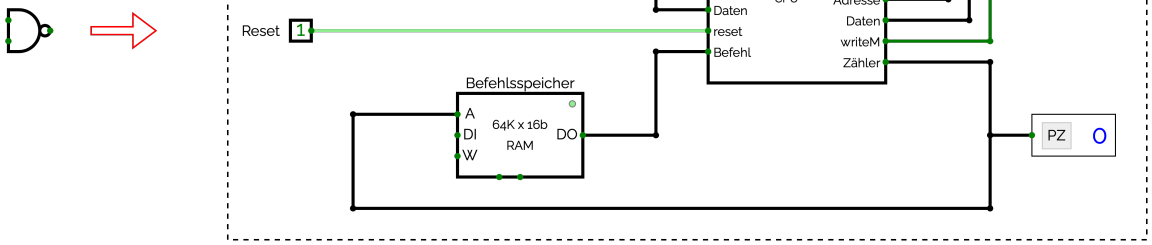
Von einfachen Schaltungen zu einem funktionierenden Computer



M. Anderegg

Vorwort

Das Ziel dieser Unterrichtseinheit, die sich an Schüler*innen im Ergänzungsfach Informatik in einem Schweizer Gymnasium richtet, ist das Ausleuchten der tiefliegenden Abstraktionsschichten der Informatik, die häufig als Black-Box verwendet werden: Wie kann ein Computer digitale Daten speichern und verarbeiten?

Um diese Frage zu beantworten, bauen die Schüler*innen selbst einen Computer! Damit die Probleme der physischen Welt ausgeklammert bleiben, arbeiten die Schüler*innen mit einer Simulationssoftware für logische Schaltungen „CircuitVerse“ (`https://circuitverse.org`)

Die Idee zu dieser Arbeit ist in Grundzügen schon früh entstanden. Im Januar 2020 während einer der ersten Vorlesungen der GymInf-Ausbildung, „Einführung in Computersysteme“ von Ulrich Ultes-Nitsche (Université de Fribourg) haben wir in einem Logikgatter-Simulator die Funktionsweise eines FlipFlops untersucht, einen adressierten Speicher modelliert, Halb- und Volladdierer entworfen und zu einem Addierer kombiniert. Diese Aufgaben haben mir viel Spass bereitet und einen ersten Einblick in den inneren Aufbau eines Computers ermöglicht. Zu diesem Zeitpunkt habe ich mich ein erstes Mal gefragt, ob es nicht möglich wäre, einen vollständigen Computer auf diese Art zu simulieren. In der Vorlesung haben wir das Thema jedoch nicht weiter vertieft und einen Sprung auf eine höhere Abstraktionsebene gemacht, in welcher die Funktionsweise von Betriebssystemen besprochen wurde. Nach meinem Empfinden blieb zwischen den NAND-Gattern und dem Linux-Kernel eine Lücke.

Den entscheidenden Hinweis, der schliesslich zu der vorliegenden Arbeit geführt hat, habe ich im Frühling 2021 von Matthias Hauswirth (Università della Svizzera italiana) erhalten. Er hat mich auf das nand2tetris-Projekt von Simon Schocken und Noam Nisan [3], [4] aufmerksam gemacht, wofür ich ihm sehr dankbar bin! In den darauffolgenden Wochen habe ich voller Begeisterung die zwölf Projekte des nand2tetris-Kurses absolviert. Beginnend mit NAND-Gattern habe ich den Bau eines Computers in einer Hardwarebeschreibungssprache simuliert und anschliessend den Weg von einer einfachen Maschinensprache zu einer höheren Programmiersprache nachgezeichnet. Wie der Name des Kurses verrät, endet das Projekt damit, dass man das Spiel Tetris auf dem selbst gebauten Computer implementieren kann. Damit war die Lücke geschlossen, und zwar auf eine enorm motivierende Art, die mit der grossen Befriedigung verbunden ist, einen eigenen Computer zu bauen - ganz ohne Lötkolben!

In der vorliegenden Arbeit wurden die Ideen von nand2tetris ergänzt, neu formuliert und so adaptiert, dass sie sich für den Einsatz im Rahmen des Ergänzungsfachs Informatik an einem Schweizer Gymnasium eignen.

Inhaltsverzeichnis

Zu dieser Unterrichtseinheit

Das Ziel dieser Unterrichtseinheitist das Ausleuchten der tiefliegenden Abstraktionsschichten der Informatik, die häufig als Black-Box verwendet werden: Wie kann ein Computer digitale Daten speichern und verarbeiten?

Um diese Frage zu beantworten, haben wir einen besonderen Weg gewählt: Sie werden die Funktionsweise eines Computers nachvollziehen, indem Sie selbst einen Computer bauen! Sie brauchen dazu aber keinen Lötkolben! Der Computer, den wir bauen werden, existiert nicht in der physischen Welt, sondern nur innerhalb einer Simulationssoftware für logische Schaltungen: „CircuitVerse“ `https://circuitverse.org`.

Der Einsatz der Simulationssoftware hat zwei Vorteile: Zum einen werden Sie beim Konstruieren eines Bauteils sofort feststellen, ob Sie dessen Funktionsweise richtig verstanden haben. Zum anderen können Sie alle Probleme der physikalischen Umsetzung ausklammern und sich auf die wesentlichen Fragen dieser Einheit konzentrieren: Wie funktionieren die Komponenten eines Computers? Wie kommunizieren sie miteinander? Wie werden die Befehle, die wir in einer höheren Programmiersprache eingeben, im Computer verarbeitet?

Auf dem Weg von einfachen Relais bis zu einem programmierbaren Computer werden Sie ein fundiertes Wissen darüber erlangen, wie ein Computer aufgebaut ist und wie die einzelnen Komponenten funktionieren. Ausserdem werden Sie erleben, wie zwei zentrale Konzepte der Informatik vielfach zur Anwendung kommen: die Modularisierung und die Abstraktion. Zur Lösung von Problemen werden Sie Module entwerfen, die in abstrahierter Form zur Lösung von komplexeren Problemen wieder herbeigezogen werden. Nur so wird es überhaupt möglich sein, den Weg vom Relais zum Computer nachzuzeichnen, ohne dabei den Überblick zu verlieren.

1 Boolesche Logik

Als der irische Mathematiker George Boole im Jahr 1847 mit seinem Werk *The Mathematical Analysis of Logic* [1] erstmals eine formale Beschreibung der Logik lieferte, konnte er nicht ahnen, dass seine Ideen die Grundlage für die Entwicklung von Computern bilden würden. Der Zusammenhang zwischen logischen Formeln und digitalen Schaltkreisen wurde erst 90 Jahre später durch den US-amerikanischen Mathematiker und Elektrotechniker Claude Shannon entdeckt und in seiner Arbeit *A Symbolic Analysis of Relay and Switching Circuits* von 1937 veröffentlicht [5].

In diesem Kapitel werden wir die Grundzüge der Booleschen Algebra kennenlernen, bevor wir sie im darauffolgenden Kapitel mit elektronischen Schaltkreisen in Verbindung bringen.

Aufgabe 1.1: Einstiegsproblem

Zoé schlägt ihren Freund*innen Adam, Bea, Chloé, Damian und Fiona einen Ausflug vor. Chloé sagt: „Ich komme, aber nur, falls Adam und Bea nicht beide kommen.“ Damian meint hingegen: „Ich komme, aber nur, falls Adam auch dabei ist, oder Bea zu Hause bleibt.“ Fiona meint schliesslich: „Wenn Adam kommt und Bea zuhause bleibt, oder umgekehrt, wenn Adam zuhause bleibt und Bea kommt, bin ich auch dabei. Ansonsten bleibe ich zu Hause.“

Je nach Teilnahme von Adam und Bea werden die anderen drei also ebenfalls teilnehmen oder dem Ausflug fernbleiben.

Verschaffen Sie sich anhand der nachstehenden Tabelle einen Überblick darüber, wer unter welchen Bedingungen mit von der Partie ist. Darin bedeutet eine 1, dass die entsprechende Person kommt, und eine 0, dass sie fernbleibt. Die vier Zeilen der Tabelle entsprechen den vier Bedingungen, zu denen Adam und Bea teilnehmen könnten (beide bleiben fern, nur Bea nimmt teil, nur Adam nimmt teil, beide nehmen teil). Die dritte Spalte spiegelt die Aussage von Chloé wider, dass sie teilnimmt, falls Adam und Bea nicht beide mitkommen.

Füllen Sie die beiden Spalten von Damian und Fiona entsprechend aus, sodass die Einträge zur Aussage der jeweiligen Person passen.

Adam	Bea	Chloé	Damian	Fiona
0	0	1		
0	1	1		
1	0	1		
1	1	0		

Die Tabelle aus dem „Einstiegsproblem“ist ein erstes Beispiel für das, was man in der Logik eine Wahrheitstabelle nennt. Betrachten Sie dazu den bereits ausgefüllten Bereich der Tabelle mit den Angaben zu Adam, Bea und Chloé. Die Teilnahme von Chloé kann eindeutig aus den Informationen über die Teilnahme von Adam und Bea ermittelt werden. Chloés Aussage, dass sie genau dann am Ausflug teilnimmt, wenn Adam und Bea nicht beide erscheinen, kann als eine Funktion der Form $(a, b) \mapsto c(a, b)$ aufgefasst werden, wobei a, b und c den Wert 1 oder 0 haben und für die Teilnahme von Adam, Bea und Chloé stehen. Dies ist ein Beispiel für eine sogenannte Boolesche Funktion:

Neue Konzepte und Begriffe : Boolesche Logik, Variablen und Funktionen

Die *Boolesche Logik* handelt von Operationen und Funktionen auf Variablen, die genau zwei Werte annehmen können. Solche Variablen nennt man *Boolesche Variablen* und die Werte, welche sie annehmen können, werden je nach Kontext mit wahr/falsch, ja/nein, hohe/tiefe Spannung oder auf viele weitere Arten bezeichnet. Der Einfachheit halber werden wir sie mit 1 und 0 bezeichnen – genauso wie auch die Werte einzelner Bits in der Informatik.

Eine *Boolesche Funktion von n Variablen* ($n \in \mathbb{N}$) ist eine Funktion, die als Input die Werte von n Booleschen Variablen annimmt und deren Output 1 oder 0 ist. Solche Funktionen kann man mit einer sogenannten *Wahrheitstabelle* definieren, in welcher zu jeder möglichen Belegung der Inputvariablen der entsprechende Output angegeben wird.

Schauen wir uns gleich konkrete Beispiele für Boolesche Funktionen an:

Beispiel 1.1: Erste Boolesche Funktionen

Die Boolesche Funktion NEG(x) invertiert den Wert der Inputvariable und wird häufig mit $\text{NEG}(x) = \overline{x}$ bezeichnet. Die entsprechende Wahrheitstabelle ist:

x	$\overline{x}$
0	1
1	0

Die beiden bekanntesten Booleschen Funktionen in zwei Variablen heissen *AND* und *OR*. Bezeichnet werden sie mit den beiden Symbolen $x \wedge y$ für $\text{AND}(x, y)$ beziehungsweise mit $x \vee y$ für $\text{OR}(x, y)$. Ihre Wahrheitstabellen sehen wie folgt aus:

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Anstatt mit einer Wahrheitstabelle können neue Boolesche Funktionen auch als eine Verkettung der Booleschen Operationen aus Beispiel 1.1 angegeben werden.

Beispiel 1.2: XOR als Boolescher Ausdruck

Die Funktion *XOR* kann durch den folgenden *Booleschen Ausdruck* definiert werden:

$$\text{XOR}(x, y) = (x \wedge \overline{y}) \vee (\overline{x} \wedge y).$$

Die Wahrheitstabelle der Funktion XOR lässt sich ausgehend von der Definition als Verkettung der drei Operationen NEG, AND, OR leicht herleiten, indem man von innen nach aussen arbeitet:

x	y	$\overline{x}$	$\overline{y}$	$x \wedge \overline{y}$	$\overline{x} \wedge y$	$\text{XOR}(x, y)$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

In der Wahrheitstabelle erkennt man, dass die XOR-Funktion genau dann den Wert 1 annimmt, wenn genau eine der beiden Inputvariablen den Wert 1 hat. So gelangen wir zu der Bezeichnung XOR als Abkürzung für „eXclusive OR“.

Aufgabe 1.2: Wahrheitstabellen für Boolesche Ausdrücke aufstellen

1. Die Funktion *NOR* ist definiert durch

   $$\text{NOR}(x, y) = \overline{x \vee y}.$$

   Geben Sie die Wahrheitstabelle der Funktion NOR an.

2. Die Funktion *XNOR* kann wie folgt definiert werden:

   $$\text{XNOR}(x, y) = (x \wedge y) \vee (\overline{x} \wedge \overline{y}).$$

   Geben Sie die Wahrheitstabelle der Funktion XNOR an. Vergleichen Sie das Resultat mit der Wahrheitstabelle von XOR. Was fällt Ihnen auf?

3. Wir haben nun einige Boolesche Funktionen mit zwei Inputvariablen untersucht. Wie viele unterschiedliche Boolesche Funktionen dieser Art gibt es insgesamt?

4. Ein Beispiel für eine Boolesche Funktion mit drei Inputs wäre:

   $$f(x, y, z) = \overline{x} \wedge (y \vee z).$$

   Geben Sie die Wahrheitstabelle dieser Funktion an.

Bisher haben wir stets ausgehend von einem Booleschen Ausdruck die entsprechende Wahrheitstabelle hergeleitet. Ist auch der umgekehrte Weg möglich? In anderen Worten: Lässt sich jede Boolesche Funktion als eine Verkettung der drei Operationen NEG, AND und OR darstellen? Das ist tatsächlich möglich! Der Beweis dieser Aussage ist aber nicht ganz leicht und wir werden hier darauf verzichten, zeigen aber anhand eines Beispiels das grundsätzliche Vorgehen.

Beispiel 1.3: Disjunkte Normalform einer Funktion

Wir betrachten eine Boolesche Funktion f in drei Variablen, die durch die folgende Wahrheitstabelle definiert ist:

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Nun richten wir unsere Aufmerksamkeit auf die drei Zeilen, in denen die Funktion den Wert 1 annimmt. Für jede dieser Zeilen definieren wir unter Verwendung der Operatoren NEG und AND einen Booleschen Ausdruck in x, y und z, der genau dann den Output 1 liefert, wenn die Inputvariablen den Wert entsprechend der betrachteten Zeile aufweisen. Der passende Ausdruck für die zweite Zeile in der Wahrheitstabelle der Funktion f ist $(\overline{x} \wedge \overline{y}) \wedge z$ und dessen Wahrheitstabelle sieht wie folgt aus:

x	y	z	$(\overline{x} \wedge \overline{y}) \wedge z$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Ein analoger Ausdruck für die vierte Zeile in der Wahrheitstabelle von f ist $(\overline{x} \wedge y) \wedge z$ und jener für die siebte Zeile lautet $(x \wedge y) \wedge \overline{z}$.

Die OR-Verknüpfung dieser drei Hilfsfunktionen entspricht dann der Funktion f:

$$f(x,y,z) = ((\overline{x} \wedge \overline{y}) \wedge z) \vee ((\overline{x} \wedge y) \wedge z) \vee ((x \wedge y) \wedge \overline{z}).$$

Ein solche Darstellung einer Booleschen Funktion als eine OR-Verknüpfung von UND-verknüpften Booleschen Variablen oder deren Negation nennt man *disjunkte Normalform.*

Genauso wie in dem Beispiel kann jede Boolesche Funktion auf einen Ausdruck zurückgeführt werden, der nur die NEG-, AND- und OR-Operationen verwendet. Dabei spielt es keine Rolle, wie viele Inputvariablen die Funktion hat. Die ganze Tragweite dieser Aussage kann man erahnen, wenn

man bedenkt, dass es zum Beispiel mehr als 10^{300} Boolesche Funktionen mit 10 Inputvariablen gibt – und jede einzelne kann als eine Verknüpfung der drei Grundoperationen dargestellt werden!

Aufgabe 1.3: Disjunkte Normalform einer Funktion

1. Folgen Sie dem Beispiel 1.3 und stellen Sie die folgende Funktion als einen Booleschen Ausdruck dar.

x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

2. Wie viele unterschiedliche Boolesche Funktionen mit n Inputvariablen gibt es?
   Tipp: Bestimmen Sie zuerst, wie viele unterschiedliche Belegungen die n Inputvariablen haben können.

Die ganze Vielfalt an Booleschen Funktionen haben wir damit auf bloss drei Operationen zurückgeführt. Das ist eine beachtliche Leistung, aber es geht noch besser...

Aufgabe 1.4: OR aus AND und NEG

Stellen Sie die Boolesche Funktion OR als eine Kombination von AND- und NEG-Funktionen dar.
Tipp: Es ist nur eine AND-Verknüpfung nötig.

Wenn sich jede Boolesche Funktion aus einer Kombination von NEG-, AND- und OR-Funktionen darstellen lässt und letztere wiederum eine Verkettung der ersten beiden ist, kann man folglich alle logischen Funktionen als eine Kombination der beiden Funktionen NEG und AND erzeugen.

Es reichen also bloss zwei logische Funktionen, um alle anderen zu erzeugen! Das ist erstaunlich, aber es geht sogar noch ein wenig besser: Eine einzige Funktion allein reicht aus, um alle anderen Booleschen Funktionen zu erzeugen!

Die Boolesche Funktion *NAND* ist definiert als

$$\text{NAND}(x,y) = \overline{x \wedge y}.$$

Die Wahrheitstabelle der NAND-Funktion lautet:

x	y	NAND(x,y)
0	0	1
0	1	1
1	0	1
1	1	0

Aufgabe 1.5: NEG und OR aus NAND

1. Füllen Sie die folgende Wahrheitstabelle aus. Um welche Funktion handelt es sich?

x	NAND(x, x)
0	
1	

2. Stellen Sie die OR-Funktion als eine Verkettung von NAND-Funktionen dar. Tipp: Die Übung 1.4 könnte helfen.

3. Stellen Sie auch die AND-Funktion als eine Verkettung von NAND-Funktionen dar.

Die Erkenntnisse aus den Übungen 1.4 und 1.5 kombiniert mit jenen aus dem Beispiel 1.3 implizieren, dass jede erdenkliche Boolesche Funktion letztlich eine Verkettung von NAND-Funktionen ist!

Beispiel 1.4: Boolesche Funktionen als Verkettung von NAND-Funktionen

Wir erinnern uns an das Einstiegsproblem 1.1, an den Ausflug, den Zoé mit ihren Freund*innen unternehmen möchte. Die Teilnahme von Chloé c, Damian d und Fiona f kann je als eine Boolesche Funktion aufgefasst werden, wobei die Inputvariablen a und b für das Erscheinen von Adam und Bea stehen.

a	b	$c(a,b)$	$d(a,b)$	$f(a,b)$
0	0	1	1	0
0	1	1	0	1
1	0	1	1	1
1	1	0	1	0

Um jede dieser Funktionen als Verkettung von NAND-Verknüpfungen darzustellen, werden sie zuerst als Boolescher Ausdruck angegeben. Dazu kann man beispielsweise die entsprechende disjunkte Normalform bestimmen wie in Beispiel 1.3. Bei den Funktionen im vorliegenden Beispiel geht es allerdings etwas einfacher und man findet leicht die folgenden Booleschen Ausdrücke:

$$c(a,b) = \overline{a \wedge b}, \qquad d(a,b) = a \vee \overline{b}, \qquad f(a,b) = \mathrm{XOR}(x,y) = (x \wedge \overline{y}) \vee (\overline{x} \wedge y).$$

Nun kann man mit den Ergebnissen aus Aufgabe 1.5 die vorkommenden NEG-, OR- und AND-Funktionen der Reihe nach durch NAND-Verkettungen ersetzen. Für die Funktion $c(a,b)$ erhält man:

$$\begin{aligned} c(a,b) &= \overline{a \wedge b} \\ &= \mathrm{NAND}(a,b) \end{aligned}$$

Analog können auch die beiden anderen Funktionen nur mit NAND erzeugt werden:

$$\begin{aligned} d(a,b) &= \mathrm{NAND}(\mathrm{NAND}(a,a),b) \\ f(a,b) &= \mathrm{NAND}(\mathrm{NAND}(a,\mathrm{NAND}(b,b)),\mathrm{NAND}(\mathrm{NAND}(a,a),b)). \end{aligned}$$

Zusammenfassung

Variablen, die nur zwei unterschiedliche Werte 0/1 annehmen können, werden Boolesche Variablen genannt. Entsprechend nennt man Funktionen, die mit Booleschen Variablen operieren, Boolesche Funktionen. Sie können durch Angabe ihrer Wahrheitstabellen oder durch einen Booleschen Ausdruck definiert werden.

Wichtige Boolesche Funktionen sind:

- $\text{AND}(x, y) = x \wedge y$. Der Output der AND-Funktion ist genau dann 1, wenn beide Inputs den Wert 1 aufweisen.
- $\text{OR}(x, y) = x \vee y$. Der Output der OR-Funktion ist genau dann 1, wenn mindestens ein Input den Wert 1 aufweist.
- $\text{NEG}(x) = \bar{x}$. Der Output der NEG-Funktion ist genau dann 1, wenn der Input 0 ist.
- $\text{NAND}(x, y) = \overline{x \wedge y}$. Der Output der NAND-Funktion ist genau dann 1, wenn mindestens einer der Inputs den Wert 0 hat.
- $\text{XOR}(x, y)$. Der Output der XOR-Funktion ist genau dann 1, wenn die Inputs unterschiedliche Werte aufweisen.
- $\text{XNOR}(x, y)$. Der Output der XNOR-Funktion ist genau dann 1, wenn die Inputs die gleichen Werte aufweisen.

Jede Boolesche Funktion lässt sich als eine Verkettung von NAND-Funktionen darstellen.

2 Logikgatter und logische Schaltungen

Im Werk von George Boole stehen die beiden möglichen Werte 1 und 0 der Booleschen Variablen für den Wahrheitsgehalt „true“ oder „false“ einer Aussage. Viele Resultate der Booleschen Algebra lassen sich aber auch auf andere Situationen übertragen, in denen zwei Zustände unterschieden werden, zum Beispiel – das war die bahnbrechende Idee von Claude Shannon – auf elektronische Schaltkreise. So ist eine fruchtbare Analogie zwischen Booleschen Funktionen und elektronischen Schaltkreisen entstanden.

In diesem Kapitel werden Sie lernen, wie man für jede Boolesche Funktion eine elektronische Schaltung realisieren kann, welche für jeden möglichen Input den richtigen Output automatisch erzeugt. Damit dies möglich ist, müssen als Erstes die beiden Werte 1 und 0 der Booleschen Variablen innerhalb eines Schaltkreises durch eine physikalische Grösse repräsentiert werden: Liegt an einem Ein- oder Ausgangssignal eine hohe Spannung an, wird das als eine 1 interpretiert, eine tiefe Spannung hingegen als eine 0.

Die ersten elektronischen Schaltungen, die Boolesche Funktionen realisieren, wurden mit mechanischen Relais gebaut. Betrachten Sie den folgenden schematischen Aufbau eines Relais:

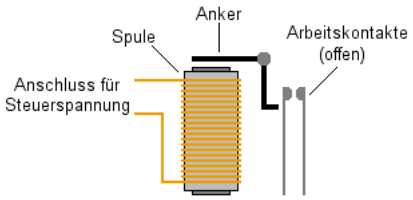


Abbildung 1: Relais im Ruhemodus
Quelle: `https://commons.wikimedia.org/wiki/File:Relais_ruhe.png`, gemeinfrei

Wird Spannung an den Steuerschaltkreis links im Bild angelegt, fliesst Strom durch die Spule. Diese erzeugt ein magnetisches Feld, das den Anker anzieht und die beiden Arbeitskontakte rechts im Bild verbindet.

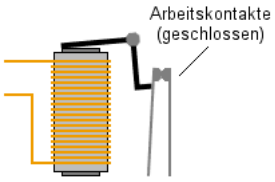


Abbildung 2: Wenn Spannung am Steuerschaltkreis anliegt, werden die Kontakte rechts verbunden
Quelle: `https://commons.wikimedia.org/wiki/File:Relais_arbeit.png`, gemeinfrei

Das in den Abbildungen dargestellte Relais ist ein sogenanntes „Relais mit Arbeitskontakt“ (kurz: AK-Relais), das den Arbeitsstromkreis genau dann schliesst, wenn Spannung am Steuerschaltkreis anliegt. Daneben gibt es auch „Ruhekontakt-Relais“ (kurz: RK-Relais), welche die Arbeitskontakte

genau dann verbinden, wenn am Steuerschaltkreis *keine* Spannung anliegt.

Mit diesen beiden Relais-Arten als Bausteinen lässt sich ein vollständiger Computer bilden! Bereits ein Jahr nach Erscheinen von Claude Shannons Werk begann Konrad Zuse mit dem Bau der ersten digitalen Rechenmaschine – der Z3. Sie wurde 1941 fertiggestellt und bestand aus rund zweitausend mechanischen Relais [6].

Heute werden anstelle der mechanischen Relais Transistoren eingesetzt, welche zuverlässiger arbeiten und vor allem viel weniger Platz benötigen. Wie aber kann man aus mehreren Relais einen Computer bauen? Nun: Schritt für Schritt!

Wir beginnen damit, dass wir ein RK-Relais als eine elektronische Schaltung auffassen, die eine Boolesche Funktion realisiert. Betrachten Sie die folgende Schaltung:

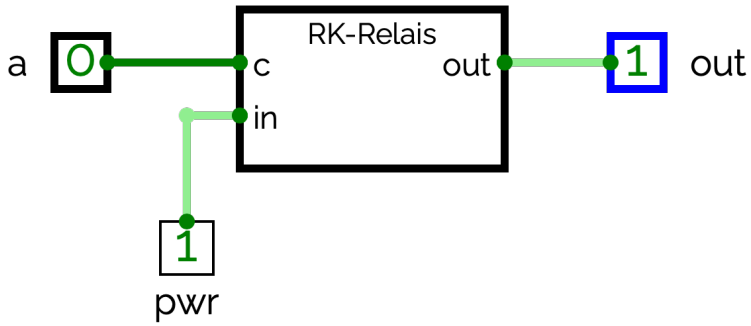


Abbildung 3: Elektronische Schaltung aus einem Relais mit Input $a = 0$

Die Schaltung besteht aus einem RK-Relais mit zwei Anschlüssen: c für die Steuerspannung und in für das Arbeitssignal. Ein weiterer Bestandteil der Schaltung ist das Signal pwr, an das konstant eine hohe Spannung anliegt.

Die beiden Signale a und out interpretieren wir als In- respektive Outputsignal der Schaltung. Das Bild zeigt die Berechnung des Outputs $f(a)$ einer Booleschen Funktion f für den konkreten Input $a = 0$. Weil in diesem Fall keine Spannung am Steuersignal c des RK-Relais anliegt, sind die beiden Arbeitssignale in und out miteinander verbunden. Damit liegt am Outputsignal eine hohe Spannung an und der von der Schaltung berechnete Funktionswert ist $f(0) = 1$.

Die Berechnung des Werts der Funktion f für den Input $a = 1$ wird in hingegen in Abbildung 4 dargestellt.

Wegen der hohen Spannung am Steuersignal c des RK-Relais sind in diesem Fall die beiden Signale in und out nicht verbunden und am Outputsignal liegt somit keine Spannung an: $out = 0$. Für die

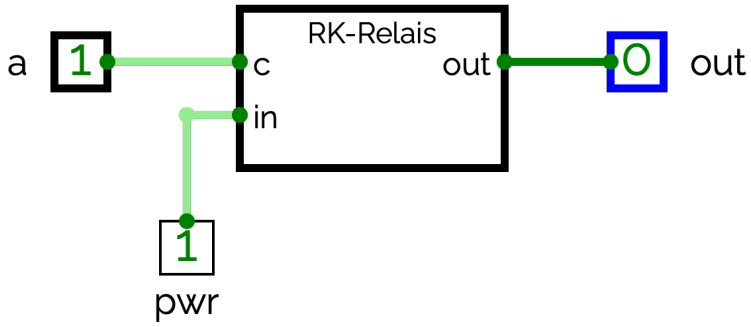


Abbildung 4: Elektronische Schaltung aus einem Relais mit Input $a = 1$

Funktion f, die von dieser Schaltung realisiert wird, gilt $f(1) = 0$. Zusammengefasst gilt $f(0) = 1$ und $f(1) = 0$. Die Schaltung realisiert die Boolesche Funktion NEG!

Analog lassen sich mit den Relais-Bausteinen auch Schaltungen entwerfen, die Boolesche Funktionen in mehreren Variablen realisieren. Betrachten Sie die folgende Schaltung:

Abbildung 5: Elektronische Schaltung aus einem Relais mit Input $a = 0, b = 0$

Diese Abbildung stellt die Berechnung des Outputs der elektronischen Schaltung für die zwei konkreten Inputs $a = 0$ und $b = 0$ dar. Die beiden Inputsignale a und b werden an ein Relais mit Arbeitskontakt geleitet. Aus der Perspektive des AK-Relais übernimmt das Signal a die Rolle des Steuersignals (mit c bezeichnet) und b jene des Arbeitssignals (mit in bezeichnet). Weil die Spannung am Steuersignal tief ist ($a = 0$), wird im AK-Relais keine Verbindung zwischen in und out hergestellt. Am Outputsignal des Relais liegt unter diesen Voraussetzungen keine Spannung an ($out = 0$).

Abbildung 6 zeigt die vier Berechnungen des Outputs für alle vier möglichen Belegungen der Inputsignale:

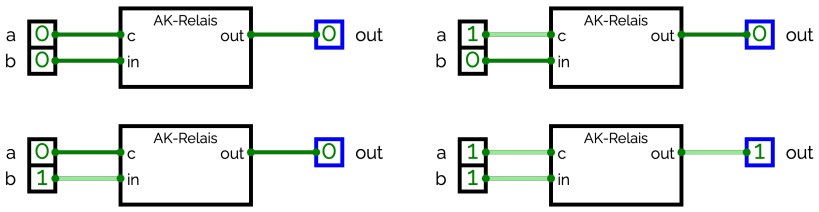


Abbildung 6: Elektronische Schaltung aus einem Relais

In den beiden Fällen links im Bild beträgt der Wert des Inputsignals $a = 0$. Damit stellt das AK-Relais keine Verbindung zwischen den Signalen *in* und *out* her und am Outputsignal liegt keine Spannung an ($out = 0$). Die Bilder rechts stellen hingegen die Situation dar, in der der Input $a = 1$ ist. In diesem Fall werden die Arbeitskontakte im AK-Relais verbunden. Wenn $a = 1$ ist, liegt somit am Outputsignal des Schaltkreises genau dann Spannung an, wenn am Inputsignal b Spannung anliegt.

Der Wert des Outputsignals *out* der Schaltung in Abhängigkeit der beiden Inputsignale a und b kann in einer Wahrheitstabelle wie folgt zusammengefasst werden:

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Wir erkennen darin die Wahrheitstabelle der AND-Funktion wieder. Die vorgestellte elektronische Schaltung aus einem AK-Relais ist also in der Lage, die Werte der AND-Funktion selbstständig zu berechnen!

Die beiden Schaltungen lassen sich auch kombinieren. Betrachten Sie die folgende Anordnung:

Erneut fassen wir die beiden Signale a und b als Inputs, *out* als Output und $pwr = 1$ als festen Bestandteil der Schaltung auf. Das Bild zeigt also die Berechnung des Wertes $f(a, b)$ einer Booleschen Funktion f für die beiden konkreten Inputwerte $a = 0$ und $b = 1$. Die beiden Inputsignale a und b werden als Kontrollsignal c und als Inputsignal *in* an ein Relais mit Arbeitskontakt geleitet. Weil die Spannung am Kontrollsignal tief ist, wird im AK-Relais keine Verbindung zwischen *in* und *out* hergestellt. Am Outputsignal des Relais liegt keine Spannung an.

Das nachfolgende Relais mit Ruhekontakt erhält als Kontrollsignal den Output des AK-Relais. Weil daran keine Spannung anliegt, ist im RK-Relais das *in*-Signal mit dem *out*-Signal verbunden, womit schliesslich am Outputsignal Spannung anliegt, was durch die Eins im blauen Quadrat rechts dargestellt ist.

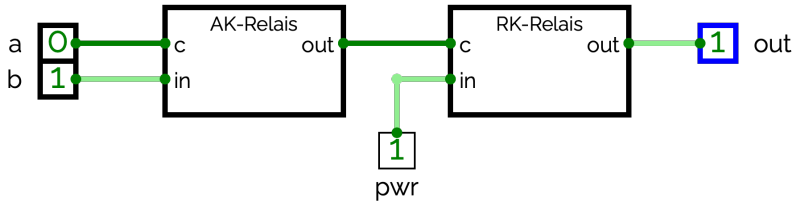


Abbildung 7: Elektronische Schaltung mit zwei Relais

Aufgabe 2.1: Kombination aus zwei Relais

1. Überlegen Sie, wie sich der Zustand des Outputsignals *out* der Schaltung in Abbildung 7 in Abhängigkeit der Inputs a und b verändert, und vervollständigen Sie die Tabelle:

a	b	out
0	0	
0	1	1
1	0	
1	1	

2. Vergleichen Sie die Tabelle mit den Wahrheitstabellen aus Kapitel 1. Welche Boolesche Funktion wird mit dieser elektronischen Schaltung realisiert?

In Kapitel 1 haben wir bereits nachvollziehen können, dass sich jede logische Funktion als eine Verkettung von NAND-Funktionen darstellen lässt. Nun, da wir wissen, dass sich die NAND-Funktion als elektronische Schaltung realisieren lässt, ist auch augenblicklich klar, dass dies für jede Boolesche Funktion gilt!

Neue Konzepte und Begriffe : Logikgatter und logische Schaltungen

Eine elektronische Schaltung, die eine Boolesche Grundfunktion (NEG, OR, AND, XOR, NOR, NAND, XNOR) realisiert, nennt man *Logikgatter* oder kurz *Gatter*.

Die Verknüpfung mehrerer Logikgatter zu einem komplexeren Bauteil mit mehreren In- und Outputsignalen nennt man eine *logische Schaltung*.

Fortan werden wir uns einer symbolischen Darstellung von Logikgattern bedienen. Abbildung 8 zeigt das Symbol für das NAND-Gatter.

Die Inputsignale können über die beiden Anschlüsse a und b an das Gatter geleitet werden. Innerhalb des Gatters wird der Funktionswert $\text{NAND}(a, b)$ berechnet und an das Outputsignal *out* weitergeleitet.

Hier haben wir zum ersten Mal ein Prinzip angewandt, das uns während des gesamten Weges hin

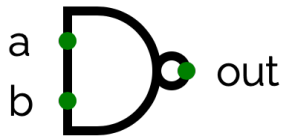


Abbildung 8: Schematische Darstellung des NAND-Gatters

zu einem funktionierenden Computer begleiten wird. Durch das Verwenden des Symbols für das NAND-Gatter haben wir dessen inneren Aufbau abstrahiert. In der Einführungsaufgabe haben wir nachvollzogen, dass es möglich ist, ein solches Logikgatter herzustellen. Ob dies nun mit zwei Relais oder mit Transistoren gemacht wird, ist für die weiteren Überlegungen nicht mehr wichtig. Von Bedeutung ist bloss, dass es ein solches Bauteil gibt, das die NAND-Funktion als Schaltung realisiert!

Ausgehend vom NAND-Gatter können wir nun weitere Boolesche Funktionen als logische Schaltungen realisieren:

Aufgabe 2.2: Logische Schaltungen für die Grundfunktionen im Simulator bauen

Für die Übungen werden wir den Logikgatter-Simulator `https://circuitverse.org` verwenden. Navigieren Sie zur Seite und starten Sie den Simulator.

1. Bilden Sie die logische Schaltung in Abbildung 9 nach. Die nötigen Elemente finden Sie im Menü CIRCUIT ELEMENTS unter Input, Output und Gates. Wenn Sie auf den Input klicken, können Sie dessen Zustand wechseln. Welche Boolesche Funktion wird mit der Schaltung in Abbildung 9 realisiert?

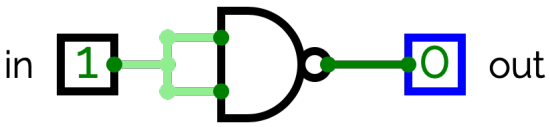


Abbildung 9: Eine erste logische Schaltung

2. Verbinden Sie im Simulator zwei Inputsignale, drei NAND-Gatter und ein Outputsignal so, dass damit die OR-Funktion realisiert wird. (Tipp: vgl. Übung 1.5).
   Testen Sie Ihre Lösung, indem Sie die vier möglichen Zustände der Inputsignale einstellen und prüfen, ob das Outputsignal den richtigen Zustand gemäss der Wahrheitstabelle der OR-Funktion annimmt.
3. Entwerfen Sie im Simulator eine Schaltung für die AND-Funktion als eine Kombination von NAND-Gattern.
4. In Beispiel 1.2 haben wir die XOR-Funktion kennengelernt: $\text{XOR}(x, y) = (x \wedge \overline{y}) \vee (\overline{x} \wedge y)$. Bauen Sie im Simulator eine logische Schaltung für die XOR-Funktion als eine Verkettung von NAND-Gattern.

Für die Funktionen AND, OR, NEG und XOR gibt es eigene Gattersymbole. Die Abbildung 10 zeigt diese symbolische Darstellung der Logikgatter. Auf der linken Seite sind jeweils die Anschlüsse für die beiden Inputsignale a und b und rechts jene für das Outputsignal *out* dargestellt.

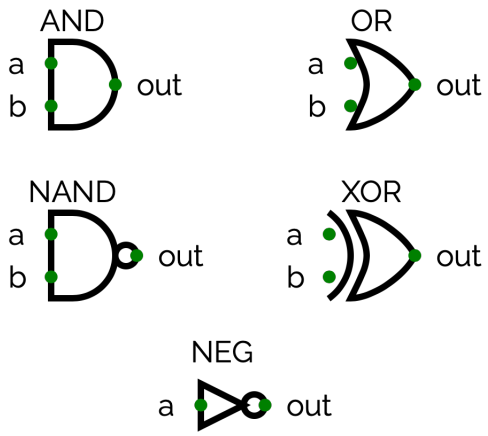


Abbildung 10: Gattersymbole

Genauso wie sich jede komplexe Boolesche Funktionen auf eine Kombination von einfacheren Funktionen wie NEG, AND und OR (oder nur NAND!) zurückführen lässt, kann man die einfachen Gatter aus Abbildung 10 so verdrahten, dass sie komplexere Boolesche Funktionen realisieren.

Beispiel 2.1: Logische Schaltung für eine vorgegebene Funktion

Die Funktion $f(x, y, z) = \overline{x} \wedge (y \vee z)$ wird durch die Schaltung in Abbildung 11 realisiert.

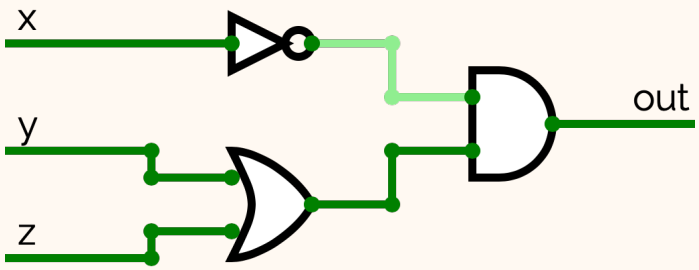


Abbildung 11: Schaltung für die Funktion f

Aufgabe 2.3: Komplexere Schaltungen im Simulator bauen

In dieser Übung werden Sie einige logische Schaltungen im Simulator bauen, die wichtige Rollen innerhalb eines Computers übernehmen.
Für die Übungen werden wir wieder den Logikgatter-Simulator `https://circuitverse.org` verwenden. Navigieren Sie zur Seite und starten Sie den Simulator.

1. Kombinieren Sie mehrere OR-Gatter, um eine OR-Funktion mit vier Inputs zu realisieren: $\mathrm{OR}_4(a, b, c, d) = a \vee b \vee c \vee d$. Der Output dieser Funktion ist genau dann 1, wenn mindestens ein Input den Wert 1 aufweist.

2. In Übung 1.2 haben wir die XNOR-Funktion eingeführt als

$$\mathrm{XNOR}(x, y) = (x \wedge y) \vee (\overline{x} \wedge \overline{y}).$$

   Entwerfen Sie eine Schaltung für die Funktion XNOR als eine Verknüpfung von NEG-, AND- und OR-Gatter. Testen Sie durch Manipulation der Inputzustände, ob Ihre logische Schaltung die Wahrheitstabelle von XNOR erfüllt:

x	y	$\mathrm{XNOR}(x, y)$
0	0	1
0	1	0
1	0	0
1	1	1

3. Ein *Multiplexer* (kurz *Mux*) ist eine logische Schaltung mit drei Inputs a, b, sel und einem Output *out*.

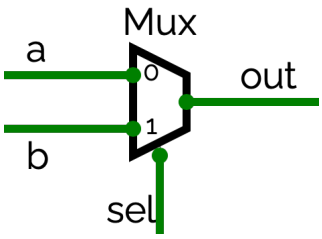


Abbildung 12: Symbol für die Mux-Schaltung

Je nach Wert des „Selektors“ *sel* soll das Signal des Inputs a oder jenes von b an den Output *out* weitergeleitet werden. Die Wahrheitstabelle des Multiplexers lautet:

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

In kompakter Schreibweise kann die Wahrheitstabelle auch wie folgt angegeben werden:

sel	out
0	a
1	b

Bauen Sie im Simulator einen Multiplexer als eine Kombination von NEG-, AND- und OR-Gatter.

4. Das Gegenstück zum Multiplexer wird *Demultiplexer* (kurz *DMux*) genannt. Ein DMux hat zwei Inputsignale *in* und *sel* und zwei Outputsignale *a* und *b*.

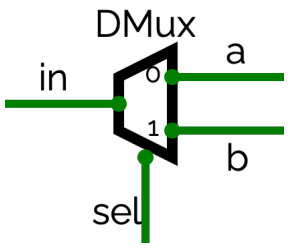


Abbildung 13: Symbol für die DMux-Schaltung

Je nach Wert von *sel* wird das Inputsignal *in* an *a* oder *b* weitergeleitet. Die (kompakte) Wahrheitstabelle von DMux lautet:

sel	*a*	*b*
0	*in*	0
1	0	*in*

Kombinieren Sie im Simulator NEG-, AND- und OR-Gatter, um einen Demultiplexer zu realisieren.

Zusammenfassung : Logikgatter und logische Funktionen

Die Booleschen Grundfunktionen (AND, OR, NEG, NAND, XOR, XNOR) lassen sich als elektronische Schaltungen realisieren. Diese Schaltungen nennt man *Logikgatter*.

Durch eine Kombination von Logikgattern in eine *logische Schaltung* lassen sich komplexere logische Verknüpfungen realisieren. Speziell erwähnenswert sind zwei Schaltungen, die im weiteren Verlauf häufig eingesetzt werden:

- Der *Multiplexer* ist eine logische Schaltung, mit welcher gewählt werden kann, welches Inputsignal an den Output weitergeleitet wird.
- Der *Demultiplexer* hingegen ist eine Schaltung, bei der gewählt werden kann, an welchen Output das Eingangssignal weitergeleitet werden soll.

3 Logische Schaltungen für Addition und Subtraktion

In Kapitel 2 haben wir gesehen, dass sich jede Boolesche Funktion durch eine logische Schaltung realisieren lässt. Im Folgenden werden wir einfache Schaltungen zu immer komplexeren Gebilden zusammenfügen, bis wir schliesslich in der Lage sein werden, Zahlen damit zu addieren und zu subtrahieren.

Auf den ersten Blick mag der Weg von den Booleschen Funktionen zur Addition von Zahlen weit erscheinen. Das Gegenteil ist der Fall! Der Zusammenhang zeigt sich, wenn man die schriftliche Addition von Binärzahlen genauer unter die Lupe nimmt.

3.1 Addition von Binärzahlen

Bestimmt kennen Sie die Binärdarstellung von Zahlen schon und wissen auch, wie man die Grundoperationen darin durchführt (vgl. Kapitel 1 in [2]). Damit die Beziehung zwischen der Addition von Binärzahlen und den Booleschen Funktionen klar erkennbar wird, repetieren wir anhand eines Beispiels die schriftliche Addition von zwei vierstelligen Binärzahlen.

Beispiel 3.1: Addition der beiden Binärzahlen 1010_2 und 1011_2

Betrachten wir als Beispiel die schriftliche Addition der beiden Binärzahlen 1010_2 und 1011_2:

$$\begin{array}{rr} & {\scriptstyle 1\ \ 1\ \ \ \ } \\ & 1010_2 \\ + & 1011_2 \\ \hline & 10101_2 \end{array}$$

Nun gehen wir schrittweise durch die Überlegungen, die zum Resultat der Addition führen. Zunächst wird die Einerziffer ganz rechts betrachtet: $0_2 + 1_2 = 1_2$:

$$\begin{array}{rr} & {\scriptstyle 1\ \ 1\ \ \ \ } \\ & 1010_2 \\ + & 1011_2 \\ \hline & 10101_2 \end{array}$$

Als Nächstes richten wir die Aufmerksamkeit auf die zweite Ziffer von rechts: die Zweierziffer. Hier rechnen wir $1_2 + 1_2 = 10_2$. Die Zweierziffer des Resultats (in grün dargestellt) wird als Übertrag zur nächsthöheren Stelle dazuaddiert:

$$\begin{array}{rr} & {\scriptstyle 1\ \ 1\ \ \ \ } \\ & 1010_2 \\ + & 1011_2 \\ \hline & 10101_2 \end{array}$$

Bei der dritten Ziffer von rechts werden nun drei Bits addiert: $1_2 + 0_2 + 0_2 = 1_2$:

$$\begin{array}{rr} & {\scriptstyle 1\ 1} \\ & 1010_2 \\ + & 1011_2 \\ \hline & 10101_2 \end{array}$$

Bei der Addition der Achterziffer entsteht wiederum ein Übertrag:

$$\begin{array}{rr} & {\scriptstyle 1\ 1} \\ & 1010_2 \\ + & 1011_2 \\ \hline & 10101_2 \end{array}$$

Schliesslich wird noch die Sechszehnerziffer als die Summe des Übertrags mit den beiden führenden Nullen der Zahlen gebildet:

$$\begin{array}{rr} & {\scriptstyle 1\ 1} \\ & 01010_2 \\ + & 01011_2 \\ \hline & 10101_2 \end{array}$$

Aufgabe 3.1: Addition von Binärzahlen

Versuchen Sie es selbst:

1. Berechnen Sie nach dem Muster oben: $11010_2 + 110110_2$. Kontrollieren Sie das Resultat, indem Sie die beiden Zahlen und die Summe ins Dezimalsystem übertragen.

2. Die schriftliche Addition beginnt immer mit der Ziffer ganz rechts - dem niedrigstwertigen Bit. In dieser Aufgabe wollen wir genauer untersuchen, wie diese Ziffern im Laufe der Addition verarbeitet werden. Dazu nennen wir die Einerziffer des ersten Summanden a_0 und jene des zweiten b_0. Ausserdem soll s_0 die Einerziffer der Summe bezeichnen und c den Übertrag (englisch „carry“). Füllen Sie die folgende Tabelle aus:

a_0	b_0	c	s_0
0	0	0	0
0	1		
1	0		
1	1		

   Mit welchen uns bekannten Booleschen Funktionen lassen sich s_0 und c aus a_0 und b_0 ermitteln?

3.2 Eine logische Schaltung für die Addition

Offenbar ist also die schriftliche Addition von Binärzahlen sehr eng mit den logischen Funktionen AND und XOR verwandt. Diese lassen sich wiederum als elektronische Schaltungen realisieren. In diesem Kapitel kombinieren wir diese beiden Erkenntnisse und werden im Simulator eine Schaltung bauen, die in der Lage ist, 4-Bit-Zahlen zu addieren.

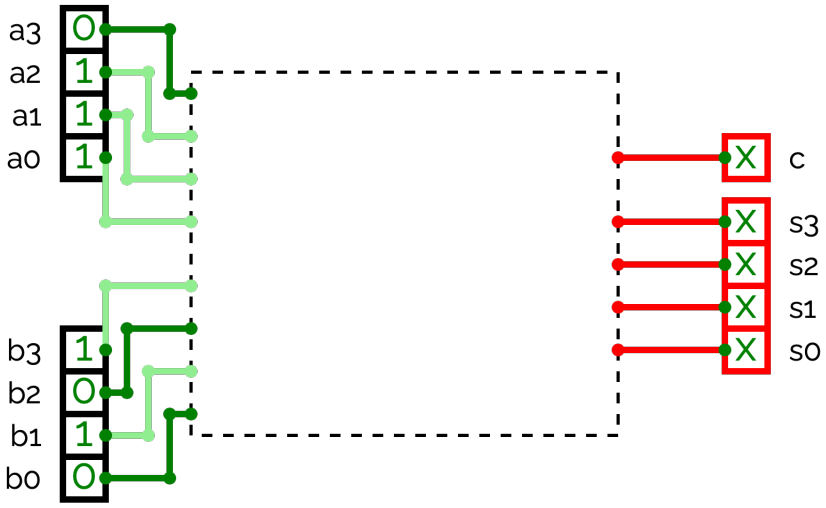


Abbildung 14: Schnittstellen der logischen Schaltung für die Addition der Zahlen $a = 0111_2$ und $b = 1010_2$

Abbildung 14 zeigt die Schnittstellen der Schaltung, die wir konstruieren werden. Der Wert der beiden 4-Bit-Summanden a und b und der Summe s wird dabei durch den Zustand von je vier Signalen angegeben, wobei jedes Signal für eine Ziffer der entsprechenden Zahl steht. Das Outputsignal c dient als Übertrag für den Fall, dass die Summe der beiden 4-Bit-Zahlen grösser als 15 ist und damit fünf binäre Ziffern umfasst.

Im Beispiel, das in Abbildung 14 dargestellt wird, sollen also die beiden Zahlen $a = 0111_2$ und $b = 1010_2$ addiert werden. Die Summe dieser beiden Zahlen lautet $0111_2 + 1010_2 = 10001_2$. Die gesuchte logische Schaltung müsste also bei dieser Belegung der Inputsignale den Output $c = 1, s_3 = 0, s_2 = 0, s_1 = 0, s_0 = 1$ erzeugen.

Man könnte nun natürlich direkt mit der vorgegebenen Schnittstelle beginnen und die acht Inputsignale so lange mit logischen Funktionen bearbeiten, bis man damit das gewünschte Resultat erzeugt. Allerdings würde die so konstruierte Schaltung unweigerlich sehr unübersichtlich werden – sofern wir überhaupt erfolgreich sein würden. Stattdessen werden wir diese grosse Aufgabe in einfachere Teilprobleme gliedern. Wir orientieren uns an der schriftlichen Addition und beginnen mit der Verarbeitung der niedrigstwertigen Bits.

Wenn die beiden Einerziffern a_0 und b_0 addiert werden, entsteht eine 2-Bit-Summe mit den Ziffern c und s_0, wobei s_0 die Einerziffer der Summe ist und c den Übertrag auf die nächsthöhere Ziffer angibt:

$$\begin{array}{rr} & {\scriptstyle c} \\ & xxxa_0 \\ + & xxxb_0 \\ \hline & xxxs_0 \end{array}$$

Neue Konzepte und Begriffe : Der Halbaddierer

Der *Halbaddierer* ist eine logische Schaltung mit zwei Inputsignalen a_0 und b_0 und zwei Outputsignalen c und s_0, welche die 1-Bit-Addition realisiert. Die Wahrheitstabelle für den Halbaddierer lautet also:

a_0	b_0	c	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Aufgabe 3.2: Konstruktion des Halbaddierers

1. Überlegen Sie zuerst ohne Simulator, wie man die Logikgatter der Grundfunktionen zu einem Schaltkreis kombinieren kann, der in der Lage ist, die Outputs des Halbaddierers zu berechnen. Fertigen Sie von Hand eine Skizze des Schaltkreises.

2. Starten Sie den Simulator und öffnen Sie ein neues Projekt. Benennen Sie das Projekt und den Schaltkreis im Fenster „PROPERTIES“ so, wie in der Abbildung vorgegeben.

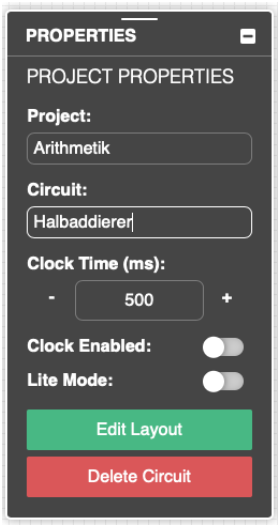


Abbildung 15: Projekt- und Schaltkreisname

3. Fügen Sie dem Schaltkreis „Halbaddierer“ zwei Inputsignale a_0 und b_0 und zwei Outputsignale c und s_0 hinzu. Nutzen Sie die „Label“ im „PROPERTIES“ Menü, um Bezeichnungen für die Signale einzuführen.

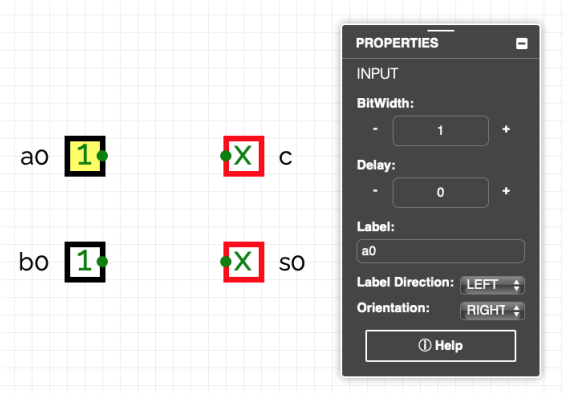


Abbildung 16: Das ausgewählte Inputsignal wird mit a0 bezeichnet

4. Kombinieren Sie nun die Inputsignale und die AND- und XOR-Gatter, um die gewünschte Ausgabe gemäss der Wahrheitstabelle in Definition 3.2 zu erzeugen.

5. Speichern Sie das Projekt. Wir werden es wieder benötigen!

Mit dem Halbaddierer sind wir einen grossen Schritt in Richtung Addition von Binärzahlen vorwärtsgekommen. Die beiden niedrigstwertigen Bits können damit bereits addiert werden. Dem Verfahren der schriftlichen Addition folgend, richten wir unsere Aufmerksamkeit auf das nächste Bit – die Zweierziffer.

Im Unterschied zur Addition der Einerziffern ist hier zu beachten, dass insgesamt drei Bits addiert werden. Zu den beiden Ziffern a_1 und b_1 der Summanden kommt nämlich der vermeintliche Übertrag c_1 aus der Addition der nächsttieferen Ziffern dazu:

$$\begin{array}{rr} & {\scriptstyle c_2 c_1} \\ & xxa_1x \\ + & xxb_1x \\ \hline & xxs_1x \end{array}$$

Diese Berechnung wird genau gleich für alle weiteren Ziffern durchgeführt. Deswegen werden wir auf die 1 in den Indizes verzichten und suchen etwas allgemeiner formuliert nach einer logischen Schaltung, die drei 1-Bit-Zahlen a, b und c_{in} addiert und die beiden Ziffern c_{out} und s der Summe ausgibt.

Neue Konzepte und Begriffe : Der Volladdierer

Der *Volladdierer* ist eine logische Schaltung mit drei Inputsignalen a, b und c_{in} und zwei Outputsignalen c_{out} und s, welche die 1-Bit-Addition von drei Zahlen realisiert. Dabei werden die Inputs a, b und c als binäre, einstellige Zahlen interpretiert, der Output hingegen als die zweistellige binäre Zahl $a + b + c$.

Die Wahrheitstabelle für den Volladdierer lautet also:

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Aufgabe 3.3: Konstruktion des Volladdierers

1. Öffnen Sie das Projekt „Arithmetik“, das Sie in Aufgabe 3.2 angelegt haben, und beginnen Sie darin eine neue logische Schaltung mit dem Namen „Volladdierer“.

Abbildung 17: Neuen Schaltkreis anlegen

2. Fügen Sie einen Halbaddierer in die neue Schaltung ein. Das geht, indem Sie eine „SubCircuit“ einfügen.

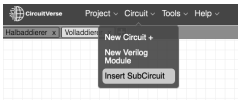


Abbildung 18: SubCircuit einfügen

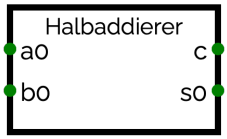


Abbildung 19: So sieht der eingefügte Halbaddierer aus.

Tipp: Sie können das Erscheinungsbild des Halbaddierers anpassen, wenn Sie zum Schaltkreis des Halbaddierers zurückkehren und dort im „PROPERTIES“-Menü auf „Layout“ klicken.

3. Konstruieren Sie einen Volladdierer mit den drei Inputsignale a, b und c_{in} und den zwei Outputs c_{out} und s. Überprüfen Sie die Lösung mit der Wertetabelle in Definition 3.2.

Mit dem Halb- und dem Volladdierer haben wir Bauteile entwickelt, welche die Addition der einzelnen Ziffern zweier Binärzahlen durchführen können. Damit ist es an der Zeit, das Ziel dieses Teilkapitels in Angriff zu nehmen.

Aufgabe 3.4: Konstruktion eines 4-Bit-Addierers und Incrementer

1. Fügen Sie dem Projekt „Arithmetik“ einen neuen Schaltkreis mit dem Namen „4-Bit-Addierer“ an. Realisieren Sie darin einen 4-Bit-Addierer mit Schnittstellen gemäss Abbildung 14.

2. Entwerfen Sie eine logische Schaltung „4-Bit-Incrementer“, die eine 4-Bit-Zahl um eins erhöht. Nutzen Sie dazu die konstanten Signale im Simulator unter „CIRCUIT ELEMENTS/Input/Constant Value“. Deren Wert kann mit einem Doppelklick angegeben werden.

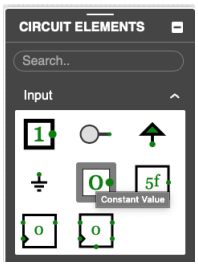


Abbildung 20: Signal mit konstantem Wert einfügen

3. Der Simulator stellt einen Addierer als eigenen Baustein zur Verfügung. In dieser Aufgabe soll dessen Funktionsweise erkundet werden.

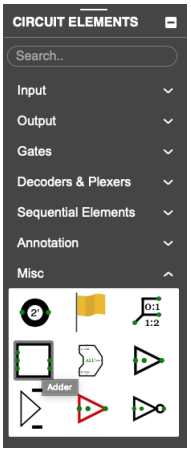


Abbildung 21: Eingebauter Addierer

   (a) Bauen Sie den Simulator-Addierer in einen Schaltkreis mit drei Input- und zwei Outputsignalen ein. Testen Sie die Funktionsweise des Addierers. Welche logische Schaltung, die Sie bereits gebaut haben, hat die gleiche Funktion?

(b) Der Simulator-Addierer funktioniert auch mit Signalen, die eine höhere Bandbreite aufweisen. Das kann man sich so vorstellen, dass In- und Outputsignale aus dickeren Leitungen bestehen, die mehrere einzelne Signale umfassen. Ein Inputsignal mit der Bandbreite vier entspricht dann vier einzelnen Inputbits.

Im Simulator kann die Bandbreite der In- und Outputs jedes Bauteils im „PROPERTIES“-Menu eingestellt werden:

Abbildung 22: Einstellung der Bandbreite beim eingebauten Addierer

Ändern Sie die Bandbreite der Inputs, Outputs und des Addierers, sodass damit die Funktion des 4-Bit-Addierers realisiert wird. Achtung: Nicht jeder In- und Output muss die Bandbreite vier aufweisen!

4. Fakultative Aufgabe:

   Der Computer, den Sie in Kapitel 6 bauen werden, wird mit 16-Bit-Zahlen rechnen, und Sie können den Simulator-Addierer mit Bandbreite 16 dafür verwenden. Es kann aber besonders befriedigend sein, jedes Bauteil für den Computer im Simulator selbst herzustellen.

   Entwerfen Sie im Simulator einen 16-Bit-Addierer und einen 16-Bit-Incrementer.

   Tipp: Bilden Sie zuerst einen 4-Bit-Volladdierer, der ähnlich wie der bereits gebaute 4-Bit-Addierer funktioniert, aber im Unterschied dazu zusätzlich einen 1-Bit-Input c_{in} entgegennimmt. Setzen Sie den 4-Bit-Volladdierer dann ein, um den 16-Bit-Addierer zu bauen.

Beim Entwurf des Schaltkreises für die Addition von 4-Bit-Zahlen haben wir ein Vorgehen gewählt, das in der Informatik sehr häufig zur Anwendung kommt. Das ursprüngliche Problem der Addition von 4-Bit-Zahlen haben wir zuerst in zwei unabhängige Teilprobleme gegliedert: die Addition der niedrigstwertigen und jene aller anderen Ziffern. Anschliessend haben wir für jedes dieser Teilprobleme ein separates Bauteil entworfen: den Halb- und den Volladdierer. Diese Bausteine haben wir schliesslich so kombiniert, dass damit das ursprüngliche Problem der Addition von 4-Bit-Zahlen gelöst wird.

Neue Konzepte und Begriffe : Der modulare Entwurf

Beim *modularen Entwurf* handelt es sich um eine Lösungsstrategie für komplexe Probleme. Dabei wird das Hauptproblem zunächst in unabhängige Teilprobleme unterteilt. Für jedes dieser Teilprobleme wird dann eine Lösung entworfen. Diese Teillösungen bilden schliesslich die Bauteile, aus denen die Lösung des Hauptproblems zusammengesetzt wird.

Diese Herangehensweise hat viele Vorteile. Zum einen ist die Lösung des Hauptproblems so viel übersichtlicher. Stellen Sie sich vor, wie kompliziert die Schaltung für den 4-Bit-Addierer wäre, wenn man alle darin vorkommenden Halb- und Volladdierer durch ihre innere Implementation aus XOR- und AND-Gatter ersetzen würde! Ausserdem kann man auf diese Art viel leichter allfällige Fehler finden, weil die einzelnen Bausteine separat voneinander getestet werden können, bevor sie zur Lösung des Hauptproblems kombiniert werden.

3.3 Negative Zahlen und Subtraktion

Wir haben schon einiges erreicht und konnten nachvollziehen, wie die Summe von Binärzahlen mit einer logischen Schaltung berechnet werden kann. Nun ist es an der Zeit, die Subtraktion anzugehen. Dabei müssen wir nicht wieder von vorne zu beginnen! Stattdessen nutzen wir die Tatsache, dass für zwei (Binär-)Zahlen a und b gilt:

$$a - b = a + (-b).$$

Für die Durchführung der Subtraktion könnten wir also den Addierer wiederverwenden. Zuerst muss jedoch die Frage geklärt werden, wie die Gegenzahl $(-b)$ von b in einer logischen Schaltung abgebildet wird, damit der Addierer auch beim Rechnen mit negativen Zahlen das richtige Resultat liefert.

Beim 4-Bit-Addierer haben wir die vier Inputs b_0, b_1, b_2 und b_3 als die vier binären Stellen der Zahl b interpretiert. So konnten wir mit Zahlen zwischen $0_{10} = 0000_2$ und $15_{10} = 1111_2$ rechnen. Neu werden wir mit dem Signal b_3 – der Ziffer ganz links – das Vorzeichen der Zahl angeben: Wenn $b_3 = 0$ ist, sollen die Signale b_2, b_1 und b_0 wie gewohnt eine Zahl zwischen 0 und 7 darstellen. Lauten die Inputsignale zum Beispiel $b_3b_2b_1b_0 = 0101$, so wird damit die nicht-negative ($b_3 = 0$) Zahl $101_2 = 5_{10}$ dargestellt.

Analog wird die die Bitfolge $b_3b_2b_1b_0 = 1001$ wegen $b_3 = 1$ eine negative Zahl darstellen, nämlich die Zahl -001_2. Es gibt aber ein Problem: Diese Codierung negativer Zahlen ist nicht mit dem 4-Bit-Addierer kompatibel! Wenn die Signale $a_3a_2a_1a_0 = 0101$ für die Zahl $101_2 = 5_{10}$ und $b_3b_2b_1b_0 = 1001$ für die Zahl $-001_2 = -1_{10}$ direkt an den Addierer geleitet würden, wäre der Output nicht wie erwartet 0100, was die Zahl $100_2 = 4_{10}$ codiert (siehe Abbildung 23).

In anderen Worten: Damit der Addierer das gewünschte Resultat erzeugt, müssen negative Zahlen zuerst in eine neue Darstellung umgewandelt werden. Wie aber müsste die neue Darstellung negativer Zahlen sein, damit sie mit dem Addierer kompatibel ist?

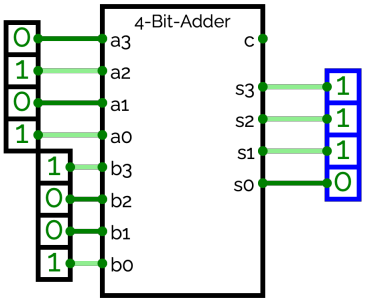


Abbildung 23: Addition zweier 3-Bit-Zahlen mit Vorzeichen

Aufgabe 3.5: Mit dem Addierer kompatible Darstellung negativer Zahlen

1. Öffnen Sie das Projekt „Arithmetik“ im Simulator und bauen Sie die Schaltung aus Abbildung 23 nach. Die Signale a_3, a_3, a_1, a_0 sollen wie in der Abbildung die Zahl $5_{10} = 0101_2$ codieren. Welchen Wert müssen die Signale b_3, b_2, b_1 und b_0 aufweisen, damit das Resultat $0100_2 = 4_{10}$ ist?

   Was wäre damit ein guter Kandidat für die Darstellung der Zahl -1 in einer logischen Schaltung?

2. Für die Darstellung der Zahl -2 können die Ziffern so gewählt werden, dass die Rechnung $2+(-2) = 0$ mit dem Addierer die richtige Lösung liefert. Finden Sie die fehlenden Ziffern:
   $$0010 + 1xyz = 0000.$$

3. Gehen Sie analog vor und bestimmen Sie die mit dem Addierer kompatible Codierung jeder Zahl $n \in \{-3, -4, -5, -6, -7\}$.

4. Dies ist eine herausfordernde Aufgabe: Vergleichen Sie die binäre Darstellung der Zahlen von 1 bis 7 mit der gefundenen Codierung ihrer Gegenzahlen. Erkennen Sie ein Muster, wie man aus der 4-Bit-Codierung einer Zahl zwischen 0 und 1 jene der Gegenzahl finden kann?

Neue Konzepte und Begriffe : Das Zweierkomplement

Die mit dem Addierer kompatible Codierung von negativen Zahlen nennt sich das *Zweierkomplement*. Dabei wird ein n-Bit-Signal aufgeteilt in ein Vorzeichenbit und $(n-1)$-Bits, die den Betrag der Zahl codieren.

$$\underbrace{0}_{\text{Vorzeichenbit}} \quad \underbrace{01011001\cdots}_{n \text{ Betragsbits}}.$$

Ist das Vorzeichenbit $= 0$, so wird eine nicht-negative Zahl codiert. Die $(n-1)$ folgenden Bits entsprechen dann der binären Darstellung der Zahl.

Beispiel: 8-Bit-Signal der Zahl 45:

$$00101101 = +0101101_2\,(= +45_{10}).$$

Ist das Vorzeichenbit $= 1$, wird eine negative Zahl codiert. Die binäre Darstellung des Betrags der Zahl findet man in diesem Fall, indem von den $(n-1)$ folgenden Bits zuerst 1 subtrahiert wird und anschliessend alle Bits invertiert werden.

Beispiel: 8-Bit-Signal der Zahl -81:

$$10101101 = -\overline{0101101_2 - 1_2} = -\overline{0101110}_2 = -1010001_2\,(= -81_{10}).$$

Aufgabe 3.6: Zweierkomplement

1. Die folgenden 8-Bit-Sequenzen sind Zweierkomplement-Codierungen von Zahlen. Um welche Zahlen handelt es sich? (Geben Sie die Antwort als Dezimalzahl an!)
   (a) 01011101
   (b) 11010110
   (c) 11111111

2. Geben Sie die 8-Bit-Zweierkomplement-Codierungen der folgenden Dezimalzahlen an:
   (a) 10
   (b) -10
   (c) -120
3. Welches ist die höchste Zahl, die man mit dieser Art der Codierung mit 8-Bits darstellen kann? Welches ist die kleinste?
4. Öffnen Sie das Projekt „Arithmetik“ im Simulator. Beginnen Sie einen neuen Schaltkreis mit dem Namen „Zweierkomplement“. Entwerfen Sie darin eine logische Schaltung, welche die Gegenzahl eines 4-Bit-Inputs berechnet; zum Beispiel: Der zum Input $a_3 = 0, a_2 = 1, a_1 = 1, a_0 = 1$ gehörende Output wäre $out_3 = 1, out_2 = 0, out_1 = 0, out_0 = 1$.
   Tipp: Sie können Ihre bereits entwickelten 4-Bit-Bauteile aus Aufgabe 3.4 wiederverwenden.
5. Entwerfen Sie im Projekt „Arithmetik“ einen „Subtrahierer“. Aus zwei 4-Bit-Inputs a und b soll der Subtrahierer den Output $out = a - b$ erzeugen.

Zusammenfassung

Wird die schriftliche Addition von Binärzahlen ziffernweise betrachtet, so kann sie als logische Verknüpfung interpretiert und durch eine logische Schaltung realisiert werden - den *Addierer*. Für die Konstruktion eines solchen Addierers kann das Problem aufgeteilt werden. Der *Halbaddierer* ist die logische Schaltung, mit welcher die niedrigstwertigen Bits verarbeitet werden. Der *Volladdierer* hingegen übernimmt die Verarbeitung der höherwertigen Stellen.

Bei der Konstruktion des Addierers wurde das Prinzip der Abstraktion angewandt: Einzelne logische Schaltungen wie der Halbaddierer oder der Volladdierer wurden gespeichert und in Form von „Sub-Circuits“ als Bauteile in komplexere Schaltungen wie den Addierer eingesetzt.

Das Zweierkomplement ist eine Codierung von ganzen Zahlen, die kompatibel mit dem Addierer ist. Dabei fungiert das erste Bit als Vorzeichenbit und gibt an, ob die codierte Zahl negativ ist. Die restlichen Bits bestimmen den Betrag der codierten Zahl. Bei nicht-negativen Zahlen entsprechen die Betragsbits der Binärdarstellung der Zahl. Bei negativen Zahlen kann der Wert nicht direkt aus den Betragsbits herausgelesen werden.

Mit dem Zweierkomplement kann der Addierer auch die Subtraktion von Zahlen durchführen.

4 Die arithmetisch-logische Einheit

Wir haben in den vergangenen Kapiteln nachvollzogen, wie man elektronische Schaltungen bilden kann, die in der Lage sind, logische und arithmetische Operationen durchzuführen. In diesem Kapitel werden wir alle bisher gebauten Logik-Bauteile zur *arithmetisch-logischen Einheit* (engl. arithmetic logic unit, kurz *ALU*) kombinieren. Diese logische Schaltung stellt einen Quantensprung auf unserem Weg in Richtung Computer dar. Wie die meisten Schaltungen, die wir bereits untersucht haben, wird die ALU auch zwei Inputsignale a und b verarbeiten. Allerdings wird sie zusätzlich ein Steuersignal *op* als Input erhalten, mit dem gewählt werden kann, welche Operation die ALU durchführen soll. In anderen Worten: Die ALU ist ein Bauteil, das Befehle entgegennehmen kann!

Die ALU und viele andere Bauteile im Computer, den wir in Kapitel 6 bauen werden, arbeiten häufig mit Signalen, die aus mehreren Bits bestehen. Weil sich die Breite der Signale häufig aus dem Kontext ergibt, werden wir sie in der Regel nicht explizit angeben. Manchmal aber kann es für das Verständnis hilfreich sein, zu wissen, aus wie vielen Bits die Signale bestehen. Dann werden wir die Signalbreite in eckigen Klammern angeben. Ein Signal a beispielsweise, das aus 16-Bits besteht, bezeichnen wir dann mit $a[16]$ und die einzelnen Bits eines solchen Signals mit $a_0, a_1, \ldots, a_{15}$.

Die beiden Inputsignale $a = a[16]$ und $b = b[16]$ der ALU bestehen je aus 16 Bits. Das Signal $op = op[5]$, mit welchem die auszuführende Operation definiert wird, besteht aus fünf Bits. Diese Informationen werden an die ALU geleitet, wo das Resultat $out[16]$ der Operation berechnet wird.

$$\underbrace{00100}_{\text{Operation } op}\quad\underbrace{0110101101110110}_{\text{Input } a}\quad\underbrace{0011101100110111}_{\text{Input } b}\rightarrow\boxed{\text{ALU}}\rightarrow\underbrace{0111101101110111}_{\text{Output } out}.$$

Das im Beispiel angegebene Steuersignal $op = 00100$ steht für die Operation OR. Die ALU erhält also den Befehl, die OR-Operation auf den beiden Inputsignalen a und b durchzuführen, und berechnet automatisch das Resultat $out = \text{OR}(a, b)$.

Vielleicht haben Sie schon einmal eine Assemblersprache kennengelernt? Dann wird Ihnen diese Art der Befehlseingabe bekannt vorkommen. Sie entspricht genau einem Assemblerbefehl der Form „or a,b“. Tatsächlich wird ein solcher Assembler-Befehl bei der Ausführung zuerst in Maschinensprache übersetzt. Dabei wird jeder Bestandteil des Befehls in eine Bitfolge umgewandelt. In unserem Computer wird der Assemblerbefehl „or“ in die Bitfolge 00100 umgewandelt werden. In diesem Kapitel geht es letztlich darum, dass wir die bereits konzipierten Bausteine so miteinander verbinden, dass mit einem Steuersignal *op* gewählt werden kann, welche Operation durchgeführt werden soll.

Alle Bauteile, die wir für die ALU benötigen, haben wir bereits kennengelernt. In diesem Kapitel werden wir allerdings auf die im Simulator verfügbaren Varianten unserer Bauteile zurückgreifen. Dadurch können wir die Bandbreite der Signale mühelos vergrössern und mit 16-Bit-Signalen arbeiten. Der innere Aufbau eines solchen 16-Bit-Addierers ist im Prinzip gleich wie jener des 4-Bit-Addierers (siehe auch die fakultative Frage in Übung 3.4).

Wie bereits bei der Entwicklung des Addierers werden wir schrittweise vorgehen und zuerst Subschaltungen entwerfen, die wir anschliessend in abstrahierter Form beim Bau der ALU wiederverwenden. Damit bleiben die Schaltungen übersichtlich und die Etappenziele erreichbar.

Die drei Hilfsschaltungen, welche die ALU bilden, sind:

- Die logische Einheit. Sie ist in der Lage, logische Operationen (z.B. AND) auf den beiden Inputs durchzuführen.
- Die arithmetische Einheit. Mit ihr werden arithmetische Operationen (z.B. die Addition) durchgeführt.
- Die Vorverarbeitungseinheit manipuliert die Inputsignale, bevor sie an die logische oder arithmetische Einheit weitergegeben werden.

Wir beginnen mit dem Entwurf der logischen Einheit. Deren Schnittstellen sehen wie folgt aus:

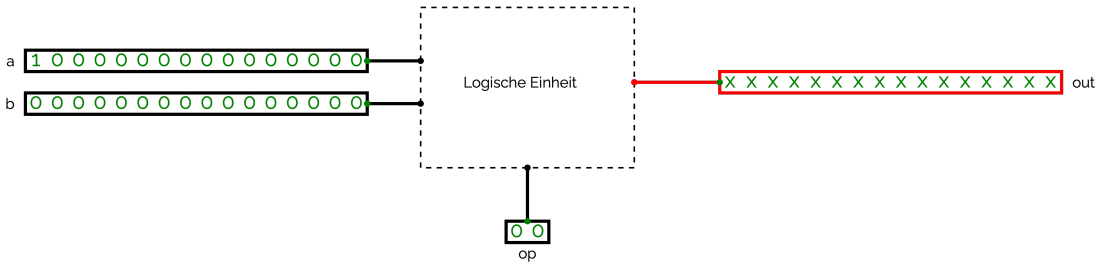


Abbildung 24: Schnittstellen der logischen Einheit

Insgesamt verfügt die logische Einheit über zwei 16-Bit-Inputsignale $a = a[16]$ und $b = b[16]$ und ein 2-Bit-Steuersignal $op = op[2]$. Letzteres entscheidet darüber, welche logische Funktion mit den beiden Inputs ausgeführt wird.

Neue Konzepte und Begriffe : Die logische Einheit

Die *logische Einheit (LU)* ist eine logische Schaltung mit zwei Inputsignalen a und b, einem 2-Bit-Steuersignal op und einem Outputsignal out. Das Steuersignal op definiert die Operation, die mit den Signalen a und b durchgeführt wird, nach folgender Tabelle:

op	out
00	AND(a, b)
01	OR(a, b)
10	XOR(a, b)
11	NEG(a)

Aufgabe 4.1: Bau der logischen Einheit

Auch in diesem Kapitel werden wir einige Bauteile in späteren Aufgaben wiederverwenden. Damit das funktioniert, sollten Sie ein neues Projekt mit dem Namen „ALU“ erstellen und fertiggestellte Schaltungen regelmässig speichern.

Bauen Sie eine logische Einheit im Simulator. Nutzen Sie dafür die im Simulator verfügbaren Bauteile:

- In- und Outputsignale der Bandbreite 16-Bit beziehungsweise 2-Bit für das Steuersignal.
- AND, OR, XOR und NEG mit Bandbreite 16-Bit.
- Multiplexer mit Bandbreite 16-Bit und Kontrollsignal der Breite 2-Bit.

Tipp: Konstruieren Sie die logische Schaltung so, dass jedes mögliche Resultat erzeugt wird und das Kontrollsignal bloss dazu dient, das gewünschte Resultat an den Output zu leiten.

Mit der logischen Einheit können wir nun steuern, welche logische Operation mit den Inputs durchgeführt wird. Analog dazu gibt es eine arithmetische Einheit, welche in der Lage ist, unterschiedliche arithmetische Operationen mit den Inputs durchzuführen.

Neue Konzepte und Begriffe : Die arithmetische Einheit

Die *arithmetische Einheit (AU)* ist eine logische Schaltung mit zwei Inputsignalen $a[16]$ und $b[16]$, einem 2-Bit-Steuersignal $op[2]$ und einem Outputsignal $out[16]$, die je nach Wert von op die folgende Operation durchführt:

op	out
00	$a + b$
01	$a - b$
10	$a + 1$
11	$a - 1$

Aufgabe 4.2: Bau der arithmetischen Einheit

Erstellen Sie im Projekt „ALU‘ einen neuen Schaltkreis mit dem Namen „AU“.

Bauen Sie darin eine arithmetische Einheit. Verwenden Sie dafür die im Simulator verfügbaren Bauteile:

- In- und Output mit Bandbreite 16-Bit beziehungsweise 2-Bit für das Steuersignal.
- Addierer mit Bandbreite 16-Bit. Diese sind in der Kategorie „Misc“ verfügbar.
- Zweierkomplement mit Bandbreite 16-Bit (auch unter „Misc“ verfügbar).
- Multiplexer mit Bandbreite 16-Bit und 2-Bit-Kontrollsignal.
- Konstante Signale (unter „Input“ verfügbar - der konstante Wert kann mit einem Doppelklick definiert werden).

Mit der logischen und der arithmetischen Einheit liesse sich bereits eine vollwertige ALU bilden. Allerdings werden wir davor noch eine *Vorverarbeitungseinheit* bauen, welche die Funktionalität

der arithmetisch-logischen Einheit weiter ausbaut.

Diese Vorverarbeitungseinheit ist eine logische Schaltung mit zwei Inputsignalen $a[16]$ und $b[16]$, zwei 1-Bit-Kontrollsignalen sw und za und zwei Outputsignaln $a_{out}[16]$ und $b_{out}[16]$.

Die Bezeichnung sw steht für „switch“: Wenn $sw = 1$ ist, werden die beiden Signale vertauscht.

Mit za (für „zero a“) wird *anschliessend* bestimmt, ob das Outputsignal out_a auf null gesetzt wird.

za	sw	a_{out}	b_{out}
0	0	a	b
0	1	b	a
1	0	0	b
1	1	0	a

Aufgabe 4.3: Bau der Vorverarbeitungseinheit

Öffnen Sie einen neuen Schaltkreis im Projekt „ALU“ mit dem Namen „PRE“. Entwerfen Sie darin eine Vorverarbeitungseinheit.

Mit den drei Bausteinen LU, AU und PRE können wir nun die arithmetisch-logische Einheit ALU bauen.

Aufgabe 4.4: Bau der ALU

1. Bauen Sie innerhalb des Projekts „ALU“ einen Schaltkreis „ALU“ nach folgendem Vorbild:

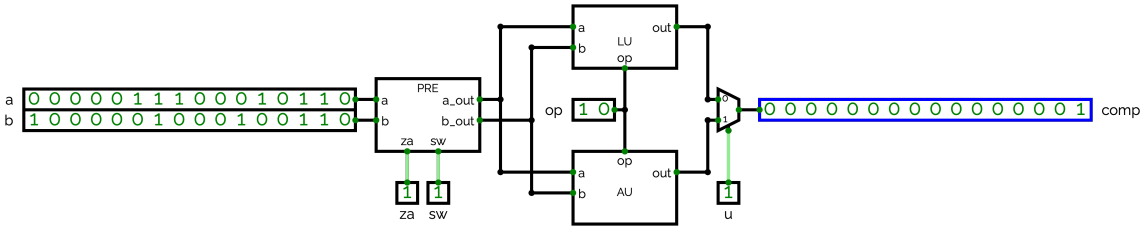


Abbildung 25: Aufbau der ALU

Nehmen Sie zur Kenntnis: Das Steuersignal u (für „unit“) entscheidet darüber, ob der Output der logischen oder der arithmetischen Einheit an den Output weitergeleitet wird. Das Outputsignal der ALU bezeichnen wir mit *comp* (für „computation“).

2. Vervollständigen Sie die Wahrheitstabelle der ALU:

u	op	za	sw	$comp$
0	00	0	0	$\text{AND}(a, b)$
0	00	0	1	
0	00	1	0	
0	00	1	1	
0	01	0	0	
0	01	0	1	
0	01	1	0	
0	01	1	1	
0	10	0	0	
0	10	0	1	
0	10	1	0	
0	10	1	1	
0	11	0	0	
0	11	0	1	
0	11	1	0	
0	11	1	1	
1	00	0	0	$a + b$
1	00	0	1	
1	00	1	0	
1	00	1	1	
1	01	0	0	
1	01	0	1	
1	01	1	0	
1	01	1	1	
1	10	0	0	
1	10	0	1	
1	10	1	0	
1	10	1	1	
1	11	0	0	
1	11	0	1	
1	11	1	0	
1	11	1	1	-1

Kontrolle: Einige Belegungen der Kontrollbits führen zum gleichen Output. Insgesamt sollte die ALU mit dieser Bauweise aber immerhin 19 unterschiedliche Operationen durchführen können.

Zusammenfassung : Arithmetisch-logische Einheit

Die ALU ist eine logische Schaltung, die in der Lage ist, arithmetische und logische Operationen mit zwei Inputsignalen durchzuführen. Mit einem Steuersignal kann festgelegt werden, welche Operation die ALU durchführen soll.

So wie wir die ALU konzipiert haben, besteht das Steuersignal aus den fünf Bits $u, op[2]za$ und sw. Wenn die ALU beispielsweise zwei Zahlen addieren soll, lautet das entsprechende Steuersignal: „10000“.

5 Speicher

In diesem Kapitel werden wir nachvollziehen, wie man mit Logikgattern einen Datenspeicher bilden kann. Bevor wir jedoch bis ins Detail gehend untersuchen, wie ein Speicher aus den uns bekannten Logik-Bausteinen gebaut werden kann, ist es gut, wenn wir zuerst klarstellen, was ein Speicher genau zu leisten hat: Unter einem Speicher verstehen wir eine Reihe von Speicherzellen – *Register* genannt -, die in der Lage sind, ein Signal von einem oder mehreren Bits Länge zu speichern. Jede dieser Speicherzellen besitzt eine eindeutige (binäre) *Adresse* innerhalb des Speichers. Mit Angabe der Adresse kann man auf die Inhalte der entsprechenden Speicherzelle zugreifen oder sie überschreiben. Dieses Prinzip ermöglicht so etwas wie die Verwendung von Variablen in einer Programmiersprache. Anstatt mit Variablennamen wird auf dieser Stufe jedoch mit der binären Adresse des entsprechenden Registers operiert.

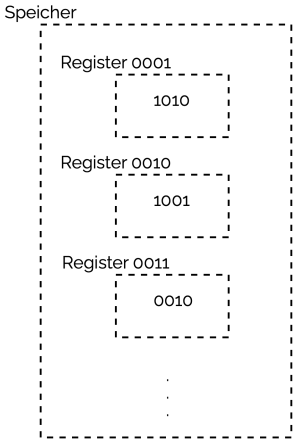


Abbildung 26: Schematische Darstellung eines Speichers als Folge von Registern mit einer Adresse. Im Register mit der Adresse 0001 ist die Bitfolge 1010 gespeichert.

Die Hauptfunktion eines Speichers ist die Konservierung eines Datensignals über die Zeit: Ein Signal kann im Speicher abgelegt und zu einem späteren Zeitpunkt wieder abgerufen werden. Die Zeit spielt also eine prominente Rolle, wenn es um die Modellierung eines Speichers geht. Bisher haben wir diese Dimension allerdings gänzlich ausgelassen, denn sie sorgt für einige Schwierigkeiten.

5.1 Das Problem mit der Zeit...

Bis anhin sind wir stillschweigend davon ausgegangen, dass für die Bearbeitung von Signalen in den Schaltungen keine Zeit nötig ist. In unserer Vorstellung ändert sich der Zustand des Outputsignals gewissermassen gleichzeitig und ohne Zeitverzögerung mit einer allfälligen Änderung am Input. In Wirklichkeit ist dies aber nicht der Fall. So benötigt die ALU zum Beispiel für die Ermittlung der Summe von zwei Inputsignalen a und b eine kleine Zeitspanne. Das Outputsignal wird dann erst

am Ende dieser Zeitspanne das richtige Resultat $a + b$ codieren.

Wird ein Outputsignal zu früh weiterverarbeitet, so stimmt das Resultat möglicherweise noch nicht, weil die Vorverarbeitung noch im Gange ist. Zu lange darf man mit dem Auslesen des Resultates aber auch nicht warten, sonst werden in der Zwischenzeit vielleicht bereits die nächsten Daten an die Inputsignale weitergereicht. Damit das Zusammenspiel aller Komponenten in einem Computer funktioniert, müssen diese nach einem strengen Zeitplan funktionieren. Sie müssen „getaktet“ sein!

Neue Konzepte und Begriffe : Das Taktsignal

Das Taktsignal (engl. *clock signal* oder kurz *clock*) ist der Dirigent im Computer, der dafür sorgt, dass alle Bauteile synchron arbeiten. Dieses 1-Bit-Signal springt regelmässig zwischen den beiden Werten 0 und 1 hin und her und definiert sogenannte *Taktzyklen*. Jeder Taktzyklus beginnt damit, dass der Wert des Taktsignals auf 0 springt. In der Mitte des Zyklus wechselt das Clocksignal auf 1 und mit dem erneuten Wechsel zu 0 wird schliesslich der nächste Taktzyklus eingeläutet.

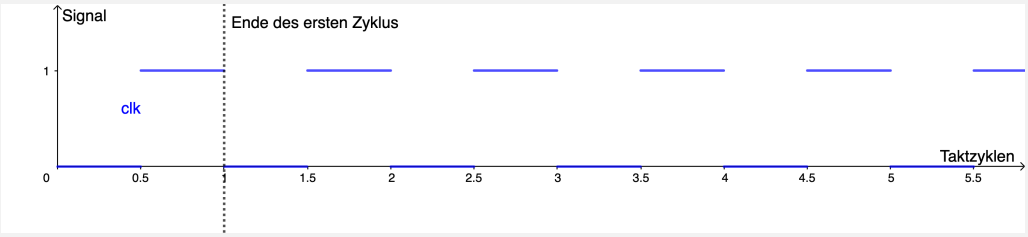


Abbildung 27: Das Taktsignal

Ein Taktzyklus muss mindestens so lange dauern, dass alle Bauteile genug Zeit haben, den gewünschten Endzustand zu erreichen. Nur am Ende des Zyklus besteht die Gewissheit, dass die Outputsignale stimmen. Jedoch ist es von Vorteil, die Länge der Taktzyklen so kurz wie möglich zu halten, damit der Computer in der Lage ist, mehr Berechnungen pro Zeiteinheit durchzuführen. Heutzutage haben Computer häufig Prozessoren mit einer Taktfrequenz von ungefähr 4 GHz. Der Taktzyklus hat dann eine unvorstellbar kurze Länge von einem Viertel einer Nanosekunde!

Weil die Verwertung der Signale in den Subschaltungen innerhalb der Taktzyklen noch im Gange ist, interessieren wir uns nur für den Zustand der Outputsignale, der jeweils am Ende eines Zyklus vorliegt. Wir werden die Zeit somit in diskreten Zeitschritten messen, wobei jeder Zeitschritt die Länge eines Taktzyklus aufweist. Die sich in der Zeit verändernden Werte des Outputs einer logischen Schaltung werden wir wiederum – wenn erforderlich – als Funktion der diskreten Zeitschritte auffassen und bezeichnen den Output nach n Zyklen mit $out(n)$.

Aufgabe 5.1: D-Latch und D-FlipFlop

In dieser Übung werden Sie ein erstes Speicher-Bauteil kennenlernen, das in der Lage ist, ein Signal für die Zeitspanne eines Taktzyklus zu konservieren.

1. Bauen Sie im Simulator die folgende Schaltung für den sogenannten D-Latch (Delay-Latch) nach:

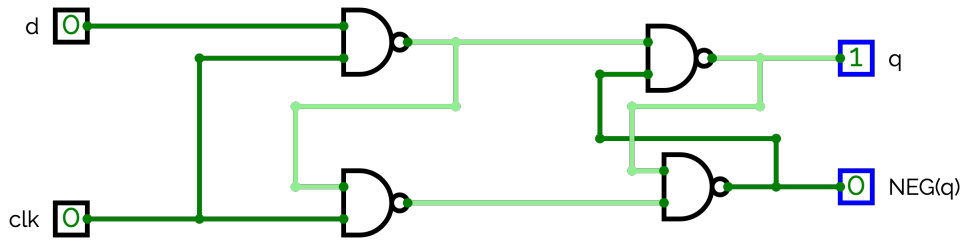


Abbildung 28: Logische Schaltung eines D-Latch

2. Manipulieren Sie die Inputwerte der logischen Schaltung, um das folgende Verhalten zu beobachten: Während der Wert der Clock (clk) 1 beträgt, wird der Wert des Outputs q auf jenem des Inputsignals d gesetzt. Dieser festgesetzte Wert wird über die 0-Phase der Clock konserviert – unabhängig davon, wie sich der Wert von d verändert.

3. Der Zustand der In- und Outputsignale in Abhängigkeit der Zeit kann im Simulator auch grafisch dargestellt werden. Bearbeiten Sie dazu die Schaltung aus Aufgabe 1 wie folgt:

   - Ersetzten Sie das Inputsignal für die Clock durch das Clock-Bauteil, das im Simulator unter „Sequential Elements“ verfügbar ist.
   - Zur Beobachtung der Zustände einzelner Signale sind sogenannte „Flags“ nötig. Diese finden Sie unter „Misc“. Führen Sie drei Flags ein: einen zur Beobachtung des clk-Signals, einen für das d-Signal und einen für das q-Signal.
     Die Schaltung sollte nun etwa so aussehen:

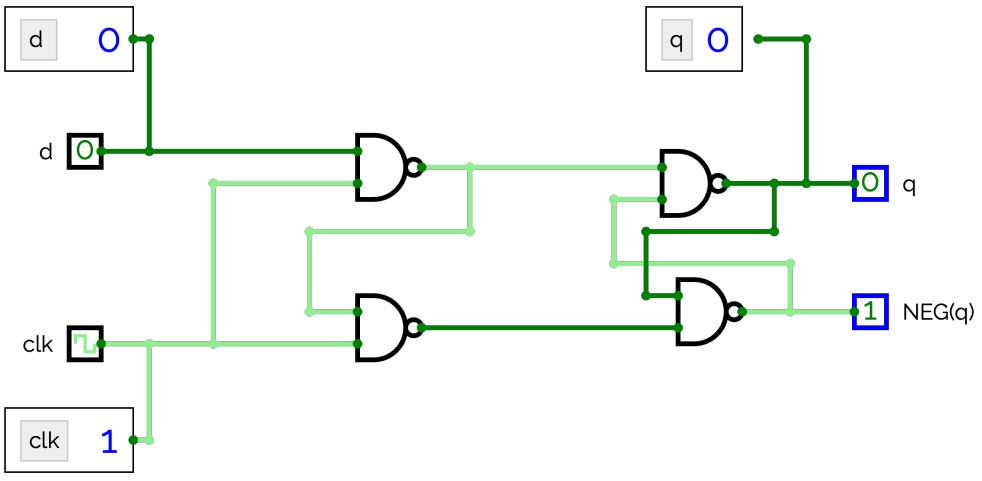


Abbildung 29: D-Latch mit Flags zur Beobachtung

4. Öffnen Sie nun das „TIMING DIAGRAM“ und beobachten Sie, wie sich der Wert von q verändert, wenn Sie d verändern.

5. Der Wert von q kann während der gesamten 1-Phase der Clock gesetzt werden. Aus diesem Grund kann der D-Latch ein Signal bloss für einen *halben* Taktzyklus konservieren – vom Ende der 1-Phase bis zum Beginn der nächsten 1-Phase der Clock. Um diese Zeitspanne auf einen ganzen Zyklus auszudehnen, wird das Clock-Signal vorverarbeitet. Bauen Sie die folgende Schaltung nach und beobachten Sie das Verhalten der Signale im „TIMING DIAGRAM“.

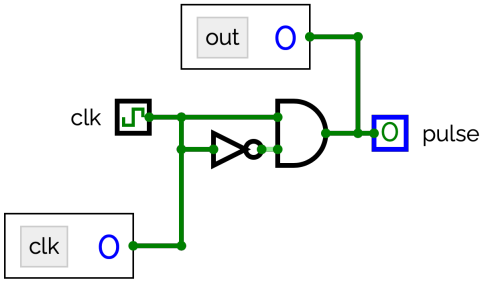


Abbildung 30: Pulser

Erklären Sie, wie es zum beobachteten Verhalten kommt.

6. Erklären Sie, wie man den D-Latch und den Pulse-Baustein kombinieren kann, um ein Speicher-Bauteil zu bilden, das ein Inputsignal über einen ganzen Taktzyklus konservieren kann.

7. Der Simulator ist nicht mächtig genug, um das Verhalten der Kombination der beiden Schaltungen aus Aufgabe 1. und 5. richtig zu simulieren. Stattdessen werden wir auf ein im Simulator vorgegebenes Bauteil zurückgreifen, das dieselbe Funktion aufweist: das D-FlipFlop. Sie finden es bei den „Sequential Elements“.
Bilden Sie die folgende Schaltung nach und prüfen Sie im „TIMING DIAGRAM“, dass das D-FlipFlop die Eigenschaft $out(n+1) = in(n)$ aufweist.

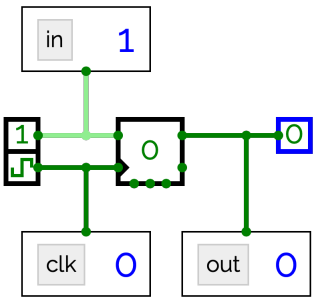


Abbildung 31: Schaltung, um das Verhalten des D-FlipFlops zu untersuchen

5.2 Adressierter Speicher

Mit dem D-FlipFlop haben wir einen Meilenstein erreicht: Wir sind nun in der Lage, den Wert eines Bits über einen Taktzyklus hinweg zu konservieren. Das scheint auf den ersten Blick nicht viel zu sein, aber in diesem Kapitel werden Sie sehen, dass das D-FlipFlop leicht zu einem Speicher erweitert werden kann, dar in der Lage ist, grosse Datenmengen über beliebig lange Zeitintervalle zu speichern.

Als Erstes werden wir das D-FlipFlop so erweitern, dass es Daten über längere Zeitspannen speichern kann.

Neue Konzepte und Begriffe : Register

Ein Register ist eine getaktete logische Schaltung mit zwei Inputsignalen *in* und *write* und einem Outputsignal *out*, wobei:

$$\text{out}(n+1) = \begin{cases} \text{out}(n) & \text{, falls write}(n) = 0 \\ \text{in}(n) & \text{, falls write}(n) = 1. \end{cases}$$

Die Funktion eines Registers kann so beschrieben werden: Wenn das *write*-Signal aktiviert wird, wird das Inputsignal im Register gespeichert. Danach wird das Register diesen gespeicherten Wert ausgeben, und zwar so lange, bis der Inhalt des Registers durch erneutes Setzen des write-Signals überschrieben wird.

Aufgabe 5.2: Register

Wieder werden wir in diesem Kapitel logische Schaltungen bilden, die aufeinander aufbauen. Damit dies möglich ist, sollten Sie im Simulator ein neues Projekt namens „Speicher“ eröffnen. Vergessen Sie nicht, das Projekt regelmässig zu speichern, damit Ihre Arbeit nicht verloren geht.

1. Bauen Sie ein 1-Bit-Register mit den folgenden Schnittstellen:

Abbildung 32: Schnittstellen des 1-Bit-Registers

Verwenden Sie dazu die folgenden im Simulator verfügbaren Elemente:

- Drei Inputsignale *in*, *write* und *clk* der Bandbreite 1. *Beachten Sie*: Für die spätere Wiederverwendung des Registers ist erforderlich, dass Sie für das Clock-Signal nicht den Clock-Baustein des Simulators, sondern einen ganz gewöhnlichen 1-Bit-Input benutzen.
- Ein 1-Bit-Outputsignal *out*.

- Ein Multiplexer und ein D-FlipFlop.

Tipp: Solange $write = 0$ ist, muss der Output des D-FlipFlops wieder an dessen Input geleitet werden, um den gespeicherten Wert über mehrere Taktzyklen zu speichern. Nutzen Sie den Multiplexer, um zu steuern, ob das *in*-Signal oder das *out*-Signal an das D-FlipFlop weitergeleitet wird.

2. Erstellen Sie eine neue logische Schaltung mit dem Namen „16-Bit-Register“. Diese soll genau gleich wie das 1-Bit-Register funktionieren, jedoch mit dem Unterschied, dass In- und Outputsignale die Bandbreite 16-Bit haben. Dazu benötigen Sie die folgenden Simulator-Bausteine:

   - Zwei Inputsignale $clk, write$ der Bandbreite 1.
   - Je ein In- und ein Outputsignal $in = in[16], out = out[16]$ der Bandbreite 16.
   - Sechzehn 1-Bit-Register.
   - Zwei Splitter, mit denen Sie das Inputsignal der Bandbreite 16 in sechzehn Signale der Bandbreite 1 aufteilen können – und umkehrt. Bei der Verwendung des Splitters müssen Sie die Bandbreite des Eingangssignals angeben (16) und die gewünschte Aufteilung in sechzehn Signale der Bandbreite 1 (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1).

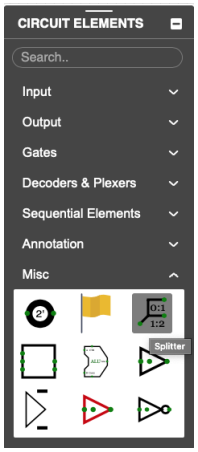


Abbildung 33: Bauteil Splitter

Tipp: In den Eigenschaften des Splitters können Sie unter „Direction“ wählen, in welche Richtung das Bauteil schaut.

Im nächsten Schritt werden wir mehrere Register zu einem adressierten Speicher kombinieren:

Neue Konzepte und Begriffe : Random Access Memory

Ein *Random Access Memory* oder kurz *RAM* ist eine getaktete logische Schaltung mit drei Inputsignalen *in*, *address* und *write* und einem Outputsignal *out*.

Der RAM besteht aus mehreren Registern, die über das *address*-Signal angesteuert werden können. Das Outputsignal *out* gibt den Wert aus, der in dem durch die Adresse definierten Register gespeichert ist. Wenn *write* auf 1 gesetzt ist, wird das Inputsignal in das Register gespeichert, das durch die Adresse definiert ist.

Zur besseren Vorstellung betrachten wir als Beispiel den groben Aufbau eines RAM mit vier 16-Bit-Registern.

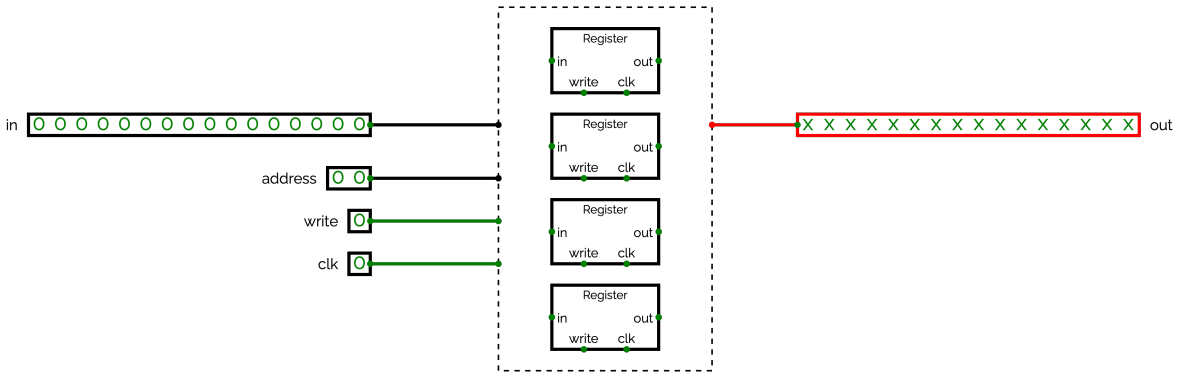


Abbildung 34: Aufbau eines RAM4

Damit die vier Register im Beispiel über das *address*-Signal angesteuert werden können, muss dieses die Bandbreite 2 aufweisen. Umgekehrt bestimmt die Bandbreite des Address-Signals die maximale Anzahl Register, die in dem RAM untergebracht werden können: Bei einer Adresse der Länge k können höchstens 2^k Register angesteuert werden.

Aufgabe 5.3: RAM

Bilden Sie alle Schaltungen in dieser Übung innerhalb des Projekts „Speicher“, in welchem Sie schon die Register implementiert haben.

1. Bilden Sie eine logische Schaltung „RAM8“, die einen adressierten Speicher mit acht 16-Bit-Registern realisiert.
   Tipp: Das Inputsignal kann gleichzeitig an alle Register weitergeleitet werden. Nutzen Sie das *address*-Signal zusammen mit Multiplexer und Demultiplexer mit Steuersignal der Breite 3, um zu steuern, welchem Register das *write*-Signal übermittelt und welches Outputsignal ausgegeben wird.
2. Kombinieren Sie nun acht RAM8-Bausteine, um einen „RAM64“ mit 64 Speicherplätzen zu bauen.
   Tipp: Mit dem Splitter können Sie das 6-Bit-Adress-Signal in zwei 3-Bit-Signale auf-

teilen. Mit den drei höherwertigen Bits können Sie dann auswählen, welcher der acht RAM8 angesprochen wird. Die drei niedrigstwertigen Bits definieren hingegen, welches Register innerhalb des gewählten RAM8-Bausteins angesteuert wird.

Zum Schluss des Kapitels bauen wir zwei weitere Bauteile, die innerhalb eines Rechners eine wichtige Rolle spielen:

Neue Konzepte und Begriffe : Programmzähler

Der *Programmzähler* ist eine getaktete logische Schaltung mit zwei 1-Bit-Inputsignalen $write$ und $reset$ und je einem 16-Bit-In- beziehungsweise Outputsignal $in[16]$ und $out[16]$, wobei für den Output nach $n+1$ Taktzyklen ($n \geq 0$) gilt:

$$out(n+1) = \begin{cases} 0 & \text{, falls } reset = 1 \\ in(n) & \text{, falls } write = 1 \\ out(n)+1 & \text{, falls } reset = write = 0. \end{cases}$$

Die Aufgabe des Programmzählers im Computer wird darin bestehen, Buch darüber zu führen, welcher Befehl des Programms als nächster durchgeführt werden soll. In vielen Situationen werden die Befehle der Reihe nach abgearbeitet. In diesen Fällen haben die beiden Inputsignale $reset$ und $write$ den Wert 0 und der im Befehlszähler gespeicherte Wert wird mit jedem Taktschritt um eins erhöht.

Kommt im Programm aber zum Beispiel eine Schleife vor oder wird eine Funktion aufgerufen, springt der Computer jeweils zu einer bestimmten Stelle im Programmcode. Einige Programmiersprachen kennen dafür den Befehl „goto“ oder „jmp“, mit dem man zu einem bestimmten Befehl springen kann. In diesem Fall wird $write = 1$ gesetzt und der Inputwert $in[16]$ gibt die Stelle an, an welche der Zähler springen soll.

Schliesslich besitzt der Programmzähler auch einen $reset$-Befehl, womit der Zähler auf null zurückgesetzt werden kann. In unserem Computer wird dieser Befehl so etwas wie einen Neustart des Computers bewirken.

Aufgabe 5.4: Programmzähler

Bauen Sie im Simulator einen Programmzähler. Nutzen Sie dabei auch den Addierer, der im Simulator unter „Misc“ zur Verfügung steht.

Wenn wir an eine While-Schleife wie „while index < 0 do ...“ denken, dann wird sofort klar, dass der Programmzähler häufig nur unter gewissen Bedingungen einen Sprung machen soll. Für unseren Computer werden wir dafür ein Bauteil entwerfen, das einen Input $comp$ (das Resultat einer ALU-Berechnung) auf drei Bedingungen prüft ($comp < 0, comp = 0, comp > 0$) und je nachdem den Befehl für einen Sprung ausgibt.

Dieses Bauteil nennen wir *Jumper* und wir verstehen darunter eine logische Schaltung mit einem 16-Bit-Inputsignal $comp[16]$, einem 3-Bit-Inputsignal $j[3]$ und einem 1-Bit-Outputsignal jmp, wobei

j	jmp
000	0
001	1, falls $comp > 0$
010	1, falls $comp = 0$
011	1, falls $comp \geq 0$
100	1, falls $comp < 0$
101	1, falls $comp \neq 0$
110	1, falls $comp \leq 0$
111	1

ist. In allen anderen Fällen soll jmp $= 0$ gesetzt werden.

Aufgabe 5.5: Jumper

Bauen Sie im Simulator einen Jumper. Verwenden Sie die im Simulator verfügbaren Splitter, um das $j[3]$-Signal aufzuteilen und das Vorzeichen von *comp* zu ermitteln.

Zusammenfassung

Die Aufgabe, ein Signal über die Zeit zu konservieren, wurde in diesem Kapitel in mehrere Teilprobleme aufgeteilt. Die Lösung jedes Teilproblems besteht in einer logischen Schaltung, welche die gewünschte Aufgabe übernimmt und in späteren Schaltungen wieder eingesetzt werden kann:

- Ein D-Latch kann ein Signal über einen halben Taktzyklus speichern.
- Ein D-FlipFlop kann ein Signal über einen ganzen Taktzyklus erhalten.
- Ein Register ist eine Speicherzelle, die ein Signal so lange konservieren kann, bis es überschrieben wird.
- Ein RAM besteht aus mehreren Registern, die über ein Adresssignal angesprochen werden können.

6 Der Computer

Mit den Speicher-Bauteilen aus Kapitel 5 und der arithmetisch-logischen Einheit aus Kapitel 4 haben wir die Hauptbestandteile eines Computers im Simulator bereits gebaut. In diesem Kapitel werden wir unsere Arbeit abschliessen und aus den bereits entwickelten Schaltungen einen programmierbaren Computer zusammensetzen!

Dabei gehen wir so vor, dass wir uns zuerst einen Überblick über den groben Aufbau des Computers verschaffen. Welche Bestandteile kommen darin vor? Welche Funktion erfüllen sie? Welcher Austausch findet zwischen den einzelnen Teilen statt? Wenn diese Fragen einmal geklärt sind, werden wir in die einzelnen Komponenten „hineinzoomen“. Wie müssen diese gebaut sein, damit sie ihre Funktion auf der übergeordneten Ebene erfüllen? Teilweise werden wir dann die einzelnen Komponenten wieder in mehrere Bestandteile gliedern, in die wir tiefer „hineinzoomen“, bis wir eine Ebene erreicht haben, in der die geforderten Funktionen mit den bereits gebauten Bauteilen realisiert werden können. Sobald wir soweit sind, „zoomen“ wir wieder heraus und verknüpfen dabei die gebauten Subschaltungen.

Viele der logischen Schaltungen, die Sie in den vorhergehenden Kapiteln entworfen haben, werden hier wieder benötigt. Für den Fall, dass in der Zwischenzeit etwas verloren gegangen ist, finden Sie hier alle Bauteile, auf die wir in diesem Kapitel aufbauen werden: Vorlage für den Bau des Computers. Um die Vorlage zu kopieren, müssen Sie mit einem Benutzerkonto bei `https://circuitverse.org` angemeldet sein.

6.1 Der innere Aufbau des Computers

Der Computer, den wir in diesem Kapitel bauen werden, besteht im Wesentlichen aus drei Komponenten: einem Datenspeicher, einem Befehlsspeicher und einem Hauptprozessor (CPU) (siehe Abbildung 35)

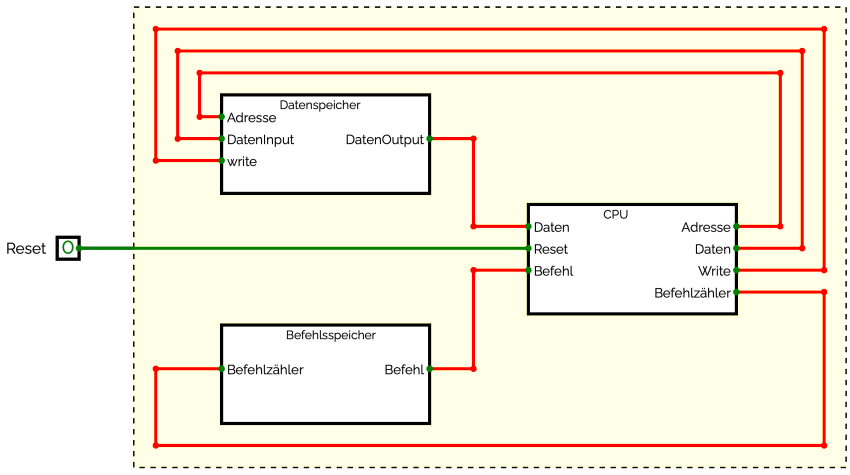


Abbildung 35: Aufbau und Hauptkomponenten des Computers

Neue Konzepte und Begriffe

Diesen Aufbau mit der Trennung von Daten- und Befehlsspeicher nennt man eine *Harvard-Architektur*. Der Hauptvorteil dieses Aufbaus besteht darin, dass Befehle und Daten gleichzeitig in die CPU geladen werden können. Das macht den inneren Aufbau der CPU schlanker und übersichtlicher.

Auf der anderen Seite ist die strikte Trennung von Daten und Befehlen natürlich auch einschränkend. Neben der unflexiblen Handhabung von Speicherplatz liegt ein grosser Nachteil darin, dass die CPU keine neuen Befehle in den Speicher schreiben kann. Aus diesem Grund verwenden die meisten Computer heute die sogenannte *Von-Neumann-Architektur*, bei welcher Daten und Befehle im gleichen Speicher abgelegt werden.

Die angesprochenen Nachteile der Harvard-Architektur werden wir im Rahmen unseres Projektes nicht zu spüren bekommen, die Vorteile – insbesondere den schlanken, übersichtlichen Aufbau – dafür umso mehr. Aus diesem Grund eignet sich die Harvard-Architektur ausgezeichnet für unser Vorhaben.

Werfen wir nun einen Blick auf die einzelnen Komponenten und deren Funktion.

Der *Datenspeicher* ist ein gewöhnlicher RAM-Baustein, wie Sie ihn im Kapitel 4 konstruiert haben. Der Hauptprozessor kann Daten im Datenspeicher lesen und schreiben. Der Einfachheit halber werden wir sowohl für die einzelnen Register im Speicher wie auch für das Adress-Signal eine Bandbreite von 16-Bits verwenden. Der Datenspeicher wird somit insgesamt $2^{16} = 64$k Speicherplätze à 16-Bit umfassen, was insgesamt eine Speicherkapazität von 128 kByte bedeutet.

Für den *Befehlsspeicher* werden wir auch einen RAM-Baustein mit der gleichen Speicherkapazität wie des Datenspeichers verwenden. Im Unterschied zum Datenspeicher können die Inhalte des Befehlsspeichers von der CPU zwar gelesen, aber nicht beschrieben werden. Jeder Befehl, der im Befehlsspeicher abgelegt ist, besteht aus einem 16-Bit-Signal.

Der *Hauptprozessor* besteht im Wesentlichen aus der arithmetisch-logischen Einheit zusammen mit zwei 16-Bit-Registern, die wir mit $A = A[16]$ (für Adresse) und $D = D[16]$ (für Daten) bezeichnen, und einem 16-Bit-Programmzähler. Grob gesagt sorgt der Hauptprozessor dafür, dass die ALU die im aktuellen Befehl definierte Operation ausführt, wobei die Signale in den beiden Registern oder im Datenspeicher als Input dienen. Der Programmzähler führt Buch darüber, welcher Befehl aus dem Befehlsspeicher als Nächstes ausgeführt werden muss.

Zusätzlich zu diesen drei inneren Komponenten verfügt der Computer ausserdem über ein *Reset-Inputsignal*. Dieses kann man sich wie eine Art Neustart-Knopf vorstellen, der dem Computer das Signal gibt, die Berechnung von vorne zu beginnen.

Wie wird nun ein Programm auf dem Computer ausgeführt? Alles beginnt mit einem Wechsel des Reset-Signals von 1 auf 0. Dann beginnt der Computer damit, dass er den Befehl im Speicherplatz mit der Adresse 0_{16} des Befehlsspeichers der CPU übergibt. Jeder Befehl besteht aus einem 16-Bit-Signal. Darin codiert ist die Information, aus welcher Quelle Signale an die ALU gereicht werden, welche Operation die ALU ausführen und wo das Resultat abgelegt werden soll. Typische Befehle können die folgende Bedeutung haben: „Schreibe die Zahl 42 ins Register A“ oder „Bilde die Summe aus den Zahlen, die im Register A und D gespeichert sind, und schreibe das Resultat ins Register

D“ oder „Schreibe den Inhalt von Register D in den Datenspeicher an die Adresse, die im Register A gespeichert ist“. Nachdem der Befehl ausgeführt wurde, wird der Programmzähler in der Regel um eins erhört, wodurch der nächste Befehl aus dem Befehlsspeicher an die CPU übergeben wird.

Im Hinblick auf die Implementation des Computers im Simulator müssen wir uns über Daten- und Befehlsspeicher keine Sorgen machen. Hierfür können wir die bereits entwickelten Speicherbausteine verwenden. Der innere Aufbau der CPU ist jedoch komplexer. Zoomen wir also hinein!

6.2 Der innere Aufbau der CPU

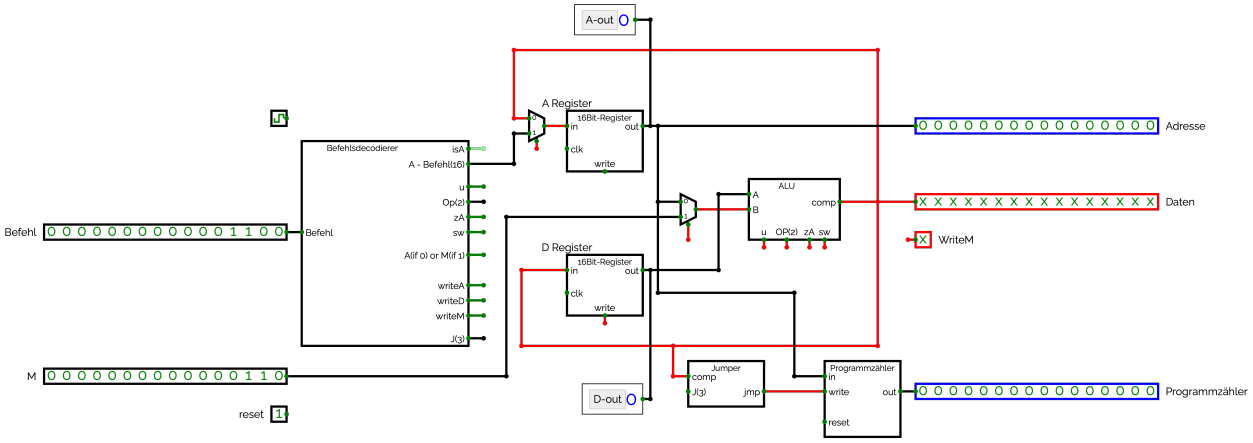


Abbildung 36: Innerer Aufbau des Hauptprozessors

Die Abbildung 36 zeigt den inneren Aufbau des Hauptprozessors. Für eine bessere Übersicht über die wichtigsten Komponenten wurden in der Abbildung einige Verbindungen weggelassen.

Ganz links in der Abbildung sind die drei Inputsignale des Hauptprozessors zu sehen. Das sind zum einen der 16-Bit-Befehl aus dem Befehlsspeicher, zum anderen der Inhalt $M(A)$ der Speicherzelle mit der Adresse A im Datenspeicher und schliesslich auch das Resetsignal. Der Befehl wird zuerst an den Befehlsdecodierer geleitet. Hier wird das 16-Bit-Signal interpretiert und in seine Komponenten aufgeteilt. Diese Befehlsteile werden dann an die weiteren Komponenten der CPU verteilt.

Der Prozessor verfügt ausserdem über zwei 16-Bit-Register, die wir A und D nennen. Häufig werden wir diese Register so verwenden, dass der Inhalt von Register D als Daten interpretiert wird und jener von A als Adresse im Daten- oder Befehlsspeicher.

Die ALU spielt im Hauptprozessor eine wichtige Rolle. Sie verarbeitet gemäss dem aktuellen Befehl die Daten aus den Registern oder aus dem Datenspeicher. Der Output der ALU wird an verschiedenen Stellen weitergeleitet: an die beiden Register A und D, an den Datenspeicher und an den Jumper.

Die beiden Bauteile unten rechts in der Übersicht definieren schliesslich, welcher Befehl aus dem Befehlsspeicher als Nächstes durchgeführt werden soll. Häufig wird der Programmzähler bloss um eins erhöht, wodurch die Befehle im Befehlsspeicher der Reihe nach ausgeführt werden. Unter gewissen Umständen – zum Beispiel, wenn das Programm eine Schleife beinhaltet – kann der

Zähler jedoch zu der im Register A definierten Adresse springen. Ob ein Sprung vollzogen werden soll, wird im Bauteil „Jumper“ ermittelt, das Sie in Übung 5.5 bereits gebaut haben. Dieses prüft, ob der Output der ALU die Sprungvoraussetzungen erfüllt, die im aktuellen Befehl vorgegeben werden.

ALU, Register, Jumper und den Programmzähler haben wir alle schon in den vorangegangenen Kapiteln entwickelt. Für den Bau des Hauptprozessors benötigen wir noch den Befehlsdecodierer. Wir zoomen also noch eine Stufe weiter hinein...

6.3 Die 16-Bit-Befehle und der Befehlsdecodierer

Bevor wir den inneren Aufbau des Befehlsdecodierers angehen können, müssen wir klären, wie die 16-Bit-Befehle aufgebaut sind und wie unser Computer diese interpretieren soll. Wir müssen im Grunde also eine gemeinsame Sprache zwischen uns und dem Computer festlegen – die sogenannte *Maschinensprache*. Dabei müssen wir berücksichtigen, dass der Computer bloss 16-Bit-„Wörter“ verstehen kann. Ein konkretes Beispiel wäre der Befehl 1111010000010000. Laut der Definition der Sprache, die wir unten vorstellen werden, bedeutet dieser Befehl etwa: „Bilde die Summe aus den Zahlen, die im Register A und D gespeichert sind, und schreibe das Resultat ins Register D“.

Neue Konzepte und Begriffe : Maschinensprache und Befehlssatz

Die Menge der Befehle, die von einer CPU ausgeführt werden können, nennt man den *Befehlssatz* des Prozessors. Diese Befehle werden als Binärcode – in der sogenannten *Maschinensprache* – durch die CPU gelesen und verarbeitet.

Offensichtlich sind Hardware, Befehlssatz und Maschinensprache stark voneinander abhängig. Der Befehlssatz wird nur Befehle enthalten, die von der CPU innerhalb eines Taktzyklus ausgeführt werden können. Ausserdem wird bei vielen Befehlen ein Teil des 16-Bit-Codes aus den fünf Kontrollbits für die ALU bestehen.

Für den Befehlsdecodierer ist der letzte Punkt besonders wichtig. In dieser Schaltung sollen die Befehle in deren Bestandteile aufgetrennt und an die richtigen Stellen geleitet werden. Dessen Aufbau und Verknüpfung mit den anderen Elementen des Hauptprozessors wird dann sicherstellen, dass der Computer die Befehle gemäss deren Bedeutung in der Maschinensprache korrekt umsetzt.

Grundsätzlich werden wir zwei Befehlsarten unterscheiden. Einen 16-Bit-Befehl, der mit einer 0 beginnt, nennen wir α-Befehl. Dieser dient dazu, eine Konstante (häufig eine Adresse des Datenspeichers) in das A-Register zu schreiben.

Beginnt der 16-Bit-Befehl mit einer 1, sprechen wir hingegen von einem β-Befehl. Diese Befehle codieren meistens eine Berechnung, die von der ALU durchgeführt werden soll.

6.3.1 α-Befehle

Ein α-Befehl hat in der Maschinensprache unserer CPU die Form

$$0xxxxxxxxxxxxxx.$$

Bei einem solchen Befehl soll der Hauptprozessor das gesamte 16-Bit-Signal (samt der führenden 0) in das A-Register speichern.

Die Aufgabe des Befehlsdecodierers ist es, bei einem α-Befehl das Outputsignal $is\alpha$ auf 1 zu setzen und das 16-Bit-Eingangssignal an den Output α-Befehl[16] weiterzuleiten. Alle anderen Outputsignale sollen auf 0 gesetzt werden.

Aufgabe 6.1: Decodierung der α-Befehle

Starten Sie im Simulator ein neues Projekt mit dem Namen „Computer“. Fertigen Sie darin einen ersten Entwurf für den Befehlsdecodierer mit einem 16-Bit-Inputsignal $Befehl$ und zwei Outputsignalen $is\alpha$ (1-Bit) und α-$Befehl$ (16-Bit).

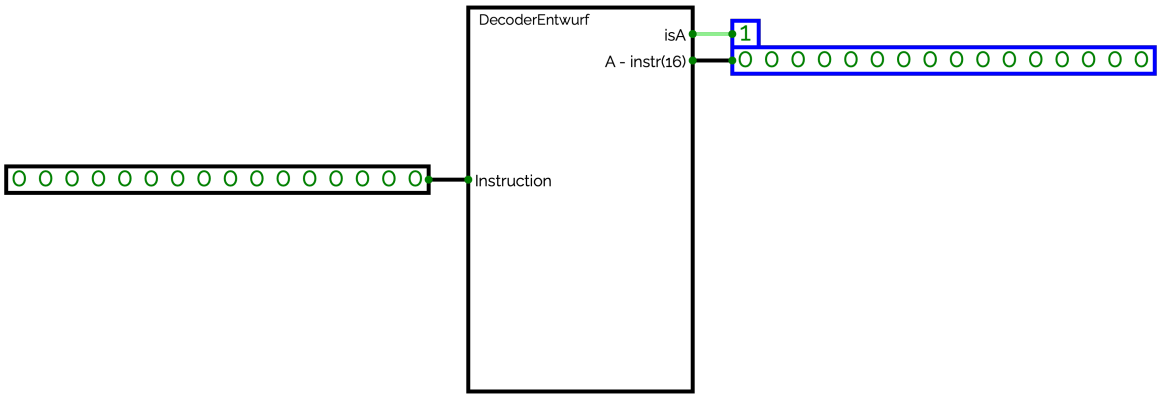


Abbildung 37: Schnittstellen des Entwurfs des Befehlsdecodierers

Falls es sich beim eingehenden Befehl um einen α-Befehl handelt, soll $is\alpha = 1$ und α-$Befehl = Befehl$ gesetzt werden. Andernfalls sollen beide Outputsignale 0 sein.

6.3.2 β-Befehle

Beginnt der 16-Bit-Befehl mit einer 1, so handelt es sich um einen β-Befehl. Bei dieser Art Befehl soll die ALU eine Berechnung durchführen. Die Bits im Befehl definieren, aus welcher Quelle die Signale an die ALU weitergeleitet werden, welche Operation durchgeführt wird, wo das Resultat gespeichert wird und unter welchen Bedingungen der Programmzähler neu gesetzt wird. Der Aufbau eines β-Befehls ist wie folgt:

$$1\,111\,s\,ccccc\,zzz\,jjj.$$

Das erste Bit ganz links identifiziert den Befehl als β-Befehl. Die darauffolgenden drei Bits werden nicht verwendet. Der Übersicht halber legen wir fest, dass sie im Falle eines β-Befehls immer 1 betragen.

Das fünfte Bit von links bezeichnen wir mit s für Quelle (source). Mit diesem Bit legen wir fest, welche Signale von der ALU verarbeitet werden sollen. Betrachten Sie nochmals die Abbildung 36. Der Input a der ALU ist stets das Signal, das im Register D gespeichert ist. Das Signal für den Input b der ALU kann aber aus Register A stammen oder aus der Speicherzelle im Datenspeicher an der Addresse A (Signal $M(A)$). Wenn im Befehl das s-Bit gesetzt ist, wird $M(A)$ verarbeitet, andernfalls A.

Die nächsten 5 Bits (mit c für computation bezeichnet) definieren die Operation, die von der ALU durchgeführt werden soll (vgl. Tabelle in Übung 4.4):

$$ccccc = u, op_1, op_0, za, sw.$$

Die hinteren sechs Bits des Befehls codieren, was nach der ALU-Berechnung geschehen soll.

Mit den drei Bits zzz legt man fest, an welchen Stellen das Resultat gespeichert wird. Ist das erste z-Bit gesetzt, wird der Output der ALU ins Register A geschrieben. Das zweite z-Bit legt fest, ob das Resultat in Register D gespeichert wird, und das dritte, ob es im Datenspeicher an der in Register A angegebenen Adresse abgelegt wird. Es sind auch Kombinationen möglich: Bei $zzz = 101$ wird die Berechnung sowohl im Datenspeicher $M(A)$ als auch in Register A gespeichert.

zzz	write A	write D	write M
000	0	0	0
001	0	0	1
010	0	1	0
011	0	1	1
100	1	0	0
101	1	0	1
110	1	1	0
111	1	1	1

Die letzten drei Bits des Befehls beinhalten die Information, unter welchen Umständen der Programmzähler einen Sprung macht. Entscheidend ist dabei der Wert des Outputs *comp* der ALU. Falls $jjj = 100$, springt der Programmzähler, wenn $comp < 0$ ist. Bei $jjj = 010$ im Falle $comp = 0$ und bei $jjj = 001$ springt er, falls $comp > 0$ ist. Es sind aber auch folgende Kombinationen möglich: Bei $jjj = 101$ springt der Zähler, falls der Output der ALU < 0 *oder* > 0 ist. Der Jumper, den Sie in Aufgabe 5.5 gebaut haben, prüft genau diese Bedingungen.

Damit ist die Maschinensprache vollständig beschrieben. Eine Auflistung aller Befehle finden Sie im Anhang 7.7.

Aufgabe 6.2: Befehlsdecodierer

Ergänzen Sie den Entwurf des Befehlsdecodierers aus der vorangegangenen Übung, sodass β-Befehle richtig decodiert werden.

Bei einem β-Befehl sollen die Outputs $is\alpha = 0$ und $\alpha\text{-}instr[16] = 0$ gesetzt werden. Ausserdem soll in diesem Fall der Befehl in seine Bestandteile zerlegt und an die entsprechenden Outputs

gemäss der Abbildung weitergeleitet werden. Achten Sie insbesondere auf die Bandbreite der Outputsignale.

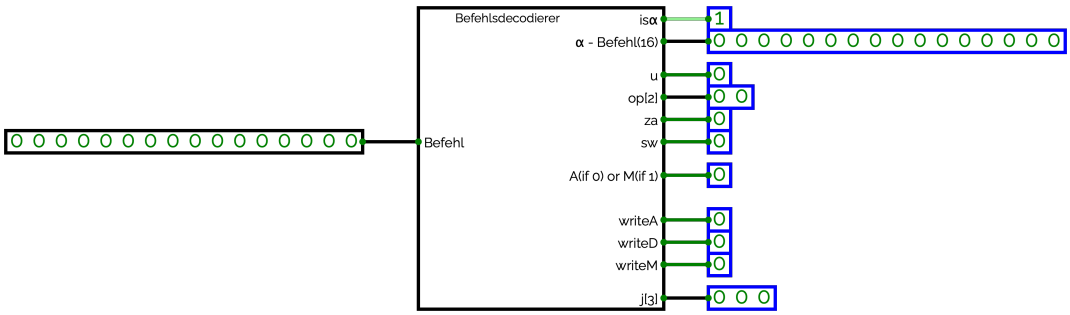


Abbildung 38: Schnittstellen des Befehlsdecodierers

6.4 Der Bau der CPU

Mit dem Befehlsdecodierer haben wir das letzte Puzzlestück zum Bau des Computers hergestellt. Nun ist es an der Zeit, wieder „rauszuzoomen“ und Stück für Stück die Elemente so zu verbinden, das der Computer entsteht.

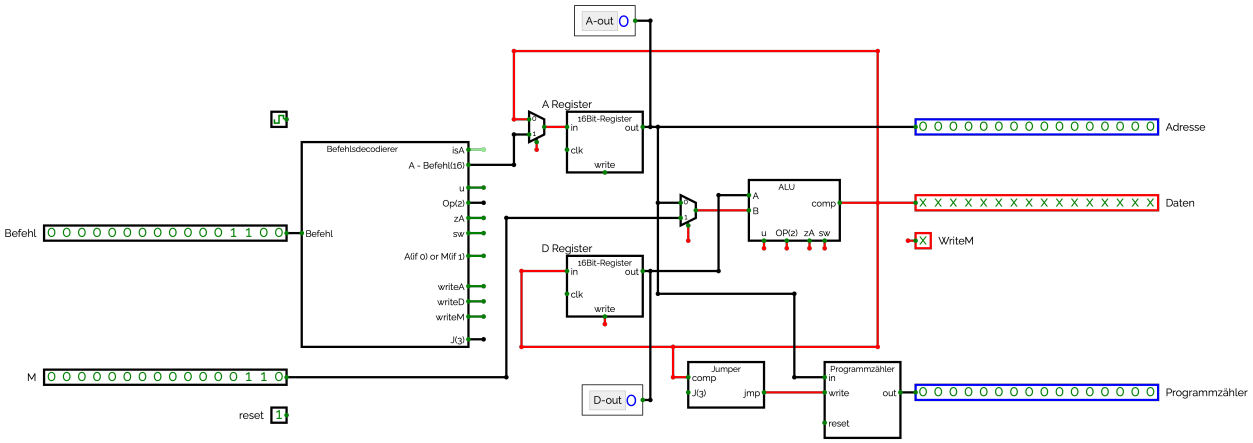


Abbildung 39: Der innere Aufbau der CPU

Aufgabe 6.3: Bau der CPU

In dieser Übung bauen Sie den Hauptprozessor des Computers:

1. Im Hauptprozessor kommen einige Bauteile zum Einsatz, die Sie bereits in anderen

Projekten gebaut haben:

- ALU
- 16-Bit-Register
- Programmzähler
- Jumper

Kopieren Sie diese Elemente aus den jeweiligen Projekten „ALU “ und „Speicher“ in das aktuelle Projekt „Computer“. Dazu müssen Sie in jedem benötigten Schaltkreis (z.B. „ALU“) mit ctrl-A / ctrl-C alles kopieren und in einen neuen Schaltkreis im Projekt „Computer“ mit ctrl-V einfügen. Allfällige Subschaltkreise werden automatisch mitimportiert.
Für den Fall, dass in der Zwischenzeit etwas verloren gegangen ist, finden Sie hier alle Bauteile, auf die wir in diesem Kapitel aufbauen: Vorlage für den Bau des Computers.

2. Setzen Sie den Hauptprozessor gemäss Abbildung 39 oben zusammen und ergänzen Sie die fehlenden Verbindungen.

3. Ergänzen Sie die Schaltung so wie in der Abbildung vorgegeben um zwei Flags, welche den Output der beiden Register überwachen. Wir werden sie nutzen, um die korrekte Funktionsweise der CPU zu testen.

Bevor wir weiter herauszoomen und auf der obersten Ebene den Hauptprozessor mit den beiden Speichern verbinden, um den Computer fertigzustellen, kontrollieren wir mit einem kleinen Testprogramm, ob die CPU richtig funktioniert. Das Testprogramm besteht aus vier 16-Bit-Befehlen und soll die Summe der beiden Zahlen 2 und 3 berechnen.

Das Testprogramm lautet:

```
0000000000000010 # A = 2
1111000110010000 # D = A
0000000000000011 # A = 3
1111010000010000 # D = A + D
```

In den Kommentaren wird schon angedeutet, was die einzelnen Befehle leisten. Aber schauen wir uns im Detail an, wie sie die CPU interpretiert (vgl. auch die Tabellen in Abschnitt 7.7):

- 0000000000000010.
  Wegen der führenden Null handelt es sich hierbei um einen α-Befehl. Die 16-Bit-Sequenz, die in diesem Fall die Dezimalzahl 2 codiert, wird im A-Register gespeichert werden.

- 1111000110010000.
  - **1** 111000110010000. Die führende 1 identifiziert den Befehl als β-Befehl.
  - 1 **111** 000110010000. Die drei folgenden 1 werden nicht verwendet.
  - 1111 **0** 01001010000. Das s-Bit zeigt an, dass das Signal von Register A (und nicht jenes aus dem Speicher M) zusammen mit jenem aus Register D an die ALU weitergeleitet wird.

  - 11110 **00110** 010000. Diese fünf Bits geben an, welche Operation die ALU durchführen soll. Gemäss der ALU-Wahrheitstabelle (vgl. Aufgabe 4.4) wird der ALU-Input b ausgegeben. Das entspricht dem Wert des A-Registers.
  - 1111000110 **010** 000. Hier wird codiert, wo der Output der ALU gespeichert werden soll. In diesem Fall in Register D.
  - 1111000110010 **000**. Die letzten drei Bits zeigen an, dass der Programmzähler nicht springen soll. Damit wird als Nächstes der folgende Befehl ausgeführt.

- 0000000000000011.
  Die Dezimalzahl 3 wird in Register A gespeichert.

- 1111010000010000.
  Die ALU addiert die Werte aus den Registern A und D und speichert das Resultat in Register D.

Aufgabe 6.4: Test der CPU

In dieser Aufgabe prüfen Sie, ob Ihre CPU das Testprogramm korrekt ausführt.

Schalten Sie dazu die Clock auf manuellen Betrieb um, indem Sie Clock Enabled ausschalten:

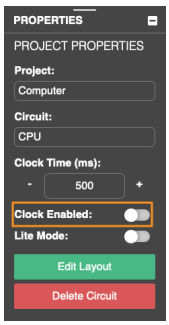


Abbildung 40: Einstellung zum manuellen Betrieb des Clock-Signals

Sie können nun die Clock von Hand steuern, indem Sie auf das entsprechende Element im Schaltkreis klicken.

Setzen Sie nun das Clock-Signal auf 0 und stellen Sie das Inputsignal „Befehl“ so ein, dass es dem ersten 16-Bit-Befehl des Testprogramms entspricht. Klicken Sie anschliessend zweimal auf die Clock, damit ein Taktzyklus absolviert wird.

Wenn alles korrekt ausgeführt wurde, sollte die Flag „A-out“ anzeigen, dass die Zahl 2 im A-Register gespeichert ist.

Fahren Sie mit den nächsten Befehlen des Testprogramms fort und kontrollieren Sie jedes Mal mit den Flags, ob der Befehl korrekt ausgeführt wurde.

6.5 Der Bau des Computers

Zum Schluss zoomen wir eine weitere Stufe heraus und bauen die letzte Schaltung – den Computer.

Aufgabe 6.5: Bau des Computers

Bauen Sie den Computer im Simulator:

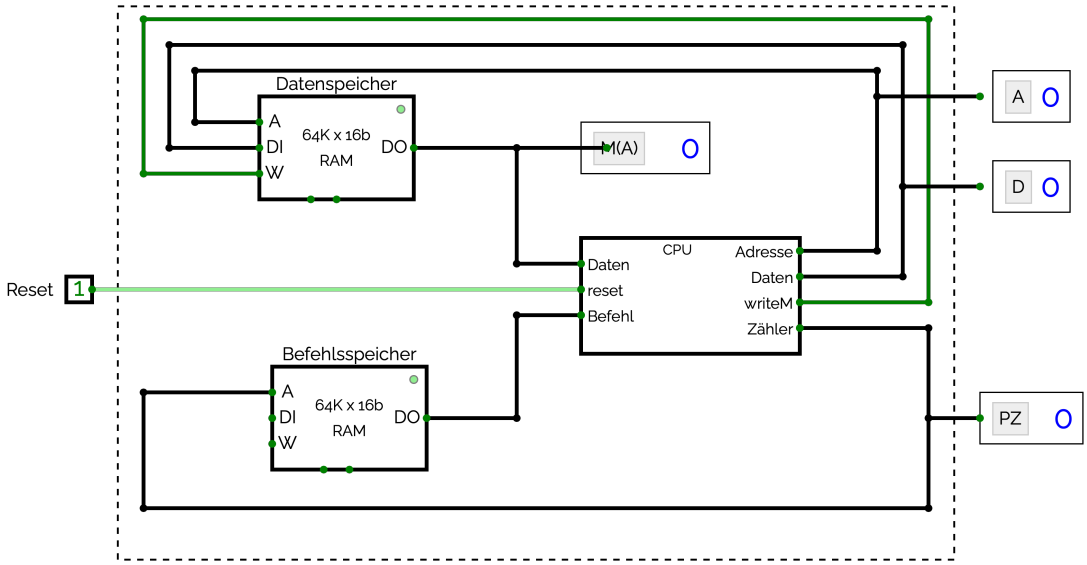


Abbildung 41: Innerer Aufbau des Computers

Verwenden Sie für die beiden Speicher die RAM-Bausteine, die im Simulator unter „Sequential Elements“ zur Verfügung stehen.

Setzen Sie die vier Flags, um die Signale überwachen zu können.

Wieder wollen wir anhand eines kleinen Programms testen, ob der Computer ordnungsgemäss funktioniert. Dazu verwenden wir erneut das Programm, das wir schon zum Testen der CPU eingesetzt haben. Allerdings ergänzen wir es um einige Zeilen, damit auch das Zusammenspiel mit den beiden Speichern überprüft werden kann. Das Testprogramm lautet:

```
0000000000000010 # A = 2
1111000110010000 # D = A
0000000000000011 # A = 3
1111010000010000 # D = A + D
0000000000000000 # A = 0
1111000111001000 # M = D
0000000000000111 # A = 7
1111000000000111 # Jump!
```

Die Kommentare geben an, was der jeweilige 16-Bit-Befehl bewirkt. Dabei beziehen sich die Buchstaben A, D und M auf die 16-Bit-Werte, die in den Registern A, D und im Speicher $M(A)$ an der in A angegebenen Adresse gespeichert sind.

Die ersten vier Befehle sind gegenüber dem Testprogramm für die CPU unverändert und bewirken, dass die Summe $2+3$ berechnet und in Register D gespeichert wird. Die Zeilen 5 und 6 geben dem Computer die Anweisung, den Wert aus Register D im Datenspeicher an der Adresse 0 abzulegen. Die letzten beiden Zeilen sorgen dafür, dass der Programmzähler nach Beendigung des Programms nicht weiterläuft.

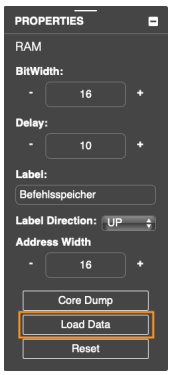
Aufgabe 6.6: Erster Test des Computers

In dieser Aufgabe werden Sie das Programm in den Befehlsspeicher laden und mit dem Computer ausführen.

Die Inhalte können in unterschiedlichen Formaten in den Befehlsspeicher geladen werden. Damit der Simulator erkennt, dass unsere Befehle in Binärdarstellung angegeben sind, muss jeweils ein „0b“ davor gesetzt werden.

1. Setzen Sie das Reset-Signal des Computers auf 1. Der Computer befindet sich damit im Stillstand.
2. Laden Sie das Testprogramm in den Befehlsspeicher. Klicken Sie beim Befehlsspeicher auf die Schaltfläche „Load Data“ und kopieren Sie die folgende Zeile in das Eingabefeld:

   0b0000000000000010, 0b1111000110010000, 0b0000000000000011,
   0b1111010000010000, 0b0000000000000000, 0b1111000111001000,
   0b0000000000000111, 0b1111000000000111



3. Starten Sie das Programm, indem Sie das Reset-Signal auf 0 setzen. Beobachten Sie den Ablauf des Programms. Achten Sie insbesondere auf folgende Punkte:
   - Bei $PZ = 5$ sollte das Resultat der Berechnung im Datenspeicher abgelegt sein. Kontrollieren Sie, ob zu diesem Zeitpunkt $M(A) = 5$ ist.
   - Ab $PZ = 7$ sollte sich der Programmzähler nicht mehr verändern.

   Tipp: Durch die Clock-Time-Einstellung in den Project-PROPERTIES können Sie wählen, wie schnell das Programm ausgeführt wird.

So, wie wir den Computer gebaut haben, kann er im Grunde jedes erdenkliche Programm ausführen! Zur Illustration testen wir ein zweites Programm, das mit einer Schleife die Summe der ersten 10 natürlichen Zahlen berechnet. In Python lautet das Programm:

```
n = 10
summe = 0
while n > 0 :
    summe = summe + n
    n = n - 1
```

Dieses Programm kann in unsere Maschinensprache aus 16-Bit-Befehlen übersetzt werden. Diese Übersetzung ist genau das, was ein Compiler leistet: Beim Kompilieren werden Befehle einer höheren Programmiersprache in Maschinensprache übersetzt.

Für die Maschinensprache unserer CPU haben wir keinen Compiler. Bei kurzen Programmen können wir die Übersetzungsarbeit jedoch gut selbst leisten. In der Sprache unseres Computers lautet das obige Programm:

```
# Dummy-Befehl.
# Wenn beim Setzen von reset = 0, clk = 1 ist,
# wird der erste Befehl nicht vollstaendig ausgefuehrt.
# Deshalb beginnen wir mit einem Befehl, der nicht benoetigt wird.
0000000000000000

# Speichern der Anzahl n = 10 der Schleifendurchlaeufe in M(0):
0000000000001010 # A = 10
1111000110010000 # D = A
0000000000000000 # A = 0
1111000111001000 # M = D

# Initialisiere den Wert summe = 0 in M(25):
0000000000000000 # A = 0
1111000110010000 # D = A
0000000000011001 # A = 25
1111000111001000 # M = D

# Falls n <=0 ist, soll ans Ende des Programms gesprungen werden.
# Der erste Befehl in diesem Block wird im Befehlsspeicher
# an der Adresse 9 abgelegt.
0000000000000000 # A = 0
1111100110010000 # D = M
0000000000011001 # A = 25 (Adresse des Programmende)
1111000111000110 # Jump, if D <= 0

# Aktualisiere die summe = summe + 1 (M(25) += M(0)):
0000000000011001 # A = 25
1111100110010000 # D = M  (Summe in D laden)
0000000000000000 # A = 0  (Summand aus M lesen)
1111110000010000 # D = D + M (Aktualisierte Summe in D speichern)
0000000000011001 # A = 25
1111000111001000 # M = D (Summe im Datenspeicher ueberschreiben)

# Anzahl verbleibende Schleifendurchlaeufe n = n - 1 anpassen (M(0) = M(0) - 1)
0000000000000000 # A = 0
1111111101010000 # D = M - 1
1111000111001000 # M = D

```

```
40 # zum Schleifen-Anfang = Befehl Nr. 9 springen:
41 0000000000001001 # A = 9
42 1111000000000111 # Jump!
43
44 # Programm-Ende:
45 0000000000011001 # A = 25
46 1111000000000111 # Jump! Dieser Befehl hat im Befehlsspeicher die Adresse 25.
```

Aufgabe 6.7: Test einer Schleife

1. Versetzen Sie den Computer durch Setzen des *reset*-Signals auf 1 in den Stillstand.
2. Kopieren Sie die folgenden 26 Befehle und laden Sie sie in den Befehlsspeicher.

   0b0000000000000000, 0b0000000000001010, 0b1111000110010000,
   0b0000000000000000, 0b1111000111001000, 0b0000000000000000,
   0b1111000110010000, 0b0000000000011001, 0b1111000111001000,
   0b0000000000000000, 0b1111100110010000, 0b0000000000011001,
   0b1111000111000110, 0b0000000000011001, 0b1111100110010000,
   0b0000000000000000, 0b1111110000010000, 0b0000000000011001,
   0b1111000111001000, 0b0000000000000000, 0b1111111101010000,
   0b1111000111001000, 0b0000000000001001, 0b1111000000000111,
   0b0000000000011001, 0b1111000000000111
3. Legen Sie unter den „Project-PROPERTIES“ fest, dass die Clock-Time ca. 100 ms beträgt.
4. Starten Sie das Programm durch Setzen von $reset = 0$.
5. Der Computer führt nun das Programm aus. Warten Sie ungefähr 30 Sekunden, bis sich der Wert des Programmzählers nicht mehr verändert.
6. Überprüfen Sie, ob der Speicher das richtige Resultat wiedergibt. Beachten Sie, dass die Werte der Flags im Hexadezimalsystem angegeben werden. Der erwartete Wert von $M(A)$ ist $37_{16} = 55_{10}$.
7. Zusatzfrage: Der Wert der berechneten Summe wird im Datenspeicher an der Adresse 25 gespeichert. Wieso wurde diese Adresse gewählt?

War der Test erfolgreich? Dann gratulieren wir Ihnen herzlich: Sie haben einen vollständigen Computer gebaut!

Mit dem Befehlssatz (vgl. Abschnitt 7.7) können Sie den Computer dazu anleiten, beliebige Programme auszuführen:

Aufgabe 6.8: Zusatzaufgabe – Eigene Programme schreiben

1. Verändern Sie das Testprogramm so, dass die Summe der ersten 20 natürlichen Zahlen berechnet wird.
2. Verändern Sie das Testprogramm so, dass die Summe der ersten 10 ungeraden Zahlen berechnet wird.

3. Schreiben Sie selbst ein Programm, das mit einer Schleife das Produkt der beiden Zahlen 3 und 4 berechnet und das Resultat im Speicher ablegt.

Zusammenfassung

Ein Computer in der Harvard-Architektur besteht im Wesentlichen aus einem Hauptprozessor und zwei Speichereinheiten für Daten und Befehle.

Bei der Ausführung eines Programms erhält die CPU die Befehle aus dem Befehlsspeicher. Diese sind in Maschinensprache angegeben: Jeder Befehl ist ein Binärcode, der die Anweisungen codiert, die von der CPU ausgeführt werden sollen.

Einzelne Anweisungen im Befehlssatz der CPU können Angaben darüber enthalten, welche Operation die ALU durchführen soll, in welche Register oder an welcher Speicheradresse die Resultate abgelegt werden sollen und ob der Programmzähler an eine bestimmte Stelle springen soll.

7 Lösungen

7.1 Lösungen der Aufgaben aus Kapitel 1

Lösung der Aufgabe : 1.1

Beginnen wir mit Damian. Um seine Spalte auszufüllen, muss man seine Aussage analysieren: „Ich komme, aber nur, falls Adam auch dabei ist, oder Bea zu Hause bleibt.“ Im ersten Teil der Aussage wird deutlich, dass Damian immer dann mitkommt, wenn Adam auch dabei ist. Damit kann man bereits einen Teil der Tabelle ausfüllen:

Adam	Bea	Chloé	Damian	Fiona
0	0	1		
0	1	1		
1	0	1	1	
1	1	0	1	

Dem zweiten Teil seiner Aussage entnimmt man, dass Damian auch dann teilnimmt, wenn Adam zwar nicht kommt, dafür Bea zu Hause bleibt. Nur im Fall, dass Bea ohne Adam erscheint, bleibt Damian zu Hause.

Adam	Bea	Chloé	Damian	Fiona
0	0	1	1	
0	1	1	0	
1	0	1	1	
1	1	0	1	

Auf dieselbe Art kann man auch Fionas Aussage auswerten und erhält schliesslich die vollständige Tabelle:

Adam	Bea	Chloé	Damian	Fiona
0	0	1	1	0
0	1	1	0	1
1	0	1	1	1
1	1	0	1	0

Lösung der Aufgabe : 1.2

1. Analog zu Beispiel 1.2 werten wir die Teile im Booleschen Ausdruck $\text{NOR}(x, y) = \overline{x \vee y}$ von innen nach aussen aus:

x	y	$x \vee y$	NOR(x, y)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

2. Mit der gleichen Herangehensweise können die einzelnen Bausteine in XNOR$(x, y) = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ schrittweise ausgewertet werden.

x	y	$\overline{x}$	$\overline{y}$	$x \wedge y$	$\overline{x} \wedge \overline{y}$	XNOR(x, y)
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

3. Für jede der vier Zeilen der Wahrheitstabelle gibt es zwei mögliche Outputs. Insgesamt gibt es also $2^4 = 16$ Boolesche Funktionen mit zwei Inputvariablen:

4.

x	y	z	$\bar{x}$	$y \vee z$	$f(x, y, z)$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	0	1	0

Lösung der Aufgabe : 1.3

1. Für jede Zeile der Wahrheitstabelle, in welcher die Funktion f den Wert 1 annimmt, bilden wir einen Ausdruck aus NEG- und AND-Verknüpfungen, der genau dann den Wert 1 liefert, wenn die Inputvariablen den Wert entsprechend der betrachteten Zeile aufweisen:

$$f(0,0,0) = 1 \longrightarrow (\overline{x} \wedge \overline{y}) \wedge \overline{z}$$
$$f(0,1,1) = 1 \longrightarrow (\overline{x} \wedge y) \wedge z$$
$$f(1,0,0) = 1 \longrightarrow (x \wedge \overline{y}) \wedge \overline{z}$$
$$f(1,1,1) = 1 \longrightarrow (x \wedge y) \wedge z$$

   Die Funktion f ist dann die OR-Verknüpfung der gefundenen Booleschen Ausdrücke:

$$f(x, y, z) = ((\overline{x} \wedge \overline{y}) \wedge \overline{z}) \vee ((\overline{x} \wedge y) \wedge z) \vee ((x \wedge \overline{y}) \wedge \overline{z}) \vee ((x \wedge y) \wedge z).$$

2. Für die n Inputvariablen gibt es insgesamt 2^n mögliche Belegungen. In anderen Worten: Die vollständige Wahrheitstabelle einer Booleschen Funktion von n Variablen besteht aus 2^n Zeilen. Für jede Zeile gibt es dann zwei mögliche Outputs. Insgesamt gibt es damit $2^{(2^n)}$ unterschiedliche Boolesche Funktionen mit n Inputvariablen.

Lösung der Aufgabe : 1.4

Unser Tipp weist darauf hin, dass der gesuchte Ausdruck von der Form $A \wedge B$ oder $\overline{A \wedge B}$ sein muss, wobei die Ausdrücke A und B aus den Inputs x, y oder deren Negation bestehen. Wenn man zusätzlich berücksichtigt, dass die OR-Operation symmetrisch ist, kommen nur vier mögliche Ausdrücke infrage:

$$x \wedge y, \quad \overline{x} \wedge \overline{y}, \quad \overline{x \wedge y}, \quad \overline{\overline{x} \wedge \overline{y}}.$$

Für jede dieser vier Möglichkeiten kann man die Wahrheitstabelle aufstellen und mit jener der OR-Verknüpfung vergleichen. So findet man leicht:

$$x \vee y = \overline{\overline{x} \wedge \overline{y}}.$$

Lösung der Aufgabe : 1.5

1.

x	NAND(x, x)
0	1
1	0

NAND$(x, x) =$ NEG(x)

2. Aus Aufgabe 1.4 wissen wir, dass $x \vee y = \overline{\overline{x} \wedge \overline{y}}$. Um die OR-Funktion als eine Verkettung von NAND-Funktionen darzustellen, können wir im Ausdruck oben jede NEG(x)-Operation durch NAND(x, x) ersetzen und erhalten:

$$x \vee y = \text{NAND}(\overline{x}, \overline{y}) = \text{NAND}\left(\text{NAND}(x, x), \text{NAND}(y, y)\right).$$

3. Aus der Definition der NAND-Funktion sieht man leicht, dass

$$\text{AND}(x, y) = \text{NEG}(\text{NAND}(x, y)).$$

Indem wir wieder die NEG(x) durch NAND(x, x) ersetzen, erhalten wir:

$$x \wedge y = \overline{\text{NAND}(x, y)} = \text{NAND}\left(\text{NAND}(x, y), \text{NAND}(x, y)\right).$$

7.2 Lösungen der Aufgaben aus Kapitel 2

Alle logischen Schaltungen in den Übungen dieses Kapitels können hier eingesehen werden: Schaltungen zu den Aufgaben aus Kapitel 2

Lösung der Aufgabe : 2.1

1. Betrachten wir den Fall $a = 0, b = 0$ genauer: Weil an a keine Spannung anliegt ($a = 0$), wird im AK-Relais keine Verbindung zwischen *in* und *out* hergestellt. Am Output des AK-Relais legt damit auch keine Spannung an. Dies sorgt wiederum dafür, dass die beiden Signale *in* und *out* im RK-Relais verbunden sind, womit am Outputsignal der Schaltung Spannung anliegt:

a	b	out
0	0	1
0	1	1
1	0	
1	1	

Wenn $a = 1$ und $b = 0$ sind, werden die Signale in und out im AK-Relais zwar verbunden, aber an out liegt wegen $b = 0$ immer noch keine Spannung an. Damit hat sich die Situation im RK-Relais gegenüber den ersten beiden Fällen nicht verändert und für den Output der Schaltung gilt weiterhin $out = 1$.

a	b	out
0	0	1
0	1	1
1	0	1
1	1	

Für $a = 1, b = 1$ gilt hingegen, dass am Output des AK-Relais' eine hohe Spannung anliegt. Damit wird die Verbindung zwischen in und out im RK-Relais unterbrochen und am Outputsignal der Schaltung gilt $out = 0$.

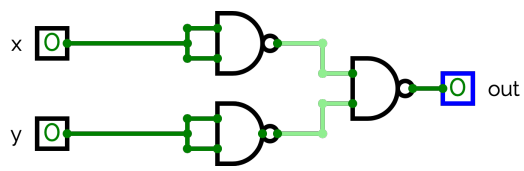
a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

2. Die elektronische Schaltung realisiert die NAND-Funktion.

Lösung der Aufgabe : 2.2

1. Es handelt sich um die NEG-Funktion. Vergleiche auch die Erkenntnisse $\text{NEG}(x) = \text{NAND}(x, x)$ aus Aufgabe 1.5.

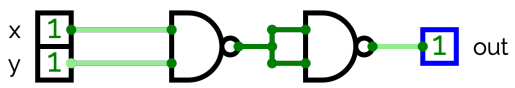
2. Wir setzen die Formel $x \vee y = \text{NAND}\left(\text{NAND}(x, x), \text{NAND}(y, y)\right)$ aus Aufgabe 1.5 um und erhalten die Schaltung:



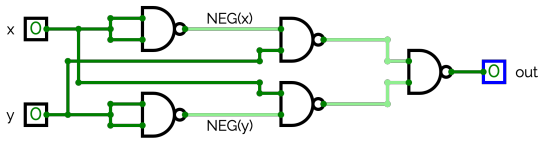
3. Erneut verwenden wir eine Formel aus Aufgabe 1.5:

$$x \wedge y = \text{NAND}\left(\text{NAND}(x, y), \text{NAND}(x, y)\right).$$

Die dazu passende Schaltung ist dann:

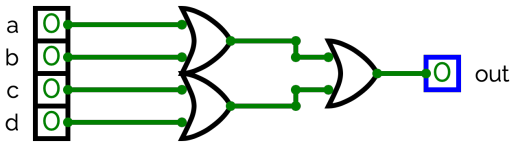


4. Wir gehen so vor, dass wir in einem ersten Bereich der Schaltung die Negationen $\overline{x}$ und $\overline{y}$ berechnen, und diese dann entsprechend dem vorgegebenen Booleschen Ausdruck für XOR kombinieren.



Lösung der Aufgabe : 2.3

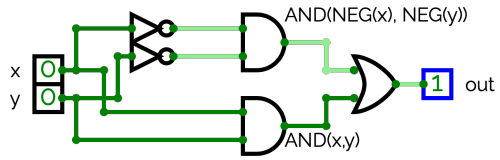
1. Dies ist eine logische Schaltung für die OR-Funktion mit vier Inputs:



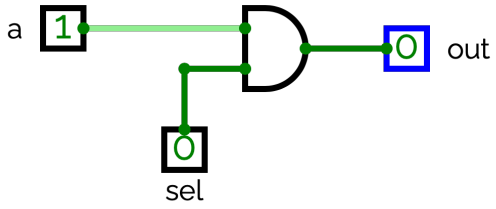
2. Wir gehen so vor, dass wir zuerst Schaltungen für die beiden geklammerten Teilausdrücke in

$$\text{XNOR}(x, y) = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$$

bauen und diese schliesslich mit einem OR-Gatter verbinden:

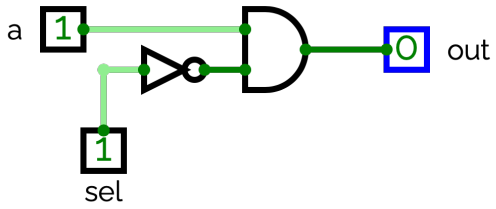


3. Um einen Multiplexer zu realisieren, ist es hilfreich, das AND-Gatter als eine Art AK-Relais aufzufassen.

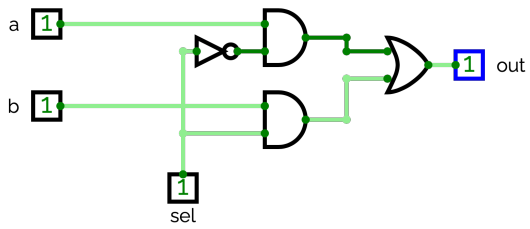


Wenn am *sel*-Signal Spannung anliegt, wird das Signal a an den Output weitergeleitet. Andernfalls ist (unabhängig vom Wert von a) $out = 0$.

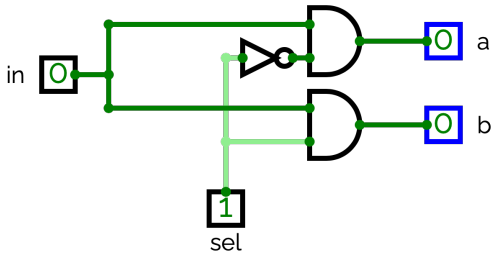
Analog kann man die folgende Schaltung als RK-Relais auffassen:



Diese beiden Teilschaltungen können dann leicht zu einem Multiplexer kombiniert werden:



4. Aus den Hilfsschaltungen für die vorangegangene Frage kann analog eine logische Schaltung für den DMux erstellt werden:



7.3 Lösungen der Aufgaben aus Kapitel 3

Alle logischen Schaltungen in den Übungen dieses Kapitels können hier eingesehen werden: Schaltungen zu den Aufgaben aus Kapitel 3

Lösung der Aufgabe : 3.1

1. $11010_2 + 110110_2 = 1010000_2$ oder in Dezimaldarstellung: $26 + 54 = 80$

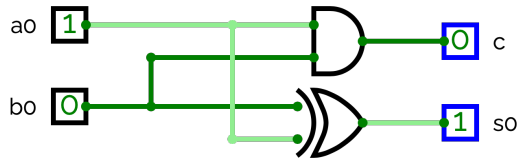
2.

a_0	b_0	c	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Offensichtlich gilt: $c = \text{AND}(a_0, b_0)$ und $s_0 = \text{XOR}(a_0, b_0)$

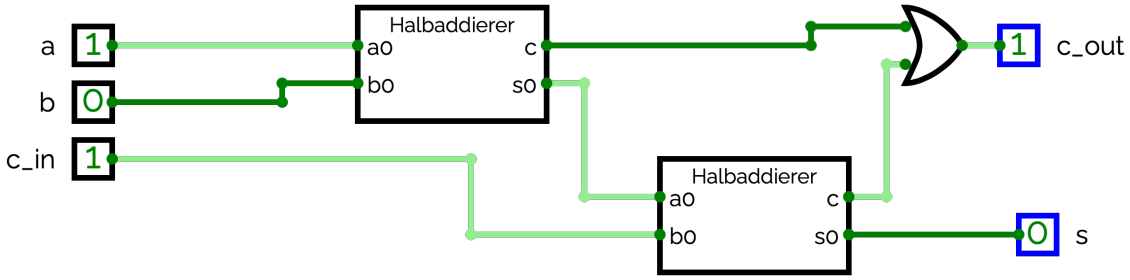
Lösung der Aufgabe : 3.2

Wir setzen die Erkenntnisse $c = \text{AND}(a_0, b_0)$ und $s_0 = \text{XOR}(a_0, b_0)$ aus Aufgabe 3.2 in eine Schaltung um:



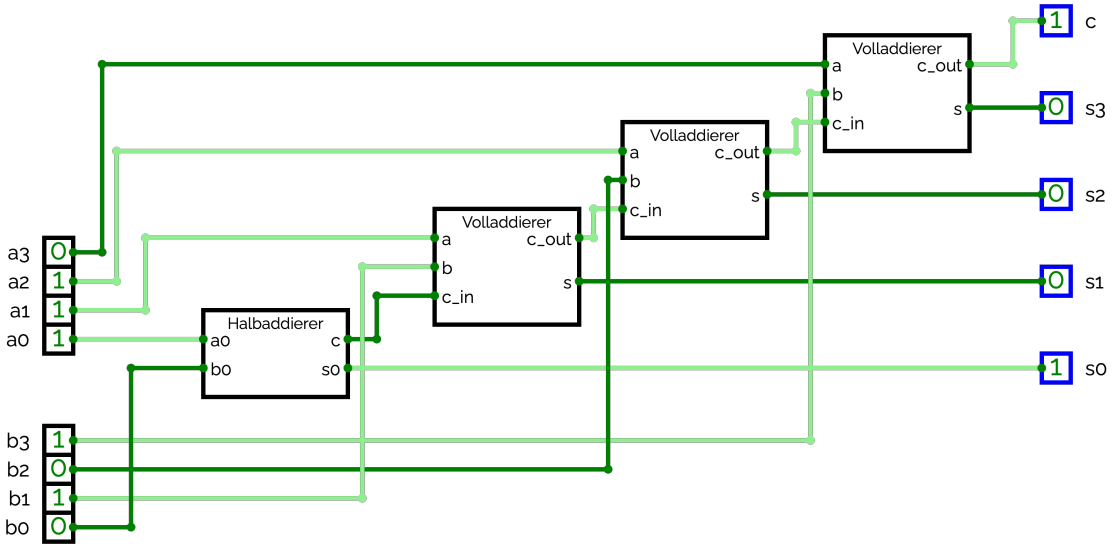
Lösung der Aufgabe : 3.3

Im Unterschied zum Halbaddierer werden beim Volladdierer drei anstatt zwei Bits addiert. Um dies in einer Schaltung umzusetzen, können die ersten beiden Summanden a und b in einem Halbaddierer addiert werden. Anschliessend wird mit einem zweiten Halbaddierer die Summe der ersten beiden Bits mit dem dritten gebildet.

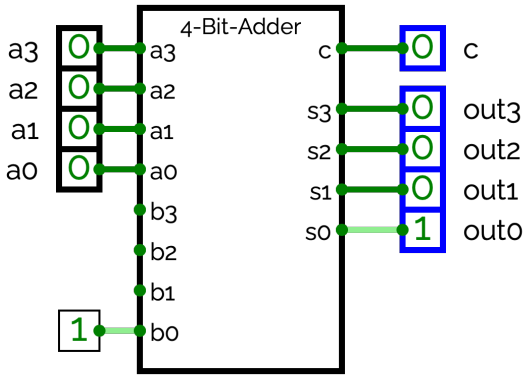


Lösung der Aufgabe : 3.4

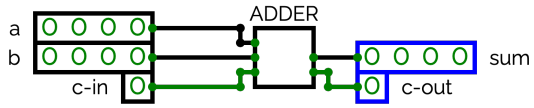
1. Die Schaltung für den 4-Bit-Addierer widerspiegelt die schriftliche Addition von 4-Bit-Zahlen: Zunächst werden mit einem Halbaddierer die beiden niedrigstwertigen Bits addiert. Alle anderen Stellen werden zusammen mit dem Übertrag aus der nächsttieferen Ziffer in einem Volladdierer aufsummiert.



2. Logische Schaltung für den 4-Bit-Incrementer

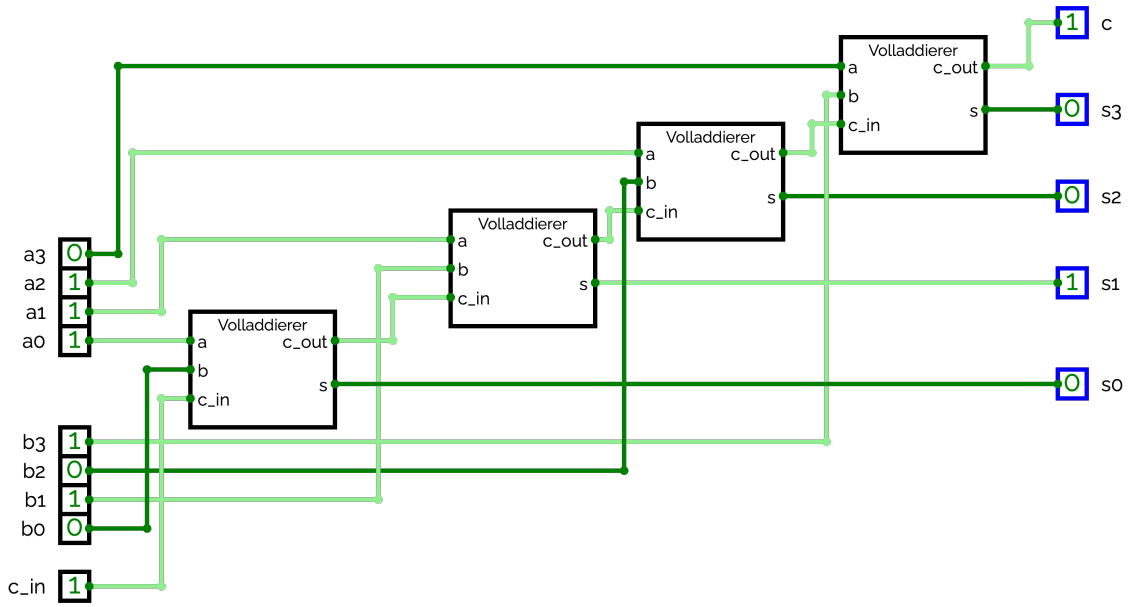


3. Der im Simulator eingebaute ADDER entspricht dem Addierer aus Teilaufgabe 1. Eine logische Schaltung mit dem Simulator-Addierer und Signalen der Bandbreite 4 sieht wie folgt aus:

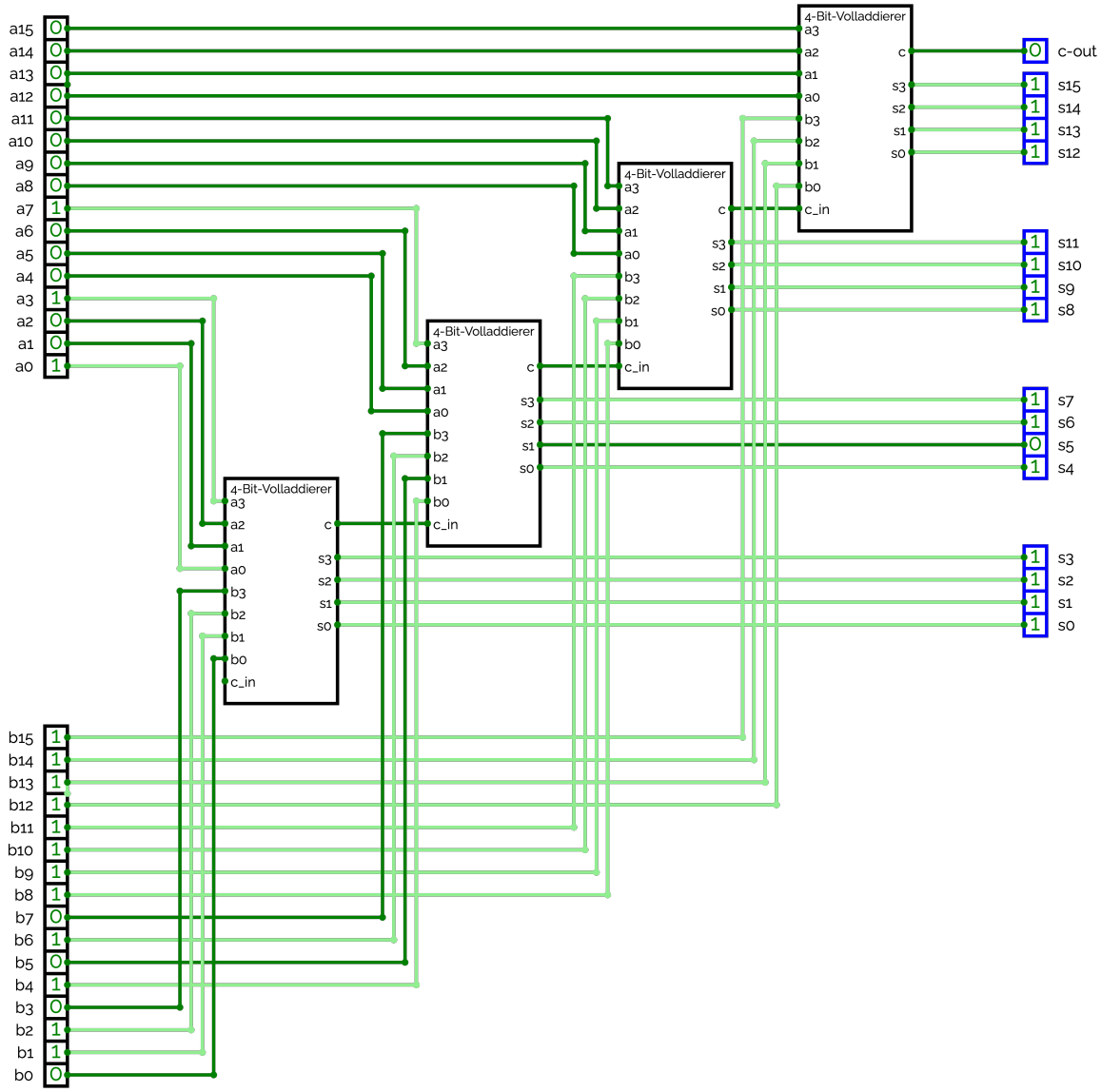


4. Wir gehen analog zum Aufbau des 4-Bit-Addierers vor mit dem Unterschied, dass wir als Grundbaustein nicht die Summe von 1-Bit-Zahlen verwenden, sondern von 4-Bit-Zahlen. Dazu müssen wir den 4-Bit-Addierer zuerst so verändern, dass er neben den beiden 4-Bit-Zahlen als Input zusätzlich einen 1-Bit-Übertrag verarbeitet:

5. Dies ist eine logische Schaltung für das Hilfselement 4-Bit-Volladdierer

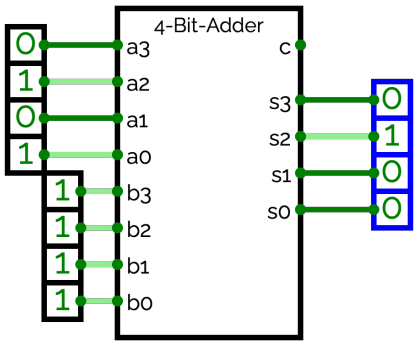


Nun kombinieren wir vier solche 4-Bit-Volladdierer zu einem 16-Bit-Addierer. Dabei gehen wir gleich vor wie bei der Erstellung des 4-Bit-Addierers, verarbeiten dabei aber immer Gruppen aus 4-Bits.



Lösung der Aufgabe : 3.5

1. Durch Probieren findet man heraus, dass die folgende Belegung von b zum gewünschten Resultat führt.

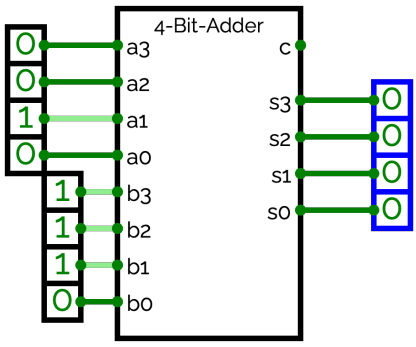


   Ein guter Kandidat für die neue Darstellung der Zahl -1 im Schaltkreis wäre demnach $b = 1111$.

2. Wieder kann man die wenigen Möglichkeiten mit dem 4-Bit-Addierer testen und findet:

$$0010 + 1110 = 0000.$$

3. Am einfachsten geht es, wenn man mit dem 4-Bit-Addierer unterschiedliche Inputs ausprobiert, bis die richtige Kombination gefunden ist:



   Insgesamt findet man dann die folgenden neuen Darstellungen für die negativen Zahlen:

$$-3 \to 1101, -4 \to 1100, -5 \to 1011, -6 \to 1010, -7 \to 1001.$$

4. Das Muster ist nicht leicht zu erkennen. Vielleicht fällt Ihnen auf, dass ein direkter Zusammenhang zwischen der Codierung der Zahl $-n$ und jener der Zahl $n-1$ besteht. Betrachten Sie dazu die folgende Tabelle und als Beispiel die markierten Einträge:

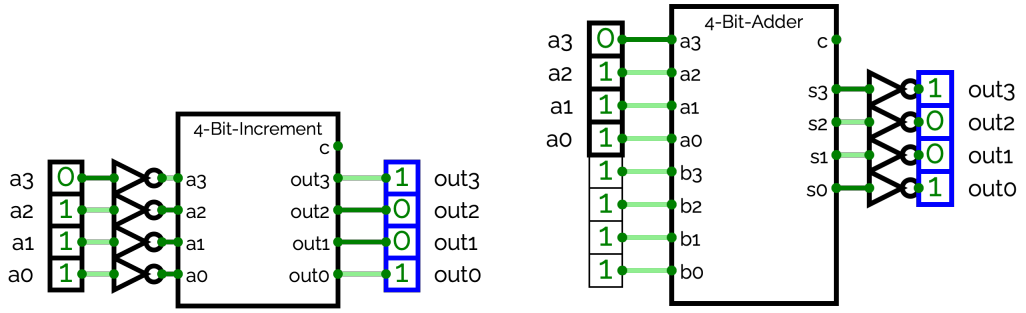
Codierung der Zahl	Codierung der Gegenzahl
0000	0000
0001	1111
0010	1110
0011	1101
0100	1100
0101	1011
0110	1010
0111	1001

Aus der Beobachtung folgt: Die 4-Bit-Codierung der Gegenzahl von n lässt sich finden, indem man zuerst die Zahl Codierung $n-1$ ermittelt und anschliessend jedes Bit invertiert.

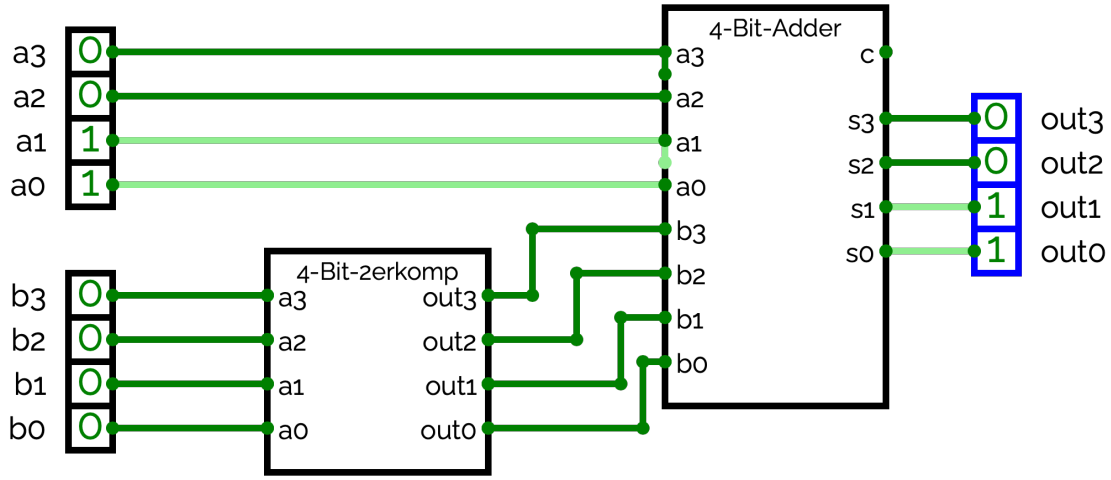
Alternativ kann die Beobachtung auch so ausgelegt werden: Die 4-Bit-Codierung der Gegenzahl von n lässt sich finden, indem man zuerst jedes Bit invertiert und anschliessend die binäre Zahl 1 addiert.

Lösung der Aufgabe : 3.6

1. Zweierkomplement $\rightarrow$ Dezimalzahl.
   (a) $01011101 \rightarrow 93_{10}$
   (b) $11010110 \rightarrow -42_{10}$
   (c) $11111111 \rightarrow -1_{10}$
2. Dezimalzahl $\rightarrow$ Zweierkomplement.
   (a) $10_{10} \rightarrow 00001010$
   (b) $-10_{10} \rightarrow 11110111$
   (c) $-120_{10} \rightarrow 10001000$
3. Die höchste Zahl ist $2^7-1 = 127$. Ihre 8-Bit-Zweierkomplement-Codierung ist 01111111. Die kleinste Zahl ist -128. Ihre 8-Bit-Zweierkomplement-Codierung ist 10000000.
4. Je nachdem, wie die Aufgabe 3.5 beantwortet wurde (siehe die beiden Varianten in den Lösungen), drängt sich die eine oder die andere der folgenden Schaltungen für das Zweierkomplement auf.:



5. Gemäss unserer Strategie kann die Codierung von $a - b$ berechnet werden als $a + (-b)$, wobei $-b$ in der neuen Codierung dargestellt wird. Damit erhält man die folgende Schaltung für die Subtraktion:

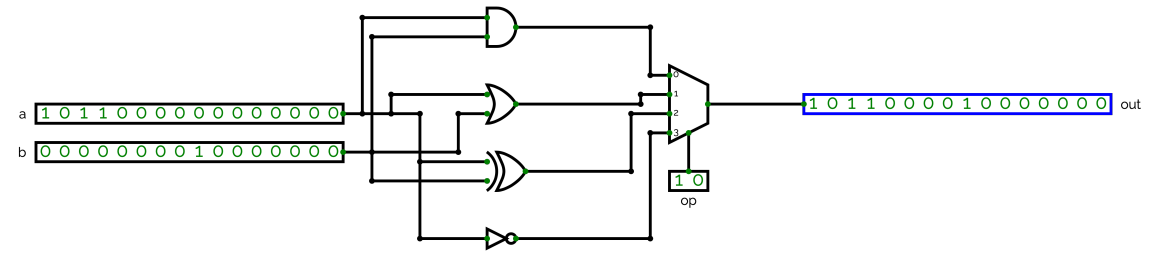


7.4 Lösungen der Aufgaben aus Kapitel 4

Alle logischen Schaltungen in den Übungen dieses Kapitels können hier eingesehen werden: Schaltungen zu den Aufgaben aus Kapitel 4

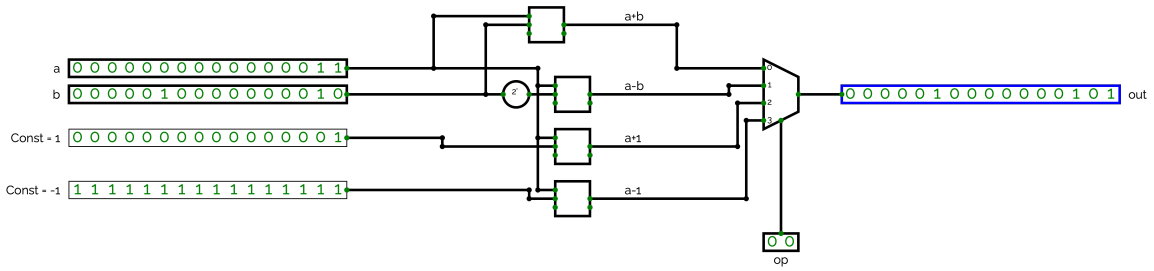
Lösung der Aufgabe : 4.1

Die Idee ist, dass in einem ersten Schritt alle möglichen Resultate der LU berechnet werden. Diese werden an einen Multiplexer weitergereicht. Das *op*-Signal entscheidet, welches der Resultate an das Outputsignal geleitet wird.

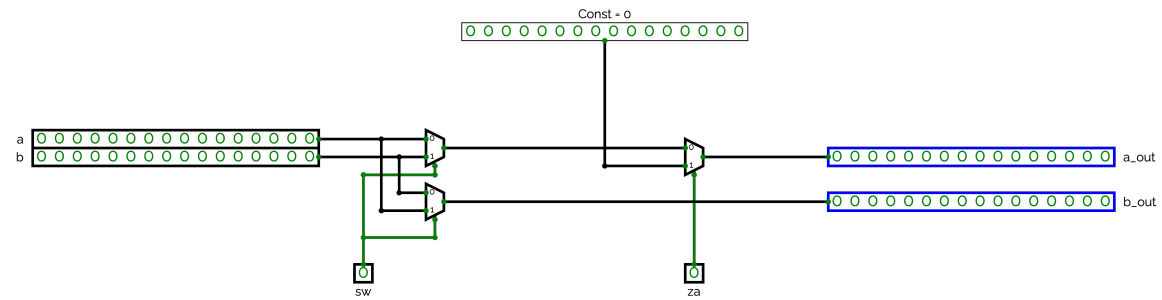


Lösung der Aufgabe : 4.2

Analog zur Konstruktion der AU werden alle möglichen Resultate berechnet und mit einem MUX wird entschieden, welches an den Outputkanal geleitet wird. Zur Berechnung der Subtraktion wird zuerst das Zweierkomplement von b ermittelt. Für die Operationen $a + 1$ und $a - 1$ werden konstante Signale verwendet.



Lösung der Aufgabe : 4.3



Lösung der Aufgabe : 4.4

1. Die Lösung ist bereits in der Aufgabenstellung angegeben. Wenn Sie konzentriert arbeiten, kann eigentlich nichts schief gehen.

2. Zur Veranschaulichung der Vorgehensweise betrachten wir den Fall, dass die Kontrollbits die folgenden Werte aufweisen: $u = 1, op = 01, za = 1, sw = 0$. Was wird in diesem Fall durch die ALU ausgegeben? Die Belegung $u = 1$ bedeutet, dass das Resultat der arithmetischen Einheit ausgegeben wird. Dort wird die Operation $op = 01$ ausgeführt, was für eine Subtraktion steht. Die beiden Inputsignale werden aufgrund von $sw = 0$ nicht vertauscht. Vor der Subtraktion wird jedoch der Input a wegen $za = 1$ durch 0 ersetzt. Insgesamt wird also $0 - b = -b$ ausgegeben.

   Die vollständige Wahrheitstabelle der ALU ist:

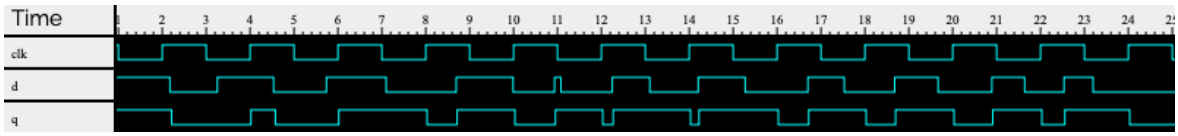
u	op	za	sw	out
0	00	0	0	AND(a, b)
0	00	0	1	AND(b, a)
0	00	1	0	0
0	00	1	1	0
0	01	0	0	OR(a, b)
0	01	0	1	OR(b, a)
0	01	1	0	b
0	01	1	1	a
0	10	0	0	XOR(a, b)
0	10	0	1	XOR(b, a)
0	10	1	0	a
0	10	1	1	b
0	11	0	0	NEG(a)
0	11	0	1	NEG(b)
0	11	1	0	-1
0	11	1	1	-1
1	00	0	0	$a + b$
1	00	0	1	$b + a$
1	00	1	0	b
1	00	1	1	a
1	01	0	0	$a - b$
1	01	0	1	$b - a$
1	01	1	0	$-b$
1	01	1	1	$-a$
1	10	0	0	$a + 1$
1	10	0	1	$b + 1$
1	10	1	0	1
1	10	1	1	1
1	11	0	0	$a - 1$
1	11	0	1	$b - 1$
1	11	1	0	-1
1	11	1	1	-1

7.5 Lösungen der Aufgaben aus Kapitel 5

Alle logischen Schaltungen in den Übungen dieses Kapitels können hier eingesehen werden: Schaltungen zu den Aufgaben aus Kapitel 5

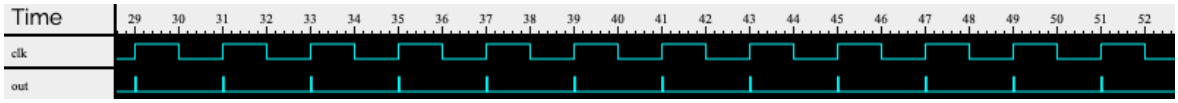
Lösung der Aufgabe : 5.1

1. ...
2. ...
3. ...
4. Das Zeitdiagramm sieht zum Beispiel wie folgt aus. Beachten Sie, wie das Signal q „verzögert“ auf Änderungen des Inputs d reagiert.

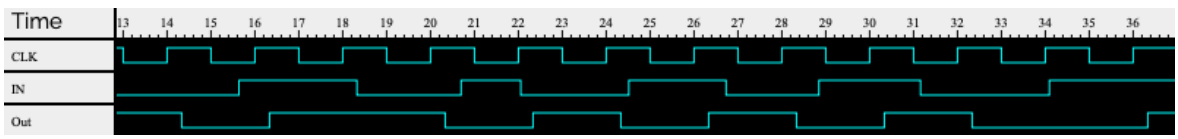


5. Zum Verhalten, das im Zeitdiagramm unten ersichtlich ist, kommt es, weil das NEG-Gatter eine gewisse Zeitspanne benötigt, um das eingehende Signal zu verarbeiten. Beim Wechsel der Clock von 0 auf 1 wird es also einen kurzen Moment geben, in welchem das Outputsignal des NEG-Gatters noch nicht auf 0 gewechselt hat. In diesem kurzen Augenblick ist der Outputwert $pulse = 1$.

   (Der Simulator ist nicht mächtig genug, um den kurzen 1-Output ausserhalb des Timing-Diagramms anzuzeigen.)

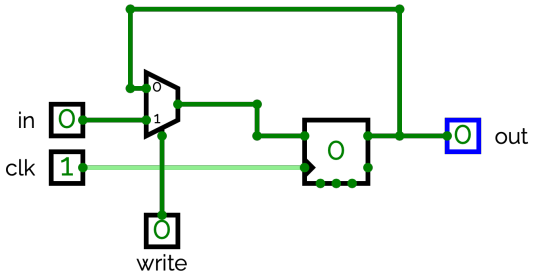


6. Der Pulser kann das Taktsignal so verändern, dass es innerhalb eines Taktzyklus nur für eine ganz kurze Zeitspanne den Wert 1 annimmt. Wenn dieses „gepulste“ Signal an den D-Latch weitergereicht wird, kann sich das Outputsignal Q nur während dieser kurzen Zeitspanne an den Wert von d anpassen. Zwischen zwei Puls-Signalen wird somit der Wert von d für (nahezu) einen Taktzyklus konserviert.
7. Das Zeitdiagramm sieht beispielsweise wie folgt aus. Beachten Sie, dass sich der Zustand des Outputs nur jeweils zu den Zeitpunkten an jenem von d anpassen kann, die unmittelbar auf den Sprung des Clock-Signals von 0 auf 1 folgen.

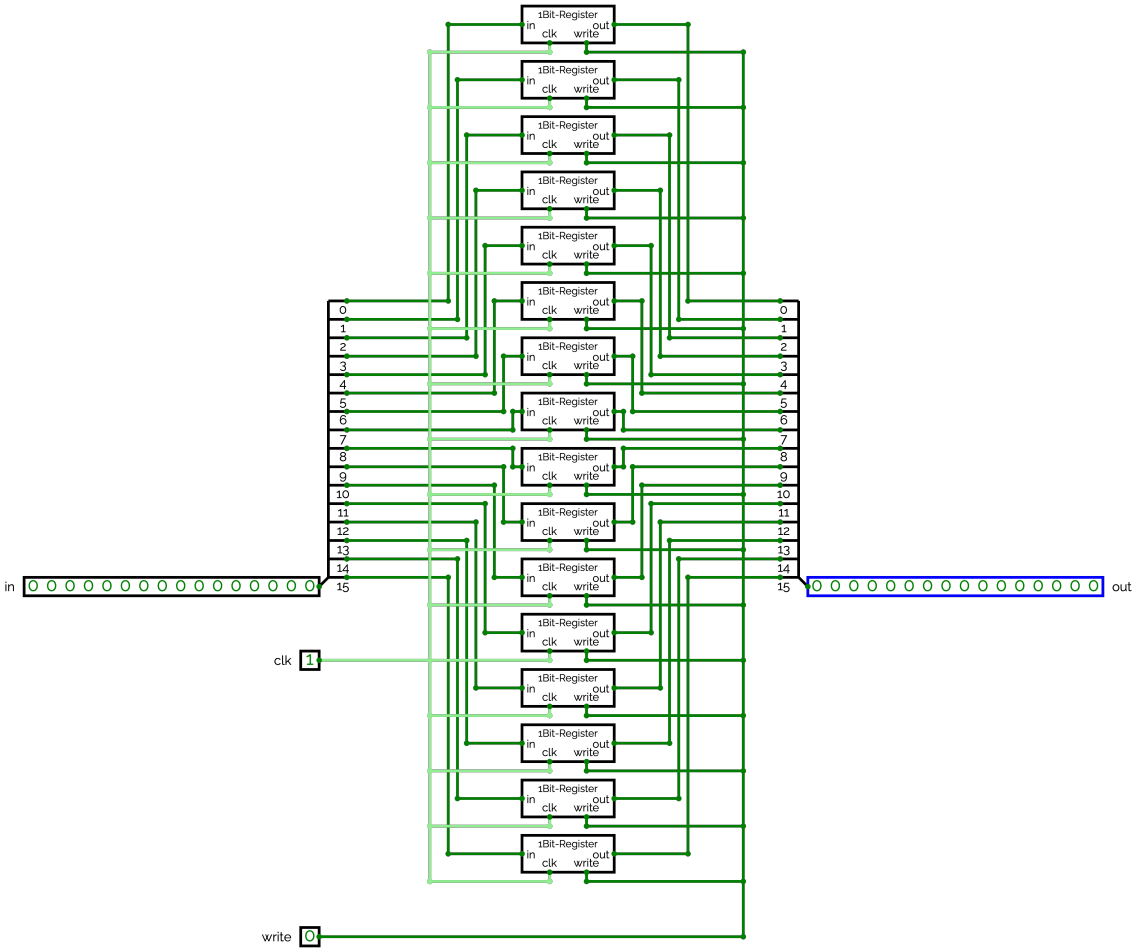


Lösung der Aufgabe : 5.2

1. Das ist eine logische Schaltung für ein 1-Bit-Register: Gemäss dem Tipp wird mit einem Multiplexer gewählt, ob ein neues (bei $write = 1$) oder das bereits gespeicherte Signal (bei $write = 0$) an ein D-Latch weitergereicht wird.



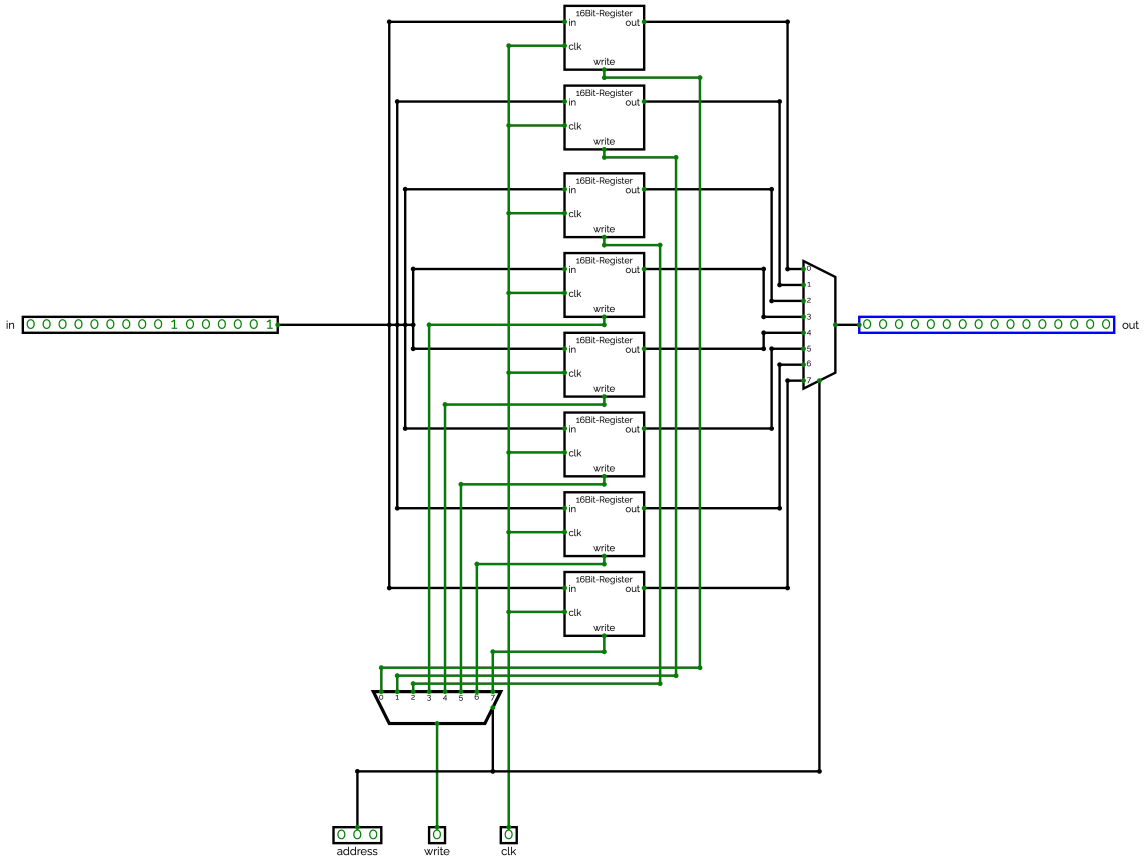
2. Das ist eine logische Schaltung für ein 16-Bit-Register:



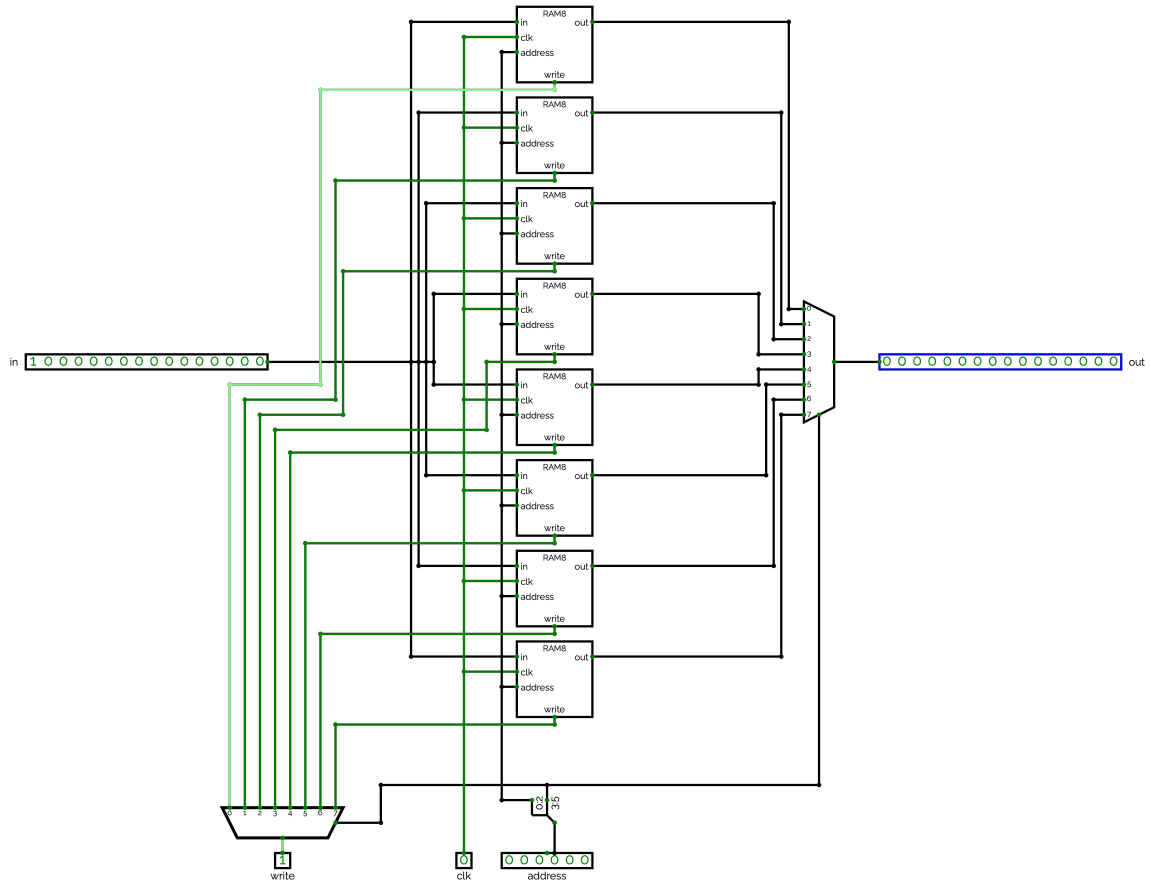
Lösung der Aufgabe : 5.3

1. Logische Schaltung für einen RAM8-Speicher: Die Grundidee wird im Tipp beschrieben. Das Inputsignal *in* wird gleichzeitig an alle Register im RAM geleitet. Mit dem einem Demultiplexer (unten in der abgebildeten Schaltung) wird gesteuert, welches Register das *write*-Signal erhält.

   Der Multiplexer rechts in der abgebildeten Schaltung entscheidet darüber, welches Register ausgelesen wird.

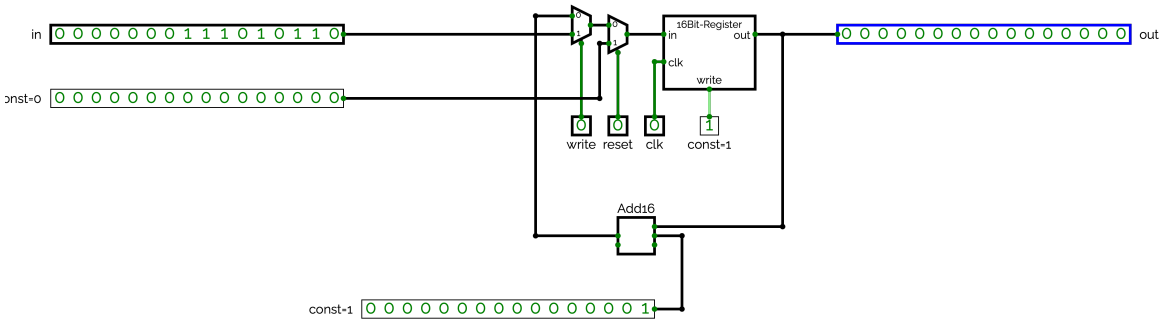


2. Das ist eine logische Schaltung für einen RAM64-Speicher:



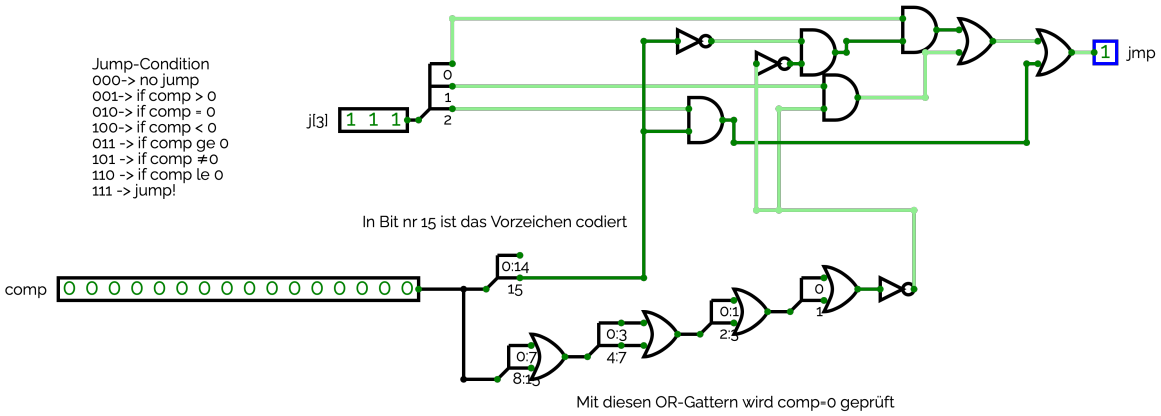
Lösung der Aufgabe : 5.4

Das ist eine logische Schaltung für einen Programmzähler:



Lösung der Aufgabe : 5.5

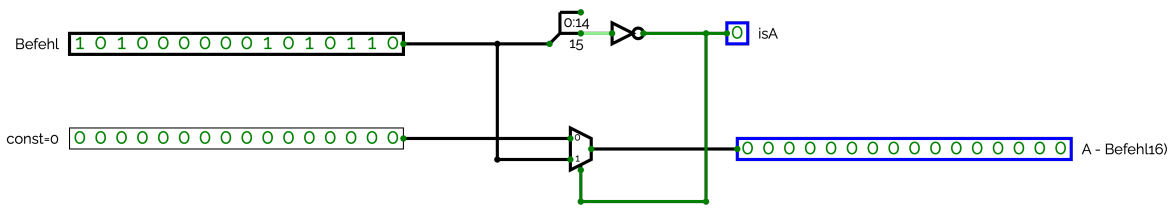
Das ist eine logische Schaltung für einen Jumper:



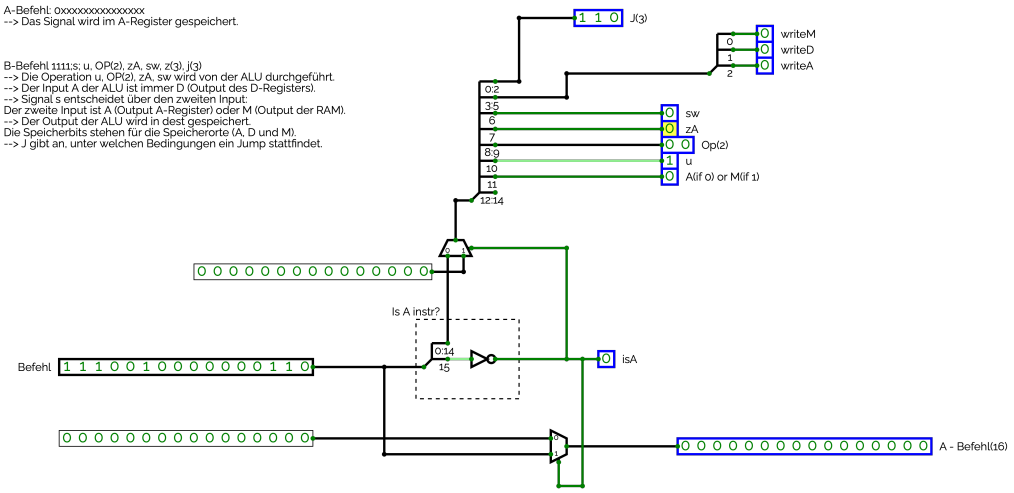
7.6 Lösungen der Aufgaben aus Kapitel 6

Alle logischen Schaltungen in den Übungen dieses Kapitels können hier eingesehen werden: Schaltungen zu den Aufgaben aus Kapitel 6

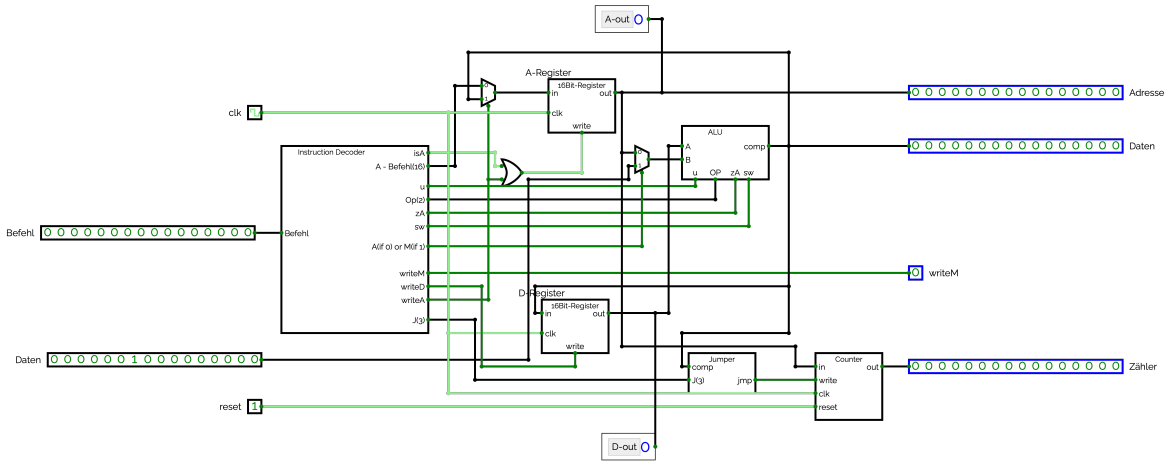
Lösung der Aufgabe : 6.1



Lösung der Aufgabe : 6.2



Lösung der Aufgabe : 6.3



Lösung der Aufgabe : 6.4

Die folgende Tabelle zeigt den erwarteten Zustand der Outputsignale des A- und D-Registers, nachdem ein Befehl eingegeben und ein Taktzyklus (0-1-0) absolviert wurde.

Befehl Nr.	*A*-out	*B*-out
1	2	Zustand unverändert
2	2	2
3	3	2
4	3	5

Lösung der Aufgabe : 6.5

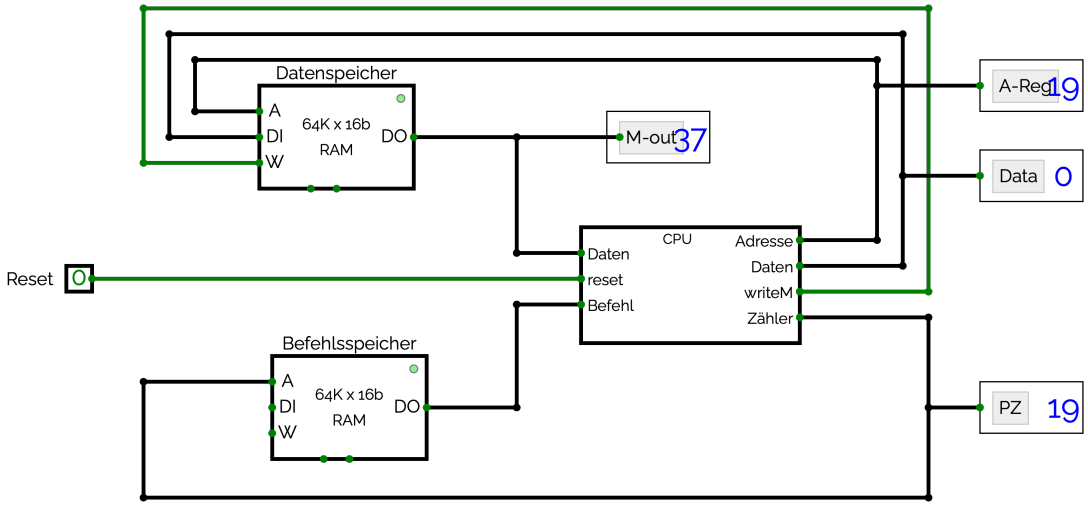
Gehen Sie nach der Anleitung vor. Es kann nichts schief gehen.

Lösung der Aufgabe : 6.6

Gehen Sie nach der Anleitung vor. Es kann nichts schief gehen.

Lösung der Aufgabe : 6.7

Am Ende des Programms trägt das *A*-Register die Adresse des Programmendes (Befehl Nr. 25). Gleichzeitig wird durch den Inhalt von Register *A* die Adresse definiert, an welcher der Inhalt von M-out gelesen wird. Die Adresse für den Speicherplatz der Summe wurde also gewählt, sodass am Ende des Programms das Resultat unter M-out ablesbar ist.



Lösung der Aufgabe : 6.8

1. Damit das Testprogramm die Summe der ersten 20 Zahlen berechnet, muss man bloss die Zeile 8 im Programmcode ersetzen durch *0000000000010100 # A = 20*.

2. Eine mögliche Lösung in Python ist die folgende:

```
1 n = 10
2 summe = 0
```

```
3 while n > 0 :
4     summe = summe + 2*n - 1
5     n = n - 1
6
```

Dabei muss aber berücksichtigt werden, dass unsere ALU keine Multiplikation durchführen kann. In Maschinensprache wird $summe + 2 * n$ darum ersetzt durch zwei Additionen: $summe + n + n$. Damit das Resultat am Schluss aus M-out gelesen werden kann, wurde ausserdem der Speicherplatz für die Summe angepasst. Insgesamt lautet die Lösung in Maschinensprache:

```
1  # Dummy-Befehl.
2  # Wenn beim Setzen von reset = 0, clk = 1 ist,
3  # wird der erste Befehl nicht vollstaendig ausgefuehrt.
4  # Deshalb beginnen wir mit einem Befehl, der nicht benoetigt wird.
5  0000000000000000
6
7  # Speichern der Anzahl n = 10 der Schleifendurchlaeufe in M[0]:
8  0000000000001010 # A = 10
9  1111000110010000 # D = A
10 0000000000000000 # A = 0
11 1111000111001000 # M = D
12
13 # Initialisiere den Wert summe = 0 in M[27]:
14 0000000000000000 # A = 0
15 1111000110010000 # D = A
16 0000000000011011 # A = 27
17 1111000111001000 # M = D
18
19 # Falls n <=0 ist, soll ans Ende des Programms gesprungen werden.
20 # Der erste Befehl in diesem Block wird im Befehlsspeicher
21 # an der Adresse 9 abgelegt.
22 0000000000000000 # A = 0
23 1111100110010000 # D = M
24 0000000000011011 # A = 27 (Adresse des Programmendes)
25 1111000111000110 # Jump, if D <= 0
26
27 # Summe in D laden
28 0000000000011011 # A = 21
29 1111100110010000 # D = M  (Summe in D laden)
30 # summe = summe + M[0] + M[0] -1
31 1111110000010000 # D = D + M
32 1111110000010000 # D = D + M
33 1111011100010000 # D = D - 1
34
35
36 0000000000011011 # A = 27
37 1111000111001000 # M = D (Summe im Datenspeicher ueberschreiben)
38
39 # Restliche Schleifendurchlaeufe n = n - 1 anpassen (M[0] = M[0] - 1)
40 0000000000000000 # A = 0
41 1111111101010000 # D = M - 1
42 1111000111001000 # M = D
43
44 # Zum Schleifen-Anfang = Befehl Nr. 9 springen:
45 0000000000001001 # A = 9
```

```
1111000000000111 # Jump!

# Programm-Ende:
0000000000011011 # A = 27
1111000000000111 # Jump! Dieser Befehl hat im Befehlsspeicher die
    Adresse 27.

```

Sofern Sie die Lösung testen möchten, können Sie dies mittels der nachstehenden Version tun, die sich mit Copy/Paste in den Befehlsspeicher laden lässt:

0b0000000000000000, 0b0000000000001010, 0b1111000110010000,
0b0000000000000000, 0b1111000111001000, 0b0000000000000000,
0b1111000110010000, 0b0000000000011011, 0b1111000111001000,
0b0000000000000000, 0b1111100110010000, 0b0000000000011011,
0b1111000111000110, 0b0000000000011011, 0b1111100110010000,
0b0000000000000000, 0b1111110000010000, 0b1111110000010000,
0b1111011100010000, 0b0000000000011011, 0b1111000111001000,
0b0000000000000000, 0b1111111101010000, 0b1111000111001000,
0b0000000000001001, 0b1111000000000111, 0b0000000000011011,
0b1111000000000111

3. Eine mögliche Lösung ist:

```
0000000000000000 # Dummy-Befehl
## Setze M[0] = 3, M[1] = 4, M[28] = 0
0000000000000011 # A = 3
1111000110010000 # D = A
0000000000000000 # A = 0
1111000111001000 # M = D
0000000000000100 # A = 4
1111000110010000 # D = A
0000000000000001 # A = 1
1111000111001000 # M = D
0000000000000000 # A = 0
1111000110010000 # D = A
0000000000011100 # A = 28
1111000111001000 # M = D

## Schleife
## while M[0] >=0: M[28] = M[28] + M[1]
0000000000000000 # A = 0 Schleifenbeginn (Befehl 13)
1111100110010000 # D = M
0000000000011100 # A = 28 (Adresse des Programmendes)
1111000111000110 # Jump if D <= 0
0000000000000001 # A = 1
1111100110010000 # D = M
0000000000011100 # A = 28
1111110000010000 # D = D + M
1111000111001000 # M = D
0000000000000000 # A = 0
1111111101010000 # D = M - 1
1111000111001000 # M = D
0000000000001101 # A = 13
1111000000000111 # Jump!
```

```
32
33 ## Programm-Ende
34 0000000000011100 # A = 28
35 1111000000000111 # Jump
```

Wenn Sie die Lösung testen möchten: Hier ist eine Version, die Sie mit Copy/Paste in den Befehlsspeicher laden können:

0b0000000000000000, 0b0000000000000011, 0b1111000110010000,
0b0000000000000000, 0b1111000111001000, 0b0000000000000100,
0b1111000110010000, 0b0000000000000001, 0b1111000111001000,
0b0000000000000000, 0b1111000110010000, 0b0000000000011100,
0b1111000111001000, 0b0000000000000000, 0b1111100110010000,
0b0000000000011100, 0b1111000111000110, 0b0000000000000001,
0b1111100110010000, 0b0000000000011100, 0b1111110000010000,
0b1111000111001000, 0b0000000000000000, 0b1111111101010000,
0b1111000111001000, 0b0000000000001101, 0b1111000000000111,
0b0000000000011100, 0b1111000000000111

7.7 Der Befehlssatz und die Maschinensprache

α-Befehle $0xxxxxxxxxxxxxx$:
Das Signal $0xxxxxxxxxxxxxx$ wird in Register A gespeichert.

β-Befehle $1111saaaaazzzjjj$:
Wahl der Operation (= Belegung des s- und der a-Bits); die Variablen D, A und M in der Output-Spalte beziehen sich auf die Werte, die in Register D, A oder im Datenspeicher $M = M(A)$ (an der Adresse A) gespeichert sind.

s	u	op	za	sw	out
0	0	00	0	0	AND(D, A)
0	0	00	1	0	0
0	0	01	0	0	OR(D, A)
0	0	01	1	0	A
0	0	01	1	1	D
0	0	10	0	0	XOR(D, A)
0	0	11	0	0	NEG(D)
0	0	11	0	1	NEG(A)
0	0	11	1	0	-1
0	1	00	0	0	$D + A$
0	1	01	0	0	$D - A$
0	1	01	0	1	$A - D$
0	1	01	1	0	$-A$
0	1	01	1	1	$-D$
0	1	10	0	0	$D + 1$
0	1	10	0	1	$A + 1$
0	1	10	1	0	1
0	1	11	0	0	$D - 1$
0	1	11	0	1	$A - 1$
1	0	00	0	0	AND(D, M)
1	0	01	0	0	OR(D, M)
1	0	01	1	0	M
1	0	10	0	0	XOR(D, M)
1	0	11	0	1	NEG(M)
1	1	00	0	0	$D + M$
1	1	01	0	0	$D - M$
1	1	01	0	1	$M - D$
1	1	01	1	0	$-M$
1	1	10	0	1	$M + 1$
1	1	11	0	1	$M - 1$

Bemerkung: Die vom Simulator zur Verfügung gestellten RAM-Bausteine sind nicht taktflankengesteuert. Befehle wie zum Beispiel $M = M + 1$, bei welchen der Wert in einer Speicherzelle aktualisiert wird, funktionieren mit den RAM-Bausteinen nicht richtig. Stattdessen müssen Sie den neuen Wert in Register D zwischenspeichern: $D = M + 1; M = D$.

Ziel (Belegung der z-Bits)

zzz	write A	write D	write M
000	0	0	0
001	0	0	1
010	0	1	0
011	0	1	1
100	1	0	0
101	1	0	1
110	1	1	0
111	1	1	1

Sprung-Bedingungen (Belegung der j-Bits)

jjj	Sprungbedingung
000	kein Sprung
001	$comp > 0$
010	$comp = 0$
011	$comp \geq 0$
100	$comp < 0$
101	$comp \neq 0$
110	$comp \leq 0$
111	Sprung!

Literatur

[1] George Boole. *The Mathematical Analysis of Logic*. Cambridge: Macmillan, Barclay, Macmillan; London: George Bell, 1847.

[2] Juraj Hrmokovic u. a. *Informatik - Data Science und Sicherheit*. Klett und Balmer Verlag, 2022.

[3] Nand to Tetris. *Nand to Tetris Website*. Accessed: 2024-11-24. URL: `https://www.nand2tetris.org`.

[4] Noam Nisan und Simon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press, 2021.

[5] Claude E. Shannon. „A Symbolic Analysis of Relay and Switching Circuits“. In: *Transactions of the Institute of American Engineers* (1938).

[6] Wikipedia. *Zuse Z3 - Wikipedia*. Accessed: 2024-11-24. URL: `https://de.wikipedia.org/wiki/Zuse_Z3`.

Abbildungsverzeichnis