1977: „*National Semiconductors Corp.*“, Santa Clara, U.S.A

[4] Engineer’s Notebook - A Handbook of Integrated Circuit Applications, 1980: „*Mims Forest M.*“, Radio Shack, U.S.A

[5] Fricke Klaus, 2001: „*Digitaltechnik*“, Vieweg Verlag, Braunschweig

[6] Lamport Leslie, 1986: „*LaTeX User’s Guide and Reference Manual*“, Addison-Wesley Verlag

[7] National Semiconductor Corp., 1989: „*Programmable Logical Devices - Databook and Design Guide*“, Santa Clara, U.S.A.

[8] Schiffmann Wolfram und Schmitz Robert, 2001: „*Technische Informatik 1*“, Springer Verlag, Berlin

[9] Spiess Heinz und Borel Claire, 1984: „*IFT1224 - Ordinateurs et systèmes (notes de cours)*“, Université de Montréal, Dép. informatique et recherche opérationnelle, Canada

Programmverzeichnis

[10] Bradley John: „*XV - an interactive image manipulation program for X11*“ (Version 3.10a), Home page: `http://www.trilon.com/xv/`

[11] Conitec AG: „*GALEP32 - Programmiersoftware für GALs, EPROMs, ...*“ (Version 1.14), Home page: `http://www.conitec.de`

[12] Lattice Semiconductor Inc.: „*ispLever - Programmable Logic Development Software*“ (Version 1.0.30 Starter Kit), Home page: `http://www.latticesemi.com`

[13] Nau Thomas: „*PCB - an interactibe printed circuit board editor*“ (Version Version 1.7.2), Home page: `http://bach.ece.jhu.edu/h̃aceaton/pcb/`

[14] RedHat Inc.: „*RedHat Linux*“ (Version 7.3), Home page: `http://www.redhat.com`

[15] Sutanthavibul Supoj et al,: „*Xfig - a vector graphics editor for X11*“ (Version 3.2), Home page: `http://www.xfig.org`

Anhang

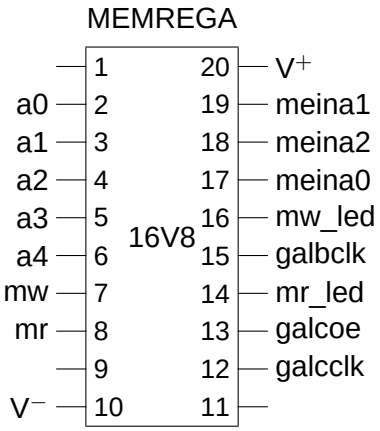
A GAL Programmierung

A.1 Lese-Schreibspeicherregister

Adressdecodierung

Abel-Programm:

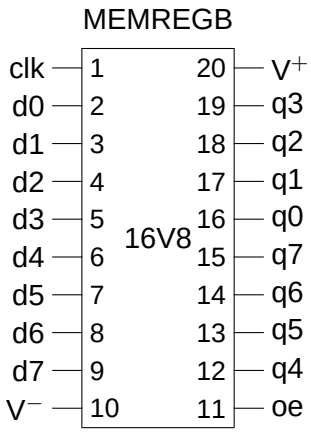
```
MODULE memrega

TITLE 'Read-Write-Memoryregister: Gal A - Adressdekodierung'

"inputs
        a0..a4  pin 2..6;
        mw      pin 7;
        mr      pin 8;
        meina0  pin 17 istype'input';
        meina1  pin 19 istype'input';
        meina2  pin 18 istype'input';
        a = [a4..a0];

"outputs
        galbclk pin 15 istype'com';
        galcoe  pin 13 istype'com';
        galcclk pin 12 istype'com';
        mr_led  pin 14 istype'com';
        mw_led  pin 16 istype'com';

"constants
        meina3 = 1;
        meina4 = 1;

DECLARATIONS
        meinja = [meina2..meina0];
        meina = [1,1,meina2..meina0];
        aok = (meina == a);

EQUATIONS
        galbclk = aok & mw;
        galcoe = aok & mr & !mw;
        mr_led = aok & mr & !mw;
        mw_led = aok & mw;
        galcclk = 0;

END
```



Eingabe und Register

Abel-Programm:

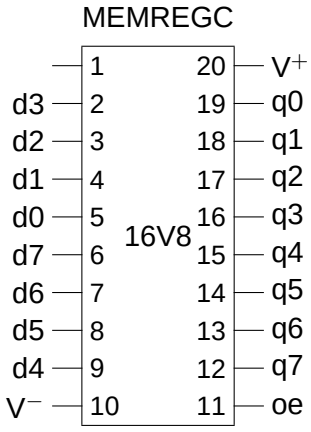
```
MODULE memregb

TITLE 'Read-Write-Memoryregister: Gal B - Datenspeicher'

"inputs
        d0..d7  pin 2..9;
        clk     pin 1;
        oe      pin 11;

"outputs
        q0..q7  pin 16,17,18,19,12,13,14,15 istype'reg';

DECLARATIONS
        d = [d7..d0];
        q = [q7..q0];

EQUATIONS
        q = d;
        q.clk = clk;
        q.oe = !oe;

END
```



Ausgabe auf Datenbus

Abel-Programm:

```
MODULE memregc

TITLE 'Read-Write-Memoryregister: Gal C - Ausgabe auf Datenbus'

"inputs
        d0..d7  pin 5,4,3,2,9,8,7,6;
        "clk    pin 1;
        oe      pin 11;

"outputs
        q0..q7  pin 19..12;

DECLARATIONS
        d = [d7..d0];
        q = [q7..q0];

EQUATIONS
        q = d;
        q.oe = oe;

END
```

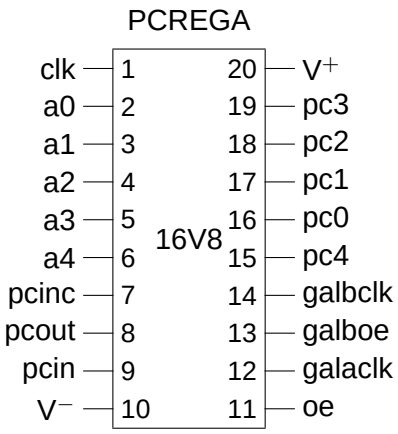


A.2 Programmzähler und Adressregister

Eingabe und Register

Abel-Programm:

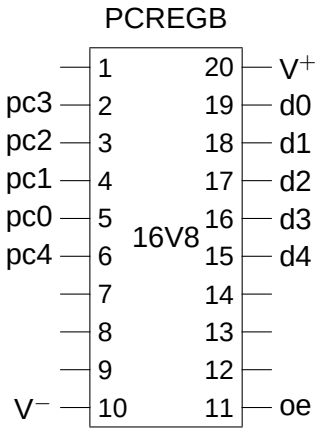
```
MODULE pcrega

TITLE 'PC-Register Gal A - Speicher und Inkrementzähler'

"inputs
        a0..a4  pin 2..6;
        clk     pin 1;
        oe      pin 11;
        pcinc   pin 7;
        pcout   pin 8;
        pcin    pin 9;

"outputs
        pc0..pc4        pin 16,17,18,19,15 istype'reg';
        galboe  pin 13;
        galbclk pin 14;
        galaclk pin 12;

DECLARATIONS
        a = [a4..a0];
        pc = [pc4..pc0];

EQUATIONS
        pc = (!pcinc & a) # (pcinc & (pc + 1));
        pc.clk = clk;
        pc.oe = !oe;
        galboe = !(pcout & !pcin);
        galaclk = pcin;
        galbclk = 0;

END
```



Ausgabe auf Datenbus

Abel-Programm:

```
MODULE pcregb

TITLE 'PC-Register Gal B - Ausgabe auf Datenbus'

"inputs
        pc0..pc4        pin 5,4,3,2,6;
        oe      pin 11;

"outputs
        d0..d4  pin 19..15;

DECLARATiONS
        pc = [pc4..pc0];
        d = [d4..d0];

EQUATIONS
        d = pc;
        d.oe = !oe;

END
```

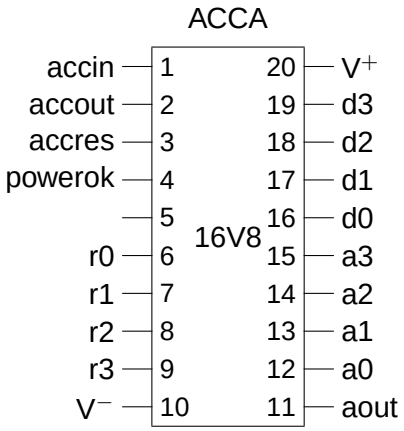


A.3 Akkumulator

Bits 0 bis 3

Abel-Programm:

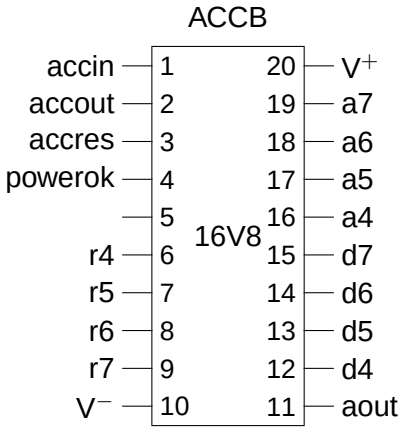
```
MODULE ACC_A

TITLE 'Akkumulator Bits 0 bis 3'

"inputs
        accin   pin 1;
        accout  pin 2;
        accres  pin 3;
        powerok pin 4;
        r0..r3  pin 6..9;
        aout    pin 11;
        r = [r3..r0];

"in/out
        d0..d3  pin 16..19 istype'com';
        d = [d3..d0];

"outputs
        a0..a3  pin 12..15 istype'reg';
        a = [a3..a0];

EQUATIONS
        a = (((accres==1)&r)#((accres==0)&d))&powerok;
        a.clk = accin;
        a.oe = !aout;
        d = a;
        d.oe = accout;
END
```



Bits 4 bis 7

Abel-Programm:

```
MODULE ACC_B

TITLE 'Akkumulator Bits 4 bis 7'

"inputs
        accin   pin 1;
        accout  pin 2;
        accres  pin 3;
        powerok pin 4;
        r4..r7  pin 6..9;
        aout    pin 11;
        r = [r7..r4];

"in/out
        d4..d7  pin 12..15 istype'com';
        d = [d7..d4];

"outputs
        a4..a7  pin 16..19 istype'reg';
        a = [a7..a4];

EQUATIONS
        a = (((accres==1)&r)#((accres==0)&d))&powerok;
        a.clk = accin;
        a.oe = !aout;
        d = a;
        d.oe = accout;

END
```

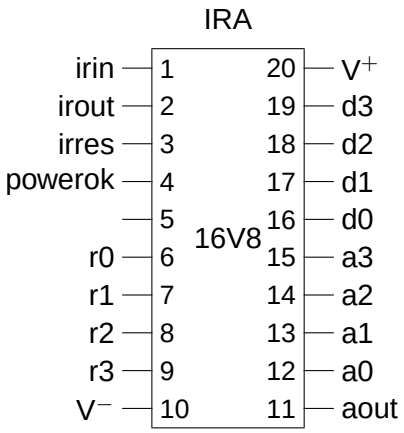


A.4 Instruktionsregister

Bits 0 bis 3

Abel-Programm:

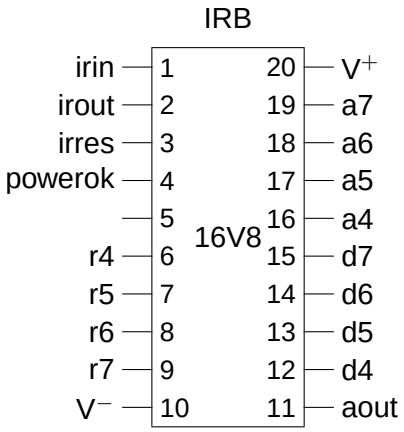
```
MODULE irgala

TITLE 'Instruktionsregister Bits 0 bis 3'

"inputs
        irin    pin 1;
        irout   pin 2;
        irres   pin 3;
        power   pin 4;
        r0..r3  pin 6..9;
        aout    pin 11;
        r = [r3..r0];

"in/out
        d0..d3  pin 16..19 istype'com';
        d = [d3..d0];

"outputs
        a0..a3  pin 12..15 istype'reg';
        a = [a3..a0];

EQUATIONS
        a = ((irres==1)&r&power)#((irres==0)&d&power);
        a.clk = irin;
        a.oe = !aout;
        d = a;
        d.oe = irout;

END
```



Bits 4 bis 7

Abel-Programm:

```
MODULE IR_B

TITLE 'Instruktionsregister Bits 4 bis 7'

"inputs
        irin    pin 1;
        irout   pin 2;
        irres   pin 3;
        powerok pin 4;
        r4..r7  pin 6..9;
        aout    pin 11;
        r = [r7..r4];

"in/out
        d4..d7  pin 12..15 istype'com';
        d = [d7..d4];

"outputs
        a4..a7  pin 16..19 istype'reg';
        a = [a7..a4];

EQUATIONS
        a = (((irres==1)&r)#((irres==0)&d))&powerok;
        a.clk = irin;
        a.oe = !aout;
        d4 = a7;
        d5 = 0;
        d6 = 0;
        d7 = 0;
        d.oe = irout;

END
```

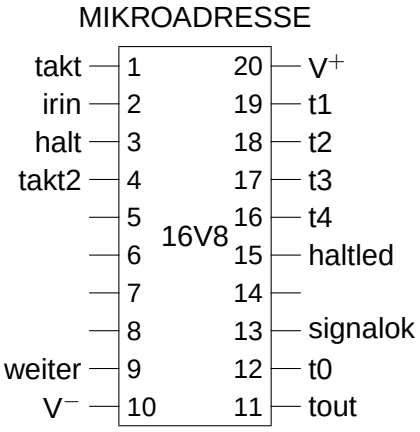


A.5 Schalteinheit

Mikrotaktzähler

Abel-Programm:

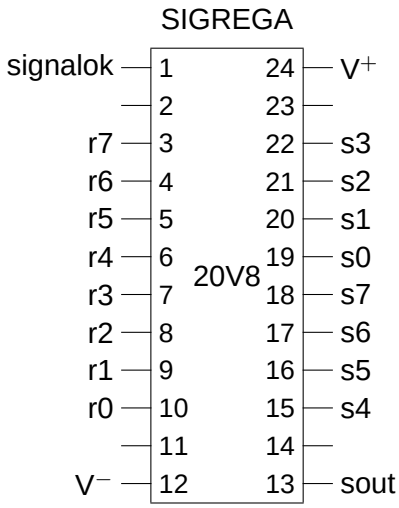
```
MODULE mikroaddresse

TITLE 'Taktzaehler fuer Mikrocode und Signallogik'

"inputs
        takt    pin 1;
        irin pin 2;
        weiter  pin 9;
        halt    pin 3;
        tout    pin 11;
        takt2   pin 4;

"outputs
        haltled pin 15 istype'com';
        signalok        pin 13 istype'com';
        t0..t4  pin 12,19..16 istype'reg';
        t = [t4..t0];

EQUATIONS
        t.oe = !tout;
        t.clk = takt;
        t = (!irin & !(halt & !weiter))&(t+1) # ((halt & !weiter)&t);
        haltled = halt & !weiter;
        signalok = !takt2;

END
```



Kontrollsignalpuffer A

Abel-Programm:

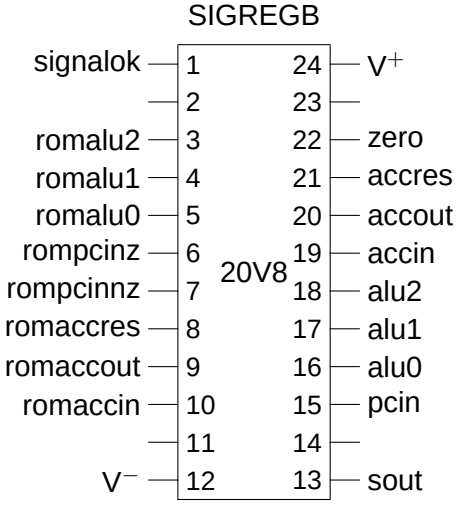
```
MODULE sigrega

TITLE 'Puffer für Kontrollsignale IR,MAR,PC,MEM'

"inputs
        signalok pin 1;
        r0..r7   pin 10..3;
        sout     pin 13;
        r = [r7..r0];

"outputs
        s0..s7   pin 19..22,15..18 istype'reg';
        s = [s7..s0];

EQUATIONS
        s = r;
        s.clk = signalok;
        s.oe = !sout;

END
```



Kontrollsignalpuffer B

Abel-Programm:

```
MODULE sigregb

TITLE 'Puffer für Kontrollsignale ACC und ALU'

"inputs
        signalok  pin 1;
        rompcinnz pin 7;
        romaccres pin 8;
        romaccout pin 9;
        romaccin  pin 10;
        romalu2   pin 3;
        romalu1   pin 4;
        romalu0   pin 5;
        rompcinz  pin 6;
        sout      pin 13;
        zero      pin 22;

"outputs
        accres  pin 21 istype 'reg';
        accout  pin 20 istype 'reg';
        accin   pin 19 istype 'reg';
        alu2    pin 18 istype 'reg';
        alu1    pin 17 istype 'reg';
        alu0    pin 16 istype 'reg';
        pcin    pin 15 istype 'reg';
        s = [accres, accout, accin, alu2, alu1, alu0, pcin];

EQUATIONS
        accres = romaccres;
        accout = romaccout;
        accin = romaccin;
        alu2 = romalu2;
        alu1 = romalu1;
        alu0 = romalu0;
        pcin = (zero & rompcinz) # (!zero & rompcinnz);
        s.clk = signalok;
        s.oe = !sout;

END
```

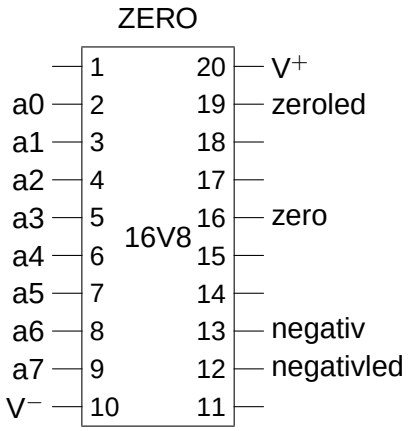


A.6 Stecker-GAL-Stecker

Null-Detektor

Abel-Programm:

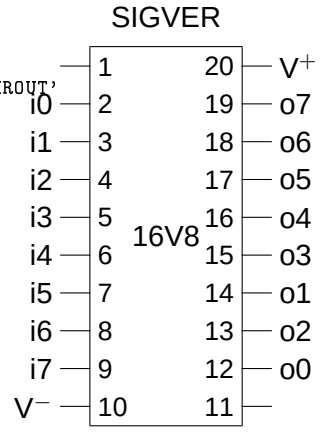
```
MODULE zero

TITLE 'Nullwerttest für Akkumulator'

"inputs
        a0..a7  pin 2..9;

"outputs
        zero    pin 18 istype 'com';
        zeroled pin 19 istype 'com';
        negativled pin 12 istype 'com';
        negativ pin 13 istype 'com';

DECLARATiONS
        a = [a7..a0];

EQUATIONS
        zero = (a == 0);
        zeroled = !(a == 0);
        negativ = a7;
        negativled = a7;

END
```



Kontrollsignalverteiler

Abel-Programm:

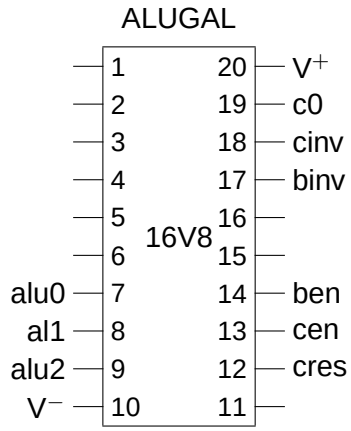
```
MODULE puffer

TITLE 'Puffer für Kontrollsignale MR, MW, MARIN, PCIN, PCOUT, PCINC, IRIN, IROUT'

"inputs
        i0..i7  pin 2..9;
        i = [i7..i0];

"outputs
        o0..o7  pin 19,18,17,16,15,13,14,12 istype'com';
        o = [o7..o0];

EQUATIONS
        o = i;

END
```



ALU Kontrollsignalumwandler

Abel-Programm:

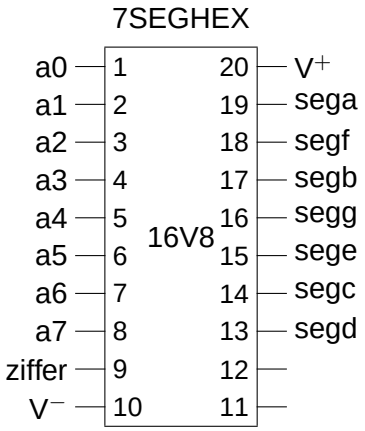
```
MODULE alugal

TITLE 'Gal für das Generieren der Kontrollsignale der ALU'

"inputs
        alu0    pin 7;
        alu1    pin 8;
        alu2    pin 9;

"outputs
        binv    pin 17 istype 'com';
        cinv    pin 18 istype 'com';
        c0      pin 19 istype 'com';
        cres    pin 12 istype 'com';
        cen     pin 13 istype 'com';
        ben     pin 14 istype 'com';

DECLARATIONS
        alu = [alu2..alu0];
        aluadd = (alu == 0);
        alusub = (alu == 1);
        aluinc = (alu == 2);
        aludec = (alu == 3);
        alunot = (alu == 4);
        aluand = (alu == 5);
        aluor = (alu == 6);
        aluxor = (alu == 7);

EQUATIONS
        ben = aluadd # alusub # aluand # aluor # aluxor;
        cen = aluadd # alusub # aluinc # aludec;
        cres = aluand # aluor;
        c0 = alusub # aluinc # aluor;
        cinv = aluadd # alusub # aluinc # aludec # aluand # aluxor # alunot;
        binv = aluadd # aluinc # aluand # aluor # aluxor;

END
```



A.7 Zweistellige Hexadezimalanzeige

Abel-Programm:

```
MODULE hexanzeige

TITLE 'ZWEISTELLIGE SIEBENSEGMENT HEXANZEIGE'

"input
        a0..a7 PIN 1..8;
        !ziffer PIN 9;
        a = [a7..a0];
        z0 = [a3..a0];
        z1 = [a7..a4];

"output
        sega PIN 19 istype'com';
        segb PIN 17 istype'com';
        segc PIN 14 istype'com';
        segd PIN 13 istype'com';
        sege PIN 15 istype'com';
        segf PIN 18 istype'com';
        segg PIN 16 istype'com';

"konstanten
        an = 0;
        ab = 1;
        x = .X.;

truth_table
       ([ziffer, z0, z1] -> [sega, segb, segc, segd, sege, segf, segg])
        [0,0,x]  -> [an, an, an, an, an, an, ab];
        [0,1,x]  -> [ab, an, an, ab, ab, ab, ab];
        [0,2,x]  -> [an, an, ab, an, an, ab, an];
        [0,3,x]  -> [an, an, an, an, ab, ab, an];
        [0,4,x]  -> [ab, an, an, ab, ab, an, an];
        [0,5,x]  -> [an, ab, an, an, ab, an, an];
        [0,6,x]  -> [an, ab, an, an, an, an, an];
        [0,7,x]  -> [an, an, an, ab, ab, ab, ab];
        [0,8,x]  -> [an, an, an, an, an, an, an];
        [0,9,x]  -> [an, an, an, an, ab, an, an];
        [0,10,x] -> [an, an, an, ab, an, an, an]; "A
        [0,11,x] -> [ab, ab, an, an, an, an, an]; "b
        [0,12,x] -> [an, ab, ab, an, an, an, ab]; "C
        [0,13,x] -> [ab, an, an, an, an, ab, an]; "d
        [0,14,x] -> [an, ab, ab, an, an, an, an]; "E
        [0,15,x] -> [an, ab, ab, ab, an, an, an]; "F

        [1,x,0]  -> [an, an, an, an, an, an, ab];
        [1,x,1]  -> [ab, an, an, ab, ab, ab, ab];
        [1,x,2]  -> [an, an, ab, an, an, ab, an];
        [1,x,3]  -> [an, an, an, an, ab, ab, an];
        [1,x,4]  -> [ab, an, an, ab, ab, an, an];
        [1,x,5]  -> [an, ab, an, an, ab, an, an];
        [1,x,6]  -> [an, ab, an, an, an, an, an];
        [1,x,7]  -> [an, an, an, ab, ab, ab, ab];
        [1,x,8]  -> [an, an, an, an, an, an, an];
        [1,x,9]  -> [an, an, an, an, ab, an, an];
        [1,x,10] -> [an, an, an, ab, an, an, an]; "A
        [1,x,11] -> [ab, ab, an, an, an, an, an]; "b
        [1,x,12] -> [an, ab, ab, an, an, an, ab]; "C
        [1,x,13] -> [ab, an, an, an, an, ab, an]; "d
        [1,x,14] -> [an, ab, ab, an, an, an, an]; "E
        [1,x,15] -> [an, ab, ab, ab, an, an, an]; "F

END
```



B EPROM Programmierung

B.1 Schaltbare dreistellige 7-Segment-Anzeige

Programmaufruf: `awk -f anzeige.awk`

Ausgabedatei: `anzeige.bin`

AWK-Programm `anzeige.awk`:

```
function digit(val,dig,form,str)
{
    str = sprintf(format[form],fname[form]=="sdec"?(val>127?val-256:val):val)
    if(substr(str,1,2)=="-1") str=" N"substr(str,3,2)
    return substr(str,5-dig,1);
}

function reverse(val)
{
    res = 0;
    if(val>=128){res+=1;val-=128}
    if(val>=64){res+=2;val-=64}
    if(val>=32){res+=4;val-=32}
    if(val>=16){res+=8;val-=16}
    if(val>=8){res+=16;val-=8}
    if(val>=4){res+=32;val-=4}
    if(val>=2){res+=64;val-=2}
    if(val>=1){res+=128;val-=1}
    return res
}

function byte(val,dig,form,sg)
{
    sg = seg[digit(val,dig,form)]
    b = 0;
    if(sg~/a/)b+=bit["a"]
    if(sg~/b/)b+=bit["b"]
    if(sg~/c/)b+=bit["c"]
    if(sg~/d/)b+=bit["d"]
    if(sg~/e/)b+=bit["e"]
    if(sg~/f/)b+=bit["f"]
    if(sg~/g/)b+=bit["g"]
    if(sg~/p/)b+=bit["p"]
    return b
}

function complement(val)
{
    return -val-1;
}


BEGIN {
    format["0"]="  %02x"
    format["1"]=" %03o"
    format["2"]="%4d"
    format["3"]="%4u"
    fname["0"]="hex"
```

```
    fname["1"]="oct"
    fname["2"]="sdec"
    fname["3"]="udec"

    bit["a"]=128
    bit["b"]=32
    bit["c"]=4
    bit["d"]=2
    bit["e"]=8
    bit["f"]=64
    bit["g"]=16
    bit["p"]=1

    seg[" "] = ""
    seg["-"] = "g"
    seg["N"] = "bcg"
    seg["0"] = "abcdef"
    seg["1"] = "bc"
    seg["2"] = "abdeg"
    seg["3"] = "abcdg"
    seg["4"] = "bcfg"
    seg["5"] = "acdfg"
    seg["6"] = "acdefg"
    seg["7"] = "abc"
    seg["8"] = "abcdefg"
    seg["9"] = "abcdfg"
    seg["a"] = "abcefg"
    seg["b"] = "cdefg"
    seg["c"] = "adef"
    seg["d"] = "bcdeg"
    seg["e"] = "adefg"
    seg["f"] = "aefg"

    ind=0;
    for(f=0 ; f<4 ; f++)
        for(d=1; d<=4 ; d++)
            for(i=0 ; i<256 ; i++){
                rom[ind] = complement(byte(reverse(i),d,f))
                printf("%c",rom[ind++])>"anzeige.bin"
            }

    exit
}
```

B.2 Codierung der Schaltsignale

Programmaufruf: `awk -f mikrorom.awk mikrocode.tab`

Ausgabedatei: `mikrorom.bin`

Mikrocode Eingabedatei `mikrocode.tab`:

```
DA   0   ;  Mikroadressenverschiebung
RS   128 ;  ROM Grösse

S     1 ACCIN
S     2 ACCOUT
S     4 ACCRES
```

```
S     8 PCINNZ
S    16 PCINZ
S    32 ALU0
S    64 ALU1
S   128 ALU2

S   256 HALT
S   512 PCOUT
S  1024 PCINC
S  2048 IRIN
S  4096 IROUT
S  8192 MW
S 16384 MR
S 32768 MARIN

; zusammengesetzte signale
S     0 PCIN PCINZ PCINNZ
S     0 ALUADD
S     0 ALUSUB           ALU0
S     0 ALUINC      ALU1
S     0 ALUDEC      ALU1 ALU0
S     0 ALUNOT ALU2
S     0 ALUAND ALU2      ALU0
S     0 ALUOR  ALU2 ALU1
S     0 ALUXOR ALU2 ALU1 ALU0

I 0 GOTO
T 1 IROUT                       ;Operand -> Datenbus
T 2 IROUT PCIN MARIN            ;Adresse -> PC und MAR
T 3 MR                          ;Speicherinhalt -> Datenbus
T 4 MR IRIN                     ;nächste Instruktion -> IR

I 1 IFZ
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 2 PCINC PCIN                  ;PC inkrementieren
T 3 IROUT                       ;Operand -> Datenbus
T 4 PCINZ IROUT                 ;falls ACC=0: Adresse -> PC
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 2 IFNZ
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 2 PCINC PCIN                  ;PC inkrementieren
T 3 IROUT                       ;Operand -> Datenbus
T 4 PCINNZ IROUT                ;falls ACC!=0: Adresse -> PC
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 3 NOT
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 ACCRES ALUNOT               ;ACC+ALU vorbereiten für NOT
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 ACCRES ACCIN ALUNOT         ;ACC -> NOT(ACC)
T 3 PCOUT                       ;PC -> Datenbus
```

```
T 4 PCOUT MARIN                ;PC -> MAR
T 5 MR                         ;Speicherinhalt -> Datenbus
T 6 MR IRIN                    ;nächste Instruktion -> IR

I 4 LOADI
T 1 PCINC                      ;PC-Inkrement vorbereiten
T 1 IROUT                      ;Direkter Operand -> Datenbus
T 2 PCINC PCIN                 ;PC inkrementieren
T 2 IROUT ACCIN                ;Direkter Operand -> ACC
T 3 PCOUT                      ;PC -> Datenbus
T 4 PCOUT MARIN                ;PC -> MAR
T 5 MR                         ;Speicherinhalt -> Datenbus
T 6 MR IRIN                    ;nächste Instruktion -> IR

I 5 HALT
T 1 HALT                       ;Anhalten bis WEITER=1
T 2 PCINC                      ;PC-Inkrement vorbereiten
T 3 PCINC PCIN                 ;PC inkrementieren
T 4 PCOUT                      ;PC -> Datenbus
T 5 PCOUT MARIN                ;PC -> MAR
T 6 MR                         ;Speicherinhalt -> Datenbus
T 7 MR IRIN                    ;nächste Instruktion -> IR

I 6 ADDI
T 1 PCINC                      ;PC-Inkrement vorbereiten
T 1 IROUT                      ;Direkter Operand -> Datenbus
T 1 ACCRES ALUADD              ;ACC+ALU vorbereiten f.Addition
T 2 PCINC PCIN IROUT           ;PC inkrementieren
T 2 ACCIN ACCRES ALUADD        ;ACC -> ACC+Operand
T 3 PCOUT                      ;PC -> Datenbus
T 4 PCOUT MARIN                ;PC -> MAR
T 5 MR                         ;Speicherinhalt -> Datenbus
T 6 MR IRIN                    ;nächste Instruktion -> IR

I 7 SUBI
T 1 PCINC                      ;PC-Inkrement vorbereiten
T 1 IROUT                      ;Direkter Operand -> Datenbus
T 1 ACCRES ALUSUB              ;ACC+ALU vorbereiten f.Subtrakt.
T 2 PCINC PCIN IROUT           ;PC inkrementieren
T 2 ACCIN ACCRES ALUSUB        ;ACC -> ACC-Operand
T 3 PCOUT                      ;PC -> Datenbus
T 4 PCOUT MARIN                ;PC -> MAR
T 5 MR                         ;Speicherinhalt -> Datenbus
T 6 MR IRIN                    ;nächste Instruktion -> IR

I 8 STORE
T 1 PCINC                      ;PC-Inkrement vorbereiten
T 1 IROUT                      ;D-Operand -> Datenbus
T 2 PCINC PCIN                 ;PC inkrementieren
T 2 IROUT MARIN                ;D-Operand -> MAR
T 3 ACCOUT                     ;ACC -> Datenbus
T 4 ACCOUT MW   ‘              ;ACC -> Speicher
T 5 PCOUT                      ;PC -> Datenbus
T 6 PCOUT MARIN                ;PC -> MAR
T 7 MR                         ;Speicherinhalt -> Datenbus
T 8 MR IRIN                    ;nächste Instruktion -> IR

I 9 AND
```

```
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 ACCRES ALUAND               ;ACC+ALU vorbereiten für AND
T 3 MR                          ;Datenspeicherinhalt -> Datenbus
T 4 MR ACCRES ALUAND ACCIN      ;ACC -> ACC & Speicherinhalt
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 10 OR
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 ACCRES ALUOR                ;ACC+ALU vorbereiten für OR
T 3 MR                          ;Datenspeicherinhalt -> Datenbus
T 4 MR ACCRES ALUOR ACCIN       ;ACC -> ACC # Speicherinhalt
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 11 XOR
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 ACCRES ALUXOR               ;ACC+ALU vorbereiten für XOR
T 3 MR                          ;Datenspeicherinhalt -> Datenbus
T 4 MR ACCRES ALUXOR ACCIN      ;ACC -> ACC XOR Speicherinhalt
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 12 LOAD
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 MR                          ;Speicher -> Datenbus
T 4 MR ACCIN                    ;Speicher -> ACC
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 13 NOOP
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 2 PCINC PCIN                  ;PC inkrementieren
T 3 PCOUT                       ;PC -> Datenbus
T 4 PCOUT MARIN                 ;PC -> MAR
T 5 MR                          ;Speicherinhalt -> Datenbus
T 6 MR IRIN                     ;nächste Instruktion -> IR
```

```
I 14 ADD
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 ACCRES ALUADD               ;ACC+ALU vorbereiten f.Addition
T 3 MR                          ;Datenspeicherinhalt -> Datenbus
T 4 MR ACCRES ALUADD ACCIN      ;ACC -> ACC + Speicherinhalt
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR

I 15 SUB
T 1 PCINC                       ;PC-Inkrement vorbereiten
T 1 IROUT                       ;D-Operand -> Datenbus
T 2 PCINC PCIN                  ;PC inkrementieren
T 2 IROUT MARIN                 ;D-Operand -> MAR
T 3 ACCRES ALUSUB               ;ACC+ALU vorbereiten f.Subtrakt.
T 3 MR                          ;Datenspeicherinhalt -> Datenbus
T 4 MR ACCRES ALUSUB ACCIN      ;ACC -> ACC - Speicherinhalt
T 5 PCOUT                       ;PC -> Datenbus
T 6 PCOUT MARIN                 ;PC -> MAR
T 7 MR                          ;Speicherinhalt -> Datenbus
T 8 MR IRIN                     ;nächste Instruktion -> IR
```

AWK-Programm `mikrorom.awk`:

```
BEGIN   {
          instruktion = -999999
          romsize = 256
        }

$1=="DA" { # signal: adressverschiebung ($2)
           da = $2
          }

$1=="RS" { # rom size ($2)
           romsize = $2
         }

$1=="S" { # signal: wert ($2), name ($3) und andere signale
          signal[$3] = $2
          for(i=4; i<=NF; i++) signal[$3] += signal[$i]
        }

$1=="I" { # befehl: opcode ($2) und name ($3)
          opcode[$3]=$2
          befehl[$2]=$3
          instruktion=$2
        }

$1=="T" { # mikroinstruktion:  takt($2) und signale ($3,...)
          a = 16*instruktion+$2 + da
          split($0,teile,";")
          $0 = teile[1];
          for(i=3; i<=NF; i++) mprog[a] += signal[$i]
        }
```

```
END     {
           for(a=0 ; a<256 ; a++){
                print "mprog["a"]="mprog[a];
                rom[a] = mprog[a]%256;
                rom[a+256] = int(mprog[a]/256);
           }
           for(n=0 ; n<romsize/4 ; n++)
               for(a=0 ; a<512 ; a++)
                    printf("%c",rom[a]) > "mikrorom.bin";
        }
```