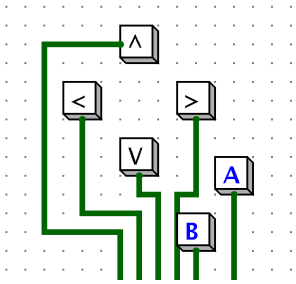
Projekt: Entwurf einer 8bit-Computer-Architektur in Logisim

Ein Bericht von Andreas Dzialocha



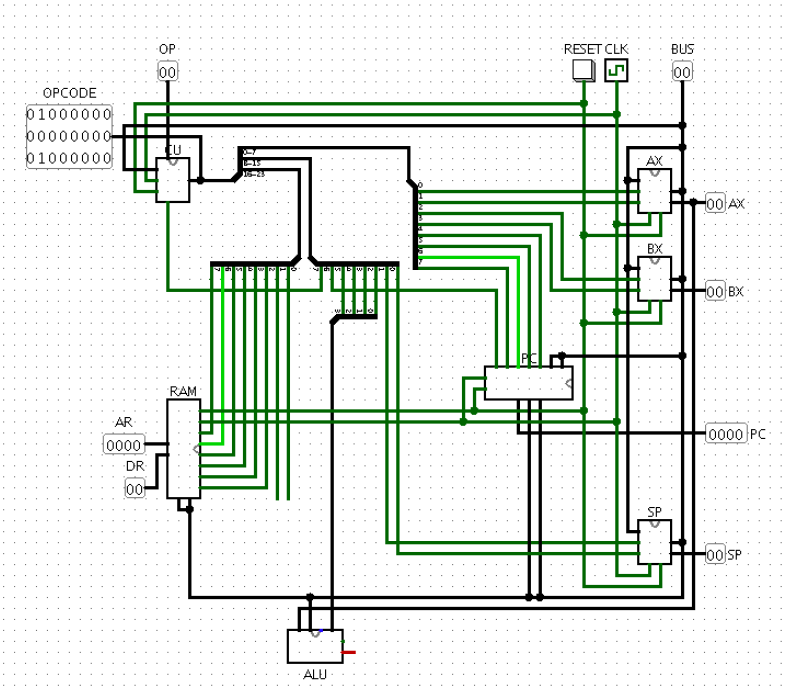
Matrikel-Nr. 539792 - BA Zweitfach Medienwissenschaft - Modul VI - Projektmodul (PO 2014) - Datum: 28.12.2017 (WS 17/18) - Prüfer: Herr Dr. Stefan Höltgen - Humboldt Universität zu Berlin - Kultur-, Sozial- und Bildungswissenschaftliche Fakultät

Inhaltsverzeichnis

Einführung

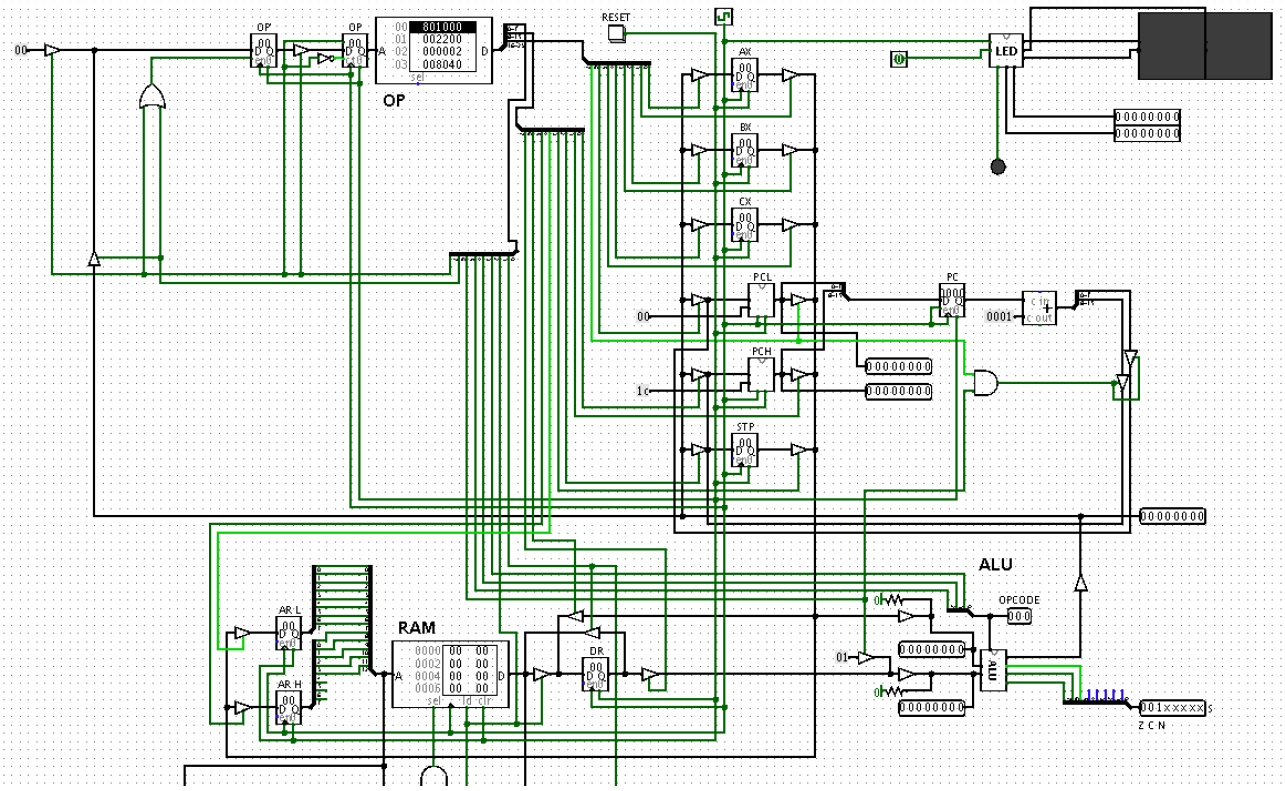
Das Projekt ist ein Versuch, eine eigene Computer-Architektur zu entwickeln, um so die grundlegenden Prozesse auch heutiger Rechner besser zu verstehen. Das System sollte der Ausführung komplexerer Programme dienen, also nicht »nur bytes von A nach B schieben«, sondern am besten sogar einfache Computerspiele ausführen. Somit war klar, dass das System eine Spieler-Eingabe (Tastatur oder Joystick) und graphische Ausgabe benötigt (LED Matrix) und relativ hohe Ansprüche an die Architektur haben wird. In meiner ersten Vorstellung sollten mindestens folgende Komponenten vorhanden sein:

- 8bit-Datenbus-Architektur mit 64 kB adressierbaren Arbeitsspeicher. Nach dem dritten Versuch habe ich diese Idee verworfen und mich für einen 256 byte Arbeitspeicher entschieden
- Verschiedene Adressierungsarten (absolut, unmittelbar, indirekt, indexiert)
- Ein Stackpointer-Register, um Sprünge in und aus Unterprogrammen zu ermöglichen oder Daten im Stack abzulegen
- branch Befehle für Schleifen oder Konditionen
- Zwei Arbeitsregister. Nach einiger Zeit fiel auf, dass ein weiteres Ergebnisregister für die ALU noch zusätzlich benötigt wird
- Eine ALU (Arithmetisch-logische Einheit), welche die gängigen mathematischen- und bit-Operationen beherrscht
- Ein Controller, welcher Tastatureingaben liest und in einen Buffer schreibt. Anfangs war eine Interrupt-Routine (IRQ und ISR) geplant, welche aber später verworfen wurde
- Ein Controller, welcher LED Pixel ansteuert und einfach zu programmieren ist



Zweiter Versuch (ca. Mitte 2017), etwas sauberer als der erste Versuch, hier noch ohne funktionierender ALU

Fast alle diese Punkte konnten mit Umwegen umgesetzt werden, manchmal musste ich sie aber vereinfachen, da die Komplexität oder der Umfang zu groß wurde. Dieser Bericht soll auch einige dieser Stolpersteine und Umentscheidungen beschreiben.



Erster Versuch (ca. Mitte 2016) mit 64 kB Arbeitsspeicher (siehe AR L und AR H Register unten links für 16 bit RAM-Adressen)

Das Projekt ist über einen langen Zeitraum entstanden. Schon Mitte 2016 sind die ersten Versuche passiert. Insgesamt sind drei verschiedene Versuche geschehen, wobei der letzte der endgültige ist und ich mich auch hauptsächlich auf diesen im Bericht berufen werde. Die finale Architektur wurde durch die Vorarbeit und Vereinfachung der Problemstellung innerhalb von circa zwei Wochen entwickelt, relativ kurz im Vergleich zum Umfang der bisherigen Versuchen.

Mit der Entwicklung des Computers sind einige weitere Nebenprojekte entstanden, welche mir die Arbeit sehr erleichtert haben und ich auch im Bericht zu sprechen drauf komme. Unter anderem ist so ein Editor entstanden, welcher mir bei der Programmierung der micro instructions und der mühsamen Arbeit des »bit-Setzen« half und außerdem konnte ich auch einen Assemblierer schreiben, welcher Assemblersprache in den für meine Architektur verständlichen Maschinencode übersetzt, was die Entwicklung von Programmen sehr vereinfacht. Alle Notizen, Projektdaten (auch die ersten Versuche), Codes usw. sind in einem GitHub-Repository festgehalten und können unter https://github.com/adzialocha/8bit eingesehen werden.

Ziel dabei war es, sich nicht nur mit bestehenden Architekturen auseinanderzusetzen, sondern auch »eigene« Lösungsansätze für Probleme zu finden, die bei der Entstehung auftauchten.

Insofern ist das Ergebnis des Projekts sicherlich kein »idealer« Weg eine Computer-Architektur zu entwerfen, sondern vielmehr ein Lernprozess mit der selbstauferlegten (und manchmal gebrochenen) Regel, nicht zu viel von bestehenden System abzuschauen, auch wenn es sich an vielen Stellen nicht leugnen lässt, dass die 6502 bzw. 6510 Architektur durch mein bisheriges Vorwissen oft als Quelle diente. Auch mein vorhergehendes Studium der Informatik, vor allem das Modul »Digitale Systeme« und mein Informatik-Kurs der Schulzeit half mir sehr.

Dieser Bericht soll als grobe Zusammenfassung des Projekts dienen. So möchte ich auf die einzelnen Bestandteile der Architektur und ggf. ihrer Besonderheiten eingehen. Zuletzt stelle ich einen »snake« clone vor, welcher mithilfe des Assemblers entwickelt wurde und auf dem Computer zu spielen ist.

Hinweis: Alle Binärzahlen sind »Most Significant Bit First« (0x12 = 0001 0010).

Logisim als Versuchsumgebung

Als Versuchsumgebung habe ich Logisim von Carl Burch in der Version von 2011 verwendet (http://www.cburch.com/logisim/), so wie ich sie auch schon zur Schulzeit kennengelernt habe (im Studium wurde Logisim nicht verwendet). Die Version is veraltet, lief aber trotz einiger Abstürze und bugs relativ stabil. Für zukünftige Projekte möchte ich die modernere Variante »Logisim evolution« empfehlen (https://github.com/reds-heig/logisim-evolution), welche nach Aufgabe von Burch durch Schweizer Hochschulen geforkt und weiterentwickelt wurde. Für mein eigenes Projekt bin ich bei der Original-Version geblieben, da die Umstellung zwar problemlos, aber zu umfangreich war (die Objekte haben neue Größen erhalten) - insofern laufen alle meine Logisim-Schaltungen mit der letzten Version von 2011. Das Projekt verwendet ausschließlich Bauelemente, welche in der Grundversion vorhanden sind, also logische Gatter, J/K Flip-Flops, einfache ROM und RAM-Elemente.

Logisim erlaubt die ausschließliche Auseinandersetzung mit logischen Schaltkreisen, sodass Fragen nach Stromversorgung, Spannungen, Taktungen usw. ausbleiben. Trotz der einfachen Umgebungsoberfläche können sehr komplexe Projekte in Logisim aufgebaut werden, vor allem ist das durch die Modularisierung von Schaltungen in eigene »ICs« gut gelöst.

Entwurf einer 8bit-Computer-Architektur

Problemstellung

Die ersten beiden Versuche eine 8bit-Computer-Architektur zu entwickeln scheiterten an meiner Herangehensweise an das Problem. Zunächst versuchte ich eine Architektur zu entwerfen, indem ich logischen Schaltkreise auf der Mikroebene entwickelte, also Probleme löste, wie »wie bringe ich bytes von Register AX nach BX«, »wie lese ich am besten Daten aus dem Arbeitsspeicher«, »wie wird der program counter zurückgesetzt«.

Das funktionierte so lange gut, bis komplexere Probleme auftauchten. Im ersten und zweiten Versuch konnte ich schon Programme ausführen, welche einen frei adressierbaren 64 kB Speicher mit 8 bit Werten beschreiben, auslesen und sogar einfache mathematische mit Ihnen ausführen konnten. Es bestand sogar eine Aufteilung des Speichers in einen Kernel-Bereich, welcher die Startroutine behandelte (den program counter auf die richtige Startadresse setzte) und einem ROM Bereich, wo später das statische Programm liegen sollte (inspiriert aus cartridges von Konsolen wie z.B. der »Super Nintendo«). Trotzdem gelang es mir nicht mit dieser Denkweise umfangreichere Zusammenhänge zu lösen, wie »wie springe ich, wenn die zero-flag gesetzt ist, an eine andere Programmstelle«, was plötzlich mehrere Komponenten mit einbezog. Das Problem wurde mir zu abstrakt, die Stellschrauben zu viele und ich wusste nicht wo ich ansetzen sollte, ohne alles nochmal neu anzufassen. Im zweiten Versuch habe ich den Neuversuch mit derselben Methode nochmal gestartet, in dem Glauben, dass ich so nochmal alles durchdenken kann, aber die Herangehensweise nur über die Schaltung lief wieder in eine Sackgasse und ich musste an derselben Stelle (branch instructions) aufhören.

Auch das Denken in 16 bit Adressen in einer 8 bit Architektur erschwerte den Prozess. Fast jeder Opcode und jede Schaltung musste das Rechnen mit Binärzahlen bis 16 bit ermöglichen, obwohl immer nur 8 bit gleichzeitig zur Verfügung standen. Dieser zusätzliche Denkschritt bei jedem Problem machte es noch schwieriger.

Im dritten Versuch nahm ich mir vor 1. die Architektur auf 8 bit Adressen zu beschränken 2. den Computer nicht vonseiten der Schaltung zu denken, sondern aus Sicht der Software die auf ihm laufen soll. Insofern wählte ich mir zuerst eine praktische Anwendung, welche dann von einer erstmal unbekannten Architektur ausgeführt werden sollte.

So schrieb ich zunächst ein einfaches Programm (*asm/6502-test.asm*) in 6502 Assembler (mit Hilfe der Seite http://rtro.de/):

```
; Beispielprogramm in 6502 Assembler

; Initialisierung

LDA #$00
STA $01           ; Spieler Position

; Haupt-Routine

main:
  JSR draw        ; Zeichne aktuelle Spieler Position
  JMP check_input ; Überprüfe Tastatur-Eingaben
JMP main

; Routine: Spieler zeichnen

draw:
  LDX $01         ; Lade Spieler-Position in X
  LDA #$01        ; Lade Farbe Weiß in A
  STA $0200,X     ; Zeiche Spieler mit Farbe aus A an Position X
RTS

; Routine: Spieler löschen

clear:
  LDX $01         ; Lade Spieler-Position in X
  LDA #$00        ; Lade Farbe Schwarz in A
  STA $0200,X     ; Schreibe schwarz (leer) an Stelle des Spielers
RTS

; Routine: Überprüfe Tastatur-Eingabe

check_input:
  LDA $FF         ; Tastaturbuffer abfragen
  CMP #$61        ; "A"?
  BEQ left
  CMP #$64        ; "D"?
  BEQ right
  JMP main        ; .. zurück zum Hauptprogramm wenn nichts passiert ist

; Routine: Tastaturbuffer löschen

clear_input:
  LDA #0          ; Tastaturbuffer ..
  STA $ff         ; .. löschen
RTS

; Routine: Spieler nach links bewegen

left:
  JSR clear_input
  JSR clear       ; Lösche letzte Position auf Bildschirm
  LDA $01         ; Lade aktuelle Position in A und
  TAX             ; .. kopiere nach X
  DEX             ; .. um Zahl um 1 zu verringern
  CPX #$ff        ; Haben wir den linken Rand erreicht?
  BEQ left_border
  STX $01         ; Speichere neue Position
  JMP main
left_border:
  LDA #$0         ; Setze Position auf maximalen linken Rand
  STA $01
  JMP main
```

```
; Routine: Spieler nach rechts bewegen

right:
  JSR clear_input
  JSR clear       ; Lösche letzte Position auf Bildschirm
  LDA $01         ; Lade aktuelle Position in A und
  TAX             ; .. kopiere nach X
  INX             ; .. um Zahl um 1 zu erhöhen
  CPX #$0f        ; haben wir den rechten Rand erreicht?
  BEQ right_border
  STX $01         ; Speichere neue Position
  JMP main
right_border:
  LDA #$0e        ; Setze Position auf maximalen rechten Rand
  STA $01
  JMP main
```

Das Programm stellt einen Pixel auf dem Display dar, welcher dann durch die Tasten A und D um einen Schritt nach links bzw. rechts bewegt werden kann. Außerdem wird überprüft, ob ein Rand erreicht wurde, in diesem Fall wurde die Bewegung nicht ausgeführt.

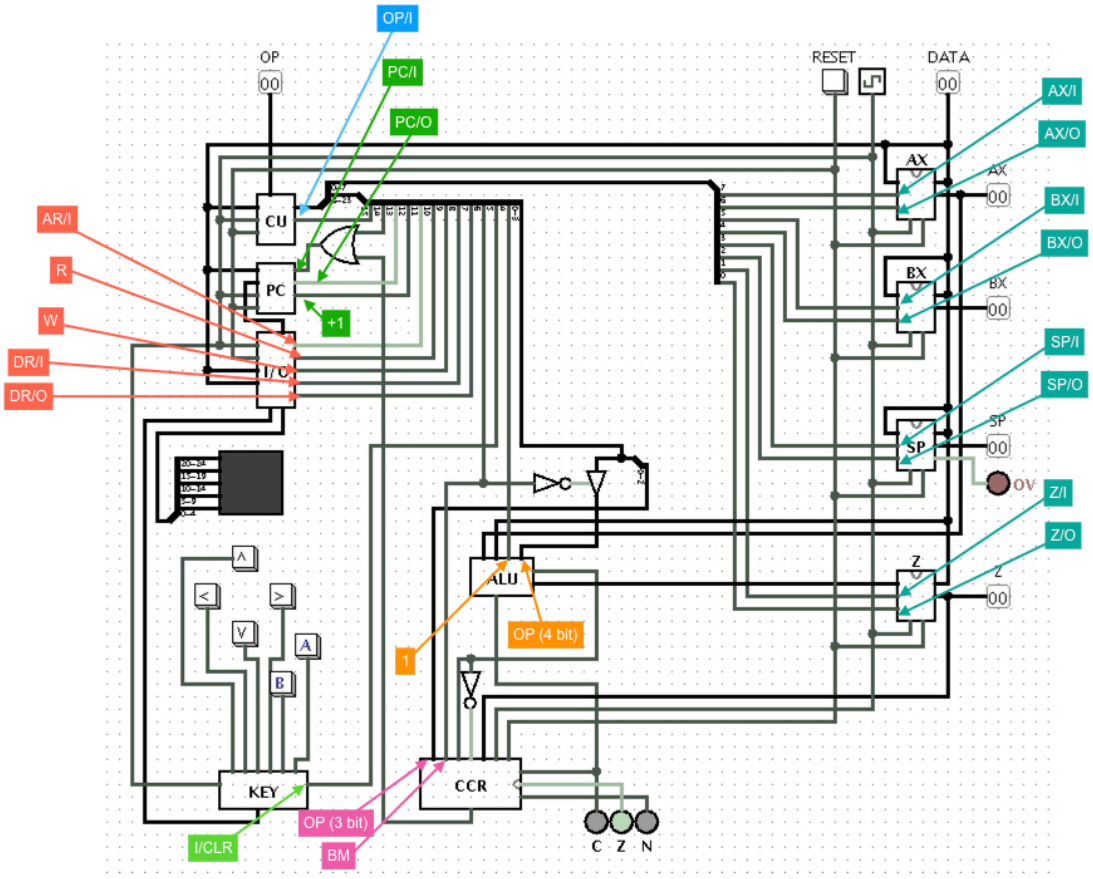
Die Speicheradressen waren zunächst so angelegt, wie sie beim C64 vorhanden sind, Variablen wurden in die Zeropage $00-$ff gelegt, Bildschirmausgaben starteten ab Adresse $0200. Das musste natürlich in einem späteren Schritt für die eigene Architektur angepasst werden, was ich aber erst gemacht habe, sobald ich mehr über diese wusste. Im Anhang C kann die Version für die eigene Rechnerarchitektur eingesehen und verglichen werden.

Mit diesem Programm begann ich den Entwurf von vorne und setze erstmal die Arbeitsregister und einfachen Schreibe- und Lesevorgänge um, später folgten dann weitere Adressierungsarten (indexiert, wie auch beim Programm benötigt), ALU Operationen (Inkrement, Dekrement) und dann zuletzt wieder die Sprünge, welche dann aber viel einfacher zu realisieren waren.

Grundlegender Aufbau

Die finale Version des Computers besteht aus ingesamt 11 Elementen, welche fast alle (bis auf den LED Controller) auf den ersten Blick sichtbar sind: CU (control unit), PC (program counter), I/O (input / output controller mit RAM, Zufallszahlen-Generator und LED controller), KEY (Tastatur-Buffer und controller), ALU (arithmetic-logical unit), CCR (condition code register), AX (Arbeitsregister, Akkumulator), BX (Arbeitsregister), SP (stack pointer Register), Z (ALU Ergebnisregister).

Es gibt einen 8 bit Datenbus, welcher an fast allen Elementen angeschlossen liegt (bis auf CCR und KEY) und durch Umstellen der Lese- und Schreibeflags der jeweiligen Elemente gelesen oder beschrieben werden kann. Diese Flags werden als Steuerbefehle unter anderem von dem 24 bit Steuerbus gesteuert.



Übersicht über den finalen Computer und der Komponenten mit Markierungen der einzelnen Steuersignale

Der I/O controller ist nicht günstig benannt, behandelt aber alle Komponenten, die mit einer Speicheradresse ausgelesen oder beschrieben werden können, wie den Arbeitsspeicher, den Videospeicher, den Zufallszahlgenerator (only read) oder auch den Tastaturbuffer (only read).

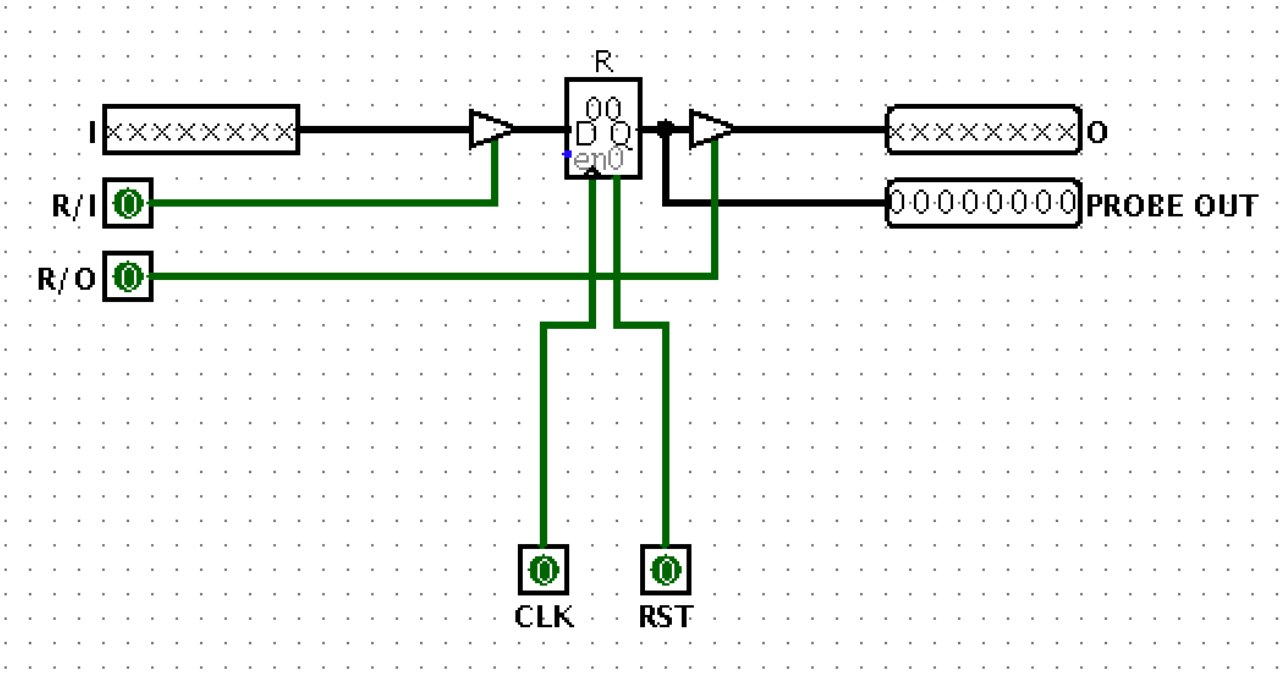
Zusätzlich sieht man eine monochrome LED Pixel-Matrix von 5x5 Pixeln, also ingesamt 25 Pixeln und 6 Taster, mit einfachen Pfeiltasten und A und B Funktionstasten - angelehnt an den »Gameboy« in seiner ersten Fassung von 1989.

Zur Überprüfung besitzt die Logisim-Schaltung einen Einblick (via sogenannte »probes«) in den Steuerbus (oben links, »OP«), dem aktuellen Inhalt des Datenbus (oben rechts, »DATA«) und in die jeweiligen Register (AX, BX, SP und Z). Drei LEDs zeigen den Status der letzten ALU Operation an (Carry, Zero, Negative flag) und eine LED ob ein Stack Overflow besteht (OV LED, rechts).

Eine RESET Taste oben rechts führt dazu, dass der Computerzustand zurückgesetzt wird, ohne den RAM zu löschen (was die Reset-Funktion von Logisim macht). So ist es möglich, den Computer neu zu starten, ohne das Programm wieder neu laden zu müssen. Außerdem sorgt die RESET Taste dafür, dass der Stack Pointer an die richtige Anfangsadresse $e0 gesetzt wird

(Logisim unterstützt hier leider keine eigenen default Werte für Register). Vor jeder korrekten Ausführung muss die RESET Taste also gedrückt werden.

AX, BX, Z, SP: Arbeits-, Ergebnisregister und Stack Pointer



Register-Schaltung

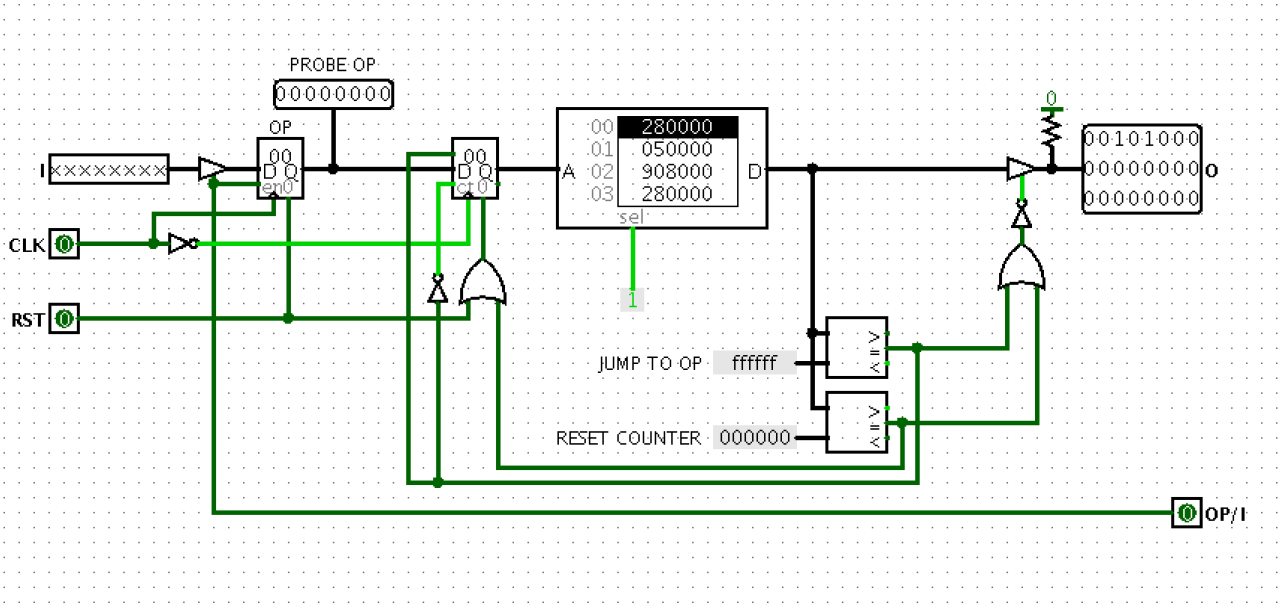
Die Arbeitsregister AX und BX, aber auch das Ergebnisregister Z und der Stack Pointer SP (in leichter Variation) basieren auf dieselbe Schaltung: Ein einfacher 8bit Speicher erwartet auf der Leitung I einen Wert, welcher aber durch den Buffer R/I erst freigeschaltet werden kann. Wenn ein Wert auf I anliegt und R/I aktiv ist, wird beim nächsten Takt der Wert in das Register geschrieben. Dasselbe gilt für den Ausgang: Ist der Buffer R/O gesetzt, wird der aktuelle Registerwert beim nächsten Takt auf die Leitung O gelegt. RST dient dem Zurücksetzen des Registers, PROBE OUT ist ein Testausgang, um den aktuellen Wert an einer anderen Stelle anzuzeigen.

Eine jede einzelne micro instruction im Steuerprogramm der CU kontrolliert unter anderem diese Buffer, erlaubt also dem jeweiligen Register einen Wert auf der Leitung anzunehmen oder abzulegen.

CU: Control Unit

Wie schon oben erwähnt, werden die einzelnen Buffer und Komponenten des Computers mithilfe eines 24 bit control bus gesteuert. In der Übersicht-Abbildung am Anfang des Kapitels sind alle Steuer-bits gekennzeichnet (OP besteht aus 4 bits).

In der CU befindet sich die ROM mit dem microcode, die mit 8 bit adressiert werden kann, bis zu 256 micro instructions enthält und jeweils 24 bit lang sind. In der finalen Schaltung sind insgesamt 222 Adressen mit micro instructions belegt.



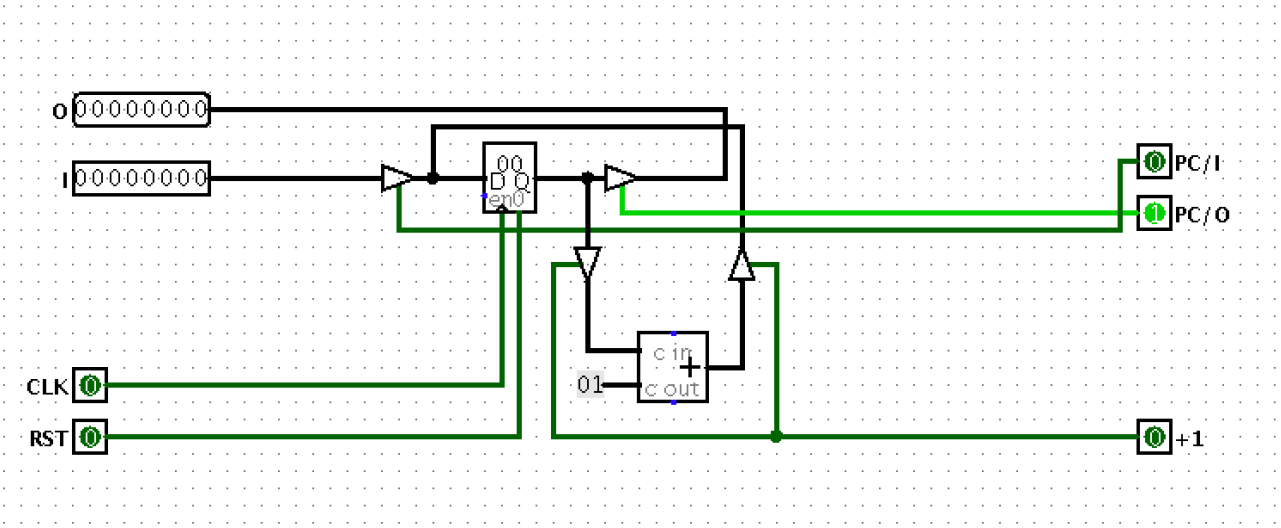
CU / control unit Schaltung

Ein Counter erhöht bei jedem Takt die ROM Adresse um 1, sodass die nächste micro instruction ausgelesen und auf den control bus gelegt werden kann. Falls eine micro instruction nur aus Nullen besteht (0000 0000 0000 0000 0000 0000) wird der counter auf $00 zurückgesetzt.

Wenn eine micro instruction nur aus Einsen besteht (1111 1111 1111 1111 1111 1111) wird der counter auf den Wert des Registers OP gesetzt und damit auch die nächste ROM Adresse definiert. So sind Sprünge innerhalb des microcodes möglich.

PC: Program Counter

Der Programmzähler oder Program Counter (PC) dient zur Adressierung des Arbeitsspeicherinhalts für den nächsten Programmbefehl und ggf. operand. Er besteht wie die Arbeitsregister und der Stack Pointer aus einem einfachen 8 bit Register, welches mithilfe des I/O Datenbus und den jeweiligen PC/I und PC/O Flags beschrieben oder ausgelesen werden kann. Zusätzlich besitzt der PC eine +1 Flag, welche den aktuellen Registerinhalt mit einer konstanten 1 addiert und wieder zurückschreibt, der Registerinhalt wird so um 1 erhöht.



PC / program counter Schaltung mit aktivem PC/O Buffer. Der Register-Inhalt $00 wird so auf den Datenbus gelegt.

I/O: Arbeitsspeicher, Zufallszahlen-Generator

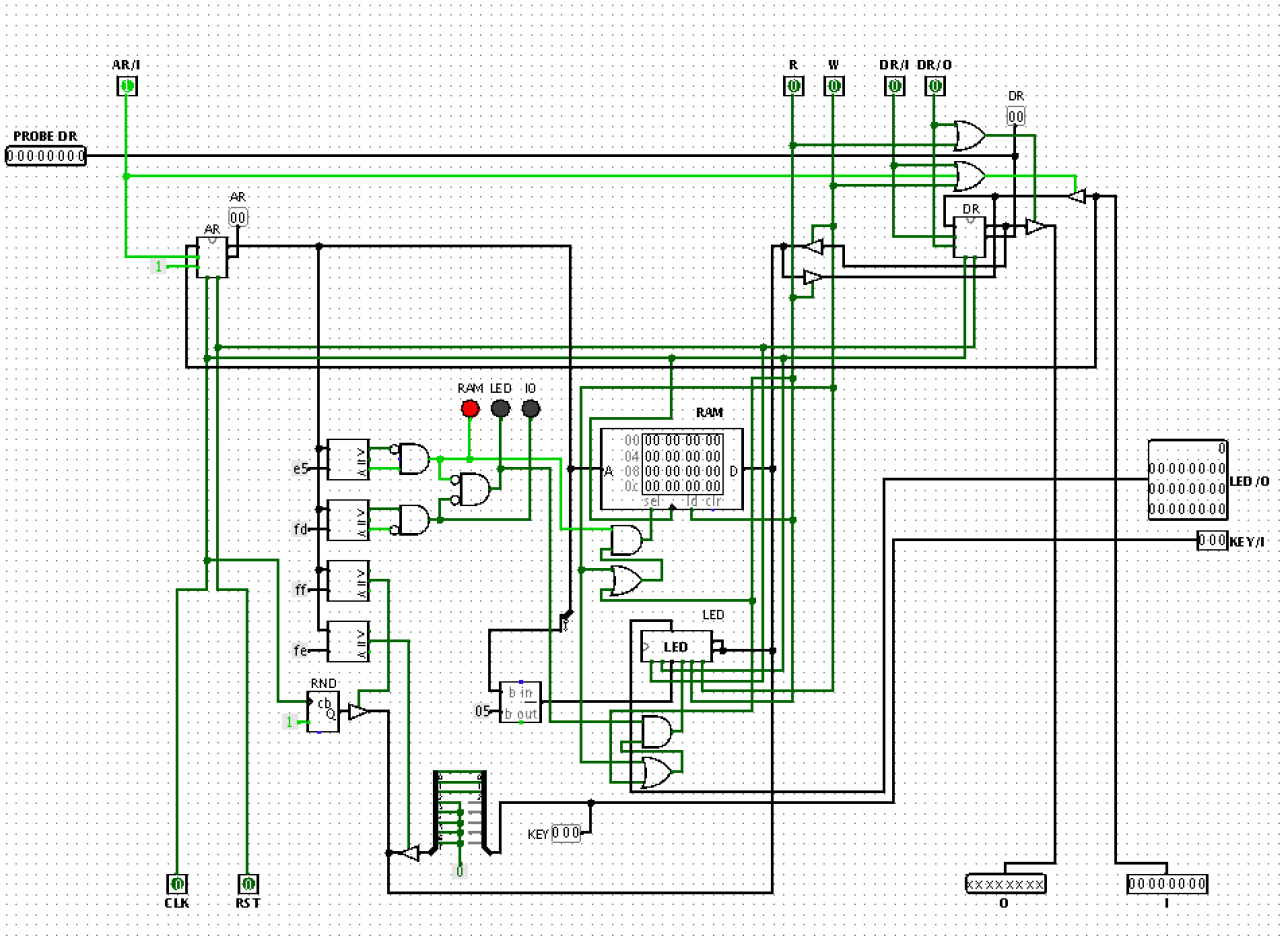
Das I/O Element umfasst mehrere Subelemente, welche jeweils eine adressierbare, auslesbare oder beschreibbare Speichereigenschaft haben. Der Name ist nicht glücklich gewählt, versucht aber einigermaßen gut diese Eigenschaften zusammenzufassen. Innerhalb des I/O ist auch der LED controller angesiedelt, welcher hier separat behandelt wird.

Kernstück ist hier der Arbeitsspeicher, ein RAM Element, welches mithilfe des AR Register (address register) adressiert werden kann. Mithilfe der R und W flags können so an der adressierten Speicherstelle Werte aus DR (data register) in den RAM geschrieben oder nach DR kopiert und somit der RAM ausgelesen werden.

Der Arbeitsspeicher kann mit 8 bit adressiert werden und enthält jeweils 8 bit Daten, ist also insgesamt 256 byte groß, was die Größe unserer Programme sehr klein macht und die Arbeit an umfangreicheren Programmen erschwert. Zusätzlich wird der frei nutzbare Speicher weiter verkleinert, da wir einen Teil des Adressbereichs für spezielle Funktionen bereitstellen müssen. So kann auch der Bildschirmspeicher im LED controller hier angesprochen, der Tastaturbuffer oder der Zufallsgenerator ausgelesen werden.

Der Adressbereich ist wie folgt unterteilt:

```
- $00 - $DF (000 - 223) free usable memory (224 Byte)
- $E0 - $E4 (224 - 228) stack (5 Byte)
- $E5 - $FD (229 - 253) 5x5 Pixel Display (25 Byte)
- $FE (254) currently pressed key (1 Byte) [read-only]
- $FF (255) random number (1 Byte) [read-only]
```



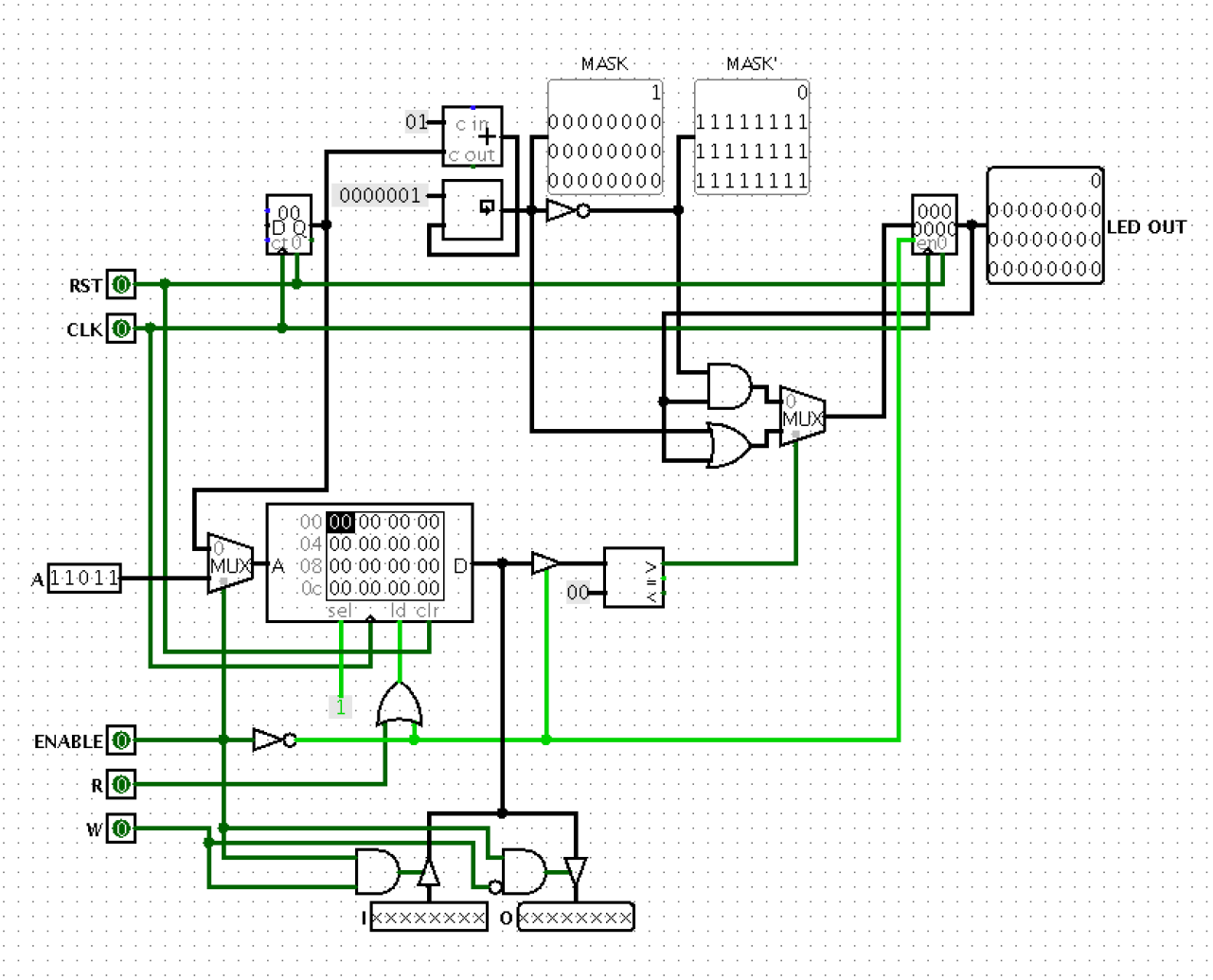
IO Schaltung mit Arbeitsspeicher (RAM mittig), Zufallszahlengenerator (RND unten links), Tastaturbuffer-Inhalt (KEY unten) und LED controller (LED mittig unten)

Für das eigentliche Programm stehen nur 224 byte zur Verfügung, 5 byte im Arbeitsspeicher reservieren wir uns für den Stack, wir können also insgesamt 5 x 8bit Werte ablegen oder fünf Level tief in Unterprogramme springen. Das ist nicht sehr viel, für ein Programm der maximalen Länge von 224 byte aber doch ausreichend.

Die Adresse $e5 bis $fd wird für den Bildschirmspeicher verwendet, welcher mit 5x5 Pixeln aus eben 25 Bytes besteht. $fe und $ff sind read-only Sonderadressen, um die zuletzt gedrückte Taste oder einen Zufallswert auszulesen. Hier stellt uns Logisim einfach einen Baustein zur Verfügung, welcher zu jedem Takt eine neue Zahl auswirft.

Es fällt auf, dass keine Zero-page benötigt wird. Bei einem nur 8bit großen Adressraum sind alle Adressen gleich schnell erreichbar, es ist aber zu beachten, dass beim Programmieren das Programm meistens bei $00 beginnt und die Variablen weiter oben angesiedelt werden sollten (und erst recht nicht ausversehen den Programmcode überschreiben sollten). Insofern wird die Programmgröße je nach Variablenanzahl noch weiter eingeschränkt, was auch später bei der Entwicklung des snake Spiels deutlich wird.

LED: Pixel controller



LED Schaltung zur Steuerung von 25 virtuellen LED Pixeln, ein »echter« controller wird sicherlich anders aussehen

Das Element LED besteht aus einem einfachen RAM Speicherelement (25 byte groß), welches mit den Werten $00 und $ff beschrieben werden kann. Zwei flags R und W dienen zur Aktivierung drei unterschiedlicher Operationsmodi. Sind beide flags nicht gesetzt (R = 0 und W = 0), durchläuft ein counter alle Speicheradressen des Bildschirmspeichers bis er das Ende erreicht und wieder von vorne beginnt. Durch eine Bitmaske und ihrer negation kann mit einer einfachen logischen AND und OR Operation der aktuell ausgelesener Wert umformatiert ($ff = 1 und $00 = 0) an die richtige Stelle ins LED OUT Register geschrieben werden, welches mit 1 x 25 bit den Status aller 25 LEDs beinhaltet. Die Routine konvertiert und kopiert also Schritt für Schritt die Werte aus dem Bildschirmspeicher an das LED OUT Register, welches für die Darstellung der LED Matrix Daten in diesem Format benötigt. Nach 25 Takten ist also der Bildschirmspeicher einmal ausgelesen und an die LEDs übertragen.

Die Maskierung wird mit dem folgenden Trick bewerkstelligt: Der aktuelle LED OUT Registerinhalt wird, wenn das Pixel gesetzt ist, mit der Bitmaske »verodert« (OR) und dann zurückgeschrieben.

```
0000 0000 0000 0000 0000 0010 0 (Maske)
0000 0000 1111 0000 1100 0000 0 (LED OUT Register)
============================== OR
0000 0000 1111 0000 1100 0010 0 (LED OUT Register')
```

Um ein einzelnes Pixel zu deaktivieren wenden wir dasselbe Verfahren an, jedoch mit einer negierten Maske und einem AND Gatter.

```
0000 0000 0000 0000 0000 0010 0 (Maske)
1111 1111 1111 1111 1111 1101 1 (Maske’ negiert)
0000 0000 1111 0000 1100 0010 0 (LED OUT Register')
============================== AND
0000 0000 1111 0000 1100 0000 0 (LED OUT Register’')
```

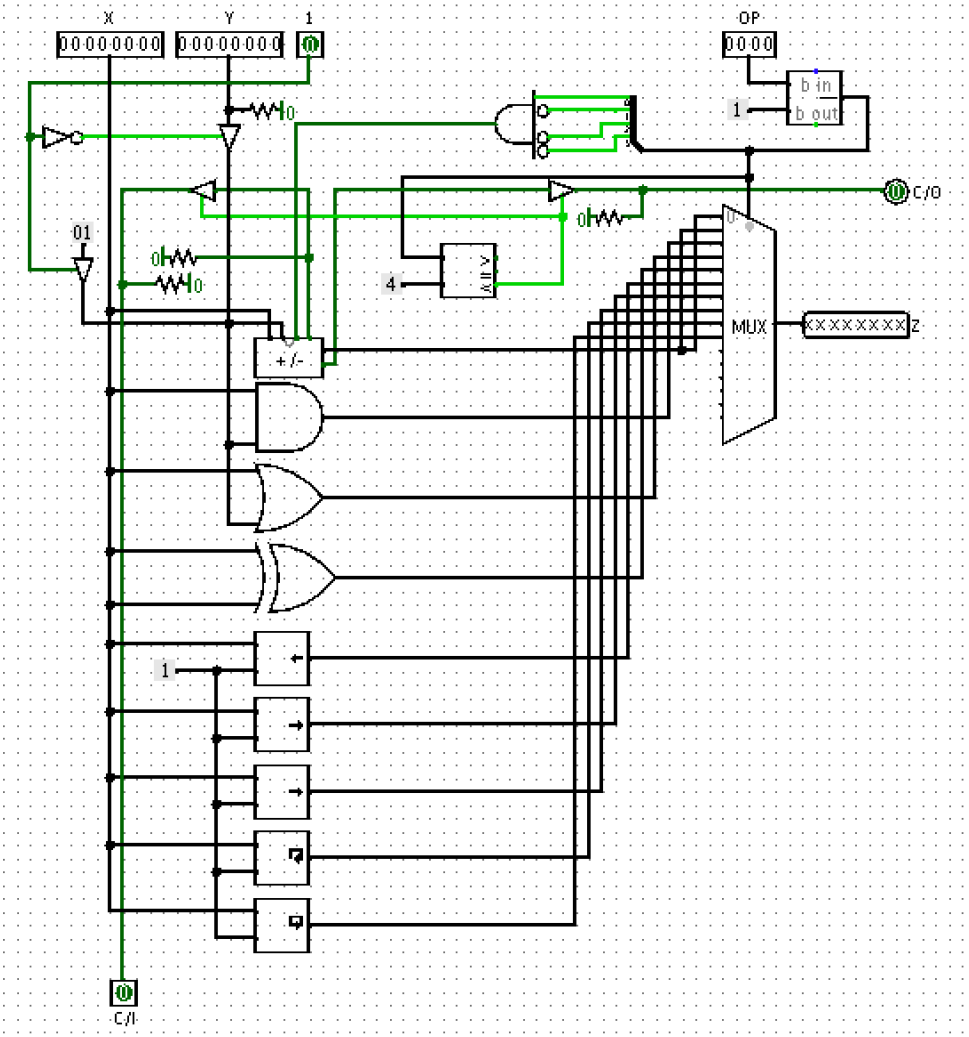
ALU: Arithmetisch Logische Einheit

Die ALU erwartet zwei Variablen X und Y, führt die durch einen opcode gewählte mathematische oder logische Operation mit diesen Variablen aus und gibt das Ergebnis an den Ausgang Z weiter. Durch die spezielle »1« flag kann der Eingang Y auf die Konstante 1 gelegt werden. Ein Carry Out flag gibt uns an, ob ein carry oder borrow bei der Addition bzw. Subtraktion aufgetreten ist. Diese flag können wir dann im CCR festhalten und ggf. weiterverwerten, um beispielsweise Zahlen auszurechnen, welche größer als 8 bit sind.

Die folgenden opcodes werden von der ALU akzeptiert:

- **0001** Z = X + Y
- **0010** Z = X - Y
- **0011** Z = X AND Y
- **0100** Z = X OR Y
- **0101** Z = X XOR Y
- **0110** Z = Y logically shifted left
- **0111** Z = Y logically shifted right
- **1000** Z = Y arithmetically shifted left
- **1001** Z = Y rotated left
- **1010** Z = Y rotated right

Opcodes beginnen erst ab dem Wert 1 (0001), um zu vermeiden, dass bei Nicht-ALU Operationen (Opcode = 0000) trotzdem indirekt Werte ausgerechnet und fälschlicherweise die Carry-Flag gesetzt wird.



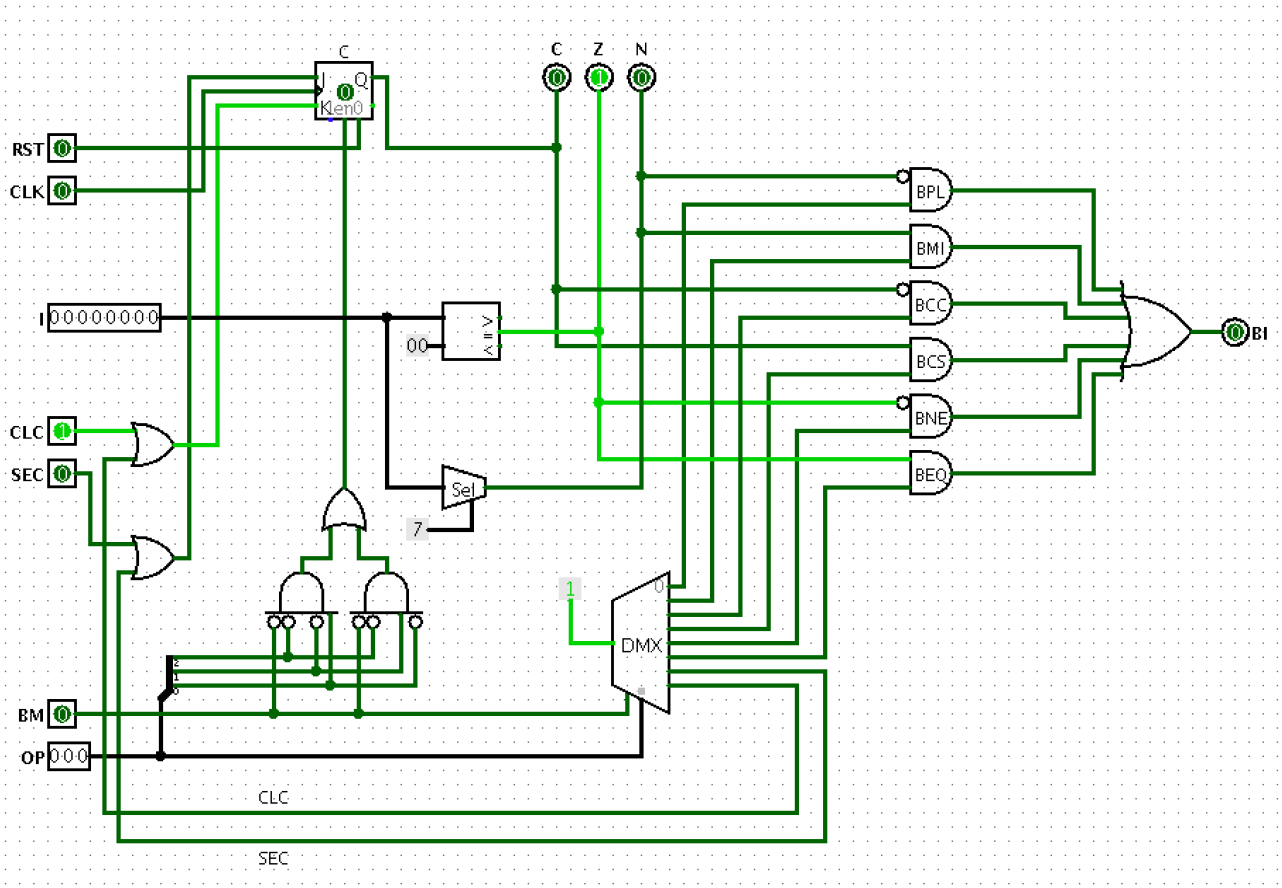
ALU Schaltung

CCR: Condition Code Register

Das CCR verwaltet die status flags des Computers und manipuliert den PC, falls eine branch instruction vorliegt, welche ausgeführt werden soll. Mithilfe der opcodes und des speziellen branch mode flags (BM), kann ein Sprung angefordert werden, welcher je nach positiven oder negativen test des jeweiligen Status flag den PC manipuliert oder unberührt lässt.

Insgesamt sind drei verschiedene flags vorhanden:

- C = Carry (Übertrag bei Addition oder Subtraktion)
- Z = Zero (Ergebnis in Z entspricht dem Wert $00)
- N = Negative (Das letzte bit im Ergebnis ist gesetzt)



CCR / condition code register Schaltung, mit Statusflags C, Z und N

Die folgenden branch instructions können mithilfe der jeweiligen opcodes und BM = 1 ausgeführt werden:

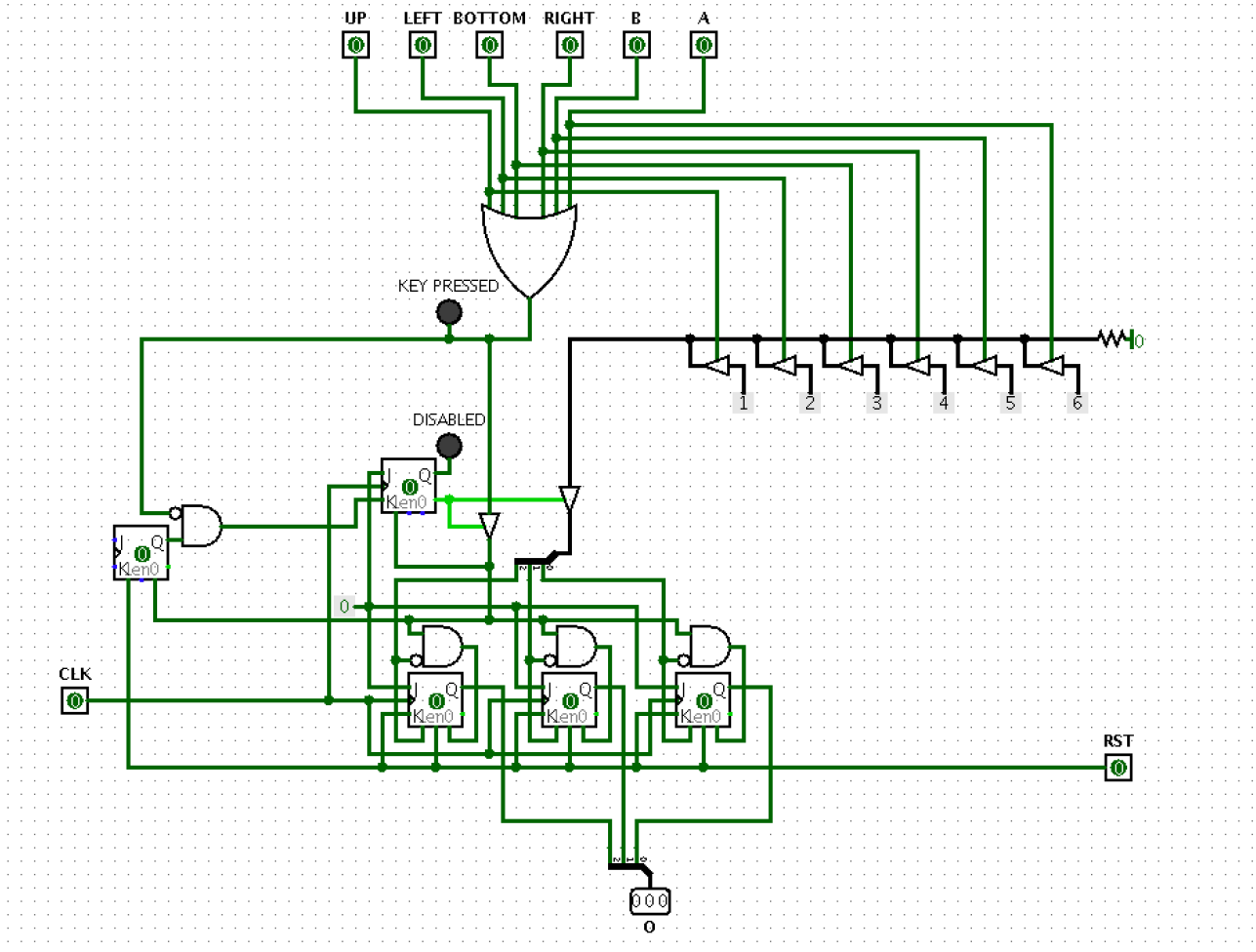
- **000** BPL Branch on plus (N)
- **001** BMI Branch on minus (N)
- **010** BCC Branch on carry clear (C)
- **011** BCS Branch on carry set (C)
- **100** BNE Branch on not equal (Z)
- **101** BEQ Branch on equal (Z)

Wobei die letzten beiden opcodes keine branch instructions sind, sondern die carry flag entweder manuell setzen oder löschen können:

- **110** SEC Set carry flag
- **111** CLC Clear carry flag

Eine Besonderheit des Aufbaus des CCR ist die etwas aufwändigere logische Schaltung, welche das J/K FlipFlop für die carry flag aktiviert oder deaktiviert und so auch beim sich ändernden Takt unberührt lassen kann. Dies ist notwendig, um den Zustand der carry flag zu behalten, auch wenn das Programm längst fortgeschritten ist. So kann das carry flag nur bei den ALU Operationen **0001** und **0010** verändert werden, also da wo die ALU eine Addition oder Subtraktion ausführt.

KEY: Tastatur-Controller und Buffer



KEY / Tastatur controller Schaltung

Der Tastatur-Controller KEY kodiert einfache Tastendrucks in einen 3 bit code und speichert diesen in drei J/K FlipFlops, dem Tastaturbuffer.

Durch einen besonderen Mechanismus kann dieser erst wieder beschrieben werden, wenn 1. kein Tastendruck aktuell mehr vorliegt und 2. das RST flag (»reset«) aktiv ist. Dies ist eine Art debouncing Mechanismus, welcher dafür sorgt, dass Tastatureingaben immer nacheinander erfolgen müssen. Das RST flag kann durch den speziellen opcode **cib** (»clear input buffer«) aktiviert werden.

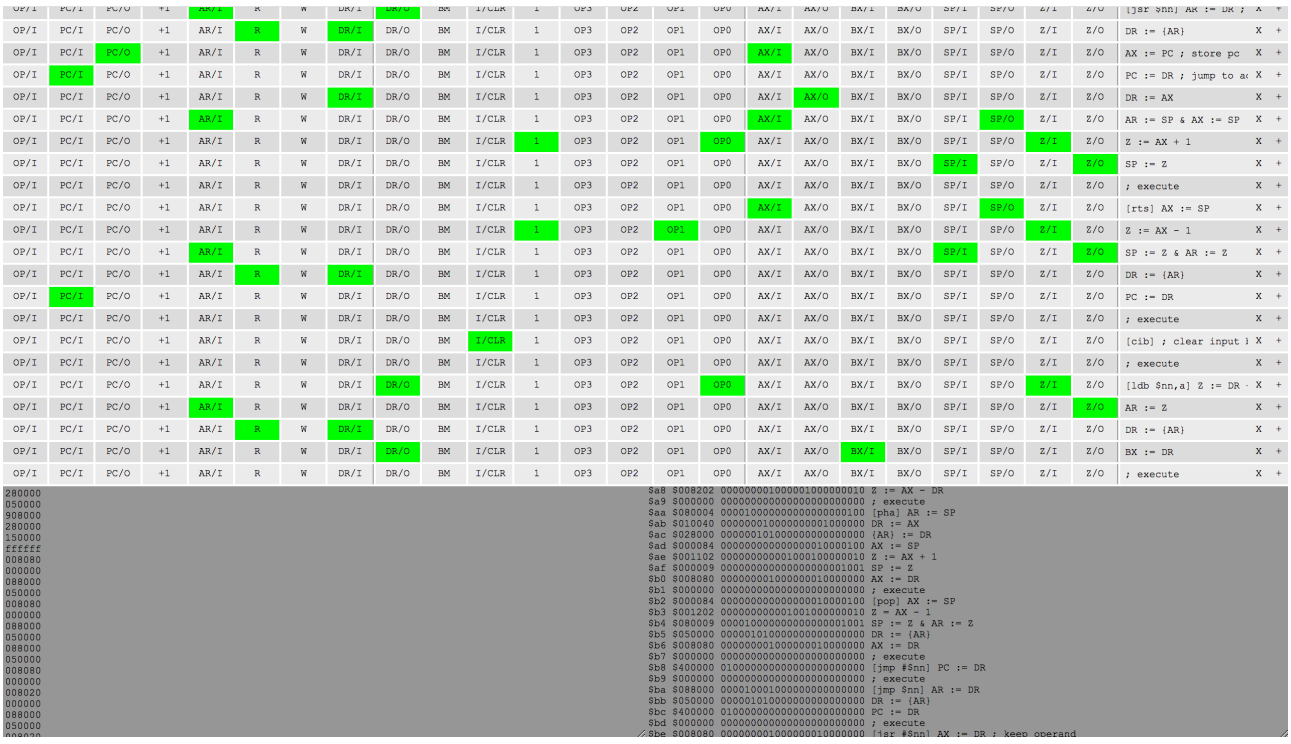
Microbefehle und Opcodes

Microcode

Der microcode in der ROM des CU beinhaltet einen vereinfachten Von-Neumann-Zyklus, welcher aus 6 micro instructions von Adresse $00 bis $05 besteht:

```
$00 $280000 001010000000000000000000 AR := PC ; fetch addr
$01 $050000 000001010000000000000000 DR := {AR}
$02 $908000 100100001000000000000000 OP := DR & PC := PC+1
$03 $280000 001010000000000000000000 AR := PC ; fetch op
$04 $150000 000101010000000000000000 DR := {AR} & PC := PC+1
$05 $ffffff 111111111111111111111111 ; decode
```

Der aktuelle Wert im PC wird als nächste Adresse an das AR Register für den Arbeitspeicher (RAM) geführt ($00) und der ausgelesene Wert aus dem Speicher ins Daten Register (DR) gelegt ($01). Dieser Wert beschreibt die Adresse unserer erwünschten micro instruction und wird deshalb in das OP register im CU gelegt, der program counter um 1 erhöht ($02). Nun wird dasselbe



Micro instruction editor, um Steuer-bits einfach zu setzen und micro instructions zu entwerfen. Microcodes können einfach für Logisim exportiert oder importiert werden.

Verfahren erneut ausgeführt, der neue PC Wert als RAM Adresse verwendet ($03) und der Wert ausgelesen und in DR gelegt ($04). Dieser Wert steht für den operand, welcher als Variable zur Ausführung der micro instruction dient. Der program counter wird nochmals um 1 erhöht, um den nächsten Zyklus vorzubereiten (auch $04).

In $05 finden wir den Wert $ffffff, welcher nur aus Einsen besteht. Dieser Wert sorgt dafür, dass im CU zu der micro instruction gesprungen wird, die wir uns im Zyklus aus dem Arbeitsspeicher geholt und in OP zwischengespeichert haben. Diese micro instruction muss mit dem Wert $000000 enden, damit wieder zurück an Stelle $00 im microcode gesprungen werden kann. So können wir nun verschiedene micro instructions abarbeiten und den Computer damit steuern.

Alle micro instructions wurden mithilfe eines selbstgeschriebenen tools, dem »micro instruction editor« entwickelt: eine einfache HTML-Seite mit JavaScript und graphischer Oberfläche, welche es ermöglicht durch einfache clicks die jeweiligen Steuer-bits zu setzen, um micro instructions zu entwerfen. Zuletzt stellt das Programm das Ergebnis in Hexadezimal-Zahlen dar, sodass es einfach in Logisim in die ROM kopiert geladen kann.

Die 24 zu steuernden bits lauten (mit * markierte bits sind besonderer Art):

- 23 OP/I (CU)
- 22 PC/I (PC)
- 21 PC/O (PC)
- 20 +1   (PC)
- 19 AR/I (RAM)
- 18 R   (RAM)
- 17 W   (RAM)
- 16 DR/I (RAM)

- 15 DR/O (RAM)
- 14 BM   (Branch Mode Flag *)
- 13 I/CLR (Clear I/O buffer *)
- 12 1   (ALU)
- 11 OP3   (ALU)
- 10 OP2   (ALU / CCR when BM set)
- 09 OP1   (ALU / CCR when BM set)
- 08 OP0   (ALU / CCR when BM set)

- 07 AX/I (AX)
- 06 AX/O (AX)
- 05 BX/I (BX)
- 04 BX/O (BX)
- 03 SP/I (SP)
- 02 SP/O (SP)
- 01 Z/I   (Z)
- 00 Z/O   (Z)

Sie kontrollieren meistens, ob ein Speicher gelesen oder geschrieben wird (I steht für Input, also Schreiben, O für Output, also Lesen). Bit 20 erhöht den PC um 1 beim nächsten Takt (der Umweg über die ALU war zu kompliziert, insofern wurde ein eigenes micro instruction bit dafür eingeführt), bit 8-11 beschreiben den 4 bit opcode der ALU. Um Platz zu sparen, wird dieser opcode aber auch verwendet, um das CCR anzusteuern, welches mit einem 3 bit opcode arbeitet, um branch instructions auszuführen oder die carry flag zu setzen oder zu löschen. Um den opcode nicht für die ALU sondern für das CCR zu verwenden, muss das bit 14 gesetzt sein (BM steht für branch mode). Wenn das bit 12 gesetzt wird, wird auf die ALU eine konstante 1 gelegt, um Inkrement und Dekrement Operationen ausführen zu können. Bit 13 stellt eine besondere Eigenart des Computers dar und wurde eingeführt, um den Tastatur-Buffer zu leeren. Dies ist sicherlich nicht die schönste Form dieses Problem zu lösen (man könnte dies auch durch Schreiben einer $00 an die richtige Adresse ermöglichen), stellte sich aber doch als einfachste Lösung heraus.

Der microcode besteht aus 222 micro instructions (von $00 bis $dd) mit jeweils 24 bit, ist also insgesamt 666 byte groß. Der microcode kann kopiert und in den Editor importiert werden (als einfache Textdatei oder auch im JSON Format), um ihn auch dort darzustellen oder wieder als JSON Datei zu exportieren. Siehe Anhang A für den gesamten microcode.

Opcodes und Adressierungsarten

Wie schon im microcode zu lesen, bietet dieser insgesamt 53 opcodes, bzw. 33 verschiedene mit unterschiedlichen Adressierungsarten. Jeder Opcode ist 16 bit lang, wobei die ersten 8 bit die micro instruction Adresse festhalten und die darauffolgenden 8 bit den operand.

```
| --------- | --------- |
| 0000 0000 | Adresse   |
| 0000 0000 | Operand   |
| --------- | --------- |
```

Leider werden auch bei opcodes ohne operand 16 bit verwendet, auch wenn der operand nur aus sinnlosen Werten besteht. Dies verschwendet zwar wertvollen Speicherplatz (der gerade bei der Programmierung von Snake gefehlt hat), vereinfachte aber den Von-Neumann-Zyklus und die Computer Architektur.

Vor allem für Lese- und Speicherbefehle sind verschiedene Adressierungsmodi implementiert, insgesamt sind folgende vorhanden: Implizit, Absolut $nn, Unmittelbar #$nn, Indexiert $nn,a, Indexiert + indirekt ($nn,a), Indirekt ($nn), Indirekt + indexiert ($nn),a.

Alle Opcodes sind im Anhang B mit den jeweiligen Adressierungsarten aufgelistet.

Beispielprogramm: Snake

Neben dem Testprogramm, welches eingangs in 6502 Assembler geschrieben und für die eigene Architektur umgeschrieben wurde (siehe Anhang C oder Datei *asm/test.asm*) habe ich einen kleinen Snake clone für den Computer geschrieben.

Die Entwicklung wurde dort kompliziert, wo der Speicher nicht mehr ausreichte. So mussten manche Funktionen vereinfacht oder gänzlich ausgelassen werden. Diese Snake Version hat dadurch ein paar Eigenarten, welche auf schöner Weise zeigen, wo die Limitierungen der gewählten Architektur liegen.

Das Programm ist insgesamt 208 Byte lang und damit exakt an der maximalen Obergrenze, wenn wir noch genug Speicher für Variablen zur Verfügung haben möchten.

Assembler

Zur Entwicklung in Assembler wurde parallel zum Computer eine Software in C entwickelt, welche den Assemblercode in Maschinensprache übersetzt. Dies geschieht automatisch in das von Logisim erwünschte Format, sodass der Code einfach mit Linksclick auf das RAM Element und »Load image« ausgewählt und importiert werden kann. Nach click auf »RESET« ist das Programm nun startklar.

Der Source-Code des Assemblers kann im Anhang D oder in der Datei *tools/assembler/assembler.c* eingesehen werden. Mithilfe des Programmes können so *.asm* Dateien in die jeweiligen executables konvertiert werden:

```
> ./assembler program.asm program.out
```

Der konvertierte code sieht bei snake beispielsweise so aus (im header eine Versions und Format-Markierung für Logisim):

```
v2.0 raw
06 eb 2c d0 06 ec 2c d1 06 ed 2c d2 06 02 2c de
be b2 be c8 08 fe a4 00 9e 14 be 34 be 9a be 26
be bc be c8 b8 14 08 de 6a 00 d9 d0 36 d1 6a 00
94 2a d1 00 08 fe a4 01 9e 48 a4 02 9e 5a a4 03
9e 6c a4 04 9e 88 d1 00 08 d0 5f 05 2c d0 a4 e5
98 54 d1 00 57 19 2c d0 d1 00 08 d0 6a 00 2c d0
a4 e4 9e 66 d1 00 06 e9 2c d0 d1 00 08 d0 57 05
2c d0 a4 fe 9a 7c a4 04 98 82 d1 00 5f 19 2c d0
d1 00 57 e7 2c d0 d1 00 08 d0 67 00 2c d0 a4 fe
9e 94 d1 00 06 f9 2c d0 d1 00 08 df a6 d0 9c aa
08 de 67 00 2c de be b2 d1 00 0c d0 a4 ff 9e ce
d1 00 08 ff 6d 18 57 e5 2c df d1 00 08 de 12 00
4c d0 12 ff 3f d0 d1 00 12 ff 3f df d1 00 b8 ce
```

Snake

Ziel war es, so nahe wie möglich an das Snake Spiel heranzukommen. Im Anhang E kann der kommentierte Assembler-Code eingesehen und so die Implementierung nachvollzogen werden.

Anleitung zur Ausführung

1. Logisim Projekt *circuits/v3/CPU.circ* öffnen
2. Sicherstellen, dass unter *Simulate* > *Tick Frequency* der höchste Wert ausgewählt (4,1 KHz) und »Simulation enabled« aktiviert ist
3. Mit Doppelklick auf »I/O« die Schaltung öffnen
4. Linksklick auf das RAM Element, »Load Image« auswählen
5. Die Datei *asm/snake.out* auswählen
6. Zurück zur Hauptansicht »CPU«
7. »RESET« Taste oben rechts klicken, um den Stack Pointer auf den Standardwert zu setzen
8. *Simulate* > *Ticks Enabled* auswählen, um Simulation zu beginnen. Das Spiel beginnt erst mit dem ersten Tastendruck auf eine der Pfeiltasten

Fazit

Die Schlange im Computerspiel Snake bewegt sich immer langsamer je länger sie wird. Das ist ein schönes Beispiel dafür, wie der Computer arbeitet. Selbst in seiner 4,1 kHz Taktung durch Logisim können wir die »Takte« fühlen, die Schleifen im Programm werden immer länger, je mehr Segmente der Schlange auf dem Bildschirm bewegt werden müssen. Bei modernen Systemen können wir diese Unterschiede natürlich nicht mehr wahrnehmen, mit dem selbstgebauten Computer aber doch.

Diese Erlebnisse und noch viele andere waren für mich eine gute Erfahrung, einem grundsätzlichen »Verständnis« oder auch »Gefühl« für Computer nahezukommen. Natürlich wäre es nun reizvoll diese Architektur auch in Hardware umzusetzen, was vielleicht ein neues Projekt werden kann.

Quellenverzeichnis

Wie Eingangs erwähnt, habe ich versucht auf Quellen soweit wie es geht zu verzichten. Manchmal halfen aber doch die folgenden Seiten, YouTube-Channels oder Bücher aus:

- https://eater.net/ »Built a programmable 8-bit computer from scratch on breadboards using only simple logic gates.« Video tutorials von Ben Eater
- http://rtro.de/ Online 6502 Assembler und Emulator
- http://www.6502.org/tutorials/6502opcodes.html 6502 Opcode Tabelle
- Rodnay Zaks »Programmierung des 6502« Übersetztes Buch von 1981, 2. Auflage
- http://www.cburch.com/logisim/ Logisim Software für logische Schaltungen

Anhangverzeichnis

A - Microcode

```
$00 $280000 001010000000000000000000 AR := PC ; fetch addr
$01 $050000 000001010000000000000000 DR := {AR}
$02 $908000 100100001000000000000000 OP := DR & PC := PC+1
$03 $280000 001010000000000000000000 AR := PC ; fetch op
$04 $150000 000101010000000000000000 DR := {AR} & PC := PC+1
$05 $ffffff 111111111111111111111111 ; decode
$06 $008080 000000001000000010000000 [lda #$nn] AX := DR
$07 $000000 000000000000000000000000 ; execute
$08 $088000 000010001000000000000000 [lda $nn] AR := DR
$09 $050000 000001010000000000000000 DR := {AR}
$0a $008080 000000001000000010000000 AX := DR
$0b $000000 000000000000000000000000 ; execute
$0c $088000 000010001000000000000000 [lda ($nn)] AR := DR
$0d $050000 000001010000000000000000 DR := {AR}
$0e $088000 000010001000000000000000 AR := DR
$0f $050000 000001010000000000000000 DR := {AR}
$10 $008080 000000001000000010000000 AX := DR
$11 $000000 000000000000000000000000 ; execute
$12 $008020 000000001000000000100000 [ldb #$nn] BX := DR
$13 $000000 000000000000000000000000 ; execute
$14 $088000 000010001000000000000000 [ldb $nn] AR := DR
$15 $050000 000001010000000000000000 DR := {AR}
$16 $008020 000000001000000000100000 BX := DR
$17 $000000 000000000000000000000000 ; execute
$18 $088000 000010001000000000000000 [ldb ($nn)] AR := DR
$19 $050000 000001010000000000000000 DR := {AR}
$1a $088000 000010001000000000000000 AR := DR
$1b $050000 000001010000000000000000 DR := {AR}
$1c $008020 000000001000000000100000 BX := DR
$1d $000000 000000000000000000000000 ; execute
$1e $088000 000010001000000000000000 [ldb ($nn),a] AR := DR
$1f $050000 000001010000000000000000 DR := {AR}
$20 $008102 000000001000000100000010 Z := AX + DR
$21 $080001 000010000000000000000001 AR := Z
$22 $050000 000001010000000000000000 DR := {AR}
$23 $008020 000000001000000000100000 BX := DR
$24 $000000 000000000000000000000000 ; execute
$25 $008102 000000001000000100000010 [ldb ($nn,a)] Z := DR + AX
$26 $080001 000010000000000000000001 AR := Z
$27 $050000 000001010000000000000000 DR := {AR}
$28 $088000 000010001000000000000000 AR := DR
$29 $050000 000001010000000000000000 DR := {AR}
$2a $008020 000000001000000000100000 BX := DR
$2b $000000 000000000000000000000000 ; execute
$2c $088000 000010001000000000000000 [sta $nn] AR := DR
$2d $010040 000000010000000001000000 DR := AX
$2e $028000 000000101000000000000000 {AR} := DR
$2f $000000 000000000000000000000000 ; execute
$30 $088000 000010001000000000000000 [sta ($nn)] AR := DR
$31 $050000 000001010000000000000000 DR := {AR}
$32 $088000 000010001000000000000000 AR := DR
$33 $050000 000001010000000000000000 DR := {AR}
$34 $008080 000000001000000010000000 AX := DR
$35 $000000 000000000000000000000000 ; execute
$36 $008102 000000001000000100000010 [stb $nn,a] Z := DR + AX
$37 $080001 000010000000000000000001 AR := Z
$38 $010010 000000010000000000010000 DR := BX
$39 $028000 000000101000000000000000 {AR} := DR
```

```
$3a $000000 000000000000000000000000 ; execute
$3b $088000 000010001000000000000000 [stb $nn] AR := DR
$3c $010010 000000010000000000010000 DR := BX
$3d $028000 000000101000000000000000 {AR} := DR
$3e $000000 000000000000000000000000 ; execute
$3f $088000 000010001000000000000000 [stb ($nn)] AR := DR
$40 $050000 000001010000000000000000 DR := {AR}
$41 $088000 000010001000000000000000 AR := DR
$42 $010010 000000010000000000010000 DR := BX
$43 $028000 000000101000000000000000 {AR} := DR
$44 $000000 000000000000000000000000 ; execute
$45 $088000 000010001000000000000000 [stb ($nn),a] AR := DR
$46 $050000 000001010000000000000000 DR := {AR}
$47 $008102 000000001000000100000010 Z := AX + DR
$48 $080001 000010000000000000000001 AR := Z
$49 $010010 000000010000000000010000 DR := BX
$4a $028000 000000101000000000000000 {AR} := DR
$4b $000000 000000000000000000000000 ; execute
$4c $008102 000000001000000100000010 [stb ($nn,a)] Z := DR + AX
$4d $080001 000010000000000000000001 AR := Z
$4e $050000 000001010000000000000000 DR := {AR}
$4f $088000 000010001000000000000000 AR := DR
$50 $010010 000000010000000000010000 DR := BX
$51 $028000 000000101000000000000000 {AR} := DR
$52 $000000 000000000000000000000000 ; execute
$53 $000060 000000000000000001100000 [tab] BX := AX
$54 $000000 000000000000000000000000 ; execute
$55 $000090 000000000000000010010000 [tba] AX := BX
$56 $000000 000000000000000000000000 ; execute
$57 $008102 000000001000000100000010 [adc #$nn] Z := AX + DR
$58 $000081 000000000000000010000001 AX := Z
$59 $000000 000000000000000000000000 ; execute
$5a $088000 000010001000000000000000 [adc $nn] AR := DR
$5b $050000 000001010000000000000000 DR := {AR}
$5c $008102 000000001000000100000010 Z := AX + DR
$5d $000081 000000000000000010000001 AX := Z
$5e $000000 000000000000000000000000 ; execute
$5f $008202 000000001000001000000010 [sbc #$nn] Z := AX - DR
$60 $000081 000000000000000010000001 AX := Z
$61 $000000 000000000000000000000000 ; execute
$62 $088000 000010001000000000000000 [sbc $nn] AR := DR
$63 $050000 000001010000000000000000 DR := {AR}
$64 $008202 000000001000001000000010 Z := AX - DR
$65 $000081 000000000000000010000001 AX := Z
$66 $000000 000000000000000000000000 ; execute
$67 $001102 000000000001000100000010 [inc] Z := AX + 1
$68 $000081 000000000000000010000001 AX := Z
$69 $000000 000000000000000000000000 ; execute
$6a $001202 000000000001001000000010 [dec] Z := AX - 1
$6b $000081 000000000000000010000001 AX := Z
$6c $000000 000000000000000000000000 ; execute
$6d $008302 000000001000001100000010 [and #$nn] Z = AX AND DR
$6e $000081 000000000000000010000001 AX := Z
$6f $000000 000000000000000000000000 ; execute
$70 $088000 000010001000000000000000 [and $nn] AR := DR
$71 $050000 000001010000000000000000 DR := {AR}
$72 $008302 000000001000001100000010 Z := AX AND DR
$73 $000081 000000000000000010000001 AX := Z
$74 $000000 000000000000000000000000 ; execute
$75 $008402 000000001000010000000010 [ora #$nn] Z = AX OR DR
$76 $000081 000000000000000010000001 AX := Z
$77 $000000 000000000000000000000000 ; execute
$78 $088000 000010001000000000000000 [ora $nn] AR := DR
$79 $050000 000001010000000000000000 DR := {AR}
$7a $008402 000000001000010000000010 Z := AX OR DR
```

```
$7b $000081 000000000000000010000001 AX := Z
$7c $000000 000000000000000000000000 ; execute
$7d $008502 000000001000010100000010 [eor #$nn] Z = AX XOR DR
$7e $000081 000000000000000010000001 AX := Z
$7f $000000 000000000000000000000000 ; execute
$80 $088000 000010001000000000000000 [eor $nn] AR := DR
$81 $050000 000001010000000000000000 DR := {AR}
$82 $008502 000000001000010100000010 Z := AX XOR DR
$83 $000081 000000000000000010000001 AX := Z
$84 $000000 000000000000000000000000 ; execute
$85 $000602 000000000000011000000010 [lsl] Z = AX LSL
$86 $000081 000000000000000010000001 AX := Z
$87 $000000 000000000000000000000000 ; execute
$88 $000702 000000000000011100000010 [lsr] Z = AX LSR
$89 $000081 000000000000000010000001 AX := Z
$8a $000000 000000000000000000000000 ; execute
$8b $000802 000000000000100000000010 [asl] Z = AX ASL
$8c $000081 000000000000000010000001 AX := Z
$8d $000000 000000000000000000000000 ; execute
$8e $000902 000000000000100100000010 [rol] Z = AX ROL
$8f $000081 000000000000000010000001 AX := Z
$90 $000000 000000000000000000000000 ; execute
$91 $000a02 000000000000101000000010 [ror] Z = AX ROR
$92 $000081 000000000000000010000001 AX := Z
$93 $000000 000000000000000000000000 ; execute
$94 $004000 000000000100000000000000 [bpl]
$95 $000000 000000000000000000000000 ; execute
$96 $004100 000000000100000100000000 [bmi]
$97 $000000 000000000000000000000000 ; execute
$98 $004200 000000000100001000000000 [bcc]
$99 $000000 000000000000000000000000 ; execute
$9a $004300 000000000100001100000000 [bcs]
$9b $000000 000000000000000000000000 ; execute
$9c $004400 000000000100010000000000 [bne]
$9d $000000 000000000000000000000000 ; execute
$9e $004500 000000000100010100000000 [beq]
$9f $000000 000000000000000000000000 ; execute
$a0 $004600 000000000100011000000000 [sec]
$a1 $000000 000000000000000000000000 ; execute
$a2 $004700 000000000100011100000000 [clc]
$a3 $000000 000000000000000000000000 ; execute
$a4 $008202 000000001000001000000010 [cmp #$nn] Z := AX - DR
$a5 $000000 000000000000000000000000 ; execute
$a6 $088000 000010001000000000000000 [cmp $nn] AR := DR
$a7 $050000 000001010000000000000000 DR := {AR}
$a8 $008202 000000001000001000000010 Z := AX - DR
$a9 $000000 000000000000000000000000 ; execute
$aa $080004 000010000000000000000100 [pha] AR := SP
$ab $010040 000000010000000001000000 DR := AX
$ac $028000 000000101000000000000000 {AR} := DR
$ad $000084 000000000000000010000100 AX := SP
$ae $001102 000000000001000100000010 Z := AX + 1
$af $000009 000000000000000000001001 SP := Z
$b0 $008080 000000001000000010000000 AX := DR
$b1 $000000 000000000000000000000000 ; execute
$b2 $000084 000000000000000010000100 [pop] AX := SP
$b3 $001202 000000000001001000000010 Z = AX - 1
$b4 $080009 000010000000000000001001 SP := Z & AR := Z
$b5 $050000 000001010000000000000000 DR := {AR}
$b6 $008080 000000001000000010000000 AX := DR
$b7 $000000 000000000000000000000000 ; execute
$b8 $400000 010000000000000000000000 [jmp #$nn] PC := DR
$b9 $000000 000000000000000000000000 ; execute
$ba $088000 000010001000000000000000 [jmp $nn] AR := DR
$bb $050000 000001010000000000000000 DR := {AR}
```

```
$bc $400000 01000000000000000000000000 PC := DR
$bd $000000 00000000000000000000000000 ; execute
$be $008080 00000000100000001000000000 [jsr #$nn] AX := DR ; keep operand
$bf $080004 00001000000000000000000100 AR := SP ; save sp
$c0 $210000 00100001000000000000000000 DR := PC
$c1 $028000 00000010100000000000000000 {AR} := DR
$c2 $010040 00000001000000000001000000 DR := AX ; change pc
$c3 $400000 01000000000000000000000000 PC := DR
$c4 $000084 00000000000000000010000100 AX := SP ; increase sp
$c5 $001102 00000000000010001000000010 Z := AX + 1
$c6 $000009 00000000000000000000001001 SP := Z
$c7 $000000 00000000000000000000000000 ; execute
$c8 $088000 00001000100000000000000000 [jsr $nn] AR := DR ; read jump address
$c9 $050000 00000101000000000000000000 DR := {AR}
$ca $200080 00100000000000000010000000 AX := PC ; store pc
$cb $400000 01000000000000000000000000 PC := DR ; jump to address
$cc $010040 00000001000000000001000000 DR := AX
$cd $080084 00001000000000000010000100 AR := SP & AX := SP
$ce $001102 00000000000010001000000010 Z := AX + 1
$cf $000009 00000000000000000000001001 SP := Z
$d0 $000000 00000000000000000000000000 ; execute
$d1 $000084 00000000000000000010000100 [rts] AX := SP
$d2 $001202 00000000000010010000000010 Z := AX - 1
$d3 $080009 00001000000000000000001001 SP := Z & AR := Z
$d4 $050000 00000101000000000000000000 DR := {AR}
$d5 $400000 01000000000000000000000000 PC := DR
$d6 $000000 00000000000000000000000000 ; execute
$d7 $002000 00000000001000000000000000 [cib] ; clear input buffer
$d8 $000000 00000000000000000000000000 ; execute
$d9 $008102 00000000100000010000000010 [ldb $nn,a] Z := DR + AX
$da $080001 00001000000000000000000001 AR := Z
$db $050000 00000101000000000000000000 DR := {AR}
$dc $008020 00000000100000000010000000 BX := DR
$dd $000000 00000000000000000000000000 ; execute
```

B - Opcodes table

Storage

lda #$nn
lda $nn
lda ($nn)
ldb #$nn
ldb $nn
ldb ($nn)
sta $nn
sta ($nn)
stb $nn
stb $nn,a
stb ($nn),a
stb ($nn,a)
tab transfer from AX to BX
tba transfer from BX to AX

Math

adc #$nn AX = AX + $nn
adc $nn AX = AX + {$nn}
sbc #$nn AX = AX - $nn
sbc $nn AX = AX - {$nn}
inc AX = AX + 1
dec AX = AX - 1

Bitwise

and #$nn AX = AX AND $nn
and $nn AX = AX AND {$nn}
ora #$nn AX = AX OR $nn
ora $nn AX = AX OR {$nn}
eor #$nn AX = AX XOR $nn
eor $nn AX = AX XOR {$nn}
lsl AX logical shift left
lsr AX logical shift right
asl AX arithmetic shift left
rol AX rotate left one bit
ror AX rotate right one bit

Branch

bpl $nn branch on plus (N)
bmi $nn branch on minus (N)
bcc $nn branch on carry clear (C)
bcs $nn branch on carry set (C)
bne $nn branch on not equal (Z)
beq $nn branch on equal (Z)

Flags

sec set carry flag
clc clear carry flag

Registers

cmp #$nn compare $nn to AX
cmp $nn compare {$nn} to AX

Stack

pha push AX on stack
pop pop stack top to AX

Jump

jmp #$nn jump to location $nn
jmp $nn jump to location {$nn}
jsr $nn jump to location $nn and save return address
jsr {$nn} jump to location {$nn} and save return address
rts return from subroutine

Input

cib clear input buffer

C - test.asm

```
; Beispielprogramm für fiktive 8bit CPU

; Initialisierung

lda #$00
sta $c0           ; Spieler Position

jmp draw          ; Zeichne initiale Spieler-Position

; Haupt-Routine

main:
  lda $fe         ; Tastaturbuffer abfragen
  cmp #$02        ; "<"?
  beq left
  cmp #$04        ; ">"?
  beq right
  jmp main

; Routine: Spieler zeichnen

draw:
  lda $c0         ; Lade Spieler-Position in AX
  ldb #$ff        ; Lade Farbe Weiß in BX
  stb $e5,a       ; Zeiche Spieler mit Farbe aus BX an Position AX
  jmp main

; Routine: Spieler löschen

clear:
  lda $c0         ; Lade Spieler-Position in AX
  ldb #$00        ; Lade Farbe Schwarz in BX
  stb $e5,a       ; Schreibe schwarz (leer) an Stelle des Spielers
rts

; Routine: Spieler nach links oder rechts bewegen

left:
  cib             ; Clear input buffer
  jsr clear       ; Lösche letzte Position auf Bildschirm
  lda $c0         ; Lade aktuelle Position in A
  dec             ; .. um Position um 1 zu verringern
  cmp #$ff        ; Haben wir den linken Rand erreicht?
  beq left_border
  sta $c0         ; Speichere neue Position
  jmp draw
left_border:
  lda #$00        ; Setze Position auf maximalen linken Rand
  sta $c0
  jmp draw

right:
  cib             ; Clear input buffer
  jsr clear       ; Lösche letzte Position auf Bildschirm
  lda $c0         ; Lade aktuelle Position in A
  inc             ; .. um Position um 1 zu erhöhen
  cmp #$19        ; haben wir den rechten Rand erreicht?
  beq right_border
  sta $c0         ; Speichere neue Position
  jmp draw
right_border:
```

```
lda #$18        ; Setze Position auf maximalen rechten Rand
sta $c0
jmp draw
```

D - assembler.c

```
/*
 * assembler.c
 * ------
 * Reads a 8bit cpu assembler file and writes a compiled executable
 * for our architecture.
 *
 * Compile project via: clang assembler.c -o assembler
 *
 * Usage: ./assembler program.asm program.out
 */

#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VERSION                 "0.2"

#define PROGRAM_MEMORY_SIZE     256

#define MAX_JUMP_COUNT          64
#define MAX_LABEL_COUNT         32
#define MAX_LABEL_LENGTH        32
#define MAX_LINE_LENGTH         128
#define OPCODE_COUNT            33


/*
 * Struct: label
 * ------
 * Placeholder for occurring label and jump declaration, mapping
 * the address in the program memory to the label name.
 */

struct label
{
    int memory_addr;
    char label[MAX_LABEL_LENGTH];
};


/*
 * Struct: opcode
 * ------
 * Holds data about opcode mnemonic and its microcode addresses
 * for the regarding addressing modes.
 */

typedef struct
{
    char mnemonic[3];   // Mnemonic of opcode

    int a;              // Implicit / single-byte instruction (no operand)

    int a_abs;          // Absolute             $nn
    int a_imm;          // Immediate            #$nn
    int a_idx;          // Indexed              $nn,a
    int a_idx_ind;      // Indexed Indirect     ($nn,a)
    int a_ind;          // Indirect             ($nn)
    int a_ind_idx;      // Indirect Indexed     ($nn),a
```

```
    int a_label;        // Labelled (internal)
} opcode;


/*
 * Table: OPCODES[]
 * ------
 * Lists all given opcodes of our architecture defining
 * their microcode addresses in the control unit (CU).
 */

static const opcode OPCODES[] =
{
    { "adc", 0x00, 0x5a, 0x57, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "and", 0x00, 0x70, 0x6d, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "asl", 0x8b, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "bcc", 0x00, 0x00, 0x98, 0x00, 0x00, 0x00, 0x00, 0x98 },
    { "bcs", 0x00, 0x00, 0x9a, 0x00, 0x00, 0x00, 0x00, 0x9a },
    { "beq", 0x00, 0x00, 0x9e, 0x00, 0x00, 0x00, 0x00, 0x9e },
    { "bmi", 0x00, 0x00, 0x96, 0x00, 0x00, 0x00, 0x00, 0x96 },
    { "bne", 0x00, 0x00, 0x9c, 0x00, 0x00, 0x00, 0x00, 0x9c },
    { "bpl", 0x00, 0x00, 0x94, 0x00, 0x00, 0x00, 0x00, 0x94 },
    { "cib", 0xd7, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "clc", 0xa2, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "cmp", 0x00, 0xa6, 0xa4, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "dec", 0x6a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "eor", 0x00, 0x80, 0x7d, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "inc", 0x67, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "jmp", 0x00, 0xba, 0xb8, 0x00, 0x00, 0x00, 0x00, 0xb8 },
    { "jsr", 0x00, 0xc8, 0xbe, 0x00, 0x00, 0x00, 0x00, 0xbe },
    { "lda", 0x00, 0x08, 0x06, 0x00, 0x00, 0x0c, 0x00, 0x00 },
    { "ldb", 0x00, 0x14, 0x12, 0xd9, 0x25, 0x18, 0x1e, 0x00 },
    { "lsl", 0x85, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "lsr", 0x88, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "ora", 0x00, 0x78, 0x75, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "pha", 0xaa, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "pop", 0xb2, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "rol", 0x8e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "ror", 0x91, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "rts", 0xd1, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "sbc", 0x00, 0x62, 0x5f, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "sec", 0xa0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "sta", 0x00, 0x2c, 0x00, 0x00, 0x00, 0x30, 0x00, 0x00 },
    { "stb", 0x00, 0x3b, 0x00, 0x36, 0x4c, 0x3f, 0x45, 0x00 },
    { "tab", 0x53, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { "tba", 0x55, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
};


/*
 * Function: print_syntax_error
 * ------
 * Prints a formatted error message for the user.
 *
 * message: Error message string
 * token: Related artefact
 * line_index: Line number in code (-1 when not given)
 */

void print_syntax_error (char *message, char *token, int line_index)
{
    if (line_index > -1)
    {
        printf(
```

```
            "Found syntax error in '%s' @ line %i (%s)!\n",
            token,
            line_index,
            message
        );
    }
    else
    {
        printf(
            "Found syntax error in '%s' (%s)!\n",
            token,
            message
        );
    }
}


/*
 * Function: remove_comments
 * ------
 * Removes all assembler-style comments (;) from a string
 *
 * str: The processed string
 */

void remove_comments (char *str)
{
    int i;

    for (i = 0; str[i] != 00; i++)
    {
        if (str[i] == ';')
        {
            str[i] = 00;
            break;
        }
    }
}


/*
 * Function: is_hex_str
 * ------
 * Checks if string holds only symbols describing a hexadecimal number
 *
 * str: The processed string
 *
 * returns: 1 if string represents a hexadecimal number, otherwise 0
 */

bool is_hex_str (char *str)
{
    int i;
    bool result = 1;

    for (i = 0; str[i] != 00; i++)
    {
        if (!isxdigit(str[i]))
        {
            result = 0;
            break;
        }
    }

    return result;
```

```
}


/*
 * Function: is_abs, is_idx, is_imm, is_idx_ind, is_ind, is_ind_idx
 * ------
 * Checks if address string uses an specific addressing mode
 *
 * token: The string to check against
 *
 * returns: 1 if string uses the adressing mode, otherwise 0
 */

bool is_abs (char *token)
{
    if (
        token[0] == '$' &&
        token[strlen(token) - 2] != ','
    )
    {
        return 1;
    }

    return 0;
}


bool is_idx (char *token)
{
    if (
        token[0] == '$' &&
        token[strlen(token) - 2] == ',' &&
        token[strlen(token) - 1] == 'a'
    )
    {
        return 1;
    }

    return 0;
}


bool is_imm (char *token)
{
    if (token[0] == '#')
    {
        return 1;
    }

    return 0;
}


bool is_idx_ind (char *token)
{
    if (
        token[0] == '(' &&
        token[1] == '$' &&
        token[strlen(token) - 3] == ',' &&
        token[strlen(token) - 2] == 'a' &&
        token[strlen(token) - 1] == ')'
    )
    {
        return 1;
    }
```

```
    return 0;
}


bool is_ind (char *token)
{
    if (
        token[0] == '(' &&
        token[1] == '$' &&
        token[strlen(token) - 3] != ',' &&
        token[strlen(token) - 2] != 'a' &&
        token[strlen(token) - 1] == ')'
    )
    {
        return 1;
    }

    return 0;
}


bool is_ind_idx (char *token)
{
    if (
        token[0] == '(' &&
        token[1] == '$' &&
        token[strlen(token) - 3] != ')' &&
        token[strlen(token) - 2] != ',' &&
        token[strlen(token) - 1] == 'a'
    )
    {
        return 1;
    }

    return 0;
}


/*
 * Function: is_valid_address
 * ------
 * Checks if token is a correct address mode
 *
 * opcode_index: Used opcode
 * str: The processed string
 *
 * returns: 1 if string is a valid address, otherwise 0
 */

bool is_valid_address (int opcode_index, char *str)
{
    if (
        (OPCODES[opcode_index].a_abs && is_abs(str)) ||
        (OPCODES[opcode_index].a_ind && is_ind(str)) ||
        (OPCODES[opcode_index].a_imm && is_imm(str)) ||
        (OPCODES[opcode_index].a_ind_idx && is_ind_idx(str)) ||
        (OPCODES[opcode_index].a_idx_ind && is_idx_ind(str)) ||
        (OPCODES[opcode_index].a_idx && is_idx(str))
    )
    {
        return 1;
    }

    return 0;
```

```
}

/*
 * Function: find_opcode_index
 * ------
 * Returns the index of the opcode lookup table
 *
 * mnemonic: The mnemonic of the opcode
 *
 * returns: index of opcode in lookup table
 */

int find_opcode_index (char *mnemonic)
{
    int i;
    int found_index = -1;

    for (i = 0; i < OPCODE_COUNT; i++)
    {
        if (strncmp(OPCODES[i].mnemonic, mnemonic, 3) == 0)
        {
            found_index = i;
            break;
        }
    }

    return found_index;
}


/*
 * Function: handle_opcode
 * ------
 * Adds the single opcode (1byte) microcode instruction to the program.
 *
 * program: The program
 * index: Index of the current line in program
 * op: Index of the opcode in lookup-table
 */

void handle_opcode (int *program, int index, int op)
{
    program[index] = OPCODES[op].a;
    program[index + 1] = 0x00;
}


/*
 * Function: handle_opcode_and_operand
 * ------
 * Adds the opcode + operand (2byte) microcode instruction to the program.
 *
 * program: The program
 * index: Index of the current line in program
 * op: Index of the opcode in lookup-table
 * addr: Operand / address
 */

void handle_opcode_and_operand (int *program, int index, int op, char *addr)
{
    char* stripped_addr;

    if (is_imm(addr))
    {
        // Immediate adressing mode
```

```
        program[index] = OPCODES[op].a_imm;
        stripped_addr = addr + 2;
    }
    else if (is_idx(addr))
    {
        // Indexed adressing mode
        program[index] = OPCODES[op].a_idx;
        stripped_addr = addr + 1;
        stripped_addr[strlen(stripped_addr) - 2] = 0;
    }
    else if (is_abs(addr))
    {
        // Absolute adressing mode
        program[index] = OPCODES[op].a_abs;
        stripped_addr = addr + 1;
    }
    else if (is_ind(addr))
    {
        // Indirect adressing mode
        program[index] = OPCODES[op].a_ind;
        stripped_addr = addr + 2;
        stripped_addr[strlen(stripped_addr) - 1] = 0;
    }
    else if (is_ind_idx(addr))
    {
        // Indirect indexed adressing mode
        program[index] = OPCODES[op].a_ind_idx;
        stripped_addr = addr + 2;
        stripped_addr[strlen(stripped_addr) - 3] = 0;
    }
    else if (is_idx_ind(addr))
    {
        // Indexed indirect adressing mode
        program[index] = OPCODES[op].a_idx_ind;
        stripped_addr = addr + 2;
        stripped_addr[strlen(stripped_addr) - 3] = 0;
    }

    if (!is_hex_str(stripped_addr))
    {
        print_syntax_error("Invalid address format", addr, 0);
        exit(EXIT_FAILURE);
    }

    int hex_int = (int) strtol(stripped_addr, NULL, 16);

    if (hex_int < 0 || hex_int > PROGRAM_MEMORY_SIZE - 1)
    {
        print_syntax_error("Invalid address range", addr, 0);
        exit(EXIT_FAILURE);
    }

    program[index + 1] = hex_int;
}


/*
 * Function: print_and_save_program
 * ------
 * Writes the compiled program to a file and
 * prints the result to the screen.
 *
 * file: Write file
 * program: The compiled program
 * size: Length of the program (in bytes)
```

```
 */

void print_and_save_program (FILE *file, int *program, int size)
{
    int i;

    // Header for Logisim
    fprintf(file, "v2.0 raw\n");

    for (i = 0; i < size; i++)
    {
        fprintf(file, "%02x", program[i]);
        printf("%02x", program[i]);

        // Pretty print for screen
        if (i % 16 == 15)
        {
            fprintf(file, "\n");
            printf("\n");
        }
        else
        {
            fprintf(file, " ");
            printf(" ");
        }
    }
}


/*
 * Function: main
 * ------
 * Reads a 8bit cpu assembler file and writes a compiled executable
 * for our architecture.
 *
 * Usage: ./assembler program.asm program.out
 */

int main (int argc, char **argv)
{
    printf("=============================================\n");
    printf("            8bit cpu assembler v%s\n", VERSION);
    printf("=============================================\n");

    // Get input file
    char *file_path = argv[1];
    FILE *file;

    file = fopen(file_path, "r");

    if (file == 0)
    {
        printf("Error: Please specify a valid file path.\n");
        exit(EXIT_FAILURE);
    }

    // Check output file
    char *write_file_path = argv[2];

    if (!write_file_path)
    {
        printf("Error: Please specify a valid out file path.\n");
        exit(EXIT_FAILURE);
    }
```

```
// Variables for stream reading
char line[MAX_LINE_LENGTH];
char *token = NULL;

// Variables for parsing and error output
int opcode_index = -1;
int current_line_index = 1;

// The compiled program variables
int program[PROGRAM_MEMORY_SIZE];
int program_adr = 0;

// Lookup table for labels
struct label label_table[MAX_LABEL_COUNT];
int label_table_count = 0;

// Lookup table for labelled jump instructions
struct label jump_table[MAX_JUMP_COUNT];
int jump_table_count = 0;

// Read file line by line
while (fgets(line, sizeof line, file) != NULL)
{
    remove_comments(line);
    token = strtok(line, "\n\t\r ");

    while (token)
    {
        // Expects operand (address or label) of two-byte instruction
        if (opcode_index > -1)
        {
            if (is_valid_address(opcode_index, token))
            {
                // Found an address
                handle_opcode_and_operand(
                    program,
                    program_adr,
                    opcode_index,
                    token
                );
            }
            else if (OPCODES[opcode_index].a_label)
            {
                // Found a label
                if (jump_table_count > MAX_JUMP_COUNT)
                {
                    print_syntax_error(
                        "Exceeded jump count",
                        token,
                        current_line_index
                    );

                    exit(EXIT_FAILURE);
                }
                else if (strlen(token) > MAX_LABEL_LENGTH)
                {
                    print_syntax_error(
                        "Label name is too long",
                        token,
                        current_line_index
                    );

                    exit(EXIT_FAILURE);
                }
```

```
            // Write instruction to program with placeholder
            program[program_adr] = OPCODES[opcode_index].a_label;
            program[program_adr + 1] = 0x00;

            // Store position for final processing
            jump_table[jump_table_count].memory_addr = program_adr + 1;
            strcpy(jump_table[jump_table_count].label, token);
            jump_table_count += 1;
        }
        else
        {
            print_syntax_error(
                "Invalid or missing operand",
                token,
                current_line_index
            );

            exit(EXIT_FAILURE);
        }

        opcode_index = -1;
        program_adr += 2;
    }
    else
    {
        // Handle label or opcode
        int op = find_opcode_index(token);

        // Is token an opcode?
        if (op > -1)
        {
            // Will an operand be expected in next token?
            if (!OPCODES[op].a)
            {
                // Prepare two-byte instruction
                opcode_index = op;
            }
            else
            {
                // Handle single byte instruction
                handle_opcode(
                    program,
                    program_adr,
                    op
                );

                program_adr += 2;
            }
        }
        else if (token[(strlen(token) - 1)] == ':')
        {
            // Token is a label, strip the last character (:)
            token[strlen(token) - 1] = 0;

            if (label_table_count > MAX_LABEL_COUNT)
            {
                print_syntax_error(
                    "Exceeded label count",
                    token,
                    current_line_index
                );

                exit(EXIT_FAILURE);
            }
            else if (strlen(token) > MAX_LABEL_LENGTH)
```

```
                {
                    print_syntax_error(
                        "Label name is too long",
                        token,
                        current_line_index
                    );

                    exit(EXIT_FAILURE);
                }

                // Store it in table for later processing
                label_table[label_table_count].memory_addr = program_adr;
                strcpy(label_table[label_table_count].label, token);

                label_table_count += 1;
            }
            else
            {
                print_syntax_error(
                    "Unkown opcode",
                    token,
                    current_line_index
                );

                exit(EXIT_FAILURE);
            }
        }

        if (program_adr > PROGRAM_MEMORY_SIZE)
        {
            print_syntax_error(
                "Program exceeds memory size",
                token,
                current_line_index
            );

            exit(EXIT_FAILURE);
        }

        token = strtok(NULL, "\n\t\r ");
    }

    current_line_index += 1;
}

fclose(file);

// Finally replace labels with memory addresses
int i, n;
bool found_label;

for (i = 0; i < jump_table_count; i++)
{
    found_label = 0;

    for (n = 0; n < label_table_count; n++)
    {
        if (strcmp(jump_table[i].label, label_table[n].label) == 0)
        {
            // Replace label with actual memory location in program
            program[jump_table[i].memory_addr] = label_table[n].memory_addr;
            found_label = 1;
            break;
        }
    }
```

```
        if (!found_label)
        {
            print_syntax_error("Could not find label", jump_table[i].label, -1);
            exit(EXIT_FAILURE);
        }
    }

    // Save binary to file and print result
    printf(
        "Successfully compiled program (%i bytes):\n\n",
        program_adr
    );

    FILE * write_file = fopen(write_file_path, "w+");
    print_and_save_program(write_file, program, program_adr);
    fclose(write_file);

    printf("\n\nWrite output to '%s'.\n", write_file_path);

    return 0;
}
```

E - snake.asm

```
; snake clone for my 8bit cpu

;
;
;
;
;
;
;

; variables

; the following variables describe the
; snakes position on our 5x5 pixel screen
;
; note: the screen is organized as follows:
;
; e5 e6 e7 e8 e9
; ea eb ec ed ee
; ef f0 f1 f2 f3
; f4 f5 f6 f7 f8
; f9 fa fb fc fd
;
; the segment positions of the snake are
; pointers to the screen positions.
;
; prepare the following variables:

    lda #$eb            ; position
    sta $d0             ; ... of snake head

    lda #$ec            ; first segment
    sta $d1             ; ... of snake body

    lda #$ed            ; second segment
    sta $d2             ; ... of snake body
```

```
    lda #$02            ; length
    sta $de             ; ... of snake body

; note: the machine only gives 224 byte of free
; memory. We need 2 bytes for variables ($de, $df)
; plus a maximum of 14 bytes for the snake head
; & body ($d0-$dd) which restricts the whole
; program size to 208 bytes ($00-$cf)!

; initial routines

    jsr apple           ; generate a random apple position
    jsr draw_apple      ; ... & draw the apple

; main loop routine

; note: the program jumps directly to game
; over after executing the main routine for
; the second time, if there is no moving-
; direction defined. To avoid this we do not
; start the game when the user didn't press
; any arrow keys yet.

main:
    lda $fe             ; check the pressed key
    cmp #$00            ; ... when nothing was pressed yet
    beq main            ; ... then do nothing!

    jsr move
    jsr collision
    jsr update
    jsr draw_snake
    jsr draw_apple

    jmp main

; routine: pass over body segments to next position ahead

update:
    lda $de             ; get snake length
    dec
update_loop:
    ldb $d0,a           ; get body segment
    stb $d1,a           ; ... store it one address after
    dec
    bpl update_loop
rts

; routine: check pressed keys & check if the move is maybe illegal

; note: we do not check for wall collisions
; to let the snake run infinitely. Also when leaving
; right or left the snake appears on a different
; line on the other side than before. Both decisions
; are mainly design restrictions due to the
; small memory size.

move:
    lda $fe             ; get currently pressed key

    cmp #$01            ; [up] pressed?
    beq move_up
    cmp #$02            ; [left] pressed?
    beq move_left
    cmp #$03            ; [down] pressed?
```

```
    beq move_down
    cmp #$04            ; [right] pressed?
    beq move_right
rts

move_up:
    lda $d0             ; load snake head position
    sbc #$05            ; ... substract a whole screen line
    sta $d0             ; ... store it again

    cmp #$e5            ; did we reach the top?
    bcc move_up_end     ; ... then ...
rts

move_up_end:
    adc #$19            ; ... come up from the bottom again
    sta $d0
rts

move_left:
    lda $d0             ; load snake head position
    dec                 ; ... decrement it by one
    sta $d0             ; ... store it again

    cmp #$e4            ; did we reach the up left border?
    beq move_left_end   ; ... then
rts

move_left_end:
    lda #$e9            ; ... come out on the right corner
    sta $d0
rts

move_down:
    lda $d0             ; load snake head position
    adc #$05            ; ... add a whole screen line
    sta $d0             ; ... store it again

    cmp #$fe            ; did we reach the border?
    bcs move_down_endh

    cmp #$04            ; we also have to check for this since
    bcc move_down_endl  ; the screen memory is (weirdly) organized
rts

move_down_endh:
    sbc #$19            ; come up from the top again
    sta $d0
rts

move_down_endl:
    adc #$e7            ; ... same here
    sta $d0
rts

move_right:
    lda $d0             ; load snake head position
    inc                 ; ... increment it by one
    sta $d0             ; ... store it again

    cmp #$fe            ; did we reach the bottom right?
    beq move_right_end  ; ... then
rts

move_right_end:
```

```
    lda #$f9            ; ... come out on the left corner
    sta $d0
rts

; routine: check if snake collided with apple or itself

; note: we check first if the snake's head sits
; on the same address as the apple. if not
; then any set pixel must be a part of the snake.

collision:
    lda $df             ; load apple position
    cmp $d0             ; ... check if its the heads position
    bne collision_snake

    lda $de             ; increase snake length
    inc                 ; ... by one
    sta $de             ; ... & store it again

    jsr apple           ; find new apple position
rts

collision_snake:
    lda ($d0)           ; load pixel at snake head
    cmp #$ff            ; check if its set
    beq game_over
rts

; routine: place an apple randomly on the screen

apple:
    lda $ff             ; load a random number into AX
    and #$18            ; limit range to 0-24
    adc #$e5            ; add start screen address
    sta $df             ; ... & store it as the new apple position
rts

; routine: draw snake on screen

draw_snake:
    lda $de             ; get snake length
    ldb #$00            ; prepare clear pixel
    stb ($d0,a)

    ldb #$ff            ; prepare set pixel for display
    stb ($d0)           ; draw head at screen position
rts

draw_apple:
    ldb #$ff            ; pixel to set
    stb ($df)           ; ... & store at screen location
rts

; routine: game over

game_over:
    jmp game_over       ; ... is an endless loop
```