2021

Entwurf einer 8-Bit CPU

WIE MAN LOGIKGATTER PROGRAMMIERBAR MACHT

EINE BESONDERE LERNLEISTUNG

VON ALEXANDER DITINGER

13SE1
RABANUS-MAURUS-GYMNASIUM

Inhalt

Einleitung

Vorwort

In dieser BLL dokumentiere ich meinen Entwurf einer 8-Bit CPU, die dabei in einem Programm zum Simulieren logischer Schaltkreise realisiert wird. Dabei wird die Funktionsweise einer CPU erläutert, indem sie in einzelne Komponenten zerlegt wird, die nacheinander aufgebaut und vernetzt werden. Ziel ist es, am Ende ein funktionsfähige CPU zu haben, die mit verschiedenen Maschinenbefehlen programmierbar ist.

Was ist eine CPU?

CPU ist ein Akronym für das Englische Wort „Central Processing Unit“, zu Deutsch zentrale Verarbeitungseinheit. Oftmals wird sie auch Hauptprozessor oder nur Prozessor genannt. In digitalen Elektronikgeräten hat eine CPU die zentrale Aufgabe, ein bestimmtes Programm auszuführen und somit Daten zu verarbeiten. Als Rechen- und Steuerwerk führt sie arithmetische- und logische Operationen durch und steuert weitere Komponenten des Gerätes. Eingesetzt werden CPUs heutzutage entweder wie in einem PC als Einzelkomponente oder in sogenannten „eingebetteten Systemen“, also Computern, die auf ganz bestimmte Aufgaben zugeschnitten sind. Dazu zählen beispielsweise einzelne Steuerkomponenten von Fahrzeugen oder Produktionsanlagen, Smartphones und vieles mehr (Malvino, 1999, S. 213f).

Während CPUs in vielen eingebetteten Systemen nur auf eine spezielle Aufgabe zugeschnitten sind und daher nur über ein beschränktes Befehlsset verfügen, so sind CPUs von eingebetteten Systemen wie Smartphones, aber auch alleinstehende CPUs von PCs oder Mikrocontroller darauf ausgelegt, frei programmierbar zu sein, damit sie die verschiedensten Programme beziehungsweise Apps für und ausführend können. Aufgrund ihres weiten Einsatzspektrums existieren daher auch diverse unterschiedliche Architekturen, das heißt Bauarten von CPUs.

CPU-Architektur

Die Prozessorarchitektur einer CPU bestimmt deren inneren Aufbau und somit auch direkt ihre Spezifikationen. Während die Prozessorarchitektur tatsächlich den Aufbau der elektronischen beziehungsweise logischen Schaltungen der CPU bezeichnet, reduziert die Befehlsarchitektur die Spezifikationen einer CPU auf die für ihren Einsatz und ihre Programmierung relevanten Informationen.

Die in meiner BLL entworfene CPU ist eine frei programmierbare CPU. Daher erläutert die BLL nur den Aufbau für universell einsetzbare und programmierbare CPUs, wie sie beispielsweise in Computern eingesetzt werden. Der folgende Abschnitt beschäftigt sich aus diesem Grund kurz mit der Geschichte der Architektur universeller CPUs.

Eine wichtige Grundlage für moderne CPU-Architekturen war die „Von-Neumann-Architektur“ von 1945. Der Intel 8008, der erste vollwertige 8-Bit Prozessor und basierte auf dieser Architektur. Von der Von-Neumann-Architektur ging es über die von Intel entwickelte x86-Architektur hin zur heutigen x64-Architektur, die von den gängigen Herstellern für Computer CPUs, Intel und AMD, in einer eigenen Implementierung verwendet wird (Wüst, 2003, S. 14; Riley/Brailsford, 2018).

CPUs der x64-Architektur zeichnen sich durch eine Datenbreite von 64 Bit aus. Sie operieren nach einem festen Schema, wie die Befehle eines Programmes abgearbeitet werden: „Fetch, Decode, Execute“ (zu Deutsch: „Laden, Dekodieren, Ausführen“). Ein Befehl wird zunächst aus dem Hauptspeicher in die Kontrolleinheit der CPU geladen, dort dekodiert und zuletzt ausgeführt. Ein

weiterer, wichtiger Teil der x64-Architektur, der auf die Von-Neumann-Architektur zurückzuführen ist, ist die gemeinsame Unterbringung von Befehlen und Daten des Programmes in einem zentralen Hauptspeicher (Silc, 1999, S. VIIf).

Die 8 Bit CPU dieser BLL

Die CPU dieser BLL hat eine Datenbreite von 8 Bit. Sie wird in einem Programm zur Simulation von logischen Schaltungen gebaut (Logisim) auf der Ebene von Logikgattern. Das Projekt wurde durch eine Videoreihe von Ben Eater inspiriert, jedoch wurde die dort aufgebaute CPU mit einigen Änderungen und Erweiterungen versehen. Die zugrundeliegende Prozessorarchitektur ist die SAP-Architektur (SAP = „Simple As Possible"), die dafür gedacht ist, den Aufbau von heutigen CPUs leicht zu verstehen.

Das Schaubild zur SAP-Architektur (Anhang 1.6.4, Anhang 1.6.5) zeigt den groben Aufbau der CPU. Die einzelnen Komponenten im Schaubild werden im Folgenden auf Logikebene erläutert. Normalerweise befinden sich RAM sowie eine Anzeige nicht in der CPU selbst, der RAM ist jedoch essentiell zur Programmierung und Funktionsfähigkeit der CPU, eine Ausgabeanzeige wird zur Anschaulichkeit auch integriert.

Hinweis: Im Folgenden wird zwar jede Komponente und jedes Bauteil aus Logikgattern aufgebaut, jedoch wird nicht jedes Bauteil mithilfe der Logikgattern erläutert. Zum Teil werden nur die Funktionsweise und die Aufgaben einzelner Bauteile erläutert. Zudem müssen manche Bauteile zur Implementierung im Simulator ein wenig abgeändert werden oder zusammengefasst werden. Beispielsweise werden parallel verlaufende Datenkabel zu einem Kabel mit einer größeren Bitbreite zusammengefasst. Die Bilder zu den Schaltungen sind entweder direkt beigefügt oder im Anhang zu finden.

Die Logikebene (Grundlegende Logikgatter und Schreibweise)

Von der Logikebene spricht man, wenn man eine digitale Schaltung aus Logikgattern konzipiert. Ein Logikgatter ist ein elektronisches Bauteil, das eine logische Operation auf eine oder mehrere Eingaben anwendet und ausgibt. Auf der Logikebene vernachlässigt man daher der Einfachheit halber die elektrotechnischen Aspekte der Logikgatter, das heißt deren Realisierung durch Transistoren und weitere elektrotechnische Phänomene, die im Umgang mit realen Logikgattern auftreten können. Man betrachtet nur die Datenströme, die die Zustände „0" (niedriges Potential, „falsch") und „1" (hohes Potential, „wahr") annehmen können. Die vier Grundlogikgatter sind:

[1] Nicht-Gatter (NOT)

Boolesche Funktion: $\overline{A}$

Symbol:

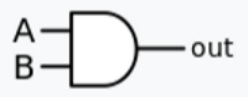
Wahrheitstabelle:

A	out
0	1
1	0

[2] Und-Gatter (AND)

Boolesche Funktion: $A \wedge B$

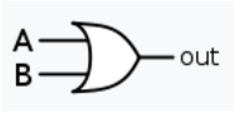
Symbol:



Wahrheitstabelle:

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

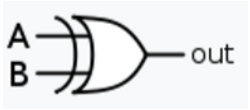
[3] Oder-Gatter (OR)
Boolesche Funktion: $A \vee B$
Symbol:



Wahrheitstabelle:

A	B	out
0	0	0
0	1	1
1	0	1
1	1	1

[4] XOR-Gatter (Exklusives Oder)
Boolesche Funktion: $A \underline{\vee} B$
Symbol:



Wahrheitstabelle:

A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

(Malvino, 1999, S. 19-23 und S. 32-43)

Bau der CPU

Register und Bussystem

Ein wichtiger Aspekt einer CPU ist die Fähigkeit, Daten in der CPU zwischenzuspeichern und zwischen den einzelnen Komponenten innerhalb der CPU austauschen zu können. Ein sogenanntes Register übernimmt die Aufgabe des Zwischenspeicherns von Daten für einzelne Komponenten der CPU. Es speichert einen bestimmten Wert, der per Steuersignal geladen beziehungsweise ausgegeben werden kann. Ein Steuersignal ist generell ein einfaches 1-Bit Signal zum Steuern einer Komponente der CPU (Malvino, 1999, S. 106).

Über das Bussystem der CPU sind deren einzelne Komponenten, die Daten untereinander austauschen müssen, verbunden. Es kann immer nur eine Komponente der CPU Daten an den Bus ausgeben, während mehrere Komponenten diese Daten auslesen können. Der Grund, warum immer nur eine Komponente Daten an den Bus ausgeben kann, ist folgender: Jeder logische Zustand, auch eine „0“, ist ein elektrischer Zustand. Geben alle Komponenten bis auf die Komponente, die tatsächlich Daten an den Bus ausgeben will, eine „0“ aus, so führt dies trotzdem dazu, dass Strom zwischen der tatsächlich ausgebenden Komponente und den auf „0“ gesetzten Komponenten fließt. Jegliche Komponenten, die nicht vom Bus lesen, müssen folglich vollständig vom Bus entkoppelt werden, während eine andere Komponente an den Bus ausgibt.

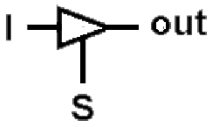
Tristate Puffer

Um dieses Problem zu lösen, nutzt man einen sogenannten Tristate. Ein Tristate ist ein elektrisches Bauteil, der über einen Dateneingang, einen Datenausgang und ein Steuersignal verfügt. Sofern das Steuersignal auf „1“ gesetzt ist, wird der Zustand des Dateneingangs direkt über den Ausgang ausgegeben. Liegt am Steuersignal eine „0“ an, so wird der Datenausgang, unabhängig vom Dateneingang, auf einen undefinierten, hochohmigen Zustand gesetzt, der somit den Dateneingang vom Datenausgang entkoppelt. Dieser undefinierte, hochohmige Zustand wird im Folgenden als **Z** bezeichnet, in Logisim als **x**.

Wahrheitstabelle:

I	S	out
I_0	0	**Z**
I_0	1	I_0

Symbol:



Ein Tristate wird an den Datenausgang aller Komponenten geschaltet, die Daten an den Bus ausgeben. Um der Komponente Schreibzugriff auf den Bus zu gewähren, wird das Steuersignal entsprechend auf „1“ gesetzt (Malvino 1999, S. 121ff).

Grundspeicherbaustein SR-Flip-Flop

Um ein Register, das Daten speichert, zu realisieren, benötigt man zunächst einen Grundbaustein, der einen Datenbit über eine gewisse Zeit speichern kann. Dazu verwendet man ein sogenanntes SR-Latch. Es verfügt über drei Eingänge: Set, $Reset$ und CLK. Über das Set Signal kann der Wert des SR-Latches auf 1 gesetzt werden. Über $Reset$ wird er auf „0“ zurückgesetzt. Da für die CPU zeitliches Timing ein wichtiger Aspekt ist, damit die Komponenten immer synchronisiert arbeiten, verfügt es auch über ein Taktgebereingang, genannt CLK (engl.: clock, dt.: Taktgeber). Set und Reset ändern den Zustand des SR-Latches nur, wenn am CLK Eingang „1“ anliegt. Hauptbestandteil des SR-Latches sind

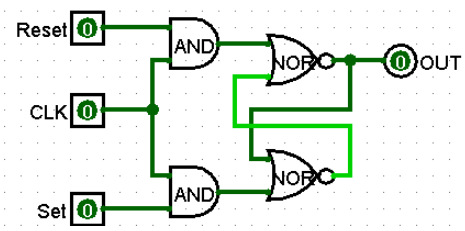


Abb. 1: SR-Latch mit CLK Signal

die zwei NOR-Gatter, normale Oder-Gatter mit einem invertierten Ausgang. Deren Ausgänge sind jeweils mit einem Eingang des anderen NOR-Gatters verbunden, somit ergeben beide Gatter zusammen eine bistabile Schaltung, die einen bestimmten Wert hält, der geändert werden kann, wenn der zweite Eingange eines der beiden NOR-Gatter gesetzt wird. Wird der Eingang des oberen NOR-Gatters gesetzt, wird das Latch auf „0“ zurückgesetzt, wird der Eingang des unteren gesetzt, wird das Latch auf „1“ gesetzt. Damit jedoch die Set und Reset Signale nur wirken, wenn das CLK Signal „1“ ist, sind Set und $Reset$ über zwei Und-Gatter mit dem CLK Signal verbunden.

Als SR-Flip-Flop bezeichnet man ein SR-Latch, das nur dann seinen Wert ändern kann, wenn das Taktsignal (CLK) von „0“ auf „1“ wechselt, während das Latch immer seinen Wert ändern kann, wenn das Taktsignal „1“ ist. Der Übergang des Taktsignals zu einem anderen Wert wird als Flanke bezeichnet. Findet dieser Übergang von „0“ nach „1“ statt, so spricht man von der steigenden Flanke, andernfalls von der fallenden Flanke. Daher wird ein SR-Flip-Flop im Gegensatz zum pegelgesteuerten Latch als taktflankengesteuert bezeichnet (Quelle [2], o.V., 2011, S. 4ff).

Das fertige Register

Mit dem SR-Latch als Grundlage kann nun ein Register als Baustein zusammengesetzt werden. Ein Register soll einen Datenwert D_1 während der steigenden Flanke speichern, wenn zusätzlich noch ein $Load$ Signal gesetzt ist, damit nicht bei jeder Flanke ein zufälliger Wert des Busses gespeichert wird, sondern nur, wenn das Register dazu aufgefordert wird mit dem Steuersignal $Load$. Dass der Wert von D_1 nur während der steigenden Flanke geladen werden soll, ist insofern wichtig, als dass sich der Inhalt des Busses der CPU nach der steigenden Flanke während der Taktgeber auf „1“ ist schon wieder ändern kann. Dieser neue Wert soll jedoch nicht mehr in das Register geladen werden soll, da dieser den ursprünglich vom Bus geladenen Wert überschreiben würde. Außerdem soll es auch eine Möglichkeit geben, das Register asynchron, also ungeachtet des CLK Signals, auf „0“ zurücksetzten zu können über ein CLR Signal.

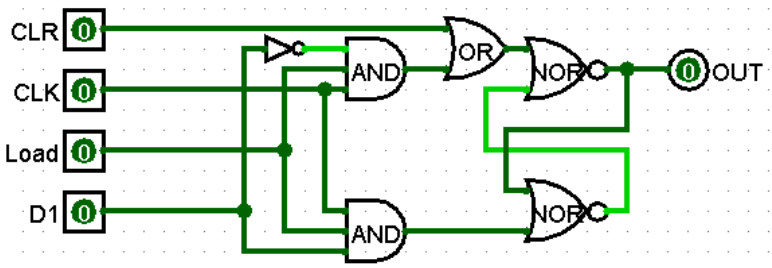


Abb. 2: 1-Bit Register ohne Rising Edge Detektor

Abb. 2 zeigt ein 1-Bit Register bestehend aus einem SR-Latch, daher noch nicht taktflanken-gesteuert. Dazu muss ein „Rising Edge Detektor“ (engl.: Rising Edge = steigende Taktflanke) vorge-schaltet werden, also eine Schaltung, die dafür sorgt, dass nur während der steigenden Flanke ein kurzes Taktsignal an das Register gegeben wird. So speichert das Register den anliegenden Datenwert nur bei der steigenden Flanke des CLK Signals (Malvino, 1999, S. 96f).

Das CLR Signal ist über ein Oder-Gatter direkt mit dem $Reset$ Signal des SR-Latches verbunden. D_1 ersetzt das Set Signal an dem einen Und-Gatter, das $Reset$ Signal am anderen Gatter wird durch das negierte D_1 Signal ersetzt. Ist D_1 „0“, so ist $Reset$ aktiv und das SR-Latch wird auf zurückgesetzt; ist D_1 „1“, so ist Set aktiv und das SR-Latch wird auf „1“ gesetzt. Zu den Und-Gattern, die das Set und $Reset$ Signal mit dem CLK Signal synchronisieren, kommt ein weiterer Eingang dazu, sodass Set und $Reset$ nur aktiv sind, wenn auch das $Load$ Signal gesetzt ist (Quelle [2], o.V., 2011, S. 7).

Ein 8-Bit Register besteht nun aus 8 nebeneinanderliegenden 1-Bit Registern. So ergeben sich die Dateneingänge D_1 bis D_8 und die Ausgänge O_1 bis O_8. CLR, CLK und $Load$ aller 8 Register werden jeweils verbunden, damit ein Steuersignal alle 8 Register anspricht. Hinter die Ausgänge aller Register wird jeweils ein Tristate geschaltet, damit das Ausgabesignal der Register deaktiviert werden kann.

Die Steuersignale der einzelnen Tristates werden zu einem vierten Steuersignal des Registers zusammengefasst, dem $Enable$ Signal, das, wenn es auf „1“ gesetzt ist, den Inhalt des Registers ausgibt. Die über das $Enable$ Signal gesteuerte Ausgabe OUT kann somit direkt an den Bus angeschlossen werden. Ein Register kann jedoch gleichzeitig mit dem Bus und zusätzlich mit einem anderen Bauteil, das durchgehend den Inhalt des Registers lesen muss, verbunden sein. Daher wird eine weitere Ausgabe, genannt $DIR\ OUT$, die vor den Tristates entnommen wird, dem fertigen 8 Bit Register hinzugefügt.

Außerdem wird dem gemeinsamen CLK Signal des Registers nun der Rising Edge Detektor vorgeschaltet. Sobald CLK von 0 nach 1 übergeht, gibt dieser einen kurzen Puls als CLK Signal aus. Normalerweise realisiert man dieses Verhalten über eine elektronische Schaltung, in Logisim wird eine Schaltung aus Logik Gattern verwendet, die nicht näher erläutert wird. Wichtig ist, dass beim Übergang von 0 nach 1 ein kurzer Signalimpuls ausgegeben wird ((Quelle [2], o.V., 2011, S. 1f). Das fertige 8 Bit Register besitzt nun die Dateneingänge D_1 bis D_8 (zusammengefasst als D), ein CLK Signal, ein $Load$ Signal, das dafür sorgt, dass die Eingabe im Register gespeichert wird, ein CLR Signal, welches das Register asynchron auf „0“ zurücksetzt, ein $Enable$ Signal, das dafür sorgt, dass die Datenausgänge O_1bis O_8 (zusammengefasst als OUT) an den Bus ausgegeben werden sowie einen direkten Datenausgang, der unabhängig von $Enable$ immer den Wert des Registers ausgibt (Anhang 1.1.1).

Das Bussystem: Grundgerüst

Um nun das Grundgerüst für die CPU zu legen, werden die wichtigsten Register der CPU mit dem Datenbus verbunden. Der Bus hat eine Datenbreite von 8 Bit, genau wie die Register und anderen Komponenten. Die ersten vier Register, die an den Bus angebunden werden, sind:

- **A-Register**: Speichert den ersten Operanden der Recheneinheit der CPU, sowie das Ergebnis. Außerdem kann es beliebig gesetzt und ausgelesen werden.
- **Temporäres Register (TMP-Register)**: Speichert den zweiten Operanden der Recheneinheit der CPU. Es wird nur als Zwischenspeicher von bestimmten Werten während der Ausführung von Befehlen genutzt.
- **B- und C-Register**: Zwei weitere Register zum Speichern und Auslesen beliebiger Werte. Sie dienen als schnell erreichbarer Speicher für zwei weitere Datenworte.

(Malvino, 1999, S. 175)

Die Dateneingänge D der Register sowie die Ausgänge OUT werden direkt an den Datenbus angeschlossen. Die CLK Signale werden zusammengeführt und an den gemeinsamen Taktgeber der CPU angeschlossen. Im Logiksimulator ist der Taktgeber ein einfacher Baustein, der ein Rechtecksignal mit variabler Frequenz erzeugt. Die CLR Signale der CPU auch zusammengeführt und an einen Taster angeschlossen, der als Reset-Taster dient, um die CPU in ihren Ausgangsstatus zurückzusetzen. Die tatsächlichen Steuersignale der Register ($Load$ und $Enable$ für alle Register), werden zunächst nur separat zu einer Seite geleitet (Anhang 1.1.2).

Die Arithmetisch Logische Einheit

Die nächste Komponente, die der CPU hinzugefügt wird, ist ihre Recheneinheit. Die sogenannte Arithmetisch Logische Einheit, kurz ALU (engl.: Arithmetic Logic Unit) übernimmt die arithmetischen Rechenaufgaben der CPU, das heißt Addition und Subtraktion, sowie logische Operationen, also die Verknüpfung ihrer Eingabewerte durch eine logische Funktion (Malvino, 1999, S. 79). Allgemein verfügt die ALU über zwei Eingänge, die Operanden A und B, sowie einen Ausgang OUT, der das Ergebnis der jeweiligen Operation ausgibt. Des Weiteren verfügt sie über einen Eingang O, über den die auszuführende Operation bestimmt wird.

Das Addierwerk der ALU

Das Addieren binärer Zahlen ist analog zu der schriftlichen Addition von Dezimalzahlen. Werden beispielsweise 0110 (6) und 0010 (2) addiert, sieht die Rechnung folgendermaßen aus:

$$\begin{array}{r} 0\ 1\ 1\ 0 \\ +\ 0_1 0_1 1\ 0 \\ \hline 1\ 0\ 0\ 0 \end{array}$$

Beginnend bei der rechten Seite ergibt 0 + 0 = 0. 1 + 1 ist in binärer Schreibweise 10, wobei die 0 in das Ergebnis geschrieben wird und die 1 als Übertrag unter die nächste Spalte geschrieben wird. 1 + 0 ergibt 1, da der Übertrag aber gesetzt ist, wird noch einmal 1 addiert, was 10 (2) ergibt. Wieder wird 0 im Ergebnis notiert und der Übertrag auf 1 gesetzt. Die letzte Spalte ist 0 + 0 + 1 aus dem Übertrag, was gesamt 1 ergibt. Aus diesem Beispiel lassen sich erste Anforderungen an einen logischen Schaltkreis ableiten, der zwei Ein-Bit Zahlen addieren kann, wobei wird zunächst der Übertrag als Eingabe ignoriert wird:

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Wahrheitstabelle für die Addition zweier Bits

A und B sind die Summanden, S bezeichnet das Ergebnis der Spalte und C_{out} den weiterzugebenden Übertrag (engl.: carry = Übertrag). Anhand der Tabelle lässt sich erkennen, dass C_{out} und S jeweils die Ergebnisse einfacher logischer Verknüpfungen von A und B sind:

$$C_{out} = A \wedge B\ ; S = A \veebar B$$

Der sich daraus ergebende Schaltkreis (Abb. 3) wird auch Halbaddierer genannt, da er zwar zwei Ein-Bit Zahlen addieren kann, dabei aber einen eventuell eingehenden Übertrag noch nicht berücksichtigt (Malvino, 1999, S. 79ff).

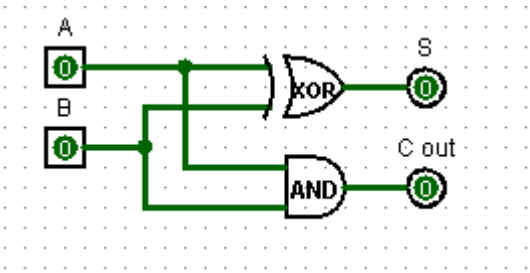


Abb. 3: 1-Bit Halbaddierer

Damit ein eingehender Übertrag berücksichtigt werden kann, muss dieser zur Summe von A und B addiert werden. Dem Halbaddierer wird ein zweiter Halbaddierer hinzugeschaltet, der $(A + B)$ und C_{in} als Eingabewerte erhält und somit die Gesamtsumme berechnet. Sollte bei einer der beiden Additionen ein Übertrag gesetzt werden, so muss auch der Übertrag der gesamten Addition gesetzt werden. Der Gesamtübertrag ist somit die logische Verknüpfung der Überträge beider Halbaddierer durch ein Oder-Gatter.

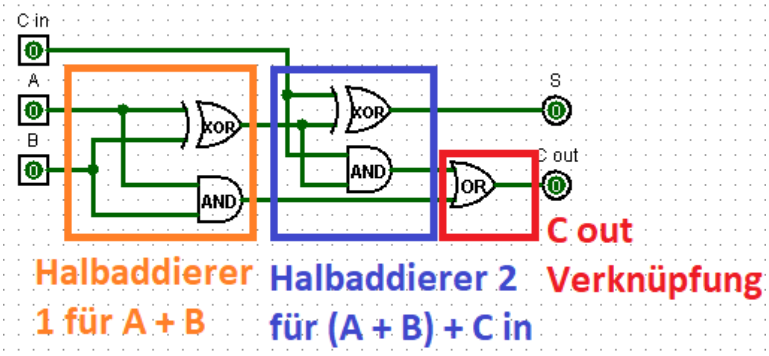


Abb. 4: 1-Bit Volladdierer

Abbildung 4 zeigt oben beschriebenen Schaltkreis für einen Volladdierer. Das Addierwerk der 8-Bit CPU soll aber natürlich 8-Bit Zahlen addieren können. Dazu werden acht 1-Bit Volladdierer parallelgeschaltet und der ausgehende Übertrag wird mit dem eingehenden Übertrag des jeweils nachfolgenden Addierers verbunden (Anhang 1.2.1, Eater, 2015, „Learn how computers add numbers and build a 4 bit adder circuit“).

Subtraktion binärer Zahlen

Der bisher entwickelte Addierer kann auch zur Subtraktion von Binärzahlen verwendet werden, indem man eine negative Zahl addiert. Doch wie werden negative Zahlen in binärer Schreibweise dargestellt? Hierzu wurde das sogenannte Zweierkomplement entwickelt. Mithilfe des Zweierkomplements lassen sich binäre Bitfolgen als negative ganze Zahlen interpretieren und durch die Addition einer im Zweierkomplement dargestellten Zahl lässt sich eine Subtraktion ausführen.

Um eine Zahl im Zweierkomplement darzustellen, darf das höchste Bit nicht von der eigentlichen Zahl verwendet werden, da es angibt, ob eine Zahl im Zweierkomplement dargestellt ist, wenn es auf „1“ gesetzt ist. Um die Zahl „-6“ darzustellen, müssen also mindestens vier Bit verwendet werden, da 6 in Binärschreibweise 110 ist und mindestens ein unbenutztes Bit an vorderster Stelle stehen muss. Für die 8-Bit CPU gehen wir also von der Zahl 00000110 aus, um eine 6 darzustellen. Zunächst werden alle Bits negiert ($\overline{00000110} = 11111001$) und im Anschluss wird 1 addiert ($11111001 + 00000001 = 11111010$). Somit lässt sich durch die Zahl „-6“ durch 11111010 in binärer Schreibweise in der 8-Bit CPU darstellen.

Man unterscheidet bei ganzen Zahlen (engl.: Integer) zwischen Signed Intergers und Unsigned Integers. Wird das höchste Bit genutzt, um eine Zahl im Zweierkomplement darzustellen, so spricht man von Signed Integers, andernfalls von Unsigned Integers. Für die beiden unterschiedlichen Interpretationen von den 8 Bits der CPU ergeben sich unterschiedliche maximale bzw. minimale Integerwerte, die dargestellt werden können. Ein 8-Bit Unsigned Integer stellt Zahlen von 0 bis 255 dar, ein 8 Bit Signed Integer stellt Zahlen von -128 bis 127 dar. Die Art und Weise der Interpretation einer Bitfolge wird immer durch das Programm bestimmt. Nur das Programm hat den Überblick darüber, wie welche Bitfolge im Hauptspeicher interpretiert wird; die CPU selbst führt ungeachtet, wie eine Bitfolge zu interpretieren ist, die Operation auf diese aus.

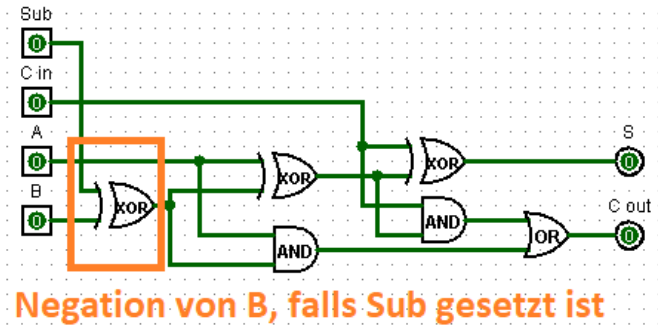


Abb. 5: 1 Bit Volladdierer mit Subtraktionsfunktion

Der aktuelle 8-Bit Addierer kann recht einfach um eine Funktion erweitert werden, die den zweiten Summanden B als Zweierkomplement darstellt und somit die Subtraktionsfunktion $A - B$ darstellt, falls ein Eingabebit Sub gesetzt ist. Zunächst wird eine „Negation auf Befehl“ benötigt, was durch ein XOR-Gatter erreicht wird (Malvino, 1999, S. 41): Ist Sub gesetzt, wird

die Negation der zweiten Eingabe, dem Summanden B, ausgegeben (Abb. 5). Um zu B „1“ zu addieren, wird das C_{in} Bit des niedrigst-wertigen 1-Bit Addierers im 8-Bit Addierer gesetzt, was einer Addition von 00000001 entspricht. C_{in} des ersten 1-Bit Addierers kann mit Sub verbunden werden, damit ein gesetztes Sub direkt das Zweierkomplement von B addiert (Malvino, 1999, S. 83-86).

So ergibt sich aus dem ergänzten Volladdierer die gesamte arithmetische Einheit (Anhang 1.2.1).

Logische Operationen und 1-Bit ALU

Um die logischen Operationen zu integrieren, gehen wir einen kleinen Schritt zurück. Anstatt die gesamte arithmetische Einheit um diese zu ergänzen, werden nur die 1-Bit Volladdierer um die logischen Operationen ergänzt für eine 1-Bit ALU und analog zur arithmetischen Einheit zusammengeschaltet, um die 8-Bit ALU zu bilden.

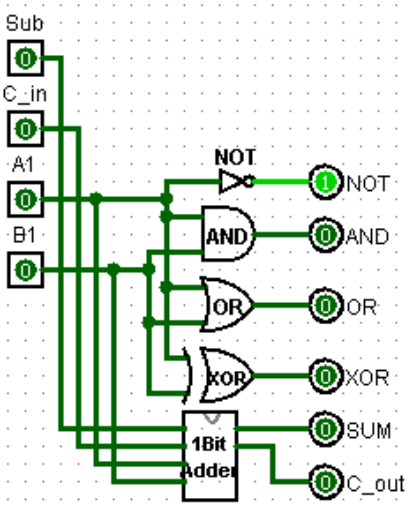


Abb. 6: 1-Bit ALU

Neben den 1-Bit Addierer werden nun die Gatter Und, Oder, Nicht und XOR geschaltet und mit den Operanden A und B verbunden. Für die Nicht-Operation wird nur A negiert. Es ergeben sich fünf verschiedene Ausgänge für die einzelnen Operationen: SUM für Addition und Subtraktion, sowie AND, OR, NOT und XOR.

Die 1-Bit ALU kann nun wie ein Volladdierer parallelgeschaltet werden, indem C_{in} und C_{out} wie bei der arithmetischen Einheit verknüpft werden, und bildet so das Grundgerüst für die 8-Bit ALU. Als nächstes muss festgelegt werden, wann welcher der fünf Ausgaben auf den Bus ausgegeben werden soll. Das ist natürlich abhängig davon, welche Operation die ALU ausführen soll. Dazu wird eine Auswahlschaltung verwendet, ein sogenannter Multiplexer. Er hat eine gewisse Anzahl an verschiedenen Eingängen, aus denen er auswählen kann, und einen Ausgang. Zudem hat er ein Steuereingang, der je nach Anzahl der Eingänge eine unterschiedliche Datenbreite besitzt. Über den Steuereingang gibt man die Nummer des zu wählenden Eingangs in binärer Schreibweise an. Das Signal am entsprechenden Eingang wird nun über den Ausgang ausgegeben. So lässt sich einer der Eingänge auswählen. Um einen solchen Multiplexer für die Operationen der ALU zu entwerfen, benötigt man die komplette Übersicht aller Operationen der ALU:

Operation Nr.	Code Multiplexer	Befehl	Ausgabe von	Beschreibung
0	0000	NOOP	---	Trennt die ALU vom Bus
1	0001	ADD	SUM	$OUT = A + B$
2	0010	SUB	SUM	$OUT = A - B$
3	0011	AND	AND	$OUT = A \wedge B$
4	0100	OR	OR	$OUT = A \vee B$
5	0101	NOT	NOT	$OUT = \overline{A}$
6	0110	XOR	XOR	$OUT = A \veebar B$
7	0111	DEC	SUM	$OUT = A - 1$
8	1000	INC	SUM	$OUT = A + 1$
9	1001	AOUT	SUM	$OUT = A$

Hinweis: Die Operationen 7 bis 9 werden später erläutert (s. Die ALU-Befehle 7 bis 9: Befehle zur Laufzeitoptimierung).

Ein Multiplexer wird hinter jede 1-Bit ALU geschaltet. Die fünf Ausgänge der 1-Bit ALU werden mit den jeweiligen Eingängen des Multiplexers verbunden. Die vier Steuereingänge der einzelnen Multiplexer werden verbunden und bilden die Eingänge O_1 bis O_4, die die Operation bestimmen, die von der ALU auf den Bus ausgegeben wird. Wird als Operation der ALU beispielsweise 0011 gesetzt, so geben die einzelnen Multiplexer der 1-Bit ALUs alle die Und-Verknüpfung als Ausgabe aus. Für den Multiplexer siehe Anhang 1.2.2.

Da auch die ALU an den Bus angeschlossen wird und Daten an diesen ausgibt, muss auch zwischen jedes OUT-Signal der Multiplexer und den Bus ein Tristate geschaltet werden. Dieser Tristate soll nur dann ALU und Bus trennen, wenn NOOP als Operation der ALU ausgewählt wurde. Der Code für die NOOP ist 0000. Sobald eine andere Operation ausgewählt ist, ist mindestens ein dieser Bits auf 1 geschaltet. Somit lässt sich durch eine Oder-Verknüpfung der Eingänge O_1 bis O_4 feststellen, ob die Tristates an den Bus ausgeben sollen oder nicht. Die Steuersignale der einzelnen Tristates können direkt mit der Ausgabe der OR Verknüpfung verbunden werden (Anhang 1.2.3).

Flags: Statusausgaben der ALU

Ein weiterer wichtiger Aspekt der ALU ist nicht nur die Ausführung von arithmetischen und logischen Operationen, sondern auch die Kommunikation mit der Kontrolleinheit der CPU über den aktuellen Status der ALU beziehungsweise über wichtige Informationen über das Ergebnis der letzten Rechenoperation. Solche Informationen können beispielsweise sein, ob das letzte Ergebnis gleich Null war. Diese Informationen können dann von der Kontrolleinheit genutzt werden, um den weiteren Programmablauf zu bestimmen und das Programm „entscheidungsfähig“ zu machen. Solche Informationen der ALU sind Statusausgaben zu der zuletzt ausgeführten Operation und werden auch Flags genannt. Drei wichtige Flags, die in meiner CPU verwendet werden, sind:

- **Zero-Flag:** Ist „1“, wenn das Ergebnis der letzten Operation gleich Null ist.
- **Sign-Flag:** Ist „1“, wenn das Ergebnis der letzten Operation negativ ist, sofern man es im Zweierkomplement interpretiert.
- **Carry-Flag:** Ist „1“, wenn das Ergebnis der letzten Operation den Übertrag des letzten der acht in Reihe geschalteten 1 Bit Addierer hervorruft, das Ergebnis also 255 überschreitet.

(Wüst, 2003, S. 16ff)

Diese drei Flags können genutzt werden, um das Programm zu steuern und Vergleichsoperationen auszuführen. In diesem Abschnitt wird zunächst nur die Logik erläutert, die nötig ist, damit die Flags zu ihren gegebenen Konditionen gesetzt werden.

Die Carry-Flag ist einfach zu realisieren: Sie ist das C_{out}-Signal der letzten 1-Bit ALU. Da dieses vorher ohnehin nicht genutzt wurde, kann es problemlos als Flag Signal verwendet werden.

Die Sign-Flag lässt sich realisieren, indem man sich das höchstwertige Bit der ALU Ausgabe anschaut. Ist dieses gesetzt, so bedeutet es für die Zahl, die von der ALU ausgegeben wird, dass sie negativ ist, wenn man sie im Zweierkomplement interpretiert. Da die Interpretation der Ausgabe immer vom ausgeführten Programm abhängt, ist auch die Aussagekraft der Sign-Flag von der Interpretation durch das Programm abhängig. Soll die Operation der ALU als einfache binäre Zahl interpretiert werden, so kann es keine negative Zahl geben und die Sign-Flag ist trotzdem gesetzt, sobald das Ergebnis der Operation größer als 127 ist (mindestens $10000000_{\text{bin}} = 127_{\text{dec}}$). Nichtsdestotrotz kann die Sign-Flag immer ausgegeben werden und entspricht einfach dem Wert des höchstwertigen Bits der ALU Ausgabe.

Die Zero-Flag soll gesetzt sein, wenn das Ergebnis der letzten Operation der ALU gleich Null ist, also alle einzelnen Bits des Ergebnisses auf „0“ gesetzt sind. Verknüpft man alle dieser Bits mit einem Oder-Gatter, so gibt dieses „1“ aus, sobald eines der Bits aktiv ist, also das Ergebnis größer als 0 ist. Das ist das genaue Gegenteil der Zero-Flag. Negiert man also die Ausgabe dieses Oder-Gatters, so erhält man die Zero-Flag (Anhang 1.2.4).

Da die Flags aber zum Teil von den direkten Ausgaben der Multiplexer abhängen, die jedoch nur aktiv sind, wenn eine bestimmte ALU-Operation ausgewählt ist, sind sie zwischen der ALU-Operation und einem anderen Befehl nicht mehr für die Kontrolleinheit der CPU verfügbar. Daher müssen die Flags, sobald eine Operation in der ALU ausgeführt wurde, in einem Register, dem sogenannten Flag Register, gespeichert werden. Das Flag Register befindet sich außerhalb der ALU und benötigt nur eine Speicherbreite von 3 Bits für die drei Flags. Neben dem CLK und CLR Signal benötigt es das Steuersignal L_F (Load Flag). Dieses wird von der Kontrolleinheit der CPU gesteuert. Das Register wird zur CPU hinzugefügt, wenn die ALU eingefügt wird (s. Implementierung von ALU und Flag-Register in der CPU).

Die ALU-Befehle 7 bis 9: Befehle zur Laufzeitoptimierung

Ein oft genutzter Befehl beim Programmieren ist das Erhöhen oder Verringern eines bestimmten Wertes um eins (zum Beispiel im Zusammenhang mit dem Durchlaufen von Schleifen). Geht man davon aus, dass man den ADD Befehl der ALU nutzen würde, um einen Wert um eins zu erhöhen, müsste man unter anderem den Wert „1“ jedes Mal separat als Operator B in die ALU laden. Das verbraucht zum einen unnötig Speicher der CPU, zum anderen dauert es, bis der Wert „1“ in das TMP-Register und damit in die ALU geladen wird. Daher soll die ALU zwei zusätzliche Befehle ausführen können, die den Operanden A um eins verringern (DEC) beziehungsweise um eins erhöhen (INC) (Malvino, 1999, S. 178).

Der $AOUT$ Befehl erscheint auf den ersten Blick sinnlos, da er einfach nur den Inhalt das A-Registers ausgibt, obwohl dieses das genauso gut auch selbst kann. Der wichtige Punkt hierbei ist aber, dass bei dem $AOUT$ Befehl als ALU Operation die Möglichkeit besteht, das Flag-Register zu laden. Der $AOUT$-Befehl wird also dazu genutzt, um die Flags für den Inhalt des A-Registers zu setzen, was andernfalls nur möglich wäre durch eine Addition mit 0, wobei sich hier aber wieder das oben beschriebene Problem ergibt. Eigentlich ist dieser Befehl nicht Teil einer typischen ALU, vor allem einer der SAP-Architektur, jedoch hat sich seine Notwendigkeit aufgrund meiner abgeänderten ALU-Architektur ergeben.

Alle drei Befehle sollen dadurch implementiert werden, dass bestimmte Parameter gesetzt werden, wenn eine der Operationen ausgewählt ist. Solche Parameter sind zum Teil bereits vorhanden: SUB und C_{in} müssen beispielsweise beide für die SUB-Operation der ALU gesetzt werden. Beide können auch für INC, DEC und $AOUT$ genutzt werden. Zusätzlich gilt jedoch für die drei Operationen, dass sie unabhängig von dem B-Operanden sind. Das bedeutet, dass, wenn eine der drei Operationen ausgewählt ist, der B-Operand in die ALU deaktiviert werden muss beziehungsweise Null sein muss. Das lässt sich realisieren, indem ein Und-Gatter vor den B-Eingang der 1-Bit ALUs geschaltet wird. Eine Eingabe des Und-Gatters ist der Wert des B-Operanden, der andere Eingang legt fest, ob dieser durchgelassen wird oder die Und-Verknüpfung immer „0“ ausgibt. Alle Steuereingänge der acht Und-Gatter werden zusammengeführt und bilden den dritten Parameter zum Steuern der ALU Operationen, B_{Active}.

Nun gilt es, die drei Parameter richtig zu setzen, wenn eine der vier Operationen SUB, INC, DEC oder $AOUT$ ausgewählt ist. Es wird jeweils ein Signal pro Operation benötigt, das aktiv ist, wenn die Operation ausgewählt ist und zum Ansteuern der Parameter für die jeweilige Operation verwendet

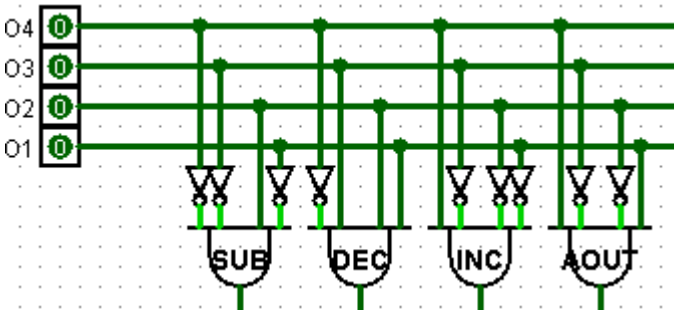


Abb. 7: Verknüpfung von O1 bis O4, um die Auswahl einer bestimmten Operation festzustellen

wird. Das Lässt sich mit Und-Gattern realisieren, die O_1 bis O_4 entweder direkt oder negiert als Eingabe verwenden. Durch Kombination aus direkter und negierter Eingabe lassen sich die Eingabesignale der Und-Gatter so verknüpfen, als dass alle aktiv sind, wenn eine bestimmte Kombination von O_1 bis O_4, also eine bestimmte Operation ausgewählt ist. Das Prinzip ist ähnlich dem eines Multiplexers.

Da ein Parameter von mehreren Operationen gesetzt werden muss, wird vor die Parameter ein OR-Gatter gesetzt, das als Eingabe alle Signale nimmt, für welche Befehle dieser Parameter gesetzt werden muss. Da für DEC, INC und $AOUT$ der Operand B **deaktiviert** und nicht aktiviert werden soll, wird der B_{Active} Parameter negiert.

Für SUB sind die Parameter einfach zusetzten. Wie bereits erklärt, müssen für die SUB Operation die Parameter C_{in} und SUB gesetzt werden.

Für DEC muss zum einen der B-Operand deaktiviert werden. Außerdem muss 1 subtrahiert, beziehungsweise -1 addiert werden. -1 ist im Zweierkomplement in der 8-Bit CPU 11111111. Also muss der B-Operand irgendwie auf 11111111 gesetzt werden. Dies lässt sich realisieren, indem der SUB Parameter gesetzt wird. Dieser negiert den B-Operanden. Da der B-Operand deaktiviert, beziehungsweise auf 00000000 gesetzt ist, sorgt der SUB Parameter dafür, dass dieser zu 11111111 wird, also zu -1.

Auch bei der Operation INC muss der B-Operand deaktiviert werden. Außerdem muss 1 addiert werden, was sich realisieren lässt, indem C_{in} gesetzt wird.

Für $AOUT$ muss nur der A-Operand ausgegeben werden, also wird B deaktiviert.

Daraus ergibt sich die ALU der CPU. Sie hat die Eingänge A und B, sowie den Ausgang OUT, die jeweils 8 Bit breit sind. Die Steuersignale O_1 bis O_4 geben die auszuführende Operation an und die drei Flags Carry, Sign und Zero stellen Informationen über die letzte Rechenoperation der ALU bereit. Für die fertige ALU siehe Anhang 1.2.5.

Implementierung von ALU und Flag-Register in der CPU

Die ALU findet ihren Platz in der CPU direkt zwischen A- und TMP-Register. Da sie direkt den Inhalt der Register auslesen soll, wird der A-Operand mit dem $DIR\ OUT$ des A-Registers verbunden, der B-Operand wird mit dem $DIR\ OUT$ des TMP-Registers verbunden. OUT der ALU wird direkt mit dem Datenbus verbunden. Die Steuersignale O_1 bis O_4 werden zunächst zur Seite geleitet, neben die Steuersignale der Register. Die drei Flags werden mit dem bereits erwähnten Flag-Register verbunden. Die CLR und CLK Signale des Flag Registers werden mit den allgemeinen CLR und CLK Signalen der CPU verbunden. Das LF Signal wird zu den anderen Steuersignalen geführt (Anhang 1.2.6).

Das Ausgaberegister

Das Ausgaberegister der CPU ist dafür verantwortlich, dass die CPU mit anderen, externen Komponenten kommunizieren kann. Im Ausgaberegister werden Daten bereitgestellt, die dann von den anderen Komponenten ausgelesen werden können. Normalerweise lädt die CPU Daten in das Ausgaberegister und setzt dann ein Statussignal, das den externen Komponenten signalisiert, dass die Daten im Ausgaberegister gültig und fertig geladen sind. Sobald die Daten ausgelesen wurden, wird dieses Signal von der Komponente, die ausgelesen hat, zurückgesetzt, damit die CPU und andere Komponenten wissen, dass der Kommunikationsprozess über das Ausgaberegister abgeschlossen wurde. In dieser 8-Bit CPU wird das ganze vereinfacht: Die CPU kann Daten in das Ausgaberegister laden, wann immer sie will. An das Ausgaberegister wird auch nur eine Komponente angeschlossen: Ein Dezimaldisplay, der das ins Ausgaberegister geladene Datenwort als 8 Bit Unsigned Integer darstellt.

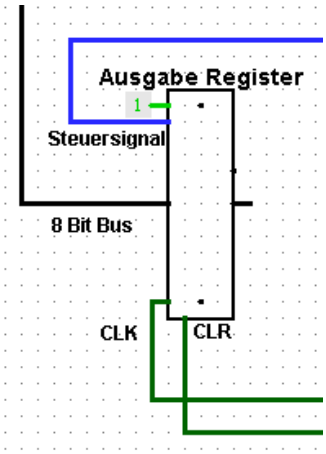


Abb. 8: Das Ausgaberegister

Das Ausgaberegister ist ein normales 8-Bit Register. Sein Eingang wird mit dem Datenbus verbunden, die CLK und CLR Signale werden mit den gemeinsamen CLK und CLR Signalen verbunden. Da das Register dauerhaft an das Display ausgeben soll, wird der $DIR\ OUT$ Ausgang verwendet, der $Enable$ Eingang wird nicht benötigt. Der Load Eingang wird ein weiteres Steuersignal, LO (Load Output Register).

Konzept: Dezimal Display

Die größte Zahl, die das Display darstellen muss, ist 255. Daher benötigt unser Dezimal Display drei Ziffern. Logisim bietet als Baustein eine 7-Segment-Anzeige (Wie die Anzeige bei einem digitalen Wecker). Mit den 8 Eingabesignalen können die sieben Segmente sowie ein Punkt angeschaltet werden. Jetzt benötigt man eine logische Schaltung, die, je nach im Ausgaberegister geladener Zahl, die drei einzelnen Displaysegmente richtig ansteuert und somit die binäre Zahl in ihrer Dezimalschreibweise dargestellt wird. Theoretisch ließe sich das ganze wieder über eine kombinatorische Schaltung mit Und-Gattern realisieren: Für die Zahl 00000110 (=6) müssen die Segmente angeschaltet werden, die die 6 darstellen würden, alle anderen bleiben aus. Eine solche Schaltung wird bei 255 Möglichen Kombinationen sehr schnell sehr groß und aufwändig. Daher wird ein sogenannter ROM-Baustein verwendet.

Adressengesteuerte Speicherbausteine (ROM)

ROM steht für Read Only Memory, zu Deutsch „Nur Lese Speicher“. Ein ROM-Speicher kann mehrere Datenworte unabhängig voneinander speichern. Um auszuwählen, welches dieser Datenworte ausgegeben werden soll, hat jedes Datenwort eine bestimmte Adresse. Über den Adressen-Eingang kann also ausgewählt werden, welches Speicherwort ausgegeben wird. Der Name Read Only Memory besagt, dass man im Falle eines ROM tatsächlich keine Daten in den einzelnen Adressen speichern, sondern nur auslesen kann. Der Inhalt der Adressen wird im Vorhinein in den ROM geschrieben, zum Beispiel bei der Herstellung. So können beispielsweise verschiedene ROM-Chips für verschiedene Zwecke direkt gefunden werden. Es gibt auch andere Arten von ROMs, die selbst beschrieben werden können (EPROMs, EEPROMs) (Malvino, 1999, S. 161, S. 130f). Mehr über adressengesteuerte Speicherbausteine im Abschnitt „Der Arbeitsspeicher (RAM)
“.

Im Falle unseres Dezimal Displays kann uns ein ROM-Speicher helfen, die binäre Zahl zu dekodieren. Ein ROM-Baustein mit 8 Adressenbits kann 255 verschieden Datenworte speichern. Ein solches Datenwort kann nun die Kombination sein, die die Segmente für die drei Anzeigen richtig schaltet und ansteuert, um die Zahl, in dessen Adresse das Datenwort liegt, darzustellen. Für drei einzelne 7-Segment Displays muss das Datenwort des ROM 24 Bit breit sein.

Logisim stellt ROM-Bausteine mit variablen Adressen- und Datenwortbreiten bereit, deren Inhalt man selbst festlegen kann.

Implementierung des Dezimal Displays

Zunächst werden die einzelnen Signale der 7-Segment Anzeigen zusammengeführt zu einem 24 Bit breiten Kontrollwort für die drei Anzeigen. Dieses 24-Bit Signal wird mit dem Ausgang der ROM verbunden. Der Adresseingang der ROM wird mit dem $DIR\ OUT$ Ausgang des Ausgaberegisters verbunden.

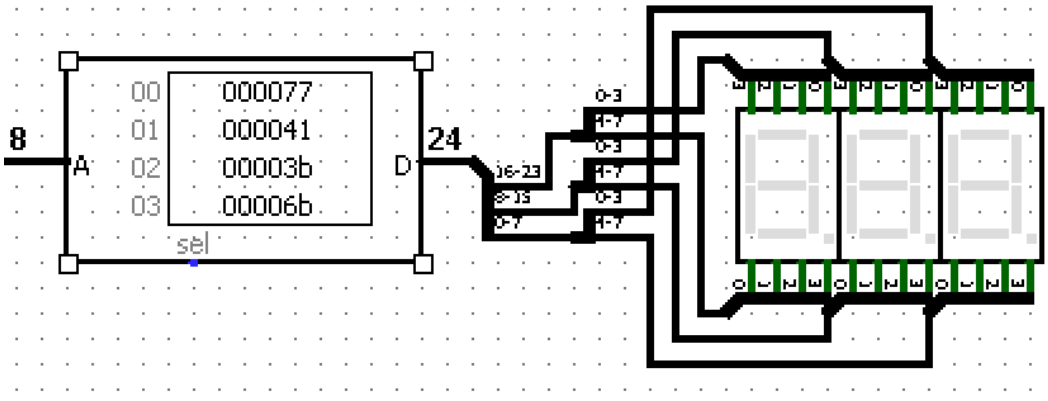


Abb. 9: Layout des Dezimal Displays: Das 24 Bit Signal des ROM-Speichers wird aufgefächert

Jetzt gilt es, den ROM richtig zu füllen. Logisim erlaubt es, eine Datei in den ROM zu laden, die nacheinander den Inhalt jeder Adresse enthält. Daher habe ich ein kleines Programm geschrieben, das jeder ROM-Adresse das passende Steuerwort zuweist, um die drei 7-Segment Displays entsprechend anzusteuern (Anhang 1.3.1, Anhang 1.3.2).

Das Dezimal Display wird in der CPU direkt neben dem Ausgaberegister platziert (Anhang 1.3.3).

Das Ausgaberegisterdesign stammt von Ben Eater (2017, „Building an 8-bit decimal display for our 8-bit computer“)

Der Arbeitsspeicher (RAM)

Was ist der RAM?

Die internen Register der CPU werden zum Zwischenspeichern einzelner Datenworte über mehrere Taktzyklen verwendet. Zwar können diese Register in heutigen Prozessoren (dort sog. Caches) mehrere Datenworte speichern, aber sind dabei auch nur auf eine Speichergröße von wenigen Megabytes begrenzt. Um größere Datenmengen zwischenzuspeichern und diese für den schnellen Zugriff bereitzuhalten, gibt es den sogenannten Arbeitsspeicher (RAM). Im RAM ist zudem das gerade auszuführende Programm gespeichert. Der RAM befindet sich eigentlich nicht in der CPU, wird aber aufgrund seiner Notwendigkeit hier erwähnt und in die CPU integriert. Normalerweise ist der Arbeitsspeicher über einen Speichercontroller (innerhalb oder außerhalb der CPU) mit der CPU verbunden (Riley/Brailsford, 2018).

RAM steht für Random Access Memory und bedeutet frei übersetzt so viel wie „Speicherbaustein mit wahlfreiem Zugriff“. Tatsächlich ist der RAM ähnlich wie der ROM ein adressengesteuerter Speicherbaustein. Es werden Datenworte unter verschiedenen Adressen gespeichert. Die Werte der einzelnen Adressen können ausgelesen werden und, im Unterschied zum ROM, auch beschrieben werden. Eine vereinfachte Darstellung eines RAM-Bausteins wären somit viele einzelne Register, von denen jeweils eins abhängig von der angewählten Adresse aktiv ist und mit dem Datenbus verbunden ist (Malvino, 1999, S. 133).

Somit besteht ein RAM Baustein aus zwei wichtigen Komponenten: Zum einen dem Adressendekodierer und zum anderen den einzelnen Speicherzellen. Der Adressendekodierer sorgt dafür, dass die richtigen Speicherzellen abhängig von der angewählten Adresse ausgewählt sind, damit diese beschrieben oder ausgelesen werden können. Die einzelnen Speicherzellen wiederum speichern die Datenworte. In heutigen RAM Bausteinen werden diese Speicherzellen nicht wie in den Registern aus bistabilen logischen Schaltungen aufgebaut („Static RAM“, SRAM), da eine solche Bauart zu groß, zu aufwändig und zu teuer ist, um große RAM-Bausteine zu fertigen. Stattdessen verwendet man für eine einzelne Speicherzelle einen Kondensator, der zyklisch immer wieder über einen Transistor aufgeladen wird, falls die Speicherzelle eine 1 gespeichert hat. Eine solche Bauart wird auch als „Dynamic RAM“, kurz DRAM bezeichnet (Malvino, 1999, S. 134).

Der RAM in der 8-Bit CPU

Der RAM in der CPU hat eine Speichergröße von 256 Byte. Über den 8 Bit breiten Adresseneingang können 2^8, also 256 Adressen, in denen jeweils ein 8 Bit (1 Byte) breites Datenwort gespeichert ist, angesteuert werden. Einen DRAM Baustein von Grund auf zu bauen, ist zu komplex. Logisim bietet aber einen RAM-Baustein an. Dieser bietet separate Daten Ein- und Ausgänge, synchrone Schreib- und Lesevorgänge und einen mit Tristates gepufferten Ausgang, der aktiviert und deaktiviert werden kann.

Der Datenausgang D_{out} und der Dateneingang D_{in} werden direkt mit dem Datenbus verbunden. Das CLK Signal des RAM wird mit dem allgemeinen CLK Signal verbunden, die beiden Signale RR (Read RAM) und WR (Write RAM) dienen als Steuersignale, um die angewählte Adresse auszulesen und an den Bus über D_{out} auszugeben, beziehungsweise um den über D_{in} eingegebenen Wert des Busses in der angegebenen Adresse zu speichern.

Die aktuelle Speicheradresse wird in einem extra Register, dem Adressenregister, gespeichert. Der Dateneingang D dieses Registers wird mit dem Bus verbunden, die direkte Ausgabe des Registers mit dem Adresseneingang des RAM. Die beiden Signale CLR und CLK werden wie üblich mit den allgemeinen CLR und CLK Signalen verknüpft. Als Steuersignal wird nur das $Load$ Signal des Registers benötigt. Mit LM (Load Memory Address) wird der Inhalt des Datenbusses in das Adressenregister

geladen. Aus RAM-Baustein und Adressenregister ergibt sich nun das RAM Modul der 8 Bit CPU (Anhang 1.4.1).

Der Befehlszähler (Pointer)

Das auszuführende Programm der CPU wird in dem RAM gespeichert. Dabei beginnt das Programm bei der Adresse 0 ($0\text{x}00_{hex}$) und die einzelnen Befehle sind in aufeinanderfolgenden Adressen gespeichert. Die CPU muss dabei jederzeit wissen, welcher Befehl des Programmes als nächstes ausgeführt werden soll. Dazu wird eine weitere Komponente benötigt, nämlich der sogenannte Befehlszähler. Dieser behält den Überblick darüber, bei welchem Befehl des Programmes die Ausführung gerade ist und aus welcher RAM Adresse der nächste Befehl geladen werden muss. Wie der Name schon sagt, muss der Befehlszähler zählen können: Nach der Ausführung eines Befehls wird der Befehlszähler erhöht, damit er auf die nächste RAM Adresse, also den nächsten Programmbefehl zeigt („zeigen“ = engl. „to point“, daher auch Pointer). Außerdem muss es zusätzlich möglich sein, eine beliebige Adresse in den Befehlszähler zu laden, von der aus er weiterzählen kann. Diese Funktion wird für Befehle benötigt, die zwischen verschiedenen Teilen des Programmes springen müssen, beispielsweise für Schleifen (Malvino, 1999, S. 112f).

Erweiterung des SR-Flip-Flops: Der JK-Flip-Flop

Um einen Zähler implementieren zu können, benötigt man einen weiteren Speicherbaustein: Das JK-Flip-Flop, der eine Erweiterung des SR-Flip-Flops darstellt.

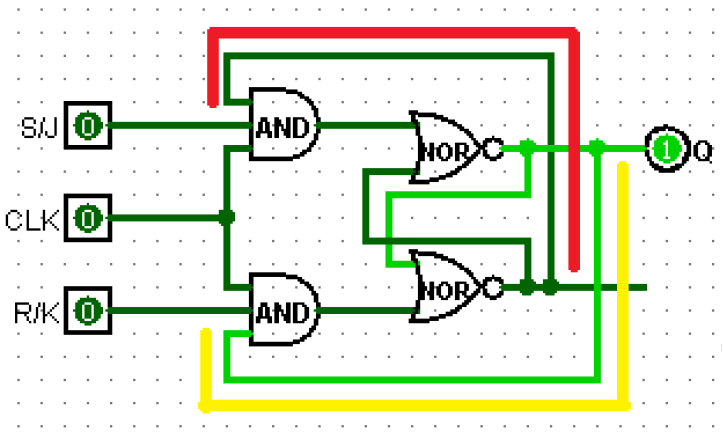


Abb. 10: Der JK-Flip-Flop als Erweiterung des SR-Flip-Flops

Im Grunde ist das JK-Flip-Flop ein normales SR-Flip-Flop. Über Set (S, jetzt J) kann das SR-Flip-Flop synchron zum Takt gesetzt, über $Reset$ (R, jetzt K) synchron zurückgesetzt werden. Die Und-Gatter, die R und S synchron zum Takt schalten, haben jetzt aber einen dritten Eingang: Das Set Und-Gatter nimmt die invertierte Ausgabe des SR-Flip-Flops als dritte Eingabe, das $Reset$ Und-Gatter die normale Ausgabe des SR-Flip-Flops. Wird nun entweder Set oder $Reset$ gesetzt, so hat diese Erweiterung keinen Einfluss: Ist $Reset$ gesetzt, soll das JK-Flip-Flop zurückgesetzt werden. Also wird das $Reset$-Signal durchgelassen, wenn eine „1“ im Flip-Flop gespeichert ist und somit am Und-Gatter anliegt, und dieser wird zurückgesetzt. Ist bereits eine „0“ im Flip-Flop gespeichert, so geht das $Reset$-Signal zwar nicht durch das Und-Gatter durch, aber das muss es auch nicht, denn das JK-Flip-Flop ist bereits zurückgesetzt. Falls das Set-Signal gesetzt wird, gilt dasselbe, jedoch wird in diesem Fall die negierte Ausgabe des Flip-Flops verwendet, denn das Set-Signal soll durch das Und-Gatter gelassen werden, wenn im Flip-Flop eine „0“ gespeichert ist.

Soweit gibt es also keinen Unterschied zwischen SR- und JK-Flip-Flop. Schaut man sich jedoch eine weitere Möglichkeit an, wie man die Eingänge beschalten kann, so zeigt sich der Unterschied. Was passiert, wenn J und K, also Set und $Reset$ gleichzeitig gesetzt sind? Das SR-Flip-Flop kann wohl nicht gleichzeitig gesetzt und wieder zurückgesetzt werden (Malvinno, 1999, S. 99f).

Tatsächlich befindet sich das SR-Flip-Flop in diesem Fall in einem undefinierten Zustand. Sowohl der Ausgang des Flip-Flops, als auch der negierte Ausgang fallen beide auf 0. Erst wenn nur noch einer der beiden Eingänge gesetzt ist, fällt das Flip-Flop wieder in einen definierten Zustand. Dieser neue Zustand ist abhängig davon, ob zuerst Set oder $Reset$ auf zurückgesetzt wurde. Versucht man beide gleichzeitig auszuschalten, so fällt das Flip-Flop in einen zufälligen Zustand, denn einer der beiden

Eingänge wird „später“ als der andere auf „0“ gesetzt werden aufgrund des Stromflusses und unterschiedlicher elektrischer Vorgänge (Quelle [2], o.V., 2011, S. 6).

Das JK-Flip-Flop hingegen ist für den Fall, das sowohl J als auch K gesetzt wird, klar definiert. Das Flip-Flop speichert als neuen Wert den negierten Wert, der vorher im Flip-Flop gespeichert war:

$$Q_{n+1} = \overline{Q_n}$$

Je nachdem, ob eine „1“ oder eine „0“ im JK-Flip-Flop gespeichert ist, ist, wie bereits erläutert, nur eines der beiden Und-Gatter „aktiv“ und lässt das Set bzw. das $Reset$ Signal durch. Ist das Flip-Flop gesetzt, so wird das $Reset$-Signal durchgelassen und das Flip-Flop zurückgesetzt. Andersherum wird das Flip-Flop gesetzt, wenn es vorher auf 0 gesetzt war und deshalb das Set-Signal durchgelassen wird (Malvino, 1999, S. 99).

Die reale Implementierung eines JK-Flip-Flops ist aber nicht ganz so einfach wie beschrieben; es ergeben sich Probleme aufgrund der Beschaffenheit der elektronischen Bauteile. Darauf wird hier nicht weiter eingegangen. Die reale Implementierung eines JK-Flip-Flops umfasst eine Schaltung bestehend aus zwei JK-Flip-Flops, zusammengeschaltet zu einem Master-Slave JK-Flip-Flop. Dieses funktioniert ohne Probleme und erfüllt die benötigten Anforderungen. Wichtiger Unterschied ist, dass das Master-Slave JK-Flip-Flop erst auf der fallenden Taktflanke seine Ausgabe umschaltet. Um in seinen neuen Zustand zu schalten, benötigt es nämlich sowohl die steigende, als auch die fallende Taktflanke, also einen ganzen Zyklus des CLK-Signals (Eater, 2017, „JK Flip-Flop Racing“, „Master-slave JK Flip-Flop“).

Einfacher Binärzähler

Das Master-Slave JK-Flip-Flop benötigt somit einen vollen CLK Zyklus, um seine Ausgabe umzuschalten. Damit das Flip-Flop einmal von „0“ auf „1“ und wieder zurückschaltet, müssen dementsprechend zwei volle CLK Zyklen durchlaufen werden. Sind bei einem Master-Slave JK-Flip-Flop J und K gesetzt, beträgt die Frequenz des Ausgabesignals Q des Flip-Flops nur noch die Hälfte der Frequenz des CLK Signals. Diese Halbierung der Frequenz lässt sich weiter fortführen, indem man die Ausgabe des ersten Flip-Flops an das CLK Signal eines zweiten Flip-Flops anschließt.

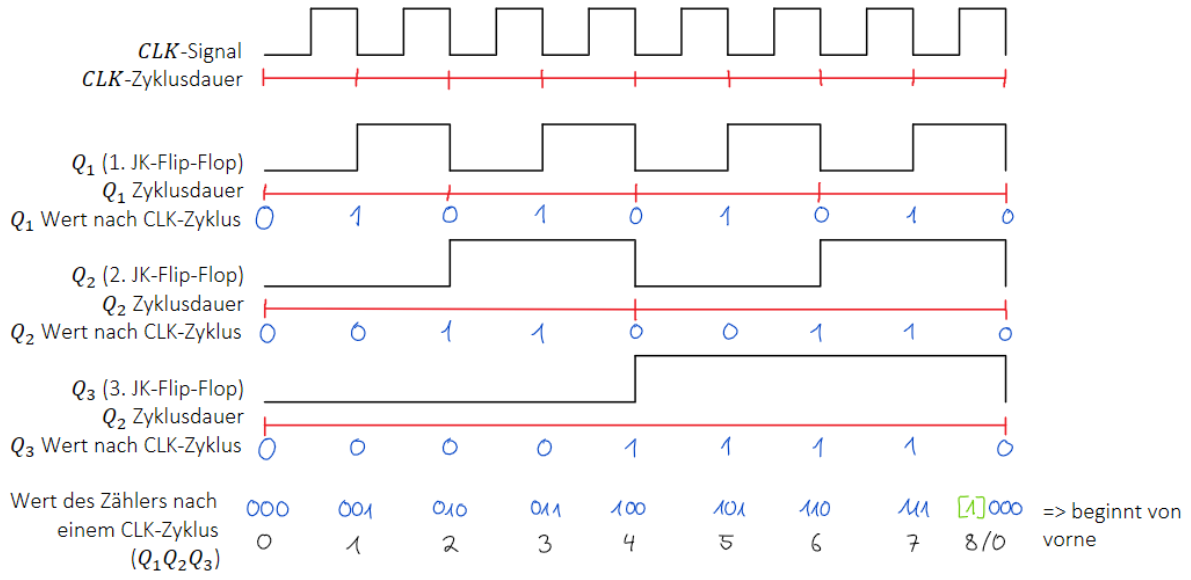


Abb. 11: Timing Diagramm für die Ausgabe einzelner, zu einem Zähler hintereinandergeschalteter JK-Flip-Flops nach Malvino, 1999, S. 111

Abb. 11 zeigt das Ausgabediagramm für die einzelnen JK-Flip-Flops und des CLK-Signals eines einfachen Binärzählers, wie er oben bereits beschrieben wurde (Anhang 1.5.1). Es wird deutlich, wie ein Zyklus des einkommenden Signals (im Diagramm darüber liegend) zu einer Zustandsänderung führt. Auch die daraus resultierende Halbierung der Frequenz ist gut ersichtlich. Um nun die Ausgabe des Binärzählers zu erhalten, schaut man sich die einzelnen Ausgaben der JK-Flip-Flops an, jeweils zu dem Zeitpunkt, wenn ein CLK Zyklus beendet ist, also jeder Master-Slave JK-Flop-Flop in seinen neuen Zustand übergangen ist, und setzt deren Ausgabebits nebeneinander. Das Flip-Flop mit der höchsten Frequenz ist dabei das niedrigstwertige Bit, das mit der niedrigsten Frequenz das höchstwertige Bit. Man sieht, dass mit jedem vollen Taktzyklus hochgezählt wird (Malvino, 1999, S. 110ff).

Vom Binärzähler zum Befehlszähler

Diesem Binärzähler fehlt nun aber die wichtige Funktion, eine bestimmte Zahl zu laden und von dieser aus weiter zu zählen. Letzteres ist dabei trivial, denn sobald eine bestimmte Zahl in den Zähler geladen ist, sorgt die Logik des Zählers dafür, dass automatisch von der geladenen Zahl aus weiter gezählt wird.

Um jedoch das Laden von Daten in den Befehlszähler zu implementieren, muss eine Logik zum richtigen Setzen der J und K Inputs, sowie des CLK Signals implementiert werden. Dazu geht man zunächst zurück von einem mehrere Bits breiten Zähler zu einem 1 Bit Zähler, der mit der Funktion zum Laden eines Wertes ausgestattet wird, aber natürlich auch auf Befehl durch Steuersignal um eins weiterzählen können soll.

Steuersignale des Befehlszählers:

- CLK: Taktgeber
- CLR: Setzt den Zähler asynchron auf 0 zurück
- C_{in}: Ausgabe des vorgeschalteten 1 Bit Zählers
- IP: Increment Pointer. Der Zähler wird mit dem nächsten CLK Zyklus erhöht
- D: Datenbit, das mit dem nächsten CLK Zyklus in den Zähler geladen wird
- LP: Lädt den Wert von D mit dem nächsten CLK Zyklus in den Zähler, wenn gesetzt

Diesen Steuersignalen muss eine entsprechende Logik verschaltet werden, damit die Eingabesignale des JK-Flip-Flops (J, K, CLK_{JK} und CLR_{JK}) entsprechend gesetzt werden.

J: Setzt das JK-Flip-Flop auf „1“. Muss gesetzt werden, wenn:

1. IP gesetzt ist. J und K müssen beide zum hochzählen aktiv sein.
2. LP und D gesetzt sind. Der Befehlszähler soll den Wert „1“ laden, das Flip-Flop muss gesetzt werden.

⇨ $J = IP \vee (LP \wedge D)$

K: Setzt das JK-Flip-Flop zurück auf „0“. Muss gesetzt werden, wenn:

1. IP gesetzt ist. J und K müssen beide zum hochzählen aktiv sein.
2. LP gesetzt ist und D nicht gesetzt ist. Der Befehlszähler soll den Wert „0“ laden, das Flip-Flop muss zurückgesetzt werden.

⇨ $K = IP \vee \left(LP \wedge \overline{D}\right)$

CLK_{JK}: Sorgt dafür, dass das Flip-Flop basierend auf J und K seinen Zustand ändert. Muss entweder zum Hochzählen oder zum Laden des Datenbits aktiviert werden, also wenn:

1. CLK und LP gesetzt sind. Ist LP gesetzt, soll zum nächsten CLK Zyklus das Flip-Flop D speichern, weshalb CLK_{JK} auch einen CLK Zyklus durchlaufen muss.

2. IP und C_{in} gesetzt sind. Ist IP gesetzt, soll der Zähler weiterzählen, basierend auf der Eingabe des vorgeschalteten Zählers.

⇨ $CLK_{JK} = (CLK \wedge LP) \vee (IP \wedge C_{in})$

Soll der Befehlszähler asynchron zurückgesetzt werden, muss nur das CLR_{JK} Signal gesetzt werden: $CLR_{JK} = CLR$.

Aus dieser Logik zum Steuern eines JK-Flip-Flops ergibt sich der 1-Bit Befehlszähler. Die Ausgabe des JK-Flip-Flops ist dabei die Ausgabe OUT des Befehlszählers (Anhang 1.5.2).

Der fertige Befehlszähler und die Implementierung in der CPU

Der 8-Bit Befehlszähler wird nun, ähnlich zur ALU, aus 8 hintereinandergeschalteten Zählern aufgebaut. Dabei wird OUT des jeweils vorherigen 1-Bit Befehlszählers mit dem C_{in}-Signal des nächsten verbunden, sodass wie beim einfachen 3-Bit Zähler die JK-Flip-Flops hochzählen können. Das CLK-Signal wird zunächst invertiert. Da das JK-Flip-Flop in den Befehlszählern erst bei der fallenden Taktflanke seinen Zustand aktualisiert, sorgt ein negiertes CLK-Signal dafür, dass dies nun bei der steigenden Taktflanke passiert. Damit läuft der Befehlszähler synchron mit den anderen CPU-Komponenten, die alle bei der steigenden Flanke aktualisieren. Das CLK-Signal des 8-Bit Befehlszählers wird zunächst mit allen CLK-Signalen der 1-Bit Befehlszähler verbunden. Außerdem wird das C_{in}-Signal des ersten Zählers mit dem CLK-Signal verbunden. Zudem werden jeweils alle CLR, IP und LP Steuersignale der einzelnen 1-Bit Befehlszähler verbunden. Da auch die Ausgabe des Befehlszählers mit dem Bus verbunden wird, muss hinter die 1-Bit Zähler jeweils ein Tristate geschaltet werden. Die Steuersignale der Tristates ergeben zusammengeführt das Steuersignal $Enable\ Pointer$ (EP). Der fertige Befehlszähler verfügt nun über die Steuersignale EP, IP und LP sowie ein CLR und ein CLK Signal, den 8 Bit breiten Dateneingang D und den ebenso breiten Datenausgang OUT.

Der Befehlszähler wird unter dem RAM-Modul platziert. D und OUT werden direkt mit dem Datenbus verbunden, CLR und CLK mit den allgemeinen Signalen der CPU. Die drei Steuersignale werden zu den anderen Steuersignalen der CPU geführt (Anhang 1.5.3).

Die Kontrolleinheit

Die Kontrolleinheit der CPU ist ihre zentrale Steuerungseinheit. Sie ist für die Ausführung eines in dem RAM abgelegten Befehls zuständig, indem sie die einzelnen Komponenten der CPU ansteuert und beispielsweise Daten über den Bus zwischen den einzelnen Komponenten austauscht oder eine Operation über die ALU ausführt (Malvino, 1999, S. 146).

Das Befehlsregister

Damit ein Befehl ausgeführt werden kann, muss er zunächst in die Kontrolleinheit geladen werden, wo er während seiner gesamten Ausführung gehalten wird. Dies geschieht über das Befehlsregister. Das Befehlsregister ist ein 8-Bit Register, das mit dem Bus verbunden wird, damit es die Befehle aus dem RAM speichern kann. Da die Kontrolleinheit dauerhaft über den gerade auszuführenden Befehl verfügen muss, wird lediglich das Steuersignal zum Laden des Befehlsregisters benötigt (Malvino, 1999, S. 141). Dieses neue Steuersignal L_I ($Load\ Instruction$) wird zu den anderen Steuersignalen geführt und mit diesen gesammelt. Des Weiteren werden das CLR und das CLK Signal des Registers mit den allgemeinen Signalen der CPU verbunden. Das Befehlsregister befindet sich nun unter dem C-Register (Anhang 1.6.1)

Außerdem muss noch eine kleine Änderung am CLK Signal vorgenommen werden. Bisher wird das CLK Signal direkt über einen Taktgeber angesteuert. Die Kontrolleinheit der CPU soll aber die Möglichkeit haben, die Ausführung eines Programms zu beenden und die CPU zu stoppen beziehungsweise anzuhalten. Dazu wird ein weiteres Steuersignal eingeführt, das HLT Signal. Hinter den Taktgeber wir ein Und-Gatter geschaltet, das das Taktgebersignal und das negierte HLT Signal als Eingabe nimmt. So wird das Taktgebersignal nur dann als allgemeines CLK Signal ausgegeben, wenn HLT nicht gesetzt ist. Andernfalls stoppt die CPU, da das CLK Signal dauerhaft „0“ ist (Anhang 1.6.2).

Steuersignale und das Steuerwort

Das Ansteuern der einzelnen Komponenten der CPU, wie es die Aufgabe der Kontrolleinheit ist, läuft über die Steuersignale der einzelnen Komponenten, die alle bereits beim Bau der jeweiligen Komponenten erwähnt wurden und alle nebeneinander gesammelt wurden. Die schematische Darstellung der Komponenten der CPU, sowie die Steuersignale und deren Erklärung findet sich im Anhang 1.6.4 und Anhang 1.6.5. Ein Steuersignal (C_C) wurde bisher noch nicht erläutert. Dieses Steuersignal wird erst später eingeführt werden und ist jetzt noch nicht relevant.

Alle Steuersignale nebeneinander ergeben ein 23 Bit breites sogenanntes Steuerwort. Will man also einzelne Steuersignale setzten, so geschieht dies, indem bestimmtes Steuerwort gesetzt wird. Dazu folgen zwei Beispiele.

Soll der Inhalt des C-Registers in das A-Register verschieben, so müssen zwei Steuersignale gesetzt werden: E_C, damit das C-Register an den Bus ausgibt, und L_A, damit das A-Register vom Bus liest. Nach der Anordnung der Steuersignale ergibt sich daraus das folgende Steuerwort: 00000000000000000100100. Dieses Steuerwort ist ein Mikrobefehl, also ein Befehl, der innerhalb der Kontrolleinheit der CPU verwendet wird, um beispielsweise den Datenfluss zwischen den Komponenten der CPU zu regeln (Malvino, 1999, S. 141f).

Der nächste Mikrobefehl soll den Inhalt des A-Registers zum Temporären Register addieren und das Ergebnis im C-Register speichern. In der ALU ist Addition die Operation 1, somit müssen O_4, O_3, O_2 und O_1 0001 sein. Da A- und TMP-Register die Operanden der ALU sind und direkt mit ihr verbunden sind, so muss nur noch L_C gesetzt werden, damit das Ergebnis in C geladen wird. Das Steuerwort für diesen Mikrobefehl lautet: 00000000000001000000010.

Die Arbeitsweise der Kontrolleinheit

Nachdem erläutert wurde, wie die einzelnen Komponenten der CPU genau angesteuert werden und wie bereits kleine Mikrobefehle intern in der CPU realisiert werden, lässt sich die Funktionsweise der Kontrolleinheit zum Ausführen komplexerer Maschinenbefehle erläutern. Zunächst muss die Kontrolleinheit selbstständig den Befehl in das Befehlsregister laden. Diesen Vorgang nennt man auch **Fetch** (engl.: herbeiholen, => laden). Sobald er geladen ist, wird er in der Kontrolleinheit dekodiert (**Decode**) und ausgeführt (**Execute**). Dieser Zyklus (**Fetch**, **Decode**, **Execute**) ist die typische Arbeitsweise einer Kontrolleinheit (Silc, 1999, VIII; Malvino, 1999, S. 146-148). Diese ist somit die Logik zum Übersetzen der Maschinenbefehle aus dem RAM in Mikrobefehle, die die einzelnen Komponenten der CPU steuern. Die Kontrolleinheit sorgt dafür, dass die Maschinenbefehle dekodiert werden und das Steuerwort passend gesetzt wird.

Diese Art von logischer Schaltung gibt somit für einen bestimmten Maschinenbefehl als Eingabe das passende Steuerwort als Ausgabe aus. Die einfachste und beste Möglichkeit ist daher wieder die Verwendung eines ROM, ähnlich wie bei der Dezimalanzeige. Dort muss das passende „Steuerwort“ für die 7-Segment-Anzeigen gesetzt werden, abhängig von der anzuzeigenden binären Zahl. Der ROM beinhaltet den Mikrocode, der jedem Maschinenbefehl eine Serie an Mikrobefehlen zuweist, damit der Befehl ausgeführt wird (Malvino, 1999, S. 161).

Exkurs: Terminologie zu Befehlen auf Maschinenebene

Bei der Erläuterung der Kontrolleinheit wird das Wort „Befehl“ häufig verwendet, dabei auch oft in unterschiedlichem Kontext. Dazu eine kleine Übersicht:

Als **Maschinenbefehl** versteht man einen Befehl, der im RAM des Computers (der CPU) abgelegt wird und von der CPU geladen und ausgeführt wird. Ein Maschinenbefehl besteht zwingend aus dem **Opcode**, in einigen Fällen zusätzlich noch aus einem **Operanden**. Während der Opcode (Operationscode) die binäre Darstellung des Befehls ist, wie ihn die CPU in die Kontrolleinheit lädt, so ist die **Mnemonic** (engl.: Gedächtnisstütze) ein für den Menschen leicht merkbares Kürzel des Opcodes. Als Beispiel nehmen wir einen Maschinenbefehl, der eine bestimmte Zahl in das A-Register lädt, kurz genannt LVA (**L**oad **V**alue **A**, LVA ist die Mnemonic). Der Opcode für einen solchen Befehl ist natürlich für jede CPU unterschiedlich, in dem Beispiel gehe ich von dem Befehlsset dieser 8-Bit CPU aus. Der Opcode für LVA ist 00000001 in binärer Schreibweise. Dem Opcode folgt die Zahl, die in das Register geladen werden soll, beispielsweise 45, also 00101101 in Binärcode. Es ergibt sich folgender Maschinenbefehl:

a) LVA 0d45 (Mnemonic und Dezimalzahl, für den Menschen einfach lesbar)
b) $00000001\ 00101101$ (Opcode und Binärzahl, für die Maschine einfach „lesbar“)

Ein **Mikrobefehl** ist nun ein Befehl innerhalb der CPU. Es sind Steuerworte, die innerhalb der CPU dafür sorgen, dass die verschiedenen Komponenten sinnvoll verschaltet werden. Ein komplexer Maschinenbefehl wird durch die Kontrolleinheit in **Mikrocode** übersetzt, also eine Sequenz an Mikrobefehlen, die die Komponenten in der CPU so verschalten, als dass der Maschinenbefehl ausgeführt wird.

(Malvino, 1999, S. 143ff)

Wie bereits erwähnt, wird ein Maschinenbefehl über mehrere Taktzyklen hinweg ausgeführt, da er aus mehreren einzelnen Mikrobefehlen besteht. Somit ist das auszugebende Steuerwort der Kontrolleinheit nicht nur von dem Befehl im Befehlsregister abhängig, sondern auch davon, bei welchem Schritt der Ausführung des Befehls sich die CPU gerade befindet. Ein Binärzähler behält den

Überblick darüber, bei welchem Ausführungsschritt eines Befehls sich die CPU gerade befindet. Dieser Binärzähler hat eine Ausgabe von 3 Bit und zählt somit von 0 bis 7. Der Mikrocode für einen Maschinenbefehl dieser 8-Bit CPU besteht somit aus maximal acht Mikrobefehlen.

Als Zähler wird der 3-Bit Zähler aus Anhang 1.5.1 verwendet. Er wird unter dem Befehlsregister platziert und mit dem CLK Signal verbunden. Das CLR Signal des Zählers wird über ein Oder-Gatter zum einen mit dem allgemeinen CLR Signal verbunden, zum anderen mit dem letzten Steuersignal C_C, das den Zähler am Ende der Ausführung eines Maschinenbefehls zurücksetzt. So wird der nächste Befehl geladen und seine Ausführung beginnt bei Schritt „0“ (Malvino, 1999, S. 162f) (Anhang 1.6.6).

Der ROM der Kontrolleinheit wird daher abhängig von dem Befehlsregister und dem Zähler adressiert. Die Adresse ist eine Kombination aus beidem, hat also eine Breite von 11 Bit. Somit beinhaltet jede individuelle Adresse des ROM den Mikrobefehl für einen einzelnen Ausführungsschritt eines Maschinenbefehls (Eater, 2017, „8-bit CPU control logic: Part 3“).

11 Bit Adresse der ROM der Kontrolleinheit

Die Adresse: 00000000 000
Gelb: 8-Bit Befehlsregister. Ermöglicht somit maximal $2^8 = 256$ verschiedene Maschinenbefehle
Blau: 3-Bit Taktzähler. Ein Maschinenbefehl wird über maximal $2^3 = 8$ Taktzyklen ausgeführt

Um den ROM der Kontrolleinheit in der CPU zu implementieren, wird der fertige und programmierbare ROM Baustein von Logisim verwendet. Als Adresseneingang wird die erwähnte Zusammenführung des Befehlsregisters und des Zählers verwendet, die Ausgabe ist das Steuerwort. Dieses wird aufgefächert und mit den einzelnen Steuersignalen verbunden (Anhang 1.6.7). Der finale Schritt zum Fertigstellen der CPU ist nun, den Mikrocode für die einzelnen Maschinenbefehle zu entwickeln und diesen entsprechend in den ROM zu programmieren. Im Folgenden wird nur der Mikrocode an sich erläutert, nicht jedoch dessen Programmierung in den ROM. Dafür wird das bereits verwendete Programm (Anhang 1.3.1) in abgeänderter Form genutzt.

Der Fetch-Zyklus

Der Fetch-Zyklus ist eine Codesequenz, die aus zwei Mikrobefehlen besteht und somit über zwei Taktzyklen ausgeführt wird. Die Sequenz sorgt dafür, dass der nächste Maschinenbefehl in das Befehlsregister geladen wird, wobei der Fetch-Zyklus immer zu Beginn der Ausführung eines neuen Befehls ausgeführt werden muss (Malvino, 1999, S. 148). Ein neuer Befehl wird genau dann ausgeführt, wenn der Zähler in der Kontrolleinheit zurückgesetzt wird, egal, ob das Befehlsregister leer ist (Programm wurde gerade gestartet) oder ob dort noch der vorherige Befehl gespeichert ist, der gerade ausgeführt wurde. Damit der Fetch-Zyklus stets während der ersten beiden Ausführungsschritte ausgeführt wird, muss der Fetch-Zyklus alle Adressen XXXXXXXX 000 und XXXXXXXX 001 belegen. So wird er ungeachtet des Inhalts des Befehlsregisters immer zu Beginn des Ausführungszyklus ausgeführt.

Die beiden Ausführungsschritte des Fetch-Zyklus sehen folgendermaßen aus:

1. Aus dem Befehlszähler wird die Adresse des nächsten Maschinenbefehls im RAM in das Adressenregister geladen.

⇨ Steuersignale: EP, LM

2. Der Befehl wird aus der Adresse des RAM gelesen und in das Befehlsregister geladen. Außerdem wird der Befehlszähler um eins erhöht, damit eventuelle Operanden ausgelesen werden können oder nach der Ausführung der nächste Befehl geladen werden kann.

⇨ Steuersignale: RR, LI, IP

Jetzt kann der tatsächliche Mikrocode für die Maschinenbefehle in den ROM programmiert werden. Dabei startet die eigentliche individuelle Ausführung eines Befehls erst, wenn der Zähler bei 2 (010) angekommen ist. Nun werden für einen bestimmten Befehl (Beispiel LVA, Opcode 00000001) die ROM-Adressen 00000001 010, 00000001 011, ... individuell mit dem richtigen Mikrocode belegt.

Befehle I: NOOP und Register

NOOP (00000000): Ein Befehl, um eine zeitliche Verzögerung einzubauen. Während der gesamten Ausführung des Befehls wird kein Steuerwort ausgegeben.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Die CPU bleibt im Leerlauf	---
…	…	---
111	Die CPU bleibt im Leerlauf	---

LVA (00000001): Mit Operanden. Lädt den Operanden in das A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für nächsten Befehl. Der Operand wird in das A-Register geladen	IP, RR, LA
100	Zähler zurücksetzen	CC

LMA (00000010): Mit Operanden. Lädt den Inhalt der per Operanden bestimmten RAM-Adresse in das A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für nächsten Befehl. Der Operand wird als Adresse in das Adressenregister geladen	IP, RR, LM
100	Liest den RAM aus und lädt den Wert in das A-Register	RR, LA
101	Zähler zurücksetzen	CC

Die Befehle **LVB** (00000011), **LMB** (00000100), **LVC** (00000101) und **LMC** (00000110) sind analog zu **LVA** und **LMA**, wobei nicht in das A-Register, sondern in das B- bzw. C-Register geladen wird.

Befehle II: Befehle mit ALU Operationen

ADD (00000111): Mit Operanden. Addiert den Operanden zu dem A-Register und speichert das Ergebnis wiederum im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für den für den nächsten Befehl. Der Operand wird in das TMP-Register geladen	IP, RR, LT
100	ALU gibt die Addition aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O1, LA, LF$
101	Zähler zurücksetzen	CC

ADDB (00001000): Addiert das B-Register zum A-Register und speichert das Ergebnis wiederum im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Inhalt des B-Registers in das TMP-Register verschieben	EB, LT
011	ALU gibt die Addition aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O1, LA, LF$
100	Zähler zurücksetzen	CC

Der Befehl **ADDC** (00001001) ist analog zu **ADDB**. Es wird der Inhalt des C-Registers zum A-Register addiert.

Diese Befehlsgruppe **ADD**, **ADDB** und **ADDC** ist außerdem Vorlage für die Maschinenbefehle, die alle weiteren ALU-Operationen abdecken. Dazu gehören die arithmetischen, sowie die logischen Befehle. Der Unterschied besteht nur darin, dass andere ALU Steuersignale gesetzt werden, damit die ALU eine andere Operation ausführt. Die weiteren Befehle sind:

- **SUB** (00001010), **SUBB** (00001011) und **SUBC** (00001100) mit $O2$

⇨ $A = A - (P, B, C)$

- **AND** (00001101), **ANDB** (00001110) und **ANDC** (00001111) mit $O1$, $O2$

⇨ $A = A \wedge (P, B, C)$

- **OR** (00010000), **ORB** (00010001) und **ORC** (00010010) mit $O3$

⇨ $A = A \vee (P, B, C)$

- **XOR** (00010011), **XORB** (00010100) und **XORC** (00010101) mit $O3$, $O2$

⇨ $A = A \underline{\vee} (P, B, C)$

NOT (00010110): Mit Operanden. Negiert den gegebenen Operanden und speichert das Ergebnis im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für den nächsten Befehl. Der Operand wird in das A-Register geladen	IP,RR, LA
100	ALU gibt die Negation aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O3, O1, LA, LF$
101	Zähler zurücksetzen	CC

NOTA (00010111): Negiert den Inhalt des A-Register und speichert das Ergebnis im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	ALU gibt die Negation aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O3, O1, LA, LF$
011	Zähler zurücksetzen	CC

NOTB (00011000): Negiert den Inhalt des B-Registers und speichert das Ergebnis im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Inhalt des B-Registers in das A-Register verschieben	EB, LA
011	ALU gibt die Negation aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O3, O1, LA, LF$
100	Zähler zurücksetzen	CC

Der Befehl **NOTC** (00011001) ist analog zu **NOTB**.

DECAD (00011010): Mit Operanden. Verringert den Wert der per Operanden bestimmten RAM-Adresse um eins und speichert das Ergebnis im A-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für den nächsten Befehl. Der Operand wird als Adresse in das Adressenregister geladen	IP, RR, LM
100	Liest den RAM aus und lädt den Wert in das A-Register	RR, LA
101	ALU führt die DEC Operation aus und speichert das Ergebnis im A-Register. Die Flags für die Operation werden in das Flag-Register geladen	$O3, O2, O1, LA, LF$
110	Zähler zurücksetzen	CC

DEC (00011011), **DECA** (00011100) und **DECB** (00011101) und **DECC** (00011110) sind analog zu einer ALU-Operation mit nur einem Operanden, wie beispielsweise die Gruppe der **NOT** Befehle. **DEC** verringert den gegebenen Parameter, **DECA**, **DECB** und **DECC** den Inhalt des jeweiligen Registers.

Des Weiteren gibt es die Reihe der **INC** Befehle, die auch zu den Befehlen der ALU gehören. Diese sind wiederum analog zu den **DEC** Befehlen, erhöhen aber stattdessen den jeweiligen Wert um eins. Die Befehle sind **INCAD** (00011111), **INC** (00100000), **INCA** (00100001), **INCB** (00100010) und **INCC** (00100011).

Der letzte Maschinenbefehl der ALU Operationen ist der Befehl **AOUT** (00100100). Dieser setzt die Flags im Flag-Register nur für den Inhalt des A-Registers, ohne eine bestimmte Operation durchzuführen.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	ALU führt die AOUT Operation durch. Die Flags werden in das Flag-Register geladen	$O4, O1, LF$
011	Zähler zurücksetzen	CC

Befehle III: Die Ausgabebefehle

OUTA (00100101): Gibt den Inhalt des A-Registers über die Dezimal-Anzeige aus.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Inhalt des A-Registers in das Ausgaberegister laden	EA, LO
011	Zähler zurücksetzen	CC

Analog dazu sind **OUTB** (00100110) und **OUTC** (00100111), um den Inhalt der anderen Register auszugeben.

Befehle IV: Speicherbefehle und Transferbefehle

STA (00101000): Mit Operanden. Der Inhalt des A-Registers wird in der per Operanden gegebenen RAM-Adresse gespeichert.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressregister geladen	EP, LM
011	Befehlszähler erhöhen für den nächsten Befehl. Der Operand wird als Adresse in das Adressregister geladen	IP, RR, LM
100	Der Inhalt des A-Registers wird in den RAM geschrieben	EA, WR
101	Zähler zurücksetzen	CC

Neben **STA** können die Inhalte des B- und C-Registers per **STB** (00101001) und **STC** (00101010) in den RAM geschrieben werden.

Die nächste Befehlsgruppe erlaubt es, die Werte zwischen den einzelnen Registern (A, B und C) zu verschieben.

A2B (00101011): Verschiebt den Inhalt des A-Registers in das B-Register.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Verschiebt den Inhalt des A-Registers in das B-Register	EA, LB
011	Zähler zurücksetzen	CC

Alle weiteren Transferbefehle sind:

- **A2C** (00101100): A-Register → B-Register
- **B2A** (00101101): B-Register → A-Register
- **B2C** (00101110): B-Register → C-Register
- **C2A** (00101111): C-Register → A-Register
- **C2B** (00110000): C-Register → B-Register

Befehle V: Vergleichsbefehle und (bedingte) Sprungbefehle

JMP (00110100): Mit Operanden. Der Sprungbefehl springt innerhalb des Programms während der Ausführung zu einer anderen Stelle, nämlich zu der Adresse, die per Operand bestimmt ist. So können beispielsweise Wiederholungen im Programm realisiert werden, indem man zum Anfang dieser Schleife zurückspringt. Um den Sprungbefehl zu implementieren, wird die Adresse, zu der gesprungen werden soll, in den Befehlszähler geladen.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Adresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Der Operand wird in den Befehlszähler geladen als Adresse des nächsten Befehls	RR, LP
100	Zähler zurücksetzen	CC

Der **JMP** Befehl unterliegt keine Bedingung. Er springt immer zu einer bestimmten Stelle des Programms und eignet sich somit für unendliche Schleifen. Aber wie kann man ein Programm entscheidungsfähig machen und es unter bestimmten Bedingungen unterschiedlich Verzweigen lassen? Dazu werden bedingte Sprungbefehle verwendet. Diese bedingten Sprungbefehle verhalten sich wie der **JMP** Befehl, aber nur, wenn eine bestimmte Kondition gegeben ist. Andernfalls wird das

Programm normal weiter ausgeführt. Diese Konditionen sind die Flags des Flag-Registers. Der **JC** (Jump Carry; 00110101) Befehl springt nur unter der Bedingung, dass die Carry-Flag gesetzt ist, für **JZ** (Jump Zero; 00110110) muss die Zero-Flag gesetzt sein und für **JN** (Jump Negative; 00110111) muss die Sign-Flag gesetzt sein (Malvino, 1999, S. 179). Das bedeutet, dass die Kontrolleinheit je nach gesetzten Flags für diese Befehle unterschiedliche Steuerworte ausgeben muss. Damit dies erreicht wird, wird die Adresse der Kontroll-ROM um die drei Flags erweitert (Anhang 1.6.8).

14 Bit Adresse der ROM der Kontrolleinheit

Die Adresse: 000 00000000 000
Grün: 3-Bit Flag-Register. Carry-Flag, Zero-Flag und Sign-Flag
Gelb: 8-Bit Befehlsregister. Erlaubt somit maximal $2^8 = 256$ verschiedene Maschinenbefehle
Blau: 3-Bit Taktzähler. Ein Maschinenbefehl wird über maximal $2^3 = 8$ Taktzyklen ausgeführt

Der Mikrocode aller Befehle, die nicht von den Flags abhängig sind, wird für alle möglichen Kombinationen der Flags in den ROM programmiert, damit unabhängig von den gesetzten Flags immer der gleiche Mikrocode ausgeführt wird. Für die bedingten Sprungbefehle hingegen wird abhängig von den Flags unterschiedlicher Mikrocode in den ROM programmiert (Eater, 2018, „Conditional jump instructions“).

JC (00110101): Mit Operanden. Springt, wenn mindestens die Carry-Flag gesetzt ist. Falls dies der Fall ist, wird der gleiche Mikrocode wie für **JMP** ausgeführt. Andernfalls wird folgender Mikrocode ausgeführt:

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Befehlszähler um eins erhöhen, damit der Operand übersprungen wird	IP
011	Zähler zurücksetzen	CC

Die beiden anderen bedingten Sprungbefehle werden analog zu **JC** in den ROM programmiert, nur unter der Bedingung einer anderen Flag.

Um zwei Zahlen miteinander zu vergleichen, subtrahiert man sie voneinander. Will man A mit B vergleichen, so schaut man sich die Flags an, die gesetzt werden, wenn die Operation $A - B$ ausgeführt wird. Für $A < B$ ist die Sign-Flag gesetzt, für $A = B$ ist die Zero-Flag gesetzt und ist keine der beiden Flags gesetzt, so ist $A > B$. Da man aber nicht immer bei einer Vergleichsoperation den Inhalt des A-Registers überschreiben will, wie es bei dem **SUB** Befehl der Fall wäre, gibt es die Befehlsgruppe **CMP**. Diese ist vollkommen gleich zu den **SUB** Befehlen bis auf den Unterschied, dass das Ergebnis der ALU Operation nicht im A-Register gespeichert wird, sondern lediglich das Flag-Register aktualisiert wird. Die Vergleichsbefehle sind **CMP** (00110001), **CMPB** (00110010) und **CMPC** (00110011)

Befehle VI: Unterprogramme, CALL, RET und HLT

Als Unterprogramm bezeichnet man einen Teil eines Programms, das von diesem unabhängig ausgeführt werden kann, indem es von Teilen des Programms oder auch anderen Programmen aufgerufen wird. Dabei stellt es meist eine oft genutzte Funktion oder einen Programmabschnitt bereit. Nach der Beendigung des Unterprogramms kehrt das Programm automatisch zu der Stelle zurück, von der das Unterprogramm aufgerufen wurde. Ein Beispiel für ein Unterprogramm dieser CPU könnte eine Funktion zum Multiplizieren zweier Werte sein, die spontan aufgerufen werden kann, zwei Werte multipliziert, das Ergebnis speichert und schließlich zum eigentlichen Programm zurückkehrt. Von dort aus kann das Ergebnis des Unterprogramms verwendet werden. Den Befehl,

der ein Unterprogramm aufruft und dorthin verzweigt, nennt man **CALL**. Der Befehl, der von dort aus zum Hauptprogramm zurückkehrt, wird **RETURN**, kurz **RET** genannt. Der Datenaustausch zwischen Haupt- und Unterprogramm kann unterschiedlich realisiert werden. Am einfachsten ist dabei die Übergabe über die Register der CPU oder einen festgelegten Speicherort im RAM (Wüst, 2003, S. 81f).

CALL (11111111): Mit Operanden. Ruft ein Unterprogramm beginnend an der per Operanden mitgegebenen Adresse auf. Die Adresse des nächsten Befehls im Hauptprogramm, von dem die Unterfunktion aufgerufen wurde, wird in der RAM-Adresse 11111111 gespeichert. Damit dies möglich ist, wird der Opcode des Befehls als Adresse in das Adressenregister des RAM geladen. Um eine Verwendung des Opcodes während er weiteren Ausführung eines Maschinenbefehls zu ermöglichen, muss jedoch der Fetch-Zyklus minimal verändert werden: Während Schritt zwei wird der Opcode nicht nur in das Befehlsregister, sondern auch in das TMP-Register für die spätere Verwendung geladen. Dies beeinflusst an keiner Stelle die Ausführung eines Befehls.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Dresse des Operanden wird aus dem Befehlszähler in das Adressenregister geladen	EP, LM
011	Befehlszähler erhöhen für nächsten Befehl. Der Operand (der Beginn des Unterprogramms) wird im A-Register zwischengespeichert	IP, RR, LA
100	Der Opcode des Befehls wird aus dem TMP-Register in das Adressenregister geladen	ET, LM
101	Die Adresse des nächsten Befehls im Hauptprogramm wird in der RAM-Adresse 11111111 gespeichert	EP, WR
110	Die Adresse des Beginns des Unterprogramms wird aus dem A-Register in den Befehlszähler geladen	EA, LP
111	Zähler zurücksetzen	CC

RET (11111110): Kehrt vom Unterprogramm zum Hauptprogramm zurück, indem die Adresse des nächsten Befehls aus dem RAM ausgelesen wird. Auch hier wird wieder der Opcode als Adresse verwendet, muss aber vorher um eins erhöht werden, damit aus der Adresse 11111111 ausgelesen wird.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Lädt den Opcode in das A-Register	ET, LA
011	Die ALU-Operation erhöht den Inhalt des A-Registers um eins und speichert diesen direkt im Adressenregister	$O4, LM$
100	Die Adresse des nächsten Befehls des Hauptprogramms wird aus dem RAM ausgelesen und im Befehlszähler gespeichert	RR, LP
101	Zähler zurücksetzen	CC

So wie der **RET** Befehl ein Unterprogramm beendet, so beendet der **HLT** Befehl das Hauptprogramm, indem das HLT Signal gesetzt wird und somit der Taktgeber angehalten wird.

Zähler	Beschreibung Mikrobefehl	Steuersignale
010	Setzt das HLT Signal und stoppt den Taktgeber	HLT

Die Gesamtheit aller Befehle, die eine CPU versteht, wird als Befehlsset bezeichnet. Eine Übersicht über das Befehlsset dieser 8-Bit CPU findet sich im Anhang 1.6.9. Das Befehlsset der 8-Bit CPU ist eine Zusammenstellung aus Befehlen beschrieben von Malvino (1999), Wüst (2003) und Eater (2017/2018). Zusätzlich wurden eigene Maschinenbefehle passend zur Architektur dieser CPU

hinzugefügt. Die Implementierung der Maschinenbefehle und des Mikrocodes ist hauptsächlich selbst entworfen oder von Eater übernommen.

Programmierung der CPU

Nach der Fertigstellung der CPU werden im Folgenden die Möglichkeiten der 8-Bit CPU demonstriert. Dazu habe ich simple Programme geschrieben, die verschiedene Maschinenbefehle verwenden. Die Programme sind in einer assemblerähnlichen Sprache notiert und müssen zur Ausführung durch die CPU mithilfe des Befehlssets in Opcode übersetzt werden. Alle Programme sind voll funktionsfähig und erfüllen ihre Aufgabe.

Zweierpotenzen

Das Programm:

```
LVA 0x02
#LOOP
OUTA
A2B
ADDB
JMP #LOOP
```

Zunächst wird der Startwert „2“ in das A-Register geladen. „#LOOP“ bezeichnet man als ein Label. Sprungbefehle können auf ein Label verweisen. So muss in dieser anschaulichen Darstellung des Programms nicht eine Adresse hinter einem Sprungbefehl angegeben werden, sondern man sieht direkt, wo der Sprungbefehl hinspringt, nämlich zum Befehl nach dem Label. Erst beim Übersetzen in den Opcode wird das Label entfernt und durch die Sprungadresse ersetzt. Nach dem Laden des Startwerts wird der Inhalt des A-Registers in das B-Register kopiert. So befinden sich in A- und B-Register die gleiche Zahl. Nach deren Addition wird das Ergebnis $A_1 = A_0 + B = A_0 + A_0 = 2 * A_0 = 2^2$ mit $A_n = B$ (dank A2B) und $A_0 = 2$ in das A-Register geladen und außerdem auf der Dezimalanzeige ausgegeben. Der Sprungbefehl springt zurück zum Anfang und wieder wird $A_2 = A_1 + B = A_1 + A_1 = 2 * A_1 = 2 * 2^2 = 2^3$ ausgeführt. So wird die Reihe der Zweierpotenzen ausgerechnet, indem durch die Addition des gleichen Wertes in B immer wieder mit 2 multipliziert wird.

Fibonacci-Folge

Das Programm:

```
LVA 0x01
#LOOP
ADDB
JC #HLT
OUTA
C2B
A2C
JMP #LOOP
#HLT
HLT
```

Die Fibonacci-Folge ist bestimmt durch $f_n = f_{n-1} + f_{n-2}$ mit $f_0 = 0$ und $f_1 = 1$. Vor dem Befehl ADDB hält das A-Register den Wert f_{n-1}, da das Ergebnis der vorherigen Addition durch ADDB wiederum im A-Register gespeichert wurde. Das B-Register stellt f_{n-2} bereit. Dies ist wird realisiert, indem f_n noch im selben Schleifendurchlauf wie die Addition vom A-Register in das C-Register vershoben wird. Im nächsten Schleifendurchlauf ist dies nun f_{n-1}, was dann vom C-Register in das B-Register verschoben wird. Wiederum im nächsten Schleifendurchlauf wird dieser Wert aus dem B-Register für die Addition verwendet und stellt in diesem Schleifendurchlauf nun f_{n-2} dar. Mit LVA 0x01 werden die Register initialisiert mit den Startwerten der Folge. Über JC #HLT wird überprüft, ob ein Übertrag bei der Addition auftritt. Falls also das Ergebnis der Addition über 255 ist, springt das Programm an sein Ende und stoppt per HLT die CPU. Die größte Zahl der Fibonacci-Folge unter 255 ist 233. Sobald diese Zahl auf dem Display erscheint, stoppt die Ausführung des Programms.

Bedingte Anweisungen (IF, ELSE IF und ELSE Ausdruck):

Dieses Programm vergleicht den Inhalt des A- und des B-Registers. Für $A < B$ wird „1“ ausgegeben, für $A = B$ eine „2“ und für $A > B$ eine „3“.

Zunächst werden die beiden Zahlen, die verglichen werden sollen, in das A- und das B-Register geladen. Dann wird der CMPB Befehl ausgeführt, sodass die Flags für $A - B$ gesetzt werden. Falls $A = B$, wird ist die Zero-Flag gesetzt und das Programm springt zum Label #Zero. Dort wird entsprechend eine „2“ in das A-Register geladen und dann zum Ende des Programms gesprungen. Falls die Zero-Flag nicht gesetzt wurde, wird auf die Sign-Flag überprüft. Falls diese gesetzt ist, ist $A < B$ und das Programm springt zum #Neg Label. In diesem Abschnitt wird eine „1“ in das A-Register geladen. Falls keine der beiden Flags gesetzt wurden, werden beide bedingten Sprungbefehle übersprungen und das Programm lädt eine „3“ in das A-Register. Danach springt es auch zum Ende des Programms. Die tatsächliche bedingte Anweisung befindet sich also nur im mittleren Teil des Programms. Zuletzt wird bei #END der Inhalt des A-Registers ausgegeben und die CPU gestoppt.

Das Programm:

```
LVA 0x05
LVB 0x05
CMPB

JZ #Zero
JN #Neg
LVA 0x03
JMP #END
#Zero
LVA 0x02
JMP #END
#Neg
LVA 0x01

#END
OUTA
HLT
```

Ein Unterprogramm zum Multiplizieren

Das Unterprogramm „Mult“ multipliziert zwei Zahlen, gespeichert in den RAM-Adressen 0xfe und 0xfd, miteinander. Das Ergebnis wird wiederum in der Adresse 0xfe gespeichert. Das Hauptprogramm ruft das Unterprogramm auf und lädt nach seiner Ausführung das Ergebnis in das A-Register, um es anzuzeigen. Zum Schluss wird die CPU angehalten.

Das Unterprogramm verwendet zur Multiplikation wiederholte Addition. Zunächst wird das C-Register gelöscht. Dann wird der Operand aus 0xfe in das B-Register geladen. Während der wiederholten Multiplikation wird der Operand 0xfd im C-Register zwischengespeichert und zur Addition verwendet. Währenddessen wird der Inhalt der RAM-Adresse dekrementiert, damit die Schleife unterbrochen werden kann, wenn die Multiplikation abgeschlossen ist. Zu Beginn der Schleife wird 0xfd dekrementiert und wieder im RAM gespeichert. Falls dabei die Sign-Flag gesetzt wurde, wird die Multiplikation beendet. Mit den nächsten drei Befehlen wird das C-Register zu B addiert und wieder im C-Register gespeichert. Während sich im B-Register immer die gleiche Zahl befindet, die auf C addiert wird, wird im C-Register das Zwischenergebnis gespeichert, beginnend bei 0. Durch das anfängliche Löschen des C-Registers, das Dekrementieren der Adresse 0xfd vor der Addition und das Überprüfen auf die Sign-Flag unterstützt das Unterprogramm auch das Multiplizieren mit Null.

Das Programm:

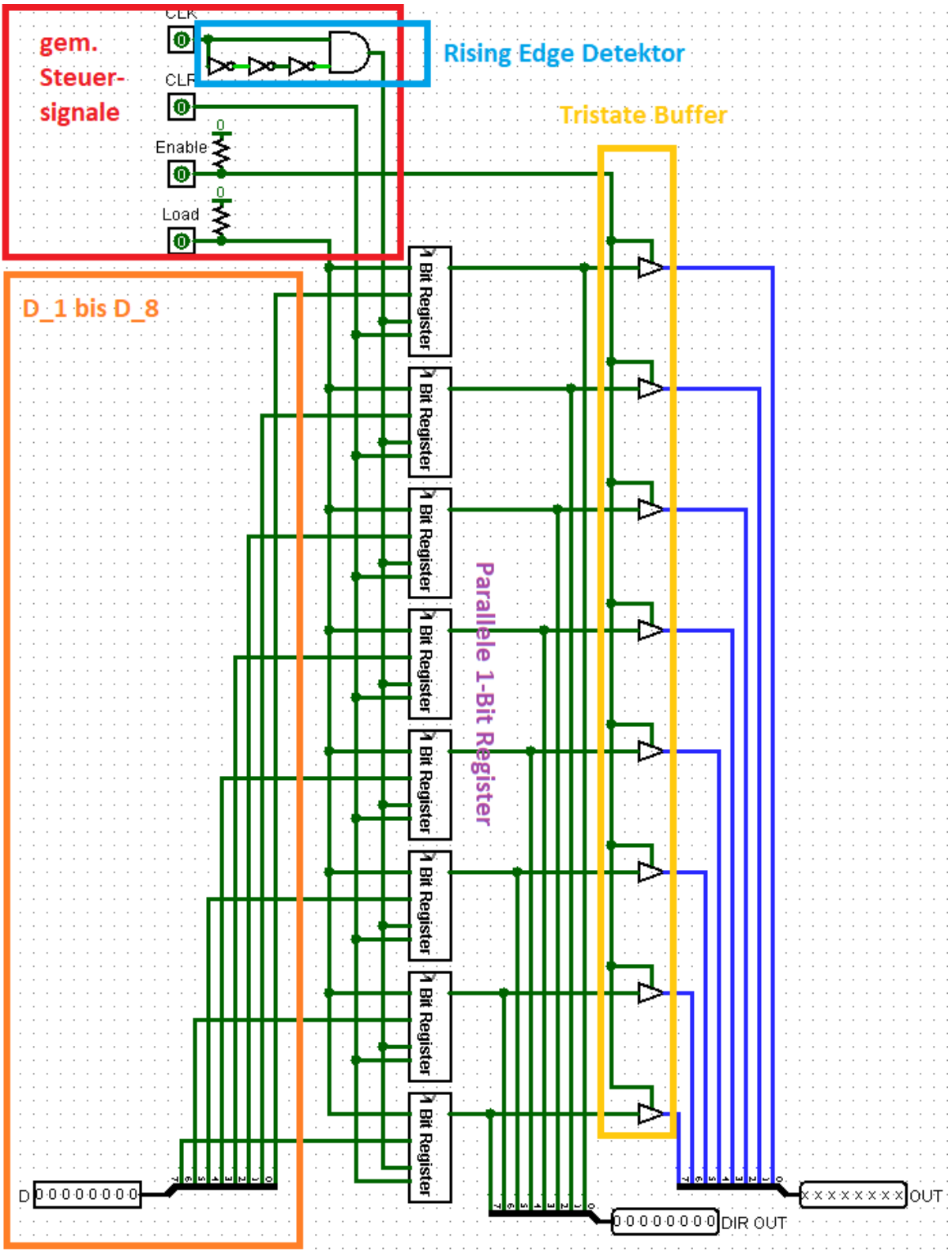
```
CALL #Mult
LMA 0xfe
OUTA
HLT

#Mult
LVC 0x00
LMB 0xfe
#MultLoop
DECAD 0xfd
STA 0xfd
JN #EndMult
C2A
ADDB
A2C
JMP #MultLoop
#EndMult
STC 0xfe
RET
```

Anhangsverzeichnis

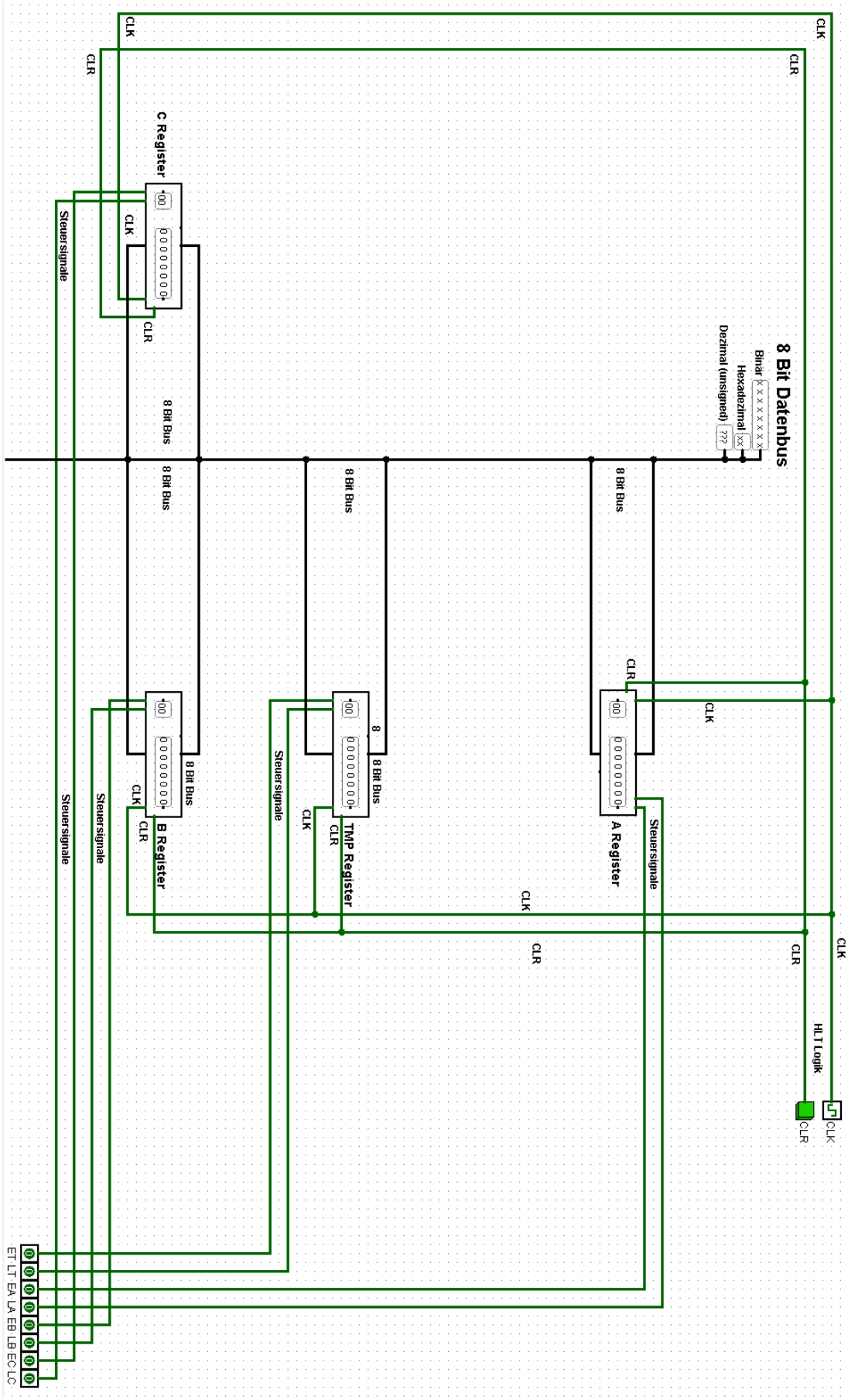
Anhang 1: Bau der CPU

Anhang 1.1: Register und Bussystem

Anhang 1.1.1: Fertiges 8 Bit Register

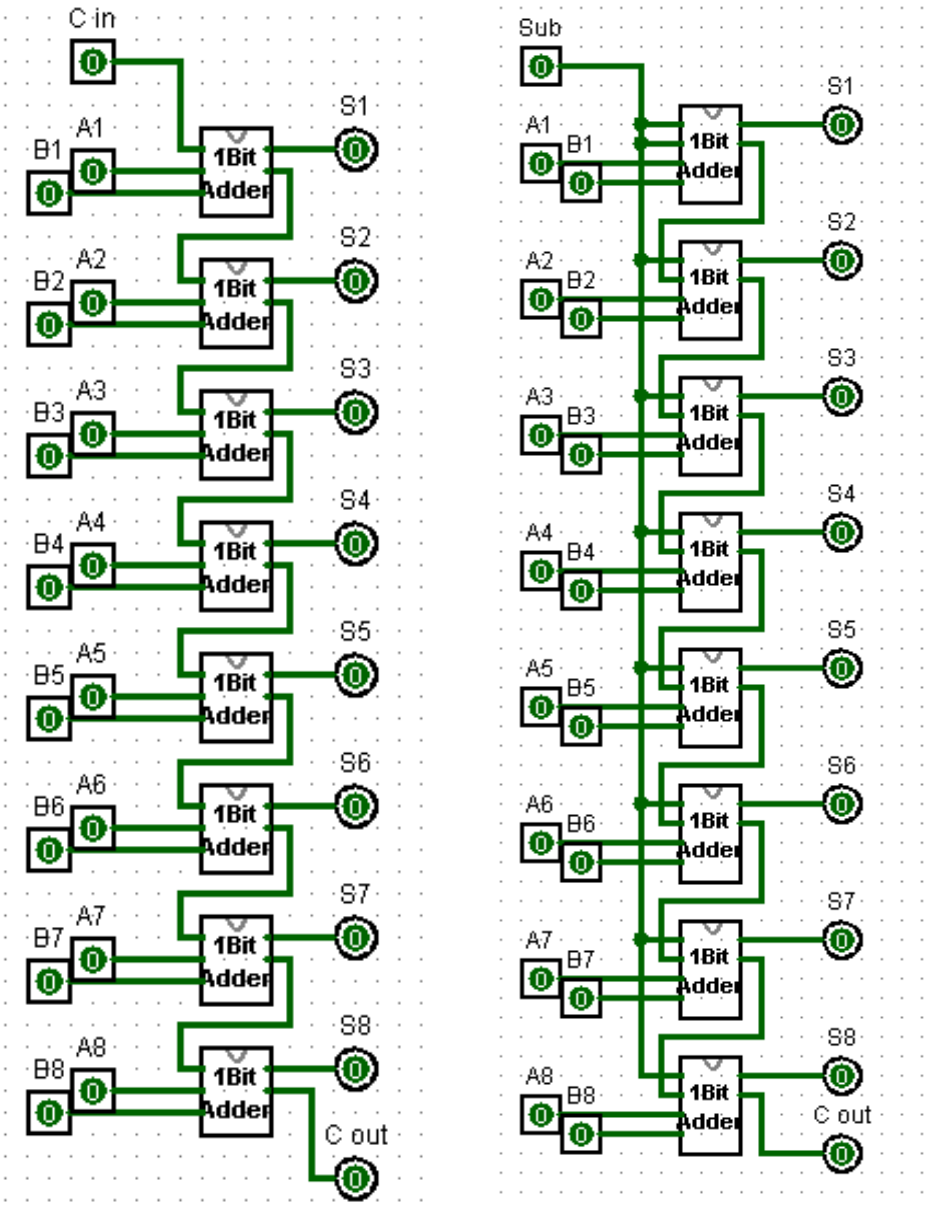


Anhang 1.1.2 Bussystem Nr. 1

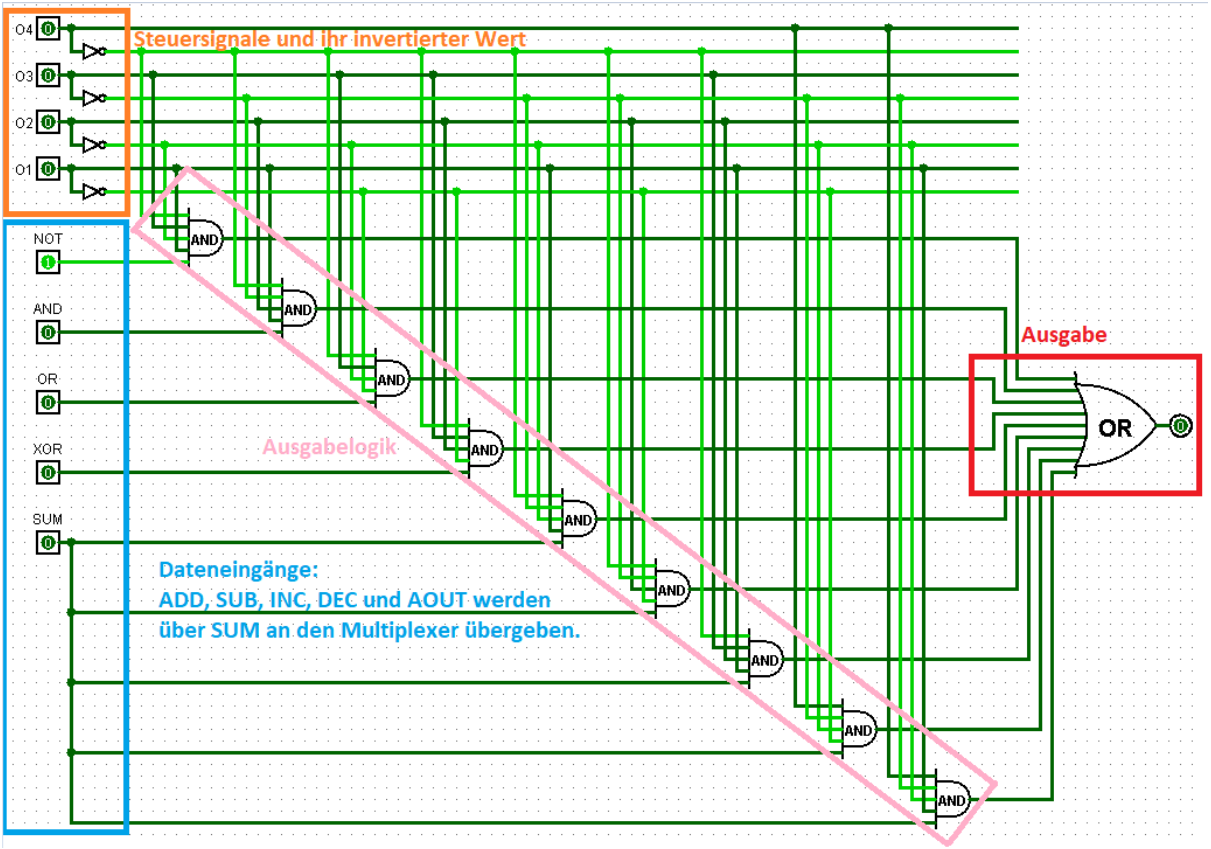


Anhang 1.2: Die Arithmetisch Logische Einheit

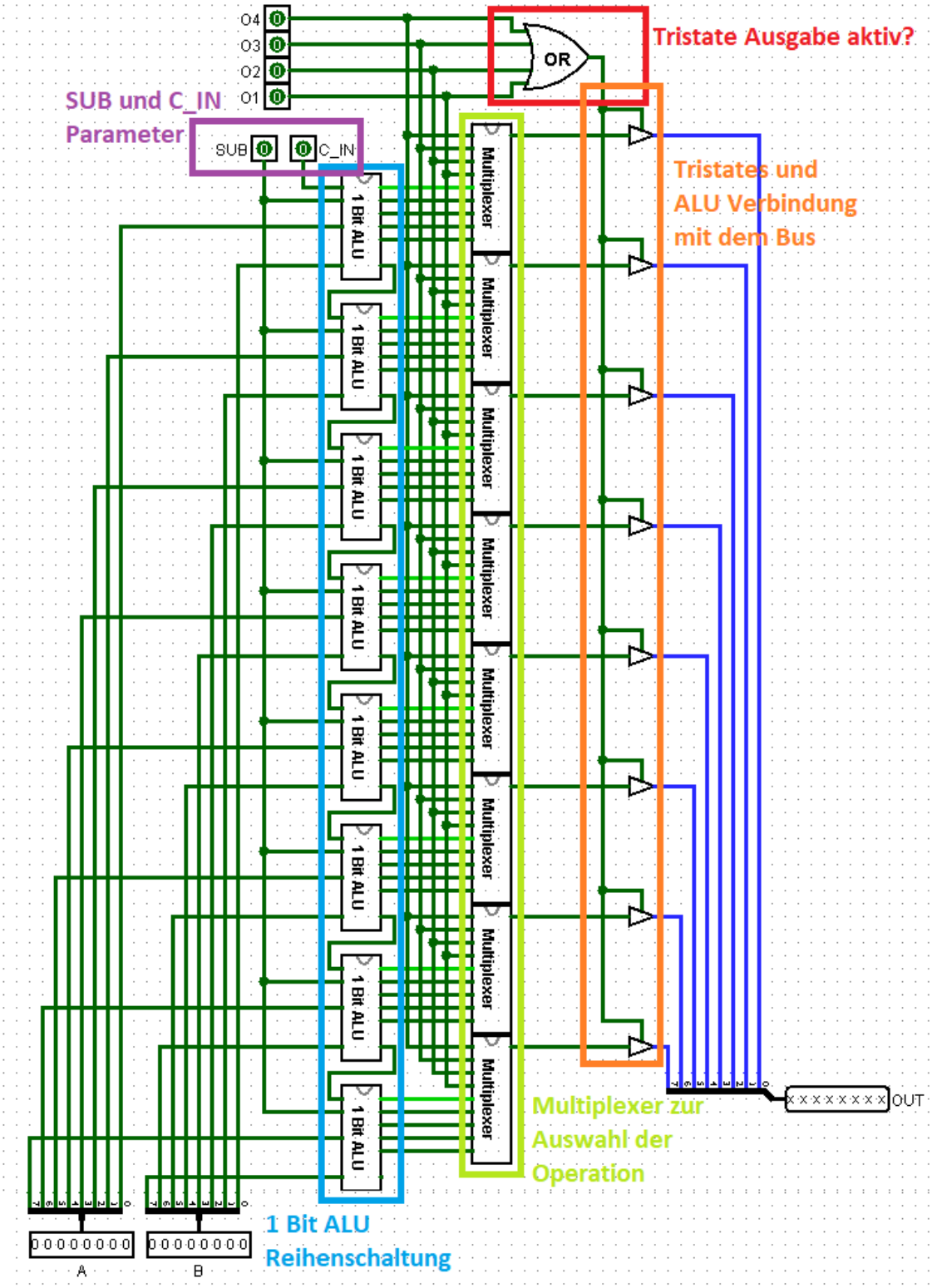
Anhang 1.2.1: 8-Bit Volladdierer und arithmetische Einheit der CPU



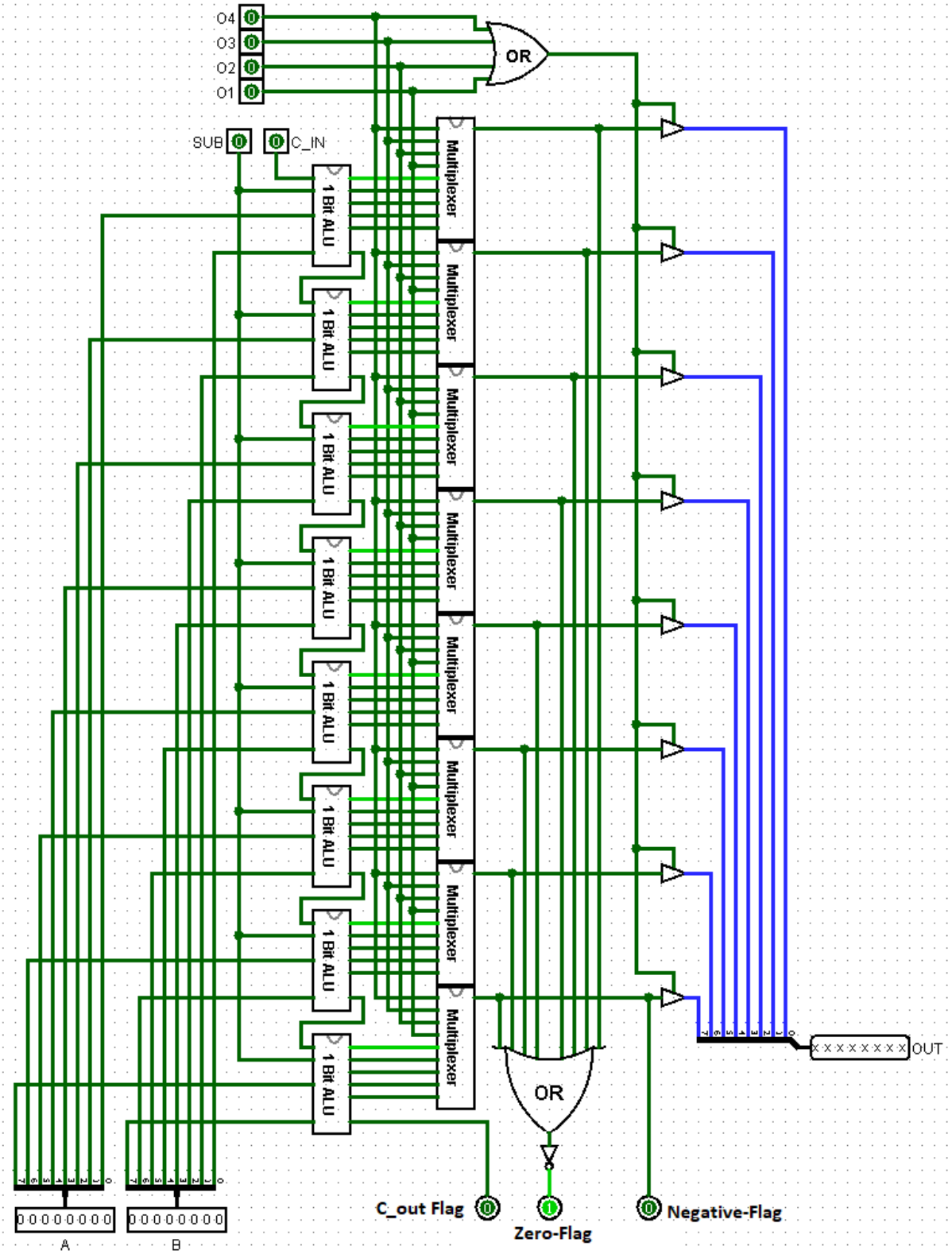
Anhang 1.2.2: Multiplexer der ALU



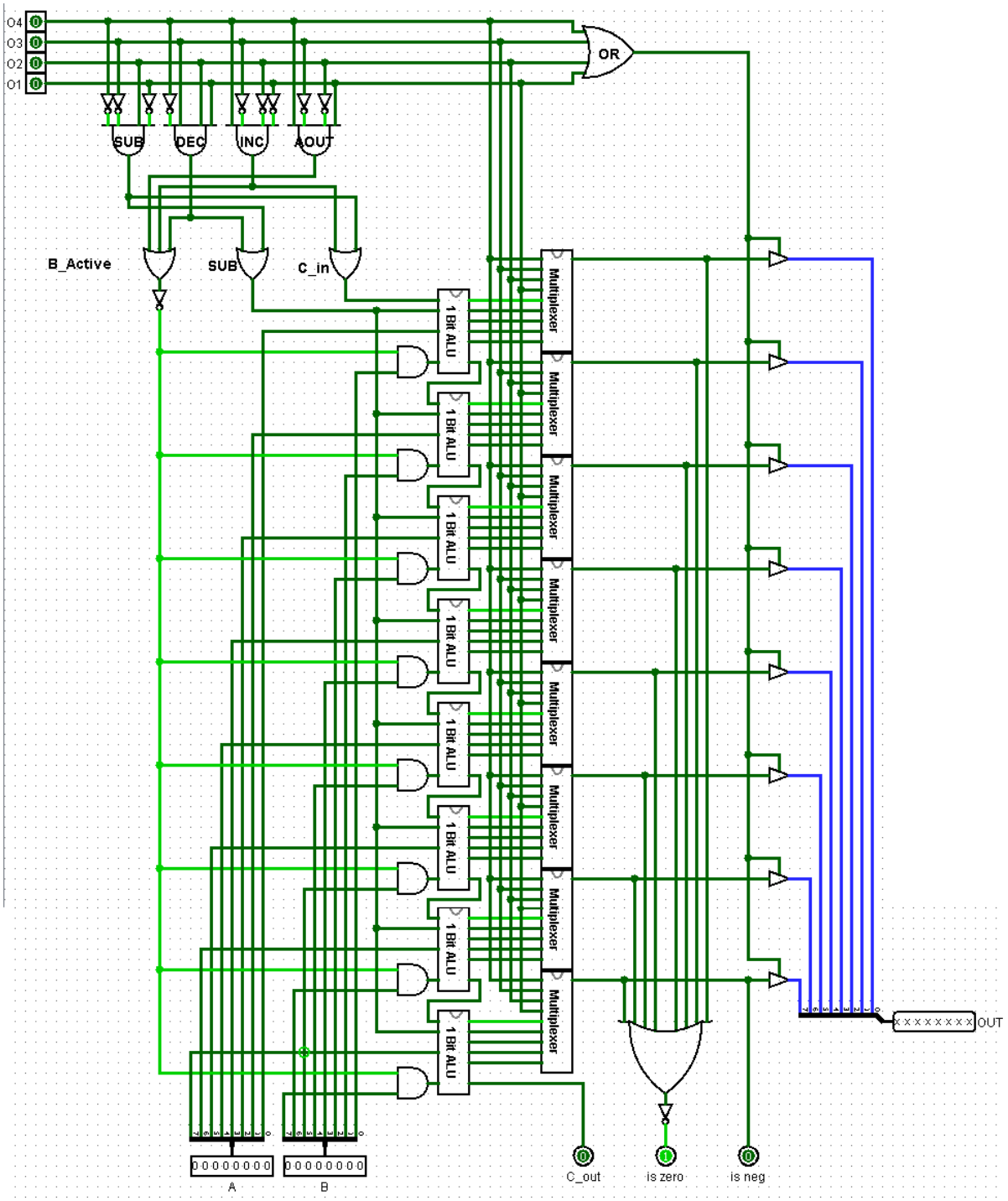
Anhang 1.2.3: 8 Bit ALU mit Multiplexern und Tristates



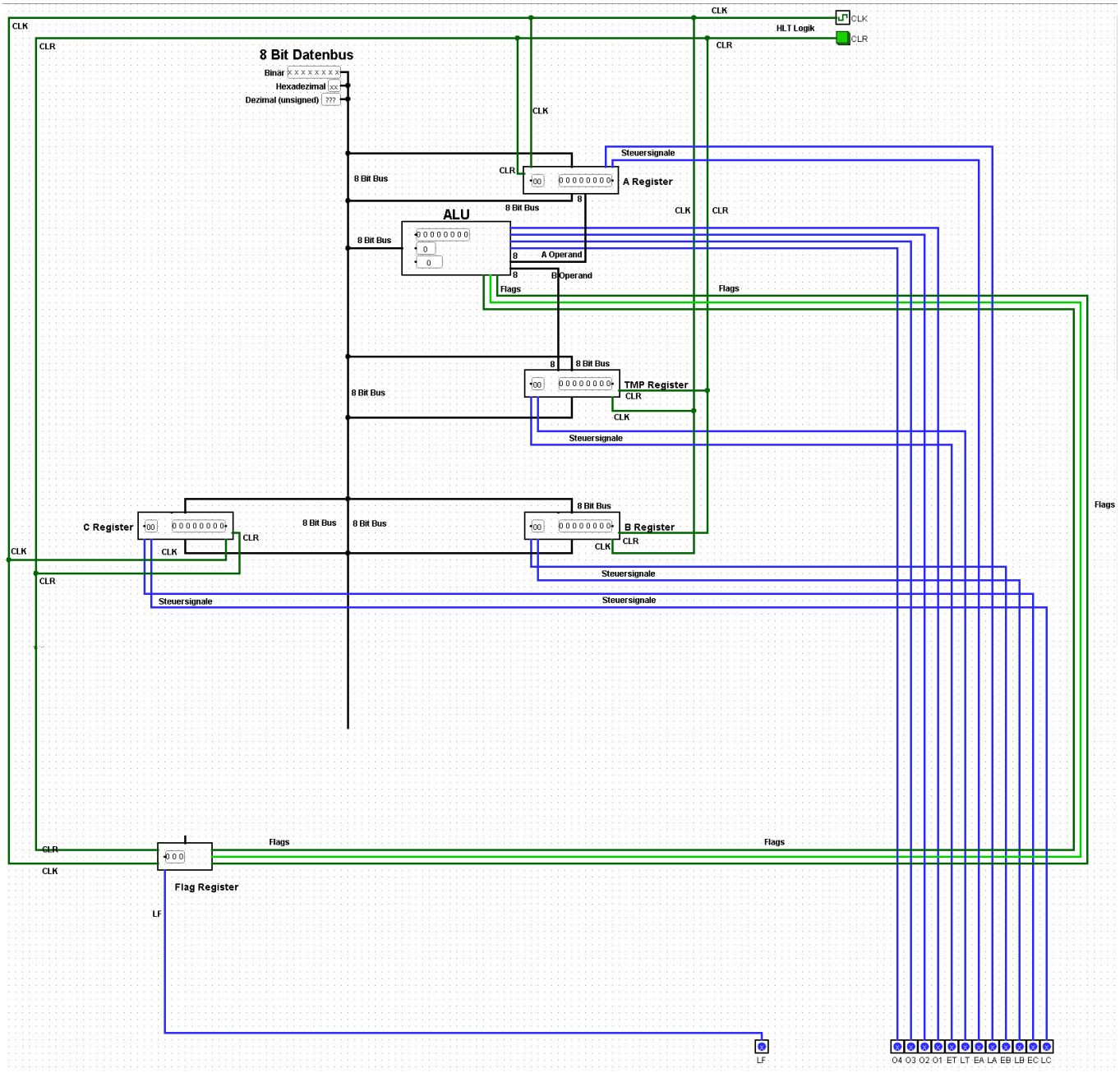
Anhang 1.2.4: 8 Bit ALU mit Logik zur Bestimmung der Flags



Anhang 1.2.5: Fertige 8-Bit ALU



Anhang 1.2.6: ALU und Flag-Register in der CPU



Anhang 1.3: Das Ausgaberegister

Anhang 1.3.1: Programm zum Beschreiben einer Logisim ROM

Alle Programme sind in Python geschrieben und benötigen keine Bibliotheken, die nicht in der Standardbibliothek von Python vorhanden sind.

```
# data: int base 10 array
def write_content(data):
    content = "v2.0 raw\n"
    for d in data:
        content += hex(d).replace("0x", "") + " "
    rom = open("rom", "w+")
    rom.write(content)
    rom.close()


write_content([0xff for _ in range(256)])
```

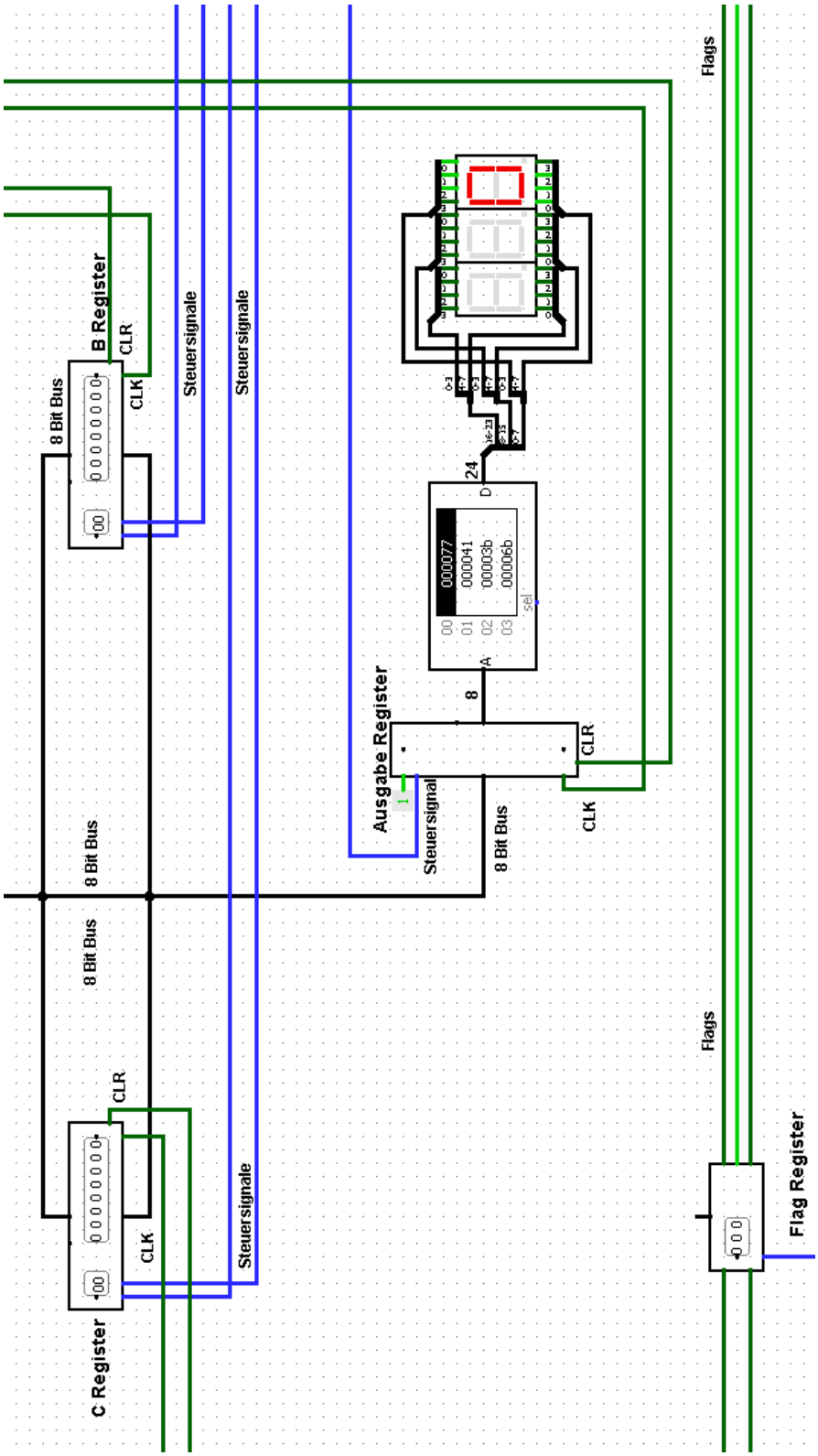
Dieses Programm beinhaltet eine Funktion „write_content“, die ein Array, das den Inhalt der ROM enthält, in eine Datei schreibt im Format, das Logisim lesen kann und damit eine Logisim ROM beschreibt. Das Array hat die Länge der Anzahl an Adressen der Rom und beinhaltet für jede Adresse das entsprechende Steuerwort.

Anhang 1.3.2: Programm zum Erstellen der ROM für die Dezimalanzeige

```
from rom_writer import write_content

disp_decode = {"0": 0b01110111, "1": 0b01000001, "2": 0b00111011,
               "3": 0b01101011, "4": 0b01001101, "5": 0b01101110,
               "6": 0b01111110, "7": 0b01000011, "8": 0b01111111,
               "9": 0b01101111, " ": 0b00000000}
d = []

for n in range(256):
    n = ("  " + str(n))[::-1]
    d.append(
        (disp_decode.get(n[2]) << 16)
        | (disp_decode.get(n[1]) << 8)
        | (disp_decode.get(n[0]))
    )

write_content(d)
```

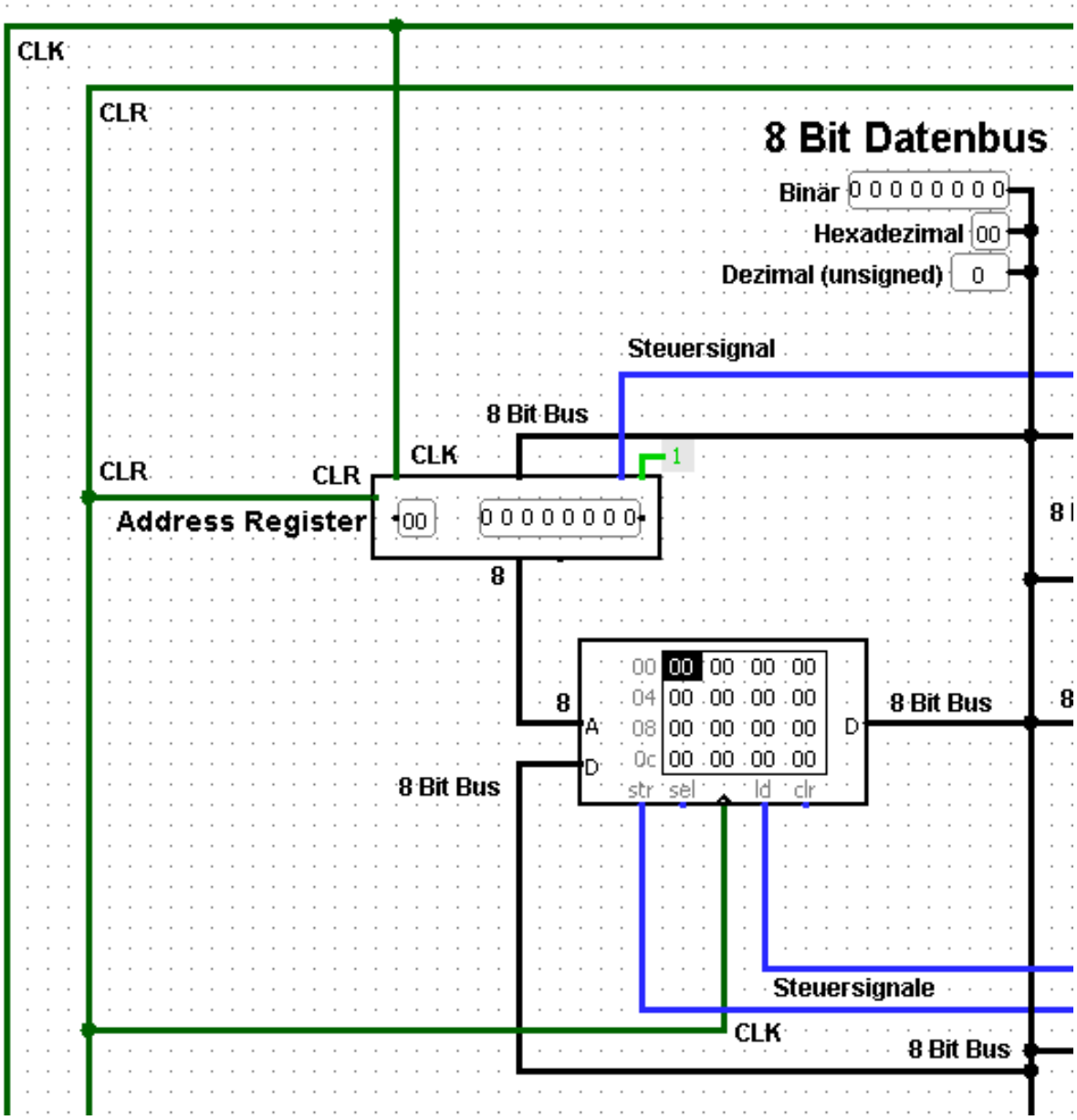
Dieses Programm erstellt den ROM-Inhalt für die Dezimal-Anzeige. Für jede einzelne Ziffer ist das passende Steuerwort hinterlegt, um ein 7-Segment Display anzusteuern. Für jede Zahl von 0 bis 255 werden die passenden drei Ziffern hintereinander gereiht, um so das 24 Bit Steuerwort für die drei 7-Segment Displays zu bilden. Am Ende wird die „write_content“ Funktion aufgerufen, die den ROM Inhalt speichert. Diese Datei wird in die Logisim ROM geladen.

Anhang 1.3.3: Das Ausgaberegister und die Dezimalanzeige in der CPU



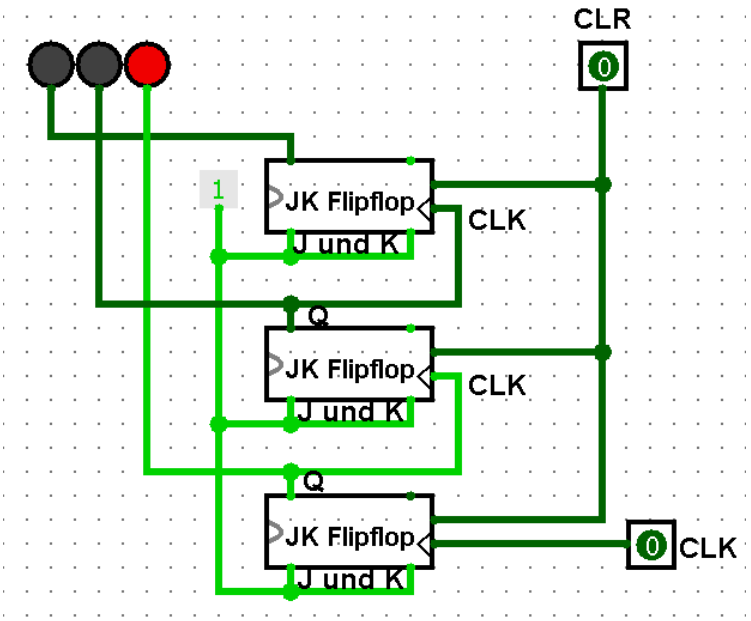
Anhang 1.4: Der Arbeitsspeicher (RAM)

Anhang 1.4.1 Das RAM Modul in der CPU integriert

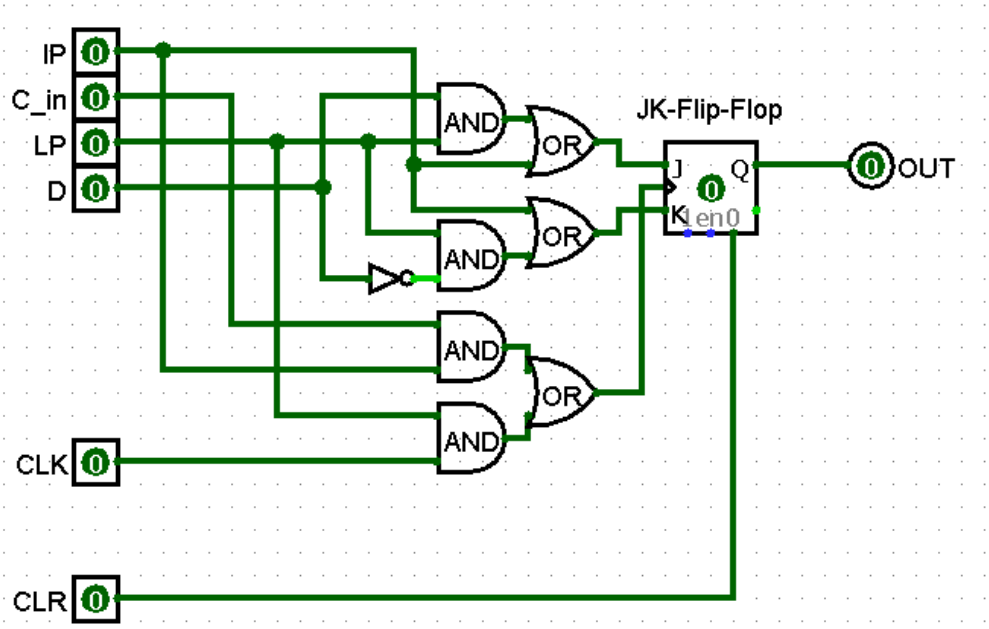


Anhang 1.5: Der Befehlszähler (Pointer)

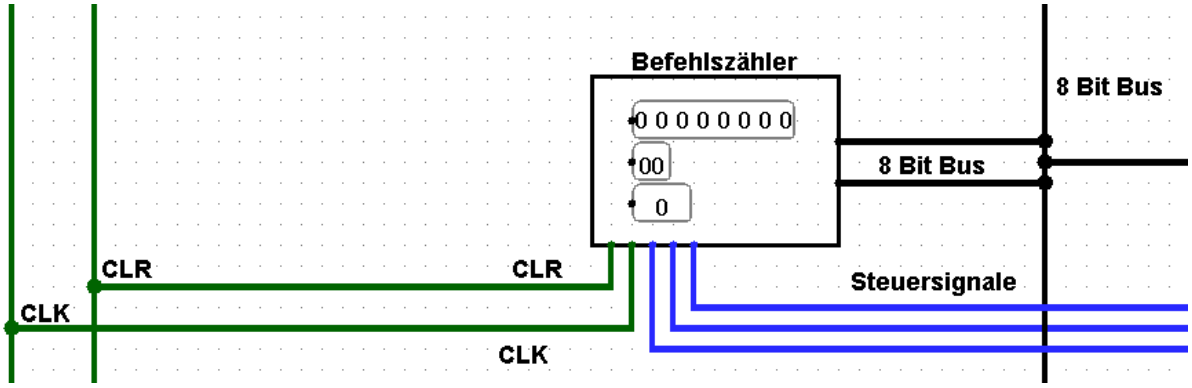
Anhang 1.5.1: Einfacher 3 Bit Binärzähler



Anhang 1.5.2: 1 Bit Befehlszähler

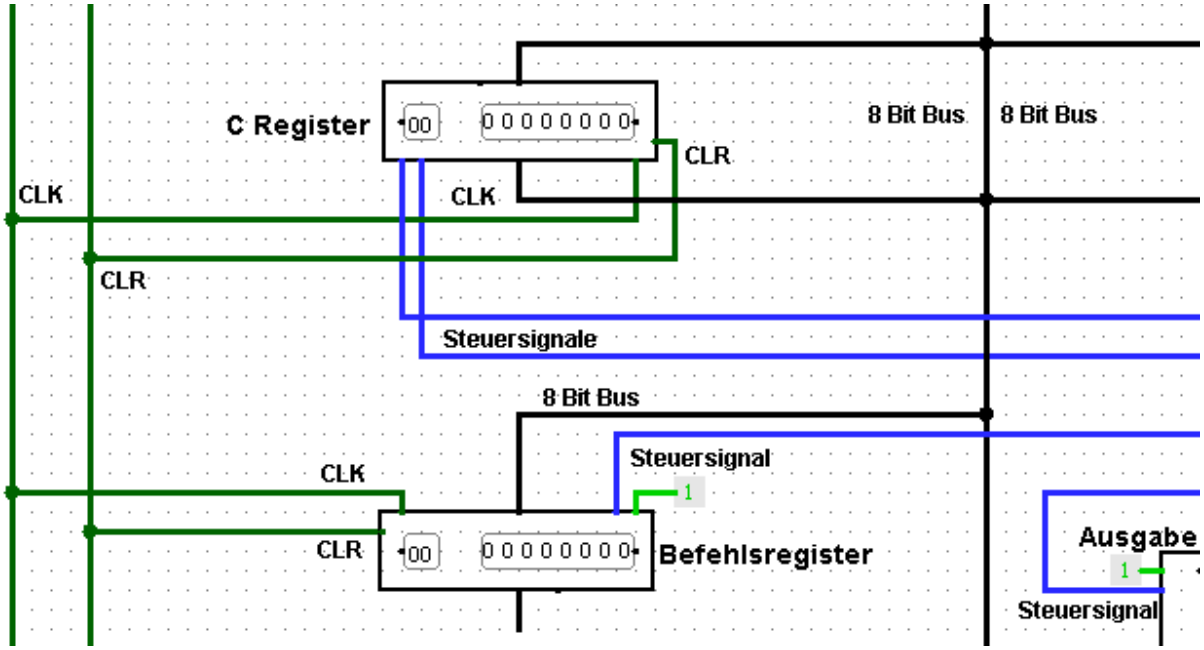


Anhang 1.5.3: Der Befehlszähler in der CPU

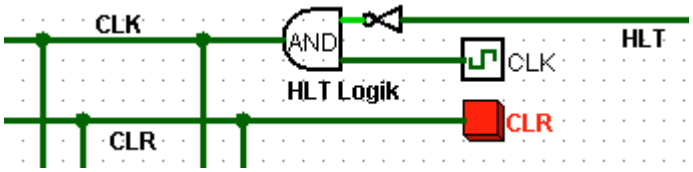


Anhang 1.6: Die Kontrolleinheit

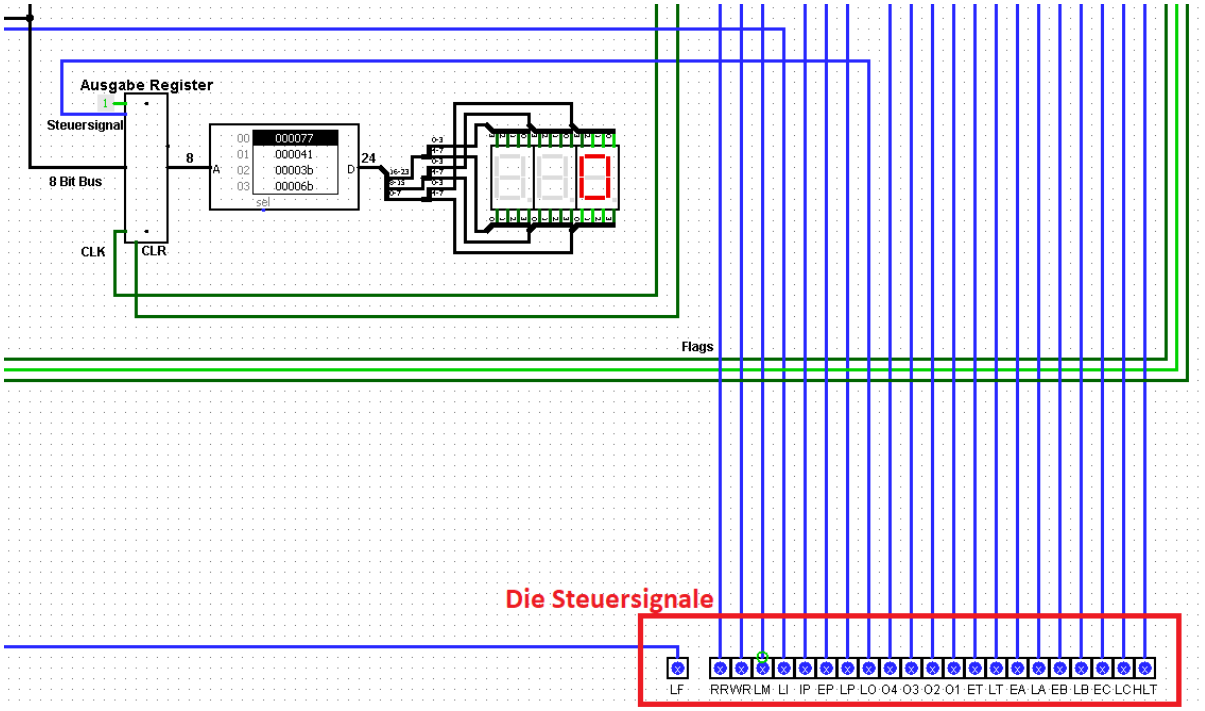
Anhang 1.6.1: Das Befehlsregister



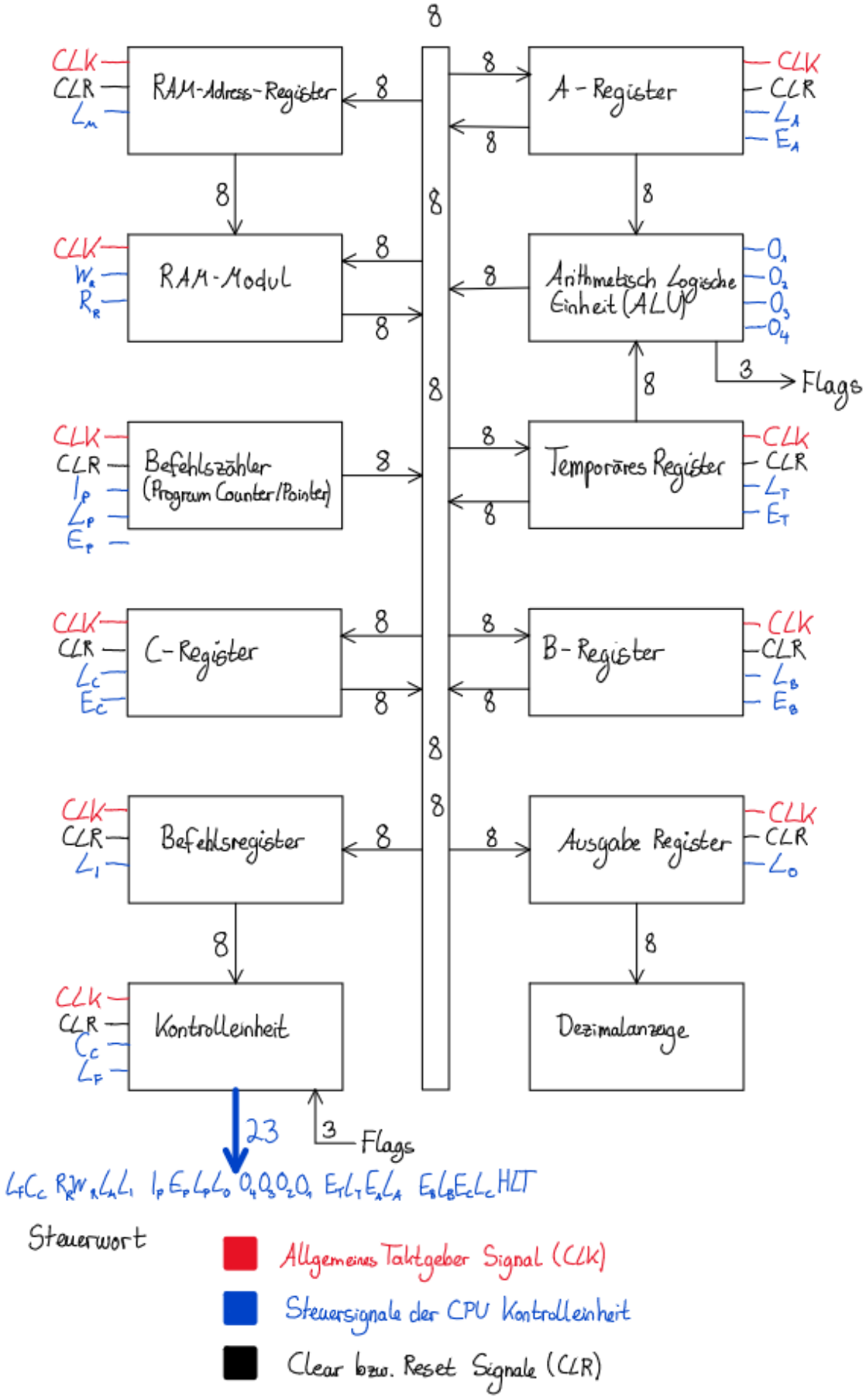
Anhang 1.6.2: Das HLT Signal



Anhang 1.6.3: Die Steuersignale



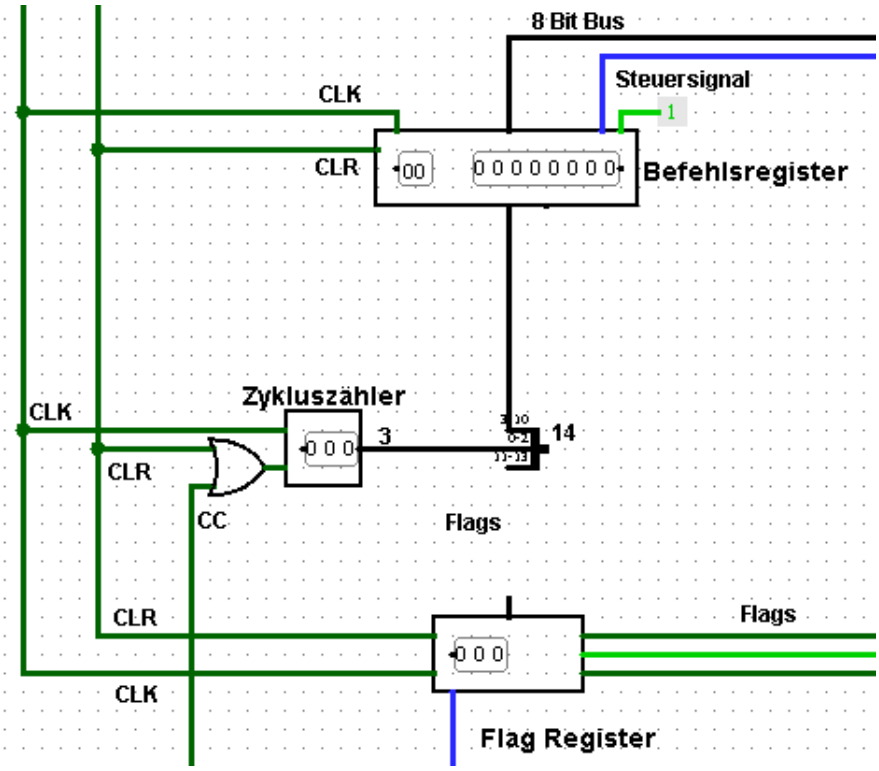
Anhang 1.6.4: Übersicht SAP-Architektur und Steuersignale



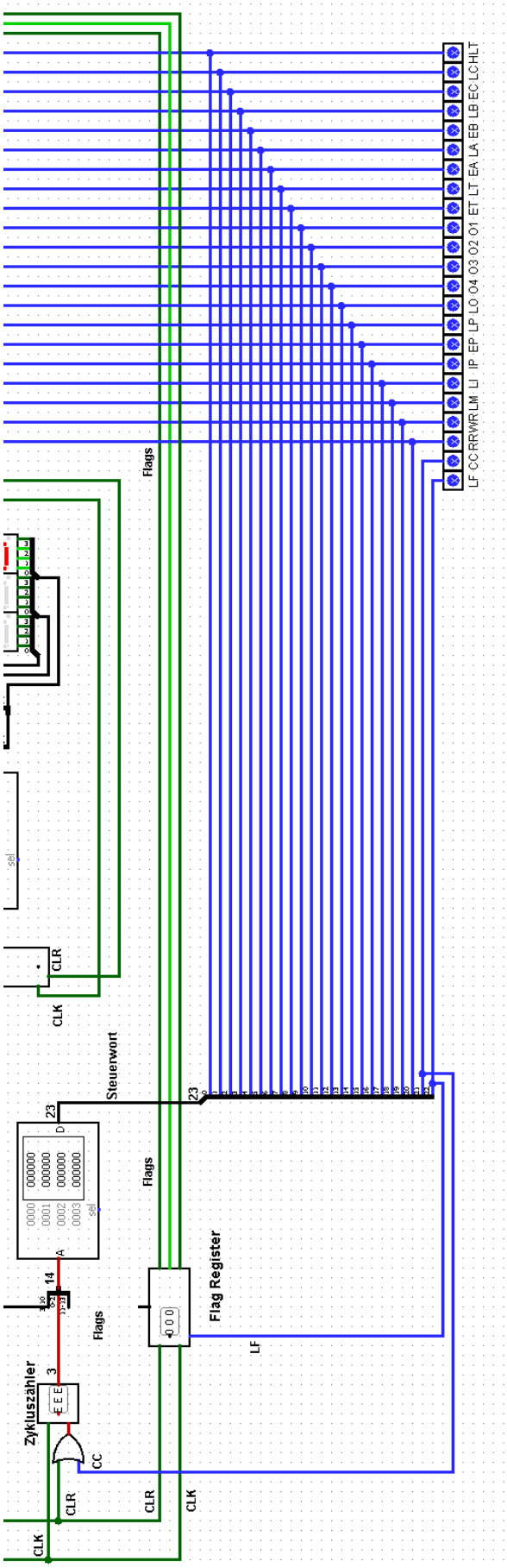
Anhang 1.6.5 Erklärung der Steuersignale

LF	Load Flag Register	Speichert die aktuell in der ALU gesetzten Steuerflags in der Kontrolleinheit
CC	Cyle Clear	Setzt den Mikrobefehlszähler in der Kontrolleinheit auf 0 zurück, damit der nächste Befehl ausgeführt werden kann
RR	Read RAM	Liest den Inhalt der im MAR gesetzten Adresse aus und überträgt deren Inhalt auf den Bus
WR	Write RAM	Schreibt den Wert des Busses in die im MAR gesetzte Adresse des RAM
LM	Load MAR	Lädt den Wert des Busses in das Adressenregister (Memory Address Register; MAR)
LI	Load Instruction	Lädt den Wert des Busses in das Befehlsregister
IP	Increment Pointer	Erhöht die aktuell im Befehlszähler gespeicherte Adresse des nächsten Befehls um eins
EP	Enable Pointer	Überträgt den Inhalt des Befehlszählers auf den Bus
LP	Load Pointer	Lädt den Wert des Busses in den Befehlszähler
LO	Load Output	Lädt den Wert des Busses in das Ausgaberegister (und zeigt somit die Dezimaldarstellung auf der Dezimalanzeige an)
$O1, O2, O3, O4$	Operation 1, 2, 3, 4	Setzt die auszuführende Operation der ALU und überträgt das Ergebnis auf den Bus. Die Operanden sind die zu dem Zeitpunkt der Anfrage gesetzten Werte des A und B Registers
ET, EA, EB, EC	Enable Temporary, A, B, C	Überträgt den Inhalt des Temporären (A, B, C) Registers auf den Bus
LT, LA, LB, LC	Load Temporary, A, B, C	Lädt den Wert des Busses in das Temporäre (A, B, C) Register
HLT	Halt	Hält den Taktgeber der CPU an und stoppt die aktuelle Ausführung

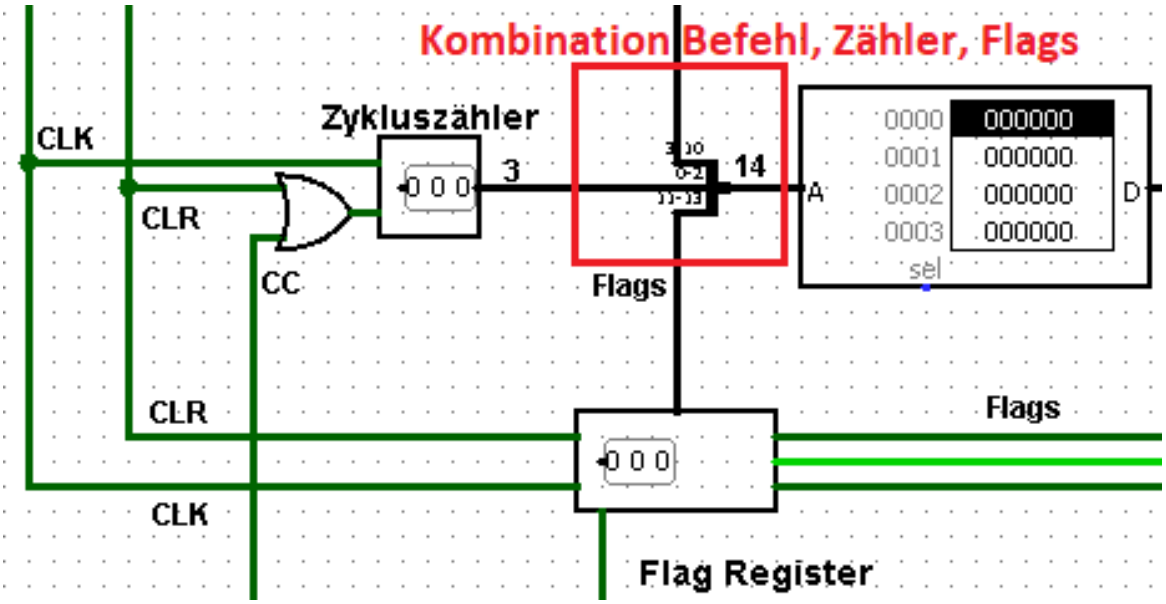
Anhang 1.6.6: Der Zykluszähler der Kontrolleinheit



Anhang 1.6.7: Der ROM der Kontrolleinheit



Anhang 1.6.8: Das Flag-Register an dem Kontroll-ROM



Anhang 1.6.9: Das Befehlsset der 8 Bit CPU

A bezeichnet das A-Register
B bezeichnet das B-Register
P bezeichnet den Operanden des Maschinenbefehls
RAM(P) bezeichnet den Inhalt der RAM-Adresse P

Mnemonic	Opcode Bin	Opcode Hex	Beschreibung
NOOP und Registerbefehle			
NOOP	0b00000000	0x00	Hält die CPU für 8 Taktzyklen im Leerlauf
LVA	0b00000001	0x01	A = P
LMA	0b00000010	0x02	A = RAM(P)
LVB	0b00000011	0x03	B = P
LMB	0b00000100	0x04	B = RAM(P)
LVC	0b00000101	0x05	C = P
LMC	0b00000110	0x06	C = RAM(P)
ALU Operationen			
ADD	0b00000111	0x07	A = A + P
ADDB	0b00001000	0x08	A = A + B
ADDC	0b00001001	0x09	A = A + C
SUB	0b00001010	0x0a	A = A - P
SUBB	0b00001011	0x0b	A = A - B
SUBC	0b00001100	0x0c	A = A - C
AND	0b00001101	0x0d	A = A AND P
ANDB	0b00001110	0x0e	A = A AND B
ANDC	0b00001111	0x0f	A = A AND C
OR	0b00010000	0x10	A = A OR P
ORB	0b00010001	0x11	A = A OR B
ORC	0b00010010	0x12	A = A OR C
XOR	0b00010011	0x13	A = A XOR P
XORB	0b00010100	0x14	A = A XOR B
XORC	0b00010101	0x15	A = A XOR C
NOT	0b00010110	0x16	A = NOT P
NOTA	0b00010111	0x17	A = NOT A
NOTB	0b00011000	0x18	A = NOT B
NOTC	0b00011001	0x19	A = NOT C
DECAD	0b00011010	0x1a	A = RAM(P) - 1
DEC	0b00011011	0x1b	A = P - 1
DECA	0b00011100	0x1c	A = A - 1
DECB	0b00011101	0x1d	A = B - 1
DECC	0b00011110	0x1e	A = C - 1
INCAD	0b00011111	0x1f	A = RAM(P) + 1
INC	0b00100000	0x20	A = P + 1
INCA	0b00100001	0x21	A = A + 1
INCB	0b00100010	0x22	A = B + 1
INCC	0b00100011	0x23	A = C + 1
AOUT	0b00100100	0x24	Setzt die Flags für den Inhalt des A-Registers

Ausgabebefehle			
OUTA	0b00100101	0x25	Ausgeben von A auf der Dezimalanzeige
OUTB	0b00100110	0x26	Ausgeben von B auf der Dezimalanzeige
OUTC	0b00100111	0x27	Ausgeben von C auf der Dezimalanzeige
Speicherbefehle und Transferbefehle			
STA	0b00101000	0x28	RAM(P) = A
STB	0b00101001	0x29	RAM(P) = B
STC	0b00101010	0x2a	RAM(P) = C
A2B	0b00101011	0x2b	B = A
A2C	0b00101100	0x2c	C = A
B2A	0b00101101	0x2d	A = B
B2C	0b00101110	0x2e	C = B
C2A	0b00101111	0x2f	A = C
C2B	0b00110000	0x30	B = C
HLT	0b00111000	0x38	Hält die CPU an
Vergleichsbefehle und (bedingte) Sprungbefehle			
CMP	0b00110001	0x31	Setzt die Flags für A - P
CMPB	0b00110010	0x32	Setzt die Flags für A - B
CMPC	0b00110011	0x33	Setzt die Flags für A - C
JMP	0b00110100	0x34	Springt zu RAM Adresse P für dem nächsten Befehl
JC	0b00110101	0x35	Springt zu RAM Adresse P, wenn die Carry-Flag gesetzt ist
JZ	0b00110110	0x36	Springt zu RAM Adresse P, wenn die Zero-Flag gesetzt ist
JN	0b00110111	0x37	Springt zu RAM Adresse P, wenn die Negative-Flag gesetzt ist
Unterprogrammbefehle			
RET	0b11111110	0xfe	Kehrt von der Subroutine zurück zur Adresse, die in 0xff gespeichert ist
CALL	0b11111111	0xff	Springt zu der Subroutine bei P, speichert den Pointer in 0xff

Quellenverzeichnis

Literaturquellen:

[1] Malvino, Albert P. und Brown, Jerald A. (1999), Digital Computer Electronics, 3. Aufl., o.O. https://pdfcookie.com/download/digital-computer-electronics-albert-paul-malvino-and-jerald-a-brownpdf-5lq3xewng8v7

[2] o.V. (2011), Computer Science 220: Assembly Language & Comp. Architecture-Topic Notes: Sequential Circuits, Mitschrift einer Vorlesung des Siena College, Siena https://pdfhoney.com/compress-pdf.html?fileurl=https://silo.tips/downloadFile/topic-notes-sequential-circuits&title=Topic+Notes%3A+Sequential+Circuits&utm_source=silotips&utm_medium=queue&utm_campaign=-1

[3] Prof. Dr. Wüst, Klaus (2003), Die Assemblersprache der Intel 80x86-Prozessoren, Gießen https://homepages.thm.de/~hg6458/AS.pdf

[4] Silc, J., Robic, B., Ungerer, T. (1999), Processor Architecture - From Dataflow to Superscalar and Beyond, 1. Aufl., o.O.

Videoquellen:

[5] Ben Eater (2015 - 2018), Building an 8-bit breadboard computer!, YouTube, https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU

[6] Riley, S. [Computerphile] (2018, 13. Februar) im Interview mit Professor Brailsford, Von Neumann Architecture - Computerphile, YouTube, https://www.youtube.com/watch?v=Ml3-kVYLNr8

Weiterführende Literatur:

[7] Tannenbaum, Andrew S. und Todd, Austin (2014), Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner, 6. Aufl., o.O.

[8] Schiffmann, Wolfram und Schmitz, Robert (2004), Technische Informatik 1: Grundlagen der Digitalen Elektronik, 5. Aufl., o.O.

[9] Schiffmann, Wolfram und Schmitz, Robert (2005), Technische Informatik 2: Grundlagen der Computertechnik, 5. Aufl., o.O.