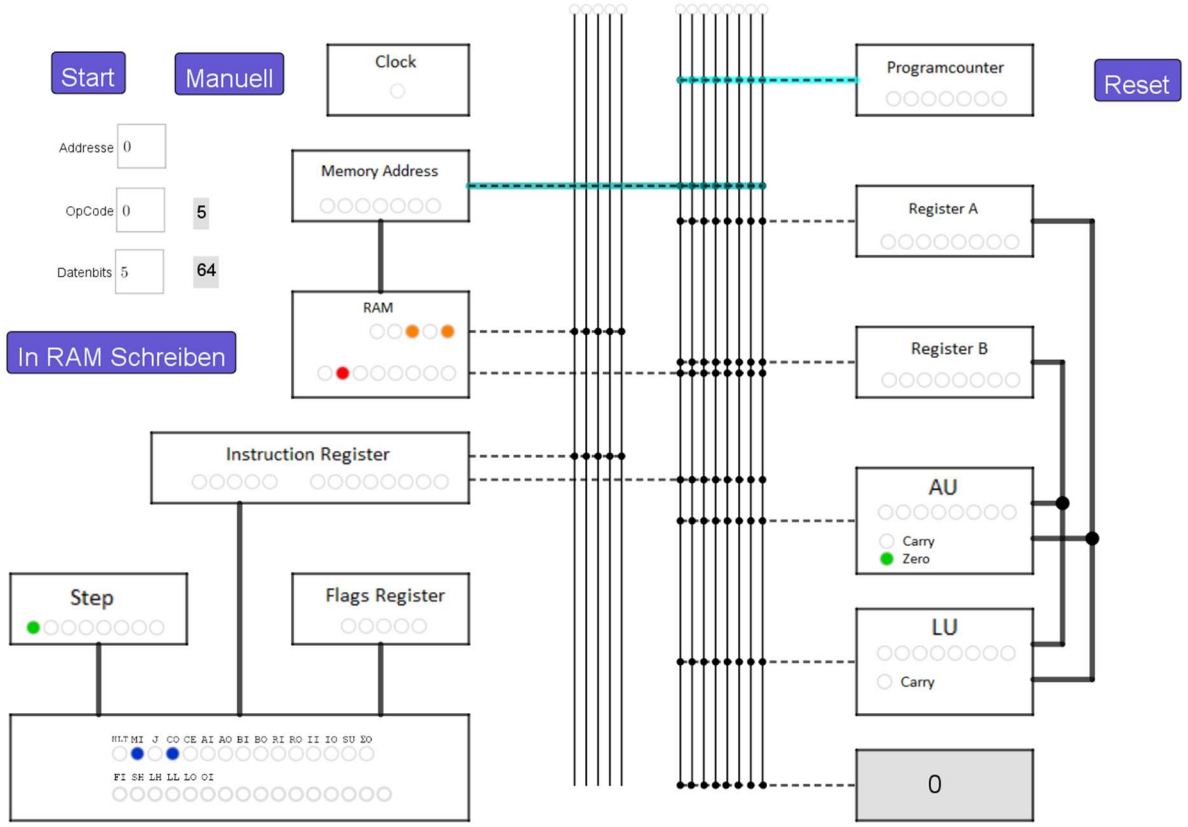
Simulation eines 8-Bit-Computers

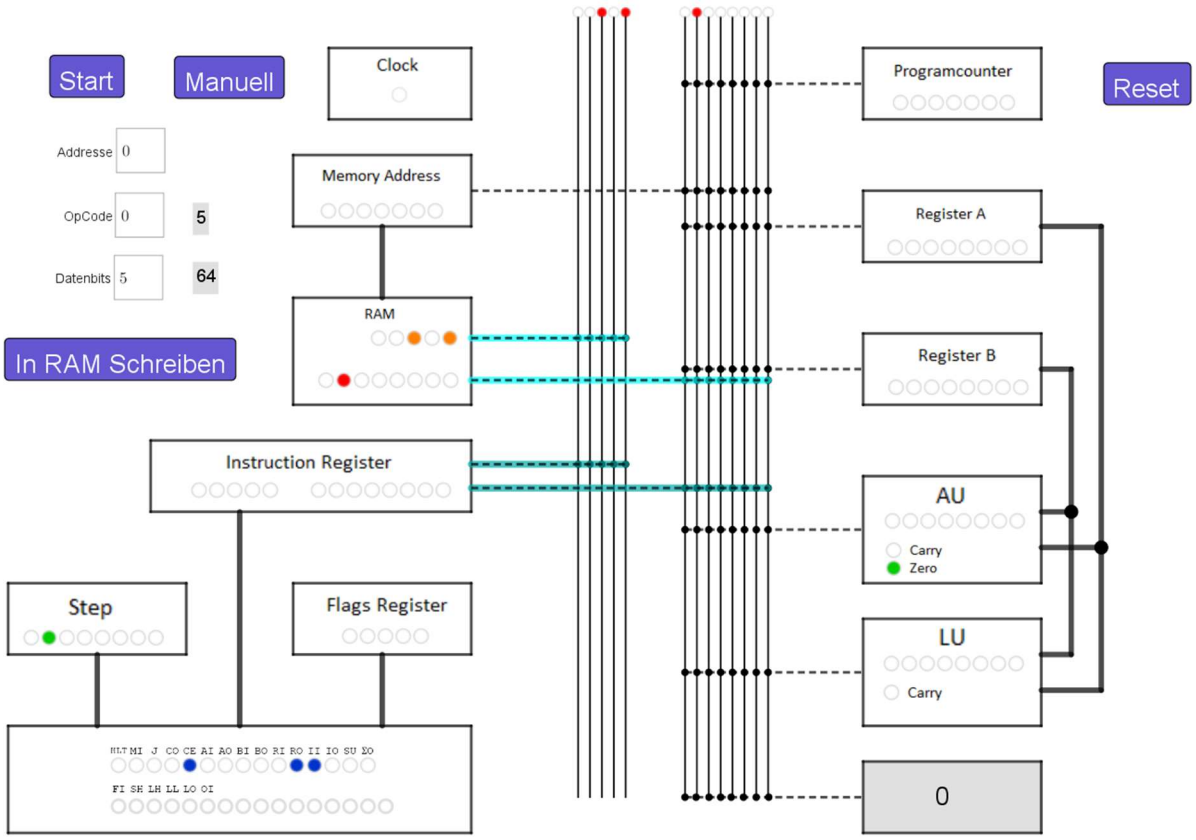
Maschinen- und Mikrobefehle

Die Idee zu dieser Simulation beruht auf den Videos des Youtubers Ben Eater. Die meisten Module existieren auch bereits als auf Platinen gelötete Schaltkreise. Der Vorteil des „physischen Modells“ besteht vor allem in der höheren Ausführungsgeschwindigkeit, aber auch die vorliegende Simulation ist für das Verständnis sehr gewinnbringend. Abweichend von Ben Eaters Vorlage wird der Operationscode auf einem eigenen 5-Bit-Bus vom RAM zum Instruction Register kommuniziert. Rechts davon liegt der 8-Bit-Bus für die Daten und Adressen. So können 32 Maschinenbefehle erstellt und auf einen 128 Adressen umfassenden RAM zugegriffen werden. Auf die Definition eigener Maschinenbefehle mit den vorhandenen Mikrobefehlen wird weiter unten eingegangen. Gleichfalls auf die Erweiterung mit eigenen Modulen und weiteren Mikrobefehlen.



Die Clock gibt den Takt der Programmausführung an. Beim Wechsel von AUS auf AN werden die 13 Bit auf den Bussen in die aktivierten Register geladen. Im Bild oben ist das das Memory Address Register, dessen Verbindung auf den rechten Bus mit einem dunklen Türkis indiziert ist. Links unten im Kontrollwort leuchtet entsprechend die mit `MI` („Memory In“) bezeichnete LED. Wechselt die Clock von AN auf AUS, so wird der Step eins weiter gezählt und es werden die zugehörigen neuen Mikrobefehle im Kontrollwort eingestellt. Das sind etwa die Register, in die beim anschließenden Wechsel der Clock von AUS auf AN geschrieben werden soll, und natürlich diejenigen Register oder Komponenten, die ihren Inhalt auf die Busse geben dürfen.

In obigem Bild also der Programcounter, dessen Verbindung mit dem rechten Bus mit einem hellen Türkis indiziert ist. Links unten im Kontrollwort leuchtet entsprechend die mit `CO` („Counter Out“) bezeichnete LED. Die im ersten Schritt („Step 1“) ausgeführten Mikrobefehle `CO` und `MI` bewirken schlicht, dass der Wert des Programcounter in das Memory Address Register geladen wird. Ersterer zählt der Reihe nach die Adressen des RAM beginnend bei null durch. Sein Wert zeigt gewissermaßen stets diejenige Programmzeile an, die gerade ausgeführt wird. Dagegen zeigt das Memory Address Register direkt auf sie; das RAM stellt immer gerade die 13 Bit des Arbeitsspeichers zur Verfügung, auf deren Adresse das Memory Address Register zeigt. Mit dem Mikrobefehl `RO` können sie auf die beiden Busse geschalten werden. Da im Arbeitsspeicher neben den Befehlen des Programms auch Variablen gespeichert werden, muss die Adresse im Memory Address Register geändert werden können, ohne dass die Informationen über die gerade ausgeführte Programmzeile (vgl. Programcounter) oder den auszuführenden Befehl verloren gehen. Letzteres wird dadurch gewährleistet, dass die 13 Bit des RAM in das Instruction Register kopiert werden; das Memory Address Register wird also durch diese Kopie „freigemacht“. Das nächste Bild zeigt diesen zweiten Schritt (Step 2).



Mit dem Wechsel der Clock von AUS auf AN („steigende Flanke“) wird die Kopie wie oben beschrieben im Instruction Register angelegt. Im Kontrollwort leuchtet außerdem die mit `CE` („Counter enable“) bezeichnete LED. Im anschließenden Halbtakt der Clock wird daher auch der Programcounter eins weiter gezählt und zeigt schon auf den nächsten Befehl. Für alle Maschinenbefehle sind diese ersten beiden Schritte identisch und Variationen geschehen erst im dritten Schritt („Step 3“). In den rechten 8 Bit des Instruction Register kann nun ein Zahlwert oder die Adresse einer Variable im RAM stehen. Im ersten Falle kann der Zahlwert mit `IO` („Instruction Out“) auf den rechten Bus geschalten werden und in der Folge mit `AI` oder `BI` in eines der Register geladen werden, die direkt mit der ALU verbunden sind – im vorliegenden

Modell also genauer mit der Arithmetischen Einheit (AU) bzw. der Logischen Einheit (LU). Eine Adresse des RAM hingegen würde mit `MI` in das Memory Address Register geladen werden, das dann automatisch die unter dieser Adresse hinterlegten Daten aufriefe. Diese könnten wieder mit `RO` auf die Busse geschalten und in eines der Register geladen werden. Folgende Beispiele für Maschinenbefehle sollen diese „Schaltlogik“ verdeutlichen.

`LDA# <Zahl>`

Step 1	`CO`	`MI`	
Step 2	`RO`	`II`	`CE`
Step 3	`IO`	`AI`	

Die im Operanden angegebene Zahl wird direkt in das Register A geladen.

`LDA <Adresse>`

Step 1	`CO`	`MI`	
Step 2	`RO`	`II`	`CE`
Step 3	`IO`	`MI`	
Step 4	`RO`	`AI`	

Die unter der angegebenen Adresse gespeicherte Zahl wird in das Register A geladen.

Für die Ausführung arithmetischer oder logischer Operationen bietet es sich meist an, einen Zahlwert aus dem RAM in das Register B zu laden und das Ergebnis wieder in Register A zu speichern. So sähe etwa der Maschinenbefehl `ADD` aus:

`ADD <Adresse>`

Step 1	`CO`	`MI`	
Step 2	`RO`	`II`	`CE`
Step 3	`IO`	`MI`	
Step 4	`RO`	`BI`	
Step 5	`ΣO`	`AI`	`FI`

Die ersten vier Schritte laden die unter der Adresse gespeicherte Zahl in das Register B (*sie bildeten also den Befehl* `LDB <Adresse>`). Die arithmetische Einheit (AU) bildet stets die Summe der in den Registern A und B gespeicherten Werte und legt sie auf den Mikrobefehl `ΣO` hin auf den rechten Bus. Im fünften Schritt (Step 5) wird weiter das Register A angewiesen, diesen Wert zu speichern. Gleichzeitig wird auch das Flags Register angewiesen, die von AU und LU erzeugten „Flags“ zu speichern. Beide Speichervorgänge geschehen mit der nachfolgenden steigenden Flanke der Clock. Bei genauerem Nachdenken kann man es als problematisch ansehen, dass der Summenwert $a + b$ mit der Speicherung in das Register A von der AU automatisch als Summand interpretiert und die neue Summe $(a + b) + b$ gebildet würde. Genauso verhält es sich auch. Da aber die Speicherung genau mit dem *Wechsel* von AUS zu AN erfolgt, wird das gewünschte Ergebnis nicht mit dieser neuen Summe überschrieben. Auf die Realisierung solcher taktflankengesteuerten Register in der Simulation wird weiter unten noch eingegangen. Die Flags sind einzelne Bits, die das Eintreten eines besonderen Ereignisses vermerken. Für die AU sind dies das Zero-Flag, wenn der Summenwert von Register A und B gleich null ist, und das Carry-Flag, wenn der Summenwert 255 übersteigt. Mit dem Mikrobefehl `SU` kann die AU angewiesen werden, die Differenz der Register A und B zu bilden; das gesetzte Zero-Flag zeigte sodann die Gleichheit der Werte in A und B an. Da sich mit der Speicherung

des Summenwerts in das Register A auch die Werte der Flags ändern könnten, müssen sie für eine weitere Verarbeitung in das Flags Register gespeichert werden. Ein weiterer wichtiger Maschinenbefehl ist die Speicherung eines Ergebnisses in den RAM.

`STA <Adresse>`

Step 1	`CO`	`MI`	
Step 2	`RO`	`II`	`CE`
Step 3	`IO`	`MI`	
Step 4	`AO`	`RI`	

Beim Vergleich mit dem Maschinenbefehl `LDA` fällt sofort ins Auge, dass der einzige Unterschied im vierten Schritt darin besteht, dass die Rollen von RAM und Register A ausgetauscht wurden.

Jetzt fehlen nur noch bedingte Verzweigungen des Programms. Das Kontrollwort, welches mit der fallenden Flanke der Clock alle Register festlegt, welche über die Busse kommunizieren dürfen (*senden bzw. empfangen*), und welche arithmetischen oder logischen Berechnungen in AU und LU erfolgen sollen, wird von drei Zuständen im Prozessor bestimmt. Dies ist der Operationscode im Instruction Register (5 Bit), der Step (3 Bit) und das Flags Register (3 Bit). Der Mikrobefehl `J` könnte auch als `CI` („Counter In“) verstanden werden und weist den Program Counter mit der nächsten steigenden Flanke der Clock an, die auf dem rechten Bus abgelegte Adresse zu speichern. Unabhängig davon, ob ein Sprung bedingt ist oder nicht, lautet die Folge von Mikrobefehlen:

`JMP <Adresse>`

Step 1	`CO`	`MI`	
Step 2	`RO`	`II`	`CE`
Step 3	`IO`	`J`	

Den Unterschied zu einem bedingten Sprung wie `JAC`, der nur ausgeführt wird, wenn die AU das Carry-Flag gesetzt hat, macht allein die Logik zur Bildung des Kontrollworts aus. Für den Fall, dass das Carry-Flag der AU im Flags Register als gesetzt gespeichert ist, wird der gleichlautende `JMP` Befehl ausgeführt. Sonst wird der dritte Schritt durch Nullen ersetzt und die Folge von Mikrobefehlen entspricht dem `NOP` Befehl, also „no operation“ und die Programmausführung geht mit dem nächsten Maschinenbefehl weiter.

Beispielprogramme:

```
  0   ADD  64
  1   OUT
  2   JMP   0
(…)
 64   NOP   5
(…)
```

Beim Neustart werden alle Register auf null zurückgesetzt und die Programmausführung beginnt mit der Adresse null im RAM. Auf den Wert null des Registers A wird der unter der Adresse 64 hinterlegte Wert fünf addiert und in das Register A gespeichert. Der Befehl `OUT` beinhaltet im dritten Schritt die Mikrobefehle `AO` und `OI`. Mit letzterem wird das Output Register

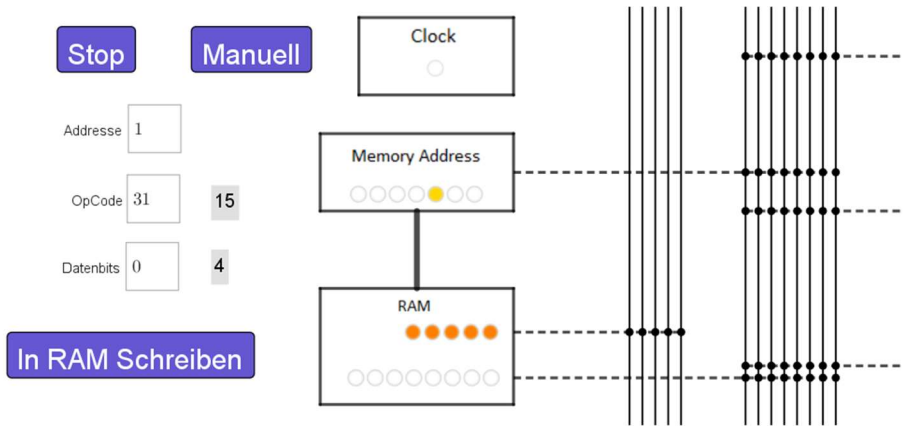
angewiesen, den auf dem rechten Bus abgelegten Wert (*hier also den Inhalt des Registers A*) zu speichern. In der Simulation beschränkt sich die weitere Verarbeitung des Inhalts des Output Registers auf die bloße Anzeige. In der Zukunft sollen hierüber (*mit einem weiteren Mikrobefehl zur Unterscheidung zwischen Daten und Kommandos*) auch anspruchsvollere Ausgaben möglich werden, wie beispielsweise die Angabe von Ergebnis und Rest bei der Division ganzer Zahlen. Abschließend wird der Programmzähler wieder auf null gesetzt und es erfolgt eine weitere Addition der Zahl fünf.

```
  0    ADD  64
  1    JAC   4
  2    OUT
  3    JMP   0
  4    HLT
(...)
 64    NOP  80
(...)
```

Der Beginn ist gleich zum ersten Beispiel, nur dass jetzt pro Schritt 80 addiert werden. In der zweiten Programmzeile geschieht ein bedingter Sprung je nachdem, ob die Summe 255 übersteigt oder nicht. Ist dies nicht der Fall, so läuft das Programm genauso ab, wie im ersten Beispiel. Sonst wird der Programmzähler auf vier gesetzt und dort im RAM ist der Befehl `HLT` abgelegt. Der gleichnamige Mikrobefehl hält die Clock an. Mit einem Reset des Prozessors wird er aufgehoben und das Programm startet neu – bzw. kann in der Simulation mit der Schaltfläche „Start“ erneut ausgeführt werden.

Programme in der Simulation erstellen

Jede Programmzeile wird durch eine Adresse im RAM repräsentiert und besteht aus zwei Zahlenwerten zwischen 0 und 255. Genaugenommen stellt die erste Zahl den Operationscode zwischen 0 und 31 (*entsprechend 5 Bit*) dar und die zweite einen Zahlenwert (8 Bit) oder eine Adresse zwischen 0 und 127 (*entsprechend 7 Bit bzw. der Größe des RAM*). In der Simulation kann der RAM (*anders als in der elektronischen Umsetzung*) unabhängig vom Memory Address Register beschrieben werden. Die an der gewählten Adresse gespeicherten Werte werden rechts von den Eingabefeldern grau hinterlegt angezeigt. Erst nach Betätigung der Schaltfläche „In RAM Schreiben“ werden beide Werte übernommen.



Die nachstehende Tabelle zeigt die voreingestellten Maschinenbefehle mit ihren Operanden und den zugehörigen Operationscode:

0	NOP	XXX	Keine Auswirkungen.
1			
2			
3	LDA#	Zahl	Lade die Zahl direkt in das Register A.
4	LDA	Adresse	Lade die unter Adresse gespeicherte Zahl in das Register A.
5	ADD	Adresse	Addiere die adressierte Zahl auf den Wert in Register A.
6	SUB	Adresse	Subtrahiere die adressierte Zahl von Register A.
7	STA	Adresse	Speichere den Wert von Register A an der Adresse im RAM.
8	RSH	XXX	Shifte Register A um eins nach rechts.
9	LSH	XXX	Shifte Register A um eins nach links.
10	AND	Adresse	Register A & Wert an Adresse in Register A.
11	OR	Adresse	Register A | Wert an Adresse in Register A.
12	INV	XXX	Invertiere Register A.
13			
14	JMP	Adresse	Führe das Programm ab Adresse weiter aus.
15	JAC	Adresse	JMP, wenn AU das Carry-Flag gesetzt hat.
16	JZ	Adresse	JMP, wenn AU das Zero-Flag gesetzt hat.
17	JF3	Adresse	JMP, wenn das dritte Flag des Flag Registers gesetzt wurde.
18	JF4	Adresse	JMP, wenn das vierte Flag des Flag Registers gesetzt wurde.
19	JF5	Adresse	JMP, wenn das fünfte Flag des Flag Registers gesetzt wurde.
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30	OUT	XXX	Kopiere Register A in das Output Register → Ausgabe von A.
31	HLT	XXX	Anhalten der Clock. Ende der Programmausführung.

Die nicht besetzten Operationscodes sind mit `NOP` substituiert, damit Tippfehler nicht zum Programmabbruch oder unvorhersehbaren Veränderungen führen.

Die Realisierung zweier Basistechniken aus C (`if` *und* `for`) könnten wie folgt aussehen:

`if`-Abfrage für $a == b$

```
0   LDA  64
1   SUB  65
2   JZ    5
3   ...
4   JMP   6
5   ...
6   -Fortsetzung-
```

```
(…)
 64   NOP „a“
 65   NOP „b“
(…)
```

Der Wert von a wird in das Register A geladen. Anschließend wird der Wert von b subtrahiert. Wenn die Differenz gleich null ist, also $a == b$ erfüllt ist, dann wird die Programmausführung ab Zeile `5` fortgesetzt und mündet sodann in das weiterlaufende Programm. Hierdurch wir der in C in geschweiften Klammern gesetzte Anweisungsblock realisiert. Sonst wird mit dem Code in Zeile `3` fortgefahren, was dem Anweisungsblock nach `else` entspräche, und es erfolgt ein Sprung auf Zeile `6`, wo das Programm fortgesetzt wird.

`if`-Abfrage für $a \leq b$

```
  0   LDA  64
  1   SUB  65
  2   JAC   5
  3   ...
  4   JMP   6
  5   ...
  6   -Fortsetzung-
(…)
 64   NOP „a“
 65   NOP „b“
(…)
```

Die Subtraktion zweier Zahlen a und b erfolgt als Addition von a und dem Invertierten von b plus eins. Man macht sich hierbei zu Nutze, dass das Register A über 255 hinausgehende Zahlen nicht speichern kann, sondern die letzten acht Bit abschneidet. Es ist $a - b = a + 256 - b = a + 255 - b + 1 = a + \bar{b} + 1$, wobei die „aus der Luft gegriffenen“ 256 als Carry-Flag vermerkt werden – *wenn* $a - b$ eine nicht-negative Zahl ist. Ein zweiter Blick auf die trickreiche Summe zeigt, dass im Fall $a - b < 0$ die Addition von 256 nicht über 255 hinausführt und das Carry-Flag ungesetzt bleibt. Für diesen Fall wäre also $a < b$. So lassen sich mit den beiden Flags der AU alle Relationen von a und b eindeutig bestimmen.

`for`-Schleife

Nachfolgend wird die Schleife `for( a = 3; a < 10; a += 2 )` realisiert. Hierfür werden die benötigten Werte erst in den Speicher geschrieben.

```
  0   LDA#  3              // Initialisierung der Variablen
  1   STA  64
  2   LDA#  2
  3   STA  65
  4   LDA# 10
  5   STA  66
```

```
 6   LDA  64              // Abfrage in der Schleife
 7   SUB  66
 8   JAC  14
 9   ...                  // Schleifenkörper
10   LDA  64              // Inkrementierung
11   ADD  65
12   STA  64
13   JMP   6
14   -Fortsetzung-
(…)
64   NOP „a"
65   NOP „2"
66   NOP„10"
(…)
```

Auf Basis dieser einfachen Routinen lassen sich beispielsweise Suchroutinen für Lösungen diophantischer Gleichungen realisieren. Auch das Rechnen mit Zahlen, die mehr als 8 Bit für ihre Darstellung benötigen, lässt sich umsetzen.

Eine weitere wichtige Technik stellen die Funktionsaufrufe dar. Mittels einer Überschreibung der Sprungadresse des `JMP`-Befehls gelingt der benötigte `Return`. Das nachstehende Beispielprogramm ruft eine Funktion zur Addition zweier Zahlen auf, die als „Parameter" übergeben werden. Der Rückgabewert wird in eine eigene Zelle gespeichert; die entsprechende Zieladresse könnte gleichfalls überschrieben werden…

```
 0   LDA  64          // „Übergabe" der ersten beiden Zahlen
 1   STA  68
 2   LDA  65
 3   STA  69
 4   LDA#  7          // Adresse für Return
 5   STA  35
 6   JMP  32          // Funktionsaufruf
 7   LDA  70          // Laden des Rückgabewerts und Ausgabe
 8   OUT
 9   LDA  66          // „Übergabe" der zweiten beiden Zahlen
10   STA  68
11   LDA  67
12   STA  69
13   LDA# 16          // Adresse für Return
14   STA  35
15   JMP  32          // Funktionsaufruf
16   LDA  70          // Laden des Rückgabewerts und Ausgabe
17   OUT
18   HLT
```

```
(…)
 32    LDA  68          // Laden der Parameter a und b
 33    ADD  69
 34    STA  70          // Speichern des Rückgabewerts
 35    JMP  xx          // Return
(…)
 64    NOP „4“
 65    NOP „8“
 66    NOP „5“
 67    NOP „1“
 68    NOP „a“
 69    NOP „b”
 70    NOP „r”
(…)
```

Die Eingabe der Maschinenbefehle in den RAM ist mühevoll. Daher habe ich eine Schaltfläche „Load“ ergänzt, mit welcher der RAM effizienter beschrieben werden kann. Mein Tip hierzu lautet, dass man erst zeilenweise Adresse, Operationscode und Operand tippt und dann die Zeilenumbrüche wieder löscht.

0, 4, 64,

1, 7, 68,

2, 4, 65,

3, 7, 69,

4, 3, 7,

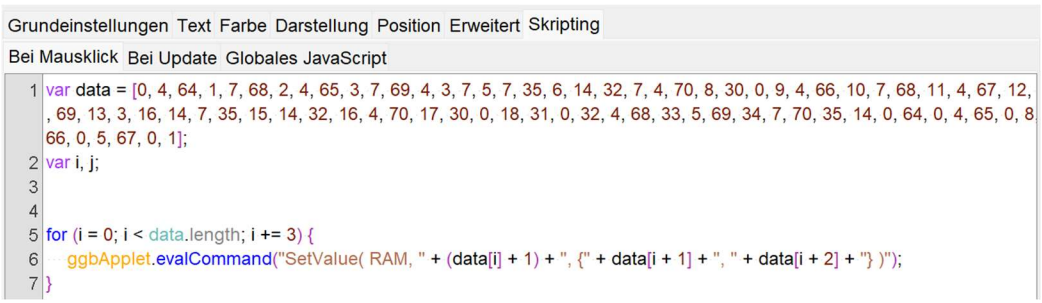
5, 7, 35,

6, 14, 32,

etc.

0, 4, 64, 1, 7, 68, 2, 4, 65, 3, 7, 69, 4, 3, 7, 5, 7, 35, 6, 14, 32, 7, 4, 70, 8, 30, 0, 9, 4, 66, 10, 7, 68, 11, 4, 67, 12, 7, 69, 13, 3, 16, 14, 7, 35, 15, 14, 32, 16, 4, 70, 17, 30, 0, 18, 31, 0, 32, 4, 68, 33, 5, 69, 34, 7, 70, 35, 14, 0, 64, 0, 4, 65, 0, 8, 66, 0, 5, 67, 0, 1

Nach diesen Vorbereitungen macht man einen Rechtsklick auf die Schaltfläche „Load“, ruft die Eigenschaften und dann den Reiter „Skripting“ – „Bei Mausklick“ auf. Sodann kopiert man die Zahlenfolge in die eckigen Klammern von `var data = [];` in der ersten Zeile.



Nach dem Schließen des Fensters und Betätigung der Schaltfläche „Load“ wird der RAM weisungsgemäß überschrieben.

Eigene Maschinenbefehle

In der Simulation sind die Maschinenbefehle als Listen OP-Nr.-, also OP00, OP01 etc., angelegt. Sie enthalten selbst wieder Listen der Mikrobefehle jeden Schritts. Jedem Mikrobefehl entspricht eine gleichnamige gestrichene Variable, deren Wert der Stelle im Kontrollwort entspricht. Zum Beispiel {{CO', MI'}, {RO', II', CE'}, {IO', AI'}} für den Befehl `LDA#`. In der Algebra-Ansicht von *geogebra* steht lediglich {{4,2}, {12,13,5}, {14,6}}. Nachstehende Tabelle zeigt die bereits besetzten Mikrobefehle:

1	`HLT`	Anhalten der Clock.
2	`MI`	Memory Address Register In
3	`J`	Program Counter In
4	`CO`	Program Counter Out
5	`CE`	Program Counter Enable
6	`AI`	Register A In
7	`AO`	Register A Out
8	`BI`	Register B In
9	`XXX`	XXX
10	`BO`	Register B Out
11	`RI`	RAM In (*nur rechter Bus*)
12	`RO`	RAM Out
13	`II`	Instruction Register In
14	`IO`	Instruction Register Out (*nur rechter Bus*)
15	`SU`	AU: Subtraktion
16	`EO`	AU: Sum Out
17	`FI`	Flags Register In
18	`SH`	Shift-Richtung; H: links; L: rechts
19	`LH`	LU: High-Bit
20	`LL`	LU: Low-Bit
		(LH, LL) = (0, 0): Shift (LH, LL) = (0, 1): AND (LH, LL) = (1, 0): OR (LH, LL) = (1, 1): Inverse
21	`LO`	LU: Result Out
22	`OI`	Output Register In
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		

Eine Sonderstellung nimmt die Nummer 9 ein. Im elektronischen Modell ist das zugehörige Bit fest verdrahtet mit dem Reset des Step-Zählers und erfüllt die Rolle eines „Skip“ am Ende der Liste von Mikrobefehlen. Da die Maschinenbefehle meist weniger als acht Schritte benötigen, aber auch unterschiedlich lang sind, werden so sonst „leere Takte“ der Clock eingespart. In der Simulation kann dies einfach durch die Länge der OP-Nr.- Listen gesteuert werden. Wie oben schon ausgeführt, sind die ersten beiden Schritte obligatorisch. Bis zu sechs weitere Schritte können nach Belieben ergänzt werden.

Eigene Mikrobefehle

Die Clock ist in der Simulation repräsentiert durch die Variablen t und u, deren Werte vom Schieberegler s bestimmt werden, der wiederholt die Werte von null bis eins durchläuft. So ist t für alle Werte $s \geq 0{,}5$ gleich eins und null sonst. Weiter ist $u = 1 - t$, im binären Sinne also das Inverse von t. Wenn t seinen Wert von null auf eins wechselt, so entspricht das der „steigenden Flanke“ der Clock und das unter t abgelegte Skript wird ausgeführt. Gleichzeitig wird der Wert von tFlanke auf eins gesetzt, damit es bei einem einmaligen Aufruf bleibt, bis der Wert erneut von null auf eins wechselt. Entsprechend verhält es sich mit der Variablen u, unter welcher gleichfalls ein Skript abgelegt ist.

Skript unter u

Hier werden diejenigen Mikrobefehle verwaltet, die Registerinhalte auf die Busse schreiben. Möchte man etwa ein `DO` („Register D Out“) für ein neues Register D einrichten, so muss dieses Skript erweitert werden. In *geogebra* wird die Variable DO‘ ($= 24$ *oder eine andere freie Nummer*) angelegt und ihr Wert für das Skript beispielsweise in die Variable N_DO eingelesen. Für einen Vermerk, ob `DO` im Kontrollwort enthalten ist, ergänzt man im Skript noch die Variable DO. Der Block, in welchem das Kontrollwort auf `DO` hin geprüft wird, kann quasi wortgleich von einem der anderen Blöcke übernommen werden. Zu guter Letzt ist die Prüfsumme, die sicherstellt, dass immer nur eine Komponente ihre Daten auf die Busse legt, noch um den Summanden DO zu ergänzen.

```
var u = ggbApplet.getValue("u");
var uF = ggbApplet.getValue("uFlanke");

if (u == 1 && uF == 0) // steigende Flanke der invertierten Clock
{
   ggbApplet.setValue("uFlanke", 1);

   var S = ggbApplet.getValue("Step");
   var L = ggbApplet.getValue("Length(OP)");

   // Inkrement von Step (Startwert bei null...)
   if ((S + 1) + 1 > L) {
      S = 0;
   } else {
      S += 1;
   }
   ggbApplet.setValue("Step", S);

   var N_CO = ggbApplet.getValue("CO'");
   var N_RO = ggbApplet.getValue("RO'");
   var N_IO = ggbApplet.getValue("IO'");
   var N_AO = ggbApplet.getValue("AO'");
```

```
var N_EO = ggbApplet.getValue("EO'");
var N_LO = ggbApplet.getValue("LO'");

var b;
var CO, RO, IO, AO, EO, LO;
var BusD, BusOp;

// Prüfe, ob Ausgaben auf den Bus geschehen
b = ggbApplet.getValue("IndexOf(" + N_CO + ", Kontrollwort)"); // CO
if (b > 0) {
    CO = 1;
    BusD = ggbApplet.getValue("Counter");
    BusOp = 0;
} else {
    CO = 0;
}
b = ggbApplet.getValue("IndexOf(" + N_RO + ", Kontrollwort)"); // RO
if (b > 0) {
    RO = 1;
    BusD = ggbApplet.getValue("Element(RAMAkt, 2)");
    BusOp = ggbApplet.getValue("Element(RAMAkt, 1)");
} else {
    RO = 0;
}
b = ggbApplet.getValue("IndexOf(" + N_IO + ", Kontrollwort)"); // IO
if (b > 0) {
    IO = 1;
    BusD = ggbApplet.getValue("IData");
    BusOp = 0;
} else {
    IO = 0;
}
b = ggbApplet.getValue("IndexOf(" + N_AO + ", Kontrollwort)"); // AO
if (b > 0) {
    AO = 1;
    BusD = ggbApplet.getValue("AData");
    BusOp = 0;
} else {
    AO = 0;
}
b = ggbApplet.getValue("IndexOf(" + N_EO + ", Kontrollwort)"); // EO
if (b > 0) {
    EO = 1;
    BusD = ggbApplet.getValue("SumData");
    BusOp = 0;
} else {
    EO = 0;
}
b = ggbApplet.getValue("IndexOf(" + N_LO + ", Kontrollwort)"); // LO
if (b > 0) {
    LO = 1;
    BusD = ggbApplet.getValue("LogData");
    BusOp = 0;
} else {
    LO = 0;
}
```

```
    if (CO + RO + IO + AO + EO + LO == 1) {
        ggbApplet.setValue("BusData", BusD);
        ggbApplet.setValue("BusOpCode", BusOp);
    } else {
        ggbApplet.setValue("BusData", 0);
        ggbApplet.setValue("BusOpCode", 0);
    }

    // Wenn das HLT-Bit gesetzt ist, dann wird die Clock angehalten

    var N_HLT = ggbApplet.getValue("HLT'");
    var HLT;

    b = ggbApplet.getValue("IndexOf(" + N_HLT + ", Kontrollwort)"); // HLT
    if (b > 0) {
        HLT = 1;
        ggbApplet.evalCommand("StartAnimation(s, false)");
    } else {
        HLT = 0;
    }
}

if (u == 0 && uF == 1) {
    ggbApplet.setValue("uFlanke", 0);
}
```

Skript unter *t*

Hier werden diejenigen Mikrobefehle verwaltet, die Daten von den Bussen in die Register schreiben. Möchte man etwa ein `DI` („Register D In“) für ein neues Register D einrichten, so muss dieses Skript erweitert werden. In *geogebra* wird die Variable DI‘ (= 23 *oder eine andere freie Nummer*) angelegt und ihr Wert für das Skript beispielsweise in die Variable N_DI eingelesen. Für einen Vermerk, ob `DI` im Kontrollwort enthalten ist, ergänzt man im Skript noch die Variable DI. Der Block, in welchem das Kontrollwort auf `DI` hin geprüft wird, kann wieder quasi wortgleich von einem der anderen Blöcke übernommen werden. Da mehrere Register die gleichen Daten aufnehmen dürfen, entfällt hier eine Kontrollsumme. Außerdem werden hier die bedingten Sprünge bearbeitet, wofür vor allem das Flags Register anzupassen ist.

```
var t = ggbApplet.getValue("t");
var tF = ggbApplet.getValue("tFlanke");

if (t == 1 && tF == 0) // steigende Flanke der Clock
{
    ggbApplet.setValue("tFlanke", 1);

    var N_MI = ggbApplet.getValue("MI'");
    var N_II = ggbApplet.getValue("II'");
    var N_AI = ggbApplet.getValue("AI'");
    var N_BI = ggbApplet.getValue("BI'");
    var N_FI = ggbApplet.getValue("FI'");
    var N_RI = ggbApplet.getValue("RI'");
    var N_OI = ggbApplet.getValue("OI'");

    var b, flag;
    var MI, II, AI, BI, FI, RI, OI;
    var CE;
```

```
    var BusD = ggbApplet.getValue("BusData");
    var BusOp = ggbApplet.getValue("BusOpCode");;

    // Mit steigender Flanke der Clock werden alle Register GLEICHZEITIG aktualisiert; da AI und BI zu
Änderungen der Flags führen, muss FI vor allen anderen ausgeführt werden
    b = ggbApplet.getValue("IndexOf(" + N_FI + ", Kontrollwort)"); // FI
    if (b > 0) {
        FI = 1;
        flag = ggbApplet.getValue("FlagCarry");
        ggbApplet.evalCommand("SetValue( FlagsBin, 1, " + flag + " )");
        flag = ggbApplet.getValue("FlagZero");
        ggbApplet.evalCommand("SetValue( FlagsBin, 2, " + flag + " )");
        flag = ggbApplet.getValue("FlagLogCarry");
        ggbApplet.evalCommand("SetValue( FlagsBin, 3, " + flag + " )");
    } else {
        FI = 0;
    }

    // Schreibe Daten vom Bus in die freigeschalteten Register
    b = ggbApplet.getValue("IndexOf(" + N_MI + ", Kontrollwort)"); // MI
    if (b > 0) {
        MI = 1;
        ggbApplet.setValue("Address", BusD);
    } else {
        MI = 0;
    }
    b = ggbApplet.getValue("IndexOf(" + N_II + ", Kontrollwort)"); // II
    if (b > 0) {
        II = 1;
        ggbApplet.setValue("IData", BusD);
        ggbApplet.setValue("IOpCode", BusOp);
    } else {
        II = 0;
    }
    b = ggbApplet.getValue("IndexOf(" + N_AI + ", Kontrollwort)"); // AI
    if (b > 0) {
        AI = 1;
        ggbApplet.setValue("AData", BusD);
    } else {
        AI = 0;
    }
    b = ggbApplet.getValue("IndexOf(" + N_BI + ", Kontrollwort)"); // BI
    if (b > 0) {
        BI = 1;
        ggbApplet.setValue("BData", BusD);
    } else {
        BI = 0;
    }
    b = ggbApplet.getValue("IndexOf(" + N_OI + ", Kontrollwort)"); // OI
    if (b > 0) {
        OI = 1;
        ggbApplet.setValue("Output", BusD);
    } else {
        OI = 0;
    }
```

```
// Schreibe Daten in den RAM
b = ggbApplet.getValue("IndexOf(" + N_RI + ", Kontrollwort)"); // RI
if (b > 0) {
    var Add = ggbApplet.getValue("Address");
    var Op = ggbApplet.getValue("Element(Element(RAM, " + (Add + 1) + "), 1)");

    RI = 1;
    ggbApplet.evalCommand("SetValue( RAM, " + (Add + 1) + ", {" + Op + ", " + BusD + "} )");
} else {
    RI = 0;
}

// Counter hochzählen, wenn CE gesetzt
var N_CE = ggbApplet.getValue("CE'");
var C = ggbApplet.getValue("Counter");

b = ggbApplet.getValue("IndexOf(" + N_CE + ", Kontrollwort)"); // CE
if (b > 0) {
    CE = 1;
    ggbApplet.setValue("Counter", C + 1);
} else {
    CE = 0;
}

// Counter setzen, wenn J gesetzt - und Sprung "unbedingt" oder zugehöriges Flag gesetzt ist
var N_J = ggbApplet.getValue("J'");
var Op_JAC = ggbApplet.getValue("JAC'");
var Op_JZ = ggbApplet.getValue("JZ'");
var Op_JF3 = ggbApplet.getValue("JF3'");
var Op_JF4 = ggbApplet.getValue("JF4'");
var Op_JF5 = ggbApplet.getValue("JF5'");

var J;

b = ggbApplet.getValue("IndexOf(" + N_J + ", Kontrollwort)"); // J
if (b > 0) {
    var Op = ggbApplet.getValue("OpCode");

    if (Op != Op_JAC && Op != Op_JZ && Op != Op_JF3 && Op != Op_JF4 && Op != Op_JF5) //
"Unbedingter Sprung"
    {
        J = 1;
        ggbApplet.setValue("Counter", BusD);
    } else {
        J = 0; // Wird auf Eins gesetzt, wenn das passende Flag gesetzt ist

        if (Op == Op_JAC && ggbApplet.getValue("Element(FlagsBin, 1)") == 1) // Bedingter Sprung
nach Carry-Bit in AU
        {
            J = 1;
            ggbApplet.setValue("Counter", BusD);
        }
        if (Op == Op_JZ && ggbApplet.getValue("Element(FlagsBin, 2)") == 1) // Bedingter Sprung nach
Zero-Bit in AU
        {
            J = 1;
            ggbApplet.setValue("Counter", BusD);
```

```
        }
        if (Op == Op_JF3 && ggbApplet.getValue("Element(FlagsBin, 3)") == 1) // Bedingter Sprung
        {
            J = 1;
            ggbApplet.setValue("Counter", BusD);
        }
        if (Op == Op_JF4 && ggbApplet.getValue("Element(FlagsBin, 4)") == 1) // Bedingter Sprung
        {
            J = 1;
            ggbApplet.setValue("Counter", BusD);
        }
        if (Op == Op_JF5 && ggbApplet.getValue("Element(FlagsBin, 5)") == 1) // Bedingter Sprung
        {
            J = 1;
            ggbApplet.setValue("Counter", BusD);
        }

    }
  } else {
    J = 0;
  }
}

if (t == 0 && tF == 1) {
  ggbApplet.setValue("tFlanke", 0);
}
```

Flags Register

Das Zero-Flag bzw. das Carry-Flag der AU heißen in *geogebra*
FlagZero = Wenn(SumData $\stackrel{?}{=}$ 0, 1, 0) bzw.
FlagCarry = Wenn(SU $\stackrel{?}{=}$ 0, Wenn(AData + BData > 255, 1, 0), Wenn(AData - BData < 0, 0, 1)).
Weiter ist noch das Carry-Flag der LU eingerichtet als
FlagLogCarry = Wenn(SH $\stackrel{?}{=}$ 0, Element(ADataBin, Länge(ADataBin)), Element(ADataBin, 1)).
Damit sind noch zwei weitere Flags offen für eigene Definitionen. Sie werden in *geogebra* nicht von anderen Funktionen gesetzt, sondern wie an den obigen Beispielen ersichtlich errechnet. Die Einbindung in das Skript unter t ist selbsterklärend.

```
// Mit steigender Flanke der Clock werden alle Register GLEICHZEITIG aktuali
b = ggbApplet.getValue("IndexOf(" + N_FI + ", Kontrollwort)"); // FI
if (b > 0) {
    FI = 1;
    flag = ggbApplet.getValue("FlagCarry");
    ggbApplet.evalCommand("SetValue( FlagsBin, 1, " + flag + " )");
    flag = ggbApplet.getValue("FlagZero");
    ggbApplet.evalCommand("SetValue( FlagsBin, 2, " + flag + " )");
    flag = ggbApplet.getValue("FlagLogCarry");
    ggbApplet.evalCommand("SetValue( FlagsBin, 3, " + flag + " )");
} else {
    FI = 0;
}
```

Weitere Register

Das Flags Register nimmt insofern eine Sonderrolle ein, als dass es als eine Liste von Nullen und Einsen geführt wird. So ist auch der Name „FlagsBin“ mit dem Zusatz „Bin“ zu verstehen. Für arithmetische Operationen ist es aber zweckmäßiger, die Registerinhalte von A und B als Dezimalzahlen zu führen, die von *geogebra* mühelos verrechnet werden können. Konsequenterweise gilt dies dann auch für die Busse und den RAM und die weiteren Register. Die zugehörigen Variablen heißen zum Beispiel AData und BData. Für die LU und die Steuerung der LEDs ist es dann wiederum günstiger, die entsprechenden binären Darstellungen als Listen (*mit führenden Nullen*) vorliegen zu haben. Für die Umwandlung stellt *geogebra* sehr nützliche Werkzeuge bereit. Zunächst wird die gegebene Dezimalzahl in eine Binärzahl transformiert. Der Rückgabewert ist ein Text, der in eine Liste von Unicode Zeichen übersetzt wird.

```
TextZuUnicode(ZuBasis(AData, 2))
```

Zum Beispiel würde AData = 89 übersetzt in den Text “1011001“ und dann in die Liste {49, 48, 49, 49, 48, 48, 49}. Letztere wird gespeichert als ADataBin‘. Mit dem Befehl Folge(Wenn(k > 8 - Länge(ADataBin'), Element(ADataBin', k + Länge(ADataBin') - 8) - 48, 0), k, 1, 8, 1) wird sie in die gewünschte Form einer 8-Bit-Zahl in Listenform transformiert. Das Ergebnis ist die Liste ADataBin = {0, 1, 0, 1, 1, 0, 0, 1}. Für Adressen werden 7-Bit-Zahlen erzeugt und für den Operationscode 5-Bit-Zahlen.

Bemerkungen

Eine Erweiterung um ein Modul zur Multiplikation von A und B wäre durchaus wünschenswert. Außerdem eine Berechnung von $\sqrt{2}$ mit dem Heron-Verfahren auf Taschenrechnergenauigkeit. Allerdings ist erstaunlich, dass *geogebra* es überhaupt schafft, mit der schon unfassbar großen Anzahl an Objekten (*vor allem Listen*) umzugehen. Bei höheren Geschwindigkeiten der Clock (*steuerbar über die Variable* v) und Laufzeiten über einer Minute brachen jedoch die eine oder andere Liste auch schon mal weg – sie wurden schlicht gelöscht. Man sollte daher vor Testläufen mit Erweiterungen unbedingt speichern! Abschließend möchte ich noch einmal Ben Eater meinen Dank aussprechen für ein didaktisch äußerst geschickt angelegtes Tutorial, das unter https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU (*aufgerufen am 26.3.2024 um 8:34*) zu finden ist. Mir ist in der Folge ein Kindheitstraum in Erfüllung gegangen, aber ich habe mich bewusst für verlötete Platinen anstelle von Breadboards entschieden. Weiter habe ich die Architektur des Mikroprozessors notgedrungen angepasst, weil ich keinen RAM mit 16 Byte fand, sondern nur solchen mit 128 Byte. Ich wünsche viel Spaß mit der Simulation und werde vielleicht zu einem späteren Zeitpunkt Ergänzungen zur elektronischen Umsetzung nachreichen.